Prinzipien der Modularisierung

Michael Barth, Universität Ulm
michael.barth@uni-ulm.de

Zusammenfassung

Modularisierung ist eine essentielle Methode in der Softwareentwicklung um komplexe Probleme beherrschbar zu machen, die Flexibilität von Software zu erhöhen und ihre Wartbarkeit zu erleichtern. Welche Prinzipien & Kritieren können angewandt werden um zu einer optimalen Modularisierung zu gelangen? Gibt es überhaupt eine optimale Modularisierung oder müssen gewisse Vor- und Nachteile gegeneinander abgewogen werden um die gesteckten Ziele eines Softwaresystems zu erreichen?

Was der große Vorteil der Modularisierung von Softwaresystemen ist und welche Kosten eine Modularisierung mit sich bringen kann soll in der vorliegenden Arbeit beleuchtet werden. David Parnas veröffentlichte 1972 einen Artikel mit dem Titel 'On the Criteria To Be Used in Decomposing Systems into Modules' *[1] und führte ein wichtiges Prinzip ein: Das information hiding.*

Die Auswirkungen von Parnas' Artikel auf den heutigen Stand, auf andere Prinzipien, Ratschläge und Design Patterns, zu denen sich in Parnas' Artikel bereits Ansätze erkennen lassen, sollen in der vorliegenden Arbeit ausgeführt werden.

1 Einführung

Beim Software Engineering ist die Komplexität eines Softwaresystems eines der größten Probleme, die es zu bewältigen gilt. Ein verbreiteter und sehr beliebter Ansatz ist es, das Problem solange in kleinere Probleme aufzubrechen (teilen), bis man diese in Isolation lösen kann (herrschen) – was einfacher ist als das Problem im Gesamten zu lösen, da weniger beachtet werden muss. Dies wird als *Divide and Conquer* (zu dt. *Teile und herrsche*) bezeichnet.

Häufig wird dies durch ein Modularisieren der Software umgesetzt: Das System wird nicht als ein großer, monolithischer Block entworfen, sondern in Module zerlegt, welche komplexe, sich verändernde Implementationen hinter klaren, beständigen Interfaces verbergen [2]. Dies erlaubt es, das Modul in Isolation zu entwerfen, zu entwickeln, zu testen und zu warten. Das reduziert die Komplexität, da die Aufgabe kleiner und überschaubarer, und somit leichter zu lösen wird.

Bereits 1972 sprach Parnas zum ersten Mal das Thema der Prinzipien und Kriterien, die man bei der Modularisierung eines Softwaresystems beachten sollte, an [1]. Er identifizierte dabei das Prinzip des *information hiding*, welches nach wie vor Gültigkeit besitzt und angewandt wird.

Doch das Prinzip des *information hiding* ist nicht die einzige Richtlinie, nach der sich ein Softwaresystem in Module auftrennen lässt. Weitere Prinzipien und Kriterien entstanden nach Parnas' Artikel. Welche das sind und wie sich Parnas' Artikel auf das Software Engineering ausgewirkt hat, soll in der vorliegenden Arbeit behandelt werden.

Zuerst soll ein Grundverständnis dafür geschaffen werden, was ein Modul ist, was beim Modularisieren geschieht und warum man diese Methode verwendet. Auch wird darauf eingegangen, welche Kosten eine Modularisierung mit sich bringt, wie der aktuelle Stand aussieht, wie Parnas Prinzip des *information hiding* verstanden und umgesetzt wurde, welche weiteren Prinzipien und Kriterien hinzukamen und wie diese Prinzipien durch Programmiersprachen und Tools unterstützt werden. Es werden ebenfalls Probleme besprochen, die sich beim Anwenden der behandelten Prinzipien gezeigt haben, sowie Lösungsansätze aufgezeigt.

2 Modularisierung

Der Begriff Modul wird häufig mit unterschiedlicher Bedeutung verwendet und einige Programmiersprachen, darunter alle objektorientierten Programmiersprachen wie *C++*, *Java* oder *Ruby*, bieten eigene Sprachkonstrukte (Klassen) an, um dieses Konzept direkt im Code auszudrücken. Um Missverständnisse zu vermeiden wird zuerst definiert, was mit dem Begriff "Modul" gemeint ist im Rahmen dieser Arbeit, was beim Modularisieren geschieht, sowie welche Vor- und Nachteile eine Modularisierung mit sich bringt.

2.1 Was ist ein Modul?

Ein Modul stellt eine abgeschlossene Einheit dar, eine Komponente, die bestimmte Funktionen zur Lösung von Aufgaben zur Verfügung stellt. Implementationsdetails, die zur Erbringung der Funktionalität nötig sind, werden dabei hinter einer Schnittstelle versteckt, deren Signatur sich möglichst nicht ändern sollte, also beständig ist.

Die Schnittstelle dient als Bindeglied zwischen den Nutzern der Funktionen, den zur Verfügung gestellten Funktionen eines Moduls und den Implementationsdetails. Die Schnittstelle sollte mit dem Ziel entworfen werden, dass sich mit ihr alle vom Modul erwarteten Aufgaben erfüllen lassen, ohne dabei Implementierungsdetails unnötig preiszugeben.

2.2 Was ist Modularisieren?

Modularisieren ist der Vorgang, bei dem Entwurfsentscheidungen getroffen werden, die bestimmen wie die Funktionen eines Softwaresystems auf Module aufgeteilt werden. Die Herausforderung beim Modularisieren liegt im Bestimmen der "besten" Aufteilung von Verantwortlichkeiten in Module und in der Definition der Schnittstellen, über welche diese Module miteinander interagieren können [3]. Es gibt verschiedene Prinzipien und Kriterien, auf die bei diesem Vorgang zu Achten ist. Diese Kriterien sollen helfen eine solche Aufteilung der Funktionen auf Module zu finden, dass die Erfüllung der Anforderungen an das System optimal unterstützt wird.

Es gilt besonders zu beachten, dass die Modularisierung *nicht* das Ziel eines Entwicklers darstellt, sondern ein Mittel zum Zweck – eine Methode – um die Anforderungen an die Software zu erreichen. Diese Anforderungen repräsentieren die eigentlichen Ziele, die man mit der Software erreichen will. [3].

2.3 Vorteile von Modularisierung

Softwareysteme können auf verschiedene Arten modularisiert werden, aber nicht alle Modularisierungen sind gleichwertig. Das Modularisieren bietet verschiedene Vorteile, von denen die wichtigsten in diesem Abschnitt kurz erläutert werden [3].

2.3.1 Reduzierte Komplextität

Für den individuellen Entwickler hilft der *Divide and Conquer*-Ansatz beim Beherrschen der Komplexität eines Softwaresystems, da Module für sich isoliert betrachtet, entwickelt, getestet und diskutiert werden können. Bereits 1974 stellten Stevens et al. fest: *"problem solving is faster and easier when the problem can be subdivided into pieces which can be considered separately. Problem solving is hardest when all aspects of the problem must be considered simultaneously."* [4]

2.3.2 Softwareevolution

Softwaresysteme, die sinnvoll in Module aufgeteilt wurden, sind leichter erweiter- und wartbar. Das Ziel bei einer guten Modularisierung ist es, sich verändernde Dinge in Modulen zu verstecken und zu isolieren. Dadurch können keine Abhängigkeiten auf die veränderlichen Details entstehen. Wenn diese Details sich dann ändern, kann die Änderung an einer zentralen Stelle vorgenommen werden – keine anderen Module oder Komponenten sind von dieser Änderung direkt betroffen.

Sauber getrennte Module lassen sich ebenfalls leichter austauschen. Dies erhöht die Flexibilität und Erweiterbarkeit einer Sofware beträchtlich.

2.3.3 Arbeitsaufteilung

Eine sinnvolle Auftrennung von Funktionen auf Module erlaubt eine Arbeitsteilung, bei der mehrere Entwickler parallel an verschiedenen Modulen arbeiten können. Die Schnittstellen sind hierbei essentiell um zu garantieren, dass die parallel entwickelten Module später miteinander funktionieren.

2.3.4 Wiederverwendung

Wurden allgemein nützliche Funktionen bei einem Softwaresystem identifiziert und günstig modularisiert, können diese – zusammen mit allen Abängigkeiten, die benötigt werden – aus dem System extrahiert und in anderen Projekten wiederverwendet werden.

2.4 Die Kosten von Modularisierung

Ein Softwaresystem zu modularisieren bringt nicht nur Vorteile mit sich. Welche Kosten beim Modularisieren entstehen können, wird in diesem Abschnitt kurz erläutert.

2.4.1 Performance

Beim Modularisieren versteckt man Implementationsdetails hinter Schnittstellen, das bedeutet man nutzt Programmiersprachenkonstrukte wie Unterprogramme zum Verstecken der Details. In objektorientierten Programmiersprachen bündelt man diese zusammen zu Klassen. Jede Indirektion[1] kostet geringfügig Rechenzeit und Speicherplatz.

Bei heutigen Computern ist dies zwar für die meisten Anwendungen vernachlässigbar, doch diese geringfügigen zusätzlichen Kosten können sich – an kritischen Stellen – aufsummieren und somit relevant werden. Für Anwendungen mit speziellen Anforderungen (z.B. Echtzeitanwendungen) sind selbst die kleinsten zusätzlichen Kosten relevant.

[1] Eine Indirektion ist die Möglichkeit, etwas über einen Namen, eine Referenz oder einen Container zu referenzieren statt den referenzierten Wert selbst zu verwenden. Funktionen/Methoden werden über eine Speicheradresse referenziert und aufgerufen. *Pointer* in C++ sind ein weiteres Beispiel für eine Indirektion.

2.4.2 Schwierige Einrichtung

Die verschiedenen Module später zu einem Gesamtsystem zusammenzusetzen und korrekt zu konfigurieren ist nicht immer trivial. Die Komposition der Module selbst kann sehr komplex werden, wenn viele voneinander abhängige Teile miteinander interagieren müssen oder viel Konfigurationsaufwand betrieben werden muss, bis die Module sich wie gewünscht verhalten.

Es kann auch schwierig sein nachzuvollziehen, wieso ein System ein bestimmtes Verhalten zeigt, wenn der Programmablauf durch komplizierte, wechselseitige Abhängigkeiten und Beziehungen zwischen den Modulen verschleiert wird. [3] [4]

2.4.3 Abnehmende Erträge

Beim Modularisieren besteht die Gefahr, es zu weit zu treiben und zu viele, zu kleine Module zu entwerfen, welche für sich eine kaum nennenswerte Arbeit leisten. Die Vorteile des Modularisierens werden sich hierbei immer geringfügiger auswirken, während die Kosten immer schwerer ins Gewicht fallen.

3 Auswirkungen und der heutige Stand

Die Auswirkungen von Parnas' Artikel sind weitreichend. Es lassen sich in dem Artikel von 1972 einige Ansätze entdecken, die erst später als eigene Prinzipien, *best practices* oder Design Patterns eine Bezeichnung erhielten.

3.1 Prinzipien zur Dekomposition

Die folgenden Prinzipien und Kriterien zur Dekomposition von Systemen in Module stellen nur eine unvollständige Liste der durch Parnas' Artikel beeinflussten Prinzipien und Kriterien dar. Dieser Abschnitt soll einen Einblick geben, wie sich Parnas' Artikel auf den heutigen Stand ausgewirkt hat.

3.1.1 Information Hiding

Die erwartete Verarbeitungsreihenfolge eines Programms ist ein schlechtes Kriterium für die Dekomposition eines Softwaresystems in Module, da hierbei die Verantwortlichkeiten oft über mehrere Module verteilt werden. Dies ist eine Kernaussage in Parnas' Artikel von 1972. Als Gegenmaßnahme schlug Parnas *information hiding* vor und prägte diesen Begriff. [1] *Information hiding* wird im Wesentlichen in drei Schritten angewandt:

1. Liste alle wichtigen Entwurfsentscheidungen auf, die sich ändern könnten (z.B. Veränderungen am Eingabeformat oder der Aufbau einer Datenstruktur).
2. Teile das System so in Module auf, dass jedes Modul eine dieser Entwurfsentscheidungen kapselt. Dies wird als Geheimnis des Moduls bezeichnet.
3. Entwerfe die Schnittstelle des Moduls so, dass sie sich möglichst nicht verändert, selbst wenn sich das Geheimnis verändert.

Dies führt dazu, dass sich Veränderungen weniger drastisch auf ein Softwaresystem auswirken, da sie sich auf eines oder wenige Module beschränken. Außerdem fördert es die Wiederverwendbarkeit, da eine Entwurfsentscheidung, gekapselt in einem Modul, eher eine wiederverwendbare Komponente darstellt als ein Verarbeitungsschritt herausgenommen aus einer Kette von zusammengehörigen Schritten. *Information hiding* stellt eines der Hauptprinzipien der Objektorientierung dar [5].

3.1.2 Separation of Concerns

Information hiding hat das Ziel, Entwurfsentscheidungen in Modulen zu kapseln. Betrachtet man diese Entwurfsentscheidungen auf einer höheren Ebene, stellt man fest, dass diese in Anliegen (engl. *concerns*) an das Softwaresystem wurzeln. Diese Anliegen stellen die Anforderungen an eine Software dar, welche erfüllt werden müssen, damit die Software ihrem Zweck gerecht wird.

Separation of concerns bezeichnet den Prozess, bei dem man versucht diese Anliegen möglichst gut voneinander zu trennen, um Wechselbeziehungen und sich überlappende Merkmale zu minimieren. Hierbei bedient man sich meist der Technik des Modularisierens und des Kapselns mithilfe von *information hiding*. *Separation of concerns* ist also auf einer höheren Ebene angesiedelt als *information hiding*, verfolgt aber ähnliche Ziele: Reduzierung der Komplexität, Erleichterung der Wartung, sowie der Erweiterbarkeit eines Softwaresystems und Förderung von Wiederverwendbarkeit [6].

Ein Beispiel für *separation of concerns* ist die aspektorientierte Programmierung, welche versucht das Problem der *cross-cutting concerns* zu lösen. Crosscutting concerns und aspektorientierte Programmierung werden in Kapitel 4.1 angesprochen.

3.1.3 Low Coupling and High Cohesion

Ein weiteres Kriterium einer guten Modularisierung ist es, die Kommunikation von Modulen untereinander zu betrachten, also die Verbindungen. Verbindungen eines Moduls zu anderen Modulen sollten minimiert werden, während die internen Verbindungen maximiert werden sollten. Verbindungen zu anderen Modulen stellen einen Pfad dar, auf dem sich Änderungen und Fehler in andere Teile des Softwaresystems ausbreiten können. Ändert man an einer Stelle im System etwas, kann sich dies als Fehler an anderen Stellen auswirken, was wiederum weitere Änderungen nötig macht, usw. [4]

Parnas beschreibt in seinem Artikel in Grundzügen genau dieses Prinzip in seinem Beispiel eines KWIC[2] Indexierungsprogramms, welches auf zwei verschiedene Arten modularisiert wird. In der ersten, ungünstigen Modularisierung wirken sich manche Änderungen auf das ganze System aus, während sie bei der zweiten, günstigeren Modularisierung innerhalb eines Moduls gekapselt werden. [1]

[2] *"Ein Permutiertes (alphabetisches) Register (...) ist eine besondere Form eines Registers, bei dem ganze Phrasen (Titel wie Buchtitel und Überschriften oder Schlagwortketten) mehrfach so permutiert aufgelistet werden, dass jeweils ein anderes Stichwort als hervorgehobener Registereintrag erscheint. Die Register werden in der Regel vollautomatisch erstellt (...)." [7]*

3.1.4 Don't Repeat Yourself

Bei *Don't Repeat Yourself* (kurz: DRY) handelt es sich um ein Prinzip, das eine ähnliche Idee verfolgt wie das *information hiding*: Jede Information soll eine einzelne und eindeutige Repräsentation im System haben. Dies soll garantieren, dass eine Änderung eines einzelnen Teils im System keine Änderungen in anderen Systemteilen erfordert. Das Kapseln von Entwurfsentscheidungen in Modulen oder Klassen ist eine Möglichkeit dieses Ziel zu erreichen, aber auch das Auslagern von Code in Methoden stellt eine Anwendung dieses Prinzips dar.

Duplizierter Code ist aber nicht generell ein Zeichen für schlechten Code. Es kann gute Gründe für duplizierten Code geben, wie Performanceverbesserungen oder das Verringern von Abhängigkeiten zwischen Modulen. [5]

3.1.5 Weitere Ansätze

Parnas beschreibt in seinem Artikel das Verstecken des Wissens, wie die Zeilen für das KWIC Indexierungsprogramm gespeichert und ausgelesen werden, in einem Modul. Ein ähnliches Prinzip verfolgt das *Iterator Design Pattern*, welches den Zugriff auf Daten von seiner Repräsentation mitsamt dem Wissen, wie diese Repräsentation ausgelesen werden kann, trennt.

Durch *information hiding* wird auch der Ratschlag *"Program to an interface, not an implementation"* [8] erfüllt, da das restliche Programm unabhängig von der Implementierung des Moduls zum Auslesen der Zeilen ist, es hängt einzig vom Interface des Moduls ab. Dadurch kann die Implementation später leicht ausgetauscht werden.

Ein weiteres Prinzip, zu dem sich Ansätze in Parnas Artikel finden lassen, ist das *Single Responsiblity Principle* der objektorientierten Programmierung: Es besagt das jede Klasse nur eine einzelne Verantwortlichkeit haben soll, die von der Klasse gekapselt wird. Dies basiert auf dem Prinzip der Kohäsion und des *information hiding*.

3.2 Unterstützung in Programmiersprachen

All diese Prinzipien haben die Programmiersprachen, das Grundhandwerkszeug eines jeden Entwicklers, stark beeinflusst. In diesem Kapitel soll darauf eingegangen werden, wie sich die Prinzipien in einigen beispielhaften Programmiersprachkonzepten wiederfinden und bei welchen Konzepten heutige Programmiersprachen noch Defizite haben. Hierbei versuchen die Sprachentwickler eine Balance zu finden zwischen Ausdrucksmächtigkeit und Benutzungsfreundlichkeit.

3.2.1 Was Programmiersprachen unterstützen

Nahezu jede Programmiersprache unterstützt Konzepte, die das Zerlegen von Systemen in leichter handhabbare Teile erlauben. Seien es abstrakte Datentypen, Objektorientierung, Programmmodule, Schnittstellen, formale Kontrakte oder andere Konzepte. [3]

Prozedurale Programmiersprachen[3] erlauben es dem Entwickler eine Reihe von Arbeitsschritten in *Unterprogramme*[4] auszulagern, welche eine Eingabe in die gewünschte Ausgabe transformieren und diese in unterschiedlicher Weise zurückliefern. Jedes Unterprogramm besitzt hierfür eine Schnittstelle über welche sich benötigte Eingabedaten, sowie das Ergebnis ihrer Berechnung, festlegen lassen. Die Schnittstellen sind eine schwache Form eines Kontrakts.

Der Zustand eines Programms liegt in Form von veränderlichen Variablen offen vor, die nur durch ihren Bereich (engl. *scope*) versteckt sind. Es gibt jedoch auch globale Variablen, die gegen das Prinzip des *information hiding* verstoßen und einige Probleme mit sich bringen [9]. Prozedurale Programmie-

[3] Der Begriff 'prozedurale Programmierung' ist mehrdeutig und kann sowohl den Ansatz des prozeduralen Paradigmas meinen (das Aufteilen eines Algorithmus in überschaubare Teile), als auch die Verwendung von Prozeduren.

[4] Mit Unterprogramm ist eine Prozedur, Funktion, Methode oder Routine gemeint. Es gibt feine Detailunterschiede zwischen diesen Konzepten, das für die Aussage wesentliche Prinzip ist jedoch bei allen gleich: Unterprogramme erlauben es ein Programm zu unterteilen, es lassen sich Parameter übergeben und Ergebnisse zurückliefern.

rung stellt nur ein schwaches Konzept zur Modularisierung dar, bei dem Verhalten und Zustand strikt getrennt sind, sich aber gegenseitig beeinflussen.

Objektorientierte Programmiersprachen repräsentieren Verhalten und Zustand über *Objekte*, die Schnittstellen zur Verfügung stellen um auf das Verhalten und den Zustand – welche die Objekte kapseln – zuzugreifen. Zugriffsmodifikatoren wie `private` und `public` für Methoden und Attribute unterstützen das *information hiding* besser als in prozeduralen Sprachen. Vererbung, Komposition und Polymorphie erlauben ebenfalls eine bessere Umsetzung von Konzepten wie *information hiding*, *separation of concerns* und das *DRY* Prinzip.

Ein Problem bei der prozedualren und objektorientierten Programmierung sind die Seiteneffekte, die beim Aufruf eines Unterprogrammes entstehen können. Seiteneffekte machen die Programme schwer nachvollziehbar, da sie nicht offensichtlich erkennbar sind über die Schnittstellen.

Funktionale Programmiersprachen bieten für das Problem der Seiteneffekte eine Lösung: Variablen, die Zustände speichern, sind unveränderlich. Verhalten wird in Funktionen gekapselt, welche Schnittstellen bieten. Daten werden in Form von Variablen an eine Funktion übergeben und man erhält neue Variablen – mit transformierten Daten – zurück. Innerhalb der Funktion wird jedoch keine Veränderung am bestehenden Zustand vorgenommen, es werden nur neue Daten berechnet. Ein wesentliches Konzept hiervon ist, dass dieselbe Eingabe immer zur gleichen Ausgabe führt. Dieses Konzept entstammt den Funktionen aus der Mathematik.

3.2.2 Was Programmiersprachen nicht unterstützen

Programmiersprachen unterstützen den Entwickler jedoch nur beim Umsetzen der Modularisierung, nicht beim Konzipieren und Entwerfen einer Modularisierung. Sie können den Entwickler nur auf Sourcecode-Ebene unterstützen, was eine sehr begrenzte Perspektive darstellt.

Die Prinzipien und Kriterien der Modularisierung beziehen sich aber auf weitere Teile eines Entwicklungsprozesses als nur die Implementation, wie den Systementwurf und die Wartung. Sie spielen eine viel größere Rolle beim Systementwurf. Wenn ein Softwarearchitekt ein System in Module zerlegt, steckt eine gewisse Absicht hinter jeder dieser Entscheidungen. Diese Absicht lässt sich über Programmiersprachen weder automatisch absichern, noch ausdrücken.

Aufgrund dieser Tatsache geschieht es oft, dass eine Modularisierung mit der Zeit "verwässert", sei es über den Entwicklungsprozess oder später während der Wartung oder Erweiterung der Software. Regelmäßige Code Reviews durch die Architekten verhindern dies, sind jedoch zeitaufwändig und kostenintensiv. [3]

3.3 Automatic Modularity Conformance Checking

Ein gut modularisierter Entwurf garantiert nicht immer eine gut modularisierte Implementation. Ob eine Modularisierung vom Systementwurf gut umgesetzt wurde in der Implementation muss manuell überprüft werden, was schwierig ist.

Es gibt Ansätze, die Modularisierung von der Entwurfsebene auf die Implementationsebene abzubilden und diese Abbildung auf ihre Übereinstimmung zu überprüfen. Einer dieser Ansätze ist das *Automatic Modularity Conformance Checking*. [10]

Wie Huynh et al. in ihrem Artikel [10] zeigen ist es damit möglich den Entkoppelungseffekt von Entwurfsentscheidungen auf die Implementation aufzuzeigen, Modularisierungsabweichungen der Implementation zu entdecken und implizite Abhängigkeiten zwischen Modulen mit Hilfe von *design structure matrices* explizit sichtbar zu machen.

4 Probleme

Es gibt einige Probleme die beim Modularisieren oder nach der Modularisierung auftreten können. In diesem Kapitel soll ein kleiner Einblick in mögliche Probleme, sowie aktuelle Lösungsansätze gegeben werden.

4.1 Cross-cutting Concerns

Ein *cross-cutting concern* ist ein Anliegen, das andere Anliegen beeinflusst und sich nicht sauber von anderen Anliegen trennen lässt. Dies kann sich entweder durch duplizierten Code oder durch starke Verbindungen zwischen Modulen bemerkbar machen. Während manche Probleme sich sauber in Module zerlegen lassen, verteilen sich *cross-cutting concerns* über die Module und verstricken diese miteinander.

Ein Beispiel für ein *cross-cutting concern* ist Logging, was sich nicht sauber vom Applikationscode trennen lässt. Logging muss zwangsweise in den Modulen und Funktionen, die geloggt werden sollen, gemacht werden, obwohl das Logging selbst mit den Funktionen und Modulen wenig zu tun hat. Dadurch verstrickt sich das Logging-Anliegen mit diesen Modulen. Man kann zwar die Logging-Funktionen in ein eigenes Modul kapseln, die Aufrufe an dieses Modul müssen aber dennoch über die ganze Anwendung verteilt werden.

4.1.1 Aspektorientierte Programmierung

Eine Lösung für die *cross-cutting concerns* findet sich in der aspektorientierten Programmierung (kurz: AOP). Die Modularisierung von *cross-cutting concerns* durch sogenannte Aspekte ist das explizite Ziel von AOP.

AOP führt dafür drei neue Konzepte ein: Advice, point-cuts und aspects. Bei *advice* handelt es sich um den zusätzlichen Code, mit dem die ursprüngliche Anwendung erweitert werden soll. Im Falle von einer Anwendung, die um Logging erweitert werden soll, wäre dies der Logging-Code. Dieser zusätzliche Code wird durch *point-cuts* dynamisch eingebunden und an der jeweils notwendigen Stelle im ursprünglichen Programm aufgerufen, ohne dieses zu verändern. Zusammen stellen Advices und Point-cuts die *aspects* dar. [11]

Somit lassen sich modulübergreifende Anliegen vom eigentlichen Applikationscode trennen und in den Aspekten selbst kapseln, mit allen Vorteilen.

Die aspektorientierte Programmierung bringt jedoch seine eigenen Herausforderungen mit sich: Welche Anliegen sich als Aspekte separieren lassen und die Vorteile dieser Separierung abzuschätzen ist keine leichte Aufgabe. Darüberhinaus erschweren viele Aspekte die Nachvollziehbarkeit eines Systems. [3]

Hinzu kommt, um beim Beispiel des Logging-Moduls zu bleiben, dass jetzt zwar nicht mehr viele Module auf das Logging-Modul referenzieren, umgekehrt muss der Logging-Aspekt nun aber alle Module kennen, in denen geloggt werden soll, inklusive dem aktuellen Applikationskontext um sinnvolle Log-Meldungen produzieren zu können.

4.2 The Tyranny of the Dominant Decomposition

Bei der *tyranny of the dominant decomposition* handelt es sich um eine Variante der *cross-cutting concerns*: Ein Programm kann nur auf eine Weise modularisiert werden (die "dominante" Dekomposition), aber es gibt Anliegen, die sich bei dieser Modularisierung über viele Module verteilen. Der Unterschied zu *cross-cutting concerns* ist, dass sich diese Anliegen durch eine andere Weise der Modularisierung sauber trennen lassen würden. Dies würde jedoch dazu führen, dass sich andere Anliegen über die Module verteilen. [12] [13]

Beim "normalen" Modularisieren teilt man das System auf eine bestimmte Art nach einem bestimmten Anliegen auf, sprich in einer Dimension – der Dimension der dominanten Dekomposition. Die günstigste Art – oder Dimension – der Dekomposition ist abhängig von vielen Dingen, wie die Phase des Softwareentwicklungsprozesses (in der sich das Projekt gerade befindet), die aktuelle Rolle des Entwicklers oder die aktuelle Aktivität des Entwicklers. Es gibt keine einzelne Dimension die allen Situationen gerecht wird. [14]

4.2.1 Multi-Dimensional Separation of Concerns

Multi-dimensional separation of concerns ist ein Ansatz, der es zum Ziel hat, ein System gleichzeitig auf mehrere verschiedene Arten von Anliegen aufteilbar zu machen, ohne das System massiv refaktorieren zu müssen. Diese Anliegen können sich überlappen und miteinander interagieren. Es wird

dabei von *on-demand remodularization* gesprochen, die es dem Entwickler erlauben soll, zu einer gegebenen Zeit – ohne großen Aufwand – die günstigste Art der Modularisierung auswählen zu können. [14]

Ein weiterer wichtiger Aspekt ist es, die Anliegen inkrementell erweitern zu können. Dies bedeutet, dass nicht alle Anliegen zu Beginn des Systementwurfs bekannt sein müssen, sondern während der Entwicklung neue Anliegen hinzukommen können, ohne dass das System massiv refaktoriert werden muss. Auch darf kein Anliegen wichtiger sein als die anderen, sprich es darf kein Anliegen die anderen dominieren.

Es gibt verschiedene Ansätze wie diese Ziele erfüllt werden können, wie z.B. *hyperspaces* – ein Forschungsprojekt der *IBM Research* Abteilung. [14]

5 Zusammenfassung

Die Prinzipien der Modularisierung sind ein zentrales Thema der Softwareentwicklung. Modularisierung selbst ist eine einfache Idee, die sich selbst über 40 Jahre nach Parnas' Artikel [1] noch in vielen Prinzipien, Ratschlägen, Programmiersprachen oder Methoden im Kern wiederfindet. Wie man am Beispiel der Programmiersprachen erkennen kann, werden die Möglichkeiten der Unterstützung von Modularität durch Technologien immer ausgereifter. Nach Parnas' Artikel kamen ebenso noch eine Vielzahl an Prinzipien, Kriterien und Methoden hinzu, nach denen sich Systeme in Module zerlegen lassen.

Doch es gibt auch noch einige Probleme, wie die *tyranny of the dominant decomposition*, für die noch praxistaugliche Lösungen gefunden werden müssen die sich etablieren.

Literatur

[1] Parnas, D. L.: On the Criteria To Be Used in Decomposing Systems into Modules. In: *Communications of the ACM* 15 (1972), Dezember, Nr. 12

[2] Fowler, Martin: The State of Design. In: *IEEE* 22 (2005), November, Nr. 5

[3] Hoek, André van der ; Lopez, Nicolas: A Design Perspective on Modularity. In: *AOSD* (2011), März

[4] Stevens, W. P. ; Myers, G. J. ; Constantine, L. L.: Structured Design. In: *IBM Systems Journal* 13 (1974), Nr. 2

[5] Beck, Fabian ; Diehl, Stephan: On the Congruence of Modularity and Code Coupling. In: *ACM* (2011), September

[6] Hamza, Haitham S.: Separation of Concerns for Evolving Systems: A Stability-Driven Approach. In: *ACM* (2005), Mai

[7] Wikipedia: *Permutiertes Register*. – `http://de.wikipedia.org/wiki/KWIC`, Stand: 22. Juni 2012

[8] Gamma, Erich ; Helm, Richard ; Johnson, Ralph E. ; Vlissides, John: *Design Patterns: Elements of Reusable Object-Oriented Software*. 1st. Addison-Wesley Longman, 1994

[9] Wulf, W. ; Shaw, Mary: Global Variable Considered Harmful. In: *ACM* 8 (1973), Februar, Nr. 2

[10] Huynh, Sunny ; Cai, Yuanfang: Automatic Modularity Conformance Checking. In: *ACM* 10 (2008), Mai, Nr. 18

[11] OnJava: *Overview of Aspect-oriented Programming*. – `http://onjava.com/onjava/2004/01/14/aop.html`, Stand: 15. Juni 2012

[12] Center, IBM T.J. Watson R.: *Morphogenic Software*. – `http://www.research.ibm.com/morphogenic/`, Stand: 15. Juni 2012

[13] D'Hondt, Maja ; D'Hondt, Theo: *The Tyranny of the Dominant Model Decomposition*. September 2002. – `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.9.5189`, Stand: 15. Juni 2012

[14] Ossher, Harold ; Tarr, Peri: Multi-Dimensional Separation of Concerns and The Hyperspace Approach. In: *IBM T.J. Watson Research Center*