ralf westphal

# TEST-FIRST CODIERUNG

## PROGRAMMING WITH EASE
## TEIL 1

ein buch aus dem softwareuniversum

# Test-first Codierung

## Programming with Ease - Teil 1

Ralf Westphal

Dieses Buch wird verkauft unter http://leanpub.com/test-first-codierung

Diese Version wurde veröffentlicht am 2021-12-14

# Ebenfalls von Ralf Westphal

Softwareentwurf mit Flow-Design

Software Anforderungsanalyse mit Slicing

Die IODA Architektur im Vergleich

Arbeiten im Office mit System

Produktiver und zufriedener im Office

# Inhaltsverzeichnis

# Zum Geleit

Hallo, lieber Leser! Mein Name ist Ralf Westphal und ich bin dein Guide bei der Erkundung des Thema *test-first Codierung*. Mir liegt es als ein zentraler Aspekt für saubere, nachhaltige Programmierung sehr am Herzen. Deshalb habe ich mich nach Jahren des Denkens, Tüftelns, Anwendens, Diskutierens und Unterrichtens entschieden, meine Sicht darauf einmal in einem Buch zusammenzufassen. Das, was vorher über viele Blog-Artikel und Konferenzvorträge verteilt und nur unvollständig nachlesbar manifestiert war, habe ich nun gesammelt und mit Übungsaufgaben versehen. Ich hoffe, damit gebe ich dir einen soliden Handlauf für das Selbststudium.

Gleichzeitig ist dieses Buch wie die anderen in der Reihe *Programming with Ease* aber auch Grundlage für meine interaktiven Trainings. Informationen zu denen findest du auf meiner Homepage: https://ralfw.de/

Und wer bin ich, dass ich mir zugetraut habe, dieses Buch zu schreiben?

- Seit 1986 bin ich selbstständig in der Softwarebranche tätig. Von 1988 bis 1998 als Co-Geschäftsführer einer Entwicklungsfirma für Standardsoftware in einer Handwerksbranche.
- 1998 habe ich mich dann mehr meiner Leidenschaft "Forschung und Entwicklung" hingegeben, indem ich Chefredakteur der VB-Fachzeitschrift *BasicPro* geworden bin (bis 2001). Mir lag es am Herzen, über spannende Softwaretechnologien zu berichten und andere dabei anzuleiten. Seit Anfang der 1980er hatte ich immer mal wieder in Fachmagazinen etwas beigetragen, doch nun sollte "das Weitersagen" mein Beruf werden.
- Ab 2000 war ich viele Jahre sehr in der Microsoft-Community unterwegs als sog. *Regional Director* und dann *Most Valued Professional*. Ich habe auf vielen Konferenzen in Deutschland und den USA gesprochen und auch Entwicklerveranstaltungen mit organisiert. Gleichzeitig habe ich weiter über Technologien und zunehmend Softwarearchitekturthemen geschrieben und Projektberatungen und Trainings durchgeführt.

- 2008 schließlich war das Jahr, in dem Stefan Lieser und ich angefangen haben, eine deutsche Clean Code Community aufzubauen. Angefangen haben wir mit einem Wiki - https://clean-code-developer.de/ -, in dem wir Prinzipien und Praktiken für saubere Programmierung weiterentwickelt um die Basis eines Wertesystems dokumentiert haben. In den Folgejahren haben wir viele Clean Code Developer Trainings gemacht und die Grundlagen dafür zusammengetragen, was du in diesem Buch liest. Wir haben uns intensiv und kritisch mit dem Thema Clean Code auseinandergesetzt.
- Irgendwann vielleicht ab 2018 schien mir dann der Begriff "Clean Code" zwar etabliert, doch nicht mehr ausreichend, um das zu beschreiben, wohin sich mein Denken entwickelt hatte. Aus der Clean Code Praxis der Jahre davor hatte sich schon *Flow-Design* herausgeschält als eigenständiger Ansatz zum Entwurf von sauberem Code - siehe auch das gleichnamige Buch in dieser Reihe -, d.h. einem Denkwerkzeug für die Entwicklungsarbeit vor der eigentlichen Codierung. Doch auch dieser Begriff war noch nicht umfassend genug. Zu dem, worum es seit 2008 gegangen war - nämlich einer nachhaltigen Softwareentwicklung - gehört einfach noch mehr.
- 2020 schließlich habe ich mich entschieden, die drei Säulen, die sich für nachhaltige Softwareentwicklung als notwendig und tragend bewehrt hatten, unter dem Titel *Programming with Ease* zusammenzufassen und in einer Buchreihe zu dokumentieren.

Du siehst, mein Weg hat sich vom Anwender von Softwaretechnologien über den Autor, der ihre Anwendung beschreibt, und den Trainer, der den Entwurf ihrer Anwendung vermittelt, an einen Punkt geschlängelt, wo ich nun motivieren möchte bei der ganzen Anwendung die Nachhaltigkeit nicht zu vergessen.

Weil Nachhaltigkeit so wichtig ist und aus verschiedenen Gründe sehr an grundlegenden Überzeugungen und Gewohnheiten der Softwareentwicklung und ihres Management rührt, erlaube ich mir auch den persönlichen Ton, in dem ich dich hier schon anspreche. Der mag für dich in einem Buch ungewohnt sein und eher zu Blog-Artikeln passen, doch mir liegt das Thema so am Herzen, dass ich glaube, mit einer direkteren, weniger gestelzten Ansprache es besser rüberzubringen.

Dazu kommt, dass ich damit von Anfang den Eindruck verwischen möchte, es handle sich hier um letzte Weisheiten, absolute Wahrheiten und objektive Fakten. Nein, leider kann ich all das nicht bieten und erlaube mir sogar die Behauptung, kein Autor, den du zum Thema liest, kann das, egal wie sein Name lauten oder sein Schreibstil sein mag.

Mehr als "informierte Erfahrungen im Fluss" kann ich dir nicht bieten. Was du hier lesen wirst, ist ein Schnappschuss meines Denkens. Deshalb habe ich auch lange gezögert, überhaupt ein Buch darüber zu schreiben; das Format suggeriert ein größeres Gewicht des Inhalts, als er verdient. Doch jetzt scheint mir ein Plateau in meinem Erkenntnisgewinnungsprozess erreicht, der eine solch umfängliche Aufzeichnung doch rechtfertigt.

Wenn ich dir damit helfen kann, in deinem Denken weiterzukommen, freue ich mich. Nur um Weiterkommen, nicht um Ankommen geht es.

Ich wünsche dir Erkenntnisgewinn und auch Spaß bei der Lektüre!

-Ralf Westphal

# 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 das 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 ist nötig, 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 darlege 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.

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 folgende 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 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 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*

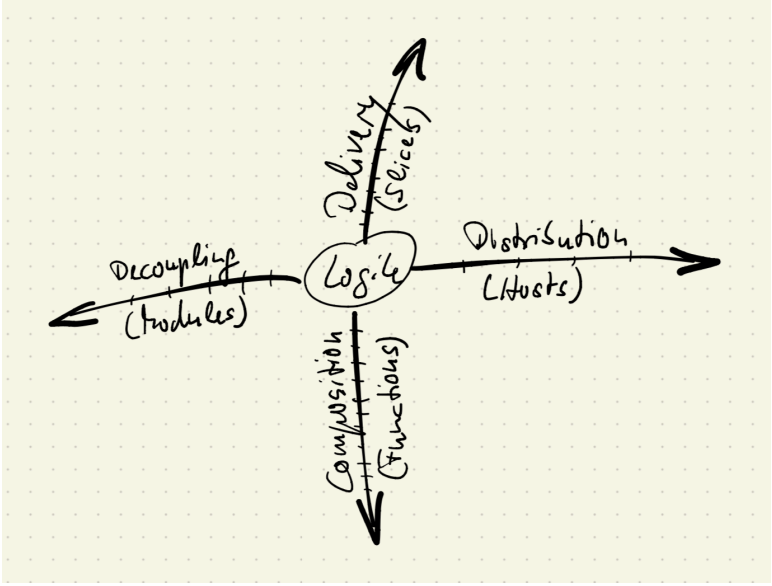
In diesem Raum spielt sich für mich alle Softwareentwicklung ab. Darin bewegst du dich mal langsamer mal schneller, mal in die eine Richtung, mal in die andere.

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 daher von höchster Wichtigkeit. 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 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.




*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 sehen, wovon ich vorher immer nur Teile gesehen hatte. Das hat mein Verständnis von Softwareentwicklung grundlegend

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

verändert.

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

# 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 bruchstückhaft 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 eingehängt werden kann.

Deshalb: Halte einen Moment durch, bis es losgeht mit der Codierung. Keine Sorge, du wirst davon genug zu sehen bekommen.

Und nun geht’s 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 der Mensch). Software soll z.B. schnell*er* oder einfach*er* oder skalierbar*er* oder sicher*er* funktionieren als die Alternative. Dann hat die **Effizienz** 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 impliziert 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 gearbeitete 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.

## 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**.

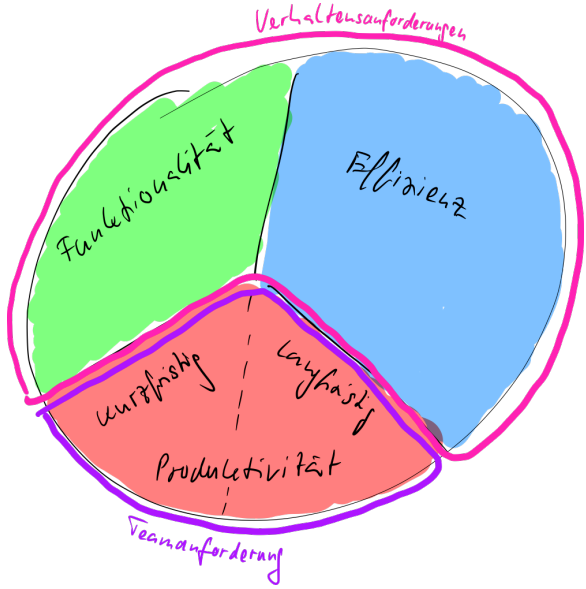
Wie Funktionalität mittels Code hergestellt wird, ist eine Sache von Programmiersprachen, Bibliotheken und Frameworks. Diese Fähigkeit ist die primäre, die du als Softwareentwickelnde(r) erwirbst und stetig verfeinerst.

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 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.

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 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 man 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”.

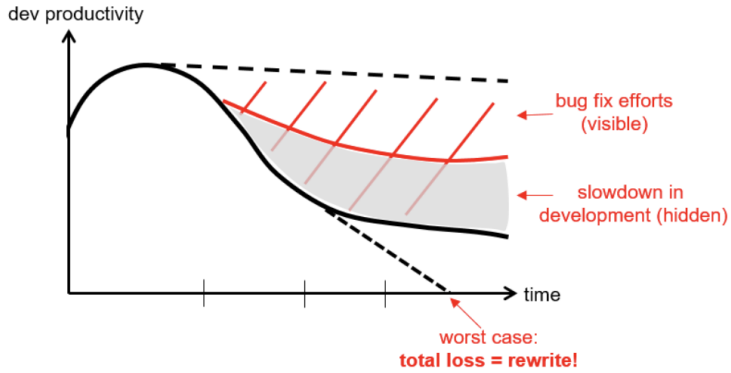



*Was der Auftraggeber will: Die Kategorien der Anforderungen*

## 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[3]):




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

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

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 das 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 noch 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"[4]. 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[5]. Besser jedoch, wenn die Softwareentwicklung gar nicht erst in diese Verlegenheit kommt. Warum nicht Bugs von vornherein einfach vermeiden?

Fehlende Korrektheit ist der Produktivitätskiller Nummer 1.

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

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

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 -, sind überraschende Qualitätsmängel unvermeidbar.

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[6] seine Meinung geändert; seine Anfor-

[6]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; das kann alle paar Wochen sein oder jeden Tag. Iterativ-inkrementelles Vorgehen ist für mich mit dem Begriff also absolut vereinbar.

> derungen sehen nun anders aus. Selbst eine korrekte Umsetzung 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.[7]

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 die 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

[7] 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!

(auch **Inkremente** genannt), die zügig umgesetzt werden können, um 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 interativ-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 Zeit, 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!

Die dann nötigen "Aufräumarbeiten" sind 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. Die Strukturierte Programmierung (structured programming[8]) ist aus dieser Erkenntnis entstanden. Man könnte wohl auch sagen, dass Objektorientierung von ihr ursprünglich inspiriert war. Ebenso das structured design[9] 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 gegeben.

Dass Robert C. Martin von *sauberem* Code und nicht von *ordentlichem* spricht, mag dem von Martin Fowler im Zusammenhang mit seinem 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 aber 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!

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

[9] 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

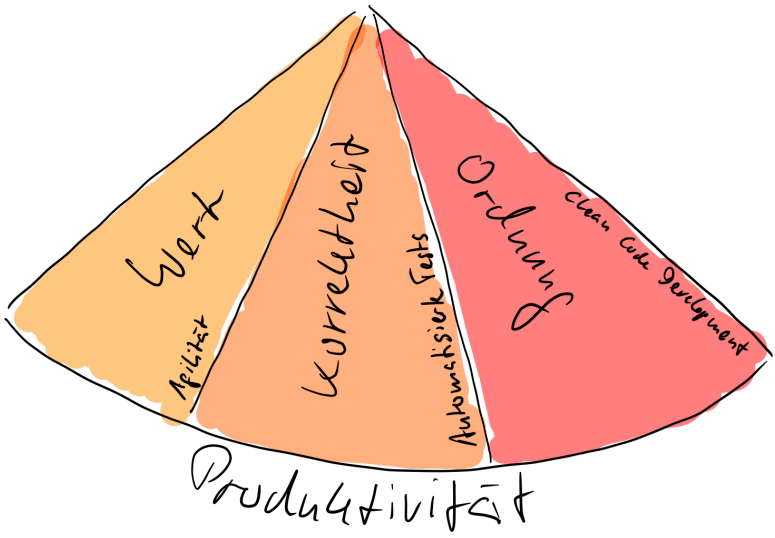
Auftraggeber wollen Software, die umfassend tut, was sie tun soll; sie soll funktional und effizient sein. Diese Qualitäten sollst du in der Softwareentwicklung stets zügig liefern; du sollst produktiv sein *und bleiben*.

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 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.**

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 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* daherhaft hohe (oder zumindest höhere) Produktivität herzustellen.

# Die Methode

# 01 - Die Anforderung-Logik Lücke

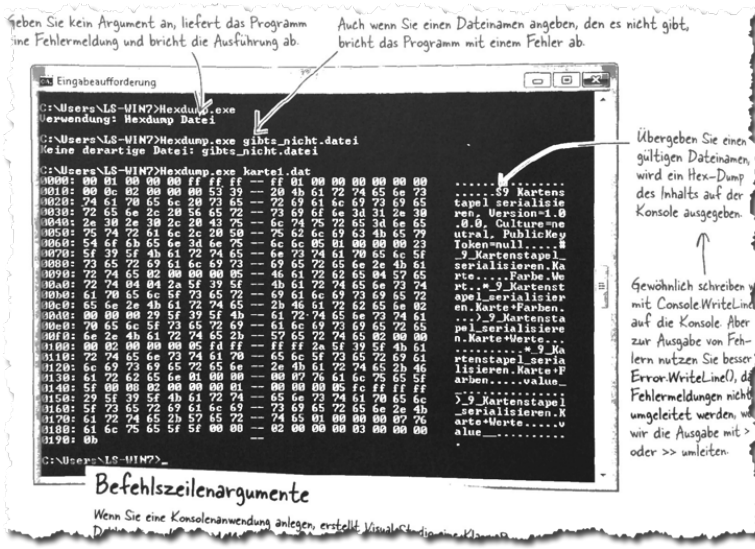
Um die Softwareentwicklung vom Kopf auf die Füße zu stellen, d.h. ihr einen Rahmen für Nachhaltigkeit zu geben, ist es hilfreich, wenn wir ihr Produkt genauer betrachten. Woraus bestehen "die Maschinen", die du in der Softwareentwicklung produzierst, von denen sich der Auftraggeber so viel Hilfe verspricht?

## 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[10]):

[10] 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 ("   ");
```

```
38              }
39      ...
```

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

Könnte `using System` gelöscht werden, ohne dass sich das Programmverhalten ändert?[11] 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.[12]

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.[13]

Logik besteht aus

- Transformationen/Operatoren, z.B. <, ++, `args.Length`

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

[12] 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.

[13] 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.

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

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 integrieren 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:

```
static int Sum(int[] numbers) {
  var sum = 0;
  foreach(var n in numbers)
    sum += n;
  return sum;
}
```

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.

## 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:[14]

[14]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[15] deiner Logik bewusst bist, gehört dazu, wenn du mit Logik den Auftraggeber 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 durch Auswahl und Anordnug von Logik erfüllen. Dann ist zusätzlich **Verteilung** gefragt, d.h. die Ausführung von Logik verteilt auf mehrere Threads gefragt.

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:

[15] 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 Sortier-Logik 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[16] 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 keinen Strukturen bestehend aus 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.

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

# Von den Anforderungen zur Logik

Angesichts des großen, bewussten und verständlichen Bedarfs an Software-Verhalten, 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.

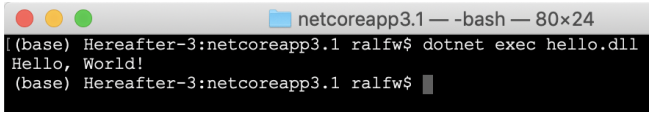
## Logik schwer definierbar

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:[17]

[17]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!");
```

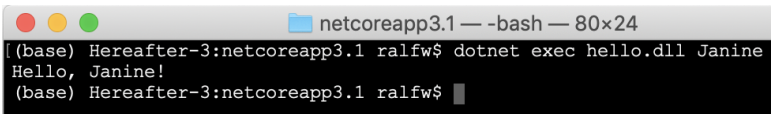
Das Programm selbst ist umfangreicher, weil noch eine Klasse und eine Funktion drumherum erforderlich sind, aber die reine Logik ist so trivial.

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.[18]

"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

[18] 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.

Erfahrung. Sie wissen doch, wie man das macht und was ich meine." Nein, weiß du nicht! Du kannst dir zwar eine Menge denken - nur bedeutet das nicht, dass es dasselbe ist, was 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 Auftraggeber 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.[19]

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 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

[19]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.

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.

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 also 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 beständig herrschenden Druck von Lieferterminen. Und genau darum geht es in den weiteren Lektionen.

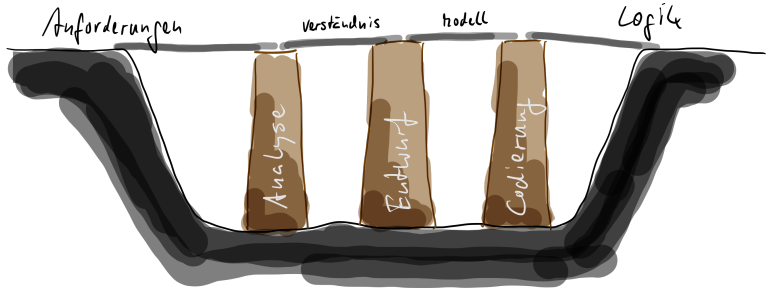
## Zusammenfassung

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.[20]

[20]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.*[21]

## 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 versteht 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 - Nachdenken

*Geschätzte Bearbeitungsdauer: 10min*

1. Stelle eine Liste zusammen mit allen Gründen, die für die Automatisierung von Tests sprechen.

[21] 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.

2. Stelle eine zweite Liste zusammen mit allen Gründen, die dafür sprechen, automatisierte Tests *vor dem* Produktionscode zu schreiben. Was spricht also für ein *test-first* Vorgehen?

Beide Listen kannst du in einer Textdatei im Repo speichern.

# Aufgabe 2 - Entwickeln

## Anforderungen

Entwickle das Programm zur Begrüßung von Party-Gästen wie in Iteration 3 spezifiziert.

## TODO: Vorgehen

1. Analysiere die Anforderungen wie bei der Anforderung-Logik Lücke beschrieben:

   *Geschätzte Bearbeitungsdauer: 10min*

   1. Definiere naheliegende Akzeptanztestfälle. (Versuche nicht perfekt/vollständig oder speziell smart zu sein. Es geht nicht darum, dass du jeden *edge case* abdeckst, sondern dir überhaupt Gedanken machst darüber, wie das Verhalten mit formalisierten Beispielen getestet werden kann.)
   2. Finde eine API-Funktion, auf die du die Akzeptanztestfälle automatisiert anwenden kannst. Sie sollte so viel Logik umfassen/repräsentieren, wie möglich - aber gleichzeitig keine übergroßen Schwierigkeiten beim Test bereiten.
2. Setze deinen Code mit “zwei Seiten” auf: auf der einen Seite steht der Testcode, auf der anderen der Produktionscode. Beide soll physisch getrennt sein.

   *Geschätzte Bearbeitungsdauer: 10min*

    1. Schreibe *zuerst* den Code für die Akzeptanztests (unter Nutzung eines Testframeworks für deine Programmierplattform[a], z.B. JUnit (Java) oder xUnit (.NET), CppTest), die du definiert hast. Wenn darin die Funktion noch von der IDE als nicht vorhanden markiert ist, macht das nichts. Du kannst sie am Ende (hoffentlich) mit einem Refactoring-Befehl der IDE automatisch generieren lassen (und verschiebst sie dann auf die Seite des Produktionscodes).

3. Nachdem du die Akzeptanztests geschrieben hast, entwickle den Produktionscode.

   *Geschätzte Bearbeitungsdauer: 20min*

    1. Strebe zuerst an, dass die Akzeptanztests "grün" sind. Konzentriere dich also auf die Logik hinter der API-Funktion aus Schritt 1.2.
    2. Wenn die Logik für die API-Funktion reif ist, fülle "drumherum" im Produktionscode die fehlende Logik hinzu, damit alles zusammen als Programm von der Kommandozeile aufrufbar ist.

Speichere deinen Code im Repo.

[a] https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks

# 02 - Vorläufig codieren im Chaos

Mit der Codierung beginnst du erst, wenn du Klarheit über einen Lösungsansatz hast. Das ist eine der wichtigsten Folgerungen aus den Phasen zur Überwindung der Anforderung-Logik Lücke. Codierung ist also nicht gleichzusetzen mit Programmierung, sondern nur ein Teil davon.

Der Lösungsansatz wird in Form eines Modells vom Entwurf geliefert, das in Beziehung gesetzte Funktionen enthält. Dazu kommen Testfälle für diese Funktionen, zu denen ja auch als Ausgangspunkt die API-Funktionen der Analyse gehören.

Konkret bedeutet das in den meisten Fällen: Die Codierung bekommt Klassen (allgemeiner: Module) und Funktionen vorgelegt, die sie in eine Programmiersprache übersetzen und dann mit Logik füllen soll.[22]

## Das Nein der Codierung

Eigentlich sollte der Startpunkt deiner Codierungsphase jedenfalls so aussehen. Eigentlich solltest du konkret mit einer Funktion(signatur) und zugehörigen Testfällen in der Hand vor dem blinkenden Cursor deiner IDE sitzen. Nicht weniger Informationen solltest du haben.

Es kann jedoch sein, dass dir die Analyse der Anforderungen kein formalisiertes Verständnis geliefert hat, was dein Auftraggeber will. Vielleicht fehlen konkrete Testfälle, vielleicht sind zwar Nutzungsbeispiele grob dokumentiert und besprochen, aber es fehlt noch eine Vorstellung über deren Repräsentation im Code durch API-Funktionen. Die Unklarheit

[22]Das mag jetzt etwas abstrakt klingen, weil ich nicht näher beschrieben habe, wie Analyse und Entwurf aussehen. Aber das Wie ist hier auch nicht so wichtig. In diesen Lektionen geht es nur um die anschließende Codierung. Es genügt für den Moment, dass du weißt, was Module/Klassen und Funktionen sind.

kann viele Formen und Ursachen haben. Wichtig ist, dass du dir ihrer bewusst bist!

Vor der Codierung kannst du also auf zwei sehr unterschiedlichen Stufen stehen:

- Mit (Akzeptanz-)Testfällen und zugehörigen Funktionen
- Ohne Testfälle und/oder zugehörige Funktionen

Diese Unterscheidung ist wichtig, denn **Produktionscode, d.h. der auszuliefernde Code, sollte nur geschrieben oder angefasst werden in größtmöglicher Klarheit**. Es besteht sonst die Gefahr unnötiger Veränderungen, die eine Belastung für die Produktionscodestruktur darstellen. Wenn du in Unklarheit den Produktionscode erst in die eine Richtung zerrst und negatives Feedback bekommst, ihn dann in eine andere richtig drückst und wieder negatives Feedback bekommt und dann nochmal durch die Mangel drehst… dann verbessert sich seine Struktur dabei nicht. Im Frust wirst du Abkürzungen nehmen und wenig geneigt sein, aufzuräumen.

Sobald du einen Lösungsansatz in Code "gießt", verhärtet er. Code, auch wenn er nur aus programmiersprachlichem Text besteht, ist schwer zu verändern. Wenn du ihn also anfasst, solltest du dir sehr sicher sein, dass und wie das nötig ist.

Ein guter Teil der Unordnung in Code, die die Produktivität so negativ beeinträchtigt, hat als Ursache "vorzeitige Codierung". Die kann als eine Art von *premature optimization* angesehen werden, also einer Variante der Wurzel allen Übels nach Donald Knuth[23].

Wer mit der Herstellung/Veränderung von Produktionscode beginnt, ohne ausreichend sicher zu sein, dass Struktur und Logik so und nicht anders sein sollten, der zielt auf schnelle Lieferung ab. Das ist eine Optimierung auf Feedback oder Wert hin – aber auch das kann selbst in einem agilen Vorgehen vorzeitig geschehen. Um dem entgegen zu wirken, lautet die klare wie pauschale Empfehlung:

Produktionscode darf nur geschrieben werden, wenn für das anliegende Issue (Feature oder Bug Fix) mindestens ein relevanter roter (Akzeptanz)Test existiert.

[23] https://en.wikiquote.org/wiki/Donald_Knuth

Ein solcher Test setzt voraus, dass Beispieldaten vorliegen und mindestens eine Funktion existiert, die den Funktionsbaum repräsentiert, in dem Logik verändert werden muss.

Sind diese Voraussetzungen für ein Issue nicht erfüllt, darfst keinen Produktionscode schreiben. Es ist schlicht nicht wirklich klar, wie das gewünschte Verhalten aussehen soll.

Deshalb musst du spätestens jetzt Nein zu einer Veränderung von Produktionscode sagen!

Ich weiß, das ist sehr schwer. Der Druck vom Auftraggeber, nach all den Gesprächen über Anforderungen endlich zu beginnen ist groß. Sollte ein iterativ-inkrementelles Vorgehen nicht die Risiken von Unklarheiten auch schon genug minimieren? Ich glaube, nein. Es gibt Unklarheit, die so groß ist, dass du sie nicht in einer "normalen" Iteration aus der Welt schaffen kannst, bei der du etwas mit Produktionscode ausprobierst. Und das Kriterium ist für mich die Existenz von codierbaren Akzeptanzkriterien. Da, so glaube ich, solltest du eine Linie ziehen und sagen "Bis hierher und nicht weiter!" Entweder, du hast mit dem Auftraggeber genügend Klarheit hergestellt in Form von Akzeptanztestfällen und API-Funktion - oder eben nicht. Wenn es dem Auftraggeber so wichtig ist, dass du produktiv wirst, d.h. Produktionscode erweiterst, dann muss er sich eben anstrengen, für sich Klarheit herzustellen, was er denn von dir will. Falls das klappt, beginnst du gern die Arbeit am Produktionscode. Falls aber nicht…

## Prototyping to the Rescue

Falls keine konkreten Akzeptanzkriterien vorliegen, bedeutet das natürlich nicht, dass du dich trotzig auf die Hinterbeine stellst und nichts tust. Mit Codierung kannst du dem Auftraggeber auch in dieser Situation der Unklarheit zu Diensten sein - nur eben nicht mit Produktionscode.

Du kannst auf andere Weise versuchen, dem Auftraggeber etwas zu geben, was ihn zu Feedback anregt. Jetzt schlägt die Stunde des **Prototypen**! Prototypen sind Werkzeuge zur Informationsgenerierung. Mit ihnen kitzelst du Klarheit aus dem Auftraggeber (und auch aus dir) heraus.

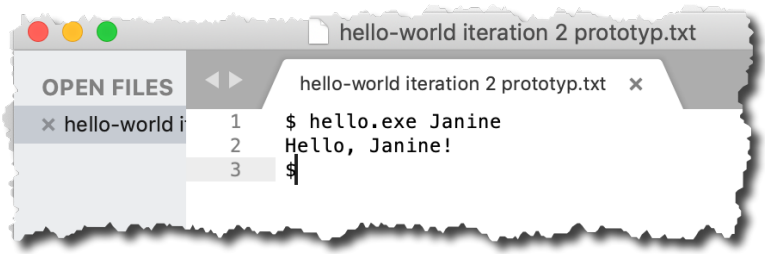
Prototypencode kann jede erdenkliche Form haben, ist aber strickt getrennt vom Produktionscode. Selbstverständlich kann prototypischer Code

jedoch Elemente des Produktionscodes benutzen. Es zahlt sich also aus, den Produktionscode so zu strukturieren, dass er in Bausteinen vorliegt, die in Prototypen "wiederverwendbar" sind. Die Notwendigkeit von Prototypen angesichts immer wieder herrschender Unklarheit, sollte mithin eine Motivationsquelle für die **Modularisierung** von Produktionscode sein.

Modularisierung sei an dieser Stelle pauschal und vereinfachend verstanden als eine Zusammenfassung von Logik in Funktionen in Klassen. Eine genauere Definition des Begriffs Modul gehört zu den Erläuterungen der Entwurfsphase.

Die Natur von Prototypen im Allgemeinen und prototypischem Code im Besonderen ist die **Vorläufigkeit**! Prototypen haben keine lange Lebensdauer, so dass bei ihrer Herstellung Korrektheit und Ordnung keine spezielle Rolle spielen. **Beim Prototypen geht es allein darum *herauszufinden*, was genau Wert für den Auftraggeber bedeutet.**

Hier ein Beispiel für einen Prototypen für Iteration 2 des Begrüßungsprogramms aus Lektion 1:



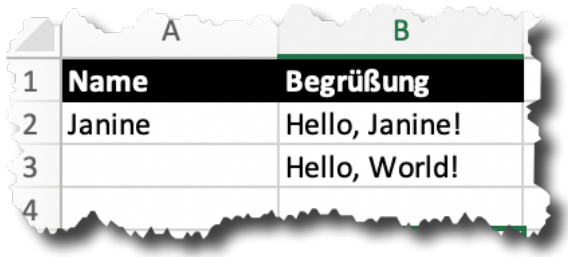

Der Prototyp ist nur eine Skizze der Bedienung des Programms. Er besteht aus etwas Text, nicht einmal Code ist nötig. Aber das reicht aus, um dem Auftraggeber Feedback zu entlocken: *"Haben Sie sich so die Bedienung des Programms vorgestellt? Meinten Sie, dass der Name auf der Kommandozeile mitgeteilt werden soll?"*

Und es zeigt eine Kerneigenschaft von Prototypen: sie sind leichtgewichtig. Wo Unklarheit herrscht, soll ohne Ballast Klarheit geschaffen werden. In Unklarheit sind Ordnung und Korrektheit noch nicht wichtig.

Unklarheit in Bezug auf die Bedienung einer Software ist ein häufiger Grund für schlechte Anforderungen. Die wachsende Zahl an UI-

Prototyping Werkzeugen bezeugt das. Doch diese Unklarheit in Bezug auf die Form sollte nicht verwechselt werden mit einer Unklarheit in Bezug auf den Inhalt, d.h. die Funktionalität. Nur, weil der Auftraggeber nicht recht weiß, wie die Bedienung genau aussehen/sich anfühlen soll, können keine Beispiele für die grundlegend geforderten Transformationen formuliert werden? Hier ist die Softwareentwicklung gefordert, differenziert hinzuschauen und auch den Auftraggeber herauszufordern.

In Bezug auf Iteration 2 könnte das bedeuten, dass du mit dem Auftraggeber auch bei Unklarheit in Bezug auf die Benutzerschnittstelle schon Klarheit in Bezug auf das grundlegende Verhalten erzielen kannst. Dafür ist nicht die Darstellung wichtig, sondern die Datenverarbeitung, die Übersetzung von Input in Output. Das könnte z.B. so aussehen:

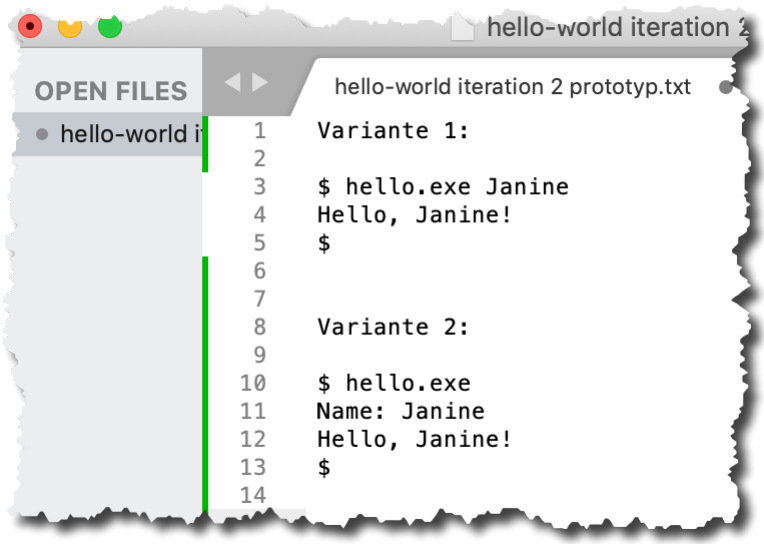


| | A | B |
|---|---|---|
| 1 | **Name** | **Begrüßung** |
| 2 | Janine | Hello, Janine! |
| 3 | | Hello, World! |
| 4 | | |

Denn daraus kannst du zumindest schon eine API-Funktion ableiten, z.B. `string Greet(string name)`. Mit einem solchen Vorgehen wäre die Unklarheit zurückgedrängt. Nicht alles ist unklar, sondern "nur" die Bedienung der Software. Du könntest also mit Produktionscode für die Übersetzung schon beginnen, während du bei der Benutzerschnittstelle noch im Prototypenmodus arbeitest.

In jedem Fall **sind Prototypen kurzlebige Werkzeuge zur Generierung von Feedback in Bezug auf sehr spezifische Aspekte von Software**, egal, ob es sich um die Benutzerschnittstelle oder Funktionalität handelt. Es sind Sonden, die "an den Auftraggeber angelegt werden", um seine Reaktion zu messen. Es ist hinlänglich bekannt, dass Auftraggeber nur schwer sagen können, was sie wollen - doch sie können sehr gut erkennen, was sie nicht wollen, wenn man ihnen Alternativen vorlegt. Du kennst sicher die Reaktion von Auftraggebern "Nein, doch nicht so! Das geht gar nicht!", wenn du ihnen etwas präsentierst, was du nach ihren unklaren Vorstellungen mit viel Mühe erarbeitet hast. Auftraggeber sind sehr, sehr

gut darin, Differenzen zu ihren Vorstellungen zu erkennen - auch wenn sie ihre Vorstellungen nicht artikulieren können.




*Ein Text-Prototyp für die Benutzerschnittstelle mit zwei Alternativen*

In Situationen der Unklarheit hilft es daher, zügig dem existierenden Zustand *S* einen Zustand *T* nebenzuordnen und zu fragen, *"Ist das besser? War es so gemeint?"* Damit wird einerseits klar gemacht, was die Softwareentwicklung bisher meint verstanden zu haben (dass *T* vom Auftraggeber gewünscht sein könnte); andererseits kann der Auftraggeber etwas sehen/fühlen, statt sich nur etwas vorzustellen, was es ihm leicht macht zu erkennen, ob es noch einen Kontrast zu seiner bisher schwammigen Qualitätserwartung gibt.[24]

Prototypen bauen einen Sandkasten auf, in dem du Softwareentwicklung spielerisch "austoben" kannst. Du kannst Feedback vom Auftraggeber generieren. Du kannst aber auch nur für dich persönlich Spikes[25] herstellen,

[24] "Aus Fehlern lernen" ist ein beliebter Spruch. Ich finde den Begriff "Fehler" aber sehr belastet und glaube nicht, dass wir zum Lernen Fehler willkommenheißen sollten. Sie sind vielleicht nicht komplett vermeidbar und wenn sie entstehen, kann man auch daraus lernen. Viel wichtiger finde ich, das was den Fehler ausmacht: den Kontrast. Wie nützlich bewusst hergestellte Kontraste für das Lernen sind, habe ich in einem Blogartikel beleuchtet.

[25] https://en.wikipedia.org/wiki/Spike_(software_development)

um zu eruieren, ob/wie Technologien genutzt oder Lösungsansätze codiert werden könnten.

Unklarheit kann also außen wie innen herrschen. In beiden Fällen solltest du dringend mit Prototypen forschen, bevor du Produktionscode anfasst.

Dass dann prototypischer Code wesentliche Qualitäten von Produktionscode vermissen lässt, liegt in seinem Zweck. Es verbietet sich daher, prototypischen Code "einfach so" in Produktionscode umzumünzen oder ohne weitere Nacharbeit dorthin zu kopieren.

**Code aus einem Prototypen muss auf dem Weg in den Produktionscode in puncto Korrektheit und Ordnung auf das notwendige Niveau für langfristig hohe Produktivität gehoben werden**. Angemessene Tests sind einzurichten, angemessene Strukturen sind zu entwerfen und zu implementieren.

Ziel des Prototypen ist es, nachzubessern, wo die Analyse bis dahin zu kurz gesprungen ist.

Nach einem Prototypen soll klar sein, wie Testfälle für eine Anforderung auszusehen haben und welche Funktionen ihnen gegenüberstehen sollen.

**Prototyping ist insbesondere eine Unterstützung der Analyse durch Codierung.** Die Softwareentwicklung ist nicht untätigt - doch sie muss klar machen, dass ihre Arbeit keinen Produktionscode erzeugt, sondern "nur" ein vorläufiges Analysehilfsmittel.

Mit diesem Nein gegenüber dem Auftraggeber schützt du deine Kapazität für die Herstellung von Features im Produktionscode. Würdest du das nicht tun und vorzeitig mit der Veränderung von Produktionscode beginnen, wären später verschwenderische Nacharbeiten in Form von Bug Fixing und Refaktorisierungen die Folge.

**Die Softwareentwicklung hat die Verantwortung, die von ihr betreute Ressource Produktionscode zu schützen. Dazu gehört ein klares Nein in Unklarheit, was Verständnis von Anforderungen oder auch Technologien angeht.**

# Die Schwierigkeit der Umsetzung einstufen

Die an der Codierung anliegenden Anforderungen bzw. ihre Lösungsansätze sind unterschiedlich schwierig umzusetzen mit Code im Allgemeinen und Logik im Speziellen. Mehr oder weniger schwierig bedeutet, dass die Herstellung des Ergebnisses mehr oder weniger lange braucht und/oder mehr oder weniger punktgenau die gewünschte Qualität haben wird.

Anforderungen, die nicht schwierig sind, können schnell und verlässlich mit allen Qualitäten codiert werden. Anforderungen, die hingegen sehr schwierig sind, brauchen mehr, gar viel mehr Zeit oder sogar unbekannt lange, bis sie womöglich nur mit unvollständiger Qualität geliefert werden.

Wie schwierig ist aber eine bestimmte Anforderung (z.B. formuliert als User Story) umzusetzen? Die Agilität lässt Entwickelnde das oft mittels einer Story Point Schätzung grob beurteilen. Damit bekommt der Scrum Product Owner als Auftraggeber einen Eindruck davon, wie viel Mühe eine Umsetzung machen könnte. Diese Angabe kann er dann in Bezug setzen zum Wert, den er einer User Story beimisst, um ihr eine Priorität in der Entwicklungsreihenfolge zu geben oder über eine weitere Verfeinerung (Granularität) zu entscheiden.

In der Agilität dient die Einstufung des Umsetzungsschwierigkeitsgrades von Anforderungen der Steuerung der Entwicklung. In der Codierung hingegen, wenn schon entschieden ist, dass eine Anforderung umzusetzen ist, dient sie der Wahl der Methode, wie die Umsetzung betrieben werden soll.

Das Kriterium *"Verständnis liegt dokumentiert durch Testfälle und Funktionen vor"* dient einer ersten groben Einschätzung des Schwierigkeitsgrades. Wird das Kriterium nicht erfüllt, ist der Schwierigkeitsgrad im Grunde unbekannt hoch.

Problemschwierigkeit ist für mich ein Kontinuum. Das reicht von trivial bis unbekannt. Dieses Kontinuum wird durch das Kriterium mithin in mindestens zwei Bereiche geteilt: Anforderungen, für die keine codierbaren Akzeptanzkriterien vorliegen, liegen darin pauschal im **Chaos**.


Die Bezeichnung "chaotisch" ist dem Cynefin Framework[26] entlehnt. Sie soll darauf hinweisen, dass im Angesicht von Chaos die angemessene Reaktion das unverzügliche Tun ist. "Taten statt Warten" ist angezeigt.

Mit unverzüglichem Tun meine ich natürlich nicht Aktionismus. Wildes Flügelschlagen ist kontraproduktiv. Vielmehr soll das, was getan wird, gegründet sein in einem Wertesystem. Das bedeutet, es gibt ein Fundament an Regeln und Prinzipien, die der Ordnung der Welt dienen. Nur wenn eine solche Ordnung herrscht, ist sichergestellt, dass Ziele erreicht werden können.

Für das gesellschaftliche Leben gehört zum Wertesystem z.B. Ehrlichkeit oder Rechtsstaatlichkeit. Wo Menschen im Miteinander nicht mehr erwarten können, dass ihr Gegenüber ebenfalls diese Werte schätzt und beherzigt, herrscht Chaos. Und umgekehrt, wo gesellschaftliches Chaos herrscht, besteht die unverzügliche Tat darin, ehrlich und rechtsstaatlich zu handeln.

[26] https://en.wikipedia.org/wiki/Cynefin_framework

In der Codierung, wie hier vorgestellt, gilt als Wert *"Produktionscode steht hinter einer Funktion unter automatisiertem Test"*. Ist das nicht der Fall, herrscht eben Chaos. In der Literatur wird diese Form von Code auch Legacy Code[27] genannt:

> *"Michael Feathers introduced a definition of legacy code as code without tests, which reflects the perspective of legacy code being difficult to work with in part due to a lack of automated regression tests."*

**Produktionscode ohne automatisierte Tests "als Gegengewicht" bzw. "Sicherheitsnetz" ist chaotisch nicht aufgrund seiner Struktur, sondern aufgrund der fundamentalen Unsicherheit, was passiert, wenn man ihn verändert.** Ob man den Effekt erzielt, den man erzielen will, kann nicht zügig, nachvollziehbar und personenunabhängig festgestellt werden. Es besteht keine klare Grenze zwischen korrekt und inkorrekt.

Automatisierte Tests ziehen eine klare Grenze zwischen korrekt und inkorrekt.

Wenn automatisierte Tests ausgeführt werden, ist unzweideutig klar, ob Produktionscode diesseits oder jenseits dieser Grenze steht.

## Zusammenfassung

Automatisierte Tests lassen in der Softwareentwicklung mit ihrer Grenze eine fundamentale **Dualität** entstehen: Korrektheit vs. Inkorrektheit. Sie ist genauso fundamental wie die von Tag und Nacht, Himmel und Erde, Gut und Böse.

Die Einführung von Dualitäten lassen in Schöpfungsgeschichten Ordnung aus Chaos entstehen. Dasselbe leisten automatisierte Tests. Ihr Vorhandensein grenzt chaotische von nicht-chaotischen Umsetzungssituationen

[27] https://en.wikipedia.org/wiki/Legacy_code

ab. Das ist eine erste wesentliche Unterscheidung und damit eine Entscheidung für eine Methode: **im Chaos ist Prototyping die Methode der Wahl**.

Wenn du Korrektheit und Ordnung herstellen willst, musst du zuerst erkennen, ob dafür überhaupt schon die Grundbedingungen gegeben sind. Im Chaos ist das nicht der Fall. Deshalb diese deutliche Grenzziehung, dieses kategorische Nein zur Arbeit am Produktionscode, solange Testfälle mit zugehörigen Funktionen fehlen.

Das ist der erste methodische Pfeil in deinem Köcher für die Codierung: Fange gar nicht erst mit ihr an, wenn die Anforderungen nicht ein Mindestmaß an Qualität haben. Dieses Mindestmaß besteht im Vorhandensein von automatisierbaren Akzeptanzkriterien. Punkt. Das musst du deinem Auftraggeber immer wieder klar machen. Weniger geht gar nicht - zumindest was Produktionscode betrifft. Du bist ja willig, mit ihm Unklarheit auszuräumen; um dabei Produktionscode zu verändern, hast du aber einen gewissen professionellen Anspruch an dein Verständnis und an die Überprüfbarkeit der Korrektheit deiner Eingriffe.

Ein Akzeptanztest steht mithin für Arbeit am Produktionscode immer am Anfang. So viel *test-first* muss sein: Erst der Akzeptanztestcode, dann der Produktionscode. Und wenn das nicht geht, dann eben ein Prototyp der einen oder anderen Art.

# 03 - Sofort codieren in der Trivialität

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

## Trivialität als Gegenteil von Chaos

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

## Vorsicht vor Selbstüberschätzung!

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

# 04 - Schrittweise codieren in der Einfachheit

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

## Trittsteine legen

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Pear Programming

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

## Die Kunst der Problemskalierung

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Sichtbarkeit von Variationsdimensionen

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Variationen ordnen

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Mengendimensionen



### Kategoriendimensionen



## Am Beispiel



### Akzeptanztest



### Testinkrement 1



### Testinkrement 2



### Testinkrement 3

### Testinkrement 3.1

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Testinkrement 4

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Testinkrement 5

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Reflexion

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

## Zusammenfassung

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

## 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 1

*Geschätzte Bearbeitungsdauer: 15min*

Klassifiziere folgende Problemstellungen als trivial, einfach oder chaotisch. Denke dich einen Moment ein in die Problemstellungen mit möglichen Lösungsansätzen, aber löse die Probleme nicht vollständig. Es geht ja darum, ohne Lösung ein Gefühl für unterschiedliche Schwierigkeitsgrade zu bekommen:

- Berechne das Ergebnis eines einfachen mathematischen Ausdrucks, der positive ganze Zahlen und die Operatoren `+, -, *, /` enthalten kann, z.B. `"2 + 30 * 400"`. Es gelten *keine* Operatorpräzedenzen, so dass das Ergebnis für das Beispiel `12800` ist.
- Bestimme die nächste Generation für die "Lebewesen" in einem Game-of-Life[a], bei dem "die Welt" durch `bool[,]` repräsentiert ist.
- Finde eine Lösung für das NQueen-Problem[b]: Für ein gegebenes N (1..20) platziere N Damen auf einem NxN Schachbrett so, dass sie sich nicht gegenseitig schlagen können.
- Wandle eine Binärzahl in ihr dezimales Äquivalent um, z.B. würde `13` der Binärzahl `1101` entsprechen.

[4] https://en.wikipedia.org/wiki/Conway's_Game_of_Life
[5] https://en.wikipedia.org/wiki/Eight_queens_puzzle

# Aufgabe 2

## Anforderungen

Wandle eine römische Zahl in ihre dezimales Äquivalent um, z.B. würde aus `MCMLXXXIV` die Dezimalzahl `1984`.

Die Funktion wird nur mit gültigen römischen Zahlen aufgerufen.

## TODO: Dokumentiere dein Verständnis

*Geschätzte Bearbeitungsdauer: 10min*

1. Der erste Akzeptanztest steckt in der Anforderungsbeschreibung. Finde für einen zweiten Akzeptanztest die römische Zahl zur Dezimalzahl `1492`.
2. Überlege dir eine Signatur für die Funktion, die das Problem löst.

## TODO: Inkrementelle Codierung

Ich nehme mal an, dass du das Problem nicht für trivial hältst und die Logik "einfach so" hinschreibst. Andererseits sind die Anforderungen klar, so dass das Problem nicht chaotisch ist.

## 1. Testfälle finden

*Geschätzte Bearbeitungsdauer: 15min*

Fasse das Problem daher als einfach auf. Finde eine Reihe von inkrementellen Tests, die schwieriger und schwieriger werden. Überlege, ob und welche Variationsdimensionen es gibt.

## 2. Test-first codieren

*Geschätzte Bearbeitungsdauer: 30min*

Implementiere dann wie folgt:

1. Codiere die Akzeptanztests und lege dabei die Produktionfunktion mit ihrer Signatur leer an.
2. Codiere den ersten inkrementellen Test.
3. Codiere die Logik in der Produktionsfunktion, um den inkrementellen Test zu befriedigen.
4. Codiere den nächsten inkrementellen Test und danach die nötige Produktionslogik.
5. Fahre in diesem Rhythmus fort, bis die Funktion reif im Sinne der Akzeptanztests ist.

Gehe also wieder *test-first* vor. Schreibe aber nicht zuerst alle Tests hin, sondern *wechsle Tests und Produktionscode ab.* Nur die Akzeptanztests legst du dir einmal zu Anfang als Latte auf, über die du ultimativ springen willst.

Gehe also im Sinne der üblichen Testwerkzeuge im *red-green* Wechsel vor. cTDD fordert dich nach *green* zwar noch auf, über eine Refaktorisierung nachzudenken, doch das ist jetzt noch nicht das Thema.

Versuche beim Codieren nach jedem Schritt an ein *Commit* auf deinem Code-Repository zu denken. Wenn du willst, lege für die Übung einen persönlichen Branch an, so dass im Review die Commits sichtbar bleiben, auch wenn andere auf dem Repository gearbeitet haben sollten.

# 05 - Komplementär codieren in der Kompliziertheit

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

## Zerlegung in komplementäre Teilprobleme

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Funktionen repräsentieren Lösungen

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

## Integration Operation Segregation Principle

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

## Zerlegungsbeispiel I

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

## Leitfragen für die Zerlegung



## Analyse & Entwurf



## Codierung



### Teilproblem “Verschweißen”



### Teilproblem “Übersetzen”



### Teilproblem “Zerlegung”



#### Iteration 1

### Iteration 2

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Iteration 3

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Refaktorisierung

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Iteration 4

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

## Integration

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Refaktorisierung

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

# Reflexion

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

# Buddelschiff Programmierung

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

# Zerlegungsbeispiel II

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Stepwise refinement 2.0

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

## Bottom-up Codierung

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Domäne I - Zählung der bisherigen Besuche

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Domäne II - Vorlage wählen

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Domäne III - Vorlage ausfüllen

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Persistenz

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Integration

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Refaktorisierung

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

## Reflexion

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

# Zusammenfassung

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

# Übungsaufgaben

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

# 06 - Tests als Treiber der Modularisierung

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

## Akzeptanztests

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Triviale Probleme

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Einfache Probleme

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Komplizierte Probleme

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

## Gerüsttests

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Gerüsttestfälle erhalten I - Akzeptanztests



### Gerüsttestfälle erhalten II - Modultests



## Zusammenfassung



## Übungsaufgaben

# 07 - Testbarkeit steigern mit Surrogaten

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

## Logik dynamisch binden

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Statische Bindung

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Dynamische Bindung mit Funktionszeigern

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Dynamische Bindung mit Objekten

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

#### Duck Typing

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Vererbung



### Interfaces



## Zusammenfassung



# Surrogate in Tests einsetzen



## Extraktion einer Klasse und Abstraktion mit Interface



## Injektion einer Objektfabrik



## Surrogate unterschieben

## Reflexion

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Vorsicht vor Whiteboxing

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Vorsicht vor komplexen Surrogaten

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

# IOSP over Surrogates

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

## Extraktion eines Belangs

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

## Refactoring to Functional Core

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

## Schritte in die Objektorientierung

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

## Zusammenfassung

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

## Übungsaufgaben

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

# 08 - Experimentell codieren in der Komplexität

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

## Experimentelles Vorgehen im Testcode

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### TDD as if you meant it (TDDaiymi)

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

## Beispiel #1: FromRoman revisted

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Inkrement 1

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Inkrement 2

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Inkrement 3



### Inkrement 4



### Inkrement 5



### Inkrement 6



### Reflexion



## Beispiel #2: Eine ruhige Bowlingkugel schieben



### Analyse

### Inkrementelle Testfälle

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

## Codierung

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Inkrement 1

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Inkrement 2

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Inkrement 3

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Inkrement 4

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Inkrement 5

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Refaktorisierung I



### Inkrement 6



### Akzeptanztest



### Refaktorisierung II



### Überraschungstest



## Reflexion



# Zusammenfassung

# Übungsaufgaben

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

# 09 - Test-first refaktorisieren

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

## Frustrierende Lektüre

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Fehlende Bedeutung

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

#### Zutaten für klare Bedeutungen

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Fehlende Zusammenhänge

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

#### Zutaten für deutliche Zusammenhänge

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

## Fehlende Testbarkeit

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Zutaten für hohe Testbarkeit

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

# Warum refaktorisieren?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

## Softwarewartung erhält Ordnung proaktiv

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

# Schrittweise aufräumen I

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

## Bestimme das System-to-Refactor (S2R)

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Zweck klären

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Grenzen ziehen

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Entry Points finden

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Abhängigkeiten aufdecken

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Zusammenhänge herstellen

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

## Refactor to Test-First

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Refactor to Testability

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Reflexion

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

# Schrittweise aufräumen II

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

## Refactor to IOSP

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Was sind Verantwortlichkeiten?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Pass 1: Verantwortlichkeiten identifizieren

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Pass 2: Extrahieren

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

## Refactor to Modules

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Mit Datenklassen gegen die Primitive Obsession

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Mit Serviceklassen für mehr Testbarkeit



## Dokumentieren



## Reflexion



# Zusammenfassung



# Übungsaufgaben

# 10 - Finale mit Testmatrix

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

# 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[28] an, was ich dir ergänzend an Trainings und Coaching bieten kann.

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

[28] https://ralfw.de/

# Musterlösung: 01 - Die Anforderung-Logik Lücke

## Aufgabe 1 - Gründe für automatisiertes Testen

Bevor du dich einlässt auf *test-first* Codierung, willst du sicher sein, dass der Aufwand, den das bestimmt bedeutet, auch gerechtfertigt ist. Deshalb habe ich dich ermuntert, selbst mal zu überlegen, welche guten Gründe es geben könnte, sich darauf einzulassen.

Wie du an den folgenden Listen siehst, finde ich, dass es eine ganze Menge solcher guten Gründe dafür gibt. Ja, ich weiß, dass *test-first* Vorgehen manchmal nervt. Und manchmal "klappt es nicht" - doch daraus würde ich nicht ableiten, dass es nicht das Ideal ist, dem du zustreben solltest. *Test-first* ist für mich zum Standard avanciert, zur Norm. So sieht fachgerechte Programmierung von Produktionscode einfach heute aus.

### Beispielhafte Gründe für die Testautomatisierung

Zuerst aber "nur" die Testautotmatisierung. Das ist weniger als *test-first* und schließt also auch *test-after* ein (wenn es denn "dazu kommt"). Warum sollte Software überhaupt automatisiert getestet werden, statt manuell, indem du die Software aufrufst und "durchspielst"?

- Höhere **Stabilität** durch Detektion von Regressionen: Weil automatisierte Tests jederzeit "auf Knopfdruck" ausgeführt werden können, besteht kein Grund auf einen Test, der schon einmal ausgeführt wurde, zu verzichten. Die Reifetests von heute werden so zu Stabilitätstests von morgen.

- Dokumentation der **Anforderungen**: Tests sind ausführbare Spezifikationen. Die Summe aller Tests beschreibt die Erwartungen an Software unzweideutig.
- Dokumentation der **Nutzung** von Code: Testcode ist der erste Nutzer von Produktionscode. Wie jener diesen bedient ist beispielhaft. Mit Tests können seltene Sonderfälle und allgemeine wie umfassende Use Cases beschrieben werden.
- Geringere **Kosten** - über die Zeit: Auch wenn die Einstandskosten für automatisierte Tests hoch sein mögen - einen Test jetzt zu schreiben, statt das Programm nur mal eben auszuführen für eine Reifeprüfung, macht viel mehr Arbeit -, liegt der *break-even point* in nicht allzu weiter Zukunft. Das gilt für Reifeprüfungen wie die noch wichtigeren Stabilitätsprüfungen.
- Höhere "**Benutzerfreundlichkeit**": Automatisierte Tests können von jedem Teammitglied oder gar automatisiert in einer CB/CI-Pipeline ausgeführt werden.
- **Nachvollziehbarkeit**: Nur mit automatisierten Tests ist jederzeit nachvollziehbar, was überhaupt getestet wurde und wird.
- Beobachbare **Testabdeckung**: Nur mit automatisierten Tests lässt sich die Testabdeckung beobachten und damit beurteilen, ob das Testen überhaupt schon/noch ausreichend ist.
- Größere **Testumgebungsvielfalt**: Automatisierte Tests lassen sich nicht nur an den Rechnern ausführen, an denen gerade Menschen sitzen. Sie können vielmehr ausgerollt werden auf virtuelle Maschinen, die in jeder erdenklichen (und relevanten) Weise konfiguriert sind.
- Größere **Ordnung**: Um automatisiert testen zu können, muss Code in geeigneter Weise strukturiert sein; *Testbarkeit* ist eine Voraussetzung für automatisierte Tests. Testbar ist Code, wenn eine gewisse Entkopplung von Codeteilen existiert, was die Ordnung erhöht und den Code wandelbarer macht.

## Beispielhafte Gründe für *test-first* Testautomatisierung

Nach der obigen Liste sollte es keine zwei Meinungen mehr geben, hoffe ich, dass automatisierte Tests "alternativlos" sind. Jetzt musst du sie nur noch schreiben. Aber wann? Ich meine: konsequent *test-first*. Denn das hat weitere Vorteile:

- Höhere **Verlässlichkeit**: Automatisierte Tests werden überhaupt geschrieben. Denn wenn automatisierte Tests nicht zuerst geschrieben werden, dann ist später die Motivation gering und die Zeit für solche "zusätzliche Arbeit" meist knapp.
- Höhere **Testbarkeit**: Wer den Produktionscode vom Test her denkt, weil er den zuerst schreibt, achtet früher (oder überhaupt) darauf, dass der Produktionscode auch einfach zu testen ist, also eine hohe Ordnung hat.
- Bessere **Schnittstellen**: Schnittstellen bieten Dienstleistungen an und sollten daher vom Nutzer aus gedacht werden. Wenn Entwickelnde zuerst in Tests die Schnittstellen ihres Produktionscodes nutzen müssen, bevor sie sie "in Code gießen", sind sie sensibler dafür, was andere Nutzer später für einfach/verständlich halten könnten. Test-first Codierung ist eine Form von "eat your own dog food".

*Test-first* geht über "nur" automatisierte Tests hinaus. Du schreibst die automatisierten Tests verlässlich vor dem Produktionscode. Aber wie? *Test-first* ist noch nicht *Test-Driven Development (TDD)* wie du es vielleicht kennst.

Was Gründe für TDD sein könnten, frage ich dich allerdings hier nicht. Diese Praktik ist Thema eines eigenen Kapitels. *Test-first* ist für mich wichtiger und grundlegender. Der Titel der beiden Bände ist für mich Programm.

# Aufgabe 2 - Eine Anwendung test-first entwickeln

## Analyse

### Akzeptanztestfälle

Das Programm wird mehrfach gestartet und muss daher seine "Erinnerung" (Zustand) an bisherige Gäste persistieren.

Die Funktionsweise kann "von außen" überprüft werden, indem nacheinander Gäste im Rahmen desselben Szenarios begrüßt werden.

1. *System Under Test (SUT) starten.*
2. Roger -> Hello, Roger!
3. Janine -> Hello, Janine!
4. *SUT stoppen und wieder starten.*
5. Roger -> Welcome back, Roger!
6. Roger -> Hello my good friend, Roger!
7. *SUT stoppen und wieder starten.*
8. Janine -> Welcome back, Janine!
9. 20x Roger
10. Roger -> Hello my good friend, Roger!
11. Roger -> Hello my good friend, Roger! Congrats! You are now a platinum guest!
12. Roger -> Hello my good friend, Roger!

### API-Funktion

Um das SUT inkl. Zustand einfach starten/stoppen zu können, wird die Begrüßungfunktion einer Instanzklasse zugeordnet:

```
class HelloBackend {
  public string Greet(string name) {...}
}
```

Diese Funktion überspannt die gesamte Logik des Programms, außer der für die Benutzerschnittstelle. Auf diese Weise kann maximal viel Logik im Akzeptanztest überprüft werden.

Eine Konfiguration des Backend zumindest mit einem Pfad (connection string) für die Persistenz, scheint allerdings hilfreich für das Rücksetzen-/Starten im Test.

## Entwurf der Persistenz

Das Programm muss sich über Starts hinweg die begrüßten Gäste merken. Laut der Anforderungen ist mit ungefähr 3*100*25*2= 15.000 Besuchen zu rechnen. Die Zahl der verschiedenen Gäste ist sicher geringer.

Das ist keine sehr große Zahl, so dass die persistente Zustandshaltung sehr simpel sein kann: es genügt eine Textdatei, deren Inhalt komplett in-memory gehalten werden könnte.

Weder ein Speicherplatz- noch ein Performanceproblem würde ich erwarten.

Jeder Gast könnte mit einem Tupel *(Name, Anzahl der Besuch)* in der Gästeliste vertreten sein (Option *Map*), z.B.

```
Roger, 3
Janine, 2
Mark, 7
Henry, 1
```

Oder die Gästeliste besteht schlicht aus einer fortgeschriebenen Liste von Namen (Option *Event Stream*):

```
Roger
Janine
Mark
Roger
Henry
Mark
Janine
```

Letztere Lösung scheint mir flexibler, auch wenn sie mehr Speicherplatz benötigt. Bei der geringen Besuchszahl ist Speicherplatz jedoch kein Problem.

# Codierung

## 1. Akzeptanztests

```
[Fact]
public void Acceptance_test_scenario()
{
  const string TEST_DB = "test.db";
  File.Delete(TEST_DB);

  var sut = new HelloBackend(TEST_DB);
  sut.Greet("Roger").Should().Be("Hello, Roger!");
  sut.Greet("Janine").Should().Be("Hello, Janine!");

  sut = new HelloBackend(TEST_DB); // auf diese Weise wird vermieden,
  // dass Zustand nur in-mem gehalten wird
  sut.Greet("Roger").Should().Be("Welcome back, Roger!");
  sut.Greet("Roger").Should().Be("Hello my good friend, Roger!");

  sut = new HelloBackend(TEST_DB);
  sut.Greet("Janine").Should().Be("Welcome back, Janine!");

  for (var i = 1; i <= 20; i++)
    sut.Greet("Roger");

  sut.Greet("Roger").Should().Be("Hello my good friend, Roger!");
  sut.Greet("Roger").Should().Be("Hello my good friend, Roger! Congrats! You are now a \
platinum guest!");
  sut.Greet("Roger").Should().Be("Hello my good friend, Roger!");
}
```

Die Akzeptanzkriterien habe ich alle in einem Szenariotest (Use Case) zusammengefasst. Dadurch muss ich den Zustand des *System under Test* nicht für einzelne Testfälle gesondert setzen und überprüfen. Vielmehr baut sich der Zustand natürlich durch fortschreitende Nutzung auf und wird durch korrekte Ergebnisse in der weiteren Verarbeitung implizit verifiziert. Lediglich am Anfang muss der persistente Zustand durch Löschen der "Datenbank" "auf Null gesetzt" werden, um für jeden Durchlauf gleiche Ausgangsbedingungen zu schaffen.

## 2. Produktionscode I: System under Test (SUT)

```
public class HelloBackend
{
  private readonly string _dbFilePath;

  public HelloBackend(string dbFilePath) {
    _dbFilePath = dbFilePath;
  }

  public string Greet(string name) {
    File.AppendAllLines(_dbFilePath, new[]{name});
    var names = File.ReadAllLines(_dbFilePath);

    var n = names.Count(x => x == name);

    var greeting = n switch {
        1 => $"Hello, {name}!",
        2 => $"Welcome back, {name}!",
        _ => $"Hello my good friend, {name}!"
    };
    if (n == 25) greeting += " Congrats! You are now a platinum guest!";

    return greeting;
  }
}
```

Die komplette zu testende Logik ist in einer Klasse gekapselt. Eine weitere Modularisierung scheint mir angesichts des geringen Schwierigkeitsgrades, des Abstraktionsgrades der C#-Mittel und der Wahl des Persistenzformates sowie des Tests im Rahmen eines Szenarios nicht nötig.

Das SUT enthält auch die Persistenzlogik, um sicherzustellen, dass die korrekt ist. Sie nicht zu testen, wäre leichtfertig. Sie jedoch getrennt zu testen, würde ein Zusammenspiel mit der Domänenlogik ungetestet lassen.

Die Konfiguration des SUT mit dem Datenbanknamen findet über den Konstruktor statt. Damit ist sie "aus dem Weg" bei der Nutzung des Backend-Objektes. Meine Annahme dabei ist, dass während der Existenz des Backends die Datenbank nicht gewechselt werden muss.

## 3. Produktionscode II: Einbettung des SUT in ein Programm mit Benutzerschnittstelle

```
class Program
{
  static void Main(string[] args) {
    var backend = new HelloBackend("guests.txt"); // Konfiguration

    while (true) {
      Console.Write("Name: ");
      var name = Console.ReadLine();
      if (string.IsNullOrWhiteSpace(name)) continue;

      var greeting = backend.Greet(name); // Nutzung

      Console.WriteLine(greeting);
    }
  }
}
```

Um das automatisiert getestete SUT siehst du jetzt natürlich weitere Logik. Die ist nicht (einfach) automatisiert testbar. In diesem Fall ist sie allerdings trivial; bei einem abschließenden manuellen Test würdest du sofort merken, wenn darin ein Bug sitzt, denke ich.

Aber nimm solche Logik nicht auf die leichte Schulter. In einem späteren Kapitel werden wir uns damit näher beschäftigen. Solche Mischungen von Logik und einem Funktionsaufruf wie `backend.Greet()` in `Main()` sind weder gut für die Verständlichkeit noch für die Testbarkeit.

# Musterlösung: 04 - Schrittweise codieren in der Einfachheit

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

## Aufgabe 1 - Einschätzung des Schwierigkeitsgrades

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Mathematischen Ausdruck berechnen

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

#### Beispielimplementation

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

#### Akzeptanztest

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Inkrementeller Test 1: Nur 1 Operand



### Inkrementeller Test 2: 2 Operanden mit Addition



### Inkrementeller Test 3: Beliebig viele Operanden durch Addition verknüpft



### Inkrementeller Test 4: Multiplikation als weiterer Operator



### Abschließender Test mit allen Operatoren



## Game-of-Life



## NQueen

## Binärzahlenkonvertierung



### Beispielimplementation



# Aufgabe 2 - Römische Zahlen in Dezimalzahlen wandeln



## Verständnis dokumentieren



## Inkrementelle Testfälle definieren



## Test-first codieren



### Akzeptanztests

### Inkrement 1

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Inkrement 2

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Inkrement 3

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Inkrement 4

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

## Reflexion

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

# Musterlösung: 05 - Komplementär codieren in der Kompliziertheit



## Aufgabe 1 - Römische Zahlen kompliziert wandeln



### Zerlegung für Lösungsansatz 1



### Zerlegung für Lösungsansatz 2



### Reflexion

# Aufgabe 2 - Game of Life



## Verständnis dokumentieren



## Schrittweise zerlegen



### Zerlegungsebene 1



### Zerlegungsebene 2



#### Matrix laden



#### Matrix speichern

### Kalkulation der nächsten Generationen



### Zerlegungsebene 3



### Reflexion



## Akzeptanztests



## Teilprobleme bottom-up lösen



### Matrix I - Grundfunktionen



### Matrix II - Nachbarschaft bestimmen

### Serialisierung



### Deserialisierung



### Dateiname für eine Generation



### Persistenz



### Domäne I - Zellenaufzählung



### Domäne II - Den Zellenzustand der nächsten Generation bestimmen



### Endspurt - Bottom-up Integration

### Korrektur eines unerwarteten Fehlers



### Das Finale - Aufruf der Lösung als Programm



## Reflexion

# Musterlösung: 06 - Tests als Treiber der Modularisierung



## Analyse



### Inkrementelle Teilprobleme



## Zerlegung der Inkremente inkl. Codierung



### Akzeptanztest

### Inkrement I

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Inkrement II

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Inkrement III

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Inkrement IV

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Inkrement V

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Reflexion

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

## Refaktorisierung mit nachträglicher Modularisierung

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

## Zusammenfassung

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

# Musterlösung: 07 - Testbarkeit steigern mit Surrogaten



## Aufgabe 1 - CSV-Daten tabellieren



### Analyse



### Planung



### Umsetzung

### Domänenlogik erweitern um die Trennzeichenkonfiguration



### Kommandozeilenparameter lesen



### Tabelle anzeigen



### Komposition I - Application



### Komposition II - Konstruktion



## Reflexion



### Säuberung der Kommandozeilenanalyse

# Aufgabe 2 - Game-of-Life



## Analyse



## Planung



## Umsetzung



### Application einführen



### Fortschrittsanzeige extrahieren



### Persistenz extrahieren

### Dynamische Bindungen einführen

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Akzeptanztest

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

## Reflexion

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

# Zusammenfassung

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

# Musterlösung: 08 - Experimentell codieren in der Komplexität



## Analyse



## Planung



### Zerlegung in Teilprobleme



### Inkrementelle Testfälle für die komplexen Probleme

### Lange Worte zerschneiden



### Worte zu neuen Zeilen zusammensetzen



# Codierung



## Akzeptanztests



## TDDaiymi - Lange Worte zerschneiden



### Inkrement 1



### Inkrement 2

### Inkrement 3

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

#### Refaktorisierung

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Inkrement 4

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Inkrement 5

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

#### Refaktorisierung I

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

#### Refaktorisierung II

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

## TDDaiymi - Zeilen zusammensetzen

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

## Inkrement 1



## Inkrement 2



## Inkrement 3



### Refaktorisierung I



### Refaktorisierung II



### Refaktorisierung III



## Inkrement 4

### Inkrement 5

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

#### Refaktorisierung I

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

## Triviale Probleme lösen

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Zeilen verschweißen

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Text in Worte zerlegen

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

## Integration

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

## Refaktorisierung in Module

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Hilfe für Zerlegung und Zusammenbau



### Worte zerschneiden



## Reflexion



## Zurückhaltung als Tugend

# Musterlösung: 09 - Test-first refaktorisieren

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

## Abgrenzung des System to Refactor (S2R)

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

## Characterization Test

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### S2R Entry Point testbar machen

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Console kapseln

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

#### Adapter-Klasse einführen

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Adapter injizieren



### Interface für Adapter-Klasse



## Characterization Test aufsetzen



### Goldmaster generieren



### Test mit Surrogat



# Verantwortlichkeiten identifizieren



# Entwurf der neuen Codestruktur

# Funktionen extrahieren

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

## Kommandozeilenanalyse

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

## HexDump I - Entzerrung der Logik

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Blöcke aus Datei lesen

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Zusammenstellung der HexDump-Blöcke

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

### Ausgabe formatieren

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

## HexDump II - Refactor to IOSP

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

# Refactor to Modules

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

# Bonus Verbesserungen

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

## Auflösung einer logischen Abhängigkeit

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

## Fehlerkorrektur in der Ausgabe

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.

# Zusammenfassung

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-first-codierung.