TECHNISCHE UNIVERSITÄT
CHEMNITZ

Evaluation von Programmierstrategien: Entwicklung eines Selbsteinschätzungs-Fragebogens für Softwareentwickler

Masterarbeit

zur Erlangung
des akademischen Grades
M.Sc.

Fakultät für Informatik
Professur Softwaretechnik

Eingereicht von: Florian Grabs
Matrikel Nr.: -
Einreichungsdatum: 18.03.2026

Betreuerin: Univ.-Prof. Dr. -Ing. Janet Siegmund,
Belinda Schantong

Abstract

Die vorliegende Arbeit untersucht die Schnittstelle zwischen domänenspezifischen Programmierstrategien und allgemeinen psychologischen Problemlösungsmodellen. Ziel der Untersuchung war es, theoretisch fundierte Praktiken der Softwareentwicklung (wie Dekomposition, TDD oder Scent-Following) empirisch auf kognitive Mechanismen (wie schemagesteuertes Lösen oder Mittel-Zweck-Analyse) abzubilden. Im Rahmen einer quantitativen Studie ($N = 31$) wurde mittels eines neu entwickelten Fragebogens die Nutzungshäufigkeit dieser Strategien erhoben und deren Zusammenhang mit der Programmiererfahrung analysiert.

Die Ergebnisse zeigen, dass die Problemzerlegung und der Einsatz von LLM-Tools die am häufigsten angewandten Strategien im modernen Entwicklungsprozess darstellen. Die Korrelationsanalyse nach Spearman verdeutlicht, dass steigende Expertise mit einer Abkehr von ungerichteten *Trial-and-Error*-Verfahren hin zu systematischen, manuellen Debugging-Methoden und der Nutzung komplexer Wissensschemata einhergeht. Während sich das Mapping für metakognitive und operative Strategien durch hohe interne Konsistenzen (Cronbachs Alpha bis zu 0,81) bestätigen ließ, zeigten abstrakte Strategien wie die Analogiebildung einen Bedarf an präziserer Operationalisierung. Die Arbeit liefert damit wichtige Erkenntnisse für die Informatikdidaktik und die Gestaltung von Assistenzsystemen zur Unterstützung kognitiver Prozesse beim Programmieren.

Inhaltsverzeichnis

Inhaltsverzeichnis 3

Abbildungsverzeichnis 5

Tabellenverzeichnis 6

1. Einleitung 7
1.1. Motivation und Problemstellung 7
1.2. Zielsetzung 7
1.3. Aufbau der Arbeit 7

2. Theoretischer Rahmen 8
2.1. Definition Strategie 8
2.2. Problemlösungsstrategien in der Psychologie 8
2.3. Problemlösungsstrategien in der Informatik 12
2.3.1. Strategien im Designprozess 12
2.3.2. Strategien im Programmierprozess 17
2.3.3. Strategien in der Programmanalyse 21
2.3.4. Strategien im Debugging 25

3. Methodik 29
3.1. Konstruktion des Fragebogens 29
3.1.1. Designprozess: Herleitung der Items aus den Strategien 29
3.1.2. Programmierprozess: Herleitung der Items aus den Strategien 33
3.1.3. Programmanalyse: Herleitung der Items aus den Strategien 36
3.1.4. Debugging: Herleitung der Items aus den Strategien 39
3.2. Mapping Programmier- auf psychologische Problemlösungsstrategien 42
3.3. Detaillierte Erläuterung der Item-Mappings 44
3.3.1. Hinweis zur Mehrfachzuordnung 49
3.4. Forschungsfragen 49
3.5. Studienteilnehmer 50
3.6. Durchführung 51

4. Ergebnisse 57
4.1. Beschreibung der Teilnehmer 57
4.2. Datenauswertung 61
4.2.1. Deskriptive Statistik der Fragen 61

4.2.2. Deskriptive Statistik der Programmierstrategien 63
4.2.3. Deskriptive Statistik der psychologischen Strategien ohne Feedback 65
4.2.4. Deskriptive Statistik der psychologischen Strategien mit Feedback 67
4.2.5. Auswertung einzelner Teilnehmer 69
4.2.6. Pearson Korrelationsanalyse psychologische Strategien 72
4.2.7. Spearman Analyse Programmiererfahrung und Programmierstrategien 74
4.2.8. Spearman Analyse Programmiererfahrung und psychologische Strategien 76
4.2.9. Cronbachs Alpha 78

5. Diskussion 79
5.1. Beantwortung der Forschungsfragen 79
5.1.1. Forschungsfrage 1 79
5.1.2. Forschungsfrage 2 80
5.1.3. Forschungsfrage 3 81
5.1.4. Forschungsfrage 4 82
5.2. Qualitative Auswertung des Teilnehmerfeedbacks 84
5.2.1. Validierung der Selbsteinschätzung 84
5.2.2. Detaillierte Analyse spezifischer Strategien 84
5.2.3. Kritische Würdigung des methodischen Vorgehens 85
5.2.4. Zusammenfassung der qualitativen Ergebnisse 85

6. Einschränkungen der Validität 86
6.1. Konstruktvalidität 86
6.2. Interne Validität 86
6.3. Externe Validität 87

7. Fazit und Ausblick 88
7.1. Kritische Würdigung und Beitrag 88
7.2. Ausblick 88

Literaturverzeichnis 89

A. Tabellen 93

B. Github Link 95

Abbildungsverzeichnis

3.1. Fragebogen Seite 1 (Einleitung) 51
3.2. Fragebogen Seite 2 (Programmiererfahrung Frage 1) 52
3.3. Fragebogen Seite 4 (Strategien im Programmierprozess) 54
3.4. Fragebogen Seite 7 (Feedback psychologische Strategie: Schemagesteuertes Lösen & Domänenspezifisches Wissen) 55
3.5. Fragebogen abschließendes Feedback 56

4.1. Auswertung demografische Frage 1 57
4.2. Auswertung demografische Frage 2 58
4.3. Auswertung demografische Frage 3 59
4.4. Auswertung demografische Frage 4 60
4.5. Top 10 / Flop 10 Mittelwerte der beantworteten Fragen 61
4.6. Nutzungshäufigkeit Programmierstrategien 63
4.7. Verteilung psychologischer Strategien (ohne Feedback) 65
4.8. Verteilung psychologischer Strategien (mit Feedback) 67
4.9. Netzdiagramm Teilnehmer 12 70
4.10. Netzdiagramm Teilnehmer 21 71
4.11. Korrelationsmatrix psychologische Strategien (ohne Feedback) 72
4.12. Korrelationsmatrix psychologische Strategien (mit Feedback) 73
4.13. Heatmap Spearman Analyse: Nutzung spezifischer Programmierstrategien je nach Programmiererfahrung 74
4.14. Heatmap Spearman Analyse: Nutzung psychologische Strategien je nach Programmiererfahrung 76

Tabellenverzeichnis

3.1. Operationalisierung der Programmierstrategien (Designprozess) . . . 29
3.2. Operationalisierung der Programmierstrategien (Programmierprozess) 33
3.3. Operationalisierung der Programmierstrategien (Programmanalyse) . 36
3.4. Operationalisierung der Programmierstrategien (Debugging) 39
3.5. Mapping der Items auf psychologische Problemlösungsstrategien . . . 42
3.6. Übersicht der standardisierten Fragen zur Programmiererfahrung . . . 53

4.1. Übersicht der Gruppenergebnisse 69
4.2. Interne Konsistenz (Cronbachs Alpha) der psychologischen Strategiekategorien ($N = 31$) 78

A.1. Deskriptive Statistik der Fragebogen-Items 93

1. Einleitung

1.1. Motivation und Problemstellung

Softwareentwicklung ist im Kern ein hochkomplexer Problemlösungsprozess. Programmierer navigieren täglich durch einen dichten Dschungel aus abstrakten Anforderungen, kryptischen Fehlermeldungen und sich ständig wandelnden technologischen Frameworks [22]. Während die Informatik eine Vielzahl an "Best Practices" und Entwurfsmustern hervorgebracht hat, bleibt die Frage oft unbeantwortet, welche kognitiven Mechanismen hinter diesen Praktiken stehen. Warum zerlegt ein Senior-Entwickler ein Problem instinktiv in Module, während ein Novize verzweifelt im *Trial-and-Error*-Modus verharrt [20]?

Die psychologische Problemlöseforschung bietet hierfür theoretische Erklärungsmodelle, doch klafft zwischen der allgemeinen Kognitionswissenschaft und der spezifischen Disziplin der Softwareentwicklung eine Lücke. Bestehende Ansätze konzentrieren sich oft nur auf Teilbereiche wie das Debugging oder die Syntaxerlernung, vernachlässigen aber das ganzheitliche strategische Vorgehen vom ersten Entwurf bis zur finalen Code-Analyse.

1.2. Zielsetzung

Die vorliegende Arbeit setzt an dieser Lücke an. Das primäre Ziel ist es, ein theoretisches Mapping zu entwickeln, das Programmierstrategien auf generische psychologische Problemlösungsmodelle überträgt und dieses empirisch zu validieren. Dabei soll untersucht werden, wie sich das Strategierepertoire in Abhängigkeit von der Erfahrung verändert.

1.3. Aufbau der Arbeit

Um diese Fragen zu beantworten, wird zunächst im *Theoretischen Rahmen* (Kapitel 2) die Definition von Strategien in der kognitiven Psychologie und der Informatik erläutert. In Kapitel 3 (*Methodik*) wird die Konstruktion eines spezialisierten Fragebogens beschrieben, der diese Strategien operationalisiert. Nach der Darstellung der *Ergebnisse* (Kapitel 4) folgt eine detaillierte *Diskussion* (Kapitel 5), die das theoretische Mapping kritisch hinterfragt und die Grenzen der Validität aufzeigt. Die Arbeit schließt mit einer Zusammenfassung und einem Ausblick in Kapitel 6.

2. Theoretischer Rahmen

Im folgenden Kapitel werden ausgewählte psychologische Problemlösungsstrategien und explizite Programmierstrategien erläutert. Zunächst folgt eine kurze Definition des Begriffs "Strategie".

2.1. Definition Strategie

Der Begriff der Strategie wird im wissenschaftlichen Diskurs der kognitiven Psychologie und Problemlöseforschung als ein strukturierter Plan oder eine systematische Sammlung von Regeln definiert, um spezifische Ziele zu erreichen und Barrieren kriteriengeleitet zu überwinden [11, 18]. Dabei fungiert eine Strategie primär als Technik, die im Problemlösungsprozess als orientierender Leitfaden dient, ohne dabei notwendigerweise eine Lösung zu garantieren [25]. In komplexen Entscheidungssituationen legen Strategien fest, wie aus einer Vielzahl vorhandener Optionen eine zielführende Auswahl getroffen werden kann [18]. Diese methodischen Vorgehensweisen können sowohl klar definierten Algorithmen als auch unscharfen Heuristiken folgen. Ein klassisches Beispiel für eine effiziente Suchstrategie ist das *Hill climbing* (Bergsteigen), bei dem sukzessive jene Optionen gewählt werden, welche die Distanz zum angestrebten Zielzustand maximal reduzieren [11]. Letztlich repräsentieren Strategien somit komplexe Wissensstrukturen über die Lösung spezifischer Problembereiche und dienen in angewandten Kontexten der langfristigen Steuerung, Weiterentwicklung und Problemlösung [34].

2.2. Problemlösungsstrategien in der Psychologie

Schemagesteuertes Lösen & Domänenspezifisches Wissen Das schemagesteuerte Problemlösen basiert auf der Aktivierung von Problem-Schemata, bei denen es sich um im Langzeitgedächtnis gespeicherte Wissenscluster handelt, die Informationen über Problemziele, Einschränkungen und Lösungsverfahren enthalten [15]. Sobald ein Problemlöser einen vertrauten Problemtyp erkennt, wird das entsprechende Schema aktiviert, was eine direkte Implementierung des Lösungsverfahrens mit nur geringem Suchaufwand ermöglicht [15][10]. Diese Strategie ist kennzeichnend für Experten, die in wissensreichen Domänen über tiefgreifende, lösungsorientierte Wissensrepräsentationen verfügen. Ein wesentlicher Bestandteil dieser Expertise ist die Mustererkennung (eng. Perceptual Concepts), die es erlaubt, relevante Konfigurationen, wie etwa vertraute Figurenkonstellationen im Schach oder spezifische Diagrammmerkmale in der Geometrie, sofort zu

identifizieren [16].

Darüber hinaus umfasst dieses Wissen domänenspezifisches, logisches Denken, das durch das Lösen situierter Probleme in Fachbereichen wie der Medizin, dem Jura oder der Psychologie entwickelt wird [15]. In der Praxis manifestiert sich dies beispielsweise in der Physik durch das "Vorwärtsarbeiten", bei dem Experten von den gegebenen Informationen ausgehend direkt die passenden Formeln wählen [15]. Weitere Anwendungen finden sich in der Geometrie durch spezifische Beweisstrategien für kongruente Winkel oder in der Informatik durch die rekursive Zerlegung komplexer Software-Designprobleme [16].

Problemzerlegung & Unterzielen Die Problemzerlegung (eng. Decomposition) und das Setzen von Unterzielen (eng. Subgoaling) sind zentrale Strategien, bei denen ein komplexes Problem in kleinere, handhabbare Unterprobleme unterteilt wird [15] [10]. Dieser Prozess reduziert den Suchraum exponentiell und ist besonders nützlich bei Designproblemen in der Architektur oder dem Software-Design. Experten wenden die Zerlegung oft rekursiv an, während Novizen dies häufig versäumen [15]. Ergänzend dazu wird beim Setzen von Unterzielen ein Zwischenzustand auf dem Lösungspfad als temporäres Ziel ausgewählt [10]. Diese Vorgehensweise ist eng mit der Planungsstrategie verknüpft, bei der das Problem strukturiert und eine geeignete Reihenfolge für den Abschluss der Unterprobleme festgelegt wird. Während die klassische Mittel-Zweck-Analyse (eng. Means-Ends Analysis) an ihre Grenzen stoßen kann, wenn eine vorübergehende Vergrößerung der Differenz zum Ziel nötig ist, kann das Setzen von Unterzielen diese Einschränkung aufheben. Eine bewährte Heuristik besteht darin, zunächst mit dem Rückwärtsarbeiten zu beginnen und anschließend vorwärts zu operieren oder das Hauptziel systematisch zu zerlegen [10].

Arbeiten Vorwärts (Vorwärtssuche) Die Strategie des Vorwärtsarbeitens (eng. Working Forward) ist ein Prozess, bei dem der Problemlöser von den gegebenen Informationen ausgeht und sich schrittweise zur unbekannten Zielgröße vorarbeitet. Dabei werden gezielt Operatoren oder Formeln angewendet, für welche die benötigten Variablen bereits bekannt sind [16]. In Domänen wie der Physik wählen Experten Gleichungen, die die vorhandenen Gegebenheiten enthalten, und berechnen nacheinander Unbekannte, bis die endgültige Lösung gefunden ist [16].
Diese Vorgehensweise wird primär bei Experten beobachtet, da sie durch Mustererkennung die anzuwendenden Formeln direkt identifizieren können [16]

Arbeiten Rückwärts (Rückwärtssuche) Das Rückwärtsarbeiten (auch Rückwärts-

suche oder eng. Backward-Chaining genannt) ist eine suchbasierte Strategie, die typischerweise vom Zielzustand ausgeht und Operationen findet, die diesen Zustand aus einem dem Anfang näher liegenden Zustand erzeugen können [10] [16]. Diese Methode wird häufig von Novizen angewendet, beispielsweise bei Physikproblemen, wo sie eine Gleichung wählen, die das Ziel enthält, und dann Unterziele setzen, um weitere Unbekannte zu finden [15]. Der Prozess beinhaltet eine Mittel-Zweck-Analyse, um Operatoren anwendbar zu machen und vom Schluss auf die Prämissen zu schließen [16]. Obwohl das Rückwärtsarbeiten Planung erfordert und als anspruchsvoller als die reine Vorwärtssuche gilt, kann die Anzahl der "Rückwärtszweige" im Suchraum kleiner sein als bei der Vorwärtssuche [10]. Eine hilfreiche Heuristik besteht zudem darin, mit dem Rückwärtsarbeiten zu beginnen und anschließend vorwärts zu arbeiten oder das Hauptziel in Unterziele zu zerlegen [10].

Generieren und Testen Die Problemlösungsstrategie "Generieren und Testen" (eng. Generate-and-Test) beinhaltet das Erstellen einer Reihe möglicher Lösungen, die anschließend einzeln auf ihre Korrektheit überprüft werden [10]. Diese Heuristik ist besonders dann nützlich, wenn die Menge der potenziellen Lösungen klein und die Überprüfung einfach ist oder wenn der spezifische Pfad zur Lösung keine primäre Rolle spielt [10]. In der Praxis findet dieses Verfahren häufig Anwendung in der wissenschaftlichen Forschung, bei medizinischen Diagnosen oder im Bereich des Troubleshootings [17]. Während Novizen diese Strategie oft wahllos einsetzen, zeichnen sich Experten dadurch aus, dass sie den Prozess mit einer deutlich genaueren Hypothese beginnen [10]. Im Kontext des strategischen Handelns nutzen Troubleshooter beispielsweise Symptome, um gezielt Hypothesen über verschiedene Fehlerzustände zu generieren und diese systematisch zu testen [17].

Problemlösung durch Analogie Bei der Problemlösung durch Analogie sucht der Problemlöser gezielt nach einem analogen Problem, dessen Lösung bereits bekannt ist [15]. Diese Strategie erweist sich als besonders nützlich für den Wissenserwerb in Bereichen wie der Textbearbeitung, Geometrie und Programmierung [15] [16]. Ein wesentlicher Unterschied in der Anwendung liegt darin, dass Novizen dazu neigen, lediglich oberflächliche Analogien zu verwenden, anstatt solche heranzuziehen, die auf den tatsächlichen Lösungsverfahren basieren [15].
Die Analogiebildung beinhaltet zudem das Analysieren von Relationen zwischen Item-Paaren, um durch den Vergleich dieser Beziehungen entsprechende Übereinstimmungen zu finden [16].

Metakognitive und Affektive Strategien Metakognitive und affektive Strategien beziehen sich auf die Selbstregulierung sowie auf die affektiven Aspekte einer Person während des Problemlösungsprozesses [17]. Die Metakognition umfasst dabei das

Bewusstsein über den eigenen Lernprozess, die Einschätzung der Schwierigkeit einer Aufgabe, die Überwachung des Verständnisses sowie die Bewertung des Lernfortschritts [17] [15]. Diese Strategien werden oft als allgemeine Methoden eingestuft, die typischerweise später im Lösungsprozess angewendet werden, um die Regulierung der Problemlösung zu unterstützen [17]. Bei der Bearbeitung von Mathematikproblemen zeigt sich beispielsweise, dass gute Problemlöser ihre Ziele klären, ihr Verständnis ständig überwachen und die gewählten Handlungen bewerten [17]. Ein wesentlicher Bestandteil sind zudem konative Elemente wie zielgerichtete Teilnahme, Anstrengung und das Beharren auf der Aufgabe [17]. Hierbei beeinflusst das Selbstvertrauen beziehungsweise die Selbstwirksamkeit maßgeblich, wie viel bewusster Aufwand und Beharrlichkeit investiert werden, um auch komplexe oder schlecht strukturierte Anforderungen erfolgreich zu bewältigen [17].

Planung/Planungsstrategie Die Planungsstrategie ist eine allgemeine Suchstrategie, die darauf basiert, ein komplexes Problem in Unterprobleme zu zerlegen und eine geeignete Reihenfolge für deren Abschluss zu finden [15]. In diesem Prozess wird der Problemraum vereinfacht, indem von unwichtigeren Merkmalen abstrahiert wird. Die in diesem vereinfachten Raum entwickelte Lösung dient anschließend als Leitfaden für die Suche im ursprünglichen, komplexen Problemraum [16]. Diese Strategie wird bevorzugt angewendet, wenn bestimmte Differenzen zwischen dem Anfangs- und Zielzustand als besonders wichtig erachtet werden [16]. Durch die Zerlegung in kleinere Untereinheiten (eng. Subgoaling) kann der Suchraum exponentiell reduziert werden, wobei dieser Planungsprozess als anspruchsvoller gilt als eine reine Vorwärtssuche [10]. Es ist jedoch zu beachten, dass die erfolgreiche Anwendung dieser Strategie durch das vorhandene domänenspezifische Wissen eingeschränkt sein kann [15]. Der wesentliche Unterschied zur Strategie "Problemzerlegung & Unterzielen" liegt in der Abstraktionsebene: Während die Planungsstrategie zunächst ein vereinfachtes Modell des Problems erstellt, um eine strategische Marschroute festzulegen, konzentriert sich die Problemzerlegung auf das operative Aufbrechen der Komplexität in konkrete, handhabbare Teilschritte.

2.3. Problemlösungsstrategien in der Informatik

2.3.1. Strategien im Designprozess

Agile und Lean Methoden Agile und Lean Methoden bilden umfassende Rahmenwerke zur Steuerung des gesamten Entwicklungsprozesses, die insbesondere darauf ausgelegt sind, auf volatile Geschäftsanforderungen und rasche technologische Änderungen zu reagieren. Zu den populärsten Ausprägungen zählen Extreme Programming (XP), welches stark praxisorientiert ist, und Scrum, das den Fokus auf Projektmanagement legt [30]. Ergänzend adaptiert Lean Software Development die Prinzipien des Toyota-Produktionssystems auf die Softwareentwicklung. Lean liefert dabei oft die theoretische Basis für agile Praktiken und fokussiert sich auf sieben Kernprinzipien: Eliminierung von Verschwendung, Integrierung von Qualität, Wissensaufbau, späte Entscheidungsfindung, schnelle Lieferung, Respekt vor Menschen und die Optimierung des Gesamtsystems [31].
Der operative Ablauf wird durch iterative Zyklen strukturiert. Zu Beginn jeder Iteration erfolgt ein "Iteration Planning", um Ziele festzulegen und Aufgaben zu verteilen. In Scrum wird dies als "Sprint Planning" bezeichnet, während Extreme Programming das "Planning Game" nutzt. Anforderungen werden hierbei häufig in Form von "User Stories" formuliert, die dazu dienen, Kunden näher an die Entwicklung zu binden und Kernanforderungen aufzudecken [30].
Zur täglichen Steuerung und Synchronisation nutzen Teams Daily Meetings / Stand-up Meetings. Diese dienen dazu, den Projektstatus kurzfristig zu klären, Aufgaben für den nächsten Tag zu planen und Hindernisse zu identifizieren. In der Praxis werden diese Meetings von Entwicklern teilweise auch genutzt, um Designprobleme unmittelbar zu lösen. Zur Visualisierung des Fortschritts und der gemeinsamen Ziele kommen Story- oder Task-Boards zum Einsatz. Diese basieren oft auf dem "Story Board" aus Extreme Programming , auf dem Aufgaben und deren Status für das gesamte Team sichtbar gemacht werden [30].
Am Ende der Zyklen stehen Iteration Reviews oder Sprint Reviews, in denen funktionierende Software demonstriert wird. Diese Reviews fördern die Kommunikation zwischen Entwicklern und Stakeholdern und helfen, Abhängigkeiten zwischen Features und Anforderungen zu klären. Auf technischer Ebene wird dieser Prozess durch "Continuous Integration" gestützt. Diese Praxis sorgt dafür, dass ständig neue Builds für Testzwecke verfügbar sind und erleichtert somit die Qualitätssicherung und den Informationsfluss über den Status des Endprodukts [30].
In den vorliegenden Quellen wird nicht explizit kategorisiert, ob es sich hierbei generell um eine Strategie nur für Experten oder Novizen handelt. Es wird jedoch angemerkt, dass erfahrene Entwickler, die an traditionelle planungsorientierte Prozesse gewöhnt sind, Widerstand gegen Praktiken wie offene Büroräume leisten können, während Teams, die neu in agilen Methoden sind, von Coaching und Unterstützung durch Experten profitieren [30].

Test Driven Development Test-Driven Development (TDD) ist weit mehr als eine reine Testtechnik. Es handelt sich um einen umfassenden Design-Ansatz, dessen primäres Ziel sauberer, funktionierender Code ist. Der Kernprozess wird durch das Mantra "Red/Green/Refactor" definiert. Zunächst schreibt der Entwickler einen kleinen, fehlschlagenden Test ("Red"), der oft initial nicht einmal kompiliert. Darauf folgt die Implementierung von gerade genug Code, notfalls auch durch vorübergehend unsaubere Lösungen ("Sünden"), um den Test schnellstmöglich bestehen zu lassen ("Green"). Erst im dritten Schritt wird der Code bereinigt und Duplizierung entfernt ("Refactor")[6].
Als Programmierstrategie verlangt Test Driven Development ein systematisches Vorgehen. Dies beginnt mit der Auflistung aller Benutzerszenarien basierend auf den Anforderungen, gefolgt von der Erstellung eines Tests für ein spezifisches Szenario und der Verifizierung, dass dieser fehlschlägt. Nach der Implementierung werden Design- und Performance-Probleme behoben, wobei stets sichergestellt wird, dass alle Tests weiterhin bestehen[22].
Bezüglich der Eignung für Experten oder Novizen zeichnen die Quellen ein differenziertes Bild. Einerseits wird Test Driven Development als Technik beschrieben, die der großen Mehrheit der Softwareentwickler helfen kann, ihr Potenzial besser auszuschöpfen und Ängste im Entwicklungsprozess zu bewältigen [6]. Andererseits zeigen empirische Untersuchungen, dass Test Driven Development in der Praxis eher Merkmale einer Expertenstrategie aufweist bzw. signifikante Übung erfordert. In Studien wählten Entwickler, die sich selbst steuerten, selten Test Driven Development, sondern bevorzugten intuitivere Zerlegungsstrategien [22]. Teilnehmer, die explizit zur Nutzung von Test Driven Development angeleitet wurden, empfanden ihre Arbeit zwar als organisierter und systematischer , jedoch machten nur diejenigen mit TDD-Vorerfahrung signifikante Fortschritte bei der funktionalen Implementierung. Teilnehmer ohne Erfahrung verbrachten hingegen unverhältnismäßig viel Zeit damit, den Prozess des Testschreibens und Planens überhaupt erst zu erlernen, was darauf hindeutet, dass die effektive Anwendung von Test Driven Development ein gewisses Maß an Expertise voraussetzt[22].

Design Patterns und Muster Sprachen Die Strategie der Design Patterns (Entwurfsmuster) und Muster-Sprachen basiert auf der systematischen Erfassung und Formalisierung bewährter Lösungen für wiederkehrende Entwurfsprobleme. Diese Muster fungieren als kodifiziertes Expertenwissen und werden klassischerweise, wie im wegweisenden Katalog von Gamma et al. (Gang of Four) [14] strukturiert, in Kategorien wie Erzeugungsmuster (eng. Creational), Strukturmuster (eng. Structural) und Verhaltensmuster (eng. Behavioral) unterteilt [14]. Ein Muster ist dabei mehr als eine bloße Vorlage, es definiert explizit den Kontext, die abzuwägenden Kräfte (eng. Forces) sowie die Konsequenzen der Anwendung, um eine optimale Balance für eine spezifische Situation zu finden [8]. Um dieses Wissen effektiv zu transferieren, werden die Muster häufig in Muster-Sprachen

organisiert. Diese bilden eine hierarchische Struktur, oft als gerichteter azyklischer Graph, die Designer von abstrakten Problemen auf hoher Ebene zu detaillierten Lösungen leitet [8]. In der Praxis dienen diese Sprachen als gemeinsame "Lingua Franca", die es interdisziplinären Teams ermöglicht, Design-Expertise verständlich auszutauschen und als "Corporate Memory" zu bewahren [8].
In Bezug auf den Experten-Novizen-Status wird diese Strategie in den Quellen differenziert betrachtet. Grundsätzlich stellen Muster die Erfahrung und Werte des Programmierers dar und sind somit Expertenwissen [8]. Ihre Anwendung dient jedoch oft explizit dazu, die Lücke zwischen Experten und Anfängern zu schließen. Borchers hebt hervor, dass Entwurfsmuster ein praktisches Mittel sind, um weniger erfahrene Entwickler in ein Fachgebiet einzuführen [8]. In seiner Untersuchung konnten selbst Studienanfänger durch die Nutzung einer Mustersprache schnell passende Lösungen identifizieren und ein Vokabular entwickeln, um Designoptionen fast wie Experten zu diskutieren [8].
Gleichzeitig wird die selbstständige und korrekte Anwendung komplexer Muster als Unterscheidungsmerkmal für Expertise gewertet. Kesselbacher und Bollin identifizieren in ihrer Analyse von Programmiersequenzen die Verwendung wiederkehrender, effizienter Muster, wie z. B. Loop Solutions mit variablen Blocktypen, als klare Strategie erfahrener Programmierer [19]. Im Gegensatz dazu agieren wirkliche Novizen oft ohne solche Strategien, nutzen nur begrenzte Blocktypen oder führen Programme wahllos aus, ohne erkennbare Muster zu bilden [19]. Die Nutzung von Mustern ist also einerseits ein Werkzeug für Novizen, um zu lernen, und andererseits ein Merkmal, an dem man Experten erkennt [19].

Dekomposition / Zerlegung Die Dekomposition, auch Zerlegung genannt, stellt eine zentrale Problemlösungsstrategie in der Softwareentwicklung dar, bei der ein komplexes Gesamtproblem in kleinere, handhabbare Unterprobleme aufgeteilt wird, die anschließend unabhängig voneinander implementiert werden können. Im Kontext von Entwurfsaufgaben äußert sich diese Strategie konkret darin, dass Entwickler funktionale Anforderungen analysieren, um diese in verständliche Teilaufgaben zu gliedern, was ihnen hilft, die Arbeit besser zu organisieren, das Problem tiefergehend zu durchdringen und einen klaren Einstiegspunkt für die Lösung zu finden [22].
In der Forschung zur Softwareentwicklungs-Expertise wird die Fähigkeit zur Abstraktion und Dekomposition explizit als eine Fertigkeit hervorgehoben, die Experten besitzen sollten. Ein Experte zeichnet sich diesem Verständnis nach dadurch aus, dass er fähig ist, ein riesiges Problem in kleine Stücke zu zerlegen, die einzeln gelöst werden können, um wieder ein Ganzes zu ergeben. Auch das Wissen über Softwarearchitektur, welches Konzepte der Modularisierung und Dekomposition beinhaltet, wird als spezifisches Merkmal von Expertenwissen identifiziert [5]. Dennoch ist diese Vorgehensweise nicht ausschließlich auf Experten beschränkt. In einer Studie zur Anwendung von Programmierstrategien zeigte sich,

dass die Dekompositionsstrategie auch von Entwicklern in einer selbstgeleiteten Gruppe intuitiv am häufigsten gewählt wurde, wobei sie von 64 % dieser Teilnehmer angewandt wurde [22].
Ergänzend wird in der Literatur darauf hingewiesen, dass Lehrende Problemlösungsstrategien wie "Divide and Conquer" (deu. Teile und Herrsche) aktiv vermitteln sollten, um Studierende bei der Entwicklung dieser Fähigkeiten zu unterstützen [36]. Trotz der Vorteile für das Problemverständnis birgt diese Strategie jedoch auch Risiken, da Fehler in der initialen Zerlegung zu späterem Mehraufwand führen können, wenn Lösungen aufgrund einer fehlerhaften Strukturierung neu entworfen werden müssen [22].

Typkonstruktion als Strategie Die Typkonstruktion fungiert in der funktionalen Programmierung nicht als strikt linearer Vorgang, sondern als iterativer, zyklischer Prozess, bei dem Programmierer kontinuierlich zwischen der Bearbeitung von Typdefinitionen und der Erstellung von Ausdrücken wechseln. In diesem Kontext bestimmt häufig die konkrete Anwendung die Form des Typs "usage drive the type". Programmierer entwerfen zunächst Beispielausdrücke oder nutzen Variablen, bevor deren Typen finalisiert sind, um Designentscheidungen ergonomisch abzusichern. Ergänzend nutzen diese Programmierer die typgesteuerte Top-Down-Zerlegung "following the types", bei der Typannotationen als erster Implementierungsschritt dienen, um komplexe Probleme zu strukturieren und die Aufmerksamkeit auf isolierte Funktionsbereiche zu lenken. Diese Praktiken wurden spezifisch bei praktizierenden Programmierern mit meist mehrjähriger Erfahrung beobachtet. Während hierarchische Strategien in der Literatur oft als Experten Strategie gelten, hebt die Studie hervor, dass diese Experten flexibel zwischen hierarchischen und opportunistischen Methoden wechseln. Die ist ein Verhalten, das sie teilweise von in verwandten Arbeiten untersuchten Novizen unterscheidet [24].

Skizzierung (Sketching) Die Skizzierung (eng. Sketching) ist eine Programmierstrategie, bei der Programmierer Teile ihres Codes bewusst unvollständig lassen, um sich zunächst auf die übergeordnete Absicht zu konzentrieren und Low-Level-Details auszublenden. Dabei entstehen sogenannte "Programmlöcher". Diese Löcher fungieren oft als Aufgabenliste, die den Fokus auf der aktuellen Ebene hält, wobei sie entweder textuell im Editor oder rein mental konstruiert werden können. Zur Füllung dieser Lücken wurden bei den untersuchten Programmierern drei Hauptstrategien identifiziert. Beim Breiten-Erste-Ansatz (eng. breadth-first) wird zunächst ein Skelett der Lösung auf einer einzigen Abstraktionsebene erstellt, bevor fehlende Details ergänzt werden. Im Gegensatz dazu referenziert der Programmierer bei der Tiefen-Erste-Strategie (eng. depth-first) eine noch nicht definierte Variable und schließt diese Lücke fast unmittelbar danach, oft um das Programm schnellstmöglich in einen ausführbaren Zustand zu versetzen. Schließlich

nutzen einige Anwender das Schwer-Erste-Vorgehen (eng. hard-first), bei dem die schwierigsten Löcher priorisiert werden, um die "Planungsunsicherheit" zu verringern, da schwierige Probleme oft weitere Probleme nach sich ziehen. Diese Beobachtungen stammen aus einer Studie mit praktizierenden Programmierern, die über mehrjährige Berufserfahrung verfügen. Es handelt sich somit um Strategien, die von Experten bzw. professionellen Anwendern im Arbeitsalltag eingesetzt werden [24].

Analytisches Denken & Trade-offs Im Rahmen der Theorie zur Softwareentwicklungsexpertise werden analytisches Denken und das Abwägen von Trade-offs explizit als Strategien und Fähigkeiten von Experten klassifiziert. Experten zeichnen sich durch ausgeprägte Problemlösungskompetenzen aus, die analytisches und logisches Denken sowie die Fähigkeit zur Abstraktion und Dekomposition beinhalten. Dazu zählt das Zerlegen großer Probleme in lösbare Teilbereiche, um die Problemdomäne effektiv in den Lösungsraum zu übertragen. Ergänzend dazu ist die Fähigkeit, Kompromisse (eng. Trade-offs) zu bewerten, ein kennzeichnendes Merkmal von Experten. Diese müssen oft zwischen konkurrierenden Zielen wie Designqualität, Wartbarkeit und Performance abwägen und sind dabei in der Lage, zwischen verfrühter Optimierung und langfristig kritischen Designentscheidungen zu unterscheiden [5].

2.3.2. Strategien im Programmierprozess

Explizite Programmierstrategien (High-Level) Explizite Programmierstrategien sind vom Menschen ausführbare Verfahren zur Bewältigung von Programmieraufgaben, die als eine strukturierte Folge von High-Level-Schritten oder Handlungen dokumentiert werden [3]. Ein zentrales Ziel dieser Vorgehensweise ist es, das eigene Wissen so zu externalisieren und zu dokumentieren, dass die Programme und die zugehörigen Lösungsschritte von anderen Personen klar verstanden und erfolgreich ausgeführt werden können [2] [3]. In den Quellen wird hervorgehoben, dass es sich hierbei primär um Expertenstrategien handelt, da erfahrene Entwickler dieses strategische Wissen über Jahre hinweg internalisieren, während es für andere oft unsichtbar bleibt [2]. Um diese Strategien übertragbar zu machen, können sie in einer strukturierten Notation wie "Roboto" verfasst werden, die natürliche Sprache mit einfachen Kontrollflusskonstrukten wie Bedingungen und Schleifen kombiniert [3]. Im Gegensatz dazu agieren Novizen oft ohne solche festen Strategien und verfallen stattdessen in ineffektives Ausprobieren "Trial-and-Error", weshalb das gezielte Unterrichten dieser expliziten Expertenansätze ihnen hilft, Aufgaben systematischer und erfolgreicher zu lösen [20]. Plattformen wie "HowToo" wurden speziell dafür entwickelt, dieses strategische Wissen zu teilen, zu finden und zu kuratieren, um sicherzustellen, dass die Beschreibungen für ein breites Publikum verständlich und in der Praxis direkt anwendbar sind [4]. Letztlich dient diese Form der Dokumentation dazu, die Lücke zwischen dem rein technischen Wissen über Programmiersprachen und der tatsächlichen Problemlösungskompetenz zu schließen [3].

Programmier-Pläne (Sub-Algorithmisch) Die Strategie der Programmier-Pläne (Sub-Algorithmisch) basiert auf der Erkenntnis, dass Experten über ein Repertoire an stereotypischen, abstrakten Lösungen für wiederkehrende Teilprobleme verfügen, die als "Pläne" bezeichnet werden. Diese Pläne operieren auf einer sub-algorithmischen Ebene. Sie bilden meist keine vollständigen Algorithmen, sondern fungieren als Bausteine (z. B. eine Schleife zum Aufsummieren oder ein Schutzmechanismus für Berechnungen), die zu einer Gesamtlösung zusammengesetzt werden müssen [32].
Zu den zentralen Plänen, die explizit unterrichtet werden, gehören:
Initialisation Plan Das explizite Setzen von Startwerten für Variablen (z. B. Zähler oder Summenvariablen) vor deren Verwendung, um undefinierte Zustände zu vermeiden [32].
Sentinel-Controlled Input Plan Ein Schleifenkonstrukt, das Eingaben so lange verarbeitet, bis ein spezifischer "Wächterwert" (Sentinel) eingegeben wird, wobei dieser Wert selbst nicht als operative Datenverarbeitung (z. B. in einer Summe) berücksichtigt werden darf [32].
Guarded Division Plan Eine Strategie zum Schutz vor Laufzeitfehlern, bei der vor einer Division geprüft wird, ob der Divisor Null ist (z. B. mittels einer if-Abfrage), um einen Programmabsturz oder falsche Ergebnisse zu verhindern [32].
Diese Pläne stehen nicht isoliert, sondern werden durch Integrationsmethoden

wie Abutment (Aneinanderreihung), Nesting (Verschachtelung) und Merging (Verschmelzung, z. B. das gleichzeitige Zählen und Summieren in einer einzigen Schleife) zu einem funktionierenden Programm verbunden [32].
In den Quellen wird diese Vorgehensweise explizit als eine Expertenstrategie klassifiziert. Die Studie von de Raadt bestätigte, dass Experten diese Pläne konsistent als Teil ihres "stillen Wissens" (eng. tacit knowledge) anwenden, während Anfänger oft daran scheitern, diese Strukturen selbstständig zu entwickeln. Ziel der Strategie ist es daher, dieses implizite Expertenwissen zu externalisieren und Novizen explizit zu vermitteln [32].

Selbstreguliertes Lernen (SRL) Beim Selbstregulierten Lernen (SRL) im Kontext der Programmierung wenden Lernende proaktive Strategien an, die sich je nach Phase der Aufgabenbearbeitung unterscheiden. In den frühen Phasen dominieren dabei Organisations- und Planungsstrategien, zu denen insbesondere die Informationssuche, die Entwicklung von Arbeitsplänen sowie Team-Meetings zur Definition von Aufgaben gehören. Im weiteren Verlauf des Arbeitsprozesses verschiebt sich der Fokus hin zur Anwendung und Transformation, was die konkrete Implementierung von theoretischem Wissen in praktischen Code umfasst. Ein wesentlicher Bestandteil dieses Prozesses ist die Selbstreflexion, bei der die eigene Leistung überwacht und die Erreichung gesetzter Ziele bewertet wird. Bezüglich des Expertise-Niveaus zeigt sich hierbei folgender Unterschied. Während Studierende (Novizen) diese Reflexion oft nur generisch in Bezug auf das bloße Erreichen von Zielen durchführen, wird eine tiefergehende (Selbst-)Reflexion, wie das Durchdenken von Problemen vor dem Programmieren und das Bewusstsein über eigene Fehleranfälligkeit, explizit als eine Eigenschaft von Experten identifiziert. Auch erfolgreiche Studierende nutzen Strategien wie "Sitzen und Nachdenken" (Reflexion), um Blockaden beim Lernen zu lösen [29].

Inkrementelles/Systematisches Vorgehen Das inkrementelle und systematische Vorgehen wird in der Forschung als eine effektive Strategie für Programmieranfänger (Novizen) identifiziert, die signifikant zu besseren Lernergebnissen beiträgt. Sharma et al. belegen, dass erfolgreiche Novizen (in der Studie als "Intellects" kategorisiert) ihre Unit-Tests seltener ausführen, dafür aber zwischen den Testläufen umfangreichere Code-Änderungen vornehmen. Dieses Verhalten steht im Gegensatz zum "Trial and Error" Ansatz schwächerer Gruppen und deutet darauf hin, dass sich leistungsstarke Anfänger bewusst Zeit nehmen, um den Code systematisch zu ändern und Fehler zu beheben [36].
Ergänzend dazu identifizieren Begum et al. das sogenannte "Short-cut coding" als eine Strategie, die verstärkt von leistungsstarken Novizen ("stronger novice programmers") angewandt wird. Dabei nutzen Studierende ihr Wissen über einfachere Lösungswege oder Code-Wiederverwendung, was eine positive Auswirkung

auf die Benotung hat [7]. Beide Quellen klassifizieren diese systematischen Herangehensweisen explizit als Strategien, die erfolgreiche Novizen von schwächeren Anfängern unterscheiden.

Code-Wiederverwendung Die Strategie der Code-Wiederverwendung, die im Kontext der Softwareentwicklung als "Template"-Strategie bezeichnet wird, umfasst das Suchen nach externen Ressourcen (wie Beispielcode), das Kopieren dieser Fundstücke und deren Nutzung als Vorlage für die eigene Implementierung. In der Untersuchung von LaToza et al. wurde dieser Ansatz bei der Gruppe der "selbstgeleiteten" Entwickler beobachtet, die ihre Vorgehensweise frei wählten, anstatt einer formalisierten Strategie zu folgen. Obwohl dieser Ansatz den Entwicklern einen klaren Startpunkt bot, berichteten diese von Schwierigkeiten, den fremden Code vollständig zu durchdringen, was den Fortschritt teilweise behinderte [22].
In Bezug auf die Einordnung als Experten- oder Novizenstrategie deutet die Literatur darauf hin, dass es sich hierbei eher um eine gängige Standardpraxis handelt als um eine hocheffektive Expertenstrategie. Während Ko und Myers anmerken, dass Experten beim Debugging durchaus Code durchsuchen, um ihre Spekulationen über Fehlerursachen zu überprüfen [21], zeigte sich die Template-Strategie im Design-Kontext bei LaToza et al. als wenig erfolgreich, da nur einer von vier Anwendern die Anforderungen der Aufgabe tatsächlich erfüllen konnte [22].

exploratives Programmieren Beim explorativen Programmieren betrachten Entwickler das Schreiben von Code oft weniger als bloßen Transfer eines mentalen Plans in den Editor, sondern vielmehr als einen "Prozess der Klärung". Dies äußert sich beispielsweise darin, dass Programmierer die oberste Ebene ihrer Datei als "Notizblock" verwenden, um Gedanken zu strukturieren oder Pipelines zu entwerfen, wobei sie teils syntaktisch ungültigen Code als Brainstorming-Hilfe akzeptieren, bevor sie zur eigentlichen Implementierung übergehen. Ergänzend greifen sie häufig auf explizite case-Ausdrücke zurück, um alle möglichen Fälle unabhängig voneinander visuell zu erfassen und zu handhaben. Sobald der Code funktional ist und Muster erkennbar werden, dient dieser explorative Entwurf als Basis für eine Überarbeitung, bei der der Code in elegantere Strukturen, wie etwa Pipelines (Funktionskompositionen), refaktioniert wird. Während streng hierarchische Planungsstrategien oft als Merkmal von Experten gelten, zeigt die Studie, dass auch erfahrene, funktionale Programmierer diese Strategien nutzen, ähnlich wie Muster, die in früheren Arbeiten bei Novizen beobachtet wurden [24].

LLM-unterstützte Codegenerierung Die Strategie der LLM-unterstützten Codegenerierung umfasst den Einsatz von Large Language Models (LLMs)

wie GPT-4, um Programmierlösungen zu erstellen oder bestehenden Code zu optimieren. Ein zentraler Aspekt dieser Methode ist das systematische "Prompt Engineering", das die Interaktion mit dem Modell steuert, um die Qualität der Ergebnisse zu maximieren. Für grundlegende Aufgaben und den Erwerb von Basiswissen erweisen sich einfache Anfragen ohne komplexe Zusatzanweisungen oft als ausreichend, da diese Probleme in den Trainingsdaten der Modelle gut repräsentiert sind. Bei komplexeren Herausforderungen, wie etwa Programmierwettbewerben oder fortgeschrittenen Optimierungsproblemen, werden hingegen differenzierte Strategien benötigt. Dazu gehören "Multi-Step Conversational Prompts", die den Lösungsprozess in iterative Schritte (z. B. Pseudocode-Erstellung, Logik-Überprüfung) unterteilen , sowie "Specific Prompt Engineering", das detaillierte Anweisungen für spezifische Szenarien liefert. Diese fortgeschrittenen Strategien werden explizit für Lernende empfohlen, die bereits über ein fundiertes Verständnis der Konzepte verfügen, und zielen somit auf ein Experten- bzw. Fortgeschrittenenniveau ab [38].

2.3.3. Strategien in der Programmanalyse

Programmverständnis & Scent-Following Das Programmverständnis ist eine essentielle Voraussetzung für effektives Debugging, da Entwickler den Quellcode explorieren müssen, um die relevanten Stellen für eine Änderung zu lokalisieren [9]. In der Praxis wenden professionelle Entwickler hierfür strukturierte, wiederkehrende Strategien an, wobei Erfahrung eine entscheidende Rolle spielt, um relevante Einstiegspunkte zu identifizieren und irrelevanten Code effizient zu filtern. Im Gegensatz dazu tendieren unerfahrene Entwickler eher dazu, das Verständnis auf das Nötigste zu beschränken, etwa durch das Klonen von Code, um komplexe Zusammenhänge zu umgehen. Die Information Foraging Theory (IFT) modelliert dieses Navigationsverhalten analog zur Nahrungssuche in der Natur. Entwickler folgen einem "Information Scent", der wahrgenommenen Wahrscheinlichkeit, dass ein Hinweis im Code, wie etwa Variablennamen oder Kommentare, zur Lösung führt. Empirische Untersuchungen belegen, dass dieses "Scent-Following" bei der Fehlersuche weitaus häufiger beobachtet wird als das explizite Bilden von Hypothesen [33].

Kognitive Modelle & Abstraktion Programmierstrategien, die auf kognitiven Modellen und Abstraktion basieren, sind essenziell für das Verständnis von Code und die Problemlösung. Dabei spielen Hypothesen eine zentrale Rolle. Frühere Theorien beschreiben einen Top-Down-Prozess, bei dem Programmierer Hypothesen über die Programmstruktur bilden und diese durch das Identifizieren von "Beacons" (markanten Code-Merkmalen) verifizieren. Andere beschreiben kognitive Modelle, die sowohl Bottom-Up- als auch Top-Down-Hypothesenbildung beinhalten [23]. In der Literatur wird hierbei explizit zwischen Experten und Anfängern unterschieden. Während Novizen oft mangels vorhandener mentaler Strukturen auf Trial-and-Error-Methoden oder Mittel-Ziel-Analysen zurückgreifen, verfügen Experten über entwickelte Schemata, die es ihnen ermöglichen, Informationen effizient zu kategorisieren, Muster abzugleichen und abstrakte Repräsentationen zu nutzen. Um Blockaden zu überwinden, setzen erfolgreiche Studenten gezielt Abstraktionsstrategien der Kategorie "Abstract/understand stuff" ein, zu denen unter anderem "Divide and Conquer", das Einnehmen einer Vogelperspektive sowie der Transfer von Wissen durch das Herstellen von Verbindungen und das Erkennen von Mustern gehören [26]. Auch Sharma et al. betonen, dass Problemlösungsstrategien wie "Divide and Conquer" oder "Top-Down" gelehrt werden sollten, da leistungsstarke Studenten ("Intellects") im Gegensatz zu schwächeren "Probers" seltener testen und Code gezielter verändern, statt sich auf Versuch und Irrtum zu verlassen [36].

Compiler als Assistent Die Strategie "Compiler als Assistent" beschreibt eine fortgeschrittene Arbeitsweise professioneller Entwickler, bei der der Compiler über seine klassische Rolle der Code-Übersetzung hinaus als interaktives Werkzeug zur

Steuerung des Entwicklungsprozesses genutzt wird. Nach Lubin und Chasins lässt sich diese Nutzung in zwei Hauptfunktionen unterteilen:

- Als korrektives Werkzeug dient der Compiler dazu, das mentale Modell des Entwicklers über die Problemdomäne zu validieren. Entwickler führen den Compiler hierbei oft bewusst auf unvollständigem Code aus, um durch Fehlermeldungen zu prüfen, ob ihre Annahmen über Typen und Beziehungen korrekt sind, bevor sie die eigentliche Logik implementieren [24].
- Als direktives Werkzeug fungiert der Compiler als dynamische "To-Do-Liste": Entwickler nehmen gezielte Änderungen vor (z. B. an einer Typdefinition) und arbeiten anschließend systematisch die resultierenden Fehlermeldungen ab, um Refactorings durchzuführen oder Implementierungslücken zu schließen [24].

Ergänzend hierzu beobachten Roehm et al. bei professionellen Entwicklern die Nutzung des Compilers zur Gewinnung struktureller Informationen [33]. Ein konkretes Beispiel hierfür ist das absichtliche Umbenennen von Konstanten oder Methoden, um über die daraus resultierenden Compiler-Fehler alle Verwendungsstellen im Code zu identifizieren. Diese Strategie dient teils als Ersatz für nicht bekannte IDE-Funktionen [33]. Auch beim "Cloning" von Code wird der Compiler genutzt. Entwickler kommentieren große Codeblöcke aus und reaktivieren nur jene Methoden, die der Compiler als fehlend meldet, um unnötigen Code auszufiltern [33]. Beide Studien untersuchen professionelle Entwickler, wobei Lubin und Chasins die Methode qualitativ als eine fortgeschrittene Technik klassifizieren, während Roehm et al. anmerken, dass die Nutzung des Compilers zur Strukturanalyse (statt spezialisierter Tools) auch auf Unkenntnis von IDE-Features bei erfahrenen Entwicklern hindeuten kann [24] [33].

Interaktion mit der UI & Startpunkten Um Software effizient zu verstehen, versetzen sich professionelle Entwickler häufig in die Rolle des Endanwenders und nutzen die Interaktion mit der grafischen Benutzeroberfläche (UI) als zentrale Verständnishilfe [33]. Diese Strategie dient zwei Hauptzwecken. Zum einen testen Entwickler durch das Eingeben von Werten oder das Klicken von Schaltflächen, ob sich die Anwendung wie erwartet verhält und ob ihre Annahmen über die Funktionsweise korrekt sind [33]. Zum anderen nutzen sie diese Interaktion gezielt, um konkrete Startpunkte für die Code-Analyse zu finden, beispielsweise indem sie identifizieren, welche Methoden durch einen bestimmten Button-Klick im Code ausgelöst werden [33]. Dieser Ansatz wird oft als Alternative zum reinen Lesen von Quellcode oder zum Debugging gewählt [33].
Sobald ein Einstiegspunkt gefunden ist, wenden Entwickler eine Filterstrategie an. Sie ignorieren Code-Bereiche, die für die aktuelle Aufgabe als irrelevant erachtet werden (z. B. reine Berechnungslogik, wenn es um ein UI-Problem geht), um kognitive Ressourcen zu sparen [33]. Die Quellen heben hervor, dass es sich hierbei

um eine expertenabhängige Strategie handelt [33]. Es wird explizit betont, dass die Erfahrung der Entwickler eine entscheidende Rolle spielt, um korrekte Startpunkte schnell zu identifizieren und irrelevante Code-Teile sicher auszufiltern [33].

Wissensnutzung & Mustererkennung Die Nutzung von Vorwissen und die Entwicklung mentaler Schemata stellen fundamentale Unterscheidungsmerkmale zwischen Experten und Novizen dar. Experten zeichnen sich durch den Besitz ausgeprägter Schemata aus, die es ihnen ermöglichen, Informationen effizient zu kategorisieren, zu verarbeiten und Problemlösungsstrategien basierend auf Ähnlichkeiten mit früheren Erfahrungen auszuwählen [27]. Da Schemata Bedeutung und Beziehungen organisieren, verfügen Experten über abstraktes und übertragbares Wissen, das sie auch in neuen, aber verwandten Kontexten anwenden können [27][5]. Im Gegensatz dazu fehlen Novizen häufig ausreichende nützliche Schemata für die Problemlösung, weshalb sie oft nur über unzusammenhängende Wissensfragmente verfügen und auf mühsame Methoden wie "Trial and Error" oder Mittel-Ziel-Analysen zurückgreifen müssen [27]. Die Bildung solcher Schemata gilt daher als ein zentrales Ziel, um aus einem Anfänger einen Experten zu machen [27].

Bei der konkreten Analyse von Code durch Lernende lassen sich verschiedene Formen der Mustererkennung beobachten, die teils als Teststrategien und teils als Verständnisstrategien fungieren [12]. Eine von Novizen angewandte Strategie ist die Mustererkennung basierend auf externem Wissen, bei der syntaktische Komponenten wie Schleifenstrukturen identifiziert werden, um deren Verhalten aufgrund bekannter Regeln vorherzusagen [12]. Eine weitere, oft fehleranfällige Strategie von Anfängern ist die zeitlich selbstreferenzielle Mustererkennung, bei der Lernende annehmen, dass eine Aufgabe einer vorherigen Aufgabe im selben Test ähnelt [12]. Diese Annahme führt häufig zu falschen Schlussfolgerungen, da sie eher eine Test-Taktik als eine echte Strategie zum Codeverständnis darstellt [12].

Es zeigen sich jedoch auch bei Lernenden bereits anspruchsvollere Ansätze, die der Arbeitsweise von Experten näherkommen. So nutzen einige Studierende die Mustererkennung, um eine höhere semantische Bedeutungsebene aus dem Code abzuleiten [12]. Anstatt den Code nur mechanisch abzuarbeiten, erfassen sie intuitiv den Zweck des Programms, indem sie es mit bekannten Algorithmen, wie etwa einem Sortierverfahren, verknüpfen [12]. Diese Strategie wird mit der Evaluationsstufe der Bloomschen Taxonomie in Verbindung gebracht und stellt einen tieferen Lernansatz dar, da hier Prinzipien und Muster gesucht werden, anstatt nur Fakten abzurufen [12]. Ergänzend dazu verwenden Lernende Strategien zur Erschließung neuer semantischer Konzepte, indem sie unbekannte Ideen aktiv mit bereits verstandenem Vorwissen in Beziehung setzen oder Beispiele durcharbeiten, um das Verständnis zu sichern [12].

SRL: Auflösung von Schwierigkeiten Die Strategie zur Auflösung von Schwierigkeiten im Rahmen des selbstregulierten Lernens zeichnet sich durch eine Vielfalt an Lösungsansätzen aus, wobei das Einholen sozialer Unterstützung (eng. Seeking Social Assistance, SOA) eine zentrale Rolle spielt [29]. Um Blockaden zu überwinden, wenden sich Studierende dabei primär an Lehrende, Teamkollegen sowie erfahrene Kommilitonen oder externe Personen [29]. Ergänzend wird die Interaktion in Online-Communities genutzt, wobei Studierende diese spezifische Ressource vorwiegend in den frühen Projektphasen als Mittel zur Problemlösung angaben [29]. Wenngleich die Quellen diese Vorgehensweise nicht explizit als reine Experten- oder Novizenstrategie klassifizieren, ordnen die Autoren das Hilfesuchen in professionellen Netzwerken als gängige Praxis der realen Arbeitswelt ein. Die Anwendung dieser Strategien dient folglich dazu, den Übergang von der Programmierung auf Einstiegsniveau hin zu professionellen Arbeitsumgebungen und fortgeschrittenen Kompetenzen zu erleichtern [29].

2.3.4. Strategien im Debugging

Strategischer Dreiklang Die als "Strategischer Dreiklang" bezeichnete Programmierstrategie spiegelt exakt den in der DBGBENCH-Studie untersuchten professionellen Workflow wider, der sich strikt in die Phasen der Fehlerlokalisierung (eng. Fault Localization), der Fehlerdiagnose (eng. Bug Diagnosis) und der Fehlerbehebung (eng. Software Repair) gliedert [9]. In den Quellen wird dieses strukturierte Vorgehen explizit als Expertenstrategie identifiziert. Die Autoren verwarfen die Arbeit mit Studenten nach einer Vorstudie, da diese signifikant ineffizienter waren und durchschnittlich acht Stunden für einen einzigen Fehler benötigten, während professionelle Softwareingenieure 27 Fehler in 21,5 Stunden behoben [9]. Im Gegensatz zu einer simplen Fokussierung auf einzelne Codezeilen zeichnet sich die Expertendiagnose durch eine Kausalanalyse aus, welche die komplexe Interaktion zwischen mehreren Anweisungen und Ereignissen erklärt, die zum Fehler führen [9]. Obwohl Experten diesen systematischen Dreiklang anwenden, zeigt die Studie auch die Komplexität dieses Prozesses auf, da selbst erfahrene Entwickler gelegentlich plausible, aber inkorrekte Änderungen verfassen, die eher Symptome behandeln als die eigentliche Ursache zu beheben [9].

Traditionelle & Manuelle Strategien Traditionelle und manuelle Programmierstrategien konzentrieren sich primär auf die Beobachtung des Programmflusses und des Zustands, um Fehler im Code systematisch zu lokalisieren [39]. Zu diesen grundlegenden Techniken gehören das Programmlogging durch das Einfügen von print-Statements sowie die Verwendung von Assertions, um spezifische Bedingungen während der Laufzeit zu verifizieren [39]. Ergänzend dazu erlauben Breakpoints das gezielte Anhalten der Programmausführung, während Profiling-Verfahren zur Überwachung von Metriken wie Speichernutzung oder Ausführungszeit eingesetzt werden [39]. Eine weitere zentrale Methode ist die Hand-Simulation, bei der Entwickler den Code durch "Desk-checking" oder "Walkthroughs" gedanklich oder schriftlich nachvollziehen [12].

Die manuellen Strategien werden in der Forschung explizit als eine Herausforderung für Novizen beschrieben, da diese oft über weniger Wissen verfügen und Schwierigkeiten haben, den Programmzustand ohne externe Hilfsmittel korrekt zu verfolgen [40]. Im Gegensatz dazu wenden Experten diese manuellen Techniken oft intuitiver an, wobei sie komplexe Aktivitäten wie das Navigieren von Programmabhängigkeiten durchführen, die von automatisierten Werkzeugen allein oft nicht vollständig ersetzt werden können [28]. Studien zeigen jedoch, dass selbst Novizen eine Vielzahl dieser Strategien einsetzen, ihre Anwendung jedoch häufig unsystematisch bleibt oder je nach Problemstellung stark variiert [12].

Code Tracing / Following Execution Code Tracing, oft auch als "Hand Simulation" bezeichnet, ist das systematische Verfolgen der Programmausführung,

um den Kontroll- und Datenfluss zu emulieren, was als notwendige Voraussetzung für das Schreiben, Debuggen und Warten von Code gilt [40] [23]. Da Programmieranfänger häufig Schwierigkeiten haben, fragile mentale Modelle besitzen und unter einer hohen kognitiven Belastung leiden, wenn sie versuchen, Variablenwerte im Gedächtnis zu behalten, schlagen Xie et al. eine explizite Strategie vor, die das strikte zeilenweise Abarbeiten des Codes mit dem Erstellen einer externen "Memory Table" (Speichertabelle) kombiniert [40]. Diese Strategie unterstützt den Lernprozess, indem sie Novizen dazu anleitet, für jeden Methodenaufruf eine neue Tabelle zu erstellen, um den Gültigkeitsbereich (eng. Scope) von Variablen korrekt abzubilden. Anstatt Variablenwerte nur im Kopf zu aktualisieren oder zu radieren, werden in dieser Tabelle neue Werte eingetragen und alte durchgestrichen, wodurch der Verlauf der Datenänderungen sichtbar bleibt und das Arbeitsgedächtnis entlastet wird [40].

In der Literatur wird hinsichtlich der Anwendung dieser Strategie zwischen verschiedenen Erfahrungsstufen unterschieden. Während Experten das Tracing ohne Werkzeuge beherrschen müssen, um Code tiefgreifend zu verstehen, zeigt die Forschung von McCartney et al., dass auch erfolgreiche Studierende das Tracing gezielt als konkrete Handlungsstrategie ("Concrete / do stuff") einsetzen, um Verständnis-
blockaden zu lösen [40] [26]. Ergänzend dazu beschreiben Lawrance et al. in Abgrenzung zur manuellen Simulation die Strategie des "Following Execution", die durch moderne Entwicklungsumgebungen ermöglicht wird. Hierbei handelt es sich um eine "Forward-Reasoning"-Strategie, bei der Programmierer Werkzeuge wie Debugger nutzen, um die Ausführung für einen bestimmten Input Schritt für Schritt zu verfolgen und dabei zu beobachten, welche Codezeilen die Daten verändern, anstatt die Berechnungen rein mental durchzuführen [23].

Systematische Fehlerlokalisierung (Explizit) Die Systematische Fehlerlokalisierung (Explizit) ist eine Programmierstrategie, die darauf abzielt, Softwarefehler durch methodisches Rückwärtsarbeiten von der fehlerhaften Ausgabe bis zur Fehlerursache zu identifizieren, anstatt sich auf das Raten von Ursachen zu verlassen [22]. Diese Strategie basiert auf dem Algorithmus des "Precise Backwards Dynamic Slicing", der ursprünglich im automatisierten Debugging-Tool "Whyline" implementiert wurde und hier als manuell ausführbare Prozedur für Entwickler adaptiert ist [22].
Der Kern der Strategie besteht darin, systematisch die Kette der Kausalität zurückzuverfolgen:

- Zuerst wird verifiziert, ob die Codezeile, die die falsche Ausgabe erzeugte, überhaupt ausgeführt wurde und ob sie selbst defekt ist.
- Falls die Zeile korrekt ist, wird die Quelle des falschen Werts identifiziert, der in dieser Zeile verwendet wurde (Verfolgung von Datenabhängigkeiten).

- Die Strategie weist den Entwickler an, alle Zeilen zu sammeln, die diesen falschen Wert produziert haben könnten und diese systematisch zu überprüfen.
- Dieser Prozess wird rekursiv fortgesetzt, bis die eigentliche Fehlerursache gefunden ist.

In den Quellen wird diese Vorgehensweise explizit als Expertenstrategie bezeichnet [22]. Sie beschreibt, wie Experten typischerweise Probleme lösen, und wurde in Studien verwendet, um Novizen (oder weniger erfahrenen Entwicklern) zu helfen, systematischer und erfolgreicher zu arbeiten, indem sie das oft unsystematische "Guessing and Checking" ersetzt, das Novizen häufig anwenden [22].

Trial and Error / Guess & Check Die Programmierstrategie "Trial and Error" wird explizit als eine Vorgehensweise von Novizen identifiziert, wobei insbesondere schwächere Anfänger diese unsystematische Methode häufig anwenden [7]. Diese Strategie ist durch eine hohe Frequenz von Testläufen bei gleichzeitig nur minimalen Codeänderungen geprägt und korreliert nachweislich negativ mit dem Lernerfolg sowie der Leistung bei Programmieraufgaben [36]. Ergänzend dazu wird der Ansatz des "Guess & Check" als eine wenig hilfreiche und zeitaufwendige Methode beschrieben, da Entwickler hierbei oft nur auf Basis vager Vermutungen agieren [22]. Während Experten auf strukturierte und explizite Vorgehensweisen setzen, führt dieses unsystematische Vorgehen bei Entwicklern häufig in Sackgassen, weshalb die Strategie in der Praxis meist zugunsten systematischerer Prozesse aufgegeben wird [3] [22].

Output-Zentrierte Strategien (Whyline) Die Whyline-Technik, bekannt als "Interrogatives Debugging", nutzt menschliche Intelligenz, um den Fehlersuchprozess gezielt zu unterstützen [41]. Dieser Ansatz wurde primär entworfen, um Novizen bei der Formulierung von Hypothesen und dem Stellen intuitiver Fragen zum Programmverhalten zu helfen. Die Strategie ermöglicht es Entwicklern, direkt mit der Software zu "kommunizieren", indem sie "Why did"- oder "Why not"-Fragen zur Programmausgabe stellen [41]. Ein typisches Beispiel für eine solche Interaktion ist die Frage: "Warum ist dieses Rechteck blau?" [28]. Das Werkzeug kombiniert hierfür Visualisierungen, dynamisches Slicing und automatische Schlussfolgerungen, um den Nutzer direkt zu den für das Verhalten verantwortlichen Anweisungen zu führen [28]. In empirischen Studien konnten Teilnehmer unter Verwendung von Whyline ihre Aufgaben doppelt so schnell abschließen als mit einem traditionellen Debugger [28]. Neben der rein technischen Unterstützung kann die bewusste Auseinandersetzung mit solchen Strategien den Lernprozess fördern, wobei ein Teilnehmer angab, dass dies ihm helfe, aus Fehlern zu lernen und darüber nachzudenken, was er in zukünftigen Arbeitsprojekten tun sollte [3].

Slicing-basierte Techniken Slicing-basierte Techniken sind spezialisierte Methoden zur Extraktion derjenigen Programmteile, welche die an einem spezifischen Programmpunkt berechneten Werte, das sogenannte Slicing-Kriterium, beeinflussen können. Diese Verfahren dienen primär dazu, die Suchdomäne für Fehler signifikant zu reduzieren, indem Abhängigkeiten entlang des Informationsflusses verfolgt werden [13]. Das statische Program Slicing nutzt hierfür Systemabhängigkeitsgraphen (SDG), während sich das Dynamic Slicing auf die Identifizierung von Programmteilen konzentriert, die bei einer spezifischen Programmausführung tatsächlich involviert sind [13]. Eine fortgeschrittene Variante stellt das "Conditional Trace Slicing" (CTS) dar, welches speziell für textuell umfangreiche Berechnungen in der "Rewriting Logic" entwickelt wurde, um die Größe von Ausführungsspuren durch das automatische Ausfiltern irrelevanter Daten drastisch zu verringern [1]. In der Forschung werden diese Techniken als traditionelle Strategien für menschliche Programmierer charakterisiert, die zur präzisen Fehlerlokalisierung eingesetzt werden, wenn automatisierte statistische Analysen allein nicht ausreichen. Während Experten werkzeuggestützte Abhängigkeitsanalysen wie das Slicing für komplexe Aufgaben nutzen, gilt das grundlegende, werkzeuglose Tracing als essenziell für das Expertenverständnis von Code. Novizen haben häufig Schwierigkeiten mit Tracing-Aufgaben [40]. Daher wird das Verfolgen von Abhängigkeiten mittels Slicing als eine Expertenstrategie zur Bewältigung anspruchsvoller Debugging Szenarien in großen Systemen eingestuft.

Automatisierte Fehlerlokalisierung (FL) Die automatisierte Fehlerlokalisierung (FL) bezeichnet den Prozess der Identifizierung von Fehlerstellen in einem Programm. Da die manuelle Suche nach Fehlern aufgrund der zunehmenden Größe und Komplexität moderner Software oft nicht mehr durchführbar ist, besteht ein hoher Bedarf an Techniken, die Entwickler mit minimalem menschlichem Eingriff zu den Fehlerquellen leiten [39]. Ein gängiger Ansatz vieler Techniken besteht darin, Programmkomponenten nach ihrer "Verdächtigkeit" (eng. suspiciousness) zu rangieren, um dem Programmierer eine priorisierte Liste potenziell fehlerhafter Stellen zu liefern. Die spektrumsbasierte Fehlerlokalisierung (SBFL) ist eine weit verbreitete Kategorie, die Programmspektren aus erfolgreichen und fehlgeschlagenen Testläufen nutzt, um diese Verdächtigkeit zu berechnen [39]. Untersuchungen belegen, dass diese Werkzeuge als Expertenstrategie bei einfacheren Aufgaben tatsächlich dazu beitragen können, Fehler schneller zu finden. Im Gegensatz dazu profitieren Novizen oft nicht von diesen Techniken und sind teilweise nicht in der Lage, Aufgaben innerhalb eines Zeitlimits abzuschließen [28].

3. Methodik

3.1. Konstruktion des Fragebogens

Um die theoretisch fundierten Programmierstrategien messbar zu machen, wurden die in Kapitel 2.3.1 bis 2.3.4 beschriebenen Konzepte in konkrete Selbsteinschätzungs-
fragen (Items) überführt. Ziel dieser Operationalisierung ist es, die Anwendungshäufigkeit und das subjektive Kompetenzempfinden der Probanden zu erfassen.

Die Konstruktion der Items folgte einem deduktiven Ansatz, bei dem die Kerncharakteristika der jeweiligen Strategie direkt aus der Literatur abgeleitet wurden. Dabei wurden pro Strategiebereich jeweils ein bis zwei komplementäre Fragen formuliert, um eine höhere Validität der Antworten zu gewährleisten.

3.1.1. Designprozess: Herleitung der Items aus den Strategien

Die Operationalisierung der theoretischen Konstrukte in messbare Indikatoren erfolgte durch eine systematische Ableitung der Kernpraktiken jeder Strategie. Ziel war es, sowohl die operative Anwendung als auch das dahinterliegende methodische Verständnis abzufragen.

Designprozess: Übersicht der Fragen Tabelle 3.1 fasst die entwickelten Fragen und ihre theoretische Zuordnung zusammen.

Table 3.1.: Operationalisierung der Programmierstrategien (Designprozess)

Nr.	Frage zur Selbsteinschätzung	Zugehörige Strategie
D1	Ich setze bei der Entwicklung agile Prinzipien (z.B. Scrum) zur Steuerung des Gesamtprozesses ein.	Agile & Lean Methoden
D2	Ich nutze Abstimmungsrunden (z.B. Daily Meetings), um die Aufgabenverteilung zu klären.	Agile & Lean Methoden

Fortsetzung auf der nächsten Seite

Tabelle 3.1 fortgesetzt

Nr.	Frage zur Selbsteinschätzung	Zugehörige Strategie
D3	Ich definiere notwendige Tests für eine Funktion, bevor ich mit der eigentlichen Code-Implementierung beginne.	Test-Driven Development (TDD)
D4	Ich implementiere erst neue Funktionalitäten, wenn alle bereits existierenden Tests erfolgreich durchlaufen.	Test-Driven Development (TDD)
D5	Ich verwende bekannte Entwurfsmuster (Design Patterns) als bewährte Lösungen für wiederkehrende Architekturprobleme.	Design Patterns
D6	Ich kann Entwurfsmusterpräzise benennen und in strukturierten Sammlungen wiederfinden und anwenden.	Design Patterns
D7	Ein komplexes Problem zerlege ich aktiv in kleinere, voneinander unabhängig implementierbaren Unterprobleme.	Dekomposition
D8	Ich beginne eine Entwurfsaufgabe mit einer klaren funktionalen Zerlegung, um die Gesamtstruktur zu definieren.	Dekomposition
D9	Ich beginne die Implementierung einer Funktion, indem ich zuerst festlege, welche Typen sie benötigt und zurückgibt, bevor ich den Funktionskörper schreibe.	Typkonstruktion
D10	Ich definiere zuerst die benötigten Datentypen, um die logische Struktur des Problems top-down zu klären.	Typkonstruktion
D11	Ich definiere bewusst unvollständige Code-Abschnitte ("Programmlöcher"), um zuerst das Gesamt-Skelett der Lösung zu erstellen.	Skizzierung (Sketching)
D12	Ich implementiere zuerst die schwierigsten oder komplexesten Codeteile.	Skizzierung (Sketching)
D13	Ich wäge Kompromisse (Trade-offs) zwischen konkurrierenden Zielen wie Performance und Wartbarkeit analytisch ab.	Analytisches Denken
D14	Meine Entwurfsentscheidungen basieren auf einer logischen Analyse der Vor- und Nachteile jeder möglichen Lösung.	Analytisches Denken

Agile und Lean Methoden (D1–D2): Die Items **D1** und **D2** adressieren die operative Steuerungsebene. Während agile Methoden primär als Antwort auf volatile Anforderungen beschrieben werden [30], dient Item **D1** der Erfassung, ob diese Rahmenwerke (z.B. Scrum) bewusst zur Prozesssteuerung eingesetzt werden. Da die tägliche Synchronisation ein zentrales Element zur Identifikation von Hindernissen darstellt, wurde Item **D2** spezifisch auf *Daily Meetings* und die damit verbundene Aufgabenverteilung ausgerichtet [30].

Test-Driven Development (D3–D4): Die Operationalisierung von TDD stützt sich auf das Prinzip des sauberen Codes durch iterative Zyklen [6]. Item **D3** prüft die fundamentale "Test-First"-Vorgabe (die "Red"-Phase), während Item **D4** die Disziplin innerhalb des Prozesses erfasst, um neue Funktionalität nur bei stabilen Basistests zu implementieren. Dies ist besonders relevant, da Novizen häufig Schwierigkeiten zeigen, diesen systematischen Overhead gegenüber intuitivem Programmieren beizubehalten [22].

Design Patterns (D5–D6): Entwurfsmuster fungieren in der Softwareentwicklung als kondensiertes Expertenwissen [8]. Item **D5** zielt auf die Anwendung dieser Muster als Architekturwerkzeuge ab, um wiederkehrende Probleme zu lösen. In Ergänzung dazu adressiert Item **D6** die Rolle von Mustern als "Lingua Franca". Die Fähigkeit, Muster präzise zu benennen und in Katalogen (wie der *Gang of Four*) zu identifizieren, gilt als Indikator für eine ausgeprägte kommunikative Expertise [14, 8].

Dekomposition (D7–D8): Die Fähigkeit, ein komplexes Gesamtproblem in handhabbare Teilaufgaben zu zerlegen, wird als zentrales Merkmal von Experten angesehen [5]. Item **D7** operationalisiert diese Fähigkeit zur Bildung unabhängiger Unterprobleme. Item **D8** fokussiert hingegen auf den Zeitpunkt der Anwendung, da die bewusste funktionale Zerlegung zu Beginn der Entwurfsaufgabe maßgeblich zur Organisation komplexer Anforderungen beiträgt [22].

Typkonstruktion (D9–D10): Strategien in der statisch typisierten Programmierung sind durch einen stetigen Wechsel zwischen hierarchischen und opportunistischen Methoden geprägt [24]. Die Items **D9** und **D10** erfassen das Prinzip des "following the types", bei dem Typannotationen als strukturbildendes Element der Top-Down-Zerlegung genutzt werden, um Designentscheidungen ergonomisch abzusichern, bevor die eigentliche Implementierung erfolgt [24].

Skizzierung / Sketching (D11–D12): Diese Items basieren auf der Identifikation von Programmlöchern als Mechanismus zur kognitiven Entlastung [24]. Item **D11** prüft, ob Probanden bewusst unvollständige Skelette nutzen, um die übergeordnete Absicht zu wahren. Item **D12** operationalisiert die "Hard-First"-Strategie, bei der die schwierigsten Lücken priorisiert werden, um die Planungsunsicherheit frühzeitig zu minimieren [24].

Analytisches Denken & Trade-offs (D13–D14): Abschließend wird die Kompetenz zum Abwägen konkurrierender Ziele operationalisiert [5]. Item **D13** fragt explizit nach dem Bewusstsein für Kompromisse, wie etwa zwischen Performance und Wartbarkeit. Item **D14** dient der Validierung, ob Entwurfsentscheidungen auf einer logischen Analyse von Vor- und Nachteilen basieren, was Experten von einer eher ungeplanten Ausführung bei Novizen unterscheidet [19].

3.1.2. Programmierprozess: Herleitung der Items aus den Strategien

In Ergänzung zum Designprozess werden in diesem Abschnitt die Strategien während der eigentlichen Implementierungsphase operationalisiert. Die Items zielen darauf ab, den Übergang von theoretischen Programmierkonzepten hin zur praktischen Problemlösungskompetenz messbar zu machen.

Designprozess: Übersicht der Fragen Die folgende Tabelle 3.2 fasst die entwickelten Fragen und ihre theoretische Zuordnung zusammen.

Table 3.2.: Operationalisierung der Programmierstrategien (Programmierprozess)

Nr.	Frage zur Selbsteinschätzung	Zugehörige Strategie
P1	Ich nutze detaillierte, schrittweise Anleitungen, um konsistente Ergebnisse zu erzielen.	Explizite Strategien
P2	Ich dokumentiere meine Programme so, dass sie von anderen klar verstanden und ausgeführt werden können.	Explizite Strategien
P3	Ich weise allen Variablen einen Startwert zu, bevor ich sie in Berechnungen verwende.	Programmier-Plans
P4	Bei Berechnungen achte ich aktiv auf Spezialfälle, die zu Fehlern führen könnten (z.B. die Division durch Null).	Programmier-Plans
P5	Ich überwache meinen Arbeitsfortschritt aktiv und reflektiere meine Ziele, um meine nächsten Schritte zu planen.	SRL
P6	Ich organisiere meine Arbeitsschritte und entwickle einen Arbeitsplan, bevor ich mit der eigentlichen Implementierung beginne.	SRL
P7	Ich programmiere größere Code-Blöcke auf einmal, bevor ich den Code ausführe.	Inkrementelles Vorgehen
P8	Bei der Gestaltung eines Programms suche ich nach einem existierenden Code-Beispiel in externen Quellen, um es als Vorlage anzupassen.	Code-Wiederverwendung

Fortsetzung auf der nächsten Seite

Tabelle 3.2 fortgesetzt

Nr.	Frage zur Selbsteinschätzung	Zugehörige Strategie
P9	Ich schreibe bewusst experimentellen Code, um Erkenntnisse zu gewinnen, die ich später zur Verfeinerung der endgültigen Lösung nutze.	Exploratives Programmieren
P10	Während des Programmierens schreibe ich mir temporäre Notizen direkt in den Code, um meine Gedanken zu strukturieren, bevor ich die endgültige Implementierung vornehme.	Exploratives Programmieren
P11	Ich nutze LLM-Tools (z.B. ChatGPT) aktiv zur Generierung von Lösungsvorschlägen für Programmieraufgaben.	LLM-Unterstützung

Die Ableitung der Items (P1–P11) orientiert sich an der Unterscheidung zwischen expliziten Expertenansätzen und intuitiven Novizenstrategien innerhalb des Programmierprozesses.

Explizite Programmierstrategien (P1–P2): Diese Strategien dienen der Externalisierung von Wissen, um den Lösungsweg sowohl für sich selbst als auch für Dritte transparent zu gestalten. Item **P1** erfasst die Nutzung strukturierter, schrittweiser Anleitungen, die als High-Level-Verfahren zur Bewältigung komplexer Aufgaben dokumentiert werden [3]. Item **P2** adressiert die Dokumentation als Mittel zur Wissensübertragung, da die Verständlichkeit der Lösungsschritte für Außenstehende als Kernmerkmal dieser Expertenstrategie gilt [2].

Programmier-Pläne (P3–P4): Programmier-Pläne repräsentieren stereotype Lösungen für sub-algorithmische Teilprobleme, die Experten als "stilles Wissen" internalisiert haben. Item **P3** operationalisiert den *Initialisation Plan* durch die Abfrage, ob Variablen systematisch Startwerte zugewiesen werden [32]. Der Schutz vor Laufzeitfehlern wird durch Item **P4** in Form des *Guarded Division Plan* erhoben, welcher die explizite Prüfung von Divisoren auf Nullwerte beinhaltet [32].

Selbstreguliertes Lernen (SRL) (P5–P6): Die proaktive Steuerung des Lern- und Arbeitsprozesses ist ein Kennzeichen erfolgreicher Entwickler. Item **P5** zielt auf die Phase der Selbstreflexion ab, in der der Arbeitsfortschritt aktiv überwacht und Ziele bewertet werden [29]. Item **P6** erfasst die initiale Planungsphase, wobei die Entwicklung eines Arbeitsplans vor der Implementierung als differenzierendes Merkmal zwischen Experten und Novizen dient [29].

Inkrementelles / Systematisches Vorgehen (P7): Im Gegensatz zum

ungeplanten "Trial-and-Error" zeichnen sich leistungsstarke Programmierer durch umfangreichere, aber systematische Code-Änderungen zwischen Testläufen aus. Item **P7** erhebt dieses Verhalten, indem die Tendenz abgefragt wird, größere Blöcke vor der Ausführung zu implementieren, was auf eine bewusstere Auseinandersetzung mit der Logik hindeutet [36].

Code-Wiederverwendung (P8): Die Nutzung externer Ressourcen als Startpunkt für die eigene Entwicklung wird oft als Standardpraxis beobachtet. Item **P8** operationalisiert diese sogenannte "Template"-Strategie, bei der existierende Code-Beispiele als Vorlage kopiert und angepasst werden [22].

Exploratives Programmieren (P9–P10): Das Programmieren wird hierbei als "Prozess der Klärung" verstanden, bei dem der Editor als Brainstorming-Hilfe fungiert. Item **P9** adressiert das Schreiben von experimentellem Code zur Erkenntnisgewinnung, während Item **P10** die Nutzung von Notizen zur mentalen Strukturierung innerhalb der Datei erfasst [24].

LLM-Unterstützung (P11): Der Einsatz von Large Language Models zur Codegenerierung stellt eine moderne Erweiterung des Strategierepertoires dar. Item **P11** erhebt die Nutzung von Tools wie ChatGPT zur Erstellung von Lösungsvorschlägen, wobei fortgeschrittene Anwender diese Werkzeuge oft für komplexe Optimierungen oder iterative Logik-Überprüfungen einsetzen [38].

3.1.3. Programmanalyse: Herleitung der Items aus den Strategien

Der folgende Teil befasst sich mit der Programmanalyse. Hierbei steht im Vordergrund, wie Entwickler existierenden Code explorieren, verstehen und mentale Modelle aufbauen, um Fehler zu lokalisieren oder Erweiterungen zu planen.

Programmanalyse: Übersicht der Fragen Die folgende Tabelle 3.3 fasst die entwickelten Fragen und ihre theoretische Zuordnung zusammen.

Table 3.3.: Operationalisierung der Programmierstrategien (Programmanalyse)

Nr.	Frage zur Selbsteinschätzung	Zugehörige Strategie
A1	Ich lasse mich bei der Code-Navigation von meinem subjektiven Gefühl leiten, welche Informationen am wahrscheinlichsten zur Lösung führen.	Scent-Following
A2	Ich versuche Muster zu erkennen und stelle Verbindungen zwischen verschiedenen Codeabschnitten her.	Kognitive Modelle
A3	Ich nutze “Divide and Conquer Methoden”, um den Code zu verstehen.	Kognitive Modelle
A4	Ich interpretiere Compiler-Fehlermeldungen als direkte Anweisungen für meine nächsten Schritte.	Compiler als Assistent
A5	Ich nutze Compiler-Fehler, um mein mentales Modell über den Code zu korrigieren und zu präzisieren.	Compiler als Assistent
A6	Um Code zu verstehen oder einen Einstiegspunkt zu finden, interagiere ich zunächst mit der Benutzeroberfläche (UI) des Programms.	Interaktion mit UI

Fortsetzung auf der nächsten Seite

Tabelle 3.3 fortgesetzt

Nr.	Frage zur Selbsteinschätzung	Zugehörige Strategie
A7	Ich erkenne bekannte algorithmische Muster oder Entwurfsmuster und erfasse dadurch schnell die höhere semantische Bedeutung des Codes.	Wissensnutzung
A8	Mein abstraktes Wissen über Code und Design-Prinzipien kann ich leicht auf neue, mir unbekannte Problemstellungen übertragen.	Wissensnutzung
A9	Wenn ich bei einem Problem nicht weiterkomme, suche ich gezielt die Interaktion mit Kollegen oder Online-Communities.	SRL (Schwierigkeiten)
A10	Bei Blockaden in der Programmentwicklung bitte ich Lehrpersonen oder erfahrene Entwickler aktiv um Hilfe.	SRL (Schwierigkeiten)

Die Items **A1 bis A10** basieren auf kognitiven Theorien des Programmverständnisses sowie auf empirisch beobachteten Verhaltensmustern professioneller Entwickler.

Programmverständnis & Scent-Following (A1): Da eine vollständige Code-Analyse oft zu ressourcenintensiv ist, navigieren Entwickler häufig entlang von Hinweisen wie Variablennamen oder Kommentaren. Dieser als "Information Scent" bezeichnete Prozess wird in Item **A1** adressiert. Es erfasst, inwieweit Probanden bei der Navigation eher einer intuitiven Wahrnehmung von Hinweisen folgen, anstatt explizite Hypothesen zu bilden [33, 9].

Kognitive Modelle & Abstraktion (A2–A3): Die Bildung mentaler Repräsentationen erfordert sowohl Bottom-Up- als auch Top-Down-Prozesse. Item **A2** operationalisiert die Fähigkeit zur Mustererkennung und zum Herstellen von Verbindungen, was Experten erlaubt, Informationen effizient zu kategorisieren [26]. Item **A3** zielt auf die Anwendung von Abstraktionsstrategien wie "Divide and Conque"“ ab, die insbesondere leistungsstarken Entwicklern helfen, komplexe Strukturen systematisch zu durchdringen [36].

Compiler als Assistent (A4–A5): Der Compiler wird in der professionellen Praxis über die reine Fehlermeldung hinaus als interaktives Werkzeug genutzt. Item **A4** erhebt die Nutzung als direktives Werkzeug (dynamische To-Do-Liste), bei der Fehlermeldungen als Arbeitsanweisungen verstanden werden [24]. Item **A5**

prüft die korrektive Funktion, bei der Fehlermeldungen gezielt provoziert werden, um das eigene mentale Modell über Typbeziehungen und Code-Abhängigkeiten zu validieren [33, 24].

Interaktion mit der UI & Startpunkten (A6): Um Einstiegspunkte in unbekanntem Code zu finden, nutzen Entwickler häufig die Perspektive des Endanwenders. Item **A6** operationalisiert diese Strategie, bei der die Interaktion mit der grafischen Benutzeroberfläche dazu dient, die auslösenden Methoden im Quellcode zu identifizieren und irrelevante Logikbereiche auszufiltern [33].

Wissensnutzung & Mustererkennung (A7–A8): Experten greifen auf tiefgehende Schemata zurück, um semantische Konzepte zu erschließen. Item **A7** erfasst die Fähigkeit, algorithmische Muster intuitiv zu erkennen und dadurch die Semantik des Programms auf einer hohen Abstraktionsebene zu erfassen [27]. Item **A8** adressiert den Transfer von abstraktem Wissen auf neue, unbekannte Problemstellungen als zentrales Merkmal fortgeschrittener Expertise [5].

SRL: Auflösung von Schwierigkeiten (A9–A10): Das selbstregulierte Lernen (SRL) umfasst auch die soziale Unterstützung bei Blockaden (Seeking Social Assistance). Item **A9** operationalisiert die gezielte Interaktion mit Online-Communities als externe Ressource zur Problemlösung [29]. Item **A10** erhebt das aktive Hilfesuchen bei Lehrpersonen oder erfahrenen Kollegen, was als gängige Praxis beim Übergang in professionelle Arbeitsumgebungen gilt [29].

3.1.4. Debugging: Herleitung der Items aus den Strategien

Das Debugging stellt eine der komplexesten kognitiven Anforderungen in der Softwareentwicklung dar. In diesem Abschnitt werden die Strategien zur Fehleridentifikation und -behebung operationalisiert, um zwischen systematischen Expertenvorgehen und eher ungeplanten Novizenansätzen zu differenzieren.

Debugging: Übersicht der Fragen Die folgende Tabelle 3.4 fasst die operationalisierten Items für den Bereich Debugging zusammen.

Table 3.4.: Operationalisierung der Programmierstrategien (Debugging)

Nr.	Frage zur Selbsteinschätzung	Zugehörige Strategie
B1	Mein Debugging-Prozess gliedert sich klar in die Phasen Fehlerlokalisierung, Diagnose der Ursache und Behebung.	Strategischer Dreiklang
B2	Wenn ich mit einem Fehler konfrontiert werde, stelle ich zuerst eine Hypothese auf, teste diese und verwerfe oder modifiziere sie, falls sie fehlschlägt.	Strategischer Dreiklang
B3	Ich nutze print-Statements oder Logging, um den Programmfluss und den Zustand der Variablen zu beobachten.	Manuelle Strategien
B4	Ich nutze Breakpoints und einen Debugger, um den Programmzustand zu bestimmten Zeitpunkten zu untersuchen.	Manuelle Strategien
B5	Ich verfolge die Programmausführung schrittweise (Tracing), um den Kontroll- und Datenfluss systematisch nachzuvollziehen.	Code Tracing
B6	Ich visualisiere den Speicher (z.B. mit einer Speichertabelle), um nachzuvollziehen, wie sich Variablenwerte während der Ausführung ändern.	Code Tracing

Fortsetzung auf der nächsten Seite

Tabelle 3.4 fortgesetzt

Nr.	Frage zur Selbsteinschätzung	Zugehörige Strategie
B7	Ich arbeite bei einem Fehler systematisch rückwärts von der fehlerhaften Ausgabe zur ursprünglichen Quelle des falschen Werts.	Fehlerlokalisierung
B8	Ich sammle alle möglichen Fehlerursachen und überprüfe diese anschließend systematisch.	Fehlerlokalisierung
B9	Ich nehme bewusst zufällige Code-Änderungen vor und teste sofort, um zu sehen, ob der Fehler verschwindet.	Vemeidung (T&E)
B10	Ich nutze die Programmausgabe, um direkte Fragen nach dem Zustand des Programms zu stellen (z.B. "Warum ist dieser Wert entstanden?").	Output-Zentriert
B11	Ich lerne aus Fehlern, um diese in zukünftigen Projekten zu vermeiden.	Output-Zentriert
B12	Ich nutze Techniken wie zum Beispiel Program Slicing, um die Menge an zu untersuchendem Code auf die fehlerrelevanten Teile zu reduzieren.	Slicing-Techniken
B13	Ich nutze Tools zur Fehlerlokalisierung, die mir eine Rangliste der wahrscheinlichsten Fehlerstellen liefern.	Automatisierte FL
B14	Ich vertraue auf automatisierte Verfahren, die Code-Änderungen als verdächtig einstufen, um den Fehler schneller zu finden.	Automatisierte FL

Die Items **B1 bis B14** decken das Spektrum von manuellen Kontrollflussanalysen bis hin zu hochgradig automatisierten Verfahren der Fehlerlokalisierung ab.

Strategischer Dreiklang (B1–B2): Professionelles Debugging zeichnet sich durch eine klare Phasentrennung aus. Item **B1** operationalisiert den Workflow aus Lokalisierung, Diagnose und Behebung, der in der Forschung als Kennzeichen für Experteneffizienz gilt [9]. Item **B2** zielt auf die systematische Hypothesenbildung ab, da Experten im Gegensatz zu Novizen komplexe Kausalanalysen durchführen, anstatt lediglich Symptome auf Zeilenebene zu betrachten [9].

Traditionelle und Manuelle Strategien (B3–B4): Grundlegende Techniken zur Beobachtung des Programmzustands bilden das Fundament des Debuggings. Item **B3** erfasst die Nutzung von Logging und print-Statements zur Laufzeitbeobachtung [39]. Item **B4** adressiert den Einsatz von Breakpoints und Debuggern, um die Ausführung gezielt anzuhalten und Variablenwerte zu verifizieren [39, 28].

Code Tracing / Following Execution (B5–B6): Um die kognitive Belastung beim Verfolgen des Datenflusses zu reduzieren, nutzen erfolgreiche Entwickler externe Hilfsmittel. Item **B5** operationalisiert das schrittweise Verfolgen der Ausführung (Tracing), um den Kontrollfluss präzise zu emulieren [23]. Item **B6** zielt auf die Visualisierung des Speichers ab, was besonders Novizen dabei unterstützt, den Gültigkeitsbereich von Variablen korrekt abzubilden [40].

Systematische Fehlerlokalisierung (B7–B8): Das methodische Rückwärtsarbeiten von einer fehlerhaften Ausgabe zur Ursache ist eine explizite Expertenstrategie. Item **B7** erfasst dieses systematische Verfolgen der Kausalitätskette [22]. Item **B8** operationalisiert das Sammeln und systematische Prüfen aller potenziellen Ursachen, um das ineffiziente Raten von Fehlern zu vermeiden [22].

Vermeidung von Trial and Error (B9): Unsystematische Vorgehensweisen korrelieren negativ mit dem Lernerfolg. Item **B9** dient als Indikator für die "Trial and Error"-Strategie, bei der häufige, aber zufällige Code-Änderungen ohne fundierte Hypothese vorgenommen werden [36, 7].

Output-Zentrierte Strategien (B10–B11): Das interrogative Debugging nutzt spezifische Fragen an das Programmverhalten. Item **B10** operationalisiert das Stellen von "Why"-Fragen zur Programmausgabe, um direkt zu den verantwortlichen Anweisungen geführt zu werden [41, 28]. Item **B11** zielt auf die reflexive Komponente ab, bei der Fehler als Lerngelegenheit für zukünftige Projekte begriffen werden [3].

Slicing-Techniken (B12): Zur Reduktion der Suchdomäne in großen Systemen werden Abhängigkeitsanalysen eingesetzt. Item **B12** erfasst die Nutzung von Program Slicing, um gezielt nur jene Programmteile zu untersuchen, die ein bestimmtes fehlerhaftes Kriterium beeinflussen [13, 1].

Automatisierte Fehlerlokalisierung (B13–B14): Moderne Werkzeuge unterstützen Entwickler durch statistische Analysen. Item **B13** adressiert die Nutzung von Ranglisten zur Priorisierung potenzieller Fehlerstellen [39]. Item **B14** prüft das Vertrauen in automatisierte Verfahren zur Identifikation verdächtigen Codes, was insbesondere bei komplexen Systemen als Expertenstrategie zur Effizienzsteigerung gilt [28].

3.2. Mapping Programmier- auf psychologische Problemlösungsstrategien

In diesem Abschnitt wird die theoretische Brücke zwischen den domänenspezifischen Programmierstrategien des Fragebogens und den allgemeinen psychologischen Problemlösungsstrategien geschlagen. Das Mapping ordnet die operationalisierten Items den kognitiven Mechanismen zu, die im Theorieteil (siehe Kapitel 2.2) definiert wurden. Diese Zuordnung ermöglicht eine tiefergehende Analyse des Programmierverhaltens auf Basis psychologischer Modelle.

Tabelle 3.5 fasst dieses Mapping zusammen, wobei die Items innerhalb der Kategorien nach den Phasen Design (D), Programmierung (P), Analyse (A) und Debugging (B) sortiert sind.

Table 3.5.: Mapping der Items auf psychologische Problemlösungsstrategien

Nr.	Strategie (Fragebogen)	Psychologische Strategie
D5, D6	Design Patterns & Muster Sprachen	**Schemagesteuertes Lösen**
P1	Explizite Programmierstrategien (High-Level)	**Schemagesteuertes Lösen**
P3, P4	Programmier-Plans (Sub-Algorithmisch)	**Schemagesteuertes Lösen**
A2	Kognitive Modelle & Abstraktion	**Schemagesteuertes Lösen**
A7	Wissensnutzung & Mustererkennung	**Schemagesteuertes Lösen**
D7, D8	Dekomposition / Zerlegung	**Problemzerlegung & Unterziele**
A3	Kognitive Modelle & Abstraktion (Divide and Conquer)	**Problemzerlegung & Unterziele**
B12	Slicing-basierte Techniken	**Problemzerlegung & Unterziele**
P7	Inkrementelles/Systematisches Vorgehen	**Arbeiten Vorwärts**
A4	Compiler als Assistent	**Arbeiten Vorwärts**
B5, B6	Code Tracing / Following Execution	**Arbeiten Vorwärts**
D9, D10	Typkonstruktion als Strategie	**Arbeiten Rückwärts**

Fortsetzung auf der nächsten Seite

Tabelle 3.5 fortgesetzt

Nr.	Strategie (Fragebogen)	Psychologische Strategie
A6	Interaktion mit der UI & Startpunkten	**Arbeiten Rückwärts**
B7	Systematische Fehlerlokalisierung (Explizit)	**Arbeiten Rückwärts**
B10	Output-Zentrierte Strategien (Whyline)	**Arbeiten Rückwärts**
D3, D4	Test-Driven Development (TDD)	**Generieren und Testen**
P9	Exploratives Programmieren	**Generieren und Testen**
P11	LLM-unterstützte Codegenerierung	**Generieren und Testen**
B2	Strategischer Dreiklang	**Generieren und Testen**
B3, B4	Traditionelle & Manuelle Strategien	**Generieren und Testen**
B8	Systematische Fehlerlokalisierung (Explizit)	**Generieren und Testen**
B9	Trial and Error & Guess & Check	**Generieren und Testen**
B13, B14	Automatisierte Fehlerlokalisierung (FL)	**Generieren und Testen**
P8	Code-Wiederverwendung	**Problemlösung durch Analogie**
A8	Wissensnutzung & Mustererkennung	**Problemlösung durch Analogie**
D2	Agile & Lean Methoden (Daily Meetings)	**Metakognitive und Affektive Strat.**
D13, D14	Analytisches Denken & Trade-offs	**Metakognitive und Affektive Strat.**
P2	Explizite Programmierstrategien (High-Level)	**Metakognitive und Affektive Strat.**
P5	Selbstreguliertes Lernen (SRL)	**Metakognitive und Affektive Strat.**
P10	Exploratives Programmieren (Notizen)	**Metakognitive und Affektive Strat.**
A1	Programmverständnis & Scent-Following	**Metakognitive und Affektive Strat.**
A5	Compiler als Assistent (Mentale Modellkorrektur)	**Metakognitive und Affektive Strat.**

Fortsetzung auf der nächsten Seite

Tabelle 3.5 fortgesetzt

Nr.	Strategie (Fragebogen)	Psychologische Strategie
A9, A10	SRL: Auflösung von Schwierigkeiten (SOA)	**Metakognitive und Affektive Strat.**
B11	Output-Zentrierte Strategien (Lernen aus Fehlern)	**Metakognitive und Affektive Strat.**
D1	Agile & Lean Methoden (Scrum)	**Planung/Planungsstrategie**
D11, D12	Skizzierung (Sketching)	**Planung/Planungsstrategie**
P6	Selbstreguliertes Lernen (SRL: Arbeitsplan)	**Planung/Planungsstrategie**
B1	Strategischer Dreiklang (Phasengliederung)	**Planung/Planungsstrategie**

3.3. Detaillierte Erläuterung der Item-Mappings

In diesem Abschnitt werden alle Items des Fragebogens detailliert erläutert und ihre Zuordnung zu den psychologischen Problemlösungsstrategien begründet.

Schemagesteuertes Lösen & Domänenspezifisches Wissen Diese Items erfassen die Aktivierung von im Langzeitgedächtnis gespeicherten Wissensclustern (Problem-Schemata), die Experten eine effiziente Lösung bekannter Problemtypen ermöglichen.

- **D5**: Design Patterns fungieren als hochgradig kondensiertes Expertenwissen. Die Nutzung dieser Muster entspricht der Aktivierung bewährter Architektur-Schemata für wiederkehrende Problemstellungen.
- **D6**: Das Identifizieren von Mustern in strukturierten Sammlungen beschreibt den Prozess des Wissensabrufs, bei dem eine konkrete Anforderung einem bekannten Lösungsschema zugeordnet wird.
- **P1**: Schrittweise Anleitungen stellen vordefinierte Handlungsschemata dar, die den kognitiven Aufwand reduzieren, indem sie einen bewährten Pfad zur Ergebnissicherung vorgeben.
- **P3**: Der *Initialisation Plan* ist ein fundamentales sub-algorithmisches Wissenscluster. Die systematische Zuweisung von Startwerten zeigt die Anwendung dieses Basisschemas zur Vermeidung von undefinierten Zuständen.
- **P4**: Der *Guarded Division Plan* ist ein spezialisiertes Schema zur Fehlervermeidung. Die aktive Prüfung von Spezialfällen basiert auf domänenspezifischem Wissen über potenzielle Laufzeitfehler.

- **A2**: Die Suche nach Mustern und Verbindungen im Code dient der Aktivierung mentaler Repräsentationen. Bekannte Code-Strukturen fungieren hier als *Beacons* für vorhandene Schemata.
- **A7**: Das Erkennen algorithmischer Muster ermöglicht eine schnelle semantische Erfassung des Codes, da die höhere Bedeutung direkt aus dem aktivierten Expertenwissen abgeleitet wird, anstatt jede Zeile einzeln zu analysieren.

Problemzerlegung & Unterzielen Diese Kategorie umfasst Strategien, die darauf abzielen, einen komplexen Problemraum durch die Bildung handhabbarer Teilprobleme zu reduzieren.

- **D7**: Die aktive Aufteilung in kleinstmögliche, unabhängige Unterprobleme ist die direkte Umsetzung der Strategie, um die kognitive Last der Gesamtkomplexität zu bewältigen.
- **D8**: Eine funktionale Zerlegung zu Beginn des Entwurfsprozesses definiert die notwendigen Unterziele, die erreicht werden müssen, um die Gesamtstruktur der Lösung zu realisieren.
- **A3**: *Divide and Conquer* Methoden beim Code-Verständnis nutzen das Prinzip der Zerlegung, um komplexe Logik in verständliche Teilbereiche aufzubrechen.
- **B12**: *Program Slicing* ist eine spezialisierte Form der Zerlegung im Debugging, bei der der Code in fehlerrelevante und irrelevante Teile getrennt wird, um das Problem einzugrenzen.

Arbeiten Vorwärts (Vorwärtssuche) Items dieser Kategorie beschreiben einen datengesteuerten Prozess, der von den gegebenen Informationen ausgehend schrittweise zum Zielzustand führt.

- **P7**: Die Implementierung größerer Blöcke vor der Ausführung entspricht einem Vorwärtsarbeiten vom Anfangszustand zum Ziel, wobei die Operatoren (Code-Zeilen) ohne unmittelbare Rückkopplung angewendet werden.
- **A4**: Die Nutzung von Compiler-Fehlermeldungen als direkte Arbeitsanweisung stellt eine Vorwärtssuche dar, bei der das Tool den jeweils nächsten notwendigen Operator vorgibt, um den Code in den Zielzustand zu überführen.
- **B5**: Das schrittweise Verfolgen des Kontrollflusses (*Tracing*) emuliert die Programmausführung vom Startpunkt aus und folgt somit der logischen Vorwärtsrichtung des Programms.
- **B6**: Die Visualisierung der Variablenänderungen im Zeitverlauf bildet den Datenfluss in Vorwärtsrichtung ab, um die Transformation der Werte nachzuvollziehen.

Arbeiten Rückwärts (Rückwärtssuche) Diese Strategien starten beim gewünschten Zielzustand oder einer fehlerhaften Ausgabe und arbeiten sich zur ursprünglichen Quelle oder den Prämissen vor.

- **D9**: Der Entwurf beginnt hier mit der Zielstruktur (Typen/Signatur). Die Implementierung des Funktionskörpers erfolgt erst danach, was ein Rückwärtsarbeiten vom Ergebnis zur Logik darstellt.
- **D10**: Die Top-Down-Klärung der logischen Struktur mittels Datentypen nutzt den Zielzustand des Datenmodells als Ausgangspunkt für die weitere Problemlösung.
- **A6**: Die Interaktion mit der Benutzeroberfläche nutzt die sichtbare Ausgabe (Ziel), um rückwärts den auslösenden Code-Einstiegspunkt zu identifizieren.
- **B7**: Die systematische Rückverfolgung von einer fehlerhaften Ausgabe zur ursprünglichen Fehlerquelle ist die klassische Anwendung der Rückwärtssuche im Debugging.
- **B10**: Die Formulierung von "Warum"-Fragen ausgehend von einem entstandenen Wert nutzt das fehlerhafte Ergebnis als Startpunkt für eine rückwärtsgerichtete Kausalitätsanalyse.

Generieren und Testen In dieser Kategorie werden Lösungen oder Hypothesen erstellt und anschließend systematisch auf ihre Korrektheit überprüft.

- **D3**: Die Definition von Tests vor der Implementierung (TDD) etabliert ein Prüfverfahren, gegen das die später generierte Lösung unmittelbar getestet werden kann.
- **D4**: Das Prinzip, neue Funktionalität erst bei erfolgreichen Regressionstests zu implementieren, stellt sicher, dass jede generierte Änderung die Korrektheit des Gesamtsystems wahrt.
- **P9**: Experimenteller Code fungiert als generierte Hypothese über einen Lösungsweg, deren Tauglichkeit durch das anschließende Testen verifiziert wird.
- **P11**: Die Nutzung von LLM-Tools generiert Lösungsvorschläge, die aufgrund der potenziellen Fehleranfälligkeit der KI zwingend einer anschließenden Prüfung unterzogen werden müssen.
- **B2**: Das Aufstellen und Modifizieren von Hypothesen bei Fehlern ist der Kern des "Generieren und Testen"-Verfahrens im Debugging.
- **B3**: Print-Statements und Logging dienen als Werkzeuge zum Testen der Erwartungswerte gegenüber dem tatsächlich generierten Programmzustand.

- **B4**: Breakpoints ermöglichen die gezielte Verifikation (Testen) von Hypothesen über den Zustand an kritischen Ausführungspunkten.
- **B8**: Das Sammeln potenzieller Ursachen ist ein Prozess der Hypothesengenerierung, der durch systematisches Ausschlussverfahren (Testen) abgeschlossen wird.
- **B9**: Zufällige Änderungen ohne fundierte Basis entsprechen einem ungerichteten Generieren von Lösungen mit sofortigem Testen (Trial and Error).
- **B13**: Automatisierte Fehlerlokalisierung generiert eine Rangliste von Hypothesen über Fehlerstellen, die der Entwickler nacheinander prüfen muss.
- **B14**: Die automatische Einstufung von Code-Änderungen als verdächtig dient der Generierung eines Fokusbereichs für die anschließende manuelle Überprüfung.

Problemlösung durch Analogie Diese Items messen die Fähigkeit, Wissen von bekannten, analogen Problemstellungen auf neue Situationen zu übertragen.

- **P8**: Die Suche nach externen Code-Beispielen nutzt die Analogie zu bereits gelösten Problemen, um deren Struktur als Vorlage für die eigene Aufgabe zu adaptieren.
- **A8**: Der Transfer von abstraktem Wissen auf unbekannte Probleme basiert auf der Erkennung tieferliegender struktureller Analogien zwischen verschiedenen Design-Prinzipien.

Metakognitive und Affektive Strategien Diese Kategorie umfasst die Selbstregulation, die Überwachung des eigenen Verständnisses sowie die Bewertung des Fortschritts während des Problemlösens.

- **D2**: Abstimmungsrunden und Daily Meetings dienen der sozialen Unterstützung und der externen Überwachung des Fortschritts zur Selbstregulierung im Team.
- **D13**: Das analytische Abwägen von Trade-offs zwischen Performance und Wartbarkeit ist ein metakognitiver Prozess der Zielklärung und Bewertung.
- **D14**: Die logische Analyse von Vor- und Nachteilen spiegelt eine bewusste Überwachung der eigenen Entwurfsentscheidungen wider.
- **P2**: Die Ausrichtung der Dokumentation auf die Verständlichkeit für andere erfordert eine metakognitive Reflexion über die Klarheit der eigenen Lösung.

- **P5**: Die aktive Überwachung des Fortschritts und die Reflexion über Ziele sind Kernprozesse des selbstregulierten Lernens (SRL).
- **P10**: Temporäre Notizen im Code dienen als kognitive Entlastung und zur Strukturierung der eigenen Gedanken während des Prozesses.
- **A1**: *Scent-Following* basiert auf dem subjektiven Gefühl für Erfolgswahrscheinlichkeiten und steuert als metakognitiver Prozess die Informationssuche.
- **A5**: Die Nutzung von Fehlern zur Präzisierung des mentalen Modells ist eine Form der Selbstreflexion und Überwachung des eigenen Verständnisses.
- **A9**: Die gezielte Interaktion mit Communities zur Problemlösung ist eine Form der sozialen Hilfeleistung (*Seeking Social Assistance*) zur Überwindung von Blockaden.
- **A10**: Das aktive Bitten um Hilfe bei Lehrpersonen ist ein wesentlicher Bestandteil der Selbstregulation in Lernumgebungen.
- **B11**: Die reflexive Verwertung von Fehlern für zukünftige Projekte baut metakognitives Wissen über eigene Fehlermuster auf.

Planung / Planungsstrategie Die Planungsstrategie vereinfacht den Problemraum durch Abstraktion und legt die Reihenfolge der Bearbeitung fest.

- **D1**: Die Nutzung agiler Prinzipien dient der übergeordneten Planung und Steuerung des Gesamtprozesses in iterativen Zyklen.
- **D11**: Das Erstellen eines Skeletts mittels "Programmlöchern" ist ein Planungsprozess, der von Details abstrahiert, um zunächst die Gesamtstruktur festzulegen.
- **D12**: Die Priorisierung der schwierigsten Teile (Risikominimierung) ist ein strategisches Element der Planung, um Unsicherheiten frühzeitig zu adressieren.
- **P6**: Die Entwicklung eines expliziten Arbeitsplans vor der Implementierung ist die klassische Umsetzung einer Planungsstrategie zur Strukturierung der Arbeitsschritte.
- **B1**: Die explizite Gliederung des Debugging-Prozesses in Phasen zeigt eine strategische Planung der Abfolge von Lokalisierung, Diagnose und Behebung.

3.3.1. Hinweis zur Mehrfachzuordnung

Es ist anzumerken, dass die Abgrenzung der beschriebenen Programmierstrategien in der praktischen Anwendung nicht immer vollkommen trennscharf verläuft. Viele der operationalisierten Items weisen funktionale Überschneidungen auf, die theoretisch eine Zuordnung zu mehreren psychologischen Problemlösungsstrategien rechtfertigen würden. Um jedoch eine eindeutige statistische Auswertung der Häufigkeiten und eine konsistente Strukturierung der Methodik zu gewährleisten, wurde für jedes Item eine bewusste Festlegung auf eine primäre psychologische Kategorie vorgenommen.

Beispiele für diese potenziellen Mehrfachzuordnungen sind:

- **Test-Driven Development (D3, D4)**: Diese Praktik wurde primär der Strategie "Generieren und Testen" zugeordnet, da sie auf dem iterativen Zyklus von Implementierung und Verifikation basiert. Sie könnte jedoch gleichermaßen als "Planungsstrategie" gewertet werden, da die Definition der Tests eine abstrakte Zielsetzung vor der eigentlichen Codegenerierung erzwingt.

- **Program Slicing (B12)**: Dieses Item ist als "Problemzerlegung" klassifiziert, da es die Menge des zu untersuchenden Codes auf fehlerrelevante Teile reduziert. Gleichzeitig erfüllt es Merkmale einer "Metakognitiven Strategie", da es der bewussten Steuerung der Aufmerksamkeit und der Reduktion der kognitiven Belastung während der Fehlersuche dient.

3.4. Forschungsfragen

Basierend auf dem erstellten Fragebogen aus Kapitel 3.1 und dem Mapping aus Kapitel 3.2 ergeben sich folgende Forschungsfragen:

1. In welchem Ausmaß und mit welcher Verteilung wenden Programmierende die im Fragebogen definierten Programmierstrategien in ihrem Entwicklungsprozess an?

2. In welchem Ausmaß und mit welcher Verteilung kommen die generischen, psychologischen Problemlösungsstrategien im Vorgehen der Programmierenden zum Einsatz?

3. Inwiefern beeinflusst der Grad der Programmiererfahrung die Präferenz für spezifische Programmierstrategien?

4. Inwieweit lässt sich das theoretische Mapping von domänenspezifischen Programmierstrategien auf generische, psychologische Problemlösungsstrategien empirisch bestätigen?

3.5. Studienteilnehmer

Die Auswahl der Teilnehmer erfolgte im Rahmen einer Gelegenheitsstichprobe, ergänzt durch eine gezielte Ansprache innerhalb relevanter Fachgruppen. Um eine hohe Varianz hinsichtlich der Programmiererfahrung und der fachlichen Expertise sicherzustellen, wurden Probanden über zwei primäre Kanäle rekrutiert:

- Direktansprache im beruflichen Umfeld und Netzwerk: Potenzielle Teilnehmer aus der Softwareentwicklung wurden direkt im beruflichen Kontext sowie über das persönliche Netzwerk des Verfassers angesprochen. Dies ermöglichte insbesondere den Zugang zu erfahrenen Softwareentwicklern mit mehrjähriger Praxiserfahrung.

- Akademisches Umfeld: Ein signifikanter Teil der Rekrutierung erfolgte über die Verteilung des Fragebogens in Lehrveranstaltungen der betreuenden Professur. Hierdurch konnten gezielt Studierende der Informatik (oder verwandter Fachrichtungen) gewonnen werden, die unterschiedliche Ausbildungsstände repräsentieren.

Zentrales Kriterium für die Teilnahme war eine aktive Tätigkeit oder ein Studium im Bereich der Informatik. Die Stichprobe wurde bewusst heterogen gestaltet, um die Forschungsfrage über verschiedene Kompetenzstufen hinweg untersuchen zu können. Das Spektrum der Probanden reicht somit von Programmieranfängern (Studierende in frühen Semestern) über fortgeschrittene Studierende bis hin zu Senior-Entwicklern mit langjähriger Berufspraxis. Insgesamt nahmen 39 Personen an der Befragung teil. Die Teilnahme erfolgte auf freiwilliger Basis. Die Anonymität der Datenverarbeitung wurde zu jedem Zeitpunkt zugesichert. Eine detaillierte Aufschlüsselung der demografischen Merkmale und des jeweiligen Skill-Levels findet sich im Abschnitt Ergebnisse.

3.6. Durchführung

Die Datenerhebung fand im Zeitraum vom 13. November bis zum 04. Dezember 2025 statt. Als technische Plattform wurde die Umfragesoftware LimeSurvey genutzt, welche auf den internen Servern der Technischen Universität gehostet wurde. Dies stellte sicher, dass die erhobenen Daten den geltenden Datenschutzbestimmungen (DSGVO) entsprachen und lokal gespeichert wurden. Alle Limesurvey Dateien befinden sich im Anhang B.

Der Ablauf der Befragung gliederte sich in vier wesentliche Phasen:

Onboarding und Datenschutz Nach dem Aufrufen des Umfragelinks landeten die Teilnehmer auf einer Begrüßungsseite (Abbildung 3.1). Hier konnten sie zwischen einer deutschen und einer englischen Sprachversion wählen. Vor Beginn der eigentlichen Befragung wurden die Probanden umfassend über das Forschungsziel, die Freiwilligkeit der Teilnahme sowie die anonymisierte Verarbeitung ihrer Daten aufgeklärt. Die explizite Zustimmung zur Datenschutzerklärung war Voraussetzung für den Start der Umfrage.

Abbildung 3.1.: Fragebogen Seite 1 (Einleitung)

Erhebung demografischer Daten Zur Charakterisierung der Stichprobe sowie zur späteren Differenzierung der Ergebnisse nach Erfahrungsstufen wurde ein Block soziodemografischer Fragen sowie Fragen zum fachlichen Hintergrund vorangestellt. Sämtliche Fragen in diesem Abschnitt wurden als Pflichtfragen deklariert, um eine vollständige Datenbasis für die anschließende Korrelationsanalyse sicherzustellen.

Die Auswahl und Formulierung der Fragen erfolgte in enger Abstimmung mit der betreuenden Professur. Es wurden gezielt standardisierte Items verwendet, die bereits in vorangegangenen Studien des Lehrstuhls zum Einsatz kamen. Dieses Vorgehen gewährleistet eine methodische Konsistenz und ermöglicht eine Vergleichbarkeit der Ergebnisse mit zukünftigen oder bereits bestehenden Datensätzen der Professur (Benchmarking). Abbildung 3.2 zeigt die grafische

Darstellung der Frage PE1 um Fragebogen.

Bitte beantworten Sie folgende Fragen zu Ihrer Programmiererfahrung.

*Seit wie vielen Jahren programmieren Sie bereits?

Bitte wählen Sie eine der folgenden Antworten:

- <1 Jahr
- 1-5 Jahre
- 5-10 Jahre
- 10-15 Jahre
- >15 Jahre

Abbildung 3.2.: Fragebogen Seite 2 (Programmiererfahrung Frage 1)

Der Fragenblock umfasst vier zentrale Dimensionen der Programmierbiografie:

Table 3.6.: Übersicht der standardisierten Fragen zur Programmiererfahrung

Nr.	Frage	Antwortmöglichkeiten
PE1	Seit wie vielen Jahren programmieren Sie bereits?	< 1 Jahr, 1–5 Jahre, 5–10 Jahre, 10–15 Jahre, > 15 Jahre
PE2	Wie hoch schätzen Sie Ihre eigene Programmiererfahrung im Verhältnis zu Ihren Mitstudierenden oder Kollegen ein?	sehr unerfahren, etwas weniger erfahren, etwa ähnlich erfahren, etwas erfahrener, deutlich erfahrener
PE3	Wie häufig programmieren Sie?	einmal im Monat oder seltener, mehrmals im Monat, einmal pro Woche, mehrmals in der Woche, täglich
PE4	An wie vielen verschiedenen Projekten waren Sie bereits programmiertätig beteiligt?	weniger als 4, zwischen 4 und 7, mehr als 7

Durch diese differenzierte Abfrage wird sichergestellt, dass das "Skill-Level" der Teilnehmer nicht allein an der Dauer (Jahre), sondern auch an der Intensität (Frequenz) und der Vielfalt (Projekte) der praktischen Anwendung gemessen wird.

Fachspezifischer Fragebogen (Kernbefragung) Die eigentliche Untersuchung der Arbeitsprozesse war in vier thematische Abschnitte unterteilt, die jeweils auf einer eigenen Seite präsentiert wurden:

1. Designprozess
2. Programmierprozess
3. Programmanalyse
4. Debugging

Um neben den quantitativen Daten auch qualitative Einblicke zu gewinnen, wurde jeder dieser vier Seiten ein offenes Textfeld für Feedback hinzugefügt. Dies ermöglichte es den Teilnehmern, ihre Antworten zu präzisieren oder Besonderheiten ihres individuellen Workflows zu erläutern. In Abbildung 3.3 ist die Seite des Programmierprozesses zu sehen.

Programmierung

Im folgenden Teil bewerten Sie Ihr Vorgehen beim Programmieren. Bitte schätzen Sie für jede Aussage ein, ob diese auf Sie zutrifft.

	Trifft überhaupt nicht auf mich zu	Trifft eher nicht auf mich zu	Teils/Teils	Trifft eher auf mich zu	Trifft voll und ganz auf mich zu	keine Antwort
Ich nutze detaillierte, schrittweise Anleitungen, um konsistente Ergebnisse zu erzielen.	○	○	○	○	○	○
Ich dokumentiere meine Programme so, dass sie von anderen klar verstanden und ausgeführt werden können.	○	○	○	○	○	○
Ich weise allen Variablen einen Startwert zu, bevor ich sie in Berechnungen verwende.	○	○	○	○	○	○
Bei Berechnungen achte ich aktiv auf Spezialfälle, die zu Fehlern führen könnten (z.B. die Division durch Null).	○	○	○	○	○	○
Ich überwache meinen Arbeitsfortschritt aktiv und reflektiere meine Ziele, um meine nächsten Schritte zu planen.	○	○	○	○	○	○
Ich organisiere meine Arbeitsschritte und entwickle einen Arbeitsplan, bevor ich mit der eigentlichen Implementierung beginne.	○	○	○	○	○	○
Ich programmiere größere Code-Blöcke auf einmal, bevor ich den Code ausführe.	○	○	○	○	○	○
Bei der Gestaltung eines Programms suche ich nach einem existierenden Code-Beispiel in externen Quellen, um es als Vorlage anzupassen.	○	○	○	○	○	○
Ich schreibe bewusst experimentellen Code, um Erkenntnisse zu gewinnen, die ich später zur Verfeinerung der endgültigen Lösung nutze.	○	○	○	○	○	○
Während des Programmierens schreibe ich mir temporäre Notizen direkt in den Code, um meine Gedanken zu strukturieren, bevor ich die endgültige Implementierung vornehme.	○	○	○	○	○	○
Ich nutze LLM-Tools (z.B. ChatGPT) aktiv zur Generierung von Lösungsvorschlägen für Programmieraufgaben.	○	○	○	○	○	○

Haben Sie Gedanken, Anekdoten oder sonstige Anmerkungen zu den genannten Strategien? (z.B. unter welchen Umständen Sie sie gerne verwenden/nicht gerne verwenden)

Abbildung 3.3.: Fragebogen Seite 4 (Strategien im Programmierprozess)

Jede Frage konnte mit einer der folgenden Optionen beantwortet werden:

- Trifft überhaupt nicht auf mich zu (Wertung 1)
- Trifft eher nicht auf mich zu (Wertung 2)
- Teils/Teils (Wertung 3)
- Trifft eher auf mich zu (Wertung 4)
- Trifft voll und ganz auf mich zu (Wertung 5)
- keine Antwort (Wertung 0)

Für die Auswertung hat jede Antwortmöglichkeit eine Wertung bekommen. Diese entspricht dem Wert in Klammern.

Präsentation des Mappings und Validierung Nach Abschluss der Selbstauskunft folgte der interaktive Teil der Umfrage. Auf Basis der zuvor gegebenen Antworten erfolgte eine automatisierte Berechnung (Mapping), welche die Passung des Teilnehmers zu acht verschiedenen psychologischen Strategien ermittelte. Diese Strategien wurden den Probanden sukzessive (eine Strategie pro Seite) vorgestellt. Die Visualisierung erfolgte durch eine Prozentangabe in Kombination mit einem Balkendiagramm, welches veranschaulichte, wie stark die jeweilige Strategie auf den Teilnehmer zutrifft. Die Prozentzahl wurde aus dem Mittelwert der beantworteten Fragen gebildet. Wenn ein Teilnehmer beispielsweise die Fragen einer Kategorie alle mit "Teils/Teils" beantwortet hat, so zeigt die Prozentangabe 50% an.

Zur Validierung des Mappings wurden die Teilnehmer gebeten, die Korrektheit dieser algorithmischen Einschätzung auf einer dreistufigen Skala zu bewerten:

- Zu gering eingestuft
- Richtig eingestuft
- Zu hoch eingestuft

Zusätzlich bot ein weiteres Textfeld pro Strategie Raum für ergänzende Anmerkungen zur subjektiv empfundenen Passgenauigkeit. In Abbildung 3.4 ist die Feedback Seite des *Schemagesteuerten Lösen* zu sehen.

Abbildung 3.4.: Fragebogen Seite 7 (Feedback psychologische Strategie: Schemagesteuertes Lösen & Domänenspezifisches Wissen)

Abschluss der Befragung Die Umfrage endete mit einer finalen Seite (Abbildung 3.5), auf der die Teilnehmer die Möglichkeit hatten, allgemeines Feedback zur Studie, zum Aufbau des Fragebogens oder zum Design der Präsentation zu hinterlassen.

Feedback

Vielen Dank für Ihre Teilnahme. Ihre Antworten liefern uns wertvolle Einblicke dazu, welche Arten von Problemlösungsstrategien in der Softwareentwicklung angewendet werden. Wenn Sie Fragen zu der Umfrage oder zum Forschungsprojekt haben, kontaktieren Sie uns gerne.

Florian Grabs

E-Mail: florian.grabs@s2018.tu-chemnitz.de

Belinda Schantong

E-Mail: belinda.schantong@informatik.tu-chemnitz.de

Janet Siegmund

E-Mail: janet.siegmund@informatik.tu-chemnitz.de

Wenn Sie noch weitere Anmerkungen oder generelles Feedback zum Fragebogen haben, können Sie diese gerne hier eintragen:

Zurück

Absenden

Abbildung 3.5.: Fragebogen abschließendes Feedback

4. Ergebnisse

Die Rohdaten der Studie, die Umfragedatei aus LimeSurvey, alle Diagramme, Tabellen, und Scripte finden sich im Anhang B. Die Ergebnisse der Online-Umfrage wurden mit Hilfe eines Python Scripts ausgewertet. 8 der 39 Teilnehmer haben die Umfrage nicht komplett ausgefüllt und wurden daher bei der Auswertung nicht berücksichtigt.

4.1. Beschreibung der Teilnehmer

Demographie 1 Zur Charakterisierung der Stichprobe wurde zunächst die Dauer der bisherigen Programmiertätigkeit erhoben. Die zeitliche Erfahrung der Teilnehmenden in Jahren, unterteilt in fünf Kategorien von weniger als einem Jahr bis über 15 Jahren, ist in Abbildung 4.1 dargestellt.

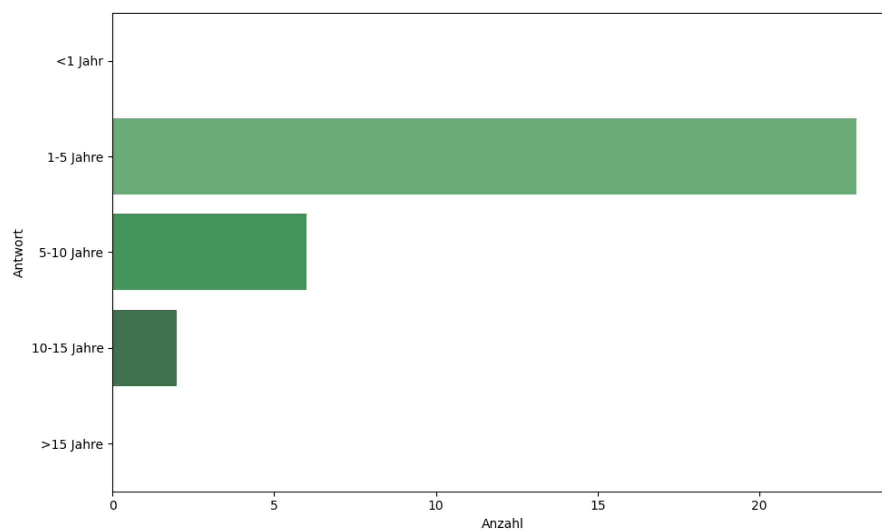

Abbildung 4.1.: Auswertung demografische Frage 1

Die Analyse der Programmiererfahrung verdeutlicht, dass die Stichprobe überwiegend aus Teilnehmenden mit einer moderaten Erfahrungshistorie besteht. Mit einem Anteil von 74,19 % gibt die deutliche Mehrheit der Befragten an, seit einem Zeitraum von eins bis fünf Jahren aktiv zu programmieren. Ein deutlich geringerer Teil verfügt über eine längerjährige Erfahrung von fünf bis zehn Jahren (19,35 %), während lediglich zwei Personen (6,45 %) eine Erfahrung von zehn bis 15

Jahren aufweisen. Teilnehmer mit einer Expertise von weniger als einem Jahr oder mehr als 15 Jahren waren in der Stichprobe nicht vertreten. Dies deutet darauf hin, dass die Untersuchung primär Personen im Stadium der fortgeschrittenen Ausbildung oder frühen Berufstätigkeit erfasst.

Demographie 2 Neben der reinen Zeitdauer spielt die subjektive Wahrnehmung der eigenen Expertise eine wesentliche Rolle für die Einordnung der Probanden. Abbildung 4.2 zeigt die Selbsteinschätzung der Teilnehmer bezüglich ihrer Programmiererfahrung im direkten Vergleich zu ihren Mitstudierenden oder Kollegen.

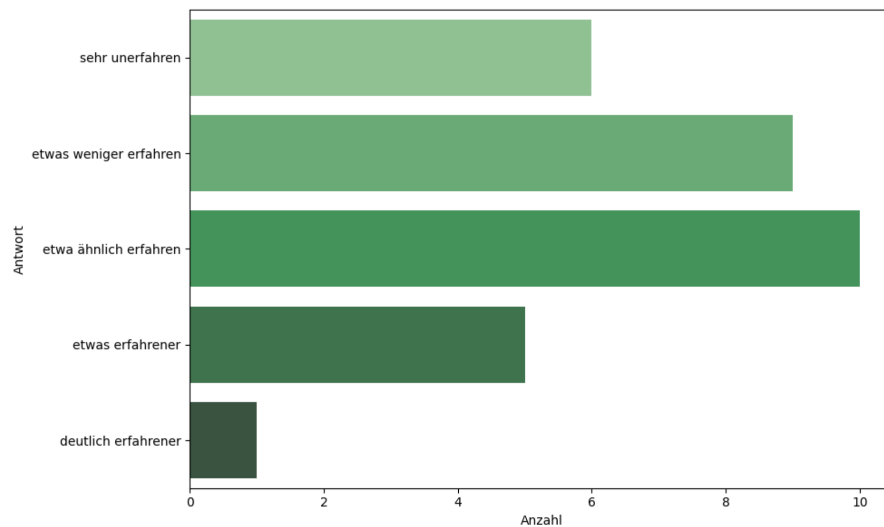

Abbildung 4.2.: Auswertung demografische Frage 2

Bezüglich der subjektiven Selbsteinschätzung der Programmierkompetenz im Vergleich zu Mitstudierenden oder Kollegen zeigt sich eine annähernde Normalverteilung mit einer leichten Tendenz zum unerfahreneren Spektrum. Der größte Anteil der Probanden (32,26 %) schätzt die eigene Erfahrung als "etwa ähnlich" zu der ihrer Mitsudierenden oder Kollegen ein. Gleichzeitig ordnet sich ein beachtlicher Teil der Stichprobe (insgesamt 48,38 %) als "sehr unerfahren" oder "etwas weniger erfahren" ein. Demgegenüber steht eine Minderheit von insgesamt 19,36 %, die sich als "etwas erfahrener" oder "deutlich erfahrener" (3,23 %) wahrnimmt. Diese Ergebnisse legen nahe, dass die Teilnehmenden ihre eigenen Fähigkeiten eher konservativ bewerten oder sich in einem kompetitiven Umfeld verorten.

Demographie 3 Um die Intensität der regelmäßigen Auseinandersetzung mit Programmieraufgaben zu erfassen, wurde zudem die Häufigkeit der praktischen

Tätigkeit abgefragt. Die Verteilung der Antwortmöglichkeiten, von seltener als einmal im Monat bis hin zu täglicher Praxis, ist Abbildung 4.3 zu entnehmen.

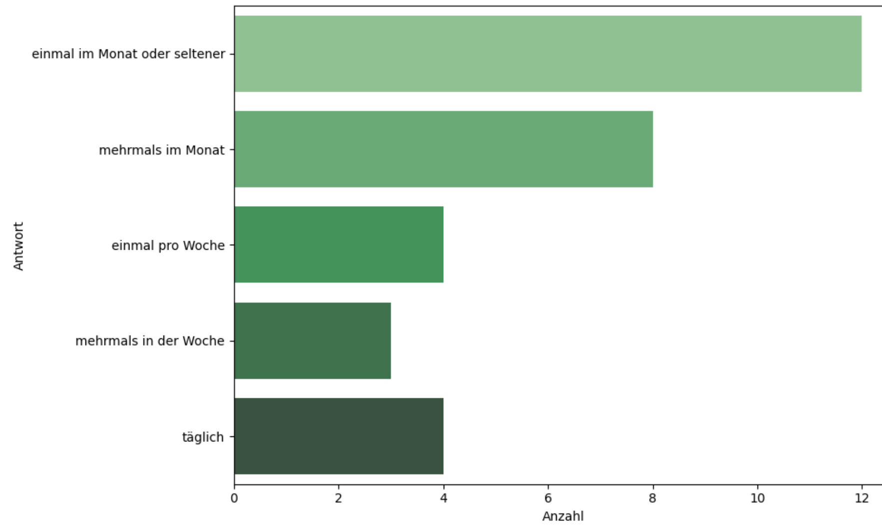

Abbildung 4.3.: Auswertung demografische Frage 3

Die Auswertung der Programmierhäufigkeit offenbart, dass das Programmieren für einen Großteil der Befragten keine tägliche Routine darstellt, sondern eher punktuell erfolgt. 38,71 % der Teilnehmenden geben an, lediglich einmal im Monat oder seltener zu programmieren, gefolgt von 25,81 %, die mehrmals monatlich tätig sind. Eine regelmäßige wöchentliche Praxis (einmal oder mehrmals pro Woche) wird von insgesamt 22,58 % der Probanden gepflegt. Lediglich 12,9 % der Stichprobe programmieren täglich. Diese Verteilung lässt darauf schließen, dass die Programmiertätigkeit bei der Mehrheit der Befragten wahrscheinlich an spezifische Ereignisse, wie etwa Studienprojekte oder Prüfungsphasen, gebunden ist und weniger als kontinuierliches Hobby oder Kernbestandteil der täglichen Arbeit ausgeübt wird.

Demographie 4 Ergänzend zur zeitlichen Dauer wurde die praktische Erfahrung durch die Anzahl der bisherigen Projektbeteiligungen im akademischen oder privaten Umfeld quantifiziert. Abbildung 4.4 illustriert, an wie vielen Projekten die befragten Personen zum Zeitpunkt der Erhebung bereits mitgewirkt hatten.

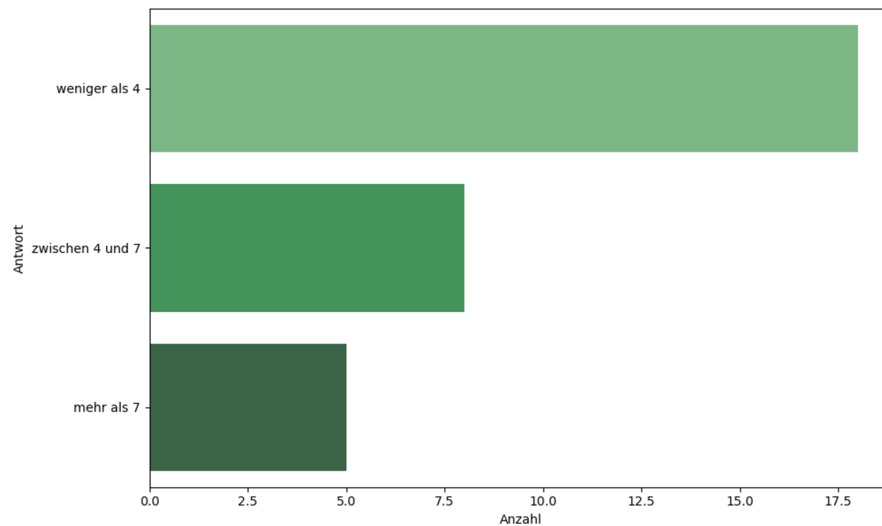

Abbildung 4.4.: Auswertung demografische Frage 4

In Bezug auf die praktische Projekterfahrung bestätigt sich das Bild einer eher am Anfang stehenden Expertise. Mehr als die Hälfte der Teilnehmenden (58,06 %) war bislang an weniger als vier Projekten beteiligt. Eine mittlere Projekterfahrung von vier bis sieben Projekten weisen 25,81 % der Befragten auf. Lediglich eine kleine Gruppe von 16,13 % blickt auf eine Beteiligung an mehr als sieben Projekten zurück. Diese Daten korrespondieren mit der Angabe der Programmierjahre und unterstreichen, dass die Stichprobe primär aus Personen besteht, die noch am Beginn ihrer professionellen oder akademischen Projektlaufbahn stehen.

4.2. Datenauswertung

4.2.1. Deskriptive Statistik der Fragen

Die Analyse der einzelnen Fragebogen-Items ($N = 31$) verdeutlicht die unterschiedliche Ausprägung der zugrundeliegenden Verhaltensweisen. Die Bewertung erfolgte auf einer Skala von 1 (nie) bis 5 (sehr oft) bzw. 0 (keine Antwort). In Abbildung 4.5 sind die jeweils zehn Items mit den höchsten und niedrigsten Mittelwerten dargestellt. Eine Tabelle mit den Werten aller Fragen befindet sich im Anhang in Tabelle A.1.

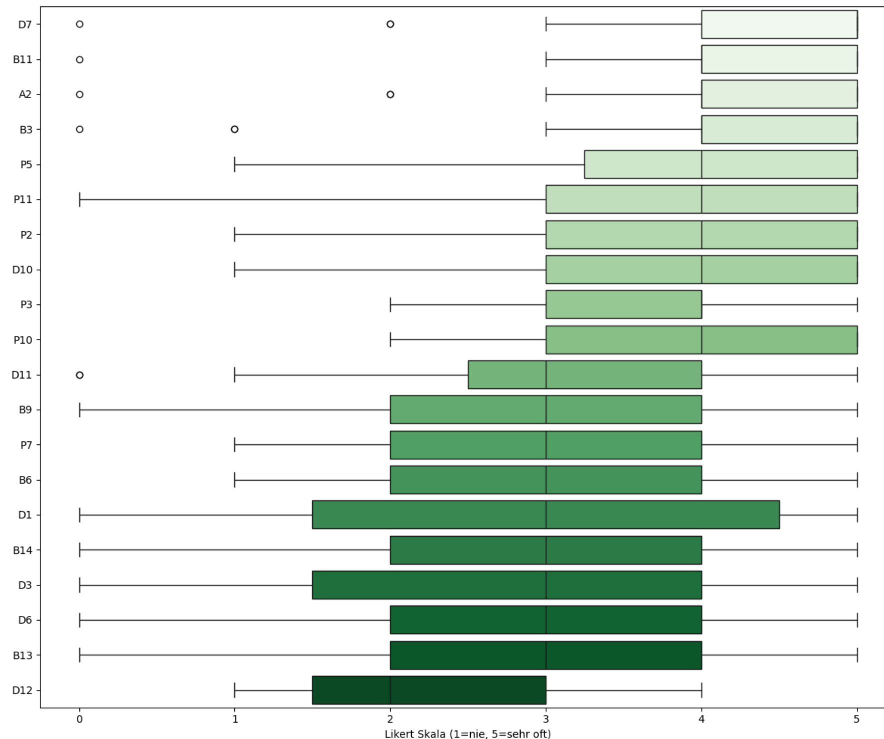

Abbildung 4.5.: Top 10 / Flop 10 Mittelwerte der beantworteten Fragen

Häufigste Strategie-Items (Top 10) Das am intensivsten genutzte Item der Erhebung ist *D7* mit einem Mittelwert von $M = 4,22$ ($SD = 1,15$), gefolgt von *B11* ($M = 4,13$; $SD = 1,09$). Ebenfalls hohe Mittelwerte im Bereich von $M \approx 4,0$ weisen die Items *A2* und *B3* (beide $M = 3,97$) sowie *P5* ($M = 3,97$; $SD = 1,22$) auf. Die Items *P11* und *P2* erreichen jeweils einen Mittelwert von $M = 3,94$. Die Top 10 werden durch *D10* ($M = 3,87$) sowie *P3* und *P10* (beide $M = 3,81$) vervollständigt. Der Median liegt für die Items der Top 10 einheitlich bei $4,0$ oder höher, wobei der Maximalwert von $5,0$ bei fast allen dieser Items erreicht wurde.

Seltenste Strategie-Items (Flop 10) Das am seltensten genutzte Item der Untersuchung ist *D12* mit einem Mittelwert von $M = 2,29$ ($SD = 1,01$). Mit deutlichem Abstand folgen die Items *B13* und *D6* (beide $M = 2,81$) sowie *D3* ($M = 2,84$; $SD = 1,61$). Ebenfalls im unteren Bereich der Skala befinden sich *B14* ($M = 2,90$), *D1* ($M = 2,94$) sowie *B6* und *P7* (beide $M = 2,97$). Die Items *B9* ($M = 3,06$) und *D11* ($M = 3,10$) markieren die oberen Werte der Flop-10-Liste. Die Mediane in dieser Gruppe liegen überwiegend bei $3,0$, mit Ausnahme von *D12*, dessen Median bei $2,0$ liegt.

Streuung und Varianz der Items Hinsichtlich der Standardabweichung zeigt sich bei den Flop-10-Items eine teils höhere Varianz als bei den Top-Items. Insbesondere die Items *D1* ($SD = 1,65$), *D3* ($SD = 1,61$) und *D6* ($SD = 1,58$) weisen eine starke Streuung auf, was auf ein sehr heterogenes Nutzungsverhalten der Probanden hindeutet. Die geringste Varianz innerhalb der betrachteten Items zeigt sich bei *P3* ($SD = 0,83$) und *P2* ($SD = 0,93$), wo die Antworten der Teilnehmenden vergleichsweise nah um den Mittelwert konzentriert sind.

4.2.2. Deskriptive Statistik der Programmierstrategien

Basierend auf den vorliegenden deskriptiven Statistiken (N = 31) lassen sich die Ergebnisse der Untersuchung wie folgt zusammenfassen.

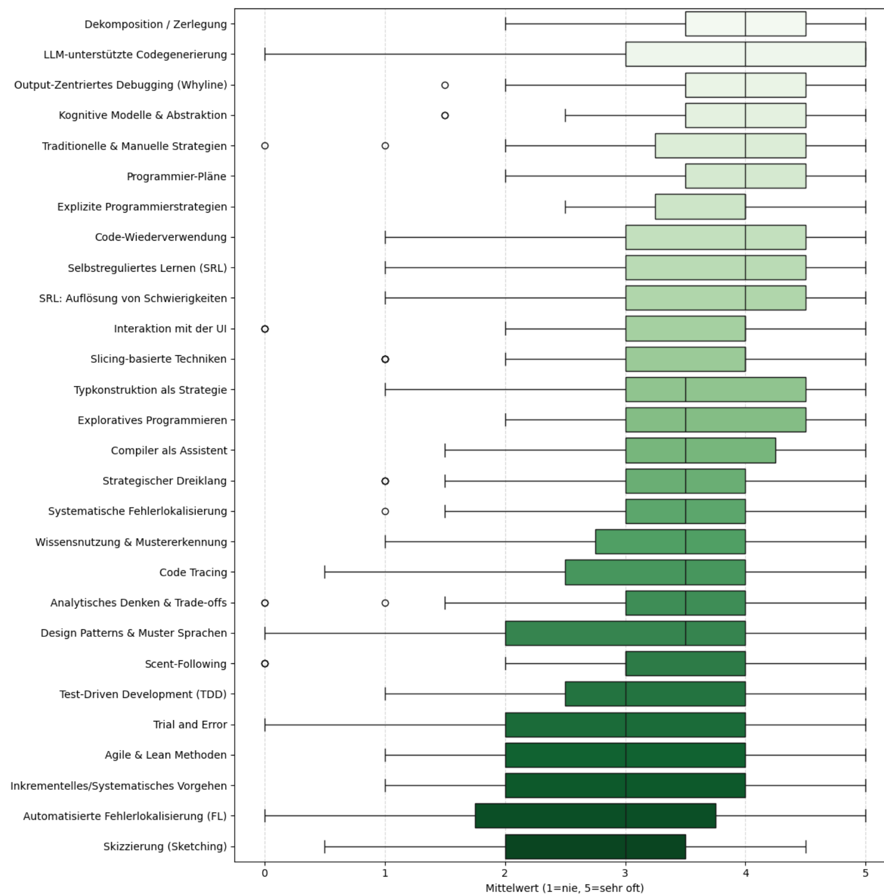

Abbildung 4.6.: Nutzungshäufigkeit Programmierstrategien

Strategien mit hoher Nutzungshäufigkeit Die am intensivsten genutzten Strategien weisen Mittelwerte zwischen $M = 3,71$ und $M = 3,94$ auf. Die höchsten Werte erzielen dabei die *Dekomposition / Zerlegung* sowie die *LLM-unterstützte Codegenerierung* (beide $M = 3,94$; $SD = 0,90$ bzw. $1,29$). Ebenfalls eine sehr hohe Anwendung finden das *Output-Zentrierte Debugging (Whyline)* ($M = 3,90$; $SD = 0,92$), *Kognitive Modelle & Abstraktion* ($M = 3,85$; $SD = 0,92$) sowie *Traditionelle & Manuelle Strategien* ($M = 3,84$; $SD = 1,19$). Die *Programmier-Pläne* liegen im Durchschnitt bei $M = 3,82$ ($SD = 0,75$). Weitere häufig genutzte Ansätze sind die *Code-Wiederverwendung* ($M = 3,74$), *Explizite Programmierstrategien* ($M = 3,74$) und das *Selbstregulierte Lernen (SRL)* ($M = 3,71$). Der Median liegt für alle Strategien dieser Gruppe bei $4,0$.

Strategien mit mittlerer Nutzungshäufigkeit Im mittleren Bereich der Nutzungshäufigkeit ($3,26 \leq M \leq 3,68$) finden sich vorwiegend analytische und unterstützende Techniken. Hierzu zählen *Seeking Social Assistance (SOA)* ($M = 3,68$), *Exploratives Programmieren* ($M = 3,63$), der *Compiler als Assistent* ($M = 3,60$) sowie der *Strategische Dreiklang* ($M = 3,47$). In ähnlicher Frequenz werden die *Interaktion mit der UI* ($M = 3,45$), die *Systematische Fehlerlokalisierung* ($M = 3,45$) und die *Wissensnutzung & Mustererkennung* ($M = 3,40$) eingesetzt. Die Gruppe wird durch *Slicing-basierte Techniken* ($M = 3,32$), *Code Tracing* ($M = 3,29$) und *Analytisches Denken & Trade-offs* ($M = 3,26$) vervollständigt. Die Mediane dieser Strategien bewegen sich zwischen $3,5$ und $4,0$.

Strategien mit geringerer Nutzungshäufigkeit Strategien mit Mittelwerten unter $3,2$ umfassen methodische Frameworks und spezialisierte Verhaltensweisen. Dazu gehören *Scent-Following* ($M = 3,16$), *Test-Driven Development (TDD)* ($M = 3,06$), *Trial and Error* ($M = 3,06$) sowie *Agile & Lean Methoden* ($M = 3,05$). Die geringste Anwendung innerhalb der Stichprobe finden das *Inkrementelle/Systematische Vorgehen* ($M = 2,97$), *Design Patterns & Muster Sprachen* ($M = 2,97$), die *Automatisierte Fehlerlokalisierung (FL)* ($M = 2,85$) und schließlich die *Skizzierung (Sketching)* mit dem niedrigsten Wert von $M = 2,69$ ($SD = 1,03$). Der Median dieser Gruppe liegt einheitlich bei $3,0$.

Streuung und Antwortvarianz Hinsichtlich der Konsistenz der Angaben zeigt sich die geringste Varianz bei den *Expliziten Programmierstrategien* ($SD = 0,73$), was auf eine homogene Nutzungshäufigkeit hindeutet. Im Gegensatz dazu weisen *Trial and Error* ($SD = 1,46$), die *Interaktion mit der UI* ($SD = 1,41$) sowie *Design Patterns* und die *Automatisierte Fehlerlokalisierung* (beide $SD = 1,39$) die höchsten Standardabweichungen auf. In diesen Fällen decken die Antworten der Teilnehmenden die gesamte Breite der Skala von 0 bzw. 1 bis 5 ab.

4.2.3. Deskriptive Statistik der psychologischen Strategien ohne Feedback

Für alle acht untersuchten Problemlösestrategien umfasste die Stichprobe jeweils $N = 31$ Beobachtungen. Im Folgenden werden die deskriptiven statistischen Kennwerte der einzelnen Strategien dargestellt.

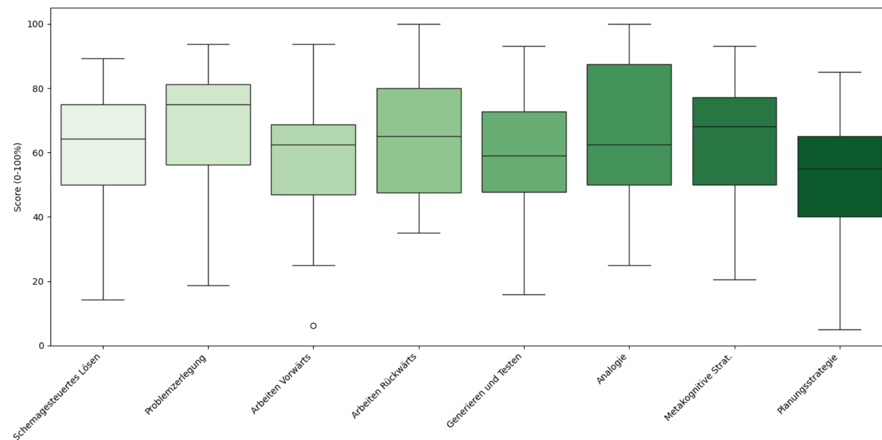

Abbildung 4.7.: Verteilung psychologischer Strategien (ohne Feedback)

Schemagesteuertes Lösen Beim schemagesteuerten Lösen ergab sich ein Mittelwert von $M = 61.64$ bei einer Standardabweichung von $SD = 19.14$. Die Werte reichten von einem Minimum von 14.29 bis zu einem Maximum von 89.29. Der Median lag bei 64.29.

Problemzerlegung Für die Strategie der Problemzerlegung wurde ein Mittelwert von $M = 68.35$ ($SD = 18.81$) berechnet. Das Minimum lag bei 18.75 und das Maximum bei 93.75, wobei sich ein Median von 75.00 ergab.

Arbeiten Vorwärts Die Auswertung der Strategie Arbeiten Vorwärts zeigte einen Mittelwert von $M = 57.06$ ($SD = 18.87$). Die Spannweite erstreckte sich von einem minimalen Wert von 6.25 bis zu einem maximalen Wert von 93.75. Der Median betrug 62.50.

Arbeiten Rückwärts Beim Arbeiten Rückwärts lag der Mittelwert bei $M = 64.03$ mit einer Standardabweichung von $SD = 19.12$. Der niedrigste gemessene Wert war 35.00, der höchste 100.00. Der Median lag hier bei 65.00.

Generieren und Testen Für die Strategie Generieren und Testen ergab die Auswertung einen Mittelwert von $M = 58.36$ ($SD = 20.16$). Die Werte reichten von einem Minimum von 15.91 bis zu einem Maximum von 93.18, bei einem Median von 59.09.

Analogie Die Anwendung der Analogie-Strategie resultierte in einem Mittelwert von $M = 65.32$ ($SD = 21.58$). Das Minimum betrug 25.00 und das Maximum 100.00. Als Median wurde ein Wert von 62.50 ermittelt.

Metakognitive Strategien Für die metakognitiven Strategien wurde ein Mittelwert von $M = 64.52$ ($SD = 17.10$) berechnet. Die Werte bewegten sich zwischen einem Minimum von 20.45 und einem Maximum von 93.18. Der Median lag bei 68.18.

Planungsstrategie Die Planungsstrategie wies einen Mittelwert von $M = 51.61$ ($SD = 19.93$) auf. Das Minimum bildete mit 5.00 den niedrigsten gemessenen Einzelwert aller Strategien, das Maximum lag bei 85.00. Der Median betrug 55.00.

4.2.4. Deskriptive Statistik der psychologischen Strategien mit Feedback

Um die rein rechnerischen Rohwerte des Fragebogens mit der subjektiven Reflexion der Probanden zu verknüpfen, wurde eine Adjustierung der Scores vorgenommen. Hierbei floss das qualitative Feedback zu den einzelnen Strategiekategorien (Items S10 bis S80) als Korrekturfaktor in die Berechnung ein. Gab ein Teilnehmer an, eine Strategie bewusst und erfolgreich eingesetzt zu haben, wurde der jeweilige Score um einen festgesetzten Adjustierungsfaktor von 15 Prozentpunkten erhöht. Im Falle einer negativen Rückmeldung zur Anwendung wurde der Score entsprechend um 15 Prozentpunkte gemindert. Die resultierenden adjustierten Werte wurden auf das Intervall zwischen 0 % und 100 % begrenzt, um eine konsistente prozentuale Skalierung zu gewährleisten.

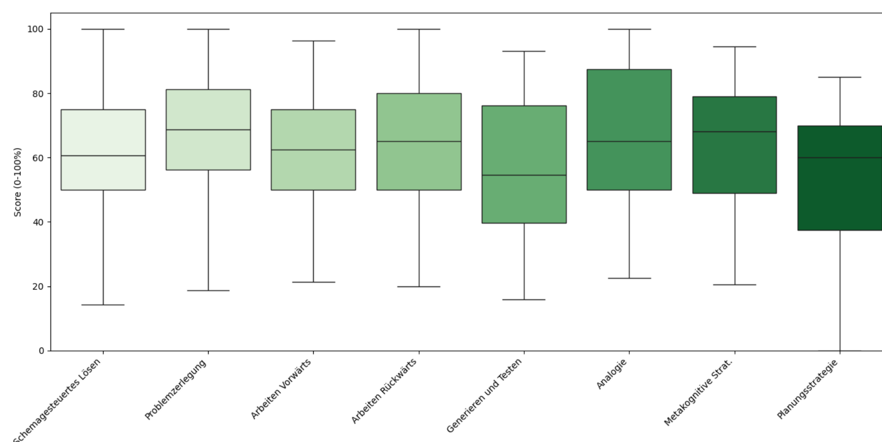

Abbildung 4.8.: Verteilung psychologischer Strategien (mit Feedback)

Schemagesteuertes Lösen Beim schemagesteuerten Lösen ergab sich ein Mittelwert von $M = 58.71$ bei einer Standardabweichung von $SD = 22.25$. Die Werte reichten von einem Minimum von 14.29 bis zu einem Maximum von 100.00. Der Median lag bei 60.71.

Problemzerlegung Für die Strategie der Problemzerlegung wurde ein Mittelwert von $M = 68.27$ ($SD = 19.10$) berechnet. Das Minimum lag bei 18.75 und das Maximum bei 100.00, wobei sich ein Median von 68.75 ergab.

Arbeiten Vorwärts Die Auswertung der Strategie Arbeiten Vorwärts zeigte einen Mittelwert von $M = 61.41$ ($SD = 18.28$). Die Spannweite erstreckte sich von einem minimalen Wert von 21.25 bis zu einem maximalen Wert von 96.25. Der Median betrug 62.50.

Arbeiten Rückwärts Beim Arbeiten Rückwärts lag der Mittelwert bei $M = 63.06$ mit einer Standardabweichung von $SD = 19.82$. Der niedrigste gemessene Wert war 20.00, der höchste 100.00. Der Median lag hier bei 65.00.

Generieren und Testen Für die Strategie Generieren und Testen ergab die Auswertung einen Mittelwert von $M = 56.91$ ($SD = 22.39$). Die Werte reichten von einem Minimum von 15.91 bis zu einem Maximum von 93.18, bei einem Median von 54.55.

Analogie Die Anwendung der Analogie-Strategie resultierte in einem Mittelwert von $M = 67.18$ ($SD = 22.53$). Das Minimum betrug 22.50 und das Maximum 100.00. Als Median wurde ein Wert von 65.00 ermittelt.

Metakognitive Strategien Für die metakognitiven Strategien wurde ein Mittelwert von $M = 63.55$ ($SD = 19.33$) berechnet. Die Werte bewegten sich zwischen einem Minimum von 20.45 und einem Maximum von 94.55. Der Median lag bei 68.18.

Planungsstrategie Die Planungsstrategie wies einen Mittelwert von $M = 55.00$ ($SD = 20.78$) auf. Das Minimum bildete mit 0.00 den niedrigsten gemessenen Einzelwert aller Strategien, das Maximum lag bei 85.00. Der Median betrug 60.00.

4.2.5. Auswertung einzelner Teilnehmer

In diesem Abschnitt werden die Ergebnisse der Clusteranalyse sowie die daraus resultierenden Gruppenstatistiken beschrieben, die zur Identifikation spezifischer Strategieprofile der Teilnehmer ($N = 31$) durchgeführt wurden. Als Grundlage dienten die mittels K-Means-Clustering gruppierten adjustierten Scores der acht psychologischen Strategiekategorien.

Ergebnisse der Clusterbildung und Gruppenstatistik Die Einteilung der Probanden erfolgte in drei Leistungsgruppen basierend auf ihrem Ø Gesamtscore über alle Strategien hinweg. Die Gruppe *Sehr gut* ($n = 9$) erreichte einen Ø Gesamtscore von $M = 81,76\,\%$. Das *Mittelfeld* stellt mit $n = 16$ die größte Gruppe dar und erzielt einen Ø Gesamtscore von $M = 60,24\,\%$. Die Gruppe *Schlecht* ($n = 6$) weist einen Ø Gesamtscore von $M = 35,82\,\%$ auf.

Gruppe	Abweichung	Gesamtscore	Teilnehmer
Sehr gut	9,40	81,76	9
Mittelfeld	13,79	60,24	16
Schlecht	14,37	35,82	6

Table 4.1.: Übersicht der Gruppenergebnisse

Analyse der Strategie-Ausgeglichenheit (Standardabweichung) Die durchschnittliche Standardabweichung innerhalb der Gruppen dient als Maß für die Ausgeglichenheit des jeweiligen Strategieprofils (interne Varianz pro Teilnehmer). Die Gruppe *Sehr gut* weist mit $M = 9,40$ die geringste Abweichung auf, was auf ein vergleichsweise ausgeglichenes Nutzenprofil über alle Kategorien hindeutet. Im *Mittelfeld* liegt die Ø Standardabweichung bei $M = 13,79$. Die Gruppe *Schlecht* zeigt mit $M = 14,37$ die höchste durchschnittliche Abweichung, was auf eine ungleichmäßige Verteilung der Strategieanwendung innerhalb dieser Probandengruppe hinweist.

Vergleich der Original- und Adjustierungswerte Die Auswertung umfasst den direkten Vergleich zwischen den ursprünglichen Rohwerten (Original) und den durch qualitatives Feedback modifizierten Werten (Angepasst). In den Einzelprofilen der Teilnehmer zeigt sich durch die Einbeziehung des Feedbacks (± 15 Prozentpunkte) eine Veränderung der Flächenausdehnung in den Netzdiagrammen. Als Beispiel folgt die individuelle Auswertung von Teilnehmer 12 und 21.

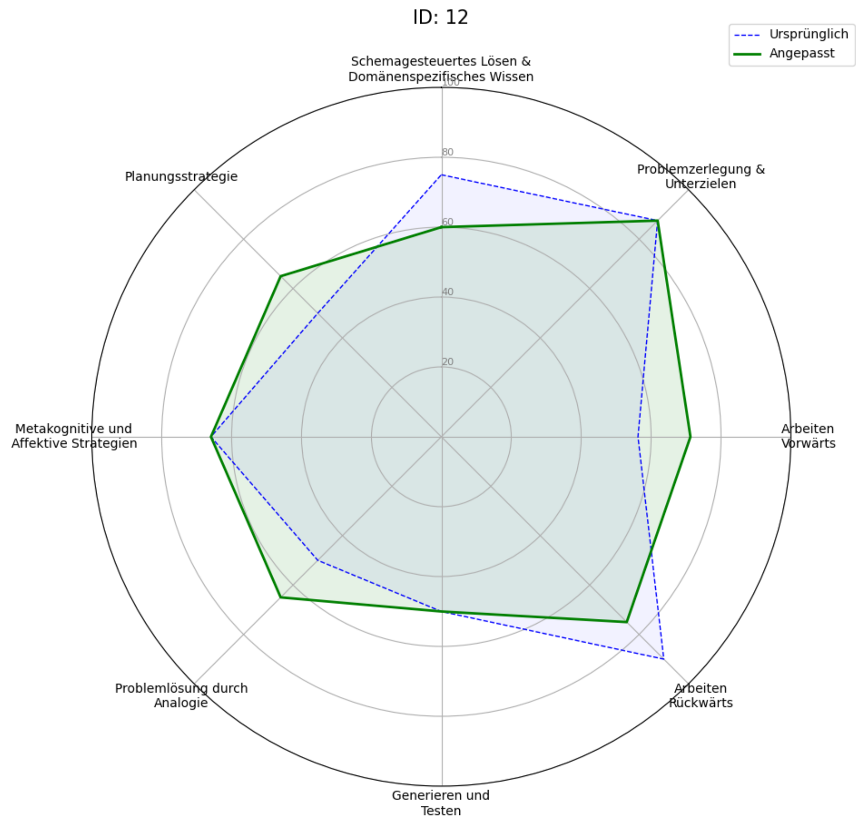

Abbildung 4.9.: Netzdiagramm Teilnehmer 12

Abbildung 4.10.: Netzdiagramm Teilnehmer 21

4.2.6. Pearson Korrelationsanalyse psychologische Strategien

In diesem Abschnitt werden die internen Zusammenhänge der psychologischen Strategiekategorien detailliert beschrieben. Hierbei wird zwischen den ursprünglichen Rohdaten (Original) und den durch das qualitative Feedback modifizierten Scores (Angepasst) unterschieden. Die Korrelationen wurden mittels des Pearson-Korrelationskoeffizienten (r) berechnet [35].

Korrelationsanalyse der Rohdaten (Original) Die Analyse der ursprünglichen Fragebogendaten zeigt durchweg positive und überwiegend starke Zusammenhänge zwischen den Strategiekategorien. Den höchsten linearen Zusammenhang weist die *Planungsstrategie* mit der Kategorie *Generieren und Testen* auf ($r = 0,83$). Ebenfalls sehr starke Korrelationen zeigen sich zwischen *Generieren und Testen* und dem *Arbeiten Vorwärts* ($r = 0,81$) sowie zwischen der *Problemzerlegung* und *Generieren und Testen* ($r = 0,75$). Die *Planungsstrategie* korreliert zudem deutlich mit dem *Arbeiten Vorwärts* ($r = 0,73$) und der *Problemzerlegung* ($r = 0,71$). Vergleichsweise geringere, aber dennoch moderate Korrelationen finden sich bei der Strategie *Analogie*, insbesondere im Verhältnis zur *Problemzerlegung* ($r = 0,39$) und zum *Schemagesteuerten Lösen* ($r = 0,41$).

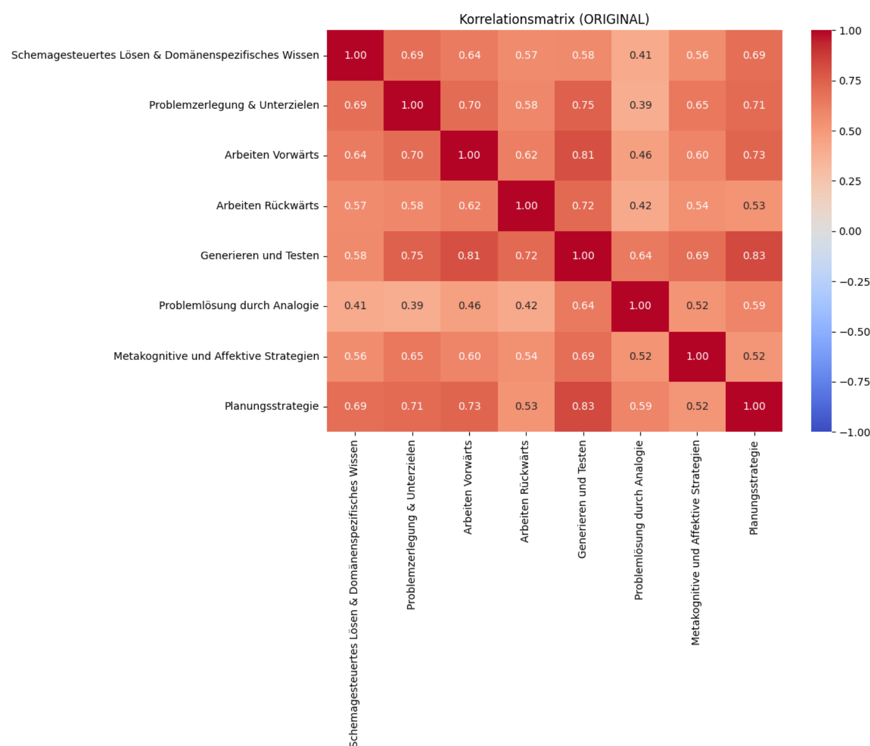

Abbildung 4.11.: Korrelationsmatrix psychologische Strategien (ohne Feedback)

Korrelationsanalyse der angepassten Daten Nach der Adjustierung der Scores durch den Korrekturfaktor von ±15 Prozentpunkten bleiben die Korrelationen positiv, weisen jedoch teilweise eine geringere Stärke und eine veränderte Struktur auf. Die stärksten Zusammenhänge zeigen sich nun zwischen der *Planungsstrategie* und *Generieren und Testen* ($r = 0,78$) sowie zwischen der *Planungsstrategie* und der *Problemzerlegung* ($r = 0,77$). Ein nahezu identisch starker Zusammenhang besteht zwischen dem *Schemagesteuerten Lösen* und der *Problemzerlegung* ($r = 0,77$). Das *Arbeiten Vorwärts* korreliert weiterhin deutlich mit der *Problemzerlegung* ($r = 0,75$) und dem *Arbeiten Rückwärts* ($r = 0,73$). Die geringsten Korrelationen in den adjustierten Daten finden sich zwischen dem *Arbeiten Rückwärts* und der *Planungsstrategie* ($r = 0,48$) sowie zwischen dem *Arbeiten Rückwärts* und den *Metakognitiven und Affektiven Strategien* ($r = 0,48$). Im Vergleich zu den Rohdaten führt die Adjustierung somit zu einer differenzierteren Verteilung der Zusammenhänge, wobei insbesondere die Korrelationen der Planungsstrategie zu den metakognitiven Aspekten leicht abnehmen.

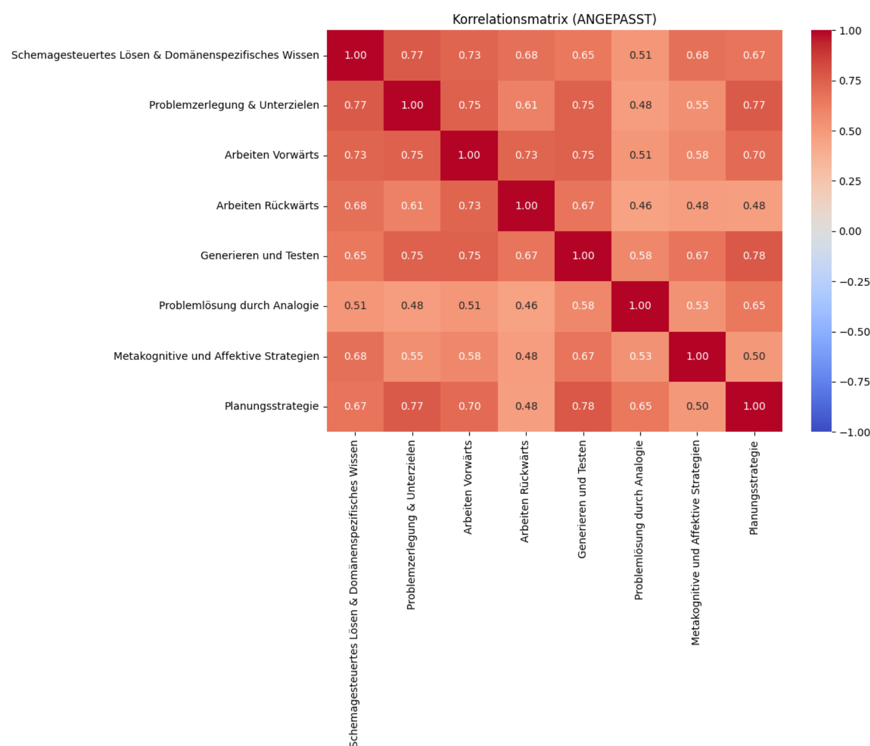

Abbildung 4.12.: Korrelationsmatrix psychologische Strategien (mit Feedback)

4.2.7. Spearman Analyse Programmiererfahrung und Programmierstrategien

Die Korrelationsanalyse nach Spearman (r_s) untersucht den Zusammenhang zwischen der Nutzungshäufigkeit der Programmierstrategien und den vier erhobenen Erfahrungsdimensionen. Die Koeffizienten geben Aufschluss darüber, wie stark die Anwendung einer Strategie mit der jeweiligen Form der Expertise korreliert [42].

Abbildung 4.13.: Heatmap Spearman Analyse: Nutzung spezifischer Programmierstrategien je nach Programmiererfahrung

Zusammenhang mit den Programmierjahren (PE1) Hinsichtlich der Dauer der Programmiererfahrung in Jahren zeigen sich die stärksten positiven Korrelationen bei den *Traditionellen & Manuellen Strategien* ($r_s = 0,57$). Ebenfalls deutlich positiv assoziiert sind das *Inkrementelle/Systematische Vorgehen* ($r_s = 0,39$) sowie die Anwendung von *Programmier-Plänen* ($r_s = 0,38$) und *Design Patterns & Muster Sprachen* ($r_s = 0,35$). Demgegenüber stehen negative Korrelationen bei der *Code-Wiederverwendung* ($r_s = -0,26$), dem *Explorativen Programmieren* ($r_s = -0,25$) sowie der *Systematischen Fehlerlokalisierung* ($r_s = -0,23$). Dies deutet darauf hin,

dass mit steigender Anzahl an Erfahrungsjahren vermehrt auf bewährte, manuelle und systematische Methoden zurückgegriffen wird, während explorative Ansätze und die bloße Wiederverwendung von Code abnehmen.

Zusammenhang mit der Selbsteinschätzung (PE2) Die subjektive Selbsteinschätzung der Kompetenz im Vergleich zu Peers weist die höchsten Korrelationen mit den *Traditionellen & Manuellen Strategien* ($r_s = 0,59$) sowie der *Wissensnutzung & Mustererkennung* ($r_s = 0,56$) auf. Weitere signifikante Zusammenhänge bestehen zum *Strategischen Dreiklang* ($r_s = 0,48$), den *Programmier-Plänen* ($r_s = 0,43$) und der *Dekomposition / Zerlegung* ($r_s = 0,41$). Teilnehmer, die sich selbst als erfahrener einschätzen, nutzen zudem häufiger das *Output-Zentrierte Debugging* ($r_s = 0,34$). Eine negative Korrelation zeigt sich hier erneut bei der *Code-Wiederverwendung* ($r_s = -0,20$) sowie bei der *LLM-unterstützten Codegenerierung* ($r_s = -0,17$), was darauf hindeutet, dass eine höhere selbsteingeschätzte Expertise mit einer geringeren Abhängigkeit von externen Generierungswerkzeugen einhergeht.

Zusammenhang mit der Programmierhäufigkeit (PE3) In Bezug auf die Regelmäßigkeit der Programmiertätigkeit korreliert die *Dekomposition / Zerlegung* am stärksten positiv ($r_s = 0,54$). Weitere positive Zusammenhänge zeigen sich beim *Code Tracing* ($r_s = 0,38$), den *Traditionellen & Manuellen Strategien* ($r_s = 0,37$) sowie beim *Output-Zentrierten Debugging (Whyline)* ($r_s = 0,35$) und dem *Explorativen Programmieren* ($r_s = 0,34$). Auffällig ist, dass eine hohe Programmierfrequenz negativ mit *Trial and Error* ($r_s = -0,26$) und der *Code-Wiederverwendung* ($r_s = -0,23$) korreliert. Wer häufig programmiert, scheint somit verstärkt analytische Techniken (Zerlegung, Tracing) einzusetzen und weniger auf unsystematisches Ausprobieren zu setzen.

Zusammenhang mit der Projektanzahl (PE4) Die Anzahl der bearbeiteten Projekte korreliert am intensivsten mit der *Dekomposition / Zerlegung* ($r_s = 0,55$) und den *Traditionellen & Manuellen Strategien* ($r_s = 0,51$). Bemerkenswert sind hier zudem die starken positiven Korrelationen zu den *Agile & Lean Methoden* ($r_s = 0,48$), der *Wissensnutzung & Mustererkennung* ($r_s = 0,47$) sowie den *Design Patterns & Muster Sprachen* ($r_s = 0,43$). Auch die *Expliziten Programmierstrategien* ($r_s = 0,38$) und *Programmier-Pläne* ($r_s = 0,38$) zeigen einen deutlichen Zusammenhang mit der Projektvielfalt. Eine negative Korrelation zur Projektanzahl ist bei der *Systematischen Fehlerlokalisierung* ($r_s = -0,10$) und der *Code-Wiederverwendung* ($r_s = -0,07$) zu beobachten, wenngleich diese Zusammenhänge schwächer ausgeprägt sind als in den anderen Dimensionen.

4.2.8. Spearman Analyse Programmiererfahrung und psychologische Strategien

Die Korrelationsanalyse nach Spearman (r_s) untersucht den Zusammenhang zwischen den adjustierten Scores der psychologischen Strategien und den vier Erfahrungsdimensionen: Programmierjahre (PE1), Selbsteinschätzung (PE2), Programmierhäufigkeit (PE3) und Projektanzahl (PE4) [42].

Abbildung 4.14.: Heatmap Spearman Analyse: Nutzung psychologische Strategien je nach Programmiererfahrung

Zusammenhang mit den Programmierjahren (PE1) In Bezug auf die Dauer der Programmiererfahrung in Jahren weisen alle psychologischen Strategien mit Ausnahme der Kategorie *Analogie* positive Korrelationskoeffizienten auf. Den stärksten Zusammenhang zeigt das *Schemagesteuerte Lösen* mit $r_s = 0,29$, gefolgt von der *Planungsstrategie* mit $r_s = 0,20$. Geringere positive Korrelationen finden sich bei der *Problemzerlegung* ($r_s = 0,12$), dem *Arbeiten Rückwärts* ($r_s = 0,09$) und dem *Arbeiten Vorwärts* ($r_s = 0,08$). Die Kategorien *Metakognitive Strategien* ($r_s = 0,06$) sowie *Generieren und Testen* ($r_s = 0,04$) zeigen nahezu neutrale Zusammenhänge. Einzig die Strategie *Analogie* weist eine geringfügig negative Korrelation von $r_s = -0,03$ auf.

Zusammenhang mit der Selbsteinschätzung (PE2) Die subjektive Selbsteinschätzung der Programmiererfahrung im Vergleich zu Peers zeigt durchgehend pos-

itive Korrelationen zu allen untersuchten psychologischen Strategien. Der deutlichste Zusammenhang besteht zum *Schemagesteuerten Lösen* mit $r_s = 0,56$. Eine Gruppe von Strategien, bestehend aus der *Problemzerlegung* ($r_s = 0,36$), dem *Arbeiten Vorwärts* ($r_s = 0,35$), den *Metakognitiven Strategien* ($r_s = 0,34$), der *Planungsstrategie* ($r_s = 0,34$), dem *Generieren und Testen* ($r_s = 0,34$) sowie der *Analogie* ($r_s = 0,34$), weist moderate positive Koeffizienten auf. Der geringste Zusammenhang mit der Selbsteinschätzung zeigt sich beim *Arbeiten Rückwärts* ($r_s = 0,19$).

Zusammenhang mit der Programmierhäufigkeit (PE3) Hinsichtlich der Regelmäßigkeit der Programmiertätigkeit wird die stärkste positive Korrelation bei der *Problemzerlegung* mit $r_s = 0,36$ beobachtet. Moderate Zusammenhänge bestehen zudem beim *Arbeiten Vorwärts* ($r_s = 0,18$), den *Metakognitiven Strategien* ($r_s = 0,17$) sowie dem *Schemagesteuerten Lösen* ($r_s = 0,16$). Geringere Koeffizienten finden sich beim *Generieren und Testen* ($r_s = 0,13$), der *Planungsstrategie* ($r_s = 0,13$) und der *Analogie* ($r_s = 0,08$). Ein nahezu verschwindender Zusammenhang zeigt sich bei der Strategie *Arbeiten Rückwärts* ($r_s = 0,01$).

Zusammenhang mit der Projektanzahl (PE4) Die Anzahl der bearbeiteten Projekte korreliert ebenfalls positiv mit allen betrachteten psychologischen Strategien. Analog zur Selbsteinschätzung zeigt das *Schemagesteuerte Lösen* mit $r_s = 0,55$ den stärksten Zusammenhang. Deutliche positive Korrelationen finden sich zudem bei den *Metakognitiven Strategien* ($r_s = 0,38$), der *Planungsstrategie* ($r_s = 0,33$) und der *Problemzerlegung* ($r_s = 0,32$). Die weiteren Kategorien weisen moderate Zusammenhänge auf, namentlich das *Arbeiten Rückwärts* ($r_s = 0,29$), das *Generieren und Testen* ($r_s = 0,26$) sowie die *Analogie* ($r_s = 0,25$). Die geringste Korrelation mit der Projektanzahl weist das *Arbeiten Vorwärts* ($r_s = 0,12$) auf.

4.2.9. Cronbachs Alpha

Zur Überprüfung der Reliabilität der im Fragebogen verwendeten Skalen für die psychologischen Strategien wurde die interne Konsistenz mittels Cronbachs Alpha (α) berechnet ($N = 31$) [37]. Die Koeffizienten geben Aufschluss darüber, inwieweit die Items innerhalb einer Strategiekategorie ein homogenes Konstrukt messen.

Table 4.2.: Interne Konsistenz (Cronbachs Alpha) der psychologischen Strategiekategorien ($N = 31$)

Strategiekategorie	**Cronbachs Alpha** (α)
Generieren und Testen	0,811
Metakognitive Strategien	0,785
Schemagesteuertes Lösen	0,726
Arbeiten Rückwärts	0,631
Planungsstrategie	0,579
Problemzerlegung	0,474
Arbeiten Vorwärts	0,449
Analogie	0,263

Skalen mit hoher interner Konsistenz Die höchste interne Konsistenz innerhalb der untersuchten Kategorien weist die Strategie *Generieren und Testen* mit einem Koeffizienten von $\alpha = 0,81$ auf. Ebenfalls im Bereich einer guten bis akzeptablen Reliabilität liegen die Skalen für die *Metakognitiven Strategien* ($\alpha = 0,79$) und das *Schemagesteuerte Lösen* ($\alpha = 0,73$).

Skalen im mittleren Reliabilitätsbereich Einen moderaten Reliabilitätswert zeigt die Strategie *Arbeiten Rückwärts* mit einem Alpha-Koeffizienten von $\alpha = 0,63$. Die *Planungsstrategie* erreicht einen Wert von $\alpha = 0,58$, was auf eine eher geringe interne Konsistenz der zugehörigen Items hindeutet.

Skalen mit niedriger interner Konsistenz Werte unterhalb der Schwelle von 0,5 wurden für drei Strategiekategorien verzeichnet. Die Skalen für die *Problemzerlegung* ($\alpha = 0,47$) und das *Arbeiten Vorwärts* ($\alpha = 0,45$) weisen eine niedrige interne Konsistenz auf. Den geringsten Koeffizienten zeigt die Strategie *Analogie* mit einem Alpha-Wert von $\alpha = 0,26$.

5. Diskussion

5.1. Beantwortung der Forschungsfragen

5.1.1. Forschungsfrage 1

> *In welchem Ausmaß und mit welcher Verteilung wenden Programmierende die im Fragebogen definierten Programmierstrategien in ihrem Entwicklungsprozess an?*

Die erste Forschungsfrage untersuchte, in welchem Ausmaß und mit welcher Verteilung Programmierende die definierten Programmierstrategien in ihrem Entwicklungsprozess anwenden. Basierend auf den deskriptiven Statistiken in Abschnitt 4.2.1 und 4.2.2 lassen sich folgende zentrale Erkenntnisse ableiten:

Nutzungsintensität und Favorisierte Strategien Die Ergebnisse verdeutlichen, dass Programmierstrategien keineswegs homogen eingesetzt werden, sondern eine klare Hierarchie in der Anwendung existiert. Wie in Abbildung 4.6 dargestellt, erreichen die am intensivsten genutzten Strategien Mittelwerte im Bereich von $M = 3,71$ bis $M = 3,94$. Besonders hervorzuheben sind hierbei die *Dekomposition / Zerlegung* sowie die *LLM-unterstützte Codegenerierung* (beide $M = 3,94$). Dies deutet darauf hin, dass sowohl klassische analytische Ansätze als auch moderne, KI-gestützte Werkzeuge integrale Bestandteile des heutigen Entwicklungsprozesses sind. Auch das *Output-Zentrierte Debugging* ($M = 3,90$) und *Traditionelle & Manuelle Strategien* ($M = 3,84$) weisen eine sehr hohe Anwendungshäufigkeit auf.

Verteilung und Varianz im Nutzungsverhalten Die Verteilung der Strategien zeigt ein breites Spektrum. Während die Top-Strategien Mediane von $4,0$ aufweisen (vgl. Abschnitt 4.2.2), zeigt sich bei methodischeren oder spezialisierteren Ansätzen ein deutlicher Abfall der Nutzungshäufigkeit. Strategien wie *Design Patterns* ($M = 2,97$), *Automatisierte Fehlerlokalisierung* ($M = 2,85$) oder das *Skizzierung (Sketching)* ($M = 2,69$) bilden das Schlusslicht der Verteilung.Interessant ist hierbei die Heterogenität der Anwendung: Wie die Analyse der Streuung in Abschnitt 4.2.2 zeigt, weisen insbesondere *Trial and Error* ($SD = 1,46$) und *Design Patterns* ($SD = 1,39$) hohe Standardabweichungen auf. Dies lässt darauf schließen, dass diese Strategien stark individuell geprägt sind – während einige Teilnehmende sie intensiv nutzen, spielen sie für andere nahezu keine Rolle. Im Gegensatz dazu werden *Explizite Programmierstrategien* ($SD = 0,73$) wesentlich konsistenter über die gesamte Stichprobe hinweg angewendet.

Zusammenfassende Einordnung Zusammenfassend lässt sich festhalten, dass die Teilnehmenden primär auf pragmatische und direkt ergebnisorientierte Strategien setzen (Zerlegung, LLM-Tools, Debugging). Methodisch-formale Ansätze wie *Test-Driven Development (TDD)* ($M = 3,06$) oder das Arbeiten mit Entwurfsmustern sind in der untersuchten Stichprobe, die überwiegend aus Personen in der fortgeschrittenen Ausbildung oder frühen Berufstätigkeit besteht (siehe Abschnitt 4.1), deutlich weniger verankert. Die Programmiertätigkeit wird somit eher durch punktuelle Problemlösung und den Einsatz moderner Hilfsmittel als durch strikte methodische Frameworks charakterisiert.

5.1.2. Forschungsfrage 2

> *In welchem Ausmaß und mit welcher Verteilung kommen die generischen, psychologischen Problemlösungsstrategien im Vorgehen der Programmierenden zum Einsatz?*

Die zweite Forschungsfrage befasst sich mit dem Ausmaß und der Verteilung der generischen psychologischen Problemlösungsstrategien. Hierbei wurde zwischen den rein rechnerischen Rohwerten und den durch qualitatives Feedback adjustierten Werten unterschieden, um ein realistischeres Bild der bewussten Strategieanwendung zu erhalten.

Ausmaß der psychologischen Strategieanwendung Die Analyse der Rohdaten in Abschnitt 4.2.3 zeigt, dass die *Problemzerlegung* mit einem Mittelwert von $M = 57,06\,\%$ die am stärksten ausgeprägte psychologische Strategie darstellt (siehe Abbildung 4.7). Es folgen die Strategien *Analogie* ($M = 54,84\,\%$) und *Arbeiten Vorwärts* ($M = 54,44\,\%$). Am geringsten ausgeprägt sind die *Metakognitiven Strategien* ($M = 45,16\,\%$).Ein signifikanter Anstieg des Ausmaßes zeigt sich nach der Einbeziehung des qualitativen Feedbacks (Abschnitt 4.2.4). Durch die Adjustierung stieg der Mittelwert der *Problemzerlegung* auf $M = 68,27\,\%$ und der der *Analogie* auf $M = 66,13\,\%$ (Abbildung 4.8). Diese Verschiebung nach oben verdeutlicht, dass die Teilnehmenden ihren tatsächlichen Strategieeinsatz in der Reflexion oft höher und erfolgreicher bewerten, als es die reinen Item-Antworten vermuten lassen.

Verteilung und individuelle Variabilität Die Verteilung der Strategien innerhalb der Stichprobe ist durch eine hohe Heterogenität gekennzeichnet. Besonders die Strategie *Analogie* weist mit $SD = 20,57\,\%$ (Original) bzw. $SD = 23,47\,\%$ (Adjustiert) die größte Streuung auf. Dies deutet darauf hin, dass das Denken in Analogien eine sehr individuell ausgeprägte Fähigkeit ist. Im Gegensatz dazu ist die Strategie *Generieren und Testen* ($SD = 12,23\,\%$) am gleichmäßigsten über die Probanden verteilt. Die Clusteranalyse in Abschnitt 4.2.5 gibt weiteren Aufschluss über die Verteilung:

- Die Gruppe *Sehr gut* ($n = 9$) nutzt psychologische Strategien nicht nur intensiver ($M = 81,76\,\%$), sondern auch **ausgeglichener** ($SD = 9,40$).

- Die Gruppe *Schlecht* ($n = 6$) zeigt hingegen ein sehr ungleichmäßiges Profil ($SD = 14,37$), was auf eine fragmentierte Anwendung von Problemlösungstechniken hindeutet.

Interne Zusammenhänge der Strategien Wie die Korrelationsanalyse in Abschnitt 4.2.6 zeigt, treten psychologische Strategien selten isoliert auf. Besonders starke Synergien bestehen zwischen der *Planungsstrategie* und *Generieren und Testen* ($r = 0,83$) sowie der *Problemzerlegung* ($r = 0,71$). Dies lässt den Schluss zu, dass eine strukturierte Zerlegung des Problems fast zwangsläufig mit einer planvollen Generierung von Lösungsansätzen einhergeht.

Zusammenfassende Einordnung Zusammenfassend lässt sich festhalten, dass psychologische Problemlösungsstrategien im Mittel zu etwa 50 % bis 68 % (je nach Metrik) zum Einsatz kommen. Die *Problemzerlegung* bildet dabei das kognitive Fundament des Programmierprozesses. Ein deutliches Defizit zeigt sich jedoch im Bereich der *Metakognitiven Strategien*, die über alle Auswertungen hinweg die niedrigsten Werte erzielen. Dies deutet darauf hin, dass die Überwachung und Regulation des eigenen Denkprozesses während des Programmierens seltener bewusst erfolgt als die operative Problemlösung selbst.

5.1.3. Forschungsfrage 3

> *Inwiefern beeinflusst der Grad der Programmiererfahrung die Präferenz für spezifische Programmierstrategien?*

Die dritte Forschungsfrage untersuchte den Zusammenhang zwischen dem Grad der Programmiererfahrung – differenziert nach Jahren (PE1), Selbsteinschätzung (PE2), Häufigkeit (PE3) und Projektanzahl (PE4) – und der Präferenz für spezifische Strategien. Die Ergebnisse der Spearman-Analyse (Abschnitte 4.2.7 und 4.2.8) zeigen deutliche Korrelationsmuster.

Einfluss auf die Wahl der Programmierstrategien Der Grad der Erfahrung beeinflusst maßgeblich die Abkehr von explorativen hin zu systematischen und analytischen Methoden. Wie in Abbildung 4.13 ersichtlich, korrelieren sowohl die Programmierjahre ($r_s = 0,57$) als auch die Selbsteinschätzung ($r_s = 0,59$) stark positiv mit *Traditionellen & Manuellen Strategien*. Erfahrene Programmierende greifen somit verstärkt auf gefestigtes Handwerkswissen zurück.Besonders hervorzuheben ist die Rolle der *Dekomposition / Zerlegung*. Diese zeigt eine starke Korrelation mit der Programmierhäufigkeit ($r_s = 0,54$) und der Projektanzahl ($r_s = 0,55$). Dies legt nahe, dass die Fähigkeit, komplexe Probleme in handhabbare Teilaufgaben zu zerlegen, ein direktes Resultat intensiver praktischer Übung und Projektroutine ist. Im Gegensatz dazu korreliert eine höhere Expertise negativ mit der *Code-Wiederverwendung* und dem *Explorativen Programmieren*. Interessanterweise zeigt

sich bei Teilnehmern mit hoher Selbsteinschätzung auch eine geringere Präferenz für die *LLM-unterstützte Codegenerierung* ($r_s = -0,17$), was auf ein höheres Vertrauen in die eigenen Problemlösungsfähigkeiten hindeutet.

Einfluss auf psychologische Problemlösungsstrategien Im Bereich der psychologischen Strategien (siehe Abbildung 4.14) ist das *Schemagesteuerte Lösen* der am stärksten durch Erfahrung beeinflusste Faktor. Es korreliert signifikant mit der Selbsteinschätzung ($r_s = 0,56$) und der Projektanzahl ($r_s = 0,55$). Dies bestätigt kognitionspsychologische Modelle, nach denen Experten verstärkt auf mentale Repräsentationen (Schemata) zurückgreifen, um bekannte Problemstrukturen effizient zu adressieren.Ein weiterer wichtiger Aspekt ist die Entwicklung metakognitiver Fähigkeiten. Mit steigender Anzahl an bearbeiteten Projekten nimmt die Nutzung *Metakognitiver Strategien* zu ($r_s = 0,38$). Während Anfänger eher operativ mit dem Code interagieren, scheint die Erfahrung in vielfältigen Projekten die Fähigkeit zur Überwachung und Regulation des eigenen Denkprozesses zu fördern.

Zusammenfassende Einordnung Zusammenfassend lässt sich festhalten, dass Programmiererfahrung zu einer Professionalisierung des Vorgehens führt: weg von *Trial and Error* und reiner Code-Manipulation, hin zu analytischer Zerlegung, schemagesteuerter Problemlösung und der Anwendung expliziter Entwurfsmuster. Die Erfahrung fungiert hierbei als Katalysator für eine strukturiertere und bewusstere Strategiewahl, wobei die Projektanzahl und die subjektive Kompetenzwahrnehmung stärkere Prädiktoren für fortgeschrittene Strategien sind als die reine Anzahl der Berufsjahre.

5.1.4. Forschungsfrage 4

> *Inwieweit lässt sich das theoretische Mapping von domänenspezifischen Programmierstrategien auf generische, psychologische Problemlösungsstrategien empirisch bestätigen?*

Die vierte Forschungsfrage untersuchte die empirische Belastbarkeit des theoretischen Mappings von Programmierstrategien auf generische, psychologische Problemlösungsstrategien. Als zentrales Prüfmaß für die Güte dieses Mappings diente die interne Konsistenz (Cronbachs Alpha), die angibt, inwieweit die einem Konstrukt zugeordneten Items tatsächlich eine gemeinsame Dimension messen (siehe Abschnitt 4.2.9).

Empirisch bestätigte Mappings Eine starke empirische Bestätigung fand das Mapping für drei der acht untersuchten Kategorien. Wie in Tabelle 4.2 dargestellt, weisen die Skalen für *Generieren und Testen* ($\alpha = 0,81$), *Metakognitive Strategien* ($\alpha = 0,79$) und *Schemagesteuertes Lösen* ($\alpha = 0,73$) eine gute bis akzeptable Reliabilität auf. Dies belegt, dass die für diese Kategorien gewählten Programmier-Items

im Erleben der Probanden ein homogenes psychologisches Konstrukt abbilden. Das theoretische Modell lässt sich hier somit direkt auf die Praxis übertragen.

Eingeschränkt bestätigte Mappings Ein moderater Zusammenhang innerhalb der Kategorien zeigte sich beim *Arbeiten Rückwärts* ($\alpha = 0,63$) und der *Planungsstrategie* ($\alpha = 0,58$). Während diese Werte noch im vertretbaren Bereich für explorative Studien liegen, deutet die geringere Konsistenz darauf hin, dass die Probanden die zugehörigen Programmieraktivitäten (z.,B. das Definieren von Zielen vs. das Erstellen von Ablaufplänen) als teilweise unterschiedliche Konzepte wahrnehmen. Das Mapping ist hier zwar erkennbar, aber weniger trennscharf.

Problematische Mappings und Revisionsbedarf Kritisch zu betrachten sind die Kategorien *Problemzerlegung* ($\alpha = 0,47$), *Arbeiten Vorwärts* ($\alpha = 0,45$) und insbesondere die *Analogie* ($\alpha = 0,26$). Die sehr niedrigen Alpha-Werte signalisieren, dass die diesen psychologischen Strategien zugeordneten Programmier-Items empirisch kaum miteinander korrelieren.

- Bei der *Problemzerlegung* könnte dies daran liegen, dass die Teilnehmenden das "Zerlegen in Module" (top-down) psychologisch anders bewerten als das "inkrementelle Hinzufügen von Funktionen" (bottom-up), obwohl beides theoretisch zur Dekomposition gehört.

- Der extrem niedrige Wert der *Analogie* legt nahe, dass der Transfer von bestehendem Code oder Mustern auf neue Probleme ein zu komplexes und heterogenes Feld ist, um durch die gewählten Items einheitlich erfasst zu werden. Zudem waren dieser Strategie nur 2 Fragen zugeordnet.

Zusammenfassende Einordnung Zusammenfassend lässt sich das theoretische Mapping für etwa die Hälfte der Kategorien (insbesondere operative und metakognitive Strategien) empirisch bestätigen. Für die eher abstrakten Problemlösungsstile (Analogie, Vorwärts-Strategie) muss das Mapping jedoch als instabil betrachtet werden. Die Ergebnisse der Korrelationsanalyse (Abschnitt 4.2.6) stützen dies. Die starken Zusammenhänge zwischen den Kategorien (z.,B. Planung und Generieren/Testen mit $r = 0,83$) deuten darauf hin, dass Programmierende psychologische Strategien eher als vernetztes "Gesamtpaket" anwenden, statt sie voneinander zu isolieren.

5.2. Qualitative Auswertung des Teilnehmerfeedbacks

Zusätzlich zu den quantitativen Daten wurden die qualitativen Rückmeldungen der Teilnehmer analysiert, um ein tieferes Verständnis für die Anwendung von Problemlösestrategien und die Wahrnehmung des Selbsteinschätzungsfragebogens zu gewinnen. Die Auswertung erfolgte durch eine strukturierte Inhaltsanalyse der Freitextantworten.

5.2.1. Validierung der Selbsteinschätzung

Ein Großteil der Teilnehmer, die Feedback gaben, bestätigte die Genauigkeit der durch das Tool ermittelten Profile. Teilnehmer 27 betonte wiederholt, dass die Strategien mit hohen Prozentwerten exakt seinem üblichen Problemlösungsmuster entsprächen („The assessment is largely accurate; the high-percentage traits match my usual problem-solving pattern"). Auch Teilnehmer 14 und 15 stuften die Ergebnisse als effektiv und weitgehend korrekt (ca. 80–82 % Übereinstimmung) ein.

5.2.2. Detaillierte Analyse spezifischer Strategien

Die Freitextantworten geben Aufschluss darüber, wie einzelne Strategien in der Praxis interpretiert und angewendet werden:

- **Problemzerlegung (S2):** Diese Strategie wird als sehr wertvoll erachtet, jedoch oft nicht linear angewendet. Teilnehmer 1 gab an, ein großer Fan der Zerlegung zu sein, den Prozess aber „weniger strukturiert" und „eher nach Gefühl" als in einer festen Reihenfolge durchzuführen. Dabei wird oft mit einem Kernaspekt begonnen und das System von dort aus erweitert.

- **Arbeiten Vorwärts vs. Rückwärts (S3 & S4):** Hier zeigte sich eine Präferenz für intuitive, vorwärtsgerichtete Prozesse. Teilnehmer 1 beschrieb einen iterativen Ansatz über ein *Minimum Viable Product* (MVP), anstatt vom Zielzustand rückwärts zu planen.

- **Analogien (S6):** Während einige Teilnehmer Analogien gezielt nutzen, berichtete Teilnehmer 21 von Schwierigkeiten, sich bei Problemen an frühere, ähnliche Lösungen zu erinnern („wenn mir auffällt dass der code seltsam aussieht, dann erinnere ich mich oft nicht an frühere problem").

- **Metakognition und Planung (S7 & S8):** Teilnehmer assoziieren Planung oft mit To-Do-Listen. Ein Hindernis für die Anwendung hochstrukturierter Strategien (wie „Generieren und Testen") ist laut Feedback oft ein subjektiv empfundener Mangel an eigener Struktur.

5.2.3. Kritische Würdigung des methodischen Vorgehens

Die qualitativen Rückmeldungen enthielten auch wichtige Hinweise zur Verbesserung des Erhebungsinstruments:

- **Verständlichkeit:** Mehrere Teilnehmer (P1, P9) merkten an, dass einige Strategiedefinitionen und Fragen zu lang oder schwer verständlich seien. Es wurde der Wunsch nach präziseren Erklärungen geäußert, um die Strategien besser voneinander abgrenzen zu können.
- **Bias-Gefahr:** Teilnehmer 31 äußerte eine methodische Kritik: Durch das Anbieten von Lösungswegen in den Fragen könnten Probanden dazu verleitet werden, zu analysieren, wie sie eine Aufgabe lösen *sollten*, anstatt ehrlich zu reflektieren, wie sie diese tatsächlich lösen.
- **Hilfsmittel:** Teilnehmer 21 hob die Bedeutung von *Large Language Models* (LLMs) hervor, insbesondere wenn die Vertrautheit mit einer Programmiersprache gering ist.

5.2.4. Zusammenfassung der qualitativen Ergebnisse

Zusammenfassend lässt sich festhalten, dass das Tool eine hohe Face-Validity besitzt. Die Teilnehmer identifizieren sich mit den Mustern, weisen aber darauf hin, dass Problemlösungsprozesse in der Realität oft weniger strukturiert und intuitiver ablaufen, als es die theoretischen Modelle der Strategien suggerieren. Die Flexibilität, Probleme in Teilschritte zu zerlegen, wird als Kernkompetenz angesehen, während starre, rückwärtsgerichtete oder rein schematische Ansätze seltener bewusst wahrgenommen werden.

6. Einschränkungen der Validität

Die Ergebnisse der vorliegenden Untersuchung müssen vor dem Hintergrund methodischer Einschränkungen interpretiert werden. Im Folgenden werden die Konstruktvalidität, die interne Validität sowie die externe Validität kritisch reflektiert.

6.1. Konstruktvalidität

Die Konstruktvalidität befasst sich mit der Frage, inwieweit die gewählten Items tatsächlich die theoretischen Konstrukte (hier: psychologische Problemlösungsstrategien) abbilden.

- **Operationalisierung:** Das Mapping von domänenspezifischen Programmierpraktiken auf generische psychologische Strategien stellt eine theoretische Transferleistung dar. Wie die Analyse von Cronbachs Alpha in Abschnitt 4.2.9 zeigt, weisen Kategorien wie *Analogie* ($\alpha = 0,26$) oder *Arbeiten Vorwärts* ($\alpha = 0,45$) eine geringe interne Konsistenz auf. Dies deutet darauf hin, dass die gewählten Items für diese komplexen Konstrukte keine homogene Dimension erfassen.
- **Subjektivität der Selbsteinschätzung:** Da die Daten auf Selbstauskünften basieren, besteht die Gefahr eines *Social Desirability Bias* oder von Fehleinschätzungen der eigenen Kompetenz. Wie in Abschnitt 3.6 angemerkt, könnten Teilnehmer durch die Fragen-Formulierung dazu verleitet worden sein, ein idealisiertes statt des tatsächlichen Vorgehens anzugeben.

6.2. Interne Validität

Die interne Validität gibt an, inwieweit die beobachteten Zusammenhänge (z.B. zwischen Erfahrung und Strategiewahl) eindeutig interpretierbar sind.

- **Störvariablen:** Da die Erhebung als Gelegenheitsstichprobe (siehe 3.5) durchgeführt wurde, konnten kognitive Grundfähigkeiten (z. B. fluide Intelligenz) oder die spezifische Art der Ausbildung nicht kontrolliert werden. Diese Faktoren könnten die Präferenz für bestimmte Strategien (z. B. *Dekomposition*) stärker beeinflusst haben als die reine Programmiererfahrung.

- **Kausalität:** Aufgrund des Querschnitt-Designs der Studie lassen sich Korrelationen, aber keine kausalen Wirkmechanismen belegen. Es bleibt unklar, ob eine hohe Expertise zur Nutzung systematischer Strategien führt oder ob die Anwendung dieser Strategien den Erwerb von Expertise erst ermöglicht.

6.3. Externe Validität

Die externe Validität beschreibt die Generalisierbarkeit der Ergebnisse auf andere Kontexte und Populationen.

- **Stichprobencharakteristika:** Die Stichprobe ($N = 31$) besteht überwiegend aus Personen mit einer Erfahrung von 1–5 Jahren (74,19 %) und Teilnehmenden aus dem akademischen Umfeld der Technischen Universität (siehe 4.1). Senior-Entwickler mit über 15 Jahren Erfahrung waren nicht vertreten. Eine Übertragbarkeit auf die Gesamtheit der professionellen Softwareentwickler ist daher nur eingeschränkt möglich.

- **Methodischer Kontext:** Die Befragung fand in einem künstlichen Setting mittels LimeSurvey statt. Das reale Verhalten unter Zeitdruck oder in hochkomplexen industriellen Projekten könnte von den hier erhobenen, eher reflexiven Selbsteinschätzungen abweichen.

7. Fazit und Ausblick

Die vorliegende Untersuchung konnte zeigen, dass Programmierstrategien weit mehr sind als nur technisches Handwerkszeug Sie sind der sichtbare Ausdruck tiefliegender kognitiver Problemlöseprozesse. Die Dekomposition hat sich dabei als der "Königsweg" der Programmierung bestätigt. Sie ist die am häufigsten genutzte Strategie und korreliert am stärksten mit Projektroutine und Erfahrung.

Ein überraschendes Ergebnis war die hohe Akzeptanz von LLM-unterstützter Codegenerierung, die sich bereits auf Augenhöhe mit klassischen analytischen Methoden etabliert hat. Die Studie verdeutlicht zudem den "Experten-Shift". Mit steigender Erfahrung wandelt sich das Vorgehen von einer opportunistischen Code-Manipulation hin zu einer schemagesteuerten, systematischen Analyse. Der Compiler fungiert dabei für Experten nicht mehr nur als Fehlermelder, sondern als direktives Werkzeug zur Validierung mentaler Modelle.

7.1. Kritische Würdigung und Beitrag

Trotz der wertvollen Einblicke zeigt die Arbeit auch methodische Herausforderungen auf. Während operative Strategien (wie das Testen oder Planen) empirisch stabil zugeordnet werden konnten, erwiesen sich abstrakte Konzepte wie das Analogiedenken als schwieriger messbar. Die niedrigen Reliabilitätswerte in diesen Kategorien deuten darauf hin, dass die menschliche Intuition beim Programmieren komplexer vernetzt ist, als es einfache Item-Mappings erfassen können. Dennoch leistet die Arbeit einen wichtigen Beitrag, indem sie eine Brücke zwischen der Psychologie und der Informatik schlägt und ein Instrumentarium zur Reflexion der eigenen strategischen Kompetenz liefert.

7.2. Ausblick

Für zukünftige Forschungsvorhaben wäre es vielversprechend, die hier erhobenen Selbsteinschätzungen durch objektive Verhaltensdaten (z. B. Eyetracking) zu ergänzen. Zudem stellt sich die Frage, wie die wachsende Rolle von KI-Assistenten die langfristige Entwicklung metakognitiver Strategien bei Anfängern beeinflusst. Werden LLMs das Erlernen systematischer Zerlegung fördern oder eher zu einer kognitiven Abhängigkeit führen? Die hier entwickelten Mappings bieten die Basis, um diese Entwicklungen in einer sich rasant verändernden Programmierwelt weiter zu beobachten.

Literaturverzeichnis

[1] Alpuente, M., Ballis, D., Frechina, F., Romero, D.: Using conditional trace slicing for improving maude programs 80, 385–415, https://www.sciencedirect.com/science/article/pii/S0167642313002505

[2] Arab, M., LaToza, T.D., Ko, A.J.: An exploratory study of writing and revising explicit programming strategies, http://arxiv.org/abs/2004.00701

[3] Arab, M., LaToza, T.D., Liang, J., Ko, A.J.: An exploratory study of sharing strategic programming knowledge. In: Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems. pp. 1–15. CHI '22, Association for Computing Machinery, https://dl.acm.org/doi/10.1145/3491102.3502070

[4] Arab, M., Liang, J., Yoo, Y., Ko, A.J., LaToza, T.D.: HowToo: A platform for sharing, finding, and using programming strategies. In: 2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). pp. 1–9. https://ieeexplore.ieee.org/document/9576337, ISSN: 1943-6106

[5] Baltes, S., Diehl, S.: Towards a theory of software development expertise. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 187–200. ESEC/FSE 2018, Association for Computing Machinery, https://dl.acm.org/doi/10.1145/3236024.3236061

[6] Beck, K.: Test Driven Development: By Example. Addison-Wesley Professional, google-Books-ID: zNnPEAAAQBAJ

[7] Begum, M., Nørbjerg, J., Clemmensen, T.: NOVICE PROGRAMMING STRATEGIES https://aisel.aisnet.org/siged2018/25

[8] Borchers, J.O.: A pattern approach to interaction design. In: Proceedings of the 3rd conference on Designing interactive systems: processes, practices, methods, and techniques. pp. 369–378. DIS '00, Association for Computing Machinery, https://dl.acm.org/doi/10.1145/347642.347795

[9] Böhme, M., Soremekun, E.O., Chattopadhyay, S., Ugherughe, E.J., Zeller, A.: How developers debug software — the DBGBENCH dataset. In: 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C). pp. 244–246. https://ieeexplore.ieee.org/document/7965318

[10] Chi, M.T.: PROBLEM SOLVING ABILITIES (1983)

[11] Dorsch: Lexikon der psychologie, stichwort: Strategie, https://dorsch.hogrefe.com/stichwort/strategie, zugriff am 02.03.2026

[12] Fitzgerald, S.: Strategies that students use to trace code | proceedings of the first international workshop on computing education research, https://dl.acm.org/doi/10.1145/1089786.1089793

[13] Galindo, C., Pérez, S., Silva, J.: Program slicing of java programs 130, 100826, https://www.sciencedirect.com/science/article/pii/S2352220822000797

[14] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: Elements of reusable object-oriented software

[15] Gick, M.L.: Problem-Solving Strategies (1986)

[16] Greeno, J.G., Simon, H.A.: PROBLEM SOLVING AND REASONING (1988)

[17] Jonassen, D.H.: Toward a Design Theory of Problem Solving (2000)

[18] Jungermann, H., Pfister, H.R., Fischer, K.: Die Psychologie der Entscheidung. Spektrum, Heidelberg (2010)

[19] Kesselbacher, M., Bollin, A.: Discriminating programming strategies in scratch: Making the difference between novice and experienced programmers. In: Proceedings of the 14th Workshop in Primary and Secondary Computing Education. pp. 1–10. WiPSCE '19, Association for Computing Machinery, https://dl.acm.org/doi/10.1145/3361721.3361727

[20] Ko, A.J.: Teaching explicit programming strategies to adolescents | proceedings of the 50th ACM technical symposium on computer science education, https://dl.acm.org/doi/10.1145/3287324.3287371

[21] Ko, A.J., Myers, B.A.: Extracting and answering why and why not questions about java program output 20(2), 4:1–4:36, https://dl.acm.org/doi/10.1145/1824760.1824761

[22] LaToza, T.D., Arab, M., Loksa, D., Ko, A.J.: Explicit programming strategies 25(4), 2416–2449, https://doi.org/10.1007/s10664-020-09810-1

[23] Lawrance, J., Bogart, C., Burnett, M., Bellamy, R., Rector, K., Fleming, S.D.: How programmers debug, revisited: An information foraging theory perspective 39(2), 197–215, https://ieeexplore.ieee.org/document/5674060

[24] Lubin, J., Chasins, S.E.: How statically-typed functional programmers write code 5, 155:1–155:30, https://dl.acm.org/doi/10.1145/3485532

[25] Mayer, R.E.: Thinking, Problem Solving, Cognition. W. H. Freeman, New York (1983)

[26] McCartney, R., Eckerdal, A., Mostrom, J.E., Sanders, K., Zander, C.: Successful students' strategies for getting unstuck. In: Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education. pp. 156–160. ITiCSE '07, Association for Computing Machinery, `https://dl.acm.org/doi/10.1145/1268784.1268831`

[27] Mead, J., Gray, S., Hamer, J., James, R., Sorva, J., Clair, C.S., Thomas, L.: A cognitive approach to identifying measurable milestones for programming skill acquisition 38(4), 182–194, `https://dl.acm.org/doi/10.1145/1189136.1189185`

[28] Parnin, C., Orso, A.: Are automated debugging techniques actually helping programmers? In: Proceedings of the 2011 International Symposium on Software Testing and Analysis. pp. 199–209. ISSTA '11, Association for Computing Machinery, `https://dl.acm.org/doi/10.1145/2001420.2001445`

[29] Pedrosa, D., Cravino, J., Morgado, L., Barreira, C.: Self-regulated learning in computer programming: Strategies students adopted during an assignment. In: Allison, C., Morgado, L., Pirker, J., Beck, D., Richter, J., Gütl, C. (eds.) Immersive Learning Research Network. pp. 87–101. Springer International Publishing

[30] Pikkarainen, M., Haikara, J., Salo, O., Abrahamsson, P., Still, J.: The impact of agile practices on communication in software development 13(3), 303–337, `https://doi.org/10.1007/s10664-008-9065-9`

[31] Poppendieck, M.: Lean software development. In: 29th International Conference on Software Engineering (ICSE'07 Companion). pp. 165–166. `https://ieeexplore.ieee.org/document/4222727`

[32] de Raadt, M.: Teaching programming strategies explicitly to novice programmers, `https://research.usq.edu.au/item/9yy76/teaching-programming-strategies-explicitly-to-novice-programmers`

[33] Roehm, T., Tiarks, R., Koschke, R., Maalej, W.: How do professional developers comprehend software? In: 2012 34th International Conference on Software Engineering (ICSE). pp. 255–265. `https://ieeexplore.ieee.org/document/6227188`, ISSN: 1558-1225

[34] Schmid, U.: Computermodelle des problemlösens. In: Müsseler, J. (ed.) Allgemeine Psychologie, pp. 601–630. Spektrum, Heidelberg/Berlin, 3. auflage edn. (2017)

[35] Sedgwick, P.: Pearson's correlation coefficient (2012), https://doi.org/10.1136/bmj.e4483

[36] Sharma, K., Mangaroska, K., Trætteberg, H., Lee-Cultura, S., Giannakos, M.: Evidence for programming strategies in university coding exercises. In: Pammer-Schindler, V., Pérez-Sanagustín, M., Drachsler, H., Elferink, R., Scheffel, M. (eds.) Lifelong Technology-Enhanced Learning. pp. 326–339. Springer International Publishing

[37] Tavakol, M., Reg, D.: Making sense of cronbach's alpha (2011), https://doi.org/10.5116/ijme.4dfb.8dfd

[38] Wang, T., Zhou, N., Chen, Z.: Enhancing computer programming education with LLMs: A study on effective prompt engineering for python code generation, http://arxiv.org/abs/2407.05437

[39] Wong, W.E., Gao, R., Li, Y., Abreu, R., Wotawa, F.: A survey on software fault localization 42(8), 707–740, https://ieeexplore.ieee.org/document/7390282

[40] Xie, B., Nelson, G.L., Ko, A.J.: An explicit strategy to scaffold novice program tracing. In: Proceedings of the 49th ACM Technical Symposium on Computer Science Education. pp. 344–349. SIGCSE '18, Association for Computing Machinery, https://dl.acm.org/doi/10.1145/3159450.3159527

[41] Xie, X., Liu, Z., Song, S., Chen, Z., Xuan, J., Xu, B.: Revisit of automatic debugging via human focus-tracking analysis. In: Proceedings of the 38th International Conference on Software Engineering. pp. 808–819. ICSE '16, Association for Computing Machinery, https://dl.acm.org/doi/10.1145/2884781.2884834

[42] Zar, J.H.: Spearman rank correlation". in encyclopedia of biostatistics (2005), https://doi.org/10.1002/0470011815.b2a15150

A. Tabellen

Table A.1.: Deskriptive Statistik der Fragebogen-Items

Item	count	mean	std	min	25%	50%	75%	max
D7	31	4.23	1.15	0	4	5	5	5
B11	31	4.13	1.09	0	4	4	5	5
A2	31	3.97	1.11	0	4	4	5	5
B3	31	3.97	1.30	0	4	4	5	5
P5	30	3.97	1.22	1	3.25	4	5	5
P11	31	3.94	1.29	0	3	4	5	5
P2	31	3.94	0.93	1	3	4	5	5
D10	31	3.87	1.09	1	3	4	5	5
P3	31	3.81	0.83	2	3	4	4	5
P10	31	3.81	1.05	2	3	4	5	5
P4	30	3.80	1.03	1	3	4	4.75	5
A9	31	3.74	1.34	0	3	4	5	5
P8	31	3.74	1.03	1	3	4	4.50	5
A3	31	3.74	1.29	0	3	4	5	5
B4	31	3.71	1.35	0	3	4	5	5
B5	30	3.67	1.58	0	3	4	5	5
A4	31	3.65	1.05	1	3	4	4	5
D8	31	3.65	1.14	0	3	4	4	5
B10	30	3.63	1.33	0	3	4	4	5
A10	31	3.61	1.23	1	3	4	5	5
D9	31	3.61	1.09	1	3	4	4.50	5
B1	31	3.58	1.29	1	3	4	5	5
A5	31	3.55	1.31	0	3	4	4.50	5
P1	31	3.55	1.09	0	3	4	4	5
A8	31	3.48	1.23	1	2.50	4	4	5

Fortsetzung auf der nächsten Seite

Tabelle A.1 fortgesetzt

Item	count	mean	std	min	25%	50%	75%	max
B7	30	3.47	1.20	0	3	4	4	5
B8	30	3.47	1.22	1	3	4	4	5
P9	31	3.45	1.26	1	2	4	4	5
A6	31	3.45	1.41	0	3	4	4	5
P6	31	3.42	1.23	1	3	3	4	5
B2	31	3.35	1.52	0	3	4	4	5
A7	31	3.32	1.30	0	3	4	4	5
B12	31	3.32	1.25	1	3	4	4	5
D14	31	3.32	1.42	0	2.50	4	4	5
D4	31	3.29	1.22	1	2.50	3	4	5
D2	30	3.20	1.32	1	2	3	4	5
D13	31	3.19	1.38	0	3	3	4	5
A1	31	3.16	1.24	0	3	3	4	5
D5	31	3.13	1.59	0	2.50	4	4	5
D11	31	3.10	1.27	0	2.50	3	4	5
B9	31	3.06	1.46	0	2	3	4	5
P7	31	2.97	1.08	1	2	3	4	5
B6	31	2.97	1.25	1	2	3	4	5
D1	31	2.94	1.65	0	1.50	3	4.50	5
B14	31	2.90	1.42	0	2	3	4	5
D3	31	2.84	1.61	0	1.50	3	4	5
D6	31	2.81	1.58	0	2	3	4	5
B13	31	2.81	1.49	0	2	3	4	5
D12	31	2.29	1.01	1	1.50	2	3	4

B. Github Link

https://github.com/fepo172/Masterarbeit