Entwicklung modulithischer Softwaresysteme mithilfe von Domain Driven Design

Masterarbeit zur Erlangung des akademischen Grades
Master of Science
im Studiengang Computer Science - Software Engineering
an der Fakultät für Informatik und Ingenieurwissenschaften
der Technischen Hochschule Köln

vorgelegt von: Marco Reitano
Matrikel-Nr.: [redacted]
Adresse: [redacted]
marcoreitano85 (at) gmail.com

eingereicht bei: Prof. Dr. Stefan Bente
Zweitgutachter: Prof. Dr. Christian Kohls

Gummersbach, 09.01.2024

Technology
Arts Sciences
TH Köln

Entwicklung modulithischer Softwaresysteme mithilfe von Domain Driven Design

Abstrakt

Die vorliegende Masterarbeit leitet die Definition von Modulen, zugehörige Konzepte der Kohäsion und Kopplung nach Constantine & Yourdon und Myers und das Konzept der Connascence nach Page-Jones aus den Originalquellen her. Ausgewählte Entwurfsmuster und modulbildende Strukturen der Objektorientierten-Programmierung nach Kay und des Domain Driven Design nach Evans und Vernon werden auf ihre Auswirkungen auf die Kohäsion und Kopplung von Modulen untersucht. Die Arbeit beweist weiterhin die Wirksamkeit des Modulith-Architekturstils im Allgemeinen und des Spring Modulith Framework im Speziellen gewünschte Moduleigenschaften zu erzeugen, mithilfe von architektonischen Fitnessfunktionen überprüfbar zu machen und so eine Erosion von architektonischen Softwarequalitäten zu verhindern.

Development of modulithic software systems using Domain Driven Design

Abstract

This Master's thesis derives the definition of modules, related concepts of cohesion and coupling according to Constantine & Yourdon and Myers, and the concept of Connascence by Page-Jones from original sources. Selected design patterns and modular structures of Object-Oriented Programming according to Kay, and Domain Driven Design according to Evans and Vernon, are examined for their effects on the cohesion and coupling of modules. The work further proves the effectiveness of the modulith architectural style in general and the Spring Modulith framework in particular in generating desired module properties, making them verifiable through architectural fitness functions, and thus preventing the erosion of architectural software qualities.

Inhaltsverzeichnis

1 Einleitung

Seit seiner Einführung Mitte der 2010er Jahre hat sich der **Microservice-Architekturstil** zunehmend zur präferierten Architekturlösung für hinreichend komplexe, webbasierte Softwaresysteme entwickelt. Einer Umfrage aus dem Jahr 2021 zufolge nutzen bereits 75% der Unternehmen mit einer Belegschaft von 1000 bis 2999 Mitarbeitern diesen Architekturstil, während weitere 23% den Einsatz in der Zukunft planen. Betrachtet man größere Organisationen mit mindestens 5000 Angestellten, steigt der derzeitige Nutzungsgrad von Microservices sogar auf 85%, vgl. [SoftwareAG, 2021, S. 17].

Das Wesen der Microservice-Architektur liegt dabei in der Dekomposition eines Softwaresystems in kleine, lose gekoppelte Dienste, die unabhängig voneinander entwickelt und betrieben werden können.

Dieser Architekturstil verspricht hierbei nach [Newman, 2015, S.4ff] unter anderem die folgenden Vorteile:

- **Organizational Alignment**
  Die einzelnen Microservices können jeweils von kleinen, agilen Entwicklungsteams implementiert und betrieben werden. Der Quellcode der einzelnen Microservices ist dabei (vermeintlich) hochgradig wart-, änder- und erweiterbar, wodurch eine weitgehend autonome Arbeitsweise der Entwicklungsteams begünstigt wird und nötige Absprachen auf ein Minimum reduziert werden.
- **Technology Heterogeneity**
  Die Kommunikation zwischen Microservices ist auf die Nutzung von REST (Representational State Transfer) und Messagebrokern begrenzt und sorgt so für eine (vermeint-

lich) lose, technische Kopplung der Services. Die technische Entkopplung ermöglicht es den Entwicklungsteams ihren Technologiestack je Microservice frei zu wählen und individuellen, problemorientierten Bedürfnissen anzupassen.

- **Resilience**
  Die Resilienz des Gesamtsystems ist erhöht, da mögliche Ausfälle auf einzelne (vermeintlich) entkoppelte Microservices begrenzt bleiben und die Funktionalität des Gesamtsystems so zu großen Teilen erhalten bleibt.
- **Scalability**
  Das System kann auf lokale Lastspitzen durch horizontale Skalierung der betroffenen Services reagieren. Einzelne Microservices werden repliziert und die Last unter ihnen mithilfe eines Loadbalancers aufgeteilt werden.

Allerdings ergeben sich aus dieser Architektur auch gewisse Nachteile, insbesondere im Hinblick auf die gesteigerte technische Komplexität des Gesamtsystems:

- Der verteilte Betrieb der Microservices macht **Netzwerkeffekte** (Latenz, Bandbreite, Netzwerkausfall, etc.) relevant, die in Architektur und Implementierung der Microservices berücksichtigt werden müssen.
- Es entsteht ein **technisch komplexerer Kommunikationsbedarf**, welcher Service- und Systemgrenzen kreuzt und so die Einführung neuer Technologien in Form von Messagebrokern, Service-Discoveries, Loadbalancer und Ähnlichem nötig macht.
- Das Deployment einer Vielzahl an Services und zugehörigen, heterogenen Technologiestacks stellt **gesteigerte Anforderungen an die Betriebsumgebung des Gesamtsystems** und macht die Nutzung von Containerisierungstechnologien wie Docker oder Kubernetes unumgänglich.

Der resultierende zusätzliche technische Aufwand ist insbesondere für kleinere Organisationen oft kaum zu bewältigen. Die weite Verbreitung des Microservice-Architekturstil steht deshalb zunehmend in der Kritik, da Herausforderungen und Nachteile von Microservices oft übersehen werden und/oder die versprochenen Vorteile eingefordert werden ohne, dass sie wirklich benötigt werden. Der Microservice-Architekturstil wird so zum De-Facto-Standard erklärt und oft als *Silverbullet*-Lösung für jegliche Art von Software angepriesen.

Selbst der Softwareconsultant Thoughtworks[1], dessen Mitarbeiter*innen maßgeblich an der Entwicklung des Microservice-Architekturstils beteiligt waren und sind, warnt mittlerweile vor einer zu vorschnellen Entscheidung zur Verwendung von Microservices, vgl. [Thoughtworks, 2018]. Sie mahnen dabei an, dass sorgfältig abgewogen werden sollte, ob die Vorteile des Architekturstils wirklich benötigt werden und warnen vor den zusätzlichen Kosten, die durch oben beschriebene Nachteile entstehen können.

Der Gegenentwurf zum Microservice-Architekturstil findet sich in der traditionellen **monolithischen Softwarearchitektur**, in der eine Anwendung als integriertes, einheitliches System konzipiert, entwickelt und betrieben wird. Alle Komponenten einer monolithischen Software werden dabei in einer einzigen Codebasis entwickelt und später gemeinsam in Form eines einzigen Prozesses betrieben.

Vorteile liegen dabei insbesondere in einer geringeren technischen Komplexität der Kommunikation, da Interaktionen zwischen einzelnen Komponenten auf einen Prozess begrenzt sind. Physische System- und Servicegrenzen müssen bei einer Monolith-Architektur nicht überwunden werden und die Kommunikation beschränkt sich auf einfache Methodenaufrufe oder das Verschicken prozessinterner Nachrichten und Ereignisse.

Auffällig ist, dass in der Diskussion über das Für und Wider der beiden obigen Architekturstile der Begriff des Monolithen oft missverständlich verwendet wird. Der Begriff des Monolithen kann dabei nach [Dowalil, 2019] entweder als **Architektur-** oder als **Deployment-Monolith** verstanden werden:

Der Begriff des **Architektur-Monolithen** beschreibt hierbei primär den internen Aufbau einer monolithischen Anwendung und kritisiert dabei eine unzureichende interne Strukturierung der Codebasis. Die mangelnde interne Struktur spiegelt sich hierbei oft in unklaren Klassen-, Komponenten- oder Paketgrenzen und einer starken Kopplung der gesamten Codebasis wider. Im Extremfall spricht man auch von einer *Big Ball Of Mud*-Architektur, bei der Grenzen gänzlich verschwinden und die gesamte Codebasis als untrennbare Einheit angesehen werden kann.

[1] https://www.thoughtworks.com/

Aufgrund der unzureichenden, internen Strukturierung wird dem **Architektur-Monolithen** meist eine schwache Wart-, Änder- und Erweiterbarkeit zugesprochen. Bei Verwendung des Begriffes Monolith im Sinne eines Architektur-Monolithen wird also eine wertende Aussage über dessen inhärente Softwarequalitäten gemacht.

Der Begriff **Deployment-Monolith** beschreibt hingegen einzig die Form der Bereitstellung und des Betriebes einer monolithischen Anwendung. Die gesamte Anwendung wird dabei als ein einzelnes, zusammenhängendes Artefakt (bspw. in Form einer Jar-Datei) bereitgestellt und wird in Form eines einzelnen Prozesses betrieben. Eine wertende Aussage über die interne Struktur und über daraus folgende Softwarequalitäten wird mit dem Begriff **Deployment-Monolith** hingegen nicht getroffen.

Deployment-Monolithen können also durchaus eine adäquate interne Struktur in Form von Komponenten oder Modulen besitzen, die die Wart-, Änder- und Erweiterbarkeit des Quellcodes fördert und einen kostengünstigen Entwicklungsprozess möglich macht.

Der beschriebene Paradigmenwechsel weg von monolithischen Softwaresystemen hin zu einem Microservice-Architekturstil in den vergangenen Jahren bleibt deshalb nicht ohne Kritik. Kritiker behaupten, dass insbesondere der Vorteil des Organisational Alignment einer Microservice-Architektur auch durch eine intern gut strukturierte Deployment-Monolithen-Architektur erreicht werden kann. Sie führen an, dass klar abgegrenzte und lose gekoppelte Komponenten innerhalb eines Monolithen genauso gut kleinen, agilen Entwicklungsteams zugeordnet und durch diese unabhängig entwickelt werden können. Problematisch ist hierbei allerdings, dass strukturelle, architektonische Vorgaben und Eigenschaften nicht durch programmiersprachliche Mittel oder übliche funktionale Tests überprüft werden können. Es besteht deshalb die Gefahr, dass strukturelle Vorgaben und Eigenschaften der Software im Laufe des Entwicklungsprozess erodieren und so Modulgrenzen geschwächt werden und die Kopplung zwischen Komponenten steigt. Obwohl anders intendiert, kann so die Erreichung hoher Softwarequalitäten im Bereich der Wart-, Änder- und Erweiterbarkeit nicht dauerhaft sichergestellt werden.

Ein weiteres Missverständnis liegt in der Annahme, dass die Nutzung des Microservice-Architekturstils

automatisch zur losen Kopplung der Dienste führt. Die Erfahrung zeigt, dass der Microservice-Architekturstil die lose Kopplung der einzelnen Dienste zu einem gewissen Maß begünstigt, indem sie technologische, physische Grenzen zwischen den Services zieht und Entwickler*innen gezwungen sind diese zu überwinden. Betrachtet man aber Beispiele in der Realität finden sich zunehmend Microservice-Architekturen in denen der Kommunikationsaufwand mit zunehmender Zahl an Services immer komplexer und größer wird und darunter die Autonomie der einzelnen Services leidet. Es kommt zur Entwicklung eines sogenannten *verteilten Big Ball of Muds*, der sowohl die versprochenen Vorteile des Microservice-Architekturstils zunichtemacht als auch die angesprochenen Nachteile in Kauf nimmt, vgl. [Brown, 2014]. Ein *verteilter Big Ball of Mud* ähnelt so in den Softwarequalitäten der Wart-, Änder- und Erweiterbarkeit eher einem Architektur-Monolithen.

Diese Beobachtung stellt die lose Kopplung der Services infrage, denn eine starke Entkopplung der Services sollte intuitiv eher zur Verringerung des Kommunikationsbedarfs beitragen. Es ist deshalb zu vermuten, dass die rein technische, zuweilen künstliche, Einführung von Servicegrenzen nicht ausreicht, um Services gänzlich zu entkoppeln.

Ein tieferes konzeptionelles Verständnis des Kopplungsbegriffs, das zur Lösung dieser Problematik beitragen könnte, scheint oft zu wenig beachtet oder fehlt komplett.

Der neue Ansatz des **Modulith-Architekturstils** kann als Reaktion auf die oben beschriebenen Missverständnisse und Probleme angesehen werden. Der Modulith-Architekturstil legt dabei, wie der Name schon sagt, einen stärkeren Fokus auf das Konzept der Modularisierung und die verwandten Begriffe der Kopplung und Kohäsion. Der Architekturstil strebt so, ähnlich wie die Microservice-Architektur, eine Stärkung der Softwarequalitäten der Wart-, Änder- und Erweiterbarkeit durch Dekomposition der Software an.

Die modulithische Architektur verzichtet allerdings gleichzeitig auf einen dezentralen Betrieb der Software und ähnelt in dieser Hinsicht eher einem Deployment-Monolithen. Sie vermeidet so den oben beschriebenen zusätzlichen technischen Aufwand des Microservice-Architekturstils. Die modulithische Architektur wird deshalb auch oft als hybride Architektur bezeichnet, die versucht Vorteile von Microservices und Monolithen zu vereinen, vgl.

[Junker, 2020].

Das in diesem Jahr veröffentlichte Spring Modulith Framework unterstützt die Umsetzung eines Modulith-Architekturstils im Spring Framework. Es verfolgt dabei den Ansatz des Domain Driven Design (DDD) und nutzt insbesondere dessen Aggregate Konzept als Ausgangspunkt zur Modularisierung der Domäne und Software. Zur eigentlichen Dekomposition der Quellcodes in Module nutzt Spring Modulith die Java Paketstruktur. Modulgrenzen werden so im Quellcode deutlich sichtbar gemacht und architektonische Moduleigenschaften können in Form von architektonischen Tests überprüft werden. Spring Modulith versucht so eine Erosion der strukturellen Eigenschaften der Software zu verhindern und eine erhoffte hohe Softwarequalität nachhaltig sicherzustellen.

1.1 Ziel und Vorgehen

Eine anfängliche Literaturrecherche zur Modulith-Architektur hat schnell gezeigt, dass dem Konzept der Modularisierung sowie den zugehörigen Metriken der Kohäsion und Kopplung in modernen Literaturquellen aus dem Bereich der Softwarearchitektur eine zentrale Rolle zugesprochen wird. Eine detaillierte, vornehmlich konzeptionelle Definition und Herleitung der Begriffe sind in moderner Literatur aber kaum zu finden. Meist beschränken sich moderne Quellen auf eine eher technische Sichtweise auf Module und definieren den Begriff der Kopplung als rein technisch begründete Abhängigkeiten.

Ziel der vorliegenden Arbeit ist daher, zunächst die grundlegenden Konzepte der Modularisierung, Kopplung und Kohäsion aufzuarbeiten, historisch herzuleiten und konzeptuell zu definieren. Hauptaugenmerk wird hierbei auf Originalquellen gelegt, in denen entsprechende Begriffe und Konzepte erstmalig eingeführt wurden.

Darauf aufbauend wird die Anwendung und Umsetzung des Modulkonzepts in der Objektorientierten Programmierung (OOP) und dem Entwicklungsansatz des Domain Driven Design (DDD) betrachtet. Es wird untersucht, inwieweit Moduleigenschaften durch die Nutzung der

beiden Ansätze gestärkt werden und ob/wie deren Einhaltung nachhaltig sichergestellt werden kann.

In einem letzten Schritt erfolgt eine detaillierte Analyse des Modulith-Architekturstils am Beispiel des Spring Modulith Frameworks. Ziel ist er hierbei zunächst zu verstehen welche Konzepte aus OOP und DDD zur Modularisierung genutzt werden und ob/wie das Framework eine Erosion der strukturellen Moduleigenschaften nachhaltig verhindern kann.

1.2 Forschungsfragen

Es ergeben sich folgende Forschungsfragen, die im Rahmen dieser Arbeit beantwortet werden:

- Welche Arten von Abhängigkeiten gibt es und wie entstehen sie?
- Welchen Einfluss haben die verschiedenen Abhängigkeitsarten auf die Kohäsion innerhalb und den Kopplungsgrad zwischen Modulen?
- Welche Pattern und Vorgehensweisen finden sich im Domain Driven Design, die Modulkohäsion stärken und den Kopplungsgrad der Module im Gesamtsystem mindern?
- Welche technischen Lösungen bietet der Modulith-Architekturstil und das Spring Modulith Framework, die diese Pattern umsetzen und wie wirksam sind sie?

1.3 Aufbau der Arbeit

Der erste Teil der vorliegenden Arbeit fasst in **Kapitel 2** die Ergebnisse der Literaturrecherche zusammen und beschreibt grundlegende Konzepte. Sie beginnt mit der Definition von Modulen aus dem Bereich des Structured Design und bespricht zugehörige Themen der Kohäsion und Kopplung. Der darauf folgende Abschnitt untersucht das Designprinzip *High Cohesion & Low Coupling*, stellt das Modularisierungsvorgehen vor und bespricht die Auswirkungen auf die Softwarequalitäten der Wart-, Änder- und Erweiterbarkeit.

Kapitel 3 überführt die erarbeiteten Grundlagen in die Welt der objektorientierten Programmierung und untersucht, inwieweit die Nutzung von Objekten und Klassen eine Form der Modularisierung widerspiegelt. Das Kapitel führt des Weiteren das Konzept der Connascence zwischen Klassen ein und bespricht Ähnlichkeiten und Unterschiede zu den Konzepten der Kohäsion und Kopplung.

Kapitel 4 gibt zunächst einen Überblick über den Entwicklungsansatz des *Domain Driven Design* und beschreibt dessen Zielsetzung. Das Kapitel stellt dann zunächst den Bereich des *strategischen Design* vor und untersucht, inwieweit die Aufteilung der Domäne in Bounded Context als eine Form der Modularisierung angesehen werden kann. Weiterhin wird der Bereich des *taktischen Design* und dessen grundlegende *"Building Blocks"* Value Object, Entity und Aggregate vorgestellt. Ausgewählte Desingpattern zur Erstellung von Aggregaten werden im Detail betrachtet und auf modulbildende Eigenschaften untersucht.

Kapitel 5 bespricht zunächst den Architekturstil des Modulithen und stellt verschiedene Frameworks und Möglichkeiten der Umsetzung einer solchen Architektur vor. Das Framework Spring Modulith wird daraufhin im Detail besprochen.

Kapitel 6 fasst zunächst die Ergebnisse der vorliegenden Arbeit zusammen und beantwortet dann die in Abschnitt 1.2 aufgestellten Forschungsfragen abschließend. Im Anschluss werden die Ergebnisse diskutiert. Die Arbeit schließt mit einem Ausblick auf mögliche zukünftige Arbeiten im Gebiet der Modularisierung im Allgemeinen und der Modulith-Architektur im Speziellen.

2 Grundlagen der Modularisierung

Die Literaturrecherche zum Thema der Modularisierung und den zugehörigen Begriffen der Köhasion und Kopplung hat ergeben, dass es hierbei um eines der frühesten und fundamentalsten Konzepte der Informatik handelt. So sagt Robert C. Martin in [Martin, 2018, S. 28f], dass die ersten Ansätze zur Definition des Modulbegriffs bereits bei Edsger W. Dijstra in den 1960er Jahren zu suchen sind. Dijkstra argumentierte in seinem kontroversen Paper „Go to Statement Considered Harmful“ aus dem Jahr 1968, dass Sprunganweisungen und *go-to*-Befehle in damals vorherrschenden, assemblernahen Programmiersprachen einen äußerst komplexen Kontrollfluss erzeugen, vgl. [Dijkstra, 1968]. Jeder Sprungbefehl übergibt dabei den Kontrollfluss an die verschiedensten Stellen des gesamten Softwaresystems. Das nötige Wissen von Entwickler*innen ist deshalb bei Änderungen am Programmcode nicht auf die aktuell zu ändernde, lokale Stelle im Quellcode begrenzt, sondern erstreckt sich potenziell auf die Gesamtheit des Quellcodes im Softwaresystem.

Aus diesem Grund schlägt Dijkstra den Ansatz des **Structured Programming** vor und fordert die rekursive Dekomposition von Programmcode in kleinere überprüfbare Einheiten (*engl. provable units*). Die Korrektheit dieser kleineren Einheiten soll unabhängig vom restlichen Programmcode *überprüfbar* (heute eher *testbar*) sein und begrenzt so automatisch das nötige Wissen der Entwickler*innen auf den Umfang der aktuell zu bearbeitenden Einheit. Die kleinen, unabhängig voneinander überprüfbaren Einheiten von Programmcode nennt Dijkstra auch **Module**.

2.1 Structured Design

> *„For most of the computer systems ever developed, the structure was not methodically laid out in advance - it just happened. The total collection of pieces and their interfaces with each other typically have not been planned systematically.“*
>
> [Yourdon and Constantine, 1979, S.1]

Structured Design, aus dem Jahre 1979, vorgeschlagen von Edward Yourdon und Larry L. Constantine ist einer der ersten konkreten Versuche Softwarestruktur durch die Einführung von Modulen und der Dekomposition von Quellcode zu erzeugen. Sie argumentieren ähnlich wie Dijkstra, dass unstrukturierter Quellcode nur wenig wart-, veränder- und erweiterbar ist. Sie führen hierzu an, dass Entwickler*innen von unstrukturiertem Quellcode bei jeder Veränderung eine Vielzahl von möglichen Seiteneffekten in den verschiedensten Teilen des Softwaresystems beachten müssen. Die Entwickler*innen benötigen also oft Wissen über Quellcodeabschnitte, die semantisch weit entfernt von ihrem gegenwärtigen Arbeitsbereich liegen und deren Entwicklung eventuell nicht unter ihrer Kontrolle liegt. Dieses Wissen ist nötig um Folgefehler der jeweiligen Änderung oder Erweiterung zu vermeiden, vgl. [Yourdon and Constantine, 1979, S.67ff].

Das nötige Wissen definieren die Autoren von Structured Design als Abhängigkeiten (englisch: dependencies) der Entwickler*innen und, in deren Vertretung, ihres Quellcodes von anderen Softwareteilen des Systems. Sie argumentieren des Weiteren, dass die Entwicklungsarbeit der Entwickler*innen mit steigender Zahl und Komplexität dieser Abhängigkeiten immer fehleranfälliger wird. Frühe Forschungen im Bereich der Kognition haben gezeigt, dass Menschen nur 7+-2 Informationseinheiten (english: chunks) gleichzeitig im Kurzzeitgedächtnis gegenwärtig haben und beachten können (die sogenannte Millersche Zahl nach [Miller, 1956]). Jede Abhängigkeit des Quellcodes ist je nach ihrer Art als mindestens eine solche Informationseinheit einzuordnen. Wenn die Anzahl dieser Informationseinheiten die Millersche Zahl übersteigt, übersehen Entwickler*innen kritische Informationen und die Fehleranfälligkeit steigt rasant. Infolgedessen sinkt die Effektivität und Effizienz der Entwicklungsarbeit und Wart-, Veränder- und Erweiterbarkeit der Software wird gemindert.

Um die Anzahl und Komplexität der Abhängigkeiten zu minimieren schlägt Structured Design, analog zu Dijkstra, die Dekomposition von Software in kleinere Einheiten vor. Anders als Dijkstra konzentrieren sich Yourdon und Constantine dabei aber zunächst nicht auf den Software-Quellcode, sondern fordern als ersten Schritt die Anwendung der *Divide & Conquer*-Strategie auf das zu lösende Problem an sich. Sie argumentieren, dass Abhängigkeiten in erster Linie Aspekte des Problemraums sind und durch diesen bedingt werden. Sie schlagen deshalb die Dekomposition des Problems in kleinere Teilprobleme vor, die folgende Kriterien, erfüllen sollen:

1. einfach beziehbar zur Applikation (*engl. easibily related to the application*)
2. handhabbar klein
3. separat lösbar
4. separat korrigierbar
5. separat modifizierbar

[Yourdon and Constantine, 1979, S. 18ff]

Die Kriterien 2 bis 5 folgen hierbei dem Vorschlag von Dijkstra und teilen den Problemraum in handhabbare Teilprobleme auf, die unabhängig lös-, korrigier- und modifizierbar sind. Das erste Kriterium geht allerdings über den Grundgedanken des Structured Programming hinaus und fordert, dass Strukturen und Inhalten des Problemraums möglichst deckungsgleich zu den Strukturen und Inhalten der zu erstellenden Software, also dem Lösungsraum, sein sollten.

> *„[. . .]each piece of the system corresponds to exactly one small, well-defined piece of the problem and each relationship between a system's pieces corresponds only to a relationship between pieces of the problem.“*
>
> [Yourdon and Constantine, 1979, S.20]

Die Komplexität des nötigen Wissens von Entwickler*innen soll so weiter verringert werden, da einzelne Teile der Software (Module) sich leichter auf Teile des Problemraums übertragen lassen. Eine zusätzliche mentale Überführung von Strukturen des Problemraums in neue, andere Strukturen im Lösungsraum (und umgekehrt) entfällt und die mentale Belastung von Entwickler*innen und Architekt*innen sinkt.

2.2 Definition von Modulen

Um die Effizienz der Entwicklungsarbeit zu verbessern schlagen Yourdon und Constantine deshalb die Einführung von Modulen als strukturgebende Elemente vor, die das nötige Wissen und Abhängigkeiten minimieren und Seiteneffekte reduzieren sollen.

> *„A module is a lexically contiguous sequence of program statements, bounded by boundary elements, having an aggregate identifier."*
>
> [Yourdon and Constantine, 1979, S.37]

Unter einem Modul versteht [Yourdon and Constantine, 1979, S.37] hierbei, zunächst abstrakt, eine lexikalisch zusammenhängende Sequenz von Statements, die durch begrenzende Elemente aneinander gebunden werden. Das durch die Aggregation der Elemente entstandene Modul muss zusätzlich über einen Aggregateidentifikator benannt und genutzt werden können.

Überführt man die oben genannte abstrakte Moduldefinition ins Konkrete so versteht Structured Design unter einem Modul einen Quellcodeabschnitt, dessen einzelne Statements in einer Prozedur zusammengefasst und je nach Programmiersprache durch Begin- und End-Statements oder geschweiften Klammern begrenzt werden. Der Prozedurname dient nach diesem Verständnis als Aggregateidentifikator, welcher die Prozedur (das Modul) referenzier- und aufrufbar macht, sowie als Abhängigkeit für andere Module bereitstellt.

[Yourdon and Constantine, 1979] versteht unter dem Konzept von Modulen, anders als heute, also noch Prozeduren, da zum Zeitpunkt der Veröffentlichung prozedurale Programmiersprachen, wie COBOL und Fortran, die Softwareentwicklung dominieren und heute relevante objektorientierte Aspekte von Modulen bleiben aus dieser Sichtweise noch unbeachtet.

Die im Folgenden ausgeführten Konzepte bleiben allerdings auch aus einer heutigen objektorientieren Perspektive gültig, da sie sich leicht auf das Verhalten von Objekten (also deren Methodensatz) und sonstige in Objekten gebundene Statements übertragen lassen. Eine aktuellere objektorientierte Sichtweise auf Module und deren Abhängigkeiten folgt in Kapitel 3.3 Connascence.

Um Abhängigkeiten in andere Softwareteile zu minimieren schlagen die Yourdon & Constantine zunächst eine genauere Betrachtung der funktionalen Verwandtschaftsbeziehungen zwischen einzelnen Statements vor und identifizieren insgesamt sieben Beziehungstypen, vgl. [Yourdon and Constantine, 1979, S. 108]:

- Zufällige Assoziation *(coincidental association)*
- Logische Assoziation *(logical association)*
- Temporale Assoziation *(temporal association)*
- Prozedurale Assoziation *(procedural association)*
- Kommunikative Assoziation *(communicational association)*
- Sequentielle Assoziation *(sequential association)*
- Funktionale Assoziation *(functional association)*

Die Stärke der funktionalen Verwandtschaft nimmt, folgend der oben aufgeführten Reihenfolge, jeweils zu. Yourdon & Constantine merken aber an, dass es sich hierbei nicht um einen linearen Anstieg handelt. Eine detailliertere Betrachtung der Stärke der funktionalen Verwandtschaft findet sich am Ende von Abschnitt **2.3 Kohäsion**.

Die hohe funktionale Verwandtschaft zwischen Statements verstärken hierbei den Grad des nötigen Wissens über das Entwickler*innen bei Änderungen an einem Statement verfügen müssen. Bestehen solche starken Verwandtschaftsverhältnisse in die verschiedensten Teile des Quellcodes muss nötiges Wissen erst aufwändig zusammengetragen werden, um mögliche Seiteneffekte bei Änderungen zu vermeiden oder um die Korrektheit eines lokalen Statements zu überprüfen.

2.3 Kohäsion

Die funktionalen Verwandtschaftsverhältnisse können, neben der reinen Beschreibung von Beziehungen zwischen Statements, auch als Heuristik zur Erstellung von Modulen genutzt werden.

Während eine starke funktionale Verwandtschaft zu (im Quellcode) weit entfernten Statements eher unerwünscht sind, macht sie gleichzeitig deutlich, dass die verwandten Statements zur Bereitstellung einer bestimmten Funktionalität benötigt werden. Die Modularisierung nutzt deshalb die funktionalen Verwandtschaftsverhältnisse, aggregiert alle für eine Funktionalität benötigten Statements und bindet sie in einem prozeduralen Modul. Module versammeln so nötiges Wissen und machen es lokal im Modul verfügbar, sodass eine Korrektheitsprüfung der Funktionalität (durch Entwickler*innen oder Tests) ohne den Zugriff auf externes, globales Wissen möglich wird. Die funktionale Verwandtschaft von im Modul gebundenen Statements wird dabei auch *Kohäsion* genannt - sie sorgt für den *inneren Zusammenhalt* eines Moduls.

Die sieben die Kohäsionstypen, die aus den unterschiedlichen funktionalen Verwandtschaftsbeziehungen entstehen, werden im Folgenden im Detail definiert:

- **Zufällige Kohäsion**
  Zufällige Kohäsion tritt auf, wenn zwischen Codeelementen keine oder nur wenig funktionale Verwandtschaft besteht und diese ohne ersichtlichen Grund zusammengefasst werden.

  Es ergibt sich ein zufälliges, praktisch nicht oder nur kaum kohäsives Modul.

  Ein solches Modul kann nach [Yourdon and Constantine, 1979, S.109] beispielsweise entstehen, wenn zufällig mehrfach auftretende Quellcodeabfolgen in einer Prozedur zusammengefasst werden nur um diese Wiederholungen zu vermeiden. Es wird also argumentiert, dass das in der Informatik oft zitierte 'Don't repeat Yourself'-Prinzip (DRY-Prinzip) durchaus ungewollte Abhängigkeiten erzeugen und so architektonischen Schaden anrichten kann.

- **Logische Kohäsion**
  Eine logische Kohäsion besteht, wenn zwei Codeelemente eine ähnliche, aber trotzdem unterschiedliche Funktionalität anbieten.

  Als Beispiel nennt [Yourdon and Constantine, 1979, S.114] verschiedene Codeelemente, die alle logisch zum Input des Human Interface gehören, etwa Elemente, die die

Mauseingabe und die Tastatureingabe umsetzen. Diese setzen zwar eine ähnliche Funktionalität um, sind aber trotzdem verschieden voneinander.

- **Temporale Kohäsion**
  Temporale Kohäsion entsteht bei der Zusammenfassung von Codeelementen, die im gleichen, eng abgesteckten Zeitintervall des Programmablaufs ausgeführt werden.

  Beispielsweise werden verschiedene Initialisierungsabläufe eines Systems in einem Modul zusammengefasst, um eine gemeinsame, zeitlich zusammengefasste Ausführung beim Programmstart zu vereinfachen.

- **Prozedurale Kohäsion**
  Prozedurale Kohäsion zwischen Elementen entsteht nach [Yourdon and Constantine, 1979, S.117ff], wenn sie durch Kontrollstrukturen miteinander verknüpft sind.

  Beispielsweise sind Elemente im Anweisungsblock einer Schleife oder in zwei Pfaden einer Verzweigung prozedural kohäsiv.

  Neuere Quellen definieren die prozedurale Kohäsion rein durch das Vorhandensein einer definierten Ausführungsreihenfolge von Elementen, vgl. [Richards and Ford, 2020, S.41]. Diese Sichtweise ist laut [Yourdon and Constantine, 1979, S. 118] zwar korrekt, aber stellt nur die leichteste Form der prozeduralen Ähnlichkeit dar, da sie sich kaum von der temporalen Kohäsion unterscheidet.

- **Kommunikative Kohäsion**
  Kommunikative Kohäsion besteht zwischen Codeelementen, die denselben Eingangsdatensatz verarbeiten oder zur Erzeugung desselben Ausgangsdatensatzes beitragen.

  Laut [Yourdon and Constantine, 1979, S.121] handelt es sich bei der kommunikativen Kohäsion um die erste Kohäsionsart in dieser Aufzählung, die sich inhärent aus dem Problemraum selbst ergibt und nicht rein durch technische Entscheidungen der Entwickler*in bzw. der Architekt*in im Lösungsraum entsteht.

- **Sequentielle Kohäsion**
  Die sequentielle Kohäsion besteht zwischen zwei Codeelementen bei denen die Ergebnisdaten des ersten Elements als Eingangsdaten des zweiten Elements verwendet werden.

- **Funktionale Kohäsion**
  Die funktionale Kohäsion beschreibt die höchste Form der Kohäsion von Modulen. Sie lässt sich allerdings nicht allein durch die funktionale Verwandtschaft der gebundenen Elemente konkret definieren.

  [Yourdon and Constantine, 1979, S.127] bietet stattdessen, die folgende Definition an:

  > *„In a completely functional module, every element of processing is an integral part of, and is essential to, the performance of a single function."*

  Und weiter:

  > *„Functional cohesion is whatever is not sequential, communicational, procedural, temporal, logical, or coincidental."*

  Die funktionale Kohäsion definiert also zum einen die Mindestgröße eines Moduls und zum anderen aber auch die Maximalgröße eines Moduls, da laut Yourdon & Constantine nur die Elemente gebunden werden dürfen, die für Umsetzung *einer* Funktionalität benötigt werden.

  Hiermit erinnert die funktionale Kohäsion stark an das Single-Responsibility-Principle (SRP, deutsch Prinzip der eindeutigen Verantwortlichkeit) nach [Martin, 2008, S.138], das besagt, dass es nie mehr als einen Grund geben sollte, ein Modul (eine Klasse) zu ändern.

Die Stärke der Kohäsion und gleichzeitig der zugrundeliegenden funktionalen Verwandtschaftsbeziehung nimmt entsprechend der Reihenfolge in der Aufzählung zu.

Lösungsraum-bedingte Kohäsionsarten
- **Zufällige Kohäsion (0)**
- **Logische Kohäsion (1)**
- **Temporale Kohäsion (3)**

Prozedurale Kohäsion (5)

Problemraum-bedingte Kohäsionsarten
- **Kommunikative Kohäsion (7)**
- **Sequentielle Kohäsion (9)**
- **Funktionale Kohäsion (10)**

Abbildung 2.1: Stärke der Kohäsionstypen (als Punktebewertung in den Klammern) und Zuordnung zum Problem- und Lösungsraum
nach [Yourdon and Constantine, 1979, S. 136]

Nach [Yourdon and Constantine, 1979, S. 136] ist dabei zusätzlich zu beachten, dass sich die ersten drei Kohäsionstypen (zufällige, logische und temporale Kohäsion) aus dem Lösungsraum ergeben. Sie entstehen also durch Entscheidungen der Entwickler*innen bzw. Architekt*innen bei der Entwicklung des Quellcode oder der Architektur. Diese Lösungsraum-bedingten Kohäsionstypen sind so im Zweifelsfall vermeidbar und im eigentlichen Sinne nicht Teil des zu lösenden Problems.

Die letzten drei Kohäsiontypen (kommunikative, sequentielle und funktionale Kohäsion) ergeben sich hingegen aus dem Problemraum selbst und sind diesem inhärent. Diese Problemraum-bedingten Kohäsiontypen sind deshalb bei der Dekomposition des Gesamtproblems in kleine Teilprobleme von besonderer Bedeutung.

Die prozedurale Kohäsion kann wiederum weder eindeutig dem Lösung-, noch dem Problemraum zugeordnet werden.

Daraus folgt, dass bei der Modularisierung und Dekomposition des Problems (und der resultierenden Software) genau die Statements zusammengefasst werden, zwischen denen eine möglichst starke, Problemraum-bedingte funktionale Verwandtschaft besteht.

Eine Einschätzung der relativen Stärke der Kohäsionsarten bzw. der zugrundeliegenden funktionalen Verwandtschaftstypen findet sich in Abbildung 2.1.

2.4 Kopplung

Während die funktionale Verwandtschaft Beziehungen zwischen Statements beschreibt, betrachtet der Begriff der *Kopplung* die Abhängigkeiten zwischen prozeduralen Modulen.

Zwei Module A und B sind gekoppelt, wenn Modul A Statements enthält, die den Aggregateidentifier des Moduls B oder in Modul B gebundene Statements (und Daten) referenzieren.

Jede dieser Referenzen erzeugt ein Interface, welches die Modulgrenze öffnet und den Daten- und Kontrollfluss zwischen beteiligten Modulen ermöglicht, vgl. [Yourdon and Constantine, 1979, S. 87]. Jedes Modul besitzt mindestens ein Interface, da Module a priori zu einem System gehören und das Verhalten des Gesamtsystems in irgendeiner Form beeinflussen müssen. Yourdon & Constantine argumentieren nach [Yourdon and Constantine, 1979, S.87f], dass im Idealfall auch genau ein Interface, genannt *Identity Interface*, für die gesamte Funktionalität eines Moduls ausreicht, solange es die folgenden, grundlegenden Fähigkeiten besitzt:

Das Identity Interface muss...

- **Daten empfangen können** bspw. als Eingabeparameter einer Prozedur
- **Daten zurückgeben können** bspw. als Rückgabewert einer Prozedur
- **die Kontrolle erhalten können** bspw. durch den eigentlichen Prozeduraufruf durch ein anderes Modul
- **die Kontrolle wieder abgeben können** bspw. durch das Ende der Prozedurausführung. Hierbei ist es wichtig anzumerken, dass die Abgabe der Kontrolle nicht die aktive Auswahl des nächsten Kontrollstatements in einem anderen Modul meint, sondern den impliziten Rücksprung zum aufrufenden Modul und die Fortführung von dessen Kontrollfluss. Eine aktive Auswahl würde eine weitere entgegengerichtete Verbindung bzw. Interface neben dem Identity Interface erzeugen und vom Idealfall abweichen.

Das Identity Interface entspricht dabei aus technischer Sicht (und in heute typischen Programmiersprachen) dem Prozedurkopf bestehend aus dem Rückgabetyp, dem Prozedurnamen (Aggregatsidentifikator) und einer Liste von Argumenten. Im Sinne des Structured Design können aber prozedurale Module auch über mehrere Interfaces verfügen, die bspw. aus einer Referenz auf lokal gebundene Variablen oder dem externen Zugriff auf eine Untermenge der gebundenen Statements enstehen.

> *„Obviously, what we are striving for is loosely coupled systems - that is, systems in which one can study (or debug, or maintain) any one module without having to know very much about any other modules in the system.“*
>
> [Yourdon and Constantine, 1979, S.85]

Anders als heute üblich versteht Structured Design die Stärke der Abhängigkeit/Kopplung zwischen Modulen zunächst nicht als technische Metrik, sondern definiert sie wiederum als Wahrscheinlichkeit, dass Entwickler*innen bei der Modifizierung eines Moduls Wissen über andere Module berücksichtigen müssen. Eine Abhängigkeit eines prozeduralen Moduls manifestiert sich allerdings auf technischer Ebene durch den Aufruf bzw. Nutzung eines fremden prozeduralen Moduls mithilfe seines Interface.

Der Grad der Kopplung wird dabei laut [Yourdon and Constantine, 1979, S.86f] von vier Faktoren beeinflusst:

1. **Verbindungtyp zwischen Modulen eines Systems**
   Ein System wird **minimal verbunden** (englisch: *minimally connected*) genannt, wenn alle Module nur das *Identity Interface* besitzen. Ein minimal verbundenes System ist dabei gleichzeitig, unter Berücksichtigung der anderen drei, weiter unten ausgeführten, Einflussfaktoren, **minimal gekoppelt**.

   Die nächststärkere Form der Kopplung findet sich in **normal verbundenen** (englisch: *normally connected*) Systemen. **Normal verbundene** Systeme sind aus prozeduralen Modulen aufgebaut, die entweder…

   - mehr als ein Interface besitzen (neben dem Identity Interface),

- deren Identity Interface es dem aufrufenden Modul ermöglicht, die Rücksprungadresse explizit zu steuern
  oder
- deren Identity Interface die Kontrollübertragung ohne impliziten Rücksprung erlaubt.

Die stärkste Form der Kopplung finden sich in **pathologisch verbundenen** (englisch: *pathologically connected*) Systemen.

Pathologisch verbundene Systeme bestehen aus Modulen, die…

- Referenzen auf Datenelemente innerhalb der Grenze fremder Module besitzen
  oder
- die Kontrolle an Kontrollstrukturen innerhalb der Grenzen fremder Module ohne einen impliziten Rücksprung abgeben.

2. **Komplexität des Interface**
   Die Komplexität des Interface bzw. der einzelnen Verbindungen stellt den zweiten Einflussfaktor auf den Grad der Kopplung dar.

   Die Komplexität entspricht dabei der Anzahl an lexikalischen Elementen und deren Struktur, die zur Nutzung und zum Verständnisses des Interface durch Entwickler*innen benötigt werden. Hierunter fallen bspw. benötigte Typenbezeichungen oder (verschachtelte) Variablennamen, nicht aber die dahinterliegende Datenmenge an sich.

   Vereinfacht gesagt ist ein Interface, das aus einer Prozedur mit zwei Übergabeparametern und einem primitiven Rückgabewert, weniger komplex als ein Interface, dessen Prozedursignatur aus zwanzig Übergabeparametern und einem komplexen Datentyp als Rückgabewert besteht.

3. **Informationsfluss entlang der Verbindungen**
   Der dritte Einflussfaktor auf den Grad der Kopplung liegt in der Art von Informationen, die entlang einer Verbindung zwischen Modulen fließen.

Die Art der Informationen können dabei unterschieden werden in die Kategorien Dateninformationen, Kontrollinformationen und einer Mischform der beiden Informationsarten:

- **Datenkopplung**
  [Yourdon and Constantine, 1979, S.90f] zeigt zunächst das der reine Austausch von Daten zwischen Modulen für jegliche gewünschte Funktionalität eines Gesamtsystems ausreicht. Die Datenkopplung stellt deshalb unter Berücksichtigung der anderen Einflussfaktoren die schwächste Form der Kopplung dar, da jegliche sonstige Art der Informationsübertragung die Verbindung bzw. das Interface schwerer verständlich machen würde. Mit Daten sind hierbei solche Informationen gemeint, die sich direkt aus dem Problemraum selbst ergeben, bspw. beschreibende Daten wie Adressen, Alter, Haarfarbe oder zur Identifikation nutzbare Daten wie Namen, Kunden- oder Produktnummern und Ähnliches.

- **Kontrollkopplung**
  Eine stärkere Art der Kopplung besteht beim Austausch von Kontrollinformationen.

  Als Kontrollinformation sieht [Yourdon and Constantine, 1979, S.91f] hier schon die reine, aktive Übertragung des Kontrollflusses an. Sie erzeugt ein ***aktivierende* Kontrollkopplung**. Die Entscheidung wann fremdes Modul die Kontrolle übernimmt liegt hierbei beim aufrufenden Modul und wird bspw. in Form eines Prozeduraufrufs übertragen. Die Unabhängigkeit bzw. Autonomie des aufgerufenden Moduls wird durch Abnahme dieser Entscheidung gemindert.

  Kontrollinformationen können darüber hinaus auch zusätzliche Parameter in Form von Flags enthalten, die direkten Einfluss auf den internen Kontrollfluss des aufgerufenen Moduls nehmen. Hierbei wird nicht nur die reine Übertragung des Kontrollfluss kommuniziert, sondern auch eine Entscheidung über die interne Funktionalität des aufgerufenen Moduls übertragen - das aufrufende Modul *koordiniert* den Ablauf des aufgerufenen Moduls.

Diese Form der Kopplung wird deshalb ***koordinierende*** **Kontrollkopplung** genannt und mindert die Unabhängigkeit des aufgerufenen Moduls weiter.

- **Hybride Kopplung**
  Die stärkste Form der Kopplung (unter der Annahme, dass die anderen Einflussfaktoren gleichbleiben) besteht, wenn ganze Kontrollsequenzen durch das aufrufende Modul übertragen und vom aufgerufenen Modul ausgeführt werden müssen. Diese Kontrollsequenzen sind aus Sicht des aufrufenden Moduls als Dateninformationen, aus Sicht des aufgerufenen Moduls als Kontrollinformationen, anzusehen. Die resultierende Kopplung wird deshalb ***hybride*** **Kopplung** genannt, vgl. [Yourdon and Constantine, 1979, S.92f].

  Das aufgerufene Modul verliert nahezu sämtliche autonome Entscheidungsgewalt und ist zu großen Teilen abhängig vom aufrufenden Modul.

4. **Bindungszeit intermodularer Verbindungen / Inhaltskopplung**
   Der letzte Einflussfaktor auf den Kopplungsgrad wird in der Originalquelle zu Structured Design [Yourdon and Constantine, 1979, S.93ff] zunächst etwas verwirrend mit **Bindungszeit intermodularer Verbindungen** (englisch: *Binding time of intermodular connections*) betitelt. Diese Benennung beschreibt zwar die Dimension des Kopplunsgrads, nicht aber die eigentlich relevante Verbindungsart, welche von dieser betroffen ist.

   Der Faktor findet hierbei Anwendung auf ***inhaltsgekoppelte*** **Verbindungen**: Verschiedene Module *beinhalten* denselben Datensatz oder dieselbe Funktionalität (bspw. in Form einer Prozedur) - sie sind *inhaltsgekoppelt*. Die Nutzung derselben Daten kann dabei je nach technischer Umsetzung in den verschiedensten Phasen des Softwareentwicklungs- und Softwarelebenszyklus (hier Bindungszeit genannt) entstehen.

   Ein anschauliches, wenn auch vereinfachtes, Beispiel findet sich in der Nutzung derselben, zunächst hart gecodeten Zahlwertes in den verschiedenen Modulen:

- Wird dieser Zahlenwert als sogenannte *magic number* in Schleifenköpfen in verschiedenen Modulen verwendet, werden die Module zur **Entwicklungszeit** aneinander *gebunden* und gekoppelt.
- Verschiebt man diese *magic number* aber in eine Konstante, werden die Module erst zur **Kompilierzeit** *gebunden*, da die eigentliche Schleifenvariablenbelegung erst beim Kompilieren erfolgt.
- Weiterhin gibt es die Möglichkeit die Konstante in eine externe, statische Bibliothek zu verschieben und die Bindung/Koppelung findet erst zum **Zeitpunkt des Linken** des Programmcodes statt.
- Werden die Daten von den verschiedenen Modulen bei Programmstart geladen, besteht eine Kopplung erst zum **Initialisierungszeitpunkt**.
- Verschiebt man den Zeitpunkt weiter nach hinten und die verschiedenen Module laden die Daten je nach Bedarf, also *just-in-time*, spricht man von einer Bindung/-Kopplung zur **Laufzeit**.

Mit jedem Schritt nimmt der *inhaltliche* Kopplungsgrad jeweils ab. Eine Inhaltskopplung kann hierbei natürlich auch durch komplexere Datensätze wie beispielsweise Kunden-, Nutzer- oder Bestellungsdaten entstehen, die dann typischerweise erst zur Laufzeit in Gänze oder nur zum Teil in verschiedenen Modulen geladen werden.

Das angeführte Beispiel lässt sich auch für funktionale Inhalte (wie Prozeduren) veranschaulichen:

- Wird dieselbe Funktionalität in verschiedenen Modulen benötigt, entsteht eine starke Kopplung zur Entwicklungszeit durch das einfache Kopieren einer Prozedur oder Statement-Sequenz in die verschiedenen Module.
- Eine Auslagerung in eine statische Bibliothek verschiebt die Kopplung auf den Zeitpunkt des Linken des Programmcodes.

- Wird dieselbe Prozedur durch RPC (*Remote Procedure Calls*) oder REST-Calls in verschiedenen Modulen verwendet, spricht man von einer inhaltlichen Bindung/-Kopplung zur Laufzeit.

Der *inhaltliche* Kopplungsgrad nimmt wie oben mit jedem Schritt ab.

Eine Änderung an den geteilten Inhalte betrifft immer alle inhaltlich gekoppelten Module - wie aufwendig diese Änderung allerdings ist, hängt von der Bindungszeit der Inhalte ab.

Anzumerken ist desweiteren, dass die Inhaltskopplung (englisch: content coupling) nach [Richards and Ford, 2020, S. 53] die stärkste Form der Kopplung darstellt und nach Möglichkeit gänzlich vermieden werden sollte. Zur Vermeidung müssen dabei Beziehungen im Problemraum erneut untersucht, bewertet und neue Modulgrenzen gezogen werden - eine detailliertere Betrachtung hierzu findet sich in Kapitel 2.3 *Kohäsion*. Sollte sich eine inhaltliche Koppelung trotzdem nicht vermeiden lassen, gibt die hier beschriebene Möglichkeit der Minderung im Lösungsraum der Anwendung, also durch technische Softwaremuster.

Einige der oben genannten Faktoren der Kopplungsstärke sind aus heutiger Sicht zugegebenermaßen nur schwer verständlich, da heutige Programmiersprachen (insbesondere Java) bspw. den Zugriff auf lokale Variablen einer Prozedur verhindern. Eine pathologische Kopplung hinein in eine (Java) Prozedur ist somit nicht mehr möglich. Dennoch kann diese Sichtweise insbesondere bei der Bewertung von Klassenkopplung nützlich sein, da (Java) Klassen einzelne Variablen (per Zugriffsoperator) öffentlich sichtbar machen können und so wieder eine pathologische Kopplung ermöglichen.

Eine deutlich einfacher verständliche und übersichtlichere Kategorisierung der unterschiedlicher Kopplungstypen findet sich in [Myers, 1975, S.33ff]:

- **Content coupling** *Inhaltskopplung*
  Ein Modul ist mit einem zweiten Modul inhaltlich gekoppelt (*content coupled*), falls es…

  - auf Daten des zweiten Moduls zugreift oder sie modifiziert
    oder
  - Teile der Funktionalität des zweiten Moduls, bspw. per Prozeduraufruf, nutzt.

  Die Daten bzw. die Rückgabewerte der genutzten Prozedur sind dabei zwingend für die Funktionalität des ersten Moduls notwendig.

  Myers beschreibt zusätzlich eine Inhaltskopplung, die durch die Überlappung der Speicherbereiche der Module entsteht. Der beschriebene geteilte Besitzanspruch auf denselben Speicherbereich wird allerdings in modernen gemanagten Programmiersprachen nicht mehr unterstützt.

  Änderung an den internen Daten des zweiten Moduls oder an der Logik der genutzten Prozedur betreffen mit hoher Wahrscheinlichkeit auch die Funktionalität des ersten Moduls. Eine isolierte Wart- oder Erweiterung des zweiten Moduls ist kaum möglich.

  Inhaltskopplung stellt im Allgemeinen die stärkste Form der Kopplung dar.

- **Common (environment) coupling** *Umgebungskopplung*
  Zwei Module sind durch eine gemeinsame Umgebung gekoppelt (*common coupled*), falls beide Module auf denselben, *komplexen* Datensatz zugreifen. Der betreffende Datensatz liegt hierbei nicht in einer der Module selbst, sondern befindet sich in der externen Umgebung der Module.

  Diese Art der Kopplung kann beispielweise durch den Zugriff beider Module auf dieselbe globale, komplexe Variable entstehen. Ein heute üblicheres Beispiel liegt im Zugriff der beiden Module auf dieselbe Datenbank-Entity oder Datenbanktabelle.

- **External coupling** *Externe Kopplung*
  Zwei Module sind extern gekoppelt, falls beide Module auf dasselbe externe Datenelement zugreifen - gemeint sind hierbei *primitive* Daten, aber nicht Daten in Form eines komplexen Datentyps. Auch hier liegt das betreffende Datenelement in der externen Umgebung der beiden Module.

Der primitive Datentyp macht nötige Änderungen (im Vergleich zu umgebungsgekoppelten Modulen) aber unwahrscheinlicher und einfacher durchzuführen.

- **Control coupling** *Kontrollkopplung*
  Ein Modul ist mit einem zweiten Modul kontrollgekoppelt, wenn das erste Modul Informationen (sogenannte Kontrollelemente) übergibt, die den Kontrollfluss des zweiten Moduls beeinflussen. Beispiele hierfür sind die Übergabe von einfachen booleschen Flags oder die Übergabe von Funktionsobjekte als Argumente eine Prozeduraufrufs.

- **Stamp coupling** *Typkopplung*
  Zwei Module sind typgekoppelt, falls sie denselben komplexen Daten- oder Objekttyp verwenden. Im Falle eines Datentyps entsteht die Kopplung hierbei nicht durch die Daten an sich (also die Wertbelegung einer komplexen Variable), sondern durch die definierte Struktur der Daten.

- **Data coupling** *Datenkopplung*
  Zwei Module sind datengekoppelt, wenn alle der obigen Kopplungsarten *nicht* bestehen und die Module alle nötigen Daten in Form von primitiven Datentypen austauschen.

  Datenkopplung ist die schwächst mögliche Kopplungsform zwischen Modulen.

Die Stärke der Kopplungstypen nimmt folgend der obigen Reihenfolge jeweils ab.

2.5 Designprinzip der hohen Kohäsion und losen Kopplung

Nimmt man die beiden oben beschriebenen Metriken zusammen, ergibt das weitläufig bekannte Designprinzip der hohen Kohäsion & losen Kopplung (*High Cohesion & Low Coupling*). Die Beachtung dieses Designprinzip hat dabei direkte Auswirkung auf die Morphologie und architektonischen Eigenschaften der Module und des Gesamtsystems.

„. . . a highly modular design is achieved by maximizing the relationships among the elements of a module and minimizing the relationships among modules, the scale for coupling is inverse to the scale for strength.“

[Myers, 1975, S.33]

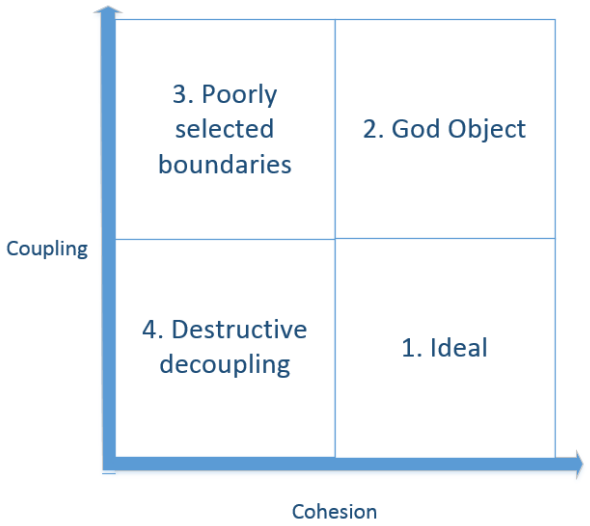


Abbildung 2.2: Verschiedene Quellcodetypen unter Berücksichtigung von Kopplung und Kohäsion
Bildquelle: [Khorikov, 2015]

Je nach Befolgung der beiden Dimensionen des Designprinzips lässt sich die resultierende Architektur nach [Khorikov, 2015] in die folgenden vier Kategorien einteilen:

- **Ideal**
  Die Architektur folgt den oben aufgeführten Designkriterien vollständig, sie besitzt maximale Kohäsion und ist minimal gekoppelt - sie ist ideal. Funktional verwandte Elemente der Domäne (und des Codes) bilden Cluster und formen ein Modul. Nötiges funktional verwandtes Wissen ist lokalisiert. Die Softwarequalitäten der Test-, Wart-, Änder- und Erweiterbarkeit sind hier maximal.

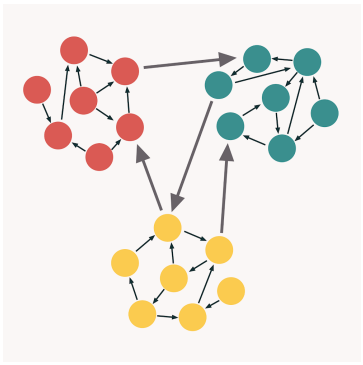


Abbildung 2.3: Hoch kohäsive und lose gekoppelte, ideale Architektur
Bildquelle: [Khorikov, 2015]

- **Gott Objekt**
  Die Software ist gleichzeitig stark gekoppelt und hoch kohäsiv, Modulgrenzen sind nicht mehr zu erkennen und das System kann nur als eine große Einheit, als Gott Objekt, betrachtet werden. Ein Gott Objekt entspricht hierbei der Big Ball of Mud-Architektur oder kann als Architektur-Monolith bezeichnet werden. Der Quellcode/der Problemraum ist wenig strukturiert, es bestehen Abhängigkeiten zwischen den verschiedensten Teilen der Software und nötiges Wissen für Änderungen ist so im ganzen System verteilt. Eine solche Architektur ist nur minimal test-, wart-, änder- und erweiterbar.

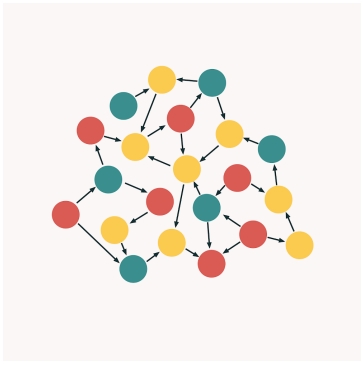


Abbildung 2.4: Hoch kohäsive und stark gekoppelte Architektur, auch bekannt als Gott Objekt oder Big Ball of Mud-Architektur
Bildquelle: [Khorikov, 2015]

- **Schlecht gewählte Modulgrenzen**
  Schlecht gewählte Modulgrenzen ergeben hoch gekoppelte und wenig kohäsive Module. Strukturen im Quellcode sind zwar erkennbar - die Module wirken aber zusammenhangslos, da ihre Elemente keine funktionale Verwandtschaft aufweisen. Eine solche Architektur ist nur minimal test-, wart-, änder- und erweiterbar.

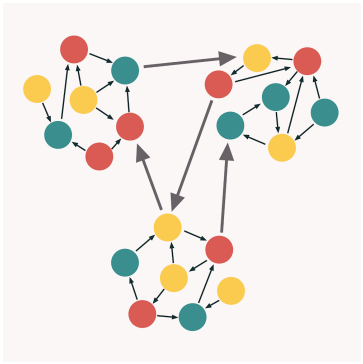


Abbildung 2.5: Wenig kohäsive und lose gekoppelte Architektur als Resultat schlecht gewählter Modulgrenzen
Bildquelle: [Khorikov, 2015]

- **Destruktive Entkopplung**
  Bei geringer Kohäsion und geringer Kopplung ergibt sich ein destruktiv entkoppelter Code. Das System wurde so weit dekomponiert bzw. modularisiert bis es nurnoch als Sammlung einzelner Elemente angesehen werden kann. Eine Bindung der funktional verwandten Elemente findet nicht statt und nötiges, verwandtes Wissen bleibt im System verteilt bzw. wird sogar weiter voneinander entfernt. Strukturen und Module sind, ähnlich zum Gott Objekt, nicht mehr zu erkennen. Auch hier sind die Softwarequalitäten der Test-, Wart-, Änder- und Erweiterbarkeit minimal.

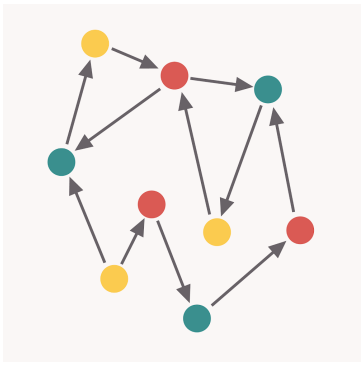
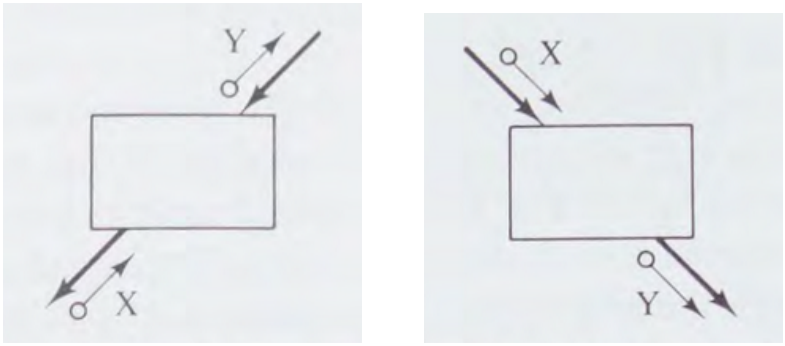


Abbildung 2.6: Wenig kohäsive und lose gekoppelte Architektur als Resultat einer übertriebenen, destruktiven Entkopplung
Bildquelle: [Khorikov, 2015]

2.6 Weitere Moduleigenschaften

Structured Design kennt des Weiteren verschiedene andere Kategorien von Modulen, die im Folgenden exemplarisch besprchen werden.

Die erste Kategorie unterteilt Module anhand des vorherrschenden Datenflusses:



(a) (b)

Abbildung 2.7: (a) afferenter und (b) efferenter Datenfluss zwischen Modulen
Bildquelle: [Yourdon and Constantine, 1979, S. 153f]

- **Afferente Module** erhalten Daten von einem untergeordnetem Modul bspw. als Rückgabewert eines Methodenaufrufs und gibt diese an ein übergeordnetes Modul zurück. Daten- und Kontrollfluss laufen in entgegengesetzter Richtung.
- **Efferente Module** erhalten hingegen Daten von einem übergeordneten Modul, gibt sie an ein oder mehrere Module weiter. Der Datenfluss folgt hierbei dem Kontrollfluss.

Die zweite Kategorie unterteilt Module anhand des vorherrschenden Kontrollflusses zwischen Modulen:

- Der **Fan-Out** (auch genannt *span of control*) eines Moduls beschreibt wie viele andere Module von ihm aufgerufen werden.
- Der **Fan-In** eines Moduls beschreibt von wie vielen anderen Module es aufgerufen wird.

3 Objektorientierte Programmierung

Da aus heutiger Sicht eine Dekomposition der Software in prozedurale Module nicht ausreicht, beschäftigt sich dieses Kapitel mit dem Programmierparadigma der objektorientierten Programmierung (OOP). Die objektorientierte Programmierung führt hierbei zunächst Objekte bzw. Klassen als neues strukturgebende Elemente in die Programmierung ein. Das Programmierparadigma wurde Ende der 1960er Jahre durch Alan C. Kay erfunden und mit der Programmiersprache Smalltalk erstmalig implementiert.

3.1 Definition

Alan C. Kay defniniert die objektorientierte Programmierung wie folgt:

> *„1. Everything is an object*
> *2. Objects communicate by sending and receiving messages (in terms of objects)*
> *3. Objects have their own memory (in terms of objects)*
> *4. Every object is an instance of a class (which must be an object)*
> *5. The class holds the shared behavior for its instances (in the form of objects in a program list)*

> *6. To eval a program list, control is passed to the first object and the remainder is treated as its message“*
>
> [Kay, 1993, S. 78]

Alan Kay postuliert hierbei in Punkt 1, dass eine Software aus Objekten aufgebaut sein sollte. Er impliziert damit gleichzeitig, dass der Quellcode einer Software dekomponiert werden sollte und bietet so eine neue Form der Modularisierung an[1].

Objekte entstehen dabei durch die Instanzierung von Klassen (Punkt 4) und binden eine Menge von Statements, die Verhalten der Objekte erzeugen (Punkt 5), an eine Menge von Daten (Punkt 3). Die gebundene Menge der Statements besteht hierbei aus prozeduralen Modulen, die im objektorientierten Umfeld auch Methoden genannt werden.

3.2 Bewertung der OOP aus Sicht des Modulbegriffs

Die Klassen und Objekte der objektorientierten Programmierung erfüllen hierbei die Definition von Modulen aus Kapitel 2:

- Sie fassen erstens eine **lexikalisch zusammenhängende Sequenz von Programmstatements** zusammen, also typischerweise den Klassenkonstruktor, Variablendeklaration und -zuweisung, und zugehörige Methoden.
- Sie verfügen zweitens über ein **begrenzendes Element**, typischerweise in Form von geschweiften Klammern, die den Klassenrumpf definieren.
- Sie besitzen drittens einen **Aggregatsidentifikator** in Form des Klassennamens bzw. in Form der Speicheradresse eines Objekts.

[1] Hierbei ist anzumerken, dass sich die OOP zunächst nur auf die Dekomposition der Software, also dem Lösungsraum, selbst konzentriert, nicht aber auf die Dekomposition der Problemstellung an sich. Erst mit dem Aufkommen der Objektorientieren Analyse wird die Modularisierung durch Objekte auch auf den Problemraum übertragen, indem Objekte in der Software als Abbildung von realen Objekten in der Welt angesehen werden.

Objekte binden hierbei genau die Daten, die durch einzelne Statements der gebundenen Methoden genutzt werden. Diese Nutzung impliziert eine funktionale Verwandtschaft/Abhängigkeit zwischen ihnen, die im Idealfall eher hohe Problemraum-bedingte Kohäsiontypen abbildet. Die in Kapitel 2.3 aufgeführten Kohäsiontypen können hierbei also genutzt werden, um möglichst hochkohäsive Klassen- und Objektmodule zu erzeugen.

Die Punkte 2 und 6 der oben genannten Definition beschäftigen sich mit der Interaktion von Objekten und schlagen eine Kommunikation über das Versenden (und Empfangen) von Nachrichten vor. Eine Nachricht ist hierbei nach [Page-Jones, 2000, S. 19f] das Mittel, mit dem ein Senderobjekt einem Zielobjekt eine Aufforderung übermittelt, eine seiner Methoden anzuwenden. Nachrichten bestehen hierbei aus:

- Dem **Identifier des Zielobjekts**
  Typischerweise in Form der Speicheradresse des Objekts, die in Vorhinein im Senderobjekt als Referenz in einer Variable gespeichert wurde.
- Dem **Namen der Methode des Zielobjekts**, die ausgeführt werden soll
- Alle **zusätzlichen Informationen** (Argumente), die das Zielobjekt für die Ausführung seiner Operation benötigt

Die hier beschriebene Form der Kommunikation über Nachrichten beschreibt also eine unidirektionale Kommunikation, da eine Beantwortung der Nachrichten ohne Informationen über den Absender gar nicht erst möglich ist. Eine derartige Kommunikation würde eine höchststarke, inhaltliche Kopplung zwischen Objekten nahezu ausschließen, da eine Nutzung von Daten oder Funktionalität des Zielobjekts durch das Senderobjekt eine Antwort voraussetzen würde. Nachrichten würden so zur losen Kopplung zwischen Klassen- und Objektmodulen beitragen und die Modularisierung des Gesamtsystems unterstützen.

Überraschenderweise findet das Konzept der Kommunikation über Nachrichten in der heutigen Lehrmeinung im Bereich der OOP kaum Anwendung. Vielmehr wird heute die Kommunikation zwischen Objekten meist durch den Methodenaufrufe mit möglichen Rückgabewerten abgebildet. Dieses potenziell bidirektionale Form der Kommunikation erlaubt eine inhaltliche

Kopplung von Objekten und ignoriert die entkoppelnden Eigenschaften der nachrichtenbasierten Kommunikation.

Dieser Umstand wird auch von Alan Kay bedauert, wie folgendes Zitat beweist:

> *„The big idea is 'messaging' [...] The Japanese have a small word - ma - for 'that which is in between' - perhaps the nearest English equivalent is 'interstitial'. The key in making great and growable systems is much more to design how its modules communicate rather than what their internal properties and behaviors should be.“*
>
> [Kay, 1998]

Die Idee der nachrichtenbasierten Kommunikation von Modulen zur Unterstützung einer loseren Kopplung schien lange vergessen und findet heute eher auf makroarchitektonischer Ebene, etwa bei Microservices, Anwendung. Ihre Bedeutung auch auf mikroarchitektonischer Ebene wird aber mit zunehmender Relevanz von Kopplungskonzepten immer wichtiger, vgl. [Schlosser, 2019].

3.3 Connascence

Meilier Page-Jones erweitert in [Page-Jones, 1995, S. 180f] das Verständnis von prozeduralen Modulen des Structured Design um Objektmodule. Er argumentiert, dass eine Modularisierung und Strukturierung von Quellcode mit dem Aufkommen der objektorientierten Programmierung nun auf verschiedenen Ebenen von Software stattfinden kann.

Die bekannte Bindung einzelner verwandter Programmstatements hin zu prozeduralen Modulen ist für ihn dabei nur ein erster Schritt. Ein zweiter Schritt besteht in der Bindung von verwandten prozeduralen Modulen und von ihnen genutzte Daten in Klassen und Objekten, siehe Abbildung 3.1.

Page-Jones erwähnt hier sogar damals schon mögliche weitere Schritte, die verwandte Klassen zu Modulen auf noch höheren Ebenen zusammenfasst, vgl [Page-Jones, 1995, S.181]. Diese

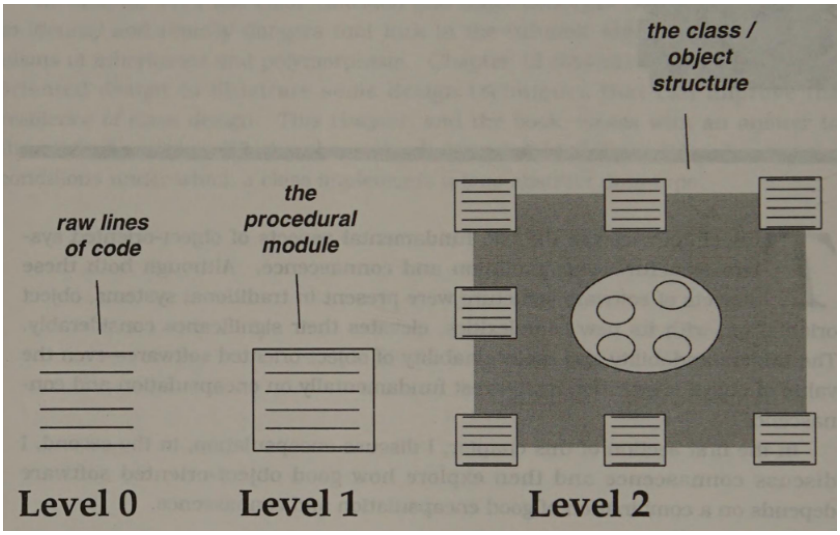


Abbildung 3.1: Ebenen der Modularisierung: atomaren Programmstatements (links) über prozedurale Module (Mitte) hin zu Klassen- und Objektstrukturen der OOP Bildquelle: [Page-Jones, 1995, S. 180]

höheren Modulstrukturen bezeichnet man heute als Komponenten oder als Module, die durch Paketstrukturen gebildet werden.

Nach Page-Jones bestehen jeweils leicht unterschiedliche Metriken zur Bewertung von Abhängigkeiten, je nachdem welche Kombination von abhängigen Modultypen (und einfachen Programmstatements) betrachtet wird, siehe Tabelle 3.1.

Zur Vereinfachung schlägt Page-Jones deshalb die verallgemeinerte Metrik der Connascence vor. Connascence beschreibt hierbei die Art und Weise, wie sich Änderungen in einem Teil eines Systems auf andere Teile auswirken.

Der Begriff leitet sich dabei aus dem Lateinischen ab („com“ und „nasci“) und bedeutet etwa „zusammen geboren“. Connascene beschreibt also die enge Verbundenheit zwischen „zusammen geborenen“ Teilen von Software und impliziert laut Page-Jones gleichzeitig deren gemeinsames Schicksal im Lebenszyklus der Software.

Je höher das Connascence-Niveau zwischen zwei Teilen eines Systems ist, desto stärker sind

To: From:	level-0 construct (line of code)	level-1 construct (procedure)	level-2-construct (class)
level-0 construct (line of code)	Structured programming	Message Fan-out	-
level-1 construct (procedure)	Cohesion	Coupling	-
level-2 contruct (class)	-	Class cohesion	class coupling

Tabelle 3.1: Metriken zur Bewertung von Abhängigkeiten unterschiedlicher Modultypen, aus [Page-Jones, 1995, S. 182]

sie gekoppelt, was bedeutet, dass Änderungen an einem Teil wahrscheinlicher unerwünschte Auswirkungen auf den anderen Teil haben.

Laut [Page-Jones, 1995, S.183] tritt Connascence zwischen zwei Teilen der Software A und B auf, falls

- Änderungen an Teil A denkbar sind, die gleichzeitig eine Änderung an Teil B nötig machen

  oder

- Änderungen an anderer Stelle denkbar sind, die eine gleichzeitige Änderung beider Teile nötig macht.

Page-Jones unterscheidet hierbei nach [Page-Jones, 1995, S. 185ff] zwischen neun verschiedenen Arten der Connascence und teilt diese zunächst in den beiden grundlegenden Kategorien der statischen Connascence und der dynamische Connascence ein:

Static connascence
Die statischen Arten der Connascence entstehen zur Entwicklungszeit der Software.

- **Connascence of Name**
  Zwei Elemente besitzen Connascence of Name, wenn sie sich auf eine Benennung von Klassen, Methoden oder Variablen geeinigt haben. Wenn diese Benennung geändert wird, muss das abhängige Element die Änderung übernehmen, damit die entsprechende Referenz weiterhin gültig und auswert- bzw. aufrufbar bleibt.

```
1 int i;
2 i = 42;
```

Listing 3.1: Connascence of Name
am Beispiel von Variablen-deklaration und -zuweisung

Zwischen den beiden Programmstatements in Listing 3.1 besteht Connascence of Name, da eine Änderung des Variablenbenennung bei der Deklaration in Zeile 1, gleichzeitig eine Änderung bei der Zuweisung in Zeile 2 nötig macht.

Connascence of Name stellt allgemein die leichteste Form der Connascence dar. Ihr Auftreten lässt sich nicht vermeiden, da Elemente eines Gesamtsystems per Definition etwas miteinander zu tun haben müssen.

Glücklicherweise sind Änderungen die auf Connascence of Name basieren heute leicht und kostenneutral zu realisieren. Heutige Entwicklungsumgebungen stellen hierfür Werkzeuge zum Refactoring bereit, die Benennungen über Klassen-, Modulgrenzen hinweg konsistent ändern können.

- **Connascence of Type or Class**
  Zwei Elemente besitzen Connascence of Type oder Class, wenn sie sich auf Nutzung eines bestimmten (Variablen-)Typs geeinigt haben.

  Zwischen den beiden Programmstatements in Listing 3.1 besteht gleichzeitig auch Connascence Of Type. Eine Änderung des Variablentyps (bspw. *String*, statt *int*) bei der Deklaration in Zeile 1, würde gleichzeitig eine Änderung der Zuweisung in Zeile 2 nötig machen, bspw. *i = "foobar";* oder *i = "3";*.

- **Connascence of Meaning or Convention**
  Zwei Elemente besitzen Connascence of Meaning oder Convention, wenn sie sich auf eine semantische Bedeutung geeinigt haben.

  Ein typisches Beispiel findet sich in der Zuordnung von TRUE und FALSE zu den Integerwerten 1 und 0.

- **Connascence of Algorithm**
  Zwei Elemente besitzen Connascence of Algorithm, wenn sie sich auf die Nutzung eines bestimmten Algorithmus geeinigt haben.

  Ein Beispiel findet sich im Bereich Security: Nutzt ein Teil der Software den Hashalgorithmus SHA256 zur Erzeugung von Signaturen, müssen andere Teile der Software denselben Hashalgorithmus nutzen, um die erzeugten Signaturen überprüfen zu können. Soll der Algorithmus ausgetauscht werden, müssen gleichzeitig beide Elemente/Teile der Software abgeändert werden.

- **Connascence of Position**
  Zwei Elemente besitzen Connascence of Position, wenn sie sich auf eine bestimmte Position oder Reihenfolge geeinigt haben.

  Ein typisches Beispiel ist die Reihenfolge von Argumenten einer Prozedur, siehe Listing 3.2.

```
1  int divide(int dividend, int divisor){
2      return dividend / divisor;
3  }
4  int quotient = divide(30, 10)
```

Listing 3.2: Connascence of Position
am Beispiel von Prozedurargumenten

Zeilen 1 und Zeile 4 in Listing 3.2 besitzen Connascence of Position, da sie sich auf die Reihenfolge der Prozedurargumente geeinigt haben. Ändert man den Prozedurkopf hin zu *int divide(int divisor, int dividend)* muss auch der Prozeduraufruf geändert werden zu *int quotient=divide(10, 30)*, um weiterhin ein korrektes Ergebnis zu erhalten.

Dynamic connascence

Die dynamischen Arten der Connascence enstehen erst zur Laufzeit der Software.

- **Connascence of Execution**
  Zwei Elemente besitzen Connascence of Execution, wenn eine bestimmte Ausführungsreihenfolge geeinigt haben.

  Beispielsweise muss die Deklaration einer Variablen immer vor der Variablenzuweisung geschehen. Wird die Deklaration im Quellcode nach hinten verschoben, müssen auch alle Zuweisungen hinter die neue Position im Quellcode verschoben werden.

- **Connascence of Timing**
  Zwei Elemente besitzen Connascence of Timing, wenn ihre Ausführung die Einhaltung eines bestimmten Zeitintervalls voraussetzt.

  Beispielweise besteht zwischen Client und Server Connascence of Timing, wenn sie sich auf einen Timeout für Abfragen geeinigt haben. Wird die Beantwortung der Abfrage durch eine Änderung komplexer und dauert so länger, muss auch Clientcode angepasst werden und der Timeout der Abfrage verlängert werden.

- **Connascence of Value**
  Zwei Elemente besitzen Connascence, wenn sie zwei Werte nutzen, die voneinander abhängen und nur konsistent zueinander geändert werden können. Es besteht eine Invariante zwischen den Werten - eine Änderung der Werte muss mit transaktionaler, striker Konsistenz erfolgen.

  Beispielsweise besitzt das Ergebnis einer Addition Connascence of Value zu den Summanden, da zu keinem Zeitpunkt einer der Summanden geändert werden kann ohne auch die Summe zu ändern.

- **Connascence of Identity**
  Zwei Elemente besitzen Connascence of Identity, wenn beide Referenzen auf die selbe Entität (die selbe Identiät) besitzen.

Beispiel: Ein Bestellungsobjekt besitzt Referenzen auf mehrere Produkte. Ein Rechnungsobjekt referenziert dieselben Produkte zur Berechnung des Rechnungsbetrags. Werden nun die Produktreferenzen im Bestellobjekt geändert, müssen sich auch die Produktreferenzen im Rechnungsobjekt ändern, um den korrekten Rechnungsbetrag berechnen zu können.
Das Bestellungsobjekt und das Rechnungsobjekt besitzen also Connascence of Identity.

Die Stärke der oben aufgeführten Arten der Connascence nimmt jeweils zu.

Betrachten wir Abhängigkeiten zwischen Modulen empfiehlt Page-Jones, analog zum Low Coupling-Prinzip des Structured Design, die Reduktion der Connascence, durch Nutzung eher schwacher Connascencearten, siehe Abbildung 3.2. Ein Refactoring hin zu schwacher Connascence stärkt so die Struktur und Modularisierung des Gesamtsystems.

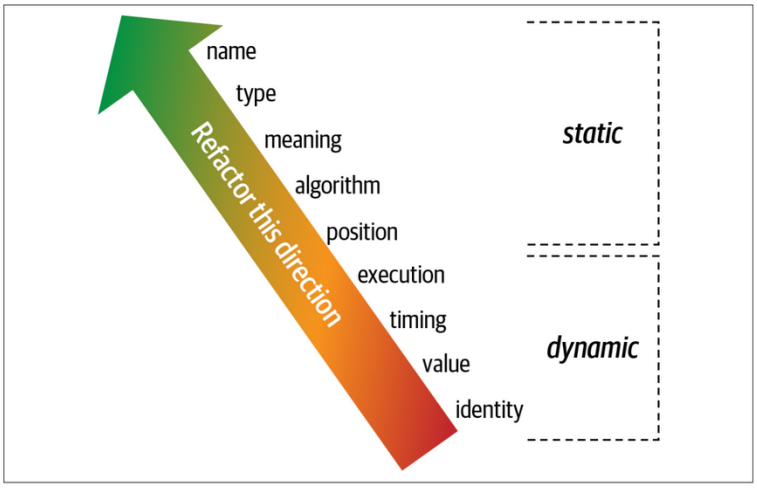


Abbildung 3.2: Refactoring nach Connascence zwischen Modulen
Bildquelle: [Richards and Ford, 2020, S. 51]

Neben der Art der Connascence führen neuere Quellen zwei weitere Dimensionen der Connascence ein, vgl. [Richards and Ford, 2020, S. 52]:

- **Lokalität** *Locality*
  Die Lokalität der Connascence misst die Entfernung zwischen abhängigen Elementen

der Software ein. Die Entfernung wird hierbei durch die Zugehörigkeit der abhängigen Elemente in der Modulhierarchie gemessen. Die Connascence zweier Elemente, die sich in unterschiedlichen Klassen-/Objektmodulen befinden, wiegt hierbei schwerer als Connascence zwischen zwei Elementen, die sich in unterschiedlichen prozeduralen Modulen, aber im selben Klassenmodul, befinden.

- **Grad** *Degree*
  Während die Art der Connascence eine qualitative Aussage über Abhängigkeiten zwischen Modulen trifft, beschreibt der Grad der Connascence die quantitative Dimension von Abhängigkeiten zwischen Modulen.

  Eine hohe Anzahl von abhängigen Elementen in verschiedenen Modulen erzeugt einen hohen Grad der Connascence der Module. Ein einzelnes Paar von Element in verschiedenen Modulen erzeugt einen geringen Grad der Connascence der Module.

Betrachtet man das Gesamtsystem schlägt Page-Jones nach [Page-Jones, 2000, S. 222ff] ein allgemeines Vorgehen zur Steigerung der Wart- und Änderbarkeit eines Systems vor:

1. *„Minimiere Connascence im Gesamtsystem durch Aufteilung des Systems in verkapselte Elementen (engl.: encapsulated elements).*
2. *Minimiere verbleibende Connascence, die sich über Kapselungsgrenzen (engl.: encapsulating boundaries) hinweg erstreckt.*
3. *Maximiere Connascence innerhalb der Kapselungsgrenzen.“*

Das beschriebene Vorgehen ist nahezu deckungsgleich zum Designprinzip der High Cohesion & Low Coupling des Structured Design. Eine Unterscheidung der Metrik bei intramodulare Abhängigkeiten (Kohäsion) und intermodularen Abhängigkeiten (Kopplung) bleibt allerdings aus.

Vergleicht man weiterhin die einzelnen Arten der Connascence mit den Kopplungstyp des Structured Design, finden sich ähnliche Konzepte insbesondere im Bereich der statischen Connascencearten, siehe Abbildung 3.3.

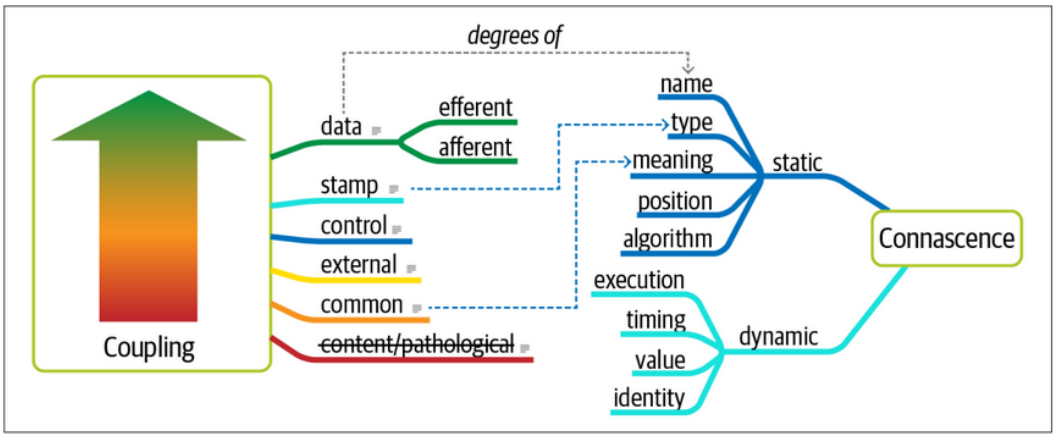


Abbildung 3.3: Vergleich zwischen Kopplung und Connascence, Bildquelle [Richards and Ford, 2020, S. 53]

Auch im Vergleich zwischen Connascencearten und den Kohäsionstypen nach Yourdon & Constantine fallen einige ähnliche Konzepte auf. Deckungsgleiche Konzepte sind dabei aber schwerer zu identifizieren insbesondere unter Berücksichtigung der Stärkegrade. Es scheint deshalb sinnvoll, dass Entwickler*innen und Architekt*innen sowohl die Metriken der Kohäsion und Kopplung als auch die Metrik der Connascence kennen, um die Abhängigkeit zwischen Softwareteilen möglichst genau einschätzen zu können.

4 Domain Driven Design

Der Entwicklungsansatz des Domain Driven Design (DDD) wurde im Jahre 2004 von Eric Evans veröffentlicht, vgl. [Evans, 2004]. Evans schlägt hierbei im Kern vor, dass sich das Design einer Anwendungsarchitektur primär an den Objekten, Strukturen und Zusammenhänge in der ”Domäne” selbst orientieren sollte. Der Begriff der *Domäne* meint hierbei den spezifischen Geschäftsbereich des Kunden oder das Problemfeld, welches die zu erstellende Software adressieren soll.

Evans widerspricht hiermit der damals weit verbreiteten Praxis architektonische Strukturen eher an technischen Aspekten auszurichten - sie also im Lösungsraum der zu entwickelnden Anwendung selbst zu suchen. DDD setzt dabei den im Kapitel 2.1 angesprochenen Ansatz des Structured Design fort und spricht sich für die Reduktion der Komplexität durch die konzeptionelle Nähe der zu designenden Lösungen zum Problemraum aus.

Evans schlägt sowohl eine strategische, makroarchitektonische als auch eine taktische, mikroarchitektonische Perspektive auf die Domäne vor:

Das strategische Design nimmt zunächst die gesamte Dömane in den Blick und teilt sie anhand sprachlicher Grenzen in sogenannte Bounded Context ein. Jeder Bounded Context grenzt dabei die Validität eines bestimmten Sprachmodells auf einen Teil der Gesamtdomäne ein. Die Domäne wird so dekomponiert bzw. modularisiert und einzelne Bounded Context könnten als Module angesehen werden. Die Grenzen dieser Module sind aber in gängigen Umsetzungen von domänengetriebenen Architekturen im Code oft nicht eindeutig zu erkennen. Sie werden zwar im Microservice-Architekturstil oft dazu genutzt um Servicegrenzen zu bestimmen, werden aber dabei meist nur als Obergrenze verstanden. Die folgenden Ausführungen

beschäftigen sich aus diesem Grund mit der zweiten Perspektive des DDD, dem taktischen Design.

Das taktische Design nimmt eine bottom-up Perspektive ein und konzentriert sich auf mikroarchitektonische Designentscheidungen. Es führt hierzu modulbildende Strukturen, genannt *Building Blocks des taktischen Designs*, auf Klassen- und Klassenverbundsebene ein. Das taktische Design dekomponiert und modularisiert hierbei die oben angesprochenen Bounded Context bzw. Subdomänen.

4.1 Building Blocks des takischen Designs

Das taktische Design führt zunächst die grundlegenden Strukturen **Entity**, **ValueObject**, **Aggregate**, **Repository** und **DomainService** ein:

Eine **Entity** beschreibt hierbei Objekte der Domäne, die geprägt sind durch ihren fortlaufenden Lebenszyklus im System. Aus einer objektorientierten Perspektive fasst sie aus Domänensicht zusammengehörige Daten- und Verhaltenselemente zusammen und bindet sie aneinander. Damit eine Entity durch andere Elemente des Systems genutzt und referenziert werden kann, besitzt sie zusätzlich eine Identität. Diese Identität wird dynamisch bei Erschaffung der Entity vergeben und bleibt über den gesamten Lebenszyklus unverändert auch wenn die Entity und ihre verinnerlichten Daten modifiziert werden. Im Code wird die Identität typischerweise dabei entweder durch eine identifizierende, immutable Variable des Typs UUID (Universally Unique Identifier) oder durch ein identifizierendes ValueObject (siehe unten) modelliert.

Demgegenüber steht das Konzept des **ValueObjects**, das eine Menge zusammengehöriger Daten zusammenfasst, aber über keine eigene, dedizierte Identität verfügt. Die Identität eines ValueObjets ist dabei vielmehr durch den Datensatz (also beinhaltende Variablen und deren Wertbelegung) selbst geprägt. Um diese Form der aus Daten abgeleiteten Identität zu gewährleisten, wird der Datensatz vor Modifizierung geschützt. Ist der Datensatz eines ValueObjects nicht mehr aktuell, so wird also nicht das ursprüngliche ValueObject verändert, sondern ein

neues ValueObject als modifizierte Kopie muss erzeugt. Eine Instanz eines ValueObjects wird dabei genau einer, besitzenden Entity zugeordnet und als Komposition an sie gebunden.

Ein **Repository** fasst die Menge aller Entity-Instanzen zusammen und hält diese zur späteren Nutzung vor. Das Repository wird dabei häufig mit einer Datenbank verwechselt, stellt aber im eigentlichen Sinne nur eine Collection unabhängig von jeglichen Persistenzmechanismen dar. Repositories sind genau einem Entitytyp zugeordnet und stellen typische Collection Methoden zur Aufbewahrung, Suchen und Löschen von Entities bereit.

Ein **DomainService** fasst Verhalten genau eines Entitytypen zusammen, welches sich nicht genau einer Entityinstanz zuordnen lässt.

Ein **Aggregate** besteht aus genau einer Entity in der Rolle des AggregateRoot (Aggregateswurzel), mehreren verschachtelten Entities und jeweils zugehörigen ValueObjects. Verschachtelte Entities verlieren dabei ihre globale Identität und werden (im Idealfall) nicht mehr von anderen Teilen des Gesamtsystems referenziert. Das AggregateRoot hingegen behält seine globale Identität und bleibt aus anderen Systemteilen referenzierbar. Das AggregateRoot dient somit als Interface für das Aggregate und verwaltet alle Änderungen betreffend gebundener Entities und ValueObjects, siehe Abbildung 4.1.

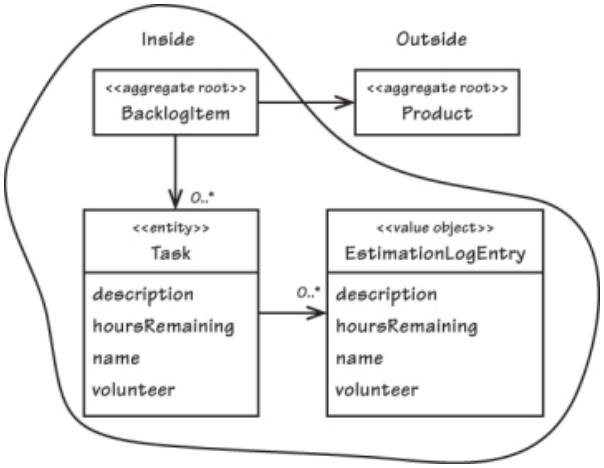


Abbildung 4.1: Aufbau eines Aggregate und seine Aggregatsgrenze
Bildquelle: [Vernon, 2013, S. 360]

4.2 Aggregate aus Sicht des Modulbegriffs

Aggregates (und Entities) als Spezialisierung des Klassenkonzepts übernehmen zunächst die grundsätzliche Moduleigenschaften von Klassen und Objekten aus Kapitel 3. Sie verfügen darüber hinaus zusätzlich über eine Identifier Variable, die als Aggregatesidentifikator im Sinne eines Moduls verstanden werden kann. Die Identität eines Aggregate bzw. Entity wird so ein echter Teil des Aggregat- und Entityobjekts selbst.

> *„Cluster the ENTITIES and VALUE OBJECTS into AGGREGATES and define boundaries around each. Choose one ENTITY to be the root of each AGGREGATE, and control access to the objects inside the boundary through the root. Allow external objects to hold references to the root only.“*
>
> [Evans, 2004, S. 129]

Das begrenzende Element (siehe Moduldefinition) ist aus konzeptueller Sicht, bspw. in Abbildung 4.1, klar als Aggregatesgrenze erkennbar. Die Aggregatesgrenze bindet hier das AggregateRoot, verschachtelte Entities und zugehörige ValueObjects. Sucht man diese Aggregatesgrenze allerdings im Code, wird eine anschauliche Definition eines begrenzenden Elements für Aggregate schwieriger. Die den Klassenrumpf einfassenden geschweiften Klammern des AggregateRoot begrenzt zwar Daten und Verhalten des AggregateRoot-Objekts selbst, nicht aber die Daten und Verhalten von verschachtelten Entities. Auch eine lexikalisch zusammenhängende Sequenz von Programmsequenzen eines Aggregates lässt sich nicht mehr einfach nachweisen, da im Aggregate gebundene Statements sich auf die Klassen der beteiligten Entities verteilen.

Aus konzeptueller Sicht kann aber argumentiert werden, dass zwischen dem AggregateRoot und verschachtelten Entities (und ValueObjects) eine Kompositionsbeziehung herrscht, da jegliche Nutzung verschachtelter Entities ohne die Existenz des AggregateRoots als Interface unmöglich wird. Die resultierende Teil-Ganzes-Beziehung macht verschachtelte Entities zum Teil des Aggregates. Aus dieser Perspektive kann argumentiert werden, dass auch die auf verschiedene Klassen verstreuten Programmstatements der Entities Teil der Aggregateklasse

werden und somit die geschweiften Klammern des Klassenrumpfs als begrenzendes Element annehmen.

Aus technischer Sicht ist eine solche Kompositions- bzw. Teil-Ganze-Beziehung und somit eine echte Aggregatsgrenze im Quellcode nicht erkennbar - sie stellt sich als architektonische Eigenschaft im Code nicht offensichtlich dar. Die eingrenzenden Eigenschaften eines Aggregates sind also sowohl für Entwickler*innen als auch für Tests nicht ohne weiteres erkenn- und überprüfbar. Es besteht deshalb die Gefahr der Erosion dieser Grenzen im Laufe des Entwicklungsprozesses. Als Resultat können die Softwarequalitäten der Wart-, Änder- und Erweiterbarkeit leiden, da sie auf durch die Grenzeigenschaft gestärkten Metriken der hohen Kohäsion und losen Kopplung beruhen.

4.3 Kohäsion von Aggregaten

Die Auswahl zusammengehöriger Entites (und deren Zusammenfassung zu Aggregates) basiert auf der Existenz von Invarianten zwischen beteiligten Entities (und ValueObjects). Eine Invariante bezeichnet hierbei Aussagen, die während der Ausführung spezifischer Operationen beständig bleiben. Das bedeutet, sie bleiben sowohl vor als auch nach der Ausführung dieser Befehle wahr und verändern sich somit nicht - sie ist invariant. Die Einhaltung der Invarianten erzeugt eine *strikte Konsistenz* zwischen den Entitäten, die typischerweise durch die Verwendung von Transaktionen umgesetzt und erzeugt wird.

Das Vorhandensein von Invarianten deutet darüber hinaus auf ein funktionales Verwandtschaftsverhältnis zwischen betroffenen Entities hin. Diese funktionale Verwandtschaft bewegt sich dabei mindestens auf dem Level einer kommunikativen oder sequentiellen Assoziation. Die Bindung in einem Aggregats(modul) erzeugt also gleichzeitig (mindestens) kommunikative oder sequentielle Kohäsion. Beides sind, wie bereits in Kapitel 2.3 angemerkt, die zweit- und drittstärksten Problemraum-bedingten Kohäsionstypen. Eine Auswahl von Entities (und ValueObjects) auf Basis von Invarianten sorgt also für eine hohe Kohäsion des Aggregate(moduls).

4.4 Kopplung zwischen Aggregaten

Das Aggregatskonzept und zugehörige Designregeln sorgen gleichzeitig für eine lose Kopplung zwischen Aggregaten. Die Auswirkungen der einzelnen Designregeln auf die Kopplung zwischen Aggregaten wird im Folgenden näher untersucht.

4.4.1 Eventuelle Konsistenz zwischen Aggregaten

Das im vorigen Abschnitt beschriebene Vorgehen zur Bestimmung von Aggregatsgrenzen sorgt gleichzeitig dafür, dass zwischen Entities in zwei verschiedenen Aggregaten **keine** Invarianten bestehen. Die Entities der beiden Aggregate können also unabhängig voneinander geändert werden. Die Notwendigkeit einer transaktionalen Änderung und die Einhaltung einer strikten Konsistenz besteht hier also nicht. Dies bedeutet gleichzeitig, dass Connascence of Value, die zweitstärkste Form der Connascence bzw. Kopplung, ausgeschlossen ist.

DDD gibt desweiteren vor, dass Operationen die mehrere Aggregate betreffen, nur mit eventueller Konsistenz ausgeführt werden. Siehe hierzu folgendes Zitat:

> *„Any rule that spans AGGREGATES will not be expected to be up-to-date at all times. Through event processing, batch processing, or other update mechanisms, other dependencies can be resolved within some specific time.“*
>
> [Evans, 2004, S. 128]

Eventuelle Konsistenz meint hierbei, dass Daten von beteiligten Aggregaten für eine gewisse Zeit inkonsistent sein können. Eventuelle Konsistenz stellt also nur sicher, dass die Daten konsistent werden, nicht aber wann.

Die Berücksichtigung dieser Regel schließt somit auch Connascence of Timing, die drittstärkste Form der Connascence, zwischen den Aggregaten aus.

Zusammengenommen schließt das Nichtvorhandensein von Invarianten und die eventuelle Konsistenz zwischen Aggregaten also zwei der drei stärksten Formen der Connascence aus. Eine lose Kopplung zwischen Aggregaten wird so gefördert.

4.4.2 Assoziation von Aggregaten per Id-Objekt

[Vernon, 2013, S.359ff] empfiehlt des Weiteren auf eine direkte Objektreferenz zwischen Aggregaten, wie sie in Listing 4.1 (Zeile 4) zu sehen ist, zu verzichten. Er begründet dies mit der Gefahr, dass durch Nutzung dieser Objektreferenzen die Konsistenz- und Aggregatesgrenzen der Aggregate verletzt werden könnten.

```
1  public class BacklogItem{
2
3      private UUID id;
4      private Product product;
5      ...
6  }
7  ------
8  public class Product{
9
10     private UUID id;
11 }
```

Listing 4.1: Assoziation zwischen den Aggregaten BacklogItem und Produkt per Objektreferenz (Zeile 4)

Als Alternative schlägt [Vernon, 2013] den Aufbau eines Aggregatsgraphs mittels globalen Identifizierungsobjekten (Id-Objekte) vor, die die Identitäten der beteiligten Aggregate(Roots) abbilden, siehe Listing 4.2 (Zeile 4). Diese Id-Objekte werden hierbei in Zeile 8 in Form von ValueObjects umgesetzt und kapseln eine global eindeutige UUID.

Zum Aufbau des Aggregatsgraphs wird das Id-Objekt (ProductId) des fremden Aggregats (Product) kopiert und im eigenen Aggregat abgespeichert (Zeile 4). Die Id-Objekte von fremden Aggregaten werden also Teil des Aggregats und liegen innerhalb der eigenen Aggregatsgrenzen, siehe Abbildung 4.2.

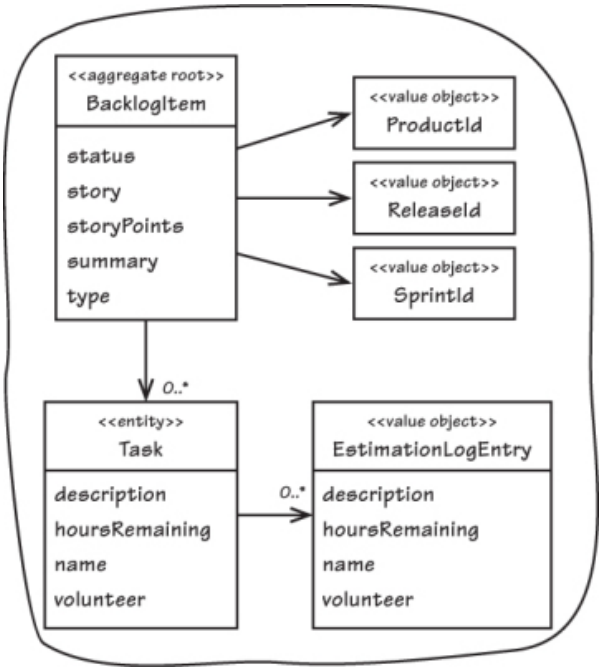


Abbildung 4.2: Nutzung von Identifier-ValueObjects zu Assoziation mit anderen Aggregaten
Bildquelle: [Vernon, 2013, S. 361]

Der Kopie des Id-Objekte (Zeile 4) stellt eine assoziative Beziehung zwischen den beiden Aggregaten her, eine echte referenzielle Beziehung zwischen den Aggregaten entfällt. Eine direkte Nutzung der Daten oder Methoden mittels direkter Referenz auf fremde Aggregate ist so nicht mehr möglich ist. Hierüber wird sichergestellt, dass Änderungen (mit strikter Konsistenz) zunächst nur am betroffenen Aggregate vorgenommen werden können. Eine mögliche Aufweitung von Transaktionen auf weitere Aggregate und die Erzeugung strikter Konsistenz zwischen Aggregaten wird so effektiv verhindert.

```
1  public class BacklogItem{
2
3      private BacklogItemId id;
4      private ProductId productId;
5      ...
6  }
7  -----
8  public record ProductId(UUID id){
9  }
10 public class Product extends Entity {
11
12     private ProductId id;
13     ...
14 }
```

Listing 4.2: Assoziation zwischen den Aggregaten BacklogItem und Produkt per Id-Referenz (Zeile 4)

Die Nutzung von Id-Objekten verringert hierdurch zusätzlich die Gefahr einer möglichen Erosion der losen Kopplung durch Entwickler*innen im Softwarelebenszyklus. Eine unbeabsichtigte Wiedereinführung von Connascence of Value und Connascence of Time zwischen den Aggregaten wird so unwahrscheinlicher.

Durch den Wegfall einer echten Objektreferenz ist gleichzeitig auch Connascence of Identity, die stärkste Form der Connascence ausgeschlossen.

Zusammengenommen werden durch Einhaltung der oben beschriebenen Regeln also die drei stärksten Formen der Connascence vermieden. Gleichzeitig ist eine inhaltliche Kopplung (dem stärksten Kopplungstypen nach [Myers, 1975]) ausgeschlossen, da ein Zugriff auf die Inhalte

fremder Aggregate verhindert wird. Eine Modularisierung der Domäne und Software in Aggregate und die Einhaltung der entsprechenden Designregeln des DDD machen also eine lose Kopplung zwischen Modulen sehr wahrscheinlich.

4.4.3 Kommunikation zwischen Aggregaten

Die im vorigen Abschnitt beschriebene Assoziation von Aggregaten mittel Id-Objekten macht eine direkte Kommunikation zwischen Aggregaten mittel Methodenaufruf unmöglich. Weil aber Änderungen an einzelnen Aggregaten unter Umständen trotzdem an andere Aggregate kommuniziert werden müssen, führt das taktische Design den Building Block des DomainEvents (*Domänenereignis*) ein.

Das Konzept der DomainEvents setzt hierbei das Observer- bzw. Publish-Subscribe- Entwurfsmuster nach [Gamma et al., 1994, S. 293] um.

> *„All a subject knows is that it has a list of observers, each conforming to the simple interface of the abstract Observer class. The subject doesn't know the concrete class of any observer. Thus the coupling between subjects and observers is abstract und minimal.“*
>
> aus [Gamma et al., 1994, S.296]

DomainEvents beschreiben und veröffentlichen (*publishen*) hierbei Änderungen an einem Aggregate A und machen diese Informationen für andere Aggregate verfügbar. Ein anderes Aggregate B kann (muss aber nicht) sein Interesse an Änderungen des Aggregate A anmelden und das DomainEvent abonnieren (*subscriben*).

DomainEvents verhindern so die in Kapitel 2.4 beschriebene aktivierende Kontrollkopplung, da eine aktive Kontrollübergabe des Aggregate A nicht mehr stattfindet. Es entsteht die geringere Form der Datenkopplung.

Aggregate B kann dabei selbst entscheiden, ob und wann es auf Änderungen an Aggregate A reagiert. Die in Abschnitt 4.4.1 geforderte eventuelle Konsistenz zwischen Aggregaten bleibt gewahrt und Connascence of Timing zwischen den Aggregaten wird so vermieden. Müssen zusätzlich mehrere Aggregate (B und C) auf Änderungen von Aggregate A reagieren, findet hierdurch auch eine weitere Entkopplung zwischen den Aggregaten B und C statt. Da Aggregate B und C selbst entscheiden können wann sie auf Änderungen von Aggregate A reagieren, wird ist die Reihenfolge ihrer Reaktion unbekannt. Die beiden Aggregate vermeiden so Connascence of Execution.

Gleichzeitig greift eine Kommunikation mittels DomainEvents den in Kapitel 3 beschriebenen, vergessen geschienenen, Ansatz der OOP zur Kommunikation zwischen Objekten mittels Messages auf. Beide Konzepte (Events und Messages) sind hierbei nahezu synonym, da beide eine eher unidirektionale Kommunikation modellieren und Rückgabewerte bzw. Antworten zunächst ausgeschlossen sind. Wie bereits in Abschnitt 3.2 erwähnt wird hierdurch insbesondere die Gefahr einer starken inhaltlichen Kopplung minimiert.

Auch das Design der DomainEvents an sich kann zur Reduktion der Connascence zwischen den Aggregaten beitragen. Ein Beispiel für ein typisches DomainEvent findet sich in Listing 4.3.

```
1 public record RoundStarted(
2         GameId gameId,
3         RoundId roundId,
4         Integer number) implements DomainEvent {}
```

Listing 4.3: Codebeispiel eines DomainEvents

Durch den Verzicht der Nutzung des eigentlichen Aggregatstyps Round und Beschreibung der Änderungen mittels Id-Objekts (Zeile 2 und 3) und primitiven Datentypen (Zeile 4), wird die Erzeugung von Connascence of Type or Class verhindert. Stattdessen entsteht Connascence of Name, die leichteste Form der Connascence, zwischen den Aggregaten. Die Kopplung der Aggregate wird so weiter minimiert.

4.5 Gefahr der Erosion architektonischer Moduleigenschaften

Die Einhaltung der in den Abschnitten 4.3 - 4.4 vorgestellten Entwurfsmuster und -regeln unterstützen, wie besprochen, die Modularisierung der Domäne (und der Software). Resultierende Module (in Form von Aggregaten) sind zunächst hoch kohäsiv und gleichzeitig lose gekoppelt. Dadurch werden die Softwarequalitäten der Wart-, Änder- und Erweiterbarkeit der Module im Speziellen und der Gesamtsoftware im Allgemeinen maximiert.

Weiterhin problematisch ist allerdings die Tatsache, dass die entsprechenden Muster und Regeln (und zugrundeliegende Konzepte der Modularisierung) von allen beteiligten Entwickler*innen im Detail verstanden und im Entwicklungsalltag strikt befolgt werden müssen, um die gewünschten Softwarequalitäten zu erzeugen und zu erhalten. Es besteht weiterhin die Gefahr, dass Entwickler*innen einige der Regeln ignorieren und so gewonnene architektonische Softwarequalitäten erodieren.

Ein Beispiel für eine solche mögliche Erosion findet sich in der Umgehung der in Abschnitt 4.4.2 beschriebenen Regel, dass eine Assoziation von Aggregaten nur über Id-Objekte geschehen sollte:

Entwickler*innen könnten hierfür bspw. das Repository eines fremden Aggregates injecten, siehe Listing 4.4 (Zeilen 5-6). In einem zweiten Schritt kann dann das Repository genutzt werden, um eine echte Objektreferenz auf das fremde Aggregat erhalten (Zeile 9) und schlussendlich, zur Umsetzung einer neuen Funktionalität, genutzt werden (Zeilen 8-12).

```
1  public class BacklogItem{
2
3      private BacklogItemId id;
4      private ProductId productId;
5      @Autowired
6      private ProductRepository productRepository;
7
8      private void newFuctionality(){
9          private Product product = productRepository.findById();
10         product.doSomething();
```

```
11              ...
12          }
13          ...
14  }
```

Listing 4.4: Mögliche Erosion der *Assoziation per Id-Objekt-Regel*

Die Realität zeigt, dass Entwickler*innen hierzu, insbesondere unter Zeitdruck, leicht verleitet sind. Sie bauen so technische Schuld auf und mindern bereits erreichte oder gewollte Softwarequalitäten.

Um diese oder ähnliche Umgehungen zu verhindern, wäre die Möglichkeit der automatisierte Überprüfung der architektonischen Designmuster und -regeln durch den Compiler oder Tests wünschenswert. Eine solche Überprüfung ist allerdings bisher nicht möglich, da im Quellcode nötige Informationen über die architektonischen Konzepte fehlen:

- Den Klassen ist nicht ohne Weiteres anzusehen, ob es sich um ein AggregatRoot, eine Entity oder ValueObject handelt.
- Die Zugehörigkeit von verschachtelten Entities zu Aggregaten ist nicht zu erkennen.
- Bei Referenzen ist nicht erkennbar, ob sie Aggregatsgrenzen verletzen.
- Klare Aggregatsgrenzen bleiben dem Compiler verborgen.
- etc.

Einige dieser Informationen können zwar mit programmiersprachlichen Mitteln (bspw. Zugriffsoperatoren) angenähert werden, eine komplette Codierung der nötigen Informationen ist aber nicht möglich. Eine Erosion kann so nur durch Vertrauen in die Entwickler*innen und/oder durch ständige Kontrollen der Architekt*in verhindert werden.

4.6 Modulbegriff im DDD

Obwohl das Konzept der Aggregate, wie Abschnitt 4.2 bewiesen, über modulbildende Eigenschaften verfügt, werden sie in den gängigen Quellen nicht als Module bezeichnet.

Die Originalquellen kennen allerdings trotzdem das Modulkonzept an sich und verstehen Module als (Java) Pakete, vgl. [Evans, 2004, S.109ff] und [Vernon, 2013, S.333ff].

Sie empfehlen hierbei die Nutzung der Paketstrukturen, um Konzepte der Domäne von sonstigen, technischen und infrastrukturellen Objekten und Klassen zu separieren:

> *„Use packaging to separate domain layer from other code. Otherwise, leave as much freedom as possible to the domain developers to package the domain objects in ways that support their model and design choices.“*
>
> [Evans, 2004, S.115]

Weitere Empfehlungen wie und wie groß Pakete aus Domänensicht geschnitten werden sollten, finden sich in den Originalquellen nicht.

5 Der Modulith-Architekturstil am Beispiel von Spring Modulith

Wie bereits in der Einleitung der vorliegenden Arbeit angesprochen, handelt es sich bei dem Modulith-Architekturstil um keine grundsätzlich neue Entwicklung. Es handelt sich bei Modulithen vielmehr um einen durch Modularisierung stark strukturierten Deployment-Monolithen.

Der Architekturstil ist deshalb allgemein mit verschiedenen anderen, bereits älteren, Frameworks umsetzbar:

- **OSGi Spezifikation**
  Die OSGi Spezifikation aus dem Jahr 2000 definiert ein dynamisches Modulsystem für die Java Virtual Machine (JVM). Einzelne Module (genannt Bundles) des Gesamtsystems können zur Laufzeit geladen werden und stellen so einzelne Services bereit. Neben dem eigentlichen Java-Quellcode und sonstigen Ressourcen verfügt hierbei jedes Bundle über ein Manifest, das zum einen zum Betrieb nötige Abhängigkeiten und zum anderen die angebotene Serviceschnittstelle in Form von exportierten Java-Paketen definiert.

  Der Softwareconsultant ThoughtWorks rät allerdings schon seit längerem von der Verwendung der OSGi Spezifikation ab, da sie durch zusätzliche Abstraktionen ungewollte, zufällige Komplexität erzeugt, vgl. [Thoughtworks, 2015].

- **Jigsaw Modulsystem**
  Beim Jigsaw Modulsystem handelt es sich um eine mit Java 9 veröffentlichte Spracher-

weiterung, die hauptsächlich für die Modularisierung der Java-Plattform selbst gedacht ist. Ein Jigsaw-Modul ist hierbei eine Gruppe von Java-Packages, die zu einer sogenannten *Modular JAR*-Datei paketiert werden. In einem Modul-Deskriptor werden Modulname, Abhängigkeiten zu anderen Modulen und die Sichtbarkeit der beinhalteten Pakete definiert, vgl. [Grammes et al., 2018].

Da das Jigsaw Modulsystem eher für die Modularisierung von Bibliotheken entwickelt wurde, wird heute oft von der Nutzung zur Entwicklung von Anwendungssystemen abgeraten.

Mit dem Framework Spring Modulith[1] wurde im Jahr 2023 eine neue Möglichkeit zur Entwicklung von Modulith-Architekturen veröffentlicht. Das Framework kombiniert dabei das in Kapitel 2.1 vorgestellte Konzept der Modularisierung mit den Konzepten des DDD (Kapitel 4). Es verfolgt dabei den Ansatz architektonische, konzeptuelle Strukturen im Quellcode sichtbar zu machen und stellt Hilfsmittel bereit die architektonische Treue des Quellcodes zu überprüfen.

Einige ausgewählte Konzepte, die diesen Ansatz umsetzen, werden in den folgenden Kapiteln vorgestellt.

5.1 Sichtbarkeit architektonischer Konzepte

Das Spring Modulith Framework bindet zunächst (optional) die Bibliothek jMolecules[2] ein. Die Bibliothek bietet verschiedene abstrakte Typen, Interfaces und Annotationen an, die Konzepte des DDD und anderer Architekturstile abbilden und implementiert deren grundlegenden Eigenschaften. jMolecules erlaubt es Entwickler*innen und Architekt*innen dadurch, bspw. durch Ableitung von diesen Typen, architektonische Konzepte im Quellcode sichtbar und evident zu machen.

[1] https://spring.io/projects/spring-modulith/
[2] https://github.com/xmolecules/jmolecules

[Drotbohm et al., 2022] argumentieren, dass eine typische Umsetzung von architektonischen Konzepten als POJO-Klassen (*Plain Old Java Objekt*-Klassen) einen zusätzlichen mentalen Transformationsschritt nötig macht, um Strukturen im Quellcode auf architektonische Konzepte abbilden zu können.

Ein Beispiel hierfür findet sich in Listing 5.1:
Die *Entity*-Annotation in Zeile 1 und die Id-Variable in Zeile 4, *könnten* Entwickler*innen den Eindruck vermitteln, dass es sich hierbei um eine Entity des DDD handelt. Dieser Eindruck ist allerdings trügerisch, da es sich bei der Entity Annotation um eine Annotation aus der Jakarta Persistenz API handelt - sie beschreibt also vielmehr ein Konzept aus dem Bereich von relationalen Datenbanken. Dass es sich hierbei eigentlich um eine DDD Entity oder sogar genauer um ein AggregateRoot handelt, ist aus dem Quellcode nicht ersichtlich und muss im Zweifelsfall aufwendig in der Dokumentation der geplanten domänengetriebenen Architektur nachgeschaut werden.

```
1  @Entity
2  public class Robot {
3
4      private final RobotId id;
5
6      private final GameId gameId;
7      private final Playerid playerId;
8
9      private final PlanetId planetid;
10
11     protected Robot(GameId gameId, PlayerId playerId, PlanetId
       planetId) {
12         ...
13     }
14 }
```

Listing 5.1: Beispiel eines als POJO umgesetzten Aggregates

Leitet man stattdessen die eigenen Domänenklassen von den von jMolecules zur Verfügung gestellten Architekturklassen ab, wird die Verknüpfung zwischen Code und Architektur- bzw. Domänenmodell schnell deutlich, siehe Listing 5.2. Entwickler*innen erkennen hier recht schnell, dass es sich bei der Robot Klasse um ein AggregateRoot handeln muss, eine zu-

sätzliche mentale Transformation ist nicht mehr nötig.

```
1 public class Robot extends AbstractAggregateRoot<Robot>
    implements AggregateRoot<Robot, RobotId> {}
```

Listing 5.2: Beispiel eines als POJO umgesetzten Aggregates

Auch andere architektonische Beziehungen, die sich im Quellcode sonst nicht widerspiegeln, können so deutlich gemacht werden. So ist bei der Entity *Account* in Listing 5.3 (Zeile 2) klar, dass es sich um ein verschachteltes Entity handelt, das dem Aggregat *Shop* zugehörig ist und von ihm gebunden wird. Die in Abschnitt 4.2 besprochenen konzeptuellen Aggregatsgrenzen des DDD werden im Code sichtbar. Sie könnten so bspw. ein erstes Zeichen für Entwickler*innen sein, dass Referenzen auf die Account-Entity von außerhalb des Aggregats vermieden werden sollten.

```
1 public class Shop extends AbstractAggregateRoot<Shop>
    implements AggregateRoot<Shop, ShopId> {...}
2 public class Account implements Entity<Shop, AccountId> {...}
```

Listing 5.3: Beispiel einer geschachtelten Entity und zugehöriges AggregateRoot

Die kombinierte Nutzung von Spring Modulith und jMolecules rückt so die Strukturelemente des Lösungsraums näher an Strukturen im Problemraum, der mittels DDD dekomponiert wurde und folgt damit den Empfehlungen des Structured Design, vgl. Kapitel 2.1.

jMolecules erlaubt zusätzlich eine sauberere Trennung zwischen Domain- und Infrastrukturschicht herzustellen. Die zusätzliche jMolecules Integrations Bibliothek[3] erkennt hierbei die genutzten Architekturkonzepte und reichert sie mithilfe von ByteBuddy[4] in einem Buildschritt um nötige Persistenz Annotationen an. Eine sonst nötige Annotation von relationalen Konzepten (*@Entity*, *@OneToMany*, *@Embeddable*, *@ElementCollection* und Ähnliches) entfällt so in den meisten Fällen. Klassen der Domänenschicht wirken so deutlich aufgeräumter und Entwickler*innen können sich besser auf die eigentliche Funktionalität der Domänenklasse fokussieren.

[3] https://github.com/xmolecules/jmolecules-integrations
[4] https://bytebuddy.net

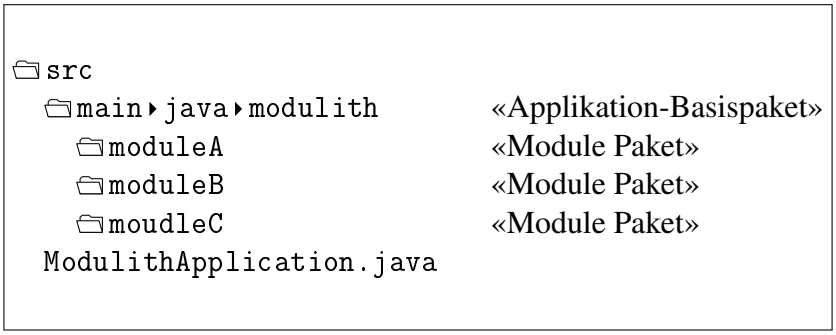


Abbildung 5.1: Paketstruktur von Spring Modulith

5.2 Modulstruktur

Zur Erzeugung von Modulen nutzt Spring Modulith die Java Paketstruktur, vgl. [Spring, 2023]. Das Framework folgt dabei dem Vorschlag des DDD und empfiehlt eine primär fachliche Aufteilung der Module und Pakete, vgl. Abschnitt 4.6. Eine technisch getriebene Trennung der Pakete, bspw. der Schichtenarchitektur folgend, entfällt entweder komplett oder wird in interne Subpakete verschoben.

Jedes Unterpaket des Applikation-Basispakets erzeugt dabei ein eigenes fachliches Modul, vgl. Abbildung 5.1. Modulgrenzen werden so durch die Paketstruktur explizit und werden sowohl für Entwickler*innen als auch für den Compiler erkennbar.

Es scheint hierbei sinnvoll, dass jedes Aggregate der Domäne auch sein eigenes Modul erzeugt[5]. Die technischen Paket- und Modulgrenzen und die konzeptuellen Aggregatsgrenzen werden so deckungsgleich und bilden einander ab. Dieser Vorgehensweise kann im weiteren Verlauf genutzt werden, um sowohl technische als auch konzeptuelle Grenzen gleichzeitig zu testen.

Eine Ebene tiefer teilt Spring Modulith die Module in einen öffentlichen und eine privaten, internen Bereich auf, siehe Abbildung 5.2. Das öffentliche Interface des Moduls bilden alle

[5] Spring Modulith gibt dieses Vorgehen nicht vor und erlaubt grundsätzlich auch größere Module, die mehrere Aggregate umfassen.

Klassen, die im Modulpaket selbst liegen und manuell (per *package-info*-Klasse) veröffentlichte Subpakete.

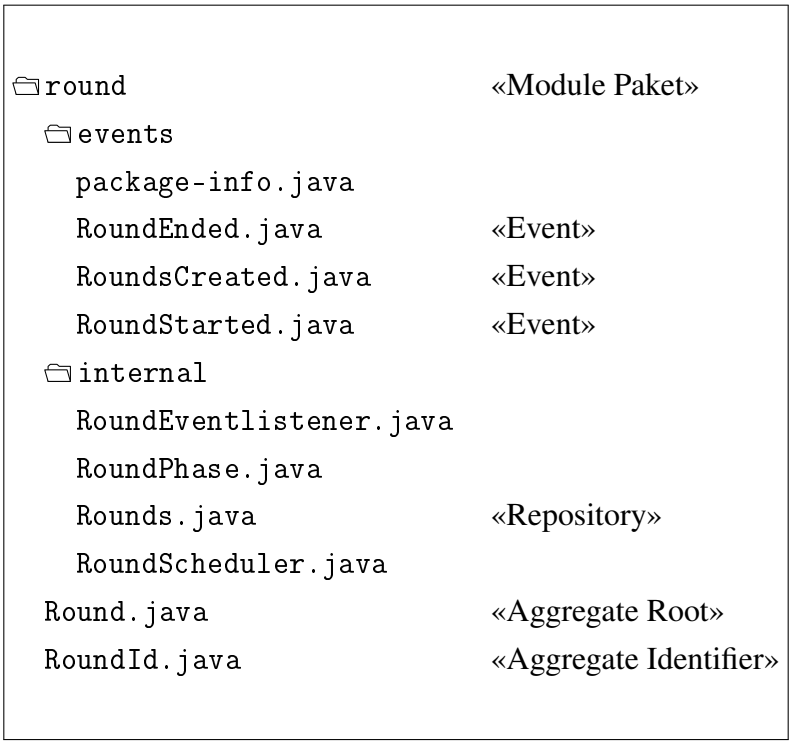


Abbildung 5.2: Interne Struktur eines Moduls

Im vorliegenden Fall wird das Interface also aus dem Aggregate *Round*, dessen Identifier *RoundId* und aus diversen Events im *events*-Paket gebildet. Sonstige Subpakete werden als Interna des Moduls gewertet und so vor dem Zugriff von außen geschützt.

Insbesondere das Repository des Aggregates im Subpaket *internal* steht somit nicht mehr als Abhängigkeit für andere Module bereit. Eine Umgehung der DDD-Regel zur Assoziation von Aggregaten per Id-Referenz, wie sie in Abschnitt 4.5 beschrieben ist, wird so verhindert. Die Gefahr der Erosion architektonischer Eigenschaften und resultierender Softwarequalitäten wird so gemindert.

5.3 Architektonische Tests

Zur Überprüfung von technischen Paketmodulgrenzen und konzeptuellen Aggregatesgrenzen bietet Spring Modulith die Implementierung eines einfachen SpringApplicationTests an, siehe Listing 5.4. Spring Modulith nutzt hierbei die Bibliothek ArchUnit[6], welche die Durchführung von architektonischen (nicht funktionalen) Tests anhand von architektonischen Regeln erlaubt.

```
1 @SpringBootTest
2 class AmundsenApplicationTests {
3     @Test
4     void verifyApplicationModuleStructure() {
5         ApplicationModules.of(AmundsenApplication.class).verify
    ();
6     }
7 }
```

Listing 5.4: Architektonischer Test von Modul- und Architektureigenschaften

Dieser architektonische Test überprüft nach [Spring.io, 2023b] zunächst, die in Kapitel 2 besprochenen, grundlegenden Moduleigenschaften:

- **Sind zirkuläre Abhängigkeiten zwischen Modulen vorhanden?**
  Eine zirkuläre Abhängigkeit entsteht entweder direkt als wechselseitige Abhängigkeit zwischen zwei Modulen oder indirekt (über die reflexiv-transitive Hülle) als Zirkel im Abhängigkeitsgraph. Zirkuläre Abhängigkeiten stellen die stärkste Form der Kopplung zweier Module dar und sind ein Anzeichen dafür, dass die Modulgrenzen falsch gezogen wurden.

- **Besitzen Module Abhängigkeiten auf interne Klassen des fremden Moduls vorhanden?**
  Eine solche Abhängigkeit erzeugt eine pathologische Kopplung zwischen den Modulen und verletzt die Modulgrenze. Sie kann als Anzeichen für eine inhaltliche Kopplung (stärkster Kopplungstyp) gewertet werden.

[6] https://www.archunit.org/

Bei Nutzung der jMolecules Bibliothek werden zusätzlich Architekturstil spezifische Regelsätze (in diesem Fall DDD) überprüft:

- **Besitzen alle Entities und Aggregates einen Identifier?**
  Zunächst wird grundlegend getestet, ob alle Entities und Aggregate über eine Identität verfügen, siehe Kapitel 4.
- **Sind alle Entities einem Aggregate zugeordnet?**
  Nicht zugeordnete Entities würden selbst ein Aggregate bilden und müssten entsprechend vom Typ Aggregate ableiten.
- **Sind Aggregate per Id assoziiert?**
  Überprüft die Einhaltung des in Abschnitt 4.4.2 beschriebene Pattern der Assoziation von Aggregaten per Id-Objekt.
- **Besitzen ValueObjects Referenzen auf Aggregate?**
  Eine Referenz auf ein Aggregate innerhalb eines ValueObjekts würde die Immutable-Eigenschaft des ValueObjekts zerstören, da eine Änderung am referenzierten Aggregate aus konzeptueller Sicht einer Änderung am ValueObject selbst gleichkommt.

Zusammengenommen überprüft der Test die in Kapitel 2 und 4 beschriebenen konzeptuellen, architektonischen Eigenschaften und Pattern des Structured Design und DDD. Bei der Entwicklung kann so zunächst die architektonische Treue des Quellcodes überprüft werden und im Verlauf eine Erosion derselbigen effektiv verhindert werden.

5.4 Kommunikation zwischen Modulen

Zur Umsetzung der in Abschnitt 4.4.3 besprochenen eventbasierten Kommunikation zwischen Aggregaten nutzt Spring Modulith die, aus dem Spring Framework bekannten, ApplicationEvents.

Zunächst problematisch daran ist allerdings nach [Spring.io, 2023c], dass ApplicationEvents im Normalfall, zum Zeitpunkt des Eventversands bestehende, Transaktionen auf empfangende EventListener aufweitet. Folgeänderungen an benachrichtigten Aggregaten sind so nur mit strikter Konsistenz zur ursprünglichen Änderung möglich. Die, in Abschnitt 4.4.1 geforderte, eventuelle Konsistenz wird somit nicht erfüllt.

Aus diesem Grund stellt Spring Modulith mit der Annotation *@ApplicationModuleListener* eine angepasste Form des EventListeners zur Verfügung, siehe Listing 5.5.

```
@Async
@Transactional(
    propagation = Propagation.REQUIRES_NEW
)
@TransactionalEventListener
...
public @interface ApplicationModuleListener {
    ...
    @AliasFor(
        annotation = EventListener.class,
        ...
    )
    String id() default "";
    ...
}
```

Listing 5.5: Architektonischer Test von Modul- und Architektureigenschaften

Die Annotation *ApplicationModuleListener* kombiniert also die folgenden Annotationen:

- ***@Transactional(propagation = Propagation.REQUIRES_NEW)***
  Erzeugt eine vom Kontrollfluss des Event erzeugenden Moduls unabhängige, neue Transaktion für die Eventbehandlung.

  Die unabhängigen Transaktionen machen ***eventuell konsistente*** Änderungen an den beteiligten Aggregaten möglich.

- ***@TransactionalEventListener***
  Dies stellt sicher, dass die isolierte Änderung am Aggregat des empfangenden Moduls wiederum mit ***strikter Konsistenz*** durchgeführt wird, um dessen Invarianten zu wahren.

- ***@Async***
  Markiert die annotierte Methode für eine asynchrone Ausführung.

  Die Methode wird also in einem neuen Thread ausgeführt und die Event erzeugende Methode wartet **nicht** auf die Beendigung der Eventbehandlung. Die Annotation erzeugt die ***eventuelle Konsistenz*** zwischen Event erzeugendem und Event konsumierendem Modul. Die beteiligten Module werden ***temporal entkoppelt***.

5.5 Verbesserte Testbarkeit

Die oben besprochene Umsetzungen der in DDD erarbeiteten Pattern sorgt neben einer hohen Wart-, Änder- und Erweiterbarkeit des Module und des Gesamtsystems auch zur Verbesserung der Testbarkeit von Modulen. Da die so entwickelten Module maximal entkoppelt sind, besitzen sie nur wenige Abhängigkeiten zu anderen Modulen. Die übriggebliebenen Abhängigkeiten beschränken sich auf Assoziationen per Id-Objekt, vgl. Abschnitt 4.4.2. Diese können in Tests leicht durch Erzeugung von zufälligen ID-Objekten gemockt werden.

Diesen Effekt macht sich Spring Modulith zu nutze und bietet die Annotation *@ApplicationModuleTest* an, siehe [Spring.io, 2023a] Die so annotierten Testfälle laden ausschließlich das zu testende Modul in den TestContext. Die Laufzeit der Tests wird hierdurch signifikant verbessert und die Testbarkeit des Systems steigt.

6 Fazit

Die vorliegende Arbeit endet mit der abschließenden Beantwortung der Forschungsfragen, einer Diskussion der Ergebnisse und einem Ausblick auf mögliche zukünftige Forschungsarbeiten im Bereich der besprochenen Thematiken. Den Abschluss bilden einige persönliche Bemerkungen des Autors über gemachte Erfahrungen in der Auseinandersetzung mit den besprochenen Themen und in der Einarbeitung in den modulithischen Architekturstil.

6.1 Forschungsfragen

Die Forschungsfragen werden im Folgenden noch einmal zusammenfassend beantwortet:

Welche Arten von Abhängigkeiten gibt es und wie entstehen sie?
Abhängigkeiten entstehen grundsätzlich durch die funktionale Verwandtschaft von Quellcodeelementen auf der Ebene einzelner, atomarer Programmstatements. Die Stärke dieser funktionalen Verwandtschaft ergibt sich aus Art der Assoziation, die zwischen zwei Elementen besteht.

Kommunikative, sequentielle und funktionale Assoziationen entstehen im Problemraum (in der Domäne) selbst und erzeugen eine starke funktionale Verwandtschaft der Elemente. Problemraum bedingte Assoziationen sind unvermeidbar und nötig für die Funktionalität des Gesamtsystems. Eine Beeinflussung der Art und Stärke von Problemraum-bedingten Assoziationen durch technische Entscheidungen im Lösungsraum (im Quellcode der Software selbst) ist ausgeschlossen.

Zufällige, logische und temporale Assoziationen entstehen erst durch Entscheidungen von Entwickler*innen und Architekt*innen im Lösungsraum (im Quellcode der Software) und erzeugen eine eher schwache funktionale Verwandtschaft. Die Art und Stärke von Lösungsraum bedingten Assoziationen können durch die geschickte Ziehung von Modulgrenzen beeinflusst und gemindert werden oder ihre Erzeugung nach Möglichkeit sogar ganz vermieden werden.

Die prozedurale Assoziation zwischen Elementen kann nicht eindeutig dem Problem- oder Lösungsraum zugeordnet werden, erzeugt eine mittelstarke funktionale Verwandtschaft und wird im Normalfall geduldet.

Das Konzept der Modularisierung gruppiert und bindet Mengen dieser Elemente in verschiedenen Problem- und Quellcodestrukturen, genannt Module. Das Konzept von Modulen findet sich hierbei auf den unterschiedlichsten Abstraktionsebenen der Softwarearchitektur, bspw. in Form von Prozeduren, Klassen und Objekten, Komponenten, Paketen und Aggregates.

Die Summe der funktionalen Verwandtschaft zwischen Elementen innerhalb von Modulen wird als *Kohäsion* bezeichnet. Die Summe der funktionalen Verwandtschaft zwischen Elementen, die sich in unterschiedlichen Modulen befinden, beschreibt die *Kopplung* zwischen den Modulen.

Abhängigkeiten bestehen aus konzeptueller Sicht demzufolge, anders als im heute üblichen Sprachgebrauch, sowohl innerhalb von Modulen (intramodular) als auch zwischen Modulen (intermodular).

Im objektorientierten Umfeld können die verschiedenen Abhängigkeitsarten auch nach dem Konzept der Connascence kategorisiert und eingeschätzt werden. Connascence vereint die Begriffe der Kohäsion und Kopplung und beschreibt hierbei allgemein die Notwendigkeit der gemeinsamen Änderung Modulen.

Welchen Einfluss haben die verschiedenen Abhängigkeitsarten auf die Kohäsion innerhalb und den Kopplungsgrad zwischen Modulen?
Die Art der Abhängigkeit bestimmt darüber wie viel und welches Wissen Entwickler*innen

bei der Änderung eines Moduls über andere abhängige Module haben müssen, um die Korrektheit des Systems sicherstellen zu können. Entwickler*innen können dabei natürlicherweise nur eine begrenzte Menge an Wissen gleichzeitig gegenwärtig haben. Übersteigt das nötige Wissen über andere Elemente diese Grenze, erschwert dies den Änderungsprozess, da fortlaufend der Quellcode anderer Module zurate gezogen und analysiert werden muss.

Um das nötige Wissen lokal im Modul zu sammeln und zu minimieren, sollten Module auf Basis der größtmöglichen funktionalen Verwandtschaft gebildet werden. Die Auswahl der Quellcodeelemente sollte dabei ausschließlich Problemraum-bedingte Assoziations- und Abhängigkeitsarten berücksichtigen. Bei der Modulbildung nicht berücksichtigte Abhängigkeitsarten beschränken sich (im Idealfall) auf schwächere, Lösungsraum bedingte Assoziationen. Die Kohäsion der Module wird so maximiert und die Softwarequalitäten der Wart-, Änder- und Erweiterbarkeit steigen, indem nötiges Wissen in einem Modul lokalisiert wird und sowohl für Entwickler*innen als auch für die Funktionalität der Module gut erreichbar ist.

Gleichzeitig entsteht so auch eine losere Kopplung der Module. Die Kopplung kann weiter minimiert werden, indem Assoziationen und Kommunikationspattern zwischen Modulen überdacht werden. Hierzu sollten insbesondere klassische, eingeübte Muster der objektorientierten Programmierung auf den Prüfstand gestellt werden.

Eine in der objektorientierten Programmierung übliche Bildung von Assoziationen per Objektreferenz (über Modulgrenzen hinweg) kann hierbei nämlich bereits Schaden anrichten und die Kopplung der Objekte (und Module) ungewollt stärken. Eine Neubewertung der ursprünglichen Definition der objektorientierten Programmierung, insbesondere der messagegetriebenen Kommunikation von Objekten, kann dabei helfen Module weiter zu entkoppeln.

Gelingt eine starke Entkopplung der Module, stärkt dies die Autonomie der Module zur Laufzeit und ermöglicht so eine spätere Dezentralisierung der Software (hin zu einer Microservice Architektur) oder eine horizontale Skalierung einzelner Services.

Module werden (im Idealfall) unabhängig voneinander prüf- und testbar - die Korrektheit ihrer Funktionalität im mathematischen Sinne beweisbar. Die Softwarequalität der Testbarkeit

steigt so dramatisch, da auf ein Mocking von Abhängigkeiten verzichtet werden kann. Einige Integrationstests können so durch Unittests ersetzt werden.

Welche Pattern und Vorgehensweisen finden sich im Domain Driven Design, die Modulkohäsion stärken und den Kopplungsgrad der Module im Gesamtsystem mindern?
Als Entwurfsmuster sind zunächst einige der Building Blocks des taktischen Domain Driven Designs zu nennen:

- **Entity**
  Eine Entity bindet (den Prinzipien des OOP folgend) funktional verwandte Daten und Verhalten und internalisiert ihre eigene Identität in Form eines Identifiers. Das Entwurfsmuster bildet den ersten Schritt zu Erzeugung hoher Kohäsion im DDD.

- **ValueObject**
  ValueObjects werden insbesondere genutzt, um funktional verwandte Daten zu binden, was zur wiederum zur Erhöhung der Kohäsion beiträgt.

- **Aggregate**
  Aggregate gruppieren Mengen von Entities und zugehörige ValueObjects auf Basis von Invarianten und machen sie zu Modulen. Invarianten beschreiben hierbei hochkohäsive Assoziationsbeziehungen zwischen den Entities (und ValueObjects). Die Nutzung von Invarianten als Heuristik zur Modularisierung ist der wichtigste Faktor in der Erzeugung hochkohäsiver (Aggregats)-Module. Die beteiligten Entities (und ValueObjects) werden von der Aggregatesgrenze umschlossen und verlieren dabei ihre globale Identität. Dies verhindert pathologische Abhängigkeiten, die die Aggregates- bzw. Modulgrenze kreuzen und verhindert so eine der stärksten Formen der Kopplung.

- **AggregateRoot**
  Die AggregateRoot bildet das Interface eines Aggregates und öffnet die Aggregatesgrenze für die konzentrierte Abwicklung der Interaktion mit anderen Aggregaten. Die AggregateRoot reduziert so mögliche „Einfallpunkte“für Abhängigkeiten von außen und reduziert so gleichzeitig ihre Anzahl - die Kopplung wird minimiert. Sie nimmt

zusätzlich eingehende Operationen entgegen und ändert die gebundenen Entities mit strikter Konsistenz um angesprochene Invarianten einzuhalten.

- **DomainEvent**
  Ein DomainEvent beschreibt Änderungen an einem Aggregate und stellt diese Informationen für andere Aggregate zur Verfügung. DomainEvents implementieren eine eventbasierte, asynchrone und unidirektionale Kommunikation zwischen den Aggregaten. Folgeänderungen an anderen Aggregaten werden so mit eventueller Konsistenz ausgeführt und beteiligte Aggregate werden temporal entkoppelt.

Zusätzlich ist die architektonische Regel der Assoziation von Aggregaten mittels Id-Referenzen zu nennen. Sie verhindert eine direkte Objekt-Referenz auf fremde Aggregate und erschwert die Nutzung aggregatesfremder Daten oder Funktionalität maßgeblich. Eine starke inhaltliche Kopplung zwischen Aggregaten wird so unwahrscheinlich(er).

Welche technischen Lösungen bietet der Modulith-Architekturstil und das Spring Modulith Framework, die diese Pattern umsetzen und wie wirksam sind sie?

- **(Optionale) Einbindung der jMolecules Bibliothek**
  Die jMolecules bietet abstrakte Typen und Interfaces für die taktische Building Blocks und zugehörige Konzepte des Domain Driven Design: *ValueObject, Entity, Identifier, Aggregate Root, Association, DomainEvent, Service, Repository, EventListener, etc.*
  Die Nutzung dieser Typen und Interfaces durch Ableitung eigener Domänen-objekte macht architektonische Konzepte im Code evident und überprüfbar. jMolecules rückt die Objekte und Strukturen im Lösungsraum (dem Quellcode) näher an Objekte und Strukturen im Problemraum selbst und verringert so die Komplexität für Entwickler*innen bei der mentalen Überführung und Transformation zwischen den Räumen.

- **(Optionale) Einbindung der JMolecules DDD Integrationen**
  Die jMolecules-DDD-Integrationen können genutzt werden um mithilfe von Bytebuddy typischen Boilerplate-Code in einem Buildstep nachträglich den Domänenobjekten

hinzuzufügen. Die Domänenobjekte werden so insbesondere von infrastrukturellem Code zur Nutzung der Jakarta Persistenz befreit. Es resultiert eine klarere Trennung von Domänen- und Infrastrukturcode und der Quellcode Domänenobjekte im Lösungsraum rückt näher an den Problemraum bzw. die Domäne selbst. Die Komplexität der mentalen Transformation zwischen den beiden Räumen sinkt hierdurch.

- **Nutzung von Java Paketen zur Abbildung von Modul- und Aggregatsgrenzen**
  Spring Modulith nutzt Java Pakete als Module - die Erstellung von einem Paket/Modul pro Aggregat werden Modul- und Aggregatesgrenzen deckungsgleich. Die Aggregatesgrenzen werden so im Code evident und durch Tests überprüf- und erzwingbar.
- **Nutzung von ArchUnit und Bereitstellung von ArchUnit-Regeln für das Modulkonzept und Konzepte des Domain Driven Design**
  Spring Modulith bindet die ArchUnit Bibliothek ein und stellt einen Satz an Modulkonzept- und DDD-Regeln zur Verfügung. Die Nutzung von ArchUnit ermöglicht so die automatisierte Überprüfung architektonischer Konzepte mithilfe von architektonischen Tests an. Die Gefahr der Erosion architektonischer Modul- und DDD-Eigenschaften wird so nahezu ausgemerzt.
- **Bereitstellung der ApplicationModuleListener-Annotation** Die Annotation ApplicationModuleListener nutzt gängige Spring ApplicationEvents, setzt hierbei aber die in DDD geforderte eventuelle Konsistenz zwischen Aggregaten um. Events werden asynchron behandelt und gesonderte Transaktionen für Sender- und Empfänger-Aggregat erzeugt. Sie ermöglicht eine asynchrone, Eventgetriebene, eventuell konsistente Kommunikation zwischen Aggregaten und fördert so deren Entkopplung.

Zusammenfassend lässt sich sagen, dass Spring Modulith zugrundeliegende Konzepte der Modularisierung und des Domain Driven Design konsequent umsetzt. Das Framework ermöglicht die Umsetzung einer Modultih-Architektur, die als Hybrid zwischen Deployment-Monolith und Microservice-Architektur einige Vorteile der beiden Architekturstile vereint und Nachteile vermeidet. Als Vorteile ist hierbei eine geringere Komplexität im Deployment und Betrieb

zu nennen - vermiedene Nachteile liegen insbesondere in der geringeren technischen Komplexität der Kommunikation gegenüber einer Microservice Architektur.

6.2 Diskussion der Ergebnisse

Die vorliegende Arbeit hat die Konzepte der Modularisierung und zugehörige Begriffe der Kohäsion & Kopplung und den Begriff der Connascence aus den historischen Originalquellen hergeleitet. Sie hat dabei eine Brücke geschlagen zum oft oberflächlichen Verständnis der Konzepte in der modernen Literatur der Softwarearchitektur und offene Lücken geschlossen. Hierbei sei insbesondere die Unterscheidung zwischen Problemraum- und Lösungsraumbedingten Kohäsionsarten angesprochen, die in aktuellen Büchern und Aufsätzen zum Thema nicht zu finden ist. Die Arbeit bietet so eine neue bzw. wiederentdeckte Metrik, die beträchtlich zur Stärkung der Kohäsion von Modulen im Speziellen und der Modularisierung des Gesamtsystems im Allgemeinen beitragen kann.

Des Weiteren hat die Arbeit in ähnlicher Weise die Anfänge der objektorientierten Programmierung analysiert und insbesondere das Konzept der messagebasierten Kommunikation zwischen Objekten herausgearbeitet. Die Arbeit hat die Bedeutung dieses Kommunikationstyps für die Entkopplung von Modulen (und Objekten) deutlich gemacht. Sie hat dabei gezeigt, dass eine message- bzw. eventbasierte Kommunikation sich nicht nur im Bereich der Makroarchitektur eignet, sondern hat deutlich gemacht, dass sie auch auf mikroarchitektonischer Ebene von erheblicher Bedeutung ist.

Die vorliegende Forschungsarbeit hat des Weiteren Entwurfsmuster des Domain Driven Design untersucht und ihre modularisierenden Eigenschaften hergeleitet und bewiesen. Die beschriebene Forschung hat eine tiefgreifende Verknüpfung zwischen den taktischen Building Blocks und zugehörigen Mustern des Domain Driven Design und Mustern bzw. Konzepte der objektorientierten Programmierung und des Structured Design aufgezeigt. Diese Verknüpfung kann zu einer konzepttreuen Umsetzung domänengetriebener Architekturen, wie die Modulith- oder Microservice-Architektur, beitragen.

Die Arbeit beweist zudem die Tauglichkeit des Modulith-Architekturstils (bei Verwendung von Spring Modulith) stark kohäsive und lose gekoppelte Module zu erzeugen und so die Softwarequalitäten der Wart-, Änder-, Erweiter- und Testbarkeit zu stärken. Sie hat gezeigt, dass so ein hybrider Architekturstil zwischen Deployment-Monolith- und Microservice-Architekturstil ermöglicht wird, der einige der Vorteile beider Stile kombiniert und Nachteile vermeidet.

6.3 Ausblick

Aufgrund der Breite des bearbeiteten Themenfelds und seiner Auswirkung auf die verschiedensten Architektur- und Programmierstile ergibt sich eine Vielzahl an möglichen Folgearbeiten:

- **Modulith Architekturstil als Grundlage für Microservices**
  Neben der in der Einleitung beschriebenen Kontroverse, über das Für und Wider von Monolithen und Microservices, gibt es unter Softwarearchitekt*innen eine Diskussion darüber, ob man mit einem gut strukturierten Monolith beginnen sollte, auch wenn man eine Microservice Architektur als Ziel hat.

  Bspw. Martin Fowler spricht sich für die Implementierung eines gut strukturierten Monolithen als ersten Schritt aus, da in einem Greenfieldprojekt die Domäne zunächst unklar ist, vgl. [Fowler, 2015]. Er argumentiert, dass ein dadurch öfter erforderliches Verschieben von Funktionalität in andere Teile des Systems bzw. Module leichter fällt. Als zweiten Schritt sieht Fowler dann eine schrittweise Transformation der Architektur, indem sukzessive einzelne Microservices aus dem Monolithen mithilfe des Strangler Pattern ausgelöst werden, vgl. [Fowler, 2004] und [Richardson, 2018]. Gegenstimmen sprechen sich aber für eine direkte Umsetzung der Microservice-Architektur aus, da sie den zusätzlichen Aufwand der Architekturtransformation als zu groß erachten, vgl. [Tilkov, 2015].

Die Modulith-Architektur und insbesondere Spring Modulith könnte hier eine interessante neue Perspektive geben. Spring Modulith unterstützt nämlich bereits die Externalisierung von DomainEvents und die Nutzung eines externen Messagebrokers und vereinfacht so vermeintlich das Heraustrennen von Microservice aus dem Modulithen. Eine Untersuchung dieser Möglichkeit des Spring Modulith Frameworks mit einem Fokus auf die Evolvierbarkeit von Architekturen, vgl. [Ford et al., 2017], könnte wichtige Erkenntnisse für größere Softwareprojekte im realen Alltag der Wirtschaft erbringen.

- **Historische Herleitung anderer „moderner“Architekturregeln bzw. Entwurfsmuster**
  Das Vorgehen der vorliegenden Arbeit, aktuelle Architekturstile auf historischen Originalquellen herzuleiten, könnte auf andere moderne Architekturregeln und Entwurfsmuster übertragen werden. Beispielweise die SOLID-Prinzipien oder bekannte Leitsprüche wie *Don't repeat yourself* (DRY) könnten so besser begründet und auf ihre Validität untersucht werden. Die Arbeit hat gezeigt wie wichtig es ist auch als allgemeingültig angenommene Sachverhalte oder Pattern, wie beispielsweise die Kommunikation in der OOP, öfters infrage zu stellen. Das Resultat könnten überraschende neue, aber grundlegende Erkenntnisse sein, die Auswirkungen auf diverse Bereiche des aktuellen Wissensstand haben könnten.

- **Modularisierung in der funktionaler Programmierung**
  Die vorliegende Arbeit nutzt Grundlagen der objektorientierten Programmierung und Analyse und modularisiert den Problem- und Lösungsraum entsprechend in objektähnlichen Modulen. Da die funktionale Programmierung einen immer höheren Stellenwert einnimmt, wären ihre Auswirkungen auf das Modulkonzept des Strategischen Design interessantes Thema. Im gleichen Zuge könne untersucht werden inwieweit sich die Grundlagen des Domain Driven Design mit der funktionalen Programmierung vereinbaren lassen. Insbesondere eine Quellcode nahe, funktionale Formulierung des Aggregatekonzepts wäre wichtig, da sowohl Originalquellen als auch sonstige Literatur meist nur objektorientierte Quellcodebeispiele bieten.

- **Abbildung der DDD Bounded Context, Subdomänen in Spring Modulith**

Die vorliegende Arbeit nutzt Modulgrenzen des Spring Modulith Frameworks, um konzeptuelle Aggregatesgrenzen des taktischen DDD im Quellcode sichtbar, evident und überprüfbar zu machen. Ein ähnliches Vorgehen könnte genutzt werden, um das Bounded Context und/oder das (Sub-)Domänen Konzept des strategischen DDD im Code abzubilden. Es würde sich eine verschachtelte Modulhierarchie ergeben, die die Software auf oberster Ebene in Bounded Context, dann in Subdomänen und zuletzt in Aggregate unterteilt. Hierzu müssten die von Spring Modulith bereitgestellten Architekturregeln auf die vorgeschlagene Modulhierarchie angepasst werden. Eine Umsetzung würde den Lösungsraum, die modularisierte Software, noch näher an den durch DDD modularisierten Problemraum (Domäne) rücken.

- **Erarbeitung von Pattern zur Auflösung von zyklischen Abhängigkeiten**
  Die Erfahrung in der nebenläufig bearbeiteten Fallstudie zur Einarbeitung in Spring Modulith zeigt, dass man sowohl in der Domänenarbeit als auch beim eigentlichen Programmieren ständig mit dem Problem der zyklischen Abhängigkeiten konfrontiert wird.

  Eine zyklische Abhängigkeit entsteht dabei durch die wechselseitige Abhängigkeit von zwei Modulen oder Klassen und stellt aus technischer Sicht die stärkste Form der Kopplung dar. Diese Art der Kopplung lässt sich manchmal durch weitere Iterationsschleifen in der Domänenarbeit auflösen. In anderen Fällen sind Entwickler*innen versucht die beteiligten Modul zu verschmelzen und die zyklische Abhängigkeit als kohäsives Element zu betrachten. Eine unüberlegte Verschmelzung der Module kann aber zur vermeintlich unnötig großen Modulen führen und es besteht die Gefahr, dass die Entkopplung der Domäne/Software durch solche „auswuchernden“Module geschwächt wird.

  Offen ist die Frage, ob zyklische Abhängigkeiten ausschließlich im Lösungsraum entstehen und auf eine unzureichende Modularisierung des Problemraums zurückzuführen sind. Einige Stimmen sehen als zweite Möglichkeit die Existenz von Problemraum-inhärenten zyklischen Abhängigkeiten, die somit nicht aufgelöst werden können und sollten, vgl. [Lakos, 1996, S. 213ff]. Eine Beantwortung dieser Frage und eine Erarbeitung von typischen Dekompositionsmustern für zyklische Abhängigkeiten hätte große Relevanz in der Praxis.

6.4 Fallstudie

Im Rahmen dieser Arbeit wurde das Forschungs- und Lehrprojekt *Microservice-Dungeon* der TH Köln in Auszügen als Fallbeispiel umgesetzt, vgl. [Bente and et al., 2023].

Die Implementierung nutzt das Spring Modulith Framework, folgt dem Domain Driven Design Ansatz nach [Vernon, 2013] und setzt eine exemplarische Modulith-Architektur um.

Die gemachten Erfahrungen sind in die Ergebnisse dieser Arbeit eingeflossen - einige der Listings sind dem Sourcecode dieser Umsetzung entnommen. Der Sourcecode ist auf Github zu finden, siehe [Reitano, 2024].

Literaturverzeichnis

[Bente and et al., 2023] Bente, S. and et al. (2023). Longterm project »the microservice dungeon«. https://www.archi-lab.io/compounds/dungeon_main.html, [abgerufen am: 21.08.2023].

[Brown, 2014] Brown, S. (2014). Distributed big balls of mud. https://dzone.com/articles/distributed-big-balls-mud, [abgerufen am: 29.12.2023].

[Dijkstra, 1968] Dijkstra, E. W. (1968). Letters to the editor: Go to statement considered harmful. *Commun. ACM*, 11(3):147–148.

[Dowalil, 2019] Dowalil, D. (2019). Modulith first! der angemessene weg zu microservices. https://www.informatik-aktuell.de/entwicklung/methoden/modulith-first-der-angemessene-weg-zu-microservices.html [abgerufen am: 29.11.2023].

[Drotbohm et al., 2022] Drotbohm, O., Schwentner, H., and Pirnbaum, S. (2022). Architecturally evident applications – how to bridge the model-code gap? https://static.odrotbohm.de/jmolecules/jmolecules-paper.pdf [abgerufen am: 06.11.2023].

[Evans, 2004] Evans, E. (2004). *Domain-driven Design - Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, Boston.

[Ford et al., 2017] Ford, N., Parsons, R., and Kua, P. (2017). *Building Evolutionary Architectures - Support Constant Change*. O'Reilly Media, Inc., Sebastopol.

[Fowler, 2004] Fowler, M. (2004). Strangler fig application. https://martinfowler.com/bliki/StranglerFigApplication.html [abgerufen am: 15.12.2023].

[Fowler, 2015] Fowler, M. (2015). Monolithfirst. https://martinfowler.com/bliki/MonolithFirst.html [abgerufen am: 29.11.2023].

[Gamma et al., 1994] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns - Elements of Reusable Object-Oriented Software*. Pearson Education, Amsterdam.

[Grammes et al., 2018] Grammes, D. R., Lehmann, M., and Schaal, D. K. (2018). Java 9 bringt das neue modulsystem jigsaw. https://www.informatik-aktuell.de/entwicklung/programmiersprachen/java-9-das-neue-modulsystem-jigsaw-tutorial.html [abgerufen am: 11.12.2023].

[Junker, 2020] Junker, A. (2020). Microservices oder monolithen? beides! https://entwickler.de/software-architektur/microservices-oder-monolithen-beides, [abgerufen am: 29.12.2023].

[Kay, 1993] Kay, A. C. (1993). The early history of smalltalk. *SIGPLAN Not.*, 28(3):69–95.

[Kay, 1998] Kay, A. C. (1998). Alan kay on messaging. http://wiki.c2.com/?AlanKayOnMessaging, [abgerufen am: 29.12.2023].

[Khorikov, 2015] Khorikov, V. (2015). Cohesion and coupling: the difference. https://enterprisecraftsmanship.com/posts/cohesion-coupling-difference/, [abgerufen am: 29.12.2023].

[Lakos, 1996] Lakos, J. (1996). *Large-scale C++ Software Design*. Addison-Wesley, Amsterdam, 1 edition.

[Martin, 2008] Martin, R. C. (2008). *Clean Code - A Handbook of Agile Software Craftsmanship*. Pearson Education, Amsterdam.

[Martin, 2018] Martin, R. C. (2018). *Clean Architecture - A Craftsman's Guide to Software Structure and Design*. Prentice Hall, London.

[Miller, 1956] Miller, G. A. (1956). The magical number seven, plus or minus two: Some limits on our capacity for processing information. `http://www.musanim.com/miller1956/`, originally published in The Psychological Review, 1956, vol. 63, pp. 81-97 (reproduced here, with the author's permission, by Stephen Malinowski), [abgerufen am: 29.11.2023].

[Myers, 1975] Myers, G. J. (1975). *Reliable Software Through Composite Design*. Van Nostrand Reinhold, New York, 1. edition.

[Newman, 2015] Newman, S. (2015). *Building Microservices - Designing Fine-Grained Systems*. Ö'Reilly Media, Inc.", Sebastopol.

[Page-Jones, 1995] Page-Jones, M. (1995). *What Every Programmer Should Know about Object-oriented Design*. Dorset House Pub., New York.

[Page-Jones, 2000] Page-Jones, M. (2000). *Fundamentals of Object-oriented Design in UML*. Addison-Wesley Professional, Boston, 1 edition.

[Reitano, 2024] Reitano, M. (2024). Amundsen - fallstudie modulith dungeon. `https://github.com/MarcoReitano/amundsen`.

[Richards and Ford, 2020] Richards, M. and Ford, N. (2020). *Fundamentals of Software Architecture - An Engineering Approach*. Ö'Reilly Media, Inc.", Sebastopol.

[Richardson, 2018] Richardson, C. (2018). *Microservices Patterns - With examples in Java*. Simon and Schuster, New York.

[Schlosser, 2019] Schlosser, H. (2019). Eine radikal andere objektorientierung: 'der kernbegriff ist messaging'. `https://entwickler.de/dotnet/eine-radikal-andere-objektorientierung-der-kernbegriff-ist-messaging`, [abgerufen am: 29.12.2023].

[SoftwareAG, 2021] SoftwareAG (2021). Api integration trendreport 2021. https://www.softwareag.com/content/dam/softwareag/global/marketing-material/en/ebook/webmethods/vanson-bourne-report-2021.pdf, [abgerufen am: 29.12.2023].

[Spring, 2023] Spring (2023). Spring modulith - fundamentals. https://docs.spring.io/spring-modulith/reference/fundamentals.html, [abgerufen am: 18.10.2023].

[Spring.io, 2023a] Spring.io (2023a). Spring modulith - integration testing application modules. https://docs.spring.io/spring-modulith/reference/testing.html, [abgerufen am: 18.10.2023].

[Spring.io, 2023b] Spring.io (2023b). Spring modulith - verifying application module structure. https://docs.spring.io/spring-modulith/reference/verification.html, [abgerufen am: 18.10.2023].

[Spring.io, 2023c] Spring.io (2023c). Spring modulith - working with application events. https://docs.spring.io/spring-modulith/reference/events.html, [abgerufen am: 18.10.2023].

[Thoughtworks, 2015] Thoughtworks (2015). Technology radar 2015 - osgi. https://www.thoughtworks.com/de-de/radar/platforms/osgi, [abgerufen am: 18.10.2023].

[Thoughtworks, 2018] Thoughtworks (2018). Technology radar 2018 - microservice envy. https://www.thoughtworks.com/de-de/radar/techniques/microservice-envy, [abgerufen am: 29.12.2023].

[Tilkov, 2015] Tilkov, S. (2015). Don't start with a monolith ... when your goal is a microservices architecture. https://martinfowler.com/articles/dont-start-monolith.html [abgerufen am: 29.11.2023].

[Vernon, 2013] Vernon, V. (2013). *Implementing Domain-Driven Design*. Addison-Wesley, Amsterdam.

[Yourdon and Constantine, 1979] Yourdon, E. and Constantine, L. L. (1979). *Structured Design - Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice Hall, London.

Abbildungsverzeichnis

Listings

Eidesstattliche Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer oder der Verfasserin/des Verfassers selbst entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Ort, Datum

Unterschrift