Technische Hochschule Ingolstadt

Fakultät Informatik

Konzeptionierung von Security Design Anti-Pattern und deren Suche im Software Design am Beispiel von Sequenzdiagrammen

Masterarbeit zur Erlangung des Grades

eines Master of Science Informatik

Erstprüfer: Prof. Dr.-Ing. Hans-Joachim Hof

Zweitprüfer: Prof. Dr.-Ing. Ernst-Heinrich Göldner

Eingereicht von: Timo Meilinger

Ausgabedatum: 17. Juni 2020

Abgabedatum: 18. August 2020

Inhaltsverzeichnis

Abbildungsverzeichnis

Quellcodeverzeichnis

Abkürzungsverzeichnis

AES	Advanced Encryption Standard
BSI	Bundesamt für Sicherheit in der Informationstechnik
CD	Continuous Delivery
CI	Continuous Integration
DES	Data Encryption Standard
DSGVO	Datenschutz Grundverordnung
JSON	JavaScript Object Notation
ML	Machine Learning
NIST	National Institute of Standards and Technology
OMG	Object Management Group
UML	Unified Modeling Language
SDL	Security Development Lifecycle
XMI	XML Metadata Interchange
XML	Extensible Markup Language

1 Einleitung

Die Sicherheit von modernen Softwaresystemen erhält in der vernetzten Welt einen immer höheren Stellenwert. Viele verschiedene Aspekte werden durch Software unterstützt oder umgesetzt. Diese reichen von vernetzten Autos mit Assistenzsystemen, die in das Fahrverhalten eingreifen können, bis zu Programmen die persönliche Daten verarbeiten. Bei den meisten Anwendungen ist die Sicherheit der Anwendungen essentiell, z.B. bei vernetzten Autos mit Assistenzsystemen besteht Gefahr für Leib und Leben der Insassen. Bei Programmen die persönliche Daten verarbeiten geht es nicht um die Gefahr für Leib und Leben, jedoch ist auch hier die Sicherheit essentiell, denn private Daten müssen aus rechtlicher und ethischer Sicht sicher sein. Durch die Verletzung der Sicherheit drohen hohe Strafen aus rechtlicher Sicht und ein möglicher Imageverlust für eine Firma, was wiederum einen hohen finanziellen Schaden verursachen kann. Doch ist auch die ethische Komponente, aus der persönliche Daten sicher sein sollen, nicht zu vernachlässigen, denn der Missbrauch dieser Daten kann für einzelne Personen großen Schaden verursachen. Diese Beispiele zeigen, wie essentiell es ist die Sicherheit von Anwendungen zu stärken und bei der Neuentwicklung von Software frühzeitig zu berücksichtigen.

Zusammenfassend kann festgehalten werden, Mängel im Bereich der Informationssicherheit können zu erheblichen Problemen führen. Dies wird auch im *IT-Grundschutz-Kataloge* des BSI - Bundesamt für Sicherheit in der Informationstechnik erläutert.[6]

Um die Informationssicherheit zu steigern, muss der Aspekt Sicherheit schon beim Softwaredesign berücksichtigt werden. Dies wird in einer Arbeit des Fraunhofer-Institut für Sichere Informationstechnologie SIT mit dem Titel *Entwicklung sicherer Software durch Security by Design* erkennbar. In dieser wird erläutert was Security by Design bedeutet, dass schon in der Entwurfsphase systematisches Vorgehen und Methoden angewendet

werden müssen um die Sicherheit von Software zu stärken. Das Ganze wird auch durch Beispiele untermauert, warum die Sicherheit von Software derartig essentiell ist. Ein Beispiel beschreibt, dass eine Demokratie auf den sicheren Austausch von Informationen, ohne das diese manipuliert werden, basiert, denn nur so ist eine informierte und fundierte Meinungsbildung möglich.[11] Auch in der Datenschutz Grundverordnung (DSGVO) wird beschrieben, dass schon bei der Entwicklung von Software der Datenschutz und damit die Sicherheit von Daten möglichst frühzeitig berücksichtigt werden soll.[10] Ein weit verbreitetes Vorgehen bei der Entwicklung sicherer Software ist der Security Development Lifecycle (SDL) von Microsoft. In diesem folgt auf die Entwurfsphase die Implementierung und danach das Testen.[21] Unter der Berücksichtigung der genannten Punkte ist es interessant, zusätzliches Testen oder eine automatisierte Prüfung schon vor der Implementierung einzuführen. Hierbei können mögliche Schwachstellen bereits vor der Implementierung entdeckt werden. Weiter stellt dieses Vorgehen eine systematische Methode dar, wodurch Security by Design, wie vom Fraunhofer-Institut für Sichere Informationstechnologie SIT beschrieben, erreicht wird.

An dieser Stelle soll diese Arbeit anknüpfen. Im weiteren Verlauf soll ein Konzept zur Definition von Security Design Anti-Pattern und der Suche nach diesen in einem Softwaredesign ausgearbeitet werden. Dies ist ein neuer Ansatz, bei dem versucht wird, potentielle Sicherheitslücken schon in der Entwurfsphase von Software zu entdecken. Hierfür muss ein Konzept ausgearbeitet werden, wie die Security Design Anti-Pattern definiert werden können. Diese sollen wiederverwendbar sein, daher müssen bei der Definition Möglichkeiten gefunden werden, wie diese auf verschiedene Softwaredesigns anwendbar sind. Weiter ist eine Datenstruktur zu finden, in der ein Softwaredesign verarbeitet werden kann, welche die Suche von Anti-Pattern erlaubt. Die Suche der Security Design Anti-Pattern in einem Softwaredesign muss auch konzeptionell ausgearbeitet werden. Für das Konzept soll ein Prototyp implementiert werden, um die Funktionsweise aufzuzeigen und die Umsetzbarkeit zu beweisen. Diese Konzepte und die Implementierung fokussieren sich auf die Sequenzdiagramme als ein Beispiel für Software Design Diagramme. Die vorgestellten Konzepte sollen mit Änderungen auch auf andere Diagrammtypen anwendbar sein. Weiter soll ausgearbeitet werden, ob mithilfe des Konzeptes potentielle Sicherheitslücken frühzeitig und bereits nach der Entwurfsphase von Software entdeckt werden können.

Zunächst werden Grundlagen, die für diese Arbeit relevant sind, diskutiert. Es werden verwandte Arbeiten vorgestellt und Unterschiede zu diesen aufgezeigt. Die anschließende Analyse wird Aufschluss geben, welche Anforderungen an die Security Design Anti-Pattern und die Suche von diesen in einem Softwaredesign zu stellen sind. Im Design werden der Aufbau und die einzelnen Komponenten des Konzepts vorgestellt. Hierbei wird die Definition und Speicherung von Security Design Anti-Pattern, sowie die Suche von diesen in einem Softwaredesign ausgearbeitet. Weiter wird eine Prototypimplementierung präsentiert, welche die Funktionsweise aufzeigt und die Umsetzbarkeit des Konzeptes beweisen wird. In der Evaluierung wird aufgezeigt, dass die Suche von Security Design Anti-Pattern großes Potential hat und das frühzeitige Erkennen von potentiellen Schwachstellen in der Entwurfsphase möglich ist. Abschließend wird in der Zusammenfassung die Arbeit nochmals komprimiert vorgestellt und ein Ausblick beschrieben, wie dieses Konzept in Zukunft erweitert werden kann und welches Potential dieses Konzept für weitere Funktionalitäten und Erweiterungen bietet.

2 Grundlagen

Nachfolgend sollen grundlegende Informationen zu den verwendeten Komponenten erläutert werden, welche als Wissensgrundlage für die folgende Arbeit dienen.

2.1 Sequenzdiagramme in der Unified Modeling Language (UML)

Die UML definiert eine allgemein verwendbare Modellierungssprache. Diese stellt verschiedene Diagramme und Notationselemente zur Verfügung, mit denen sowohl statische als auch dynamische Aspekte modelliert werden können. Dadurch bietet UML verschiedene Vorteile. Darunter sind: Eindeutigkeit, Verständlichkeit, Ausdrucksstärke, Standardisierung, Plattform- und Sprachunabhängigkeit.[14] UML bietet eine Vielzahl an verschiedenen Diagrammen. Diese können in Struktur- und Verhaltensdiagramme unterteilt werden.

Bei den Verhaltensdiagrammen gibt es die Untergruppe der Interaktionsdiagramme, diese sind eine Spezialform bei der nicht nur das Verhalten, sondern auch die Interaktion betrachtet wird. Die Strukturdiagramme modellieren die statischen und damit zeitunabhängigen Elemente eines Systems. Das bekannteste dieser Diagramme ist das Klassendiagramm, denn dieses stellt das zentrale Konzept der UML dar und ist deshalb aus der heutigen Softwareentwicklung nicht mehr wegzudenken. In Klassendiagrammen werden statische Bestandteile und Attribute von Systemen und deren Beziehungen untereinander dargestellt. Außer auf die Definition von Operationen werden alle dynamischen Aspekte außer acht gelassen. Im Gegensatz dazu legen die Verhaltensdiagramme den Fokus auf die dynamischen Aspekte und auf das Verhalten eines Systems. In diesen können Anwendungsfälle

oder auch Zustände und deren Übergänge in Diagrammen modelliert werden. Bekannte Beispiele für diese sind das Zustandsdiagramm und das Aktivitätsdiagramm. In der Gruppe der Interaktionsdiagramme sind bekannte Beispiele die Sequenzdiagramme und die Kommunikationsdiagramme. Eine Übersicht der verschiedenen Diagramme soll in Abbildung 1 gegeben werden.[14, 22]

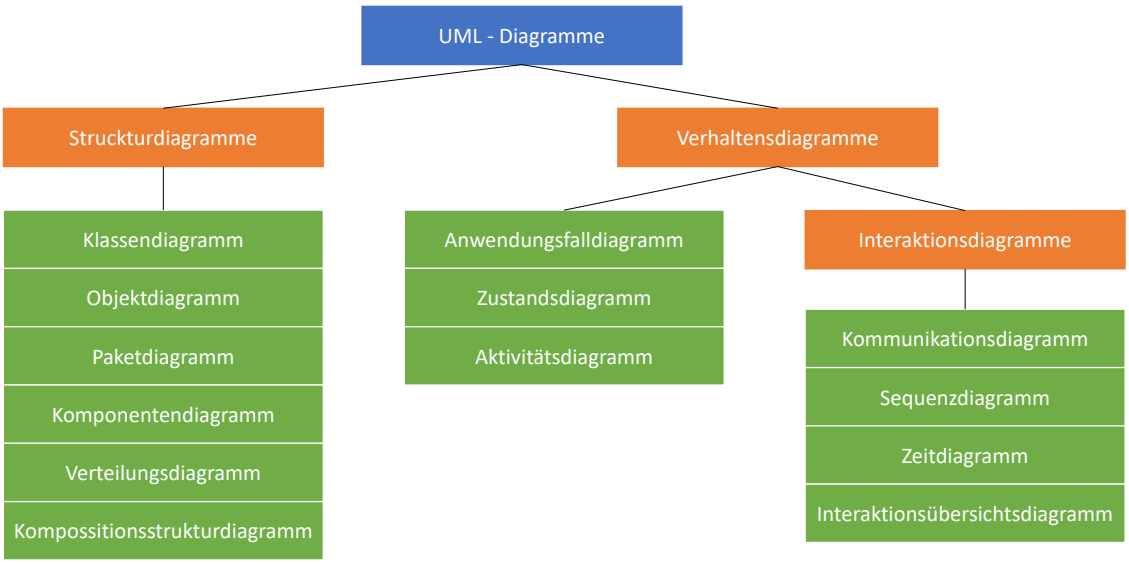

Abbildung 1: Übersicht der verschiedenen UML Diagramme[1]

Für die weitere Arbeit sind die Sequenzdiagramme wichtig, bei diesen handelt es sich um Verhaltensdiagramme bzw. genauer um Interaktionsdiagramme. Diese definieren den Nachrichtenfluss zwischen Objekten und zeigen den zeitlichen Ablauf von Nachrichten. Es sind keine Informationen über die Beziehungen der Objekte vorhanden. Nachrichten sind ein Informationsaustausch, auch der Aufruf von Funktionen kann als Nachrichtenaustausch angesehen und daher in Sequenzdiagrammen modelliert werden. Funktionsaufrufe sind eine Art von Informationsaustausch, da Informationen durch die Argumente und Rückgabewerte der Funktion übermittelt werden. Weiter ist auch der Aufruf ohne Argumente und Rückgabewerte ein Informationsaustausch, da allein die Tatsache des Ausführens einer Funktion eine Informationsweitergabe darstellt.

[1] In Anlehnung an https://www.edrawsoft.com/de/uml-introduction.html (aufgerufen am 11.07.2020)

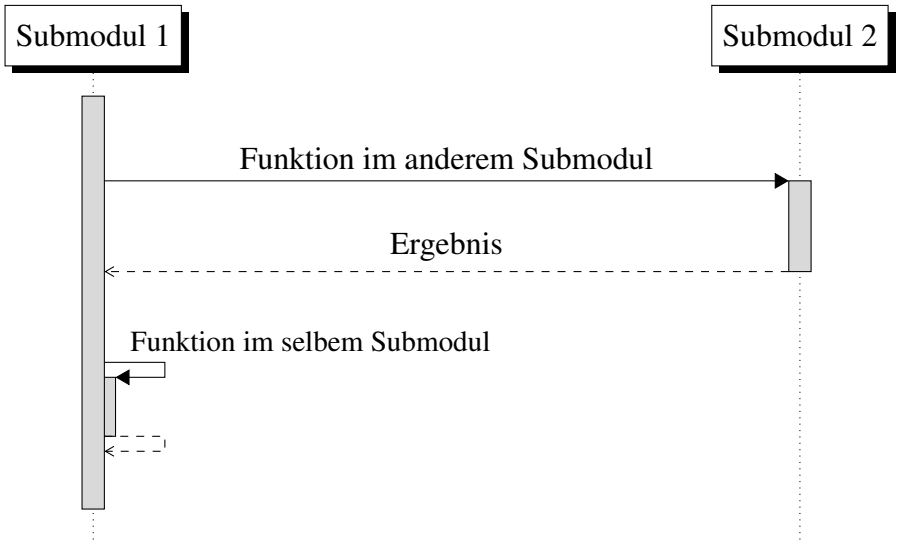

Abbildung 2: Einfaches Sequenzdiagramm mit zwei Funktionsaufrufen

In der Abbildung 2 ist ein Beispiel für ein Sequenzdiagramm zu sehen, dass sehr einfach und klein ist. In diesem wird von dem Submodul 1 eine Funktion des Submodul 2 aufgerufen. Diese Ausführung liefert ein Ergebnis an das Submodul 1 zurück. Danach wird eine Unterfunktion des Submodul 1 aufgerufen die keinen Wert an die ursprüngliche Funktion zurück gibt. Durch dieses Beispiel soll gezeigt werden, wie Abläufe in einem Programm durch ein Sequenzdiagramm dargestellt werden.

UML Diagramme können im XML (Extensible Markup Language) Format, genauer gesagt in XMI (XML Metadata Interchange) Format gespeichert werden. Dieses wurde von der OMG (Object Management Group) spezifiziert[2] und ermöglicht die Speicherung sowie den Austausch von UML Diagrammen in einer standardisierten Form.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <uml:Model xmi:version="20131001" xmlns:xmi="http://www.omg.org/spec/XMI/20131001" xmlns:uml="http://www.eclipse.org/
       uml2/5.0.0/UML" xmi:id="_MoAUgKTXEeqS64DNJzP7SQ" name="grundlagenExample">
3    <packageImport xmi:id="_VRd7IKTLEeq6AtKS1PGlbA">
4      <importedPackage xmi:type="uml:Model" href="pathmap://UML_LIBRARIES/UMLPrimitiveTypes.library.uml#_0"/>
5    </packageImport>
6    <packagedElement xmi:type="uml:Interaction" xmi:id="_interaction" name="Interaction">
7      <lifeline xmi:id="_submodul1" name="Submodul 1" coveredBy="_submodul1ExecStart _submodul1Main _msgExternSend
           _msgExternReplayReceive _msgInternSend _msgInternReceive _msgInternReplaySend _msgInternReplayReceive
           _submodul1InternEnd _submodul1MainEnd"/>
8      <lifeline xmi:id="_submodul2" name="Submodul 2" coveredBy="_msgExternReceive _msgExternReplaySend
           _submodul2ExternEnd"/>
9      <fragment xmi:type="uml:ExecutionOccurrenceSpecification" xmi:id="_submodul1ExecStart" name="" covered="_submodul1
           " execution="_submodul1Main"/>
10     <fragment xmi:type="uml:ActionExecutionSpecification" xmi:id="_submodul1Main" name="" covered="_submodul1" finish=
           "_submodul1InternEnd" start="_submodul1ExecStart"/>
```

[2] https://www.omg.org/spec/XMI/ (aufgerufen am 02.06.2020)

```
11      <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="_msgExternSend" name="" covered="_submodul1"
            message="_msgExtern"/>
12      <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="_msgExternReceive" name="" covered="_submodul2"
            message="_msgExtern"/>
13      <fragment xmi:type="uml:ActionExecutionSpecification" xmi:id="_submodul2ExternEnd" name="" covered="_submodul2"
            finish="_msgExternReplaySend" start="_msgExternReceive"/>
14      <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="_msgExternReplaySend" name="" covered="_submodul2"
             message="_msgExternReplay"/>
15      <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="_msgExternReplayReceive" name="" covered="
            _submodul1" message="_msgExternReplay"/>
16      <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="_msgInternSend" name="" covered="_submodul1"
            message="_msgIntern"/>
17      <fragment xmi:type="uml:ActionExecutionSpecification" xmi:id="_submodul1MainEnd" name="" covered="_submodul1"
            finish="_msgInternReplaySend" start="_msgInternSend"/>
18      <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="_msgInternReceive" name="" covered="_submodul1"
            message="_msgIntern"/>
19      <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="_msgInternReplaySend" name="" covered="_submodul1"
             message="_msgInternReplay"/>
20      <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="_msgInternReplayReceive" name="" covered="
            _submodul1" message="_msgInternReplay"/>
21      <fragment xmi:type="uml:ExecutionOccurrenceSpecification" xmi:id="_submodul1InternEnd" name="" covered="_submodul1
            " execution="_submodul1Main"/>
22      <message xmi:id="_msgExtern" name="Funktion im anderem Submodul" receiveEvent="_msgExternReceive" sendEvent="
            _msgExternSend"/>
23      <message xmi:id="_msgExternReplay" name="Ergebnis" messageSort="reply" receiveEvent="_msgExternReplayReceive"
            sendEvent="_msgExternReplaySend"/>
24      <message xmi:id="_msgIntern" name="Funktion im selbem Submodul" receiveEvent="_msgInternReceive" sendEvent="
            _msgInternSend"/>
25      <message xmi:id="_msgInternReplay" name="" messageSort="reply" receiveEvent="_msgInternReplayReceive" sendEvent="
            _msgInternReplaySend"/>
26    </packagedElement>
27  </uml:Model>
```

Quellcode 1: XMI Format der Abbildung 2

In dem Quellcode 1 ist das Sequenzdiagramm, dass in Abbildung 2 zu sehen ist, im XMI Format spezifiziert. Es ist zu erkennen, dass für die Lebenslinien (Lifeline) alle Ereignisse unter der Eigenschaft 'coveredBy' enthalten sind. Weiter sind in den aufgelisteten Ereignissen die zugehörigen Objekte enthalten. Beispielsweise in einer 'MessageOccurrenceSpecification' ist die zugehörige Nachricht unter der Eigenschaft 'message' enthalten. Für jedes Senden oder Empfangen einer Nachricht wird ein 'MessageOccurrenceSpecification' Objekt dem Diagramm hinzugefügt, indem die Informationen zu dieser enthalten sind. Dasselbe gilt für die Ausführungen, bei Start und Ende einer Ausführung wird ein 'ActionExecutionSpecification' bzw. eine 'ExecutionOccurrenceSpecification' hinzugefügt. In dieser wird auf die Ausführung 'execution' verwiesen und zusätzliche Informationen zu dieser gespeichert. Durch diese Form sind sowohl die Reihenfolge der verschiedenen Ausführungen als auch die aufkommenden Nachrichten spezifiziert und können zu den Elementen zugeordnet werden. Diese sind vollständig definiert, inklusive deren Ereignissen für Senden, Empfangen, Ausführungsstart und Ausführungsende.

2.2 Entwurfsmuster im sicheren Software Design

Entwurfsmuster können in verschieden Bereichen und Stufen des Entwicklungsprozesses angewendet werden. Es gibt Muster, die einen Prozess zur Entwicklung von sicherer Software beschreiben, wie den Secure Development Lifecycle von Microsoft.[21] Entwurfsmuster sind an verschiedenen Stellen im Entwicklungsprozess anzutreffen, denn diese können ein Muster zur Anforderungserstellung sein oder auch als eine Lösung für ein wiederkehrendes Problem im Softwareentwurf in der Designerstellung auftreten.

In der nachfolgenden Arbeit liegt der Fokus auf Entwurfsmuster im Software Design. Daher können diese definiert werden als *eine anpassbare Lösung für ein nicht triviales und immer wiederkehrendes Problem im Design und der Architektur von Software*.[12] Damit stellen diese Lösungen für verschiedene Herausforderungen im Software Design dar. Da das zu lösende Problem ein wiederkehrendes ist, sind die Entwurfsmuster wiederverwendbar. Entwurfsmuster sind auch unter den Namen 'Design Pattern', 'Solution Pattern' oder 'Software Pattern' bekannt.[12] Es gibt für viele verschiedene Problemstellungen, in verschiedensten Bereichen, Entwurfsmuster. Weiter können die Lösungen in statischer oder dynamischer Form modelliert werden, dies ist abhängig von der zu lösenden Herausforderung. Oft sind die Entwurfsmuster in UML modelliert, da diese alle benötigten syntaktischen Mittel zur Modellierung von statischen und auch dynamischen Mustern bereitstellt. Muster, welche auf die Lösung von Herausforderungen in der Informationssicherheit abzielen, werden als 'Security Design Pattern' bezeichnet.

Es lassen sich auch Entwurfsmuster entwickeln, die eine schlechte Lösung des Problems darstellen. Diese werden als 'Anti-Pattern' bezeichnet . Diese stellen zwar ein Lösungsmuster (Pattern) dar, haben jedoch negative Seiteneffekte oder andere Nachteile. Somit lassen sich auch verbreitete Fehler im Softwaredesign als Anti-Pattern darstellen und ein Musterkatalog erstellen, in dem diese gesammelt werden.

Daraus ergeben sich in der Kombination 'Security Design Anit-Pattern'. Diese können definiert werden *als eine unsichere Lösung für ein häufiges und wiederkehrendes Problem im Softwaredesign, die eine Verwundbarkeit für das gesamte System darstellen können.* Bei diesen handelt es sich jedoch um eine im Softwaredesign bewusst getroffene Entscheidung und nicht um Implementierungsfehler, die in der Umsetzung unbewusst entstanden sind.

Diese potentiellen Schwachstellen stellen in der Hierarchie der Schwachstellen sehr frühzeitig entstandene und durch den 'Fehler' im Softwaredesign einen konzeptionellen Fehler dar. Solche sind im späteren Verlauf der Entwicklung nur schwer zu beheben, da diese tief im Grundkonzept der Anwendung verwurzelt sind. Diese Security Design Anti-Pattern sollen in dieser Arbeit genauer betrachtet werden.

2.3 Graph Matching Problem

Das Finden von einem Graphen in einem anderen größeren Graphen, also die Identifizierung des Subgraphen, ist ein Forschungsthema an dem derzeit noch gearbeitet wird. Das sogenannte Graph Matching Problem gehört der Komplexitätsklasse NP an und bislang ist für dieses noch kein Polynomzeitalgorithmus, der das Problem löst, bekannt. Weiter ist derzeit kein NP-Vollständigkeitsnachweis gefunden worden.[25] Diese Komplexität kommt daher, dass die Graphen möglichst flexibel die komplexen Strukturen aus Informationen und Beziehungen abbilden sollen. Zum Beispiel kann über zwei Verbindungen von Knoten die boolesche Aussage getroffen werden ob diese identisch sind. Dies trifft zu, wenn die Knoten dieselben und alle Eigenschaften der Verbindung (z.B. Name) gleich sind. Jedoch ist es nicht so einfach eine Aussage darüber zu treffen, wie gleich zwei Graphen sind.[3, 19] Es müssen Aussagen über viele Knoten und deren Verbindung getroffen werden. Vor allem wenn das Ergebnis nicht nur eine vollständige, sondern auch eine Aussage über Ähnlichkeiten getroffen werden soll. Das bezieht sich auch auf die Frage der Teilübereinstimmung von zwei Graphen, denn für diese muss eine Aussage getroffen werden in welchem Maß zwei Graphen gleich sind.[9]

Für die verschiedenen Arten von Graphstrukturen und deren Ausprägungen gibt es verschiedene Algorithmen zum Finden von Subgraphen. Jedoch gibt es keinen Algorithmus, der für alle Strukturen effizient ist, denn die Effizienz der Verfahren ist stark von der Struktur und je nach Aufbau auch von der Höhe (wie viele Ebenen von Knoten es gibt) abhängig. Bei großen Strukturen spielen die Rechenleistung und Speicherausstattung von Computersystemen eine Rolle, denn bei manchen Verfahren müssen größere Teile des Graphen im Hauptspeicher vorgehalten werden als in anderen.[17]

Weiter ist das Graph Matching Problem, wenn es sich um baumähnliche Strukturen handelt, komplex. Jedoch sind hierfür effiziente Algorithmen bekannt, die das Problem des Vergleichs von zwei Baumgraphen lösen, beziehungsweise mathematisch beweisen.[1, 15] Bei baumähnlichen Strukturen können Knoten beliebig viele Kindknoten aufweisen und jeder Knoten kann zusätzliche Beziehungen zu anderen Knoten haben. Wenn zwei Graphen verglichen werden, kann die Aussage, ob zwei Knoten identisch sind, erst getroffen werden, wenn alle Kindknoten verglichen wurden. Denn ein Knoten kann nur identisch zu einem anderen sein, wenn alle Kindknoten der beiden Knoten identisch sind. Hierbei ist zu erkennen, dass es sich um ein rekursives Problem handelt. Denn mit der steigenden Anzahl an Kindknoten steigt der Aufwand zur Überprüfung stark an. Weiter muss auch verglichen werden, ob die Beziehung zwischen den Knoten identisch sind. Deswegen ist der Vergleich von breiten und hohen Strukturen, also Bäume, die viele Kindknoten über mehrere Ebenen haben, nicht trivial und auch rechenintensiv.

3 Verwandte Arbeiten

Weiter sollen Arbeiten mit ähnlichen Konzepten für die automatische Identifizierung von Sicherheitsschwachstellen, oder der Vermeidung derer in der Design Phase der Software Entwicklung, oder der späteren Identifizierung im abgeschlossenen Software Design, vorgestellt werden. Es soll auch differenziert werden, worin der Unterschied zu dieser Arbeit liegt.

3.1 Quelltext zentrierte Erkennung von Sicherheitsschwachstellen

Das Paper [23] ist ein Beispiel für Arbeiten, die Verwundbarkeiten von Software automatisiert erkennen wollen und hierzu den Quelltext der Programme analysieren. Hierbei ist die Fokussierung auf der Analyse des Source Codes gelegt. Im genannten Beispiel werden Schwachstellen mithilfe eines Machine Learnings (ML) Algorithmus gesucht. Dafür ist dieser mit unsicheren Beispielprogrammen angelernt worden.

Bei der Suche von potentiellen Schwachstellen im Quelltext können hauptsächlich Implementierungsfehler gefunden werden. Zum Teil ist es möglich, Entwurfsentscheidungen zu entdecken die Verwundbarkeiten verursachen. So kann zum Beispiel eine unsichere Verschlüsselungsmethode entdeckt werden. Jedoch sind Entwurfs- bzw. Designentscheidungen die eine Schwachstelle verursachen schwer zu entdecken und die Ausnahme.

Ein weiteres Beispiel für die Quelltext zentrierte Erkennung von potentiellen Schwachstellen, ist das häufig in Softwareprojekten verwendete SonarQube.[3] Dieses findet potentielle

[3] https://www.sonarqube.org (aufgerufen am 06.07.2020)

Fehler oder Verstöße gegen die Implementierungsregeln (Coding Guidelines) sowie potentielle Schwachstellen. Um diese Auffälligkeiten erkennen zu können, werden Sammlungen von Regeln angewendet, in denen definiert wurde wie die Coding Guidelines sind oder welche Funktionen und Bibliotheken unsicher sind und deshalb nicht verwendet werden sollen. Dies geschieht über die Definition von Regelsätzen. Bei der Verletzung einer der Regeln wird dies automatisch evaluiert und angezeigt. Diese Regeln können im weitesten Sinne als Anti-Pattern betrachtet werden, da diese Zustände beschreiben, die zu potentiellen Sicherheitsschwachstellen führen. Bei der Evaluation werden nur potentielle Schwachstellen oder Fehler entdeckt die in der Implementierung entstanden sind und nicht in dem Design der Software zugrunde liegen, da es sich hierbei um eine statische Quelltextanalyse handelt.

In dieser Arbeit ist der Fokus auf das Finden von Fehlern im Design der Software gerichtet. Daher werden auch Anti-Pattern definiert, die im Software Design zu finden sind und nicht erst nach der Implementierung. Der Vorteil hierbei ist, dass potentielle Schwachstellen viel früher entdeckt und somit einfacher beseitigt werden können. Dieses Vorgehen ist meist auch kostengünstiger, da die Behebung vor der Implementierung geschehen kann. Weiter bietet diese Arbeit durch die Suche von Anti-Pattern im Software Design eine Möglichkeit, die Entwürfe zu Testen und somit automatisiert zu evaluieren. Hiermit wird eine Möglichkeit geschaffen, potentielle Schwachstellen früher im Entwicklungsprozess zu finden. Zudem sollen diese automatisiert erkannt werden, ohne dass Spezialisten diese in einem durch Menschen durchgeführtes Review finden müssen.

3.2 Security Design Pattern, Guidlines und Prinzipien

Eine weitere verbreite Methodik um Schwachstellen, die in der Design Phase entstehen, zu vermeiden sind die Security Design Pattern. Diese stellen eine anpassbare und sichere Lösung für ein nicht triviales und immer wiederkehrendes Problem im Software Design dar. Es gibt große Sammlungen von solchen Entwurfsmustern, die eine bestimmte Lösung, oder eine Empfehlung zum Vorgehen beschreiben können.[13]

Zu den Pattern ähnlich sind die sogenannten Security Design Guidlines, welche rein theoretische Informationen und Richtlinien darstellen.[20] Diese können auch in Form von Normen oder Standards auftreten und so grundlegende Handlungsempfehlungen oder Anweisungen aufstellen. Im Gegensatz zu den Pattern sind die Guidelines rein theoretisch und kein praktisches Muster das direkt übernommen werden kann.

Die Security Design Prinzipien gehören zu den Guidelines, stellen jedoch eine spezifische Gruppe dar. Diese Guidelines sind bewährte Regeln für das Erstellen von sicherer Software und müssen bei bestimmten Problemen angewendet werden. Ein solches Prinzip ist das 'Principle of Least Privilege', welches aussagt, das ein Individuum nur so viele Rechte erhalten soll, wie es auch wirklich benötigt. Weiter ist 'Principe of Separation of Privilege' ein verbreitetes und wichtiges Prinzip, jenes sagt aus, das ein System den Zugriff nicht aufgrund einer einzigen Bedingung gewähren soll.[20] Hierbei ist ersichtlich, dass die Prinzipien und auch die Guidelines allgemeine Empfehlungen bzw. Regeln sind und keine Lösung für ein bestimmtes Problem darstellen.

Der Unterschied der vorgestellten Methoden zu dieser Arbeit ist, dass in der weiteren Ausarbeitung keine Prinzipien oder spezifische Lösungen vorgestellt werden. Die Sicherheit von Softwaresystemen soll erhöht werden, in dem bekannte Schwachstellen und unsichere Lösungen im Software Design gefunden und diese durch ein Pattern oder die vorgestellten Prinzipien behoben werden. Weiter ist zu beachten, dass die Prinzipen, Guidelines und Pattern von dem Softwaredesigner als Person beachtet und umgesetzt werden müssen. Das bedeutet, dieser muss das Wissen haben und umsetzen können. In dieser Arbeit sind eine Automatisierung bzw. maschinelle Verarbeitung das Ziel. Die Anti-Pattern sollen in einer Sammlung definiert werden und in der Entwicklung des Softwaredesigns automatisiert in einem Design erkannt werden. Daher ist Ziel dieser Arbeit, Stellen im Software Design zu finden die nicht den sicheren Prinzipien oder Guidelines entsprechen und stellt somit eine Ergänzung zu den genannten Prinzipien dar, beziehungsweise bauen darauf auf. Da im weitesten Sinne aus diesen die Anti-Pattern abgeleitet und dadurch das Software Design auf die Einhaltung dieser Regeln überprüft werden kann. Weiter wird das Design automatisiert evaluiert und somit ist eine automatische Überprüfung auf die Einhaltung dieser Regeln möglich.

3.3 Security Software Design Anti-Pattern

Die Security Software Design Anti-Pattern stellen das Gegenteil von Security Design Pattern, also eine unsichere Lösung für ein nicht triviales und immer wiederkehrendes Problem dar. Das bedeutet, diese Anti-Pattern stellen eine potentielle Verwundbarkeit oder Schwachstelle dar. Anti-Pattern sind meist in für Menschen verständlicher Sprache definiert und nicht in einer formellen, für Computer maschinell verarbeitbaren Form.

Es gibt erste Ansätze die Anti-Pattern in maschinell verarbeitbarer Form zu definieren und dann automatisiert in Datenflussdiagrammen zu suchen.[26] Dieser Ansatz soll in der weiteren Arbeit aufgegriffen werden, jedoch liegt der Fokus auf Sequenzdiagrammen und nicht auf Datenflussdiagrammen. Daher soll sich nicht nur auf den Datenfluss fokussiert werden, sondern auf allgemeine Ablaufdiagramme wie sie die Sequenzdiagramme darstellen. Weiter soll ein Ansatz entwickelt werden, der nicht auf eine Art von Diagrammen beschränkt ist. Die formale Definitionssprache UML hat verschiedenste Diagrammtypen, sowohl dynamische als auch statische. Es gibt für die verschiedensten Anwendungsfälle das passende Diagramm.[14] Ziel dieser Arbeit ist eine Möglichkeit zu entwickeln, die auf verschiedene Arten der UML Diagramme anwendbar ist. Denn so können für jeden Diagrammtyp Anti-Pattern definiert und automatisiert nach diesen gesucht, sowie im weiteren Verlauf damit Tests zur Stärkung des Software Designs gegen Verwundbarkeiten entwickelt werden. Weiter soll die Möglichkeit der automatischen Evaluation berücksichtigt werden, damit Sammlungen von Anti-Pattern gebildet und auf verschiedene Diagramme angewendet werden können. Damit können bereits bei der Entwicklung und damit bei der Erstellung des Software Designs potentielle Schwachstellen frühzeitig gefunden und geschlossen werden.

4 Analyse der aktuellen Situation und Anforderungen an Anti-Pattern und deren Suche

Nachfolgend soll zunächst die aktuelle Situation dargestellt werden. Danach wird analysiert, welche Anforderungen an Security Design Anti-Pattern und deren Suche im Software Design gestellt werden.

4.1 Aktuelle Situation

Die Sicherheit von Software wird in der vernetzen Welt immer wichtiger. Das liegt an der steigenden Komplexität von modernen Anwendungen, aber auch an der stetig wachsenden Verbindung der Programme durch das Internet.[20] Derzeit gibt es verschiedene Arten um die Sicherheit in der Entwicklung von Software voranzutreiben. Eine ist das Testen der Programme, durch Testfälle, die bekannte und verbreitete Fehler austesten. Beispielsweise die von dem amerikanischen 'National Institute of Standards and Technology' (NIST) bereitgestellte Test Suite Juliet. Diese stellt insgesamt über 80.000 Testfälle zur Verfügung für die Programmiersprachen JAVA und C/C++, durch die verschiedene und verbreitete Sicherheitslücken gefunden werden können.[4] Damit ist die Sammlung von Tests ein gutes Beispiel für die Reduzierung von Sicherheitslücken durch das Testen.

Eine weitere Methode zum Finden von Schwachstellen ist das automatisierte statische Code Review. In diesem wird der Source Code analysiert und nach bekannten bzw. verbreiteten Fehlern oder Schwachstellen gesucht. Ein Beispiel hierfür ist das Programm

SonarQube.[4] Diese Analyse ist Source Code zentriert und lässt dabei meist die Architektur außer acht, es handelt sich dabei um ein automatisiertes Source Code Review. Dies geschieht durch die Überprüfung von Regeln die bestimmte unsichere oder ungewollte Implementierungen definieren, wodurch Verstöße gegen die Coding Guidelines, mögliche Fehler oder potentielle Schwachstellen, entdeckt werden können. Jedoch ist dies nur im Quelltext möglich und es werden auch keine Fehlentscheidungen, die im Design getroffen wurden, entdeckt. Auch ist das Code Review und das Testing erst nach, oder während der Implementierung möglich.

Es gibt jedoch auch Möglichkeiten vor der Implementierung potentielle Sicherheitsschwachstellen zu reduzieren. Damit ist die Berücksichtigung von Security Anforderungen im Software Design gemeint. In der Entwicklung von Software Architekturen und damit in der Erstellung des Designs ist das Beachten von Sicherheitseigenschaften essentiell. Denn hier entstehende Fehler und Sicherheitslücken sind nach der Implementierung meist nur aufwendig korrigierbar. Daher sollte großes Augenmerk auf die Architektur und das damit verbundene Software Design gelegt werden.

Um bei der Planung eine sichere Architektur zu entwickeln, werden verschiedene Techniken angewendet. Eine davon sind die 'Security Design Guidelines' und die 'Security Design Principles'. Diese beinhalten die grundlegenden Regeln und Prinzipien, die eine Software Architektur einhalten muss. Darunter sind Forderungen wie das 'Prinzip der geringsten Privilegien' (Principle of Least Privilege), durch das Rechte immer restriktiv vergeben und diese vor sicherheitskritischen Aktionen überprüft werden müssen.[20]

Eine verwandte Technik sind die Security Pattern. Diese stellen ein wiederverwendbares Lösungsmuster für ein immer wiederkehrendes Problem im Software Design dar und somit eine Art Blaupause für sichere Lösungen. Daher sollten wenn möglich solche Pattern angewendet werden, auch da die Seiteneffekte und Sicherheitseigenschaften von verbreiteten Mustern meist erforscht und bekannt sind.

Nach Abschluss der Erstellung einer Architektur wird meist ein 'Security Design Review' durchgeführt. In diesem analysieren mehrere Fachleute das entstandene Software Design und versuchen Fehler oder Sicherheitslücken zu entdecken.[20]

[4] https://docs.sonarqube.org/latest/user-guide/security-rules/ (aufgerufen am 06.07.2020)

4.2 Anforderungen an Security Design Anti-Pattern

An Security Design Anti-Pattern stellen sich verschiedene Anforderungen, die sich aus unterschiedlichen Aspekten ergeben. Diese leiten sich aus der automatisierten Identifikation und auch aus den Entwurfsmustern ab. Nachfolgend sollen diese Forderungen dargestellt und erörtert werden.

4.2.1 Anforderung der maschinellen Verarbeitbarkeit

Da die Anti-Pattern durch Computersysteme in Programmen entdeckt werden sollen, müssen diese Anti-Pattern maschinell verarbeitbar sein. Das bedeutet, die Struktur und das Speicherformat von den Anti-Pattern muss so gewählt werden, dass ein solches System diese einlesen und verarbeiten kann. Hierbei muss beachtet werden, dass eine graphische Darstellung der Anti-Pattern für Menschen meist besser lesbar und verständlicher ist. Dies gilt jedoch nicht für Computersysteme. Diese können Text, Vektoren und Zahlen viel besser verarbeiten als grafische oder bildliche Darstellungen.

Die Anti-Pattern sollen automatisiert in Softwareprojekten entdeckt werden. Um diese in einem solchen System verwenden zu können, ist es notwendig, dass die Speicherung und Struktur auf die Verarbeitung durch Computersysteme optimiert ist. Das bedeutet, diese müssen in einem standardisierten und spezifizierten Schema gespeichert werden, wie es das XML oder JSON Format darstellen. Somit ist das automatische Einlesen und Speichern ohne Missverständnisse möglich. Weiter wird durch die Spezifikation die Möglichkeit geschaffen, dass das Format auch in einem anderen Programm verwendet werden kann. Dadurch können andere Applikationen auch mit den definierten Anti-Pattern arbeiten und eine Zusammenarbeit mehrerer Anwendungen wird möglich. Dies ist sinnvoll, denn so kann die Suche auch mit anderen Diagrammen zusammenarbeiten, beispielsweise dem Programm, in dem die Software Designs erstellt werden. Durch die Zusammenarbeit mit anderen Anwendungen ergeben sich viele mögliche Anwendungsfälle, die bei der Entwicklung von sicheren Software Designen unterstützen können.

4.2.2 Allgemeine Anforderungen an Security Design Anti-Pattern

Die Autoren Laverdiere, Mourad, Hanna und Debbabi des Papers „Security Design Patterns: Survey and Evaluation“ haben Anforderungen an Security Design Patterns ausgearbeitet, die zum Teil auch auf Security Design Anti-Patterns angewendet werden können:[18]

- **Wiederverwendbarkeit:** Ein Pattern muss in verschiedenen Kontexten angewandt werden können.
- **Kombinierbarkeit:** Die Pattern müssen untereinander kombinierbar sein.
- **Unterscheidbarkeit:** Ein Pattern muss von einem anderen unterscheidbar sein.
- **Vollständigkeit:** Ein Pattern muss vollständig und korrekt spezifiziert sein.
- **Überprüfbarkeit:** Die Verwendung von Pattern muss validierbar sein.
- **Freiheit:** Ein Pattern ist definiert, ohne eine spezifische Programmiersprache oder Framework zu fordern.

Eine essenzielle Anforderung an die Security Design Anti-Pattern ist die Wiederverwendbarkeit. Diese ist wichtig, damit diese nicht nur auf ein spezifisches Design anwendbar sind, sondern allgemein auf verschiedene Softwaredesigns angewendet werden können. Durch diese Eigenschaft wird es möglich, eine Sammlung von Security Design Anti-Pattern zu erstellen, die auf verschiedene Diagramme oder sogar Projekte angewendet werden kann.

Die Kombinierbarkeit sagt aus, dass wenn ein Pattern identifiziert wird, auch andere an derselben Stelle gefunden werden können, denn ein Pattern kann auch als Teil eines anderen Musters sein. Weiter ist es gut vorstellbar, dass ein ’großes’ Pattern aus mehreren einzelnen Teil-Pattern zusammengesetzt ist. Oder auch, dass an derselben Stelle im Diagramm ein anderes Anti-Pattern gefunden wird.

Die Unterscheidbarkeit beschreibt, dass Anti-Pattern voneinander unterschieden werden können. Das bedeutet, diese müssen sich durch einen eindeutigen Namen oder deren Aufbau, voneinander abgrenzen. Diese Eigenschaft ist nötig um das Finden von einem Anti-Pattern erkenntlich zu machen, um eine Zuordnung zwischen dem Erkennen und einem bestimmten Anti-Pattern bilden zu können.

Die Vollständigkeit ist eine wichtige Anforderung für Software Design Pattern. Für Security Design Anti-Pattern ist die Vollständigkeit nur in einem begrenzten Maß eine Anforderung.

Dieses muss nur hinreichend vollständig sein um eine potentielle Schwachstelle erkennen zu können und es muss die Möglichkeit geschaffen werden, dass ein Anti-Pattern offen definiert und nicht vollständig und damit abgeschlossen ist. So ist es möglich, dass Vorhandensein eines beliebigen oder zum Teil definierten Objekts zu beschreiben. Das kann beispielsweise durch die Beschreibung des Objekttyps oder einen Teil des Namens sein, dem ein Objekt entsprechen muss. Solche Anti-Pattern sind nicht vollständig, da nicht alle Komponenten vollständig beschrieben werden, jedoch sind diese hinreichend vollständig, damit eine potentielle Schwachstelle beschrieben wird.

Die Überprüfbarkeit ist eine wichtige Anforderung von Software Design Pattern. Hierbei wird gefordert, dass evaluiert werden kann ob das Muster verwendet wird oder nicht. Dies gilt auch für die Anti-Pattern, jedoch ist die Überprüfbarkeit nur zu einem begrenzten Maß erreichbar, da eine vollständige Überprüfbarkeit nur geschaffen werden kann, wenn ein Anti-Pattern auch vollständig beschrieben ist. Wie ausgeführt, müssen Anti-Pattern nicht vollständig sein, da es so möglich ist nur relevante Teile von Komponenten zu definieren. Daher ist keine vollständige Überprüfbarkeit gefordert, sondern nur eine hinreichende. Dies bedeutet, dass für die beschriebenen Komponenten und deren definierten Eigenschaften überprüfbar sein muss, ob diese mit einer Komponente oder deren Eigenschaften aus einem Software Design übereinstimmen.

Die Eigenschaft der Freiheit von Software Design Pattern sagt aus, dass diese unabhängig von einer Programmiersprache, oder einem verwendeten Framework sein müssen. Dies ist auf die Anti-Pattern nur begrenzt anwendbar, denn diese können durch deren Beschreibung eines Ablaufs oder einer Eigenschaft auf die Objektorientierung zurückgreifen, sowie eine Eigenschaft beschreiben, die es nur in der Objektorientierung gibt, wie die Vererbung. Diese Eigenschaften geben bei der Suche in einem Design einer nicht objektorientierten Programmiersprache wenig Sinn und können zum Teil auch nicht vollständig aus einer objektorientierten Programmiersprache wie JAVA zu einer wie C++ übernommen werden. Daher sind diese nur begrenzt frei und die Forderung nach Freiheit ist begrenzt, da es möglich sein soll, auch Eigenschaften der Objektorientierung zu beschreiben. Jedoch soll die Freiheit soweit gefordert werden, dass kein bestimmtes Framework oder eine bestimmte Programmiersprache gefordert wird, sich jedoch je nach verwendeter Programmiersprache die Anti-Pattern unterscheiden können.

4.2.3 Weitere Anforderungen an Security Design Anti-Pattern

Es müssen auch noch weitere Aspekte berücksichtigt werden die Anforderungen an ein solches Konzept stellen. Denn Sammlungen von Entwurfsmustern müssen erweiterbar sein. Die Erweiterbarkeit ist wichtig, denn eine Sammlung von Security Anti-Pattern muss in Zukunft um neue Anti-Pattern ergänzt werden können. Durch die Forschung, aber auch durch die Evaluation von Softwaredesigns, werden immer neue Muster entdeckt oder spezifiziert die in eine Sammlung von Anti-Pattern aufgenommen werden können. Auch muss ein Austausch von Pattern möglich sein, dieser kann beispielsweise durch neue Entwicklungen einer Programmiersprache gefordert sein, dass bisher als schlecht geltende Muster in Zukunft als annehmbar deklariert werden.

Ein weiterer Aspekt den es zu beachten gilt ist die Flexibilität. Verschiedene Entwurfsmuster fordern unterschiedliche Betrachtungsweisen. So betrachten manche Muster die Beziehungen zwischen verschiedenen Klassen, wie Vererbungseigenschaften. Oder andere das Verwenden mehrerer Bibliotheken oder verschiedener Implementierungen der gleichen Verschlüsselungsfunktion. Dies kann eine potentielle Schwachstelle aufzeigen, da es möglich ist, dass eine der verwendeten Implementierungen einen Fehler beinhaltet und dadurch unsicher ist und nicht aktualisiert wird, da die Verwendung einer zweiten Implementierung in Vergessenheit geraten ist. Daher sollte immer nur eine Umsetzung einer Verschlüsselungsfunktion verwendet werden und nicht mehrere.

Andere Entwurfsmuster hingegen spezifizieren einen sequenziellen Ablauf. Dies kann sein, dass beispielsweise erst eine Verschlüsselungsmethode aufgerufen wird und erst danach eine Methode zur Speicherung von Daten. Wodurch darauf geschlossen werden kann, dass die Daten höchstwahrscheinlich in verschlüsselter Form gespeichert werden und nicht im Klartext. Aus diesen Beispielen wird ersichtlich, das ein Konzept zur Definition von Security Anti-Pattern Flexibilität in der Beschreibung aber auch in der Analyse fordert. Dieser Forderung muss nachgegangen werden, um möglichst viele Security Anti-Pattern definieren zu können. Außerdem muss die genannte Erweiterbarkeit gewährleistet sein.

Die aufgezählten Anforderungen müssen bei der Entwicklung von Security Design Anti-Pattern berücksichtigt werden.

4.3 Anforderungen an die Erkennung von Security Anti-Pattern

Auch an das Erkennen und Auffinden von Security Anti-Pattern im Software Design gibt es Anforderungen. Die Erkennung muss eine eindeutige boolesche Aussage über das Vorhandensein von einem Anti-Pattern liefern. Denn ein solches ist vorhanden oder nicht und darüber muss eine eindeutige Aussage getroffen werden. Dies ist nötig, um das Ergebnis der Suche weiter verwenden zu können, denn in der weiteren Betrachtung wird eine eindeutige Aussage über das mögliche Vorhandensein von sicherheitskritischen Mustern benötigt.

Weiter muss die Lokalität des gefundenen Musters erkennbar sein und die ungefähre Position im Software Design ausgegeben werden, da es zur Beseitigung der potentiellen Schwachstelle wichtig ist wo diese gefunden wurde. Dies ist vor allem in großen Entwürfen notwendig. Auch um das Ergebnis der Suche verifizieren zu können ist es notwendig, eine Angabe über die Lokalität zu bekommen. Denn ohne der Position ist es auch möglich, dass die Suche das Muster an einer Stelle gefunden hat an der dieses nicht vorhanden ist.

Die Zuverlässigkeit ist wichtig für die Suche nach Anti-Pattern. Denn wenn die Suche nur in einem von einhundert Fällen das Muster erkennt, ist diese nicht zuverlässig und auch nicht verwendbar. Weiter ist es im Hinblick auf die Zuverlässigkeit wichtig, dass die false-positiv Rate gering ist. Denn auch wenn die Suche zu oft ein Anti-Pattern erkennt, obwohl es sich nicht um ein Anti-Pattern handelt, ist diese nicht verwendbar. Außerdem ist unter der Zuverlässigkeit zu verstehen, wenn die Suche mehrmals für dasselbe Software Design und denselben Anti-Pattern durchgeführt wird, dass die Suche immer das selbe Ergebnis liefern muss.

Die Performance ist wichtig für eine Suche. Um eine Suche skalierbar und performant zu gestalten, ist es wichtig, dass gleichzeitig nach mehreren Anti-Pattern gesucht werden kann. So kann auf Systemen mit Mehrkernarchitektur die Suche auf verschiedenen Kernen gleichzeitig ausgeführt werden. Das ist wichtig, denn die Anzahl von Anti-Pattern ist nicht begrenzt, es handelt sich um ein offenes System in dem beliebig neue Anti-Pattern hinzugefügt werden können.

4.4 Analyse der benötigten Erweiterungen von Design Pattern zu Anti-Pattern

Die Elemente zum Beschreiben von Software Design Pattern sind nicht ausreichend um Anti-Pattern zu beschreiben. Hierfür muss die Möglichkeit geschaffen werden Elemente nur zum Teil zu definieren, denn nur so ist es möglich, wiederverwendbare Anti-Pattern zu bilden, die in verschiedenen Software Designs identifiziert werden können.

Eine Teilbeschreibung eines Elementes kann der Typ eines bestimmten Elementes sein ohne dessen Namen zu definieren oder Anforderungen an dessen Namen zu stellen. Hiermit kann das Vorhandensein eines beliebigen Objektes eines bestimmten Typs in einem Anti-Pattern gefordert werden.

Durch das Definieren einer Sammlung von Namen, die als möglicher Name eines Elements gesetzt werden können, ist es möglich ein Element nur zum Teil zu beschreiben. Hierdurch ist es möglich, den Typ eines Objekts zu definieren und auch den Namen, der in einer Sammlung von verschiedenen Namen enthalten sein muss. Anhand dieser Beschreibung ist es möglich ein Element zu beschreiben und dabei verschiedene Umsetzungen oder Versionen zu berücksichtigen durch das definieren einer Liste von möglichen Namen.

Ein Element kann auch beschrieben werden aufgrund des Typs und das Fordern, dass ein bestimmtes Wort oder eine bestimmte Zeichenfolge in dem Namen des Elementes enthalten ist. Hierdurch können wiederverwendbare Anti-Pattern beschrieben werden, da es möglich ist, nach einem bestimmten Element zu suchen durch die Definition des Typs und eines Teilnames. Der Teilnamen bietet die Möglichkeit nach einer bestimmten Kategorie von Elementen zu suchen, da diese meist einen übereinstimmenden Teilnamen enthalten, der dessen Funktionalität beschreibt.

Zusätzlich zu der Teilübereinstimmung muss die Möglichkeit geschaffen werden zu definieren, dass ein bestimmtes Element nicht vorhanden ist. Das ein definiertes Element nicht in einem Pattern vorkommt, ist bei der Definition eines Pattern möglich, in dem das Objekt nicht enthalten ist. Dies ist in einem Anti-Pattern nicht der Fall. Denn in einem Anti-Pattern bedeutet das Nichtvorhandensein eines Elementes nicht, dass dieses nicht vorhanden ist, sondern nur das es irrelevant für dieses Anti-Pattern ist. Aufgrund dieser Tatsache muss

die Möglichkeit geschaffen werden auf das Nichtvorhandensein eines Elementes zu prüfen und auch in einem Anti-Pattern zu fordern, dass ein bestimmtes Objekt nicht vorkommt.

Diese beschriebenen Erweiterungen müssen für die Design Pattern geschaffen werden, um Software Design Anti-Pattern beschreiben zu können. Weiter müssen diese Erweiterungen auch in der Suche von Security Design Anti-Pattern berücksichtigt werden. Eine Suche muss auf die Teilübereinstimmung von Objekten oder das geforderte Nichtvorhandensein eines Elementes ausgelegt sein und entsprechende Entscheidungen treffen. Die alleinige Definition in den Anti-Pattern ist wenig hilfreich, wenn ein Suchalgorithmus nicht mit diesen Anforderungen umgehen kann. Bei der Suche handelt es sich um einen Vergleich von UML Diagrammen, diese werden meist als Graphen dargestellt. Somit muss auch eine Möglichkeit gefunden werden, unter der Berücksichtigung des Graph Matching Problems, zwei Datenstrukturen die einen Graphen darstellen zu vergleichen.

Abschließend können die Anforderungen an Security Design Anti-Pattern und deren Suche im Software Design folgendermaßen zusammengefasst werden. Die Anti-Pattern müssen maschinell und automatisiert verarbeitbar sein um eine automatische Prüfung durchführen zu können. Weiter müssen die Anti-Pattern wiederverwendbar, kombinierbar und unterscheidbar sein. Die Vollständigkeit, Überprüfbarkeit und Freiheit sind nur begrenzt gefordert, weil eine bedingungslose Umsetzung dieser Anforderungen bedeuten würde, dass die Anti-Pattern nicht auf verschiedene Software Designs angewendet werden können, weil es nicht möglich wäre, Anti-Pattern zu definieren deren Name nicht vollständig beschrieben ist. Damit Teilnamen, eine Sammlung von Namen oder ein beliebiger Name beschrieben werden kann, darf nicht gefordert sein, dass die Elemente eines Anti-Pattern vollständig sind. Es muss auch berücksichtigt werden, dass die Möglichkeit gegeben ist, Sammlungen von Anti-Pattern zu bilden, die offen für neue oder Änderungen der enthaltenen Muster sind. An die Suche der Anti-Pattern werden folgende Anforderungen gestellt. Diese muss ein eindeutiges Ergebnis liefern und im Fall einer gefundenen Instanz auch die Lokalität anzeigen. Zusätzlich muss die Suche zuverlässig sein und auch Potential bieten mit der Anzahl von zu suchenden Anti-Pattern zu skalieren. Diese Anforderungen sollen im nachfolgenden Konzept berücksichtigt werden.

5 Konzeptionierung von Security Design Anti-Pattern und deren Erkennung im Software Design

Es soll die Möglichkeit geschaffen werden, Modelle aus dem Software Design automatisiert auf mögliche Sicherheitsschwachstellen zu untersuchen. Hierfür sollen Anti-Pattern in dem Software Design maschinell erkannt werden. Um dies zu ermöglichen, ist es nötig verschiedene Aspekte zu betrachten.

Im weiteren Verlauf wird das Vorhandensein von Software Design Modellen angenommen. Diese können durch Software Architekten in der Design bzw. Architektur Phase der Entwicklung entstanden sein, oder auch durch die automatisierte Erstellung aus vorhandenen Quelltexten. Das Generieren von Design Modellen auf der Grundlage von Source Code soll kein Teil dieser Arbeit sein, da im nachfolgendem der Fokus auf das Konzeptionieren von Anti-Pattern und die Suche derer in Design Modellen liegt. Auch ist diese automatische Erzeugung noch Gegenstand von Forschungsprojekten und anderen Arbeiten. Dies ist kein triviales Problem.[24] Vor allem die Ableitung von Sequenzdiagrammen aus dem Quelltext ist für größere und komplexe Softwareprojekte noch nicht vollständig gelöst. Es wird jedoch an einer entsprechenden Technik geforscht.[2, 5, 27]

Weil im Weiteren das Konzept auch beispielhaft für die Sequenzdiagramme implementiert werden soll, ist die automatische Generierung von UML Diagrammen aus dem Source Code kein Teil dieser Arbeit, da es den Umfang überschreiten würde und hier der Fokus auf der Definition und Suche von Security Design Anti-Pattern liegt.

5.1 Überblick des Gesamtkonzeptes

Ziel der Arbeit ist es Security Design Anti-Pattern definieren zu können und diese in einem Software Design zu identifizieren. Aufgrund dieser Technik ist es möglich potentielle Schwachstellen einer Anwendung vor der Implementierung zu identifizieren und Entwurfsfehler vor der Umsetzung zu erkennen.

Dafür ist es nötig, Security Design Anti-Pattern definieren zu können, um diese in verschiedenen Software Designs zu suchen. Dabei müssen Anti-Pattern die Möglichkeit bieten, ein Element eines Diagramms nur zum Teil zu beschreiben. Dies soll in Abschnitt 5.3 genauer erläutert werden. Weiterführend muss ein Suchalgorithmus konzeptioniert werden dem es möglich ist, Anti-Pattern in einem Software Design zu identifizieren. Das geschieht in Kapitel 5.4 Dazu ist es notwendig, eine Datenstruktur zu finden, in der das Software Design und die Anti-Pattern verarbeitbar sind. Diese wird in Abschnitt 5.2 vorgestellt.

Der Ablauf ist folgender. Das Konzept liest zuerst das zu überprüfende Software Design ein. Anschließend wir ein Suchalgorithmus gestartet, der die Security Design Anti-Pattern aus einer Sammlung von Anti-Pattern einließt und danach in dem Software Design sucht. Die Suche liefert die gefundenen potentiellen Schwachstellen in dem Software Design als Ergebnis zurück. Dies soll bildlich in der Abbildung 3 dargestellt werden.

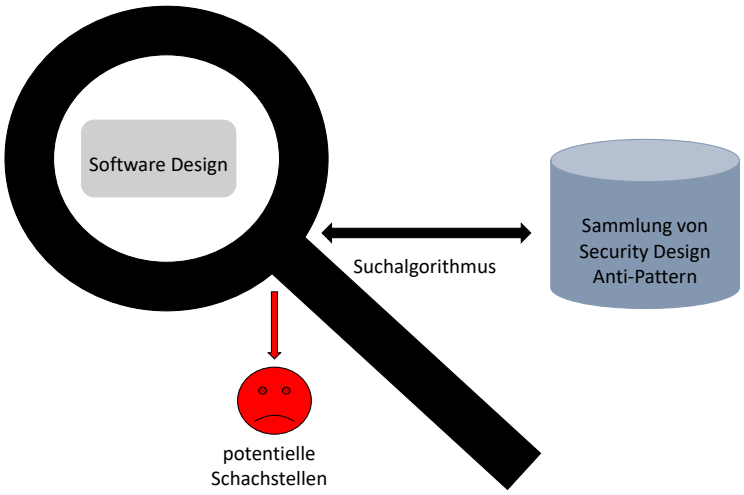

Abbildung 3: Allgemeiner Aufbau des Konzeptes

5.2 Verarbeitung von Software Design Modellen

Die Software Design Modelle sind in UML spezifiziert. Hierdurch ist es möglich diese in einem standardisierten Format einzulesen. UML Modelle zu verwenden ist aufgrund der weiten Verbreitung sinnvoll. Ein weiteres Argument für die Verwendung von UML ist, dass es sich als weltweiten Standard in der Softwarebranche durchgesetzt hat.[14] Zudem sollen die Modelle in einer Baumstruktur verarbeitet werden. In der Spezifikation von UML Diagrammen sind die einzelnen Elemente in 'Knoten' und 'Pfade' unterteilt.[22] Außerdem ist es für die weitere Verarbeitung sehr sinnvoll diese Unterteilung zu übernehmen, dies wird am Beispiel der Sequenzdiagramme ersichtlich. Da jedes Element ein Elternelement hat, gibt es Elemente die weitere Kindelemente haben können (Knoten), aber auch Elemente, die keine zusätzlichen Kinder haben, jedoch zwei Elemente miteinander verbinden (Pfade). Die Lebenslinie stellt hierbei auch einen Knoten dar, der ein Kindelement von dem Sequenzdiagramm ist. Das Element Sequenzdiagramm repräsentiert den Wurzelknoten und stellt damit eine Spezialform von Konten dar. Das Wurzelelement eines Baums hat beliebig viele Kindelemente, wie alle Knoten, jedoch hat er kein Elternelement da dieser den Ursprung des Modells darstellt.

Diese Struktur kann auf alle UML Diagramme angewendet werden, da die Elemente immer in Knoten und Pfade unterteilt werden können. Daher kann für jedes UML Diagramm eine Baumstruktur erzeugt werden. Es stellt eine strukturierte Datenhaltung dar, durch die es möglich ist, Muster innerhalb des Baums oder Teilbäume zu suchen. Bei der Suche in dieser Struktur gibt es folgendes zu beachten. Ein Knoten kann als 'identisch' zu einem Knoten in einem Pattern angesehen werden, wenn alle Kindknoten des Pattern-Knotens 'identische' Knoten in der Menge der Kindknoten des Modell-Knotens haben. Dies soll in Kapitel 5.4 genauer erläutert werden.

Die Anti-Pattern stellen Ausschnitte aus einem Software Design Modell dar. Deshalb können diese auch in UML definiert und in einem standardisierten Format gespeichert werden. Es sind jedoch Erweiterungen des normalen Standards nötig, um Anti-Pattern sinnvoll darstellen zu können. Mehr hierzu in Kapitel 5.3. Jedoch bedeutet dies, dass die Anti-Pattern, wie die Design Modelle, in einer Baumstruktur aufgebaut werden können. Dies bietet den Vorteil, dass die Pattern sehr ähnlich zu den Modellen verarbeitbar sind.

Im weiteren Konzept wird das vollständige Software Modell geladen, bei umfangreichen Modellen wird dadurch ein größerer Bedarf an Hauptspeicherkapazität verursacht, jedoch ist die weitere Verarbeitung schneller und einfacher, da keine Teile des Modells nachgeladen werden müssen. Es besteht die Möglichkeit das Konzept zu erweitern, sodass Teile des Modells dynamisch geladen werden. Dies ist ebenfalls kein Gegenstand dieser Arbeit, da im weiteren Verlauf der Fokus auf die grundlegende Funktionalität und die Suche gelegt wird.

5.3 Konzept von Anti-Pattern

Die Anti-Pattern stellen einen Ausschnitt aus einem gesamten UML Diagramm dar. Dieser repräsentiert eine potentielle Sicherheitsschwachstelle. Wie in Kapitel 4.4 dargestellt wurde, muss die Möglichkeit geschaffen werden ein Anti-Pattern nicht vollständig spezifizieren zu müssen, um die Portabilität auf verschieden Software Designs anwenden zu können. Um Anti-Pattern so zu definieren, dass ein Name eines Objekts nicht vollständig definiert ist, muss der UML Standard um folgende Operatoren erweitert werden.

- **Wildcard Operator:** Steht für einen beliebigen Text als Namen eines Elements.
- **Wörterbuch Operator:** Durch diesen ist es möglich eine beliebige Liste von Namen zu definieren, die für ein Element angewendet werden kann.
- **Enthält Operator:** Durch diesen ist es möglich nach einer Teilübereinstimmung zu suchen. Beispielsweise soll der Name eines Elements den Subtext 'encrypt' enthalten.
- **Nicht Operator:** Dieser beschreibt, dass ein spezifiziertes Element nicht enthalten sein soll. Wenn dieses vorkommt kann in dem aktuellen Pfad die Suche abgebrochen werden.

5.3.1 Wildcard Operator

Durch den Wildcard Operator ist es möglich das Vorkommen eines bestimmten Elementtyps zu fordern ohne Eigenschaften über dessen Namen machen zu müssen. Dies kann beispielsweise in einem Sequenzdiagramm verwendet werden, wenn der Name einer Lifeline unwichtig ist, jedoch auf das aufeinanderfolgen von zwei Funktionsaufrufen geachtet

werden soll, welches in Abbildung 4 verdeutlicht wird. In dieser ist zu sehen, dass der Name der Lifeline irrelevant ist, und das Muster das Aufeinanderfolgen der Funktion_1 und der Funktion_2 spezifiziert. Es ist für jedes Element möglich den Wildcard Operator als Namen zu setzen. So kann auch eine Ausführung oder eine Nachricht den Wildcard Operator als Namen nutzen. In der weiteren Arbeit wird der Wildcard Operator als '*' abgekürzt bzw. dargestellt.

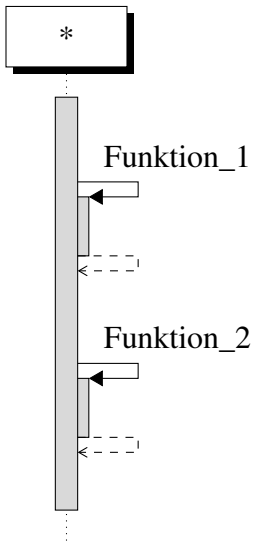

Abbildung 4: Beispiel für die Verwendung des Wildcard Operators

5.3.2 Wörterbuch Operator

Durch die Erweiterung des Wörterbuch Operators ist es möglich mehrere mögliche Namen für ein Element zu spezifizieren. So ist es möglich, eine Liste von Namen für ein Element anzugeben. Bei der Suche wird überprüft, ob ein Name aus der Liste mit dem des Elements übereinstimmt. Ist dies der Fall, ist das Element gefunden. Der Vorteil des Wörterbuch Operators ist, dass viele verschiedene und mögliche Namen angegeben werden können. Dadurch wird die Wiederverwendbarkeit eines Patterns und die Wahrscheinlichkeit des Findens in einem Modell das es enthält erhöht. Um einen Wörterbuch Operator in UML einbinden zu können, muss ein bestimmtes Schema für den Namen von Wörterbüchern erstellt werden. Weiter muss beim Einlesen von UML Modellen ein neuer Abschnitt im standardisierten Format eingefügt werden der die Wörterbücher enthält. Bei dem eingelesenen Anti-Pattern muss dann die Wörterbuch Eigenschaft gesetzt werden. In der

fortlaufenden Arbeit wird der Wörterbuch Operator mit folgendem Schema erkennbar. Der Name startet mit einem '$' Zeichen, gefolgt von dem Namen des Wörterbuchs und endet mit einem '$' Zeichen. Dargestellt soll dies in der Abbildung 5 werden.

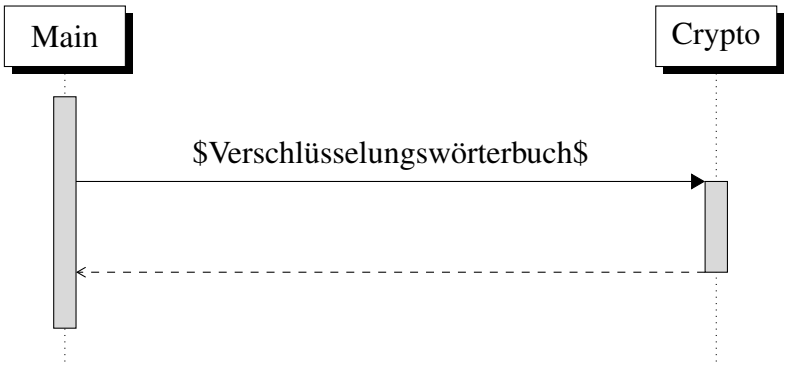

Abbildung 5: Beispiel für die Verwendung des Wörterbuch Operators

In der Abbildung 5 ist zu erkennen, dass nach einem Aufruf von Main in Crypto gesucht wird und verschiedene Möglichkeiten für den Namen der gesuchten Funktion in dem Wörterbuch 'Verschlüsselungswörterbuch' enthalten sind.

5.3.3 Enthält Operator

Die Erweiterung des Enthält Operators ermöglicht die Überprüfung auf eine Teilübereinstimmung. Hierbei kann ein Teil eines Namens spezifiziert werden, der in dem Namen des Elements enthalten sein muss. Es ist zu beachten, dass die Groß- bzw. Kleinschreibung ignoriert wird. Durch diesen Operator ist es möglich die Wiederverwendbarkeit von Anti-Pattern und auch die Robustheit bei Änderungen zu steigern, da auf das Enthaltensein von spezifischen Subnamen geprüft wird. Zum Beispiel ist in den meisten Funktionen die eine Verschlüsselung vornehmen der Text 'encrypt' enthalten. Durch den Enthält Operator ist es möglich, spezifisch nach diesem Subnamen zu suchen. Wie auch bei dem Wörterbuch Operator ist die Erkennung, dass dieser verwendet werden muss, wichtig. Dies geschieht auch über ein Schema für die Angabe des Subnamen. Weiter muss beim Einlesen des Modells das Schema und damit der Subname erkannt und die Enthält Eigenschaft des Elements gesetzt werden. In der weiteren Arbeit wird der Enthält Operator mit folgendem Schema erkennbar. Der Name startet mit einem '€' Zeichen, gefolgt von dem Subnamen welcher

enthalten sein soll und endet mit einem '€' Zeichen. Dargestellt soll dies beispielhaft in der Abbildung 6 werden.

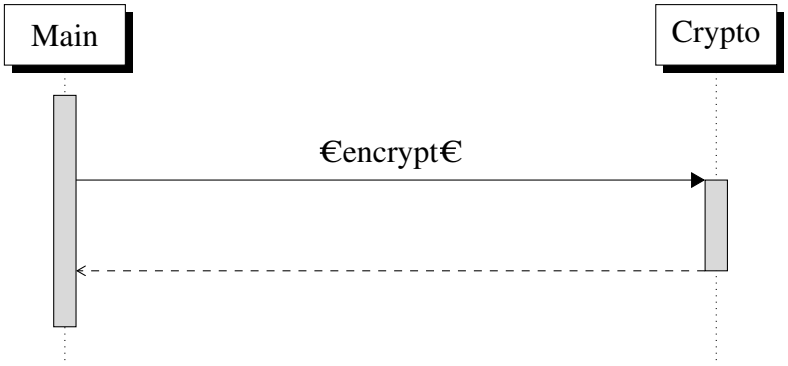

Abbildung 6: Beispiel für die Verwendung des Enthält Operators

In diesem Beispiel ist zu erkennen, dass nach einem Muster gesucht wird, in dem von Main eine Nachricht oder Funktionsaufruf an Crypto gesendet wird in dessen Namen 'encrypt' enthalten ist. Aus dem Aufruf einer Funktion die den Namen 'encrypt' enthält könnte darauf geschlossen werden, dass eine Verschlüsselungsfunktion aufgerufen wird. An diesem Beispiel ist zu erkennen, dass durch die Spezifikation von Subnamen es möglich ist wiederverwendbare Anti-Pattern zu bilden, die auf verschiedenste Software Design Modelle angewendet werden können.

5.3.4 Nicht Operator

Der Nicht Operator erweitert die UML Notation, um die Funktionalität auf das Nichtvorhandensein eines bestimmten Elements zu prüfen. Hierbei sind der Name und die Art des Elements entscheidend. Durch den Nicht Operator ist es möglich, innerhalb von Abfolgen zu prüfen ob wichtige Schritte fehlen, die eine Verwundbarkeit ausschließen würden, oder die Sicherheit erhöhen könnten. Dies ist bei der Definition vieler Security Anti-Pattern wichtig, da für diese oft auf das Nichtvorhandensein geprüft wird. Jedoch bietet ein Nicht Operator Herausforderungen in der Suche von Anti-Pattern. Dies soll in Abschnitt 5.4 erläutert werden. Um in einem Modell den Nicht Operator für ein Element anzuwenden, ist ein Schema für den Namen notwendig. Dieses wird bei dem Einlesen des Modells erkannt und daraufhin die Eigenschaft des Nicht Operators bei dem aktuellen Element gesetzt. In der weiteren Arbeit ist der Nicht Operator durch folgendes Schema erkennbar. Der Name

des Elements startet mit einem '!' Zeichen, darauf folgt der Name des Elements und endet mit einem '!' Zeichen. Dargestellt ist dies graphisch in Abbildung 7.

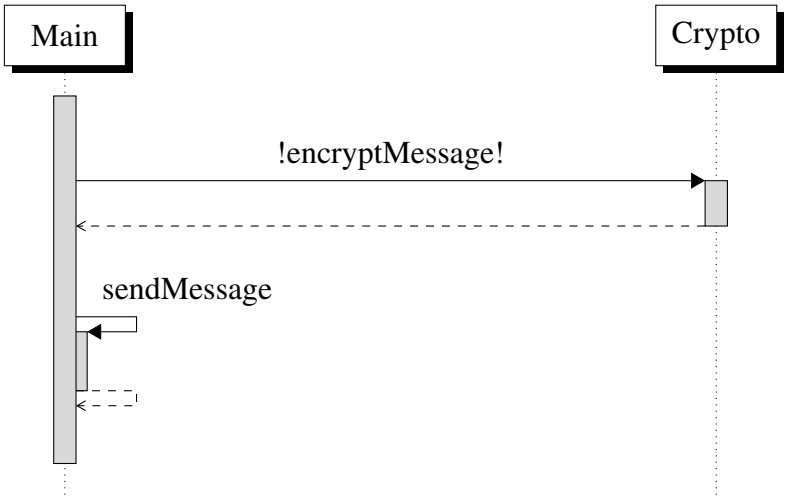

Abbildung 7: Beispiel für die Verwendung des Nicht Operators

In dem Beispiel ist zu erkennen, dass nach einem Muster gesucht wird, wobei eine Nachricht gesendet wird, jedoch zuvor nicht die spezifische Funktion 'encryptMessage' aufgerufen wurde. Daraus lässt sich schließen das eine Nachricht im Klartext gesendet wird. Dies kann einen ungewollten Informationsabfluss darstellen und ist dadurch eine Verwundbarkeit. Hierbei ist zu erkennen wie wichtig der Nicht Operator ist, da ohne diesen es nicht möglich ist zu prüfen, ob die Nachricht verschlüsselt wurde bevor diese gesendet wird.

5.3.5 Die Kombination von Operatoren

Volles Potential können die vorgestellten Erweiterungen jedoch erst in der Verbindung entfalten, denn es muss möglich sein, die verschiedenen Erweiterungen zu kombinieren. Dadurch kann die Wiederverwendbarkeit und auch die Möglichkeit Anti-Pattern zu spezifizieren massiv erhöht werden.

Ein Beispiel für das Verbinden von Erweiterungen ist die Kombination aus dem Wörterbuch und Enthält Operator. Folglich wird es möglich, nach dem Enthaltensein von einem Subnamen aus dem Wörterbuch zu suchen. Das bedeutet, in einem Wörterbuch können mögliche Subnamen für einen Typ von Funktionen vorgehalten werden und durch den Enthält Operator wird geprüft, ob einer dieser Subnamen enthalten ist. Beispielsweise könnten

so verschiedene Subnamen für Verschlüsselungsfunktionen in verschiedenen Sprachen in einem Wörterbuch definiert werden, wie 'encrypt' und 'verschlüsselung'. Weiter wird durch den Enthält Operator abgeprüft ob einer der definierten Subnamen enthalten ist. Dies erhöht die Wiederverwendbarkeit des Pattern da verschiedene Sprachkonventionen in einem Pattern abgebildet werden können.

Es ist auch möglich den Nicht Operator mit einem der anderen Operatoren zu verbinden. So kann auch auf das Nichtvorhandensein eines Subnamen aus einem Wörterbuch geprüft werden, welche eine Verbindung aus dem letzten Beispiel mit dem Nicht Operator darstellen würde. Hierbei ist zu erkennen, dass durch die Kombination die Wiederverwendbarkeit sowie die Möglichkeiten enorm gesteigert werden. Ein Beispiel mit der Kombination aus verschiedenen Operatoren soll in Abbildung 8 gezeigt werden.

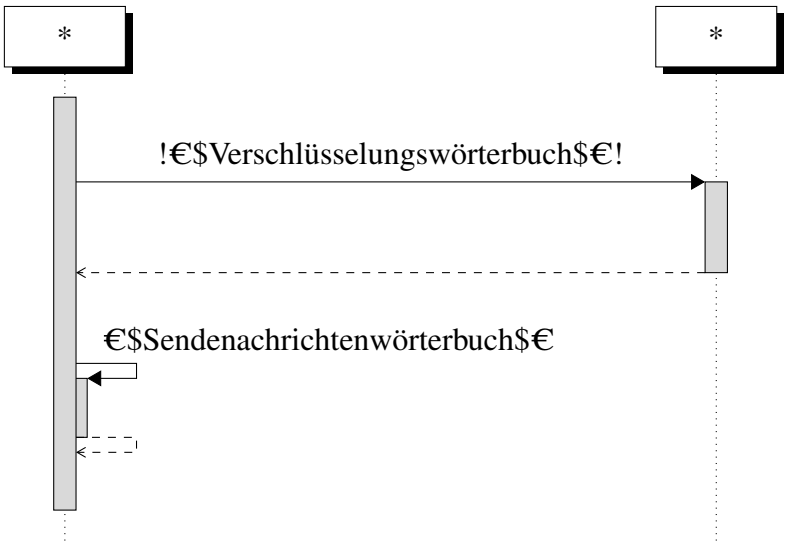

Abbildung 8: Beispiel für die Kombination aus verschiedenen Operatoren

In diesem Beispiel ist zu erkennen, dass beide Lifelines den Wildcard Operator anwenden wodurch der Name der sendenden und empfangenden Lifeline irrelevant wird und auf jede Lifeline in einem Sequenzdiagramm zutrifft. Darauffolgend ist eine Kombination aus dem Nicht, Enthält und Wörterbuch Operator. Diese soll prüfen, dass keine Funktion aufgerufen wird die einen Subnamen aus dem Verschlüsselungswörterbuch enthält. Weiter prüft die Verbindung aus Enthält und Wörterbuch Operator ob ein Funktionsaufruf einen Subnamen aus dem Sendenachrichtenwörterbuch beinhaltet. Insgesamt werden dadurch Stellen im Sequenzdiagramm gesucht, in denen eine Nachricht versendet wird, ohne dass zuvor eine Verschlüsselungsfunktion aufgerufen wird.

Diese Erweiterungen können nicht nur auf Sequenzdiagramme angewendet werden, sondern auch auf andere Modelltypen von UML. Dadurch stellen die Erweiterungen ein wichtiges Element in der Definition von Security Anti-Pattern dar. Auch weil diese die Wiederverwendbarkeit erhöhen, da durch diese die Anti-Pattern allgemeiner spezifiziert werden können und nicht auf ein Modell zugeschnitten sind. Weiter bieten diese viele Möglichkeiten auch komplexere Muster in der Software Architektur bzw. Design zu erkennen und auch potentielle Schwachstellen zu finden. Durch die Erweiterungen ist eine allgemeine Definition möglich, die benötigt wird, um eine Testsuite mit verschiedenen Anti-Pattern aufzubauen, die auf verschiedenste Software Designs angewendet werden kann.

5.4 Konzept zur Suche in der Baumstruktur

Die Suche von Teilbäumen in einer Baumstruktur ist nicht trivial, vor allem wenn die Teilübereinstimmung berücksichtigt werden soll. Dies liegt an dem Graph Matching Problem.[19] Daher muss ein Weg gefunden werden effizient nach übereinstimmenden Unterbäumen zu suchen. Bei dieser Suche sollen Anti-Pattern gefunden werden, die wie in Abschnitt 5.3 beschrieben verschiedene Erweiterungen beinhalten können. Deshalb ist es möglich nach einer Teilübereinstimmung zu suchen. Weiter muss die Suche berücksichtigen, dass ein Kindelement eines Patternelements im Diagramm kein direktes Kindelement ist, sondern ein Kindelement eines Kindelements oder sogar noch tiefer liegt.

Für die Suche muss eine allgemeine Definition zur Übereinstimmung von einem Element des Modells und einem Element des Patterns gefunden werden. *Ein Element eines Modells ist zu einem anderen Element in einem Pattern gleich, wenn der Typ des Elements derselbe ist, der Namen übereinstimmt und alle Kindelemente des Patternelements einen nach dieser Definition gleichen Repräsentanten in der Menge der Kindelemente des Modellelements haben.* Hierbei ist zu beachten, dass die Definition rekursiv ist, da die Übereinstimmung für alle Kindelemente überprüft werden muss. Daher wird bei einer Überprüfung bis in das letzte Blattelement des Patternbaums geprüft. Dieses kann ein Knoten ohne weitere Kindelemente, oder ein Pfad sein.

Aus dieser Tatsache lässt sich ableiten, dass die Überprüfung für große Modelle und Pattern rechenintensiv ist. Da für die Überprüfung von einem Knoten alle Kindelemente des Patternknoten überprüft werden müssen und eventuell auch rekursiv alle Kindelemente des Modells bis zu den Blättern falls zuvor keine Übereinstimmung gefunden wurde. Eine Ausnahme existiert, diese sorgt dafür, dass kein weiteres Kindelement in dem aktuellen Pfad geprüft werden muss. Bei dieser handelt es sich um eine Übereinstimmung mit einem Patternelement, dass die Nicht Erweiterung beinhaltet. Denn wenn eine solche gefunden wurde, muss die Suche an dieser Stelle abgebrochen und an anderer Stelle, vor der Übereinstimmung, fortgesetzt werden. Folglich kann die Verwendung des Nicht Operators zu einer Beschleunigung des Suchvorgangs führen, da möglicherweise größere Teile des Baums nicht durchsucht werden müssen weil die weiteren Elemente des Anti-Pattern irrelevant sind, da ab diesem Punkt das Anti-Pattern nicht mehr erfüllt werden kann.

Für die restlichen Erweiterungen sind die Besonderheiten der Suche sehr begrenzt. Bei dem Wildcard Operator wird die Überprüfung des Elements auf den Typ und die Übereinstimmung der Kindelemente begrenzt. Die Prüfung des Namens ist in diesem Fall immer positiv, da der Wildcard Operator einen Platzhalter für jeden möglichen Namen darstellt. Für die restlichen Erweiterungen (Wörterbuch und Enthält Operator) gibt es keine Einschränkungen oder Besonderheiten die bei der allgemeinen Suche beachtet werden müssen. Diese Erweiterungen erfordern nur Aufmerksamkeit bei dem Vergleich des Namens von zwei Elementen. Es kann der allgemeine Suchablauf wie in Abbildung 9 zusammengefasst werden.

In Abbildung 9 ist zu erkennen, dass die Suche mit einem Modellknoten und einem Patternknoten startet. Diese sind jeweils die Wurzelknoten der beiden Baumstrukturen. Um die Übereinstimmung der Strukturen bilden zu können, wird zuerst ein direkter Vergleich der beiden Knoten durchgeführt. Hierbei sind der Typ und der Name von Bedeutung. Falls keine Übereinstimmung gefunden werden kann, ist der Suchalgorithmus an diesem Punkt fertig mit dem Ergebnis 'keine Übereinstimmung'. Im anderen Fall wird geprüft ob der Patternknoten Kindknoten besitzt. Falls nicht ist die Suche mit dem Ergebnis 'Übereinstimmung' abgeschlossen. In anderem Fall muss geprüft werden, ob das Modellelement Kindknoten besitzt. Falls nicht ist die Überprüfung mit dem Ergebnis 'keine Überein-

stimmung' abgeschlossen. Im anderen Fall wird die Menge aller Kindknoten von Modell- und Patternknoten gebildet. Darauffolgend wird versucht für jeden Patternknoten einen übereinstimmenden Modellknoten zu finden. Hierfür muss jeder Patternknoten solange mit Modellknoten verglichen werden, bis eine Übereinstimmung gefunden wurde. Ist diese gefunden wird der nächste Patternknoten mit den übrigen Modellknoten verglichen bis ein positiver Vergleich vorliegt. Dies wird solange wiederholt bis es keine Patternknoten mehr gibt ('Übereinstimmung') oder es keine Modellknoten mehr gibt ('keine Übereinstimmung'). Um die Knoten miteinander vergleichen zu können ruft der Suchalgorithmus rekursiv sich selbst mit dem Modell- und Patternknoten, die miteinander verglichen werden sollen, auf.

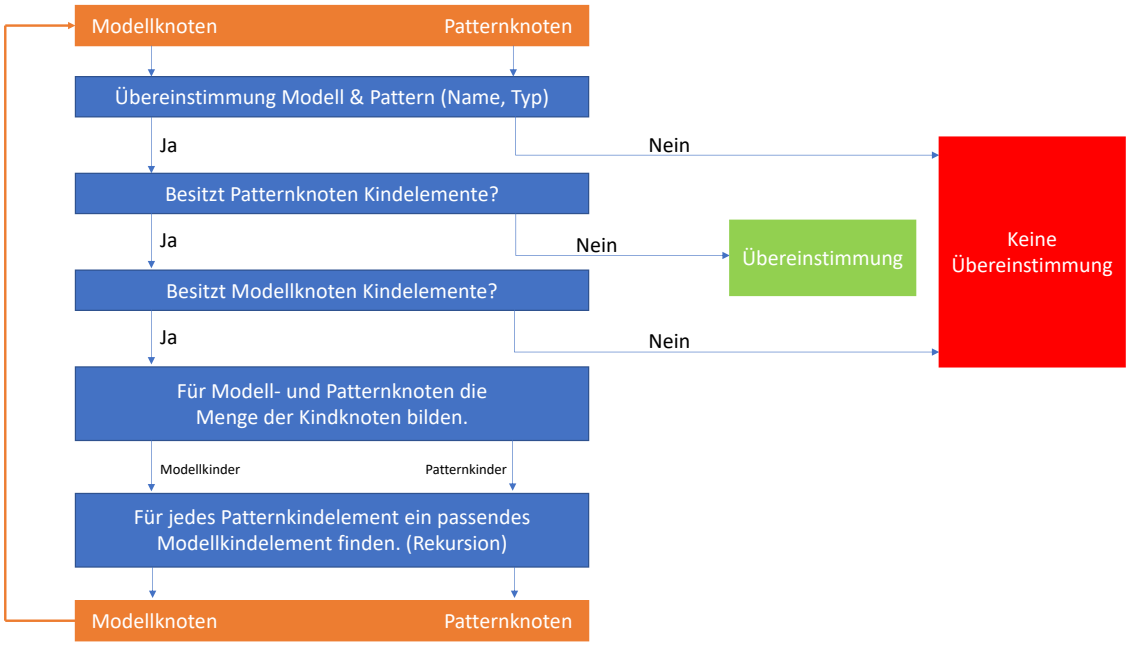

Abbildung 9: Allgemeiner Aufbau des Suchalgorithmus

Bei dem Vergleich der Kindknoten von Modell- und Patternknoten ist zu beachten, dass die Reihenfolge erhalten bleiben muss. Das bedeutet, es wird über die Patternknoten iteriert und auch über die Modellknoten. Wird eine Übereinstimmung gefunden sind nur noch Kindknoten des Modells zur Verfügung, die nach dem übereinstimmenden Knoten kommen. Wird nicht für jeden Patternkindknoten eine übereinstimmender Modellkindknoten gefunden, kann die Suche auf der nächsten 'Ebene' fortgesetzt werden. Das bedeutet die Suche für die aktuellen Patternkindknoten wird für die Kindknoten aller Modellkindknoten

gestartet. Dies bietet den Vorteil, dass eine Teilübereinstimmung gefunden werden kann. Möglich wird es dadurch, dass es für die Pattern irrelevant ist ob zwischen den definierten Knoten andere unspezifizierte Knoten kommen. Denn dürften diese nicht gefunden werden, müssten Sie definiert und die Nicht Eigenschaft gesetzt sein. Hierbei würde bei einer Übereinstimmung mit der Nicht Eigenschaft die Suche abgebrochen und auch nicht für die Kindknoten der Kindknoten die weitere Suche gestartet werden.

Bei der Übereinstimmung von Pfaden gilt es zu beachten, dass nicht nur der Namen und Typ übereinstimmen müssen da ein Pfad keine Kindelemente hat. Weil ein Pfad meist zwei Punkte verbindet (Ausnahme sind gefundene bzw. verlorene Nachrichten), muss ausgehend von dem zweiten Punkt die Übereinstimmung der Elternknoten des Pfades und des Pattern überprüft werden. Dies soll anhand von einem Beispiel in Abbildung 10 verdeutlicht und erklärt werden.

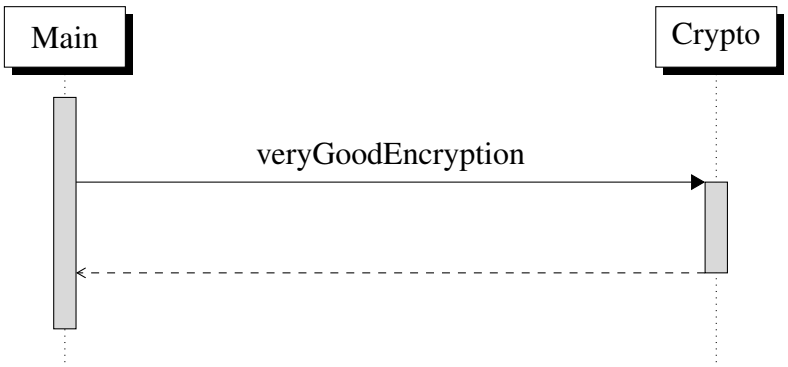

Abbildung 10: Beispiel für die Pfadüberprüfung

Es ist ein Pattern enthalten, welches in einem Software Design gesucht werden soll. Hierbei ist zu erkennen das eine Stelle gesucht wird, in der die Lebenslinie 'Main' eine Nachricht 'veryGoodEncryption' an die Lebenslinie 'Crypto' sendet. Vorausgesetzt der Suchalgorithmus startet an der Lebenslinie 'Main' und findet eine Nachricht mit dem Namen 'veryGoodEncryption', dann ist dies noch kein Treffer, sondern es muss erst überprüft werden ob die Nachricht auch zu der Lebenslinie 'Crypto' geht oder zu einer Ausführung, die ein Kindelement von dieser ist. Um diese Überprüfung bewerkstelligen zu können ist auch ein Rekursiver Algorihtmus nötig, der die Elternelemente des verbunden Knoten mit den Elternelementen des Pattern vergleicht.

5.5 Konzept zur Erweiterung der Suche

Die in Abschnitt 5.4 vorgestellte Suche macht es möglich definierte Muster in einem Software Design zu finden. Jedoch kann die Suche noch erweitert werden um mehrere Vorkommen zu finden und diese auch über verschiedene Hierarchien zu erkennen. Die Grundsuche versucht Knoten eines Pattern, die sich auf derselben Ebene befinden, auch auf einer Ebene im Modell zu finden. Dies ist nicht immer sinnvoll und es sollte die Möglichkeit geschaffen werden über mehrere Ebenen aufgeteilt die Pattern zu finden. Diese Erweiterungen können zum Teil nicht auf alle Diagrammtypen von UML angewendet werden.

Die erste Erweiterung ist das Abzweigen bei einer Übereinstimmung von Modell- und Patternknoten. Hierbei ist der Gedanke die Suche bei einer Übereinstimmung, wie in der Grundsuche, fortzusetzen mit der Suche nach dem nächsten Patternknoten. Aber gleichzeitig die Suche fortzusetzen, wie wenn es keine Übereinstimmung gegeben hätte. Durch die Suche als gäbe es noch keine Übereinstimmung, ist es möglich das Muster nochmals an anderer, späterer Stelle im Baum zu entdecken. Weiter ist es hierdurch möglich das Vorkommen eines Musters öfter in einem Design zu erkennen und trotzdem zu finden, wenn diese sich überschneiden. Diese Erweiterung beschreibt eine Ergänzung der allgemeinen Suchstrategie und ist für die Suche in allen Arten von Diagrammen einsetzbar.

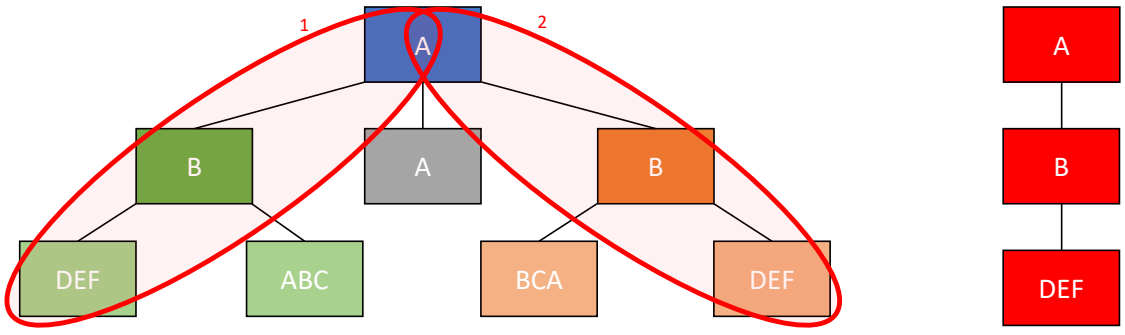

Abbildung 11: Beispiel für die Erweiterung der Suche zum Finden mehrerer Instanzen

In Abbildung 11 soll dies verdeutlicht werden. Links ist ein Baum mit verschiedenen Objekten zu sehen der ein Software Design repräsentiert und rechts ein weiterer Baum mit Objekten der ein Anti-Pattern darstellen soll. Das Anti-Pattern ist in dem Baum zweimal enthalten (rot umkreist). Die grundlegende Suche würde nur das erste Vorkommen des

Anti-Pattern entdecken (Blau → Grün), da diese nach der Übereinstimmung des Knotens 'B' nach einem Kindknoten von 'B' sucht mit dem Text 'DEF'. In der erweiterten Version der Suche wird nicht nur nach diesem Knoten gesucht, sondern auch die Suche fortgesetzt als wäre noch keine Übereinstimmung entdeckt worden. Dadurch wird dann auch das zweite Vorkommen (Blau → Orange) entdeckt.

Die zweite Erweiterung ist das Abzweigen bei jedem Modellknoten. Die Idee dieser Erweiterung ist, nicht wie in der Grundsuche nur die weiteren Nachbarknoten zu überprüfen, sondern die Patternknoten die in der Ebene gefunden werden sollen, einerseits wie bisher in der Ebene zu suchen, aber auch gleichzeitig die Suche nach den restlichen Patternknoten für die Kindelemente des Knotens zu starten. Das bedeutet, die Suche überprüft wie bisher die Nachbarknoten auf eine Übereinstimmung, jedoch zusätzlich auch die Kindknoten. Durch diese Verzweigung und auch weitere werden mehr mögliche Wege gegangen und aus diesem Grund auch mehr Muster entdeckt. Jedoch ist diese Art der Verzweigung nicht für alle Arten von Diagrammen im Software Design sinnvoll. Für Sequenzdiagramme ist diese nützlich, für manch andere Arten weniger. In Sequenzdiagrammen ist es irrelevant ob eine Funktion selbst eine Funktionalität aufruft, oder eine Unterfunktion die diese Funktionalität ausführt. Daher muss für jede Art von Diagrammen entschieden werden ob die Einbeziehung der Kindelemente oder die Übertragung von Eigenschaften auf die Kindelemente in der Suche berücksichtigt werden soll.

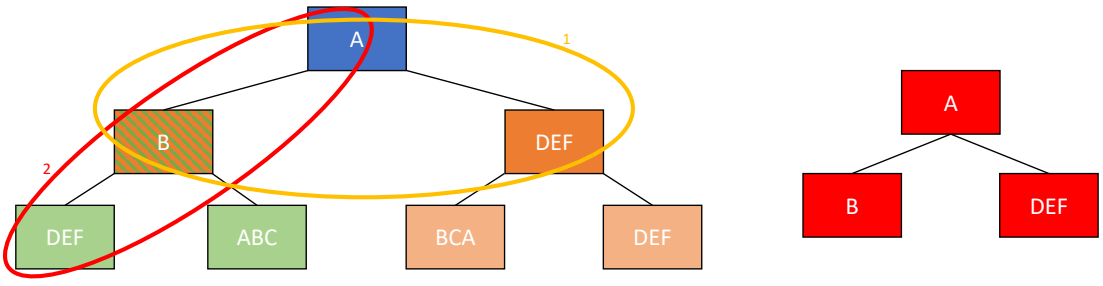

Abbildung 12: Beispiel für die Erweiterung der Suche zum Finden in Unterebenen

Die Erweiterung zum Finden von Suchergebnissen soll in Abbildung 12 gezeigt werden. Links ist ein Baum mit verschiedenen Objekten zu sehen der ein Software Design repräsentiert und rechts ein weiterer Baum mit Objekten der ein Anti-Pattern darstellt. Das Anti Pattern ist in dem Baum zweimal enthalten, wenn die Ebene irrelevant ist. In Sequenz-

diagrammen ist dies wie bereits erläutert der Fall. Daher ist für diese irrelevant ob 'DEF' das nächste Element der Ausführung 'A' oder 'B' ist. Entscheidend ist der Umstand, dass 'DEF' nach 'B' ausgeführt wird. Die grundlegende Suche würde nur das gelb umrandete Muster entdecken. Durch die Erweiterung der Suche um die Prüfung in der Unterebene und in der Selbigen ist es möglich das rot umrandete Vorkommen auch zu finden.

Die Erweiterungen bieten die Möglichkeit das zu suchende Muster mehrfach in einem Software Design zu entdecken und auch zu finden wenn die Ebenen nicht übereinstimmen. Dies ist wichtig, da mehrere Instanzen eines Anti-Patterns gefunden werden sollen, da jede Instanz eine Sicherheitsschwachstelle darstellen kann. Die Suche über mehrere Ebenen ist wichtig, da es für das Ablaufen irrelevant ist, ob eine Funktion selbstständig die definierten Schritte ausführt, oder ob diese weitere Funktionen aufrufen, die diese Schritte ausführen.

Weiter stellen diese Ergänzungen des Suchalgorithmus eine Steigerung des Aufwandes dar. Das neue Starten eines Suchvorgangs an jedem Knoten, wie in der zweiten Erweiterung beschrieben, führt zu einer exponentiellen Zunahme des Rechenaufwandes. Hinzu kommt die Kombination mit der ersten Erweiterung, wobei bei jeder Übereinstimmung ein zusätzlicher Suchvorgang zu dem bestehenden gestartet wird. Durch diese Steigerungen des Rechenaufwandes wird die Suche aufwändiger. Jedoch ist zu beachten, dass keine Änderungen des Modells oder Anti-Pattern vorgenommen werden. Hierdurch bietet sich eine Parallelisierung der Suchen an, wodurch auf Mehrkernsystemen die Dauer der Suche verkürzt werden kann da der Suchalgorithmus keinerlei Änderungen vornimmt und dadurch mehrere Instanzen gleichzeitig das Design Modell und die Anti-Pattern lesen können. Nur wenn eine Instanz Änderungen an den Objekten vornimmt kommt es zu Problemen wenn parallele Zugriffe auf ein Objekt geschehen. Der Sinn des Suchalgorithmus ist jedoch keinerlei Änderungen vorzunehmen und nur lesend auf die Objekte zuzugreifen. Deshalb stellt die Parallelisierung eine gute Möglichkeit dar die Suche trotz hohem Aufwand zu beschleunigen. Diese kann auch mit einer steigenden Anzahl von Anti-Pattern skalieren.

6 Prototypumsetzung des Konzepts

Die Implementierung des Konzepts ist beispielhaft für die Sequenzdiagramme umgesetzt worden. Diese wurden gewählt da jene zur Klasse der Interaktionsdiagramme gehören die dynamische Eigenschaften abbilden. Solche sind in einem statischen Code Review nur schwer zu identifizieren da in diesem meist strukturelle Eigenschaften entdeckt werden. Bei der Umsetzung wurde darauf geachtet, dass die Erweiterbarkeit für andere Diagrammtypen von UML eingehalten wird. Es wurde sich für die Sequenzdiagramme entschieden, da in diesen bereits bekannte Anti-Pattern gut modelliert und viele Muster, die eine Verwundbarkeit darstellen, erkannt werden können. Aufgrund dieser Entscheidung wurden in der vorangegangenen Arbeit viele Beispiele anhand von Sequenzdiagrammen erläutert.

Die Prototypumsetzung kann in drei verschiedene Aufgaben unterteilt werden. Zum Einen das Einlesen und Speichern von Sequenzdiagrammen bzw. Anti-Pattern. Zum Anderen die interne Datenhaltung mit der Modellierung der eingelesenen Datenstrukturen und als letztes die Suche von Anti-Pattern in den Sequenzdiagrammen.

Der allgemeine Ablauf des Programms zum Finden von Anti-Pattern in einem Sequenzdiagramm ist folgendermaßen. Der Prozess wird gestartet, dabei wird eine XMI-Datei übergeben die das Sequenzdiagramm enthält, das Verzeichnis in dem alle Anti-Pattern gespeichert sind (als XMI Datei) und ein Verzeichnis in dem die gefundenen Instanzen der Anti-Pattern abgelegt werden sollen. Danach wird das Programm die Dateien einlesen und die internen Datenstrukturen aufbauen. Sind diese fertig erstellt beginnt die Suche für jedes Anti-Pattern in dem Sequenzdiagramm. Die in der Suche gefundenen Instanzen der Anti-Pattern in dem Sequenzdiagramm werden dann im XMI-Format in dem Ergebnis-

verzeichnis abgelegt. Das Suchen und Speichern der Ergebnisse geschieht parallel für die verschiedenen Anti-Pattern.

6.1 Einlesen und Speichern von Anti-Pattern und Sequenzdiagrammen

Für die Verarbeitung von UML Diagrammen wurde eine eigene interne Struktur implementiert. Diese ist im Abschnitt 6.2 genauer beschrieben. Um auch Diagramme in die Prototypumsetzung importieren oder exportieren zu können wurde das Einlesen und Schreiben von XMI implementiert. Hierdurch bietet sich die Möglichkeit, die Sequenzdiagramme und Anti-Pattern auch in anderen Programmen zu öffnen die diesen Standard unterstützen. Beispielsweise das Programm UMLDesigner,[5] dieses unterstützt XMI, wodurch in diesem Diagramme und Anti-Pattern die von der Umsetzung erstellt wurden, dargestellt werden können.

Um Anti-Pattern in dem XMI Standard speichern zu können werden wie in Kapitel 5.3 beschriebene Erweiterungen benötigt. Folgende benötigten dabei keinerlei Anpassungen des Standards: Nicht, Enthält und Wildcard Operator. Da diese Operatoren durch den Namen eines Objektes abgebildet werden können sind keinerlei Anpassungen nötig. Dies wurde implementiert wie im Konzept beschrieben mit dem '*' für den Wildcard Operator, das Einrahmen durch '!' für den Nicht Operator und das Umklammern durch das '€' um den Enthält Operator zu kennzeichnen. Lediglich für den Wörterbuch Operator ist es notwendig Anpassungen vorzunehmen. Denn dieser kann zwar durch das Umschließen von '$' im Namen angedeutet werden, jedoch muss in einem anderen Abschnitt definiert sein welche Wörter das Wörterbuch enthalten soll.

Für das Definieren von Wörterbüchern wurde im XMI ein neuer Abschnitt angelegt, dieser befindet sich vor dem eigentlichen Diagramm. In diesem werden alle Wörterbücher mit deren Namen und auch den Wörtern die diese enthalten sollen definiert. Dies soll im Quellcode 2 gezeigt werden. In diesem ist von Zeile sechs bis neun die Erweiterung für Wörterbücher enthalten. Konkret wird in Zeile sieben das Wörterbuch 'encryptMessage',

[5] http://www.umldesigner.org (aufgerufen am 23.06.2020)

dass die Wörter ’encrypt’, ’verschlüsseln’ enthält, und in Zeile acht die Sammlung aus den Wörtern ’send’, ’senden’, ’schicken’ mit dem Namen ’sendMessage’ definiert. Das folgende Sequenzdiagramm in Zeile zehn bis elf ist leer. In diesem könnten die definierten Wörterbücher verwendet werden, indem der Name eines Elements, wie beispielsweise ’$encryptMessage$’ oder auch mit dem Enthält Operator kombiniert ’€$sendMessage$€’ verwendet wird. Auch eine Kombination mit dem Nicht Operator ist möglich.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <uml:Model xmi:version="20131001" xmlns:xmi="http://www.omg.org/spec/XMI/20131001" xmlns:uml="http://www.eclipse.org/
       uml2/5.0.0/UML" xmi:id="_000000001" name="Model">
3    <packageImport xmi:type="uml:PackageImport" xmi:id="_000000002">
4      <importedPackage xmi:type="uml:Model" href="pathmap://UML_LIBRARIES/UMLPrimitiveTypes.library.uml#_0"/>
5    </packageImport>
6    <packagedElement xmi:type="uml:extension" xmi:id="extensionDictionary" name="Dictionary">
7      <dictionary xmi:type="uml:extension:Dictionary" name="encryptMessage" value="encrypt verschlüsseln"/>
8      <dictionary xmi:type="uml:extension:Dictionary" name="sendMessage" value="send senden schicken"/>
9    </packagedElement>
10   <packagedElement xmi:type="uml:Interaction" xmi:id="_000000003" name="Sequenzediagramm" >
11   </packagedElement>
12 </uml:Model>
```

Quellcode 2: Beispiel für die Wörterbucherweiterung in XMI

Anhand von den genannten Beispielen wird ersichtlich, dass für die Erweiterung nur Anpassungen des XMI Standards bezüglich der Wörterbücher nötig sind. Der Vorteil dieser Implementierung unter dem Abschnitt ’packagedElement’ ist, dass Programme, die diese Erweiterung nicht kennen, wie UMLDesigner, dies einfach ignorieren, wodurch die Diagramme weiterhin in anderen Programmen geöffnet werden können.

Die Speicherung von Diagrammen in dem XMI Format ist hauptsächlich für die Speicherung von Suchergebnissen gedacht. Hierdurch ist es möglich gefundene Anti-Pattern in der Form von kleinen unvollständigen Diagrammen auszugeben. Dies bietet den Vorteil, dass ersichtlich wird wo das Anti-Pattern gefunden wurde und durch das allgemeine Format von XMI ist eine Wiederverwendbarkeit, sowie eine maschinelle Weiterverarbeitung möglich. Dadurch können die gefunden Abschnitte weiterverarbeitet oder analysiert werden.

Da es sich nur um eine Prototypumsetzung handelt und der Rahmen dieser Arbeit begrenzt ist, wurden nur das Einlesen und Speichern von einfachen Sequenzdiagrammen mit den Komponenten Lifeline, ExecutionSpecification und Message implementiert, da diese die wichtigsten Objekte eines Sequenzdiagramms darstellen. Intern ist die Verarbeitung mit weiteren Knoten und Pfaden möglich. Jedoch wird in den folgenden Abschnitten dargestellt,

dass die Unterscheidung hauptsächlich auf Knoten und Pfad beruht wodurch der genaue Typ eines Elements zweitrangig ist. Dies wird in Abschnitt 6.2 und 6.3 ersichtlicher. Weiter unterstützt auch das Programm UMLDesigner nur diese wichtigsten Komponenten.[6]

6.2 Datenhaltung und Modellierung von Sequenzdiagrammen

Bei der Prototypumsetzung des Konzepts wurde die Programmiersprache JAVA verwendet. Diese Entscheidung ist aus mehreren Gründen gefallen. Einer ist die Objektorientierung mit den Vererbungseigenschaften dieser Sprache. Dadurch ist es möglich die Elemente der Sequenzdiagramme aufzuteilen und die gemeinsamen Eigenschaften zu kapseln.

Ein weiterer wichtiger Grund für die Verwendung von JAVA ist die Leistungsfähigkeit und Plattformunabhängigkeit. Die Umsetzung ist dadurch auf verschiedenen Plattformen ausführbar und trotzdem bietet diese Sprache viele Möglichkeiten die Leistungsfähigkeit zu erhöhen, wie die einfache Parallelisierung von Ausführungen. Weiter bietet JAVA auch einfache Möglichkeiten Dateien zu verarbeiten was für die Umsetzung eines Prototyps sehr hilfreich ist. Das solche nativen Methoden einfach anwendbar und das Programm auf verschiedenen Plattformen ausführbar sind ist ein großer Vorteil dieser Sprache. Weshalb diese für die Umsetzung verwendet wurde.

6.2.1 Eigenschaften der UML Elemente und deren Vererbung

Die einzelnen Komponenten wurden durch verschiedene Klassen dargestellt. Durch die Vererbung wird es möglich, die Eigenschaften der Klassen zu verallgemeinern und so verschiedene Gruppen von Elementen zu bilden.

Die Grundlage hierbei stellt die Klasse 'UMLPatternExtension' dar. Diese speichert für ein Element alle Eigenschaften, die nicht durch ein Standard UML Objekt abgedeckt sind. Wie im Kapitel 5.3 erläutert, handelt es sich hierbei um die vier Erweiterungen: Wildcard, Wörterbuch, Enthält und Nicht Operator.

[6] http://www.umldesigner.org/ref-doc/implement-the-application.html (aufgerufen am 23.06.2020)

Die Basisklasse von jedem UML Element ist die Klasse 'UMLObject', diese stellt alle Eigenschaften dar die jedes Element von UML besitzt, wie den Namen oder die ID eines Elements. Weiter stellt die Klasse Funktionen zur Verfügung zum Vergleichen von zwei Objekten im Hinblick auf den Namen und die Art des Objekts.

Zu den bereits erläuterten abstrakten Klassen 'UMLPatternExtension' und 'UMLObject' gibt es die Basisklassen für Knoten und Pfade von Sequenzdiagrammen 'SequenceDiagrammNode' und 'SequenceDiagrammPath'. Diese sind auch abstrakt, da eine Instanz von solchen wenig Sinn ergibt und nur die erweiterten Klassen, die einzelne Elemente eines Sequenzdiagramms darstellen, erzeugt werden sollen.

In der Abbildung 13 ist die Vererbung der abstrakten Basisklassen eines Sequenzdiagramms zu sehen. Weiter wird dargestellt, dass die Operatoren in eigenen Klassen ausgelagert wurden und die Klasse 'UMLPatternExtension' diese Eigenschaft speichert. Durch diese Aufteilung ergibt sich der allgemeine Zugriff auf Knotenelemente über die Klasse 'SequenceDiagrammNode' und auf Pfad Elemente über die Klasse 'SequenceDiagrammPath'. Der Zugriff allgemein auf ein Element eines UML Diagramms kann über die Klasse 'UMLObject' abgebildet werden.

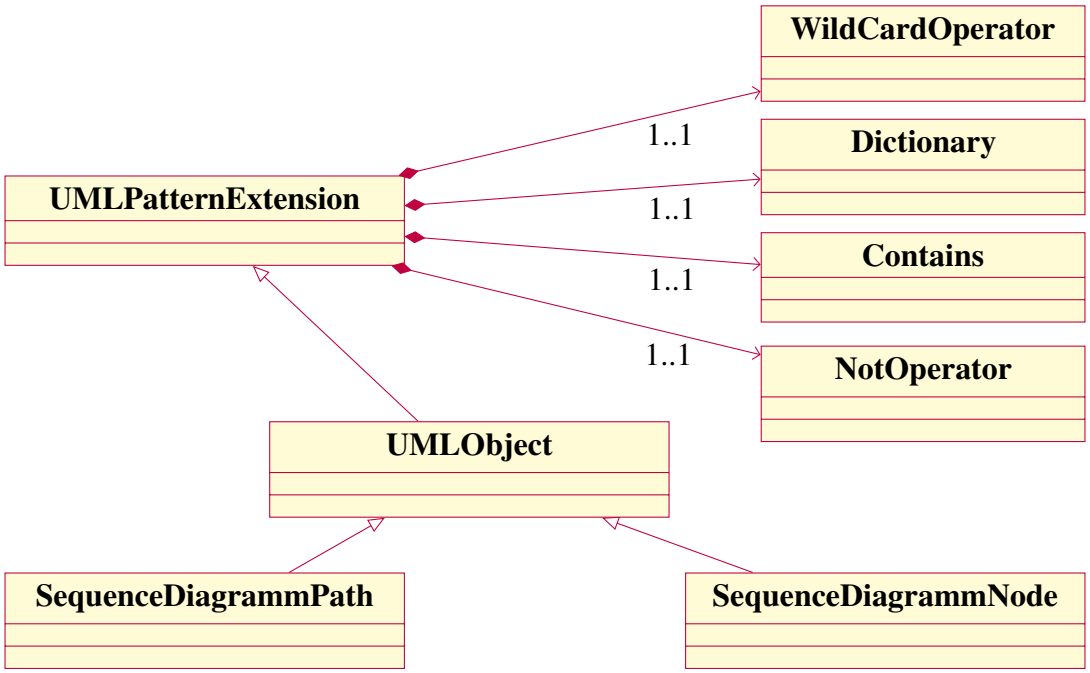

Abbildung 13: Vererbung der abstrakten Basisklassen eines Elements

Die abstrakte Klasse 'SequenceDiagrammNode' stellt die Grundlage für alle Knoten eines Sequenzdiagramms dar. Diese sind zum Beispiel das Sequenzdiagramm selbst, die Lebenslinien und auch die einzelnen Ausführungen. Auf weitere Knoten wie ein Frame oder CombinedFragment soll nicht weiter eingegangen werden, aber auch für diese bildet die abstrakte Klasse 'SequenceDiagrammNode' die Grundlage. Es gibt noch weitere Knoten, die in der Spezifikation von UML erläutert werden.[22] Diese wurden größtenteils umgesetzt, in dieser Arbeit finden diese keine weitere Erwähnung, da jene genau wie alle anderen Knoten behandelt werden. In Abbildung 14 sind ein Teil der Klassen für Knoten zu sehen, sowie das diese von der übergeordneten Klasse erben.

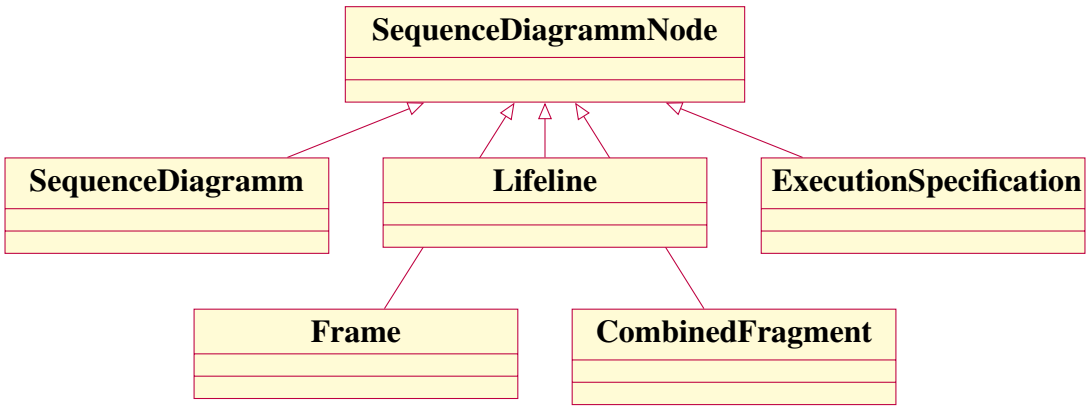

Abbildung 14: Vererbung der allgemeinen Eigenschaften auf die Knotenklassen

Von der Klasse 'SequenceDiagrammPath' werden alle Nachrichten und Pfade des Sequenzdiagramms abgeleitet. Hierbei ist keine weitere Unterscheidung nötig, da eine Nachricht einen Pfad, also eine Verbindung zwischen zwei Knoten, darstellt. In Sequenzdiagrammen sind die wichtigsten Nachrichten: Message, LostMessage und FoundMessage. Die letzten zwei stellen eine Spezialform der Message dar, da diese keinen Sender oder Empfänger haben. Es werden in der Spezifikation noch weitere Pfade erwähnt welche auch im Prototyp umgesetzt sind, jedoch sollen diese weiter keine Erwähnung finden, da diese genau gleich wie alle Pfade behandelt werden.[22] Als Beispiel für diese wird der am meisten verbreiteten Pfad Message verwendet. In der Abbildung 15 ist die abstrakte Klasse 'SequenceDiagrammPath' und die von erbenden Klassen 'Message', 'LostMessage' und 'FoundMessage' zu sehen die von dieser abgeleitet werden.

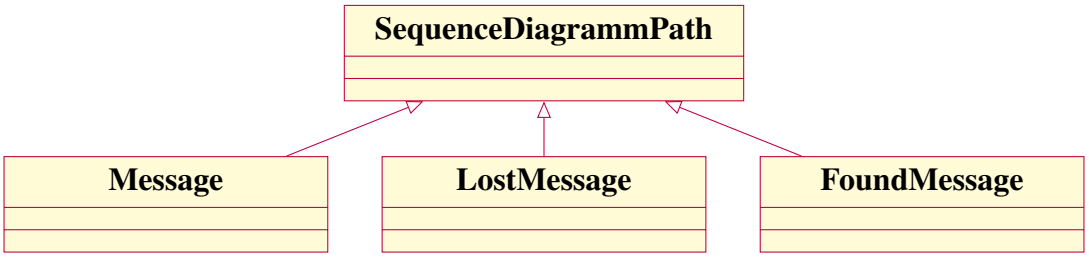

Abbildung 15: Vererbung der allgemeinen Eigenschaften auf die Pfadklassen

6.2.2 Aus einem Sequenzdiagramm abgeleitete Baumstruktur

Aufgrund der Aufteilung zwischen Knoten und Pfaden bietet sich eine Baumstruktur zur Verarbeitung der Daten an. In dieser kann jeder Knoten beliebig viele Kindknoten besitzen und zusätzlich auch beliebig viele Pfade zu anderen Knoten. Dabei können die Pfade auch als Kindelemente des Knotens angesehen werden. Wichtig dabei ist die Einhaltung der Reihenfolge der Kindelemente, da die gerade im Sequenzdiagramm von essentieller Bedeutung ist. Da bei diesem Diagramm die Reihenfolge und das Ablaufen von Aktionen dargestellt werden soll und es somit essentiell ist, die Reihenfolge in einer Datenstruktur zu erhalten.

Daraus lässt sich ableiten, dass in dem allgemeinen Aufbau der Baumstruktur nur zwischen Knoten (’SequenceDiagrammNode’) und Pfaden (’SequenceDiagrammPath’) unterschieden wird, nicht aber auf die genaue Art von Knoten- oder Pfadelement. Deshalb wurde auch im Abschnitt 6.2.1 nicht genauer auf die einzelnen Arten dieser Gruppen eingegangen. Weiter wird noch eine weitere Verallgemeinerung getroffen, dass Knoten nicht beliebig viele Kindknoten haben, sondern das Knoten beliebig viele Kindelemente der Klasse ’UMLObject’ haben. Hierdurch wird es möglich die Kindelemente in der richtigen Reihenfolge in einem Knoten zu speichern, unabhängig von deren Art. Weiter kann angenommen werden, dass jedes Element mindestens ein Elternelement hat, da jeder Knoten eine und jeder Pfad zwei Elterninstanzen haben muss. Die einzige Ausnahme ist der Wurzelknoten (’SequenceDiagramm’), dieser stellt das Wurzelelement dar und hat deshalb als einziger kein Elternelement. Auch die Lost- und FoundMessage sind eine Ausnahme, da diese die einzigen Pfade sind die nur einen Elternknoten haben und nicht wie die anderen Pfade zwei Elterninstanzen.

Die Baumstruktur wurde verwendet, da sich in dieser die Informationen über Reihenfolge und Kindelemente gut darstellen lässt, weil durch den Aufbau des Baums diese Informationen bereits vorhanden sind. Da die Reihenfolge der Elemente eines Baums gleichbleibend ist und auch unterschieden werden kann zwischen Kindelementen und den Kindelementen der Kindelemente. Dadurch ergibt sich die implizite Speicherung der Informationen.

Ein Baum, der sich aus diesem Umstand ableiten lässt, soll in Abbildung 16 dargestellt werden. In dieser ist links der Baum eines Sequenzdiagramms zu sehen. Erkennbar ist, dass die Reihenfolge implizit durch die Reihenfolge der Ankerpunkte enthalten ist und auch das eine Ausführung aber auch eine Nachricht ein Kindobjekt eines Knotens sein kann. Auf der rechten Seite ist derselbe Baum, ohne den konkreten Typ der einzelnen Elemente zu sehen, sondern nur die Unterscheidung von Knoten und Pfad getroffen worden. Dabei ist zu erkennen, dass in der Baumstruktur der genaue Typ des Elements irrelevant ist und nur die Unterscheidung zwischen Knoten und Pfad getroffen werden muss. Der genaue Typ wird erst in der Suche wichtig, da bei dem Vergleich von zwei Elementen auch der Typ der Elemente berücksichtigt werden muss. Denn wenn nach einer Ausführung gesucht werden soll reicht nicht die Abstraktion auf das allgemeine Element Knoten, denn ein Knoten kann auch eine Lifeline oder ein anderes Element darstellen. Dieser Umstand soll in Abschnitt 6.3 genauer erläutert werden. Daraus ist zu erkennen, dass die Abstraktion in der Implementierung diese Verallgemeinerung in der Struktur zu treffen, die Wiederverwendbarkeit der implementierten Algorithmen erhöht. Dadurch können viele Funktionen auch für andere Diagrammarten wiederverwendet werden und das System ist erweiterbar für weitere Arten von UML Diagrammen.

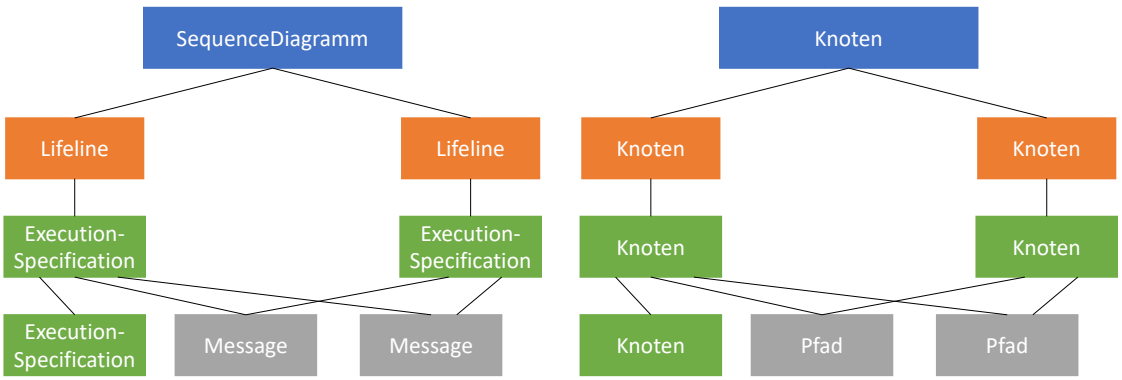

Abbildung 16: Allgemeiner Aufbau der Baumstruktur

In der Abbildung 16 ist ein sehr einfaches Sequenzdiagramm abgebildet. Für größere und komplexere Diagramme wird die Baumstruktur auch komplexer und verzweigter. Daher soll in dem Bild nur der grundsätzliche Aufbau gezeigt werden, denn für größere und verzweigte Diagramme ist die Baumstruktur nur schwer in einer Abbildung übersichtlich darzustellen. Es gelten jedoch auch für diese die selben Voraussetzungen und in der maschinellen Verarbeitung gibt es auch in einem größeren Diagramm keine Unterschiede zu diesem kleinen Beispiel.

6.3 Anti-Pattern Suche in Sequenzdiagrammen

Der implementierte Suchalgorithmus orientiert sich an der in Kapitel 5.4 vorgestellten Grundsuche ohne die erwähnten Erweiterungen. Da diese Prototypumsetzung eine einfache Umsetzung des Konzeptes darstellen soll, sowie im Konzept noch weitere Optimierungsmöglichkeiten und auch erweiterte Aspekte zum Finden von verschachtelten Pattern vorgestellt werden.

Der implementierte Suchalgorithmus startet mit einem Vergleich von dem Wurzelobjekt des Anti-Pattern mit jedem Knoten des zu durchsuchenden Diagramms. Dies ist nötig, da ein Anti-Pattern nicht immer ein vollständiges Diagramm darstellen muss. Auch wenn es XMI nicht unterstützt, da in diesem Format ein vollständiges Diagramm definiert werden muss. Es ist denkbar, dass Anti-Pattern nur aus einem Teil eines Diagramms bestehen. Zum Beispiel kann ein Anti-Pattern für ein Sequenzdiagramm nur aus einer Ausführung und einer Nachricht, die von dieser Ausführung gesendet wird, bestehen. Dies stellt kein vollständiges UML Sequenzdiagramm dar, denn in diesem muss jede Ausführung einer Lebenslinie und diese einem Diagramm untergeordnet sein. Da die implementierte Suche nicht auf diese feste Struktur festgelegt sein soll und auch die vorgestellte Datenstruktur die Möglichkeit bietet, dass jeder Knoten der Wurzelknoten eines Patterns sein kann, wurde diese Suche so implementiert. Falls bei dem Abgleich des Wurzelobjektes des Anti-Pattern mit einem Knoten des Diagramms eine Übereinstimmung gefunden wurde wird für diesen Knoten der weitere Suchalgorithmus ausgeführt. Somit wird der Algorithmus zur Suche mehrfach und für verschiedene Knoten ausgeführt, wodurch auch mehrere unterschiedliche Instanzen des Pattern in einem Diagramm gefunden werden können.

Bei einer Übereinstimmung des Wurzelknotens versucht der implementierte Suchalgorithmus einen übereinstimmenden Knoten für jeden Kindknoten des Anti-Patterns in den Kindknoten des Diagramms zu finden. Die Übereinstimmung von zwei Knoten kann definiert werden durch *das Übereinstimmen des Typs des Knotens und das Übereinstimmen der Namen bzw. der Übereinstimmung der Namen unter Einbeziehung der Erweiterungen wie Enthält, Wörterbuch oder Wildcard Operator und des Vorkommens eines übereinstimmenden Kindknotens in dem Diagramm für jeden Kindknoten des Anti-Pattern in der richtigen Reihenfolge. Falls es sich um einen Pfad und nicht um einen Knoten handelt, bedeutet die Übereinstimmung, das Übereinstimmen des Typs und des Namens unter Einbeziehung möglicher Erweiterungen und das Übereinstimmen von Anfang- und Endknoten des Pfades.* Für die Suche wurde in der Implementierung eine Optimierung umgesetzt, wenn eine Lebenslinie den Namen '**' hat, ist das die verstärkte Form des Wildcard Operators, da für diese keine Instanz gesucht werden muss, sondern alle Lebenslinien zutreffend sind. Das ist der Fall, wenn für Anti-Pattern eine beliebige Lebenslinie zum Entgegennehmen von Nachrichten benötigt wird, dieser Umstand wird in den Beispielen im Abschnitt 6.4 ersichtlicher.

Bei der Übereinstimmung der Knoten von Anti-Pattern und Diagramm muss berücksichtigt werden, dass eine Übereinstimmung eines Knotens des Anti-Pattern, bei dem der Nicht Operator gesetzt ist, mit einem Knoten des Diagramms zu einem Abbrechen des Suchalgorithmus führt. Hierbei wird die Aussage 'keine Übereinstimmung' getroffen. Denn nach einer Übereinstimmung eines Knotens und einem Patternknoten mit der Nicht Eigenschaft kann das Muster in dem nachfolgenden Diagramm nicht mehr gefunden werden.

Der implementierte Suchalgorithmus optimiert die Suche durch das parallele Suchen verschiedener Anti-Pattern. Auch wenn in dem im Kapitel 6.4 vorgestellten Beispiel es sich um ein relativ kleines Diagramm handelt für das die Suche schnell abgeschlossen ist, soll diese Optimierung zeigen, dass das vorgestellte Konzept auch für größere Diagramme und vor allem die Suche nach vielen Anti-Pattern skalieren und dadurch auch für große Diagramme schnell sein kann. Dies ist ein sinnvoller Optimierungsschritt, da viele moderne Computer nicht nur einen Prozessorkern zur Verfügung haben, sondern mehrere. Damit kann diese Parallelisierung einen deutlichen Vorteil bieten, außerdem wird weiter die Möglichkeit aufgezeigt die Parallelität noch zu erweitern wie in Abschnitt 5.5 erwähnt

bieten sich hierfür mehrere Möglichkeiten. Durch die parallele Suche der Anti-Pattern sollte in dieser Prototypimplementierung die Umsetzbarkeit von diesen Optimierungen gezeigt werden.

Der Suchalgorithmus endet nicht nur mit dem Ergebnis ob ein Anti-Pattern gefunden wurde oder wie oft dieses entdeckt wurde, sondern liefert zusätzlich einen Diagrammausschnitt mit den Übereinstimmenden Objekten des Diagramms und des Anti-Patterns. Hierfür soll in Abschnitt 6.4 Beispiele gezeigt werden.

6.4 Ausführung der Umsetzung anhand eines Beispieldiagramms

Für die Ausführung der Prototypumsetzung wird ein Sequenzdiagramm und verschiedene Anti-Pattern benötigt. Diese wurden mit Hilfe von JUnit in einem Test in der Prototypimplementierung erzeugt und danach mithilfe der Exportfunktion der Umsetzung im XMI Format gespeichert. Dadurch kann die Portabilität der exportierten Diagramme getestet werden. Das für die Suche verwendete Sequenzdiagramm ist in Abbildung 17 zu sehen.

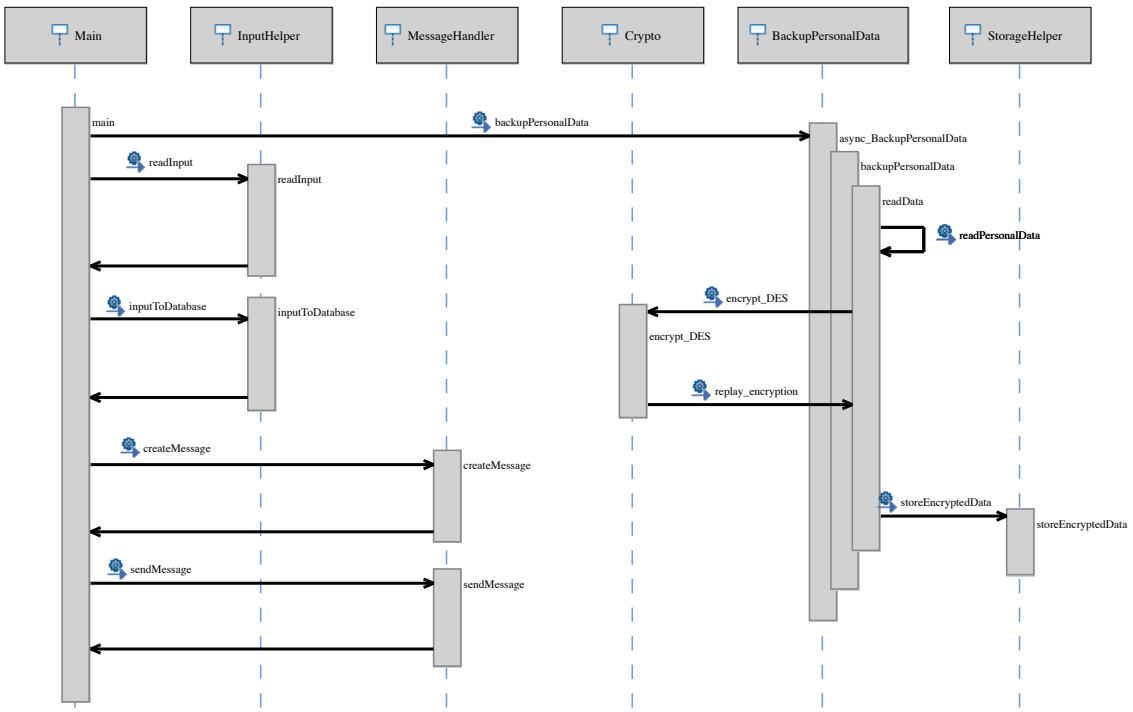

Abbildung 17: Sequenzdiagramm in dem nach Anti-Pattern gesucht werden soll

Diese Abbildung wurde mithilfe von UMLDesigner nach dem Import des Diagramms erstellt. Hierbei ist die Kompatibilität die durch das verwendete XMI Format entsteht ersichtlich. Bei dem Import mussten lediglich optische Verschönerungen vorgenommen werden, um ein übersichtlicheres Diagramm zu erhalten.

In dem Sequenzdiagramm sind sechs verschiedene Lebenslinien zu sehen. Die Ausführungen werden von der Hauptlebenslinie Main koordiniert. Hierbei startet Main anfangs eine asynchrone Nachricht wodurch das Sichern von persönlichen Daten angeregt wird. Danach werden Eingaben eingelesen und verarbeitet die daraufhin in eine Datenbank geschrieben werden sollen. Bei dem Speichern der persönlichen Daten ist der erste Schritt diese einzulesen, danach zu verschlüsseln und auf einem Speichermedium zu persistieren. Dabei ist zu sehen, dass es sich um ein übersichtliches und leicht verständliches Sequenzdiagramm handelt.

Damit potentielle Schwachstellen in Form von Anti-Pattern gefunden werden können, müssen diese zuerst definiert werden. In diesem Beispiel wurden vier verschiedene Anti-Pattern modelliert die in dem Diagramm enthalten sind, aber deren Definition allgemein gehalten wurde, wodurch diese Anti-Pattern auch in anderen Sequenzdiagrammen gefunden werden können. Die Anti-Pattern sollen nun im weiteren Verlauf einzeln vorgestellt werden.

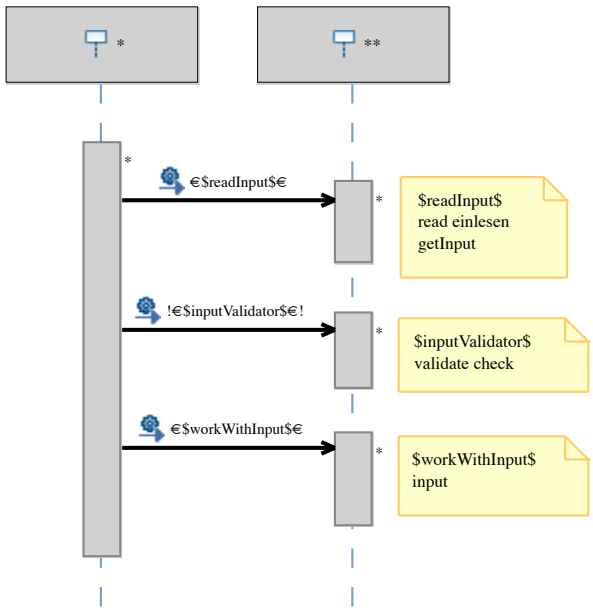

Abbildung 18: Anti-Pattern zur fehlenden Eingabeüberprüfung

In der Abbildung 18 ist ein Anti-Pattern zu sehen, dass Stellen in einem Sequenzdiagramm identifiziert in dem Eingaben verarbeitet werden ohne einer vorgeschalteten Validierung. Dies kann zu einer Sicherheitsschwachstelle führen wenn bei der weiteren Verarbeitung von dem Input dieser in Datenbanken geschrieben oder in andere Subsysteme geteilt wird, da hier durch Injektion Angriffe möglich werden.[8] In diesem Beispiel ist auch zu sehen wie die Erweiterung des Wildcard Operators sinnvoll sein kann. Denn die zweite Lebenslinie mit dem Namen '**' hat keine Ausführung, die nicht mit dem Wildcard Operator versehen ist und nimmt auch nur Nachrichten entgegen. Daher muss in einem Diagramm für diese Lifeline kein übereinstimmendes Element gefunden werden, da diese auch durch mehrere verschiedenen Lebenslinien abgebildet sein können und dies impliziert durch die Prüfung der Nachrichten (Pfade) gefunden wird. Weiter wurden bei der Definition verschiedene Erweiterungen verwendet. So wurde die Nicht Erweiterung benutzt um auf das Nichtvorhandensein von einer Validierungsfunktion zu prüfen. Auch wurde eine Kombination aus dem Enthält und Wörterbuch Operator verwendet um auf die einzelnen Aufrufe der Funktionalitäten zum Einlesen, Validieren und Weiterverwenden der Eingaben zu erkennen. In der Abbildung wurden die Wörter die in dem Wörterbuch enthalten sind als Kommentar seitlich hinzugefügt.

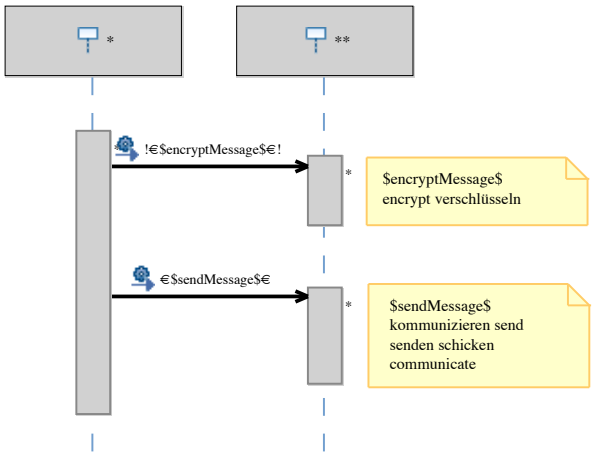

Abbildung 19: Anti-Pattern zur fehlenden Verschlüsselung von Nachrichten

In der Abbildung 19 ist ein Anti-Pattern abgebildet, welches die unverschlüsselte Kommunikation finden soll. Hierfür wird überprüft, ob eine Funktion einen Teilnamen aus einem Wörterbuch enthält in dem verschiedene Namen für eine sendende oder kommu-

nizierende Funktion spezifiziert sind. Es wird auch festgelegt das zuvor keine Funktion aufgerufen werden darf, die das Verschlüsseln einer Nachricht übernehmen könnte. Auch in diesem Beispiel sind alle Erweiterungen für die Definition von Anti-Pattern enthalten und auch nötig um das Anti-Pattern wiederverwendbar und allgemein definieren zu können. Dieses Muster stellt eine potentielle Verwundbarkeit dar, denn Kommunikation kann abgehört werden und somit kann ein ungewollter Informationsabfluss entstehen. Daher sollte Kommunikation, wenn möglich, in verschlüsselter Form erfolgen.

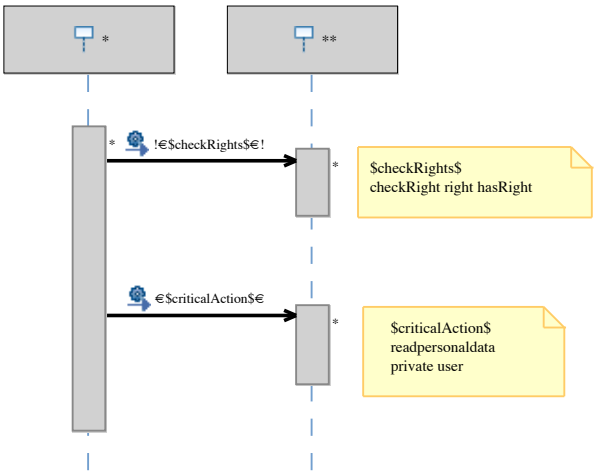

Abbildung 20: Anti-Pattern zur fehlenden Rechteüberprüfung

Ein weiteres Anti-Pattern ist zum Finden von fehlenden Überprüfungen der Rechte bevor kritische Aktionen ausgeführt werden. Dieses Muster ist in Abbildung 20 zu sehen. In jenem wird zuerst die fehlende Überprüfung der Rechte definiert auf die ein Aufruf einer kritischen Funktion folgen muss. Was eine kritische Funktionalität ist, ist stark von der Anwendung und deren Kontext abhängig. Meist handelt es sich hierbei um persönliche Daten oder streng vertrauliche bzw. geheime Informationen in einem Betrieb. Da es je nach Anwendung hierbei Unterschiede gibt, kann das Muster allgemein formuliert werden wie in diesem Fall. Jedoch sind die verwendeten Wörterbücher sehr unvollständig und stark von den Implementierungsrichtlinien einer Anwendung abhängig. Dieser Umstand gilt für alle in dieser Arbeit vorgestellten Anti-Pattern. Die Wörterbücher müssen je nach verwendetem Umfeld und auch auf die Sprache angepasst werden und sollen in diesen Beispielen nur zum Demonstrieren der Funktionalität verwendet werden.

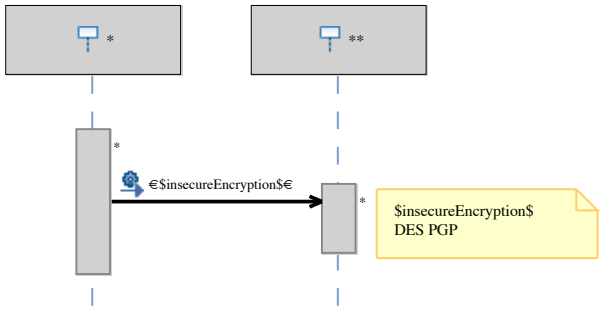

Abbildung 21: Anti-Pattern zur Verwendung unsicherer Verschlüsselungen

In der Abbildung 21 wird ein Muster aufgezeigt bei dem unsichere Verschlüsselungsfunktionen verwendet werden. Die Empfehlung des BSI zur Verwendung von symmetrischen Verschlüsselungsverfahren ist die Benutzung von AES mit einer Schlüssellänge von mindestens 128 Bit. Weiter wird beschrieben, dass in neuen Applikationen auf die ältere Verschlüsselung DES verzichtet werden sollte. Dies geht aus einer Technischen Richtlinie hervor die vom BSI im März 2020 veröffentlicht wurde.[7] Bei der Erstellung von Applikationen sollten die Technischen Richtlinien des BSI unbedingt beachtet werden, da diese wichtige Hinweise auf die Sicherheit geben und auch Abschätzungen in die Zukunft zur Sicherheit von Algorithmen enthalten. Mit diesem Anti-Pattern wird nach der Verwendung von Verschlüsselungen gesucht die als unsicher gelten oder für einen bestimmten Verwendungszweck im implementierten Umfeld unsicher sind. Hierzu müssen die Funktionen mit unsicheren Verschlüsselungsmethoden nur in das Wörterbuch aufgenommen werden und die Suche findet diese. Das Muster soll ein Beispiel für Anti-Pattern sein die nicht nur Entwurfsentscheidungen finden können die zu einer potentiellen Schwachstelle führen, sondern auch Implementierungsentscheidungen. Denn die Entscheidung für eine unsichere Verschlüsselungsmethode kann sowohl im Design als auch in der Implementierung getroffen werden. Somit können durch diese Art von Muster potentielle Schwachstellen im Software Design gefunden werden die im Design entstanden sind, aber auch für solche die in der Implementierung getroffen wurden. Durch das mögliche Reverse Engineering vom Quelltext zu den Software Design Diagrammen kann auch eine Anti-Pattern Suche für bereits implementierte Software durchgeführt werden.

Für die Ausführung der Prototypumsetzung sind die Anti-Pattern in dem Verzeichnis 'AntiPattern' gespeichert und die Ergebnisse sollen in dem Ordner 'Output' gesichert

werden. Beim Starten des Programms wird der Pfad zu dem Diagramm, das Verzeichnis für die Anti-Pattern und der Pfad zu dem Ordner für die Ergebnisse übergeben. Nach dem Starten liest das Programm zuerst das Diagramm welches im XMI Format gespeichert ist ein. Danach werden alle Anti-Pattern die sich in dem Ordner befinden und im XMI Format abgespeichert sind eingelesen. Für jedes eingelesene Anti-Pattern wird ein neuer Thread gestartet der nach Vorkommnissen im Diagramm sucht. Die gefundenen Ergebnisse werden im XMI Format im Zielverzeichnis abgelegt. Hierbei erhalten die gefundenen Vorkommnisse den Dateinamen im folgenden Format: zuerst kommt der Name der Datei des Anti-Pattern, danach die Zahl der Vorkommnis und die Dateiendung '.uml'. Zum Beispiel wird das Anti-Pattern 'mostPattern.uml' zum dritten Mal in dem Diagramm gefunden, erhält es den Dateiname 'mostPattern.uml3.uml'. Somit ist das Anti-Pattern direkt ersichtlich, sowie die gefundenen Instanzen eines Anti-Patterns durch deren Nummern direkt unterschieden werden können.

Das Programm fand in dem Diagramm (Abbildung 17) für jedes der vorgestellten Anti-Pattern (Abbildungen 18, 19, 20, 21) jeweils eine Instanz. Diese wurden als Teil eines Sequenzdiagramm im XMI Format gespeichert. Es ist nicht möglich diese Teile von Sequenzdiagrammen in UMLDesigner zu importieren, daher wurde die Abbildung 22, der gefunden Instanzen, manuell erstellt. UMLDesigner kann die im XMI Format gespeicherten Instanzen nicht öffnen da diese keine vollständigen UML Modelle sind, sondern nur Ausschnitte. Das bedeutet, diese enthalten zum Beispiel Nachrichten, die von keiner Ausführung empfangen werden, da lediglich das Vorkommen der Nachricht dargestellt wird und kein vollständiges Diagramm spezifiziert werden soll. Das ist ein gewolltes Verhalten, damit die Ausschnitte mit den gefundenen Instanzen übersichtlich bleiben. Denn ein vollständiges Diagramm kann einen zu großen Ausschnitt aus dem Gesamtdiagramm darstellen, in dem es nicht immer eindeutig ersichtlich sein muss, an welcher Stelle die Suche das Anti-Pattern identifiziert hat. Weil in dieser Umsetzung eindeutig ersichtlich sein soll an welcher Stelle das potentiell unsichere Entwurfsmuster gefunden wurde, ist die Ausgabe von unvollständigen Diagrammen sinnvoll.

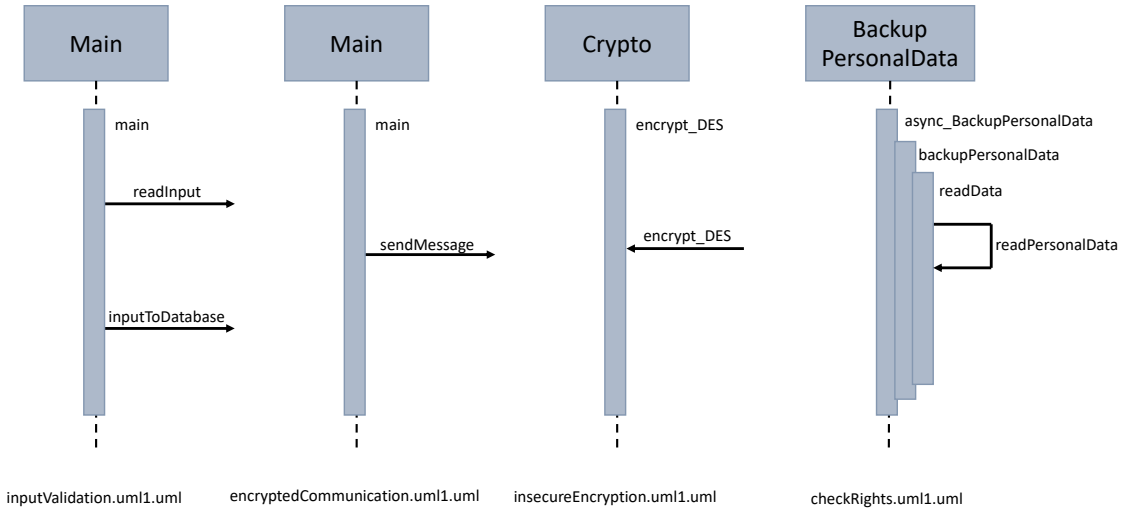

Abbildung 22: Gefundene Instanzen der Anti-Pattern

Hierbei ist zu sehen, dass die Prototypimplementierung das XMI Modell sowohl vom Diagramm als auch von den Anti-Pattern korrekt einließt und in der darauffolgenden Suche die Instanzen richtig identifiziert hat. Weiter ist aufgrund des gespeicherten Formats ersichtlich wo die Instanzen im Modell gefunden wurden und auch das Vorkommen von Mustern, die eine Ausführung einer Ausführung darstellen, wurden identifiziert.

Durch den Namen der Lebenslinie, Ausführungen und Nachricht ist auch schnell ersichtlich wo die potentielle Verwundbarkeit gefunden wurde. Somit kann an der betroffenen Stelle eine Verbesserung vorgenommen werden. Außerdem ist in der Abbildung ersichtlich, warum es sich um Ausschnitte aus Sequenzdiagrammen handelt, da bei den Nachrichten nur der Sender oder Empfänger dargestellt wird.

In der Abbildung 23 ist das Sequenzdiagramm zu sehen und die gefundenen Anti-Pattern wurden darin farblich hervorgehoben. Zusätzlich wurde neben den Instanzen der Muster der Name der Datei geschrieben in dem das Pattern definiert ist. Das ist, wie in Abbildung 22 zu sehen ist, der selbe Name der unterhalb der gefundenen Instanzen zu sehen ist. Anhand dieser Gegenüberstellung der ausgegebenen Dateien und der Markierungen im Modell soll gezeigt werden, dass durch die unvollständigen Teildiagramme die betroffenen Stellen im Software Design gut identifiziert werden können.

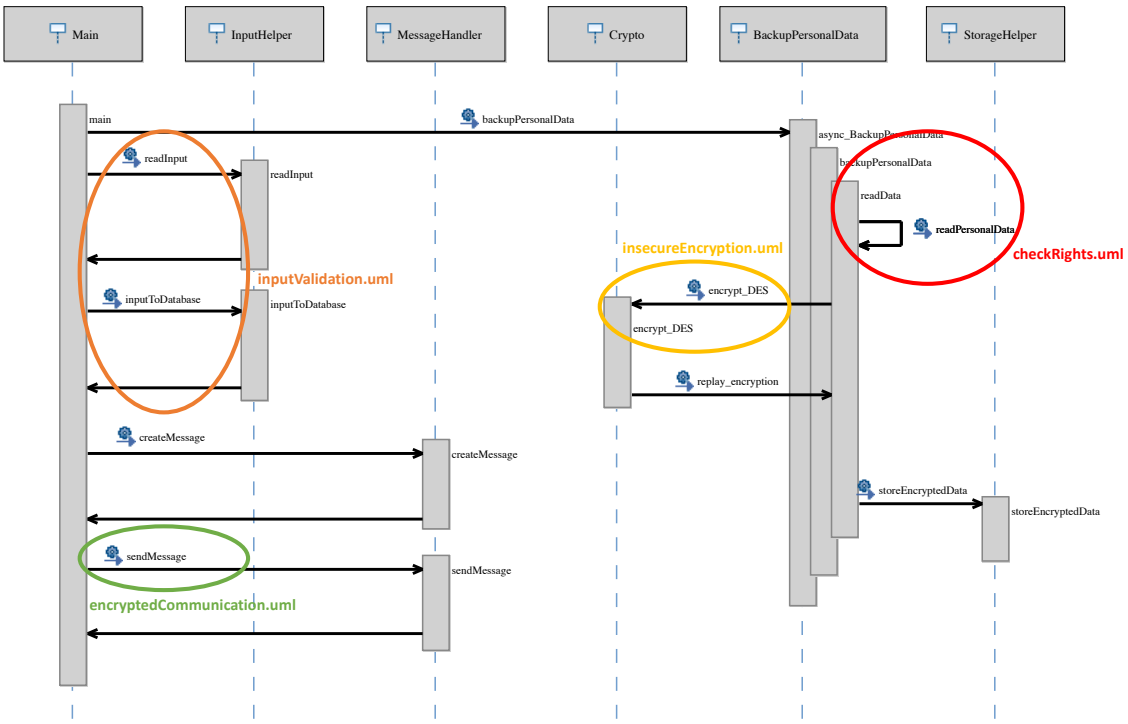

Abbildung 23: Gefundene Instanzen der Anti-Pattern im Model

Dies ist möglich, da durch die Angabe der Lifeline und der entsprechenden Ausführung und durch das Beschränken der nötigen Nachrichten die Stelle im Gesamtdiagramm einfach identifiziert werden kann. Durch das genannte Vorgehen ist es möglich, dass ein System automatisiert die Stelle im Gesamtdiagramm entdeckt. Das kann auch durch die Möglichkeit geschaffen werden, indem jedes Element des Ausschnittes auf die ID des Objekts im Gesamtdiagramm verweist. Hierdurch kann für Programme eine einfachere Möglichkeit geschaffen werden um die Objekte zu verknüpfen und eine Visualisierung der gefunden Instanzen der Anti-Pattern im Gesamtdiagramm darzustellen. Dies ist kein Teil der Arbeit und wurde auch nicht umgesetzt. Nur die Möglichkeit für ein solches Vorgehen wird an dieser Stelle aufgezeigt.

Weiter wird daraus ersichtlich, dass die zur Umsetzung verwendeten Methoden eine Erweiterung des Systems um nützliche Komponenten unterstützen. Es wird somit deutlich, dass ein für Erweiterungen offenes System entstanden ist.

7 Evaluierung von Security Design Anti-Pattern und deren Suche

Die Implementierung und der daraus folgende Prototyp beweisen, dass eine Umsetzung von Security Design Anti-Pattern und deren Suche im Softwaredesign möglich ist. Auch werden potentielle Schwachstellen durch die Suche von definierten Anti-Pattern im Software Design entdeckt. Es ist somit möglich, durch die Definition von wiederverwendbaren unsicheren Entwurfsmustern, die Sicherheit von Programmen schon in der Entwurfsphase zu stärken.

Alle Anforderungen an Anti-Pattern und deren Suche in Abschnitt 4.2 wurden mit dem vorgestellten Konzept eingehalten. Die Anti-Pattern sind maschinell verarbeitbar da diese durch die Definition im XMI Format in einem standardisierten und für die automatische Verarbeitung angelegten Schema gespeichert werden. Mit der Ergänzung des XMI Formats ergeben sich mehr Möglichkeiten Anti-Pattern zu definieren. Weiter wurden auch die allgemeinen Anforderungen: Wiederverwendbarkeit, Kombinierbarkeit, Unterscheidbarkeit eingehalten. Auch die nur im begrenzten Maß anwendbaren Anforderungen: Vollständigkeit, Überprüfbarkeit und Freiheit wurden in diesem gesetzten Kontext umgesetzt. Das vorgestellte Konzept bietet aufgrund der vorgestellten Erweiterungen die Möglichkeit wiederverwendbare Anti-Pattern zu definieren da eine allgemeine Beschreibung formuliert werden kann ohne auf konkrete Namen von Elementen einzugehen. Hierbei ist zu erkennen, dass die Vollständigkeit nur in dem geforderten Kontext umgesetzt wurde und die Anti-Pattern im Kontext der Erweiterungen vollständig definiert werden können. Mit der Erweiterung von Enthält und Wörterbuch Operator ist es möglich, Anti-Pattern zu formulieren die auf verschiedene Diagramme angewendet oder durch den Austausch von

Wörterbüchern speziell auf ein neues Diagramm spezifiziert werden können. Anhand des Austausches eines Wörterbuchs ist ein Pattern wiederverwendbar, da die Ablaufeigenschaften und weitere Informationen erhalten bleiben. Auch eine Kombinierbarkeit der Anti-Pattern ist gegeben, denn das Vorkommen eines Anti-Patterns schließt das Vorhandensein eines anderen Musters nicht aus. Außerdem zeigt der Prototyp, dass durch der unabhängigen Suche von Anti-Pattern diese auch an derselbe Stelle gefunden werden können. Die Kombination mehrerer Anti-Pattern zu einem großen Anti-Pattern ist theoretisch möglich, wenn auch nicht weiter ausgeführt. Das die Anti-Pattern überprüfbar sind, ist aus der Tatsache ableitbar, dass bei der Suche nach diesen in einem Diagramm die Entscheidung getroffen werden kann ob jene enthalten sind. Es ist im Konzept genau festgehalten wann eine Übereinstimmung zwischen einem Anti-Pattern und einem Diagramm vorliegt wodurch die Muster gefunden werden können. Anhand dieser Definition ist eine Entscheidung über das Vorkommen möglich, wodurch die Überprüfbarkeit in dem geforderten Maß gegeben ist. Weiter können die Anti-Pattern und die gefundenen Instanzen unterschieden werden, wodurch auch die geforderte Unterscheidbarkeit gegeben ist. Auch die Freiheit ist umgesetzt worden, denn die Anti-Pattern sind nur begrenzt von einer Programmiersprache abhängig.

Das vorgestellte Konzept entspricht der Anforderung erweiterbare Sammlungen von Anti-Pattern erstellen zu können. Durch die Wiederverwendbarkeit der Anti-Pattern ist es möglich Sammlungen anzufertigen die auf verschiedene Projekte angewendet werden können. Der Prototyp zeigt auch, dass solch eine Sammlung erweiterbar ist, da in diesem ein Ordner angegeben wird in dem alle Anti-Pattern gespeichert sind, die in dem Diagramm gesucht werden sollen. Durch den Austausch von Anti-Pattern in dem Ordner ist es möglich diese Sammlung zu erweitern oder auch Anti-Pattern zu entfernen. Damit ist die Erweiterbarkeit für Sammlungen mit diesem Konzept gegeben.

Die Flexibilität Anti-Pattern auf verschiedene Arten von UML Diagrammen anzuwenden ist in dem Konzept vorgesehen, wurde jedoch nicht weiter ausgeführt, da dieses Konzept beispielhaft für Sequenzdiagramme vorgestellt wurde. Es ist jedoch möglich, dass Konzept auch auf andere Diagrammarten anzuwenden. Hierfür kann eine Anpassung der Suche nötig sein, da bei anderen Diagrammtypen die Reihenfolge nicht immer zwingend beachtet werden muss, wie es bei Sequenzdiagrammen der Fall ist. So kann es zum Beispiel bei

Klassendiagrammen egal sein in welcher Reihenfolge Klassen von einer Elternklasse erben und nur der Umstand des Erbens ist ausschlaggebend. Das vorgestellte grundlegende Konzept ist somit, mit unter Umständen erforderlichen Anpassungen, auch auf andere Diagrammtypen anwendbar.

Auch alle Anforderungen die an eine Erkennung von Security Design Anti-Pattern im Softwaredesign gestellt wurden sind mit dem ausgearbeiteten Konzept eingehalten worden. So wird für jedes potentiell unsichere Entwurfsmuster eine boolesche Aussage über das Vorhandensein in einem Softwaredesign getroffen. Durch diese ist eine eindeutige Aussage über das Vorkommen von sicherheitskritischen Mustern möglich. Weiter wird durch den ausgegebenen Ausschnitt aus dem Softwaredesign auch die Lokalität eines vorkommenden Anti-Patterns ausgegeben. Dies wird durch den Prototypen gezeigt, durch die Ausgabe der gefundenen Instanzen als kleines Diagramm bzw. als Ausschnitt aus dem Gesamtdiagramm, der im XMI Format spezifiziert ist. Dadurch ist die Bestimmung der Lokalität eines Anti-Patterns ersichtlich und kann anhand der Ausgabe auch weiter verarbeitet werden.

Die Anforderung der Zuverlässigkeit ist erfüllt, da die Suche in einem vorgegebenen Schema abläuft welches bei jeder Ausführung gleich ist. Folglich wird auch bei mehrfacher Suche nach dem gleichen Muster das Ergebnis gleich bleiben. Weiter kann durch die im Konzept vorgestellte Erweiterung des Suchalgorithmus, für die verzweigende Suche bei einer Übereinstimmung, die Zuverlässigkeit erhöht werden, da es hierdurch möglich ist mehrere Instanzen desselben Anti-Patterns in einem Softwaredesign zu finden.

Auch die Anforderungen an eine performante Suche wurden eingehalten, hierfür sind mehrere Aspekte genannt worden, wodurch die Suche beschleunigt wird. Einerseits wird durch die Prüfung auf eine Übereinstimmung mit dem Nicht Operator zu Beginn des Suchprozesses die Suche für einen Unterabschnitt des Baums frühzeitig abgebrochen und somit Performance gewonnen. Andererseits sind mehrere Möglichkeiten zur Parallelisierung aufgezeigt worden, wodurch auf modernen Computern ein Gewinn an Performance möglich ist da jene oft mehrere Prozessorkerne zur Verfügung haben. Diese Parallelisierung ist an verschiedenen Stellen möglich, beispielsweise bei der erweiterten Suche bei dem Verzweigen des Suchvorgangs, wenn mehrere Wege durch die Baumstruktur gegangen werden sollen. Oder auch mittles der parallelen Suche nach verschiedenen Anti-Pattern in einem Diagramm. Das die Parallelisierung möglich ist, wurde durch genanntes Vorgehen

bei der Suche nach verschiedenen Anti-Pattern in einem Diagramm mit dem Prototypen gezeigt. Es wird ersichtlich, dass das vorgestellte Konzept zur Suche von Security Design Anti-Pattern im Softwaredesign eine performante Lösung ist und auch Möglichkeiten zur Beschleunigung des Suchvorgangs bietet.

Weiter soll analysiert werden ob durch die Definition von Security Design Anti-Pattern und die Suche von diesen in einem Softwaredesign die Sicherheit von Programmen erhöht werden kann. Hierfür ist zu betrachten, dass es durch die Definition möglich ist eine Sammlung von potentiell unsicheren Lösungen zu erstellen und auch zu teilen. Dadurch wird die Möglichkeit geschaffen, Wissen über sichere bzw. unsichere Lösungen zu teilen und auch maschinell verarbeitbar zu machen. In Softwareprojekten wird es somit möglich Wissen zusammenzutragen und zu kapseln. Mit der Suche nach den definierten Anti-Pattern können im Softwaredesign potentielle Schwachstellen, oder Stellen in denen Vorgaben zur Absicherung verletzt wurden, entdeckt werden. In Bezug auf Softwareprojekten wird es möglich, durch das gekapselte Wissen in den Anti-Pattern das erstellte Softwaredesign von allen Mitgliedern zu testen und potentielle Verwundbarkeiten frühzeitig zu identifizieren. Auch wird die Möglichkeit geschaffen, automatisiert auf die Einhaltung von Vorgaben zur Verarbeitung von personenbezogenen Daten zu prüfen, da es durch die Definition von Anti-Pattern möglich ist häufige Fehler zu identifizieren. Das ist im Bezug auf die DSGVO von Vorteil, da in den Gründen für die Verordnung unter Absatz 78 beschrieben wird, dass Hersteller die Programme entwickeln die personenbezogene Daten verarbeiten, ermutigt werden sollen, schon bei der Gestaltung und Entwicklung den Datenschutz zu berücksichtigen.[10] Woraus sich schließen lässt, dass der Datenschutzaspekt frühzeitig in der Entwicklung berücksichtigt werden muss. Durch das vorgestellte Konzept ist es möglich bereits in der Design Phase der Softwareentwicklung, Verletzungen von Grundsätzen des Datenschutzes zu erkennen. Vorausgesetzt es wurden die verbreitetsten Fehler im Bezug auf den Datenschutz als Anti-Pattern definiert.

Abschließend kann festgehalten werden, dass das vorgestellte Konzept alle Anforderungen die an Security Design Anti-Pattern und deren Suche im Softwaredesign gestellt wurden umsetzt. Weiter folgt daraus, dass durch dieses Konzept die Sicherheit von Software schon frühzeitig in der Entwicklung gestärkt werden kann, wodurch auch die Qualität der Software gesteigert wird. Es wurde die Möglichkeit geschaffen, automatisiert potentielle Schwachstellen im Softwaredesign zu entdecken.

8 Zusammenfassung und Ausblick

In dieser Arbeit wurde ein Konzept für Security Design Anti-Pattern und deren Suche im Software Design ausgearbeitet. Es ist somit möglich potentielle Schwachstellen schon in der Entwurfsphase der Software zu identifizieren. Die Prototypimplementierung zeigt Beispielsweise für die Sequenzdiagramme die Umsetzbarkeit und auch, dass dieses Konzept auf weitere Diagramme des Software Designs anwendbar ist. Somit kann festgehalten werden, dass die Suche nach potentiellen Sicherheitsschwachstellen durch die Suche von Anti-Pattern im Software Design, eine sinnvolle und wünschenswerte Möglichkeit ist, damit die Sicherheit von Programmen erhöht werden kann. In dieser Arbeit sollte das zugehörige Konzept mit einer Prototypimlementierung ausgearbeitet werden was erfolgreich gelungen ist. In Zukunft sollte dieses Konzept in ein produktives System ausgeweitet werden. Hierfür gibt es noch weitere sinnvolle Funktionen und Erweiterungen die in dieses Konzept eingearbeitet werden können. Ein Teil von diesen Erweiterungen wird in den nachfolgenden Absätzen kurz vorgestellt.

Es ist möglich, das Konzept auf alle Arten von UML Diagrammen zu erweitern und möglichst viele verschiedene Diagrammtypen zu unterstützen. Folglich ergibt sich die Möglichkeit, Anti-Pattern für verschiedene Diagrammarten zu definieren. Durch die Unterstützung von verschiedenen Diagrammtypen wird es möglich sowohl dynamische als auch statische Eigenschaften in Anti-Pattern zu definieren. Dadurch wird die Menge der potentiellen Schwachstelle, die das System entdecken kann stark vergrößert. Außerdem wird das System dadurch flexibler und die Wiederverwendbarkeit damit vergrößert. Dies ist sinnvoll, da bei der Entwurfsphase von Software meist mehrere verschiedenen Diagramme verwendet werden um das Programm zu spezifizieren.[14] Somit können auch verschiedene Arten von potentiellen Schwachstellen identifiziert werden.

Durch die Erweiterung eines Open-Source Editors wie Eclipse oder UMLDesigner ergibt sich die Möglichkeit die Erweiterungen zur Definition von Security Design Anti-Pattern in diesen umzusetzen. Dadurch besteht die Möglichkeit, die Anti-Pattern mithilfe von grafischer Unterstützung zu definieren. Durch die Erweiterung besteht ebenso die Möglichkeit unvollständige Diagramme anzuzeigen, wodurch die gefundenen Instanzen besser visuell dargestellt und dadurch einfacher im Diagramm gefunden werden können. Es ist auch denkbar, dass eine Erweiterung anhand der gefundenen Teildiagramme die Stelle im Diagramm visuell hervorhebt, durch eine farbliche Markierung oder auf andere Weise.

Durch die im Konzept beschriebenen Erweiterungen für die Suche nach Security Design Anti-Pattern im Abschnitt 5.5, kann die Suche deutlich verbessert werden. Hierdurch ist es möglich, mehrere Instanzen eines Anti-Pattern im Software Design zu identifizieren und auch in Sequenzdiagrammen zu finden, wenn diese über mehrere Ebenen verschachtelt sind. Dies ist notwendig, um die Suche genauer und robuster zu gestalten. Vor allem bei großen Software Design Diagrammen ist dies nötig, da bei diesen die Verschachtelung größer ist und auch weil in diesen es nicht unrealistisch ist, dass ein Anti-Pattern mehrfach in einem Diagramm vorhanden ist.

Zusätzlich besteht die Möglichkeit durch Reverse Engineering Software Design Diagramme aus dem Quelltext zu generieren. Diese Möglichkeit befindet sich derzeit zum Teil noch in der Forschung.[2, 5, 16, 24, 27] Die automatisch generierten Diagramme können auf Anti-Pattern überprüft werden, wodurch in dem umgesetzten Programm potentielle Schwachstellen gefunden werden können. Dadurch ist es möglich, sowohl Design- als auch Implementierungsentscheidungen, die zu einer unsicheren Software führen, zu entdecken.

Die Integration in eine automatisierte Umgebung wie eine CI/CD Plattform bietet die Möglichkeit kontinuierlich bereits während der Entwicklung auf Security Design Anti-Pattern zu prüfen. Diese Technik von CI/CD wird in der modernen Softwareentwicklung verwendet um mit jedem neuen Eintrag in einer Versionsverwaltung Tests auszuführen, die Software zu bauen und gegebenenfalls auch auszuliefern. Aufgrund der Ergänzung der Tests in diesem Prozess durch die Suche nach Security Design Anti-Pattern, ergibt sich die Möglichkeit automatisiert und kontinuierlich auf potentielle Schwachstellen zu prüfen. Folglich können diese frühzeitig identifiziert und behoben werden. Es besteht die

Möglichkeit in Design Diagrammen, die in eine Versionsverwaltung eingespielt werden, nach den Security Design Anti-Pattern zu suchen, oder wie beschrieben durch Reverse Engineering in dem Quelltext enthaltene Anti-Pattern. Die sinnvollste Umsetzung ist, beide Techniken anzuwenden, da so auch Design Schwachstellen entdeckt werden können, die nicht im Software Design enthalten sind und durch eine Abweichung der Implementierung zum Software Design in das Programm eingefügt wurden.

Eine weitere Möglichkeit für eine Erweiterung des Konzeptes entsteht durch das Anlernen eines ML (Machine Learning) Algorithmus. Dieser wird durch Quelltext und mittels Reverse Engineering gefundenen Security Design Anti-Pattern angelernt. Hieraus ergibt sich die Forschungsfrage, ob nach dem erfolgreichen anlernen des ML Algorithmus es diesem möglich ist, aus Quelltext automatisiert und selbständig potentielle Sicherheitslücken zu entdecken und im nächsten Schritt sogar auszunutzen. Durch eine solche Automatisierung besteht die Möglichkeit, noch mehr und komplexere Anti-Pattern zu entdecken, da durch das ML die Möglichkeit besteht, dass Kombinationen oder sogar neue Anti-Pattern entdeckt werden, die nicht genau so spezifiziert wurden, da es der ML Funktion möglich sein kann diese aufgrund von Verknüpfungen zu entdecken.

Aus diesen Punkten ergibt sich, dass das in dieser Arbeit vorgestellte Konzept noch Potential für Erweiterungen bietet. Das Grundkonzept wurde ausgearbeitet und durch die Prototypimplementierung umgesetzt. Das Konzept von Security Design Anti-Pattern und deren Suche im Software Design am Beispiel von Sequenzdiagrammen ist hierdurch abgeschlossen und sollte in Zukunft weiter ausgebaut sowie in ein produktives System umgesetzt werden.

Literaturverzeichnis

[1] Alfred V. Aho, John E. Hopcroft und Jeffrey D. Ullman.
The Design and Analysis of Computer Algorithms.
Pearson Education (US), 1. Jan. 1974. 480 Seiten. ISBN: 0201000296.

[2] Chafik BAIDADA, El Mahi BOUZIANE und Abdeslam JAKIMI.
„A New Approach for Recovering High-Level Sequence Diagrams from Object-Oriented Applications Using Petri Nets“.
In: *Procedia Computer Science* 148 (2019), Seiten 323–332.
DOI: 10.1016/j.procs.2019.01.040.

[3] Boaz Barak, Chi-Ning Chou, Zhixian Lei, Tselil Schramm und Yueqi Sheng.
„(Nearly) Efficient Algorithms for the Graph Matching Problem on Correlated Random Graphs“. In: *Advances in Neural Information Processing Systems 32*.
Herausgegeben von H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox und R. Garnett. Curran Associates, Inc., 2019, Seiten 9190–9198. URL:
http://papers.nips.cc/paper/9118-nearly-efficient-algorithms-for-the-graph-matching-problem-on-correlated-random-graphs.pdf.

[4] Paul E Black. *Juliet 1.3 test suite: changes from 1.2*. Technischer Bericht. Juni 2018.
DOI: 10.6028/nist.tn.1995.

[5] L. C. Briand, Y. Labiche und J. Leduc. „Toward the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software“.
In: *IEEE Transactions on Software Engineering* 32.9 (Sep. 2006), Seiten 642–663.
DOI: 10.1109/tse.2006.96.

[6] BSI - Bundesamt für Sicherheit in der Informationstechnik.
IT-Grundschutz-Kataloge. Website.
URL: https://download.gsb.bund.de/BSI/ITGSK/IT-Grundschutz-Kataloge_2016_EL15_DE.pdf (aufgerufen am 02. 07. 2020).

[7] BSI - Bundesamt für Sicherheit in der Informationstechnik.
Kryptographische Verfahren: Empfehlungen und Schlüssellängen. Website.
URL: https://download.gsb.bund.de/BSI/ITGSK/IT-Grundschutz-Kataloge_2016_EL15_DE.pdf (aufgerufen am 28. 06. 2020).

[8] BSI - Bundesamt für Sicherheit in der Informationstechnik.
Sicherheit von Webanwendungen: Maßnahmenkatalog und Best Practices. Website.
URL: https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/Studien/WebSec/WebSec.pdf (aufgerufen am 28. 06. 2020).

[9] Andrew D.J. Cross, Richard C. Wilson und Edwin R. Hancock.
„Inexact graph matching using genetic search“.
In: *Pattern Recognition* 30.6 (Juni 1997), Seiten 953–970.
DOI: 10.1016/s0031-3203(96)00123-9.

[10] Europäische Parlament und Rat. *Verordnung zum Schutz natürlicher Personen bei der Verarbeitung personenbezogener Daten, zum freien Datenverkehr und zur Aufhebung der Richtlinie 95/46/EG (Datenschutz-Grundverordnung)*. URL: https://eur-lex.europa.eu/legal-content/DE/TXT/PDF/?uri=CELEX:32016R0679&from=DE (aufgerufen am 30. 06. 2020).

[11] Fraunhofer-Institut für Sichere Informationstechnologie SIT.
Entwicklung sicherer Software durch Security by Design. Website.
URL: https://www.kastel.kit.edu/downloads/Entwicklung_sicherer_Software_durch_Security_by_Design.pdf (aufgerufen am 30. 06. 2020).

[12] Matthias Geirhos. *Entwurfsmuster. Das umfassende Handbuch.* 1. Auflage.
Rheinwerk Verlag GmbH, 1. Juni 2015. ISBN: 978-3-8362-2762-9.

[13] Thomas Heyman, Koen Yskout, Riccardo Scandariato und Wouter Joosen.
„An Analysis of the Security Patterns Landscape“. In: *Third International Workshop*

on Software Engineering for Secure Systems (SESS'07: ICSE Workshops 2007). IEEE, Mai 2007, Seiten 3–3. DOI: 10.1109/sess.2007.4.

[14] Christoph Kecher, Alexander Salvanos und Ralf Hoffmann-Elbern. *UML 2.5*. 6. Auflage. Rheinwerk Verlag GmbH, Dez. 2017. ISBN: 3836260182.

[15] Paul Kelly. „A congruence theorem for trees“. In: *Pacific Journal of Mathematics* 7.1 (März 1957), Seiten 961–968. DOI: 10.2140/pjm.1957.7.961.

[16] E. Korshunova, M. Petkovic, M.G.J. van den Brand und M.R. Mousavi. „CPP2XMI: Reverse Engineering of UML Class, Sequence, and Activity Diagrams from C++ Source Code“. In: *2006 13th Working Conference on Reverse Engineering*. IEEE. DOI: 10.1109/wcre.2006.21.

[17] Longbin Lai u. a. „Distributed Subgraph Matching on Timely Dataflow“. In: *Proceedings of the VLDB Endowment* 12.10 (Juni 2019), Seiten 1099–1112. DOI: 10.14778/3339490.3339494.

[18] M. Laverdiere, A. Mourad, A. Hanna und M. Debbabi. „Security Design Patterns: Survey and Evaluation“. In: *2006 Canadian Conference on Electrical and Computer Engineering*. IEEE, 2006, Seiten 1605–1608. DOI: 10.1109/ccece.2006.277727.

[19] Lorenzo Livi und Antonello Rizzi. „The graph matching problem“. In: *Pattern Analysis and Applications* 16.3 (Aug. 2012), Seiten 253–283. DOI: 10.1007/s10044-012-0284-8.

[20] P. H. Meland und J. Jensen. „Secure Software Design in Practice“. In: *2008 Third International Conference on Availability, Reliability and Security*. IEEE, März 2008, Seiten 1164–1171. DOI: 10.1109/ARES.2008.48.

[21] Microsoft. *Vereinfachte Implementierung des Microsoft SDL*. Website. URL: https://www.microsoft.com/de-de/download/details.aspx?id=12379 (aufgerufen am 03. 07. 2020).

[22] OMG - Object Management Group.
OMG Unified Modeling Language (OMG UML). Website.
URL: https://www.omg.org/spec/UML/2.5/PDF (aufgerufen am 12. 05. 2020).

[23] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood und Marc McConley. „Automated Vulnerability Detection in Source Code Using Deep Representation Learning“. In: *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, Dez. 2018. DOI: 10.1109/icmla.2018.00120.

[24] Umair Sabir, Farooque Azam, Sami Ul Haq, Muhammad Waseem Anwar, Wasi Haider Butt und Anam Amjad. „A Model Driven Reverse Engineering Framework for Generating High Level UML Models From Java Source Code“. In: *IEEE Access* 7 (2019), Seiten 158931–158950.
DOI: 10.1109/access.2019.2950884.

[25] Uwe Schöning. „Graph isomorphism is in the low hierarchy“.
In: *Journal of Computer and System Sciences* 37.3 (Dez. 1988), Seiten 312–323.
DOI: 10.1016/0022-0000(88)90010-4.

[26] Laurens Sion, Katja Tuma, Riccardo Scandariato, Koen Yskout und Wouter Joosen. „Towards Automated Security Design Flaw Detection“. In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*. IEEE, Nov. 2019, Seiten 49–56. DOI: 10.1109/asew.2019.00028.

[27] K. Taniguchi, T. Ishio, T. Kamiya, S. Kusumoto und K. Inoue.
„Extracting Sequence Diagram from Execution Trace of Java Program“.
In: *Eighth International Workshop on Principles of Software Evolution (IWPSE'05)*. IEEE, Seiten 148–151. DOI: 10.1109/iwpse.2005.19.

Ehrenwörtliche Erklärung

Ich erkläre hiermit, dass ich die Arbeit selbstständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Ingolstadt, den 18. August 2020

.....

Timo Meilinger