

MIT

Flow Design

ZU

CLEAN CODE

STEFAN LIESER



Impressum

Stefan Lieser, Subbelrather Str. 540, 50827 Köln, stefan@lieser-online.de
“Mit Flow Design zu Clean Code”

© 2020 Stefan Lieser
Alle Rechte vorbehalten

Das Werk, einschließlich seiner Teile, ist urheberrechtlich geschützt. Jede Verwertung ist ohne Zustimmung des Autors unzulässig. Dies gilt insbesondere für die elektronische oder sonstige Vervielfältigung, Übersetzung, Verbreitung und öffentliche Zugänglichmachung.

Inhaltsverzeichnis

Einführung	4
Entstehung von Flow Design	6
Warum Flow Design?	9
Flow Design am Beispiel	19
Interaktion: Start	28
Interaktion: NextPage	35
Interaktion: PrevPage	42
Interaktionen: FirstPage und LastPage	48
Teil I	49
Anforderungen zerlegen	53
Der Übergang von den Anforderungen zum Entwurf	65
Flow Design: Datenflussdiagramme	69
Verfeinerung des Entwurfs durch Zerlegung	76
Der Übergang vom Entwurf zur Umsetzung	101
Nachrichten	105
Beispiel WordCount	110
Zustand innerhalb einer Interaktion	123
Zustand über Interaktionen hinweg	131
Beispiel TicTacToe	136
Zustand im Portal	150
Verwendung von Ressourcen	154
Beispiel WordWrap	158
Datenmodelle	168
Abhängigkeiten	172
Zerlegung der UI	189
Beispiel Questionnaire	203
Streams	237
Beispiel LOCcount	249
Fallunterscheidungen	258
Alternativen wieder zusammenführen	266
Beispiel Haushaltsbuch	269
Asynchrone Ausführung	280
Beispiel Wecker	286

Prinzipien	304
Teil II	321
Syntax	322
Aspekte	324
Datenflüsse und Nachrichten	327
Abhängigkeiten	334
Zustand	336
Split	339
Join	341
Map	344
Fallunterscheidung	346
Verfeinerung	348
Klassennamen	352
Nebenläufigkeit	356
Vorgehensweise	360
Anforderungen	363
Entwurf	369
Arbeitsorganisation	376
Implementation	378
Code Review	379
Abnahme durch den Product Owner	380
Übersetzung in Code	381
Eingehende Datenflüsse	382
Ausgehende Datenflüsse	385
Split	396
Hierarchien	397
Fallunterscheidungen	402
Join	405
Zustand	406
UI	409
Glossar	411

Einführung

Durch sein Buch *Clean Code* hat Bob C. Martin, vielen als *Uncle Bob* bekannt, eine große Bewegung gestartet. Plötzlich diskutieren Entwickler über Prinzipien wie *Single Responsibility Principle* (SRP) oder *Don't Repeat Yourself* (DRY). Die SOLID Prinzipien sind plötzlich jedem bekannt, auch wenn sie schon in einem seiner früheren Bücher genannt werden. Auch mich hat das Buch lange beschäftigt und mein Blick auf Softwareentwicklung wurde dadurch nochmal stark verändert. Und doch blieb für mich immer eine Frage offen: Wozu sollen Entwickler all diese Prinzipien einhalten? Vordergründig ist die Antwort schnell gefunden. Manchen geht es um *Qualität*, anderen um *Professionalität*, wieder anderen um *Handwerkskunst*. Doch was haben alle diese Beweggründe gemeinsam? Worauf lässt sich *Clean Code* reduzieren? Für mich lautet die Antwort:

Verbesserungen am Softwareentwicklungsprozess müssen die Linearisierung des Aufwands zum Ziel haben.

Die folgende Abbildung zeigt, wie sich der Aufwand für die Realisierung eines Features typischerweise über die Zeit entwickelt: er steigt exponentiell an.

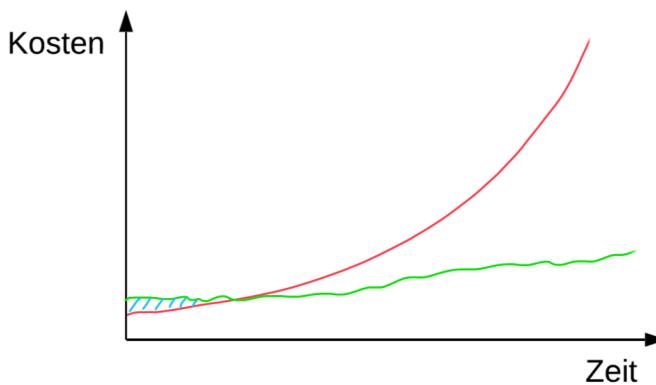


Abbildung 1: Anstieg des Aufwands im Zeitverlauf

Ein Feature ist zu Beginn des Projekts deutlich schneller realisiert, als in einer späteren Phase. Dieses Phänomen lässt sich gewiss nicht vollständig vermeiden. Zu Beginn ist die Codebasis überschaubar, verfügt nur über einige Tausend Zeilen Code. Da ist ein Feature schneller realisiert, als in einer Codebasis mit Hunderttausenden oder gar Millionen Zeilen Code. Insofern ist von einem Anstieg des Aufwands auch zukünftig auszugehen.

Doch es muss mit aller Kraft verhindert werden, dass der Anstieg exponentiell erfolgt. Dann landet das Projekt nämlich zwangsläufig irgendwann in einer Phase, in der Änderungen, Ergänzungen oder Fehlerbehebungen praktisch ausgeschlossen sind.

Die angestrebte linear ansteigen Kurve hat zwei Merkmale: erstens sind An- und Abstiege erkennbar. Das liegt daran, dass kurzzeitig sogenannte *technische Schulden* angesammelt werden, die dann jedoch durch Refactoring Maßnahmen bereinigt werden. Natürlich ist es völlig in Ordnung, aus Sicherheitsgründen kurz vor einem Release keine größeren Refactorings mehr zuzulassen. Doch nach dem Release muss dann aufgeräumt werden. Andernfalls droht erneut der exponentielle Anstieg.

Die zweite Beobachtung: zu Beginn des Projekts ist ein höherer Aufwand erforderlich, wenn man versucht, alles "richtig" zu machen. Nach den Regeln der Kunst zu arbeiten bedeutet, sich ständig weiterzubilden, Dinge auszuprobieren und zu üben. Sofort auf Qualität zu setzen kann somit auch bedeuten, anfangs etwas langsamer voranzukommen. Doch der Break Even, also der Punkt, an dem sich der Aufwand bezahlt macht, ist sehr schnell erreicht. Meine Überzeugung ist, dass kein Weg daran vorbei geht, Softwareprojekte immer und von Anfang an nach den Regeln der Kunst zu bearbeiten.

An diesem Punkt zeigt sich eine weitere Herausforderung: was sind denn die Regeln der Kunst? Gibt es in der Softwareentwicklung anerkannte Regeln, die von jedem Team befolgt werden? Ich fürchte, die Antwort lautet nein. So lange Entwickler zu Beginn eines jeden Projekts diskutieren, wie denn nun dieses Mal gearbeitet werden soll, sind wir noch weit weg von einem anerkannten Regelsatz. Dieses Buch möchte unter anderem dazu einen Beitrag leisten, eine standardisierte Vorgehensweise zu etablieren.

Doch egal, was Sie lieber Leser in Ihren Projekten tun oder nicht tun: jedes einzelne Prinzip, jede einzelne angewandte Praxis muss sich der Frage unterwerfen, ob sie zur Linearisierung des Aufwands beiträgt.

Entstehung von Flow Design

Clean Code

Der Ursprung dieses Buchs über *Flow Design* geht auf das Thema *Clean Code* zurück. Mein Kollege *Ralf Westphal* und ich haben im Jahr 2008 zufällig das gleiche Buch gelesen: das schon genannte Buch *Clean Code* von Bob C. Martin. Aus einem Telefonat, in dem wir feststellten, dass in diesem Buch viele wichtige Prinzipien behandelt werden, entwickelte sich unsere *Clean Code Developer Initiative*¹. Wir haben seinerzeit in einem Wiki über 40 Prinzipien und Praktiken zusammengetragen, die zu “besserem Code” führen. Damals war uns noch nicht in allen Einzelheiten klar, was “besserer Code” im Detail bedeutet. Es brauchte einige Zeit des Diskutierens, Ausprobierens, Forschens und Lehrens, bis wir beim Wert der *Wandelbarkeit* angekommen waren. Software muss wandelbar sein. Die Wandelbarkeit stellt den Investitionsschutz dar, den unsere Kunden von uns erwarten. Unsere Kunden möchten auch in 10, 20 oder 30 Jahren noch mit weiteren Ergänzungswünschen zu uns kommen, ohne dass wir als Entwickler jedesmal stöhnen, dass dies aber nun eigentlich nicht mehr geht, weil der Code doch so undurchsichtig ist.

Erinnern Sie sich noch an das Jahr-2000-Problem? Softwareentwickler konnten sich in den ’50er, ’60er und ’70er Jahren nicht vorstellen, dass ihre Software jemals einen Jahrtausendwechsel erleben würde. Teilweise laufen diese “alten Schinken” immer noch. Und bereiten immer noch große Probleme bei der Wandelbarkeit. Denn mangelnde Wandelbarkeit dürfte der Grund dafür sein, dass Passwörter in manchen Bankensystemen noch auf 5 Zeichen begrenzt sind. Niemand wird dort ernsthaft glauben, dass dies kein Sicherheitsproblem sei. Doch man ist schlicht nicht in der Lage, eine entsprechende Änderung vorzunehmen, weil diese das System möglicherweise in einen kritischen Zustand versetzen würde, was im Zweifel die Unsicherheit noch erhöhen würde. Wandelbarkeit hat also einen sehr langen zeitlichen Horizont.

Für wandelbare Software müssen Prinzipien eingehalten und Praktiken angewandt werden. Das ist die Aussage der Clean Code Developer Initiative. Das ist zwar alles gut und richtig, doch anfangs fehlte uns im Bereich der Praktiken Wesentliches.

¹ <http://clean-code-developer.de> (deutsch) sowie <http://clean-code-developer.com> (englisch)

In unseren Workshops fühlten wir uns mit den Prinzipien und Praktiken allein nicht wohl. Wir konnten unseren Teilnehmern zwar im Code Review mitteilen, welche Prinzipien verletzt waren. Und wir konnten ihnen Praktiken wie automatisiertes Testen ans Herz legen. Doch anfangs konnten wir ihnen nur wenig methodisches Vorgehen an die Hand geben, um so Prinzipienverletzungen von vorne herein zu vermeiden. Genau danach haben wir dann geforscht: wir haben eine Methode gesucht, mit der die Lücke von den Anforderungen zum Code geschlossen werden kann. Denn es ist schnell klar, dass drauf-los-codieren sicher nicht zu wandelbarer Software führt.

Heute wissen wir, dass die Anforderungen systematisch zerlegt werden müssen. Für einen ausreichend kleinen Ausschnitt an Anforderungen muss jeweils der Entwurf einer Lösung erstellt werden, bevor die Lösung dann in Form von Programmcode umgesetzt werden kann.

Heute ist unsere Methode “fertig”. So fertig, wie etwas im IT Umfeld fertig sein kann. Bekanntlich ist nichts beständiger als der Wandel. Die Methode ist erprobt und ausgereift. Wir haben die Methode an viele Workshopteilnehmer weitergegeben, sie mit ihnen ausprobiert und angewandt. Wir haben in ungezählten Artikeln ein Beispiel nach dem anderen mit der Methode bearbeitet. Wir haben Teams begleitet, welche die Methode im Produktiveinsatz anwenden. Und nicht zuletzt wenden viele Entwickler die Methode an, ohne dass wir sie im einzelnen kennen oder begleitet haben. Davon erfahren wir immer wieder, wenn Entwickler uns auf Konferenzen ansprechen. Insofern sind wir ganz sicher: *Flow Design* ist Erwachsen geworden und es ist Zeit, die Methode mit diesem Buch in die Welt zu tragen.

Wer sind "wir"?

Im Buch werden Sie an einigen Stellen davon lesen, dass “wir” *Flow Design* entwickelt haben, dass “wir” diese und jene Erfahrung gemacht haben. Mit “wir” sind Ralf Westphal und der Autor Stefan Lieser gemeint. Wir haben Anfang 2009 unser erstes Clean Code Developer Training gemeinsam geplant und durchgeführt. Ursprünglich hatten wir den Plan, nur das erste Training gemeinsam zu halten, um so gemeinsam das Konzept zu überprüfen und zu verbessern. Nach der ersten Beta-Schulung zu zweit wollten wir die Trainings eigentlich einzeln durchführen. Doch bereits nach einem halben Tag wurde uns klar, dass es viele Vorteile hat, Trainings zu zweit durchzuführen. So haben wir an diesem Konzept über mehrere Jahre festgehalten und Trainings im Team-Teaching gehalten. Während dieser Trainings sind viele der hier beschriebenen Ideen und Konzepte entstanden. Immer wieder haben wir uns am Abend nach dem Training zusammengesetzt und unsere Methode diskutiert, verfeinert, modifiziert und ausprobiert. Immer wieder sind wir gemeinsam zu Fuß vom Hotel zum Kunden gegangen.

gen, um gemeinsame Zeit zum Diskutieren zu haben. Und immer wieder haben wir uns zu Retreats zurückgezogen, um in aller Ruhe an unserer Vorgehensweise feilen zu können. Das Ergebnis ist *Flow Design* als Entwurfsmethode aber auch als Vorgehensmodell für den Softwareentwicklungsprozess.

Warum Flow Design?

Requirements Logic Gap

Als Entwickler stehen wir Tag für Tag vor der gleichen Herausforderung: wie gelangen wir von den Anforderungen zum Code? In besonders einfachen Fällen mag es möglich sein, mit einem beherzten Sprung über den schmalen Graben zu springen. In realen Projekten ist der Abstand allerdings zu groß, um sofort von den Anforderungen zum Code zu springen. Es braucht allermindestens einen Entwurf der Lösung.

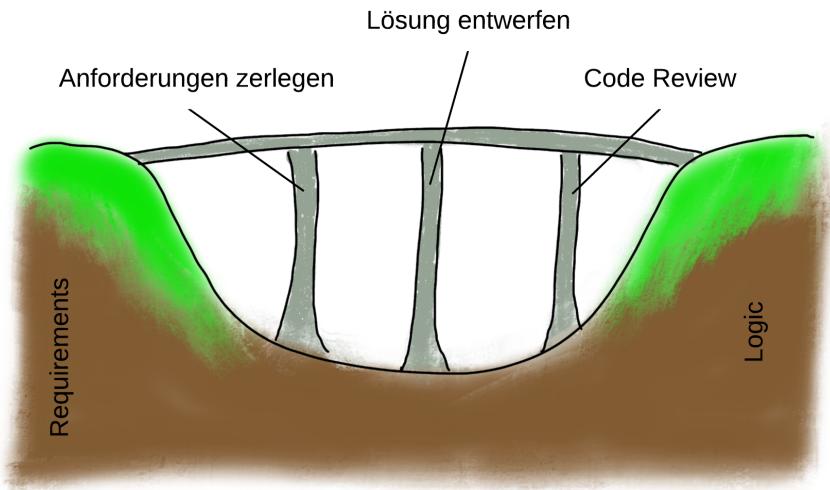


Abbildung 2: Requirements Logic Gap

Mit dem Aufkommen der agilen Methoden wie *Extreme Programming* (XP), *Scrum* und *Kanban* hat sich die Erkenntnis durchgesetzt, dass in *Iterationen* und in *Inkrementen* vorgegangen werden muss. Es werden also nicht mehr alle Anforderungen auf einmal weiter bearbeitet, sondern jeweils nur ein Ausschnitt. Das Vorgehen in Form des sogenannten *Wasserfalls* hat sich als ungeeignet erwiesen.

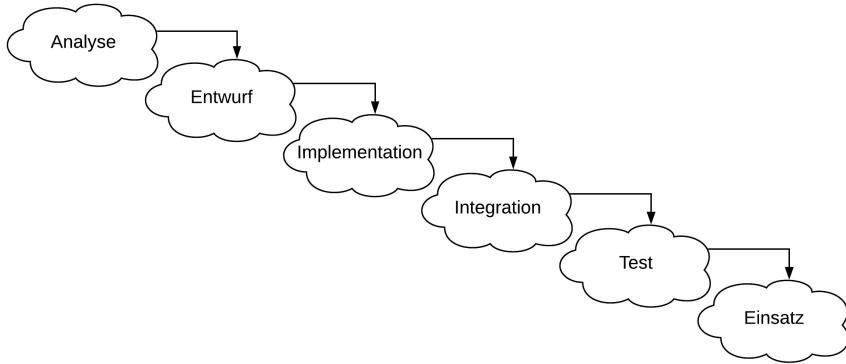


Abbildung 3: Wasserfall

Der Versuch, ein Softwareentwicklungsprojekt in Phasen zu zerlegen, war naheliegend und wichtig für die Weiterentwicklung der Informatik. Doch inzwischen haben wir gelernt, dass die Feedbackzyklen sehr viel kürzer sein müssen, um ein Projekt zum erfolgreichen Abschluss zu führen.

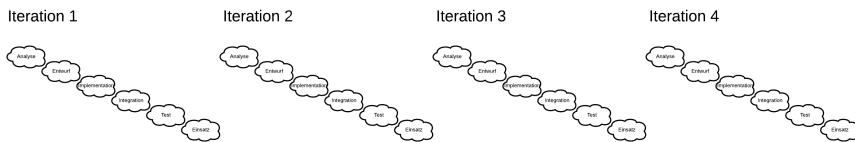


Abbildung 4: Iteratives Vorgehen

Durch iteratives Vorgehen wird häufiger und schneller Feedback generiert. Der *Product Owner* erhält regelmäßig Ausschnitte des Softwaresystems geliefert, statt das gesamte Softwaresystem auf einmal am Ende des Projekts. So wird er in die Lage versetzt, den Entwicklern kontinuierlich Rückmeldung darüber zu geben, ob die Anforderungen richtig umgesetzt wurden. Ferner eröffnet dies die Möglichkeit herauszufinden, ob es sich um die richtigen Anforderungen gehandelt hat. Oftmals kann erst dadurch, dass Teile des Systems realisiert werden, herausgefunden werden, was genau die Anforderungen sind. Im Extremfall von wasserfallartigem Vorgehen, kann dies nur am Ende nach vollständiger Lieferung festgestellt werden und dann ist es zu spät für Korrekturen. Kurze Iterationen begünstigen somit Änderungen und Flexibilität.

Da iteratives Vorgehen so wichtig und wertvoll ist, weil es das Potential von häufigem und kurzfristigem Feedback eröffnet, haben wir es 2008 in die Praktiken der Clean Code Developer Initiative aufgenommen. Die Praktik

findet sich dort im blauen Grad². Erst später ist uns klar geworden, dass iteratives Vorgehen unbedingt durch die Arbeit in *Inkrementen* ergänzt werden muss. Ein Inkrement stellt einen vertikalen Schnitt durch das Softwaresystem dar. Ziel ist es, ein für den Anwender greifbares Stück Software zu liefern.

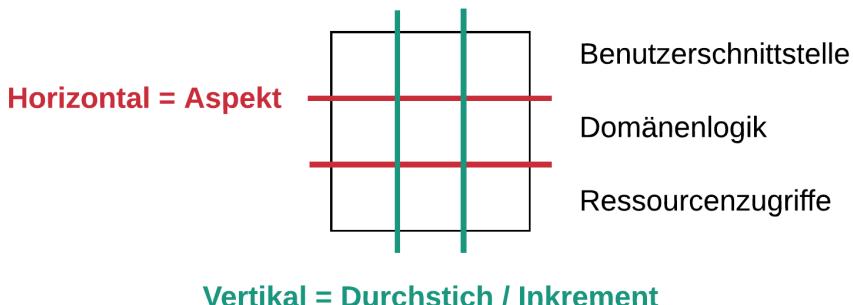


Abbildung 5: Vertikal vs. horizontal schneiden

Nur durch vertikales Schneiden entstehen mit jeder Iteration Ausschnitte des Systems, die vom Product Owner bewertet werden können. In einem Inkrement wird ein *Feature* realisiert. Das Team realisiert einen Teil der Benutzerschnittstelle, einen Teil der Domänenlogik sowie einen Teil der Ressourcenzugriffe. Das Ergebnis ist ein Stück lauffähige Software. Dieser Ausschnitt des Systems kann installiert und ausprobiert werden, ist greifbar. Der Product Owner kann diese Software bedienen und sein Feedback geben. Scrum spricht in diesem Zusammenhang vom *Potentially Shippable Product*, dem potentiell lieferbaren Produkt.

Wird dagegen horizontal geschnitten, realisiert das Team in einer Iteration einen Teilaspekt des Systems. Diese Funktionalität ist nicht eigenständig installierbar und kann auch nicht vom Product Owner bedient werden. Es kann höchstens ein spezieller Prüfstand geschaffen werden, auf dem der Product Owner die realisierte Funktionalität begutachten kann. Realisiert das Team beispielsweise einen Persistenzmechanismus und verwendet dazu eine Datenbank, kann weder der Product Owner noch das Team mit Gewissheit sagen, ob die realisierte Funktionalität korrekt ist. Ohne nämlich vertikal von den Anforderungen her zu kommen, ist nicht klar, wie genau die Schnittstelle zur realisierten Funktionseinheit später benötigt wird. Nur konsequentes vertikales Schneiden und damit ein Arbeiten in Inkrementen, stellt sicher, dass vom Product Owner Feedback gegeben werden kann.

² http://clean-code-developer.de/die-grade/blauer-grad/#Iterative_Entwicklung

Halten wir also fest, dass die Lücke zwischen Anforderungen und Code in jedem Fall in Form von vertikalen Schnitten geschlossen werden muss. Doch es bleibt die Frage offen, welche Arbeitsschritte dabei hilfreich sind.

Die Frage wurde im *Extreme Programming* mit *TDD* beantwortet: *Test Driven Development* soll sicherstellen, dass die Anforderungen erstens korrekt und zweitens wandelbar implementiert werden. Die Korrektheit der Implementation wird bei TDD dadurch hergestellt, dass automatisierte Tests geschrieben werden. Durch Analyse werden aus den Anforderungen Testfälle abgeleitet und diese in geeigneter Reihenfolge *test-first* implementiert. Es wird als jeweils erst ein Test geschrieben und anschließend gerade so viel implementiert, dass dieser Test grün wird. Auch *test-first* haben wir als Praktik im blauen Grad³ der Clean Code Developer Initiative aufgenommen, TDD jedoch ganz bewusst nicht. Das hat seinen Grund. Wir können nämlich nicht beobachten, dass TDD sein zweites Versprechen einlöst, wandelbaren Code zu produzieren.

Die Idee von TDD ist, getrieben durch die Tests, eine Implementation entstehen zu lassen, die testbar ist. Dieser Punkt wird erreicht. Aber der Anspruch, dass die dabei entstehenden Codestrukturen "clean" sind, wird unserer Beobachtung nach regelmäßig verfehlt. TDD mag für einfache algorithmische Aufgabenstellungen geeignet sein. Für den Entwurf ganzer Softwaresysteme ist allerdings mehr erforderlich: Nachdenken! Zwischen den Anforderungen und dem Code, der diese umsetzt, liegt eine Lösungsidee. Die Lösung des Problems muss entworfen werden. Das Entwerfen der Lösung geschieht bei TDD verwoben mit dem Umsetzen der Lösung. Der Entwickler oder das Pair schreibt einen Test, denkt kurz nach, und implementiert dann. Damit sind Entwurf der Lösung und Umsetzung der Lösung nicht klar genug getrennt. Und das wichtigste: es wird keine für den Entwurf einer Lösung geeignete Visualisierung verwendet. Im Pair Programming sprechen die beiden Entwickler zwar über ihre Lösungsideen, aber sie visualisieren diese nicht in ausreichender Weise. Es wird vollständig auf eine textuelle Darstellung gesetzt, in Form von Quellcode. Eine grafische Darstellung fehlt. Natürlich verbietet XP nicht, auch mal ein Diagramm zu zeichnen. Es gehört aber nicht zur Methodik, ganz konsequent erst die Lösung zu entwerfen und sie erst dann in Quellcode zu gießen. Erst durch konsequentes Entwerfen der Lösung entsteht ein Entwicklungsprozess, der zu wandelbarer Software führt.

Der Entwurf einer Lösung ist ein kreativer Prozess. Kreative Prozesse profitieren davon, von mehreren Personen durchgeführt zu werden. Auf der Ebene von Code, einer textuellen Darstellung, können im Grunde nur zwei Entwickler gut zusammenarbeiten. Mit Beamer geht etwas mehr, aber dann kommt der zweite limitierende Faktor ins Spiel: Code ist nicht flexibel genug. Code muss der sehr peniblen Syntax der verwendeten Programmiersprache

³ http://clean-code-developer.de/die-grade/blauer-grad/#Test_first

genügen. Eine Lösungsidee in Code zu skizzieren, um darüber zu diskutieren, fällt schwer. Ein Diagramm, bestehend aus einer Hand voll Symbole wie Kreise und Pfeile, am Whiteboard zu zeichnen, ist dazu ein geeignetes Werkzeug. Solche Diagramme lassen sich viel leichter verändern als Code. Sie enthalten ganz bewusst nicht alle Details der späteren Implementation. Durch das höhere Abstraktionsniveau wird es leichter, über die Lösung nachzudenken und zu diskutieren.

Der Entwurf kann mit einer Landkarte verglichen werden. In einer Karte sind nicht alle Details der realen Landschaft verzeichnet. Es ist eben gerade nicht jeder einzelne Baum eingezeichnet, sondern eine große Fläche als Wald markiert. Erst durch Weglassen von Details wird es möglich, die Karte zur Navigation zu verwenden. Das gleiche gilt für einen Entwurf. Durch das höhere Abstraktionsniveau ist ein Navigieren in der Lösung möglich, weil bewusst auf Details verzichtet wird. Erst bei der Implementation des Entwurfs, beim Kodieren, sind die Details relevant.

Unser Plädoyer lautet daher: zwischen Anforderungen und Code liegt der Entwurf der Lösung. Und genau darum geht es bei Flow Design. Flow Design ist eine leichtgewichtige Entwurfsmethode, mit der die Lücke zwischen Anforderungen und Code geschlossen werden kann. Mit Flow Design wird jeweils eine Lösung für einen Ausschnitt der Anforderungen entworfen. Die Lösung kann dann im Anschluss implementiert werden. Auf diese Weise sind die Phasen Entwurf der Lösung einerseits und Umsetzung des Entwurfs andererseits, klar getrennt. Ferner werden so für die Phasen angemessene Darstellungsformen verwendet. Der Entwurf wird grafisch dargestellt, während die Implementation textuell erfolgt.

Entwurf, Architektur, Design: eine Begriffsklärung

Bei Anforderungen wird zwischen zwei Arten unterschieden: *Funktionalen Anforderungen* beschreiben, was das System bereitstellen soll. Die *nicht-funktionalen Anforderungen* beschreiben dagegen, auf welche Weise die funktionalen Anforderungen erbracht werden sollen. Ein Beispiel: die funktionalen Anforderungen einer Buchhaltungssoftware beschreiben, welche Buchhaltungsvorgänge mit dem zukünftigen Softwaresystem bearbeitet werden können. Es sollen beispielsweise Zahlungseingänge verbucht werden. Die nicht-funktionalen Anforderungen könnten lauten, dass auf das Softwaresystem von mehreren Standorten über das Internet zugegriffen werden muss, oder dass 1.000 Buchungssätze pro Stunde erfasst werden können.

Für beide Kategorien von Anforderungen, funktionale wie nicht-funktionale, muss eine Lösung entworfen werden. Für den Entwurf der nicht-funktionalen Anforderungen verwenden wir den Begriff *Architektur*. Die Lösung der funktionalen Anforderungen hat in der Literatur keine allgemein

etablierte Bezeichnung. Eine weitere Herausforderung besteht darin, dass es einen Überbegriff für beide Lösungsbereiche geben sollte. In diesem Buch verwenden wir den Begriff *Entwurf* für die Lösung der funktionalen Anforderungen. Der Überbegriff für Architektur und Entwurf ist für uns *Design*. Das Design besteht aus der Architektur und dem Entwurf. Die Architektur beschreibt eine Lösung, für die nicht-funktionalen Anforderungen, während der Entwurf eine Lösung der funktionalen Anforderungen beschreibt.

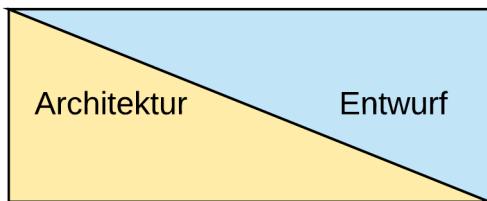


Abbildung 6: Das Design besteht aus Architektur und Entwurf.

Was ist Flow Design?

Flow Design ist eine Entwurfsmethode für funktionale Anforderungen. Entwurfsmethoden dienen ganz allgemein dazu, die Lösung für ein Problem zu beschreiben. Mit Flow Design können Lösungen für funktionale Anforderungen erarbeitet und beschrieben werden. Die folgende Abbildung zeigt ein Beispiel.

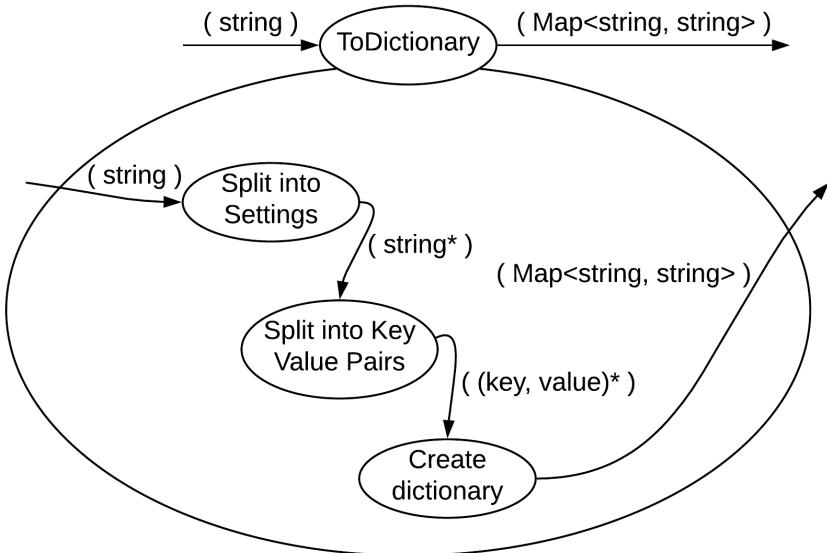


Abbildung 7: Entwurf für *ToDictionary*

Das Diagramm zeigt eine Lösung für das Problem, aus einem *String* der Form

`"a=1 ; b=2 ; c=3"`

ein *Dictionary* mit den Einträgen

`{ {"a", "1"}, {"b", "2"}, {"c", "3"} }`

zu erstellen.

Das Diagramm stellt eine Lösung des *ToDictionary* Problems grafisch dar. Es ist ein Entwurf der funktionalen Anforderungen.

Anforderungen gliedern sich in funktionale und nicht-funktionale Anforderungen. Die funktionale Anforderung für *ToDictionary* ist mit dem oben dargestellten Beispiel angedeutet. Ein *String* soll nach bestimmten Regeln in ein *Dictionary* transformiert werden. In einem realen Softwareentwicklungsprojekt würden viele weitere Beispiele ergänzt werden, um klar auszudrücken, wie genau sich diese Funktion verhalten soll. Eine nicht-funktionale Anforderung könnte lauten, dass diese Funktion ein maximales Zeitlimit nicht überschreiten darf. Auch die Länge der Eingangsdaten oder die zu erwartende Größe des Dictionaries sind Details der nicht-funktionalen Anforderungen. Funktionale und nicht-funktionale Anforderungen werden vom Auftraggeber explizit formuliert.

Es gibt allerdings einen dritten Bereich von Anforderungen, über den in der Regel nicht explizit gesprochen wird: den *Investitionsschutz*. Unsere Auftraggeber erwarten, dass sie langfristig über viele Jahre oder sogar Jahrzehnte Änderungen anbringen können, die wir am bestehenden System

umsetzen. Solche Änderungen ergeben sich in vielen Bereichen aufgrund von notwendigen Anpassungen an gesetzliche Veränderungen. Auch das Beheben von Fehlern gehört zu Änderungen, die der Kunde langfristig erwartet. Es gibt keinen Zeitpunkt, ab dem wir sagen können, dass das Softwaresystem nun nicht mehr modifiziert wird. Dies kann für einzelne Versionen gelten, aber nicht für das System als Ganzes.

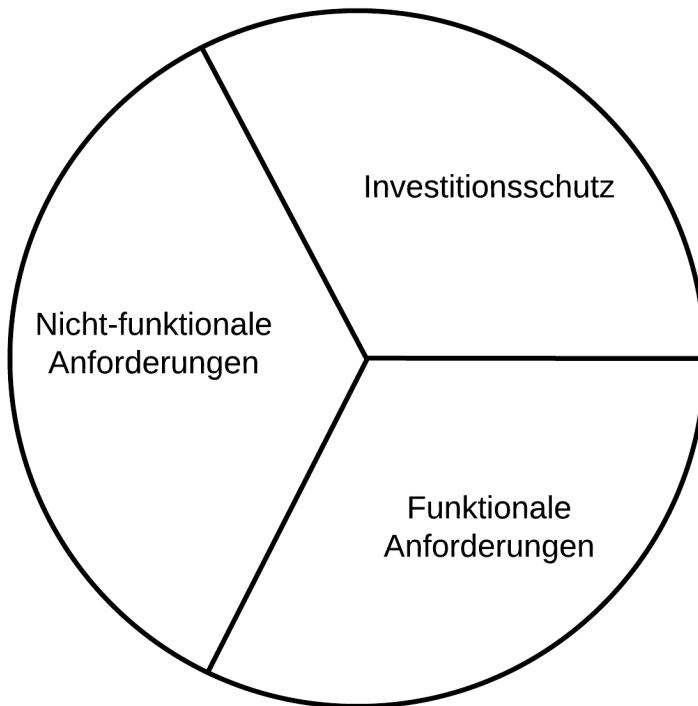


Abbildung 8: Kategorien von Anforderungen

Der Investitionsschutz ist eine *implizite* Anforderung. Auftraggeber sprechen nur selten davon, dass sie das zu erstellende Softwaresystem über Jahre verändern möchten. Und dennoch hat der Auftragnehmer eine Verpflichtung, auch diese unausgesprochene Anforderung umzusetzen. Mindestens im Wettbewerb wird sich andernfalls ein gravierender Nachteil zeigen. Der Investitionsschutz für den Auftraggeber drückt sich somit aus in *Wandelbarkeit*. Wenn das erstellte System wandelbar ist, lässt es sich zukünftig an neue Anforderungen anpassen. Oder umgekehrt formuliert: fehlt die Wandelbarkeit, werden Änderungen, Ergänzungen und Fehlerbe seitigung zunehmend aufwendiger.

Eine wichtige Frage ist, ob eine Entwurfsmethode die Wandelbarkeit explizit im Blick hat. Wandelbarkeit stellt sich nicht von alleine ein. Eine Entwurfsmethode muss zentrale Prinzipien der Wandelbarkeit wie das Trennen von Aspekten unterstützen oder sogar herausfordern. Bei der Entwicklung von Flow Design stand von Anfang an, neben den funktionalen Anforderungen, das Thema Wandelbarkeit im Fokus. Flow Design hat sich aus der Beschäftigung mit den Clean Code Developer Prinzipien, Praktiken und Werten entwickelt. Entwürfe dienen als Grundlage für die Implementation. Wenn es um Wandelbarkeit geht, muss folglich am Ergebnis der Implementation erkennbar sein, dass wichtige Prinzipien eingehalten sind. Prinzipien, die sich positiv auf die Wandelbarkeit auswirken. Insofern muss eine Entwurfsmethode die Basis für Wandelbarkeit legen. Wird die Wandelbarkeit bereits beim Entwurf aus dem Blick verloren, kann sie sich in der nachfolgenden Implementation nur zufällig einstellen.

Während der Entwicklung von Flow Design als Entwurfsmethode für funktionale Anforderungen, haben wir weitere Methoden entwickelt, für den Bereich der nicht-funktionalen Anforderungen. Ferner sind Methoden zur Zerlegung von Anforderungen entstanden, um das konsequente Arbeiten an Inkrementen zu befördern. Das vorliegende Buch behandelt die Entwurfsmethode sowie die Vorgehensweise.

Aufbau des Buches

Das Buch besteht aus zwei Teilen. Im ersten Teil werden die Möglichkeiten von Flow Design an Beispielen dargestellt. Gezeigt werden jeweils Anforderungen, Entwürfe für eine Lösung dieser Anforderungen, sowie eine Implementation der Lösung. Für die Implementation werden unterschiedliche Sprachen wie C#, Java, C++ und JavaScript verwendet. Zusammengekommen repräsentieren diese Sprachen einen großen Teil der heutigen Projekte. Natürlich eignet sich Flow Design auch für eine Umsetzung auf hier nicht genannten Plattformen und Sprachen. Anhand der Beispiele lernt der Leser die verschiedenen Aspekte von Flow Design kennen. Einerseits werden die Beispiele im Verlauf des Buchs umfangreicher, andererseits zeigen sich mit jedem Beispiel andersartige Herausforderungen. So werden nach und nach alle Möglichkeiten von Flow Design vorgestellt.

Im zweiten Teil des Buchs wird Flow Design in Form einer Referenz dargestellt. Der zweite Teil zeigt die gesamte Notation in eher abstrakter Form anstelle von Beispielen. Ferner wird dargestellt, wie die einzelnen Bestandteile von Flow Design konkret mit den Sprachmitteln gängiger Programmiersprachen implementiert werden können. Hierbei wird Wert gelegt auf ein Prinzip der Clean Code Developer Initiative: die Implementa-

tion spiegelt den Entwurf⁴. In der Implementation muss der Entwurf wiedererkennbar sein. Die Implementation darf insbesondere nicht vom Entwurf abweichen. Daher ist es wesentlich, mögliche Übersetzungen des Entwurfs in die Implementation zu kennen, um sich jeweils für eine angemessene Variante entscheiden zu können.

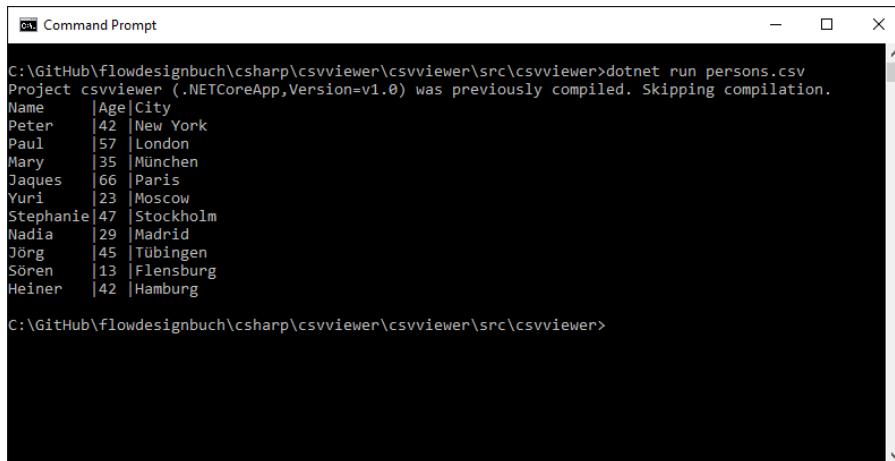
Die Einführung endet nun mit einem ersten Beispiel, mit dem Flow Design vorgestellt wird.

⁴ http://clean-code-developer.de/die-grade/blauer-grad/#Implementation_spiegelt_Entwurf

Flow Design am Beispiel

Anforderungen

Das folgende Beispiel zeigt Flow Design “in Action”. Mit dem Beispiel soll ein Eindruck von der Methode vermittelt werden. Manches Detail wird dabei nicht weiter besprochen, zugunsten des großen Ganzen. Die Details folgen in Teil I mit fokussierten Beispielen, in denen jeweils ein kleiner Ausschnitt der Möglichkeiten von Flow Design besprochen wird. Ferner dient Teil II als Referenz und Nachschlagewerk dazu, weitere Detailfragen zu beantworten. Jede Aufgabenstellung eines Entwicklers beginnt mit den *Anforderungen*. Ohne Anforderungen gäbe es keinen Grund, Software zu erstellen. Somit beginnt auch dieses Einführungsbeispiel mit konkreten Anforderungen: ein CSV Viewer soll erstellt werden. Die folgende Abbildung zeigt, wie sich das Werkzeug auf der Kommandozeile darstellt.



```
Command Prompt

C:\GitHub\flowdesignbuch\csharp\csvviewer\csvviewer\src\csvviewer>dotnet run persons.csv
Project csvviewer (.NETCoreApp, Version=v1.0) was previously compiled. Skipping compilation.
Name    |Age|City
Peter   |42 |New York
Paul    |57 |London
Mary    |35 |München
Jaques  |66 |Paris
Yuri    |23 |Moscow
Stephanie|47 |Stockholm
Nadia   |29 |Madrid
Jörg    |45 |Tübingen
Sören   |13 |Flensburg
Heiner  |42 |Hamburg

C:\GitHub\flowdesignbuch\csharp\csvviewer\csvviewer\src\csvviewer>
```

Abbildung 9: CSV Viewer Kommandozeile

Es handelt sich beim CSV Viewer um ein Kommandozeilenprogramm, mit dem Dateien im CSV Format seitenweise angezeigt werden können. Die Ausgabe erfolgt in Form einer Tabelle, die mit den Möglichkeiten der Konsole formatiert ist. Im Wesentlichen besteht die Anforderung für die Ausgabe darin, die Spaltenbreiten der Tabelle an die jeweiligen Werte der einzelnen Spalten anzupassen. In der Tabelle wird jeweils eine Seite der auszugebenden Datei angezeigt. Die Seitenlänge ist mit 10 Datensätzen

vorgegeben und kann optional über einen weiteren Parameter auf der Kommandozeile überschrieben werden. Die folgende Abbildung zeigt die Ausgabe bei Angabe einer Seitenlänge von 3 Datensätzen.

```
C:\GitHub\flowdesignbuch\csharp\csvviewer\csvviewer\src\csvviewer>dotnet run persons.csv 3
Project csvviewer (.NETCoreApp,Version=v1.0) was previously compiled. Skipping compilation.
Name |Age|City
Peter|42|New York
Paul |57|London
Mary |35|München

F)irst, P)rev, N)ext, L)ast, E)xit
n
Name |Age|City
Jaques |66|Paris
Yuri |23|Moscow
Stephanie|47|Stockholm

F)irst, P)rev, N)ext, L)ast, E)xit
```

Abbildung 10: CSV Viewer mit Seitenlänge 3

Unterhalb der Tabelle wird jeweils ein Menü angezeigt. Per Tastendruck kann nun seitenweise in der Tabelle geblättert werden. Mit *P* wie *Previous* wird eine Seite zurück geblättert, mit *N* wie *Next* dagegen eine Seite weiter. Ferner kann mit *F* wie *First* auf die erste Seite gesprungen werden und mit *L* wie *Last* auf die letzte Seite. Durch *E* wie *Exit* wird das Programm verlassen.

Dies sind in Kürze die funktionalen Anforderungen. In unseren Workshops haben wir mehr als einmal beobachtet, dass Entwickler, spätestens nach einer kurzen didaktischen Pause, sofort beginnen wollen, die Aufgabe zu implementieren: Notebookdeckel auf, coden! Das Beispiel erscheint ihnen einerseits klein genug, um sofort von den Anforderungen zum Code zu springen. Andererseits ist dies ihre gewohnte Arbeitsweise. Nur die wenigsten Teams entwerfen eine Lösung vor ihrer Umsetzung. Doch bevor wir zum Entwurf kommen, muss der Blick erneut auf die Anforderungen gerichtet werden. Bislang wurden nämlich nur *funktionale* Anforderungen geschildert. Bevor die *nicht-funktionalen* Anforderungen klar sind, kann keine angemessene Lösung entworfen werden. Sie werden mir zustimmen, dass sich die Lösung deutlich unterscheidet, je nachdem, wie groß die CSV Dateien sind, die zur Anzeige gebracht werden sollen. Lautet die nicht-funktionalen Anforderung, dass die Dateien nur wenige Kilobyte bis ein paar Megabyte groß sind, spricht nichts dagegen, den Dateiinhalt vollständig in den Hauptspeicher einzulesen. Sind die Dateien dagegen groß, muss eine andere Lösung her. "Groß" könnte bedeuten, dass die Dateien doppelt so groß sind, wie der zur Verfügung stehende Hauptspeicher. Eine Lösung, die

darauf basiert, den Dateiinhalt in den Speicher zu lesen, muss bei dieser Dateigröße scheitern. Das wissen wir bereits, ohne diese Lösungsidee weiter zu konkretisieren oder gar zu implementieren. Es ist also schnell erkennbar, dass die nicht-funktionalen Anforderungen eine wesentliche Rolle beim Entwurf einer Lösung spielen.

Für den weiteren Verlauf des Beispiels soll die Frage nach der Dateigröße so beantwortet sein, dass die Dateien klein sind, also im Bereich einiger Kilobyte bis zu wenigen Megabytes. Später wird die Frage zu beantworten sein, was Wandelbarkeit in Bezug auf diese Anforderung bedeutet. Welche Vorsorge können wir treffen, um auch mit einer Änderung dieser nicht-funktionalen Anforderung umgehen zu können, ohne das gesamte Programm neu schreiben zu müssen? Eine kurze Antwort sei vorweggenommen: Aspekte müssen getrennt werden.

Zerlegung der Anforderungen

Gehen wir nun davon aus, dass die Anforderungen klar definiert sind, dann stellt sich die Frage, wie wir in den Entwurf einer Lösung einsteigen können. Typischerweise sind die Anforderungen zu umfangreich, um gleich "für alles" eine Lösung zu entwerfen. Wir müssen die Anforderungen vertikal zerlegen, in Inkrementen vorgehen. Es sprengt den Rahmen dieses Buchs, das Thema Anforderungszerlegung vollständig zu behandeln. Im weiteren wird daher nur die unterste Ebene der Zerlegungshierarchie behandelt, bestehend aus Dialogen, Interaktionen und Features. Allerdings ist dies auch der wichtigste Bereich aus der Domänenzerlegung, da er sozusagen täglich im Entwickleralltag relevant ist. Die weiter oben liegenden Zerlegungen fallen eher in den Bereich von Architektur. Die folgende Abbildung zeigt die vollständige Hierarchie der Domänenzerlegung, um den Kontext zu geben.

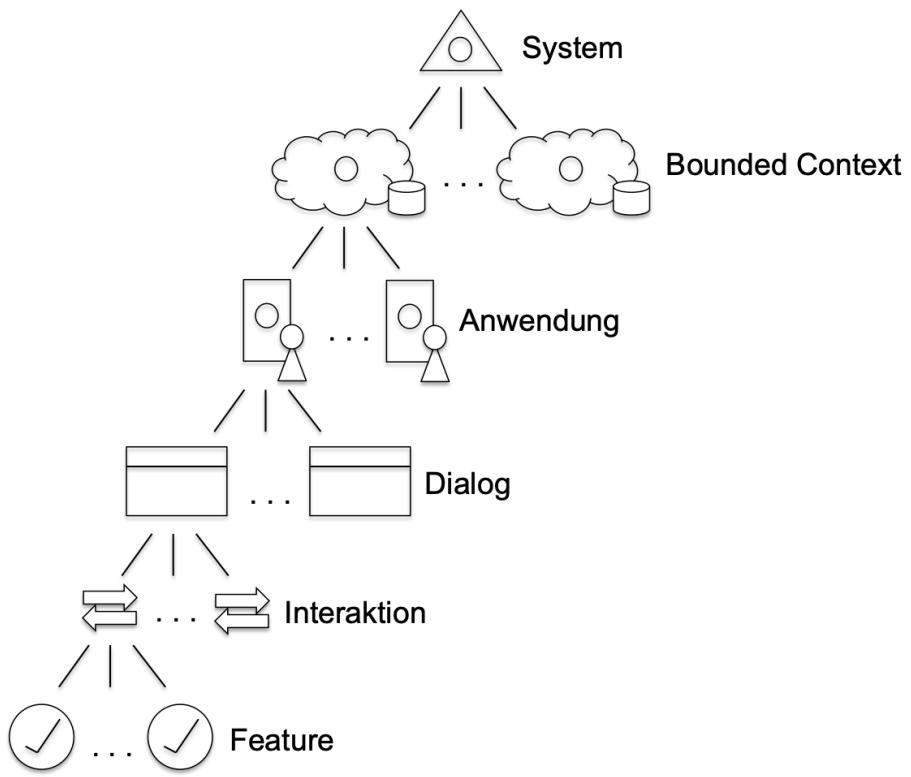


Abbildung 11: Domänenzerlegung, vollständige Hierarchie

Für das Beispiel CSV Viewer genügt eine Zerlegung der Anforderungen in einen Dialog mit mehreren Interaktionen. Die folgende Abbildung zeigt den Dialog der Anwendung mit den Interaktionen.

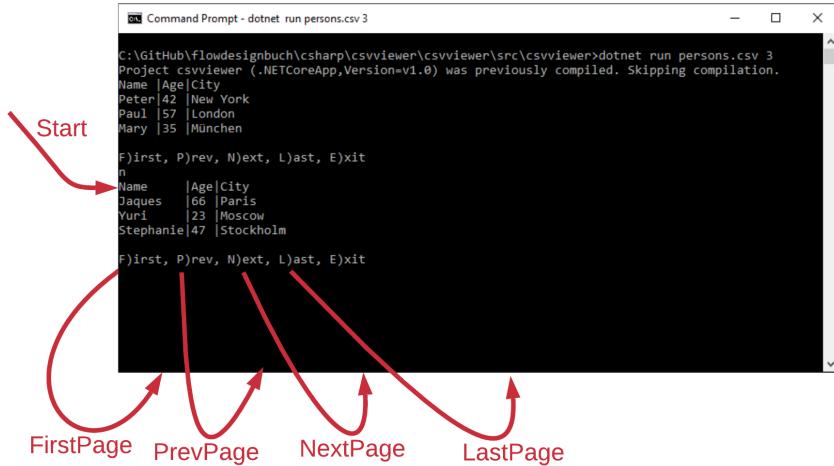


Abbildung 12: Interaktionsdiagramm: der CSV Viewer Dialog mit seinen Interaktionen

Der Anwender eines Softwaresystems interagiert auf verschiedenartige Weise mit dem System. Bei Programmen mit grafischer Benutzeroberfläche (GUI) kann der Anwender beispielsweise Schaltflächen betätigen oder Menüpunkte auswählen. Hinter jeder dieser Interaktionen erwartet er eine Reaktion des Systems. Als Entwickler müssen wir für jede Interaktion *Domänenlogik* implementieren, um das in den Anforderungen beschriebene Verhalten des Systems zu realisieren. Interaktionen sind somit für uns der Grund, Domänenlogik zu realisieren und damit der Ausgangspunkt für den Entwurf.

Definition: **Domänenlogik** - Logik, die das Thema des Softwaresystems betrifft.

Jede einzelne Interaktion stellt einen vertikalen Schnitt durch die Anforderungen dar. Somit ist jede Interaktion ein möglicher Einstiegspunkt in den Entwurf. Am Beispiel der Interaktion *Start* soll dies verdeutlicht werden. Der Product Owner kann entscheiden, dass das Team mit der Realisierung von *Start* beginnen soll. Das mag naheliegend erscheinen oder sogar als die einzige Möglichkeit gelten. Doch er könnte genau so gut entscheiden, dass *Last Page* die erste zu realisierende Interaktion sein soll. Als Entwickler müssten wir dann möglicherweise Teile des Systems in Form von Attrappen realisieren. Grundsätzlich ist der Product Owner frei in seiner Auswahl der Interaktion für die nächste Iteration. Er wählt aus, was ihm den größten Nutzen bringt.

Entwurf der obersten Ebene - in die Breite

Nachdem durch Analyse der Anforderungen die Dialoge und Interaktionen identifiziert sind, kann mit dem Entwurf einer Lösung begonnen werden. Bei Vorliegen mehrerer Interaktionen bestehen zwei Möglichkeiten: es kann in die Breite oder die Tiefe entworfen werden. In die Breite zu entwerfen bedeutet, für jede Interaktion einen Entwurf zu erstellen. Jeder einzelne dieser Entwürfe bleibt jedoch oberflächlich. Beim Entwurf in die Tiefe wählen wir zunächst eine einzelne Interaktion aus und entwerfen diese dann vollständig im Detail, so dass anschließend eine Implementation möglich ist. Befasst sich das Team erstmalig mit einem Dialog, wird es die darin identifizierten Interaktionen zunächst in der Breite entwerfen. Dabei zeigen sich häufig Aspekte, die über mehrere Interaktionen hinweg relevant sind. Im Anschluss entscheidet der Product Owner, welche Interaktion als nächste realisiert werden soll. Das Team entwirft dann diese eine Interaktion in der Tiefe. Bevor weitere Interaktionen im Detail entworfen werden, sollte der gerade erstellte detaillierte Entwurf unbedingt erst implementiert werden. Ferner muss der Product Owner im Anschluss zu diesem Inkrement sein Feedback geben, es abnehmen. Erst danach ist es sinnvoll, eine weitere Interaktion zu verfeinern. Dieses Vorgehen minimiert das Risiko, Entwürfe "auf Halde" zu produzieren. Stellt sich nämlich nach der Umsetzung der ersten Interaktion heraus, dass es Missverständnisse bei den Anforderungen gab oder ändert der Product Owner die Anforderungen, können diese Erkenntnisse in den nächsten Entwurf bereits mit einfließen.

Die folgende Abbildung zeigt die Entwürfe auf der obersten Ebene für die Interaktionen des CSV Viewers.

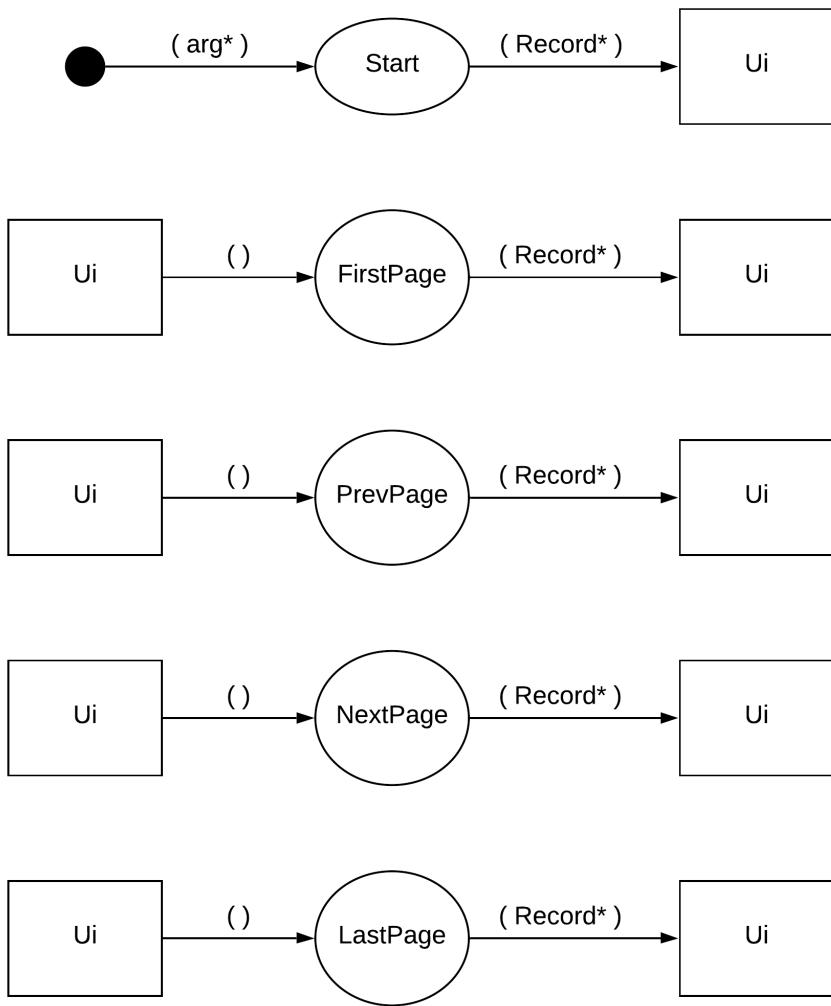


Abbildung 13: CSV Viewer Interaktionen: Entwürfe auf oberster Ebene

In Flow Design stehen Rechtecke, Kreise und Dreiecke für *Funktionseinheiten*. Funktionseinheit ist der Überbegriff für etwas, das eine gewisse Funktionalität enthält. Praktisch gesehen werden Funktionseinheiten beim Übergang in die Implementation als Methode oder Klasse implementiert. Als weiteres Symbol wird der Pfeil in Flow Design verwendet für einen *Datenfluss*. Am Pfeil wird in Klammern notiert, welche Daten fließen. Dies bezeichnen wir auch als *Nachricht* oder *Message*. So fließt im Beispiel oben von der Funktionseinheit *Start* zur *Ui* eine Aufzählung von *Record* Objekten. Dass es nicht ein einzelner Record, sondern mehrere sind, ist am Stern

erkennbar. Der Stern bedeutet bei Daten eine Aufzählung, also “viele davon”. Beim Übergang vom Entwurf in die Implementation muss jeweils entschieden werden, durch welchen Typ der verwendeten Sprache die Aufzählung repräsentiert wird. Das können Arrays oder Listen sein, aber auch Interfaces wie *IEnumerable* im Falle von .NET.

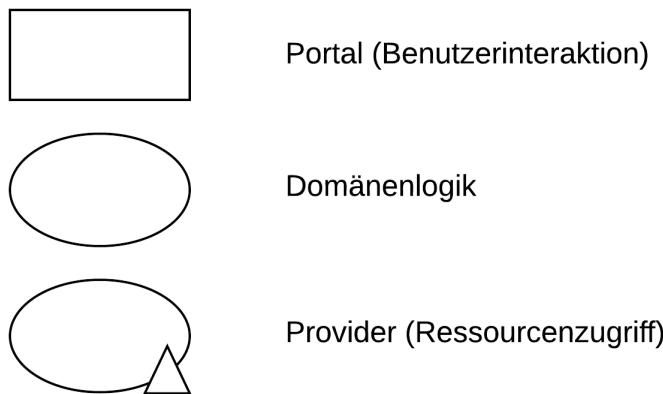


Abbildung 14: Rechteck, Kreis und Dreieck als Funktionseinheiten

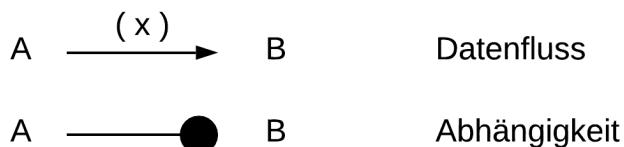


Abbildung 15: Datenfluss und Abhängigkeit

Der Entwurf der obersten Ebene entsteht aus den Dialogen und Interaktionen auf kanonische Weise: Jede Interaktion führt zu einem Entwurf. Die Bezeichnung der Interaktion wird zur Bezeichnung der Funktionseinheit in der Mitte der jeweiligen Entwürfe. Diese Funktionseinheiten werden als *Interaktoren* bezeichnet, da sie die gesamte Domänenlogik repräsentieren, die bei dieser Interaktion abläuft. Ein Interaktor realisiert die Domänenlogik einer Interaktion.

Definition: **Interaktor** - Ein Interaktor realisiert die Domänenlogik einer Interaktion.

Im Interaktionsdiagramm verlaufen die Pfeile ganz absichtlich auf eine bestimmte Art und Weise. Die Interaktion *Start* verläuft von außerhalb des Dialogs auf den Dialog. Dies stellt dar, dass Daten von außen in den Dialog hinein fließen. Genau genommen interagiert der Anwender bei *Start* noch nicht mit dem Softwaresystem, sondern mit dem Betriebssystem, welches daraufhin die Anwendung startet. Alle weiteren Interaktionen im obigen Beispiel beginnen im Dialog und enden im selben Dialog. Jeder Dialog einer Anwendung, in diesem Fall gibt es lediglich einen einzigen, stellt Interaktionssmöglichkeiten bereit. Durch Betätigen der im Menü angezeigten Tasten kann der Anwender mit der Anwendung interagieren. Somit löst die Interaktion in einem Dialog die Ausführung der Domänenlogik aus. Als Resultat kehren Daten, die von der Domänenlogik erzeugt wurden, zurück zum Dialog und werden von diesem angezeigt. Beim Übergang von der Anforderungsanalyse in den Entwurf werden hier also alle Interaktionen in ein Flow Design Diagramm, bestehend aus *Portalen*, *Domänenlogik* und *Datenflüssen* überführt. In der Analyse der Anforderungen sprechen wir von Dialogen und Interaktionen. Beim Entwurf einer Lösung werden Dialoge durch Portale realisiert.

Ziel des Entwurfs in die Breite ist es, sich einen Überblick über die Interaktionen des Dialogs zu verschaffen. Häufig geht es darum, die Datentypen zu definieren, die in den einzelnen Datenflüssen verwendet werden. Ferner muss sich häufig mit der Frage befasst werden, wo der Zustand der Anwendung gehalten werden soll. Im oben dargestellten Entwurf ist erkennbar, dass der Zustand der Anwendung in den Funktionseinheiten in der Mitte, den Interaktoren, gehalten werden soll. Beim Blättern fließt keine Information von der UI zu den Interaktoren. Folglich müssen die Interaktoren *Start*, *Next Page*, *Previous Page*, etc. Zustand darüber halten, welche Seite gerade angezeigt wird, um von dieser zur nächsten oder vorigen blättern zu können. Die UI ist hier zustandslos.

Interaktion: Start

Entwurf verfeinern - in die Tiefe

Nachdem wir durch den Entwurf in die Breite einen Überblick über den Zusammenhang aller Interaktionen erhalten haben, können wir nun beginnen, in die Tiefe zu gehen. Das Ziel dabei ist es, den Entwurf so weit zu verfeinern, dass eine Implementation möglich ist. Im Idealfall soll bei der Implementation keine Frage zur Lösung des Problems auftreten. Es geht dann "nur" noch um die Umsetzung der Lösung, stupides Codieren.

Beginnen wir mit der *Start* Interaktion. Die folgende Abbildung zeigt die Verfeinerung des Entwurfs.

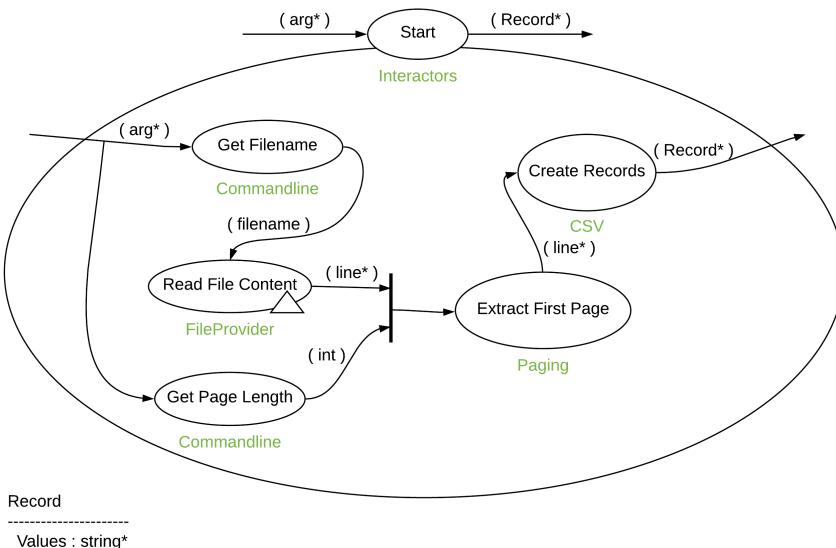


Abbildung 16: Verfeinerung des Entwurfs der *Start* Interaktion

Zunächst entnimmt die Funktionseinheit *GetFilename* den Dateinamen aus den Kommandozeilenparametern. Der Dateiname fließt zur Funktionseinheit *ReadFileContent*. Diese liest den Inhalt der Datei und liefert eine Aufzählung von Strings zurück. Die Notation *string** steht für "viele Strings", realisiert

etwa durch ein Array oder eine Liste. Hier wird der gesamte Inhalt der Datei in den Speicher gelesen. Diese Lösung ist folglich nur geeignet, für überschaubar große Dateien, so wie es die nicht-funktionalen Anforderungen vorgeben.

Im Anschluss wird durch die Funktionseinheit *ExtractFirstPage* die erste Seite der Datei extrahiert. Da hierfür nicht nur der Dateiinhalt sondern auch die Länge einer Seite benötigt wird, laufen die Datenflüsse aus *ReadFileContent* und *GetPageLength* in einem sogenannten *Join* zusammen. *GetPageLength* entnimmt der Kommandozeile die optional angegebene Seitenlänge. Falls keine Seitenlänge angegeben wurde, wird der Standardwert 10 geliefert.

Der *Join* drückt aus, dass die Daten beider Datenflüsse zur Verfügung stehen müssen, bevor es mit *ExtractFirstPage* weitergehen kann. Beide Informationen, Seitenlänge und Zeilen, fließen in die Funktionseinheit *ExtractFirstPage*. Sie liefert die Kopfzeile sowie die erste Seite der Datenzeilen zurück. Diese Aufzählung von Strings gelangt schließlich zur Funktionseinheit *CreateRecords*, die Zeile für Zeile in einen *Record* wandelt. Die Datenstruktur *Record* besteht aus der Eigenschaft *Values*, einer Aufzählung von Strings. In *CreateRecords* wird jede CSV formatierte Zeile am Semikolon getrennt, so dass die einzelnen Werte der Zeile extrahiert werden.

Da der Entwurf vorsieht, dass eine Aufzählung von *Record* Objekten an die *Ui* zur Anzeige geliefert wird, sollte die *Ui* nun ebenfalls verfeinert werden. Sie ist dafür zuständig, die Datensätze als formatierte Tabelle auszugeben. Dies ist ohne Entwurf einer Lösung zu viel für eine unmittelbare Implementation. Ohne eine sehr konkrete Vorstellung davon, wie die Formatierung erfolgt, würde Code entstehen, der wenig wandelbar ist. Ferner wird die Testbarkeit ohne den Entwurf einer Lösung nicht optimal sein.

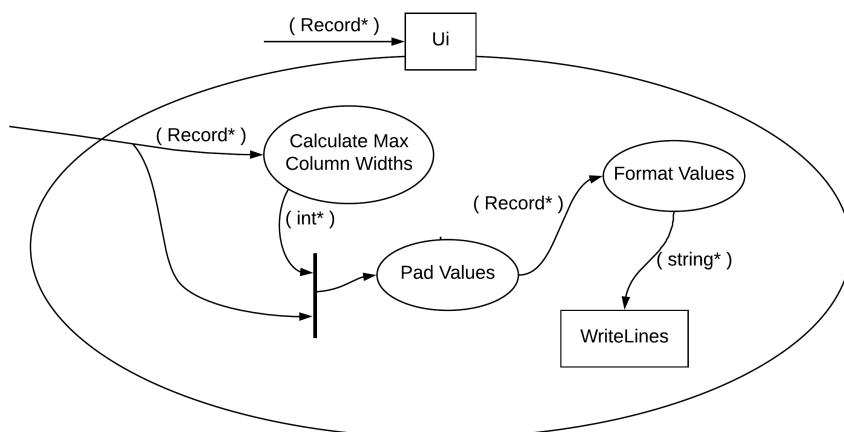


Abbildung 17: Verfeinerung der *Ui*

Im ersten Schritt werden durch *CalculateMaxColumnWidths* die maximalen Spaltenbreiten zu den *Records* ermittelt. Die Spaltenbreiten sowie die *Records* fließen anschließend zur Funktionseinheit *PadValues*. Diese liefert *Records* zurück, in denen die Werte der einzelnen Spalten so mit Leerzeichen aufgefüllt sind, dass sie jeweils der Spaltenbreite entsprechen. Danach werden die so vorbereiteten *Records* von *FormatRecords* in Strings umgewandelt. Dabei werden Trennzeichen zwischen die Spalten eingefügt. Letzter Schritt ist die Ausgabe der Strings auf der Konsole durch die Funktionseinheit *WriteLines*. Nun können die Funktionseinheiten *Start* und *Ui* implementiert werden.

Klassenzuordnung

Beim Übergang vom Entwurf zur Implementation muss zu jeder Funktionseinheit entschieden werden, ob sie in eine *Methode* oder eine *Klasse* übersetzt werden soll. *Start* und alle darin enthaltenen Funktionseinheiten lassen sich problemlos in Methoden übersetzen. Die *Ui* zeichnet sich dadurch aus, dass sie mehrere ein- und ausgehende Datenflüsse hat. Daher wird sie als Klasse implementiert.

Da Methoden in den meisten objektorientierten Sprachen in Klassen abgelegt werden müssen, muss zu jeder Methode entschieden werden, in welcher Klasse sie implementiert wird. Zum einen müssen also nun noch Klassennamen gefunden werden. Zum anderen muss überlegt werden, welche Methoden in der selben Klasse zusammengefasst werden und welche bewusst getrennt werden. Maßgebend sind hier die Aspekte, für die die einzelnen Methoden verantwortlich sind. Aspekte sind zu trennen, folglich müssen Methoden, welche für unterschiedliche Aspekte verantwortlich sind, in unterschiedlichen Klassen abgelegt werden.

Die Methoden *GetFilename* und *GetPageLength* befassen sich beide mit dem Thema Kommandozeilenparameter. Daher landen beide zusammen in der Klasse *CommandLine*. Die Methode *ReadFileContent* stellt einen Ressourcenzugriff dar und wird daher mit keiner der anderen Methoden zusammengelegt, da keine der anderen Methoden etwas mit Ressourcenzugriffen allgemein bzw. Dateizugriffen im speziellen zu tun haben. Der Zugriff auf Ressourcen wird generell getrennt von Domänenlogik und Benutzerschnittstelle. *ReadFileContent* wird in der Klasse *FileProvider* abgelegt. Dass die Klasse auf *Provider* endet, hat einen Grund: alle Ressourcenzugriffe erfolgen über *Provider*. Um Ressourcenzugriffe schon am Klassennamen als solche identifizieren zu können, wird hier die Konvention verwendet, *Provider* als Suffix anzuhängen. Eine detaillierte Betrachtung dazu folgt in einem späteren Kapitel.

Bei der Funktionseinheit *ExtractFirstPage* geht es um das Thema Blättern. Der Klassename lautet daher *Paging*. *CreateRecords* ist dafür zuständig, nach den Regeln von CSV, *Records* aus Strings zu erstellen. Die Methode wird daher in der Klasse *Csv* realisiert. Bleibt als letzte Funktionseinheit noch die Integrationsmethode *Start*. Sie repräsentiert eine Interaktion des Anwenders. Daher lautet ihr Klassename *Interactors*. Der Plural wird verwendet, weil hier später auch die anderen Interaktoren wie *FirstPage*, *NextPage* etc. aufgenommen werden. Die Methoden- und Klassennamen sind alle in englisch gehalten. Das muss nicht zwingend so sein. Die Domänsprache kann auch auf deutsch herausgebildet werden. Einheitlich sollte es allerdings sein.

Implementation

Das Kodieren des Entwurfs kann nun beginnen. Man kann dabei top-down oder bottom-up vorgehen. Top-down bedeutet, von oben zu starten, also bspw. bei *Start*. Die Methode *Start* repräsentiert den Flow aus der Abbildung weiter oben. Schreibt man die Methode in der Entwicklungsumgebung seiner Wahl einfach von oben nach unten hin, fehlen die Methoden, die *Start* verwendet. Die meisten Entwicklungsumgebungen bieten eine Funktion, mit der nicht vorhandene Klassen und Methoden mehr oder weniger automatisch angelegt werden können. Insofern bietet die top-down Vorgehensweise den Vorteil, dass die Tipparbeit auf diese Weise reduziert wird. Startet man dagegen bottom-up, werden zunächst die *Operationen* implementiert. So werden die Methoden bezeichnet, die von *Start* integriert werden. Die Codegenerierung der Entwicklungsumgebung ist dadurch nur sehr eingeschränkt nutzbar. Erst wenn zuletzt *Start* implementiert wird, geht das Schreiben schneller, weil nun alle benötigten Methoden gefunden werden.

Das folgende Listing zeigt die *Start* Methode:

```
public IEnumerable<Record> Start(string[] args) {
    var filename = commandLine.GetFilename(args);
    var pageLength = commandLine.GetPageLength(args);
    var lines = fileProvider.ReadFileContent(filename);
    var firstPage = paging.ExtractFirstPage(lines, pageLength);
    var records = Csv.CreateRecords(firstPage);
    return records;
}
```

Die meisten der Methoden sind als statische Methoden angelegt. Lediglich das Thema Blättern in Form der Methode *ExtractFirstPage* ist als Instanzmethode realisiert. Das liegt daran, dass beim Blättern Zustand gehalten werden muss. Sobald im nächsten Schritt *ExtractNextPage* realisiert wird, muss die Klasse *Paging* sich merken, welche Seite gerade angezeigt wird. Daher wird eine Instanz der Klasse *Paging* erzeugt und während der gesamten Laufzeit des Programmes verwendet.

Die Funktionseinheit *Ui* sieht auf oberster Ebene wie folgt aus:

```
public class Ui
{
    public void Display(IEnumerable<Record> records) {
        var columnWidths = CalculateMaxColumnWidths(records);
        var paddedRecords = PadRecords(records, columnWidths);
        var lines = FormatRecords(paddedRecords);
        WriteLines(lines);
    }

    // ...
}
```

Die Methode *Display* stellt die Integrationsmethode dar. Sie ruft die einzelnen Operationen des Flows auf. Letzter Schritt ist die Integration von *Interactors.Start* und *Ui.Display* in der Klasse *Program*:

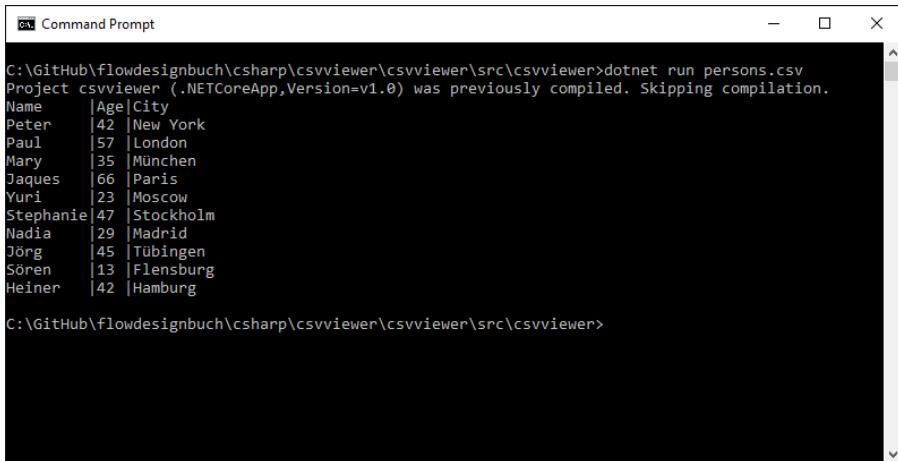
```
public class Program
{
    public static void Main(string[] args) {
        var interactors = new Interactors();
        var ui = new Ui();

        void Start() {
            var records = interactors.Start(args);
            ui.Display(records);
        }

        Start();
    }
}
```

Die Klasse *Program* enthält den Einstiegspunkt in das Programm. Ihre Verantwortung besteht darin, die benötigten Klassen zu instanziieren und zu integrieren. Hier werden die Datenflüsse der obersten Ebene hergestellt. Nach dem jeweils eine Instanz der Klassen *Interactors* und *Ui* erzeugt sind, wird eine lokale Methode *Start* definiert und aufgerufen. In dieser Methode wird die Verdrahtung der Funktionseinheiten *Interactors.Start* und *Ui.Display* vorgenommen.

In dieser ersten Iteration ist der CSV Viewer noch nicht fertig geworden. Es ist lediglich die Interaktion *Start* realisiert.



The screenshot shows a Windows Command Prompt window titled "Command Prompt". The command entered was "dotnet run persons.csv". The output displays a table of names, ages, and cities. The table has three columns: Name, Age, and City. The data is as follows:

Name	Age	City
Peter	42	New York
Paul	57	London
Mary	35	München
Jaques	66	Paris
Yuri	23	Moscow
Stephanie	47	Stockholm
Nadia	29	Madrid
Jörg	45	Tübingen
Sören	13	Flensburg
Heiner	42	Hamburg

C:\GitHub\flowdesignbuch\csharp\csvviewer\csvviewer\src\csvviewer>

Abbildung 18: Die Anwendung nach der Realisierung von *Start*.

Interaktion: NextPage

Entwurf verfeinern

Wie schon bei *Start* muss der Entwurf von *NextPage* vor der Implementation verfeinert werden. Die Verfeinerung ist ausreichend detailliert, wenn alle Aspekte getrennt sind und die einzelnen Funktionseinheiten in kurzer Zeit implementiert werden können. Wir empfehlen, die Funktionseinheiten so klein zu halten, dass sie in maximal 4 Stunden von einem einzelnen Entwickler implementiert werden können. Der Grund dafür ist, dass jedes Inkrement nach 1-2 Tagen fertiggestellt werden soll. Wird die Implementation einer einzelnen Funktionseinheit bereits einen halben Tag in Anspruch nehmen, ist die Wahrscheinlichkeit groß, dass in ihr weitere Herausforderungen lauern und das Inkrement nach 2 Tage nicht abgeschlossen werden kann. Für solche Funktionseinheiten lohnt es sich daher, den Entwurf weiter zu verfeinern, um die verbliebene Unsicherheit auszuräumen.

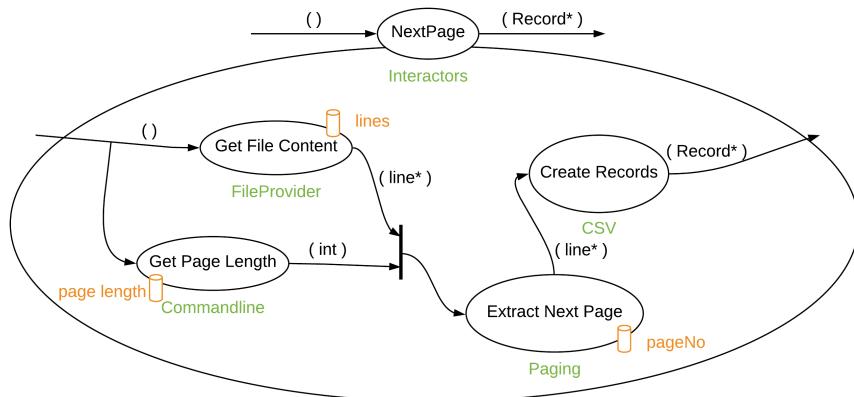


Abbildung 19: Verfeinerung von *NextPage*.

Bei der Interaktion *NextPage* stehen wir vor der Herausforderung, dass diese Interaktion sich gemeinsamen Zustand mit der Interaktion *Start* teilt. Dem Entwurf ist zu entnehmen, dass in *NextPage* keine Daten hinein fließen. Folglich muss *Start* den benötigten Zustand aufbauen, damit *NextPage* diesen dann nutzen kann. In Abbildung 1 sind drei Funktionseinheiten mit Zustand ausgestattet:

- *GetFileContent* greift auf die eingelesenen Zeilen der Datei zu (*lines*).
- *ExtractNextPage* muss wissen, welche Seite gerade angezeigt wird, um zur nächsten zu blättern (*pageNo*).
- *GetPageLength* muss die Seitenlänge liefern, die in *Start* aus dem optionalen Kommandozeilenparameter ermittelt wurde (*page length*).

Für alle drei Zustände gilt, dass *Start* diese setzen muss. Der Entwurf für *Start* ist daher, wie in der folgenden Abbildung zu sehen, zu ergänzen.

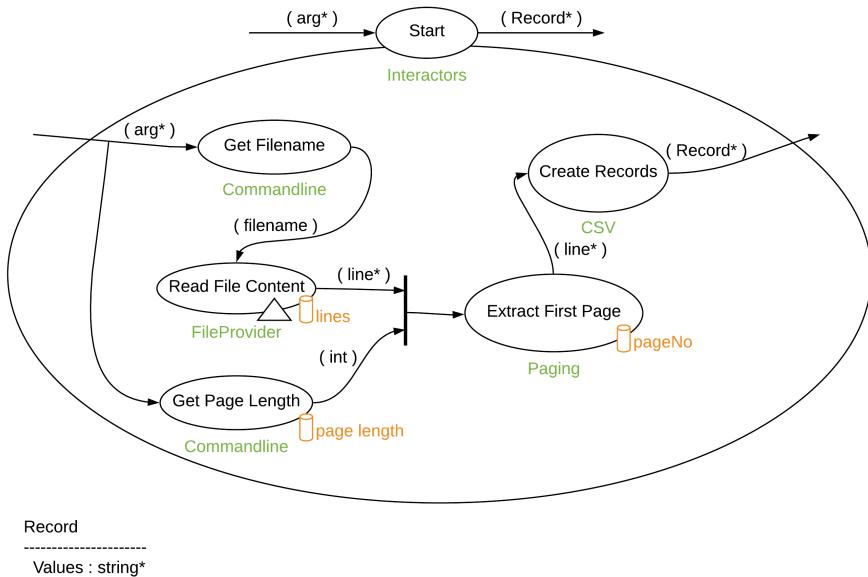


Abbildung 20: Verfeinerung von *Start* mit Ergänzung des Zustands.

Implementation

Die Implementation geht nun leicht von der Hand, da bereits einige Funktionseinheiten vorhanden sind. Zunächst sind Anpassungen an *Start* erforderlich, die der Entwurf in der Abbildung zeigt. In der Klasse *CommandLine* muss eine parameterlose Variante der Methode *GetPageLength* ergänzt werden. Diese liefert die zuvor aus den Kommandozeilenargumenten ermittelte Seitenlänge. Weil die Klasse nun Zustand hält, ist es erforderlich, mit Instanzen zu arbeiten statt mit statischen Methoden. Folgendes Listing zeigt die geänderte Klasse *CommandLine*:

```

public class CommandLine
{
    private int _pageLength;

    public string GetFilename(string[] args) {
        return args[0];
    }

    public int GetPageLength(string[] args) {
        _pageLength = args.Length > 1 ? int.Parse(args[1]) : 10;
        return _pageLength;
    }

    public int GetPageLength() {
        return _pageLength;
    }
}

```

Der Zustand, der im Entwurf als Tonne an der Funktionseinheit notiert ist, wird in der Implementation als Feld der Klasse umgesetzt. Ähnliche Anpassungen sind in den Klassen *FileProvider* und *Paging* erforderlich. Die Klasse *FileProvider* merkt sich beim Einlesen der Datei mit *ReadFileContent* den Inhalt, der dann von *GetFileContent* geliefert werden kann. Wieder wird der Zustand über ein Feld der Klasse realisiert und wieder müssen daher Instanzen statt statischer Methoden verwendet werden.

Bei der Klasse *Paging* wurde die Notwendigkeit, Zustand zu halten, bereits im ersten Entwurf vorhergesehen. Daher ist hier keine Änderung von statisch zu Instanz erforderlich. Allerdings treffen solche “Vorhersehungen” in der Praxis nicht immer ein, so dass im Zweifel besser für die gerade im Moment vorliegenden Anforderungen implementiert werden sollte.

Die Integrationsmethode *Interactors.Start* musste ebenfalls angepasst werden, um nun mit Instanzen zu arbeiten. Ferner ist hier der Interactor für *NextPage* hinzugekommen.

```
public class Interactors
{
    private readonly Paging paging = new Paging();
    private readonly CommandLine commandLine = new CommandLine();
    private readonly FileProvider fileProvider = new FileProvider();

    public IEnumerable<Record> Start(string[] args) {
        var filename = commandLine.GetFilename(args);
        var pageLength = commandLine.GetPageLength(args);
        var lines = fileProvider.ReadFileContent(filename);
        var firstPage = paging.ExtractFirstPage(lines, pageLength);
        var records = Csv.CreateRecords(firstPage);
        return records;
    }

    public IEnumerable<Record> NextPage() {
        var pageLength = commandLine.GetPageLength();
        var lines = fileProvider.ReadFileContent();
        var nextPage = paging.ExtractNextPage(lines, pageLength);
        var records = Csv.CreateRecords(nextPage);
        return records;
    }
}
```

Das Starten der Anwendung in der Methode *Program.Main* muss ergänzt werden, um den Interactor *NextPage* aufzunehmen.

```
public class Program
{
    public static void Main(string[] args) {
        var interactors = new Interactors();
        var ui = new Ui();

        void Start() {
            var records = interactors.Start(args);
            ui.Display(records);
        }

        ui.MoveNext += () => {
            var records = interactors.NextPage();
            ui.Display(records);
        };

        Start();
        ui.Run();
    }
}
```

Ferner muss die *Ui* so angepasst werden, dass jeweils das Menü angezeigt und ein Tastendruck entgegengenommen wird. Diese Schleife wird bei Desktop Anwendungen mit grafischer *Ui* implizit durch das Window System bereitgestellt.

```
public class Ui
{
    public event Action MoveNext;

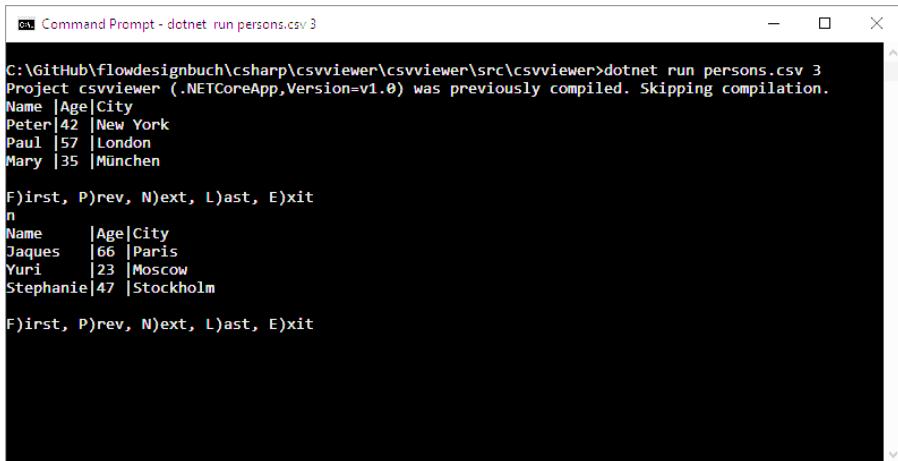
    public void Run() {
        var exit = false;
        do {
            Console.WriteLine();
            Console.WriteLine("F)irst, P)rev, N)ext, L)ast, E)xit");

            var key = Console.ReadKey().KeyChar.ToString().ToUpper();
            Console.WriteLine();

            switch (key) {
                case "E":
                    exit = true;
                    break;
                case "N":
                    MoveNext();
                    break;
            }
        } while (!exit);
    }

    // ...
}
```

Nicht gezeigt sind die Details, wie etwa die Methoden zum Blättern, mit denen die benötigten Zeilen extrahiert werden. Der vollständige Quellcode des Beispiels ist bei *github* unter folgendem Link zu finden:
<https://github.com/slieser/flowdesignbuch/tree/master/csharp/csvviewer/csvviewer>



The screenshot shows a Windows Command Prompt window titled "Command Prompt - dotnet run persons.csv 3". The window displays the following text:

```
C:\GitHub\flowdesignbuch\csharp\csvviewer\csvviewer\src\csvviewer>dotnet run persons.csv 3
Project csvviewer (.NETCoreApp,Version=v1.0) was previously compiled. Skipping compilation.
Name |Age|City
Peter|42 |New York
Paul |57 |London
Mary |35 |München

F)irst, P)rev, N)ext, L)ast, E)xit
n
Name      |Age|City
Jaques    |66 |Paris
Yuri     |23 |Moscow
Stephanie|47 |Stockholm

F)irst, P)rev, N)ext, L)ast, E)xit
```

Abbildung 21: Konsolenausgabe des aktuellen Stands.

Interaktion: PrevPage

Entwurf verfeinern

Die Interaktion *PrevPage* muss vor der Implementation ebenfalls verfeinert werden. Da uns bereits ein Entwurf für *NextPage* vorliegt, fällt das nicht besonders schwer. Die folgende Abbildung zeigt das Ergebnis.

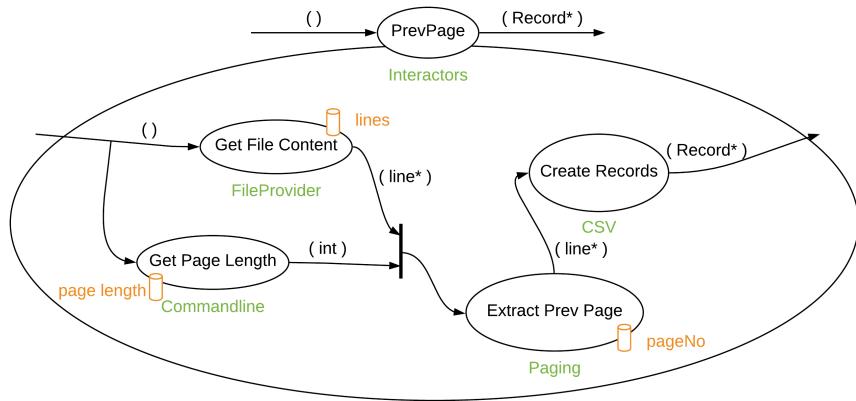


Abbildung 22: Verfeinerung von *PrevPage*.

Der einzige Unterschied zu *NextPage* besteht beim Extrahieren der relevanten Zeile. Anstelle von *ExtractNextPage* wird nun *ExtractPrevPage* benötigt.

Implementation

Auch die Implementation geht nun flott von der Hand, da durch die vorhergehenden Inkremeante bereits fast alles vorhanden ist. Die Methode *Paging.ExtractPrevPage* existiert noch nicht und muss ergänzt werden. Ferner muss in der Klasse *Interactors* die Methode *PrevPage* ergänzt werden. Sie repräsentiert den oben gezeigten Entwurf.

```
public IEnumerable<Record> PrevPage() {
    var pageLength = commandLine.GetPageLength();
    var lines = fileProvider.GetFileContent();
    var nextPage = paging.ExtractPrevPage(lines, pageLength);
    var records = Csv.CreateRecords(nextPage);
    return records;
}
```

In der UI muss ein Event *PrevPage* ergänzt und an der richtigen Stelle ausgelöst werden.
Schließlich muss der neue Event in der *Program.Main* mit der zusätzlichen Methode *Interactors.PrevPage* verbunden werden.

```

public class Program
{
    public static void Main(string[] args) {
        var interactors = new Interactors();
        var ui = new Ui();

        void Start() {
            var records = interactors.Start(args);
            ui.Display(records);
        }

        ui.MoveNext += () => {
            var records = interactors.NextPage();
            ui.Display(records);
        };
        ui.MovePrev += () => {
            var records = interactors.PrevPage();
            ui.Display(records);
        };

        Start();
        ui.Run();
    }
}

```

Vielleicht haben Sie bemerkt, dass nun die beiden Integrationsmethoden *Interactors.NextPage* und *Interactors.PrevPage* eine Dopplung enthalten. Hier ist das DRY Prinzip verletzt: *Don't Repeat Yourself*, wiederhole dich nicht. Da es sich um eine 1:1 Übersetzung der Entwürfe handelt und der Code trivialer Integrationscode ist, akzeptieren wir diese Dopplung. Zwar wäre es möglich, die beiden Methoden auf eine gemeinsame Basis zu stellen und lediglich eine unterschiedliche Paging Methode aufzurufen. Doch wäre dann eine Änderung an einem einzelnen Flow aufwendiger, weil diese Änderung sich durch das Herausziehen der gemeinsamen Codezeilen nicht mehr so ohne weiteres bewerkstelligen lässt. Das Herausziehen der Gemeinsamkeiten müsste dann zunächst zurückgenommen werden. Wenn die Entwürfe der einzelnen Interaktionen im Laufe der Zeit stabiler werden, spricht nichts dagegen, Dopplungen dann herauszuziehen. Wir empfehlen allerdings, dies nicht zu früh vorzunehmen. Wohlgernekt gilt dies nur für trivialen Integrationscode.

Die Methode *Paging.ExtractPrevPage* habe ich selbstverständlich nicht als Kopie der schon bestehenden Methode *Paging.ExtractNextPage* erzeugt. Stattdessen habe ich die Gemeinsamkeiten herausgezogen, so dass beide Methoden zwar ähnlich aussehen, aber keine DRY Verletzung darstellen. Nach der Ergänzung von *ExtractPrevPage* sieht die Implementation wie folgt aus:

```

public class Paging
{
    private int _pageNo;

    public IEnumerable<string> ExtractFirstPage(
        IEnumerable<string> lines, int pageLength) {
        _pageNo = 1;
        return lines.Take(pageLength + 1);
    }

    public IEnumerable<string> ExtractNextPage(
        IEnumerable<string> lines, int pageLength) {
        IncrementPageNo(pageLength, lines.Count());
        var linesToSkip = CalculateLinesToSkip(pageLength);
        return ExtractLinesForPage(lines, pageLength, linesToSkip);
    }

    public IEnumerable<string> ExtractPrevPage(
        IEnumerable<string> lines, int pageLength) {
        DecrementPageNo(pageLength);
        var linesToSkip = CalculateLinesToSkip(pageLength);
        return ExtractLinesForPage(lines, pageLength, linesToSkip);
    }

    private void IncrementPageNo(int pageLength, int numberofLines) {
        if (_pageNo * pageLength + 1 < numberofLines) {
            _pageNo++;
        }
    }

    private void DecrementPageNo(int pageLength) {
        if ((_pageNo - 1) * pageLength + 1 > 1) {
            _pageNo--;
        }
    }

    private int CalculateLinesToSkip(int pageLength) {
        return (_pageNo - 1) * pageLength + 1;
    }

    private static IEnumerable<string> ExtractLinesForPage(
        IEnumerable<string> lines, int pageLength, int linesToSkip) {
        return lines
            .Take(1)
            .Union(lines.Skip(linesToSkip).Take(pageLength));
    }
}

```

Die beiden Methoden *ExtractNextPage* und *ExtractPrevPage* sind nun keine Operationen mehr sondern Integration. Sie integrieren die Operationen *IncrementPageNo*, *DecrementPageNo*, *CalculateLinesToSkip* und *ExtractLi-*

nesForPage. Auch hier gilt wieder, dass die DRY Verletzung in den beiden Integrationsmethoden hingenommen werden kann, da es trivialer Code ist.

Interaktionen: FirstPage und LastPage

Entwurf verfeinern

Die beiden noch verbleibenden Interaktionen *FirstPage* und *LastPage* stellen nun keine Herausforderung mehr dar. Auch hier unterscheiden sich die Entwürfe lediglich darin, die gewünschte Seite zu extrahieren. Mal ist es die erste Seite durch *Paging.ExtractFirstPage*, mal die letzte durch *Paging.ExtractLastPage*.

Implementation

Die Implementation muss nun auf die gleiche Weise ergänzt werden, wie zuvor schon bei der Interaktion *PrevPage*. Im GitHub Repository zum Buch ist der Quellcode für die vollständige CSV Viewer Anwendung abgelegt.

Fazit

Die dargestellte Vorgehensweise bei der Realisierung des CSV Viewers unterscheidet sich in zwei Punkten wesentlich von der Vorgehensweise, wie wir sie üblicherweise in Softwareentwicklungsteams vorfinden:

- *Anforderungen zerlegen*: Die Anforderungen wurden in der Analysephase konsequent zerlegt. Dabei kommt eine einfache aber wirkungsvolle Methode zum Einsatz: Dialoge und Interaktionen.
- *Entwerfen der Lösung*: Aufbauend auf der Analyse wurde die Lösung vor der Implementation entworfen. Für jede Interaktion aus der Analysephase wurde ein Entwurf erstellt. Ferner wurden die Entwürfe so weit verfeinert, dass jede Funktionseinheit nur noch für einen Aspekt zuständig ist.

Diese beiden Schritte, Analyse und Entwurf, machen einen großen Unterschied!

Teil I

Das Softwareuniversum

Der Entwurf von Software ist ein kreativer Prozess, der viele Facetten hat. Vielen Entwicklern ist gar nicht bewusst, dass sie Software nicht nur implementieren sondern auch entwerfen. Das Implementieren nimmt viel Zeit in Anspruch: Entwickler sitzen am Rechner und codieren, schreiben also Quellcode in einer Programmiersprache wie C#, Java oder JavaScript. Dass sie dabei gleichzeitig eine Lösung für die anstehenden Anforderungen entwerfen, ist ihnen häufig nicht bewusst. Jedenfalls finden die Prozesse Entwurf und Umsetzung nicht klar getrennt voneinander statt. Der Entwickler arbeitet in seiner Entwicklungsumgebung (IDE), codiert, und denkt gleichzeitig darüber nach, wie er das anstehende Problem lösen könnte. Nur selten tritt er zurück, diskutiert mit seinen Kollegen und trennt dadurch die beiden Phasen Entwurf einer Lösung und Umsetzung der Lösung. Und selbst wenn ein Team einmal über den Entwurf einer Lösung diskutiert, findet dies nicht methodisch statt. Die wenigsten Teams verwenden regelmäßig eine grafische Notation für ihre Entwürfe. Sie haben keine Übung darin, gemeinsam über die Lösung nachzudenken.

Ein wesentlicher Schritt in Bezug auf die Verbesserung des gesamten Softwareentwicklungsprozesses besteht darin, die Phasen Entwurf und Umsetzung klar zu trennen. Dies bietet mindestens die beiden folgenden Vorteile: der Entwurf der Lösung kann durch die Loslösung von der Umsetzung in einer anderen „Sprache“ durchgeführt werden, nämlich in einer grafischen statt textuellen. Eine grafische Repräsentation eignet sich besser zur Visualisierung von Entwürfen. Ferner können auf diese Weise mehrere Entwickler gemeinsam am Entwurf der Lösung arbeiten, was auf rein textueller Basis eher nicht so gut möglich ist.

Um einen Überblick darüber zu gewinnen, welche Anteile von Softwareentwurf in diesem Buch behandelt werden, sind in der folgenden Abbildung die wesentlichen Aspekte dargestellt, die den Entwurf von Software betreffen.

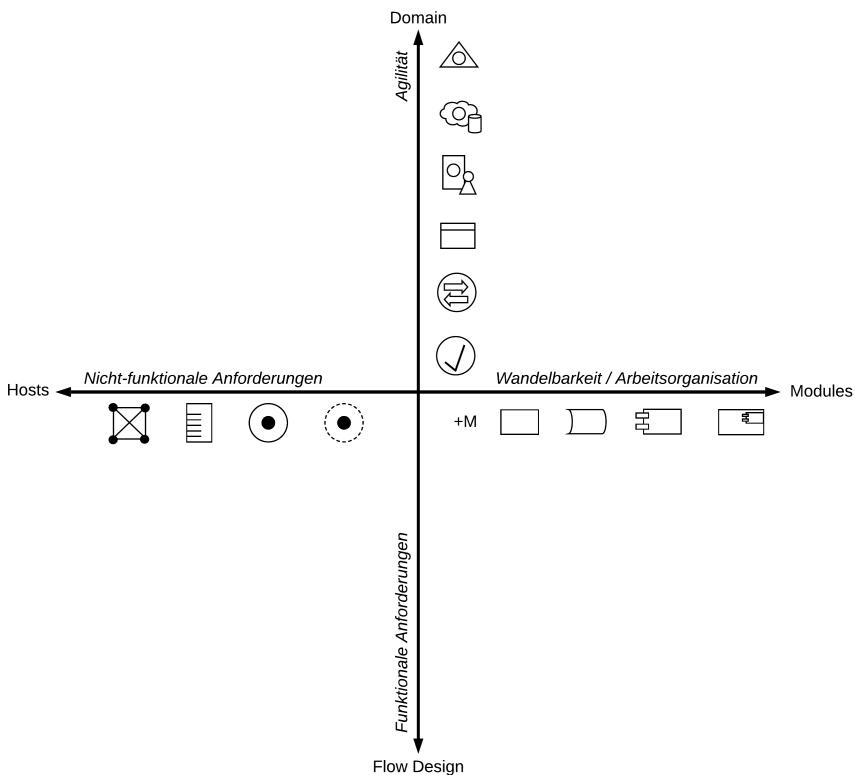


Abbildung 23: Das Softwareuniversum. Die eingefärbte Fläche zeigt, welche Bereiche in diesem Buch behandelt werden.

Das Softwareuniversum besteht aus vier Achsen:

- **Domain:** System, Bounded Context, Application, Dialog, Interaktion, Feature
Betrachtet die Anforderungen und zerlegt diese so weit, dass ein kleiner Ausschnitt entsteht, der als *Inkrement* realisiert werden kann. Ziel dieser Dimension ist die *Agilität*. Damit wird eine Vorgehensweise bezeichnet, die zu regelmäßiger und kurzfristigem Feedback führt.
- **Module:** Methode, Klasse, Bibliothek, Komponente, Microservice
Betrachtet die Orte, an denen Funktionalität abgelegt werden kann. Diese Dimension hat zum einen die *Wandelbarkeit* zum Ziel. Die Funktionalität muss so auf Module verteilt werden, dass Entwickler in die Lage versetzt werden, das Softwaresystem über viele Jahrzehnte anzupassen und zu ergänzen. Neben der Wandelbarkeit geht es in dieser Dimension um die *Arbeitsorganisation*.

Teams sollen durch eine angemessene Aufteilung der Software auf Module in die Lage versetzt werden, arbeitsteilig gemeinsam einen Ausschnitt der Anforderungen zu realisieren. Dazu dient die Komponentenorientierung.

- **Hosts: Thread, Process, Machine, Site**
Betrachtet die Laufzeit der Funktionalität. Das Ziel sind die *nicht-funktionalen Anforderungen*. Aufgrund von nicht-funktionalen Anforderungen kann es bspw. notwendig sein, ein Softwaresystem zu verteilen, um so den Zugriff von unterschiedlichen Standorten zu ermöglichen.
- **Flow Design**
Diese Dimension betrachtet die *funktionalen Anforderungen*. Das Entwerfen einer konkreten Lösung mit Hilfe von Datenflüssen ist das zentrale Thema des Buches.

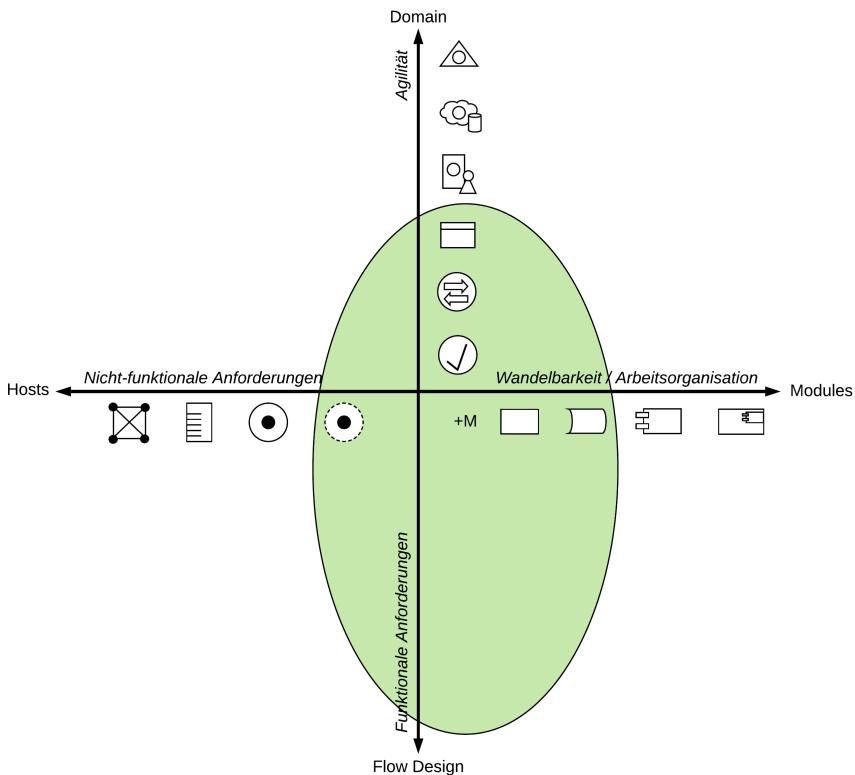


Abbildung 24: Das Softwareuniversum. Die eingefärbte Fläche zeigt, welche Bereiche in diesem Buch behandelt werden.

In der Abbildung sind die Bereiche der Dimensionen eingefärbt, welche maßgeblich im Buch behandelt werden. Manche Teilspekte werden

bewusst nur angerissen, wo es dazu dient, das Verständnis für den gesamten Softwareentwicklungsprozess zu schärfen. Im Kern geht es darum, mit Flow Design Diagrammen einen funktionalen Entwurf der Lösung zu erstellen.

Die folgenden Kapitel stellen dar, wie der Entwurf einer Lösung sowie die Umsetzung konkret durchgeführt wird.

Anforderungen zerlegen

Wasserfall vs. Agilität

Über viele Jahrzehnte wurde versucht, Softwareprojekte in Phasen zu unterteilen, die nacheinander durchlaufen werden. In Anlehnung an den Bau eines Hauses wurde versucht, zunächst alle Anforderungen zu analysieren, dann einen Entwurf zu erstellen für die gesamten Anforderungen, dann den gesamten Entwurf zu implementieren und so fort. Dieses als *Wasserfall* bezeichnete Vorgehen hat viele Nachteile und gilt als gescheitert. Zum einen ist das Vorhaben „Softwareentwicklung“ zu komplex, um es auf diese Weise in Phasen zu zerlegen. Es zeigen sich während der Durchführung meist Unzulänglichkeiten aus vorhergehenden Phasen, die dazu führen, dass an dem Plan nicht festgehalten werden kann. Zum anderen ist das Risiko, am Ende des Budgets mit leeren Händen da zu stehen sehr groß, da erst sehr spät einsatzfähige Software produziert wird. So lange der Fokus nicht konsequent auf der Auslieferung kleiner Ausschnitte des Gesamten liegt, wird die Auslieferung auf das Ende des gesamten Prozesses verschoben. Auf diese Weise entsteht ein extremes Risiko, am Ende mit leeren Händen dazustehen. Die Anlehnung an Architektur oder Maschinenbau hat in der Softwareentwicklung über viele Jahre hinweg ein ungeeignetes Bild geschaffen. Ein Gebäude kann nicht zu 5% ausgeliefert werden. Dort gilt eher ein „alles oder nichts“. Auch Maschinen können eher nur vollständig ausgeliefert werden. Software ist andererseits so flexibel weil virtuell, dass es technische überhaupt kein Problem ist, zunächst lediglich einen Bruchteil der gesamten Anforderungen umzusetzen und auszuliefern. Sukzessive kann Funktionalität ergänzt werden. Einem Gebäude kann nicht nachträglich ein Keller hinzugefügt werden. Bei Software sind solche Ergänzungen möglich. Dabei sollte man die Bilder und Parallelen zur Architektur schnell verlassen und den „Keller“ nicht als das Fundament einer Software betrachten. Natürlich gibt es auch im Bereich Software fundamentale Entscheidungen, die nicht ohne weiteres verändert werden können. Eine Desktopanwendung kann nicht mal eben so zur Webanwendung umgebaut werden. Doch sollten wir uns hüten, aus den falschen Bildern abzuleiten, dass Software daher nur „in einem Guss“ erstellt und ausgeliefert werden kann. Das Gegenteil ist der Fall und muss angestrebt werden.

Die Agilitätsbewegung hat zu einem Vorgehen in *Inkrementen* geführt. Es wird jeweils nur ein kleiner Ausschnitt der Anforderungen betrachtet. Dieser wird durch alle Phasen geführt, so dass am Ende einer jeden Iteration

potentiell lieferbare Software steht. Geht in diesem Modell das Budget zur Neige, sind dem Kunden bereits einige lauffähige Inkremeante geliefert worden. Ferner ist der in der jeweiligen Phase betrachtete Ausschnitt des Gesamtprojekts jeweils überschaubar klein. Ziel der Agilität ist regelmäßiges und kurzfristiges Feedback. Nicht am Ende des Projektes, sondern regelmäßig jeweils im Abstand weniger Tage muss Feedback eingeholt werden, um so sicherzustellen, dass wirklich die Anforderungen des *Product Owners* umgesetzt werden.

Doch nun stellt sich die zentrale Frage: wie können die Anforderungen auf gute Weise zerlegt werden? Diese Zerlegung muss so erfolgen, dass für einen kleinen Ausschnitt der Anforderungen ein Entwurf einer Lösung erstellt werden kann, der im Anschluss umgesetzt wird. Ferner müssen die Anforderungen so zugeschnitten sein, dass ein vertikaler Durchstich, oder auch *Inkrement*, bearbeitet wird. Ein solches Inkrement ist ein installierbares Stück Software, zu dem der Product Owner sein Feedback geben kann. Es zeichnet sich dadurch aus, dass von allen betroffenen Aspekten ein kleiner Teil realisiert ist, so dass eine lauffähige Anwendung vorliegt. Das Inkrement verfügt noch nicht über die gesamte gewünschte Funktionalität, sondern realisiert nur einen Ausschnitt der Anforderungen. Im Gegensatz zu einem horizontal geschnittenen Ausschnitt hat das Inkrement einen echten Nutzen für den Product Owner bzw. den Kunden. Die folgende Abbildung zeigt den Unterschied zwischen horizontalem und vertikalem Schneiden.

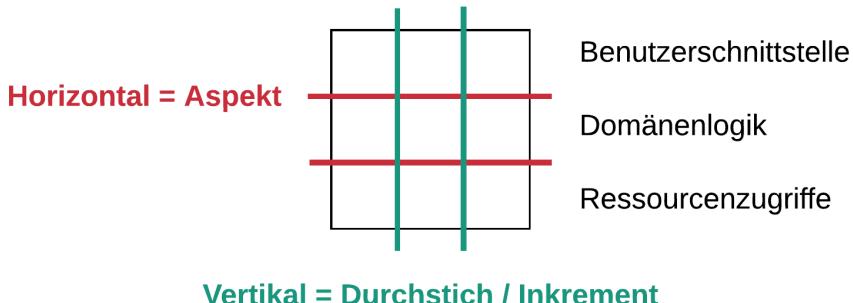


Abbildung 25: Horizontaler vs. vertikaler Schnitt

In der Abbildung stellt das Rechteck die gesamte Lösung dar. Horizontal dargestellt sind die wesentlichen Aspekte Benutzerschnittstelle, Domänenlogik und Ressourcenzugriffe. Dies stellt eine starke Vereinfachung dar, denn eine Lösung besteht bei näherer Betrachtung aus vielen weiteren Aspekten. Für die Unterscheidung von horizontalem bzw. vertikalem Schnitt kann das vernachlässigt werden.

Eine mögliche Vorgehensweise wäre, Aspekt für Aspekt zu realisieren. Das Erstellen der gesamten Lösung würde dann in drei Iterationen eingeteilt. Während der ersten Iteration wird der gesamte benötigte Ressourcenzugriff realisiert. In der nächsten Iteration kommt die gesamte Benutzerschnittstelle an die Reihe und zum Abschluss in der dritten Iteration die Domänenlogik. Die Reihenfolge ist willkürlich gewählt. Bei horizontaler Vorgehensweise ergibt sich die Herausforderung, dass der Product Owner erst am Ende der letzten Iteration lauffähige Software erhält. Somit ist der Product Owner erst in der Lage Feedback zu geben, wenn die gesamte Lösung umgesetzt ist. Schneiden wir stattdessen vertikal, wird in der ersten Iteration ein kleiner Ausschnitt der Benutzerschnittstelle realisiert, sowie jeweils kleine Teile der Domänenlogik und des Ressourcenzugriffs. Das Ergebnis jeder Iteration ist daher ein Inkrement. Der Product Owner kann bereits zum ersten Inkrement Feedback geben.

Definition: **Agilität** bedeutet, regelmäßiges und kurzfristiges Feedback.

Das regelmäßige und vor allem kurzfristige Feedback des Product Owners ist wesentlich für einen flüssigen Entwicklungsprozess. Nur wenn das Team kontinuierlich Feedback erhält, kann es die Spur halten. Das ist vergleichbar mit einer Autofahrt. Der Fahrer muss kontinuierlich die Umgebung beobachten um daraus abzuleiten, ob er bremsen oder beschleunigen muss, ob er etwas nach links oder rechts steuern muss. Ohne kontinuierliches Feedback kann ein Auto nicht von einem Ort zum anderen bewegt werden. Man kann nicht den Weg planen und das Auto so programmieren, dass es exakt nach Plan fährt. Auch selbstfahrende Autos benötigen kontinuierlich Feedback, um daraus die Steuerung abzuleiten.

Beim Produzieren von Software ist ebenfalls kontinuierliches Feedback erforderlich. Wenn das Team horizontal schneidet, entstehen Ergebnisse, zu denen niemand fundiertes Feedback geben kann. Der Ressourcenzugriff mag tipp topp implementiert sein. Hunderte von automatisierten Tests sind grün. Dennoch zeigt sich erst beim konkreten Verwenden des produzierten Teils, ob er passgenau die Anforderungen umsetzt. Es könnte sein, dass sich beim Verwenden herausstellt, dass einzelne realisierte Teile garnicht benötigt werden. Genauso gut könnte es sein, dass die realisierte Funktionalität im Detail ein klein wenig anders benötigt wird. Nur durch vertikales Schneiden kann dies vermieden werden. Wird jeweils ein kleiner Ausschnitt der Aspekte realisiert, können diese sofort zu einem Inkrement integriert werden. Erst durch die Integration zeigt sich, ob alles zusammen passt. Nur zu einem installierbaren, ausführbaren Stück Software kann der Product Owner wirklich Feedback geben. Nun stellt sich weiterhin die Frage, wie die Anforderungen so formuliert werden können, dass das Team sie für den vertikalen Schnitt verwenden kann. Bevor die vielfach verwendeten *User*

Stories kurz behandelt werden, schauen wir im folgenden Abschnitt auf das *System-Umwelt-Diagramm*.

System-Umwelt-Diagramm

Jedes Softwaresystem ist eingebettet in seine Umwelt. Auf der einen Seite haben wir Rollen, die vom System abhängen. Die Rollen sind letztlich der Grund dafür, dass das System erstellt wird. Würde es keine Rolle geben, die vom zukünftigen System abhängt, gäbe es keinen Grund, das System zu erstellen. Auf der anderen Seite ist das System innerhalb seiner Umwelt abhängig von Ressourcen. Das können aus einer Systembetrachtung heraus wiederum Systeme sein. So könnte das zu erstellende System davon abhängen, dass es mit einem bereits bestehenden System in seiner Umwelt Daten austauscht. Weitere Ressourcen sind der wichtige Bereich *Persistenz*, also Datenbanken, Dateien, etc. Auch Schnittstellen zu spezieller Hardware sind Ressourcen in der Umgebung eines Systems. Software zur Ansteuerung von Maschinen ist davon abhängig, dass die zu steuernden Maschinen sich in der Umwelt des Systems befinden.

Zur Darstellung des Systems in seiner Umwelt dient das *System-Umwelt-Diagramm*. Die folgende Abbildung zeigt ein einfaches System-Umwelt-Diagramm mit den verwendeten Symbolen.

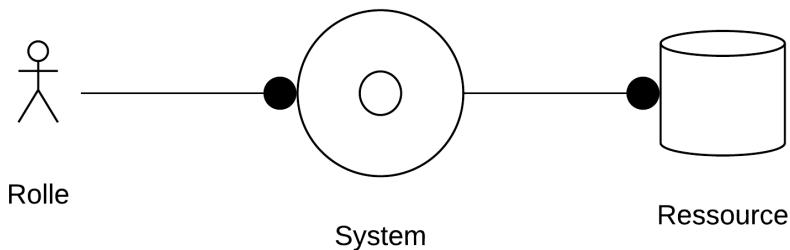


Abbildung 26: Die Symbole im System-Umwelt-Diagramm

Auf der linken Seite stehen die Rollen, rechts die Ressourcen. In der Mitte befindet sich das System. Die Abhängigkeiten verlaufen von den Rollen zum System und vom System zu den Ressourcen. Abhängigkeiten werden in Flow Design durch eine Linie mit einem Punkt am Ende dargestellt.

Mit *Rolle* werden Anwender des Systems bezeichnet, die jeweils unterschiedliche Tätigkeiten mit dem System durchführen. Mehrere Rollen werden unterschieden, wenn diese unterschiedliche Ansprüche an die Benutzerschnittstelle des System haben. Typisch für häufig wiederkehrende Tätigkeiten sind Benutzerschnittstellen, die zügig mit der Tastatur bedient

werden können. Für Tätigkeiten, die sehr oft durchgeführt werden, lohnt sich der Aufwand, die Bedienung mit Tastenkürzeln oder sogar Befehle auf der Kommandozeile zu erlernen. Da die wiederkehrenden Tätigkeiten dann schneller ausgeführt werden, als bei einer Bedienung mit der Maus, lohnt es sich, Tastenkürzel und Befehle etc. zu lernen. Noch extremer wird es im Bereich der Administration. Hier sind oft Kommandozeilenprogramme hilfreich, da sich diese leicht mithilfe von Skripten automatisieren lassen.

Am anderen Ende des Spektrums stehen Tätigkeiten, die eher selten ausgeführt werden. Für diesen Fall ist es angemessen, eine grafische Benutzerschnittstelle mit Mausbedienung zu realisieren. Ferner ist es für solche Rollen hilfreich, Tooltips, ein Hilfesystem, Wizards, etc. anzubieten. Statt nun alle Rollen mit einer einheitlichen Benutzerschnittstelle auszustatten, sollen die unterschiedlichen Ansprüche der Rollen in jeweils angemessener Weise differenziert umgesetzt werden. Spätestens wenn eine Rolle ebenfalls wieder ein System ist wird klar, dass nicht jede Rolle mit dem selben Zugang zum System ausgestattet werden kann. Die Anbindung eines Systems an das neu zu erstellende System erfolgt dann möglicherweise über https/REST, während die Anwender eine GUI erhalten.

Das System-Umwelt-Diagramm dient dazu, die unterschiedlichen Rollen zu identifizieren und darüber mit dem Product Owner ins Gespräch zu kommen. Aus Sicht des Product Owners ist dies wichtig, da es sich bei den Rollen und ihren Benutzerschnittstellen um einen Aspekt der Anforderungen handelt. Aus Sicht der Entwickler ist diese Information relevant, da sie für die unterschiedlichen Rollen möglicherweise unterschiedliche Benutzerschnittstellen realisieren müssen.

Auf der rechten Seite des System-Umwelt-Diagramms stehen die Ressourcen, von denen das System abhängig ist. Diese werden jeweils als Tonne visualisiert. Wichtig bei den Ressourcen ist die Sicht auf den Inhalt. Es geht nicht darum auszudrücken, dass das System eine Datenbank oder Dateien benötigt. Stattdessen wird inhaltlich zusammengetragen, von welchen Daten das System abhängt. So können in einem System bspw. Produktdaten und Bestellungen als Ressourcen modelliert werden. Die Frage nach der technischen Lösung wird erst später geklärt. Im System-Umwelt-Diagramm wird inhaltlich dargestellt, dass das System von den aufgeführten Ressourcen abhängt. Bei der Umsetzung kann die Antwort lauten, dass mehrere Ressourcen in ein und derselben Datenbank persistiert werden.

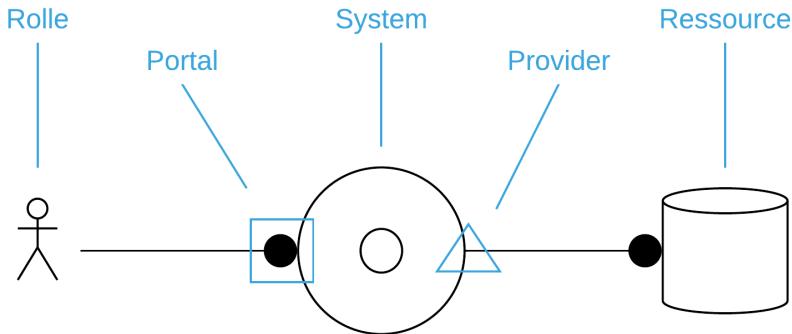


Abbildung 27: Portal, Provider und Domäne

Aus einem System-Umwelt-Diagramm ergibt sich bereits eine minimale Grobstruktur des Systems. Da die Wahrscheinlichkeit groß ist, dass sich in der Umgebung des Systems Veränderungen ergeben, diese jedoch nur einen geringen Einfluss auf den Kern des Systems haben sollen, wird das System gegen die Umwelt abgeschottet. Aus gutem Grund erinnert die Darstellung des Systems an eine Zelle, bestehend aus Zellkern und Membran. Aufgabe einer Zellmembran ist es, einen definierten Übergang zwischen "drinnen" und "draußen" zu schaffen. Dieses Konzept ist auch für Softwaresysteme nutzbar und wichtig. Es soll erreicht werden, dass Veränderungen in der Umwelt an der Membran des Systems behandelt werden können. Hier wird entschieden, wie der Zugriff einer Rolle auf das System erfolgt. Ändern sich die Anforderungen bzgl. des Zugangs, kann das Portal modifiziert werden. Der Kern bleibt konstant und ist von dieser Veränderung in der Umgebung nicht betroffen. Gleiches gilt für die Ressourcen. Sollen Daten zukünftig nicht mehr im Dateisystem sondern in einer Clouddatenbank abgelegt werden, wirkt sich dies auf den Provider aus. Dieser wird verändert, während der Kern des System stabil bleibt.

Da sich in der Umgebung des Systems ständig Veränderungen ergeben können, wird jeglicher Zugriff der Rollen auf das System jeweils über ein *Portal* geführt. Jede Rolle erhält ihr eigenes Portal. Damit lassen sich die unterschiedlichen Anforderungen an die Benutzerschnittstelle der Rolle jeweils in einem eigenen Portal realisieren. Menschliche Benutzer erhalten grafische Benutzerschnittstellen (GUIs) oder Kommandozeilenportale, während Fremdsysteme bspw. Zugriff über eine Web API erhalten. Portale werden durch Rechtecke dargestellt.

Auch die Abhängigkeit des Systems von den Ressourcen wird über eine dünne Softwareschnittstelle geführt. Das System greift nicht unmittelbar auf die Ressource zu, sondern diese Zugriffe erfolgen immer über einen *Provider*. Auch hier gilt, wie bei den Rollen, dass jede Ressource ihren eigenen Provider erhält. Auf diese Weise führt eine Veränderung bei der

Ressource nicht sogleich dazu, dass das System in seinem Kern angepasst werden muss. Es muss in der Regel lediglich im Provider auf die Veränderung reagiert werden. Provider werden als Dreiecke dargestellt.

Im Kern des Systems befindet sich das, was das System ausmacht: die *Domänenlogik*. Sie ist das Wesentliche eines jeden Softwaresystems. Aus diesem Grund ist es fundamental, dass dieser Kern gegenüber der Umwelt klar abgegrenzt wird. Eine Buchhaltung bleibt eine Buchhaltung, auch wenn sich in der Umgebung die Rollen oder Ressourcen verändern. Ob eine Buchhaltung über eine Desktop UI oder im Browser bedient wird, ändert nichts am Kern der Buchhaltung. Das gleiche gilt für die Ressourcen. Ob die Buchhaltungsdaten in deiner relationalen Datenbank abgelegt werden oder in der Cloud, ändert ebenfalls nichts am Kern der Buchhaltung. Aus diesem Grund muss der Kern jedes Softwaresystems frei gehalten werden von den Aspekten Benutzerschnittstelle und Ressourcenzugriff. Diese grobe Trennung der Aspekte ist die Mindestvoraussetzung für die Wandelbarkeit eines Softwaresystems.

Domänenlogik wird durch Kreise bzw. Ellipsen dargestellt.

Im nun folgenden Abschnitt wird kurz auf *User Stories* eingegangen. Dabei geht es in erster Linie darum herauszustellen, dass eine Ergänzung erforderlich ist, bevor der Übergang zum Entwurf auf gute Weise gelingen kann. Diese Ergänzung ist die Domänenzerlegung mit den Elementen *Dialog*, *Interaktion* und *Feature*.

User Stories

Softwareentwickler stehen vor der Herausforderung, Anforderungen auf eine Weise zu zerlegen und zu präzisieren, die es ihnen ermöglicht, in fast winzigen Schritten vorzugehen. Im Wasserfall wurden Zeiträume von Monaten betrachtet. Die Agilitätsbewegung, vor allem geprägt durch *Scrum*, führte zu Zeiträumen von wenigen Wochen. Doch auch 2 - 4 Wochen sind als Zeitraum zu lang. Das Feedback muss alle 1 - 2 Tage erfolgen. Auf diese Weise wird erreicht, dass das Team mit wenig Verschnitt exakt das produziert, was der Product Owner benötigt. Beim Autofahren kann Sekundenschlaf in den Graben führen. Damit vergleichbar können Feedbackzyklen von 2 - 4 Wochen ebenfalls "in den Graben" führen. Das Risiko derart langer Phasen ist hoch. Es wird potentiell in die falsche Richtung entwickelt, was durch die dann erforderlichen Korrekturen zu hohen Kosten führt. Erreicht werden muss in diesem Sinne, dass die Anforderungen so klein zerlegt werden, dass die Inkremeante in 1 - 2 Tagen realisiert werden können.

Gesucht ist demnach eine Vorgehensweise, die in jedem Fall zu Inkrementen, also vertikalen Durchstichen, führt. Ferner müssen die so entstehenden Ausschnitte des Gesamtsystems möglichst klein sein, weil dies die

Voraussetzung für kontinuierliches Feedback des Product Owners sicherstellt.

In agilen Entwicklungsprozessen wird häufig mit *User Stories* gearbeitet. Diese haben allerdings einen großen Nachteil: der Umfang einer User Story ist meist zu groß für ein kleinschrittiges Vorgehen in Inkrementen. Ferner geben User Stories nur sehr wenig Führung für den Analyseprozess. Stark vereinfacht gesagt, laufen User Stories darauf hinaus, einen Lückentext auszufüllen:

Als Rolle möchte ich Ziel/Wunsch damit ich Nutzen.

Das Verfeinern bzw. die Dekomposition von User Stories erfordert viel Erfahrung. Es ist kein einfacher schematischer Vorgang. Eine User Story kann sehr kleinteilig sein wie im folgenden Beispiel:

“Als Autor möchte ich nach dem Start der Anwendung mein zuletzt bearbeitetes Dokument sehen, um Zeit zu sparen.”

Diese User Story würden wir gemäß der im Folgenden beschriebenen Vorgehensweise als ein *Feature* bezeichnen. Die User Story befasst sich mit einem relativ kleinen Detail einer Anwendung. Ein weiteres Beispiel zeigt eine User Story, die zu groß für ein Inkrement wäre:

“Als Autor möchte ich Text in Fett/Kursiv/Unterstrichen auszeichnen können, um ein Buch schreiben zu können.”

Diese User Story würde sich in der im weiteren Verlauf vorgestellten Begrifflichkeit aus mehreren *Interaktionen* zusammensetzen (Text markieren, Auszeichnung ändern). Sie würde also eher einem *Dialog* entsprechen. Innerhalb der Methodik von User Stories könnte eine solche Story in zwei Stories aufgeteilt werden:

“Als Autor möchte ich Text markieren können, um mich auf einen Textausschnitt beziehen zu können.”

“Als Autor möchte ich einen markierten Text in Fett/Kursiv/Unterstrichen auszeichnen können, um ein Buch schreiben zu können.”

Nun wären zwei getrennte Benutzerinteraktionen erkennbar. Allerdings gibt es keine einfachen Regeln, die man beim Schreiben von User Stories anwenden kann, um sie zu einer angemessenen Größe zu führen. Als Ergänzung zu User Stories wird im folgenden Abschnitt vorgestellt, wie Anforderungen mit Dialogen, Interaktionen und Features zerlegt werden können. Diese Vorgehensweise bietet den Vorteil, dass sie unmittelbar zum Entwurf führt. Die einzelnen Entwürfe könne dadurch jeweils einer An-

forderung zugeordnet werden. Das vereinfacht die Nachvollziehbarkeit. Der wesentliche Vorteil der im folgenden beschriebenen Vorgehensweise ist, dass sie sehr schematisch angewandt werden kann.

Dialoge und Interaktionen

Jedes Softwaresystem verfügt über eine Benutzerschnittstelle. Der Anwender des Systems muss in die Lage versetzt werden, mit dem System zu interagieren, um eine gewünschte Funktionalität ausführen zu können. Bei Desktop Anwendungen ist die Benutzerschnittstelle eine grafische Schnittstelle, meist als GUI (*Graphical User interface*), bezeichnet. Webanwendungen präsentieren sich dem Anwender im Browser. In beiden Fällen können wir im Softwaresystem *Dialoge* identifizieren, mit deren Hilfe der Anwender mit dem System in Kontakt tritt.

Innerhalb eines Dialogs kann der Anwender mit den unterschiedlichen Steuerelementen, engl. Controls, interagieren. Er kann bspw. eine Schaltfläche betätigen oder einen Menüpunkt aus einem Menü auswählen. Diese Interaktion des Anwenders mit dem System ist der Grund dafür, dass wir *Domänenlogik* schreiben müssen. Ohne eine Interaktion des Anwenders mit dem System gäbe es überhaupt keinen Grund, das System zu erstellen. Letztlich lassen sich alle Anforderungen den Interaktionen eines Anwenders zuordnen. Oder um es mit den Begriffen des System-Umwelt-Diagramms auszudrücken: alle Anforderungen lassen sich den Interaktionen der Rollen mit dem System zuordnen. Die Interaktionen sind damit der Ursprung jeder Logik.

An dieser Stelle ist es wichtig, die unterschiedlichen Arten von Logik unterscheiden zu können. Im Vordergrund steht hier der Begriff der *Domänenlogik*. Damit wird jene Logik bezeichnet, die sich mit dem Thema der Anwendung befasst. In einer Buchhaltungssoftware ist Domänenlogik all die Logik, die sich mit dem Thema Buchhaltung befasst. In einem TicTacToe Spiel dreht sich bei der Domänenlogik alles um das Spiel und seine Regeln.

Definition: **Domänenlogik** ist solche Logik, die das Thema der Anwendung betrifft.

Darüberhinaus sind andere Kategorien von Logik erforderlich, um ein Softwaresystem realisieren zu können. Zum Speichern von Daten in einer Datenbank muss Datenbanklogik implementiert werden. Zur Anzeige von Daten muss UI Logik implementiert werden. Soll beim Start einer Anwendung ein bestimmter Dialog geöffnet werden, ist es auch dafür erforderlich, Logik zu implementieren. Vielleicht muss dazu in einer *Main* Methode ein Fenster erzeugt und angezeigt werden. Dies ist jedoch keine Domänenlogik. Denn die zeichnet sich dadurch aus, dass sie das Thema der Anwendung

betrifft. Das Öffnen eines Dialogs hat jedoch nichts speziell mit dem Thema einer bestimmten Anwendung zu tun sondern fällt in die Kategorie der Benutzerschnittstelle.

Halten wir also fest: die Interaktion eines Benutzers mit dem System ist der Grund dafür, dass wir Domänenlogik realisieren müssen. Die Interaktion findet jeweils innerhalb eines Dialogs statt. Ein Dialog fasst ggf. mehrere Interaktionen zusammen.

Zur Zerlegung der Anforderungen verwenden wir die *Interaktion* eines Benutzers mit dem Softwaresystem als zentrales Element. Jede Interaktion findet innerhalb eines Dialogs statt. Und jede Interaktion löst die Ausführung von Domänenlogik aus. Zur Darstellung skizzieren wir den Dialog und zeichnen die Interaktionen als Pfeile ein, wie es das Beispiel in der folgenden Abbildung zeigt.

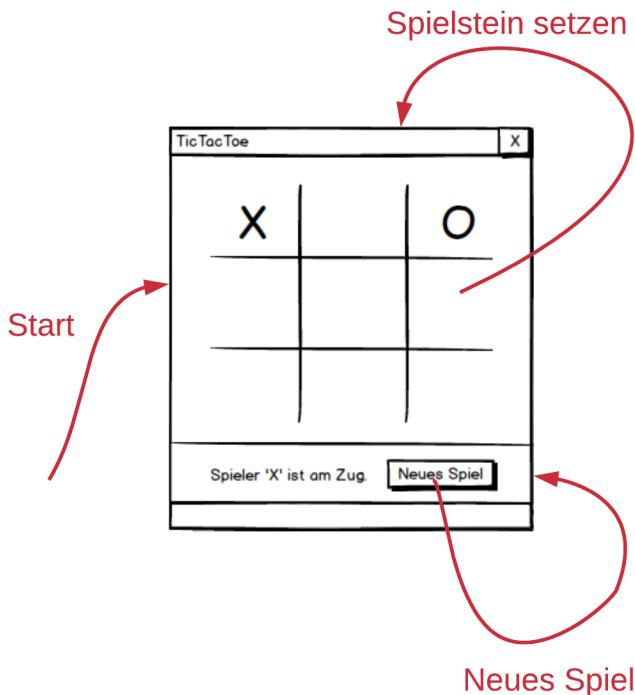


Abbildung 28: Dialog und Interaktionen in einem TicTacToe Spiel

Über diesen Dialog hat der Anwender die Möglichkeit, mit der Anwendung zu interagieren. Im Kern der Anwendung steht die Domänenlogik, die das TicTacToe Spiel realisiert. Diese Domänenlogik kann der Anwender verwen-

den, in dem er in einem Dialog eine Interaktion auslöst. Im Beispiel ist dazu eine grafische Benutzerschnittstelle realisiert. Alternativ könnte die Interaktion auch über ein Konsolen UI erfolgen. Die Domänenlogik im Kern des Softwaresystems würde sich dadurch nicht ändern. Relevant sind die Interaktionen. Sie bestimmen, welche Domänenlogik es geben muss und wie sich diese verhalten soll.

Im oben gezeigten Dialog für das TicTacToe Spiel gibt es einige Steuerelemente. Die Schaltfläche *Neues Spiel* löst eine Interaktion aus. Die *Schließen* Schaltfläche am rechten oberen Fensterrand ist dagegen keine Interaktion. Offensichtlich wird also nicht durch jede Schaltfläche eine Interaktion ausgeführt. Das Kriterium ist hier, ob Domänenlogik erforderlich ist. Es mag für beide Schaltflächen erforderlich sein, Logik zu implementieren. Allerdings ist nur die Logik, die durch die Schaltfläche *Neues Spiel* ausgelöst wird, Domänenlogik. Das Beenden der Anwendung mit der Schließen Schaltfläche ist nichts, was speziell mit dem Thema TicTacToe zu tun hätte. Insofern ist diese Schaltfläche für die Analyse und Zerlegung der Anforderungen nicht relevant.

Das sähe anders aus, wenn die TicTacToe Anwendung beim Beenden den Spielstand sichern soll, damit das Spiel beim erneuten Start an dieser Stelle fortgesetzt werden kann. In diesem Fall wäre Domänenlogik erforderlich, weil es nicht mehr nur um das Beenden der Anwendung im technischen Sinne gehen würde.

Features

Häufig ist eine einzelne Interaktion noch zu groß, um sie als Inkrement im Zeitrahmen von ein bis zwei Tagen zu realisieren. So kann sich hinter der Interaktion einer Schaltfläche ein komplizierter Satz von Anforderungen befinden. Ziel jeder Iteration ist ein Inkrement, ein vertikaler Durchstich durch die Anforderungen.

Sind die Anforderungen, die mit einer Interaktion verbunden sind, zu groß, unterteilen wir eine Interaktion in *Features*. Dies lässt sich am Beispiel des TicTacToe Spiels zeigen.

Die Interaktion *Spielstein setzen* besteht aus den folgenden Features:

- **Spielerwechsel**
Nach dem Zug wird ermittelt, welcher Spieler als nächstes an der Reihe ist.
- **Spielende erkennen**
Es wird erkannt, dass alle Felder besetzt sind und kein Zug mehr vorgenommen werden kann.
- **Gewinnermittlung**
Es wird erkannt, welcher Spieler gewonnen hat.

- **Gültigen Zug erkennen**

Es wird erkannt, ob ein Spielzug gültig ist.

Die Unterteilung der Interaktion in Features mag an diesem Beispiel künstlich erscheinen, sind doch die einzelnen Features sehr klein. Schon die Interaktion erscheint auf den ersten Blick nicht weiter teilbar. Doch selbst eine derart kleine Interaktion ist noch zerlegbar. In der Praxis fällt es Teams anfangs meist schwer, eine Interaktion in Features zu zerlegen. Es hilft hier, sich die Features als Funktionalität vorzustellen, die zunächst nicht realisiert wird. Es geht also eher darum zu überlegen, was zunächst weggelassen werden kann. Ziel ist nicht eine vollständige Zerlegung einer Interaktion in ihre Bestandteile. Stattdessen geht es darum, dass durch Weglassen ein so kleiner Ausschnitt der Anforderungen entsteht, dass dieser in 1-2 Tagen realisiert werden kann. Später werden dann die weiteren Features der Interaktion ergänzt.

Lässt man bei der Interaktion *Spielstein setzen* des TicTacToe Spiels alle oben aufgeführten Features weg, bleibt eine sehr simple Benutzeroberfläche übrig, in der man in die Spielfelder klicken kann, um dort einen Spielstein zu platzieren. Es wird immer die gleiche Form platziert, also entweder ein X oder ein O. Es erfolgt keine Prüfung, ob in dem Feld bereits ein Spielstein platziert wurde, der Spielerwechsel wird nicht durchgeführt, etc. Natürlich ist ein solches sehr vereinfachtes Inkrement noch nicht an den Endkunden auslieferbar. Und dennoch kann der Product Owner wichtiges Feedback geben. Er hat nun mehr als einen UI Entwurf auf Papier vorliegen. Diese UI kann er schon bedienen und dabei herausfinden, ob das Bedienungskonzept aus seiner Sicht funktioniert oder angepasst werden muss. Beim Zerlegen der Anforderungen sollte immer die Frage im Vordergrund stehen, ob mit dem kleinen Ausschnitt von Anforderungen Feedback des Product Owners generiert werden kann. Wesentlich dabei ist der vertikale Schnitt durch die Anforderungen.

Im nun folgenden Kapitel wird der Übergang von den Anforderungen zum Entwurf der Lösung vollzogen. Ging es bei der Zerlegung der Anforderungen noch darum, mit dem Product Owner ins Gespräch zu kommen, um einige Details aufzudecken, geht es beim Entwurf darum, eine Lösung für die Anforderungen zu entwickeln. Dies ist nicht zu verwechseln mit der Umsetzung der Lösung. Das Codieren folgt erst nach dem Entwurf der Lösung.

Der Übergang von den Anforderungen zum Entwurf

Ausgehend von den Dialogen und Interaktionen können wir mit dem Entwurf einer Lösung beginnen. Das Suchen der Dialoge und Interaktionen zählt noch zur Analyse der Anforderungen. An dieser Stelle findet nun der Übergang zur Lösungssuche statt. Üblicherweise springen Entwickler spätestens jetzt gleich zur Implementation. Die Lösungssuche bzw. das gezielte Entwerfen einer Lösung findet dann nicht getrennt von der Umsetzung statt. Das hat allerdings zur Folge, dass Lösungssuche und Umsetzung der Lösung als Tätigkeiten ineinander übergehen. Hinzu kommt, dass auf diese Weise Quellcode auch zur Darstellung und Diskussion der Lösung verwendet wird. Quellcode stellt jedoch sehr hohe syntaktische Anforderungen, weit höher, als ein Diagramm in UML oder Flow Design. Auch ein Diagramm muss syntaktischen Anforderungen genügen. Der Nutzen eines Diagramms geht verloren, wenn die verwendeten Formen keine klare Bedeutung haben. So repräsentiert ein Pfeil in Flow Design einen Datenfluss. Es wäre wenig hilfreich, den Pfeil gleichzeitig auch mit einer anderen Bedeutung zu versehen. In dem Fall wäre nicht mehr ersichtlich, welche Bedeutung das Diagramm hat. Das gleiche gilt für Quellcode. Auch hier muss die Syntax eingehalten werden. Allerdings ist es weitaus anspruchsvoller, die sehr detailreiche Syntax einer Programmiersprache einzuhalten, als die einfache Syntax eines Diagramms. Ein Diagramm ist abstrakter als Quellcode und unterstützt dadurch den kreativen Prozess des Entwurfs einer Lösung viel besser, als dies mit Code durchführbar wäre. Dabei ist gerade der Verzicht auf Details der Schlüssel zur Kreativität.

Durch die Vermischung von Lösungssuche und Umsetzung ist es nicht möglich, als Team gemeinsam eine Lösung zu entwerfen und diese dann, möglicherweise sogar arbeitsteilig, umzusetzen. Eine Trennung von Entwurf der Lösung und Umsetzung der Lösung ist dringend erforderlich. Es muss sich um zwei getrennte Phasen handeln. Der Übergang muss bewusst erfolgen und nicht unbewusst und unreflektiert. Überhaupt sollte Entwicklern jederzeit klar sein, in welcher Phase sie sich gerade befinden.

Entwurf der obersten Ebene

Der Übergang von Dialogen und Interaktionen zum Entwurf der Lösung ist einfach: Jede Interaktion wird in ein Flow Design Diagramm überführt. Dazu wird die Interaktion mit dem Kreis als Symbol für Domänenlogik aufgezeichnet. Die Dialoge, bei denen die Interaktion beginnt und endet, werden

jeweils an den Anfang und das Ende eines Datenflusses gestellt. In der Mitte steht die Domänenlogik der Interaktion. Für die Interaktion *Spielstein setzen* aus dem TicTacToe Beispiel sieht das wie folgt aus.

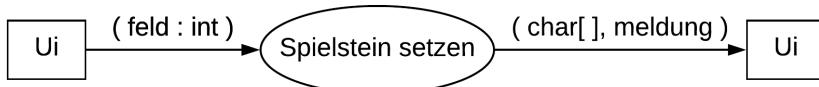


Abbildung 29: Entwurf der obersten Ebene für die Interaktion *Spielstein setzen*

Solange das Softwaresystem nur aus einem einzigen Dialog besteht oder jedenfalls nur ein einzelner Dialog betrachtet wird, können wir diesen mit "Ui" bezeichnen. Sind zwei unterschiedliche Dialoge beteiligt, müssen sie sofort explizit benannt werden, um sie unterscheiden zu können. Die Daten fließen aus der Ui zur Domänenlogik und wieder zurück zur Ui.

Dieser Schritt stellt im Grunde keine große Herausforderung dar. Es geht im wesentlichen darum, die Pfeile aus dem Interaktionsdiagramm in ein Flow Design Diagramm zu überführen.

Das Erstellen der Grundstruktur ist simpel. Der kreative Akt besteht darin herauszuarbeiten, welche Art von Daten auf den Datenflüssen transportiert werden sollen. Im TicTacToe Beispiel muss die Ui der Domänenlogik eine Information darüber liefern, in welches Feld der Benutzer einen Spielstein setzen möchte. Ferner muss die Domänenlogik Informationen zurück zur Ui liefern, damit diese den veränderten Spielstand anzeigen kann. Ganz wichtig: die Ui selbst trägt natürlich nicht zur Lösung des Problems TicTacToe bei. Die gesamte Domänenlogik liegt in der Mitte, in der Funktionseinheit *Spielstein setzen*.

Die nachfolgende Abbildung zeigt die oberste Ebene des Entwurfs für alle Interaktionen des TicTacToe Spiels.

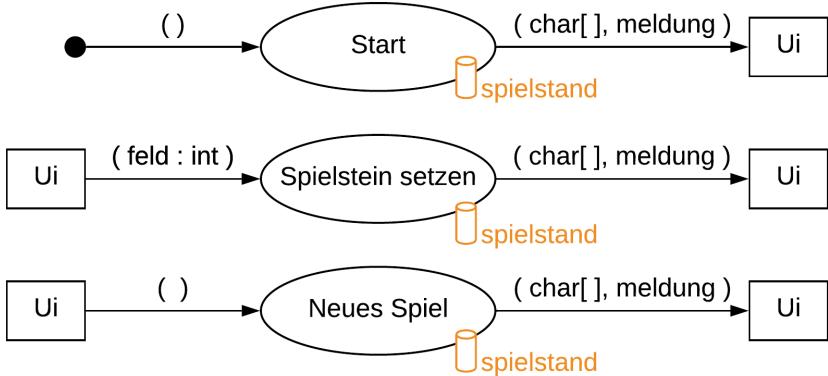


Abbildung 30: Entwurf der obersten Ebene für TicTacToe

Entwurf in die Breite vs. Tiefe

Die Herausforderung im Entwurf der obersten Ebene besteht vor allem darin, die Datenflüsse mit geeigneten Datentypen zu versehen. Oft ist es hilfreich, dazu alle Interaktionen in einen Entwurf der obersten Ebene zu überführen. Später wird zuerst eine Auswahl einer Interaktion erfolgen und nur diese wird dann weiter verfeinert und umgesetzt. Der Entwurf aller Interaktionen geht somit zunächst in die Breite, nicht in die Tiefe. Die folgende Abbildung verdeutlicht den Unterschied. Auf der linken Seite sind fünf Interaktionen auf oberster Ebene entworfen. Dies stellt einen Entwurf in die Breite dar, weil einerseits mehrere Interaktionen betrachtet werden. Andererseits wird dabei auf die Details jeder einzelnen Interaktion verzichtet, sie werden jeweils nur oberflächlich betrachtet.

Die rechte Seite zeigt dagegen den Entwurf einer einzelnen Interaktion. Hier wird statt in die Breite in die Tiefe gegangen. Es werden die Details einer einzigen Interaktion betrachtet.

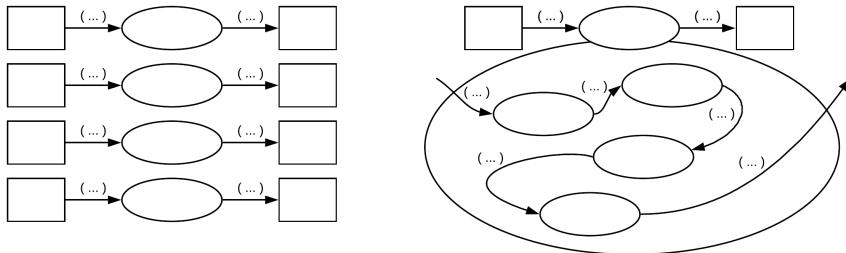


Abbildung 31: Entwurf in die Breite vs. Tiefe

In dieser Phase des Entwicklungsprozesses, dem Übergang von der Anforderungsanalyse zum Entwurf, geht es darum, die Zusammenhänge zu sehen. Es muss entschieden werden, welche Daten bei den einzelnen Interaktionen fließen sollen. Oft beeinflussen sich die Entscheidungen gegenseitig. Beim TicTacToe Spiel hängt das Datenformat der Interaktion *Spielstein setzen* davon ab, welche Daten von *Start* an die UI geliefert werden. Wird von *Start* ein zweidimensionales Array verwendet, kann *Spielstein setzen* zwei Koordinaten bestehend aus *x* und *y* liefern. Alternativ kann auch eine Liste von Spielfeldern verwendet werden, auf die sich *Spielstein setzen* dann mit einem Index bezieht. Insofern ist es hilfreich, die in einem Dialog identifizierten Interaktionen einmal in der Breite zu entwerfen, um eine für alle Interaktionen angemessene Form der Daten zu finden. Auf der anderen Seite sollte der Entwurf hier nicht zu viel vorwegnehmen. Fällt es schwer, zu einer Interaktion ein geeignetes Datenformat zu finden, sollte dieser Punkt offen gelassen werden. Sobald die betreffende Interaktion vom Product Owner für die nächste Iteration ausgewählt wurde, muss sie ohnehin verfeinert werden. Bei der Verfeinerung einer Interaktion wird die Aufgabenstellung viel tiefer durchdrungen, so dass es dann leichter fällt, eine gute Entscheidung für die Datentypen zu treffen. Es gilt hier also, eine Balance zu finden. Einerseits können Entscheidungen in der Breite für mehrere Interaktionen getroffen werden, andererseits können Entscheidungen verschoben werden bis zu dem Moment, an dem die einzelne Interaktion präzisiert wird.

Ein weiterer Aspekt des Entwurfs in die Breite ist die Frage, wo der Zustand der Anwendung gehalten wird. Im Entwurf der obersten Ebene des TicTacToe Spiels ist der Spielstand als Zustand an den Funktionseinheiten *Start*, *Spielstein setzen* und *Neues Spiel* notiert. Dies drückt aus, dass alle drei Funktionseinheiten auf gemeinsamen Zustand zugreifen. *Start* und *Neues Spiel* setzen diesen Zustand in einen initialen Zustand zurück, während *Spielstein setzen* den Spielstand verändert. Alternativ könnte der Zustand auch in der UI gehalten werden. Diese wichtige Entscheidung kann mit Hilfe des Entwurfs der obersten Ebene visualisiert werden. Auf diese Weise wird eine fundierte Diskussion über die Vor- und Nachteile der jeweiligen Entscheidung unterstützt.

Flow Design: Datenflussdiagramme

Datenflüsse

Beim Entwurf der Lösung für die in der Analysephase gefundenen Interaktionen kommen bei Flow Design *Datenflussdiagramme* zum Einsatz. Datenflussdiagramme sind nicht die einzige Art von Diagrammen. Beim Entwurf von Softwaresystemen kommen die drei folgenden Kategorien von Diagrammen zum Einsatz:

- **Abhängigkeitsdiagramme**
Zeigen die Abhängigkeiten von Funktionseinheiten. Das *Klassendiagramm* der UML ist der wichtigste Vertreter dieser Kategorie.
- **Kontrollflussdiagramme**
Zeigen, auf welche Weise die Kontrolle von einer zur nächsten Funktionseinheit fließt. In der UML gehört das *Aktivitätsdiagramm* in diese Kategorie. Aber auch *Struktogramme* und *Programmablaufpläne* gehören in die Kategorie der Kontrollflussdiagramme.
- **Datenflussdiagramme**
Zeigen, wie die Daten zwischen den Funktionseinheiten fließen. In den *Aktivitätsdiagrammen* der UML können auch Datenflüsse dargestellt werden. Ferner werden in der Strukturierten Analyse nach Tom DeMarco Datenflussdiagramme verwendet.

Die in diesem Buch dargestellten *Flow Design* Diagramme gehören zur Kategorie der Datenflussdiagramme. Mit *Flow Design* bezeichnen wir das methodische Vorgehen. Ergebnis dieser Vorgehensweise sind *Flow Design Diagramme*, die als Grundlage für die Implementation dienen. Als Kurzform verwenden wir den Begriff *Flow Design* auch stellvertretend für ein *Flow Design Diagramm*.

Die verwendete Syntax der *Flow Design* Diagramme ist ganz absichtlich sehr einfach gehalten. Unserer Beobachtung nach wird in Softwareentwicklungsprojekten nur sehr selten konsequent entworfen. Die in der UML angebotenen Diagrammarten sind vielfältig und detailreich. Die Verwendung der UML hat zur Folge, dass man sich in neun Diagrammarten einarbeiten muss. Dabei muss man mindestens so tief einsteigen, dass man entscheiden kann, welche Diagrammarten man verwenden möchte und auf welche man bewusst verzichtet. Mit *Flow Design* stellen wir eine Methode vor, die einerseits mit deutlich weniger auskommt, andererseits trotzdem als Grundlage für die nachfolgende Implementation dienen kann. Ziel des Entwurfs ist es, die Lösung so detailreich darzustellen, dass alle relevanten Aspekte durchdacht wurden. Dies ist mit *Flow Design* möglich.

Flow Design Diagramme bestehen aus zwei wesentlichen Elementen: *Funktionseinheiten* und *Datenflüsse*. Eine Funktionseinheit verarbeitet Daten. Das zugrunde liegende Prinzip ist ein Kernkonzept der Datenverarbeitung: Eingabe - Verarbeitung - Ausgabe, oder auch EVA.

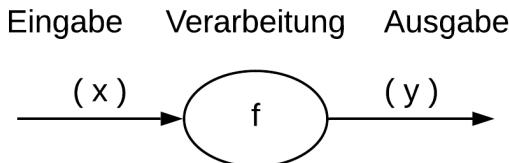


Abbildung 32: EVA: Eingabe - Verarbeitung - Ausgabe

Eine Funktionseinheit erhält Daten als Eingabe, verarbeitet diese und liefert ein Resultat als Ausgabe. Ferner kann eine Funktionseinheit zu einem Seiteneffekt führen. Das Speichern von Daten in einer Datei oder die Ausgabe auf dem Bildschirm sind solche Seiteneffekte. In diesem Fall muss eine Funktionseinheit keine Ausgabe erzeugen, sondern führt intern lediglich zum Seiteneffekt.

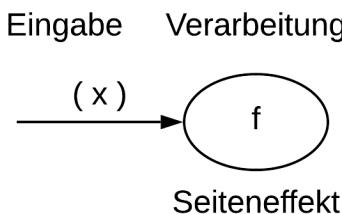


Abbildung 33: Eingabe - Verarbeitung - Seiteneffekt

Ein Datenfluss wird durch einen Pfeil dargestellt. Am Pfeil sind die Daten angeschrieben, die fließen. Ganz absichtlich werden die Daten dabei in Klammern gesetzt, um syntaktische Feinheiten unterscheiden zu können. Funktionseinheiten können auch ohne Eingabedaten auftreten. In diesem Fall enthält der Datenfluss leere Klammern, um anzudeuten, dass keine Daten fließen. Was bedeutet das aber im Detail, wenn auf einem Datenfluss keine Daten fließen? Insbesondere geht es hier um die Frage, was der Unterschied zwischen den folgenden Datenflüssen ist.

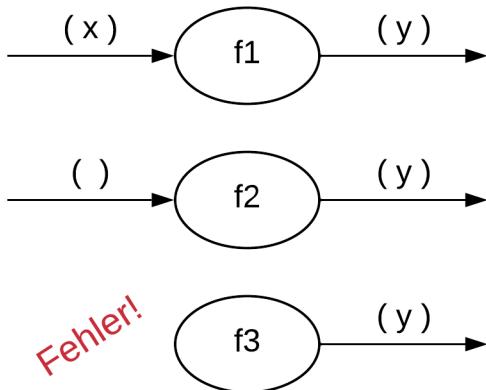


Abbildung 34: Datenfluss mit einem (x) vs. Datenfluss ohne Daten vs. kein eingehender Datenfluss

Der Unterschied wird deutlich, wenn wir uns klarmachen, dass mit den Daten auch die Kontrolle fließt. In der Abbildung zeigt f_1 eine Funktionseinheit, in die ein x als Eingabe fließt und die daraus ein y produziert. Eingabe - Verarbeitung - Ausgabe.

Bei Funktionseinheit f_2 ist das etwas anders. Hier ist die Eingabe leer. Jedoch fließt mit der leeren Eingabe die Kontrolle zu f_2 . Durch den leeren Datenfluss wird ausgedrückt, dass nun f_2 mit der Verarbeitung an der Reihe ist.

Und genau darin liegt das Problem bei f_3 . Wir können ohne eingehenden Datenfluss nicht erkennen, zu welchem Zeitpunkt f_3 ein y produziert.

Daraus können wir ableiten: jede Funktionseinheit muss über einen eingehenden Datenfluss verfügen, weil mit dem Datenfluss gleichzeitig die Reihenfolge der Ausführung ausgedrückt wird. Benötigt eine Funktionseinheit keine Eingabedaten, wird durch einen leeren eingehenden Datenfluss ausgedrückt, zu welchem Zeitpunkt die Funktionseinheit die Kontrolle erhält. Alle Datenflüsse sind bis auf weiteres *synchron sequentiell*. Die einzelnen Funktionseinheiten sind also nicht autonom sondern es wird zu einem Zeitpunkt immer genau eine Funktionseinheit ausgeführt. Die Reihenfolge der Ausführung ergibt sich aus der Richtung der Datenflüsse. Der einfachste Fall ist ein linearer Datenfluss, wie er in der folgenden Abbildung zu sehen ist.

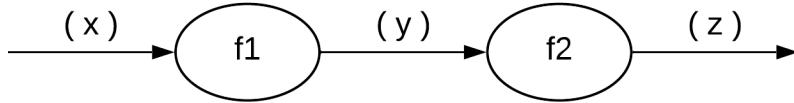


Abbildung 35: Linearer Datenfluss

Es ist offensichtlich, dass bei synchron sequentieller Ausführung zunächst $f1$ ausgeführt wird und im Anschluss $f2$. Diese Ausführungsreihenfolge kann weder umgedreht werden, noch können die Funktionseinheiten gleichzeitig parallel ausgeführt werden. Dieser Zusammenhang ergibt sich eindeutig aus dem Datenfluss. Die folgende Abbildung zeigt dagegen einen Datenfluss, bei dem unterschiedliche Ausführungen möglich sind.

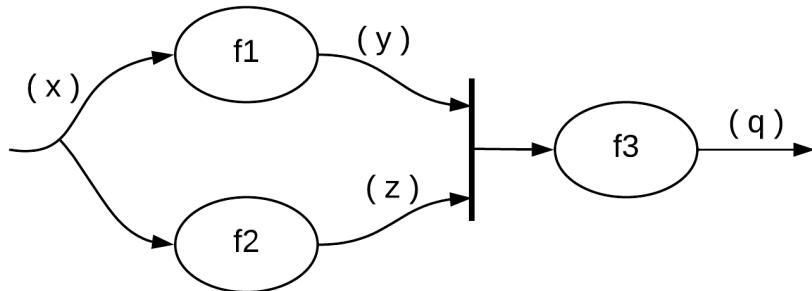


Abbildung 36: Paralleler Datenfluss

Hier spaltet sich der Datenfluss auf. Das x fließt sowohl zu $f1$ als auch zu $f2$. In diesem Fall spielt es keine Rolle, ob $f1$ oder $f2$ zuerst ausgeführt wird. Insbesondere wäre auch eine parallele, gleichzeitige Ausführung möglich. Ohne weitere Annotation gelten die Ausführungen allerdings immer als synchron sequentiell. Bei der Implementation muss der Entwickler somit entscheiden, ob $f1$ oder $f2$ zuerst aufgerufen wird. Der Entwurf bringt klar zum Ausdruck, dass es semantisch keine Rolle spielt, welche Funktionseinheit zuerst an die Reihe komme. Relevant ist lediglich, dass es mit $f3$ erst weitergehen kann, wenn zuvor sowohl $f1$ als auch $f2$ ausgeführt wurden.

Abhängigkeiten

Neben den Datenflüssen ist es manchmal hilfreich, auch Abhängigkeiten ausdrücken zu können. Dazu wird in Flow Design ein Strich mit einem Punkt

am Ende verwendet. Die folgende Abbildung zeigt ein Abhängigkeitsdiagramm.

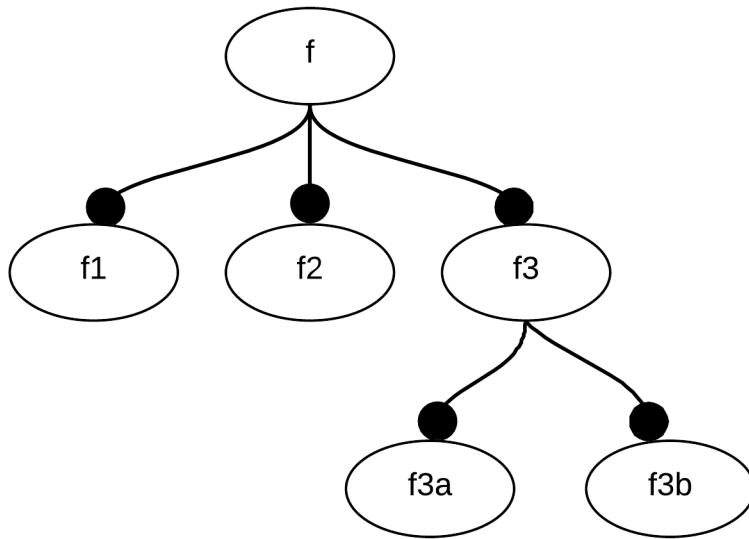


Abbildung 37: Abhängigkeiten

Das Diagramm drückt aus, dass f abhängig ist von f_1 , f_2 und f_3 . Ferner ist f_3 abhängig von f_{3a} und f_{3b} .

Abhängigkeiten spielen bei Flow Design eine untergeordnete Rolle. Durch eine klare Trennung von *Integration* und *Operation* wird erreicht, dass Abhängigkeiten als eigenständiger Aspekt betrachtet werden. Im Wesentlichen beschäftigt sich der Entwurf damit, wie die Daten zwischen den einzelnen Funktionseinheiten fließen. Abhängigkeiten sind unvermeidlich, um die Datenflüsse realisieren zu können. Sie spielen allerdings während der wichtigsten Tätigkeit, dem Entwurf der Datenflüsse, keine Rolle. Die folgende Abbildung zeigt erneut das Datenflussdiagramm zum Problem ToDictionary.

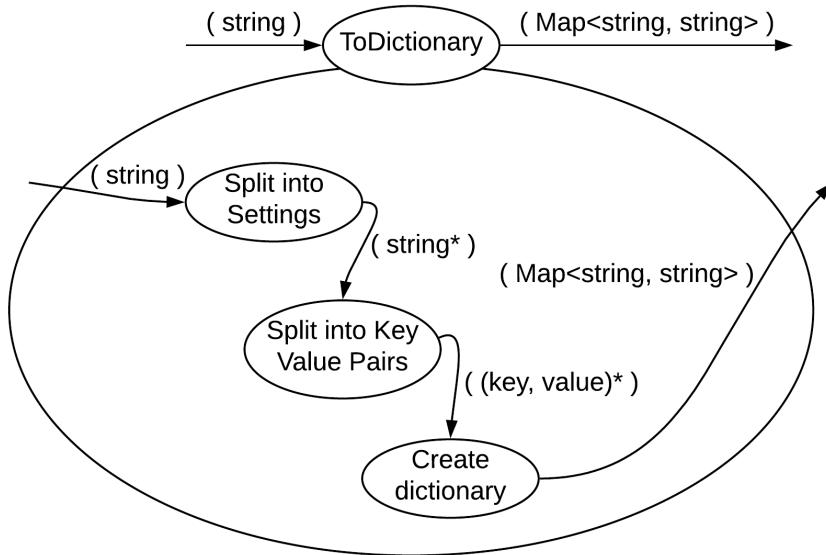


Abbildung 38: *ToDictionary* Datenflussdiagramm

Aus diesem Datenflussdiagramm ergibt sich ganz offensichtlich, dass die Funktionseinheit *ToDictionary* von den drei Funktionseinheiten *Split into Settings*, *Split into KeyValue Pairs* und *Create Dictionary* abhängig ist.

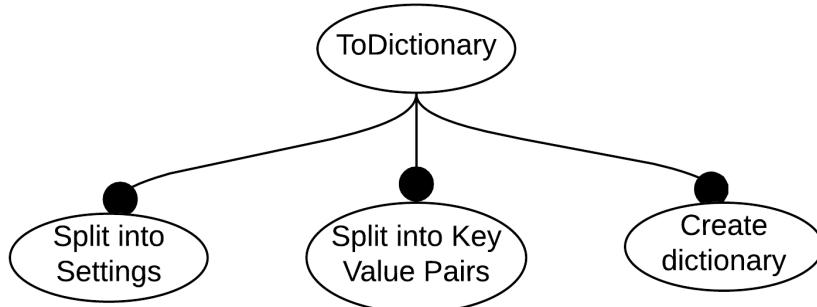


Abbildung 39: *ToDictionary* Abhängigkeitsdiagramm

Die Funktionseinheit *ToDictionary* ist durch den Entwurf verfeinert worden. Damit ist sie zu einer *Integration* geworden. Die Aufgabe von *ToDictionary* ist es, die drei anderen Funktionseinheit so zu integrieren, dass der gezeigte Datenfluss hergestellt wird. Da die drei anderen Funktionseinheiten nicht

weiter verfeinert wurden, sind sie *Operationen*. Sie tragen zur Lösung des Problems bei.

Aus einem Datenflussdiagramm ergeben sich somit immer Abhängigkeiten zwischen den Funktionseinheiten. Da diese strukturell auf immer gleiche Weise entstehen, müssen sie normalerweise nicht in Form eines Abhängigkeitsdiagramms näher betrachtet werden.

Verfeinerung des Entwurfs durch Zerlegung

Hierarchische Zerlegung

Der Entwurf einer Lösung muss so detailliert sein, dass die Lösung anschließend implementiert werden kann. Ein Entwurf dient dem Entwickler als Vorlage für die Implementation. Während der Implementation soll nicht mehr über die Lösung des Problems nachgedacht werden. Es geht bei der Implementation darum, eine fertig entworfene Lösung zu realisieren. Natürlich denkt der Entwickler im Kleinen darüber nach, wie er den Entwurf in Quellcode überführt. Er sucht also nach einer Übersetzung des Flow Design Diagramms in Quellcode. Diese Übersetzung ist auf eine überschaubare Anzahl Muster begrenzt, so dass dieser Vorgang recht schematisch erfolgt. Innerhalb einer Funktionseinheit kann es bei der Implementation darum gehen, über die Lösung der anstehenden Aufgabe genau dieser Funktionseinheit nachzudenken. Manchmal muss ein Algorithmus gefunden werden, manchmal ein technisches Problem gelöst werden. Das große Ganze liegt aber bereits vor ihm. Insbesondere die Struktur der Lösung, bestehend aus Klassen und Methoden, ist vollständig entworfen und muss quasi nur "abgetippt" werden.

Um die Analogie zum Hausbau aufzugreifen: bevor ein Haus gebaut wird, muss es geplant werden. Der Plan enthält alle relevanten Informationen für die Umsetzung. Während der Umsetzung wird nicht darüber nachgedacht, welche Stärke eine Wand haben soll oder in welchem Abstand Wände gemauert werden sollen. Diese Informationen gehen alle aus dem Plan hervor. Es wird klar getrennt zwischen den Phasen Entwurf einer Lösung und Umsetzung dieser Lösung. Und auch hier kommt es bei der Umsetzung zu Detailfragen. Der Maurer muss entscheiden, auf welche Weise er die zur Verfügung stehenden Steine zu einer Wand zusammenfügt. Im Flow Design müssen die erstellten Entwürfe so detailliert sein, dass sie umgesetzt werden können, ohne während der Umsetzung weiter am Entwurf der Lösung arbeiten zu müssen.

Auf der anderen Seite dürfen die Pläne natürlich nicht zu viele Details enthalten. Dann würden sich die Phasen Entwurf und Umsetzung überlappen. Es würden dann im Plan Details vorweggenommen, die in die Zuständigkeit der Umsetzung fallen. Müssen zum Beispiel Daten in einer Schleife bearbeitet werden, ist es Aufgabe des Entwurfs, dies auszudrücken. Aufgabe der Implementation ist es, zu entscheiden, ob dabei eine *for*-Schleife, eine *foreach*-Schleife oder ein funktionaler Ansatz mit map/reduce zum Einsatz kommt. Diese Details werden im Entwurf nicht

vorweggenommen. In der Analogie zum Hausbau: in der Zeichnung mit den Abmessungen ist nicht vorgegeben, wieviele Steine nebeneinander und übereinander gesetzt werden. Es sind lediglich die Wandstärke und die Maße der Wand vorgegeben. Die Entscheidung darüber, wie die Steine zu einer Wand zusammengesetzt werden, wird bei der Umsetzung getroffen. Natürlich werden dabei die "Regeln der Kunst" beachtet. Das gilt auch für die Umsetzung eines Flow Design Diagramms: die Umsetzung, also die Codierung in einer Programmiersprache, erfolgt nach den Regeln der Kunst, also bspw. unter Verwendung automatisierter Tests.

Eine andere Analogie ist eine Landkarte. Eine Landkarte enthält gerade so viele Details, dass damit eine Routenplanung möglich ist. Zu viele Details würden die Routenplanung und die Orientierung erschweren. Eine Landkarte enthält ganz absichtlich nicht jeden Baum, sondern lediglich die Information, ob es sich um freie oder bewaldete Flächen handelt. Auf der anderen Seite müssen ausreichend viele Details in der Karte enthalten sein, so dass eine Orientierung in der realen Umgebung möglich ist. Auf die gleiche Weise soll ein Flow Design Entwurf so viele Details enthalten, dass damit eine Orientierung in der Implementation möglich ist. Zu viele oder auch zu wenige Details würden diese Orientierung erschweren.

Damit ein Flow Design Entwurf das richtige Maß an Details für die nachfolgende Implementation enthalten kann, muss eine solcher Entwurf hierarchisch angelegt werden können. Auf der obersten Ebene steht ein Datenfluss von der UI zur Domänenlogik und wieder zurück zur UI. Diese Ebene hilft bei der Orientierung. Vor der Umsetzung müssen einzelne Funktionseinheiten jedoch verfeinert werden. Dies geschieht in Form einer hierarchischen Zerlegung. Die folgende Abbildung zeigt dies am Beispiel der *ToDictionary* Methode.

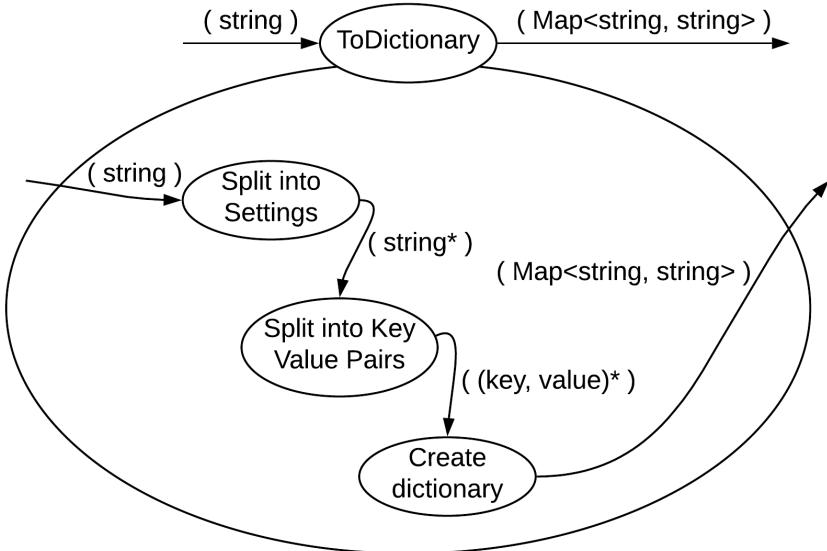


Abbildung 40: Entwurf von *ToDictionary*

Auf der obersten Ebene sehen wir die Funktionseinheit *ToDictionary*. Ein *string* fließt als Eingabe hinein und die Funktionseinheit produziert daraus eine *Map<string, string>*. Wie das Problem genau gelöst wird, ist der Verfeinerung zu entnehmen. Die Funktionseinheit *ToDictionary* integriert drei Funktionseinheiten, die in der gezeigten Weise nacheinander die Eingabe zur Ausgabe verarbeiten. Zunächst wird die Eingabe von *SplitIntoSettings* in mehrere *strings* zerlegt. Diese *strings* werden anschließend von *SplitIntoKeyValuePairs* in eine Aufzählung von Paaren zerlegt. Jedes Paar besteht aus einem *Key* und einem *Value*. Zuletzt fügt dann die Funktionseinheit *CreateDictionary* die Paare in ein Dictionary ein, welches als Resultat zurückgegeben wird.

Durch die hierarchische Zerlegung ist nun klar, wie das Problem gelöst wird. Der Entwurf stellt die Lösung der Aufgabe dar. Auf Basis dieses Entwurfs kann die Lösung dann umgesetzt, implementiert, werden.

Bei einer hierarchischen Zerlegung einer Funktionseinheit muss darauf geachtet werden, dass die Verfeinerung syntaktisch zur übergeordneten Funktionseinheit passt. Fließen oben Daten in die Funktionseinheit hinein, können in der Verfeinerung nur genau diese Daten verwendet werden; die Anschlüsse müssen passen. Das gleiche gilt für den ausgehenden Datenfluss. Auch hier muss sichergestellt sein, dass die Verfeinerung exakt die Daten liefert, die auf der übergeordneten Ebene angezeigt sind.

Die Zerlegung kann hierarchisch über beliebig viele Ebenen erfolgen. Folgende Abbildung zeigt das auf abstrakte Weise.

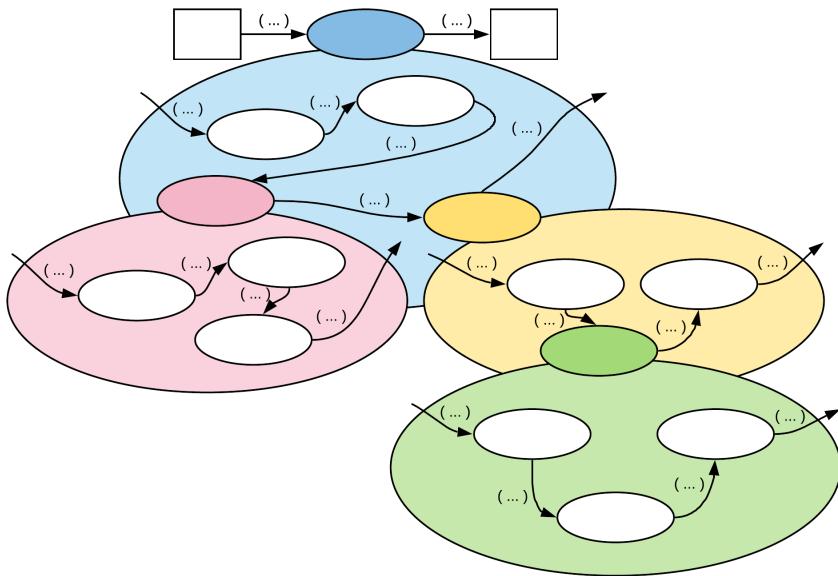


Abbildung 41: Hierarchische Zerlegung über mehrere Ebenen

Die oberste Funktionseinheit ist zerlegt in fünf Funktionseinheiten. Zwei davon sind wiederum zerlegt. Innerhalb der linken Zerlegung ist wiederum eine Funktionseinheit weiter zerlegt. Diese Hierarchie kann beliebig tief fortgeführt werden. Dadurch sind auch komplexe Probleme mit Flow Design darstellbar.

Bei der Zerlegung einer Funktionseinheit stellt sich regelmäßig die Frage, ob die Zerlegung bereits ausreichend detailliert ist. Zu jeder Funktionseinheit können dabei zwei Fragen betrachtet werden:

1. Ist die Funktionseinheit für genau einen Aspekt zuständig?
2. Ist die Funktionseinheit ausreichend klein, so dass sie zügig implementiert werden kann?

Die Frage nach den Aspekten ist wichtig für die Wandelbarkeit. Dabei stellt sich zunächst die Frage, was mit **Aspekt** gemeint ist. Dazu eine Definition:

Definition: Ein **Aspekt** ist eine Zusammenfassung von zusammengehörenden Eigenschaften, die sich getrennt von anderen Eigenschaften verändern können.

Dies entspricht der umgangssprachlichen Verwendung des Begriffs. Ein Aspekt meiner Persönlichkeit ist meine Kleidung. Dieser Aspekt besteht aus verschiedenen Eigenschaften wie Farben, Stoffe, Oberbekleidung etc. Ein weiterer Aspekt meiner Persönlichkeit sind meine Hobbys. Die beiden

Aspekte können sich unabhängig voneinander verändern. Ob ich lieber ein Hemd oder lieber ein Poloshirt anziehe ist unabhängig davon, ob mein Hobby das Trompetenspiel oder die Fotografie ist.

Weil sich Aspekte unabhängig voneinander verändern können, ist es wichtig, sie im Entwurf zu trennen. Je klarer die Aspekte getrennt sind, desto einfacher ist es, auf Veränderungen zu reagieren. Heute kommen die Daten aus einer Datei, morgen aus einer Datenbank. So lange es eine getrennte Funktionseinheit für den Aspekt "Datenquelle" gibt, fällt die Änderung leicht. Ist der Aspekt jedoch mit anderen Aspekten vermischt, sind plötzlich andere Funktionseinheiten von der Änderung betroffen. Es ist daher zwingend notwendig, die Aspekte im Entwurf zu identifizieren und zu trennen.

Die zweite Forderung an Funktionseinheiten sagt etwas über die Dauer der Implementation aus. Die Empfehlung lautet, dass jede Funktionseinheit in maximal vier Stunden zu implementieren ist. Stellt ein Entwickler im Team dies während des Entwurfs in Frage, sollte die Funktionseinheit in der Entwurfsphase verfeinert werden. So wird sichergestellt, dass keine Überraschungen während der Umsetzung auftreten, die ein zügiges Feedback durch den Product Owner verhindern würden. Stellt man beim Verfeinern fest, dass die Funktionseinheit doch nicht so umfangreich ist, wie befürchtet, hat das Team eine wichtige Erkenntnis gewonnen. Andererseits kann es auch sein, dass dabei festgestellt wird, dass die Funktionseinheit tatsächlich umfangreicher ist, als zunächst angenommen. Dann hilft die Verfeinerung dabei, die Umsetzungsphase flüssig zu durchlaufen. Das Material der Entwurfsphase ist viel flexibler formbar, als das Material der Implementationsphase. Code stellt hohe syntaktische Anforderungen, da er enorm viele Details enthält. Entwurfsdiagramme sind abstrakter und damit flexibler änderbar. Im Zweifel wird man lieber mehr Zeit mit dem Entwurf verbringen. Die Folge ist eine flüssige Umsetzungsphase.

Integration vs. Operation

Wenn Funktionseinheiten hierarchisch zerlegt werden, wie im vorhergehenden Abschnitt erläutert, entstehen zwei unterschiedliche Kategorien von Funktionseinheiten. Da sind solche, die verfeinert sind und solche, die nicht weiter zerlegt wurden. In der folgenden Abbildung sind f_1 und f_2 nicht weiter zerlegt, während f und f_3 zerlegt sind. Die Funktionseinheiten f und f_3 sind für die *Integration* zuständig, während f_1 , f_2 , f_{3a} und f_{3b} *Operationen* sind.

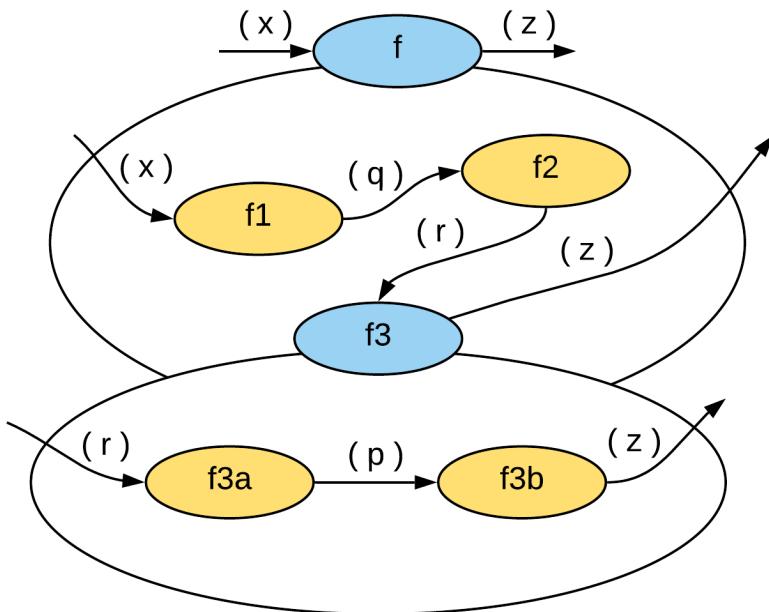


Abbildung 42: Hierarchische Zerlegung eines Entwurfs

Die Abbildung zeigt den Datenfluss, der durch die Funktionseinheiten realisiert wird. Eine andere Sicht auf die selben Funktionseinheiten zeigt das folgende Abhängigkeitsdiagramm.

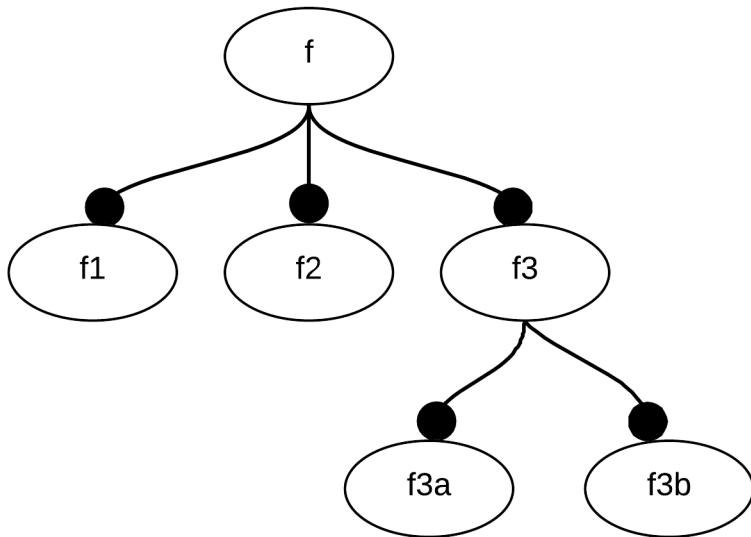


Abbildung 43: Abhängigkeitsdiagramm zum Datenflussdiagramm

Die Funktionseinheit f hängt ab von den Funktionseinheiten f_1 , f_2 und f_3 . Die Funktionseinheit f_3 hängt ab von den Funktionseinheiten f_{3a} und f_{3b} . Diese Abhängigkeiten ergeben sich aus der Tatsache, dass f dafür verantwortlich ist, den Datenfluss von f_1 , f_2 , und f_3 herzustellen. Ebenso ist f_3 dafür verantwortlich, den Datenfluss von f_{3a} und f_{3b} herzustellen. Damit f die beteiligten Funktionseinheiten so aufrufen kann, dass der gezeigte Datenfluss entsteht, muss sich f von ihnen abhängig machen. Das gilt analog für f_3 . Die nicht verfeinerten Funktionseinheiten f_1 , f_2 , f_{3a} und f_{3b} sind dagegen nicht von anderen Funktionseinheiten abhängig. Diese wichtige Erkenntnis führt zu den zwei zentralen Begriffen *Integration* und *Operation*. Eine Funktionseinheit, deren Aufgabe es ist, andere Funktionseinheiten zu integrieren, ist zuständig für *Integration*. Eine Funktionseinheit, die dagegen nicht integriert, sondern stattdessen Logik enthält, ist eine *Operation*.

Definition: Integration - Eine Funktionseinheit, die andere Funktionseinheiten integriert und keine Logik enthält. Integration stellt den Flow her, verbindet andere Funktionseinheiten.

Definition: Operation - Eine Funktionseinheit, die Logik enthält. Operationen sind frei von Abhängigkeiten zu anderen Funktionseinheiten.

Wir werden weiter unten noch auf zwei wichtige Prinzipien stoßen, die von jeder Funktionseinheit eingehalten werden müssen.

Vorgehensweise beim Entwurf

Der Entwurf einer Lösung ist ein kreativer Prozess. Es ist daher unrealistisch davon auszugehen, dass man den Entwurf einer Lösung einfach so hinzeichnet, quasi "aus dem Ärmel schüttelt". Es ist daher zu empfehlen, den Entwurf nicht mit dem Anspruch zu beginnen, dass gleich beim ersten Zeichnen alles perfekt ist. Stattdessen sollte man vor allem seiner Kreativität Raum geben und einfach anfangen. Durch das Aufzeichnen der ersten Datenflüsse vertieft sich das Verständnis für die Problemlösung. Es tauchen plötzlich neue Ideen auf. Auch Aspekte, an die man zuerst nicht gedacht hat, werden plötzlich sichtbar. Es ist daher hilfreich, die Diagramme so zu erstellen, dass man sie ohne großen Aufwand verändern kann. Am besten geeignet ist ein großes Whiteboard, verschiedenefarbige Stifte und ein Wischer. Alternativ kann man auch ein Flipchart verwenden.

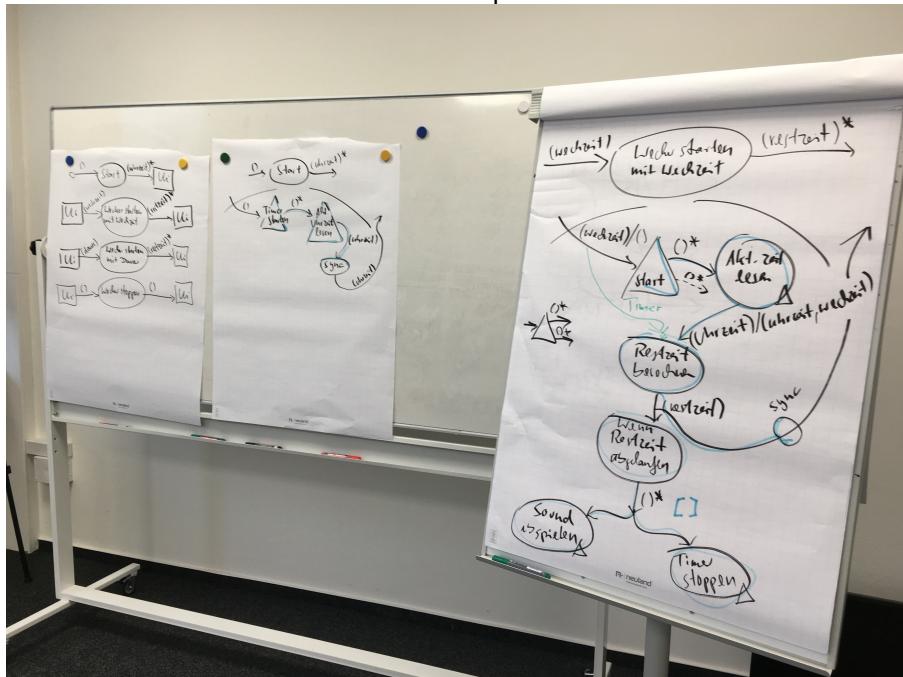


Abbildung 44: Ergebnisse am Flipchart

Durch das große Format wird es möglich, mit mehreren Entwicklern gemeinsam zu entwerfen. Kreative Prozesse profitieren davon, dass mehrere Personen beteiligt sind und ihre unterschiedlichen Ideen und Sichtweisen

einbringen. Entwerfen Sie alleine, sollten Sie unbedingt darauf achten, dass Sie Wischen können. Papier und Stifte sind wunderbar geeignet, solange das Resultat wieder ausgerichtet werden kann. Der klassische Bleistift und ein Radierer sind eine Möglichkeit. Wer es etwas moderner und insbesondere mehrfarbig mag, kann radierbare Tintenroller wie bspw. Pilot Frixion Stifte verwenden. Noch moderner geht es mit iPad und Pencil.

Beim Entwurf kann man grundsätzlich zwei verschiedene Richtungen unterscheiden: *top-down* geht von oben nach unten vor, vom Allgemeinen zu den Details, während *bottom-up* von unten nach oben vorgeht. Es gibt hier keine bevorzugte Richtung. Wenn das Problem bereits gut durchdrungen ist, kann top-down gut funktionieren. Merkt man, dass es nur zäh voran geht, ist es hilfreich, in die bottom-up Richtung zu wechseln. Man zeichnet einige der Details in das Diagramm, um dann später zu erkennen, wie diese Details zu etwas Größerem zusammengefasst werden können. Flow Design ist eine Entwurfsmethode, die eher bottom-up orientiert ist. Es werden erst die Datenflüsse, bestehend aus mehreren Funktionseinheiten, erstellt. Erst kurz vor der Umsetzung werden dann Klassen gebildet. Andere Entwurfsmethoden wie etwa *Object Oriented Analysis and Design* (OOAD) gehen eher top-down vor. Sie beginnen mit Klassen und ordnen die benötigte Funktionalität den Klassen zu.

Die bottom-up Vorgehensweise führt zu einem flüssigen Prozess und einer guten Klassenstruktur. Gut meint hier, eine dem Problem angemessene Klassenstruktur. Die Klassen und ihre Beziehungen können sich erst ergeben, wenn ihr Inhalt klar ist. Insofern unterliegt ein top-down Vorgehen vielen Vermutungen darüber, welche Methoden möglicherweise benötigt werden. Bottom-up sind die Methoden bereits klar und es kann leichter herausgefunden werden, wie die beteiligten Methoden zu einer angemessenen Klassenstruktur zusammengefügt werden.

Dazu ein Vergleich: vor Ihnen liegt ein Berg von Legosteinen. Sie können nun, ohne sich die Legosteine anzuschauen, erst einige Schachteln beschriften und dann beginnen, die Legosteine in die Schachteln zu sortieren. Die Wahrscheinlichkeit ist sehr hoch, dass sie dabei feststellen, dass Ihnen Schachteln fehlen, manche Schachteln zu voll sind, manche vielleicht sogar leer bleiben. Die Wahrscheinlichkeit, dass die Schachteln eine angemessene Ordnung für den Berg von Legosteinen darstellen steigt, wenn Sie vorher bereits wissen, welche Legosteine sich in dem Berg befinden. Leichter wird es sein, die Ordnung herzustellen, wenn Sie zunächst die Legosteine aus dem Berg entnehmen und nach Kategorien ordnen. Gleiche oder ähnliche legen Sie zusammen, unterschiedliche legen Sie weiter auseinander. So entdecken Sie die Struktur, die Ordnung, und sind dann sehr gut in der Lage, eine angemessene Anzahl Schachteln für die vorgefundenen Steine zu beschriften.

Bei der Softwareentwicklung verhält es sich ähnlich. Erst die Klassen zu finden und dann zu beginnen, den Klassen Methoden zuzuordnen, ist eine

große Herausforderung. Viel einfacher geht es, wenn man erst über die Methoden nachdenkt und diese im Anschluss zu Klassen zusammenfasst. Bei der hierarchischen Zerlegung eines Flow Design Diagramms stellt sich immer wieder die Frage, ob man eine Funktionseinheit zerlegt bzw. ob man mehrere Funktionseinheiten zu einer übergeordneten Funktionseinheit zusammenfassen soll. Manchmal ist auch zu überlegen, ob man eine Hierarchieebene wieder entfernt und die Details direkt in das übergeordnete Diagramm einfügt.

Dies ist vergleichbar mit den Quellcode Refaktorisierungen *Extract Method* und *Inline Method*. Manchmal fasst man mehrere Codezeilen zu einer Methode zusammen (*Extract Method*) und zieht diese aus einer größeren Methode heraus. An anderer Stelle ist es jedoch vielleicht hilfreich, den Methodenaufruf wieder durch den Inhalt der Methode zu ersetzen (*Inline Method*). Ebenso verhält es sich beim Entwurf. Manchmal fasst man mehrere Funktionseinheiten zusammen, manchmal löst man eine solche Zusammenfassung wieder auf. Am Ende geht es darum, dass ein Flow Design Diagramm möglichst leicht verständlich ist. Wenn es aus wenigen Elementen besteht, führt eine zusätzliche Hierarchie eher nicht zu besserer Verständlichkeit. Enthält ein Datenflussdiagramm dagegen viele Elemente und sind die Datenflüsse kompliziert, hilft die Einführung einer Hierarchie. In jedem Fall kommen dabei die Methoden zuerst an die Reihe. Klassen entstehen im Anschluss aus den vorgefundenen Methoden.

Verzweigungen: Split und Entscheidung

Ein Datenfluss kann sich verzweigen. Dabei müssen zwei unterschiedliche Formen von Verzweigungen unterschieden werden. Im einen Fall fließen die selben Daten an unterschiedliche Funktionseinheiten. Es wird keine Entscheidung getroffen, sondern der Datenfluss verzweigt sich und die selben Daten fließen von einer Quelle zu unterschiedlichen Senken. Im anderen Fall wird durch eine Funktionseinheit eine Entscheidung getroffen. Von dieser Entscheidung hängt es dann ab, wohin die Daten weiter fließen. *Split* und *Entscheidung* sind zwei unterschiedliche Formen von Datenflüssen, in denen es zu einer Verzweigung des Flows kommt.

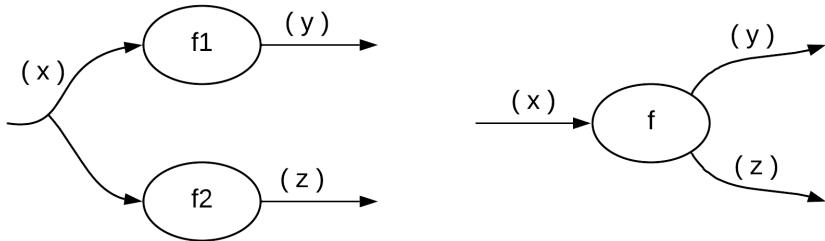


Abbildung 45: Die beiden Verzweigungen: Split und Entscheidung

Die Abbildung zeigt die zwei unterschiedlichen Arten von Verzweigungen, also Flows, die sich aufteilen. Im ersten Fall fließt das x sowohl zur Funktionseinheit f_1 als auch zu f_2 . Es wird keinerlei Entscheidung getroffen, sondern die Daten fließen ohne weitere Bedingung zu beiden Funktionseinheiten. Diese Form des Flows bezeichnet man als *Split*. Im rechten Teil der Abbildung ist ein Flow gezeigt, in der das x zunächst nur zur Funktionseinheit f fließt. Diese entscheidet dann darüber, ob im weiteren Verlauf ein y oder ein z fließt. Eine Funktionseinheit, welche eine Entscheidung trifft, kann im Unterschied zum Split zwei oder mehr völlig unterschiedliche Datenflüsse zur Folge haben. Im Beispiel aus der Abbildung kann ein y oder ein z fließen. Denkbar wäre allerdings auch, dass zuerst ein y fließt und im Anschluss ein z . Natürlich wäre es auch andersherum möglich, erst ein z dann ein y . Und um auch den letzten Fall abzudecken: es wäre auch möglich, dass weder ein y noch ein z fließt. Ein Datenfluss, der sich aufgrund einer Entscheidung verzweigt, bietet deutlich mehr Möglichkeiten, als ein einfacher Split.

Im Beispiel *CSV Viewer* aus der Einleitung ist ein Split, also eine einfache Verzweigung enthalten. Dort fließen die Kommandozeilenargumente sowohl zur Funktionseinheit *GetFilename* als auch zur Funktionseinheit *GetPageLength*.

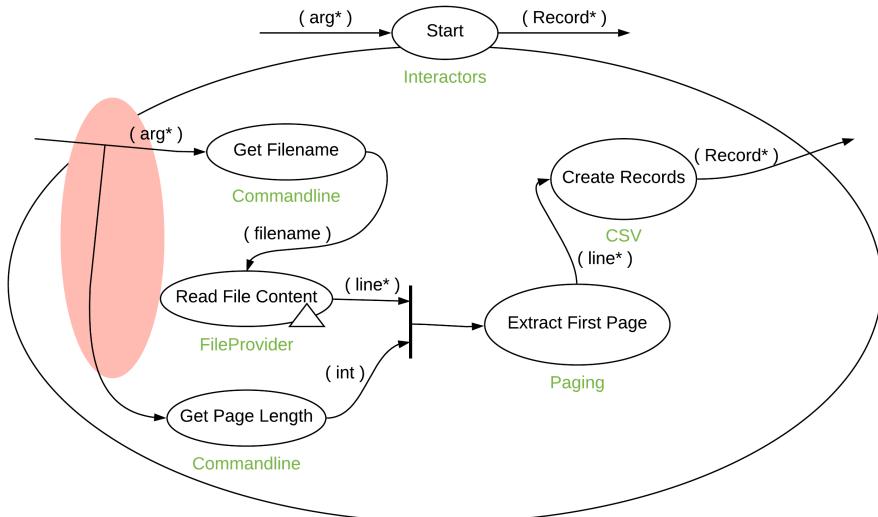


Abbildung 46: Ausschnitt aus CSV Viewer / Start

Ein Flow Design Diagramm macht keine eindeutige Aussage darüber, in welcher Reihenfolge die Datenflüsse bei einer Verzweigung durchlaufen werden. Ob also im Beispiel zuerst *GetFilename* oder zuerst *GetPageLength* ausgeführt wird, sagt das Diagramm nicht aus. Üblicherweise wird die Implementation von oben nach unten und von links nach rechts durchgeführt, so dass im Beispiel erst *GetFilename* und danach *GetPageLength* ausgeführt würden. In den allermeisten Fällen spielt die Reihenfolge semantisch auch gar keine Rolle, so dass es dem Entwickler freistehet, wie er den Datenfluss in Code übersetzt. Sollte die Reihenfolge relevant sein, kann man das entweder durch eine Notiz erläutern oder die Funktionseinheiten im Datenfluss hintereinander stellen. Beispiele dazu folgen im Verlauf des Buches.

Die Übersetzung des Entwurfs in C# sieht wie folgt aus:

```

public IEnumerable<Record> Start(string[] args) {
    var filename = commandLine.GetFilename(args);
    var pageLength = commandLine.GetPageLength(args);
    // ...
}

```

Die Kommandozeilenargumente *args* fließen in die beiden Methoden

GetFilename und *GetPageLength*, in dem diese beiden Methoden nacheinander mit dem selben Parameter aufgerufen werden.

Die zweite Form der Verzweigung, die Entscheidung, verhält sich anders. Hier trifft eine Funktionseinheit die Entscheidung darüber, auf welchen ausgehenden Datenflüssen Daten fließen sollen. In der folgenden Abbildung ist aufgrund der Namensgebung ersichtlich, dass entweder der eine oder der andere Datenfluss durchlaufen wird.

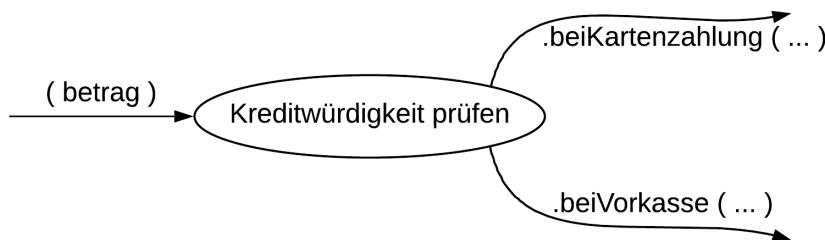


Abbildung 47: "Kreditwürdigkeit prüfen" als Funktionseinheit mit zwei alternativen Ausgängen

Mit ein wenig Kenntnis über die Domäne kann man diesem Beispiel entnehmen, dass entweder die Kreditkartenzahlung oder die Bezahlung per Vorkasse durchgeführt wird. Niemand käme aufgrund der Kenntnisse über die Domäne auf die Idee, dass beide Datenflüsse durchlaufen werden. Beim folgenden Beispiel sieht das anders aus.

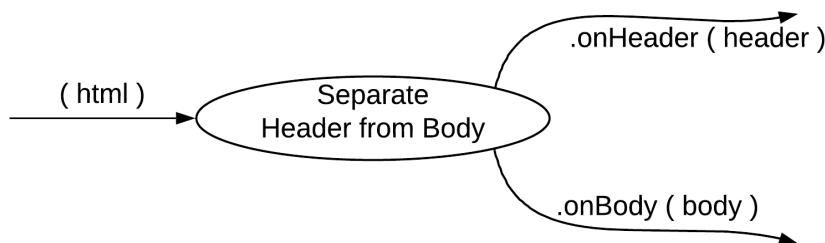


Abbildung 48: HTML Header und Body trennen

Die Funktionseinheit trennt den HTML Header vom Body. Ergebnis sind zwei Datenflüsse, die beide durchlaufen werden. Hier ist mit ein wenig Kenntnis über die Domäne *HTML* erneut klar, dass es kein entweder-oder ist, sondern ein sowohl-als-auch.

Es braucht also keine spezielle Syntax, um deutlich zu machen, ob mehrere ausgehende Datenflüsse alternativ oder kumulativ durchlaufen werden. Der

Kontext, insbesondere die Bezeichnung der Funktionseinheiten und der Datenflüsse, macht dies deutlich.

Hat eine Funktionseinheit mehrere ausgehende Datenflüsse, stellt sich die spannende Frage, auf welche Weise dies implementiert werden kann. Bevor wir dazu kommen, sollen Datenflüsse zunächst wieder zusammengeführt werden.

Und wieder zusammenführen: Join

Häufig müssen Daten, die auf getrennten Datenflüssen hergestellt wurden, für die Weiterverarbeitung wieder zusammengeführt werden. Dies geschieht über einen *Join*. Die folgende Abbildung zeigt erneut einen Ausschnitt aus dem *CSV Viewer* Beispiel.

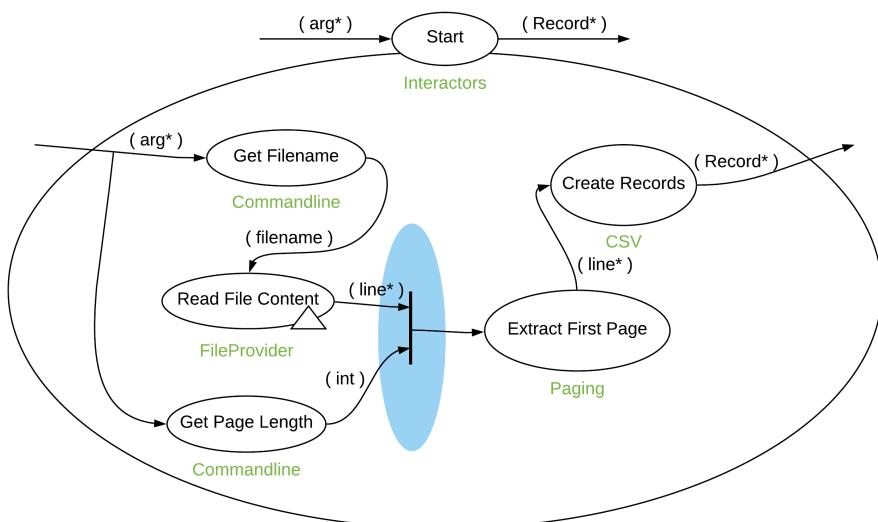


Abbildung 49: Join im Entwurf von CSV Viewer / Start

Die Funktionseinheit *ExtractFirstPage* benötigt sowohl die Zeilen der Datei als auch die Seitenlänge aus den Kommandozeilenparametern. Nur mit beiden Eingabedaten kann sie ihre Aufgabe durchführen und die Zeilen der ersten anzuzeigenden Seite aus den gesamten Zeilen extrahieren. Der *Join* führt daher die beiden Datenflüsse zusammen und sorgt dafür, dass *ExtractFirstPage* mit zwei Eingabewerten aufgerufen wird. Die Übersetzung in C# sieht wie folgt aus:

```

public IEnumerable<Record> Start(string[] args) {
    // ...
    var pageLength = commandLine.GetPageLength(args);
    var lines = fileProvider.ReadFileContent(filename);
    var firstPage = paging.ExtractFirstPage(lines, pageLength);
    // ...
}

```

Die von *ExtractFirstPage* benötigten Eingabedaten werden von den Methoden *GetPageLength* und *ReadFileContent* bereitgestellt. Um nun *ExtractFirstPage* mit den beiden Werten aufrufen zu können, werden die Ausgabedaten der beiden Methoden in den Variablen *pageLength* und *lines* abgelegt. Die beiden Variablen dienen dann als Eingangsparameter für *ExtractFirstPage*. Die Übersetzung eines Join ist also im Code nicht explizit als eigene Zeile o.ä. zu erkennen. Es werden lediglich mehrere Parameter an die Funktionseinheit übergeben, die am Ausgang des Joins steht.

Beim Join gilt als Konvention, dass die Daten am Ausgang des Joins einem Tupel der Eingangsdaten entsprechen. Fließen ein *x* und ein *y* in den Join, steht am Ausgang des Joins per Konvention das Tupel (x, y) zur Verfügung. Natürlich können auch mehr als zwei Eingänge zu einem Tupel zusammengefasst werden.

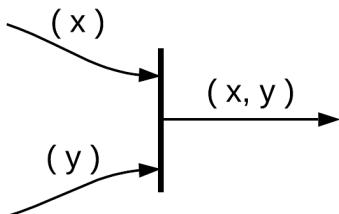


Abbildung 50: Ausgang des Join ist ein Tupel der Eingangsdaten

Ein Join kann dazu verwendet werden, die Ausführung von Funktionseinheiten in eine erforderliche Reihenfolge zu bringen. Funktionseinheiten, bei denen die Reihenfolge der Ausführung keine Rolle spielt, können in parallele Datenflüsse gezeichnet werden. Um dann die weitere Ausführung zu synchronisieren und eine Reihenfolgeabhängigkeit auszudrücken, werden die parallelen Datenflüsse über einen Join wieder zusammengeführt. Wichtig hervorzuheben ist, dass die Implementation eines Flow Design Diagramms *synchron sequentiell* erfolgt. Das bedeutet, zu einem gegebenen Zeitpunkt befindet sich immer genau eine Funktionseinheit in Aus-

führung. Es findet keine asynchrone Ausführung statt. Die Abarbeitung der einzelnen Funktionseinheiten erfolgt sequentiell nacheinander. Es geht hier also noch nicht darum, durch parallele Datenflüsse Nebenläufigkeit auszudrücken. Gleichwohl kann durch parallele Datenflüsse angezeigt werden, dass eine Parallelisierung möglich wäre. Weitere Details zur asynchronen Ausführung folgen in einem späteren Kapitel.

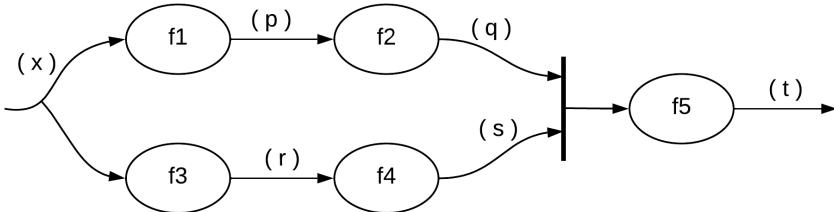


Abbildung 51: Join zum Synchronisieren paralleler Datenflüsse

Bei einem Join stellt sich manchmal die Frage, wie sich dieser bei mehrfachem Durchlaufen des Datenflusses verhält. Verfügt der Join bspw. über zwei Eingänge, müssen initial an beiden Eingängen Daten ankommen, bevor es am Ausgang des Joins weitergehen kann. Wenn nun aber einer der beiden eingehenden Datenflüssen erneut Daten liefert, ist zu klären, ob dann auf dem anderen Eingang die Daten verwendet werden, die zuvor dort anstanden, oder ob auch auf dem anderen Eingang auf Daten gewartet wird. Diese Unterscheidung ist bei der bislang gezeigten Implementation eines Joins nicht relevant, weil immer beide Eingänge Daten liefern. Wir werden jedoch später sehen, dass ein Join auch als Klasse mit Events und Methoden realisiert werden kann. In diesem Fall ist zu unterscheiden, ob der Join sich die Daten auf einem Eingang merkt und wiederverwendet, oder ob es zu einem Reset kommt, nachdem die Daten am Ausgang abgeliefert wurden.

Entscheidungen implementieren

Kommen wir nun zur Frage zurück, wie eine Entscheidung, also eine Verzweigung, bei der mehrere Datenflüsse aus einer Funktionseinheit herausfließen, implementiert werden kann. Die Übersetzung in Code hängt davon ab, welche syntaktischen Möglichkeiten die verwendete Programmiersprache bietet.

Soll eine Funktionseinheit mehrere ausgehende Datenflüsse liefern, kann dies nicht über den Ergebnistyp der Methode realisiert werden. Der Rückgabewert einer Methode liefert immer Daten und kann ferner nicht Daten unterschiedlichen Typs liefern.

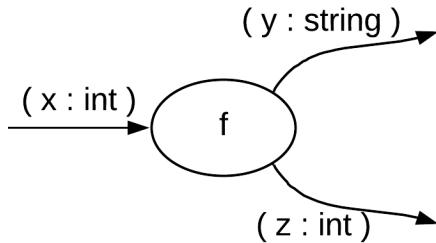


Abbildung 52: Eine Entscheidung mit zwei Ausgängen

Hier entscheidet die Funktionseinheit f , ob sie ein y vom Typ $string$ liefert oder ein z vom Typ int . Eine Implementation in C# sieht folgendermaßen aus:

```

public void f(int x, Action<string> onY, Action<int> onZ) {
    if (x < 0) {
        onY("negativ");
    }
    else {
        onZ(x);
    }
}

```

Hier werden *Callbacks* für die beiden Ausgänge verwendet. Der .NET Datentyp $Action<T>$ stellt einen Zeiger auf eine Methode dar. $Action<string>$ bedeutet demnach, dass ein Zeiger auf eine Methode übergeben werden muss, die einen $string$ als Eingabeparameter hat. Der Aufruf der Methode f erfolgt dann folgendermaßen:

```
f(42,  
    onY : y => Console.WriteLine(y),  
    onZ : z => Console.WriteLine(z)  
);
```

Der erste Parameter der Methode *f* entspricht der Eingabe *x* aus dem Flow. Wenn *f* die Entscheidung trifft, mit einem *y* vom Typ *string* fortzusetzen, wird die erste *Lambda Expression* ausgeführt. Kommt es zur Entscheidung, mit einem *z* vom Typ *int* fortzusetzen, wird die zweite Lambda Expression ausgeführt. Eine Lambda Expression ist nichts anderes, als eine Methode, deren Inhalt direkt inline an Ort und Stelle hingeschrieben ist. Alternativ kann der Inhalt der Lambda Expression in eine Methode ausgelagert werden:

```
public void Verwendung_von_f() {  
    f(42,  
        onY : Weiter_mit_Y,  
        onZ : z => Console.WriteLine(z)  
    );  
}  
  
private static void Weiter_mit_Y(string  
y) {  
    Console.WriteLine(y);  
}
```

Die Implementation einer Verzweigung durch Callbacks darf nicht verwechselt werden mit *out* Parametern. Man könnte versuchen, den Sachverhalt wie folgt zu implementieren:

```
public void g(int x, out string y, out int z) {  
    if (x < 0) {  
        y = "negativ";  
    }  
    else {  
        z = x;  
    }  
    // syntaktisch falsch, da beim Verlassen beide  
    // out Parameter zugewiesen sein müssen  
}
```

Das Problem hierbei ist zunächst ein syntaktisches. Beiden *out* Parametern der Methode müssen vor dem Verlassen der Methode Werte zugewiesen werden. Bei Referenztypen wie *string* könnten wir den Parameter mit *null* belegen. Doch bei Werttypen wie *int* ist das nicht möglich. Davon einmal abgesehen wäre es für den Aufrufer nicht erkennbar, in welchem *out* Parameter ein Wert vorliegt, sprich ob es mit einem *y* oder einem *z* weiter gehen soll. *out* Parameter können verwendet werden, wenn eine Funktionseinheit mehr als einen Wert, also ein Tupel, als Ausgabe liefert. Für alternative Ausgaben sind *out* Parameter jedoch nicht geeignet.
Greifen wir das Beispiel zur Kreditwürdigkeitsprüfung aus Abbildung oben noch einmal auf. Eine Implementation mit Callbacks sieht wie folgt aus:

```
public void Kreditwürdigkeit_prüfen(
    double betrag,
    Action beiKartenzahlung,
    Action beiVorkasse) {
    if (betrag <= 1.000) {
        beiKartenzahlung();
    }
    else {
        beiVorkasse();
    }
}
```

Die Methode wird beim Aufrufer wieder mit Lambda Expressions verwendet:

```
public void Verwendung() {
    var betrag = 2.39; // ... irgendein Betrag
    Kreditwürdigkeit_prüfen(betrag,
        beiKartenzahlung: () => {
            // ... Code, der bei Kartenzahlung
            // ausgeführt wird
        },
        beiVorkasse: () => {
            // ... Code, der bei Vorkasse
            // ausgeführt wird
        });
}
```

Entwickler, die mit Lambda Expressions und Callbacks nicht vertraut sind, stellen immer wieder die Frage, aus welchem Grund wir diese Form der Implementation empfehlen. Eine Alternative wäre doch folgende:

```
public bool Kreditwürdigkeit_prüfen(double betrag) {
    if (betrag <= 1.000) {
        return true;
    }
    else {
        return false;
    }
}
```

Die Entscheidungsmethode liefert nun einen *bool* zurück. Damit kann der Aufrufer entscheiden, wie es weitergeht:

```
var betrag = 2.39; // ... irgendein Betrag
if (Kreditwürdigkeit_prüfen(betrag)) {
    // ... Code, der bei Kartenzahlung
    // ausgeführt wird
}
else {
    // ... Code, der bei Vorkasse
    // ausgeführt wird
};
```

Offensichtlich ist zunächst, dass jetzt der Methodenname nicht mehr gut passt. Naheliegender wäre es nun, die Methode *IstKreditwürdig* zu nennen:

```
var betrag = 2.39; // ... irgendein Betrag
if (IstKreditwürdig(betrag)) {
    // ... Code, der bei Kartenzahlung
    // ausgeführt wird
}
else {
    // ... Code, der bei Vorkasse
    // ausgeführt wird
};
```

Es sollte dann auch im Entwurf *Ist Kreditwürdig* als Bezeichnung der Funktionseinheit notiert sein.

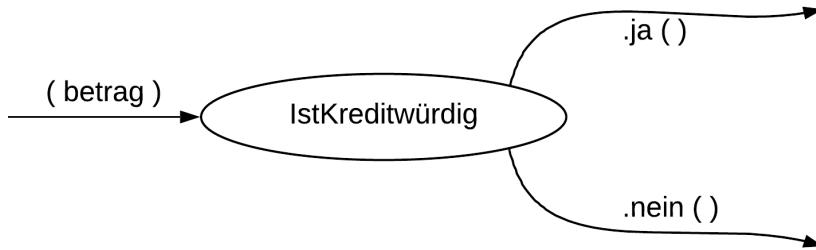


Abbildung 53: Kreditwürdigkeitsprüfung umbenannt

Der eigentliche Grund dafür, dass wir eine Implementation mit Callbacks empfehlen, ist jedoch folgender: der Ausdruck `if (IstKreditwürdig(betrag))` wird bei späteren Ergänzungen des Code häufig so modifiziert, dass plötzlich die Bedingung nicht mehr vollständig von der Methode `IstKreditwürdig` ermittelt wird. Es könnte bspw. zu folgender Änderung kommen:

```

if (IstKreditwürdig(betrag) || kunde.Status == Status.Gold) {
    // ... Code, der bei Kartenzahlung
    // ausgeführt wird
}
else {
    // ... Code, der bei Vorkasse
    // ausgeführt wird
};

```

Es wird also flugs die Bedingung

`|| kunde.Status == Status.Gold`

ergänzt, ohne dass dies im Entwurf dargestellt wird. Schlimmer noch, dieser Code kann nicht mehr in Form eines Flow Design Diagramms dargestellt werden. Aus diesem Grund raten wir ganz dringend dazu, mit Callbacks zu arbeiten, wo dies die Programmiersprache in einer syntaktisch gut lesbaren Weise ermöglicht. Ist man bspw. auf ältere Java Versionen angewiesen, die noch keine Unterstützung für Lambda Expression bieten, ist es schwieriger zu entscheiden, ob man die dann etwas aufwändigeren Syntax mit anonymen Klassen verwendet, oder doch einen booleschen Rückgabewert und eine `if` Anweisung einsetzt. JavaScript/NodeJS Entwickler dagegen sind Callbacks gewohnt und werden eher keine Herausforderung darin sehen, Entscheidungen auf diese Weise zu implementieren.

Wesentlich bei der Diskussion ist noch ein weiterer Aspekt, nämlich die Testbarkeit. Steckt die Entscheidung vollständig in einer Funktion, kann

diese Funktion mit automatisierten Tests isoliert getestet werden. Sobald jedoch der Ausdruck der *if* Anweisung nicht mehr nur aus einem Funktionsaufruf besteht, ist dieser isolierte Test nicht mehr möglich. Es muss dann die umschließende Methode im Test aufgerufen werden, um den Ausdruck der *if* Anweisung zu durchlaufen. Insofern ist es sehr wichtig darauf zu achten, dass eine Entscheidung vollständig durch eine Funktion getroffen wird. Nur so ist es möglich, die Entscheidung selbst isoliert zu testen.

Isoliert testbar:

```
if (IstKreditwürdig(betrag)) {  
    // ...  
}
```

In dieser Form kann die Bedingung der *if* Anweisung vollständig isoliert getestet werden. Dazu wird die Funktion *IstKreditwürdig* im Test mit unterschiedlichen Parametern aufgerufen und das Resultat mit einem *Assert* überprüft.

Nicht isoliert testbar:

```
if (IstKreditwürdig(betrag) || kunde.Status == Status.Gold) {  
    // ...  
}
```

Hier kann die Bedingung der *if* Anweisung nicht mehr isoliert getestet werden. Der Ausdruck kann im Test nicht isoliert aufgerufen werden. Die Lösung besteht darin, den Ausdruck mittels *Extract Method* Refactoring in eine Funktion auszulagern. Dann kann diese im Test isoliert aufgerufen werden.

```
if (IstKreditwürdig(betrag, kunde)) {
    // ...
}

private bool IstKreditwürdig(double betrag, Kunde kunde) {
    return IstKreditwürdig(betrag) || kunde.Status == Status.Gold;
}
```

Der Übergang vom Entwurf zur Umsetzung

Prüfungen vor der Umsetzung

Bevor ein Entwurf umgesetzt wird, müssen mindestens zwei Dinge überprüft werden:

- Ist jede Funktionseinheit nur für genau einen Aspekt zuständig?
- Kann jede Funktionseinheit in maximal vier Stunden implementiert werden?

Verstößt eine der Funktionseinheiten gegen eine dieser Eigenschaften, muss der Entwurf weiter verfeinert werden. Sind die beiden Punkte eingehalten, steht einer Umsetzung nichts mehr im Wege.

Dass jede Funktionseinheit nur für genau einen Aspekt zuständig sein darf, ist eines der wichtigsten Prinzipien der Informatik. Das Prinzip stellt das Fundament für die Wandelbarkeit von Software dar. Der Grund liegt auf der Hand: ändern sich die Anforderungen in einem Aspekt der Software, soll dies nur lokal begrenzte Änderungen an Funktionseinheiten zur Folge haben. Sobald eine Funktionseinheit für mehr als einen Aspekt zuständig ist, ist sie von Änderungen aus unterschiedlichen Gründen betroffen. Damit steigt die Gefahr, dass andere, ebenfalls enthaltene Aspekte, in Mitleidenschaft gezogen werden. Das übergeordnete Prinzip lautet "Aspekte sind zu trennen". Es ist auch unter der Bezeichnung *Separation of Concerns* bzw. *Single Responsibility Principle* bekannt.

Die Empfehlung, Funktionseinheiten so klein zu zerlegen, dass sie in maximal 4 Stunden implementiert werden können, ergibt sich aus der Dauer einer Iteration. Diese sollte maximal ein bis zwei Tage dauern, um auf diese Weise kurzfristiges Feedback vom Product Owner zu erhalten. Vermutet einer der Entwickler, dass die Umsetzung einer der Funktionseinheiten möglicherweise länger als vier Stunden dauern könnte, steigt die Wahrscheinlichkeit, dass die Umsetzung des gesamten Inkrements länger dauert als ein bis zwei Tage. Aus diesem Grund sollte eine Funktionseinheit, für die der Verdacht besteht, sie könnte zu groß sein, im Zweifel weiter verfeinert werden, um so die Sicherheit zu erhöhen, das gesamte anstehende Inkrement in maximal zwei Tagen fertig zu stellen. Ggf. wird ein Feature weggelassen. Im Vordergrund steht der kontinuierliche Flow von Inkrementen an den Product Owner.

Methode vs. Klasse

Nachdem für jede Funktionseinheit geklärt ist, dass sie ausreichend klein und nur für einen Aspekt zuständig ist, geht es um die Frage, in welches Form die Funktionseinheiten übersetzt werden sollen.

Wenn wir von objektorientierten Programmiersprachen ausgehen, gibt es zwei Möglichkeiten, eine Funktionseinheit in Code zu übersetzen. Eine Funktionseinheit kann entweder in eine *Methode* oder eine *Klasse* übersetzt werden.

Eine Funktionseinheit wird als Methode implementiert, in dem alle Inputs der Funktionseinheit zu Parametern der Methode werden. Alle Outputs werden zu Rückgabewerten der Methode.

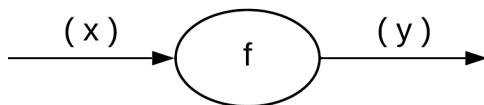


Abbildung 54: Einfache Funktionseinheit

Die oben abgebildete Funktionseinheit kann auf folgende Weise als Methode realisiert werden:

```
public int f(int x) {  
    var y = x + 42; // Calculate value of y  
    return y;  
}
```

Die Übersetzung in eine Methode gelingt solange, wie die Funktionseinheit lediglich einen eingehenden Datenfluss hat. Sollen mehrere eingehende Datenflüsse realisiert werden, muss die Umsetzung als Klasse erfolgen. Mehrere ausgehende Datenflüsse können dagegen auch als Methode realisiert werden, wie wir weiter unten sehen werden.

Funktionseinheiten mit mehreren eingehenden Datenflüssen müssen als Klasse realisiert werden. Jeder eingehende Datenfluss wird dann zu einer Methode innerhalb der Klasse. Ausgehende Datenflüsse werden als Callbacks oder als Events realisiert.

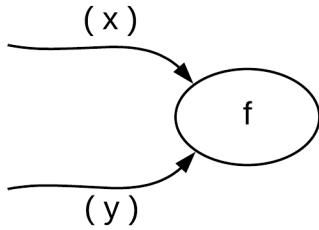


Abbildung 55: Funktionseinheit mit zwei eingehenden Datenflüssen

```

public class F
{
    public event Action<int> Result;

    public void OnX(int x) {
        var z = x + 42; // Calculate value of z
        Result(z);
    }

    public void OnY(int y) {
        var z = y + 42; // Calculate value of z
        Result(z);
    }
}

```

Hier sind die beiden eingehenden Datenflüsse jeweils als Methode realisiert. Die eingehenden Daten werden jeweils zu Parametern der Methoden. Der ausgehende Datenfluss ist als Event realisiert.

Die Übersetzung einer Funktionseinheit in Form einer Klasse sollte nur gewählt werden, wenn dies aufgrund der Struktur der Funktionseinheit erforderlich ist. Natürlich muss auch die Methode in eine Klasse eingefasst werden und kann nicht für sich alleine stehen. Die Umsetzung eines Entwurfs sollte so leichtgewichtig wie möglich sein, um einerseits zügig voran zu kommen und andererseits die Wandelbarkeit zu unterstützen.

Als Klasse werden typischerweise die UIs realisiert. UIs zeichnen sich dadurch aus, dass sie die Quelle mehrerer Datenflüsse sind. Jede Interaktion in einem Dialog führt zu einem Datenfluss, der als Ausgang einer UI entworfen und implementiert wird. Folglich verfügen die UI Klassen über

mehrere Events, an denen ein Datenfluss beginnt, wenn der Benutzer mit dem Dialog interagiert. Ähnliches gilt auf der Eingangsseite eines Dialogs. Auch dort existieren meist mehrere eingehende Datenflüsse, die jeweils als Methode der UI Klasse realisiert werden.

Stehen in der verwendeten Programmiersprache syntaktisch keine Events zur Verfügung, muss mit Funktionszeigern oder Observern gearbeitet werden.

Nachrichten

Syntax der Datenflüsse

Ein Flow Design Diagramm besteht aus Funktionseinheiten und Datenflüssen. Die Datenflüsse drücken aus, welche Daten in eine Funktionseinheit hinein fließen bzw. welche Daten aus ihr heraus fließen. Der Pfeil gibt die Quelle und das Ziel der Daten an. Am Pfeil werden die Daten näher benannt, die fließen. Die Daten werden in Klammern geschrieben. Der Grund wird ersichtlich, wenn es um den Stern "*" in der Notation geht. Wir müssen die beiden folgenden Fälle unterscheiden können:

(x^*) vs. (x) *

Im einen Fall gehört der Stern zum x und bedeutet, eine Aufzählung bestehend aus vielen x . Im anderen Fall gehört er zu den Klammern und bedeutet, dass mehrfach Daten fließen. Salopp formuliert: im ersten Fall fließen einmalig viele x , im anderen Fall fließen die Daten mehrfach, jedesmal aber nur ein einzelnes x . Steht der Stern außerhalb, sprechen wir von *Streams*. Streams werden in einem späteren Kapitel näher beschrieben. Schauen wir uns zunächst genauer an, wie die Daten eines Datenflusses spezifiziert werden können. Im einfachsten Fall fließt auf einem Datenfluss ein Wert in einem einfachen Datentyp wie *string*, *int*, *bool*, etc. Wir notieren dann den Typ am Datenfluss in Klammern.



Abbildung 56: Einfacher Datentyp

In den meisten Fällen ist die Bedeutung der Daten aus dem Zusammenhang erkennbar. Eine Funktionseinheit mit dem Namen *GetFilename*, die einen *string* als Ausgabe produziert, wird wohl einen Dateinamen liefern. Zur Verdeutlichung kann anstelle des Typs ein Name angegeben werden. Per Konvention werden solche Namen mit kleinem Anfangsbuchstaben notiert. Diese Konvention stimmt mit der Konvention von C# überein, bei der die Parameternamen einer Methode klein geschrieben werden.

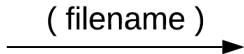


Abbildung 57: Es fließt ein *filename*

In dieser Form ist der Inhalt bzw. die Bedeutung der Daten durch den Namen näher beschrieben und der Typ leitet sich aus dem Kontext ab. Bei *filename* ist es naheliegend, dass der Typ ein *string* ist. Soll auch das näher spezifiziert werden, um für noch mehr Klarheit zu sorgen, können Name und Typ gemeinsam angegeben werden. Der Doppelpunkt trennt den Namen vom Typ.

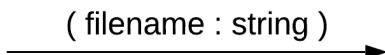


Abbildung 58: filename : string

Aufzählungen

Durch einen Stern “*” wird angezeigt, dass es sich um eine Aufzählung handelt. Die Notation *string** bedeutet also, eine Aufzählung von *string* Elementen. Hier wird durch den Stern bewusst eine abstrakte Notation verwendet, anstatt bereits im Entwurf festzulegen, welcher konkrete Typ wie bspw. Array oder Liste zu verwenden ist. Ferner sollte der Entwurf unabhängig sein von einer konkreten Programmiersprache. Daher wird erst beim Übergang vom Entwurf zur Implementation festgelegt, welcher konkrete Datentyp verwendet werden soll. Es ist wichtig, den Entwurf abstrakter zu halten als die Implementation. Auf diese Weise kann schneller vorangeschritten werden, weil nicht so viele Details auftreten. Die Syntax des Entwurfs ist abstrakter als die Syntax der Implementation. Auf diese Weise ist ein flexiblerer Umgang mit dem Entwurf möglich. Gerade weil noch nicht alle Details festgelegt sind, kann der Entwurf leicht umgeformt werden.

Datentypen beschreiben

Die Konvention, einen Namen mit kleinem Anfangsbuchstaben zu beginnen, ist wichtig, um Namen von eigenen Datentypen unterscheiden zu können. Denn nicht immer genügt es, einen einfachen Datentyp zu verwenden. Es wäre eine starke Einschränkung, wenn die Funktionseinheiten nur über einfache Datentypen wie *int*, *string*, *bool* etc. kommunizieren dürften. Die

Einführung von Datentypen, die speziell für einen kleinen Ausschnitt des Softwaresystems konzipiert wurden, sorgt für bessere Verständlichkeit. Beginnt die Bezeichnung der Daten an einem Datenfluss mit großem Anfangsbuchstaben, weist dies darauf hin, dass es sich um einen Datentyp handelt, der üblicherweise zur Lösung gehört. Der Typ ist also nicht zwingend ein vorhandener Datentyp aus einer verwendeten Bibliothek. Im Beispiel *CSV Viewer* wurde von dieser Möglichkeit bereits Gebrauch gemacht. Dort erzeugte die Funktionseinheit *CreateRecords* eine Aufzählung von *Record* Elementen.



Abbildung 59: CreateRecords aus dem CSV Viewer.

Der Datentyp muss im Entwurf näher erläutert werden. Für den Datentyp *Record* sah dies wie folgt aus:

Record

Values : string*

Diese tabellarische Notation ist so zu lesen, dass der Datentyp *Record* eine Eigenschaft mit dem Namen *Values* besitzt, die vom Typ *string** ist. Der Stern weist wiederum auf eine Aufzählung hin. In der konkreten Umsetzung in eine Programmiersprache muss dann vom Entwickler entschieden werden, ob er ein Array, eine Liste oder einen anderen Aufzählungstyp verwendet.

Die tabellarische Darstellung kann auch geschachtelt verwendet werden.

Rechnung

Rechnungsnummer : string
Positionen : Position*

Artikel : string
Menge : int
Preis : double

In diesem Beispiel enthält eine Rechnung zwei Eigenschaften: eine Rechnungsnummer sowie mehrere Positionen. Jede einzelne Position enthält einen Artikel, eine Menge sowie den Preis.

Bei dieser Notation von Datentypen steht eine flüssige Vorgehensweise im Vordergrund. Es soll dem Team ermöglicht werden, Alternativen schnell diskutieren zu können. Dazu muss ein Lösungsansatz schnell notiert werden können. Der Detaillierungsgrad soll angemessen für den Entwurf einer Lösung sein. Es müssen nicht so viele Details enthalten sein, wie in der späteren Implementation. Insbesondere können auch hier wieder die konkreten Typen wie bspw. *string* oder *int* weggelassen werden, wenn sie sich aus dem Kontext ergeben. Bevor mit der Implementation begonnen wird, muss das Team ohnehin alle Datentypen konkret festlegen.

Tupel

In vielen Fällen ist es erforderlich, mehr als einen Wert fließen zu lassen. Dazu können Tupel verwendet werden. So kann eine Koordinate als Tupel aus einem *x* und einem *y* Wert dargestellt werden.

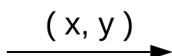


Abbildung 60: Ein Tupel bestehend aus einem *x* und einem *y*

Für den Empfänger dieses Datenflusses ist es leicht, dies umzusetzen. Wird die Funktionseinheit als Methode realisiert, erhält sie zwei Parameter. Allerdings kann es für den Sender solcher Daten schwierig sein, dies zu implementieren. Gehen wir davon aus, dass die Funktionseinheit in eine Methode übersetzt werden soll, muss diese ein Tupel als Rückgabewert liefern. Dies ist nur in wenigen Programmiersprachen unmittelbar möglich. Nur wenige Sprachen bieten syntaktisch die Möglichkeit, folgendes zu schreiben:

```
return (x, y);
```

In dem Fall kann man überlegen, ob es einen passenden Namen für das Tupel gibt. Im Fall von *x* und *y* könnte der Begriff *Koordinate* passen. Dann kann entweder ein vorhandener Typ aus der Laufzeitumgebung der verwendeten Sprache herangezogen werden, oder es wird ein eigener Typ definiert:

```
Koordinate
-----
x : int
y : int
```

Kann keine gute Bezeichnung für die Zusammenfassung der Daten gefunden werden, kann je nach Plattform ein generischer Tupel Datentyp verwendet werden. Bei .NET wäre dies *Tuple<T1, T2>*. Steht ein solcher allgemein Tuple Typ nicht im Framework zur Verfügung, lässt er sich leicht selbst implementieren.

Liefert eine Funktionseinheit mehrere Werte zurück und man findet keinen guten Namen für die zusammengefasste Ausgabe, kann dies allerdings auch ein Hinweis darauf sein, dass die Methode keine klare Zuständigkeit hat. Ist eine Funktionseinheit für genau einen Aspekt zuständig, wird man meist für ihre Ausgabedaten einen guten Namen finden.

Kombinationen

Natürlich können Tupel und Aufzählungen kombiniert werden. Das folgende Beispiel zeigt dies:

```
( ( int, string* )* )
```

Dies bedeutet, es fließt eine Aufzählung von Tupeln. Dies ist erkennbar, am rechten Stern, der sich auf das davor stehende Tupel bezieht. Jedes einzelne Tupel besteht dann aus zwei Elementen: einem *int* und einer Aufzählung von *string* Elementen. In C# könnten man dies wie folgt übersetzen:

```
List<Tuple<int, List<string>>>
```

Es dürfte ersichtlich sein, dass solche Definitionen in Datenflüsse vermieden werden sollten. Ohne Kontext ist nicht erkennbar, was die Bedeutung sein könnte. Es sollten dann mindestens Bezeichner verwendet werden, aus denen sich der Inhalt erschließen lässt.

```
( ( anzahl, label* )* )
```

Ggf. lässt sich auch ein guter Bezeichner für einen eigenen Typ finden.

Beispiel WordCount

In den vorangehenden Abschnitten wurde beschrieben, wie man als Softwareentwickler von den Anforderungen über einen Entwurf zum Code gelangen kann. Auf den folgenden Seiten wird die Vorgehensweise an einem Beispiel demonstriert.

Anforderungen

Um die Aufgabenstellung technisch einfach zu halten, soll eine Konsolenanwendung erstellt werden. Später werden wir uns auch mit dem Thema grafische UI befassen. Die Anwendung soll auf der Konsole mit dem Befehl *wordcount* aufgerufen werden:

```
$> wordcount  
Enter your text: Mary had a little lamb.  
Number of words: 5
```

Nach dem Aufruf wartet die Anwendung darauf, dass der Benutzer einen Text eingibt. Im Beispiel oben hat der Benutzer den Text “Mary had a little lamb.” eingegeben. Das Programm antwortet daraufhin mit der Anzahl der Worte, aus denen der Text besteht, im Beispiel fünf.

Die Zerlegung dieser Anforderungen mit Hilfe von *Dialogen* und *Interaktionen* ist hier sehr einfach. Die Anwendung besteht lediglich aus einem einzigen Dialog mit einer einzigen Interaktion. Auch wenn die Anwendung keine grafische UI hat, kann man dennoch von einem Dialog sprechen. Schließlich wartet das Programm darauf, dass der Benutzer einen Text eingibt. Erst durch diese Interaktion des Benutzers mit der Anwendung wird die Domänenlogik ausgeführt. In diesem Fall besteht die Domänenlogik darin, den Text in Worte zu zerlegen und diese dann zu zählen.

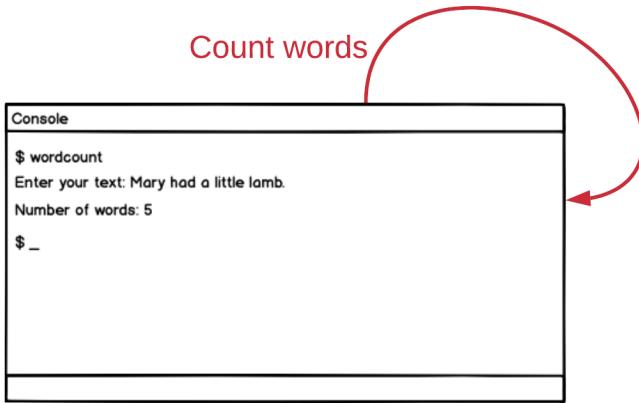


Abbildung 61: Dialog und Interaktion

Aus diesem Interaktionsdiagramm kann der Entwurf der obersten Ebene abgeleitet werden.

Entwurf der obersten Ebene

Die in der vorigen Abbildung gezeigte Interaktion kann in ein Flow Design Diagramm überführt werden.



Abbildung 62: Entwurf der obersten Ebene

Dieser Entwurf leitet sich unmittelbar aus der Interaktion *Count Words* ab. Die kreative Leistung besteht weniger darin, den Datenfluss in seiner Struktur zu zeichnen, als in der späteren Zerlegung bzw. Verfeinerung. Zur Überfügung des Interaktionsdiagramms in einen Entwurf wird der Interaktionspfeil betrachtet und in einen Datenfluss überführt. Anfangs- und Endpunkt der Interaktion werden zu Quelle und Senke des Datenflusses. Da die Interaktion *Count Words* in der UI beginnt und auch dort wieder endet, gilt dies auch für den Datenfluss. Er beginnt in der UI und verläuft über die Domänenlogik zurück zur UI. Die Domänenlogik ist dafür zuständig, das von der Interaktionen des Benutzers ausgelöste Verhalten der Anwendung herzustellen.

Nachdem die Struktur gezeichnet ist, werden die beiden Datenflüsse mit angemessenen Nachrichten versehen. Auch dies ist im Beispiel trivial. Da der Benutzer einen Text eingibt, liegt es nahe, dass dieser Text von der UI zur Weiterverarbeitung bereitgestellt wird. Der Text fließt somit zur Funktionseinheit *Count Words*. Diese ermittelt die Anzahl der Worte und liefert einen *int* Wert zur UI. Aufgabe der UI ist es, diese Zahl hübsch formatiert darzustellen.

Bei den Nachrichten der Datenflüsse kann man wieder von der UI ausgehen. Zu Beginn steht die Frage im Raum, welche Daten die UI an die Domänenlogik liefern könnte. Aus Sicht der UI sind dies jeweils die Eingaben des Benutzers. Da die UI im Beispiel den Benutzer bittet, einen Text einzugeben, ist schnell klar, dass dieser Text an die Domänenlogik zu liefern ist. Am Ausgang der Domänenlogik ist zu überlegen, was diese an die UI liefern sollte. Im Beispiel muss die UI die Ausgabe "Number of words: 5" anzeigen. Dazu hat die Domänenlogik zwei Möglichkeiten: sie könnte den vollständigen Text liefern oder lediglich die Anzahl der Wörter. Hier ist zu entscheiden, was man als Aufgabe der UI betrachtet. Die UI ist für die Präsentation der ermittelten Anzahl gegenüber dem Benutzer zuständig. Diese Präsentation könnte durchaus zukünftig an eine andere Landessprache angepasst werden. Als Entwickler erwartet man, dass diese Anpassung in der UI durchzuführen ist. Würde also die Domänenlogik den fertig formatierten String liefern, würde sie sich in Belange der UI einmischen. Eine klare Trennung der Zuständigkeiten liegt vor, wenn die Domänenlogik lediglich den anzugebenden Wert bestimmt und der UI die Aufgabe überlässt, den Wert in angemessener Weise dem Benutzer anzugeben.

Verfeinerung des Entwurf

Da das Beispiel so klein ist, könnte man geneigt sein, nun sofort mit der Implementation zu beginnen. Es liegt ein Entwurf der obersten Ebene der Anwendung vor, aus dem sich die Struktur des Programms ergibt. Die Implementation der Domänenlogik dürfte keine große Herausforderung sein. Würde man allerdings auf dieser Ebene des Entwurfs einer Lösung stehen bleiben, würde etwas fehlen. Es geht im Kern nicht um die grobe Struktur des Programms. Die dürfte für sehr viele Kommandozeilenprogramme sehr ähnlich aussehen. Stattdessen geht es beim Entwurf darum, eine Lösung für die Anforderungen zu erarbeiten. Die Anforderung lautet, die Anzahl der Worte in einem Text zu ermitteln. Der Entwurf muss zeigen, wie diese Herausforderung gelöst werden kann.

Schon bei der Beschreibung der Anforderungen war die Rede davon, dass der Text zunächst in Worte zerlegt wird und diese im Anschluss gezählt werden. Es liegt also nahe, genau diese Lösungsidee in einer Verfeinerung des Entwurfs darzustellen.

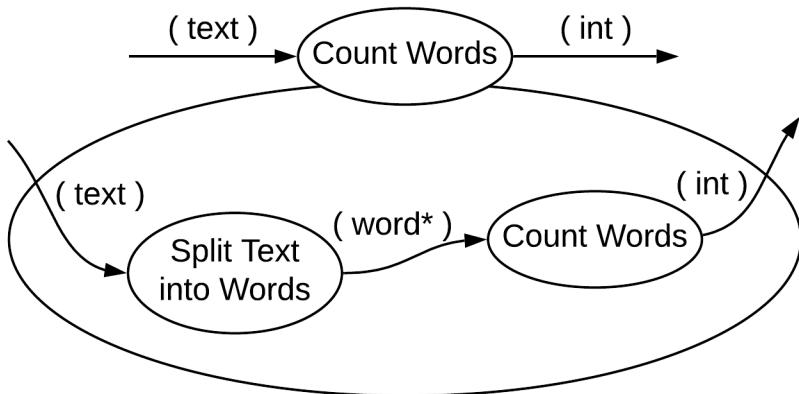


Abbildung 63: Verfeinerung von Count Words

Die Verfeinerung zeigt die beiden Funktionseinheiten *Split Text into Words* und *Count Words*. Wir werden später sehen, dass es günstig ist, diese Zerlegung vorzunehmen. Es werden nämlich weitere Anforderungen betrachtet. Dabei wird sich herausstellen, dass diese leichter umzusetzen sind, wenn die beiden Aspekte *Text zerlegen* und *Wörter zählen* bereits getrennt sind. Zwar kann diese Zerlegung auch zu einem späteren Zeitpunkt erfolgen. Doch in der Praxis zeigt sich immer wieder, dass eine spätere Zerlegung meist unterbleibt. An dieser Stelle ist wieder einmal die Entscheidung zu treffen, ob man frühzeitig optimiert und die Zerlegung sofort durchführt, obschon die Funktionseinheit *Count Words* im ersten Schritt sicher überschaubar aussehen wird. Oder unterlässt man die konsequente Trennung der Aspekte und vertraut darauf, dass diese später bei konkretem Bedarf nachgeholt wird. Die Empfehlung lautet ganz deutlich, die Trennung der Aspekte sofort durchzuführen. An die Vorgaben im Code wird sich meist gehalten. Nachfolgende Entwickler orientieren sich an dem Code, den sie vorfinden. Sind dabei die Aspekte klar getrennt, werden sie diese Trennung weiterhin einhalten. Geht es dagegen „durch Kraut und Rüben“, wird auch dieses Muster fortgeführt werden. Der Schaden ist größer, wenn zu wenig, nicht wenn zu viel zerlegt wird.

Die oben gezeigte Verfeinerung stellt die Lösung des Problems „Gib die Anzahl der Wörter in einem Text aus“ gut dar. Das Beispiel ist absichtlich klein gewählt, um die Darstellung von Flow Design in kleinen Schritten zu ermöglichen. In realen Projekten mag es seltener zu solcherart kleinen Anforderungen kommen. Doch auch dann lohnt es sich, das Problem durch den Entwurf einer Lösung vollständig zu durchdringen. Andernfalls befinden sich Entwickler zu früh auf der Ebene von Code.

Umsetzung

Bevor wir beginnen, den Entwurf mittels Code umzusetzen, müssen wir entscheiden, in welchen Klassen die Funktionseinheiten abgelegt werden sollen.

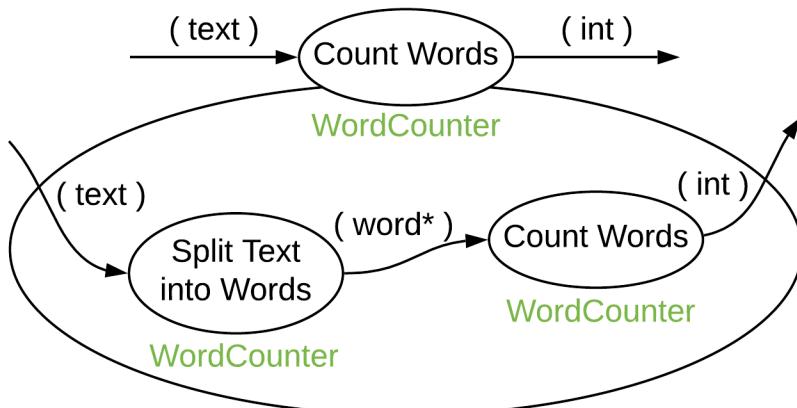


Abbildung 64: Ergänzen von Klassennamen

Die Entscheidung wird anhand der Aspekte getroffen, für die die Funktionseinheiten jeweils zuständig sind. Methoden, die sich mit demselben Aspekt befassen, werden zu Klassen zusammengefasst. Es ergibt sich eine hohe Kohäsion, die durch die Zusammenfassung in derselben Klasse ausgedrückt wird. So wie unterschiedliche Aspekte zu trennen sind, darf ein einzelner Aspekt nicht über die Codebasis verstreut werden. Es würde dann der Zusammenhang verloren gehen.

Funktionseinheiten die für unterschiedliche Aspekte verantwortlich sind, werden in getrennten Klassen abgelegt. Im Beispiel gilt dies zunächst nur für die Benutzerschnittstelle (Ui) und die Domänenlogik. Innerhalb der Domänenlogik sind die drei Funktionseinheiten für den selben Aspekt verantwortlich, folglich können sie zusammen in die Klasse *WordCounter* abgelegt werden.

Der Begriff *Klasse* wird hier synonym verwendet mit *Datei*. Wird zur Implementation eine Sprache verwendet, in der die Zusammenfassung von Funktionen und Methoden zu Klassen optional ist, kann häufig auf die Verwendung von Klassen verwiesen werden. In nicht-OO Sprachen existiert das Konzept von Klassen evtl. gar nicht. Dennoch ist es sinnvoll, Funktionalität zu Strukturen zusammenzufassen. In der Objektorientierung hat sich dazu der Begriff der Klasse etabliert. Im folgenden wird eine Implementation des Beispiels in JavaScript/NodeJS gezeigt. Hier wird auf Klassen verzichtet. Dennoch werden die Funktionen nicht alle in die selbe Datei

gestellt, sondern es wird anhand der Aspekte entschieden, ob Funktionen in die selbe Datei oder in getrennte Dateien abgelegt werden.
Die eigentlich Umsetzung des Entwurfs geht nun leicht von der Hand.
Folgende Abbildungen zeigen den Quellcode.

```
const ui = require("./consoleui.js");
const wordcount = require("./wordcount.js");

let text = ui.GetText();
let numberofWords = wordcount.CountWords(text);
ui.ShowResult(numberofWords);
```

Das oberste Modul ist in der Datei `app.js` abgelegt. Das Modul stellt den Einstiegspunkt in die Anwendung dar. Hier werden die beiden anderen benötigten Module eingebunden und aufgerufen.
Das Modul `consoleui.js` enthält die beiden Funktionseinheiten zur Kommunikation mit dem Anwender.

```
const readlineSync = require('readline-sync');

exports.GetText = function() {
    return readlineSync.question('Enter your text: ');
};

exports>ShowResult = function(numberofWords) {
    console.log(`Number of words: ${numberofWords}`);
};
```

Die Domänenlogik ist im Modul `wordcount.js` abgelegt.

```
exports.CountWords = function(text) {
    let words = SplitTextIntoWords(text);
    let numberofWords = CountWords(words);
    return numberofWords;
};

function SplitTextIntoWords(text) {
    return text.match(/[a-zA-Z]+/g);
}

function CountWords(words) {
    return words.length;
}
```

Und natürlich gibt es auch Tests der Domänenlogik.

```

const expect = require("chai").expect;
const rewire = require("rewire");

const wordcount = require("../wordcount.js");
const wordcountRewired = rewire("../wordcount.js")

describe("wordcount", () => {
    it("counts 'Mary had a little lamb.' as 5 words", () =>
    {
        expect(wordcount.CountWords(
"Mary had a little lamb.")).to.equal(5)
    });

    it("counts 'Wörter' as 2 words", () => {
        expect(wordcount.CountWords("Wörter")).to.equal(2)
    })
});

describe("wordcount.SplitIntoWords", () => {
    const SplitIntoWords = wordcountRewired.__get__(
"SplitIntoWords");

    it("splits a string at blanks into words", () => {
        expect(SplitIntoWords("a bb c ddd")).to.deep.equal([
"a", "bb", "c", "ddd"])
    });
});

describe("wordcount.CountWords", () => {
    const CountWords = wordcountRewired.__get__("CountWords");

    it("counts the array elements", () => {
        expect(CountWords(["a", "a", "a", "a"])).to.equal(4)
    });
});

```

Die Tests des Moduls `wordcount.js` testen einerseits die öffentliche Methode `CountWords`. Da diese durch das Modul exportiert wird, kann sie einfach aufgerufen werden. Die Methode `CountWords` ist eine Integration. Um auch die Operationen `SplitIntoWords` und `CountWords` zu testen,

werden diese mit dem *rewire* Modul für den Test zugänglich gemacht. Zu beachten ist hier ferner, dass es eine Integrationsmethode *CountWords* gibt, die zu einem Text die Anzahl der Wörter liefert. Ferner gibt es eine Operation *CountWords*, die zu einem Array von Wörtern die Anzahl liefert. Diese Überladung von Methodennamen innerhalb eines Moduls kann zu Verwirrungen führen. Sie wird im nächsten Inkrement aufgehoben, weil die Struktur dann ohnehin verändert werden muss.

Weitere Anforderungen

Nachdem der Product Owner zum ersten Inkrement sein OK gegeben hat, wünscht er sich eine Erweiterung der Anwendung. Das Programm soll eine Datei mit dem Namen `stopwords.txt` berücksichtigen. Diese Datei enthält Wörter, die nicht gezählt werden sollen. Diese bezeichnet der Product Owner als *Stoppwörter*. Jede Zeile der Textdatei enthält ein Wort. Die folgende Liste zeigt ein Beispiel:

```
in  
the  
a  
an
```

Bei diesem Beispiel der `stopwords.txt` Datei würde die Zählung des Beispielsatzes "Mary had a little lamb." auf die Anzahl vier kommen. Das liegt daran, dass das Wort "a" nicht mitgezählt wird.

Wir haben es nun mit einer Ergänzung der Anforderungen zu tun und müssen uns überlegen, wie diese sich im Entwurf niederschlägt. Zunächst einmal stellen wir fest, dass die neue Anforderung keinen Einfluss auf das Interaktionsdiagramm hat. Es kommt weder ein neuer Dialog noch eine weitere Interaktion hinzu. Dass die Stoppwörter nicht gezählt werden, ist ein Feature der Interaktion *Count Words*. Somit bleibt auch der Entwurf der obersten Ebene unverändert.

In der Verfeinerung des Entwurfs müssen wir jetzt dafür sorgen, dass die Stopwortdatei eingelesen und berücksichtigt wird. Das Einlesen der Datei ist ein fundamental anderer Aspekt als die bereits vorliegende Domänenlogik. Auch das Ausfiltern der Wörter vor dem Zählen hat weder mit der Zerlegung des Textes in Wörter noch mit dem Zählen zu tun. Wir können somit das neue Feature wie folgt in den vorhandenen Entwurf ergänzen.

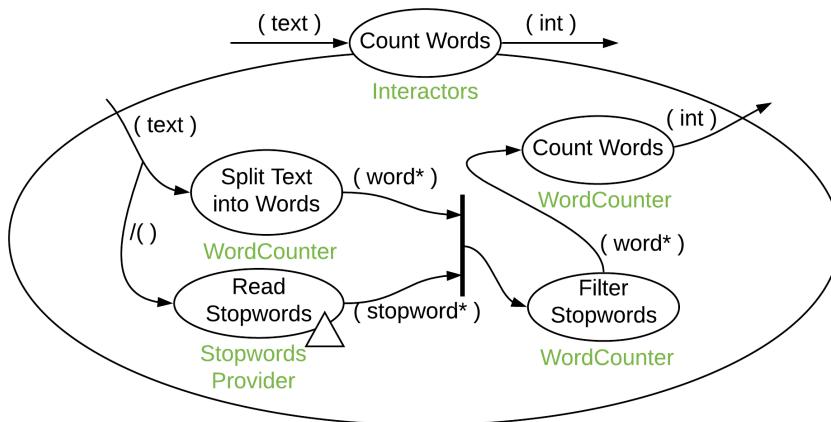


Abbildung 65: Ergänzung des Entwurfs

Die übergeordnete Funktionseinheit *Count Words* ist weiterhin für die Integration der enthaltenen Funktionseinheiten zuständig. Allerdings integriert sie nun nicht mehr ausschließlich Funktionseinheiten, die zum selben Aspekt *Domänenlogik* gehören. Das Einlesen der Datei stellt einen Ressourcenzugriff dar und dieser muss in einem Provider realisiert werden. Aus diesem Grund ist die Funktionseinheit *Read Stopwords* im Entwurf mit einem Dreieck markiert. So kann man schnell erkennen, dass dazu eine gesonderte Klasse erforderlich ist.

Die Erweiterung hat somit Einfluss auf die Klassen sowie die Zuordnung der Methoden zu den Klassen. Die übergeordnete Funktionseinheit *Count Words* wird nun in die Klasse *Interactors* verlagert. Diese Klasse kann beliebige Abhängigkeiten zu anderen Klassen haben, da sie genau für diesen Aspekt zuständig ist: *Integration*. Dagegen bleibt die Klasse *WordCounter* für die Domänenlogik zuständig, enthält Operationen und darf somit keinesfalls Abhängigkeiten zu anderen Klassen haben. Insbesondere nicht zu Klassen, die für den Ressourcenzugriff zuständig sind. Die Folge wäre andernfalls, dass die Domänenlogik von Ressourcenzugriffen abhängig wäre.

Umsetzung

Auch durch das neue Feature entsteht einfacher Quellcode. Beim automatisierten Testen entsteht lediglich die Herausforderung, dass nun mit der Datei `stopwords.txt` eine Ressource im Test benötigt wird. Durch die Strategie, beim Testen mit vielen Unit Tests und wenigen Integrationstests zu arbeiten, wirkt sich die Abhängigkeit des Programms von der Ressource

allerdings nur auf einen einzigen Integrationstest aus. Dieser Integrationstest stellt sicher, dass die einzelnen Funktionseinheit korrekt zusammenspielen. Die Masse der Logik wird durch isolierte Unit Tests überprüft, von denen keiner von der Ressource abhängig ist. Lediglich der Test des Providers benötigt eine Testdatei.

Die oberste Ebene der Anwendung im Modul app.js ändert sich nur leicht. Hier wird nun nicht mehr auf das Modul wordcount.js zugegriffen, sondern auf interactors.js.

```
const ui = require("./consoleui.js");
const interactors = require("./interactors.js");

let text = ui.GetText();
let numberofWords = interactors.CountWords(text);
ui.ShowResult(numberofWords);
```

Das Modul interactors.js integriert nun die Module wordcount.js und stopwordsprovider.js.

```
const wordcount = require("./wordcount.js");
const stopwordsprovider = require("./stopwordsprovider.js");

exports.CountWords = function(text) {
    let words = wordcount.SplitTextIntoWords(text);
    let stopwords = stopwordsprovider.ReadStopwords();
    let filteredWords = wordcount.FilterStopwords(words, stopwords);
    let numberofWords = wordcount.CountWords(filteredWords);
    return numberofWords;
};
```

Durch die Verlagerung der Integration in das Modul interactors.js, enthält das Modul wordcount.js jetzt nur noch Operationen. Diese werden sämtlich exportiert, damit sie im Modul interactors.js sichtbar sind und verwendet werden können.

```
exports.SplitTextIntoWords = function(text) {
    return text.match(/[a-zA-Z]+/g);
}

exports.CountWords = function(words) {
    return words.length;
}

exports.FilterStopwords = function(words, stopwords) {
    return words.filter(function (word) {
        return !stopwords.includes(word);
    });
}
```

Zuletzt liefert dann noch das Modul `stopwordsprovider.js` die Liste der Stopwörter aus der Datei `stopwords.txt`.

```
const fs = require("fs");

exports.ReadStopwords = function () {
    return fs.readFileSync("stopwords.txt").toString().split("\n");
}
```

Die Tests sind nun aufgeteilt in Unit Tests der Operationen im Modul `wordcount.js` sowie Integrationstests des Interactors im Modul `interactors.js`.

```

const expect = require("chai").expect;
const wordcount = require("../wordcount.js");

describe("wordcount.SplitIntoWords", () => {
    it("splits a string at blanks into words", () => {
        expect(wordcount.SplitTextIntoWords("a bb c ddd"))
            .to.deep.equal(["a", "bb", "c", "ddd"])
    });
});

describe("wordcount.CountWords", () => {
    it("counts the array elements", () => {
        expect(wordcount.CountWords(["a", "a", "a", "a"]))
            .to.equal(4)
    });
});

describe("wordcount.FilterStopwords", () => {
    it("filters the stopwords", () => {
        expect(wordcount.FilterStopwords(["a", "b", "a", "c"], ["a", "x"]))
            .to.deep.equal(["b", "c"])
    });
});

```

```

const expect = require("chai").expect;
const interactors = require("../interactors.js");

describe("interactors", () => {
    it("counts 'Mary had a little lamb.' as 4 words", () => {
        expect(interactors.CountWords("Mary had a little lamb."))
            .to.equal(4)
    });

    it("counts 'Wörter' as 2 words", () => {
        expect(interactors.CountWords("Wörter"))
            .to.equal(2)
    })
});

```

Mit diesen beiden Iterationen verlassen wir nun das WordCount Beispiel und wenden uns dem Thema *Zustand* zu.

Zustand innerhalb einer Interaktion

Unterschiedliche Arten von Zustand

In einem Softwaresystem können unterschiedliche Arten von Zustand auftreten. Wir unterscheiden vor allem Zustand, der innerhalb einer Interaktion benötigt wird von Zustand, der über mehrere Interaktionen hinweg verfügbar sein muss. Zustand innerhalb einer Interaktion muss im Entwurf nicht weiter gekennzeichnet werden, da jede Funktionseinheit Zustand halten kann. Auch ohne dass eine Funktionseinheit besonders gekennzeichnet wird, darf sie jederzeit Zustand halten. Wenn es der Klarheit und Verständlichkeit des Entwurfs dient, spricht aber nichts dagegen, auch in diesem einfachen Fall den Zustand explizit zu notieren.

Wird Zustand über mehrere Interaktionen hinweg benötigt, sollte er in jedem Fall explizit notiert werden, weil andernfalls das Verständnis des Entwurfs schwerfallen wird.

Zustand wird an den Funktionseinheiten durch eine Tonne notiert, wie in der folgenden Abbildung zu sehen. Zusätzlich kann der Inhalt des Zustands benannt werden.

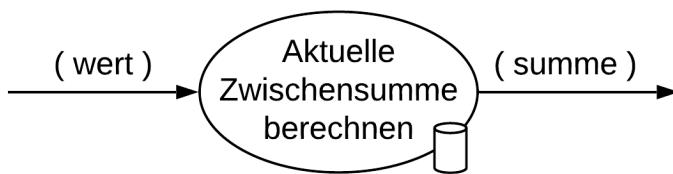


Abbildung 66: Zustand innerhalb einer Interaktion

Zustand, der über mehrere Interaktionen hinweg benötigt wird, sollte im Entwurf sichtbar sein, da diese Information wichtig ist, um den Entwurf der Lösung nachvollziehen zu können.

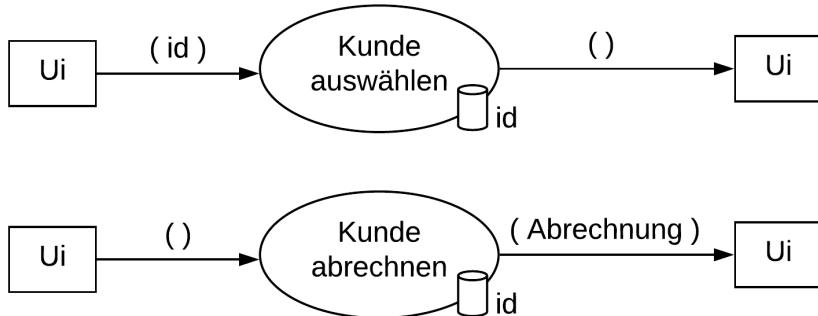


Abbildung 67: Zustand über zwei Interaktionen hinweg

In diesem Beispiel ist die Interaktion *Kunde auswählen* dafür zuständig, einen Kunden zu selektieren. Der Anwender selektiert in der Ui einen Kunden. Dessen *id* wird an die Funktionseinheit *Kunde auswählen* geliefert. Eine weitere Interaktion *Kunde abrechnen* wird durch den Anwender in der Ui ausgelöst. Dabei wird nicht erneut die *id* des selektierten Kunden geliefert. Insofern müssen die beiden Interaktionen durch gemeinsamen Zustand zusammen arbeiten. Das wird durch die Tonne an den beiden Funktionseinheiten ausgedrückt. Durch das Anschreiben von "id" wird der Inhalt näher spezifiziert. Auf diese Weise ist es leichter verständlich, wie die beiden Interaktionen zusammenarbeiten.

Mit Zustand über mehrere Interaktionen befasst sich das nächste Kapitel. Im folgenden Abschnitt geht es zunächst um Zustand, der innerhalb einer Interaktion benötigt wird.

Zustand innerhalb einer Funktionseinheit

Manchmal ist es erforderlich, dass eine einzelne Funktionseinheit Zustand hält. Nehmen wir als Beispiel die Funktionseinheit *Aktuelle Zwischensumme berechnen* aus dem folgenden Entwurf.

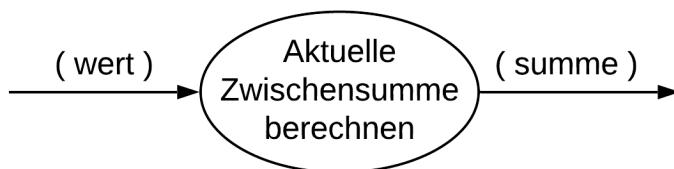


Abbildung 68: Aktuelle Zwischensumme berechnen

Die Funktionseinheit erhält eine Zahl als Eingabe und liefert jeweils die aktuelle Zwischensumme als Ausgabe. Um den Ausgabewert zu ermitteln, werden die Eingaben jeweils aufsummiert. Dazu muss sich die Funktionseinheit intern die aktuelle Zwischensumme als Zustand merken, um auf diesen Wert den Eingabewert addieren zu können. Eine Implementation in C# könnte wie folgt aussehen:

```
public class Zwischensumme
{
    private int _summe;

    public int Aktuelle_Zwischensumme_berechnen(int wert) {
        _summe += wert;
        return _summe;
    }
}
```

Die Methode *Aktuelle_Zwischensumme_berechnen* legt ihren Zustand in einem Feld der Klasse *Zwischensumme* ab. Das Feld *_summe* stellt somit den Zustand der Methode dar. In der Abbildung ist nicht erkennbar, dass die Funktionseinheit Zustand hält. Wenn es der Klarheit und besseren Verständlichkeit des Entwurfs dient, kann durch eine Tonne als Symbol an der Funktionseinheit explizit ausgedrückt werden, dass diese Funktionseinheit Zustand hält. Auf der anderen Seite ist es Funktionseinheiten auch ohne diese explizite Markierung gestattet, Zustand zu halten.



Abbildung 69: Eine Tonne drückt den Zustand explizit aus

Als zusätzliche Information kann der Zustand auch benannt werden. Dies ist wichtig, wenn aus dem Kontext nicht unmittelbar erkennbar ist, was der Inhalt des Zustands ist. Im Beispiel *CSV Viewer* muss die aktuelle Seite, die gerade ausgegeben wurde, als Zustand gehalten werden. Dort lautete die Beschriftung des Zustands an der Funktionseinheit *ExtractFirstPage*

“pageNo”. Damit wird hier ausgedrückt, dass der Zustand die Seitennummer enthält und nicht etwa den Seitenindex. Der Unterschied ist wichtig, weil hier ein klassisches 0/1 Problem lauert. Durch die Angabe *pageNo* wird ausgedrückt, dass der Zustand eine 1-basierende Seitennummer hält und nicht einen 0-basierten Index. Auf diese Weise kann die Beschriftung des Zustands der Klarheit und Verständlichkeit des Entwurfs dienen.

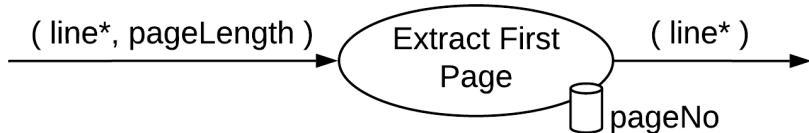


Abbildung 70: Zustand von *ExtractFirstPage*

In anderen Fällen kann die Beschriftung des Zustands dazu dienen, innerhalb eines Diagramms mehrere unterschiedliche Zustände zu unterscheiden. Auch dies tauchte im Beispiel *CSV Viewer* bereits auf. Im Entwurf für die Interaktion *NextPage* wird die schon beschriebene *pageNo* als Zustand gehalten. Darüber hinaus werden zusätzlich die eingelesenen Zeilen im Zustand *lines* gehalten. Damit aus dem Entwurf ersichtlich ist, welche Funktionseinheit die einzelnen Zustände benötigt, ist es wichtig, sie zu beschriften.

Zustand aus der Funktionseinheit herausgezogen

Als Alternative zu Funktionseinheiten, die den von ihnen benötigten Zustand selbst halten, kann der Zustand auch von außen reingereicht werden. Die einzelnen Funktionseinheiten sind dann selbst zustandslos und erhalten die benötigten Daten vollständig als Eingabe über den Datenfluss zur Verfügung gestellt. Das Beispiel *Aktuelle Zwischensumme berechnen* von oben sieht dann wie folgt aus.

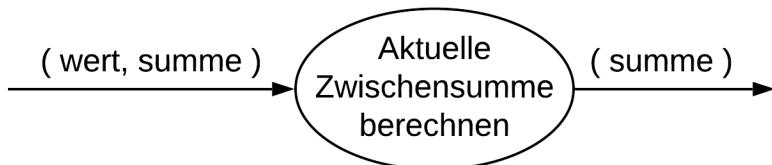


Abbildung 71: Der Zustand fließt

Das Herauslösen des Zustands aus einer Funktionseinheit wirkt sich meist positiv auf die Testbarkeit der Funktionseinheit aus. Das liegt daran, dass der Zustand nun nicht mehr die private Angelegenheit der Funktionseinheit ist. Sehen wir uns an, wie die Implementation des Entwurfs aussieht.

```
public class State<T>
{
    private T _item;

    public void Set(T item) {
        _item = item;
    }

    public T Get() {
        return _item;
    }
}
```

Die generische Klasse *State*<*T*> dient dazu, beliebigen Zustand zu implementieren. Die Klasse hält eine Instanz des generischen Typs *T*. Reicht man eine Instanz von *State*<*T*> in eine Funktionseinheit, kann diese den Zustand lesen, modifizieren und zurückschreiben. Dieselbe Instanz kann an unterschiedliche Funktionseinheiten gegeben werden, so dass diese den Zustand dann gemeinsam verwenden.

```

public int Aktuelle_Zwischensumme_berechnen(int wert, State<int> summe) {
    var s = summe.Get();
    s += wert;
    summe.Set(s);
    return s;
}

```

Die Funktion *Aktuelle_Zwischensumme_berechnen* erhält als ersten Parameter wieder den aufzusummierenden Wert. Über den zweiten Parameter erhält sie zusätzlich den Zustand: Dieser enthält jeweils die aktuelle Summe, so dass nun zunächst der bisherige Wert mit *Get* gelesen wird. Anschließend kann der Wert verändert und mit *Set* zurückgeschrieben werden. Hier ist zu beachten, dass es nicht genügen würde, als zweiten Parameter die bisherige Summe als *int* reinzureichen. Da es sich bei *int* um einen *Valuetype* und nicht um einen *Referenztype* handelt, würde der Wert zwar innerhalb der Methode verändert, aber nicht wieder nach außen gegeben. Somit ginge der Wert verloren und wäre eben gerade nicht als Zustand gehalten. Die Klasse *State<T>* vermeidet dies, in dem sie technisch gesehen einen Zeiger auf den Zustand darstellt. Innerhalb der Methode wird dann nicht der Zeiger verändert, sondern der Wert auf den der Zeiger zeigt. Bei Referenztypen wie *List<T>* etc. wäre es nicht notwendig, diese in eine Instanz von *State<T>* zu verpacken, weil sie ohnehin schon einen Zeiger auf die Daten darstellen. Die Idee dahinter wird allerdings klarer, wenn auch in diesem Fall der Zustand durch *State<List<T>>* implementiert wird.

Ein Aufrufer muss nun die Methode *Aktuelle_Zwischensumme_berechnen* zusätzlich mit dem Zustand aufrufen.

```

var state = new State<int>();
var summe = Zwischensumme.Aktuelle_Zwischensumme_berechnen(1, state);

```

Zustand in den Flow gestellt

Im vorhergehenden Beispiel hat die Methode *Aktuelle_Zwischensumme_berechnen* den aufzusummierenden Wert sowie die bisherige Zwischensumme als Eingabe erhalten. Die Methode hat dann folgende Aufgaben übernommen:

- Die bisherige Zwischensumme aus dem Zustand entnehmen.

- Den neuen Wert auf die Zwischensumme aufaddieren.
- Die aktualisierte Zwischensumme in den Zustand zurückschreiben.
- Die aktualisierte Zwischensumme als Ergebnis zurück liefern.

Damit sind in der Methode allerdings zwei Aspekte vermischt:

- Umgang mit Zustand
- Berechnen der Zwischensumme

Diese Vermischung von Aspekten lässt sich vermeiden, wenn der Umgang mit dem Zustand aus der Methode herausgelöst wird. Im Entwurf sieht das wie folgt aus:

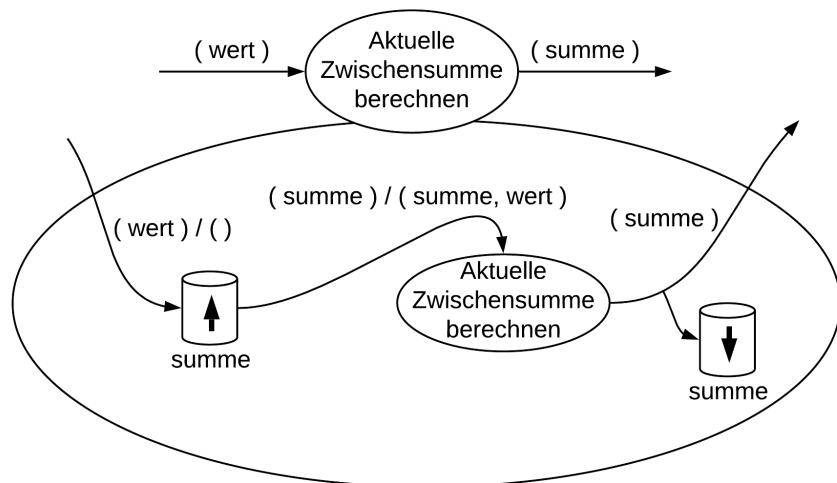


Abbildung 72: Zustand in den Flow gestellt

Nun ist die Aufgabe im Entwurf in drei Schritte zerlegt. Diese drei Schritte werden von der Funktionseinheit *Aktuelle Zwischensumme berechnen* integriert. Es werden also gleich an zwei Stellen Aspekte getrennt: der Umgang mit dem Zustand wird getrennt von der Berechnung der Zwischensumme. Ferner werden Integration und Operation getrennt. Das folgende Listing zeigt die zum Entwurf gehörende Implementation.

```

public class Zwischensumme
{
    private readonly State<int> _state = new State<int>();

    public int Aktuelle_Zwischensumme_berechnen(int wert) {
        var bisherige_Summe = _state.Get();
        var aktuelle_Summe = Aktuelle_Zwischensumme_berechnen(wert, bisherige_Summe);
        _state.Set(aktuelle_Summe);
        return aktuelle_Summe;
    }

    private int Aktuelle_Zwischensumme_berechnen(int wert, int summe) {
        return wert + summe;
    }
}

```

Der Zustand liegt nun als Feld `_state` in der Klasse. Die öffentliche Methode `Aktuelle_Zwischensumme_berechnen` kann automatisiert getestet werden. Allerdings wird dabei der Zustand implizit mit getestet. Dies ist für diesen konkreten Fall kein Problem. Ganz allgemein kann Zustand allerdings das automatisierte Testen erschweren, da der Zustand für die einzelnen Testfälle zunächst hergestellt werden muss. Gelingt dies ausschließlich über die öffentliche API, fallen Tests mitunter länger aus. In dem Fall ist zu prüfen, ob der Zustand für den Test zugänglich gemacht werden sollte. Bei C#/.NET kann dies über `internal` und das Attribut `InternalsVisibleTo` erreicht werden. Das folgende Listing zeigt dies exemplarisch. Damit die Tests in der Testassembly auf die Interna der Implementationsassembly zugreifen können, müssen die Interna dort mit dem Attribut `InternalsVisibleTo("beispiele.tests")` sichtbar gemacht werden.

```

[Test]
public void Test() {
    var sut = new Zwischensumme();
    sut._state.Set(42);

    var result = sut.Aktuelle_Zwischensumme_berechnen(8);
    Assert.That(result, Is.EqualTo(50));
}

```

Hier wird im Test der Zustand verändert durch den Aufruf von `sut._state.Set(42)`.

Zustand über Interaktionen hinweg

Was war doch nochmal eine Interaktion?

Ein Softwaresystem besteht aus Dialogen, über die der Benutzer mit dem System in Kontakt treten kann. Bei einer Web- oder Desktopanwendung ist der Dialog ein Bildschirmfenster, eine HTML Seite, ein Formular, o.ä. Bei Konsolenprogrammen kann man ebenfalls von einem Dialog sprechen, auch wenn dies auf den ersten Blick etwas künstlich erscheint. Ein typisches Konsolenprogramm wird mit Kommandozeilenparametern aufgerufen. Mindestens der Aufruf ist eine erste Interaktion des Benutzers mit dem Programm. Manche Programme stellen im Verlauf der Ausführung Rückfragen an den Anwender, so dass auch hier weitere Interaktionen stattfinden. Dialoge in Desktop- oder Webanwendungen enthalten Steuerelemente, die zur Anzeige und Interaktion genutzt werden können. Der Anwender kann durch Betätigen einer Schaltfläche oder eines Menüpunkts die Ausführung von Domänenlogik im System initiieren. Auch hier findet also eine Interaktion des Benutzers mit dem System statt.

Bei Interaktionen unterscheidet man zwei unterschiedliche Kategorien. Manche Interaktionen wirken eigenständig und sind nicht von anderen Interaktionen abhängig. Andere Interaktionen wirken gemeinsam. In realen Anwendungen gibt es viele Fälle, in denen mehrere Interaktionen zusammenwirken und voneinander abhängen. Genau um diesen Fall geht es hier im Zusammenhang mit Zustand. Wenn mehrere Interaktionen des Benutzers zu einem zusammenhängenden Ergebnis innerhalb der Anwendung führen, wird Zustand benötigt, der über die Grenze einer einzelnen Interaktion hinaus geht.

Am Beispiel des TicTacToe Spiels soll dies verdeutlicht werden. Hier macht es einen Unterschied, ob der Anwender mehrfach die Interaktion *Spielstein setzen* auslöst, oder ob zwischendrin die Interaktion *Neues Spiel* ausgelöst wird.

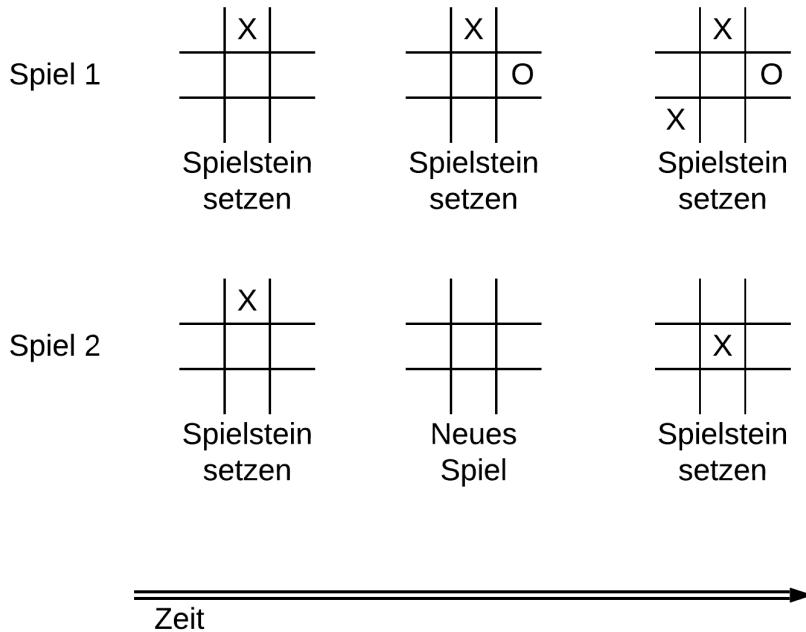


Abbildung 73: Mehrfache *Spielstein setzen* vs. *Spielstein setzen* und zwischendrin *Neues Spiel*

Es ist leicht zu erkennen, dass die beiden Abfolgen von Interaktionen zu einem unterschiedlichen Endergebnis des Spiels führen. Obwohl beide Abfolgen mit dem gleichen Spielstand beginnen, unterscheidet sich das Ergebnis. Die beiden Interaktionen *Spielstein setzen* und *Neues Spiel* beeinflussen sich gegenseitig. *Neues Spiel* beeinflusst die Interaktion *Spielstein setzen* dadurch, dass der Zustand des Spielbretts so verändert wird, dass er ein leeres Spielbrett repräsentiert. Umgekehrt beeinflusst *Spielstein setzen* die Interaktion *Neues Spiel*, weil im Anschluss der Zustand kein leeres Spielbrett mehr repräsentiert. Beide Interaktionen arbeiten zwangsläufig auf gemeinsamem Zustand. Im Entwurf können wir dies dadurch ausdrücken, dass wir auf der obersten Ebene des Entwurfs das Zustandssymbol an die beiden Interaktionen setzen.

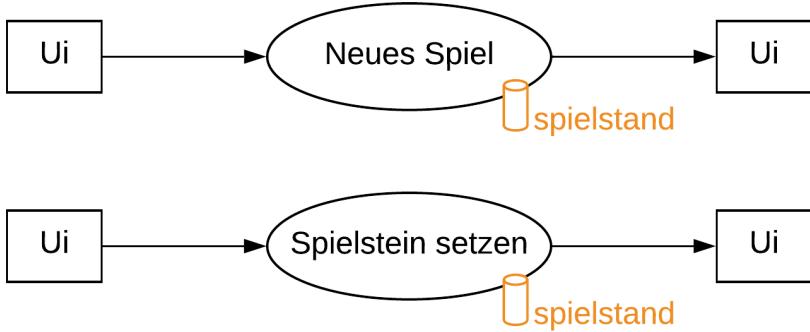


Abbildung 74: *Spielstein setzen* und *Neues Spiel* mit Zustandssymbol

Auf diese Weise wird ausgedrückt, dass die beiden Interaktionen sich auf gemeinsamen Zustand beziehen. Sowohl *Neues Spiel* als auch *Spielstein setzen* haben Zugriff auf den Zustand *Spielfeld*. Obschon es den Funktionseinheiten auch ohne die Tonne erlaubt ist, Zustand zu halten, ist die explizite Information hier für das Verständnis des Entwurfs wesentlich. Ohne die Tonne an den Funktionseinheiten würde man sich fragen, wie das Zusammenspiel der beiden Funktionseinheit funktionieren wird. Die Annotation der Zustandstonne unterstützt das Verständnis des Entwurfs. Es fällt so leichter, den Zusammenhang zwischen den beiden Funktionseinheiten zu erkennen. Die Zustandstonne sollte also vor allem dann ergänzt werden, wenn der Zustand von mehr als einer Funktionseinheit gemeinsam gehalten wird.

Zustand in mehreren Interaktionen verfügbar machen

Für die Implementation bietet es sich an, wieder das Konzept des expliziten Zustands zu verwenden. Der Zustand wird auf diese Weise aus den Funktionseinheiten, die ihn benötigen, herausgezogen und in den Datenfluss gestellt. Da in der Implementation an mehreren Stellen dieselbe Instanz des Zustands eingesetzt wird, teilen sich mehrere Interaktionen den selben Zustand.

```

public class TicTacToe
{
    private readonly State<char[,]> _state;

    public TicTacToe() {
        _state = new State<char[,]>();
    }

    public char[,] Spielstein_setzen(int x, int y) {
        var spielbrett = _state.Get();
        // Spielbrett manipulieren
        _state.Set(spielbrett);
        return spielbrett;
    }

    public char[,] Neues_Spiel() {
        var spielbrett = Leeres_Spielbrett();
        _state.Set(spielbrett);
        return spielbrett;
    }

    private static char[,] Leeres_Spielbrett() {
        return new char[3, 3];
    }
}

```

In diesem Code Beispiel wird die Instanz `_state` in den beiden Methoden `Spielstein_setzen` und `Neues_Spiel` verwendet. Somit teilen sich die beiden Interaktionen den selben Zustand. Damit das ganze funktioniert, muss sichergestellt sein, dass während des Programmablaufs ein und dieselbe Instanz der Klasse `TicTacToe` verwendet wird, da sie den Zustand hält. Wird eine neue Instanz erzeugt, wird auch der Zustand neu erzeugt. Die Klasse `TicTacToe` hat hier unter anderem die Verantwortlichkeit, den Zustand zu instanziieren. Die Klasse sorgt selbst dafür, dass der Zustand hergestellt wird und so von den beiden Methoden verwendet werden kann.

Um in der Lage zu sein, Zustand über mehrere Instanzen hinweg zu verwenden, muss der Zustand beim Instanziieren des Objekts injiziert werden.

```
public TicTacToe(State<char[,]> state) {  
    _state = state;  
}
```

Nun wird der Zustand im Konstruktor der Klasse *TicTacToe* injiziert. Damit liegt die Verantwortlichkeit, die Instanz des Zustands zu halten, nun außerhalb der Klasse *TicTacToe*. Derjenige, der eine Instanz von *TicTacToe* erzeugt, muss im Konstruktor eine Instanz für den Zustand bereitstellen. Damit liegt es nun in der Verantwortung des Ausrufers dafür zu sorgen, den Zustand selbst herzustellen. Der Aufrufer hat damit die Wahl, ob mit jeder neuen Instanz der Klasse *TicTacToe* auch ein neuer Zustand erzeugt wird, oder ob der Zustand über mehrere Instanzen der Klasse hinweg verwendet wird.

Beispiel TicTacToe

Am Beispiel eines TicTacToe Spiels wird nun das Thema Zustand nochmals im Zusammenhang dargestellt. Ferner wird die gesamte Vorgehensweise von den Anforderungen zum Code an diesem Beispiel dargestellt.

Anforderungen

Es soll ein TicTacToe Spiel realisiert werden, mit dem zwei Personen TicTacToe spielen können. Das Programm ist dafür zuständig, die Spielregeln zu überwachen. Es wird nicht gegen das Programm gespielt, sondern dieses stellt lediglich sicher, dass keiner der beiden Spieler einen ungültigen Zug machen kann. Ferner erkennt das Programm das Spielende und ermittelt, welcher der beiden Spieler gewonnen hat. Der Einfachheit halber beginnt immer der Spieler mit den "X" Spielsteinen.

Die folgende Abbildung zeigt, wie der Dialog der Anwendung aussehen könnte.

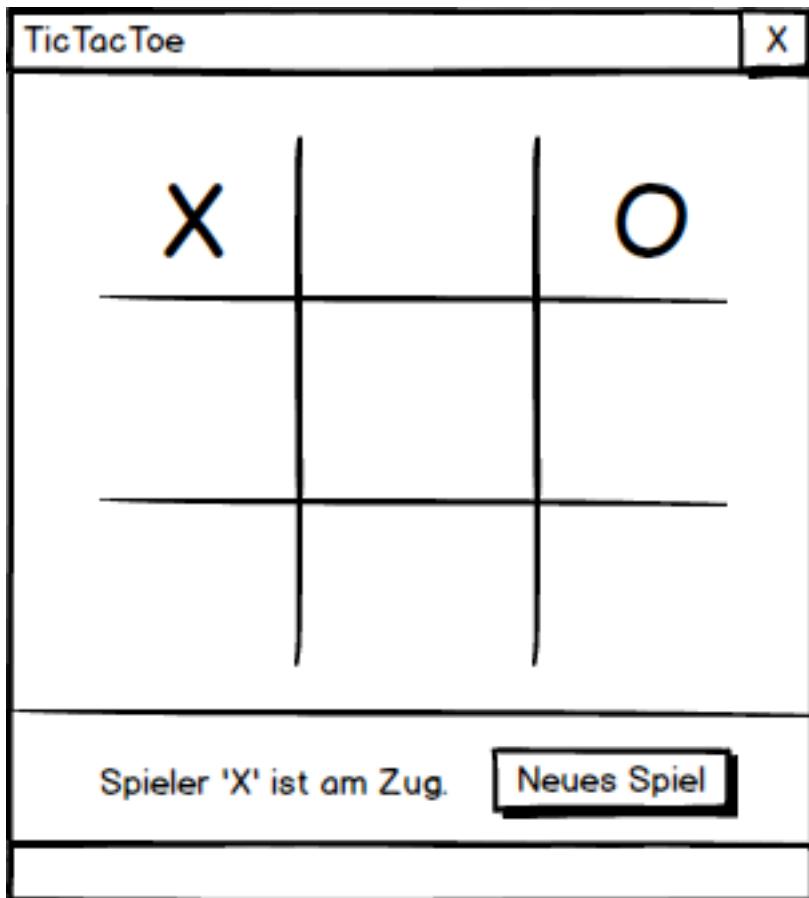


Abbildung 75: Mockup des Dialogs

Dialoge und Interaktionen

Die Analyse der Anforderungen beginnt mit dem Interaktionsdiagramm. Dieses schafft einerseits mehr Klarheit über die Anforderungen und bietet andererseits die Möglichkeit, Inkremeante zu bilden. Jede Interaktion stellt einen vertikalen Durchstich durch die Anforderungen dar, so dass der Product Owner jeweils für die folgende Iteration eine der Interaktionen auswählen kann. Die folgende Abbildung zeigt das Interaktionsdiagramm für das TicTacToe Spiel.

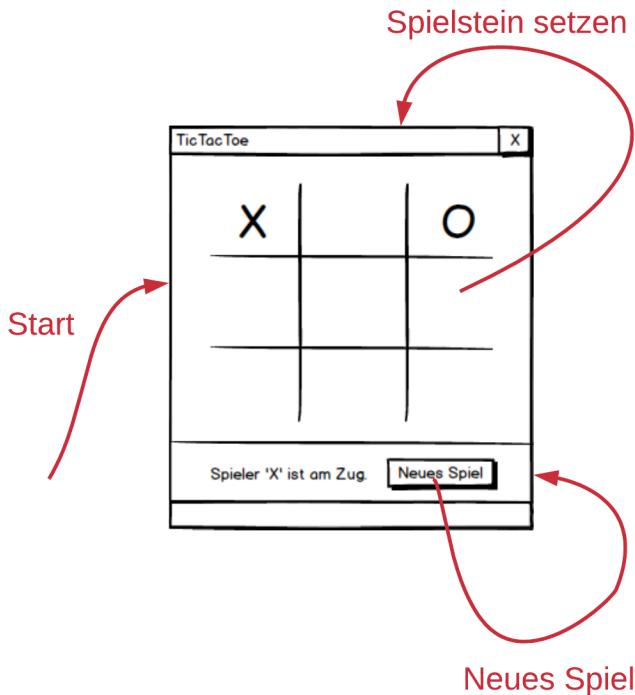


Abbildung 76: Interaktionsdiagramm für das TicTacToe Spiel

Entwurf der obersten Ebene

Im Anschluss an das Interaktionsdiagramm können die Entwickler den Entwurf der obersten Ebene erstellen. Hierbei geht es vor allem darum festzulegen, welche Nachrichten bei den einzelnen Interaktionen fließen. Ferner geht es darum festzulegen, ob der Sessionzustand in der Benutzeroberfläche oder der Domänenlogik gehalten werden soll.

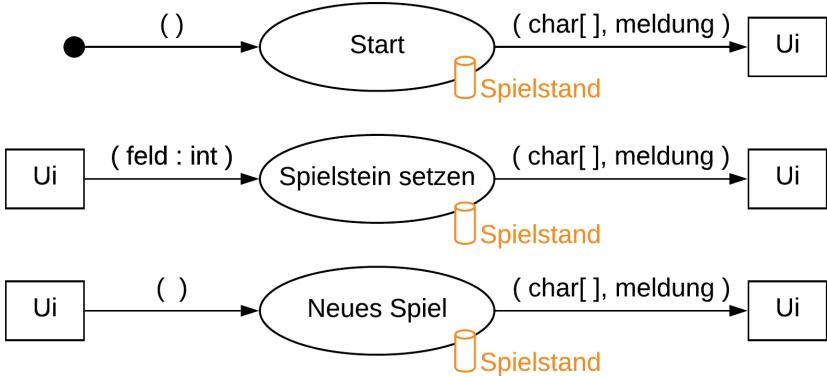


Abbildung 77: Entwurf der obersten Ebene

Die erste Interaktion des Benutzers ist der Start der Anwendung. Beim Start muss ein leeres Spielfeld hergestellt werden. Ferner muss eine Meldung angezeigt werden, die darauf hinweist, dass Spieler "X" am Zug ist. Das Spielfeld wird als *char* Array zur *Ui* übertragen. Die neun Spielfelder sind von links oben nach rechts unten jeweils als ein Zeichen im Array hinterlegt. Zu Beginn wird das leere Spielfeld als ein Array aus neun Leerzeichen dargestellt. Die Meldung ist ein *string*, den die *Ui* ohne weitere Interpretation anzeigt. Dies stellt einen Kompromiss dar. Sollte eine landessprachliche Anpassung vorgenommen werden, müssen die Meldungen entsprechend übersetzt werden. Für diesen Aspekt ist eigentlich die *Ui* zuständig. Das würde allerdings bedeuten, dass die Domänenlogik der *Ui* die Meldung bspw. als *enum* zur Verfügung stellt, den die *Ui* dann zu einem *string* wandelt. Für die ersten Iterationen wird der Aufwand reduziert, wenn man auf diese Übersetzung von *enum* Werten verzichtet. Gleichzeitig sollte der Aspekt der Meldungstexte in der Domänenlogik so freigestellt werden, dass der Wechsel zu einer anderen Realisierung bei Bedarf leicht von der Hand geht.

Der Spielstand und damit der Zustand der Anwendung wird hier in der Domänenlogik gehalten. Die *Ui* erhält alle relevanten Informationen, die sie zur Anzeige des Spielbretts benötigt. Bei der Interaktion *Spielstein setzen* liefert die *Ui* daher lediglich den Index des Feldes, das der Anwender angeklickt hat. Klickt er links oben in das erste Feld, liefert die *Ui* eine null als Index. Klickt der Anwender unten rechts in das Feld, liefert sie acht als Index. Durch die Interaktion wird ggf. der Spielstand verändert. Als Ergebnis liefert die Domänenlogik den aktualisierten Spielstand sowie eine Meldung an die *Ui*.

Bei der Interaktion *Neues Spiel* wird der Zustand des Spielbretts wieder auf ein leeres Spiel initialisiert. Zur UI wird erneut das Spielbrett sowie eine Meldung übertragen.

Verfeinern der Interaktion Start

Bevor mit der Implementation des Spiels begonnen werden kann, müssen die Entwürfe der obersten Ebene verfeinert werden. An dieser Stelle findet der Übergang statt vom Entwurf in die Breite zum Entwurf in die Tiefe. Es wird jetzt nicht mehr mit allen Interaktionen gearbeitet, sondern eine einzelne ausgewählt und nur diese wird verfeinert. Beginnen wir mit der Interaktion *Start*.

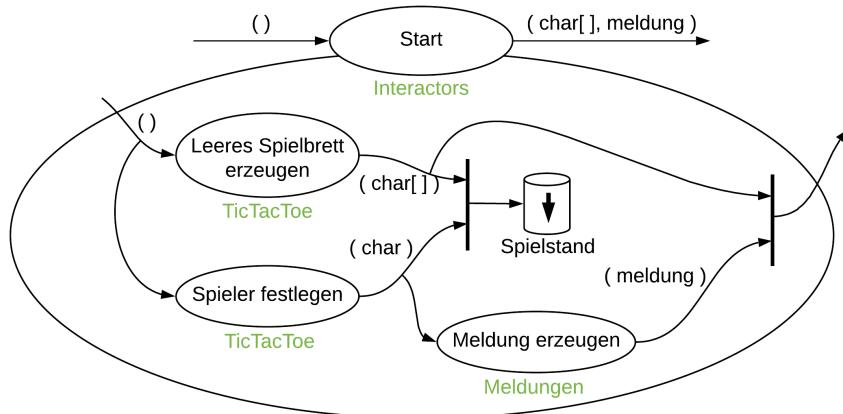


Abbildung 78: Verfeinerung der Interaktion *Start*

Die Verfeinerung beginnt damit, dass ein leeres Spielbrett hergestellt wird. Ferner wird festgelegt, welcher Spieler beginnt. Dies ist derzeit immer der Spieler "X". Beide Informationen werden als Zustand *Spielstand* abgelegt. Zuletzt wird eine Meldung erzeugt. Dazu fließt die Information, welcher Spieler am Spiel ist, in Form eines einzelnen Zeichens in die Funktionseinheit. Aufgabe von *Meldung erzeugen* ist es, aus dem "X" oder "O" die passende Meldung zu erzeugen.

Umsetzung von Start

Für die Umsetzung der TicTacToe Anwendung habe ich C# mit Mono auf dem Mac gewählt. Als UI Technologie wird das Open Source Framework

*Eto.Forms*⁵ verwendet. Damit ist die Anwendung plattformunabhängig unter MacOS, Windows und Linux lauffähig. Das folgende Listing zeigt die Klasse *Interactors* mit der Methode *Start*.

```
public class Interactors
{
    private char[] _spielbrett;
    private char _spieler;

    public (char[] Spielbrett, string Meldung) Start() {
        var spielbrett = TicTacToe.Leeres_Spielbrett_erzeugen();
        var spieler = TicTacToe.Spieler_festlegen();

        _spielbrett = spielbrett;
        _spieler = spieler;

        var meldung = Meldungen.Meldung_erzeugen(spieler);

        return (spielbrett, meldung);
    }
}
```

Der Ablauf entspricht dem zuvor gezeigten Entwurf. Die Klasse *Interactors* ist abhängig von zwei weiteren Klassen, die ebenfalls bereits im Entwurf benannt sind: *TicTacToe* enthält die Domänenlogik, *Meldungen* ist für das Erzeugen der Meldungen zuständig. Die Methoden dieser Klassen sind statisch, da sie keinen Zustand benötigen.

⁵ <https://github.com/picoe/Eto>

```
public static class TicTacToe
{
    private const char Leeres_Feld = ' ';

    public static char[] Leeres_Spielbrett_ergeben() {
        var spielbrett = new char[9];
        for (int i = 0; i < 9; i++) {
            spielbrett[i] = Leeres_Feld;
        }
        return spielbrett;
    }

    public static char Spieler_festlegen() {
        return 'X';
    }
}
```

Die beiden Methoden sind trivial, da sie lediglich dafür verantwortlich sind, den initialen Zustand des Spiels herzustellen. Desgleichen gilt für die Klasse *Meldungen*, auch sie ist trivial.

```
public class Meldungen
{
    public static string Meldung_ergeben(char spieler) {
        return $"Spieler '{spieler}' ist am Zug.";
    }
}
```

Der aufwendigste Teil dieser ersten Iteration ist, wie so häufig, die grafische UI.

```
public class Ui : Form
{
    private readonly Button[] _buttons;
    private readonly Label _message;

    public event Action<int> Token_set;

    public event Action New_game;

    public Ui() {
        Title = "TicTacToe";
        ClientSize = new Size(260, 160);

        Menu = new MenuBar {
            QuitItem = new Command((sender, e) => Application.Instance.Quit()) {
                MenuText = "Quit",
                Shortcut = Application.Instance.CommonModifier | Keys.Q
            }
        };
        _buttons = CreateButtons().ToArray();

        _message = new Label();
        Content = new TableLayout {
            Spacing = new Size(5, 5),
            Padding = new Padding(10, 10, 10, 10),
            Rows = {
                new TableRow(CreateStackLayout(_buttons[0], _buttons[1], _buttons[2])),
                new TableRow(CreateStackLayout(_buttons[3], _buttons[4], _buttons[5])),
                new TableRow(CreateStackLayout(_buttons[6], _buttons[7], _buttons[8])),
                new TableRow(_message) {ScaleHeight = true}
            }
        };
    }
}
```

```

private StackLayout CreateStackLayout(params Button[] buttons) {
    var stackLayout = new StackLayout{ Orientation = Orientation.Horizontal};
    foreach (var button in buttons) {
        stackLayout.Items.Add(button);
    }

    return stackLayout;
}

private IEnumerable<Button> CreateButtons() {
    for (var i = 0; i < 9; i++) {
        var button = new Button();
        var feld = i;
        button.Click += (o, e) => {
            Token_set?.Invoke(feld);
        };
        yield return button;
    }
}

public void Show_score(char[] board, string message) {
    for (var i = 0; i < 9; i++) {
        _buttons[i].Text = board[i].ToString();
    }
    _message.Text = message;
}
}

```

Die Struktur des Dialogs ist in Form einer Tabelle aufgebaut. Die Tabelle besteht aus vier Zeilen. Die ersten drei enthalten jeweils ein *StackLayout* Element für je drei *Button* Elemente. Die letzte Zeile enthält ein *Label* zur Anzeige der Meldungen.

An den Buttons wird lediglich die *Tag* Eigenschaft verwendet. Sie nimmt den Index der Schaltfläche auf. Dies ist bereits ein Vorgriff auf die nächste Iteration, in der jede Schaltfläche einen Event auslösen muss. Da der Event den Index des angeklickten Spielfeldes als Parameter tragen muss, wird er hier bereits im *Tag* abgelegt.

Das Ergebnis der ersten Iteration zeigt die folgende Abbildung.

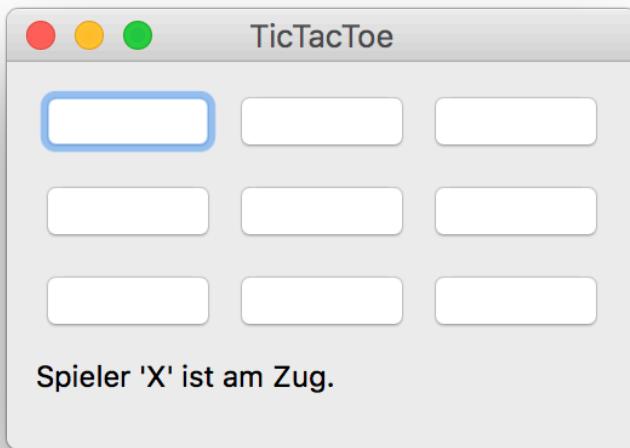


Abbildung 79: Das Ergebnis der *Start* Iteration

Verfeinern der Interaktion *Spielstein setzen*

In der nächsten Iteration soll die Interaktion *Spielstein setzen* realisiert werden. Der Anwender möchte auf eines der Spielfelder klicken, um dort seinen Spielstein zu platzieren. Diese Interaktion besteht aus mehreren Features. Die Aufzählung der Features soll es ermöglichen, zunächst Details wegzulassen, um so die Wahrscheinlichkeit zu erhöhen, kurzfristig Feedback durch den Product Owner zu erhalten. Es geht also nicht darum, eine vollständige Auflistung aller Features zu erstellen, aus denen die Interaktion besteht. Vielmehr geht es darum Features zu identifizieren, die zunächst weggelassen werden können. Ziel ist es, innerhalb kurzer Zeit einen Durchstich, ein Inkrement, zu realisieren. Wenn in kleinen Beispielen Features identifiziert werden können, die zunächst weggelassen werden, ist dies auch in realen Projekten möglich.

Die Interaktion *Spielstein setzen* verfügt über folgende Features:

- **Spieler wechseln**
Nachdem der Spielstein gesetzt wurde, wird der Spieler nach den Regeln des Spieles gewechselt. Konnte kein Stein platziert werden, wird der Spieler nicht gewechselt.

- **Validierung des Spielzugs**
Es wird geprüft, ob der Spielstein in das Feld platziert werden darf.
Das ist nur möglich, wenn das Spielfeld leer ist.
- **Erkennung des Spielendes**
Das Programm erkennt, ob das Spiel beendet ist, weil alle Felder besetzt sind.
- **Gewinnermittlung**
Der Gewinner des Spiels wird ermittelt.

Die aufgelisteten Features erheben nicht den Anspruch auf Vollständigkeit. Möglicherweise kann die Interaktion noch feiner zerlegt werden. Im Vordergrund steht die Möglichkeit, Details weglassen zu können.

Als nächstes kann die Interaktion nun entworfen werden. Dabei können zunächst alle Features berücksichtigt werden. Erst bei der Implementation wird dann entschieden, einzelnen Features zunächst wegzulassen, um schnell zu einem lauffähigen Inkrement zu gelangen. Zeigen sich allerdings schon beim Entwurf größere Herausforderungen, können auch im Entwurf zunächst Features weggelassen werden. Manchmal gelangt ein Team durch einen Teilentwurf und seine Umsetzung zu den tieferen Erkenntnissen, die erforderlich sind, um den gesamten Entwurf zu vervollständigen.

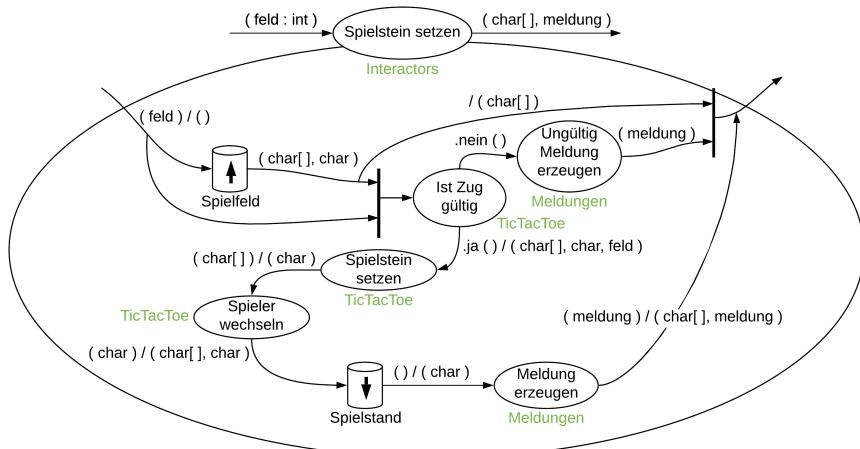


Abbildung 80: Verfeinerung der Interaktion *Spielstein setzen*

Umsetzung von *Spielstein setzen*

Für die Umsetzung habe ich zunächst auf die Erkennung des Spielendes und des Gewinners verzichtet. Die Methode, welche die Interaktion umsetzt, zeigt folgendes Listing:

```
public (char[] Spielbrett, string Meldung) Spielstein_setzen(int feld) {
    var spielbrett = _spielbrett;
    var spieler = _spieler;
    var meldung = "";

    TicTacToe.Ist_Zug_gültig(spielbrett, feld,
        onUngültig: () => {
            meldung = Meldungen.Ungültig_Meldung_erzeugen();
        },
        onGültig: () => {
            spielbrett = TicTacToe.Stein_setzen(spielbrett, spieler, feld);
            spieler = TicTacToe.Spieler_wechseln(spieler);
            _spielbrett = spielbrett;
            _spieler = spieler;
            meldung = Meldungen.Meldung_erzeugen(spieler);
        });
}

return (spielbrett, meldung);
}
```

Zuerst wird geprüft, ob der Zug überhaupt gültig ist. Dazu fließt das Spielbrett sowie der Index des gewählten Spielfeldes in die Methode *Ist_Zug_gültig*. Ferner werden zwei Callbacks übergeben. Die erste wird aufgerufen, wenn der Zug ungültig ist, die andere wenn er gültig ist. Die Implementation dieser Methode in C# sieht wie folgt aus:

```

public static void Ist_Zug_gültig(
    char[] spielbrett,
    int feld,
    Action onUngültig,
    Action onGültig) {
    if (spielbrett[feld] == Leeres_Feld) {
        onGültig();
    }
    else {
        onUngültig();
    }
}

```

Es wird geprüft, ob das Spielfeld am gegebenen Index leer ist. Falls das der Fall ist, wird die Action *onGültig* aufgerufen, andernfalls die Action *onUngültig*. Weitere Details zu einer solchen Fallunterscheidung folgen später im Kapitel über Fallunterscheidungen.

Die beiden Interaktionen *Start* und *Spielstein setzen* verwenden gemeinsamen Zustand. Ferner bleibt der Zustand über mehrere Aufrufe von *Spielstein setzen* hinweg erhalten. Es macht also einen Unterschied, ob und mit welchen Parametern *Spielstein setzen* aufgerufen wird. Dieser gemeinsame Zustand wird hier in Form von privaten Feldern der Klasse *Interactors* gehalten. Das Feld *_spielbrett* enthält das Spielbrett, das Feld *_spieler* den Spieler, der gerade am Zug ist. Um deutlich zu machen, wann der Zustand gelesen wird und wann er wieder zurückgeschrieben wird, wird er im Code zu Beginn in lokale Variablen kopiert. Ferner wird er später nach den Änderungen wieder in die Felder zurückkopiert. Im Entwurf sind diese beiden Vorgänge ebenfalls sichtbar in Form der Zustandstonne mit Pfeil nach oben für das Lesen und Pfeil nach unten für das Schreiben des Zustands.

Fazit

Das TicTacToe Spiel ist damit noch nicht fertig implementiert. Und trotzdem kann der Product Owner auf dem jetzigen Stand bereits wertvolles Feed-

back geben. Die Wichtigkeit der Arbeit in kleinen Inkrementen kann gar nicht genug betont werden. Immer wieder hören wir in Trainings, dass eine weitere Zerlegung nicht möglich sei. Und dann geht es doch. Es braucht etwas Übung, konsequent in Interaktionen und Features zu arbeiten. Der Vorteil dieser Arbeitsweise ist, dass kein Code “auf Halde” produziert wird, von dem das Team lange nicht weiß, ob er so benötigt wird. Nach jedem Inkrement kann der Product Owner entscheiden, ob das Inkrement abgeschlossen ist, oder ob Details korrigiert werden müssen. Damit weiß das Team zu jedem Zeitpunkt, wo es steht. Ferner sieht der Product Owner in kurzen Abständen den Fortschritt. Das schafft Vertrauen zwischen Team und Product Owner und ersetzt vielfach das Reporting durch ein Burndown Chart o.ä.

Zustand im Portal

Sessions

Betrachtet man den Zustand eines Softwaresystems, ergibt sich eine weitere Differenzierung. Der Anwender befindet sich innerhalb der Anwendung im Zeitraum vom Start bis zum Beenden der Anwendung innerhalb einer sogenannten *Session*. Mit dem Start der Anwendung wird die Session eröffnet, beim Beenden wird sie verlassen. Darüberhinaus gibt es bei den meisten Anwendungen Zustand in Form von persistenten Daten. Eine Datenbank oder Dateien im Dateisystem sind Beispiele für persistente Daten. Diese zeichnen sich dadurch aus, dass sie das Beenden der Anwendung überdauern.

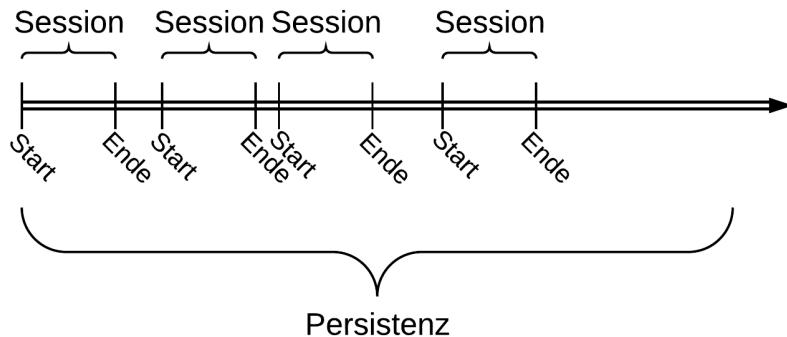


Abbildung 81: Zeitstrahl mit Persistenz und Sessions

Über persistente Daten wird weiter unten im Abschnitt über Ressourcen zu sprechen sein. Betrachten wir zunächst die Session etwas genauer. Eine Session hält den aktuellen Zustand der Anwendung bezogen auf einen einzelnen Benutzer fest. Nehmen wir dazu wieder das TicTacToe Spiel als Beispiel. Solange die Anwendung läuft, hält sie den Zustand des Spielbretts fest. Beim Start der Anwendung ist das Spielbrett leer. Klickt der Anwender nun in eines der Spielfelder, wird dort ein Spielstein gesetzt. Klickt er anschließend in das selbe Feld, wird dort kein Spielstein mehr gesetzt, da die Anwendung sich "erinnert" dass dort durch den vorhergehenden Zug

bereits ein Spielstein abgelegt wurde. Es stellt sich die Frage, wo dieser Zustand des Spiels abgelegt wird.

Wenn wir uns die oberste Ebene des Entwurfs anschauen, ergeben sich zwei Möglichkeiten, die Daten einer Session abzulegen. Wir können die Session im Portal oder in der Domänenlogik ablegen. Im vorherigen Abschnitt lag der Zustand in der Domänenlogik. Doch er kann auch im Portal liegen.

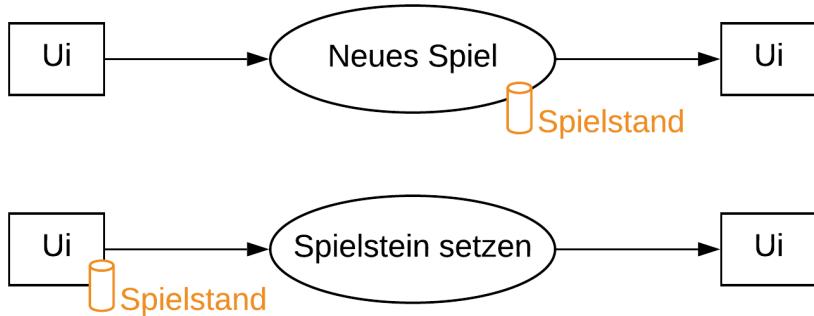


Abbildung 82: Zustand im Portal oder in der Domänenlogik

Solange Portal und Domänenlogik im selben Prozess auf der gleichen Maschine laufen, sind wir in der Entscheidung relativ frei. Liegt der Zustand im Dialog, müssen Informationen aus dem Zustand beim Auslösen einer Interaktion zur Domänenlogik übertragen werden. Liegt der Zustand dagegen in der Domänenlogik, entfällt diese Übertragung des Zustands. Die Verortung des Zustands wirkt sich also auf die Datenflüsse aus.



Abbildung 83: Spielstein setzen, wenn der Zustand in der Domänenlogik liegt

Bei einer typischen Desktopanwendung, wie beim Beispiel des TicTacToe Spiels, können wir frei entscheiden, ob das Portal oder die Domänenlogik den Zustand *Spielbrett* enthält. Natürlich hat die Entscheidung Einfluss darauf, welche Daten zwischen Portal und Domänenlogik fließen. Hält das

Portal die Information über das Spielbrett, muss sie den Zustand des Spielbretts mit jedem Zug an die Domänenlogik liefern. Diese entscheidet dann über die Veränderung des Spielbretts und liefert ein möglicherweise geändertes Spielbrett zurück zum Portal.

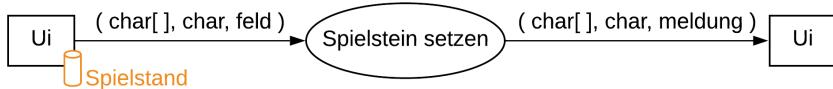


Abbildung 84: Spielstein setzen, wenn der Zustand im Portal liegt

Legen wir den Zustand des Spielbrett dagegen in der Domänenlogik ab, braucht das Portal die Domänenlogik lediglich darüber zu informieren, in welches Spielfeld der Anwender geklickt hat. Die Domänenlogik hält den Zustand des Spielbretts und hat somit wieder alle Daten zur Verfügung, um einen möglicherweise geänderten Zustand des Spielbretts an das Portal zu liefern. Das Portal muss hier lediglich den Spielstand anzeigen und braucht sich keine Gedanken darüber machen, das Spielbrett intern so abzulegen, dass es bei Bedarf wieder an die Domänenlogik geliefert werden kann. Bei Desktopanwendungen oder allgemeiner, solange Portal und Domänenlogik im selben Prozess oder wenigstens auf der selben Maschine laufen, ist die Entscheidung sehr frei. Handelt es sich jedoch um eine verteilte Anwendung, kann es Vorteile haben, die Domänenlogik frei von Sessionzustand zu halten. Verteilte Anwendung bedeutet hier, dass die Bestandteile der Anwendung auf unterschiedlichen Maschinen laufen.

Verteilte Anwendung

Handelt es sich um eine verteilte Anwendung, ist es vorteilhaft, wenn der Server sich nicht um den Zustand der einzelnen Anwender zu kümmern braucht. Liefert ein Client seinen Sessionzustand mit jedem Request an den Server gleich mit, kann der Server allein aufgrund der eintreffenden Daten entscheiden, was zu tun ist. Sind mehrere Instanzen des Servers erforderlich, um die hohe Last eines Softwaresystems auf mehrere Maschinen zu verteilen, ist dies ein großer Vorteil. Wenn in jedem Request alle relevanten Daten mitgeliefert werden, spielt es keine Rolle, auf welcher der Serverinstanzen ein Request bearbeitet wird. Schließlich liefert der Client mit jedem Request alle relevanten Informationen. Die Domänenlogik des Servers läuft dann ab, ohne dass sie selbst die Zuständigkeit hat, die Sessions der Benutzer zu verwalten.

Liegt der Sessionzustand dagegen in der Domänenlogik, muss dafür gesorgt werden, dass alle Requests eines Clients auf demselben Server landen, bei dem der erste Request des Clients eingetroffen ist. Alternativ

müsste der Sessionzustand zwischen den Serverinstanzen ausgetauscht werden. Die Skalierbarkeit einer Anwendung würde dadurch erschwert. Es bietet somit bei verteilten Anwendungen einen Vorteil, den Sessionzustand im Portal abzulegen. Auf diese Weise kann der Client mit jedem Request alle relevanten Informationen zum Server senden. Allerdings muss hierbei die Datenmenge berücksichtigt werden. Ist der Sessionzustand sehr umfangreich, ergibt es irgendwann wenig Sinn, diese große Datenmenge mit jeder Interaktion vom Client zum Server zu übertragen.

Zustand über das Beenden der Anwendung hinaus

Sollen Daten das Beenden einer Anwendung überdauern, müssen sie persistiert werden. Dazu eignen sich beispielsweise eine Datenbank oder Dateien im Dateisystem. Werden alle relevanten Daten spätestens beim Beenden der Anwendung dauerhaft gespeichert, können sie beim erneuten Start wieder eingelesen werden. So überdauern die Daten das Beenden der Anwendung und damit eine Session.

Die Daten liegen dann außerhalb des Systems in seiner Umwelt und werden im System-Umwelt-Diagramm als *Ressourcen* bezeichnet. Um die Wandelbarkeit eines Softwaresystems zu erhöhen, ist es wichtig, den Zugriff auf externe Ressourcen als einen eigenständigen Aspekt aufzufassen. Das ist Thema des folgenden Abschnitts.

Verwendung von Ressourcen

Externe Ressourcen werden immer dann benötigt, wenn das zu erstellende Softwaresystem auf Daten oder andere Ressourcen wie Hardware zugreifen muss. Daten können in vielfältiger Form benötigt werden. Ein typischer Anwendungsfall ist Zustand, der das Beenden der Anwendung überdauern soll. In diesem Fall muss der Zustand beim Beenden als externe Ressource gespeichert werden, um ihn beim Neustart wieder laden zu können. Aber auch während der Programmausführung werden vielfältige Ressourcen benötigt, von Dateien über Datenbanken zu Hardwareschnittstellen.

Ressourcenzugriffe

Ressourcenzugriffe sind etwas grundsätzlich anderes als Domänenlogik. Das gleiche gilt für die Benutzerschnittstelle. Der Zugriff einer Rolle auf das System ist ebenfalls ein eigenständiger Aspekt. Sei es ein Anwender, der das System mit diversen Eingabegeräten wie Tastatur, Maus, Mikrofon, Touch, etc. bedient oder ein Fremdsystem, für das eine HTTP/REST Schnittstelle bereitgestellt wird.

Schon aus dem System-Umwelt-Diagramm ergab sich die Forderung, Zugriffe auf Ressourcen mithilfe von *Providern* durchzuführen. Ferner erfolgen die Zugriffe der Rollen auf das System jeweils über ein *Portal*.

Ein Provider ist eine dünne Softwareschnittstelle, mit der die konkrete API einer Ressource eingekapselt wird. Dadurch bleibt der Kern des Systems frei von Abhängigkeiten zu dieser API.

Domänenlogik muss völlig frei sein von Ressourcenzugriffen. Ich wiederhole das nochmal, da es so zentral ist: **in der Domänenlogik haben jegliche Zugriffe auf Ressourcen nichts verloren**. Zum einen erleichtert dies die Testbarkeit, zum anderen ändern sich die Verhältnisse in der Umgebung des Systems erfahrungsgemäß häufiger. Heute werden die Daten in einer Datei gespeichert, morgen in einer Datenbank. Die Domänenlogik darf von solchen Veränderungen nicht betroffen sein. Stattdessen wird ein neuer Provider erstellt und eingebunden.

Die Testbarkeit der Domänenlogik ist ein sehr wichtiges Kriterium für die Korrektheit des Gesamtsystems. Kann die Domänenlogik nur schwer isoliert getestet werden, sind viele Integrationstests erforderlich. Diese zeichnen sich dadurch aus, dass sie eine Funktionseinheit nicht einzeln in Isolation testen, sondern die abhängigen Funktionseinheiten den Test ebenfalls durchlaufen. Ist also Domänenlogik von Ressourcenzugriffen abhängig, müssen entweder die Ressourcen für Integrationstests bereitgestellt werden, oder die Abhängigkeiten müssen im Test durch Attrappen ersetzt

werden. Beides ist aufwendig. Viel einfacher werden die Tests, wenn die Domänenlogik erst gar keine Abhängigkeiten zu Ressourcen hat. Dann kann sie leicht isoliert getestet werden, da sie ohne Abhängigkeiten ja bereits isoliert ist.

Bei der Betrachtung von Ressourcen ist es wichtig, auf den Inhalt, die Bedeutung der Ressource zu zielen. Sollen Kundendaten und Lieferantendaten persistiert werden, lautet die Ressource nicht "Datenbank" sondern "Kundendaten" und "Lieferantendaten". Es handelt sich also zum einen um zwei getrennte Ressourcen. Zum anderen ist damit nicht sogleich festgelegt, dass alle Daten mit derselben Technologie zu persistieren sind. Vielleicht werden am Ende Kunden- und Lieferantendaten in je eigenen Tabellen in derselben Datenbank abgelegt. Dies muss jedoch nicht von Anfang an festgelegt werden. Die Trennung der Provider bringt Flexibilität und somit bessere Wandelbarkeit. Und natürlich spricht nichts dagegen, innerhalb mehrerer Provider gemeinsamen Code wiederzuverwenden.

Neben Daten finden wir andere Ressourcen in der Umgebung eines Softwaresystems. Benötigt ein Softwaresystem Zugriff auf Hardware, die nicht typischer Bestandteil des Rechners ist, auf dem das System läuft, wird auch hierfür ein Provider benötigt. Soll zum Beispiel eine Maschine angesteuert werden, kommuniziert das System immer durch einen Provider mit der konkreten Hardware. Einerseits ist damit erneut die Domänenlogik von jeglichen Abhängigkeiten zur Hardwareressource befreit, andererseits bietet dieses Vorgehen die Möglichkeit, bei automatisierten Integrationstests mit Attrappen zu arbeiten.

Notation

Ressourcenzugriffe werden in Flow Design Entwürfen durch Dreiecke dargestellt. Dadurch erkennen wir im Entwurf auf einen Blick, dass es sich um einen fundamental anderen Aspekt handelt und werden Ressourcenzugriffe nicht in einen Topf werfen mit Domänenlogik. Dreiecke, Ellipsen und Rechtecke verwenden wir, um deutlich erkennbar zu machen, dass die Aspekte Ressourcenzugriffe, Domänenlogik und Benutzerinteraktion zu trennen sind.

Beim Zeichnen eines Entwurfs haben Dreiecke den Nachteil, dass sich die Bezeichnung der Funktionseinheit oft nur schwer in ein Dreieck schreiben lässt. Alternativ verwenden wir daher meist eine Ellipse mit einem Dreieck als Annotation.

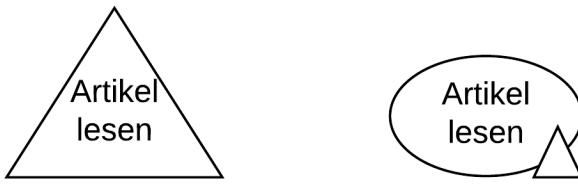


Abbildung 85: Dreieck vs. Ellipse mit Dreieck

Zuständigkeit

Auf Ressourcen wird über Provider zugegriffen, so viel ist nun klar. Doch müssen wir für die Provider immer wieder abwägen, welche Zuständigkeit noch im Bereich des Providers liegt und was in andere Funktionseinheiten verlagert wird. Verwendet der Provider für den Zugriff auf die Ressource eine spezielle API, ist auf jeden Fall darauf zu achten, dass Details dieser API nicht außerhalb des Providers sichtbar sind. Dies gilt vor allem für Datentypen, die von dieser API zur Verfügung gestellt werden. Soll bspw. durch einen Provider auf einen Webservice zugegriffen werden, dürfen die dabei verwendeten Typen wie WebClient nicht außerhalb des Providers sichtbar sein. Oder nehmen wir einen Provider für CSV Dateien. Auch dieser darf seine in der API definierten Datentypen wie CsvTable nicht nach außen reichen.

Es ist hilfreich, Provider in eigenen Projekten bzw. Packages abzulegen und die benötigten Frameworks nur in diesem Projekt bzw. Package zu referenzieren. So ist schon von der Sichtbarkeit her sichergestellt, dass die API nicht nach außen dringt, da sie dort aufgrund der fehlenden Sichtbarkeit syntaktisch nicht verstanden wird.

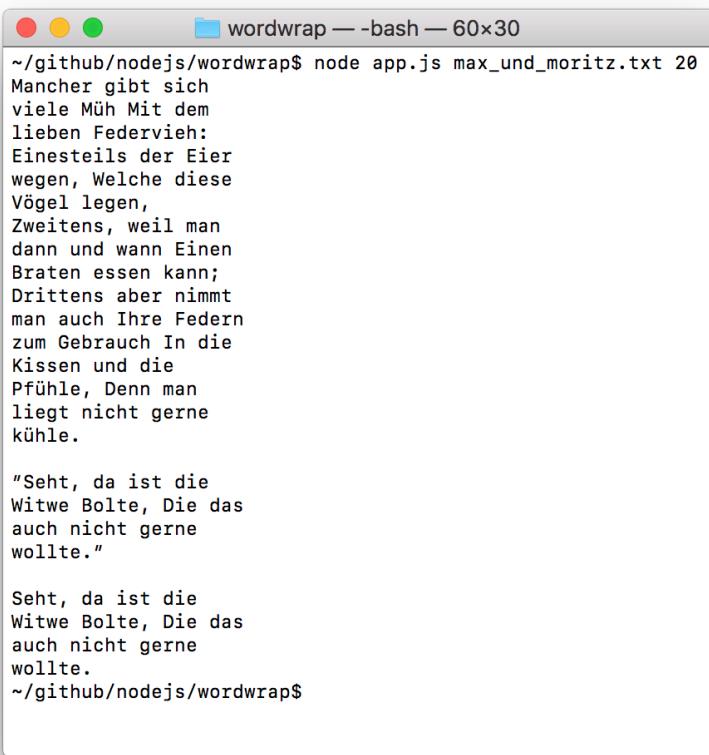
Eine weitere wichtige Regel für Provider ist, dass diese ihre interne Repräsentation der persistierten Daten nicht nach außen geben dürfen. Alle Funktionseinheiten, die Daten von einem Provider empfangen, erhalten diese Daten in einem Format, das nicht speziell zum Provider und seiner API gehört. Wenn wir zum Beispiel einen O/R-Mapper wie Hibernate, NHibernate oder Entity Framework für den Zugriff auf eine relationale Datenbank einsetzen, dürfen die Datentypen, welche auf Tabellen der Datenbank abgebildet sind, nicht außerhalb des Providers verwendet werden. Dadurch würde die interne Repräsentation der Persistenz nach außen sichtbar. So kann die Abbildung der Datenbank auf eine Tabellenstruktur nicht mehr verändert werden, ohne dass davon Funktionseinheiten in anderen Bereichen, insbesondere der Domänenlogik, betroffen sind. Folglich muss der Provider die interne Repräsentation abbilden auf eine externe Darstellung. Um diese unterschiedlichen Formen von Modellen geht

es im Kapitel Datenmodelle. Doch zuvor soll erneut ein Beispiel demonstrieren, wie mit Ressourcen im Entwurf und der Kodierung umgegangen wird.

Beispiel WordWrap

Anforderungen

Es ist eine Anwendung zu erstellen, mit der Textdateien neu umbrochen werden können. Die Anwendung ist als Konsolenanwendung zu erstellen. Beim Aufruf sind zwei Parameter anzugeben: der Dateiname sowie die Anzahl Zeichen pro Zeile. Die Abbildung zeigt, wie sich die fertige Anwendung im Terminal von mac OS verhält.



The screenshot shows a macOS terminal window titled "wordwrap — -bash — 60x30". The command entered is `~/github/nodejs/wordwrap$ node app.js max_und_moritz.txt 20`. The output displays a poem from "Max und Moritz" wrapped to 20 characters per line. The poem discusses the challenges of hatching eggs and the resulting mess. Below the poem, two additional stanzas are shown, followed by the command prompt again.

```
~/github/nodejs/wordwrap$ node app.js max_und_moritz.txt 20
Mancher gibt sich
viele Müh Mit dem
lieben Federvieh:
Einesteils der Eier
wegen, Welche diese
Vögel legen,
Zweitens, weil man
dann und wann Einen
Braten essen kann;
Drittens aber nimmt
man auch Ihre Federn
zum Gebrauch In die
Kissen und die
Pföhle, Denn man
liegt nicht gerne
kühle.

"Seht, da ist die
Witwe Bolte, Die das
auch nicht gerne
wollte."

Seht, da ist die
Witwe Bolte, Die das
auch nicht gerne
wollte.
~/github/nodejs/wordwrap$
```

Abbildung 86: Die fertige Anwendung im macOS Terminal

Hier handelt es sich um eine node.js Anwendung, geschrieben also in JavaScript. Der Aufruf erfolgt daher mit dem Befehl

```
node app.js
```

Alle weiteren Parameter werden an die Anwendung übergeben, in diesem Fall `max_und_moritz.txt` als Dateiname und `20` als neue Zeilenbreite. Das Programm liest die Textdatei ein und gibt sie neu formatiert auf der Konsole aus. Die Zeilenlänge wird dabei auf die angegebene Anzahl Zeichen begrenzt. Die Absätze des Textes bleiben dabei erhalten. Absätze werden dadurch gebildet, dass eine Leerzeile im Text eingefügt wird. Alle anderen Zeilenschaltungen, die der Trennung der Zeilen dienen, werden neu berechnet, so dass die Zeilen maximal so lang sind, wie angegeben.

Sollte ein einzelnes Wort breiter sein als die maximal Zeilenbreite, wird dieses Wort vollständig in einer Zeile ausgegeben und überragt dann die Zeilenbreite.

Dialoge und Interaktionen

In diesem Beispiel einer Konsolenanwendung gibt es lediglich eine einzige Interaktion: der Anwender startet das Programm.

```

wordwrap — bash — 60x30
~/github/nodejs/wordwrap$ node app.js max_und_moritz.txt 20
Mancher gibt sich
viele Müh Mit dem
lieben Federvieh:
Einesteils der Eier
wegen, Welche diese
Vögel legen,
Zweitens, weil man
dann und wann Einen
Braten essen kann;
Drittens aber nimmt
man auch Ihre Federn
zum Gebrauch In die
Kissen und die
Pföhle, Denn man
liegt nicht gerne
kühle.

"Seht, da ist die
Witwe Bolte, Die das
auch nicht gerne
wollte."

Seht, da ist die
Witwe Bolte, Die das
auch nicht gerne
wollte.
~/github/nodejs/wordwrap$
```

Abbildung 87: Interaktionsdiagramm für WordWrap

Entwurf der obersten Ebene

Der Entwurf der obersten Ebene geht leicht von der Hand. Vom Betriebssystem werden die Kommandozeilenargumente arg^* an das Programm geliefert. Das Programm produziert daraufhin die Ausgabe und übergibt sie an eine Konsolen UI.



Abbildung 88: Entwurf der obersten Ebene

Verfeinern des Entwurfs

Die Verfeinerung des Entwurfs ist erforderlich, da die Funktionseinheit WordWrap ohne Verfeinerung für mehr als einen Aspekt zuständig wäre. Bereits ohne detaillierte Analyse liegt es auf der Hand, dass es einen Ressourcenzugriff geben wird, um die Textdatei einzulesen. Ferner müssen die Kommandozeilenargumente berücksichtigen werden. Und ein wenig Domänenlogik wird es wohl auch geben. Ohne Verfeinerung geht es also nicht.

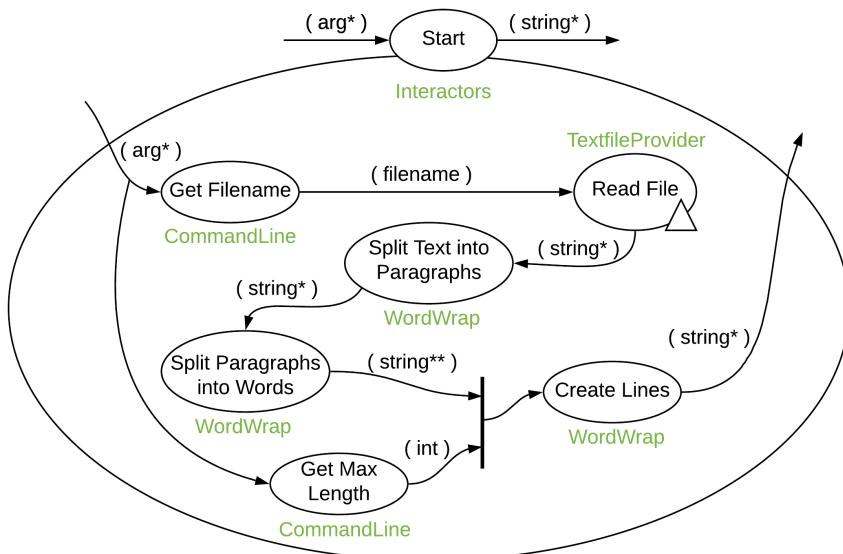


Abbildung 89: Verfeinerung des Entwurfs *WordWrap*

Zunächst wird aus den übergebenen Kommandozeilenargumenten der Dateiname ermittelt. Anschließend kann die Datei eingelesen und in Paragraphen getrennt werden. Danach werden die einzelnen Paragraphen dann jeweils in Wörter zerlegt. Um nun aus den Wörtern eines Paragraphen den neu umbrochenen Paragraphen zu erstellen, wird die neue Zeilenbreite benötigt. Diese wird ebenfalls aus den Kommandozeilenargumenten extrahiert. Zuletzt können dann die neuen Zeilen gebildet werden.

Umsetzung mit node.js

Der Einstiegspunkt in die Anwendung liegt in der Datei `app.js`. Die Anwendung wird auf der Konsole mit dem Befehl

```
node app.js
```

gestartet. Das Modul `app.js` ist dafür zuständig, den Datenfluss auf der obersten Ebene herzustellen. Es integriert die Module `integration.js` und `consoleui.js`.

```
const integration = require("./integration.js");
const ui = require("./consoleui.js");

var args = process.argv;

var lines = integration.WordWrap(args);
ui.ShowResult(lines);
```

Zunächst werden die beiden Module `integration.js` und `consoleui.js` eingebunden. Anschließend wird die Funktion `WordWrap` mit den Kommandozeilenparametern aufgerufen und das Ergebnis an die Methode `ShowResult` der `Ui` übergeben.

Die nächste Ebene der Anwendung befindet sich im Modul `integration.js`. Hier wird der weiter oben gezeigte Flow realisiert.

```
const commandline = require("./commandline.js");
const textfileprovider = require("./textfileprovider.js");
const wordwrap = require("./wordwrap.js");

exports.WordWrap = function(args) {
    var filename = commandline.GetFilename(args);
    var lines = textfileprovider.ReadFile(filename);
    var paragraphs = wordwrap.SplitTextIntoParagraphs(lines);
    var words = wordwrap.SplitParagraphsIntoWords(paragraphs);
    var length = commandline.GetMaxLength(args);
    var newLines = wordwrap.CreateLines(words, length);

    return newLines;
};
```

Wieder werden zunächst die benötigten Module eingebunden. Anschließend werden die einzelnen Funktionen aufgerufen gemäß dem Entwurf.

Die unterste Ebene bilden dann die einzelnen Operationen. Da wäre zunächst das Modul `commandline.js`, zuständig für das Ermitteln der benötigten Werte aus den Kommandozeilenparametern.

```
exports.GetFilename = function(args) {
    return args[2]
};

exports.GetMaxLength = function(args) {
    return args[3]
};
```

Als nächstes ist das Modul `textfileprovider.js` dafür zuständig, die Textdatei einzulesen.

```
const fs = require('fs');

exports.ReadFile = function(filename) {
    return fs.readFileSync(filename).toString().split("\n")
};
```

Zuletzt ist dann das Modul `wordwrap.js` für die Domänenlogik zuständig.

```
exports.SplitTextIntoParagraphs = function(lines) {
    let paragraph = "";
    let paragraphs = [];
    lines.forEach(line => {
        if(line !== "") {
            if(paragraph === "") {
                paragraph = line;
            } else {
                paragraph += " " + line;
            }
        } else if(paragraph !== "") {
            paragraphs.push(paragraph);
            paragraph = "";
        }
    });
    if(paragraph !== "") {
        paragraphs.push(paragraph);
    }
    return paragraphs;
};
```

```
exports.SplitParagraphsIntoWords = function(
  paragraphs) {
  let words = [];
  paragraphs.forEach(paragraph => {
    let wordsOfParagraph = [];
    paragraph.split(" ")
      .filter(w => w !== "")
      .forEach(word =>
        wordsOfParagraph.push(word));
    words.push(wordsOfParagraph);
  });
  return words;
};
```

```

exports.CreateLines = function(paragraphs, length) {
    let lines = [];
    let line = "";

    paragraphs.forEach(paragraph => {
        if(lines.length > 0) {
            lines.push("");
        }
        paragraph.forEach(word => {
            if(line === "") {
                line = word;
            } else if((line + word).length + 1 <= length) {
                line += " " + word;
            } else {
                lines.push(line);
                line = word;
            }
        });
        if(line !== "") {
            lines.push(line);
            line = "";
        }
    });

    return lines;
};

```

Durch den Entwurf gelingt eine Umsetzung, in der die einzelnen Aspekte der Anwendung klar getrennt in Modulen abgelegt sind. Diesen Zusammenhang kann man nicht oft genug betonen. Erst durch die Arbeit an einem Entwurf mit grafischen Mitteln entsteht eine Struktur, die rein auf der Ebene von Quelltext in der Syntax einer Programmiersprache nicht entstehen würde. Selbst ein konsequentes nachträgliches Refactoring stößt an eine Grenze, die nur dadurch Durchbrochen werden kann, dass bereits vor dem Codieren über die Lösung nachgedacht wird.

Neben der klaren Trennung der Aspekte entsteht durch den Entwurf eine Trennung von Integration und Operation. Die einzelnen Module lassen sich damit leicht testen. Enthält ein Modul Operationen, lassen diese sich leicht

in Form von Unit Tests automatisiert testen. Da die Operationen keine Abhangigkeiten haben, sind es per se bereits Units.

Methoden, die fur die Integration anderer Methoden verantwortlich sind, enthalten keine Domanenlogik. Aus diesem Grund ergibt es keinen Sinn, solche Module von ihren Abhangigkeiten zu befreien und Attrappen einzusetzen. Integrationsmethoden werden durch automatisierte Integrationstests getestet. Somit ergibt sich aus dem Entwurf auch ein groer Vorteil fur die Testbarkeit des Softwaresystems. Die Aspekte Integration und Operation werden weiter unten im Kapitel uber das IOSP nochmals aufgegriffen.

Datenmodelle

Viewmodel, Domainmodel, Datamodel

In einem Softwaresystem werden verschiedene Arten von Datenmodellen verwendet. Im Laufe der Zeit haben sich zahlreiche Begriffe etabliert, die allerdings nicht immer eindeutig mit der gleichen Bedeutung verwendet werden. Daher soll der folgende Abschnitt eine klare Bedeutung der Begriffe zeigen, so wie sie hier im Buch verwendet werden.

Moderne Technologien zur Realisierung von Benutzerschnittstellen verwenden häufig *Data Binding*, um Daten in der Benutzerschnittstelle anzuzeigen bzw. von dort entgegenzunehmen. In diesem Kontext werden Daten in *Viewmodels* abgelegt. Ein Viewmodel enthält die Daten in einer Art und Weise, wie sie für die Visualisierung im zugehörigen *View* optimal geeignet ist. Das Viewmodel dient damit dem *View*. Bei .NET müssen Viewmodels zur Signalisieren von Änderungen das Interface *INotifyPropertyChanged* implementieren, um auf diese Weise Änderungen an Eigenschaften mitzuteilen. Das Interface erfordert die Einführung eines Events, der ausgelöst wird, wenn eine Eigenschaft des Viewmodels geändert wurde. Auf diese Weise können sich Controls im *View* an diesen Event binden und werden dann benachrichtigt, wenn geänderte Daten angezeigt werden müssen. Dieser Mechanismus ist einzig in der Benutzerschnittstelle erforderlich. Aspekte wie die Domänenlogik oder die Persistenz interessieren sich nicht für diesen Mechanismus. Je nach Implementation kann er dort sogar hinderlich sein, sofern die Events zu eifrig auslösren. Die Implementation dieses speziellen Interface ist also den Datenmodellen vorbehalten, die innerhalb der Benutzerschnittstelle verwendet werden. Datenmodelle die anderen Aspekten dienen, werden andere Anforderungen stellen.

Neben solchen technischen Anforderungen an Viewmodels werden die Daten im Viewmodel inhaltlich und strukturell so abgelegt, dass die *View* die Daten leicht anzeigen kann. Der Zweck der Viewmodels liegt ja gerade darin, Daten für die Visualisierung und Benutzerinteraktion aufzubereiten. Ein Währungsbetrag könnte im Viewmodel als *string* abgelegt werden, fertig formatiert inklusive Rundung und Währungssymbol. Für die Visualisierung im *View* mag das optimal sein, für die Domänenlogik wäre das ungünstig. Dort müsste der *string* interpretiert werden, um den Betrag als *decimal* zu ermitteln. Nur so könnten damit Berechnungen durchgeführt werden. Wie soll das Datenmodell nun modelliert werden? Als *string* für die *View* oder *decimal* für die Domain? Statt sich hier für eine Variante zu entscheiden, wird den beiden Aspekten jeweils ein eigenes Datenmodell zur Verfügung

gestellt. So kann auf unterschiedliche Anforderungen der einzelnen Aspekte leicht eingegangen werden. Ferner werden so die Abhängigkeiten zwischen den Aspekten reduziert. Auf der anderen Seite muss eine Übertragung vom einen in das andere Modell stattfinden.

Stellt man eine Anwendung in Schichten dar, steht üblicherweise am unteren Ende der Ressourcenzugriff, oft in Form von Datenbanken. Auch hier kommen häufig Datenmodelle zum Einsatz, etwa um Datenstrukturen auf Datenbanktabellen abzubilden. Diese Modelle nennen wir *Persistence-model*.

Je nach Domäne kommen auch im Bereich der Domäne eigene Modelle zum Einsatz. Dies bietet sich bspw. an, wenn auf diese Weise Algorithmen leichter implementiert werden können. Im Bereich der Domäne sprechen wir von einem *Domainmodel*. Es bildet die Domänenkenntnisse in einem Modell ab.

Die folgende Abbildung zeigt, in welcher Weise die wesentlichen Aspekte einer Anwendung von den unterschiedlichen Datenmodell abhängig sind.

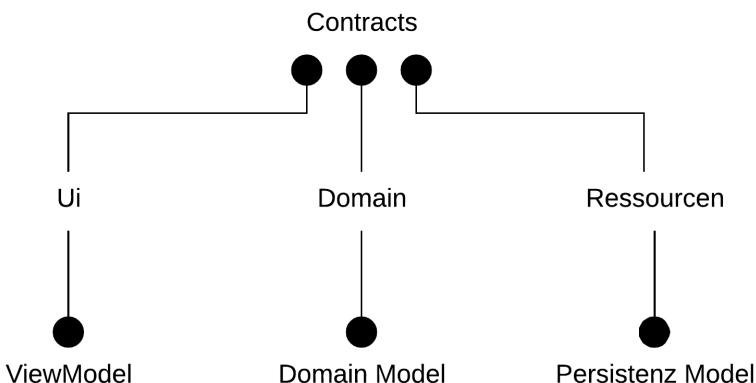


Abbildung 90: Datenmodelle und ihre Abhängigkeiten

Damit nun Daten von einem Aspekt zum anderen Aspekt übertragen werden können, wird ein weiteres Datenmodell benötigt. Dieses *Contracts* genannte Modell hat die Aufgabe, einen Datenaustausch zwischen den Aspekten Benutzerschnittstelle, Domäne und Ressourcenzugriff zu ermöglichen. Damit die einzelnen Funktionseinheiten nicht gezwungen sind, auf primitive Datentypen wie *string*, *int* oder *bool* zurückzufallen, werden im Bereich der Contracts Datenklassen geschaffen, mit denen ein Austausch strukturierter Daten möglich wird.

Als Konsequenz ergibt sich daraus die Notwendigkeit, zwischen den unterschiedlichen Modellen zu *mappen*. Schauen wir uns das an einem einfachen Beispiel an. In der Benutzerschnittstelle gibt der Benutzer Daten

ein, die von der Domänenlogik zu verarbeiten sind. Zuletzt sollen die Daten in einer Datenbank abgelegt werden.

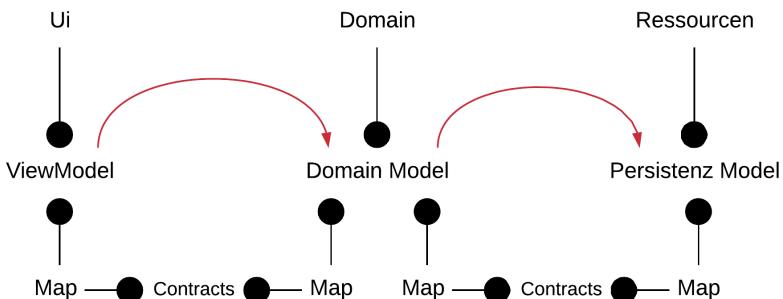


Abbildung 91: Mapping vom Viewmodel zum Domainmodel und weiter zum Persistencemodel

Auf der anderen Seite können auch Daten aus der Datenbank gelesen und direkt zur Benutzerschnittstelle übertragen werden. Es ist nicht notwendig, dass dazwischen in ein Domänenmodell gemappt wird. Wir haben es hier nicht mit einem Schichtenmodell zu tun, in dem nur zur jeweiligen direkten Nachbarschicht Kontakt aufgenommen werden darf. Es darum, Aspekte zu trennen und Abhängigkeiten zu reduzieren.

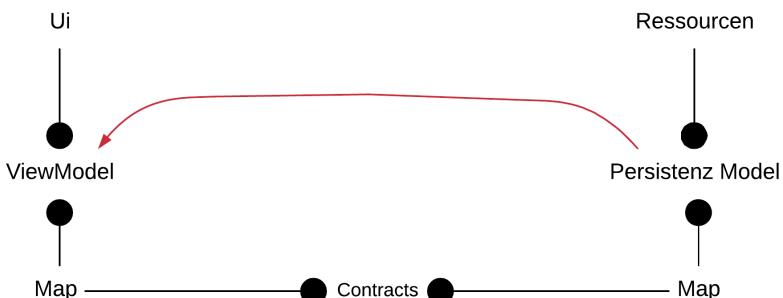


Abbildung 92: Mapping vom Datenbankzugriff zur Ui

Der Vorteil, der sich aus der Trennung der Datenmodelle ergibt, liegt in einer besseren Wandelbarkeit. Wenn das Datenbankschema verändert werden soll, hat dies lediglich Auswirkungen auf das Persistencemodell. Durch das Mapping vom Persistencemodell in das Contractsmodel ergibt sich ein definierter Übergang. Hier können Änderungen am Persistencemodell so gekapselt werden, dass sie sich nicht auf die anderen Aspekte auswirken.

Nun wird häufig eingewandt, dass die Mappings doch einen höheren Aufwand bedeuten. Ferner würden die Mappings auch dazu führen, dass die Anwendung weniger performant ist, da die Daten dauernd hin und her kopiert würden.

Zum höheren Aufwand ist zu sagen, dass dabei meist übersehen wird, dass ein Softwareentwicklungsprozess grob gesagt aus zwei Phasen besteht: dem Schreiben des Codes und dem Lesen bzw. Ändern des Codes. Meist wird mit dem vorgenannten Argument, das Erstellen der zusätzlichen Modelle und der Mappings sei zeitaufwendig, auf die Phase des Schreibens optimiert. Es ist allerdings so, dass die nachfolgende Phase des Lesens, Verstehens und Ändern des Codes länger dauert als das Schreiben. Insofern sollten alle Tätigkeiten daraufhin optimiert sein, dass die Wandelbarkeit des Codes möglichst hoch ist, um diese längere dauernde Phase optimal zu unterstützen. Ferner müssen die Mappings beim Einsatz entsprechender Bibliotheken nicht vollständig handkodiert werden. Durch Tools wie AutoMapper⁶ für .NET oder ModelMapper⁷ für Java wird der Aufwand deutlich reduziert. Ferner stellen die Mappings trivialen Code dar, der zügig erstellt und modifiziert werden kann.

Das Argument der Laufzeitperformance ist in wenigen Systemen tatsächlich ein wichtiger Einwand. In den meisten Systemen spielt es allerdings keine Rolle, ob die Daten transformiert werden oder nicht. Zugunsten der Wandelbarkeit sollte auch hier ein kaum messbarer Nachteil der Laufzeitperformance in Kauf genommen werden. Und bevor die Aufteilung in mehrere Datenmodell über Bord geworfen wird, sollte unbedingt mit entsprechenden Profilingwerkzeugen überprüft werden, ob tatsächlich das Kopieren der Daten beim Mapping das Problem verursacht. Oft liegen die Probleme nur gefühlt in diesem Bereich und erweisen sich bei näherer Betrachtung als unkritisch. Bei Performanceproblemen sollte daher in jedem Fall versucht werden, einen Zwischenweg zu finden, bei dem Wandelbarkeit und Laufzeitperformance in einem günstigen Verhältnis stehen.

⁶ <http://automapper.org>

⁷ <http://modelmapper.org>

Abhangigkeiten

Ui, Domanenlogik, Ressourcen

In vielen Anwendung findet wahrend einer Interaktion des Benutzers ein typischer Fluss von Daten statt:

- Der Benutzer gibt Daten in der Ui ein.
- Von dort flieen die Daten zur Domanenlogik, wo sie validiert, berechnet, angereichert werden.
- Zuletzt werden die Daten in der Datenbank persistiert.

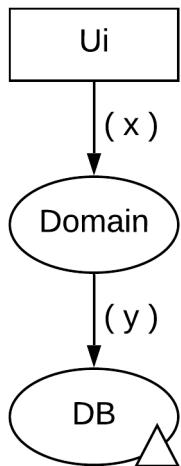


Abbildung 93: Typischer Datenfluss durch eine Anwendung

Fur die grobe Struktur einer solchen Anwendung werden in der Praxis immer wieder eine ganze Reihe Abhangigkeiten in Kauf genommen. Es scheint auf den ersten Blick unausweichlich, dass die Ui eine Abhangigkeit zur Domanen haben muss. Schlielich muss die Ui doch die Domanen aufrufen, um ihr die Eingabedaten zu ubergeben. Ebenso unausweichlich scheint die Abhangigkeit von der Domanen zur Datenbank. Auch hier scheint es notwendig, dass die Domanen die Datenbank "kennt" um ihr die Daten ubergeben zu konnen. Es entsteht folglich eine Abhangigkeitsstruktur, wie sie in der folgenden Abbildung zu sehen ist.

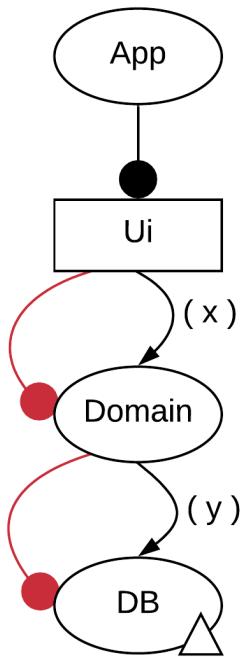


Abbildung 94: Typische Abhängigkeiten in einer Desktopanwendung

Die Abbildung zeigt, wie die Abhängigkeiten in typischen Desktopanwendungen meist gestaltet werden. Aufgabe der *App* ist es, die benötigten Objekte zu erzeugen. Sie instanziert bspw. die *Ui* und übergibt dann die Kontrolle an das Hauptfenster, damit der Benutzer mit der Anwendung interagieren kann. Sobald der Benutzer Daten eingegeben und eine Schaltfläche betätigt hat, müssen diese Daten von der Domänenlogik weiterverarbeitet werden. Hier erfolgt zum Beispiel eine Validierung oder Transformation der vom Benutzer eingegebenen Daten. Und weil die *Ui* Daten an die Domänenlogik liefern muss, erscheint es schon fast unausweichlich, dass die *Ui* die Domänenlogik aufrufen muss. Damit hat die *Ui* eine Abhängigkeit zur *Domain*.

Genauso unausweichlich erscheint die nächste Abhängigkeit. Da die Daten nach der Bearbeitung durch die Domänenlogik in die Datenbank geschrieben werden sollen, muss die Domänenlogik von der Datenbank abhängig sein. Wie sollte sie schließlich sonst auf die Datenbank zugreifen können? Noch deutlicher wird es, wenn die Domänenlogik für ihre Arbeit Daten aus der Datenbank benötigt. Auch in diesem Fall liegt es nahe, dass die Domänenlogik die Datenbank aufruft, um an die benötigten Daten zu

gelangen. Am Ende ergibt sich eine Kette von Abhängigkeiten, wie sie in der oben gezeigten Abbildung dargestellt ist.

Damit die Software dann trotz der Abhängigkeiten automatisiert getestet werden kann, werden zusätzlich Interfaces eingeführt. Die Ui ist damit nicht mehr direkt von der Domain abhängig, sondern sie ist von einem Interface abhängig, welches die Domain implementiert. Auf diese Weise wird es möglich, Abhängigkeiten im Test durch Attrappen zu ersetzen. Dabei kommen Mock Frameworks wie Moq⁸, Mockito⁹ oder SinonJS¹⁰ zum Einsatz. Im Test werden die Attrappen angewiesen, sich auf eine bestimmte Weise zu verhalten, damit das gewünschte Resultat automatisiert überprüft werden kann. So wird bspw. die Datenbank angewiesen, bestimmte Daten zu liefern, die von der Domain angefordert werden. Am Ende kann dann in der Domain geprüft werden, ob das gewünschte Ergebnis vorliegt, ohne dass dabei die reale Datenbank zum Einsatz kam. Ferner kann durch die Interfaces und Mock Frameworks überprüft werden, ob eine Attrappe auf eine bestimmte Art und Weise aufgerufen wurde. So könnte ein Test überprüfen, ob die Daten nach der Validierung in die Datenbank geschrieben werden. Dazu wird die Datenbankattrappe nach Ausführung der zu testenden Methode befragt, ob sie auf die erwartete Weise aufgerufen wurde.

Der Einsatz von Interfaces und Attrappen ist für viele Entwickler völlig selbstverständlich. Und dennoch muss man festhalten, dass diese Vorgehensweise einen höheren Aufwand bedeutet. Eine Funktion zu testen, die aus einer Eingabe eine Ausgabe produziert, ist deutlich einfacher. Stellt man die Struktur einer Anwendung nur leicht um, entfällt der zusätzliche Aufwand, der durch die Abhängigkeiten und den Einsatz von Interfaces und Attrappen entsteht.

Bevor wir zu einer besseren Struktur kommen, zunächst noch ein Blick auf Webanwendungen. Bei Webanwendungen sieht die Struktur leicht anders aus. Das liegt daran, dass bei Webanwendungen die Ui in der Regel durch die Infrastruktur instanziert wird. Web Frameworks wie ASP. NET MVC, Spring, o.ä. arbeiten so, dass *Routen* definiert werden, die jeweils eine URL auf eine Ui abbilden. Es wird bspw. ausgedrückt, dass die URL “/login” auf den Dialog “login.html” abgebildet wird. Zur Laufzeit wird die Ui durch das verwendete Web Framework instanziert. Ein Request wird vom Framework empfangen und von diesem auf eine Ui geroutet. Folglich instanziiert das Framework die über die Route identifizierte Ui. Und damit die Ui “den Rest” der Anwendung verwenden kann, wird über Dependency Injection alles Nötige in die Ui reingereicht. Auch bei Webanwendungen ist also am Ende eine Struktur vorzufinden, in der die Ui die Domäne aufruft und diese wiederum die Datenbank direkt verwendet.

⁸ <https://github.com/moq/moq4>

⁹ <http://site.mockito.org>

¹⁰ <http://sinonjs.org>

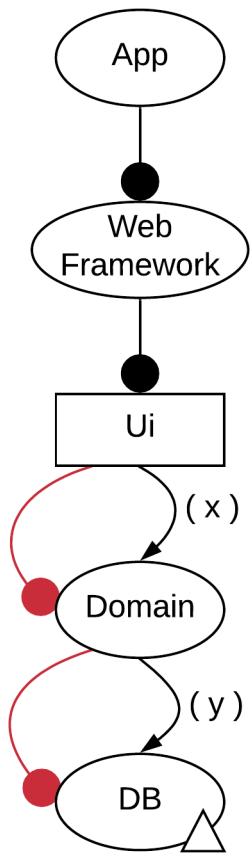
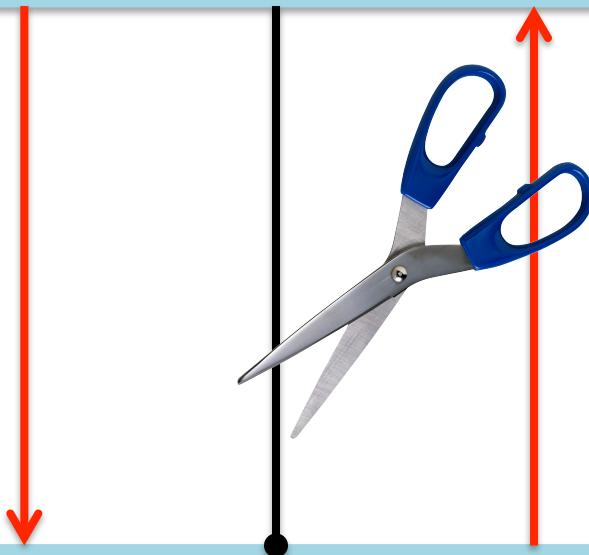


Abbildung 95: Typische Abhängigkeiten in einer Webanwendung

Ob Desktop oder Web: diese Struktur von Abhängigkeiten hat vor allem im Bereich der Domäne einen wesentlichen Nachteil: Die Domäne ist von Ressourcenzugriffen abhängig. Auch wenn die Ressourcen hinter einem Provider gekapselt sind und trotz Einsatz von Interfaces und Dependency Injection: es bleibt eine Abhängigkeit von der Domäne zu diesen Interfaces. In der Folge müssen für automatisierte Tests Attrappen in die Domänenlogik gereicht werden, um die Logik losgelöst von den realen Ressourcenzugriffen testen zu können.

Domänenlogik



DB

Abbildung 96: Freistellen der Domäne von der Abhängigkeit zur Datenbank

Im Kern werden hier zwei Aspekte vermischt: *Integration* und *Operation* sind zusammengefasst. Die Domänenlogik ist bereits für einen wesentlichen Teil des Systems zuständig: die Logik der Domäne. Durch die Abhängigkeit vom Ressourcenzugriff wird sie zusätzlich verantwortlich gemacht für die *Integration* der Ressourcenzugriffe. Und das, obwohl die Alternative so einfach und naheliegend ist. Für Desktopanwendungen zeigt die folgende Abbildung eine bessere Struktur.

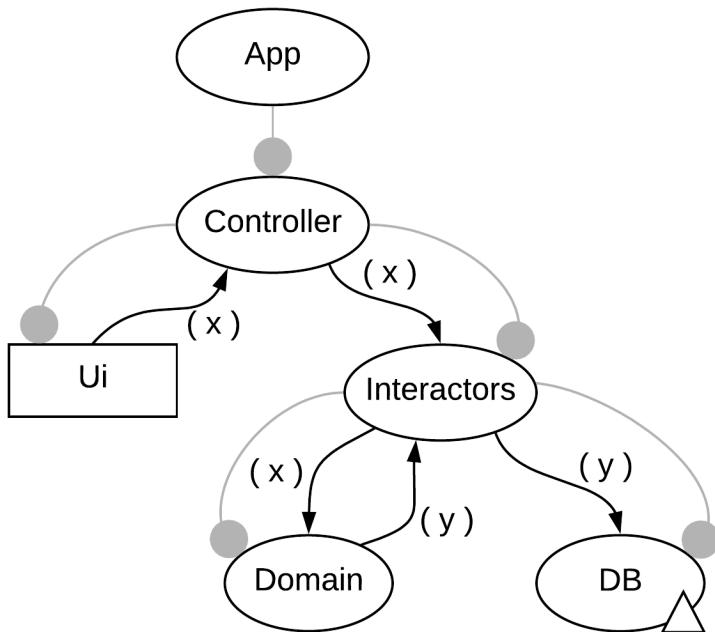


Abbildung 97: Empfohlene Struktur für Desktopanwendungen

Die *App* ist nach wie vor dafür zuständig, alle benötigten Funktionseinheiten zu instanzieren, Abhängigkeiten aufzulösen und zuletzt die Kontrolle an die Benutzerschnittstelle zu übergeben. Der Kernaspekt der App ist ihre Verantwortlichkeit für den Einstiegspunkt in die gesamte Anwendung. Hier liegt die *Main* Methode. Auf der nächsten Ebene liegt nun der *Controller*. Er ist dafür zuständig, die weiteren Bestandteile zu integrieren. Der Aspekt der Integration, der zuvor noch in der *Ui* und der Domäne vorhanden war, ist in Teilen hierhin verlagert worden. Der *Controller* verbindet die *Ui* mit ihrem *Interactor*. Der *Interactor* stellt die Beziehung her zwischen Domänenlogik und Ressourcenzugriffen. Hier werden die einzelnen Operationen so integriert, dass das gewünschte Verhalten für die einzelnen Interaktionen hergestellt wird.

Damit sind die drei Aspekte *Ui*, Domäne und Ressourcenzugriff freigestellt von jeglichen Abhängigkeiten. Das Zusammenwirken der drei Aspekte erfolgt durch den *Controller* und den *Interactor*.

Die Benutzereingaben fließen von der *Ui* nun allerdings nicht durch direkten oder indirekten Aufruf zur Domäne, sondern werden dem *Controller* gemeldet. Dieser liefert die Daten dann weiter an den *Interactor*. Damit hat die *Ui* keine Kenntnis und vor allem keine Abhängigkeit mehr von der Domäne und den Ressourcenzugriffen. Das gleiche gilt für die Verbindung

zwischen Domain und Datenbank. Die Domain erhält vom Interactor die Daten und liefert ihr Ergebnis zurück an den Interactor. Erst der Interactor leitet die Daten dann an die Datenbank weiter. Dadurch ist die Domain vom Ressourcenzugriff befreit.

Es liegen nun zwei Stellen vor, an denen integriert wird: Controller und Interactor. Dies hat seinen Grund darin, dass auf diese Weise Integrationstests möglich sind, die keine Abhängigkeit zur UI haben. Das gesamte Verhalten der einzelnen Interaktionen wird jeweils vom Interactor hergestellt. Die Methoden des Interactors erwarten die Benutzereingaben als Parameter und liefern ggf. Ergebnisse zurück über ihren Rückgabewert. Ferner lösen sie ggf. Seiteneffekte aus, wie das Speichern in einer Datenbank. Integrationstests für die Methoden des Interactors zu schreiben ist damit kein Problem. Im Einzelfall können Ressourcenzugriffe durch Interfaces, Dependency Injection und Attrappen im Test ersetzt werden,

Für Webanwendungen sieht die Struktur etwas anders aus, weil hier nach wie vor das Web Framework das Instanziieren der UI übernimmt. Es wäre technisch möglich, in diesen Instanziierungsmechanismus einzugreifen, um UI, Domäne und Ressourcen miteinander zu verbinden. Allerdings wäre das aufwändiger, als die hier gezeigte Variante. Wir machen uns hier also weiterhin die Dependency Injection zunutze, die in den Web Frameworks zur Verfügung gestellt wird. Allerdings wird nun nicht die Domäne in die UI injiziert, sondern der *Interactor*. Auch hier ist der Interactor für die Integration der Aspekte Domäne und Ressourcenzugriffe zuständig. Der wesentliche Unterschied zur gängigen Praxis ist auch hier erreicht: die Domäne hat keine Abhängigkeiten mehr.

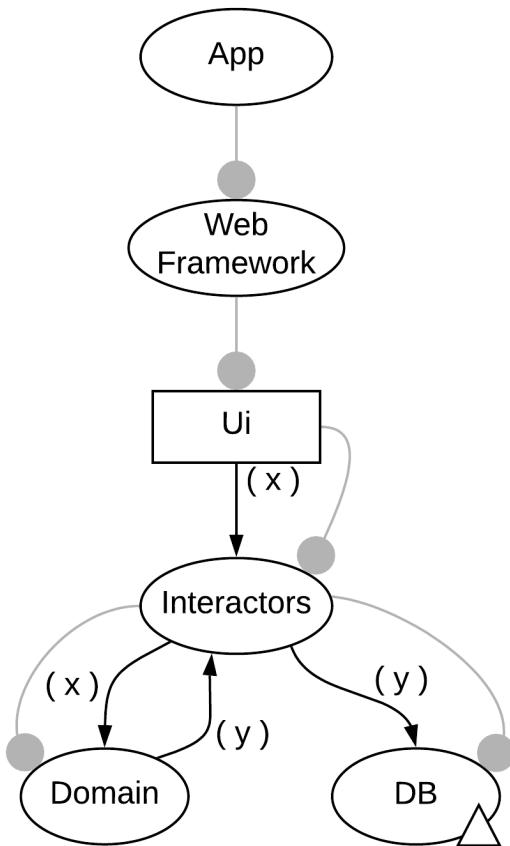


Abbildung 98: Empfohlene Struktur für Webanwendungen

In der Webanwendung fließen die Daten von der UI zum Interactor. Dieser Datenfluss wird wie zuvor durch eine direkte Abhängigkeit realisiert. Die UI erhält den Interactor injiziert und kann ihn somit direkt aufrufen. Im Gegensatz zur landläufigen Umsetzung bezieht sich diese Abhängigkeit allerdings auf eine dünne Integrationsschicht, hier *Interactor* genannt. Die UI kommuniziert nicht direkt mit der Domäne. Das ist Aufgabe des Interactors. Wie bei der Desktopvariante erläutert, ist der Interactor für die Integration von Domäne und Ressourcenzugriffen zuständig. Dadurch ist der Integrationsaspekt aus der Domäne herausgelöst worden. Die Domäne hat keine Abhängigkeit zu Ressourcenzugriffen. Somit ist die Testbarkeit deutlich vereinfacht und es kann auf den Einsatz von Interfaces und Mock Frameworks verzichtet werden.

Beispiel für eine Desktopanwendung

Im folgenden wird anhand der Beispielanwendung “My Books” demonstriert, wie die Struktur einer Desktopanwendung aussieht, wenn die Abhängigkeiten aus den Operationen herausgelöst werden. Die Anwendung dient dazu, eine Liste von Büchern zu verwalten. Stellen Sie sich dazu vor, Sie hätten ein riesiges Bücherregal. Gerne verleihen Sie Bücher an Ihre Freunde. Um den Überblick darüber zu behalten, wem Sie ein Buch geliehen hatten, wünschen Sie sich eine kleine Anwendung. In dieser können Bücher hinzugefügt und gelöscht werden. Ferner kann zu einem Buch vermerkt werden, an wen es ausgeliehen wurde. Umgekehrt kann ein verliehenes Buch als zurückerhalten markiert werden.

The screenshot shows a Windows-style application window titled "My Books". The main area is a table with three columns: "Title", "Lender", and "Lended at". The data rows are:

Title	Lender	Lended at
Flow Design		
Clean Code	Paul	11.06.2018
Refactoring Legacy Code		
C# for Beginners		
JavaScript Secrets		

At the bottom of the window is a toolbar with four buttons: "New book", "Lend" (which is highlighted with a dashed border), "Get back", and "Remove".

Abbildung 99: My Books in Aktion

Die Anwendung verwendet zum Speichern der Bücher einen *Event Store*. Das bedeutet, es werden nicht die Daten der Bücher gespeichert, sondern die Ereignisse zu den Büchern. So wird bspw. gespeichert, dass ein neues Buch mit dem Titel “Flow Design” angelegt wurde. Liest man alle Ereignisse ein und interpretiert sie, kann jeweils der aktuelle Stand der Buchverwaltung ermittelt werden. Diese Spezialität der Anwendung ist für die folgende Betrachtung der Abhängigkeiten nicht weiter relevant, jedoch für das Verständnis des Quellcodes hilfreich.

Beginnen wir beim Einstiegspunkt in die Anwendung, der Datei `Program.cs`. Die Verantwortung der `Main` Methode ist es, die benötigten

Funktionseinheiten zu instanziieren. Ferner werden hier die Datenflüsse der obersten Ebene der Anwendung hergestellt. Für die Interaktion *Start* wird zunächst die Funktionseinheit *Start* aus der Klasse *Interactors* aufgerufen. Diese liefert eine Liste von Büchern, die an die Ui übergeben werden. Dazu wird die Methode *Update_books* aufgerufen. Die anderen Interaktionen beginnen jeweils in der Ui. Erfasst der Anwender bspw. ein neues Buch, löst die Ui den Event *New_book* aus. Dieser Event trägt den Titel des Buches als Parameter. Der Titel wird an die Methode *New_book* der Klasse *Interactors* übergeben. Die Methode liefert alle Bücher als Rückgabewert und diese werden wiederum an die Ui zur Anzeige übergeben.

```

public static class Program
{
    [STAThread]
    public static void Main() {
        var mainWindow = new MainWindow();
        var interactors = new Interactors();
        var app = new Application { MainWindow = mainWindow };

        void start() {
            var books = interactors.Start();
            mainWindow.Update_books(books);
        }

        mainWindow.New_book += title => {
            var books = interactors.New_book(title);
            mainWindow.Update_books(books);
        };
        mainWindow.Lend_book += (id, name) => {
            var books = interactors.Lend_book(id, name);
            mainWindow.Update_books(books);
        };
        mainWindow.Book_got_back += (id) => {
            var books = interactors.Book_got_back(id);
            mainWindow.Update_books(books);
        };
        mainWindow.Remove_book += (id) => {
            var books = interactors.Remove_book(id);
            mainWindow.Update_books(books);
        };

        start();
        app.Run(mainWindow);
    }
}

```

Die Klasse *Program* ist somit dafür zuständig, die Verbindung zwischen der *Ui* und der eigentlichen Anwendung herzustellen. In den Ausführungen weiter oben wurde dazu ein *Controller* verwendet. In diesem einfachen Beispiel ist die *Main* Methode sowohl Einstiegspunkt als auch dafür zuständig, die Verbindungen zwischen *Ui* und *Interactor* herzustellen. In größeren Anwendungen wird dieser Aspekt rausgelöst und in einem *Controller* implementiert.

Die Anwendung selbst ist in Form der einzelnen Interaktionen in der Klasse *Interactors* realisiert. Auf dieser Ebene der Anwendung wird es nun interessant. Die Klasse *Interactors* ist dafür zuständig, die einzelnen Interaktionen zu realisieren. Dies ist die Ebene, auf der eine Integration von Domänenlogik und Ressourcenzugriff stattfindet. Für die Interaktionen *Start* und *New Book* sieht das wie folgt aus:

```
public class Interactors
{
    private readonly EventStoreProvider _eventStoreProvider =
        new EventStoreProvider();
    private readonly Booklending _booklending =
        new Booklending();

    public IEnumerable<Book> Start() {
        var events = _eventStoreProvider.Read_all_events();
        var books = _booklending.Create_list_of_books(events);
        return books;
    }

    public IEnumerable<Book> New_book(string titel) {
        var bookCreatedEvent = _booklending.Create_book(titel);
        _eventStoreProvider.Save_event(bookCreatedEvent);
        var events = _eventStoreProvider.Read_all_events();
        var books = _booklending.Create_list_of_books(events);
        return books;
    }
}
```

Die Interaktion *Start* wird hier folgendermaßen realisiert: zunächst wird der *EventStoreProvider* angewiesen, alle Events zu liefern. Anschließend werden diese Events an die Domänenlogik übergeben. Aufgabe der Domänenlogik ist es, aus den Events die aktuelle Liste der Bücher herzustellen. Dazu interpretiert die Domänenlogik alle Ereignisse und stellt so die Bücherliste zusammen. Das folgende Beispiel soll den Unterschied zu einer klassischen Datenbank darstellen. Nehmen wir an, folgende Events liegen im Event Store:

Correlation Id	Timestamp	Event	Data
1	2018-01-10	New book	Title="Flow Design"
1	2018-02-15	Lended	Name="Jakob"

1	2018-02-30	Returned	
1	2018-03-01	Lended	Name="Paul"

Die Events werden über eine *Correlation Id* zusammengefasst. Alle Events mit der selben Correlation Id beziehen sich auf das selbe Objekt, in diesem Fall ein Buch. Um die Events in die richtige Reihenfolge bringen zu können, erhalten sie einen Zeitstempel, hier der Einfachheit halber lediglich ein Datum. Ferner wird zu jedem Event verzeichnet, welches Ereignis eingetreten ist. Optional können zu einem Ereignis zusätzliche Daten gespeichert werden, die das Ereignis näher beschreiben. Das erste Ereignis besagt, dass ein Buch mit dem Titel "Flow Design" angelegt wurde. Im Anschluss wurde das Buch an "Jakob" ausgeliehen, dann von diesem zurückgegeben und anschließend an "Paul" ausgeliehen. In einer klassischen Datenbank würde für diese vier Ereignisse der endgültige Zustand des Buches abgelegt:

Id	Title	Lended At	Lender
12345	Flow Design	2018-03-01	Paul

Über die Vorgeschichte wäre in der Datenbank nichts verzeichnet. Insbesondere könnte man nicht erkennen, wann das Buch angelegt wurde bzw. an wen es ggf. vorher bereits ausgeliehen war. Natürlich kann man ein Datenbankmodell schaffen, dass diese Informationen ebenfalls aufnehmen kann. Jedoch entsteht eine solche Historie bei Verwendung von Event Stores ganz natürlich. Gleichzeitig kann durch das Abspielen aller Events der endgültige Zustand ebenfalls hergestellt werden. Soviel für den Moment zum Thema Event Store. Für die Darstellung der Abhängigkeiten ist es nicht weiter relevant, ob die Persistenz der Daten über einen Event Store oder eine klassische relationale Datenbank realisiert wird.

Für die Interaktion *New book* sieht der Vorgang ähnlich aus wie bei *Start*. Zunächst wird die Domänenlogik angewiesen, zu dem vom Benutzer eingegebenen Buchtitel eine neues Buch anzulegen. Anschließend wird dieser Datensatz an den *EventStoreProvider* übergeben, damit dieser einen neuen Event erzeugt und persistiert. Um zuletzt wieder die Liste aller Bücher zu liefern, werden wie schon bei *Start* alle Events gelesen und interpretiert.

Klar erkennbar ist hier, dass die Domänenlogik keine Abhängigkeit zum Ressourcenzugriff hat. Nicht die Domänenlogik weist den Event Store an, die Daten zu speichern, sondern um diesen Aspekt der Integration kümmert sich die Klasse *Interactors*. Und auch die UI ist unabhängig. Sie ruft weder die Domänenlogik noch die Integration direkt auf. Stattdessen löst sie Events aus. Erst durch die Verbindung dieser Events mit Methoden wird der Datenfluss von der UI zur Klasse *Interactors* hergestellt. Die Abhängigkeit der UI zur Klasse *Interactors* wäre allerdings nicht weiter tragisch. Wir

werden weiter unten sehen, dass dies bei Web Anwendungen sogar der Normalfall ist. Der zentrale Punkt ist die Klasse *Interactors*, da hier die Domänenlogik mit den Ressourcenzugriffen integriert wird. Somit kann die Domänenlogik leicht automatisiert getestet werden. Die Methode *Create_book* erhält einen Buchtitel und erstellt daraus ein *Book* Objekt. Dies ist natürlich eine sehr einfache Domänenlogik. Doch es wird deutlich, dass diese Methode leicht zu testen ist, da sie keinen Seiteneffekt in einer Ressource hervorruft, sondern ihr Ergebnis als Rückgabewert liefert. Würde *Create_book* nach dem Erstellen des Buches den Event Store selbst aufrufen, müsste im Test geprüft werden, ob im Event Store das richtige Ereignis gespeichert wurde. Alternativ würde man eine Attrappe des Event Stores in die Domänenlogik reinreichen und auf dieser dann prüfen, ob die Event Store Methode zum speichern korrekt aufgerufen wurde. Viel einfacher wird der Test dagegen bei der hier dargestellten Trennung von Integration und Operation.

Beispiel für eine Webanwendung

Im Folgenden wird die Anwendung zur Verwaltung von Büchern als Webanwendung dargestellt. Um nicht auf eine spezielle UI Technologie zu setzen, wird ein REST Service gezeigt. Die Interaktionen *Start*, *Neues Buch*, *Buch ausleihen*, *Buch zurückgeben* und *Buch löschen*, sind jeweils über eine URL und das zugehörige Verb erreichbar. Auf diese Basis kann dann eine beliebige Benutzerschnittstelle aufsetzen.

Die Anwendung ist mit dem Framework NancyFX¹¹ realisiert. Bei NancyFX müssen alle Routen über sogenannte *Modules* definiert werden. Ein Modul zeichnet sich dadurch aus, dass es von der Klasse *NancyModule* ableitet. Nancy sammelt beim Start der Anwendung alle Klassen auf, die von *NancyModule* ableiten und instanziiert diese. Auf diese Weise werden per Konvention die Routen definiert. Das folgende Listing zeigt das Modul, in dem die Routen für die "My Books" Anwendung eingerichtet werden.

¹¹ <http://nancyfx.org>

```

public class BooksApiModule : NancyModule
{
    public BooksApiModule(Interactors interactors) {
        Get["/books"] = _ => {
            var books = interactors.Start();
            return Response.AsJson(books);
        };

        Put["/books?title={title}"] = parms => {
            string title = parms.title;
            var books = interactors.New_book(title);
            return Response.AsJson(books);
        };

        Delete["/books?id={id}"] = parms => {
            string id = parms.id;
            var books = interactors.Remove_book(new Guid(id));
            return Response.AsJson(books);
        };

        Put["/books/{id}/lend?name={name}"] = parms => {
            string id = parms.id;
            string name = parms.name;
            var books = interactors.Lend_book(new Guid(id), name);
            return Response.AsJson(books);
        };

        Put["/books/{id}/back"] = parms => {
            string id = parms.id;
            var books = interactors.Book_got_back(new Guid(id));
            return Response.AsJson(books);
        };
    }
}

```

Die Interaktionen sind hier in die folgenden Routen übersetzt worden:

- GET /books Start
- PUT /books Neues Buch
- DELETE /books Buch löschen
- PUT /books/{id}/lend Buch ausleihen
- PUT /books/{id}/back Buch zurückgeben

Je nach Route werden unterschiedliche Parameter benötigt. Aufgabe des Moduls ist es lediglich, diese Details so zu definieren, wie es aus technischer Sicht beim Einsatz von NancyFX erforderlich ist. Das Modul selbst darf keine Domänenlogik enthalten. Es ist bereits für den Aspekt API bzw.

Routing zuständig. Sind durch eine Route die Operation und die erforderlichen Parameter bestimmt, erfolgt die eigentliche Ausführung der Interaktion wiederum durch Methoden der Klasse *Interactors*. Damit das Modul darauf zugreifen kann, erhält der Konstruktor eine Instanz der Klasse *Interactors* übergeben. Hier ist es Aufgabe von NancyFX dafür zu sorgen, dass zur Laufzeit eine Instanz erzeugt und reingereicht wird. Da es sich hier um Infrastrukturcode handelt, der nicht isoliert getestet werden soll, bleibt es bei der unmittelbaren Abhängigkeit des Moduls von der Klasse *Interactors*. Eine Ebene weiter in der Klasse *Interactors* sind ebenfalls direkte Abhängigkeiten vorhanden. Die Klasse *Interactors* ist für die Integration der anderen Bestandteile verantwortlich. Die Abhängigkeiten sind gerade der wesentliche Aspekt dieser Klasse. Das folgende Listing zeigt erneut einen Ausschnitt der Klasse *Interactors*, die sich in keiner Weise von der weiter oben bereits gezeigten Klasse unterscheidet. Es kommt hier somit der selbe Code zum Einsatz wie bereits bei der Desktopanwendung.

```
public class Interactors
{
    private readonly IEventStoreProvider _eventStoreProvider;
    private readonly Booklending _booklending = new Booklending();

    public Interactors(IEventStoreProvider eventStoreProvider) {
        _eventStoreProvider = eventStoreProvider;
    }

    public IEnumerable<Book> Lend_book(Guid id, string name) {
        var bookLendedEvent = _booklending.Lend_book(id, name);
        _eventStoreProvider.Save_event(bookLendedEvent);
        var events = _eventStoreProvider.Read_all_events();
        var books = _booklending.Create_list_of_books(events);
        return books;
    }
}
```

Auch in der Webanwendung ist damit erreicht, dass die Domäne freigestellt ist von Ressourcenzugriffen. Es ist die Aufgabe der Klasse *Interactors*, die unterschiedlichen Aspekte zu integrieren. Klar erkennbar ist das am Code der Methode *Lend_book* aus der Klasse *Booklending*. Die Klasse ist für die Domänenlogik des Bücherverleihs zuständig.

```
public LendedEvent Lend_book(Guid id, string lenderName) {
    return new LendedEvent {
        CorrelationId = id,
        Lender = lenderName,
        Timestamp = DateTime.Today,
        LendingDate = DateTime.Today
    };
}
```

Die Methode erzeugt lediglich einen Event. Das Speichern dieses Events im Event Store beauftragt die Domänenlogik nicht selbst. Stattdessen erfolgt dieser Aufruf eine Ebene darüber in der Klasse *Interactors*, die für die Integration der Aspekte zuständig ist.

Zerlegung der UI

Data Binding und MVVM

In den bislang gezeigten Entwürfen ist die Benutzerschnittstelle lediglich lapidar als Rechteck mit der Beschriftung *Ui* dargestellt. In realen Projekten stellt sich allerdings die Frage, wie die innere Struktur der Benutzerschnittstelle aussehen kann. Die Frage nach der inneren Struktur der *Ui* mag sich bei einem sehr einfachen Dialog noch nicht stellen. Doch spätestens, wenn ein Dialog aufwendiger wird, muss darüber nachgedacht werden, wie die innere Struktur gestaltet werden soll. Nichts anderes tun wir bei den anderen Aspekten eines Softwaresystems: ständig stehen wir vor der Frage, ob die Aspekte klar getrennt sind und ob die Funktionseinheiten ausreichend klein sind. Das ist einer der Gründe dafür, dass wir Domänenlogik entwerfen. Hier dürfen wir immer davon ausgehen, dass ein Entwurf erforderlich ist, sich eine Struktur lohnt. Bei der *Ui* ist es nicht anders: wir müssen regelmäßig darüber nachdenken, ob die Aspekte klar getrennt sind und ob mehr Struktur die Wandelbarkeit erleichtern würde. Des Weiteren ist gerade im Bereich der Benutzerschnittstelle darauf zu achten, dass die Testbarkeit gegeben ist. Automatisierte Tests werden aufwendig, sobald dabei die Technologie der Benutzerschnittstelle innerhalb der Tests eine Rolle spielt.

Im Bereich der Benutzerschnittstelle gibt es viele Technologien und Frameworks. Diese alle hier zu behandeln, würde den Umfang des Buches deutlich überschreiten. Es gibt jedoch ein Konzept, welches in vielen der Frameworks zum Einsatz kommt: *Data Binding*. Im Kern geht es bei Data Binding darum, die darzustellenden Daten zu trennen von ihrer Darstellung. Dabei kommen ein *Viewmodel* und ein *View* zum Einsatz. Das Viewmodel enthält die darzustellenden Daten, der View die Controls zur Darstellung und Bearbeitung dieser Daten.

Das Viewmodel dient dem View. Daher gibt der View vor, wie das Viewmodel gestaltet sein soll. Aus diesem Grund sind die Daten im Viewmodel so abgelegt, dass der View sie ohne weitere Deutung darstellen kann. Eine Deutung oder Interpretation durch den View soll nicht mehr erfolgen, weil dies häufig domänen spezifische Deutungen wären. Sollen bspw. numerische Werte, die in einem bestimmten Bereich liegen, farblich anders dargestellt werden, sollte diese Deutung nicht von der *Ui* vorgenommen werden. Stattdessen muss das Viewmodel, neben den Werten selbst, auch die anzugebende Farbe enthalten, oder zumindest einen Wert, der ohne weitere Interpretation auf eine Farbe gemappt werden kann. Auf diese

Weise ist sichergestellt, dass die Entscheidung an anderer Stelle getroffen wird, nämlich in einer Klasse, die der Domänenlogik zugeordnet ist. So wird das automatisierte Testen dieses aus Domänensicht definierten Verhaltens immer leichter sein, wenn die Ui nicht beteiligt ist.

Das derzeit am häufigsten verwendete Muster ist das *MVVM* Muster. MVVM steht für *Model View Viewmodel*. Zu beachten ist, dass es sich um ein Muster zur Strukturierung der Ui handelt. Der *Model* Anteil repräsentiert in diesem Muster den gesamten Rest der Anwendung. Das Muster macht keinerlei Aussage darüber, wie der *Model* Anteil weiter strukturiert werden sollte. Insofern ist MVVM eine Ergänzung zum restlichen Entwurf unserer Anwendung. Oder um es anders auszudrücken: der *Model* Anteil wird mittels Flow Design entworfen und strukturiert. Eine Anwendung hat somit niemals eine “MVVM Architektur”, sondern verwendet allenfalls das MVVM Pattern innerhalb der Ui.

In der Ui enthalten sind somit die beiden Anteile *View* und *Viewmodel*. Der *View* repräsentiert den visuellen Anteil, während das *Viewmodel* die darzustellenden Daten enthält. Der *View* bezieht die Daten für die einzelnen Controls aus dem *Viewmodel*. Über das Data Binding wird definiert, welche Eigenschaften des *Viewmodels* mit den Eigenschaften der Controls verbunden werden sollen.

Die im folgenden dargestellten Entwürfe zeigen für die Ui zwei unterschiedliche Datenflüsse. Im einen Fall fließen Daten in die Ui, im anderen fließen sie aus der Ui. Schauen wir uns an, wie beide Szenarien realisiert werden können.

Datenfluss in die Ui

Während es bei den meisten Funktionseinheiten eines Entwurfs möglich und sinnvoll ist, sie als Methoden zu realisieren, ist dies bei der Ui nicht praktikabel. Die Ui ist aus Sicht des Entwurfs ebenfalls eine Funktionseinheit. Sie wird allerdings mindestens in Form einer Klasse realisiert. Bei Anwendung des MVVM Musters kommen sogar mehrere Klassen zum Einsatz. Dass die Ui mindestens eine Klasse sein sollte, liegt vor allem daran, dass sie typischerweise über mehrere Eingänge und Ausgänge verfügt.

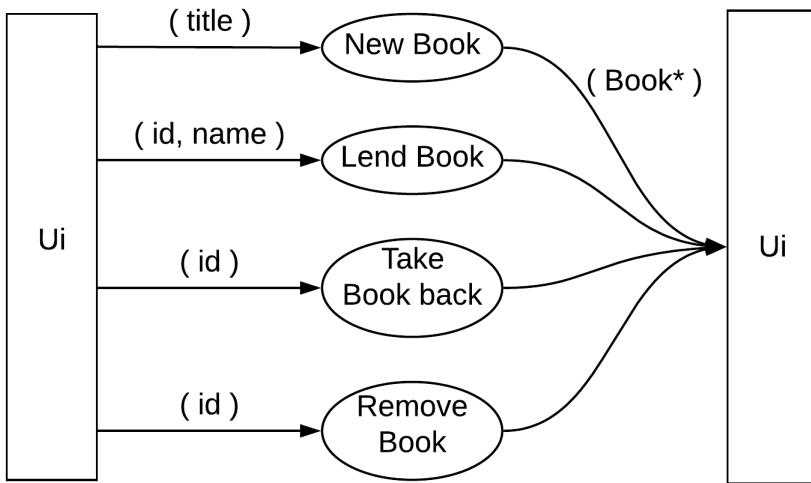


Abbildung 100: Interaktionen eines Dialogs

In der Abbildung sehen wir die oberste Ebene des Entwurfs für einen typischen Dialog mit seinen Interaktionen. Die einzelnen Interaktionen beginnen jeweils in der Ui und enden nach Durchlaufen des Interactors wieder in ihr. Im Entwurf starten daher die einzelnen Datenflüsse in der Ui und enden wieder in ihr. Somit hat die Ui fast immer mehrere Ausgänge sowie häufig mehrere Eingänge. Zwar zeichnen wir die Ui sonst für jede Interaktion getrennt auf, doch ändert das nichts daran, dass jeder ausgehende Datenfluss der Ui zur selben Funktionseinheit gehört. Damit scheidet die Umsetzung der Ui als Methode aus. Für einfache Dialoge kann die Ui durchaus aus nur einer einzigen Klasse bestehen. Bei Verwendung des MVVM Pattern sind es mindestens zwei Klassen: die View und das Viewmodel.

Bei der Umsetzung von Funktionseinheiten durch Klassen werden die Eingänge als Methoden und die Ausgänge als Events realisiert. Das folgende Listing zeigt das Grundgerüst der Ui für das oben dargestellte Beispiel.

```
public partial class MainWindow : Window
{
    public void Update_books(IEnumerable<Book> books) {

    }

    public event Action<string> New_book;

    public event Action<Guid, string> Lend_book;

    public event Action<Guid> Book_got_back;

    public event Action<Guid> Remove_book;
}
```

Nun können wir die Funktionseinheit Ui verfeinern und uns die innere Struktur genauer anschauen. Bei Anwendung des MVVM Musters kommen innerhalb der Ui ein *View* und ein *Viewmodel* zum Vorschein. Der View ist abhängig vom Viewmodel, da er das Viewmodel beim Data Binding verwendet, sich an Eigenschaften des Viewmodels bindet. Der eingehende Fluss der Daten muss somit innerhalb der Ui im Viewmodel eintreffen. Die Daten fließen in das Viewmodel und werden durch den Data Binding Mechanismus in der View angezeigt.

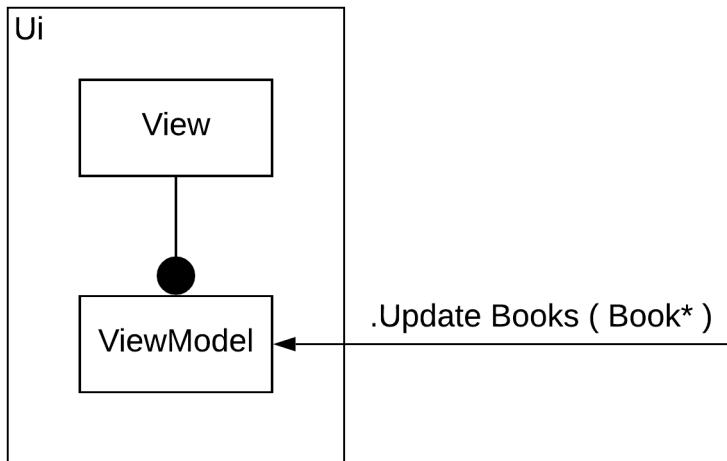


Abbildung 101: Die Beziehung zwischen View und Viewmodel

Konkret sind nun zwei mögliche Umsetzungen denkbar:

- Das *Viewmodel* enthält eine Methode, über die die Daten in das Viewmodel fließen. Die Übertragung der Daten in die Eigenschaften des Viewmodels findet innerhalb des Viewmodels statt.
- In der zweiten Variante wird das Übertragen der Daten in das Viewmodel vom Viewmodel selbst getrennt. Eine gesonderte *Map* Methode ist für die Übertragung zuständig.

Die erste Variante sieht im Quellcode wie folgt aus:

```

public void Update_books(IEnumerable<Book> books) {
    var items = new BookEntries();
    items.Update(books);
    dataGrid.DataContext = items;
}

```

Das Viewmodel für die Bücherliste ist hier vom Typ *BookEntries*. Im Viewmodel ist die Methode *Update* für das Mapping von den Contracts Daten (Typ *IEnumerable<Book>*) in das Viewmodel (in sich selbst) zuständig. Folgendes Listing zeigt das Viewmodel.

```

public class BookEntries : List<BookEntry>
{
    public void Update(IEnumerable<Book> books) {
        var items = books.Select(book => new BookEntry
        {
            Id = book.CorrelationId,
            Title = book.Title,
            Lender = book.Lender,
            LendingDate = book.LendingDate
        });
        Clear();
        AddRange(items);
    }
}

```

Zieht man das Mapping der übergebenen Daten auf die Eigenschaften des Viewmodels aus diesem heraus, entsteht folgender Code im View:

```

public void Update_books(IEnumerable<Book> books) {
    var items = Map(books);
    dataGridView.DataContext = items;
}

private List<BookEntry> Map(IEnumerable<Book> books) {
    var items = books.Select(book => new BookEntry
    {
        Id = book.CorrelationId,
        Title = book.Title,
        Lender = book.Lender,
        LendingDate = book.LendingDate
    });
    return items.ToList();
}

```

Als Entwurf sieht dies wie folgt aus:

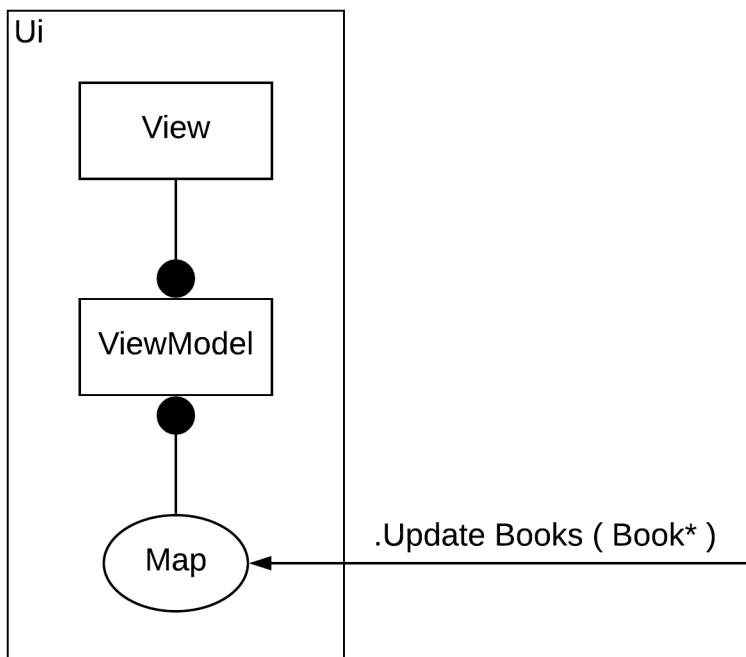


Abbildung 102: Die Beziehung zwischen View und Viewmodel mit expliziter Map Methode

Datenfluss aus der Ui

Bei der Umsetzung einer Funktionseinheit durch eine Klasse werden ausgehende Datenflüsse als *Event* implementiert. In manchen Programmiersprachen, wie etwa C#, werden Events unmittelbar in der Sprache angeboten. Bei Verwendung von Sprachen ohne explizite Event Syntax kommen Funktionszeiger oder Interfaces zum Einsatz. Konzeptionell geht es darum, ein Ereignis mit einer Reaktion darauf zu verbinden. In einer Klasse, hier konkret der *Ui*, wird ein Ereignis festgestellt. Mithilfe von Events soll nun erreicht werden, dass die Reaktion auf das Ereignis nicht durch dieselbe Klasse aufgerufen wird. Das folgende Listing zeigt, wie die Verbindung zweier Klassen über Events und Methoden in C# aussieht.

```

[STAThread]
public static void Main() {
    var mainWindow = new MainWindow();
    var interactors = new Interactors();
    var app = new Application {
        MainWindow = mainWindow
    };

    void start() {
        var books = interactors.Start();
        mainWindow.Update_books(books);
    }

    mainWindow.New_book += titel => {
        var books = interactors.New_book(titel);
        mainWindow.Update_books(books);
    };
    mainWindow.Lend_book += (id, name) => {
        var books = interactors.Lend_book(id, name);
        mainWindow.Update_books(books);
    };
    mainWindow.Book_got_back += (id) => {
        var books = interactors.Book_got_back(id);
        mainWindow.Update_books(books);
    };
    mainWindow.Remove_book += (id) => {
        var books = interactors.Remove_book(id);
        mainWindow.Update_books(books);
    };

    start();
    app.Run(mainWindow);
}

```

Die Zeile

```
mainWindow.New_book += title => { ... };
```

bedeutet, dass eine Lambda Expression an den Event `New_book` gebunden wird. Eine Lambda Expression ist eine nicht benannte Methode. Der Event-mechanismus führt dazu, dass der View nun den Event `New_book` auslösen kann. Dazu ruft der View den Event wie eine Methode auf. Letztlich ist diese Indirektion vergleichbar mit einem Funktionszeiger.

Eine Alternative zu Events sind *Callbacks*. Die Verbindung zwischen Ui und dem Interactor kann auch so hergestellt werden, dass der Ui für die verschiedenen Ereignisse jeweils eine Callback Methode übergeben wird. Auf diese Weise kann die Ui die Callbacks aufrufen, ohne sie konkret zu kennen. Es existiert damit also auch hier keine direkte Abhängigkeit von der Ui zum Interactor. Die Aufgabe der Integration von Ui und Interactor wird von einer gesonderten Klasse übernommen, hier von der Klasse *Program*, die gleichzeitig auch den Einstiegspunkt in die Anwendung darstellt. Auf diese Weise werden die Prinzipien IOSP und PoMO eingehalten. IOSP steht für *Integration Operation Segregation Principle*. Das Prinzip besagt, dass Integration und Operation zu trennen sind. Da die Ui bereits Logik für die Benutzerschnittstelle enthält, somit also eine Operation ist, soll sie nicht zusätzlich für die Integration der Domänenlogik zuständig sein. Die Abkürzung PoMO steht für *Principle of Mutual Oblivion*, die *gegenseitig Nichtbeachtung*. Dieses Prinzip besagt, dass zwei Funktionseinheiten, zwischen denen ein Datenfluss stattfindet, sich gegenseitig nicht beachten sollen. Insbesondere sollen sie keine Abhängigkeiten haben, sich also nicht gegenseitig aufrufen. Stattdessen ist eine weitere Funktionseinheit dafür zuständig, den Datenfluss herzustellen. Weitere Einzelheiten zu den Prinzipien IOSP und PoMO finden Sie im Kapitel über Prinzipien.

Eine direkte Abhängigkeit zwischen Ui und Domänenlogik kann auch durch Callbacks verhindert werden. Das Ziel lautet jeweils, dass die Ui die Domänenlogik nicht direkt aufrufen sollte. Stehen syntaktisch keine Events zur Verfügung, können Callbacks verwendet werden. Im folgenden Listing wird eine Ui in Java angedeutet. Im Konstruktor werden die benötigten Callbacks übergeben. Somit kann die Klasse nur instanziert werden, wenn die benötigten Callbacks übergeben werden. Alternativ könnte man Setter-Methoden für die Callbacks vorsehen. In dem Fall wäre die API der Klasse allerdings schwieriger zu verstehen, da es nicht genügt, eine Instanz lediglich mit dem Konstruktor zu erzeugen, man müsste schließlich auch die Callbacks zuweisen. Über Konstruktorparameter ist diese Fehlbedienung nicht möglich.

Innerhalb der Ui werden die Callbacks aufgerufen wie exemplarisch in der Methode `OnNewBook` demonstriert.

```

import java.util.function.BiConsumer;
import java.util.function.Consumer;

public class Ui {
    private final Consumer<String> onNewBook;
    private final BiConsumer<Integer, String> onLend;
    private final Consumer<Integer> onBack;
    private final Consumer<Integer> onDelete;

    public Ui(
        Consumer<String> onNewBook,
        BiConsumer<Integer, String> onLend,
        Consumer<Integer> onBack,
        Consumer<Integer> onDelete) {
        this.onNewBook = onNewBook;
        this.onLend = onLend;
        this.onBack = onBack;
        thisonDelete = onDelete;
    }

    public void OnNewBook() {
        String title = "..."; // Comes from text input
        onNewBook.accept(title);
    }
}

```

Ein Datenfluss aus der Ui heraus kann also mit Events oder Callbacks umgesetzt werden. Kommt das MVVM Muster zum Einsatz, liegen die Daten, die fließen sollen, im Viewmodel. Der Ablauf ist wie folgt:

- Der Benutzer gibt seine Werte im View in die einzelnen Controls ein.
- Die Controls sind mittels Data Binding mit dem Viewmodel verbunden.
- Das verwendete Framework sorgt durch das Data Binding dafür, dass die vom Benutzer eingegebenen Daten aus den Controls in das Viewmodel übertragen werden.
- Irgendwann wird dann vom Benutzer die Interaktion ausgelöst, zum Beispiel dadurch, dass er eine Schaltfläche betätigt.

Diese Interaktion soll nun dazu führen, dass Daten aus der Ui heraus zum Interactor fließen.

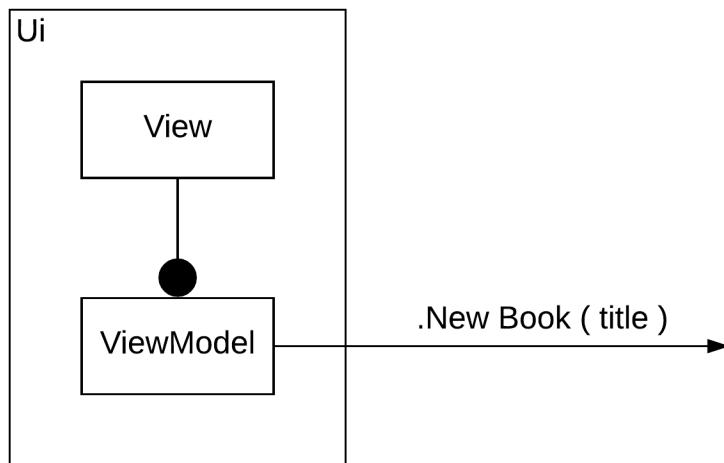


Abbildung 103: Die innere Struktur der Ui, das Viewmodel hält den Event

Wie schon bei den eingehenden Datenflüssen können nun auch bei den ausgehenden Datenflüssen unterschiedliche Formen der Implementation angewandt werden. Eine Variante ist, dass das Viewmodel entsprechende Events zur Verfügung stellt.

Alternativ kann auch die View die Events oder Callbacks bereitstellen. Da die View ohnehin vom Viewmodel abhängig ist, kann die View auf die Daten des Viewmodels zugreifen und diese mit dem Event oder Callback an den Nachfolger im Datenfluss weiterreichen.

```
public partial class MainWindow : Window
{
    public event Action<string> New_book;
    public event Action<Guid, string> Lend_book;
    public event Action<Guid> Book_got_back;
    public event Action<Guid> Remove_book;
```

```
public MainWindow() {
    InitializeComponent();
    btnNew.Click += (o, e) => {
        var title = InputBox.Show("Title:");
        if (!string.IsNullOrEmpty(title)) {
            New_book(title);
        }
    };
    btnLend.Click += (o, e) => {
        var id = Id_of_selected_book();
        if (!id.HasValue) {
            return;
        }
        var name = InputBox.Show("Name:");
        if (!string.IsNullOrEmpty(name)) {
            Lend_book(id.Value, name);
        }
    };
    btnGetBack.Click += (o, e) => {
        var id = Id_of_selected_book();
        if (!id.HasValue) {
            return;
        }
        Book_got_back(id.Value);
    };
    btnRemove.Click += (o, e) => {
        var id = Id_of_selected_book();
        if (!id.HasValue) {
            return;
        }
        Remove_book(id.Value);
    };
}
```

```
public void Update_books(IEnumerable<Book> books) {
    var items = new BookEntries();
    items.Update(books);
    dataGrid.DataContext = items;
}

private Guid? Id_of_selected_book() {
    var item = dataGrid.SelectedItem as BookEntry;
    return item?.Id;
}
}
```

Im Entwurf sieht das dann wie folgt aus:

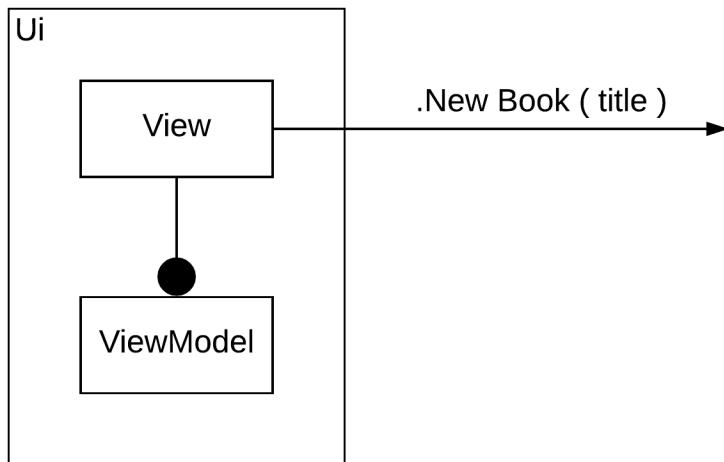


Abbildung 104: Die innere Struktur der Ui, der View hält den Event

Beispiel Questionnaire

Bis hier wurde ein Großteil der Syntax von Flow Design eingeführt. Ferner ist im vorangehenden Abschnitt die innere Struktur der Benutzerschnittstelle dargestellt worden. Nun ist es an der Zeit, ein weiteres Beispiel zu sehen, an dem alle bisher eingeführten Elemente von Flow Design im praktischen Einsatz erläutert werden. Ferner wird so die Vorgehensweise im Softwareentwicklungsprozess nochmals verdeutlicht.

Anforderungen

Es soll eine Anwendung entwickelt werden, mit der Befragungen durchgeführt werden können. Die Anwendung lädt beim Start einen Fragebogen aus der Datei `questionnaire.txt`. Dieser Fragebogen wird in einer grafischen UI angezeigt, damit ein Befragter seine Antworten eingeben kann. Nach der Befragung wird eine Auswertung angezeigt. Diese zeigt zunächst die Anzahl der korrekten Antworten. Ferner wird dann zu jeder Frage angezeigt, ob die gegebene Antwort korrekt war. Bei falscher Beantwortung wird ferner die korrekte Antwort angezeigt. Anschließend kann eine neue Befragung gestartet werden.

Die Art der Befragung ist eine *Single Choice* Befragung. Das bedeutet, pro Frage kann aus den Antwortmöglichkeiten genau eine ausgewählt werden. Die folgende Abbildung zeigt, wie ein solcher Fragebogen in der UI aussiehen könnte.

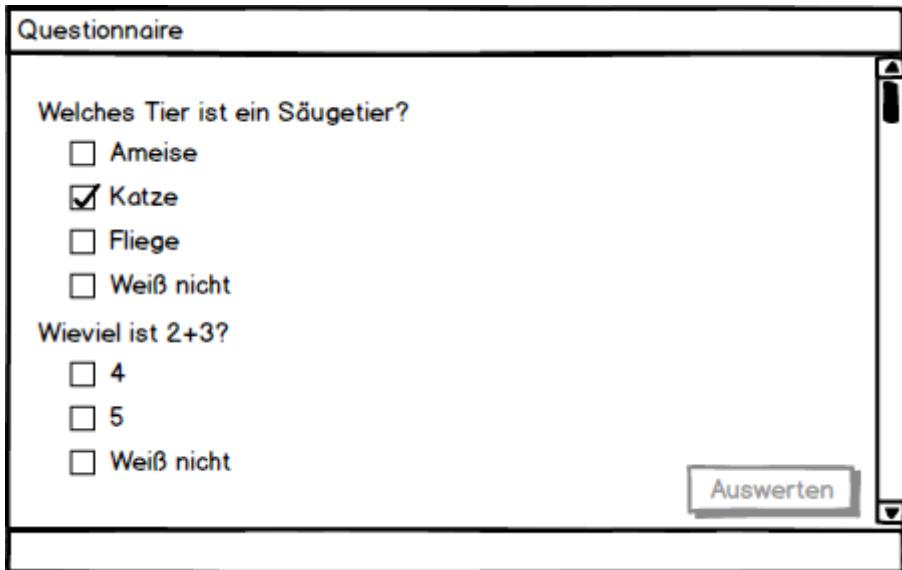


Abbildung 105: Mockup des Fragebogen Dialogs

Für den oben dargestellten Fragebogen sieht der Inhalt der Datei questionnaire.txt wie folgt aus:

```
?Welches Tier ist ein Säugetier
Ameise
*Katze
Fliege
?Wieviel ist 2+3
4
*5
```

Das Format ist leicht erkennbar. Eine neue Aufgabe beginnt mit einem Fragezeichen an der ersten Position der Zeile. Es folgt der Text der Frage. Dieser ist absichtlich nicht mit einem Fragezeichen abgeschlossen, um so die Aufgabenstellung etwas interessanter zu gestalten. Im Anschluss an die Frage folgen Zeilen mit möglichen Antworten. Das Ende der Antwortmöglichkeiten ist erreicht, wenn wieder eine Zeile mit einem Fragezeichen beginnt, oder wenn die Datei zu Ende ist. Innerhalb der Antwortmöglichkeiten ist genau eine Antwort mit einem Sternchen an der ersten Position gekennzeichnet. Damit wird die korrekte Antwort markiert. Das Sternchen muss natürlich entfernt werden, bevor der Text in der UI angezeigt wird. In der Abbildung des Dialogs ist ersichtlich, dass zu jeder Frage auch die Antwort "Weiß nicht" gegeben werden kann. Die Antwortmöglichkeit "Weiß

nicht" ist allerdings nicht in der Datei enthalten. Sie wird zu jeder Frage automatisch vom Programm ergänzt.

Der Dialog enthält am unteren Rand eine Schaltfläche, mit der die Auswertung des Fragebogens gestartet wird. Die Schaltfläche ist so lange deaktiviert, wie noch nicht alle Fragen beantwortet sind. Erst wenn zu jeder Frage genau eine Antwort gegeben ist, wird die Schaltfläche aktiviert. Der Dialog der Auswertung ist in der folgenden Abbildung dargestellt.

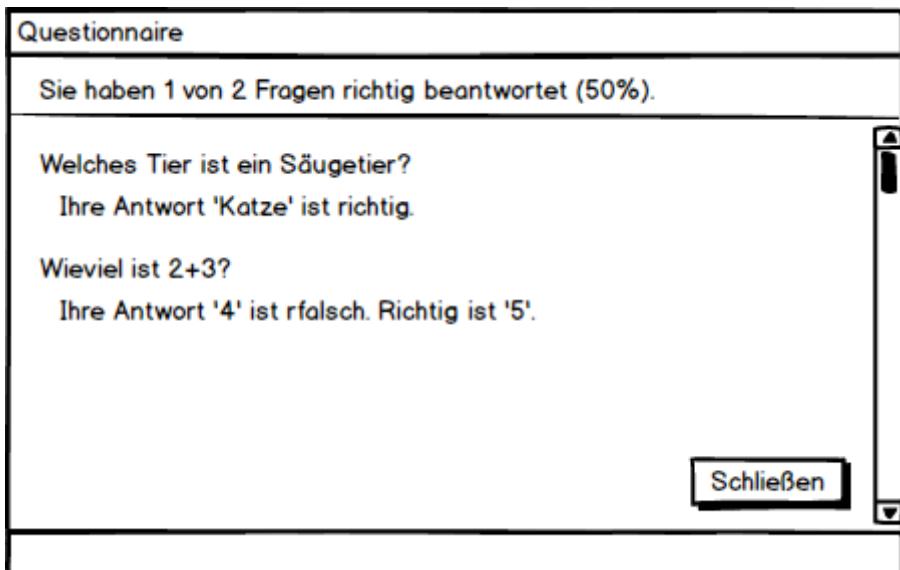


Abbildung 106: Mockup des Dialogs für die Auswertung

Die Auswertung enthält zunächst eine Kopfzeile, in der die Anzahl der korrekten Fragen angegeben ist. Ferner wird der prozentuale Anteil der richtig beantworteten Fragen angezeigt. Darunter befindet sich eine Liste der Aufgaben. Zu jeder Aufgabe wird im oben gezeigten Format angegeben, ob die Frage richtig oder falsch beantwortet wurde.

Besteht der Fragebogen aus mehr als den oben gezeigten zwei Fragen, ist ggf. ein Scrollen möglich durch den rechts neben den Fragen angezeigten Rollbalken. Das gleiche gilt für die Auswertung. Auch hier ist dann ein Scrollen möglich innerhalb der Auswertungen der einzelnen Fragen.

Dialoge und Interaktionen

Um in den Entwurf der Lösung einsteigen zu können, müssen zunächst die Anforderungen klar und eindeutig zerlegt werden. Jedes reale Stück Software ist zu groß, um es "in einem Rutsch" zu realisieren. Selbst die hier im

Buch gezeigten Übungsbeispiele sind zu groß, um alles auf einmal zu realisieren. Mindestens können die Beispiele als sportliche Übung aufgefasst werden, auch klein anmutende Anforderungen nochmals zu zerlegen. Wenn dies mit kleinen Übungsaufgaben möglich ist, gelingt es mit größeren realen Projekten allemal.

Um die Anforderungen zu zerlegen, werden in einem *Interaktionsdiagramm* die *Dialoge* mit den *Interaktionen* dargestellt. Die Interaktionen sind die größten Einheiten, die in einem Entwurf betrachtet werden. Je nachdem, wie umfangreich die Anforderungen sind, kann es sinnvoll sein, auch die Interaktionen noch weiter zu zerlegen. Es werden dann innerhalb der Interaktionen *Features* identifiziert. Bei der Zerlegung der Anforderungen geht es vor allem darum, die einzelnen Inkremeante so klein zu halten, dass sie in einem Zeitraum von 1-2 Tagen realisiert werden können. So kann regelmäßig und kurzfristig Feedback vom Product Owner eingeholt werden. Stellt man also fest, dass eine Interaktion potentiell zu groß ist, um sie als Inkrement in 1-2 Tagen zu realisieren, müssen Features der Interaktion weggelassen werden. Im Vordergrund des Softwareproduktionsprozesses steht damit der kontinuierliche Flow von Inkrementen an den Product Owner.

Die folgende Abbildung zeigt das Interaktionsdiagramm für die Questionnaire Anwendung.

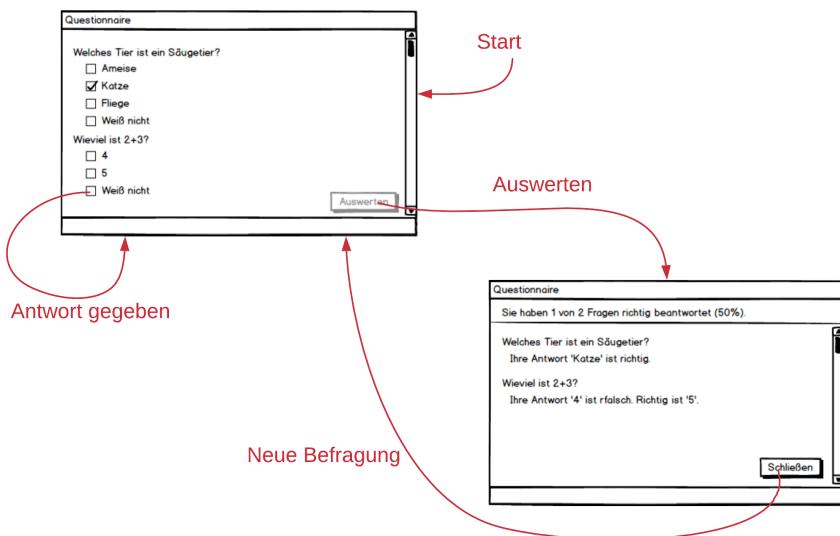


Abbildung 107: Interaktionsdiagramm für die Questionnaire Anwendung

Beim Starten der Anwendung muss der Fragebogen aus der Datei questionnaire.txt geladen werden. Ferner muss die UI so eingerichtet werden, dass der Fragebogen dort angezeigt wird. Da die Anzahl der

Aufgaben sowie die jeweilige Anzahl der Antwortmöglichkeiten variabel ist, kann die Ui nicht einfach nur angezeigt werden. Im technischen Sinne würde ein "Show Dialog" Aufruf genügen und damit gäbe es keinen Grund, *Start* als Interaktion aufzufassen. Da jedoch Domänenlogik erforderlich ist, um den Fragebogen in der Ui zu visualisieren, ist *Start* relevant im Sinne der Auflistung aller Interaktionen.

Das Anklicken einer Antwortmöglichkeit stellt eine Interaktion dar, weil die Domänenlogik daraufhin entscheiden muss, ob eine Auswertung des Fragebogens bereits möglich ist. Die Entscheidung darüber, ob die Schaltfläche "Auswerten" aktiviert werden kann, darf nicht in der Ui getroffen werden, da es eine Domänenentscheidung darstellt. Die Regel besagt hier, dass es sich um eine Single Choice Befragung handelt. Diese Regel gibt vor, wann die Schaltfläche aktiviert werden darf. Folglich muss die Entscheidung in der Domänenlogik getroffen werden. Aus diesem Grund muss der technische Event des Checkbox Controls als Interaktion des Benutzers mit der Anwendung aufgefasst werden.

Die Schaltfläche "Auswerten" löst selbstredend eine Interaktion aus. Das Erstellen der Auswertung ist Domänenlogik und fällt nicht in die Zuständigkeit der Ui.

Beim Verlassen der Auswertung soll im Fragebogen Dialog erneut ein leerer Fragebogen angezeigt werden. Die Definition, was einen leeren Fragebogen ausmacht, ist wiederum eine Domänenentscheidung. Daher ist "Neue Befragung" eine Interaktion, die dazu führt, dass die Domänenlogik einen leeren Fragebogen erstellt und an die Ui übergibt.

System-Umwelt-Diagramm

Bevor wir nun zum Entwurf der Interaktionen übergehen, soll ein System-Umwelt-Diagramm verdeutlichen, welche Rollen und Ressourcen in der Umgebung des Systems existieren. Das System-Umwelt-Diagramm ist eine gute Möglichkeit, um mit dem Product Owner ins Gespräch zu kommen. Durch die Visualisierung wird es leichter herauszufinden, ob möglicherweise weitere Rollen oder Ressourcen in der Umwelt des Systems relevant sind.

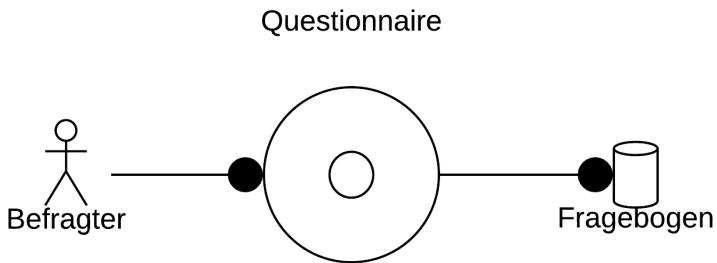


Abbildung 108: System-Umwelt-Diagramm für die Questionnaire Anwendung

Das Diagramm zeigt für unser Beispiel Questionnaire lediglich die Rolle *Befragter*. Nur diese Rolle erhält vom System eine Leistung, die darin besteht, dass ein Befragter einen Fragebogen ausfüllen kann. Eine mögliche weitere Rolle wäre denkbar für die Aufgabe, den Fragebogen zu erstellen. Der Fragebogen wird jedoch aus einer Textdatei gelesen. Die Anwendung soll keine Funktionen bereitstellen, mit dem der Fragebogen erstellt und bearbeitet werden kann. Folglich ist diese Rolle im System-Umwelt-Diagramm nicht aufgeführt. Stattdessen bearbeitet der Ersteller eines Fragebogens diesen mit einem Texteditor.

Auf der Seite der Ressourcen ist das System von der Fragebogendatei abhängig. Diese befindet sich außerhalb des Systems im Dateisystem. Weitere Ressourcen sind hier nicht relevant. Denkbar wäre in einer späteren Version, dass die Eingaben der Befragten in einer Datei gespeichert werden, um sie für eine spätere Auswertung nutzen zu können. In dem Fall müsste eine weitere Ressource zur Aufnahme der eingegebenen Daten ergänzt werden.

Entwurf der obersten Ebene

Ausgehend vom Interaktionsdiagramm können nun die einzelnen Interaktionen in einen Entwurf der obersten Ebene überführt werden. Die Struktur dieses Entwurfs ist leicht herzustellen. Es wird jeder Interaktionspfeil betrachtet. Der Name der Interaktion wird jeweils in die Mitte des Entwurfs geschrieben und mit einer Ellipse umrandet. Diese Funktionseinheit stellt die Domänenlogik der Interaktion dar. Anschließend werden Start- und Endpunkt des Interaktionspfeils betrachtet. Die dort vorgefundenen Dialoge werden vor bzw. hinter die Funktionseinheit der Domänenlogik gezeichnet und passend beschriftet. Interaktionen wie Start und Ende, die nicht in einem Dialog beginnen bzw. nicht in einem Dialog enden, werden mit einem

kleinen Kreis markiert, anstelle des Dialogs. Für Start ist das in der nachfolgenden Abbildung zu sehen, eine Ende Interaktion wird im Beispiel nicht verwendet.

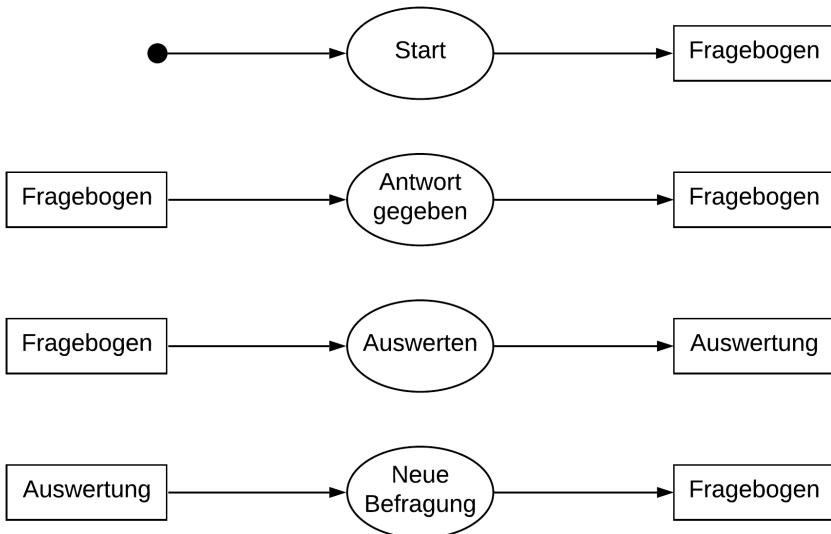


Abbildung 109: Entwurf der obersten Ebene; noch ohne konkrete Daten

Die Abbildung zeigt die Struktur des Entwurfs, bestehend aus den Dialogen und Funktionseinheiten für die Domänenlogik der einzelnen Interaktionen, die sogenannten *Interaktoren*. Ein Interaktor ist eine Funktionseinheit, die eine Interaktion realisiert.

Was fehlt, sind die Angaben zu den Daten, die auf den einzelnen Datenflüssen transportiert werden.

Für die Start Interaktion ist zunächst klar, dass keine Daten in Start hinein fließen. Ausgehend vom Betriebssystem könnten hier die Kommandozeilenparameter fließen. Bei einer typischen Desktopanwendung werden diese allerdings eher nicht verwendet. Spannender ist schon die Frage, was von Start zum Fragebogen Dialog fließt. Die UI muss anhand dieser Daten in der Lage sein, den Fragebogen darzustellen. Es müssen also zu jeder Frage aus der Datei der Text der Frage sowie die möglichen Antworten an die UI geliefert werden. Auch die Antwortmöglichkeit "Weiß nicht" muss darin schon enthalten sein, da es nicht in die Zuständigkeit der Benutzeroberfläche fällt, diese Antwortmöglichkeit zu ergänzen. Ich habe die Fragen mit den möglichen Antworten zu einer Datenstruktur *Aufgabe* zusammengefasst, die weiter unten erläutert wird.

Neben den Aufgaben muss die Information darüber, ob der Fragebogen auswertbar ist, an die UI übertragen werden. Anhand dieser Information aktiviert oder deaktiviert die UI die zugehörige Schaltfläche.

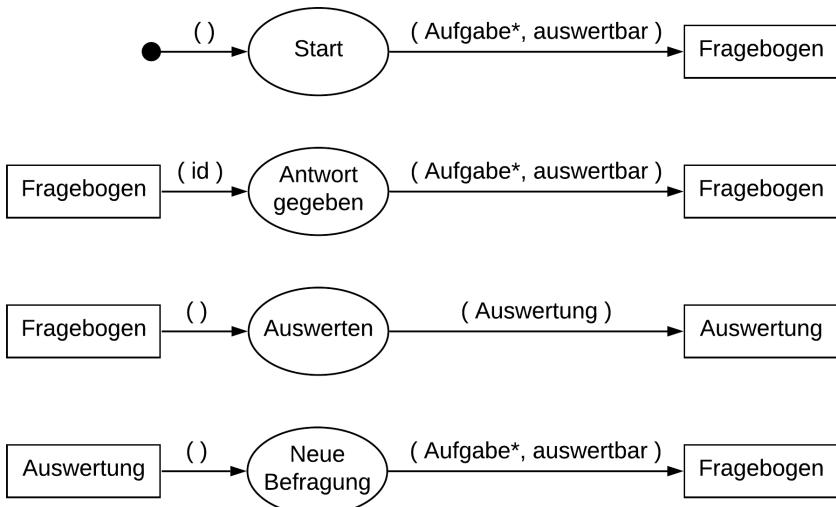


Abbildung 110: Entwurf der obersten Ebene mit den Datenflüssen

Der im Entwurf verwendete Datentyp *Aufgabe* sieht wie folgt aus:

Aufgabe

```

-----
Frage : string
Antwortmöglichkeiten : Antwortmöglichkeit*
-----
```

```

Id : int
Antwort : string
IstGegeben : bool
IstKorrekt : bool
```

Eine Aufgabe besteht zunächst aus dem Fragetext. Die Eigenschaft wird hier *Frage* genannt und ist vom Typ *String*. Ferner verfügt eine Aufgabe über eine Aufzählung von *Antwortmöglichkeiten*. Jede der Antwortmöglichkeiten hat eine eindeutige *Id*. Diese wird bei der Interaktion *Antwort gegeben* verwendet, um die Antwortmöglichkeit zu identifizieren, die vom Benutzer angeklickt wurde. Ferner enthält jede Antwortmöglichkeit den anzuzeigenden Text sowie zwei Eigenschaften, die aussagen, ob die Antwort angehakt ist (*IstGegeben*) bzw. ob es sich um die korrekte Antwort handelt (*IstKorrekt*).

Streng genommen ist für die Interaktion *Start* keine Eigenschaft erforderlich, an der die korrekte Antwort erkannt werden kann. Diese Information wird erst benötigt, wenn ein beantworteter Fragebogen ausgewertet werden soll. Insofern könnte diese Eigenschaft rein für die *Start* Interaktion zunächst weggelassen werden. Andererseits wird die Datei *questionnaire.txt* zu diesem Zeitpunkt, nämlich beim Starten der Anwendung, eingelesen. Da liegt es nahe, auch die Information über die korrekte Antwort mit in der Datenstruktur abzulegen. Andernfalls müssten später zwei unterschiedliche Datenstrukturen wieder zusammengeführt werden. Da es sich lediglich um eine einzelne Eigenschaft handelt, wäre der Aufwand für die klare Trennung zu hoch.

Neben einer Aufzählung von Aufgaben liefert die *Start* Interaktion auch ein boolesche Variable *auswertbar* an den Dialog. Der Dialog kann damit die Schaltfläche aktivieren oder deaktivieren.

Bei der Interaktion *Antwort gegeben* wird vom Dialog eine *Id* an die Funktionseinheit übergeben. Damit kann die Domänenlogik die Antwortmöglichkeit identifizieren, auf die der Benutzer geklickt hat. Ist die zugehörige Antwortmöglichkeit gefunden, muss geprüft werden, ob sie bereits markiert ist. Wenn das der Fall ist, muss die Markierung entfernt werden. Ist sie noch nicht markiert, muss die Markierung bei allen anderen Antwortmöglichkeiten entfernt werden, die zur selben Aufgabe gehören. Und natürlich muss die Markierung bei der angeklickten Antwortmöglichkeit gesetzt werden. Auf diese Weise wird das Single Choice Verhalten hergestellt. Als Resultat liefert die Funktionseinheit wieder eine Aufzählung von *Aufgaben* an den Dialog sowie das Flag *auswertbar*.

Sind alle Aufgaben bearbeitet und damit die Schaltfläche “Auswerten” aktiviert, kann der Benutzer die Auswertung anfordern. Der Dialog liefert dazu lediglich ein Signal an die Funktionseinheit, die daraufhin ein Objekt vom Typ *Auswertung* erstellt. Diese Datenstruktur sieht wie folgt aus:

Auswertung

```
RichtigeAufgaben : int
AnzahlAufgaben : int
Ergebnisse : Ergebnis*
-----
Frage : string
GegebeneAntwort : string
RichtigeAntwort : string
IstKorrekt : bool
```

Die Datenstruktur enthält alle Angaben die erforderlich sind, um den Auswertungsdialog anzeigen zu können. Dabei soll der Dialog die Daten nicht weiter interpretieren müssen. Eine Ausnahme bildet dabei der

anzuzeugende Prozentwert. Dieser wird nicht in der Datenstruktur an den Dialog übergeben, sondern muss vom Dialog selbst berechnet werden. Dem Dialog werden lediglich die Gesamtzahl der Aufgaben übergeben sowie die Anzahl der richtig beantworteten. Daraus einen Prozentwert zu berechnen stellt keine Domänenlogik im Sinne einer Fragebogenanwendung dar. Insofern ist es legitim, diese Aufgabe an die Ui zu delegieren. Der Ui steht es grundsätzlich frei, die beiden Zahlen statt als Prozentwert in Form einer Tortengrafik darzustellen. Ihre Zuständigkeit ist die Präsentation der Daten. Im Fall einer Tortengrafik würde die Ui ebenfalls nur die Rohdaten erhalten und daraus das Diagramm berechnen.

Bei den Antworten sind jeweils die Texte hinterlegt, die im Dialog angezeigt werden müssen. Dabei kommen zwei unterschiedliche Formatierungen zum Einsatz. Hat der Befragte die Aufgabe richtig beantwortet, wird nur der Text der richtigen Antwort ausgegeben. Hat er sie falsch beantwortet, werden seine falsche Antwort sowie die richtige Antwort angezeigt. Um diese beiden unterschiedlichen Formatierungen unterscheiden zu können, enthält jedes Ergebnis zusätzlich zu den Texten noch eine boolesche Eigenschaft *IstKorrekt*, die aussagt, welcher der beiden Fälle vorliegt. Auf diese Weise wird verhindert, dass der Dialog die Daten interpretiert. In diesem simplen Beispiel ist es leicht, die beiden Texte zu vergleichen um darüber zu entscheiden, ob die Frage richtig beantwortet wurde. Allerdings kann diese Domänenentscheidung auch komplizierter ausfallen und insbesondere kann sie sich im Laufe der Zeit ändern. Es soll dann nicht erforderlich sein, eine entsprechende Logik in der Ui anzupassen.

Zuletzt liefert dann die Interaktion *Neue Befragung* wieder eine Aufzählung von *Aufgaben* sowie die Information über die Auswertbarkeit an den Dialog. Zum Verständnis der Lösung ist es wichtig zu verstehen, dass die Anwendung ihren Zustand in der Domänenlogik hält. Die Dialoge erhalten alle Daten angeliefert, aus denen sie jeweils die Visualisierung herstellen. Aus diesem Grund ist es der Funktionseinheit *Auswerten* möglich, eine Auswertung zu erstellen, ohne von der Ui irgendwelche Daten zu erhalten. Zum besseren Verständnis kann der Zustand an den Funktionseinheiten annotiert werden. Dies ist insbesondere hilfreich, weil die Funktionseinheiten gemeinsamen Zustand verwenden. Dieser wird initial in *Start* und *Neue Befragung* initialisiert. Die Funktionseinheit *Antwort gegeben* verändert den Zustand dann immer wieder, bis schließlich eine Auswertung möglich ist. Die folgende Abbildung zeigt den Entwurf mit eingezeichnetem Zustand.

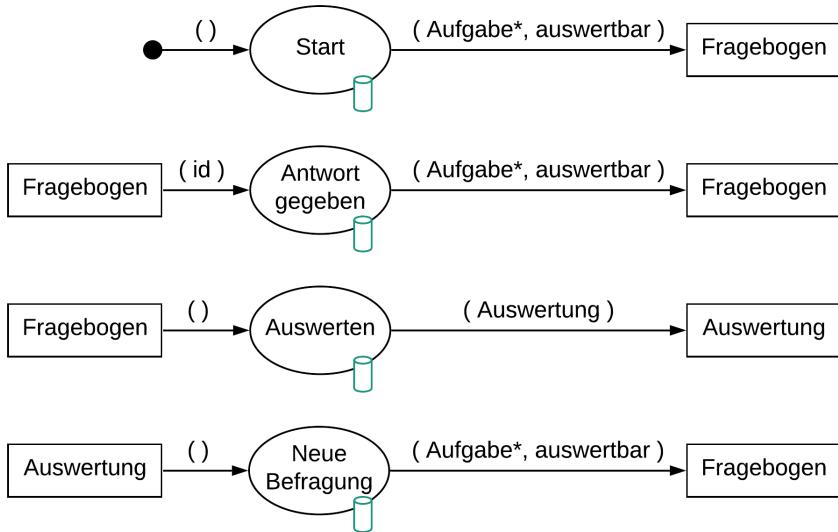


Abbildung 111: Entwurf der obersten Ebene inkl. Zustand

Auswahl einer Interaktion

Bevor der Entwurf nun weiter verfeinert werden kann, muss das Team gemeinsam mit dem Product Owner eine der Interaktionen auswählen. Der Product Owner wird die Interaktion auswählen, die für ihn bzw. die Endkunden den größten Geschäftswert hat. Möglicherweise interessiert ihn auch zuerst die Interaktion, die für Marketing und Vertrieb am interessantesten erscheinen.

Das Team hat bei der Auswahl ein Mitspracherecht. Es hat nämlich ein Interesse daran, die Interaktionen frühzeitig zu realisieren, in denen technische Herausforderungen vermutet werden. Es wäre bspw. denkbar, dass das Team eine Technologie zum ersten Mal produktiv einsetzt. Dann sollte es Interaktionen, welche diese Technologie benötigen, früh im Projekt bearbeiten, damit mögliche Unwägbarkeiten möglichst früh erkannt werden.

Grundsätzlich ist die Reihenfolge bei der Auswahl der Interaktionen nicht festgelegt. Natürlich ergibt sich häufig eine natürliche Reihenfolge. So ist es beim Questionnaire naheliegend, mit *Start* zu beginnen. Ohne Start wird es schwer, eine Antwortmöglichkeit anzukreuzen. Doch kann das Projekt auch mit der Interaktion *Antwort gegeben* begonnen werden. Es müsste dann ein Fragebogen fest im Dialog eingeprägt werden, weil die Interaktion *Start* ja noch nicht zur Verfügung steht. Die Reihenfolge ist also beliebig und unter Zuhilfenahme von Attrappen können die Vorbedingungen meist einfach hergestellt werden.

In diesem Beispiel soll es mit der Interaktion *Start* los gehen. Ziel des Inkrements ist zu sehen, dass die Datei `questionnaire.txt` korrekt eingelesen wird und dass der Fragebogen im Dialog mit den entsprechenden Controls angezeigt wird.

Verfeinerung von 'Start'

Bevor mit der Implementation des ersten Inkrements begonnen werden kann, muss der Entwurf so weit verfeinert werden, dass eine flüssige Umsetzung möglich ist. Dazu muss in der Verfeinerung dafür Sorge getragen werden, dass jede Funktionseinheit für nur einen einzigen Aspekt zuständig ist. Ferner müssen die Funktionseinheit so klein sein, dass sie jeweils in maximal 4 Stunden implementiert werden können. Hier gilt die Daumenregel, dass das gesamte Inkrement vermutlich nicht in 2 Tagen fertig sein wird, wenn einzelne Funktionseinheiten länger als 4 Stunden benötigen. Dies ist lediglich ein Anhaltspunkt. Es geht nicht darum, den Aufwand möglichst exakt zu schätzen. Wichtig ist hier, dass eine Funktionseinheit, bei der jemand vermutet, die Implementation könnte länger als 4 Stunden in Anspruch nehmen, nochmal weiter verfeinert wird. Entweder es zeigt sich dann, dass einfachere Funktionseinheit entstehen und die weitere Verfeinerung hilfreich war. Oder es zeigt sich, dass die Funktionseinheit doch nicht so aufwendig war, wie vermutet. In jedem Fall ist es besser, solche Unwägbarkeiten in der Entwurfsphase zu klären, als während der Implementation auf Probleme zu stoßen.

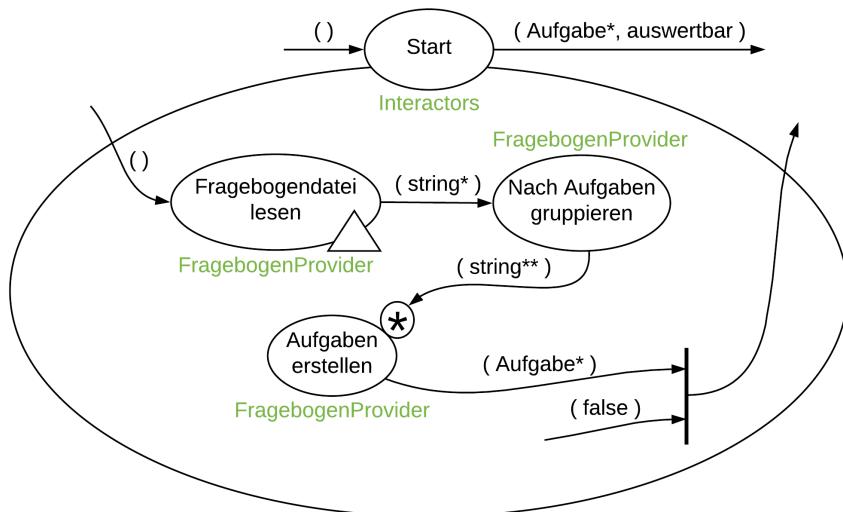


Abbildung 112: Entwurf von Start, verfeinert

Die Abbildung zeigt eine Verfeinerung von *Start*. Zunächst wird die Datei `questionnaire.txt` eingelesen. Da der Dateiname in den Anforderungen festgelegt ist, muss kein Parameter fließen. Die eingelesenen Zeilen werden im Anschluss nach den Aufgaben gruppiert. Jeder Zeile, die mit einem Fragezeichen beginnt, leitet eine neue Aufgabe ein. Diese gruppierte Stringliste wird dann von der Funktionseinheit *Aufgaben erstellen* in Aufgaben Objekte überführt. Wichtig ist hier festzuhalten, dass die Gruppierung zu einer Liste von Listen von Strings führt, angezeigt durch `string**`. In der Implementation kann dies bspw. auf Arrays oder Listen abgebildet werden. Des weiteren verwendet die Funktionseinheit *Aufgaben erstellen* eine spezielle Syntax um auszudrücken, dass sie in der Implementation aus zwei Methoden besteht. Der Stern zeigt an, dass die äußere Methode über den Input iteriert und für jedes Element eine weitere Methode aufruft. Diese Methode liefert jeweils zur Eingabe `string*` ein *Aufgabe* Objekt. Die drei Funktionseinheiten, realisiert als Methoden, werden gemeinsam in der Klasse *FragebogenProvider* abgelegt. Ich sehe hier einen engen Zusammenhang zwischen dem physischen Lesen des Fragebogens aus einer Datei einerseits, sowie dem Aufbau oder Format der Datei andererseits. Die Wahrscheinlichkeit ist sehr hoch, dass sich beim Ändern der physischen Datenquelle auch das Format ändern wird. Insofern betrachte ich das physische Lesen und das interpretieren des Formats als einen Aspekt. Durch die Trennung der Teilaufgaben in mehrere Methoden ist sichergestellt, dass eine Änderung an einem der Teilaufgaben trotzdem nicht dazu führt, dass ein nur einer einzigen Methode Änderungen vorgenommen werden müssen.

Implementation

Folgendes Listing zeigt die Implementation der Klasse *FragebogenProvider*.

```
public static class FragebogenProvider
{
    public static IEnumerable<string> Fragebogendatei_einlesen() {
        return File.ReadAllLines("questionnaire.txt");
    }

    public static IEnumerable<IEnumerable<string>> Nach_Aufgaben_gruppieren(
        IEnumerable<string> zeilen) {
        var result = new List<string>();
        foreach (var zeile in zeilen) {
            if (zeile.StartsWith("?")) {
                if (result.Count > 0) {
                    yield return result;
                    result = new List<string>();
                }
                result.Add(zeile.Substring(1) + "?");
            }
            else {
                result.Add(zeile);
            }
        }
        if (result.Count > 0) {
            yield return result;
        }
    }
}
```

```

public static IEnumerable<Aufgabe> Aufgaben_ erstellen(
    IEnumerable<IEnumerable<string>> gruppierteZeilen) {
    foreach (var gruppe in gruppierteZeilen) {
        yield return Aufgaben_ erstellen(gruppe);
    }
}

private static Aufgabe Aufgaben_ erstellen(
    IEnumerable<string> zeilenEinerAufgabe) {
    var result = new Aufgabe();
    result.Frage = zeilenEinerAufgabe.First();
    result.Antwortmöglichkeiten = Antwortmöglichkeiten_ erstellen(
        zeilenEinerAufgabe.Skip(1));
    return result;
}

private static List<Antwortmöglichkeit> Antwortmöglichkeiten_ erstellen(
    IEnumerable<string> zeilen) {
    var result = new List<Antwortmöglichkeit>();
    foreach (var zeile in zeilen) {
        var antwortmöglichkeit = new Antwortmöglichkeit();
        if (zeile.StartsWith("*")) {
            antwortmöglichkeit.IstKorrekt = true;
        }
        antwortmöglichkeit.Antwort = zeile.TrimStart('*');
        result.Add(antwortmöglichkeit);
    }
    return result;
}
}

```

Es fehlt noch die Integration der Bestandteile durch die Klasse *Interactors*.

```

using System.Collections.Generic;
using questionnaire.contracts;
using questionnaire.ressources;

namespace questionnaire
{
    public class Interactors
    {
        public static (IEnumerable<Aufgabe> aufgaben, bool auswertbar) Start() {
            var zeilen = FragebogenProvider.Fragebogendatei_einlesen();
            var gruppierteZeilen = FragebogenProvider.Nach_Aufgaben_gruppieren(zeilen);
            var aufgaben = FragebogenProvider.Aufgaben_ erstellen(gruppierteZeilen);
            return (aufgaben, false);
        }
    }
}

```

Die Klasse setzt den Entwurf 1:1 um. Um die Korrektheit sicherzustellen, ist ein Integrationstest für die Klasse *Interactors* ergänzt.

```
using System.IO;
using System.Linq;
using NUnit.Framework;

namespace questionnaire.tests
{
    [TestFixture]
    public class InteractorsTests
    {
        [SetUp]
        public void Setup() {
            Directory.SetCurrentDirectory(TestContext.CurrentContext.TestDirectory);
        }

        [Test]
        public void Integrationstest() {
            var (aufg, auswertbar) = Interactors.Start();
            var aufgaben = aufg.ToArray();

            Assert.IsFalse(auswertbar);

            Assert.That(aufgaben.Length, Is.EqualTo(2));

            Assert.That(aufgaben[0].Frage, Is.EqualTo("Welches Tier ist ein Säugetier?"));
            Assert.That(aufgaben[0].Antwortmöglichkeiten.Count, Is.EqualTo(3));
            Assert.That(aufgaben[0].Antwortmöglichkeiten[0].Antwort, Is.EqualTo("Katze"));
            Assert.That(aufgaben[0].Antwortmöglichkeiten[0].IstKorrekt, Is.True);
            Assert.That(aufgaben[0].Antwortmöglichkeiten[1].Antwort, Is.EqualTo("Ameise"));
            Assert.That(aufgaben[0].Antwortmöglichkeiten[1].IstKorrekt, Is.False);
            Assert.That(aufgaben[0].Antwortmöglichkeiten[2].Antwort, Is.EqualTo("Fliege"));
            Assert.That(aufgaben[0].Antwortmöglichkeiten[2].IstKorrekt, Is.False);

            Assert.That(aufgaben[1].Frage, Is.EqualTo("Wieviel ist 2+3?"));
            Assert.That(aufgaben[1].Antwortmöglichkeiten.Count, Is.EqualTo(3));
            Assert.That(aufgaben[1].Antwortmöglichkeiten[0].Antwort, Is.EqualTo("4"));
            Assert.That(aufgaben[1].Antwortmöglichkeiten[0].IstKorrekt, Is.False);
            Assert.That(aufgaben[1].Antwortmöglichkeiten[1].Antwort, Is.EqualTo("5"));
            Assert.That(aufgaben[1].Antwortmöglichkeiten[1].IstKorrekt, Is.True);
            Assert.That(aufgaben[1].Antwortmöglichkeiten[2].Antwort, Is.EqualTo("6"));
            Assert.That(aufgaben[1].Antwortmöglichkeiten[2].IstKorrekt, Is.False);
        }
    }
}
```

Dieser Test benötigt zur Laufzeit die Datei `questionnaire.txt`. Daher ist eine solche Datei für die Integrationstests in das Testprojekt eingebunden. Sie wird durch den Buildprozess in das Ausgabeverzeichnis `bin/Debug` kopiert und steht damit im Test zur Verfügung.

Nun ist ein Integrationstest noch kein Durchstich. Der Product Owner kann zu diesem Test kein fundiertes Feedback geben. Für einen ersten Durchstich fehlt eine UI. Da diese erwartungsgemäß etwas mehr Zeit in Anspruch nimmt, kann man zunächst auch eine deutlich einfachere UI erstellen: eine Konsolen UI. Auf diese Weise erhält der Product Owner eine erst lauffähige Version des Programmes. Natürlich kann damit noch keine Befragung

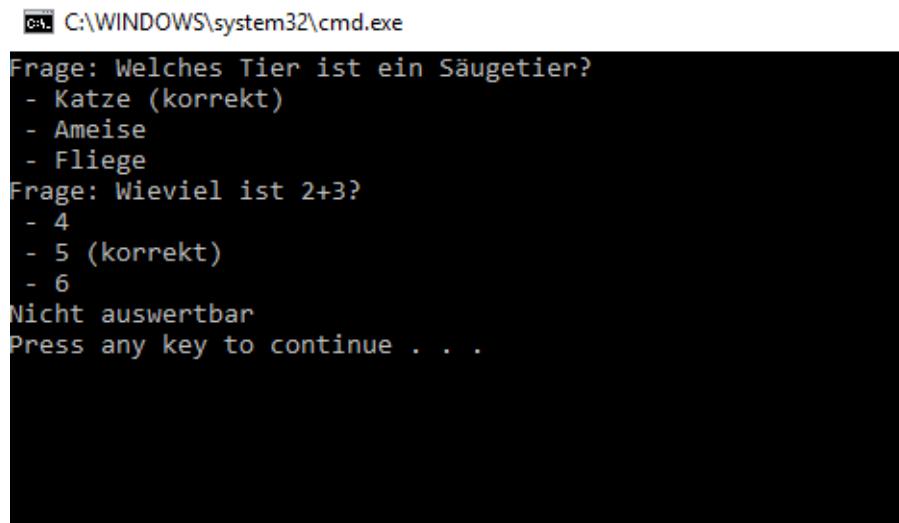
durchgeführt werden. Er kann jedoch schon mit unterschiedlichen questionnaire.txt Dateien ausprobieren, ob seine Anforderungen umgesetzt wurden. Das folgende Listing zeigt eine minimal Konsolen UI.

```
using System;
using System.Collections.Generic;
using questionnaire.contracts;

namespace questionnaire
{
    internal class Program
    {
        [STAThread]
        public static void Main(string[] args) {
            var (aufgaben, auswertbar) = Interactors.Start();
            Show(aufgaben, auswertbar);
        }

        private static void Show(IEnumerable<Aufgabe> aufgaben, bool auswertbar) {
            foreach (var aufgabe in aufgaben) {
                Console.WriteLine($"Frage: {aufgabe.Frage}");
                foreach (var antwortmöglichkeit in aufgabe.Antwortmöglichkeiten) {
                    var korrekt = antwortmöglichkeit.IstKorrekt ? " (korrekt)" : "";
                    Console.WriteLine($" - {antwortmöglichkeit.Antwort}{korrekt}");
                }
                Console.WriteLine(auswertbar ? "Auswertbar" : "Nicht auswertbar");
            }
        }
    }
}
```

Auf der Konsole führt dies zu folgender Ausgabe:



```
C:\WINDOWS\system32\cmd.exe
Frage: Welches Tier ist ein Säugetier?
- Katze (korrekt)
- Ameise
- Fliege
Frage: Wieviel ist 2+3?
- 4
- 5 (korrekt)
- 6
Nicht auswertbar
Press any key to continue . . .
```

Mit diesem ersten minimalen Durchstich hat der Product Owner die Möglichkeit, dem Team Feedback zu geben. Das Feedback bezieht sich bislang natürlich nur darauf, ob die Datei korrekt eingelesen wird. Im Anschluss kann nun die grafische UI erstellt werden.

Grafische UI

Die UI ist mit WPF unter Windows erstellt. Das folgende Listing zeigt die XAML Datei.

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition Height="64"/>
    </Grid.RowDefinitions>

    <ScrollViewer VerticalScrollBarVisibility="Auto">
        <StackPanel x:Name="_panel"/>
    </ScrollViewer>

    <StackPanel Grid.Row="1" Orientation="Horizontal">
        <Button x:Name="_btnAuswerten" Content="Auswerten" Margin="10" Padding="10"/>
    </StackPanel>
</Grid>
```

Hier wird lediglich ein Gerüst für den Dialog definiert. Das StackPanel mit dem Namen *_panel* wird in der Codebehind mit den passenden Controls gefüllt, so dass der Fragebogen auf diese Weise dynamisch angezeigt werden kann.

```
using System.Collections.Generic;
using System.Windows;
using System.Windows.Controls;
using questionnaire.contracts;

namespace questionnaire.ui
{
    public partial class FragebogenUi : Window
    {
        public FragebogenUi() {
            InitializeComponent();
        }

        public void Update(IEnumerable<Aufgabe> aufgaben, bool auswertbar) {
            _panel.Children.Clear();
            foreach (var aufgabe in aufgaben) {
                var panel = new StackPanel {
                    Orientation = Orientation.Vertical,
                    Margin = new Thickness(10) };
                var label = new Label {Content = aufgabe.Frage};
                panel.Children.Add(label);
                foreach (var antwortmöglichkeit in aufgabe.Antwortmöglichkeiten) {
                    var radio = new RadioButton {Content = antwortmöglichkeit.Antwort};
                    panel.Children.Add(radio);
                }
                _panel.Children.Add(panel);
            }
            _btnAuswerten.IsEnabled = auswertbar;
        }
    }
}
```

Zuletzt muss diese Ui noch in der *Main* Methode integriert werden.

```
using System;
using System.Collections.Generic;
using System.Windows;
using questionnaire.contracts;
using questionnaire.ui;

namespace questionnaire
{
    internal class Program
    {
        [STAThread]
        public static void Main(string[] args) {
            var ui = new FragebogenUi();
            var (aufgaben, auswertbar) = Interactors.Start();
            ui.Update(aufgaben, auswertbar);

            var app = new Application{MainWindow = ui};
            app.Run(ui);
        }
    }
}
```

Das Ergebnis kann sich schon sehen lassen! Für eine reale Anwendung müsste an der Optik noch etwas getan werden, aber für dieses Beispiel soll das genügen.

 Questionnaire

— □ ×

Welches Tier ist ein Säugetier?

Katze
 Ameise
 Fliege

Wieviel ist $2+3$?

4
 5
 6

Auswerten

Verfeinerung von 'Antwort gegeben'

In der nächsten Iteration soll die Interaktion *Antwort gegeben* realisiert werden. Dazu wird zunächst der Entwurf verfeinert. Aus dem Entwurf der obersten Ebene ist ersichtlich, dass die Funktionseinheit eine Id als Eingabe erhält. Als Resultat muss die Funktionseinheit die Liste der Aufgaben liefern. Darin sind möglicherweise Eigenschaften modifiziert, so dass sich der Zustand der UI ändern wird. Ferner liefert die Funktionseinheit mit dem Flag *auswertbar* die Information darüber, ob der Fragebogen nun ausgewertet werden kann.

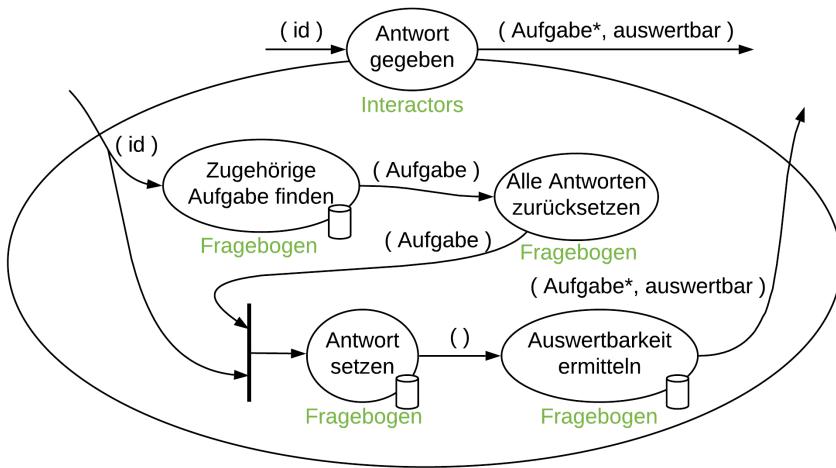


Abbildung 113: Entwurf *Antwort gegeben*

Die Funktionseinheit *Antwort gegeben* wird als Methode in der Klasse *Interactors* realisiert. Alle verwendeten Funktionseinheiten sind Operationen aus der Klasse *Fragebogen*. Es handelt sich hier um die Domänenlogik der Anwendung, daher wird ein Bezeichner verwendet, der die Domäne bestmöglich repräsentiert.

Es kommt nun darauf an, dass sich die Funktionseinheiten *Start* und *Antwort gegeben* die Aufgaben aus der Datei *questionnaire.txt* als gemeinsamen Zustand teilen. *Start* liest den Fragebogen ein. Auf diesen Zustand muss sich nun *Antwort gegeben* beziehen können. Aus diesem Grund haben die beiden Funktionseinheiten *Zugehörige Aufgabe finden* und *Auswertbarkeit ermitteln* eine Zustandstonne annotiert. Sie greifen auf die Liste der Aufgaben zu, die von *Start* aufgebaut wurde.

Der Ablauf des Interactors *Antwort gegeben* ist dann wie folgt: zunächst wird die Aufgabe ermittelt, auf die sich die vom Anwender angeklickte Antwortmöglichkeit bezieht. Dazu liefert die UI die *Id* der Antwortmöglichkeit, die der Anwender angeklickt hat. Als nächstes wird in der gefundenen Aufgabe die Eigenschaft *IstGegeben* aller Antwortmöglichkeiten auf *false* gesetzt. Damit ist nun innerhalb der Aufgabe keine Antwortmöglichkeit angekreuzt. Anschließend wird die vom Anwender angeklickte Antwortmöglichkeit von der Funktionseinheit *Antwort setzen* angekreuzt. Zuletzt wird noch ermittelt, ob der Fragebogen nun auswertbar ist. Dazu wird geprüft, ob in jeder Aufgabe eine Antwortmöglichkeit selektiert ist. Diese Information sowie alle Aufgaben werden als Ergebnis zurück an die UI übermittelt. Diese baut daraufhin alle Controls neu auf. Dadurch spiegelt sich das Anklicken des Anwenders im entsprechenden *RadioButton* Control wieder. Ferner wird die *Auswerten*

Schaltfläche nun erst dann aktiviert, wenn in jeder Aufgabe eine Antwortmöglichkeit selektiert ist.

Da *Start* die Liste der Aufgaben bislang lediglich an die UI geliefert hat, muss die Funktionseinheit so modifiziert werden, dass die Methoden der Klasse *Fragebogen* Zugriff auf die Liste der Aufgaben haben.

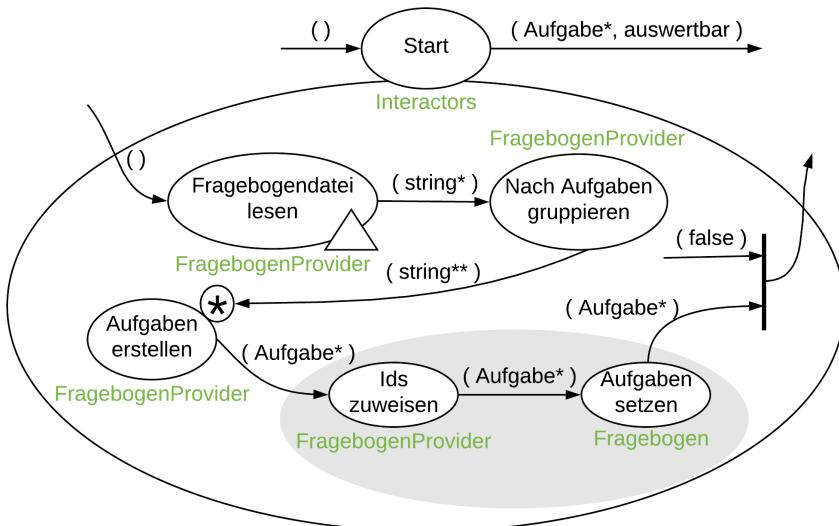


Abbildung 114: Angepasster Entwurf von Start

Es ist daher die Funktionseinheit *Aufgaben setzen* in der Klasse *Fragebogen* ergänzt worden. Auf diese Weise erhält die Klasse *Fragebogen* Zugriff auf die Liste aller Aufgaben. Doch es ist noch eine weitere Ergänzung von *Start* erforderlich. Damit die UI der Domänenlogik mitteilen kann, auf welche Antwortmöglichkeit der Anwender geklickt hat, benötigen die Antwortmöglichkeiten jeweils eine eindeutige *Id*. Dazu ist die Funktionseinheit *Start* um die Operation *Ids zuweisen* erweitert worden. Die Funktionseinheit iteriert über alle Aufgaben und innerhalb jeder Aufgabe über alle Antwortmöglichkeiten, um so jeder Antwortmöglichkeit eine fortlaufende Nummer zuzuordnen. Aufgabe der UI ist es, diese *Id* zu liefern, wenn der Anwender auf eine Antwortmöglichkeit klickt.

Implementation

Um die *Id* aufzunehmen nehmen, ist in der Datenstruktur *Antwortmöglichkeit* eine weitere Eigenschaft *Id* hinzugekommen. Ferner ist die Eigenschaft

IstGegeben ergänzt worden. In dieser booleschen Eigenschaft wird hinterlegt, ob der Benutzer diese Antwortmöglichkeit angekreuzt hat.

```
public class Antwortmöglichkeit
{
    public bool IstKorrekt { get; set; }

    public bool IstGegeben { get; set; }

    public string Antwort { get; set; }

    public int Id { get; set; }
}
```

In der UI ist der Event *Antwort_gegeben* ergänzt, der ausgelöst wird, wenn der Anwender eine der Antwortmöglichkeiten anklickt. Der Event liefert die *Id* des angeklickten *RadioButton* Controls als Parameter.

```
public event Action<int> Antwort_gegeben;
```

Damit der Event ausgelöst wird, wenn der Anwender eines der *RadioButton* Controls anklickt, ist die Methode *Update* ergänzt.

```

public void Update(IEnumerable<Aufgabe> aufgaben, bool auswertbar) {
    _panel.Children.Clear();
    foreach (var aufgabe in aufgaben) {
        var panel = new StackPanel {
            Orientation = Orientation.Vertical,
            Margin = new Thickness(10) };
        var label = new Label {Content = aufgabe.Frage};
        panel.Children.Add(label);
        foreach (var antwortmöglichkeit in aufgabe.Antwortmöglichkeiten) {
            var radio = new RadioButton {
                Content = antwortmöglichkeit.Antwort,
                Tag = antwortmöglichkeit.Id,
                IsChecked = antwortmöglichkeit.IstGegeben
            };
            radio.Checked += (o, e) => Antwort_gegeben((int)radio.Tag);
            panel.Children.Add(radio);
        }
        _panel.Children.Add(panel);
    }
    _btnAuswerten.IsEnabled = auswertbar;
}

```

Die Ids der einzelnen Antwortmöglichkeiten werden jeweils in der *Tag* Eigenschaft der *RadioButton* Controls abgelegt. Auf diese Weise kann beim Auslösen der Events die zugehörige Id ermittelt werden.

Das folgende Listing zeigt die aktualisierte *Start* Methode aus der Klasse *Interactors*.

```

public (IEnumerable<Aufgabe> aufgaben, bool auswertbar) Start() {
    var zeilen = FragebogenProvider.Fragebogendatei_einlesen();
    var gruppierteZeilen = FragebogenProvider.Nach_Aufgaben_gruppieren(zeilen);
    var aufgaben = FragebogenProvider.Aufgaben_erstellen(gruppierteZeilen).ToArray();
    FragebogenProvider.Ids_zuweisen(aufgaben);
    _fragebogen.Aufgaben_setzen(aufgaben);
    return (aufgaben, false);
}

```

Ebenfalls in der Klasse *Interactors* liegt die Methode *Antwort_gegeben*.

```
public (IEnumerable<Aufgabe> aufgaben, bool auswertbar) Antwort_gegeben(int id) {
    var aufgabe = _fragebogen.Zugehörige_Aufgabe_finden(id);
    _fragebogen.Alle_Antworten_zurücksetzen(aufgabe);
    _fragebogen.Antwort_setzen(aufgabe, id);
    var (aufgaben, auswertbar) = _fragebogen.Auswertbarkeit_ermitteln();
    return (aufgaben, auswertbar);
}
```

Die Domänenlogik befindet sich in der Klasse *Fragebogen*.

```

public class Fragebogen
{
    private Aufgabe[] _aufgaben;

    public void Aufgaben_setzen(Aufgabe[] aufgaben) {
        _aufgaben = aufgaben;
    }

    public Aufgabe Zugehörige_Aufgabe_finden(int id) {
        foreach (var aufgabe in _aufgaben) {
            foreach (var antwortmöglichkeit in aufgabe.Antwortmöglichkeiten) {
                if (antwortmöglichkeit.Id == id) {
                    return aufgabe;
                }
            }
        }
        throw new ArgumentOutOfRangeException(
            $"Keine Antwortmöglichkeit mit Id = {id} gefunden.");
    }

    public void Alle_Antworten_zurücksetzen(Aufgabe aufgabe) {
        foreach (var antwortmöglichkeit in aufgabe.Antwortmöglichkeiten) {
            antwortmöglichkeit.IstGegeben = false;
        }
    }

    public void Antwort_setzen(Aufgabe aufgabe, int id) {
        foreach (var antwortmöglichkeit in aufgabe.Antwortmöglichkeiten) {
            if (antwortmöglichkeit.Id == id) {
                antwortmöglichkeit.IstGegeben = true;
            }
        }
    }

    public (IEnumerable<Aufgabe> aufgaben, bool) Auswertbarkeit_ermitteln() {
        foreach (var aufgabe in _aufgaben) {
            var aufgabeIstAuswertbar = false;
            foreach (var antwortmöglichkeit in aufgabe.Antwortmöglichkeiten) {
                if (antwortmöglichkeit.IstGegeben) {
                    aufgabeIstAuswertbar = true;
                }
            }
            if (!aufgabeIstAuswertbar) {
                return (_aufgaben, false);
            }
        }
        return (_aufgaben, true);
    }
}

```

Damit ist das Inkrement *Antwort gegeben* abgeschlossen. Der Product Owner kann erneut sein Feedback geben. Die UI verhält sich nun bereits

wie in den Anforderungen gewünscht. Es handelt sich um eine *Single Choice* Befragung, d.h. es kann jeweils genau eine Antwortmöglichkeit ausgewählt werden. Dieses Verhalten kann der Product Owner nun in der UI überprüfen. Ferner wird die *Auswerten* Schaltfläche erst aktiviert, wenn zu jeder Aufgabe genau eine Antwortmöglichkeit selektiert ist.

Es fehlt innerhalb von *Start* noch das Feature, dass jede Aufgabe die Antwortmöglichkeit "Weiß nicht" enthält. Diese kann leicht in Start ergänzt werden. Es muss lediglich eine Funktionseinheit in den Flow gestellt werden, die zu jeder Aufgabe eine solche Antwortmöglichkeit ergänzt. Des Weiteren fehlt die Auswertung des Fragebogens.

Feature "Weiß nicht" ergänzen

Im Fragebogen soll laut den Anforderungen zu jeder Aufgabe die Antwortmöglichkeit "Weiß nicht" vorgesehen werden. Diese Antwort steht allerdings nicht in der Datei `questionnaire.txt`, sondern sie wird von der Anwendung jeweils ergänzt. Dieses Feature ist Bestandteil der Interaktion *Start*. Die Antwortmöglichkeit muss innerhalb von *Start* zu jeder Aufgabe ergänzt werden. Das muss passieren, bevor die Antwortmöglichkeiten mit einer Id versehen werden. Andernfalls würde die Antwortmöglichkeit "Weiß nicht" jeweils keine Id erhalten.

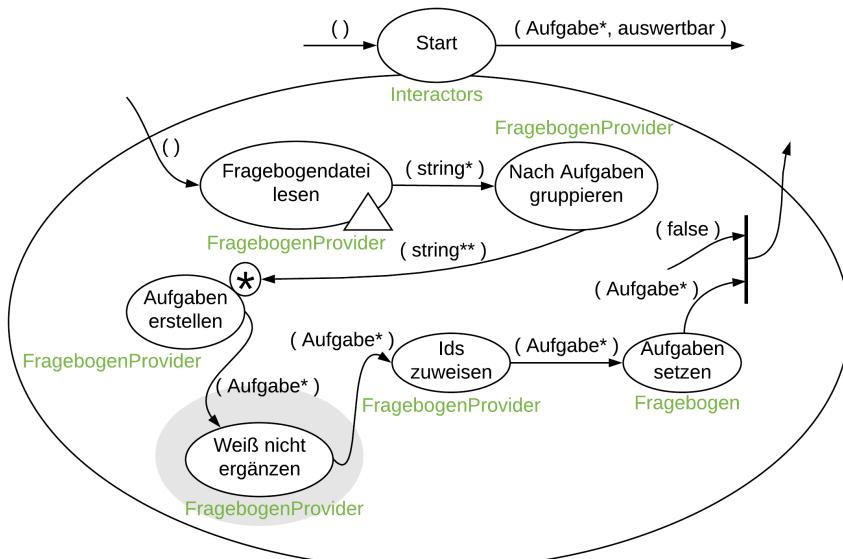


Abbildung 115: Angepasster Entwurf von Start

Nachdem alle Aufgaben erstellt sind, wird durch die Funktionseinheit *Weiß nicht ergänzen* jeweils die Antwortmöglichkeit "Weiß nicht" ergänzt. Die Implementation und Integration der Methode ist schnell erledigt.

```
public static void Weiß_nicht_ergänzen(Aufgabe[] aufgaben) {
    foreach (var aufgabe in aufgaben) {
        aufgabe.Antwortmöglichkeiten.Add(new Antwortmöglichkeit{
            Antwort = "Weiß nicht"
        });
    }
}
```

Verfeinerung von 'Auswerten'

Um die Interaktion *Auswerten* zu realisieren, muss zunächst wieder der Entwurf der obersten Ebene verfeinert werden.

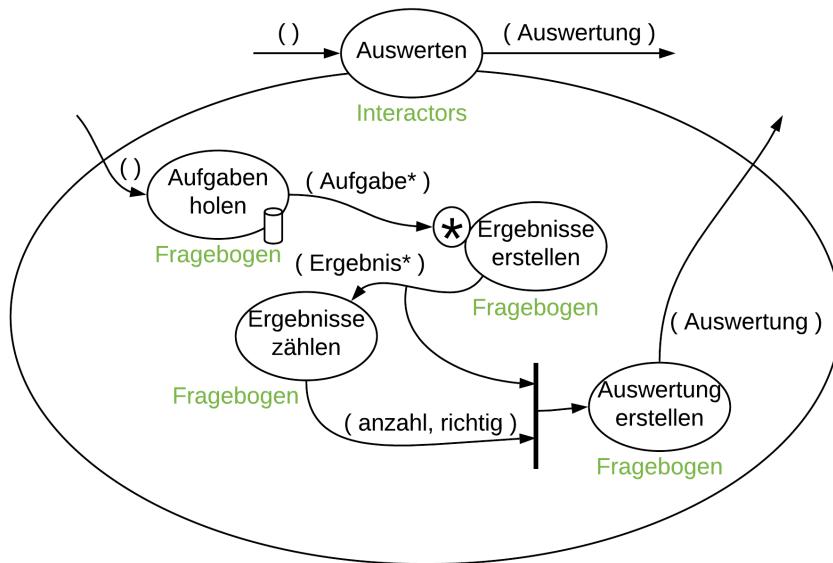


Abbildung 116: Verfeinerung des Entwurfs der Interaktion Auswerten

Die Funktionseinheit liefert ein Objekt vom Typ *Auswertung*. Dazu wird zunächst durch die Funktionseinheit *Aufgaben holen* auf die Liste der

Aufgaben zugegriffen. Diese liegen als Zustand in der Klasse *Fragebogen*. Anschließend kann die Funktionseinheit Ergebnisse erstellen zu jeder *Aufgabe* ein *Ergebnis* erstellen. Dazu prüft die Funktionseinheit, ob die Frage richtig beantwortet wurde und übernimmt alle Angaben in das Ergebnisobjekt. Im Anschluss zählt Ergebnisse zählen die Gesamtanzahl der Ergebnisse und damit der Aufgaben, sowie die richtig beantworteten. Zuletzt kann dann mit den beiden Anzahlen sowie den Ergebnissen ein Objekt vom Typ *Auswertung* erstellt werden.

Implementation

Die Implementation geht mit Hilfe des Entwurfs schnell von der Hand. In der Klasse *Interactors* wird die Methode *Auswerten* ergänzt.

```
public Auswertung Auswerten() {
    var aufgaben = _fragebogen.Aufgaben_holen();
    var ergebnisse = _fragebogen.Ergebnisse_erstellen(aufgaben);
    var (anzahl, richtig) = _fragebogen.Ergebnisse_zählen(ergebnisse);
    return _fragebogen.Auswertung_erstellen(ergebnisse, anzahl, richtig);
}
```

Anschließend können die benötigten Methoden in der Klasse *Fragebogen* ergänzt werden.

```
public IEnumerable<Aufgabe> Aufgaben_holen() {
    return _aufgaben;
}

public IEnumerable<Ergebnis> Ergebnisse_erstellen(IEnumerable<Aufgabe> aufgaben) {
    foreach (var aufgabe in aufgaben) {
        yield return Ergebnis_erstellen(aufgabe);
    }
}

private Ergebnis Ergebnis_erstellen(Aufgabe aufgabe) {
    return new Ergebnis {
        Frage = aufgabe.Frage,
        GegebeneAntwort = GegebeneAntwort(aufgabe),
        RichtigeAntwort = RichtigeAntwort(aufgabe),
        IstKorrekt = IstKorrekt(aufgabe)
    };
}
```

```

private static string GegebeneAntwort(Aufgabe aufgabe) {
    return aufgabe.Antwortmöglichkeiten.Find(a => a.IstGegeben).Antwort;
}

private static string RichtigeAntwort(Aufgabe aufgabe) {
    return aufgabe.Antwortmöglichkeiten.Find(a => a.IstKorrekt).Antwort;
}

private static bool IstKorrekt(Aufgabe aufgabe) {
    return aufgabe.Antwortmöglichkeiten.FirstOrDefault(
        a => a.IstKorrekt && a.IstGegeben) != null;
}

public (int anzahl, int richtig) Ergebnisse_zählen(
    IEnumerable<Ergebnis> ergebnisse) {
    var anzahl = ergebnisse.Count();
    var richtig = ergebnisse.Count(e => e.IstKorrekt);
    return (anzahl, richtig);
}

public Auswertung Auswertung_erstellen(
    IEnumerable<Ergebnis> ergebnisse,
    int anzahl,
    int richtig) {
    return new Auswertung {
        AnzahlAufgaben = anzahl,
        RichtigeAufgaben = richtig,
        Ergebnisse = ergebnisse.ToArray()
    };
}

```

Als nächstes kann die UI für die Auswertung erstellt werden. Die besteht aus einem simplen XAML Teil, in dem das Fenster über ein Grid in zwei Zeilen unterteilt wird. In die obere Zeile kommt ein *Label* Control, in die untere ein *ScrollViewer* mit einem *StackPanel*. *Label* und *StackPanel* werden benannt, um darauf in der Codebehind Datei zugreifen zu können.

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="64"/>
        <RowDefinition/>
    </Grid.RowDefinitions>

    <Label x:Name="_label" Grid.Row="0"/>
    <ScrollViewer Grid.Row="1" VerticalScrollBarVisibility="Auto">
        <StackPanel x:Name="_panel" Orientation="Vertical"/>
    </ScrollViewer>
</Grid>
```

In der Codebehind Datei wird durch die Methode *Update* eine Auswertung in die Controls geschrieben.

```
public void Update(Auswertung auswertung) {
    var prozent = auswertung.RichtigeAufgaben * 100.0 / auswertung.AnzahlAufgaben;
    _label.Content = $"Sie haben {auswertung.RichtigeAufgaben} von "+
        $"{auswertung.AnzahlAufgaben} richtig beantwortet ({prozent:F1}%)";
    foreach (var ergebniss in auswertung.Ergebnisse) {
        var label = new Label();
        label.Content = ergebniss.Frage + "\n ";
        if (ergebniss.IstKorrekt) {
            label.Content += $"Ihre Antwort '{ergebniss.GegebeneAntwort}' ist richtig.";
        }
        else {
            label.Content += $"Ihre Antwort '{ergebniss.GegebeneAntwort}' ist falsch. "+
                $"Richtig ist '{ergebniss.RichtigeAntwort}'." ;
        }
        _panel.Children.Add(label);
    }
}
```

Damit die neue Interaktion in der Anwendung zur Verfügung steht, muss sie noch in der Main Methode ergänzt werden. Der Ui Event *Auswerten* muss dazu verdrahtet werden.

```
ui.Auswerten += () => {
    var auswertung = interactors.Auswerten();
    var auswertungUi = new AuswertungUi();
    auswertungUi.Update(auswertung);
    auswertungUi.ShowDialog();
    Start();
};
```

Hier wird der Interactor angewiesen, eine Auswertung zu erstellen. Anschließend wird diese mit Hilfe der neuen Ui angezeigt. Die Auswertung wird modal über dem anderen Dialog angezeigt. Wird dieses Fenster geschlossen, wird erneut *Start* aufgerufen, um den Fragebogen zu leeren und erneut anzuzeigen.

Die folgende Abbildung zeigt eine Auswertung.

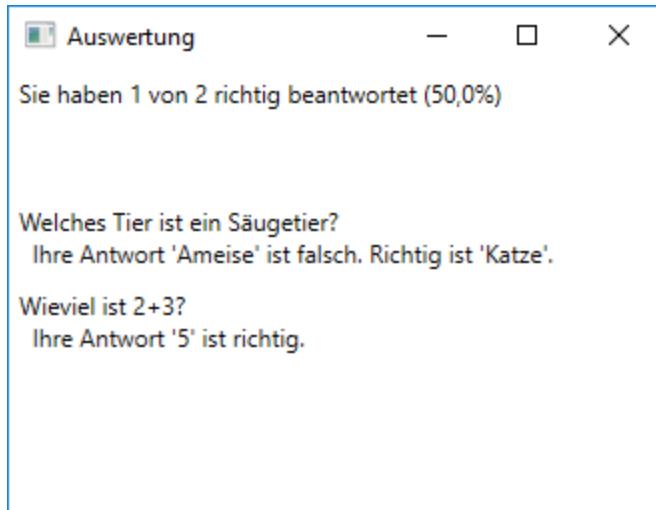


Abbildung 117: Eine Auswertung

Optisch kann an dieser Auswertung sicher noch einiges verbessert werden. Die Funktionalität ist damit jedoch bereits hergestellt.

Fazit

Damit ist die Questionnaire Anwendung fertig realisiert. Natürlich müsste optisch an der Anwendung noch geschliffen werden. Die Ui sieht noch sehr roh aus. Man konnte sehen, dass die Vorgehensweise in kleinen Inkrementen auch hier möglich war. Auch das Weglassen und spätere Ergänzen eines Features wurde demonstriert.

Streams

Manchmal sind die zu verarbeitenden Datenmengen zu groß, um sie in einem Schwung von einer zur nächsten Funktionseinheit weiterzureichen. Dazu müssten die Daten vollständig im Speicher liegen: eine Methode liefert die Daten, die nächste Methode empfängt die Daten. Sind die Daten zu groß, um sie vollständig im Speicher zu halten, muss ein Weg gefunden werden, sie schrittweise von einer zur nächsten Funktionseinheit zu übertragen.

Ein zweites Szenario, da mit Flow Design darstellbar sein muss, sind langlaufende Operationen. Bei diesen muss die Möglichkeit bestehen, den Anwender zwischenzeitig über den Fortschritt zu informieren. Auch für diesen Anwendungsfalls muss es möglich sein, dies in einem Entwurf darzustellen. Beides sind Fälle für einen Datenfluss in Form eines *Streams*.

Notation und Bedeutung

In der Regel ist die Beziehung zwischen eingehendem und ausgehendem Datenfluss eine 1:1 Beziehung. Für eine Nachricht, die in die Funktionseinheit hinein fließt, kommt am Ende genau eine Nachricht heraus. Eine Funktionseinheit, die zu einem gegebenen Verzeichnis die Dateinamen aller Quellcodedateien liefert, sieht im Entwurf wie folgt aus:

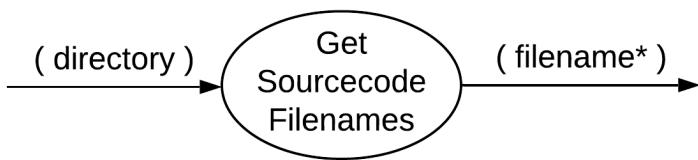


Abbildung 118: Ermitteln von Dateinamen

Die oben gezeigt Funktionseinheit *GetSourceCodeFilenames* liefert zu einem Verzeichnis eine Aufzählung von Dateinamen. Ein Aufruf der Funktion liefert genau ein Ergebnis. Die Aufzählung *filename** kann bei der Implementation durch ein Array oder eine Liste realisiert werden. Diese Form des Entwurfs führt zu einer Funktionseinheit, die zu einem gegebenen Verzeichnisnamen alle Dateinamen auf einmal ermittelt und dann auf einmal an den Aufrufer liefert. Wird eine ausreichend große Verzeichnisstruktur durchsucht, kann dies einige Sekunden dauern. In dieser Zeit ist die Anwendung blockiert. Ferner kann die Datenmenge am Ausgang der Funktionseinheit

heit größer werden, so dass eine relativ große Menge Speicher durch die Daten belegt wird. Gesucht ist nun eine Darstellungsform, die es ermöglicht, dem Nachfolger der Funktionseinheit einen Dateinamen nach dem anderen zu liefern. Dazu muss die Funktionseinheit an ihrem Ausgang einen *Stream* liefern.

Ein Stream ist ein Datenfluss, der mehrfach durchlaufen wird. Dies gilt bezogen auf den Eingang einer Funktionseinheit. Für einen einmaligen Zufluss von Daten fließen bei einem Stream mehrfach Daten am Ausgang der Funktionseinheit. Die folgende Abbildung zeigt dies am Beispiel der Funktionseinheit *GetSourcecodeFilenames*.

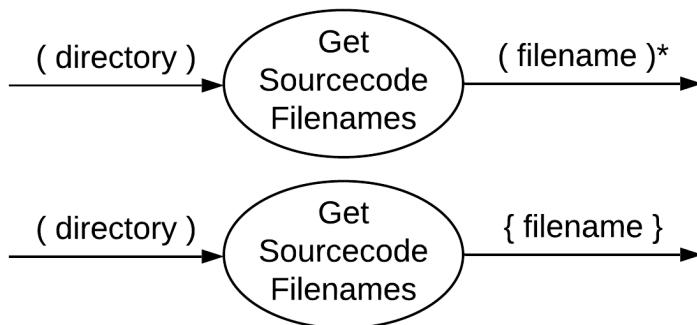


Abbildung 119: Ermitteln von Dateinamen als Stream

Die Funktionseinheit *GetSourcecodeFilenames* wird einmal aufgerufen und bei diesem Aufruf wird ihr ein Verzeichnis als Parameter übergeben. Als Ergebnis liefert die Funktionseinheit Dateiname für Dateiname als Datenstrom. Es ist wichtig zu berücksichtigen, dass das Sternchen hier außerhalb der Klammern steht. Alternativ können geschweifte Klammern verwendet werden. Die Bedeutung: für einen einzigen Aufruf mit einem Verzeichnis als Parameter fließt immer wieder ein einzelner Dateiname. Die beiden folgenden Datenflüsse unterscheiden sich also ganz wesentlich:

- (string*)
- (string)*

Im ersten Fall fließt einmal eine Aufzählung von Strings. Im zweiten Fall fließt mehrfach ein einzelner String. Für das Beispiel der Funktionseinheit *GetSourcecodeFiles* bedeutet dies, dass zu einem Verzeichnisnamen mehrere Dateinamen als ausgehender Datenfluss geliefert werden. Die nachfolgende Funktionseinheit erhält also mehrfach einen einzelnen Dateinamen geliefert. Damit erfolgt die weitere Bearbeitung Datei für Datei. Insbesondere blockiert die Funktionseinheit *GetSourcecodeFiles* nicht mehr die Anwendung. Sobald sie den ersten Dateinamen ermittelt hat, liefert sie diesen als Ergebnis an ihren Nachfolger.

Mehrere Ausgänge

In Verbindung mit Streams wird es häufig erforderlich sein, dass eine Funktionseinheit mehr als einen Ausgang hat. Im vorliegenden Beispiel der Funktionseinheit GetSourcecodeFiles ist es wichtig, dass die Funktionseinheit nach Aufzählung aller Dateinamen signalisiert, dass nun kein weiterer Dateiname mehr folgen wird. Die folgende Abbildung zeigt dies in einem Flow Design.

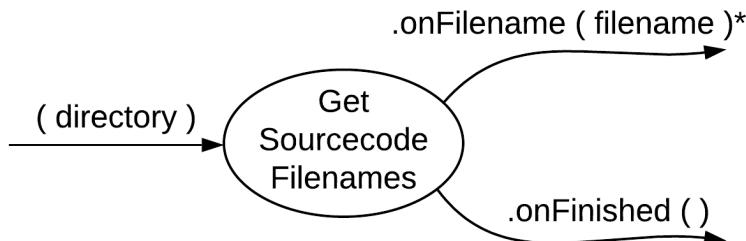


Abbildung 120: Zwei Ausgänge

Der eine Ausgang mit der Bezeichnung *onFilename* liefert jeweils einen Dateinamen. Der andere Ausgang mit dem Namen *onFinished* liefert das Signal, dass nun kein weiterer Dateiname folgen wird. Er signalisiert das Ende des Streams, auch *End of Stream* genannt.

Es stellt sich die Frage, wie eine solche Funktionseinheit in Code übersetzt werden kann. Der eingehende Datenfluss wird wieder zu einem Parameter der Methode. Doch wie wird der ausgehende Datenfluss übersetzt, der ja mehrfach stattfindet, bezogen auf einen einmaligen Aufruf der Methode? Und wie werden Funktionseinheiten mit mehreren Ausgängen in eine Methode übersetzt?

Eine Form der Übersetzung, die in den meisten Programmiersprachen zur Verfügung steht, ist die Verwendung von *Callbacks*. Der Methode werden nach den Eingangsparametern ein oder mehrere weitere Parameter übergeben. Diese zusätzlichen Parameter sind jeweils *Zeiger auf eine Funktion*. Diese Funktionszeiger können von der Methode aufgerufen werden, ohne dass die Methode damit den konkreten Nachfolger kennt.

Der Callback für einen Stream wird pro Element des Datenflusses aufgerufen. Im Folgenden werden die Übersetzungen in unterschiedlichen Programmiersprachen gezeigt.

Übersetzung in C#

In C# sieht die Übersetzung der Funktionseinheit *GetSourcecodeFilenames* wie folgt aus:

```
public static void GetSourcecodeFiles(string directory, Action<string> onFilename, Action onFinished) {
    var filenames = Directory.EnumerateFiles(directory, "*.cs", SearchOption.AllDirectories);
    foreach (var filename in filenames) {
        onFilename(filename);
    }
    onFinished();
}
```

Die Methode erhält als ersten Parameter das Verzeichnis, welches nach Dateien durchsucht werden soll. Als zweiter Parameter wird das Callback *onFilename* übergeben. Dieser Callback hat einen String Parameter, was in C# als *Action<string>* notiert wird. Für jeden Dateinamen, den die Methode im angegebenen Verzeichnis findet, wird diese Action aufgerufen. Dritter Parameter der Methode ist ein weiteres Callback. Dieses ist parameterlos, in C# vom Typ *Action*. Dieser Callback wird einmal am Ende der Methode aufgerufen, nachdem alle Dateinamen ermittelt wurden. Mit diesem Callback wird der Aufrufer darüber benachrichtigt, dass nun kein weiterer Aufruf des Callbacks *onFilename* mehr erfolgen wird. Ohne dieses Callback kann der Aufrufer nicht wissen, ob *onFilename* ein weiteres Mal aufgerufen wird. Ganz wichtig ist in diesem Beispiel zu beachten, dass die verwendete Methode *Directory.EnumerateFiles* die Liste der Dateinamen nicht vollständig auf einmal ermittelt und in den Speicher lädt. Würde die Methode ein Array *string[]* oder eine *List<string>* als Rückgabewert liefern, wäre nichts gewonnen. Dann würde die Methode *GetSourcecodeFiles* zwar nach außen einen Stream liefern, intern aber trotzdem die gesamten Daten auf einmal ermitteln und in den Speicher laden. Bei Angabe eines im Verzeichnisbaum weit oben liegenden Verzeichnisses würde die Methode dann einige Sekunden arbeiten und das laufende Programm solange blockieren. C# bietet mit dem Typ *IEnumerable<>* und den Schlüsselworten *yield return* syntaktische Unterstützung für Streams. Oben gezeigte Implementation verhält sich zur Laufzeit tatsächlich so, dass die aufgerufene Methode *Directory.EnumerateFiles* jeweils nur einen Dateinamen ermittelt und diesen an den Aufrufer übergibt. Das liegt daran, dass der Rückgabewert der Methode *Directory.EnumerateFiles* vom Typ *IEnumerable<string>* ist. Es handelt sich hier um eine verzögerte Ausführung, die zur Laufzeit erst dann tätig wird, wenn aus dem Enumerator ein Element entnommen wird. Die Implementation der Methode *GetSourcecodeFiles* erwartet zwei Callback Methoden. Das Callback *onFilename* wird für jeden gefundenen Dateinamen aufgerufen. Doch woran soll der Aufrufer erkennen, dass der letzte Dateiname gefunden wurde und nun kein weiterer Aufruf mehr folgt?

Dies ist die Aufgabe des zweiten Callbacks. Das Callback *onFinished* wird einmalig aufgerufen, nachdem alle Dateinamen abgeliefert wurden.

Das folgende Listing zeigt die Verwendung der Methode. Aus den Kommandozeilenparametern wird das Verzeichnis entnommen und als erster Parameter an die Methode *GetSourcecodeFiles* übergeben.

```
public static void Main(string[] args) {
    var directory = CommandLine.GetDirectory(args);
    CodefileProvider.GetSourcecodeFiles(directory,
        onFilename: filename => {
            Console.WriteLine(filename);
        },
        onFinished: () => {
            Console.WriteLine("Finished!");
        });
}
```

Als zweiter und dritter Parameter werden Lambda Expressions an die Methode *GetSourcecodeFiles* übergeben. Die erste wird für jeden ermittelten Dateinamen aufgerufen. Der Parameter *filename* der Lambda Expression enthält dann jeweils den Dateinamen. Im Beispiel oben wird der Dateiname lediglich auf der Konsole ausgegeben. Dritter und letzter Parameter ist eine weitere Lambda Expression, die aufgerufen wird, nachdem der letzte Dateiname ermittelt wurde. Diese Lambda Expression ist parameterlos.

Eine andere Variante für eine C# Implementation der Funktionseinheit *GetSourcecodeFiles* zeigt das folgende Listing.

```
public IEnumerable<string> GetSourcecodeFiles(string directory) {
    var filenames = Directory.EnumerateFiles(directory, "*.cs", SearchOption.AllDirectories);
    return filenames;
}
```

Hier gibt die Methode den von *Directory.EnumerateFiles* erhaltenen Rückgabewert an den Aufrufer zurück. Da die Framework Methode bereits ein *IEnumerable<string>* liefert und intern mittels *yield return* implementiert ist, führt das zum gewünschten Resultat. Die Ausführung erfolgt verzögert. Die Verwendung dieser Methode sieht sehr gewohnt aus:

```
public static void Main(string[] args) {
    var directory = CommandLine.GetDirectory(args);
    var filenames = CodefileProvider.GetSourcecodeFiles(directory);
    foreach (var filename in filenames) {
        Console.WriteLine(filename);
    }
    Console.WriteLine("Finished!");
}
```

Bei diesem Aufruf der Methode *GetSourcecodeFiles* sieht es so aus, als würden mit einem mal alle Dateinamen ermittelt und als Aufzählung geliefert. Tatsächlich kehrt die Methode zur Laufzeit sofort zum Aufrufer zurück und übergibt ihm einen Enumerator. Erst wenn diesem Elemente entnommen werden, beginnt das tatsächliche Ermitteln der Dateinamen. Im Beispiel wird also erst beim Durchlaufen der Aufzählung innerhalb der *foreach* Schleife das Ermitteln der Dateinamen angestoßen. Die Implementation entspricht exakt dem weiter oben gezeigten Entwurf.

Übersetzung in JavaScript

In JavaScript werden Callbacks an vielen Stellen verwendet. Vor allem in Verbindung mit node.js kommen Callbacks zum Einsatz, weil dort viele Funktionen asynchron angelegt sind. Sie liefern ihr Ergebnis nicht sofort als Rückgabewert. Stattdessen ermitteln diese Funktionen ihr Ergebnis im Hintergrund und rufen die Callback Funktion auf, sobald das Ergebnis zur Verfügung steht. Die Funktionseinheit *GetSourcecodeFiles* sieht in JavaScript wie folgt aus:

```
const walk = require("walk");
const path = require("path");

exports.GetSourcecodeFiles = function(directory, onFilename, onFinish) {
    let options = {
        followLinks: false
    };

    walker = walk.walk(path.resolve(directory), options);

    walker.on("file", function (root, fileStats, next) {
        let fullQualifiedName = path.join(root, fileStats.name);
        if(fullQualifiedName.endsWith(".js"))
            onFilename(fullQualifiedName);
        next();
    });

    walker.on("end", function() {
        onFinish();
    });
};
```

Die Verwendung der Methode *GetSourcecodeFiles* sieht wie folgt aus:

```

const commandline = require("./commandline");
const filesystemprovider = require("./filesystemprovider");
);
const codefileprovider = require("./codefileprovider");
const loc = require("./loc");
const ui = require("./ui");

const path = commandline.GetPath(process.argv);

filesystemprovider.GetSourcecodeFiles(path,
    filename => {
        let lines = codefileprovider.ReadFile(filename);
        let locstat = loc.CountLoc(filename, lines);
        ui.ShowLoc(locstat);
    },
    () => {
        ui.ShowSum();
    }
);

```

Erster Parameter ist der als Kommandozeilenparameter übergebene Verzeichnisname. Als zweiter Parameter wird eine Lambda Expression angegeben. Diese in JavaScript *Arrow-Function* genannte Funktion wird für jeden ermittelten Dateinamen aufgerufen. In diesem Fall wird der Dateinhalt eingelesen und anschließend werden die Anzahl der Codezeilen (Lines of Code = LOC) ermittelt und ausgegeben.

Als dritter Parameter wird eine Funktion angegeben, die zum Abschluss aufgerufen wird, nachdem *GetSourcecodeFiles* alle Dateinamen geliefert hat. Im Beispiel weist die Funktion die Ui an, eine Summe auszugeben.

Übersetzung in Java

In Java sieht die Implementation wie folgt aus:

```

public static void GetSourcecodeFiles(String directory, Consumer<String> onFilename, Runnable onFinished) {
    try {
        Files.walk(Paths.get(directory))
            .filter(path -> Files.isRegularFile(path) && path.toString().endsWith(".java"))
            .forEach(path -> onFilename.accept(path.getAbsolutePath().toString()));
        onFinished.run();
    } catch (Exception e) {
        throw new RuntimeException();
    }
}

```

Seit Java 8 können die Dateien eines Verzeichnisses mit der Methode `Files.walk` aufgelistet werden. Die Methode liefert einen Java Stream, der zunächst mit der `filter` Methode auf reguläre Dateien eingeschränkt wird. Ferner werden die Daten eingeschränkt auf Dateinamen, die auf ".java" enden.

Im Anschluss an den Filter wird der Stream mit `forEach` durchlaufen. Für jeden einzelnen Dateinamen wird dann `onFilename.accept` aufgerufen. Der Datentyp `Consumer<String>` entspricht dem .NET Typ `Action<string>`.

Nachdem alle Dateinamen durchlaufen wurden, wird das Callback `onFinished` aufgerufen. Dieses Callback ist parameterlos und vom Typ `Runnable`. Die gezeigte Implementation hat allerdings ein kleines Problem: die Auflistung der Dateinamen findet vollständig statt, bevor das erste Mal `onFilename.accept` aufgerufen wird. Es bleibt einem nichts anderes übrig, als das Durchsuchen selbst zu implementieren, damit so wenigstens verzeichnisweise gesucht wird.

```

public static void GetSourcecodeFiles(String directory, Consumer<String> onFilename, Runnable onFinished) {
    Walk(directory, onFilename);
    onFinished.run();
}

private static void Walk(String directory, Consumer<String> onFilename) {
    File root = new File(directory);
    File[] list = root.listFiles();

    if (list == null) return;

    for (File file : list) {
        if (file.isDirectory()) {
            Walk(file.getAbsolutePath(), onFilename);
        } else if (file.getAbsolutePath().endsWith(".java")){
            onFilename.accept(file.getAbsolutePath());
        }
    }
}

```

Die neue Implementation verwendet einen rekursiven Ansatz. `GetSourcecodeFiles` ruft die Methode `Walk` auf. Diese arbeitet rekursiv und ruft sich selbst auf, wenn sie auf ein Verzeichnis trifft. Wird ein Dateiname gefunden, wird das Callback `onFilename` aufgerufen.

Der Aufruf der Methode `GetSourcecodeFiles` ist in folgendem Listing zu sehen:

```

public static void main(String[] args) {
    ConsoleUi ui = new ConsoleUi();

    String directory = CommandLine.GetDirectory(args);
    FilesystemProvider.GetSourcecodeFiles(directory,
        filename -> {
            List<String> lines = CodefileProvider.ReadFile(filename);
            LocStat locstat = Loc.CountLoc(filename, lines);
            ui.ShowLoc(locstat);
        },
        () -> {
            ui.ShowSum();
        });
}

```

Hierbei handelt es sich um die *Main* Methode einer Anwendung, mit der die Codezeilen in einem Verzeichnis gezählt werden. Als erster Parameter wird das zu durchsuchende Verzeichnis übergeben. Es folgen dann zwei Lambda Expressions, die für jeden einzelnen Dateinamen bzw. am Ende aufgerufen werden. In der ersten Lambda Expression wird zum gefundenen Dateinamen ermittelt, wieviele Codezeilen die Datei enthält. Dazu wird die Datei zunächst eingelesen. Anschließend werden die gelesenen Zeilen von der *CountLoc* Methode gezählt. Ergebnis ist ein *LocStat* Objekt, welches den Dateinamen, die Anzahl der Codezeilen und die Gesamtanzahl der Zeilen enthält. Diese Information werden zuletzt auf der Console ausgegeben. In der zweiten Lambda Expression wird die Ui angewiesen, die ermittelten Summen auszugeben.

Übersetzung in C++

In C++ können Callbacks als Funktionszeiger realisiert werden. Seit der Einführung von Lambda Expressions wird die Verwendung syntaktisch deutlich vereinfacht.

```

#include <iostream>

void GetSourceCodeFiles(
    std::string directory,
    std::function<void(std::string filename)> onFilename,
    std::function<void()> onFinish) {
    onFilename("datei1.txt");
    onFilename("datei2.txt");
    onFilename("datei3.txt");
    onFinish();
}

int main() {
    GetSourceCodeFiles("./",
        [] (std::string filename) {
            std::cout << filename << std::endl;
        },
        [] () {
            std::cout << "Finished!" << std::endl;
        });
    return 0;
}

```

Das Listing zeigt die Methode `GetSourceCodeFiles`. Da es hier um die Syntax der Callbacks geht, ist die Methode nicht wirklich fertig implementiert. Sie liefert stattdessen immer die gleichen drei Dateinamen. In der `main` Methode ist der Aufruf von `GetSourceCodeFiles` zu sehen. Erster Parameter ist das Verzeichnis, welches durchsucht werden soll. Die beiden folgenden Parameter sind die beiden Callbacks. Die erste Lambda Expression wird für jeden gefundenen Dateinamen aufgerufen und gibt den Namen auf der Konsole aus. Die zweite Lambda Expression wird zum Abschluss aufgerufen, nachdem alle Dateinamen aufgezählt wurden. Die Ausgabe des Programms innerhalb von JetBrains CLion zeigt die folgende Abbildung.

```
streams x
/Users/stefanlieser/github/flowdesignbuch/cpp/streams/cmake-build-debug/streams
datei1.txt
datei2.txt
datei3.txt
Finished!

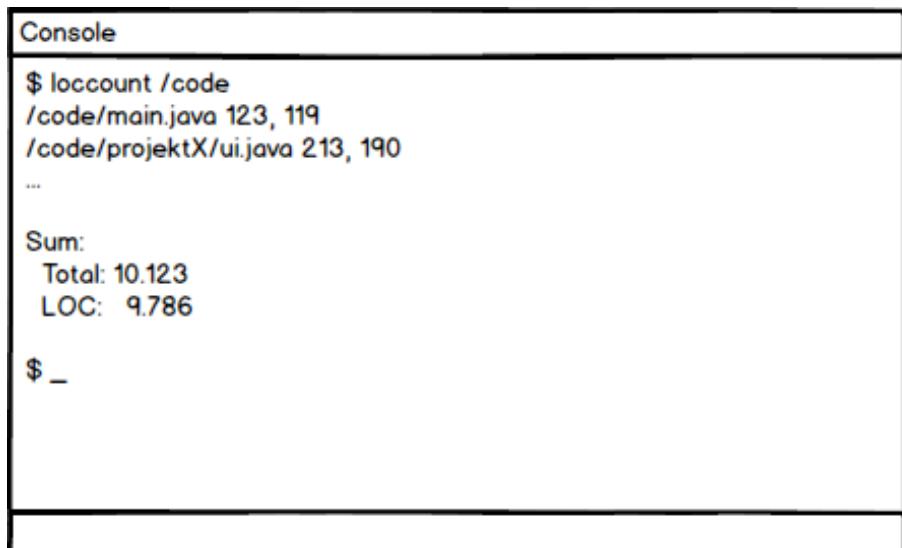
Process finished with exit code 0
```

Beispiel LOCcount

Nachdem im vorangegangen Kapitel die Streams eingeführt wurden, können diese nun in einem weiteren Beispiel verwendet werden.

Anforderungen

Um herauszufinden, wie viele Codezeilen sein aktuelles Projekt enthält, wünscht sich der Product Owner ein Kommandozeilenprogramm, welches die *Lines of Code* (LOC) zählt. Das Programm soll auf der Konsole aufgerufen werden mit einem Pfad als Parameter. Die folgende Abbildung zeigt beispielhaft die Bedienung des Programms.



The screenshot shows a terminal window titled "Console". The user has run the command \$ loccount /code. The output lists several Java files with their line counts: /code/main.java 123, 119, /code/projektX/ui.java 213, 190, followed by an ellipsis. Below this, the total sum is displayed: Sum: Total: 10.123 and LOC: 9.786. The prompt \$ _ is shown at the bottom.

```
$ loccount /code
/code/main.java 123, 119
/code/projektX/ui.java 213, 190
...
Sum:
Total: 10.123
LOC: 9.786
$ _
```

Abbildung 121: *loccount* auf der Konsole

Das Programm durchsucht das angegebene Verzeichnis inklusive aller Unterverzeichnisse nach Codedateien. Diese werden daran erkannt, dass die Dateinamen auf *.java* enden. Jede gefundene Datei wird mit ihrem voll qualifizierten Dateinamen aufgelistet. Ferner wird die Gesamtanzahl der Zeilen der Datei ausgegeben sowie die Anzahl der Codezeilen. Die Auflistung der einzelnen Dateien soll sofort erfolgen, nachdem sie im Verzeichnis gefunden wurden. Dem Product Owner ist es wichtig, während der Laufzeit des Programms den Fortschritt daran zu erkennen, dass immer wieder eine weitere Codedatei aufgelistet wird. Insbesondere beim Durchsuchen eines

großen Verzeichnisses mit vielen Unterverzeichnissen sollen die einzelnen Dateien schon aufgelistet werden, obwohl die Suche noch nicht beendet ist. Es darf also nicht am Anfang des Programmes zu einer langen Verzögerung kommen, weil zu diesem Zeitpunkt sämtliche Dateinamen auf einmal ermittelt werden.

Nachdem alle Quellcodedateien aufgelistet wurden, soll zuletzt jeweils die Summe der Gesamtanzahl und der Codezeilen ausgegeben werden.

Die Definition einer Codezeile ist wie folgt: Es werden alle Zeilen als Codezeilen gezählt, außer Kommentarzeilen und Leerzeilen. Kommentarzeilen zeichnen sich dadurch aus, dass sie mit einem doppelten Schrägstrich „//“ beginnen. Vor dem Kommentarzeichen kann auch Leerraum bestehend aus Leerzeichen und Tabulatoren stehen. Eine Leerzeile enthält keine Zeichen bzw. lediglich Leerzeichen und Tabulatoren. Folgende Beispiel werden nicht als Codezeilen gezählt:

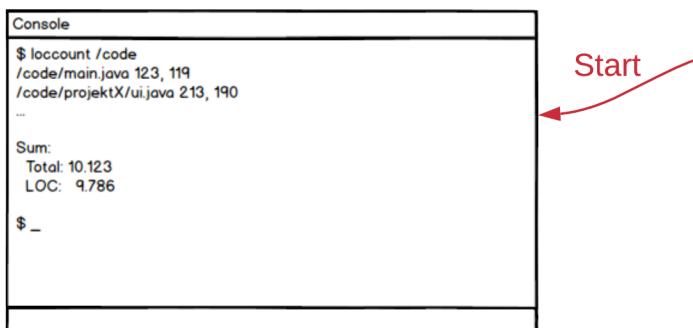
```
// this is a comment  
    // another comment
```

Die folgende Zeile wird als Codezeile gezählt, da der Kommentar nicht am Anfang der Zeile beginnt:

```
int a = 42;      // this is the magic answer
```

Dialoge und Interaktionen

Das Programm wird auf der Kommandozeile aufgerufen. Es erhält die benötigten Eingaben als Kommandozeilenparameter. Nach Aufruf des Programms arbeitet es seine Aufgabe ab und beendet sich dann wieder. Eine weitere Interaktion des Benutzers mit dem Programm ist nicht vorgesehen. Es gibt daher lediglich einen Dialog und die Interaktion *Start*.

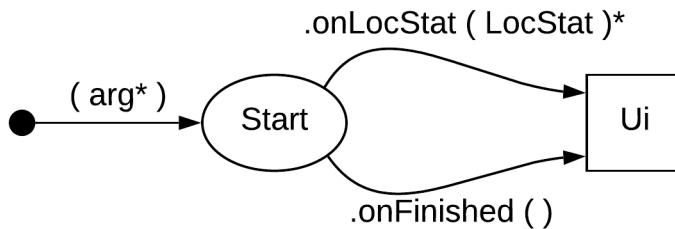


```
Console  
$ loccount /code  
/code/main.java 123, 119  
/code/projektX/ui.java 213, 190  
...  
  
Sum:  
Total: 10.123  
LOC: 9.786  
  
$ _
```

Abbildung 122: Dialog und Interaktion von *loccount*

Entwurf der obersten Ebene

Ausgehend von der Interaktion *Start* sieht der Entwurf der obersten Ebene der Anwendung wie folgt aus.



LocStat

Filename : string
Total : int
Loc : int

Abbildung 123: Entwurf der obersten Ebene von *loccount*

Das Programm erhält vom Betriebssystem die Kommandozeilenparameter liefert. Diese fließen zur Funktionseinheit *Count LOC*. Als Resultat liefert *Count LOC* zur *Ui* immer wieder *LocStat* Objekte. Diese enthalten den Dateinamen, die Gesamtanzahl Zeilen sowie die Codezeilen als Eigenschaften. Aufgabe der *Ui* ist es, diese Angaben auf der Konsole auszugeben. Ferner ist es Aufgabe der *Ui*, die beiden benötigten Summen zu bilden.

Nachdem *Count LOC* zu allen gefundenen Quellcodedateien ein zugehöriges *LocStat* Objekt liefert hat, signalisiert die Funktionseinheit der *Ui* das Ende der Streambearbeitung. Daraufhin kann die *Ui* die Summen ausgeben.

In diesem Entwurf ist ganz absichtlich beim Datenfluss zur *Ui* das Sternchen außerhalb der Klammern. Es bedeutet, dass der Datenfluss als Stream stattfindet. Es fließt hier also mehrfach ein *LocStat* Objekt zur *Ui*.

Verfeinerung

Die Funktionseinheit *Count LOC* ist dafür verantwortlich, zu den übergebenen Kommandozeilenparametern die *LocStat* Objekte zu erzeugen und zuletzt das Ende der Operation zu signalisieren. Die folgende Abbildung zeigt eine Verfeinerung des Entwurfs.

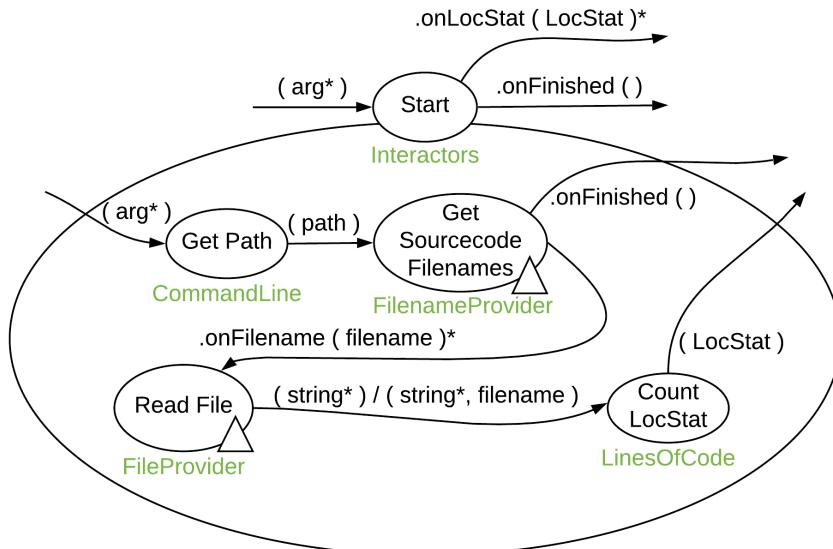


Abbildung 124: Verfeinerung von *Start*

Zunächst wird aus den Kommandozeilenargumenten *arg** der Pfad ermittelt. Dieser wird an die Funktionseinheit *GetSourcecodeFiles* übergeben. Die liefert alle passenden Dateinamen als Stream. Zum Abschluss liefert sie einmalig den Datenfluss *onFinished*, um so zu signalisieren, dass der Stream zu seinem Ende gekommen ist.

Mit jeweils einem Dateinamen wird die Funktionseinheit *ReadFile* aufgerufen. Sie liefert den Inhalt der Datei als Aufzählung von Strings. Dabei ist jede einzelne Zeile der Datei ein String in der Aufzählung. *ReadFile* liest tatsächlich die gesamte Datei in den Speicher und arbeitet nicht mit Streams. Die Strings werden im Anschluss von der Funktionseinheit *CountLocStat* gezählt. Ergebnis ist ein *LocStat* Objekt. Da im *LocStat* Objekt auch der Dateiname enthalten sein muss, wird dieser ebenfalls an *CountLocStat* geliefert. Syntaktisch wird hier im Flow ein Join in der Kurzschreibweise verwendet. Während *ReadFile* nur die Strings liefert, erwartet *CountLocStat* sowohl die Strings als auch den Dateinamen. Der Datenfluss zwischen den beiden Funktionseinheiten lautet daher

```
( string* ) / ( string*, filename )
```

Dies bedeutet, der Vorgänger liefert lediglich *string**, während der Nachfolger ein Tupel aus *string** und *filename* erhält. Weitere Details zu Joins und der Kurznotation finden Sie unter Join im Referenzteil II des Buches.

Implementation in Java

Das folgende Listing zeigt die oberste Ebene einer Java Implementation, die *main* Methode.

```
public static void main(String[] args) {
    ConsoleUi ui = new ConsoleUi();

    CountLOC(args, locstat -> ui.ShowLoc(locstat), () -> ui.ShowSum());
}

private static void CountLOC(String[] args, Consumer<LocStat> onLocStat, Runnable onFinish) {
    String path = CommandLine.GetPath(args);
    FilesystemProvider.GetSourcecodeFiles(path,
        filename -> {
            List<String> lines = CodefileProvider.ReadFile(filename);
            LocStat locstat = Loc.CountLocStat(filename, lines);
            onLocStat.accept(locstat);
        },
        () -> {
            onFinish.run();
        });
}
```

Zunächst wird eine Instanz der Klasse *ConsoleUi* erzeugt. Die Funktionalität der Benutzerschnittstelle ist hier als Instanz ausgelegt, da die Benutzerschnittstelle für die Summenbildung verantwortlich ist. Sie benötigt daher Zustand und kann somit nicht sinnvoll als statische Methode realisiert werden. Die Methode *CountLOC* und die von ihr verwendeten Operationen *CommandLine.GetPath*, *FilesystemProvider.GetSourcecodeFilenames* etc. sind dagegen statisch realisiert, da sie keinen Zustand benötigen. Die Methode *CountLOC* realisiert den weiter vorne gezeigten Entwurf. Es handelt sich um eine Integration, da diese Methode andere Methoden der Lösung auruft, diese also integriert. Aufgabe der Methode ist es, den im Entwurf gezeigten Datenfluss herzustellen.

Die Benutzerschnittstelle ist sehr einfach, da es sich um eine Konsolenanwendung handelt.

```

public class ConsoleUi {
    private LocStat sum = new LocStat();

    public void ShowLoc(LocStat locstat) {
        sum.Loc += locstat.Loc;
        sum.Total += locstat.Total;
        System.out.println(locstat.Filename + ", " + locstat.Loc + ", " + locstat.Total);
    }

    public void ShowSum() {
        System.out.println();
        System.out.println("Sum:");
        System.out.println(" LOC: " + sum.Loc);
        System.out.println(" Total: " + sum.Total);
    }
}

```

Die Methode *ShowLoc* wird jeweils für ein *LocStat* Objekt aufgerufen. Ihre Aufgabe ist es, die Summe zu aktualisieren und das *LocStat* Objekt auf der Konsole auszugeben.

Aufgabe der Klasse *CommandLine* ist es, aus den Kommandozeilenparametern einen Verzeichnisnamen zu ermitteln.

```

public class CommandLine {
    public static String GetPath(String[] args) {
        return args.length > 0 ? args[0] : "./";
    }
}

```

Ist auf der Kommandozeile kein Verzeichnisname übergeben worden, liefert die Methode mit dem Defaultstring „./“ das aktuelle Verzeichnis.

Die Methode *GetSourcecodeFiles* ist weiter oben bereits ausführlich besprochen worden. Die Methode ruft die Methode *Walk* auf, die rekursiv für jedes Unterverzeichnis arbeitet.

```

public static void GetSourcecodeFiles(String directory, Consumer<String> onFilename, Runnable onFinish) {
    Walk(directory, onFilename);
    onFinish.run();
}

```

Die eigentliche Arbeit wird durch die rekursive Methode *Walk* übernommen.

```

private static void Walk(String directory, Consumer<String> onFilename) {
    File root = new File(directory);
    File[] list = root.listFiles();

    if (list == null) return;

    for (File file : list) {
        if (file.isDirectory()) {
            Walk(file.getAbsolutePath(), onFilename);
        } else if (file.getAbsolutePath().endsWith(".java")){
            try {
                onFilename.accept(file.getCanonicalPath());
            } catch (IOException e) {
                throw new RuntimeException(e);
            }
        }
    }
}

```

Sie durchsucht das als ersten Parameter übergebene Verzeichnis. Für jede Datei deren Dateiname auf .java endet ruft sie den Callback *onFilename* auf. Ist ein gefundener Name ein Unterverzeichnis, ruft die Methode sich rekursiv selbst auf.

Die Klasse *CodefileProvider* ist dafür zuständig, zu einem Dateinamen der Inhalt der Datei zu liefern.

```

public class CodefileProvider {
    public static List<String> ReadFile(String filename) {
        try {
            return Files.readAllLines(Paths.get(filename));
        } catch (IOException e) {
            throw new RuntimeException();
        }
    }
}

```

Um die ermittelten Daten an die Benutzerschnittstelle übergeben zu können, ist der Datentyp *LocStat* mit den benötigten Eigenschaften wie folgt definiert:

```
public class LocStat {  
    public String Filename;  
  
    public long Loc;  
  
    public int Total;  
}
```

Die Domänenlogik der Anwendung befindet sich in der Klasse *Loc*. Diese Klasse enthält die Logik, um Codezeilen zu zählen.

```
public class Loc {  
    public static LocStat CountLocStat(String filename, List<String> lines) {  
        LocStat result = new LocStat();  
  
        result.Filename = filename;  
        result.Total = lines.size();  
        result.Loc = lines  
            .stream()  
            .filter(line -> !isLoc(line))  
            .count();  
  
        return result;  
    }  
  
    private static boolean isLoc(String line) {  
        String trimmedLine = line.trim();  
        return trimmedLine.isEmpty() || trimmedLine.startsWith("//");  
    }  
}
```

Die Methode *CountLocStat* erhält den Dateinamen und den Inhalt der Datei als Parameter geliefert. Als Rückgabewert liefert die Methode ein *LocStat* Objekt. Das eigentliche Zählen der Codezeilen wird durch Java Streams mit Hilfe der Methoden *filter* und *count* realisiert. Zunächst werden mit *filter* die Zeilen gefiltert, die nicht gezählt werden sollen. Anschließend werden die verbleibenden Zeilen mit *count* gezählt.

Abschließende Betrachtung

Durch die Verwendung von Streams in einem Entwurf kann die Ausführung der einzelnen Funktionseinheiten so gestaltet werden, dass diese elementweise arbeiten. Statt einer Funktionseinheit eine komplette Auflistung von

Elementen zu übergeben, erhält sie jeweils ein einzelnes Element. Damit ist es möglich, potentiell unendlich große Datenmengen zu bearbeiten. Diese könnten nicht auf einmal in den Speicher geladen werden. Durch elementweises abarbeiten ist es jedoch möglich, sehr große Datenmengen zu bearbeiten und dies in einem Entwurf darzustellen.

Neben großen Datenmengen bieten sich Streams dort an, wo der Anwender während der Bearbeitung einer länger laufenden Operation regelmäßig Feedback benötigt. Liefert eine Funktionseinheit die Daten elementweise an ihren Nachfolger, kann dieser Datenfluss auch verwendet werden, um elementweise eine Rückmeldung an den Anwender zu geben.

Fallunterscheidungen

Manchmal muss in einem Entwurf eine Fallunterscheidung vorgenommen werden. Je nach dem, welcher Fall vorliegt, wird der Datenfluss auf die eine oder andere Weise fortgesetzt. Die Fallunterscheidung selbst wird dabei von einer Funktionseinheit getroffen. Diese hat dann entweder einen optionalen Ausgang oder mehrere Ausgänge. Bei einem optionalen Ausgang wird der Datenfluss anhand der Entscheidung entweder fortgesetzt oder an der Stelle beendet. Mehrere mögliche Ausgänge bedeuten, dass der Datenfluss auf die eine oder andere Weise fortgesetzt wird. Im folgenden wird beschrieben, wie Fallunterscheidungen in flow Design dargestellt werden. Ferner wird gezeigt, wie diese implementiert werden können.

Darstellung im Entwurf

Die einfachste Möglichkeit einer Fallunterscheidung ist ein optionaler Ausgang.

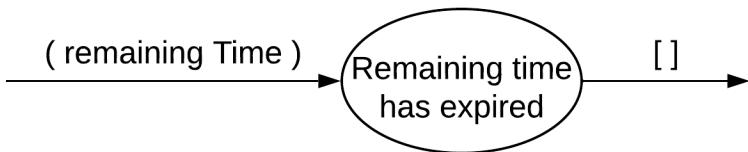


Abbildung 125: Optionaler Ausgang als simple Fallunterscheidung

Die Darstellung bedeutet, dass die Funktionseinheit *RemainingTimeHasExpired* feststellt, ob die Weckzeit erreicht ist. Nur wenn das der Fall ist, wird der ausgehende optionale Datenfluss durchlaufen. Ist die Weckzeit noch nicht erreicht, findet dieser Datenfluss nicht statt. Optionale Datenflüsse können ganz allgemein als Stream dargestellt werden. Präziser ist jedoch die Angabe mit eckigen Klammern, da diese klarer ausdrücken, dass der Datenfluss garnicht oder einmal stattfindet.

Eine Fallunterscheidung kann auch mehrere Ausgänge beinhalten, wie folgendes Beispiel zeigt.

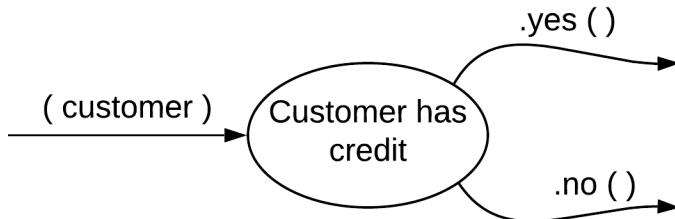


Abbildung 126: Fallunterscheidung mit zwei Ausgängen

Die Funktionseinheit *CustomerHasCredit* prüft, ob der Kunde kreditwürdig ist. Wenn das der Fall ist, geht es auf dem oberen Datenfluss weiter und der Kunde kann per Kreditkarte zahlen. Im anderen Fall geht es unten weiter und der Kunde muss in bar zahlen.

Bei einer einfachen Fallunterscheidung mit einem optional ausgehenden Datenfluss, muss dieser nicht näher bezeichnet werden. Führt die Fallunterscheidung jedoch zu mehr als einem Ausgang, müssen die Ausgänge benannt werden. In der Abbildung oben sind die Ausgänge mit *.yes* bzw. *.no* bezeichnet. Die Bezeichner sollten sich exakt so in der Implementation wieder finden.

Implementation

Für die Implementation von Fallunterscheidungen gilt es eine Besonderheit zu berücksichtigen: die Funktionseinheit führt mal zu einem ausgehenden Datenfluss, ein anderes mal jedoch nicht. Das bedeutet, der Datenfluss ist optional. Wenn wir die Funktionseinheit als Methode bzw. Funktion übersetzen möchten stellt sich die Frage, wie wir den optional ausgehenden Datenfluss erreichen können. Ausgehende Datenflüsse wurden bislang als Rückgabewert einer Funktion übersetzt. Wenn eine Methode jedoch einen Rückgabewert mit *return* liefert, muss sie dies immer tun. Es gibt keine optionalen Rückgaben. Wir greifen daher wieder zum Konzept des *Callbacks*, wie das schon bei Streams der Fall ist.

```

public void WhenAlarmTimeArrives(TimeSpan remainingTime, Action onAlarmTime) {
    if (remainingTime.TotalSeconds <= 0) {
        onAlarmTime();
    }
}

```

Das Listing zeigt die Methode *WhenAlarmTimeArrives* wie sie im Abschnitt zuvor als Flow dargestellt war. Eingehend erhält die Funktionseinheit ein *TimeSpan* Objekt, welches die Restzeit bis zum Wecken darstellt. Der optional ausgehende Datenfluss wird als weiterer Parameter an die Methode übergeben. Da dieser Datenfluss keine Daten transportiert, wird der Delegatetyp *Action* verwendet. Die Methode ruft diese Action optional auf, nämlich dann, wenn die Restzeit abgelaufen ist.

```
public void Alarm() {
    var remainingTime = GetRemainingTime();
    WhenAlarmTimeArrives(remainingTime,
        () => {
            Console.WriteLine("Alarm!");
        });
}
```

Deutlicher wird das, wenn wir die Verwendung der Methode sehen. Beim Aufruf erhält sie als ersten Parameter die Restzeit. Als zweiten Parameter erhält sie eine Lambda Expression. Der Code der Lambda Expression wird nur dann ausgeführt, wenn die Restzeit abgelaufen ist. Im Beispiel oben wird dann "Alarm!" auf der Konsole ausgegeben.

Eine alternative Implementation kommt ohne Funktionszeiger und Lambda Expression aus. Das folgende Listing zeigt die Methode *AlarmTimeArrives*. Hier wurde ein etwas anderer Bezeichner für die Methode verwendet, damit der Quellcode an der Verwendungsstelle besser zu lesen ist. Die Methode erhält als Parameter wieder den eingehenden Datenfluss, hier also wieder die berechnete Restzeit.

```
public bool AlarmTimeArrives(TimeSpan remainingTime) {
    return remainingTime.TotalSeconds <= 0;
}
```

Statt nun einen Callback aufzurufen, liefert die Methode lediglich einen *bool* zurück. Das Ergebnis ist *true*, wenn die Restzeit abgelaufen ist. Damit ist nun die integrierende Methode dafür verantwortlich, den Nachfolger dieser Funktionseinheit aufzurufen.

```
public void Alarm() {
    var remainingTime = GetRemainingTime();
    if (AlarmTimeArrives(remainingTime)) {
        Console.WriteLine("Alarm!");
    }
}
```

Hier steht nun in der Integrationsmethode ein *if*. Abhängig vom Ergebnis des Aufrufs von *AlarmTimeArrives* wird der optionale Nachfolger nun aufgerufen oder nicht. Diese Form der Implementation einer Fallunterscheidung wird von vielen Entwicklern als natürlicher angesehen. Gleichzeitig lauert hier eine Herausforderung. Häufig wird nämlich darauf verzichtet, einen so simplen Ausdruck, wie ihn die Funktion *AlarmTimeArrives* darstellt, in eine Funktion auszulagern. Viele Entwickler halten die folgende Variante für gleichwertig:

```
public void Alarm() {
    var remainingTime = GetRemainingTime();
    if (remainingTime.TotalSeconds <= 0) {
        Console.WriteLine("Alarm!");
    }
}
```

Auf diese Weise entstehen zwei Probleme. Erstens kann nun der Ausdruck *remainingTime.TotalSeconds <= 0*

nicht isoliert getestet werden. Der Ausdruck steht innerhalb des *if* Statements in der Methode *Alarm*. Die einzige Möglichkeit, diesen Ausdruck in einem automatisierten Test auszuführen ist nun, die *Alarm* Methode aufzurufen. Doch diese liefert keinen in einem automatisierten Test überprüfbaren Rückgabewert, sondern führt manchmal zu dem Seiteneffekt einer Konsoleausgabe. Die Konsoleausgabe könnte mit vertretbarem Aufwand automatisiert getestet werden, doch bliebe dabei die Frage, warum man den Code nicht in einer Weise schreibt, dass er besser testbar ist.

In meinen Trainings stelle ich mich an dieser Stelle meist auf ein Bein und schildere den Teilnehmern, dass ich mir ein Bein hochbinden und so zum Bahnhof hüpfen könnte. Aber warum sollte ich das tun, wenn ich stattdessen auch bequem auf zwei Beinen laufen kann? Die gleiche Frage stellt sich in Bezug auf die Testbarkeit in oben gezeigtem Code. Warum sollte ich den Ausdruck in die Methode schreiben, wenn er doch herausgezogen in eine Funktion viel leichter zu testen ist?

Das zweite Argument dagegen, den Ausdruck direkt in die Methode *Alarm* zu schreiben, ist eine IOSP (Integration Operation Segregation Principle) Verletzung. Die Methode *Alarm* integriert den Aufruf von *GetRemainingTime*. Damit gehört sie zur Kategorie der *Integration*. Der Ausdruck des *if* Statements ist allerdings ein Detail und fällt somit in die Kategorie der *Operation*. Damit ist die Zuständigkeit der Methode *Alarm* nicht klar Integration oder Operation, sondern von beidem etwas. Das erschwert die Lesbarkeit der Methode. Die Lösung besteht übrigens nicht darin, nun auch noch den Code der Methode *GetRemainingTime* in die Methode *Alarm* direkt aufzunehmen. Dann wäre der Code von *Alarm* nur noch auf einem niedrigen Abstraktionsniveau und damit schwerer verständlich. Methodenaufrufe führen zu einem höheren Abstraktionsniveau und damit zu besserer Lesbarkeit. Des weiteren wäre das SRP (Single Responsibility Principle) verletzt, wenn alle Details in der Methode *Alarm* stehen würde. Dann wäre sie nämlich sowohl dafür verantwortlich, die Restzeit zu berechnen, als auch die Entscheidung zu treffen, als auch die Konsoleausgabe vorzunehmen.

Zurück zur Implementation von Fallunterscheidungen. Wenn ein Team von Entwicklern sich dagegen ausspricht, Fallunterscheidungen mit Callbacks zu realisieren, muss es konsequent darauf achten, dass in den Integrationsmethoden im *if* Statement kein Ausdruck steht. Dort darf dann lediglich ein Funktionsaufruf stehen, der isoliert getestet werden kann.

Liegen bei einer Fallunterscheidung mehr als ein Ausgang vor, erfolgt die Implementation mit Callbacks im Prinzip so, wie bei einem einzelnen Ausgang. Jeder Ausgang wird zu einem Callback der Methode. Das folgende Listing zeigt dies am Beispiel der Funktionseinheit *WhenCustomerHasCredit*.

```
public void WhenCustomerHasCredit(Customer customer, Action onYes, Action onNo) {
    if (customer.NumberOfTransactions > 5) {
        onYes();
    } else {
        onNo();
    }
}
```

Die Verwendung dieser Funktionseinheit setzt wieder auf den Einsatz von Lambda Expressions.

```
public void DoCheckout() {
    var customer = new Customer();
    WhenCustomerHasCredit(customer,
        onYes: () => {
            // ...
        },
        onNo: () => {
            // ...
        });
}
```

Hier wurden die in C# möglichen expliziten Bezeichner der Parameter *onYes* und *onNo* verwendet und deutlich zu machen, für welchen Ausgang der Funktionseinheit die jeweilige Lambda Expression zuständig ist.

In diesem Fall stellt sich erneut die Frage, wie eine Implementation aussieht, die auf Callbacks und Lambda Expressions verzichtet. Bei zwei möglichen Ausgängen könnte erneut eine boolesche Entscheidungsmethode zum Einsatz kommen. Davon ist jedoch in jedem Fall abzuraten, weil an der Verwendungsstelle schnell die Frage aufkäme, welche Bedeutung *true* und *false* als Rückgabewert haben. Spätestens bei mehr als zwei Ausgängen muss ohnehin zu einem *enum* gegriffen werden. Daher lautet die

Empfehlung, bei Fallunterscheidungen sofort einen *enum* einzuführen, weil damit der Code leichter verständlich wird.

```
public enum Credibility
{
    HasNoCredit,
    HasCredit
};

public Credibility GetCustomerCredibility(Customer customer) {
    return customer.NumberOfTransactions > 5
        ? Credibility.HasCredit
        : Credibility.HasNoCredit;
}
```

Nun ist an der Verwendungsstelle ein *switch* Statement erforderlich, um nicht nur auf die beiden definierten Fälle zu reagieren, sondern auch Fälle mit einer Exception zu berücksichtigen, die im Code vergessen wurden.

```
public void DoCheckout() {
    var customer = new Customer();
    var credibility = GetCustomerCredibility(customer);
    switch (credibility) {
        case Credibility.HasNoCredit:
            // ...
            break;
        case Credibility.HasCredit:
            // ...
            break;
        default:
            throw new ArgumentOutOfRangeException();
    }
}
```

Hier zeigt sich die Schwachstelle von *enum* und *switch*. Der Code ist länger und fehleranfälliger als bei Verwendung von Callbacks und Lambda Expressions. Ein Callback in der Signatur einer Methode kann beim Aufruf nicht vergessen werden, da der Compiler uns zwingt, alle Parameter anzugeben.

Ein weiterer *enum* Eintrag ist schnell definiert. Doch die *switch* Statements sind damit noch nicht alle gleich identifiziert. Auch hier muss das Team also in Code Reviews sehr genau arbeiten und darauf achten, dass *enum* und *switch* jeweils zusammen passen.

Alternativen wieder zusammenführen

Nach einer Fallunterscheidung ist es oftmals sinnvoll, zwei Datenflüsse wieder zusammenzuführen. Die Fallunterscheidung sorgt dafür, dass Details je nach vorliegendem Fall speziell behandelt werden. Anschließend kann häufig eine gemeinsame Fortsetzung sinnvoll sein, in dem die vorher verzweigten Datenflüsse nun wieder zusammengeführt werden. Wie dies im Entwurf dargestellt wird und wie die zugehörige Implementation aussieht, wird im folgenden dargestellt.

Darstellung im Entwurf

Die folgende Abbildung zeigt, wie zwei Datenflüsse nach einer Fallunterscheidung wieder zusammengeführt werden können. In diesem abstrakten Beispiel wird durch die Funktionseinheit f eine Fallunterscheidung getroffen. Abhängig von der Entscheidung wird der Datenfluss entweder bei A oder B fortgesetzt. Resultat ist in beiden Fällen ein k . Unabhängig davon, ob das k von A oder B produziert wurde, wird es im Anschluss von C weiterbearbeitet.

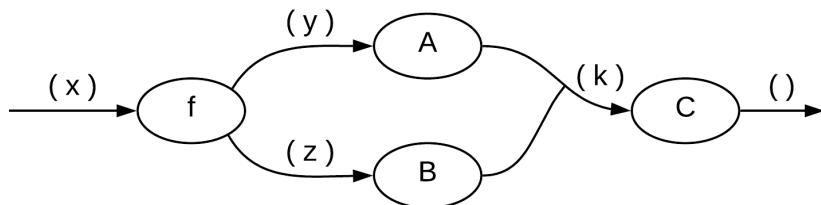


Abbildung 127: Zusammenführen von Datenflüssen

Das Zusammenführen ist im Entwurf dadurch dargestellt, dass zwei Datenflüsse zusammenlaufen und in einem einzigen Datenfluss fortgeführt werden. Zeichnerisch ist dies einfach darstellbar. Es gilt lediglich darauf zu achten, dass die beiden zusammengeführten Datenflüsse exakt den gleichen Datentyp transportieren müssen. Andernfalls ist es syntaktisch nicht möglich, sie zusammenzuführen. Spätestens bei der Implementation würde ansonsten auffallen, dass es zu einem Typkonflikt kommt. Die folgende Abbildung zeigt an einem Beispiel, wie das Zusammenführen nicht funktionieren kann.

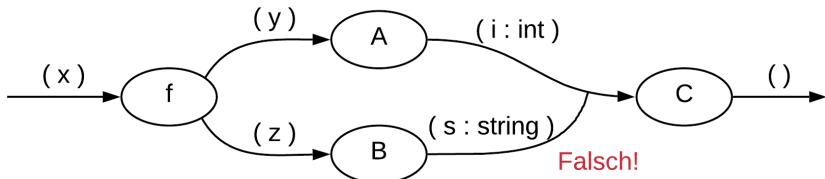


Abbildung 128: Fehlerhaftes Zusammenführen verschiedener Datentypen

Hier wird auf dem einen Zweig ein *int* produziert, während es auf dem anderen Zweig ein *string* ist. Eine solche Zusammenführung von Datenflüssen mit unterschiedlichem Typ ist syntaktisch nicht möglich. Hier würde sich bei der Implementation die Frage ergeben, von welchem Typ der Parameter von *C* sein soll.

Das Problem unterschiedlicher Typen könnte dadurch gelöst werden, dass bei einem der beiden Datenflüsse eine Typkonvertierung vorgenommen wird. Allerdings müsste dann trotzdem sichergestellt sein, dass die beiden zusammengeführten Datenflüsse inhaltlich dasselbe liefern. Zwei Datenflüsse auf denselben Datentyp zu konvertieren führt nicht zwangsläufig dazu, dass die nachfolgende Funktionseinheit mit beiden Inhalten etwas anfangen kann. Sind die Inhalte gleich und nur der Typ unterschiedlich, würde sich die Frage stellen, aus welchem Grund die beiden Datenflüsse einen unterschiedlichen Typ verwenden. Es geht also beim Zusammenführen nicht primär darum, auf denselben Typ zu achten, sondern es muss inhaltlich das gleiche fließen.

Implementation

Die Implementation einer Zusammenführung ist wenig spektakulär. Mal liefert die eine Funktionseinheit einen Wert, mal die andere. Der Nachfolger wird jedenfalls mit einem Wert aufgerufen, der durch den einen oder den anderen Vorgänger produziert wurde.

```
public void Integration(int z) {  
    var k = 0;  
    f(z,  
        onX: x => k = A(x),  
        onY: y => k = B(y));  
    C(k);  
}
```

Dieses Listing zeigt das abstrakte Beispiel aus dem Abschnitt zuvor. Hier trifft *f* die Fallunterscheidung. Ausgehend von dieser Entscheidung wird entweder *A* oder *B* aufgerufen. Beide produzieren ein *k*, welches zuletzt von *C* weiterbearbeitet wird. Die Zusammenführung der Datenflüsse findet dadurch statt, dass zuletzt *C* aufgerufen wird und zwar unabhängig davon, in welchem Zweig das *k* produziert wurde.

Beispiel Haushaltsbuch

Das nachfolgende Beispiel verdeutlicht nochmal den Umgang mit Fallunterscheidungen.

Anforderungen

Es soll eine Konsolenanwendung erstellt werden, mit der man seine Einnahmen und Auszahlungen verwalten kann. Ziel der Anwendung ist es, einen Überblick darüber zu erhalten, welche Beträge man in unterschiedlichen Kategorien wie Miete, Lebenshaltung, etc. monatlich ausgibt. Die Anwendung wird auf der Konsole mit Kommandozeilenparameter aufgerufen. Folgendes Beispiel zeigt einen typischen Lauf der Anwendung.



The screenshot shows a terminal window titled "Debug — -bash — 78x25". The window contains the following command-line session:

```
$ mono haushaltsbuch.exe einzahlung 4000
Kassenbestand: 4000,00 EUR
: 0,00 EUR
$ mono haushaltsbuch.exe einzahlung 300
Kassenbestand: 4300,00 EUR
: 0,00 EUR
$ mono haushaltsbuch.exe auszahlung 400 Miete
Kassenbestand: 3900,00 EUR
Miete: 400,00 EUR
$ mono haushaltsbuch.exe übersicht 9 2018
Miete: 400,00 EUR
$ mono haushaltsbuch.exe auszahlung 120 Lebenshaltung
Kassenbestand: 3780,00 EUR
Lebenshaltung: 120,00 EUR
$ mono haushaltsbuch.exe auszahlung 340 Reisen
Kassenbestand: 3440,00 EUR
Reisen: 340,00 EUR
$ mono haushaltsbuch.exe übersicht 9 2018
Miete: 400,00 EUR
Lebenshaltung: 120,00 EUR
Reisen: 340,00 EUR
$
```

Abbildung 129: Beispieldaten der Anwendung

Die Anwendung verfügt über drei Kommandos, die jeweils als erster Kommandozeilenparameter angegeben werden. Je nach Kommando werden weitere Parameter benötigt, manche sind optional. Die Syntax der Kommandozeilenparameter ist wie folgt:

haushaltsbuch übersicht [Monat Jahr]

Das Kommando *Übersicht* gibt eine Übersicht über die Ausgaben in den verwendeten Kategorien aus. Ferner wird der aktuelle Kassenstand angezeigt. Werden Monat und Jahr weggelassen, wird der aktuelle Monat aus dem Tagesdatum herangezogen. Wenn Monat und Jahr angegeben werden, müssen beide Parameter angegeben werden.

haushaltsbuch einzahlung [Datum] Betrag

Mit dem Kommando *Einzahlung* kann ein Betrag auf den Kassenstand aufaddiert werden. Der Betrag muss als Parameter angegeben werden. Optional kann das Datum der Einzahlung angegeben werden. Wird es weggelassen, wird das aktuelle Tagesdatum verwendet.

haushaltsbuch auszahlung [Datum] Betrag Kategorie
[Memo]

Durch das Kommando *Auszahlung* wird ein Betrag vom Kassenstand abgezogen und einer Kategorie zugeordnet. Zu einer Auszahlung muss der Betrag und die Kategorie angegeben werden. Optional kann das Datum der Auszahlung angegeben werden. Wenn es weggelassen wird, wird das aktuelle Tagesdatum verwendet. Optional kann ferner ein Memotext angegeben werden. Dieser wird im Programm nicht weiter verwendet, soll aber persistiert werden, so dass er später bei Bedarf herangezogen werden kann.

Interaktionsdiagramm

Nachdem die Anforderungen geklärt sind, geht es im nächsten Schritt darum, die Interaktionen zu identifizieren. Auf den ersten Blick könnte es so aussehen, dass es drei Interaktionen gibt: *Einzahlung*, *Auszahlung* und *Übersicht*.

```

$ mono haushaltsbuch.exe einzahlung 4000
Kassenbestand: 4000,00 EUR
: 0,00 EUR
$ mono haushaltsbuch.exe einzahlung 300
Kassenbestand: 4300,00 EUR
: 0,00 EUR
$ mono haushaltsbuch.exe auszahlung 400 Miete
Kassenbestand: 3900,00 EUR
Miete: 400,00 EUR
$ mono haushaltsbuch.exe übersicht 9 2018
Miete: 400,00 EUR
$ mono haushaltsbuch.exe auszahlung 120 Lebenshaltung
Kassenbestand: 3780,00 EUR
Lebenshaltung: 120,00 EUR
$ mono haushaltsbuch.exe auszahlung 340 Reisen
Kassenbestand: 3440,00 EUR
Reisen: 340,00 EUR
$ mono haushaltsbuch.exe übersicht 9 2018
Miete: 400,00 EUR
Lebenshaltung: 120,00 EUR
Reisen: 340,00 EUR
$ 

```

Einzahlung →

Auszahlung →

Übersicht →

Abbildung 130: Interaktionsdiagramm mit drei Interaktionen

Die Interaktion wird in diesem Fall durch das Kommando identifiziert, welches der Benutzer als Parameter auf der Kommandozeile angibt. Eine Alternative besteht darin, lediglich das Starten der Anwendung als Interaktion zu betrachten.

```

$ mono haushaltsbuch.exe einzahlung 4000
Kassenbestand: 4000,00 EUR
: 0,00 EUR
$ mono haushaltsbuch.exe einzahlung 300
Kassenbestand: 4300,00 EUR
: 0,00 EUR
$ mono haushaltsbuch.exe auszahlung 400 Miete
Kassenbestand: 3900,00 EUR
Miete: 400,00 EUR
$ mono haushaltsbuch.exe übersicht 9 2018
Miete: 400,00 EUR
$ mono haushaltsbuch.exe auszahlung 120 Lebenshaltung
Kassenbestand: 3780,00 EUR
Lebenshaltung: 120,00 EUR
$ mono haushaltsbuch.exe auszahlung 340 Reisen
Kassenbestand: 3440,00 EUR
Reisen: 340,00 EUR
$ mono haushaltsbuch.exe übersicht 9 2018
Miete: 400,00 EUR
Lebenshaltung: 120,00 EUR
Reisen: 340,00 EUR
$ 

```

Start →

Abbildung 131: Interaktionsdiagramm mit nur einer Interaktion

Die Anforderungen innerhalb einer *Start* Interaktion zu modellieren, hat den Vorteil, dass das Zerlegen der Kommandozeilenparameter ebenfalls innerhalb von *Start* stattfindet. Bei der Variante mit drei getrennten Interaktionen stellt sich die Frage, wo die Verantwortlichkeit liegt, die Kommandozeilenparameter zu interpretieren. Irgendwo muss die Entscheidung getroffen

werden, ob es sich um eine Einzahlung, Auszahlung oder die Übersicht handelt.

Entwurf der obersten Ebene

Die oberste Ebene des Entwurfs wird in der folgenden Abbildung gezeigt.

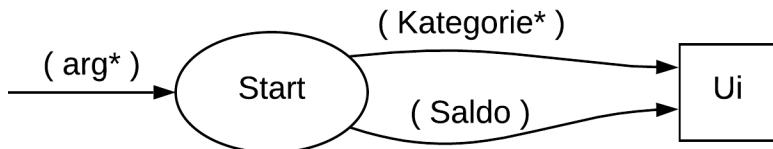


Abbildung 132: Oberste Ebene des Entwurfs

Der Interactor *Start* erhält die Kommandozeilenparameter als Eingabe. Ergebnis ist entweder eine Liste von Kategorien oder ein Saldo. Die Liste der Kategorien wird geliefert, wenn der Benutzer die Übersicht gewählt hat. Bei einer Ein- oder Auszahlung wird dagegen der aktuelle Saldo der betroffenen Kategorie geliefert.

Bei Fehlbedienung des Programms kommt es hier noch zu einem undefinierten Verhalten. Diese Fehlerbehandlung wird später ergänzt. Im ersten Schritt geht es darum, die gewünschte Funktionalität herzustellen. Die verwendeten Datentypen sind in den folgenden Tabellen zu sehen:

Kategorie

```
-----  
Bezeichnung : string  
Betrag : double
```

Saldo

```
-----  
Bezeichnung : string  
Betrag : double  
Saldo : double
```

Verfeinerung von 'Start'

Die Aufgabe von *Start* liegt darin, die Kommandozeilenparameter zu interpretieren und dann die zugehörige Domänenlogik auszuführen. Die folgende Abbildung zeigt die Verfeinerung der Interaktion *Start*.

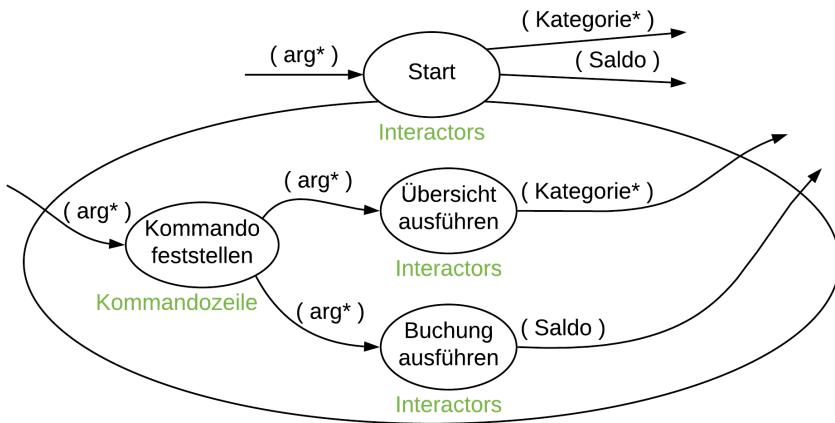


Abbildung 133: Verfeinerung von Start

Die Funktionseinheit *Kommando feststellen* prüft anhand des ersten Parameters, ob es sich um die Übersicht oder um eine Buchung handelt. Als Buchung gilt sowohl die Einzahlung als auch die Auszahlung. An die Funktionseinheit *Übersicht ausführen* werden die verbleibenden Kommandozeilenparameter weitergegeben. Der erste Parameter "Übersicht" wird entfernt, da er ja bereits interpretiert wurde. Da die Funktionseinheit *Buchung ausführen* feststellen muss, ob es sich um eine Einzahlung oder eine Auszahlung handelt, werden ihr alle Parameter übergeben.

Verfeinerung von 'Buchung ausführen'

Ist durch die Funktionseinheit *Kommando feststellen* entschieden worden, dass es sich um eine Buchung handelt, muss die zugehörige Logik ablaufen. Bei einer Buchung kann es sich um eine Einzahlung oder eine Auszahlung handeln.

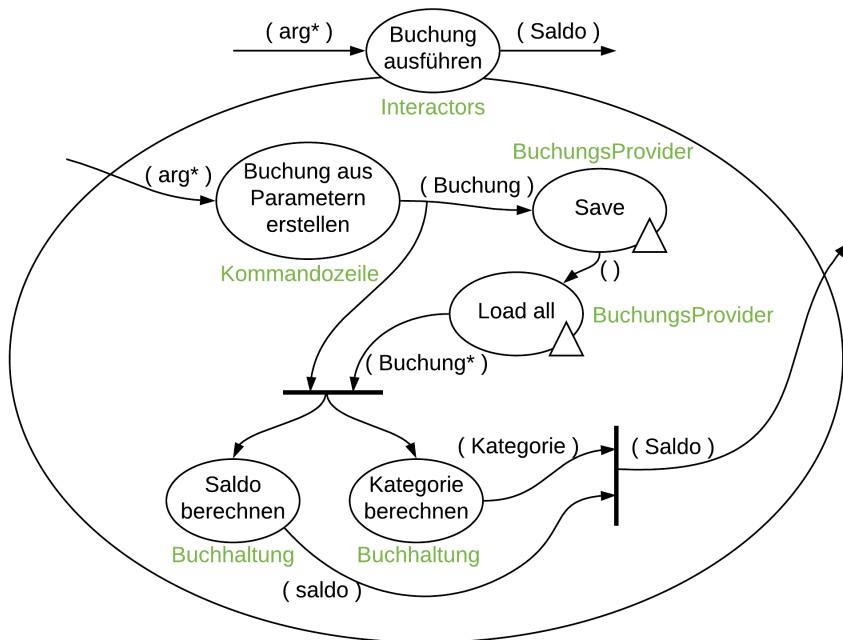


Abbildung 134: Verfeinerung des Entwurfs für *Buchung ausführen*

Zunächst wird ausgehend von den Kommandozeilenargumenten ein Datensatz vom Typ *Buchung* erstellt. Dieser neue Datensatz wird anschließend durch die Funktionseinheit *Save*persistiert. Die vorliegende Implementation verwendet dazu eine CSV Datei, in der die einzelnen Buchungssätze Zeile für Zeile abgelegt werden.

Nachdem der neue Datensatz persistiert wurde, werden durch *Load all* alle Buchungsdatensätze eingelesen. Die vollständige Liste aller Buchungssätze sowie der gerade neu hinzugefügte Datensatz gehen im Anschluss an die Funktionseinheit *Saldo berechnen*. Der aktuelle Buchungssatz ist erforderlich, um daraus das Datum der Buchung zu ermitteln. In Verbindung mit sämtlichen Buchungssätzen kann so der Saldo bezogen auf das Buchungsdatum ermittelt werden. Es werden einfach alle Einzahlungen aufaddiert und alle Auszahlungen abgezogen. Dies gilt nur für Datensätze, die vor dem Buchungsdatum liegen.

Als nächstes wird die Summe für die bei der Buchung verwendete Kategorie berechnet. Die Funktionseinheit *Kategorie berechnen* erhält dazu die aktuelle Buchung sowie die komplette Liste der Buchungssätze. Auch hier dient die aktuelle Buchung dazu, auf das Buchungsdatum zuzugreifen. Ferner wird die bei der Buchung verwendete Kategorie benötigt, um zu dieser Kategorie die Summe zu berechnen.

Der Datentyp *Buchung* ist wie folgt definiert.

Buchung

```
Buchungstyp : { Einzahlung, Auszahlung }
Buchungsdatum : DateTime
Betrag : double
Kategorie : string
Memo : string
```

Abbildung 135: Der Datentyp *Buchung*

Das Erstellen der Buchung aus den Kommandozeilenparametern übernimmt die Funktionseinheit *Buchung aus Parametern erstellen*. Der folgende Entwurf zeigt eine Verfeinerung dieser Funktionseinheit.

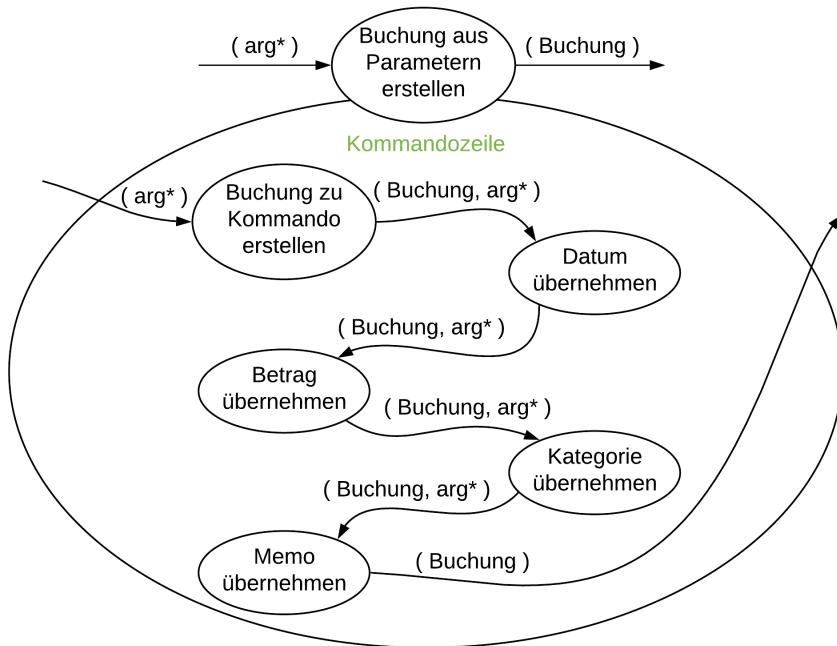


Abbildung 136: Verfeinerung des Entwurfs für *Buchung aus Parametern erstellen*

Zunächst wird eine *Buchung* erstellt. In dieser ist lediglich der *Buchungstyp* gesetzt. Dazu wird das erste Argument aus der Liste der Kommandozeilenparameter interpretiert. Ergebnis ist ein Buchungsobjekt sowie die Liste der Parameter. Allerdings enthält die Liste der Parameter nun nicht mehr das

Kommando. Die Arbeitsweise ist hier so, dass jede Funktionseinheit den Anfang der Kommandozeilenparameter betrachtet und den Wert interpretiert. Werte die von einer Funktionseinheit verwendet wurden, werden in das Buchungsobjekt übertragen. Ferner werden sie nicht mehr an den Nachfolger weitergereicht. Die Aufzählung *arg** wird so von den einzelnen Funktionseinheiten immer weiter verkürzt.

Verfeinerung von 'Übersicht ausführen'

Die nächste größere Funktionseinheit *Übersicht ausführen* ist dafür zuständig, eine Übersicht über die Ausgaben der Kategorien bezogen auf einen Monat zu erstellen.

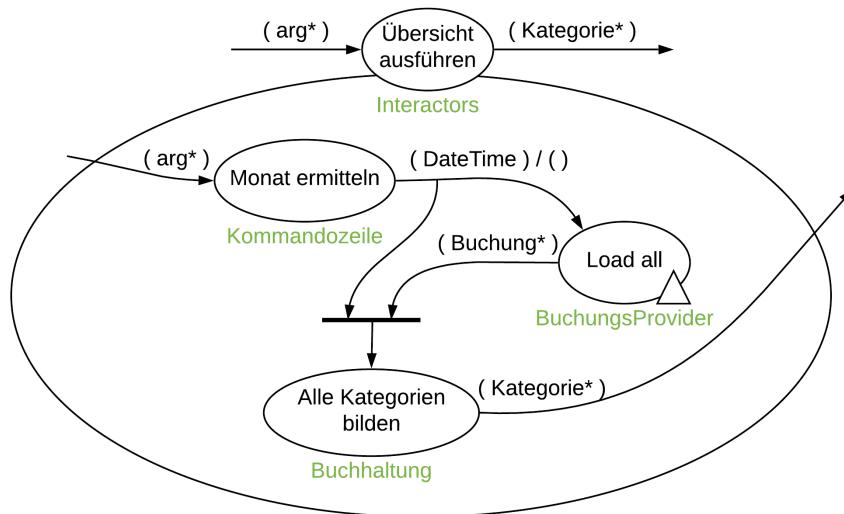


Abbildung 137: Verfeinerung des Entwurfs für *Übersicht ausführen*

Dazu erhält die Funktionseinheit als eingehenden Datenfluss die Kommandozeilenparameter *arg**. Zu beachten ist hier, dass der erste Parameter, das Kommando, bereits aus den Parametern entfernt wurde. Hier gilt wieder die Annahmen, dass eine Funktionseinheit, welche einen Parameter interpretiert, diesen aus der Liste der Parameter entfernt. So muss sich im Anschluss keine andere Funktionseinheit erneut mit dem Parameter befassen.

Zunächst werden aus den Parametern *Monat* und *Jahr* ermittelt. Dies ist erforderlich, um herauszufinden, auf welchen Monat sich die Übersicht über die Ausgaben beziehen soll. Im Anschluss werden sämtliche Buchungssätze aus der Datei *buchungen.csv* eingelesen. Da die

Funktionseinheit *Load all* das ermittelte Datum nicht als Eingabe benötigt, kommt hier wieder einmal die Split Syntax zum Einsatz. Der Ausdruck (*DateTime*) / () besagt, dass der Vorgänger ein *DateTime* Objekt liefert, der Nachfolger allerdings ohne Parameter aufgerufen wird. Dabei geht das *DateTime* Objekt natürlich nicht verloren. Es wird an den Join weitergereicht, so dass die Funktionseinheit *Alle Kategorien bilden* als Eingabe den Monat sowie alle Buchungssätze erhält. Ergebnis ist eine Liste von Kategorien. Diese enthalten jeweils die Bezeichnung sowie den Betrag.

Implementation

Die Umsetzung der Entwürfe soll hier nur angerissen werden. Die vollständige Implementation liegt im GitHub Repository des Buches unter <https://github.com/slieser/flowdesignbuch/tree/master/csharp/haushaltbuch>.

Die oberste Ebene der Anwendung befindet sich in der *Main* Methode.

```
private static void Main(string[] args) {
    Interactors.Start(args,
        kategorien => ConsoleUi.Übersicht_anzeigen(kategorien),
        saldo => ConsoleUi.Kategorie_anzeigen(saldo));
}
```

Hier wird die oberste Ebene des Entwurfs abgebildet. Es erfolgt hier die Fallunterscheidung, ob die Übersicht oder die Kategorie der Buchung angezeigt werden soll. Die nächste Ebene der Integration findet in der Klasse *Interactors* statt. Hier der Code der *Start* Methode:

```

public static void Start(string[] args,
    Action<IEnumerable<Kategorie>> onKategorien,
    Action<Saldo> onSaldo) {
    Kommandozeile.Kommando_feststellen(args,
        a => {
            var kategorien = Übersicht_ausführen(a);
            onKategorien(kategorien);
        },
        a => {
            var saldo = Buchung_ausführen(a);
            onSaldo(saldo);
        });
}

```

Weiter geht es dann mit den Methoden *Übersicht_ausführen* und *Buchung_ausführen*.

```

internal static Saldo Buchung_ausführen(IEnumerable<string> args) {
    var buchung = Kommandozeile.Buchung_aus_Parametern_errechnen(args);
    Buchungsprovider.Save(buchung);
    var buchungen = Buchungsprovider.Load_All();

    var saldo = Buchhaltung.Saldo(buchung, buchungen);
    var kategorie = Buchhaltung.Kategorie_berechnen(buchung, buchungen);
    return new Saldo {
        TheSaldo = saldo,
        Bezeichnung = kategorie.Bezeichnung,
        Betrag = kategorie.Betrag
    };
}

```

```

internal static IEnumerable<Kategorie> Übersicht_ausführen(
    IEnumerable<string> args) {
    var monat = Kommandozeile.Monat_ermitteln(args);
    var buchungen = Buchungsprovider.Load_All();
    var kategorien = Buchhaltung.Alle_Kategorien_bilden(buchungen, monat);
    return kategorien;
}

```

Auch hier werden die Entwürfe getreu abgebildet. Die Methoden gehören im Sinne des IOSP zur Kategorie der Integration. Sie integrieren weitere

Funktionseinheiten. Die meisten davon sind Operationen, mit Ausnahme der Methode *Buchung_aus_Parametern_erstellen*, die wiederum eine Integration darstellt. Auch diese Struktur spiegelt den Entwurf wieder.

Asynchrone Ausführung

Ein Flow Design Entwurf ist grundsätzlich als *synchron sequentiell* anzusehen. Zu einem gegebenen Zeitpunkt wird jeweils nur genau eine Funktionseinheit ausgeführt. Ferner werden die Funktionseinheiten in der im Flow angegebenen Reihenfolge sequentiell nacheinander ausgeführt. Doch auch asynchrone Vorgänge sind mit Flow Design darstellbar.

Threads einfärben

Werden mehrere Funktionseinheiten asynchron ausgeführt, bedeutet dies, dass zu einem gegebenen Zeitpunkt mehrere Funktionseinheiten gleichzeitig laufen. Dazu sind mehrere Threads erforderlich. Die gleichzeitige Ausführung von Funktionseinheiten bietet Vorteile. Gleichzeitig steigt damit aber auch die Komplexität. Insofern wird man nur an den Stellen zu einer asynchronen Ausführung greifen, an denen damit eine nicht-funktionale Anforderung umgesetzt wird. So darf die Benutzerschnittstelle bei langlaufenden Operationen natürlich nicht einfrieren. Folglich muss die langlaufende Operation im Hintergrund auf einem anderen Thread ausgeführt werden, als die Benutzerschnittstelle, um diese nicht zu blockieren. In diesem Fall ergibt sich die Notwendigkeit zu Multithreading aus der nicht-funktionalen Anforderung.

Die asynchrone Ausführung ist im realen Leben der Standard. Wenn ich per Telefon eine Pizza bestelle, läuft der Vorgang des Pizzabackens parallel zu meinem Leben ab. Mein Lebensfaden wird nicht angehalten, bis die Pizza geliefert ist. Das wäre bei synchroner Ausführung der Fall: eine Funktionseinheit gibt die Kontrolle ab an eine andere. Bei asynchroner Ausführung dagegen laufen mehrere Funktionseinheiten gleichzeitig. Die Gleichzeitigkeit entsteht dadurch, dass ein weiterer *Thread* eröffnet wird. Soll eine Funktionseinheit asynchron gegenüber einer anderen ausgeführt werden, werden dazu mehrere Threads benötigt. Die wesentliche Information in einem Entwurf ist daher, auf welchem Thread eine Funktionseinheit ausgeführt wird. Bei allen bislang gezeigten Entwürfen werden sämtliche Funktionseinheiten auf ein und demselben Thread ausgeführt: *synchron sequentiell*; jeweils nur einer, einer nach dem anderen.

Eine weitere wichtige Eigenschaft der Datenflüsse in Flow Design Diagrammen ist, dass mit den Daten auch die Kontrolle fließt. In der folgenden Abbildung produziert *f1* ein *x*. Dieses fließt zur Weiterverarbeitung zur Funktionseinheit *f2*.

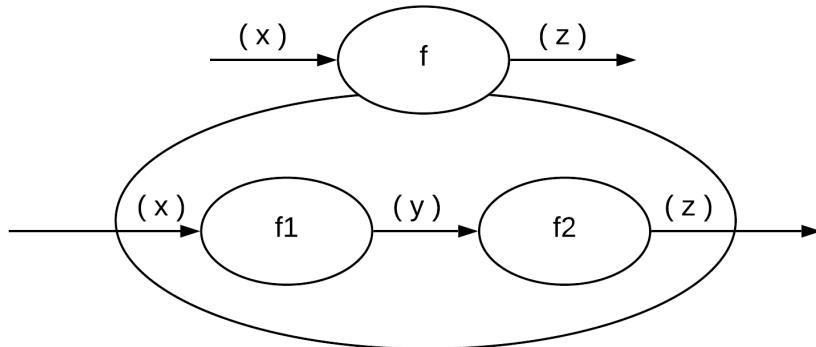


Abbildung 138: Synchron sequentielle Ausführung

Flow Design Diagramme sind Datenflussdiagramme. Das bedeutet, die Pfeile drücken den Fluss der Daten aus. Die Implementation dieser Diagramme erfolgt *synchron sequentiell*. Das bedeutet: zu einem Zeitpunkt hat immer nur genau eine Funktionseinheit die Kontrolle. Im Beispiel oben hat zunächst die Methode f die Kontrolle. Die Methode f ruft dann die Methode f_1 auf. Damit hat f nun die Kontrolle abgegeben an f_1 . Das bedeutet, f_1 wird ausgeführt, f ist angehalten. Irgendwann gibt f_1 seine Kontrolle wieder ab. Damit hat nun wieder f die Kontrolle. Im Anschluss ruft f nun f_2 auf, das Spiel wiederholt sich. In flow Design Diagrammen fließt also mit den Daten auch die Kontrolle.

Im asynchronen Fall sieht die Situation anders aus. Jetzt können f_1 und f_2 parallel ausgeführt werden. Dazu muss allerdings ein weiterer Thread ins Spiel kommen. In Flow Design werden Funktionseinheiten und Datenflüsse eingefärbt, um auf diese Weise mehrere Threads unterscheiden zu können.

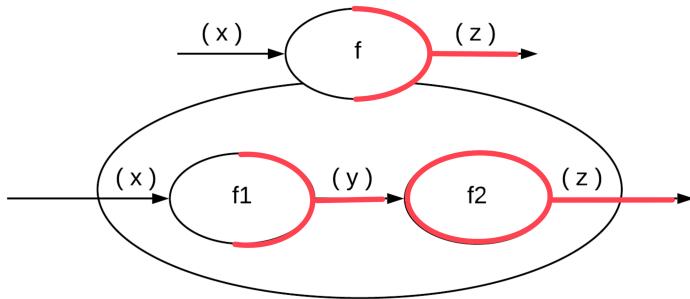


Abbildung 139: Funktionseinheiten auf mehreren Threads

In der Abbildung ist f_1 in zwei Farben dargestellt, um so auszudrücken, dass innerhalb von f_1 ein Threadwechsel stattfindet. Zunächst erhält f_1 die Kontrolle auf dem "schwarzen Thread", dem Mainthread. Innerhalb von f_1 wird dann ein weiterer Thread eröffnet. Damit läuft f_1 weiterhin auf dem Mainthread, während ein Teil von f_1 gleichzeitig auf dem roten Hintergrundthread abläuft. Irgendwann kehrt die Kontrolle auf dem Mainthread zurück von f_1 zu seinem Aufrufer f . Gleichzeitig geht die Kontrolle auf dem roten Hintergrundthread über zu f_2 . Somit laufen f_1 und f_2 parallel auf unterschiedlichen Threads.

Da es innerhalb der Verfeinerung von f zu einem Threadwechsel kommt, wird auch f in zwei unterschiedlichen Farben dargestellt. Dies drückt aus, dass f auf zwei Threads läuft, mithin also innerhalb von f ein Threadwechsel stattfindet.

Implementation

Die Implementation eines Threadwechsels ist sehr plattformabhängig. Im Folgenden wird ein Threadwechsel anhand eines C# Beispiels gezeigt. Später wird das Thema asynchrone Ausführung dann am Beispiel einer Wecker Anwendung vertieft.

In C#, bzw. allgemeiner auf der .NET Plattform, kann ein Threadwechsel auf unterschiedliche Weise realisiert werden. Zwei Möglichkeiten werden im Folgenden gezeigt:

- das explizite Starten eines neuen Threads,
- das implizite Starten eines neuen Threads über einen periodischen Timer.

Ein Thread kann explizit gestartet werden. Dazu wird eine Instanz des Typs *Thread* erzeugt. Als Parameter wird dem Konstruktor eine Lambda Expression übergeben. Dieser Code wird ausgeführt, sobald der Thread mit *Start* gestartet wird. Natürlich erfolgt die Ausführung dann auf einem neuen Thread.

```
[Test]
public void ExplicitThread() {
    var thread1 = new Thread(() => { DoSomething(1); });
    var thread2 = new Thread(() => { DoSomething(2); });
    var thread3 = new Thread(() => { DoSomething(3); });

    thread1.Start();
    thread2.Start();
    thread3.Start();

    Thread.Sleep(5000);
}

private void DoSomething(int threadNo) {
    do {
        Console.WriteLine($"Hello from thread {threadNo}");
        Thread.Sleep(100);
    } while (true);
}
```

Das Listing demonstriert mit einem automatisierten Test, wie Threads explizit gestartet werden können. Die Ausführung dieses “Tests” führt dazu, dass folgende Meldungen auf der Konsole ausgegeben werden:

⌚ ExplicitThread [5.0 sec]

✓ Success

asynchronous.AsyncDemos.ExplicitThread

```
Hello from thread 3  
Hello from thread 1  
Hello from thread 2  
Hello from thread 2  
Hello from thread 3  
Hello from thread 1  
Hello from thread 3  
Hello from thread 2  
Hello from thread 1
```

Der Test läuft 5 Sekunden lang, da die Testmethode mit `Thread.Sleep(5000)` für 5.000 Millisekunden angehalten wird. Da zuvor drei Threads gestartet wurden, die nun im Hintergrund ablaufen, erfolgt eine Ausgabe auf der Konsole, obschon die Testmethode angehalten ist. Eine andere Variante, eine Funktionseinheit im Hintergrund auszuführen, stellen *Timer* dar. Ein Timer löst periodisch ein Ereignis aus. An dieses Ereignis kann wiederum eine Lambda Expression gebunden werden, die dann periodisch im Hintergrund ausgeführt wird. Ein Timer verwendet dazu intern ebenfalls einen Thread. Folgendes Listing demonstriert dies mit einem einfachen Konsolenprogramm.

```
public static void Main(string[] args) {  
    var timer = new Timer(o => {  
        Console.WriteLine(  
            $"Hello from the background at {DateTime.Now.ToString('T')}");  
    });  
    timer.Change(0, 500);  
    Thread.Sleep(5000);  
}
```

Das Programm führt zu folgender Ausgabe:

Nach diesem kurzen Überblick über die Implementation nebenläufiger Funktionseinheiten wird im Folgenden wieder ein vollständige Anwendung gezeigt.

Beispiel Wecker

Das Beispiel einer Wecker Anwendung demonstriert den Umgang mit asynchronen Funktionseinheiten.

Anforderungen

Es ist eine Anwendung zu erstellen, mit der man auf zwei unterschiedliche Arten einen Wecker stellen kann. Die folgende Abbildung zeigt den zugehörigen Dialog.

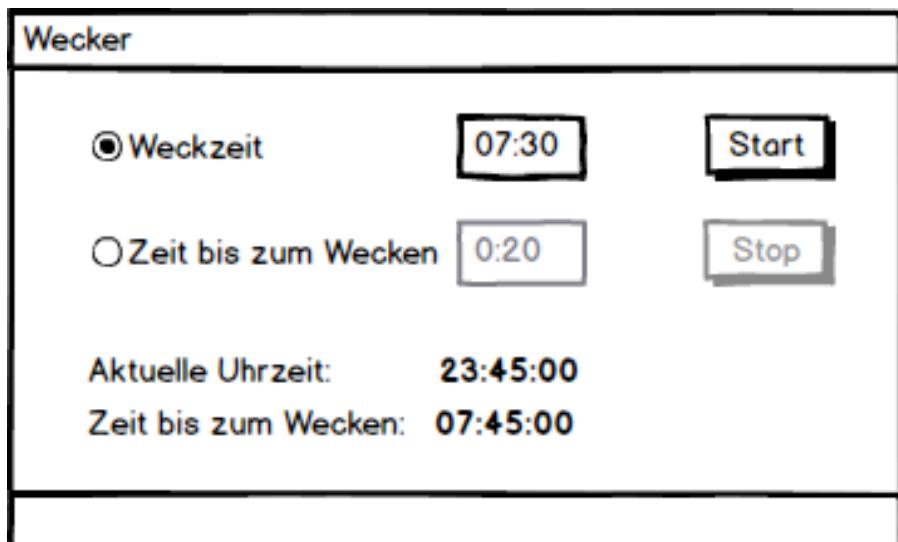


Abbildung 140: Wecker Dialog

Wenn die Wecker Anwendung gestartet wird, zeigt sie sekundengenau die aktuelle Uhrzeit an. Der Wecker kann mit einer konkreten Weckzeit gestartet werden. Dazu gibt man den Zeitpunkt ein, an dem man geweckt werden möchte und betätigt die Schaltfläche *Start*. Alternativ kann die Zeit bis zum Wecken angegeben werden, bspw. in 20 Minuten. Wurde der Wecker gestartet, wird die Restzeit bis zum Wecken fortlaufend sekundengenau angezeigt. Wenn die Weckzeit erreicht ist, wird die MP3-Datei `bimmel.mp3` abgespielt. Ferner wird die Restzeit dann nicht mehr angezeigt, die aktuelle Uhrzeit läuft jedoch weiterhin sekundengenau weiter.

Während der Wecker läuft, kann er mit der Schaltfläche *Stop* gestoppt werden. Auch in diesem Fall wird die verbleibende Restzeit dann nicht weiter angezeigt.

Interaktionsdiagramm

Die folgende Abbildung zeigt, welche Möglichkeiten der Anwender hat, mit der Anwendung zu interagieren.

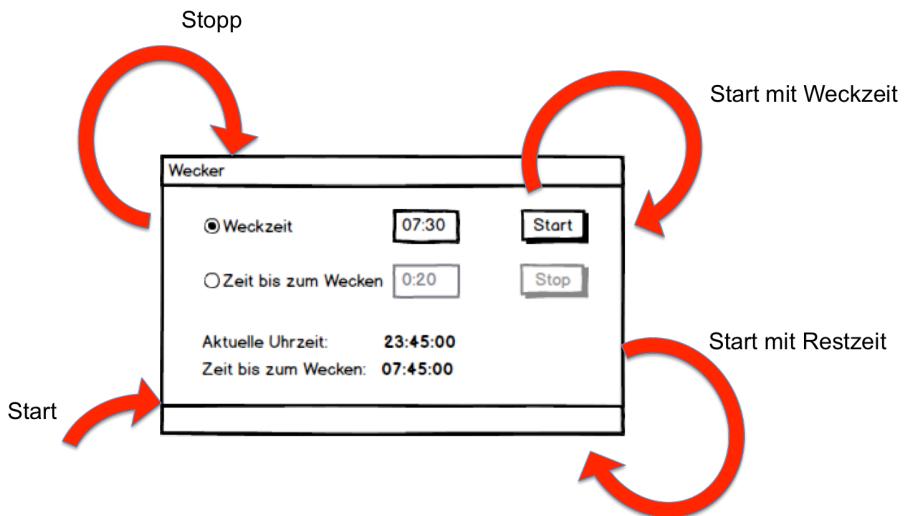


Abbildung 141: Interaktionsdiagramm des Weckers

Die Interaktion *Start* meint das Starten der Anwendung. Diese Interaktion ist relevant, da sie der Grund dafür ist, dass periodisch die aktuelle Uhrzeit angezeigt wird. Die drei weiteren Interaktionen *Start mit Weckzeit*, *Start mit Restzeit* sowie *Stopp* sind offensichtlich: sie führen zum Starten bzw. Stoppen des Weckers.

Entwurf der obersten Ebene

Aus dem Interaktionsdiagramm ergibt sich unmittelbar die oberste Ebene des Entwurfs.

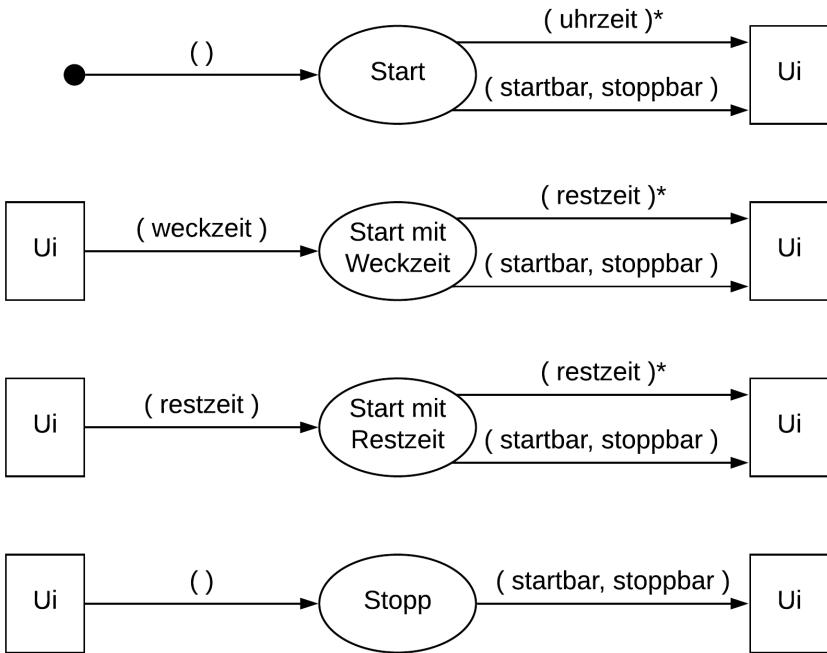


Abbildung 142: Oberste Ebene des Entwurfs

Das Starten der Anwendung führt dazu, dass immer wieder die aktuelle Uhrzeit an die *Ui* geliefert wird. An dieser Stelle kommt wieder ein Stream zum Einsatz: das Sternchen im Datenfluss (*uhrzeit*)^{*} ist absichtlich außerhalb der Klammern gesetzt, um damit auszudrücken, dass der Datenfluss mehrfach stattfindet. Immer wieder fließt aus der Funktionseinheit *Start* die aktuelle Uhrzeit zur *Ui*.

Die Interaktionen *Start mit Weckzeit* bzw. *Start mit Restzeit* sehen sehr ähnlich aus. Sie unterscheiden sich lediglich dadurch, dass im einen Fall eine Weckzeit im anderen Fall eine Restzeit von der *Ui* geliefert wird. Aus der Restzeit und der aktuellen Uhrzeit lässt sich die Weckzeit berechnen. Insofern ist bereits erkennbar, dass hier eine der Interaktionen leicht auf die andere abgebildet werden kann. Ergebnis der beiden Interaktionen ist jeweils ein Stream von Restzeiten. Einmal pro Sekunde liefert die Funktionseinheit die aktuelle Restzeit an die *Ui*. Ferner werden jeweils zwei boolesche Werte *startbar* und *stoppbar* an die *Ui* geliefert. Damit wird die *Ui* in die Lage versetzt, die Schaltflächen zum Starten und Stoppen des Weckers zu Aktivieren bzw. Deaktivieren. Nur wenn der Wecker gestartet ist, kann er gestoppt werden. Und nur wenn er gerade nicht läuft, kann er gestartet werden.

Verfeinerung von 'Start'

Das Starten der Wecker Anwendung führt dazu, dass in der Anwendung periodisch einmal pro Sekunde die aktuelle Uhrzeit angezeigt wird. Der folgende Entwurf zeigt, wie das funktioniert.

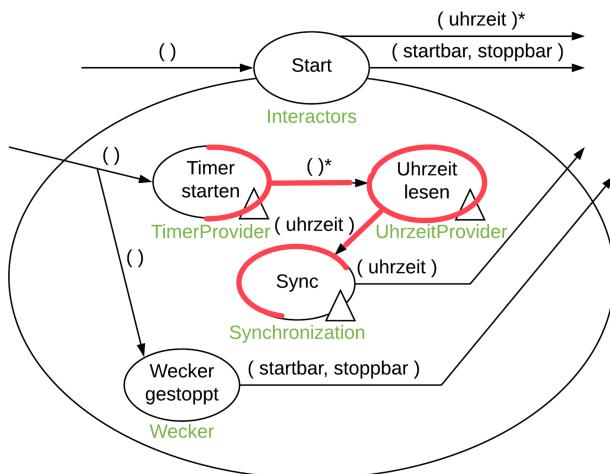


Abbildung 143: Verfeinerung von *Start*

Zunächst wird ein Timer gestartet, der einmal pro Sekunde ein Callback aufruft. Am Ausgang der Funktionseinheit *Timer starten* steht $()^*$ um auszudrücken, dass es sich um einen Stream handelt. Dieser Datenfluss wird einmal pro Sekunde durchlaufen. Das Einfärben des Datenflusses zeigt an, dass dieser Datenfluss auf einem Hintergrundthread stattfindet, der implizit durch den Timer gestartet wurde. Auf diesem Hintergrundthread wird anschließend die aktuelle Uhrzeit ermittelt. Diese wird dann zuletzt über die Funktionseinheit *Sync* herausgereicht. Das Synchronisieren sorgt dafür, dass der Datenfluss, der aus *Start* herausführt, wieder synchronisiert auf den Mainthread stattfindet. Andernfalls würde die UI "meckern". Dies ist eine Spezialität von .NET Benutzerschnittstellen und somit nicht für jeden Entwickler relevant.

Wenn diese Synchronisation auf der verwendeten Plattform notwendig ist, zeigt der Entwurf, wie mit dem Aspekt umgegangen werden kann. In den meisten Fällen dürfte das Problem bislang wohl innerhalb der UI gelöst worden sein, so wie Microsoft es in seinen Beispielen vormacht. Damit sind allerdings Aspekte vermischt, denn die UI hat ja bereits eine Aufgabe zu

erfüllen. Ihr sollte nicht auch noch die Verantwortlichkeit für die Synchronisation übertragen werden. Ein Dialog, der bislang auf dem Mainthread angekommen wurde, sollte nicht geändert werden, nur weil plötzlich eine Funktionseinheit mit mehreren Threads arbeitet. Dieser Aspekt kann, wie oben im Entwurf gezeigt, aus der UI herausgelöst werden. Die Funktionseinheit, die Multithreading einführt, ist auch für die Synchronisation verantwortlich.

Implementation von 'Start'

Die Umsetzung des Entwurfs geht wieder leicht von der Hand. Auf der obersten Ebene ist die *Main* Methode dafür zuständig, die benötigten Objekte zu instanziieren.

```
public static class Program
{
    [STAThread]
    public static void Main() {
        var mainWindow = new MainWindow();
        var app = new Application { MainWindow = mainWindow };
        var interactors = new Interactors();

        interactors.Start(
            (startbar, stoppbar) => mainWindow.Neuer_Zustand(startbar, stoppbar),
            uhrzeit => mainWindow.Neue_Uhrzeit(uhrzeit));

        app.Run(mainWindow);
    }
}
```

Es wird die Methode *Start* der Klasse *Interactors* aufgerufen. Diese erhält zwei Lambda Expressions als Parameter. Die erste Lambda erhält die beiden booleschen Werte und übergibt diese an die UI. Die zweite Lambda wird sekündlich aufgerufen. Die dabei übermittelte Uhrzeit wird ebenfalls an die UI übergeben.

Die UI ist mit WPF erstellt und besteht aus einer Xaml Datei sowie einer Codebehind Datei.

```

<Grid>
    <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition />
        <RowDefinition />
        <RowDefinition />
        <RowDefinition />
    </Grid.RowDefinitions>

    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="2*"/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>

    <DockPanel Grid.Column="0" Grid.Row="0" >
        <RadioButton x:Name="radioWeckzeit" Content="Weckzeit" Margin="10" FontSize="12"/>
        <TextBox x:Name="txtWeckzeit" VerticalAlignment="Top" HorizontalAlignment="Right"
            Margin="10" Width="60"/>
    </DockPanel>

    <DockPanel Grid.Column="0" Grid.Row="1" >
        <RadioButton x:Name="radioRestzeit" Content="Restzeit" Margin="10" FontSize="12"/>
        <TextBox x:Name="txtRestzeit" VerticalAlignment="Top" HorizontalAlignment="Right"
            Margin="10" Width="60"/>
    </DockPanel>

    <Button x:Name="btnStart" Grid.Column="1" Grid.Row="0" Content="Start" Margin="5"
        Padding="4" HorizontalAlignment="Center" VerticalAlignment="Center"/>
    <Button x:Name="btnStopp" Grid.Column="1" Grid.Row="1" Content="Stopp" Margin="5"
        Padding="4" HorizontalAlignment="Center" VerticalAlignment="Center"/>

    <Label x:Name="lblUhrzeit" Grid.Column="0" Grid.Row="2" Grid.ColumnSpan="2"
        Content="Uhrzeit" HorizontalAlignment="Center" VerticalAlignment="Center"/>
    <Label x:Name="lblRestzeit" Grid.Column="0" Grid.Row="3" Grid.ColumnSpan="2"
        Content="Restzeit" HorizontalAlignment="Center" VerticalAlignment="Center"/>

```

```
public partial class MainWindow : Window
{
    public MainWindow() {
        InitializeComponent();
        Weckzeit_Eingabe();

        radioWeckzeit.Click += (o, e) => {
            Weckzeit_Eingabe();
        };

        radioRestzeit.Click += (o, e) => {
            Restzeit_Eingabe();
        };

        btnStart.Click += (o, e) => {
            if (radioWeckzeit.IsChecked.Value) {
                if (DateTime.TryParse(txtWeckzeit.Text, out var weckzeit)) {
                    Start_mit_Weckzeit(weckzeit);
                }
            }
            else {
                if (TimeSpan.TryParse(txtRestzeit.Text, out var restzeit)) {
                    Start_mit_Restzeit(restzeit);
                }
            }
        };
        btnStopp.Click += (o, e) => Stopp();
    }

    private void Weckzeit_Eingabe() {
        txtWeckzeit.IsEnabled = true;
        txtRestzeit.IsEnabled = false;
        radioWeckzeit.IsChecked = true;
        radioRestzeit.IsChecked = false;
    }
}
```

```

private void Restzeit_Eingabe() {
    txtWeckzeit.IsEnabled = false;
    txtRestzeit.IsEnabled = true;
    radioWeckzeit.IsChecked = false;
    radioRestzeit.IsChecked = true;
}

public event Action<DateTime> Start_mit_Weckzeit;

public event Action<TimeSpan> Start_mit_Restzeit;

public event Action Stopp;

public void Neue_Restzeit(TimeSpan restzeit) {
    lblRestzeit.Content = restzeit.ToString(@"hh\:mm\:ss");
}

public void Neuer_Zustand(bool startable, bool stoppable) {
    btnStart.IsEnabled = startable;
    btnStopp.IsEnabled = stoppable;

    lblRestzeit.Visibility = stoppable ? Visibility.Visible : Visibility.Hidden;
}

public void Neue_Uhrzeit(DateTime uhrzeit) {
    lblUhrzeit.Content = uhrzeit.ToString("HH:mm:ss");
}
}

```

Die UI sieht simpel aus, erfordert aber trotzdem einiges an Arbeit. Allerdings ist es reine Routinearbeit, die lediglich Zeit in Anspruch nimmt.

Weiter geht es mit der Klasse *Interactors*. Sie ist dafür zuständig, die im Entwurf gezeigten Bestandteile zu integrieren.

```

public class Interactors
{
    private TimerProvider _timer;
    private readonly Synchronizer _sync = new Synchronizer();

    public void Start(Action<bool, bool> onZustand, Action<DateTime> onTimer) {
        var (startbar, stoppbar) = Wecker.Wecker_gestoppt();
        onZustand(startbar, stoppbar);

        _timer = new TimerProvider();
        _timer.Tick += () => {
            var uhrzeit = UhrzeitProvider.Aktuelle_Uhrzeit();
            _sync.Process(() => {
                onTimer(uhrzeit);
            });
        };
    }
}

```

Die Start Methode ist eine Integrationsmethode. Aus diesem Grund darf sie selbst keine Domänenlogik enthalten sondern lediglich Aufrufe anderer Methoden, die zur Lösung gehören.

Etwas Gewöhnungsbedürftig sind möglicherweise die Lambda Expressions. Der *TimerProvider* verfügt über einen Event *Tick*, der sekündlich ausgelöst wird. An diesen Event wird eine Lambda Expression gebunden. Innerhalb dieser Lambda wird zunächst die Uhrzeit ermittelt. Der Callback *onTimer* soll nun allerdings nicht innerhalb des *Tick* Events direkt aufgerufen werden, weil dies dann auf dem Hintergrundthread erfolgen würde. Daher wird der Aufruf in die Methode *_sync.Process* eingeschachtelt. Erneut kommt dabei eine Lambda Expression zum Einsatz.

```

public class TimerProvider
{
    private Timer _timer;
    public event Action Tick;

    public TimerProvider() {
        _timer = new Timer(state => TimerTick(), null, 1000, 1000);
    }

    private void TimerTick() {
        Tick?.Invoke();
    }
}

```

Der *TimerProvider* verwendet intern ein .NET *Timer* Objekt. Der Provider kapselt die Details dieser API. Im nächsten Inkrement wird diese Klasse erweitert.

```
public class UhrzeitProvider
{
    public static DateTime Aktuelle_Uhrzeit() {
        return DateTime.Now;
    }
}
```

Die Klasse *UhrzeitProvider* kapselt den *DateTime.Now* Aufruf.

```
public class Synchronizer
{
    private readonly Dispatcher dispatcher;

    public Synchronizer() {
        dispatcher = Dispatcher.CurrentDispatcher;
    }

    public void Process(Action action) {
        dispatcher.Invoke(action);
    }
}
```

Um die UI davon zu befreien, sich mit der Synchronisation von Aufrufen zu befassen, die auf einem Hintergrundthread eintreffen, synchronisiert die Funktionseinheit *Start* den Callback Aufruf. Technisch ist in der Klasse *Synchronizer* der .NET *Dispatcher* gekapselt.

Damit ist das erste Inkrement des Weckers realisiert. Die UI zeigt nach dem Start der Anwendung sekündlich die aktuelle Uhrzeit an.

Verfeinerung von 'Start mit Weckzeit'

Die Verfeinerung des Entwurfs für die Interaktion *Start mit Weckzeit* ist etwas umfangreicher. Auf den ersten Blick mag der Entwurf kompliziert aussehen, doch bei näherer Betrachtung ist er einfach.

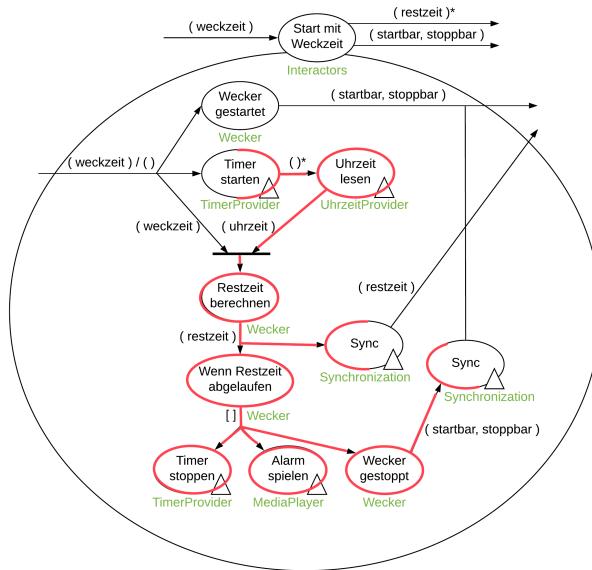


Abbildung 144: Verfeinerung von *Start mit Weckzeit*

Zunächst liefert die Funktionseinheit *Wecker gestartet* die beiden Werte *startbar* und *stoppbar*. Damit ist die UI in der Lage, die Schaltflächen zu aktivieren bzw. deaktivieren. Anschließend wird der Timer gestartet, der dafür zuständig ist, sekündlich ein Callback auszuführen. An diesem Datenfluss, der wieder einen Stream darstellt, wird zunächst die aktuelle Uhrzeit ermittelt. Aus dieser und der Weckzeit wird anschließend die Restzeit berechnet. Damit die Restzeit in der UI dargestellt werden kann, muss der Datenfluss durch die Funktionseinheit *Sync* auf den Hauptthread synchronisiert werden. Das Einfärben der Datenflüsse zeigt an, dass *Timer starten* einen weiteren Thread eröffnet hat. Durch *Sync* wird nun wieder zurückgewechselt auf den Mainthread.

Nachdem die Restzeit berechnet ist, wird durch die Funktionseinheit *Wenn Restzeit abgelaufen* geprüft, ob die Restzeit abgelaufen ist. Wenn das der Fall ist, wird der Timer gestoppt und der Alarm abgespielt. Ferner wird durch die Funktionseinheit *Wecker gestoppt* mitgeteilt, dass sich der Zustand des

Weckers erneut geändert hat. Auch diese Information wird durch *Sync* auf den Mainthread synchronisiert.

Der ausgehende Datenfluss an der Funktionseinheit *Wenn Restzeit abgelaufen* ist mit eckigen Klammern dargestellt. Das bedeutet, er ist optional. Runde Klammern würden bedeuten, dass die Methode für jeden Aufruf mit einer Restzeit den ausgehenden Datenfluss auslöst. Das ist aber nicht der Fall. Der ausgehende Datenfluss findet nur statt, wenn die Restzeit kleiner oder gleich null ist.

Implementation von 'Start mit Weckzeit'

Die Klasse *Interactors* ist wieder zuständig für die Integrationsmethode *Start_mit_Weckzeit*. Erster Parameter ist der eingehende Datenfluss, hier die Weckzeit in Form eines *DateTime* Objekts. Zweiter und dritter Parameter sind jeweils zwei *Action* Objekte. Diese Funktionszeiger repräsentieren die beiden ausgehenden Datenflüsse. Der Callback *onZustand* wird aufgerufen, wenn der Wecker seinen Zustand ändert. Dies ist der Fall, wenn er gestartet wird bzw. wenn die Weckzeit erreicht ist und der Wecker alarmiert. Beim ersten Aufruf des Callbacks muss der Aufruf nicht synchronisiert werden, da sich die Methode hier noch auf dem Mainthread befindet. Alles was danach innerhalb von *_timer.Start* stattfindet, muss allerdings auf den Mainthread synchronisiert werden, bevor es die Methode verlässt und zur UI geht. Da liegt daran, dass durch *_timer.Start* implizit ein neuer Thread eröffnet wird, so dass die Lambda Expression im Hintergrund aufgerufen wird.

```

public void Start_mit_Weckzeit(
    DateTime weckzeit,
    Action<bool, bool> onZustand,
    Action<TimeSpan> onRestzeit) {
    var (startbar, stoppbar) = Wecker.Wecker_gestartet();
    onZustand(startbar, stoppbar);

    _timer.Start(() => {
        var uhrzeit = UhrzeitProvider.Aktuelle_Uhrzeit();
        var restzeit = Wecker.Restzeit_berechnen(uhrzeit, weckzeit);
        _sync.Process(() => onRestzeit(restzeit));

        Wecker.Wenn_Restzeit_abgelaufen(restzeit, () => {
            _timer.Stopp();
            Media_Player.Alarm_abspielen();
            _sync.Process(() => {
                (startbar, stoppbar) = Wecker.Wecker_gestoppt();
                onZustand(startbar, stoppbar);
            });
        });
    });
}

```

Wir haben es nun quasi mit zwei Timern zu tun. Der eine läuft ständig weiter, um die Uhrzeit an die UI zu liefern. Der andere läuft nur, wenn der Wecker gestartet ist. Würden tatsächlich zwei getrennte Instanzen der Timer Klasse verwendet, würden die Callbacks mit einem zeitlichen Versatz von bis zu einer Sekunde aufgerufen. Das würde dazu führen, dass die Uhrzeit und die Restzeit in der UI leicht zeitversetzt aktualisiert würden. Um dies zu vermeiden, verwendet die Klasse *TimerProvider* intern lediglich eine Instanz eines .NET Timers. Mit jedem Auslösen des “echten” Timers wird der *Tick* Event ausgelöst, um darüber die Uhrzeit auszugeben. Ferner wird geprüft, ob ein mit *Start* gestarteter weiterer Callback aufgerufen werden muss.

```

public class TimerProvider
{
    private Timer _timer;
    private bool _started;
    private event Action _stoppableTick;

    public event Action Tick;

    public TimerProvider() {
        _timer = new Timer(state => TimerTick(), null, 1000, 1000);
    }

    private void TimerTick() {
        Tick?.Invoke();
        if (_started) {
            _stoppableTick?.Invoke();
        }
    }

    public void Start(Action onTick) {
        _stoppableTick = onTick;
        _started = true;
    }

    public void Stop() {
        _stoppableTick = null;
        _started = false;
    }
}

```

Auf diese Weise kann nun ein Callback mit *Start* sekündlich aufgerufen werden und es ist möglich, diesen mit *Stop* wieder zu stoppen.

Weiters wurde die Klasse Wecker ergänzt, mit den Operationen *Restzeit_berechnen* und *Wenn_Restzeit_abgelaufen*. Ferner sind dort die beiden Methoden *Wecker_gestartet* und *Wecker_gestoppt* abgelegt, die jeweils zurückliefern, ob der Wecker gestartet bzw. gestoppt werden kann.

```
public static class Wecker
{
    public static (bool startbar, bool stoppbar) Wecker_gestartet() {
        return (false, true);
    }

    public static (bool startbar, bool stoppbar) Wecker_gestoppt() {
        return (true, false);
    }

    public static TimeSpan Restzeit_berechnen(DateTime uhrzeit, DateTime weckzeit) {
        return weckzeit - uhrzeit;
    }

    public static void Wenn_Restzeit_abgelaufen(TimeSpan restzeit, Action continueWith) {
        if (restzeit <= new TimeSpan()) {
            continueWith();
        }
    }
}
```

Zuletzt ist noch die Klasse *Media_Player* hinzugekommen. Sie ist dafür verantwortlich, den Alarmsound abzuspielen. Hier wurde der Einfachheit halber eine Standard Windows Datei gewählt.

```
public static class Media_Player
{
    public static void Alarm_abspielen() {
        var player = new SoundPlayer {
            SoundLocation = @"C:\Windows\media\Alarm02.wav"
        };
        player.Play();
    }
}
```

Damit ist das zweite Inkrement des Weckers hergestellt.

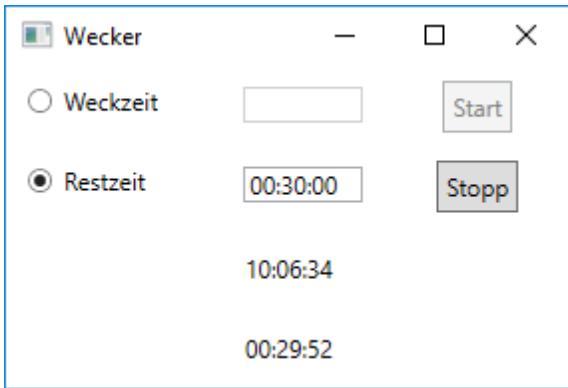


Abbildung 145: Der Wecker in Action

Verfeinerung von 'Stopp'

Die Interaktion *Stopp* ist wenig spektakulär. Es muss lediglich der Timer gestoppt werden. Ferner muss die UI über die Zustandsänderung informiert werden, damit dort die Schaltflächen entsprechend aktiviert bzw. deaktiviert werden können.

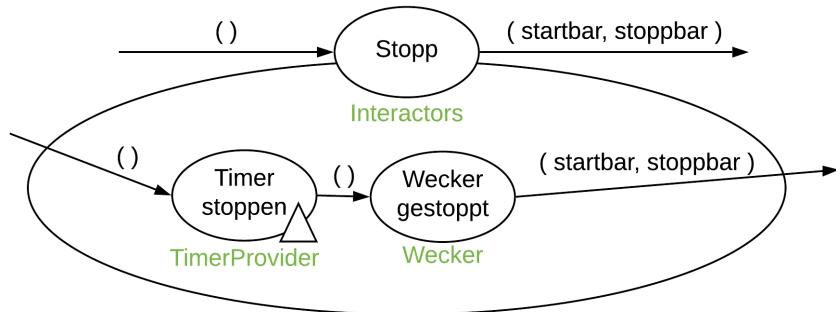


Abbildung 146: Verfeinerung von *Stopp*

Implementation von 'Stopp'

Die Implementation von *Stopp* beschränkt sich darauf, die Operationen im *Interactor* aufzurufen.

```

public void Stopp(Action<bool, bool> onZustand) {
    _timer.Stopp();
    var (startbar, stoppbar) = Wecker.Wecker_gestoppt();
    onZustand(startbar, stoppbar);
}

```

Verfeinerung von 'Start mit Restzeit'

Auch die Interaktion *Start mit Restzeit* ist dank der Vorarbeiten schnell erledigt. Es muss lediglich aus der Restzeit und der aktuellen Uhrzeit die Weckzeit bestimmt werden. Anschließend wird die Interaktion *Start mit Weckzeit* wiederverwendet.

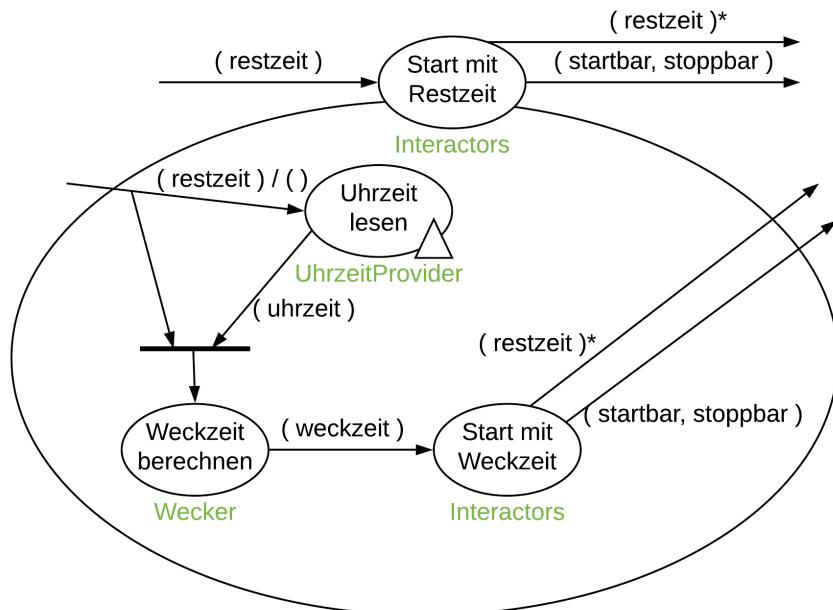


Abbildung 147: Verfeinerung von *Start mit Restzeit*

Implementation von 'Start mit Restzeit'

Für die Implementation muss wieder der *Interactor* ergänzt werden.

```
public void Start_mit_Restzeit(
    TimeSpan restzeit,
    Action<bool, bool> onZustand,
    Action<TimeSpan> onRestzeit) {
    var uhrzeit = UhrzeitProvider.Aktuelle_Uhrzeit();
    var weckzeit = Wecker.Weckzeit_berechnen(restzeit, uhrzeit);
    Start_mit_Weckzeit(weckzeit, onZustand, onRestzeit);
}
```

Ferner fehlt in der Klasse Wecker die Methode zur Berechnung der Weckzeit aus der Restzeit und der Uhrzeit.

```
public static TimeSpan Restzeit_berechnen(DateTime uhrzeit, DateTime weckzeit) {
    return weckzeit - uhrzeit;
}
```

Fazit

Eine Anwendung wie ein Wecker stellt uns vor die Herausforderung, mit Multithreading arbeiten zu müssen. Im Entwurf hilft es sehr, die unterschiedlichen Threads durch Einfärben der Datenflüsse sichtbar zu machen. Auf diese Weise kann schnell erkannt werden, wo eine Synchronisation erforderlich ist.

Beim Wecker tauchte die Herausforderung zwar nicht auf, doch es kann auf diese Weise auch erkannt werden, ob konkurrierende Zugriffe auf Zustand auf unterschiedlichen Threads stattfinden. In einem solchen Fall kann dann durch *Locking* sichergestellt werden, dass der Zustand konsistent bleibt.

Prinzipien

In diesem Kapitel werden einige Prinzipien erläutert, die wesentlichen Einfluss auf die Wandelbarkeit und Korrektheit von Software haben. Dabei geht es vor allem darum, die Begriffe klar zu definieren.

Typische Strukturen von Abhängigkeiten

Wir finden in der Softwareentwicklung zwei typische Strukturen von Abhängigkeiten. Im einen Fall ist eine Funktionseinheit von mehreren anderen Funktionseinheiten abhängig, im anderen Fall sind mehrere Funktionseinheiten von einer Funktionseinheit abhängig. Für die folgende Betrachtung genügt es, sich eine Funktionseinheit als eine Klasse vorzustellen.

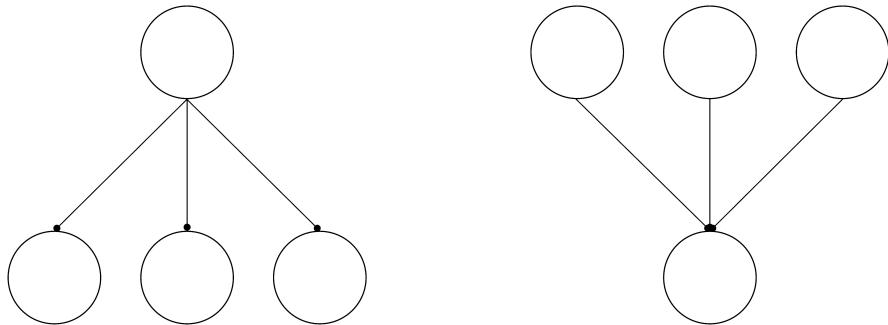


Abbildung 148: Typische Strukturen von Abhängigkeiten

Im weiteren Verlauf wird dargestellt, welche Herausforderungen mit beiden Strukturen einhergehen. Es geht nicht darum, diese Strukturen zu vermeiden. Sie werden aus guten Gründen verwendet. Jedoch soll aufgezeigt werden, welche Prinzipien befolgt werden müssen, damit diese Strukturen keine Probleme verursachen.

Betrachten wir zunächst den Fall, dass eine Funktionseinheit von mehreren Funktionseinheiten abhängig ist (linke Seite in der Abbildung). In dieser Struktur gibt es bei Änderungen eine Herausforderung. Sobald wir an einer der unteren, unabhängigen, Funktionseinheiten etwas ändern, müssen wir potentiell an der oberen, davon abhängigen, Funktionseinheit ebenfalls etwas ändern. Strukturelle Änderungen sind dabei der vergleichsweise harmlose Fall. Ändern sich beispielsweise Namen, die Signatur einer Methode oder der Datentyp einer Eigenschaft, unterstützt uns jede moderne IDE sowie ggf. der Compiler dabei, diese Änderung durchzuführen oder ein Problem damit zu entdecken. Bei dynamischen Sprachen wird die Heraus-

forderung allerdings etwas größer sein, so dass es hier zwingend ist, automatisierte Tests zu schreiben, die möglicherweise entstehende Problem rechtzeitig offenlegen.

Bestimmte Änderungen können allerdings zu deutlich größeren Problemen führen. Verhält sich nämlich eine der unteren unabhängigen Funktionseinheiten plötzlich anders, muss auf diese Verhaltensänderung in der übergeordneten Funktionseinheit reagiert werden. Diese logische Abhängigkeit kann nicht von der IDE erkannt werden. Hier helfen nur automatisierte Tests. Und auch diese stellen niemals vollkommen sicher, dass ein Problem identifiziert wird.

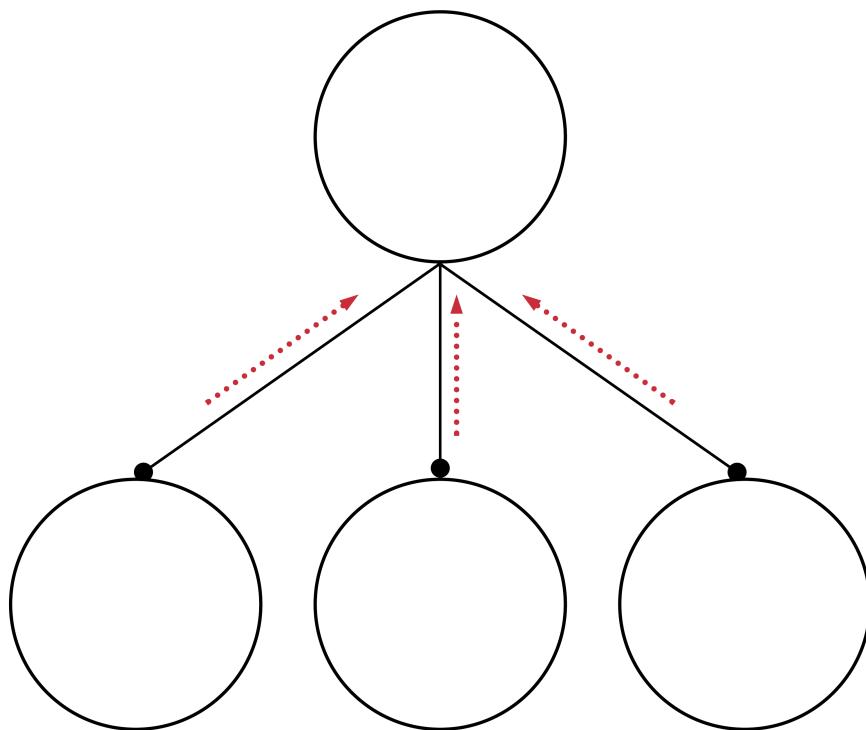


Abbildung 149: Änderungen wirken sich in der Gegenrichtung der Abhängigkeiten aus

Die Herausforderung bei dieser Struktur von Abhängigkeiten ist also, dass sich Änderungen in der Gegenrichtung der Abhängigkeiten auswirken.

IOSP - Integration Operation Segregation Principle

Was können wir nun tun, um das Problem zu entschärfen? Wohlgerne geht es hier nicht darum, diese Struktur von Abhängigkeiten zu vermeiden. Die Struktur ist absolut sinnvoll. Letztlich setzt wandelbare Software immer auf diese Struktur. Jedes Problem wird solange zerlegt, bis ausreichend kleine Teilprobleme entstanden sind. Weiter unten werden wir die Struktur in Form eines Baums noch größer fassen.

Wenn nun diese Struktur sinnvoll ist, andererseits aber zu Problemen führt, stellt sich die Frage, wie wir einen guten Umgang mit dieser Struktur finden. Wir müssen dafür sorgen, dass die Änderungen leichter fallen, sich insbesondere nicht so gravierend auf die übergeordneten Funktionseinheiten auswirken. Dies wird möglich, in dem wir die übergeordnete Funktionseinheit von Domänenlogik frei halten.

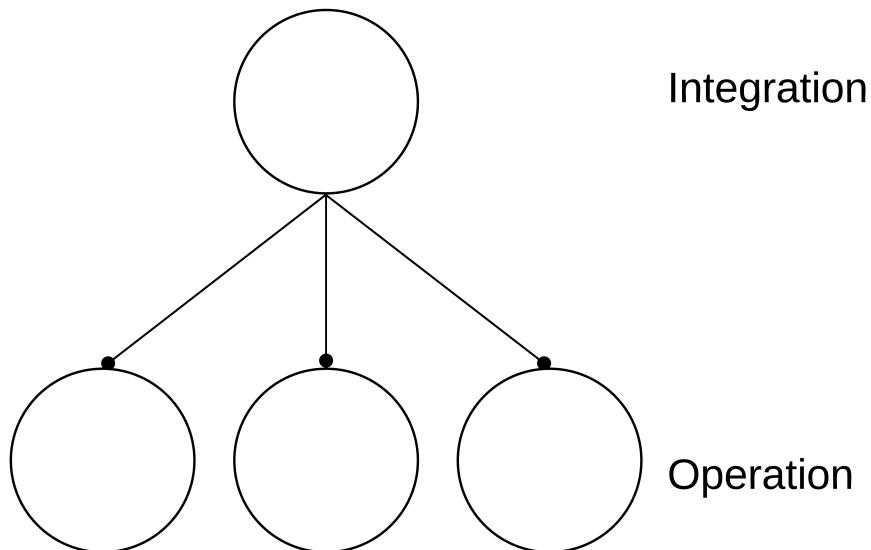


Abbildung 150: Abhängige Funktionseinheit darf keine Domänenlogik enthalten

Die Funktionseinheit A hat bereits eine Aufgabe. Sie integriert die Funktionseinheiten A_1 , A_2 und A_3 . Es ist ihr Job, sich darum zu kümmern, dass die drei Funktionseinheiten, von denen sie abhängt, in geeigneter Weise zusammenarbeiten. Folglich sollten wir diese Funktionseinheit nicht mit einer weiteren Zuständigkeit überfrachten, nämlich auch noch selbst zur Lösung des Problems beizutragen.

Das daraus abgeleitete Prinzip nennen wir *Integration Operation Segregation Principle* kurz *IOSP*. Segregation bedeutet Abtrennung.

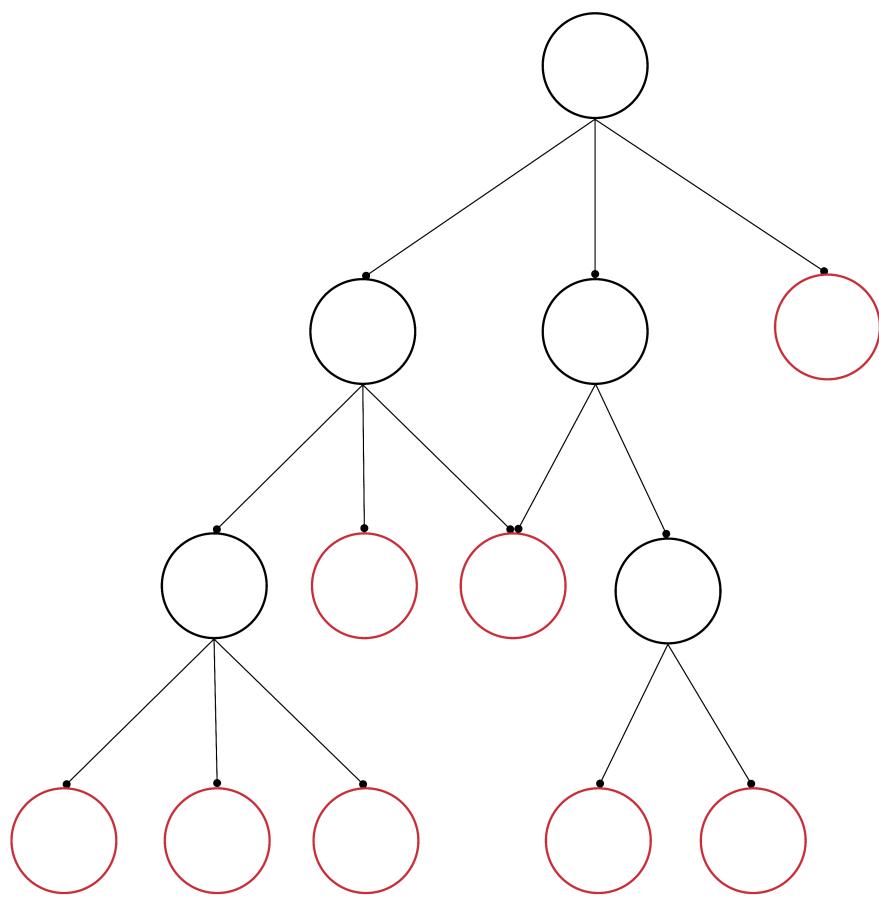
Definition: **Integration Operation Segregation Principle - IOSP**. Eine Funktionseinheit enthält entweder Domänenlogik, dann ist sie eine *Operation*, oder integriert andere Funktionseinheiten, dann ist sie eine *Integration*.

Durch das IOSP müssen wir uns bei jeder Funktionseinheit entscheiden, ob diese Domänenlogik enthält oder integriert. Enthält eine Funktionseinheit Domänenlogik, darf sie keine andere Funktionseinheit aufrufen, denn dann würde sie diese integrieren. Ruft eine Funktionseinheit dagegen andere auf, darf sie selbst keine Domänenlogik enthalten.

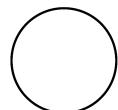
Wichtig zu unterscheiden sind hier die Aufrufe von Methoden. Natürlich darf eine Operation andere Methoden aufrufen, nämlich solche, aus dem verwendeten Framework oder anderen Bibliotheken. Diese APIs sind notwendig, damit die Operation ihren Teil der Domänenlogik realisieren kann. Was die Operation nicht aufrufen darf, sind andere Operationen, sprich Methoden, die ebenfalls Domänenlogik enthalten. Dann wären Operation und Integration vermischt. Operationen dürfen also keine Methoden aufrufen, die selbst zur Lösung gehören, die also in derselben Codebasis liegen wie die Operation.

Das IOSP ist nicht nur bei der Neuentwicklung von Software ein wichtiges Prinzip. Es dient auch als Ziel für Refaktorisierungen bei Legacy Code. Dort findet man sehr häufig Methoden, die das IOSP verletzten. Solche Methoden enthalten selbst Domänenlogik, rufen aber gleichzeitig auch andere Methoden der Lösung auf. Durch Refaktorisierungen kann hier dafür gesorgt werden, dass die Anteile Domänenlogik in Methoden ausgelagert werden, so dass eine reine Integrationsebene entsteht.

Betrachten wir nun Funktionseinheiten in einem größeren Kontext. Die Struktur, in der eine Funktionseinheit von mehreren Funktionseinheiten abhängig ist, finden wir in einem Baum wieder.



Blätter, Operation, Unit Tests



Knoten, Intergration, Integrationstests

Abbildung 151: Baumstruktur

In einem solchen Baum können wir *Blätter* und *Knoten* unterscheiden. Die Blätter stehen am unteren Rand des Baumes und haben keine Abhängigkeiten. Somit sind die Blätter Operationen, enthalten also die Domänenlogik. Die Knoten zeichnen sich dadurch aus, dass sie abhängig sind. Folglich ist ihre Aufgabe die Integration.

Diese Struktur und das IOSP als wesentliches Prinzip führen zu zwei sehr wesentlichen Konsequenzen:

- Eine solche Struktur ist leicht verständlich.
- Software ist gut automatisiert testbar, wenn sie in dieser Struktur vorliegt.

Die Verständlichkeit ergibt sich aus der Hierarchie von Funktionseinheiten. Jede Integration ist leicht verständlich, da sie lediglich Methodenaufrufe enthält. Die einzelnen Operationen befassen sich ausschließlich mit Details und sind daher ebenfalls leicht verständlich. Als zusätzliche Voraussetzung für die leichte Verständlichkeit der Operationen müssen diese fokussiert sein: eine Operation darf nur für einen Aspekt oder eine Verantwortlichkeit zuständig sein. Das *Single Responsibility Principle* SRP drückt genau das aus.

IOSP und automatisierte Tests

Software automatisiert zu testen, bedeutet immer, eine gute Mischung aus den drei Kategorien Systemtests, Integrationstests und Unit Tests zu verwenden. Die folgende Pyramide stellt dies dar. Die Breite der Pyramide gibt an, wieviele Tests der jeweiligen Kategorie verwendet werden sollten.

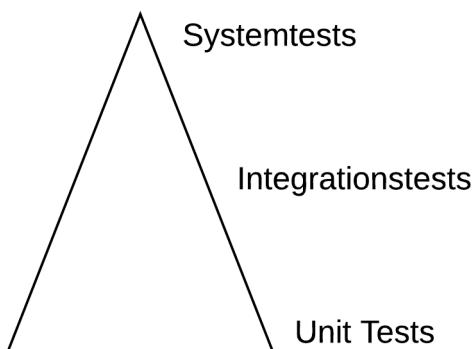


Abbildung 152: Testpyramide

Eine gesunde Teststrategie setzt auf wenige Systemtests. Diese zeichnen sich dadurch aus, dass sie das gesamte System in voller Integration testen. Es wird alles getestet, inklusive der Benutzerschnittstelle und den

Ressourcenzugriffen. Nichts wird durch Attrappen ersetzt. Man kann sich vorstellen, dass diese Kategorie von automatisierten Tests mit großen Herausforderungen verbunden sind. Genau aus diesem Grund wird man ihre Anzahl klein halten.

Die nächste Ebene bilden die Integrationstests. Auch bei Integrationstests werden große Teile des Gesamtsystems getestet. Typischerweise wird aber die Benutzerschnittstelle nicht mit getestet. Die Tests greifen dazu unterhalb der Benutzerschnittstelle an. Die Ressourcen werden bei Integrationstests in der Regel mit getestet.

Ganz am unteren Ende der Testpyramide haben dann die Unit Tests die Aufgabe, die vielen Sonderfälle abzudecken. Hier wird versucht, mit der kombinatorischen Explosion umzugehen.

Betrachtet man nun die Testpyramide, wird deutlich, dass es nicht ohne System- bzw. Integrationstests geht. Unit Tests alleine stellen nicht sicher, dass das Zusammenspiel der Funktionseinheiten korrekt funktioniert. Folglich werden die Funktionseinheit, die für die Integration zuständig sind, durch Integrationstests überprüft. Die Operationen werden durch Unit Tests geprüft. Bei konsequenter Einhaltung des IOSP entfällt damit ein wesentlicher Bereich von Tests: Unit Tests auf Knoten. Ein Knoten verstößt gegen das IOSP, wenn er Domänenlogik enthält. Schließlich hat ein Knoten Abhängigkeiten, wodurch seine Aufgabe die Integration ist. Solange er auch noch Domänenlogik enthält, muss diese möglichst isoliert getestet werden. Auf diese Weise kommen Tests zustande, bei denen die Abhängigkeiten durch Attrappen ersetzt werden. Enthält der Knoten aber keine Domänenlogik, gibt es nichts isoliert zu testen. Dies ist eine sehr wesentliche Erkenntnis, die jeden Softwareentwickler dazu anhalten sollten, das IOSP streng einzuhalten. Schließlich ergeben sich dadurch die beiden Vorteile *leichte Verständlichkeit* und *gute Testbarkeit*. Der Einsatz von Interfaces und Attrappen wird bis auf wenige Randfälle überflüssig!

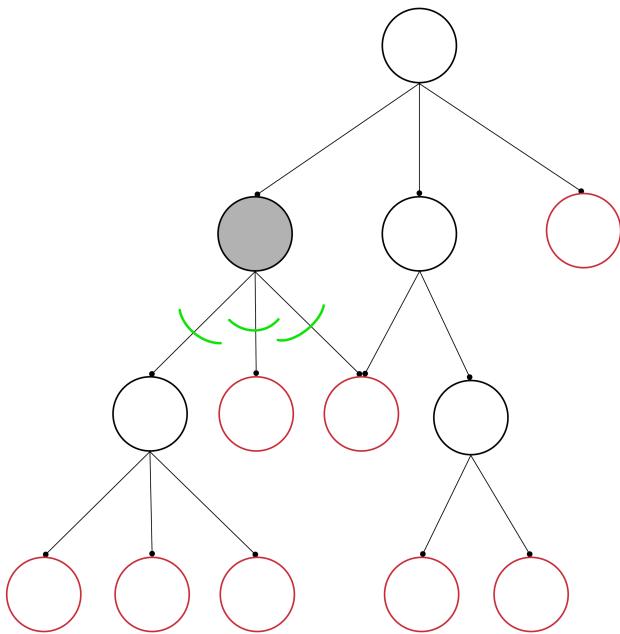


Abbildung 153: Freistellen von Integrationen entfällt

PoMO - Principle of Mutual Oblivion

Wenn wir uns die Struktur der Abhängigkeiten nochmals anschauen, wirkt das IOSP in vertikaler Richtung.

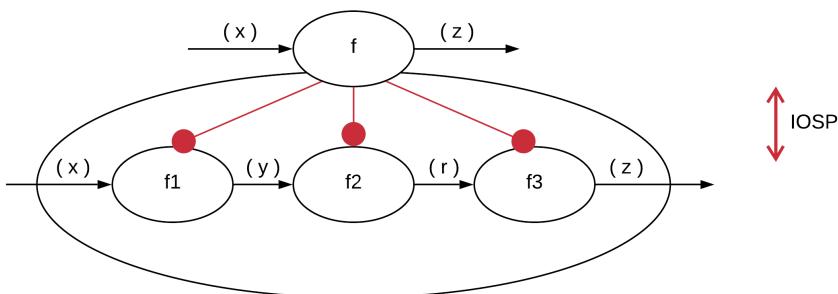


Abbildung 154: Abhängigkeitsstruktur eines Datenflusses

Die obere Funktionseinheit integriert die unteren, daher ist hier auf die Trennung von Integration und Operation zu achten. Die Funktionseinheit f integriert die Operationen f_1 , f_2 und f_3 . Folglich ist sie eine Integration und darf selbst keine Domänenlogik enthalten.

Es gibt ein weiteres Prinzip, das in horizontaler Richtung wirkt. Das Prinzip der *gegenseitigen Nichtbeachtung*, auf English *Principle of Mutual Oblivion* kurz PoMO, besagt, dass die einzelnen Operationen sich gegenseitig nicht beachten dürfen.

Definition: **Principle of Mutual Oblivion - PoMO.** Das Prinzip der gegenseitigen Nichtbeachtung. Operationen dürfen nicht voneinander abhängig sein. Eine Operation kennt weder ihren Vorgänger noch ihren Nachfolger. Die Integration geschieht durch eine übergeordnete Funktionseinheit.

Die in der Abbildung gezeigten Funktionseinheiten f , f_1 , f_2 und f_3 realisieren einen Datenfluss. Man könnte versucht sein, diesen Datenfluss ohne f herzustellen, wie es folgende Abbildung zeigt.

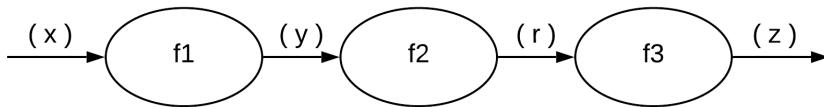


Abbildung 155: Datenfluss aus f_1 , f_2 und f_3 .

Bei der Implementation würde das allerdings zu Abhängigkeiten zwischen den Operationen führen. Falsch wäre die folgende Implementation:

```
public int f1(int x) {  
    var y = x + 2;  
    return f2(y);  
}  
  
public int f2(int y) {  
    var z = y * 2;  
    return f3(z);  
}  
  
public int f3(int z) {  
    var r = z + 42;  
    return r;  
}
```

Hier ruft nämlich *f1* seinen Nachfolger *f2* auf. Genauso falsch ist es, dass *f2* seinen Nachfolger *f3* aufruft. Diese Aufrufe verletzen das *Principle of Mutual Oblivion (PoMO)*. Auf deutsch übersetzt heißt das sowie wie die “gegenseitige Nichtbeachtung”. Das PoMO sorgt dafür, dass eine Funktionseinheit weder ihren Vorgänger noch ihren Nachfolger kennt und beachtet. Die oben gezeigte Verletzung des Principle of Mutual Oblivion kann leicht korrigiert werden, in dem *f* die Aufgabe der Integration übernimmt.

```
public int f(int x) {  
    var y = f1(x);  
    var z = f2(y);  
    var r = f3(z);  
    return r;  
}  
  
public int f1(int x) {  
    var y = x + 2;  
    return y;  
}  
  
public int f2(int y) {  
    var z = y * 2;  
    return z;  
}  
  
public int f3(int z) {  
    var r = z + 42;  
    return r;  
}
```

In dieser Implementation sind f_1 , f_2 und f_3 voneinander unabhängig. Die drei Funktionen haben keine Kenntnis voneinander, beachten sich gegenseitig nicht. Der Vorteil dieser Art der Integration liegt darin, dass es nun viel leichter ist, den Datenfluss zu verändern. Dazu muss lediglich die Funktionseinheit f modifiziert werden. Integrationsmethoden enthalten trivialen Code, der dadurch leicht zu modifizieren ist. Somit ist die Wandelbarkeit höher, dadurch dass es keine Abhängigkeiten zwischen den einzelnen Operationen gibt.

Daten

Bislang haben wir die Struktur betrachtet, in der eine Funktionseinheit von mehreren abhängig ist. Drehen wir das ganze nun um, finden wir mehrere Funktionseinheiten, die von einer gemeinsamen abhängig sind.

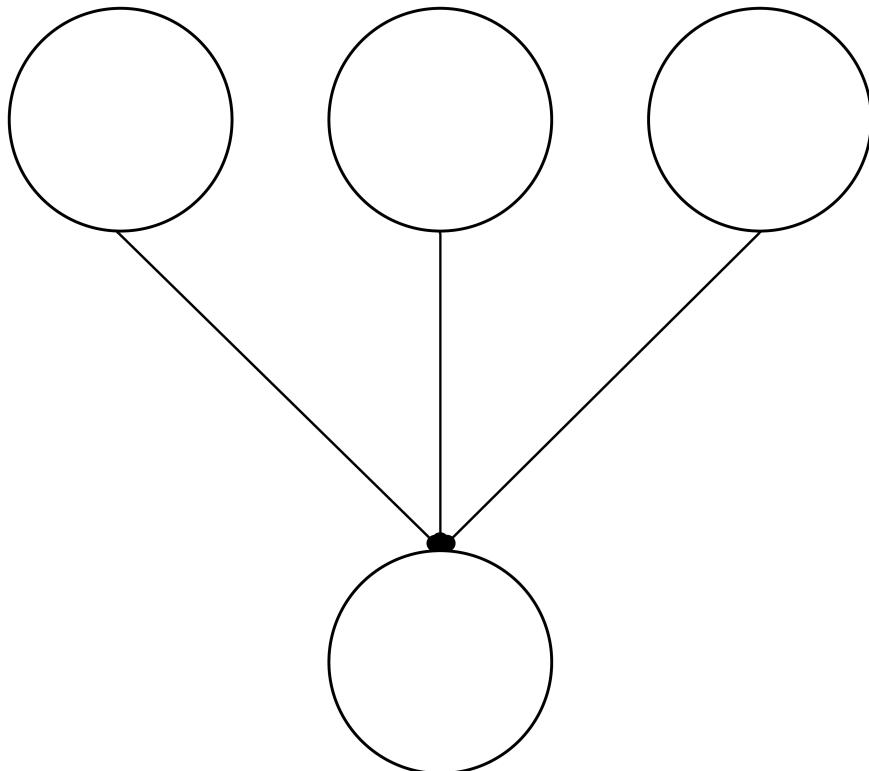


Abbildung 156: Mehrere Funktionseinheiten sind von einer Funktionseinheit abhängig

Auch hier gilt, dass diese Struktur in Softwaresystemen anzutreffen ist und dass es nicht darum geht, sie zu vermeiden. Es geht stattdessen darum, die Herausforderungen dieser Abhängigkeitsstruktur zu erkennen und einen guten Umgang damit zu finden, so dass diese Struktur keine Probleme bei der Wandelbarkeit verursacht.

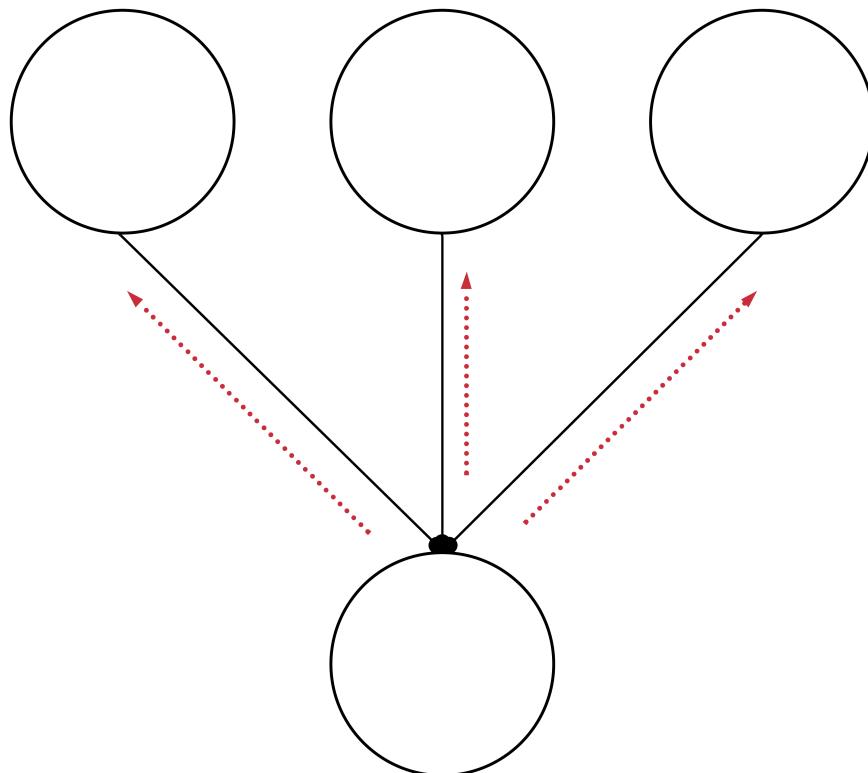


Abbildung 157: Auswirkung bei Änderungen

Die Herausforderung dieser Struktur liegt darin, dass die oberen Funktionseinheiten von Änderungen an der unteren betroffen sind. Wird die untere Funktionseinheit modifiziert, müssen alle davon abhängigen Funktionseinheiten betrachtet werden. Je mehr Funktionseinheiten von einer gemeinsamen Funktionseinheit abhängig sind, desto stärker wirkt sich das Problem aus.

Hierin liegt für viele Softwaresysteme eine große Herausforderung. In der Vergangenheit wurde die Objektorientierung häufig angewandt, ohne dabei auf Kontexte zu achten. Zentrale Konzepte einer Domäne, wie etwa Kunde, Produkt oder ähnliches, treten dabei an vielen Stellen auf. In der Folge ergibt sich daraus die Abhängigkeitsstruktur, von der hier die Rede ist: viele Funktionseinheiten sind von diesen zentralen Konzepten abhängig. Je mehr

Funktionalität an diesen zentralen Funktionseinheiten realisiert wurde, desto schwieriger wird es, diese zentralen Funktionseinheiten zu modifizieren. Eric Evans empfiehlt in seinem Buch *Domain Driven Design* aus gutem Grund die Einführung von *Bounded Contexts*. Die Konzepte einer Domäne haben je nach Kontext unterschiedlichen Bedarf an Daten und Funktionalität. Führt man also mehrere Kontexte ein, gibt es mehrere Repräsentationen der zentralen Konzepte, die sich je nach Kontext in ihren Details unterscheiden. Damit wird das Problem der Abhängigkeiten entschärft, da nun eine geringerer Anzahl an Funktionseinheiten von den zentralen Konzepten abhängig sind.

Wir gehen noch einen Schritt weiter. Die Einführung von *Bounded Contexts* ist eine gute Empfehlung, die wir ganz klar unterstützen. Zusätzlich empfehlen wir, in der gezeigten Struktur von Abhängigkeiten darauf zu verzichten, in den unabhängigen Funktionseinheiten Domänenlogik unterzubringen.

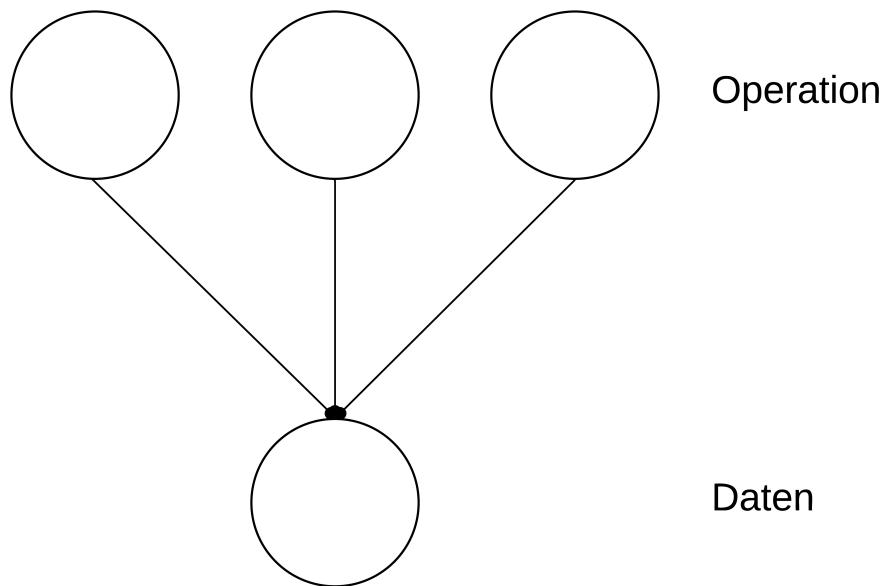


Abbildung 158: Datenstrukturen

Die untere Funktionseinheit, von der mehrere abhangig sind, sollte lediglich Daten enthalten, keine Domanenlogik. Bei anderungen sind dann nach wie vor mehrere Funktionseinheiten betroffen. Allerdings sind die Auswirkungen von anderungen an einer Datenstruktur uberschaubar. An einer Datenstruktur kann eine Eigenschaft wegfallen oder hinzukommen, es kann sich der Name oder der Typ ndern. In den allermeisten Fallen wird uns unsere Entwicklungsumgebung bei solchen anderungen unterstützen. Spatestens der Compiler wird feststellen, dass etwas nicht mehr zusammenpasst. Beim

Einsatz von dynamischen sprachen wie JavaScript oder Ruby müssen die automatisierten Tests die Aufgabe übernehmen, uns auf Probleme hinzuweisen.

IODA - Integration Operation Data API Architecture

Fasst man nun die beiden typischen Strukturen von Abhängigkeiten zu einem Bild zusammen, ergibt sich die in der folgenden Abbildung gezeigte Struktur. Zusätzlich sind noch APIs ergänzt. Das sind bspw. die verwendeten Frameworks sowie 3rd Party Bibliotheken.

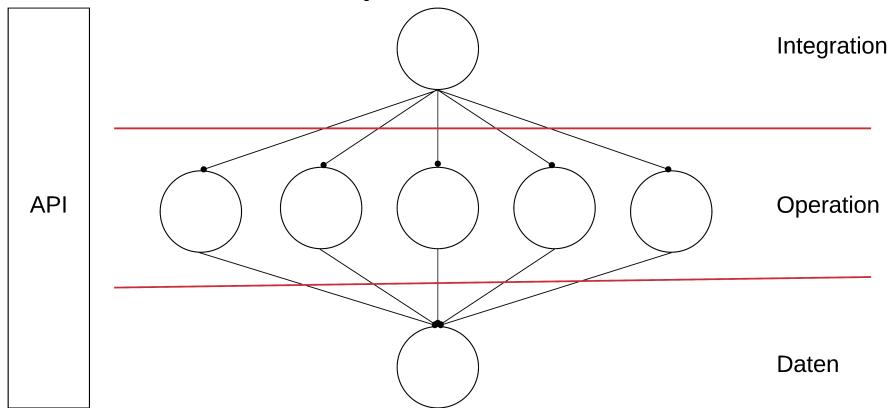


Abbildung 159: IODA Architecture

Auf der obersten Ebene befindet sich der Aspekt der *Integration*. Die Aufgabe der obersten Ebene liegt darin, die Operationen zu Datenflüssen zusammenzustellen. Hier haben wir maximale Abhängigkeiten, dafür aber trivialen Code.

In der Mitte liegt der wichtige Bereich der *Operationen*. Hier ist die gesamte Domänenlogik realisiert, ferner die Benutzerschnittstelle (Ui) sowie Ressourcenzugriffe. Hierbei handelt sich oft um komplizierten Code, der aber ohne Abhängigkeiten auskommt. Da die Operationen voneinander unabhängig sind, fällt das automatisierte Testen leicht.

Auf der unteren Ebene liegen dann die *Datenstrukturen*, die von den Operationen verwendet werden, um einen gemeinsamen Fluss von Daten zu realisieren. Operationen können somit auch mit strukturierten Daten arbeiten und sind nicht auf *string*, *int*, *bool* etc. beschränkt.

Auf allen drei Ebenen können APIs verwendet werden. Daraus ergibt sich die Bezeichnung der Struktur: *Integration Operation Data API Architecture* kurz IODA.

IODA für Projekte

Die IODA Architektur kann wunderbar auch auf die anderen Ebenen der Modulhierarchie angewandt werden. Häufig stehen Entwickler vor der Frage, wie sie den Quellcode einer Anwendung in Projekten organisieren soll. Auch hier gilt gemäß IOSP: es muss auf der Ebene von Projekten entschieden werden, ob ein Projekt integriert, dann darf es andere Projekte referenzieren, sich von ihnen abhängig machen. Oder ein Projekt enthält Logik, dann muss es frei sein von Abhängigkeiten. Ergänzt man die Idee der reinen Datenstrukturen, die sich aus der Betrachtung über Abhängigkeiten ergeben hat, stehen auf der untersten Ebene Projekte mit gemeinsam verwendeten Datenstrukturen.

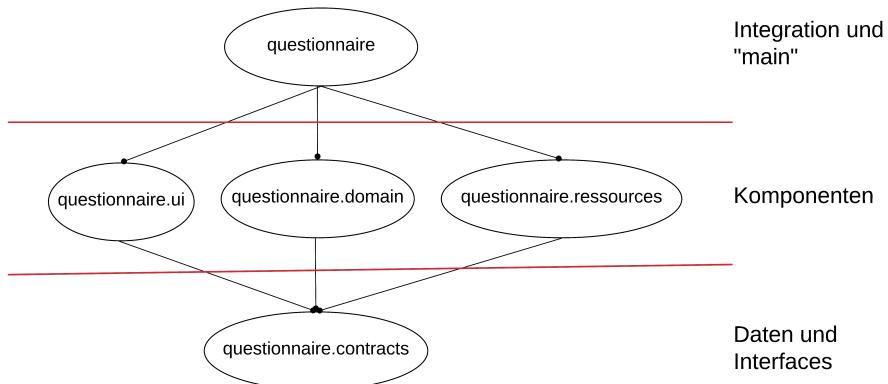


Abbildung 160: IODA für Projekte

Auf diese Weise hat jedes Projekt zunächst die klare Verantwortlichkeit für *Integration* vs. *Operation* vs. *Daten*. Ganz wesentlich dabei ist, dass es zwischen den Operationen keine Abhängigkeiten gibt. Dies würde dem *Principle of Mutual Oblivion* PoMO widersprechen.

Ferner muss jedes Projekt eine eindeutige inhaltliche Verantwortlichkeit übernehmen. Ein Projekt für die UI Bestandteile hat eine fokussierte Zuständigkeit und darf somit gemäß *Single Responsibility Principle* SRP nicht auch noch für andere Aspekte zuständig sein.

Die Implementation spiegelt den Entwurf

Es ist wesentlich, vor der Implementation eine Lösung zu entwerfen. Doch genauso wesentlich ist es, sich bei der Implementation exakt an den Entwurf zu halten. Wenn die Implementation den Entwurf nicht widerspiegelt, kann der Entwurf nicht all seine Vorteile entfalten.

Betrachten wir den Entwurf nochmal als Landkarte. Mit Hilfe der Landkarte kann ein Entwickler sich in der Lösung orientieren. Er kann auf einem relativ hohen Abstraktionsniveau über die Lösung nachdenken, sie nachvollziehen. Trifft er dann anschließend auf eine Implementation, die vom Entwurf abweicht, fällt es schwer, sich zu orientieren. Es ist so, als hätte er in der Landkarte einen Weg von A nach B gefunden, der in der Realität jedoch nicht existiert.

Die Abweichungen der Implementation vom Entwurf können subtil sein. Schon ein anderer Bezeichner führt zu Verwirrung. Im Entwurf hieß die Methode noch *Lese Kunde*, nun heißt sie in der Umsetzung *Read Customer*. Oder im Entwurf heißt sie *Read Customer*, in der Umsetzung jedoch *Get Customer*. Alle diese Abweichungen bei den Bezeichnern erschweren die Navigation im Code. Es muss jeweils eine Übersetzung erfolgen. So als wäre in der Wanderkarte ein Weg mit A1 bezeichnet, in der Realität heißt er jedoch X5. Irgendwann wird man es merken und trotzdem seinen Weg finden. Es ist jedoch deutlich beschwerlicher, als notwendig.

Ein ähnliches Problem entsteht auf umgekehrtem Weg. Stellen wir uns vor, dass in der Implementation eine Codestelle identifiziert wird, die näher betrachtet werden soll. Dazu will das Team den Entwurf heranziehen, um abstrakter auf den Sachverhalt schauen zu können. Entsprechen sich Entwurf und Implementation nicht, wird es erneut schwierig, den Codeausschnitt im Entwurf zu identifizieren.

Es ist folglich zwingend, dass die Implementation den Entwurf spiegelt. Das bedeutet, dass sich die Implementation jeweils getreu an den Entwurf hält. Alle Bezeichner werden exakt übernommen, die Struktur der Methoden und Klassen entspricht den entworfenen Funktionseinheiten. Jegliche Abweichungen sind zu vermeiden.

Fällt dem Team während der Implementation ein besserer Bezeichner ein, ist es natürlich legitim, diesen zu verwenden. Allerdings muss dann der Entwurf entsprechend angepasst werden, um die Realität der Implementation widerzuspiegeln. Oft ergeben sich während oder nach der Implementation tiefere Erkenntnisse, die zurückwirken auf den Entwurf. Auch in diesem Fall ist es sehr vorteilhaft, diese Erkenntnisse zu verwenden. Allerdings müssen die Erkenntnisse sowohl in der Implementation als auch im Entwurf ihren Niederschlag finden. Beides muss angepasst werden.

Teil II

Einführung

In diesem zweiten Teil des Buchs wird zunächst die Syntax der Flow Design Diagramme erläutert. Die einzelnen Elemente werden kurz dargestellt. Anschließend wird auf die Vorgehensweise in einem Softwareentwicklungsprozess eingegangen. Dort wird kurz beschreiben, welche Schritte ein Team jeweils in den Iterationen durchlaufen sollte.

Im dritten Teil wird dargestellt, wie die einzelnen Bestandteile von Flow Design in Code übersetzt werden können. Dies wird für C# dargestellt.

Dieser Teil des Buchs dient als Nachschlagewerk für Detailfragen. Beispiele zur Methode sowie umfangreiche Erläuterungen finden Sie in Teil I.

Syntax

In diesem Teil des Buches wird die Syntax der Diagramme beschrieben. Ganz bewusst wird darauf verzichtet, auf die mögliche Übersetzung in eine konkrete Programmiersprache hinzuweisen. Der Übersetzung der Syntax in Code ist ein gesonderter Abschnitt weiter unten gewidmet.

System-Umwelt-Diagramm

Ein System-Umwelt-Diagramm dient dazu, das Softwaresystem innerhalb seiner Umwelt darzustellen. Dabei geht es während der Beschäftigung mit den Anforderungen darum herauszufinden, welche relevanten *Rollen* und *Ressourcen* sich in der Umwelt des Systems befinden. Rollen sind abhängig vom System, während das System von Ressourcen abhängig ist. Der Sinn dieser Betrachtung liegt darin, den Kern des Systems, seine *Domänenlogik*, von der Umwelt abzugrenzen. Das System mit seinem Kern in der Mitte soll an eine Zelle erinnern, deren Kern von einer Membran umschlossen ist. Die Membran des Systems übernimmt eine ähnliche Aufgabe wie bei einer realen Zelle. Es soll erreicht werden, dass definierte Übergänge in das System sowie aus ihm heraus bestehen. Auf diese Weise wird erreicht, dass Veränderungen in der Umwelt des Systems nicht ungehindert auf den Kern des Systems durchschlagen. Stattdessen kann auf Veränderungen reagiert werden, in dem an den Übergangspunkten zwischen Umwelt und System Anpassungen vorgenommen werden. Damit das praktikabel ist, werden die Rollen über *Portale* an das System angebunden. Ferner wird das System über *Provider* an Ressourcen angebunden.

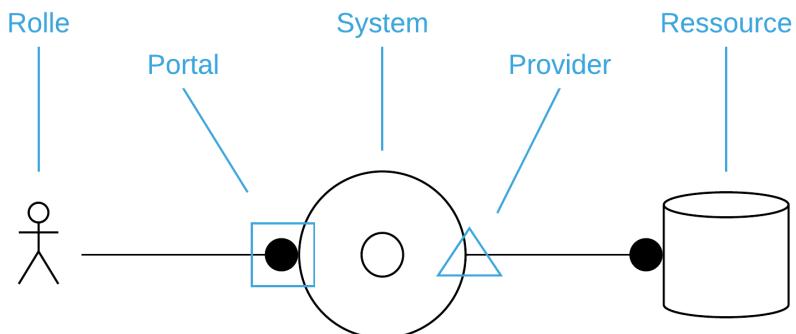


Abbildung 161: System-Umwelt-Diagramm

In einem System-Umwelt-Diagramm können mehrere Rollen vom System abhängen. Ferner kann das System von mehreren Ressourcen abhängen. Ziel der Analysephase ist es, gemeinsam mit dem Product Owner herauszuarbeiten, welche Rollen und Ressourcen er aus Sicht der Anforderungen in der Umwelt sieht. Diese Analyse ist nicht technisch geprägt, sondern orientiert sich an den Anforderungen. In ein System-Umwelt-Diagramm werden keine Portale und Provider eingezeichnet. Dies dient oben nur der Veranschaulichung. Die Regel zur Einführung von Portalen und Providern in der späteren Umsetzung ist so einfach, dass es nicht notwendig ist, diese jeweils einzuziehen: jede Rolle erhält ihr Portal und jede Ressource ihren Provider. Portale und Provider sind somit Implementationsdetails.

Speziell bei den Ressourcen muss darauf geachtet werden, dass diese nicht technisch dargestellt werden, sondern inhaltlich. Es geht darum herauszuarbeiten, dass das System bspw. Kundendaten benötigt. Ob diese in einer Datei oder einer Datenbank abgelegt werden, ist für die Analysephase nicht relevant. Ressource sind also die Kundendaten und nicht eine Datei oder eine Datenbank. Die folgende Abbildung zeigt ein Beispiel für ein System-Umwelt-Diagramm mit mehreren Rollen und Ressourcen.

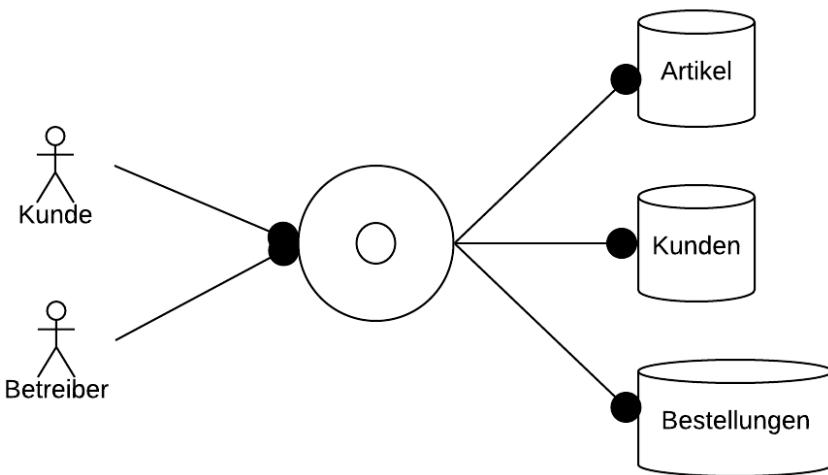


Abbildung 162: System-Umwelt-Diagramm für einen Online Shop

Aspekte

Aus dem System-Umwelt-Diagramm leitet sich ab, dass es drei fundamental unterschiedliche Aspekte in einem Softwaresystem gibt: Portale, Domänenlogik und Provider. Diese drei Arten von Funktionseinheiten sind für so unterschiedliche Aspekte verantwortlich, dass es sinnvoll ist, ihnen jeweils ein eigenes Symbol zuzuordnen. Auf diese Weise soll deutlich gemacht werden, dass die Umsetzung dieser drei Aspekte niemals vermischt werden darf. Domänenlogik hat nichts in der Ui zu suchen und Ressourcenzugriffe gehören nicht in die Domänenlogik.

Portal

Ein Portal ist eine dünne Softwareschnittstelle, die für den Zugang einer Rolle zum System verantwortlich ist. Die konkrete Ausgestaltung eines Portals hängt von der jeweiligen Rolle ab. Ist die Rolle ein Benutzer, kommen grafische Benutzerschnittstellen (GUI) oder Konsolen Uis in Betracht. Abhängig von den nicht-funktionalen Anforderungen handelt es sich dann um ein Web- oder Desktop-Ui.

Als Rollen werden auch Fremdsysteme verstanden, die Zugriff auf das betrachtete System benötigen. In dem Fall wird das Portal bspw. als HTTPS/REST Schnittstelle implementiert.

Portale werden in den Flow Design Entwürfen als Rechtecke dargestellt.



Abbildung 163: Portal

Domänenlogik

Im Kern des Systems befindet sich die Domänenlogik. Sie ist dafür verantwortlich, das gewünschte Verhalten des Systems herzustellen. Domänenlogik ist frei von Benutzerinteraktionen sowie frei von Zugriffen auf Ressourcen. Die inhaltlich so deutlich unterschiedlichen Aspekte sollten ebenso deutlich getrennt werden. Ferner macht die Domänenlogik im Kern das System aus. Portale und Provider können ausgetauscht oder angepasst werden. Im Kern wird das System geprägt von seiner Domänenlogik. Um diesen

wesentlichen Teil des Systems leicht automatisiert testen zu können, ist es sehr vorteilhaft, ihn frei zu halten von Benutzerinteraktionen und Ressourcenzugriffen.



Abbildung 164: Domänenlogik

Provider

Ein Provider ist, wie ein Portal, eine dünne Softwareschnittstelle. Sie dient dazu, den Zugriff des Systems auf eine Ressource zu kapseln. Das System soll nicht unmittelbar von der Ressource abhängig sein. Dann würde sich die API der Ressource, insbesondere spezielle Datentypen, in das System ausbreiten und dort zu ungewollten Abhängigkeiten führen.



Abbildung 165: Provider, alternative Darstellungen

In den frühen Iterationen eines Systems ist es manchmal sinnvoll, einen Provider in einer sehr einfachen Technologie zu realisieren. Der Fokus in den frühen Iterationen liegt darauf, so schnell wie möglich erstes Feedback vom Product Owner zu erhalten. Um schnell voran zu kommen, können daher Provider zunächst in einer einfachen Technologie gelöst werden, die später durch eine andere Technologie ersetzt wird. So können bspw. zu Beginn die Artikeldaten aus einer einfachen Textdatei gelesen werden. So lange es sich um wenige Daten handelt, ist das schnell realisiert und für ein erstes Feedback ausreichend. Erst wenn sich aufgrund funktionaler oder nicht-funktionaler Anforderungen die Notwendigkeit ergibt, wird der Provider auf eine andere Technologie wie bspw. eine relationale Datenbank umgestellt. Die Austauschbarkeit einer Ressource durch die dünne Schnittstelle des Providers dient also nicht primär dazu, so flexibel zu sein, unterschiedliche Datenbankhersteller zu unterstützen. Es geht darum, die

Domänenlogik frei zu halten von jeglichen Ressourcenzugriffen. Daraus ergibt sich dann zusätzlich die Freiheit, Provider zunächst in einer vereinfachten Form zu realisieren, um schnell Feedback erhalten zu können.

Datenflüsse und Nachrichten

Funktionseinheit werden mit Datenflüssen verbunden. Datenflüsse dienen dazu, Nachrichten zwischen Funktionseinheiten auszutauschen.

Datenfluss

Datenflüsse werden in Flow Design als Pfeile dargestellt. Die Daten werden in Klammern an den Pfeil annotiert. Jeder Datenfluss hat eine Richtung. Die Daten fließen zwischen zwei Funktionseinheiten, von der Quelle zur Senke.

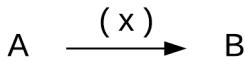


Abbildung 166: Datenfluss von A nach B

Zu jedem Datenfluss ist es wichtig zu erfahren, von wo nach wo die Daten fließen. Mindestens genauso wichtig ist die Information darüber, welche Daten fließen. In den folgenden Abschnitten wird dargestellt, wie die Daten beschrieben werden können.

Benennung

In den meisten Fällen ist es nicht notwendig, den Datenfluss selbst zu benennen. Hat eine Funktionseinheit einen einzigen eingehenden Datenfluss, muss dieser nicht benannt werden, da es nicht notwendig ist, ihn von anderen eingehenden Datenflüssen zu unterscheiden. Das gleiche gilt für einen einzigen ausgehenden Datenfluss.

Allerdings gibt es Fälle, in denen mehrere ein- oder ausgehende Datenflüsse erforderlich sind. In diesen Fällen ist es notwendig, die Datenflüsse benennen zu können. Dazu verwendet Flow Design die “Punkt-Notation”. Ein Datenfluss kann sowohl an seinem Anfang als auch an seinem Ende benannt werden. Die Benennung am Anfang des Datenflusses benennt damit den Ausgang der Funktionseinheit. Entsprechend benennt das Ende des Datenflusses den Eingang einer Funktionseinheit.

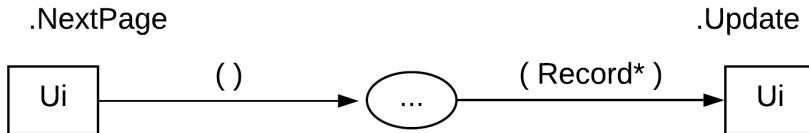


Abbildung 167: Datenflüsse, benannt am Anfang bzw. Ende

In der Abbildung ist der ausgehende Datenfluss der Ui benannt mit "NextPage". Auf der rechten Seite ist der eingehende Datenfluss benannt mit "Update". Die Ui ist ein typischer Vertreter für benannte Datenflüsse. Die Ui wird regelmäßig als Klasse realisiert, da sie typischerweise mehrere Ein- und Ausgänge hat. Folglich müssen diese Ein- und Ausgänge benannt werden, um eine Vorgabe für die Implementation zu liefern.

Die Bezeichner von ein- oder ausgehenden Datenflüssen werden in die Implementation übernommen. Bezeichner von eingehenden Datenflüssen werden zu Methodennamen, Bezeichner von ausgehenden Datenflüssen zu Event- oder Callbackbezeichnern.

Konvention Groß-/Kleinschreibung

Bei der Beschreibung des Inhalts der Datenflüsse wird die Konvention verwendet, dass die Schreibweise des Anfangsbuchstabs darüber Auskunft gibt, ob es sich um einen Typ oder den Inhalt handelt. Typen werden mit großem Anfangsbuchstaben geschrieben, also bspw. *Kunde*. Mit kleinem Anfangsbuchstaben beginnen einerseits übliche Standardtypen wie *string*, *int*, *bool*, etc. Andererseits steht ein kleiner Anfangsbuchstabe quasi für einen Variablen- oder Parameternamen. Häufig wird der Typ dann weggelassen, wenn er aus dem Kontext ersichtlich ist. Steht bspw. *name* an einem Datenfluss, so ist aufgrund des Bezeichners klar, dass hier inhaltlich ein Name fließt. In der Regel wird man einen Namen als *string* typisieren. Sollte sich die Typisierung nicht aus dem Kontext ergeben, kann der Typ auch explizit angegeben werden.

Typ ergänzen

Soll zusätzlich zum Inhalt auch der Typ benannt werden, wird er einfach mit einem Doppelpunkt an den Bezeichner angehängt. Um zu präzisieren, dass der *name* als *string* zu implementieren ist, kann man dann schreiben: *name : string*.

Die Typinformation oder der Bezeichner können aber weggelassen werden, wo dies nicht zu mehr Klarheit führt. Im Vordergrund steht eine leichtgewichtige Syntax, die durchaus gewisse Unschärfen haben kann, zugunsten einer leichten Veränderbarkeit. Folgende Kombinationen können verwendet werden:

string	Nur die Typinformation, Inhalt ergibt sich aus dem Kontext.
name : string	Bezeichnung des Inhalts sowie Angabe des Typs.
name	Kleinschreibung, also Variablenname, Bezeichnung des Inhalts, Typ ergibt sich aus dem Kontext.
Kunde	Nur Typinformation. Bei einem eigenen Typ bringt ein zusätzlicher Bezeichner selten mehr Informationen.
kunde	Nur der inhaltliche Bezeichner. Der Typ ergibt sich aus dem Kontext.
k u n d e : Kunde	Inhaltliche Bezeichnung sowie Typinformation, hier redundant.

Tupel

Ein Tupel, bestehend aus mehreren Werten, wird einfach als kommagetrennte Liste beschrieben. Ein Tupel bestehend aus den Werte x und y wird somit wie folgt beschrieben:

(x , y)

Tupel sind natürlich nicht auf zwei Elemente beschränkt.

(x , y , z)

Eigener Datentyp

Häufig trägt ein eigener Datentyp dazu bei, eine Lösung leichter verständlich zu machen. Ferner dient ein Datentyp dazu, die Anzahl der Tupel zu reduzieren. Man könnte folgendes als Tupel schreiben:

```
( strasse, hausnummer, plz, ort )
```

Naheliegender ist allerdings, dafür den Typ *Adresse* einzuführen, mit den entsprechenden Eigenschaften.

Wird in einem Flow Design ein Datentyp verwendet, der speziell ist für dieses Softwaresystem und nicht aus einem Framework stammt, muss er weiter spezifiziert werden. Schließlich muss man den Entwurf einerseits verstehen können, andererseits muss er ausreichend detailliert definiert sein, um ihn später implementieren zu können. Für die Definition eines Datentyps werden Tabellen verwendet, in denen die Bestandteile des Typs eingerückt werden. Der Datentyp *Adresse* kann wie folgt definiert werden:

Adresse

```
-----
```

```
Strasse  
Hausnummer  
Plz  
Ort
```

Wenn es zur Klarheit beiträgt, können auch hier wieder Typen ergänzt werden:

Adresse

```
-----
```

```
Strasse : string  
Hausnummer : string  
Plz : string  
Ort : string
```

Datentypen können geschachtelt werden:

Kunde

```
-----
```

```
Name  
Vorname  
Adresse
```

Als zusätzliche Vereinfachung kann auch der geschachtelte Datentyp “inline” definiert werden:

```
Kunde
-----
Name
Vorname
Adresse
-----
Strasse
Hausnummer
Plz
Ort
```

Die Regeln zur Definition von Datentypen sind nicht in Stein gemeißelt. Es steht einem Team frei, diese nach ihren Bedürfnissen anzupassen. Allerdings sollte dabei darauf geachtet werden, dass die Definition so präzise ist, dass alle Entwickler das gleiche Verständnis darüber haben. Vor allem muss eine Übersetzung in Konstrukte der verwendeten Programmiersprache möglich sein. Gleichzeitig sollte die Syntax so leichtgewichtig sein, dass schnell unterschiedliche Varianten diskutiert werden können. Zu viele syntaktische Details sind dabei eher hinderlich.

Aufzählungen

Neben einem Datentyp, in dem unterschiedliche Eigenschaften zusammengefasst sind, ist eine *Aufzählung* von Daten ein wichtiges Element des Entwurfs. Eine Aufzählung wird je nach verwendeter Programmiersprache übersetzt in ein Array, eine Liste oder auch ein Interface wie *ICollection* oder *IEnumerable*. Im Entwurf wird bewusst recht abstrakt von einer Aufzählung gesprochen, um keine zu frühe Festlegung auf eine bestimmte Datenstruktur vorzunehmen. Ferner ist der Entwurf unabhängig von einer konkreten Programmiersprache. Letztlich muss ein Entwurf in jeder Programmiersprache realisiert werden können. Insofern wäre es ungünstig, im Entwurf eine strikte Festlegung zu treffen.

Eine Aufzählung von Elementen wird durch ein Sternchen “*” ausgedrückt. So bedeutet *string** eine Aufzählung von Strings. In der späteren Umsetzung kann dies ein Array von Strings sein oder auch eine Liste, je nachdem, was die verwendete Programmiersprache bietet und was angemessen für die Lösung ist.

Im Abschnitt über Streams wird deutlich werden, dass es wesentlich ist, die Daten eines Datenflusses in Klammern zu setzen. Wie später deutlich wird, müssen die beiden folgenden Fälle unterschieden werden können:

(x^*) vs. (x) *

Im einen Fall gehört das Sternchen zum x und drückt eine Aufzählung aus. Im anderen Fall gehört das Sternchen zum Datenfluss und drückt aus, dass dieser mehrfach stattfindet. Würde man auf die Klammern verzichten, wären die beiden Fälle nicht mehr unterscheidbar. Bevor die Streams behandelt werden, geht es im folgenden Abschnitt zunächst um optionale Ausgänge.

Optionaler Ausgang

Funktionseinheiten können über Eingänge und Ausgänge verfügen. Der Ausgang kann entfallen, wenn die Funktionseinheit einen Seiteneffekt bewirkt, wie etwa das Speichern von Daten in einer Datei. Manchmal werden allerdings Funktionseinheiten benötigt, die optional einen Wert liefern, dies also nicht mit jedem Aufruf tun. Es muss daher möglich sein, von der strikten 1:1 Beziehung zwischen ein- und ausgehendem Datenfluss abweichen zu können. Die folgende Abbildung zeigt dies am Beispiel eines Weckers. Eine Funktionseinheit *Wenn Restzeit abgelaufen* soll am Ausgang nur Daten liefern, wenn die Weckzeit in der Vergangenheit liegt.

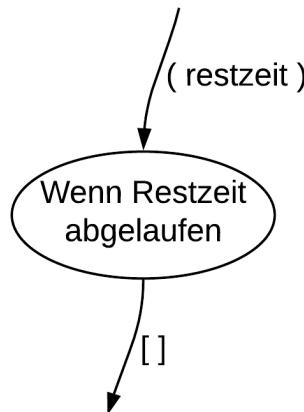


Abbildung 168: Funktionseinheit mit optionalem Ausgang

In diesem Fall wird der optionale Ausgang durch eckige Klammern anstelle der runden Klammern ausgedrückt. Dies bedeutet, dass für einen eingehenden Datenfluss optional ein ausgehender Datenfluss stattfindet. Die Beziehung zwischen ein- und ausgehendem Datenfluss ist nicht mehr 1:1 sondern 1:0..1.

Streams

Durch eine Verallgemeinerung des optionalen Ausgangs erhält man einen *Stream*. Beim Stream ist die Beziehung zwischen ein- und ausgehendem Datenfluss 1:n. Für einen eingehenden Datenfluss können beliebig viele Datenflüsse am Ausgang produziert werden. Insbesondere auch kein Datenfluss am Ausgang. Der optionale Fall ist also im Stream enthalten. Ausgedrückt wird ein Stream durch ein Sternchen außerhalb der Klammern.

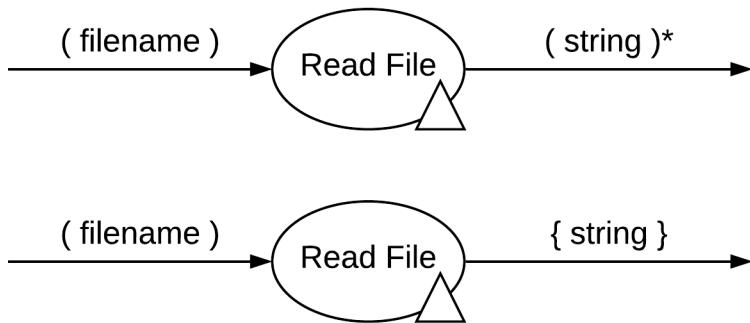


Abbildung 169: Stream mit Sternchen außerhalb der Klammern bzw. geschweiften Klammern

Alternativ zum Sternchen kann der Stream auch durch geschweifte Klammern ausgedrückt werden. Durch geschweifte Klammern lässt sich ein Stream von einem optionalen Datenfluss unterscheiden. Meist ergibt sich diese Unterscheidung ohnehin aus dem Kontext des Entwurfs, so dass man grundsätzlich beides mit einem Sternchen ausdrücken kann. Zur Verdeutlichung und Präzisierung können allerdings auch die unterschiedlichen Klammern verwendet werden. Die Bedeutung der Klammern zeigt folgende Tabelle.

()	1:1
[]	1:0..1
{ }	1:n
()*	1:n

Abhängigkeiten

Geht man nach Flow Design im Entwurf vor, wird sich nur selten die Notwendigkeit ergeben, Abhängigkeiten explizit darzustellen. Flow Design Diagramme legen den Fokus auf die durchzuführenden Aktionen und den Fluss der Daten. Abhängigkeiten zwischen Funktionseinheiten ergeben sich implizit innerhalb einer Integration. Aufgabe einer Integration ist es, den Datenfluss herzustellen. Dazu macht sich die Integration von den zu integrierenden Funktionseinheiten abhängig. Alle anderen Funktionseinheiten sind voneinander unabhängig.

Darstellung

Abhängigkeiten werden in Flow Design dargestellt durch eine Linie mit einem Punkt am Ende.

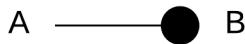


Abbildung 170: A ist abhängig von B

Implizit ergeben sich aus einem Flow, der Verfeinerungen enthält, eine Reihe von Abhängigkeiten.

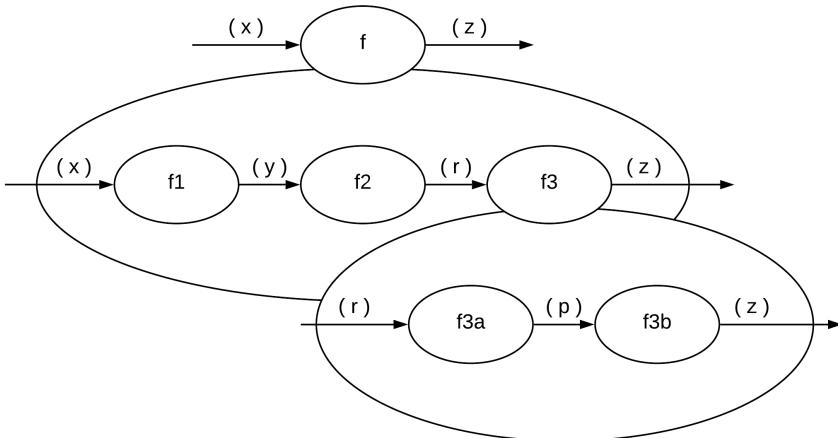


Abbildung 171: Funktionseinheit mit Verfeinerungen

Hier wird f verfeinert zu f_1 , f_2 und f_3 . Die Funktionseinheit f_3 ist dann nochmals verfeinert zu f_{3a} und f_{3b} . Aus diesem Entwurf ergeben sich die folgenden Abhängigkeiten.

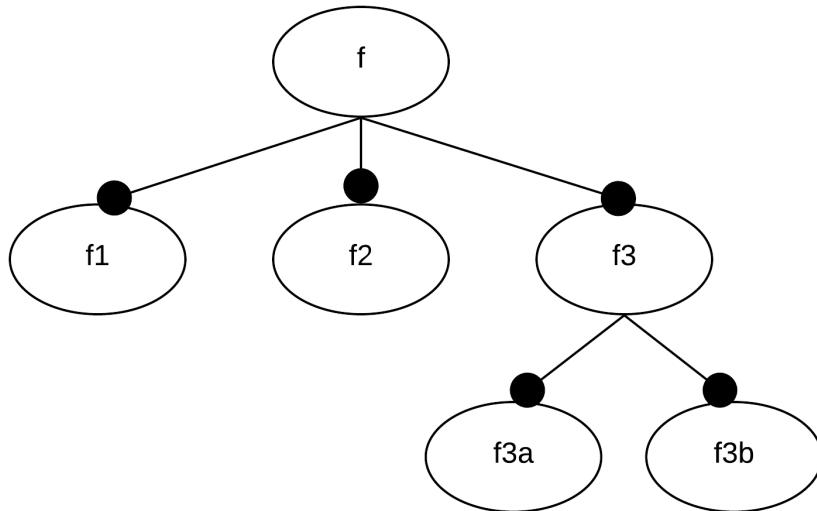


Abbildung 172: Zugehöriges Abhängigkeitsdiagramm

Die Aufgabe von f ist es, den Datenfluss bestehend aus f_1 , f_2 und f_3 herzustellen. Folglich ist f von diesen Funktionseinheiten abhängig. Ähnliches gilt für f_3 . Aufgabe dieser Funktionseinheit ist es, den Datenfluss bestehend aus f_{3a} und f_{3b} zu realisieren. Folglich ist f_3 abhängig von f_{3a} und f_{3b} . Durch die klare Trennung von Integration und Operation, stellen diese Abhängigkeiten kein Problem dar. Die Funktionseinheiten f und f_3 sind Integrationen, während f_1 , f_2 , f_{3a} und f_{3b} Operationen sind.

Zustand

Jede Funktionseinheit kann Zustand halten, ohne dass dies auf besondere Weise im Entwurf gekennzeichnet werden muss. Flow Design bevorzugt nicht die funktionale Programmierung, sondern kann genauso gut auch mit objektorientierten Sprachen verwendet werden. Zustand ist einerseits völlig natürlich und für viele Szenarien erforderlich, daher muss er nicht explizit angegeben werden. Andererseits sind mit Zustand besondere Herausforderungen verbunden, wenn Verteilung oder Nebenläufigkeit ins Spiel kommen. Insofern kann es hilfreich sein, im Entwurf auszudrücken, wo sich innerhalb eines Entwurfs Zustand befindet.

Zustand innerhalb einer Funktionseinheit

Zustand kann in Flow Design auf zwei Arten notiert werden:

- Eine Funktionseinheit ist zustandsbehaftet.
- Zustand ist explizit im Flow modelliert.

Hält eine Funktionseinheit Zustand, kann dies durch eine Tonne an der Funktionseinheit notiert werden.

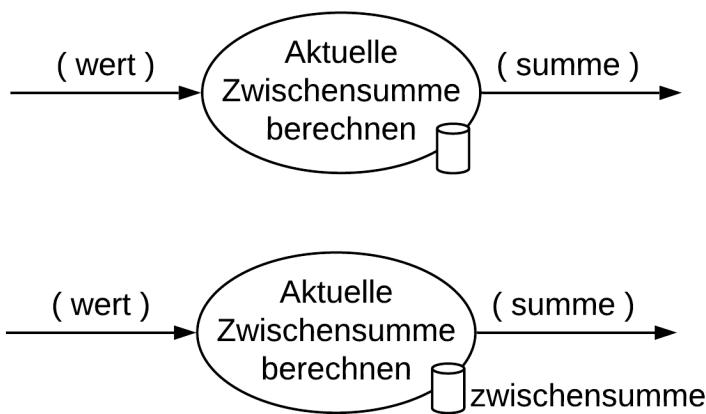


Abbildung 173: Funktionseinheit mit Zustand. Mal unbenannt, mal benannt

Der Zustand kann inhaltlich häufig aus dem Kontext abgeleitet werden. Wenn es der Verständlichkeit des Entwurfs dient, kann der Zustand zusätzlich benannt werden. Auf diese Weise können Missverständnisse über den

Zustand vermieden werden. Wird der Zustand bspw. *Index* bzw. *Number* bezeichnet, kann daraus abgeleitet werden, dass er 0-basierend oder 1-basierend ist. Im Zweifel sollte der Zustand benannt werden, um für Klarheit zu sorgen.

In der obigen Abbildung bedeutet die Zustandstonne an den Funktionseinheiten, dass die Funktionseinheit *Aktuelle_Zwischensumme_berechnen* zustandsbehaftet ist. Die Funktionseinheit erhält als Input jeweils einen Wert und liefert als Output die jeweils aktuelle Summe über alle bislang reingereichten Werte. Diese Funktionalität ist nur mittels Zustand realisierbar. Wenn die Funktionseinheit als Methode implementiert wird, genügt es nicht, lokale Variablen in der Methode anzulegen, da deren Werte beim Verlassen der Methode verloren gehen. Folglich muss die Methode ihren Zustand über mehrere Aufrufe behalten können. Dies ist bspw. über ein Feld der umschließenden Klasse realisierbar.

Die Zustandstonne sollte nicht verwechselt werden mit einem Ressourcenzugriff. Obschon häufig für Datenbanken eine Tonne zur Darstellung verwendet wird, meint die Tonne hier als Annotation der Funktionseinheit, dass diese zustandsbehaftet ist. Funktionseinheiten die auf Ressourcen zugreifen, werden mit einem Dreieck markiert.

Zustand explizit im Flow

Eine Funktionseinheit, die Zustand hält, hat damit in manchen Fällen zu viele Verantwortlichkeiten. Prinzipiell kann das Halten von Zustand auf einer sehr niedrigen Abstraktionsebene als ein eigenständiger Aspekt betrachtet werden. Kommt der Funktionseinheit noch eine weitere Aufgabe zu, wie etwa die Durchführung einer Berechnung, wären bereits mehrere Aspekte vermischt. Es ist nicht sinnvoll, in allen Fällen so streng zu urteilen und damit Zustand generell aus Funktionseinheiten zu verbannen. Auf der anderen Seite kann es in manchen Fällen sinnvoll sein. So kann die Testbarkeit einer Funktionseinheit vereinfacht werden, wenn sie von ihrem Zustand befreit wird. Logischerweise stellt sich dann die Frage, wie die gesamte Aufgabe bewerkstelligt werden kann, denn die inhaltliche Notwendigkeit für Zustand bleibt ja bestehen.

Statt den Zustand innerhalb einer Funktionseinheit zu halten, kann der Zustand alternativ explizit als eigenständiger Aspekt in den Datenfluss gestellt werden.

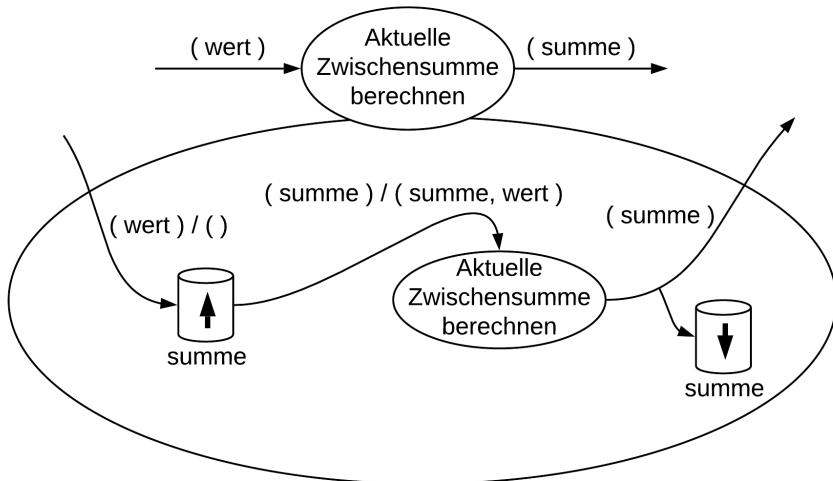


Abbildung 174: Zustand explizit im Flow

Es stehen zwei Symbole für den Umgang mit Zustand zur Verfügung: das Lesen und Schreiben des Zustands. Die Tonne mit dem Pfeil nach oben liefert den Zustand und stellt ihn so als Datenfluss zur Verfügung. Die Tonne mit dem Pfeil nach unten nimmt ein Datum aus dem Datenfluss auf und verwahrt es intern als Zustand.

Im gezeigten Entwurf wird die Slash-Notation verwendet um auszudrücken, dass zum Lesen des Zustands der Wert nicht benötigt wird. Die Notation $(\text{wert}) / ()$ bedeutet, dass die Nachricht *wert* ignoriert wird. Als Ergebnis der Operation *Zustand lesen* wird die aktuelle *summe* geliefert. Da die nächste Funktionseinheit neben der Summe auch den Wert benötigt, wird dieser im Datenfluss, ebenfalls durch die Slash-Notation, wieder ergänzt. Die Nachricht $(\text{summe}) / (\text{wert}, \text{summe})$ drückt aus, dass der Vorgänger die Summe liefert, während an den Nachfolger ein Tupel aus dem Wert und der Summe übergeben wird.

Auf der äußersten Ebene hat die Funktionseinheit *Aktuelle Zwischensumme berechnen* Zustand. In der Verfeinerung kann man erkennen, dass der Zustand zu Beginn gelesen wird. anschließend werden die aktuelle Summe sowie der zu addierende Wert an die innere Funktionseinheit *Aktuelle Zwischensumme berechnen* übergeben. Diese kann daraus die Summe neu berechnen. Bevor dieser Wert zurückgeliefert wird, muss der Zustand noch aktualisiert werden. Aus diesem Grund fließt die Summe auch in die Zustandstonne hinein.

Lesende und schreibende Zugriffe auf Zustand, der wie oben dargestellt in den Datenfluss gestellt ist, werden mittels Felder der Klasse realisiert.

Split

Manchmal ist es in einem Flow Design erforderlich, dass ein und die selben Daten zu mehr als einer Funktionseinheit fließen. In diesem Fall kann ein Datenfluss verzweigen und die Daten an mehrere Funktionseinheit liefern. Dies bezeichnen wir als *Split*.

Darstellung

Im folgenden Beispiel soll ein x sowohl an die Funktionseinheit $f1$ als auch an $f2$ geliefert werden.

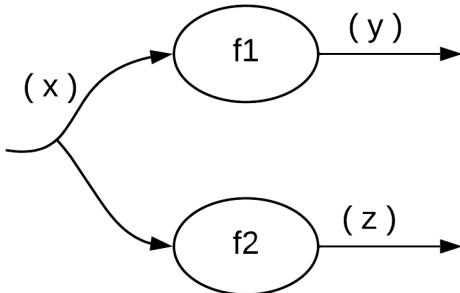


Abbildung 175: Split

Durch den Split des Datenflusses fließen die Daten nun sowohl zu $f1$ als auch zu $f2$. Zu beachten ist, dass die Ausführung nach wie vor synchron sequentiell erfolgt. Das oben abgebildete Flow Design Diagramm drückt nicht aus, welche der beiden Funktionseinheiten zuerst ausgeführt wird. Es zeigt damit, trotz synchron sequentieller Ausführung, allerdings das Potenzial für eine Parallelisierung auf. Dieser Entwurf weist explizit darauf hin, dass es egal ist, ob zunächst $f1$ oder $f2$ ausgeführt wird.

Sollte die Reihenfolge relevant sein, müssen die beiden Funktionseinheiten hintereinander in einen Datenfluss gestellt werden, wie es die folgende Abbildung zeigt.

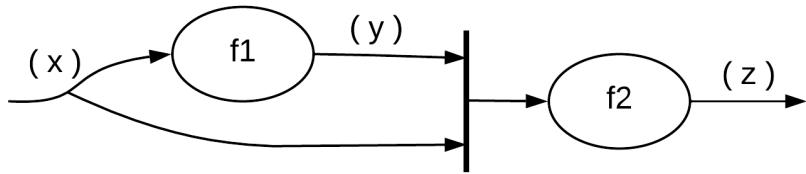


Abbildung 176: Split, Reihenfolge ist relevant

Bei diesem Split wird das x ebenfalls an f_1 und f_2 geleitet. Da hier die Reihenfolge relevant ist, sind die beiden Funktionseinheit hintereinander in einen Datenfluss gestellt. Vor f_2 kommt es dann zu einem *Join*.

Join

Benötigt eine Funktionseinheit Daten aus unterschiedlichen Quellen, hilft häufig ein *Join*. In einem Join werden mehrere Datenflüsse zusammengefasst. Dabei entsteht ein Datenfluss, der die Daten der eingehenden Datenflüsse als Tupel zusammenfasst.

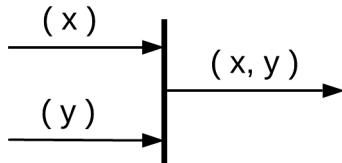


Abbildung 177: Join

In diesem Fall fließen x und y aus unterschiedlichen Quellen zum Join. Am Ausgang des Joins steht ein Tupel aus x und y zur Verfügung. Das Tupel muss am Ausgang des Joins nicht explizit notiert werden. Per Konvention liefert der Join ein Tupel aller eingehenden Daten.

Explizit

Das Zusammenführen mehrerer Datenflüsse ist die Aufgabe des *Join*. Wurde eine Aufgabe in einem Flow Design so aufgeteilt, dass mehrere Funktionseinheiten Zwischenergebnisse berechnen, müssen diese im Anschluss wieder zusammengefasst werden können. Diese als *Join* bezeichnete Aufgabe wird durch eine senkrechte Linie dargestellt. Der Join hat mehrere eingehende Datenflüsse sowie einen ausgehenden Datenfluss.

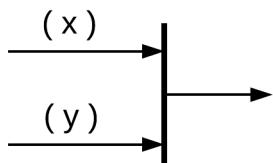


Abbildung 178: Join aus zwei Eingängen

Für einen Join gilt die Konvention, dass der ausgehende Datenfluss ein Tupel aller eingehenden Datenflüsse darstellt. Fließen also an drei Eingän-

gen jeweils (x) , (y) und (z) in den Join, steht am Ausgang das Tupel (x, y, z) zur Verfügung.

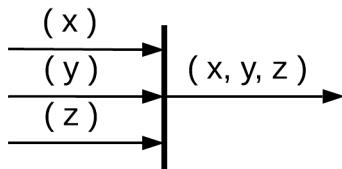


Abbildung 179: Join aus drei Eingängen, Ausgang explizit angegeben

Die explizite Angabe der Daten am Ausgang kann entfallen, da sie sich immer als Tupel aus den Eingangsdaten ergibt.

Slash Notation zur Abkürzung

Alternativ zur expliziten Notation kann der Join mit der *Slash Notation* abgekürzt werden. Folgende Abbildung zeigt dies.

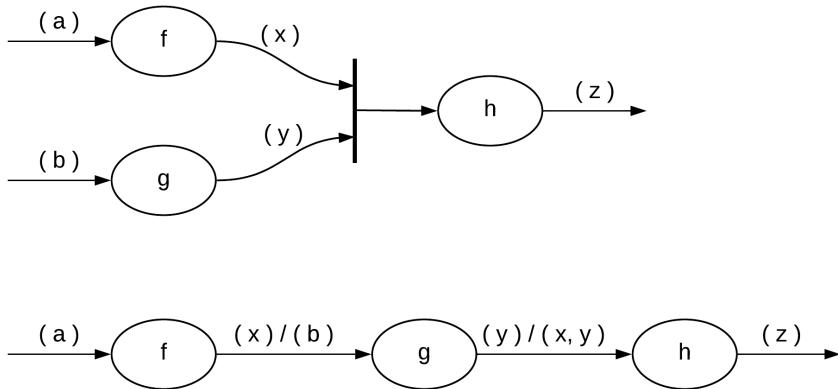


Abbildung 180: Join explizit sowie mittels Slash Notation

Die Abbildung zeigt zwei Varianten, die dasselbe ausdrücken. Durch die Slash Notation wird das Zusammenführen von Daten nicht explizit über Datenflüsse auf ein Join Symbol dargestellt, sondern in kürzerer Notation. Allerdings legt die untere Variante des Entwurfs die Reihenfolge der Ausführung von f und g fest. Da nun ein Datenfluss von f nach g stattfindet, muss f vor g ausgeführt werden. Die obere Variante lässt diese Entscheidung offen, obschon die Implementation in der Regel von links oben nach rechts unten stattfindet.

Die Bedeutung der Slash Notation ist einfach: vor dem Slash stehen die Daten, die vom Vorgänger, der Quelle des Datenflusses, geliefert werden. Nach dem Slash stehen die Daten, die zum Nachfolger, der Senke, fließen. Natürlich muss hier darauf geachtet werden, dass auf der rechten Seite des Slash nicht Daten verwendet werden, die erst später im Flow zur Verfügung gestellt werden.

Die folgende Abbildung zeigt einen fehlerhaften Entwurf. Der Wert y wird durch die Funktionseinheit h geliefert, die jedoch erst nach g ausgeführt wird. Insofern steht y auf dem Datenfluss von f nach g noch nicht zur Verfügung.

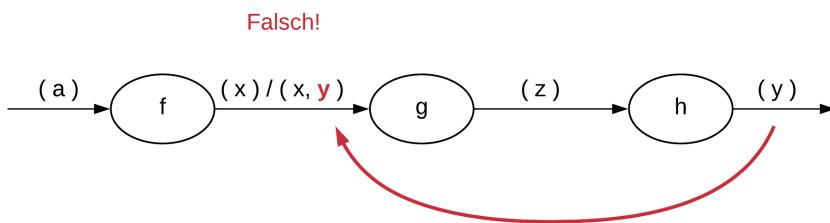


Abbildung 181: Fehlerhafter Slash, die Daten für y stehen noch nicht zur Verfügung

Ob man in einem Entwurf den expliziten Join verwendet oder die abgekürzte Slash Notation, hängt davon ab, wie übersichtlich die Struktur des Flows aussieht. In einem komplizierten Flow mit vielen Datenflüssen ist es oft leichter, den Zusammenhang durch einen expliziten Join zu erkennen. In einem solchen Fall auf die implizite Darstellung über Bezeichner innerhalb der Slash Notation zu setzen, führt eher zu undurchsichtigen Flows, die schwer nachzuvollziehen sind. In einem überschaubaren Flow spricht nichts dagegen, auf den expliziten Datenfluss und einen Join zu verzichten und das gleiche durch die Slash Notation auszudrücken. Häufiger kommt die Slash Notation allerdings beim Map zum Einsatz.

Map

Es gibt Fälle, in denen Ein- und Ausgang zweier Funktionseinheiten nicht zusammenpassen. Trotzdem soll jedoch ein Datenfluss zwischen beiden realisiert werden. Dann hilft häufig ein *Mapping* zwischen den Daten weiter. Die Ausgangsdaten der einen Funktionseinheit werden dazu so transformiert, dass sie zum Eingang der nachfolgenden Funktionseinheit passen.

Explizit

Immer wieder gibt es in einem Flow die Situation, dass eine Funktionseinheit die Daten, die gerade fließen, für ihre Arbeit nicht benötigt. Natürlich werden die Daten zu einem späteren Zeitpunkt im Flow sehr wohl benötigt. Insofern kann nicht ganz auf die Daten verzichtet werden. Es wäre allerdings auch nicht hilfreich, zunächst nicht benötigte Daten durch eine Funktionseinheit durchzureichen, weil diese dann Daten erhalten und wieder zurück liefern würde, für die sie selbst keine Verwendung hat. Hier hilft ein Mapping, welches die Nachricht des Datenflusses vollständig entfernt.

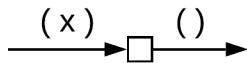


Abbildung 182: Map explizit

Durch eine solche *Map* Funktionseinheit werden Daten aus dem Datenfluss entfernt oder allgemeiner, transformiert bzw. gemappt. Folgende Abbildung zeigt, dass auf diese Weise auch einzelne Werte aus einem Tupel gemappt werden können.

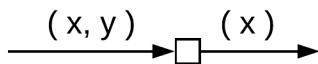


Abbildung 183: Map aus einem Tupel

Der Datenfluss liefert ein Tupel bestehend aus einem x und einem y . Benötigt der Nachfolger für seine Arbeit lediglich das y , kann dies wie oben gezeigt explizit ausgedrückt werden.

Die explizite Darstellung eines Mapping hat den Nachteil, dass damit der Flow in seiner Struktur etwas komplizierter wird. Es entsteht eine Funktionseinheit, die nur dafür zuständig ist, Daten zu mappen. Dabei findet keinerlei Transformation der Daten statt, sondern es wird lediglich aus einer Menge

von Variablen ausgewählt. In solchen Fällen empfiehlt sich die Verwendung der *Slash Notation* als Kurzform für Mappings, wie sie im folgenden Abschnitt beschrieben wird.

Slash Notation zur Abkürzung

Die Beispiele aus dem vorherigen Abschnitt kann man in einem Flow kürzer und einfacher notieren, in dem man die *Slash Notation* anwendet. Wie schon beim Join dargestellt, wird damit eine Auswahl getroffen, welcher Teil der Nachricht weiter fließt. Wurden beim *Join* zusätzliche Daten hinzugezogen, werden beim *Map* Daten weggelassen.

$$\begin{array}{c} (x) / () \\ \hline \longrightarrow \\ (x, y) / (x) \\ \hline \longrightarrow \end{array}$$

Abbildung 184: Map mit der Slash Notation

Der Ausdruck $(x) / ()$ bedeutet, dass die Quelle des Datenflusses ein x liefert. Dieses x wird allerdings nicht an den Nachfolger weitergegeben. Die Senke erhält keine Daten, ausgedrückt durch die leeren Klammern. Dies ist die einfachste Form eines Map, bei dem sämtliche Daten weggeschnitten werden. Es kann auch eine Auswahl aus den Daten getroffen werden, wenn dort ein Tupel anliegt.

$$(x, y, z) / (y, z)$$

Dieses Mapping drückt aus, dass von dem Tupel bestehend aus x, y und z lediglich y und z weitergegeben werden. Die Quelle des Datenflusses produziert ein Tupel (x, y, z) . An die Senke wird jedoch lediglich ein Tupel (y, z) weitergereicht.

Fallunterscheidung

Mit dem Split haben wir bereits eine Verzweigung eines Datenflusses gesehen. Beim Split fließen die Daten in allen Zweigen weiter. Eine andere Variante eines sich verzweigenden Datenflusses ist die *Alternative*, bei der eine Funktionseinheit eine Entscheidung trifft. In diesem Fall geht es nicht zwangsläufig auf allen Datenflüssen weiter.

Darstellung

Fallunterscheidungen sind ein wichtiges Mittel, um Komplexität zu reduzieren. Häufig lässt sich damit ein ursprünglich kompliziert aussehendes Feature auf mehrere Abstraktionsebenen zerlegen.

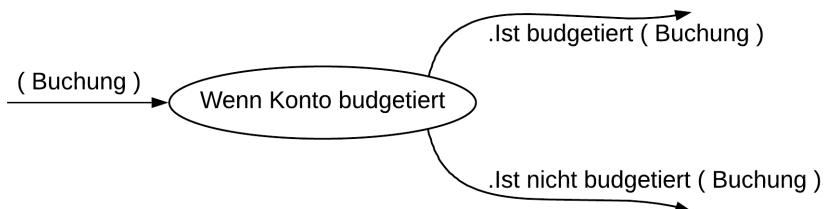


Abbildung 185: Fallunterscheidung

In diesem Beispiel soll eine Buchung, die vom Anwender erfasst wurde, verbucht werden. Der Ablauf ist aus domänensicht unterschiedlich, je nachdem, ob das Konto, auf das sich die Buchung bezieht, budgetiert ist oder nicht. Die Funktionseinheit *Wenn Konto budgetiert* trifft die Entscheidung aufgrund der eingehenden Buchung. Anschließend geht es entweder am Ausgang *Ist budgetiert* weiter oder am Ausgang *Ist nicht budgetiert*.

Die Funktionseinheit *Wenn Konto budgetiert* hat also zwei Ausgänge. Bei Fallunterscheidungen ist es notwendig, dass die Funktionseinheit, welche die Entscheidung trifft, entweder einen optionalen Ausgang hat, oder über mehr als einen Ausgang verfügt. Nur so kann es aufgrund der Entscheidung zu unterschiedlichen Datenflüssen kommen.

Auch im Beispiel der Wecker Anwendung wurde eine Fallunterscheidung benötigt.

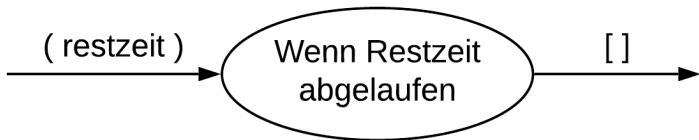


Abbildung 186: Optionaler Ausgang als Fallunterscheidung

In diesem Fall findet der ausgehende Datenfluss nur statt, wenn die Funktionseinheit festgestellt hat, dass die Restzeit abgelaufen ist, sprich die Weckzeit erreicht wurde.

Wenn eine Funktionseinheit über mehr als einen Ausgang verfügt ist es erforderlich, die Ausgänge mit der Punktnotation zu benennen. Nur so ist ersichtlich, welche Bedeutung die einzelnen Ausgänge haben. Ferner dient der Bezeichner später in der Implementation ebenfalls zur Benennung.

Verfeinerung

Das Verfeinern von Funktionseinheiten ist, wie die Fallunterscheidung, ein wesentliches Werkzeug der Informatik. *Teile und herrsche* sagten schon die Römer. In Flow Design können Funktionseinheiten in beliebiger Tiefe hierarchisch zerlegt werden.

Notwendigkeit von Verfeinerungen

Durch die Analyse der Anforderungen gelangen wir zu Dialogen und Interaktionen. Beim Übergang von den Anforderungen zum Entwurf wird für jede Interaktion eine Funktionseinheit im Flow Design notiert. Aufgabe dieser Funktionseinheit ist es, die gesamte Domänenlogik bereitzustellen, die für diese Interaktion benötigt wird.

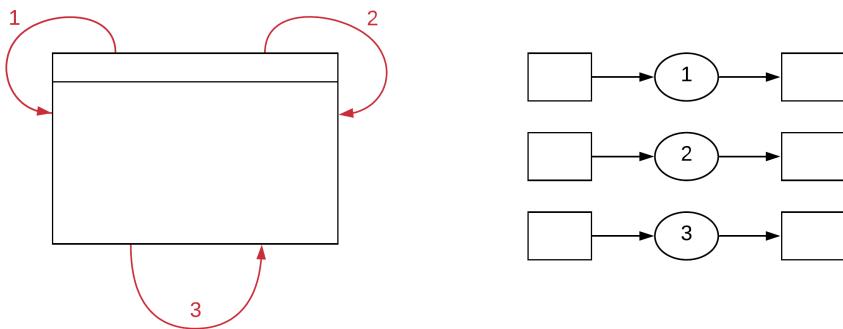


Abbildung 187: Vom Interaktionsdiagramm zum Entwurf der obersten Ebene

In seltenen Fällen mag die Domänenlogik so trivial sein, dass sie vollständig durch eine einzige Funktionseinheit realisiert werden könnte. In den meisten Fällen wird sie jedoch so umfangreich sein, dass eine Verfeinerung durchgeführt werden muss. Eine solche Verfeinerung ist zum einen erforderlich, um die unterschiedlichen Aspekte zu trennen. Nur so wird die Wandelbarkeit des Softwaresystems hergestellt. Zum anderen ist eine Verfeinerung die Voraussetzung für die Arbeitsorganisation. Selbst wenn das Softwaresystem nur von einem einzigen Entwickler hergestellt wird, ist es für diesen sinnvoll, jeweils kleine Einheiten zu implementieren. Der Grund ist mindestens die bessere Testbarkeit kleiner Einheiten. Spätestens bei einer verteilten Entwicklung durch ein Team von Entwicklern ist eine Verfeinerung die Voraussetzung für ein flüssiges arbeitsteiliges Vorgehen.

Darstellung

Die Verfeinerung einer Funktionseinheit wird durch eine Ellipse dargestellt, welche die Verfeinerung umschließt. Die Verfeinerung der Funktionseinheit wird innerhalb dieser Ellipse dargestellt und wie durch eine Lupe betrachtet. Es zeigen sich die Details.

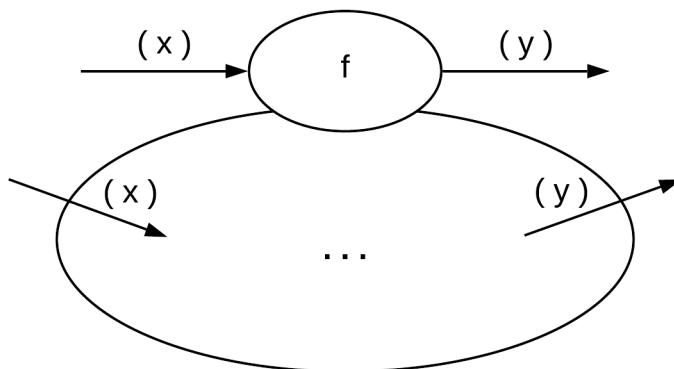


Abbildung 188: Verfeinerung der Funktionseinheit f

Eine solche Verfeinerung kann in beliebiger Tiefe durchgeführt werden. Jede einzelne Funktionseinheit innerhalb einer Verfeinerung kann erneut weiter verfeinert werden.

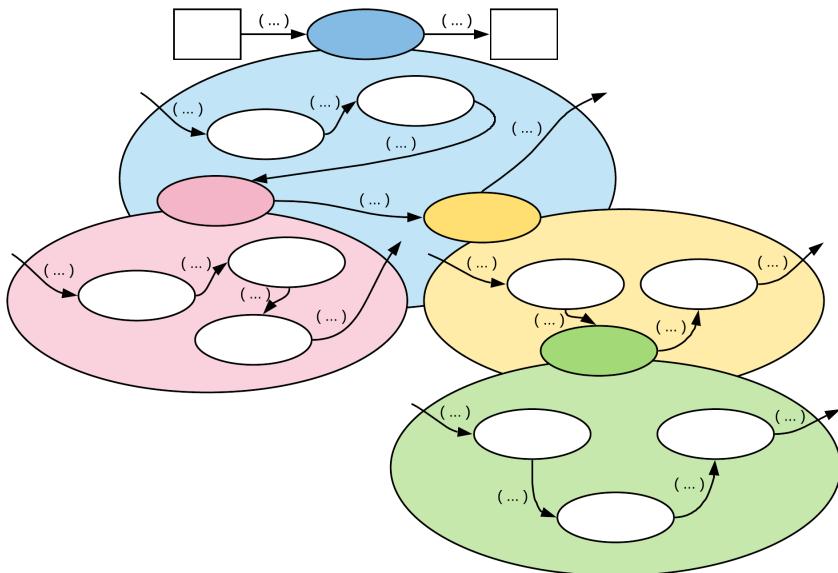


Abbildung 189: Verfeinerung in beliebiger Tiefe

Da die Verfeinerungsebenen nicht durch eine Struktur abgebildet werden, die darauf beruht, dass die Elemente in der Struktur enthalten sind, ist tatsächlich eine unendliche Tiefe erreichbar. Die Abbildung in Code erfolgt nicht dadurch, dass die oberste Ebene durch eine Klasse abgebildet wird und die darin enthaltene nächste Ebene dann durch Methoden. So wäre schnell die Grenze erreicht. Allenfalls lokale Methoden innerhalb von Methoden wären dann noch nutzbar. Die Struktur wird durch die klare Trennung zwischen *Integration* und *Operation* gebildet und ist somit unendlich fortführbar.

Übernehmen der Ein- und Ausgänge

Wichtig bei jeder Verfeinerung ist, dass die Ein- und Ausgänge syntaktisch passen müssen. Eine Verfeinerung ist syntaktisch nur dann korrekt, wenn alle Ein- und Ausgänge der übergeordneten Funktionseinheit exakt auch in der Verfeinerung auftauchen.

Günstig ist es daher, vor allem zu Beginn im Umgang mit Flow Design, zunächst die Ein- und Ausgänge in die Verfeinerung zu übernehmen. Erst danach beginnt man damit, die Verfeinerung durch Funktionseinheiten auszufüllen. Mit steigender Erfahrung kann man darauf

verzichten, sofort den oder die Ausgänge einzumecken. Dennoch muss sichergestellt sein, dass am Ende syntaktisch alles zusammenpasst.

Klassennamen

Das derzeit am weitesten verbreitete Paradigma ist die Objektorientierung. Sprachen wie Java, C#, C++ und viele weitere sind objektorientierte Sprachen. Diese Sprachen haben gemeinsam, dass Methoden in der Regel in Klassen abgelegt werden müssen.

In Sprachen, die dies nicht zwingend erforderlich machen, müssen die Methoden allerdings mindestens in Dateien abgelegt werden. Insofern stellt sich in jedem Fall die Frage, wo nun die einzelnen Methoden eines Entwurfs abgelegt werden.

Klassen bilden

Bevor ein Entwurf in einer objektorientierten Sprache implementiert werden kann, müssen Klassennamen festgelegt werden. In Sprachen, die keine Klasse benötigen, muss dennoch festgelegt werden, in welchen Dateien die einzelnen Methoden abgelegt werden.

Die einzelnen Funktionseinheiten eines Entwurfs verfügen bereits über eine Bezeichnung und werden in der Regel in Methoden übersetzt. Diese Methoden können allerdings häufig nicht frei stehen, sondern müssen in Klassen angelegt werden. Daher besteht die Aufgabe aus den beiden folgenden Teilen:

- Es muss entschieden werden, welche Funktionseinheiten zusammengefasst werden bzw. welche getrennt abgelegt werden sollen.
- Für die so entstehenden Zusammenfassungen, implementiert als Klassen oder physisch gespeichert in Dateien, müssen Bezeichner gefunden werden.

Ob Methoden gemeinsam oder getrennt in Klasse abgelegt werden, richtet sich nach ihrem Aspekt. Methoden, die denselben Aspekt betreffen, werden gemeinsam in der selben Klasse abgelegt, weil sie eine hohe Kohäsion haben. Methoden unterschiedlicher Aspekte werden getrennt.

Zur Notation wird der Klassename im Entwurf unter die Funktionseinheit geschrieben. Dazu sollte möglichst eine andere Farbe verwendet werden, um den Entwurf auf diese Weise übersichtlicher zu gestalten.

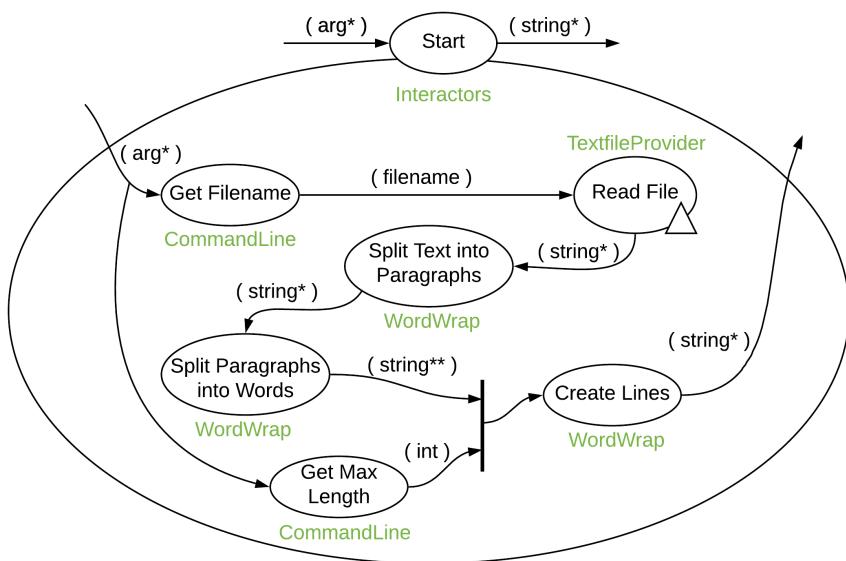


Abbildung 190: Klassennamen an Funktionseinheiten

Funktionseinheiten zusammenfassen

Um Funktionseinheiten in einem Entwurf zu einer Klasse zusammenzufassen, kann man sie gemeinsam umrahmen. Auch hier hilft es, wenn man unterschiedliche Farben verwendet.

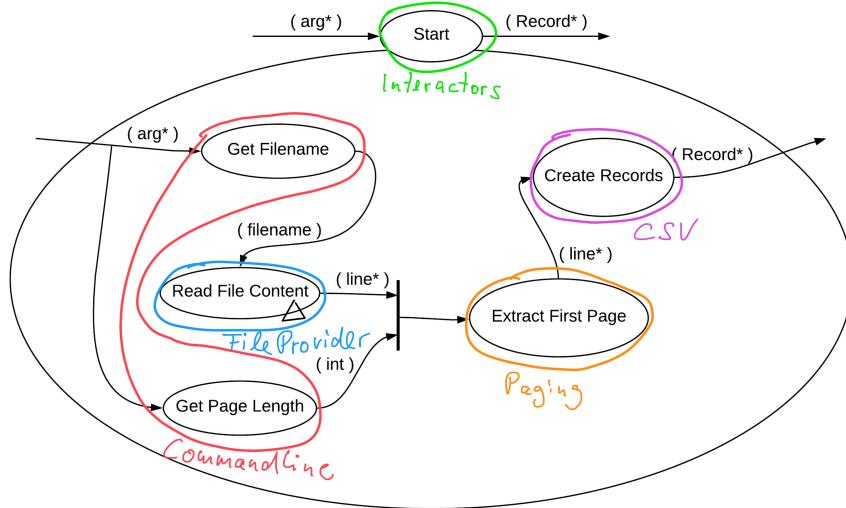


Abbildung 191: Funktionseinheiten umrahmt

Wo ein Umrahmen nicht möglich ist, weil die Funktionseinheiten zu weit auseinander liegen bzw. sich die Umrahmungen überlappen würden, werden lediglich Klassennamen an die Funktionseinheit geschrieben. Durch die Wiederholung von Klassennamen ergibt sich dann die Zugehörigkeit mehrerer Funktionseinheiten zur selben Klasse.

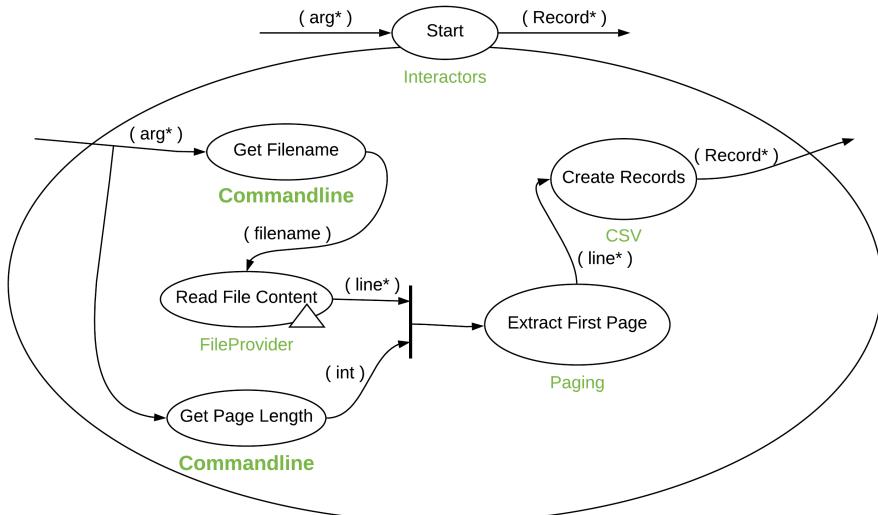


Abbildung 192: Mehrere Funktionseinheiten mit dem selben Klassennamen

Nebenläufigkeit

Flow Design Diagramm sind zunächst grundsätzlich synchron sequentiell. Zu einem Zeitpunkt wird genau eine Funktionseinheit ausgeführt. Technisch ausgedrückt findet die gesamte Ausführung auf einem einzigen Thread statt. Werden mehrere Threads verwendet, müssen die Datenflüsse und Funktionseinheit eingefärbt werden, um die Threadzugehörigkeit auszudrücken.

Threads einfärben

Es handelt sich bei Flow Design um Datenflussdiagramme, nicht um Kontrollflussdiagramme. Aufgrund der synchron sequentiellen Ausführung folgt der Kontrollfluss allerdings dem Datenfluss. Fließt ein y von A nach B , so wandert auch die Kontrolle von der Funktionseinheit A zu B .

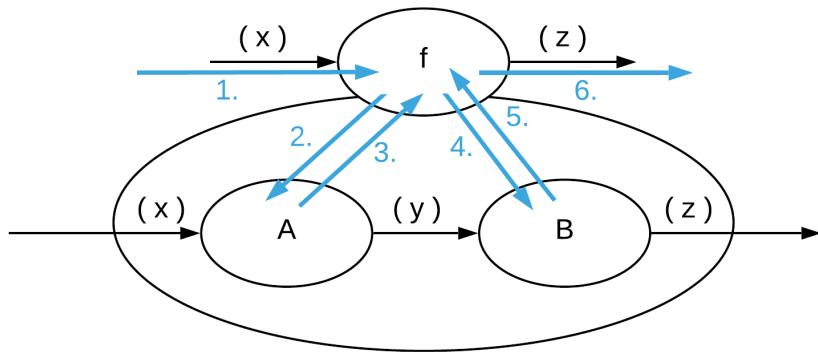


Abbildung 193: Kontrollfluss vs. Datenfluss

Zuerst wird f ausgeführt, welches für die Integration von A und B verantwortlich ist. Während dieser Zeit werden weder A noch B ausgeführt. Als erstes übergibt f das x an A . Damit geht auch die Kontrolle von f an A . Sobald A sein y produziert hat, wird dieses y von der Integration f an B übergeben. Für einen kurzen Moment hat A die Kontrolle an f zurückgegeben. Die Integration f ruft nun B auf. Dadurch hat B die Kontrolle und f ist angehalten. Zuletzt gibt auch B seine Kontrolle wieder ab, wodurch wieder f an der Reihe ist.

Wird eine Funktionseinheit auf einem eigenen Lebensfaden (Thread) ausgeführt, verlaufen Daten- und Kontrollfluss nicht mehr zwingend gleich.

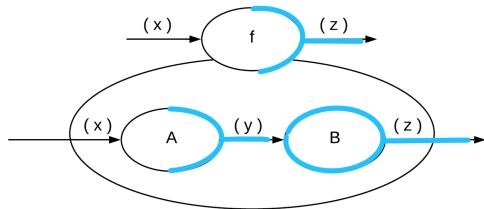


Abbildung 194: Nebenläufige Funktionseinheiten

In diesem Beispiel erhält f zunächst die Kontrolle. Es übergibt die Kontrolle an A . A verlagert nun einen Teil seiner Tätigkeit auf einen weiteren Thread. Auf diesem zusätzlichen Thread wird anschließend das y zu B transportiert und B ausgeführt. Gleichzeitig fällt allerdings die Kontrolle von A zurück an f , denn ‘‘der Rest’’ von A läuft auf einem anderen Thread als B . Um die unterschiedlichen Threads zu visualisieren, werden die Funktionseinheiten sowie die Datenflüsse eingefärbt. Die Ausführung von A beginnt auf dem Hauptthread, hier nicht weiter eingefärbt und somit in schwarz dargestellt. Da A nun einen weiteren Thread startet, findet ein Teil der Ausführung von A auf einem anderen Thread statt. Daher wird A zur Hälfte in einer anderen Farbe eingefärbt, hier blau. Da der Datenfluss von A zu B und auch die Ausführung von B nun auf dem blauen Thread stattfinden, sind B und der Datenfluss in blau eingefärbt. Daraus ergibt sich dann natürlich, dass f ebenfalls auf zwei unterschiedlichen Threads ausgeführt wird. Die Ausführung von f beginnt auf dem schwarzen Thread. Es fällt dann die Kontrolle zurück an den Aufrufer von f . Gleichzeitig läuft f auf dem blauen Thread weiter und produziert irgendwann den Wert z .

Zum besseren Verständnis ist im folgenden die Implementation zu diesem Beispiel abgebildet.

```
public void f(int x, Action<int> onResult) {
    Console.WriteLine("Calling A...");
    A(x, onResult);
    Console.WriteLine("A finished");
}
```

```
private void A(int x, Action<int> onResult) {
    var y = x + 42;

    Console.WriteLine("Calling B on new thread...");
    var thread = new Thread(() => {
        var z = B(y);
        onResult(z);
    });
    thread.Start();
    Console.WriteLine("B finished");
}
```

```
private int B(int y) {
    Console.WriteLine("B is working...");

    // Simulate long running calculation
    Thread.Sleep(1000);

    Console.WriteLine("B did its work");
    return y + 1;
}
```

Da das Ergebnis von f auf einem anderen als dem Hauptthread produziert wird, kann f das Resultat nicht als Rückgabewert der Methode zurückgeben. Schließlich endet die Ausführung von f auf dem Hauptthread und die Kontrolle fällt an den Aufrufer zurück. Aus diesem Grund muss das Resultat auf anderem Weg geliefert werden, sobald der Hintergrundthread das Ergebnis hergestellt hat. Im Listing ist eine *Continuation* zur Implementation gewählt. Der Funktionszeiger `onResult` wird auf dem Hintergrundthread aufgerufen, sobald das Ergebnis vorliegt.

Vorgehensweise

Flow Design ist eine Entwurfsmethode. Die Methode kann allerdings nur Wirkung entfalten, wenn sie tatsächlich angewandt wird. Das ist trivial und trotzdem wesentlich. Denn Flow Design ist nicht die einzige Entwurfsmethode. Wir beobachten „in der freien Wildbahn“, dass dort eher garnicht entworfen wird. Teams springen in den meisten Fällen mehr oder weniger direkt von den Anforderungen zum Code. Offensichtlich genügt es also nicht, dass die Informatik eine Reihe von Entwurfsmethoden hervorgebracht hat. Vielleicht sind einige davon zu kompliziert in ihrer Anwendung und deshalb in der täglichen Praxis nicht anzutreffen. Flow Design ist aus diesem Grund ganz absichtlich sehr einfach gehalten, kommt mit wenigen Symbolen aus.

Eine weitere Voraussetzung für die Anwendbarkeit einer Entwurfsmethode ist allerdings, dass sie eingebettet sein muss, in eine konsequente Vorgehensweise. Zwischen den Anforderungen und dem Code liegen einige wesentliche Schritte. In den folgenden Abschnitten wird dargestellt, wie Entwickler oder Entwicklerteams im Sinne eines Entwicklungsprozesses vorgehen können. Es wird aufgezeigt, welche Arbeitsschritte jeweils zwischen den Anforderungen und dem Code liegen und durchgeführt werden müssen. Der Entwurf einer Lösung für die funktionalen Anforderungen mittels Flow Design ist einer dieser Schritte. Es bedarf jedoch weiterer.

Darstellung im Softwareuniversum

Zur Abgrenzung, welche Bereiche des Softwareentwicklungsprozess im folgenden näher beschrieben werden, soll ein Blick in das Softwareuniversum dienen. Es stellt in vier Dimensionen die unterschiedlichen Bereiche dar, für die jeder Softwareentwicklungsprozess Lösungen bieten sollte.

- Die Dimension der *Domänenzerlegung* dient dazu, die Anforderungen so zu zerlegen, dass im Sinne der Agilität regelmäßig und kurzfristig Feedback eingeholt werden kann.
- In der Dimension der *Module* geht es um die Frage, wie die Funktionalität strukturiert werden kann, um eine hohe Wandelbarkeit zu erreichen.
- Bei der *Hosts* Dimension werden die nicht-funktionalen Anforderungen betrachtet.
- Last but not least geht es in der *Flow Design* Dimension um den Entwurf der funktionalen Anforderungen.

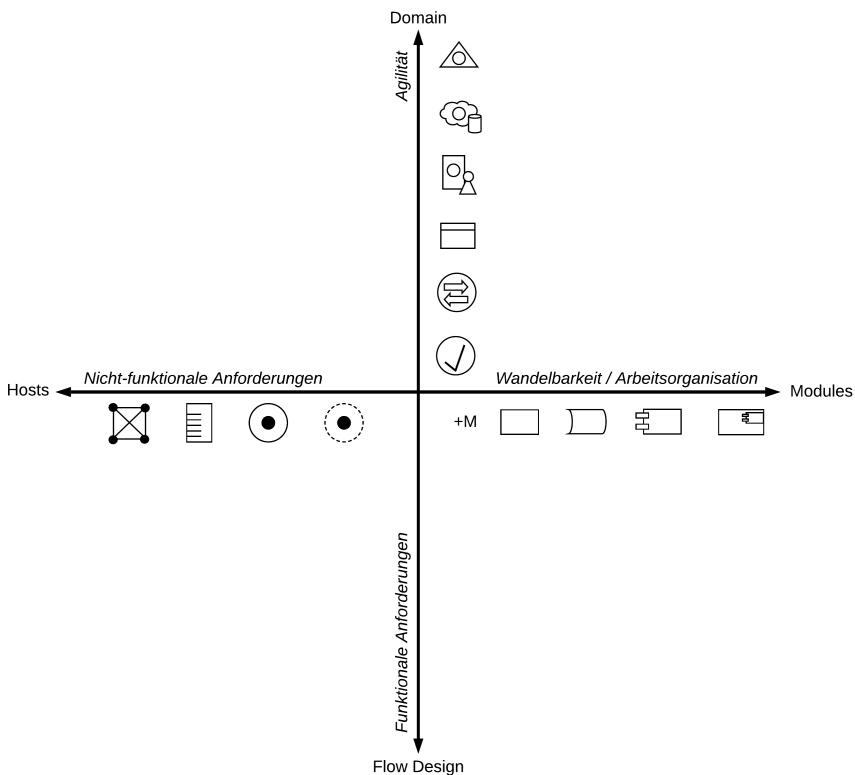


Abbildung 195: Das Softwareuniversum

In den folgenden Abschnitten wird der Softwareentwicklungsprozess für ein Team von Entwicklern dargestellt. Im Softwareuniversum ist erkennbar, welche Themen hier dargestellt werden. Bei der Zerlegung der Anforderungen werden *Dialoge*, *Interaktionen* und *Features* dargestellt. Die darüber liegenden Bereiche *App* und *Bounded Context* betreffen die Grobstruktur großer Softwaresysteme und werden hier nicht weiter vertieft.

Im Bereich der Module werden *Methoden*, *Klassen* und *Bibliotheken* dargestellt. *Komponenten* und *Services* werden nicht behandelt.

Die *Hosts* Zerlegung findet lediglich im Bereich mehrerer *Threads* statt. Eine Vertiefung in Richtung *Prozess*, *Maschine* und *Netzwerksite* liegt außerhalb des Themas dieses Buchs.

Der untere Bereich *Flow Design*, also der Entwurf einer Lösung für die funktionalen Anforderungen, ist das Kernthema dieses Buchs.

Von den Anforderungen zum Code

Die Aufgabe eines Softwareentwicklungsprozesses besteht darin zu definieren, wie ein Team von Entwicklern von den Anforderungen zum Code kommt. Die Anforderungen stellen sozusagen den Input in den Prozess dar, während der Code der Output des Prozesses ist. Es dürfte klar sein, dass nur bei sehr überschaubaren Anforderungen die Vorgehensweise darin besteht, unmittelbar Code für die Anforderungen zu schreiben. Selbst bei einer solchen Vorgehensweise finden die beiden Schritte Entwurf einer Lösung und Umsetzung der Lösung statt, wenn auch zeitlich eng verzahnt. Der Entwickler denkt ein wenig über die Lösung nach, codiert ein wenig, denkt nach, codiert, etc. Eine solche Vorgehensweise ist allerdings nicht dazu geeignet, die üblichen Anforderungen realer Softwareentwicklungsprojekte in Code zu übersetzen. Es braucht dazu die im folgenden dargestellten Schritte. Zusätzlich aufgeführt sind die Diagrammarten bzw. Methoden, die beim jeweiligen Schritt verwendet werden.

Anforderungen

- Anforderungen zerlegen (*System-Umwelt-Diagramm, Interaktionsdiagramm*)
- Entwurf in die Breite (*Flow Design*)
- Auswahl einer Anforderung
- Entwurf in die Tiefe (*Flow Design*)
- Arbeitsorganisation (*Komponentenorientierung, Contract-first*)
- Implementation inkl. Tests
- Integration inkl. Tests
- Code Review

Code

In den folgenden Abschnitten werden die einzelnen Schritte näher beschrieben.

Anforderungen

Dialoge und Interaktionen identifizieren

Bevor mit dem Entwurf einer Lösung begonnen werden kann, muss ein ausreichend kleiner Ausschnitt der Anforderungen identifiziert werden. Dazu dienen *Dialoge* und *Interaktionen*. Jede Iteration des Entwicklerteams beginnt damit, dass das Team gemeinsam mit dem Product Owner einen Bereich des Softwaresystems auswählt und über diesen konkret in Form eines Dialogs mit seinen Interaktionen spricht. In manchen Fällen ist mehr als ein Dialog betroffen. Typischerweise sollten ein oder maximal zwei Dialoge relevant sein. Der Umfang einer Iteration soll im Bereich von ein bis zwei Tagen liegen. In so kurzen Iterationen können nur selten mehr als zwei Dialoge betrachtet werden.

Innerhalb eines Dialogs werden die *Interaktionen* identifiziert. Die Zusammenstellung der Interaktionen des Dialogs sollte vollständig erfolgen. Nur so ist gewährleistet, dass beim ersten Entwurf in die Breite alle Datenflüsse des Dialogs betrachtet werden können.

Ergebnis der Betrachtung von Dialogen und Interaktionen ist ein *Interaktionsdiagramm*.

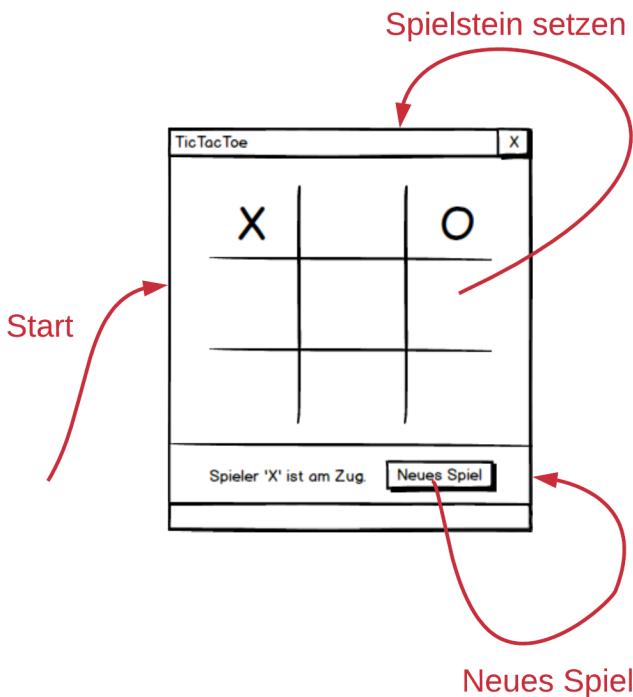


Abbildung 196: Interaktionsdiagramm für ein TicTacToe Spiel

In einem Interaktionsdiagramm wird dargestellt, welche Möglichkeiten der Anwender hat, mit der Anwendung zu interagieren. Jede Interaktion ist ein Grund dafür, dass wir als Entwickler Domänenlogik realisieren müssen. Ohne dass der Anwender mit der Anwendung interagiert, bliebe die Anwendung völlig passiv.

Aus Sicht der Anforderungen ist diese Erkenntnis sehr wesentlich. Daraus ergibt sich nämlich die Tatsache, dass jede einzelne Anforderung, die der Product Owner formuliert, einer Interaktion zugeordnet werden kann. Ohne Interaktion kein Verhalten. Damit bieten die Interaktionen einen sehr guten Einstiegspunkt in die Zerlegung der Anforderungen.

Die klare Struktur aus Dialogen und Interaktionen zwingt Product Owner und Team dazu, strukturiert über die Anforderungen zu sprechen. Jeder Wunsch des Product Owners muss einer Interaktion zugeordnet werden können. Anders kann das Verhalten der Anwendung nicht ausgelöst werden. Diese strukturierte Vorgehensweise ist für Teams ein großer Gewinn. Statt über User Stories zu brüten und sich zu fragen, wie diese kleiner geschnitten werden können, bieten Dialoge und Interaktionen einen

visuellen Weg. User Stories haben an anderer Stelle ihren Nutzen. Im Bereich der Produktentwicklung kann mit ihnen das gewünschte Verhalten skizziert werden, lange bevor die Anforderungen so weit konkretisiert sind, dass bereits über Dialoge nachgedacht wird. Für den Einstieg in den Entwurf lassen User Stories jedoch zu viel Spielraum. Hier ist eine strukturiertere Vorgehensweise erforderlich, wie sie durch Dialoge und Interaktionen geliefert wird.

Vier Arten von Interaktionen

In einem Interaktionsdiagramm gibt es vier verschiedene Arten von Interaktionen.

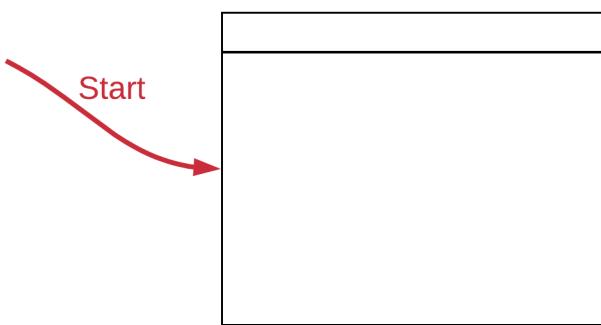


Abbildung 197: Interaktion beim *Start* der Anwendung

Beim Start der Anwendung findet die Interaktion des Benutzers streng genommen noch nicht mit der Anwendung statt. Der Anwender interagiert eigentlich mit dem Betriebssystem, welches dafür sorgt, dass die Anwendung gestartet wird. Der Pfeil, der die Interaktion repräsentiert, startet daher "aus dem nichts" und endet im Dialog. Die *Start* Interaktion ist immer dann relevant, wenn beim Starten der Anwendung bereits Domänenlogik ausgeführt werden muss. Genügt es, einen Dialog im rein technischen Sinne zu öffnen und darzustellen, ist dies keine relevante Interaktion. Meist müssen im initialen Dialog jedoch bereits Daten dargestellt werden. Für die Bereitstellung der Daten ist Domänenlogik erforderlich, da die Daten aus irgendeiner Quelle besorgt werden müssen. Die reine Darstellung im Dialog ist UI Logik. Doch die Frage, welche Daten besorgt werden müssen, wo sie herkommen, etc. ist eine Domänenentscheidung.

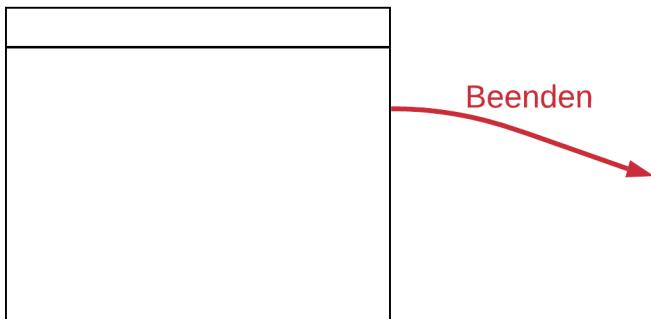


Abbildung 198: Interaktion beim Beenden Anwendung

Das Beenden einer Anwendung ist in der Regel ein rein technischer Vorgang. Bei einer Desktop Anwendung klickt der Anwender dazu die Schließen Schaltfläche des Dialogs an. Im technischen Sinne ist Logik erforderlich, um die Anwendung zu beenden. Doch dies ist keine Domänenlogik. Jede Anwendung wird auf dieselbe Art und Weise beendet. In einigen Fällen mag es jedoch erforderlich sein, beim Beenden der Anwendung eine Domänenentscheidung zu treffen. Soll bspw. die Anwendung beim erneuten Start dort fortgesetzt werden, wo sie gerade verlassen wird, muss dieser Punkt im Sinne der Domäne gespeichert werden. In einem solchen Fall stellt das Verlassen der Anwendung eine Interaktion dar, die im Sinne der Anforderungen relevant ist.

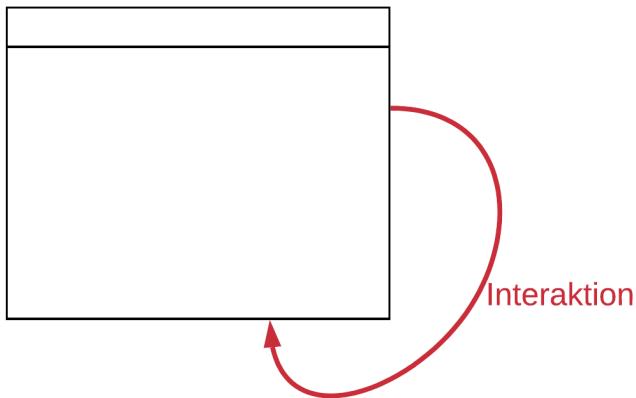


Abbildung 199: Interaktion innerhalb eines Dialoges

Die am häufigsten anzutreffende Form einer Interaktion beginnt in einem Dialog und endet im selben Dialog. Im TicTacToe Beispiel ist dies bei der Interaktion *Spielstein setzen* der Fall. Innerhalb des Dialogs klickt der Anwender hier auf ein Spielfeld, um dort seinen Spielstein zu setzen. Nach Ausführung der Domänenlogik, die für diese Interaktion verantwortlich ist, geht die Kontrolle wieder zurück zum selben Dialog. Daher startet und endet der Interaktionspfeil hier im selben Dialog.

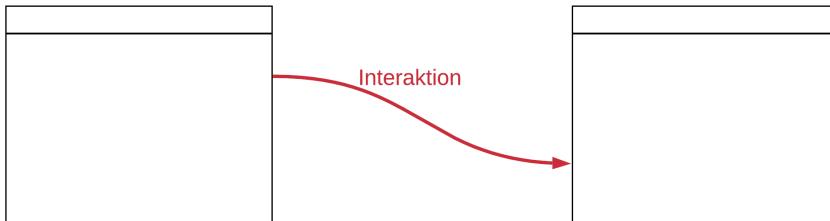


Abbildung 200: Interaktion, die in einen anderen Dialog führt

Eine Interaktion kann auch zu einem anderen Dialog führen. Im Beispiel der Fragebogen Anwendung *Questionnaire* führt die Schaltfläche *Auswerten* vom Fragebogen zur Auswertung. Hier sind zwei Dialoge an der Interaktion beteiligt. Die Interaktion des Anwenders führt hier dazu, dass Daten an einen anderen Dialog geliefert werden. Handelt es sich um einen modalen Dialog, fließt mit den Daten auch die Kontrolle in den anderen Dialog. Bei nicht-modalen Dialogen, von denen mehrere autonom gleichzeitig ausgeführt werden, fließen lediglich Daten von einem zum anderen Dialog. Die Kontrolle muss nicht zwangsläufig wechseln.

In einer typischen Master-Detail Darstellung führt die Auswahl eines Datensatzes im Master Dialog dazu, dass im Detail Dialog eine größere Menge Daten zum selektierten Datensatz dargestellt wird. Die Interaktion im Master Dialog führt also dazu, dass Domänenlogik ausgeführt wird. Diese Domänenlogik sorgt wiederum dafür, dass die benötigten Details ermittelt und an den Detail Dialog übertragen werden.

Features

Die Identifizierung der möglichen Interaktionen in einem Dialog soll immer vollständig sein. Das bedeutet, dass das Team zusammen mit dem Product Owner herausarbeiten muss, welche Möglichkeiten der Interaktion der Anwender hat. Nimmt man alle Interaktionen eines Dialogs zusammen, sind darin die gesamten Anforderungen enthalten, die für diesen Dialog relevant sind.

In vielen Fällen ist eine einzelne Interaktion allerdings noch zu groß, um die dafür erforderliche Domänenlogik innerhalb von ein bis zwei Tagen zu realisieren. In dem Fall ist es erforderlich, innerhalb der Interaktion *Features* zu identifizieren, die zunächst weggelassen werden. Bei der Zusammenstellung der Features einer Interaktion geht es allerdings nicht darum, eine Auflistung zu erstellen, die in ihrer Summe vollständig alle Anforderungen der Interaktion enthält. Es geht lediglich darum, Teile der Anforderungen zu finden, die zunächst in der nächsten Iteration nicht berücksichtigt werden.

Ein typisches Beispiel stellt die Validierung der Benutzereingaben dar. In der ersten Iteration kann die Domänenlogik einer Interaktion vielleicht in zwei Tagen realisiert werden, allerdings nur, wenn auf die Validierung der Eingaben verzichtet wird. Damit stellt die Validierung ein Feature der Interaktion dar. Am Ende muss natürlich auch dieses Feature realisiert werden, bevor das Produkt ausgeliefert werden kann. Im Sinne eines agilen Vorgehens ist es jedoch wichtig, kurzfristig Feedback vom Product Owner zu erhalten. Für das Team ist es daher besser, zuerst Feedback zu erhalten zur Domänenlogik, die bei korrekter Eingabe abläuft. Der Spezialfall, dass die Domänenlogik auch bei inkorrekt Eingabe funktioniert bzw. nicht abläuft und stattdessen eine Meldung an den Anwender ausgegeben wird, ist hier zweitrangig.

Ein weiteres typisches Beispiel für ein Feature sind die Spezialfälle. Auch hier kann zunächst in den ersten Iterationen der Normalfall betrachtet werden. Erst wenn dieser gelöst ist und der Product Owner diesen Fall abgenommen hat, sollten die Spezialfälle betrachtet werden. Auch auf diese Weise wird es möglich, bereits nach zwei Tagen Feedback zu erhalten. Statt also sofort die gesamte Domänenlogik einer Interaktion auf einmal zu betrachten, werden Features bewusst zeitlich verschoben, um kurzfristig Feedback zu erhalten.

Entwurf

Nachdem mit dem Product Owner die Dialoge und Interaktion identifiziert wurden, hat das Team die Aufgabe, den Entwurf einer Lösung zu erstellen.

Entwurf in die Breite

Wenn alle Interaktionen des Dialogs identifiziert sind, wird die oberste Ebene des Entwurfs erstellt. Dies stellt den Übergang von den Anforderungen zum Entwurf dar. Ziel des Entwurfs in die Breite ist es, die Daten zu entwerfen, die bei den Interaktionen fließen. Ferner wird an dieser Stelle entschieden, wo der Zustand der Session gehalten wird.

Der Übergang vom Interaktionsdiagramm zum Entwurf ist zunächst simpel: jede Interaktion wird zu einem Datenfluss. Die folgenden Abbildungen zeigen dies für die vier möglichen Interaktionen.

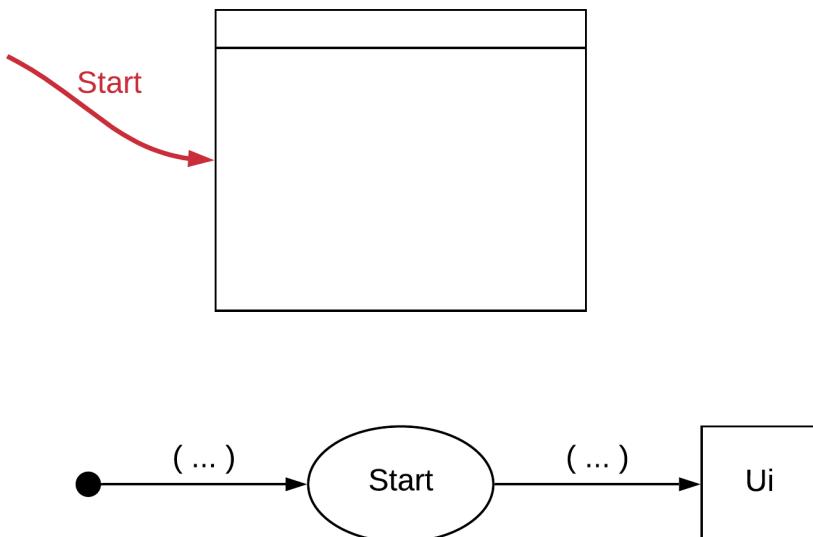


Abbildung 201: Start der Anwendung

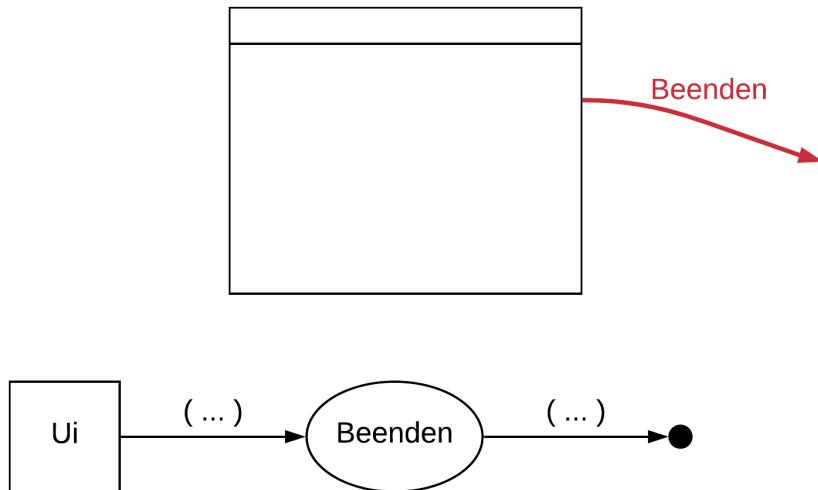


Abbildung 202: Beenden der Anwendung

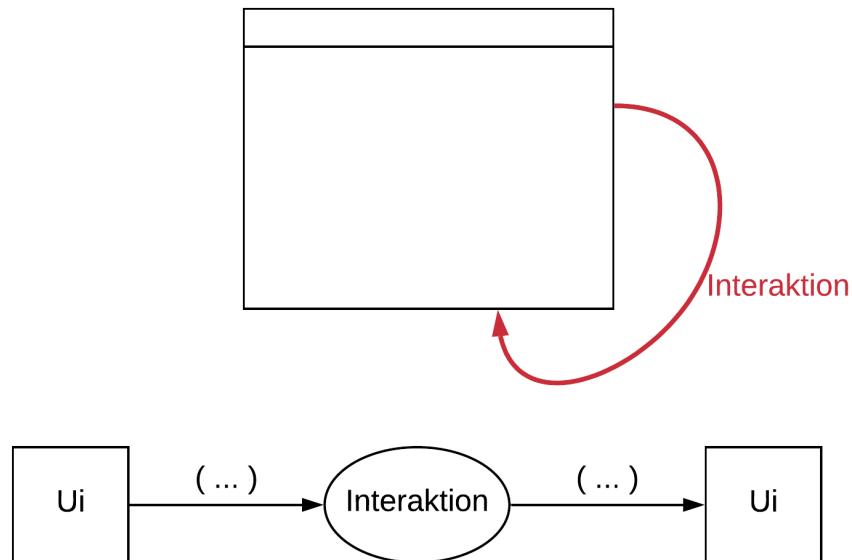


Abbildung 203: Interaktion innerhalb eines Dialogs

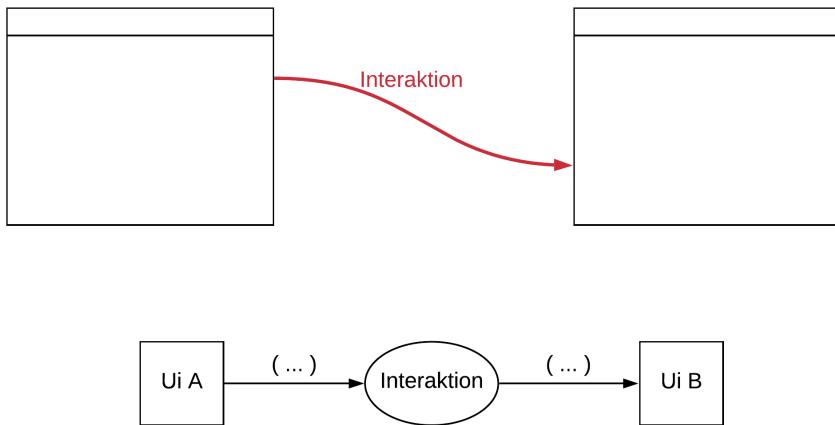


Abbildung 204: Interaktion zwischen zwei Dialogen

Die Struktur der Datenflüsse ergibt sich unmittelbar aus den Interaktionsspitzen. Der Name der Interaktion wird zum Namen der Funktionseinheit. Die Interaktion wird zu einem Datenfluss. Dieser beginnt und endet entweder beim Betriebssystem oder in einem Dialog.

Die Herausforderung beim Entwurf besteht nun darin zu entscheiden, welche Daten auf den einzelnen Datenflüssen fließen sollen. Die folgende Abbildung zeigt dies erneut am Beispiel des TicTacToe Spiels.

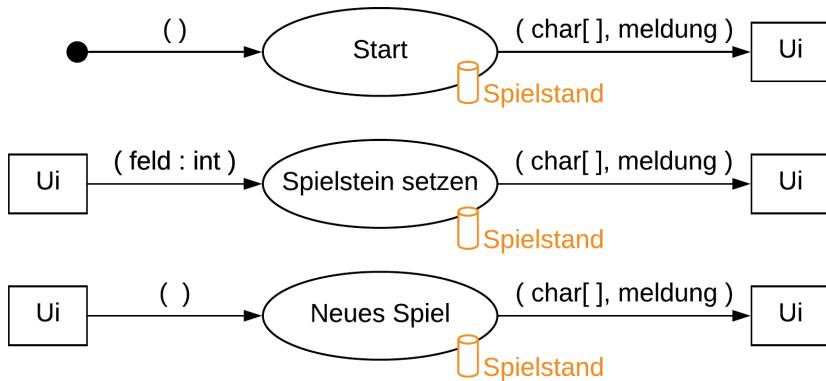


Abbildung 205: Entwurf der obersten Ebene des TicTacToe Spiels

Hier wurde neben den Daten auch entschieden, dass der Zustand der Anwendung in der Domänenlogik gehalten werden soll, erkennbar an den Zustandstonnen an den Interaktoren.

Der Entwurf der obersten Ebene dient einerseits dazu, einen einfachen Übergang von den Anforderungen zum Entwurf zu ermöglichen. Es ergibt sich hieraus ein weiterer Vorteil: jede einzelne Interaktion spiegelt sich in einer Funktionseinheit wider. Auf diese Weise ist eine optimale Navigation im Quellcode gegeben. Ausgehend von den Interaktionsdiagrammen kann zu jeder einzelnen Interaktion jeweils eine Methode innerhalb der Codebasis gefunden werden. So ist der Einstieg in den zur Anforderung gehörenden Code erleichtert.

Es gibt noch einen weiteren Grund, den Entwurf zunächst in die Breite für alle Interaktionen auf einmal vorzunehmen: manchmal gibt es Abhängigkeiten zwischen den Interaktionen. Werden bspw. in einer Interaktion Daten selektiert, auf die sich dann eine weitere Interaktion bezieht, sollte diese Abhängigkeit näher betrachtet werden. Hier ist die Frage relevant, wie sich die logische Abhängigkeit auf die fließenden Daten auswirkt. Soll eine Interaktion eine *Id* für den selektierten Datensatz liefern, müssen diese *Ids* zuvor an den Dialog liefert werden sein. Würde man hier nur die Start Interaktion betrachten, würde man keine *Id* einführen, da sie rein zur Anzeige der Daten nicht benötigt wird.

Eine weitere Aufgabe beim Entwurf in die Breite besteht darin, sich über den Zustand Gedanken zu machen. Insbesondere die Entscheidung, ob der Zustand in der UI oder der Domänenlogik liegen soll, muss getroffen werden.

Auswahl einer Interaktion

Spätestens nach dem Entwurf in die Breite muss der Product Owner eine Interaktion auswählen, die im folgenden in der Tiefe entworfen und anschließend realisiert wird. Dabei orientiert sich der Product Owner am größten Nutzen. Er wählt die Interaktion, die ihm bzw. den realen Kunden den größten wirtschaftlichen Nutzen bringt. Im Hinterkopf sollte der Gedanke mitschwingen, was passiert, wenn das Projekt plötzlich gestoppt wird. Es sollte daher jeweils die Interaktion als nächste realisiert werden, die unter diesem theoretischen vorzeitigen Ende des Projekts unbedingt noch realisiert werden sollte.

In späteren Phasen entscheidet der Product Owner sich ggf. auch für ein Feature. Wurden bei der Realisierung einer Interaktion Features zeitlich verschoben, müssen diese irgendwann einmal an die Reihe kommen.

Das Entwicklerteam hat bei der Auswahl insofern ein Mitspracherecht, dass es Interaktionen favorisieren sollte, die ein hohes technisches Risiko tragen. Das Risiko sollte so früh wie möglich ausgeräumt werden. Daher kann es aus dieser Betrachtung heraus sinnvoll sein, eine Interaktion früh zu realisieren, obschon sie für den Product Owner nicht die höchste Priorität hat.

Grundsätzlich droht ein Projekt an technischen Risiken zu scheitern. Daher sollten diese so früh wie möglich angegangen werden.

Der Product Owner muss die zu realisierenden Interaktionen übrigens nicht vollständig in eine Reihenfolge bringen. Es genügt, wenn er zwei bis drei Interaktionen oder Features benennt, die als nächste realisiert werden sollen. Zentral ist dabei, die eine zu benennen, die ihm am wichtigsten ist. Ein bis zwei weitere als Ersatz zu benennen ist sinnvoll, falls es bei der ursprünglich gewählten, aus welchen Gründen auch immer, nicht weitergehen kann. Manchmal übersieht das Team ein Detail, das es daran hindert, die Interaktion sofort zu realisieren. Oder es tauchen Rückfragen auf, die nicht sofort vom Product Owner beantwortet werden können. Für diese Fälle ist es sinnvoll, ein oder zwei Ersatzinteraktionen zu benennen, die dann stattdessen realisiert werden. Das Team sollte allerdings möglichst gemeinsam an nur einer einzigen Interaktion arbeiten. Multitasking ist hier nicht förderlich. Es geht nichts schneller voran, dadurch, dass mehrere Interaktionen gleichzeitig in einem Team realisiert werden. Stattdessen steigt die Wahrscheinlichkeit, dass systematische Probleme sich auf mehr als eine "Baustelle" auswirken.

Entwurf in die Tiefe, Verfeinerung

Nachdem eine Interaktion ausgewählt wurde, muss deren Entwurf so weit in die Tiefe verfeinert werden, dass sie im Anschluss flüssig kodiert werden kann. Das bedeutet einerseits, dass alle Bereiche soweit durchdacht werden, dass jeweils eine umsetzbare Lösung entworfen wird. Es sollte während der Kodierung nicht notwendig sein, tiefer über die Lösung nachzudenken. Des Weiteren muss der Entwurf so detailliert sein, dass alle Aspekte klar getrennt sind. Nur so wird die Wandelbarkeit erreicht.

Gleichwohl bleiben dann und wann innerhalb einer Funktionseinheit Herausforderungen bestehen, über die ein Entwickler während des Kodierens nachdenken muss. Es geht dann allerdings lediglich darum, eine Lösung auf der Ebene eines Algorithmus zu finden. Die Einbettung der Funktionseinheit in die gesamte Lösung für das Inkrement ist bereits entworfen und durchdacht. Ferner ist jede einzelne zu realisierende Funktionseinheit bereits auf einen Aspekt reduziert.

Die Verfeinerung des Entwurfs beginnt bei der Funktionseinheit, welche die gesamte Interaktion repräsentiert. Als Beispiel zeigt die folgende Abbildung den Entwurf der Interaktion *Spielstein setzen* des TicTacToe Spiels.

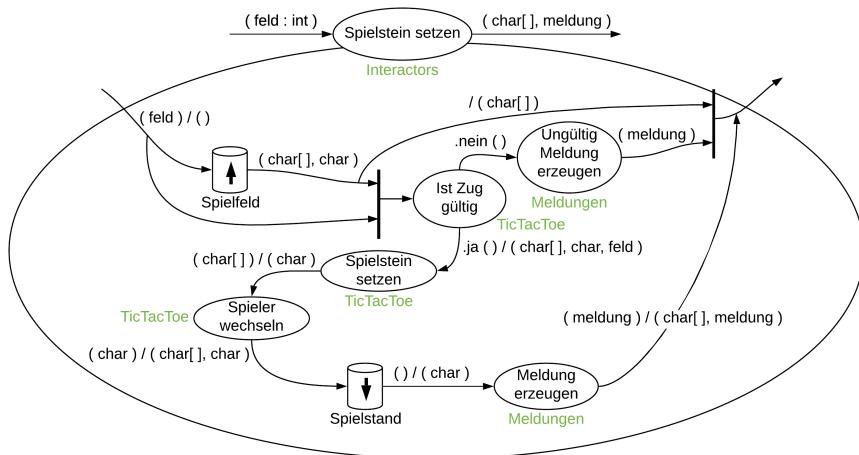


Abbildung 206: Verfeinerung der Interaktion *Spielstein setzen*

Zum Abschluss des Entwurfs in die Tiefe muss sich das Team zu jeder einzelnen Funktionseinheit die beiden folgenden Fragen stellen:

- Ist die Funktionseinheit für genau einen Aspekt zuständig?
- Kann die Funktionseinheit in maximal vier Stunden durch einen einzelnen Entwickler inkl. Tests implementiert werden?

Die erste Frage nach der Trennung der Aspekte ist relevant, um das *Single Responsibility Principle* (SRP) einzuhalten. Werden Aspekte vermischt, ist die Wandelbarkeit in Gefahr.

Die Frage nach der Implementationsdauer ist relevant um sicherzustellen, dass das gesamte Inkrement innerhalb von maximal zwei Tagen hergestellt ist. Wenn für eine einzelne Funktionseinheit länger als vier Stunden für die Implementation benötigt wird, droht das Risiko, dass die zwei Tage überschritten werden. Dabei geht es hier nicht um eine exakte Abschätzung des Aufwands. Es geht vielmehr darum, die Funktionseinheiten zu identifizieren, die möglicherweise zu groß sind. Diese sollten dann nochmals verfeinert werden, um so herauszufinden, ob die Funktionseinheit tatsächlich so umfangreich ist, wie vermutet. Wenn das der Fall ist, können möglicherweise Features weggelassen werden. Stellt man bei der Verfeinerung fest, dass die Funktionseinheit doch nicht so umfangreich ist, wie angenommen, ist das nicht weiter schlimm. Wichtig ist hier ein sorgfältiger Umgang mit dem Entwurf um sicherzustellen, dass die nachfolgende Implementation flüssig von der Hand geht. Die Zeit ist gut in den Entwurf investiert.

Übersetzung in Methode oder Klasse

Beim Übergang vom Entwurf zur Umsetzung muss zu jeder Funktionseinheit entschieden werden, ob sie als Methode oder Klasse realisiert wird. Die meisten Funktionseinheiten werden zu Methoden. Typischerweise wird die UI als Klasse realisiert.

Klassennamen finden, Funktionseinheiten zuordnen

Im Anschluss an die Entscheidung, ob Funktionseinheiten als Methoden oder Klassen umgesetzt werden, müssen die Klassennamen vergeben werden. Dazu ist vor allem bei den Methoden darauf zu achten, dass eine gute Zuordnung von Methoden zu Klassen erfolgt. Einerseits sollen keine Aspekte vermischt werden (niedrige Kohäsion), andererseits sollen zusammengehörige Methoden (hohe Kohäsion) in der selben Klasse implementiert werden.

Arbeitsorganisation

Ein Entwurf bietet die Möglichkeit, Software arbeitsteilig zu implementieren. Ohne Entwurf ist eine arbeitsteilige Realisierung nicht sinnvoll möglich. Das liegt daran, dass die Strukturen und Schnittstellen erst durch den Entwurf so weit detailliert werden, dass sinnvoll mit mehreren Entwicklern gleichzeitig an der selben Interaktion gearbeitet werden kann. Ohne Entwurf können nur sehr grobe Einheiten gleichzeitig realisiert werden. Sobald es zu Schnittstellen zwischen den Einheiten kommt, müssen diese detailliert durchdacht werden. Andernfalls passen die parallel entwickelten Einheiten im Anschluss nicht zusammen, oder die Einheiten sind so groß, dass kein kurzfristiges Feedback möglich ist.

Komponenten finden

Wenn die Umsetzung des Inkrement arbeitsteilig mit mehreren Entwicklern erfolgen soll, muss die Zusammenarbeit organisiert werden. Es ist erforderlich, die Funktionseinheiten so zu trennen, dass parallel gleichzeitig an ihnen gearbeitet werden kann, ohne dass sich die Entwickler dabei gegenseitig behindern. Diese flüssige Zusammenarbeit gelingt am besten mit einer Arbeitsorganisation, die auf *Komponenten* basiert. Eine Komponente ist eine binäre Funktionseinheit mit separatem Kontrakt. Das bedeutet, der Kontrakt wird getrennt abgelegt von der Implementation des Kontrakts. Ganz praktisch gesehen wird der Kontrakt in einer eigenen Assembly oder einem eigenen JAR File abgelegt. Gegen diesen Kontrakt können dann mehrerer Entwickler gleichzeitig implementieren, ohne dass es zu Überschneidungen innerhalb der Codebasis kommt.

Soll eine Lösung arbeitsteilig implementiert werden, ist vor dem Kodieren eine Aufteilung der Lösung in Komponenten erforderlich. Die Klassen des Entwurfs werden dazu Komponenten zugeordnet, indem sie im Diagramm mit Komponentennamen versehen werden.

Kontrakte schreiben

Damit bei der arbeitsteiligen Umsetzung keine Missverständnisse entstehen, werden für die zu realisierenden Klassen *Interfaces* erstellt. Diese werden in einer getrennten Bibliothek (in .NET eine Assembly, in Java eine JAR Datei) abgelegt. Die Bibliothek wird anschließend von allen Komponenten referenziert. Im Anschluss kann dann die arbeitsteilige Implementation der einzelnen Klassen erfolgen. Dabei wird jeweils gegen

die hier definierten Interfaces implementiert. So ist sichergestellt, dass die gemeinsam entworfenen Methodennamen und Signaturen eingehalten werden. Neben den Interfaces müssen auch die im Entwurf verwendeten eigenen Datentypen im Kontrakt abgelegt werden, damit sie von den beteiligten Komponenten gemeinsam verwendet werden können.

Zuordnen von Funktionseinheiten zu Teammitgliedern

Der letzte Schritt der Arbeitsorganisation besteht darin, die Komponenten jeweils einem oder zwei Entwicklern zuzuordnen. In der Regel genügt es, einen Entwickler pro Komponente zuzuordnen. Durch die Entwurfsphase ist allen Entwicklern klar, was zu tun ist. Sie haben als Team gemeinsam das Problem durchdacht und eine Lösung entworfen. Insofern geht es bei der Umsetzung um das bloße Kodieren. Es entsteht nur in sehr seltenen Fällen ein Vorteil durch *Pair Programming*. Dies kann hilfreich sein, um neue Kollegen in die Domäne einzuführen. Ferner kann es die Einarbeitung in eine neue Programmiersprache oder Plattform unterstützen. Der Regelfall sollte allerdings sein, dass jeder Entwickler einzeln eine Komponente umsetzt.

Implementation

Nach all den Vorarbeiten kann nun das Inkrement kodiert werden.

Implementation, mit Tests

Zur Kodierung gehören automatisierte Tests. Diese sind Stand der Technik. Es gibt heute keine Entschuldigung dafür, keine Tests zu schreiben. Vorteilhaft ist es, *test-first* vorzugehen. Man schreibt erst einen Test und implementiert dann gerade so viel, dass der Test grün ist. Anschließend wird der nächste Test geschrieben und die Implementation ergänzt. Diese Vorgehensweise hat den Vorteil, dass es zu einer hohen Testabdeckung kommt. Dadurch, dass der Test zuerst da ist, wird die Implementation sehr fokussiert vorgenommen. Es geht mit jedem kleinen Schritt nur darum, einen kleinen Testfall auf grün zu kriegen. Dadurch entsteht ein sehr bewusstes Arbeiten an der Kodierung. Des Weiteren führt eine *test-first* Vorgehensweise zu Code, der testbar ist. Statt im Anschluss an das Kodieren festzustellen, dass der Code nur schwer automatisiert getestet werden kann, entsteht der Test zuerst.

Integration, mit Tests

Nachdem alle entworfenen Funktionseinheiten implementiert sind, müssen diese integriert werden. Die Integrationsmethoden können erst implementiert werden, wenn die Funktionseinheiten, die integriert werden, verfügbar sind. Zwar kann dies durch den Einsatz von Interfaces so vorgezogen werden, dass die Umsetzung zeitlich entkoppelt ist. Doch wird man dies lediglich im Falle einer arbeitsteiligen Umsetzung in Betracht ziehen, weil dann die Interfaces ohnehin als Kontrakte erforderlich sind. Und auch bei der arbeitsteiligen Umsetzung spricht nichts dagegen, die Integration nach der Umsetzung der einzelnen Operationen gemeinsam im Team vorzunehmen. Dies kann die Einleitung des ohnehin anstehenden Code Reviews sein: das Team trifft sich am Beamer und beginnt zunächst damit, die Integrationsmethoden gemeinsam zu realisieren.

Wie zuvor bei der Implementation der einzelnen Bestandteile müssen nun auch bei der Integration automatisierte Tests erstellt werden. Diese *Integrationstests* sind ein wesentlicher Bestandteil der Teststrategie.

Code Review

Nach der Umsetzung des Inkrements muss dieses vom Team einem Code Review unterzogen werden. Dies ist erforderlich, um gemeinsam den entstandenen Code daraufhin zu untersuchen, ob er alle Anforderungen umsetzt. Ferner muss diskutiert werden, ob die Codequalität den Ansprüchen genügt. Das Code Review ist auch die Stunde der Wahrheit, was die automatisierten Tests angeht. Hier sollte unbedingt drauf geachtet werden, dass eine ausreichende Testabdeckung erreicht wird und die Tests leicht verständlich sind. Es wird also die Qualität sowohl des Implementationscodes diskutiert als auch die des Testcodes.

Abnahme durch den Product Owner

Als letzter Schritt des Entwicklungsprozesses wird das Inkrement an den Product Owner zur Abnahme übergeben. Der Product Owner muss eindeutig bewerten, ob er das Inkrement akzeptiert oder zurückweist. Erst wenn das Inkrement akzeptiert wurde, kann mit dem nächsten Inkrement begonnen werden. Bei einer Zurückweisung werden zunächst die Nachbesserungen oder Änderungen vorgenommen, bevor das Team mit der Arbeit am nächsten Inkrement beginnt.

Übersetzung in Code

In den folgenden Abschnitten sind die einzelnen Elemente der Flow Design Syntax aufgeführt. Es wird jeweils gezeigt, wie die einzelnen Bestandteile eines flow Design Entwurfs in konkreten Code übersetzt werden können. Dabei sind die Übersetzungen in C# aufgeführt. Eine Übertragung auf andere Programmiersprachen sollte kein Problem darstellen.

Die Übersetzung eines Entwurfs in Programmcode muss ein sehr wesentliches Prinzip erfüllen: die Implementation spiegelt den Entwurf¹². Die Implementation muss ein Abbild des Entwurfs sein, da der Entwurf nur so zur Orientierung innerhalb der Implementation dienen kann. Wird der Entwurf abweichend implementiert, besteht schon im darauf folgenden Code Review Verunsicherung, da sich niemand mehr zurecht findet. Die klaren Übersetzungsregeln sind dabei gar nicht als Nachteil zu betrachten. Sie geben Orientierung und bilden das Gerüst, um ausgehend vom Entwurf zum konkreten Code zu gelangen.

¹² http://clean-code-developer.de/die-grade/blauer-grad/#Implementation_spiegelt_Entwurf

Eingehende Datenflüsse

Eine Funktionseinheit kann über unterschiedliche Formen von Eingängen verfügen. Ein eingehender Datenfluss ist zwingend erforderlich. Andernfalls wäre nicht klar, zu welchem Zeitpunkt die Funktionseinheit ausgeführt wird. Benötigt eine Funktionseinheit keine Eingabedaten, ist der Datenfluss leer, existiert jedoch als Signal für den Ausführungszeitpunkt.

Eingehende Datenflüsse werden im folgenden dargestellt und in Code übersetzt.

Single Path

Ein einzelner eingehender Datenfluss ist der am häufigsten anzutreffende Fall für eine Funktionseinheit.

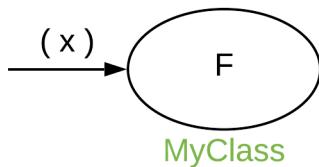


Abbildung 207: Single Path Input

Die Funktionseinheit kann in diesem Fall sehr leicht in eine Methode übersetzt werden. Die eingehenden Daten werden zu Parametern der Methode.

Der Name der Funktionseinheit wird in diesem Fall zum Methodennamen. Es fehlt somit noch die Zuordnung der Methode zu einer Klasse. Der Klassenname muss daher im Entwurf ergänzt werden, bevor mit der Implementation begonnen werden kann. In der Regel wird der Klassenname in einer anderen Farbe unter der Funktionseinheit notiert.

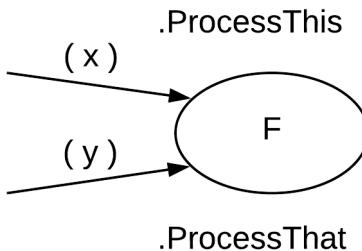
```

public class MyClass
{
    public void F(int x) {
        // ...
    }
}

```

Multiple Path

Verfügt eine Funktionseinheit über mehr als einen eingehenden Datenfluss, kann sie nicht mehr vollständig als Methode realisiert werden. Die eingehenden Datenflüsse können voneinander unabhängig Daten zur Funktionseinheit transportieren.



heit transportieren.

Bei einer Übersetzung in eine Methode müsste die Methode mit unterschiedlichen Daten aufgerufen werden können. Das wäre zwar im technischen Sinne machbar, würde jedoch dazu führen, dass es innerhalb der Methode zu Fallunterscheidungen käme. Um gut lesbaren und damit leicht verständlichen Code zu erreichen, wird eine Funktionseinheit mit mehr als einem eingehenden Datenfluss in eine Klasse übersetzt. Die eingehenden Datenflüsse werden zu Methoden dieser Klasse. Der Name der Funktionseinheit aus dem Entwurf wird nun zum Namen der Klasse. Es fehlen die Namen für die Methoden.

Da nun die eingehenden Datenflüsse namentlich unterscheidbar gehalten werden müssen, muss bereits im Entwurf jeweils ein Name für die eingehenden Datenflüsse vorgesehen werden. Der Name der Funktionseinheit genügt nicht mehr, da er zum Klassennamen wird.

```

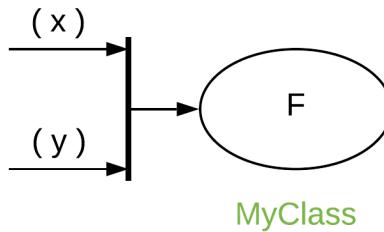
public class F
{
    public void ProcessThis(int x) {
        // ...
    }

    public void ProcessThat(string y) {
        // ...
    }
}

```

Joined

Bei einem Join werden die eingehenden Datenflüsse so zusammengefasst, dass ein Tupel der Daten entsteht.



MyClass

Daher wird die Funktionseinheit in eine Methode übersetzt. Die einfließen-den Daten werden jeweils zu Parametern der Methode.

```

public class MyClass
{
    public void F(int x, int y) {
        // ...
    }
}

```

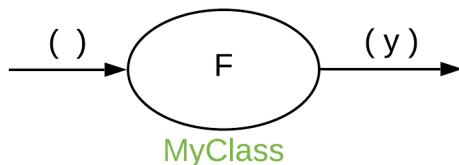
Ausgehende Datenflüsse

Eine Funktionseinheit kann über ausgehende Datenflüsse verfügen. Im Gegensatz zu eingehenden Datenflüssen sind diese tatsächlich optional, da Funktionseinheiten häufig zu einem Seiteneffekt führen, statt Ausgangsdaten zu produzieren.

In den Abbildungen sind die Funktionseinheiten lediglich mit ausgehenden Datenflüssen dargestellt, da dies der Fokus des Kapitels ist. Funktionseinheiten ohne einen eingehenden Datenfluss sind syntaktisch falsch, weil so unklar bleibt, zu welchem Zeitpunkt die Funktionseinheit ausgeführt wird.

Single Path

Bei ausgehenden Datenflüssen ist der einfachste Fall eine Funktionseinheit mit einem ausgehenden Datenfluss.



Wird die Funktionseinheit als Methode übersetzt, kann der eine ausgehende Datenfluss als Rückgabewert der Methode implementiert werden.

```
public class MyClass
{
    public int F() {
        // ...
        return 42;
    }
}
```

Schwierig kann es werden, wenn die Funktionseinheit auf dem einen Datenfluss mehr als ein Datum liefert. Viele Programmiersprachen bieten inzwischen syntaktisch die Möglichkeit, direkt ein Tupel als Rückgabewert zu liefern. Man kann also bspw. schreiben `return (x, y)`. Bei Sprachen, die das nicht bieten, müssen die Werte explizit zu einem Tupel Datentyp zusammengefasst werden. Entweder wird dazu ein generischer Tupel Typ aus dem Framework verwendet, oder es wird ein Typ eigens für den Anwendungsfalls definiert.

Alternativ kann der ausgehende Datenfluss als *Continuation* oder *Callback* realisiert werden.

```
public class MyClass
{
    public void F(Action<int> onResult) {
        // ...
        onResult(42);
    }
}
```

Die Methode erhält als zusätzlichen Parameter einen Funktionszeiger. Die so übergebene Methode wird aufgerufen, um den produzierten Ausgangswert weiterzugeben. Diese Form der Übersetzung ist erforderlich, wenn der ausgehende Datenfluss optional ist oder als *Stream* mehrfach ausgeführt werden kann.

Eine Verallgemeinerung der Continuation ist ein *Event* der Klasse.

```
public class MyClass
{
    public event Action<int> OnResult;

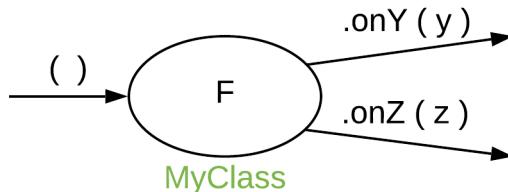
    public void F() {
        // ...
        OnResult(42);
    }
}
```

Dies ist dann sinnvoll, wenn mehr als ein Nachfolger mit dem ausgehenden Datenfluss verbunden werden muss, da in diesem Fall das Multicast Verhalten des Events genutzt werden kann. Wichtig ist festzuhalten, dass die Verständlichkeit der API der Klasse darunter leidet. Nun muss der Aufrufer daran denken, vor dem Aufruf der Methode den Empfänger des

Ergebnisses an den Event zu binden. Verfügt die Methode selbst über einen Continuation Parameter, kann sie gar nicht ohne den Handler aufgerufen werden.

Multiple Path

Verfügt eine Funktionseinheit über mehr als einen ausgehenden Datenfluss, müssen diese entweder als Continuation oder als Event implementiert werden.



Da die Daten auf mehreren ausgehenden Datenflüssen unabhängig voneinander fließen können, kann dieses Verhalten nicht mit dem Rückgabewert einer Methode realisiert werden.

```
public class MyClass
{
    public void F(Action<int> onY, Action<int> onZ) {
        // ...
        onY(56);
        onZ(42);
    }
}
```

Stattdessen werden der Methode für jeden Ausgang Callbacks als Parameter übergeben. Die Methode kann dann diese Callbacks aufrufen, wenn ein Wert auf dem ausgehenden Datenfluss übertragen werden soll.

Alternativ zu Callbacks können auch hier wieder Events verwendet werden. In dem Fall wird die Funktionseinheit in eine Methode übersetzt, die alle eingehenden Daten per Parameter erhält. Alle ausgehenden Daten werden über Events ausgedrückt. Die Methode ruft dann die Events auf anstelle der Callbacks, um ausgehende Daten fließen zu lassen.

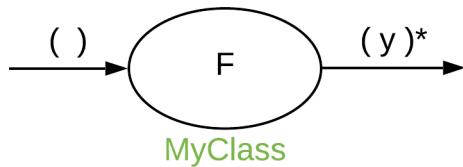
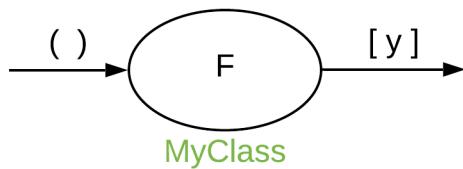
```
public class MyClass
{
    public event Action<int> OnY;

    public event Action<int> OnZ;

    public void F() {
        // ...
        OnY(56);
        OnZ(42);
    }
}
```

Optional

Die Darstellung eines optionalen Ausgangs ist auf zwei Weisen möglich. Entweder durch eckige Klammern oder allgemeiner durch den Stern außerhalb der Klammern. Der Stern außerhalb der Klammern ist im Vergleich zu den eckigen Klammern nicht so präzise, da dies $0..n$ Datenflüssen ausdrückt. Die eckigen Klammern stehen explizit für $0..1$ Datenflüsse.



Ein optionaler ausgehender Datenfluss wird am besten mittels Callback realisiert. Das Callback wird abhängig von einer Bedingung aufgerufen und ist somit optional. Die Beziehung zwischen einem Aufruf der Methode und dem Datenfluss ist somit *0..1*.

```

public class MyClass
{
    public void F(Action<int> onResult) {
        if( ... ) {
            onResult(42);
        }
    }
}

```

Alternativ können wieder Events verwendet werden.

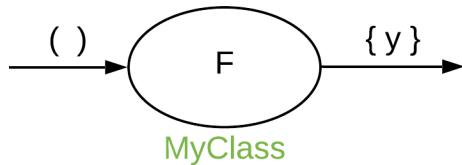
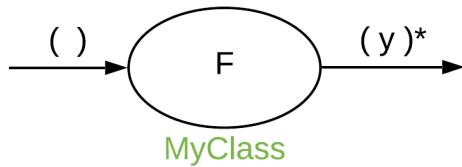
```
public class MyClass
{
    public event Action<int> OnResult;

    public void F() {
        if( ... ) {
            OnResult(42);
        }
    }
}
```

Eine Übersetzung als Rückgabewert der Methode ist nicht möglich, da dieser Rückgabewert technisch gesehen nicht optional sein kann. Das *return* Statement ist bindend für eine Methode mit Rückgabewert. Einen speziellen Wert zurück zu liefern, der ausdrückt, dass kein Wert vorliegt, sollte vermieden werden. Dies würde für den Aufrufer bedeuten, dass er den Rückgabewert überprüfen muss. Damit wäre das *if*-Statement beim Aufrufer, einer Integration, anstatt in der Operation. Sollte doch mit einem Rückgabewert gearbeitet werden, darf hier keinesfalls *null* die Bedeutung haben, dass kein Wert vorliegt. Dies birgt die Gefahr unnötiger *null* Zugriffe, weil Entwickler vergessen, den Wert vor dem Zugriff auf *null* zu prüfen. Die Alternative zu *null* wäre hier ein Typ *Option<T>*, der allerdings im .NET Framework nicht standardmäßig zur Verfügung steht. Es muss dazu ein entsprechendes NuGet Paket eingebunden werden.

Stream

Ein ausgehender Datenfluss, der mehrfach Daten liefert, wird als *Stream* bezeichnet.



Seine Umsetzung erfolgt entweder mit einem Callback oder einem Event. Die Continuation liefert bei jedem Aufruf ein Element des Streams. Da der Aufrufer nicht erkennen kann, wann das letzte Element des Streams geliefert wurde, muss dies durch ein spezielles end-of-stream Signal mitgeteilt werden. Dazu kann bspw. nach dem letzten Element einmal *null* als end-of-stream Kennzeichen geliefert werden. Allerdings bringt dies wieder das Risiko, dass der Aufrufer dies nicht bedenkt und es somit zu Null Pointer Exceptions kommt.

```

public class MyClass
{
    public void F(Action<string> onResult) {
        // ...
        foreach(var s in new[]{"1", "2"}) {
            onResult(s);
        }
        onResult(null); // end-of-stream
    }
}

```

Alternativ kann das Ende des Streams über einen weiteren Callback an den Aufrufe mitgeteilt werden. So wird dem Aufrufer explizit klar, dass das Ende des Streams signalisiert wird.

```
public class MyClass
{
    public void F(Action<string> onResult, Action onEndOfStream) {
        // ...
        foreach(var s in new[]{"1", "2"}) {
            onResult(s);
        }
        onEndOfStream();
    }
}
```

Anstelle der Callbacks können auch hier wieder Events zum Einsatz kommen.

```
public class MyClass
{
    public event Action<string> OnResult;

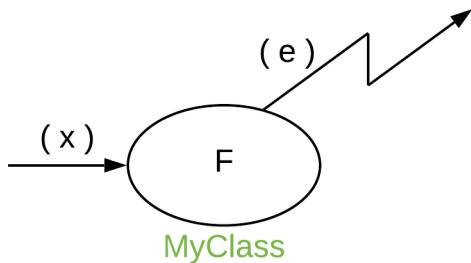
    public void F() {
        // ...
        foreach(var s in new[]{"1", "2"}) {
            OnResult(s);
        }
        OnResult(null); // end-of-stream
    }
}
```

Unter .NET steht mit *yield return* ein syntaktisches Konstrukt zur Verfügung, mit dem Streams auf sehr elegante Weise implementiert werden können. Die einzelnen Elemente des Streams können mit einer *foreach* Schleife iteriert werden. Innerhalb des Enumerator Protokolls steht implizit die end-of-stream Erkennung zur Verfügung, da der Aufrufer das IEnumerable<T> in einer Schleife iteriert.

```
public class MyClass
{
    public IEnumerable<string> F() {
        // ...
        foreach(var s in new[]{"1", "2"}) {
            yield return s;
        }
    }
}
```

Exceptions

Die Übersetzung einer Ausnahme wird syntaktisch durch das Auslösen einer *Exception* gelöst. Wir unterscheiden hier lediglich, ob die Funktionseinheit neben der Exception auch über einen ausgehenden Datenfluss verfügt.

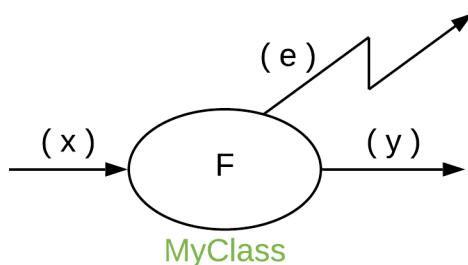


```

public class MyClass
{
    public void F(int x) {
        // ...
        if(true /* ... */) {
            throw new Exception("Error");
        }
    }
}

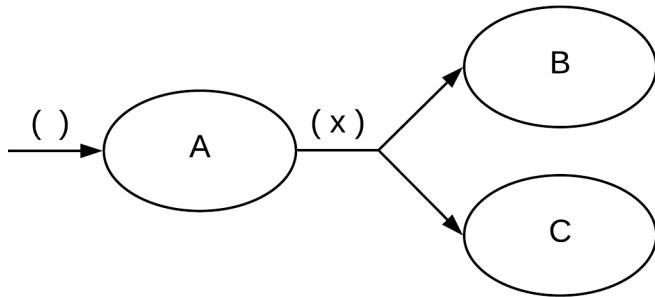
```

Verfügt die Funktionseinheit zusätzlich zur Exception über einen ausgehenden Datenfluss, wird dieser, wie sonst auch, als Rückgabewert der Funktion realisiert.



```
public class MyClass
{
    public int F(int x) {
        // ...
        if(false /* ... */) {
            throw new Exception("Error");
        }
        else {
            return 42;
        }
    }
}
```

Split



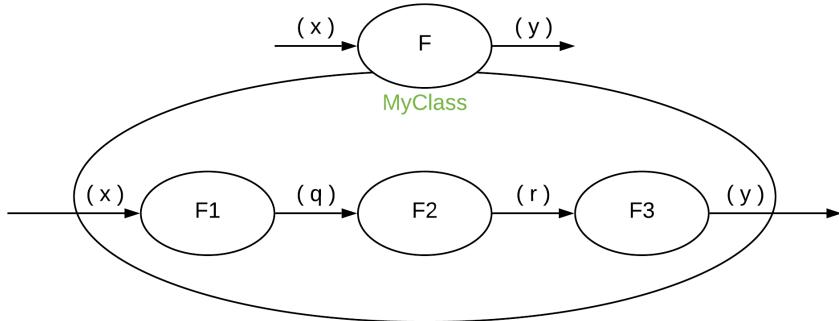
Das Splitten eines Datenflusses findet in der Implementation keinen expliziten Niederschlag. Die Daten, die in der Implementation in Variablen liegen, werden einfach an mehrere Funktionseinheiten übergeben.

```
var x = A();  
B(x);  
C(x);
```

In diesem Beispiel wird der Wert x sowohl an B als auch an C übergeben.

Hierarchien

Im folgenden Beispiel ist die Funktionseinheit F verfeinert durch die drei Funktionseinheiten $F1$, $F2$ und $F3$.



Die Implementation durch Methoden ist simpel: alle Funktionseinheiten werden jeweils in eine Methode übersetzt. Die Methode F ruft die drei Methoden $F1$, $F2$ und $F3$ auf. Im folgenden Listing sind alle vier Methoden in der selben Klasse $MyClass$ implementiert.

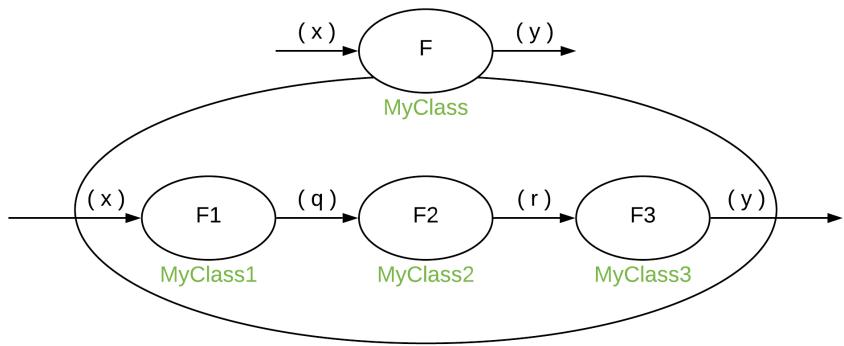
```
public class MyClass
{
    public int F(int x) {
        var q = F1(x);
        var r = F2(q);
        var y = F3(r);
        return y;
    }

    private int F1(int x) {
        return x + 1;
    }

    private int F2(int q) {
        return q * 2;
    }

    private int F3(int r) {
        return r - 3;
    }
}
```

Sind die Aspekte der Methoden $F1$, $F2$ und $F3$ unterschiedlich, sollten diese aufgrund des *Single Responsibility Principle* (SRP) auf unterschiedliche Klassen verteilt werden.



In der Implementation müssen dann ggf. Instanzen der Klassen erzeugt werden, sofern die Methoden nicht statisch definiert sind.

```
public class MyClass
{
    private MyClass1 _myClass1;
    private MyClass2 _myClass2;
    private MyClass3 _myClass3;

    public MyClass() {
        _myClass1 = new MyClass1();
        _myClass2 = new MyClass2();
        _myClass3 = new MyClass3();
    }

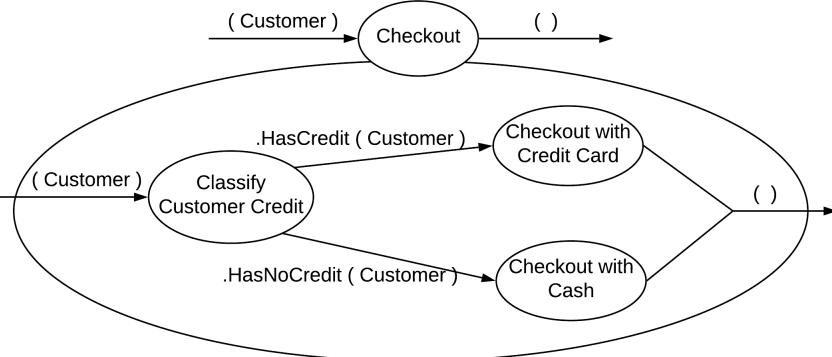
    public int F(int x) {
        var q = _myClass1.F1(x);
        var r = _myClass2.F2(q);
        var y = _myClass3.F3(r);
        return y;
    }
}
```

Hier werden die Instanzen der beteiligten Klassen *MyClass1*, *MyClass2* und *MyClass3* von der integrierenden Klasse *MyClass* in ihrem Constructor erzeugt. Da die Klasse *MyClass* ausschließlich für die Integration der anderen Klassen zuständig ist, kann in der Regel auf Dependency Injection verzichtet werden.

```
public class MyClass1 {  
    public int F1(int x) {  
        return x + 1;  
    }  
}  
  
public class MyClass2 {  
    public int F2(int q) {  
        return q * 2;  
    }  
}  
  
public class MyClass3 {  
    public int F3(int r) {  
        return r - 3;  
    }  
}
```

Fallunterscheidungen

Eine Fallunterscheidung zeichnet sich dadurch aus, dass die Funktionseinheit mehrere Ausgänge hat. In der Regel werden auf den Ausgängen unterschiedliche Daten fließen. Somit kann eine Fallunterscheidung nicht als "normale" Methode mit einem Rückgabewert implementiert werden.



Gesucht ist eine Übersetzung in eine Methode, die unterschiedliche Rückgabewerte zur Verfügung stellen kann. Dazu gibt es mehrere Möglichkeiten. Eine Möglichkeit besteht darin, die einzelnen Ausgänge jeweils als *Continuation* oder *Callback* zu implementieren. Die Methode trifft die Entscheidung und ruft dann den passenden Nachfolger auf. Dieser Aufruf erfolgt jedoch nicht unmittelbar sondern mithilfe von Funktionszeigern, die der Methode als Parameter übergeben werden.

Als Alternative kann die Fallunterscheidung auch so implementiert werden, dass die Methode einen *bool* oder einen *enum* als Rückgabewert liefert. In diesem Fall muss die Integrationsfunktion auf den Rückgabewert reagieren und dann entsprechend den richtigen Nachfolger aufrufen. Dies birgt die Gefahr, dass später die Bedingung, auf der die Fallunterscheidung getroffen wird, in der Integrationsfunktion ergänzt wird. Diese Gefahr besteht bei Verwendung von Continuations nicht. Andererseits sind Continuations für manche Entwickler immer noch syntaktisches Neuland. Das spricht manchmal dafür, mit Rückgabewerten zu arbeiten.

Continuations

```
public class CheckoutProcessor
{
    public void Checkout(Customer customer) {
        ClassifyCustomerCredit(customer,
            CheckoutWithCreditCard,
            CheckoutWithCash);
    }

    public void ClassifyCustomerCredit(
        Customer customer,
        Action<Customer> hasCredit,
        Action<Customer> hasNoCredit) {
        if(customer.Balance >= 1000.0) {
            hasCredit(customer);
        } else {
            hasNoCredit(customer);
        }
    }

    public void CheckoutWithCreditCard(Customer customer) {
        // ...
    }

    public void CheckoutWithCash(Customer customer) {
        // ...
    }
}
```

if mit bool oder enum

```
public class CheckoutProcessor
{
    public void Checkout(Customer customer) {
        if(ClassifyCustomerCredit(customer)) {
            CheckoutWithCreditCard();
        }
        else {
            CheckoutWithCash();
        }
    }

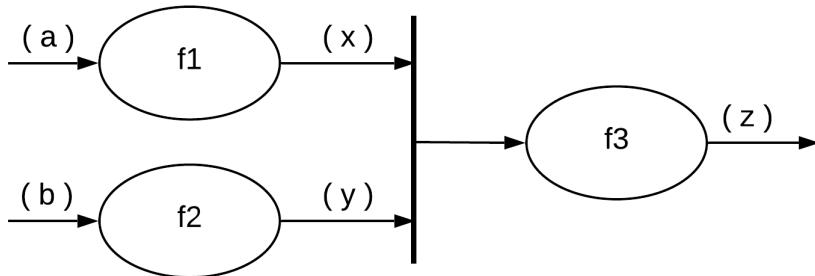
    public bool ClassifyCustomerCredit(Customer customer) {
        return customer.Balance >= 1000.0;
    }

    public void CheckoutWithCreditCard() {
        // ...
    }

    public void CheckoutWithCash() {
        // ...
    }
}
```

Join

Ein *Join* drückt sich im Code nicht durch einen eigenständigen Methodenaufruf aus. Der Join findet in der Implementation dadurch implizit statt, dass mehrere Variablen an den nächsten Methodenaufruf übergeben werden. Der implizite Join entsteht somit dadurch, dass Variablen aus dem Aufruf anderer Funktionseinheiten gesetzt werden und diese Werte dann gemeinsam in einem weiteren Methodenaufruf verwendet werden.



```
var x = f1(a);
var y = f2(b);
var z = f3(x, y);
```

Zustand

Eine Funktionseinheit kann Zustand halten, ohne dass dies speziell gekennzeichnet werden muss. In einigen Fällen ist es allerdings hilfreich, den Zustand in einem Flow Design Diagramm explizit auszudrücken, um so das Verständnis zu erleichtern bzw. die spätere Implementation zu präzisieren.

Zustand wird in Flow Design durch "Tonnen" dargestellt. Dabei unterscheiden wir zwischen Zustand, der innerhalb einer Funktionseinheit gehalten wird und Zustand, der im Flow steht. Die beiden folgenden Abbildungen zeigen die beiden Varianten.

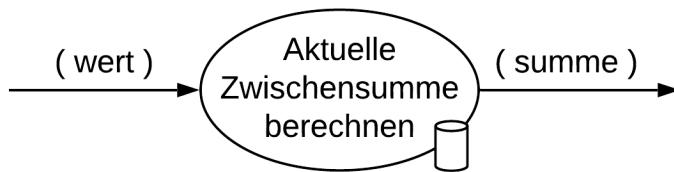


Abbildung 208: Eine Funktionseinheit mit Zustand

Hier hat die Funktionseinheit selbst Zustand. Um zu jedem neu gelieferten Wert die aktualisierte Zwischensumme liefern zu können, merkt sich die Funktionseinheit den Wert intern.



Abbildung 209: Eine Funktionseinheit ohne Zustand. Der Zustand fließt stattdessen.

Alternativ kann die Zwischensumme berechnet werden, in dem der Funktionseinheit die bisherige Summe zusätzlich zum Wert übergeben wird. Die Funktionseinheit aktualisiert die Summe und liefert diese als Resultat. Damit ist das Problem des Zustands zum Aufrufer verlagert worden, da dieser nun die Zwischensumme mit anliefern muss. Um den Aufrufer von dieser Pflicht

zu entbinden, können wir zur ursprünglichen Form zurückkehren und durch eine Verfeinerung ausdrücken, wie das Thema Zustand gelöst werden soll.

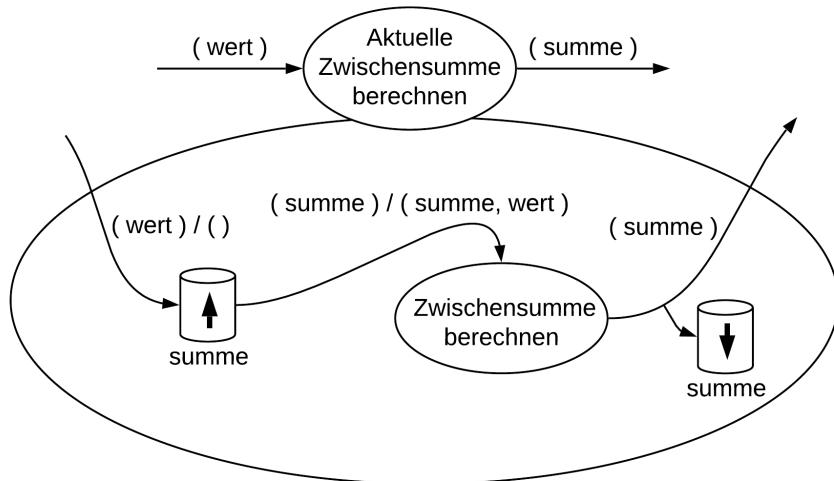


Abbildung 210: Der Zustand ist in den Flow gestellt.

In diesem Entwurf wird verdeutlicht, wie der Zustand innerhalb der Funktionseinheit realisiert wird. Die Verfeinerung zeigt, dass zunächst die bisherige Summe aus dem Zustand gelesen wird. Die Tonne mit Pfeil nach oben bedeutet, Zustand lesen. Anschließend fließt diese Summe mit dem aufzusummierenden Wert zur Funktionseinheit *Zwischensumme berechnen*. Diese liefert die korrigierte Summe als Ergebnis. Bevor dieses Ergebnis an den Aufrufen zurückgeliefert wird, muss der Zustand noch aktualisiert werden. Die Tonne mit Pfeil nach unten drückt aus, dass der Zustand geschrieben also verändert wird.

Die Implementation dieses Datenflusses ist in der folgenden Abbildung zu sehen.

```

public class Zwischensumme
{
    private int _summe;

    public int AktuelleZwischensummeBerechnen(int wert) {
        _summe = ZwischensummeBerechnen(_summe, wert);
        return _summe;
    }

    private int ZwischensummeBerechnen(int summe, int wert) {
        return summe + wert;
    }
}

```

Der Zustand, der in der Verfeinerung gelesen und geschrieben wird, ist in der Implementation als Feld der Klasse realisiert. Die öffentliche Methode *AktuelleZwischensummeBerechnen* reicht den Zustand *_summe* sowie den Wert als Parameter in die Methode *ZwischensummeBerechnen*. Innerhalb dieser Methode wird die Summe berechnet und als Rückgabewert geliefert. Vor der Rückgabe an den Aufrufer wird die aktualisierte Summe als neuer Zustand in das Feld *_summe* geschrieben.

Die oben gezeigte Implementation dient hier als Beispiel, wie man Zustand in einen Datenfluss stellen kann. Das Beispiel soll lediglich verdeutlichen, wie Zustand aus Operationen herausgehalten werden kann. Niemand soll eine Summenberechnung auf solch komplizierte Art implementieren. Das Beispiel zeigt aber auf, dass sich hieraus Vorteile für das automatisierte Testen ergeben.

Die Funktion *AktuelleZwischensummeBerechnen* verwendet intern den Zustand im Feld *_summe*. Wenn man einen Test schreiben möchte, der prüfen soll, was beim Aufruf passiert, wenn der interne Zustand einen bestimmten Wert hat, muss der Zustand zunächst über die öffentliche API hergestellt werden. Der Zustand ist nicht von außen direkt erreichbar. Dies ist auch gut so, denn dadurch sind Zustand und Methoden die auf dem Zustand arbeiten gekapselt. Häufig ist es allerdings wünschenswert, für die Tests auch auf Interna zugreifen zu können. Im obigen Sourcecode könnte das private Feld *_summe* oder die private Methode *ZwischensummeBerechnen* für Tests erreichbar gemacht werden. Bei .NET wird dies durch die Sichtbarkeit *internal* in Verbindung mit dem *InternalsVisibleTo* Attribut erreicht. Bei Java kann dies durch die *package private* Sichtbarkeit erreicht werden.

Es sei nochmals betont, dass die oben gezeigte Implementation ein konstruiertes Beispiel darstellt. Sie soll zeigen, dass man bei jeder Funktionseinheit entscheiden kann, ob diese selbst den Zustand hält oder ihn als Parameter mit jedem Aufruf reingereicht kriegt. Dies gilt es im Sinne der Testbarkeit abzuwägen.

UI

Bei der Verbindung zwischen der Benutzerschnittstelle (Ui) und “dem Rest” der Anwendung kommt es ganz wesentlich darauf an, die Abhängigkeiten in der richtigen Richtung zu gestalten: von der Ui zu den Interactors.

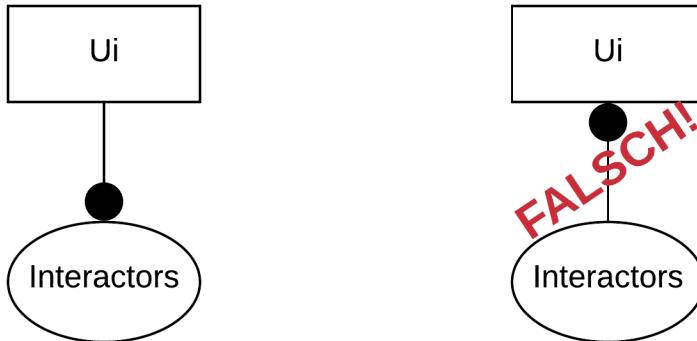


Abbildung 211: Die Ui darf von den Interactors abhängig sein, nicht umgekehrt!

Diese Abhängigkeitsrichtung ist wesentlich für die einfache Testbarkeit der Anwendung. Wenn man es falsch macht und die Interactors von der Ui abhängig sind, hat man es im Test mit der Ui zu tun! Mithilfe von Interfaces und Attrappen ist die Testbarkeit zwar grundsätzlich gegeben. Allerdings sind solche Tests deutlich aufwendiger. Insofern ist es fundamental, dass die Ui von den Interactors abhängig ist und nicht andersherum.

Die Abhängigkeit der Ui zu den Interactors kann direkt oder indirekt sein. Bei einer direkten Abhängigkeit ruft die Ui die Methoden des Interactors auf. In der indirekten Variante stellt die Ui Events/Observer/Callbacks zur Verfügung, mit denen die Interactors verbunden werden.

Keine Abhängigkeit zwischen Ui und Interactors

Die konsequenteste Umsetzung der Beziehung zwischen der Ui und den Interactors kommt ohne Abhängigkeit aus. Die Ui ruft die Interactors nicht selbst auf, sondern die Integration erfolgt eine Ebene höher.

```
var result = interactors.Start();
consoleUi.Spielbrett_anzeigen(result.Spielbrett, result.Meldung);
```

Dieser Codeausschnitt zeigt, dass die *Start* Methode des Interactors ein Ergebnis liefert, das dann an die Ui zur Anzeige übergeben wird. Auf diese Weise lässt sich *Start* leicht automatisiert testen. Es besteht keine Abhängigkeit zwischen Ui und Interactors. Die Beziehung wird durch den oben gezeigten Integrationscode hergestellt. Dieser Integrationscode kann bei einfachen Anwendungen in der *Main* Methode liegen. Alternativ wird dazu eine eigene Klasse angelegt, welche für die Integration zuständig ist.

Wenn man mit einer Abhängigkeit zwischen Ui und Interactors implementieren möchte, muss die Ui den Interactor aufrufen. Nicht umgekehrt. Es wäre fatal, wenn die *Start* Methode selbst die Ui aufruft. Dann liefert *Start* kein im Test überprüfbares Ergebnis. Stattdessen würde lediglich die Ausgabe in der Ui als Seiteneffekt produziert. Dieser Seiteneffekt ist nur schwer automatisiert zu testen.

Direkte Abhängigkeit der Ui zum Interactor

Wenn auf die externe Integration von Ui und Interactors verzichtet werden soll, kann die Ui die Interactors direkt ansprechen. Dies ist bspw. bei Web Anwendungen sinnvoll, bei denen die Ui Dialoge von einem Framework instantiiert werden. In dem Fall ist es meist recht kompliziert, in den Instantiierungsmechanismus einzugreifen, um die Integration der Ui mit den Interactors vorzunehmen.

Die oben bereits gezeigte *Start* Methode des Interactors wird in diesem Fall direkt von der Ui aufgerufen.

Glossar

Agilität

Unter **Agilität** versteht man eine Vorgehensweise, die zu **regelmäßigem** und **kurzfristigem Feedback** führt.

Anwendung, Application, App

Im Rahmen der **Anforderungszerlegung** liegt die **Anwendung** in der Hierarchie zwischen dem Bounded Context und dem Dialog. Somit lassen sich die Anforderungen eines Bounded Context potentiell auf mehrere Anwendungen verteilen. Diese Aufteilung dient dazu, den unterschiedlichen Rollen eine jeweils geeignete Benutzerschnittstelle anzubieten.

In der Softwarestruktur stellt die **Anwendung** die oberste Ebene dar. Sie entspricht in vielen Sprachen der **main** Methode.

Aspekt

Ein **Aspekt** ist eine Zusammenfassung von **zusammengehörenden Eigenschaften**, die sich **getrennt** von anderen Eigenschaften **verändern** können.

Dialog

Eine **Dialog** ist eine Zusammenfassung von Interaktionen. Technisch wird ein Dialog als ein Fenster realisiert, in dem der Anwender die Möglichkeit hat, über diverse Steuerelemente mit der Anwendung zu interagieren.

Domänenlogik

Mit **Domänenlogik** wird die Logik bezeichnet, die sich dem **Thema des Softwaresystems** widmet.

Durchstich

Ein **Durchstich**, vertikaler Schnitt oder auch **Inkrement**, ist ein lauffähiges Programm, das installiert und vom Anwender bedient werden kann. Ein Durchstich realisiert einen kleinen Ausschnitt der Anforderungen, in dem vertikal durch die verschiedenen Aspekte geschnitten wird. Im Gegensatz dazu steht das horizontal Vorgehen, beim dem der Fokus auf einzelnen Aspekten liegt, anstatt auf einem Ausschnitt von Anforderungen.

Funktionseinheit

Funktionseinheit ist im Entwurf der Überbegriff für etwas, das eine bestimmte Funktionalität liefert. Beim Übergang vom Entwurf zur Implementation ist für jede Funktionseinheit zu entscheiden, ob sie in eine Methode oder Klasse übersetzt wird.

Inkrement

Siehe **Durchstich**.

Integration

Eine Funktionseinheit, die andere Funktionseinheiten integriert, nennt sich **Integration**. Technisch gesehen handelt es sich um eine Methode, die andere Methoden der Lösung aufruft. In einer Integration werden keine API oder Frameworkmethoden aufgerufen. Ferner werden keine Ausdrücke verwendet. In dem Fall würde es sich um eine Operation handeln.

Interaktion

Mit **Interaktion** bezeichnen wir das Bedienen einer Anwendung durch den Anwender. Der Anwender interagiert mit der Anwendung, indem er Steuerelemente bedient. Er betätigt bspw. Schaltflächen oder wählt Menüpunkte aus. In Abgrenzung zu technischen Ereignissen, die dabei auftreten, zeichnet sich eine Interaktion dadurch aus, dass für das gewünschte Verhalten Domänenlogik implementiert werden muss.

Interaktionsdiagramm

In einem **Interaktionsdiagramm** wird dargestellt, auf welche Weise der Anwender mit der Anwendung interagieren kann. Dazu werden die **Dialoge** skizziert und in diese die **Interaktionen** eingezeichnet.

Interaktor

Ein **Interaktor** (engl. Interactor) ist eine Funktionseinheit, welche die Domänenlogik einer Interaktion realisiert.

Iteration

Eine zeitliche Einheit eines Softwareentwicklungsprozesses wird **Iteration** genannt. In Scrum stellt ein Sprint eine Iteration dar. Der Sprint kann allerdings auch in kleinere Iterationen aufgeteilt werden.

Komponente

Eine **Komponenten** ist eine binäre Funktionseinheit mit separatem Kontrakt. Binär bedeutet hier, dass die Komponente als übersetzte, eben binäre, Bibliothek verwendet wird statt im Quelltext. Der Kontrakt ist separat in einer weiteren Bibliothek abgelegt. Komponenten vereinfachen die arbeitsteilige Implementation.

Modul

Modul ist der Überbegriff für Behälter, die Code bzw. Funktionalität enthalten. Module sind Methoden, Klassen, Bibliotheken, Komponenten und Services.

Operation

Eine Funktionseinheit, die Logik enthält, ist eine **Operation**. Technisch gesehen handelt es sich um eine Methode, die Ausdrücke, Verzweigungen, Schleifen und API Aufrufe enthält. Einer Operation ist es nicht gestattet, andere Methoden des Softwaresystems aufzurufen. Dann wären die Aspek-

te Integration und Operation vermischt und somit gegen das IOSP verstößen.

Portal

Ein **Portal** stellt die Schnittstelle dar zwischen einer Rolle und dem System. Eine Rolle interagiert durch ein Portal mit dem System. Für menschliche Rollen wird das Portal in Form von Dialogen realisiert. Fremdsysteme als Rollen erhalten bspw. eine HTTPS/REST Schnittstelle als Portal.

Product Owner

Der **Product Owner** ist gegenüber dem Team dafür verantwortlich, dem Team verbindlich die Anforderungen mitzuteilen und am Ende einer Iteration das Ergebnis abzunehmen und Feedback zu geben.

Provider

Ein **Provider** stellt die Schnittstelle zwischen dem System und einer Ressource dar. Es handelt sich um eine dünne Softwareschnittstelle, welche die konkrete API der Ressource kapselt.

Slash Notation

Mit der **Slash Notation** können Map und Join Operationen in einem Flow Design Diagramm reduziert werden. Dadurch können die Diagramme übersichtlicher werden.

Beispiel: (x, y) / (x)

Softwaresystem

Als **Softwaresystem** oder **System** bezeichneten wird die Zusammenfassung aller Anforderungen. In der Domänenzerlegung steht das System an oberster Stelle. Es wird zerlegt in Bounded Contexts, Anwendungen, Dialoge, Interaktionen und Features.

User Story

Mit **User Stories** können Anforderungen auf textuelle Weise beschrieben werden.

IOSP

Die Abkürzung **IOSP** steht für das *Integration Operation Segregation Principle*. Es besagt, dass eine Funktionseinheit entweder für Integration oder Operation zuständig sein soll.

PoMO

Mit der Abkürzung **PoMO** wird das *Principle of Mutual Oblivion* bezeichnet. Zu deutsch: gegenseitige Nichtbeachtung. Das Prinzip besagt, dass eine Funktionseinheit weder ihren Vorgänger noch ihren Nachfolger kennen sollte. Andernfalls würde es eine Abhängigkeiten geben. Die Verbindung wird stattdessen durch eine übergeordnete Integration hergestellt.

IODA Architektur

Mit **IODA Architektur** wird die Trennung von Integration, Operation, Daten und API bezeichnet. Die Abkürzung steht für *Integration Operation Data API* Architektur.