Effizientere Softwareentwicklung im Team durch Komponenten

Stefan Lieser

Effizientere Softwareentwicklung im Team durch Komponenten

Komponentenorientierung als Mittel der Arbeitsorganisation

Stefan Lieser

Dieses Buch können Sie hier kaufen
http://leanpub.com/komponentenorientierung

Diese Version wurde auf 2014-06-07 veröffentlicht

Kaum ein anderer Begriff der Softwareentwicklung wird so inhaltsleer verwendet, wie der Begriff Komponente. Entwickler diskutieren munter, wie sie ein Problem angehen wollen und verwenden dabei Komponente synonym mit Klasse, Assembly, GUI-Control und anderem. Auf Nachfrage ist den Entwicklern oft nicht bewusst, dass sie keine Definition für den Begriff nennen können. Oft weichen sie auf andere aus, wie etwa Modul, zu dem sie dann ebenfalls keine Definition nennen können. Einerseits ist das verwunderlich, andererseits findet sich auch in der Literatur meist nur eine schwammige oder viel zu komplizierte Definition. Doch lag in der Komponentenorientierung nicht einige Jahre die Hoffnung der Softwareentwicklungsbranche? Endlich wie die Automobilindustrie vorgefertigte Teile zusammenstecken und nicht immer wieder alles neu bauen? Wieso ist davon so wenig in der Softwareentwicklung angekommen? Dieses Buch befasst sich mit Komponenten. Insofern dürfen Sie davon ausgehen, dass ich die Komponentenorientierung weder für gescheitert noch für einen alten Hut halte. Und damit von Anfang an Klarheit herrscht, nenne ich Ihnen hier meine Definition von Komponente:

***Eine Komponente ist eine binäre Funktionseinheit mit separatem Kontrakt**.*

Punkt. Das ist alles. So einfach kann es sein.

Inhaltsverzeichnis

1 Einleitung

Komponenten sind ein Mittel der Arbeitsorganisation. Es geht in diesem Buch also nicht um Entwurf oder Architektur, sondern „nur" darum, wie die Arbeit in einem Team technisch gesehen so organisiert wird, dass das Team mit maximaler Geschwindigkeit und bester Qualität arbeiten kann. Erst Komponentenorientierung ermöglicht arbeitsteiliges Vorgehen. Damit ist sie Voraussetzung dafür, dass mehrere Entwickler am gleichen Softwaresystem arbeiten. Das mag sich banal anhören und Sie fragen sich möglicherweise, wieso Sie dazu Komponenten benötigen. Weiter oben habe ich von „maximaler Geschwindigkeit" und „bester Qualität" geschrieben. Natürlich besteht rein technisch keine zwingende Notwendigkeit, Software aus Komponenten zusammenzusetzen. Nicht selten habe ich Softwaresysteme gesehen, die aus einer Visual Studio Solution bestanden. Manchmal waren darin viele Projekte zu finden. Manchmal sehr viele… Eine wirklich komponentenorientierte Lösung habe ich selten gesehen.

Aber vergleichen Sie das mit der Automobilindustrie. Rein technisch gesehen kann ein Auto natürlich aus Einzelteilen auf kleinster Ebene zusammengesetzt werden. Aus Schrauben, Muttern, Wellen, Achsen, Zahnrädern, Blechen, Gussteilen, usw. Doch dann würde die Arbeitsteilung unendlich komplex. Erst klare Schnittstellen an den Komponentengrenzen machen es möglich, dass Motor, Fahrwerk, Sitze, Reifen und viele andere Bestandteile von unterschiedlichen Herstellern in unterschiedlichen Werken hergestellt werden. Mir ist bewusst, dass der Vergleich von Softwareindustrie und Automobilbau viele Fallstricke liefert. Doch in einem bin ich sicher: wir, in der Softwareindustrie, können den Automobilbauern noch einiges abschauen. Ohne ein Komponentenkonzept müsste die Automobilindustrie jeden Motor erst in ein Auto einbauen und

damit auf die Autobahn fahren, um ihn zu überprüfen. Macht sie aber nicht. Dafür gibt es Prüfstände. Der Motor „merkt" gar nicht, ob er in einem Auto oder in einem Prüfstand eingebaut wurde. So lange der Prüfstand die Schnittstelle des Motors korrekt bedient, ist alles in Ordnung. Diese Form der Zerlegung in Komponenten und die daraus resultierende Arbeitsteilung ist auch in der Softwareentwicklung möglich und notwendig. Und dafür steht dieses Buch. Sie werden nach dem Lesen feststellen, dass eigentlich alles ganz einfach ist. Und möglicherweise werden Sie sich sogar fragen, warum Sie nicht schon immer so gearbeitet haben.

1.1 Danksagungen

Dieses Buch wäre ohne meinen Kollegen Ralf Westphal nicht entstanden. Ich habe seit Jahren das große Vergnügen, mit ihm gemeinsam arbeiten zu können. In zahlreichen Seminaren haben wir das Konzept der Komponentenorientierung unterrichtet, ausprobiert und auch immer wieder verfeinert. Ohne Ralfs initiale Vorarbeit wäre das jedoch nicht möglich gewesen. Danke Ralf!

Ein weiterer Dank gebührt meiner Familie, die viel Geduld mit mir bewiesen hat. Sie haben mich immer wieder ermuntert, weiter zu schreiben und mir dann auch die Zeit eingeräumt.

1.2 Zum Aufbau des Buchs

Ein Team von Entwicklern sollte in optimaler Weise zusammenarbeiten. Die Art und Weise, wie das Team seine Arbeit so organisiert, dass es effizient arbeiten kann, wird als Arbeitsorganisation bezeichnet. Im nun folgenden Kapitel **„Herausforderung Arbeitsorganisation"** sind die Herausforderungen beschrieben, denen sich ein Team in Bezug auf die Arbeitsorganisation stellen muss. An diesen Herausforderungen muss sich die Komponentenorientierung messen lassen und Lösungen anbieten.

Das darauf folgende Kapitel **„Verwendete Notation“** beschreibt die im Buch verwendete Notation.

Um einen Eindruck davon zu vermitteln, wie ein Softwaresystem aussieht, das komponentenorientiert entwickelt ist, folgt ein konkretes Beispiel im Kapitel **„Komponentenorientierung am Beispiel“**. Das Beispiel ist bewusst klein gehalten, damit die Sicht frei bleibt auf das Wesentliche, nämlich die Art und Weise der Zerlegung des Gesamtsystems in Komponenten.

Im anschließenden Kapitel **„Verzeichnisstruktur und Projektorganisation“** wird die Verzeichnisstruktur detailliert beschrieben. Eine solche Struktur ist im Prinzip unabhängig davon, ob ein Softwaresystem komponentenorientiert aufgebaut wird oder nicht. Diese Struktur ist noch vor den Komponenten die Basis für eine reibungslose Zusammenarbeit mehrerer Entwickler an einem Softwaresystem. Ferner wird die Organisation der Visual Studio Solutions und Projekte beschrieben.

Es folgt das Kapitel **„Zerlegung in Komponenten“** über die Zerlegung eines Entwurfs in Komponenten. Hier wird der Begriff der Komponenten definiert und die konkrete Umsetzung in .NET beschrieben. Ferner wird dargelegt, nach welchen Kriterien die Zerlegung in Komponenten erfolgt.

Nach der Zerlegung müssen die Komponenten zum großen Ganzen zusammengefügt werden. Die Integration ist das Thema des anschließenden Kapitels **„Integration der Komponenten“**. Dabei geht es um das Zusammenfügen der einzelnen Komponenten zu einer lauffähigen Software sowie um die Automatisierung der Übersetzung der einzelnen Bestandteile (Buildprozess).

Die hier vorgestellten konkreten Verfahren zur Komponentenorientierung sind mit .NET realisiert. Komponentenorientierung ist natürlich auch mit anderen Plattformen möglich und sinnvoll. Der Inhalt des vorliegenden Buches ist mithin auf andere Plattformen übertragbar.

2 Herausforderung Arbeitsorganisation

Software zu entwickeln ist meist eine komplexe Angelegenheit. Software im Team zu entwickeln ist immer eine komplexe Angelegenheit. Das liegt daran, dass neben den Herausforderungen, die das eigentliche Entwickeln der Software ohnehin schon mit sich bringt, weitere Herausforderungen hinzu kommen. Um Software mit mehreren Menschen im Team zu entwickeln, müssen diese Menschen zunächst einmal ein Team bilden. Das ist nicht so selbstverständlich, wie es klingen mag. Zwar ist es heute üblich, Software mit mehreren Personen gemeinsam zu entwickeln, doch häufig entsteht dabei lediglich eine Gruppe und noch kein Team. Der wesentliche Unterschied zwischen Gruppe und Team liegt in der Selbstorganisation. Entwickelt sich die Selbstorganisation so weit, dass eine gemeinsame Verantwortung entsteht, spricht man von einem Team. In einer Gruppe dagegen ist jeder Einzelne für seine (Teil-)Ergebnisse verantwortlich. Nur im Team wird das Ergebnis als gemeinsames Ergebnis betrachtet, während in Gruppen noch die Einzelergebnisse im Vordergrund stehen.

Desweiteren ergeben sich durch Teamarbeit Herausforderungen im technischen Sinne. Schon dadurch, dass mehrere Personen an der Software arbeiten, entsteht schneller ein größeres System, als bei einer Einzelperson: es kommt in gleicher Zeit mehr Code zusammen. Vor allem muss nun aber die Zusammenarbeit so organisiert werden, dass nur geringe Reibungsverluste entstehen. Teams die einfach drauf los entwickeln, werden schnell die Grenzen einer adhoc Arbeitsorganisation feststellen.

2.1 Feature Developer vs. Feature Team

Eine der wichtigsten Fragen der Arbeitsorganisation lautet, wie die Zuständigkeit für ein Feature organisiert ist. Dabei ist ein Feature ein kleiner Ausschnitt aus den gesamten Anforderungen. Für die weitere Betrachtung ist eine trennscharfe Definition von Feature von geringerer Bedeutung. Es genügt, sich ein Feature als Teilfunktionalität oder Ausschnitt aus den Anforderungen vorzustellen. Nun gibt es zwei Möglichkeiten, die Umsetzung eines Features zu organisieren: es kann entweder jedes Feature einem einzelnen Entwickler zugeordnet werden, oder das gesamte Team setzt Feature für Feature gemeinsam um. Im ersten Fall spricht man von Feature Developer. Jeder Entwickler arbeitet an „seinem" Feature. Eine wirkliche Zusammenarbeit findet nicht statt. Zwangsläufig ergibt sich für eine Gruppe von Entwicklern daraus die Konsequenz, dass jeweils mehrere Features gleichzeitig in Bearbeitung sind. Übrigens ergibt sich aus der Organisation mehrerer Feature Developer auch die Notwendigkeit für Daily-Standups in Scrum. Die Entwickler müssen nicht miteinander reden, solange jeder an „seinem" Code arbeitet. Logischerweise muss man dann das Reden im Team organisieren, da es sich nicht von selbst ergibt. Für mich ist das höchstens die zweitbeste Lösung. Natürlich hilft organisiert vereinbartes Reden dem Team dabei, zusammen zu wachsen. Doch viel wichtiger ist es, die Arbeit so zu organisieren, dass wirklich zusammen gearbeitet wird. Dann entsteht die Notwendigkeit oder sogar der Wunsch, miteinander zu Reden, ganz natürlich von alleine.

Arbeitet das komplette Team gemeinsam an einem einzigen Feature, spricht man von einem Feature Team. Damit ein Team in der Lage ist, parallel am selben Feature zu arbeiten, muss die Arbeit natürlich so organisiert sein, dass dies überhaupt effizient möglich ist. Es bedarf vor allem einiger technischer Lösungen, weil das gesamte Feature natürlich in geeigneter Weise zerlegt werden muss, so dass Einzelteile entstehen, die jeweils von einer Einzelperson umgesetzt

werden können. An dieser Stelle kommen die Komponenten ins Spiel. Sie bilden in einem Team die größte Einheit, an der ein Entwickler alleine arbeiten kann. Doch bevor ich zur Lösung übergehe, möchte ich noch darauf eingehen, warum die Organisation der Arbeit in Form von Feature Teams unbedingt erstrebenswert ist.

Wir sind heute Rechner gewohnt, deren Prozessor mehrere Kerne enthält. Dadurch ist inzwischen echtes Multitasking möglich, also das wirklich parallele Abarbeiten von Programmen. Doch obwohl mehrere Kerne zur Verfügung stehen, sind auch heute in der Regel mehr Programme gleichzeitig aktiv, als Kerne zur Verfügung stehen. Der Trick dabei: der Prozessor wird reihum mehr oder weniger gleichmäßig auf die diversen anstehenden Aufgaben verteilt. Zu einem Zeitpunkt bearbeitet der Prozessor ein Programm. Durch den schnellen Wechsel entsteht für uns Anwender der Eindruck, die Programme würden parallel ablaufen. Dagegen ist nichts einzuwenden. Es entsteht aber die Frage, ob dies die beste Strategie ist, wenn eine der Aufgaben besonders schnell abgearbeitet werden soll. Übertragen wir die Analogie auf ein Entwicklerteam. Das Team hat die Möglichkeit, pseudo-gleichzeitig an mehreren Features gleichzeitig zu arbeiten. Dadurch entsteht für den Betrachter außerhalb des Teams der Eindruck, alle Features wären in Arbeit. Doch wird dabei jedes Feature in bestmöglicher Zeit fertig? Ganz klar: nein. Ein Feature, das so schnell wie möglich fertig werden soll, muss als einziges Feature vom gesamten Team bearbeitet werden. Es muss das einzige Feature sein, das überhaupt in Bearbeitung ist. Aus dieser Betrachtung ergibt sich eine Frage: Sollte ein Team jeweils ein einzelnes Feature in bestmöglicher Zeit fertigstellen, oder sollte es gleichzeitig an mehreren Features arbeiten, die dann jedes für sich nicht in optimaler Zeit fertig werden? Bei der Beantwortung der Frage hilft ein Blick zur *Theorie of Constraints*[1]. Aus ihr ergibt sich die Konsequenz, dass eine lokale Optimierung nicht zwingend zum globalen Optimum führt. Ob also ein Entwickler

[1] Zur Theorie of Constraints siehe z.B. http://de.wikipedia.org/wiki/Theory_of_Constraints

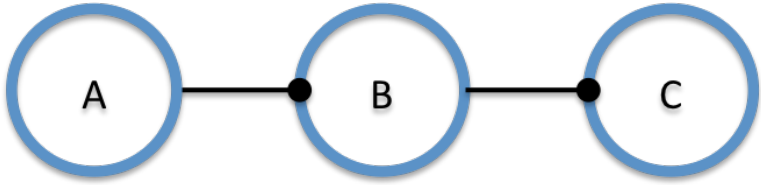
gerade Leerlaufzeit hat oder nicht, spielt eine untergeordnete Rolle. Dieses lokale „Problem“ zu beseitigen, in dem ein Team an mehreren Features arbeitet, erhöht nicht die Wahrscheinlichkeit, dass das Gesamtsystem optimal arbeitet. Im Gegenteil: meist führt die lokale Optimierung zu großen Problemen, die unerkannt bleiben. Schon daraus lässt sich ableiten, dass Teams gut daran tun, immer nur ein Feature in Bearbeitung zu haben und dies erst vollständig fertigstellen, bevor sie das nächste beginnen.

Eine weitere Betrachtung mag helfen, sich von der Organisation als Feature Developer zu verabschieden. Sobald mehrere Features gleichzeitig in Bearbeitung sind und ein systematischer Fehler entdeckt wird, ist die Wahrscheinlichkeit hoch, dass der Fehler mehrfach begangen wurde. Nehmen wir eine Architekturentscheidung als Beispiel. Wurde zu Beginn entschieden, für die Persistenz eine relationale Datenbank einzusetzen, dann kann sich während der Implementation herausstellen, dass eine der nicht-funktionalen Anforderungen auf diese Weise nicht oder nur schwer umsetzbar ist. Beispielsweise könnte der geforderte Durchsatz an Benutzertransaktionen pro Zeiteinheit nicht erreichbar sein. Sind nun bereits mehrere Features in Bearbeitung, wirkt sich diese Erkenntnis möglicherweise auf mehrere dieser Features aus. Vermutlich muss dann an mehreren Features eine Änderung vorgenommen werden. Wäre nur ein einzelnes Feature in Bearbeitung, hätte die Erkenntnis geringere Auswirkungen, denn dann würde sich die Nachbesserung in engeren Grenzen bewegen. Vor allem könnte die Erkenntnis dann bei den anderen Features sofort von Anfang an bedacht und genutzt werden.

Am Ende steht also als klare Erkenntnis: Software sollte im Team so entwickelt werden, dass immer ein Feature nach dem anderen bearbeitet wird. Es gibt keine angefangene Arbeit, die „auf Halde“ liegt, sondern ein begonnenes Feature wird erst vollständig bearbeitet, bevor mit dem nächsten begonnen wird.

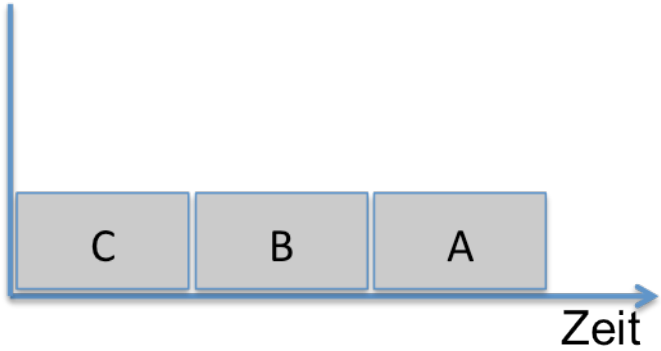
2.2 Typische Herausforderungen

Damit ein Team von Entwicklern gemeinsam an einem Feature arbeiten kann, muss das Feature in kleinere Teile zerlegt werden. Diese kleineren Teile bezeichne ich ganz abstrakt zunächst als *Funktionseinheiten*. Konkrete Ausprägung einer Funktionseinheit können sein, eine Methode, eine Klasse, eine Assembly, eine Komponente, ein Programm, etc. Die im folgenden geschilderten Herausforderungen sollen aufzeigen, welche Anforderungen an Funktionseinheiten gestellt werden, die von einem Team parallel entwickelt werden. Es ist klar, dass es dabei auf die *Komponente* als zentralen Begriff dieses Buches hinausläuft. Doch ohne die Lösung gleich parat zu haben, welche Herausforderungen bietet denn diese Form eines gemeinschaftlichen Entwicklungsprozesses? ### Vorgegebene zeitliche Entwicklungsreihenfolge durch Abhängigkeiten Sobald eine Funktionseinheit in kleinere Funktionseinheiten zerlegt wird, ergibt sich die Frage nach den Abhängigkeiten. Manchmal lassen sie sich vermeiden, aber am Ende bleiben immer Abhängigkeiten. Abhängigkeiten können die Reihenfolge der Implementation vorgeben. Dazu ein Beispiel: die Abbildung zeigt drei Funktionseinheiten A, B und C. Diese sind voneinander abhängig. A hängt ab von B, B hängt ab von C.

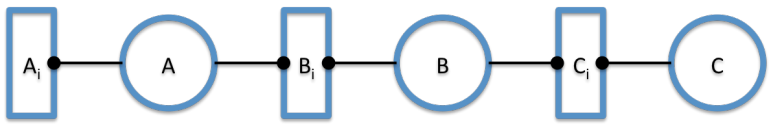


In manchen Fällen mag die erzwungene Reihenfolge der Implementation unkritisch sein. Handelt es sich beispielsweise um Methoden, stellt das aufgrund des Codeumfangs kein Problem dar. Sind die Funktionseinheiten A,B und C jedoch Teile, an denen mehrere Entwickler gleichzeitig arbeiten sollen, dann behindert die vorgegebene Reihenfolge einen flüssigen Entwicklungsprozess. Aus den in der

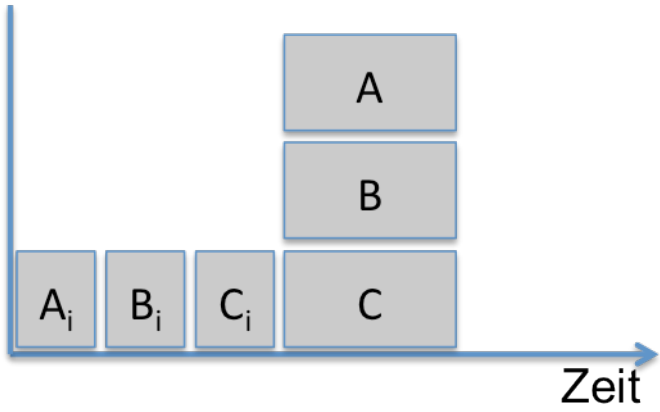
Abbildung gezeigten Abhängigkeiten ergibt sich für A, B und C folgende Entwicklungsreihenfolge:



Die Lösung der Herausforderungen: Komponenten benötigen Kontrakte. Durch diese wird die Reihenfolge der Entwicklung von den Abhängigkeiten entkoppelt.



Durch die Kontrakte ergibt sich für die Komponenten eine beliebige Reihenfolge bei der Implementation. Zeitlich gesehen müssen erst die Kontrakte realisiert werden. Die Reihenfolge für die Realisierung der Kontrakte ist mehr oder weniger willkürlich, weil zwischen ihnen keine Abhängigkeiten bestehen. Es können lediglich Abhängigkeiten zu Datentypen existieren, die im Kontrakt stehen. Die Kontrakte sind vergleichsweise schnell umgesetzt, weil hierzu lediglich Interfaces erstellt werden müssen. Im Anschluss können die Komponenten in beliebiger Reihenfolge, vor allem auch zeitgleich, erstellt werden. In der folgenden Abbildung ist das dadurch deutlich gemacht, dass die Komponenten übereinander über einander angeordnet sind und somit zum gleichen Zeitpunkt realisiert werden können.



Lösung: Kontrakte

2.2.1 Wildwuchs von Abhängigkeiten, ungeplante Abhängigkeiten

Abhängigkeiten lassen sich in Softwaresystemen nicht vermeiden. Entwickeln sie sich jedoch ungeplant, entsteht ein Wildwuchs von Abhängigkeiten, der nicht mehr zu durchschauen ist. Insofern muss ein Entwicklerteam stets Sorge tragen, dass sich die Abhängigkeiten in einer geordneten Art und Weise entwickeln. Werden die Abhängigkeiten sozusagen sich selbst überlassen, verstärkt sich das Problem selbst. Sogar bei gutem Willen ist es dann irgendwann nicht mehr möglich, an der Situation etwas zu ändern. Die Lösung kann daher nur darin liegen, die Abhängigkeiten zu planen. Statt drauf los zu programmieren und zuzuschauen, wie sich die Abhängigkeiten entwickeln, muss ein Team die Abhängigkeiten vor der Umsetzung planen. Des weiteren muss während der Umsetzung sichergestellt werden, dass die Abhängigkeiten nicht vom Entwurf abweichen.

Lösung: contract-first

2.2.2 Keine Übersicht über die Abhängigkeiten

Eine Konsequenz aus der ungeplanten Entwicklung von Abhängigkeiten ist die Unübersichtlichkeit. Sobald Abhängigkeiten einfach so entstehen, weil ein Entwickler sie nach Bedarf herstellt, geht die Übersicht verloren. Der Blick auf das große Ganze ist nur möglich, wenn Strukturen existieren, die abstrakter sind als Quellcode. Schon aus diesem Grund sind Kontrakte notwendig. Doch für die Übersicht ist es erforderlich, die Kontrakte vor der Umsetzung zu entwerfen.

Lösung: contract-first

2.2.3 Konflikte bei der Quellcodeorganisation

Sobald mehrere Entwickler beginnen, gemeinsam an einer Quellcodebasis zu arbeiten, steht das Risiko von Konflikten beim Zugriff im Raum. Selbstverständlich wird dabei natürlich ein Versionskontrollsystem eingesetzt. Doch selbst beim Einsatz eines noch so leistungsfähigen Merge-Tools ist die effizientere Alternative stets, ohne Konflikte auszukommen. Einerseits soll das Team gemeinsam an einem Feature arbeiten, andererseits sollen dabei aber keine Konflikte auf Quellcodeebene auftreten. Dieser scheinbare Widerspruch wird aufgelöst, in dem jede Komponente eine abgeschlossene Einheit bildet, auch auf Quellcodeebene. Für .NET bedeutet das konkret, jede Komponente in einer eigenen Solution abzulegen. Innerhalb der Solution befinden sich dann die benötigten Projekte: in der Regel mindestens eines für die Implementation der Komponente, sowie ein weiteres für die zugehörigen Tests. Ergebnis dieser so genannten Komponentenwerkbank ist eine Assembly, die an anderer Stelle weiterverwendet wird. Die Details zur Ausgestaltung einer Komponentenwerkbank folgen im Kapitel Verzeichnisstruktur und Projektorganisation.

Lösung: Komponentenwerkbänke

2.2.4 Zu breiter Scope, zu geringer Fokus

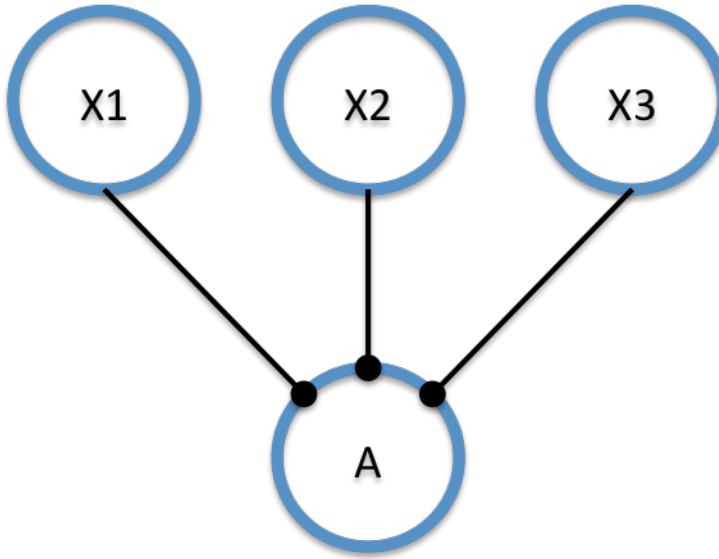
Jeder Entwickler hat schon die sprichwörtliche Situation erlebt, vor lauter Bäumen den Wald nicht mehr zu sehen. Plötzlich hat man so viele Dinge im Kopf, dass man nicht mehr weiß, wo man anfangen soll. Gibt der Quellcode dann keinen Rahmen vor, verliert man sich schnell und wird völlig unproduktiv. Statt im Quellcode jeweils alles zu sehen, sollte der Scope deutlich enger gefasst sein und dadurch Fokus bieten. Auch diese Herausforderung wird mit Hilfe von Komponentenwerkbänken gemeistert. In einer Komponentenwerkbank sehe ich jeweils nur den Quellcode, der genau zu dieser einen Komponente gehört. Das schafft Klarheit und bietet Fokus.

Lösung: Komponentenwerkbänke

2.3 Arten von Abhängigkeiten

Bis hier her wurde nun bereits mehrfach betont, dass Abhängigkeiten in einem Softwaresystem unvermeidbar sind. Im weiteren Verlauf wird mit der Komponentenorientierung eine Lösung beschrieben, wie damit auf der Ebene der Arbeitsorganisation umzugehen ist. Doch zuvor soll noch auf zwei grundsätzlich unterschiedliche Formen von Abhängigkeiten eingegangen werden. Daraus lassen sich hilfreiche Schlüsse für den Umgang mit Abhängigkeiten ziehen. Es gibt zwei grundsätzlich verschiedene Arten von Abhängigkeiten: • Viele Funktionseinheiten sind von einer anderen Funktionseinheit abhängig. • Eine Funktionseinheit ist von vielen anderen Funktionseinheiten abhängig. Natürlich gibt es dazwischen alle möglichen Varianten. Dennoch hilft es, sich Gedanken zu machen über diese beiden Extremfälle.

2.3.1 Variante 1: X1, X2, X3 sind von A abhängig

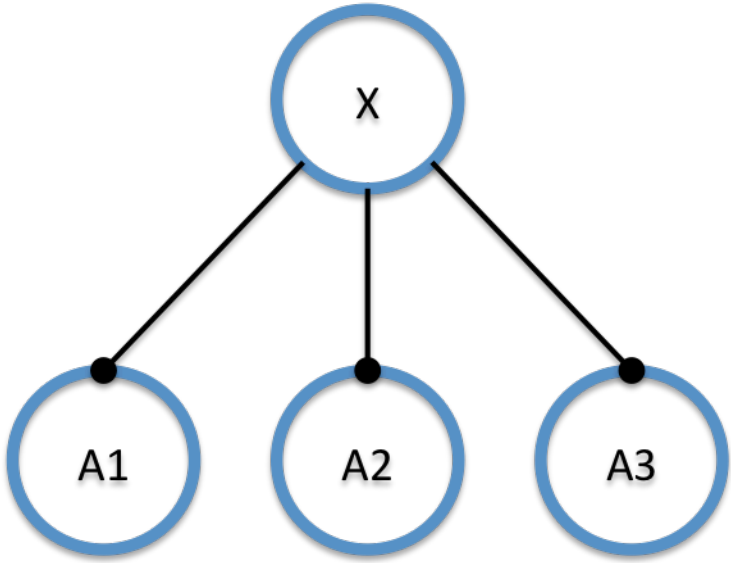


Bei Variante 1 sind X1, X2 und X3 von A abhängig und dadurch potentiell von jeder Änderung an A betroffen. Da viele von A abhängig sind, haben Änderungen an A jeweils starke Auswirkungen. Nun geht es nicht darum, diese Form der Abhängigkeit zu verteufeln mit dem Ziel, sie zu vermeiden. Es gibt immer wieder gute Gründe dafür, dass Abhängigkeiten in dieser Weise auftreten. Wenn das der Fall ist, sollten wir jedoch eine Konsequenz daraus ziehen: wir sollten uns fragen, ob wir an der Beschaffenheit der X1, X2 und X3 und des A etwas tun können, so dass die Auswirkungen keine große Bedeutung haben. Zu betrachten sind dabei zwei Aspekte: die Häufigkeit mit der A geändert wird, sowie das Maß der Auswirkung auf X1, X2 und X3.

Vereinfacht gesagt erhöht sich die Wahrscheinlichkeit, dass an A Änderungen vorgenommen werden müssen, mit dem Codeumfang. Je mehr Code A enthält, desto häufiger ist A von Änderungen

betroffen. Und desto häufiger sind auch die X1, X2 und X3 von den Änderungen betroffen. Folglich sollte es bei dieser Konstellation der Abhängigkeiten erstrebenswert sein, A so einfach wie möglich zu halten. Im Idealfall enthält A überhaupt keinen Logikcode sondern ist lediglich eine Datenstruktur. Betrachten wir auch den anderen Aspekt, das Maß der Auswirkungen auf X1, X2 und X3. Auch hier gilt, dass die Auswirkungen deutlich sind, je mehr Logikcode A enthält. Denn je mehr Logik in A steckt, desto größer ist die Wahrscheinlichkeit, dass die X1, X2 und X3 an Änderungen dieser Logik angepasst werden müssen. Auch hier lautet also die Folgerung, dass A möglichst einfach gehalten sein soll, weil dann die Auswirkungen auf X1, X2 und X3 nicht so groß sind.

2.3.2 Variante 2: X ist von A1, A2, A3 abhängig



Im zweiten Fall häufen sich die Abhängigkeiten in der anderen Richtung. Ein X ist von vielen anderen Funktionseinheiten abhängig. Damit muss X immer dann angepasst werden, wenn sich bei

A1, A2 oder A3 etwas ändert. Auch hier können wir überlegen, welche Forderungen sich daraus ergeben. An der Struktur der Abhängigkeiten wollen wir auch hier nicht rütteln. Die Anzahl der Abhängigkeiten zu verändern, scheidet somit als Strategie aus. Daraus ergibt sich, dass X von den Änderungen an den A1 – A3 immer betroffen ist, egal wie wir uns drehen und wenden. Zu überlegen ist daher, wie X beschaffen sein sollte, damit die Auswirkungen dieser Änderungen leicht beherrschbar bleiben.

Betrachten wir, was passiert, wenn X komplizierte Logik enthält. Diese Logik muss potentiell bei einer Änderung an einem der A1, A2, A3 angepasst werden. Ist die Logik kompliziert, fällt die Anpassung vermutlich schwierig aus. Ist darüber hinaus auch noch sehr viel Logik enthalten, steigt die Wahrscheinlichkeit weiter, dass X tatsächlich angepasst werden muss. Die Lösung besteht also darin, X möglichst einfach zu halten. Im Idealfall enthält X keine Logik, denn dann haben die Änderungen an A1, A2 oder A3 keinen Einfluss auf X. Wenn denn „keine Logik" nicht vorstellbar ist, dann sollte die Logik wenigstens sehr einfach gehalten sein.

Zu erwähnen sei hier noch, dass sich die Betrachtung der enthaltenen Logik immer auf dieselbe Domäne beziehen muss. Enthalten die A's und X's beispielsweise Logik aus dem Bereich der Anwendungslogik, gilt oben gesagtes uneingeschränkt. Ein anderer Fall liegt allerdings in folgendem Beispiel vor: ein IoC Container ist dafür zuständig, die Abhängigkeiten zwischen Funktionseinheiten während der Laufzeit aufzulösen. Dazu muss der Container zwangsläufig alle Funktionseinheiten kennen, die zur Erfüllung von Abhängigkeiten zur Verfügung stehen. Somit liegt hier also der Fall vor, dass ein Container A von vielen Typen X1, X2 und X3 abhängig ist. Die Folgerung wäre somit, den Container A so einfach wie möglich zu gestalten. Doch ein Container ist ein ziemlich kompliziertes Stück Software und enthält daher sehr viel Logik. Dieser scheinbare Widerspruch löst sich auf, sobald man sich klarmacht, dass es hier um zwei unterschiedliche Domänen geht. Die Typen, die vom Container verwaltet werden, enthalten

zwar möglicherweise ebenfalls sehr viel Logik. Diese gehört jedoch nicht zur Domäne des Containers. Gegenstand der Betrachtung sollten Abhängigkeiten sein, in denen die Logik von Abhängigen und Unabhängigen sich in derselben Domäne befinden.

3 Verwendete Notation

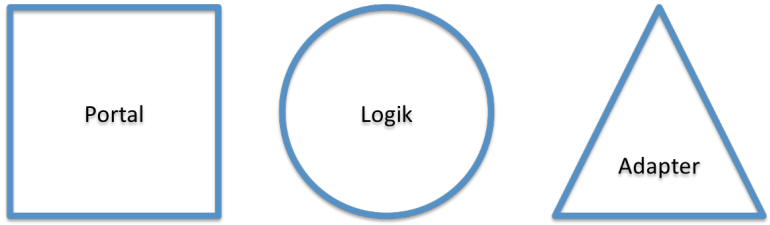
Die in diesem Buch gewählte Notation für Entwürfe besteht aus wenigen Symbolen. Das macht die Entwürfe leicht verständlich. Vor allem soll es Teams ermutigen, gemeinsam zu entwerfen. Wenn vor dem gemeinsamen Entwurf zunächst umfangreiche Symbolbibliotheken erlernt werden müssen, steht dies der Teamarbeit im Weg. Ferner ist durch die Verwendung weniger, einfacher Symbole kein Software-Werkzeug erforderlich: Papier und Stifte oder ein Whiteboard genügen völlig.

Die Notation besteht aus drei Symbolen für *Funktionseinheiten* sowie zwei unterschiedlichen Verbindungen. Mit einer der Verbindungen werden *Abhängigkeiten* zwischen zwei Funktionseinheiten notiert, mit der anderen *Datenflüsse*. Die überwiegende Anzahl der Beispiele im Buch benötigen lediglich die Abhängigkeitsverbindung. Im nächsten Kapitel Die Lösung am Beispiel finden Sie ein umfangreicheres Beispiel, in dem die Notation verwendet wird.

3.1 Funktionseinheiten

Funktionseinheiten können in drei Ausprägungen vorliegen, je nachdem zu welchem Aspekt der darin enthaltene Code gehört:

- Portal
- Provider
- Logik



Als Aspekt wird hier eine Menge von zusammengehörigen Eigenschaften bezeichnet, die sich getrennt von einer anderen Menge von Eigenschaften verändern. Es dient der Verständlichkeit und somit der Evolvierbarkeit, wenn Aspekte in Softwaresystemen getrennt werden. Die drei Ausprägungen *Portal*, *Provider* und *Logik* sind für drei so fundamental unterschiedliche Aspekte verantwortlich, dass es wichtig ist, diese schon im Entwurf anhand unterschiedlicher Symbole zu unterscheiden.

3.1.1 Portal

Ein Portal ist eine Funktionseinheit, deren Aufgabe die Interaktion mit dem Client ist. Der Begriff Client ist hier sehr weitgehend gemeint. Es kann damit ein Anwender bezeichnet sein, der mithilfe einer grafischen Benutzeroberfläche mit dem System interagiert. In dem Fall wäre das Portal vielleicht mit WPF oder WinForms realisiert. Im anderen Fall kann es aber auch ein Anwender mit einer Konsolenschnittstelle sein. Tatsache ist in beiden Fällen, dass das Portal von einem bestimmten API wie beispielsweise WPF, WinForms oder Console abhängt.

Eine andere Ausprägung von Client kann aber auch ein WebService sein. In dem Fall stellt die WebService Schnittstelle das Portal gegenüber dem Client dar. Hier ist das Portal dann von einem API wie beispielsweise WCF abhängig.

3.1.2 Provider

Mit einem Provider[1] tritt das Softwaresystem mit seiner Umwelt in Kontakt. Auch hier besteht eine Abhängigkeit zu einem API. In der Regel ist ein Provider von einem API abhängig, der sich um Ressourcen wie das Dateisystem, eine Datenbank, den Drucker, die Systemzeit, oder ähnliches kümmert. Wie auch das Portal dient der Provider hier dazu, das Softwaresystem von der Umgebung zu kapseln. Im Gegensatz zum Portal ist beim Provider jedoch das System der Client.

3.1.3 Logik

Der ganze Rest von Funktionseinheiten wird mit Logik bezeichnet. Dabei geht es in den meisten Fällen um die Geschäfts- oder Domänenlogik in Abgrenzung zu ganz allgemeiner Logik.

3.2 Abhängigkeiten

Abhängigkeiten zwischen Funktionseinheiten werden durch eine Verbindungslinie mit einem Kuller am einen Ende dargestellt. Der Kuller weist auf die Unabhängige Funktionseinheit. Die folgende Abbildung zeigt, dass A von B abhängig ist, bzw. umgekehrt, dass B von A unabhängig ist.

Abhängigkeit: A ist abhängig von B

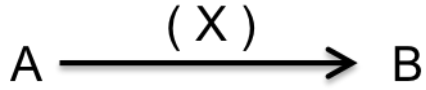
3.3 Datenflüsse

In vielen Fällen ist es hilfreich, zu verstehen, welche Daten zwischen Funktionseinheiten fließen. Mit *Flow Design* steht sogar

[1] In früheren Texten haben wir Provider als Adapter bezeichnet.

eine vollständige Entwurfsmethode zur Verfügung, die sich ganz deutlich auf Datenflüsse konzentriert. Im Kontext dieses Buches über Komponentenorientierung werden Datenflüsse eine eher untergeordnete Rolle spielen. Nichtsdestoweniger wird die Notation hier eingeführt, um sie an geeigneter Stelle einsetzen zu können.

Datenflüsse werden durch Pfeile symbolisiert. Dabei fließen die Daten in Pfeilrichtung. In der folgenden Abbildung fließen die Daten von A nach B.



Datenfluss: Daten X fließen von A nach B

Am Pfeil wird notiert, um welche Art von Daten es sich handelt.

4 Komponentenorientierung am Beispiel

Ein Beispiel soll nun zeigen, wie die komponentenorientierte Implementation einer ganz einfachen Anwendung aussieht.

4.1 Anforderungen

Implementiert werden soll die Kommandozeilenanwendung `cat`. Der aus der Unix Welt stammende Befehl `cat` gibt den Inhalt mehrerer Textdateien auf der Konsole aus.

- Die Dateinamen werden als Kommandozeilenparameter übergeben.
- Die Ausgabe der Dateien erfolgt in der Reihenfolge der Dateinamen.

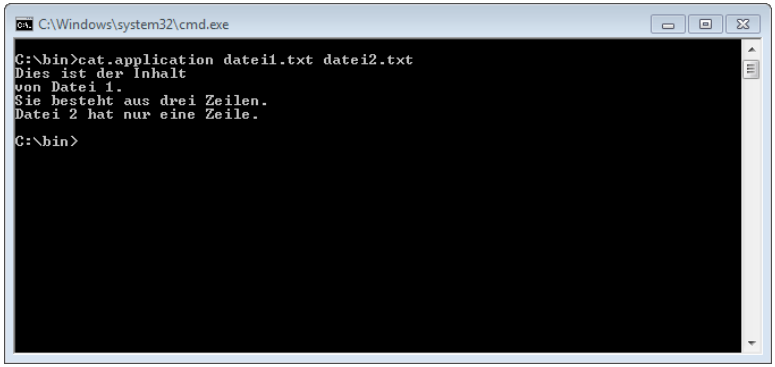
Beispiele:

```
cat file1.txt
```

Gibt die Datei file1.txt Zeile für Zeile auf der Konsole aus.

```
cat file1.txt file2.txt file3.txt
```

Gibt den Inhalt der drei Dateien nacheinander auf der Konsole aus. Die Ausgabe erfolgt in der Reihenfolge, in der die Dateinamen angegeben sind.

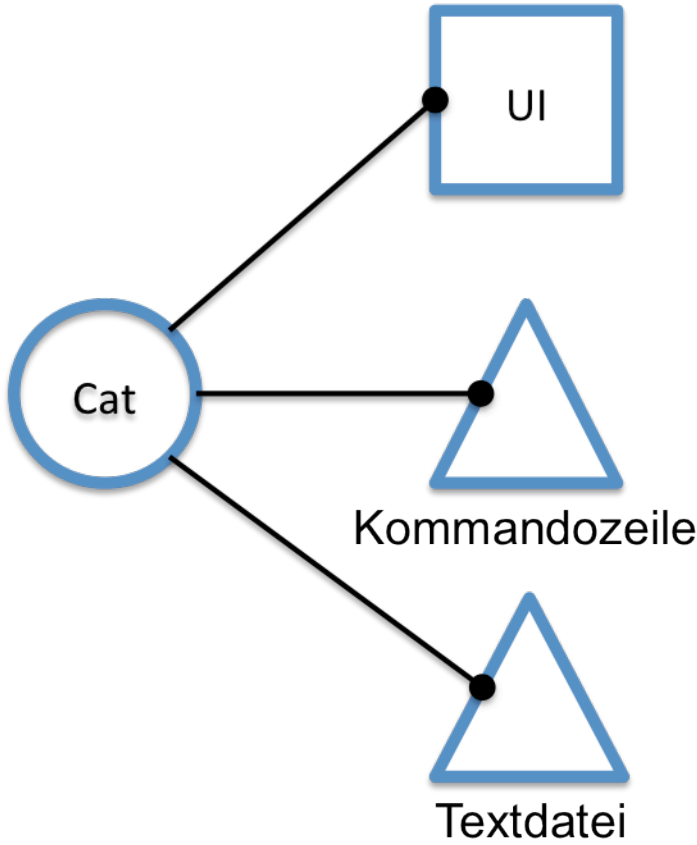


4.2 Entwurf

Es gibt viele Möglichkeiten, diese Anforderungen umzusetzen. In jedem Fall sollte vor der Implementation ein Entwurf stehen. Die Implementation erfolgt in textueller Weise in einer Programmiersprache. Dabei geht es um sehr viele Details. Der Entwurf dagegen soll einen Blick auf eine abstraktere Form ermöglichen, in der die Details ganz bewusst noch nicht auftauchen. Nur in dieser abstrakten Form lässt sich über mögliche Umsetzungen der Anforderungen im Team diskutieren. Ferner liegt der Entwurf nicht in textueller sondern in grafischer Weise vor, was die Diskussion darüber gut unterstützen kann.

Wie man zu einem geeigneten Entwurf kommt, ist eine spannende Frage. Da mir jedoch in diesem Buch der Fokus auf die Arbeitsorganisation ganz wichtig ist, wird es hier nicht darum gehen, wie man zu einem Entwurf kommt. Die Frage, die sich im Folgenden stellt ist, wie man einen vorhandenen Entwurf auf Komponenten verteilt.

Die folgende Abbildung zeigt einen Entwurf für die Umsetzung der Anforderungen. In diesem Entwurf wird mit Klassen gearbeitet, die zueinander in Abhängigkeiten stehen. Die einzelnen Klassen sind jeweils für einen Aspekt der Anwendung verantwortlich.

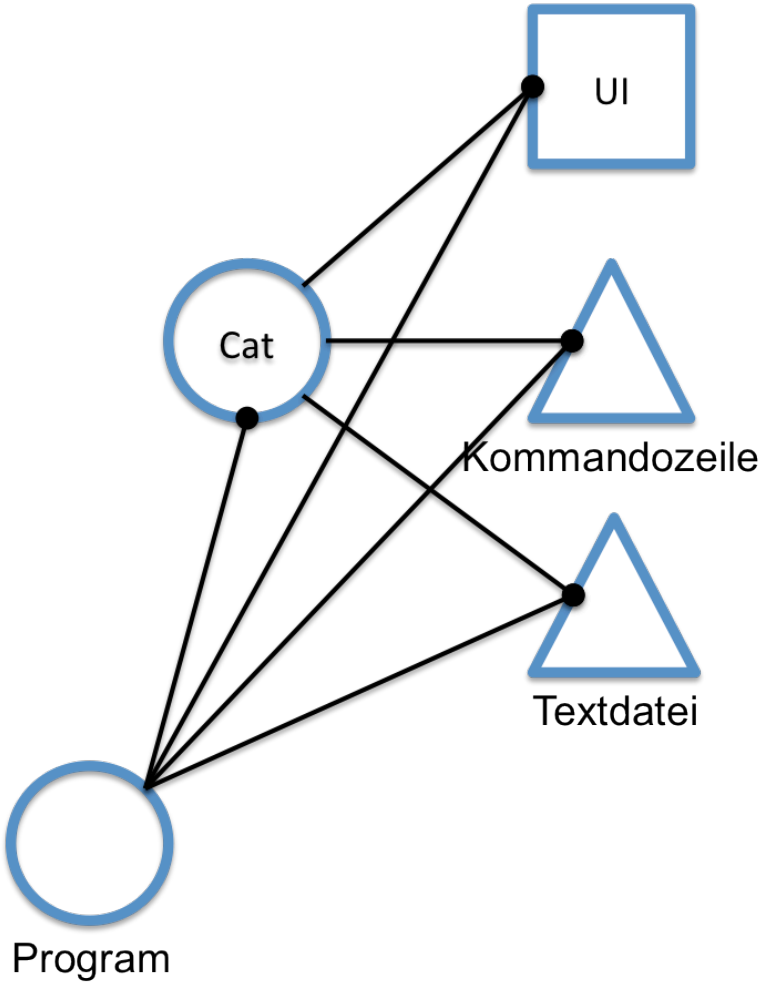


Die Klasse `Cat` ist die zentrale Funktionseinheit dieses Entwurfs. Sie ist von drei weiteren Klassen abhängig. Anhand der Formen der Funktionseinheiten wird bereits deutlich, zu welcher Kategorie sie gehören:

- `UI` ist ein *Portal*. Hier findet die Ausgabe an den Benutzer statt.
- `Kommandozeile` ist ein *Provider*. Mit diesem Provider wird auf die Ressource Kommandozeilenparameter in der Umwelt des Systems zugegriffen.

- `Textdatei` ist ein *Provider*. Er ist für den Zugriff auf die Textdateien zuständig, die ebenfalls als Ressource in der Umwelt des Systems liegen.
- `Cat` enthält die *Logik* des Systems.

Im Entwurf nicht dargestellt ist die Klasse `Program`, die für die Integration der restlichen Klassen zuständig ist. Ergänzt man den Entwurf um diese Klasse sowie die Abhängigkeiten, entsteht folgendes Bild:



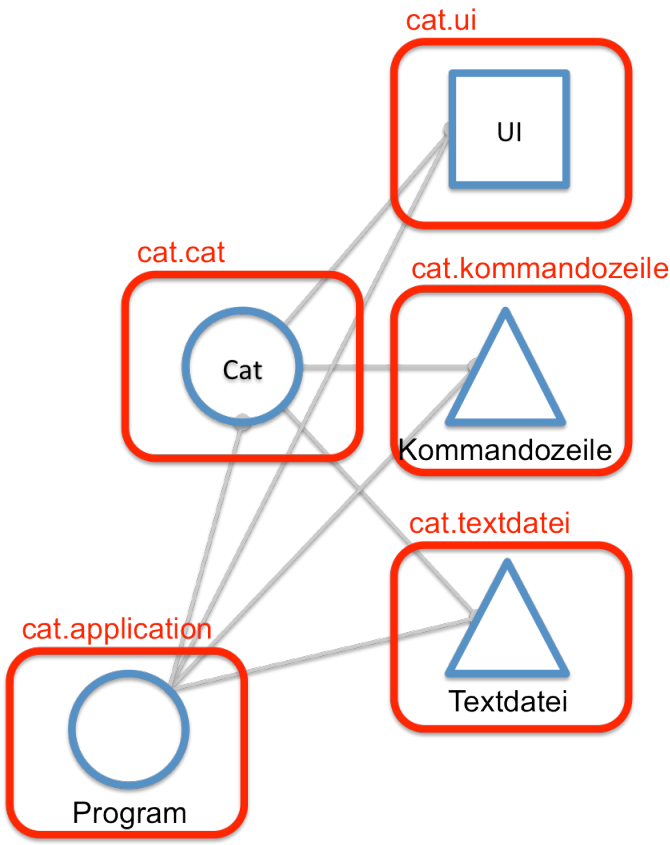
Die Abbildung wird durch die Ergänzung der Abhängigkeiten sehr unübersichtlich. Das liegt daran, dass die Klasse `Program` von allen anderen Klassen abhängig ist. Sie muss alle diese Klassen instanziieren und die Abhängigkeiten dieser Klassen untereinander auflösen. Weil die Klasse `Program` typischerweise von allen anderen Klassen abhängig ist und die Abbildungen dadurch sehr unübersichtlich

werden, lassen wir sie in zukünftigen Entwürfen weg.

4.3 Zerlegung in Komponenten

Nun liegt also ein Entwurf vor, in dem die einzelnen Aspekte des Systems auf Klassen verteilt wurden. Versetzen Sie sich nun einmal gedanklich in die Situation, dass ein Team mit mehreren Entwicklern jetzt mit der Implementation beginnen möchte. Die Herausforderung besteht darin, die gemeinsame Arbeit am System so zu organisieren, dass alle Entwickler gleichzeitig arbeiten können. Alle Klassen in ein und demselben Visual Studio Projekt anzulegen scheidet aus. Das würde dazu führen, dass alle Entwickler dasselbe Projekt öffnen und darin Klassen anlegen. Spätestens beim Übertragen in die Versionskontrolle käme es zu Problemen durch Mergekonflikte. Es liegt daher nahe, die Klassen auf mehrere Visual Studio Projekte zu verteilen. Das Beispiel ist bewusst klein gehalten, damit der Überblick gewahrt bleibt. Allerdings führt das nun dazu, dass die Zerlegung des Systems in Komponenten etwas übertrieben erscheinen mag. Lassen Sie sich davon nicht irritieren. In realen Systemen enthalten die Komponenten typischerweise mehr als eine Klasse.

Die folgende Abbildung zeigt, wie eine Zuordnung der Klassen zu Komponenten aussehen könnte. Ich habe für jede Klasse eine eigene Komponente vorgesehen. Geleitet hat mich dabei, dass die Klassen jeweils für völlig unterschiedliche Aspekte zuständig sind. Ich möchte vermeiden, dass unterschiedliche Aspekte eines Systems in einer Komponente zusammengefasst werden. Auf die Kriterien für das Zerlegen eines Entwurfs in Komponenten wird später noch detaillierter eingegangen.



Für die Bezeichnung der Komponenten verwende ich folgende Konvention:

- Alle Namen der Komponenten beginnen mit dem *Namen des Systems*. In diesem Beispiel ist das `cat`.
- Auf den Systemnamen folgt der Bezeichner für die *Komponente*, getrennt durch einen Punkt.
- Die Bezeichnung der Komponenten erfolgt vollständig in Kleinbuchstaben. Für jede Komponente muss ein Visual Studio Projekt angelegt werden. Beim Anlegen eines Projektes

übernimmt Visual Studio den Projektnamen als *Default Namespace*. Die Verwendung von Kleinbuchstaben hat den Vorteil, dass damit auch der Default Namespace in Kleinbuchstaben angelegt wird. Dadurch entstehen keine Konflikte zu Klassennamen.

Speziell die Konvention, Komponenten mit Kleinbuchstaben zu bezeichnen, sollten Sie unbedingt übernehmen. Bei einem komponentenorientierten System tritt häufig der Fall ein, dass eine Klasse genauso heißt, wie die Komponente. Da für jede Komponente ein Namespace angelegt wird, käme es immer zu einem Konflikt zwischen Namespace und Klassenname. Man müsste dann den Klassennamen jeweils durch den vorangestellten Namespace qualifizieren, also zum Beispiel folgendes schreiben:

```
var cat = new Cat.Cat();
```

Durch Namespaces in Kleinbuchstaben entfällt die Notwendigkeit, den Namespace vor den Klassennamen schreiben zu müssen.

4.4 Erstellen der Kontrakte

Für das Erstellen der Kontrakte ist es natürlich erforderlich, dass ein Entwurf vorliegt. Bislang sind in den Abbildungen zum Entwurf allerdings nur die Klassennamen gezeigt. Über welche Methoden die Klassen verfügen, geht daraus noch nicht hervor. Selbstverständlich ist das der zentrale Punkt eines Entwurfs: herauszufinden, welche Funktionalität benötigt wird und wie man sie auf Methoden, Klassen und Komponenten verteilt. Da das Beispiel überschaubar ist und der Fokus auf der Arbeitsorganisation mittels Komponenten liegt, werde ich hier nicht weiter ausführen, wie ich auf die Methoden gekommen bin. Entwurf ist Thema für ein anderes Buch.

4.4.1 Der Kontrakt ICat

```
namespace cat.contracts
{
    public interface ICat
    {
        void Run();
    }
}
```

Die `Run` Methode ist der Einstiegspunkt der Anwendung. Sie wird später in der `Program.Main` Methode des EXE-Projektes aufgerufen. Ihre Aufgabe ist die Integration der anderen Funktionseinheiten. Sie koordiniert den Aufruf der Methoden der anderen beteiligten Klassen.

4.4.2 Der Kontrakt IUi

```
using System.Collections.Generic;

namespace cat.contracts
{
    public interface IUi
    {
        void Ausgeben(IEnumerable<string> zeilen);
    }
}
```

Die Komponente `UI` ist für die Ausgabe von Textzeilen zuständig. Sie verfügt dazu über die Methode `Ausgeben`, die eine Aufzählung von Strings als Parameter erhält und diese auf die Konsole ausgibt.

4.4.3 Der Kontrakt IKommandozeile

```
using System.Collections.Generic;

namespace cat.contracts
{
    public interface IKommandozeile
    {
        IEnumerable<string> Dateinamen();
    }
}
```

Die Komponente `Kommandozeile` ist ein Provider für die Ressource *Kommandozeilenparameter*, die sich in der Umwelt des zu erstellenden Systems befinden. Sie verfügt über eine Methode `Dateinamen`, mit der alle Dateinamen, die auf der Kommandozeile übergeben wurden, ermittelt werden.

4.4.4 Der Kontrakt ITextdatei

```
using System.Collections.Generic;

namespace cat.contracts
{
    public interface ITextdatei
    {
        IEnumerable<string> Einlesen(string dateiname);
    }
}
```

Der Zugriff auf den Inhalt der einzelnen Dateien erfolgt durch die Komponente `Textdatei`. Sie enthält die Methode `Einlesen`, die den gesamten Inhalt einer Datei als Aufzählung von `Strings` liefert.

4.5 Implementieren der Komponenten

Die Aufgabenstellung des Programms ist überschaubar. Daher sind die Komponenten nicht sehr umfangreich. Jede Komponente ist als einzelne Klasse realisiert. Das muss natürlich nicht immer so sein. In größeren Systemen bestehen Komponenten durchaus aus mehreren Klassen.

4.5.1 Die Komponente Cat

Aufgabe der Komponente `Cat` ist die Integration der drei Komponenten `UI`, `Textdatei` und `Kommandozeile`. Voraussetzung dafür ist, dass `Cat` die drei Komponenten kennt. Natürlich darf hier aber keine direkte Abhängigkeit zwischen den Komponenten entstehen. `Cat` muss die Dienste der anderen Komponenten in jedem Fall über den *Kontrakt* in Anspruch nehmen. Die Kontrakte werden in C# typischerweise durch *Interfaces* realisiert.

```
1  using cat.contracts;
2
3  namespace cat.cat
4  {
5      public class Cat : ICat
6      {
7          private readonly IUi ui;
8          private readonly ITextdatei textdatei;
9          private readonly IKommandozeile kommandozeile;
10
11         public Cat(IUi ui, ITextdatei textdatei, IKomma\
12 ndozeile kommandozeile) {
13             this.ui = ui;
14             this.textdatei = textdatei;
15             this.kommandozeile = kommandozeile;
16         }
```

```
17
18          public void Run() {
19              var dateinamen = kommandozeile.Dateinamen();
20              foreach (var dateiname in dateinamen) {
21                  var zeilen = textdatei.Einlesen(dateina\
22  me);
23                  ui.Ausgeben(zeilen);
24              }
25          }
26      }
27  }
```

`Cat` kann die benötigten Klassen nicht selbst instanzieren. Dazu wäre eine Referenz auf die Implementation der Komponenten erforderlich. Das würde die Komponentenorientierung ad absurdum führen. Aus diesem Grund werden die drei benötigten Komponenten dem Konstruktor von `Cat` als Parameter übergeben und in Feldern der Klasse abgelegt. Dadurch hat die Methode `Run` Zugriff auf die Komponenten und kann deren Methoden in der erforderlichen Weise aufrufen.

Am Beispiel der Komponente `Cat` kann man klar erkennen, dass `Cat` implementiert werden kann, ohne dass die drei anderen Komponenten bereits existieren. Lediglich die Kontrakte müssen vorliegen. Auf diese Weise wird eine Arbeitsorganisation im Team ermöglicht, die ein gleichzeitiges Arbeiten an den Komponenten zulässt.

4.5.2 Die Komponente UI

Die Ausgabe von `Strings` auf der Konsole ist die Aufgabe der Komponente `UI`. Ihre Methode `Ausgeben` erhält eine Aufzählung von `Strings` als Parameter. Darüber iteriert die Methode in einer Schleife und gibt jeden `String` mit `Console.WriteLine` auf die Konsole aus.

Keine große Sache; möglicherweise entsteht daher der Wunsch, diese Funktionalität im Hauptprogramm unterzubringen. Schließlich erscheint der Overhead für das Erstellen der Komponente relativ groß im Verhältnis zu den wenigen Zeilen Code, welche die Funktionalität erbringen. Doch es ist ganz wichtig, hier nicht die falschen Kriterien anzulegen. Die Anzahl der Codezeilen sollte kein Kriterium sein für die Frage, ob es sich „lohnt“, eine weitere Komponente zu erstellen. Stattdessen sollte die Frage im Vordergrund stehen, ob die Komponente dazu beiträgt, die Aspekte des Systems zu trennen. Und das ist hier bei der Komponente `UI` definitiv gegeben. Die Komponente `UI` isoliert den Aspekt der Ausgabe der Daten. Dieser Aspekt kann sich getrennt von anderen Aspekten des Systems verändern. Es könnte zum Beispiel der Wunsch entstehen, das System mit einer grafischen Oberfläche auszustatten. In diesem Fall wäre die Komponente `UI` zu ändern. Die anderen Komponenten sollten nicht geändert werden müssen. Andernfalls wäre das ein Hinweis darauf, dass die Aspekte nicht klar getrennt sind.

```
1  using System;
2  using System.Collections.Generic;
3  using cat.contracts;
4
5  namespace cat.ui
6  {
7      public class Ui : IUi
8      {
9          public void Ausgeben(IEnumerable<string> zeilen\
10 ) {
11             foreach (var zeile in zeilen) {
12                 Console.WriteLine(zeile);
13             }
14         }
15     }
16 }
```

4.5.3 Die Komponente Kommandozeile

Das Programm erhält die Namen der auszugebenden Textdateien als Parameter auf der Kommandozeile übergeben. Für den Zugriff auf diese Parameter ist die Komponente `Kommandozeile` zuständig. Ihre Methode `Dateinamen` liefert die Kommandozeilenparameter als Aufzählung von `Strings`.

Auch hier scheint die geringe Anzahl von Codezeilen dafür zu sprechen, die benötigte Funktionalität im Hauptprogramm unterzubringen. Zumal die Methode `Program.Main`, die zur Laufzeit als Einstiegspunkt in das Programm dient, die Kommandozeilenparameter als Methodenparameter übergeben kriegt. Doch auch hier geht es darum, die Aspekte zu trennen. Die Anforderungen könnten sich beispielsweise so ändern, dass die Dateinamen nicht über die Kommandozeile übergeben werden, sondern aus einer Steuerdatei gelesen werden sollen. In diesem Fall wäre lediglich die Komponente `Kommandozeile` von der Änderung betroffen. Das Trennen der Aspekte ist somit gut für die Evolvierbarkeit des Systems.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using cat.contracts;
5
6  namespace cat.kommandozeile
7  {
8      public class Kommandozeile : IKommandozeile
9      {
10         public IEnumerable<string> Dateinamen() {
11             return Environment.GetCommandLineArgs().Ski\
12 p(1);
13         }
14     }
15 }
```

4.5.4 Die Komponente Textdatei

Das Lesen der Zeilen der Textdatei ist Aufgabe der Komponente `Textdatei`. Sie verfügt über eine Methode `Einlesen`, die den Dateinamen der einzulesenden Datei als Parameter erhält. Als Ergebnis liefert die Methode den Inhalt der Datei als Aufzählung von `Strings`.

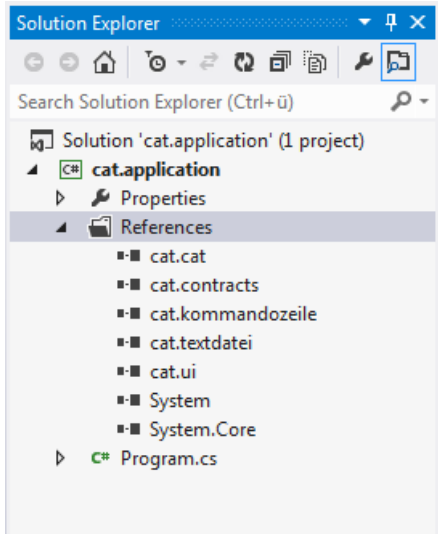
```
1  using System.Collections.Generic;
2  using System.IO;
3  using cat.contracts;
4
5  namespace cat.textdatei
6  {
7      public class Textdatei : ITextdatei
8      {
9          public IEnumerable<string> Einlesen(string date\
10 iname) {
11             return File.ReadLines(dateiname);
12         }
13     }
14 }
```

4.6 Implementieren der App

Die Funktionalität des gesamten Systems ist nun auf die oben beschriebenen Komponenten verteilt. Was nun noch fehlt, ist eine *App*, die alle benötigten Komponenten referenziert und die benötigten Klassen instanziert. Schließlich muss der Einstiegspunkt des Systems, in diesem Fall die Methode `Cat.Run`, aufgerufen werden.

Das Visual Studio Projekt `cat.application` ist das einzige Projekt, das Referenzen auf die Komponentenimplementationen erhält. Alle anderen Projekte referenzieren lediglich die Kontrakte. Auf diese

Weise wird erreicht, dass die Komponenten in beliebiger Reihenfolge und auch parallel entwickelt werden können. Die folgende Abbildung zeigt die Referenzen des App Projekts.



Mit der Implementation der App kann logischerweise erst begonnen werden, wenn alle Implementationen der Komponenten vorliegen. Allerdings müssen die Komponenten dazu nicht vollständig implementiert sein, sondern es genügt, die Visual Studio Projekte aufzusetzen. Die Implementationen der einzelnen Methoden können zunächst leer gelassen werden. Im Ergebnis kann dann mit der Arbeit an der App begonnen werden, da das App Projekt dann bereits alle benötigten Komponenten referenzieren kann.

```
using cat.cat;
using cat.kommandozeile;
using cat.textdatei;
using cat.ui;

namespace cat.application
{
    internal class Program
    {
        private static void Main() {
            var ui = new Ui();
            var kommandozeile = new Kommandozeile();
            var textdatei = new Textdatei();

            var cat = new Cat(ui, textdatei, kommandoze\
ile);

            cat.Run();
        }
    }
}
```

Die App der Anwendung `Cat` instanziert zunächst die drei Klassen der Komponenten `UI`, `Kommandozeile` und `Textdatei`. Anschließend kann die Klasse `Cat` instanziert werden. Ihr werden im Konstruktor die drei anderen Instanzen der Klassen übergeben. Zum Abschluss ist nichts weiter zu tun, als mit `cat.Run` die Anwendung zu starten.