Adaption von etablierten Qualitätssicherungsmethoden in der Programmierung von Low-Code und No-Code-Anwendungen

Fokus Quellcode-Analyse, Performance-Analyse und Testing mit dem modellbasierten Low-Code und No-Code-Framework Posity

Masterarbeit

im Studiengang Wirtschaftsinformatik

Vorgelegt von

Matthias Christen
Matr.-Nr.: 06-527-154
[redacted]

und

Marion Mürner
Matr.-Nr.: 06-530-471
[redacted]

am

28. Mai 2022

an der ZHAW School of Management and Law

Betreut von

Adrian Moser
(Co-Betreuer: Max Meisterhans)

Management Summary

Low-Code und No-Code (LCNC) ist ein neuer, schnellwachsender Ansatz zur Entwicklung von Software-Anwendungen. Der Ansatz von LCNC schliesst die Lücke zwischen IT und Business und ermöglicht nicht-technischen Fachexperten, sich aktiv am Entwicklungsprozess zu beteiligen und komplette Anwendungen zu erstellen. Die Entwicklungsumgebungen verbergen komplexe Abläufe vor den Programmierenden, indem sie vorgefertigte Softwarebausteine im Baukastenformat anbieten. Mit LCNC werden neue Konzepte und Eigenschaften eingeführt. Allerdings ist in der Wissenschaft noch nicht viel zu Herausforderungen und Methoden der Qualitätssicherung in der Programmierung von LCNC-Anwendungen untersucht worden. Diese Forschungslücke motiviert diese Arbeit dazu, ausgewählte Qualitätssicherungsmethoden aus der traditionellen Softwareentwicklung (Performance-Analyse, Quellcode-Analyse und Testing) auf deren Eignung zur Qualitätssicherung für LCNC zu prüfen.

Initial wird in dieser Arbeit die Wissensbasis der Forschung durch eine Literaturrecherche analysiert und die Probleme und Anforderungen der Anwendungsdomäne anhand der Durchführung einer Fokus-Gruppe mit Vertretern aus der Praxis und der Forschung identifiziert. Basierend auf dem Stand der Forschung und den identifizierten Anforderungen wird für die Performance-Analyse und die Quellcode-Analyse je ein Software-Artefakt entwickelt und für das Testing werden zwei konzeptionelle Artefakte für unterschiedliche Testverfahren, im Kontext der kommerziellen LCNC-Entwicklungsplattform Posity, als Stellvertreter der LCNC-Plattformen, erstellt. Das Resultat beurteilt die erstellten Artefakte anhand der, in der Analyse, ermittelten Erfolgskriterien, bewertet die Eignung der untersuchten Methoden in der Anwendung für LCNC-Programmierung und formuliert Handlungsempfehlungen für Praxis und Forschung.

Die Validierung aller drei untersuchten Qualitätssicherungsmethoden (Performance-Analyse, Quellcode-Analyse und Testing) haben ergeben, dass eine Verwendung für LCNC-Programmierung einerseits möglich ist und andererseits aus den Analyse- und Test-Resultaten qualitätssichernde Massnahmen abgeleitet werden könnten.

Die Allgemeingültigkeit der Ergebnisse sind insofern limitiert, als dass sie im Kontext von nur einer Plattform, stellvertretend für alle LCNC-Plattformen, validiert worden sind. Die Erkenntnisse lassen jedoch folgende generalisierbare Schlussfolgerung zu. Um eine

Analyse, auf eine in LCNC-Code erstellten Anwendung, ausführen zu können, ist es unumgänglich, dass die Plattformhersteller entsprechende Werkzeuge anbieten und dass sich das Ergebnis einer Analyse auf der gleichen Abstraktionsstufe befindet, wie der «Code» selbst. Nur so kann der Entwickler Analysen und Tests erstellen, die Resultate einordnen und entsprechende Massnahmen ergreifen, ohne dass tiefergehende Programmierkenntnisse notwendig sind. Die Vielfalt der Plattformen und das Fehlen von Standards, hat zur Folge, dass jeder Hersteller eine eigene Lösung für die Integration dieser Techniken bauen muss.

Quantitative Performance-Messgrössen, wie die Ausführungszeit, konnten in verschiedenen Bereichen einer LCNC-Anwendung erfasst werden. Das Identifizieren von Performance-Problemen bedingt eine geeignete Visualisierung. Die in der Softwareentwicklung bekannte Darstellungsform Call-Tree wurde dafür als äusserst geeignet beurteilt. Ein Anknüpfungspunkt für weitere Forschung wäre die Untersuchung von Methoden oder Verfahren, die für Performance-Probleme Lösungsvorschläge bieten, oder diese sogar automatisiert beheben könnten.

Für die Quell-Codeanalyse im Speziellen, ist die Eignung der Methode wesentlich von den zur Verfügung stehenden Regeln und Metriken abhängig. In dieser Arbeit wurden ausgewählte Regeln, aus der traditionellen Programmierung, untersucht und implementiert. Einen ausgereiften und allgemeingültigen Satz an Regeln gibt es bis anhin nicht und ist Gegenstand weiterer Forschungsarbeit.

Durch Testen können völlig unerwartete Fehler aufgedeckt werden, die nur durch das Ausführen der Programme erkennbar werden. Deshalb ist das Testing eine wichtige Methode zur Qualitätssicherung. Die Arbeit analysiert spezifikationsorientierte und diversifizierende Testverfahren, als zwei verbreitete Testtechniken. Beide Testverfahren wurden als gleichermassen geeignet für den Einsatz in der LCNC-Programmierung validiert. Die Ergebnisse dieser Forschung können als Grundlage für die Implementierung der Qualitätssicherungsmethoden für andere Plattform-Hersteller und als Leitfaden für die Untersuchung und Entwicklung weiterer Methoden für die Forschung verwendet werden.

Inhaltsverzeichnis

Management Summary **II**

Inhaltsverzeichnis **IV**

Abbildungsverzeichnis **VIII**

Tabellenverzeichnis **XII**

Abkürzungsverzeichnis **XIII**

Glossar **XIV**

Ehrenwörtliche Erklärung **XV**

Vorwort / Danksagung **XVI**

1 Einleitung **1**

1.1 Ausgangslage 1

1.2 Forschungslücke 2

1.3 Forschungsfragen 2

1.4 Abgrenzungen 3

1.5 Beitrag der Arbeit 4

1.6 Relevanz 4

1.7 Aufteilung der Arbeit und Zuständigkeiten 4

2 Related Work **6**

2.1 Qualitätssicherungsmassnahmen Performance-Analyse und spezifikationsorientiertes Testverfahren 7

2.2 Qualitätssicherungsmassnahmen Quellcode-Analyse und diversifizierende Testverfahren 8

3 Vorgehen und Methoden **11**

4 Analyse **15**

4.1 Stakeholder (involvierte Akteure) 16

4.2 Organisationale Strukturen 17

4.3 Technische Strukturen 17

5 Artefakt DevTools **20**

5.1 Spezifikation 20

5.2 Implementation 21

6 Artefakt Performance-Analyse **24**

6.1 Anwendungskontext 24

6.1.1 Datenbank-Abfragen 25
6.1.2 Applikationslogik 27
6.2 Anforderungen 31
6.2.1 Strukturelle Anforderungen 31
6.2.2 Nicht Funktionale Anforderungen 31
6.2.3 Funktionale Anforderungen 32
6.2.4 Abgrenzungen 32
6.3 Erfolgskriterien 33
6.4 Spezifikation 34
6.4.1 Datenbank-Abfragen 34
6.4.2 Applikationslogik 40
6.5 Implementation 49
6.5.1 Datenbank-Abfragen 50
6.5.2 Applikationslogik 55
6.6 Validierung 64
6.6.1 Geeignete Applikationen bestimmen 64
6.6.2 Kundenapplikation Rezeptparameter Verwaltung für Kabelfabrikationsanlagen (HS Propara) 65
6.6.3 Kundenapplikation Kundenofferte Verwaltung für Solaranlagen (Sol ERP) 65
6.6.4 Bewertung der Erfolgskriterien 68

7 Artefakt Quellcode-Analyse – Posity Linter 77

7.1 Anwendungskontext 77
7.1.1 Programmieren mit Posity Design Studio 77
7.1.2 Bestehende Qualitätskontrollen im Posity Design Studio 96
7.1.3 Elemente von Interesse für die Quellcode-Analyse 98
7.1.4 Posity Shape 100
7.2 Anforderungen 102
7.2.1 Strukturelle Anforderungen 102
7.2.2 Nicht Funktionale Anforderungen 102
7.2.3 Funktionale Anforderungen 102
7.2.4 Abgrenzung 103
7.3 Erfolgskriterien 104
7.4 Spezifikation 105
7.4.1 Bestandteile der Quellcode-Analyse 106
7.4.2 Posity Linter 107
7.4.3 Regeln und Metriken 111
7.4.4 Posity-Shapes und Anwendung von Regeln (Mapping) 124
7.4.5 Issue 125
7.4.6 Qualitätsprofil 126
7.4.7 Ignore-List 126

7.5 Implementation 128
7.5.1 Ablauf, Struktur und Klassenhierarchie 129
7.5.2 Posity Linter 133
7.5.3 Regeln und Metriken 137
7.5.4 Posity-Shapes und Anwendung von Regeln (Mapping) 151
7.5.5 Issue 152
7.5.6 Ignore-List 153
7.6 Validierung 155
7.6.1 Geeignete Posity-Applikationen bestimmen 155
7.6.2 Kundenapplikation ERP für ein Unternehmen in der Solarbranche 156
7.6.3 Kundenapplikation Normenhilfe für die Norm EN378 für die Kältemittelbranche (Schweizerischer Verband für Kältetechnik) 160
7.6.4 Bewertung der Erfolgskriterien 164

8 Konzept für spezifikationsorientierte Testverfahren (Blackbox-Testing) 177
8.1 Anwendungskontext 177
8.1.1 Testschritte 178
8.1.2 Testverfahren 178
8.1.3 Teststufen 180
8.1.4 Kandidaten für Blackbox-Testing in Posity Design Studio 180
8.2 Anforderungen 180
8.2.1 Strukturelle Anforderungen 181
8.2.2 Nicht Funktionale Anforderungen 181
8.2.3 Funktionale Anforderungen 181
8.2.4 Abgrenzungen 182
8.3 Erfolgskriterien 182
8.4 Spezifikation 183
8.4.1 Testfälle und Test-Szenarien definieren 183
8.4.2 Testfall Modul-Diagramm 184
8.4.3 Testfall Query-Diagramm 185
8.4.4 Testbedingungen definieren 185
8.4.5 Stellvertreter (Mocks) definieren 186
8.4.6 Datenbank-Mock 186
8.4.7 Query-Mock 187
8.4.8 Posity-GUI-Mock 188
8.4.9 Testfälle ausführen 188
8.4.10 Test-Diagramm Ausführung 189
8.4.11 Testresultate visualisieren 189
8.5 Validierung 189

9 Konzept für diversifizierende Testverfahren (Regressionstest) 195
9.1 Anwendungskontext 196

9.1.1 Regressionstests 196
9.1.2 Posity Diagramme und Regressionstest 197
9.2 Anforderungen 198
9.2.1 Strukturelle Anforderungen 198
9.2.2 Nicht Funktionale Anforderungen 198
9.2.3 Funktionale Anforderungen 199
9.2.4 Abgrenzungen 199
9.3 Erfolgskriterien 199
9.4 Spezifikation 200
9.4.1 Aufzeichnen und von Testfällen im Posity Design Studio 201
9.4.2 Ausführen von Testfällen im Posity Design Studio 203
9.4.3 Testergebnisse und Validierung 205
9.5 Validierung 206

10 Schlussfolgerungen und Handlungsempfehlungen 211

10.1 Qualitätssicherungsmassnahmen Performance-Analyse und spezifikationsorientiertes Testverfahren 211
10.1.1 Performance-Analyse 211
10.1.2 Spezifikationsorientierte Verfahren – Blackbox-Testing 214
10.2 Qualitätssicherungsmassnahmen Quellcode-Analyse und diversifizierende Testverfahren 216

11 Fazit 221

12 Literaturverzeichnis 222

Anhang i

A Kapitelübersicht Autoren i
B SQL-Stored Procedure für Query-Diagramm: Produkt auswählen ii
C SQL-Stored Procedure für System View sys. dm_exec_procedure_stats Abfragen vi
D Listening GetTraceTree Methode ix
E Fokus-Gruppe Miro Board xii

Abbildungsverzeichnis

Abbildung 1: Forschungsmodell nach Peffers et al. 11
Abbildung 2: Forschungszyklen nach Hevner 13
Abbildung 3: Resultat Mentimeter-Umfrage Einstufung Codequalitätskriterien (Quelle: Eigene Darstellung aus Mentimeter) 15
Abbildung 4: Resultat Mentimeter-Umfrage Einstufung Performance-Messgrössen (Quelle: Eigene Darstellung aus Mentimeter) 16
Abbildung 5: Resultat Mentimeter-Umfrage Einstufung Testtypen (Quelle: Eigene Darstellung aus Mentimeter) 16
Abbildung 6: Posity Diagramme und Abhängigkeiten (Quelle: interne Dokumentation Posity AG) 18
Abbildung 7: Posity Diagrammtypen und Abhängigkeiten (Quelle: interne Dokumentation Posity AG) 19
Abbildung 8: Web-Developer-Tools im Webbrowser Microsoft Edge mit Ansicht zu Hinweisen zum HTML-Code der angezeigten Webseite (Quelle: Eigene Darstellung) 21
Abbildung 9: UML Klassendiagramm Artefakt DevTools 22
Abbildung 10: Artefakt Posity DevTools (Quelle: Eigene Darstellung) 22
Abbildung 11: Möglichkeiten für das Öffnen der DevTools (Quelle: Eigene Darstellung) 23
Abbildung 12: Beispiel Tabelle Projekt im Datenmodell (Quelle: Eigene Darstellung) 25
Abbildung 13: Beispiel Query-Diagramm (Quelle: Eigene Darstellung) 26
Abbildung 14: Beispiel Aufruf der Query-Diagramm Stored Procedure (Quelle: Eigene Darstellung) 27
Abbildung 15: Beispiel Module-Diagramm Mehrwertsteuer berechnen (Quelle: Eigene Darstellung) 27
Abbildung 16: Modul-Diagramm Kontrollstrukturen (Quelle: Eigene Darstellung) 28
Abbildung 17: Zusammenhang Posity-Prozesse, Module und Posity Virtual Machines (Quelle: Eigene Darstellung) 31
Abbildung 18: Kontextdiagramm Datenbank-Abfragen in Posity (Quelle: Eigene Darstellung) 34
Abbildung 19: Einflussstellen der Faktoren der Datenbank-Abfragen Performance (Quelle: Eigene Darstellung) 35
Abbildung 20: Microsoft SQL-Server Query Store (Quelle: Eigene Darstellung) 39
Abbildung 21: Benutzerprozesse und PVM Betriebssystem-Threads (Quelle: Eigene Darstellung) 41
Abbildung 22: Beispiel Flame Graph (Quelle: Brendan's site: Flame Graphs, https://www.brendangregg.com/flamegraphs.html, Abgerufen: 10.05.2022) 47
Abbildung 23: Beispiel Flame Graph von Posity.com im Webbrowser Edge (Quelle: Eigene Darstellung) 48
Abbildung 24: Call-Tree von Call-Stack vom Posity Login Vorgang (Quelle: Eigene Darstellung) 49
Abbildung 25: Benutzeroberfläche Query Performance DevTool (Quelle: Eigene Darstellung) 52

Abbildung 26: UML Klassendiagramm der Datenstrukturen für die Erfassung von Performance-Messdaten der PVM (Quelle: Eigene Darstellung) 57
Abbildung 27: Abbildung Funktionsbausteine nach Performance-Messdaten-Baumstruktur (Quelle: Eigene Darstellung) 58
Abbildung 28: Abbildung Strukturbaustein Switch-Case nach Performance-Messdaten-Baumstruktur (Quelle: Eigene Darstellung) 59
Abbildung 29: Abbildung Strukturbaustein For nach aggregierte Performance-Messdaten-Baumstruktur (Quelle: Eigene Darstellung) 59
Abbildung 30: UML Klassendiagramm TraceTree (Quelle: Eigene Darstellung) 60
Abbildung 31: Call-Tree von Modul-Diagramm (Quelle: Eigene Darstellung) 62
Abbildung 32: Microsoft SQL-Server User defined Table Type RuntimeExecutedComponent (Quelle: Eigene Darstellung) 63
Abbildung 33: Benutzeroberfläche Artefakt Performance Applikationslogik (Quelle: Eigene Darstellung) 64
Abbildung 34: Rezeptverwaltung Datenbank-Abfragen Performance-Messdaten (Quelle: Eigene Darstellung) 65
Abbildung 35: Datenbank-Abfragen Performance-Messdaten der Kundenofferten-Verwaltung 66
Abbildung 36: Call-Tree der Performance-Messdaten Kundenofferte anzeigen (Quelle: Eigene Darstellung) 67
Abbildung 37: Call-Tree der Performance-Messdaten der Applikationslogik für Kundenofferte Dokument generieren (Quelle: Eigene Darstellung) 67
Abbildung 38: Posity Diagramme und Verwendung in Analyse - grün ist Teil der Analyse (Quelle: Eigene Darstellung) 78
Abbildung 39: Übersicht Entwicklungsumgebung Posity Design Studio (Quelle: Eigene Darstellung) 80
Abbildung 40: Tabellen-Diagramm mit Tabellen einer Applikation aufgelistet (Quelle: Eigene Darstellung) 81
Abbildung 41: Elemente für die Modellierung des Diagrammtyps Datenmodell (Quelle: Eigene Darstellung) 81
Abbildung 42: Teil eines modellierten Tabellen-Diagramms (Quelle: Eigene Darstellung) 82
Abbildung 43: Tabelle und wichtige Bestandteile (Quelle: Eigene Darstellung) 82
Abbildung 44: Query-Diagramme einer Applikation aufgelistet (Quelle: Eigene Darstellung) 84
Abbildung 45: Elemente für die Modellierung von Diagrammtyp Query (Quelle: Eigene Darstellung) 84
Abbildung 46: Modelliertes Query-Diagramm mit diversen Elementen (Quelle: Eigene Darstellung) 84
Abbildung 47: Query-Diagramm und wichtige Bestandteile (Quelle: Eigene Darstellung) 85
Abbildung 48: GUI-Diagramme einer Applikation aufgelistet (Quelle: Eigene Darstellung) 87
Abbildung 49: Shapes für die Modellierung von Diagrammtyp GUI (Quelle: Eigene Darstellung) 87

Abbildung 50: Modelliertes GUI-Diagramm mit diversen Shapes (Quelle: Eigene Darstellung) 87
Abbildung 51: GUI-Diagramm und wichtige Bestandteile (Quelle: Eigene Darstellung) 88
Abbildung 52: Beispiel Event "Read" in Modul "Typ-Liste" (Quelle: Eigene Darstellung) 89
Abbildung 53: Beispiel von zwei Modul-Funktionen (Quelle: Eigene Darstellung) 90
Abbildung 54: Ausschnitt der zur Verfügung stehenden Modul-Funktionen (Quelle: Eigene Darstellung) 90
Abbildung 55: Beispiel eines Datenflusses im Modul (Quelle: Eigene Darstellung) 91
Abbildung 56: Beispiel der Verwendung von Konstanten in Modul (Quelle: Eigene Darstellung) 92
Abbildung 57: Beispiel der Verwendung von Parameter in Modul (Quelle: Eigene Darstellung) 92
Abbildung 58: Aufrufen von Ereignissen, Modulen oder Prozessen (Quelle: Eigene Darstellung) 93
Abbildung 59: Ein Beispiel von einem if-else-Case (Quelle: Eigene Darstellung) 93
Abbildung 60: Beispiel For-Schleife in Modul (Quelle: Eigene Darstellung) 94
Abbildung 61: Beispiel einer For-Each-Schleife in Modul (Quelle: Eigene Darstellung) 94
Abbildung 62: Beispiel einer Sequenz in Modul (Quelle: Eigene Darstellung) 95
Abbildung 63: Beispiel try-catch im Modul (Quelle: Eigene Darstellung) 96
Abbildung 64: Error-Checker in Diagrammen (Quelle: Eigene Darstellung) 97
Abbildung 65: Shape und Sub-Shape (Quelle: Eigene Darstellung) 100
Abbildung 66: Auszug der Klassenhierarchie Shape und SubShape (Quelle: Eigene Darstellung) 101
Abbildung 67: Mockup Benutzerinterface Posity Linter Analyse (Quelle: Eigene Darstellung) 108
Abbildung 68: Mockup Benutzerinterface Posity Linter Ignore-Liste (Quelle: Eigene Darstellung) 109
Abbildung 69: Ausrichtung von Verbindungslinien von Ports (Quelle: Eigene Darstellung) 119
Abbildung 70: Zuordnung der Severity für eine Regel (Quelle: (SonarQube, 2022)) 121
Abbildung 71: Metadaten von einem Issue (Quelle: Eigene Darstellung) 126
Abbildung 72: Bestandteile Posity-Framework (Quelle: Eigene Darstellung) 128
Abbildung 73: Sub-Projektstruktur und Verortung im Hauptprojekt PosityApp (Quelle: Eigene Darstellung) 129
Abbildung 74: Klassendiagramm Posity Linter und Abhängigkeiten (Quelle: Eigene Darstellung) 131
Abbildung 75: Klassendiagramm Posity Linter (nur Posity Linter) (Quelle: Eigene Darstellung) 132
Abbildung 76: Klassendiagramm Posity Linter - nur IDENamespace (Quelle: Eigene Darstellung) 133
Abbildung 77: Klassendiagramm Posity Linter und DevTools (Quelle: Eigene Darstellung) 134
Abbildung 78: Posity Linter starten (Quelle: Eigene Darstellung) 135

Abbildung 79: Benutzeroberfläche Posity Linter – Tab «Analyse» (Quelle: Eigene Darstellung) 136
Abbildung 80: Benutzeroberfläche Posity Linter - Tab Ignore list (Quelle: Eigene Darstellung) 137
Abbildung 81: Klassendiagramm Regeln (Quelle: Eigene Darstellung) 140
Abbildung 82: Klassendiagramm ILintCandidates (Quelle: Eigene Darstellung) 140
Abbildung 83: Posity-Shapes und ILintCandidates Interfaces (Quelle: Eigene Darstellung) 141
Abbildung 84: Datenmodell Regel und Metadaten (Quelle: Eigene Darstellung) 142
Abbildung 85: Klassendiagramm Regel und Metadaten (Quelle: Eigene Darstellung) 151
Abbildung 86: Klassendiagramm Ausschnitt Mapping Regel und Shape (Quelle: Eigene Darstellung) 152
Abbildung 87: Klassendiagramm Ausschnitt Issue (Quelle: Eigene Darstellung) 153
Abbildung 88: Klassendiagramm Ausschnitt Posity Linter und Ignore-Liste (Quelle: Eigene Darstellung) 154
Abbildung 89: Datenmodell Ignore-Liste (Quelle: Eigene Darstellung) 155
Abbildung 90: Resultat Analyse GUI Kundenofferte SolarvilleERP (Quelle: Eigene Darstellung) 159
Abbildung 91: Resultat Analyse Modul Kundenofferte SolarvilleERP (Quelle: Eigene Darstellung) 159
Abbildung 92: Resultat Analyse Query Kundenofferte SolarvilleERP (Quelle: Eigene Darstellung) 160
Abbildung 93: Resultat Analyse Datenmodell (Tabellen) SolarvilleERP (Quelle: Eigene Darstellung) 160
Abbildung 94: Resultat Analyse EN378-Tool (Quelle: Eigene Darstellung) 163
Abbildung 95: Übersicht Bestandteile eines Testrahmens (Quelle: (Andreas & Tilo, 2012, S. 109)) 178
Abbildung 96: Testfälle und Test-Szenarien definieren für Query- und Modul-Diagramm (Quelle: Eigene Darstellung) 184
Abbildung 97: Test-Diagramm für Modul-Diagramm Testfall Kundenofferte speichern (Quelle: Eigene Darstellung) 185
Abbildung 98: Test-Diagramm für Query-Diagramm Testfall Kundenofferte öffnen (Quelle: Eigene Darstellung) 185
Abbildung 99: Query Mock-Daten mit Debugger speichern (Eigene Darstellung) 187
Abbildung 100: Testfall für die Ausführung selektieren 188
Abbildung 101: Testresultate visualisieren – Mockup Benutzeroberfläche Testtreiber 189
Abbildung 102: Prinzip des Regressionstest (Quelle: Eigene Darstellung angelehnt an (Liggesmeyer, 2009, S. 194)) 198
Abbildung 103: Modul-Diagramm Markierung von Testpunkten und Mock-Objekten (Quelle: Eigene Darstellung) 202
Abbildung 104: Mockup Starten einer Testaufzeichnung (Quelle: Eigene Darstellung) 202
Abbildung 105: Mockup Visualisierung Testtreiber (Quelle: Eigene Darstellung) 203
Abbildung 106: Modul-Diagramm Testfall mit deaktiviertem Testpunkt (Quelle: Eigene Darstellung) 204

Abbildung 107: Darstellung Haltepunkte in Visual Studio 2019 (Quelle: Screen-Print von Visual Studio) 205
Abbildung 108: Miro Board Fokus-Gruppe Code-Metriken (Quelle: Eigene Darstellung) xiii
Abbildung 109: Miro Board Fokus-Gruppe: Performance (Quelle: Eigene Darstellung) xiii
Abbildung 110: Miro Board Fokus-Gruppe: Testing (Quelle: Eigene Darstellung) xiv

Tabellenverzeichnis

Tabelle 1: DSR-Aktivitäten und verwendete Methoden 11
Tabelle 2: Die Forschungszyklen in dieser Arbeit 14
Tabelle 3: Beispiele fCode Repräsentation von Module-Diagramm Komponenten 29
Tabelle 4: Erfolgskriterien Artefakt Performance-Analyse 33
Tabelle 5: System View dm_exec_procedure_stats 37
Tabelle 6: Übersicht der Query Stores 38
Tabelle 7: Modul-Diagramm Logikbausteintypen 42
Tabelle 8: Modul-Diagramm Logibaustein fCode Repräsentation 42
Tabelle 9: Übersicht Performance-Messdaten Logikbaustein 44
Tabelle 10: Neue Ende-fCodes für Strukturbausteine 56
Tabelle 11: Legende Erfüllungsgrad 68
Tabelle 12: Bewertung der Erfolgskriterien für die 2 Posity-Applikation Rezeptverwaltung und Kundenofferten-Verwaltung 69
Tabelle 13: Erfolgskriterien Artefakt Quellcode-Analyse 104
Tabelle 14: Legende Symbol – Musskriterien vs. Wunschkriterien (Spezifikation Code-Analyse) 106
Tabelle 15: Definition Bestandteile der Quellcode-Analyse 106
Tabelle 16: Betrachtungsbereiche (Scope) 109
Tabelle 17: Legende Symbol für Regeln - Implementiert vs. Konzept (Spezifikation Code-Analyse) 112
Tabelle 18: Regeln für Namenskonventionen 112
Tabelle 19: Regeln für Kommentare 114
Tabelle 20: Regeln für beschreibende Namen 114
Tabelle 21: Spezifizierende Regeln 115
Tabelle 22: Regeln zu Komplexität 116
Tabelle 23: Regeln zu Formatierung und Ausrichtung (Quellcode Struktur) 118
Tabelle 24: Metadaten einer Regel 120
Tabelle 25: Mapping von Regeln zu Posity Shapes 125
Tabelle 26: Ignore-List Konstellationen und Reichweite 127
Tabelle 27: Facets, Implementationen der Regeln und Beispiele 138
Tabelle 28: Implementierte Regeln für Namenskonventionen 143
Tabelle 29: Implementierte Regeln für Kommentare 145
Tabelle 30: Implementierte Regeln für beschreibende Namen 146

Tabelle 31: Implementierte Regeln für Spezifikation 147
Tabelle 32: Implementierte Regeln für Komplexität 148
Tabelle 33: Implementierte Regeln für Formatierung und Ausrichtung 150
Tabelle 34: Übersicht Diagramme der Applikation 156
Tabelle 35: Regeln für Applikation SolarvilleERP 157
Tabelle 36: Übersicht Diagramme der Applikation 161
Tabelle 37: Regeln für Applikation EN378 161
Tabelle 38: Legende Erfüllungsgrad für Bewertung 164
Tabelle 39: Bewertung Erfolgskriterien 165
Tabelle 40: Bankautomat Ursachen und Wirkungen 179
Tabelle 41: Testfälle aus Ursache-Wirkungs-Graph Analyse 179
Tabelle 42: Erfolgskriterien Artefakt Konzept Blackbox-Testing 182
Tabelle 43: Auswertung der Erfolgskriterien Blackbox-Testing 190
Tabelle 44: Erfolgskriterien Konzept Regressions-Testing 200
Tabelle 45: Legende Statussymbole Testtreiber 204
Tabelle 46: Legende Erfüllungsgrad für Bewertung 206
Tabelle 47: Erfolgskriterien validiert durch Delphi-Runde 207

Abkürzungsverzeichnis

ERD	Entity Relationship Diagram
fCode	Funktionaler Code
GUI	Graphical User Interface (deut. Benutzeroberfläche)
IDE	Integrated Development Environment
LCNC	Low-Code und No-Code
PDS	Posity Design Studio
PVM	Posity Virtual Machine
ZHAW	Zürcher Hochschule für Angewandte Wissenschaften
InIT	Institut für Informationstechnologie

Glossar

Code-Smell Code-Smells sind in der Programmierung Merkmale im Quellcode eines Programms, die möglicherweise auf ein Problem hinweisen und bei denen ein Refactoring angebracht wäre.

Citizen developer Citizen Developer bedeutet so viel wie "Fachbereichsentwickler" und beschreibt Mitarbeitende mit grossem Domänenwissen, die ohne Programmierkenntnisse softwarebasierte Anwendungen für ihren jeweiligen Fachbereich erstellen.

Diagramm Mit Diagramm sind die Posity Diagramme gemeint. Insgesamt gibt es 8 verschiedene Diagrammtypen, mit welchen sich eine Applikation im Posity Design Studio programmieren und konfigurieren lässt.

Issue Ein Issue entsteht, wenn bei der Quellcode-Analyse einer Applikation eine Regelverletzung festgestellt wird. Das Issue enthält Informationen über die Regel, welche verletzt wurde, als auch in welchem Teil der Applikation die Verletzung vorliegt.

Metrik Funktion, die eine Eigenschaft der Software in einem Wert abbildet. Der Wert, muss in diesem Kontext nicht zwingend ein Zahlenwert sein, sondern kann auch eine binäre Repräsentation (erfüllt/nicht erfüllt) haben oder in Textform ausgewiesen werden. Metriken werden von Regeln verwendet, um allfällige Verletzungen zu identifizieren.

Regel Eine Regel ist in unserem Kontext eine auf Erfahrung basierende Richtlinie für die Programmierung und wird explizit definiert. Programmcode kann daraufhin überprüft werden, ob Verletzungen von Regeln vorliegen. Die Regel verwendet dabei Metriken, um allfällige Verletzungen zu detektieren und zu bemessen.

Shape Beim Erstellen der Diagramme werden in Posity verschiedene Elemente zum Diagramm hinzugefügt, welche das Modell beschreiben. Diese Elemente werden in Posity als Shapes bezeichnet. Die Diagramme enthalten verschieden Typen von Shapes, wie bspw. einen Button im GUI-Diagramm oder eine Schleifen-Struktur im Modul-Diagramm.

Posity-GUI Eine Benutzeroberfläche, die in Posity in einem GUI-Diagramm spezifiziert wird und durch ein Modul in Posity ausgeführt, respektive angezeigt werden kann.

Posity Design Studio (PDS) Die Entwicklungsumgebung von Posity, welche für die Applikationsentwicklung eingesetzt wird. Wird auch als Posity IDE bezeichnet.

Ehrenwörtliche Erklärung

Wir erklären ehrenwörtlich, dass wir die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benutzt und die den Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen inländischen oder ausländischen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht. Die vorliegende Fassung entspricht der eingereichten elektronischen Version.

Ort, Datum: Winterthur, 28.05.2022

Unterschrift:

Marion Mürner

Matthias Christen

Vorwort / Danksagung

An dieser Stelle ist es uns ein besonderes Anliegen, uns bei all denjenigen zu bedanken, die uns während des Verfassungsprozesses dieser Arbeit begleitet haben.

Zuallererst gebührt unser Dank Adrian Moser und Max Meisterhans, die unsere Arbeit mit ihrer persönlichen und fachlichen Unterstützung begleitet haben und uns mit hilfreichen Anregungen und konstruktiver Kritik unterstützten.

Ein besonderer Dank gilt Prof. Jürgen Spielberger, dem Erfinder von Posity, mit dem wir während der ganzen Arbeit in gewohnt temperamentvollen Diskussionen regen Austausch pflegten.

Wir möchten uns ebenfalls recht herzlich bei der Forschungsgruppe Software-Engineering vom Institut für Informationstechnologie der ZHAW, welche sich unteranderem intensiv mit der Thematik Rapid Application Development und Model-Driven Engineering auseinandersetzt, für die anregende Diskussion bedanken.

Abschliessend bedanken wir uns auch noch gegenseitig für die ausgezeichnete Teamarbeit und den ausgesprochenen Ermunterungen.

Und natürlich Gourmet-Mäse!

1 Einleitung

Das Ziel dieses Kapitels ist, die Lesenden an das Thema heranzuführen und sie auf die eigentliche Arbeit vorzubereiten.

1.1 Ausgangslage

Mit der Qualität einer Software steht und fällt die Akzeptanz bei ihren Anwendern. Fehlerhafte Programmabläufe, falsche Resultate, inakzeptable Reaktionszeiten sind nur ein paar Beispiele, die bei Missachtung von minimalen Qualitätsstandards während der Softwareentwicklung entstehen können. Letztendlich beinträchtigen diese allerdings den Nutzen für den Endanwender. Mit der Entstehung von Mobile- und Web-Apps und deren Vertrieb über App Stores als globale Marktplätze, sind die Erwartungen der Endanwender an die Qualität einer Software nochmals gestiegen, weil die Vielzahl an Apps stetig Wünsche bezüglich neuer Funktionalität weckt. In diesen Marktplätzen bestimmen oft Marktführer, welche Basisfunktionalitäten oder welches Aussehen ein Softwareprodukt standardmässig bieten muss, und somit indirekt auch die Erwartungen an die Softwarequalität. Das heisst, die Qualität einer Software ist ein massgebender Faktor, ob sich eine Software verbreitet und gegen Konkurrenzprodukte am Markt durchsetzen kann. Ob eine Anwendung qualitativ hochwertig erscheint, basiert hauptsächlich auf Qualitätsmerkmalen, die für den Anwender nicht direkt ersichtlich sind. Im Entstehungsprozess einer Software werden in den einzelnen Konstruktionsschritten mit Hilfe von Metriken solche Softwarequalitätsmerkmale fortlaufend gemessen, evaluiert und verbessert. In der traditionellen textbasierten Softwareentwicklung haben sich für die Messung von Code-Qualität und für das Testing diverse Standards etabliert wie zum Beispiel statische und dynamische Codeanalyse, Unit-Testing, Versionskontroll- und Quellcodeverwaltungssysteme mit Hilfe von Tools wie SonarQube, Git, SQL Query Analyzers u.v.m. Meistens sind diese Qualitätsprozesse automatisiert und laufen im Hintergrund ab.

Neben der traditionellen Softwarenentwicklung gewinnt eine weitere Art zunehmend an Popularität und Verbreitung. Über sogenannte Low-Code und No-Code-Entwicklungswerkzeuge (LCNC) kann Software mit einfach bedienbaren, meist graphischen Programmiereditoren gebaut werden (Hurlburt, 2021). Eine Low-Code-Umgebung erfordert noch einige Programmierkenntnisse, während eine No-Code-Umgebung es erlaubt, mit geringen oder gar keinen Programmierkenntnissen Anwendungen zusammenzustellen. Damit

können auch weniger technisch versierte Entwickler oder Business-Analysten, sogenannte «citizen developers», komplexe Softwaresysteme bauen. Dabei übernehmen LCNC-Editoren die Handhabung von komplexen Abläufen und verbergen diese vor dem Entwickler oder bieten vorgefertigte Softwarebausteine im Baukastenformat an (Asif, 2020). Dies ist in etwa vergleichbar mit der Art und Weise, wie ein Kind Legosteine zu einem Bauwerk zusammensetzt. Die Anforderungen an die Softwarequalität bleiben aber dieselbe wie bei der Entwicklung einer Anwendung mit textbasierten Programmiersprachen.

1.2 Forschungslücke

Mit Low-Code und No-Code werden neue Konzepte und Merkmale eingeführt. Allerdings sind diese in der akademischen Forschung noch nicht genügend untersucht worden, um die bestehenden Herausforderungen und Möglichkeiten von Qualitätssicherungsmethoden in der Programmierung von LCNC-Software aufzuzeigen.

1.3 Forschungsfragen

In der klassischen, textuellen Programmierung haben sich diverse Qualitätssicherungsmethoden etabliert, die idealerweise in der LCNC basierten Softwareentwicklung wiederverwendet werden könnten, um auch dort die Qualität der Anwendungen messen und sicherstellen zu können.

Daraus ergeben sich für diese Forschungsarbeit die folgenden Forschungsfragen (FF) je Thema. Da diese Arbeit von zwei Personen ausgeführt wird, wird bei den Forschungsfragen jeweils die verantwortliche Person aufgeführt.

Bezogen auf die Analyse von Code und Performance, werden folgende zwei Forschungsfragen bearbeitet.

> *FF 1 (Matthias Christen): Welche Performance-Metriken und Messmethoden aus der klassischen, textuellen Programmierung können auf Low-Code und No-Code-Entwicklungswerkzeuge angewandt und mit diesen eine Performance-Steigerung erreicht werden?*

Kapitel 6 widmet sich der Beantwortung dieser Frage.

FF 2 (Marion Mürner): Welche Metriken für die Erkennung von Code-Smells aus der klassischen, textuellen Programmierung können auf Low-Code- und No-Code-Entwicklungswerkzeuge zur Verbesserung der Code-Qualität angewandt werden?

Kapitel 7 widmet sich der Beantwortung dieser Frage.

Bezogen auf die Thematik von Testverfahren wird die Arbeit um eine weitere Forschungsfrage ergänzt, wobei sich diese aus zwei Teilfragen zusammensetzt.

FF 3 (Gemeinsam): Eignen sich dynamische Software-Testverfahren für Anwendungen im Bereich Low-Code und No-Code und wie können diese eingesetzt werden?

Teil FF 3.1 (Matthias Christen): Eigenen sich spezifikationsorientierte Testmethoden für das Testen von Anwendungen im Bereich Low-Code und No-Code und wie können diese eingesetzt werden?

Kapitel 8 widmet sich der Beantwortung dieser Frage.

Teil FF 3.2 (Marion Mürner): Eigenen sich diversifizierende Testmethoden für das Testen von Anwendungen im Bereich Low-Code No-Code und wie können diese eingesetzt werden?

Kapitel 9 widmet sich der Beantwortung dieser Frage.

1.4 Abgrenzungen

Die Forschungsfragen werden nicht durch eine theoretische Betrachtung von verschiedenen LCNC- Plattformen bearbeitet. Für die Validierung der Softwarequalität von mit LCNC erstellten Anwendungen, soll ein Proof-of-Concept mit der modellbasierten LCNC-Softwareentwicklungsplattform Posity Design Studio (PDS) der Firma Posity AG erstellt werden. Das Posity Design Studio ermöglicht Applikationsentwicklung vollständig grafisch, ohne eine Zeile Code (No-Code) von Hand schreiben zu müssen und gilt deshalb aus Sicht der Autoren als repräsentativer Kandidat für Plattformen im LCNC-Bereich. Die Wahl ist nicht zuletzt auch deshalb auf das Posity Design Studio gefallen, weil die Autoren massgeblich an der Entwicklung dieser Plattform beteiligt sind und sich

aus der Arbeit erhoffen, die Code-Qualität der Anwendungen, welche mit Posity Design Studio erstellt werden, zu verbessern.

1.5 Beitrag der Arbeit

Am Ende dieser Arbeit werden insgesamt vier Artefakte entstehen. Für die beiden Forschungsfragen FF1 und FF2 wird jeweils ein Software-Prototyp erstellt, mit welchen Code- und Performance-Analysen von Anwendungen, welche mit dem Posity Design Studio erstellt worden sind, durchgeführt werden können. Für die Forschungsfrage FF3 wird jeweils ein Konzept für den Einsatz von spezifikationsorientierten und diversifizierenden Testverfahren erstellt. Das Konzept bezieht sich ebenfalls auf das Posity Design Studio, welches für dies Arbeit als Repräsentant der LCNC-Plattformen ausgewählt wurde.

1.6 Relevanz

Das Konzept der LCNC-Plattformen gibt es schon seit Jahrzehnten, noch bevor der Begriff verwendet wurde. Das Programmieren mit Low-Code- und No-Code- Editoren ist einfacher und somit auch für andere Berufsgruppen ausserhalb der klassischen Softwareentwicklung interessant. Die Idee ist, dass Anwendungen auf diesen Plattformen von Personen erstellt werden können, die über weniger technisches Fachwissen als ein professioneller Programmierer verfügen, aber dennoch leistungsstarke Technologien nutzen können, beispielsweise für Datenbankenanwendungen, Finanzanalysen, Webentwicklung und maschinelles Lernen. Dabei dürfen aber Qualitätsstandards nicht vernachlässigt werden. Die neuen Konzepte und Eigenschaften, die durch LCNC eingeführt werden, sind nach Khorram et al. noch zu jung, als dass die Forschung bereits Herausforderungen und Möglichkeiten im Bereich der Qualitätssicherungsmethoden aufzeigen könnte. Nach Lethbridge häufen sich in der Praxis, in auf solchen Plattformen geschriebene Software, jedoch oft grosse Mengen an komplexem Code an, dessen Wartung schwieriger sein kann, als bei herkömmlichen Sprachen, da die LCNC-Plattformen etablierte technische Verfahren wie Versionskontrolle, Trennung von Zuständigkeiten, automatisierte Tests und Clean Code Programmierung in der Regel nicht angemessen unterstützen.

1.7 Aufteilung der Arbeit und Zuständigkeiten

Diese Arbeit wird von zwei Master-Studierenden zusammen durchgeführt, da die Thematik die beiden Studierenden gleichermassen tangiert und interessiert. Matthias Christen

und Marion Mürner arbeiten beide als Softwareingenieure an der LCNC-Plattform Posity Design Studio. Die modellbasierte Entwicklungsumgebung ermöglicht die Erstellung von Individualsoftware von A-Z mit grafischen Komponenten. Durch die Zusammenarbeit können Synergien für die zu bearbeitenden Themen genutzt werden, um ein möglichst breites und fundiertes Forschungsresultat zu erzielen. Die Themen werden so weit wie sinnvoll voneinander abgegrenzt, um die individuellen Leistungsbeiträge der einzelnen Studierenden klar identifizieren zu können.

Die Aufteilung der Themen ist bei den Forschungsfragen hinterlegt. Anhang A enthält eine Übersicht aller Kapitel und der dafür verantwortlichen Person. Einige Kapitel wurden sinngemäss gemeinsam erstellt.

2 Related Work

Die Geburtsstunde der Disziplin des Software-Engineerings lässt sich ins Jahr 1968 zurückführen, als hierzu in Garmisch eine NATO-Konferenz stattgefunden hat (O'Regan, 2008). An dieser Konferenz wurde die Krise der Software diskutiert, da sich Probleme mit Projekten, die Software verspätet oder mit schlechter Qualität lieferten, häuften. Das Software-Engineering befasst sich seitdem mit soliden Techniken zur Entwicklung qualitativ hochwertiger Software, die die Anforderungen der Kunden erfüllt. Es umfasst Methoden zur Definition von Anforderungen, zum Entwurf, zur Entwicklung und zum Testen von Software, sowie zum Management von Änderungen. Qualitätsanforderung an und Bewertung von Software ist also seit mehr als 50 Jahren ein beherrschendes Thema von Studien des Software-Engineerings. In der klassischen Softwareentwicklung gibt es mittlerweile etablierte Methoden und Standards, wie beispielsweise die Normenreihe ISO/IEC 25000 der International Organization for Standardization, auch bekannt als SQuaRE (System and Software Quality Requirements and Evaluation), die das Ziel hat, einen allgemeinen Rahmen für die Bewertung der Qualität von Softwareprodukten zu schaffen.

Die Komplexität von Softwaresystemen ist ein in der Praxis häufig genannter und erlebter Grund, warum Erstellung, Betrieb und Wartung von Software zu einem Problem werden kann. Abstraktion und Modularisierung sind eine mögliche Antwort darauf Komplexität zu reduzieren. Je nach Abstraktionsniveau kann Software als grafisches Modell, Code in einer Programmiersprache, Maschinencode oder sogar als elektrische Spannung angesehen werden. Abhängig vom Level der Abstraktion gilt es, ganz unterschiedliche Herausforderungen der Komplexität zu bewältigen. Die Java-Entwicklerin ist dankbar dafür, dass sie sich beim Programmieren einer While-Schleife nicht um die Ladung von Elektronen kümmern muss. Genauso kann argumentiert werden, dass sich der Entwickler einer ERP-Applikation bei der Implementierung einer Datenbankabfrage nicht mit Transaktionslogik auseinandersetzen will. Der Ansatz von Low-Code und No-Code verfolgt das Ziel, den Abstraktionslevel, um eine weitere Ebene anzuheben und dadurch die zu Grunde liegende technische Komplexität möglichst zu verbergen. Wie schon in der Einleitung erwähnt, sollen dadurch auch Domänenexperten, mit wenig oder keiner Erfahrung in Softwareentwicklung, komplexe Anwendungen erstellen können.

Softwareinspektion und Testing spielen eine Schlüsselrolle beim Erstellen und Beurteilen von Software. Während diese Verfahren in der klassischen Softwareentwicklung etabliert

sind, ist die Programmierung mit Low-Code und No-Code nach Khorram et al. noch zu jung, als dass die Forschung bereits Herausforderungen und Möglichkeiten im Bereich der Qualitätssicherungsmethoden aufzuzeigen konnte.

2.1 Qualitätssicherungsmassnahmen Performance-Analyse und spezifikationsorientiertes Testverfahren

Performance-Analysen in der Software-Entwicklung sind grundsätzlich nichts neues und werden, wenn das Software-System in Betrieb geht, gefordert, weil das Software-System häufig zu langsam arbeitet – der Autor wagt auch zu behaupten, es gibt kein wissenschaftliches Paper für zu schnelle Software-Systeme. Software-Systeme verlangen demnach Schnittstellen, mit denen die Performance für die Abarbeitung von Aufgaben gemessen werden kann, um diese fortlaufend zu verbessern und die zur Verfügung stehenden Ressourcen optimal zu nutzen. In dieser Arbeit werden unter Performance quantifizierbare Messgrössen (z.B. Antwortzeiten, Verarbeitungszeiten, Datendurchsatz, usw.) (Dennis Kafura, 1985) verstanden und nicht Qualitätsmerkmale eines Software-Systems (Len et al., 2013), wie z.B. die Stabilität, Sicherheit oder Verfügbarkeit.

Die quantifizierbaren Messgrössen stehen immer in einem bestimmten Kontext. So hat zum Beispiel jede CPU ihre Zeit, also Kontext, in der sie Herausragendes leistet, um kurze Zeit später vom Tron gestossen zu werden. Gleiches gilt für Software-Systeme und ihre zu erledigenden Aufgaben. Erst eine Betrachtung inklusive Umfeld, in dem das Software-System eingebettet ist, lässt eine realistische Aussage zur Software-System-Performance zu. Besonders markant kommt dies bei Embedded Software-Systemen zu tragen, wo die verfügbaren Ressourcen per se limitiert sind.

Aus dieser Tatsache, dass in jedem Software-System die Ressourcen für die Verarbeitung von Aufgaben limitiert sind und so der Wunsch entsteht diese zu optimieren, folgt, dass auch eine in LCNC entwickelte Anwendung Möglichkeiten erfordert die Performance in ihrem Umfeld mit einem Analyse Werkzeugen zu untersuchen. Wissenschaftliche Arbeiten spezifisch zum Thema Performance mit LCNC erstellten Anwendungen gibt es zum aktuellen Zeitpunkt nicht. Dies wahrscheinlich, weil sich LCNC-Entwicklungsumgebungen erst seit kurzem als Trendthema etablieren (Asif, 2020). Im Gegensatz hierzu lassen sich für klassische Programmiersprachen die benötigten Methoden für die Ermittlung der quantitativen Performance-Messgrössen in zahlreich etablierten Performance-Messwerk-

zeugen, wie z.B. dotTrace (JetBrains, 2021b), oder direkt eingebettet in Software-Entwicklungsumgebungen, wie z.B. in Visual Studio (Microsoft, 2021b), finden. Diese Methoden eignen sich auch für LCNC-Entwicklungsumgebungen, weil dieselben Performance-Messgrössen dargestellt werden müssen.

Neben der Beurteilung eines Software-Systems anhand seiner Perfomance ist es wichtig zu wissen, ob die geforderten Aufgaben korrekt abgearbeitet werden. Dies ist nicht garantiert, weil die Systeme meistens nicht nur von einer Person gestaltet und von derselben Person verwendet wird. Vielfach sind beim System-Design dutzende Parteien involviert, die die nötigen funktionalen und nicht funktionalen Anforderungen gemeinsam für das Software-System bestimmen. Dass es dabei zu Missverständnissen und unterschiedlichen Meinungen und Ansprüchen kommt, ist menschlich. Deshalb gilt, etwas salopp gesagt, wie in der Küche «Viele Köche verderben den Brei» auch im Software-System-Engineering «Viele Entwickler verderben das System». Natürlich wird weiterhin in Kooperation mit Anderen Brei gekocht und werden Software-Systeme gebaut, weil sich in beiden Welten Verfahren etabliert haben, trotz Problemen qualitativ befriedigende Resultate zu erzielen.

Beim Bau von Software-Systemen ist ein Verfahren die spezifikationsorientierten Tests, respektive dynamischen Tests (Andreas & Tilo, 2012) mit denen die fertig entwickelten Komponenten aus dem Software-System gezielt auf ihre zu erfüllende Funktionalität hin überprüft werden. Eine beliebte Testmethode, mit der sich generell alle Komponenten testen lassen, ist das Blackbox-Testing (Andreas & Tilo, 2012). So wird vermutet, dass das Blackbox-Testing sich auch in LCNC-Entwicklungsumgebungen anwenden lässt. Zum Beispiel bieten einige Hersteller bereits die Möglichkeiten von Unit-Tests in ihren LCNC-Entwicklungsumgebungen an (Khorram et al., 2020).

2.2 Qualitätssicherungsmassnahmen Quellcode-Analyse und diversifizierende Testverfahren

Möglicherweise wurde in der Vergangenheit die Thematisierung der Qualitätssicherung von in LCNC erstellten Anwendungen vernachlässigt, weil davon ausgegangen wird, dass Qualitätssicherung auf diesem Abstraktionsniveau weniger relevant ist und darunterliegende Schichten die Qualität mit entsprechenden Massnahmen sicherstellen müssen. Aus Sicht der Autorin ist diese Annahme jedoch falsch. Sie geht davon aus, dass auf jeder Abstraktionsstufe qualitative Mängel vorhanden sein können. Diese Mängel werden über

die Ebenen hinweg nicht zwangsläufig identisch sein, sondern sich auf der jeweils gleichen Abstraktionsstufe, wie der erstelle Code befinden. LCNC führt neue Konzepte und Eigenschaften ein, wie dies beispielweise auch schon mit der Einführung der objektorientierten Programmierung (OOP) der Fall war. OOP, mit ihrem Verständnis der Abstraktion, hat wohl einige Probleme gelöst, aber auch neue Probleme geschaffen. Deshalb dürfen Qualitätsstandards auf keiner Abstraktionsstufte, und somit auch für LCNC nicht, vernachlässigt werden.

Nach Lethbridge häufen sich in der Praxis, in auf LCNC-Plattformen geschriebener Software, oft grosse Mengen an komplexem Code an, dessen Wartung schwieriger sein kann, als bei herkömmlichen Sprachen, da die LCNC -Plattformen etablierte technische Verfahren wie Versionskontrolle, Trennung von Zuständigkeiten, automatisierte Tests und Clean Code Programmierung in der Regel nicht angemessen unterstützen. Daher muss sichergestellt werden, dass LCNC-Plattformen moderne Software-Engineering-Praktiken genauso gut unterstützen, wie herkömmliche Sprachen. Sie müssen vor allem die Möglichkeit bieten, die Qualitätssicherungsmassnahmen der Anwendung auf dem gleichen Abstraktionslevel ausführen zu können, in der die Anwendung geschrieben worden ist, damit mit denselben Programmierkenntnissen gegen Mängel vorgegangen werden kann.

Neben vielen weiteren Methoden, welche das Ziel haben, die Anzahl Fehler in einem Programm zu reduzieren, ist die manuelle Inspektion (Code Review) ein sehr effektives Mittel zur Sicherung der Qualität der untersuchten Codes (Andreas & Tilo, 2012, S. 78). Manuelle Inspektion in umfangreichen Projekten erfordert jedoch sehr viel personelle Ressourcen, deshalb ist eine automatisierte Analyse des Quellcodes effizienter. Dank dem Umstand, dass Menschen dazu neigen, immer wieder in dieselben Fallen zu tappen, fallen die meisten Fehler in bekannte Kategorien. Statische Analyse-Programme, nutzen diese Muster und untersuchen Quellcode mit einem Satz an Regeln auf Fehler.

Die statische Code-Analyse ist eine in der traditionellen Softwareentwicklung etablierte Methode zur Qualitätssicherung. Viele, der in einem Programm vorhandenen Fehler sind für den Compiler nicht sichtbar. Statische Code-Analyse ist eine Möglichkeit, solche Fehler zu finden und die Mängel in einer Software zu reduzieren. In den 1970er Jahren hat Johnson, damals in den Bell Laboratories tätig, Lint geschrieben, ein Werkzeug um Quellcode von C Programmen, die fehlerfrei kompilieren, auf Fehler zu untersuchen, die der Compiler nicht detektieren konnte.

Von dem Nutzen, den statische Code-Analyse-Programme der traditionellen Entwicklung bringen, sollten auch LCNC-Plattformen profitieren. In der Forschung gibt es hierzu kaum Material. Deshalb orientiert sich diese Arbeit stark an den Regeln und Metriken der traditionellen Softwareentwicklung. In der Vorstudie der Masterarbeit von Marion Mürner und Matthias Christen wurden Metriken aus der objektorientierten Programmierung untersucht und gemäss ihrer Eignung für eine Adaption für LCNC kategorisiert. Auf Basis dessen wurden einige Metriken, welche gemäss Vorstudie, als besonders geeignet für eine Adaption und besonders wichtig für die Beurteilung der Qualität angesehen, in diese Arbeit aufgenommen und im Artefakt implementiert.

Neben den statischen Qualitätssicherungsmassnahmen gibt es die dynamischen Testverfahren. Dabei wird Software durch Ausführung getestet. Die statische Code-Analyse ersetzt das Testen nicht, sondern ergänzt sie. Denn durch Testen können völlig unerwartete Fehler aufgedeckt werden, die nur durch das Ausführen der Programme erkennbar werden. Al Alamin et al. stellen in ihrer empirischen Studie, in der sie die Diskussion von Entwicklern über Herausforderungen bei der Entwicklung von LCNC-Software untersuchen, fest, dass das automatisierte Testen ein grosses Bedürfnis ist. In der Verantwortung der Autorin liegt die konzeptionelle Bearbeitung der diversifizierenden Testverfahren, bei welchen der Vergleich nicht zwischen Testergebnis und Spezifikation (erwartetem Resultat) gemacht wird, sondern zwischen zwei konkreten Testergebnissen. Ein wesentlicher Vorteil der diversifizierenden Testtechnik, im Gegensatz zu spezifikationsorientierten Testverfahren, ist die Automatisierung des Erfassens der Testfälle und des Vergleichs (Liggesmeyer, 2009, S. 180).

3 Vorgehen und Methoden

Das Forschungsdesign wird auf die Methodik des Design Science Research (DSR) nach Hevner et al. ausgelegt. Dies lässt sich damit begründen, dass sich dieser Ansatz stark an der Erstellung von Artefakten orientiert und in dieser Masterarbeit insgesamt vier Artefakte entstehen. Für den Forschungsprozess wird das Rahmenmodell nach Peffers et al. verwendet (Abbildung 6) und als Ausgangslage der problemzentrierte Einstieg gewählt. Es wird bei der Analyse und Identifikation des Problems (siehe Kapitel 4) gestartet und mit der Motivation, dieses Problem zu beheben, werden die weiteren Schritte angegangen.

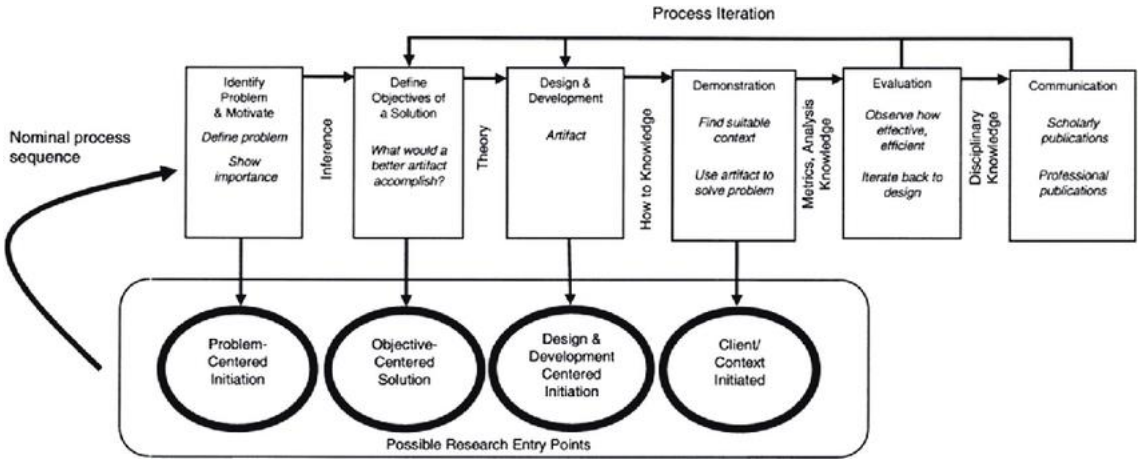


Abbildung 1: Forschungsmodell nach Peffers et al.

Die nachfolgende Aufstellung formuliert für jeden der sieben Prozessschritte des Forschungsmodells die gewählten Methoden.

Tabelle 1: DSR-Aktivitäten und verwendete Methoden

DSR-Aktivität	**Methode / Vorgehen**
Identifikation	Forschungsproblem durch **Literaturrecherche** definieren und den Wert der Lösung erläutern. Mit einer **Fokus-Gruppe** das Problem und die Anforderungen an die Lösung ergründen und priorisieren.

	Das Problem soll so weit abstrahiert werden, dass dieses in **Anforderungen transferiert** werden kann.
Ziel / Zweck der Lösung definieren	Das Ziel und der Zweck der Lösung werden aus dem Ergebnis der **Fokus-Gruppe** abgeleitet und entsprechende Erfolgskriterien für die Artefakte definiert.
Design und Implementation	Es werden vier Artefakte LCNC-Plattform Posity erstellt und instanziiert: • (Ar 1): Performance-Analysator als ausführbare **Software.** • (Ar 2): Quellcode-Analysator als ausführbare **Software.** • (Ar 3): **Konzept** für Erstellung, Erfassung und Ausführung von spezifikationsorientierten Testverfahren (Black-box-Tests). • (Ar 4): **Konzept** für Erstellung, Erfassung und Ausführung von diversifizierenden Testverfahren (Regressionstests).
Demonstration	Die Software-Artefakte werden mittels **Fallstudie** an jeweils zwei bestehenden Kundenapplikation ausgewertet. Die Konzepte werden durch Diskussion mit einer **Delphi-Runde** bewertet und validiert. • (Ar 1): Performance von einem wichtigen Prozess (Kernaufgabe) pro Applikation messen und bewerten (HS Rezeptverwaltung und SolERP). • (Ar 2): Ausgewählte Diagramme (Code) von wichtigen Prozessen (Kernaufgaben) pro Applikation analysieren und bewerten (NormenhilfeEN378 und SolERP). • (Ar 3) + (Ar 4): Konzept in Diskussionsrunde der Delphi-Runde vorstellen und bewerten.

Evaluation	Für die Evaluation werden **quantitative Faktoren**, wie die Anzahl und Legitimation der gefundenen Code-Smells und Performanceanalysen an Kundenapplikation der Posity AG, **gemessen und ausgewertet**. Sowie **qualitative Faktoren**, wie Bedienbarkeit der Software-Artefakte und Verständlichkeit der Modelle und Konzepte durch die Autoren selbst und durch einen Posity-Entwickler **untersucht**. Die Erfahrungen werden in einer anschliessenden gemeinsamen Diskussion mit einer **Delphi-Runde** zusammengetragen und die **Erfolgskriterien bewertet**.
Kommunikation	Für die Bekanntmachung der Lösung und entsprechende Handlungsempfehlung dient als **schriftliche Berichterstattung diese Masterarbeit**.

Bei der Durchführung wird nach den von Hevner vorgeschlagenen Forschungszyklen vorgegangen. In dieser Arbeit wird anhand der drei Zyklen wie sie in Abbildung 2 abgebildet sind vorgegangen. Abhängig vom Zyklus werden die in Tabelle 1 beschriebenen Aktivitäten ausgeführt.

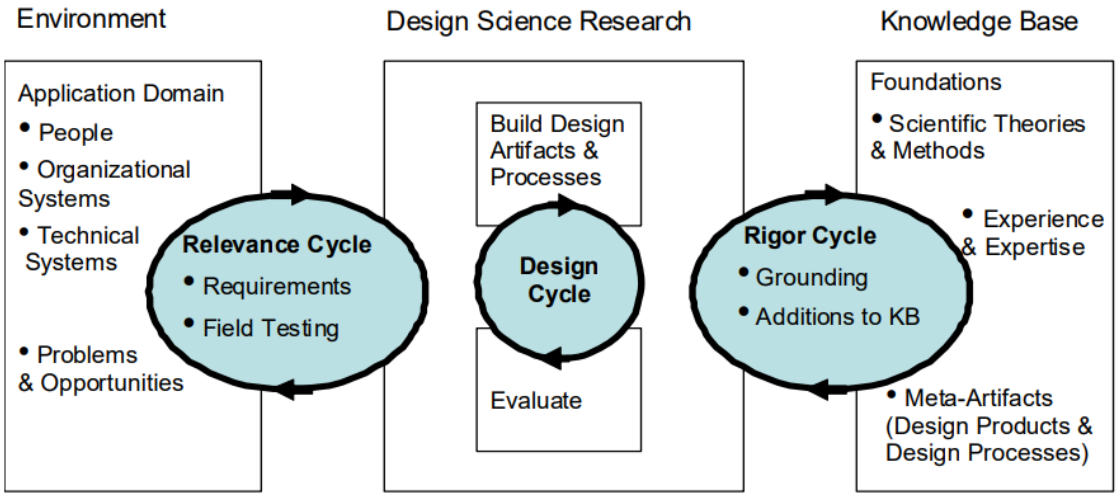


Abbildung 2: Forschungszyklen nach Hevner

Der Relevance-Zyklus verbindet das Umfeld des Forschungsprojekts mit den Design Science Aktivitäten. Der Rigor-Zyklus verbindet die Design Science Aktivitäten mit der Knowledge Base aus wissenschaftlichen Grundlagen, Erfahrungen und Fachwissen, die

dem Forschungsprojekt zugrunde liegt. Der zentrale Design-Zyklus iteriert zwischen den Kernaktivitäten der Implementation und der Evaluierung der Artefakte der Forschung. Tabelle 2 beschreibt die geplanten Aktivitäten in den einzelnen Zyklen während dieser Arbeit.

Tabelle 2: Die Forschungszyklen in dieser Arbeit

Zyklus	Aktivitäten
#1 Zyklus: Der Relevance-Zyklus	Identifikation des Problems, Ziel und Zweck der Lösung definieren durch Analyse der Umgebung (Stakeholder, Organisationale Strukturen und Technische Strukturen). Evaluation und Demonstration der Artefakte (siehe Design und Implementation in Tabelle 1) in Fokusgruppe und mit bestehenden Posity-Applikationen.
#2 Zyklus: Rigor-Zyklus	Literaturrecherche Fokusgruppe (Related Work). Schlussfolgerungen und Handlungsempfehlungen ableiten und in der Masterarbeit kommunizieren.
#3 Zyklus: Design-Zyklus	Anforderungen definieren und Artefakt erstellen, anhand Demonstration und Evaluation bewerten. Es werden zwei Iterationen innerhalb dieses Zyklus gemacht.

4 Analyse

Dieses Kapitel beschreibt den Kontext der Problemstellung, indem Stakeholder, organisationale und technische Strukturen analysiert werden.

Literatur, spezifisch zur Codequalität in LCNC-Anwendungen, ist Stand heute wenig bis keine verfügbar. Es musste sich auf Literatur aus klassischer, textueller Programmierung und einige wenige wissenschaftliche Arbeiten abgestützt werden. Deshalb wurde früh in der Arbeit entschieden, das fehlende Wissen, die Probleme und Anforderungen an eine Lösung mit einer Fokus-Gruppe qualitativ zu erarbeiten. Hierzu wurden 7 Mitarbeitende aus dem Institut für Angewandte Informationstechnologie (InIT) der ZHAW aus dem Forschungsbereich Software Engineering für eine Fokus-Gruppe eingeladen. Vorab wurde mit den Mitarbeitenden vom InIT, mittels einer Mentimeter[1]-Umfrage, die Bedeutung von bestimmten Codequalitätskriterien, Performance-Messgrössen und Testtypen eingestuft. Die Resultate dieser Umfrage (siehe Abbildung 3, Abbildung 4 und Abbildung 5) hatte Relevanz für die Bestimmung der Anforderungen der zu erstellenden Artefakte und war auch der Einstieg für die Bearbeitung des Themas Codequalitätskriterien in LCNC-Entwicklungsanwendungen in der Fokus-Gruppe.

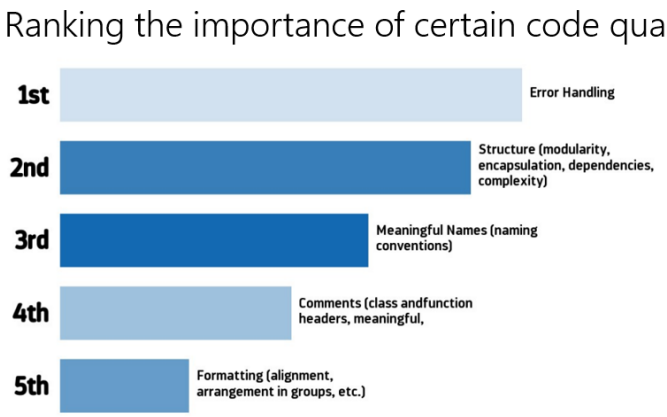


Abbildung 3: Resultat Mentimeter-Umfrage Einstufung Codequalitätskriterien (Quelle: Eigene Darstellung aus Mentimeter)

[1] Mentimeter, https://www.mentimeter.com/, (Abgerufen: 25.05.2022)

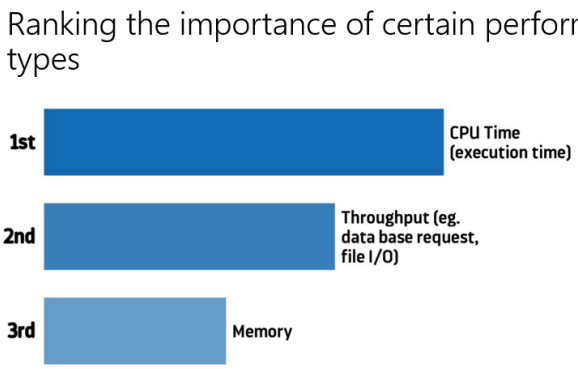


Abbildung 4: Resultat Mentimeter-Umfrage Einstufung Performance-Messgrössen (Quelle: Eigene Darstellung aus Mentimeter)

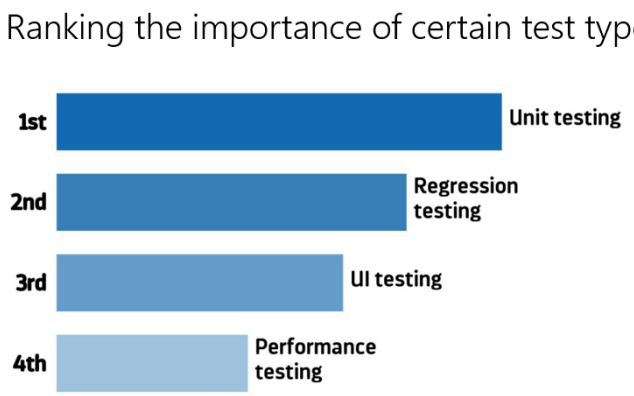


Abbildung 5: Resultat Mentimeter-Umfrage Einstufung Testtypen (Quelle: Eigene Darstellung aus Mentimeter)

In der Fokus-Gruppe wurde mittels vorbereiteter Fragen versucht die Anforderungen für das Artefakt zu definieren und zu bestimmen, auf welchen Codequalitätsmetriken aus der klassisch textuellen Softwareentwicklung der Fokus gesetzt werden soll, um damit die Codequalität von LCNC-Anwendungen zu verbessern. Die Fragen und die erarbeiteten Antworten aus der Diskussion in der Fokus-Gruppe sind in Anhang E ersichtlich. Aus diesen Ergebnissen wurden die Anforderungen und Erfolgskriterien für die einzelnen Artefakte definiert.

4.1 Stakeholder (involvierte Akteure)

Als Hersteller, von der für die Forschung verwendete LCNC-Entwicklungsumgebung Posity Design Studio, steht die Posity AG als Hauptakteur im Zentrum dieser Arbeit. Ihr Interesse besteht darin, qualitativ möglichst hochwertige Software an ihre Kundschaft auszuliefern. Deshalb sollen die entstandenen Artefakte nach Möglichkeit in der LCNC-

Entwicklungsumgebung Posity Design Studio produktiv eingesetzt werden. Dadurch profitieren die Entwickler bei der Posity AG und indirekt auch deren Kundschaft.

Stark verknüpft ist die Posity AG auch mit dem InIT an der ZHAW, wo in der Forschungsgruppe Software-Engineering das Posity Design Studio in Forschungsprojekten im Bereich Model Driven Engineering eingesetzt wird. Resultate aus dieser Arbeit fliessen also direkt in weitere Forschungsprojekte. Ob andere Hersteller von LCNC-Entwicklungsumgebungen von dieser Arbeit profitieren können, ist fraglich, weil die Arbeit sehr stark auf dem proprietären Produkt der Posity AG aufsetzt. Für die Wirtschaftsinformatik können, die in den Schlussfolgerungen erwähnten, Anknüpfungspunkte für weitere Forschung interessant sein.

4.2 Organisationale Strukturen

Zurzeit umfasst das Entwicklungsteam bei der Posity AG 11 Mitarbeitende, die Businessanwendungen und Industrie-Software im deutschsprachigen Raum entwickeln. Vor allem die Businessanwendungen werden nach Möglichkeit mit dem Kernprodukt Posity Design Studio erstellt. Die Autoren sind als Softwareingenieure bei der Firma angestellt und sind auch bei der Weiterentwicklung des Posity Design Studios involviert. Der Geschäftsführer leitet am InIT den Forschungsbereich Software Engineering, wodurch eine enge Bindung zur ZHAW besteht. Dadurch steht die Posity AG im direkten Austausch mit der Software Engineering-Forschungsgruppe und kann Know-how aus aktueller Forschung direkt in ihren Produkten anwenden.

4.3 Technische Strukturen

Im Posity Design Studio werden alle Teile einer Software in graphischen Diagrammen entwickelt. Gebaut wird eine Anwendung, indem der Entwickler über eine graphische Benutzeroberfläche Bausteine in diesen Diagrammen mit der Computermaus platziert und miteinander verknüpft. Eine lauffähige Applikation entsteht, wenn auch die einzelnen Diagramme untereinander verknüpft sind (siehe Abbildung 6). Zum Beispiel können Attribute aus einer Datenbank-Abfrage (Query) im Posity-GUI-Diagramm als Steuerelemente eingefügt werden, oder das Modul-Diagramm ist verantwortlich für die Anzeige der im Posity-GUI-Diagramm spezifizierten Benutzeroberfläche. Mit dieser Software-Darstellungsform und diesem Entwicklungs-Mechanismus ist es möglich, eine Software zu entwickeln, ohne je eine Zeile Programmcode zu schreiben. Deshalb kann das Posity Design Studio als NC-Entwicklungsumgebung eingestuft werden.

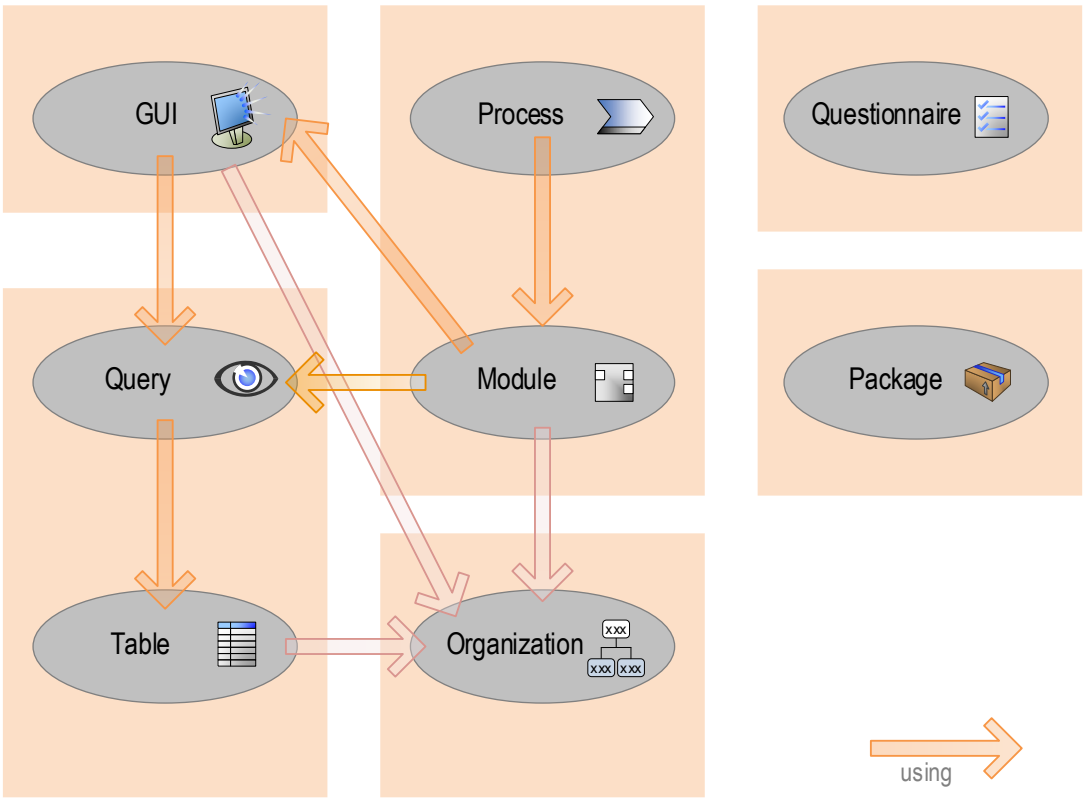


Abbildung 6: Posity Diagramme und Abhängigkeiten
(Quelle: interne Dokumentation Posity AG)

Technisch gesehen wird aus den graphischen Diagrammen immer ein Programmcode (als fCode bezeichnet, Posity eigener Bytecode) erzeugt. Beim Table- und Query-Diagramm werden zusätzlich entsprechende Datenbankobjekte (Datenbank-Tabellen und SQL-Stored Procedures) auf dem Datenbank-Server (siehe Abbildung 7) erstellt. Der fCode wird durch eine Laufzeitumgebung (Runtime Environment) interpretiert und so die Applikation ausgeführt. Diese Architektur zu kennen ist für die Entwicklung der Artefakte wichtig, weil mögliche Messpunkte für die Codequalitätsmetriken nicht nur in den Diagrammen vorkommen.

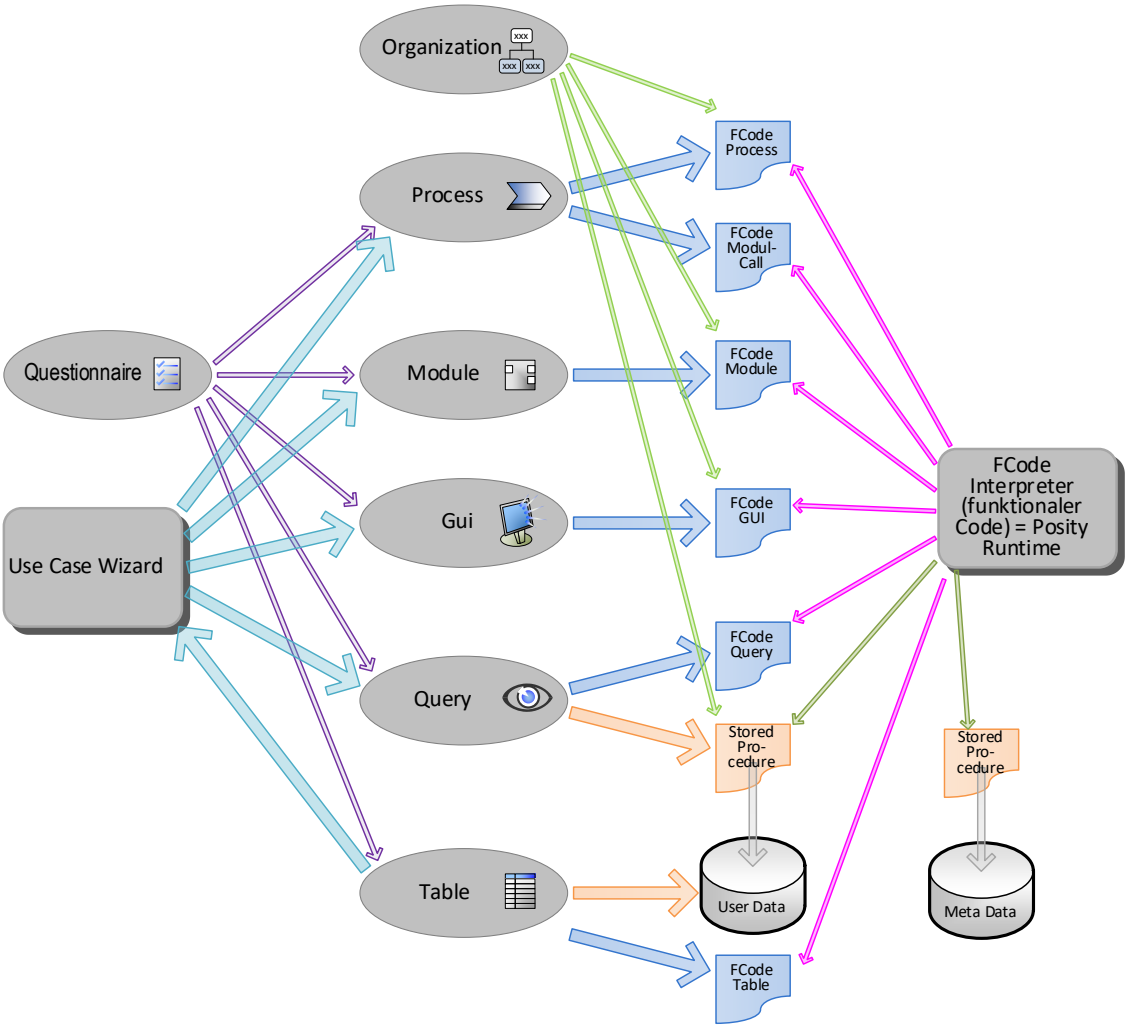


Abbildung 7: Posity Diagrammtypen und Abhängigkeiten

(Quelle: interne Dokumentation Posity AG)

5 Artefakt DevTools

Dieses Kapitel beschreibt das Artefakt DevTools mit dem das Artefakt Quellcode-Analyse und das Artefakt Performance-Analyse in die Posity IDE integriert werden.

5.1 Spezifikation

Das Artefakt Quellcode-Analyse sowie das Artefakt Performance-AnalyseArtefakt Performance-Analyse sind für die Anwendung durch einen Posity-Entwickler bestimmt und sollen ihn bei seiner Arbeit als Software-Entwickler in der Posity IDE unterstützen. Die Artefakte sind nicht nötig, um eine Anwendung in der Posity IDE zu bauen, aber sie helfen dem Software-Entwickler beim Erstellen und Unterhalten der Applikation im Development, respektive im Maintenance Lebenszyklus einer Applikation (David, 2008). Weil die beiden Artefakte für die Entwicklung einer Posity-Applikation nicht unbedingt nötig sind, sollen diese eher im Hintergrund der Posity IDE untergebracht werden und nur bei Bedarf in den Vordergrund rücken. Ein ähnliches Bedienkonzept für Entwicklungs-Tools findet sich bei den Web-Developer-Tools in den Webbrowsern Mozilla Firefox, Google Chrome und Microsoft Edge. Mit der Funktionstaste F12 erscheinen die Web-Developer-Tools angedockt an die Webseite oder als eigenes Fenster, mit denen ein Web-Entwickler einen tieferen Einblick in das Leben oder den Aufbau der angezeigten Webseite erhält. Ähnlich wie die Web-Developer-Tools soll das Artefakt DevTools funktionieren und dem Posity-Entwickler mehr Informationen über den Code und das Laufzeitverhalten einer Posity-Applikation bieten.

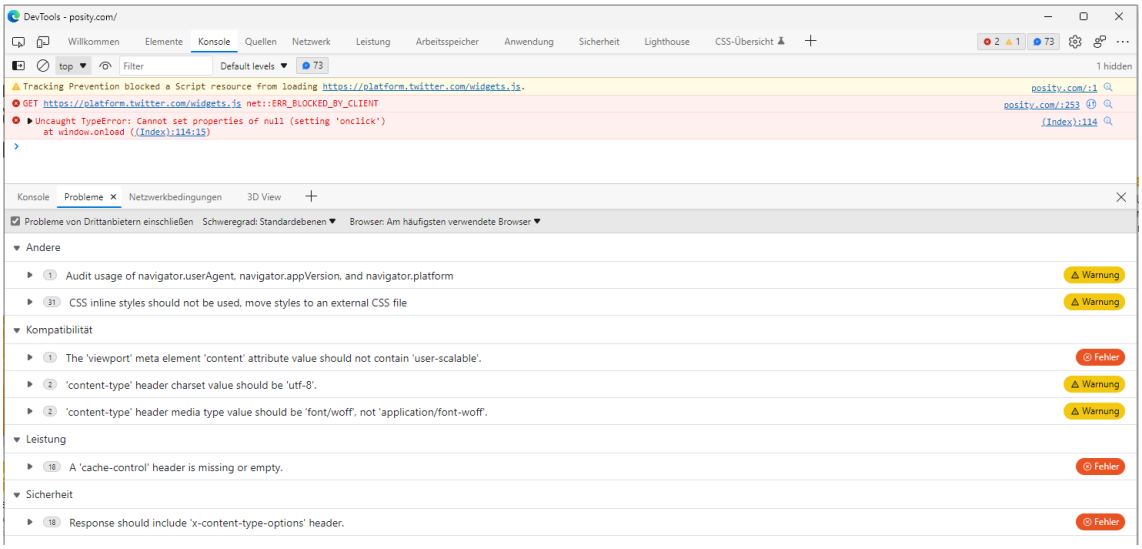


Abbildung 8: Web-Developer-Tools im Webbrowser Microsoft Edge mit Ansicht zu Hinweisen zum HTML-Code der angezeigten Webseite (Quelle: Eigene Darstellung)

5.2 Implementation

Die Klassen der Implementation für das Artefakt DevTools befinden sich in der Datei `DevTools.cls`. Die DevTools bestehen aus den Klassen `Form`[2], mit welcher sich ein beliebiges Fenster in einer C# Anwendung darstellen lässt, und einem `TabControl`[3], welches das Artefakt Quellcode-Analyse und das Artefakt Performance-Analyse jeweils als `TabPage`[4] aufnehmen kann. Weitere Artefakte, respektive Tools für den Posity-Entwickler, können einfach über eine zusätzliche TabPage eingebunden werden. Die Abbildung 9 zeigt in einem UML Klassendiagramm, wie die aus C# zur Verfügung stehenden Klassen verwendet wurden, um die Software für die DevTools zu strukturieren und zu implementieren.

[2] C# Form Klasse, https://docs.microsoft.com/en-us/dotnet/api/system.windows.forms.form?view=windowsdesktop-6.0, (Abgerufen: 11.05.2022)

[3] C# TabControl Klasse: https://docs.microsoft.com/de-de/dotnet/api/system.windows.forms.tabcontrol?view=windowsdesktop-6.0, (Abgerufen: 11.05.2022)

[4] C# TabPage Klasse: https://docs.microsoft.com/de-de/dotnet/api/system.windows.forms.tabpage?view=windowsdesktop-6.0, (Abgerufen: 12.05.2022)

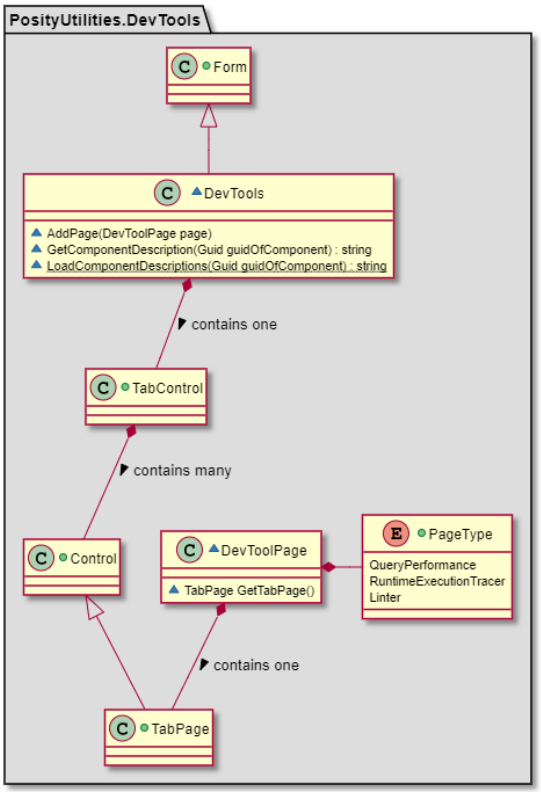


Abbildung 9: UML Klassendiagramm Artefakt DevTools

Die Abbildung 10 zeigt anhand einer Bildschirmaufnahme des Artefakt DevTools, wie die aktuelle Implementation von diesem Artefakt aussieht.

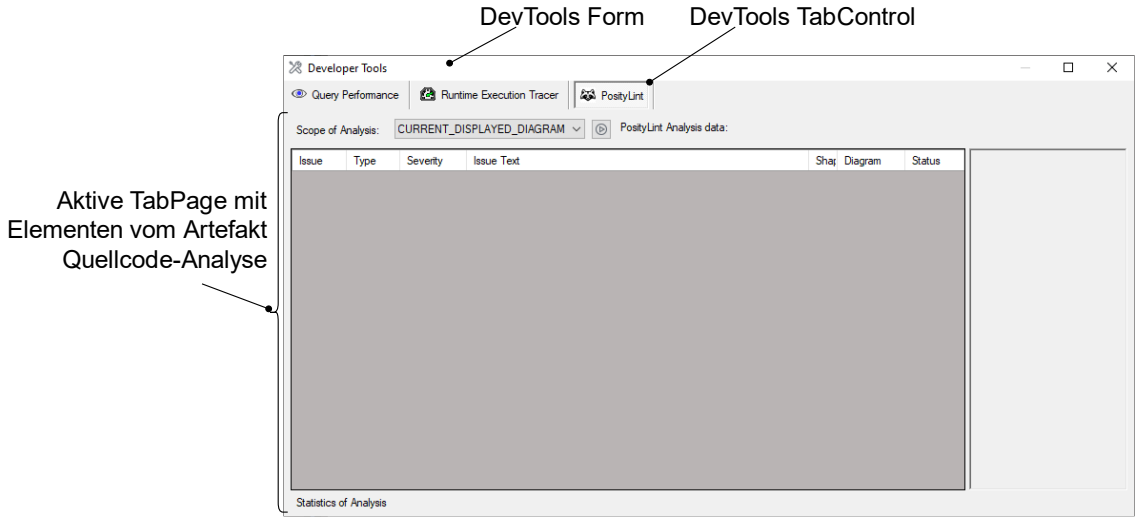


Abbildung 10: Artefakt Posity DevTools (Quelle: Eigene Darstellung)

Die DevTools können in Posity an zwei Stellen aufgerufen werden. Entweder öffnet der Entwickler die DevTools über das Fenster-Menü oder über ein Icon in der Status-Leiste.

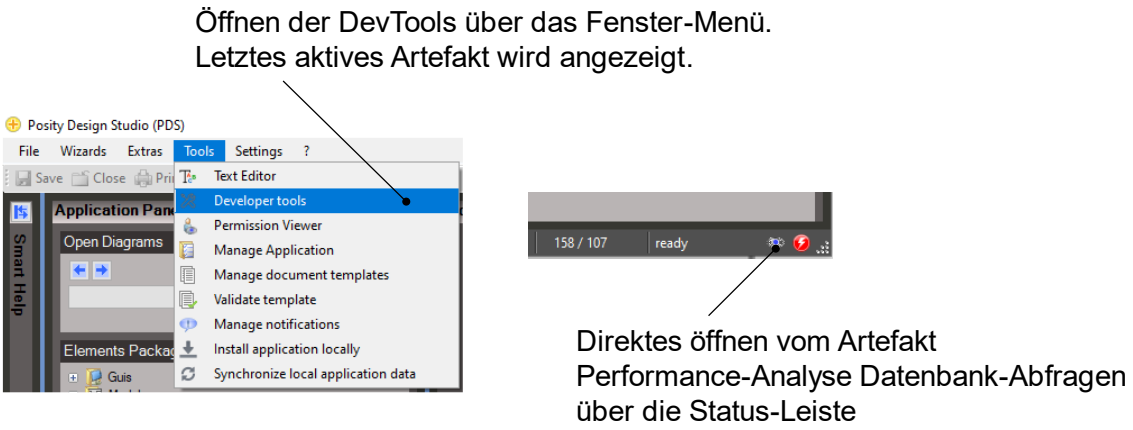


Abbildung 11: Möglichkeiten für das Öffnen der DevTools
(Quelle: Eigene Darstellung)

Über die Status-Leiste werden die DevTools geöffnet und ein bestimmtes Artefakt direkt angezeigt. Beim Öffnen über das Fenster-Menü wird das letzte aktive Artefakt angezeigt. In den folgenden Kapiteln 6 und 7 wird beschrieben wie die Tools, die im Artefakt DevTools verwendet werden können, entwickelt wurden.

Oft verwenden die in die DevTools integrierten Tools die Bezeichnungen von den Komponenten in den verschiedenen Posity-Diagrammen (z.B. Name eines Buttons). Deshalb werden beim ersten Starten der DevTools diese Bezeichnungen mit der SQL-Stored Procedure `SystemData.ComponentDescriptions_Fill` vom Datenbank-Server geladen. Die SQL-Stored Procedure gibt eine Liste aller Bezeichnungen und die GUID der jeweiligen Komponente zurück, die in den DevTools als C# `Dictionary`[5] abgelegt werden. Damit kann jedes Tool Bezeichnungen für Komponenten zentral bei der `DevTool` Klasse abfragen (Statische Methode `DevTools.GetComponentDescription`).

[5] C# Dictionary Klasse, https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.dictionary-2?view=net-6.0, (Abgefragt: 14.05.2022)

6 Artefakt Performance-Analyse

Dieses Kapitel beschreibt das Artefakt, welches im Rahmen dieser Masterarbeit im Bereich der Performance-Analyse für die LCNC-Plattform Posity entstanden ist.

6.1 Anwendungskontext

Das entwickelte Artefakt besteht aus zwei Teilen, mit denen sich Performance-Analysen in den Bereichen Datenbank-Abfragen und Applikationslogik für eine in der Posity IDE entwickelten Anwendung durchführen lassen. Die Posity IDE eignet sich besonders für die Entwicklung von datenbankzentrierten[6] Anwendungen. In einer datenbankzentrierten Anwendung rücken die Datenbank-Abfragen in den Mittelpunkt einer Anwendung und bilden deshalb deren Rückgrat. Dies heisst wiederum, Datenbank-Abfragen sind als Performance kritisch zu betrachten und sind bei einer Performance-Analyse einer Anwendung nicht ausser Acht zu lassen.

Die in Posity-Programmcode geschriebene Applikationslogik besteht im einfachsten Fall aus wenigen Komponenten, wird aber beim Abbilden von komplexer Business-Logik ebenso komplex wie diese und kann hunderte von Komponenten umfassen. In der klassischen Softwareentwicklung wird der Entwickler mit Software-Patterns und Best-Practices bei der Übersetzung von komplexer Business-Logik in Programmcode unterstützt. Diese sind als Leitfaden für gewisse Problemstellungen zu verstehen und geben dem Entwickler einen Rahmen und ein Werkzeug in die Hand, um das Problem verständlich und performant in Programmcode zu übersetzen. Auch für die Entwicklung von Posity-Programmcode gibt es Patterns und Best-Practices. Eine Posity Best-Practice ist z.B. in einem Modul nur die Applikationslogik für die Funktionalitäten von einem Posity-GUI zu implementieren. Aber ohne Kenntnisse der Posity Best-Practices, oder wer der Möglichkeit in der Posity IDE vom einfachen graphischen Zusammen-Klicken einer Applikation erliegt, ist das Chaos vorprogrammiert und es entstehen meist schlecht wartbare und wenig performante Applikationen. Deshalb ist es wichtig, auch im Bereich der Applikationslogik ein Werkzeug zu haben, mit dem nicht performanten Programmteile identifiziert werden können, um diese anschliessend zu verbessern.

[6] Database-centric architecture, https://en.wikipedia.org/wiki/Database-centric_architecture (Abgerufen: 30.04.2022)

Die folgenden Kapitel 6.1.1 und 6.1.2 beschreiben, wie Datenbank-Abfragen und Applikationslogik in der Posity IDE umgesetzt werden. Das dient im späteren Verlauf dem besseren Verständnis, an welchen Stellen Performance Messungen mit dem Artefakt durchgeführt werden.

6.1.1 Datenbank-Abfragen

Datenbank-Abfragen (engl. Queries) werden in Posity mit dem Query-Diagramm erstellt. Das Query-Diagramm bietet eine graphische Schnittstelle. Über diese können die im Datenmodell enthaltenen Entitätsmengen, respektive Tabellen für die Formulierung einer Abfrage miteinander verknüpft werden. Dabei können die in SQL bekannten SELECT- und JOIN-Operationen für die Verknüpfung und WHERE-Klauseln für das Filter der Daten angewendet werden. Die Abbildung 12 und Abbildung 13 zeigen anhand einer Tabelle Projekt, wie diese graphisch im Datenmodell modelliert ist und für eine Abfrage im Query-Diagramm verwendet werden kann.

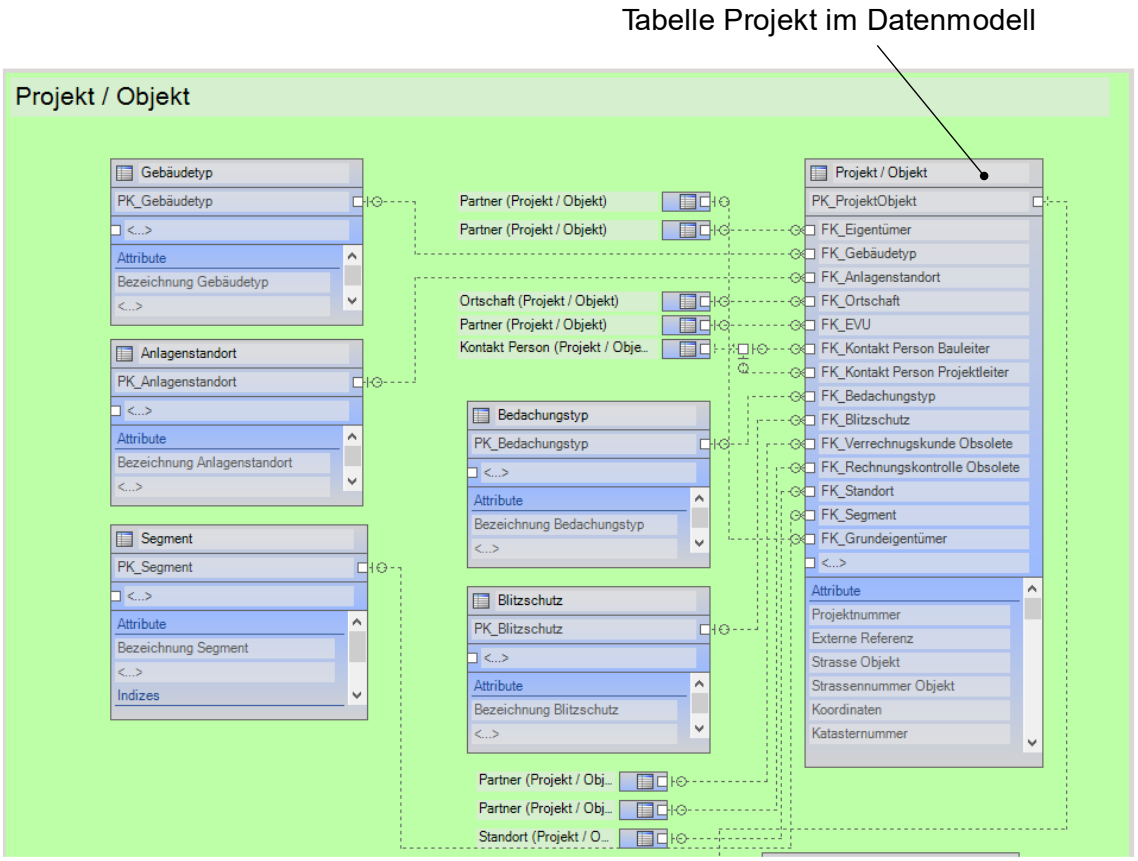


Abbildung 12: Beispiel Tabelle Projekt im Datenmodell (Quelle: Eigene Darstellung)

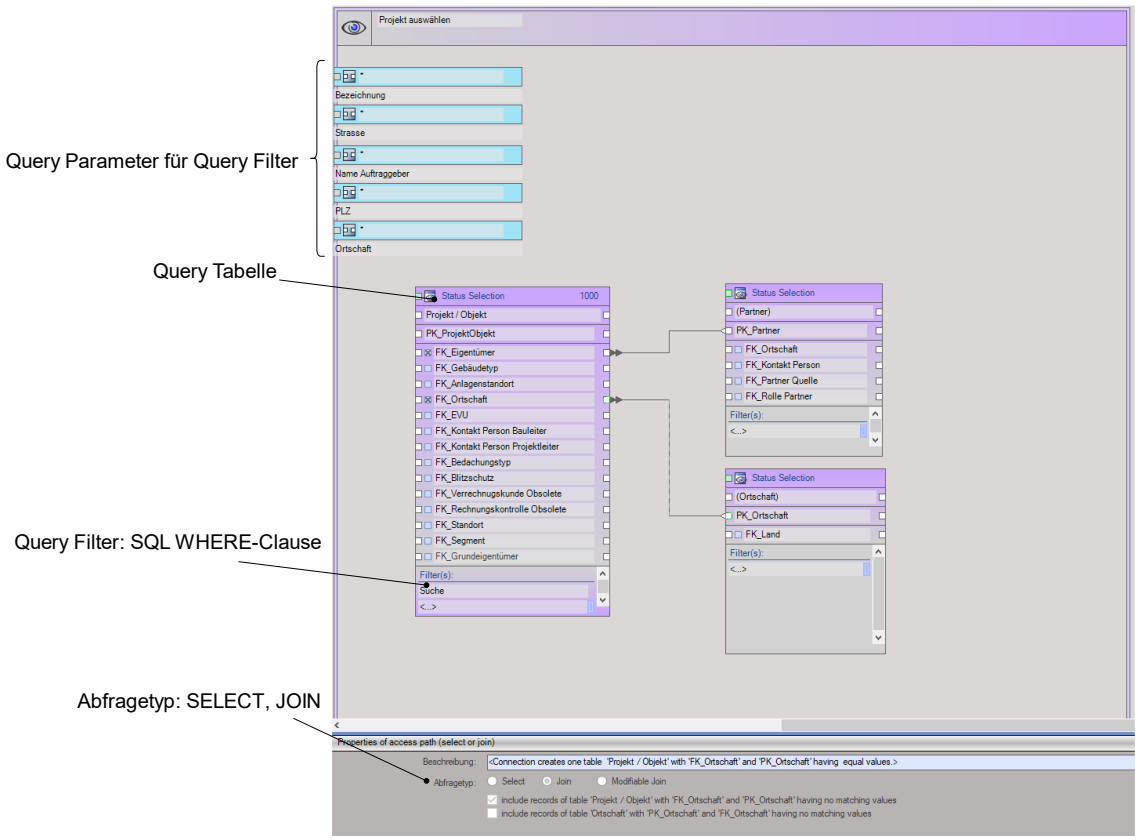


Abbildung 13: Beispiel Query-Diagramm (Quelle: Eigene Darstellung)

Um diese graphische Repräsentation einer Abfrage im Query-Diagramm zur Programmlaufzeit ausführen zu können, wird aus dieser zur Entwicklungszeit eine SQL-Stored Procedure generiert und auf dem Datenbank-Server gespeichert. Der vollständig generierte SQL-Stored Procedure-Code für das Query-Diagramm in Abbildung 13 ist im Anhang B ersichtlich. Zur Programmlaufzeit werden die Daten mit einem Aufruf der Stored Procedure durch die Programm-Logik beim Datenbank-Server bezogen und an die nächste Komponente im Programmablauf weitergereicht. Posity verwendet als Datenbank-Server den Microsoft SQL-Server[7] in den Versionen 2017 und 2019.

[7] Microsoft SQL-Server 2019, https://www.microsoft.com/en-us/sql-server/sql-server-2019 (Abgerufen: 30.04.2022)

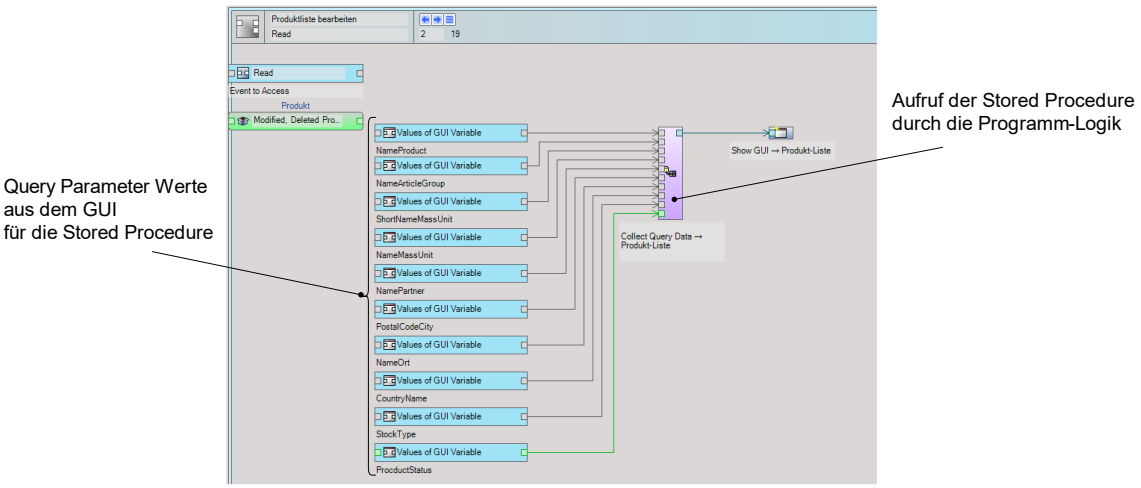


Abbildung 14: Beispiel Aufruf der Query-Diagramm Stored Procedure
(Quelle: Eigene Darstellung)

6.1.2 Applikationslogik

Die Applikationslogik wird in der Posity IDE im Modul-Diagramm abgebildet, das analog zum Query-Diagramm, eine graphische Benutzerschnittstelle bietet, mit dem die benötigten Komponenten für die Applikationslogik definiert und miteinander verknüpft werden können. Die Abbildung 15 zeigt eine Applikationslogik für das Neuberechnen der Mehrwertsteuer für ein bestimmtes Attribute und aktualisiert dieses auf der Benutzeroberfläche.

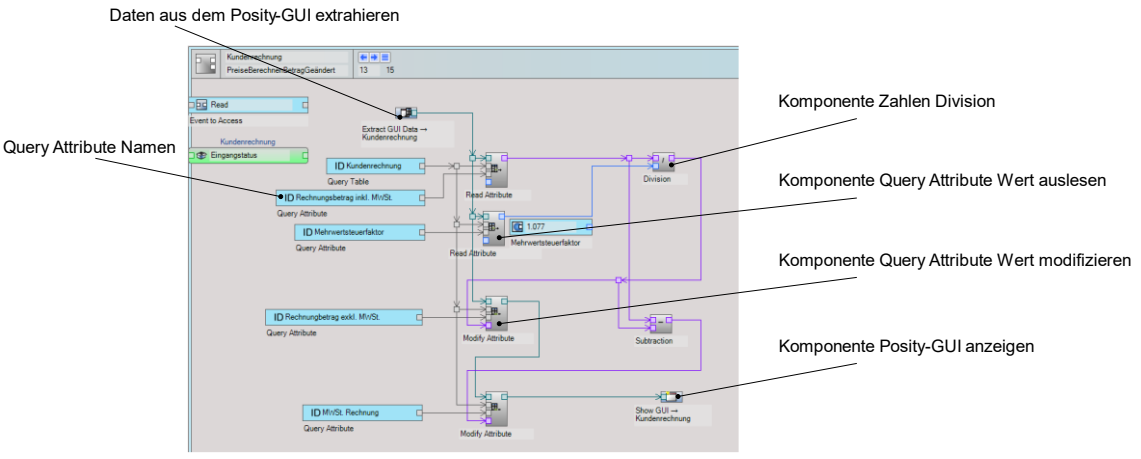


Abbildung 15: Beispiel Module-Diagramm Mehrwertsteuer berechnen
(Quelle: Eigene Darstellung)

Kontrollstrukturen, mit denen der Programmablauf beeinflusst werden kann, sind ebenfalls Komponenten. Wie in anderen Programmiersprachen, z.B. C#, Java oder Python, kann im Posity-Programmcode mit bekannten Kontrollstrukturen wie Schleifen (For, ForEach), Entscheidungen (If, Switch-Case), Try-Catch, usw., die benötigte Applikationslogik gebaut werden. Die Abbildung 16 zeigt die Kontrollstrukturen Sequence, ForEach und Switch-Case und wie diese ineinander verschachtelt werden können. Anmerkung: In der klassisch textuellen Programmierung ist die Programm-Sequenz durch die Anreihung von Befehlen in der Programm-Datei hintereinander implizit vorgegeben. In Posity muss, falls eine bestimmte Reihenfolge zwingend ist, eine Ausführungsreihenfolge explizit mit der Sequence-Kontrollstruktur definiert werden, weil sonst die Ausführungsreihenfolge von Komponenten im Module-Diagramm nicht vollständig deterministisch ist.

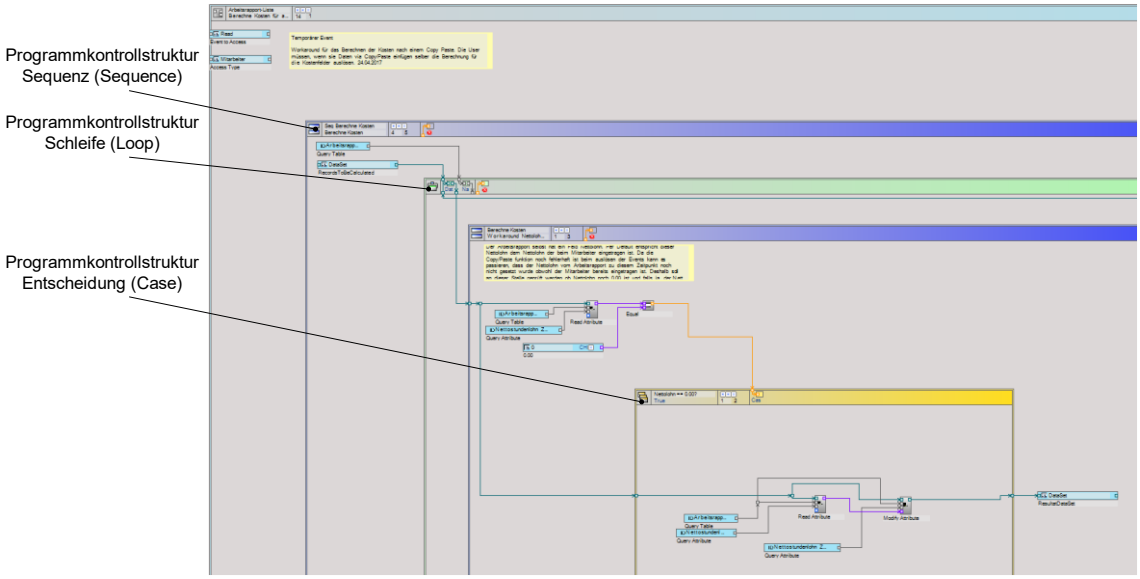


Abbildung 16: Modul-Diagramm Kontrollstrukturen (Quelle: Eigene Darstellung)

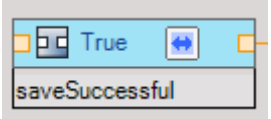
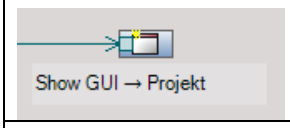
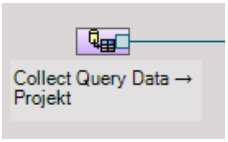
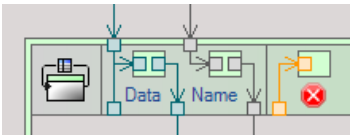
Posity bietet dutzende weitere Komponenten und Möglichkeiten für die Gestaltung einer Applikationslogik. Beispiele für weitere Komponenten sind Events, Posity-GUI Events, Sub-Module, Process-Call, Event-Call, usw. die, wenn nötig in den folgenden Kapiteln im Detail erklärt werden.

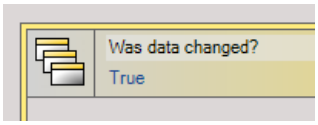
6.1.2.1 fCode

Ausführbar wird die im Module-Diagramm definierte Applikationslogik erst, wenn aus dieser der fCode (funktionaler Code) generiert wurde. Beim automatischen Generieren

von fCode, wovon der Entwickler keine Kenntnis nimmt, entsteht für jede graphische Komponente im Module-Diagramm ein textueller Programmcode (fCode).

Tabelle 3: Beispiele fCode Repräsentation von Module-Diagramm Komponenten

Module-Diagramm Komponente	**fCode**
Variable 	VarCom [VAriable COMponent]: 8c686394-2ed6-493c-b3bb-c0263ea3c18e SubTy [SUB TYpe]: VariableLocal DomTy [DOMain TYpe]: Boolean VarIn [VARiable INput port]: <null> VarOu [VARiable OUtput port]: 185 Value [VALUE]: True Unit [obsolete - UNIT (h=hours, m=minutes, etc.) for parameter+variable type duration]: IdMon [GuID of fkcurrencyofMONey]: f322d5d8-f2eb-4078-823f-78d60bb5098b IdVar [GuID of mouduleVARiable]: 37915668-4ab6-4a01-ae52-c8beee5c95b1 UsrDp [is USeR DePendent]: False LngDp [is LaNGuage DePendent]: False TmZon [TiMeZONe of dateTime field]: c0764e9f-c8c3-451d-ac87-90351ca0aae1
Posity-GUI anzeigen (ShowGUI) 	GuiCom [execute GUI COMponent (guid)]: 8d47f702-e981-42a6-a2bf-1641e3c0a496 SubTy [SUB TYpe]: Show GuiNa [GUI NAme (guid)]: 2de2d2ac-1d64-419f-8500-20dee25084da DatIn [nr of variable DATa set IN]: 181
Datenbank-Abfrage (Collect Query Data) 	QueCom [execute QUEry COMponent (guid)]: a5d6f1bf-c4ab-430d-a629-3ae49b570bdb SubTy [SUB TYpe]: CollectQueryData QueId [QUEry ID (guid)]: d1d2a22b-b9d9-4931-aecd-47dcd7a3438c QueNa [QUEry component fCode file NAme]: EM_3_ProjectEdit DatOu [nr of variable DATa set OUt]: 181
ForEach 	ForEch [FOREaCH structure]: a102ee2e-3e8c-4650-83af-1a6c2dcf954e DatIn [nr of variable DATa set IN]: 112 DatOu [nr of variable DATa set OUt]: 339 TblNa [TaBLe NaMe]: 236 TblIn [TaBLe INput port]: 340 BrkOu [BReaK OUt port]: 341 BrkTo [jump TO if BReaK]: 1159 BrkIn [BReaK INput port]: <null> DaSIn [INput DAta Set]: <null>
Switch-Case	SwitCh [SWITCH structure]: 09aaac25-95d2-4b34-8c9a-94492a4104b5 DftTo [if DeFaulT case jump TO]: -1

	BrkTo [jump TO if BReaK]: 1536 CseIn [INput port]: 185 CseTp [input port TyPe]: Boolean CsePt [CaSE ParT]: True:1531 CsePt [CaSE ParT]: False:1534

Der fCode wiederum wird zur Programmlaufzeit, respektive beim Starten einer in der Posity IDE erstellten Anwendung, von einer Posity Virtual Machine (PVM) interpretiert und die einzelnen fCodes abgearbeitet. Eine ähnliche Architektur für die Entwicklung von Anwendungen findet sich bei Java oder bei den .NET Sprachen (C#, F# und Visual Basic). Beim Kompilieren von Java- respektive .NET-Programmcode wird Bytecode für die Java Virtual Machine (JVM) (Oracle, 2017) oder die .NET Common Language Runtime (CLR) (Microsoft, 2021a) erstellt. Dies bietet den Vorteil eine Applikation ohne Anpassungen auf mehreren Plattformen (Linux, Windows, Mac OS) laufen lassen zu können, sofern für die Plattform eine JVM oder .NET CLR existiert. Denselben Ansatz verfolgt Posity mit dem fCode der Modul-Diagramme, um damit die Applikationslogik auf mehreren Plattformen ausführen zu können. Zurzeit ist eine PVM für Windows erhältlich und eine PVM für Web-Applikationen im Bau.

6.1.2.2 Posity Virtual Machine (PVM)

Ein Anwender kann zur Laufzeit die Business Prozesse, die in einem Prozess-Diagramm definiert und mit einem Modul verknüpft sind, ausführen. Beim Starten eines Posity-Prozesses wird der fCode von dem verknüpften Modul geladen, eine PVM als eigener Betriebssystem-Thread gestartet und die Komponenten im Start-Event von diesem Modul-Diagramm abgearbeitet. Die PVM bleibt so lange bestehen, bis alle durch diese PVM erzeugten Posity-GUIs geschlossen und alle fCodes abgearbeitet wurden. Mit dieser Architektur kann der Anwender mehrere Business-Prozesse parallel ausführen. Die Abbildung 17 zeigt den Zusammenhang zwischen einem Modul und einem Posity-Prozess, der in der Posity Anwendung vom Benutzer ausgeführt werden kann. Dies triggert schlussendlich die Abarbeitung vom fCode, des mit dem Posity-Prozess verknüpften Modules, durch eine PVM.

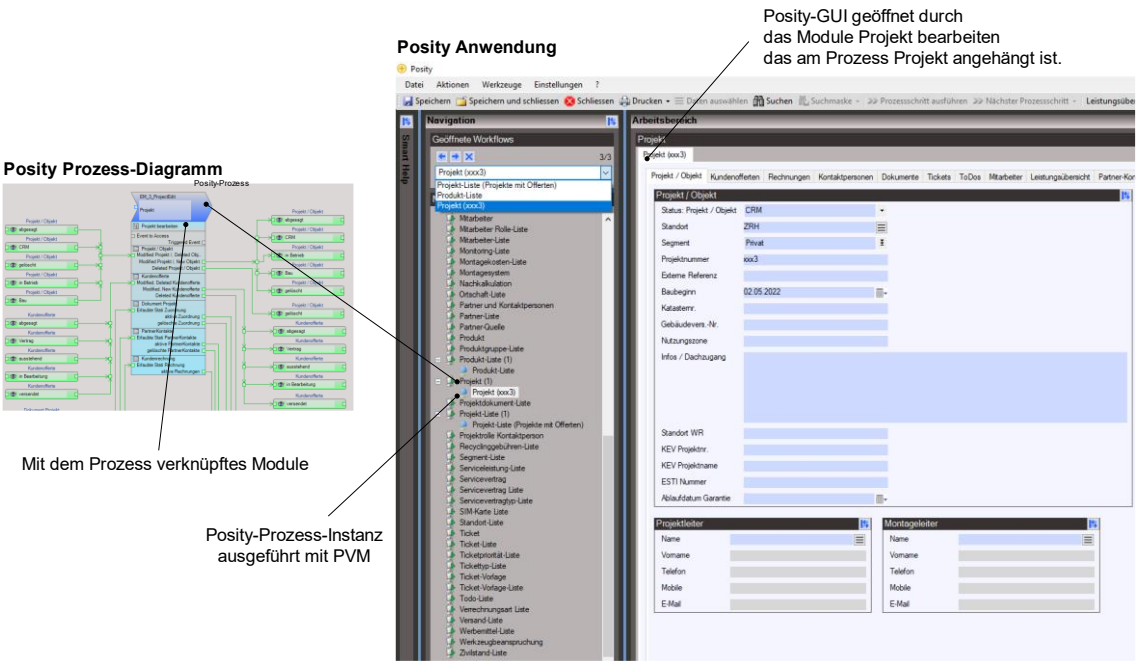


Abbildung 17: Zusammenhang Posity-Prozesse, Module und Posity Virtual Machines (Quelle: Eigene Darstellung)

6.2 Anforderungen

Diese Kapitel beschreibt die Anforderungen, die das Artefakt für die Performance-Analyse für die LCNC Entwicklungsumgebung Posity zu erfüllen hat, damit Performance-Messungen im Bereich der Datenbank-Abfragen und der Applikationslogik möglich sind.

6.2.1 Strukturelle Anforderungen

- Anpassungen am fCode sind, wenn möglich zu vermeiden.
- Erweiterungen der Diagramme sind, wenn möglich zu vermeiden.
- Es soll möglich sein Performance Daten für die Ausführung von fCode von allen Diagrammen (Module, Posity-GUI, Query) aufzeichnen zu können.

6.2.2 Nicht Funktionale Anforderungen

- Die Messung der Performance darf die Anwendungsausführung nicht einschränken.
- Vorhandene Performance-Analyse Tools der .NET CLR oder dem Microsoft SQL-Server sollen, wenn möglich mit eingebunden werden.
- Die Bedienung soll an bekannten Performance-Analyse-Tools ausgerichtet sein wie:

    - die WebDeveloper Tools in Firefox (Firefox Team, 2022), Chrome (Google Chrome Team, 2022) oder Edge (Microsoft Edge Team).
    - dotNet Trace, dotNet Memory (JetBrains, 2021b).
    - Microsoft Visual Studio Profiler (Microsoft, 2021b) .
    - Microsoft SQL Server Monitoring Tools (SQL-Docs Team, 2020).
- Die Ausführungsreihenfolge der Applikationslogik muss graphisch aufbereitet sein.

6.2.3 Funktionale Anforderungen

- Nur Entwickler sollten die Performance-Analyse Tools verwenden dürfen.
- Performance Messdaten sollen persistiert werden können.
- Performance Probleme sollen im Module-Diagramm respektive im Query-Diagramm dargestellt werden.
- Es sollte die Performance für die Applikationslogik aller parallel ausgeführten Prozesse gemessen werden.
- Es sollen die Ausführungszeiten, der Speicherverbrauch und der Datendurchsatz mit Peripherie-Systemen (Netzwerk, Datenbank, GPU, Lokale Harddisk) gemessen werden.

6.2.4 Abgrenzungen

- Die Anbindung der Performance-Analyse an eine CI-Pipeline wird im Moment nicht implementiert, wäre aber in Zukunft denkbar, wenn eine CI-Pipeline aufgebaut wird.
- Die Performance-Messdaten Aufzeichnung soll vom Entwickler nur lokal durchgeführt und nicht bei einem Kunden im Hintergrund ausgeführt werden. Die anfallende Datenmenge ist zu gross.
- Die Performance-Messdaten Aufzeichnung wird in der Posity IDE ausgelöst und nicht von einem externen Programm. Damit können die Stellen mit Performance Problemen besser direkt in den Posity-Diagrammen dargestellt werden.

6.3 Erfolgskriterien

In diesem Kapitel sind die Kriterien gelistet, nach denen das Artefakt in der Validierungsphase beurteilt wird. Nach der Auswertung der Erfolgskriterien können die eingangs aufgestellten Forschungsfragen beantwortet werden.

Tabelle 4: Erfolgskriterien Artefakt Performance-Analyse

Kriterium Performance-Analyse	**Beschreibung**
KPA1	Die Performance-Analyse kann generell für alle in einem Module-Diagramm erstellten Applikationslogiken angewandt werden.
KPA2	Das Artefakt unterstützt die Posity-Entwickler bei ihrer täglichen Arbeit.
KPA3	Ein Posity-Entwickler kann das Artefakt ohne Anleitung bedienen.
KPA4	Die Qualität einer mit der Posity IDE erstellen Applikation kann gesteigert werden.
KPA5	Das Artefakt kann so erweitert werden, dass es auch für weitere Posity Diagrammtypen gebraucht werden kann.
KPA6	Das Artefakt unterstützt die Basisfunktionalitäten gängiger Performance-Analyse Tools.
KPA7	Das Anwenden vom Artefakt führt **nicht** zu Performance-Einbrüchen der eigentlichen Applikation.
KPA8	Es kann festgestellt werden, wo in der Applikationslogik oder in welcher Datenbank-Abfrage Performance Probleme vorliegen.
KPA9	Das Artefakt läuft robust.
KPA10	Das Artefakt kann mit wenig Aufwand für weitere Posity-Entwickler verfügbar gemacht werden.

KPA11	Mit dem Artefakt lässt sich die Performance von Datenbank-Abfragen, die mit der No-Code Posity IDE erstellt wurden, messen.
KPA12	Mit dem Artefakt lässt sich die Performance von Applikationslogiken, die mit der No-Code Posity IDE erstellt wurde, messen.

6.4 Spezifikation

Dieses Kapitel spezifiziert die technische Umsetzung für die im Kapitel 6.2 gelisteten Anforderungen für die Implementation des Artefakts zur Performance-Analyse. Zudem werden die technischen Möglichkeiten und Rahmenbedingungen, die durch die Posity IDE und den verwendeten Datenbank-Server Microsoft SQL-Server vorgegeben sind und bei der Umsetzung eine Rolle spielen, erklärt.

6.4.1 Datenbank-Abfragen

Im Kapitel 6.1 und Kapitel 6.1.1 ist erklärt, wie Datenbank-Abfragen in der Posity IDE mit Hilfe des Query-Diagramms graphisch modelliert werden. Aus diesen Query-Diagrammen wird anschliessend pro Query-Diagramm eine SQL-Stored Procedure auf dem Microsoft SQL-Server erzeugt, welche eine Posity Applikation verwenden kann, um die Daten der im Query-Diagramm verwendeten Tabellen bei der Datenbank abzufragen. Die Abbildung 18 zeigt wie Datenbank-Abfragen generiert und wie später die Daten bei der Datenbank über die SQL-Stored Procedure vom Modul bezogen werden können.

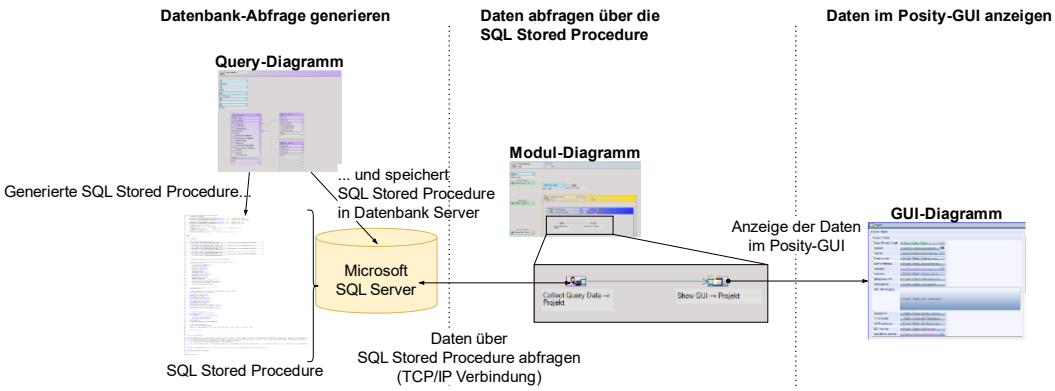


Abbildung 18: Kontextdiagramm Datenbank-Abfragen in Posity
(Quelle: Eigene Darstellung)

Ob die Daten in performanter Weise von der Datenbank auf die Benutzeroberfläche übertragen werden, hängt von mehreren Faktoren ab:

1. Enthält die SQL-Stored Procedure performanten SQL-Code?
2. Wie gross ist die zu übertragende Datenmenge?
3. Ist die Bandbreite der TCP/IP-Verbindung angemessen für die zu übertragenden Daten?
4. Fragt der Entwickler im Query-Diagramm nur benötigte Attribute einer Tabelle ab?
5. Fragt der Entwickler im Modul-Diagramm nicht mehrmals dieselben Daten ab?

Die Faktoren 1 und 3 kann der Posity-Entwickler nicht beeinflussen. Die Posity IDE ist verantwortlich für die Erzeugung von performantem SQL-Code und der Betreiber der Posity Applikation muss die nötige Bandbreite für die Verbindung zwischen der Posity Applikation und dem Datenbank-Server bereitstellen. Der Faktor 5 ist ein Teil der Applikationslogik, weshalb dieser eher mit einer Performance-Analyse der Applikationslogik bewertet werden kann. Es bleiben die Faktoren 2 und 4, die ein Posity-Entwickler im Rahmen von Datenbank-Abfragen beeinflussen kann. Faktor 4 kann durch Software nicht bewertet werden. Dieser hängt von der benötigten Business-Logik ab und welche Daten ein Kunde über die Benutzeroberfläche verwalten möchte. Aus Faktor 2 können aber Rückschlüsse gezogen werden, die allenfalls ein Indiz für Handlungsbedarf bei Faktor 4 sind. Zum Beispiel ist beim Feststellen von grossen übertragenen Datenmengen zu prüfen ob Bilder oder Dokumente im Query abgefragt werden und diese wirklich in jedem Fall benötigt werden.

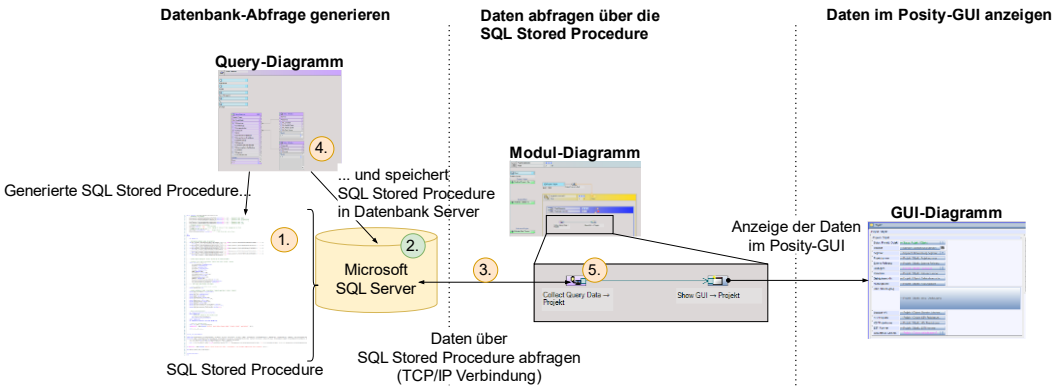


Abbildung 19: Einflussstellen der Faktoren der Datenbank-Abfragen Performance (Quelle: Eigene Darstellung)

Übrig bleibt Faktor 2 für den es gilt Performance-Messdaten zu erheben, um so dem Posity-Entwickler ein Mass zu geben, ob an der Stelle von Faktor 2 (siehe Abbildung 19), sprich im SQL-Server oder indirekt bei Faktor 4 im Query-Diagramm Anpassungen nötig sind, um die Performance zu verbessern. Als Messpunkte für die Performance-Messdaten eignen sich die in der Vorstudie zu dieser Arbeit (Marion Mürner & Matthias Christen, 2021, S. 12–13) beschriebenen Messpunkte CPU-Time, I/O-Throughput und Speicherbedarf. Diese Messpunkte sind im Microsoft SQL-Server für jede Query vorhanden und können über SQL-Server interne Tabellen bezogen werden. Das nächste Kapitel 6.4.1.1 beschreibt ausführlich welche Performance-Messdaten im Microsoft SQL-Server im Bereich der Datenbank-Abfragen zur Verfügung stehen und welche für das Artefakt Datenbank-Abfragen verwendet werden.

6.4.1.1 Microsoft SQL-Server Performance-Messdaten

Der MS SQL-Server bietet zwei Möglichkeiten Performance-Messdaten für SQL-Stored Procedures zu beziehen respektive zwei Orte, wo diese aufgezeichnet werden:

1. in der System View sys.dm_exec_procedure_stats
2. im Query Store

6.4.1.2 System View sys.dm_exec_procedure_stats

In der System View sys.dm_exec_procedure_stats (SQL-Docs Team, 2022b) können die vom MS SQL-Server aggregierten Performance-Messdaten für die im Cache abgelegten Ausführungspläne für SQL-Stored Procedures abgerufen werden. Ein Ausführungsplan einer SQL-Stored Procedure definiert die beste Vorgehensweise, wie auf die in der SQL-Stored Procedure verwendeten Tabellen zugegriffen wird. Im Ausführungsplan ist die Zugriffsreihenfolge auf die Tabellen, mit welcher Methode die Daten aus der Tabelle gelesen und mit welcher Methode z.B. die Daten aggregiert, sortiert oder Berechnungen durchgeführt wird, festgelegt. Die SQL-Stored Procedure-Ausführungspläne werden in einem Cache abgelegt, um das Ausführen der SQL-Stored Procedure nicht durch das Erstellen vom SQL-Stored Procedure-Ausführungsplan zu verzögern. Über die System View dm_exec_procedure_stats können also Performance-Messdaten für eine SQL-Stored Procedure und ihrem letzten generierten Ausführungsplan bezogen werden. Folgende Performance-Messdaten sind für das Artefakt Datenbank-Abfragen relevant:

Tabelle 5: System View dm_exec_procedure_stats

Attribute in der View	**Beschreibung**
Object_id	Aus der Object_id kann der Name der SQL-Stored Procedure hergeleitet werden.
type	Enthält den Wert ‘P’ wenn es sich um eine SQL-Stored Procedure handelt. Für das Artefakt irrelevant sind die Typen ‘PC’ (Assembly (CLR) Stored Procedure und ‘X’ (Extended SQL-Stored Procedure).
cached_time	Wann wurde die SQL-Stored Procedure zum Cache hinzugefügt.
execution_count	Wie viel Mal wurde die SQL-Stored Procedure ausgeführt.
min_elapsed_time	Minimum der benötigten Zeit [ms] für die SQL-Stored Procedure auszuführen.
max_elapsed_time	Maximum der benötigten Zeit [ms] für die SQL-Stored Procedure auszuführen.
total_elapsed_time	Total der Zeiten [ms] die für das Ausführen der SQL-Stored Procedure verwendet wurde. Zusammen mit execution_count kann eine durchschnittliche Ausführungszeit [ms] (avg_elapsed_time) errechnet werden.

Entwicklungshinweise:

- Die Performance-Messdaten in der System View sys.dm_exec_procedure_stats sind immer verfügbar, respektive das Aufzeichnen dieser muss nicht speziell aktiviert werden.
- Es werden nur SQL-Stored Procedures in der System View sys.dm_exec_procedure_stats angezeigt für die ein Ausführungsplan generiert wurde.
- Der Datenbank-Benutzer welcher Zugriff auf diese View möchte, benötigt das VIEW SERVER STATE Recht.

6.4.1.3 Query Store

Detailliertere Performance-Messdaten für SQL-Stored Procedures können im Microsoft SQL-Server mit Hilfe vom Query Store (SQL-Docs Team, 2022a) aufgezeichnet werden. Im Query Store wird über die Ausführung von Datenbank-Abfragen inklusive ihren Ausführungsplänen eine Statistik erfasst. Diese umfasst nicht nur Performance-Messdaten von SQL-Stored Procedures, sondern es sind zu allen Datenbank-Abfragen innerhalb einer SQL-Stored Procedure Performance-Messdaten verfügbar. Das object_id Attribute in der Query Store Tabelle sys.query_store_query gibt Aufschluss darüber welche Datenbank-Anfragen zu welcher SQL-Stored Procedure gehört. Zudem bietet der Query Store nicht nur eine aktuelle Sicht zu den sich im Cache befindenden Ausführungsplänen der Datenbank-Anfragen, sondern es werden auch Änderungen an den Ausführungsplänen festgehalten. Damit können Performance-Probleme, die mit Ausführungsplänen zusammenhängen, besser aufgedeckt werden. Der Query Store ist standardmässig nicht aktiviert und muss mit dem SQL-Befehl:

```
ALTER DATABASE <database_name> SET QUERY_STORE = ON;
```

für eine Datenbank vor dem Aufzeichnen von Performance-Messdaten aktiviert werden. Anschliessend können in den folgenden Stores Performance-Messdaten bezogen werden:

Tabelle 6: Übersicht der Query Stores

Query Store	**Beschreibung**
plan store	Enthält Informationen zu den verwendeten Ausführungsplänen der einzelnen Datenbank-Abfragen.
runtime stats store	Enthält Laufzeit Performance-Messdaten zu den einzelnen Datenbank-Abfragen.
wait stats store	Enthält eine Statistik wie lange einzelne Datenbank-Abfragen auf Ressourcen (z.B. CPU oder Harddisk) warten müssen.

Diese aufgezeichneten Daten können in eigenen Datenbank-Abfragen verwendet werden. Z.B. liefert folgende Abfrage die Datenbank-Abfragen mit der durchschnittlich längsten Ausführungszeit in der letzten Stunde (SQL-Docs Team, 2022a).

```
SELECT TOP 10 rs.avg_duration, qt.query_sql_text, q.query_id,
    qt.query_text_id, p.plan_id, GETUTCDATE() AS CurrentUTCTime,
    rs.last_execution_time
FROM sys.query_store_query_text AS qt
JOIN sys.query_store_query AS q
    ON qt.query_text_id = q.query_text_id
JOIN sys.query_store_plan AS p
    ON q.query_id = p.query_id
JOIN sys.query_store_runtime_stats AS rs
    ON p.plan_id = rs.plan_id
WHERE rs.last_execution_time > DATEADD(hour, -1, GETUTCDATE())
ORDER BY rs.avg_duration DESC;
```

Des Weiteren sind diese Daten im Query Store in eigenen Ansichten aufbereitet:

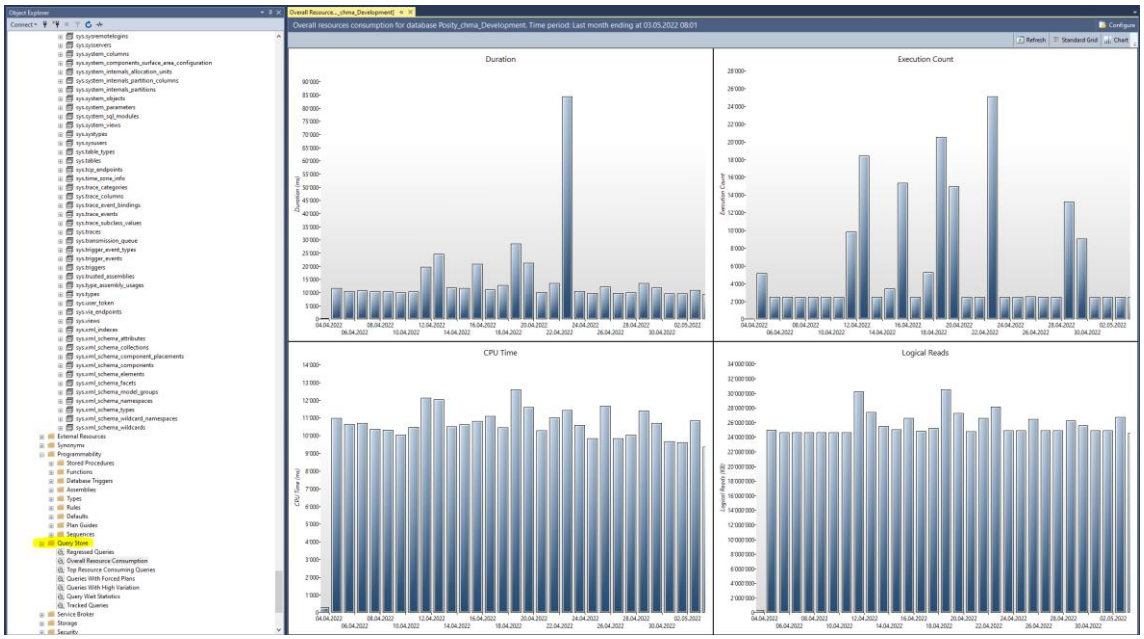


Abbildung 20: Microsoft SQL-Server Query Store (Quelle: Eigene Darstellung)

Entwicklungshinweise:

- Der Query Store muss vor Gebrauch für eine Datenbank aktiviert werden.

6.4.1.4 Technologie Entscheid Performance-Messdaten SQL-Server

Im Artefakt Datenbank-Abfragen soll die in Kapitel 6.4.1.2 System View sys.dm_exec_procedure_stats beschriebene technische Möglichkeit für die Performance-

Messdatenaufzeichnung für Datenbank-Abfragen ausfolgenden Gründen implementiert werden:

- Die System View sys.dm_exec_procedure_stats ist immer aktiv.
- Für eine erste Performance-Analyse für die in den Query-Diagrammen definierten Datenbank-Abfragen reicht die System View sys.dm_exec_procedure_stats vollständig aus.
- Der Query Store benötigt kompliziertere Datenbank-Abfragen und Einstellungen und somit mehr Zeit für die Implementation. Es soll mehr Zeit für das Artefakt Applikationslogik zu verfügen stehen, weil dieses einen höheren Zusatznutzen verspricht.
- Einige Perfomance-Messdaten aus dem Query Store sind schon im Microsoft SQL-Server Management Studio (Microsoft, 2022) grafisch aufbereitet, sodass es in diesem Artefakt nicht nötig ist, diese in die Posity IDE zu integrieren. Wenn erwünscht können die im Query Store vorhandenen Performance-Messdaten von einem Experten für eine detailliertere Performance-Analyse im Microsoft SQL-Server Management Studio direkt abgerufen werden.

Der Query Store soll im Artefakt nur aktiviert oder deaktiviert werden können. Im Anschluss an diese Masterarbeit sollen die Performance-Messdaten aus dem Query Store für die Posity-Entwickler in die Posity IDE eingebunden werden, um damit dem Posity-Entwickler detailliertere Performance-Messdaten zu den einzelnen Datenbank-Abfragen einer SQL-Stored Procedure direkt in der Posity IDE anzuzeigen.

6.4.2 Applikationslogik

In der Posity IDE wird die im Modul-Diagramm definierte Applikationslogik vor dem Ausführen in fCode übersetzt, der wiederum zur Laufzeit durch eine Posity Virtual Machine (PVM) interpretiert wird. Eine PVM wird als eigenständiger Betriebssystem-Thread ausführt, sodass ein Benutzer mehrere Prozesse, respektive jeweils das mit dem Prozess verknüpften Modul, parallel ausführen kann. Dies heisst konkret, man kann z.B. eine bestimmte Offerte in einer Listenansicht suchen und gleichzeitig im Hintergrund für eine andere Offerte das abgabefertige Offert-Dokument erstellen. Damit kann der Benutzer länger dauernde Prozesse im Hintergrund weiterlaufen lassen und sich parallel neuen Aufgaben widmen. Die Abbildung 21 zeigt schematisch, wie die Prozesselemente auf der Posity Benutzeroberfläche intern als Betriebssystem-Threads abgebildet werden.

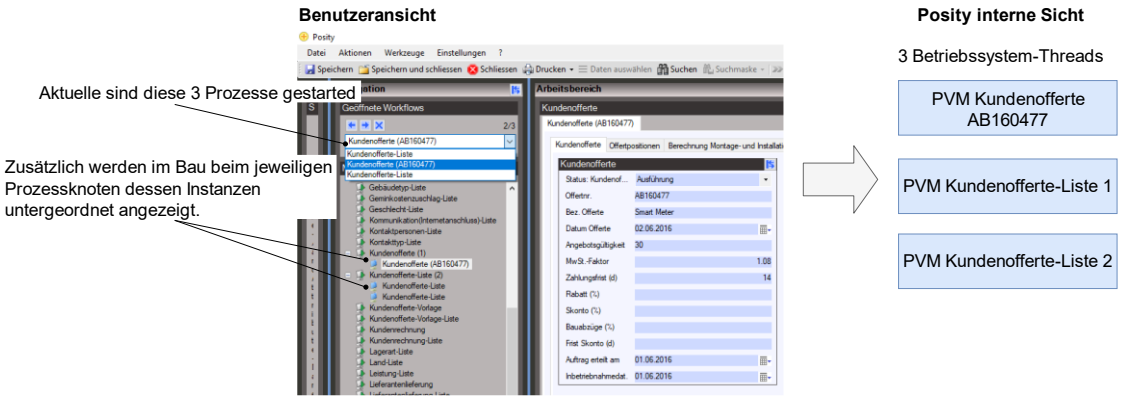


Abbildung 21: Benutzerprozesse und PVM Betriebssystem-Threads
(Quelle: Eigene Darstellung)

Performance-Messdaten über die Applikationslogik sind also in den einzelnen PVMs aufzuzeichnen und wegen der Parallelität dieser an einer zentralen Stelle abzulegen. Mit der Ablage der Performance-Messdaten an einer zentralen Stelle kann später die Ausführungsreihenfolge der Applikationslogik über alle ausgeführten Prozesse wieder hergeleitet werden.

Ob Daten in performanter Weise in der Applikationslogik verarbeitet werden, lässt sich nur schwer direkt aus dem Modul-Diagramm ablesen. Ein besseres Verfahren mögliche Performance-Probleme in der Applikationslogik aufzudecken, ist das direkte Ausführung der Applikationslogik und dabei die Ausführungszeiten der Logikblöcken festzuhalten. Für eine Performance-Messung von einem Modul-Diagramm müssen also die Ausführungszeiten für die jeweiligen fCodes aufgezeichnet werden. Im Artefakt Applikationslogik liegt der Fokus auf den Ausführungszeiten (CPU-Time). Bei Bedarf wird nach dieser Masterarbeit das Artefakt um die Performance-Messdaten I/O-Throughput und Speicherverbrauch erweitert.

6.4.2.1 Aufzeichnung fCode

Die Applikationslogik im Modul-Diagramm wird grundsätzlich aus 2 Logikbausteintypen gebaut:

Tabelle 7: Modul-Diagramm Logikbausteintypen

Logikbausteintyp	**Beispiele**
Funktionsbaustein	SubtractNumber, NumberToText, CollectQuery, ShowGui, usw.
Strukturbaustein	If-Else, Try-Catch, Sequence, Lane, ForEach, For, usw.

Funktionsbausteine können mit einem fCode Code abgebildet werden. Im Unterschied dazu müssen Strukturbausteine mit mehreren fCode Codes im fCode abgebildet werden.

Tabelle 8: Modul-Diagramm Logibaustein fCode Repräsentation

Logikbaustein	**fCode Repräsentation**

Funktionsbaustein NumberToText	`StdCom [execute STanDard COMponent]: 2c66cbf4-453b-49c4-a993-ab0268345815`
	`SubTy [SUB TYpe]: NumberToText`
	`NumIn [nr of variable NUMber IN]: 1`
	`TxtOu [nr of variable TeXT OUT]: 0`

Strukturbaustein If-Else	`SwitCh [SWITCH structure]: 4127de16-29e9-4372-9f75-a1711bf535a2`
	`DftTo [if DeFaulT case jump TO]: -1`
	`BrkTo [jump TO if BReaK]: 25`
	`CseIn [INput port]: 3`
	`CseTp [input port TyPe]: Boolean`
	`CsePt [CaSE ParT]: True:20`
	`CsePt [CaSE ParT]: False:23`
	
	`// Hier steht normalerweise der fCode für den If-Teil vom If-Else Konstruct`
	
	`JumpTo [JUMP TO fCode row]: 71dde703-85d9-4f82-b5bf-018060d1333e`
	`JmpNr [fCode row NR]: 25`
	
	`// Hier steht normalerweise der fCode für den Else-Teil vom If-Else Konstruct`

	JumpTo [JUMP TO fCode row]: 71dde703-85d9-4f82-b5bf-018060d1333e JmpNr [fCode row NR]: 25 25: nächste fCode-Zeile
Strukturbaustein For	52: ForLop [FOR LOoP structure]: d1581260-74a0-48a7-8065-c8104ad40a06 MinIn [INport MINimum]: 6 MaxIn [INport MAXimum]: 7 CntOu [CouNTer OUt port]: 11 BrkOu [BReaK OUt port]: 12 BrkTo [jump TO if BReaK]: 45 BrkIn [BReaK INput port]: <null> // Hier steht normalerweise der fCode für die Logikbausteine innerhalb des For-Loops 54: JumpTo [JUMP TO fCode row]: d1581260-74a0-48a7-8065-c8104ad40a06 JmpNr [fCode row NR]: 52

Für die Berechnung von der CPU-Time für einen Funktionsbaustein reicht es vor und nach dem Ausführen des Funktionsbaustein die Zeit festzuhalten und die Differenz $t_{NACH} - t_{VOR}$ ergibt die Ausführungszeit t_{EXE} für den Funktionsbaustein. Diese Messmethode, respektive Formel gilt auch für die Strukturbausteine, nur werden zwischen t_{VOR} und t_{NACH} mehrere fCodes ausgeführt. Zusätzlich zu t_{VOR} und t_{NACH} muss die Identifikation für den Logikbaustein festgehalten werden. Hierzu genügt die im fCode-Code-Header stehende GUID. Damit kann später, z.B. direkt von der Benutzeroberfläche für die Performance-Messdaten, zum jeweiligen Logikbaustein im Modul-Diagramm gesprungen werden. Für die Identifikation des Modul-Diagramms muss die GUID vom Modul-Diagramm gespeichert werden. Weil dasselbe Modul mit unterschiedlichen Prozessen verknüpft sein kann, ist es erforderlich auch die GUID des Prozesses zu erfassen.

Für die Anzeige vom Logikbaustein in der Benutzeroberfläche der Performance-Messdaten wird im Artefakt vorerst der fCode vom Logikbaustein als Beschreibungstext gespeichert. Das sieht etwas kryptisch aus, aber ist performanter als zusätzlich noch die Beschreibungstexte mitaufzuzeichnen. Ein anderer Ansatz wäre die Beschreibungstexte aus dem geladenen Modul-Diagramm direkt bei den Logikbausteinen auszulesen. Mit der

aufgezeichneten GUID vom Logikbaustein lässt sich dessen Beschreibungstext im Modul-Diagramm ermitteln.

Die Ausführungsreihenfolge ist gegeben durch das sequenzielle Eintragen in den Performance-Messdatenspeicher. Trotzdem soll eine Ausführungsnummer gespeichert werden, um auch ohne Messdatenspeicher die Ausführungsreihenfolge reproduzieren zu können. Ein Zeitstempel kann nicht verwendet werden, um die Ausführungsreihenfolge festzuhalten, weil mehrere Logikbausteine pro Tick respektive pro 100 Nanosekunden in der höchsten Auflösung (1 Tick = 100 Nanosekunden[8]) vom Zeitstempel ausgeführt werden. Zusammenfassend müssen pro Logikbaustein folgende Performance-Messdaten aufgezeichnet werden:

Tabelle 9: Übersicht Performance-Messdaten Logikbaustein

Performance-Messdaten Logikbaustein	**Beschreibung**
GUID vom Logikbaustein	Damit kann der Logikbaustein zu einem späteren Zeitpunkt im Modul-Diagramm wieder gefunden werden.
GUID vom Modul in dem sich der Logikbaustein befindet	Damit wird das Modul eindeutig identifiziert in dem sich der Logikbaustein befindet.
GUID vom Prozess welcher das Modul ausgeführt hat.	Damit kann der Prozess, welcher die Ausführung vom Modul bewirkt hat, identifiziert werden.
t_{VOR}	Zeit (UTC) vor der Ausführung des Logikbausteines.
t_{NACH}	Zeit (UTC) nach der Ausführung des Logikbausteines t_{NACH}.

[8] C# Hinweise TimeSpan Objekt, https://docs.microsoft.com/de-de/dotnet/api/system.timespan?view=net-6.0, (Abgerufen: 10.05.2022)

Beschreibung von Logikbaustein	fCode vom Logikbaustein als Beschreibungstext für die Anzeige auf der Benutzeroberfläche der Performance-Messdaten.

6.4.2.2 Problem Strukturbaustein Ende-fCode

Ein Problem beim Messen der Ausführungszeiten von Strukturbausteinen ist das Fehlen einer Markierung für das Ende eines Strukturbausteines. In der Tabelle 8 ist ersichtlich, dass das Ende von einem If-Else- oder dem For-Strukturbaustein nur mit einem JumpTo fCode implizit definiert ist und nicht explizit über einen Ende-fCode wie z.B. IfEnd respektive ForEnd. Dies bedeutet man muss zur Laufzeit auf einem Stack den Start von einem Strukturbausteinen festhalten und wenn ein JumpTo fCode verarbeitet, wird mit Hilfe dieses Stacks das Ende eines Strukturbausteines ermitteln (Der JumpTo fCode markiert das Ende von dem obersten auf dem Stack liegenden Strukturbaustein.). Dieses zusätzliche Führen vom Stack und Ermitteln der Enden von Strukturbausteinen würde die Ausführungszeit der Applikationslogik zu fest beeinflussen und die Performance-Messdaten verfälschen. Deshalb muss für jeden Strukturbaustein eine neuer Ende-fCode Code eingeführt werden z.B. ForEnd, TryEnd, usw. Damit werden beim Aufzeichnen der ausgeführten Applikationslogik automatisch auch die Enden der Strukturen in den Performance-Messdaten abgelegt und bei der Auswertung der aufgezeichneten Performance-Messdaten kann die genaue Ausführungsreihenfolge der Applikationslogik wiederhergestellt werden.

6.4.2.3 Persistieren der fCode Performance-Messdaten

Die Aufzeichnung der Performance-Messdaten aus Tabelle 9 soll ähnlich wie die Aufzeichnung von einem Video auf einem Mobiltelefon funktionieren. Mit einem Start-Button wird das Aufzeichnen gestartet und mit einem Stopp-Button beendet. Zwischen Start und Stopp kann der Benutzer die Posity Applikation wie gewohnt bedienen und dabei werden für die ausgeführten Logikbausteine Performance-Messdaten erfasst. Die aufgezeichneten Performance-Messdaten sollen in internen Datenstrukturen gehalten werden, woraus nachdem Beenden einer Aufzeichnung direkt eine graphische Darstellung erstellt werden kann. Eine neue Aufzeichnung verwirft die aktuelle in den Datenstrukturen gehaltenen Performance-Messdaten einer vorangehenden Aufzeichnung. Um die Perfor-

mance-Messdaten zu sichern, um diese damit zu einem späteren Zeitpunkt wieder einsehen oder zwei Performance-Messungen vergleichen zu können, sollen diese in der Umgebungsdatenbank (z.B. Test, Development, usw.) einer Posity Applikation in der gemessen wird, abgelegt werden können. Alle Performance-Messdaten der Logikbausteine sollen einer Aufzeichnung zugeordnet und die Aufzeichnung benannt werden können. Beim Entwickeln des Datenmodels und der Datenstrukturen soll zudem darauf geachtet werden, dass auch die Ausführung von fCode aus anderen Posity-Diagrammen (Query, Prozess, Posity-GUI) aufgezeichnet werden kann. Dies bietet zu einem späteren Zeitpunkt die Möglichkeit Performance-Messdaten auch in anderen Teilen einer Posity-Applikation aufzuzeichnen und zu sichern.

6.4.2.4 Darstellung Performance-Messdaten

Die sequenziell abgelegten Performance-Messdaten repräsentieren den Call-Stack[9] (deu. Aufrufstapel) von allen ausgeführten Logikbausteine in einer Performance-Messung. Für die Darstellung der hierarchischen Daten von einem Call-Stack eignen sich folgende Darstellungsformen:

- Flame Graphs
- Call-Tree

6.4.2.5 Flame Graphs

Mit Flame Graphs (Brendan, 2022) können hierarchische Daten aus Call-Stacks so visualisiert werden, dass sehr schnell die meisten aufgerufenen Programm-Pfade und Funktionen, respektive in Posity die Logikbausteine, ersichtlich sind. Auf der X-Achse wird die Ausführungszeit (CPU-Time) der aufgerufenen Funktion und auf der Y-Achse durch welche Funktion eine Funktion aufgerufen wurde dargestellt. Dadurch entsteht eine graphische Darstellung vom Call-Stack wie es in Abbildung 22 zu sehen ist.

[9] Call-Stack, https://en.wikipedia.org/wiki/Call_stack, (Abgerufen: 10.05.2022)

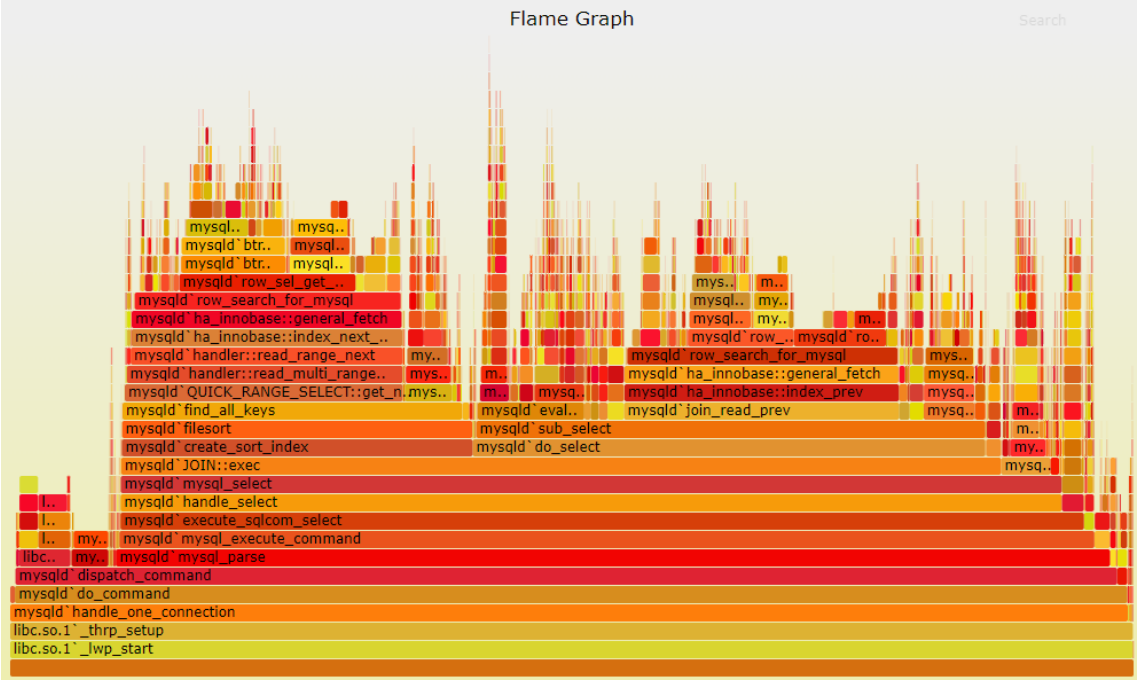


Abbildung 22: Beispiel Flame Graph (Quelle: Brendan's site: Flame Graphs, https://www.brendangregg.com/flamegraphs.html, Abgerufen: 10.05.2022)

In den Webbrowser-Entwickler-Tools von Firefox, Chrome und Edge werden Flame Graphs für die Darstellung von Performance-Messdaten von Webseiten verwendet. Sie sind 180° gedreht, sodass die Funktionsaufrufe nach unten abgebildet werden.

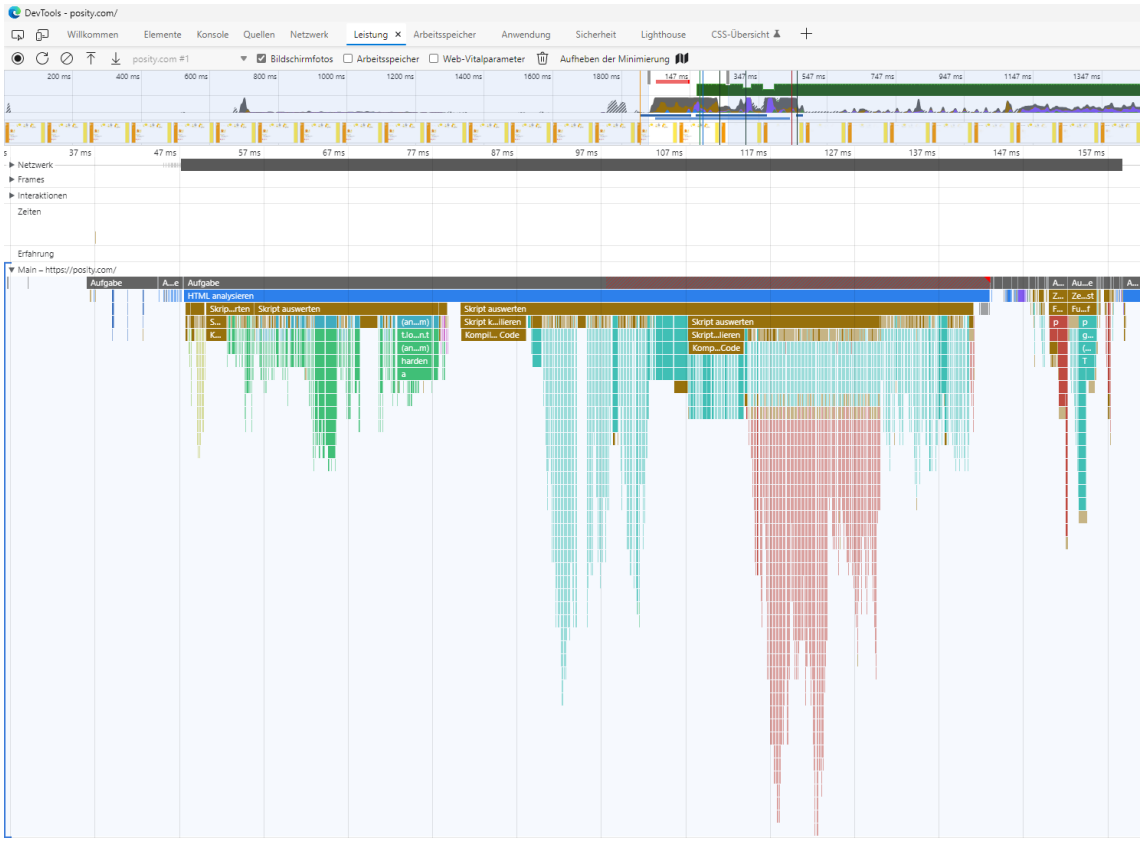


Abbildung 23: Beispiel Flame Graph von Posity.com im Webbrowser Edge (Quelle: Eigene Darstellung)

6.4.2.6 Call-Tree

In einem Call-Tree wird die Hierarchie der Funktionsaufrufe vom Call-Stack in einer Baum-Ansicht abgebildet, zum Beispiel mit dem C# TreeView Control[10]. Dabei werden die Funktionen als Knoten im Baum abgebildet. Ruft eine Funktion eine andere Funktion auf, wird die aufgerufene Funktion als Kind-Knoten beim Knoten der aufrufenden Funktion (Eltern-Knoten) eingefügt. Daraus ist dann die Aufrufreihenfolge der einzelnen Funktionen ersichtlich. Zusätzlich wird bei jedem Knoten im Baum die Ausführungszeit (CPU-Time) vom aktuellen Knoten, plus die der Kind-Knoten und zu wieviel Prozent diese Ausführungszeit, der vom Eltern-Knoten entspricht, angezeigt. Die Abbildung 24 zeigt einen Call-Tree vom Login-Vorgang in der Posity-Applikation im Performance Tracing Tool dotTrace (JetBrains, 2021b).

[10] C# TreeView Control, https://docs.microsoft.com/en-us/dotnet/api/system.windows.forms.treeview?view=windowsdesktop-6.0, (Abgerufen: 10.05.2022)

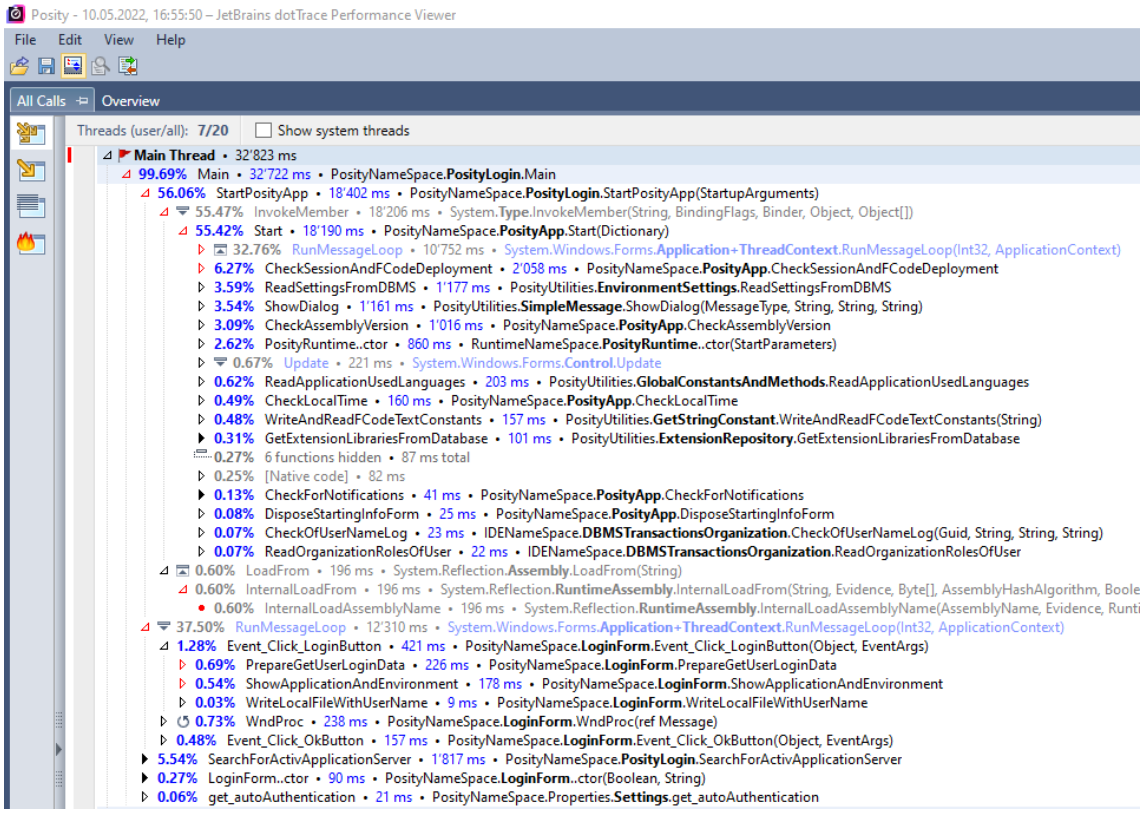


Abbildung 24: Call-Tree von Call-Stack vom Posity Login Vorgang
(Quelle: Eigene Darstellung)

6.4.2.7 Entscheid Darstellungsform für den Call-Stack

Wie in Kapitel 6.4.2.5 und 6.4.2.6 beschrieben eigenen sich prinzipiell beide Darstellungsformen für eine graphische Darstellung für den Call-Stack. Optimal wäre beide im Artefakt zu implementieren und somit auch auswerten zu können, welche sich besser in welchen Anwendungsfällen eignet. Aus Zeitgründen soll aber nur der Call-Tree im Artefakt umgesetzt werden. Erfahrungsgemäss benötigt eine eigene Implementation einer graphischen Darstellung von Informationen um ein Vielfaches mehr an Zeit, als wenn bereits eine vorgefertigte Darstellungsmöglichkeit, wie es das TreeView-Control von C# für den Call-Stack bieten würde, verwendet werden kann. Infolgedessen soll im Artefakt der Call-Stack mit dem C#-TreeView-Control abgebildet werden.

6.5 Implementation

Dieses Kapitel beschreibt und erklärt wie aus den im Kapitel 6.4 spezifizierten Anforderungen und Rahmenbedingungen das Artefakt Performance-Analyse umgesetzt wurde.

6.5.1 Datenbank-Abfragen

Wie in Kapitel 6.4.1.4 Technologie Entscheid Performance-Messdaten SQL-Server beschrieben sollen die Performance-Messdaten von Datenbank-Abfragen über die SQL-Server System View sys.dm_exec_procedure_stats bezogen werden. In Posity werden generell keine Datenbank-Abfragen direkt im C# formuliert, sondern die Abfrage der Daten geschieht immer über einen Aufruf einer SQL-Stored Procedure, in der die eigentliche Abfrage definiert ist. Nach diesem Vorgehen wurde auch eine SQL-Stored Procedure für den Zugriff auf die System View sys.dm_exec_procedure_stats erstellt, die dann die Performance-Messdaten der Datenbank-Abfragen zurück liefert.

6.5.1.1 Query-Diagramm Namen ermitteln

In der Benutzeroberfläche möchte der Posity-Entwickler den Namen des Query-Diagrammes sehen zu welchem die angezeigte Performance-Messdaten gehören. In der System View `sys.dm_exec_procedure_stats` ist aber nicht direkt der Name des Query-Diagrammes vorhanden, sondern nur der Name der SQL-Stored Procedure vom Query-Diagramm. Deshalb muss aus dem Namen der SQL-Stored Procedure, der Namen des Query-Diagrammes indirekt ermittelt werden. Die aus Query-Diagrammen generierten SQL-Stored Procedures sind im Datenbank-Schema UserData abgelegt und haben als Namen folgendes Format:

```
UserData.Query_[MetaData.Query.PhysicalQueryName]_Fill
```

Der Wert PhysicalQueryName ist in der Metadaten-Tabelle für die Query `MetaData.Query` (diese Tabelle enthält alle Informationen zu einem Query-Diagramm) beim Datensatz für das entsprechende Query-Diagramm abgelegt. Mit dem Primärschlüssel vom Datensatz aus der `MetaData.Query` Tabelle lässt sich in der Metadaten-Tabelle `MetaData.QueryLanguage` der Namen, mit welchem der Posity-Entwickler ein Query-Diagramm beschriftet hat, ermitteln.

6.5.1.2 SQL-Stored Procedure SystemData.QueryPerformance_Fill

Mit der Stored Procedure `SystemData.QueryPerformance_Fill` werden aus der Microsoft SQL-Server System View `sys.dm_exec_procedure_stats` die Performance-Messdaten für die Datenbank-Abfragen der Query-Diagramme zusammengetragen und an die Posity IDE übermittelt. Sie kann ebenfalls mit den SQL-Stored Procedure

Parametern `@SchemaName` und `@StoredProcedureNameFilter` für beliebig andere Datenbank-Abfragen verwendet werden, respektive SQL-Stored Procedures Performance-Messdaten abzufragen, z.B. aus dem `MetaData` Schema. Mit dem folgenden SQL Statement werden die Performance-Messdaten abgefragt:

```
SELECT sch.[name] + '.' + OBJECT_NAME( procstats.object_id ) As
ProcedureName , REPLACE(REPLACE(TRIM(OBJECT_NAME( obj.object_id )),
'Query_', ''), '_Fill', ''), [sql_handle] , cached_time,
execution_count, min_elapsed_time, max_elapsed_time,
total_elapsed_time, total_elapsed_time / execution_count As
avg_elapsed_time FROM sys.dm_exec_procedure_stats procstats
       JOIN sys.objects obj ON obj.object_id = procstats.object_id
       JOIN sys.schemas sch ON sch.schema_id = obj.schema_id
     WHERE sch.[name] = @SchemaName AND OBJECT_NAME(
procstats.object_id ) LIKE @StoredProcedureNameFilter
```

Anschliessend wird über jeden der zurückgegebenen Datensätze iteriert und versucht den logischen Query Namen zu finden:

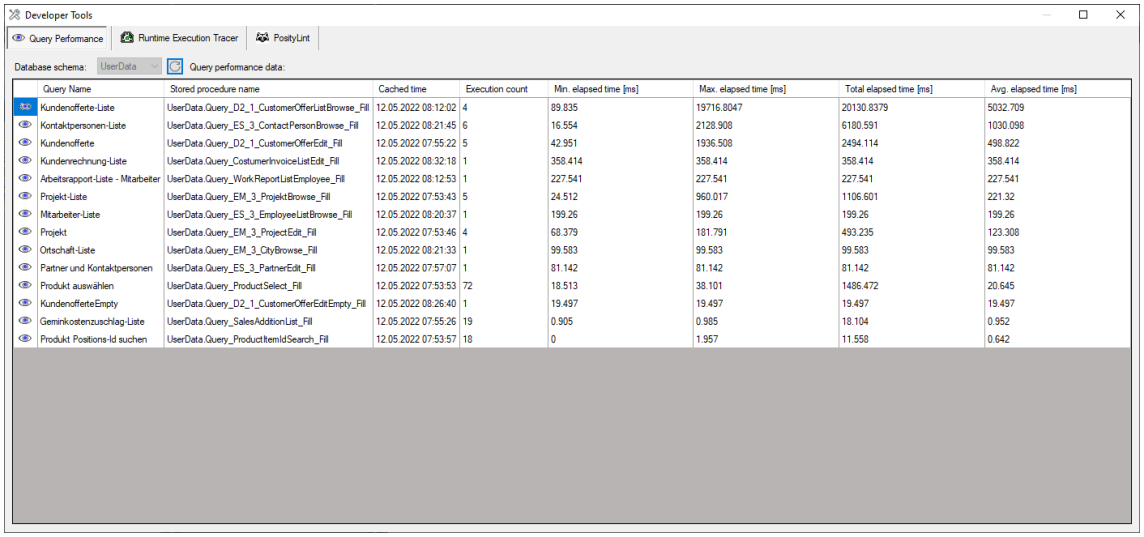
```
-- try to find the meta data for the query by its physical query name
     SELECT @PKQuery = PKQuery FROM [MetaData].[Query] AS q WHERE
q.PhysicalQueryName = @PhysicalQueryName;
     IF @@ROWCOUNT > 0
     BEGIN
       -- if found use the logical query name from the query language
table
       SELECT TOP 1 @LogicalQueryName = LogicalQueryName FROM
[MetaData].[QueryLanguage] AS ql WHERE ql.PKQueryLanguage = @PKQuery
AND TRIM(ql.FKLanguageOfQueryLanguage) = @language1;
       IF @@ROWCOUNT = 0
       BEGIN
         -- try to find a logical name in the second language of the
user
         SELECT TOP 1 @LogicalQueryName = LogicalQueryName FROM
[MetaData].[QueryLanguage] AS ql WHERE ql.PKQueryLanguage = @PKQuery
AND TRIM(ql.FKLanguageOfQueryLanguage) = @language2;
       END
       ELSE
       BEGIN
         -- no logical name in the second language of the user found >
try to find a logical name in any name
```

```
            SELECT TOP 1 @LogicalQueryName = LogicalQueryName FROM
[MetaData].[QueryLanguage] AS ql WHERE ql.PKQueryLanguage = @PKQuery;
            IF @@ROWCOUNT = 0
            BEGIN
               SELECT @LogicalQueryName = 'This query is missing a
language entry!';
            END
         END
      END
      ELSE
      BEGIN
         SELECT @LogicalQueryName = 'Has no logical query name';
      END
```

Der vollständige SQL-Code für die SQL-Stored Procedure befindet sich im Anhang C.

6.5.1.3 Benutzeroberfläche

Schlussendlich werden die aus der `SystemData.QueryPerformance_Fill` zurückgegebenen Daten auf einer eigenen DevTool-Page namens Query Performance (siehe Kapitel 5) dargestellt. Der Benutzer kann im Moment die Daten nur aktualisieren. Andere Interaktionsmöglichkeiten gibt es im DevTool Query Performance nicht. Die Abbildung 25 zeigt, wie die Query Performance-Messdaten auf der Query Performance DevTool-Page dargestellt werden.



Query Name	Stored procedure name	Cached time	Execution count	Min. elapsed time [ms]	Max. elapsed time [ms]	Total elapsed time [ms]	Avg. elapsed time [ms]
Kundenofferte-Liste	UserData.Query_D2_1_CustomerOfferListBrowse_Fill	12.05.2022 08:12:02	4	89.835	19716.8047	20130.8379	5032.709
Kontaktpersonen-Liste	UserData.Query_E5_3_ContactPersonBrowse_Fill	12.05.2022 08:21:45	6	16.554	2128.908	6180.591	1030.098
Kundenofferte	UserData.Query_D2_1_CustomerOfferEdit_Fill	12.05.2022 07:55:22	5	42.951	1936.508	2494.114	498.822
Kundenrechnung-Liste	UserData.Query_CostumerInvoiceListEdit_Fill	12.05.2022 08:32:18	1	358.414	358.414	358.414	358.414
Arbeitsrapport-Liste - Mitarbeiter	UserData.Query_WorkReportListEmployee_Fill	12.05.2022 08:12:53	1	227.541	227.541	227.541	227.541
Projekt-Liste	UserData.Query_EM_3_ProjektBrowse_Fill	12.05.2022 07:53:43	5	24.512	960.017	1106.601	221.32
Mitarbeiter-Liste	UserData.Query_E5_3_EmployeeListBrowse_Fill	12.05.2022 08:20:37	1	199.26	199.26	199.26	199.26
Projekt	UserData.Query_EM_3_ProjectEdit_Fill	12.05.2022 07:53:46	4	68.379	181.791	493.235	123.308
Ortschaft-Liste	UserData.Query_EM_3_CityBrowse_Fill	12.05.2022 08:21:33	1	99.583	99.583	99.583	99.583
Partner und Kontaktpersonen	UserData.Query_E5_3_PartnerEdit_Fill	12.05.2022 07:57:07	1	81.142	81.142	81.142	81.142
Produkt auswählen	UserData.Query_ProductSelect_Fill	12.05.2022 07:53:53	72	18.513	38.101	1486.472	20.645
KundenofferteEmpty	UserData.Query_D2_1_CustomerOfferEditEmpty_Fill	12.05.2022 08:26:40	1	19.497	19.497	19.497	19.497
Gemeinkostenzuschlag-Liste	UserData.Query_SalesAdditionList_Fill	12.05.2022 07:55:26	19	0.905	0.985	18.104	0.952
Produkt Positions-Id suchen	UserData.Query_ProductItemIdSearch_Fill	12.05.2022 07:53:57	18	0	1.957	11.558	0.642

Abbildung 25: Benutzeroberfläche Query Performance DevTool
(Quelle: Eigene Darstellung)

Für jeden Datensatz aus der `SystemData.QueryPerformance_Fill`-SQL-Stored Procedure wird ein Objekt der Klasse `QueryPerfInfo` erstellt und in einem Objekt der Klasse `QueryPerfInfoList` abgelegt. Das Objekt `QueryPerfInfoList` wird bei einer C#-DataGridView dem `DataSource` Attribute zugewiesen, womit alle `QueryPerfInfo`-Objekte im DataGridView angezeigt werden. Folgende Methode zeigt das Abfragen der `SystemData.QueryPerformance_Fill` und das Erzeugen der Objekte vom Typ `QueryPerfInfo`:

```
/// <summary>
        /// Find performance information for SQL-Stored Procedures
with a specific name in a specific database schema
        /// </summary>
        /// <param name="schemaName">Return only SQL-Stored Procedure
in this schema</param>
        /// <param name="storedProcedureNameFilter">Return only SQL-
Stored Procedure which match the stored procedure name filter</param>
        /// <param name="language1">First language of the user to
return the logical name of the query which belongs to the found SQL-
Stored Procedure</param>
        /// <param name="language2">Second language of the user to
return the logical name of the query which belongs to the found SQL-
Stored Procedure</param>
        private QueryPerfInfoList FindQueryPerformanceInfos(string
schemaName, string storedProcedureNameFilter, string language1, string
language2)
        {
            SqlDataAdapter dataAdaptor = new SqlDataAdapter();
            DataSet dataSet = new DataSet();
            QueryPerfInfoList listOfQueryPerfInfos = new
QueryPerfInfoList();

            string logicalQueryName;
            string procedureName;
            string cachedTime;
            int execCount;
            int minElapsedTime;
            int maxElapsedTime;
            int totalElapsedTime;
            int avgElapsedTime;
            QueryPerfInfo queryPerfInfo;
```

```
            dataAdaptor.SelectCommand = new
SqlCommand("[SystemData].[QueryPerformance_Fill]");
            dataAdaptor.SelectCommand.Parameters.Clear();
            dataAdaptor.SelectCommand.CommandType =
CommandType.StoredProcedure;
            dataAdaptor.SelectCommand.Connection =
DataBaseAccess.sqlConnectionMetaDataEnvironment;

            dataAdaptor.SelectCommand.Parameters.Add("@SchemaName",
SqlDbType.NVarChar, 100);

dataAdaptor.SelectCommand.Parameters.Add("@StoredProcedureNameFilter",
SqlDbType.NVarChar, 100);
            dataAdaptor.SelectCommand.Parameters.Add("@language1",
SqlDbType.NChar, 4);
            dataAdaptor.SelectCommand.Parameters.Add("@language2",
SqlDbType.NChar, 4);
            // set values for parameters
            dataAdaptor.SelectCommand.Parameters["@SchemaName"].Value
= schemaName;

dataAdaptor.SelectCommand.Parameters["@StoredProcedureNameFilter"].Val
ue = storedProcedureNameFilter;
            dataAdaptor.SelectCommand.Parameters["@language1"].Value =
language1;
            dataAdaptor.SelectCommand.Parameters["@language2"].Value =
language2;


PosityUtilities.DataBaseAccess.ExecuteOpenStatement(dataAdaptor.Select
Command.Connection);

PosityUtilities.DataBaseAccess.ExecuteFillStatement(dataAdaptor,
dataSet);

PosityUtilities.DataBaseAccess.ExecuteCloseStatement(dataAdaptor.Selec
tCommand.Connection);
            foreach (DataRow dataRow in dataSet.Tables[0].Rows)
            {
                logicalQueryName = dataRow.ItemArray[0].ToString();
                procedureName = dataRow.ItemArray[1].ToString();
```

```
                cachedTime = dataRow.ItemArray[2].ToString();
                execCount =
int.Parse(dataRow.ItemArray[3].ToString());
                minElapsedTime =
int.Parse(dataRow.ItemArray[4].ToString());
                maxElapsedTime =
int.Parse(dataRow.ItemArray[5].ToString());
                totalElapsedTime =
int.Parse(dataRow.ItemArray[6].ToString());
                avgElapsedTime =
int.Parse(dataRow.ItemArray[7].ToString());
                queryPerfInfo = new QueryPerfInfo(logicalQueryName,
procedureName, cachedTime, execCount, minElapsedTime, maxElapsedTime,
totalElapsedTime, avgElapsedTime);
                listOfQueryPerfInfos.Add(queryPerfInfo);
            }
            return listOfQueryPerfInfos;
        } // method FindQueryPerformanceInfos
```

Weitere Details zur Implementation von den Query Performance DevTools befinden sich in der Datei PerformanceTools.cs in der Klasse `QueryPerformancePage`.

6.5.2 Applikationslogik

Die folgenden Unterkapitel beschreiben wie Performance-Messdaten in den einzelnen Posity Virtual Machines aufgezeichnet, diese in eine geeignete Datenstruktur abgelegt und schlussendlich auf einer Benutzeroberfläche dargestellt werden.

6.5.2.1 Vorbedingung Ende-fCode bei Strukturbausteinen

Im Kapitel 6.4.2.2 Problem Strukturbaustein Ende-fCode wurde beschrieben, dass für jeden Strukturbaustein der fCode um ein Ende-fCode erweitert werden muss. Der Posity Modul-Diagramm zu fCode Compiler wurde erweitert, damit beifolgenden Strukturbausteinen ein Ende-fCode im fCode eingefügt wird.

Tabelle 10: Neue Ende-fCodes für Strukturbausteine

Strukturbaustein	**Ende-fCode**	**Methode in C#**
Sequenz	`SeqEnd`	`GenerateSequenceStructureEnd`
ForEach, ForLoop	`ForEnd`	`GenerateLoopStructureEnd`
Switch-Case	`SwiEnd`	`GenerateCaseStructureEnd`
Lane	`LneEnd`	`GenerateLaneStructureEnd`
Try-Catch	`TryEnd`	`GenerateTryCatchStructureEnd`

Der Programm-Code für die fCode Erweiterung findet sich in der Datei `IDE_DBMS-Transactions_Modul.cs` bei der End-Methode für den jeweiligen Strukturbaustein. Für den Try-Catch Strukturbaustein ist dies zum Beispiel die Methode `GenerateTryCatchStructureEnd`, wo der TryEnd fCode dem gesamten fCode hinzugefügt wird.

```
// add end of trycatch command to mark the trycatch end
fCodeOfModule.AddCommand("TryEnd", structureToProcess.guidOfShape);
```

6.5.2.2 Performance-Messdaten der Posity Virtual Machines erfassen

Wie im Kapitel 6.4.2 Applikationslogik beschrieben, kann eine Performance-Analyse der Applikationslogik durchgeführt werden, indem zu jedem in der PVM ausgeführten Logikbaustein, entsprechende Performance-Messdaten (siehe Tabelle 9) in der PVM erfasst und in einer Datenstruktur abgelegt werden. Die Abbildung 26 zeigt den Aufbau der Datenstruktur für die Haltung von ausgeführten Logikbausteinen.

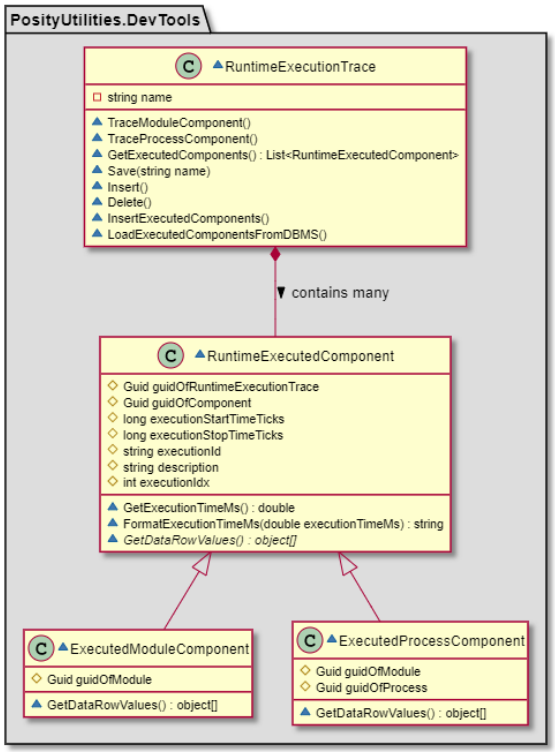


Abbildung 26: UML Klassendiagramm der Datenstrukturen für die Erfassung von Performance-Messdaten der PVM (Quelle: Eigene Darstellung)

Für alle ausgeführten Logikbausteine respektive fCodes wird ein `RuntimeExecutedComponent` Objekt erzeugt. Im Moment gibt es zwei Objekt-Typen (`ExecutedModuleComponent` und `ExecutedProcessComponent`), um zu unterscheiden, ob es sich um einen Logikbaustein vom Modul-Diagramm oder vom Prozess-Diagramm handelt. Dieser Aufbau der Datenstruktur ermöglicht es in Zukunft auch Logikbausteine, respektive ausgeführte Komponenten, anderer Posity-Diagramme z.B. dem GUI- oder Query-Diagramm zu erfassen. Beim Starten einer Aufzeichnung von Performance-Messdaten wird ein Objekt vom Typ `RuntimeExecutionTrace` erstellt in welchem alle `RuntimeExecutedComponent` Objekte zusammengefasst werden. Es werden von allen PVMs alle `RuntimeExecutedComponent` Objekte in demselben `RuntimeExecutionTrace` Objekt festgehalten. Nach einem Aufzeichnungsvorgang enthält also ein `RuntimeExecutionTrace` Objekt alle ausgeführten Logikbausteine von allen PVMs in einer Liste, in der Sequenz in der die Logikbausteine ausgeführt wurden.

6.5.2.3 Performance-Daten aufbereiten

Um die sequenziell aufgezeichneten Performance-Messdaten als Call-Tree (siehe Kapitel 6.4.2.6) darzustellen, muss die Liste mit den `RuntimeExecutedComponent`-Objekten in eine Baumdatenstruktur transformiert werden. Dazu braucht es ein Regelwerk welches bestimmt, bei welcher `RuntimeExecutedComponent` ein neuer Eltern-Knoten, ein neuer Kind-Knoten oder gar kein Knoten entstehen soll. Grundsätzlich entstehen für die Funktionsbausteine immer ein neuer Kind-Knoten und bei Strukturbausteinen ein neuer Eltern-Knoten. Damit können alle Fälle sinnvoll abgebildet werden, ausser die ForEach- und For-Strukturbausteine. Es würde auch für den ForEach und For funktionieren, nur der Call-Tree würde zu lang in der Darstellung werden, wenn bei jeder Iteration nochmals dieselben Knoten, die innerhalb von einem ForEach respektive For liegen, im Baum angefügt würden. Deshalb gibt es eine zusätzlich Aggregationsregel für das Bauen vom Call-Tree, so dass z.B. die Knoten innerhalb vom ForEach und For aggregiert dargestellt werden. In den Abbildung 27, Abbildung 28 und Abbildung 29 ist ersichtlich wie beim Ausführen der Applikationslogikbausteine sequenzielle Performance-Messdaten entstehen, die dann für die Darstellung als Call-Tree in eine Baumstruktur transformiert werden müssen.

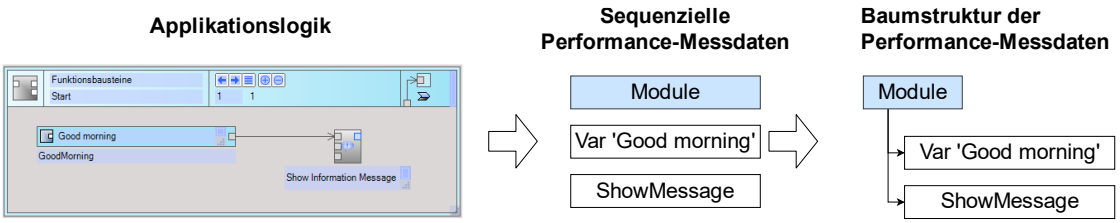


Abbildung 27: Abbildung Funktionsbausteine nach Performance-Messdaten-Baumstruktur (Quelle: Eigene Darstellung)

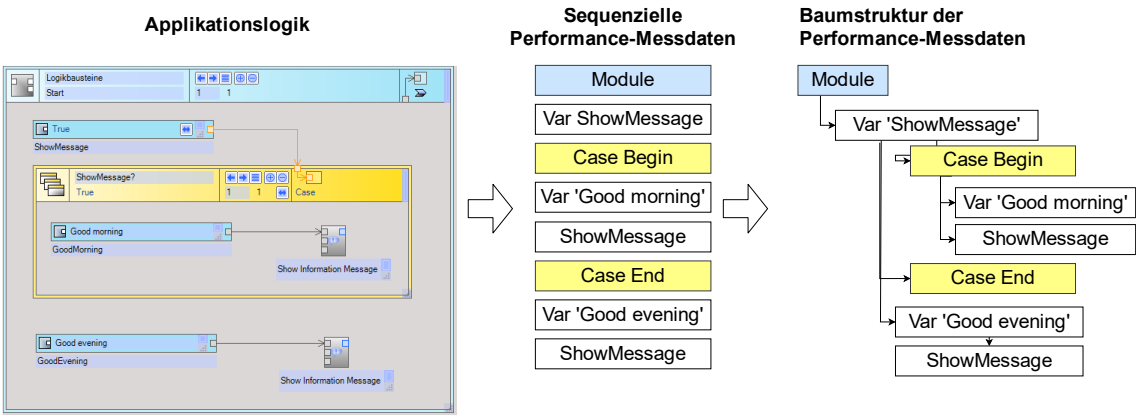


Abbildung 28: Abbildung Strukturbaustein Switch-Case nach Performance-Messdaten-Baumstruktur (Quelle: Eigene Darstellung)

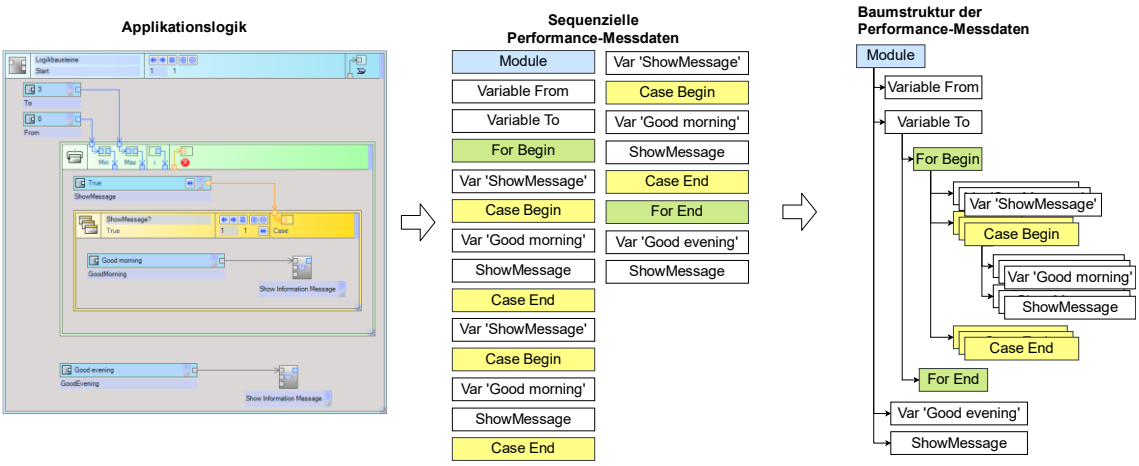


Abbildung 29: Abbildung Strukturbaustein For nach aggregierte Performance-Messdaten-Baumstruktur (Quelle: Eigene Darstellung)

Die folgende Abbildung 30 zeigt anhand eines UML-Klassendiagrammes welche Klassen und Funktionalitäten benötigt wurden, um die sequenziellen Performance-Messdaten in eine Baumstruktur zu transformieren.

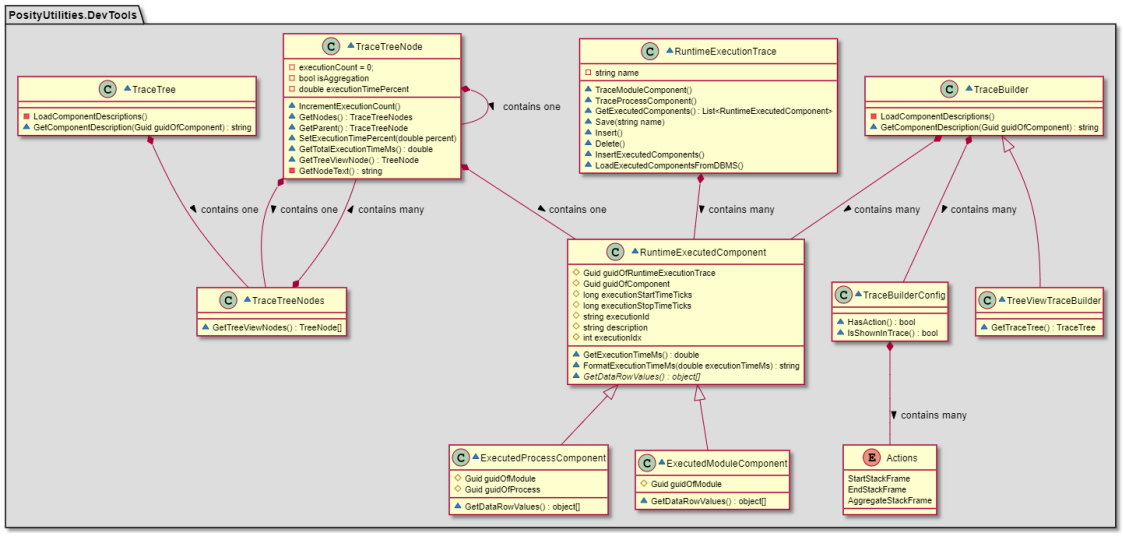


Abbildung 30: UML Klassendiagramm TraceTree (Quelle: Eigene Darstellung)

Die Klasse `TreeViewTraceBuilder` (im Diagramm ganz rechts) erstellt aus der Liste mit den `RuntimeExecutedComponent`-Objekten anhand der Regeln (Klasse `TraceBuilderConfig`) ein `TraceTree`-Objekt (im Diagramm ganz links). Das `TraceTree`-Objekt enthält ein Objekt `TraceTreeNodes` mit dessen Methode `GetTreeViewNodes` C#-`TreeNode`[11]-Objekte erstellt und in einem C#-`TreeView`[12] graphisch dargestellt werden können. Zum Beispiel wird in der `TraceBuilder`-Klasse mit folgenden zwei Regeln definiert, dass der For-Strukturbaustein aggregiert wird und ein neuer Eltern-Knoten entstehen soll.

```
configurations.Add("ForLop", new TraceBuilderConfig(true, new
List<Actions>() { Actions.StartStackFrame, Actions.AggregateStackFrame
}));
configurations.Add("ForEnd", new TraceBuilderConfig(true, new
List<Actions>() { Actions.EndStackFrame }));
```

Wie ein `TraceTree` im Detail entsteht, kann im vollständigen Auszug der Method `GetTraceTree` im Anhang D betrachtet werden. Weitere Ansichten für die Performance-Messdaten können mit dem Ableiten der Klasse `TraceBuilder` erstellt werden.

[11] C# TreeNode Objekt, https://docs.microsoft.com/en-us/dotnet/api/system.windows.forms.treenode?view=windowsdesktop-6.0, (Abgerufen: 13.05.2022)

[12] C# TreeView Control, https://docs.microsoft.com/en-us/dotnet/api/system.windows.forms.treeview?view=windowsdesktop-6.0, (Abgerufen: 13.05.2022)

Zum Beispiel mit einer neuen Klasse `ListTraceBuilder` analog zur Klasse `TreeViewTraceBuilder`, die dann die `RuntimeExecutedComponent` Objekte als Liste aufbereitet.

6.5.2.4 Performance-Daten darstellen

Sobald die Performance-Messdaten im `TraceTree` als Baumstruktur vorliegen, können diese mit dem C#-`TreeView`-Control in den DevTools dargestellt werden. Im Text der einzelnen Baum-Knoten sind folgende Informationen enthalten:

1. Wie hoch ist der Anteil der Ausführungszeit vom Knoten zu der Ausführungszeit über alle Knoten einer Ebene in Prozent.
2. Den Namen des ausgeführten Logikbausteines.
3. Wenn aggregiert, die Anzahl der Ausführungen und die Ausführungszeit pro Iteration.
4. Die Ausführungszeit des Knoten und den untergeordneten Knoten (Kinder-Knoten). Ausführungszeiten die kleiner sind als 1 Millisekunde werden mit dem Symbol `<1ms` angezeigt.

Das Berechnen vom Prozent-Anteil der Ausführungszeit kann erst erfolgen, wenn die Ausführungszeit für jeden Knoten berechnet ist. Mit dem Aufsummieren der Ausführungszeiten muss in den Knoten ohne Kinder-Knoten (Blatt-Knoten) gestartet und bis zum Knoten ohne Eltern-Knoten (Wurzel-Knoten) die Ausführungszeiten der einzelnen Knoten aufsummiert werden. Sind alle Ausführungszeiten der Knoten berechnet, kann ausgehend beim Wurzel-Knoten für alle darunterliegenden Kinder-Knoten der Anteil der Ausführungszeit berechnet werden. Zusammenfassend berechnen sich die Ausführungszeiten der Knoten in Bottom-Up- und die Anteile der Ausführungszeiten in Top-Down-Richtung. Diese Berechnungsvorgänge sind als Rekursion implementiert. Die Abbildung 31 zeigt den aus aufgezeichneten Performance-Messdaten erstellten Call-Tree von dem in der Abbildung 29 eingeführten Modul-Diagramm.

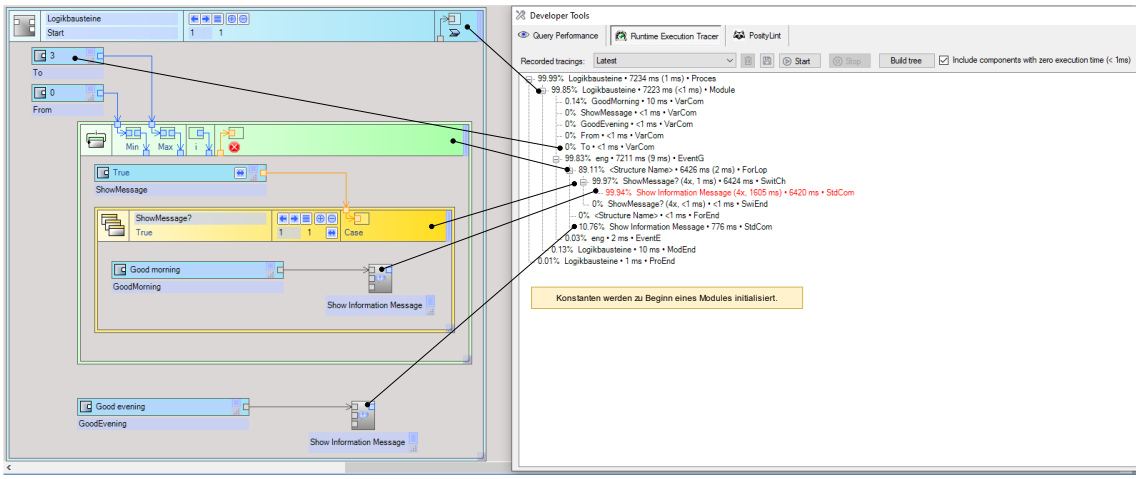


Abbildung 31: Call-Tree von Modul-Diagramm (Quelle: Eigene Darstellung)

In den aufgezeichneten Performance-Messdaten sind keine Beschreibungstexte der ausgeführten Logikbausteine vorhanden. Der Beschreibungstext einer Komponente wird beim Erstellen vom Call-Tree mit der statischen Methode `GetComponentDescription` in der `DevTools`-Klasse ermittelt (siehe Kapitel 5.2)

6.5.2.5 Persistieren der aufgezeichneten Performance-Messdaten

Eine Performance-Messung und die zugehörigen, aufgezeichneten Performance-Messdaten der Logikbausteine können in der Datenbank gespeichert und zu einem späteren Zeitpunkt wieder geladen werden. Die Performance-Messung wird in der Tabelle `SystemData.RuntimeExecutionTrace` und die ausgeführten Logikbausteine in der Tabelle `SystemData.RuntimeExecutedComponent` gespeichert. Das Speichern geschieht über den Aufruf der SQL-Stored Procedure `SystemData.RuntimeExecutionTrace_Insert` und `SystemData.RuntimeExecutedComponent_InsertTable`. Das Laden wiederum mit `SystemData.RuntimeExecutionTrace_Fill`, welche auch gleich die RuntimeExecutedComponent Datensätze zurückgibt. Aus Performance Gründen wird die SQL-Stored Procedure `SystemData.RuntimeExecutedComponent_InsertTable` eine Tabelle mit allen `RuntimeExecutedComponent` Objekten übergeben. Damit können alle `RuntimeExecutedComponent`-Objekte mit nur einem SQL-Stored Procedure-Aufruf gespeichert werden und muss nicht pro `RuntimeExecutedComponent`-Objekt jeweils ein SQL Stored Procudure Aufruf ausgeführt werden. Damit eine Tabelle einer SQL-Stored Procedure übergeben werden kann, braucht es im Microsoft SQL-Server einen User defined Table Type welcher die Struktur der Tabelle definiert (siehe Abbildung 32).

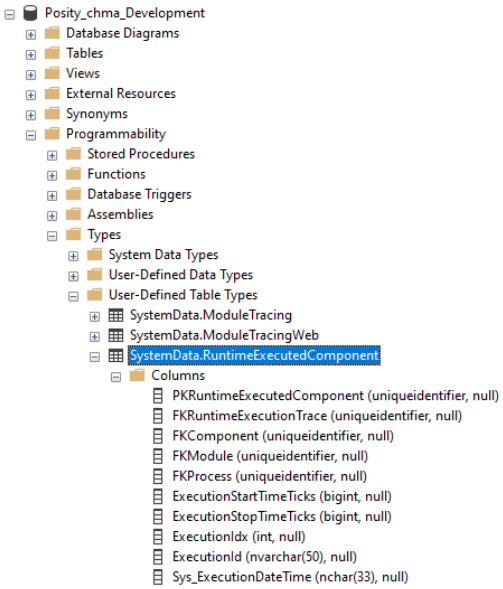


Abbildung 32: Microsoft SQL-Server User defined Table Type RuntimeExecutedComponent (Quelle: Eigene Darstellung)

6.5.2.6 Benutzeroberfläche

Über die Benutzeroberfläche des Artefaktes Applikationslogik kann das Aufzeichnen, Darstellen und das Verwalten von Performance-Messdaten gesteuert werden. Alle verfügbaren Performance-Messungen werden in einer DropDown-Liste angezeigt. Die letzte Aufzeichnung ist mit `latest` gekennzeichnet. Komponenten mit einer Ausführungszeit kleiner 1 Millisekunde können mit einer Option aus dem Call-Tree ausgeblendet werden. Zusätzlich kann ein Schwellwert für die Ausführungszeit einer einzelnen Komponente inklusive oder exklusive der Ausführungszeiten der Kind-Knoten festgelegt werden. Überschreitet die Ausführungszeit von einem Knoten den Schwellwert, wird dieser rot eingefärbt. Dies hilft für die Suche von Performance kritischen Logikbausteinen. Mit einem Rechtsklick auf einen Knoten im Call-Tree, wird bei geöffnetem Modul-Diagramm der mit dem Knoten verknüpfte Logikbaustein hervorgehoben. Damit kann eine Performance-Problemstelle sehr einfach und auch in sehr verschachtelten Modul-Diagrammen gefunden werden. Die Abbildung 33 zeigt die Benutzeroberfläche vom Artefakt Performance Applikationslogik und beschreibt die Funktionalitäten der Steuerelemente.

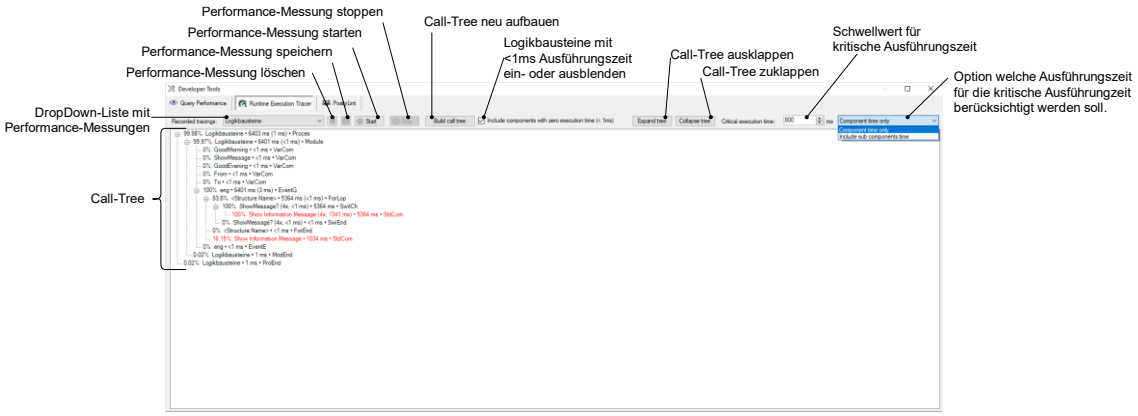


Abbildung 33: Benutzeroberfläche Artefakt Performance Applikationslogik
(Quelle: Eigene Darstellung)

6.6 Validierung

Dieses Kapitel beschreibt die Validierung des in der Implementationsphase (siehe Kapitel 6) entstandenen Performance-Analyse Artefakts. Das Artefakt wird validiert, indem eine Performance-Analyse für bekannte lange dauernde Business-Prozesse in konkreten Kundenapplikationen der Firma Posity AG durchgeführt wird. Damit können das Artefakt anhand der Erfolgskriterien bewertet und allenfalls Performance Probleme bei der Kundenapplikation aufgedeckt werden.

6.6.1 Geeignete Applikationen bestimmen

Ideale Kandidaten für eine Validation der Datenbank-Abfragen- und Applikationslogik-Performance DevTools sind Posity-Applikationen die täglich von Kunden benutzt werden und bei der Performance Engpässe bekannt sind. Zudem muss es für das Aufzeichnen der Performance-Messdaten der Applikationslogik möglich sein den fCode der Posity-Applikation anzupassen. Mit diesen Vorbedingungen ist die Auswahl auf zwei Posity-Kundenapplikationen beschränkt, wobei nur bei einer Kundenapplikation der fCode aktuell verändert werden konnte, um auch Performance-Messdaten zur Applikationslogik aufzeichnen zu können. Die anderen Posity-Kundenapplikationen werden teilweise nur in einem bestimmten Rhythmus benutzt, was für die Performance-Messdaten der Datenbank-Abfragen wenig Daten bedeutet, oder der fCode darf nicht verändert werden, weil dies den Produktionsbetrieb stören würde. Die Kapitel 6.6.2 und 6.6.3 beschreiben die für die Validation verwendeten Posity-Kundenapplikationen.

6.6.2 Kundenapplikation Rezeptparameter Verwaltung für Kabelfabrikationsanlagen (HS Propara)

In dieser Rezeptparameter Verwaltungsapplikation werden für mehrere Fabrikationsanlagen für verschiedene Kabeltypen Rezeptparameter für die verschiedenen Anlageteile der Fabrikationsanlagen verwaltet. Anpassungen in einzelner Anlageteileparameter führt zu aufwendigen Berechnungen und Prüfungen von betroffenen Anlageteilen und deren Parametern. Die Berechnungen werden hauptsächlich in SQL-Stored Procedure durchgeführt. Deshalb ist es in dieser Applikation interessant die Performance-Messdaten der Datenbank-Abfragen zu ermitteln. Die Abbildung 34 zeigt die aufgezeichneten Performance-Messdaten der Datenbank-Abfragen der Rezeptverwaltung absteigend geordnet nach durchschnittlicher Ausführungszeit.

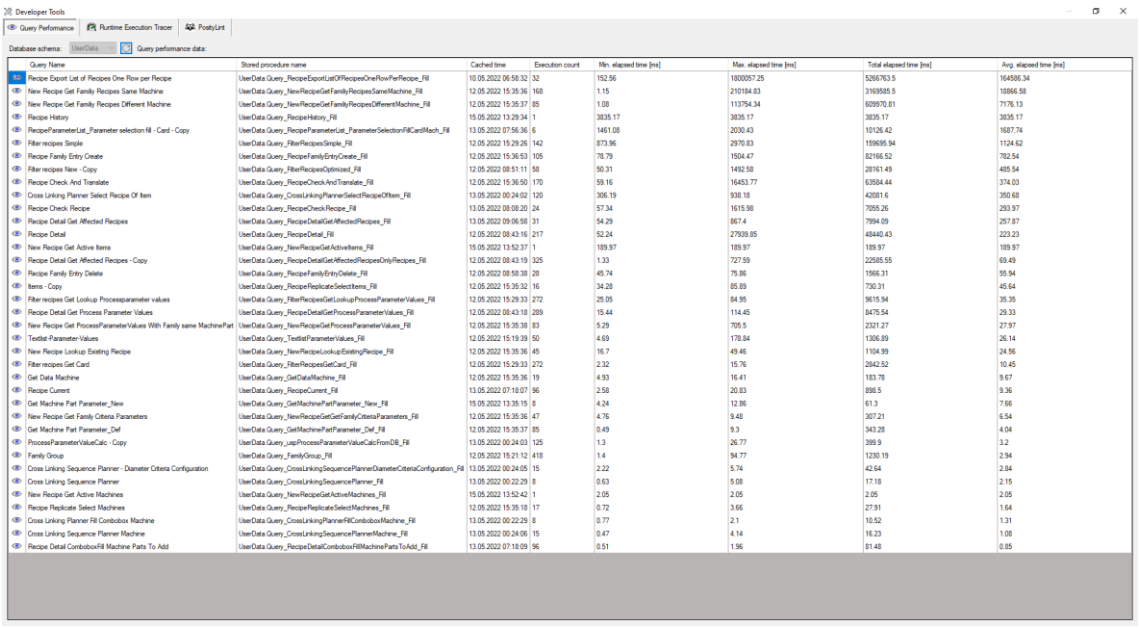


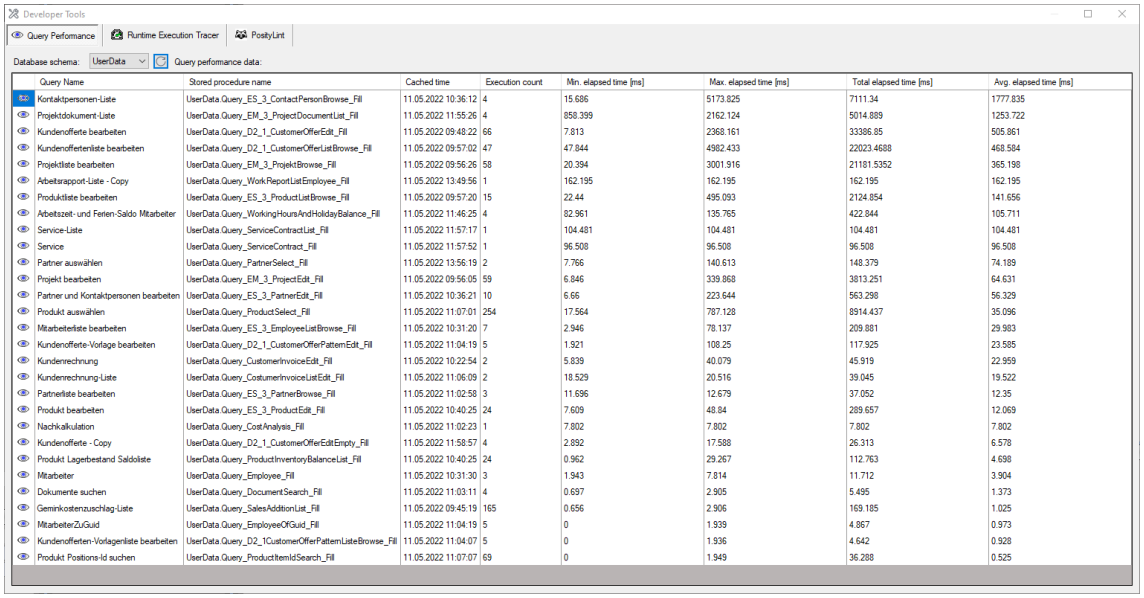
Abbildung 34: Rezeptverwaltung Datenbank-Abfragen Performance-Messdaten (Quelle: Eigene Darstellung)

Die durchschnittlich am längsten dauernde Datenbank-Abfrage läuft für ca. für 2.5 Minuten.

6.6.3 Kundenapplikation Kundenofferte Verwaltung für Solaranlagen (Sol ERP)

Die zweite Applikation wird bei einer Firma in der Solarbranche verwendet, um Kundenofferten für Solaranlagen zu verwalten, zu berechnen und Offerten-Dokumente zu erstellen. Die Kosten für das Offerten-Wesen liegen meistens beim Auftragnehmer, sodass ein

grosses Interesse besteht die Bearbeitungszeiten für Offerten möglichst gering zu halten. Das Rechnen von Offerten ist je nach Anzahl der Offerten-Positionen (Artikel in einer Offerte) zeitaufwendig und das Verwalten von Zeichnungen, Plänen und Bilder für die Offerte ist ressourcenintensiv (Arbeitsspeicher, CPU-Time). Deshalb sind die Prozesse rund um das Offerten-Wesen interessante Kandidaten, um das Performance-Analyse Artefakt anzuwenden. Bei dieser Applikation konnte auch der fCode dieser Prozesse angepasst werden, sodass zusätzlich zu den Datenbank-Abfragen auch Performance-Messdaten für die Applikationslogik erfasst werden konnten.



Query Name	Stored procedure name	Cached time	Execution count	Min. elapsed time [ms]	Max. elapsed time [ms]	Total elapsed time [ms]	Avg. elapsed time [ms]
Kontaktpersonen-Liste	UserData.Query_ES_3_ContactPersonBrowse_Fill	11.05.2022 10:36:12	4	15.686	5173.825	7111.34	1777.835
Projektdokument-Liste	UserData.Query_EM_3_ProjectDocumentList_Fill	11.05.2022 11:55:26	4	858.399	2162.124	5014.889	1253.722
Kundenofferte bearbeiten	UserData.Query_D2_1_CustomerOfferEdit_Fill	11.05.2022 09:48:22	66	7.813	2368.161	33386.85	505.861
Kundenofferteliste bearbeiten	UserData.Query_D2_1_CustomerOfferListBrowse_Fill	11.05.2022 09:57:02	47	47.844	4982.433	22023.4688	468.584
Projektliste bearbeiten	UserData.Query_EM_3_ProjektBrowse_Fill	11.05.2022 09:56:26	58	20.394	3001.916	21181.5352	365.198
Arbeitsrapport-Liste - Copy	UserData.Query_WorkReportListEmployee_Fill	11.05.2022 13:49:56	1	162.195	162.195	162.195	162.195
Produktliste bearbeiten	UserData.Query_ES_3_ProductListBrowse_Fill	11.05.2022 09:57:20	15	22.44	495.093	2124.854	141.656
Arbeitszeit- und Ferien-Saldo Mitarbeiter	UserData.Query_WorkingHoursAndHolidayBalance_Fill	11.05.2022 11:46:25	4	82.961	135.765	422.844	105.711
Service-Liste	UserData.Query_ServiceContractList_Fill	11.05.2022 11:57:17	1	104.481	104.481	104.481	104.481
Service	UserData.Query_ServiceContract_Fill	11.05.2022 11:57:52	1	96.508	96.508	96.508	96.508
Partner auswählen	UserData.Query_PartnerSelect_Fill	11.05.2022 13:56:19	2	7.766	140.613	148.379	74.189
Projekt bearbeiten	UserData.Query_EM_3_ProjectEdit_Fill	11.05.2022 09:56:05	59	6.846	339.868	3813.251	64.631
Partner und Kontaktpersonen bearbeiten	UserData.Query_ES_3_PartnerEdit_Fill	11.05.2022 10:36:21	10	6.66	223.644	563.298	56.329
Produkt auswählen	UserData.Query_ProductSelect_Fill	11.05.2022 11:07:01	254	17.564	787.128	8914.437	35.096
Mitarbeiterliste bearbeiten	UserData.Query_ES_3_EmployeeListBrowse_Fill	11.05.2022 10:31:20	7	2.946	78.137	209.881	29.983
Kundenofferte-Vorlage bearbeiten	UserData.Query_D2_1_CustomerOfferPatternEdit_Fill	11.05.2022 11:04:19	5	1.921	108.25	117.925	23.585
Kundenrechnung	UserData.Query_CustomerInvoiceEdit_Fill	11.05.2022 10:22:54	2	5.839	40.079	45.919	22.959
Kundenrechnung-Liste	UserData.Query_CostumerInvoiceListEdit_Fill	11.05.2022 11:06:09	2	18.529	20.516	39.045	19.522
Partnerliste bearbeiten	UserData.Query_ES_3_PartnerBrowse_Fill	11.05.2022 11:02:58	3	11.696	12.679	37.052	12.35
Produkt bearbeiten	UserData.Query_ES_3_ProductEdit_Fill	11.05.2022 10:40:25	24	7.609	48.84	289.657	12.069
Nachkalkulation	UserData.Query_CostAnalysis_Fill	11.05.2022 11:02:23	1	7.802	7.802	7.802	7.802
Kundenofferte - Copy	UserData.Query_D2_1_CustomerOfferEditEmpty_Fill	11.05.2022 11:58:57	4	2.892	17.588	26.313	6.578
Produkt Lagerbestand Saldoliste	UserData.Query_ProductInventoryBalanceList_Fill	11.05.2022 10:40:25	24	0.962	29.267	112.763	4.698
Mitarbeiter	UserData.Query_Employee_Fill	11.05.2022 10:31:30	3	1.943	7.814	11.712	3.904
Dokumente suchen	UserData.Query_DocumentSearch_Fill	11.05.2022 11:03:11	4	0.697	2.905	5.495	1.373
Geminkostenzuschlag-Liste	UserData.Query_SalesAddtionList_Fill	11.05.2022 09:45:19	165	0.656	2.906	169.185	1.025
MitarbeiterZuGuid	UserData.Query_EmployeeOfGuid_Fill	11.05.2022 11:04:19	5	0	1.939	4.867	0.973
Kundenofferten-Vorlagenliste bearbeiten	UserData.Query_D2_1CustomerOfferPatternListeBrowse_Fill	11.05.2022 11:04:07	5	0	1.936	4.642	0.928
Produkt Positions-Id suchen	UserData.Query_ProductItemIdSearch_Fill	11.05.2022 11:07:07	69	0	1.949	36.288	0.525

Abbildung 35: Datenbank-Abfragen Performance-Messdaten der Kundenofferten-Verwaltung

Zum Beispiel dauert die Datenbank-Abfrage Kontaktpersonen-Liste durchschnittlich ca. 1.7 Sekunden.

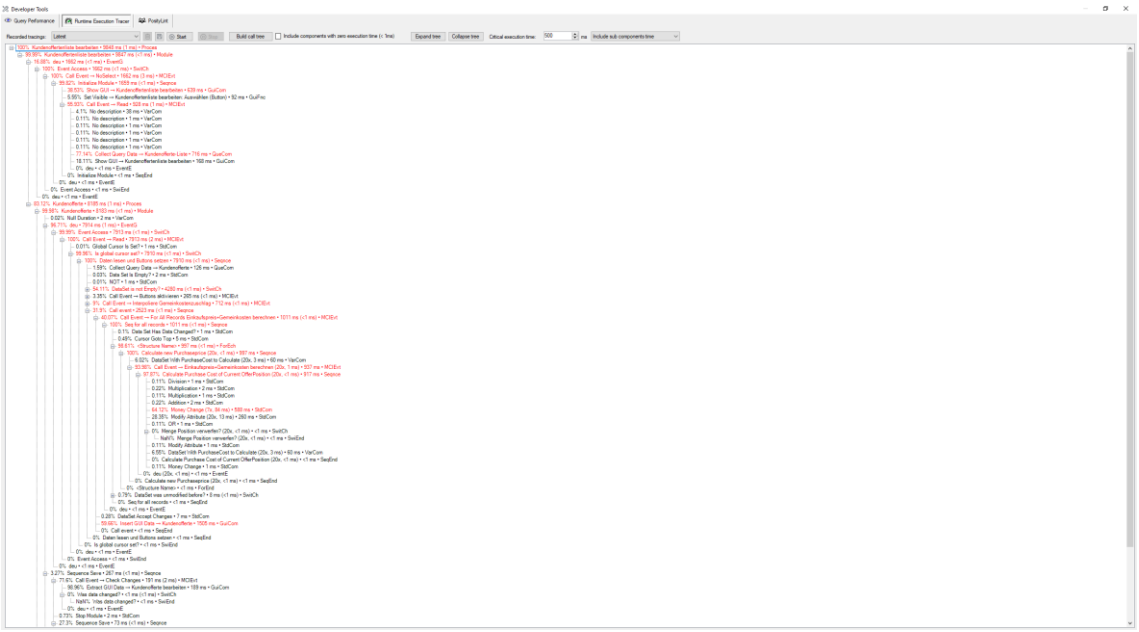


Abbildung 36: Call-Tree der Performance-Messdaten Kundenofferte anzeigen (Quelle: Eigene Darstellung)

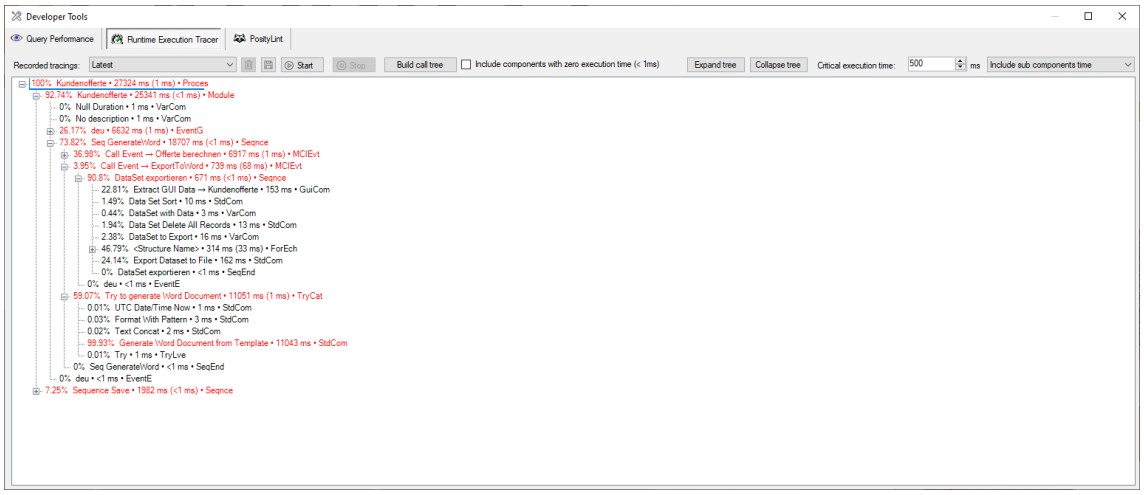


Abbildung 37: Call-Tree der Performance-Messdaten der Applikationslogik für Kundenofferte Dokument generieren (Quelle: Eigene Darstellung)

Die rot markierten Knoten im Call-Tree (siehe in der Abbildung 36 und Abbildung) zeigen Logikbausteine der Applikationslogik deren Ausführungszeit länger als 500 Millisekunden dauert.

6.6.4 Bewertung der Erfolgskriterien

In der Tabelle 12 wurden anhand der Validierung vom Artefakt Performance-Analyse mit den Posity-Kundenapplikationen Bewertungen der Erfolgskriterien eingetragen. Die Erfolgskriterien wurden mit Ja, Nein, einem Ausrufezeichen (!), der Tilde (~) oder Bindestrich (-) bewertet. Bei einer Ja-Antwort trifft das Kriterium vollständig zu, bei einer Nein-Antwort gar nicht, bei (~) nur teilweise. Ein Ausrufezeichen gibt an, dass dieses Kriterium nicht validiert werden konnte und ein Bindestrich besagt keine Relevanz für ein Erfolgskriterium für diesen Teilartefakt (z.B. KPA1 hat keine Relevanz zum Teilartefakt Datenbank-Abfragen).

Tabelle 11: Legende Erfüllungsgrad

Symbol Erfüllungsgrad	Bedeutung
Ja	Kriterium ist vollständig erfüllt.
Nein	Kriterium ist nicht erfüllt.
~	Kriterium ist teilweise erfüllt.
!	Kriterium ist nicht validierbar.
-	Kriterium hat keine Relevanz für diesen Teil.

Tabelle 12: Bewertung der Erfolgskriterien für die 2 Posity-Applikation Rezeptverwaltung und Kundenofferten-Verwaltung

Kriterium	**KCA1**	
Beschreibung	Die Performance-Analyse kann generell für alle in einem Module-Diagramm erstellten Applikationslogiken angewandt werden.	
Bewertung «HS Propara»	Bewertung	Erfüllungsgrad
	Datenbank-Abfragen	-
Bewertung «Sol ERP»	Bewertung	Erfüllungsgrad
	Datenbank-Abfragen	-
	Applikationslogik	Ja
Kommentar	-	

Kriterium	**KCA2**	
Beschreibung	Das Artefakt unterstützt die Posity-Entwickler bei ihrer täglichen Arbeit.	
Bewertung «HS Propara»	Bewertung	Erfüllungsgrad
	Datenbank-Abfragen	Ja
Bewertung «Sol ERP»	Bewertung	Erfüllungsgrad
	Datenbank-Abfragen	Ja
	Applikationslogik	Ja
Kommentar	-	

Kriterium	**KCA3**	
Beschreibung	Ein Posity-Entwickler kann das Artefakt ohne Anleitung bedienen.	
Bewertung «HS Propara»	Bewertung	Erfüllungsgrad
	Datenbank-Abfragen	Ja
Bewertung «Sol ERP»	Bewertung	Erfüllungsgrad
	Datenbank-Abfragen	Ja
	Applikationslogik	~ (siehe Kommentar)
Kommentar	Das Artefakt für die Performance-Analyse der Applikationslogik enthält im aktuellen Entwicklungsstadium Fehler in der Abbildung des Call-Trees, weil die Posity Virtual Machine um fCode Codes ergänzt wurde (siehe Kapitel 6.1.2.2). Diese Fehler wurden erst entdeckt, als die Kundenofferten-Verwaltungsapplikation validiert wurde. Grobe Fehler konnten behoben werden, sodass bis auf das Darstellen von Events, die vom Benutzer initiiert wurden, der Call-Tree stimmt. Der Call-Tree muss anschliessend an diese Arbeit mit weiteren Anwendungen validiert und geprüft werden, ob Spezialfälle, wie z.B. Submodule mit Parametern, Events mit Parametern, korrekt im Call-Tree dargestellt werden. Deshalb kann dieser Teil des Artefaktes noch nicht eigenständig von einem Posity-Entwickler verwendet werden.	

Kriterium	**KCA4**	
Beschreibung	Die Qualität einer mit der Posity IDE erstellen Applikation kann gesteigert werden.	
Bewertung «HS Propara»	Bewertung	Erfüllungsgrad
	Datenbank-Abfragen	! (siehe Kommentar)
Bewertung «Sol ERP»	Bewertung	Erfüllungsgrad
	Datenbank-Abfragen	! (siehe Kommentar)
	Applikationslogik	! (siehe Kommentar)
Kommentar	Ob Qualitätssteigerungen möglich sind durch die Einsicht in die Performance-Messdaten konnte in dieser Validierung nicht beurteilt werden. Die Ergebnisse müssen erst noch von den verantwortlichen Posity-Entwicklern der Applikationen eingesetzt werden. Erst dann kann bewertet werden, ob die aufgezeichneten Performance-Messdaten eine Hilfe für die Verbesserung der Performance der Posity-Applikationen waren.	

Kriterium	**KCA5**	
Beschreibung	Das Artefakt kann so erweitert werden, dass es auch für weitere Posity Diagrammtypen gebraucht werden kann.	
Bewertung «HS Propara»	Bewertung	Erfüllungsgrad
	Datenbank-Abfragen	Nein
Bewertung «Sol ERP»	Bewertung	Erfüllungsgrad
	Datenbank-Abfragen	Nein
	Applikationslogik	! (siehe Kommentar)
Kommentar	Die Klassenstruktur für das Aufzeichnen der Performance-Daten wurde so gewählt, dass auch andere Programmteile, die fCode verarbeiten, in dieselben Strukturen aufgenommen werden können. Ob dieses Konzept wirklich funktioniert, konnte in dieser Validierung nicht festgestellt werden. Dies kann erst beurteilt werden, wenn z.B. das Posity-GUI ebenfalls den ausgeführten fCode aufzeichnet.	

Kriterium	**KCA6**	
Beschreibung	Das Artefakt unterstützt die Basisfunktionalitäten gängiger Performance-Analyse Tools.	
Bewertung «HS Propara»	Bewertung	Erfüllungsgrad
	Datenbank-Abfragen	Nein
Bewertung «Sol ERP»	Bewertung	Erfüllungsgrad
	Datenbank-Abfragen	Nein
	Applikationslogik	Ja
Kommentar	Die Datenbank-Abfragen Performance-Analyse könnte detailliertere Einsichten in die Datenbank-Abfrage bieten.	

Kriterium	**KCA7**	
Beschreibung	Das Anwenden vom Artefakt führt **nicht** zu Performance Einbrüchen der eigentlichen Applikation	
Bewertung «HS Propara»	Bewertung	Erfüllungsgrad
	Datenbank-Abfragen	Ja
Bewertung «Sol ERP»	Bewertung	Erfüllungsgrad
	Datenbank-Abfragen	Ja
	Applikationslogik	! (siehe Kommentar)
Kommentar	Das Aufzeichnen der Performance-Messdaten für die Datenbank-Abfragen beinträchtigen die Ausführung der Posity-Applikation nicht. Diese Daten werden immer im Hintergrund vom Microsoft SQL-Server erfasst und verzögern die Posity-Applikation nicht. Das Aufzeichnen der Performance-Messdaten für die Applikationslogik kann nur qualitativ beurteilt werden. Bei der Validierung	

	sind keine merklichen Verzögerungen durch den Aufzeichnungs-vorgang entstanden. Es gibt aber sicher Verzögerungen. Um dieses Erfolgskriterium beurteilten zu können, müssten diese quantitativ erfasst werden.	
Kriterium	**KCA8**	
Beschreibung	Wo in der Applikationslogik oder in welcher Datenbank-Abfrage Performance Probleme vorliegen kann festgestellt werden.	
Bewertung «HS Propara»	Bewertung	Erfüllungsgrad
	Datenbank-Abfragen	Ja
Bewertung «Sol ERP»	Bewertung	Erfüllungsgrad
	Datenbank-Abfragen	Ja
	Applikationslogik	Ja
Kommentar	-	
Kriterium	**KCA9**	
Beschreibung	Das Artefakt läuft robust.	
Bewertung «HS Propara»	Bewertung	Erfüllungsgrad
	Datenbank-Abfragen	Ja
Bewertung «Sol ERP»	Bewertung	Erfüllungsgrad
	Datenbank-Abfragen	Ja
	Applikationslogik	~ (siehe Kommentar)
Kommentar	Das Aufzeichnen und Darstellen der Applikationslogik-Perfor-mance-Messdaten liefen in der Validierung des Artefaktes robust (keine Programabstürze, fehlerhafte Performance-Messdaten, NULL-Pointer-Exceptions). Weil aber nicht alle Varianten von	

	Applikationslogiken, die mit der Posity IDE gebaut werden können, getestet wurden, kann diese Erfolgskriterium nur teilweise mit Ja bewertet werden. Um dieses Erfolgskriterium mit Ja zu werten, müssen in weiteren Posity-Applikationen die Applikationslogik aufgezeichnet und Spezialfälle ausgetestet werden.	
Kriterium	**KCA10**	
Beschreibung	Das Artefakt kann mit wenig Aufwand für weitere Posity-Entwickler verfügbar gemacht werden.	
Bewertung «HS Propara»	Bewertung	Erfüllungsgrad
	Datenbank-Abfragen	Ja
Bewertung «Sol ERP»	Bewertung	Erfüllungsgrad
	Datenbank-Abfragen	Ja
	Applikationslogik	Nein (siehe Kommentar)
Kommentar	Es müssen vorab noch Fehler korrigiert werden.	

Kriterium	**KCA11**	
Beschreibung	Mit dem Artefakt lässt sich die Performance von Datenbank-Abfragen, die mit der No-Code Posity IDE erstellt wurden, messen.	
Bewertung «HS Propara»	Bewertung	Erfüllungsgrad
	Datenbank-Abfragen	Ja
Bewertung «Sol ERP»	Bewertung	Erfüllungsgrad
	Datenbank-Abfragen	Ja
	Applikationslogik	-
Kommentar	-	

Kriterium	**KCA12**	
Beschreibung	Mit dem Artefakt lässt sich die Performance von Applikationslogiken, die mit der No-Code Posity IDE erstellt wurde, messen.	
Bewertung «HS Propara»	Bewertung	Erfüllungsgrad
	Datenbank-Abfragen	-
Bewertung «Sol ERP»	Bewertung	Erfüllungsgrad
	Datenbank-Abfragen	-
	Applikationslogik	Ja
Kommentar	-	

Zusammenfassend sind von den **insgesamt 12 Erfolgskriterien 5** vollständig erfüllt, **4** teilweise erfüllt, **2** nicht erfüllt, und **1** Kriterium gilt als nicht validierbar.

7 Artefakt Quellcode-Analyse – Posity Linter

Für textbasierte Programmiersprachen wie beispielsweise die objektorientierten Programmiersprache Java definiert Sonar[13] zum heutigen Zeitpunkt 633 Clean Code-Regeln. Das zeigt auf, wie vielseitig das Regelwerk ausgestaltet werden kann. In dieser Arbeit geht es nicht darum einen vollständigen Satz an Regeln zu spezifizieren und diese zu implementieren. Vielmehr soll mit dem Artefakt anhand einiger weniger, vom Typ möglichst unterschiedlicher Regeln überprüft werden, ob die Adaption von Clean Code-Regeln auf modellbasierte LCNC-Programmiersprachen, am Beispiel von Posity, möglich ist und damit Problemzonen im Code aufgedeckt und eliminiert werden können.

7.1 Anwendungskontext

Dieses Kapitel beschreibt in welchem Anwendungskontext das Artefakt für die Quellcode-Analyse eingesetzt wird. Wie bereits in vorhergehenden Kapiteln beschrieben, gehört die Programmiersprache Posity zu den modellbasierten Programmiersprachen. Der Code wird nicht textuell geschrieben, sondern grafisch durch die Kombination von Komponenten erstellt. Aufgrund dieser Eigenschaft gehört Posity der Familie von LCNC an. In Posity repräsentiert eine Menge von verschiedenen Diagrammen das Anwendungsmodell. Die verschiedenen Diagrammtypen, wie damit modelliert wird und welche Qualitätskontrollen bereits zur Verfügung stehen, erklären die nachfolgenden Kapitel.

7.1.1 Programmieren mit Posity Design Studio

Der Quellcode in modellbasierten Programmiersprachen besteht nicht aus einer Sammlung von Textzeilen, sondern aus grafischen Modellen. Wie bereits in Kapitel 4.3 erläutert, besteht das Posity Design Studio aus insgesamt acht verschiedenen Diagrammtypen, welche für die Modellierung der Applikation verwendet werden. Im vorliegenden Artefakt wurde der Fokus auf die für die Definition und Ausführung einer Applikation zentralen Diagrammtypen **GUI**, **Query**, **Table** und **Module** gelegt (siehe grün hinterlegte Diagrammtypen in Abbildung 38). Für die vollständige Spezifikation einer Applikation braucht es zusätzlich auch das Prozessdiagramm und das Organisationsdiagramm. Diese spielen für die Code-Analyse vorerst eine untergeordnete Rolle und wurden deshalb,

[13] Sonarcloud Rules, https://sonarcloud.io/organizations/default/rules?languages=java, (Abgerufen: 01.05.2022)

ebenso wie die optionalen Diagrammtypen Questionnaire und Package, in der Arbeit nicht berücksichtigt.

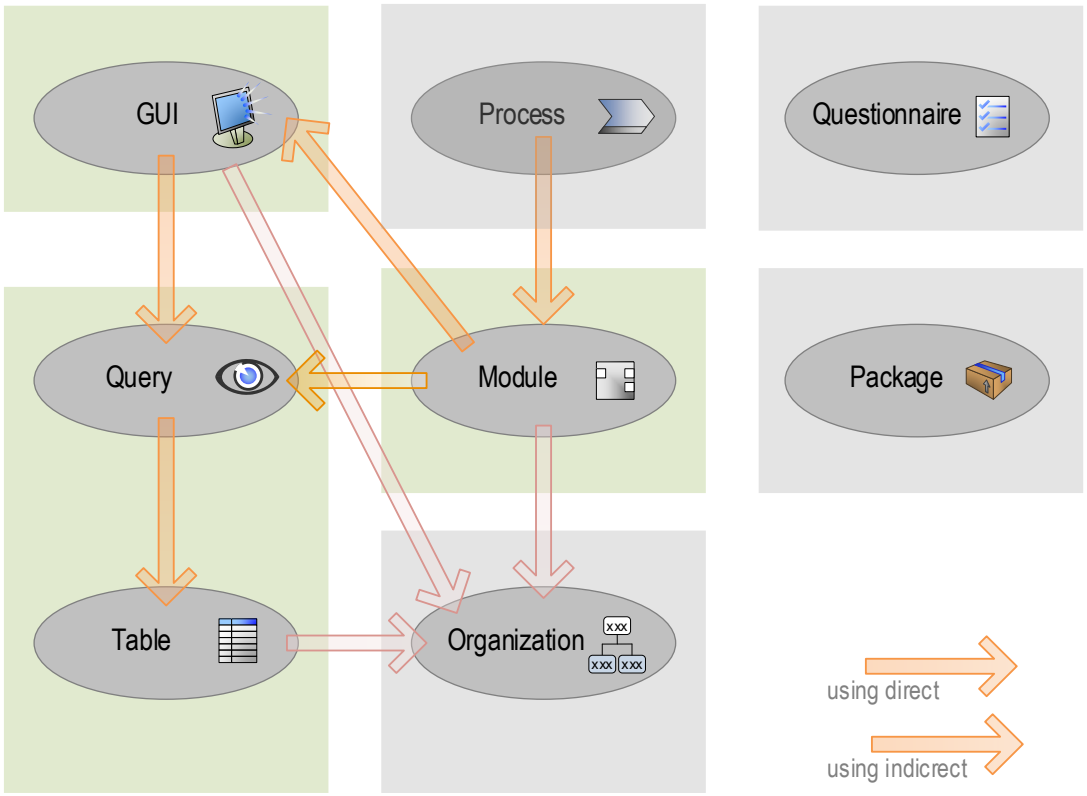


Abbildung 38: Posity Diagramme und Verwendung in Analyse - grün ist Teil der Analyse (Quelle: Eigene Darstellung)

Für die Durchführung der Analyse soll es grundsätzlich keine Rolle spielen, welcher Diagrammtyp analysiert wird. Auch wenn die verschiedenen Diagrammtypen unterschiedliche Qualitätsanforderungen aufweisen, werden gewisse Typen von Metriken wie beispielsweise Namenskonventionen auf mehrere Diagrammtypen angewandt.

Für die Modellierung konkreter Diagramme werden grafische Elemente zu den jeweiligen Diagrammen hinzugefügt. Diese Elemente werden in Posity als Shapes bezeichnet. Die acht Diagrammtypen haben jeweils unterschiedliche Shapes zur Auswahl, um die Modelle zu erstellen. So können beispielsweise im GUI-Diagramm Button-Shapes, Panel-Shapes oder Attribute-Shapes verwendet werden, um eine grafische Benutzeroberfläche zu modellieren. Im Tabellen-Diagramm stehen z.B. Table-Shapes, Primarykey-Shapes, Foreignkey-Shapes oder Attribute-Shapes zur Verfügung, um das Datenmodell der Anwendung zu modellieren.

Kapitel 7.1.4 erläutert diese Shapes im Detail. Die nachfolgenden Subkapitel beschreiben zunächst, wie die Modellierung generell konzipiert ist und wie sie im Speziellen pro Diagrammtyp abläuft. Der Beschreibung der Modellierung in den Diagrammen wird bewusst viel Platz eingeräumt, weil die Diagramme den «Code» darstellen und sie deswegen für die Umsetzung der Regeln eine zentrale Rolle spielen. Trotzdem, soll als Hinweis für den Lesenden gelten, dass die Kapitel 7.1.1.2 bis 7.1.1.5 nicht im Detail studiert werden müssen, sondern überfolgen werden können. Wichtig ist, dass der Lesende versteht, wie in der Posity IDE programmiert wird.

7.1.1.1 Grundsätzliches Vorgehen beim Modellieren der Diagramme

In Abbildung 39 werden die vier wichtigsten Bereiche der Entwicklungsumgebung (IDE) für die Programmierung in Posity dargestellt. Das «Application Panel» ganz links besteht aus den drei Bereichen «Open Diagrams», «All Elements of Application» und «Search Criterias for Elements». Für diese Arbeit sind nur die beschrifteten Bereiche von Bedeutung. Die Baumstruktur unter dem Titel «All Elements of Application» enthält pro Diagrammtyp einen Ast und jeder dieser Äste kann aufgeklappt werden, um alle bestehenden Diagramme dieses Typs für die aktuell geöffnete Applikation anzuzeigen und zu selektieren. Das Package ist ein Container, ähnlich zu einem Branch in einem Source-Verwaltungswerkzeug wie beispielsweise Git und enthält die Änderungen, welche an den Diagrammen vorgenommen worden sind.

Die «Tool Box» enthält alle Elemente, die für die Modellierung verwendet werden können. Es werden jeweils kontextbezogen, zum aktuell selektierten Diagramm, verfügbare Elemente aufgelistet. In Abbildung 39 ist kein Diagramm selektiert, deshalb werden keine Elemente aufgelistet.

Die «Edit Area» visualisiert das geöffnete Diagramm. In diesem Bereich wird programmiert, indem Elemente aus der «Tool Box» zum Diagramm hinzugefügt werden. In Abbildung 39 ist kein Diagramm selektiert, deshalb wird nichts angezeigt. Die Edit-Area enthält im übertragenen Sinne den Quellcode der Anwendung. Das «Property Panel» zeigt jeweils weitere Eigenschaften eines selektierten Elements aus dem Modell an. Wird beispielsweise im Tabellen-Diagramm ein Attribut selektiert, können im «Property Panel» Datentyp, Beschreibung und weitere Eigenschaften festgelegt werden.

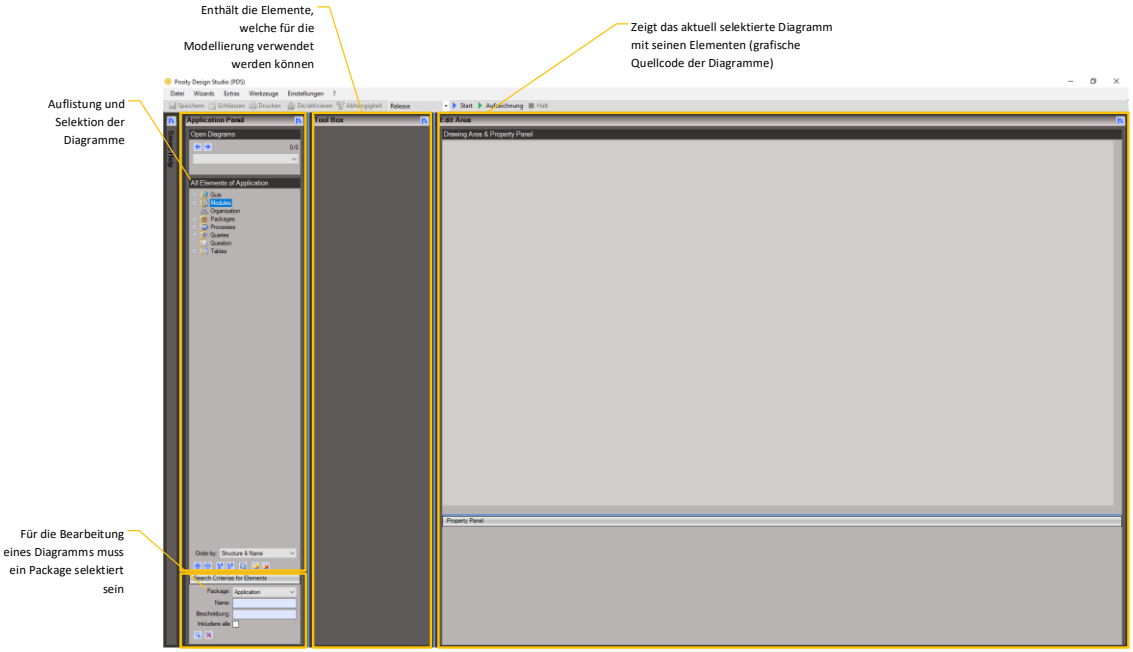


Abbildung 39: Übersicht Entwicklungsumgebung Posity Design Studio (Quelle: Eigene Darstellung)

7.1.1.2 Modellierung im Tabellen-Diagramm

Das Datenmodell beschreibt die Struktur aller Daten im Datenbanksystem. Jede Anwendung hat ihr eigenes Datenmodell. Posity verwendet ein relationales Datenbanksystem. Für die Modellierung des Datenmodells stehen Elemente wie Tabellen, Attribute, Primärschlüssel, Fremdschlüssel, Beziehungstypen etc. zur Verfügung. Der Diagramtyp für das Datenmodell unterscheidet sich zu den anderen Diagramtypen insofern, dass es nur genau eine Diagramminstanz für das Datenmodell gibt und nicht wie beispielsweise beim GUI mehrere Diagramminstanzen gibt, nämlich für jedes GUI der Applikation eine eigene Instanz.

In den folgenden vier Abbildungen (Abbildung 40, Abbildung 41, Abbildung 42 und Abbildung 43) wird am Beispiel eines Tabellen-Diagrammtyps im Posity Design Studio (PDS) aufgezeigt, welche Elemente für die Modellierung des Datenmodells zur Verfügung stehen und wie ein Modell aussehen könnte. Abbildung 40 zeigt in der Baumstruktur unter dem Titel «All Elements of Application» alle Tabellen des Modells. Abbildung 41, zeigt in der Auflistung unter dem Titel «Building Elements in Selected Folder» die Elemente, welche für das Modellieren des Diagramms verwendet werden können. In der Abbildung 42 ist das modellierte Diagramm zu sehen, welches unter anderem Tabellen-Elemente enthält. Die Bestandteile einer Tabelle sind in Abbildung 43 beschrieben.

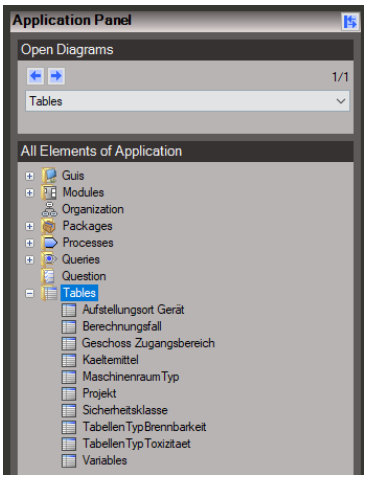


Abbildung 40: Tabellen-Diagramm mit Tabellen einer Applikation aufgelistet (Quelle: Eigene Darstellung)

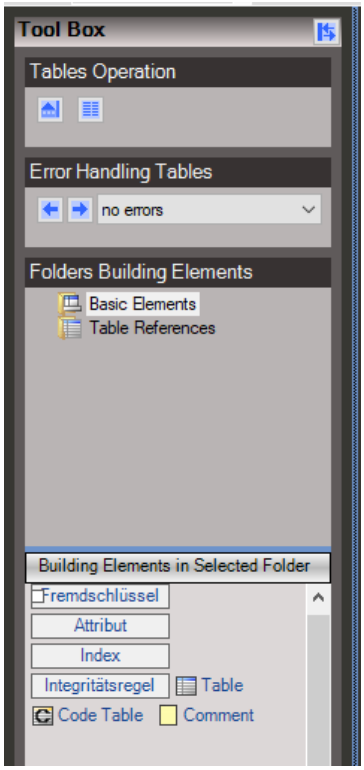


Abbildung 41: Elemente für die Modellierung des Diagrammtyps Datenmodell (Quelle: Eigene Darstellung)

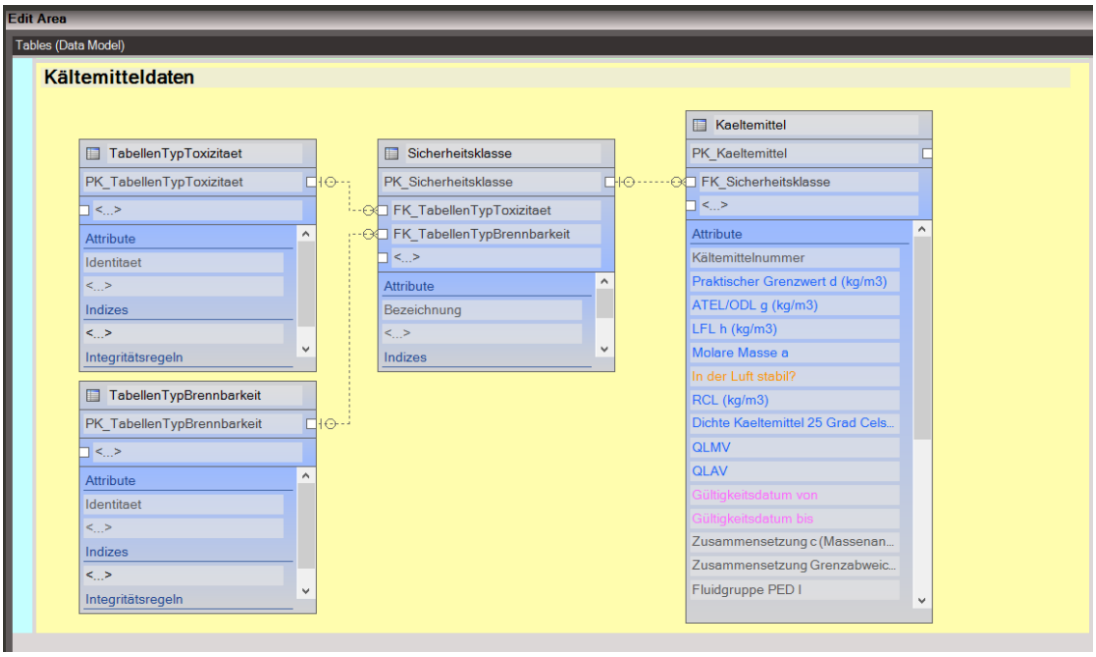


Abbildung 42: Teil eines modellierten Tabellen-Diagramms

(Quelle: Eigene Darstellung)

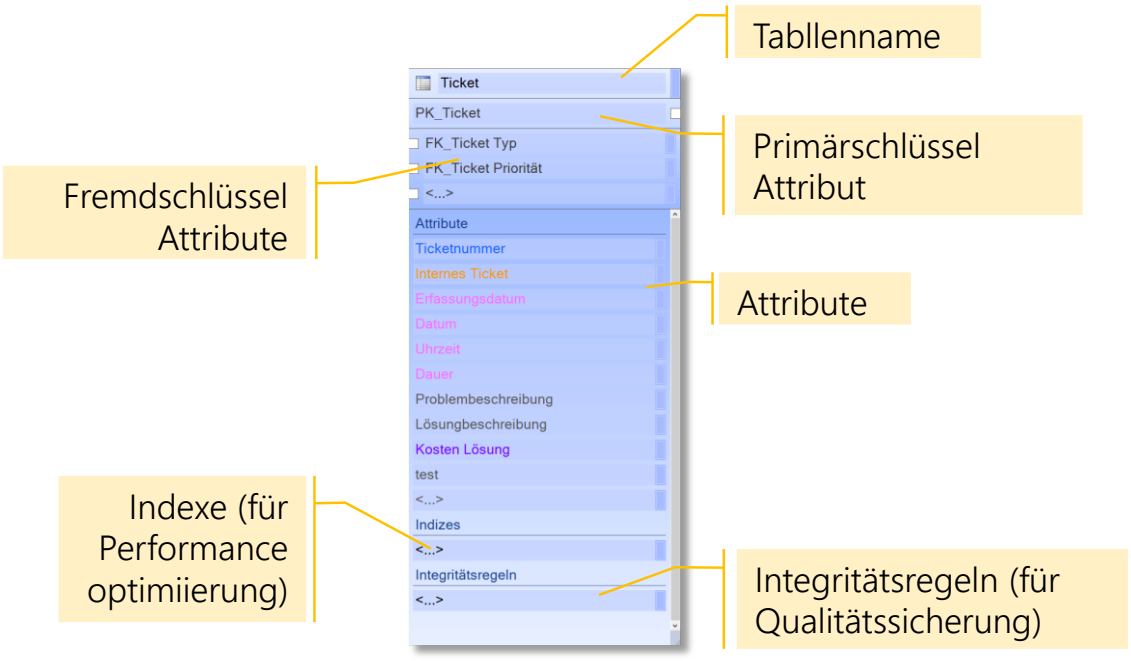


Abbildung 43: Tabelle und wichtige Bestandteile (Quelle: Eigene Darstellung)

7.1.1.3 Modellierung im Query-Diagramm

Im Query-Diagramm werden Abfragen formuliert, so dass die Daten der Tabellen ausgelesen werden können. Anders als im Tabellen-Diagramm, wo es nur genau eine Diagramminstanz für das Datenmodell gibt, können beim Query-Diagrammtyp diverse Diagramminstanzen erstellt werden. Die Abbildung 44 zeigt in der Baumstruktur unter dem Titel «All Elements of Application» einen Ast pro Diagrammtyp und im aufgeklappten Ast «Queries» alle bestehenden konkreten Diagramme dieses Typs für die aktuell geöffnete Applikation. Jedes Query-Diagramm Abbildung 44repräsentiert dabei eine individuelle Sicht auf die Daten. Aus dem Query-Diagramm wird SQL (je Query-Diagramm eine Stored Procedure) generiert, das auf dem SQL-Server ausgeführt wird. Das Resultat der Ausführung ist eine Menge von Tabellen mit Daten (ein Dataset). Die Query-Diagramme stehen in einer starken Abhängigkeit zum Tabellen-Diagramm, da die dort modellierten Tabellen als Basis für die Abfragen dienen. Abbildung 45 zeigt welche Tabellen für die Formulierung der Query zur Verfügung stehen. In der Abbildung 46 ist schliesslich das modellierte Diagramm zu sehen, welches unter anderem ein Query-Table-Element und zwei Query-Parameter und ein Filter-Element enthält. In Abbildung 47 sind einige wichtige Bestandteile des leicht komplexeren Query-Diagramms beschrieben.

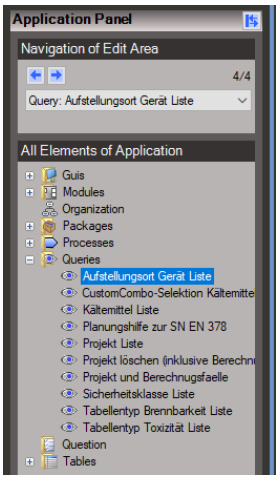


Abbildung 44: Query-Diagramme einer Applikation aufgelistet
(Quelle: Eigene Darstellung)

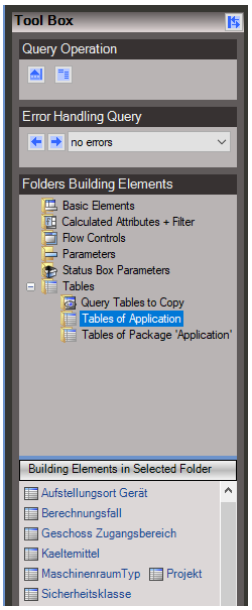


Abbildung 45: Elemente für die Modellierung von Diagrammtyp Query
(Quelle: Eigene Darstellung)

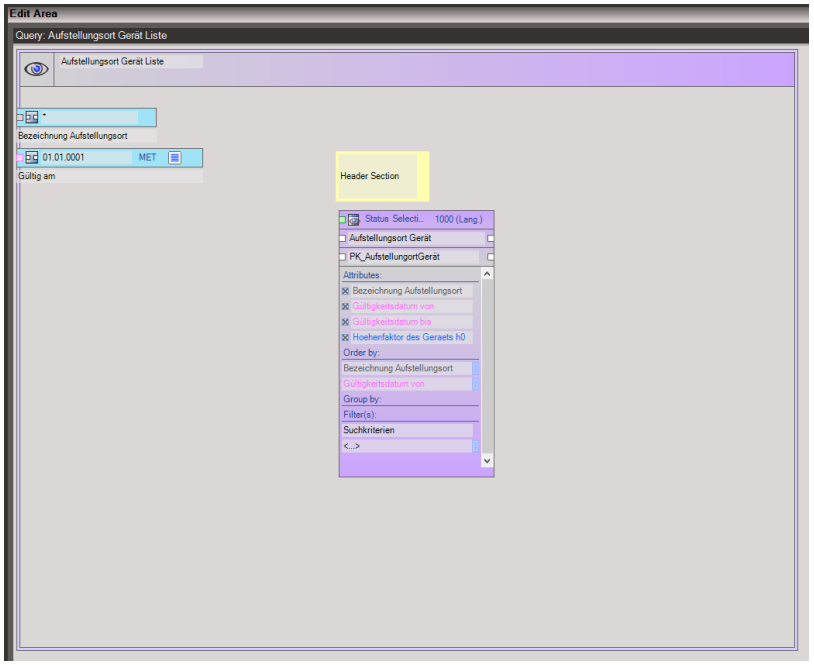


Abbildung 46: Modelliertes Query-Diagramm mit diversen Elementen
(Quelle: Eigene Darstellung)

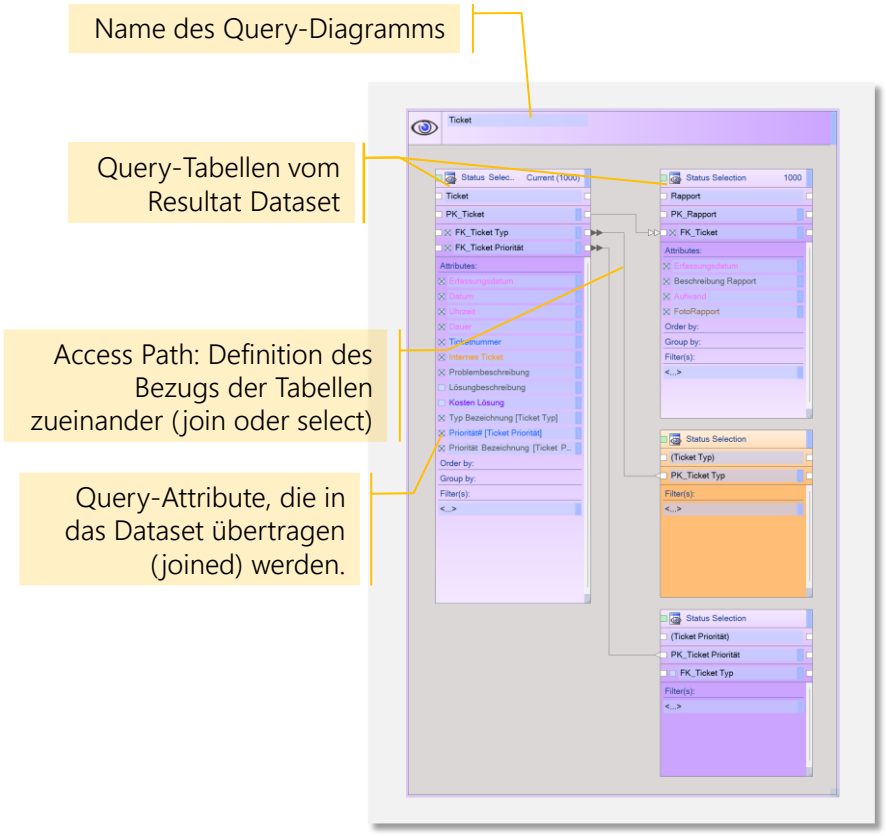


Abbildung 47: Query-Diagramm und wichtige Bestandteile
(Quelle: Eigene Darstellung)

7.1.1.4 Modellierung im GUI-Diagramm

Mit dem GUI-Diagramm wird die grafische Benutzerschnittstelle modelliert. Der Editor funktioniert nach dem WYSIWYG-Prinzip (What you See Is What You Get). Die GUI-Diagramme können die Query-Diagramme verwenden, um die Anzeige und Bearbeitung der Daten, welche in der Datenbank persistiert sind, zu ermöglichen.

Ein GUI-Diagramm kann folgende Komponenten enthalten:

- Eine durch eine Query-Abfrage erstellter Datensatz (als Liste, einzelnes Attribut, Diagramm oder Kalender), bestehend aus mehreren Tabellen
- GUI-Variablen, die von einem Modul gelesen und geändert werden können (ohne Bezug auf einen Datensatz)
- Schaltflächen (Buttons) zum Auslösen von Ereignissen (z.B. Speichern der Daten)

- Elemente zur visuellen Gestaltung (Gruppen-, Split-, Tabulator-Panels und konstanter Text).

Die Abbildung 48 zeigt in der Baumstruktur unter dem Titel «All Elements of Application» im aufgeklappten Ast «GUIs» alle bestehenden Diagramme vom Typ GUI-Diagramm für die aktuell geöffnete Applikation. Die Abbildung 49 zeigt in der Auflistung unter dem Titel «Building Elements in Selected Folder» die Shapes, welche zur Kategorie «Controls» gehören und für das Modellieren des aktuell ausgewählten Diagramms mit dem Namen «Kältemittel Liste» verwendet werden können. In der Abbildung 50 ist schliesslich das modellierte Diagramm zu sehen, welches unter anderem ein Button-Shape, ein Gruppenpanel-Shape, drei Attribute-Shapes und ein Tabellen-Shape enthält. In Abbildung 51 sind einige wichtige Bestandteile des GUI-Diagramms beschrieben.

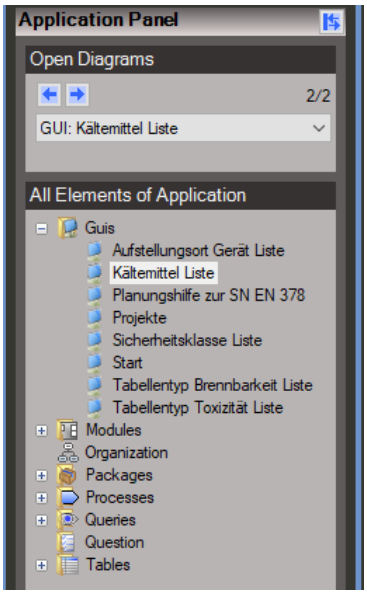


Abbildung 48: GUI-Diagramme einer Applikation aufgelistet (Quelle: Eigene Darstellung)

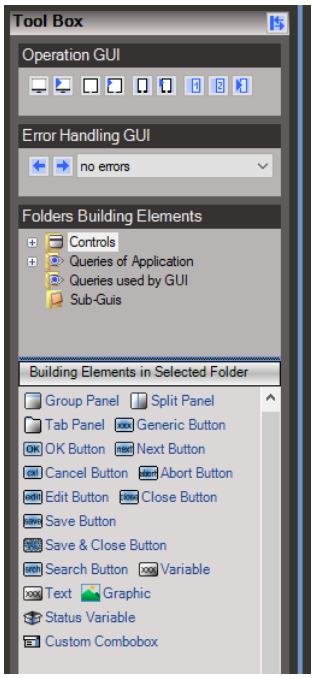


Abbildung 49: Shapes für die Modellierung von Diagrammtyp GUI (Quelle: Eigene Darstellung)

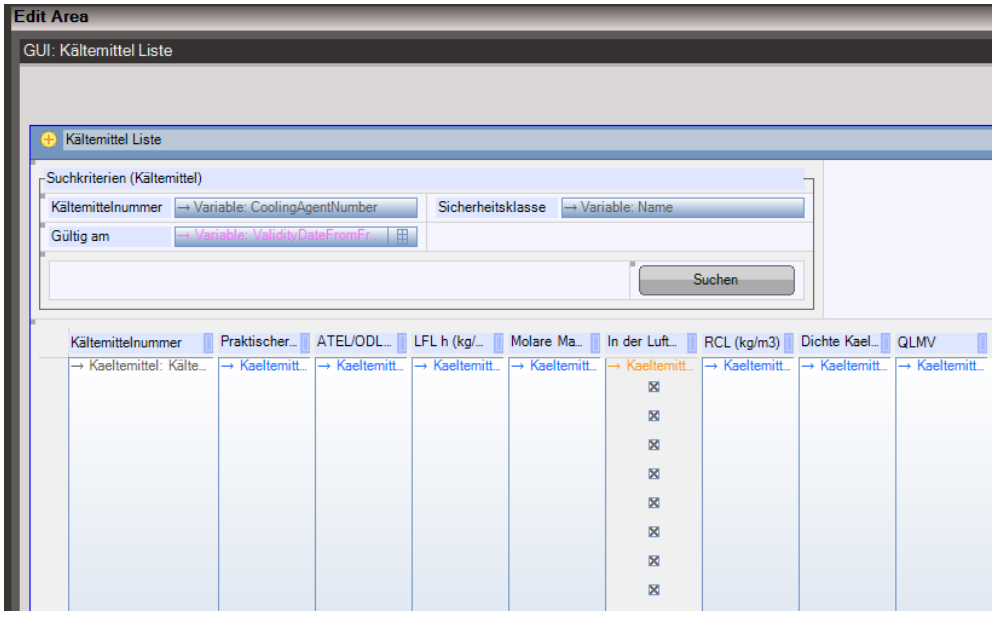


Abbildung 50: Modelliertes GUI-Diagramm mit diversen Shapes (Quelle: Eigene Darstellung)

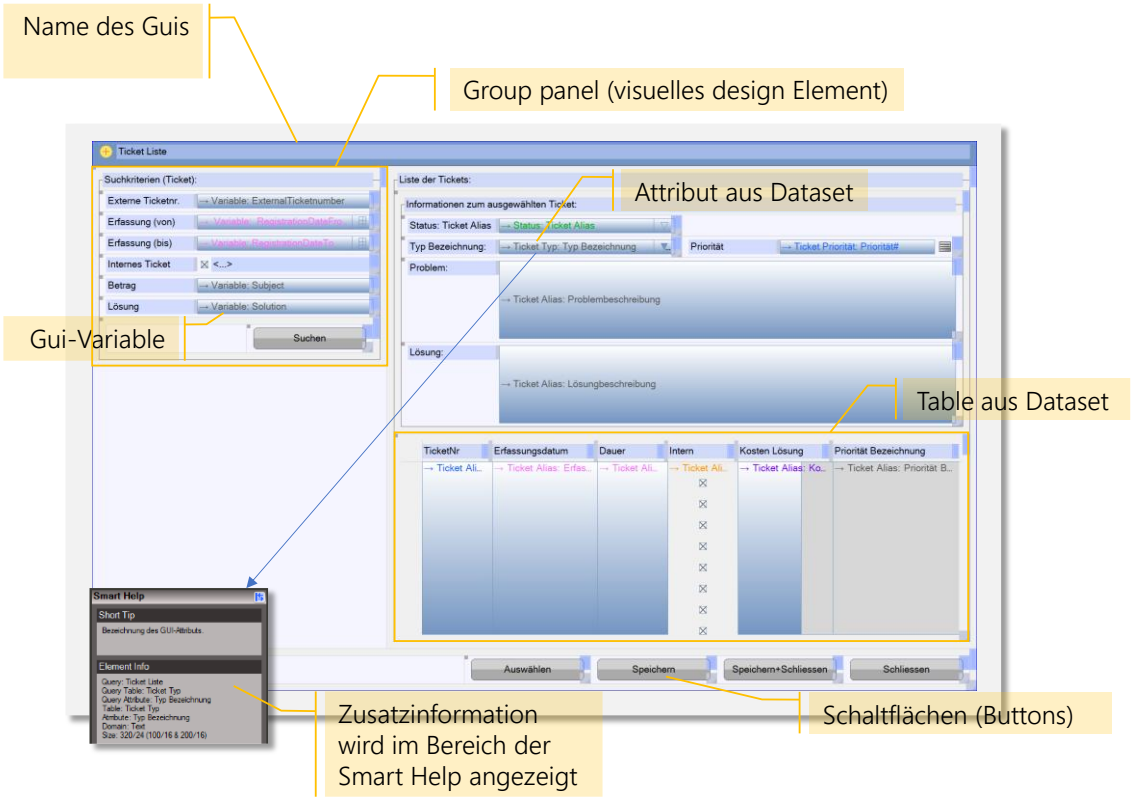


Abbildung 51: GUI-Diagramm und wichtige Bestandteile

(Quelle: Eigene Darstellung)

7.1.1.5 Modellierung im Modul-Diagramm

Das Modul-Diagramm definiert die Businesslogik der Applikation und kann als zentrale Komponente einer Posity-Applikation angesehen werden, in der alle Fäden zusammenlaufen. Dieser Diagramtyp bietet das grösste Potential für die Code-Analyse, weil in diesem Diagrammtyp am meisten Code entsteht und die Freiheiten in der Umsetzung am grössten sind. Ein Modul-Diagramm kann aus den folgenden fünf Teilen bestehen:

- Die Ereignisse (Events)
- Funktionen zur Verarbeitung von Daten oder zum Auslösen von Aktivitäten
- Der Datenfluss der zu verarbeitenden Informationen
- Variablen, Konstanten und Parameter des Moduls und der Ereignisse
- Ablaufsteuerungen zur Beeinflussung der Verarbeitung im Datenfluss

Ereignisse (Events) eines Moduls

- Wenn ein Modul gestartet wird (z.B. durch Aktivierung eines Prozesses mit angeschlossenem Modul), wird das Ereignis "Start" ausgeführt (immer).
- Werden Schaltflächen im GUI gedrückt, werden die entsprechenden, verknüpften Ereignisse ausgelöst (seriell).
- GUI-Elemente (z.B. Attribute von Abfragetabellen) können zusätzlich in bestimmten Situationen Ereignisse auslösen (z.B. Betreten oder Verlassen eines Feldes, Änderung von Daten, usw.).

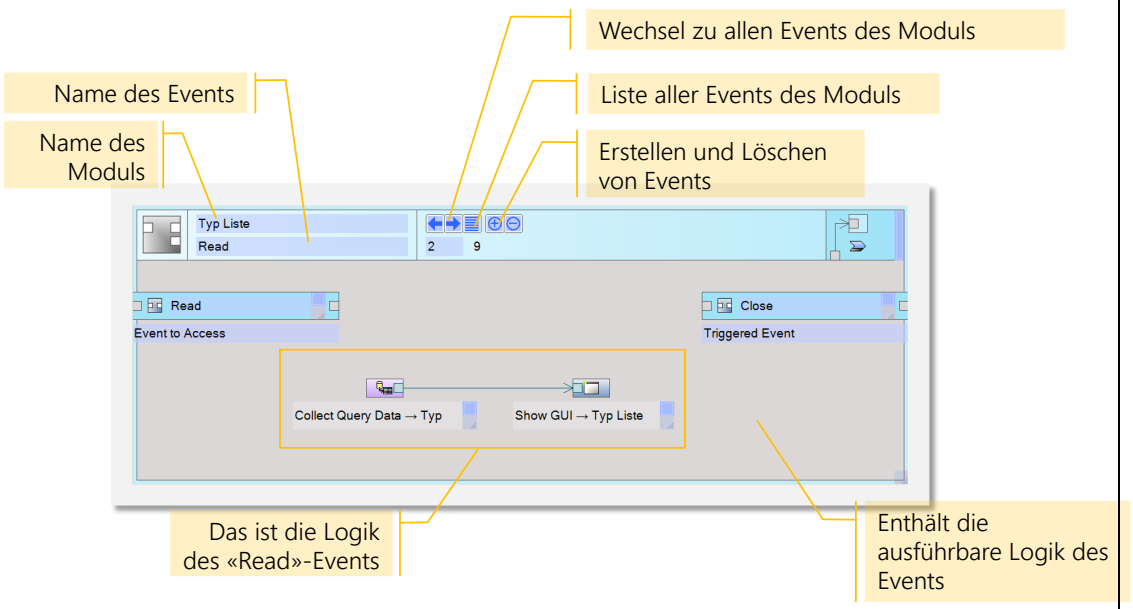


Abbildung 52: Beispiel Event "Read" in Modul "Typ-Liste"
(Quelle: Eigene Darstellung)

Funktionen

- Funktionen haben Eingangs- und Ausgangsdaten und/oder Eigenschaften.
- Funktionen verarbeiten Daten oder lösen Aktivitäten aus.

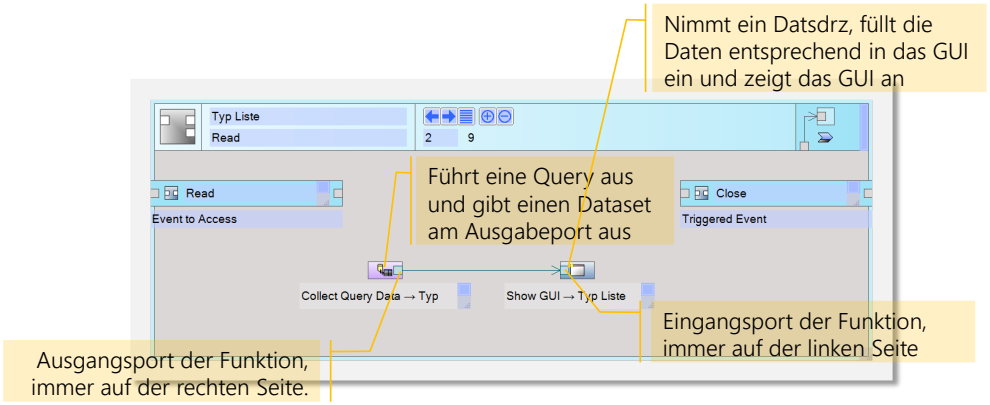


Abbildung 53: Beispiel von zwei Modul-Funktionen (Quelle: Eigene Darstellung)

- Die Liste der verfügbaren Funktionen ist lang.
- Es ist möglich, eigene Funktionen zu integrieren.

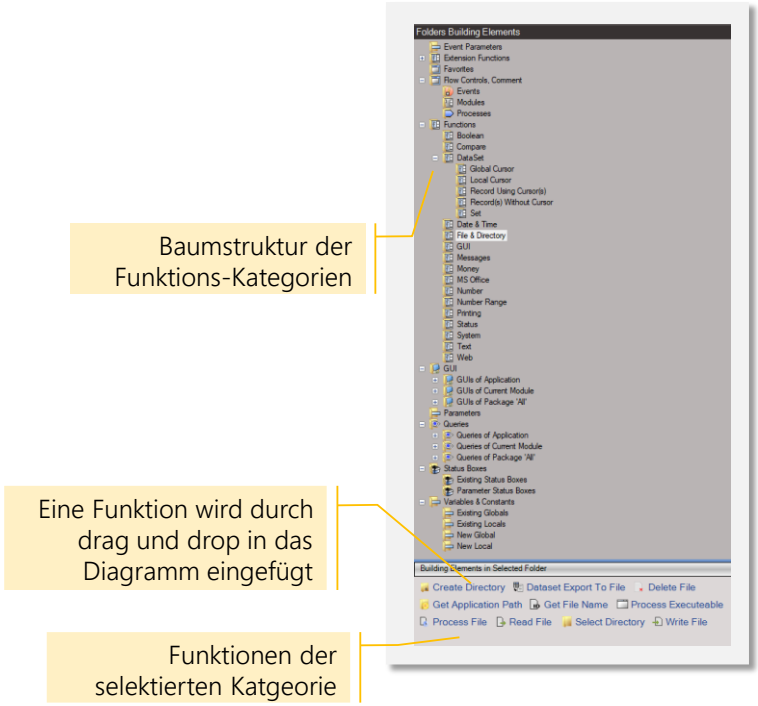


Abbildung 54: Ausschnitt der zur Verfügung stehenden Modul-Funktionen (Quelle: Eigene Darstellung)

Datenfluss

- Ein Datenfluss nimmt Daten von einem Ausgangsport und sendet sie an einen oder mehrere Eingangsports.
- Datenflüsse haben keine Eigenschaften.
- Datenflüsse haben eine Farbe, die der Art der übertragenen Daten entspricht.

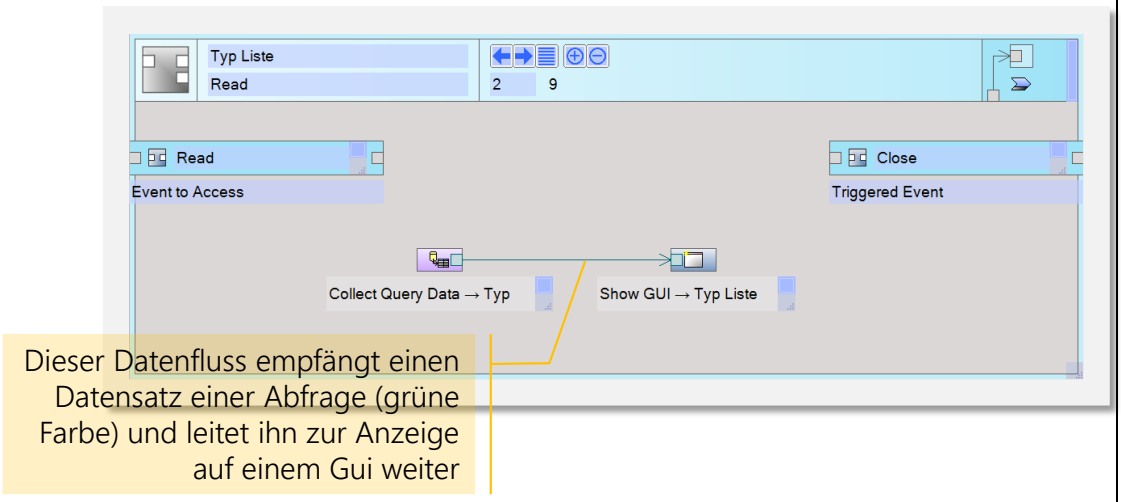


Abbildung 55: Beispiel eines Datenflusses im Modul (Quelle: Eigene Darstellung)

Variablen, Konstanten und Parameter

- Konstanten haben nur einen Ausgangsport. Variablen haben einen Eingangs- und einen Ausgangsport.
- Parameter sind Variablen, die gelesen oder geschrieben werden, wenn ein Ereignis oder ein Modul aufgerufen oder verlassen wird.
- Modulparameter sind in allen Events des Moduls verfügbar, Ereignisparameter nur in den entsprechenden Events.
- Variablen, Konstanten und Parameter haben weitere Eigenschaften, die vom Datentyp abhängen.

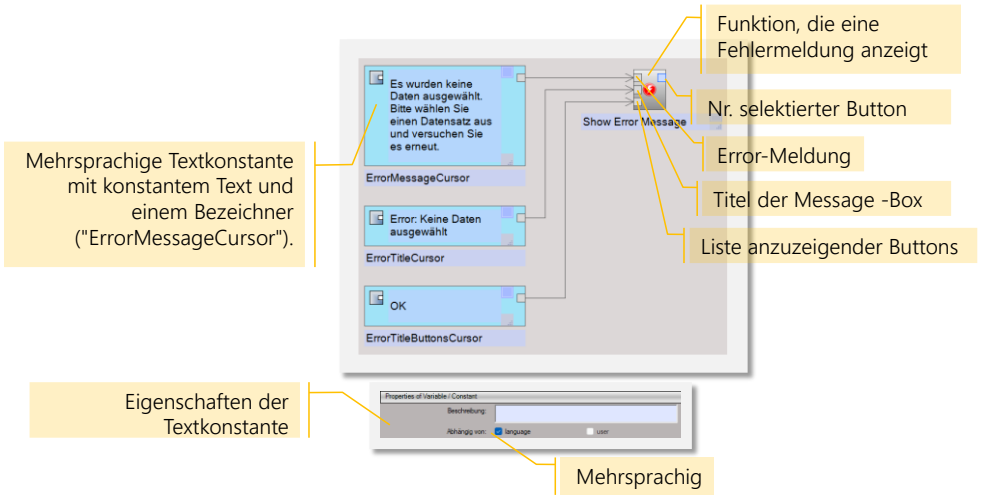


Abbildung 56: Beispiel der Verwendung von Konstanten in Modul (Quelle: Eigene Darstellung)

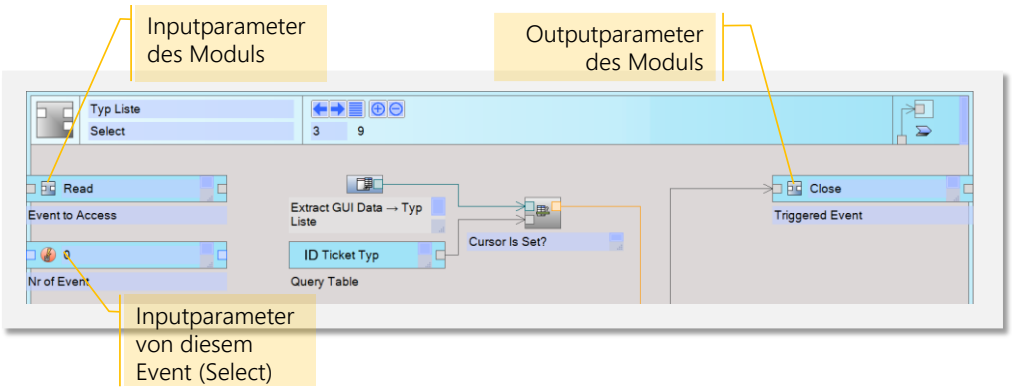


Abbildung 57: Beispiel der Verwendung von Parameter in Modul (Quelle: Eigene Darstellung)

Ablaufsteuerung

Ein Datenfluss muss die Möglichkeit von Verzweigungen und Wiederholungen haben. Posity kennt die folgenden sechs Typen:

- Aufrufen von Ereignissen, Modulen oder Prozessen.

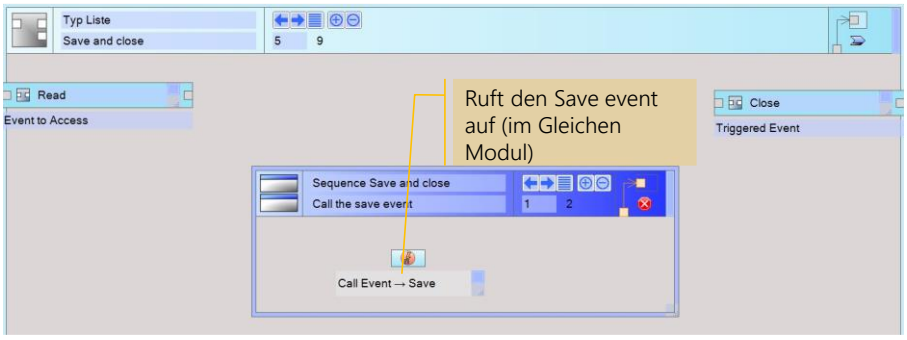


Abbildung 58: Aufrufen von Ereignissen, Modulen oder Prozessen (Quelle: Eigene Darstellung)

- Fallunterscheidung: Auswahl eines auszuführenden Codeblocks
  - In den Eigenschaften kann ausgewählt werden, ob es sich um einen Booleschen-, einen Text- oder einen Zahlen-Case handelt.

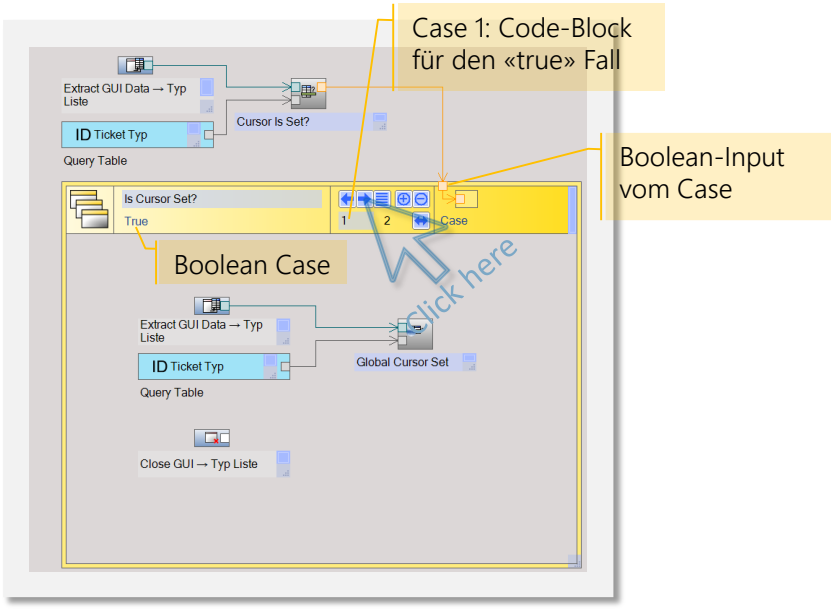


Abbildung 59: Ein Beispiel von einem if-else-Case (Quelle: Eigene Darstellung)

Ablaufsteuerung

- For-Schleife: Wiederholung eines bestimmten Codeblocks
    - Führt denselben Code eine bestimmte Anzahl von Malen aus.
    - Im Beispiel wird der Code 10-mal ausgeführt. i iteriert von 1 bis 10.

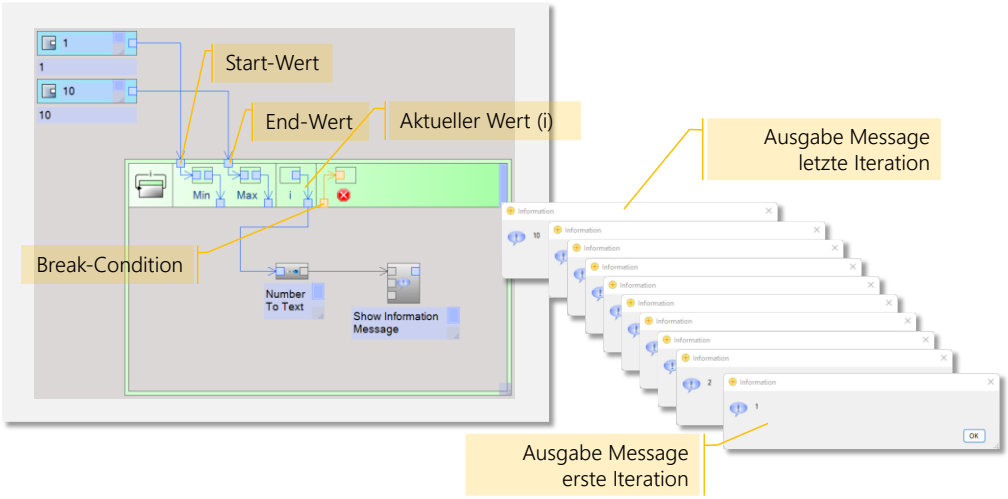


Abbildung 60: Beispiel For-Schleife in Modul (Quelle: Eigene Darstellung)

- For-each-Schleife: Wiederholung eines bestimmten Codeblocks für jeden Datensatz einer Tabelle in einem Dataset
    - Der lokale Cursor bewegt sich schrittweise durch die Daten einer Tabelle vom ersten bis zum letzten Datensatz.

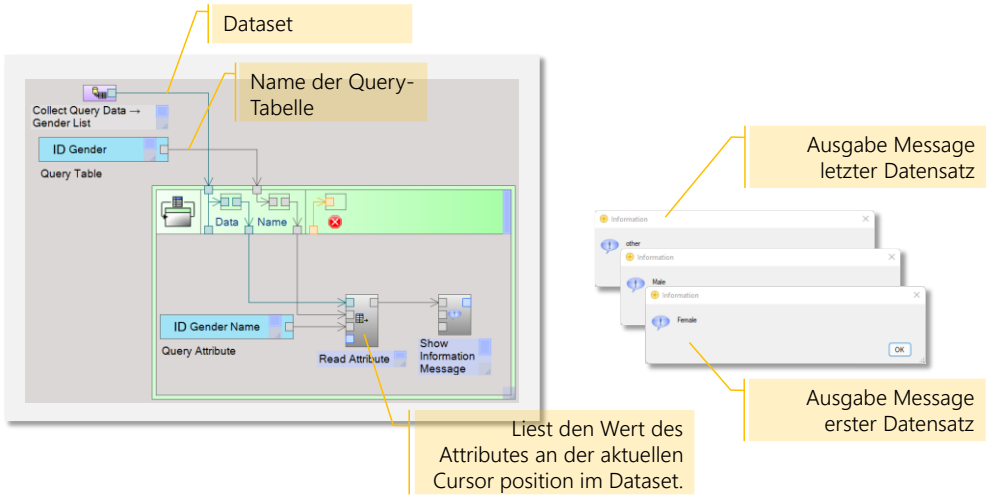


Abbildung 61: Beispiel einer For-Each-Schleife in Modul
(Quelle: Eigene Darstellung)

Ablaufsteuerung

- Sequence: Mehrere Codeblöcke nacheinander ausführen
    - Wenn der Code sequenziell ausgeführt werden muss. Im Modul ist die Ausführungsreihenfolge nicht deterministisch.
    - Auch zur besseren Strukturierung von grossen Modulen geeignet.

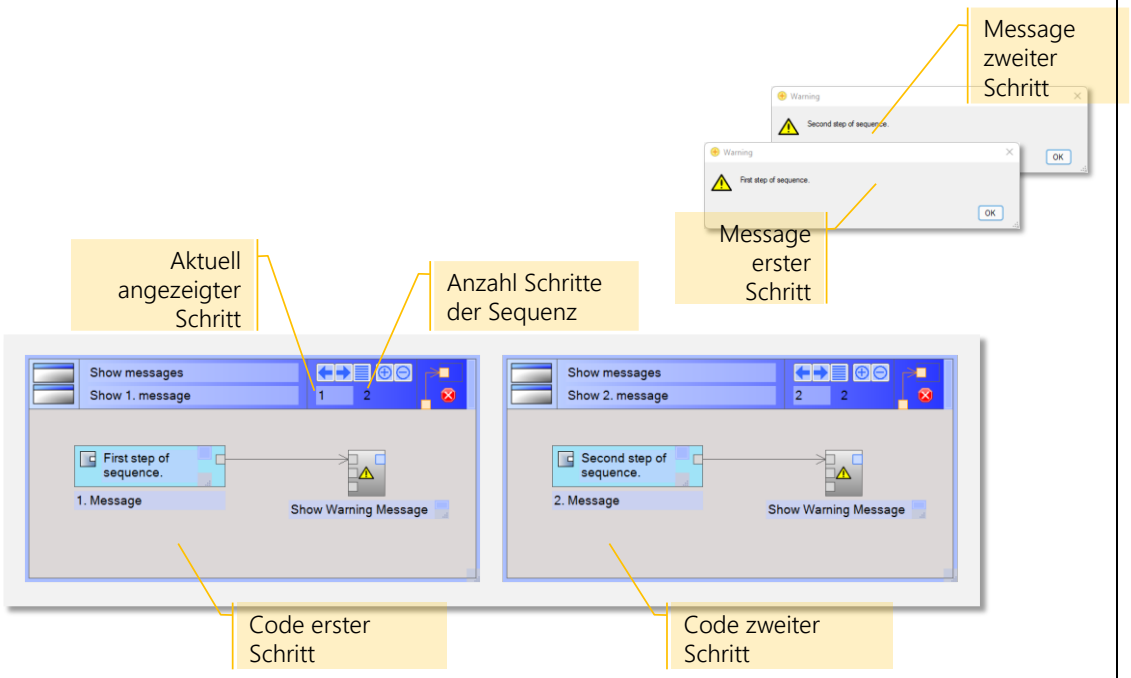


Abbildung 62: Beispiel einer Sequenz in Modul (Quelle: Eigene Darstellung)

Ablaufsteuerung

- Try-catch: Welcher Code soll im Fehlerfall ausgeführt werden
  - Es wird versucht den Code im try-Block auszuführen. Wenn ein Fehler auftritt, wird der catch-Block ausgeführt.
  - Wenn ein Fehler ohne try-catch auftritt, wird das Modul abgebrochen - andere Module sind davon nicht betroffen.

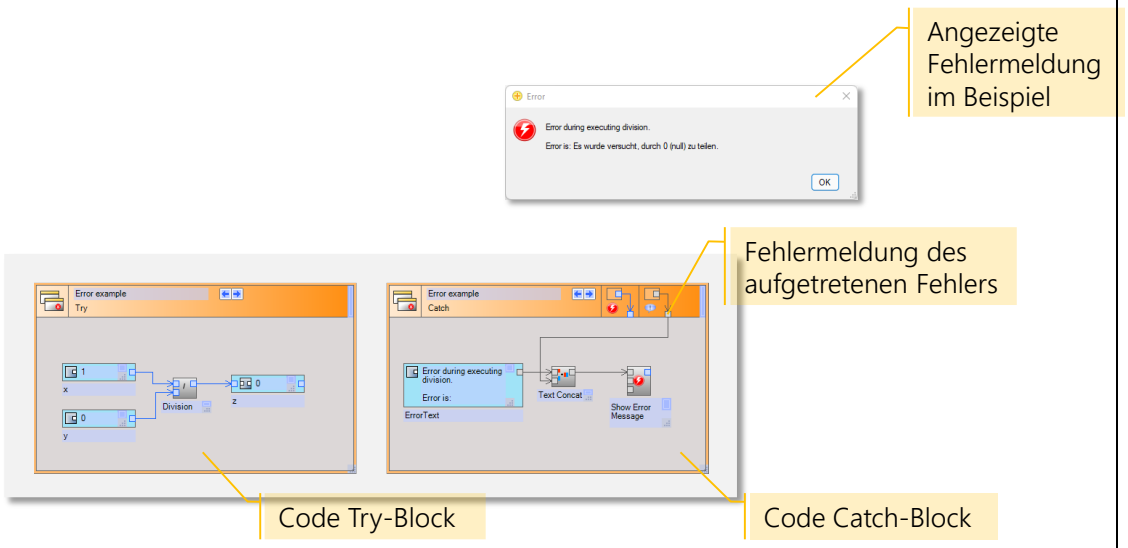


Abbildung 63: Beispiel try-catch im Modul (Quelle: Eigene Darstellung)

7.1.2 Bestehende Qualitätskontrollen im Posity Design Studio

7.1.2.1 Error-Check

Alle Diagramme führen, sobald sie eine Änderung erfahren, einen Error-Check durch, dieser überprüft die Shapes auf dem Diagramm auf Fehler, welche die Ausführung des Diagramms verhindern würden – entspricht der Funktion eines Compilers. Die Fehler werden in einer Dropdown-Box aufgelistet und müssen vor der Ausführung der Anwendung beseitigt werden. Beispiele für solche Fehler können nicht verbundene, obligatorische Eingangsports von Funktionen sein, fehlende oder mehrfach vorkommende Variablennamen oder die Zuweisung von inkompatiblen Datentypen sein. Die Aufzählung ist bei weitem unvollständig. Jedes Shape auf jedem Diagramm wird auf diverse Fehler geprüft und bei nicht Erfüllen in der Fehlerliste erfasst. Abbildung 64 zeigt ein Szenario, bei welchem ein Modul-Diagramm zwei Fehler aufweist. Bei der Fallunterscheidung

(gelbes Rechteck) ist der Eingangsport für den booleschen Wert nicht verbunden und damit fehlt dem Shape die Entscheidungsgrundlage. Der zweite Fehler betrifft die Verbindungslinie, welche vom Ausgangsport der Funktion «Cursosr Is Set» weggeführt, aber nicht zu einem Eingangsport geführt wird und somit im Nichts endet. Fehlerhafte Shapes werden im Diagramm rot dargestellt. Diese Fehler müssen behoben werden, damit der fCode vom Diagramm generiert und die Applikation ausgeführt werden kann.

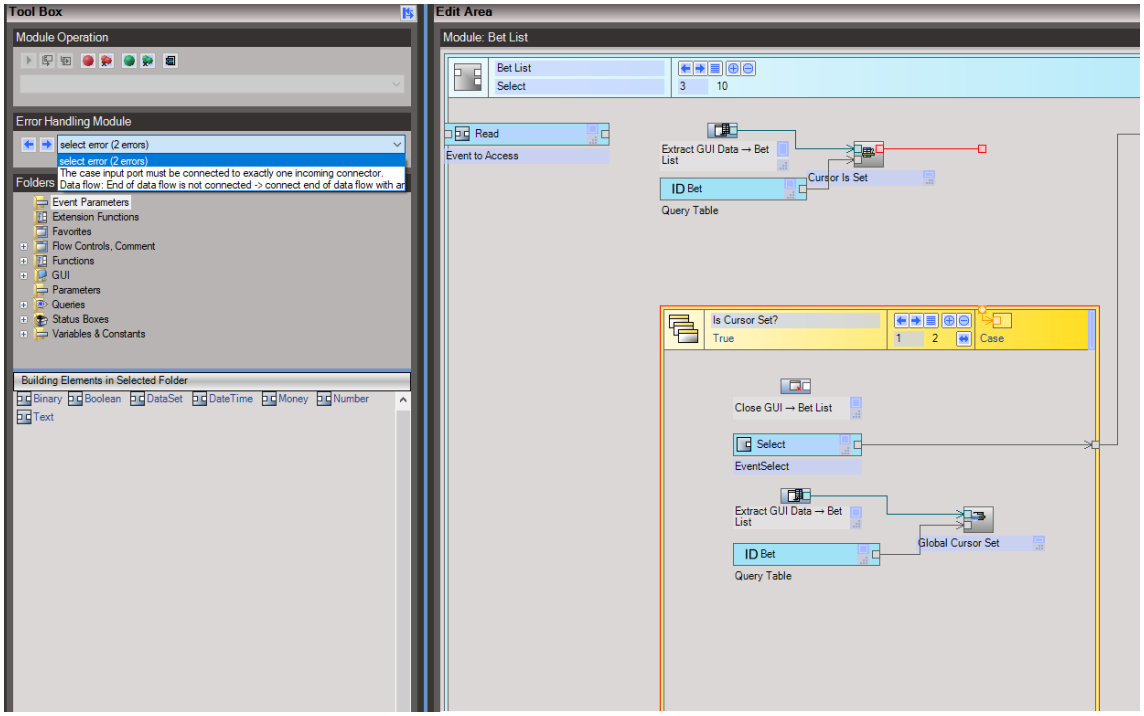


Abbildung 64: Error-Checker in Diagrammen (Quelle: Eigene Darstellung)

7.1.2.2 Einschränkungen gegeben durch das Posity Design Studio

- Wichtige Aspekte der Architektur werden durch die Diagramme vorgegeben, so sind z.B. typische Trennungen, wie z.B. die Trennung von Logik und GUI, per Design gegeben.
- Sicherheitskritische Aspekte wie Verschlüsselung sind vorgegeben.
- Transaktionslogik zu einem grossen Teil vorgegeben.

Trotzdem können viele andere Bereiche, wie beispielsweise die Einhaltung von bestimmten Namenskonventionen, nicht überprüft werden. Ebenso wenig werden Diagramme auf Komplexität und Struktur oder ungenutzte Code-Teile untersucht.

7.1.3 Elemente von Interesse für die Quellcode-Analyse

Um eine Analyse am Quellcode durchführen und auswerten zu können, werden die folgenden Elemente eine zentrale Rolle spielen. Nachfolgend wird die Bedeutung der jeweiligen Elemente generell und deren Rolle im Kontext der Quellcode-Analyse beschrieben.

Posity Design Studio: Das Posity Design Studio (PDS) spielt eine wichtige Rolle für die Quellcode-Analyse, weil es als zentrale Einheit für die Programmierung, Konfiguration und Deployment von Posity-Applikationen eingesetzt wird.

Die Quellcode-Analyse wird ebenfalls im PDS integriert sein.

Applikation: Die Applikation steht für die Einheit einer Anwendung. Eine Anwendung besteht aus mehreren ausführbaren Diagrammen. Im PDS ist jeweils maximal eine Applikation und Umgebung geöffnet.

Für die Quellcode-Analyse kann als Betrachtungsraum (Scope) die gesamte Applikation von Interesse sein. Das würde bedeuten, dass der aller Code (alle Diagramme) aller Umgebungen dieser Applikation analysiert wird.

Umgebung: Die Umgebung (Bereitstellungsumgebung) ist ein Subelement der Applikation, wobei eine Applikation aus mehreren Umgebungen (z.B. Entwicklung, Test und Produktion) bestehen kann. Dies wird vor allem für verschiedene Entwicklungsfortschritte genutzt oder wenn für die gleiche Anwendung mehrere Mandanten benötigt werden. Die Code-Basis einer Umgebung kann jederzeit auf alle anderen Umgebungen verteilt (deployed) werden. So wird die Konstellation einer Entwicklungs-, Test- und Produktions-Umgebung häufig verwendet, um ein Feature in der Entwicklungs-Umgebung zu programmieren, in der Test-Umgebung zu testen und anschliessend als Release in die produktive Umgebung zu verteilen.

Für die Quellcode-Analyse kann der Betrachtungsraum (Scope) auf die Umgebung von Interesse sein. Das würde bedeuten, dass der Code (alle Diagramme) dieser Umgebungen analysiert wird.

Package: Das Package ist ein Container, ähnlich zu einem Branch in einem Source-Verwaltungswerkzeug wie beispielsweise Git und enthält die Änderungen, welche an den Diagrammen vorgenommen worden sind. Für die Bearbeitung eines Diagramms muss ein Package ausgewählt sein.

Für die Quellcode-Analyse ist der Betrachtungsraum (Scope) auf ein Package von Interesse. Es würde damit der Code (alle Diagramme) des Packages analysiert. Das kann beispielweise dann spannend sein, wenn in einem Package ein neues Feature implementiert wurde und vor dem Deployment in die Test- oder Produktions-Umgebung eine Analyse durchgeführt wird.

Diagramm: Das einzelne Diagramm ist jeweils von einem bestimmten Typ (GUI, Module, etc.) und wird mit Hilfe von diversen Shapes modelliert. Im PDS ist jeweils ein Diagramm in der «Edit-Area» geladen und kann bearbeitet werden.

Für die Quellcode-Analyse ist der Betrachtungsraum (Scope) auf ein Diagramm von Interesse. Es würde damit der Code (alle Shapes) des Diagramms analysiert. Das kann beispielweise dann spannend sein, wenn ein Diagramm bearbeitet wird und fortwährend eine Analyse durchgeführt wird.

Shape: Das einzelne Shape ist der kleinste Baustein für die Modellierung der Diagramme. Durch die Kombination von mehreren Shapes werden die Diagramme modelliert. Verschiedenen Diagrammtypen stehen verschiedene Shapes zur Verfügung und diese können aus der Tool-Box mittels drag and drop im Diagramm platziert werden.

Für die Quellcode-Analyse ist der Betrachtungsraum (Scope) eines Shapes deshalb relevant, als dass sich dort die Problemzonen befinden werden. Während einer Analyse werden die einzelnen Shapes und Ihre Umgebung auf Probleme untersucht und im Report entsprechend adressiert. Weil Shapes eine zentrale Rolle spielen, wird im Kapitel 7.1.4 näher auf dieses Element eingegangen

Subshape: Ein Shape kann auch aus mehreren Subshapes bestehen. Subshapes selbst können nicht ohne Shape existieren und erscheinen deshalb auch nicht als eigenes Element in der Tool-Box der Diagramme.

Für die Quellcode-Analyse ist der Betrachtungsraum (Scope) eines Subshapes deshalb relevant, als dass sich, analog wie beim Shape, auch dort die Problemzonen befinden können. So ist beispielsweise der Name einer Variablen im Modul im Subshape `VariableNameTextBox` gespeichert und gehört zum Shape `VariableShape` Während einer Analyse werden die einzelnen Sub-Shapes und Shapes auf Probleme untersucht und im Report entsprechend adressiert.

7.1.4 Posity Shape

Posity Shapes werden für die Modellierung der Diagramme verwendet. Die acht Diagrammtypen haben jeweils unterschiedliche Elemente zur Auswahl, um die Modelle zu erstellen (siehe Kapitel 7.1.1). Diese Elemente werden im Weiteren auch Posity Shapes oder nur Shapes genannt. So können beispielsweise im GUI-Diagramm `Button`-Shapes, `Panel`-Shapes oder `Attribute`-Shapes verwendet werden, um eine grafische Benutzeroberfläche zu modellieren. Im Modul-Diagramm stehen unter Anderem `GuiComponent`-Shapes, `Variable`-Shapes, `Standardfunction`-Shapes, `DataFlow`-Shapes oder `QueryComponent`-Shapes zur Verfügung, um die Businesslogik der Anwendung zu modellieren. Oder im Tabellen-Diagramm werden für die Modellierung des Datenmodells `Table`-Shapes, `Primarykey`-Shapes, `Foreignkey`-Shapes oder `TableAttribute`-Shapes zur Verfügung gestellt. Die Aufzählung ist nicht vollständig, soll aber den Zweck der verschiedenen Shape-Implementationen vermitteln.

Das Posity-Shape ist das zentrale Element in der Architektur aller Diagrammtypen und weil der Quellcode der Diagramme aus einer Kombination von Shapes besteht, ebenso das zentrale Element der Code-Analyse. Die Shapes können, wie beispielweise Code-Zeilen in einer textbasierten Programmiersprache, identifiziert und referenziert werden. Ein Sub-Shape wird auf dem Diagramm als grafischer Teil des Shapes visualisiert. Abbildung 65 zeigt ein Beispiel von einem Shape und Sub-Shape im Modul-Diagramm.

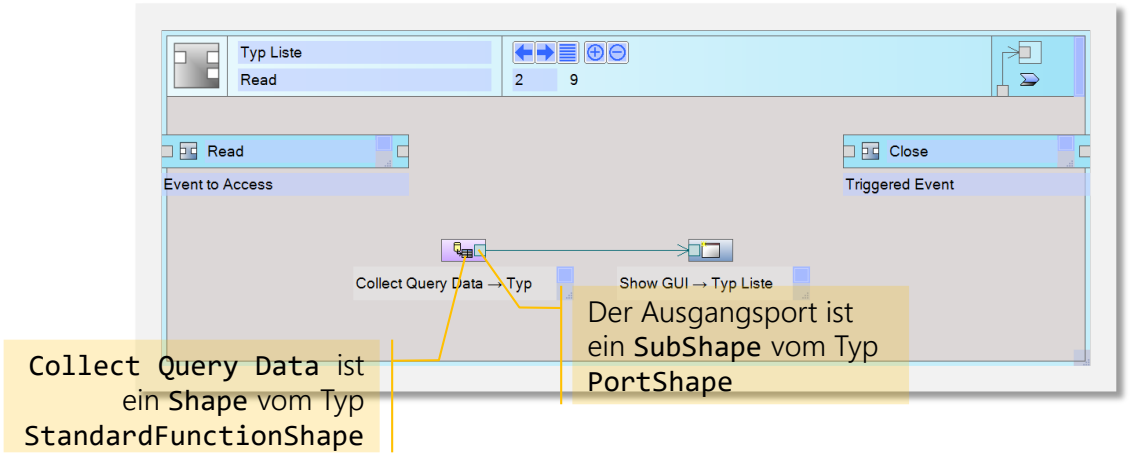


Abbildung 65: Shape und Sub-Shape (Quelle: Eigene Darstellung)

7.1.4.1 Die Klassenhierarchie von Shape

Die Klasse `Shape` hat insgesamt 208 Subklassen, die Klasse `SubShape` wird von 66 Klassen erweitert, wobei `SubShape` auch eine Subklasse von `Shape` ist. Die grosse Zahl der Erweiterungen von Shape und `SubShape` zeigt auf, wie zentral die Shape-Klassen für die Entwicklung der Diagramme sind. In Abbildung 66 wird ein kleiner Auszug der Klassenhierarchie von `Shape` und `SubShape`, gruppiert nach Diagrammtypen, abgebildet. In der Abbildung ist zusätzlich ersichtlich, wie das Diagramm mit den Shapes in Abhängigkeit steht. Die Klasse `Diagram`, ist jeweils von einem, der im Enum `DiagramType` beschriebenen Typen (`GUI`, `Modul`, `Query`, etc.) und enthält eine Liste aller Shapes, die auf dem Diagramm eingefügt worden sind.

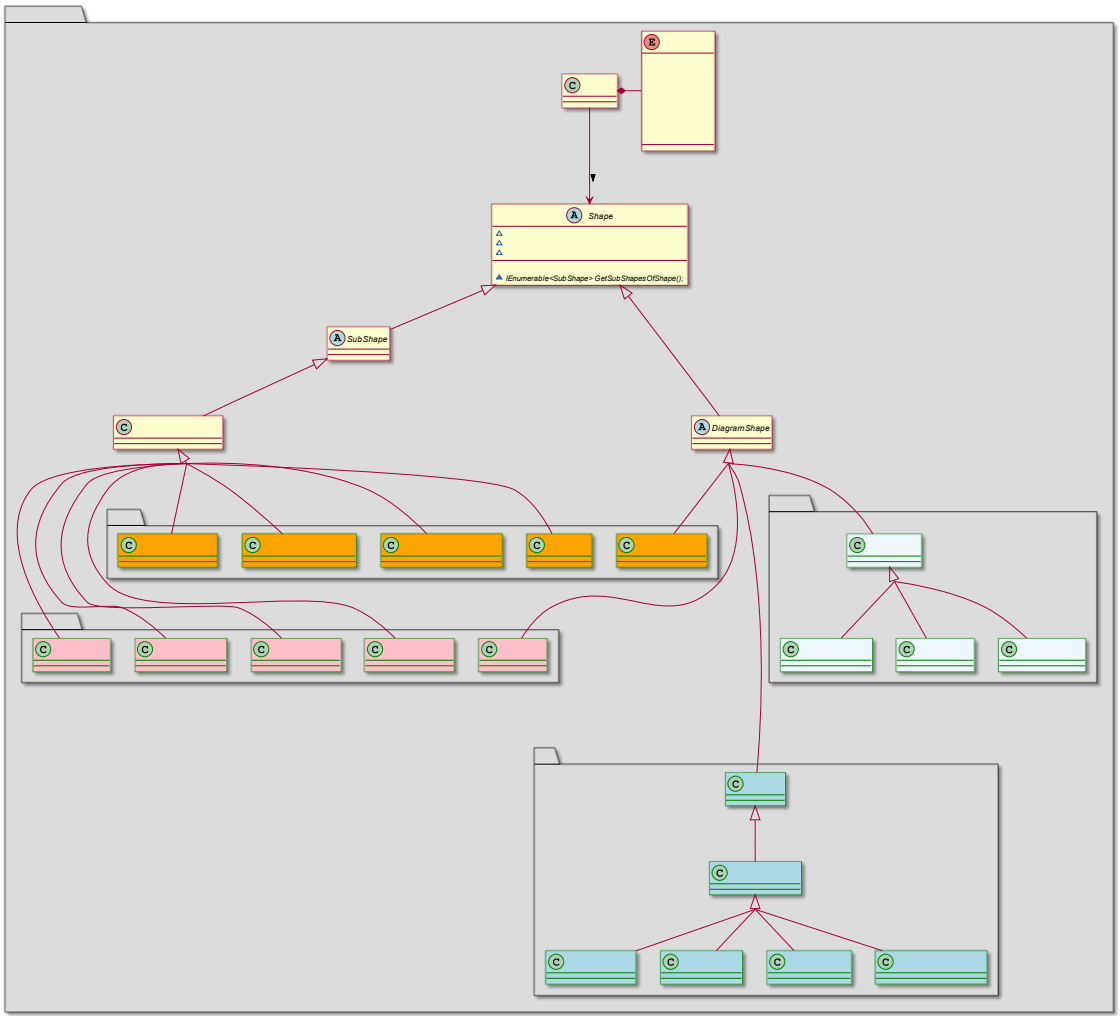


Abbildung 66: Auszug der Klassenhierarchie Shape und SubShape
(Quelle: Eigene Darstellung)

7.2 Anforderungen

Dieses Kapitel beschreibt die Anforderungen, die das Artefakt für die Quellcode-Analyse für die LCNC Entwicklungsumgebung Posity zu erfüllen hat, damit die Qualität einer Applikation oder einem Teil der Applikation mit Hilfe von Metriken und Regeln gemessen werden kann und entsprechende Code-Smells identifiziert werden können.

7.2.1 Strukturelle Anforderungen

- Die Implementation der Quellcode-Analyse, soll möglichst vom bestehenden Code entkoppelt sein, damit keine Nebenwirkungen zu befürchten sind.
- Die Metadaten zu den Regeln sollen in der Datenbank gespeichert sein.
- Neue Regeln sollen ausserhalb des Codes erfasst werden können und möglichst kein Neukompilieren des Posity-Frameworks bedingen.
- Die Architektur soll so gewählt werden, dass eine spätere Anwendung auf alle Diagrammtypen und alle Elemente mit möglichst wenig Aufwand umgesetzt werden kann.

7.2.2 Nicht Funktionale Anforderungen

- Die Performance der Posity IDE darf durch die Analyse nicht spürbar beeinträchtig werden.
- Die Analyse selbst soll möglichst schnell ein Ergebnis liefern, damit eine Analyse möglichst oft ausgeführt wird.
- Die Analyse soll Nebenläufig (parallel) durchgeführt werden.
- Die Bedienung soll möglichst intuitiv erfolgen.
- Das Resultat der Analyse soll dem Entwickler ermöglichen Problemstellen im Code einfach zu lokalisieren.
- Die Fehlertoleranz ist für das Artefakt noch hoch, wichtig ist jedoch, dass die Code-Analyse insofern zuverlässig ist, als dass sie Wiederholbar zum selben Resultat führt.

7.2.3 Funktionale Anforderungen

- Die Code-Analyse muss aus dem Posity Design Studio ausgeführt werden können.
- Die Code-Analyse muss verschiedene Betrachtungsräume (Scopes) anbieten (Einzelnes Diagramm, alle offenen Diagramme, ganze Applikation, etc.).

- Die Code-Analyse liefert ein Resultat mit den gefundenen Problemstellen.
- Die Problemstellen aus dem Resultat der Code-Analyse müssen leicht lokalisierbar sein (Analog zu textbasierten Programmiersprachen File::Codezeile).
- Regeln für die Code-Analyse müssen hinzugefügt, geändert oder gelöscht werden können.
- Die Konfiguration der Regeln erfolgt in der Datenbank.
- Es muss möglich sein Ausschlusskriterien für bestimmte Teile eines Programms festzulegen, damit diese nicht geprüft werden (Ignore-List).
- Es müssen Regeln von verschiedenen Kategorien, welche verschiedene Metriken verwenden, implementiert werden (Namensgebung, Kommentare, Struktur, Formatierung, etc.).
- Es müssen Regeln für alle vier Diagrammtypen GUI, Modul, Query und Tabellen implementiert werden.

7.2.4 Abgrenzung

Im Rahmen dieser Arbeit werden folgende Aspekte nur so weit bearbeitet, dass das Artefakt funktionsfähig ist. Folgende Teile werden deshalb bewusst minimal implementiert:

- Die Regelkonfiguration wird mit Hilfe vom Microsoft SQL Server Management Studio (Microsoft, 2022), direkt auf den Tabellen der Datenbank, vorgenommen.
- Die Regelkonfiguration wird vorerst im Kontext einer Applikation angewandt. Eine zentrale Verwaltung des Regelsatzes, der bei späteren Anpassungen auf die Applikationen synchronisiert wird, wird es vorerst nicht geben.
- Es wird nur eine Untermenge von Regeln im Artefakt umgesetzt, und zwar in dem Masse, dass die Beantwortung der Forschungsfrage(n) möglich ist.
- Es wird nur eine Untermenge von Diagramm-Elementen in die Quellcode-Analyse miteinbezogen, und zwar in dem Masse, dass die Beantwortung der Forschungsfrage(n) möglich ist.
- Das Resultat einer Code-Analyse ist flüchtig und muss entsprechend nicht persistiert werden.
- Die Visualisierung und Bedienung des Artefakts werden nur so weit vorangetrieben, dass dieses sinnvoll angewendet werden kann.
- Statistiken wie Vorher-Nachher-Vergleiche werden nicht erstellt.
- Technische Schulden werden nur in Form von Anzahl der gefunden Probleme ausgewiesen.

7.3 Erfolgskriterien

In diesem Kapitel sind die Kriterien gelistet, nach denen das Artefakt in der Validierungsphase beurteilt wird. Die Auswertung der Erfolgskriterien wird für die Beantwortung der Forschungsfragen verwendet.

Tabelle 13: Erfolgskriterien Artefakt Quellcode-Analyse

Kriterium Code-Analyse	**Beschreibung**
KCA1	Die Code-Analyse ist für eine beliebige Posity-Applikation in Posity möglich.
KCA2	Die Datenbank enthält die Regeln und die Konfiguration der Regeln, die Anwendung erfolgt im Code.
KCA3	Das Artefakt kann so erweitert werden, dass alle Shapes aller Diagramme analysiert werden können.
KCA4	Das Artefakt unterstützt die Posity-Entwickler bei ihrer täglichen Arbeit. Die Code-Analyse motiviert den Entwickler dazu, sauberen Code zu schreiben.
KCA5	Ein Posity-Entwickler kann das Artefakt ohne Anleitung bedienen.
KCA6	Folgende Qualitätsmerkmale einer mit der Posity IDE erstellen Applikation können durch die Code-Analyse gesteigert werden: - Analysierbarkeit - Anpassbarkeit - Richtigkeit
KCA7	Die Performance der Posity IDE darf durch die Analyse nicht spürbar beeinträchtig werden.
KCA8	Die Analyse muss zeitnahe Resultate liefern - 1 Diagramm < 1 Sekunde

	- 10 Diagramme < 5 Sekunden - 50 Diagramme < 10 Sekunden
KCA9	Durch das Resultat der Code-Analyse kann festgestellt werden, wo im Code welches Problem vorliegt, und es kann einfach zu der entsprechenden Stelle im Code navigiert werden.
KCA10	Eine Analyse führt bei wiederholter Ausführung immer zum gleichen Resultat, wenn sich am Code und den Regeln nichts geändert hat, die Ausschlusskriterien und der Betrachtungsraum (Scope) gleichgeblieben sind.
KCA11	Es können neue Regeln hinzugefügt, geändert und gelöscht werden. Möglichst ohne, dass die Posity IDE angepasst werden muss.
KCA12	Es können Ausschlusskriterien für Teile festgelegt werden so, dass diese nicht mehr analysiert werden.
KCA13	Für alle vier Diagrammtypen (GUI, Modul, Query und Tabellen) gibt es Regeln.

7.4 Spezifikation

Dieses Kapitel spezifiziert den Anforderungskatalog aus Kapitel 7.2 für die Implementation des Artefakts zur Quellcode-Analyse. Dabei werden die Elemente definiert und spezifiziert, welche es für eine Code-Analyse braucht. Zudem werden die technischen Möglichkeiten und Rahmenbedingungen erklärt, die durch die Posity IDE vorgegeben sind und bei der Umsetzung eine Rolle spielen.

Die Spezifikation unterscheidet zwischen Musskriterien, welche zwingend im Artefakt implementiert werden müssen und zwischen Wunschkriterien, die als aussichtsreich angesehen werden, jedoch nicht Teil des Artefakts werden. Die Unterscheidung wird jeweils gemäss Tabelle 14 hinterlegt.

Tabelle 14: Legende Symbol – Musskriterien vs. Wunschkriterien (Spezifikation Code-Analyse)

Kategorie	Symbol
Musskriterium	
Wunschkriterium	

7.4.1 Bestandteile der Quellcode-Analyse

Um eine Analyse am Quellcode durchführen und auswerten zu können, werden die folgenden Elemente definiert. Das Anforderungsprofil der in Tabelle 15 gelisteten Elemente wird in den nachfolgenden Kapiteln detailliert spezifiziert.

Tabelle 15: Definition Bestandteile der Quellcode-Analyse

Element	**Beschreibung**	
Posity Linter	Der Posity Linter führt die Analyse des Quellcodes durch. Dabei wendet er einen Satz an Regeln auf einen bestimmen Betrachtungsbereich (Scope) der Applikation an, erstellt für jede Regelverletzung ein Issue und zeigt die als Resultat die Liste der gefundenen Issues (Problemstelle) an.	
Metrik	Funktion, die eine Eigenschaft der Software in einem Wert abbildet. Der Wert, muss in diesem Kontext nicht zwingend ein Zahlenwert sein, sondern kann auch eine binäre Repräsentation (erfüllt/nicht erfüllt) haben oder in Textform ausgewiesen werden. Die Metriken werden von Regeln verwendet, um allfällige Verletzungen zu identifizieren.	
Regel	Eine Regel wird definiert und die Quellcode-Analyse überprüft, ob Verletzungen von Regeln vorliegen. Die Regel verwendet dabei Metriken, um allfällige Verletzungen zu detektieren.	

Mapping	Das Mapping bestimmt welche Regeln auf welche Komponenten/Shapes angewandt werden.	
Issue (Problemstelle)	Ein Issue entsteht, wenn bei der Analyse einer Applikation Regelverletzungen festgestellt werden. Das Issue enthält Informationen über die Regel, welche verletzt wurde, als auch in welchem Teil der Applikation die Verletzung vorliegt. Issues werden nur temporär gespeichert und nicht persistiert.	
Qualitätsprofil	Das Qualitätsprofil ist ein Satz an definierten Regeln, welche angewandt werden sollen. Das Qualitätsprofil kann pro Applikation individuell konfiguriert werden. In Zukunft soll eine Regelverwaltung, welche nicht Teil der Arbeit war, eingesetzt werden, damit ein zentraler Satz an Regeln als Basis dient und Applikationen die dort definierten Regeln einsetzen und für ihren individuellen Anwendungsfall konfigurierzyen können. Wenn Regeln ändern, sollen diese Änderungen in die existierenden Applikationen verteilt werden können.	
Ignore-List	Die Liste der zu ignorierenden Teile der Applikation. Die Liste kann dynamisch verändert werden. Ignorierte Element werden von der Analyse ausgeschlossen.	

7.4.2 Posity Linter

Unter dem Posity Linter kann der Programmteil verstanden werden, welcher die Quellcode-Analyse orchestriert. Er ist Teil von den Developer Tools, Kapitel 5, und bietet ein Benutzerinterface über welches die Analyse ausgeführt und das Resultat eingesehen werden kann. In den folgenden beiden Grafiken (Abbildung 67 und Abbildung 68) sind die Mockups für das Benutzerinterface abgebildet.

7.4.2.1 Benutzeroberfläche

Die Hauptansicht stellt die Analyse selbst dar (siehe Abbildung 67). Hier wählt der Benutzer den Betrachtungsbereich (Scope) der Analyse (siehe auch Kapitel 7.4.2.2) auf welchen die Analyse ausgeführt werden soll und kann eine Analyse starten. Nach Beendigung der Analyse wird das Resultat in einer Tabelle ausgegeben, wobei für jedes Issue

eine Zeile erstellt wird. Jedes Issue wird mit Detailinformation versehen, welche im Textfeld rechts neben der Tabelle angezeigt werden. Durch Selektieren einer Zeile in der Tabelle ändert sich der Inhalt des Textfeldes.

Eine zweite Ansicht bildet die Funktionalität rund um die Ignore-Liste ab. Um gewisse Elemente von der Analyse auszuschliessen, kann ein Issue selektiert und das betroffene Element der Ignore-Liste hinzugefügt werden. Dabei kann entweder nur das einzelne Element oder das gesamte Diagramm zur Ignore-Liste hinzugefügt werden. Die ignorierten Elemente werden in der Ansicht Ignore-Liste, wie sie in Abbildung 68 dargestellt ist, tabellarisch aufgelistet. Damit ein Element in einer späteren Analyse wieder mitberücksichtig wird, muss dieses aus der Ignore-Liste entfernt werden können.

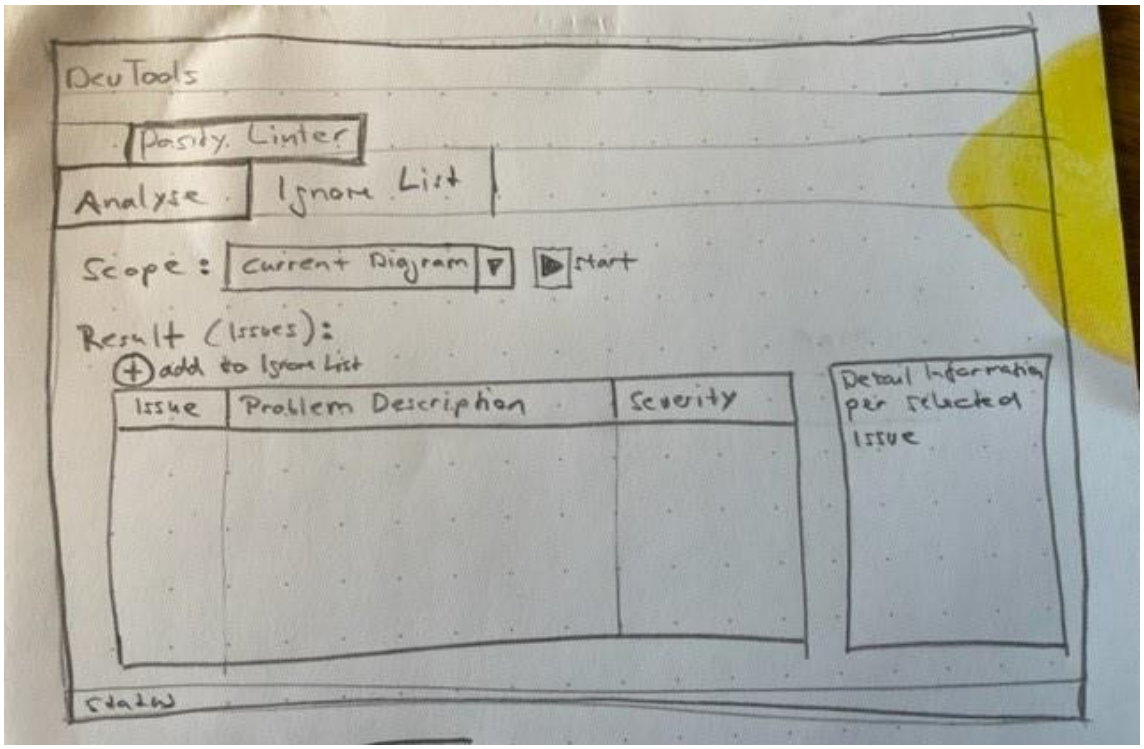


Abbildung 67: Mockup Benutzerinterface Posity Linter Analyse
(Quelle: Eigene Darstellung)

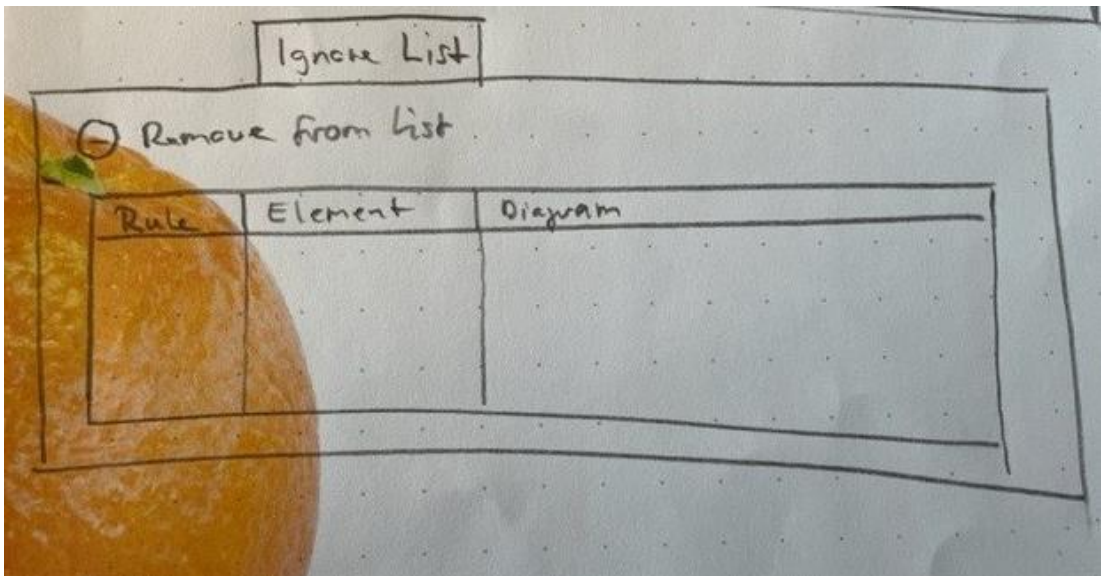


Abbildung 68: Mockup Benutzerinterface Posity Linter Ignore-Liste
(Quelle: Eigene Darstellung)

7.4.2.2 Betrachtungsbereich (Scope)

Damit eine Analyse möglichst effizient durchgeführt werden kann, soll es möglich sein, verschieden Betrachtungsbereiche festzulegen. Damit kann der Benutzer selektiv nur den Bereich, welcher für ihn von Interesse ist, analysieren und verhindert unnötig lange Analysen und unnötig lange Resultat-Listen von Issues, welche in anderen Bereichen der Anwendung gefunden wurden. Folgende Betrachtungsbereiche sollen gewählt werden können, um die Analyse auf diesem Bereich auszuführen.

Tabelle 16: Betrachtungsbereiche (Scope)

Betrachtungsbereich (Scope)	**Beschreibung**	
Application	Die gesamte Applikation mit allen Diagrammen wird analysiert. Dies kann nützlich sein, wenn ein Überblick über den Zustand der Applikation gebraucht wird.	
Package	Alle Diagramme, welche Teil vom ausgewählten Package sind. Dies kann nützlich sein, wenn ein Package in eine andere	

	Umgebung, beispielsweise die Test-Umgebung, verteilt wird. Mit der Analyse kann überprüft werden, ob die Qualitätsansprüche für ein Deployment genügend erfüllt sind.	
Current Diagram	Das Diagramm, das aktuell in der Edit-Area bearbeitet wird. Dieser Scope ist einer der wichtigsten, da die Analyse möglichst zeitnah zur Programmierung stattfinden sollte, um eventuelle Missstände rasch zu beseitigen.	
Selected Diagram(s)	Zu analysierende Diagramme können im Vorfeld selektiert werden. Dies kann nützlich sein, um einen auswählten Satz von Diagrammen einer Analyse zu unterziehen.	
All open Diagrams	Alle aktuell geöffneten Diagramme. Dieser Scope gehört ebenfalls zu den Musskriterien, weil auch hier, ähnlich wie beim Scope «Current Diagram» eine zur Programmierung zeitnahe Analyse ausgeführt werden kann. Beim Programmieren hat man meistens mehrere Diagramme offen, welche bearbeitet werden mit diesem Scope können also alle Diagramme, die potenziell geändert wurden, analysiert werden und Probleme schnell erkannt und beseitig werden.	
All Diagrams of Type	Die Analyse soll auf alle Diagramme eines bestimmten Diagrammtyps beziehen. Dies kann nützlich sein, wenn bspw. nur die Modul-Diagramme oder nur die GUI-Diagramme von Interesse sind.	
Pattern based	Mit dem Pattern-basierten Ansatz soll möglichst viel Flexibilität für die Definition des Scopes für eine Analyse erreicht werden. Ein Regex-Ausdruck soll dabei festlegen, welche Diagramme analysiert werden sollen. Dies ist nützlich, weil für ein Feature, beispielsweise die Anzeige einer Kundenliste, von mehreren Diagrammtypen eine Instanz erzeugt wird: • Prozess: Customer-List • Query: Customer-List	

	• Module: Customer-List • GUI: Customer-List Mit dem Pattern «Customer-List» könnte somit über alle Diagrammtypen hinweg nur diejenigen Diagramme analysiert werden, welche im Zusammenhang mit dem Feature Kundenliste stehen.	

7.4.3 Regeln und Metriken

Die Regel definiert den Anspruch an den Code. Bei der Analyse des Codes werden Metriken eingesetzt, um zu prüfen, ob der Code eine Regel erfüllt oder nicht. Liegt eine Regelverletzung vor, wird ein Issue (siehe Kapitel 7.4.5) erstellt. Für textbasierte Programmiersprachen wie beispielsweise die objektorientierten Programmiersprache Java definiert Sonar[14] zum heutigen Zeitpunkt 633 Clean Code-Regeln. Das zeigt auf wie vielseitig das Regelwerk ausgestaltet werden kann. Wie bereits in der Einleitung zum Kapitel 7 erwähnt, geht es in dieser Arbeit nicht darum einen vollständigen Satz an Regeln zu spezifizieren und diese zu implementieren. Vielmehr soll mit dem Artefakt anhand einiger weniger, vom Typ möglichst unterschiedlicher Regeln überprüft werden, ob die Adaption von Clean Code-Regeln auf LCNC-Programmiersprachen möglich ist und Problemzonen im Code aufgedeckt und eliminiert werden können. Die Kapitel 7.4.3.1 bis 7.4.3.6 listen und beschreiben die Regelkandidaten. Diese Sammlung enthält die Regeln, welche für diese Arbeit konzeptionell erstellt wurden. Eine Untermenge davon wurde im Artefakt implementiert. Die Regeln werden für die Unterscheidung mit einem entsprechenden Symbol gemäss Tabelle 17 versehen. Bei der Wahl der Regeln wurde darauf geachtet, dass sie aus möglichst unterschiedlichen Kategorien stammen, damit ein breites Spektrum von Regeltypen in der Arbeit berücksichtigt wird. Des Weiteren wird in Kapitel 7.4.3.7 wird beschriebenen welche Metadaten erfasst werden, um eine Regel formal zu beschreiben.

[14] Sonarcloud Rules, https://sonarcloud.io/organizations/default/rules?languages=java (Abgerufen: 01.05.2022)

Tabelle 17: Legende Symbol für Regeln - Implementiert vs. Konzept (Spezifikation Code-Analyse)

Kategorie	**Symbol**
Implementiert	●
Konzept	○

7.4.3.1 Regeln für Namenskonventionen

Die gemeinsame Nutzung von Namenskonventionen ist ein wichtiger Punkt, um eine effiziente Zusammenarbeit im Team zu ermöglichen.

Tabelle 18: Regeln für Namenskonventionen

Regel	**Beschreibung**	
Variablennamen sollten einer Namenskonvention entsprechen.	Mit dieser Regel kann überprüft werden, ob Variablennamen mit einem angegebenen regulären Ausdruck übereinstimmen.	●
Die Namen von Primärschlüssel sollten einer Namenskonvention entsprechen.	Mit dieser Regel kann überprüft werden, ob Primärschlüsselnamen mit einem angegebenen regulären Ausdruck übereinstimmen.	●
Die Namen von Fremdschlüssel sollten einer Namenskonvention entsprechen.	Mit dieser Regel kann überprüft werden, ob Fremdschlüsselnamen mit einem angegebenen regulären Ausdruck übereinstimmen.	●
Tabellennamen sollten einer Namenskonvention entsprechen.	Mit dieser Regel kann überprüft werden, ob Tabellennamen mit einem angegebenen regulären Ausdruck übereinstimmen.	●

Die Namen der Tabellen-Attribute sollten einer Namenskonvention entsprechen.	Mit dieser Regel kann überprüft werden, ob Tabellennamen mit einem angegebenen regulären Ausdruck übereinstimmen.	●
Die Namen von Tabellen-Indizes sollten einer Namenskonvention entsprechen.	Mit dieser Regel kann überprüft werden, ob Tabellen-Indizes mit einem angegebenen regulären Ausdruck übereinstimmen.	●
Die Namen der Integritätsregeln von Tabellen sollten einer Namenskonvention entsprechen.	Mit dieser Regel kann überprüft werden, ob Integritätsregeln von Tabellen mit einem angegebenen regulären Ausdruck übereinstimmen.	●
Diese Zeile steht stellvertretend für alle weitere Elemente aus dem Code, welche einer Namenskonvention folgen sollten.	• Konstanten • Query-Tabellen • Query-Attribute • GUI-Variablen • Status • …	○

7.4.3.2 Regeln für Kommentare

Die Beschreibung des Zwecks eines Elements ist wichtig für das Verständnis des Codes. Kommentare müssen immer auf dem neuesten Stand gehalten werden. Nichts irritiert mehr als ein Kommentar, der vom Code abweicht.

Tabelle 19: Regeln für Kommentare

Regel	**Beschreibung**	
Das Element (Shape) hat einen entsprechenden Kommentar.	Diese Regel kann universell für jedes Element im Diagramm, welches einen Kommentar hat, angewandt werden.	●
ToDo-Kommentare.	Durchsuche Kommentare, welche den Begriff ToDo enthalten, damit der Entwickler erkennen kann, wo diese sind und entsprechende Massnahmen ergreifen kann.	○

7.4.3.3 Regel für beschreibende Namen

In grafischen Entwicklungsumgebungen haben Komponenten oftmals einen Namen, um damit den Zweck des Elements zu beschreiben. Beispielsweise Bedingte-Anweisungsblöcke, Sequenzen oder Schleifen-Konstrukte. Für Verständlichkeit und Wartbarkeit ist es wichtig, diese Titel mit einer aussagekräftigen Beschreibung zu versehen und keinesfalls auf dem Default-Titel zu belassen. In Posity werden alle Namen, welche dem Default entsprechen mit spitzen Klammern befüllt (`<Sequence>, <Step 1>, <Foor Loop>,` etc.).

Tabelle 20: Regeln für beschreibende Namen

Regel	**Beschreibung**	
Die Titel von Strukturen wie switch case, sequence, try catch, posity script und lanes müssen festgelegt werden.	Die Titel von Strukturen wie switch case, sequence, try catch, posity script und lanes müssen so gesetzt werden, dass der Zweck des Blocks klar ist.	●

7.4.3.4 Spezifizierende Regeln

Zur Kategorie der Spezifikation, gehören Vorgaben, nach denen die Anwendung entwickelt werden soll. Dazu gehören auch erklärende Elemente, wie beispielsweise Tooltip-Informationen von Datenfeldern einer Tabelle oder eine definierte Sortierreihenfolge bei Abfragen.

Tabelle 21: Spezifizierende Regeln

Regel	**Beschreibung**	
Elemente auf der Benutzeroberfläche sollten mit einem Tooltip versehen werden.	Tooltips sollten bereitgestellt werden, damit der Benutzer der Benutzeroberfläche die Bedeutung eines Elements versteht. Zum Beispiel, welche Funktion eine Schaltfläche hat oder was der Zweck eines Feldes ist. Tooltips für GUI-Elemente, welchen eine Datenbindung mit Daten aus der Datenbank zugrunde liegt, werden in der Regel direkt im Datenmodell erfasst und gespeichert und automatisch auf den Benutzeroberflächen angezeigt, wo das entsprechende Datenfeld verwendet wird.	●
Ergebnistabellen von Datenbankabfragen sollten eine definierte Sortierreihenfolge haben.	Ergebnistabellen von Datenbankabfragen sollten eine definierte Sortierreihenfolge haben. Ansonsten ist die Reihenfolge zufällig und für den Benutzer nicht nachvollziehbar.	●

7.4.3.5 Regeln zu Komplexität

Die Komplexität eines bestimmten Codes hat damit zu tun, wie kompliziert und unhandlich es für einen Entwickler ist, ihn zu verstehen. Je grösser die Codebasis, umso schwieriger wird es für jemanden sich sich einen Überblick zu verschaffen. Dies wird noch problematischer, wenn der Code mehrere Pfade durch Fallunterscheidungen hat, mehrere Abhängigkeiten aufweist oder mit anderen Teilen der Codebasis gekoppelt ist. Komplexer Code ist fehleranfällig und ist auf jeden Fall schwer zu verstehen und daher schwer zu warten.

Tabelle 22: Regeln zu Komplexität

Regel	**Beschreibung**	
Verschachtelungstiefe	Die Komplexität der Verschachtelungstiefe ist ein Mass dafür, wie schwer der Kontrollfluss eines Modul-Events zu verstehen ist. Die Verschachtelungstiefe sollte einen bestimmten Schwellenwert nicht überschreiten. Default Schwellenwert: 5	●
Verwendung von Variablen und Parametern.	Nicht verwendete Variablen, Modul-Parameter, Event-Parameter und oder Query-Parameter sollten gelöscht werden, wenn sie nicht gebraucht werden.	○
Modul-Events sollten nicht leer sein.	Modul-Events, die keinen Code enthalten, überladen ein Projekt und sollten entfernt werden.	○
Kopplung und Kohäsion.	Kopplung und Kohäsion zwischen Subsystemen sind zwei weitere Metriken bei der Analyse der Architektur von Software. In der Regel ist es wünschenswert, dass die Subsysteme eine hohe Kohäsion innerhalb des Subsystems und eine geringe Kopplung mit anderen Subsystemen aufweisen. Hohe Kohäsion bedeutet zusammenhängende Belange (verantwortlich für zusammenhängende Belange) und geringe Kopplung bedeutet, dass Veränderungen nur eine lokale Auswirkung haben.	○
Zyklomatische Komplexität von Applikation, Modul, Modul-Event (McCabe-Metrik).	Die zyklomatische Komplexität ist eine von McCabe (1976) erstellte Metrik um Komplexität von Code zu messen. Sie misst die Anzahl der linear unabhängigen Pfade durch einen	○

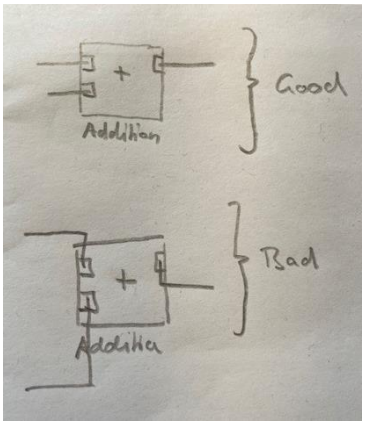
	Codeabschnitt. Ein Abschnitt ohne Entscheidungspunkte und Bedingungen wird als linear unabhängiger Pfad mit der geringstmöglichen Komplexität von 1 betrachtet. Ohne Kontrollflussanweisungen gibt es nur einen einzigen Weg durch den Code. Eine einfache Kontrollflussanweisung wie eine `IF`-Bedingung öffnet zwei verschiedene Pfade über `TRUE` und `FALSE` und erhöht damit die Komplexität des Quellcodeabschnitts auf 2. Die zyklomatische Komplexität des betrachteten Abschnitts, sollte einen bestimmten Schwellenwert nicht überschreiten. Default Schwellenwert für Modul-Events: 10	
Grösse von Modul-Event.	Ein Module-Event, der zu gross wird, tendiert dazu, viele Verantwortlichkeiten zu bündeln. Damit wird er unweigerlich schwerer zu verstehen und daher auch schwerer zu warten. Ab einem bestimmten Schwellenwert sollte eine Aufteilung in mehrere kleinere Modul-Events vorgenommen werden, die sich auf klar definierte Aufgaben konzentrieren. Maximal zulässige Elemente in einem Modul-Event: 100	◯
Duplizierter Code.	Diagramme sollten keine duplizierten Blöcke enthalten. Sobald es mindestens einen Block mit doppeltem Code in einem Modul-Diagramm gibt, wird ein Issue erzeugt.	◯

7.4.3.6 Regeln zu Formatierung und Ausrichtung (Quellcode Struktur)

Bei der Formatierung des Codes geht es um Kommunikation und Kommunikation ist für einen Entwickler ein wichtiges Gebot. Die Art der Kodierung und Lesbarkeit schafft Präzedenzfälle, die auch nach der Änderung des ursprünglichen Codes die Wartung und Erweiterbarkeit beeinflussen. Wie bei einem Text in natürlicher Sprache würde eine ungleichmässige Zeilenbreite, eine ungerade Schriftgrösse und ein unregelmässiger Zeilenumbruch das Lesen des Textes erschweren. Das Gleiche gilt für textbasierten Code und ähnliches gilt auch für grafisch basierten Code. Auch wenn es sich dabei nicht um Einrückungen oder Zeilenumbrüche handelt, so ist es genau so wichtig, die Formatierung einheitlich zu halten, damit Code transparent und leicht lesbar ist.

Tabelle 23: Regeln zu Formatierung und Ausrichtung (Quellcode Struktur)

Regel	**Beschreibung**	
Ausrichtung der Verbindungslinien beim Ein- und Ausgangs-Port.	Eingangs- und Ausgangsports sind quadratische Kästchen, bei welchen auf allen vier Seiten eine Verbindungslinie ein oder ausgehen kann. Wenn eine Verbindungslinie mit einem Eingangs- oder Ausgangsport verbunden wird, dann sollte die Ausrichtung dem zugehörigen Shape entsprechen und entweder horizontal oder vertikal liegen.	

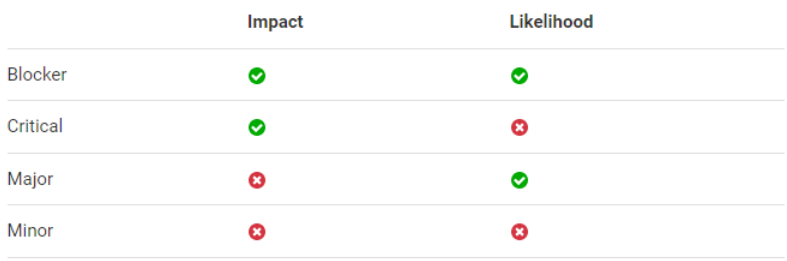
	 Abbildung 69: Ausrichtung von Verbindungslinien von Ports (Quelle: Eigene Darstellung)	
Ausrichtung von Elementen auf dem Diagramm ist einheitlich (vertikal oder horizontal).	Elemente auf den Diagrammen sollen einheitlich ausgerichtet sein, um unnötigen Lärm auf dem Diagramm zu vermeiden. Zusammenhängender Code sollte dicht beieinanderstehen und eine Gruppe bilden.	◯
Länge der Verbindungslinien im Datenmodell von Fremdschlüssel zu Primärschlüssel.	Datenmodelle für umfangreiche Applikationen können sehr komplex werden. Beispielsweise kann das Datenmodell für eine Lösung im Bereich Enterprise Resource Planning gut über 1’000 Tabelle umfassen. Damit das Modell durch die vielen Beziehungslinien zwischen Fremd- und Primärschlüssen nicht unleserlich wird, stellt Posity Tabellenreferenzen zur Verfügung, die als Stellvertreter einer Tabelle verwendet und mehrfach auf dem Datenmodell-Diagramm verwendet werden können. Damit muss die Beziehungslinie vom Fremdschlüssel nicht quer durch das Diagramm gezogen werden, sondern zur nahe platzierten Tabellenreferenz mit dem Primärschlüssel.	◯

7.4.3.7 Metadaten einer Regel

Die Regel besteht aus den in Tabelle 24 gelisteten Metadaten. Einige Metadaten wurden in Anlehnung an die Lösung von (SonarQube, 2022) gewählt.

Tabelle 24: Metadaten einer Regel

Metadaten-Element	**Beschreibung**	
Regel aktiv/inaktiv	Eine inaktive Regel wird in der Code-Analyse ignoriert.	
Rule Severity	Der Schweregrad, der einer Regel zugeordnet werden kann. **Blocker, Critical, Major, Minor, Info** Die Zuordnung kann wie in der Dokumentation von sonarQube (SonarQube, 2022) erläutert mit Hilfe der folgenden Fragestellung und Tabelle (siehe Abbildung 70) gemacht werden. Frage: Was ist das Schlimmste, dass passieren kann? Dann wird beurteilt, ob die Auswirkungen und die Wahrscheinlichkeit des Schlimmsten (siehe unten: Wie werden Schweregrad und Wahrscheinlichkeit bestimmt?) hoch oder niedrig sind, und die Antworten in eine Wahrheitstabelle eingetragen:	

	 Abbildung 70: Zuordnung der Severity für eine Regel (Quelle: (SonarQube, 2022)) **Auswirkung** (Impact): Könnte das Schlimmste dazu führen, dass die Anwendung abstürzt oder gespeicherte Daten beschädigt werden? Könnte die Ausnutzung des Schlimmsten zu einem erheblichen Schaden für einen selbst oder für den Benutzer führen? **Eintrittswahrscheinlichkeit** (Likelihood): Wie hoch ist die Wahrscheinlichkeit, dass das Schlimmste eintritt? Wie hoch ist die Wahrscheinlichkeit, dass ein Hacker in der Lage ist, die schlimmste Sache auszunutzen?	
Rule Type	Die Regeln sollen mit folgenden Typen kategorisiert werden können. **Bug, Vulnerability, Code Smell** Die Zuordnung kann wie in der Dokumentation von sonarQube (SonarQube, 2022) erläutert mit Hilfe der folgenden Fragestellung gemacht werden.	

	Handelt es sich bei der Regel um Code, der nachweislich falsch ist, oder ist es wahrscheinlicher, dass er falsch ist als nicht? Wenn die Antwort "ja" lautet, dann handelt es sich um eine Bug-Regel. Wenn nicht... **Bezieht sich die Regel auf Code, der von einem Hacker ausgenutzt werden könnte?** Wenn ja, dann handelt es sich um eine Vulnerability-Regel. Wenn nicht... **Handelt es sich bei der Regel weder um einen Bug noch um eine Sicherheitslücke?** Wenn ja, dann handelt es sich um eine Code Smell-Regel.	
Diagram-Type	**GUI, Modul, DataModel, Query, Multi** Der Diagrammtyp, für den eine Regel gilt. Wenn eine Regel auf mehrere Diagrammtypen angewandt werden kann, dann soll die Kategorie Multi verwendet werden.	
Tag	Mit Tags können Regeln klassifiziert werden. Durch die Klassifizierung können sie leichter aufgefunden werden.	
Status	Der Status gibt Aufschluss über den Reifegrad der Regel. **Beta, Deprecated, Ready** Beta: Die Regel wurde erst kürzlich implementiert, und es gab noch zu wenig Rückmeldungen von Benutzern, ob es zu falsch-positiven oder falsch-negativen Ergebnissen kommen kann. Deprecated: Die Regel sollte nicht mehr verwendet werden, da eine ähnliche, aber leistungsfähigere und genauere Regel existiert. Ready: Die Regel ist bereit, in der Produktion verwendet zu werden.	

Facet / Subfacet	Die Facet einer Regel bestimmt die Kategorie der Regel und damit wie und mit welcher Metrik die Regel überprüft wird. Mit der Subfacet kann gesteuert werden, welche Information aus dem Code für die Anwendung der Regel relevant ist. Beispiel Facet: `NamingConvention` Subfacet: `Systemname`	
Value	Mit dem Value besteht die Möglichkeit eine Regel bei Bedarf zu konfigurieren. So können beispielweise bei Namenskonventionen im Value-Attribut Regex-Ausdrücke hinterlegt werden mit welchen geprüft wird, ob eine Bezeichnung der Konvention gemäss Regex-Ausdruck entspricht oder nicht. Beispiel (Namenskonvention): `^(PK_)[a-zA-Z0-9]*$`	
Name	Der Name besteht aus einem Satz, welcher eine Aussage darüber macht, was im Sinne der Regel korrekt wäre. Beispiel: “Primarykey logical names should comply with a naming convention.”	
Kennung (Id)	Eine lesbare Bezeichnung zur Eindeutigen Identifizierung der Regel, die folgende Konvention aufweist: <Programmiersprache>:S<vierstellige fortlaufende Nummer> Beispiel: `posity:S0201` S steht dabei für Spezifikation	
Short Description	Die Kurzbeschreibung enthält Detailinformationen über die Regel. Dazu gehören beispielsweise eine Begründung, warum die Regel existiert und auch Handlungsempfehlungen, um die Regelverletzung zu vermeiden.	

Bad Practice & Good Practice	Je mindestens ein Beispiel von schlechter Praxis und von guter Praxis, damit anhand von Beispielen gezeigt werden kann, was das Problem ist und wie es besser gemacht werden kann.	

7.4.4 Posity-Shapes und Anwendung von Regeln (Mapping)

Textbasierten Programmiersprachen sind künstliche Sprachen, denen eine Grammatik zugrunde liegt. Die Sprache besteht aus einzelnen Symbolen, aus welchen sich im Allgemeinen eine unendliche Menge von Sätzen bilden lässt. Die Symbole sind reservierte Wörter (Schlüsselwörter) wie beispielsweise `class`, `boolean`, `static`, `int` etc., Konstanten, Sonderzeichen beispielsweise `+, -, *, =` etc., Bezeichner beispielsweise für Benennung von Variablen, Methoden etc. oder auch Kommentare, die besonders gekennzeichnet werden und vom Compiler ignoriert werden (Lee, 2017). Die Syntax einer Programmiersprache legt fest, was gültige Programme sind und mit dieser Syntax können auch statische Quellcode-Analysen arbeiten, den Code parsen, die Syntax interpretieren und eine Auswertung erstellen. Die einem Code zugrundeliegende Sprache muss einer statischen Quellcode-Analyse bekannt sein, damit die Regeln auf die entsprechenden Sprachelemente angewandt werden können. Im Fall von der modellbasierten LCNC-Programmiersprache Posity liegt der Quellcode nicht in textueller Form vor, sondern in Form von grafischen Elementen. Der Posity Linter muss die Syntax der Elemente kennen, um bei der Analyse die Regeln korrekt anzuwenden. Für die Zuordnung Regel zu Element, muss der Posity Linter wissen, welche Regeln für welche Elemente im Code angewandt werden. Diese Information kann nicht anhand einer Logik hergeleitet werden, sondern muss statisch festgehalten werden. Diese Mapping-Tabelle soll in der Datenbank abgelegt werden.

Die Elemente im Code von Posity sind als Shapes implementiert (siehe Kapitel 7.1.4) dabei wird in der Mapping-Tabelle der Typ der Shapes mit den Regeln verknüpft so, dass der Posity Linter in der Analyse die richtigen Regeln auf die dafür vorgesehenen Shapes anwendet. Tabelle 25 zeigt welche Informationen das Mapping enthält.

Tabelle 25: Mapping von Regeln zu Posity Shapes

Element Mapping	**Beschreibung**
Shape-Kennung (`ShapeIdentifier`)	Die Shape-Kennung entspricht dem Typnamen gemäss C#-Klassendefinition und setzt sich aus dem Namen des Namespaces der Klasse und dem Namen der Klasse selbst zusammen. Damit kann zur Laufzeit ein Objekt erkannt und das Mapping angewandt werden. Beispiel für die Klasse `GuiButtonShape` aus dem Package `IDENameSpace`: `IDENameSpace.GuiButtonShape`
Regel-Kennung (`LinterRuleIdentifier`)	Jede Regel erhält eine eindeutige Kennung, mit welche die Regel identifiziert werden kann. Siehe Kennung der Regel aus Tabelle 24. Damit kann zur Laufzeit die Regel erkannt und das Mapping angewandt werden. Beispiel für die Regel ‘Table system names should comply with a naming convention’ lautet die Kennung: `posity:S0003`

7.4.5 Issue

Während einer Analyse erstellt der Posity Linter jedes Mal ein Issue, wenn ein Teil des Codes eine Kodierungsregel verletzt. Der Satz von Regeln wird durch das zugehörige Qualitätsprofil (siehe 7.4.6) für jede Applikation definiert. Die Issues enthalten die Information über die verletzte Regel und den Ort im Code (Diagramm und Shape), wo die Verletzung der Regel vorliegt, sowie detaillierte Informationen über die Art der Verletzung und nach Möglichkeit konkrete Handlungsanweisungen. Issues werden nur temporär gespeichert und nicht persistiert.

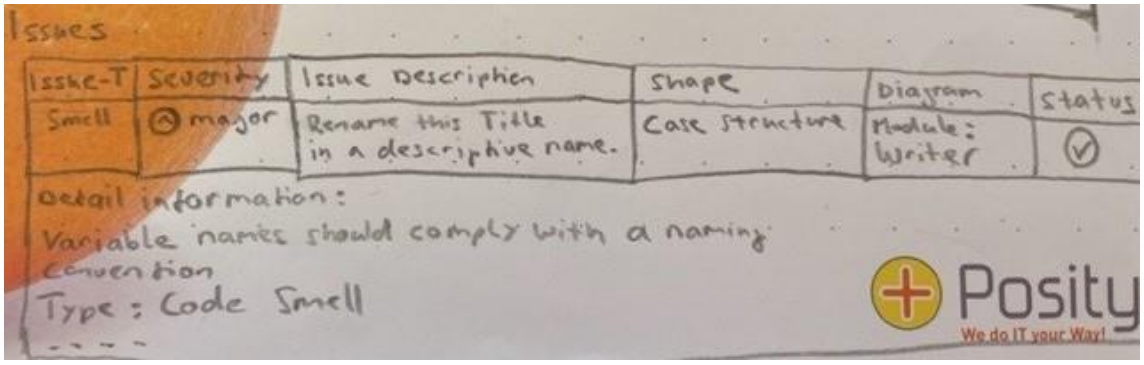


Abbildung 71: Metadaten von einem Issue (Quelle: Eigene Darstellung)

Die Issues werden vom Posity Linter visualisiert und es soll die Möglichkeit bestehen, direkt von der Auflistung der Issue zu der entsprechenden Stelle im Code zu springen, beispielsweis durch einen Doppelklick auf ein Issue.

7.4.6 Qualitätsprofil

Das Qualitätsprofil ist ein Satz an definierten Regeln, welche angewandt werden sollen. Das Qualitätsprofil kann pro Applikation individuell konfiguriert werden. In Zukunft soll eine Regelverwaltung, welche nicht Teil der Arbeit war, eingesetzt werden, damit ein zentraler Satz an Regeln als Basis dient und Applikationen die dort definierten Regeln einsetzen und für ihren individuellen Anwendungsfall konfigurieren können. Wenn Regeln ändern, sollen diese Änderungen in die existierenden Applikationen verteilt werden können.

7.4.7 Ignore-List

Wenn ein Teil der Applikation in der Quellcode-Analyse nicht berücksichtig werden soll, kann dieser von der Analyse ausgeschlossen werden, indem dieser Teil zur Liste der zu ignorierenden Teile hinzugefügt wird. Es kann entweder ein ganzes Diagramm oder ein einzelnes Shape ausgeschlossen werden. Ein ganzes Diagramm kann beispielsweise dann ausgeschlossen werden, wenn es sich um alten, bestehenden Code handelt, der nicht mehr an die geltenden Clean Code-Regeln angepasst werden soll, weil ein entsprechendes Refactoring als zu aufwändig angesehen wird. Weiter könnte auch das Szenario eintreten, dass in einer Anwendung nur bestimmte einzelne Shapes von der Analyse ausgeschlossen werden sollen, da Ausnahmen bekannter weise die Regel bestätigen. Somit kann eine Ausnahme in den folgenden Konstellationen definiert werden.

Tabelle 26: Ignore-List Konstellationen und Reichweite

Reichweite	**Beschreibung**	
Diagramm + Regel	Das gesamte Diagramm wird im Zusammenhang mit dieser Regel von der Analyse ausgeschlossen.	
Shape + Regel	Das einzelne Shape wird im Zusammenhang mit dieser Regel von der Analyse ausgeschlossen.	
Diagramm	Das gesamte Diagramm wird von der Analyse ausgeschlossen. Gilt für alle Regeln.	
Shape	Das Shape wird von der Analyse ausgeschlossen. Gilt für alle Regeln.	

7.4.7.1 Diagramme und Shapes von der Analyse ausschliessen

Ein Diagramm oder ein Shape (Element) kann direkt in der Issue-Liste des Posity Linters für zukünftige Ausführungen von der Analyse ausgeschlossen werden. Über diesen Weg ist die zugehörige Regel über den Kontext des Issues definiert und das Element kann zusammen mit der entsprechenden Regel zur Ignore-Liste hinzugefügt werden.

Zudem soll es möglich sein, ein Diagramm oder ein Shape direkt aus dem Editor der Posity IDE von der Analyse auszuschliessen. Dies kann beispielsweise mit dem Kontextmenü oder über das Applikationsmenü ausgelöst werden. Wird das Diagramm oder das Element über diesen Weg zur Ignore-Liste hinzugefügt, gibt es keinen Kontext zu einer bestimmten Regel und das zu ignorierende Element wird ganz von der Analyse ausgeschlossen.

7.4.7.2 Diagramme und Shapes wieder von der Ignore-List entfernen

Falls ein Element aus der Ignore-Liste wieder in zukünftigen Analysen berücksichtigt werden soll, kann das Element wieder von der Ignore-Liste entfernt werden. Das Entfernen ist unabhängig von der jeweiligen Konstellation gemäss Tabelle 26.

7.5 Implementation

Dieses Kapitel beschreibt, wie die spezifizierten Anforderungen gemäss Kapitel 7.4 implementiert wurden. Dabei ist zu bemerken, dass das Artefakt in das Posity-Framework integriert werden musste, damit für die Quellcode-Analyse die Diagrammdaten vollständig verfügbar sind. Die Posity IDE, auch Posity Design Studio (PDS) genannt, ist Teil vom C#-Assembly `PosityApp`. Das Posity-Framework besteht aus mehreren Teilen. Die Abbildung 72 zeigt den grob umrissenen Aufbau und wie sich die einzelnen Teile zum Posity-Framework zusammenfügen. Die Quellcode-Analyse (Posity Linter) wird bei den Utilities integriert, da es sich um ein Werkzeug handelt, dass als Dienstprogramm benutzt werden kann.

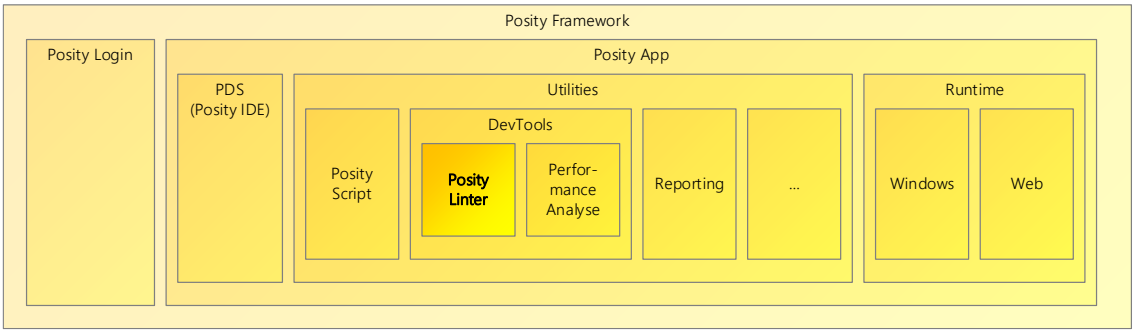


Abbildung 72: Bestandteile Posity-Framework (Quelle: Eigene Darstellung)

Im Weiteren wird auf die bestehende Architektur des Posity-Frameworks nur so weit eingegangen, wie es zum Verständnis der Implementation der Quellcode-Analyse nötig ist. Hauptfokus liegt auf dem neu erstellen Artefakt.

Die `Posity App` ist eine C#-Applikation und auf Microsoft .NET Framework 4.8 kompiliert. Alle grafischen Bestandteile sind mit dem GUI-Toolkit Windows Forms[15] implementiert. Für die Implementation des Artefakts wurden die gleichen Technologien verwendet.

[15] Microsoft .Net Windows Forms GUI-Toolkit, https://docs.microsoft.com/en-us/dotnet/desktop/winforms/overview/?view=netdesktop-6.0 , (Abgerufen: 12.05.2022)

7.5.1 Ablauf, Struktur und Klassenhierarchie

Bei den Artefakten, die im Rahmen dieser Arbeit entwickelt worden sind, handelt es sich um Dienstprogramme, welche als Unterstützung bei der Entwicklung von Posity-Applikationen dienen sollen. Die Klassen sind in der Projektstruktur des Projekts `PosityApp` unter `Utilities` im Ordner `DevTools` implementiert. Die Klassen für die Quellcode-Analyse befinden sich im Unterordner `Linter`. Die Klasse `DevTools` dient als grafische Zentrale für alle Entwicklungswerkzeuge, deshalb ist auch das Artefakt der Performance-Analyse Teil der Developer Tools (Klasse `DevTools`). Angaben zum Artefakt der Developer Tools (DevTools) sind im Kapitel 5 verortet.

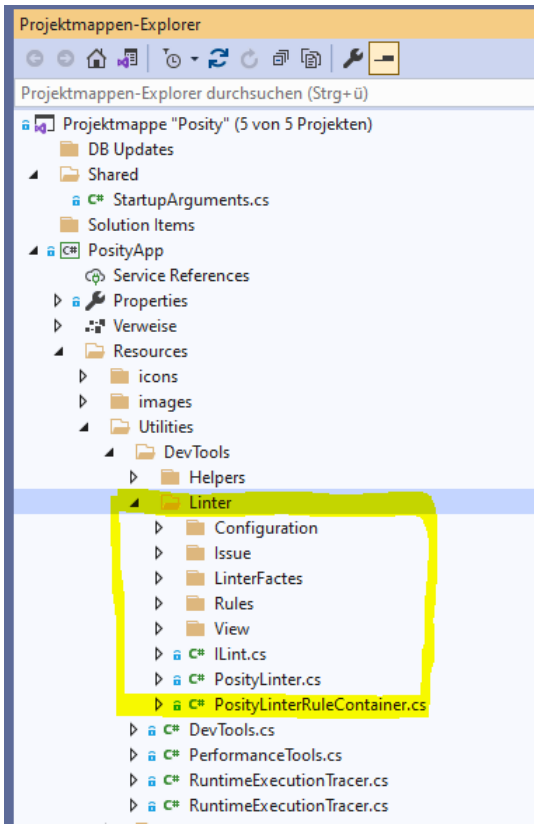


Abbildung 73: Sub-Projektstruktur und Verortung im Hauptprojekt PosityApp (Quelle: Eigene Darstellung)

Das Klassendiagramm in Abbildung 74 zeigt als Gesamtübersicht alle Klassen des Artefakts und die Abhängigkeiten zur Posity IDE. Der Posity Linter ist hierbei das zentrale Element, welches die Quellcode-Analyse durchführt. Für eine Quellcode-Analyse wird folgender Ablauf pro Diagramm durchgeführt.

1. Für jede aktive Regel werden alle potenziellen Kandidaten in dem Diagramm gesucht. Kandidaten sind Shapes, welche von dieser Regelkategorie geprüft werden können. Beispielsweise kann nicht jedes Shape auf Namenskonventionen geprüft werden, weil nicht jedes Shape einen Namen hat. Das Erkennen der Kandidaten erfolgt über den Typ der Shape-Instanzen. Damit eine Regelkategorie auf einen Kandidaten angewendet werden kann, muss der Kandidat bestimmte Fähigkeiten aufweisen, welche er durch das Implementieren eines entsprechenden Interfaces (`LinterFacets`) erlangt. Damit kann durch die Typenprüfung sichergestellt werden, dass ein Shape die Methoden bereitstellt, welche die Regeln dieser Kategorie schliesslich brauchen, um zu bestimmen, ob das Shape die Regel erfüllt oder verletzt.
2. Für jeden potenziellen Kandidaten wird nun für jede Regel aus der Kategorie geprüft, ob die Regel für den Kandidaten relevant ist. Eine Regel ist für einen Kandidaten nur dann relevant, wenn in der Mapping-Tabelle (siehe Kapitel 7.5.3.7) ein entsprechender Eintrag vorhanden ist und der Kandidat oder die Regel nicht explizit zur Ignore-Liste (siehe Kapitel 7.5.6) hinzugefügt worden sind.
3. Verletzt der Kandidat die Regel, wird ein Issue (sehe Kapitel 7.5.5) erzeugt mit detaillierten Informationen über das Problem und der Stelle im Code, an der das Problem auftritt.
4. Schritte 1-3 werden wiederholt, bis alle Regeln auf alle Kandidaten angewandt worden sind.
5. Die bei der Analyse erzeugten Issues werden als Resultat (`PosityLinterResult`) zurückgegeben und auf der Benutzeroberfläche angezeigt.

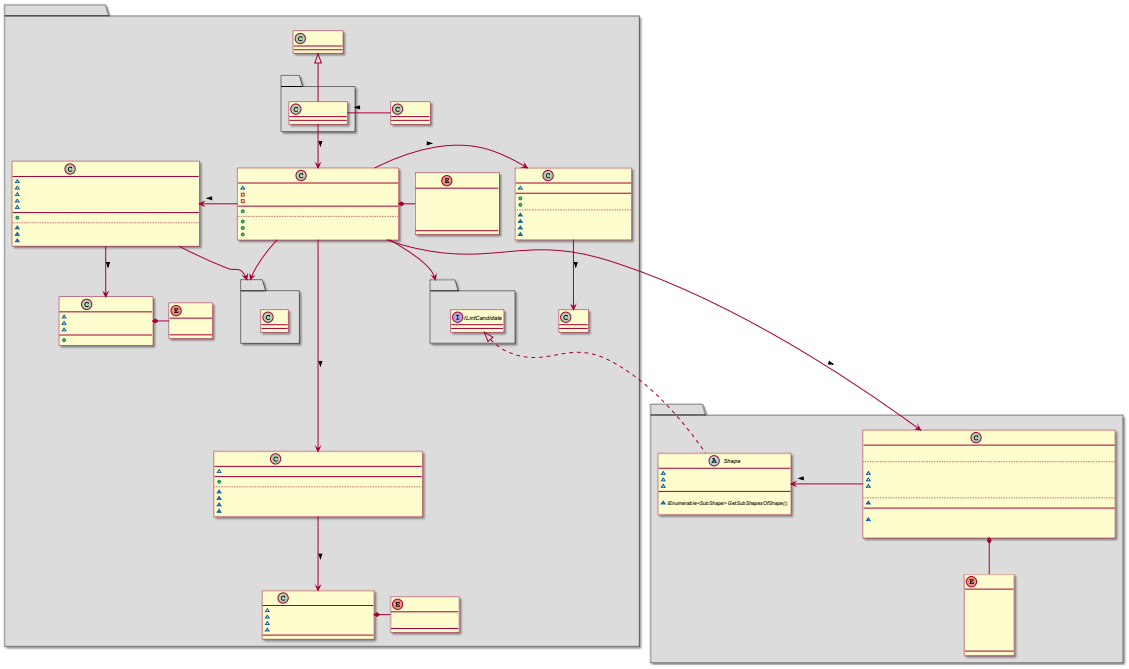


Abbildung 74: Klassendiagramm Posity Linter und Abhängigkeiten
(Quelle: Eigene Darstellung)

Der besseren Lesbarkeit halber wird das Klassendiagramm aus Abbildung 74 auf zwei einzelne Abbildungen aufgeteilt. Abbildung Abbildung 75 zeigt noch einmal nur die Klassen aus dem `PosityUtilities.DevTools.Linter` und Abbildung 76 nur die Klassen aus dem `Namespace` IDENamespace.

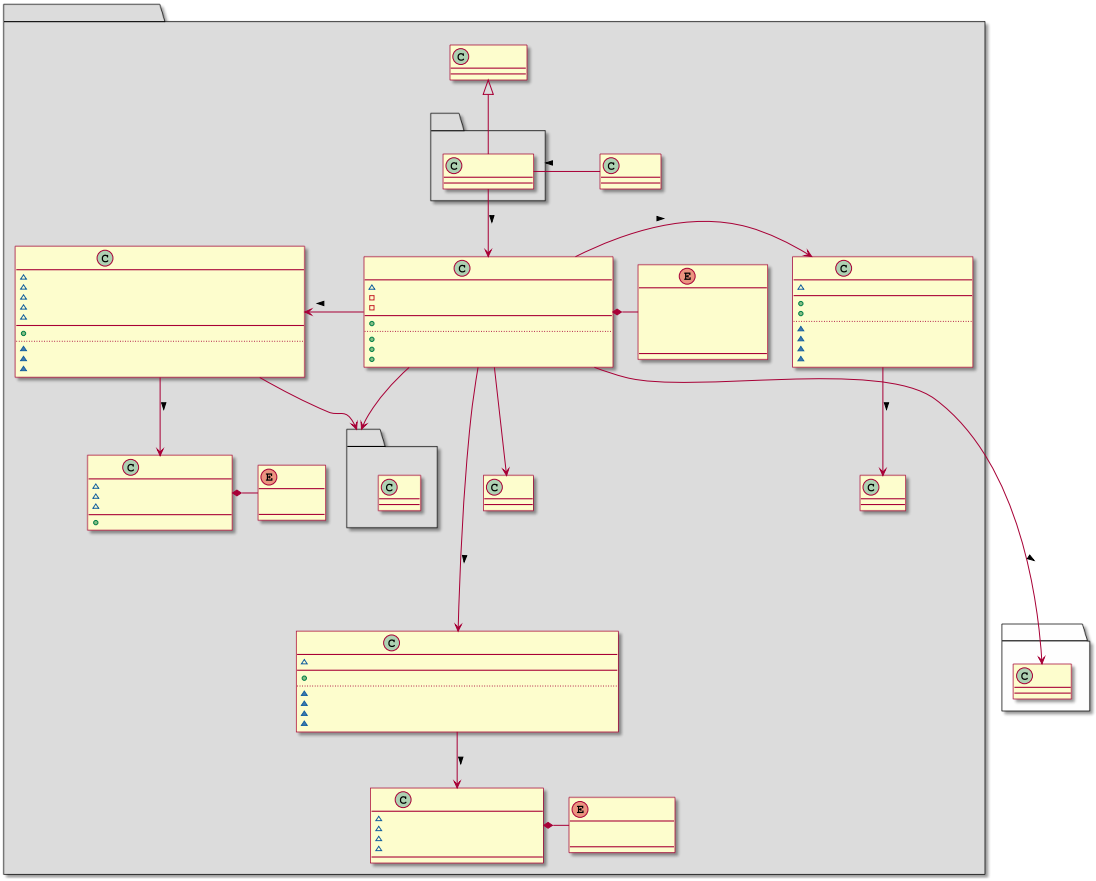


Abbildung 75: Klassendiagramm Posity Linter (nur Posity Linter)

(Quelle: Eigene Darstellung)

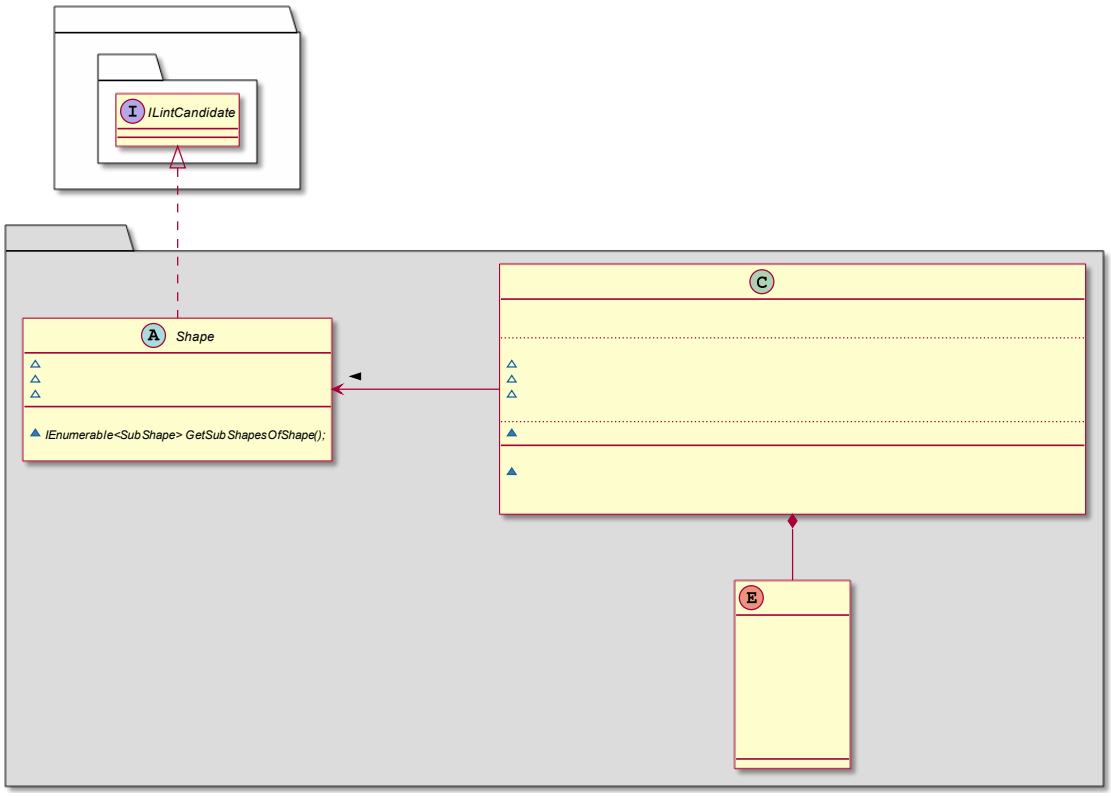


Abbildung 76: Klassendiagramm Posity Linter - nur IDENamespace
(Quelle: Eigene Darstellung)

7.5.2 Posity Linter

Wie in Kapitel 7.4.2 der Spezifikation beschrieben orchestriert der Posity Linter die Quellcode-Analyse. Der Posity Linter wird von der Posity Lint Page instanziiert. Die Posity Lint Page ist die View der Quellcode-Analyse und der Posity Linter enthält die Businesslogik für die Quellcode-Analyse. Der Posity Linter wendet die Regeln auf den Quellcode an und erzeugt daraus ein Resultat.

7.5.2.1 Posity Linter als Teil der Developer Tools

Für die grafische Darstellung des Posity Linters ist die Klasse `PosityLinterPage` zuständig. Sie erweitert die Klasse `DevToolPage` und enthält sämtliche für die Darstellung notwendigen GUI-Elemente und verarbeitet das Event-Handling, damit Benutzereingaben an die Logik `PosityLinter` weitergeleitet werden. Im Klassendiagrammausschnitt der Abbildung 77 sind die Abhängigkeiten zwischen `PosityLinter` und `DevTools` ersichtlich sowie die Hauptzuständigkeiten der jeweiligen Klassen. Die Klasse `DevTools` erzeugt die `PosityLinterPage` und fügt sie zu seiner Liste von `DevToolPages` hinzu um sie später

auf der Benutzeroberfläche als Tab in einem Tabpanel anzuzeigen. Die `PosityLinterPage` erzeugt wiederum die Instanz des Posity Linters.

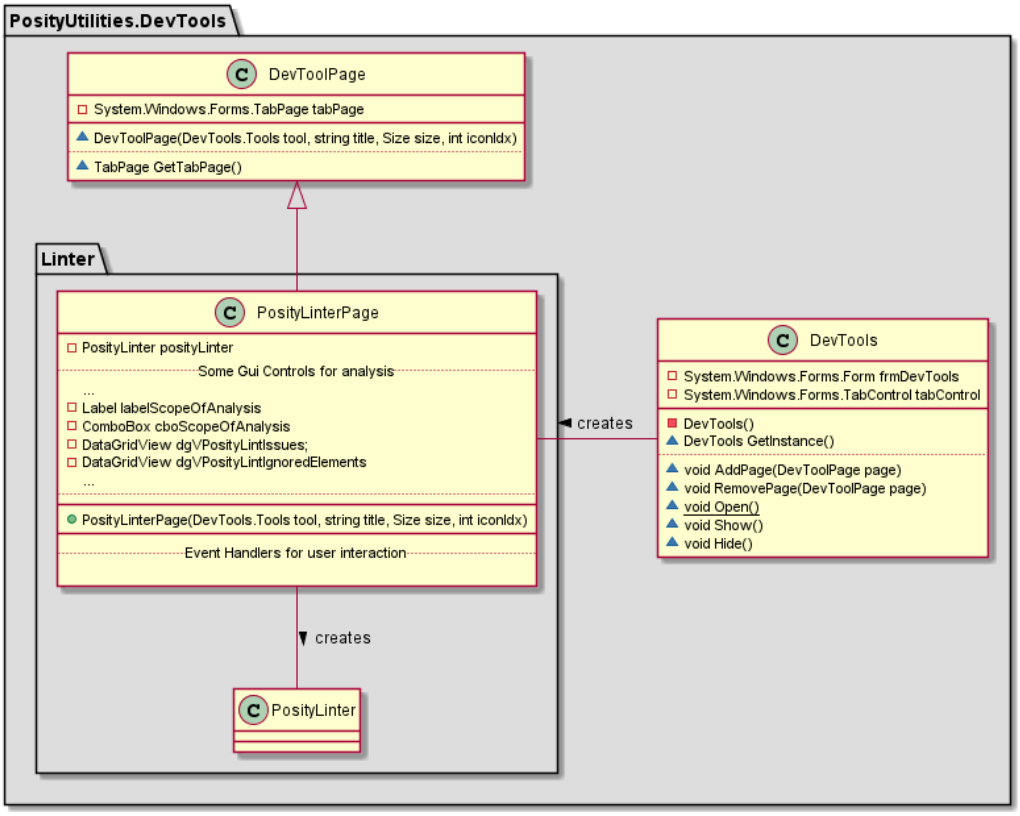


Abbildung 77: Klassendiagramm Posity Linter und DevTools
(Quelle: Eigene Darstellung)

7.5.2.2 Benutzeroberfläche

Der Posity Linter, als Teil der Developer Tools kann über zwei Wege aus dem PDS gestartet werden. Abbildung 78 zeigt die beiden Varianten. Variante 1, durch Öffnen der Entwicklertools (siehe Kapitel 5.2) über das Applikationsmenü Werkzeuge oder Variante 2, direkt über die Statusbar beim Stinktier-Symbol.

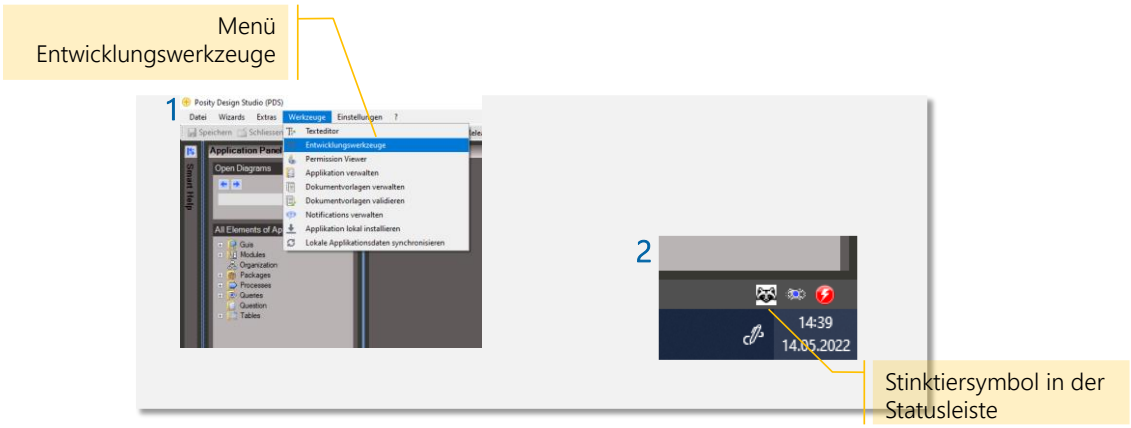


Abbildung 78: Posity Linter starten (Quelle: Eigene Darstellung)

Über die Benutzeroberfläche des Artefaktes Posity Linter, kann das Analysieren, Darstellen von Ergebnissen und das Verwalten von Ausschlusskriterien gesteuert werden. Die Benutzeroberfläche setzt sich grob aus zwei Teilen zusammen. Einmal ist dies die Ansicht für die Quellcode-Analyse selbst und als Zweites ist dies die Ansicht für die Ignore-Liste, in der verwaltet werden kann, welche Teile der Applikation von der Analyse ausgeschlossen werden sollen. In der Ansicht (Tab) für die Quellcode-Analyse selbst, werden die verfügbaren Betrachtungsbereiche (Scopes) in einer `DropDown-Liste` angezeigt. Eine Analyse kann über das Start-Symbol gestartet werden. Sobald die Analyse abgeschlossen ist, wird das Resultat in einem `DataGridView` (Tabelle) dargestellt. Die Abbildung 79 zeigt die Benutzeroberfläche vom Artefakt Quellcode-Analyse mit der Ansicht der Analyse selbst (Tab «Analyse») und beschreibt die Funktionalitäten der Steuerelemente.

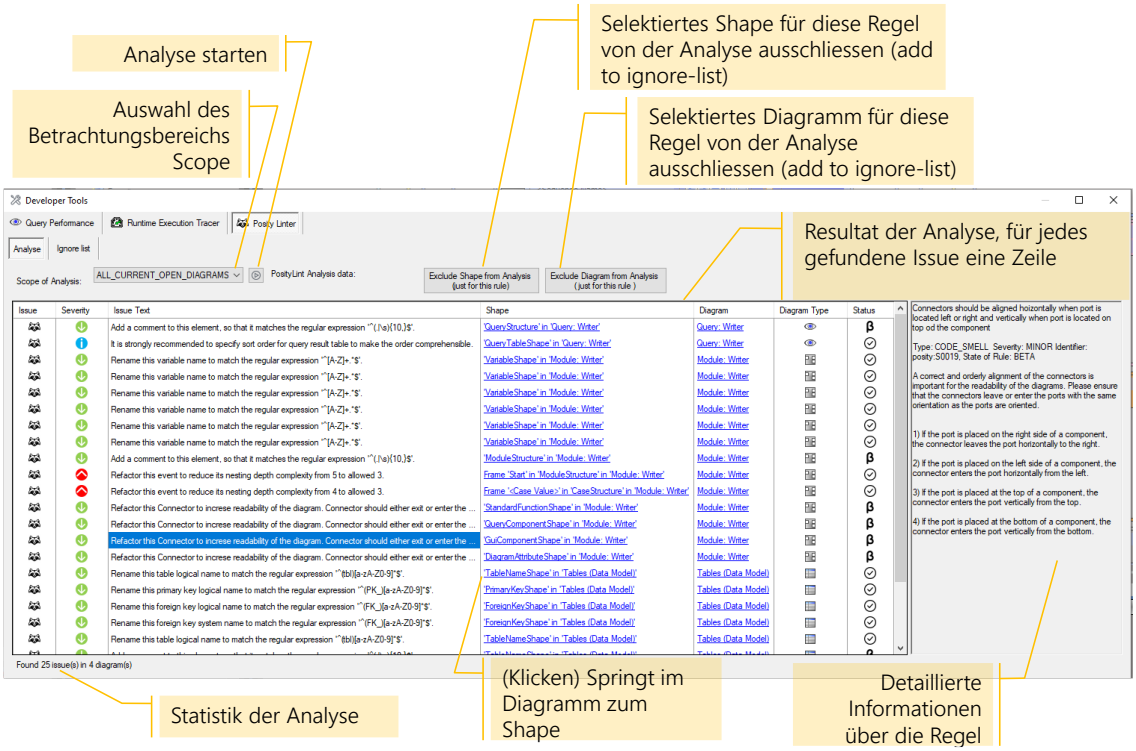


Abbildung 79: Benutzeroberfläche Posity Linter – Tab «Analyse»

(Quelle: Eigene Darstellung)

In der Ansicht (Tab «Ignore list») für die Ignore-Liste, werden alle Elemente in einem `DataGridView` (Tabelle) aufgelistet, welche von der Analyse ignoriert werden sollen. Ein Element kann jeder Zeit wieder von der Ignore-List entfernt werden und wieder in die Analyse integriert werden. Ein Element in der Ignore-Liste ist entweder ein Shape oder ein ganzes Diagramm und steht in Bezug zu einer Regel. Der Ausschluss gilt also immer für ein Shape oder Diagramm in Kombination zu einer Regel. Die Abbildung 79 zeigt die Benutzeroberfläche vom Artefakt Quellcode-Analyse mit der Ansicht der Ignore-Liste (Tab «Ignore list») und beschreibt die Funktionalitäten der Steuerelemente.

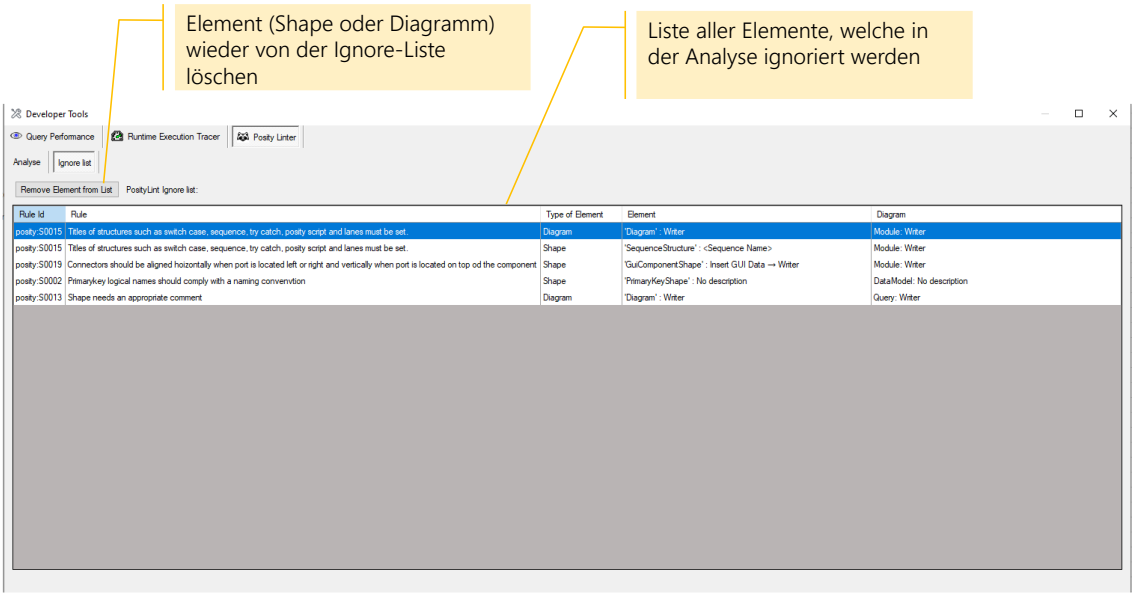


Abbildung 80: Benutzeroberfläche Posity Linter - Tab Ignore list

(Quelle: Eigene Darstellung)

7.5.3 Regeln und Metriken

Für die Implementierung der Regeln war eine wichtige Anforderung an das Design, dass sie so wenig wie möglich mit dem bestehenden Quellcode der Posity IDE gekoppelt ist und weil es viele Regeln geben wird, es einfach sein soll weitere Regeln zu implementieren. Die Anforderung an die Kopplung wird so gelöst, dass die Logik der Regeln in eigenen Klassen abgebildet und nicht als Teil von den Shapes implementiert wird. Für die Erweiterbarkeit bezüglich neuer Regeln, wurden sogenannte `Facets` eingeführt. Eine `Facet` kann als Regel-Kategorie verstanden werden, für jede `Facet` gibt es eine Regel-Implementation, welche die Klasse `Rule` erweitert. `Rule` ist die Superklasse für alle Regeln. Jede Regel-Kategorie hat ihre eigene Implementation der Methode `Rule#ExecuteRule(ILintCandidate candidate)` und bildet darin die Überprüfung der Regel ab. Zudem kann eine `Facet` zusätzliche `SubFacets` definieren, welche es erlauben unterschiedliche Regeln derselben Kategorie zu evaluieren. Die einzelne Regel wird also nicht etwa als Klasse in Posity erstellt, sondern wird mit den benötigten Daten parametrisiert und ist in der Datenbank abgelegt. Jede Regel gehört einer Regel-Kategorie (`Facet`) an und zur Laufzeit wird für jede erfasste Regel eine Instanz von der Regel-Kategorie erstellt. Tabelle 27 zeigt, wie Facet, Regel-Implementation und eine einzelne Regel zusammenhängen.

Tabelle 27: Facets, Implementationen der Regeln und Beispiele

Facet und {SubFacets}	**Implementation der Regel-Facet (Klasse)**	**Beispiele von Regeln**
NAMING_CONVENTION {SYSTEM, LOGICAL}	NamingConventionRule	"Primarykey system names should comply with a naming convention." "Table logical names should comply with a naming convention." "Variable names should comply with a naming convention."
COMMENT_CONVENTION {MAIN}	CommentConventionRule	"Shape needs an appropriate comment."
DESCRIPTIVE_TITLE_CONVENTION {TITLE}	DescriptiveNameConventionRule	"Titles of structures such as switch case, sequence, try catch, posity script and lanes must be set."
COMPLEXITY {NESTING_DEPTH}	ComplexityRule	"Complexity of nesting depth is a measure of how hard the control flow of an event is to understand. Events with high Complexity will be difficult to maintain."

SPECIFICATION {SORT_ORDER, TOOLTIP}	SpecificationRule	“Elements on the user interface should have a tooltip behind them.” “Result tables of database queries should have a defined sort order.”
ALIGNMENT {CONNECTOR_ALIGNMENT}	AlignmentRule	“Connectors should be aligned horizontally when port is located left or right and vertically when port is located on top od the component.”

Der Kandidat, auf welchen die Regel angewandt wird, wird der Methode `Rule#ExecuteRule(ILintCandidate candidate)` als Parameter übergeben. Die Kandidaten sind Shapes oder Sub-Shapes und liefern der Regel über eine festgelegte Schnittstelle (siehe Abbildung 82), die Informationen, welche die Regel braucht, um die Prüfung durchführen zu können. Die `RuleFactory` erzeugt `Rule`-Instanzen abhängig von der `Facet` der Regel. Das Klassendiagramm nach Abbildung 81 zeigt die erwähnten Klassen und ihre Abhängigkeiten, sowie eine Übersicht der implementierten Regel-Typen auf.

Ein Beispiel: Für eine Regel mit der `Facet Naming_Convention` wird eine Instanz vom Typ `NamingConventionRule` erzeugt. Für die Überprüfung erhält die Regel einen Kandidaten vom Typ `ILintNamingConventionCandidate` als Parameter. Das Interface `ILintNamingConventionCandidate` definiert Methoden, um Namen des Kandidaten abzufragen, die dann mit den Namenskonventionen, welche die Regel vorgibt, geprüft werden können.

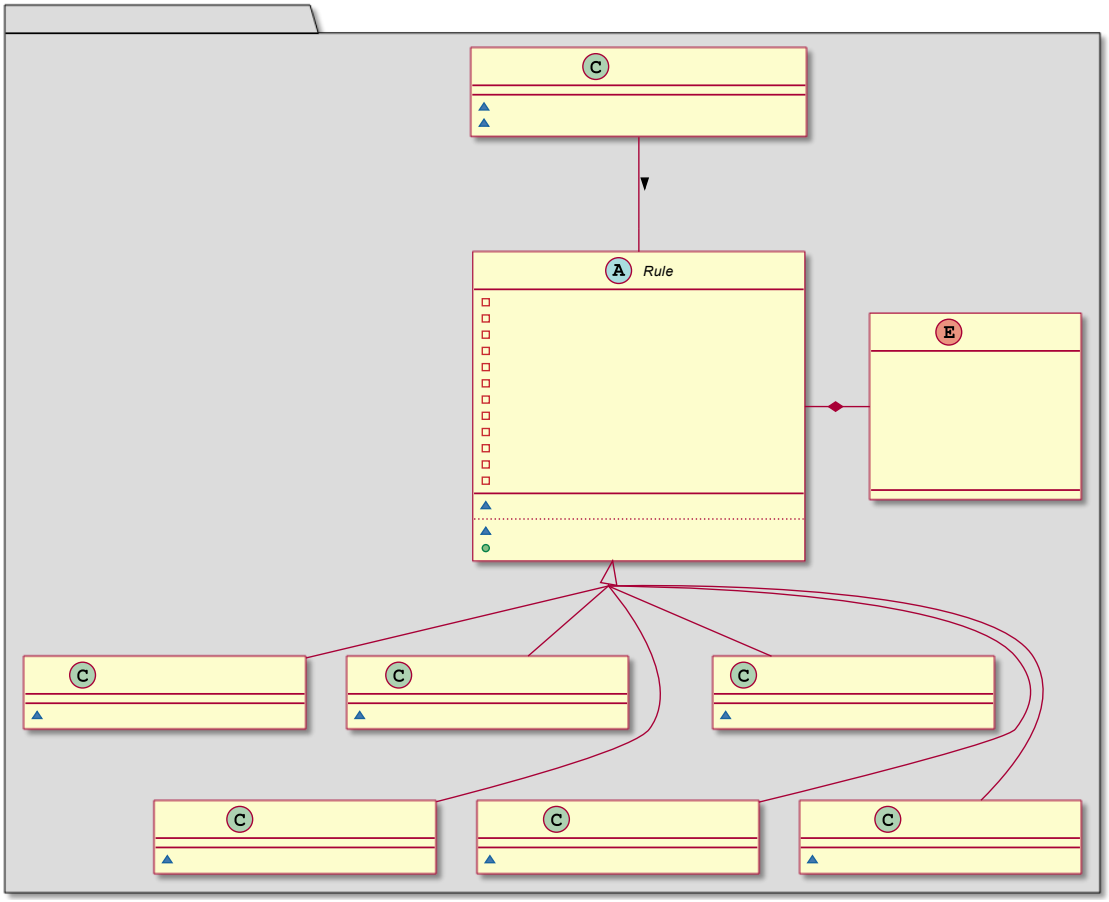


Abbildung 81: Klassendiagramm Regeln (Quelle: Eigene Darstellung)

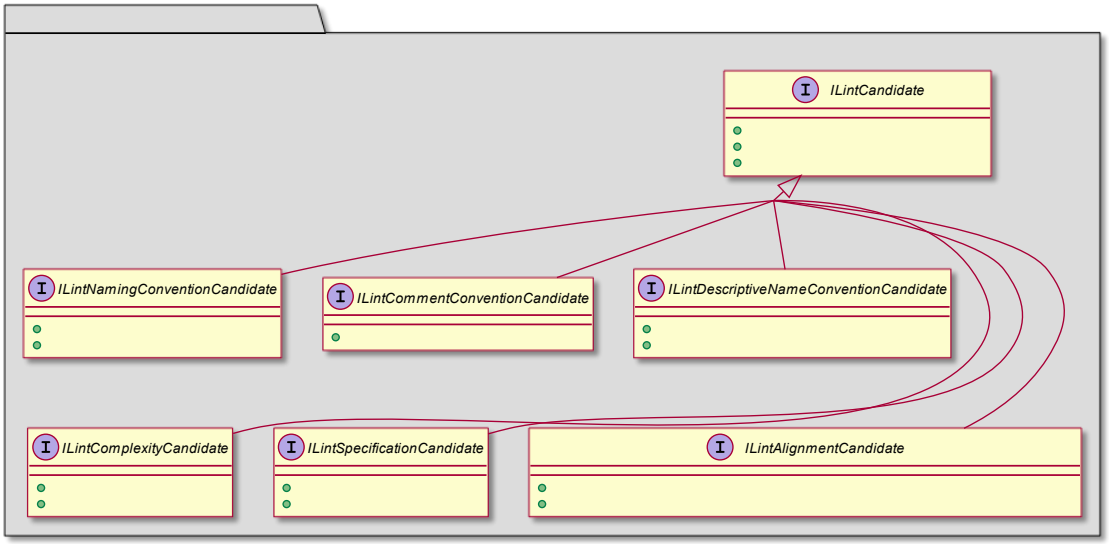


Abbildung 82: Klassendiagramm ILintCandidates (Quelle: Eigene Darstellung)

Die Kandidaten-Schnittstellen können von beliebigen Diagramm-Elementen (Shapes) implementiert werden. In Abbildung 83 ist zu sehen, welche Shapes mit der aktuellen Implementation, welche Kandidaten-Schnittstellen implementieren.

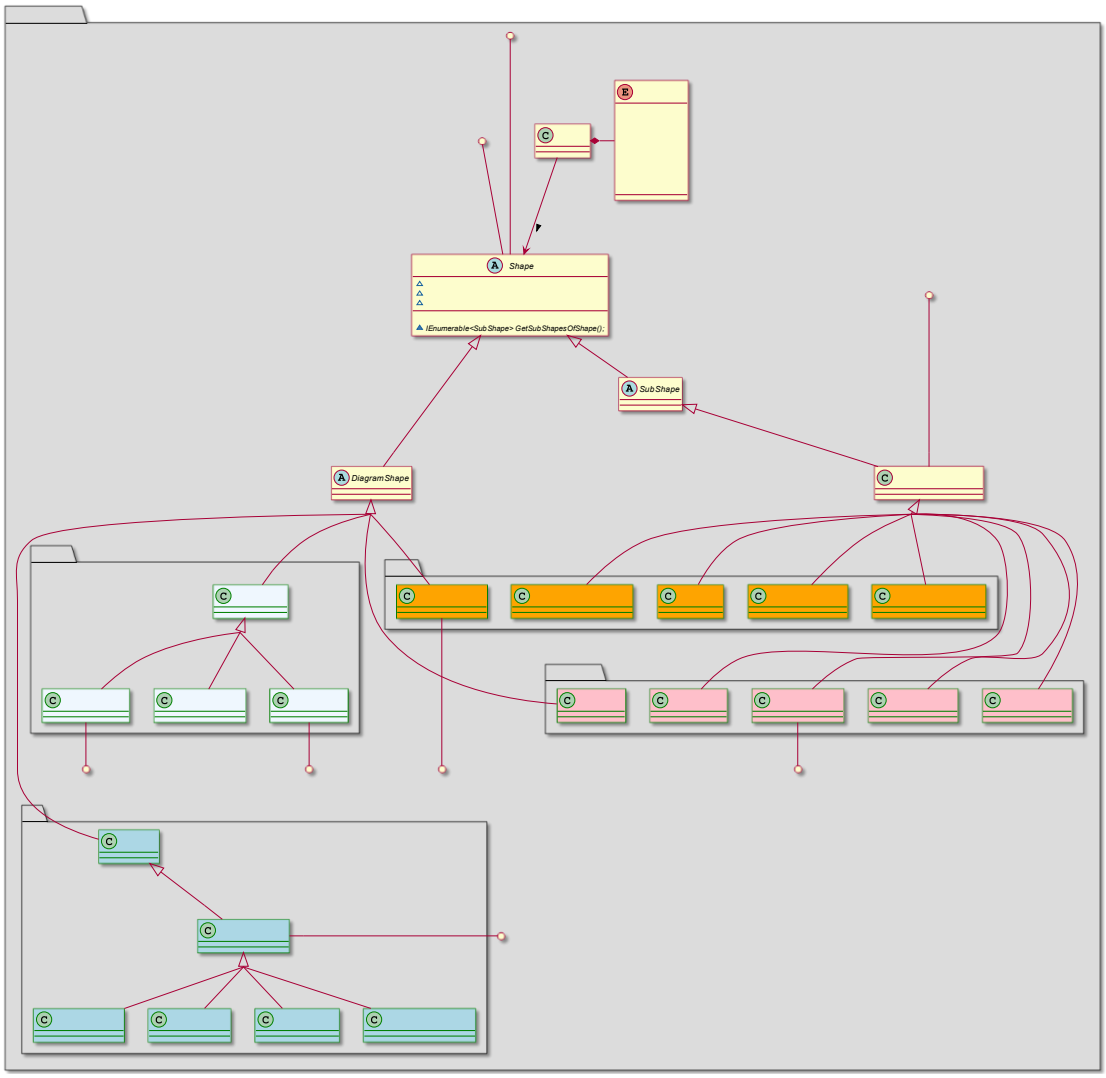


Abbildung 83: Posity-Shapes und ILintCandidates Interfaces
(Quelle: Eigene Darstellung)

Die Regeln und ihre Metadaten werden in der Datenbank der Applikation abgelegt. Die Tabelle `LinterRule` enthält alle Regeln und deren Eigenschaften. Die Tabellen `LinterRuleSeverity`, `LinterRuleStatus` und `LinterRuleType` enthalten Metadateninformationen zu Schweregrad, Status und Typ der Regel. Die Metadaten und deren Bedeutung sind im Kapitel 7.4.3.7 der Spezifikation erläutert. Abbildung 84 zeigt das Datenmodell der Regel und ihren Metadaten.

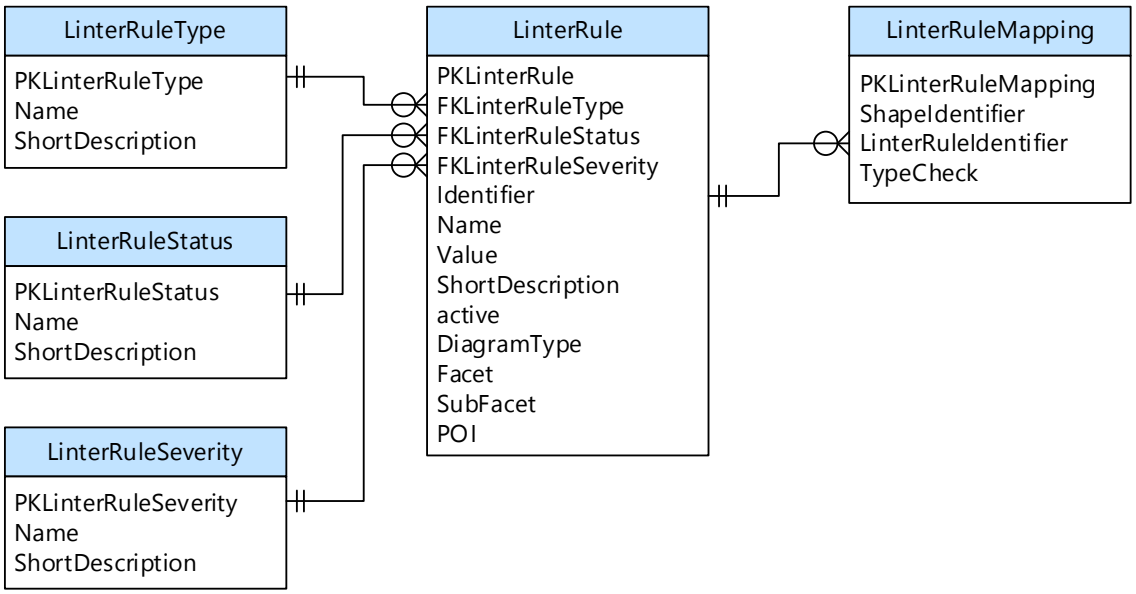


Abbildung 84: Datenmodell Regel und Metadaten (Quelle: Eigene Darstellung)

In den Kapiteln 7.5.3.1 bis 7.5.3.6 wird die Implementation der einzelnen Regeltypen beschrieben. Für jeden Regeltyp wird angegeben, welche Facet er repräsentiert, welche Sub-Facets es bis jetzt gibt, welche Klasse die Regel implementiert und welches Kandidaten-Interface von der Regel verlangt wird. Zusätzlich werden pro Regeltyp alle im Rahmen von dieser Arbeit implementierten Regeln in einer Tabelle angegeben. In Abbildung 81 und Abbildung 82 sind die entsprechenden Klassen, Interfaces und Enumerationen abgebildet.

7.5.3.1 Regeln für Namenskonventionen

Regeln, die der Kategorie Namenskonvention angehören, prüfen, ob ein Name (Text) mit einem angegebenen regulären Ausdruck übereinstimmt. Durch die `SubFacet` kann unterschieden werden, ob der Systemname oder der logische Name (sprachabhängige Bezeichnung) überprüft werden soll. Für die Überprüfung wird kontrolliert, ob der Namen mit dem zugehörigen Regex-Ausdruck übereinstimmt.

Facet	NAMING_CONVENTION
SubFacet	NameType {SYSTEM, LOGICAL}
Rule-Class	NamingConventionRule
Kandidaten-Interface	ILintNamingConventionCandidate

Implementation Der Name wird vom Kandidaten zurückgegeben und mit dem in der Regel definierten Regex-Ausdruck auf Übereinstimmung geprüft.

Tabelle 28 zeigt alle implementierten Regeln der Kategorie Namenskonventionen.

Tabelle 28: Implementierte Regeln für Namenskonventionen

Identifier	Name	Value	ShortDescription	Diagram-Type	Facet	Sub-Facet
posity:S0001	Primarykey system names should comply with a naming convention	^(PK_)[a-zA-Z0-9]*$	Sharing some naming conventions is a key point to make it possible for a team to efficiently collaborate. This rule allows to check that field names match a provided regular expression.	DATA_MODEL	NAMING_CONVENTION	SYSTEM
posity:S0002	Primarykey logical names should comply with a naming convention	^(PK_)[a-zA-Z0-9]*$	Sharing some naming conventions is a key point to make it possible for a team to efficiently collaborate. This rule allows to check that field names match a provided regular expression.	DATA_MODEL	NAMING_CONVENTION	LOGICAL
posity:S0003	Table system names should comply with a naming convention	.*	Sharing some naming conventions is a key point to make it possible for a team to efficiently collaborate. This rule allows to check that table names match a provided regular expression.	DATA_MODEL	NAMING_CONVENTION	SYSTEM
posity:S0004	Table logical names should comply with a naming convention	^(tbl)[a-zA-Z0-9]*$	Sharing some naming conventions is a key point to make it possible for a team to efficiently collaborate. This rule allows to check that table names match a provided regular expression.	DATA_MODEL	NAMING_CONVENTION	SYSTEM
posity:S0005	Foreignkey system names should comply with a naming convention	^(FK_)[a-zA-Z0-9]*$	Sharing some naming conventions is a key point to make it possible for a team to efficiently collaborate. This rule allows to check that field names match a provided regular expression.	DATA_MODEL	NAMING_CONVENTION	SYSTEM
posity:S0006	Foreignkey logical names should comply with a naming	^(FK_)[a-zA-Z0-9]*$	Sharing some naming conventions is a key point to make it possible for a team to efficiently collaborate. This rule allows to check that field names match a	DATA_MODEL	NAMING_CONVENTION	SYSTEM

	convention		provided regular expression.			
posity:S0007	Attribute system names should comply with a naming convention	^.*	Sharing some naming conventions is a key point to make it possible for a team to efficiently collaborate. This rule allows to check that field names match a provided regular expression.	DATA_MODEL	NAMING_CONVENTION	SYSTEM
posity:S0008	Attribute logical names should comply with a naming convention	^.*	Sharing some naming conventions is a key point to make it possible for a team to efficiently collaborate. This rule allows to check that field names match a provided regular expression.	DATA_MODEL	NAMING_CONVENTION	SYSTEM
posity:S0009	Tableindex system names should comply with a naming convention	^.*	Sharing some naming conventions is a key point to make it possible for a team to efficiently collaborate. This rule allows to check that field names match a provided regular expression.	DATA_MODEL	NAMING_CONVENTION	SYSTEM
posity:S0010	Tableindex logical names should comply with a naming convention	^.*	Sharing some naming conventions is a key point to make it possible for a team to efficiently collaborate. This rule allows to check that field names match a provided regular expression.	DATA_MODEL	NAMING_CONVENTION	SYSTEM
posity:S0011	Table integrity rule system names should comply with a naming convention	^.*	Sharing some naming conventions is a key point to make it possible for a team to efficiently collaborate. This rule allows to check that field names match a provided regular expression.	DATA_MODEL	NAMING_CONVENTION	SYSTEM
posity:S0012	Table integrity rule logical names should comply with a naming convention	^.*	Sharing some naming conventions is a key point to make it possible for a team to efficiently collaborate. This rule allows to check that field names match a provided regular expression.	DATA_MODEL	NAMING_CONVENTION	SYSTEM
posity:S0014	Variable names should comply with a naming convention	^[A-Z]+.*$	Sharing some naming conventions is a key point to make it possible for a team to efficiently collaborate. This rule allows to check that variable names match a provided regular expression.	MODULE	NAMING_CONVENTION	SYSTEM

7.5.3.2 Regeln für Kommentare

Regeln, die der Kategorie Kommentare angehören, prüfen, ob ein Kommentar (Text) mit einem angegebenen regulären Ausdruck übereinstimmt. Durch die `SubFacet` kann unterschieden werden, welcher Kommentar überprüft werden soll im Falle es für einen Kandidaten mehr als nur den einen Kommentar gibt. Für die Überprüfung wird kontrolliert, ob der Kommentar mit dem zugehörigen Regex-Ausdruck übereinstimmt.

Facet	`COMMENT_CONVENTION`
SubFacet	`CommentType {Main}`
Rule-Class	`CommentConventionRule`
Kandidaten-Interface	`ILintCommentConventionCandidate`
Implementation	Ähnlich wie bei der Kategorie Namenskonventionen, bietet sich für die Kommentare eine Überprüfung mit Hilfe von Regex an. Der Kommentar wird vom Kandidaten zurückgegeben und mit dem, in der Regel definierten Regex-Ausdruck auf Übereinstimmung geprüft. Damit kann beispielweise überprüft werden, ob der Kommentar eine bestimme Mindestlänge hat.

Tabelle 29 zeigt alle implementierten Regeln für die Kommentare

Tabelle 29: Implementierte Regeln für Kommentare

Identifier	Name	Value	ShortDescription	Diagram-Type	Facet	Sub-Facet
posity:S0013	Shape needs an appropriate comment	^(.|\s){10,}$	Describing the prupose of an element is important to understand the code. Make shure, that comments are always up to date.	MULTIPLE	COMMENT_CONVENTION	MAIN

7.5.3.3 Regel für beschreibende Namen

Regeln, die der Kategorie beschreibende Namen angehören, prüfen, ob ein Name (Text) beispielsweise ein Titel von seinem Standardwert abweicht und einem bestimmten Mus-

ter (Regex-Ausdruck) folgt. Dadurch kann sichergestellt werden, dass der Name zumindest vom Programmierer angepasst worden ist und bestimme Anforderungen, beispielsweise an die Länge, erfüllt. Durch die `SubFacet` kann unterschieden werden, welcher Name überprüft werden soll im Falle es für einen Kandidaten mehr als nur den einen Namen gibt.

Facet	`DESCRIPTIVE_TITLE_CONVENTION`
SubFacet	`CommentType {TITLE}`
Rule-Class	`DescriptiveNameConventionRule`
Kandidaten-Interface	`ILintDescriptiveNameConventionCandidate`
Implementation	Prüfen, ob ein Name (Text) beispielsweise ein Titel von seinem Standardwert abweicht und einem bestimmten Muster (Regex-Ausdruck) folgt. Dadurch kann sichergestellt werden, dass der Name zumindest vom Programmierer angepasst worden ist und bestimme Anforderungen beispielsweise an die Länge erfüllt.

Tabelle 30: Implementierte Regeln für beschreibende NamenTabelle 30 zeigt alle implementierten Regeln für beschreibende Namen

Tabelle 30: Implementierte Regeln für beschreibende Namen

Identifier	Name	Value	ShortDescription	Diagram-Type	Facet	Sub-Facet
posity:S0015	Titles of structures such as switch case, sequence, try catch, posity script and lanes must be set.	^[^<]*[^>]$	Titles of structures such as switch case, sequence, try catch, posity script and lanes must be set so that the intention of the block is clear.	MULTIPLE	DESCRIPTIVE_TITLE_CONVENTION	TITLE

7.5.3.4 Spezifizierende Regeln

Regeln, die der Kategorie spezifizierende Regeln angehören, prüfen, ob gängige Spezifikationen festgelegt worden sind. Durch die `SubFacet` kann unterschieden werden, welche Art von Spezifikation überprüft werden soll. Für diese Arbeit wurde eine Regel implementiert, welche prüft, ob bei einer Datenbankabfrage (Query Diagramm) eine Sortierreihenfolge festgelegt worden ist und eine weitere Regel, die prüft, ob ein GUI-Element einen Tooltip besitzt.

Facet	`SPECIFICATION`
SubFacet	`SpecificationType {SORT_ORDER, TOOLTIP}`
Rule-Class	`SpecificationRuleRule`
Kandidaten-Interface	`ILintSpecificationCandidate`
Implementation	Für die verschiedenen Varianten von `SepcificationType` werden verschiedene Überprüfungen ausgelöst. Für `SORT_ORDER` wird beim Kandidaten nachgefragt, ob er eine Sortierung festgelegt hat und für `TOOLTIP` wird beim Kandidaten abgefragt, ob ein Tooltip spezifiziert worden ist.

Tabelle 31 zeigt alle implementierten Regeln für die Spezifikation.

Tabelle 31: Implementierte Regeln für Spezifikation

Identifier	Name	Value	ShortDescription	Diagram-Type	Facet	Sub-Facet
posity:S0017	Result tables of database queries should have a defined sort order	-	Result tables of database queries should have a defined sort order. Otherwise the order is random and not comprehensible for the user.	QUERY	SPECIFICATION	SORT_ORDER
posity:S0018	Elements on the user interface should have a tooltip behind them	-	Tooltips should be provided so that the user of the user interface understands the meaning of an element. For example, what function a button has or what the purpose of a field is. Please note that tooltips for fields that originate from the database can	GUI	SPECIFICATION	TOOLTIP

```
also be stored directly
in the data model and
thus automatically appear
on the user interfaces
where the field is used.
```

7.5.3.5 Regeln zu Komplexität

Regeln, die der Kategorie Komplexität angehören, prüfen strukturelle Eigenschaften eines Diagramms. Durch die `SubFacet` kann unterschieden werden, welche Art von Komplexität überprüft werden soll. Für diese Arbeit wurde eine Regel implementiert, welche prüft, wie tief verschachtelt der Code ist. Die Regel gibt die maximal erlaubte Tiefe vor.

Facet	`COMPLEXITY`
SubFacet	`ComplexityType {NESTING_DEPTH}`
Rule-Class	`ComplexityRule`
Kandidaten-Interface	`ILintComplexityCandidate`
Implementation	Der Kandidat gibt der Regel an auf welchem Level, ausgehend vom obersten Frame des Diagramms er sich befindet und dieser Level wird mit der maximal erlaubten Verschachtelungstiefe verglichen.

Tabelle 32 zeigt alle implementierten Regeln der Kategorie Komplexität.

Tabelle 32: Implementierte Regeln für Komplexität

Identifier	Name	Value	ShortDescription	Diagram-Type	Facet	Sub-Facet
posity:S0016	Complexity (nesting depth) of events should not be too high	3	Complexity of nesting depth is a measure of how hard the control flow of an event is to understand. Events with high Complexity will be difficult to maintain.	MODULE	COMPLEXITY	NESTING_DEPTH

7.5.3.6 Regeln zu Formatierung und Ausrichtung

Regeln, die der Kategorie Formatierung und Ausrichtung angehören, prüfen die Diagramme auf Unschönheiten, welche die Lesbarkeit des Diagramms beeinträchtigen.

Durch die `SubFacet` kann unterschieden werden, welche Art Formatierung oder Ausrichtung überprüft werden soll. Für diese Arbeit wurde eine Regel implementiert, welche prüft, ob die Verbindungslinien auf dem Diagramm optimal ausgerichtet sind.

Facet	`ALIGNMENT`
SubFacet	`AlignmentType {CONNECTOR_ALIGNMENT}`
Rule-Class	`AlignmentRule`
Kandidaten-Interface	`ILintAlignmentCandidate`
Implementation	Der Kandidat gibt der Regel seine Ausgang- und Eingangs-Ports und die jeweiligen Verbindungslinien bekannt, welche in die Ports hinein oder von den Ports weggehen. Die Regel prüft schliesslich, ob die Linien korrekt, im Sinne der Lesbarkeit ausgerichtet sind.

Tabelle 33: Implementierte Regeln für Tabelle 33 zeigt alle implementierten Regeln der Kategorie Formatierung und Ausrichtung.

Tabelle 33: Implementierte Regeln für Formatierung und Ausrichtung

Identifier	Name	Value	ShortDescription	Diagram-Type	Facet	Sub-Facet
posity:S0019	Connectors should be aligned horizontally when port is located left or right and vertically when port is located on top od the component	-	A correct and orderly alignment of the connectors is important for the readability of the diagrams. Please ensure that the connectors leave or enter the ports with the same orientation as the ports are oriented. 1) If the port is placed on the right side of a component, the connector leaves the port horizontally to the right. 2) If the port is placed on the left side of a component, the connector enters the port horizontally from the left. 3) If the port is placed at the top of a component, the connector enters the port vertically from the top. 4) If the port is placed at the bottom of a component, the connector enters the port vertically from the bottom.	MULTIPLE	ALIGNMENT	CONNECTOR_ALIGNMENT

7.5.3.7 Metadaten einer Regel

Die Metadaten und die Regel sind in der Spezifikation im Kapitel 7.4.3.7 beschrieben. Für die Implementation wurden entsprechende Tabellen in der Datenbank und Klassen im Programm angelegt. Zur Laufzeit werden die Regeln vom `PosityLinterRuleContainer` geladen und später in der Analyse angewendet. Abbildung 85 zeigt die Regel und ihre Metadaten.

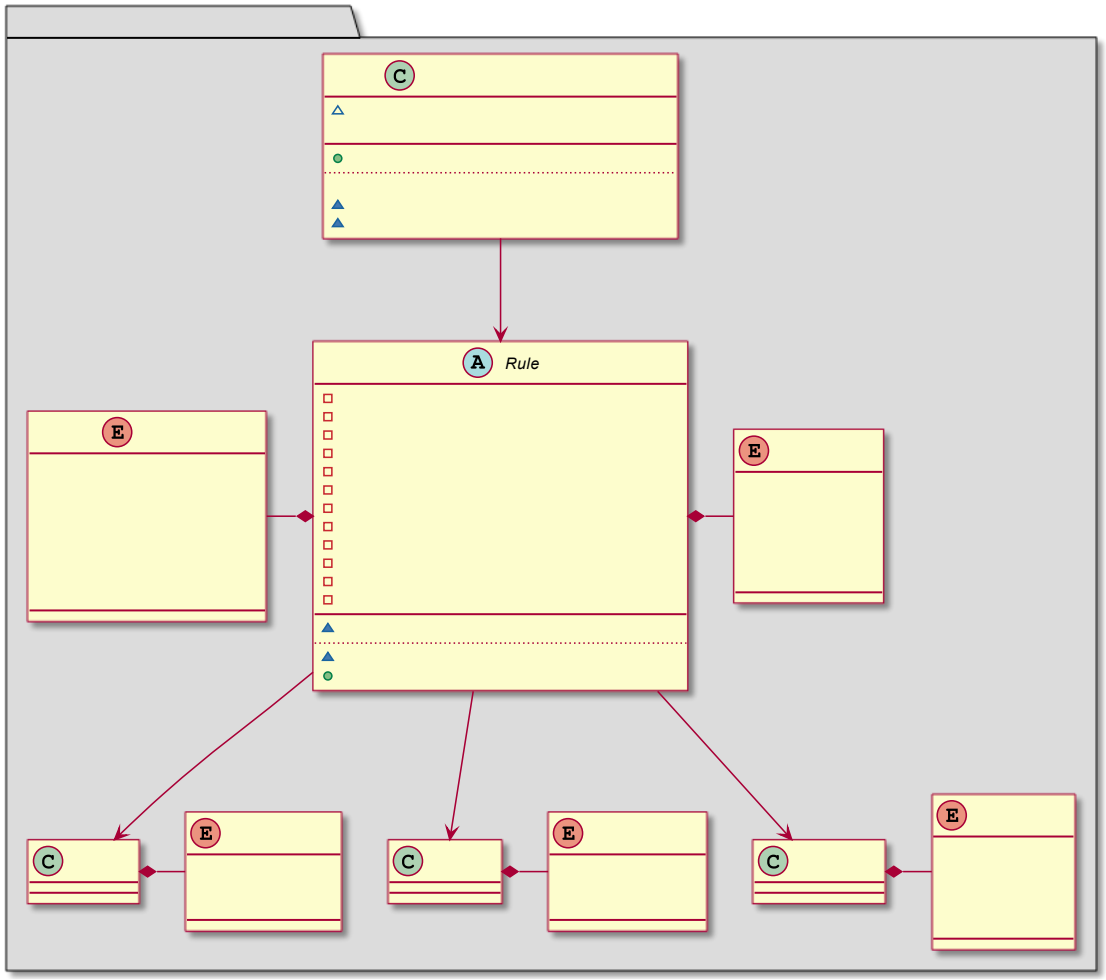


Abbildung 85: Klassendiagramm Regel und Metadaten (Quelle: Eigene Darstellung)

7.5.4 Posity-Shapes und Anwendung von Regeln (Mapping)

Für die Zuordnung Regel zu Element, muss der Posity Linter wissen, welche Regeln für welche Elemente im Diagramm angewandt werden. Diese Information wird in der Mapping-Tabelle in der Datenbank festgehalten. Die Mapping-Tabelle verknüpft Shape und Regel. Es kann festgelegt werden, ob die Regel nur für exakt diesen Typ des Shapes oder auch für Subtypen angewendet werden soll. Zur Laufzeit wird die Tabelle `LinterRuleShapeMapping` von der Klasse `PosityLinterIgnoreListContainer` eingelesen und eine Sammlung von `PosityLintIgnoredElement` angelegt. Der Klassendiagrammausschnitt in Abbildung 86 zeigt das Zusammenspiel zwischen Posity Linter und dem Regelmapping.

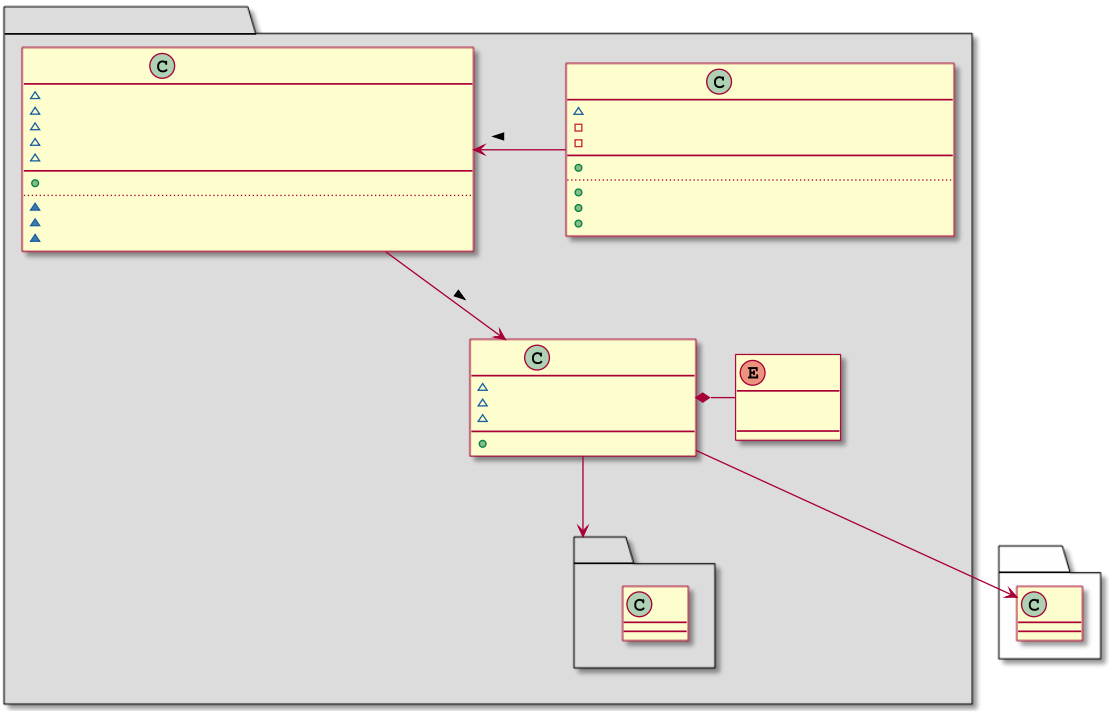


Abbildung 86: Klassendiagramm Ausschnitt Mapping Regel und Shape
(Quelle: Eigene Darstellung)

7.5.5 Issue

Für alle Regel-Überprüfungen, die während einer Analyse fehlschlagen, wird jeweils eine Issue-Instanz erzeugt. Das `Issue` enthält eine Referenz auf die Regel, welche verletzt wurde und auf das `Shape`, welches die Regel verletzt. Für die einfache Ortung des Shapes wird als weitere Referenz, das Diagramm und das Frame festgehalten, in welchen das Shape platziert ist. Damit kann aus dem Issue direkt zu der entsprechenden Stelle im Diagramm gesprungen werden. Die Issues einer Analyse werden zusammengefasst und dem Posity Linter als `PosityLinterResult` zurückgegeben. Abbildung 87, zeigt die Klasse `Issue` und seine Abhängigkeiten.

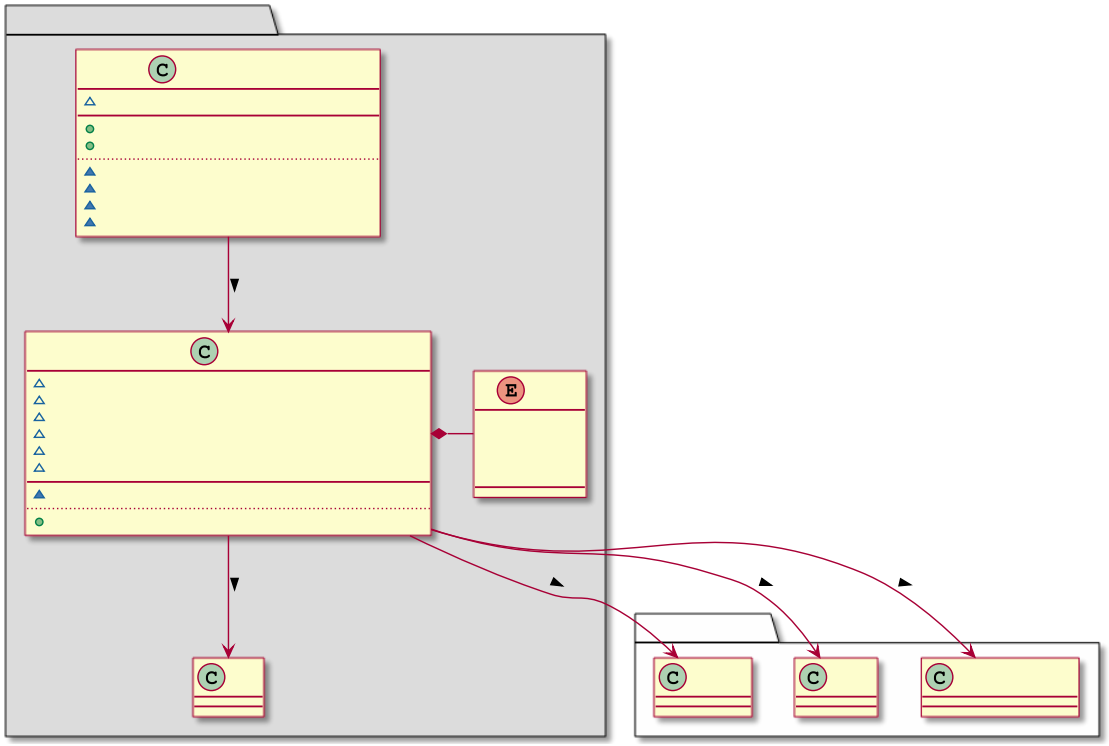


Abbildung 87: Klassendiagramm Ausschnitt Issue (Quelle: Eigene Darstellung)

7.5.6 Ignore-List

Um für Quellcode-Analysen bestimme Elemente oder ganze Diagramme von der Analyse auszuschliessen wird eine Ignore-Liste geführt. Der Posity Linter prüft vor der Anwendung der Regel, ob für diese Regel Ausnahmen definiert worden sind. Ist ein entsprechender Eintrag in der Ignore-Liste vorhanden, wird das Element ignoriert. Die Ignore-Liste wird in der Tabelle `LinterRuleIgnoredElement` in der Datenbank persistiert. Das Datenmodell ist in Abbildung 89 dargestellt. Es kann festgelegt werden, ob die Regel nur für bestimmtes Shapes oder für ein ganzes Diagramm ignoriert werden soll. Weil die Diagramme und die Shapes im Datenmodell pro Typ in jeweils einer eigenen Tabelle gespeichert werden, sind im Datenmodell, stellvertretend für alle Shape- und Diagrammtabellen die zwei Tabellen `Shape` und `Diagram`, gepunkteten dargestellt. Zur Laufzeit wird die Ignore-Liste von der Klasse `PosityLinterIgnoreListContainer` eingelesen und eine Sammlung von `PosityLintIgnoredElement` angelegt. Die Klasse bietet eine Schnittstelle, über welche neue Elemente zur Ignore-Liste hinzugefügt oder wieder von der Liste gelöscht werden können und überprüft werden kann, ob für eine Regel und ein Shape oder Diagramm ein Eintrag existiert. Auf dieser Grundlage entscheidet der Posity Linter, ob eine Regel ignoriert werden muss. Der Klassendiagrammausschnitt Abbildung 88 zeigt das Zusammenspiel zwischen Posity Linter und der Ignore-Liste.

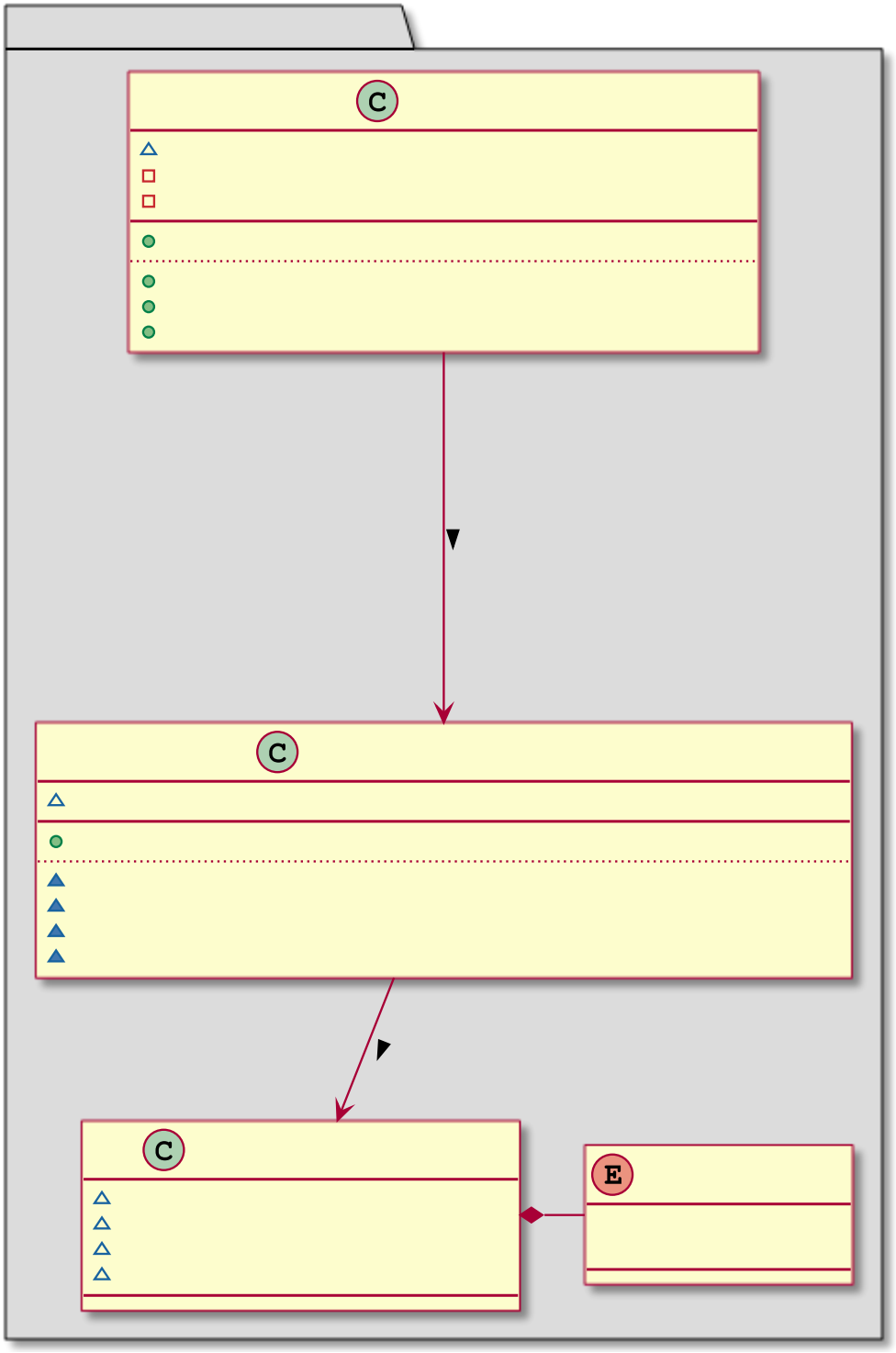


Abbildung 88: Klassendiagramm Ausschnitt Posity Linter und Ignore-Liste

(Quelle: Eigene Darstellung)

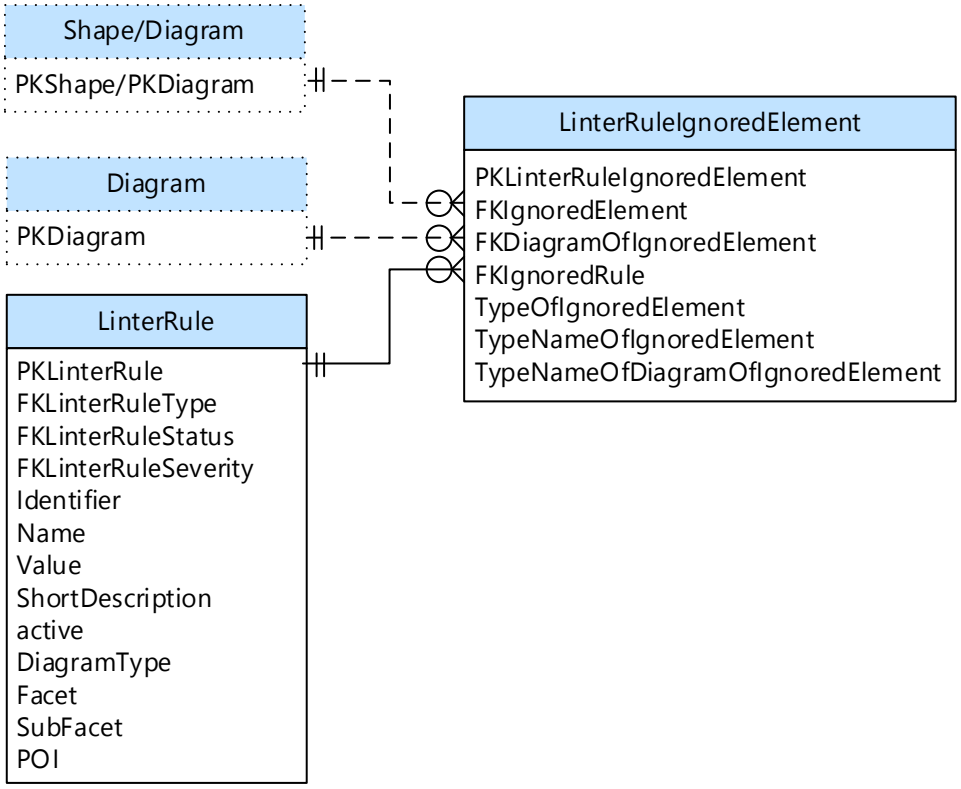


Abbildung 89: Datenmodell Ignore-Liste (Quelle: Eigene Darstellung)

7.6 Validierung

Dieses Kapitel beschreibt die Validierung des Artefakts für die Quellcode-Analyse. Das Artefakt wird validiert, indem eine Code-Analyse in bestehenden Kundenapplikationen der Firma Posity AG durchgeführt wird. Damit können das Artefakt anhand der Erfolgskriterien bewertet und allenfalls problematische Stellen im Code der Kundenapplikation aufgedeckt werden.

7.6.1 Geeignete Posity-Applikationen bestimmen

Als Kandidaten für die Validierung werden zwei möglichst unterschiedliche Kunden-Applikationen verwendet. Einmal ist dies eine sehr umfangreiche Applikation, die schon vor mehreren Jahren erstellt worden ist (~ 8 Jahre) und für die im Laufe der Jahre stets neue Features von verschiedenen Entwickelnden implementiert worden sind. Und einmal ist dies eine neu entwickelte Applikation (2021), welche nur von einer Entwicklerin erstellt worden ist und im Umfang etwa einem Viertel der ersten Applikation entspricht. Vorauszuschicken ist, dass es keine offiziellen Codierungsstandards für die Entwicklung von Posity-Anwendungen gibt, entsprechend gross ist die Herausforderung im Nachhinein einen passenden Standard, beispielsweise für Namenskonventionen, festzulegen. Die

Vielfalt der Posity-Anwendungen wird aber durch den modellbasierten Entwicklungsansatz und vorgegebener Architektur durch die Diagramme naturgemäss weniger ausufernd als dies bei einer textbasierten Sprache möglich wäre, wie beispielsweise Java oder C#.

Die Kapitel 7.6.2 und 7.6.3 beschreiben die für die Validation verwendeten Posity-Kundenapplikationen.

7.6.2 Kundenapplikation ERP für ein Unternehmen in der Solarbranche

Die Applikation SolarvilleERP wurde vor mehr als 8 Jahren erstellt, um für das damalige Startup die wichtigsten Geschäftsprozesse abzubilden. Bis zu diesem Zeitpunkt hatte das Unternehmen die Daten mit Excel-Kalkulationen verwaltet. Diese Lösung wurde jedoch mit dem wachsenden Unternehmen problematisch, weil unter anderem die Benutzung von mehreren Personen gleichzeitig nicht möglich war, viele verteilte Excel-Dateien in der Wartung fehleranfällig waren und einzelne Excel-Dateien sehr gross wurden. Die Posity AG durfte daraufhin für das Unternehmen eine Cloudlösung mit zentraler Datenhaltung, sowie strukturierten Prozessabläufe für die wichtigsten ERP-Prozesse wie CRM, Kundenaufträge und Kundenrechnung umsetzen. Mittlerweile sind viele weitere Prozesse wie Mitarbeiterverwaltung, Arbeitszeiterfassung, Serviceverträge oder Lieferantenverwaltung dazugekommen. Die Applikation ist über die Jahre stetig gewachsen und wurde von verschiedenen Programmierenden erweitert. Entsprechend spannend wird die Quellcode-Analyse sein, um zu sehen, wie konstant bspw. Namensgebungen gewählt worden sind und wie sorgfältig Kommentare nachgeführt worden sind. Tabelle 34 beschreibt den Umfang der Applikation durch die Auflistung der Anzahl Diagramme nach Diagrammtyp. Die Zahlen in der Tabelle zeigen, dass es sich um eine grössere Applikation handelt. Das ist insbesondere auch bezüglich der Beobachtung des Performanceverhaltens vom Posity Linter bei umfangreichen Diagrammen interessant.

Tabelle 34: Übersicht Diagramme der Applikation

Diagrammtyp	**#-Diagramme**
Prozess (nicht Teil der Analyse)	78
Datenmodell (Tabellen)	1 Diagramm mit 69 Tabellen
Query	113

Modul	185
GUI	78
Summe	**377**

7.6.2.1 Konfiguration der Regeln

Für die Konfiguration der Regeln für die Applikation werden bewusst hohe Freiheitsgrade bei den Konventionen zugelassen. Dies, weil die Applikation über einen langen Zeitraum weiterentwickelt worden ist. Beispielsweise wird für Attribut-Namen nur ein Grossbuchstabe zu Beginn des Namens eingefordert oder die Verschachtelungstiefe von Modul-Strukturen auf einen Grenzwertwert von 5 anstelle von 4 (Standard) festgelegt. Tabelle 35 zeigt die Regeln für die Applikation.

Tabelle 35: Regeln für Applikation SolarvilleERP

Identifier	Name	Value	active
posity:S0001	Primarykey system names should comply with a naming convention	^(PK_)[a-zA-Z0-9_]*$	1
posity:S0002	Primarykey logical names should comply with a naming convention	^(PK_).*$	1
posity:S0003	Table system names should comply with a naming convention	^[A-Z][a-zA-Z0-9]*$	1
posity:S0004	Table logical names should comply with a naming convention	^[A-Z]+.*$	1
posity:S0005	Foreignkey system names should comply with a naming convention	^(FK_)[a-zA-Z0-9_]*$	1
posity:S0006	Foreignkey logical names should comply with a naming convention	^(FK_).*$	1
posity:S0007	Attribute system names should comply with a naming convention	^[A-Z]+.*$	1
posity:S0008	Attribute logical names should comply with a naming convention	^[A-Z]+.*$	1
posity:S0009	Tableindex system names should comply with a naming convention	^.*	1
posity:S0010	Tableindex logical names should comply with a naming	^.*	1
posity:S0011	Table integrity rule system names should comply with a naming convention	^.*	1
posity:S0012	Table integrity rule logical names should comply with a naming convention	^.*	1
posity:S0013	Shape needs an appropriate comment	^(.|\s){10,}$	1

posity:S0016	Complexity (nesting depth) of events should not be too high	5	1
posity:S0018	Elements on the user interface should have a tooltip behind them		1
posity:S0014	Variable names should comply with a naming convention	^([A-Z]|[0-9])+.*$	1
posity:S0015	Titles of structures such as switch case, sequence, try catch, posity script and lanes must be set.	^[^<].*[^>]$	1
posity:S0017	Result tables of database queries should have a defined sort order		1
posity:S0019	Connectors should be aligned horizontally when port is located left or right and vertically when port is located on top of the component		1

7.6.2.2 Ergebnis der Quellcode-Analyse

Die Analyse der gesamt Applikation SolarvilleERP macht wegen der umfangreichen Grösse der Applikation weniger Sinn. Es macht mehr Sinn bestimmte, wichtige und komplexe, Diagramme zu analysieren. Bei der Analyse dieser ausgewählten Diagramme, wie beispielsweise der Kundenofferte, sind Issue aufgedeckt worden, welche der Aufmerksamkeit bedürfen und bei einem Refactoring überarbeitet werden sollten. Die Regelkonfiguration der Namenskonventionen wurde bewusst wenig einschränkend gewählt, hier könnte eine Konkretisierung angedacht werden, um bei der Namensgebung einheitlichere Vorgaben zu machen. Im Folgenden werden die Resultate der Analyse aufgelistet, wobei die Analyse für die Kundenofferte für jeden Diagrammtyp (GUI, Module, Query und Tabelle) einzeln ausgeführt wurde.

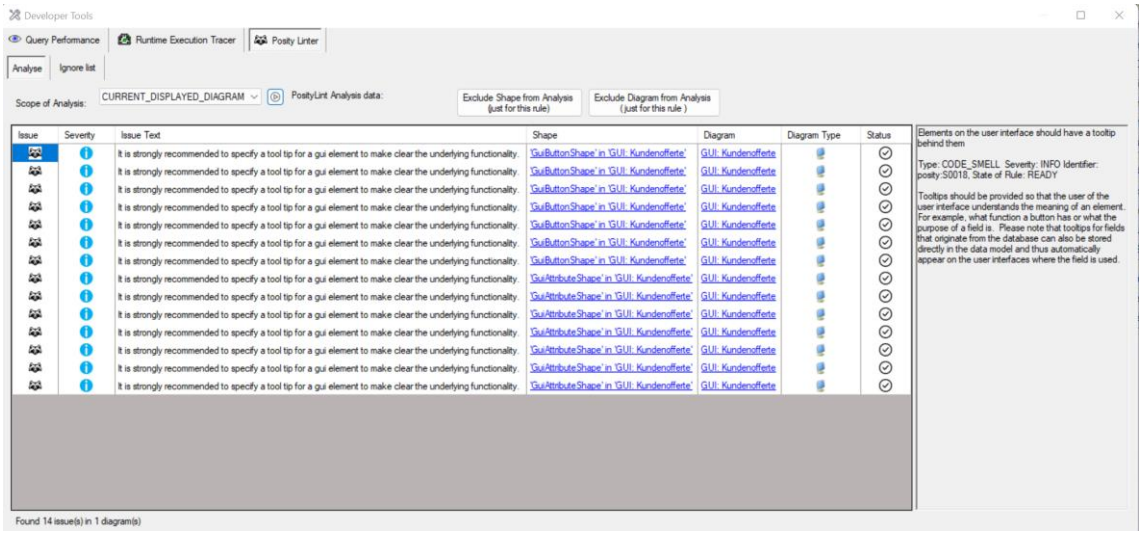


Abbildung 90: Resultat Analyse GUI Kundenofferte SolarvilleERP

(Quelle: Eigene Darstellung)

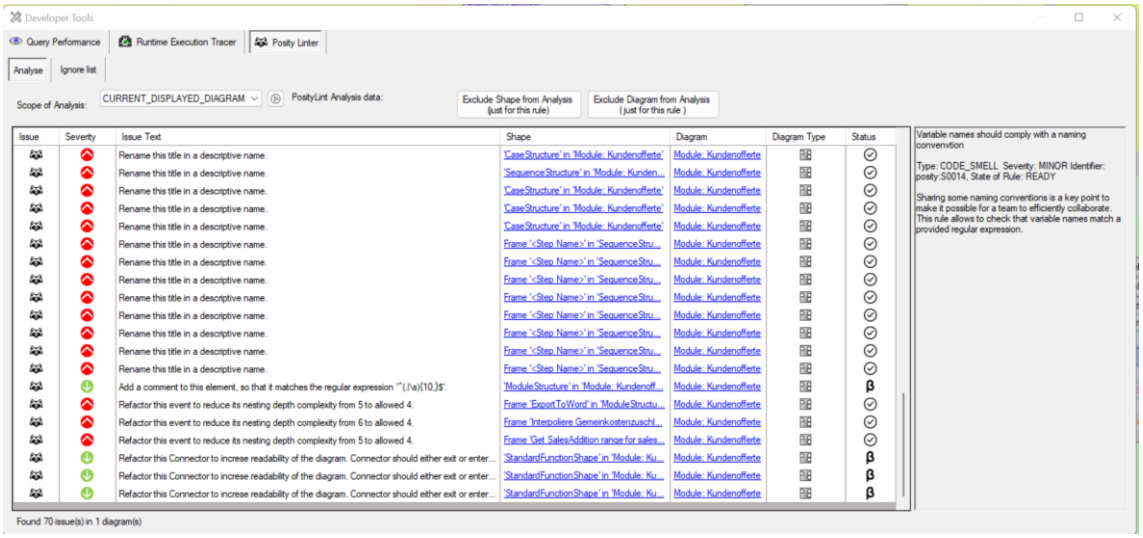


Abbildung 91: Resultat Analyse Modul Kundenofferte SolarvilleERP

(Quelle: Eigene Darstellung)

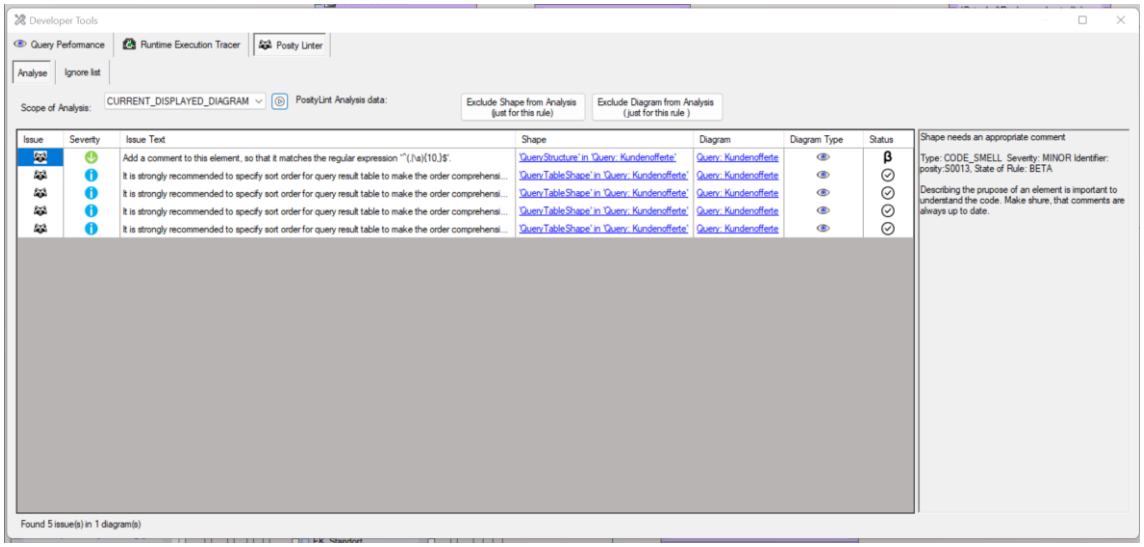


Abbildung 92: Resultat Analyse Query Kundenofferte SolarvilleERP

(Quelle: Eigene Darstellung)

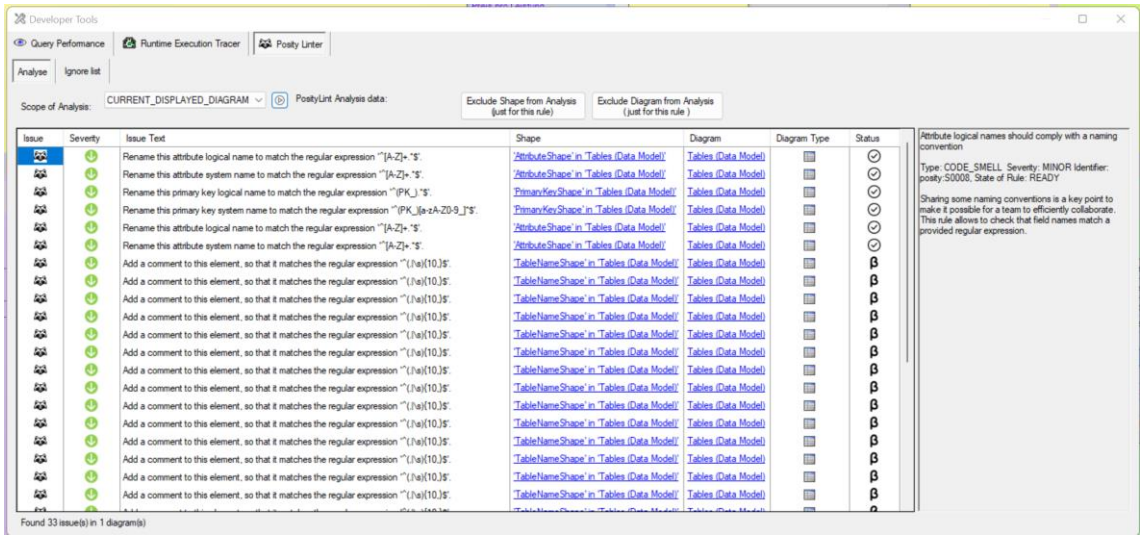


Abbildung 93: Resultat Analyse Datenmodell (Tabellen) SolarvilleERP

(Quelle: Eigene Darstellung)

7.6.3 Kundenapplikation Normenhilfe für die Norm EN378 für die Kältemittelbranche (Schweizerischer Verband für Kältetechnik)

Die Normenhilfe für die EN378-Norm wurde für die Kältemittelbranche entwickelt, um bei Bauvorhaben von Kältemittelanlagen Unterstützung in der Umsetzung und Erfüllung der Norm zu bieten. Die Applikation enthält viel Businesslogik, welche das Regelwerk der Norm implementiert. Dementsprechend sind die Modul-Diagramme gross und kom-

plex. Die Applikation wurde im Jahr 2021 gebaut, wurde von einer Entwicklerin programmiert und ist bezüglich des Umfangs, gemessen an der Zahl der Diagramme, eine kleine Anwendung. Die Wahl dieser Applikation ist darauf zu begründen, dass es die jüngste mit Posity gebaute Applikation ist und dementsprechend schon viele Erfahrungswerte aus vorherigen Applikationen miteinbezogen werden konnten, der Funktionsumfang der Posity IDE inzwischen mehr Möglichkeiten bietet, um die Entwicklung von Anwendungen zu unterstützen und weil die Applikation von nur einer Person entwickelt worden ist. Aus den erwähnten Gründen dürfte die Qualität, gemessen an der Anzahl gefunden Issues, entsprechend höher sein als im Vergleich zu der Applikation Solarville-ERP. Tabelle 36 beschreibt den Umfang der Applikation durch die Auflistung der Anzahl Diagramme nach Diagrammtyp.

Tabelle 36: Übersicht Diagramme der Applikation

Diagrammtyp	**#-Diagramme**
Prozess (nicht Teil der Analyse)	9
Datenmodell (Tabellen)	1 Diagramm mit 10 Tabellen
Query	10
Modul	9
GUI	8
Summe	**28**

7.6.3.1 Konfiguration der Regeln

Auch für diese Applikation, wurden die Regeln bewusst mit hohen Freiheitsgraden konfiguriert. Dies, weil die Entwicklerin zum Zeitpunkt der Erstellung der Applikation über keine vergebenen Konventionen verfügte. Tabelle 37 zeigt die Regeln für die Applikation. Die Konfiguration konnte bis auf den Parameter der Verschachtelungstiefe von der Konfiguration aus Kapitel 7.6.2.1 übernommen werden.

Tabelle 37: Regeln für Applikation EN378

Identifier	Name	Value	active

posity:S0001	Primarykey system names should comply with a naming convention	^(PK_)[a-zA-Z0-9_]*$	1
posity:S0002	Primarykey logical names should comply with a naming convention	^(PK_).*$	1
posity:S0003	Table system names should comply with a naming convention	^[A-Z][a-zA-Z0-9]*$	1
posity:S0004	Table logical names should comply with a naming convention	^[A-Z]+.*$	1
posity:S0005	Foreignkey system names should comply with a naming convention	^(FK_)[a-zA-Z0-9_]*$	1
posity:S0006	Foreignkey logical names should comply with a naming convention	^(FK_).*$	1
posity:S0007	Attribute system names should comply with a naming convention	^[A-Z]+.*$	1
posity:S0008	Attribute logical names should comply with a naming convention	^[A-Z]+.*$	1
posity:S0009	Tableindex system names should comply with a naming convention	^.*	1
posity:S0010	Tableindex logical names should comply with a naming convention	^.*	1
posity:S0011	Table integrity rule system names should comply with a naming convention	^.*	1
posity:S0012	Table integrity rule logical names should comply with a naming convention	^.*	1
posity:S0013	Shape needs an appropriate comment	^(.|\s){10,}$	1
posity:S0016	Complexity (nesting depth) of events should not be too high	4	1
posity:S0018	Elements on the user interface should have a tooltip behind them		1
posity:S0014	Variable names should comply with a naming convention	^([A-Z]|[0-9])+.*$	1
posity:S0015	Titles of structures such as switch case, sequence, try catch, posity script and lanes must be set.	^[^<].*[^>]$	1
posity:S0017	Result tables of database queries should have a defined sort order		1
posity:S0019	Connectors should be aligned horizontally when port is located left or right and vertically when port is located on top of the component		1

7.6.3.2 Ergebnis der Quellcode-Analyse

Da die Applikation EN378-Tool vergleichsweise klein ist, kann die ganze Applikation auf einmal analysiert werden. Das Ergebnis (siehe auch Abbildung 94) lautet, in 27 Diagrammen wurden 614 Issues aufgedeckt. Die meisten Issues betreffen Verstösse gegen-

über der Namenskonventionen, wovon die einen durchaus berechtig sind. Andere wiederum können begründet werden und bedürften wohl entweder einer Anpassung der Konvention (Regex-Ausdruck) oder dem Hinzufügen des Elements zur Ignore-Liste. Bei allen anderen Issues ist eine Überarbeitung des Codes angebracht und entweder ein entsprechendes Refactoring am Code vorzunehmen oder das Element der Ignore-Liste hinzuzufügen. Insbesondere wurden in dieser Applikation viele Issues in der Kategorie Formatierung aufgedeckt. In dieser Applikation hatten die Tooltips einen sehr hohen Stellenwert, entsprechend sorgfältig wurden diese implementiert. Das bestätig auch das Resultat der Analyse, es fehlen bis auf wenige Ausnahmen kaum Kommentare.

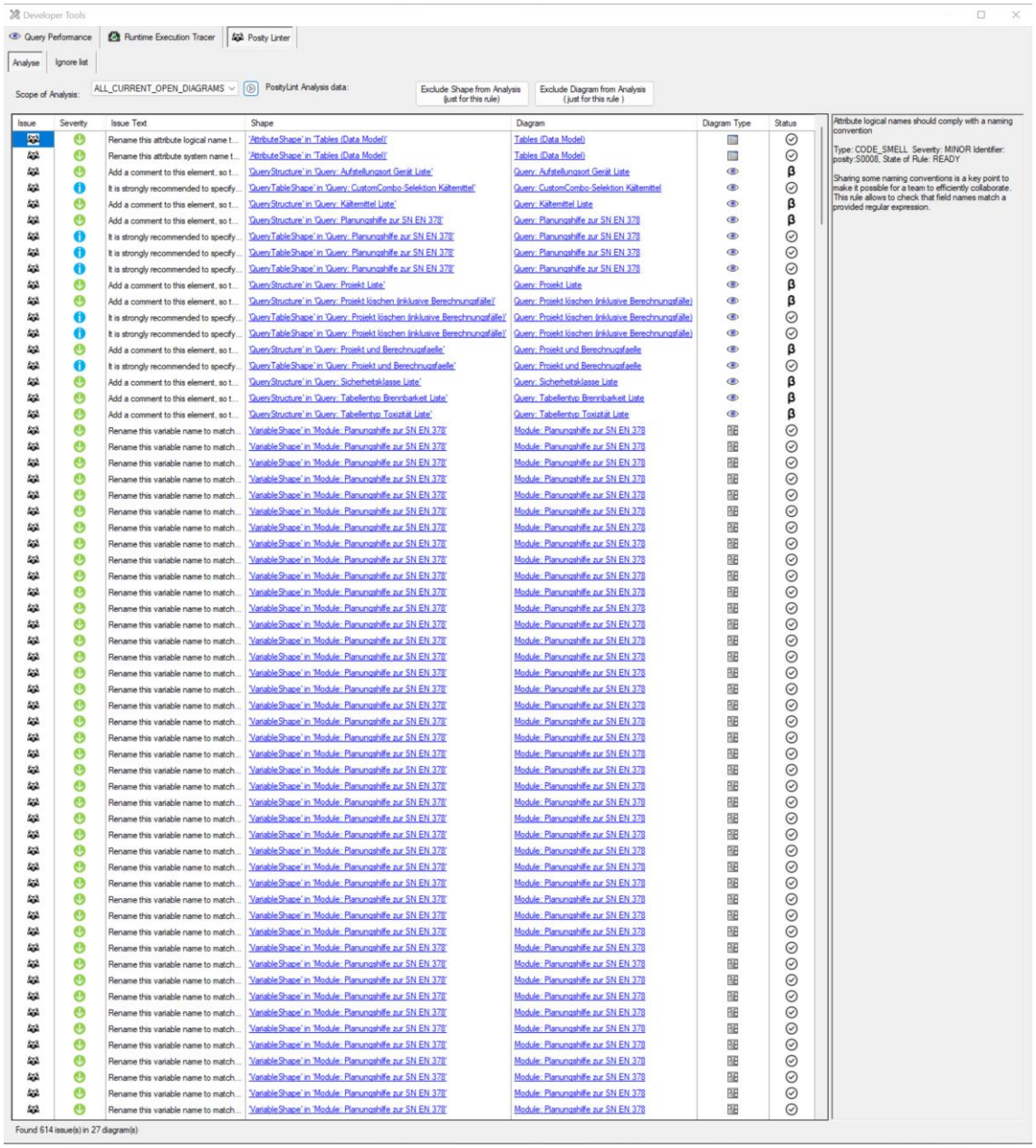


Abbildung 94: Resultat Analyse EN378-Tool (Quelle: Eigene Darstellung)

7.6.4 Bewertung der Erfolgskriterien

Die Erfolgskriterien gemäss Kapitel 7.3 werden anhand der Beobachtung des Artefakts bei der Quellcode-Analyse, der beiden Posity-Applikationen bewertet. Für die beiden Posity-Applikationen wird anstelle des vollen Namens ein Alias verwendet. Für die ERP Anwendung aus der Solarbranche wird die Bezeichnung **«Sol ERP»** und für die Normenhilfe wird die Bezeichnung **«EN378 Tool»** verwendet. Für die Bewertung werden die Erfolgskriterien, pro Validierungsfall, einem Erfüllungsgrad zugeordnet. Tabelle 38 enthält die Legende zu den verschiedenen Varianten vom Erfüllungsgrad.

Tabelle 38: Legende Erfüllungsgrad für Bewertung

Symbol Erfüllungsgrad	**Bedeutung**
Ja	Kriterium ist vollständig erfüllt
Nein	Kriterium ist nicht erfüllt
~	Kriterium ist teilweise erfüllt
!	Kriterium ist nicht validierbar
-	Kriterium hat keine Relevanz für diesen Teil

Es folgt in Tabelle 39 die Bewertung aller Erfolgskriterien mit Zuordnung des Erfüllungsgrades sowie Beschreibung zur Beobachtung des Artefakts im Kontext der beiden Kundenapplikationen.

Tabelle 39: Bewertung Erfolgskriterien

Kriterium	**KCA1**	
Beschreibung	Die Code-Analyse für eine beliebige Posity-Applikation in Posity möglich.	
Bewertung «Sol ERP»	Bewertung	Erfüllungsgrad
	Code-Analyse kann für die Anwendung «Sol ERP» ausgeführt werden.	Ja
Bewertung «EN378 Tool»	Bewertung	Erfüllungsgrad
	Code-Analyse kann für die Anwendung «EN378 Tool» ausgeführt werden.	Ja
Kommentar	Damit die Analyse durchgeführt werden können, müssen auf den Datenbanken der Applikationen die Tabellen und SQL-Prozeduren, welche der Posity Linter braucht mit einem Skript erstellt und initialisiert werden. Zusätzlich muss der Regelsatz pro Applikation individuell konfiguriert werden. So müssen beispielsweise Regex-Ausdrücke für die Namenskonventionen oder Anforderungen an die maximal erlaubte Verschachtelungstiefe festgelegt werden.	

Kriterium	**KCA2**	
Beschreibung	Die Datenbank enthält die Regeln und die Konfiguration der Regeln, die Anwendung erfolgt im Code.	
Bewertung «Sol ERP»	Bewertung	Erfüllungsgrad
	Für alle Kundenapplikation gleich	Ja
Bewertung «EN378 Tool»	Bewertung	Erfüllungsgrad
	Für alle Kundenapplikation gleich	Ja
Kommentar	Die Regeln sind in der Applikationsdatenbank abgelegt. Die Konfiguration erfolgt direkt in den entsprechenden Tabellen der Datenbank. Die Auswertung der Regeln erfolgt zur Laufzeit durch entsprechende Klassen.	

Kriterium	**KCA3**	
Beschreibung	Das Artefakt kann so erweitert werden, dass alle Shapes aller Diagramme analysiert werden können.	
Bewertung «Sol ERP»	Bewertung	Erfüllungsgrad
	Für alle Kundenapplikation gleich	Ja
Bewertung «EN378 Tool»	Bewertung	Erfüllungsgrad
	Für alle Kundenapplikation gleich	Ja
Kommentar	Damit die Quellcode-Analyse auf zusätzliche Diagramme und Shapes ausgeweitet werden kann, sind folgende zwei Schritte zu unternehmen: - Die Shape-Klassen müssen das Regelspezifische ILintCandidate-Interface und die damit geforderten Methoden implementieren	

<table>
<tr><td></td><td colspan="2">- Die Shape-Klasse für die gewünschte Regel der Mapping-Tabelle hinzufügen

Danach wird der Posity Linter diese Shapes in die Analyse mit einbeziehen.</td></tr>
<tr><td>Kriterium</td><td colspan="2">KCA4</td></tr>
<tr><td>Beschreibung</td><td colspan="2">Das Artefakt unterstützt die Posity-Entwickler bei ihrer täglichen Arbeit. Die Code-Analyse motiviert den Entwickler dazu, saubere Code zu schreiben.</td></tr>
<tr><td rowspan="2">Bewertung «Sol ERP»</td><td>Bewertung</td><td>Erfüllungsgrad</td></tr>
<tr><td>Für alle Kundenapplikation gleich</td><td>Ja</td></tr>
<tr><td rowspan="2">Bewertung «EN378 Tool»</td><td>Bewertung</td><td>Erfüllungsgrad</td></tr>
<tr><td>Für alle Kundenapplikation gleich</td><td>Ja</td></tr>
<tr><td>Kommentar</td><td colspan="2">Ein Posity-Entwickler hat den Posity Linter an einer Applikation getestet, an deren Entwicklung er massgeblich beteiligt war. Die Rückmeldung war positiv, mit den gefundenen Issues war der Entwickler einverstanden oder sogar froh, dass diese entdeckt worden sind. Die Ignore-Liste wurde als nützlich erachtet und wurde eingesetzt. Die Regex-Ausdrücke bei den Namenskonventionen sind schwierig zu interpretieren (Erklärungen im Kommentar ergänzen). Beim Issue sollten zusätzliche Informationen zum konkreten Objekt angezeigt werden, die der Identifizierung dienen. Beispielsweise wäre es nützlich zu sehen, wie die betroffene Variable in folgendem Issue-Text heisst: "Rename this variable name to match the regular expression '^([A-Z]|[0-9])+.*$'."

Das Bedürfnis, weitere Regeln zu haben, um den die Qualität zu prüfen wurde geäussert. Ebenfalls wurde der Wunsch platziert, dass die Analyse parallel zur Programmierung laufen sollte.</td></tr>
</table>

Kriterium	**KCA5**	
Beschreibung	Ein Posity-Entwickler kann das Artefakt ohne Anleitung bedienen.	
Bewertung «Sol ERP»	Bewertung	Erfüllungsgrad
	Für alle Kundenapplikation gleich	~
Bewertung «EN378 Tool»	Bewertung	Erfüllungsgrad
	Für alle Kundenapplikation gleich	~
Kommentar	Eine kleine mündliche Einweisung über die Bedienung des Tools war für den Posity-Entwickler notwendig. Hauptsächlich ging es dabei aber um das Starten des Tools. Die Analyse und das Verwenden der Ignore-Liste wurden vom Entwickler ohne Einweisung ausgeführt. Was fehlt, sind Erklärungen (Tooltips) für die verwendeten Symbole, beispielsweise beim Status oder der Severity. Zudem wurde eine Möglichkeit zur Filterung und Sortierung von Issues geäussert.	

Kriterium	**KCA6**	
Beschreibung	Folgende Qualitätsmerkmale einer mit der Posity IDE erstellen Applikation können durch die Code-Analyse gesteigert werden: - Analysierbarkeit - Anpassbarkeit - Richtigkeit	
Bewertung «Sol ERP»	Bewertung	Erfüllungsgrad
	Durch die Reife und den Umfang dieser Posity-Applikation ist nur festzustellen, dass in den bestehenden Diagrammen viel Potential für Verbesserungen der Codequalität vorhanden ist. Ein Refactoring wäre in diesem Fall ein umfangreiches Vorhaben und die Kosten/Nutzen-Frage müsse zuerst im Detail auch mit dem Kunden geklärt werden. Was aber festgestellt werden konnte, dass keine bestimmte Namenskonventionen eingehalten worden sind und viele Issues im Bereich Kommentare und Spezifikation aufgedeckt worden sind. Gerade diese Problemstellen könnten mit wenig Aufwand Sukzessive behoben werden und die Analysierbarkeit und Anpassbarkeit zu erhöhen. Auf die Richtigkeit könnte insofern eine Steigerung erfolgen, als das Fehler durch die Präzisierung der Spezifikation und der Kommentare aufgedeckt werden könnten.	!
	Bewertung	Erfüllungsgrad

Bewertung «EN378 Tool»	Im Vergleich und Verhältnis zu «Sol ERP» weisst «EN378 Tool » deutlich weniger Issues auf. Dies könnte darauf zurückzuführen sein, dass die Applikation von nur einer Entwicklerin programmiert worden ist. Viele Qualitätssteigernde Massnahmen, wie beispielsweise sorgfältiges Kommentieren, einhalten von Namenskonventionen oder beschreiben von Tooltips wurden über weite Strecken umgesetzt. Der wunde Punkt der Applikation dürfte die Komplexität sein. Die teils tiefe Verschachtelung von Code erschwert die Lesbarkeit und damit die Analysierbarkeit der Diagramme. Diese Teile müssten modularisiert und die Aufgaben auf Sub-Events verteilt werden.	!
Kommentar	Das Kriterium kann bis zum jetzigen Zeitpunkt nicht validiert werden und es können nur Annahmen getroffen werden, weil für die Beantwortung ein Vorher-Nachher-Vergleich notwendig wäre. Da die Applikationen jedoch nur analysiert aber nicht refactored worden sind, kann nur eine Annahme, auf Basis der gefundenen Issues, auf das vorhandene Verbesserungspotential der Qualitätsmerkmale gemacht werden.	

Kriterium	**KCA7**	
Beschreibung	Die Performance der Posity IDE darf durch die Analyse nicht spürbar beeinträchtig werden.	
Bewertung «Sol ERP»	Bewertung	Erfüllungsgrad
	Für alle Kundenapplikation gleich	Ja
Bewertung «EN378 Tool»	Bewertung	Erfüllungsgrad
	Für alle Kundenapplikation gleich	Ja
Kommentar	Da die Quellcode-Analyse nicht automatisch im Hintergrund ausgeführt wird, sondern explizit über die Benutzeroberfläche vom Posity Linter gestartet werden muss, wird der Entwickler während der Programmierung nicht beeinträchtig. Weiter ist die Quellcode-Analyse performant und liefert zeitnah Resultate (siehe auch KCA8). Selbsterklärend ist die Antwortzeit der Analyse abhängig von der Anzahl zu prüfenden Diagrammen.	

Kriterium	**KCA8**	
Beschreibung	Die Analyse muss zeitnahe Resultate liefern - 1 Diagramm < 1 Sekunde - 10 Diagramme < 5 Sekunden - 50 Diagramme < 10 Sekunden	
Bewertung «Sol ERP»	Bewertung	Erfüllungsgrad
	1 Diagramme in < 1 Sekunde analysiert 77 Diagramme (GUI) (526 Issues) ~ 2.11 Sekunden 155 Diagramme (GUI, Module, Table) 1359 Issues ~2.40 Sekunden	Ja
Bewertung «EN378 Tool»	Bewertung	Erfüllungsgrad
	27 Diagramme (ganze Applikation) < 2 Sekunden analysiert	Ja
Kommentar	Die zeitlichen Vorgaben werden eingehalten oder sogar übertroffen. Die Analyse ist auch für grosse und viele Diagramme erstaunlich performant.	

Kriterium	**KCA9**	
Beschreibung	Durch das Resultat der Code-Analyse kann festgestellt werden, wo im Code welches Problem vorliegt, und es kann einfach zu der entsprechenden Stelle im Code navigiert werden.	
Bewertung «Sol ERP»	Bewertung	Erfüllungsgrad
	Für alle Kundenapplikation gleich	Ja
Bewertung «EN378 Tool»	Bewertung	Erfüllungsgrad
	Für alle Kundenapplikation gleich	Ja
Kommentar	Das Resultat einer Code-Analyse durch den Posity Linter enthält für jede Regelverletzung ein Issue. Im Issue sind die Koordinaten des regelverstossenden Shapes verlinkt. Über die Benutzeroberfläche kann zu der entsprechenden Stelle gesprungen werden.	

Kriterium	**KCA10**	
Beschreibung	Eine Analyse führt bei wiederholter Ausführung immer zum gleichen Resultat, wenn sich am Code und den Regeln nichts geändert hat, die Ausschlusskriterien und der Betrachtungsraum (Scope) gleichgeblieben sind.	
Bewertung «Sol ERP»	Bewertung	Erfüllungsgrad
	Wiederholung führt zum selben Resultat	Ja
Bewertung «EN378 Tool»	Bewertung	Erfüllungsgrad
	Wiederholung führt zum selben Resultat	Ja
Kommentar	Das Resultat für jede erneute Analyse ist jeweils unverändert.	

Kriterium	**KCA11**	
Beschreibung	Es können neue Regeln hinzugefügt, geändert und gelöscht werden. Möglichst ohne, dass die Posity IDE angepasst werden muss.	
Bewertung «Sol ERP»	Bewertung	Erfüllungsgrad
	Für alle Kundenapplikation gleich	~
Bewertung «EN378 Tool»	Bewertung	Erfüllungsgrad
	Für alle Kundenapplikation gleich	~
Kommentar	Kommen neue Regeln für bestehende Regelkategorien (Facets) hinzu, können diese ohne Anpassung der Posity IDE integriert werden. Kommen neue Regeln für bestehende Regel-Typen hinzu, die erfordern, dass die Prüfung der bestehenden Regeltypen erweitert werden, muss die entsprechende Klasse für die Regel im Code angepasst werden, was wenig Aufwand macht, jedoch eine Anpassung der Posity IDE verlangt. Aufwändiger wird es, wenn für diese Implementation zusätzliche Informationen vom Kandidaten (Shape) benötigt werden, dann ist eine Interface-Anpassung des Kandidaten-Interfaces nötig und damit eine Implementation für alle Shapes, welche dieses Interface implementieren. Am meisten Aufwand macht die Erstellung einer neuen Regelkategorie, dies hat zur Folge, dass ein neuer Facet-Typ eingeführt, eine Regelklasse implementiert und ein entsprechendes Kandidaten-Interface definiert und bei den Kandidaten implementiert werden muss.	

Kriterium	**KCA12**	
Beschreibung	Es können Ausschlusskriterien für Teile festgelegt werden so, dass diese nicht mehr analysiert werden.	
Bewertung «Sol ERP»	Bewertung	Erfüllungsgrad
	Für alle Kundenapplikation gleich	Ja
Bewertung «EN378 Tool»	Bewertung	Erfüllungsgrad
	Für alle Kundenapplikation gleich	Ja
Kommentar	Programmteile können über die Ignore-Liste von der Analyse ausgeschlossen werden. So kann ein einzelnes Shape oder ein ganzes Diagramm für eine Regel von der Analyse ausgeschlossen werden. Ausgeschlossene Teile können durch Entfernen von der Ignore-Liste wieder mit in die Analyse einbezogen werden.	

Kriterium	**KCA13**	
Beschreibung	Für alle vier Diagrammtypen (GUI, Modul, Query und Tabellen) gibt es Regeln	
Bewertung «Sol ERP»	Bewertung	Erfüllungsgrad
	Für alle Kundenapplikation gleich	Ja
Bewertung «EN378 Tool»	Bewertung	Erfüllungsgrad
	Für alle Kundenapplikation gleich	Ja
Kommentar	19 Regeln für 6 Regelkategorien wurden erstellt, wovon für jeden Diagrammtyp mindestens eine Regel erstellt wurde. Kapitel 7.5.3 enthält die Aufstellung aller Regeln und weist jeweils in der Spalte «Diagram» das Diagramm aus für welches die Regel relevant ist.	

Zusammenfassend sind von den **insgesamt 13 Erfolgskriterien 10** vollständig erfüllt, **2** teilweise erfüllt, **1** Kriterium gilt als nicht validierbar und **0** Kriterien nicht erfüllt.

Die Validierung des konkreten Artefakts hat folgende Mängel und oder Wünsche zu Tage gefördert. Die Auflistung enthält sehr plattformspezifische Punkte und ist vor allem für die Weiterentwicklung des Prototyps für Posity interessant und deshalb nur bedingt in die Validierung des Artefakts miteinzubeziehen:

- Weitere Metriken und Regeln implementieren
- Instant Linting
- Quickfixes für die gefundenen Issues vorschlagen
- Die Linter-Regeln sollten in der Einstellung konfiguriert werden können (aktiv/inaktiv, Regex, ...) - Regel Konfigurator
- Regeln mit bestimmter Severity müssen beim Deployment des Packages überprüft und durchgesetzt werden - (CI)
- Alle Diagramme eines Packages sollten (geöffnet und) überprüft werden können, damit sie vor dem Deployment kontrolliert werden können
- Ignore-Liste erweitern so, dass Elemente auch unabhängig von Regeln ignoriert werden können
- Regex-Ausdrücke für Namenskonventionen sind schwierig zu formulieren und schwierig zu lesen
- Ganze Applikation prüfen muss möglich sein, ohne den Code (Diagramme) vorher zu öffnen.

8 Konzept für spezifikationsorientierte Testverfahren (Blackbox-Testing)

Dieses Kapitel beschreibt die Anwendung vom spezifikationsbasierten Software-Testing-Verfahren Blackbox-Testing in einer Low-Code Entwicklungsumgebung. Im Kapitel 8.1 wird das Blackbox-Verfahren und dessen Methoden erklärt und daraus Anforderungen für eine mögliche Implementierung in der Low-Code Entwicklungsumgebung Posity Design Studio abgeleitet. Die aus den Anforderungen abgeleitete Spezifikation wird abschliessend in einer Delphi-Runde mit den Hauptentwicklern des Posity Design Studios geprüft und damit validiert, ob sich das Blackbox-Testing-Verfahren auch in einer Low-Code Entwicklungsumgebung anwenden lässt.

8.1 Anwendungskontext

Das Blackbox-Testing-Verfahren (Blackbox) ist in die dynamischen Testverfahren einzuordnen. Bei diesem Testverfahren wird ein Software-Testobjekt mit Eingabedaten versehen und auf einem Computer ausgeführt. Damit kann die Implementation einer Komponente gegen die aus der Software-Spezifikation stammenden Anforderungen geprüft werden.

Meistens soll eine Komponente nach der Fertigstellung getestet werden, aber eine äussere Komponente fehlt noch, die sie ausführen könnte. Deshalb ist ein sogenannter Testrahmen (engl. test bed) erforderlich, in dem die Komponente als Testobjekt eingebettet und ausgeführt werden kann. Oft sind auch die Schnittstellen (z.B. andere Komponenten, Webservices, die Datenbank, usw.) die das Testobjekt ansprechen möchte, nicht fertig implementiert oder nicht verfügbar. Um trotzdem das Testobjekt testen zu können werden die fehlenden Schnittstellen in sogenannten Stellvertretern implementiert (mocking → mocks), um damit die Abhängigkeiten zu diesen Schnittstellen für das Testen zu vermeiden.

Liegt das Testobjekt eingebettet im Testrahmen, kann dieses mit unterschiedlichen Testfällen aufgerufen werden. Die Ausführung geschieht mit einem Testtreiber (engl. test runner), der zusätzlich zur Ausführung der Testfälle auch deren Testausgaben mitprotokolliert, oder gegebenenfalls die Testausführung abbricht. Die Abbildung 95 zeigt die in diesem Abschnitt erwähnten und benötigten Bestandteile eines Blackbox-Test-Systems.

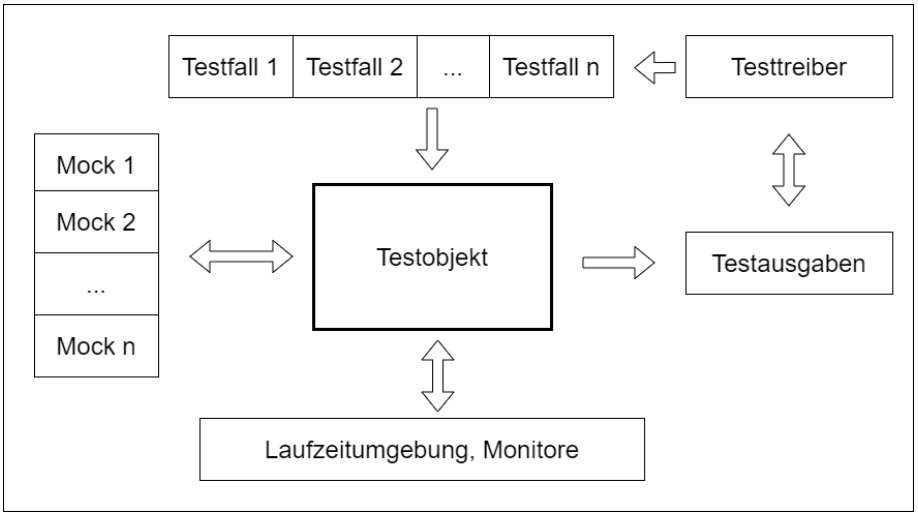


Abbildung 95: Übersicht Bestandteile eines Testrahmens
(Quelle: (Andreas & Tilo, 2012, S. 109))

8.1.1 Testschritte

Für ein erfolgreiches Blackbox-Testing sind gemäss (Andreas & Tilo, 2012) folgende Schritte notwendig:

1. Bedingungen und Voraussetzungen für den Test und die verfolgten Ziele festlegen.
2. Einzelne Testfälle spezifizieren.
3. Testausführung festlegen (in der Regel eine Aneinanderreihung von mehreren Testfällen).

8.1.2 Testverfahren

Unter Blackbox-Testing sind folgende Testverfahren einzuordnen (Andreas & Tilo, 2012):

1. Äquivalenzklassenbildung
2. Grenzwertanalyse
3. Zustandsbezogener Test
4. Ursache-Wirkungs-Graph-Analyse (Beispiel)
5. Anwendungsfallbasierter Test

Bei der Äquivalenzklassenbildung wird die Menge der möglichen Eingabeparameter für einen Komponentenparameter in Klassen (Äquivalenzklassen) eingeteilt. Zum Beispiel

kann ein Zahlenparameter (ganze Zahlen) aus 3 Äquivalenzklassen bestehen: Not a number, Int.MIN…0, 0…Int.Max). Bestmöglich wird getestet indem aus jeder Äquivalenzklasse ein beliebiger Wert verwendet wird.

Grenzwertanalysen setzen auf der Äquivalenzklassenbildung auf und versuchen mit Werten an den Grenzen der Äquivalenzklassen möglich Fehler aufzudecken. Für den Zahlenparameter wären dies z.B. die Testwerte: Int.Max Int.Max – 1 und Int.Max + 1.

Beim zustandsbezogenen Test sollen alle Kombinationen von Eingabeparameter, mit denen ein bestimmter Zustand erreicht werden kann, getestet werden. Als Beispiel kann der Bankautomat betrachtet werden. Z.B. sind alle Eingabeparameter-Kombinationen zu testen, die den Bankautomaten in den Zustand der Geldausgabe versetzen.

Bei der Ursache-Wirkungs-Graph-Analyse wird der Ursache-Wirkungs-Graph in eine Wahrheitstabelle überführt in der jede Spalte ein Testfall repräsentiert (Andreas & Tilo). Vorteil dieser Methode ist, dass Abhängigkeiten zwischen den Eingabeparametern und deren Auswirkung auf die Testausgaben berücksichtigt werden. Als Beispiel können Ursachen und Wirkungen bei einen Bankautomaten betrachtet werden.

Tabelle 40: Bankautomat Ursachen und Wirkungen

Ursache	**Wirkungen**
Bankkarte gültig	Karte einbehalten
PIN ist korrekt	Geld ausbezahlen

Daraus ergeben sich folgende Testfälle:

Tabelle 41: Testfälle aus Ursache-Wirkungs-Graph Analyse

Entscheidungstabelle		**Testfall 1**	**Testfall 2**	**Testfall 3**
Bedingungen	Bankkarte gültig	JA	NEIN	JA
	PIN ist korrekt	JA	JA	NEIN

Aktionen	Karte einbehalten	NEIN	JA	NEIN
	Geld ausbezahlten	JA	NEIN	NEIN

Beim Anwendungsbasierten Test werden aus UML Use-Case Diagrammen Testfälle abgeleitet. Für jede Aktivität soll ein Testfall generiert werden. Detailliertere Angaben zu den einzelnen Blackbox-Testverfahren sind aus (Andreas & Tilo, 2012) zu entnehmen.

8.1.3 Teststufen

Typischerweise kommen Blackbox-Tests in den Teststufen Komponenten- und Integrations-Tests zur Anwendung. Diese Teststufen folgen gleich nach der Programmierung und eignen sich deshalb besonders für das Testen von neu entwickelten oder korrigierten Komponenten. Damit wird auch vermieden, dass sich Entwicklungsfehler erst in höheren Teststufen (Systemtest oder Abnahmetest) bemerkbar machen. Fehler in der Spezifikation kann das Blackbox-Testing nicht aufdecken.

8.1.4 Kandidaten für Blackbox-Testing in Posity Design Studio

Geeignete Kandidaten für Blackbox-Testing im Posity Design Studio sind das Query- und das Modul-Diagramm, weil diese Diagramme sehr gut als Blackbox, unabhängig von anderen Applikationsteilen, betrachtet werden können. Beide Diagrammtypen verfügen über Eingabeparameter über die Werte von verschiedenen Testfällen für einen Blackbox-Test initiiert werden können. Das Modul-Diagramm besitzt auch Ausgabeparameter, die für eine Überprüfung von Testbedingungen genutzt werden können. Beim Query-Diagramm gibt es immer einen Ausgabeparameter, der einer Datenmenge entspricht. Diese kann verwendet werden, um Testbedingungen zu prüfen, indem z.B. diese gegen eine vorab gespeicherte Datenmenge geprüft wird.

8.2 Anforderungen

Diese Kapitel beschreibt die Anforderungen, die das Artefakt für das Blackbox-Testing für die LCNC Entwicklungsumgebung Posity zu erfüllen hat, damit Blackbox-Tests für im Posity Design Studio (PDS) entwickelte Komponenten möglich werden.

8.2.1 Strukturelle Anforderungen

- Anpassungen am fCode sind, wenn möglich, zu vermeiden.
- Es soll möglich sein Blackbox-Tests für das Modul- und Query-Diagramm durchzuführen.
- Für Blackbox-Tests im Query-Diagramm muss die Datenbank verfügbar sein.
- Das Posity-GUI Diagramm muss für Blackbox-Tests im Modul-Diagramm als Stellvertreter (Mock) funktionieren können.

8.2.2 Nicht Funktionale Anforderungen

- Wenn möglich sollen die Blackbox-Tests ohne das Öffnen vom Posity Design Studio (Headless mode) ausgeführt werden können. Dies ermöglicht später die Anbindung an eine Continuous-Integration[16] Pipeline (CI-Pipeline).
- Die Testausgaben sollen in einem standardisierten Format, z.B. (The NUnit Project, 2022) ausgegeben werden, sodass diese in Drittanbieter Testing-Tools (z.B. Jenkins[17]) eingebunden werden können.
- Testfälle sollen effizient und einfach mit der gewohnten Posity Design Studio Bedienung in Diagrammen erfasst werden können.
- Die Testausführung muss effizient erfolgen.

8.2.3 Funktionale Anforderungen

- Es sollen für das Query- und Modul-Diagramm Testfälle erfasst werden können.
- Die Testfälle sollen zu Test-Szenarien gruppiert werden können.
- Derselbe Testfall soll in mehreren Test-Szenarien eingebunden werden können.
- Das Posity-GUI soll für einen Modul-Diagramm Blackbox-Test gemockt werden können.
- Die Daten aus einem Query sollen für einen Modul-Diagramm Blackbox-Test gemockt (Query-Mock) werden können.
- Daten für einen Query-Mock sollen aufgezeichnet werden können.
- Daten für einen Query-Mock sollen aus einer Datei eingelesen werden können.

---

[16] Continuous-Integration (CI), https://www.atlassian.com/de/continuous-delivery/continuous-integration, (Abgerufen : 1905.2022)

[17] Jenkins, https://www.jenkins.io/, (Abgerufen: 19.05.2022)

- Es sollen Funktionsbausteine für Testbedingungen (Assertions) im Query- und im Modul-Diagramm entwickelt werden.
- Die Testausgaben sollen im Posity Design Studio visualisiert werden.
- Testresultate sollen exportiert werden können.

8.2.4 Abgrenzungen

- Die Ausführung der Testfälle soll erst im Posity Design Studio möglich sein und in einem zweiten Entwicklungsschritt durch eine CI-Pipeline getriggert werden können.
- Der fCode für das Test-Diagramm ist nicht Teil dieser Spezifikation.
- Die Test-Metriken werden später implementiert.

8.3 Erfolgskriterien

In diesem Kapitel sind die Kriterien gelistet, nach denen das Artefakt Blackbox-Testing in der Validierungsphase beurteilt wird. Nach der Auswertung der Erfolgskriterien können die eingangs aufgestellten Forschungsfragen beantwortet werden.

Tabelle 42: Erfolgskriterien Artefakt Konzept Blackbox-Testing

Kriterium Blackbox-Testing	**Beschreibung**
KBT1	Der Posity-Entwickler kann Testfälle für das Query- und das Modul-Diagramm erfassen.
KBT2	Aus den Testfällen lassen sich Test-Szenarien erstellen und Testfälle in mehreren Test-Szenarien verwenden.
KBT3	Mit Funktionsbausteinen im Query- und im Modul-Diagramm können Testbedingungen festgelegt werden.
KBT4	Das Posity-GUI kann für den Modul-Diagramm Blackbox-Text gemockt werden.
KBT5	Benutzeraktionen im Posity-GUI können für den Modul-Diagramm Blackbox-Test gemockt werden.

KBT6	Die Testfälle und Test-Szenarien können im Posity Design Studio ausgeführt werden.
KBT7	Die Testresultate werden im Posity Design Studio angezeigt.

8.4 Spezifikation

Dieses Kapitel definiert eine Spezifikation, um die Blackbox-Testing Anforderungen aus dem Kapitel 8.2 im Posity Design Studio umsetzen zu können.

8.4.1 Testfälle und Test-Szenarien definieren

Testfälle sollen im Posity Design Studio in einem Verzeichnis «Test» unterhalb eines Diagramm-Baumknoten abgelegt werden. Mit dem <…>-Knoten (Neues Diagramm erstellen) im Testverzeichnis kann ein neuer Testfall für dieses Diagramm erzeugt werden. Ein Test-Szenario ist eine spezielle Art von Testfall, der existierende Testfälle aus anderen Diagramm-Testfällen oder Testfälle innerhalb des eigenen Testverzeichnisses aufruft oder es kann Query- oder Modul-Global[18] aus mehreren Testfällen des jeweiligen Diagrammtypen ein Test-Szenario gebildet werden. Diese verlinkten Testfälle sollen als Kind-Knoten beim jeweiligen Testfall angezeigt werden. Die Abbildung 96 zeigt wie die Testfälle im Posity Design Studio im Feld `All Elements of Application` angeordnet und verknüpft sind.

[18] Query- oder Modul-Global: Man kann nicht nur Testfälle innerhalb einer Query zu einem Test-Szenario zusammenfassen, sondern kann aus allen Query-Testfällen wieder ein Query globales Test-Szenario (also ohne das es ein Query braucht) zusammenbauen kann. Dasselbe gilt für das Modul.

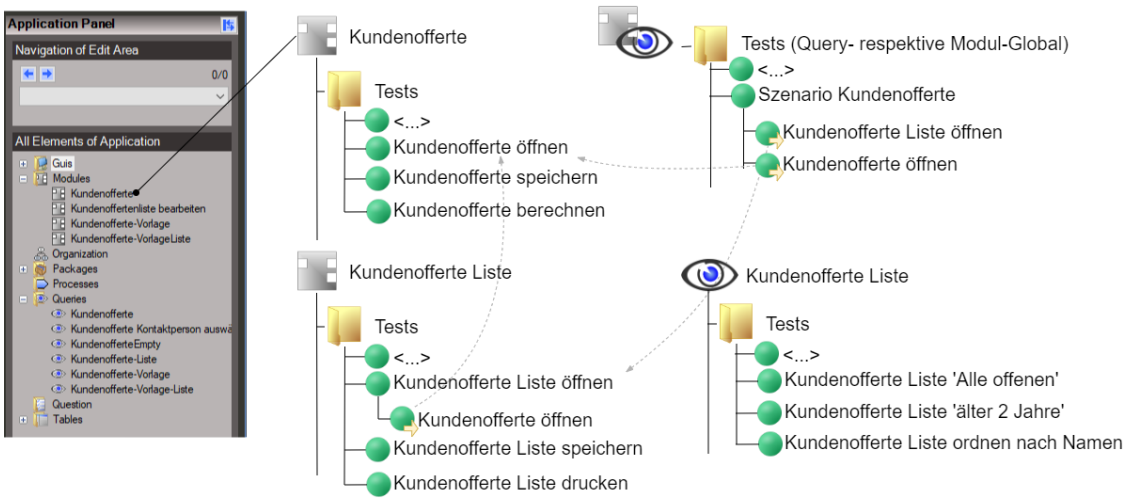


Abbildung 96: Testfälle und Test-Szenarien definieren für Query- und Modul-Diagramm (Quelle: Eigene Darstellung)

Aus dieser Anordnung folgt, dass die einzelnen Testfälle in Diagrammen abgebildet werden, wie auch z.B. das Modul, das GUI oder das Query. Hierfür wird ein neuer Diagrammtyp (z.B. Test-Diagramm) benötigt, indem ein Testfall für ein Modul oder Query definiert werden kann. Wie dies für die Query- und Modul-Testfälle im Detail funktioniert, wird in den Kapiteln 8.4.2 und 8.4.3 beschrieben.

8.4.2 Testfall Modul-Diagramm

Ähnlich wie aus einem Modul ein weiteres Modul (Sub-Modul) aufgerufen werden kann, soll im Test-Diagramm das zu testende Modul (Testobjekt) eingefügt und mit den nötigen Eingabeparametern versorgt werden. Kombinationen von verschiedenen Eingabeparameter-Werten, die z.B. bei der Äquivalenzklassenbildung oder der Grenzwertanalyse entstehen, können in weiteren Ebenen (Frames) oder in demselben Frame abgelegt werden. Die Abbildung 97 zeigt wie das Testobjekt `Kundenofferte speichern` (Event aus dem Modul) mit gemockten Daten vom Query Kundenofferte versorgt und ausgeführt und das Resultat mit dem `Assert TRUE` Logikbaustein geprüft wird, ob das Speichern erfolgreich war.

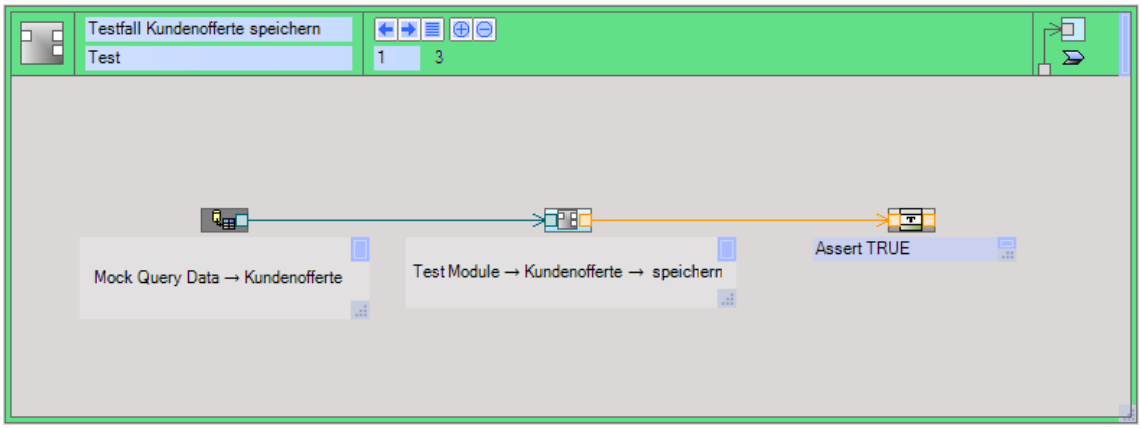


Abbildung 97: Test-Diagramm für Modul-Diagramm Testfall Kundenofferte speichern (Quelle: Eigene Darstellung)

8.4.3 Testfall Query-Diagramm

Querys werden normalerweise aus dem Modul-Diagramm mit Eingabeparametern aufgerufen. Dasselbe Prinzip kann für einen Blackbox-Test für Query-Diagramme angewendet werden, indem im Test-Diagramm das Query mit verschiedenen Eingabeparametern aufgerufen und das zurückgelieferte Resultat ebenfalls mit Assertion-Logikbausteinen überprüft wird.

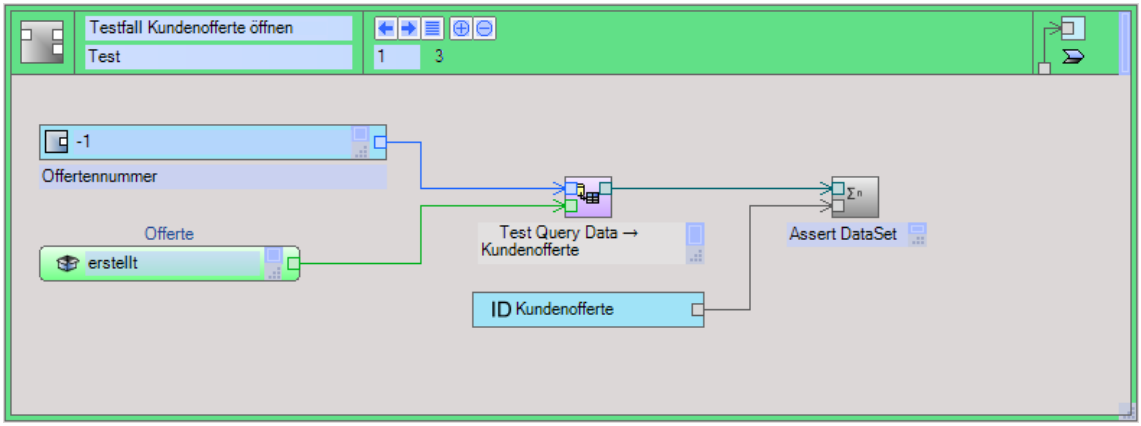


Abbildung 98: Test-Diagramm für Query-Diagramm Testfall Kundenofferte öffnen (Quelle: Eigene Darstellung)

8.4.4 Testbedingungen definieren

Entweder verläuft ein Testfall erfolgreich oder er schlägt fehl. Ohne Inhalt schlägt ein Testfall grundsätzlich fehl. Somit muss definiert werden, was erfolgreich für den Testfall

bedeutet. Dies wird mit Testbedingungen formuliert, die einen Ausgabeparameter vom Testobjekt auf einen bestimmten Wert prüfen. Damit die Testbedingungen nicht mit haufenweise If-Else-Kontrollstrukturen gebaut werden müssen, ist es üblich sogenannte Assert-Funktionen dem Entwickler zur Verfügung zu stellen. Zum Beispiel wird in der Abbildung 97 und Abbildung 98 die Assert-Funktion AssertTRUE verwendet, die ein Wert auf TRUE überprüft. Ist der Wert vom verknüpften Ausgabeparameter TRUE ist der Test erfolgreich, sonst schlägt der Testfall fehl. Zusätzlich zum Wert prüfen definieren die Assert-Funktionen das Resultat des Testfalls. Um das Test-Schreiben für den Entwickler möglichst einfach zu halten, braucht es eine gewisse Anzahl von Standard[19]- und Posity-Spezifischen[20]-Assert-Funktionen, um Testbedingungen zu formulieren. Zu Beginn reicht es aus die Standard-Assert-Funktionen zu implementieren und nach Wunsch diese, um die Posity spezifischen Assert-Funktionen, zu ergänzen.

8.4.5 Stellvertreter (Mocks) definieren

In der Abbildung 95 Übersicht Testrahmen ist ersichtlich, dass das Testobjekt möglichst isoliert getestet werden soll. Trotzdem benötigt ein Testobjekt für die Ausführung meistens Daten aus anderen Komponenten oder Schnittstellen. Um für das Testen möglichst unabhängig vom Zustand der umliegenden Komponente respektive Schnittstellen zu sein, werden diese jeweils gemockt, was so viel heisst wie es werden dem Testobjekt korrekte Daten geliefert, aber immer dieselben und angepasst an den Testfall. Für einen Query-Testfall heisst dies, dass die darunterliegende Datenbank, respektive die im Query-Diagramm verwendeten Tabellen gemockt werden müssen. Bei einem Modul-Testfall müssen das verwendete Posity-GUI und die Querys (inklusive die Posity-GUIs und die Querys der vom Modul verwendeten Sub-Module) gemockt werden. Die Kapitel 8.4.6, 8.4.7 und 8.4.8 beschreiben wie die Mocks im Posity Design Studio erstellt werden können.

8.4.6 Datenbank-Mock

Die Datenbanken auf dem Microsoft SQL-Server können gemockt werden, indem die Datenbank-Datei kopiert und die kopierte Datenbank-Datei wieder mit einem anderen

---

[19] C# Assert Klasse mit Standard Assert-Funktionen: https://docs.microsoft.com/en-us/dotnet/api/microsoft.visualstudio.testtools.unittesting.assert?view=visualstudiosdk-2022, (Abgerufen: 19.05.2022)

[20] Beispiele für Posity spezifische Assert Funktionen:

- AssertTableRowCount: Prüfen, ob eine Tabelle Zeilen enthält.
- AssertTableExists: Prüfen ob in einem Resultat einer Query eine bestimmte Tabelle existiert.

logischen Namen als neue Datenbank eingebunden wird. Damit kann nur mit Kopieren der Datenbank-Datei effizient ein neuer Datenbank-Mock erstellt werden, der wiederum bei einzelnen Testfällen verwendet werden kann.

8.4.7 Query-Mock

Mock-Daten für Querys manuell zu erfassen ist aufwendig. Am einfachsten werden die Mock-Daten direkt aus einer laufenden Anwendung heraus erzeugt. Im Modul-Diagramm gibt es einen graphischen Debugger, mit dem die Daten einer Query eingesehen werden können (siehe Abbildung 99). Dort soll eine zusätzliche Funktionalität eingebaut werden mit der diese als Mock-Daten für diese Query abgelegt werden können. Identisch wie im Kapitel 8.4.1 erwähnten Testverzeichnis, soll ein Mock-Verzeichnis pro Query angelegt werden, in welchem die Mock-Daten für eine Query abgelegt werden. Diese können dann im Test-Diagramm als Daten für den Eingabeparameter beim Testobjekt (siehe Funktionsbaustein Mock Query Data in Abbildung 97) verwendet werden. Welche Query-Mock-Daten für die Query-Logikbausteine innerhalb des Modul-Testobjekt verwendet werden, muss vorab im Testfall beim Modul-Testobjekt konfiguriert werden. Sind keine Query-Mock-Daten definiert, werden direkt die Daten aus den Querys verwendet. Query-Logikbausteine, die Daten in die Datenbank schreiben, werden ausgeführt, aber die Datenbank selbst bleibt unangetastet, ausser der Testfall ist so konfiguriert, dass er die Daten direkt aus der Datenbank beziehen soll.

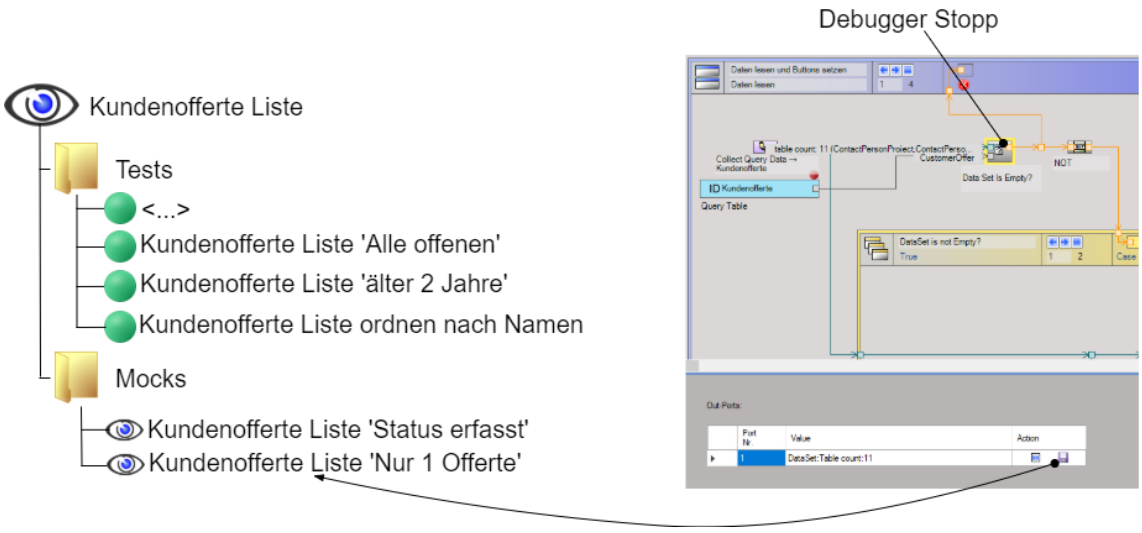


Abbildung 99: Query Mock-Daten mit Debugger speichern (Eigene Darstellung)

8.4.8 Posity-GUI-Mock

Zusammen mit einem Modul-Diagramm wird oft ein Posity-GUI verwendet, um Daten aus einer Query darzustellen. Beim Ausführen eines Testfalles soll das Posity-GUI nicht dargestellt werden, aber im Hintergrund im Speicher existieren, so dass damit die Modul-Komponenten, die mit einem Posity-GUI interagieren (z.B. `ShowGui`, `InsertGuiData`, `SetGuiVariable`, `GetGuiVariable`, usw.), funktionieren.

Benutzeraktionen auf dem Posity-GUI (z.B. Button-Klick) lösen im Modul-Diagramm einen Event aus. Beim Ausführen von einem Modul-Testfall mit Posity-GUI gibt es keine Benutzeraktionen, deshalb müssen diese beim Ausführen vom Testfall nicht speziell behandelt werden. Der Modul-Event, der normalerweise vom Posity-GUI mit einer Benutzeraktion ausgeführt wird, wird nun durch einen Testfall ausgeführt. Zum Beispiel gibt es einen Testfall Speichern, der im Modul-Diagramm den Event Speichern ausführt.

8.4.9 Testfälle ausführen

Ein einzelner Testfall kann im Test-Diagramm über die Toolbox Funktion `Testfall ausführen` ausgeführt werden. Mehrere Testfälle, respektive Test-Szenarien sollen mit einem Testtreiber (engl. test runner) ausgeführt werden. Welche Testfälle ausgeführt werden sollen, kann in einer eigenständigen Testtreiber Benutzeroberfläche konfiguriert werden. Alle Testfälle sind in einer Baumstruktur, analog wie in Abbildung 96, angeordnet und können mit einer Combobox für einen Testlauf aktiviert werden (siehe Abbildung 100).

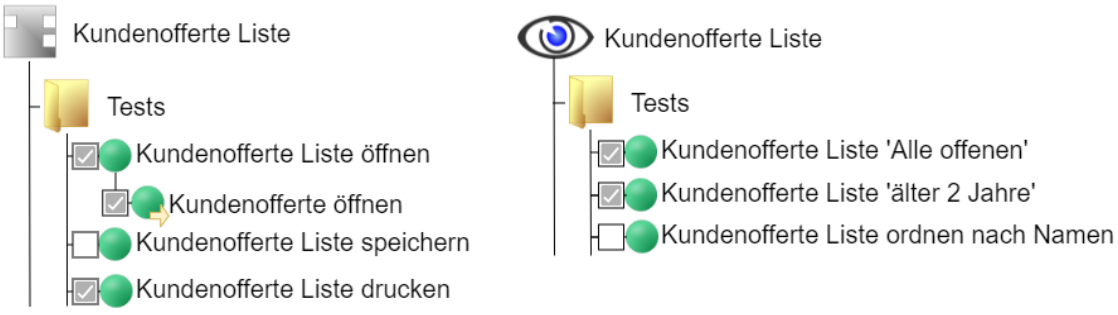


Abbildung 100: Testfall für die Ausführung selektieren

Über einen Start-Button wird ein Testlauf gestartet und kann mit einem Stopp-Button abgebrochen werden. Später ist denkbar, dass für jeden Testfall eine Anzahl an Ausführungen angegeben werden kann, um damit einfache Lasttests zu ermöglichen.

8.4.10 Test-Diagramm Ausführung

Wenn möglich soll die bereits existierende Posity Virtual Machine verwendet und ergänzt werden, um die Query- und Modul-Testobjekte auszuführen. Um ein Test-Diagramm, respektive einen Testfall vom Testtreiber auszuführen, muss für das Test-Diagramm ein eigener fCode spezifiziert werden. In diesem wird festgelegt, welche Eingabeparameter an das Testobjekt übergeben werden sollen und enthält die Assert-Funktionen. Ein ähnlichen fCode existiert bereits beim Prozess-Diagramm, in welchem die Eingabeparametern für den Aufruf von einem Modul definiert sind. Das Spezifizieren von fCode für das Test-Diagramm ist nicht Teil dieser Arbeit.

8.4.11 Testresultate visualisieren

Nach dem Ausführen der Testfälle durch den Testtreiber, sollen die Testergebnisse in tabellarischer Form in einer Testtreiber Benutzeroberfläche dargestellt werden. Falls ein Testfall fehlschlug, wird die Ursache in einem Testfall-Detail Feld beschrieben. Die Abbildung 101 zeigt einen möglichen Benutzeroberflächen-Mockup für eine Testtreiber Benutzeroberfläche.

Abbildung 101: Testresultate visualisieren – Mockup Benutzeroberfläche Testtreiber

8.5 Validierung

Dieses Kapitel beschreibt die Validierung des im Kapitel 8.4 spezifizierten Konzeptes für den Einsatz von Blackbox-Testing in der LCNC-Entwicklungsumgebung Posity Design Studio. Dieses Artefakt wurde validiert, indem in einer Delphi-Runde das Konzept mit

Expertenwissen von 3 Posity-Entwicklern geprüft und Verbesserungsvorschläge eingearbeitet wurden.

Der folgende Abschnitt beschreibt die Resultate aus der Delphi-Runde und die Verbesserungsvorschläge für die einzelnen Erfolgskriterien.

Tabelle 43: Auswertung der Erfolgskriterien Blackbox-Testing

Kriterium	**KBT1**	
Beschreibung	Der Posity-Entwickler kann Testfälle für das Query- und das Modul-Diagramm erfassen.	
Bewertung / Kommentare aus Delphi-Runde	Bewertung	Erfüllungsgrad
	Der Posity-Entwickler kann in einem eigenen Test-Diagramm Testfälle für Querys und Module definieren. Der Umbau der Baumstruktur in `All Elements of Application`, damit Tests und Mocks abgebildet werden können, benötigt ein grösseres Refactoring. Es muss ein neuer Diagrammtyp Test erstellt werden. Der fCode für das Test-Diagramm bedeutet auch ein Ausbau der Posity Virtual Machine, weil neue fCodes vom Test-Diagramm verarbeitet werden müssen. Das Prozess-Diagramm würde sich auch für Blackbox-Testing eignen. Im Moment sind die Prozesse respektive Prozess-Abläufe nicht sehr komplex und bestehen meistens nur aus einem Prozess, sodass Blackbox-Tests für Prozesse im Posity Design Studio wenig Sinn ergeben.	Ja

Kriterium	**KBT2**	
Beschreibung	Aus den Testfällen lassen sich Test-Szenarien erstellen und Testfälle in mehreren Test-Szenarien verwenden.	
Bewertung / Kommentare aus Delphi-Runde	Bewertung	Erfüllungsgrad
	In der Delphi-Runde wurde geäussert, dass Testfälle möglichst unabhängig sein müssen. Deshalb ist ein Verknüpfen von Testfällen untereinander nicht erwünscht. Es wurde definiert, dass ein Test-Diagramm einem Test-Szenario entspricht und die Testfälle in den Events vom Test-Diagramm abgebildet werden. In der Baumstruktur `All Elements of Application` werden demnach die Test-Szenarien dargestellt	Ja

Kriterium	**KBT3**	
Beschreibung	Mit Funktionsbausteinen im Query- und im Modul-Diagramm können Testbedingungen festgelegt werden.	
Bewertung / Kommentare aus Delphi-Runde	Bewertung	Erfüllungsgrad
	Es müssen neue Assert-Logikbausteine erstellt werden, die im Test-Diagramm verwendet werden können. Beim Test-Module Baustein muss festgelegt werden, welche Variablen aus dem Modul als Outport verfügbar sein sollen, um diese nicht extra mit einem Modul-Parameter nach Aussen führen zu müssen. Es muss auch eine Möglichkeit geben allenfalls Variablen, die in einem Modul-Event verwendet werden vor dem Aufruf des Testfalls setzen zu können. Zusätzlich zu den Modul-Parametern müssen auch die entsprechenden Modul-Event-Parameter als Inports bei der Test Module-Komponente angezeigt werden. Desweiteren muss beim Test-Module Baustein für die im Modul enthaltenen CollectQueryData-Bausteine ein Query-Mock festgelegt werden können.	Ja

Kriterium	**KBT4**	
Beschreibung	Das Posity-GUI kann für den Modul-Diagramm Blackbox-Text gemockt werden.	
Bewertung / Kommentare aus Delphi-Runde	Bewertung	Erfüllungsgrad
	Das Posity-GUI muss im Hintergrund ausgeführt werden, damit Aufrufe (`ShowGui, SetGuiVariable`, usw.) gegen das Posity-GUI funktionieren.	Ja

Kriterium	**KBT5**	
Beschreibung	Benutzeraktionen im Posity-GUI können für den Modul-Diagramm Blackbox-Test gemockt werden.	
Bewertung / Kommentare aus Delphi-Runde	Bewertung	Erfüllungsgrad
	Benutzeraktionen müssen nicht gemockt werden, weil diese nicht benötigt werden, respektive die ausgeführte Logik wird in einem Testfall abgebildet.	Nein

Kriterium	**KBT6**	
Beschreibung	Die Testfälle und Test-Szenarien können im Posity Design Studio ausgeführt werden.	
Bewertung / Kommentare aus Delphi-Runde	Bewertung	Erfüllungsgrad
	Testfälle können über den Testtreiber ausgeführt werden. Der Testtreiber soll auch in das Artefakt DevTools (siehe Kapitel 55) aufgenommen werden.	Ja

Kriterium	**KBT7**	
Beschreibung	Die Testresultate werden im Posity Design Studio angezeigt.	
Bewertung / Kommentare aus Delphi-Runde	Bewertung	Erfüllungsgrad
	Die Testresultate werden im Testtreiber dargestellt. Es wird gewünscht, dass bei den Assert-Bausteinen ein Kommentar angegeben werden kann, der beim Fehlschlagen vom Testfall im Testfall-Detail im Testtreiber ersichtlich wird.	Ja

Zusammenfassend sind von den **insgesamt 7 Erfolgskriterien 6** vollständig erfüllt, **1** teilweise erfüllt, und **0** Kriterien gelten als nicht validierbar.

9 Konzept für diversifizierende Testverfahren (Regressionstest)

Die diversifizierenden Tests zielen auf unterschiedliche Vergleichsmöglichkeiten verschiedener Testergebnisse ab. Die Idee der diversifizierenden Testtechniken ist, die oft aufwändige und manchmal kaum mögliche Bewertung der Korrektheit der Testergebnisse gegen die Spezifikation zu vermeiden. Der Vergleich wird nicht zwischen Testergebnis und Spezifikation gemacht, sondern zwischen zwei konkreten Testergebnissen. Ein wesentlicher Vorteil der diversifizierenden Testtechnik, im Gegensatz zu spezifikationsorientierten Testverfahren, ist die Automatisierung der Erfassen der Testfälle und des Vergleichs (Liggesmeyer, 2009, S. 180).

Es gibt verschiedene diversifizierenden Testtechniken, die unterschiedliche Ziele verfolgen. Der Back to Back-Test, dem eine n-Versionen-Programmierung zugrunde liegt, zielt beispielsweise auf eine weitgehende Testautomatisierung. Der Mutationen-Test, der keine Testtechnik im eigentlichen Sinne ist, bietet die Möglichkeit, die Leistungsfähigkeit von Testtechniken unter kontrollierten Bedingungen systematisch zu vergleichen indem auf Basis der Ausgangsversion der Software neue Versionen mit genau bekannten Fehlern erstellt wird. Der Regressionstest zielt auf die Erkennung von Folgefehlern nach Fehlerkorrekturen oder nach Erweiterung von Funktionalität in einer Versionsentwicklung. (Liggesmeyer, 2009, S. 180). Für unser Konzept liegt der Fokus auf der Testtechnik des Regressionstests, weil diese für das Posity-Framework als die wirtschaftlichste Technik der diversifizierenden Verfahren betrachtet wird und als Alternative und oder Ergänzung zu den in Kapitel 8 beschriebenen spezifikationsorientierten Testverfahren erachtet wird. Wirtschaftlich, weil durch die Möglichkeit der Aufzeichnung von Testfällen und dem automatisierten Vergleich zweier Testdurchläufe mit den bestehenden Diagrammen gearbeitet werden kann und es somit keine zusätzlichen Diagramme für die Erstellung und Validierung braucht. Im Gegensatz zu den spezifikationsorientierten Testtechniken, wo bereits ein bestimmtes Vorgehen bei der Programmierung beachtet werden muss, damit der Code Testbar ist, kann mit Regressionstests beispielweise mit der Technik von Aufzeichnung und Wiedergabe auch Legacy-Code getestet werden.

9.1 Anwendungskontext

Im Rahmen dieser Arbeit ist ein Konzept für die Implementierung von Regressionstests, als wichtiger Vertreter der diversifizierenden Testtechniken, in LCNC Applikationen am Beispiel des modellbasierten LCNC-Frameworks Posity entstanden.

9.1.1 Regressionstests

Der Regressionstest ist, nach (IEEE) Standard für Software Engineering Terminologie, ein erneuter Test, einer bereits getesteten Software nach deren Modifikation, mit dem Ziel nachzuweisen, dass Modifikationen keine unerwünschten Auswirkungen auf die Funktionalität besitzen. Denn durch Software-Modifikationen, wie beispielsweise Erweiterung von Funktionalität, Refactoring oder Fehlerkorrekturen, können neue Fehler in vorher fehlerfreie Teile eingefügt werden. Der Regressionstest verfolgt damit den Ansatz der Wiederholung von Testfällen und deckt dabei Seiteneffekte von Modifikationen auf. Eine Modifikation der Software erzeugt eine neue Version der Software. Der Regressionstest prüft das Verhalten einer aktuellen Software-Version gegen das Verhalten der Vorläuferversion. Ob alle Testfälle, oder nur eine bestimmte Teilmenge der Testfälle nach einer Modifikation erneut ausgeführt werden müssen, hängt von der Reichweite der Modifikation ab.

Die Beurteilung über die Korrektheit des Testfalls wird nicht anhand einer Spezifikation gemacht, sondern durch den Vergleich der neu erzeugten Ausgaben mit den Ausgaben der Vorläuferversion. Sind die Ausgaben identisch, so wird ein Testfall als erfolgreich angesehen. Regressionstests sind nützlich, wenn sich das Verhalten der Anwendung nicht oder nur minimal ändert, wie das beispielweise beim Refactoring der Fall ist, wo Code strukturell überarbeitet wird, sich an der Funktionalität jedoch nichts ändert. Tatsächlich wird sich das Verhalten einer Anwendung aber immer wieder ändern. Dann funktionieren Testfälle oftmals nicht mehr, schlagen Fehl und müssen angepasst werden.

Weil bei LCNC-Plattformen die Abstraktionsebene von Anweisungen sehr hoch ist, der Weg zur ausführbaren Anweisung für die Maschine (Assembler) entsprechend über viele Layer hinweg geht, können auf den verschiedensten Ebenen Fehlerursachen auftauchen. Der LCNC-Entwickler selbst, interessiert sich in erster Linie für den Code, den er selbst schreibt und nicht für die darunterliegenden Abhängigkeiten, in denen er keine Verantwortung trägt. Trotzdem kann ein Regressionstest ihn auch unterstützen, wenn er selbst keine Änderung an seiner Anwendung macht, aber beispielsweise eine neue Version des

Frameworks einführt oder eine Komponente des Frameworks durch eine andere mit gleicher Funktionalität ersetzt. Die im obigen Absatz erwähne Anfälligkeit von Regressionstests gegenüber von Änderungen an der Funktionalität führen dazu, dass Regressionstests vor allem dann von grossem Nutzen sind, wenn Anpassungen am Code gemacht werden, die an der Funktionalität nichts ändern. Für das Posity-Framework sind Regressionstests für folgende Fälle von Interesse:

- Komponenten Tests, wenn im darunterliegenden Framework etwas ändert.
- Refactoring von Diagrammen.
- Änderungen aufgrund von Performanceoptimierung.
- Erweiterung der Funktionalität, die den bestehenden Code nicht oder nur wenig betrifft.

9.1.2 Posity Diagramme und Regressionstest

Regressionstests, sollen im Rahmen für dieses Konzept einzig für den Diagrammtyp Modul angewandt werden können. Das Modul ist das zentrale Diagramm, welches die Businesslogik enthält und gehört zu den wichtigsten Diagrammtypen, für welche Regressionstests erstellt werden können müssen. Das Query Diagramm würde sich ebenfalls anerbieten, weil damit die Korrektheit von Datenbankabfragen überprüft werden kann. Das Konzept für Regressionstests im Query-Diagramm wird dem Konzept für das Modul-Diagramm sehr ähnlich sein und kann deswegen vernachlässigt werden. Das GUI-Diagramm wird in diesem Konzept nicht miteinbezogen, weil der Vergleich zweier Testresultate (Benutzeroberflächen) mit erheblich höherem Aufwand verbunden ist als dem Vergleich von zwei Testergebnissen aus dem Modul- oder Query-Diagramm, der Nutzen aber als vergleichsweise klein angesehen wird.

Das Ziel ist, die Regressionstests der Diagramme sowohl automatisiert, durch Aufzeichnung, zu erstellen als auch automatisiert auszuführen. Dafür wird dem wesentlichen Prinzip des Regressionstests gefolgt. Abbildung 102 zeigt dieses Prinzip auf.

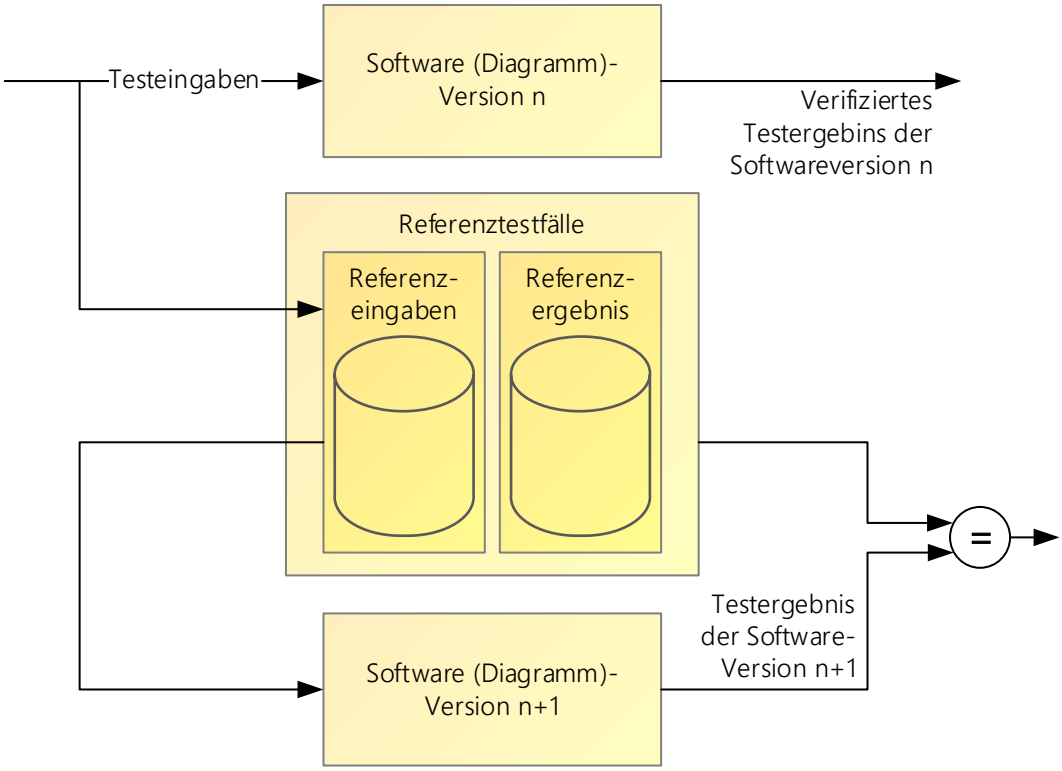


Abbildung 102: Prinzip des Regressionstest (Quelle: Eigene Darstellung angelehnt an (Liggesmeyer, 2009, S. 194))

9.2 Anforderungen

Dieses Kapitel beschreibt die Anforderungen, die das Artefakt für das Regressions-Testing für die LCNC Entwicklungsumgebung Posity zu erfüllen hat, damit Applikationen, welche mit dem Posity Design Studio (PDS) entwickelt werden durch Regressionstests getestet werden können. Als Artefakt soll ein Konzept entstehen, es wird kein Prototyp implementiert.

9.2.1 Strukturelle Anforderungen

- Anpassungen am fCode sind, wenn möglich zu vermeiden.
- Produktiven Code und Test-Code möglichst voneinander trennen, der Test-Code soll möglichst nicht in den produktiven Code eingebettet werden.
- Regressionstests für das Modul- und Query-Diagramm ermöglichen.

9.2.2 Nicht Funktionale Anforderungen

- Einhalten der FIRST Prinzipien nach (Ottinger & Langr, 2012) (Fast, Isolated, Repeatable, Self-verifiying, Timely)

- Wenn möglich sollen die Regressions-Tests ohne das Öffnen vom Posity Design Studio (Headless mode) ausgeführt werden können. Dies ermöglicht später die Anbindung an eine Continuous-Integration[21] Pipeline (CI-Pipeline).
- Die Testausgaben sollen in einen standardisierten Format wie beispielsweise (The NUnit Project, 2022) ausgegeben werden, sodass diese in Drittanbieter Testing-Tools (z.B. Jenkins[22]) eingebunden werden können.
- Testfälle sollen effizient und einfach mit der gewohnten Posity Design Studio Bedienung erfasst werden können.

9.2.3 Funktionale Anforderungen

- Es sollen für das Modul-Diagramm Testfälle erfasst werden können.
- Die Testfälle sollen automatisiert durch Aufzeichnung erstellt werden können.
- Das Posity-GUI soll für einen Modul-Diagramm gemockt werden können.
- Funktionsbausteine aus dem Modul sollen, bei Bedarf gemockt werden (beispielsweise datums-/zeitabhängige Komponenten).
- Die Testausgaben sollen im Posity Design Studio visualisiert werden.
- Testresultate sollen exportiert werden können.

9.2.4 Abgrenzungen

- Die Ausführung der Testfälle soll zuächst im Posity Design Studio möglich sein und in einem zweiten Entwicklungsschritt durch eine CI-Pipeline ausgelöst werden können.
- Es wird als Artefakt «nur» das Konzept erstellt, kein Programmcode.

9.3 Erfolgskriterien

In diesem Kapitel sind die Kriterien gelistet, nach denen das Konzept für die Regressionstest in der Validierungsphase beurteilt wird. Die Auswertung der Erfolgskriterien wird für die Beantwortung der Forschungsfragen verwendet.

[21] Continuous-Integration (CI), https://www.atlassian.com/de/continuous-delivery/continuous-integration, (Abgerufen : 19.05.2022)

[22] Jenkins, https://www.jenkins.io/, (Abgerufen: 19.05.2022)

Tabelle 44: Erfolgskriterien Konzept Regressions-Testing

Kriterium Regressions-Testing	Beschreibung
RT1	Testfälle können durch den Posity-Entwickler automatisiert für das Modul-Diagramm erfasst werden.
RT2	Testfälle können automatisiert ausgeführt werden.
RT3	Wo nötig können Abhängigkeiten durch Mock-Objekte simuliert werden.
RT4	Die FIRST Prinzipien für Tests werden eingehalten.
RT5	Die Testresultate sind selbstevaluierend und werden im Posity Design Studio angezeigt.

9.4 Spezifikation

Dieses Kapitel spezifiziert den Anforderungskatalog aus Kapitel 9.2 für das Konzept des Artefakts zur Implementation von Regressionstest. Dabei werden die Elemente definiert und spezifiziert, welche es für einen Regressionstest in der modellbasierten LCNC-Plattform Posity braucht. Die Spezifikation führt die Erstellung und das Ausführen von Testfällen aus und spezifiziert Darstellung und Validierung der Testresultate.

Für die Erstellung der Testfälle, werden für alle aufgerufenen Programmteile (Modul-Komponenten) Testdaten aufgezeichnet. Bei der Aufzeichnung werden je nach Konfiguration Eingaben, als auch die daraus entstandenen Ausgaben gespeichert. Eingaben, werden dann gespeichert, wenn diese für den Testfall gemockt werden sollen und Ausgaben werden grundsätzlich alle gespeichert, ausser die Testpunkte, werden explizit definiert. Es entsteht pro ausgeführtes Modul ein Testfall, welcher alle Testdaten für die ausgeführten Komponenten enthält. Ein solcher Testfall kann anschliessend eigenständig ausgeführt werden. Pro Aufzeichnung können, abhängig von den ausgeführten Abläufen, potenziell sehr viele Testfälle generiert werden. Bei der Ausführung des Testfalls, wird das mit dem Testfall assoziierte Modul ausgeführt. Dabei werden die Komponenten des Moduls entweder mit den Eingabedaten aus den aufgezeichneten Testdaten oder aktuell be-

rechneten Werten befüllt und die Testergebnisse mit den aufgezeichneten Testergebnissen verglichen. Weichen diese Ergebnisse voneinander ab, gilt der Testfall als fehlgeschlagen. Mehrere Testfälle werden in einem Testszenario zusammengefasst. Das Testszenario ist das Ergebnis eines Aufzeichnungsvorgangs in der Laufzeitumgebung. Die zum Vergleich beigezogenen Testergebnisse können so eingeschränkt werden, dass nur ein Teil der Testdaten, aus dem Referenz-Testergebnis mit dem Ergebnis der aktuellen Ausführung, verwendet wird. Indem der Vergleich auf einen Bereich von Resultaten reduziert wird, sollen die Testfälle gegenüber Veränderungen robuster werden.

9.4.1 Aufzeichnen und von Testfällen im Posity Design Studio

Die Testfälle werden durch die Technik der Aufzeichnung automatisch generiert und können in einem späteren Schritt (siehe Kapitel 9.4.2 und 9.4.3) ebenfalls automatisiert ausgeführt und die Testergebnisse validiert werden. Das Aufzeichnen des Testfalles erfolgt durch den Entwickler durch manuelle Bedienung der Applikation. Diese Aufzeichnung stellt nach Abbildung 102 das Referenzergebnis dar. Bei einer Aufzeichnung werden per Default alle Testdaten gespeichert, sofern der Testfallersteller keine Einschränkung durch Setzen von expliziten Testpunkten (grüne Punkte) vornimmt. Werden alle Testdaten gespeichert, kann dies zu grossen Datenmengen führen und die Performance der Testfälle potenziell vermindern. Eingangsdaten können bei Bedarf gemockt werden, das bedeutet, dass diese bei der Aufzeichnung als Testdaten gespeichert und bei der Ausführung des Testfalls wiederverwendet werden. Wie bereits in früheren Kapiteln (7.1.1.5) beschrieben, haben die Modul-Komponenten (Funktionen) Eingangs- und Ausgangsports, wobei die Daten der Ports in einem Register gespeichert werden. Bei der Aufzeichnung werden von allen ausgeführten Komponenten die Registerinhalte, der für den Testfall relevanten Eingangs- und Ausgangsports, in dem Posity eigenen fCode Format gespeichert. Als relevant gelten diejenigen Ausgangsports, welche für die Validierung der Testergebnisse verglichen werden sollen (grüne Markierung oder alle) und diejenigen Ausgangsports einer Komponente, welche für die Testausführung dieselben Daten wie im Referenzfall verwenden (Mock) sollen. Abbildung 103 zeigt ein Beispiel von einer Variable-Komponente, welche für den Testfall gemockt werden soll und einen aktiven Testpunkt bei der Komponente Addition. In diesem Fall bedeutet dies für die Aufzeichnung, dass die Daten vom Register-Ausgangsport der Variable `Cost` in den Testdaten gespeichert wird, um bei der späteren Ausführung des Testfalls denselben Wert wiederzuverwenden und das für

die Validierung des Testergebnisses nur die Resultate der Komponente Addition verglichen werden. Alle anderen Zwischenresultate, werden in der Validierung nicht verglichen.

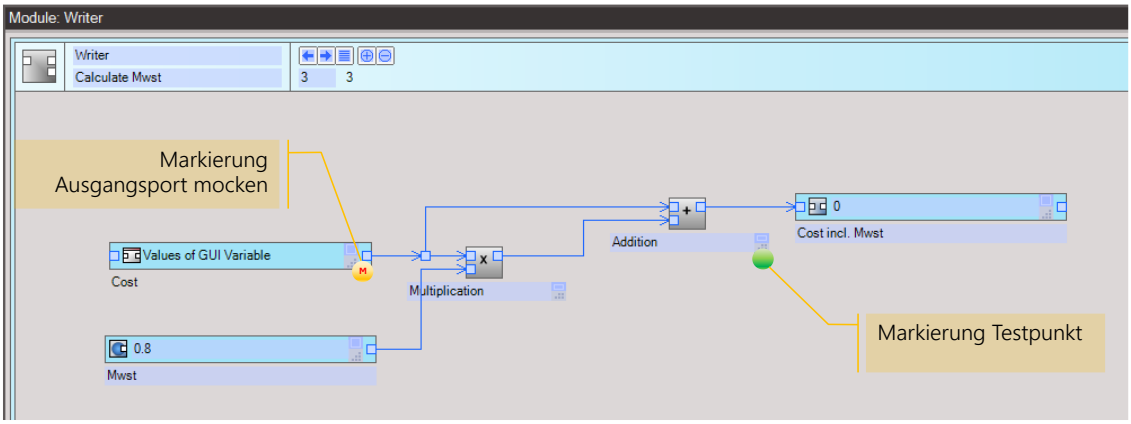


Abbildung 103: Modul-Diagramm Markierung von Testpunkten und Mock-Objekten (Quelle: Eigene Darstellung)

Das folgende Implementationsdetail der Posity Virtual Machine (PVM), Interpreter der Modul-Diagramme (siehe 6.1.2.2), ist für das Speichern der Testdaten relevant. Da es für jeden verbundenen Eingangsport immer einen entsprechenden Ausgangsport gibt, werden die Daten in der PVM nicht zweimal im Register abgelegt, sondern es genügt, wenn jeweils die Ausgangsports einen Registerplatz in der PVM belegen und die Eingangsports darauf referenzieren. Deshalb werden bei der Aufzeichnung der Testfälle jeweils nur die Registerinhalte der Ausgangsports gespeichert.

Ist die Vorbereitung (setzen von Mock- und Testpunkt-Markierungen) für die Aufzeichnung des Testfalls erfolgt, kann eine Aufzeichnung gestartet werden. Für das Starten einer Aufzeichnung soll nach dem Mockup in Abbildung 104 ein entsprechender Record-Button (grün) im Posity Design Studio implementiert werden, welcher die Posity-Runtime im Modus Record startet.

Abbildung 104: Mockup Starten einer Testaufzeichnung (Quelle: Eigene Darstellung)

9.4.2 Ausführen von Testfällen im Posity Design Studio

Für das Ausführen von Testfällen soll ein Testtreiber, analog wie im Kapitel 8.4.11 der spezifikationsorientierten Verfahren, verwendet werden. Im Testtreiber können einzelne Testfälle oder Testszenarien selektiert werden, welche ausgeführt werden sollen. Abbildung 105 zeigt ein Mockup für eine mögliche Darstellung der Testszenarien und deren Testfälle, sowie den Ergebnissen im Testtreiber.

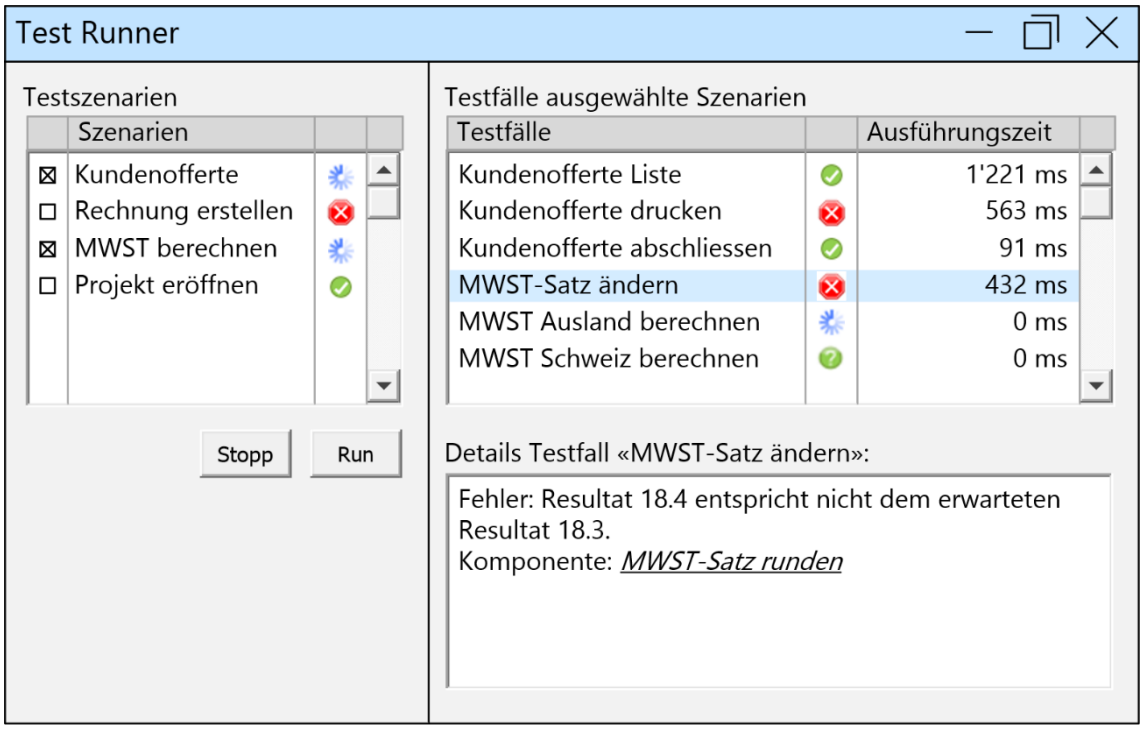


Abbildung 105: Mockup Visualisierung Testtreiber (Quelle: Eigene Darstellung)

Beim Ausführen wird der aktuelle Code ausgeführt und bei den Testpunkten mit den Daten des aufgezeichneten Testfalls verglichen und falls abweichend schlägt der Testfall fehl.

Tabelle 45: Legende Statussymbole Testtreiber

Symbol	Bedeutung
	Test noch nicht ausgeführt. Zustand undefiniert.
	Test wurde ausgeführt. Testergebnis ist positiv, also sind die Resultate von Testreferenz und aktueller Ausführung gleich.
	Test wurde nicht erfolgreich durchgeführt. In der Detailbeschreibung des Testergebnisses wird eine Fehlermeldung angezeigt.
	Test wird im Moment ausgeführt.

Zudem soll die Möglichkeit bestehen, vor dem Ausführen der Tests, eine zusätzliche Einschränkung der Testpunkte durch inaktiv setzen des grünen Testpunktes zu definieren (siehe Abbildung 106).

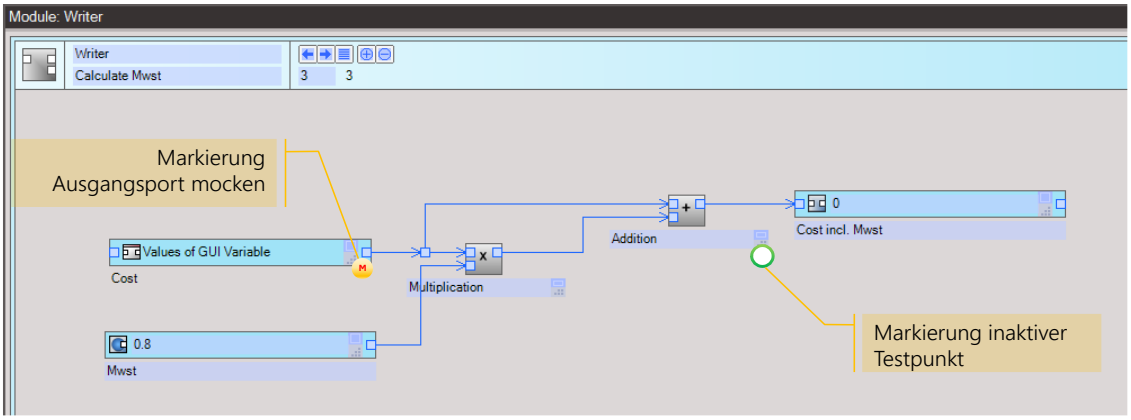


Abbildung 106: Modul-Diagramm Testfall mit deaktiviertem Testpunkt
(Quelle: Eigene Darstellung)

Dies soll ermöglichen, dass ein allenfalls nicht mehr benötigter Testpunkt «entfernt» werden kann, aber bei Bedarf zu einem späteren Zeitpunkt wieder aktiviert werden kann. Zum Bearbeiten der Testfälle muss das Diagramm in der Posity IDE in einem entsprechenden Modus geladen werden, in welchem das Diagramm selbst nicht bearbeitet werden kann, sondern nur Testpunkte aktiviert oder deaktiviert werden können. Möglich ist

auch eine Konfiguration analog dem Aktivieren/Deaktivieren von Haltepunkten in modernen IDEs, wie beispielsweise Visual Studio, zu implementieren (siehe Abbildung 107).

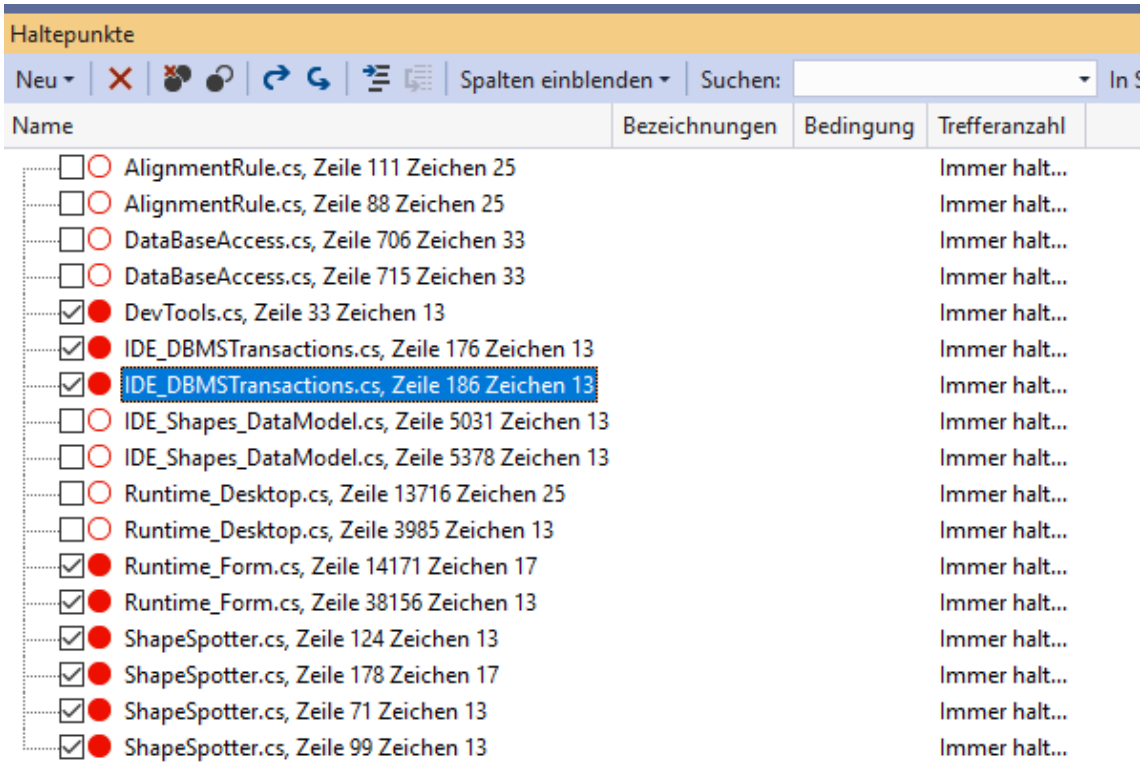


Abbildung 107: Darstellung Haltepunkte in Visual Studio 2019
(Quelle: Screen-Print von Visual Studio)

9.4.3 Testergebnisse und Validierung

Nach dem Ausführen der Testfälle durch den Testtreiber, sollen die Testergebnisse in tabellarischer Form in einer Testtreiber Benutzeroberfläche dargestellt werden. Dabei ist es wichtig, dass die Validierung der Tests ein binäres Resultat liefert (OK oder Not OK). Es soll keine manuelle Inspektion von Test-Logs nötig sein. Falls ein Testfall fehlschlägt, wird die Ursache in einem Log-Feld beschrieben, damit die Ursache eruiert werden kann. Die Abbildung 105 aus dem vorigen Kapitel zeigt ein Mockup einer möglichen Benutzeroberfläche für den Testtreiber.

9.5 Validierung

Dieses Kapitel beschreibt die Validierung des im Kapitel 9.4 spezifizierten Konzeptes für den Einsatz von Regressionstests in der LCNC-Entwicklungsumgebung Posity Design Studio. Das Konzept wurde in einer Delphi-Runde mit dem Expertenwissen von 3 Posity-Entwicklern validiert.

Für die Bewertung werden die Erfolgskriterien, pro Validierungsfall, einem Erfüllungsgrad zugeordnet. Tabelle 46 enthält die Legende zu den verschiedenen Varianten vom Erfüllungsgrad.

Tabelle 46: Legende Erfüllungsgrad für Bewertung

Symbol Erfüllungsgrad	**Bedeutung**
Ja	Kriterium ist vollständig erfüllt
Nein	Kriterium ist nicht erfüllt
~	Kriterium ist teilweise erfüllt

Tabelle 47 beschreibt die Resultate aus der Delphi-Runde für die einzelnen Erfolgskriterien.

Tabelle 47: Erfolgskriterien validiert durch Delphi-Runde

Kriterium	**RT1**	
Beschreibung	Testfälle können durch den Posity-Entwickler automatisiert für das Modul-Diagramm erfasst werden.	
Bewertung / Kommentare aus Delphi-Runde	Bewertung	Erfüllungsgrad
	Der Posity-Entwickler kann Testfälle durch Aufzeichnung für Modul-Diagramme erstellen. Die Visualisierung der Testfallaufzeichnung muss bezüglich Bedienbarkeit noch einmal kritisch überprüft werden, so dass die Übersicht über bestehende Testfälle und das Aufzeichnen und Löschen von neuen Testfällen am gleichen Ort zu finden sind. Die Testfälle der Regressionstests sollen aber nicht, wie dies für die Blackbox-Tests gemacht wird, als eigenes Diagramm in der Baumstruktur abgebildet werden, weil für die Regressionstests kein neuer Diagrammtyp entsteht. Die Darstellung könne in einem separaten Fenster, analog zum Testtreiber gemacht werden.	Ja
Kriterium	**RT2**	
Beschreibung	Testfälle können automatisiert ausgeführt werden.	
Bewertung / Kommentare aus Delphi-Runde	Bewertung	Erfüllungsgrad
	Die Testszenarien und oder einzelne Testfälle können über den Testreiber gestartet werden. Das Kriterium der automatisierten Ausführung der Testfälle ist	~

	damit erfüllt. Im Konzept noch nicht miteinbezogen, ist das automatisierte Starten von Testfällen von ausserhalb des Posity Design Studios.	
Kriterium	**RT3**	
Beschreibung	Wo nötig können Abhängigkeiten durch Mock-Objekte simuliert werden.	
Bewertung / Kommentare aus Delphi-Runde	Bewertung	Erfüllungsgrad
	Potenziell können alle Komponenten im Modul-Diagramm als Mock markiert werden, dann werden für diese Komponente die Daten der Testaufzeichnung verwendet. In der Delphi-Runde wurde die Äusserung gemacht, dass möglicherweise bestimmte Komponenten, wie bspw. GUIs und Datenbankinteraktionen grundsätzlich immer gemockt werden müssten, weil diese ansonsten die Testausführung behindern. Wenn Datenbankinteraktionen nicht gemockt werden, muss es möglich sein, den Zustand der Datenbank zurückzusetzen oder beispielweise eine Testdatenbank zu konfigurieren, welche für die Testausführung verwendet wird und nach Ausführung zurückgesetzt wird.	Ja
Kriterium	**RT4**	
Beschreibung	Die FIRST Prinzipien für Tests werden eingehalten	
	Bewertung	Erfüllungsgrad

Bewertung / Kommentare aus Delphi-Runde	FAST: Die Delphi-Runde befürchtet, dass das Fast-Prinzip bei grossen Testszenarien oder Testfällen nicht erfüllt werden kann, wenn alle Komponenten-Daten im Vergleich des Testergebnisses validiert werden müssen, es deshalb für eine performante Testausführung unumgänglich ist, mit ausgewählten Testpunkten zu arbeiten. INDEPENDENT: Die einzelnen Testszenarien Testfälle hängen nicht voneinander ab. Jeder Testfall kann unabhängig von den anderen und in beliebiger Reihenfolge durchgeführt werden. Das Prinzip wird als erfüllt betrachtet. REPEATABLE: Die Forderung nach Wiederholbarkeit der Tests ist teilweise gegeben. So können die Tests, sofern Datenbankabfragen, Zeitabhängige Komponenten oder Benutzerinteraktionen gemockt werden, mehrfach ausgeführt werden und die Ausführung führt zum selben Ergebnis. Aufgezeichnete Tests können jedoch nicht in jeder Umgebung (beispielsweise Entwicklungsumgebung, Build-System, Testumgebung, etc.) ausgeführt werden, sondern nur in der Umgebung, in welcher der Test aufgezeichnet wurde.	~

	SELF-VALIDATING: Die Tests liefern ein binäres Resultat, es ist keine manuelle Inspektion erforderlich. Wird als erfüllt betrachtet. TIMELY: Ein Regressionstest kann erstellt werden, sobald ein Modul ausgeführt werden kann. Das bedingt, dass für das Modul bereits ein entsprechender Prozess erstellt worden ist, an den das Modul gebunden wird. Regressionstests werden vom Zeitpunkt her gesehen später erstellt werden können, wie beispielsweise Unit-Tests, welche erstellt werden können, sobald ein Programmteil implementiert worden ist. Das sehen die Experten jedoch als zeitnah genug an.	
Kriterium	**RT5**	
Beschreibung	Die Testresultate sind selbstevaluierend und werden im Posity Design Studio angezeigt.	
Bewertung / Kommentare aus Delphi-Runde	Bewertung	Erfüllungsgrad
	Die Tests liefern ein binäres Resultat, es ist keine manuelle Inspektion erforderlich. Wird als erfüllt betrachtet. Das Kriterium wurde bereits als Teil von RT4 validiert. Ein Vorschlag für die Darstellung der Testresultate im Posity Design Studio wurde im Konzept beachtet und wird als erfüllt betrachtet.	Ja

Zusammenfassend sind von den **insgesamt 5 Erfolgskriterien** 3 vollständig erfüllt, **2** teilweise erfüllt, und **0** Kriterien gelten als nicht validierbar.

10 Schlussfolgerungen und Handlungsempfehlungen

Dieses Kapitel wird in zwei Unterkapitel aufgegliedert, damit die beiden Autoren individuell auf die von Ihnen bearbeiteten Forschungsfragen eingehen können. Es ist daher möglich, dass sich gewisse Inhalte überschneiden. Resultate, sowie Vorgehen und Methode, werden selbstkritisch hinterfragt, damit Limitierungen und deren Einfluss auf die Rigorosität klargestellt ist. Aus den Resultaten werden Schlussfolgerungen gezogen und aufgezeigt, welche Forschungsfragen geklärt sind und welche weitere Forschungsarbeit der Klärung bedürfen.

10.1 Qualitätssicherungsmassnahmen Performance-Analyse und spezifikationsorientiertes Testverfahren

10.1.1 Performance-Analyse

Das Thema Performance dürfte in der Software-Branche seit der ersten Geburtsstunde ein Thema sein, weil die gewählten Ansätze und Algorithmen zur Problemlösung, implementiert und getestet in der Entwicklungsumgebung, die zur Verfügung stehenden Systemressourcen in der realen Anwendung, mit mehr Daten oder mehr Anwendern, schliesslich übersteigen. Dies führt automatisch, ab einem gewissen Zeitpunkt, zum Performance-Engpass. Zu wenig Speicher, zu wenig CPU-Cores, zu wenig Bandbreiten und früher vielleicht zu wenig Röhren, um die gewünschten Datenmengen in geforderter Zeit abzuarbeiten.

Der Autor würde zudem behaupten, es entstehen öfters Performance-Probleme in höheren, abstrakteren Programmiersprachen, weil sich die Entwickler im Vorab die Limitationen ihrer Umgebung nicht aneignen müssen, oder diese gar wegen den Abstraktionslevels, die zwischen der Hardware und dem Programmcode liegen, versteckt bleiben. Ein besonders hoher Abstraktionslevel findet sich in den LCNC-Entwicklungsumgebungen, wodurch zum Beispiel versteckt ist, welcher konkrete Ressourcenverbrauch in welchem LCNC-Programmierbaustein steckt. Eine LCNC-Entwicklungsumgebung soll durch weniger Code die Codierarbeit abnehmen und nicht das Denken, was aber wegen den einfachen Bedienmöglichkeiten oft umgekehrt verstanden wird. So ist anzunehmen, dass häufig wenig performante Systeme entstehen, wenn diese ohne Konzept nur «zusammengeklickt» werden.

Gerade deshalb ist es für den LCNC-Entwickler wichtig Werkzeuge zu besitzen, mit denen er sich, wenn nötig, aus einem Performance-Engpass herausmanövrieren kann. Ohne diese Werkzeuge gibt es auf solch hohem Abstraktionslevel kaum Chancen das Performance-Problem zu identifizieren, geschweige denn zu lösen. Beim Lösen des Performance-Problems benötigt der LCNC-Entwickler hilfreiche Informationen aus tiefer liegen Applikationsschichten, in für ihn zugänglicher Form.

Der in dieser Forschungsarbeit verwendete Ansatz, die aus der klassisch textuellen Programmierung bekannten Performance-Metriken für LCNC-Entwicklungsumgebungen zu übernehmen, hat sich bereits bewährt. Zum Beispiel fliessen die ersten Performance-Messdaten aus der Validierungsphase direkt in den Entwicklungsprozess bei der Firma Posity AG mit ein und können daher als sehr praxisrelevant betrachtet werden. Ob Anpassungen am System zu einer Performance-Steigerung führen, kann aus der Performance-Analyse nicht direkt abgeleitet werden. Dies kann erst nach Änderungen an der Implementation und dem erneuten Messen abschliessend geklärt werden. Mit der Validierung konnte auch gezeigt werden, dass sich die aus der klassischen, textuellen Programmierung bekannten Performance-Metriken (z.B. Ausführungszeit) für bestimmte Teile in einer LCNC-Entwicklungsumgebung anwenden lassen und so den LCNC-Entwickler bei seiner Arbeit unterstützen können. Die eingangs gestellte Forschungsfrage kann deshalb wie folgt beantwortet werden.

> *FF 1 (Matthias Christen): Welche Performance-Metriken und Messmethoden aus der klassischen, textuellen Programmierung können auf Low-Code- und No-Code-Entwicklungswerkzeuge angewandt und mit diesen eine Performance-Steigerung erreicht werden?*

Ja es können die in der klassischen, textuellen Programmierung Performance-Metriken und Messmethoden auf LCNC-Entwicklungsumgebungen angewendet werden. Weil die reine Anwendung von den Performance-Messmethoden das Performance-Problem nicht löst, sondern nur identifiziert, kann nicht abschliessend geklärt werden ob damit eine Performance-Steigerung möglich ist.

Im entwickelten Artefakt wurde die Performance-Messung nicht direkt in der Programmiersprache der LCNC-Entwicklungsumgebung implementiert, sondern die Performance-Messungen konnten dank Zugang zum LCNC-Entwicklungsumgebungs-Quellcode, in der Programmiersprache, in der die LCNC-Entwicklungsumgebung geschrieben

ist, umgesetzt werden. Damit gab es keine Limitation bei der Entwicklung der Performance-Analysen und weil einen Abstraktionslevel tiefer eingestiegen werden konnte, auch keine Beschränkungen durch die LCNC-Programmiersprache. Dank dem Zugang zum Quellcode, gab es zu Beginn der Arbeit weniger Bedenken, ob es überhaupt möglich ist, Performance-Messdaten in der LCNC-Entwicklungsumgebung aufzuzeichnen. Dass sich die aufgezeichneten Messdaten aber sehr gut auf die LCNC-Applikationsbausteine abbilden liessen, war eher eine Überraschung. Denn wenn die in der LCNC-Entwicklungsumgebung definierte Applikationslogik z.B. direkt in Java- oder C#-Bytecode übersetzt würde, dürfte dies ein schwieriges Unterfangen werden.

In der, bei der Artefakt-Entwicklung eingesetzten, LCNC-Entwicklungsumgebung Posity Design Studio der Firma Posity AG wird die Applikationslogik in einen eigenen Zwischencode (Funktionaler Code: fCode) transformiert und durch eine virtuelle Maschine verarbeitet. Durch diese Architektur war die Aufzeichnung von Performance-Messdaten ebenfalls begünstigt, weil diese Messdaten direkt in der virtuellen Maschine erfasst werden können. Durch die Tatsache, dass ein fCode einem Logikbaustein entspricht, konnte sehr einfach ein direkter Bezug zwischen verarbeitetem Logikbaustein und dem in der LCNC-Entwicklungsumgebung hergestellt werden.

Einen Grund für den Einsatz einer LCNC-Entwicklungsumgebung ist der Wunsch, dass Entwickler mit weniger technischem Fachwissen eine Anwendung bauen können. Dies bedeutet aber auch, dass die Performance-Messdaten für die LCNC-Entwicklungsumgebung so aufbereitet werden, dass diese für den LCNC-Entwickler verständlich sind. Zum Beispiel wurde der Microsoft SQL-Server im Performance-Analyse Teilartefakt Datenbank-Abfragen verwendet. Der SQL-Server würde ein Vielfaches mehr an Performance-Messdaten liefern, die aber den LCNC-Entwickler mehr überfordern als helfen. Es muss also abgeschätzt werden, welche und wie allenfalls Performance-Messdaten aus einem tieferen Abstraktionslevel dem LCNC-Entwickler präsentiert werden. Mit der Benutzerschnittstelle vom Artefakt Performance-Analyse konnte auch gezeigt werden, dass sich die Steuerelemente (z.B. Call-Tree) von bekannten Performance-Messwerkzeugen (JetBrains, 2021a) auch für die Darstellung von Performance-Messdaten in LCNC-Entwicklungsumgebungen eignen. Damit gestaltet sich der Einstieg für Entwickler, die Performance-Analysen in einer LCNC-Entwicklungsumgebung durchführen und die Steuerelemente schon kennen, sehr einfach.

Obwohl die Erfolgskriterien für das Artefakt Performance-Analyse eher Produkt spezifisch festgelegt wurden, decken diese die generell benötigten Anforderungen (Messen und Darstellen der Messdaten) an ein Performance-Analyse-Werkzeug ab. Dadurch ist zu vermuten, dass es auch anderen LCNC-Entwicklungshersteller gelingen wird, die aus der klassisch textuellen Programmierung bekannten Performance-Metriken in ihren LCNC-Entwicklungsumgebungen anzubieten.

Die im Artefakt angewandten Performance-Messmethoden waren auf das Tracing (Kontinuierliches Aufzeichnen von Daten) beschränkt. Das Tracing bietet eine gute Übersicht über den Performancezustand einer Applikation über die Zeit. Ergänzend könnte die Sampling-Methode angewendet werden, mit der es einfacher ist, verschiedene Performancezustände in verschiedenen Zeitpunkten zu vergleichen. Des Weiteren waren die Performance-Messungen im Artefakt beschränkt auf die Ausführungszeiten. Ein aussagekräftigeres Bild über die gesamte Applikationsperformance ergibt sich erst, wenn auch der Speicherverbrauch und der Datendurchsatz zu den Umsystemen untersucht werden könnte.

Die Interpretation der Messwerte und die daraus folgenden Handlungen sind Sache des Entwicklers. Daher wäre das automatisierte Generieren von Handlungsempfehlungen für die Verbesserung der Performance aufgrund der Performance-Messdaten ein spannender Anknüpfungspunkt für weitere Forschung.

Liefert ein LCNC-Entwicklungsumgebungs-Hersteller keine Performance-Analyse-Werkzeuge, dann wäre es empfehlenswert, zumindest die Performance-Messung mit Messfunktionalitäten in der LCNC-Programmiersprache zu ermöglichen.

10.1.2 Spezifikationsorientierte Verfahren – Blackbox-Testing

Low-Code, No-Code und Low-Testing und demzufolge No-Testing? Nein, jede Software benötigt minimale Testaktivitäten vor der Auslieferung an den Anwender, auch wenn dies nur einmal Starten bedeutet - nur die Anwendung erstellen reicht nicht. Demnach sollte auch eine mit einer LCNC-Entwicklungsumgebung erstellte Anwendung je nach Komplexität entweder ein minimales Set an Tests umfassen oder zumindest zur Ausführung gebracht werden. In dieser Arbeit wurde nicht untersucht welche LCNC-Entwicklungsumgebungen Software-Testing unterstützen, sondern ob sich die aus der klassischen textuellen Softwareentwicklung bekannten dynamischen Software-Testverfahren, im speziellen die spezifikationsorientierten Testmethoden (z.B. Blackbox-Testing), in einer

LCNC-Entwicklungsumgebungen verwenden lassen. Dazu wurden eingangs folgende Forschungsfragen formuliert:

> *FF 3 (Gemeinsam): Eignen sich dynamische Software-Testverfahren für Anwendungen im Bereich Low-Code und No-Code und wie können diese eingesetzt werden?*
>
> *Teil FF 3.1 (Matthias Christen): Eigenen sich spezifikationsorientierte Testmethoden für das Testen von Anwendungen im Bereich Low-Code und No-Code und wie können diese eingesetzt werden?*

Weil diese spezifikationsorientierten Testverfahren sehr stark bei textbasierten Programmiersprachen eingesetzt werden, war zu Beginn überhaupt nicht klar, ob sich diese auf LCNC-Programmierung anwenden lassen. Zudem war es für die in diesem Artefakt untersuchte spezifikationsorientierte Testmethode Blackbox-Testing fraglich, ob überhaupt eine solche Abgrenzung, respektive eine solche Blackbox gefunden werden kann, die das zu testende Objekt von der gesamten Applikation für einen Test abgrenzt. Vor allem in objektorientierten Programmiersprachen sind diese Grenzen seit längerem bekannt und über die verschiedenen Sprachen an denselben Stellen gesetzt. In LCNC-Programmiersprachen ist dies sehr individuell, weil der Syntax der Sprache pro Hersteller einzigartig ist. In der Sprachsyntax der LCNC-Programmiersprache für das Posity Design Studio der Firma Posity AG konnte aber eine solche Grenze für eine Blackbox identifiziert werden. Damit war die Basis geschaffen, um die nach (Andreas & Tilo, 2012) geforderten Bestandteile eines Blackbox-Testing-Systems für das Posity Design Studio konzeptionell zu entwickeln. Beim weiteren Ausarbeiten des Konzepts wurde das Vorhaben, die klassischen Blackbox-Tests auf LCNC-Entwicklungsanwendungen anzuwenden, vermehrt bestätigt, weil sich z.B. die Erstellung von Testfällen oder das Definieren von Mocks sehr einfach mit der LCNC-Programmiersprache realisieren lassen.

Wenn kein No-Testing, vielleicht eher Low-Testing? Ja das Artefakt Blackbox-Testing hat gezeigt, dass sich spezifikationsorientierte Testmethoden auf einer höheren Abstraktionsschicht, respektive in einer LCNC-Programmiersprache, beschreiben lassen. Somit ist es möglich dynamische Testverfahren auch in LCNC-Entwicklungsumgebungen für das Software-Testing einzusetzen. Wird mit dem Prefix «Low» die Anzahl Tests assoziiert, die eine LCNC-Applikation für dieselbe Testabdeckung wie eine klassisch textuelle

erstellte Applikation benötigt, ist zu vermuten, dass, wenn die LCNC-Programmiersprache ähnliche Logikbausteine wie eine klassisch textuelle Programmiersprache (z.b. Funktionen mit Parametern, If-Else, Switch-Case, usw.) enthält, in der LCNC-Entwicklungsumgebung ähnlich viele Testfälle anfallen, wie in der klassisch textuellen Programmierung. Dies wird vermutet, weil zum Beispiel eine Summe-Funktion in der klassischen und in der LCNC-Programmierung die gleiche Anzahl an Parametern aufweist, für die Testfälle generiert werden müssen. Natürlich bleibt dies ein Vermutung, weil für eine präzise Aussage die Daten fehlen. Allenfalls ist dies ein Anknüpfungspunkt für weitere Forschung, weil wenn es weniger Testfälle in LCNC-Entwicklungsumgebungen bräuchte, wäre das ein weiteres Argument für den Einsatz von einer LCNC-Entwicklungsumgebung.

Ein weiteres zusätzliches Thema für Forschung im Bereich der spezifikationsorientierten Testmethoden könnte die automatisierte Testfall-Erstellung sein. Aus den Metadaten der LCNC-Entwicklungsumgebung könnten vielleicht direkt Testfälle abgeleitet werden und sogar die Testdaten für diese. Das wäre eine enorme Arbeitsersparnis und der Grund keine Tests, weil zu teuer, wäre hinfällig. Vielfach sind die produktiven Daten einer Applikation auch die besten Testdaten. Doch beim Verwenden der produktiven Daten als Testdaten wird oft Zeit verloren, weil die produktiven Daten meistens manuell aus der abgekapselten Produktivumgebung extrahiert und wieder mühsam in die Testumgebung integriert werden müssen. Wie nun diese produktiven Daten möglichst einfach und automatisiert als Testdaten bei den Testfällen eingesetzt werden können, könnte im Artefakt weiter ausgearbeitet werden. Des Weiteren wurde im Artefakt nur die Blackbox-Testing Testmethode berücksichtig, daher könnte es zielführend sein, vor einer Implementation erst weitere Testmethoden wie z.B. das Whitebox-Testing zu untersuchen.

10.2 Qualitätssicherungsmassnahmen Quellcode-Analyse und diversifizierende Testverfahren

Dass Qualitätssicherungsmassnahmen für Software in der Disziplin des Software-Engineerings, seit mindestens einem halben Jahrhundert, unumstritten eine wichtige Rolle spielen, legt O'Regan in seiner Arbeit über die historische Entwicklung der Computertechnik dar. Softwareinspektion und Testing nehmen beim Erstellen und Beurteilen von Software eine Schlüsselrolle ein. Während entsprechende Verfahren in der klassischen Softwareentwicklung weit etabliert sind, gibt es für die Programmierung mit Low-Code

und No-Code Aufholbedarf. Möglicherweise wurde die Thematisierung der Qualitätssicherung von in LCNC erstellten Anwendungen vernachlässigt, weil davon ausgegangen wird, dass es auf diesem Abstraktionsniveau weniger relevant ist und darunterliegende Schichten die Qualität mit entsprechenden Massnahmen sicherstellen müssten. Aus Sicht der Autorin ist diese Annahme jedoch falsch. Sie geht davon aus, dass auf jeder Abstraktionsstufe qualitative Mängel vorhanden sein können. Diese Mängel werden über die Abstraktionsebenen hinweg nicht zwangsläufig dieselben sein, sondern sich der Stufe angleichen, in welcher sich auch der erstellte Code befindet. LCNC führt neue Konzepte und Eigenschaften ein, wie dies beispielweise auch schon mit der Einführung der objektorientierten Programmierung (OOP) der Fall war. OOP, mit ihrem Verständnis der Abstraktion, hat wohl einige Probleme gelöst, aber auch neue geschaffen. Deshalb dürfen Qualitätsstandards auf keiner Abstraktionsstufe, und somit auch für LCNC, vernachlässigt werden.

Der Annahme, dass durch die Erstellung von Anwendungen in hoch abstrahierten LCNC -Plattformen, keine Qualitätsmängel mehr entstehen, widerspricht auch Lethbridge. Er entgegnet, dass sich in der Praxis, in auf LCNC-Plattformen geschriebener Software, oft grosse Mengen an komplexem Code häufen, dessen Wartung schwieriger sein kann als bei herkömmlichen Sprachen, da die LCNC-Plattformen etablierte technische Verfahren wie Versionskontrolle, Trennung von Zuständigkeiten, automatisierte Tests und Clean Code Programmierung in der Regel nicht angemessen unterstützen. Daher muss sichergestellt werden, dass LCNC-Plattformen moderne Software-Engineering-Praktiken genauso gut unterstützen wie herkömmliche Sprachen. Sie müssen vor allem die Möglichkeit bieten, die Qualitätssicherungsmassnahmen auf der gleichen Abstraktionsstufe ausführen zu können, auf der die Anwendung selbst geschrieben worden ist. Dadurch werden keine zusätzlichen Programmierkenntnissen vorausgesetzt. Weiter sollten die Anbieter ähnlicher Plattformen Wege zur Zusammenarbeit finden, um den Austausch von Code zwischen solchen Anwendungen zu ermöglichen.

Die letztendliche Zielsetzung dieser Arbeit war es einen Beitrag zum Fortschritt auf diesem Forschungsgebiet zu liefern, indem Methoden zur Qualitätssicherung, für von in LCNC erstellte Anwendungen, identifiziert und dargestellt werden. In diesem Kapitel werden wichtige theoretische und praktische Implikationen, Einschränkungen und Vorschläge für künftige Forschungsarbeiten und Schlussfolgerungen dargestellt.

Das Design und die Entwicklung der Artefakte wurde für die LCNC-Plattform Posity erstellt. Diese Plattform kann als repräsentativer Vertreter der modelbasierten LCNC-Plattformen angesehen werden, die Anwendungsentwicklung durch das Verwenden von grafischen Elementen ermöglicht. Der Lesende muss sich der Limitierung der Allgemeingültigkeit der Resultate, durch den gegebenen Kontext bewusst sein. Die Beantwortung der Forschungsfragen erfolgt für wesentliche Teile im Kontext dieser Plattform und es bedarf weiterer Forschungsarbeit die gemachten Aussagen weiter zu verallgemeinern.

In der Bearbeitung der Forschungsfrage:

> *FF 2 (Marion Mürner): Welche Metriken für die Erkennung von Code-Smells aus der klassischen Programmierung können auf Low-Code- No-Code-Entwicklungswerkzeuge zur Verbesserung der Code-Qualität angewandt werden?*

wurden Regeln und Metriken zur statischen Code-Analyse aus sechs verschiedenen Kategorien (Namenskonvention, Kommentare, Spezifikation, Komplexität, Formatierung und Ausrichtung) der klassischen Programmierung für das Posity Design Studio adaptiert und validiert. Durch die Anwendung der Regeln und Metriken konnten Code-Smells aufgedeckt werden, die bis anhin entweder gar nicht bemerkt worden sind oder schwer aufzufinden waren. Die Forschungsfrage lässt sich nicht abschliessend beantworten, weil nur eine Untermenge von Regeln und Metriken adaptiert worden ist. Es konnte aber gezeigt werden, dass diverse Regeln und Metriken aus der klassischen Softwareentwicklung auch für die Analyse von in LCNC geschriebenen Anwendungen verwendet werden können. Wie es auch für verschiedene klassische Programmiersprachen, einen unterschiedlichen Satz von Regeln und Metriken für die Code-Analyse braucht, so trifft dies auch für LCNC zu. Es gibt Schnittmengen von Regeln, welche allgemeingültig anwendbar sind und spezifische Regeln, deren Gültigkeitsbereich nur innerhalb eines bestimmten Programmierparadigmas liegt oder gar nur für eine bestimmte Sprache angewandt werden können. Das Bestimmen dieser, möglicherweise auch neuen Regeln und Metriken, bedarf weiterer Forschungsarbeit.

Die Studie von Al Alamin et al., 2021 erwähnt, dass Testing und Debugging für Low-Code-Software-Development aufgrund der grafischen Natur der Entwicklungsumgebungen eine grosse Herausforderung darstellt und Hersteller von LCNC-Plattformen darauf abzielen, die Komplexität von Testing, Deployment und Wartung weg zu abstrahieren.

Diesbezüglich ist die Autorin mit der Studie einig. Gegenüber der Aussage, dass bei LCNC-Plattformen in der Regel weniger Tests erforderlich sind, weil der Plattformbetreiber die bereitgestellten Module testet und überwacht, Tests dadurch weniger wichtig als in der traditionellen Entwicklung sind, nimmt sie jedoch eine andere Position ein. Sie ist der Meinung, dass Testing und damit entsprechende Werkzeuge für LCNC-Plattformen genauso wichtig sind, wie bei der klassischen Entwicklung von Software, nur erfolgt das Testing auf einem anderen Abstraktionslevel. Auch wenn LCNC-Entwickler normalerweise Anwendungen per Drag-and-Drop von bestehenden Komponenten erstellen, gibt es dennoch genügend Raum für Fehler. Angefangen von zu gross gewordenen Diagrammen bis hin zu Missverständnissen bei der Interpretation der Spezifikation, öffnet sich ein breites Spektrum von potenziellen Fehlern, die durch Testing aufgedeckt werden könnten. Als dementsprechend wichtig erachtet die Autorin, neben der Verwendung von statischen Software-Testverfahren, den Einsatz von dynamischen Software-Testverfahren. In dieser Arbeit wurde anhand von zwei Testverfahren untersucht, ob sie sich für Anwendungen im Bereich LCNC eignen. Für die Beantwortung der Forschungsfrage FF3 wurden zwei Teilfragen gestellt, wovon die Teilfrage FF 3.2 in der Verantwortung der Autorin lag.

> *FF 3 (Gemeinsam): Eignen sich dynamische Software-Testverfahren für Anwendungen im Bereich Low-Code und No-Code und wie können diese eingesetzt werden?*
>
> *Teil FF 3.2 (Marion Mürner): Eigenen sich diversifizierende Testmethoden für das Testen von Anwendungen im Bereich Low-Code und No-Code und wie können diese eingesetzt werden?*

Als Stellvertreter der diversifizierenden Testmethoden, wurde der Regressionstest gewählt, weil dieser eine weite Verbreitung in der traditionellen Softwareentwicklung geniesst. Für die Beantwortung von Teilfrage FF3.2 wurde ein Konzept zur Erstellung, Ausführung und Validierung von Regressionstests für das Posity-Framework erstellt. Das Konzept belegt, dass sich Regressionstests für das Testen der erstellten Anwendungen eigenen. Die Eigenschaft, dass Regressionstests auch für bereits bestehende Anwendungen, bei welchen die Testbarkeit bei der Entwicklung nicht beachtet worden ist, eingesetzt werden können, wird von der Autorin als besonders gewinnbringend angesehen. Die Ver-

allgemeinerung auf andere Plattformen muss auch in diesem Fall erst durch weitere Forschung belegt werden. Die Autorin ist aber optimistisch, dass das im Konzept beschriebene Verfahren für Regressionstests generell für LCNC-Plattformen anwendbar ist.

Zusammenfassend über beide Forschungsfragen lässt sich sagen, eine Herausforderung von LCNC-Plattformen, im Bereich der Qualitätssicherungsmethoden, sieht die Autorin in der heterogenen Landschaft der verschiedenen Plattformen und der jeweils vollkommen unterschiedlichen Architektur. Entwicklungsumgebungen und Entwicklungstätigkeiten sind dadurch stark an das jeweilige Framework gekoppelt und zwischen den Frameworks gibt es kaum gemeinsame, austauschbare Standards. Dies hat zur Folge, dass Verfahren wie Versionskontrolle, automatisierte Tests oder Clean Code Programmierung pro Plattform neu entwickelt werden müssen, was dazu führt, dass diese in der Regel nicht angemessen unterstützt werden. Würden sich Anbieter ähnlicher Plattformen auf gewisse Standards einigen, könnten dadurch gemeinsame Lösungen etabliert werden, ähnlich wie dies in der textbasierten Entwicklung beispielswiese mit Git als vielfältig einsetzbares Werkzeug für Versionskontrolle passiert ist. Ebenfalls bedarf es weitere Forschung in der Entwicklung eines umfangreichen Kataloges für Metriken, welche für LCNC unabhängig von den verschiedenen Plattformen eingesetzt werden können. Die Möglichkeit Softwareautomatisierungs-Pipelines von Build Automation über Continous Integration bis hin zu DevOps dürfte ebenfalls eine Anforderung sein, welche LCNC-Plattformen erfüllen sollten. Das Potenzial für weitere Forschungsarbeit im Bereich der Qualitätssicherungsmassnahmen für LCNC-Entwicklung ist entsprechend hoch. Unabhängig von statischen oder dynamischen Testverfahren oder Analysen, es gibt noch viele Lücken zu füllen. Möglicherweise ist die Verbreitung solcher Plattformen nicht unerheblich davon abhängig, wie gut Qualitätssicherungsmassnahmen in Zukunft von den LCNC-Plattformen unterstützt werden.

11 Fazit

Die vorliegende Arbeit untersuchte die Eignung dreier, aus der klassisch, textuellen Softwareentwicklung bekannten Qualitätssicherungsmethoden für die Entwicklung von LCNC-Anwendungen. Dazu wurden vier Artefakte für die LCNC-Plattform Posity erstellt. Das verfolgte Ziel war es, anhand der Artefakte, geeignete Qualitätssicherungsmethoden durch Demonstration in einer LCNC-Umgebung identifizieren und beschreiben zu können und dabei Erkenntnisse und Handlungsempfehlungen für weitere Initiativen in diesem, noch wenig erforschten, Bereich festzuhalten. Um diese Zielsetzung zu erreichen, wurden drei Techniken aus der statischen und dynamischen Analyse von Programmen ausgewählt. Dies sind die Performance-Analyse, die Quellcode-Analyse und das Testing. Die Ergebnisse aus der Validierung der Artefakte legen nahe, dass alle drei untersuchten Techniken grundsätzlich für LCNC anwendbar sind, es jedoch durch die heterogene Landschaft der LCNC-Plattformen und deren oft stark isolierte, wenig modulare Architektur jeweils eigene Implementationen der Werkzeuge braucht. Hersteller sollten Wege zur Zusammenarbeit finden und zusammen mit der Forschung Standards entwickeln, damit Werkzeuge wiederverwendet werden können oder sogar Code ausgetauscht werden kann.

Weil es bisher wenig Untersuchungen von Qualitätssicherungsmethoden in LCNC-Anwendungen gibt, war die Analyse der Anforderungen schwierig. Dank dem Austausch mit Experten auf diesem Gebiet, konnten Anforderungen ermittelt und die Resultate validiert werden. Die Resultate, welche die Artefakte aus den Analysen liefern, können bereits zur Verbesserung der Software-Qualität der analysierten Posity-Applikationen verwendet werden. Im Weiteren erhoffen sich die Autoren, dass der Plattform-Hersteller Posity gewillt ist, die Konzepte für das Testing in deren LCNC-Entwicklungsumgebung umzusetzen. Darüber hinaus liefert diese Arbeit Anknüpfungspunkte für zukünftige Forschungen.

12 Literaturverzeichnis

Al Alamin, M. A., Malakar, S., Uddin, G., Afroz, S., Haider, T. B. & Iqbal, A. (2021). An Empirical Study of Developer Discussions on Low-Code Software Development Challenges. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)* (S. 46–57). IEEE. https://doi.org/10.1109/MSR52588.2021.00018

Andreas, S. & Tilo, L. (2012). *Basiswissen Softwaretest: Aus- und Weiterbildung zum Certified Tester, Foundation Level nach ISTQB-Standard* (5th überarbeitete und aktuelle Auflage 2012).

Asif, K. (2020). *10 Best Low-Code And No-Code Application Development Platforms in 2021*. https://hackernoon.com/10-low-code-and-no-code-application-development-platforms-ew513y8q

Brendan, G. (2022, 10. Mai). *Flame Graphs.* Brendan's site. https://www.brendangregg.com/flamegraphs.html

David, C. (2008). *What is application lifecycle management?* https://web.archive.org/web/20141207012857/http://www.microsoft.com/global/application-platform/en/us/RenderingAssets/Whitepapers/What%20is%20Application%20Lifecycle%20Management.pdf

Dennis Kafura (1985). A Survey of Software Metrics. https://dl.acm.org/doi/pdf/10.1145/320435.320583

Firefox Team. (2022). *Firefox Developer Tools.* Mozilla. https://firefox-dev.tools/

Google Chrome Team. (2022). *Chrome DevTools.* Google. https://developer.chrome.com/docs/devtools/

Hevner, March, Park & Ram (2004). Design Science in Information Systems Research. *MIS Quarterly*, *28*(1), 75. https://doi.org/10.2307/25148625

Hevner, A. R. (2007). A Three Cycle View of Design Science Research, *2007*.

Hurlburt, G. (2021). Low-Code, No-Code, What's Under the Hood? *IT Professional*, *23*(6), 4–7. https://doi.org/10.1109/MITP.2021.3123415

IEEE. *IEEE Standard Glossary of Software Engineering Terminology*. Piscataway, NJ, USA. IEEE.

International Organization for Standardization [*ISO/IEC 2500n]: Quality Management Division* (ISO/IEC 25000). https://iso25000.com/index.php/en/iso-25000-standards

JetBrains. (2021a). *dotTrace - Profiling Types.* JetBrains. https://www.jetbrains.com/help/profiler/Profiling_Guidelines__Choosing_the_Right_Profiling_Mode.html

JetBrains. (2021b). *dotTrace 2021.2: Performance Profiling*.

Johnson, S. (1978). *Lint, a C Program Checker*. Murray Hill, New Jersey 07974. Bell Laboratories. http://squoze.net/UNIX/v7/files/doc/15_lint.pdf

Khorram, F., Mottu, J.-M. & Sunyé, G. (2020). Challenges & opportunities in low-code testing. In E. Guerra & L. Iovino (Hrsg.), *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings* (S. 1–10). ACM. https://doi.org/10.1145/3417990.3420204

Lee, K. D. (2017). *Foundations of Programming Languages* (2. Aufl.). *Undergraduate Topics in Computer Science*. Springer International Publishing.

Len, B., Paul, C. & Rick, K. (2013). *Software Architecture in Practice* (3. Aufl.). *SEI series in software engineering*. Addison-Wesley.

Lethbridge, T. C. (2021). Low-Code Is Often High-Code, So We Must Design Low-Code Platforms to Enable Proper Software Engineering. In T. Margaria & B. Steffen (Hrsg.), *Lecture Notes in Computer Science. Leveraging Applications of Formal Methods, Verification and Validation* (Bd. 13036, S. 202–212). Springer International Publishing. https://doi.org/10.1007/978-3-030-89159-6_14

Liggesmeyer, P. (2009). *Software-Qualität: Testen, Analysieren und Verifizieren von Software* (2. Aufl.). Spektrum, Akad. Verl.

Marion Mürner & Matthias Christen (2021). Codequalität und Testing in Low-Code-Software-Development: Vorstudie zur Masterarbeit.

McCabe, T. J. (1976). A Complexity Measure. *IEEE Transactions on Software Engineering*, *SE-2*(4), 308–320. https://doi.org/10.1109/TSE.1976.233837

Microsoft. (2021a). *Common Language Runtime (CLR) overview*. https://docs.microsoft.com/en-us/dotnet/standard/clr

Microsoft. (2021b). *Visual Studio Profiler: Measure app performance in Visual Studio*.

Microsoft. (2022). *What is SQL Server Management Studio (SSMS)?* Microsoft. https://docs.microsoft.com/en-us/sql/ssms/sql-server-management-studio-ssms?view=sql-server-ver15

Microsoft Edge Team. *Übersicht über DevTools*. https://docs.microsoft.com/de-de/microsoft-edge/devtools-guide-chromium/overview

The NUnit Project. (2022). *NUnit - Test Result XML Format*. https://docs.nunit.org/articles/nunit/technical-notes/usage/Test-Result-XML-Format.html

O'Regan, G. (2008). *A Brief History of Computing*. Springer London. https://doi.org/10.1007/978-1-84800-084-1

Oracle. (2017). *Java Virtual Machine Specification: Java SE 9 Edition.* Oracle. https://docs.oracle.com/javase/specs/jvms/se9/html/jvms-1.html#jvms-1.2

Ottinger, T. & Langr, J. (2012). Unit Tests Are FIRST: Fast, Isolated, Repeatable, Self-Verifying, and Timely. *PRAGPUB MAGAZINE*, *1*.

Peffers, K., Tuunanen, T., Rothenberger, M. A. & Chatterjee, S. (2007). A Design Science Research Methodology for Information Systems Research. *Journal of Management Information Systems*, *24*(3), 45–77. https://doi.org/10.2753/MIS0742-1222240302

SonarQube. (2022). *SonarQube Documentation*. https://docs.sonarqube.org/latest/

SQL-Docs Team. (2020). *Monitor and Tune for Performance.* Microsoft. https://docs.microsoft.com/en-us/sql/relational-databases/performance/monitor-and-tune-for-performance?view=sql-server-ver15

SQL-Docs Team. (2022a). *Monitoring performance by using the Query Store.* Microsoft. https://docs.microsoft.com/en-us/sql/relational-databases/performance/monitoring-performance-by-using-the-query-store?view=sql-server-ver15

SQL-Docs Team. (2022b). *sys.dm_exec_procedure_stats (Transact-SQL).* Microsoft. https://docs.microsoft.com/en-us/sql/relational-databases/system-dynamic-management-views/sys-dm-exec-procedure-stats-transact-sql?view=sql-server-ver15

Anhang

A Kapitelübersicht Autoren

Kapitel	Autor
Management Summary	Gemeinsam
1 Einleitung	Gemeinsam
2 Related Work	Gemeinsam
2.1 Qualitätssicherungsmassnahmen Performance-Analyse und spezifikationsorientiertes Testverfahren	Matthias Christen
2.2 Qualitätssicherungsmassnahmen Quellcode-Analyse und diversifizierende Testverfahren	Marion Mürner
3 Vorgehen und Methoden	Gemeinsam
4 Analyse	Gemeinsam
5 Artefakt DevTools	Matthias Christen
6 Artefakt Performance-Analyse	Matthias Christen
7 Artefakt Quellcode-Analyse – Posity Linter	Marion Mürner
8 Konzept für spezifikationsorientierte Testverfahren (Black-box-Testing)	Matthias Christen
9 Konzept für diversifizierende Testverfahren (Regressionstest	Marion Mürner
10 Schlussfolgerungen und Handlungsempfehlungen	Gemeinsam
10.1 Qualitätssicherungsmassnahmen Performance-Analyse und spezifikationsorientiertes Testverfahren	Matthias Christen
10.2 Qualitätssicherungsmassnahmen Quellcode-Analyse und diversifizierende Testverfahren	Marion Mürner
11 Fazit	Gemeinsam

B SQL-Stored Procedure für Query-Diagramm: Produkt auswählen

```
ALTER PROCEDURE [UserData].[Query_ProductSelect_Fill]
    -- parameters of procedure
    @sys_parameter_37381e39bc174baa859ae4c4d9e30bcc nvarchar(max) =
null, -- (parameter name: NameProduct)
    -- language parameters of procedure
    @language1 nchar(4) = 'eng', -- first language
    @language2 nchar(4) = 'eng', -- second language to search, if
first language was not found
    -- ledger and data role of user
    @Sys_Ledger uniqueidentifier = null, -- ledger of user
    @Sys_ListOfDataRolesOfUser nvarchar(max) = '' -- list of the data
roles of user

AS
BEGIN

    SET NOCOUNT ON;

    -- initializing default values of parameters
    IF (@sys_parameter_37381e39bc174baa859ae4c4d9e30bcc IS NULL) OR
(LEN(@sys_parameter_37381e39bc174baa859ae4c4d9e30bcc + '.') <= 1)
        SELECT @sys_parameter_37381e39bc174baa859ae4c4d9e30bcc = '%';


    -- create tables holding the results returned by the stored
procedure

    -- creating result table Product (logical name [deu]: Produkt)
    CREATE TABLE #Product (
        [PK_Product] uniqueidentifier,
        [FK_ArticelGroup] uniqueidentifier,
        [FK_Supplier] uniqueidentifier,
        [PK_ArticelGroup] uniqueidentifier,
        [FK_ProductGroup] uniqueidentifier,
        [PK_ProductGroup] uniqueidentifier,
        [PK_Partner] uniqueidentifier,
        [NameProduct] nvarchar(200),
        [ExternalNameProduct] nvarchar(200),
        [NameArticleGroup] nvarchar(100),
        [NamePartner] nvarchar(100),
```

```
        [NameProductGroup] nvarchar(100),
        [DescriptionProduct] nvarchar(max),
        [Sys_ExecutionDateTime_Product] nchar(33),
        [Sys_ExecutionStatus_Product] uniqueidentifier,
        [Sys_Specialization_Product] nvarchar(max)
    ); -- end create table holding the results

    -- create variables used within stored procedure
    DECLARE @NrOfRecords int;

    -- create variables of table Product (logical name [deu]: Produkt)
    DECLARE @PK_Product uniqueidentifier;
    DECLARE @FK_ArticelGroup uniqueidentifier;
    DECLARE @FK_Supplier uniqueidentifier;
    DECLARE @NameProduct nvarchar(200);
    DECLARE @ExternalNameProduct nvarchar(200);
    DECLARE @NameArticleGroup nvarchar(100);
    DECLARE @NamePartner nvarchar(100);
    DECLARE @NameProductGroup nvarchar(100);
    DECLARE @DescriptionProduct nvarchar(max);
    DECLARE @Sys_ExecutionDateTime_Product nchar(33);
    DECLARE @Sys_ExecutionStatus_Product uniqueidentifier;
    DECLARE @Sys_Specialization_Product nvarchar(max);


    -- body of stored procedure
    SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
    BEGIN TRANSACTION
        -- execute the commands of the execution plan
        GOTO Execute_Join_Product;
        Return_Execute_Join_Product:
    COMMIT TRANSACTION
    SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;

    -- return the resulting tables
    SELECT TOP (1000) * FROM #Product
       ORDER BY [NameProduct];
    IF @@ROWCOUNT = 1000 RAISERROR (N'Error: Result table of query
table ''Produkt'' was limited.', 10,1);

    GOTO ExitStoredProcedure;
```

```
Execute_Join_Product:
-------------------------------------------------------------------------
-------------------------------------------------------
INSERT INTO #Product ([PK_Product], [FK_ArticelGroup], [FK_Supplier],
[PK_ArticelGroup], [FK_ProductGroup], [PK_ProductGroup], [PK_Partner],
[NameProduct], [ExternalNameProduct], [NameArticleGroup],
[NamePartner], [NameProductGroup], [DescriptionProduct],
[Sys_ExecutionDateTime_Product], [Sys_ExecutionStatus_Product],
[Sys_Specialization_Product])
SELECT TOP (1000) [Product].[PK_Product], [Product].[FK_ArticelGroup],
[Product].[FK_Supplier], [ArticelGroup].[PK_ArticelGroup],
[ArticelGroup].[FK_ProductGroup], [ProductGroup].[PK_ProductGroup],
[Partner].[PK_Partner], [Product].[NameProduct],
[Product].[ExternalNameProduct], [ArticelGroup].[NameArticleGroup],
[Partner].[NamePartner], [ProductGroup].[NameProductGroup],
[Product].[DescriptionProduct], [Product].[Sys_ExecutionDateTime],
[Product].[Sys_ExecutionStatus], [Product].[Sys_Specialization] FROM
[Product] AS [Product]
    LEFT JOIN [ArticelGroup] AS [ArticelGroup] ON
[Product].[FK_ArticelGroup] = [ArticelGroup].[PK_ArticelGroup]
    LEFT JOIN [ProductGroup] AS [ProductGroup] ON
[ArticelGroup].[FK_ProductGroup] = [ProductGroup].[PK_ProductGroup]
    LEFT JOIN [Partner] AS [Partner] ON [Product].[FK_Supplier] =
[Partner].[PK_Partner]
    WHERE ( -- begin checking statuses of records
        [Product].[Sys_ExecutionStatus] = CONVERT(uniqueidentifier,
'372628e2-9596-4e84-b248-583213da2678')
    ) -- end checking statuses of records
    AND (((([Product].[NameProduct] COLLATE Latin1_General_CI_AI LIKE
'%' + @sys_parameter_37381e39bc174baa859ae4c4d9e30bcc + '%' OR
([Product].[NameProduct] IS NULL AND
@sys_parameter_37381e39bc174baa859ae4c4d9e30bcc = '%') OR
(@sys_parameter_37381e39bc174baa859ae4c4d9e30bcc IS NULL)) OR
([Partner].[NamePartner] COLLATE Latin1_General_CI_AI LIKE '%' +
@sys_parameter_37381e39bc174baa859ae4c4d9e30bcc + '%' OR
([Partner].[NamePartner] IS NULL AND
@sys_parameter_37381e39bc174baa859ae4c4d9e30bcc = '%') OR
(@sys_parameter_37381e39bc174baa859ae4c4d9e30bcc IS NULL))) OR
((([ArticelGroup].[NameArticleGroup] COLLATE Latin1_General_CI_AI LIKE
'%' + @sys_parameter_37381e39bc174baa859ae4c4d9e30bcc + '%' OR
([ArticelGroup].[NameArticleGroup] IS NULL AND
@sys_parameter_37381e39bc174baa859ae4c4d9e30bcc = '%') OR
```

```
(@sys_parameter_37381e39bc174baa859ae4c4d9e30bcc IS NULL)) OR
([ProductGroup].[NameProductGroup] COLLATE Latin1_General_CI_AI LIKE
'%' + @sys_parameter_37381e39bc174baa859ae4c4d9e30bcc + '%' OR
([ProductGroup].[NameProductGroup] IS NULL AND
@sys_parameter_37381e39bc174baa859ae4c4d9e30bcc = '%') OR
(@sys_parameter_37381e39bc174baa859ae4c4d9e30bcc IS NULL))) OR
([Product].[ExternalNameProduct] COLLATE Latin1_General_CI_AI LIKE
@sys_parameter_37381e39bc174baa859ae4c4d9e30bcc OR
([Product].[ExternalNameProduct] IS NULL AND
@sys_parameter_37381e39bc174baa859ae4c4d9e30bcc = '%') OR
(@sys_parameter_37381e39bc174baa859ae4c4d9e30bcc IS NULL)))))

IF @@ROWCOUNT = 1000 RAISERROR (N'Error: Result of base query table
''Product'' was limited to 1000 records (join statement)', 10,1);;

GOTO Return_Execute_Join_Product;

ExitStoredProcedure:
END
```

C SQL-Stored Procedure für System View sys. dm_exec_procedure_stats Abfragen

```
-----------------------------------------------------------------------
-----------------------------------------------------------------------
--------
-- Stored procedure: QueryStore_GetOptions
-- Autor: Matthias Christen
-- Date creation: 19. März 2022
-- Short description: Return performance data for stored procedures
from the SQL server system view sys.dm_exec_procedure_stats
-- All rights: Posity Ltd. Switzerland
-- Status: internal use only
--
-- Description:
-- Return performance data for stored procedures from the SQL server
system view sys.dm_exec_procedure_stats
-- Default: return performance data for stored procedures in the
UserData schema which belong to a Posity query diagram
-- Note: Stored procedures from query diagrams have the stored
procedure name format UserData.Query_[physical query name]_Fill
--
-- Run with:
-- EXEC [SystemData].[QueryPerformance_Fill]


ALTER PROCEDURE [SystemData].[QueryPerformance_Fill]
  @SchemaName nvarchar(100) = 'UserData', -- schema to search stored
procedures
  @StoredProcedureNameFilter nvarchar(100) = 'Query_%_Fill', -- filter
stored procedures to return by name
  @language1 nchar(4) = 'eng', -- first language to return the logical
query name
  @language2 nchar(4) = 'eng' -- second language to return the logical
query name
  AS
BEGIN
  DECLARE @PKQuery uniqueidentifier;
  DECLARE @ProcedureName nvarchar(500);
  DECLARE @PhysicalQueryName nvarchar(110);
  DECLARE @LogicalQueryName nvarchar(110);
```

```
  DECLARE @SqlHandle varbinary(60);
  DECLARE @CachedTime datetime;
  DECLARE @ExecCount bigint;
  DECLARE @MinElapsedTime bigint;
  DECLARE @MaxElapsedTime bigint;
  DECLARE @TotalElapsedTime bigint;
  DECLARE @AvgElapsedTime bigint;

  CREATE TABLE #ProcedureStats
  (
    ProcedureName nvarchar(500),
    PhysicalQueryName nvarchar(110),
    LogicalQueryName nvarchar(110),
    SqlHandle varbinary(60),
    CachedTime datetime,
    ExecCount bigint,
    MinElapsedTime bigint,
    MaxElapsedTime bigint,
    TotalElapsedTime bigint,
    AvgElapsedTime bigint
  );

  -- get all statistics data for the stored procedures which are in
the schema defined with @SchemaName and have a name matching the
@StoredProcedureNameFilter
  DECLARE cursorProcedureStats CURSOR FAST_FORWARD FOR
    SELECT sch.[name] + '.' + OBJECT_NAME( procstats.object_id ) As
ProcedureName , REPLACE(REPLACE(TRIM(OBJECT_NAME( obj.object_id )),
'Query_', ''), '_Fill', ''), [sql_handle] , cached_time,
execution_count, min_elapsed_time, max_elapsed_time,
total_elapsed_time, total_elapsed_time / execution_count As
avg_elapsed_time FROM sys.dm_exec_procedure_stats procstats
       JOIN sys.objects obj ON obj.object_id = procstats.object_id
       JOIN sys.schemas sch ON sch.schema_id = obj.schema_id
    WHERE sch.[name] = @SchemaName AND OBJECT_NAME(
procstats.object_id ) LIKE @StoredProcedureNameFilter

  -- get for the found stored procedures the logical query name (name
of the Posity query diagram)
  OPEN cursorProcedureStats;
  FETCH NEXT FROM cursorProcedureStats
```

```
    INTO @ProcedureName, @PhysicalQueryName, @SqlHandle, @CachedTime,
@ExecCount, @MinElapsedTime, @MaxElapsedTime, @TotalElapsedTime,
@AvgElapsedTime;
  WHILE @@FETCH_STATUS = 0
  BEGIN
    -- try to find the meta data for the query by its physical query
name
    SELECT @PKQuery = PKQuery FROM [MetaData].[Query] AS q WHERE
q.PhysicalQueryName = @PhysicalQueryName;
    IF @@ROWCOUNT > 0
    BEGIN
      -- if found use the logical query name from the query language
table
      SELECT TOP 1 @LogicalQueryName = LogicalQueryName FROM
[MetaData].[QueryLanguage] AS ql WHERE ql.PKQueryLanguage = @PKQuery
AND TRIM(ql.FKLanguageOfQueryLanguage) = @language1;
      IF @@ROWCOUNT = 0
      BEGIN
        -- try to find a logical name in the second language of the
user
        SELECT TOP 1 @LogicalQueryName = LogicalQueryName FROM
[MetaData].[QueryLanguage] AS ql WHERE ql.PKQueryLanguage = @PKQuery
AND TRIM(ql.FKLanguageOfQueryLanguage) = @language2;
      END
      ELSE
      BEGIN
        -- no logical name in the second language of the user found >
try to find a logical name in any name
        SELECT TOP 1 @LogicalQueryName = LogicalQueryName FROM
[MetaData].[QueryLanguage] AS ql WHERE ql.PKQueryLanguage = @PKQuery;
        IF @@ROWCOUNT = 0
        BEGIN
          SELECT @LogicalQueryName = 'This query is missing a
language entry!';
        END
      END
    END
    ELSE
    BEGIN
      SELECT @LogicalQueryName = 'Has no logical query name';
    END
```

```
    INSERT INTO #ProcedureStats (ProcedureName, PhysicalQueryName,
LogicalQueryName, SqlHandle, CachedTime, ExecCount, MinElapsedTime,
MaxElapsedTime, TotalElapsedTime, AvgElapsedTime) VALUES
(@ProcedureName, @PhysicalQueryName, @LogicalQueryName, @SqlHandle,
@CachedTime, @ExecCount, @MinElapsedTime, @MaxElapsedTime,
@TotalElapsedTime, @AvgElapsedTime);

    FETCH NEXT FROM cursorProcedureStats
      INTO @ProcedureName, @PhysicalQueryName, @SqlHandle,
@CachedTime, @ExecCount, @MinElapsedTime, @MaxElapsedTime,
@TotalElapsedTime, @AvgElapsedTime;
  END
  -- house keeping
  CLOSE cursorProcedureStats;
  DEALLOCATE cursorProcedureStats;

  -- return the stored procedures statistics
  SELECT LogicalQueryName, ProcedureName, CachedTime, ExecCount,
MinElapsedTime, MaxElapsedTime, TotalElapsedTime, AvgElapsedTime FROM
#procedureStats ORDER BY AvgElapsedTime DESC;
END
```

D Listening GetTraceTree Methode

```
internal TraceTree GetTraceTree(bool
includeComponentsWithZeroExecutionTime)
        {
            TraceBuilderConfig cfg;
            string executionId;
            string description;
            TraceTree traceTree = new TraceTree();
            TraceTreeNode aggregationNode;
            TraceTreeNode currentNode = null;
            TraceTreeNode parentNode = null;
            TraceTreeNode topStackNode = null;
            TraceTreeNodes nodes;
            Stack<TraceTreeNode> nodeStack = new
Stack<TraceTreeNode>();

            nodes = traceTree.GetNodes();

            foreach (RuntimeExecutedComponent rtc in
executedComponents)
```

```
            {
                executionId = rtc.GetExecutionId();

rtc.SetDescription(TraceBuilder.GetComponentDescription(rtc.GetGuidOfC
omponent()));
                cfg = configurations[executionId];
                if (cfg.IsShownInTrace() &&
(includeComponentsWithZeroExecutionTime || rtc.GetExecutionTimeTicks()
> 0 || cfg.HasAction(Actions.StartStackFrame) ||
cfg.HasAction(Actions.EndStackFrame)))
                {
                    // handling EndStackFrame
                    if (cfg.HasAction(Actions.EndStackFrame))
                    {
                        if (nodeStack.Count > 0)
                        {
                            parentNode = nodeStack.Pop().GetParent();

                            if (parentNode != null)
                            {
                                nodes = parentNode.GetNodes();
                            }
                            else
                            {
                                nodes = traceTree.GetNodes();
                            }
                        }
                        else
                        {
                            description = rtc.GetDescription();
                        }
                    }

                    // handling Aggregation
                    aggregationNode = null;
                    if (nodeStack.Count > 0 &&
nodeStack.Peek().IsAggregation())
                    {
                        // check if the node is already added and
increment its execution count else add as a new node
                        foreach (TraceTreeNode nodeToCheck in
nodes.GetNodes())
```

```
                        {
                            if
(nodeToCheck.GetRtc().GetGuidOfComponent() ==
rtc.GetGuidOfComponent())
                            {
                                aggregationNode = nodeToCheck;

aggregationNode.IncrementExecutionCount();
                                break;
                            }
                        }
                    }

                    // add a new trace node because the node is not
aggregated
                    if (aggregationNode == null)
                    {
                        currentNode = new TraceTreeNode(rtc,
parentNode);
                        if(cfg.HasAction(Actions.AggregateStackFrame))
                        {
                            currentNode.SetIsAggregation(true);
                        }
                        nodes.Add(currentNode);
                    }

                    // handling StartStackFrame
                    if (cfg.HasAction(Actions.StartStackFrame))
                    {

                        // check if the top stack node is an
aggregation node
                        aggregationNode = null;
                        if (nodeStack.Count > 0)
                        {
                            topStackNode = nodeStack.Peek();
                            if (topStackNode.IsAggregation() &&
topStackNode.GetRtc().GetGuidOfComponent() ==
rtc.GetGuidOfComponent())
                            {
                                aggregationNode = topStackNode;
```

```
aggregationNode.IncrementExecutionCount();
                            }
                        }

                        // add the new StartStack node to the stack
because its not a aggregation node already on the stack
                        if (aggregationNode == null)
                        {
                            nodeStack.Push(currentNode);
                            nodes = currentNode.GetNodes();
                            parentNode = currentNode;
                        }
                    }
                }
            }

            return traceTree;
        } // method GetTraceTree
```

E Fokus-Gruppe Miro Board

URL: Miro Board Fokus-Gruppe

Datei: MiroBoard-Fokus-Gruppe.pdf

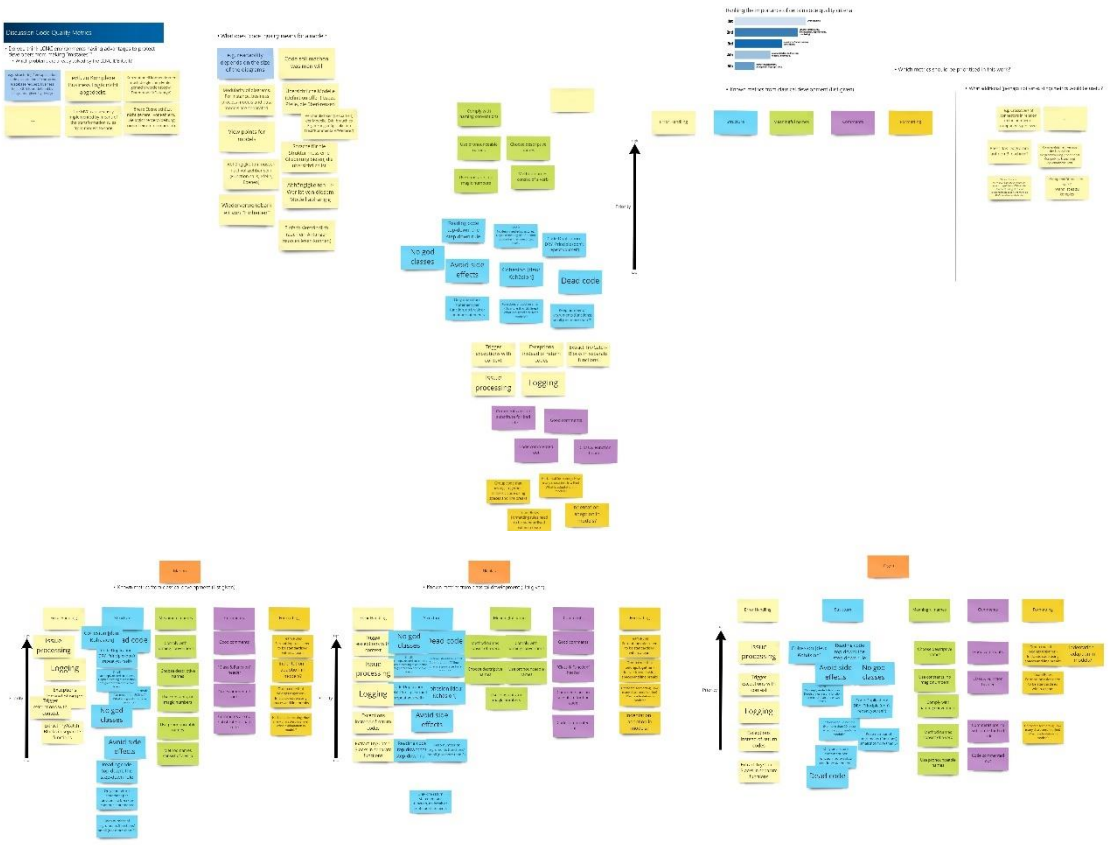


Abbildung 108: Miro Board Fokus-Gruppe Code-Metriken (Quelle: Eigene Darstellung)

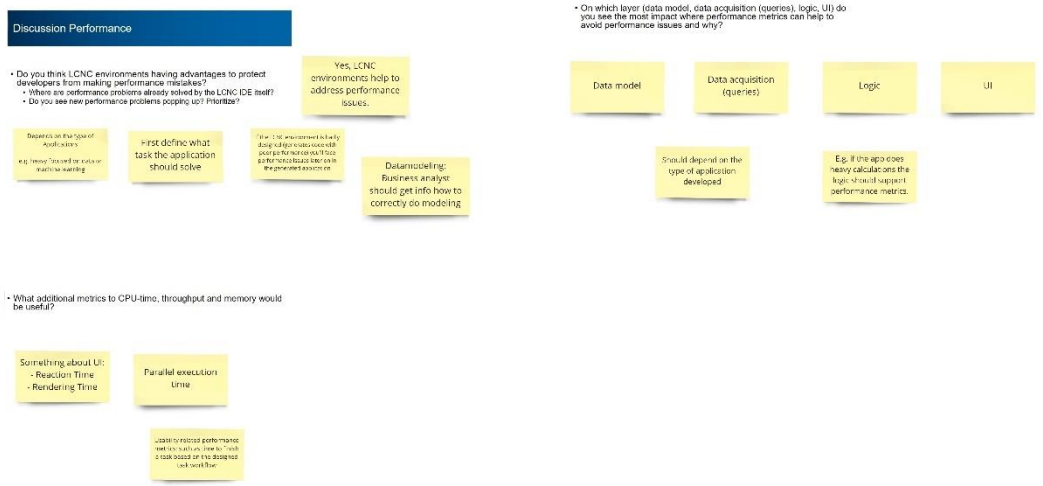


Abbildung 109: Miro Board Fokus-Gruppe: Performance (Quelle: Eigene Darstellung)

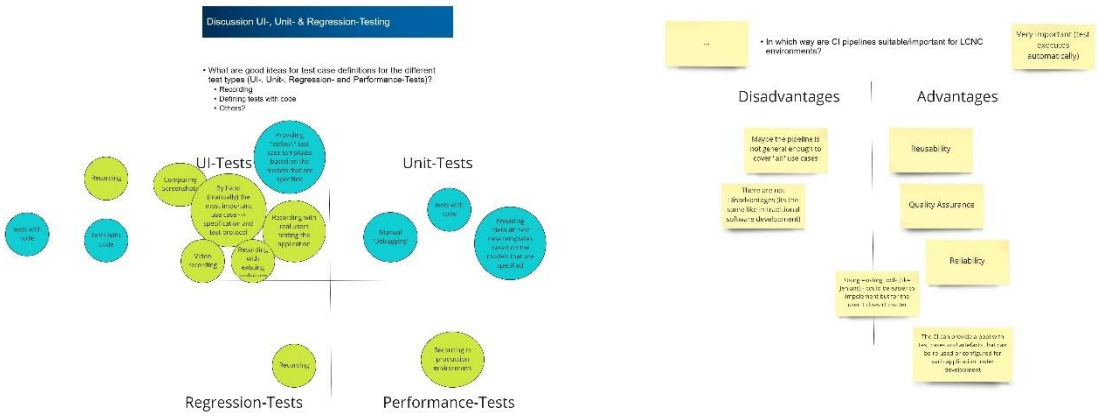


Abbildung 110: Miro Board Fokus-Gruppe: Testing (Quelle: Eigene Darstellung)