

Hochschule der Medien Stuttgart
Fakultät 1, Studiengang Medieninformatik



Flow Design

Konzeption und Implementierung einer WPF-Anwendung zur grafischen
Modellierung von Flow-Design-Entwürfen sowie Generierung von
C#-Code unter Verwendung von Microsoft Roslyn



Dennis Müller
Matrikelnummer 25675
10. Fachsemester
dennis.briefkasten@gmail.com

Erstprüfer: Professor Walter Kriha
Zweitprüfer: Kevin Erath

24. Januar 2017

Ehrenwörtliche Erklärung

Hiermit versichere ich, Dennis Müller, ehrenwörtlich, dass ich die vorliegende Bachelorarbeit mit dem Titel:

Konzeption und Implementierung einer WPF-Anwendung zur grafischen Modellierung von Flow-Design-Entwürfen sowie Generierung von C#-Code unter Verwendung von Microsoft Roslyn

selbstständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden.

Ich habe die Bedeutung der ehrenwörtlichen Versicherung und die prüfungsrechtlichen Folgen (§ 24 Abs. 2 Bachelor-SPO der HdM) einer unrichtigen oder unvollständigen ehrenwörtlichen Versicherung zur Kenntnis genommen.

Datum

Unterschrift

Abstract

Der erste Teil dieser Arbeit vermittelt dem Leser ein Grundwissen über Flow Design. Eine Entwurfsmethodik die Softwareentwickler helfen soll saubereren Code zu schreiben.

Der Hauptfokus wurde auf das Flow Design-Diagramm gelegt, eine auf Datenfluss fokussierte Entwurfsmethodik. Die Notation wird erläutert und die Implementierung des Datenflusses in C# anhand von Beispielen dem Leser nähergebracht. Es werden Regeln und Prinzipien vorgestellt, die bei der Umsetzung des Datenflusses einzuhalten sind, um Abhängigkeiten im Code zu reduzieren.

Der zweite Teil dieser Arbeit beschäftigt sich mit der Erstellung eines Prototypen eines Editors für Flow Design. Ein Editor mit dem es möglich ist Datenflüsse in Form von Flow Design zu modellieren. Dieser Prototyp bietet darüber hinaus die Möglichkeit C#-Code aus diesen Flussdiagrammen zu generieren. Zuerst wird eine Systemanalyse durchgeführt mit einem speziellen Fokus auf eine gute Usability. Danach wird der aktuelle Stand des Prototypen vorgestellt, sowie ein Einblick in die Architektur gewährt. Dabei werden auch einige ausgewählte Codeauschnitte präsentiert, die ebenfalls nach den Regeln und Prinzipien implementiert wurden, die im Grundlagenkapitel vorgestellt wurden. Diese Codeauschnitte sind komplexer als die Beispiele aus dem Grundlagenkapitel und sollen dem Leser einen Einblick gewähren, wie ein Code nach Flow Design in der Praxis aussehen kann. Damit wird dem Leser eine Möglichkeit geboten, Flow Design besser für sich bewerten zu können.

Als Arbeitstitel für den Prototypen wurde der Projektname „Dexel“ gewählt.

Dexel ist auf Github als Open-Source-Projekt unter folgender URL einsehbar:

<https://github.com/detachmode/Dexel>

Der Zustand von Dexel zum Zeitpunkt dieser Arbeit ist eine Anwendung, die für den Einsatz in einem Projekt noch nicht ausgereift genug ist. Es wurde jedoch ein Grundstein gelegt, auf dem aufgebaut werden kann. Die Anwendung hat einen Punkt erreicht hat, an dem die Richtung des Projektes deutlich wird und eine Vision wie ein Editor für Flow Design aussehen kann, vermittelt wird.

Danksagung

Danke an Kevin Erath für die großartige Unterstützung. Ich danke dir für die vielen Stunden, die du Dir Zeit genommen hast mir Flow Design zu erklären. Auch die Ratschläge wie ich diese Abschlussarbeit in Angriff nehmen konnte, waren sehr hilfreich und trugen maßgeblich dazu bei, dass sich die Arbeit in dem Zustand befindet, wie sie hier vorliegt.

Danke an Professor Walter Kriha für die Unterstützung und das Interesse an der Thematik.

Ich danke Kevin Erath, Professor Walter Kriha, Patricia Maier und Büsra Yagbasan für das Korrekturlesen während der Erstellung dieser Arbeit.

Danke an die IT-Designer Gruppe für die finanzielle Unterstützung und dafür, dass ich mich voll auf die Umsetzung meiner Bachelorarbeit konzentrieren konnte und mir das nötige Arbeitsumfeld geboten hat.

Inhaltsverzeichnis

1. Einführung	7
1.1. Motivation	7
1.2. Aufbau	7
I. Grundlagen	9
2. Entstehung und Grundgedanken	10
2.1. CCD Prinzipien	11
2.2. Weitere Prinzipien die Beachtung finden sollen	13
2.3. Flow Design - Was ist das?	14
3. Pfeile und Kreise	16
3.1. RomanNumbers Beispiel	16
4. Die Notation	17
4.1. Datenströme	17
4.2. Hierarchische Datenflüsse	18
4.3. Definition eigener Datentypen	18
4.4. Arrays	19
4.5. 0 bis n (Datenstrom)	19
4.6. Container / Listen	20
4.7. 0 bis 1 (optionaler Output)	20
4.8. Mehrere Datentypen auf einem Datenstrom	21
4.9. Joined Inputs und Pipe-Notation	21
4.10. Tonnen	22
4.11. Abhängigkeiten / Provider	23
4.12. GUIs / Programmstart/ Ende	23
4.13. Klassen / Container definieren	24
5. Implementation	25
5.1. IODA Architektur	26
5.2. Beispiel foreach und Funktionsaufruf	29
5.3. C# Features um Datenflüsse zu implementieren	33
5.4. Datenströme mit mehreren Wegen	39
5.5. Auf Rückgabewert warten	42
5.6. Nutzen von IOSP	42
5.7. Ausnahmen	43
6. Die Entwurfsmethode	45
6.1. System-Umwelt-Analyse	45
6.2. Interfaceskizze (im Falle einer GUI Anwendung)	46

6.3. Flow Design Entwurf	47
6.4. Rekursive Eigenschaft	47
II. Dexel	48
7. Vision	49
7.1. Vorteile eines digitalen Editors	49
7.2. Vorteile von einem Entwurf auf dem Papier	50
8. Anforderungen	51
8.1. Editor	51
8.2. Generierung von Code	53
8.3. Generierung von Flow Design Diagrammen aus Code	54
9. GUI Skizzen / Usabilityüberlegungen	56
9.1. Minimalistischer Aufbau. Fokus auf Produktivität.	56
9.2. Textfelder	58
9.3. Datentypen - Definition und Organisation	59
9.4. Darstellung von Joined Inputs & Split Outputs	60
9.5. Validierung des Datenflusses	61
9.6. Validierung der Syntax	61
10. Realisierung	62
10.1. Übersicht über die unterschiedlichen Projekte	62
10.2. Das Domänenmodell	64
10.3. Der Editor	69
10.3.1. Vorstellung was erreicht wurde	69
10.3.2. Views / ViewModels	74
10.3.3. Interaktionen	74
10.3.4. Validierung des Datenflusses - Farbliche Kennzeichnung	76
10.4. Roslyn - Generierung von Code aus einem Diagramm	77
10.4.1. Vorstellung - was erreicht wurde	77
10.4.2. Kleiner Einblick in die API von Roslyn	86
10.4.3. Erzeugung von Methodensignaturen	88
10.4.4. Erzeugung des Methodenrumpfes einer Integration	91
10.5. Generierung eines Diagrammes aus Code	96
11. Zusammenfassung	97
11.1. Ausblick	97
11.2. Fazit	98
Quellenverzeichnis	101
Verzeichnis der Listings	102
Abbildungsverzeichnis	103
Tabellenverzeichnis	105

1. Einführung

1.1. Motivation

Software wird häufig einfach nur herunter-programmiert. Dadurch entsteht umgangssprachlich bezeichnet „Spaghetticode“ mit vielen Abhängigkeiten innerhalb der eigenen Codebasis. Solch eine Codebasis ist schwer zu warten und auf Änderungen anzupassen. Die Alternative dazu wäre, vor dem Programmieren einen Entwurf der Architektur zu erstellen und sich vor der Umsetzung Gedanken zu der Struktur zu machen.

Viele der vorhanden Entwurfsmethodiken sind jedoch zu schwergewichtig und helfen einem nicht zwingend dabei Abhängigkeiten zu reduzieren. Zusätzlich zu Entwurfsmethodiken existieren auch Entwurfsprinzipien, die einem Helfen sollen, besser wartbare Software zu schreiben. Viele der Prinzipien sind jedoch oft nur schwer auf die eigene Codebasis anzuwenden, da sie zu abstrakt sind. Dadurch finden sie in der Praxis seltener Anwendung, als nötig wäre, um sauberen Code zu schreiben.

Hier möchte Flow Design Abhilfe schaffen, indem es eine leichtgewichtige Entwurfsmethodik bietet, mit einem speziellen Augenmerk darauf Abhängigkeiten zu reduzieren. Nebenbei hilft Flow Design einem auch viele der gängigen Entwurfsprinzipien einzuhalten.

Leider ist diese Methodik nicht weit verbreitet und das Entwerfen ist bisher auf dem Papier angedacht. Aus diesem Grund existieren keine ausgereiften Tools oder Hilfsprogramme die speziell auf die Erstellung von Flow Design Entwürfen ausgelegt sind.

Ziel dieser Arbeit ist es eine Anwendung zur Erstellung von Flow Design Diagrammen zu entwerfen und einen Prototypen zu implementieren. Diese Anwendung soll darüber hinaus auch Funktionen zur Generierung von Quellcode aus den erstellten Diagrammen bieten, damit der Einsatz von Flow Design in einem Projekt komfortabler und produktiver wird.

Unabhängig davon, ob dieses Ziel erreicht wurde oder nicht sollen die gewonnenen Erkenntnisse aus dem Versuch hier dokumentiert und ein Fazit daraus gezogen werden.

Der Prototyp soll in C# und in Teilen unter Verwendung von Flow Design selbst umgesetzt und auf Github als Open-Source-Projekt veröffentlicht werden.

1.2. Aufbau

Der erste Teil dieser Arbeit stellt ein Grundlagenkapitel dar, dass dem Leser die Methodik Flow Design näher bringen soll. Der Leser bekommt die Entwurfsmethodik Flow

Design anschaulich erklärt, dabei wird auch auf die Herkunft, den Grundgedanken dieser Methodik, sowie ihre Vor- und Nachteile eingegangen. Dazu gehören auch, dass einige neue Prinzipien und Abkürzungen erklärt werden. Des weiteren wird auf die für Flow Design speziellen, dazugehörigen Implementierungsregeln eingegangen und anhand von einfachen Codebeispielen in C# dem Leser näher gebracht. Bei dieser Gelegenheit werden in diesem Teil auch auf einige Sprachfeatures von C# eingegangen, welche das Arbeiten mit Datenströmen stark erleichtern. Auch hier werden einfache Codebeispiel zum besseren Verständnis dem Leser präsentiert. Im genauen handelt es sich hierbei um Lambdas und die Methodenbibliothek LINQ.

Hat man einmal den Vorteil von Flow Design für sich entdeckt liegt es nahe als Entwickler sich Gedanken darüber zu machen wie eine Anwendung aussehen könnte, das einem bei der Verwendung der Methodik so gut es geht unterstützt. Da es solch eine Anwendung noch nicht gibt, geht es in diesem Teil der Arbeit darum die Ansprüche einer solchen im Detail herauszufinden und sie in Form von Anforderungen aufzulisten. Anschließend werden GUI-Skizzen und einige Gedanken zu der Usability der Anwendung vorgestellt.

Am Ende sollen ausgewählte Teile des Codes dokumentiert und das Ergebnis vorgestellt werden. Außerdem wird ein Ausblick auf die Zukunft des Projektes gegeben und ein entsprechendes Fazit gezogen.

Teil I.

Grundlagen

2. Entstehung und Grundgedanken

Flow Design ist aus der Clean Code Development (CCD) Bewegung heraus entstanden. Hauptinitiator und Erfinder ist Ralf Westphal, welcher auch Mitbegründer und Erfinder der CCD Bewegung ist.

Clean Code Development ist eine Ansammlung an Prinzipien, die einem helfen sollen sauber programmierte Software zu schreiben. Viele der Prinzipien sind aus dem Buch „Clean Code“ von Robert C. Martin entnommen. Sauber programmierte Software hat vor allem die Eigenschaft, dass Änderungen und Erweiterungen an ihrer Funktionalität leicht zu realisieren sind. Laut Robert C. Martin [CLNCD] gibt es in der Softwareentwicklung keine Garantie dafür, dass sich für die Software relevanten Rahmenbedingungen nicht jederzeit plötzlich ändern können. Sauber programmierte Software kann leichter auf solche Änderungen angepasst werden.

Es bedarf zwar anfänglich mehr Zeit ein Feature zu implementieren, dass den CCD Prinzipien entspricht, auf lange Sicht steigt jedoch dieser Zeitaufwand nicht mit jedem neuen Feature stark an. Im Vergleich dazu steigt bei einer unsauber programmierten Software der Zeitaufwand ein neues Feature zu implementieren mit der Anzahl an bereits implementierten Features exponentiell an¹. Irgendwann erreicht der Quellcode der Software dann einen Punkt, an dem der Zeitaufwand ein neues Feature zu implementieren derart groß geworden ist, dass es besser ist die Software neu zu schreiben, anstatt sie anzupassen.

CCD bezeichnet eine saubere programmierte Software aus diesem Grund auch als *evolvierbar*. Software die mit dem Fokus auf Evolvierbarkeit programmiert wurde, kann leicht an neue Rahmenbedingungen oder Kundenwünsche angepasst werden.

Eine weitere Eigenschaft von sauber programmierten Software ist, dass sie gut zu lesen ist und möglichst ohne Kommentare verstanden werden kann. Programmcode wird öfters gelesen als geschrieben. Aus diesem Grund ist gute Lesbarkeit eine wichtige Eigenschaft. Gute Lesbarkeit ist auch dann von großer Bedeutsamkeit, wenn eine andere Person den Programmcode nachvollziehen muss, als diejenige Person, die ihn geschrieben hat. Somit ist in größeren Softwareprojekten eine saubere Codebasis umso unverzichtbarer.

Entwurfsprinzipien haben jedoch die Eigenheit, dass sie nicht so leicht einzuhalten sind wie konkrete Regeln. Somit ist es in der Praxis schwer die Prinzipien auf den eigenen Code anzuwenden. Hierbei soll Flow Design einen Lösungsansatz bieten. Flow Design bietet in Ergänzung zu den CCD Prinzipien eine Entwurfsmethodik und dazugehörige Implementierungsregeln, die einfach zu befolgen sind und man erhält fast automatisch einen Code, der ein Großteil der CCD Prinzipien erfüllt.

¹EVOLVIERBARKEIT.

2.1. CCD Prinzipien

Flow Design legt den Schwerpunkt vor allem auf folgende Prinzipien von CCD²

KISS (Keep It Stupid Simple)	Ein System sollte so einfach wie möglich gestaltet werden.
YAGNI (You Ain't Gonna Need It)	Es soll nicht unnötig viel Zeit damit verbracht werden für zukünftige Eventualitäten zu programmieren. Der Fokus sollte darauf liegen, was aktuell wirklich benötigt wird.
Lose Koppelung	Separate Einheiten eines Systems sollen über nur möglichst wenige Punkte miteinander kommunizieren.
Orthogonalität	Änderungen an einer Funktion des Systems sollen auf so wenig wie möglich andere Funktionen des Systems negativen Einfluss haben.

Tabelle 2.1.: CCD Prinzipien

KISS

Die Komplexität der Lösung eines Problems soll immer in Relation zu der Komplexität des Problems stehen.

Schnell passiert es, dass man die einfachste Lösung für ein Problem übersieht und das Problem unnötig verkompliziert. Das KISS Prinzip soll in erster Linie ein Bewusstsein dafür schaffen bei komplizierten Lösungen innezuhalten und sich nochmal genau zu überlegen, ob es nicht eine einfachere Lösung gibt. Manche Problemdomänen erfordern jedoch eine komplexe Lösung, da das Problem komplex ist.

Verkomplizierte Lösungen müssen am besten schon beim Entwurf erkannt werden. Hierbei soll Flow Design als Entwurfsmethode helfen, die einfachste und leicht verständlichste Lösung für ein Problem zu finden.

YAGNI

Vielen Softwareentwicklern ist es wohl schon passiert, dass sie es zu gut gemeint haben mit dem vorausschauendem Denken. Etliche Funktionen wurden für ein zukünftiges Szenario implementiert oder verkompliziert, die jedoch nie eintrafen, oder falls sie eintrafen kam es anders als gedacht, oder die Software wurde bereits durch eine andere ersetzt. YAGNI soll einem das Bewusstsein dafür schärfen, wann man sich gerade mit einer Situation beschäftigt, die die aktuellen Rahmenbedingungen überschreitet. Man

²ROTERGRAD.

sollte sich eher auf das aktuelle Szenario beschränken und keine unnötige Ressourcen für zukünftige Eventualitäten verschwenden. Flow Design bietet hierfür Implementierungsregeln, die es einem ermöglichen schnell Änderungen am Code zu realisieren. Diese Eigenschaft des Codes bietet damit die Basis diesem Prinzip auch getrost zu folgen und sich auf die aktuellen Rahmenbedingungen zu konzentrieren.

Lose Koppelung

Ein System soll aus möglichst voneinander unabhängigen Teilsystemen bestehen, die nur über eine klar definierte Stelle miteinander kommunizieren. Bei jedem Aufruf einer Funktion oder Abrufen einer Variable entsteht eine Koppelung zwischen beiden. Ändert sich die Struktur des Codes, so muss jede Stelle angepasst werden, die zu der geänderten Struktur eine Koppelung besitzt. Mit loser Koppelung möchte man veranschaulichen, dass wenn eine Koppelung nötig ist, diese an einer Stelle konzentriert sein soll und sich nicht an verschiedenen Stellen des Codes fortpflanzen soll. Flow Design fördert das Entkoppeln und zeigt Abhängigkeiten bereits bei der Modellierung.

Orthogonalität

In einem dreidimensionalen Raum sind die 3 Koordinatenachsen üblicherweise zueinander orthogonal. Beim Verschieben eines Objektes auf einer Achse bleiben die Werte der beiden anderen unverändert. Wäre eine Achse nicht orthogonal zu den beiden anderen, so würde eine Verschiebung entlang dieser Achse auch eine Änderung der Werte einer anderen Achse bewirken. Diese Eigenschaft wird nun auch auf Code und wie er auf Änderungen reagiert, projiziert. Wird an einer Stelle der Code geändert, soll diese Änderung möglichst keinen Einfluss auf andere Teile des Codes haben. Flow Design fördert eine starke Entkoppelung der einzelnen Funktionseinheiten, dadurch wird auch das Prinzip der Orthogonalität erfüllt.

2.2. Weitere Prinzipien die Beachtung finden sollen

DRY (Don't Repeat Yourself) [CLNCD, S. 80 und S. 342]	Coderedundanzen vermeiden, zerlegen in Codebestandteile, die man an mehreren Stellen wiederverwenden kann.
Kleine Funktionen / Methoden [CLNCD, S. 64]	Viele kleine Methoden sind besser als eine große.
Single Responsibility Principle [CLNCD, S. 176 f.]	Jede Funktion/Klasse soll sich nur um eine Sache kümmern. Falls eine Funktion mehrere Aufgaben erledigt, sollten sie diese nicht selbst implementieren, sondern an Unterfunktionen weitergeben.
Information Hiding Principle [SCHMZL, S. 48 f.]	Ein Untersystem soll seiner Inneren Funktionalität vor anderen Systemen verbergen und eine möglichst fokussierte Schnittstelle bieten, mit dem äußere Systeme dieses System steuern können.

Tabelle 2.2.: Weitere Prinzipien

DRY

Einer der wichtigsten Aspekte von sauberen Codebasen. Der Grund warum es überhaupt Programmstrukturen wie Funktionen, Methoden, Klassen etc. gibt. Durch Coderedundanzen (Copy-Paste) können schnell Fehler entstehen, der Code wird unverständlicher und durch die Wiederholungen schwer lesbar. Wenn man das DRY Prinzip befolgt, können viele Änderungen bereits an einer Stelle gezielt geändert werden, anstatt die Änderung an vielen Stellen machen zu müssen.

Kleine Funktionen / Methoden

Robert C. Martin schlägt vor Methoden so klein wie nur möglich zu gestalten und sobald sie länger werden den Code in kleinere Methoden auszulagern.

Nachteile:

- In den meisten Fällen fordert das Einhalten dieser Regel eine bessere Verständlichkeit des Codes. Es gibt jedoch auch Fälle, bei denen der abstraktere Namen der übergeordneten Methoden die Verständlichkeit nicht fordert und eine geringere Verschachtlung die eigentliche Funktionalität besser ausdrückt und den Programmverlauf leichter überschaubar macht.
- Das Finden von aussagekräftigen Namen erschwert sich zunehmend.

- In bestimmten Szenarien ist der Overhead eines Methodenaufrufs möglicherweise ein nicht zu verachtender Performanceaspekt (Remote Procedure Calls).

Vorteile:

- Erspart Kommentare durch aussagekräftige Methodennamen.
- Änderungen sind leichter zu realisieren, da durch kleine Methoden auch die höhere Wiederverwendbarkeit einzelner Methoden gegeben ist. Durch weniger Redundanzen kann man eine Änderung meistens gezielt an einer Stelle machen anstatt an vielen Stellen etwas ändern zu müssen.
- Erlaubt ein Denken auf höherer Abstraktionsebene, da low-Level Implementierungsdetails hinter aussagekräftigen Methodennamen verborgen sind.
- Erlaubt anderen Personen den Code leichter zu verstehen und können selbst leichter Änderungen an der Codebasis realisieren, da sie nicht den kompletten Code nachvollziehen müssen, sondern direkt zu den für sie relevanten Stellen springen können.
- Automatische Test / Unittest sind besser realisierbar, da man feingranularer testen kann.

Single Responsibility Principle

Ein Grundgedanke des OO-Designs besteht darin, die Funktionalität der Software auf mehrere Klassen zu verteilen. Jede dieser Klassen besteht aus Methoden, die thematisch zueinander gehören. Das Single Responsibility Principle besagt, dass eine Klasse nur eine Verantwortlichkeit (oder Aufgabenbereich) haben darf. Oft ist mit Single Responsibility Principle auch das Trennen von GUI, Daten und Businesslogik gemeint.

Information Hiding Principle

Eine Klasse besteht aus vielen Funktionen, diese werden jedoch nicht alle nach außen zur Verfügung gestellt. Oder eine API, die nach außen nur eine ganz bestimmte Schnittstelle bietet und die Komplexität des Systems im Inneren verbirgt. Informationen von der Außenwelt zu verstecken ist eine Kernidee der objektorientierten Programmierung und spielt auch bei Flow Design im Prinzip der gegenseitigen Nichtbeachtung eine wichtige Rolle (siehe Kapitel 5).

2.3. Flow Design - Was ist das?

Unter Flow Design versteht man zwei Dinge: Einmal das Diagramm und einmal die komplette Entwurfsmethode, wobei das Diagramm nur ein Teil dieser ist.

Flow Design soll im Gegensatz zu UML besser geeignet sein, bereits in der Entwurfsphase Anwendung zu finden. Ziel ist es sich auf dem Papier bereits ein Entwurf der Programmstruktur überlegen zu können. Nach Westphal sei es aus der Mode gekommen,

vor dem Programmieren einen Entwurf zu erzeugen, was vor allem daran läge, dass die vorhandenen Entwurfsmethodiken hinderlich seien und einen unnötigen Overhead erzeugen [SCHMZL, S. 2].

Es sei somit üblich geworden die Denkarbeit, wie man seinen Code möglichst sauber strukturieren kann, während dem Programmieren direkt im Sourcecode zu verrichten. Dies sei jedoch laut Ralf Westphal eine eher ungünstige Lösung und behindere eher den kreativen Denkprozess mit unnötiger Schreibarbeit. Auf dem Papier sei man mit einer passenden Entwurfsmethodik schneller und man könne auch verschiedene Ideen schneller ausprobieren, Änderungen machen, oder auch wieder verwerfen, als direkt im Sourcecode.

Es geht jedoch nicht darum den Sourcecode bis ins kleinste Detail in eine Art visuelle Programmiersprache zu pressen, sondern darum, wie man den Code am sinnvollsten in Funktionseinheiten zerlegt und diesen auch gleich einen verständlichen Namen zuordnet. Wie die Funktionalität auf unterster Ebene implementiert wird, wird auf dem Diagramm nicht berücksichtigt. Das ist jedoch keine negative Einschränkung, vielmehr ermöglicht dies, sich beim Entwurf nicht mit unnötigen Implementierungsdetails zu beschäftigen, sondern sich auf das große Ganze - das Zusammenspiel bzw. der Komposition der Funktionseinheiten und den Datenfluss - zu konzentrieren.

Anzumerken wäre noch, dass nicht der Kontrollfluss abgebildet wird, sondern, wie erwähnt, der Datenfluss.

3. Pfeile und Kreise

3.1. RomanNumbers Beispiel

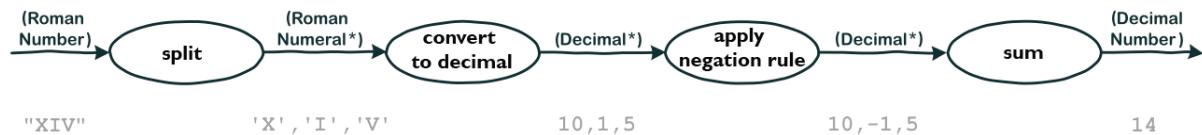


Abbildung 3.1.: Einfaches Flow Design Beispiel (die Beispieldaten unten sind nicht Teil der Notation sondern dienen hier nur dem besseren Verständnis)

Das Beispiel soll auf einfache Weise zeigen, wie ein Flow Design Diagramm aufgebaut ist. Das Programm/Unterprogramm soll eine römische Zahl in eine Dezimalzahl konvertieren.¹

Alle eingekreisten Namen sind Funktionseinheiten. Diese werden in den meisten Fällen im Code als Methoden implementiert. Die Pfeile zeigen den Datenstrom zwischen den Funktionseinheiten. Links die Inputs und rechts die Outputs.

Der einfließende Datenstrom der ersten Funktionseinheit besteht aus einem String. Dieser String wird innerhalb dieser Funktionseinheit in einzelne römische Ziffern zerlegt. Der Datenstrom fließt anschließend in eine weitere Funktionseinheit, die jede römische Ziffer zu der entsprechenden Dezimalzahl konvertiert. Anschließend muss auf den Strom noch die Subtraktionsregel angewendet werden. Diese untersucht den Strom aus Ganzzahlen auf Stellen, wo eine kleinere Zahl vor einer größeren Zahl steht und sie in dem Fall dann negiert. Am Ende wird der Datenstrom zu einer Funktionseinheit hingeleitet, die alle Zahlen aufaddiert. Das Ergebnis ist die Summe aller Zahlen.

¹Beispiel und Flow Design sind von [CODEWHISPERER] entnommen

4. Die Notation

4.1. Datenströme

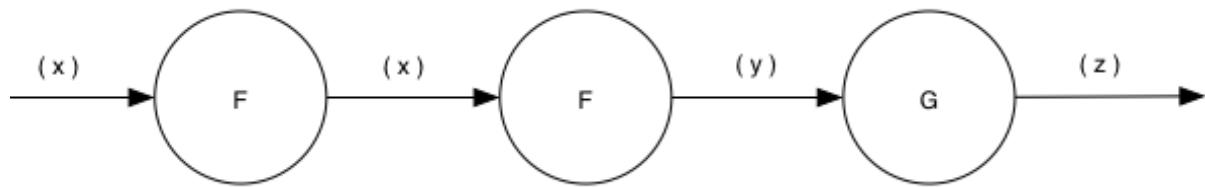


Abbildung 4.1.: Datenfluss zwischen drei Funktionseinheiten

Über den Pfeilen, die die Richtung der Datenflüsse darstellen, werden die im Datenfluss enthaltenen Daten in runden Klammern eingetragen.

Eine leere Klammer bedeutet, dass ein Datenstrom ohne Daten fließt. Die Funktionseinheit wird nur aufgerufen, ohne ihr Daten zu übergeben.

Die Notation erlaubt es auch einzelne Datentypen zusätzlich noch mit einem Namen zu versehen. Was vor allem bei primitiven Datentypen hilfreich sein kann. Der optionale Name wird dem Datentyp durch einen Doppelpunkt getrennt vorangestellt.

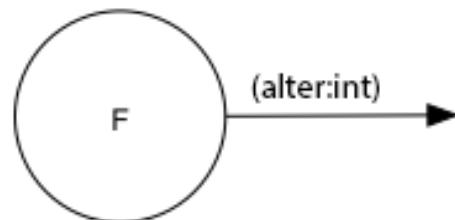


Abbildung 4.2.: Datentyp *int* mit Namen *alter*

Falls der Datentyp aus dem Namen hervorgeht - oder nicht allzu relevant ist - kann auch anstatt des Datentypen nur der Namen in die runden Klammern eingetragen werden.

4.2. Hierarchische Datenflüsse

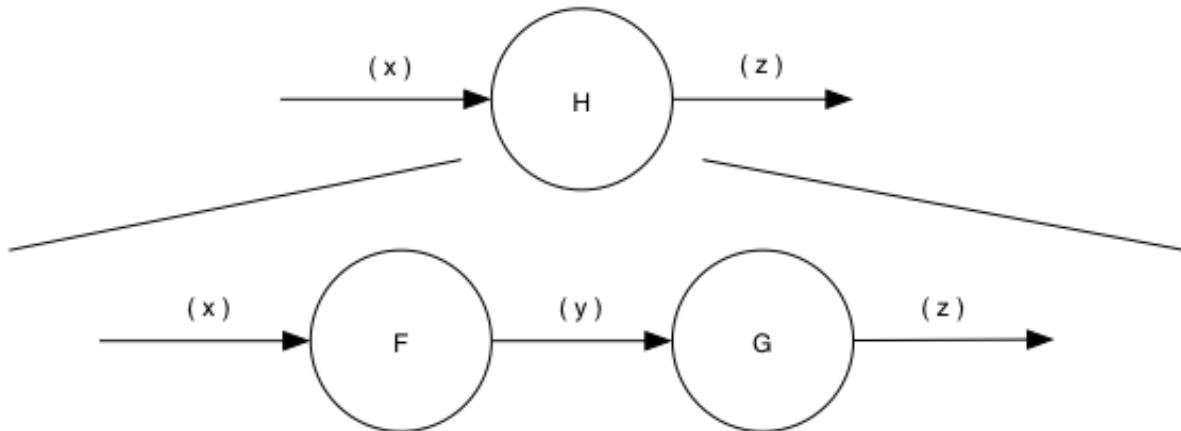


Abbildung 4.3.: Hierarchischer Datenfluss

Das Flow-Design-Diagramm unterstützt die Funktion in eine Funktionseinheit sozusagen hineinzuzoomen. Hier erkennt man die rekursive Eigenschaft der Funktionseinheiten: Eine Funktionseinheit kann wiederum aus mehreren Funktionseinheiten bestehen, die zusammen die Aufgabe erledigen, die die übergeordnete Funktionseinheit beschreibt. Eine solche übergeordnete Funktionseinheit wird als Integration bezeichnet. Hat eine Funktionseinheit keine untergeordneten Funktionseinheiten wird sie als Operation bezeichnet. Was es mit dieser Aufteilung genau auf sich hat, wird im Kapitel 4 erläutert.

4.3. Definition eigener Datentypen

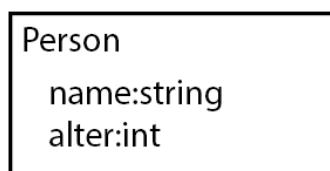


Abbildung 4.4.: Eigener Datentyp

Bei einem Datenstrom, der einen eigenen Datentyp beinhaltet, kann es helfen, den Datentyp innerhalb des Flow Design Diagrammes zu definieren. Dafür wird an einer beliebigen Stelle auf dem Papier eine Box gezeichnet, in der der Datentyp mit seinen Membervariablen aufgelistet wird.¹

¹Diese Notation ist nicht offiziell Teil der Flow Design Notation, sondern eine Ergänzung von Kevin Erath.

4.4. Arrays

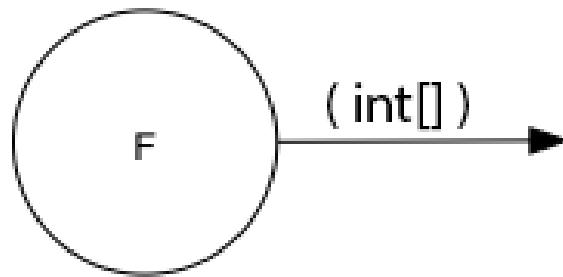


Abbildung 4.5.: Array Notation

Werden Daten als Arrays mit fester Größe übergeben, so wird hinter dem Datentyp eine leere eckige Klammer angehängt. Ist die Arraygröße bekannt, so kann man diese in die Klammer noch zusätzlich eintragen.

4.5. 0 bis n (Datenstrom)

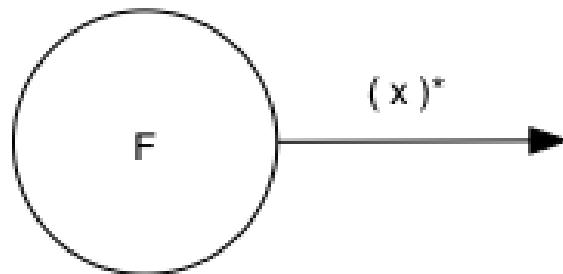


Abbildung 4.6.: Datenstrom Notation

Ein Datenstrom wird mit einem * außerhalb der Klammer dargestellt. Selten wird ein Datenstrom auch mit geschweiften Klammern dargestellt, um ihn von einem optionalen Output (0 bis 1) unterscheiden zu können: {int}.

Ein Datenstrom zeigt an, dass die in der Klammer stehenden Daten keinmal, einmal, oder auch öfters als einmal fließen können.

4.6. Container / Listen

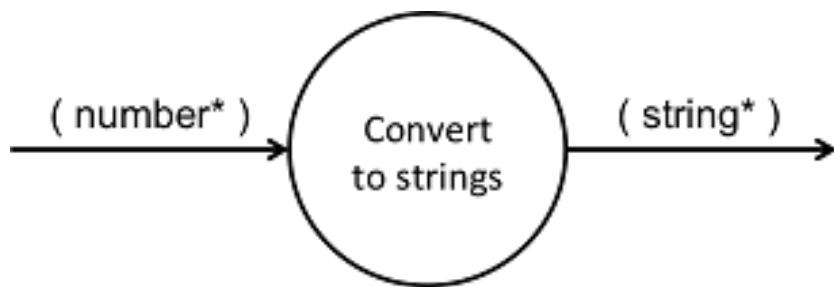


Abbildung 4.7.: Container Notation

Ein Stern innerhalb der Klammer und hinter einem Datentyp besagt, dass dieser in einem Container vorliegt. Die zu bearbeitende Daten können entweder komplett auf einmal an die Funktionseinheit gegeben werden (als Liste, Dictionary, etc.) oder aber - falls die Programmiersprache dies unterstützt - mit `yield` ähnlich wie ein Stream realisiert werden, wo einzelne Elemente bereits abgearbeitet werden können, bevor alle anderen Daten erzeugt wurden.

4.7. 0 bis 1 (optionaler Output)

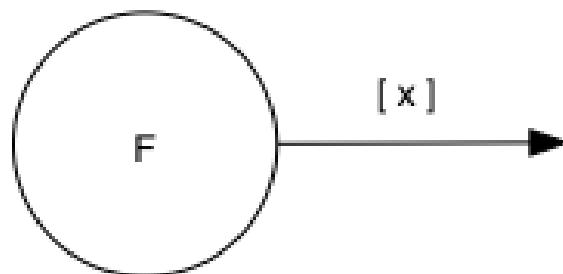


Abbildung 4.8.: Optionaler Output

Mit einer eckigen Klammer lässt sich ein optionaler Output - einmal oder keinmal - darstellen.²

Optionale Outputs können genau wie Datenströme in den meisten Programmiersprachen nicht über ein Rückgabewert einer Methode realisiert werden, da nach jedem Aufruf genau ein Rückgabewert erwartet wird. Wie solche Datenflüsse in C# realisiert werden, wird in Kapitel 5 gezeigt.

²Oft wird jedoch auf diese Notation verzichtet und auch bei optionalen Ausgängen eine runde Klammer verwendet.

4.8. Mehrere Datentypen auf einem Datenstrom

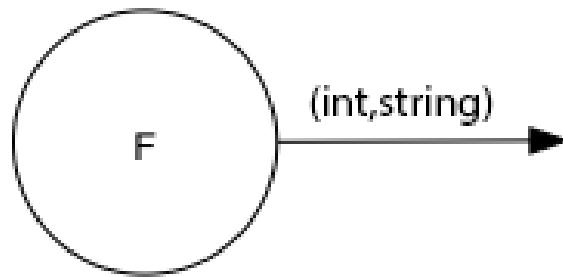


Abbildung 4.9.: Mehrere unterschiedliche Datentypen auf einem Datenstrom

Besteht ein Datenstrom einer Funktionseinheit aus mehrere Datentypen, so werden diese durch Kommas getrennt in die Klammer geschrieben.

Mehrere Outputs lassen sich nicht in allen Sprachen einfach realisieren. Wahlweise kann es mit Tupel realisiert werden, oder es wird aus mehreren Datentypen ein neuer Datentyp erzeugt, der alle Output-Daten beinhaltet.

4.9. Joined Inputs und Pipe-Notation

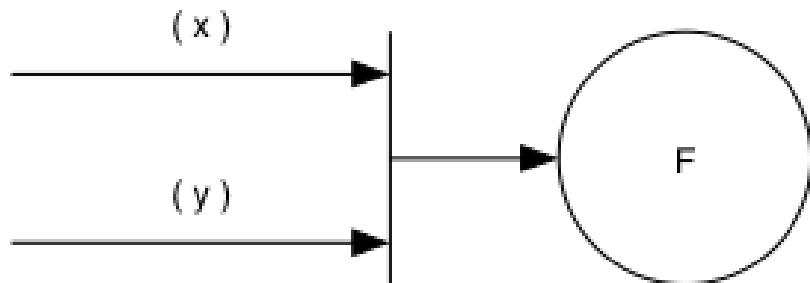


Abbildung 4.10.: Joined Inputs

Falls im Output mehrerer Funktionseinheiten in einen Datenstrom zusammenlaufen sollen und dieser dann als Input in eine anderen Funktionseinheit hineinfließen soll, wird das mit Hilfe eines s.g. Joined Inputs dargestellt. Dieser wird als Linie dargestellt an die mehrere Inputs zusammenlaufen.

Im Code wird dies meistens als Methode umgesetzt, die mehrere Inputparameter entgegennimmt. Wichtig hierbei ist, dass das Bündeln der Datenflüsse nicht Aufgabe der Funktionseinheit F ist, sondern ihrer Umgebung (z.B einer übergeordneten Methode). Die Funktionseinheit F erwartet einfach ein Datenstrom mit zwei Daten x und y und kennt deren Herkunft nicht.

Eine andere Schreibweise, die im Diagramm platzsparender ist als die Joined Inputs, ist die Pipe-Notation. Hiermit kann sich der Output von dem Input der nachfolgenden Funktionseinheit unterscheiden. Dadurch können auch Daten, die aus vorherigen Funktionseinheiten innerhalb des Datenflusses entstanden sind, entnommen und einer Funktionseinheit übergeben werden.

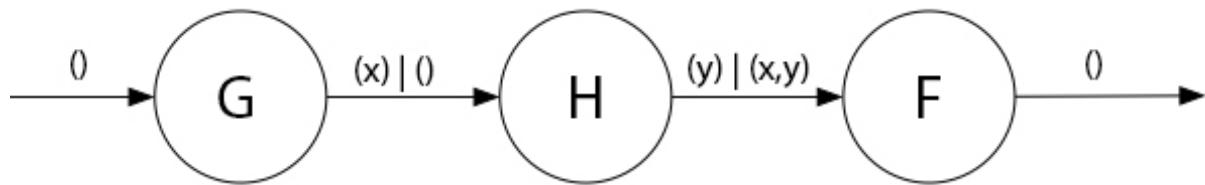


Abbildung 4.11.: Pipe-Notation

In diesem Beispiel produziert G ein x und H ein y . Sowohl G als auch H nehmen keine Daten beim Aufruf entgegen. In die Funktionseinheit F fließen die Daten aus G und H zusammen.

4.10. Tonnen

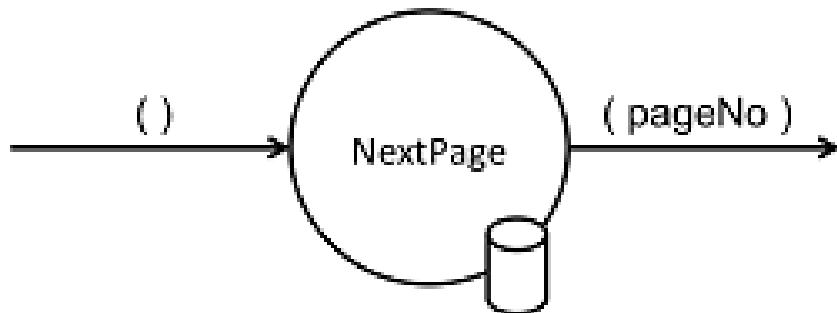


Abbildung 4.12.: Tonnensymbol

Eine Tonne zeigt an, dass die Funktionseinheit state-behaftet ist. In einer OO-Sprache wie C# wäre das in den meisten Fällen ein Lesen oder Schreiben einer Instanzvariable eines Objektes.

4.11. Abhängigkeiten / Provider

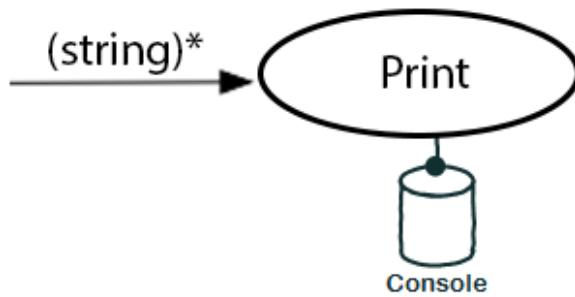


Abbildung 4.13.: Provider

Eine Tonne die mit einer Linie zu einer Funktionseinheit verbunden ist, soll anzeigen, dass die Funktion auf externe Ressourcen zugreift, wie zum Beispiel eine Datei oder Datenbank. Geschieht der Zugriff auf die Ressource über eine Helperklasse, die den Zugriff kapselt, wird anstelle der Tonne ein Dreieck als Symbol verwendet. Eine solche Klasse wird auch als Provider bezeichnet.

Den Kreis kann man sich bildlich wie eine Hand vorstellen, an der sich die Funktionseinheit festhält, und dadurch eine Abhängigkeit symbolisiert.

4.12. GUIs / Programmstart/ Ende



Abbildung 4.14.: Programmstart und Programmende

Wenn eine Funktionseinheit direkt durch den Programmstart aufgerufen wird, so wird dies mit einem leeren Kreis dargestellt. Genauso verhält es sich mit dem Programmende, mit dem Unterschied, dass noch innerhalb des Kreises ein Kreuz ist.



Abbildung 4.15.: UI

Soll dargestellt werden, dass eine Funktionseinheit von einem Event aus der GUI ausgelöst wurde, oder die ausgehenden Daten einer Funktionseinheit in das GUI übergeben werden, so wird ein Viereck am Anfang bzw. Ende des Pfeiles eingezeichnet.

4.13. Klassen / Container definieren

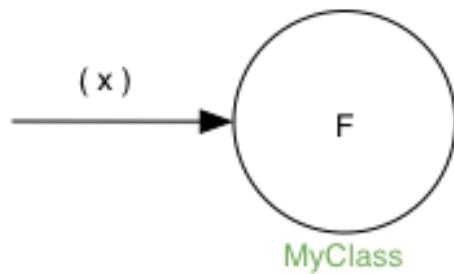


Abbildung 4.16.: Zugehörigkeit zu einer Klasse

Das Definieren von Container und Zuordnen von Funktionseinheiten ist auch einfach möglich. Unter Container versteht man: Klassen, DLLs und Anwendungen.³

Es gibt zwei Möglichkeiten eine Zugehörigkeit zu einem Container zu notieren. Entweder man schreibt direkt unter der Funktionseinheit den Namen des Containers, oder man umrandet mehrere Funktionseinheiten und notiert den Namen des Containers am Rand der Umrandung.

³Definition des Container-Begriffs: [SCHMZL, S. 50 ff.]

5. Implementation

Flow Design empfiehlt zwei Prinzipien, die bei der Implementierung, einzuhalten sind:

- Trennen von Integrationen und Operationen (IOSP),
- keine funktionale Abhängigkeiten in Operationen zu anderen Funktionseinheiten aus dem selben Programm (PoMO).

Um was es sich dabei im Detail genau handelt, wird in diesem Kapitel erläutert.

5.1. IODA Architektur

Wie schon in dem vorherigen Kapitel angemerkt, unterscheidet Flow Design zwei unterschiedliche Arten von Funktionseinheiten. Integrationen und Operationen. Die IODA Architektur beschreibt die Eigenschaften von diesen beiden genauer.

IODA steht für: Integration Operation Data API¹.

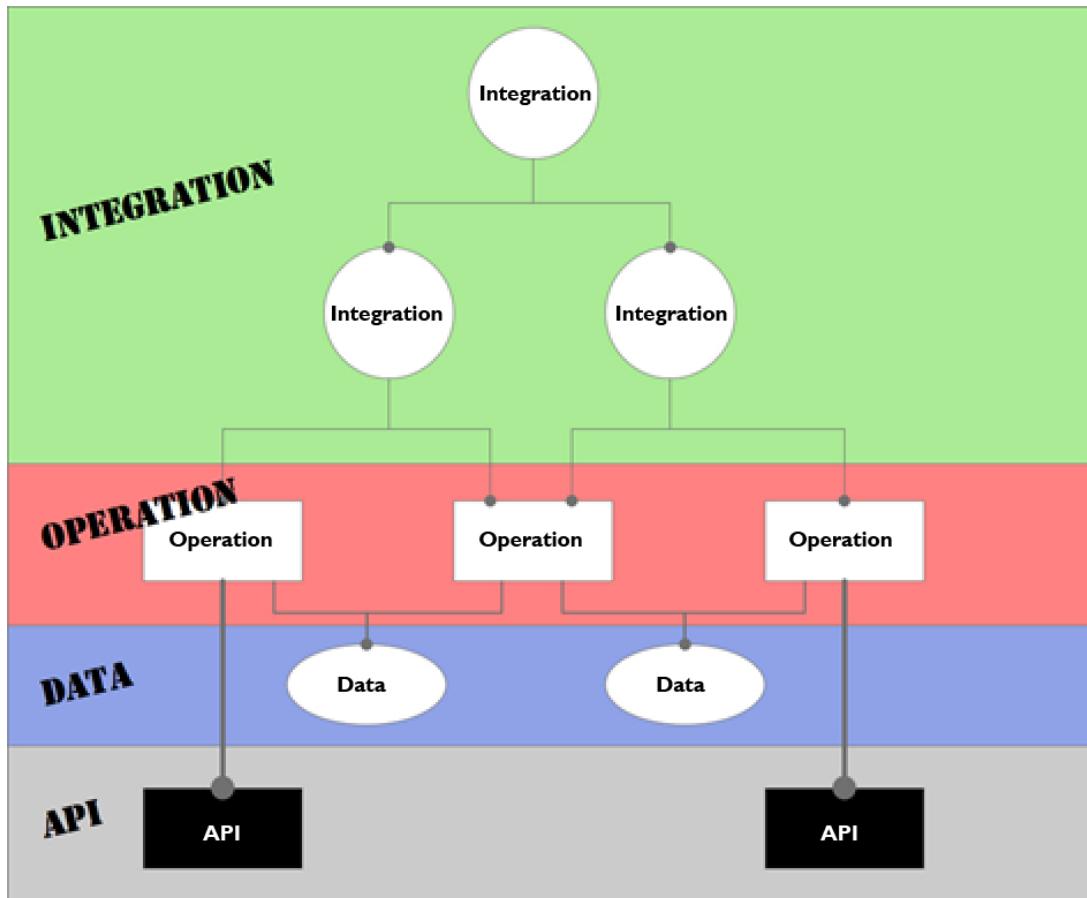


Abbildung 5.1.: IODA Architektur, Quelle:[IODA]

Erläuterung des Schaubildes

Operationen sollen komplett unabhängig vom Rest ihrer Umwelt funktionieren und dürfen aus diesem Grund andere Funktionseinheiten nicht kennen.

Die Aufgabe einer Integration ist, die unabhängigen Operationen in das große Ganze zu integrieren.²

Integrationen integrieren andere Integrationen und/oder Operationen in das Programm. Sie dürfen also von anderen Funktionseinheiten funktional abhängig sein.

¹IODA.

²Ralf Westphal spielte auch mit den Gedanken diese Funktionseinheiten als Koordinatoren oder Kompositionen zu bezeichnen.

Im Gegensatz dazu dürfen Operationen keine Integrationen oder andere Operationen kennen. Sie dürfen aber auf Daten zugreifen. Über diese entsteht auch die einzige Möglichkeit der Kommunikation zwischen Operationen. Mit Daten sind hauptsächlich nicht-persistente Daten gemeint, die als Stream durch die Operationen fließen. Daten in Form von Objekten und primitive Datentypen, die von den integrierenden Funktionseinheiten koordiniert werden. Sowohl Operationen als auch Integrationen dürfen Daten erzeugen. Beispielsweise das Aufrufen eines Konstruktors oder Deklarieren einer lokalen Variablen. Das „Verdrahten“ von Datenflüssen übernehmen jedoch die Integrationen (was auf der Abbildung 5.1 leider nicht so deutlich herauskommt). Mit Daten können auch persistente Daten auf der Festplatte gemeint sein, wobei ein Zugriff auf persistente Daten eigentlich immer über eine API geschieht und somit würden solche Aufrufe dann zu der Gruppe API zählen.

Die IODA Architektur besagt, dass API-Aufrufe sich nur innerhalb von Operationen befinden dürfen, damit diese Informationen gekapselt und die Integrationen frei von API-spezifischem Wissen bleiben.

Anhand einer Flow Design Skizze, kann man leicht herausfinden, welche Methoden Operationen sind und welche Integrationen. Alle Leaf-Knoten sind Operationen, der Rest sind Integrationen.

PoMO (Principle of Mutual Oblivion)

„Ein Producer kennt seinen Consumer nicht. Ein Consumer kennt seinen Producer nicht. Das nenne ich das Principle of Mutual Oblivion (PoMO, Prinzip der gegenseitigen Nichtbeachtung).“ (Ralf Westphal [SCHMZL, S. 80])

Dieses Prinzip besagt, dass Funktionseinheiten sich nicht gegenseitig kennen sollen. Es soll auch verhindert werden, dass eine Einheit eine andere aufruft und von deren Ergebnis abhängig ist, bzw. auf das Ergebnis wartet. Eine Funktionseinheit soll, nachdem sie die Daten bearbeitet hat, diese nach außen weiterreichen und nicht wissen, wer die Daten entgegennimmt. Dieses Prinzip verhindert eine Koppelung zwischen den einzelnen Funktionseinheiten.

Um jedoch ein Zusammenspiel zwischen den einzelnen entkoppelten Einheiten zu ermöglichen, bedarf es einen oder mehrere „Koordinatoren“, welche diesem Prinzip nicht entsprechen müssen. Nur so kann aus vielen kleinen Funktionseinheiten ein großes Ganzes werden, welches eine komplexe Aufgabe lösen kann.

Diese Koordinatoren werden Integrationen genannt, sie ermöglichen das Zusammenspiel. Da diese nun eine starke Kopplung zwischen den Funktionseinheiten eingehen, müssen diese so leichtgewichtig wie möglich sein, nur so ist diese Abhängigkeit nicht schmerhaft.

IOSP (Integration Operation Segregation Principle)

Dieses Prinzip besagt, dass eine Funktionseinheit entweder eine Operation oder eine Integration ist und beide Verantwortungsbereiche nicht vermischt werden dürfen.

1. Operationen

Operationen sind Methoden, die Logik/ Kontrollstrukturen enthalten dürfen. In C# wären das:

- if, else
- switch, case
- for, foreach,
- while, do
- try, catch, finally
- goto

Gleichzeitig müssen die Operationen das PoMO Prinzip erfüllen. Nach dieser Regel dürfen sie keine Kenntnisse über die Herkunft der Daten haben. Auch soll den Operationen unbekannt sein, was im Anschluss mit den Daten passiert. Aus diesem Grund darf auch kein Rückgabewert erwartet werden, da sich daraus Rückschlüsse bzw. Erwartungen verknüpfen würden. Ein Funktionsaufruf ist nur indirekt über Callback-Techniken, wie Funktionszeiger oder Events möglich. Wichtig dabei ist, dass der dahinterstehende Aufgerufene nicht bekannt ist und dieser auch kein Rückgabewert verwendet werden darf. Nur so kann die Entkopplung sichergestellt werden.

Durch diese Regel wird einer Operation ermöglicht eine andere Funktionseinheit aufzurufen, ohne dass sie das PoMO bricht. Sie bestimmt nicht selbst, welche Funktionseinheit sie aufruft, sondern die übergeordnete, welche die Operation aufruft (und somit automatisch eine Integration sein muss, welche die PoMO Bedingung nicht erfüllen muss). Wie das praktisch aussieht, wird im Laufe des Kapitels anhand von konkreten Beispielen genauer veranschaulicht.

Operationen sind also imperative programmiert. Imperative Programmierung ist ein Programmierstil, mit dem Fokus auf das **wie** ein Problem gelöst werden soll. Im Gegensatz dazu steht der deklarative Ansatz. Beim deklarativen Programmieren steht der Fokus auf das **was** getan werden soll und nicht so sehr, wie es im Detail genau angestellt wird. Ein Beispiel hierfür wären SQL Befehle. Hier wird nur gesagt, was man haben möchte und das Programm kann dann die Anfrage nochmal untersuchen und selbst bestimmen, wie es die Anfrage am besten ausführt.

2. Integrationen

Die Integrationen werden nach Flow Design deklarativ programmiert. Diese Funktionseinheiten dürfen anders als die Operationen, andere Funktionseinheiten aufrufen, sie also kennen. Die Integrationen erfüllen also nicht das *Principle of Mutual Oblivion*. Der Unterschied beim Flow Design ist jedoch, dass eine bewusste Trennung eingehalten wird.

Integrationen dürfen auch auf die Terminierung einer Funktionseinheit warten und den Rückgabewert an andere Funktionseinheiten weiterreichen. Dafür dürfen sie keine Logik im Sinne von Kontrollstrukturen beinhalten. Auch dürfen sie keine API-spezifischen Befehle kennen, wie zum Beispiel Zugriffe auf Ressourcen wie Konsole, UI oder Dateien.

Die Businesslogik, das was die Funktionalität erzeugt, befindet sich in Operationen und sind entkoppelt von ihrer Umgebung. Sie bekommen einfach nur von irgendwo her einen Input (bzw. bei keinen Inputparametern einfach ausgeführt werden) und führen damit die von ihnen implementierte Logik aus und geben das Ergebnis nach außen. Beim Weiterreichen nach außen kennt die Funktionseinheit jedoch den Empfänger nicht.

Tabelle - IOSP auf einen Blick

	Operationen	Integrationen
Rechenoperationen (+, *, %, ...)	Ja	Nein
Kontrollstrukturen (if, else, while, for, foreach, ...)	Ja	Nein
API-Aufrufe (Methoden von Bibliotheken)	Ja	Nein
Ressourcen-Zugriffe (Dateien, Datenbanken etc.)	Ja	Nein
Standard Library, LINQ	Ja	Ja
Namen andere Funktionseinheiten kennen	Nein	Ja
Auf Rückgabewert warten	Nein	Ja

Tabelle 5.1.: IOSP Übersicht

5.2. Beispiel foreach und Funktionsaufruf

```
static void FormatAndPrintStrings(List<string> lines)
{
    foreach(line in lines)
    {
        string s = MyComplexFormattingFunction(line);
        Console.WriteLine(s);
    }
}
```

Listing 5.1: FormatAndPrintStrings nicht IOSP-konform

Derartiger Code wird wohl in den meisten C#-Codebasen zu finden sein und doch ist er nach Flow Design falsch.

In diesem Beispiel wurde Logik (foreach) gemischt mit einem expliziten Methodenaufruf, sowie ein Zugriff auf eine externe Ressource, die Konsole.

Diese Methode ist somit nicht IOSP konform.

Es ist etwas ungewohnt, das Trennen von Integrationen und Operationen im Code auch zu berücksichtigen. Eine For-Schleife über eine Kollektion laufen zu lassen und jedes Element an eine Untermethoden weiterzureichen ist etwas, was wohl viele Programmierer regelmäßig so schreiben. Das so etwas nun nicht mehr erlaubt ist, braucht eine gewissen Umgewöhnungszeit.

Hier nun die Umsetzung in Flow Design mit einfachsten Mitteln:

```
// Integration
static void FormatAndPrintStrings(List<string> lines)
{
    List<string> formattedLines = FormatLines(lines);
    PrintLines(formattedLines);
}

// Operationen
static List<string> FormatLines(List<string> lines)
{
    List<string> result = new List<string>();
    foreach(line in lines)
    {
        string formattedstring;
        // do complex formatting here
        result.Add(formattedstring)
    }
    return result;
}

static void PrintLines(List<string> lines)
{
    foreach(line in lines)
    {
        Console.WriteLine(line);
    }
}
```

Listing 5.2: FormatAndPrintStrings Variante 1

Als Flow Design dargestellt, sieht der Code folgendermaßen aus:

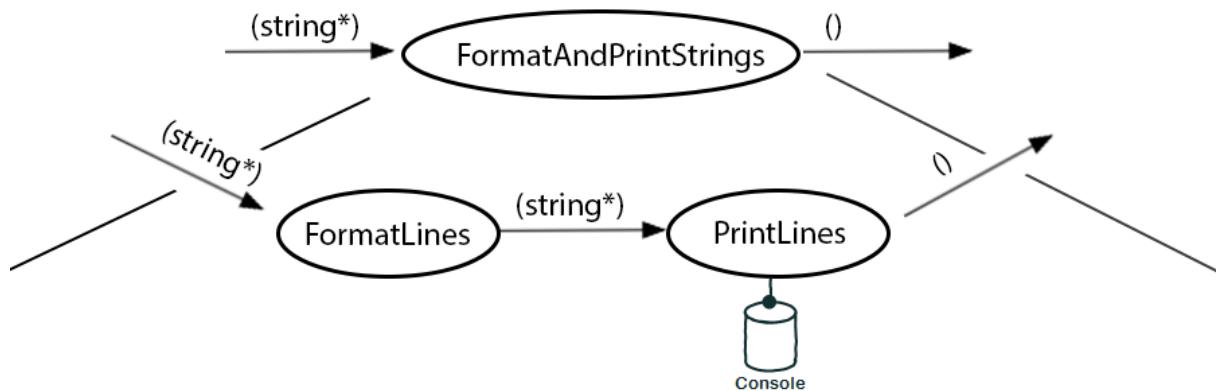


Abbildung 5.2.: FormatAndPrintStrings Variante 1 - Flow Design

Die Methode wurde aufgeteilt in eine Integration (FormatAndPrintStrings) und zwei Operationen. Im ersten Beispiel hat die Methode zwei Aufgaben erfüllt, sie hat die Formatierung-Methode integriert und das Ergebnis ausgegeben.

Nun sind Integration, Ausgabe und Formatierung sauber getrennt. Womit das SRP auch erfüllt ist. Dadurch wurde eine Entkopplung geschaffen, die Änderungen am Code leichter macht. Besonders vorteilhaft ist, dass das UI vom Rest getrennt ist.

Jedoch wurde der Code nun deutlich länger. Die Foreach-Schleife ist in beide Operationen gelandet und das Initialisieren und Befüllen der temporären Liste in FormatLines nimmt auch etwas Platz ein. Dazu kommt noch, dass die String-Formattierungslogik nun eingebettet in dieser Foreach-Schleife liegt, welche sich vorher getrennt in einer extra Methode befand.

Elegantere Lösung mit Actions:

```

// Integrationen
static void FormatAndPrintStrings(List<string> lines)
{
    IterateOverLines(lines, onLine=PrintFormat );
}

static void FormatAndPrint(string line)
{
    var formattedline = MyComplexFormattingFunction(line);
    Print(formattedline);
}

// Operationen
static void IterateOverLines(IEnumerable<string> lines,
    Action<string> onLine)
{
    foreach(line in lines)
    {
        onLine(line);
    }
}
  
```

```

    }
}

static void Print(string line)
{
    Console.WriteLine(line);
}

```

Listing 5.3: FormatAndPrintStrings Variante 2

Als Flow Design dargestellt, sieht der Code folgendermaßen aus:

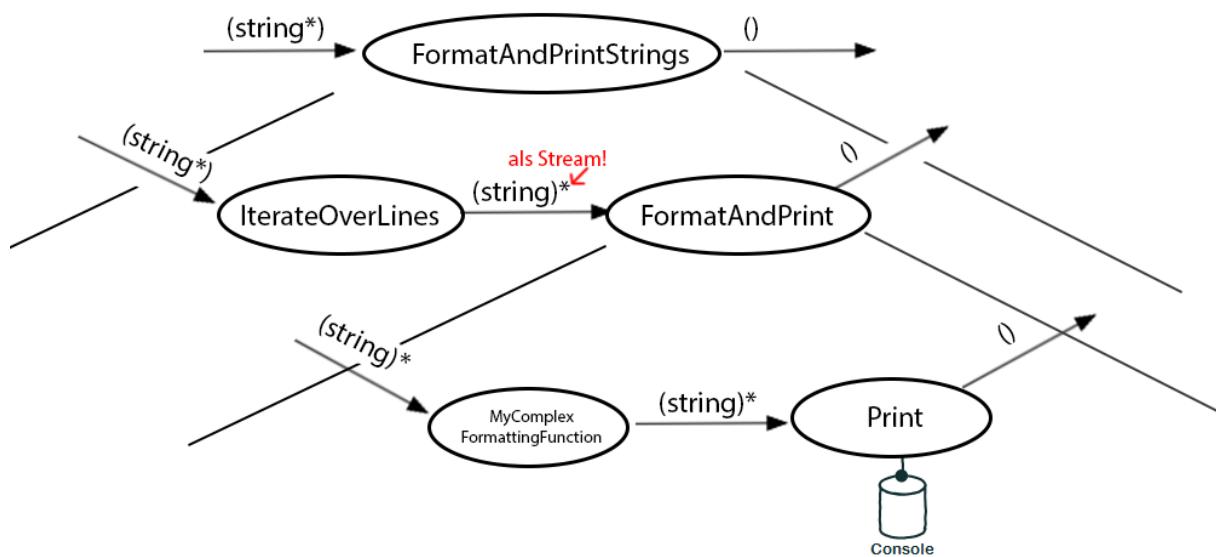


Abbildung 5.3.: FormatAndPrintStrings Variante 2 - Flow Design

Noch eleganter mit Actions und Lambdas:

```

static void FormatAndPrintStrings(List<string> lines)
{
    IterateOverLines(lines,
        line => {
            var formattedline = MyComplexFormattingFunction(line);
            Print(formattedline);
        });
}

static void IterateOverLines(IEnumerable<string> lines,
    Action<string> onLine)
{
    foreach(line in lines)
    {
        onLine(line);
    }
}

```

```

}

static void Print(string line)
{
    Console.WriteLine(line);
}

```

Listing 5.4: FormatAndPrintStrings Variante 3

Eine weitere Möglichkeit besteht darin Datenfluss orientierte Sprachfeatures zu verwenden. Somit hängt diese Möglichkeit stark von der verwendeten Programmiersprache ab. In C# existiert eine Kategorie an Methoden, die speziell auf das Arbeiten mit Datenflüssen ausgerichtet ist, diese werden zusammengefasst unter dem Namen LINQ (Language Integrated Query).

Mit Hilfe von LINQ lässt sich obiges Beispiel zu einem IOSP konformen Einzeiler reduzieren.

```

static void FormatAndPrintStrings(List<string> lines)
{
    lines.Select( x => MyComplexFormattingFunction(x)).ToList()
        .ForEach(Console.WriteLine);
}

```

Listing 5.5: FormatAndPrintStrings Variante LINQ

Man könnte sich nun darüber streiten, was man nun damit gewonnen hat. Schließlich enthält die Funktion mit LINQ im Grunde genommen fast genau die selbe Logik, wie das nicht IOSP-konforme Beispiel, nur in einer anderen Schreibweise. Der Nutzen dieser Regel erschließt sich erst, bei größeren Codebasen und kommt bei kleinen Beispielen oft nicht zum Vorschein. Erst wenn die Integrationen mehr machen, als nur eine Funktion aufrufen, wird das Entkoppeln nützlich. Außerdem ist der Fall einer Foreach-Schleife und ein Funktionsaufruf eine Koppelung, die nicht so dramatisch ist. Man könnte für diesen Fall sogar eine Ausnahme machen und sie erlauben.

Zusammenfassend kann gesagt werden, dass eine größere Lesbarkeit von IOSP konformen Programmcode entsteht, umso mehr moderne und funktionale Sprachfeatures eine Programmiersprache bietet.

5.3. C# Features um Datenflüsse zu implementieren

Um Flow Design gemäß der IODA Architektur umzusetzen, helfen einem in C# einige Features, die in diesem Kapitel vorgestellt werden.

LINQ und Lambdas

Streng genommen würde es die IODA Architektur nicht erlauben die Methoden der Standardbibliothek innerhalb von Operationen zu verwenden. Jedoch würde das den Code nur unnötig verkomplizieren ohne wirklich ein Nutzen daraus zu gewinnen. Aus diesem Grund ist es sinnvoll die Methoden der Standardbibliothek der Sprache in Operationen als auch in Integrationen zu erlauben. In C# gehört dazu auch die Methodensammlung LINQ. LINQ ist eine in C# integrierte Ansammlung an Methoden die in Verbindung mit Objekten, die das IEnumerable Interface implementieren, eingesetzt werden können. IEnumerable ist das Interface einer Aufzählungsklasse. Daran lässt sich bereits erahnen, dass LINQ auf das Arbeiten mit Datenflüssen spezialisiert ist.

In den meisten Fällen werden den LINQ Methoden ein Lambda-Ausdruck übergeben. Dieser wird auch als Selector bezeichnet, oder im Falle von Bedingungen als Predicate. Lambda-Ausdrücke werden nach Flow Design Regeln, wie eigenständige Funktionseinheiten betrachtet. Somit darf ein Lambda innerhalb einer Integrationen auch eine Operation sein.

Im folgenden Abschnitt werden hier nur ein paar der am häufigsten verwendeten Methoden erläutert.

1. Modifizieren

Folgende Methoden verändern den Datenstrom und liefern einen neuen Datenstrom zurück (mit Ausnahme von ForEach).

Select	Selektiert jedes Element und der Sequenz und modifiziert es. Zurückgegeben wird eine Sequenz der modifizierten Elemente.
ForEach (nur für List-Klasse)	Iteriert über die Sequenz und führt mit jedem Element den Selector-Ausdruck aus. Im Gegensatz zu Select wird keine Sequenz zurückgeliefert. Diese Methode ist nicht Teil von LINQ, sondern gehört ausschließlich zu der List-Klasse. Da sie jedoch oft Verwendung in LINQ-Ausdrücke findet, wird sie hier mit aufgezählt.
First, Last	Gibt das erste/letzte Element der Sequenz zurück, das eine bestimmte Bedingung erfüllt.
OrderBy	Ordnet die Sequenz mit Hilfe eines keySelector-Ausdrucks. Dieser bestimmt das Sortierkriterium. In manchen Fällen (Elemente sind Zahlenwerte, oder Strings), kann dieser weggelassen werden, falls das Standard-Verhalten gewünscht ist.
Distinct	Duplikate werden aus der Sequenz gelöst.
Join	Zwei Sequenzen werden zu einer zusammengefasst.

Tabelle 5.2.: LINQ - Sequenzen Modifizieren

2. Filtern

Where	Filtern der Sequenz anhand des Predicate. Zurückgegeben wird eine Sequenz von Elementen, die das Filterkriterium entsprachen.
-------	---

Tabelle 5.3.: LINQ - Sequenzen Filtern

3. Überprüfungen

Diese Methoden liefern einen Boolean als Rückgabewert zurück.

Any	Wendet auf jedes Element den Selector-Ausdruck solange an, bis bei einem Element der Ausdruck wahr wird. Dann wird true zurückgegeben, ansonsten false.
Contains	Ähnlich wie Any, nur dass kein Selector übergeben wird, sondern ein Element, der selben Klasse, wie die Elemente des Containers. Befindet sich das Element in dem Container, dann wird true zurückgegeben, ansonsten false.
All	Ähnlich wie Any mit dem Unterschied, dass nur dann true zurückgegeben wird, wenn für alle Elemente des Containers der Ausdruck wahr ist.

Tabelle 5.4.: LINQ - Sequenzen Überprüfen

4. Berechnungen

Bei Container mit Zahlenwerten (int, float, decimal,...) als Elementen, können nachfolgende Funktionen ohne zusätzliche Parameter aufgerufen werden. Falls dies nicht der Fall ist, muss ein Selector-Ausdruck, wahlweise als Lambda-Ausdruck, mit übergeben werden. Mit dem Selector kann bestimmt werden, wie die mathematische Rechenoperationen mit jedem Element umzugehen hat.

Sum	Aufsummieren der Elemente
Max	Gibt das Element mit dem höchsten Wert zurück.
Min	Gibt das Element mit dem niedrigsten Wert zurück.
Count	Zählt die Elemente des Containers und gibt die Anzahl zurück.
Average	Berechnet den Durchschnitt der Sequenz.

Tabelle 5.5.: LINQ - Berechnungen

5. Überspringen und Nehmen

Diese Methoden liefern genau wie die modifizierenden Methoden als Rückgabewert eine neue Sequenz an Daten zurück.

TakeWhile	Nimmt Elemente solange aus dem Container, bis eine Bedingung erfüllt ist. Es wird eine Sequenz von allen genommenen Elementen zurückgegeben.
Skip	Überspringt eine Anzahl an Elementen.
SkipWhile	Überspringt die ersten Elemente einer Sequenz, solange bis die Bedingung von einem Element nicht erfüllt wird, dann wird ohne weitere Überprüfungen der Rest der Sequenz zurückgegeben.

Tabelle 5.6.: LINQ - Überspringen und Nehmen

6. Konvertieren

Sequenzen können mit Hilfe eines einfachen Methodenaufrufs zu einem bestimmten Typ von Container konvertiert werden. Zum Beispiel: `ToList` oder `ToDictionary`.

7. Parallel Verarbeitung

Datenströme können von LINQ auch parallel verarbeitet werden. Dazu konvertiert man die Sequenz mit `ToParallel()` zu einem PLINQ Datenstrom. Anschließend ausgeführte Methoden werden, falls möglich, parallel verarbeitet.

`yield return`

Hiermit kann man ein Producer-Consumer Pattern implementieren. Voraussetzung ist hierfür, dass der Rückgabewert der Methode ein `IEnumerable` Interface ist.

Das folgende Flow Design soll mit `yield return` realisiert werden.

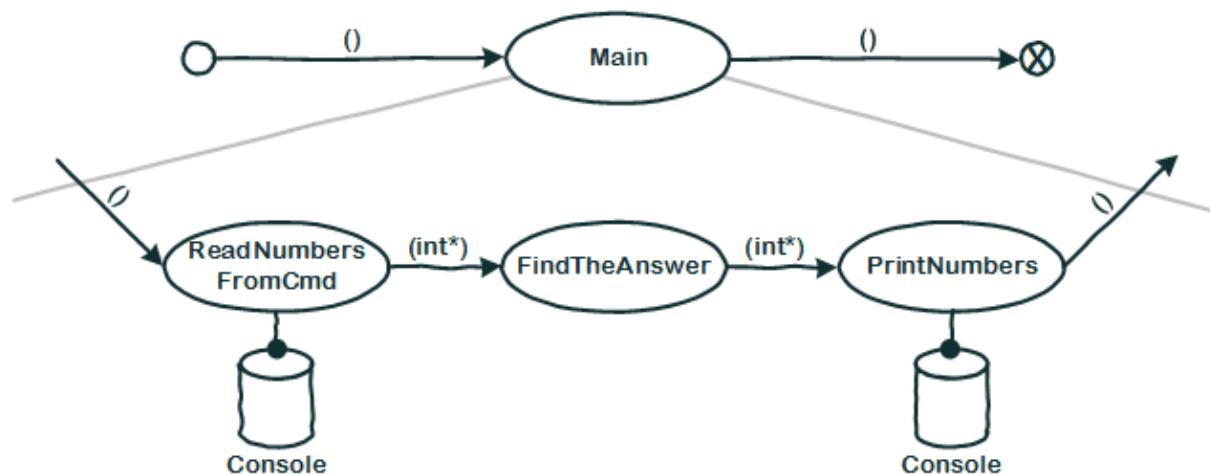


Abbildung 5.4.: FindTheAnswer - Flow Design, Quelle: [CODEWHISPERER]

Das Programm ist eine Konsolenanwendung, die dem Benutzer eine Eingabe erlaubt. Wenn die Eingabe die Zahl 42 entspricht, wird das Programm beendet, wenn nicht, dann wird die Zahl ausgegeben und der Benutzer kann wieder eine Zahl eingeben. Das wiederholt sich, solange bis der Benutzer die Zahl 42 eingetippt hat.

1. Erläuterung des Schaubildes

Die Main-Methode wird nach dem Programmstart (leerer Kreis) ohne Parameter aufgerufen. Danach ruft diese die Methode ReadNumbersFromCmd auf, welche aus der Konsole eine Eingabe liest und sie zu einem int konvertiert. Der int-Wert wird von der Main-Methode entgegengenommen und wird in dieser an die Methode FindTheAnswer weitergereicht. Diese Methode hat die Aufgabe den entgegengenommenen int-Wert mit der Zahl 42 zu vergleichen. Wenn die Zahl 42 ist, wird der Datenstrom abgebrochen. Wenn es nicht die 42 war, dann wird der int-Wert nach außen gereicht und die Main-Methode gibt die Zahl an die PrintNumber-Methode weiter. PrintNumber gibt die Zahl in die Konsole aus. Wenn der Datenstrom abbricht, returned die Main-Methode und das Programm wird beendet.

2. Implementation

```
static void Main()
{
    IEnumerable<int> numbers = ReadNumbersFromCmd();
    IEnumerable<int> answer = FindTheAnswer(numbers);
    PrintNumbers(answer);
}

public static IEnumerable<int> ReadNumbersFromCmd()
{
    while (true)
    {
        var line = Console.ReadLine();
        yield return int.Parse(line);
    }
}

private static IEnumerable<int> FindTheAnswer(
    IEnumerable<int> numbers)
{
    return numbers.TakeWhile(x => x != 42);
}

private static void PrintNumbers(IEnumerable<int> numbers)
{
    foreach (var number in numbers)
        Console.WriteLine(number);
}
```

Listing 5.6: Find the answer Implementation

Der Producer ist in dem Fall der `ReadNumbersFromCmd`. Dieser produziert einen endlosen Stream an `int`-Daten. Es wird jedoch immer nur ein Element erzeugt und erst nachdem der Consumer das Element abgefragt hat, wird ein neues Element erzeugt. Wenn nichts mehr konsumiert wird, wird auch nichts mehr produziert. Den Abbruch der Endlosschleife (also das Stoppen des Datenflusses) kann somit auch eine andere Funktion außerhalb der Schleife übernehmen.

5.4. Datenströme mit mehreren Wegen

Ein Output-Weg mehrere Empfänger

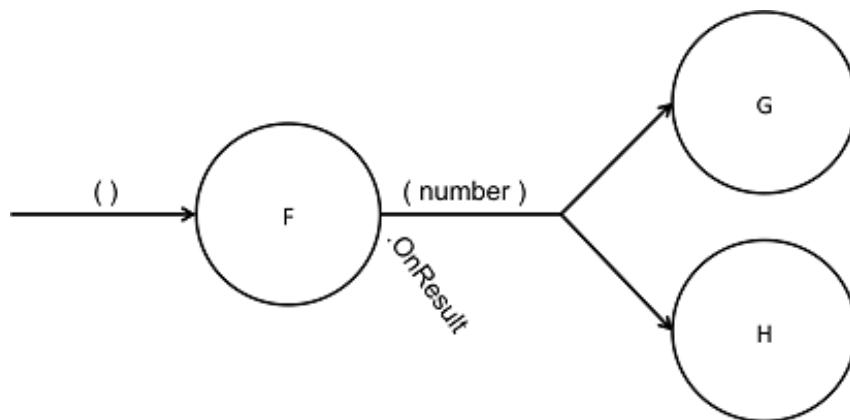


Abbildung 5.5.: Ein Ausgang mehrere Empfänger

Falls ein Output³ an mehrere Empfänger weitergereicht werden soll, so gibt es mehrere Möglichkeiten dies zu realisieren.

Die beste Möglichkeit ist, wenn die übergeordnete Integration den Rückgabewert von F an die beiden nachfolgenden Funktionseinheiten einfach weiterreicht. Eine weitere Möglichkeit wäre, wenn der Methode F eine Methodenreferenz mit übergeben wird, und die übergeordnete Integration ruft G und H in einem Lambda-Ausdruck auf. Die dritte Möglichkeit besteht darin Events zu nutzen. Leider bedarf es dann bei der Benutzung der API mehr Vorsicht, da man sich vorher auf ein Event registrieren muss, bevor man die gewünschte Funktion aufrufen kann.

```

static void Main()
{
    var number = F();
    G(number);
    H(number);
}

static int F()
  
```

³Die Notation erlaubt es einen Ausgang zu benennen, indem man diesen mit einem Punkt vorangestellt unterhalb des Pfeiles notiert. Siehe Abbildung 5.5 *.OnResult*.

```

{
    return 42;
}

static void G(int number){}
static void H(int number){}

```

Listing 5.7: Mehrere Empfänger eines Outputs

Mehrere Output-Wege

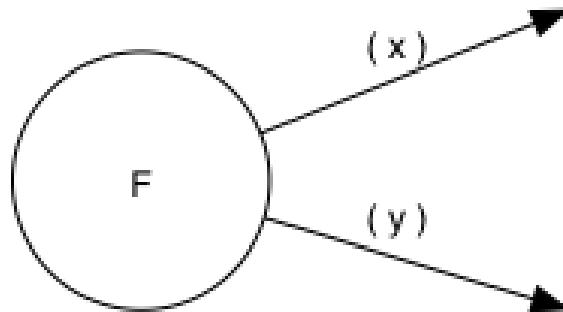


Abbildung 5.6.: Eine Funktionseinheit mit mehreren Ausgängen

Wäre es für eine Operation erlaubt eine andere Funktionseinheit zu kennen, so wäre es möglich die nachfolgenden Methoden per Namen innerhalb von *F* aufzurufen. Da aber Operationen entkoppelt von ihrer Umwelt sein sollen, geht das nicht.

Hat eine Funktionseinheit zwei Output-Wege, so gibt es zwei Deutungsmöglichkeiten: Entweder kommen immer beide Daten *x* und *y* zurück oder aber, es kann auch vorkommen, dass *x* und *y* nicht immer zurückgegeben werden. Im ersten Fall, wäre eine Umsetzung durch ein Tupel als Rückgabewert machbar. Ist jedoch nicht gewährleistet, dass immer beide Werte zurückgegeben werden, so muss man in C# die Outputwege als Methodenreferenz über die Argumente der Methode mitgeben. Somit werden die Verantwortlichkeiten bewahrt und die übergeordnete Integration koordiniert weiter den Datenfluss. Die Operation selbst kennt nun keine anderen Funktionseinheiten, sie weiß nur, dass sie zwei Ausgänge besitzt.

Ist diese Mehrdeutigkeit nicht erwünscht, so gibt es auch die Möglichkeit den Kontrollfluss in das Diagramm hier mit aufzunehmen. Man kann in den Winkel der beiden Pfeile notieren, ob beide Datenflüsse fließen, oder immer nur einer. Oft reicht es aber auch schon aus die Output-Wege zu benennen, damit ersichtlich wird, ob es sich um ein *and*, *or* oder *xor* handelt. Zum Beispiel würde ein *onError* und *onSuccess* auf ein *xor* hindeuten.

Alternativ könnte man auch hier Events nutzen, was aber nur in seltenen Fällen zu empfehlen ist. Das Problem ist die zu starke Entkopplung - die Registrierung erfolgt an anderer Stelle als die Verwendung. Dadurch verliert man schnell den Kontext.

Üblicherweise entstehen mehrere Output-Wege, sobald eine If-Else-Logik verwendet wird.

Ein typischer Programmierstil in C# veranschaulicht folgendes Beispiel:

```
static void CheckAndSaveToFile(string inputstring)
{
    var filename = @"C:/test.txt";

    if (IsCorrectFormatted(inputstring))
        SaveStringToFile(inputstring, filename);
    else
        PrintError("Wrong Input Format");
}

static bool IsCorrectFormatted(string inputstring)
{
    bool isCorrectFormatted = false;

    // do string format checking here

    return isCorrectFormatted;
}
```

Listing 5.8: Typerischer Programmierstil, der nicht IOSP-konform ist

Eine komplizierte Bedingungsüberprüfung in eine separate Methode auszulagern gilt als guter Programmierstil. Flow Design geht hier etwas weiter. Da es untersagt ist, eine Kontrollstruktur und ein Methodenaufruf in einer Methode zu kombinieren, besteht die Lösung darin, auch die If-Else-Anweisung in die ausgelagerte Methode zu extrahieren und die zwei möglichen Ausgänge als Actions in die Methode zu übergeben. Mithilfe der „Named Parameter“ lässt sich die Leserlichkeit weiter steigern (vor allem wenn es mehr als zwei Ausgänge gibt, da man jedem Ausgang einen sinnvollen Namen geben kann).

```
static void CheckAndSaveToFile(string inputstring)
{
    var filename = @"C:/test.txt";

    CheckIsCorrectFormat(inputstring,
        onCorrect: () => SaveStringToFile(inputstring, filename),
        onError: () => PrintError("Wrong Input Format"));
}

static void CheckIsCorrectFormat(string inputstring, Action
    onCorrect, Action onError)
{
    bool isCorrectFormatted = false

    // do string format checking here

    if (isCorrectFormatted)
        onCorrect();
    else
        onError();
}
```

```
    onError();
}
```

Listing 5.9: IOSP-konforme Variante

Wie in diesem Beispiel zu erkennen ist, ist es möglich innerhalb eines Lambdas auf lokale Variablen der übergeordneten Methode zuzugreifen (Closure). Dies erlaubt es der Integration einer nachfolgenden Operation (hier `SaveStringToFile`) Parameter zu übergeben, die die erste Operation (hier `CheckIsCorrectFormat`) selbst nicht kennt (hier `filename` und auch `inputstring`). Die Operation ruft eine Action ohne Parameter auf. Die Integration kann dadurch innerhalb des Lambda-Bodys frei bestimmen, welche Methoden als nächstes aufgerufen werden. Dadurch schränkt man die möglichen nachfolgenden Methodenaufrufe nicht durch die Operation ein. In Sprachen die dieses Feature nicht unterstützen, macht das die Umsetzung von Flow Design deutlich umständlicher⁴.

5.5. Auf Rückgabewert warten

In C# gibt es neben den Actions Methodenreferenzen, die keine Rückgabewerte erlauben, auch Methodenreferenzen, die einen Rückgabewert erlauben. Diese werden mit `Func<Parameter, ..., Rückgabewert>` deklariert. Eine Methode die eine andere Methode über ein Func Methodenzeiger aufruft, würde zwar das IOSP erfüllen - die Operation würde die andere Funktion nicht kennen - jedoch würde trotzdem eine funktionale Abhängigkeit entstehen. Aus diesem Grund ist das nachfolgende Beispiel nicht Flow Design konform, da es das PoMO verletzt.

```
static List<string> FormatStrings(List<string> lines ,
    Func<string, string> formatFunc )
{
    List<string> result = new List<string>();
    foreach(line in lines)
    {
        string formattedstring = formatFunc(line);
        result.Add(formattedstring)
    }
    return result;
}
```

Listing 5.10: Auf Rückgabewert warten

5.6. Nutzen von IOSP

In diesem Abschnitt wird versucht zu erschließen was der Mehraufwand für die Einhaltung des IOSP in der Praxis für einen Nutzen hat.

⁴Java unterstützt aktuell kein Closure bei Lambda-Ausdrücken

Die Perlenkette

Die Codebasis, die nach IOSP implementiert wurde, soll bildlich gesprochen einer Perlenkette ähneln. Der Code besteht aus aneinandergereihten Funktionseinheiten, die zusammen ein großes Ganzes bilden. Möchte man Änderungen an dem Programm vornehmen, so braucht man nur an einer Stelle die Kette zu öffnen und etwas hinzufügen oder entfernen. Danach schließt man die Kette wieder und das Programm läuft wieder. Beim Einfügen oder Entfernen ist nur darauf zu achten, dass die Eingänge und Ausgänge zueinander passen. Ist das nicht der Fall, so gibt es auch die Möglichkeit eine sog. Adapter-Funktionseinheit zwischenzuschalten, die die inkompatiblen Ein- und Ausgänge zu korrigieren. Eine weitere Option wäre, die betroffenen Funktionseinheiten und deren ein- und ausgehenden Datentypen entsprechend abzuändern und die Funktionseinheiten anzupassen. Die erste Variante bringt möglicherweise einen Performanceverlust mit sich. In vielen Stellen des Codes, ist dies jedoch meistens kein Problem. Falls die Funktionseinheiten an anderer Stelle verwendet werden, oder sie zu einer externen API gehören, ist möglicherweise eine Abänderung nicht einfach umsetzbar. Dann bleibt nur die Option eine Adapter-Funktionseinheit zu verwenden.

Funktionale Abhängigkeiten

Funktionale Abhängigkeiten sind auch in anderen Gebieten außerhalb von der Softwareentwicklung ein Problem. Denn wenn es darum geht produktive Arbeitsabläufe zu gestalten. Wenn jemand oder etwas seine Arbeit erst abschließen kann, wenn ein anderer seine Aufgabe erledigt hat, dann entsteht eine funktionale Abhängigkeit. Solch eine Eigenschaft gilt es wenn möglich zu verhindern. Besser ist es, wenn ein Pool an Aufgaben existiert, von dem sich jeder bedienen kann, diese unabhängig von anderen Einheiten erledigen kann und dann das Ergebnis wieder in ein Pool zurückgibt, von denen sich andere wiederum bedienen können. Flow Design untersagt solche funktionale Abhängigkeiten in Operationen. In der Praxis bewirkt das, dass die Operationen eine klare Aufgabe haben, welches sie leichter zu testen macht. Außerdem bleibt die Codebasis auch mit zunehmender Größe evolvierbar. Das Zusammenspiel der Operationen bleibt leichter zu modifizieren, wodurch die Codebasis an geänderte Anforderungen besser angepasst werden kann.

5.7. Ausnahmen

Generell gilt, wenn eine bewusste Entscheidung getroffen wird an einer Stelle gegen die IOSP oder PoMO Regel zu verstößen, ist das in Ordnung, solange sie gut überlegt ist und die Auswirkungen in dem Fall keinen großen Einfluss haben. Es gibt jedoch bereits einige Fälle, bei denen sich ein Aufheben der Regel als gut herausgestellt hat.

Rekursion

Operationen ist es erlaubt sich selbst aufzurufen und wiederum auf das Ergebnis zu warten. Besteht die Rekursion aus einer Kette an Operationen, so muss die letzte Operation

die erste Operation aufrufen und auf ihr Ergebnis warten.

LINQ / Standard-Library Methoden

Eine Koppelung an Methoden, die die Sprache selbst bereitstellt, hat keine großen negativen Auswirkungen. Würde diese Ausnahme nicht gemacht werden, hätte das zur Folge, dass unnötig viele Methodenreferenzen einer Operation mitgegeben werden müssten. Beispiele von Methoden aus der Standardbibliothek von C# sind: `int.TryParse` , `List<>.Sort`, `Dictionary<>.Insert`.

UI-Logik

Die UI-Logik ist von sich aus sehr state-behaftet und dadurch hat ein konsequentes Einhalten des IOSP und Arbeiten mit Datenflüssen innerhalb des UI-Frameworks oft nur eine Verkomplizierung des Codes zur Folge, ohne die Vorteile von IOSP wirklich zu erhalten. Die Alternative besteht darin, die UI von der Businessdomäne/-logik zu entkoppeln, sodass der Einfluss einer Nichteinhaltung keine Konsequenzen auf die Businessdomäne hat.

Eine Herangehensweise besteht darin, die so genannte *Interaktionen* des UIs zu identifizieren. Diese Interaktionen bilden die Schnittstelle zwischen UI und Businessdomäne. Jede dieser Interaktionen stellt ein eigenes Flow Design dar. Die Interaktionen und nachfolgende Methoden sind deshalb IOSP konform. Das aktuelle Model des UIs wird einer Interaktion übergeben. Dieses Model wird durch ein Flow Design modellierten Datenstrom modifiziert und am Ende an die UI zurückgegeben. Die Aufgabe der UI-Logik besteht dann anschließend darin das Ergebnis der Modifikation des Models darzustellen.

6. Die Entwurfsmethode

Wie in der Einleitung dieser Arbeit bereits erwähnt, ist Flow Design nicht nur eine Methodik um Datenflüsse auf dem Papier zu modellieren, sondern bietet neben dem Diagramm noch eine Entwurfsmethode, um die Architektur einer Software zu entwerfen.

6.1. System-Umwelt-Analyse

Der erste Schritt beim Entwerfen einer Software nach Flow Design besteht darin, die Grenzen des Systems kennenzulernen. Es soll herausgefunden werden, welche Möglichkeiten es gibt, mit dem System zu interagieren und auf welche Ressourcen das System zugreifen soll. Ziel soll sein, dass das System so entworfen wird, dass die Businessdomäne von diesen äußeren Faktoren entkoppelt bleibt. Ein einfaches Schaubild soll einem helfen diese äußeren Faktoren zu identifizieren.

Ein Kreis wird in der Mitte eines Papiers gezeichnet, dieser Kreis symbolisiert das System oder auch Domäne genannt.

Auf der linken Seite des Kreises werden die äußeren Systeme eingetragen, die auf die Domäne zugreifen, diese werden auch als Aktoren bezeichnet. Solche Aktoren greifen ausschließlich über Portale auf die Domäne zu. Die Portale stellen einen Zugriffspunkt auf die Domäne dar und sollen eine Entkoppelung der beiden Systemen gewährleisten. Beispiele für Aktoren wären: HTTP-Zugriff, Interaktion mit der Anwendung durch Kommandozeilenbefehle und GUIs.

Auf der anderen Seite des Kreises werden die externen Ressourcen eingetragen, auf die das System Zugriff haben soll. Diese Ressourcen greifen ebenso wie die Aktoren nur über klar definierte Zugriffspunkte auf die Domäne zu. Diese Zugriffspunkte werden bei Ressourcen als Provider bezeichnet. Beispiele für Ressourcen wären, ein Filesystem oder eine Datenbank.

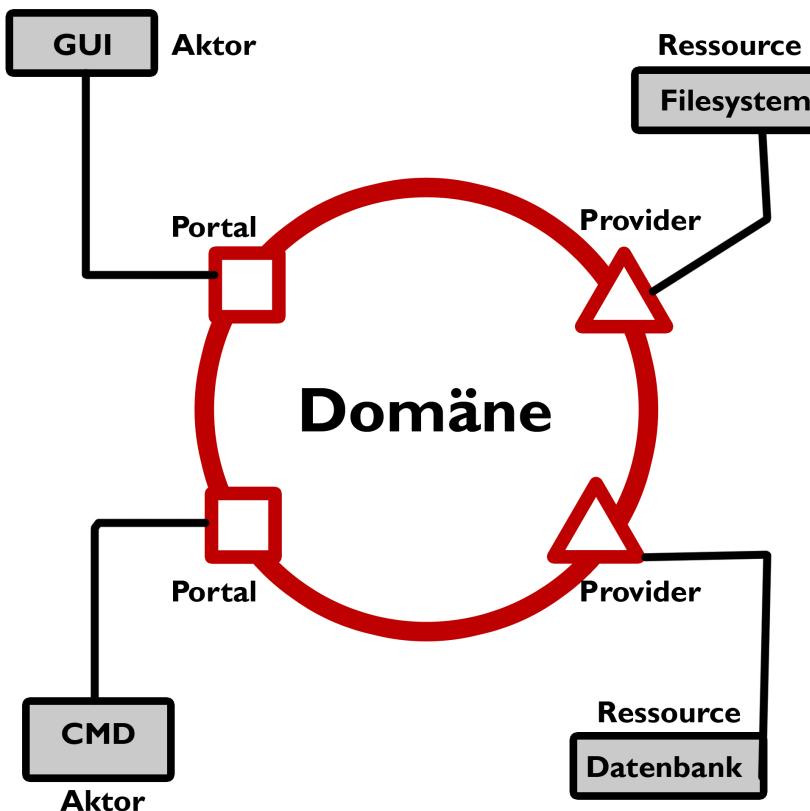


Abbildung 6.1.: System-Umwelt-Analyse

Ralf Westphal bezeichnet diesen Kreis auch als Softwarezelle, was verdeutlichen soll, dass das zu entwerfende System, sich wie eine Zelle in der Natur verhalten soll. Eine Zelle weiß nicht viel über ihre Umwelt, sie befindet sich in einer Nährstofflösung und verarbeitet die Stoffe, die durch ihre Membran in ihr Inneres gelangen. Die Membran einer Softwarezelle trennt diese von ihrer Umwelt (andere Systeme) und regelt den Austausch von Daten mit dem Inneren des Systems. Ziel ist es später in der Implementierung darauf zu achten, dass die „Schicht“ oder „Membran“, zwischen Domäne und Außenwelt möglichst dünn bleibt. Ist das System von der Umwelt entkoppelt, so lässt es sich besser testen und es lassen sich leichter neue Portale und Provider an das System anhängen.

6.2. Interfaceskizze (im Falle einer GUI Anwendung)

In Anlehnung an das User Experience Driven Development, soll schnell die Features des Systems herausgefunden werden, die der Kunde wirklich braucht. Es stellt sich heraus, das UML oder Use Cases für diese Aufgabe nur bedingt hilfreich sind. Leichter für den Kunden zu verstehen ist eine Interfaceskizze. Diese soll gemeinsam mit der Kunden erarbeitet werden. Ein wichtiger Bestandteil dieser Skizze besteht auch darin, die Interaktionen, die der Nutzer mit dieser Oberfläche auslösen kann, zu identifizieren und zu benennen.

6.3. Flow Design Entwurf

Für jede der definierten Interaktionen aus der Interfaceskizze(n) wird ein Flow Design Flussdiagramm erstellt.

Nachdem die Flow Design Diagramme erstellt wurden, muss überlegt werden, wie die einzelnen Funktionseinheiten in Klassen, DLLs und Anwendungen zusammengefasst und strukturiert werden sollen. Dazu bietet sich an, einzelnen Funktionseinheiten zusätzlich mit dem Namen der Klasse (gegebenenfalls auch DLLs und Anwendung) zu beschriften. Alternative ist es auch üblich mehrere Funktionseinheiten mit einer gestrichelten Linien zu umkreisen und auch mit unterschiedlichen Farben zu arbeiten, um die Gruppierungen und Zuordnungen zu verdeutlichen.

Als letzter optionaler Schritt gilt es, Pfeile von solchen Datenströmen farblich hervorzuheben, die parallel laufen können.

6.4. Rekursive Eigenschaft

Eine Softwarezelle hat die Eigenschaft, dass sie rekursiv ist. Das gesamte System lässt sich als eine große Softwarezelle betrachten, die wiederum aus mehreren kleineren Softwarezellen bestehen können (DLLs und Anwendungen). Diese Untersysteme sollen wieder wie das gesamte System möglichst entkoppelt sein von den restlichen Untersystemen. Einer dieser Untersysteme muss sich dann jedoch als integrierendes System verstehen, welches die anderen Untersysteme kennt. Die Aufgabe dieses Untersystems besteht bestenfalls nur daraus, das Zusammenspiel der Untersysteme zu koordinieren und keine komplexe Logik selbst zu implementieren. Diese Untersysteme bestehen dann wieder aus Klassen und Methoden, die nach IOSP implementiert sein sollen. Somit könnte man selbst die Funktionseinheiten eines Flow Design Flussdiagrammes als einzelne Softwarezellen betrachten. Ralf Westphal behauptet, diese Architektur soll weniger starr und somit universeller einsetzbar sein, als zum Beispiel das Schichtenmodell oder das Zwiebelschalenmodell.

Teil II.

Dexel

7. Vision

Im Grunde geht es bei einem Editor für Flow Design vor allem darum die Vorteile aus der digitalen Welt mit der Methodik zu vereinen, ohne die Einfachheit der Methodik auf dem Papier zu verlieren.

7.1. Vorteile eines digitalen Editors

Flow Design ist eigentlich als Entwurfsmethode auf dem Papier gedacht. Jedoch hat ein Flow Design auf dem Papier einige Nachteile, die ein Editor am Computer aufheben könnte. Das wären vor allem folgende Punkte:

Einmal eingezeichnete Elemente lassen sich nicht mehr so leicht verändern.

Während des kreativen Prozesses ein Programmierproblem zu lösen, bedarf es mehrere Iterationen und Veränderungen an dem Diagramm. Eine Möglichkeit Versionen abzuspeichern und Teile zu verschieben, umzubenennen und umzustrukturieren sind klare Vorteile von einem digitalen Editor.

Automatische Einhaltung der Notation

Ein Editor bewirkt eine automatische Einhaltung der Notation.

Automatische Validierungsprozesse

Während der Erstellung des Flow Designs können im Hintergrund Validierungsprozesse beim Erstellen des Diagrammes Hilfestellung bieten. Auto vervollständigungen könnten ebenfalls an einigen Stellen eingebaut werden, um das Tippen zu beschleunigen.

Unnötige Abtipparbeit ersparen.

Aus den Beschriftungen im Flow Design Diagramm lassen sich die Variablennamen und Methoden Signaturen leicht herleiten. Liegt das Diagramm in digitaler Form vor, wäre eine automatische Generierung von Quellcode naheliegend und würde dem Anwender unnötige Abtipparbeit ersparen. Außerdem können schnell mehrere Iterationen eines Diagrammes erstellt und in Versionskontrollsystmen eingepflegt werden. Die Diagramme können auch einfacher von unterschiedlichen Orten aus und von mehreren Personen abgerufen und bearbeitet werden.

Generierung von Methodenrümpfe

Durch die strikten Implementierungsregeln von Integrationen lassen sich für diese Methoden nicht nur die Signaturen, sondern auch die komplette Implementierung aus dem Diagramm ableiten. Eine Generierung dieser Codezeilen wäre ein zusätzlicher Komfortgewinn von einem Editor.

Roundtrip-Engineering

Eine Möglichkeit aus einer bestehenden Codebasis ein Diagramm zu erstellen - wenn auch nur teilweise - würden die Produktivität beim Einsetzen von Flow Design in einem Projekt weiter steigern. Ein Anwendungsfall wäre: Der Anwender möchte Teile seines Quellcodes in ein Flow Design überführen, um dann mit Hilfe des Flow Designs, diesen zu überarbeiten und anschließend neu zu generieren. Möglicherweise würde er die erstellten Codezeilen mit Copy&Paste in sein bestehendes Projekt übertragen.

7.2. Vorteile von einem Entwurf auf dem Papier

Ein Papier schränkt einen nicht ein und erlaubt es schnell und einfach Pfeile und Kreise zu zeichnen, Notizen einzufügen und ist einfach in der Bedienung. Bei dem Erstellen eines Editors muss deshalb ein besonders großes Augenmerk auf eine gute und intuitive Bedienung gelegt werden, damit einem das Programm bei der kreativen Arbeit nicht behindert. Endziel wäre es, dass der Anwender von sich aus lieber zum Editor greift, als zu Stift und Papier, weil ihm der Editor komfortableres und kreatives Arbeiten besser ermöglicht.

8. Anforderungen

Im Folgendem eine Auflistung der Anforderungen in Tabellenformat.

8.1. Editor

Die Anforderungen des Editors wurden aufgeteilt in drei Prioritäten: hoch, mittel und niedrig.

Anforderungen	Priorität
Erstellen von Funktionseinheiten, Benennen, Verschieben auf dem Canvas, Löschen, Duplizieren	hoch
Navigation (Panning, Zooming)	hoch
Selektieren von mehreren Funktionseinheiten um mehrere auf einmal zu bearbeiten	hoch
Definieren von Eingangs- und Ausgangs-Datenströmen, für eine Funktionseinheit	hoch
Verbinden eines Ausgangs einer Funktionseinheit mit einem Eingang einer anderen	hoch
Zusammenlaufen/ Auseinanderlaufen von Datenflüssen (Joined- und Split-Notation)	hoch
Funktionseinheit(en) einer anderen unterordnen können, um Integrationen zu erstellen inklusive visuelle Kennzeichnung	hoch
Speichern und Laden in ein Dateiformat	hoch

Tabelle 8.1.: Anforderungen mit hoher Priorität

Anforderungen	Priorität
Funktionseinheiten Klassen zuordnen	mittel
Syntaxhighlighting für die Datentypen auf den Datenflüssen	mittel
Keyboard Hotkeys / Tabstops	mittel
Automatisches Spacing	mittel
Validierung von Datenströmen	mittel
Untergeordnete Funktionseinheiten einer Integration an einer anderen Stelle definierbar machen, falls Platz knapp wird	mittel
Autosave	mittel
Undo / Redo System	mittel
Definieren von State einer Funktionseinheit	mittel
Definieren von neuen Datentypen	mittel

Tabelle 8.2.: Anforderungen mit mittlerer Priorität

Anforderungen	Priorität
Mouseover zeigt eine Vorschau des erzeugten Codes für die Funktionseinheit	niedrig
Wiederverwenden von vorhandenen Funktionseinheiten	niedrig
Autovervollständigung auf dem Textfeld der Datenströme	niedrig
Kommentarboxen	niedrig
Anfügen von Tests an Funktionseinheiten	niedrig
Mehrere Themes: Dark, Light (Print)	niedrig

Tabelle 8.3.: Anforderungen mit niedriger Priorität

Navigation

Durch Inspiration aus Grafikanwendungen: Panning (Verschieben der Kamera in der X- und Y-Achse) mit Hilfe der mittleren Maustaste. Zoomen in und aus dem Diagramm durch das Mausrad. Die Position des Mauszeigers bestimmt das Zentrum des Zooms.

8.2. Generierung von Code

Im folgenden eine Auflistung der Anforderungen einer Code-Generierung aus einem Flow-Design-Diagramm.

Anforderungen	Priorität
Generierung von Methodensignaturen anhand der ein- und ausgehenden Datenflüssen einer Funktionseinheit	hoch
Erzeugen des kompletten Methodenrumpfes einer Integration	hoch
Erzeugung von Klassen der benutzerdefinierten Datentypen	hoch
Einstellungen dem Benutzer zugänglich machen, um die Generierung zu konfigurieren	mittel
Erzeugung von Namespaces und Auflösung von Usings	niedrig
Korrekte Einfügen / Integrieren von den generierten Codezeilen in die Codebasis eines bestehendes Softwareprojektes	niedrig
Live-Generierung	niedrig

Tabelle 8.4.: Anforderungen der Code-Generierung

Erzeugung des kompletten Methodenrumpfes einer Integration

Hierbei muss erkannt werden, in welcher Reihenfolge die Methoden aufgerufen werden müssen, lokale Variablen erzeugt werden müssen und was einer Methode als Parameter übergeben werden muss. Dabei kommen `IEnumerables` und `Lambdas` zum Einsatz um Datenflüsse zu implementieren.

Einstellungen für die Generierung dem Benutzer zugänglich machen

Mögliche Optionen wären:

- wie das Programm den Methodenrumpf einer Operation standardmäßig befüllen soll: Leer, mit `NotImplementedException` oder mit einem Return-Ausdruck eines Standardwertes abhängig von der Methodensignatur.
- Ob innerhalb einer Integration der Rückgabewert einer Funktion erst in eine lokale Variable gespeichert werden soll, oder direkt der Methodenaufruf an die andere Methode weitergereicht wird. Beziehungsweise die Regel konfigurierbar machen: Ab welcher Zeilenlänge, wie die Variablen benannt werden sollen, etc.

Einfügen von generierten Codezeilen in bestehende Codebasis

Notwendig hierfür wäre, dass bestehende Klassen gefunden werden müssten, Usings korrekt eingefügt und schlussendlich die generierten Methoden und Datentypen in die jeweiligen Klassen/Dateien eingefügt werden. Dabei muss die Syntax berücksichtigt werden und möglicherweise Zugriffsberechtigungen erkannt und bei Problemen einen Dialog zur Korrektur dem Anwender anbieten.

Eine andere Option wäre es, dies Codezeilen einfach einzufügen und die Erkennung und Lösung der Probleme der IDE zu überlassen. Gerade bei C# gibt es mit Resharper viele Refactorisierungs-Tools, die einem bei der Lösung solcher Probleme unterstützen.

8.3. Generierung von Flow Design Diagrammen aus Code

Anforderungen	Priorität
Finden von Methoden und Erzeugen von Funktionseinheiten und ihre Datenströme anhand der Methodensignatur im Code	hoch
Automatisches Spacing	hoch (aber nicht unbedingt perfekt)
Den Datenfluss einer Integration erkennen und ihn in ein Flow Design Diagramm übertragen	hoch
Erkennen, ob es sich bei der Methode um eine Operation oder Integration handelt	hoch
Umgang mit Methoden die nicht das IOSP befolgen	mittel
Speichern der Inhalte, die nicht im Diagramm dargestellt werden können	mittel

Tabelle 8.5.: Anforderungen der Diagramm-Generierung

Automatische Anordnung

Unbedingt notwendig, auch wenn es nur sehr rudimentär umgesetzt wird, ansonsten liegen alle Funktionseinheiten nach dem Erstellen unübersichtlich auf einem Punkt aufeinander. Falls das Automatische Spacing an manchen Stellen nicht perfekt funktionieren sollte, kann eine gute Usability (Selektierungs- und Verschiebungsfeatures) hier dieser Imperfektion leichter verschmerzbar machen.

Schwierigkeiten

Bei Verwendung von Events kann der Datenfluss möglicherweise nicht mehr nachvollzogen werden, da die „Verdrahtung“ an beliebiger Stelle stattfinden kann und auch dynamisch zur Laufzeit.

9. GUI Skizzen / Usabilityüberlegungen

9.1. Minimalistischer Aufbau. Fokus auf Produktivität.

Die Anwendung soll möglichst viel Platz für die Zeichenfläche bieten.

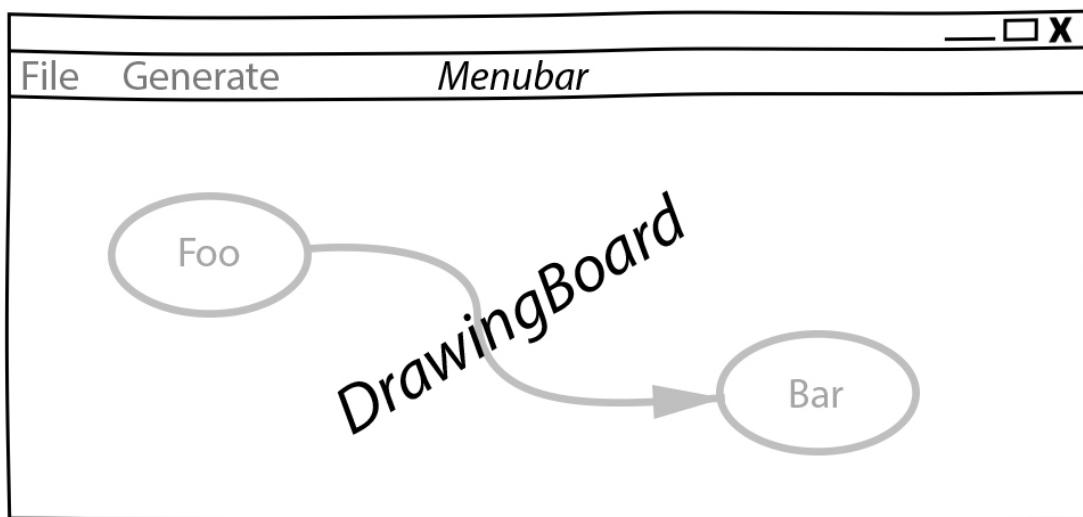


Abbildung 9.1.: Die Hauptansicht

Bei einem Rechtsklick auf eine leere Stelle in der Zeichenfläche erscheint ein Kontextmenu, dass dem Anwender erlaubt an dieser Stelle eine neue Funktionseinheit einzufügen.

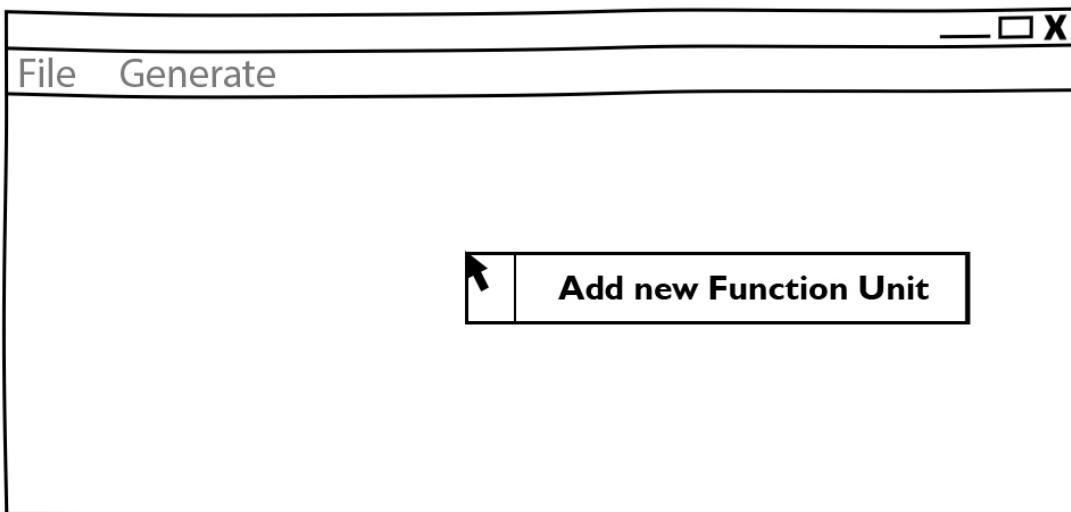


Abbildung 9.2.: Ablauf einer Erstellung einer neuen Funktionseinheit

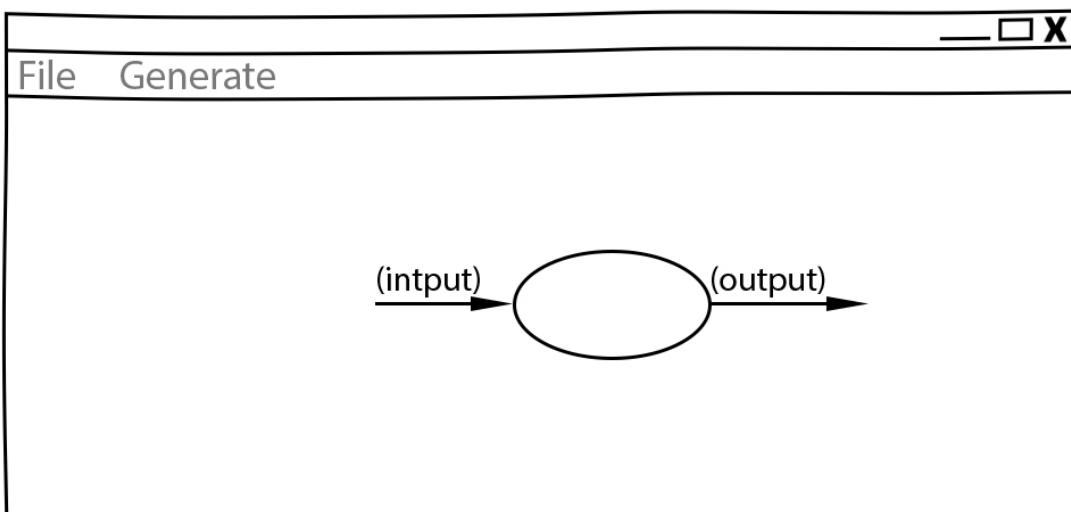


Abbildung 9.3.: Eine neue Funktionseinheit wurde erstellt

Im folgendem einige Kerngedanken über die Funktionalität des Editors:

- Keine unnötigen Menüleisten, Symbolleisten, etc. besser kontextsensitive Kontextmenüs, oder Tastenkürzel, damit die Strecke, die die Maus bewegt werden muss, gering gehalten wird.
- Tabulatorstopps einbauen, damit schnell zwischen den Textfeldern, entlang des Graphen, gesprungen werden kann.
- Verwendung von Drag&Drop, um eine intuitive Bedienung zum Verknüpfen von Funktionseinheiten zu ermöglichen. Die Flächen, die per Drag&Drop zu bedienen sind, sollen über ein Mouseover Feedback erkennbar sein. Außerdem sollen die Flächen nicht zu klein sein, damit ein leichtes Treffen des Feldes sichergestellt wird. Möglicherweise können auch unsichtbare Flächen verwendet werden,

um eine Drag&Drop Fläche künstlich leicht zu vergrößern und einfacher treffbar zu machen.

- Rectangle-Selection in Kombination mit Modifier-Keys um mehrere Funktionseinheiten schnell und komfortabel zu selektieren.
- Shift + Drag : Schnelles Duplizieren der selektierten Elemente.¹

Anwendungsfälle: Der Anwender möchte schnell ein gesamtes Diagramm duplizieren und an eine andere Stelle schieben, um dort eine weitere Iteration davon zu erstellen. Ein anderer Anwendungsfall von Duplizieren ist, dass der Anwender eine vorhandene Funktionseinheit an einer anderen Stelle im Diagramm verwendet möchte.

- Ctrl + Drag einer Funktionseinheit: Die Funktionseinheit und alle ihre Kinder werden verschoben. Anwendungsfall ist: Der Anwender möchte etwas Platz schaffen zwischen zwei Funktionseinheiten. Mit einem Ctrl+ Drag der zweiten Funktionseinheit, kann er diese und alle nachkommenden Funktionseinheiten verschieben, ohne sie vorher extra selektieren zu müssen.

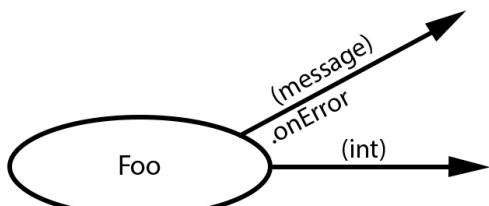
9.2. Textfelder

Textfelder müssen waagerecht bleiben. Auf dem Papier schreibt man die Daten auf die Pfeile, somit wird Text auf einem schrägen Pfeil auch entlang des Striches geschrieben. Am Computer ist so etwas schlecht umzusetzen. Man kann Textfelder bei WPF drehen, dadurch entsteht jedoch eine ungewohnte Bedienung beim Markieren von Text. Ein Drehen beim Fokussieren/Defokussieren wäre auch möglich, damit wäre jedoch eine zusätzlicher Klick nötig, falls man Text markieren möchte: Ein Mausklick zum Fokussieren/Drehen der Textbox und ein weiterer um Text zu markieren / den Cursor zu platzieren. Die beste Lösung wäre aus Usability-Sicht, wenn Textfelder nicht gedreht werden, sondern immer waagerecht dargestellt werden. Somit muss hier die Notation an manchen Stellen etwas vom original Abweichen.

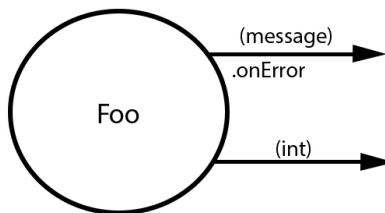
- Mehrere Ausgänge
- Pfeile zwischen zwei Funktionseinheiten, die auf unterschiedlichen Höhen platziert sind.

¹Vorbild dieser Funktion ist die CAD-Anwendung 3ds Max, das dieses Bedienkonzept an vielen Stellen einsetzt. Einmal daran gewöhnt, möchte man es nicht mehr missen.

Mehrere Ausgänge

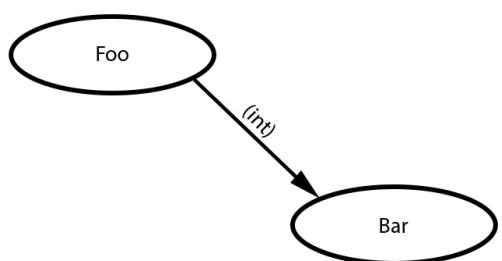


Orginal Notation

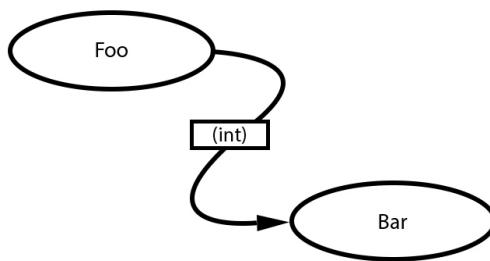


Angepasste Notation

Pfeile zwischen zwei Funktionseinheiten, die auf unterschiedlichen Höhen platziert sind



Orginal Notation



Angepasste Notation

9.3. Datentypen - Definition und Organisation

Da Flow Design mit Datenströmen arbeitet, ist das Definieren neuer Datentypen ein wesentlicher Bestandteil davon. Eine Möglichkeit wäre es, wie auf dem Papier, es zu erlauben an beliebigen Stellen im Diagramm eine Box zu erstellen, in der der Anwender einen neuen Datentyp benennen und seine Felder definieren kann. Vorteil davon wäre, dass der Anwender die nötige Information in der Nähe des Datenstroms schnell ersichtlich platzieren kann, wo die Daten auch vorkommen.

Nachteil wäre, dass der Algorithmus zum automatischen Spacing komplizierter werden würde, da nun auch eine sinnvolle Platzierung der Datentypen mit berücksichtigt werden müsste. Ein weiteres Problem dieser Lösung taucht auf, sobald der Anwender an unterschiedlichen Positionen im Diagramm den selben Datentypen verwendet. In diesem Fall müssten Doppelungen erlaubt sein, oder der Anwender würde an einer Stelle nicht die Information haben, worum es sich bei einem Datentyp handelt.

Eine andere Option wäre es, die Datentypen nicht auf dem DrawingBoard zu platzieren, sondern separate vom Flow Design getrennt in einem extra GUI-Element darzustellen und dort die Definition eigener Datentypen zu ermöglichen. Dieses GUI-Element würde in Form einer Liste alle vorhanden Datentypen beinhalten. Zusätzliche Usability-Features wären, das Typen, die im Diagramm vorkommen, jedoch nicht zu den Basistypen der Sprache gehören und noch nicht in der Anwendung definiert wurden, erkannt

und speziell hervorgehoben werden und den Anwender subtil auffordert diesen zu definieren.

Um den Vorteil einer Box innerhalb des Diagrammes etwas zu entkräften, könnten die Einträge in der Liste kontextsensitiv sein: Wenn der Anwender in ein Textfeld eines Datenstromes klickt, könnte die Liste nur jene Datentypen anzeigen, die in dem Textfeld vorhanden sind. Beim klick auf eine leere Fläche (defokussieren des Textfeldes) würden wieder alle Datentypen im gesamten Diagramm angezeigt werden. Des weiteren wäre eine visuelle Hervorhebung von nicht verwendeten Datentypen auch denkbar.



Abbildung 9.4.: Datentypen Editor

Weitere Ideen:

- Mouse-Hover über ein Datentype im Diagramm zeigt die Definition in einem Pop-Up über dem Mauszeiger an.
- Drag&Drop von Datentypen aus der Liste in das *DrawingBoard* zu ermöglichen, falls der Anwender für einen Screenshot - oder aus einem anderen Grund - diese Information im Bild haben möchte.

9.4. Darstellung von Joined Inputs & Split Outputs

Datenströme können aus verschiedenen Quellen stammen und an einer Funktionseinheit zusammenlaufen oder aus einer Quelle an viele andere Funktionseinheiten weitergereicht werden. Flow Design bietet hierfür die Pipe-Notation, oder die sog. Joined Inputs und Split Outputs an.

Vorteile der Pipe-Notation:

- Einfacher zu realisieren auf GUI Seite (Automatisches Spacing aufgrund der geringeren Anzahl an Pfeilen einfacher umzusetzen)

- Pfeile müssen seltener große Distanzen überbrücken, was das Diagramm weniger chaotisch wirken lässt

Nachteile der Pipe-Notation / Vorteile der Joined Inputs & Split Outputs:

- Datenströme sind möglicherweise nicht mehr eindeutig zu interpretieren. Bei der Verwendung von Joined Inputs ist die Herkunft eines Datenstroms eindeutig ersichtlich. Bei der Pipe-Notation kann man diese Problem durch eine Benennung der Daten auf den Datenströmen lösen. Diese Erkenntnis legt eine Validierung - einschließlich visuellem Feedback - der Datenströme auf eine eindeutige Interpretation nahe.

Da beide Notationen ihre Vor- und Nachteile haben, soll die Anwendung beide Darstellungen unterstützen.

9.5. Validierung des Datenflusses

Der Validierungsprozess soll subtil sein. Ein Blockieren beim Verbinden zweier Funktionseinheiten soll nicht geschehen. Diese würde sonst dem Ziel entgegen stehen, eine möglichst freie Gestaltung, wie beim Zeichnen auf dem Papier, zu gewährleisten. Der Anwender soll die Freiheit haben, nicht valide Verbindungen zu erstellen, die er möglicherweise erst nach dem Verbinden dann entsprechend anpasst. Eine dezente farbliche Hervorhebung soll als Feedback des Validierungsprozesses (möglicherweise indem man den Pfeil einfärbt) ausreichen. Mögliche Validierungsfehler wären:

- Pipe-Notation: Überschneidung von Datentypen.
- Fehlende Daten: Nicht alle vom Eingang der Funktionseinheit verlangten Daten sind im Datenfluss enthalten.

Im Grunde wäre jedoch auch eine Generierung von jeglichen Flow Design Diagrammen möglich, würde man folgende Regeln einführen:

Der Graph wird zurück gelaufen, bis ein passender Datentyp gefunden wird (das erste Vorkommen wird genommen). Falls der Datentyp nicht gefunden wird, wird er in der Integration als lokale Variable deklariert und mit einem Standardwert initialisiert.

9.6. Validierung der Syntax

Die Notation der Daten der Datenflüssen besteht aus einer einfachen Syntax. Diese muss zwingend eingehalten werden, damit eine Generierung des Codes möglich ist. Eine rote gewellte Linie unterhalb des nicht validen Textes soll dem Anwender anzeigen, dass ein Syntaxfehler vorliegt.

10. Realisierung

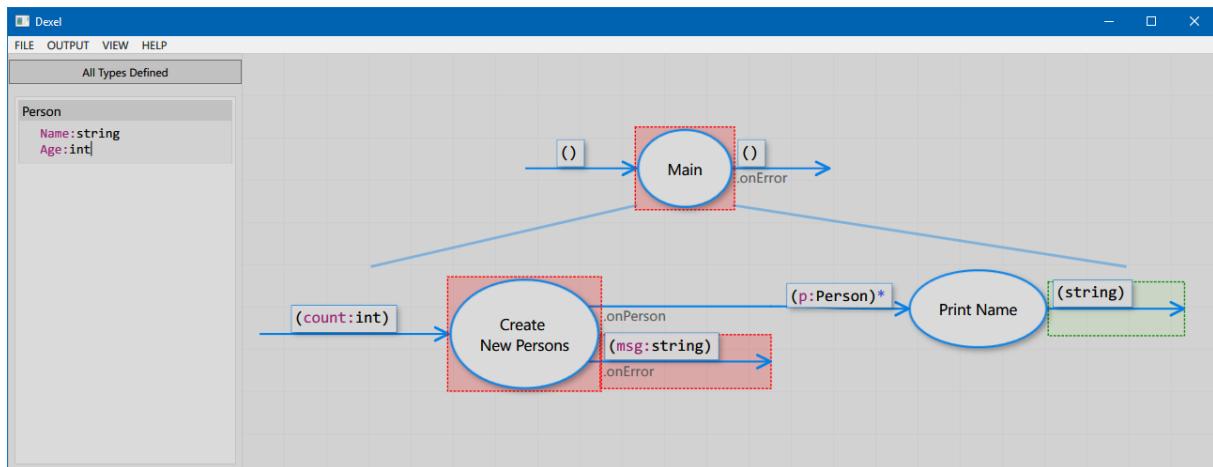


Abbildung 10.1.: Dexel

In diesem Kapitel soll vorgestellt werden, was in dem Zeitraum dieser Bachelorarbeit erreicht wurde. Die Architektur der Anwendung wurde selbst nach Flow Design umgesetzt und bietet somit dem Leser eine Quelle an weiteren konkreten Beispielen, wie eine Methode nach IOSP in der Praxis aussieht.

Als Arbeitstitel für die Anwendung wurde der Name Dexel gewählt.

10.1. Übersicht über die unterschiedlichen Projekte

Die Anwendung besteht aus einer *Solution*, die folgende Unterprojekte beinhaltet¹:

¹Bei komplexen Funktionalitäten wurde nach TDD (Test Driven Development) programmiert, bei dem zuerst der Test geschrieben wird, der das zu erwartende Ergebnis definiert, und erst anschließend die Methode implementiert wird. Diese Tests wurden in einem extra Test-Projekt für jedes Projekt zusammengefasst.

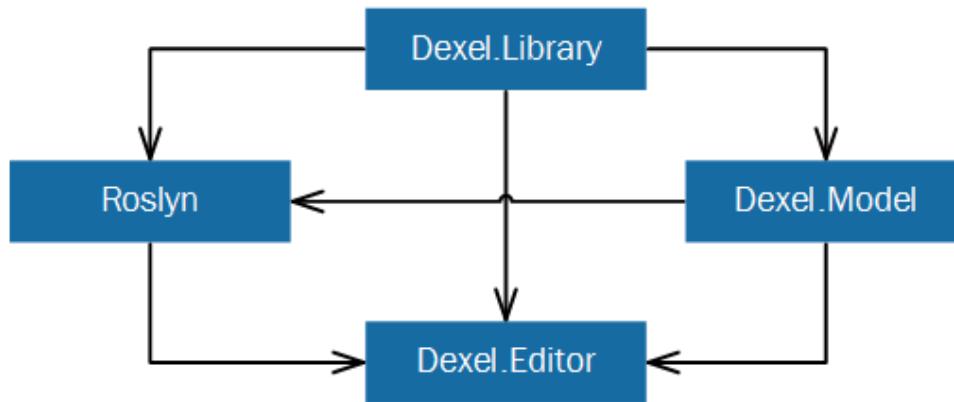


Abbildung 10.2.: Aufbau der Assembly-Struktur und Abhängigkeiten zwischen den Dexel-Unterprojekten

Dexel.Model / Dexel.Model.Tests

Dieses Projekt beinhaltet das Domänenmodell - alle Datentypen die zur internen Repräsentation eines Flow Design Diagramms nötig sind. Außerdem beinhaltet dieses Projekt statische Manager-Klassen, die das Arbeiten mit den Datentypen vereinfachen.

Dexel.Editor / Dexel.Editor.Tests

Dies ist das Hauptprojekt, das alle anderen Projekte integriert. Nach Flow Design kann man IOSP auf Projekt-Ebene anwenden, hier wurde jedoch das IOSP Prinzip nicht eingehalten. Dieses Projekt hat nicht nur die Aufgabe die anderen Projekte zu integrieren, sondern beinhaltet selbst den UI-Sourcecode. Diese Entscheidung wurde gefällt, da ein Extrahieren der UI in ein anderes Projekt sich als zu schwierig erwies. Eine Lösung dafür zu finden, wie ein Event aus dem UI mit einer Funktionalität verbunden werden könnte, dass sich in einem Projekt befindet, war zeitlich zu aufwendig. Aus diesem Grund wurde entschieden auf die zusätzliche Komplexität einer Entkopplung zu verzichten und es einfacher zu halten, indem dieses Projekt sowohl die UI beinhaltet, als auch alle anderen Projekte kennt.

Roslyn / Roslyn.Tests

Diese Projekt beinhaltet die Logik zur Erstellung von C#-Code aus einem Flow Design. Microsoft Roslyn ist die Technik, die hier zum Erstellen von Code zum Einsatz kommt. Das Flow Design Diagramm muss in Form eines MainModels aus dem Dexel.Model Projekt vorliegen. Aus diesem Grund gibt es eine Abhängigkeit zwischen diesem und dem Dexel.Model Projekt.

Eine Trennung dieser Abhängigkeit wurde versucht umzusetzen indem gegen ein Interface programmiert wurde, anstatt gegen die konkrete Implementation in Dexel.Model. Diese Interfaces der Datentypen und Manager-Klassen wurde dann in ein Contracts-Projekt ausgelagert, von dem alle anderen Projekte abhängig sein durften. Die zusätzliche Komplexität, Mehraufwand und ein Problem bei der Serialisierung von Datentypen,

die Interfaces als Eigenschaften besaßen, machte es nicht wert hier eine Entkoppelung zu schaffen. Somit wurde beschlossen die Änderung rückgängig zu machen und auf die Interfaces zu verzichten.

Dexel.Library

Dieses Projekt beinhaltet einige Methoden (meist Extension-Methoden), die ein Arbeiten nach Flow Design erleichtern. Diese wurden in ein separates Projekt ausgelagert, damit alle anderen Projekte darauf zugreifen können.

10.2. Das Domänenmodell

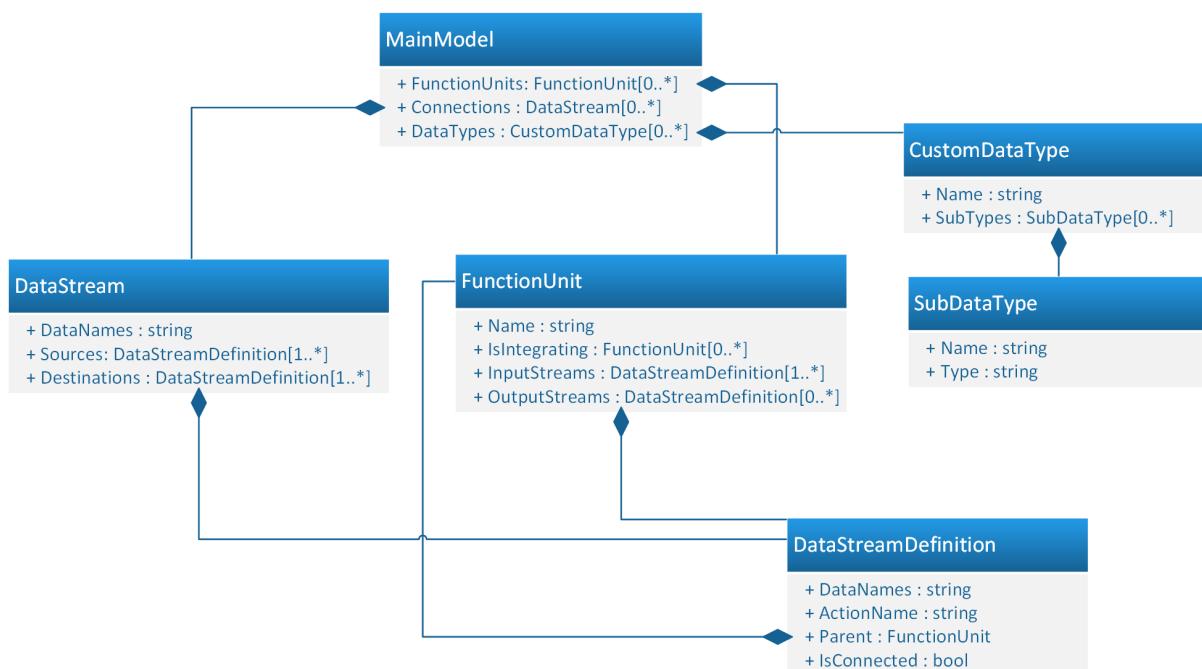


Abbildung 10.3.: Das Domänenmodell von Dexel

Um nachfolgende Kapitel der Realisierung und die darin enthaltenden Codeausschnitte verstehen zu können, braucht es ein Verständnis darüber, wie die grundlegenden Datentypen des Domänenmodells aufgebaut sind. Um die Funktionalität der jeweiligen Datentypen besser zu veranschaulichen, werden in diesem Kapitel Bilder verwendet, welche die spätere Darstellung des Datentyps in der UI zeigen.

MainModel

Das **MainModel** ist, wie der Name schon sagt, das Haupt-Modell, das alle anderen Modelle per Komposition beinhaltet. Somit kann einer Methode einfach eine Objektinstanz eines **MainModel**s übergeben werden und erhält dadurch alle Informationen über das aktuelle Flow-Design-Diagramm. Das **MainModel** beinhaltet alle Funktionseinheiten, alle

verbindenden Datenströme (*Connections*) und alle benutzerdefinierten Datentypen (*DataTypes*).

FunctionUnit

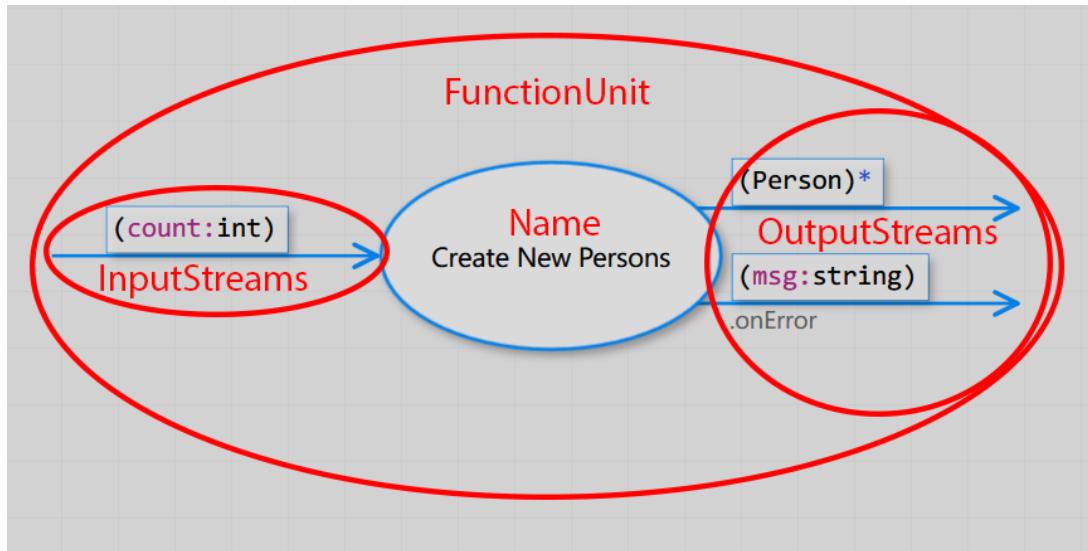


Abbildung 10.4.: Dexel-Screenshot: FunctionUnit-View

Die Name-Eigenschaft ist der Name der Funktionseinheit (der Text, der bei der Darstellung später innerhalb des Kreises erscheint). Dieser wird später per *Binding* direkt an die UI gebunden.

Die Is Integrating-Eigenschaft gibt an, ob diese Funktionseinheit eine Integration ist und welche andere Instanzen von Funktionseinheiten sie integriert. Eine leere Liste besagt, dass die Funktionseinheit keine Integration ist.

Die Input- und OutputStreams definieren die möglichen ein- und ausgehenden Datenströme der Funktionseinheiten. An diese DataStreamDefinitionen können sich DataStreams verbinden. Dadurch lassen sich die Datenflüsse modellieren.

DataStreamDefinition

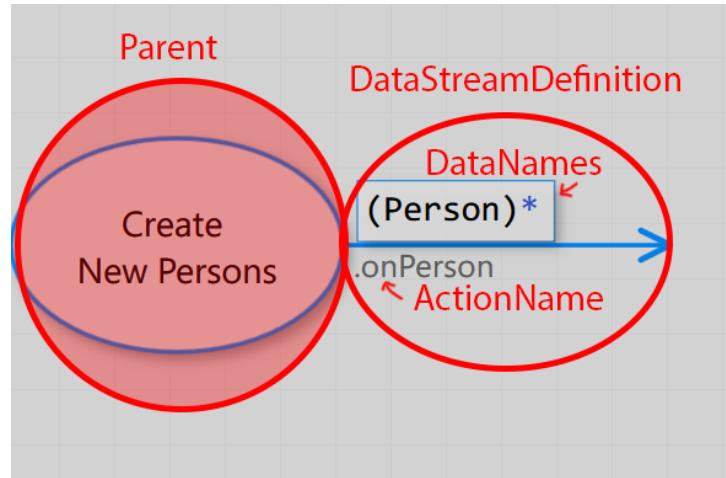


Abbildung 10.5.: Dexel-Screenshot: DataStreamDefinition-View

Eine Funktionseinheit verfügt über ein oder mehrere ein- und ausgehende DataStreamDefinitionen. Diese können verbunden sein oder nicht. Wenn eine Verbindung erstellt und gelöscht wird, muss deshalb auch die Connected-Eigenschaft immer angepasst werden, damit das Modell valide bleibt. Ob eine DataStreamDefinition verbunden ist, oder nicht, ist später für die Darstellung relevant. Eine DataStreamDefinition kennt auch die Funktionseinheit, von der sie ein Ein- oder Ausgang ist.

Eine weitere grundlegende Eigenschaft ist die Benennung der Daten, die auf dem Datenfluss fließen. Für Ausgänge ist auch die Angabe eines Names manchmal nötig. Dieser soll später unterhalb des Pfeiles dargestellt werden.

DataStream (Connections)

Um Datenflüsse zwischen Funktionseinheiten zu beschreiben, bedarf es einer Verbindungsklasse. Die DataStream-Klasse stellt diese Verbindungsklasse dar.

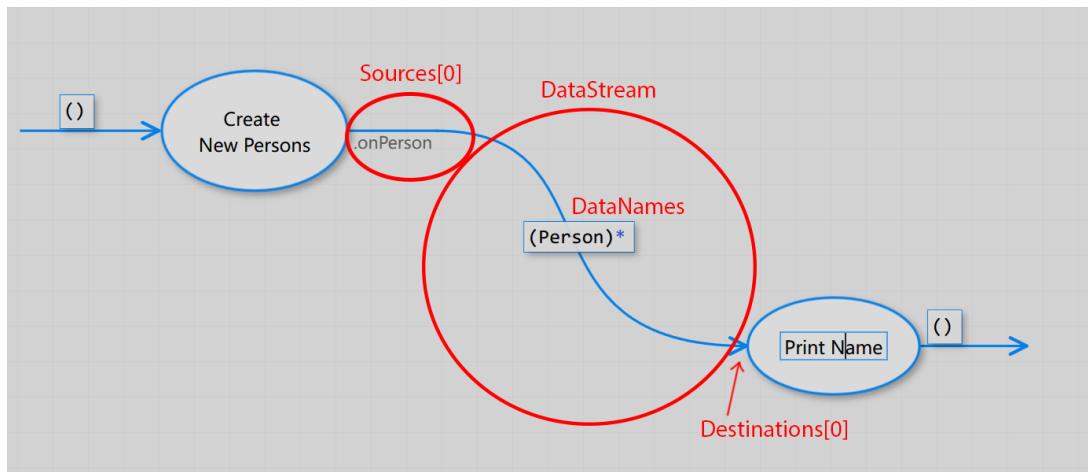


Abbildung 10.6.: Dexel-Screenshot: DataStream-View

Ein DataStream hat ein oder mehrere Referenzen an DataStreamDefinition als Quellen und ein oder mehrere Referenzen an DataStreamDefinition als Ziele. Um ein Datenstrom zu beschreiben, der aus mehreren Quellen Daten bezieht und an einer Stelle zusammenläuft, wird ein DataStream benötigt, der mehrere Einträge in der Source-Liste besitzt und ein Eintrag in der Destination-Liste. Dieser Datenstrom wäre dann ein Joined Input. Ein Datenstrom mit einer Quelle und mehreren Zielen wäre ein sog. Split.

Die DataNames-Eigenschaft beinhaltet den Text, der später in der Mitte des Pfeiles dargestellt werden soll. Eine Änderung dieser Eigenschaft bedarf einer Aktualisierung der DataNames aller Sources und Destinations. Die Aktualisierungsmethode muss die optionalen Pipe-Notation kennen und entsprechend dieser die Ein- und Ausgänge aktualisieren.

Der aktuelle Stand der UI kann nur Datenflüsse mit einer Quelle und einem Ziel darstellen.

DataType

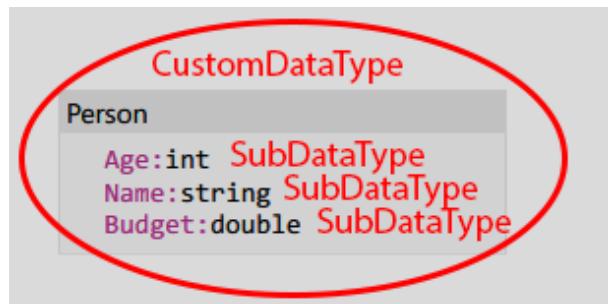


Abbildung 10.7.: Dexel-Screenshot: CustomDataType-View

Ein benutzerdefinierter Datentyp besteht aus einem Namen und einer Liste von mehreren SubDataType-Objekten.

Ein SubDataType besteht aus einem Namen und den Namen des Typen (zum Beispiel string, int oder auch ein anderer benutzerdefinierten Datentyp).

Manager-Klassen

1. MainModelManager

Einer der relevantesten Manager-Klassen ist die MainModelManager-Klasse, diese stellt die wichtigsten Funktionalitäten zur Verfügung die mit dem Arbeiten des MainModels gebraucht werden. Einige dieser Funktionalitäten wären: Verbinden und Trennen von Funktionseinheiten, vorwärts und rückwärts Traversieren entlang des Graphen, Hinzufügen und Entfernen einer Funktionseinheit von einer Integration, Hinzufügen, Löschen und Duplizieren von Funktionseinheiten, oder Teile des Graphen.

2. DataStreamManager

Diese statische Klasse bietet einige Funktionalitäten, die das Arbeiten mit Objekten der DataStream-Klasse vereinfachen soll.

Ein Beispiel hierfür wäre das Ändern der DataNames-Eigenschaft eines DataStreams. Wie bereits im letzten Abschnitt erwähnt, muss beim Ändern der Daten eines Datenflusses auch die Daten seiner Sources und Destinations angepasst werden.

Um dies nochmal zu verdeutlichen, zwei konkrete Beispiele: Falls der Datenfluss auf **(int)** | **(string)** geändert wurde, so muss die DataNames-Eigenschaft der Source-DataStreamDefinition auf **(int)** gesetzt werden und die der Destination-DataStreamDefinition auf **(string)**. Falls der Datenfluss auf **(double)** geändert wurde, so müssen Quelle- und Ziel-Daten auf **(double)** gesetzt werden.²

```
public static void ChangeDatanames(DataStream datastream,
    string newDatanames)
{
    // update datanames of connection itself
    datastream.DataNames = newDatanames;

    // update datanames of DSDs
    TrySolveWithPipeNotation(newDatanames,
        onSuccess: (outputPart, inputPart) =>
    {
        datastream.Sources.First().DataNames = outputPart;
        datastream.Destinations.First().DataNames =
            inputPart;
    },
    onNoSuccess: () =>
    {
        datastream.Sources.First().DataNames =
            newDatanames.Trim();
        datastream.Destinations.First().DataNames =
            newDatanames.Trim();
    });
}
```

Listing 10.1: ChangeDatanames-Methode

²Da aktuell nur DataStreams mit einer Quelle und einem Ziel im Editor unterstützt werden, wurde aktuell auch nur dieses Szenario implementiert. Ändert sich diese Einschränkung müsste man sich Gedanken darüber machen, was in diesen Fällen zu tun wäre. Ein Option wäre, die Pipe-Notation in diesen Fällen zu verbieten. Die UI würde die Daten der DataStreamDefinitionen direkt anzeigen und der Benutzer würde diese dann direkt ändern. Der Datenstrom selbst würde dann kein eigenes Textfeld besitzen und die DataNames-Eigenschaft hätte in diesen Fall keine Bedeutung. Vielleicht wäre es dann auch besser im Modell separate Klassen anzulegen für Split Outputs und Joined Input. Dadurch hätte die einfache DataStream-Klasse dann anstatt einer Liste von DatastreamDefinitionen nur noch eine als Quelle und eine als Ziel.

10.3. Der Editor

10.3.1. Vorstellung was erreicht wurde

Die grundlegenden Basisfunktionen aus dem Anforderungs-Kapitel wurden größtenteils implementiert. Hierbei kam WPF als GUI-Framework zum Einsatz. Im folgendem einige Bilder und Beschreibungen der GUI und Interaktionen.

Erstellen von Funktionseinheiten, Verschieben, Benennen, Selektieren

Der Zeichenbereich wurde implementiert, auf dem der Benutzer über *Rechtsklick -> Add New Function Unit* eine erste Funktionseinheit erzeugen kann. Per Drag&Drop kann er diese innerhalb des Zeichenbereichs frei verschieben. Mit einer Rectangle-Selektion ist es möglich mehrere Funktionseinheiten zu selektieren. Selektierte Funktionseinheiten können gemeinsam verschoben, dupliziert und gelöscht werden.



Abbildung 10.8.: Erstellen einer neuen Funktionseinheit

Nach dem Erstellen einer neuen Funktionseinheit, wird diese automatisch selektiert und der Tastaturfokus wird in das Textfeld des Kreises gesetzt. Dadurch lässt sich diese benennen.

Beim Klicken auf den äußeren Teil des Kreises wird die Funktionseinheit selektiert. Beim Klicken auf das Textfeld, wechselt der Tastaturfokus auf das Textfeld (ein Mouseover zeigt auch an, über welchen Teil des Kreises die Maus sich gerade befindet, auch ein Klicken und Ziehen um sofort ein Teil des Textes zu markieren bleibt damit dem Benutzer weiterhin möglich, so wie er es von einem Textfeld gewohnt ist).



Abbildung 10.9.: Textfeld innerhalb der Funktionseinheit

Erstellen/Löschen/Manipulieren von Inputs und Outputs

Eine Funktionseinheit wird standardmäßig mit einem Eingang und einem Ausgang erstellt. Die Daten, die aus diesen hinein- oder herausfließen, können auf dem Textfeld eingetragen werden. Dabei bietet diese Textfeld ein einfaches Syntaxhighlighting.

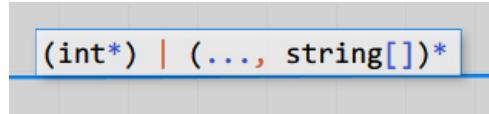


Abbildung 10.10.: Syntaxhighlighting

Eine Funktionseinheit kann mehr als nur einen Ausgang besitzen. Das Hinzufügen eines neuen Ausgangs ist über *Rechtsklick -> Add New Output* möglich. Das Löschen eines Ausgangs ist ebenso möglich.

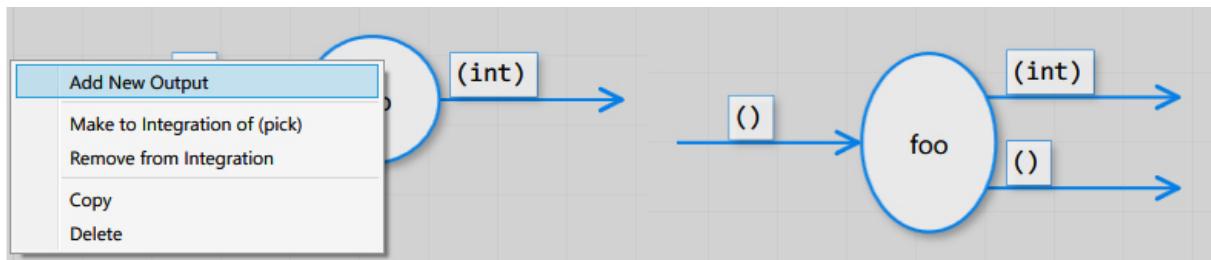


Abbildung 10.11.: Hinzufügen eines neuen Outputs

Die Reihenfolge bei mehreren Ausgängen kann per Drag&Drop verändert werden.

Der Name des Ausgangs kann unterhalb des Pfeiles eingetragen werden.

Verknüpfen und Trennen von Funktionseinheiten über Drag&Drop

Wird das Ende eines Pfeiles eines Ausgangs auf eine andere Funktionseinheit gedropt, so werden beide miteinander verbunden.

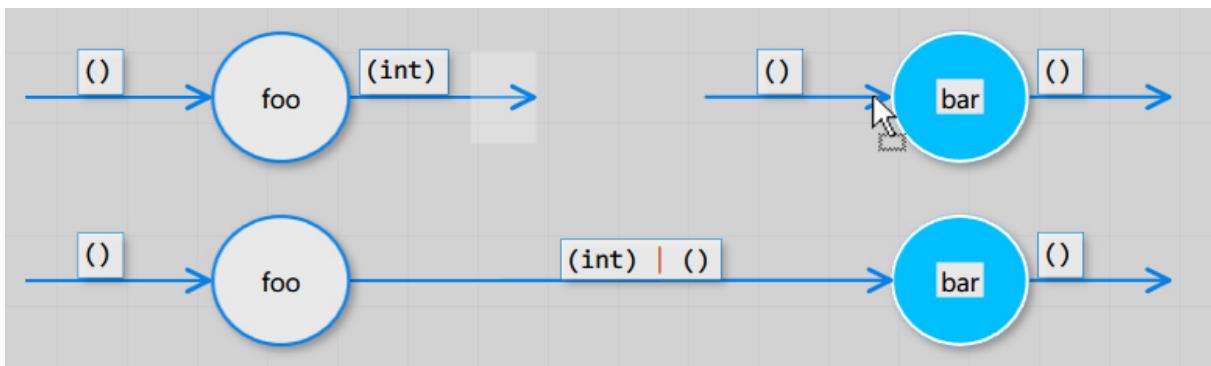


Abbildung 10.12.: Verbinden von Funktionseinheiten

Die Datennamen des Flusses werden bei nicht Übereinstimmung dieser mit Hilfe der Pipe-Notation in das Textfeld des nun neu erzeugten Datenstromes eingetragen. Stimmen beide überein, wird auf die Pipe-Notation verzichtet.

Werden die Daten eines verbundenen Datenflusses geändert, werden die Eingänge und Ausgänge mit Berücksichtigung auf die Pipe-Notation angepasst, sodass beim Trennen der beiden Funktionseinheiten, die Änderungen erhalten bleiben.

Durch Drag and Drop eines verbundenen Pfeiles auf eine leere Stelle des Zeichenbereichs kann eine Verbindung getrennt werden. Wird sie auf eine andere Funktionseinheit gedropt, so wird diese als neues Ziel gesetzt und die Verbindung zur vorherigen Funktionseinheit gelöscht.

Erstellen von Integrationen

Durch Rechtsklick auf eine Funktionseinheit kann im Kontextmenü der Eintrag *Make to Integration of (Pick)* ausgewählt werden. Der Cursor verändert sich, wird nun eine andere Funktionseinheit angeklickt, so wird der komplette Flow, von der diese Funktionseinheit Teil ist, der Integration untergeordnet.

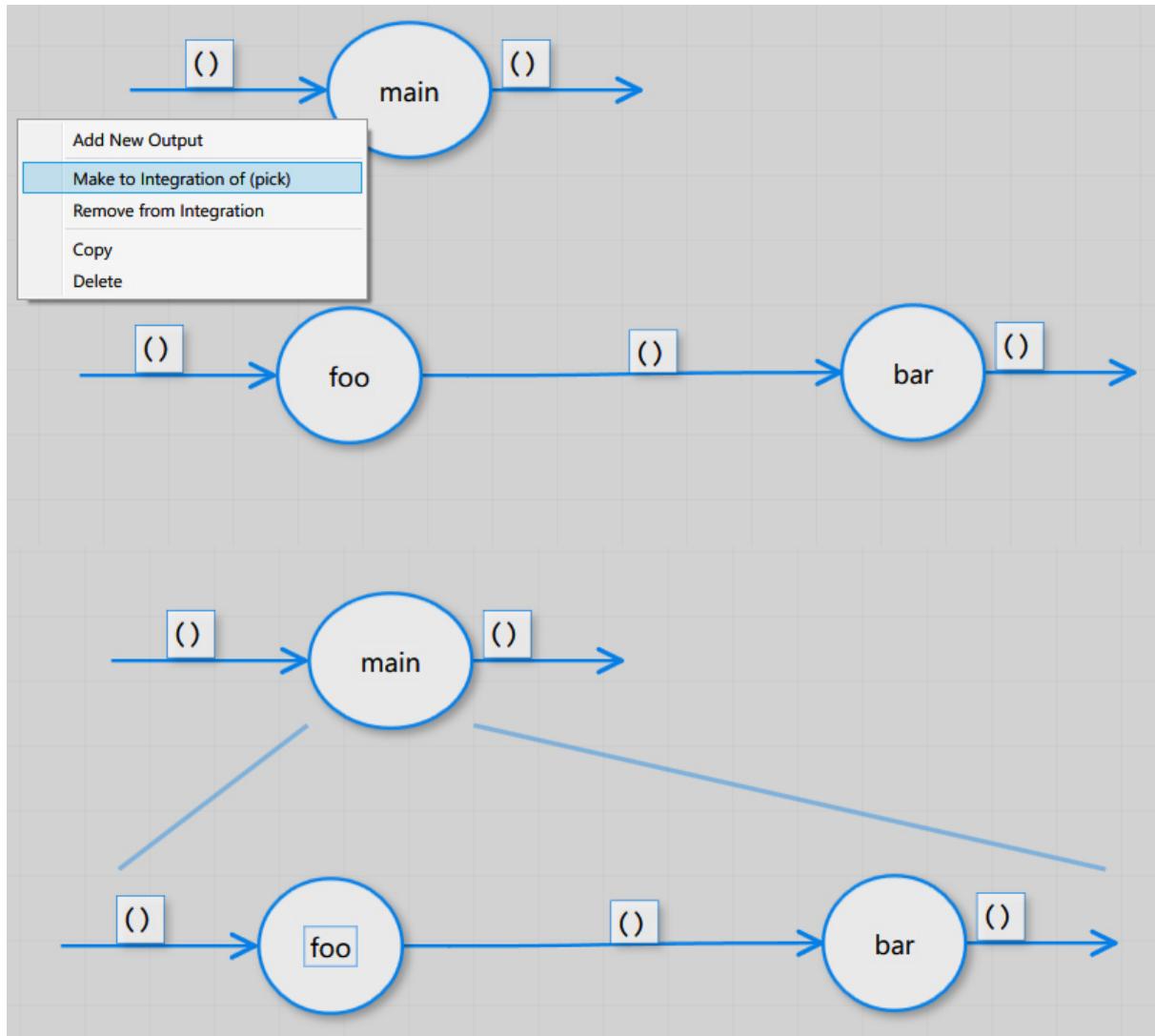


Abbildung 10.13.: Erstellen von Integrationen

Das Entfernen eines Flows von einer Integration ist auch über das Kontextmenü möglich. Dabei wird der komplette zusammenhängende Flow aus der Integration entfernt.

Bei einer Modifikation des Datenflusses innerhalb einer Integration, werden nach bestimmten Regeln die nachfolgenden (abgetrennten) Funktionseinheiten ebenfalls aus der Integration entfernt. Wird die erste Funktionseinheit entfernt, gilt diese Regel nicht.

Definieren von Datentypen

Eigene Datentypen können auf der rechten Seite des Editors angelegt werden. Durch *Rechtsklick -> Add New DataType*. Außerdem zeigt der obere Button an, ob im aktuellen Diagramm Datenflüsse mit Daten fließen, die nicht definiert sind (int, string, double, usw. werden automatisch ignoriert). Ein Klick auf diesen erstellt für jeden nicht-definierten Datentyp einen neuen Eintrag. Auch Datentypen innerhalb eines eigenen Datentypen werden überprüft, ob sie bekannt sind oder nicht.

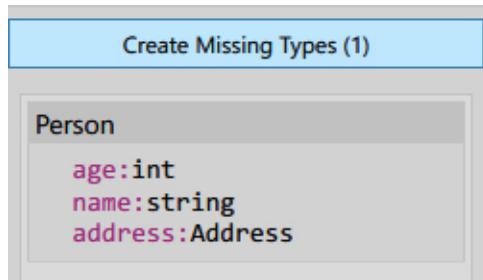


Abbildung 10.14.: Der Editor überprüft, ob für *Address* ein Datentyp definiert wurde.

Navigation und Tastenkürzel

Zur Navigation innerhalb des Zeichenbereichs wird das Mausrad verwendet. Durch Scrollen wird in den Zeichenbereich herein und heraus gezoomt. Durch gedrückt halten des Mausrades (mittlere Maustaste) und Bewegen der Maus kann die Ansicht verschoben werden (das Zeichenbereich ist endlos groß).

Ein weiteres Feature besteht darin, dass ab bestimmten Zoom-Stufen die Schriftgrößen der Namen der Funktionseinheiten angepasst werden und die Textfelder der Datenflüsse ausgeblendet werden. Außerdem können Textfelder nicht mehr fokussiert werden, was ein einfacheres Verschieben der Funktionseinheiten ermöglichen soll. Diese Gegebenheit wird visuell kenntlich gemacht, indem die Hintergrundfarbe des Textfeldes bei einer selektierten Funktionseinheit mit der Selektierungsfarbe übereinstimmt. Auch der Mauszeiger zeigt bei einem Mouseover an, dass nun nicht mehr in das Textfeld geklickt werden kann.

Weitere Features

- Speichern, Laden, Mergen

Das Speichern und Laden in drei Dateiformate wird unterstützt (yaml, json und xml - nach Dateigröße aufsteigend sortiert). Auch das Laden eines anderen Flow Designs in das aktuelle geladene wird mit der Merge-Funktion unterstützt.

- Unhandled Exception Error Dialog

Wird eine Exception geworfen, die nicht behandelt wurde, so wird eine allgemeine Fehlerbehandlung aufgerufen. Das Programm stürzt somit nicht ab, sondern ein Dialog erscheint, mit einem Stacktrace und Informationen über die geflogenen Exception. Der Benutzer kann entscheiden, ob er hier das Programm beenden, oder den Fehler ignorieren will. Der Stacktrace kann gegenfalls kopiert und an den Entwickler als Bugreport zugeschickt werden.

- Help-Window

Ein einfaches Hilfefenster, dass dem Benutzer einen Überblick über die vorhandenen Tastenkürzel zeigt und die Navigation mit der Maus erklärt.

10.3.2. Views / ViewModels

Das Projekt wurde nicht strikt nach MVVM-Pattern (Model-View-ViewModel) umgesetzt, jedoch bedient es sich der Idee, dass es eine View gibt, die als Datenkontext ein ViewModel zugewiesen bekommen hat. Durch das Zuweisen eines Datenkontextes wird es GUI-Elementen der View ermöglicht sich an Eigenschaften des ViewModels/Models zu binden. Ein *Bindung* bewirkt, dass sich das GUI-Element automatisch aktualisiert, sobald sich die dazugehörige Eigenschaft ändert. Eine Änderung einer Eigenschaft des ViewModels ändert somit automatisch die View.

Die GUI besteht aus mehreren Views (xaml-Dateien) und dazugehörigen ViewModels. Die Aufgabe des ViewModels besteht vor allem darin, ein Domänenmodell entgegenzunehmen und dieses darzustellen, bzw. die aktuelle Darstellung zu aktualisieren.

Nach jeder Änderung am Domänenmodell - zum Beispiel das Hinzufügen einer neuen Funktionseinheit - dieses komplett neu zu laden (Löschen und neu Hinzufügen aller ViewModels, die wiederum ein Neugenerieren der UI-Framework-Elemente zur Folge hatte) erwies sich als nicht sehr performant. Ab Diagrammen, mit über 20 Nodes, stieg die Zeit zur Aktualisierung der View bereits auf mehrere Sekunden an. Die Lösung bestand darin, nicht einfach alles zu Löschen und neu hinzuzufügen, sondern darin, die Änderungen am Modell zu lokalisieren und nur diese neu zu erstellen, bzw. nur die Eigenschaften neu zu setzen. Durch diese Verbesserungen wurde die Performance deutlich gesteigert, sodass Diagramme mit mehreren hundert Funktionseinheiten keine spürbaren Perfomanceverluste mit sich führen. Einzig das Duplizieren von vielen Funktionseinheiten dauert nach wie vor etwas länger.

10.3.3. Interaktionen

Wie bereits im Grundlagen-Teil erwähnt (Abschnitt Entwurfsmethode), schlägt Flow Design vor, alle Events als Interaktionen zu bezeichnen und für jedes dieser Änderungen ein eigenen Flow Design zu erstellen. Es bietet sich somit an, alle Interaktionen in einer Klasse zu sammeln. Diese bietet somit einen Überblick über alle Funktionalitäten der GUI. Da diese Integrationen sind, sind sie in der Regel leicht zu verstehen. Die Interaktionen rufen Methoden von anderen Klassen auf, die die Operationen am MainModel vereinfachen. Am Ende fast jeder Interaktion wird die ViewRedraw Methode aufgerufen, die das MainViewModel veranlasst, das Modell neu zu laden und somit die Änderungen der Interaktion in der GUI sichtbar macht. Aus diesem Grund erwies es sich als schlecht, wenn eine Interaktion eine andere Interaktion aufruft, um ihre Funktionalität umzusetzen. Stattdessen war es eine bessere Lösung, den Code der einen Interaktion in die andere zu kopieren. Dies widerspricht zwar dem DRY Prinzip, jedoch stellt sich heraus, dass Coderedundanzen innerhalb von Integrationen nichts Schlimmes sind. Integrations beinhalten schließlich keine Logik³ und haben eine hohe Abstraktion.

Beispiel dieser Aussage:

³Beim Aufruf einer Funktionseinheit, die mehrere Outputs liefert, existiert eigentlich doch Logik in der Integration (Beispiel IsIntegration-Methode: Wenn ja, dann X, wenn nicht, dann Y). Dadurch verlieren Integrationen etwas an ihrer Leichtgewichtigkeit und sind nicht mehr ganz so einfach zu verstehen.

```

public static object AppendNewFunctionUnit(FunctionUnit
    currentFunctionUnit, double offsetX, DataStreamDefinition
    outputToConnect, MainModel mainModel)
{
    var newFunctionUnit = FunctionUnitManager.CreateNew();

    newFunctionUnit.Position = currentFunctionUnit.Position;
    newFunctionUnit.MoveX(offsetX);

    // default IO of new function unit: input of new function unit
    // = output to connect to of current function unit
    newFunctionUnit.InputStreams.Add(
        DataStreamManager.NewDefinition(newFunctionUnit,
            outputToConnect));
    newFunctionUnit.OutputStreams.Add(
        DataStreamManager.NewDefinition(newFunctionUnit, "()"));
    MainModelManager.ConnectTwoDefinitions(outputToConnect,
        newFunctionUnit.InputStreams.First(), mainModel);

    mainModel.FunctionUnits.Add(newFunctionUnit);

    ViewRedraw();

    return newFunctionUnit;
}

```

Listing 10.2: AppendNewFunctionUnit

```

public static FunctionUnit
CreateNewOrGetFirstIntegrated(FunctionUnit currentFunctionUnit,
    MainModel mainModel)
{
    FunctionUnit @return = null;

    currentFunctionUnit.IsIntegration(
        isIntegration: () => @return =
            currentFunctionUnit.IsIntegrating.First(),
        isNotIntegration: () =>
    {
        var newFunctionUnit = FunctionUnitManager.CreateNew();
        newFunctionUnit.Position = currentFunctionUnit.Position;
        newFunctionUnit.MoveY(100);

        newFunctionUnit.InputStreams.Add(
            DataStreamManager.NewDefinition(newFunctionUnit,
                currentFunctionUnit.InputStreams.First()));
        newFunctionUnit.OutputStreams.Add(
            DataStreamManager.NewDefinition(newFunctionUnit, "()"));

        currentFunctionUnit.IsIntegrating.Add(newFunctionUnit);
    });
}

```

```

    mainModel.FunctionUnits.Add(newFunctionUnit);

    @return = newFunctionUnit;
    ViewRedraw();
}

return @return;
}

```

Listing 10.3: CreateNewOrGetFirstIntegrated

Die AppendNewCell-Methode erzeugt eine neue Funktionseinheit und verschiebt diese entlang der X-Position. Außerdem setzt sie den Input gleich der DataStreamDefinition die übergebenen wurde und verbindet diese beide. AppendNewCell wird durch die Tastenkombination Ctrl-Tab ausgelöst, wenn sich der Tastaturfokus innerhalb eines Textfeldes einer View einer Funktionseinheit, oder eines Ausgangs befindet. Bei dem ersten Fall wird der erste unverbundene Ausgang genommen, an dem die neue Funktionseinheit angehängt wird⁴.

Beide Methoden geben eine Domänenmodell-Instanz als object an die GUI zurück. Die GUI-Logik findet dann die dazugehörige View und setzt den Fokus auf diese.

Beide Methoden sind Methoden aus der Interaktions-Klasse, sie werden also direkt aus einem Event von der GUI ausgelöst. Beide Methoden haben ähnliche Methodenaufruufe (Neue Funktionseinheit erzeugen, sie auf die gleiche Position zu setzen wie die der übergebenen Funktionseinheit, neue Standardwerte für den Output-Stream der neuen Funktionseinheit setzen, dem MainModel die neue Funktionseinheit hinzufügen und die Ansicht neu zu zeichnen). Diese könnten in eine neue Integration ausgelagert werden, hätte jedoch eine unnötige Verschachtlung von Code als Auswirkung. Besser ist es hier die Coderedundanzen nicht als Code-Smell zu sehen und diese zu belassen. So ist auf einen Blick ersichtlich, was die Interaktion genau macht, ohne im Code herum springen zu müssen.

10.3.4. Validierung des Datenflusses - Farbliche Kennzeichnung

Eine Instanz vom Typ ValidationException kann dem ViewModel übergeben werden. Die ValidationException beinhaltet die Information, ob es sich um eine Fehler oder um eine Warnung handelt (Fehler werden rot markiert, Warnungen grün), welche Elemente den Fehler anzeigen sollen, sowie einen Fehlertext, der als Tooltip an dem Element angezeigt wird.

⁴nicht Teil der gezeigten Methode.

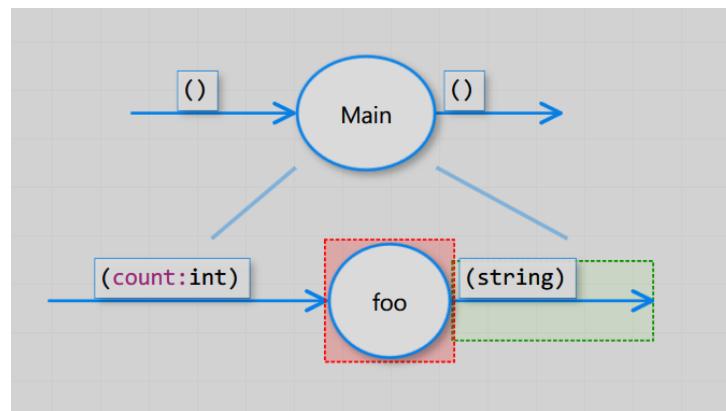


Abbildung 10.15.: Dexel - Warnung- und Fehler-Darstellung

10.4. Roslyn - Generierung von Code aus einem Diagramm

Da eine Flow Design auf unterschiedlichen Weisen im Code umgesetzt werden kann, mussten hier Entscheidungen getroffen werden, in welchen Fällen welche Weise gewählt wird. Wird das Projekt fortgeführt, könnte man in Betracht ziehen, dem Nutzer einige Optionen zur Konfiguration der Generierung zur Verfügung zu stellen. Aktuell bietet das Programm nur die hier vorgestellte Weise zur Verfügung.

10.4.1. Vorstellung - was erreicht wurde

Um den aktuellen Stand der Code-Generierung zu präsentieren, werden nachfolgend zwei Beispiele vorgestellt.

CSV Tabellieren - Beispiel aus YouTube Video von Ralf Westphal und Stefan Lieser

Die Aufgabe besteht darin, den Inhalt einer CSV-Datei in eine ACSII-Tabelle zu formalisieren [KATA, S.12].

	Name;Alter;Stadt	1	Name Alter Stadt
2	Paul;13;Köln	2	-----+-----+-----+
3	Peter;42;München	3	Paul 13 Köln
4	Maria;34;Hamburg	4	Peter 42 München
		5	Maria 34 Hamburg

Abbildung 10.16.: Aufgabe - CSV Tabellieren

Das Flow Design, das Ralf Westphal und Stefan Lieser in dem Video entwerfen sieht folgendermaßen aus.⁵

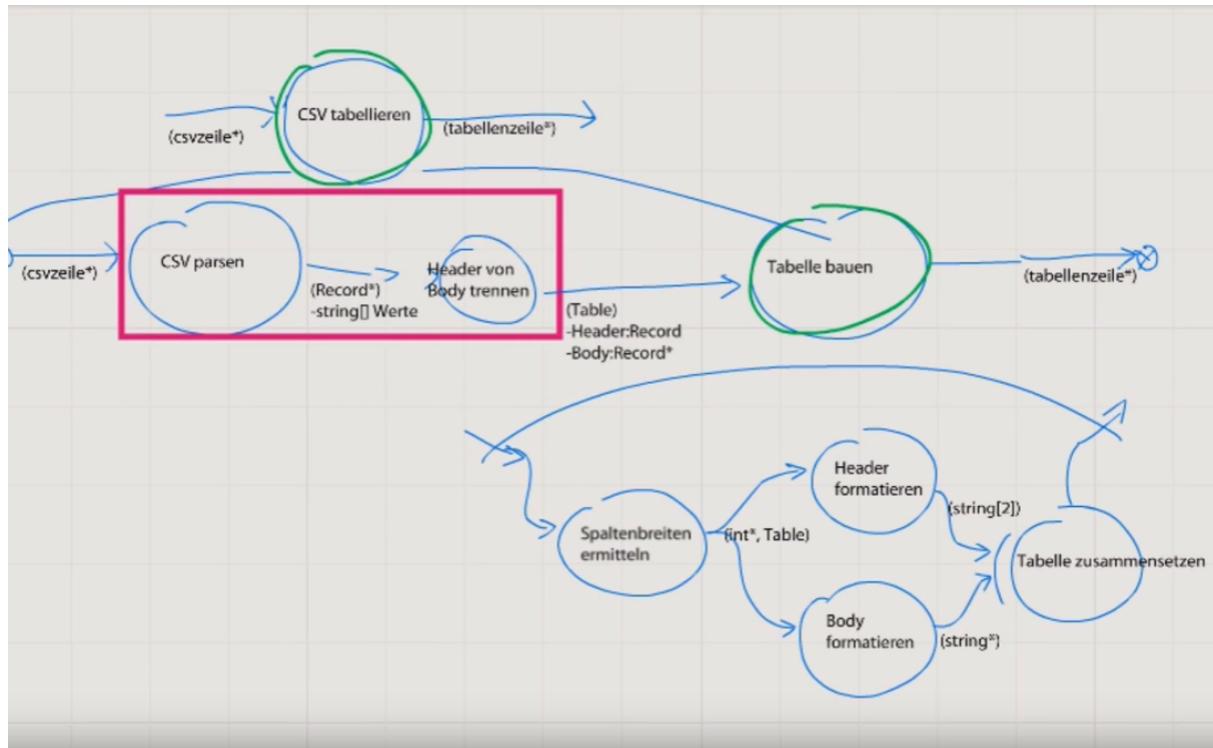


Abbildung 10.17.: CSV Tabellarisieren - Flow Design aus YouTube Video

Nun das Flow Design umgesetzt in Dexel. Zu beachten ist, dass die Split Output und Joined Input Notation mit der Pipe-Notation ersetzt wurde, da Dexel aktuell keine Joined-Notation unterstützt.

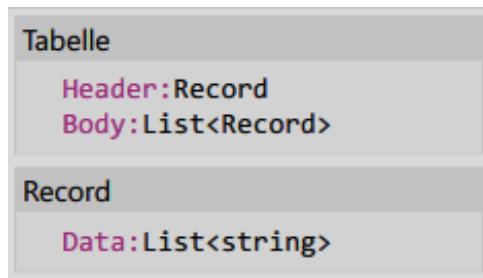


Abbildung 10.18.: CSV Tabellarisieren - Datentypen

⁵Video CSV Tabellarisieren [YT]

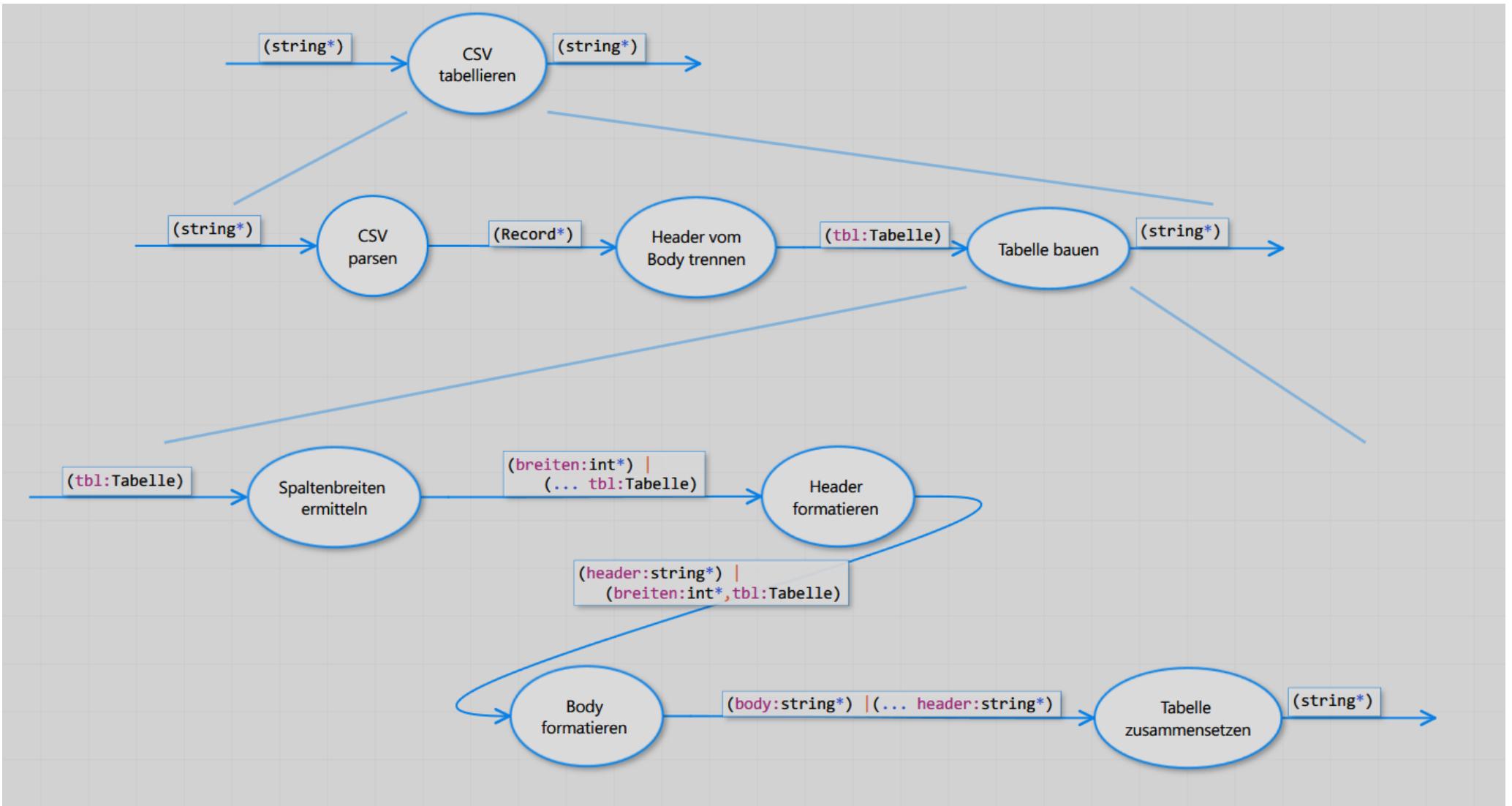


Abbildung 10.19.: CSV Tabellarisieren - Dexel Flow Design

```
// Integrationen
public static IEnumerable<string> CSVTabellieren(
    IEnumerable<string> strings)
{
    var records = CSVParsen(strings);
    var tbl = HeaderVomBodyTrennen(records);
    return TabelleBauen(tbl);
}

public static IEnumerable<string> TabelleBauen(Tabelle tbl)
{
    var breiten = SpaltenbreitenErmitteln(tbl);
    var header = HeaderFormattieren(breiten, tbl);
    var body = BodyFormattieren(breiten, tbl);
    return TabelleZusammensetzen(body, header);
}
```

Listing 10.4: CSV tabellieren mit Dexel generierter Code

```
// Datentypen
public class Tabelle
{
    public Record Header;
    public List<Record> Body;
}

public class Record
{
    public List<string> Data;
}
```

Listing 10.5: CSV tabellieren mit Dexel generierter Code

```
// Operationen
public static IEnumerable<int> SpaltenbreitenErmitteln(Tabelle tbl)
{
    throw new NotImplementedException();
}

public static IEnumerable<string>
    HeaderFormattieren(IEnumerable<int> breiten, Tabelle tbl)
{
    throw new NotImplementedException();
}

public static IEnumerable<string>
    BodyFormattieren(IEnumerable<int> breiten, Tabelle tbl)
{
    throw new NotImplementedException();
}
```

```
public static IEnumerable<string>
    TabelleZusammensetzen(IEnumerable<string> body,
    IEnumerable<string> header)
{
    throw new NotImplementedException();
}

public static Tabelle HeaderVomBodyTrennen(IEnumerable<Record>
    records)
{
    throw new NotImplementedException();
}

public static IEnumerable<Record> CSVParssen(IEnumerable<string>
    strings)
{
    throw new NotImplementedException();
}
```

Listing 10.6: CSV tabellieren mit Dexel generierter Code

Der Datenfluss in den beiden Integrationen wurde korrekt generiert. Außerdem wurden alle Datentypen und Methodensignaturen der Operationen erzeugt.

Shopping Simulator - Komplexeres Beispiel

Dieses Beispiel soll zeigen, wie auch mehreren Ausgänge von Funktionseinheiten und Streams korrekt generiert werden.

Der Methode ShoppingSimulator wird eine Anzahl an Kunden übergeben, die generiert werden soll. Jeder Kunde ist eine Person, die noch zusätzlich zu Name und Alter ein Budget hat. Diese drei Eigenschaften werden für jeden Kunde zufällig generiert. Anschließend wird ein zufälliger Einkaufswagen erzeugt und die beiden Objekte (Einkaufswagen und Kunde) durchlaufen eine Checkout-Routine. Das Ergebnis wird in die Konsole ausgegeben.

Eine mögliche Konsolenausgabe - nachdem die Methodenrümpfe der Operationen von Hand implementiert wurden - wäre:

```
Person created - age:20 budget:14,60
New shopping cart created - totalprice:102,50
Cart was too expensive. Price was: 102,50
Not enough money

Person created - age:11 budget:55,76
New shopping cart created - totalprice:133,66
Added discount
Cart was too expensive. Price was: 120,29
Not enough money

Person created - age:17 budget:54,69
New shopping cart created - totalprice:16,17
Added discount
Has enough money. Price was: 14,55
Checked out

Person created - age:31 budget:51,18
New shopping cart created - totalprice:116,67
Cart was too expensive. Price was: 116,67
Not enough money

Person created - age:18 budget:71,87
New shopping cart created - totalprice:62,41
Has enough money. Price was: 62,41
Checked out
Subscribed to newsletter
```

Abbildung 10.20.: Konsolenausgabe - ShoppingSimulator

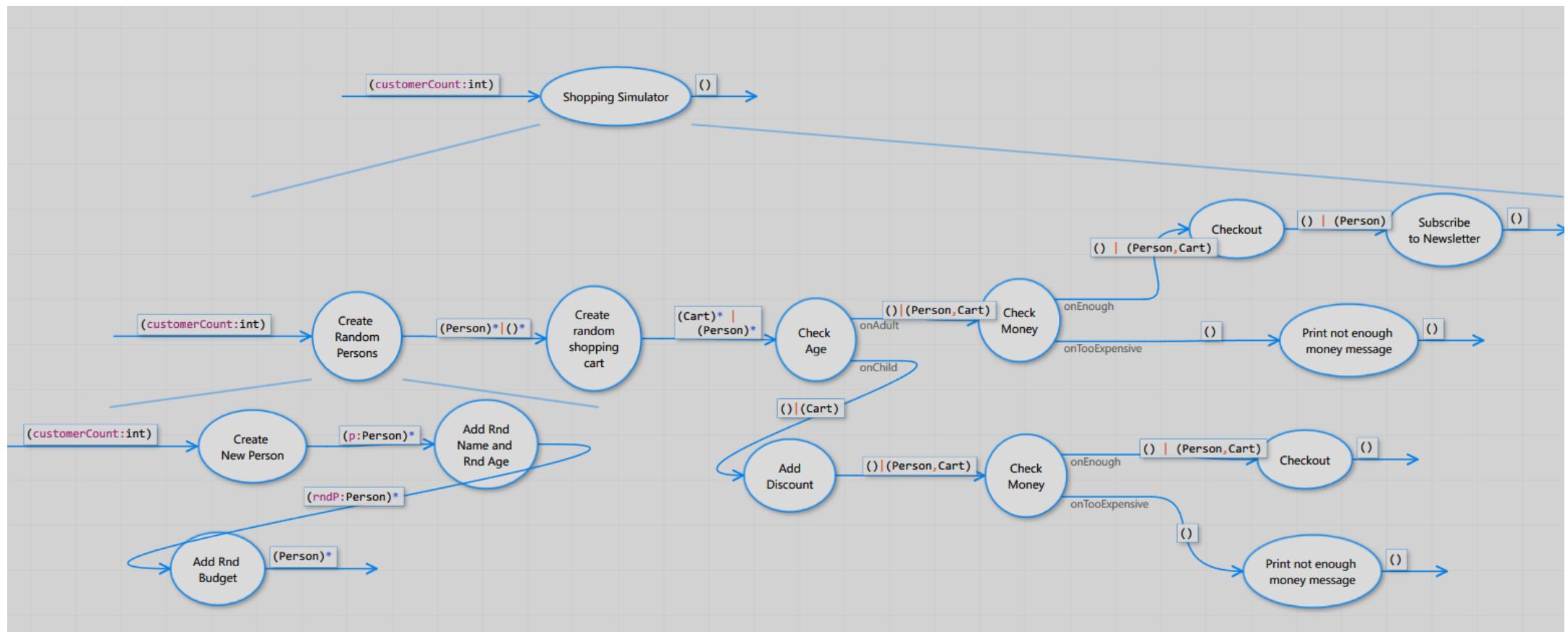


Abbildung 10.21.: Shopping Simulator

```
// Integrationen
public static void ShoppingSimulator(int customerCount)
{
    CreateRandomPersons(customerCount, person => {
        var aCart = CreateRandomShoppingCart();
        CheckAge(person, onAdult: () => {
            CheckMoney(person, aCart, onEnough: () => {
                Checkout(aCart, person);
                SubscribeToNewsletter(person);
            }, onTooExpensive: () => {
                PrintNotEnoughMoneyMessage();
            });
        }, onChild: () => {
            AddDiscount(aCart);
            CheckMoney(person, aCart, onEnough: () => {
                Checkout(aCart, person);
            }, onTooExpensive: () => {
                PrintNotEnoughMoneyMessage();
            });
        });
    });
}

public static void CreateRandomPersons(int customerCount,
    Action<Person> onPerson)
{
    CreateNewPerson(customerCount, p => {
        var rndP = AddRndNameAndRndAge(p);
        onPerson(AddRndBudget(rndP));
    });
}
```

Listing 10.7: Shopping Simulator - automatisch generierter Code mit Dexel

```
// Datentypen
public class Person
{
    public int Age;
    public string Name;
    public double Budget;
}

public class Cart
{
    public List<string> Products;
    public double Discount;
    public double PriceTotal;
}
```

Listing 10.8: Shopping Simulator - automatisch generierter Code mit Dexel

```
// Operationen

public static Cart CreateRandomShoppingCart()
{
    throw new NotImplementedException();
}

public static void CheckAge(Person aPerson, Action onAdult, Action
    onChild)
{
    throw new NotImplementedException();
}

public static void CheckMoney(Person aPerson, Cart aCart, Action
    onEnough, Action onTooExpensive)
{
    throw new NotImplementedException();
}

public static void AddDiscount(Cart aCart)
{
    throw new NotImplementedException();
}

public static void SubscribeToNewsletter(Person aPerson)
{
    throw new NotImplementedException();
}

public static void PrintNotEnoughMoneyMessage()
{
    throw new NotImplementedException();
}

public static void Checkout(Cart aCart, Person aPerson)
{
    throw new NotImplementedException();
}

public static void CreateNewPerson(int customerCount,
    Action<Person> onP)
{
    throw new NotImplementedException();
}

public static Person AddRndNameAndRndAge(Person p)
{
    throw new NotImplementedException();
}

public static Person AddRndBudget(Person rndP)
```

```
{  
    throw new NotImplementedException();  
}
```

Listing 10.9: Shopping Simulator - automatisch generierter Code mit Dexel

Aktueller Stand

Die weniger komplexeren Aufgaben wurden vollständig implementiert. Diese wären:

- Generierung der benutzerdefinierten Datentypen, und
- Generierung der Methodensignaturen einer Funktionseinheit.

Die Generierung der Methodenrümpfe von Integrationen wurde auch zum Großteil implementiert. Jedoch kann erst durch ein prototypischen Einsatz herausgefunden werden, in welchen Fällen ein Flow-Design-Diagramme noch fehlerhaft interpretiert wird. Dafür fehlt im Rahmen dieser Arbeit aber die Zeit.

Die Validierung des Datenflusses ist eng an die Generierung gekoppelt, deswegen gilt das selbe auch für diese.

Auch bei der Generierung von Namen wird aktuell noch nicht in jedem Fall überprüft, ob es zu einer Überschneidung kommt und ob gegebenenfalls der Name angepasst werden muss.

Um Code zu generieren muss im Editor in der Menüleiste der Eintrag Output angesteuert werden. Hier gibt es die Möglichkeit das aktuelle Flow Design in eine Datei auf den Desktop zu generieren, den Code in die Konsole auszugeben, oder den Code direkt in die Zwischenablage zu kopieren (um den erzeugten Code bequem an eine gewünschte Stelle eingefügt werden kann).

Außerdem gibt es die Option eine automatische Generierung einzuschalten, die den Code während der Bearbeitung immer neu generiert und in die Datei auf den Desktop schreibt. So lässt sich das Ergebnis der Generierung während der Bearbeitung des Datenflusses beobachten (vor allem in Kombination mit einem Texteditor, wie zum Beispiel Atom, die bei einer Änderung der Datei diese automatische neu laden).

Das Ergebnis wird auch immer mit in die Konsole ausgegeben, die beim Programmstart mit geöffnet wird.

10.4.2. Kleiner Einblick in die API von Roslyn

Das Projekt Roslyn von Microsoft ist ein .NET-Compiler, der eine API bietet, um beliebigen C#- und VB.NET-Quelltext zu erstellen, zu analysieren und zu modifizieren.

Das hier verwendet Nuget-Paket ist: Microsoft.CodeAnalysis

Der Datentypen auf dem die Methoden zum Erstellen von Code arbeitet lautet: SyntaxNode

Diese SyntaxNodes werden von einem Syntaxgenerator erstellt. Dieser muss beim Initialisieren konfiguriert werden.

```

var workspace = new AdhocWorkspace();
// Get the SyntaxGenerator for the specified language
var generator = SyntaxGenerator.GetGenerator(workspace,
    LanguageNames.CSharp);

```

Listing 10.10: SyntaxGenerator für C# erhalten

Das Erstellen eines Ausdrückes, Klasse oder Methode mit Hilfe von Methoden des Syntaxgenerators liefern am Ende immer eine SyntaxNode zurück. Beim Erstellen einer Methode oder Klasse kann der Inhalt des Scopes durch ein Array von SyntaxNodes bestimmt werden.

Nachfolgend ein einfaches Beispiel zum Erstellen von einem benutzerdefinierten Datentyp.

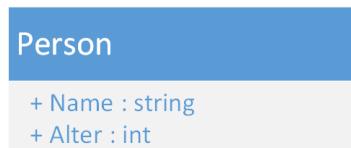


Abbildung 10.22.: Datentyp der generiert werden soll

Erzeugung eines Datentypen als einfaches Beispiel

```

SyntaxNode fieldName = generator.FieldDeclaration(
    name: "Name",
    type: generator.TypeExpression( SpecialType.System_String),
    accessibility: Accessibility.Public);

SyntaxNode fieldAlter = generator.FieldDeclaration(
    name: "Alter",
    type: generator.TypeExpression( SpecialType.System_Int32),
    accessibility: Accessibility.Public);

SyntaxNode[] allFields = new [] {fieldName, fieldAlter};

// Generate the class
SyntaxNode classDefinition = Generator.ClassDeclaration(
    "Person",
    typeParameters: null,
    accessibility: Accessibility.Public,
    modifiers: DeclarationModifiers.None,
    baseType: null,
    interfaceTypes: null,
    members:allFields
);

```

Listing 10.11: Erzeugung einer Person-Klasse mit Roslyn

Um am Ende den Code in Form eines Strings zu erhalten muss auf die oberste SyntaxNode folgende beide Methoden aufgerufen werden:

```
string code =
    classDefinition.NormalizeWhitespace().ToFullString();
```

Listing 10.12: Erhalten des Codes als string

10.4.3. Erzeugung von Methodensignaturen

Das Analysieren einzelner Funktionseinheiten anhand ihrer Ein- und Ausgänge erlaubt eine automatische Generierung der Methodensignaturen. Dieses Feature wurde vollständig implementiert, da die möglichen Faktoren, die Einfluss auf die Methodensignatur haben überschaubar sind und somit alle Kombinationen sich schnell herauskristallisiert haben.

Output über Rückgabewert

Der einfachste Fall ist eine Funktionseinheit, die nur ein Ausgang hat und dieser auch kein Stream und auch nicht optional ist.⁶

In diesem Fall werden die ausgehenden Daten einfach als Rückgabewert heraus gebracht.



Abbildung 10.23.: Dexel-Screenshot: Einfache Funktionseinheit mit benutzerdefinierten Datentyp

Die Generierung erzeugt folgenden Code:

```
public class Person
{
    public string Name;
    public int Age;
}

public static Person NewPerson(int aint, string astring)
```

⁶Im Editor ist das vorhanden sein eines Eingangs vorgeben und es gibt auch keine Möglichkeit einen weiteren Eingangs-Datenstrom zu einer Funktionseinheit hinzuzufügen. Nur Ausgänge lassen sich hinzufügen und entfernen.

```
{
    throw new NotImplementedException();
}
```

Listing 10.13: Mit Dexel generierter Code

Hat ein Output mehr als ein Datentyp, ist kein Stream und ist auch nicht optional, so wird das Ergebnis als Tupel über den Rückgabewert geliefert.



Abbildung 10.24.: Dexel-Screenshot: Funktionseinheit mit Tupel-Output

```

public static Tupel<Person, int> NewPerson(int aInt, string
aString)
{
    throw new NotImplementedException();
}
  
```

Listing 10.14: Mit Dexel generierter Code

Outputs über Actions

Durch die optionale Vergabe eines Namens für den Ausgang (*Actionname*) steuert der Anwender indirekt, ob ein Ausgang durch eine Action realisiert werden soll⁷.

Ausgänge müssen nicht zwingend mit der Anzahl an Aufrufe der Funktionseinheit übereinstimmen. Ein Action muss nicht aufgerufen werden, oder kann bei einem einzigen Aufruf der Funktionseinheit auch mehrmals aufgerufen werden (der Beginn eines Streams).

Sobald auch eine Funktionseinheit mehrere Ausgänge hat, diese jedoch nicht optional sind, werden diese jedoch trotzdem als Actions realisiert, selbst wenn sie kein Actionname zugeordnet bekommen haben. Der Name des Actions wird dann automatisch generiert⁸.

Durch diese Entscheidungen gibt es keine invaliden Kombinationen an Ausgängen einer Funktionseinheit.

Werden alle Ausgänge mit Actions realisiert, entfällt der Rückgabewert in diesem Fall komplett.

⁷Diese Regel wurde im Rahmen dieser Anwendung so festgelegt.

⁸ Durch Analyse der Datentypen. Ein *(Person)* wird zu einem *onPerson*. Ein *()* wird zu einem *continueWith*.

Die Input-Daten werden in der Signatur zuerst aufgelistet, anschließend die Actions.



Abbildung 10.25.: Dexel-Screenshot: Funktionseinheit mit zwei Outputs mit definierten Actionnamen

```

public static void NewPerson(int aint, string astring,
    Action<Person> onPerson, Action<string> onError)
{
    throw new NotImplementedException();
}
  
```

Listing 10.15: Mit Dexel generierter Code

Streams

Streams werden in Dexel ebenfalls mit Actions realisiert, mit einer Ausnahme:

Sind ist sowohl der Eingang als auch der Ausgang ein Stream und ist der *Actionname* dieses Ausgangs *nicht* angegeben, so wird implizit davon ausgegangen, dass für jeden Aufruf der Funktionseinheit dieser Ausgang erzeugt wird.

Somit kann für diesen Ausgang auf eine Action verzichtet werden und der Rückgabewert genutzt werden, was eine einfachere Verwendung dieser Funktionseinheit im Code zur Folge hat.⁹

⁹ Das Vorhandensein von einem Inputstream und einem Outputstream bedeutet eigentlich nicht zwingend, dass für jeden Input auch ein Output erzeugt wird. Die Notation unterscheidet beide Fälle nicht. Deswegen die Lösung über eine Vergabe eines *Actionnamen* als Kompromiss zu sehen, wodurch dem Benutzer hier eine Kontrolle über die Erzeugung des Codes geben wird. Wenn der Benutzer ein *Actionname* angibt, wird indirekt davon ausgegangen, dass der Benutzer den Output auch als Action umsetzen möchte und die Funktionseinheit somit beliebig oft diese Action aufrufen kann. Ob dieses Entscheidung richtig war müsste noch in der Praxis erprobt werden. Stellt sich heraus, dass sie schlecht ist, müsste man über eine Anpassung der Notation nachdenken.

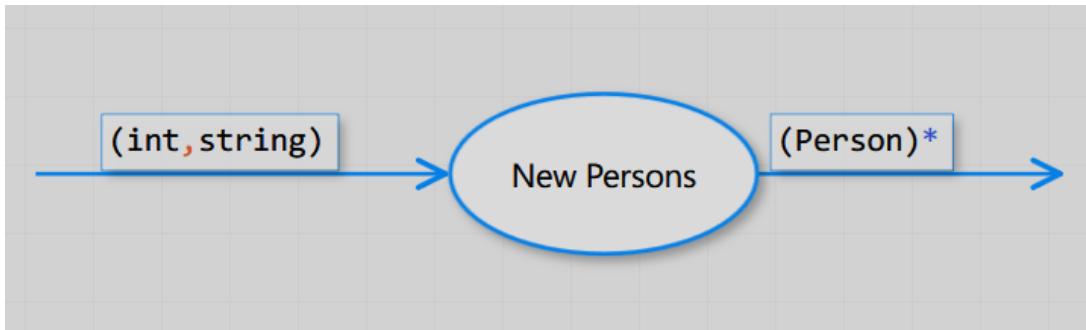


Abbildung 10.26.: Dexel-Screenshot: Funktionseinheit mit Stream als Ausgang

```
public static void NewPersons(int aInt, string aString,
    Action<Person> onPerson)
{
    throw new NotImplementedException();
}
```

Listing 10.16: Mit Dexel generierter Code

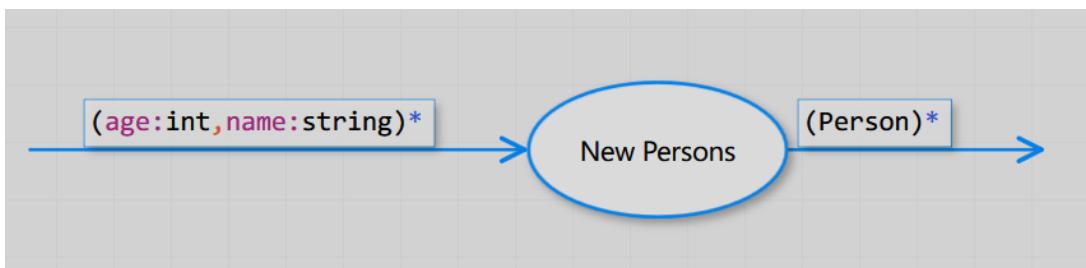


Abbildung 10.27.: Dexel-Screenshot: Funktionseinheit mit eingehendem und ausgehendem Stream und undefinierten Actionnamen

```
public static Person NewPersons(int age, string name)
{
    throw new NotImplementedException();
}
```

Listing 10.17: Mit Dexel generierter Code

10.4.4. Erzeugung des Methodenrumpfes einer Integration

Terminologie

Eine Integration kann nicht nur aus Operationen bestehen, sondern auch aus anderen Integrationen bestehen. Leider gibt es hierfür keine Überbegriff für beide. Da im in diese Kapitel jedoch öfters von Funktionseinheiten innerhalb einer Integration die Rede sein wird, muss hierfür ein Überbegriff eingeführt werden. Funktionseinheiten, die sich in einer Integration befinden, werden nachfolgend als Sub-Funktionseinheiten bezeichnet.

Die Herangehensweise

Die automatische Erzeugung einer Integration ist die komplexeste Aufgabe in diesem Projekt. Um die Aufgabe überschaubar zu halten, wurde diese im Groben in zwei Teilaufgaben zerteilt: die Analyse und die Generierung. Die Analyse besteht wiederum aus unterschiedlichen Methoden, die jeweils das Flow Design auf eine bestimmte Sache analysieren und das Ergebnis in ein Objekt abspeichern. Diese Objekt wurde als `IntegrationBody` bezeichnet. Diese beinhaltet alle Informationen aus der Analyse. Am Ende wird dieses Objekt der Generierungsmethode übergeben, die anhand diesem den Code erzeugt¹⁰.

¹⁰Ein weiterer Vorteil dieser Aufteilung ist, dass ein großer Teil frei von der Roslyn-API bleibt. Die eigenen Datentypen lassen sich dazu auch besser testen, als die SyntaxNodes von Roslyn. Möchte man das Ergebnis von Roslyn testen, bleibt einem oft keine andere Wahl, als den erzeugten Code samt White-spaces über ein String-Vergleich zu erschlagen, was bei mehrzeiligem Code doch etwas ausufern kann. Stattdessen lässt sich das Ergebnis aus den eigenen Analysen hingegen einfach über ein Vergleich von den gefundenen Objekt-Referenzen überprüfen.

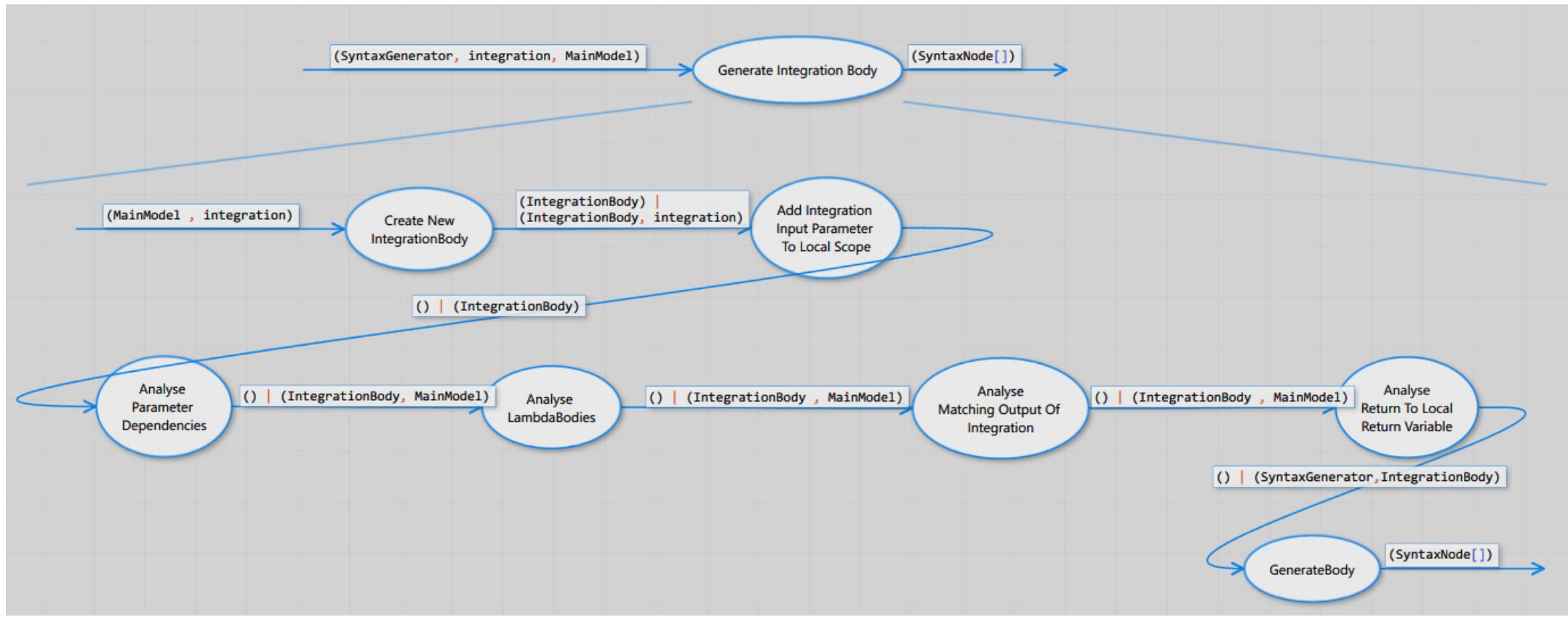


Abbildung 10.28.: Generate Integration Body - Flow Design

Analyse des Flow Designs

1. AnalyseParameterDependencies

Da die Pipe-Notation unterstützt wird, reicht es nicht einfache aus die Daten aus den Datenflüssen zu nehmen, die direkt in die aktuelle Funktionseinheit fließen. Stattdessen muss der Fluss rückwärts traversiert werden und auf Übereinstimmungen untersucht werden. Für jede Sub-Funktionseinheit muss diese Analyse durchgeführt werden. Für jeden Input-Datentyp jeder Funktionseinheit wird das Ergebnis gespeichert. Das Ergebnis beinhaltet ob der Datentyp gefunden wurde, ob er im Eingang der Integration gefunden wurde, oder ob er aus einer anderen Sub-Funktionseinheit innerhalb des Datenflusses stammt. Ein Aufruf einer Sub-Funktionseinheit kann nur generiert werden, wenn alle Datentypen gefunden wurden.

2. AnalyseLambdaBodies

Manche optionale Datenflüsse oder Stream werden über Actions realisiert. Das bedeutet, dass sich alle nachfolgenden Funktionseinheiten innerhalb des Lambda-Ausdruckes befinden müssen. Die Analyse speichert das Ergebnis in folgender Form ab.

```
public class LambdaBody
{
    public DataStreamDefinition InsideLambdaOf;
    public FunctionUnit FunctionUnit;
}
```

Listing 10.18: LambdaBody Klasse

Da die LambdaBody-Objekte in der Reihenfolge im IntegrationBody abgelegt werden, in der sie im Flow Design vorkommen, kann daraus auch später die Reihenfolge der zu generierenden Methoden abgeleitet werden.

Für jede Funktionseinheit wird solch ein Objekt angelegt, auch wenn es nicht innerhalb eines Lambda-Ausdrucks vorkommt. Ist das der Fall, so ist der Wert InsideLambdaOf null. Alle Funktionseinheiten für die dieser Wert null ist, befinden sich somit direkt im Scope der Integration und nicht innerhalb eines Lambda-Ausdrucks.

3. AnalyseMatchingOutputOfIntegration

Hat die Integration einen Ausgang der als Action realisiert werden muss, so muss herausgefunden werden, welche Sub-Funktionseinheiten diesen Ausgang bedienen. Dabei werden die Implementierungs-Stile der beiden übereinstimmenden Ausgänge mit abgespeichert. Später bei der Generierung gibt es somit vier Möglichkeiten:

- Beide sind Actions

Die Action der Integration wird direkt an die Sub-Funktionseinheit weitergereicht. Dadurch erlaubt man einer Sub-Funktionseinheit das Aufrufen des Ausgang der Integration.

```
// Integration
public static void Main(Action<string> onError)
{
    DoSomething(onError);
}

// Operation
public static void DoSomething(Action<string> onError)
{
    throw new NotImplementedException();
}
```

Listing 10.19: Action-Action-Beziehung

- Integrationsausgang ist Action, Sub-Funktionseinheitsausgang ist Rückgabewert

Die Action wird aufgerufen mit dem Methodenaufruf als Parameter.

```
public static void CreateRandomPersons(int customerCount,
    Action<Person> onPerson)
{
    CreateNewPerson(customerCount, p => {
        var rndP = AddRndNameAndRndAge(p);
        onPerson(AddRndBudget(rndP));
    });
}
```

Listing 10.20: Action-Return-Beziehung

- Integrationsausgang ist Rückgabewert, Sub-Funktionseinheitsausgang ist Action.

Eine lokale Variable muss vorher angelegt werden und mit null initialisiert werden. Danach wird innerhalb des Lambdas des Actions diese Variable beschrieben. Am Ende wird die lokale Variable als Rückgabewert ausgegeben.

```
public static void TryGetMessage(Action<string> onMessage)
{
    throw new NotImplementedException();
}

public static string Main()
{
    string @return = null;
    TryGetMessage(msg => @return = msg );
    return @return;
}
```

Listing 10.21: Return-Action-Beziehung

- Beide Ausgänge werden über Rückgabewert realisiert

Das Ergebnis der Sub-Funktionseinheit wird durch ein Rückgabewert-Ausdruck aus der Integration heraus gereicht.

```
public static string GetMessage()
{
    throw new NotImplementedException();
}

public static string Main()
{
    var msg = GetMessage();
    return msg;
}
```

Listing 10.22: Return-Return-Beziehung

4. AnalyseReturnToLocalReturnVariable

Wird der Ausgang einer Integration mit dem Rückgabewert realisiert, so muss noch herausgefunden werden, ob eine lokale Variable nötig ist, um den Rückgabewert aus einem Lambda-Ausdruck heraus zu reichen. Ist eine Return-Return-Beziehung vorhanden und befindet sich die betroffene Methode innerhalb eines Lambda-Ausdrucks, so muss der Rückgabewert in die lokale @return-Variable geschrieben werden.

10.5. Generierung eines Diagrammes aus Code

Aus zeitlichen Gründen, konnte die Generierung von einem Flow Design aus Code nicht angegangen werden. Da sich das Modell jedoch sauber getrennt von der GUI in einem separaten Unterprojekt befindet, wäre es sicherlich möglich einen anderen Studenten dies als Projekt zu übergeben. So hätte dieser bereits eine Möglichkeit seine Diagramme darzustellen, ohne sich sonderlich in die Codebasis des gesamten Dexel-Projekts einarbeiten zu müssen.

11. Zusammenfassung

11.1. Ausblick

Hier eine Auflistung dessen, was aus zeitlichen Gründen noch nicht realisiert wurde, jedoch für den Einsatz in einem Projekt noch sehr wichtig wären. Diese hat keinen Anspruch auf Vollständigkeit.

Editor

- Joined-Inputs und auch Split Outputs. Dies erfordert möglicherweise noch einmal eine kleine Überarbeitung des Domänenmodells. Auch die Darstellung wurde noch nicht realisiert. Zusätzlich wird die Generierung etwas komplexer und wirft weitere Fragen auf eine eindeutige Interpretation auf.
- Validierung der Syntax - Markierung invalider Syntax innerhalb des Editors wurden nicht realisiert (z.B. sind die Klammern richtig gesetzt).
- Search-Replace Funktionalität, um Datentypen oder andere Namen, die an vielen Stellen verwendet werden, schnell und einfach umbenennen zu können.
- Mehr Möglichkeiten die Darstellung übersichtlicher zu gestalten: Auf und Zuklappen von Integrationen. Eine 'referenzierte' Kopie einer Funktionseinheit zu erstellen, um an anderer Stelle diese weiter zu verwenden.
- Saubere und eindeutige Darstellung einer Rekursion.
- Zuordnen von Funktionseinheiten zu Klassen.
- Möglichkeit mehrere Flow Design in einer Art Projekt zu organisieren. Vielleicht auch in Form eines GUI-Skizzen-Editors, indem dann die Interaktionen eingetragen werden können.
- Autosave und Undo/Redo.
- Eine Möglichkeit große unüberschaubare Flow Designs in kleinere aufzuteilen und diese an anderen Stelle zu definieren.

Generierung von Code

- Übergabe einer Eigenschaft eines Objektes an eine Funktionseinheit. Bsp: product.Price -> CalculateDicount -> ...
- Zusammenführen von Datenströmen (Joined Inputs), in manchen Fällen fördert dass die Leserlichkeit.
- Weitere Möglichkeiten einführen um die Daten auf den Datenflüssen kurz und leserlicher zu halten. Zum Beispiel indem man Aliase für Typen definieren kann, sodass man statt breiten:int* auch einfach nur breiten schreiben kann.
- Registrieren von neuen bekannten Datentypen, sodass diese nicht als „missing Datatypes“ angezeigt werden und auch nicht generiert werden, falls man diese doch in Dexel definiert. Wenn zum Beispiel aus einer externen Bibliothek Klassen benutzt werden.
- Erkennen von vorhandenen Methoden aus der Standardbibliothek/LINQ, sodass für diese keine neuen Methoden erzeugt werden. Auch hier müsste man dann eine Notation einführen, um ausdrücken zu können, dass eine Methode des Objekts aufgerufen werden soll und nicht einer Methode das Objekt übergeben werden soll.

11.2. Fazit

Flow Design

Mir persönlich gefällt IOSP sehr gut. Nach einer Eingewöhnungszeit lernte ich die Lambda-Schreibweise lieben und finde sie nun auch sehr leserlich. Der Code der sonst in vielen Unterfunktionen versteckt ist, kann so an einer Stelle gehalten werden und trotzdem bleibt die Abstraktionsebene hoch. Ab einer gewissen Tiefe ist eine Aufteilung in unterschiedliche Methoden jedoch trotzdem sinnvoll.

Den Vorteil von entkoppelten Methoden lernte ich im Dexel Projekt kennen. Methoden können leichter angepasst werden und an unterschiedlichen Stellen verwendet werden. Die Aufgabe einer Operation ist klar definiert, was sie einfacher zu implementieren und besser testbar macht.

Die Tatsache, dass dadurch die Integrationen frei von Kontrollstrukturen bleiben finde ich auch gut. Leider gilt die Aussage, das die Integrationen dadurch leicht zu verstehen bleiben, da sie frei von „Logik“ bleiben leider nicht immer. Wenn eine Operation zwei oder mehrere Ausgänge hat und diese in der Integration „verdrahtet“ werden, so landet doch eine gute Portion Logik auch in der Integration und dadurch wird diese auch schwerer zu verstehen. Trotzdem würde ich weiter diese Art bevorzugen. Falls es mal nicht so sein sollte, steht es einem schließlich auch frei an jenen Stellen von der Regel abzuweichen.

Leider habe bei der Umsetzung von IOSP innerhalb von Dexel so gut wie nie auf eine Flow Design Diagramm zurückgegriffen, stattdessen habe ich es innerhalb der IDE „herunterprogrammiert“ und anschließend refaktorisiert. Das mag vor allem daran liegen, dass mir eine IDE durch Intellisense oft viel Arbeit erspart, gerade wenn bereits eine große Codebasis vorhanden ist. Mit Intellisense kann ich bereits vorhandene Methoden

auffinden und bekomme angezeigt, welche Parameter diese erwarten. Auch die Eigenschaften eines Objektes werden mir übersichtlich angezeigt. All das habe ich auf dem Papier (oder auch im aktuellen Stand von Dexel nicht).

Dexel

Wie bereits erwähnt, habe ich beim Programmieren von Dexel noch nicht so oft das Bedürfnis verspürt eine Flow Design Diagramm zu erstellen, wodurch ich die Sinnhaftigkeit eines Editors in Frage stellte. Auch eine automatische Generierung vollständig und ohne Fehler zu implementieren ist ein aufwendiges Unterfangen. Verglichen dazu ist der gewonnene Zeitaufwand relativ gering. Die paar Zeilen der Integration kann man auch selbst relativ schnell herunter tippen, letzte Anpassungen vornehmen und dabei auch LINQ verwenden, um auch komplexe Algorithmen runter zu schreiben.

Es bedarf vielleicht erst einen perfekten Roundtrip, um eine automatische Generierung zu rechtfertigen und sinnvoll in einem Workflow mit einbinden zu können.

Die Generierung könnte alternativ natürlich auf für Schulungszwecken verwendet werden, um Studenten/Entwicklern IOSP näher zu bringen.

Ein Entwurf auf dem Papier hat immer noch einen Einfachheit, die in Dexel noch nicht erreicht wurden. Das mag vor allem daran liegen, dass noch nicht alle Notation im Editor umsetzbar sind (Joined Inputs). Auch einfach Pfeile von A nach B zu zeichnen, ohne auf eine für den Computer eindeutige Interpretation zu achten, hat seine Vorteile.

Dexel hat jedoch auch einige Vorteile gegenüber dem Entwurf auf dem Papier. Der erste ist Geschwindigkeit. Namen können auf der Tastatur schneller geschrieben werden, als auf mit dem Stift. Mit Hilfe von Tastenkürzel lassen sich schnell neue Funktionseinheiten hinten anhängen und so den Datenfluss schnell herunter schreiben.

Einfaches Duplizieren des Schaubildes und vorhandene Elemente neu anzuordnen und Texte abzuändern ist ein klarer Pluspunkt für Dexel.

Auch eine Validierung des Datenflusses kann vorteilhaft sein.

Der aktuelle Stand ist denke ich trotzdem ein guter Start geworden, der eine Vision aufzeigt, wie ein Editor für Flow Design aussehen könnte. Mir persönlich fällt es schwer anhand des aktuellen Stands des Projektes abzuschätzen, ob ein vollausgereifter Editor nicht doch einen Platz finden könnte im „Werkzeugkasten“ eines Softwareentwicklers. Es würde trotzdem noch einige Monate an in Anspruch nehmen, um auf einen Stand zu kommen mit dem man sinnvoll arbeiten kann und erst dann lässt sich sagen, wie sinnvoll er im Einsatz eines Softwareprojektes wirklich ist. Vielleicht ergibt es sich die Möglichkeit im Rahmen einer Masterarbeit daran weiter zu arbeiten. Vielleicht findet sich auch ein anderer Student, der bereit ist, sich in ein bestehendes Projekt einzuarbeiten und im Rahmen seiner Bachelorarbeit daran weiterarbeiten möchte.

Abkürzungsverzeichnis

CCD Clean Code Development , Seite 11

DRY Don't Repeat Yourself, Seite 14

IODA Integration Operation Data API, Seite 28

IOSP Integration Operation Segregation Principle, Seite 30

KISS Keep It Stupid Simple, Seite 12

LINQ Language-Integrated Query, Seite 35

PoMO Principle of Mutual Oblivion, Seite 29

YAGNI You Ain't Gonna Need It, Seite 13

Quellenverzeichnis

- [CCD] clean-code-developer.de. [Online, abgefragt 03.01.2017]. URL: <http://clean-code-developer.de/>.
- [CLNCD] Robert C. Martin. *Clean Code - Refactoring, Patterns, Testen und Techniken für sauberen Code - Deutsche Ausgabe*. Heidelberg: MITP-Verlags GmbH & Co. KG, 2009. ISBN: 978-3-8266-5548-7.
- [CODEWHISPERER] Kevin Erath. *Code Whisperer*. [Online, abgefragt 18.01.2017]. URL: <http://www.code-whisperer.de/>.
- [EVOLVIERBARKEIT] *Evolvierbarkeit - CDD*. [Online, abgefragt 18.01.2017]. URL: <http://clean-code-developer.de/das-wertesystem/#Ervolvierbarkeit>.
- [F-D.ORG] flow-design.org. [Online, abgefragt 03.01.2017, Die Seite ist leider nie ganz fertig gestellt worden. Aktuell werden auch noch die Code-Beispiele unter der Rubrik Implementation nicht mehr angezeigt. Ein Aufsuchen der Github-Links im HTML-Code und manuelles Aufrufen dieser zeigt sie dann jedoch an.] URL: <http://flow-design.org/>.
- [IODA] *Die IODA Architekur*. [Online, abgefragt 03.01.2017]. URL: <http://blog.ralfw.de/2015/04/die-ioda-architektur.html>.
- [KATA] Ralf Westphal und Stefan Lieser. *The Architect's Napkin Kata für Kata*. Leanpub, 2014.
- [MICROSOFT] *microsoft.com - LINQ*. [Online, abgefragt 18.01.2017]. URL: https://msdn.microsoft.com/en-us/library/system.linq.enumerable_methods.
- [MSG] Ralf Westphal. *Messaging as a Programming Model*. Leanpub, 2016.
- [PERLS] *LINQ*. [Online, abgefragt 18.01.2017]. URL: <https://www.dotnetperls.com/linq>.
- [ROTERGRAD] *Roter Grad - CDD*. [Online, abgefragt 18.01.2017]. URL: <http://clean-code-developer.de/die-grade/roter-grad/>.
- [SCHMZL] Ralf Westphal. *The Architect's Napkin - Der Schummelzettel*. Leanpub, 2014.
- [YT] *Einige ca. einstündige Videos die exemplarisch die Flow Design Methode vorstellen*. [Online, abgefragt 03.01.2017]. URL: <https://www.youtube.com/user/ccdschool>.

Verzeichnis der Listings

5.1.	FormatAndPrintStrings nicht IOSP-konform	29
5.2.	FormatAndPrintStrings Variante 1	30
5.3.	FormatAndPrintStrings Variante 2	31
5.4.	FormatAndPrintStrings Variante 3	32
5.5.	FormatAndPrintStrings Variante LINQ	33
5.6.	Find the answer Implementation	38
5.7.	Mehrere Empfänger eines Outputs	39
5.8.	Typerischer Programmierstil, der nicht IOSP-konform ist	41
5.9.	IOSP-konforme Variante	41
5.10.	Auf Rückgabewert warten	42
10.1.	ChangeDatanames-Methode	68
10.2.	AppendNewFunctionUnit	75
10.3.	CreateNewOrGetFirstIntegrated	75
10.4.	CSV tabellieren mit Dexel generierter Code	80
10.5.	CSV tabellieren mit Dexel generierter Code	80
10.6.	CSV tabellieren mit Dexel generierter Code	80
10.7.	Shopping Simulator - automatisch generierter Code mit Dexel	84
10.8.	Shopping Simulator - automatisch generierter Code mit Dexel	84
10.9.	Shopping Simulator - automatisch generierter Code mit Dexel	85
10.10.	SyntaxGenerator für C# erhalten	87
10.11.	Erzeugung einer Person-Klasse mit Roslyn	87
10.12.	Erhalten des Codes als string	88
10.13.	Mit Dexel generierter Code	88
10.14.	Mit Dexel generierter Code	89
10.15.	Mit Dexel generierter Code	90
10.16.	Mit Dexel generierter Code	91
10.17.	Mit Dexel generierter Code	91
10.18.	LambdaBody Klasse	94
10.19.	Action-Action-Beziehung	95
10.20.	Action-Return-Beziehung	95
10.21.	Return-Action-Beziehung	95
10.22.	Return-Return-Beziehung	96

Abbildungsverzeichnis

3.1. Einfaches Flow Design Beispiel (die Beispieldaten unten sind nicht Teil der Notation sondern dienen hier nur dem besseren Verständnis)	16
4.1. Datenfluss zwischen drei Funktionseinheiten	17
4.2. Datentyp <i>int</i> mit Namen <i>alter</i>	17
4.3. Hierarchischer Datenfluss	18
4.4. Eigener Datentyp	18
4.5. Array Notation	19
4.6. Datenstrom Notation	19
4.7. Container Notation	20
4.8. Optionaler Output	20
4.9. Mehrere unterschiedliche Datentypen auf einem Datenstrom	21
4.10. Joined Inputs	21
4.11. Pipe-Notation	22
4.12. Tonnensymbol	22
4.13. Provider	23
4.14. Programmstart und Programmende	23
4.15. UI	23
4.16. Zugehörigkeit zu einer Klasse	24
5.1. IODA Architektur, Quelle:[IODA]	26
5.2. FormatAndPrintStrings Variante 1 - Flow Design	31
5.3. FormatAndPrintStrings Variante 2 - Flow Design	32
5.4. FindTheAnswer - Flow Design, Quelle: [CODEWHISPERER]	37
5.5. Ein Ausgang mehrere Empfänger	39
5.6. Eine Funktionseinheit mit mehreren Ausgängen	40
6.1. System-Umwelt-Analyse	46
9.1. Die Hauptansicht	56
9.2. Ablauf einer Erstellung einer neuen Funktionseinheit	57
9.3. Eine neue Funktionseinheit wurde erstellt	57
9.4. Datentypen Editor	60
10.1. Dexel	62
10.2. Aufbau der Assembly-Struktur und Abhängigkeiten zwischen den Dexel-Unterprojekten	63
10.3. Das Domänenmodell von Dexel	64
10.4. Dexel-Screenshot: FunctionUnit-View	65
10.5. Dexel-Screenshot: DataStreamDefinition-View	66
10.6. Dexel-Screenshot: DataStream-View	66
10.7. Dexel-Screenshot: CustomDataType-View	67
10.8. Erstellen einer neuen Funktionseinheit	69

10.9. Textfeld innerhalb der Funktionseinheit	69
10.10. Syntaxhighlighting	70
10.11. Hinzufügen eines neuen Outputs	70
10.12. Verbinden von Funktionseinheiten	70
10.13. Erstellen von Integrationen	72
10.14. Der Editor überprüft, ob für <i>Address</i> ein Datentyp definiert wurde.	73
10.15. Dexel - Warnung- und Fehler-Darstellung	77
10.16. Aufgabe - CSV Tabellieren	77
10.17. CSV Tabellarisieren - Flow Design aus YouTube Video	78
10.18. CSV Tabellarisieren - Datentypen	78
10.19. CSV Tabellarisieren - Dexel Flow Design	79
10.20. Konsolenausgabe - ShoppingSimulator	82
10.21. Shopping Simulator	83
10.22. Datentyp der generiert werden soll	87
10.23. Dexel-Screenshot: Einfache Funktionseinheit mit benutzerdefinierten Da- tentyp	88
10.24. Dexel-Screenshot: Funktionseinheit mit Tupel-Output	89
10.25. Dexel-Screenshot: Funktionseinheit mit zwei Outputs mit definierten Ac- tionnamen	90
10.26. Dexel-Screenshot: Funktionseinheit mit Stream als Ausgang	91
10.27. Dexel-Screenshot: Funktionseinheit mit eingehendem und ausgehendem Stream und undefinierten Actionnamen	91
10.28. Generate Integration Body - Flow Design	93

Tabellenverzeichnis

2.1. CCD Prinzipien	11
2.2. Weitere Prinzipien	13
5.1. IOSP Übersicht	29
5.2. LINQ - Sequenzen Modifizieren	35
5.3. LINQ - Sequenzen Filtern	35
5.4. LINQ - Sequenzen Überprüfen	36
5.5. LINQ - Berechnungen	36
5.6. LINQ - Überspringen und Nehmen	37
8.1. Anforderungen mit hoher Priorität	51
8.2. Anforderungen mit mittlerer Priorität	52
8.3. Anforderungen mit niedriger Priorität	52
8.4. Anforderungen der Code-Generierung	53
8.5. Anforderungen der Diagramm-Generierung	54