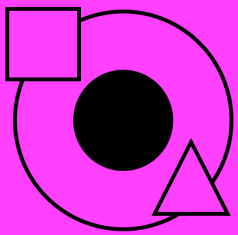
ralf westphal

SOFTWAREENTWURF MIT FLOW-DESIGN

PROGRAMMING WITH EASE TEIL 2



ein buch aus dem software universum

Softwareentwurf mit Flow-Design

Programming with Ease - Teil 2

Ralf Westphal

Dieses Buch wird verkauft unter
http://leanpub.com/softwareentwurf-mit-flow-design

Diese Version wurde veröffentlicht am 2021-04-16

Ebenfalls von Ralf Westphal

Test-first Codierung

Software Anforderungsanalyse mit Slicing

Die IODA Architektur im Vergleich

Inhaltsverzeichnis

Zum Geleit

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Motivation

Der Softwareentwicklung fehlt etwas. Was fehlt, ist eine Form von Klarheit und vor allem Gelassenheit. So ist zumindest mein Gefühl, wenn ich Softwareentwickler in meinen Clean Code Trainings oder auch an ihrem Arbeitsplatz beobachte.

Wo Klarheit und Gelassenheit sind, da ist der Tritt sicher, da ist die Zuverlässigkeit hoch, da stimmt die Qualität von vornherein umfassend und die Stimmung ist entspannt. Leider scheint mir das aber nicht die Atmosphäre in den meisten Softwareentwicklungsteams zu sein. Oder wie empfindest du es in deinem Team?

Stattdessen herrscht oft Verwirrung angesichts dessen, was der Kunde will, es sind die Backlogs voll mit Bugs und das sprichwörtliche "What the fuck?!" ist ständig hinter den Monitor-Triptychons im Team Room zu hören (oder zumindest auf den Gesichtern der Entwickelnden zu lesen).

Was mögen die Gründe dafür sein? Es gibt sicher viele. Ein ganz grundlegender scheint mir jedoch dieser: **Die Softwareentwicklung ist ins Ungleichgewicht gekommen**. **Sie erfüllt nicht in gleicher Weise systematisch und kompetent alle Anforderungen des Auftraggebers**. **Sie starrt auf die einen und lässt dabei einen blinden Fleck für die anderen entstehen**. Das führt früher oder später zu einem für den Auftraggeber sehr spürbaren Qualitätsdefizit, dessen Ausgleich schwerer und schwerer wird. Das Kind ist tief im Brunnen. Es fehlt einfach an Nachhaltigkeit.

Gelassenheit ist in solcher Situation nicht mehr möglich, wenn Klarheit über so lange Zeit so eklatant gefehlt hat. Programmierung mit Leichtigkeit sieht anders aus.

Mehr Technologie, mehr Infrastruktur ist darauf keine Antwort. Vielmehr ist - *horribile dictu!* - ein Kulturwandel nötig. Ohne grundsätzliches Umdenken geht es nicht. Die Grundhaltung ist zu verändern: **Es braucht ein Bewusstsein dafür, dass auch soetwas immaterielles wie Software, Nachhaltigkeit braucht**.

Wenn in deinem Team schon agil gearbeitet wird, hast du eine Ahnung, was Kultur und Kulturwandel bedeutet. Doch leider ist Agilität nicht genug für nachhaltige Softwareentwicklung. Sie ist zwar notwendig für die Nachhaltigkeit, die ich meine, aber nicht hinreichend.

Wie wichtig Nachhaltigkeit ist, weiß zwar schon lange jeder Koch und jeder Chirurg - doch die Softwareentwicklung hinkt leider noch hinterher. Die oberste Priorität haben bei Ersteren Sauberkeit und Hygiene; ohne sie sind Erfolge nur von kurzer Dauer. Wer eine Küche am Ende des Tages mit dreckigem Geschirr und voller Abfall zurücklässt, beschädigt die Grundlage für die Arbeit morgen. Wer heute Operationsbesteck nicht sterilisiert und am Ende einer Operation nicht zählt, riskiert Komplikationen morgen. Sauberkeit und Hygiene sind der Rahmen, in dem das Kochen und chirurgische Eingriffe stattfinden.

Ich bin überzeugt, dass für die Softwareentwicklung ein Nachhaltigkeitsrahmen erst noch solide aufgespannt werden muss. **Korrektheit und Ordnung sind noch nicht in gleicher Weise als Grundanforderungen in der Softwareentwicklung anerkannt wie Sauberkeit und Hygiene in anderen Branchen**. Das ist die fehlende Klarheit, die die Entwicklung von Gelassenheit verhindert.

Diese Situation verbessern zu helfen, ist mein Anliegen. Ich möchte dir helfen, klarer und gelassener zu programmieren. Weniger Stress durch mehr Nachhaltigkeit für deine Softwareentwicklung ist mein Ziel. Wie das erreicht werden kann, damit habe ich mich in den vergangenen 15 Jahren intensiv auseinandergesetzt. Ich hoffe, du empfindest das, was ich hier nun “an einem Ort” zusammentrage, als Hilfe in deinem Entwickleralltag.

-Ralf Westphal, 2020, Bansko (BG) / Hamburg (DE)

Programming with Ease

Nachhaltigere Softwareentwicklung in Klarheit und Gelassenheit umfasst für mich mehr, als ich dir in diesem Buch vorstellen kann. Mit ein paar Tipps&Tricks ist es nicht getan. Es geht durchaus ans Eingemachte: an deine Glaubenssätze und Gewohnheiten.

Die vielfach fehlende Nachhaltigkeit in der Softwareentwicklung ist ein so tiefliegendes Problem, dass einige Anstrengungen nötig sind, die Situation

zu ändern. Du wirst Zeit brauchen, anders wahrzunehmen, zu denken und dann zu handeln. Dein Team wird Zeit brauchen, denn in der Zusammenarbeit muss sich einiges ändern. Und schließlich wird sich sogar dein Management und dein Auftraggeber ebenfalls ändern müssen in den Erwartungen an dich und dein Team.

Das klingt nach einigem Aufwand, oder? Ja, stimmt. Leider kann ich dir den nicht ersparen. Das Wurzelproblem von "schwer wartbarer Software" liegt zu tief, als dass es dafür eine schnelle Lösung gäbe. Wenn du aber dran bleibst, dann bin ich gewiss, dass sich die Mühe lohnt.

Vermitteln möchte ich dir - und deinem Team - *Programming with Ease* als umfassende Herangehensweise an die Softwareentwicklung, die dich abholt bei der Konfrontation mit Anforderungen und begleitet bis zur Ablieferung von hochqualitativem Code.

Um moderne Technologien und technische Feinheiten geht es nicht. React, NoSql, GraphQL, Docker, Kubernetes, Kafka… all das ist darin kein Thema. Oder wenn, dann nur indirekt in Form von Prinzipien und Konzepten, die dir helfen sollen, solche Technologien einzuordnen.

Stattdessen geht es um Prinzipien und Praktiken der Softwareentwicklung. Das hört sich zwar nach "theoretischem Kram" an, doch sei gewiss, mir ist es sehr, sehr wichtig, dass die Theorie in der Praxis gegründet ist. Theoretische Überlegungen müssen zu praktisch hilfreichen Effekten führen. Deshalb kann ich es dir nicht früh genug mit auf den Weg geben:

Welche Empfehlungen du auch immer hier lesen magst, egal wie sehr ich sie begründe, sie stehen nie höher als der Zweck. Wenn du in einer bestimmten Situation also meinst, einem Zweck *nachhaltig* besser dienen zu können, als durch Befolgung einer Empfehlung… dann - *by all means* - weiche von der Empfehlung ab. Allerdings: Du solltest schon wissen, was du da tust. Habe also eine belastbare Begründung parat - wenn schon nicht mir gegenüber, dann aber für deine Teamkollegen.

Das Gesamtthema *Programming with Ease* ist also umfangreich. Wie ich es dir nahebringe, habe ich lange überlegt. Am Ende habe ich mich dann für 3 Bücher entschieden, die 1+3 Themenblöcke behandeln.

Test-first Codierung ist der erste Themenblock, auch wenn Codierung die letzte Hürde ist, die du in der Programmierung nehmen musst. Dennoch macht dieses Buch den Anfang in der Trilogie, weil es thematisch dir als Entwickler wahrscheinlich am nächsten liegt. Codierung ist

praktisch, Codierung wahrlich unausweichlich, Codierung hat technisch-technologischen Reiz. Ich hoffe, dort kann ich dich am besten abholen, wenn es schon so ans Eingemachte geht.

Im ersten Band geht es darum, dass Codierung aus meiner Sicht eben ausschließlich *test-first* stattfinden sollte. Das zu akzeptieren und dann auch zu leben, ist die erste Herausforderung auf dem Weg zu nachhaltiger Programmierung. Ich hoffe, dass ich dir die Gründe dafür im ersten Band ausführlich genug darlegen und dir diese Praxis mit verschiedenen Problemlösungsansätzen auch schmackhaft machen kann.

Softwareentwurf mit Flow-Design ist der zweite Themenblock, auch wenn Entwurf als Planung von Code der Codierung vorausgehen sollte. Weil "Planung" sich für dich aber vielleicht nicht so attraktiv anhört, wollte ich das Thema nicht im ersten Band der Reihe behandeln, auch wenn ich es für das wichtigste der drei Themen halte.

Ja, tatsächlich, ich hänge dem Glauben an, dass wir in der Programmierung mehr denken sollten. Mehr denken vor dem Codieren, ist der Nachhaltigkeit absolut zuträglich. Nicht, dass nicht gedacht würde - doch mein Eindruck ist, dass gewisse Themen dabei unberücksichtigt bleiben. Es wird z.B. viel über den rechten Einsatz von Technologien und Infrastruktur nachgedacht. Es wird auch viel über Agilität nachgedacht oder über DevOps. Und das ist alles gut und richtig. Doch es bleibt ein blinder Fleck. Um den dreht es sich bei *Programming with Ease* im Allgemeinen und bei *Flow-Design* im Speziellen: das ist die visuelle Modellierung von Softwarelösungen.

Der letzte Themenblock unter dem Bogen, den *Programming with Ease* spannt, ist dann die ***Software Anforderungsanalyse mit Slicing***. Damit gehe ich noch einen Schritt vor den Entwurf und möchte dir empfehlen, Anforderungen durch eine spezielle Entwicklerbrille zu betrachten. Durch die Brille der Agilität siehst du Anforderungen als User Stories, Storyboards, Epics oder gar Event Storms. Auch das ist alles wunderbar. Du sollst davon nichts aufgeben. Doch in meiner Erfahrung ist auch durch diese Brille etwas nicht sichtbar, das dir das Programmiererleben aber leichter machen würde.

Der agilen Herangehensweise fehlt eine gewisse technische Sicht. Das finde ich ganz verständlich, allemal da sich inzwischen Scrum und Kanban als Vorgehensmodelle etabliert haben und von XP nur noch wenig zu

hören ist.[1] Damit haben die "Softwarelaien" gewonnen, so dass Anforderungen von ihnen definiert werden, wie es für sie nachvollziehbar ist. Das soll natürlich auch so sein - nur darf eine Sichtweise, die dir als Programmierer dient, deshalb nicht vernachlässigt werden. Das scheint mir jedoch der Fall, so dass nachfolgende Phasen in der Programmierung dir schwerer fallen als nötig.

Insgesamt wird durch die Dominanz der "Softwarelaien" sogar mangelnder Qualität und Unzuverlässigkeit Vorschub geleistet. Ja, du liest richtig: *Real existierende* Agilität führt durchaus noch zu suboptimalen Ergebnissen. Das wird auch nicht besser, wenn du die Zähne noch tiefer in das agile Manifest schlägst. Es braucht einfach verschiedene Perspektiven. Agilität ist die eine. Das, was ich dir in *Programming with Ease* vermitteln will, ist eine zweite.

Codierung, Entwurf, Anforderungsanalyse sind die drei großen Themenblöcke in *Programming with Ease*. Damit verrate ich dir noch nicht zuviel an dieser Stelle. Ausführlicher begründet wird das in einem kleineren, übergreifenden Themenblock. Den umfasst die Einleitung und das erste Kapitel. Beides inklusive dieser Motivation wiederhole ich in allen Büchern, um dir zu ermöglichen, sie doch in einer anderen Reihenfolge zu lesen, als der hier vorgestellten. Zwar habe ich mir bei der Ordnung etwas gedacht - doch auch dafür gilt: zu eng solltest du das nicht sehen.

Einleitung und erstes Kapitel liefern den Hintergrund, vor dem ich die anderen Themen entfalte. Sie werden zuerst ganz grob in einem Zusammenhang entwickelt, damit du weißt, wie sie miteinander verbunden sind. Danach kommt die blockweise Vertiefung, bei der du diesen Hintergrund im Hinterkopf haben solltest.

Insgesamt ergibt sich hoffentlich für dich ein Gesamtrahmen, in dem du dich gut aufgehoben fühlst. Einfach(er) soll dir die Programmierung ja werden.

[1] Vielleicht kann man Software Craftsmanship als einen Arm der Entwicklung von XP verstehen. Der andere ist dann z.B. Scrum. Damit wären zwei Belange getrennt, die XP ursprünglich in XP vereint waren. Software Craftsmanship würde in dem Fall für die technische Seite von XP stehen. Ein blinder Fleck bliebe jedoch aus meiner Sicht. In XP wie in Software Craftsmanship findet sich schlicht zu wenig Methode. Beide sind Sammlungen von Bausteinen, zwischen denen kein roter Faden gespannt ist, an dem du dich konkret voranarbeiten könntest. Um genau den geht es mir aber.

Das Softwareuniversum

Wenn *Programming with Ease* ein Bogen ist, den ich über deinen Softwareentwicklungs*prozess* spannen möchte, also ein Bogen in der Zeit, dann ist das *Softwareuniversum* der dazugehörige Raum für Software*strukturen*

In diesem Raum spielt sich für mich alle Softwareentwicklung ab. Darin bewegst du dich mal langsamer mal schneller, mal in die eine Richtung, mal in die andere.

Allerdings ist der Raum des *Softwareuniversums* kein dreidimensionaler, sondern ein vierdimensionaler. Er besteht aus vier Dimensionen, die jede *Logik* auf eine andere Weise in Container fassen und zu Strukturen verbinden.

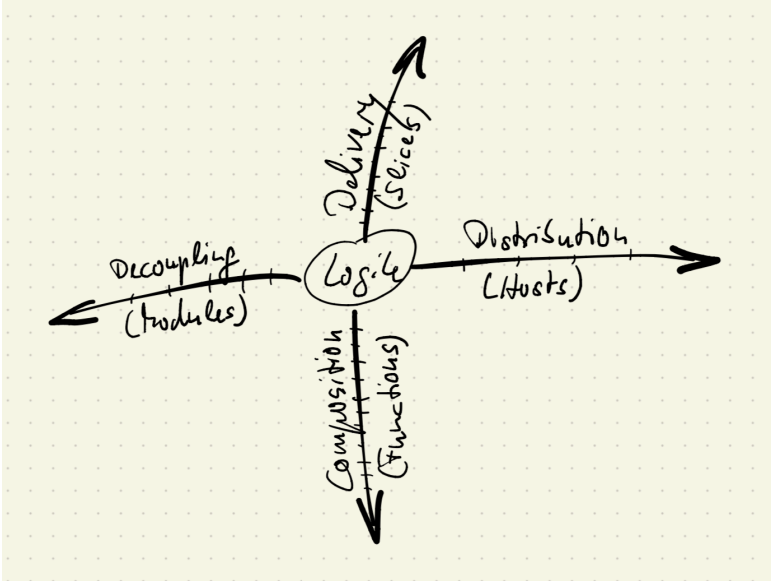
Was Logik ist, verrate ich dir in der Einleitung. An dieser Stelle nur soviel: sie ist die Essenz von Software. Dass du Logik in hoher Qualität schreibst, ist für den Kunden von höchster Wichtigkeit, denn sie bestimmt das Softwareverhalten. Du kannst sie also nicht einfach "hinklieren", sondern musst sie sorgfältig schneiden und verpacken.

1. Zunächst musst du das, was die Logik leisten soll, in möglichst feine Anforderungsscheiben schneiden beim *Slicing*. Darum geht es im dritten Band von *Programming with Ease*.
2. Dann musst du dir überlegen, wie du vor allem funktionale Anforderungen mit Logik so erfüllst, dass du sicher sein kannst, dass deine Lösung korrekt ist. Du musst dabei aus unzähligen fremden und eigenen Bausteinen *Kompositionen* herstellen, die du testen kannst. Das geschieht mit Funktionen und ist Thema des ersten Bandes und auch des zweiten Bandes.
3. Um nicht den Überblick über deine Komposite zu verlieren, teilst du sie in zweckvolle Gruppen auf mehreren Ebenen ein, die Zusammengehöriges *aggregieren* und von anderem entkoppeln; das sind die *Module* deiner Software. Darum geht es vor allem im zweiten Band, aber auch schon im ersten.
4. Und schließlich musst du dich leider noch einigen nicht-funktionalen Anforderungen widmen, die du auch mit sorgfältiger Komposition von Logik nicht lösen kannst. Es bleibt dir nichts anderes übrig, als Logik auf verschiedene *Hosts* zu verteilen. Darum geht es vor allem im zweiten Band, aber auch im dritten.

Zweck des Softwareuniversums ist es, die Strukturelemente, die du im Grunde schon aus deiner Programmierpraxis kennst - Beispielsweise Klasse, Thread, Service, Message, Funktion -, in einen Zusammenhang zu stellen. Sie bekommen alle einen klaren Zweck im Hinblick auf die umfassenden Anforderungen des Auftraggebers. Vor allem möchte ich dir jedoch zeigen, welche Rolle sie spielen in Bezug auf die Nachhaltigkeit.

Die vier Dimensionen des Softwareuniversums sind für mich:

- **Delivery**: Logik in Scheiben (*slices*) unterschiedlicher Dicke geschnitten für die iterativ-inkrementelle Lieferung an den Kunden.
- **Composition**: Logik zu **Funktionen** zusammengestellt, um funktionale wie nicht-funktionale Anforderungen zu erfüllen.
- **Decoupling**: Funktionen zu **Modulen** (z.B. Klassen) aggregiert, um Ordnung in die Vielfalt zu bringen. Testbarkeit und Wandelbarbeit sind der Gewinn.
- **Distribution**: Funktionen verteilt auf **Hosts** (z.B. Threads) und entkoppelt über asynchrone Kommunikation um weitere nichtfunktionale Anforderungen zu erfüllen.



Grobe Skizze des Softwareuniversums

Jede Zeile Logik, jeder Tropfen Essenz deiner Software, lässt sich im Softwareuniversum als Punkt im vierdimensionalen Raum verorten, da Logik immer gleichzeitig Teil einer Funktion in einem Modul in einem Host in einem Slice ist.

Das muss dir im Moment abstrakt vorkommen. Es fehlen ja auch noch viele Definitionen von Begriffen. Dennoch wollte ich dir das *Softwareuniversum* als Ausblick nicht vorenthalten. Als ich es das erste Mal erblickt habe, war es für mich ein wenig wie beim Overview Effect[2]: Ich konnte nun von außen überblicken, wovon ich vorher immer nur Teile gesehen hatte. Das hat mein Verständnis von Softwareentwicklung grundlegend verändert.

Deshalb gehören die Bände von *Programming with Ease* zu einer umfassenderen Reihe, die alle "im Softwareuniversum spielen".

[2] https://en.wikipedia.org/wiki/Overview_effect

Einleitung

Bevor ich dir konkrete “Tipps&Tricks” für die nachhaltige Softwareentwicklung gebe, möchte ich dir ein *big picture* skizzieren. Zu oft habe ich gehört und gelesen, dass einzelne Prinzipien und Praktiken empfohlen werden, ohne einen Kontext, ohne eine “Herleitung”. Bei aller Richtigkeit dieser Empfehlungen werden sie dann aber leicht missverstanden oder eben eingesetzt, wenn der Kontext nicht passt. Das führt zu Frustration. Die möchte ich dir ersparen, so weit es mir möglich ist.

Es ist schwer genug, all das in Worte, auch noch lineare zu fassen, was ich dir vermitteln will für nachhaltige Softwareentwicklung. Es wird mir auch nur bruckstückhaft gelingen. Dass du mich missverstehst, ist für mich vorhersehbar und unvermeidbar. Doch ich will mich bemühen, das zu minimieren. Und eine auf der Hand liegende Maßnahme dafür ist, dass ich etwas aushole, um einen Rahmen aufzuspannen, in dem das konkrete Thema dieses Buches und der anderen der Reihe eingehängt werden kann.

Deshalb: Halte einen Moment durch, bis es an das eigentlichen Thema dieses Bandes. Keine Sorge, du wirst davon genug zu sehen bekommen.

Und nun gehts los. Wo sonst als am Anfang jedes Softwareprojektes, bei den Anforderungen:

Anforderungskategorien

Softwareentwicklung hat Anforderungen in drei Kategorien zu erfüllen, um ihr Geld wert zu sein:

- Zunächst muss Softwareentwicklung funktionierende Software liefern. Auftraggeber haben **funktionale Anforderungen** an Software, die sie erfüllt sehen wollen. Nur dann hat die **Funktionalität** von Software hohe Qualität. Das ist so natürlich, dass es kaum der Rede wert ist - dennoch müssen wir da noch genauer hinschauen, auch wenn ich denke, mit diesen Anforderungen bist du bestens vertraut. Sie treiben dir genug Schweiß auf die Stirn.

- Funktionalität allein ist allerdings nicht genug - auch das ist dir klar - und noch nicht einmal der Grund für die Beauftragung von Softwareentwicklung. Software soll vor allem **nicht-funktionale Anforderungen** erfüllen! Sie soll Funktionalität *besser* (Komparativ!) anbieten als die Alternative (z.B. bisherige Software oder Handarbeit). Software soll z.B. schnell*er* oder einfach*er* oder skalierbar*er* oder sicher*er* funktionieren als die Alternative. Dann hat die **Effizienz**[3] von Software hohe Qualität. Das ist ebenso natürlich, dass es kaum der Rede wert ist - aber diese Anforderungen bereiten dir womöglich noch mehr Kopfschmerzen als die funktionalen.

Funktionale und nicht-funktionale Anforderungen zusammen sind **Verhaltensanforderungen** an Software. Der Auftraggeber kann durch Ausführung der Software überprüfen, ob die geforderte Qualität hergestellt wurde. Dieser Oberbegriff ist wichtig, wie du im Weiteren sehen wirst.

Vielleicht überraschend für dich, sehe ich Korrektheit darin noch nicht subsummiert. **Korrektheit** ist keine explizite weitere Anforderung an Software, sondern ist impliziet in der Erwartung, dass spezifizierte Anforderungen tatsächlich durch gelieferte Software erfüllt werden. Software ist also in dem Maße korrekt, in dem sie die Spezifikation erfüllt.

Mach dir an dieser Stelle keinen Kopf über den Begriff Spezifikation. Ich will damit keine Norm heraufbeschwören, sondern verstehe darunter lediglich eine irgendwie gearbeitet Liste von gewünschten Eigenschaften. Ob die auf einer Serviette stehen oder in einem 500seitigen Buch gebunden sind, ist einerlei. Der Kunde kann zur Laufzeit diese Liste abhaken und den Erfüllungsgrad seiner Wünsche messen. Korrektheit liegt vor, wenn der Erfüllungsgrad 100% ist. Fehlt allerdings ein Wunsch in der Spezifikation und ist deshalb nicht implementiert, ist das Verhalten der Software *nicht* inkorrekt, selbst wenn der Kunde bei der Überprüfung das Verhalten vermisst.

[3] *Effizienz* habe ich als Begriff für die nicht-funktionalen Anforderungen wie Performance, Skalierbarkeit, Benutzerfreundlichkeit, Sicherheit usw. gewählt, um sie kurz und knapp unter einer Kategorie zusammenzufassen. Die Benennung ist nicht perfekt, aber ich finde sie erstmal gut genug. Dem Kunden geht es um Funktionalität und Effizienz bei der Software. Klingt doch sinnig, oder?

It’s the productivity, stupid!

Über die Verhaltensanforderungen hinaus hat der Auftraggeber noch eine weitere Anforderung, die jedoch selten ausdrücklich formuliert oder gar vertraglich festgehalten wird. Das ist nun ein ganz wesentlicher Punkt; pass auf, denn es geht um dich und dein Team! Hier kommt die Motivation für Agilität und Clean Code Development:

- Die Software*entwicklung* soll stets *zügig* funktionale wie nichtfunktionale Anforderungen erfüllen. Auftraggeber haben also auch noch einen Anspruch an die **Produktivität** der Softwareentwicklung.

Verhaltensanforderungen werden unmittelbar durch Code erfüllt. **Die Produktivitätsanforderung hingegen ist eine an die herstellende Organisation**.

Wie Funktionalität mittels Code hergestellt wird, ist eine Sache von Programmiersprachen, Bibliotheken und Frameworks. Diese Fähigkeit ist die primäre, die du als Softwareentwickelnde(r) erwirbst und stetig verfeinerst.

Wie Effizienzen mittels Code hergestellt werden, ist ebenfalls zunächst eine Sache von Programmiersprachen, Bibliotheken und Frameworks. Diese Fähigkeit wird gewöhnlich später erworben, ist letztlich jedoch die, auf die sich viele Entwickelnde konzentrieren. Bücher wie “Algorithmen und Datenstrukturen” beschäftigen sich mit diesem Thema.

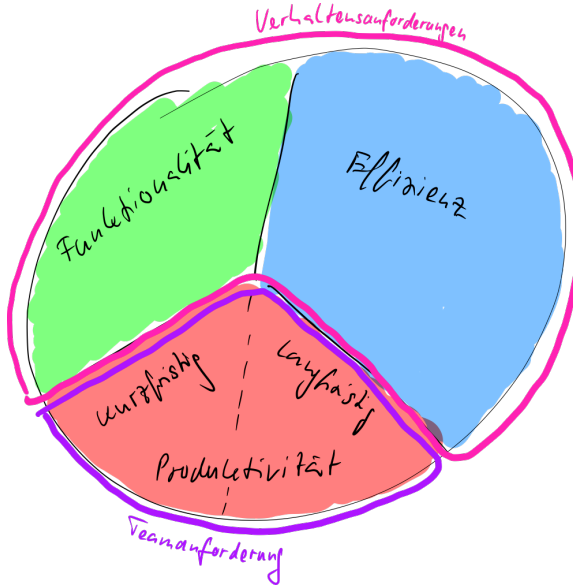
Nicht immer jedoch lässt sich damit das geforderte Qualitätsniveau schon erreichen. Performance oder Skalierbarkeit brauchen oft Unterstützung durch Verteilung von Code zur Laufzeit auf verschiedene Threads im selben Betriebssystemprozess oder in verschiedenen oder gar auf mehreren Rechnern oder in unterschiedlichen Netzwerken. Damit beschäftigt sich traditionell die Softwarearchitektur. Hier warten große Herausforderungen! Hier kannst du der Held so mancher Infrastrukturtechnologie werden.

Doch selbst wenn du gut dabei bist in der Herstellung von Funktionalität und Effizienz, kann es leicht sein, dass der Auftraggeber nicht mit dir zufrieden ist. Wie kann das sein? Du bist vielleicht einfach zu langsam.

Perfekte Verhaltensqualitäten lieferst du, nur leider zu spät. Potenziert wird das, wenn du auch noch unzuverlässig bist, d.h. die Lieferung bis zu einer Frist versprichst und dann doch nicht lieferst.

Für den Auftraggeber gibt es also zwei "Laufzeiten": die Software-Laufzeit und die Team-Laufzeit. An *beide* hat er Anforderungen. Die Software soll performen, das Team aber auch. Letzteres setzt der Auftraggeber allerdings mehr oder weniger voraus. Dafür schreibt er keine Spezifikation. Er glaubt einfach, dass du professionell arbeitest. Dazu gehört für ihn, dass du stets "flott dabei bist" und dir kein Bein stellst. Leider ist das oft nicht der Fall. Softwareentwicklung fällt immer wieder über die eigenen Füße; sie merkt sozusagen nicht, dass sie mit zusammengebundenen Schnürsenkeln läuft.

Aber wie kann das sein? Ich denke, dafür gibt es viele Gründe. Neben historischen, sozusagen systemimmanenten gibt es jedoch einen immer wieder ganz akuten: Druck. Die Softwareentwicklung wird vom Auftraggeber oft sehr mit Deadlines unter Druck gesetzt (und lässt sich auch unter Druck setzen), so dass sie meint, nie Zeit zu haben, die Schnürsenkel ordentlich zu binden. Lieber stolpert sie dahin, stets willig, dem Kunden Verhaltensanforderungen grob zu erfüllen, als dass sie sich "sauber aufstellt" und "fit hält".



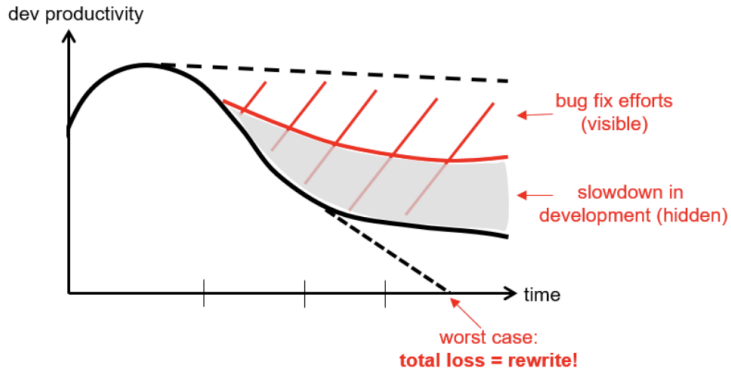
Was der Auftraggeber will: Die Kategorien der Anforderungen

Produktivitätskiller

Der Auftraggeber der Softwareentwicklung schaut gewöhnlich vor allem auf die Erfüllung von Verhaltensanforderungen. Das ist für ihn am einfachsten. Das merkst du jedes Mal, wenn Abnahme ist. Darum drehen sich dann die Diskussionen. Über den Herstellungsprozess, wie es zum präsentierten Verhalten gekommen ist, wird nicht diskutiert. Jedenfalls nicht direkt. Dafür fehlt ja eine Spezifikation. Was aber eben nicht heißt, dass der Kunde zur Team-Performance keine Meinung hätte.

Hohe Produktivität von dir und deinem Team wird einfach vorausgesetzt. Wie die Erfahrung jedoch zeigt, ist es eine naive Erwartung, dass hohe Produktivität nach einem vielleicht anfänglich guten Start "einfach so" erhalten bliebe. Die Produktivitätskurve sink vielmehr relativ schnell auf einen bedauerlich niedrigen Wert. Hier eine typische Darstellung der Entwicklung (Quelle[4]):

[4] https://blogs.sap.com/2018/05/02/introducing-agile-software-engineering-in-development/



Produktiv sind Entwickelnde nicht einfach, weil sie gerade codieren. Nur weil du dich gestresst fühlst beim Programmieren, performst du nicht automatisch im Sinne des Auftraggebers. Das mag enttäuschend klingen, ist aber die Realität. Solange es da ein Missverständnis zwischen dir und dem Auftraggeber gibt, sind Konflikte unvermeidlich.

Nicht jede geschriebene/veränderte Codezeile trägt zur Produktivität bei, wie der Auftraggeber sie sich wünscht. **Produktiv ist die Softwareentwicklung nur, wenn sie *neue* Anforderungen erfüllt, d.h. an *Features* arbeitet.** Das kann durch Codierung geschehen oder durch andere, vorgelagerte Tätigkeiten.

Je öfter du Features lieferst, d.h. Erweiterungen, Verbesserungen - keine **Bug Fixes** (!) - und die auch noch **korrekt** lieferst, desto **produktiver** bist du aus Sicht des Auftraggebers.

Wenn du also auch die (unausgesprochenen) Anforderungen des Auftraggebers an deine Produktivität erfüllen willst, tust du gut daran, alles was dabei hinderlich sein könnte, zu vermeiden. Wenn du während des Kochens eines Abendessens merkst, dass dir eine Zutat fehlt und du losrennst, um sie zu kaufen, bricht deine Produktivität ja auch ein. Dito, wenn du mit dem Kochen beginnen willst und findest die Spüle voll mit dreckigen Töpfen. Dito, wenn du dich zum Date fertigmachen willst und feststellen musst, dass deine beste Hose noch in der Wäsche ist. Wann immer also etwas fehlt, das du brauchst, um zu tun, was du eigentlich tun willst, stehst du einem Produktivitätskiller gegenüber.

Vorausgesetzt, dass du technisch und fachlich kompetent bist - auch daran

hat ein Auftraggeber Interesse -, sehe ich vor allem drei Produktivitätskiller, die du ausschalten musst:

Fehlende Korrekheit

Die Softwareentwicklung kann sehr geschäftig codieren, ohne produktiv zu sein. Das ist immer der Fall, wenn sie **Bug Fixing** betreibt.

Bugs sind Inkorrektheiten, d.h. Qualitätsmängel durch Nichterfüllung der Spezifikation.

Bugs zu fixen ist Nacharbeit (*re-work*). Nacharbeit oder Ausbesserung von Defekten ist eine der Verschwendungsarten in der Lean "Philosophie"[5]. Aus Sicht des Kunden vertust du deine Zeit mit Dingen, die schon lange hätten erledigt sein sollen. Statt Bugs zu fixen, wäre es dem Auftraggeber lieber, dass du schon wieder an neuem Verhalten arbeitest.

Jede Stunde, die du mit Bug Fixing verbringst, fehlt dir für die Feature-Produktion. Das Bug Fixing zu begrenzen, selbst wenn noch Bugs bekannt sind, ist daher eine notwendige Maßnahme, um produktiv zu bleiben[6]. Besser jedoch, wenn die Softwareentwicklung gar nicht erst in diese Verlegenheit kommt. Warum nicht Bugs von vornherein einfach vermeiden?

Fehlende Korrektheit ist der Produktivitätskiller #1.

Um die Produktivitätsanforderung des Kunden zu erfüllen, muss Korrektheit die oberste Priorität haben.[^klarheitsprämisse]

[^klarheitsprämisse]: Prämisse hierbei ist, dass klar ist, welches Verhalten die Software überhaupt haben soll. Korrektheit meine ich nur auf das, was klar spezifiziert ist. Wo Klarheit fehlt - allemal unwissentlich -,

[5] http://www.lean-production-expert.de/lean-production/7-verschwendungsarten.html

[6] Siehe dazu z.B. Zero-Bug Software Development

sind überraschende Qualitätsmängel unvermeidbar. Das sind dann jedoch keine Inkorrektheiten.

Korrektheit ergibt sich allerdings nicht einfach, sondern muss systematisch hergestellt und erhalten werden.

- Zunächst ist bei der Feature-Produktion (und auch beim Bug Fixing) Korrektheit in Form von **Reife** zu erreichen. Zu jedem Zeitpunkt bzw. spätestens vor Präsentation/Auslieferung eines Softwarestandes musst du prüfen, ob deine Software *schon* korrekt ist gem. der Spezifikation. Haben deine Anstrengungen zur Herstellung gewünschter Qualitäten schon ausreichenden Erfolg gehabt? Wenn du keine Differenz mehr siehst zwischen spezifiziertem und realem Verhalten, dann ist dein Code reif für die Präsentation beim Auftraggeber.
- Darüber hinaus ist allerdings stets sicherzustellen, dass bei der Feature-Produktion vorher erreichte Korrektheit nicht zerstört wird. Es darf keine Regression stattfinden, d.h. kein Rückfall auf ein früheres, niedrigeres Korrektheitsniveau. Der Auftraggeber erwartet **Stabilität** der Software in Bezug auf die Korrektheit. Während der Veränderung von Code bzw. spätestens vor Präsentation/Auslieferung eines Softwarestandes musst du deshalb immer wieder überprüfen, ob deine Software *noch* korrekt ist gem. der Spezifikation. **"Verschlimmbesserung" ist eines der größten Risiken in der Softwareentwicklung**.

Maßnahmen für die Korrektheit umfassen z.B. den Abnahmetest, eine Beta-Test-Phase, die Beschäftigung von Testern, die Definition eines *Done*-Zustands inkl. Akzeptanzkriterien, automatisierte Tests, eine Continuous Build/Integration Pipeline oder die Codierung nach Test-Driven Development (TDD).

Produktivität braucht Sorgfalt. Es sind "die Dinge richtig zu tun". So wird landläufig auch *Effizienz* beschrieben. Man weiß, was zu tun ist - und tut es dann auch so, wie es getan werden sollte. Die Verhaltensanforderungen sind klar, die Softwareentwickelnden sind kompetent, das Ergebnis ist korrekte Software. So sollte es zumindest sein. Das ist die Erwartung des Auftraggebers. Doch so einfach ist es nicht…

Überlege selbst, welche der obigen (oder auch weiteren) Maßnahmen in deinem Team verlässlich getroffen werden, um hohe Korrektheit zu liefern und zu erhalten.

Fehlender Wert

Aber was, wenn die Softwareentwicklung nicht weiß, was zu tun ist? Was, wenn Unklarheit herrscht? Die Voraussetzung dafür, "die Dinge richtig zu tun" ist, dass man überhaupt "die richtigen Dinge tut". So wird landläufig *Effektivität* beschrieben. Effektivität kann es nur geben, wenn Klarheit herrscht.

Solange die Softwareentwicklung aber im Unklaren darüber ist, was genau die Verhaltensanforderung ist oder solange der Auftraggeber selbst sich noch nicht ganz klar darüber ist, wie für ihn hohe Verhaltensqualität aussieht, kann Codeproduktion nicht effektiv sein. Und ohne Effektivität keine Produktivität.

Leider ist das der natürliche Zustand von Softwareprojekten:

- Der Auftraggeber hat eine nur unvollständige Vorstellung davon, was er braucht.
- Der Auftraggeber kann seine Vorstellungen nur unvollständig formulieren.
- Die Softwareentwicklung versteht die formulierten Anforderungen nur unvollständig.
- Die Softwareentwicklung setzt ihr Verständnis der Anforderungen nur unvollständig um.
- Der Auftraggeber hat in der Zeit von der Spezifikation bis zur Abnahme ihrer Umsetzung[7] seine Meinung geändert; seine Anforderungen sehen nun anders aus. Selbst eine korrekte Umsetzung

[7] Der Begriff Spezifikation mag sich für dich hier schwergewichtig anhören. Wo bleibt da die Agilität? Aber ich meine ihn ganz neutral. Er soll einfach nur ausdrücken, dass ein Auftraggeber in unmissverständlicher Weise irgendwie beschrieben hat, welche Verhaltensanforderungen der Code, den du entwickelst, erfüllen soll. "Irgendwie" bedeutet, dass ich nicht suggerieren will, in welcher Sprache, mit welchem Medium, in welchem Umfang eine Spezifikation vorliegt. Ebensowenig will ich mitschwingen lassen, wie häufig der Auftraggeber eine Spezifikation vorlegt oder ihre Umsetzung prüfen will; das kann alle paar Wochen sein oder jeden Tag. Iterativ-inkrementelles Vorgehen ist für mich mit dem Begriff also absolut vereinbar.

> der ursprünglichen Spezifikation passt daher nur unvollständig zum neuen Stand der Bedürfnisse des Auftraggebers.

Das ist die Erkenntnis der **Agilität** in der 1990ern gewesen, die zur Mindestforderung eines **iterativ-inkrementellen Softwareentwicklungsprozesses** geführt hat.

Als Produktivitätskiller hatte sich herausgestellt, dass immer wieder *überraschend* bei der Abnahme von Software nicht der erwartete *Wert* geliefert wurde. Selbst spezifikationsgemäße Lieferung hatte nicht die im praktischen Einsatz erforderlichen Nutzen.[8]

Das Missverständnis von Auftraggebern und Softwareentwicklung bis in die 1990er war (und ist leider auch heute noch in einigen Projekten), dass Verhaltensanforderungen sich in einem mehr oder weniger länglichen Prozess einmalig vor Beginn der Umsetzung festzurren lassen könnten (Stichwort "Wasserfall").

Diese Vorstellung hat zu Spezifikationen geführt, die große, unvermutete Missweisungen enthielten, die in Software gegossen große negative Überraschungen ausgelöst haben. Umfangreiche Nacharbeiten waren nötig, nicht wegen Inkorrektheit, sondern wegen Wertlosigkeit. Auch korrekt implementierte Spezifikationen haben zum Lieferzeitpunkt nichts oder zu wenig genützt.

Dem hat die Agilität eine Desillusionierung entgegen gesetzt. Nicht noch bessere, umfangreichere, längere Anforderungsanalyse soll die Produktivität steigern, sondern das Gegenteil: eine radikale Verkürzung bei gleichzeitiger Vervielfachung von Analyse, Spezifikation und Umsetzung.

In der Agilität gibt es weiterhin eine Spezifikation und insofern eine Erwartung an hohe Korrektheit (Stichwort "Definition of Done"). Doch es wird nicht mehr angenommen, dass diese Spezifikation schon "die letzte Wahrheit" sei. Stattdessen soll die Softwareentwicklung bestrebt sein, nur schmale Ausschnitte eines Gesamtverhaltens zu spezifizieren (auch **Inkremente** genannt), die zügig umgesetzt werden können, um

[8] Wert ist also nicht einfach gleich hochqualitative Software. Zum Wert gehört natürlich, dass Software hohe Qualität hat, d.h. der Spezifikation möglichst genau entspricht. Darüber hinaus muss diese hohe Qualität allerdings auch noch nützlich sein in dem Moment, wo sie geliefert wird. Daraus ergibt sich, dass Qualitätsproduktion und Wertproduktion zwei zu unterscheidende Prozesse braucht. Für Ersteren bist du als Softwareentwickler zuständig, für Letzeren z.B. in Scrum aber der Product Owner!

vom Auftraggeber Feedback zu bekommen. Wert kann man sich nur schrittweise annähern, nicht, weil Auftraggeber oder Softwareentwicklung inkompetent sind, sondern weil es in der Natur der Sache komplexer Anforderungen liegt; da ist kein geradliniger Weg zu hohem Wert sichtbar.

Man bekämpft beim iterativ-inkrementellen Vorgehen die Ineffektivität dadurch, dass man ihr den Zahn der Überraschung zieht. Denn nur die Überraschung macht aus mangelndem Wert frustrierende Nacharbeit. Ist mangelnder Wert jedoch zu erwarten, ja, geradezu die Norm, dann ist die nächste Iteration keine Nacharbeit, keine Verschwendung, sondern ein erwartetes Inkrement und insofern produktiv - auch wenn man gern schneller vorangehen würde.

Softwareentwicklung wie Auftraggeber hegen beim agilem Vorgehen nicht mehr den Glauben, dass wertvolle Software "in einem Rutsch" entstehen kann. Vielmehr muss man sich hohem Wert experimentierend mit hochqualitativen Inkrementen annähern. Das ist keine Last, das ist eine Tugend, weil unvermeidbar.

Wie steht es mit diesem Verständnis in deinem Team? Geht ihr iterativ-inkrementell vor? Versteht der Auftraggeber die Vorläufigkeit seiner Anforderungen und eurer Lösungen?

Fehlende Ordnung

Auch wenn fehlende Korrektheit der naheliegende und greifbare Produktivitätskiller ist, ging ihm geschichtlich fehlender Wert in der Bewusstwerdung der Softwareentwicklung voraus, denke ich.

Nicht genau zu wissen, was der Auftraggeber wirklich will, was für ihn Wert darstellt, für das Geld, das er auszugeben bereit ist, war zunächst ein größeres Problem. Erst als eine Verbesserung des Vorgehensmodells in den 1990ern hier mehr Klarheit gebracht hatte und dadurch die Zahl der Softwarelieferungen zur Feedback-Generierung, die potenziell inkorrekt sein konnten, gestiegen war, trat der Produktivitätskiller Inkorrektheit deutlich(er) zutage. Beleg ist aus meiner Sicht dafür die späte Erfindung automatisierter Tests. Erst Ende der 1990er bekam das Thema breite Sichtbarkeit.

Wenn man weiß, was das Richtige ist (Wert), lohnt es, das auch richtig zu tun (Korrektheit). Wenn man es kann.

Und da steckt schließlich der dritte Produktivitätskiller, den ich dir vorstellen möchte: die Unordnung. **Solange du nicht bewusst darauf achtest, Ordnung im Code herzustellen, hast du es immer schwer, das Richtige auch richtig zu tun.**

Für *langfristig* hohe Produktivität muss das Richtige ordentlich richtig getan werden.

Code, der sich mit jedem neuen Feature, mit jedem Bug Fix weniger leicht verändern lässt, wird zum Morast, in dem deine Softwareentwicklung alsbald steckenbleibt. Oder wenn nicht steckenbleibt, dann zumindest nur noch schwerfällig vorankommt. Das ist nicht, was Auftraggeber sich wünschen.

Code ist eine Ressource, mit und an der Softwareentwicklung arbeitet. Wie andere Ressourcen kann sie pfleglich behandelt werden - oder man treibt an ihr Raubbau. Ohne weitere Maßnahmen geschieht Letzteres.

Für den Auftraggeber sind Inkorrektheit und Wertarmut von Code noch vergleichsweise leicht zu spüren. Beide zeigen sich als mangelnde Qualitäten im Verhalten.

Unordnung jedoch entzieht sich der direkten und zeitnahen Wahrnehmung des Auftraggebers. Deshalb hat sie Gelegenheit, sich hinter der Fassade des Verhaltens aufzubauen. Wenn sie dann indirekt über deutlich sinkende Produktivität auch für den Auftraggeber spürbar wird, ist es jedoch eigentlich schon zu spät. Deshalb musst du ständig ein Auge auf die Ordnung haben!

Wenn du die Ordnung zu lange hast schleifen lassen, sind die nötigen "Aufräumarbeiten" meist zu umfangreich, als dass sie sich rechnen würden. Und sie ließen sich auch kaum dem Kunden gegenüber verheimlichen. Also schleppt sich die Softwareentwicklung weiter durch den selbst verschuldeten Sumpf. Denn selbst verschuldet ist er, da der Kunde sich Unordnung nicht gewünscht hat. Sie ist mangels Bewusstsein und/oder mangels Fähigkeit und/oder wider besseren Wissens "auf Befehl" (Ignoranz) und/oder in naivem Glauben an baldige Korrektur (so genannte *Technische Schuld*) entstanden.

Nicht, dass fehlende Ordnung eine neue Ursache für Produktivitätsabnahme wäre. Sie wurde schon in den 1960ern oder gar früher identifiziert. Auch die Strukturierte Programmierung (structured programming[9]) ist aus dieser Erkenntnis entstanden. Man könnte wohl auch sagen, dass Objektorientierung von ihr ursprünglich inspiriert war. Ebenso das structured design[10] und der Begriff des Moduls.

Wer mit Code zu tun hat, erwartet, ordentlichen Code vorzufinden, d.h. Code, der nicht unnötig behindert, ihn zu verstehen ("easy to reason about"), und der nicht unnötig behindert, ihn zu verändern. Denn darum geht es letztlich ja immer: Code wird nur betrachtet, um ihn neuen Anforderungen anzupassen oder zu korrigieren. Dass du Code aus Spaß am prasselnden Kaminfeuer studierst, passiert wahrscheinlich selten, oder?

Nach Jahrzehnten des mehr oder weniger latenten Bewusstseins der Branche, dass Ordnung eine Qualität ist, auf die es ebenfalls zu achten gilt bei der Softwareentwicklung, hat dann im Jahr 2008 der Begriff **Clean Code** dem Thema neue Sichtbarkeit und Konkretheit gegeben.

Dass Robert C. Martin von *sauberem* Code und nicht von *ordentlichem* spricht, mag dem von Martin Fowler im Zusammenhang mit dessen Buch *Refactoring* geprägten Begriff *code smell* geschuldet sein. Was sauber ist, riecht nicht.

Doch letztlich ist Sauberkeit als Bild zu schwach für die nötige Eigenschaft, die Code haben muss, um deine Softwareentwicklung nicht schwerfällig zu machen. Was sauber ist, kann immer noch unordentlich, d.h. unüberschaubar bis zu Unbrauchbarkeit sein.

Wenn du dir jetzt allerdings Ordnung vorstellst, denkst du sehr wahrscheinlich nicht nur an Sauberkeit als Selbstzweck, sondern auch noch an Eignung für weitere Nutzung. Sauberkeit schützt vor Schaden.

Ordnung hat als Zweck Befähigung!

[9] https://en.wikipedia.org/wiki/Structured_programming

[10] https://www.amazon.de/Structured-Design-Fundamentals-Discipline-Programme/dp/0138544719

Und genau darum geht es, wenn ich hier von ordentlichem Code, von Ordnung im Code spreche. Code soll ordentlich sein, *um zu* zügiger Veränderung zu *befähigen*.

Zusammenfassung

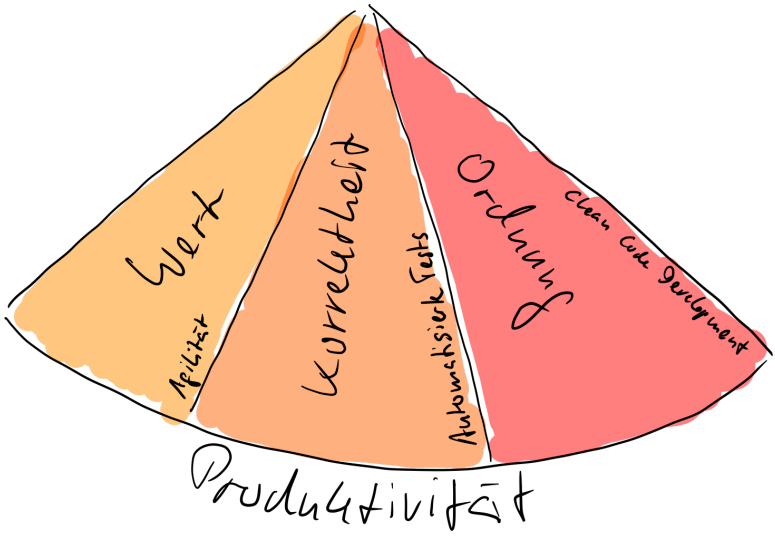
Auftraggeber wollen Software, die umfassend tut, was sie tun soll; sie soll funktional und effizient sein. Diese Qualitäten sollst du in der Softwareentwicklung stets zügig liefern; du sollst produktiv sein *und bleiben*.

Unglücklicherweise ist schon die Herstellung von funktionalem und effizientem Code eine Sache, die sehr komplex ist. Ich denke, davon kannst du ein Lied singen. Sich mit all den Technologien und Produkten und Ansätzen auszukennen, die zur Herstellung von funktionalem und effizientem Code zur Verfügung stehen, ist eine Kunst für sich.

Und nun soll die Herstellung von Code, der hochqualitatives Verhalten zeigt, auch noch stets zügig stattfinden, obwohl dieser Code ständigen Änderungen unterliegt und die Anforderungen an ihn notorisch unklar sind? Das steigert die Komplexität der Softwareentwicklung erheblich!

Wie es für die Herstellung von Verhaltensanforderungen Werkzeuge gibt, so gibt es zum Glück aber auch Werkzeuge, die dir helfen, hohe Produktivität zu produzieren.

- Agilität
- Automatisierte Tests
- Prinzipien und Praktiken des Clean Code Development



Voraussetzungen für Produktivität

Du musst diese Werkzeuge kennen und auch einsetzen. Sie sind nicht neu, sie sind womöglich noch nicht einmal schwierig zu beherrschen - doch sie haben eines gemeinsam: sie gehen ans Eingemachte. Damit du sie konsequent benutzt, musst du eine passende Grundhaltung entwickeln; die Kultur der Softwareentwicklung in deinem Team und darüber hinaus muss darauf ausgerichtet sein. Das braucht Zeit.

Die Agilität hat es inszwischen geschafft, breit ins Bewusstsein (oder zumindest auf die Lippen) der Branche zu dringen. Auf die eine oder andere Weise wird also in vielen Softwareentwicklungsteams versucht, die Wertproduktion hoch zu halten durch iterativ-inkrementelles Vorgehen.

Mit der Korrektheit und der Ordnung hingegen, steht es weniger gut. Das liegt daran, dass das eine vom anderen abhängig ist: ohne Ordnung ist es schwer, Korrektheit zuverlässig und nachvollziehbar herzustellen und zu überprüfen. Aber gerade die Ordnung hat es in sich. Nicht umsonst ist sie geschichtlich der letzte Produktivitätskiller, für den breites Bewusstsein geschaffen werden musste.

So verständlich es war, dass der Fokus von der Steigerung der Produktivität in Bezug auf den **Wert** von Software (in den 1990er Jahren) zur Steigerung der Produktivität durch Erhöhung der **Korrektheit** (in den

2000er Jahren) gewandert ist - so ist es andererseits auch verständlich, dass die Produktivitätssteigerung dann gegen eine gläserne Decke stoßen musste. Erst durch einen weiteren Wechsel des Fokus hin zur **Ordnung** (in den 2010er Jahren) kann nämlich die Behinderung aus dem Weg geräumt werden, die mehr Korrektheit und auch zügigerer Wertherstellung im Wege stand.

Dauerhaft hohe Produktivität braucht…

- eine **Organisation**, die ihr höchste Priorität zuweist, um langfristig wettbewerbsfähig zu bleiben,
- ein Verständnis dafür, was **Ordnung** im Code bedeutet, wie er stets wandlungsfähig gehalten werden kann,
- den Willen zur Produktion von **stabil korrektem Code**, um die Kapazität für die Erweiterung von Code maximal zu halten,
- den Mut, nur auf der Basis von **unzweideutigen Spezifikationen** Code zu produzieren
- und schließlich die Einsicht, dass Unklarheit und Volatilität ständige Begleiter der Softwareentwicklung sein werden, so dass **Vorläufigkeit** auf allen Ebenen akzeptiert werden muss.

Leider ist es in der Softwareentwicklung so, wie es der Buddha für das Leben konstatiert hat:

> *"Frische Milch braucht Zeit zum Sauerwerden, / Unheilsames Handeln braucht Zeit zum Reifen, / So schwelen im Toren die Folgen seines Handelns, / Wie unter der Asche verborgene glühende Kohlen.", Dhammapada - Die Weisheitslehren des Buddha, Munish B. Schiekel*

Die negativen Auswirkungen deines heutigen Handelns zeigen sich nicht immer sofort. Sie wachsen unsichtbar und schleichend an – bis du sie irgendwann und oft zu spät deutlich spürst.

Deshalb ist es wichtig, **die Produktivität als im Grunde höchstes Gut, als wichtigste Anforderung zu verstehen und die Softwareentwicklung dahingehend zu organisieren. Das ist nachhaltige Softwareentwicklung.** Zuerst und unverbrüchlich soll Produktivität geliefert werden, dann erst Funktionalität, dann Effizienz.

Maßnahmen zur korrekten Wertproduktion in Ordnung müssen einen Rahmen aufspannen, in dem konkrete funktionale und nicht-funktionale Anforderungen umgesetzt werden. Derzeit geschieht es vielfach noch umgekehrt: Verhaltensanforderungen werden "irgendwie" realisiert und insbesondere Korrektheit und Ordnung sind nachrangig.

Alles, was ich dir im Folgenden präsentiere, darfst du vor diesem Hintergrund verstehen. Ich möchte dir ein methodisches Rahmenwerk vorstellen, mit dem du systematisch für höhere Korrektheit und Ordnung in deinem Code sorgen kannst. Mich treibt die eigene leidvolle Erfahrung an, dass darauf einfach zu wenig und zu spät geachtet wird. Mir ist das früher auch oft passiert - weil ich es nicht besser wusste. Dir möchte ich diese Erfahrung ersparen. Dir möchte ich eine Guideline an die Hand geben, mit der du während der Codierung deinen Weg zur Korrektheit und Ordnung findest. Keine Angst mehr vor dem blinkenden Cursor, der dich auffordert, vor allem Funktionalität zu produzieren. Mit ein bisschen System, gutem Willen und Übung wirst du es schaffen, Funktionalität *und* dauerhaft hohe (oder zumindest höhere) Produktivität herzustellen.

Die Methode

01 - Die Anforderung-Logik Lücke

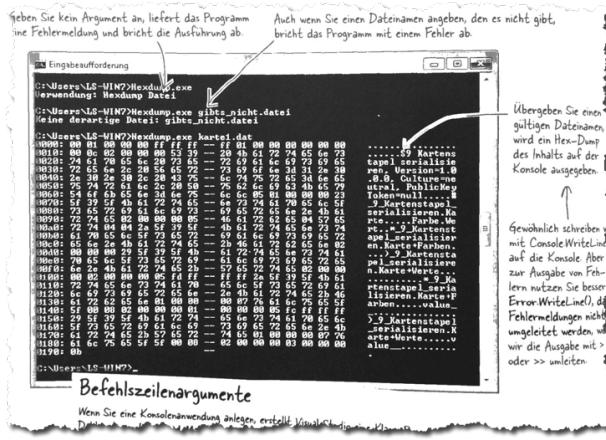
Um die Softwareentwicklung vom Kopf auf die Füße zu stellen, d.h. ihr einen Rahmen für Nachhaltigkeit zu geben, ist es hilfreich, wenn wir ihr Produkt genauer betrachten. Woraus bestehen "die Maschinen", die du in der Softwareentwicklung produzierst, von denen sich der Auftraggeber so viel Hilfe verspricht?

Logik - Der Stoff aus dem Verhalten entsteht

Die offensichtlichen Anforderungen des Auftraggebers sind die Verhaltensanforderungen an Software. Verhalten wird durch Code hergestellt - aber nicht der gesamte Code ist dafür verantwortlich.

Hier als Beispiel eine Software, die eine Datei als Hex Dump ausgeben soll wie in diesem Bild dargestellt (Quelle: C# von Kopf bis Fuß[11]):

[11] https://www.amazon.de/gp/product/B06XDVW33W



Der C#-Code dafür sieht im Ausschnitt so aus:

```
using System;
using System.IO;
using System.Text;

namespace hexdump
{
  // source: "C# von Kopf bis Fuß"
  class MainClass
  {
    public static void Main (string[] args)
    {
      if (args.Length != 1) {
        Console.Error.WriteLine ("Usage: hexdump <dateiname>");
        Environment.Exit (1);
      }

      if (!File.Exists(args[0])) {
        Console.Error.WriteLine("No such file: {0}", args[0]);
        Environment.Exit(2);
      }

      using (var input = File.OpenRead (args [0])) {
        int position = 0;
        var buffer = new byte[16];

        while (position < input.Length) {
          var charsRead = input.Read (buffer, 0, buffer.Length);
          if (charsRead > 0) {
            Console.Write ("{0}: ", string.Format ("{0:x4}", position));
            position += charsRead;

            for (int i = 0; i < 16; i++) {
              if (i < charsRead) {
                var hex = string.Format ("{0:x2}", buffer [i]);
                Console.Write (hex + " ");
              } else {
                Console.Write ("   ");
              }
    ...
```

Erkennst du, welche Zeilen des Code verhaltensrelevant sind? Die Veränderung welcher Zeilen würde für einen Anwender unmittelbar spürbar

sein?

Könnte `using System` gelöscht werden, ohne dass sich das Programmverhalten ändert?[12] Nein, das Programmverhalten würde sich nicht ändern.

Sind die Leerzeilen oder der Kommentar relevant für das Programmverhalten? Nein.

Spürt ein Anwender, ob es die Funktion `Main()` gibt? Nein.[13]

Aber wenn eine Zeile mit `Console.Error.WriteLine(...)` fehlen würde, dann würde der Anwender das (in manchen Fällen) bemerken.

Oder wenn die Zeile `if (i < charsRead)` fehlen würde oder darin das < durch ein > ersetzt würde, dann würde das zu einem anderen Verhalten des Programms führen.

Code ist also nicht gleich Code. Mancher Code/manche Codezeilen sind für das Verhalten relevant, manche nicht.

Die für das Verhalten relevanten Codezeilen stellen die **Logik** von Software dar.[14]

Logik besteht aus

- Transformationen/Operatoren, z.B. `<`, `++`, `args.Length`
- Kontrollstrukturen, z.B. `if-else`, `for`, `try-catch`
- I/O- bzw. allgemeiner API-Calls, z.B. `Console.Write()`, `File.OpenRead()`

[12] Vorausgesetzt die dadurch syntaktischen/semantischen Probleme im Quellcode würden durch weitere Eingriffe kompensiert.

[13] Dass `Main()` in C# nötig ist, um ein Programm ausführbar zu machen, ist unwesentlich. In anderen Programmiersprachen sind keine Klassen wie `MainClass` und keine Methode wie `Main()` nötig, um ein Programm (übersetzen und) laufen zu lassen.

[14] So nenne ich diesen Teil des Code jedenfalls im Weiteren, weil ich keinen anderen Namen dafür kenne. Wenn du einen besseren weißt oder einen schon etablierten, dann lass ihn mich wissen. *Statements* finde ich zu wenig, weil damit im Grunde alles gemeint ist, was in C# (und anderen Sprachen) mit einem `;` abgeschlossen wird. Dazu gehört, wie du bald lesen wirst, aber auch Code, der keine Logik ist.

Wenn nun das für Auftraggeber so wichtige Verhalten - Funktionalität + Effizienz - nur durch Logik hergestellt wird, stellt sich die Frage, was der übrige Code für Zweck hat. Welche Anforderungen hilft er erfüllen? Warum solltest du irgendetwas anderes codieren als Logik?

Nicht-Logik Code dient der Herstellung von Produktivität.

Einige Beispiele:

- Namespaces reduzieren das Rauschen im Code, das lange Namen mit redundanten Anteilen verursachen. Sie erhöhen die Ordnung.
- Leerzeilen strukturieren den Code vertikal, indem sie unterschiedliche inhaltliche Kohäsion anzeigen. Sie erhöhen die Ordnung.
- Funktionen "komponieren" Logik zu Funktionseinheiten, die Aspekte eines Verhaltens unter einem Namen zusammenfassen. Sie erhöhen die Ordnung und die Testbarkeit.
- Klassen aggregieren Funktionen (und Daten) und stellen damit zweckvolle Einheiten zusammen. Sie erhöhen die Ordnung.

Funktionalität

Die erste Kunst bei der Herstellung (oder Entwicklung) von Logik ist, sie so zu wählen, dass sie die gewünschte Funktionalität hat. Das lernst du auf alle Fälle in jedem Buch einer Programmiersprache oder einem Programmierkurs.

Logik, die die Zahlen in einem Array summiert, sieht dann z.B. so aus:

```
1  static int Sum(int[] numbers) {
2    var sum = 0;
3    foreach(var n in numbers)
4      sum += n;
5    return sum;
6  }
```

Logik, die die Zahlen in einem Array sortiert, sieht hingegen z.B. so aus:

```
// Quelle: https://www.geeksforgeeks.org/bubble-sort/
static void BubbleSort(int []arr)
{
  int n = arr.Length;
  for (int i = 0; i < n - 1; i++)
    for (int j = 0; j < n - i - 1; j++)
      if (arr[j] > arr[j + 1])
      {
        int temp = arr[j];
        arr[j] = arr[j + 1];
        arr[j + 1] = temp;
      }
}
```

Welche Logik-Bausteine du aus den von deiner Programmiersprachen, deinen Bibliotheken und Frameworks angebotenen auswählst und wie du sie in Beziehung setzt, macht den Unterschied, ob das eine oder das andere Verhalten entsteht.

Auch Code, der nur aus Logik besteht, hat insofern eine Struktur. Im BubbleSort-Beispiel ist die augenfällig durch die Schachtelung der Kontrollstrukturen.

Effizienz I - Effizienz durch Algorithmen und Datenstrukturen

Logik so zu strukturieren, dass sie die gewünschte Funktionalität hat, ist jedoch nicht alles. Sie soll auch z.B. performant sein. Logik über die Funktionalität hinaus auch noch mit Effizienzen auszustatten, ist die zweite Kunst, die du lernen musst, wenn du programmieren willst.

Hier ein Beispiel dafür, wie anders Logik aussehen kann, nur weil sie mehr Effizienz bieten soll. *Bubblesort* ist ein bekanntermaßen imperformanter Sortieralgorithmus. *Radixsort* soll diesen Makel beseitigen:[15]

[15]Die Funktionalität ist dieselbe, die Logik-Struktur ist aber eine völlig andere, weil andere Effizienzanforderungen erfüllt werden. Doch es kommt noch schlimmer: Sogar dieselbe Funktionalität mit denselben Effizienzanforderungen kann zu sehr unterschiedlicher Logik führen. Unter anderem das macht es dir so schwierig, aus Logik herauszulesen, welches Verhalten sie eigentlich erzeugt.

```
// Quelle: https://www.geeksforgeeks.org/radix-sort/
static void Radixsort(int[] arr, int n)
{
  int mx = arr[0];
  for (int i = 1; i < n; i++)
    if (arr[i] > mx)
      mx = arr[i];

  for (int exp = 1; mx/exp > 0; exp *= 10)
  {
    int[] output = new int[n];

    int i;
    int[] count = new int[10];

    for(i = 0; i < 10; i++)
      count[i] = 0;

    for (i = 0; i < n; i++)
      count[ (arr[i]/exp)%10 ]++;

    for (i = 1; i < 10; i++)
      count[i] += count[i - 1];

    for (i = n - 1; i >= 0; i--)
    {
      output[count[ (arr[i]/exp)%10 ] - 1] = arr[i];
      count[ (arr[i]/exp)%10 ]--;
    }

    for (i = 0; i < n; i++)
      arr[i] = output[i];
  }
}
```

Logik (und zugehörige Datenstrukturen) für Effizienz-Anforderungen passend zu wählen, erfordert also mehr als die Kenntnis von Logik-Bausteinen. Dass du dir z.B. der algorithmischen Komplexität[16] deiner Logik bewusst bist, gehört dazu, wenn du mit Logik den Auftraggeber umfassend erfreuen willst. Es kommt auf die Auswahl und Zusammenstellung der Logik-Bausteine an, auf ihre Komposition.

Effizienz II - Effizienz durch Verteilung

Performance und Skalierbarkeit oder auch andere Effizienzanforderungen lassen sich allerdings nicht immer allein durch Auswahl und Anordnug von Logik erfüllen. Dann ist zusätzlich **Verteilung** gefragt, d.h. die Ausführung von Logik verteilt auf mehrere Threads.

Als simples Beispiel mag die Sortierung von zwei Arrays dienen. Eine Lösung nur mit Logik kann das auch mit dem schnelleren Algorithmus nur sequenziell bewerkstelligen:

[16] https://www.bigocheatsheet.com/

```
Radixsort(arr1, arr1.Length);
Radixsort(arr2, arr2.Length);
```

Die Gesamtlaufzeit ist dann die Summe der Laufzeiten der einzelnen Aufrufe der Funktion, die die Sortierlogik kapselt.

Wenn die Sortierung jedoch parallel, d.h. auf zwei Threads (verschiedener Prozessorkerne) stattfinden kann…

```
var t1 = Task.Factory.StartNew(() => Radixsort(arr1, arr1.Length));
var t2 = Task.Factory.StartNew(() => Radixsort(arr2, arr2.Length));
Task.WaitAll(new[] {t1, t2});
```

…dann entspricht die Gesamtlaufzeit (ungefähr) nur der des Funktionsaufrufs, der länger gebraucht hat.

Logik mit mehr Effizienz auszustatten durch Verteilung ist traditionell ein Teil der Disziplin **Softwarearchitektur**. Sie kannst du als die dritte Kunst der Softwareentwicklung ansehen.

Hierarchie der Hosts

Softwarearchitektur verteilt Logik, indem sie sie in **Hosts** ausführt. So nenne ich geschachtelte Laufzeit-Kontexte/Container, die mit mehr oder weniger Infrastruktur aufgesetzt, betrieben und in Verbindung gebracht werden.

- **Thread**: Multithreading ist der erste Schritt, um Latenz zu verbergen oder zu verringern oder den Durchsatz zu erhöhen. Die Kommunikation schon zwischen Logik auf verschiedenen Threads ist aber nicht mehr direkt, d.h. langsamer als die zwischen Logik auf demselben Thread. Vorsicht ist geboten, wenn Threads auf die selben Daten zugreifen.
- **Process**: Logik parallel in verschiedenen Betriebssystemprozessen zu betreiben, entkoppelt sie stärker, was zur Robustheit beiträgt. Dass es keinen gemeinsamen Hauptspeicher mehr gibt, reduziert das Risiko von Fehlern. Allerdings ist die Kommunikation deutlich aufwändiger zwischen Prozessen.

- **Machine**: Logik in mehreren Threads verteilt auf mehrere Prozesse auf verschiedenen (physischen oder virtuellen) Maschinen auszuführen, ermöglicht ein scale-out oder auch die Ansiedelung von Logik näher an Ressourcen. Allerdings ist die Kommunikation zwischen Maschinen noch langsamer als zwischen Prozessen, so dass sehr auf Häufigkeit und Granularität der Nachrichtenübermittlung geachtet werden muss.
- **Network**: Logik auf Maschinen in verschiedenen Netzwerken zu verteilen, ist allemal unvermeidbar, wenn Speicher- und Prozessorressourcen flexibel genutzt werden sollen (Stichwort "Cloud Computing"). Der Nutzen bei der Skalierbarkeit ist mit den Gefahren für die Sicherheit abzuwägen. Und die Kommunikationsgeschwindigkeit sinkt abermals.

Effizienz durch Verteilung steigern zu müssen, ist oft unvermeidbar. Simpel ist das jedoch nicht. Die Zahl der hilfreichen Technologien nimmt jeden Tag zu und erfordert von dir ein fleißiges Studium, wenn du mithalten willst. Vorsicht ist dennoch weiterhin ganz grundsätzlich gegenüber den fallacies of distributed computing[17] geboten.

Im Weiteren spielen Hosts als Container für Logik jedoch keine größere Rolle mehr. Die Darstellungen hier drehen sich nicht um die Herstellung von Effizienzen, sondern vor allem um die Qualitäten Wert, Korrektheit und Ordnung für die Anforderung Produktivität. Du wirst es mit Strukturen zu tun bekommen, aber nur vergleichsweise wenigen Strukturen bestehend aus mehreren Hosts.

Zusammenfassung

Logik und ihre Verteilung ist das, was für den Auftraggeber unmittelbar spürbar ist. Mit Logik und Verteilung Verhalten herzustellen, sind die grundlegenden Künste der Programmierung. In ihnen können Softwareentwickelnde ständig reifen; für sie werden ständig neue Paradigmen, Technologien und Produkte entwickelt.

Logik und Verteilung in hoher Qualität herzustellen, ist auch bei guten Spezifikationen ein komplexes Unterfangen. Umso naheliegender sollte es sein, dass du diese Transformation systematisch betreibst.

[17] https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing

Von den Anforderungen zur Logik

Angesichts des großen, bewussten und verständlichen Bedarfs an Softwareverhalten, den Auftraggeber haben, ist es kein Wunder, dass sie großen Druck auf die Logik-Produktion ausüben. Du sollst möglichst schnell Features mit Logik umsetzen - alles andere ist dem Kunden wenn schon nicht egal, dann doch meistens nur wenig bewusst. Auf alles andere achtet er insofern wenig oder kann es sogar nicht einmal beurteilen.

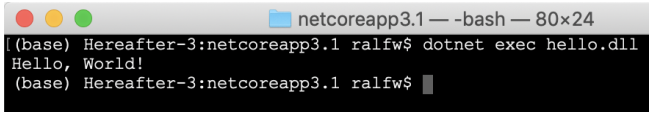
Logik schwer definierbar

Doch leider "ergibt sich" Logik nicht "einfach so". Sie liegt nicht auf der Hand. Funktionale und effiziente Logik zu finden, ist für dich auch mit viel Erfahrung eine komplexe Angelegenheit. Schon eine sehr simple Aufgabe macht das deutlich:

Iteration 1: Hello, World!

Schreibe ein Programm, dass auf der Console "Hello, World!" ausgibt.

Das Ergebnis soll von der Ausgabe her so aussehen:



Eine Beispielausgabe als Spezifikation

Welche *Logik* ist dafür nötig?

Diese Frage wirst du für deine Programmiersprache sicher aus dem Stand beantworten können. In C# sieht sie so aus:[18]

[18]Für weitere 599 Programmiersprachen kannst du dir hier die Antworten anschauen. Aber Achtung: Viele enthalten nicht nur Logik, sondern auch sprachnotwendiges "Rauschen" drumherum.

```
1  Console.WriteLine("Hello, World!");
```

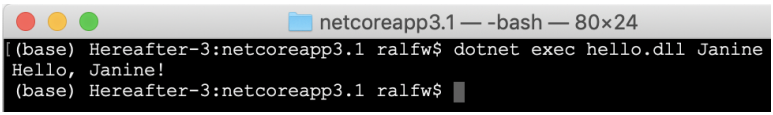
Das Programm selbst ist umfangreicher, weil noch eine Klasse und eine Funktion "als Verpackung" erforderlich sind, aber die reine Logik ist so trivial.

Auf zur nächsten Iteration:

> **Iteration 2: Persönliche Begrüßung**
>
> Das Programm aus Iteration 1 soll erweitert werden. Der Auftraggeber sagt dir:
>
> *"Bitte bauen Sie das Programm so um, dass User dem Programm ihren Namen mitteilen, um damit persönlich begrüßt zu werden. Anwenderin Janine wird nicht mehr mit "Hello, World!", sondern mit "Hello, Janine!" begrüßt. Kriegen Sie das hin?"*

Welche *Logik* brauchst du dafür?

Auch diese Frage wirst du wahrscheinlich aus dem Stand beantworten können, wenn auch vielleicht mit ein wenig Unsicherheit, wofür solch einfache Problemstellungen gut sein sollen. Ein Verhalten wie das Folgende zu erzeugen, ist nun wirklich kein Hexenwerk:



Natürlich ist das keine große Herausforderung an deine Kunst, Logik für Funktionalität zu finden.

Aber was, wenn dieses Verhalten nicht den Qualitätsanforderungen in puncto Benutzbarkeit entspricht? Das stellt der Auftraggeber fest, wenn du ihm deine neue Lösung vorstellst. Eine Anwenderin kann zwar dem Programm den Namen "mitteilen", muss dazu aber wissen, dass das auf der Kommandozeile zu geschehen hat. Das hatte der Auftraggeber nicht im Sinn mit seiner obigen Spezifikation; wie selbstverständlich hatte er

gedacht, dass eine Anwenderin natürlich nach ihrem Namen *gefragt* wird, um ihn dann mitzuteilen.[19]

"Gedacht" hatte sich der Auftraggeber ein solches Verhalten:

Das passt genauso zur verbalen Spezifikation. Die Logik dafür sieht jedoch ganz anders aus als für die erste Implementation!

```
1  // Variante 1
2  Console.WriteLine("Hello, {0}!", args[0]);
3
4  // Variante 2
5  Console.Write("Name: ");
6  var name = Console.ReadLine();
7  Console.WriteLine($"Hello, {name}!");
```

Und damit ist die Lösung immer noch nicht in trockenen Tüchern! Denn was geschieht, wenn ein Anwender keinen Namen eingibt und nur ENTER drückt? Dann passiert dies bei Variante 2:

Ist das ein erwünschtes Verhalten aus Sicht des Auftraggebers? Nein. Der hatte sich bei der Formulierung "mitteilen ... kann" gedacht, dass ohne Name weiterhin mit "Hello, World!" begrüßt wird. Es gilt allerdings: "Gedacht ist nicht gemacht!" Auftraggeber müssen mehr, als sich Anforderungen denken oder darauf vertrauen, dass du "als Fachmann" schon weißt, was gemeint sein könnte. Sieh durch den Honig durch, den dir solche Formulierungen um den Bart schmieren: "Sie haben doch Erfahrung. Sie wissen doch, wie man das macht und was ich meine." Nein,

[19] Ich habe dich hier ein wenig hereingelegt. In der ersten Iteration war die Bildschirmausgabe die Spezifikation. In der zweiten eine rein verbale, auf die ich sofort einen Screenshot habe folgen lassen. Der hat dir vielleicht suggeriert, dass das darin zu sehende Verhalten das spezifizierte ist. Aber mitnichten! Es war schon eine Interpretation der verbalen Spezifikation. Du siehst, es ist so eine Sache mit den Anforderungen. Welche gelten, wann liegen sie vor, welche Form sollten sie haben, damit du ihnen vertrauen kannst? Dazu später mehr.

weiß du nicht! Du kannst dir zwar eine Menge denken - nur bedeutet das nicht, dass es dasselbe ist, wie sich der Auftraggeber denkt oder was ihm am Ende gefällt, was Wert darstellt. Wenn du hörst "Sie als Fachmann", ist Gefahr im Verzug! Dann musst du die Anstrengungen verdoppeln, den Kunden aus der Unklarheit zu locken - oder ihm ganz klar sagen, dass du nur Vorläufiges programmieren kannst.

Eine oder drei oder auch fünf Zeilen Logik zu finden, ist in diesem Szenario nicht das Problem. Doch schon bei dieser Größenordnung fehlt es eben an Klarheit, was überhaupt Wert für den Auftraggeber darstellt.

Mit iterativem Vorgehen lässt sich der Schaden jedoch begrenzen. Wenn du dem Auftraggeber nicht vorgaukelst, dass du seine Wünsche direkt umsetzen kannst, sondern Feedback-Schleifen benötigst, führen Kontraste zwischen Wunsch und Lieferung nicht zu Konflikten, sondern zu Informationen. Motto: "Gut, dass wir darüber gesprochen haben!"

Nach zwei Iterationen kann die Lösung dann so aussehen:

```
(base) Hereafter-3:netcoreapp3.1 ralfw$ dotnet exec hello.dll
Name: Janine
Hello, Janine!
(base) Hereafter-3:netcoreapp3.1 ralfw$ dotnet exec hello.dll
Name:
Hello, World!!
(base) Hereafter-3:netcoreapp3.1 ralfw$
```

Und die Logik hat dir natürlich keine Probleme gemacht. Wenn klar ist, was gewünscht ist, ergibt sie sich quasi von selbst und sieht z.B. so aus:

```
1  Console.Write("Name: ");
2  var name = Console.ReadLine();
3  if (string.IsNullOrWhiteSpace(name)) name = "World";
4  Console.WriteLine($"Hello, {name}!");
```

Vielmehr war es der Kunde mit der unklaren Spezifikation, der zu einem Umweg geführt hat. *Garbage in, garbage out*: Das gilt auch bei der Softwareentwicklung.

> **Iteration 3: Party time!**
>
> Das Programm aus Iteration 2 soll nun abermals erweitert werden, um zur Begrüßung auf Partys eingesetzt zu werden. Der Auftragge-

ber sagt dir:

"Ich bin Veranstalter von 2-3 Partys pro Woche, die von 50-100 Gästen besucht werden. Solche Partys veranstalte ich in 20-25 Wochen pro Jahr in den nächsten 1-2 Jahren. Die neue Version des Programms möchte ich auf meinem Laptop am Eingang der Partys laufen lassen.

Jeder Gast soll darin seinen Namen eingeben und persönlich begrüßt werden. Wenn z.B. Roger das erste Mal eine dieser Partys besucht, wird er mit "Hello, Roger!" begrüßt. Kommt er zum zweiten Mal, heißt es aber "Welcome back, Roger!" Ab dem dritten Besuch lautet die Begrüßung "Hello my good friend, Roger!". Und ist Roger schließlich das 25. Mal auf einer Party, ist einmalig der Zusatz auszugeben "Congrats! You are now a platinum guest!

Ich erwarte, dass ich während der Nutzungsdauer des Programms immer denselben Laptop verwende. Der wird vor Party-Beginn hochgefahren, das Programm wird einmalig gestartet für den Abend und am Ende mit CTRL-C beendet. Eine Internetverbindung besteht am Veranstaltungsort leider nicht verlässlich.

Können Sie das Programm in dieser Weise erweitern?"

Was nun? Sind die Anforderungen wieder unklar? Eher nicht. Es ließen sich zwar noch ein paar Fragen stellen, wie sich das Programm verhalten soll, wenn verschiedene Gäste denselben Namen haben. Doch diese Restunklarheit ist bei dieser Iteration nicht das Problem. Vertrau mir.

Bei dieser Iteration liegt vielmehr die Logik selbst bei klaren Anforderungen nicht mehr auf der Hand. Sie mag am Ende 10 oder 20 Zeilen umfassen - viel wäre das allerdings immer noch nicht. Dennoch wirst du bei dieser Iteration eine deutlich größere Unsicherheit verspüren. Du siehst keinen geraden Weg mehr zur Logik; sie springt dir nicht vor dein geistiges Auge. Deine Gedanken kreisen… du kannst jetzt nicht einfach codieren, sondern musst zuerst nachdenken.

Die Funktionalität selbst stellt jetzt schon ein Problem dar, obwohl das Szenario immer noch trivial ist. Und deshalb wird auch die Korrektheit relevant. Denn wo unklar ist, welche Logik die passende ist, ist sehr schnell auch unklar, ob die ausgewählte tatsächlich die Anforderungen erfüllt.

Darüber hinaus aber kommst du nicht mehr ohne Ordnung im Code aus. Deine kreisenden Gedanken suchen nicht nur die Logik für das Verhalten, sondern auch nach einer ordentlichen Struktur, *in der* du die Logik aufhängen kannst, um deine eigene Lösung zu verstehen.[20]

Diese Struktur wird jedoch nicht durch die Logik gebildet, es geht also nicht um den Algorithmus. Vielmehr geht es um einen *Rahmen* um Logik herum, also um Nicht-Logik Code. Wenn du dabei an Funktionen und Klassen (oder allgemeiner: Module) denkst, hast du die richtige Intuition.

Die Phasen der Programmierung

Zwischen den Anforderungen des Auftraggebers und der Logik, die zumindest die spürbaren Verhaltensanforderungen erfüllt, klafft eine gewaltige Lücke: die **Anforderung-Logik Lücke**. Schon in sehr simplen Szenarien wie dem vorgestellten liegt Logik nicht auf der Hand, sondern will gewissenhaft erarbeitet werden.

Wie die Iterationen des Beispiels zeigen sollten, geschieht das in drei Phasen, die strickt aufeinander folgen. Immer. Auch bei dir. Selbst, wenn du das nicht wahrnimmst oder nicht glaubst. Und auch wenn sie iterativ, also mehrfach durchlaufen werden, tut das dem Vorhandensein und der Reihenfolge der Phasen keinen Abbruch.

1. Phase: Analyse

Konfrontiert mit Anforderungen ist die Softwareentwicklung aufgerufen, zunächst eine für sie relevante Analyse zu machen. Diese Analyse hat als Ziel, **Verständnis** zu erzeugen. Nur wenn du *wirklich* verstanden hast, solltest du dich auf den Pfad der Code-Entwicklung machen. Ansonsten ist zu befürchten, dass das Resultat keinen Wert hat und/oder inkorrekt ist.

[20] Ich habe das Experiment genügend oft live mit Entwicklergruppen gemacht, um zu wissen wovon ich rede. Während bei den ersten beiden Iterationen die Logik herausgesprudelt wird, hängen Probanden dieses Experiments bei Iteration 3 und "stammeln sich etwas zusammen". Sie können die Logik nicht "herunterbeten", sondern drehen gedankliche Schleifen auf unterschiedlichen Ebenen. Meistens wollen sie mir etwas auf dem Level von Pseudocode erzählen oder nennen mir Gliederungspunkte. Konkrete Logik ist das alles aber nicht. Und das kann auch nicht sein. Dafür ist selbst dieses Beispiel zu groß. Es im Kopf und geradlinig zu lösen, können nur die allerbesten auf Anhieb - und auch das nicht verlässlich.

Verständnis drückt sich ausschließlich zweifelsfrei in Können aus.

Das weiß jeder, der eine Mathematik-Prüfung (aus eigenen Kräften) bestanden oder auch nicht bestanden hat.

Ein konkreteres Beispiel: Wer versteht, wie Fibonacci-Zahlen berechnet werden, der kann die Folge *1, 1, 2, 3, 5, 8* beliebig fortsetzen. Der weiß, welche Zahl auf *8* folgt, der weiß, welche Zahl auf *21* folgt; der weiß auch, ob *35* eine Fibonacci-Zahl ist oder nicht.

Der unzweideutige formale Ausdruck von Verständnis besteht deshalb in "Beispielaufgaben" für dich als Entwickler bzw. für die von dir zu entwickelnde Software. Nur Software, die diese "Beispielaufgaben" fehlerfrei löst, kann als anforderungskonform und korrekt akzeptiert werden.

Vorgelegt werden die "Beispielaufgaben" natürlich in Form von automatisierten Testfällen. Andernfalls ist nicht zu erwarten, dass sie verlässlich und nachvollziehbar und personenunabhängig überprüft werden.

Wenn Produktivität nicht durch Inkorrektheit behindert werden soll, muss Software auf Reife und Stabilität stets automatisiert mit relevanter Codeabdeckung getestet werden.

Automatisierte Tests sind die erste Bastion im Kampf gegen den Morast der schleichend wachsenden Unwandelbarkeit, der deine Produktivität in die Knie zwingt.

Der automatisierte Test hat allerdings eine Voraussetzung: Es muss auch klar sein, wie ein Test "an Logik angelegt" werden kann. Wie bekommt der Test Zugang zur zu testenden Logik? Das geschieht vor allem durch Aufruf von Funktionen.

Das gewünschte Verhalten wird durch mindestens eine Funktion in seiner Gänze repräsentiert (*API-Funktion*). Die Funktion oder andere unterhalb ihr im Aufrufbaum enthalten die Logik, die im Test getriggert wird.

Verständnis als Resultat der Analyse drückt sich aus in einer Reihe von Tupeln der Form *(Testfall, Funktion)*.

Für das Beispiel der Fibonacci-Zahlen könnte das so aussehen:

- Funktion: `int[] Fib(int n)`
- Testfälle:
  - Input: `n=0`, erwartetes Resultat: `[]`
  - Input: `n=1`, erwartetes Resultat: `[1]`
  - Input: `n=4`, erwartetes Resultat: `[1,1,2,3,]`

Daraus folgt:

Softwareentwicklung, die nachhaltige Produktivität ernst nimmt als Anforderung, ist grundsätzlich *test-first* Programmierung.

Das Ergebnis der Analyse sind **Akzeptanztests** für die zu entwickelnde Logik. Ohne Erfüllung ihrer Akzeptanztests ist Logik nicht reif; Akzeptanztests sind die Reifetests "an der Außenhaut" von Software. Und ohne unausgesetzte Erfüllung bisheriger Akzeptanztests ist Logik nicht stabil. Beides ist inakzeptabel im Sinne dauerhaft hoher Produktivität.

Der zweiten Iteration des obigen Programms fehlte es an formalem, dokumentiertem Verständnis. Deshalb ist die Entwicklung in die falsche Richtung gelaufen und hat auch noch den Eindruck der Inkorrektheit gemacht.

2. Phase: Entwurf

Die dritte Iteration im Beispiel hat natürlich auch noch unter einem Mangel an dokumentiertem Verständnis gelitten. Darüber hinaus waren die Anforderungen aber so umfangreich, dass sich auch gutes Verständnis nicht mehr "einfach so" in Logik hat umsetzen lassen.

Das Nachdenken über Code vor der Codierung in der IDE, das die dritte Iteration erzwungen hat, ist das, was ich **Design** oder **Entwurf** nenne. Diese Phase ist die zentrale Provokation der Softwareentwicklung, scheint

mir. Ihr müssen sich alle Entwickelnden stellen, hier ist echte Kreativität gefragt. Und hier gibt es den größten Widerstand seit Anfang der 2000er. Entwurf scheint überflüssig, hinderlich, verlangsamend. Meine Erkenntnis ist allerdings gegenteilig: Ich sehe, dass die Produktivität leidet, weil Entwicklungsteams einen Entwurf vernachlässigen.

Im Entwurf wird eine Lösung für das Problem gefunden, das die Anforderungen aufwerfen. Das ist allerdings nur eine konzeptionelle Lösung, ein Lösungsansatz. Der manifestiert sich in Code erst in der nächsten Phase.

Entwurf findet immer statt. Du kannst ihn sehr bewusst oder ganz unbewusst durchführen. Erfolgt er bewusst, ist er allerdings noch nicht notwendig auch systematisch. Deshalb lässt die Ordnung der "entworfenen" Strukturen oft zu wünschen übrig.

Entwurf ist *per definitionem* deklarativ.

Das heißt, im Entwurf steht keine Logik zur Verfügung. Entwurf liefert keine Algorithmen, sondern plant ein **Modell**.

Das Modell als Ergebnis des Entwurfs besteht aus einer Reihe von Funktionen die in Tupeln der Form *(Funktion1, Funktion2, Beziehungen)* verbunden sind.

Beispielhafte Beziehungen zwischen Funktionen `f` und `g` des Modells sind:

- `f` ruft `g` auf (Abhängigkeit)
- `g` folgt auf `f` (Sequenz)
- `f` spezialisiert `g` (Vererbung)
- `f` und `g` haben inhaltlich etwas gemeinsam (sie verfolgen den selben Zweck)
- `f` und `g` benutzen gemeinsamen Zustand

Das mag abstrakt klingen und Modelle müssen auch nicht in Form von 3-spaltigen Excel-Blättern geliefert werden. Ein Klassendiagramm, ein Datenfluss, eine Zustandsmaschine… das und mehr sind hilfreiche Ausdrucksformen für Modelle - die sich allerdings alle auf die obige sehr allgemeine Definition zurückführen lassen.

Zentral beim Entwurf eines Modells ist, dass es ganz bewusst von konkretem Code abstrahiert. Die Feinheiten einer Programmiersprache oder eines Framework und der Detailreichtum von Logik stehen nicht zur Verfügung. Der Lösungs*ansatz* ist "mit einfacheren Mitteln" zu finden.

Diese freiwillige Selbstbeschränkung hat mehrere Gründe:

- Weniger Details erlauben eine schnellere Lösungsfindung - auf hohem Abstraktionsnivau in Form eines Durchstichs.
- Eine deklarative Lösung erlaubt die einfachere visuelle Darstellung und damit Kommunikation zwischen Teammitgliedern. Mentale Modelle lassen sich externalisieren.
- Visuelle, abstrakte Lösungsansätze lassen sich in größerer Vielfalt gegenüberstellen, was der Findung besserer Lösungen dient.
- Einen Lösungsansatz zu finden erfordert andere geistige Aktivität/-Fähigkeit als die Codierung eines Lösungsansatzes. Die explizite Modellierung vor einer Codierung dient mithin der Entzerrung des Entwicklungsprozesses; es wird ermüdendes, verlangsamendes und fehlerträchtiges Multitasking vermieden.

Ein bewusster und systematischer Entwurf stellt ein Modell her, das nicht nur die Lösung der Verhaltensanforderungen repräsentiert, sondern auch noch der Forderung nach Ordnung genügt.

Wo die Analyse eine Bastion gegen Wertarmut und Inkorrektheit ist, da ist der Entwurf eine Bastion gegen Unordnung.

3. Phase: Codierung

Die Codierung schließlich setzt den Entwurf um in Code. Du übersetzt ein Modell mit einer Programmiersprache in Funktionen, die du mit konkreter Logik ausfüllst.

Ist das Modell gut, kann dieser Phase durchaus eine gewisse Langeweile anhaften. Das Problem ist ja (theoretisch) gelöst. Die Spannung ist raus aus den Anforderungen. Insofern ist mein Ziel mit *Programming with Ease*, dir die Codierung etwas zu verleiden. Du sollst sie am Ende als mechanische Arbeit auffassen, bei der nur noch relativ wenig Kreativität nötig ist. Ok, vielleicht übertreibe ich ein wenig, aber so ungefähr stelle ich mir das vor, weil ich es selbst so erfahren habe. Je leichter ich mir die Programmierung gemacht habe, desto unspannender wurde die Codierung.

Nachlässigkeit darf sich deshalb jedoch nicht einschleichen. Die Codierung hat ihre eigenen Probleme, die noch gelöst werden wollen. Hier schlägt die Stunde des Handwerkers, der seine Technologien beherrscht.

Das Ergebnis der Codierung ist - wie sollte es anders sein - Code. Aber nicht irgendein Code, sondern Code, der erstens der Ordnung des Modells folgt und zweitens in den Detail-Ebenen unterhalb des Modells ebenfalls Ordnung walten lässt.

Darüber hinaus ist die Codierung die Phase, in der du die automatisierten Prüfungen der Korrektheit implementierst.

Codierung stellt Produktionscode und Testcode paarweise *test-first* her.

Ordnung und Korrektheit dürfen bei der Codierung auf den letzten Metern nicht kompromittiert werden. Das ist kein kleines Kunststück unter dem üblicherweise herrschenden Druck von Lieferterminen.

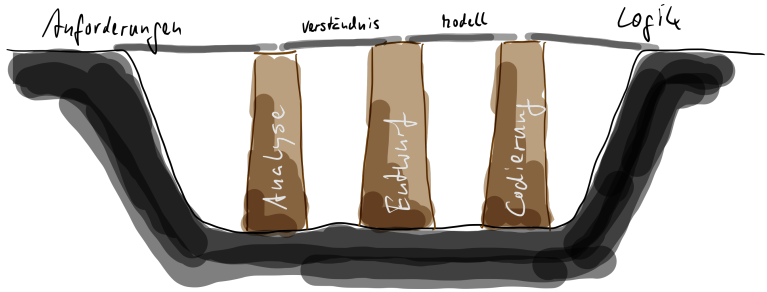
Zusammenfassung

Die Übersetzung von Anforderungen in Code ist eine komplexe Tätigkeit, die nur systematisch verlässlich alle Qualitäten herstellt: Wert in Form von Funktionalität + Effizienz, Korrektheit und Ordnung.

Die minimale Systematik, die ich dir mit *Programming with Ease* insgesamt vermitteln will, besteht darin, für gegebene Anforderungen eine für dich als Entwickler relevante Analyse durchzuführen, die nachvollziehbares Verständnis nicht nur dokumentiert, sondern auch automatisiert überprüfbar macht.

Ausgehend von diesem Verständnis wird dann im nächsten Schritt ein Lösungsansatz modelliert, der von den Feinheiten der Codierung bewusst abstrahiert für mehr Überblick, bessere Kommunizierbarkeit und größere Flexibilität.

Erst nach diesen Vorarbeiten kann alles bereit sein, um das zu tun, was man gemeinhin als die vordringliche Aufgabe von Softwareentwicklung sieht: die Codierung.



Eine Brücke über die Anforderung-Logik Lücke

Analyse → Entwurf → Codierung (AEC): Dieser Prozess ist unverbrüchlich, gar unvermeidbar. Daran glaube ich fest; das zu verstehen, hat mir die Softwareentwicklung erheblich erleichtert.

Das bedeutet jedoch nicht, dass Softwareentwicklung deshalb "im Wasserfall" oder nach BDUF (*Big Design Up-Front*) verlaufen müsste. Die Phasen AEC können beliebig häufig und beliebig schnell durchlaufen werden. Sie sollten lediglich dem Umfang und Schwierigkeitsgrad der anliegenden Anforderungen entsprechen.

Auf diese Weise wird die Lücke zwischen Anforderungen und Code systematisch und nachvollziehbar und teamfähig überwunden.

Übungsaufgaben

Übung macht den Meister! Deshalb gibt es zu (fast) jedem Kapitel Übungsaufgaben, die du in deiner Geschwindigkeit lösen kannst. Kein Druck, keine Anprüche, die andere dabei an dich haben könnten. Mach es dir gemütlich damit.

Zu allen Übungsaufgaben findest du im Anhang auch Musterlösungen. Mit denen möchte ich dir das Selbststudium erleichtern; versuche also nicht zu luschern, während du die Übungsaufgaben löst. Und bitte verstehe die Musterlösungen auch nicht als Abkürzung, mit denen du dir die eigene Lösung der Übungsaufgaben (er)sparst.

Wenn du wirklich, wirklich daran interessiert bist, zu lernen, d.h. deine Gewohnheiten zu verändern, dann brauchst du eigene Praxis. Du musst nach dem Lesen etwas tun mit dem Gelesenen. Gern kannst du natürlich die Anwendung in deinem Projektalltag versuchen; früher oder später musst du diesen Sprung ja ohnehin machen. Aber erstens sind die Probleme in deinem Projektalltag weniger überschaubar, so dass dir weniger klar ist, wie und wo mit dem Transfer des Gelesenen du anfangen kannst. Zweitens wirst du durch Anwendung des Neuen erstmal langsamer, weil du noch unsicher bist; das kann dir schnell scheele Blicke von den Kollegen einbringen und du fällst in alte Gewohnheiten zurück. Drittens kann ich dir keinerlei Feedback geben, noch nicht einmal in Form einer monologischen Musterlösung. Feedback ist aber extrem wichtig, wenn du eine neue Fähigkeit erwirbst.

Deshalb empfehle ich dir sehr, die Übungen zu machen als erste Anwendung des Lernstoffs "in einer Sandkiste". Die Aufgaben sind überschaubar, keiner redet dir rein und macht druck und mit den Musterlösungen bekommst du zumindest eine gewisse Form von Feedback bzw. Kontrast zum Nachdenken.

Um deine Lösungen der Übungsaufgaben zu dokumentieren, lege für dich ein Git-Repository an, in dem du all deine Arbeitsergebnisse speicherst. Committe häufig und vergiss am Ende das Push nicht.[21]

[21]Wenn du noch nicht so viel mit Git gearbeitet hast, kannst du einen der bequemen visuellen Git-Clients benutzen wie z.B. den kostenlosen von GitHub. Eine Übersicht findest du hier.

Ein Git-Repository ist das unterste und einfachste Sicherheitsnetz, das du für deine Programmierung spannen kannst. *Never code without it.*[22]

Reflexionsaufgabe

Bitte formuliere eine Frage *oder* eine Erkenntnis zum Kapiteltext.

- Wo bist du gedanklich hängengeblieben, was ist dir unklar, was passt für dich irgendwie nicht zusammen, wozu würdest du dir noch etwas mehr Erklärung wünschen? Steht irgendetwas zu deiner bisherigen Praxis im Widerspruch und du fragst dich, warum du etwas ändern solltest?
- Oder: Wann hast du einen Aha-Moment gehabt, was ist dir als bemerkenswert, spannend, ausprobierenswert aufgefallen? Hat irgendetwas "in dir Klick gemacht", weil dir nun ein Zusammenhang aufgegangen ist? Oder verstehst du jetzt aus deiner bisherigen Praxis irgendetwas besser?

Am besten formulierst du Frage bzw. Erkenntnis schriftlich. Indem du deine Gedanken aufschreibst, wirst du dir ihrer bewusster. Bei einer Frage kommst du dadurch vielleicht schon einer Antwort aus dir selbst heraus näher. Bei einer Erkenntnis fällt dir vielleicht schon etwas ein, das du ab jetzt anders machen kannst.

Aufgabe 1 - Erklären

Beschreibe mit min. 500 bis max. 1000 Worten den Nutzen eines explizten Entwurfs bzw. der Modellierung für die Softwareentwicklung. Warum sollte man selbst bei hohem Verständnis der Anforderungen nicht sofort loslegen mit der Codierung, sondern zuerst nachdenken und modellieren?

[22] Aber nicht nur den Codeanteil deiner Lösungen solltest du in Repository legen. Alle Artefakte sind es wert, aufbewahrt zu werden. Vielleicht schreibst du Analysen in einem .txt/.docx Dokument auf oder machst eine Zeichnung in Visio oder hast eine Skizze auf Papier (die du fotografieren kannst), dann committe all das ebenfalls. So schaffst du dir eine Dokumentation der kompletten Lösungsentwicklung.

Versuche dich an einer ganz einfachen Erklärung im Stile von ELI5: Explain it like I'm 5 years old[a].

[a] https://www.dictionary.com/e/slang/eli5/

Aufgabe 2 - Modellieren

Auf der Basis des bisher Gesagten und deinem Verständnis von dem, was Entwurf ausmacht, entwickle ein Modell für die Iteration 3 des Hello-World Problems: Gäste sollen auf Partys begrüßt werden. Wie kann ein Lösungs*ansatz* aussehen, ohne dass du auch nur eine Zeile Code schreibst. Halte also auch Abstand vom üblichen Pseudocode!

Erinnere dich, dass ein Modell deklarativ ist. Logik steht dir nicht zur Verfügung - und trotzdem soll mit einem Modell der Lösungsansatz beschrieben werden. Einem anderen Entwickler, dem du ein Modell zeigst, soll die Codierung deutlich leichter fallen, als ohne Modell.

Einerseits soll das Modell die Lösung beschreiben, also schon konkret sein. Andererseits jedoch soll das Modell nicht zu konkret sein. Es soll Abstand von Details halten, die erst in der Codierung ausgefleischt werden. Ein Modell ist mithin eine abstrakte Lösung.

Wie könnte die für das Hello-World Problem aussehen? Was sollte darin beschrieben sein - und was sollte ausgelassen werden?

Versuche dich einmal daran mit deinen bisherigen Erfahrungen mit Softwareentwürfen. Oder vielleicht hast du von anderen schonmal gehört, wie die soetwas angehen.

Keine Angst, dass du diese Aufgabe "falsch" lösen könntest. Es geht nicht darum, sie in bestimmter Weise zu erfüllen, also auf "das eine richtige" Modell zu kommen. Diese Aufgabe soll dich schlicht auf andere Weise als die erste zu einer aktiven Auseinandersetzung mit dem Entwurfsbegriff anregen.

02 - Entwurf im Überblick

Im Entwurf wird die Umsetzung vorweggenommen. Er stellt die Lösung des Problems dar, ohne "es zu tun". Er entwickelt nur eine Vorstellung davon, wie die geforderte Leistung durch Software erbracht werden könnte.

Bevor ich dir konkret erkläre, wie ich meine, dass du sehr leichtgewichtig entwerfen solltest und warum genau so in einer bestimmten Weise, möchte ich dir ausführlich darstellen, was ich grundsätzlich damit meine und was das soll.

In Kapitel 01 habe ich schon versucht, den Entwurf als unverbrüchliche Phase jeder Softwareentwicklung herauszuarbeiten. Du kommst aus meiner Sicht sowieso nicht um ihn herum, auch wenn du der Meinung bist, ihn nicht zu brauchen. Doch lass uns noch einen genauere Blick darauf werden, was das ist, der Entwurf.

Den Entwurf abstecken

Mit ein paar Aussagepflöcken stecke ich das Thema Entwurf mal grob ab. Das ist abstrakt, aber keine Sorge, du wirst später noch genügend konkrete Entwürfe sehen.

1. Ein Entwurf stellt die **Lösung eines Problems** dar, d.h. er erfüllt die Anforderungen des Auftraggebers.
2. Entwurf ist allerdings nicht die eigentliche Sache, die der Auftraggeber will. Wie auch immer ein Entwurf aussieht, es ist also kein Code, er ist **nicht ausführbar**. Das widerspräche der Definition von Entwurf.
3. Ein Entwurf ist nur eine **Beschreibung** der eigentlichen Sache, insofern ist seine Lösung nur theoretisch/konzeptionell. Nicht jede Beschreibung der eigentlichen Sache ist jedoch ein Entwurf. Damit eine Beschreibung ein Entwurf ist, muss sie ***vor* der Herstellung** der Sache angefertigt worden sein. Eine Beschreibung der eigentlichen Sache nach der Herstellung ist eine Dokumentation.

4. Ein Entwurf als Beschreibung dessen, was Code leisten soll, bevor dieser Code geschrieben wird, hat den Zweck, die **Codierung deutlich zu erleichtern**, wenn nicht gar, sie überhaupt zu ermöglichen.
5. Der Preis für die Erleichterung der Codierung darf allerdings nicht zu hoch sein. Es ist ein gutes Verhältnis zwischen Entwurfs- und Codierungsaufwand zu finden, das den Gesamtaufwand der Softwareherstellung reduziert. Indem Entwurf und Codierung zusammenkommen, soll **Energie frei werden**, die vorher gebunden war in "roundtrips" (aka Debugging, Testsitzungen, Iterationen).
6. Um eine Lösung zu sein, die vor der Codierung entwickelt werden kann, diese erleichtert und selbst nicht zu aufwändig ist, muss ein Entwurf auf einer **deutlich höheren Abstraktionsebene** stattfinden, als die Codierung. Er darf nicht durch Details der Codierung behindert werden; er sollte weniger Komplexität als der spätere Code aufweisen.
7. Die höhere Abstraktion darf jedoch nicht dazu führen, dass der Entwurf abhebt. Er muss also gleichzeitig **konkret und aussagekräftig** sein.
8. **Leichtgewichtigkeit** ist ein Kennzeichen für hilfreiche Entwurfswerkzeuge, denn sonst werden sie nicht benutzt, wenn der Druck, mit der Codierung zu beginnen, groß ist.
9. Und schließlich muss ein Entwurf durch die Realität der Codierung informiert sein und sich **geradlinig in Code übersetzen lassen**. Es braucht also Bezugspunkte im Entwurf für den Code.

Entwurf und Implementation sollen "nicht überlappen" und die Implementation soll den Entwurf "spiegeln".

- Der Entwurf löst das Problem, nur nicht genauso wie der Code.
- Und der Code lässt erkennen, dass er aus dem Entwurf abgeleitet wurde, d.h. Verbindungen zwischen ihm und den Entwurfsmitteln sind klar.

Mit dem Entwurf erarbeitest du eine Zielvorstellung für den Code, d.h. die lauffähige Lösung.

Hierarchie der Lösungen

Nun gibt es die Sichtweise, dass Code selbst schon ein Entwurf (engl. *design*) sei.[23] Denn der Code, den du in C# oder Java oder JavaScript schreibst, ist ja nicht das, was am Ende ausgeführt wird und tatsächlich das Problem des Kunden löst.

Dein Hochsprachencode wird übersetzt in Maschinencode. Das kann man als eine Form von Produktion ansehen. Und Produktion basiert auf einem Plan, einem Entwurf. Ein Haus wird nach einem Plan hergestellt, einen IKEA-Schrank baust du nach einem Plan auf.

In der materiellen Welt ist die Produktion so sichtbar und aufwändig, dass ein vorheriger Entwurf zwingend ist und auffällt. Bei der Softwareentwicklung ist die Produktion hingegen so unsichtbar und schnell, dass sie nicht auffällt - und man die Codierung für die Produktion halten könnte.

Ich mag mich diesem Verständnis jedoch nicht recht anschließen. Oder wenn ich mich ihm anschließe, dann finde ich die Aussage nicht hilfreich. Ob Code nun ein Entwurf ist oder nicht, ändert nichts an der Tatsache, dass er selbst unabhängig von jeder Kategorie sehr schwer zu schreiben und zu verstehen ist.

Wenn Code eine Form von Design darstellt, dann ist eben dieses Design sorgfältig zu produzieren. Dann muss ein Entwurf sogar vor diesem Design stattfinden.

Bei einer gegebenen manifesten Lösung, sei das Maschinencode oder Hochsprachencode, ist ein Entwurf das, was der Lösung vorhergeht und sie auf einem höheren Abstraktionsniveau vorwegnimmt. Lösungen existieren mithin nicht nur in einer Form, sondern in einer Hierarchie. Lösungen gibt es auf vielen unterschiedlichen Abstraktionsebenen. So kann des einen manifeste Lösung des anderen theoretische sein, also "nur" ihr Entwurf. Aus dieser Perspektive betrachtet sehe ich mindestens folgende Abstraktionsniveaus:

1. Maschinencode ist die manifeste Lösung, das Produkt? Dann ist Hochsprachencode der Entwurf.

[23] vgl. Code as Design: Three Essays by Jack W. Reeves, https://www.developerdotstar.com/mag/articles/PDF/DevDotStar_Reeves_CodeAsDesign.pdf

2. Hochsprachencode ist die manifeste Lösung, das Produkt? Dann ist ein **Modell** dessen Entwurf.
3. Ein Modell ist die Lösung, das Produk? Dann ist ein **Lösungsansatz** dessen Entwurf.
4. Ein Lösungsansatz ist die Lösung, das Produkt? Dann ist eine "Produktidee" dessen Entwurf.

Mit Maschinencode bist du wahrscheinlich nicht vertraut. Das macht nichts, denn ich will ja nicht die Umwandlung von Hochsprachencode in ihn behandeln. Auch wenn ich es früher immer gern schreiben wollte, ist dies kein Buch über Compilerbau.

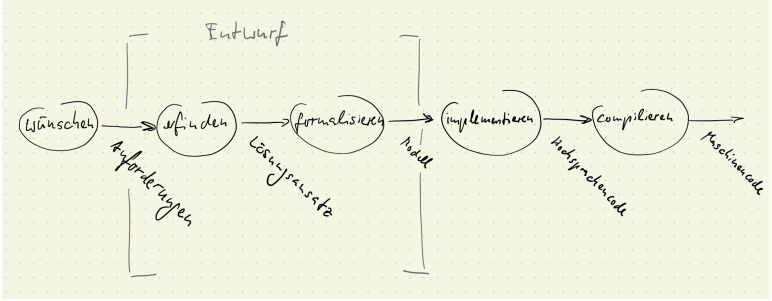
Mit Hochsprachencode bist du vertraut. Den will ich dir deshalb ebenfalls nicht erklären. Dies ist keine Einführung in die Programmierung. Aber die Übersetzung von Entwurf - genauer: Modell - in Hochsprachencode, das ist eine andere Sache, den werde ich dir ausführlich vorstellen.

Entwurf, wie ich ihn hier verstehe, führt zu Hochsprachencode und erfolgt auf zwei Ebenen: zuerst in Form eines informellen Lösungsansatzes für eine "Produktidee", dann in Form einer formalen Modellierung des Lösungsansatzes.

Die "Produktidee" - also letztlich das, was Anforderungen des Auftraggebers beschreiben - lasse ich ebenfalls aus in diesem Band. Wie du zu den Voraussetzungen für einen Entwurf von Hochsprachencode kommst, ist Thema des dritten Buches der Reihe *Programming with Ease*.

Nur soviel an dieser Stelle: Auch die Anforderungen beschreiben die Lösung. Allerdings ist dieser Entwurf so abstrakt, so weit von der Codeebene entfernt, dass man "das entwerfende Element" darin im Grunde nicht erkennt. Anforderungen sind mehr ein "Wunschkonzert". Dennoch, wenn du genau hinschaust, befinden sich Anforderungen in dem von den neun obigen Pflöcken abgesteckten Gebiet. Lediglich Punkt 9, die geradlinige Übersetzbarkeit in Code, erfüllen sie nicht.[24]

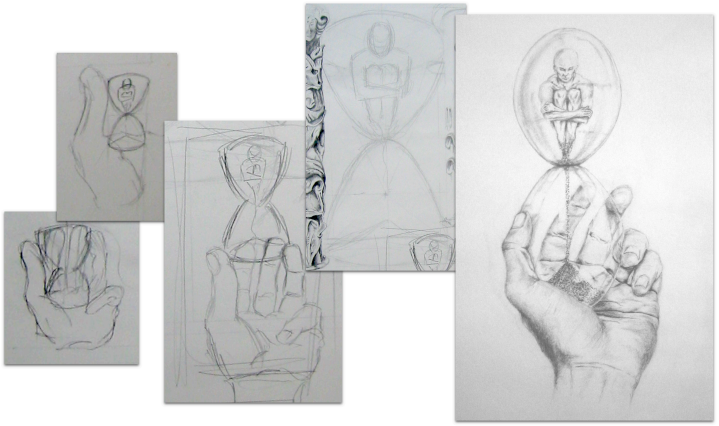
[24]Ja, sogar den Punkt 8, die Leichtgewichtigkeit, würde ich ihnen zugestehen. Darum hat sich die Agilität sehr bemüht. Die Möglichkeit der Leichtgewichtigkeit ist ein Resultat des iterativ-inkrementellen Vorgehens.



Der Entwurfsprozess - oder sogar die mehreren Entwurfsschritte hinab die Abstraktionshierarchie - kannst du als Schärfung verstehen. Ein zunächst grobes Bild, ein Wunschbild, wird schärfer, detailreicher, klarer mit jeder Phase. Wie dir die 3. Iteration des Hello-World Beispiels in Kapitel 01 gezeigt hat, ist ein Sprung vom Wunsch zum Hochsprachencode nicht möglich. Du musst dich heranarbeiten. Du musst dir erst ein grobes Bild machen, das du nach und nach verfeinerst.

Von der Kunst lernen

In der bildenden Kunst werden Skizzen und Kompositionen angefertigt, bevor der Künstler sich an die Schaffung des eigentlichen Werkes macht. Hier ein Beispiel dafür aus meiner Zeichenmappe aus Jugendjahren:



Von der groben Idee zum finalen Werk in vier Entwurfsschritten

- Am Anfang stand nur eine Idee: Ich wollte die Endlichkeit des menschlichen Lebens darstellen. Eine Sanduhr war mir dazu gleich vor Augen. “Bild mit Sanduhr und Mensch” war also meine Anforderung.
- Davon ausgehend habe ich zuerst recht pauschale Entwürfe gemacht. Die haben die Komposition geklärt, also die Grobstruktur des Werkes. Anfänglich hatte ich nur die Idee einer Sanduhr *in* einer Hand. Erst im zweiten Schritt kam die menschliche Gestalt dazu. Die war also noch nicht Bestandteil der ursprünglichen Anforderungen, sondern hat sich ergeben.
- Im dritten Schritt ist die Sanduhr *auf* die Hand gewandert, die nun auch schon etwas detaillierter ausgearbeitet ist. Auch dieser Schritt war nicht vorherzusehen, sondern ein Ergebnis dessen, dass ich mir die Idee vorher mit den beiden anderen Entwürfen vor Augen geführt hatte.
- Das vierte Bild ist eine Detailstudie. Der Entwurf konzentriert sich auf eine genauere Ausarbeitung nur der Sanduhr mit der menschlichen Figur. Dort habe ich wohl noch Unsicherheit gespürt und wollte mich vergewissern.
- Im finalen Werk spiegeln sich die Entwürfe deutlich - aber es sind auch Abweichungen zu erkennen. Die Handhaltung ist nochmal leicht anders und die Form der Sanduhr hat sich verändert. Wenn ich mich recht erinnere, hatte ich bei der Ausarbeitung gemerkt,

> dass ich meine linke Hand, die mir Modell stand, besser so mit dem Stundenglas halten konnte. Außerdem hatte das konkrete Glas, das ich hielt, diese Eiform.

Die Entwürfe waren schnell gemacht und abstrakt. Dennoch - oder gerade deshalb - konnte ich mich mit ihnen zügig an die letzte Variante der Lösung heranarbeiten, das Modell. Die Übersetzung in das endgültige Werk hat dann deutlich länger gedauert und musste das Modell nochmal der Realität der Umsetzung anpassen.

Meine Erfahrungen mit der Zeichnenkunst sagen mir: ohne Entwurf geht es nicht. Ein nicht triviales Werk entsteht nicht ohne eine erkleckliche Zahl von Entwürfen, die auf unterschiedlichen Abstraktionsniveaus liegen und sogar unterschiedliche Ausschnitte behandeln.

Entwerfen ist fachgerecht

Das kreative Werk als manifeste Lösung braucht einen iterativen Prozess. In dem wird eine Vorstellung als Skizze externalisiert, um dann in der Betrachtung zurück zu wirken auf die Vorstellung.

Vorstellungen so greifbar wie möglich vor sich zu stellen, um sie von allen Seiten auf ihre Wirkung (Lösungstauglichkeit) zu überprüfen, ist für mich genauso natürlich wie zwingend. Weniger geht nicht in einem kreativen Prozess. Wer versucht, das Kunstwerk lediglich im Kopf kurz anzudenken, um es dann sogleich in seiner finalen Form zu produzieren, wird viel Verschwendung betreiben.[25] Beim Zeichnen besteht sie aus Zeit und ist erkennbar am Verbrauch von Papier und Stiften.

Bei der Programmierung besteht die Verschwendung auch aus Zeit, aber erzeugt leider keinen Materialverbrauch. Das macht es so verführerisch,

[25]Nicht zu entwerfen und sofort zu produzieren, gibt es allerdings auch. Das ist eine eigene Kunst. Die nennt man Improvisation, würde ich sagen. Ohne Planung geht's gleich ins Tun. Im Theater gibt es dafür z.B. eine eigene Kategorie: das Improvisationstheater (Impro-Theater). Schauspieler im Impro-Theater zu sein, ist eine ganz andere Herausforderung als Schauspieler in einem normalen Stück zu sein. Im normalen Stück gibt es ein Theaterstück als "Entwurf", das durch die Aufführung "produziert" wird. Beim Impro-Theater gibt es das nicht. Es wird ohne Entwurfsschritt eine Idee aus dem Publikum sofort in Handlung umgesetzt. Das ist eine ganz eigene Kunst mit ihrem eigenen Reiz und ihren eigenen Grenzen. Ein Äquivalent in der Softwareentwicklung könnte vielleicht das Prototyping sein. Das hat seinen Nutzen und Reiz und auch seine Grenzen.

glaube ich, den Entwurf zu überspringen. Verschwendung ist von produktiver Arbeit oberflächlich schwer zu unterscheiden. Erkennbar ist Verschwendung primär an Inkorrektheit und Unordnung und sekundär an Verzögerungen und Frustrationsäußerungen.

Nach 40 Jahren Programmierung bin ich der festen Meinung: **Wer auf einen expliziten und auch noch visuellen Entwurf vor der Produktion von Hochsprachencode verzichtet, der handelt fahrlässig und verschwendet das Geld seiner Auftraggebers. Vor dem Codieren zu entwerfen, ist für mich ein Grundbaustein fachgerechter Arbeit als Softwareentwickler**. Und ebenso gehört zur fachgerechten Arbeit die test-first Codierung, wie im ersten Band der Reihe ausgeführt.

Entwerfen ist agil

Dass der explizite Entwurf seit Aufkommen der Agilität zunehmend in Verruf geraten ist, ist ein Übelstand, den ich nicht genug bedauern kann. Und wo das im Namen der Agilität geschehen ist, halte ich es für ein grobes Missverständnis der Agilität.

"*Working software over comprehensive documentation*" im Agilen Manifest[26] ist keine Aufforderung, auf Entwurf zu verzichten. Ausdrücklich ist ja *documentation* genannt, nicht *design*. Wie oben definiert, ist Entwurf jedoch keine Dokumentation, wenn auch "lediglich" eine Beschreibung und keine *working software*.

Und nur, weil es heißt *"responding to change over following a plan"*, ist das keine Aufforderung jegliches Planen sein zu lassen. Dann dürfte es ja auch kein Spring Planning in Scrum geben. Ein Entwurf ist ein Plan im Sinne einer Gestaltungsidee für einen zukünftigen Zustand der Welt. Er drückt den Glauben aus, "Ja, so wird es wohl funktionieren!" Doch deshalb ist ein Entwurf nicht unumstößlich. Meine Skizzen oben im Vergleich zum finalen Werk beweisen es: Nur, weil das Werk anders ist als die Skizzen, sind die nicht unnötig gewesen. Die Abweichung vom Plan, den Skizzen darstellen, ist selbstverständlich erlaubt, wenn bei der Ausführung neue

[26] https://agilemanifesto.org/

Erkenntnis auftauchen.[27]

Ein Entwurf steht einem *"deliver working software frequently"* aus den 12 Prinzipien des Agilen Manifests[28] ebenfalls nicht im Wege. Im Gegenteil! Durch Entwurf wird der Code korrekter und ordentlicher und also wandelbarer.

Und ein visueller Entwurf, wie ich ihn dir nahelegen werde, ich ein Beförderer des Prinzips *"the most efficient and effective method of conveying information to and within a development team is face-to-face conversation."* Wenn du eine Lösungsidee hast und kannst die nicht anders als in Code ausdrücken, dann lässt sie sich nur sehr schwer kommunizieren und diskutieren. Ohne Entwurf reduzierst du die Chance auf *face-to-face conversation.*

Schließlich: Wie willst du als agiler Programmierer dem Prinzip *"Simplicity - the art of maximizing the amount of work not done - is essential"* dienen, ohne einen Entwurf? Nur mit einem Entwurf kannst du nämlich überhaupt über Arbeit sprechen, bevor du sie tust. Sobald du an der IDE sitzt und Code tippst, steigerst du den *amount of work*. Besser, du klärst vorher ein paar Alternativen ab und diskutierst mit deinen Kollegen.

Dafür brauchst du allerdings eine klare und anfassbare Vorstellung von deiner Lösung vor deren Implementation. Zu der kommst du in zwei Schritten:

[27] Der Zweck von Planung ist, Überblick zu gewinnen und zu entzerren. Eine Form von Multitasking soll vermieden werden. Wenn ich eine Aufgabenliste abarbeite, die ich mir gestern für heute zusammengestellt habe, profitiert meine Konzentration davon, dass ich mich nicht mehr frage, "Was soll ich als nächstes tun?" Die Frage habe ich gestern beantwortet, als ich dafür in einem "speziellen Bewusstseinszustand" war. Gestern war ich kreativ, gestern hatte ich Überblick. Heute will ich nicht mehr kreativ sein, sondern Dinge nur erledigen. Dazu brauche ich einen anderen "Bewusstseinszustand". Falls ich jedoch auf ein Hindernis stoße, kann ich auch vom Plan abweichen. Neue Informationen dürfen, sollen, müssen den Plan verändern können. Hätte ich die Informationen gestern gehabt, hätte ich den Plan von vornherein anders gestaltet. Das Hindernis reißt mich heute zwar aus meinem "Bewusstseinszustand" der Abarbeitung - aber was soll's? Lässt sich nicht ändern. Ich mache das beste daraus, indem ich kurz wieder in den Planungsmodus gehe.

[28] https://agilemanifesto.org/principles.html

1. Der Lösungsansatz

In den Entwurf gehst du, wenn du die Anforderungen verstanden hast. Vorher hast du einfach nicht genügend Grundlage, auf der du eine Lösung aufbauen könntest.

Hier ein Beispiel für Anforderungen die das Kriterium auf Kapitel 01 erfüllen: Es liegen Beispiele vor und eine Funktionssignatur ist gegeben.

> Eine Liste von ganzen Zahlen soll in eine Ordnung gebracht werden, bei der für jede Zahl am Listenplatz mit Index *i* (kurz: `z[i]`) gilt: `z[i-1] <= z[i] <= z[i+1]`.
>
> Funktion:
>
> - `int[] Ordnen(int[] zahlen)`
>
> Beispiele:
>
> - `[1,5,2,9,8,4]-> [1,2,4,5,8,9]`
> - `[]-> []`

Du erkennst natürlich sofort, das mit "ordnen" hier gemeint ist "sortieren": Die Zahlen sollen aufsteigend sortiert werden.

Schon diese Klassifikation der Anforderungen ist ein erster Schritt in Richtung Lösung. Jetzt kannst du nämlich in der Literatur nachschlagen, wie man das macht. Andere haben das Problem schon vor dir gelöst. Du müsstest nicht einmal selbst entwerfen, sondern womöglich nur eine Lösung abschreiben. Oder noch besser: deine Programmiersprache oder Standardbibliothek der Programmiersprache bieten bereits eine fertige Sortierfunktion.

"Ah, das Problem lässt sich durch Sortieriung lösen!" würde ich als erste Erkenntnis im Sinne eines Lösungsansatzes in diesem Fall zählen.

Aber der Übung halber will ich davon absehen, dass es schon Lösungen für

das Sortieren gibt. Das Problem ist nämlich gut geeignet, um zu erklären, was ein Lösungsansatz ist.

Du könntest natürlich jetzt jedes weitere Nachdenken in den Wind schlagen und mit *classical TDD* in die Tasten greifen. Inkrementell könntest du versuchen, direkt eine Lösung zu codieren. Das ist bestimmt möglich. Die Testfälle könnten z.B. wie folgt schrittweise schwieriger werden:

1. `[]-> []` // Akzeptanztest und natürlicher Startpunkt
2. `[3]-> [3]`
3. `[1,3]-> [1,3]`
4. `[3,1]-> [1,3]`
5. `[3,1,4]-> [1,3,4]`
6. `[3,4,1]-> [1,3,4]`
7. `[1,5,2,9,8,4]-> [1,2,4,5,8,9]` // finaler Akzeptanztest

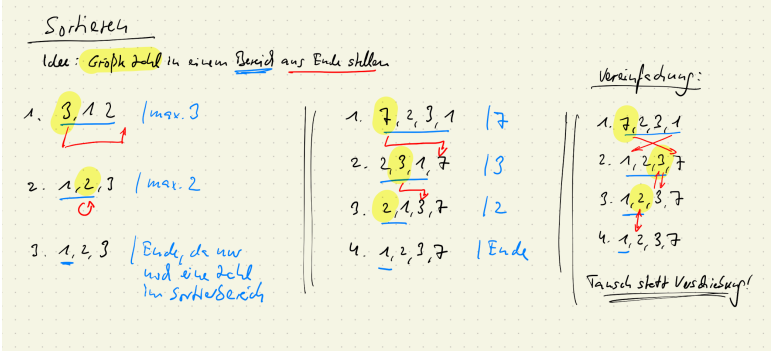
Die Zunahme des Schwierigkeitsgrades der Tests sieht plausibel aus. Ohne weitere Lösungsidee kannst du dir da jedoch nicht sicher sein. Das ist ein Grund, warum ich zwar sehr für *test-first* Codierung bin, doch nur mit einem vorherigen Entwurf.

Wenn du einmal versuchst zu vergessen, was du alles schon über Sortieralgorithmen weißt, was würdest du nach Studium der Anforderungen tun? Nein, nicht codieren, sondern im Kopf oder auf einem Stück Papier.

Jetzt ist womöglich die größte Kreativität in der Softwareentwicklung gefragt. Ich halte diese Phase jedenfalls für ihren Kern. *Dafür* sind Millonen Menschen Softwareentwickelnde geworden! Das ist der Teil, wo du an einem Problem knobeln kannst. Wer knifflige Probleme liebt, der ist im Entwurf bei der (Er)Findung eines Lösungsansatzes genau richtig.

Ich glaube, selbst dieses Problem löst du nicht im Kopf. Du musst deine Vorstellungen vor dir manifestieren. Blatt und Stift reichen dafür aus. Beim Lösungsansatz gelten keine Regeln. Alles ist erlaubt, was dich dem Ergebnis näher bringt. Das ist echte Freistil-Softwareentwicklung. In allen weiteren Phasen musst du irgendwelchen Regeln und Formalismen folgen. Genieße also die Freiheit in diesem Moment!

Wie kannst du das Problem auf einem Blatt Papier angehen? Hier ist mal ein Vorschlag:

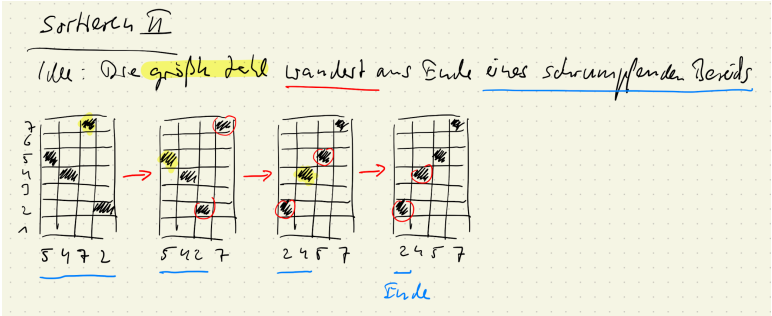


Lösungsskizze für die Sortierung

Ist das formal? Nein. Verstehst du, was ich damit meine? Wahrscheinlich nicht. Kenne ich jetzt die Lösung? Ja! Und nur das zählt.

Ich habe mit einer Notation mit graphischen Elementen und Konventionen, die ich mir spontan ausgedacht habe, eine Darstellung geschaffen, die es mir erlaubte, meine ganz grobe Lösungsidee aus meinem Kopf aufs Blatt zu bekommen. Du siehst eine Lösung als Skizze.

Es hätte aber auch anders aussehen können. Über die folgende Darstellungsart bin ich im Internet gestolpert, als ich für das Buch recherchiert habe:



Hier wird die zu sortierende Liste als Matrix dargestellt, bei der in der vertikalen die Werte an der jeweiligen Position aufgetragen sind. Dadurch ist die schrittweise Herstellung der gewünschten Ordnung visuell sehr schön nachvollziehbar. Der gelbe Wert in einem Bild *i* ist im Bild *i+1* einfach mit dem bis dahin letzten im blau markierten Listenabschnitt vertauscht

worden (rot). So formen die Punkte von Bild zu Bild zunehmend eine aufsteigende Linie.[29]

Beim Lösungsansatz geht es nur um das Verfahren. Solange das plausibel wird, konkret und erklärbar(er) ist, ist das Ziel dieser Entwurfsphase erreicht.

Wenn du mit einem Problem konfrontiert bist, kann es sein, dass du es sofort lösen kannst. Das ist der Fall bei der vorliegenden Aufgabe. Du kannst natürlich eine Liste von Zahlen sortieren.

Nur, weil du das kannst, kannst du es aber noch lange nicht auch noch erklären. Wie geht das mit dem Sortieren? Wie gehst du vor? Was ist dein Verfahren, deine Herangehensweise, deine Methode, dein Ansatz? In der ersten Phase des Entwurfs findest du nicht nur heraus, ob du "es" kannst oder "irgendwie weißt wie es geht". Nein, du musst dein Können und Wissen vermitteln können. Erste Herausforderung: Kannst du es dir selbst *erklären*? Zweite Herausforderung: Kannst du es *anderen* erklären?

In dieser Phase bist du im Grunde Erfinder. Du brauchst dafür kein wirres Haar, keine Brille und auch kein chaotisches Arbeitszimmer. Du bist Erfinder qua Aufgabe, die lautet: *Finde eine Lösung, die du erklären kannst.*

Deine Erfindung bezieht sich auf die anliegende funktionale oder nicht-funktionale Aufgabe. Im Beispiel ist es zunächst nur die funktionale, eine Liste überhaupt zu sortieren. Sie besonders effizient zu sortieren, war nicht gefragt.[30]

[29] Der Lösungsansatz "erfindet" übrigens den Sortieralgorithmus Selection Sort. Den habe ich hier gewählt, weil er so naheliegend ist. Ohne mich an *Selection Sort* konkret erinnert zu haben, fiel mir dieser Ansatz einfach ein. Wenn dir ein anderer eingefallen ist, ist das natürlich ebenso gut. Nur, wie hättest du deinen Lösungsansatz *dargestellt*?

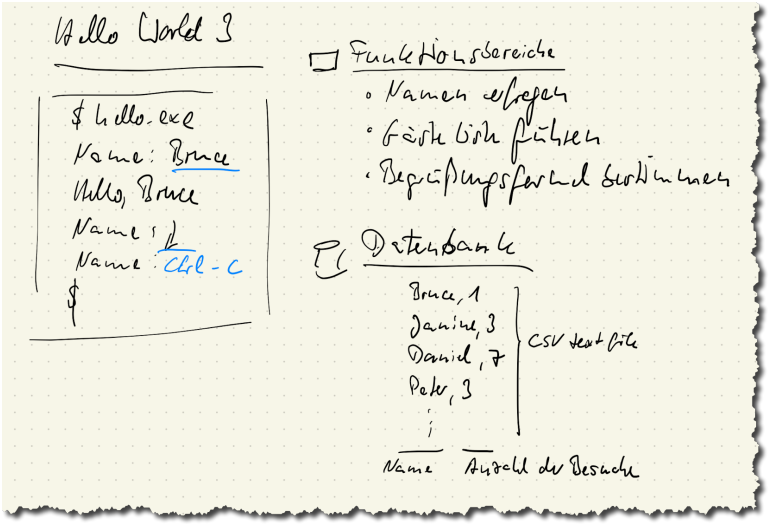
[30] Um auch noch eine besonders effiziente (hier: performante) Lösung zu finden, musst du wahrscheinlich mehrere Lösungsansätze entwickeln und vergleichen. Nimm dir also zu Anfang nicht zu viel vor: Finde zunächst einen funktionalen Lösungsansatz. Erst wenn du den hast, suche nach weiteren mit besseren Effizienzcharakteristika. Das zu trennen, fällt vielen Entwicklern schwer. Sie wollen gleich die optimale Lösung. Doch damit stehen sie sich selbst im Wege. Das halte ich für ein Rezept für Frust und Verzögerung. Außerdem verschenkt soviel Vorsatz die Chance auf kleinere Iterationen. Wenn der Auftraggeber eine funktionale und effiziente Lösung will, dann biete ihm an, zunächst nur eine funktionale zu liefern. Dann kann er schonmal überprüfen, ob das seinen Wünschen entspricht. Falls nämlich nicht, hast du keinen Optimierungsaufwand verschwendet. Und sollte alles ok sein, fängst du dann mit der Optimierung an - oder der Auftraggeber entscheidet, dass mehr Effizienz doch nicht nötig ist, da er nun gesehen hat, was eine "nur" funktionale Lösung schon bietet. Auch dann hast du keinen Optimierungsaufwand verschwendet.

Ich kann mir vorstellen, dass diese "Lösungsansatzdenke" für dich noch ein bisschen abstrakt ist. Deshalb ein weiteres Beispiel, bevor es an die nächste Entwurfsphase geht:

> **Party time again!**
>
> *"Ich bin Veranstalter von 2-3 Partys pro Woche, die von 50-100 Gästen besucht werden. Solche Partys veranstalte ich in 20-25 Wochen pro Jahr in den nächsten 1-2 Jahren. Die neue Version des Programms möchte ich auf meinem Laptop am Eingang der Partys laufen lassen.*
>
> *Jeder Gast soll darin seinen Namen eingeben und persönlich begrüßt werden. Wenn z.B. Roger das erste Mal eine dieser Partys besucht, wird er mit "Hello, Roger!" begrüßt. Kommt er zum zweiten Mal, heißt es aber "Welcome back, Roger!" Ab dem dritten Besuch lautet die Begrüßung "Hello my good friend, Roger!". Und ist Roger schließlich das 25. Mal auf einer Party, ist einmalig der Zusatz auszugeben "Congrats! You are now a platinum guest!*
>
> *Ich erwarte, dass ich während der Nutzungsdauer des Programms immer denselben Laptop verwende. Der wird vor Party-Beginn hochgefahren, das Programm wird einmalig gestartet für den Abend und am Ende mit CTRL-C beendet. Eine Internetverbindung besteht am Veranstaltungsort leider nicht verlässlich.*
>
> *Können Sie das Programm in dieser Weise erweitern?"*

Das ist wieder die 3. Iteration des Hello-World Programms. Schon diese Anforderungen umzusetzen war ja schwierig, solange ein Entwurf fehlt, wie ich versucht habe, in Kapitel 01 zu vermitteln. Wie könnte der jetzt also aussehen, um die Umsetzung zu vereinfachen? Oder genauer: Wie könnte sogar zunächst nur ein Lösungsansatz aussehen?

Für mich beginnt der Lösungansatz oft mit einer Sammlung dessen, was gebraucht wird. Welche "Komponenten" sind nötig? Was für Funktionseinheiten sind klar ersichtlich? Welche Subprobleme müssen gelöst werden? Im konkreten Fall gehört für mich dann auch dazu, welchem Ansatz die Persistenz folgen soll.



Auf der linken Seite siehst du ein Gedächtnisstütze. Ich habe die Benutzerschnittstelle skizziert, um mir währenddessen das Problem nochmal zu vergegenwärtigen. Natürlich ist die Benutzerschnittstelle schon Teil der Anforderungsdefinition, die du mit dem Auftraggeber zusammen erarbeitest. Doch zur Fokussierung auf den Entwurf ist es nicht schlecht, die Oberfläche dessen, was nun zu entwerfen ist, zu wiederholen und ggf. in ein anderes Format zu bringen, das dir als Entwickler taugt.

Rechts oben eine Liste der *Funktionsbereiche*. Aus den Anforderungen und der Vorstellung, wie die Bedienung des Programms sich anfühlen könnte, habe ich abgeleitet, was mindestens getan werden muss innerhalb des Programms. Dafür reicht erstmal eine Spiegelstrichliste ohne weitere Ordnung. Die ist sozusagen ein *brain dump* dessen, was dir so einfällt. Achte nicht auf Abstraktionsniveaus oder Beziehungen zwischen diesen Funktionsbereichen. Mir ist eingefallen:

- Irgendwie muss der Name erfragt werden. Das ist eine “Kompetenz”, die im Programm ausgebildet werden muss. Die kann z.B. leere Namen abweisen und erneut auffordern, wenn das gewollt sein sollte.
- Irgendwie muss dann auch die Liste der Gäste geführt werden, in der gezählt wird bzw. aus der abgelesen werden kann, wie oft ein Gast (identifiziert über seinen Namen) schon da war.

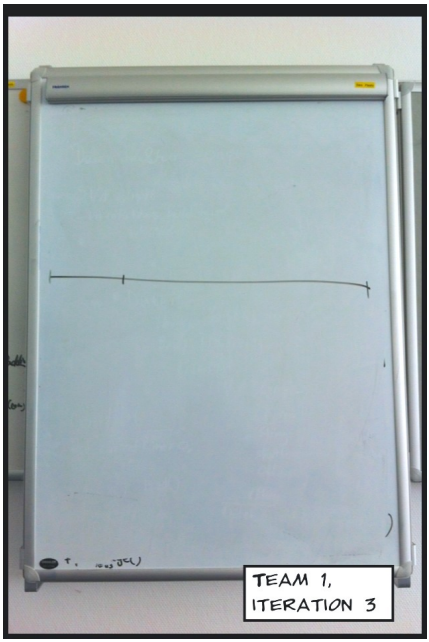
- Und irgendwie muss der Name inkl. seiner Besuchszahl in eine konkrete Begrüßung überführt werden, die dann angezeigt wird. Jenachdem, wie oft der Gast schon da war, wird hier entschieden, mit welcher Formel er begrüßt wird.

Weniger geht nicht, finde ich. Diese Funktionsbereiche stechen für mich als eigenständig heraus.

Zum Abschluss dann noch eine Idee, wie die Gästeliste über die Laufzeit des Programms hinaus persistent gemacht werden könnte. Mir scheint, dass dafür eine simple Textdatei im CSV-Format ausreicht. Zu jedem Namen wird darin vermerkt, wie oft der Gast schon da war. Kommt der Gast wieder, wird sein Besuchszähler erhöht.

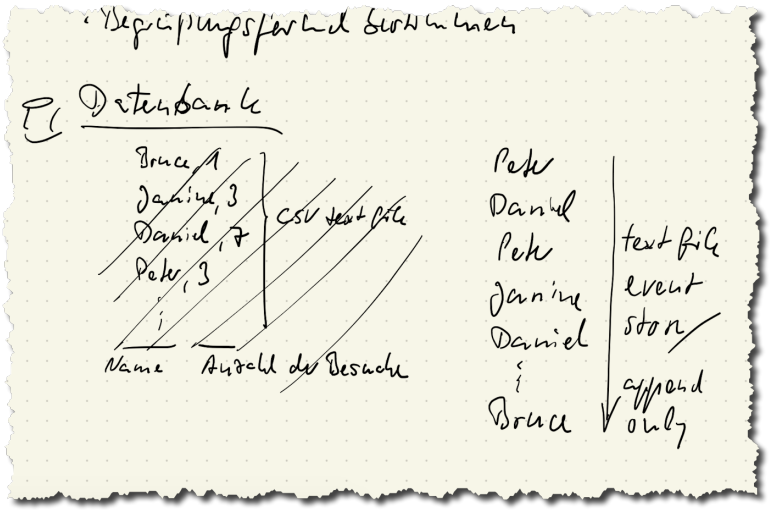
Vielleicht hast du eine ähnliche Idee gehabt für einen Lösungsansatz, vielleicht auch nicht. Wichtig ist nicht, dass er genau so aussieht wie meiner, sonder dass es überhaupt einen gibt. Jeder Lösungsansatz auf einem Blatt Papier (oder in einer iPad-App wie *Notability* oder *GoodNotes*) ist besser als keiner. Denn mit jedem aufgezeichneten, d.h. externalisierten, explizierten und visualisierten Lösungsansatz bist erstens du dir selbst klarer über die Lösung geworden und zweitens kannst du jetzt anfangen, den Lösungsansatz mit anderen zu diskutieren. Wenn du mit Kollegen das Problem angehst, dann hast du die Lösung auf einem Medium, das ihr teilt. Darauf könnt ihr beide schauen, daran könnt ihr beide arbeiten.

Aber auch wenn ich sage, dass es für den Lösungsansatz keine spezielle Form gibt, weil du möglichst frei sein sollst, deiner Kreativität Raum zu geben, gibt es Grenzen der Nützlichkeit. Hier ein Bild aus einem Clean Code Workshop. Diesen Lösungsansatz hatte ein Team am Whiteboard zurückgelassen, als es sich an die Codierung gemacht hat. Bei aller Offenheit für persönlichen Stil und individuelle Darstellungen ist mir das dann doch zu wenig.



Also: Beim Lösungsansatz geht eine Menge. Mach dir keinen Kopf, "es richtig zu tun". Wichtiger ist, dass du es überhaupt versuchst und einen Ausgangspunkt für den nächsten Entwurfsschritt schaffst. Doch achte darauf, dass ein substanzieller Bezug zum Problem *zu sehen ist.* Nur dann kann sich ein *gemeinsames* Modell entwickeln, weil alle von demselben Bild ausgehen. Ansonsten hängen nur Worte in der Luft, unsichtbar und flüchtig. Auf die kannst du ungleich schwerer Bezug nehmen. Deren Interpretation geht schnell auseinander.

Ein zu Papier gebrachter, expliziter Lösungsansatz hilft dir auch zu iterieren. Sei nicht mit der ersten Idee zufrieden. Wenn du sie vor dir siehst (oder auch nur versuchst, sie vor dich hinzustellen), kann es sein, dass du erstmalig merkst, dass die Idee doch noch nicht so gut ist. Schwierigkeiten in der Visualisierung im Speziellen oder Erklärung mit Worten im Allgemeinen sind ein gutes Signal für dich, weiter darüber nachzudenken. Und so kann es sein, dass du auf einen neuen Lösungsansatz kommst - den du selbstverständlich ebenfalls zu Papier bringst. Hier ein Beispiel für eine Revision der Persistenzidee für das Hello-World Beispiel:



Die Funktionsbereiche haben sich nicht verändert. Doch das Persistenzformat gefällt mir nicht mehr. Warum die Namen zusammen mit Besuchszählern speichern? Dann muss die Gästeliste immer komplett neu geschrieben werden, wenn es eine Textdatei ist, obwohl sich nur ein Wert verändert hat. Oder ich müsste zu einer echten Datenbank greifen, die einen gezielten Zugriff auf nur einen Datensatz bietet. Oder ich müsste die Namen statt in einer Datei auf viele verteilen, die ich getrennt aktualisieren kann.

Viel einfacher scheint mir jedoch, bei einer Datei zu bleiben, an die Namen allerdigs nur angehängt werden. Jeder Name taucht darin dann so häufig auf, wie der Gast auf einer Party war. Das kann jederzeit gezählt werden.

Diesen Ansatz nenne ich *Event Store*, weil jeder Besuch ein Ereignis ist, das minimal mit dem Besuchernamen dokumentiert wird. Das ist ein total flexibler Ansatz, der ohne Schema auskommt.[31]

Wie auch immer der Ansatz aussieht, das Wesentliche ist, überhaupt Klarheit über deine Lösungsidee zu bekommen. Versuche Anforderungen

[31]Dass der Ansatz nicht genügend skaliert, auch wenn es tausende, gar zehntausende Besucher gibt, ist nicht zu fürchten. Festplatten und Prozessoren sind schnell genug, um selbst für jeden Besucher eine solche Datei komplett zu laden (was nicht einmal sein müsste). Aber falls du das bezweifeln solltest, probiere es schnell aus. In wenigen Zeilen kannst du deine Hypothese mit einer spike solution überprüfen. Schließe einen Lösungsansatz, der Vorteile hat (hier: Flexibilität), nicht aufgrund nur eines Gefühls aus, dass die Nachteile überwiegen könnten. Mache lieber ein Experiment.

nicht sofort in Code umzusetzen. Versuche nicht einmal, eine Lösung für die Anforderungen sofort mit einem formalen Modell zu beschreiben. Nein, nimm dir die Zeit, die Lösung "im Freistil" zu erarbeiten. Deiner Kreativität sollen dabei keine Grenzen gesetzt werden. Dadurch wird der Lösungsansatz sehr wahrscheinlich auch schon ganz natürlich deklarativ. Betrachte in ihm Verhalten und/oder Daten, wie du magst. Wie es dir taugt, um Klarheit zu bekommen. "Ah, ja, so kann es gehen!" sollst du am Ende ausrufen. Dann bist du bereit für den nächsten Schritt.

Lediglich auf Papier musst[32] du deinen Lösungsansatz früher oder später bringen. Das ist ein erster Test, ob er etwas taugt. Denn wenn du ihn nicht mal "im Freistil" auf Papier beschreiben kannst, wie willst du das später im Korsett des Codes schaffen?

2. Das Modell

Der Lösungsansatz ist notwendig, aber nicht hinreichend als Entwurf. Mit ihm hast du zwar eine Lösung erarbeitet, die kannst du nur nicht geradlinig in Code übersetzen. Damit kommen wir zum Modell: Das Modell ist die Lösung in solchermaßen formalisierter Form, dass dir danach die Codierung der Lösung leicht(er) von der Hand geht.

Jede Phase im Softwareentwicklungsprozess, den ich dir im Rahmen von *Programming with Ease* empfehle, hat einen sehr konkreten, engen Zweck:

1. Die Anforderungsanalyse baut bei dir **Verständnis** für ihr Problem in einer Form auf, die konkret und testbar ist. Das Ergebnis sind Funktionen mit zugehörigen Testfällen.
2. Der Entwurf findet eine **Lösung** für das Problem, das du nach der Analyse verstanden hast. Das Ergebnis ist ein Modell.
    1. Zunächst erarbeitest du die Lösung in Form eines Lösungsansatzes. Das ist **informell**, sehr abstrakt, möglichst visuell und "auf Papier".

[32] Naja, du "musst" nicht. Aber ich lege es dir sehr, sehr ans Herz. Die Vorteile eines solchen Ausdrucks sind zu vielfältig, als das du sie in den Wind schlagen solltest. Am Anfang mag es dir schwerfallen. Mit der Übung wird es dann leichter. Am Ende kannst du dir eine Programmierung ohne "verbildlichte" Lösungsansätze nicht mehr vorstellen.

    2. Anschließend konkretisierst und **formalisierst** du den Lösungsansatz. Das Abstraktionsniveau sinkt etwas, die Lösung wird feiner ausgearbeitet, dennoch ist das resultierende Modell deklarativ und "codefrei".
3. Die Codierung übersetzt die entworfene Lösung in **Hochsprachencode**. Das Ergebnis sind Produktions- und Testcode.
    1. In der Codierung schreibst du zuerst einen **Test**, um dir eine Latte aufzulegen, über die du springen willst. Mit *test-first* vergisst du nicht, Tests zu schreiben, und du weißt sofort, wann du fertig bist mit dem Produktionscode.
    2. Der **Produktionscode** ist die Übersetzung des Modells in eine Form, die dem Kunden am Ende nutzt. Die Funktionen und Beziehungen aus dem Modell übersetzt du in Code. Anschließend füllst du die Funktionen mit Logik an, so dass tatsächlich Verhalten hergestellt wird. Das ist nun die imperative Lösung des Problems, das die Anforderungen aufgeworfen haben.
    3. Immer wieder refaktorisierst du Produktions- und durchaus auch Testcode, um das, was trotz eines guten Modells und geradliniger Übersetzung unsauber geworden ist, wieder in eine zukunftsfähige **Ordnung** zu bringen. Das passiert immer mal wieder und ist nicht schlimm. Gelegentlich weichst du auch vom Modell ab, weil du in der Codierung neue Erkenntnisse gewinnst.

Für gegebene Anforderungen ist das mehr oder weniger ein Wasserfall. Du durchschreitest diese Phasen von 1. bis 3.3. in dieser Reihenfolge. Theoretisch jedenfalls, denn praktisch gibt es darin Schleifen bzw. Rückwärtsschritte: Du gehst von der Modellierung zurück zum Lösungsansatz, weil du bei der Konkretisierung bemerkst, dass irgendetwas noch fehlt. Du gehst womöglich vom Lösungsansatz zurück zur Anforderungsanalyse und sprichst mit dem Auftraggeber, weil du bemerkst, dass dir irgendetwas noch unklar ist. Du "drehst dich im Kreis" innerhalb der Codierung während der Übersetzung eines Modells; das gehst du Funktion für Funktion mit 3.1, 3.2 und 3.3 an.

Der Wasserfall ist also entschärft. Keine Sorge, du musst keinen "Agilitätseid" brechen. Außerdem gilt der Wasserfall nur für die anliegenden Anforderungen. Es gibt keine Not, alle Anforderungen erst komplett zu analysieren. Du kannst einen beliebig kleinen Ausschnitt wählen. Manchmal rauschst du den Wasserfall in vier Stunden herunter, manchmal in

einem Tag, manchmal in zwei Tagen. Länger sollte es zumindest für Entwurf + Codierung nicht dauern. Ich glaube fest daran, dass Inkremente nicht mehr als 16 Stunden für die Umsetzung brauchen sollten, d.h. z.B. von heute 9:00 Uhr bis morgen 17:00 Uhr.[33]

Modellarten

Das Modell formalisiert den Lösungsansatz. Es konkretisiert, was der Lösungsansatz mehr oder weniger grob angedacht hat. Was bisher vielleicht nur verschwommen zu sehen war, muss nun geklärt werden. Das betrifft beide Seiten jeder Lösung: das Verhalten und die Daten. Es gibt daher zwei Arten von Modellen:

- Das **Verhaltensmodell** beschreibt, was getan werden muss, um das Problem zur Laufzeit zu lösen.
- Das **Datenmodell** beschreibt, mit welchen Daten etwas getan werden muss.

Software weißt insofern eine grundsätzliche Dualität auf. Verhalten und Daten sind deren gegenüberstehende Seiten und ergeben zusammen das Ganze. Ohne Daten kein Verhalten, ohne Verhalten keine Daten.

Allerdings hat einer dieser Aspekte für mich Priorität: das Verhalten. Dafür wird Software gemacht! Eine Problemlösung besteht immer in

[33]Eine solche Arbeitsweise nennen ich *spinning* und habe den gleichnamigen Workout im Fitness-Studio im Sinn. Dass in 16 Stunden nicht unbedingt Wert für den Auftraggeber hergestellt werden kann, ist mir klar. Doch das ist auch nicht der Zweck, so kleiner Inkremente. Nach 16 Stunden soll vielmehr eine Umsetzung vorliegen, zu der der Auftraggeber "nur" Feedback geben kann. Du willst als Programmierer einfach nicht länger als zwei Tage im Ungewissen sein, ob das, woran du arbeitest in die richtige Richtung geht. Und auch der Auftraggeber sollte nicht länger im Unklaren sein, ob du ihn verstanden hast bzw. ob er überhaupt in Auftrag gegeben hat, was er wirklich braucht. Aus mehreren Feedback-Inkrementen setzt sich dann ein Wert-Inkrement zusammen. Wann Wert für den Kunden entsteht, ist nicht dein Job zu beurteilen. Mit jeder Umsetzung produzierst du lediglich Qualitätscode in jeder Hinsicht in Bezug auf die dir vorliegenden Anforderungen. Das ist wirklich alles. Doch das ist schwierig genug. Belaste dich also nicht noch mit der Wertfrage.

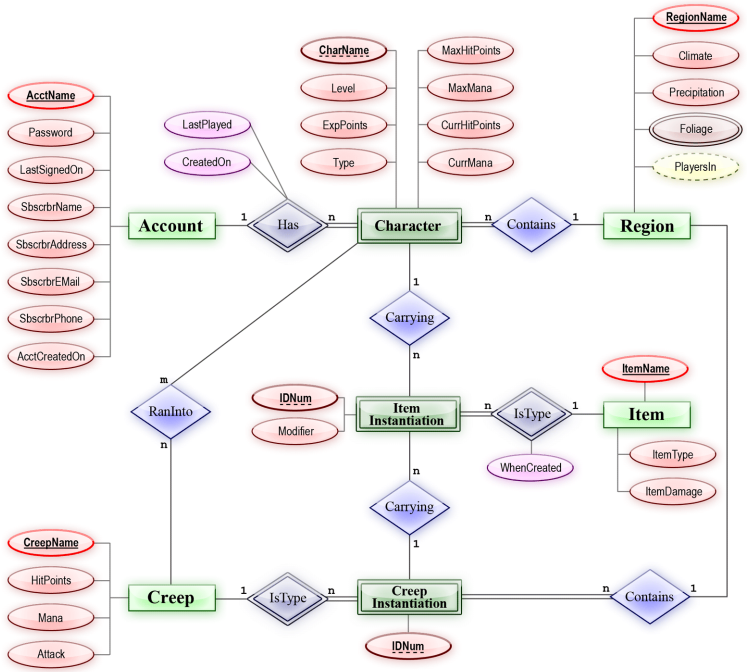
Verhalten, das Daten transformiert.[34]

Datenmodelle

Die mainstream Objektorientierung wie auch lange Jahrzehnte sehr begrenzter Hauptspeicher und daraus folgend eine Wichtigkeit von Datenbanksystemen haben aus meiner Sicht viele Softwareteams dazu verleitet, zuerst und vor allem über Datenmodelle nachzudenken. Für diesen Aspekt hat Entwurf eine gewisse Akzeptanz und Sichtbarkeit behalten. Ich nehme an, dass auch du schon z.B. von Entity-Relationship (ER)-Modellen[35] gehört hast:

[34]Oder sogar noch genauer: Software wird für größere Effizienz entwickelt. Mit genügend Ressourcen wie Zeit oder Menschen können die Probleme, die Software löst, auch ohne Software gelöst werden. Es geht also nicht primär um Funktionalität. Auftraggeber versprechen sich von Software vielmehr, dass sie effizienter ist als softwarelose Alternativen. Der Effizienzgewinn kann in performanterer Funktionalität liegen oder in benutzerfreundlicherer oder sicherer usw. Es geht um den Komparativ erkenntlich am "-er" der Effizienz-Adjektive. Manchmal scheint es zwar auszureichen, nur eine Funktionalität in Software zu gießen, um sie schlicht jederzeit für Anwender verfügbar zu haben. Aber auch damit ist eine Effizienz gemeint.

[35]https://en.wikipedia.org/wiki/Entity%E2%80%93relationship_model



Beispiel für ein Entity-Relationship Datenmodell aus Wikipedia

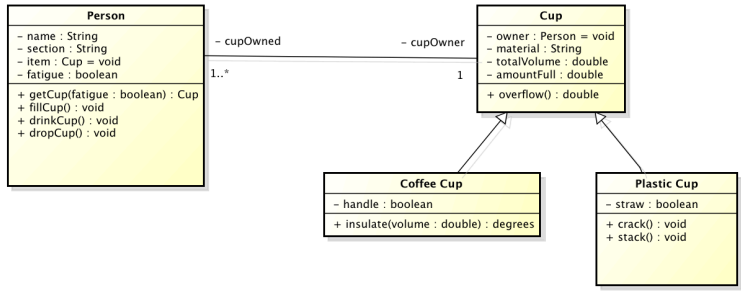
Diese Darstellung ist "nur" ein Modell, weil sie kein Code ist. Weder siehst du programmiersprachliche Anweisungen, um eine solche Datenstruktur in einer Datenbank herzustellen, noch siehst du Klassen, die sie in-memory darstellen könnten, noch ist überhaupt klar, wie die einzelnen Elemente des Modells implementiert werden.[36]

Oder hier ein anderes Datenmodell von einer Seite[37], die UML (Unified Modelling Language)[38] Klassendiagramme vorstellt:

[36]Die Implementierung des ER-Modells könnte mittels eines RDBMS oder auch einer Dokumentendatenbank geschehen. Dass und welche Relationen existieren, erzwingt kein RDBMS, auch wenn das lange die erste Wahl gewesen sein mag.

[37]https://datamodelprototype.wordpress.com/2014/01/30/uml-modeling-class-diagrams/

[38]https://en.wikipedia.org/wiki/Unified_Modeling_Language



Ein Klassendiagramm als Datenmodell

Bei Datenmodellen geht es darum, *Datenelemente* zu benennen, Daten zusammenzufassen und Zusammenfassungen in Beziehung zu stellen. Diese Daten tun nichts, vielmehr wird mit ihnen etwas getan. Das ist die Aufgabe von Verhalten.

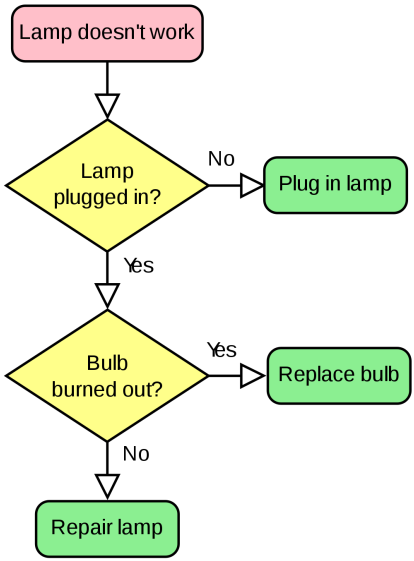
Im weiteren werde ich nicht viel zu konkreten Datenmodellierungsansätzen sagen. Ich entwerfe und benutze Datenmodelle einfach in der einen oder anderen Form. Datenmodellnotationen sind bei aller Unterschiedlichkeit der Darstellungen doch so einfach und naheliegend und weit verbreitet, dass ich mir weitere Ausführungen erspare. Ich bin gewiss, dass du mit der Datenmodellierung keine Schwierigkeiten haben wirst, wenn du erstmal weißt, um welche Daten es bei einer Lösung geht.

Das allerdings ist wiederum innerhalb der Lösungsfindung ein Problem, mit dem wir uns befassen wollen. Zu oft wird nämlich für meinen Geschmack eine vorzeitige Optimierung im Hinblick auf das Datenmodell vorgenommen. Das Datenmodell wird gesetzt und daran muss sich dann das Verhalten anlagern. Für mich steht auf diese Weise jedoch die Entwurfswelt auf dem Kopf!

Verhaltensmodelle

Wie gesagt, Priorität hat für mich das Verhaltensmodell. Im Verhaltensmodell wird beschrieben, was passieren soll. Man könnte sagen: Verhaltensmodelle drehen sich um Verben, Datenmodelle um Substantive. *Funktionseinheiten* werden benannt, zusammengefasst und Zusammenfassungen in Beziehung gesetzt. Diese Funktionseinheiten tun etwas mit Daten.

Vielleicht fällt dir sogar eine Art Verhaltensmodell ein: lange Zeit war das Flowchart[39] sehr beliebt.



Ein Flowchart nach Wikipedia als Verhaltensmodell

Das sieht konkreter aus als die bisherigen Lösungsansätze, oder? Das kannst du "runterprogrammieren", oder?

Ich halte das aber für kein nützliches Modell. Es abstrahiert für meinen Geschmack zu wenig von den Mitteln einer Hochsprache. Ein Flowchart stellt einen Kontrollfluss dar wie Hochsprachencode. Es enthält Fallunterscheidungen und vor allem auch Schleifen wie Hochsprachencode. Diese Art der Darstellung von Verhalten bietet schlicht keine Skalierbarkeit. Du kannst damit keine größeren Lösungen beschreiben: das resultierende Diagramm ist dann genauso wenig verständlich wie Hochsprachencode.

Flowcharst stammen aus einer Zeit vor der Strukturierten Programmierung[40]. In ihnen sind beliebige Verzweigungen (lies: Sprünge) erlaubt. Es gibt keine wirklich beschränkende Syntax. Sie sind mithin wenig hilfreich in der Praxis - auch wenn sie hier und da bei sehr begrenzter Funktionalität

[39] https://en.wikipedia.org/wiki/Flowchart
[40] https://de.wikipedia.org/wiki/Strukturierte_Programmierung

mal zum Einsatz kommen können.

Für mich gibt es zwei Kategorien von Lösungen: algorithmische und prozessurale. **Algorithmische Lösungen** bewegen sich sehr nah an den Mitteln der Strukturierten Programmierung von Programmiersprachen. Du bist versucht, sie mit Pseudocode oder Flowcharts zu modellieren. Der Lösungsansatz für das Sortierproblem fällt in diese Kategorie.

Als Beispiel für Pseudocode die obige Lösung für die Behandlung einer nicht funktionierenden Lampe:

```
If lamp is plugged in then
        if bulb is burned out then
                replace bulb
        else
                repair lamp
        end if
else
        plug in lamp
end if
```

Den Code verstehst du, auch wenn er zu keiner speziellen Programmiersprache gehört. Er ist eine Verallgemeinerung dessen, was in vielen Sprache an Mitteln vorhanden ist. Deshalb lässt er sich schnell hinschreiben; du musst auf keine syntaktischen/semantischen Feinheiten achten. Hauptsache er liest sich flüssig.[41]

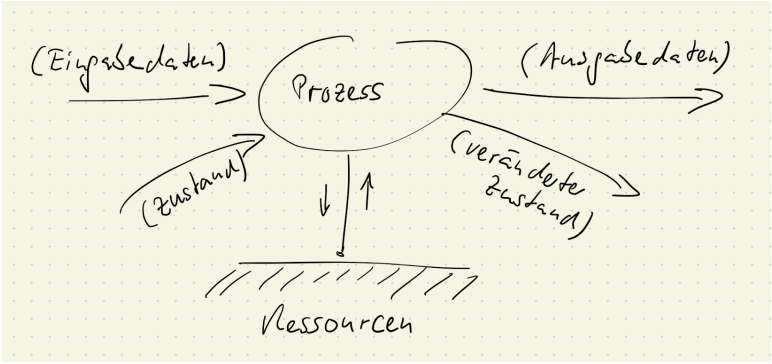
Algorithmische Lösungen sind am Ende aber die unkritischen. Sie müssen gefunden werden, klar. Doch ihr Umfang ist gewöhnlich vergleichsweise klein. Ich sage mal etwas flapsig: Der Code rein algorithmischer Lösungen passt handschriftlich auf eine DIN A4 Seite. Das ist kein Umfang, der für langfristig hohe Produktivität eine große Hürde darstellt. Solange die Logik für eine algorithmische Lösung fokussiert in einer Funktion steht und keine funktionalen Abhängigkeiten bestehen, wirst du dir ein Verständnis erarbeiten können. Debugging hilft im Zweifelsfall.

Natürlich ist das eine Vereinfachung. Mir gehts hier aber um das big picture. Wenn du auf einen "algorithmischen Kern" in einem Problem gestoßen bist, dann modelliere mit einem Flowchart. Nur vermute ich, dass du zu früh gewiss bist, dass ein Problem schon algorithmisch ist.

[41] Allerdings: Schau genau hin! Hättest du es lieber gehabt, die Lösung zuerst als Pseudocode oder zuerst als Flowchart präsentiert zu bekommen? Ich finde eine visuelle Lösung in den meisten Fällen besser zu überblicken. Sie ist zweidimensional, was Fallunterscheidungen und Schleifen zugute kommt. Im textuellen Pseudocode muss alles linearisiert werden.

Du machst es dir damit zu schwer, ein Modell (oder später den Code) zu finden.

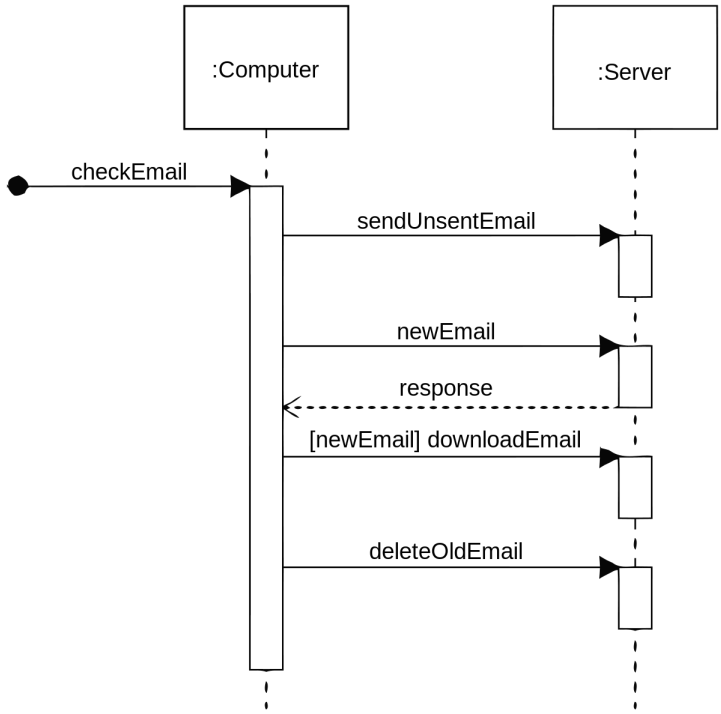
Ich glaube, dass die meisten Probleme zuerst und vor allem **prozessuale Lösungen** brauchen. In denen findest du dann keine Fallunterscheidungen oder Schleifen, sondern lediglich Funktionsaufrufe. Die stehen für Schritte in einem Prozess, der das gewünschte Verhalten erzeugt durch die Transformation von Eingabedaten in Ausgabedaten unter Verwendung von Zustand und Ressourcen.



Auch ein Verhaltensmodell: ein Datenflussmodell für die allgemeine Funktionsweise von Software-Funktionseinheiten

Algorithmische Modelle sind schon sehr nah am Wie. Prozessuale Modelle hingegen zeigen vor allem das Was. Hier als Beispiel ein UML Sequenzdiagramm aus Wikipedia[42]:

[42] https://en.wikipedia.org/wiki/Sequence_diagram



Ein Sequenzdiagramm als Verhaltensmodell

Die durchgezogenen Pfeile stellen "Funktionsaufrufe" dar. Im Bild ruft also der `Computer` den `Server` mehrfach auf im Rahmen der Lösung des Problems "check mail".[43]

Das ist ein deutlich abstrakteres Verhaltensmodell als ein Flowchart. Hier geht es nicht um einzelne Anweisungen, sondern lediglich um die zu erledigenden Schritte. Wie genau ein Schritt wie `sendUnsentEmail` sein

[43] Ich sage hier so kühn, dass es sich um Funktionsaufrufe handelt. Vielleicht liest du jedoch lediglich Datenflüsse heraus: da fließen einfach Nachrichten zwischen `Computer` und `Server`. Diese Interpretation ist für mich auch ok. Nur frage ich dich: Wie werden denn diese Nachrichten verarbeitet? Von Logik! Und wo findest du diese Logik? Eingefasst in Funktionen. Das ist die Coderealität: Am Ende erzeugt Logik das Verhalten - hier z.B. Transformation einer `newEmail`-Nachricht in eine `response`-Nachricht - und diese Logik steckt besser für sich in bzw. hinter genau einer Funktion, um klar idenfizierbar zu sein mit ihrer Verantwortlichkeit.

Teilverhalten erzeugt, ist unterhalb des Radars dieses Modells. Die Logik dafür würdest du in einer *test-first* Codierung finden. Sie ist ganz bewusst kein Bestandteil des Entwurfs.

Allerdings: Solange du in der Modellierung noch das Gefühl hast, dass so eine Funktionseinheit wie der Block für `sendUnsentEmail` zu groß ist, um in der Codierung zügig mit Logik gefüllt zu werden, solange solltest du mit dem Modell noch nicht zufrieden sein. Aber dazu später mehr.

Im Verhaltensmodell liegt der Fokus auf dem *Was*. Deshalb werde ich dir im Weiteren keine Flowcharts oder Pseudocode nahelegen. In beiden steckt für mich zuviel *Wie*. Aber auch das Sequenzdiagramm werde ich nicht weiter verwenden. Es skaliert ebenfalls nicht, wenn die Prozessschritte zu vielen "Akteuren" angehören (im Beispiel z.B. `Computer`).

Dennoch geben Sequenzdiagramme einen ersten Eindruck davon, wie ein Verhaltensmodell grundsätzlich aussieht.

Abstraktion

Verhaltensmodelle und Datenmodelle beschreiben die zwei Seiten von Software: Verarbeitung und Material. Das kann in ganz vielfältiger Weise geschehen. Die obigen Modelle sind sollen dafür nur Beispiele sein, um dir das Thema Modellierung etwas fasslicher zu machen.

Wenn du dir einen Eindruck von der Bandbreite an Modellierungsansätze verschaffen willst, dann schau dir z.B. Bücher wie *UML Distilled von Martin Fowler*[44] oder *Modellierung: Grundlagen und formale Methoden von Uwe Kastens*[45] an. Du wirst erstaunt sein, wie vielfältig du Lösungen ohne Code beschreiben kannst; oder manchmal auch nur Lösungsansätze, denn einige Modellierungsmethoden ordne ich eher der vorgelagerten Phase zu.

Egal aber, welchen Ansatz du insbesondere für die im Weiteren fokussierte Verhaltensmodellierung wählen solltest, solltest du eines nicht aus den Augen verlieren: die Umsetzung in Code. Beim Blick auf einen Modellierungs- oder allgemeiner Entwurfsansatz frage ich mich immer:

[44] UML Distilled: A Brief Guide to the Standard Object Modeling Language, Martin Fowler, Addison-Wesley, ISBN 978-0321193681

[45] Modellierung: Grundlagen und formale Methoden, Uwe Kastens, Carl Hanser Verlag, ISBN 978-3446454644

"Und wo sind die Funktionen?" Denn die Funktionen sind die Container für die Logik. Und die Logik ist das, was das Verhalten erzeugt und so schwierig korrekt hinzubekommen ist. Und um sie korrekt zu erschaffen und auch zu erhalten, ist ein *test-first* Vorgehen bei der Codiernung nötig. Und dafür wiederum sind Funktionen als Ansatzpunkte zwingend.[46]

Was du als und wie du in der Entwurfsphase die Lösung findest, am Ende musst du sie in einem Modell formalisiert formulieren, das glasklar macht, welche Funktionen mit welchen Verantwortlichkeiten der Code aufweisen muss.

Diese funktionalen Atome, die alle ihren Beitrag leisten zum Gesamtverhalten, dürfen aber natürlich nicht "einfach herumliegen". Vielmehr müssen sie in Beziehung gesetzt werden, um ein Zusammenspiel zu erreichen. Im Sequenzdiagramm oben ist das der Fall:

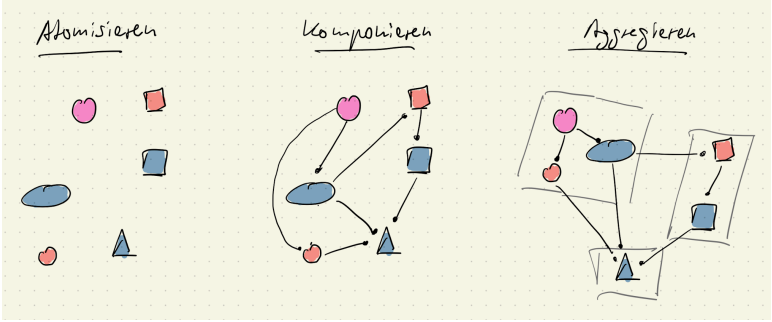
- Funktionen wie `sendUnsentEmail` oder `deleteOldEmail` sind Verhaltensatome.
- Pfeile zwischen den "Akteuren" `Computer` und `Server` setzen Funktionen in Beziehung, hier: `checkEmail` mit z.B. `sendUnsentEmail`; erstere ruft letztere auf.

Und die "Akteure" selbst setzen Funktionen ebenfalls in Beziehung. Sie fassen sie zusammen, hier: in `Server` sind `sendUnsentEmail` und `deleteOldEmail` vereint.

Modelle als konkrete, formalisierte Lösungen und Ausgangspunkte für deinen Code müssen damit mindestens Folgendes leisten:

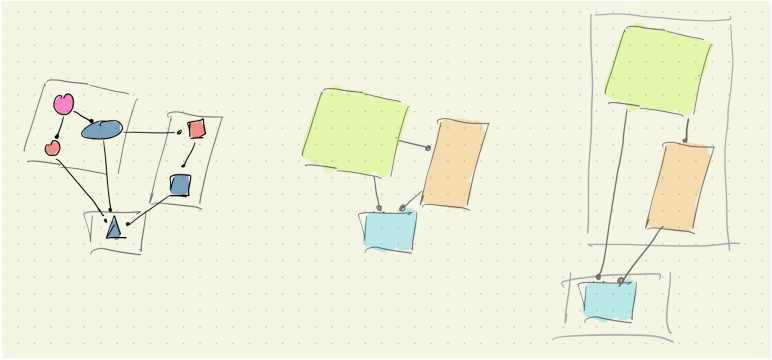
- **Atomisieren**: Die noch zu findende Logik mit Funktionen, d.h. Verhaltensatomen repräsentieren.
- **Komponieren**: Funktionen mit ihren Teilverhalten zu größerem Verhalten zusammenfassen. Aus Verschiedenem wird etwas Neues.
- **Aggregieren**: Funktionen thematisch zusammenfassen. Aus Ähnlichem wird etwas Größeres.

[46] Siehe dazu den ersten Band dieser Reihe: *Test-first Codierung*



Dass Modelle außerdem auch noch die Daten, die die Funktionen verarbeiten, beschreiben müssen, ist selbstverständlich. Wie gesagt, das halte ich jedoch für ein vergleichsweise kleines Problem und sekundär. Wenn du von der mainstream Objektorientierung geprägt sein solltest, mag dir das merkwürdig erscheinen, doch versuche einmal deine Skepsis für die folgenden Seiten auf Urlaub zu schicken.

Atomisieren, komponieren und aggregieren sind für mich Abstraktionsleistungen. Für Details, Einzelteile, Feinheiten werden Begriffe gefunden, mit denen es sich leichter umgehen lässt. Und das kann dann sogar auf beliebig vielen Ebenen stattfinden.



Das Ergebnis ist ein Abstraktionsbaum mit beliebiger Tiefe für Komposite und Aggregate.

Ohne einen solchen Baum in zwei Dimensionen - Komposition und Aggregation - bekommen wir wachsende Lösungen einfach nicht in den Griff, glaube ich. Er existiert am Ende de facto im Code – fragt sich nur,

wie es zu ihm gekommen ist. War das “Zufall”, “hat es sich ergeben”? Oder hast du ihn bewusst entworfen? Ich plädiere für Letzteres.

Plane deine Abstraktionen. Plane sie vor allem nicht allein, sondern gemeinsam mit deinen Entwicklerkollegen. Strebe nicht nur nach collective code ownership[47], wie es einmal im eXtreme Programming heißt. Ich meine, es muss auch ein *collective design ownership* geben. Ihr müsst alle zusammen hinter den Abstraktionen stehen, die die Lösung repräsentieren und formen.

Die soziale Dimension eines Entwurfs ist nicht zu verachten. Er ist ein Werkzeug für’s Denken wie für’s Kommunizieren. Deshalb ist es auch nicht so wichtig, dass ein Entwurf “für sich selbst stehen kann”. Lege deinen Lösungsansatz oder auch dein Modell nicht einfach jemandem zur Weiterverarbeitung stumm vor. Beide sind bei allem Detailreichtum “nur” Gesprächsanlässe. Entwürfe müssen für die Weiterverarbeitung mit Erklärungen übergeben werden. Am besten geschieht das im Dialog, zur Not schriftlich oder per Video. Je mehr Interaktionsmöglichkeit zwischen dem Empfänger deines Entwurfs und dir, desto besser. Denn der Empfänger wird Fragen haben. Er muss Fragen haben, weil du nie alles, was dir zu einer Lösung im Kopf herumgeht, vollständig in einem Entwurf festhalten kannst.

Zusammenfassung

Im Entwurf findest du zuerst eine Lösung und formalisierst sie dann abstrakt. Für mich gilt dabei: *Behavior first, data second.* Was das bedeutet, wirst du in den folgenden Kapiteln sehen.

Während du in der Lösungsfindung noch sehr frei bist, was den visuellen Ausdruck angeht - und visuell sollte er sein! -, engt die das Modell jedoch ganz bewusst sein. Seine Abstraktionen sollten so gestaltet sein, dass du sie leicht in Codestrukturen übersetzten kannst.

Es gibt eine Vielzahl an Modellierungswerkzeugen. Manche machen es dir schwerer, andere leichter, diese Forderungen zu erfüllen. In den folgenden Kapiteln stelle ich dir den Ansatz vor, von dem ich meine, dass er für dich

[47] https://explainagile.com/agile/xp-extreme-programming/practices/collective-code-ownership/

der erste sein sollte, durch dessen Brille du auf eine Lösung schaust, um sie zu formalisieren. Nicht der einzige, aber der erste, dein Default. Er ist breit einsetzbar und leichtgewichtig, wie ich dir hoffentlich vermitteln kann. Andere Ansätze hab gerne auch in deinem Entwurfsköcher – doch gerade um deine Lösungen in groben Strichen zu skizzieren für "das Ausmalen" in der Codierung, halte ich das *Flow-Design*, wie ich es nenne, für ideal.

Übungsaufgaben

Reflexionsaufgabe

Bitte formuliere eine Frage *oder* eine Erkenntnis zum Kapiteltext.

- Wo bist du gedanklich hängengeblieben, was ist dir unklar, was passt für dich irgendwie nicht zusammen, wozu würdest du dir noch etwas mehr Erklärung wünschen? Steht irgendetwas zu deiner bisherigen Praxis im Widerspruch und du fragst dich, warum du etwas ändern solltest?
- Oder: Wann hast du einen Aha-Moment gehabt, was ist dir als bemerkenswert, spannend, ausprobierenswert aufgefallen? Hat irgendetwas "in dir Klick gemacht", weil dir nun ein Zusammenhang aufgegangen ist? Oder verstehst du jetzt aus deiner bisherigen Praxis irgendetwas besser?

Am besten formulierst du Frage bzw. Erkenntnis schriftlich. Indem du deine Gedanken aufschreibst, wirst du dir ihrer bewusster. Bei einer Frage kommst du dadurch vielleicht schon einer Antwort aus dir selbst heraus näher. Bei einer Erkenntnis fällt dir vielleicht schon etwas ein, das du ab jetzt anders machen kannst.

Aufgabe - Lösungsansatz finden

Die SARS-CoV-2 Pandemie 2020 hat vielleicht das Interesse für Statistik in der Bevölkerung nicht erhöht, doch zumindest haben jetzt mehr Menschen von Begriffen wie Sensitivität und Spezifizität gehört und dass ein positives Testergebnis auf SARS-CoV-2 Infektion weder notwendig eine Erkrankung bedeutet, noch zwingend korrekt ist. Aus diesem Anlass folgende Aufgabe, der so genannte *Bedingte Wahrscheinlichkeiten* zugrundeliegen. Das ist Mathematik, die nicht jedem jenseits der 4. Klasse Spaß gemacht hat, doch es ist keine höhere Mathematik und lässt sich mit ein bisschen googlen zu dem Begriff gut erkunden; mehr als Grundrechenarten sind nicht nötig. Dass du dich mal mit Bedingten

Wahrscheinlichkeiten auseinandersetzt, ist ein Gewinn fürs Leben. Da bin ich gewiss.

Entwickle bitte nur einen *Lösungsansatz* für folgende Anforderungen.

> Der Auftraggeber wünscht ein Programm, mit dem Anwender bestimmen können, wie hoch die Wahrscheinlichkeit einer Erkrankung bei einem positiven Testergebnis ist.
>
> Ein Beispiel aus der Literatur:[a]
>
> > *Ein Prozent der Frauen, die sich regelmäßig einer Mammogrphie unterziehen, haben Brustkrebs. In 80% der Fälle ergibt sich für Frauen mit Tumoren in der Brust ein positiver Befund. In 9,6 % der Fälle zeigt sich jedoch auch bei gesunden Frauen ein positiver Befund.*
>
> Wie wahrscheinlich ist es nun, dass eine Frau mit positivem Testergebnis auch tatsächlich Brustkrebs hat?
>
> Nur etwas 15% der Ärzte, denen diese Frage mit den Angaben vorgelegt wurde, konnten sie korrekt beantworten. Das legt einerseits weitere Ausbildung nahe, aber auch Unterstützung durch Software kann helfen.
>
> Das Programm soll auf einer Datenbank basieren, die Prävalenzen für Diagnosen enthält, aber auch Sensitivität und Spezifizität zugehöriger Tests. Beispiel für einen Eintrag in Bezug auf das obige Mammographie-Szenario:

```
Test: Mammographie
Diagnose: Brustkrebs
Prävalenz: 0,01
Sensitivität: 0,8
Spezifizität: 0,904
```

> Der Anwender sucht nach Test oder Diagnose und gibt an, ob der Test positiv oder negativ ist. Das Programm gibt daraufhin die Wahrscheinlichkeit aus, dass das Testergebnis tatsächlich korrekt ist.
>
> Beispiel für das obige Szenario:

```
$ ergebnischeck.exe mammographie positiv
Die Wahrscheinlichkeit für eine korrekte positive Diagnose von 'Brustkrebs' ist: 0,078.
Nur bei ca. 8 von 100 getesteten Personen ist die Aussage des Tests korrekt.
$
```

[a] Ward Casscells, B.S., et al.; Interpretation by Physicians of Clinical Laboratory Results, 1978, https://www.nejm.org/doi/full/10.1056/NEJM197811022991808

Weder Modell, noch Codierung sind nötig. Mach dir also nicht zu viel Arbeit. Überlege, was hier wirklich "entwurfswürdig" nur in Bezug auf einen *Lösungsansatz* ist. Konzentriere deinen Lösungsansatz darauf. Sei so visuell wie möglich. Anhand deines Lösungsansatzes solltest du die Lösung jemand anderem leicht erklären können.

03 - Radikale Objektorientierung

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Die Welt bestehend aus Objekten?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Der Ursprung der Objektorientierung

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Wer hat's erfunden?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Die zentrale Analogie der radikalen Objektorientierung

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Principle of Mutual Oblivion (PoMO)

Unabhängigkeit

Geschlossenheit

Unidirektionalität

Ein Prinzip als Destillat

Implementationsidee

Integration Operation Segregation Principle (IOSP)

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Objekte verbinden als Verantwortlichkeit

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Ein Prinzip als Destillat

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Implementationsidee

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Philosophischer Exkurs

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Übungsaufgaben

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Aufgabe - Mit PoMO/IOSP implementieren

04 - Flow-Design mit 1-dimensionalen Datenflüssen

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

0-dimensionale Datenflüsse

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Notation

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Funktionseinheiten mit Seiteneffekten

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Implementation

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

1-dimensionale Datenflüsse

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Der Datenfluss als Scope

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Fließende Mengen

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Implementation

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Übungsaufgaben

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

05 - Flow-Design mit 2-dimensionalen Datenflüssen

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Abstraktion durch Komposition

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Stratified Design

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

2-dimensionale Datenflüsse

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Aus Operationen werden Integrationen

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Notation

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Datenflüsse als aufgemotzte Abhängigkeitsdiagramme

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Konsistenz

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Strukturierte Daten

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

n:1 Übersetzungen

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Rekursion

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Reflexion

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Übungsaufgaben

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

06 - Flow-Design mit modularisierten Datenflüssen

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Abstraktion durch Aggregation

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Physisch kategorisieren mit dem Dateisystem

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Module

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Abhängigkeiten

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Services stabilisieren

Clients immunisieren

Orthogonale Containerdimension

Die Modul-Hierarchie

Klasse - Abhängigkeiten mit Kontrakten zähmen

Hierarchische Modularisierung mit Klassen

Kriterien für die Aggregation mit Klassen

Klassen als Datenstrukturen

Abstrakte Datentypen

Kriterien für instanziierbare Klassen

Explizite Interfaces für Klassen

Namensraum - Kontraktkollisionen vermeiden

Bibliothek - Wiederverwendbarkeit ermöglichen

Paket - Abhängigkeiten stabilisieren

Paket-Funktionen als Logik

Komponente - Die Arbeitsteilung befördern

Service - Module plattformneutral machen

Wave - Softwareevolution zur Laufzeit

Die Modul-Hierarchie im Überblick

Datenflüsse modularisieren

Notation & Implementation I - Funktionen

Modularisierungsrichtung

Modularisierungskriterien

Notation & Implementation II - Daten

Wider die Primitive Obsession

Modularisierungsbeispiel

Reflexion

Übungsaufgaben

07 - Flow-Design mit 3-dimensionalen Datenflüssen

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Die wahren Übersetzungsverhältnisse

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Streams

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Einsatzgebiete für Streams

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Parallelverarbeitung mit Streams

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Implementation

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Continuation

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Event-Based Components

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Iterator

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Fallunterscheidung in der Integration

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Discriminated Unions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Polymorphie

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Warteschlange

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Reflexion

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Übungsaufgaben

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

08 - Die IODA Architektur

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Die Softwarezelle

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

System vs. Umwelt

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

“Kleiderbügelarchitektur”

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Die Membran

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Ventrale Interaktion: Portale

Dorsale Interaktion: Provider

Adapteraufgaben

“Griechische Architekturen”

Der Kern

Domänenlogik

Domänendaten

“Vitruvianische Architektur”

The Missing Concern: Integration

IOSP in der Architektur

Interactors

Application

Controller

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Der Interactor als injection point

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Processors

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

IODA: All together now!

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Übungsaufgaben

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

09 - Finale im Softwareuniversum

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Der Explizite Entwurf ist nötig

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Der Entwurf ist deklarativ

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Das Modell beschreibt Funktionen in Beziehungen

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Flow-Design im 4-dimensionalen Raum

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Orientierungshilfe für die Softwareentwicklung

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Anhang - Musterlösungen

Wenn du nachhaltig Software entwickeln willst, wie ich es mir vorstelle, dann musst du dir nicht nur ein paar Tipps&Tricks merken. Es wäre schön, wenn es so einfach wäre - doch das reicht leider nicht. Du braucht vielmehr Übung und Experimente und Reflexion. Immerhin gilt es, einige Gewohnheiten abzustreifen und Glaubenssätze zu verändern. Jedenfalls ist es mir so ergangen auf meinem Weg zu *Programming with Ease*.

Um dich zu einer solch aktiven Auseinandersetzung mit der Methode zu animieren, gehören zu den Kapiteln Übungsaufgaben. Vielleicht hast du dich an der einen oder anderen schon versucht. Das wäre super, denn dann hast du den ersten Schritt zur erfolgreichen Veränderung und zum Kompetenzaufbau schon getan.

Der zweite Schritt besteht anschließend in der Reflexion deiner Lösungen. Die kannst du allein vornehmen, indem du dich am Ende zurücklehnst und überlegst, was gut und was schlecht gelaufen ist usw. Damit bewegst du dich jedoch nur innerhalb deiner eigenen Komfortzone. Tiefer geht deine Reflexion, wenn du sie von einem Kontrast ausgehen lässt. Den möchte ich dir mit den Musterlösungen in diesem Band bieten.

Meine Vorstellung davon, wie die Übungsaufgaben gelöst werden könnten, weicht sehr wahrscheinlich von deiner ab. “Könnten” schreibe ich hier bewusst statt “sollten”, weil ich nicht glaube, dass es nur eine Lösung für die Übungsaufgaben gibt. Vielmehr gibt es eine Lösungsbandbreite, die z.B. davon bestimmt ist, wie Schwerpunkte bei der Anwendung von Prinzipien und Praktiken gesetzt werden. Hier und da würde ich zwar sagen, dass es “keine zwei Meinungen geben sollte”, doch allermeistens ist das nicht so.

Die folgenden Musterlösungen sind daher nicht *die* Lösungen. Sie sind nicht "richtig" und deine "falsch", wenn sie anders aussehen. Der Wert meiner Musterlösungen liegt nicht in einer "Wahrheit", die sie verkörpern, sondern vor allem in ihrer Andersheit.

Die *Differenz* zwischen deinen Lösungen und meinen Musterlösungen soll dich noch weiter anregen, darüber nachzudenken, warum du zu deinen gekommen bist. Hattest du etwas missverstanden oder übersehen oder sogar in gutem Willen ergänzt? Hast du einen anderen Schwerpunkt gesetzt?

Warum meine Musterlösungen sind, wie sie sind, erkläre ich natürlich. Meine Entscheidungen sind (hoffentlich) alle begründet und plausibel für dich - was jedoch nicht heißt, dass man darüber nicht diskutieren könnte. Hätte ich mich anders entschieden, wo Entscheidungsfreiheit bestand, wäre ich zu anderen Lösungen gekommen - die vielleicht näher an deinen liegen würden. Der Kürze wegen biete ich dir allerdings nur jeweils eine Musterlösung pro Übungsaufgabe - und das auch nur in einem Monolog, wie ihn ein Buch ermöglicht.

Doch eine Musterlösung ist besser als keine, würde ich sagen. Damit kannst du deine Reflexion schonmal anregen und tiefer in den Lernstoff eintauchen.

Für persönlicheres, konkreteres und dialogisches Feedback stehe ich darüber hinaus natürlich gern zur Verfügung. Melde dich jederzeit per Email oder schaue dir auf meiner Homepage[48] an, was ich dir ergänzend an Trainings und Coaching bieten kann.

Viel Erfolg und Freude bei der Lösung der Übungsaufgaben und der anschließenden Reflexion!

[48] https://ralfw.de/

Musterlösung: 01 - Die Anforderung-Logik Lücke

Aufgabe 1 - Erklären

Wie ist es dir ergangen mit der Aufgabe? Ich könnte es verstehen, wenn du dich damit schwer getan hast. Erstens überhaupt "ein Essay schreiben", zweitens die Darstellung auch noch besonders einfach in der Sprache halten. Das waren schon zwei ordentliche Herausforderungen und ich würde mich wunder, wenn du weniger als 60 Minuten dafür gebrauchst hast.

Ich selbst habe für die folgende Musterlösung auch einige Anläufe nehmen müssen. Mit 60 Minuten war es dabei nicht getan. (Aber eine größere erwartete Bearbeitungsdauer wollte ich auch nicht in der Aufgabe nennen, um dich nicht gleich abzuschrecken.)

Wie so oft, ist das Ergebnis gerade wegen der begrenzten Zeit dann etwas länger geworden. Mit mehr Zeit wäre ja Gelegenheit, Redundanzen herauszukürzen oder knappere, elegantere Formulierungen zu finden.

Andererseits ist eine Erklärung, die sich an Laien richtet, quasi notwendig länglicher, weil weniger an Begriffe und Konzepten vorausgesetzt werden kann. Man muss dann mehr mit Beispielen/Analogien arbeiten, um das Abstrakte für sie zumindest halbwegs greifbar zu machen.

Wie dein Ergebnis am Ende aussieht, ist für den Zweck der Aufgabe jedoch zweitrangig. Schön, wenn es gut lesbarer Text herausgekommen ist. Wichtiger jedoch ist aus meiner Sicht das, was vor und während dem Schreiben passiert ist.

Für eine Erklärung musstest du erstmal selbst dein Verständnis der Begriffe "Entwurf" und "Modell" aufbauen und schärfen.

Und dann musstest du für die Anforderung ELI5 dein Verständnis nochmal transformieren in eine laienverständliche Form. Du musstest auswählen und ordnen und auch noch in Worte fassen, was dir "intuitiv klar ist".

Um diesen Prozess ging es mir bei dieser Aufgabe. Ein Prozess, der beim Lernen im Allgemeinen und bei der Vermittlung von Programmierkenntnissen im Besonderen viel zu selten durchlaufen wird, finde ich. Denn auf diese Weise findet Lernen viel intensiver statt. Auf diese Weise erst eignest du dir den Stoff nämlich wirklich an. (Zumindest gilt das für Stoff, dessen Verständnis du nicht unmittelbar durch Tun überprüfen kannst.)

Nach dieser Vorrede hier nun erstmal mein Versuch:

Vom Nutzen der Modellierung für die Programmierung (ELI5)

Spielst du manchmal mit einer Puppe oder mit einem Spielzeugauto oder mit einer Dinosaurierfigur? Oder hast du womöglich sogar ein Puppenhaus oder einen Kaufladen oder Bauernhof, die du in deinem Zimmer aufbauen kannst zum Spielen?

Super, denn dann weißt du auch, was ein Modell ist. Eine Puppe ist ein Modell eines Menschen, ein Kaufladen ist ein Modell eines Supermarktes, weil Puppe und Kaufladen in vielen Dingen einem Menschen bzw. einem Supermarkt sehr ähnlich sind - aber sie sind eben viel kleiner und es fehlt ihnen auch in anderer Hinsicht so einiges.

Dennoch macht es Spaß, mit einer Puppe oder einem Kaufladen zu spielen, oder? Die haben ja auch Vorteile, z.B. dass sie dir zur Verfügung stehen, wenn du es willst. Oder ein Spielzeugauto ist viel billiger als ein echtes. Oder eine Dinosaurierfigur gibt es überhaupt, während echte Dinosaurier gar nicht mehr leben. Du könntest einen echten Dinosaurer nicht mal im Zoo besuchen, während du auf einem echten Bauernhof allerdings Urlaub mit deinen Eltern machen könntest.

Modelle gleichen dem, was sie darstellen, einerseits also sehr; wenn du damit spielst, ist es fast so, als würdest du z.B. wirklich ein Auto haben oder einen Bauernhof. Andererseits sind Modelle handlicher und günstiger. Dein Eltern können dir z.B. ein Spielzeugauto kaufen, aber für ein echtes müssten sie ganz lange sparen und du müsstest auch erstmal erwachsen werden, um es fahren zu dürfen.

Ein Modell ist also eine tolle Sache. Du kannst mit etwas spielen, was dir sonst nicht zugänglich wäre. Du kannst dir damit vorstellen, wie es wäre, z.B. einen echten Bauernhof zu haben, ohne deshalb gleich umzuziehen. Mit einer Puppe kannst du dir vorstellen, wie es wäre, ein Baby zu haben, ohne deshalb gleich wirklich ein eigenes Kind oder auch nur ein kleines Geschwister bekommen zu müssen. Modelle sind also total bequem und billig.

Wenn man nun programmiert, dann baut man im Grunde eine Art Maschine. Die besteht zu einem Teil aus einem Computer, zum anderen Teil besteht sie aus einem Rezept, das der Computer abarbeitet, um irgendetwas zu tun, z.B. als Roboter euren Rasen zu mähen oder beim Spiel auf deinem Smartphone eine Figur zu bewegen. Dieses Rezept heißt Programm oder Software.

Ein Auto ist auch eine Maschine. Für die hast du ein Modell, mit dem du dir vorstellen kannst, wie es wäre, ein richtiges Auto zu haben.

Genauso kann man als Programmierer für eine Software-Machine, die man bauen will, zunächst auch erstmal nur ein Modell herstellen; damit ist jedoch weniger der Computer gemeint, sondern vor allem das Programm, das er abarbeiten soll. Dieses Modell kann dann natürlich nicht das, was die echte Software-Maschine einmal tun soll - aber es sieht ihr eben doch ähnlich und ist viel billiger.

Mit so einem Programm-Modell hat es ein Programmierer einfacher, sich vorzustellen, wie es wäre, wenn er die spätere Software-Maschine wirklich hätte. Das ist total nützlich, weil es sehr, sehr schwierig und teuer ist, richtige Software-Maschinen zu bauen.

Einen Unterschied gibt es aber zwischen einem Spielzeugauto als Modell und einem Modell für eine Software. Das Spielzeugauto-Modell wird dem echten Auto nachempfunden; es gibt zuerst das echte Auto und dann das Modell. Beim Programmieren macht man es anders herum: Da baut man zuerst ein Modell oder auch zwei oder drei, um danach das echte Programm dem Modell nachzuempfinden.

Auf diese Weise kann sich ein Programmierer sparen, erst eine total komplizierte Software-Maschine zu bauen, um dann zu merken, dass sie doch nicht so geworden ist, wie er es gerne gehabt hätte. Besser ist es, er stellt erstmal nur ein Modell her und beschäftigt sich damit, um ein Gefühl dafür zu bekommen, wie es wäre, die echte Maschine zu haben. Wenn ihm

das nicht gefällt, macht er einfach ein neues Modell. An so einem Modell kann er sich halt eine Menge Dinge überlegen, ohne viel Arbeit zu haben: Wie soll die Maschine aussehen? Wie soll sie bedient werden? Aus welchen Teilen soll die Maschine bestehen? Wie sollen die Teile zusammengesteckt werden, damit die Maschine leicht zu bauen ist?

Natürlich möchte ein Programmierer auch mega gern mit der echten Software-Maschine arbeiten, so wie du am liebsten in einem echten Kaufladen etwas kaufen oder verkaufen würdest. Wenn der Programmierer allerdings zu früh anfängt, seine Software-Maschine zu bauen, dann hat er sich vielleicht noch gar nicht alles ausgedacht, was dazu nötig ist. Dann wäre es später voll schwierig, die echte Maschine umzubauen, wenn ihm etwas Neues einfällt, was sie können soll oder wie sie etwas anstellt. Auch deshalb ist es hilfreich, dass der Programmierer sich erstmal nur mit einem Modell beschäftigt. Das könnte ja aus soetwas wie Knete oder Lego oder auch nur Papier sein, damit er ganz schnell neue Ideen ausprobieren kann.

Vielleicht hast du das ja auch schon gemacht: Statt ein Spielzeugauto zu kaufen, hast du eines aus Lego selbst gebaut. Und als es dir nicht mehr gefallen hat, hast du es umgemodelt oder eine Rakete aus den Legosteinen gebaut.

Genau das sollten Programmierer auch tun, wenn sie eine Software-Maschine bauen müssen. Sobald sie dann ein schönes Modell haben, können sie es ja "in echt" bauen. Auf diese Weise wird es auch viel leichter, die echte Maschine zu bauen. Die Programmierer müssen sich weniger ärgern, weil sie sich vor allem während des Modellbaus vertan haben, wenn es leicht ist, etwas zu korrigieren.

Modelle machen also das Programmieren leichter und günstiger.

Reflexion

Ich fand es schwierig, eine griffige Analogie für ELI5 zu finden. Mit dem Spielzeug bin ich dann zufrieden gewesen. Vorher hatte ich es u.a. mit Rezepten und einem Bauplan fürs Haus probiert. Das fühlte sich für das Sprachniveau dann letztlich aber nicht so passend an.

Natürlich ist auch dieser Text nicht wirklich für 5jährige geeignet. Wichtiger ist, dass er möglichst wenig Jargon enthält und versucht, den Begriff *Modell* und die Vorteile des Modellierens anschaulich zu machen. Was ist

die Absicht dahinter, wenn man sich nicht negativen Konotationen und UML-Feuerwerk beirren lässt?

Diese Absicht kommt im Text hoffentlich gut rüber: Modelle erlauben es, dass man sich etwas vorstellen kann, ohne es real in den Händen zu halten. Man soll ohne großen Aufwand ein Gefühl für etwas bekommen.

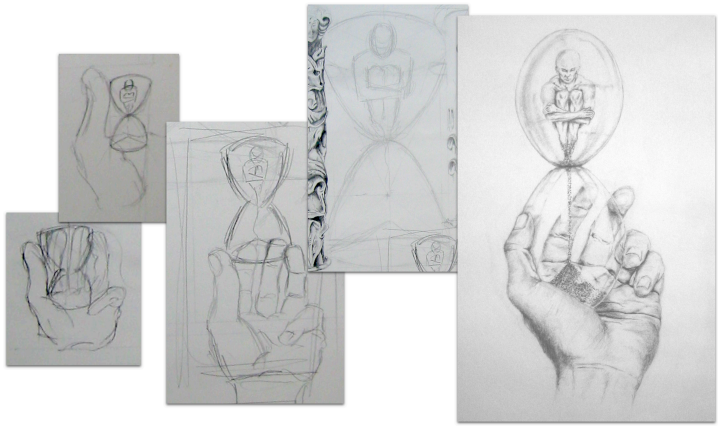
Was an Echtheit fehlt, ersetzt die Imagination. Das ist Sinn und Zweck von Modellen, weil sie dadurch eben viel, viel einfacher herzustellen sind als "the real thing". Und weil das viel, viel einfacher ist, kann man sich mehr Modelle leisten als "real things" - bzw. es sind Veränderungen an Modellen einfacher, schneller, günstiger als an "real things".

Dass Modellen viele Eigenschaften des Echten fehlen, kann mithin keine Kritik sein. Es ist vielmehr ihr Vorteil und Hauptzweck. Es geht um Abstraktion. Statt Echtheit und "high fidelity" bekommt man etwas anderes. Das sind Handlichkeit, Vereinfachung, Flexibilität und Fokus. Die sind allesamt sehr nützlich, wenn man Anforderungen in Code umsetzen soll.

Code ist so komplex, dass man sich nicht einfach auf ihn stürzen kann, sobald man meint, die Anforderungen verstanden zu haben. Besser, du machst dich als Programmierer mit dem, was der Code tun soll und wie er strukturiert sein könnte, erstmal anhand von Modellen vertraut. Du kannst dir damit einige Tränen sparen!

Aufgabe 2 - Modellieren

Beim Modellieren ist alles erlaubt - nur kein Code. Das Modell soll einerseits halbwegs "lebensecht" sein, andererseits soll ihm Wesentliches fehlen, um den Aufwand klein zu halten und sich nicht Details zu verlieren. Skizze statt Gemälde, so könnte man vielleicht sagen.



Vom Modell zum Produkt in fünf Schritten

Lösungsansatz

Ich weiß schon, dass das Programm über die Console bedient werden soll.

Außerdem ist klar, dass das Programm irgendwie die Besucherdaten über seine Laufzeit hinweg speichern muss. Der Auftraggeber will den Laptop zwischen den Parties ausschalten; ausschließlich im Hauptspeicher können die Gästedaten also nicht gehalten werden.

Auf der Festplatte könnten die Daten in einer Datenbank (z.B. RDBMS mit SQLite) gehalten werden oder sogar noch einfacher in einer Textdatei. Eine überschlägige Rechnung ergibt, dass ca. 15.000 Besucher über all die Jahre mit dem Programm begrüßt werden sollen. Das ist nicht zu viel, um sehr pauschal in einer Textdatei gespeichert zu werden. Eine 1,5MB Textdatei könnte bei Programmstart in den Hauptspeicher geladen werden. Sie könnte auch während der Programmlaufzeit ständig erweitert werden, damit keine Daten bei einem Programmabsturz verlorengehen. Wahrscheinlich könnte die Datei sogar für jeden Gast geladen werden, ohne dass eine Performanceeinbuße zu befürchten wäre. (Das kann ich mir als ein "Forschungsthema" für eine spike solution[49] merken, falls später beim Modellieren eine Entscheidung davon abhängt.)

[49] http://www.extremeprogramming.org/rules/spike.html

Das Speicherformat innerhalb einer Textdatei könnte für jeden Besucher dessen Namen und seine Besuchsanzahl festhalten.

Modell

Software hat viele Stakeholder. Deren Ansprüche an ein Modell können sehr unterschiedlich sein. Auftraggeber bzw. Anwender möchten mit einem Modell ein Gefühl dafür bekommen, wie sich der Umgang mit der Software später anfühlen wird. Für Entwickler hingegen ist es vor allem interessant, ein Gefühl dafür zu bekommen, wie die Software aufgebaut ist und ihr Verhalten erzeugt.

Es scheint mir deshalb angemessen, nicht nur ein Modell, sondern mindestens zwei zu bauen.

Oberfläche

Das Oberflächenmodell könnte im einfachsten Fall aus einem Text wie dem folgenden bestehen. Der ist zwar nicht interaktiv, aber er demonstriert das Verhalten:

```
$ helloworld.exe
Name: Janine
Hello, Janine!
Name: Peter
Welcome back, Peter!
Name:
Name: Mike
Hello my good friend, Mike!
CTRL-C
$
```

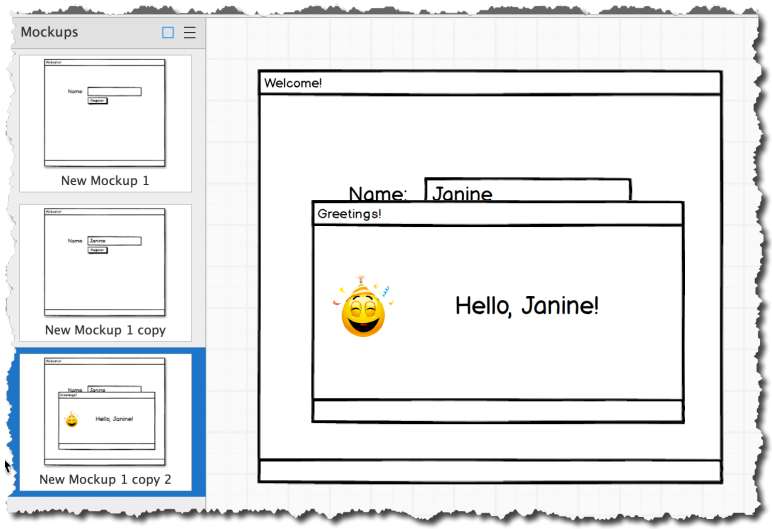
- Es ist zu sehen, dass es verschiedene Begrüßungen gibt.
- Das Layout von Abfragen, Eingaben und Ausgaben ist zu sehen.
- Ein Betrachter sieht, was passiert, wenn jemand keinen Namen eingibt.
- Es ist klar, wie das Programm zu beenden ist.

Das vermittelt ohne jeglichen Programmieraufwand schon einen guten Eindruck dafür, wie sich das Programm zur Laufzeit anfühlt.

In anderen Szenarien könnte die Codierung eines Prototyps für die Benutzerschnittstelle angezeigt sein. Das lohnt allerdings nur, wenn dessen

Code um Größenordnungen weniger umfangreich ist, als der später wirklich benötigte. UI Mockup Werkzeuge aller Art haben hier auch ihren Zweck.

Ginge es um eine graphische Benutzeroberfläche, könnte ich z.B. mit Balsamiq Mockups ein visuelles Modell so herstellen:



Dazu hätte ein Auftraggeber sicherlich eine Meinung, könnte Feedback geben - und all das, ohne auch nur eine Zeile Code zu schreiben.

Aber um diese Art Modelle geht es mir ja in diesem Buch nicht. Die liegen auf der Hand.

Interna

Das Problem bei dieser Hausaufgabe ich das Modell für dich als Entwickler, d.h. ein Modell für die Interna des Programms. Wie kann Code modelliert werden, der am Ende ja nur aus Text in Dateien besteht?

Ich könnte mit Dateien beginnen. Warum nicht aufschreiben, auf welche Dateien der Code am Ende aufgeteilt werden sollte? Technisch reicht eine einzige, z.B. `program.cs`. Aber dann ist mit einem Modell nichts gewonnen.

Aber es gibt vielleicht "Themen", deren Code in unterschiedlichen Dateien liegen könnte, z.B.

- `benutzerschnittstelle.cs`
- `datenspeicherung.cs`
- `program.cs`

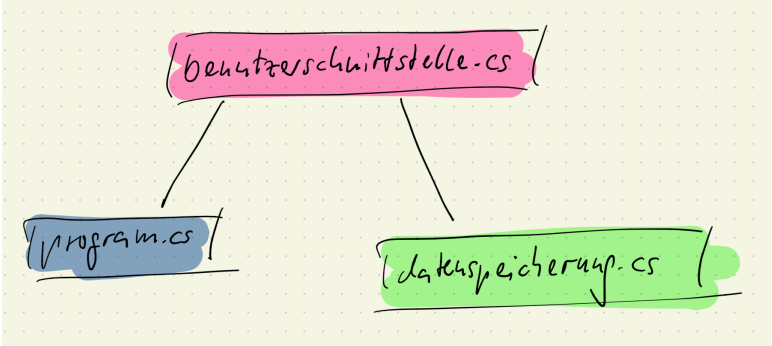
Die Ausgabe auf der Console ist etwas ganz anderes, als die Datenspeicherung. Die Logik dafür zu trennen, macht bestimmt Sinn. Und der Rest passiert dann in `program.cs`, der Datei, die es in C# sowieso gibt, um die Funktion `Main()` zu beherbergen.

Die Liste der Dateien ist ein Modell. Jemand, der die Logik schreibt, wird dadurch schon etwas angeleitet. Etwas Entscheidendes fehlt allerdings noch: die Verbindungen zwischen diesen Codebausteinen. Die Logik, die auf die Dateien verteilt wird, muss ja irgendwie zur Laufzeit zusammenarbeiten.

Eine simple Beziehung könnte schon die Reihenfolge der Dateien in einer Liste ausdrücken. Dann könnte Ausführung laut obiger Liste z.B. mit Logik in `benutzerschnittstelle.cs` beginnen, danach geht es weiter bei `datenspeicherung.cs` und schließlich in `program.cs`. Aber das hört sich nicht plausibel an, oder? Näher läge es, bei `program.cs` zu beginnen.

Außerdem soll wahrscheinlich Logik nicht nur einmal in `benutzerschnittstelle.cs` ausgeführt werden, nachdem z.B. etwas in `program.cs` passiert ist. Deshalb ist eine Reihenfolgenbeziehung zu wenig. Die Verbindungen zwischen der Logik in den Dateien müssen flexibler sein.

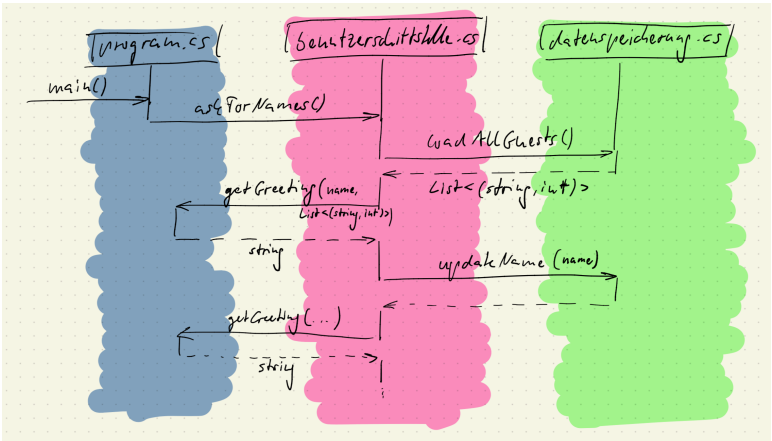
Mit Text lassen sich Verbindungen zwischen Elementen allerdings schwer flexibel und verständlich beschreiben. Besser ist es, das Medium zu wechseln und visuell zu werden.



Die Linien drücken erstmal nur aus, dass es überhaupt Beziehungen zwischen den Dateien gibt. Allerdings kann jetzt mehrfach Logik aus derselben Datei im Spiel sein.

Aber worin bestehen diese Beziehungen? Wenn die Abarbeitung von Logik nicht einfach von einer Datei in die nächste fließt, sondern hin und her, dann geht das nur über Funktionsaufrufe. Die Verbindungslinien stehen also ganz pauschal dafür, dass Logik aus der einen Datei Logik in einer anderen mittels einer Funktion aufruft, die ihr von dort bekannt ist.

Wenn das Modell hilfreich sein soll, dann braucht es also noch etwas mehr Detail in Form von Funktionsaufrufen: Welche Datei ruft wann welche Funktion in welcher anderen auf? Das kann z.B. so aussehen:



Von einer Funktion ist klar, in welcher Datei sie sich befindet: `Main()` steckt konventionshalber als Eintrittspunkt für das Programm in `program.cs`. Welche anderen Funktionen es gibt und wie sie auf die Dateien verteilt sind, ist dann meine Sache. Das festzulegen, ist ein Teil der kreativen Leistung während des Entwurfs. Die Dateinamen geben ja aber schon einen Hinweis darauf, wie ich meine, dass die Verteilung aussehen könnte.

Was du in dem "Sequenzdiagramm" siehst, ist eine Möglichkeit der Verteilung. Ich behaupte nicht, dass es die beste ist. Um die Güte geht es hier nicht, sondern darum, wie ein Modell überhaupt aussehen könnte. Nur darüber solltest du dir erstmal Gedanken machen.

Reflexion

Erfüllt das, was ich produziert habe, die Anforderungen an ein Modell?

- Ein Modell soll **deklarativ** sein. Das bedeutet, es beschreibt nicht das Wie, sondern das Was. Vor allem enthält es keine Logik. Das ist der Fall. Ich habe nicht enthüllt, wie genau die Daten dargestellt, geladen, transformiert werden. *Dass* Daten geladen werden, ist zu sehen. *Wie* Daten geladen werden, welche Logik dafür nötig wäre oder sogar welches Datenformat benutzt würde, ist nicht zu sehen.
- Ein Modell soll eine Liste von **Funktionen** liefern, die in der Codierung mit Logik gefüllt werden. Das ist auch der Fall. Mindestens fünf Funktionen sollte der Code zu diesem Modell am Ende aufweisen. Sogar deren Signaturen sind definiert.
- Ein Modell soll die **Beziehungen** zwischen den Funktionen deutlich machen. Das ist auch der Fall. Da ist die Aggregationsbeziehung, die beschreibt, welche Funktionen in einer Datei zusammengefasst werden sollen.[50] Da ist die Abhängigkeits- oder Nutzungsbeziehung zwischen den Funktionen: Welche Funktion ruft welche andere auf? Da ist die Teilenbeziehung: Welche Funktionen teilen sich Daten?[51] Das ist die Sequenzbeziehung: Welche Funktion folgt welcher anderen in der Nutzung?

[50] Der Allgemeingültigkeit über Sprachen hinweg habe ich mich auf Dateien als Aggregate von Funktionen konzentriert. Das funktioniert in Python oder Ruby oder JavaScript aus dem Stand. In C# oder Java braucht es darüber hinaus allerdings noch eine Klasse. Doch wenn die genauso wie die Datei benannt ist, dann macht sie das Modell nicht komplizierter.

[51] Das sind z.B. `loadAllGuests()` und `getGreeting()`, die beide mit der Gästeliste `List<(string,int)>` arbeiten.

Ob die Mittel der Darstellung für das Modell die besten sind, ob das Modell ein Gutes ist, sei dahingestellt. Formal erfüllt das Obige jedoch die Kriterien für ein Modell. Und ich würde sogar sagen: Selbst dieses Modell ist besser als keines.

Die Qualität des Modells besteht darin, dass es mich zwingt, mir Gedanken zu machen. Ich muss mir die Software vorstellen, wenn auch nur sehr grob. Unter Kenntnis dessen, wie ein Computer funktioniert und welche Logik mir zur Verfügung steht, kann ich im Kopf simulieren, was passieren würde. Das kann ich vergleichen mit dem, was ich an Verständnis während der Anforderungsanalyse erarbeitet habe.

Wenn dir das schwer gefallen ist, verstehe ich das gut. Auch mir ist dieses Modell schwer gefallen. Allerdings lag das daran, dass ich versucht habe, alles zu vergessen, was ich dir eigentlich sagen möchte über Modellierung. Ich habe mich zurückgenommen, um näher an deiner Situation zu sein. Du liest das Buch ja wahrscheinlich, weil du noch keine größere Erfahrung mit dem Softwareentwurf hast. Auf welche Ideen kommst du da? Das habe ich mir versucht vorzustellen.

Deshalb auch nochmal: Es kommt hier vor allem darauf an, dass du dir ernsthaft Mühe gegeben hast. Hast du versucht, die Definition für ein Modell umzusetzen? Mehr ist nicht wichtig gewesen bei dieser Übung.

Denn wenn dein Modell der Definition folgt, dann bist du weiter, als ohne. Dann hast du mehr als einen Container für Logik - aka Funktion -, was es dir schon viel leichter macht, die Logik zu finden. Logik in kleine Happen teilen, damit sie verständlicher und testbarer wird, ist das Ziel.

Dass ein Modell vieles ungesagt lässt, dass es die Lösung unterspezifiziert, ist selbstverständlich. Das gehört zu seinem Zweck und ist kein Kritikpunkt. *Dass* du modellierst, ist mithin auch nicht im Widerspruch zu jedem Anspruch der Agilität. Nur, weil du gerade nicht in die Tasten haust, bedeutet das nicht, dass du nicht dabei bist, Wert für den Kunden herzustellen. Es kommt halt aufs richtige Maß an.

Musterlösung: 02 - Entwurf im Überblick

Aufgabe - Lösungsansatz finden

Lösungsansatz für die Domänenlogik

Reflexion

Musterlösung: 03 - Radikale Objektorientierung

Aufgabe - Mit PoMO/IOSP implementieren

Modellskizze

Codierung der Integration

Codierung der Operationen

Reflexion

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Musterlösung: 04 - Flow-Design mit 1-dimensionalen Datenflüssen

Aufgabe 1 - Modellieren und implementieren

Lösungsansatz verfeinern: Prä-Modell

Modell

Implementation

Aufgabe 2 - Reverse modeling

Aufgabe 3 - Lösen, modellieren, implementieren

Lösungsansatz

Modell

Codierung

Reflexion

Musterlösung: 05 - Flow-Design mit 2-dimensionalen Datenflüssen

Aufgabe 1 - Implementation eines Modells

Reflexion

Aufgabe 2 - Die Dimensionalität eines Modells erhöhen

Reflexion

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Aufgabe 3 - Anforderungen umsetzen mit 2-dimensionalem Modell

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Verstehen

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Lösen

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Modellieren

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Zerlegen

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Verdrahten

Codieren

Reflexion

Musterlösung: 06 - Flow-Design mit modularisierten Datenflüssen

Aufgabe 1 - Datenfluss modularisieren

Schrittweise Modularisierung

Klassendiagramm

Bibibliotheken

Aufgabe 2 - Game of Life

Lösungsansatz

Modellierung

Zerlegungsbaum

Datenfluss

Modularisierung

Klassendiagramm

Bibliotheksdiagramm

Reflexion

Musterlösung: 07 - Flow-Design mit 3-dimensionalen Datenflüssen

Aufgabe 1 - Tic-Tac-Toe

Lösungsansatz

Spielerwechsel

Spielende

Das Domänenmodell

Das Domänendatenmodell

Modell

High-level Datenfluss

Port-Datenflüsse

Datenfluss-Wurzeln innerhalb von EBCs

Intergrationen verfeinern

Implementation

Inkrementelle Implementierung

Schlaglichter auf den Code

Reflexion

Musterlösung: 08 - Die IODA Architektur

Aufgabe 1 - Umbau nach IODA

Abhängigkeiten zeigen den Abstraktionsgradienten hinab

Aufgabe 2 - Enturf nach IODA inkl. Implementation

Anforderungsanalyse

Nachrichten an den Processor

Lösungsansatz

Modell

Spiel starten

Rateversuch beurteilen

Application

Implementation

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Schlaglichter auf den Code

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.

Reflexion

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/softwareentwurf-mit-flow-design.