Berner
Fachhochschule

Informatik Seminar

Analyse und Bewertung von Software-Architekturen

Studiengang Bachelor of Science in Computer Science
Autor Jonas Burkhalter
Dozent Prof. Dr. Jürgen Eckerle

Version 1.0 vom 30. Mai 2022

► Technik und Informatik

Abstract

Software kann eine über die funktionalen Anforderungen hinausgehende Qualität besitzen. Zu diesen nicht funktionalen Qualitätsmerkmalen gehören, beispielsweise Performanz, Wartbarkeit, Erweiterbarkeit, Lesbarkeit, usw.

Um einige davon zu analysieren schlug Robert Martin eine Reihe von Metriken vor rund um die main sequence line. Dabei werden primär die Abstraktion und das Coupling von Klassen analysiert.

Auch Chidamber und Kemerer haben in ihrer Arbeit ”*A metrics suite for object oriented design*” [1] sechs Metriken zur Analyse von Softwarearchitektur veröffentlicht.

Die drei Tools CodeMr, CodeScene und JDepend wurden im Verlauf dieser Arbeit zur Analyse verschiedenster Metriken verwendet. Dabei hat sich herausgestellt, dass alle drei Tools geeignet sind für die Analyse von Softwarearchitekturen. JDepend beschränkt sich sehr auf die Metriken von Robert Martin, CodeMR setzt hauptsächlich auf die Visualisierung und CodeScene vermischt diverse Metriken um vereinfachte Aussagen über den Zustand der Software zu treffen.

Glossar

- **CI/CD-Pipeline**: CI/CD-Pipelines sind eine Praxis, die sich auf die Verbesserung der Softwarebereitstellung während des gesamten Lebenszyklus der Softwareentwicklung durch Automatisierung konzentriert.
- **Commit**: Der Commit-Befehl wird verwendet, um Änderungen in Git in einem lokalen Repository zu speichern
- **Commit Message**: Die Commit Message ist ein vom Entwickler verfasster Text, welcher die Änderungen die in einem Commit beschreiben soll
- **elasticsearch**: elasticsearch ist die weltweit führende kostenlose und offene Such- und Analyselösung. CodeScene verwendet dies als einen der öffentlichen Showcases.
- **McCabe complexity**: McCabe complexity oder auch zyklomatische Komplexität ist eine Softwaremetrik, die verwendet wird, um die Komplexität eines Programms anzugeben. Die Metrik ist ein quantitatives Mass für die Anzahl linear unabhängiger Pfade durch den Quellcode eines Programms. Die Metrik wurde 1976 von Thomas J. McCabe, Sr. entwickelt.

Inhaltsverzeichnis

Abstract iii

Glossar v

1 Grundlagen 1
1.1 Anforderungstypen 1
1.1.1 Funktionale Anforderungen 1
1.1.2 Nicht funktionale Anforderungen 1
1.2 Qualitätsattribute 2
1.2.1 Security (Sicherheit) 2
1.2.2 Capacity (Kapazität) 2
1.2.3 Compatibility (Kompatibilität) 2
1.2.4 Reliability (Zuverlässigkeit) 2
1.2.5 Availability (Verfügbarkeit) 2
1.2.6 Performance (Leistung) 2
1.2.7 Scalability (Skalierbarkeit) 3
1.2.8 Usability (Benutzbarkeit) 3
1.2.9 Regulatory (Regulativ) 3
1.2.10 Environmental (Umgebungen) 3
1.2.11 Maintainability (Wartbarkeit) 3
1.2.12 Manageability (Verwaltbarkeit) 3

2 Metriken 5
2.1 Main sequence line 5
2.1.1 Efferent Coupling (Ce) 5
2.1.2 Afferent Coupling (Ca) 6
2.1.3 Instability (I) 6
2.1.4 Abstractness (A) 7
2.1.5 Normalized Distance from Main Sequence (D) . . . 8
2.1.6 Fazit 9
2.2 Chidamber and Kemerer 10
2.2.1 Weighted Methods Per Class (WMC) 10
2.2.2 Depth of Inheritance Tree (DIT) 10
2.2.3 Number of Children (NOC) 10
2.2.4 Coupling between Object Classes (CBO) 10
2.2.5 Response for a Class (RFC) 10

2.2.6 Lack of Cohesion of Methods (LCOM) 11

3 Tools 13
3.1 CodeMR 13
3.1.1 Metriken 13
3.1.2 Visualisierung 15
3.2 CodeScene 19
3.2.1 Visualisierung 19
3.3 JDepend 22
3.3.1 Metriken 22
3.3.2 Reports 23

4 Fazit 25

Literaturverzeichnis 29

Abbildungsverzeichnis 31

1 Grundlagen

In den folgenden Unterkapitel werden grundlegende Begriffe und Konzepte erläutert.

1.1 Anforderungstypen

1.1.1 Funktionale Anforderungen

Funktionale Anforderungen bestimmen das Verhalten einer Applikation oder einer Komponente. Dabei werden meist die gleichen drei Schritte ausgeführt: Dateneingabe - Verarbeitung - Datenausgabe. Die Verarbeitung kann aus Daten bearbeiten, berechnen, ablegen, ausführen von Geschäftsprozessen oder weiteren Aufgaben bestehen.

Dieser Type von Anforderung zeigt auf, **was** eine Applikation macht.

Funktionale Anforderungen beschreiben dem Softwareentwickler, wie sich eine Applikation verhalten soll. Werden diese Anforderungen nicht erfüllt, bedeutet dies, dass die Applikation aus Sicht des Benutzers nicht ordnungsgemäss funktioniert.

1.1.2 Nicht funktionale Anforderungen

Nicht funktionale Anforderungen beschreiben Qualitätsattribute einer Applikation wie die Benutzbarkeit, Sicherheit, Skalierbarkeit und viele andere.

Während funktionale Anforderungen bestimmen, was eine Applikation macht, beschreiben nicht funktionale Anforderungen, **wie** das System es macht.

Werden die nicht funktionalen Anforderungen nicht erfüllt, führt die Applikation die grundlegenden Funktionen weiterhin korrekt aus. Jedoch kann dies zu einer beschwerlichen Benutzer- oder Softwareentwicklererfahrung werden.

[2]

1.2 Qualitätsattribute

Nachfolgend werden einige Qualitätsattribute aufgelistet, die Liste ist nicht abschliessend.

1.2.1 Security (Sicherheit)

- Werden sensibel Informationen überträgen oder gespeichert?
- Wer kann auf welche Informationen zugreifen?

1.2.2 Capacity (Kapazität)

- Welche Datenmenge generiert die Applikation?
- Hängt die Datenmenge von der Anzahl Benutzer ab?
- Wie lange werden Daten gespeichert und archiviert?

1.2.3 Compatibility (Kompatibilität)

- Was sind die Mindestanforderungen an die Hardware?
- Welche Betriebssysteme und Versionen werden unterstützt?

1.2.4 Reliability (Zuverlässigkeit)

- Wie gross muss die Wahrscheinlichkeit sein, dass die Applikation über einen bestimmten Zeitraum erwartungsgemäss funktioniert?

1.2.5 Availability (Verfügbarkeit)

- Wie häufig und wie lange dürfen Ausfälle maximal sein?
- Wie lange dürfen Wartungsfenster sein?

1.2.6 Performance (Leistung)

- Wie schnell reagiert die Applikation auf die Aktionen eines Benutzers?
- Wie lange dauert eine Verarbeitung im Hintergrund?

1.2.7 Scalability (Skalierbarkeit)

- Was ist die höchste Belastung, unter der die Applikation noch funktioniert?
- Ab wie vielen Benutzern wird die Leistungsanforderung nicht mehr erfüllt?
- Kann die Applikation horizontal oder vertikal skaliert werden?

1.2.8 Usability (Benutzbarkeit)

- Wie einfach ist es, die Applikation zu verwenden?
- Wie effizient kann ein Benutzer die Applikation bedienen?
- Gibt es verschiedene Arten von Benutzern?

1.2.9 Regulatory (Regulativ)

- Gibt es gesetzliche Anforderungen die eingehalten werden müssen?
- Ist die Einhaltung anderer Standarte notwendig?

1.2.10 Environmental (Umgebungen)

- In welchen Arten von Umgebungen soll die Applikation laufen?
- Werden separate Umgebungen verwendet für Tests, Demos, Reviews usw.?

1.2.11 Maintainability (Wartbarkeit)

- Wie lange dauert es Bugs zu beheben?
- Wie einfach ist es Änderungen an der Applikation vorzunehmen?
- Wie viel Aufwand wird benötigt, um die Applikation auf eine andere Plattform zu portieren?

1.2.12 Manageability (Verwaltbarkeit)

- Wie einfach kann ein Administrator die Applikation verwalten?
- Wie einfach können Schäden und Fehlinformationen aus Bugs behoben werden?

[3]

2 Metriken

In den folgenden Unterkapitel werden verschiedene Metriken für Softwarearchitektur erläutert.

2.1 Main sequence line

Robert Martin schlug 1994 eine Reihe von Metriken für objektorientiertes Design vor. Diese Metriken verwenden nicht alle Attribute, sondern konzentrieren sich auf die Beziehung zwischen Packages und Klassen innerhalb eines Projektes.

2.1.1 Efferent Coupling (Ce)

Diese Metrik wird verwendet, um die Beziehungen zwischen den Klassen zu messen. Es handelt sich um die Anzahl von Klassen in einem bestimmten Package, die von Klassen in anderen Packages abhängt. Es ermöglicht, die Anfälligkeit des Packages für Änderungen in externen Packages zu messen.

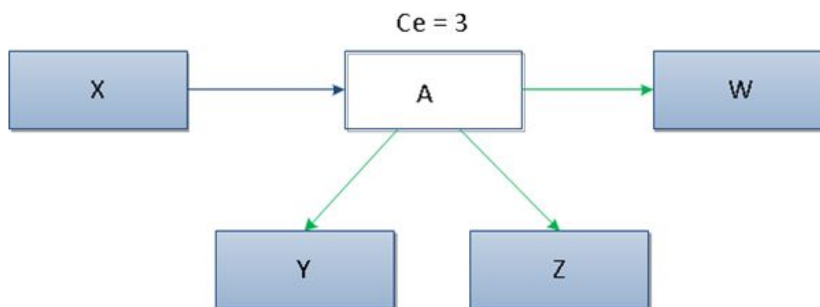

Abbildung 2.1: Efferent Coupling

In der Abbildung 2.1 ist zu sehen, dass Klasse A ausgehende Abhängigkeiten zu 3 anderen Klassen hat, deshalb ist die Metrik *Ce* für diese Klasse 3.

Ein hoher Wert der Metrik *Ce* zeigt die Instabilität eines Packages an, Änderungen in einer der zahlreichen externen Klassen können Änderungen am Package erforderlich machen. Bevorzugte Werte für die Metrik *Ce* liegen im Bereich von 0 bis 20, höhere Werte verursachen Probleme bei der Wartung und Entwicklung des Codes.

2.1.2 Afferent Coupling (Ca)

Diese Metrik ist eine Ergänzung zur Metrik *Ce* und wird verwendet, um eingehende Abhängigkeiten zwischen Packages zu messen. Es ermöglicht uns, die Empfindlichkeit der verbleibenden Packages gegenüber Änderungen im analysierten Package zu messen.

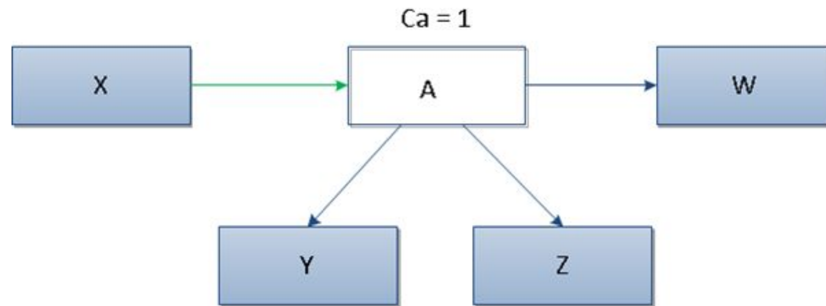

Abbildung 2.2: Afferent Coupling

In der Abbildung 2.2 ist zu sehen, dass Klasse A nur eine eingehende Abhängigkeit (von Klasse X) hat, deshalb ist der Wert für die Metrik *Ca* gleich 1.

Ein hoher Wert der Metrik *Ca* deutet normalerweise auf eine hohe Komponentenstabilität hin. Dies liegt daran, dass die Klasse von vielen anderen Klassen abhängt. Daher kann sie nicht wesentlich geändert werden, da in diesem Fall die Wahrscheinlichkeit der Verbreitung solcher Änderungen zunimmt. Bevorzugte Werte für die Metrik *Ca* liegen im Bereich von 0 bis 500.

2.1.3 Instability (*I*)

Diese Metrik wird verwendet, um die relative Anfälligkeit der Klasse für Änderungen zu messen. Gemäss der Definition ist Instabilität das Verhältnis von ausgehenden Abhängigkeiten *Ce* zu allen Package-Abhängigkeiten und akzeptiert Werte von 0 bis 1.

Die Metrik ist nach der folgenden Formel definiert:

$$I = \frac{Ce}{Ce + Ca}$$

Wobei: *Ce* – ausgehende Abhängigkeiten, *Ca* – eingehende Abhängigkeiten

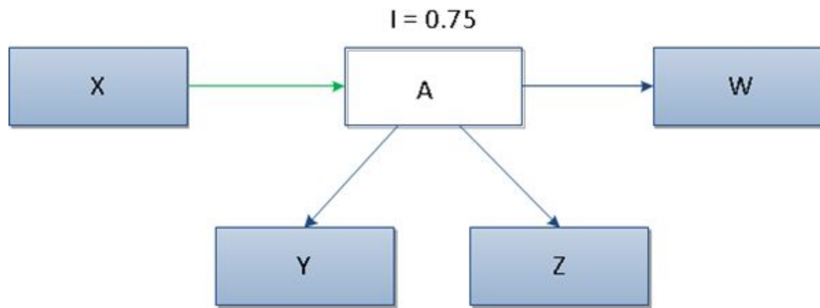

Abbildung 2.3: Instability

In der Abbildung 2.3 ist zu sehen, dass Klasse A 3 ausgehende und 1 eingehende Abhängigkeit hat, daher wird gemäss der Formel der Wert der Metrik I gleich 0,75 sein.

Basierend auf dem Wert der Metrik I können wir folgende zwei Arten von Komponenten unterscheiden:

- Komponenten die viele ausgehende Abhängigkeiten Ce und nicht viele eingehende Abhängigkeiten Ca haben. Diese haben einen Wert I der nahe bei 1 liegt. Daraus folgt, dass diese eher instabil sind aufgrund der vielen Änderungsmöglichkeiten in den ausgehenden Abhängigkeiten.
- Komponenten die viele eingehende Abhängigkeiten Ca und nicht viele ausgehende Abhängigkeiten Ce haben. Diese haben einen Wert I der nahe bei 0 liegt. Diese sind aufgrund ihrer grösseren Verantwortung schwieriger zu modifizieren.

Bevorzugte Werte für die Metrik I sollten in die Bereiche von 0 bis 0,3 oder 0,7 bis 1 fallen. Komponenten sollten sehr stabil oder instabil sein, daher sollten wir Komponenten mit mittlerer Stabilität vermeiden.

2.1.4 Abstractness (A)

Diese Metrik wird verwendet, um den Abstraktionsgrad des Packages zu messen. Definitionsgemäss ist Abstraktheit die Anzahl der abstrakten Klassen im Package zur Anzahl aller Klassen.

Die Metrik ist nach der folgenden Formel definiert:

$$A = \frac{T_{abstract}}{T_{abstract} + T_{concrete}}$$

Wobei: $\mathrm{T}_{\mathrm{abstract}}$ – Anzahl abstrakter Klassen in einem Package, $\mathrm{T}_{\mathrm{concrete}}$ – Anzahl konkreter Klassen in einem Package

Bevorzugte Werte für die Metrik A sollten Extremwerte nahe 0 oder 1 annehmen. Packages, die stabil sind (Metrik I nahe 0), was bedeutet, dass sie auf einem sehr niedrigen Niveau von anderen Packages abhängig sind, sollten auch abstrakt sein (Metrik A nahe bis 1). Die sehr instabilen Packages (Metrik I nahe 1) wiederum sollten aus konkreten Klassen bestehen (Metrik A nahe 0).

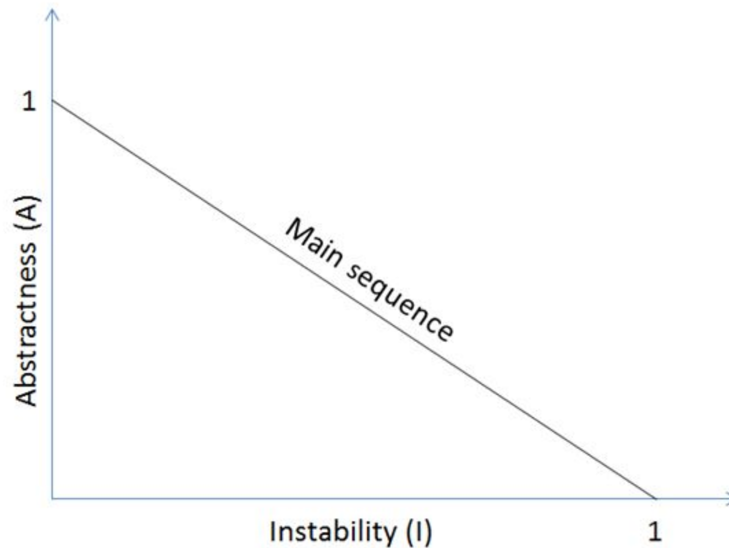

Abbildung 2.4: Abstractness

Erwähnenswert ist ausserdem, dass die Kombination von Abstraktheit und Stabilität es Robert Martin ermöglichte, eine These über die Existenz der "main sequence line" zu formulieren siehe 2.4

Im optimalen Fall wird die Instabilität der Klasse durch ihre Abstraktheit kompensiert, es ergibt sich die folgende Gleichung:

$$I + A = 1$$

Gut gestaltete Klassen sollten sich um diese Graph-Endpunkte entlang der "main sequence line" gruppieren.

2.1.5 Normalized Distance from Main Sequence (D)

Diese Metrik wird verwendet, um das Gleichgewicht zwischen Stabilität und Abstraktheit zu messen, und wird mit der folgenden Formel berechnet:

$$D = |A + I - 1|$$

Wobei: A – Abstraktheit, I – Instabilität

Der Wert der Metrik D kann folgendermassen interpretiert werden. Wenn wir eine gegebene Klasse auf den Graphen der "main sequence line" setzen (siehe Abb. 2.5), wird ihr Abstand von der "main sequence line" proportional zum Wert von D sein.

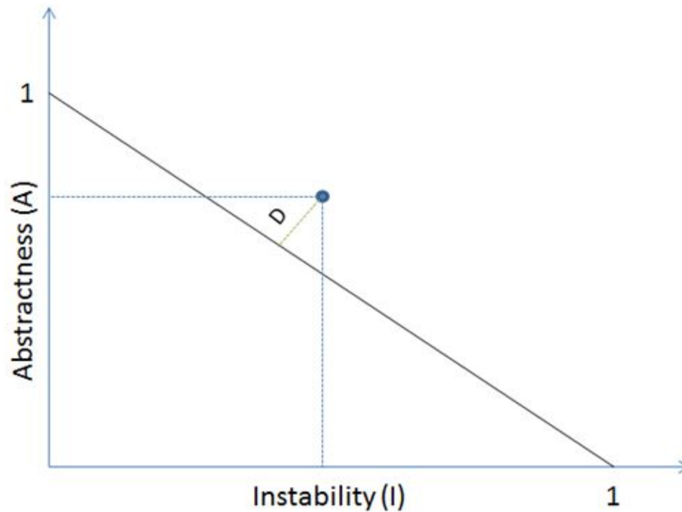

Abbildung 2.5: Normalized distance from Main Sequence

Der Wert der Metrik D sollte so niedrig wie möglich sein, damit die Komponenten nahe an der "main sequence line" liegen. Ausserdem werden die beiden äusserst ungünstigen Fälle betrachtet:

- $A = 0$ und $I = 0$, ein Package ist extrem stabil und fest, die Situation ist unerwünscht, weil das Package sehr steif ist und nicht verlängert werden kann;
- $A = 1$ und $I = 1$, eher unmögliche Situation, da ein vollständig abstraktes Package eine Verbindung nach aussen haben muss, damit die Instanz, die die in abstrakten Klassen definierte Funktionalität implementiert, die in diesem Package enthalten ist, erstellt werden kann.

2.1.6 Fazit

Anhand von Robert Martin's Metriken können wir die Beziehungen zwischen den Packages in einem Projekt leicht beurteilen und feststellen, ob es notwendig ist, Änderungen vorzunehmen, um mögliche Probleme in der Zukunft zu vermeiden.

[4]

2.2 Chidamber and Kemerer

Shyam Chidamber und Chris Kemerer veröffentlichten 1994 in ihrer Arbeit ”*A metrics suite for object oriented design*” [1] sechs Metriken zu objektorientiertem Design.

2.2.1 Weighted Methods Per Class (WMC)

Dies definiert die Summe aller Methoden in einer Klasse repräsentiert von ihrer McCabe complexity. Klassen mit vielen Methoden sind wahrscheinlich anwendungsspezifischer, was die Möglichkeit der Wiederverwendung einschränkt

2.2.2 Depth of Inheritance Tree (DIT)

DIT ist die Tiefe der Klasse im Vererbungsbaum, wobei der Wert Null Stammklassen entspricht und Mehrfachvererbung die maximale Länge zeigt. Je tiefer eine Klasse in der Hierarchie ist, desto mehr Methoden und Variablen erbt sie wahrscheinlich, was sie komplexer macht. Tiefe Bäume weisen auf eine grössere Designkomplexität hin.

2.2.3 Number of Children (NOC)

NOC definiert die Anzahl direkter Subklassen einer Klasse. Dabei ergibt sich, je mehr Subklasse eine Klasse hat, desto grösser ist der Aufwand das Verhalten dieser Subklassen beim Bearbeiten der Klasse nicht zu brechen. Infolgedessen ist es schwieriger, die Klasse zu ändern, und es sind mehr Tests erforderlich.

2.2.4 Coupling between Object Classes (CBO)

Die Anzahl Klassen mit der eine Klasse gekoppelt ist. Zwei Klassen sind gekoppelt, wenn Methoden, die in einer Klasse deklariert sind, Methoden oder Instanzvariablen verwenden, die von der anderen Klasse definiert wurden. Mehr Kopplung bedeutet, dass der Code schwieriger zu warten ist, da Änderungen in anderen Klassen auch Änderungen in dieser Klasse verursachen können. Daher sind diese Klassen weniger wiederverwendbar und erfordern mehr Testaufwand.

2.2.5 Response for a Class (RFC)

Die Anzahl der Methoden, die möglicherweise von aussen aufgerufen werden können. Wenn die Anzahl der Methoden, die in einer Klasse aufgerufen werden können, hoch ist, wird die Klasse als komplexer betrachtet und kann stark mit

anderen Klassen gekoppelt sein. Daher ist ein grösserer Test- und Wartungsaufwand erforderlich.

2.2.6 Lack of Cohesion of Methods (LCOM)

Misst wie Methoden einer Klasse zueinander in Beziehung stehen. Niedriege Kohäsion deutet darauf hin, dass die Klasse mehr als eine Verantwortung implementiert.

[5]

3 Tools

In den folgenden Unterkapitel werden einige Tools zu Analyse von Softwarearchitektur vorgestellt.

3.1 CodeMR

CodeMR ist ein Analysetool für Softwarequalität und statischen Code für Java-, Kotlin- und Scala-Projekte. CodeMR visualisiert Codemetriken und Qualitätsattribute auf hoher Ebene (Kopplung, Komplexität, Kohäsion und Grösse) in verschiedenen Ansichten, wie z. B. Packagestruktur, TreeMap, Sunburst, Abhängigkeit und Diagrammansichten. CodeMR analysiert den Quellcode auf dem lokalen Rechner und speichert alle Analysedateien in einem lokalen Arbeitsverzeichnis.

3.1.1 Metriken

CodeMR stellt diverse Metriken zur Softwareanalyse zur Verfügung. Nachfolgend werden die Gruppierungen und einige der Metriken erklärt.

Grösse

Die Grösse ist eine der einfachsten und häufigsten Formen der Softwareanalyse. Die folgende Auflistung enthält einige der Metriken, die CodeMR zur Grösse bereitstellt.

- **CLOC (Class Lines of Code)**: Die Anzahl Zeilen Code in einer Klasse
- **NOM (Number of Methods)**: Die Anzahl der Methoden in einer Klasse
- **NoCls (Number of Classes)**: Gesamtzahl der Klassen
- **NoE (Number of Entities)**: Gesamtzahl der Schnittstellen und Klassen
- **nofP (Number of Packages)**: Anzahl der Packages im Projekt
- ...

Komplekität

Komplexität bedeutet, schwer verständlich zu sein, und beschreibt die Wechselwirkungen zwischen einer Reihe von Einheiten. Höhere Komplexitätsgrade in der Software erhöhen das Risiko, unbeabsichtigt Interaktionen zu stören, und erhöhen somit die Wahrscheinlichkeit, dass bei Änderungen Fehler eingeführt werden.

- **WMC (Weighted Method Count)**: Die gewichtete Summe der Methoden einer Klasse und der zyklomatischen Komplexität der Methoden.
- **DIT (Depth of Inheritance Tree)**: Die Position der Klasse im Vererbungsbaum. Hat den Wert Null für Stamm- und nicht geerbte Klassen. Für die Mehrfachvererbung zeigt die Metrik die maximale Länge an.
- **RFC (Response For a Class)**: Die Anzahl der Methoden, die möglicherweise von aussen aufgerufen werden können.
- **SI (Specialization Index)**: Die Metrik des Specialization Index misst das Ausmass, in dem Unterklassen ihre Vorgängerklassen überschreiben.
- ...

Coupling

Coupling beschreibt die Beziehung zwischen verschiedenen Klassen oder Packages.

- **NOC (Number of Children)**: Die Anzahl der direkten Unterklassen einer Klasse
- **CBO (Coupling Between Object Classes)**: Die Anzahl der Klassen, an die eine Klasse gekoppelt ist
- **EC (Efferent Coupling)**: Die Anzahl der Klassen in anderen Packages, von denen die Klassen im Package abhängen
- **AC(Afferent Coupling)**: Die Anzahl der Klassen in anderen Packages, die von Klassen innerhalb des Package abhängen
- ...

Cohesion

Kohäsion oder Zusammenhalt zeigt an, wie gut die Methoden einer Klasse zueinander in Beziehung stehen. Hohe Kohäsion ist tendenziell vorzuziehen.

- **LCOM (Lack of Cohesion of Methods)**: Zeigt wie Methoden einer Klasse zueinander in Beziehung stehen

- **LCAM (Lack of Cohesion Among Methods(1-CAM)):** Mass des Zusammenhalts basierend auf Parametertypen von Methoden
- **LTCC (Lack Of Tight Class Cohesion (1-TCC)):** Misst den Mangel an Zusammenhalt zwischen den öffentlichen Methoden einer Klasse
- ...

3.1.2 Visualisierung

CodeMR kann Visualisierungen auf Stufe eines Projektes oder eines Package vornehmen. Zudem kann aus verschiedenen Arten von Visualisierung gewählt werden. Im nachfolgenden Unterkapitel werden einige Arten davon gezeigt.

Verteilung

Um eine Verteilung einer Metrik anzuzeigen kann ein Metric Distribution Diagramm (siehe 3.1) verwendet werden. Dabei wird eine einzelne Metrik über das Projekt oder Package dargestellt.

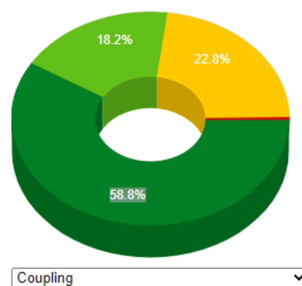

Abbildung 3.1: CodeMR - Metric Distribution Diagramm

Package Übersicht

Bei der Package Übersicht 3.2 werden die Klassen und Packages als Blasen dargestellt. Zusätzlich kann das Farbschema entsprechend den verschieden Metriken umgeschaltet werden.

Abbildung 3.2: CodeMR - Package Übersicht

Package Dependencies

Mithilfe des Package Dependencies Diagramm 3.3 können schnell und einfach alle Abhängigkeiten eines einzelnen Packages ermittelt und analysiert werden.

Abbildung 3.3: CodeMR - Package Dependencies Diagramm

Graph

Ein weiteres Diagramm ist das Graphdiagramm 3.4. Wobei Klassen als Nodes, gruppiert in den Packages, dargestellt werden und mit Pfeilen die Dependencies eingezeichnet werden.

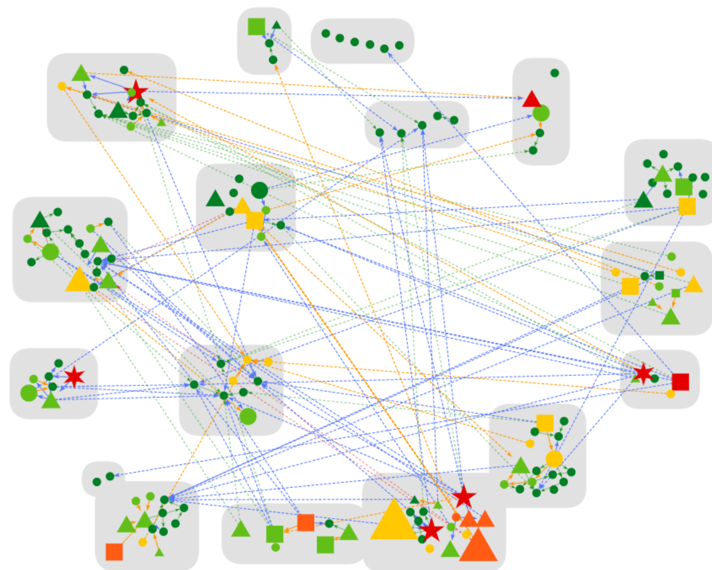

Abbildung 3.4: CodeMR - Graph Diagramm

Visuelle Eigenschaften eines Nodes im Diagramm sind seine Farbe, Größe und seine Form. Jedes Qualitätsattribut wird verschiedenen physikalischen Eigenschaften eines Nodes zugeordnet.

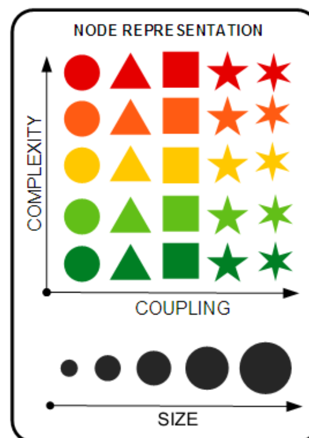

Abbildung 3.5: CodeMR - Metrik Legende

- **Farbe**: Zeigt die Komplexität der Entität an. Komplexität ändert die Farbskala in ein Grün-zu-Rot-Spektrum. Farben werden wärmer, während die Komplexität einer Entität zunimmt.
- **Form**: Während das Coupling einer Klasse zunimmt, wird die Nodeform eckiger. Eine Klasse mit hohem Coupling hat mehr Ecken, die auf mehr In-

teraktionspunkte mit anderen Klassen hinweisen. Kreis bedeutet niedriges Coupling, wobei Stern mit sechs Kanten sehr hohes Coupling bedeutet.

- **Grösse**: Die Fläche der Form stellt die Grösse der Klasse dar, wenn die Anzahl Codezeilen der Klasse zunimmt, wird der Node grösser.

Zusätzliche Informationen

In diversen Diagrammen können zusätzlich detaillierte Auswertungen für Klassen oder Packages angezeigt werden. Wie in Abbildung 3.6 zusehen, können diverse Metriken ausgewertet werden.

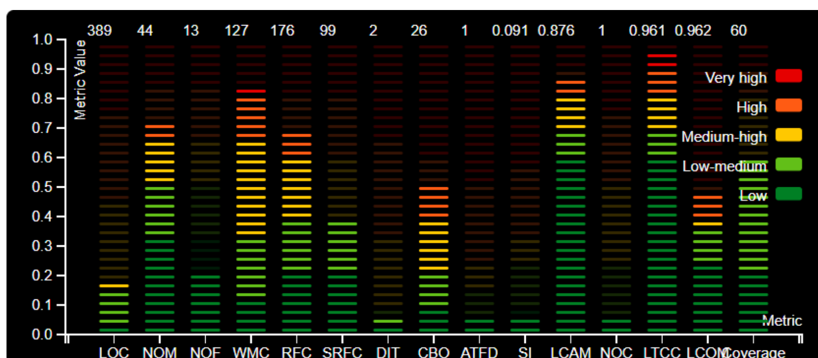

Abbildung 3.6: CodeMR - Detailmetrik

[6]

3.2 CodeScene

CodeScene ist ein Mehrzecktool, dass sowohl technische wie auch soziale Aspekte des Codes analysiert. So werden neben Metriken wie Lines of Code auch die häufigkeit von Commits einbezogen.

3.2.1 Visualisierung

Im nachfolgenden Unterkapitel werden einige Visualisierungen und Analysen von CodeScene gezeigt.

Hotspots

Die meisten Codeänderungen werden in der Regel in relativ wenigen Packages vorgenommen. Eine Hotspot-Analyse identifiziert die Packages in denen die meiste Zeit für Änderungen verbracht wird.

Hotspots werden als Ausgangspunkt für die Priorisierung von Produktivitätsverlust verwendet.

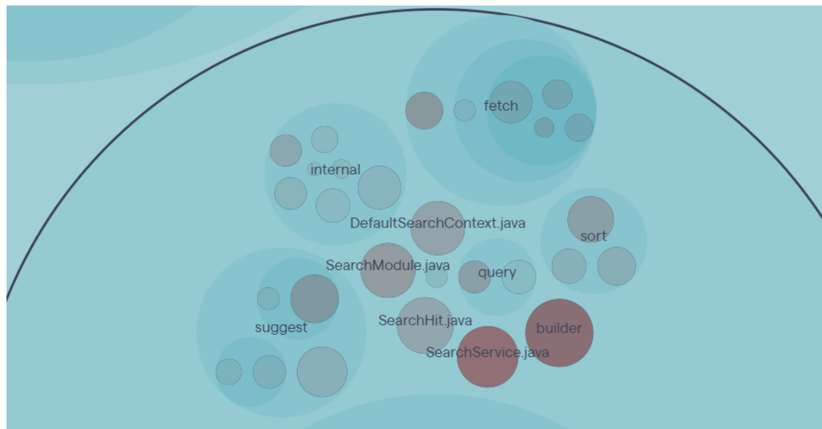

Abbildung 3.7: CodeScene - Hotspots

In der Abbildung 3.7 ist ein Ausschnitt der Hotspot-Analyse des Showcases von elasticsearch zu sehen.

Code Health

Die Code Health Metrik basiert auf bekannten Mustern, die die Wartungskosten erhöhen und den Code schwerer verständlich machen. Somit wird das Änderungsrisiko grösser und die Wartung und Weiterentwicklung verlangsamt.

Der Code Health Score wird aus diversen Faktoren zusammengerechnet, unter anderem die Folgenden:

- **Brain Method**: Eine einzelne Methode, die zu viel Verhalten zentriert und zu einem lokalen Hotspot wird
- **Developer Congestion**: Code der zu einem Koordinationsengpass wird, wenn mehrere Entwickler parallel daran arbeiten müssen
- **Don't Repeat Yourself Violations**: CodeScene erkennt doppelte Logik, die zusammen in vorhersagbaren Mustern geändert wird.
- **Primitive Obsession**: Code, der ein hohes Mass an integrierten Primitives wie Ganzzahlen, Strings und Floats verwendet, hat häufig keine Domänensprache, die die Validierung und Semantik von Funktionsargumenten kapselt.
- ...

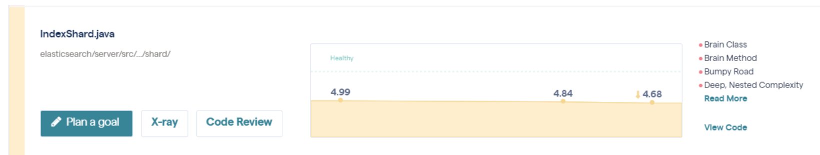

Abbildung 3.8: CodeScene - Code Health

In der Abbildung 3.8 ist ein Ausschnitt der Code Health-Analyse des Showcases von elasticsearch zu sehen.

Die Code Health-Trends können automatisch in einer CI/CD-Pipeline überwacht werden.

Change Coupling

Change Coupling erkennt wenn zwei oder mehr Klassen über einen gewissen Zeitraum, gemeinsam angepasst werden. Change Coupling an sich ist weder gut noch schlecht, es hängt davon ab, welche Klassen gekoppelt sind und aus welchem Grund sie sich gemeinsam entwickeln.

Dabei werden Klassen als Change Coupling betrachtet sofern eine oder mehrere der folgenden Bedingungen regelmässig erfüllt werden.

- Die Klassen werden im gleichen Commit bearbeitet
- Die Klassen werden innerhalb einer bestimmten Zeit von demselben Entwickler geändert
- Die Commit Message der Änderungen beziehen sich auf das selbe Ticket

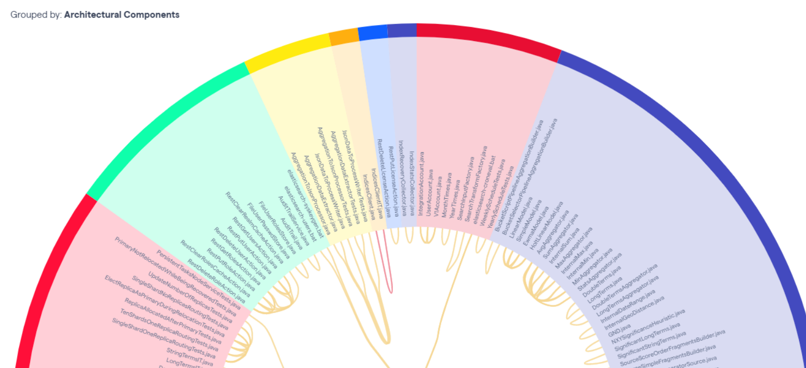

Abbildung 3.9: CodeScene - Change Coulping

In der Abbildung 3.9 ist ein Ausschnitt der Change Coupling-Analyse des Showcases von elasticsearch zu sehen.

Knowledge distribution

Die Knowledge distribution-Analyse misst diverse Aspekte darunter unter anderem die Folgenden:

- **Coordination bottlenecks**: Teile des Codes, an denen mehrere Teams ihre Arbeit koordinieren müssen
- **Knowledge Island**: Der Hauptteil des Code wurde von nur einem Entwickler geschrieben
- **Complex Code by Former Contributors**: Stellt Code mit niedriger Code Health dar, wobei der Grossteil dieses Codes von ehemaligen Mitwirkenden geschrieben wurde.

[7]

3.3 JDepend

JDepend ist ein OpenSource Softwareanalysetool für Java. JDepend durchläuft Klassen und generiert Softwarearchitekturmetriken für jedes Package. Mit JDepend kann die Qualität einer Architektur hinsichtlich seiner Erweiterbarkeit, Wiederverwendbarkeit und Wartbarkeit automatisch gemessen werden, um Package-Dependencies effektiv zu verwalten und zu kontrollieren.

3.3.1 Metriken

JDepend analysiert die folgenden Metriken:

- **Anzahl Klassen und Interfaces**: Die Anzahl der konkreten und abstrakten Klassen (und Interfaces) im Package
- **Anzahl konkreter Klassen (CC)**: Die Anzahl der konkreten Klassen im Package
- **Anzahl abstrakter Klassen (AC)**: Die Anzahl der abstrakten Klassen (und Interfaces) im Package
- **Afferent Couplings (Ca)**: Die Anzahl anderer Packages, die von Klassen innerhalb des Package abhängen
- **Efferent Couplings (Ce)**: Die Anzahl anderer Packages, von denen die Klassen im Package abhängen
- **Abstractness (A)**: Das Verhältnis der Anzahl der abstrakten Klassen (und Interfaces) im analysierten Package zur Gesamtzahl der Klassen im analysierten Package

- **Instability (I)**: Das Verhältnis der Efferent Couplings (Ce) zur Summe der Couplings (Ce + Ca)
- **Distance from the Main Sequence (D)**: Der senkrechte Abstand eines Packages von der "main sequence line"
- **Package Dependency Cycles**: Listet Packages mit zyklischer Abhängigkeit auf.

3.3.2 Reports

JDepend bietet Reports als textuelle oder XML-basierte Dateien oder auch als grafische Benutzeroberfläche.

Grafische Benutzeroberfläche

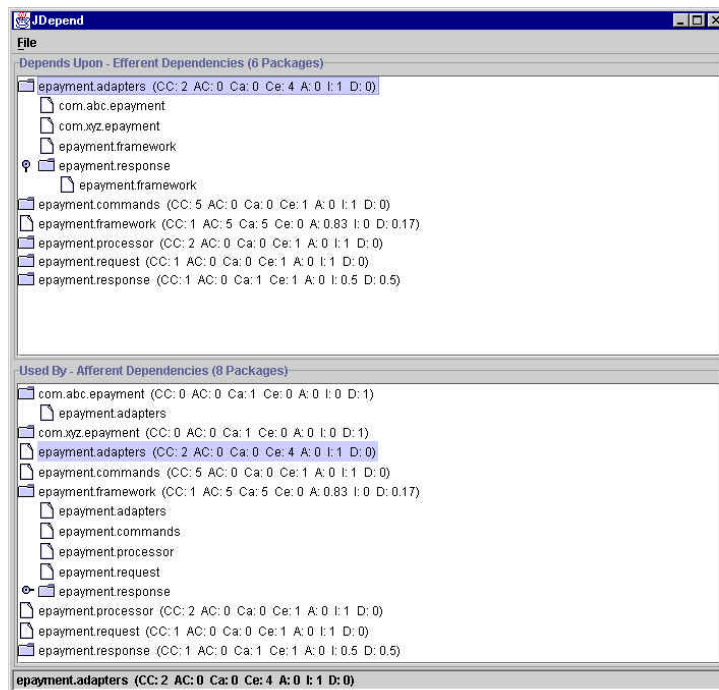

Abbildung 3.10: JDepend - UI

In der Abbildung 3.10 wird das Standard UI von JDepend gezeigt.

XML-basierter Report

```xml
<?xml version='1.0'?>
<JDepend>
    <Packages>
        <Package name='ch.soobr.cockpit.service.identity'>
            <Stats>
                <TotalClasses>7</TotalClasses>
                <ConcreteClasses>6</ConcreteClasses>
                <AbstractClasses>1</AbstractClasses>
                <Ca>2</Ca>
                <Ce>31</Ce>
                <A>0.14</A>
                <I>0.94</I>
                <D>0.08</D>
                <V>1</V>
            </Stats>

            <AbstractClasses>
                <Class sourceFile='VerificationProvider.java'>
                    ch.soobr.cockpit.service.identity.VerificationProvider
                </Class>
            </AbstractClasses>

            <ConcreteClasses>
                <Class sourceFile='IdentityService.java'>
                    ch.soobr.cockpit.service.identity.IdentityService
                </Class>
                <Class sourceFile='IdentityServiceTest.java'>
                    ch.soobr.cockpit.service.identity.IdentityServiceTest
                </Class>
                <Class sourceFile='VerificationService.java'>
                    ch.soobr.cockpit.service.identity.VerificationService
                </Class>
                <Class sourceFile='VerificationService.java'>
                    ch.soobr.cockpit.service.identity.VerificationService$1
                </Class>
                <Class sourceFile='VerificationSmsProvider.java'>
                    ch.soobr.cockpit.service.identity.VerificationSmsProvider
                </Class>
                <Class sourceFile='VerificationTotpProvider.java'>
                    ch.soobr.cockpit.service.identity.VerificationTotpProvider
                </Class>
            </ConcreteClasses>
```

Abbildung 3.11: JDepend - XML

In der Abbildung 3.11 ist ein Ausschnitt aus einem JDepend XML-Report zu sehen.

[8]

4 Fazit

Für die Analyse von nicht funktionalen Anforderungen in Bezug auf Softwarearchitektur gibt es diverse Metriken. Die meisten davon beziehen sich auf die Abstraktion und das Coupling von Klassen. Dabei gibt es nicht eine perfekte Klasse, unterschiedliche Werte werden erwartet für verschiedene Arten von Klassen.

Die drei untersuchten Tools zur automatisierten Auswertung von Softwarearchitektur-Metriken haben alle ihre Vor- wie Nachteile.

CodeScene gibt sehr einfach verständliche Aussagen über die Qualität der Softwarearchitektur und des Codes wieder. Zudem kann das Tool einfach in eine CI/CD-Pipeline eingebunden und somit eine automatisierte Auswertung vorgenommen werden. Leider sind die meisten Werte des Tools eine Mischung aus diversen Metriken, weshalb es nicht möglich ist genaue Aussagen über spezifische Aspekte der Softwarearchitektur zu machen.

JDepend ist ein sehr minimalistisches Tool, dass sich auf wenige Metriken konzentriert. Die von JDepend erfassten Werte beziehen sich stark auf die von Robert Martin vorgeschlagen Metriken rund um die "main sequence line". Leider ist die Nutzung des Tools eher beschwerlich und die Visualisierung für grössere Projekte kaum nutzbar.

CodeMR ermöglicht die einfache Visualisierung vieler Softwarearchitektur-Metriken. Wodurch ein schneller Verständnisaufbau der Architektur ermöglicht wird. Jedoch ist das Tool kaum geeignet für automatisierte Auswertung in einer CI/CD-Pipeline, da die Auswertungen manuell überprüft und interpretiert werden müssen.

Eigenständigkeitserklärung

Hiermit erkläre ich, dass ich die hier vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Sämtliche Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, habe ich als solche kenntlich gemacht.

Hiermit stimme ich zu, dass die vorliegende Arbeit in elektronischer Form mit entsprechender Software überprüft wird.

30. Mai 2022

J. Burkhalter

Literaturverzeichnis

[1] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[2] Victoria Puzhevich. scand: Functional vs non-functional requirements: The definitive guide.

[3] Paula Rome. perforce: What are non functional requirements — with examples.

[4] Artur Ogonowski. future-processing: Object-oriented metrics by robert martin.

[5] Aivosto Oy. aivosto: Chidamber and kemerer object-oriented metrics suite.

[6] codemr: Measure, visualise and improve your code quality.

[7] codescene: See the invisible. know your priority.

[8] Mike Clark and Pascal-Louis Perez. github - jdepend.

Abbildungsverzeichnis

2.1 Efferent Coupling 5
2.2 Afferent Coupling 6
2.3 Instability 7
2.4 Abstractness 8
2.5 Normalized distance from Main Sequence 9

3.1 CodeMR - Metric Distribution Diagramm 15
3.2 CodeMR - Package Übersicht 16
3.3 CodeMR - Package Dependencies Diagramm 17
3.4 CodeMR - Graph Diagramm 18
3.5 CodeMR - Metrik Legende 18
3.6 CodeMR - Detailmetrik 19
3.7 CodeScene - Hotspots 20
3.8 CodeScene - Code Health 21
3.9 CodeScene - Change Coulping 21
3.10 JDepend - UI 23
3.11 JDepend - XML 24