



Fakultät für Informatik

Studiengang Software- und Systems-Engineering

Erkennung von Design Patterns in Quellcode durch Machine Learning

Master Thesis

von

Mehmet Aslan

Datum der Abgabe: tt.mm.jjjj

Erstprüfer: Prof. Dr. Marcel Tilly

Zweitprüfer: Prof. Dr. Kai Höfig

EIGENSTÄNDIGKEITSERKLÄRUNG / DECLARATION OF ORIGINALITY

Hiermit bestätige ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken (dazu zählen auch Internetquellen) entnommen sind, wurden unter Angabe der Quelle kenntlich gemacht.

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Rosenheim, den tt.mm.jjjj

Vor- und Zuname

Kurzfassung

text

Schlagworte:

Inhaltsverzeichnis

1	Motivation	1
1.1	Einführung in Design Patterns	2
1.2	Untersuchungsfragen	3
1.3	Übersicht des Prozesses	3
2	Literaturrecherche	4
2.1	Design Patterns in der Software-Entwicklung	5
2.1.1	Design Pattern Katalog	6
2.1.2	Rollenkatalog	7
2.2	Herausforderungen und Probleme bei der Erkennung von Design Patterns	8
2.2.1	Variabilität der Implementierung	8
2.2.2	Steigende Komplexität von Software-Systemen	8
2.2.3	Iterative Evolution des Quellcodes	9
2.2.4	Mangel an expliziter Dokumentation	9
2.3	Maschine Learning	10
2.3.1	Metriken für Evaluierung von Modellen	10
2.3.2	Validierung und Optimierung von Modellen	11
2.3.3	Betrachtete Klassifizierer	14
2.4	Angewandte Ansätze für die Erkennung von Design Patterns	19
2.4.1	Graphen-basierende Ansätze	19
2.4.2	Maschine Learning Ansätze	22
2.4.3	Query Ansätze	26
2.4.4	Diverse Ansätze	27
3	Methodologie	28
3.1	Verfügbare Datensätze	29
3.2	Extrahierte Features	30
3.3	Betrachtete Multi-Klassifizierer für Rollen	31
3.4	Training und Hyperparameter-Tuning des Klassifizierers	32
3.5	Evaluation des trainierten Modells	33
3.6	Ermittlung übereinstimmendes Design Patterns	34
3.7	Diskussion der Methodologie	35
4	Implementierung	36
4.1	Präprozessierung des Datensatzes	37
4.2	Extraktion von Features aus Quellcode	38
4.3	Analyse des Datensatzes	39
4.4	Training und Hyperparameter-Tuning	40
4.5	Bestimmung des Design Patterns	41
4.6	Evaluation der Ergebnisse	42

5	Schluss	43
5.1	Fazit	44
5.2	Zukünftige Aussichten	45
A	Erstes Kapitel des Anhangs	46
	Literaturverzeichnis	47

Abbildungsverzeichnis

2.1 Kreuzvalidierung mit $n = 5$	11
2.2 Grafische Darstellung einer SVM im zwei-dimensionalen Raum	14
2.3 Hyperplanes mit Margen	15
2.4 Nicht mögliche Einteilung durch lineare Hyperplanes	16
2.5 Transformierter Datensatz aus Abbildung 2.4 durch RBF	16
2.6 Verteilung der Kategorien der Ansätze für die Erkennung von Design Patterns in Quellcode	19
2.7 Referenzklassendiagramm des Bridge Patterns mit Submustern	21
2.8 <i>Colored UML</i> für eine Mikroarchitektur	22
2.9 Ausschnitt an Annoation von Guhram et al.	26

Tabellenverzeichnis

2.1 Metriken und Maße für Rollen nach Uchiyama et al.	24
2.2 Extrahierte Features für DPD_F	25

1 Motivation

1.1 Einführung in Design Patterns

Entwurfsmuster oder auf Englisch ‘Design Patterns’ sind bewährte Lösungsansätze für wiederkehrende Probleme, die bei der Konzeption der Software-Architektur oder während der Implementierung der Software eingesetzt werden kann. Dabei dienen diese Entwurfsmuster als eine Art Blaupause, die es Software-Entwicklern ermöglicht, erprobte Lösungsstrategien für häufig auftretende Probleme in der Software-Entwicklung anzuwenden. Durch den Einsatz von etablierten Entwurfsmustern können Software-Entwickler für die Software bei korrekter Anwendung unter anderem erhöhte Wartbarkeit, Wiederverwendbarkeit von Komponenten, Verständlichkeit und Skalierbarkeit ermöglichen das wiederum in qualitativ besserer Software resultiert. Dabei sollte beachtet werden, dass Design Patterns als Vorlage zu betrachten sind. Je nach Einsatzgebiet muss die Anwendung des Entwurfsmusters evaluiert und für den konkreten Fall individualisiert werden. Deshalb existiert keine universelle anwendbare Iteration eines Design Patterns, die unabhängig von Anwendungskontext eingesetzt werden kann. Dies resultiert in variierenden Anwendung von Entwurfsmustern abhängig von jeweiligem Einsatzgebiet. Im weiteren Entwicklungszyklus der Software werden durch neue oder geänderte Anforderungen bereits eingesetzte Implementierungen von Entwurfsmustern modifiziert, entfernt oder neue werden hinzugefügt. Währenddessen besteht die Gelegenheit, dass durch mangelnder Dokumentation oder anderer Gründe die Entscheidungen, weshalb Entwurfsmuster so eingesetzt sind wie es eingesetzt worden, verloren gehen. Dadurch besteht die Gefahr, dass angewendete Design Patterns im weiteren Verlauf derer Entwicklung nicht mehr wiederzuerkennen sind. Aus diesem Grund ist die Etablierung eines Prozesses von Vorteil, das in der Lage ist, Implementierungen von Entwurfsmustern aus einem Software-System zu extrahieren und dieses konkret benennen. Vor allem der Einsatz von Maschine Learning für die Klassifizierung ist hier vorteilhaft, wodurch das Potenzial besteht, vorher nicht gesehene Implementierung von Design Patterns zu erkennen. Durch solch einen Prozess können durch die Erkennung von eingesetzten Entwurfsmustern auf konkrete und verlorenen gegangene Design-Entscheidungen zurückgeschlossen werden, welche zukünftige Design-Entscheidungen für das Software-System beeinflussen können. Der Fokus dieser Arbeit besteht daran, solch ein Prozess zu etablieren, welches für ein gegebenes Set von Quellcode-Dateien mithilfe von Maschine Learning einem potenziellen Entwurfsmuster zuzuteilen.

1.2 Untersuchungsfragen

Das Ziel dieser Arbeit besteht aus der Etablierung eines Prozesses, womit durch Einsatz von Maschine Learning für ein Set von Quellcode-Dateien ein Design Pattern zuzuordnen. Hierfür dienen eine Menge an Quelldateien als Eingabe für den Prozess und durch phasenweiser Transformationen und Bearbeitungen sollen ein möglichst passendes aus dem in Kontext dieser Arbeit betrachteten Entwurfsmusters zugeordnet werden. Um solch ein Prozess zu entwickeln, werden in Kontext dieser Arbeit folgende Fragen beantwortet:

RQ1. Welche Design Patterns werden berücksichtigt?

RQ2. Was für ein Datensatz eignet sich für solch ein Prozess?

RQ3. Wonach wird exakt klassifiziert?

RQ4. Welche Merkmale, die aus Quellcode-Dateien extrahierbar sind, eignen sich für Klassifizierung durch Maschine Learning Modelle?

RQ5. Welche Klassifizierer eignen sich?

RQ6. Wie ist das Endresultat zu beurteilen?

1.3 Übersicht des Prozesses

2 Literaturrecherche

2.1 Design Patterns in der Software-Entwicklung

Entwurfsmuster definieren gängige Lösungsblaupausen für häufig auftretende Probleme in Software-Entwicklung, vor allem in der Design-Phase der Architektur des Software-Systems als auch während der konkreten Implementierung. Jedoch sind diese als Schablone zu verstehen, die für den jeweiligen Einsatzfall angepasst werden müssen. Ein Werk, das das Verständnis von Design Patterns für das objektorientierte Programmieren maßgeblich geprägt ist, ist das von Gamma et al. verfasste Werk "Design Patterns: Elements of Reusable Object-Oriented Software". In diesem wird ein Katalog von 23 Entwurfsmustern definiert, welche in drei Kategorien aufgeteilt. Dieser Katalog wird von Software-Entwicklern als "Gang of Four"-Entwurfsmuster bezeichnet. Gamma et al. definieren folgende Elemente für die Identifikation eines Entwurfsmusters:[Gam94, S. 3]

- **Pattern Name:** Der Name des Entwurfsmusters beschreibt in wenigen Worten, welches das zu lösende Problem, die Lösung und welche Folgen dessen Einsatz mit sich bringt. Durch die Einführung eines Bezeichners wird eine Schicht der Abstraktion hinzugefügt, welches das Verständnis und Dokumentation des Design Patterns vereinfacht.
- **Problem:** Das Problem beschreibt, wo das Entwurfsmuster angewendet werden soll. Dabei kann es sich um ein konkretes Entwurfsproblem, Klassen- oder Objektstrukturen oder eine Liste von Bedingungen darstellen, die zu erfüllen sind.
- **Solution:** Das Lösungselement beschreibt die Beziehungen, Verantwortlichkeiten und Zusammenarbeit der einzelnen Elemente, die die Struktur des Design Patterns definieren. Dabei werden diese Elemente in Objekte und Klassen, die die Grundbausteine der objektorientierten Programmierung repräsentieren, aufgeteilt und deren Interaktionen miteinander stellen die Verantwortlichkeiten und Beziehungen dar.
- **Consequences:** Die Folgen diskutieren, wie der Einsatz des betrachteten Entwurfsmusters sich auf das Software-System einwirkt und welche Vor- und Nachteile dadurch resultieren. Diese beeinflussen unter anderem die Zeit- und Speicherkomplexität, Erweiterbarkeit, Flexibilität und Portabilität des Software-Systems.

Im Kontext dieser Arbeit werden zu klassifizierende Strukturen, die potenziell einem Design Pattern zugeordnet werden können, als Mikroarchitekturen bezeichnet, die aus einer Menge von interagierenden Komponenten bestehen, denen je eine Rolle zugewiesen wird. Die jeweilige Rolle beschreibt, welche Funktionalität und Verantwortung diese im Kontext der Mikroarchitektur übernimmt und wie diese mit anderen Komponenten interagiert. Als Komponenten mit Rollen werden hier konkrete bzw. abstrakte Klassen oder Schnittstellen definiert, die die erforderliche Rolle im Rahmen der Mikroarchitektur erfüllen.

Im weiteren Verlauf dieser Sektion werden die drei erwähnten Entwurfsmusterkategorien erläutert und zu dem werden im Kontext dieser Arbeit betrachte spezifische Design Patterns genauer betrachtet.

2.1.1 Design Pattern Katalog

Creational Design Patterns

Die Kategorie der Creational Design Patterns oder Erzeugungsentwurfsmuster beschäftigt sich mit der Abstraktion des Prozesses der Initialisierung[Gam94, S. 81]. Entwurfsmuster dieser Kategorie fokussieren sich auf die Unabhängigkeit wie Objekte erstellt, zusammengesetzt und repräsentiert werden. Die Entwurfsmuster dieser Kategorie mit Fokus auf Klassen nutzen den Mechanismus der Vererbung, um zu beeinflussen, wie Komponenten instantiiert werden, während dahingegen Design Patterns mit einem Fokus auf Objekten die Instantiierung auf andere Objekte delegieren. Creational Design Patterns werden dann bedeutend, wenn mit steigender Komplexität des Software-Systems sich von Vererbung distanziert wird und die Komposition aus einzelnen definierten Objekt mehr an Bedeutung gewinnt[Gam94, S. 81]. Dabei wird das Verhalten einer Komponente auf eine Menge von einzelnen kleinere Objekten delegiert und durch Zusammensetzung innerhalb der Komponente und deren Interaktion das erwünschte Verhalten erzeugt. Dadurch wird die Instantiierung von Software-Komponenten komplexer, da die Instantiierung von mehreren Objekten koordiniert werden muss. Creational Design Patterns liefern hierbei Hilfestellung, weil die exakte Komposition der konkreten Objekte, die Teil der zu instantiierenden Komponente sind, und der exakte Prozess der Instantiierung im Inneren des Entwurfsmusters verborgen werden. Nach außen hin sind dahingegen nur die Schnittstellen sichtbar, die die Komponente zur Verfügung stellt, während dessen interne Logik die Ausführung auf andere Objekte delegiert. Im Kontext dieser Arbeit werden folgende Entwurfsmuster aus der Kategorie der Creational Design Patterns betrachtet:

Structural Design Patterns

Structural Design Patterns oder Strukturentwurfsmuster fokussieren sich darauf, wie einzelne Klassen und Objekte zusammengesetzt werden können, um größere Strukturen zu erzeugen[Gam94, S. 137]. Entwurfsmuster dieser Kategorie sind vorteilhaft, wenn unabhängig voneinander entwickelte Klassen oder Objekte aus verschiedenen Bibliotheken oder Frameworks miteinander interagieren müssen. Anstatt konkrete Implementierung und Schnittstellen zu nutzen, bedienen sich Structural Design Patterns der Komposition aus Objekten, um neue Funktionalitäten zur Verfügung zu stellen[Gam94, S. 137]. Die dadurch gewonnene Flexibilität ermöglicht das Ändern der Zusammensetzung des Objektes dynamisch zu der Laufzeit, welches mit statischer Komposition durch Klassen nicht möglich ist.[Gam94, S. 137]. Im Kontext dieser Arbeit werden folgende Entwurfsmuster aus der Kategorie der Structural Design Patterns betrachtet:

Behavioral Design Patterns

Behavioral Design Patterns oder Verhaltensentwurfsmuster konzentrieren sich auf Algorithmen und der Zuweisung von Verantwortlichkeiten zwischen Objekten[Gam94, S. 221]. Dabei wird nicht nur Struktur der Entwurfsmuster betrachtet, sondern auch die Kommunikation und Interaktion der Objekte, die Teil des Entwurfsmusters sind. Charakteristisch für Design Patterns dieser Kategorie ist der Fokus auf Verknüpfung der einzelnen Teilobjekte des Entwurfsmusters, anstatt des Kontrollflusses, welcher zur Laufzeit schwer nachvollziehbar sein kann[Gam94, S. 221]. Im Kontext dieser Arbeit werden folgende Entwurfsmuster aus der Kategorie der Behavioral Design Patterns betrachtet:

2.1.2 Rollenkatalog

2.2 Herausforderungen und Probleme bei der Erkennung von Design Patterns

In diesem Abschnitt der Arbeit werden mögliche Herausforderungen diskutiert, die bei dem Entwerfen des Prozesses für die Erkennung von Entwurfsmustern auftreten können.

2.2.1 Variabilität der Implementierung

Design Patterns stellen in der Software-Entwicklung bewährte Lösungsmuster für bereits begegnete Herausforderung dar. Aufgrund der abstrakten und wiederverwendbaren Natur der Entwurfsmuster, muss für diese eine konkrete Implementierung definiert werden, die von dem Einsatzfall, Kontext und anderen Faktoren wie verwendeter Programmiersprache, Bibliotheken und Erfahrungsstand des Software-Entwicklers. Dadurch, dass jedes Entwurfsmuster einen konzeptionellen Rahmen darstellt und jede Implementierung von nicht statischer Außenfaktoren beeinflusst wird, resultiert dies in einem breiten Spektrum an Implementierungen für ein gegebenes Entwurfsmuster. Aus diesem Grund ist eine Definition einer starren Definition eines Design Patterns, was als Startpunkt und Referenz für die Erkennung des jeweiligen Entwurfsmusters dienen könnte, nicht möglich. Deshalb ist eine definitive Antwort auf die Frage, ob eine betrachtete Mikroarchitektur eine Instanz eines Entwurfsmusters, nicht beantwortbar, weshalb die Antwort von automatisierten Prozessen von Design Patterns eher mit einem Wert besteht, welches die Ähnlichkeit zu einem Design Pattern beschreibt. Um einen zufriedenstellenden Wert für diese Frage zu liefern, bedarf es eines breiten Spektrums an Implementierungsvariationen als Referenz für die Erkennung.

2.2.2 Steigende Komplexität von Software-Systemen

Die steigende Komplexität von Software-Systemen stellt eine erhebliche Herausforderung bei der Erkennung von Entwurfsmustern in Entwurfsmustern dar. Dies ist besonders bei langjährigen Software-Projekten der Fall, an denen über die Zeit konstante Änderungen wegen Wartung und neuen bzw. geänderten Anforderungen unterliegen. Diese Art von Software-Projekten tendieren dazu, dass mit der Zeit deren Komplexität zunimmt [Suh10, S. 7]. Bei kleineren Software-Projekten mit geringem Umfang und Komplexität sind Entwurfsmuster leichter zu erkennen und zu implementieren. Dahingegen bei langjährigen Software-Projekten steigt mit wachsender Gesamtkomplexität die Komplexität der angewendeten Entwurfsmuster in deren Quellcode, wodurch die Identifizierung dieser proportional mitsteigt. Entwurfsmuster werden durch diese Entwicklung weiter modifiziert und angepasst, womit diese von der ursprünglichen leichter zu identifizierbaren Iterationen weiter abweichen. Deshalb beinhaltet die Erkennung von Entwurfsmustern nicht nur auf die momentane Iteration, sondern auch die Erfassung derer historischen Evolution und die Entwicklung dieser innerhalb der Codebasis.

2.2.3 Iterative Evolution des Quellcodes

Damit ein Software-System seine Anforderungen im Verlauf dessen Lebenszyklus in einer zufriedenstellenden Art und Weise erfüllen kann, muss dieses adaptieren, um diesen Anforderungen gerecht zu werden [Leh96, S. 108]. Dies hat zu Folge, dass dessen Quellcode iterativen Änderungen unterliegt. Ursprüngliche Implementierungen in der Codebasis werden analysiert und es wird überprüft, ob diese ihre Aufgaben zufriedenstellend erfüllen oder nicht. Falls nicht, werden diese so modifiziert, sodass diese Anforderungen auf erwartete Art und Weise erfüllt werden können. Implementierungen von angewandten Design Patterns als Teil des Quellcodes unterliegen ebenfalls dieser Analyse. Diese werden als Teil des Analyseprozesses genauer betrachtet und werden nach Bedarf modifiziert und angepasst. Diese Entwicklung führt wie das im vorherigen Abschnitt diskutierten Fall, dass Entwurfsmuster von ihrer ursprünglichen leichter zu identifizierbaren Iteration weiter abweichen und die Erkennung von Entwurfsmustern nicht nur die momentane Implementierung, sondern auch die historische Entwicklung berücksichtigen werden muss. In automatisierten Prozess der Erkennung von Entwurfsmustern kann dieser Aspekt nur bedingt berücksichtigt, weil das Einschließen der Historie der betrachtenden Implementierung und dessen Kontext in der Codebasis nicht pauschal und in einer allgemeinen Ansicht betrachtet werden kann.

2.2.4 Mangel an expliziter Dokumentation

Das Erstellen und Warten von Dokumentation für Software-Systeme ist eine Tätigkeit, die von Software-Entwicklern als wichtig eingestuft wird, jedoch in diese Tätigkeit relativ wenig Zeit investiert wird [Zhi15, S. 162]. Dies ist die Folge der Dominanz des agilen Software-Entwicklungsprozesses, in dieser die Verwendung von Zeit und Ressourcen für die Dokumentation eher als Verschwendung betrachtet wird, da diese keinen direkten Mehrwert für die Auslieferung des Software-Produktes an den Endkunden liefert [Zhi15, S. 159]. Das dies das Software-System komplett betrifft, sind Design Patterns in dessen Codebasis ebenfalls betroffen. Diese werden meist nicht direkt gekennzeichnet. Zwar kann durch Nomenklatur und Kontrollfluss indirekt Rückschlüsse auf die potenziellen Entwurfsmuster abgeleitet werden, jedoch erfordert dies konkretes Fachwissen und Erfahrungen, die nicht von jedem Software-Entwickler erfüllt werden kann. Bei automatisierten Prozessen für die Erkennung von Entwurfsmustern kann diese berücksichtigt werden, sollte aber nicht als alleiniger Faktor bei dem Identifikationsprozess dienen.

Aufgrund der Variation an Implementierungsmöglichkeiten, Änderungen im Quellcode und Mangel an Dokumentation ist die manuelle Identifikation von Entwurfsmustern in Quellcode ein Prozess dar, in der einen gewissen Grad an Mitdenken erfordert. Im nächsten Abschnitt der Arbeit werden bereits entwickelte Verfahren betrachtet, die das Mitdenken bis zu einem gewissen Grad automatisieren und dieses als Teil des Prozesses mitincludieren.

2.3 Maschine Learning

Der Fokus dieser Arbeit ist die Ermittlung von Design von Quellcode durch die Anwendung von Maschine Learning. Dieser Teil der Arbeit ist für die Erläuterung von verwendeten Metriken, angewandten Techniken und ausgewählter Klassifizierer gewidmet.

2.3.1 Metriken für Evaluierung von Modellen

Um die Leistung eines Klassifizierers zu ermitteln, bedarf es einer Menge an Metriken, womit beurteilt werden, ob der Klassifizierer wie erwartet performt. In Kontext dieser Aussage wird in der Domäne des Maschine Learnings bekannte Metriken aus dem Bereich der Statistik angewendet. In diesem Abschnitt wird eine Auswahl von solchen Metriken erläutert, die in der Literaturrecherche als auch in den Sektionen der Methodik und Implementierung erwähnt werden. Für die Ermittlung der Metriken werden folgende Größen definiert:

TP (True Positive): Anzahl echt positiver Klassifizierungen

TN (True Negative): Anzahl echt negativer Klassifizierungen

FP (False Postive): Anzahl falsch positiv Klassifizierungen

FN (False Negative): Anzahl falsch negativ Klassifizierungen

Anhand dieser Größen werden folgende Metriken definiert:

Accuracy (Genauigkeit): Dieser Wert gibt an, wie das Verhältnis zwischen den klassifizierten Beobachtungen zur Gesamtzahl der Beobachtungen ist.

$$Accuracy = \frac{\text{Anzahl der korrekten Vorhersagen}}{\text{Gesamtzahl der Vorhersagen}} = \frac{TP+TN}{TP+TN+FP+FN}$$

Precision (Präzision): Dieser Wert gibt das Verhältnis an, wie die korrekt positiv klassifizierten Beobachtungen zur Gesamtzahl der als positiv klassifizierten Beobachtungen stehen.

$$Precision = \frac{TP}{TP+FP}$$

Recall (Sensitivität): Dies ist das Verhältnis der positiv klassifizierten Beobachtungen zur Gesamtzahl der tatsächlichen positiven Beobachtungen.

$$Recall = \frac{TP}{TP+FN}$$

F1-Score: Der F1-Score ist das harmonische Mittel von Präzision und Recall und gibt ein besseres Maß für die unbalancierten Klassen als die Genauigkeit allein.

$$F1-Score = 2 * \frac{Precision * Recall}{Precision + Recall} = \frac{2 * TP}{2 * TP + FP + FN}$$

2.3.2 Validierung und Optimierung von Modellen

Neben dem Zusammentragen von relevanten Datenpunkte für die Erhebung eines geeigneten Datensatzes ist das Validierung und Optimierung des Maschine Learning Modells eines der relevanten Schritte im gesamten Prozess, um damit das Modell dessen Aufgabe möglichst zufriedenstellend erfüllen kann. In diesem Teil der Arbeit werden Techniken bzw. Methoden erläutert, die in der Validierungs- oder Optimierungsphase angewendet werden können.

Kreuzvalidierung von Modellen

Nach dem Trainieren des Modells für einen Datensatz ist von Interesse, wie das Modell neue unbekannte Datenpunkte umgeht und diese Leistung anhand einer Metrik zu messen. Dies stellt den Kernpunkt der Validierungsphase dar. Ein naiver Ansatz, die gleiche Menge an Datenpunkten zu verwenden, worauf das Modell in der Trainingsphase trainiert wurde. Dabei ist jeder Datenpunkt des Datensatzes dem Modell bereits bekannt, wodurch es nicht auf wirklich unbekannte Datenpunkte validiert. Eine Möglichkeit dem entgegenzukommen ist das Aufteilen des Datensatzes in Trainings- und Validationsdatensatz, wobei meist der Trainingsdatensatz die größere Menge an Datenpunkten zugesprochen wird. Jedoch besteht hier der Nachteil, das in der Trainingsphase die Datenpunkte in dem Validationsdatensatz wegfallen. Um diesen Nachteil zu negieren, kann die Technik der Cross Validation oder Kreuzvalidierung angewendet werden.

Hierbei wird zunächst eine natürliche Zahl n bestimmt, die angibt, in wie viele gleich große Teile der Datensatz aufgeteilt wird. Die Trainings- und Validierungsphase wird hierbei zu einem Schritt zusammengefasst und das Modell wird iterativ n -Mal trainiert und anhand der vorgegebenen Metrik evaluiert.

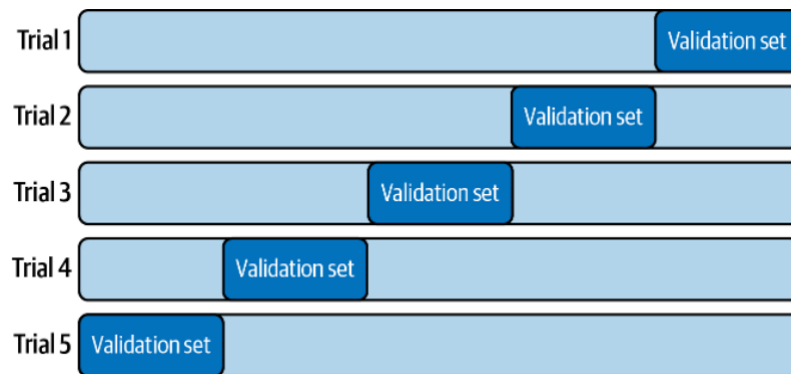


Abbildung 2.1 Kreuzvalidierung mit $n = 5$

Wie aus Abbildung 2.1 zu entnehmen ist, wird in der i -ten Iteration das $k - i$ -te Sektion des Datensatzes für die Validierung und restlichen für das Training des Modells verwendet. Das Resultat dieser Operation ist abhängig von Implementierung der Kreuzvalidierung die Iteration des Modells mit der besten Evaluation und eine Liste von Werten, die Leistung des Modells in der jeweiligen Iteration angibt.

Optimieren von Modellen durch Hyperparameter-Tuning

Maschine Learning Modelle können mithilfe von Parametern konfiguriert werden. Dabei wird zwischen zwei Arten von Parametern entschieden: Modellparameter und Hyperparameter. Modellparameter werden während der Trainingsphase automatisiert direkt durch den Trainingsdatensatz bestimmt und sind nicht von Außen zu manipulieren. Dahingegen bestimmen Hyperparameter unter anderem wie sich das Modell während der Trainingsphase zu verhalten und beeinflussen dessen Architektur. Jedes Modell definiert hierbei selbst, welche Hyperparameter verfügbar sind. Hyperparameter werden während der Instantiierung des Modells definiert. Die Ermittlung der bestmöglichen Kombinationen an Hyperparametern-Werten ist hilfreich, falls Leistung des Modells nicht den Erwartungen entspricht und das verwendete Datensatz nicht weiter argumentiert werden kann. Um die Leistung des Modells zu evaluieren, wird wie in der Kreuzvalidierung eine Metrik bestimmt, die Leistung der jetzigen Iterationen zu beurteilen. Das Ziel der Hyperparameter-Optimierung besteht hier darin, eine Kombination an Hyperparametern-Werten zu bestimmen, wodurch die Metrik maximiert bzw. minimiert. Dies kann entweder manuell durchgeführt oder es werden Methoden eingesetzt, die dies automatisiert erledigen können. Im Folgendem wird eine Auswahl an automatisierten Strategien aufgelistet und erläutert:

Random Search/Grid Search: In diesen Methoden werden zunächst Wertebereiche für die jeweiligen Hyperparameter bestimmt. In der Regel wird hierbei eine Menge an diskreten Werten bestimmt, die der jeweilige Hyperparameter annehmen kann. Jede Permutation der Hyperparameter-Konfiguration wird als eine Zelle in einem Gitter erfasst. Jede besuchte Zelle dient dabei als Konfiguration für eine neue Instanz des Modells, welches im weiteren Verlauf durch Einsatz von Kreuzvalidierung trainiert und evaluiert wird. Das Ergebnis der Evaluation wird vermerkt. Bei Random Search wird zufällig bestimmt, welche Zellen besucht werden, während bei Grid Search alle Zellen besucht werden. Das Ergebnis ist die Konfiguration des Modells mit der besten Evaluierung.

Bayessche Optimierung: Bei Grid bzw. Random Search wird jede Permutation der Hyperparameter-Werte eine Instanz des Modells trainiert und evaluiert. Unter Umständen kann der Trainingsprozess je nach Modell und Datensatz rechenintensiv, wodurch es lange dauern, bis diese Methoden ein Resultat liefern. In solch einem Fall kann das Verfahren der Bayesschen Optimierung angewendet, um die Dauer der Evaluierung zu verkürzen. Im Kern wird das Maschine-Learning Modell als Black Box Funktion interpretiert, welches als Eingabe eine Permutation an Hyperparameter-Werten akzeptiert und der Wert der Zielmetrik für die jeweilige Permutation dient als Rückgabewert der Black Box Funktion. Dadurch, dass die Optimierung der Black Box Funktion durch Ableitungsverfahren nicht möglich, wird dieses eine Ersatzfunktion substituiert, welches die Black Box Funktion approximiert. Bei der Ersatzfunktion handelt es sich um ein probabilistisches Modell, welches genutzt wird, um eine Wahrscheinlichkeitsverteilung über die möglichen Werte der Zielmetrik für verschiedene Kombinationen an Hyperparameter-Werten zu erstellen. Die Bayessche Optimierung beginnt typischerweise mit einer zufälligen Auswahl von Hyperparametern, um einige Datenpunkte zu generieren. Diese werden verwendet, um das probabilistische Modell zu initialisieren. Anschließend wird iterativ die Akquisitionsfunktion angewendet, um neue und vielversprechende Hyperparameter-Kombinationen zu identifizieren und zu testen. Diese Funktion bestimmt, welche Permutation am Hyperparameter-Werten auf Basis

des jetzigen Standes des probabilistischen Modells als Nächstes bewertet werden soll. Sie balanciert die Exploration unbekannter Bereiche des Hyperparameterraums mit der Exploitation von Bereichen, die voraussichtlich zu besseren Ergebnissen führen. Nach jedem Schritt wird das Modell mit den neuen Ergebnissen aktualisiert, was zu einer kontinuierlichen Verbesserung der Schätzungen und Entscheidungen führt. Am Ende der Methode ist die Kombination an Hyperparameter-Werten bekannt, welches für die Zielmetrik das globale Minimum bzw. Maximum als Rückgabewert zurückgibt.

Tree-structured Parzen Estimators (TPE): Bei TPEs handelt es sich um einen speziellen Anwendungsfall der Bayesschen Optimierung. Anstatt eine Wahrscheinlichkeitsverteilung für die Kombination an Hyperparameter-Werten zu verwenden, wird diese in zwei aufgeteilt: eine für Kombinationen, die zu besseren Werten für die Zielmetrik führen ("gute Verteilung") und eine für solche, die zu schlechteren Ergebnissen führen ("schlechte Verteilung"). Wie bei der normalen Bayesschen Optimierung wird eine Aquisefunktion eingesetzt, um zu bestimmen, welche Kombination an Hyperparameter-Werten als Nächstes evaluiert werden soll. Hierbei basiert die Akquisitionsfunktion bei TPE auf das Verhältnis der Dichten dieser beiden Verteilungen. Durch den Einsatz von Parzen Estimatoren werden für beide Wahrscheinlichkeitsverteilungen Dichtefunktionen approximiert, wodurch die Dichten in Akquisitionsfunktion bestimmt werden können. Sie wählt die nächste zu evaluierende Hyperparameter-Kombination, indem sie Bereiche bevorzugt, in denen das Verhältnis der Dichte der "guten" Verteilung zur Dichte der "schlechten" Verteilung hoch ist. Wie in der normalen Bayesschen Optimierung wird das probabilistische Modell iterativ mit neuen Werten aktualisiert, um ein globales Minimum bzw. Maximum der Black Box Funktion zu ermitteln.

2.3.3 Betrachtete Klassifizierer

Für die Bestimmung eines Design Patterns wird eine Menge an Rollen bestimmt, die die Submuster innerhalb des jeweiligen Entwurfsmusters erfüllen. Die Bestimmung der Rollen für einzelne Submuster fällt in dem Bereich des Machine-Learnings in den Bereich der Klassifikation. Im Sinne dieser werden in diesem Abschnitt der Arbeit eine Auswahl von Klassifizierern betrachtet und erläutert, die im Kontext dieser Arbeit in Betracht gezogen werden.

Support Vector Machines

Support Vector Machines oder SVMs sind überwachte Machine-Learning Modelle, die sowohl für Regressions- als auch für Klassifikationsaufgaben einsetzbar ist. Die Datenpunkte werden in einen n -dimensionalen Raum abgebildet, wobei n die Anzahl der Features der Datenpunkte beschreibt. Bei der Klassifizierung trennen SVMs den Raum der Datenpunkte in verschiedene Zonen. Jede Zone entspricht einem Label, zu welchen die Datenpunkte in der jeweiligen Zone zugeordnet werden. Die Grenzen der einzelnen Zonen werden durch Hyperplanes beschrieben. Das Ziel ist, diese Hyperplanes so zu bestimmen, sodass die Datenpunkte möglichst distinkt einer Zone zugeordnet werden.

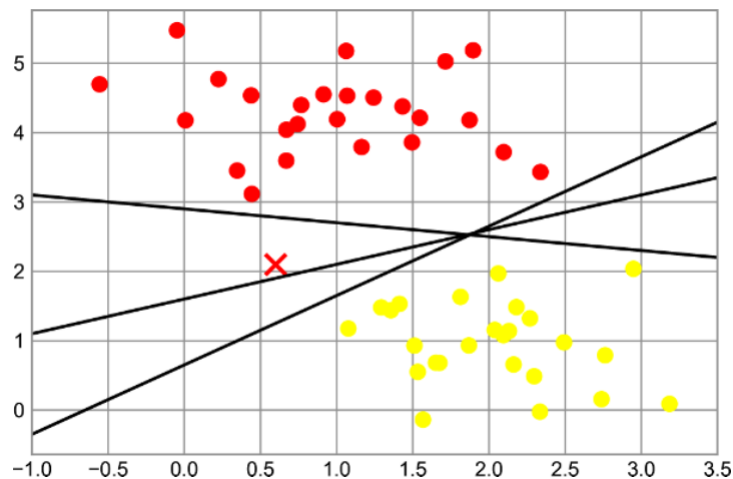


Abbildung 2.2 Grafische Darstellung einer SVM im zwei-dimensionalen Raum

Abbildung 2.2 grafisch, wie eine SVM funktioniert. Punkte sind die Datenpunkte des Datensatzes und die Farben der Punkte die Klasse, in der die Punkte eingeordnet sind. Die Linien beschreiben die mögliche Hyperplanes, womit Grenzen zwischen den Klassen determiniert wird. Das Kreuz zeigt einen neuen Datenpunkt, der zu klassifizieren ist. Mit diesem neuen Datenpunkt stellt sich die Frage, welche der drei möglichen Hyperplanes zu wählen ist, um die Datenpunkte möglichst distinkt einer Klasse zuzuordnen.

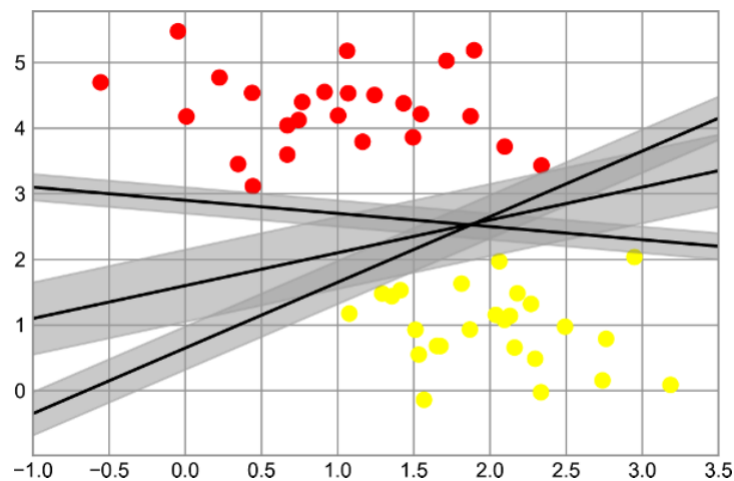


Abbildung 2.3 Hyperplanes mit Margen

Wie in Abbildung 2.3 zu sehen ist, wird für jede Hyperplane die Marge zwischen der Hyperplane und der Datenpunkte und den nächsten Datenpunkten bestimmt. Die Hyperplane mit der größten Marge, die mit der größten Distanz zwischen sich und den nächsten Datenpunkten, wird als finale Hyperplane für die Grenzbildung zwischen den Klassen gewählt. Im Falle von Abbildung 2.3 ist dies mittlere von den drei Kandidaten. Mit jedem neuen Datenpunkt wird während Trainings wird die ideale Hyperplane neu bestimmt. Jedoch stellt sich die Frage, wie mit Datensätzen umzugehen, die nicht durch lineare Hyperplanes in Klassen aufgeteilt werden.

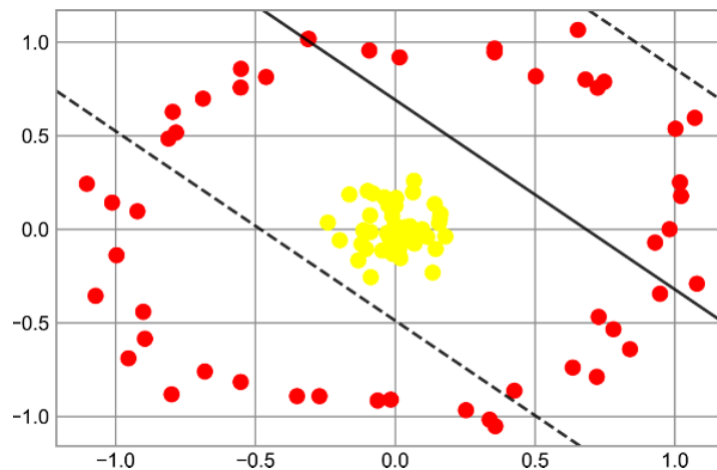


Abbildung 2.4 Nicht mögliche Einteilung durch lineare Hyperplanes

Abbildung 2.4 zeigt einen Datensatz, dessen Datenpunkte nicht distinkt durch lineare Hyperplanes zugeordnet werden können. Um trotzdem lineare Hyperplanes zu bilden, wird der bei SVMs Kernel verwendet. Bei dem Kernel Trick handelt es sich um eine Funktion, die den Datensatz in einen Raum mit einer höheren Dimension projiziert. Durch die Projektion in einem Raum mit einer höheren Dimension ist es möglich, lineare Hyperplanes zu bestimmen. Diese Funktion werden als Kernel bezeichnet und werden bei der Instantiierung des Modells als Hyperparameter mitgegeben.

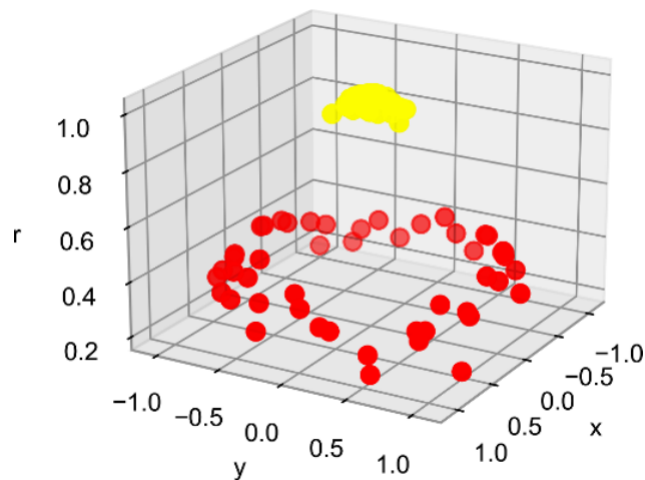


Abbildung 2.5 Transformierter Datensatz aus Abbildung 2.4 durch RBF

Abbildung 2.5 zeigt, wie durch Anwendung des Radial-Basis-Function Kernel, das den ursprünglich zwei-dimensionalen Raum auf ein drei-dimensionales projiziert wird. Durch die Transformation kann nun eine Hyperplane als Ebene bestimmt werden, wodurch die Klassifikation ermöglicht wird.

k-Nearest Neighbor Classifier

k-Nearest Neighbors Classifier oder KNN kann wie die SVM für Regressions- als auch für Klassifikationsprobleme eingesetzt werden. Dabei wird bei KNN unter der Annahme agiert, dass ähnliche Datenpunkte in der Nähe zueinander sind. Der Hyperparameter k definiert die Anzahl der nächsten Nachbarn, die verwendet werden, um den neu eingefügten Datenpunkten eine Klasse zuzuordnen. Um die k nächsten benachbarten Datenpunkte zu identifizieren, ist ein Maß erforderlich, welches die Distanz zwischen diesen beschreibt. Dieses wird als Hyperparameter bei der Instantiierung des Modells diesem übergeben. Folgende Größen können als Metrik für die Distanz verwendet werden:

Euklidische Distanz ist die Länge des Liniensegmentes zwischen den beiden Punkten im euklidischen Raum.

$$d_{\text{euclidean}}(P, Q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \dots + (p_n - q_n)^2}$$

mit

$$P = (p_1, p_2, \dots, p_n)$$

$$Q = (q_1, q_2, \dots, q_n)$$

Manhattan Distanz ist die Summe der absoluten Differenz zwischen den Komponenten der Punkte.

$$d_{\text{manhattan}}(P, Q) = |p_1 - q_1| + |p_2 - q_2| + \dots + |p_n - q_n|$$

mit

$$P = (p_1, p_2, \dots, p_n)$$

$$Q = (q_1, q_2, \dots, q_n)$$

Minkowski Distanz ist eine allgemeinere Metrik, um die Distanz zwischen zwei Punkten im n -dimensionalen Raum zu ermitteln.

$$d_{\text{minkowski}}(P, Q) = (|p_1 - q_1|^p + \dots + |p_n - q_n|^p)^{\frac{1}{p}}$$

mit

$$P = (p_1, p_2, \dots, p_n)$$

$$Q = (q_1, q_2, \dots, q_n)$$

p = Typ der zu kalkulierenden Distanz ($p = 1 \rightarrow$ Manhattan Distanz, $p = 2 \rightarrow$ Euklidische Distanz)

Die Klassifizierung des neuen Datenpunktes erfolgt im Kontext der k nächsten benachbarten Datenpunkte. Die Klasse, welche am häufigsten in den k benachbarten Punkten vorkommt, wird dem neuen Datenpunkt zugewiesen.

Random Forest Classifier

2.4 Angewandte Ansätze für die Erkennung von Design Patterns

Bei der Erkennung von Design Patterns in Quellcode wurden verschiedene Verfahren entwickelt werden, die auf unterschiedlichen Methoden beruhen, um das gesetzte Ziel zu erreichen. Yarahmadi et al. führten in ihrer Arbeit eine Untersuchung über die Methoden, die angewandt worden, um Design Patterns in Quellcode zu erkennen, und kategorisierten diese [Yar20, S. 5805].

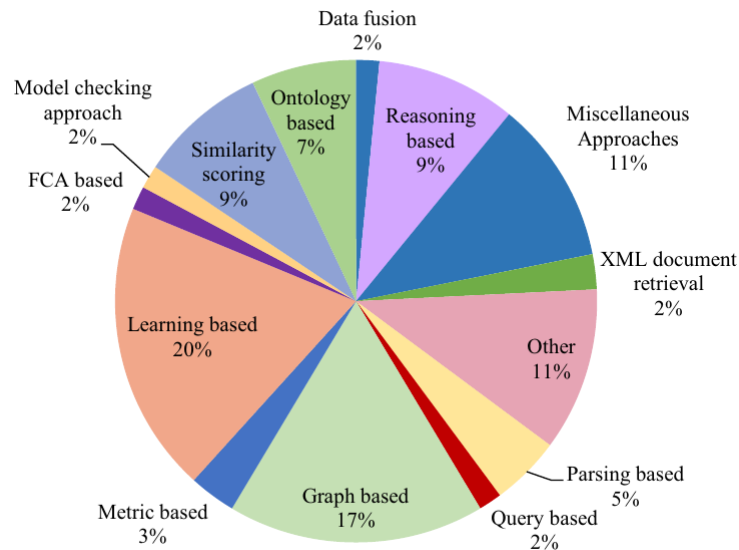


Abbildung 2.6 Verteilung der Kategorien der Ansätze für die Erkennung von Design Patterns in Quellcode

Wie aus Figur 2.6 zu entnehmen ist, wurden die entwickelten Prozesse von Yarahmadi et al. auf eine begrenzte Menge an Kategorien eingestuft. In dieser Sektion der Arbeit werden die vier größten Kategorien aus Figur 2.6 genauer erläutert und es werden exemplarisch Arbeiten diskutiert, die in die jeweilige Kategorie zugeordnet werden können.

2.4.1 Graphen-basierende Ansätze

Die Methodik der Reduktion stellt in der Berechenbarkeitstheorie einen Ansatz dar, um Lösungswege für neue unbekannte Probleme zu entwickeln. Dabei wird durch einen Algorithmus das unbekannte Problem in ein bereits gelöstes Problem, dessen Lösungsweg bereits vorhanden ist, umgewandelt. In Graphen-basierenden Methoden für die Erkennung von Design Pattern in Quellcode wird diese angewendet, um Quellcode in Graphen zu transformieren und diese Graphen werden als Eingabe für diverse Graphenalgorithmen verwendet.

Formell betrachtet ist ein Graph definiert als [Siu98, S. 9]:

$$G = \{V(G), E(G)\}$$

mit

G : der zu betrachtende Graph

$V(G)$: Nicht leere Menge von Knoten in G

$E(G)$: Menge an ungeordneten Tupeln von distinkten Elementen von $V(G)$

Der Quellcode selbst ist als roher Text zu betrachten, welches verschiedene Entitäten wie Klassen, Objekte und Schnittstellen beinhaltet, und definiert, wie diese miteinander interagieren. Als Graph G werden die Entitäten aus dem Quellcode als Knoten $V(G)$ und die Relationen und Interaktionen wie Vererbung oder Methodenaufrufe werden als Kanten $E(G)$ dargestellt. Hierbei stellt Unified Modeling Language (UML) eine in der Software-Entwicklung verbreitete Modellierungssprache dar und definiert verschiedene Arten von Graphen dar, um Software und andere Systeme zu modellieren. Die Diagramme aus der UML-Domäne werden in diesem Kontext als Graphen aufgefasst, da diese aus einer Menge aus Kanten und Knoten bestehen und anhand der obigen Definition als Graphen interpretiert werden können. Eine Diagrammart aus der Domäne, welches eingesetzt wird, um objektorientierte (OO) Software-Systeme zu modellieren, sind Klassendiagramme. Klassendiagramme beschreiben, wie Klassen und deren Relation zueinander im Kontext des Paradigmas der OO-Programmierung aufgefasst werden. Pradhan et al. nutzen Klassendiagramme als Eingabe für ihre entworfene Methode und generieren diese für das zu analysierende Software-System und Implementierung von Entwurfsmustern, die als Referenz genutzt werden [Pra15, S. 2]. Diese werden als gerichtete Graphen erfasst, wobei die Klassen als Knoten und die Assoziation wie Vererbung zwischen diesen als Kanten darstellen. Zusätzlich werden die Kanten je nach Art der Assoziation unterschiedlich gewichtet [Pra15, S. 2]. Im weiteren Verlauf werden mögliche Menge Kandidaten aus dem Graphen des Software-Systems als dessen Subgraphen durch die Eigenschaft der Graphenisomorphie extrahiert und anhand der normalisierten Kreuzrelation der Maß der Übereinstimmung bestimmt [Pra15, S. 3]. Graphenisomorphie beschreibt, ob zwei Graphen strukturell identisch sind, sodass jede Kante des einen Graphen einer Kante im anderen Graphen entspricht und umgekehrt [Siu98, S. 10]. Die normalisierten Kreuzrelation ist ein Maß, dessen Wertebereich zwischen 0.0 und 1.0 definiert ist. Je näher der Wert an 1.0, desto identischer sind die zwei Graphen. Zu der Evaluierung des Prozesses wurden vier Open Source Software-Systeme hergezogen, aus welchen fünf existierende Entwurfsmuster zu erkennen sind [Pra15, S. 6]. Dabei wurde von Pradhan et al. dokumentiert, ob ein Entwurfsmuster komplett oder partiell im Quellcode entdeckt wurde. Nach eigener Auswertung von Pradhan et al. wurden die als Referenz genommen Implementierung der Design Patterns mehrfach komplett als auch partiell in der Codebasis der zu dem Test hergezogen Software-Systeme identifiziert [Pra15, S. 6].

In ihrer Arbeit verfolgen Dongjin et al. in ihrer vorgeschlagenen Methodik ebenfalls die Erkennung von Entwurfsmustern der strukturellen Kategorie in Quellcode durch den Einsatz von Graphen. In Kontext dieser werden wie im vorherigen Verfahren beschriebenen Klassendiagramme als gerichtete gewichtete Graphen eingesetzt. Entitäten wie Klassen, Objekte oder Schnittstellen als Knoten und die Assoziation der Entitäten wie Vererbung als gerichtete gewichtete Kanten des repräsentiert [Yu13, S. 582]. Zudem werden Referenzimplementierung der zu identifizierenden Entwurfsmuster in gewichtete Klassendiagramme transformiert und innerhalb dieser werden Submuster definiert, die in Summe das Entwurfsmuster repräsentieren [Yu13, S. 580].

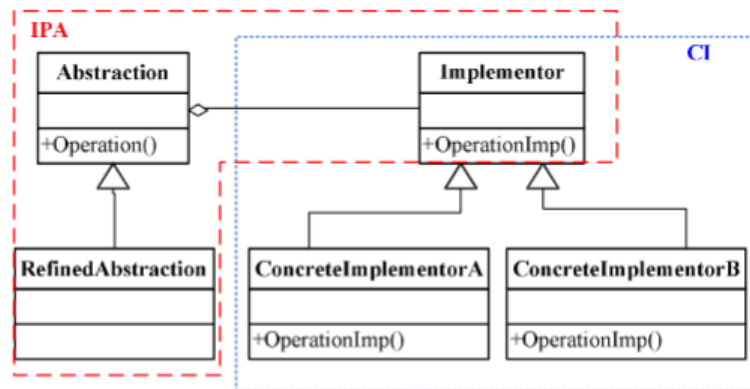


Abbildung 2.7 Referenzklassendiagramm des Bridge Patterns mit Submustern

Beispielhaft spiegelt die Abbildung 2.7 von Dongjin et al. definierte Referenzklassendiagramm für das Bridge Pattern wider. Innerhalb dieses Klassendiagramms werden die Submuster **IPA** und **CI** festgelegt. Wie von der Abbildung 2.7 zu entnehmen ist, bestehen die Submuster aus einzelnen Entitäten und deren Assoziationen. Innerhalb der Klassendiagramme des Quellcodes, welches als Eingabe dient, wird nach solchen Submustern durch Einsatz von Graphenisomorphie gesucht [Yu13, S. 584]. Im weiteren Verlauf werden identifizierte relevante Submuster zu einer Kandidatsstruktur verschmolzen. Dabei ist anzumerken, dass nur die Submuster, die in Kontext des betrachtenden Entwurfsmusters assoziiert sind, im Vorgang der Verschmelzung involviert sind. Als finaler Schritt in dieser Methodik werden durch Analyse des Verhaltensmusters der Kandidatsstrukturen falsch positiv identifizierte Instanzen herausgefiltert [Yu13, S. 584]. Im Kontext dient Quellcode aus vier Open Source Software-Systemen als Eingabe für dieses Verfahren [Yu13, S. 585]. Zu Evaluierung der Resultate der Methodik wird die Metrik der *Precision* verwendet. Dabei wird das Verhältnis zwischen der Anzahl der positiv falschen identifizierten Instanzen zu der Summe der Anzahl der positiv falsch und falsch positiv klassifizierten Instanzen gebildet [Yu13, S. 585]. Je höher diese ist, desto besser ist das Ergebnis einzustufen. Nach eigenständiger Evaluierung von Dongjin et al. wird für alle Software-Systeme für alle betrachteten strukturellen Entwurfsmuster eine hohe Rate der *Precision* ermittelt [Yu13, S. 586].

2.4.2 Maschine Learning Ansätze

Mit der ersten öffentlich zugänglichen Version des Open Source Maschine Learning Frameworks TensorFlow am 9. November 2015, wurde der Zugang für praktisch angewandtes Maschine Learning für Software-Entwickler erleichtert. Durch TensorFlow wird eine Abstraktionsschicht für das Entwerfen, Trainieren und den Betrieb von Machine Learning Modellen, vor allem neuronalen Netzwerken, vereinfacht und somit die Anwendung von Maschine Learning in Produkten und in der Forschung populärer. Dieses ist auch der Fall für die Erkennung von Design Patterns in Quellcode. Wie aus Abbildung 2.6 zu entnehmen, stellen nach der Untersuchung von Yarahmadi et al. Verfahren, die Maschine Learning einsetzen, einen signifikanten Teil dar. Im Kontext dieser Aussage werden in diesem Abschnitt ausgewählte Verfahren erläutert, die Maschine Learning als Teil des Erkennungsprozesses involvieren. Der Fokus in dieser Sektion liegt neben den angewandten Modellen für die Klassifikation das Format der Eingabe, in der die Quelldateien für die Klassifikation transformiert werden.

Eine Möglichkeit, um Entitäten aus dem Quellcode wie Klassen oder Schnittstellen und deren Assoziation darzustellen, ist die Darstellung als Klassendiagramme aus der UML-Domäne. In ihrer Methodik nutzen Wang et al. UML Klassendiagramme als Grundlage und modifizieren diese mit der Inkorporation von Farb- und Symbolcodierungen. Dieses Format benennen Wang et al. als *Colored UML* [Wan22, S. 6].

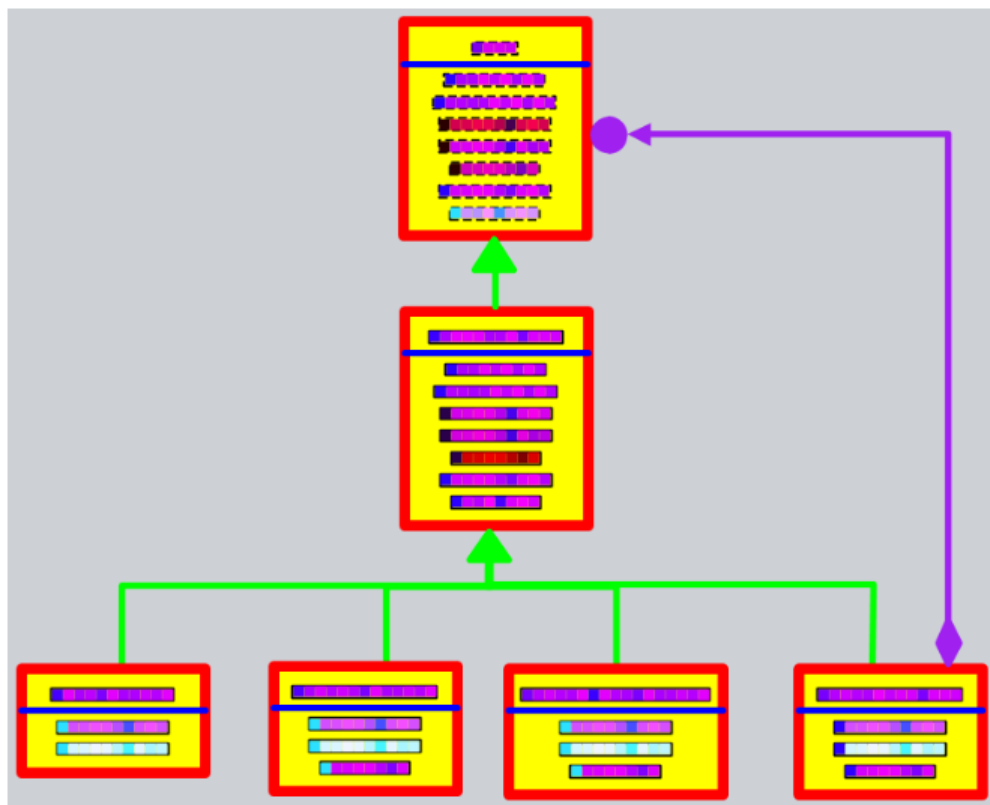


Abbildung 2.8 *Colored UML* für eine Mikroarchitektur

Die Abbildung 2.8 zeigt von Wang et al. erstelltes Exemplar für *Colored UML* [Wan22, S. 11]. Wie aus der Abbildung 2.8 zu entnehmen ist, werden Bezeichner, Modifizierer und Kanten für die Assoziation farblich und/oder symbolisch encodiert. Hierbei ist zu erwähnen,

dass die Einkodierung von Klassenattributen im Kontext dieser Arbeit von Wang et al, nicht berücksichtigt wird. Die Rechtecke, die in Sektion aufgeteilt, beschreiben die Klassen im Klassendiagramm. Die obere Sektion beinhaltet den Klassenbezeichner und dessen Modifizierer, während die untere Sektion die Methodenname und deren Modifizierer beinhaltet. Jeder Bezeichner werden als Sequenzen von Charakteren aufgefasst, in diesem für jeden Charakter algorithmisch eine Farbe aus dem Rot-Grün-Blau Farbraum ermittelt wird [Wan22, S. 9, S. 10]. Jedem Charakter wird algorithmisch eine Farbe aus dem Rot-Grün-Blau Farbraum zugewiesen [Wan22, S. 6]. Modifizierer für Methoden werden als Teil der Charaktersequenz betrachtet und mitenkodiert, während die Modifizierer für Klassen im Rand des Rechtecks enkodiert, welches die Farben für den Klassenbezeichner umschließt [Wan22, S. 6]. Die Kanten des Klassendiagramms, welche die Assoziation zwischen Entitäten darstellt, werden in *Colored UML* farblich annotiert und mit symbolisch an den Enden der Kanten je nach Art der Assoziation enkodiert [Wan22, S. 6].

Zunächst werden im ersten Schritt des Prozesses aus Quelldateien durch Software-Werkzeuge UML Klassendiagramme erstellt, welche im nächsten Schritt in *Colored UML* umgewandelt. Auf den enkodierten Quelldaten wird Bildklassifizierer VGGNet angewendet, welches Features aus der Eingabe als numerischen Vektor mit einer Länge von 1000 extrahiert. Im finalen Schritt dient dieser Feature-Vektor als Eingabe für eine Single Vector Maschine, welches die Klassenzifikation übernimmt, zu welchen Entwurfsmuster die Eingabe zugeordnet werden kann [Wan22, S. 13]. Das hier präsentierte Modell für zwölf Design Patterns trainiert [Wan22, S. 15] Für das Training des Modells und Evaluierung der Resultate des Verfahrens wurden drei Open Source Software-Systeme verwendet. Zusätzlich wird das Verfahren auf mit drei nicht Machine Learning Verfahren auf *Precision* und *Recall* verglichen [Wan22, S. 20]. Nach eigener Evaluierung von Wang et. al weist das ihrerseits entwickelte Verfahren verglichen zu den anderen Methoden ähnliche oder bessere Klassifizierungsleistung auf [Wan22, S. 22].

Ein weitere Möglichkeit, um Entwurfsmuster im Quellcode zu ermittelt, ist die Extraktion von Metriken und Maßen aus dem Quellcode, welche typische Instanzen für das betrachtete Design Pattern beschreiben. In ihrer Arbeit fokussieren sich Uchiyama auf diesen Punkt und definieren einen Katalog aus Metriken und Maßen, die als Eingabe für einen Klassifizierer verwendet wird. In dieser Arbeit definieren Uchiyama et al. Design Patterns als eine Summe von Strukturen von Klassen, Schnittstellen und Objekten, die im Kontext des Entwurfsmuster einer Rolle zugeordnet werden, und ihren Relationen zueinander. [Uch14, S. 3]. Im Kontext dieser Arbeit limitieren sich Uchiyama et al. auf die Erkennung fünf Entwurfsmuster mit insgesamt 12 Rollen im Quellcode. [Uch14, S. 4]. Für dieses Rollen werden Metriken und Maße ermittelt, die charakteristisch die Rollen beschreiben. Um die Elemente des Katalogs zu bestimmen, wird in dieser Arbeit die Goal-Question-Metric Methode angewendet [Uch14, S. 4]. Hierbei werden gezielt Fragen, mit dem Ziel, die jeweilige Rolle zu bestimmen. Für die Beantwortung dieser Fragen werden Metriken bzw. Maße definiert, und als Teil des Katalogs aufgenommen. Um die zwölf Rollen zu bestimmen, definieren Uchiyama et. al folgende Metriken [Uch14, S. 7]:

Tabelle 2.1 Metriken und Maße für Rollen nach Uchiyama et al.

Abkürzung	Beschreibung
NOF	Anzahl der Felder
NSF	Anzahl der statischen Felder
NOM	Anzahl der Methoden
NSM	Anzahl der statischen Methoden
NOI	Anzahl der implementierten Schnittstellen
NOAM	Anzahl abstrakter Methoden
NORM	Anzahl der überschriebenen Methoden
NOPC	Anzahl der privaten Konstruktoren in der Klasse
NOTC	Anzahl der Konstruktoren mit Objektparametern
NOOF	Anzahl der an Feldern mit Objekttypen
NCOF	Anzahl der anderer, die die Klasse/Schnittstelle als Feld referenzieren
NMGI	Anzahl der Methoden, die Instanzen generieren

Im ersten Schritt ihrer Methode extrahieren Uchiyama et al. die aus der Tabelle 2.1 definierten Metriken für jede Entität aus den Quelldateien. Dieser Metriken werden als Vektor aufgefasst und diesen als Eingabe für eine Klassifizierer verwendet, welche die Rollen-zuweisung für den Eingabevektor bestimmt [Uch14, S. 5]. Bei dem in dieser Methodik angewandten Klassifizierer handelt es sich um ein neuronales Netzwerk [Uch14, S. 4]. Die Ausgabe des Klassifizierers sind Werte, mit welcher Sicherheit der Eingabevektor zu der jeweiligen Rolle zugeordnet werden kann [Uch14, S. 5]. Anhand dieser Ausgabe werden die Rollen mit dem höchsten Konfidenzwert pro Eingabevektor als Eingabe für die Bestimmung des Entwurfsmusters genommen. Unter der Berücksichtigung der Relationen der Rollen in dem potenziellen Entwurfsmuster wird ein Wert ermittelt, welcher angibt, mit welcher Wahrscheinlichkeit die Rollen zu dem jeweiligen Design Pattern passen. [Uch14, S. 6]. Je höher dieser Wert, desto höher die Konfidenz, dass es sich hierbei um das Entwurfsmuster handelt. Für das Training ihres Klassifizierers nutzen Uchiyama et al. 60 Instanzen aus klein-skalierten Software-Systemen und 158 aus drei größeren Open Source Software-Systemen. [Uch14, S. 7] Für die Evaluierung der Methode bedienen sich Uchiyama et. al den Metriken der *Precision* und *Recall* und ermitteln diese für Design Pattern. Dabei werden für

Instanzen aus klein-skalierten Software-System besserer *Precision*- und *Recall*-Werte verglichen zu den Instanzen aus den Open Source Software-Systemen [Uch14, S. 8]

Durch Charakterisierung der Klassen, Schnittstellen und Objekte durch Metriken und Maße sind zwar Rückschlüsse auf Struktur und Verhalten möglich, jedoch wird der lexigraphische und syntaktische Aspekt nicht berücksichtigt. Um den gezeigten zu wirken, präsentieren Nazar et. al in ihrer Arbeit ihre Methode namens DPD_F , die diese Aspekte des Quellcodes mitberücksichtigt [Naz20, S. 1]. Initial wird aus dem Quellcode durch statische Codeanalyse folgende Metriken extrahiert [Naz20, S. 5]:

Tabelle 2.2 Extrahierte Features für DPD_F

Featurebezeichner	Beschreibung
ClassName	Name der Java Klasse
ClassModifiers	Public, Protected und Private-Schlüsselwörter
ClassImplements	Binäres Features (0/1), falls eine Schnittstelle implementiert wird
ClassExtends	Binäres Features (0/1), ob Klasse von einer anderen vererbt
MethodName	Methodname in der Klasse
MethodReturnType	Typ des Rückgabewerts einer Methode
MethodBodyLineType	Art des Ausdrucks (z.B Variablenzuweisung, Boolean-Ausdruck)
MethodNumVariables	Anzahl der Variablen/Attribute in der Klasse
MethodNumMethods	Anzahl der Methodenaufrufe in der Klasse
MethodsNumLine	Anzahl der Zeilen in Methode
MethodIncomingMethod	Anzahl an Methoden, die in eine Methode aufrufen
MethodIncomingName	Name der Methoden, die von der Methode aufgerufen werden
MethodOutgoingMethod	Anzahl ausgehender Methoden
MethodOutgoingName	Name der ausgehenden Methoden

Die ersten vier in der Tabelle 2.2 aufgelisteten Features werden pro Klasse, während die restlichen elf pro Methode in der Klasse ermittelt werden. Jeder dieser Features wird als eine Auflistung von Schlüssel-Werte-Paaren in natürlicher Sprache abgelegt. Da jede Auflistung der Evaluierung eines Methodenblocks entspricht, werden Features von der Klasse, in der die Methode implementiert ist, in diesen wiederholt. Jede Auflistung wird als Zeile in natürlicher Sprache in einer Textdatei zusammengefasst. Das dabei entstehende Format wird von Nazar et al. als Syntactic and Lexical Representation (SSLR) benannt [Naz20, S. 1]. Um SSLR in eine numerische Form zu bringen, wird SSLR als Eingabe für Word2Vec verwendet, welches für jeden Token mit Einbezug der umgebenden Tokens als Kontext der SSLR einen numerischen Vektor mit einer Länge von 100 determiniert, dass das jeweilige Token repräsentiert [Naz20, S. 6]. Die resultierenden Vektoren dienen als Eingabe für einen Random Forest Classifier, welches das meist passende Entwurfsmuster bestimmt [Naz20, S. 7]. Für das Training der Modelle und Evaluierung von DPD_F wird ein eigener Korpus angelegt, bestehend aus Quelldateien aus *Github Java Corpus*. Durch Crowd-Sourcing wurden zwölf Entwurfsmuster mit je 100 Instanzen identifiziert [Naz20, S. 4]. Für die Evaluierung wird für jedes betrachtete Design Pattern im Korpus der *F1*-, *Precision*- und *Recall*-Wert ermittelt. Nach eigener Evaluierung von Nazar et al. mit dem selbsterstellten Korpus, erreicht ihre Methode im Durchschnitt einen *Precision*-Wert von über 80% und einem *Recall*-Wert von 79% [Naz20, S. 8].

2.4.3 Query Ansätze

Ein weitere Möglichkeit, um Design Patterns in Quellcode zu erfragen, ist diese durch Anfragen an das Software-System zu identifizieren. Die Antwort auf die gestellte Anfrage ist dabei solch eine, die die in der Anfrage formulierten Bedingungen bestmöglichst befriedigt. Solch einen Ansatz verfolgen Mäder et al. in ihrer Methodik. Dabei liegt der Fokus in ihrer Methode durch Annotationen im Quellcode, die Aufschlüsse auf mögliche Entwurfsmuster geben. In ihre Arbeit definieren Mäder et al. Annotation als Teil des Quellcodes, die keine direkte Einwirkung auf Programsemantik haben [Mä10, S. 521] und bieten zusätzliche Informationen für statische Analyse durch Software-Werkzeuge und Verständnis des Software-Systems. Initial wird ein Katalog von Annotation definiert, welche manuell von Software-Entwickler in Quellcode des Software-Software aufgenommen wird [Mä10, S. 521].

- (1) `@abstract {notification [|interface, access_to_subsystem|state_management|list_traversal|object_identity|handling},`
- (2) `@compose {object} from {different_objects|related_objects},`
- (3) `@decouple {receiver} from {sender},`
- (4) `@object {instances} {share_by_introducing} {state_handlers|intrinsic_state|central_instance},`
- (5) `@decouple {implementation} by {dynamic variation lists},`
- (6) `@decouple {sender} from {receiver},`
- (7) `@provide {handlers} for {requests|expressions},`
- (8) `@ instantiation {eager|lazy|replaceable}.`

Abbildung 2.9 Ausschnitt an Annotation von Guhlam et al.

Abbildung 2.9 zeigt einen Ausschnitt des Annotationskatalog, welche im Kontext der Arbeit definiert wird. Dabei sind diese Annotation so bestimmt, dass diese Informationen über die Struktur und Verhalten des Software-Artifakts widerspiegeln. Der annotierte Quellcode wird im weiteren Verlauf durch das Software-Werkzeug Enterprise Architect prozessiert. Dieses Werkzeug generiert für das Quellcode-Modell und inkludiert mit den Beziehungen zwischen Software-Artefakten [Mä10, S. 521]. Information über Aggregation, Delegation und Freundschaften sind nicht Teil dieses Modells [Mä10, S. 521]. Um die Existenz von Entwurfsmustern zu erkennen, wird ein Katalog an Regeln definiert, die verschiedene Aspekte von Design Patterns durch Mitberücksichtigung der Annotationen widerspiegelt [Mä10, S. 523]. Diese Regeln werden in SQL-Anfragen und reguläre Ausdrücke transformiert und in Sets von Regeln aufgeteilt, die die Existenz des zugewiesenen Design Patterns überprüfen [Mä10, S. 523]. Zu der Evaluierung ihrer Methode vergleichen Mäder et al. ihre Methode mit zwei anderen Verfahren. Dabei wird verglichen, welche Entwurfsmuster diese erkennen [Mä10, S. 525]. Konkrete Metriken sind zu dem Zeitpunkt des Verfassens ihrer Arbeit nicht verfügbar [Mä10, S. 525].

Anstatt die Modellierung an ein professionales Software-Werkzeug zu delegieren, entwerfen Stencel et al. ein eigenes Metamodell, wonach Entitäten aus dem Quellcode und ihre Relationen zueinander modelliert werden [Ste08, S. 27]. Dabei besteht das Metamodell aus Kernelementen wie types, attributes operations und instances und strukturellen und verhaltensbezogenen Relation zwischen diesen Kernelementen. Sowohl die Kernelemente und als auch die Relationen in diesem Metamodell sind möglichst ähnlich der aus der objekt-orientierten Programmierung bekannten Verständnis definiert [Ste08, S. 27]. Jedes des im Kontext dieser Arbeit betrachteten Design Patterns wird von Stencel et al. vorgeschlagenen Metamodell enkodiert [Ste08, S.28 - 29]. Diese dienen als Referenz für die Erkennung des jeweiligen Entwurfsmusters im Quellcode und werden Grundlage für SQL-Anfragen genutzt [Ste08, S. 29]. In der Implementierung wird durch Strukturanalyse, Analyse des Datenflusses und der Methodenaufrufe werden die Artefakte in das Metamodell transformiert und in einer relationalen Datenbank abgelegt. Durch die SQL-Abfragen wird in der Datenbank nach Instanzen in der Datenbank abgefragt, die die von der Referenzdefinition bereitgestellten Bedingungen erfüllen. Zu Evaluierung verglichen Stencel et al. ihre Methode mit zwei anderen und überprüfen anhand von Software-Systemen, welche Entwurfsmuster erkannt werden [Ste08, S. 30]. Als Resultat erkennt das hier erläuterte Modell mehr Instanzen an als die anderen Methoden [Ste08, S. 29].

2.4.4 Diverse Ansätze

In ihrer Arbeit präsentieren Kramer et al. eine der frühesten Verfahren, um Entwurfsmuster in Quellcode zu erkennen. Mit der Absicht, die Wartbarkeit von Software-Systemen zu verbessern, nutzen die Kramer et al. die Programmiersprache PROLOG, um die zu erkennenden Design Pattern als einzelne Regeln zu repräsentieren [Kra01, S. 2]. Diese dienen als Referenz im weiteren Verlauf der Methode. Initial wird der zu verarbeitende Quellcode durch ein Software-Werkzeug in ein Object-Modelling Technique transformiert in eine Diagramm transformiert und im nächsten Schritt erstellte Diagramm in Prolog repräsentiert. Die als Referenz genutzten Definition der Entwurfsmuster dienen als Anfrage, die auf den transformierten Quellcode nach Instanzen des jeweiligen Design Patterns suchen [Kra01, S. 2]. Zu Evaluierung werden von Kramer et al. die Definitionen für die Design Patterns Adapter, Bridge, Decorater und Proxy angelegt und die Methode an vier C++ Bibliotheken angewendet. Nach eigener Evaluierung von Kramer et al. erzielt die Methode einen *Precision*-Wert zwischen 14% und 50% [Kra01, S. 7].

Anstatt ein Verfahren von Grund aus neu entwerfen, bedienen sich Binun et al. in ihrer Methode auf bereits etablierte Software-Werkzeuge und Methoden. Sie erläutern, dass unterschiedliche Verfahren die einzelnen betrachteten Entwurfsmuster unterschiedlich präzise erkennen [Bin09, S.15 - 17]. In ihrer Methode wird die Ausgabe von 5 Verfahren zu einem finalen Wert zusammengefasst, der angibt, mit welcher Wahrscheinlichkeit es sich um das determinierte Design Pattern handelt. Falls die einzelnen Methoden die Eingabe unterschiedlich klassifizieren, wird durch die Identifikation einzelner Rollen durch die Methoden das Entwurfsmuster zu rekonstruieren und dadurch die Eingabe einem Entwurfsmuster zuzuordnen [Bin09, S. 15 - 16]. Zur Evaluierung dienen zwölf Software-Systeme als Eingabe für die Methode, in diesen Instanzen von fünf Entwurfsmustern gesucht wird [Bin09, S. 9, S. 23]. Nach eigener Evaluierung von Binun et al. werden für drei von fünf Entwurfsmustern mit einem *Precision*-Wert von über 70% erkannt, während für die restlichen zwei mehrheitlich von den Methoden nicht erkannt werden [Bin09, S. 22].

3 Methodologie

3.1 Verfügbare Datensätze

3.2 Extrahierte Features

3.3 Betrachtete Multi-Klassifizierer für Rollen

3.4 Training und Hyperparameter-Tuning des Klassifizierers

3.5 Evaluation des trainierten Models

3.6 Ermittlung übereinstimmendes Design Patterns

3.7 Diskussion der Methodologie

4 Implementierung

4.1 Präprozessierung des Datensatzes

4.2 Extraktion von Features aus Quellcode

4.3 Analyse des Datensatzes

4.4 Training und Hyperparameter-Tuning

4.5 Bestimmung des Design Patterns

4.6 Evaluation der Ergebnisse

5 Schluss

5.1 Fazit

5.2 Zukünftige Aussichten

A Erstes Kapitel des Anhangs

Wenn Sie keinen Anhang benötigen, dann bitte einfach rausnehmen.

Literaturverzeichnis

- [Bin09] A. Binun und G. Kniesel-Wünsche. Witnessing Patterns: A Data Fusion Approach to Design Pattern Detection. 05 2009.
- [Gam94] E. Gamma, R. Helm, R. Johnson und J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 Aufl., 1994.
- [Kra01] C. Kramer, C. Gmbh, L. Prechelt und F. Informatik. Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software. 04 2001.
- [Leh96] M. M. Lehman. Laws of software evolution revisited. In C. Montangero, Hg., *Software Process Technology*, S. 108–124. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996.
- [Mä10] G. R. I. P. P. Mäder. Design pattern recovery based on annotations. *Advances in Engineering Software (Thomson Reuters) 2010-apr vol. 41 iss. 4, 41*, apr 2010.
- [Naz20] N. Nazar und A. Aleti. Feature-Based Software Design Pattern Detection. *CoRR*, abs/2012.01708, 2020.
- [Pra15] P. Pradhan, A. K. Dwivedi und S. K. Rath. Detection of design pattern using Graph Isomorphism and Normalized Cross Correlation. In *2015 Eighth International Conference on Contemporary Computing (IC3)*, S. 208–213. 2015.
- [Siu98] M.-K. Siu. Introduction to graph theory (4th edition), by Robin J. Wilson. Pp. 171. £14.99. 1996. ISBN : 0-582-24993-7 (Longman). *The Mathematical Gazette*, 82:343 – 344, 1998.
- [Ste08] K. Stencel und P. Wegrzynowicz. Detection of Diverse Design Pattern Variants. S. 25–32. 01 2008.
- [Suh10] I. Suh, Steve D.; Neamtii. [IEEE 2010 21st Australian Software Engineering Conference - Auckland, New Zealand (2010.04.6-2010.04.9)] 2010 21st Australian Software Engineering Conference - Studying Software Evolution for Taming Software Complexity. 2010.
- [Uch14] S. Uchiyama, A. Kubo, H. Washizaki und Y. Fukazawa. Detecting Design Patterns in Object-Oriented Program Source Code by Using Metrics and Machine Learning. *Journal of Software Engineering and Applications*, 7, 01 2014.
- [Wan22] L. Wang, T. Song, H.-N. Song und S. Zhang. Research on Design Pattern Detection Method Based on UML Model with Extended Image Information and Deep Learning. *Applied Sciences*, 12(17), 2022.
- [Yar20] H. Yarahmadi und S. M. H. Hasheminejad. Design pattern detection approaches: A systematic review of the literature. *Artificial Intelligence Review*, 53:5789–5846, 2020.

- [Yu13] D. Yu, Y. Zhang, J. Ge und W. Wu. From Sub-patterns to Patterns: An Approach to the Detection of Structural Design Pattern Instances by Subgraph Mining and Merging. In *2013 IEEE 37th Annual Computer Software and Applications Conference*, S. 579–588. 2013.
- [Zhi15] J. Zhi, V. Garousi-Yusifoğlu, B. Sun, G. Garousi, S. Shahnewaz und G. Ruhe. Cost, benefits and quality of software development documentation: A systematic mapping. *Journal of Systems and Software*, 99:175–198, 2015.