Bachelorarbeit

Vererbung von Sicherheitseigenschaften in objektorientierten Kontexten

Volkan Topcu
1. Juli 2019

Gutachter: Prof. Dr. Jan Jürjens
Sven Peldszus

Prof. Dr. Jan Jürjens
Institut für Softwaretechnik
Institut für Informatik
Universtität Koblenz
Universitätsstraße 1
56070 Koblenz
https://rgse.uni-koblenz.de

Volkan Topcu
vtopcu54@uni-koblenz.de
Matrikelnummer: 215101132
Studiengang: Bachelor of Science Informatik
Prüfungsordnung: BPO2012

Sotwaretechnik
Thema: Vererbung von Sicherheitseigenschaften in objektorientierten Kontexten

Eingereicht: 1. Juli 2019

Betreuer: Sven Peldszus

Prof. Dr. Jan Jürjens
Institut für Softwaretechnik
Institut für Informatik
Universtität Koblenz
Universitätsstraße 1
56070 Koblenz

Ehrenwörtliche Erklärung

Hiermit bestätige ich, dass die vorliegende Arbeit von mir selbständig verfasst wurde und ich keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe und die Arbeit von mir vorher nicht in einem anderen Prüfungsverfahren eingereicht wurde. Die eingereichte schriftliche Fassung entspricht der auf dem elektronischen Speichermedium (CD-ROM).

Koblenz, den 1. Juli 2019

Volkan Topcu

Abstrakt

Für die Entwicklung sicherer Softwaresysteme erweitert UMLsec die UML durch die Definition von Sicherheitsanforderungen, die erfüllt sein müssen. In Klassendiagrammen können Attribute und Methoden von Klassen Sicherheitseigenschaften wie *secrecy* oder *integrity* haben, auf die nicht authorisierte Klassen keinen Zugriff haben sollten. Vererbungen zwischen Klassen erzeugen eine Komplexität und werfen die Frage auf, wie man mit der Vererbung von Sicherheitseigenschaften umgehen sollte. Neben der Option in sicherheitskritischen Fällen auf Vererbungen zu verzichten, gibt es im Gebiet der objektorientierten Datenbanken bereits viele allgemeine Recherchen über die Auswirkungen von Vererbung auf die Sicherheit. Das Ziel dieser Arbeit ist es, Ähnlichkeiten und Unterschiede von der Datenbankseite und den Klassendiagrammen zu identifizieren und diese Lösungsansätze zu übertragen und zu formalisieren. Eine Implementierung des Modells evaluiert, ob die Lösungen in der Praxis anwendbar sind.

Abstract

For the development of secure software systems, UMLsec extends UML by defining security requirements that have to be fulfilled. In class diagrams, attributes and methods of classes may have security properties like *secrecy* or *integrity* which should not be accessed by unauthorized classes. Inheritance between classes generates complexity and poses a question about how to deal with the inheritance of security properties. Beside the option to neglect inheritance in critical security cases, there has been many general research done for object-oriented databases about the impact of inheritance on security. The purpose of this paper is to identify similarities and differences of elements from the database side and the class diagram, project those solution approaches and formalize them. An implementation of the model is going to evaluate, if the solution is applicable in practice.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Im Folgenden wird in Unterkapitel 1.1 die Arbeit zunächst motiviert und eine Reihe von wichtigen Hintergrundinformationen bereitgestellt, ehe auf die eigentliche Fragestellung und Ziele der Arbeit eingegangen wird. Im Anschluss darauf wird in Unterkapitel 1.2 der Aufbau der Arbeit dargestellt.

1.1 Motivation und Hintergrund

Laut einer von Jupiter durchgeführten Studie über die Zukunft von Cyberkriminalität und -sicherheit, steigen die durch Datenschutzverletzungen ausgelösten Kosten ständig an und werden aufgrund der zunehmenden Digitalisierung von Unternehmen und Verbraucherleben bis zum Jahr 2020 einen geschätzten Schaden von 2,5 Billionen Dollar erreichen [Chr16] [Moa17]. Dies zeigt die Wichtigkeit von IT-Sicherheit aus wirtschaftlicher Sicht.

Ein Anwendungsbereich, bei denen Sicherheitsanalysen sehr wichtig sind und das in dieser Arbeit als Fallbeispiel zur Demonstration von verschiedenen Szenarios verwendet wird, sind Chat-Applikationen. In diesen können Benutzer mit anderen Benutzern über das für jeden zugängliche Internet kommunizieren. Da die ausgetauschten Nachrichten Einblicke in das Privatleben der Benutzer geben, ist es von großer Bedeutung, dass diese Nachrichten vor unbefugten Zugriffen geschützt sind. Zwei wichtige Sicherheitseigenschaften, die für diese Nachrichten gelten müssen, sind Vertraulichkeit und Integrität. Vertraulichkeit handelt von der Festlegung und Durchführung von angebrachten Zugangsebenen für Informationen, um Leckagen zu unerwünschten Personen zu vermeiden. Integrität besagt, dass die Daten vor Veränderungen durch Unbefugte oder ungewollten Änderungen geschützt werden müssen [Per08].

Es ist erstrebenswert, solche Sicherheitsanforderungen an Daten in der frühen Phase der Softwareentwicklung, beispielsweise mittels UML, zu modellieren, etwa um von Anfang an sicherheitsrelevante Aspekte in die Modellierung miteinzubeziehen. Zu diesem Zweck kann UML mit UMLsec erweitert werden, um Anforderungen an die Sicherheit zu formulieren [Jür05]. Diese können sich auch auf die Vertraulichkeit und Integrität von gewissen Daten beziehen. UML verfügt über 13 verschiedene Diagrammarten. Eines davon ist das UML Klassendiagramm aus der Kategorie der Strukturdiagramme. Mit ihr werden die Beziehung zwischen Klassen in objektorientierten Systemen graphisch dargestellt.

Ein wesentliches Merkmal des objektorientierten Programmierparadigmas ist das Konzept der Vererbung. Man strukturiert Klassen in einer Hierarchie, wobei Klassen niedriger in der Hierarchie Unterklassen genannt werden und Klassen höher in

der Hierarchie Oberklassen. Auf diese Weise erhält man Generalisierungen und Spezialisierungen von Klassen. Eine Subklasse erbt alle Felder von seinen Oberklassen, welche dann auch ihr verfügbar sind. Felder, die von Oberklassen geerbt werden, können auch überschrieben werden, was in der objektorientierten Programmierung oft als *Overriding* (Überschreiben) bezeichnet wird. Diese Felder können dann eine andere Form der Implementierung haben als die überschriebenen Methode ihrer Oberklasse.

In UMLsec können alle Member von Klassen Sicherheitseigenschaften wie secrecy und integrity besitzen. Der Secure Dependency Check stellt sicher, dass Entwickler daran denken, dass diese Felder nur von Klassen aufgerufen werden, die ihre Sicherheitseigenschaften respektieren. Dadurch, dass Felder in UMLsec mit Sicherheitseigenschaften annotiert sind und diese Felder in UML vererbt werden können, ist auch eine Vererbung von Felder mit Sicherheitseigenschaften denkbar. Die Vererbung von Sicherheitseigenschaften würde jedoch kritische Sicherheitsfragen aufwerfen. In Abbildung 1.1 ist hierfür ein Abschnitt aus der ChatApp eingefügt, welche in dieser Arbeit als Fallbeispiel verwendet wird.

Die Klasse *Benutzer* hat call-Abhängigkeiten zu den Klassen *SecretChat* und *Gruppenchat*. Sie kann Methoden aus diesen beiden Klassen aufrufen. Für die Methode *senden* gilt in der Klasse *SecretChat* die Sicherheitseigenschaft *secrecy*. Da diese Sicherheitseigenschaft nicht im Clienten *Benutzer* vorhanden ist, schlägt der *Secure Dependency Check* fehl und das Modell muss verändert werden.

Gruppenchat ist eine Unterklasse von *SecretChat*. Sie erbt unter anderem die Methode *senden*, spezifiziert aber in ihrer Klassendefinition keine Sicherheitseigenschaften für diese Methode. *Benutzer* könnte mittels der call-Abhängigkeit auf *Gruppenchat* die Methode *senden* aufrufen. Dabei würde der *Secure Dependency Check* nicht fehlschlagen, es allerdings dennoch zu einer Verletzung unserer Sicherheitsan-

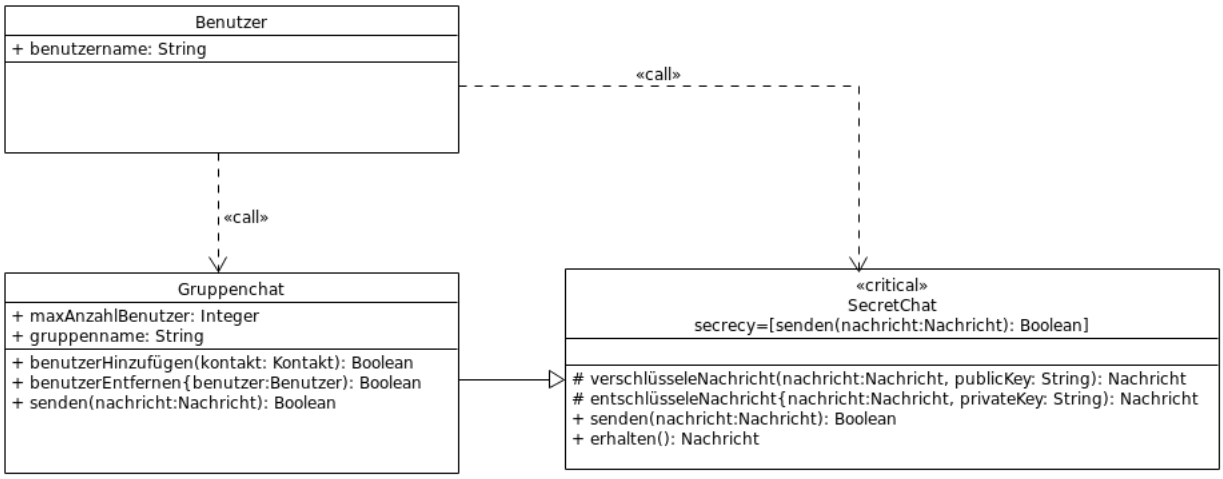

Abbildung 1.1: Ausschnitt aus der ChatApp. Die Klasse *SecretChat* wird von *Gruppenchat* geerbt. Für die Methode *senden* sind nur in *SecretChat* Sicherheitseigenschaften definiert.

forderung führen. Mögliche Sicherheitslücken wie diese zeigen auf, dass das Konzept der Vererbung nicht ohne weitere Maßnahmen in UMLsec verwendet werden kann.

In UMLsec wird das Konzept der Vererbung abstrahiert, sodass Fälle wie in Abbildung 1.1 nicht eintreten können. Allerdings wurden ebenso zu diesem Thema bereits generelle Untersuchungen im Bereich der objektorientierten Datenbanken durchgeführt. Verschiedene Zugriffskontrollmodelle regeln dort auf der Basis von verschiedenen Kriterien, welcher Benutzer auf welche Daten zugreifen und welche Daten modifizieren darf. Eine wichtige Richtlinie liefert dabei das Bell-LaPadula Modell, das in besonders sicherheitsrelevanten Systemen Anwendung findet. Eine neuere Form der Zugriffskontrolle ist die rollenbasierte Zugriffskontrolle (Role-based Access Control, RBAC). Es erlaubt Vererbungen von Erlaubnissen und kann so modifiziert werden, dass es die Bell-La-Padula Richtlinien durchsetzt.

In dieser Arbeit wird gezeigt, welche Ähnlichkeiten zwischen den Problemen und Lösungen auf der Datenbankseite mit den Vererbungen von Sicherheitseigenschaften im Klassendiagramm bestehen, und eine Lösung erarbeitet, welche Vererbungen von Sicherheitseigenschaften gemäß den Bell-LaPadula Richtlinien erlaubt. Dieser Lösungsansatz wird dann in einem formalen Modell formuliert, das allgemeingültig ist. Dieses Modell wird im Anschluss erweitert um auch Überschreibungen von Feldern miteinzubeziehen. Anschließend wird das Modell evaluiert, indem es in der Programmiersprache Java implementiert und überprüft wird, ob das Modell in der Praxis anwendbar ist.

1.2 Aufbau der Arbeit

In Kapitel 2 wird zunächst das in der Arbeit verwendete Fallbeispiel *ChatApp* im Detail vorgestellt. Im Anschluss werden in Kapitel 3 alle benötigten Grundlagen, die zum Verständnis der Arbeit wichtig sind, erläutert: Der Vererbungsmechanismus im objektorientierten Paradigma, UMLsec mit seinen für diese Arbeit relevanten Elementen sowie verschiedene Zugriffskontrollmodelle wie Discretionary Access Control, Mandatory Access Control, Role-based Access Control und das Bell-LaPadula Modell. Ab Kapitel 4 beginnt der Hauptteil dieser Arbeit. Dort werden Zusammenhänge zwischen den Zugriffskontrollmodellen und dem Klassendiagramm aufgezeigt und diese dann in ein formales Modell überführt. Anschließend wird die Auswirkung von Memberüberschreibungen auf die Sicherheit erörtert und das formale Modell dann mit diesen Erkenntnissen erweitert. Daraufhin folgt in Kapitel 5 eine Implementierung des Modells als CARiSMA Plugin. Diese Implementierung wird darauffolgend in Kapitel 6 validiert, indem Testfälle ausgeführt werden und die Ergebnisse des neuen Plugins mit dem bisher vorhandenen Plugin verglichen werden. In Kapitel 7 werden relevante Arbeiten vorgestellt, welche ebenfalls als Ziel haben, eine stärkere Sicherheit durchzusetzen und gleichzeitig über das Konzept der Vererbung verfügen. Im achten und letzten Kapitel folgt ein Fazit, bei dem die Ergebnisse dieser Arbeit zusammengefasst werden und anschließend ein Ausblick auf zukünftige Arbeiten in diesem Gebiet gegeben wird.

2 Anwendungsbeispiel

Eine Anwendung, auf der Sicherheitsanalysen sehr wichtig sind, sind Chat-Applikationen. In diesen kommunizieren Benutzer mit anderen Benutzern über das Internet, das für jeden zugänglich ist. Die ausgetauschten Nachrichten sind sehr sicherheitskritisch: Es ist im Interesse des Benutzers, dass seine versendeten Nachrichten nur von den gewünschten Adressaten gelesen werden und nicht von unbefugten Dritten. Gleichzeitig möchte er sich vergewissern können, dass die empfangenen Nachrichten tatsächlich von seinem Gesprächspartner stammen und sie nicht von einem Dritten unter dessen Namen versendet wurde. Auch muss für alle ausgetauschten Nachrichten gelten, dass eine Veränderung dieser Nachrichten von Dritten unterbunden wird. Aus diesen Gründen sind Chat-Applikationen sehr sicherheitskritische Systeme und deshalb wird in dieser Arbeit als Anwendungsbeispiel eine Chat-Applikation namens *ChatApp* verwendet. Diese ChatApp soll einige sicherheitsrelevante Szenarios rund um die Vererbung von Sicherheitseigenschaften bereitstellen. Sie ist nur Mittel zum Zweck und hat keine tiefere Verbindung zum Thema dieser Arbeit.

In Abb. 2.1 ist die ChatApp in einem UML Klassendiagramm modelliert. Sie wird in dieser Arbeit als Beispiel benutzt, um sicherheitskritische Szenarien vorzustellen. Sie ist aus der Sicht eines Benutzers modelliert. Es wird zwischen dem Benutzer und seinen Kontakten, mit denen er Nachrichten über die ChatApp austauschen kann, unterschieden. Der Benutzer verfügt über einen Benutzernamen, einer E-Mail Adresse sowie einem geheimen Passwort. Mit diesen kann sich ein neuer Benutzer erstmals in unserer Applikation registrieren. Außerdem kann sich ein registrierter Benutzer mit seinem Benutzernamen und Passwort immer wieder im System anmelden. Weiterhin verfügt er über einen Schlüsselpaar, bestehend aus einem privaten und öffentlichen Schlüssel, die für sicherheitsrelevante Mechanismen in der ChatApp benötigt werden. Der Benutzer kann über beliebig viele Kontakte verfügen und neue Kontakte hinzufügen und entfernen. Da aus der Sicht eines Benutzers modelliert, ist in der Sicht dieses Modells lediglich der Benutzername und der öffentliche Schlüssel der Kontakte bekannt. Möchte der Benutzer Nachrichten mit einem seiner Kontakte austauschen, so muss er ein neues Chatfenster öffnen. Dabei hat er die Wahl zwischen drei verschiedenen Chat-Modi:

1. **Ein normaler Chat**. In diesem kann er beliebig viele Nachrichten versenden und empfangen. Hierfür verfügt die Klasse über die beiden Methoden *senden* und *empfangen*. Die Methode *senden* nimmt als Parameter *Nachricht* (die zu sendende Nachricht) entgegen und gibt ein Boolean zurück, das angibt, ob die

Nachricht erfolgreich gesendet wurde. Diese Nachrichten sind unverschlüsselt und unsigniert.

2. **Ein geheimer Chat**. Dieser verfügt über die selben Funktionen wie ein normaler Chat (d.h, es verfügt ebenso über die Methoden *senden* und *empfangen*, die in dieser Klasse überschrieben werden), verschlüsselt aber zusätzlich die gesendeten Nachrichten. Dafür wird der private und öffentliche Schlüssel des Benutzers sowie der öffentliche Schlüssel der Kontakte von den beiden Methoden *verschlüsseleNachricht* und *entschlüsseleNachricht* als Parameter verwendet sowie beide Male die zu ver- oder entschlüsselnde Nachricht. Auf diese Weise wird die Vertraulichkeit der ausgetauschten Nachrichten gewährleistet.

3. **Ein sicherer Chat**. Dieser Chat erweitert den geheimen Chat mit der Funktion, die ausgetauschten Nachrichten zu signieren und die Signatur zu überprüfen, um damit die Integrität des Senders sicherzustellen. Dafür verwendet es die Methode *signiereNachricht* (zum Signieren einer Nachricht) und *überprüfeSignatur* (zum Verifizieren der Signatur) mit den Parametern privater oder öffentlicher Schlüssel sowie die zu signierende oder bereits signierte Nachricht. Die beiden Methoden zur Verschlüsselung werden unverändert von *SecretChat* geerbt und die Methoden *senden* und *empfangen* überschrieben.

Möchte man Nachrichten mit mehreren Personen austauschen, so bietet sich der Gruppenchat an. In einen Gruppenchat können beliebig viele Kontakte (mindestens ein Kontakt) des Benutzers hinzugefügt werden. Die ausgetauschten Nachrichten sind für alle Gruppenmitglieder lesbar. Außerdem ist es möglich, Kontakte wieder aus der Gruppe zu entfernen. Ebenso wie im geheimen Chat soll die Vertraulichkeit der Nachrichten durch eine Verschlüsselung gewährleistet werden. Ein Gruppenchat hat eine Beschränkung an eine maximale Anzahl von Benutzern und hat einen Namen.

Eine Nachricht hat einen Inhalt in Textform und verfügt über einen Datum, an dem es versendet wurde. Im Laufe dieser Arbeit wird diese Applikation noch um einige Funktionen erweitert, um speziellere Szenarien aufzuzeigen, aber zunächst wird diese Form der Anwendung genügen. Um speziellere Fälle aufzuzeigen, werden auch nur kleinere Ausschnitte aus diesem Klassendiagramm verwendet.

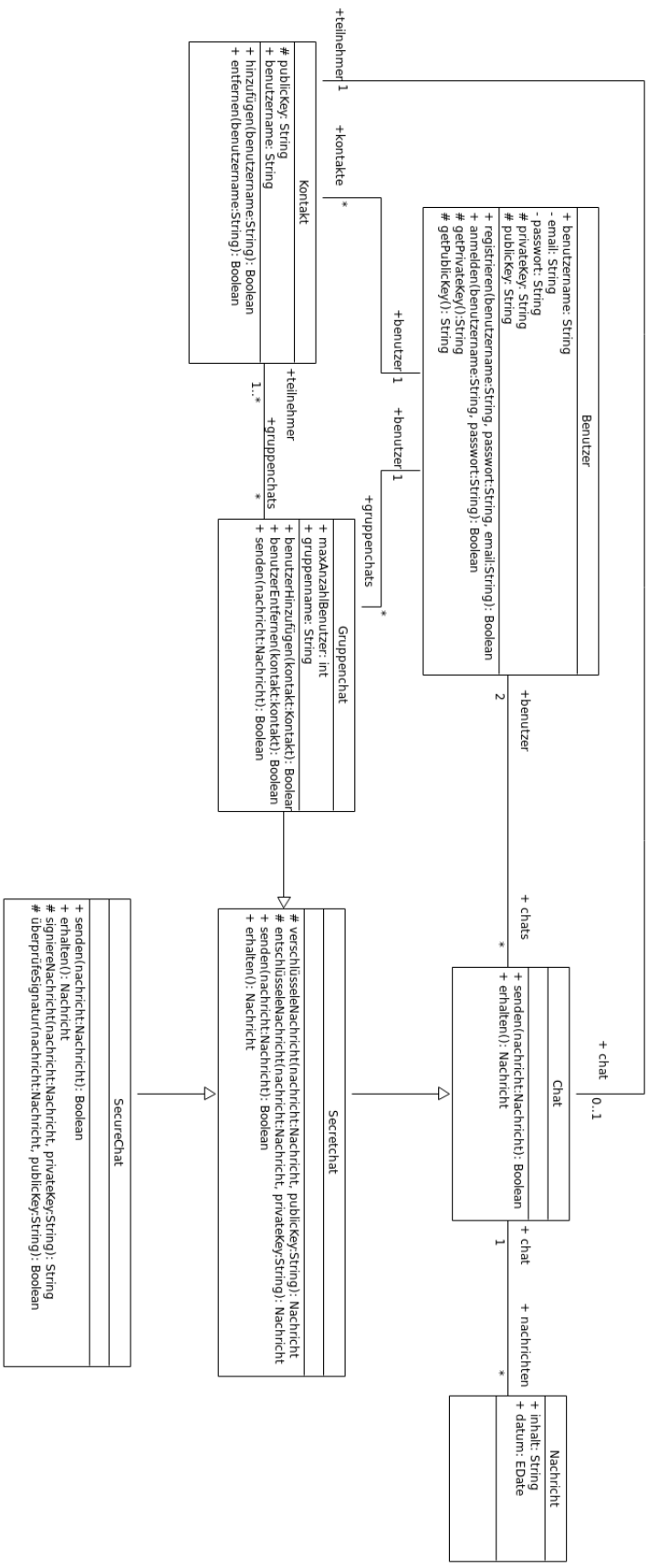

Abbildung 2.1: Die ChatApp modelliert als UML Klassendiagramm.

3 Grundlagen

In diesem Kapitel werden die Grundlagen erläutert, die für das Verständnis dieser Arbeit notwendig sind. Nachdem zunächst in Unterkapitel 3.1 ein kurzer Einblick in UML gegeben wird, wird in Unterkapitel 3.2 das Konzept der Vererbung vorgestellt, da es dort einige Details gibt, die in dieser Arbeit wichtig sind. So gilt seit UML 2.0 eine neue Regel, dass beim Subclassing eine überschreibende Methode als Parameter nur den gleichen Typen oder einen Subtypen als Parameter haben darf. Außerdem gibt es einige Differenzen zwischen Vererbung in UML und in der Programmiersprache Java, welche von vielen als Referenz genommen wird. Auf diese Diffrenzen wird in Unterkapitel 3.3 eingegangen, um Missverständnisse vorzubeugen. Anschließend wird in 3.4 UMLsec vorgestellt. UMLsec ist eine Erweiterung von UML, um Sicherheitseigenschaften zu definieren und zu modellieren. Mit UMLsec wird dann die ChatApp um Sicherheitseigenschaften erweitert und Sicherheitsanforderungen festgelegt, die im System gelten müssen. Es wird deutlich werden, dass Vererbungen zwischen Klassen in der ChatApp Auswirkungen auf definierte Sicherheitseigenschaften haben, aber Vererbung in UMLsec bisher abstrahiert wird. Daraufhin werden in Unterkapitel 3.5 verschiedene Zugriffskontrollmodelle wie das Bell-LaPadula Modell oder Role-based Access Control vorgestellt. In Role-based Access Control gibt es ebenfalls das Konzept der Vererbung. Lösungen dort werden uns in dieser Arbeit als Idee dienen, um Vererbungen von Sicherheitseigenschaften zu ermöglichen.

3.1 UML

Das in Abbildung 2.1 aufgezeigte Modell ist ein Diagramm unserer ChatApp modelliert als ein UML Klassendiagramm. Die Unified Modeling Language (UML) ist ein weit verbreiterter Standard für die Modellierung von Softwaresystemen von der Object Management Group (OMG) [OMG17]. Sie verfügt über 13 verschiedene Diagrammarten, allerdings wird für diese Arbeit das UML Klassendiagramm aus der Kategorie der Strukturdiagramme genügen. Ein Klassendiagramm beschreibt die Struktur eines Systems, indem es Klassen und ihre Beziehungen graphisch darstellt.

3.2 Vererbung

Zwischen verschiedenen Klassen gibt es des Öfteren Ähnlichkeiten. Oft besitzen ähnliche Klassen über die selben Attribute und Methoden. Indem man Klassen in einer Hierarchie ordnet, bei denen Klassen mit einer niedrigeren Hierarchie *Unterklassen* und Klassen mit einer höheren Hierarchie *Oberklassen* genannt werden, erhält man Spezialisierungen und Generalisierungen von Klassen. Mit dem Konzept

der Vererbung kann man also Ähnlichkeiten in den Oberklassen generalisieren und Unterschiede in den Unterklassen spezialisieren.

Dies ist nützlich, da beispielsweise auf diese Art meist für Beziehungen, die zwischen Klassen gelten sollen, nur eine Assoziation notwendig ist, um eine Beziehung zwischen mehreren Klassen zu beschreiben. Im Klassendiagramm für die ChatApp in Abbildung 2.1 ist z.B. nur eine Assoziation zwischen der Klasse *Nachricht* und *Chat* gegeben. Da die Klasse *Chat* aber eine Generalisierungen der Klassen *SecretChat* und *SecureChat* ist, sind damit auch automatisch alle Beziehung zwischen der Klasse *Nachricht* und den Klassen *SecretChat* und *SecureChat* definiert, nämlich auf die selbe Weise wie zwischen ihrer Oberklasse *Chat* und der Klasse *Nachricht*. Für diese gilt in diesem Fall eine Eins-zu-Viele-Beziehung. Man hat daher also eine Modularität und Veränderungen am Programmcode können einfacher umgesetzt werden, da nur eine Änderung an einer Stelle notwendig wird statt an mehreren Stellen.

Eine Unterklasse erbt rekursiv alle Attribute und Methoden von seinen Oberklassen. Attribute und Methoden, die in den jeweiligen Unterklassen überschrieben werden, werden in einem UML Klassendiagramm erneut in der Unterklasse aufgeführt. Wenn Attribute und Methoden nicht überschrieben und von den Oberklassen unverändert übernommen werden, so wird in der Unterklasse auf eine erneute Auflistung dieser Attribute und Methoden verzichtet.

In Abbildung 3.1 ist ein Ausschnitt aus der ChatApp dargestellt. Zu sehen sind die drei Klassen *Chat*, *SecretChat* und *SecureChat* und ihre Vererbungshierarchien. Die Klasse *Chat* ist die Oberklasse. Sie beinhaltet die beiden Methoden *senden* und *erhalten*. Die Unterklasse *SecretChat* erbt von ihr diese beiden Methoden und definiert darüber hinaus noch die beiden Methoden *verschlüsseleNachricht* sowie *entschlüsseleNachricht*. Da die beiden geerbten Methoden überschrieben werden, sind diese dennoch erneut in der Klasse aufgeführt. Die dritte Klasse, *SecureChat*, erbt wiederum von seiner Oberklasse *SecretChat* und damit auch rekursiv von der Oberklasse *Chat*. Damit erbt sie die vier Methoden *senden*, *erhalten*, *verschlüsseleNachricht* und *entschlüsseleNachricht*. Da nur die ersten beiden Methoden überschrieben werden sollen, wurden die Methoden zum Ent- und Verschlüsseln innerhalb der Klasse nicht erneut aufgeführt. Die Klasse verfügt neben den geerbten Methoden weiterhin über die beiden Methoden *signiereNachricht* und *überprüfeSignatur*, mit der Nachrichten signiert und ihre Signature überprüft werden sollen.

Seit UML 2.0 hat Subclassing eine präzisere Definition in Bezug auf Überschreibungen als in den Versionen zuvor. Eine Unterklasse darf die Parameter bei geerbten Methoden nur mit Subtypen von diesen ersetzen [BG04]. So werden im Klassendiagramm für die ChatApp die beiden Methoden *senden* und *empfangen* von *SecretChat* und *SecureChat* überschrieben, die Parameter bleiben allerdings unverändert. Das ist in UML legitim. Um uns beispielhafte Szenarien anzusehen, wie es sich mit veränderten Parametern verhält, ist in Abbildung 3.2 ein Ausschnitt aus der ChatApp

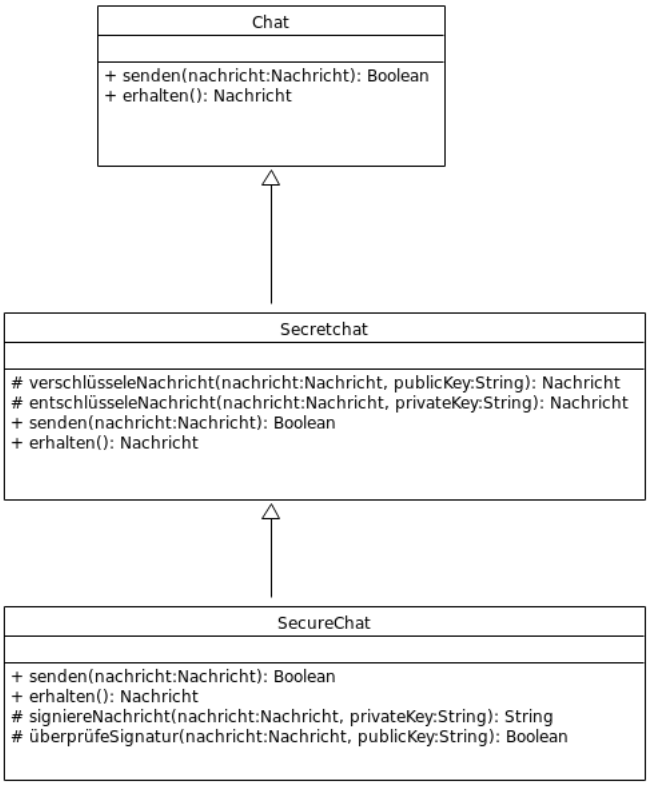

Abbildung 3.1: Ausschnitt aus der ChatApp. Die Klasse Chat wird von *SecretChat* und diese wiederum *SecureChat* geerbt.

dargestellt, bei dem einige kleinere Änderungen vorgenommen wurden. Neben den drei Klassen *Chat*, *SecretChat* und *SecureChat* sowie zusätzlich zur Klasse *Nachricht* sind die beiden Klassen *VerschlüsselteNachricht* und *SichereNachricht* hinzugefügt. Die Klasse *VerschlüsselteNachricht* ist eine auf irgendeine Art verschlüsselte Nachricht und wird von der Klasse *SecureChat* verwendet und die Klasse *SichereNachricht* ist eine auf irgendeine Weise signierte Nachricht, die von der Klasse *SecureChat* verwendet wird. Diese beiden Klassen erben direkt von der Klasse *Nachricht* und sind somit Unterklassen von diesen. Außerdem ist der Parameter *Nachricht* der Methoden *senden* und *empfangen* in den Klassen *SecretChat* und *SecureChat* nun jeweils mit den neuen Attributen *VerschlüsselteNachricht* und *SichereNachricht* ausgetauscht. Mit der neuen Definition von Subclassing hätte dies die folgende Bedeutungen:

- Das Überschreiben der Methode *senden* von der Klasse *Chat* seitens der Klasse *SecretChat* wäre legitim, da es sich bei dem überschriebenen Parameter *Nachricht* um eine Oberklasse des neuen Parameters *VerschlüsselteNachricht* handelt.

- Das erneute Überschreiben des Parameters *VerschlüsselteNachricht* der Klasse *SecretChat* seitens ihrer Unterklasse *SecureChat* ist jedoch nicht zulässig. Hierbei handelt es sich bei dem überschriebenen Parameter *VerschlüsselteNachricht* um keine Unterklasse des neuen Parameters *SichereNachricht*. Da *SichereNachricht* kein Subtyp von *VerschlüsselteNachricht* ist und es sich auch nicht um den selben Typen handelt, ist diese Art der Überschreibung seit UML 2.0 nicht erlaubt.
  Nimmt man, wie in Abbildung 3.3 dargestellt, eine kleine Änderung an der

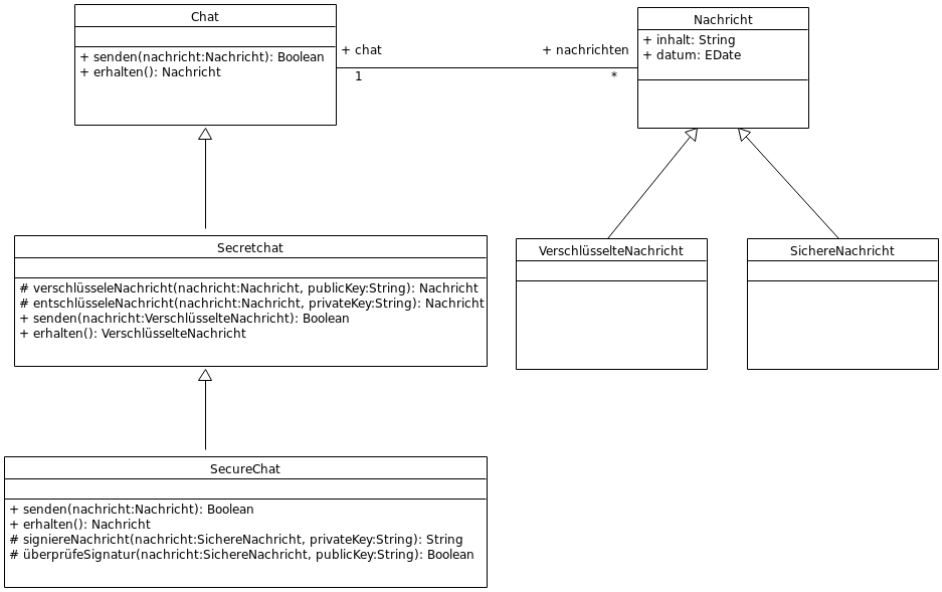

Abbildung 3.2: Die Methode *senden* von der Klasse *SecretChat* ersetzt den Parameter der Methode mit einem Subtypen.

Vererbungshierarchie zwischen den Klassen *Nachricht*, *VerschlüsselteNachricht* und *SichereNachricht* vor, so dass *SichereNachricht* nicht eine direkte Unterklasse von *Nachricht* ist, sondern von *VerschlüsselteNachricht* erbt, so ist das Überschreiben der Methode *senden* und *erhalten* legitim. Der Grund hierfür ist, dass der Parameter *nachricht* durch einen Subtypen in der Unterklasse *SichereNachricht* ersetzt worden ist.

Analog zu Überschreibungen von Methoden gelten auch die selben Regeln für Überschreibungen von Attributen und Assoziationen. Attributtypen und Assoziationstypen dürfen ebenfalls nur mit Subtypen überschrieben werden [BG04]. Außerdem gilt auch für Rückgabetypen, dass diese nur mit dem gleichen Typen oder einem Subtypen in überschreibenden Methoden ersetzt werden dürfen.

3.3 Unterschiede zwischen Java und UML

Da die objektorientierte Programmiersprache Java von vielen genutzt und als Referenz genommen wird, werden in diesem Unterkapitel die Unterschiede in Bezug auf Überschreibungen in UML und Java aufgezeigt. In Java ist das Überschreiben von Methoden anders definiert [JaD]. Dort müssen die Parameter einer überschreibenden Methode in einer Unterklasse vom selben Typ sein wie die Parameter der überschriebenen Methode in der Oberklasse. Anders als in UML ist es also nicht möglich, dass ein Subtyp eines Parameters in der überschreibenden Methode in der Unterklasse verwendet wird. Für den Rückgabetyp in Java gilt ebenso, dass für void (kein Rückgabewert) oder einen primitiven Typen exakt der gleiche Typ in den Unterklassen verwendet werden muss. Für Referenzrückgabetypen gilt in Java, dass diese von dem gleichen Typen oder einem Subtypen in der überschreibenden Methode ersetzt werden können. Dies ist in UML ebenso der Fall.

In dieser Arbeit wird mit UML und ihrer Definitionen gearbeitet und nicht mit Java oder einer anderen spezifischen Programmiersprache.

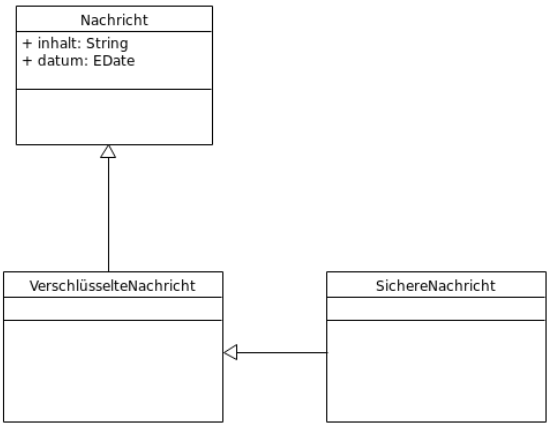

Abbildung 3.3: Die Vererbungshierarchie zwischen den drei Klassen *Nachricht*, *VerschlüsselteNachricht* und *SichereNachricht*.

3.4 UMLsec

Für die Benutzer der ChatApp ist es wichtig, dass ihre privat ausgetauschten Nachrichten vor Zugriffen von außen geschützt sind. Zum einen wird damit die Vertraulichkeit der Nachrichten sichergestellt und zum anderen die Integrität ihrer Inhalte. Für die Vetraulichkeit ist die Klasse *SecretChat* und, zusätzlich für die Integrität, die Klasse *SecureChat* erstellt. Als nächstes wird in diesen Klassen Sicherheitsanforderungen für die Nachrichten definiert, die gelten müssen. Dafür wird UMLsec verwendet. UMLsec ist eine Erweiterung von UML, um Anforderungen an die Sicherheit zu formulieren [Jür05]. Zunächst werden die in dieser Arbeit benötigten Elemente von UMLsec eingeführt, um die Sicherheitseigenschaften der Applikation in einem Klassendiagramm zu modellieren.

UMLsec spezifiziert Sicherheitseigenschaften, indem es das Angreifer-Modell verwendet [Jür05]. In diesem Modell wird davon ausgegangen, dass ein Angreifer im System vorhanden ist und Nachrichten in diesem System versenden und verändern kann [DY06]. Für diese Arbeit wichtige Sicherheitseigenschaften von UMLsec sind *secrecy* und *integrity*. Secrecy bezieht sich auf die Vertraulichkeit von Daten. Eine Nachricht soll nur für Berechtigte zugänglich sein. Ein System stellt die Vertraulichkeit von Daten in UMLsec dann sicher, wenn das System niemals Daten sendet, falls es für einen mit dem System interagierenden Angreifer lesbar sein könnte [Jür05]. Die Sicherheitseigenschaft integrity bezieht sich auf die Integrität von Daten. Integrität bedeutet, dass Daten vor unbefugten Modifikationen geschützt sind. Falls während der Ausführung des Systems, einer Variable ein ungewünschter Wert zugewiesen werden sollte, so wird davon ausgegangen, dass ein Angreifer dies verursacht hat und die Integrität gilt als beschädigt [Jür05].

UMLsec nutzt die Standart UML Stereotypen zusammen mit *Tags*, um diese Sicherheitseigenschaften zu formulieren. Außerdem werden *constraints* verwendet, um festzustellen, ob die gestellten Anforderungen durch das Systemdesign durchgesetzt werden. Ein Stereotyp von UMLsec ist *critical*. Ein mit critical markiertes Element zeigt an, dass sie Daten beinhaltet, die in irgendeinerweise kritisch sind. Der dazugehörige Tag (secrecy, integrity) spezifiert, auf welche Art diese Daten kritisch sind. Diese können die zuvor erklärten Sicherheitseigenschaften secrecy oder integrity sein. Der Tag secrecy beinhaltet den Namen des Objekts, dessen Vertraulichkeit bewahrt werden soll. Der Tag integrity besteht aus einer Variablen, deren Integrität geschützt werden soll. Der Stereotyp critical besitzt über keine Einschränkungen. Ein weiterer Stereotyp ist die *Secure Dependency*. Sie stellt sicher, dass dass die call-Abhängigkeiten zwischen Objekten die Sicherheitsanforderungen der ausgetauschten Daten respektieren. Beispielsweise darf ein Element ohne den Stereotyp critical keine call-Abhängigkeit zu Elementen haben, die über ein critical-Stereotyp mit einem der Tags secrecy, integrity oder high verfügen [Jür05].

3.4.1 Sicherheitsdefinitionen für die ChatApp

In Abbildung 3.4 ist erneut das Modell der ChatApp abgebildet. Hier ist sie mit UMLsec modelliert und wurde um Sicherheitseigenschaften erweitert, die nun gelten

sollen. An den beiden Klassen *SecretChat* und *SecureChat* sind nun die Stereotypen *critical* hinzugefügt. Erstens besitzt die Klasse *SecretChat* nun über einen Stereotyp *critical* mit dem dazugehörigen Tag *secrecy* für die Methode *senden*. Die Methode *senden* ist also vertraulich und ist daher auch mit einem *secrecy-Tag* gekennzeichnet. Zweitens besitzt die Klasse *SecureChat* ebenfalls über den Stereotyp *critical*. Hier lautet der dazugehörige Tag *integrity*, der sich ebenfalls auf die Methode *senden* bezieht. Außerdem gehen vier *call-Pfeile* von der Klasse Benutzer aus. Damit kann die Klasse *Benutzer* die Methoden der Klassen *Chat*, *SecretChat* sowie *Gruppenchat* aufrufen.

Der *Secure Dependency Check* soll nun sicherstellen, dass die Integrität und Vertraulichkeit der Methode *senden* in den Klassen *SecretChat* und *SecureChat* sichergestellt ist. Es soll nicht an Klassen zugänglich sein, die nicht in ihrer Klassendefinition angeben, dass sie diese Sicherheitseigenschaften respektieren (dies ist der Fall, wenn die Klasse nicht über einen Stereotyp *critical* mit dem passenden Tag (*secrecy, integrity*) verfügt).

Als nächstes werden die vier call-Pfeile der Klasse *Benutzer* betrachtet und überprüft, ob diese Abhängigkeiten seitens des *Secure Dependency Checks* legitim sind oder definierte Sicherheitseigenschaften verletzen:

1. Die Klasse *Benutzer* ruft die Methoden der Klasse *Chat* auf. Die Klasse *Chat* verfügt über keinen Stereotyp. Dies bedeutet, dass es keine Attribute oder Methoden besitzt, die sicherheitskritisch sind. Damit ist der Aufruf zulässig.

2. Die Klasse *Benutzer* ruft die Methoden der Klasse *SecretChat* auf. Die Klasse *SecretChat* verfügt über den Stereotyp critical, welche Vertraulichkeit für die Methode *senden* definiert. Die Klasse *Benutzer* besitzt nicht über ein critical-Stereotyp, respektiert also nicht die Vertraulichkeit der Methode *senden*. Daher scheitert hier der Secure Dependency-Check.

3. Die Klasse *Benutzer* ruft die Methoden der Klasse *SecureChat* auf. Die Klasse *SecureChat* verfügt über den Stereotyp critical, welche Integrität für die Methode *senden* definiert. Die Klasse *Benutzer* besitzt nicht über ein integrity-Stereotyp, respektiert also nicht die Vertraulichkeit der Methode *senden*. Daher scheitert hier der Secure Dependency-Check.

4. Die Klasse *Benutzer* ruft die Methoden der Klasse *Gruppenchat* auf. Die Klasse *Gruppenchat* verfügt über keinen Stereotyp. Dies bedeutet, dass sie keine Attribute oder Methoden besitzt, die sicherheitskritisch sind. Damit ist der Aufruf zulässig.

Nun sind die ersten drei call-Aufrufe in UMLsec klar definiert und der Secure Dependency Check lässt den Aufruf im ersten Fall zu und scheitert beim zweiten. Der dritte Fall ist scheinbar analog zum Ersten. Allerdings muss hier in Betracht gezogen werden, dass die Klasse *Gruppenchat* eine Unterklasse der Klasse *SecretChat* und somit ihre Attribute und Methoden erbt. Damit ist die sicherheitskritische Methode *senden* in der Klasse *Gruppenchat* enthalten, die Klasse selbst jedoch nicht mit

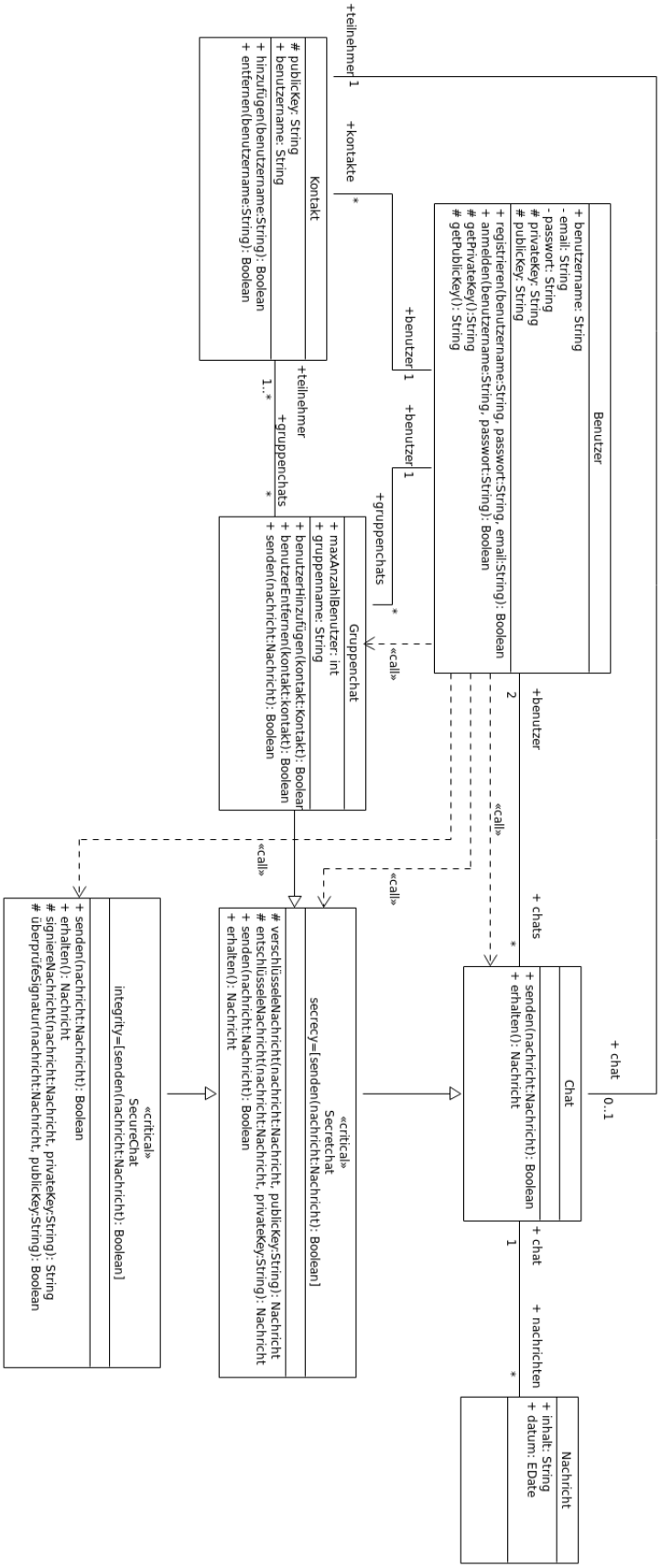

Abbildung 3.4: Die ChatApp modelliert mit UMLsec. Der Stereotyp critical zeigt an, dass eine Klasse sicherheitskritische Daten beinhaltet.

einem critical-Stereotyp markiert. Dadurch ist die sicherheitskritische Methode *senden* auch durch die call-Abhängigkeit in der Klasse *Benutzer* verfügbar. Auch dies hat eine Beschädigung der Sicherheitsanforderung Vertraulichkeit für die Methode *senden* zur Folge, dies wird jedoch nicht vom Secure Dependency Check erkannt.

Bei dem Secure Dependency Check handelt es sich um eine Form der Zugriffskontrolle. Daher wird im folgenden Kapitel das Konzept der Zugriffskontrollmodelle allgemein eingeführt und verschiedene Zugriffskontrollmodelle vorgestellt. Es wird erläutert, dass auch dort das Mechanismus der Vererbung vorhanden ist. Der dort angewendeten Lösungsansatz wird in ähnlicher Weise im Klassendiagramm übernommen werden und damit ein Modell aufgestellt, dass die Vererbung von Sicherheitseigenschaften erlaubt und weiterhin die Sicherheit des Systems gewährleistet. Denn aktuell abstrahiert UMLsec das Konzept der Vererbung. Es wird in späteren Kapiteln erläutert, dass es zu sicherheitskritischen Komplikationen dadurch kommen kann.

3.5 Zugriffskontrollmodelle

Zugriffskontrollmodelle regeln den Zugriff von Subjekten auf Objekte in einem System. Mit Subjekten bezeichnet man die Benutzer in einem System oder auch Programme (Prozesse), die nach dem Willen eines Benutzers agieren können [RSS94]. Subjekte führen Aktionen auf den Dateien in einem System aus. Meist handelt es sich dabei um eine der drei häufigsten Zugriffsarten *Lesen*, *Schreiben* oder *Ausführen*. Doch nicht immer dürfen diese Aktionen ausgeführt werden. Nicht jeder Benutzer soll über Lese-, Schreibe-, und Ausführrechte für alle Dateien in einem System verfügen. Die Datei, auf welche die Aktion ausgeführt werden soll, wird Objekt genannt. Subjekte können dabei auch selbst Objekte sein, schließlich könnte ein Subjekt auf eine ausführbare Datei zugreifen, das selbst wiederum auf andere Objekte zugreifen kann [RSS94]. Zugriffskontrolle wird meist durch eine Zugriffsmatrix modelliert. Eine Implementierung dieser Zugriffsmatrix wird häufig durch eine Zugriffskontroll-Liste vorgenommen [RSS94]. In den folgenden Unterkapiteln werden verschiedene Zugriffskontroll-Modelle vorgestellt. Zunächst wird die *Discretionary Access Control* betrachtet, welche auf der Basis der Identität der Subjekte den Zugriff auf Objekte kontrolliert. Anschließend wird der *Mandatory Access Control* erläutert, das Subjekte und Objekte klassifiziert und den Zugriff auf Basis dieser Klassifikation regelt. Danach werden die *Bell-LaPadula Richtlinien* vorgestellt, die beide zuvor genannten Zugriffskontrollmodelle umsetzt. Zuletzt wird der *Role-based Access Control* erläutert, das auch über das Konzept der Vererbung verfügt. Es wird in Erfahrung gebracht, dass mit Role-based Access Control die Bell-LaPadula Richtlinien simuliert werden können.

3.5.1 Discretionary Access Control

Discretionary Access Control (DAC) regelt den Zugriff von Subjekten zu Informationen auf der Basis der Identität des Subjekts. Ein Subjekt kann dabei ein Benutzer oder eine Benutzergruppe in einem System sein. Eine Benutzergruppe ist

dabei eine Gruppierung von mehreren Benutzern, welche die gleichen Rechte teilen. Die verschiedenen Rechte pro Objekt werden dabei vom Besitzer des jeweiligen Objekts vergeben und können auch von ihm wieder entzogen werden [Anh]. Sobald ein Benutzer eine Aktion auf einem Objekt ausführen möchte, wird überprüft, ob dieser Benutzer für diese Aktion berechtigt ist. Ist der Benutzer für diese Aktion berechtigt, so wird die Handlung genehmigt. Andernfalls wird dem Benutzer der Zugriff auf dieses Objekt verwehrt [RSS94].

Welche Rechte welcher Benutzer hat, sind in einer Zugriffsmatrix erkennbar. In den Zeilen sind die einzelnen Subjekte gelistet und in den Spalten die einzelnen Objekte. Damit kann man für jedes Subjekt in Kombination mit jedem Objekt ablesen, welche Rechte dem Subjekt zur Verfügung stehen. In Abbildung 3.5.1 ist eine einfache, beispielhafte Zugriffsmatrix dargestellt. Dort hat der Benutzer (Subjekt) Alice Lese- (r), Schreibe- (w) und Ausführrechte (e) für die Datei 1 (Objekt) und nur Lese- und Ausführrechte für Datei 2.

Subjekt	Datei 1	Datei 2
Alice	r, w, e	r, e
Bob	r, e	e
Charlie	r	

Tabelle 3.1: Eine einfache Zugriffskontrollmatrix mit den Subjekten Alice, Bob und Charlie sowie den Objekten Datei 1 und Datei 2 und ihre einzelnen Berechtigungen.

3.5.2 Mandatory Access Control

Mandatory Access Control (MAC) ist eine Form der Zugriffskontrolle, bei welcher der Zugriff von Subjekten auf Objekten auf der Basis einer Klassifikation von Subjekten und Objekten erfolgt [RSS94]. Jedem Subjekt und Objekt wird eine Sicherheitsstufe zugeordnet. Die Sicherheitsstufe des Subjekts beschreibt seine Vertrauenswürdigkeit, zuverlässig mit Informationen umzugehen. Je zuverlässiger das Subjekt, desto höher ist auch seine Sicherheitsstufe. Die Sicherheitsstufe eines Objekts beschreibt die Sensitivität seiner Inhalte. Je höher die Sensitivität, desto höher ist auch die Sicherheitsstufe. Im einfachsten Fall entsprechen die Sicherheitsstufen einer hierarchisch geordneten Menge [RSS94]. Eine Sicherheitsstufe dominiert alle Sicherheitsstufen, die niedriger sind sowie sich selbst. Im Hinblick auf die Vertraulichkeit erhält ein Subjekt Zugriff auf ein Objekt, falls entweder ein *Read down* (nach unten lesen) oder ein *Write up* (nach oben schreiben) gegeben ist. Read down bedeutet, dass die Sicherheitsstufe des Subjekts die Sicherheitsstufe des Objektes dominieren muss. Dann darf das Subjekt einen lesenden Zugriff auf das Objekt erhalten. Bei einem Write up muss die Sicherheitsstufe des Subjektes von der des Objektes dominiert werden. Nur dann erhält das Subjekt einen schreibenden Zugriff auf das Objekt. Diese beiden Prinzipien verhindern den Abfluß von Informationen von höheren zu niedrigeren Stufen [RSS94]. Die selben Prinzipien können sich auch auf die Integrität der Informationen beziehen. Statt eines Read downs und Write ups hat man dann die beiden analogen Richtlinien Read Up und Write down.

3.5.3 Bell-LaPadula Modell

Das Bell-LaPadula Modell ist eine Sicherheitsrichtlinie und setzt sowohl die DAC als auch die MAC durch. Es hat drei Eigenschaften. Die erste Eigenschaft ist die *-Eigenschaft. Sie besagt, dass ein Subjekt Schreibzugriff auf ein Objekt hat, wenn seine Sicherheitsstufe niedriger ist als die Sicherheitsstufe des Objektes [Bel05]. Die zweite Eigenschaft, die Simple Security-Eigenschaft, besagt, dass ein Subjekt Lesezugriff auf ein Objekt hat, wenn seine Sicherheitsstufe die Sicherheitsstufe des Objektes dominiert [Bel05]. Mit der *-Eigenschaft und der Simple Security-Eigenschaft wird die MAC durchgeführt. Die dritte Eigenschaft, die Discretionary Security-Eigenschaft, setzt die DAC durch. Mit der Nutzung einer Zugriffsmatrix wird zusätzlich definiert, welche Subjekte auf welche Objekte zugreifen dürfen. Ein Subjekt darf nur auf ein Objekt zugreifen, wenn alle drei Eigenschaften erfüllt sind [Lin06]. Durch die Verwendung zweier Zugriffskontrollen stellt das Bell-LaPadula Modell sowohl die Vertraulichkeitsanforderungen als auch die Flexibilität der Zugriffskontrolle sicher [Lin06].

In Abbildung 3.5 ist das Bell-LaPadula Modell bildhaft dargestellt. Dort ist zu sehen, dass ein Subjekt Leserechte auf ein Objekt hat, wenn seine Sicherheitsstufe die Sicherheitsstufe des Objekts dominiert. So kann beispielsweise Subjekt 1 Objekt 1 und Objekt 2 lesen, wohingegen Subjekt 2 nur Objekt 2 lesen darf, da Objekt 1 eine höhere Sicherheitsstufe hat als es selbst (No Read up). Ein Subjekt hat Schreibrechte auf ein Objekt, wenn seine Sicherheitsstufe niedriger ist als die Sicherheitsstufe des Objektes. So hat beispielsweise Subjekt 3 Schreibrechte auf Objekt 1 und Objekt 2. Subjekt 2 hingegen hat nur Schreibrecht auf Objekt 1, da das Sicherheitslevel von Objekt 2 niedriger ist als sein eigenes (No Write down).

Die Bell-LaPadula Richtlinien werden in Systemen eingesetzt, die sehr sicherheitskritisch sind und in denen mit Informationen sehr strikt umgegangen werden muss, um zu verhindern, dass diese nicht an Unbefugte gelangen. Diese Richtlinien sind für andere Systeme zu strikt. Dort wird gerne eine neuere Form der Zugriffskontrolle benutzt, die im nächsten Kapitel erörtert wird.

3.5.4 Role-based Access Control

Wie bereits in den letzten Kapiteln erwähnt, ist das Bell-LaPadula Modell nicht für alle Systeme geeignet. MAC werden häufig in militärischen Applikationen verwendet und ihre Richtlinien sind für betriebliche Anwendungssysteme meist zu strikt. DAC haben ihren Ursprung in kooperativen, doch autonomen Gebieten, wie der Forschung und sind für betriebliche Anwendungen meistens zu schwach [RSS94].

Eine neuere Form der Zugriffskontrolle, die sich in betrieblichen Anwendungssystemen besser eignet, ist Role-based Access Control (RBAC). Anstatt einem realen Benutzer direkt Berechtigungen zu vergeben, wie es in MAC und DAC der Fall ist, führt die RBAC das Konzept der Rolle ein. Eine Rolle ist eine Menge von Aktionen, die mit einer gewissen Arbeitstätigkeit zusammenhängen. In einem Unternehmen könnte eine Rolle also einen Jobtitel darstellen, dem ein Angestellter zugewiesen ist.

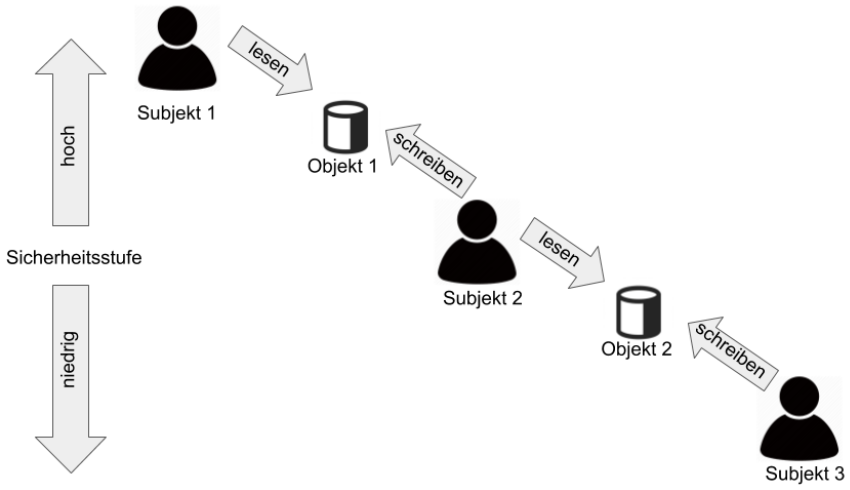

Abbildung 3.5: Subjekte haben Schreibzugriff auf Objekte mit einer höheren Sicherheitsstufe und Leserechte auf Objekte mit einer niedrigeren Sicherheitsstufe.

Statt dem Benutzer werden die Berechtigungen den Rollen zugewiesen. Möchte ein Benutzer b eine Aktion x ausführen, so muss x einer Rolle r zugewiesen sein, dessen Mitglied b ist.

Demnach gibt es in RBAC drei Entitäten: Benutzer, Rollen und Berechtigungen [USMDAS14]. Ein Benutzer ist wie in MAC und DAC ein Mensch oder ein Software-Agent und eine Berechtigung ebenso wie in MAC und DAC die Gestattung, eine bestimmte Aktion auszuführen. Da Benutzer Mitglieder einer Rolle sind und Berechtigungen an Rollen vergeben werden, ergeben sich zwei Relationen: Die Benutzervergabe-Relation (user-assignment) zwischen Benutzer und Rollen sowie die Berechtigungsvergabe-Relation (permission-assignment) zwischen Rollen und Berechtigungen [FCK95]. Die beiden Relationen Benutzervergabe und Berechtigungsvergabe haben eine Viele-zu-Viele-Beziehung [OSM00]. Ein Benutzer kann also einer beliebigen Anzahl von Rollen angehören und eine Rolle kann eine beliebige Anzahl von Benutzern als Mitglieder haben. Es ist auch möglich, dass ein Benutzer keiner Rolle angehört oder, dass eine Rolle keine Mitglieder hat. Analog kann eine Berechtigung an beliebig viele Rollen vergeben werden und eine Rolle beliebig viele Berechtigungen beinhalten. Die Aufteilung der Authorisationen in zwei Teile erleichtert außerdem das Sicherheitsmanagement [RSS94]. Ändert sich die Aufgabe eines Benutzers, so muss man lediglich seine Rolle entfernen und ihm eine neue Rolle zuweisen. Andernfalls müsste man alle Berechtigungen des Benutzers einzeln entfernen und ihm neue zuweisen. Eine vereinfachte Darstellung des RBAC ist in Abbildung 3.6 dargestellt. Dort sind die drei Entitäten Benutzer, Rolle und Berechtigungen zu sehen. Zwischen diesen sind als Pfeile die beiden Relationen Benutzervergabe und Berechtigungsvergabe gekennzeichnet. Außerdem geht ein Pfeil von Rolle aus und endet in eben diesem mit der Aufschrift Rollenhierarchie. Das Konzept der Rollen-

hierarchie wird folgend noch erklärt.

Da es durch die Viele-zu-Viele-Beziehung möglich ist, dass ein einzelner Benutzer mehreren Rollen angehört und mehrere Berechtigung durch diese Mitgliedschaften erhält, ist es denkbar, dass ein Fall auftritt, welche dem Benutzer die Möglichkeit eines betrügerischen Aktes erlaubt. Beispielsweise sollte ein Benutzer in einem Unternehmen nicht die Rollen Einkaufsleiter und Buchhalter verfügen [SCFY96], da dabei ein solches Szenario möglich ist. Um solche Arten von Missbrauch zu verhindern, gibt es in RBAC disjunkte Rollen [OSM00]. Rollen, die zusammen einem Benutzer zu viele Berechtigungen gestatten würden, sodass ein Missbrauch möglich wäre, würden als disjunkt definiert werden und der selbe Benutzer dürfte nicht Mitglied von Rollen sein, wenn er bereits Mitglied in einer Rolle ist, die zu dieser disjunkt sind [SCFY96]. Auf diese Art wird in RBAC eine Funktionstrennung durchgesetzt.

Es ist vorstellbar, dass in einem Unternehmen eine höherrangige Rolle über alle Berechtigungen einer untergeordneten Rolle verfügen soll, aber auch noch über zusätzliche Berechtigungen. Beispielsweise kann in einer IT-Abteilung die Rolle des Projektmanagers alle Berechtigungen haben wie ein Projektmitglied, aber darüber hinaus auch über Berechtigungen, die nur ihm vorenthalten und einem normalen Projektmitglied nicht zugänglich sind. Um zu vermeiden, Berechtigungen mehrfach an Rollen zu vergeben, verfügt die RBAC über das Konzept der Rollenhierarchie. Sie definiert, dass höherrangige Rollen Berechtigungen von Rollen niedrigeren Ranges erben [SCFY96]. Diese Vererbung ist transitiv. Das heißt, dass eine höherrangige Rolle auch Berechtigungen von einer Rolle erbt, die diese von einer wiederum niedrigeren Rolle geerbt hat. In Abbildung 3.7 erben die Rollen Softwaretester und Softwareentwickler die Berechtigungen der niederrangigen Rolle Projektmitglied. Die höchstrangige Rolle des Projektmanagers erbt die Berechtigungen von den Rollen Softwaretester und Softwareentwickler und damit auch die transitiv die Berechtigungen der Rolle des Projektmitglieds. Die Rolle Projektmitglied hingegen hat nur

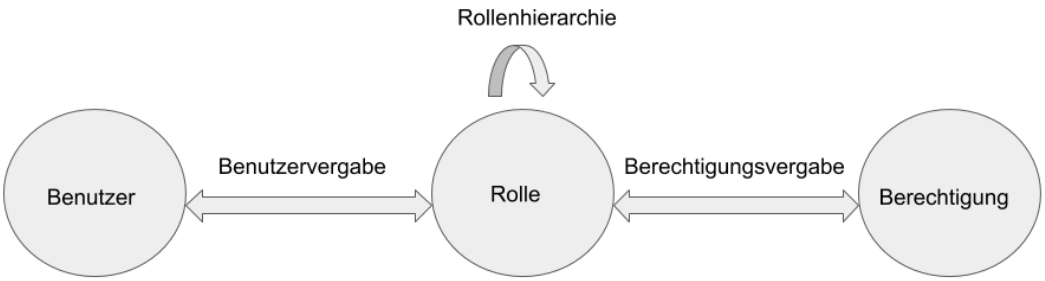

Abbildung 3.6: Eine vereinfachte Darstellung des RBAC

die Berechtigung, die ihm explizit zugewiesen wurden.

Die Rollenhierarchie ist für uns interessant, da sie die Vererbung von Rollen und damit auch Berechtigungen erlaubt. Ähnlich wie die Vererbung von Rollen in RBAC möchten wir in unserem Klassendiagram die Vererbung von Sicherheitseigenschaften ermöglichen. RBAC legt keine eigene Sicherheitsrichtlinie fest, kann aber bekannte Sicherheitsrichtlinien wie DAC und MAC durchsetzen. Im nächsten Kapitel werden wir aufzeigen, dass es auch möglich ist, das Bell-LaPadula Modell mit RBAC zu simulieren [FCK95].

3.5.5 Simulation der Bell-LaPadula Richtlinien

Neben den zuvor genannten Vorteilen, bringt RBAC auch einige Nachteile mit sich. In der Rollenhierarchie hat eine höherrangige Rolle Zugriff auf alle Berechtigungen, die an niedrigere Rollen zugewiesen wurden. Dies kann in vielen Organisationen, in denen eine höherrangige Rolle nicht zwingend genügend Kentnisse oder Erfahrungen in den Bereichen von einigen niedrigeren Rollen hat, unangebracht sein [GB98]. Weiterhin ist es nicht möglich, eine Funktionstrennung zwischen zwei Rollen einzuführen, die eine gemeinsame höhere Rolle haben [Kuh97]. Auch ist es nicht denkbar, eine multilevel sichere Systeme mit RBAC zu implementieren, da alle Vererbungen von Berechtigungen innerhalb der Rollenhierarchie aufwärts sind, wohingegen im Bell-LaPadula Schreibberechtigungen sich abwärts orientieren [BL73].

Eine Möglichkeit, diese Nachteile zu korrigieren, ist es, mit RBAC die Bell-LaPadula Richtlinien zu simulieren. Dies ist möglich, da RBAC richtlinienneutral ist und andere Richtlinien nachahmen kann [ZC08]. In [Cra03] wird ein anderes Modell verwendet, um diese Nachteile anzusprechen und zu beseitigen. Es erlaubt

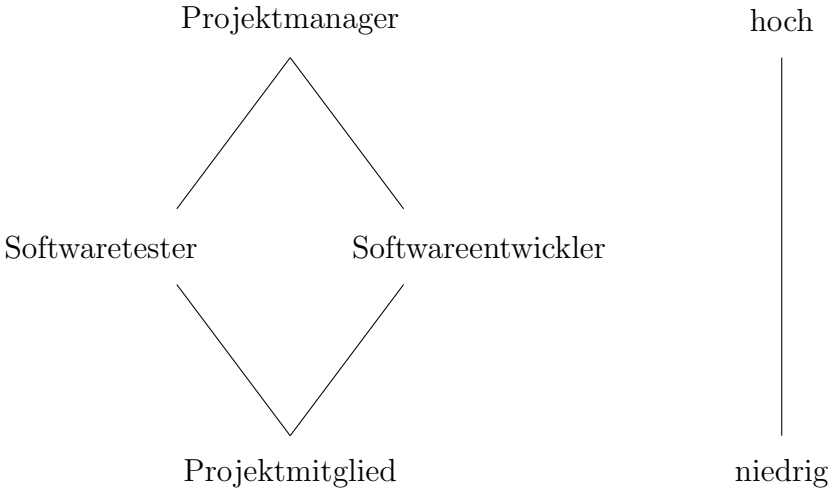

Abbildung 3.7: Beispiel einer Rollenhierarchie

Berechtigungen innerhalb der Rollenhierarchie zusammenzufassen. Rollen werden dort entweder von höherrangigen Rollen, niedrigrangigeren Rollen oder keinen Rollen geerbt. Weiterhin wird eine explizite Verbindung zwischen dem Sicherheitslevel eines Subjekts und der Benutzervergabe-Relation eingeführt und die *-Eigenschaft als eine Form der Funktionstrennung in RBAC angesehen. Eine andere Möglichkeit die Bell-LaPadula Richtlinien mithilfe von RBAC nachzuahmen, wurde in [ZC08] vorgestellt. Dort wird ein Algorithmus eingeführt, der in acht Schritten zunächst die MAC Richtlinien in RBAC Richtlinien überträgt und anschließend die DAC Richtlinien durchsetzt.

Da RBAC ähnlich wie unser Klassendiagramm das Konzept der Vererbung besitzt und die wie in diesem Kapitel besprochen die Bell-LaPadula Richtlinien nachahmen kann, ist es im folgenden Kapitel unser Ziel, ebenfalls die Bell-LaPadula Richtlinien in unserem Klassendiagramm durchzusetzen.

4 Vererbung von Sicherheitseigenschaften

In Kapitel 3.5 wurde das Bell-LaPadula Modell mit seinen drei Richtlinien vorgestellt:

1. Die Simple Security-Eigenschaft, welches ein *Read up* verhindert.
2. Die *-Eigenschaft, welches ein *Write down* verhindert.
3. Die Discretionary Security-Eigenschaft, welches zusätzlich die DAC durchsetzt.

Das Bell-LaPadula Modell ist für viele kommerzielle Systeme zu strikt. Daher wird oft in kommerziellen Systemen RBAC verwendet, welches das Konzept der Rollen einführt. Benutzer werden Rollen zugeordnet und den Rollen werden wiederum Berechtigungen verteilt. Ob ein Benutzer für eine gewisse Aktion berechtigt ist, entscheidet sich somit durch seine Zugehörigkeit zu Rollen. Außerdem existiert eine Rollenhierarchie, bei dem Rollen Berechtigungen von anderen Rollen erben. Wie in Kapitel 3.5.5 aufgezeigt wurde, ist es zudem möglich, die Bell-LaPadula Richtlinien zu simulieren. Es gilt festzuhalten, dass 1., RBAC die Bell-LaPadula Richtlinien durchsetzen kann, und 2., dass es über das Konzept der Vererbung verfügt. In diesem Kapitel wird zunächst aufgezeigt, welche Folgen es für in UMLsec definierte Sicherheitsanforderungen hat, wenn das Konzept der Vererbung im System vorhanden ist. Anschließend wird erläutert, wie die Bell-LaPadula Richtlinien im Klassendiagramm nachgeahmt werden können. Dabei wird zunächst eine grobe Idee skizziert, welche die Gemeinsamkeiten zwischen RBAC und dem Klassendiagramm aufzeigt und Lösungen in RBAC auf das Klassendiagramm überträgt. Anschließend wird ein formales Modell entworfen, das diese Lösungsideen generell umsetzt.

4.1 Auswirkung von Vererbung auf Sicherheitseigenschaften

In Abbildung 4.1 ist ein Abschnitt der ChatApp abgebildet. Die Klasse *Benutzer* hat eine call-Abhängigkeit auf die Klasse *Gruppenchat*. Die Klasse *Gruppenchat* wiederum, ist eine Unterklasse ihrer Oberklasse *SecreChat*. Die Klasse *SecretChat* ist mit einem critical-Stereotyp gekennzeichnet, d.h, sie besitzt sicherheitskritische Daten. Für die Methode *senden* soll die Vertraulichkeit gelten, da sie mit dem Tag *secrecy* markiert ist. Allerdings erbt die Klasse *Gruppenchat* diese Methode (sie überschreibt sie sogar) zusammen mit den anderen Methoden, die in *SecretChat* vorhanden sind, ohne dabei selbst über ein critical-Stereotyp zu verfügen. Dies ist der Fall, da in UMLsec das Konzept der Vererbung bislang abstrahiert wird. Doch dies hat Folgen, die über die Klasse *Gruppenchat* hinausgehen. So hat auch die Klasse *Benutzer* über ihre call-Abhängigkeit Zugriff auf Attribute und Methoden von *Gruppenchat*

und somit auch auf die sicherheitskritische Methode *senden*. Dabei besitzt *Benutzer* ebenso wenig über ein critical-Stereotyp mit dem passenden secrecy-Tag und respektiert damit ebenfalls nicht die Vertraulichkeit von *senden*. Damit ist die Vertraulichkeit von *senden* beschädigt. Allerdings schlägt der *Secure Dependency Check* dennoch nicht fehl. Sie betrachtet bei der call-Abhängigkeit nur den Client *Benutzer* und den Supplier *Gruppenchat*. Für den *Secure Dependency Check* besitzen beide Klassen über keine Sicherheitseigenschaften und die Klasse *Benutzer* ist erlaubt, Methoden und Attribute von Gruppenchat aufzurufen. Dass die Klasse *Gruppenchat* von ihrer Oberklasse *SecretChat* die sicherheitskritische Methode *senden* erbt, ist außerhalb der Reichweite des Checks, da *Gruppenchat* für diese sicherheitskritische Methode keine Sicherheitseigenschaften definiert.

Dieser Fall zeigt auf, dass ein Konzept notwendig ist, wenn man ein System in UMLsec modelliert hat und darin auch Vererbungen zwischen Klassen vorhanden sind. Ansonsten sind solche Verstöße gegen definierte Sicherheitsanforderungen möglich.

Um das Konzept von Vererbungen in UMLsec miteinzubeziehen und die Sicherheit von Daten in den modellierten Diagrammen zu bewahren, wird in dieser Arbeit die Vererbung von Sicherheitseigenschaften erlaubt. Um dies zu erreichen, wird im nächsten Unterkapitel untersucht, welche Verbindungen zwischen den Elementen im Klassendiagramm und den Eigenschaften des Bell-LaPadula Modells sowie der Rollenhierarchie aus RBAC bestehen. Diese werden dann in ähnlicher Form im Klassendiagramm angewendet.

4.2 Simulation der Bell-LaPadula Richtlinien im Klassendiagramm

In Kapitel 3.5.4 wurde in Erfahrung gebracht, dass RBAC eine Rollenhierarchie mit Vererbungsmechanismen verwendet, um Rollen und Berechtigungen zu aggregieren. Ebenso wurde in Kapitel 4.1 festgestellt, dass das Konzept der Vererbung Sicherheitslücken verursachen kann, wenn in einem UMLsec Klassendiagramm Ober-

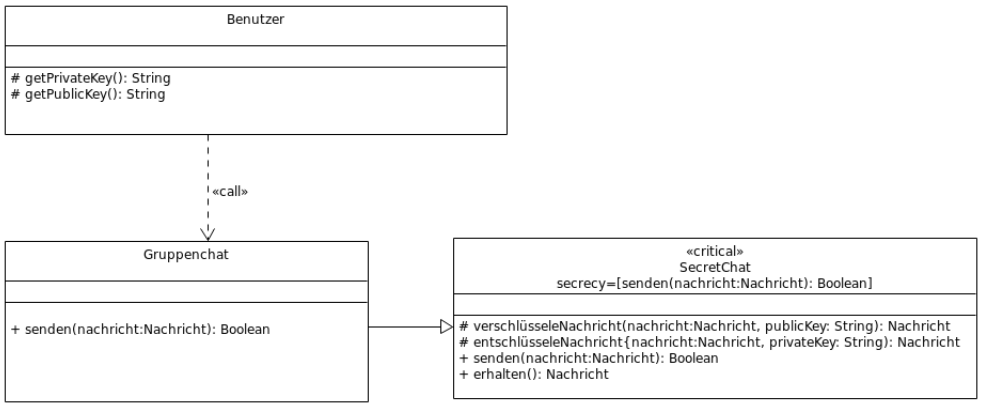

Abbildung 4.1: Die Klasse Benutzer hat eine call-Abhängigkeit zur Klasse Gruppenchat, welche die Methode *senden* von ihrer Superklasse *SecretChat* erbt, die dort als vertraulich gekennzeichnet ist.

klassen Sicherheitseigenschaften besitzen. Nun wird ein ähnlicher Lösungsansatz wie bei RBAC versucht, um diese sicherheitskritischen Fragen zu beantworten und zu lösen. Zunächst wird erläutert, wie verschiedene Szenarien im Klassendiagramm behandelt werden müssen in Bezug zu Sicherheitsrichtlinien des Bell-LaPadula Modells.

4.2.1 Secrecy als Simple Security-Eigenschaft

Der Tag *secrecy* bedeutet, dass die Vertaulichkeit der zugehörigen Variable geschützt werden muss. Dies ist laut dem Secure Dependency-Check der Fall, wenn es nur von Klassen zugegriffen wird, die ebenfalls über dieses Tag verfügen und damit ihre Vertraulichkeit bewahren. Dies ist ähnlich zu der Simple Security-Eigenschaft des Bell-LaPadula Modells. Dort wird vorausgesetzt, dass ein Subjekt auf ein Objekt nur dann zugreifen kann, wenn seine Sicherheitsstufe nicht niedriger ist als die Sicherheitsstufe des Objekts. Als Beispiel wird ein Abschnitt des Klassendiagramms für die ChatApp verwendet wie sie in Abbildung 4.2 zu sehen ist. Sie beinhaltet die beiden Klassen *Benutzer* und *SecretChat*. Außerdem ist noch die Klasse *VertraulicherBenutzer* hinzugefügt. Dieser verfügt über die selben Attribute und Methoden wie die Klasse *Benutzer*, hat aber zusätzlich den Stereotypen *critical* mit dem Tag *secrecy* für die Methode *senden*. Der Tag *secrecy* der Klasse *SecretChat* wird nun als Sicherheitsstufe dieser Klasse definiert. Jede Klasse, die ebenfalls über einen Tag *secrecy* mit der gleichen Methode verfügt, wird als eine Klasse mit einer nicht niedrigeren Sicherheitsstufe angesehen. Diese dürfen call-Abhängigkeiten zu dieser Klasse haben. Dies ist der Fall für die Klasse *VertraulicherBenutzer*. Andere Klassen, die nicht über diesen Tag mit der gleichen Methode verfügen, sind Klassen mit einer niedrigeren Sicherheitsstufe. Dies ist der Fall für die Klasse *Benutzer*. Diese Klasse hat eine niedrigere Sicherheitsstufe als *SecretChat* und dies würde ein *Read up* bedeuten, was die Simple Security-Eigenschaft verletzen würde. Daher darf diese Klasse keine call-Abhängigkeit zu *SecretChat* haben.

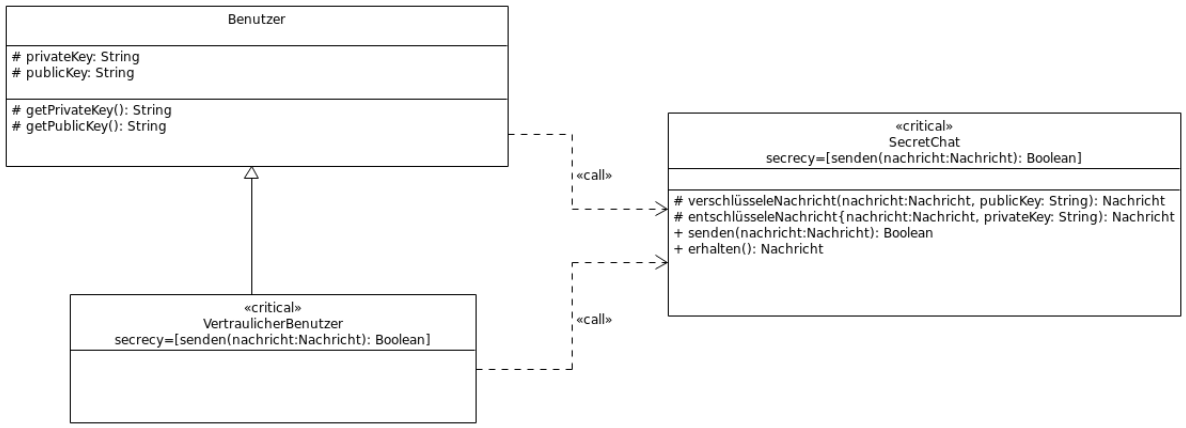

Abbildung 4.2: Die beiden Klassen *Benutzer* und *VertraulicherBenutzer* haben eine call-Abhängigkeit zur Klasse *SecretChat*.

4.2.2 Integrity als *-Eigenschaft

Ein ähnlicher Ansatz wird in Bezug auf den Tag *integrity* verwendet. Dieser besagt, dass die Integrität der Variable geschützt werden muss. Sie ist geschützt, wenn sie ausschließlich von Klassen verändert werden kann, die ebenfalls über einen *integrity* Tag verfügen und somit garantieren, dass sie nur zulässige Änderungen daran vornehmen. Wir möchten das Beispiel nun etwas abändern und den Tag *secrecy* mit einem *integrity*-Tag in der Klasse *VertraulicherBenutzer* austauschen und statt der Klasse *SecretChat* call-Abhängigkeiten auf die Klasse *SecureChat* haben, die den Stereotypen *critical* mit dem Tag *integrity* für die Methode *senden* hat, wie es in Abbildung 4.3 abgebildet ist. Der Tag *integrity* wird nun mit der *-Eigenschaft aus dem Bell-LaPadula Modell verglichen und erneut als die Sicherheitsstufe für diese Klasse interpretiert. Es wird festgelegt, dass alle Klassen, die über kein *integrity*-Tag mit der gleichen Methode verfügen, ein höhere Sicherheitslevel haben als Klassen, die ein solches besitzen. Dies ist der Fall für die Klasse *Benutzer*. Diese Klasse hat ein höheres Sicherheitslevel als die Klasse *SecureChat* und darf keine call-Abhängigkeit zu dieser haben, da dies ein *Write down* darstellen würde. Klassen, die über ein solches *integrity* Tag für die gleiche Methode verfügen und somit ein nicht höheres Sicherheitslevel haben, dürfen call-Abhängigkeiten zu *SecureChat* haben. Dies ist der Fall für die Klasse *VertraulicherBenutzer*.

4.2.3 Vererbung von Sicherheitseigenschaften und Rollenhierarchie

Die nächste Verbindung ist zwischen der Rollenhierarchie des RBAC und der Vererbung im UML Klassendiagramm. In RBAC erben höherrangige Rollen alle niedrigeren Rollen sowie ihre Berechtigungen und sind höher in der Rollenhierarchie angesiedelt. Als man die Bell-LaPadula Richtlinien mittels RBAC nachgeahmt hat, war ein Ansatz, die Menge an zugewiesenen Rollen an einen Benutzer als Sicherheitslabel für diesen Benutzer festzulegen. In Abbildung 4.4 ist erneut ein Abschnitt aus der ChatApp dargestellt. In dieser sind die drei Klassen *Chat*, *SecretChat* und *SecureChat* zu sehen. *Chat* ist die Oberklasse, von ihr erbt ihre Unterklasse *SecretChat*

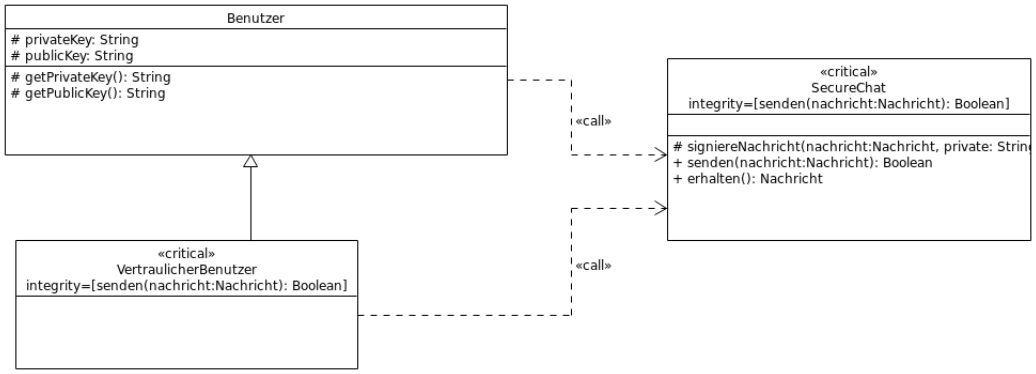

Abbildung 4.3: Die beiden Klassen *Benutzer* und *VertraulicherBenutzer* haben eine call-Abhängigkeit zur Klasse *SecureChat*.

ihre Attribute und Methoden. *SecretChat* ist wiederum die Oberklasse von *SecureChat* und *SecureChat* erbt ebenso die Attribute und Methoden von *SecretChat* und darüber hinaus, wegen der Transitivität, auch die Attribute und Methoden der Klasse *Chat*. Ähnlich zur Rollenhierarchie in RBAC wäre in unserer neuen Hierarchie die Klasse *SecureChat* ganz oben, gefolgt von *SecretChat* und zuletzt ganz unten *Chat*. Wenn wir nun die Menge an Tags als ein Sicherheitslevel für diese Klasse definieren, stellen wir fest, dass eine erbende Klasse entweder die gleiche Sicherheitsstufe hat wie ihre Oberklasse (wenn sie selbst keine zusätzlichen Sicherheitseigenschaften definiert) oder eine höhere Sicherheitsstufe (wenn Sie selbst noch zusätzliche Sicherheitseigenschaften definiert). Das heißt, die Klasse *SecureChat* hat die höchste Sicherheitsstufe. Sie verfügt über das Sicherheitslabel *secrecy* für die Methode *senden*, die sie von *SecretChat* erbt, hat darüber hinaus aber auch das Sicherheitslabel *integrity* für die Methode *senden*. *SecretChat* hat nur das Sicherheitslabel *secrecy* für die Methode *senden* und ist damit niedriger in der Hierarchie als *SecureChat*. *Chat* verfügt über gar keine Sicherheitslabel und ist somit ganz unten in der Hierarchie.

Mit dieser Erkenntnis wird deutlich, was passieren muss, wenn in Abbildung 4.5 die Klasse *Benutzer* eine call-Abhängigkeit zur Klasse *Gruppenchat* hat. *Gruppenchat* erbt von der Klasse *SecretChat* das Sicherheitslabel secrecy und hat damit eine höhere Sicherheitsstufe als *Benutzer*. Dies würde ein *Read up* zur Folge haben und

Abbildung 4.4: Die Vererbungshierarchie zwischen den Klassen *Chat*, *SecretChat* und *SecureChat*.

die Simple Security-Eigenschaft verletzen, weswegen der *Secure Dependency Check* hier fehlschlagen muss.

4.3 Formales Modell

Das letzte Kapitel zeigt auf, wie man die Bell-LaPadula Richtlinien im Klassendiagramm simulieren kann. Doch handelt es sich dabei nicht um eine formale Definition, sondern lediglich um eine Skizze einer Lösungsidee, bei dem Elemente aus dem Bell-LaPadula Modell mit den Elementen im Klassendiagramm in Verbindung gebracht und die Bell-LaPadula Richtlinien in das Klassendiagramm übertragen werden. Dabei wird für den Tag *secrecy* die Simple Security-Eigenschaft verwendet. Die Simple Security-Eigenschaft besagt, dass ein Subjekt ein Objekt nur dann lesen darf, wenn seine Sicherheitsstufe die Sicherheitsstufe des Objekts dominiert, d.h, kein *Read up* vorliegt. Analog wird für den *integrity* Tag die *-Eigenschaft benutzt. Diese besagt, dass ein Subjekt ein Objekt nur dann lesen darf, wenn seine Sicherheitsstufe von der Sicherheitsstufe des Objekts dominiert wird, d.h, kein *Write down* vorliegt. Zuletzt wird die Rollenhierarchie und ihre Agglomeration von Rollen als Idee dafür verwendet, wie man Sicherheitseigenschaften im Klassendiagramm vererben kann. Nun wird für diese Lösungsideen ein formales Modell entworfen. In Kapitel 4.3.1 wird eine allgemeine Definition für die Simple Security-Eigenschaft sowie die *-Eigenschaft und in Kapitel 4.3.2 anschließend die allgemeine Definition für die Vererbung von Sicherheitseigenschaften formuliert.

4.3.1 Simple Security Eigenschaft und *-Eigenschaft

In Abbildung 4.6 ist der im Folgenden beschriebene Fall abgebildet. Eine Klasse A hat eine call-Abhängigkeit zu einer Klasse B. Klasse A und B sind beide jeweils mit einem critical-Stereotyp annotiert. Klasse B verfügt über eine Methode *m*. S(a) annotiert die Menge an Sicherheitseigenschaften für Klasse A und S(b) die Menge

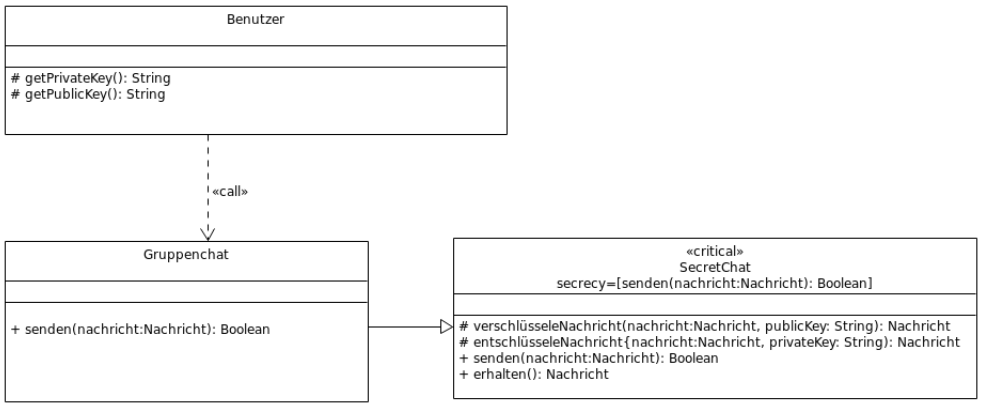

Abbildung 4.5: Die Klasse Benutzer hat eine call-Abhängigkeit zur Klasse *Gruppenchat*, welche die Sicherheitseigenschaft von ihrer Superklasse *SecretChat* erbt. Der Secure Dependency-Check schlägt fehl.

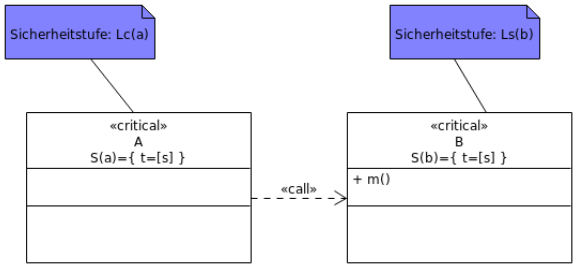

Abbildung 4.6: Klasse A hat eine call-Abhängigkeit auf Klasse B. Beide Klassen sind mit einem critical-Stereotypen annotiert und haben eine Menge an Sicherheitseigenschaften S(a) und S(b) sowie die Sicherheitsstufen Lc(a) und Ls(b).

an Sicherheitseigenschaften für Klasse B. Außerdem ist kommentiert, dass mit Lc(a) die Sicherheitsstufe von Klasse A und mit Ls(b) die Sicherheitsstufe von Klasse B bezeichnet wird. Die Menge an Sicherheitseigenschaften und die Sicherheitsstufen werden in den folgenden Abschnitten definiert.

Sei $S(x) = \{(t_v, s_v), ..., (t_w, t_w)\}$ mit $v, w \in \mathbb{N}_0$ die Menge aller Sicherheitseigenschaften einer Klasse x, t ein Tag vom Stereotypen critical und s die dazugehörige Signatur. Für jede call-Abhängigkeit zwischen einer Klasse A, dem *Client*, und einer Klasse B, dem *Supplier*, gilt: Die Menge an Sicherheitseigenschaften von Client A entspricht $S(a) = \{(t_h, s_h), ..., (t_i, s_i)\}$ und die Sicherheitseigenschaft von Supplier B entsprechen $S(b) = \{(t_j, s_j), ..., (t_k, s_k)\}$, mit h,i,j,k $\in \mathbb{N}_0$. Ferner definieren wir mit $D(b) = \{d_n, ..., d_m\}$ mit $n, m \in \mathbb{N}_0$ die Menge an Definitionen vom Supplier, d.h. alle Attribute und Methode, welche die Supplier-Klasse definiert.

Im Abschnitt unserer ChatApp in Abbildung 4.5 bezogen, bedeutet dies, dass für die Menge an Sicherheitseigenschaften der Klasse *SecretChat* S(SecretChat) gilt: S(SecretChat) = (secrecy,[senden(nachricht:Nachricht): Boolean]) . Die anderen beiden Klassen *Benutzer* und *Gruppenchat* haben keine Menge an Sicherheitseigenschaften, da sie über keinen critical-Stereotyp verfügen.

Außerdem bezeichnen wir mit *Lc(a)* die Sicherheitsstufe (Security Level) eines Tag-Signatur-Paares des Clients A und die Sicherheitsstufe eines Tag-Signatur-Paares des Suppliers B mit *Ls(b)* (eine Klasse kann sowohl Client als auch Supplier sein). Ist ein Tag-Signatur-Paar vorhanden, so ist ihr Wert 1. Ist sie nicht vorhanden, so ist ihr Wert 0. Weiterhin sind nun die beiden Bell-LaPadula Richtlinien folgendermaßen definiert:

1. **Simple Security-Eigenschaft**: Diese Richtlinie bezieht sich auf Sicherheitseigenschaften mit dem Tag *secrecy*. Gilt für die Sicherheitsstufe Lc(a) des Clients A und für die Sicherheitsstufe Ls(b) des Suppliers B die Ordnung Lc(a) $<$ Ls(b) so liegt ein *Read up* vor und die *Simple-Security-Eigenschaft* ist beschädigt. Dann schlägt auch der Secure Dependency Check fehl.

2. ***-Eigenschaft**: Diese Richtlinie bezieht sich auf Sicherheitseigenschaften mit dem Tag *integrity*. Gilt für die Sicherheitsstufe Lc(a) des Clients A und für die Sicherheitsstufe Ls(b) des Suppliers B die Ordnung Lc(a) > Ls(b), so liegt ein *Write down* vor und die *-Eigenschaft ist beschädigt. Dann schlägt auch der Secure Dependency Check fehl.

Damit die Secure Dependency Eigenschaft nicht verletzt ist, muss bei einer call-Abhängigkeit von einem Client zu seinem Supplier für alle Sicherheitseigenschaften S(A) (des Clients) und S(B) (des Suppliers) mit den Tags t die folgenden Bedingungen erfüllt sein:

1. Wenn Tag t *secrecy* entspricht, muss gelten:

$$(secrecy, s1) \in S(A) \iff (secrecy, s1) \in S(B) \cap s1 \in D(B)$$

   D.h, verfügt der Supplier oder der Client über eine Sicherheitseigenschaft mit dem Tag secrecy und einer bestimmten Signatur, so muss auch die andere Klasse über eine Sicherheitseigenschaft secrecy mit der selben Signatur verfügen. Dabei gilt jedoch die Beschränkung für den Supplier, dass die Sicherheitseigenschaften auch in ihrem Klassenkörper definiert sein müssen. Sicherheitseigenschaften, die nicht von ihr definiert werden, müssen nicht in der Client-Klasse vorhanden sein.

2. Wenn Tag t *integrity* entspricht, muss gelten:

$$(integrity, s2) \in S(A) \iff (integrity, s2) \in S(B) \cap s2 \in D(B)$$

   D.h, verfügt der Supplier oder der Client über eine Sicherheitseigenschaft mit dem Tag integrity und einer bestimmten Signatur, so muss auch die andere Klasse über eine Sicherheitseigenschaft integrity mit der selben Signatur verfügen. Dabei gilt jedoch die Beschränkung für den Supplier, dass die Sicherheitseigenschaft auch in ihrem Klassenkörper definiert sein müssen. Sicherheitseigenschaften, die nicht von ihr definiert werden, müssen nicht in der Client-Klasse vorhanden sein.

Falls 1. für nur eine Sicherheitseigenschaft in S(A) oder S(B) nicht gilt, so folgt Lc(a)=0 und Ls(b)=1 und damit auch Lc(a) < Ls(b). Damit gilt die Simple Security-Eigenschaft als verletzt.

Analog gilt, wenn nur für eine Sicherheitseigenschaft in S(A) oder S(B) für 2. nicht erfüllt ist, so ist Lc(a)=1 und Lc(b)=0 und damit auch Lc(a) > Ls(b). Damit gilt die *-Eigenschaft als verletzt.

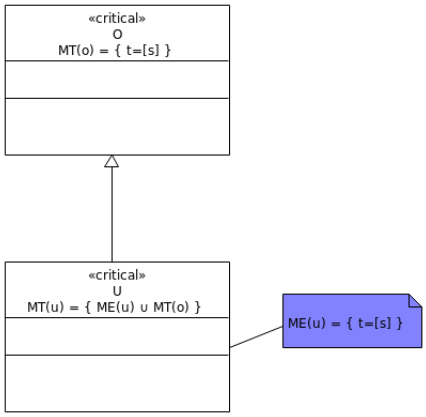

Abbildung 4.7: Klasse U erbt Sicherheitseigenschaft von Klasse O. MT(o) notiert die expliziten und geerbten Sicherheitseigenschaften von Klasse O und MT(u) die expliziten und geerbten Sicherheitseigenschaften von Klasse U.

4.3.2 Vererbung von Sicherheitseigenschaften

Die im vorherigen Unterkapitel vorgestellte Definition klärt, was passiert, wenn eine Klasse, eine call-Abhängigkeit zu einer anderen Klasse hat, die Sicherheitseigenschaften besitzt. Jedoch ist noch nicht definiert, was passiert, wenn z.B., wie in Abbildung 4.5 abgebildet, eine Klasse *Gruppenchat* von einer Klasse *SecretChat* mit Sicherheitseigenschaften erbt und eine Klasse *Benutzer* eine call-Abhängigkeit auf Klasse *Gruppenchat* hat. In UMLsec gilt aktuell in diesem Fall, dass *Gruppenchat* alle Attribute und Methoden von *SecretChat* erbt, jedoch nicht über ein critical-Stereotyp und über keinen zugehörigen Tag *secrecy* verfügt. *Benutzer* kann also sicherheitskritische Methoden und Attribute von Gruppenchat aufrufen ohne anzugeben, dass es diese respektiert. Um dies zu verhindern, definieren wir nun die Vererbung von Sicherheitseigenschaften. Die folgende Definition ist auch in Abbildung 4.7 abgebildet.

Wir schließen bei der nun folgenden Definition den Fall aus, dass eine Oberklasse ein Attribut oder eine Methode mit einer beliebigen Sicherheitseigenschaft besitzt und eine Unterklasse diese überschreibt. In Kapitel 4.4 wird auf diesen Sachverhalt im Speziellen eingegangen, erörtert und unser Modell damit erweitert.

Ist eine Klasse U eine Unterklasse einer Oberklasse O, so verfügt sie automatisch über alle Sicherheitseigenschaften von Klasse O. Es wird jetzt zwischen explizit angegebenen Sicherheitseigenschaften und geerbten Sicherheitseigenschaften unterschieden. Es werden die folgenden Definitionen verwendet:

1. $ME(x) = \{(t_v, s_v), ..., (t_w, s_w)\}$ mit $v, w \in \mathbb{N}_0$ steht für die Menge von explizit angegebenen Sicherheitseigenschaften einer Klasse X, wobei x die Klasse angibt, die diese Sicherheitseigenschaften hat.

2. $MV(x) = \{(t_v, s_v), ..., (t_w, s_w)\}$ mit $v, w \in \mathbb{N}_0$ steht für die Menge von geerbten Sicherheitseigenschaften einer Klasse X, wobei x die Klasse angibt, die diese Sicherheitseigenschaften hat.

3. $MT(x) = \{ME(x) \cup MV(x)\}$ steht für die Gesamtmenge an Sicherheitseigenschaften einer Klasse. Diese setzt sich aus einer Vereinigung aus explizit angegebenen und geerbten Sicherheitseigenschaften einer Klasse zusammen.

Hat Klasse U eine Oberklasse O und ist MT(o) die Gesamtmenge an Sicherheitseigenschaften von Klasse O, so gilt für die Gesamtmenge an Sicherheitseigenschaften MT(u), die Klasse U besitzt:

$$MT(u) = ME(u) \cup MT(o).$$

D.h, eine Unterklasse erbt nicht nur alle explizit angegeben Sicherheitseigenschaft seiner Oberklassen, sondern auch die von der Oberklassen geerbten Sicherheitseigenschaften. Damit gilt die Transitivität innerhalb der Vererbungshierarchie.

4.4 Überschreiben von sicherheitskritischen Membern

Im vorherigen Kapitel wurde ein formales Modell entworfen, um Sicherheitseigenschaften entlang von Vererbungshierarchien zu vererben. Eine Unterklasse erbt alle Sicherheitseigenschaften von seinen Oberklassen und ein Client mit einer call-Abhängigkeit zu der Unterklasse, dem Supplier, muss in seiner Klassendefinition angeben, dass es nicht nur die explizit angegebenen Sicherheitseigenschaften des Suppliers respektiert, sondern auch alle Sicherheitseigenschaften, die der Supplier erbt. Jedoch wird in diesem Modell bislang ausgeschlossen, dass eine Oberklasse ein Member mit einer beliebigen Sicherheitseigenschaft besitzt und eine Unterklasse diesen Member überschreibt. In diesem Kapitel wird auf dieses Sachverhalt eingegangen, eine Lösung entwickelt und das Modell so erweitert, dass auch das Überschreiben von sicherheitskritischen Membern miteinbezogen ist. Dabei werden zunächst nur Methoden betrachtet und im Anschluss darauf auch das Überschreiben von Attributen.

4.4.1 Auswirkung von Memberüberschreibungen auf Sicherheitseigenschaften

Wie in Kapitel 3.2 beschrieben, darf seit UML 2.0 bei der Überschreibung von Methoden die Parameter der überschreibenden Methode nur mit Subtypen von diesen ersetzt werden. In Abbildung 4.8 ist die leicht modifizierte Variante unserer ChatApp abgebildet. Die Klasse Nachricht hat zwei Unterklassen, *VerschlüsselteNachricht* und *SichereNachricht*, wobei *VerschlüsselteNachricht* wiederum eine Unterklasse von *SichereNachricht* ist. Außerdem sind die Parameter der Methoden *senden* der beiden Klassen *SecretChat* und *SecureChat* verändert: Nachricht hat in der Klasse *SecretChat* den Typen *VerschlüsselteNachricht* und in der Klasse *SecureChat* den Typen *SichereNachricht*. Gleiches gilt für die beiden Rückgabetypen der Methode *senden*. Die Methode *senden* hat in allen drei Klassen *Chat*, *SecretChat* und *SecureChat* eine andere Implementierung ihrer Funktionalität. Die Klasse *Chat*

versendet unverschlüsselte Nachrichten, die Klasse *SecretChat* verschlüsselte Nachrichten und die Klasse *SecureChat* verschlüsselte und signierte Nachrichten. Dies macht für diese drei Methoden andere Anforderungen an die Sicherheit erforderlich. Für die Methode *senden* in *Chat* können nicht die selben Sicherheitseigenschaften gelten wie für die überschreibende Methode *senden* der Klasse *SecretChat* und analog gilt dasselbe für die überschreibende Methode *senden* der Klasse *SecureChat*. Es wäre sinnvoll, dass für die Methode *senden* in den drei Klassen die folgenden Sicherheitseigenschaften gelten:

1. Keine Sicherheitseigenschaften in der Klasse *Chat*, da diese Klasse unverschlüsselte und unsignierte Nachrichten versendet und keine Sicherheitsanforderungen stellt.

2. Die Sicherheitseigenschaft *secrecy* in der Klasse *SecretChat*, da diese Klasse ihre Nachrichten verschlüsselt sendet, und diese für Unbefugte nicht lesbar sein sollen.

3. Die Sicherheitseigenschaften *secrecy* und *integrity*, da diese Klasse ihre Nachrichten verschlüsselt sendet, die für Unbefugte nicht lesbar sein sollen, und darüber hinaus die Nachrichten signiert, so dass eine nachträgliche Modifizierung dieser Nachrichten erkennbar sein sollen.

Das heißt, die Methode *senden* wird mit jeder Spezialisierung ihrer Klasse sicherheitskritischer.

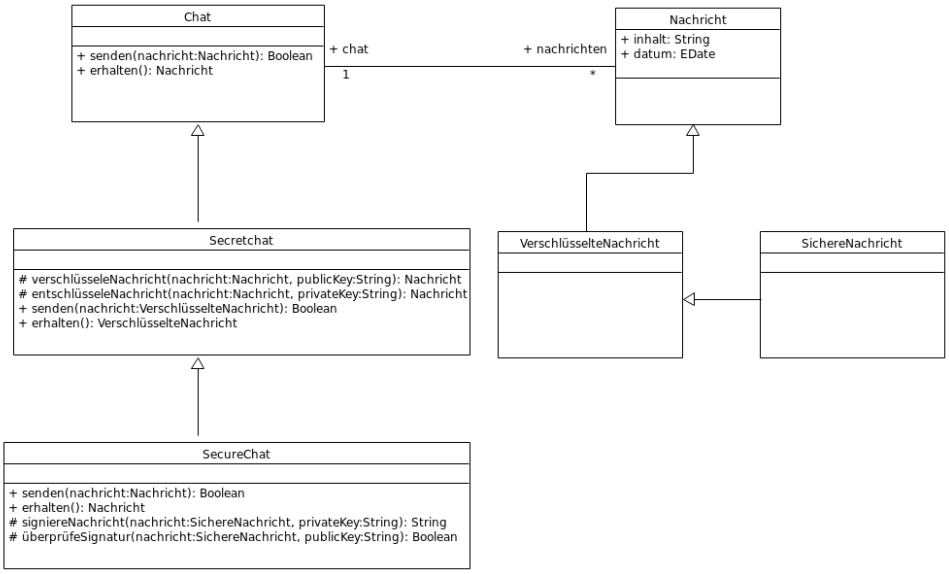

Abbildung 4.8: Die Methode *senden* von der Klasse *SecretChat* überschreibt den Parameter der Methode *senden* der Klasse *Chat* und wird von der Methode der Klasse *SecureChat* überschrieben.

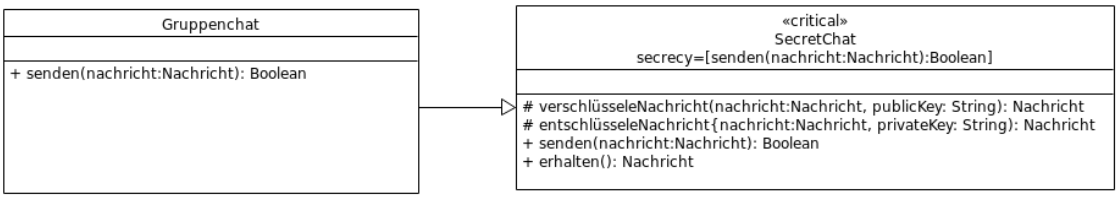

Abbildung 4.9: Die Klasse *Gruppenchat* erbt von ihrer Oberklasse *SecretChat*. Die Methode *senden* wird überschrieben.

In Abbildung 4.9 ist erneut ein Ausschnitt unserer ChatApp abgebildet. Dort erbt die Unterklasse *Gruppenchat* von ihrer Oberklasse *SecretChat*. Die Methode *senden* von *SecretChat* wird von der gleichnamigen Methode in *Gruppenchat* überschrieben. In *SecretChat* gilt diese Methode als vertraulich (sie hat den Stereotypen *secrecy*). In *Gruppenchat* ist diese Sicherheitseigenschaft aufgrund ihrer Sicherheitsdefinition nicht erforderlich. Sie ist weniger sicherheitskritisch als die überschriebene Methode und ist deshalb nicht vertraulich. Man könnte daher annehmen, dass es eine sinnvolle Lösung ist, Sicherheitseigenschaften bei Methodenüberschreibung nicht zu vererben. Allerdings muss man in Betracht ziehen, dass man die Unterklasse *Gruppenchat* im Kontext ihrer Oberklasse *SichererChat* verwenden kann, ohne dies zu wissen. Dann müssten alle Sicherheitseigenschaften gewahrt werden. Wenn aber einige Sicherheitseigenschaften nicht vererbt werden, so könnte es zu einer Missachtung von Sicherheitseigenschaften kommen, obwohl der Secure Dependency Check nicht fehlschlägt. In unserem Beispiel, könnte die Klasse *Benutzer* mit einer call-Abhängigkeit auf die Klasse *Gruppenchat* die Methode *senden* im Kontext ihrer Oberklasse *SecretChat* aufrufen, ohne selbst in ihrer Klassendefinition anzugeben, diese Sicherheitseigenschaft zu respektieren. Damit würde der Secure Dependency Check nicht fehlschlagen, aber unsere Sicherheitseigenschaft *secrecy* verletzt werden. Die Sicherheitseigenschaften von Oberklassen müssen deswegen an ihre Unterklassen vererbt werden.

Betrachtet man nun, wie in Abbildung 4.10 dargestellt, einen anderen Abschnitt der ChatApp, bei dem der call-Pfeil von der Klasse *Benutzer* auf die Oberklasse *Chat* zeigt, welche eine Unterklasse namens *SecretChat* hat. Die Klasse *Benutzer* hat keine Sicherheitseigenschaften. Die Klasse *Chat* verfügt ebenfalls über keine Sicherheitseigenschaften, hat aber die Methode *senden*. Diese Methode wird in ihrer *Unterklasse* überschrieben. Zusätzlich hat *SecretChat* die Sicherheitseigenschaft *secrecy* für die überschreibende Methode *senden*, da Nachrichten nur vertraulich (verschlüsselt) versendet werden. Da der Supplier *Chat* wie der Client *Benutzer* keine Sicherheitseigenschaft besitzt, ist der Aufruf ihrer Methoden in *Benutzer* legitim. Allerdings überschreibt die Unterklasse *SecretChat* diese Methode und fügt dieser die Sicherheitseigenschaft *secrecy* zu. In UML2 ist es möglich, dass die Klasse *Benutzer* ein Verhalten ausführt, welches wiederum die überschreibende Methode *senden* der Klasse *SecretChat* aufruft, statt die überschriebene Methode ihrer Oberklasse *Chat*. Dann ist die Sicherheitsdefinition der Klasse *Benutzer* unausreichend, da sie

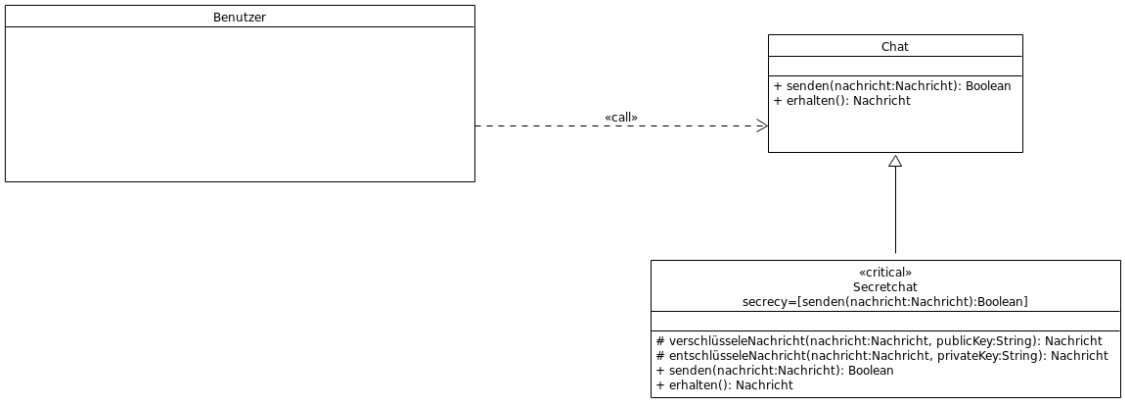

Abbildung 4.10: Die Klasse *Benutzer* hat eine call-Abhängigkeit zur Klasse *Chat*, deren Unterklasse *SecretChat* die Methode *senden* mit Sicherheitseigenschaften überschreibt.

nicht *secrecy* für die Methode *senden* definiert. Es sollten demnach nur die Methoden der Oberklasse aufgerufen werden, damit der Secure Dependency Check nicht fehlschlägt, allerdings kann dies in UML2 nicht gewährleistet werden. Es ergeben sich die folgenden zwei Lösungsansätze, um mit diesem Sachverhalt umzugehen:

1. **Eine Unterklasse darf bei überschreibenden Methoden keine Sicherheitseigenschaften hinzufügen.** Dann würde es bei call-Abhängigkeiten zu Oberklassen, wie zwischen *Benutzer* und *Chat* in Abbildung 4.10, keinen Unterschied für den *Secure Dependency Check* machen, ob *Benutzer* nun eine Methode der Oberklasse aufruft oder eine Methode seiner Unterklassen. Sicherheitseigenschaften der Oberklasse werden an ihre Unterklasse vererbt, aber die Unterklasse darf keine Sicherheitseigenschaften hinzufügen. Damit haben überschreibende und überschriebene Methoden stets die gleiche Menge an Sicherheitseigenschaften. Schlägt der *Secure Dependency Check* für die call-Abhängigkeit zur Oberklasse nicht fehl, so gilt dies auch für alle Fälle, in denen überschreibende Methoden der Unterklassen aufgerufen werden.

2. **Eine Unterklasse darf bei überschreibenden Methoden weitere Sicherheitseigenschaften hinzufügen.** Dann können bei call-Abhängigkeiten zu Oberklassen, wie zwischen *Benutzer* und *Chat* in Abbildung 4.10, seitens des Clients die Methode der Oberklasse *senden* aufgerufen werden oder die überschreibende Methode *senden* der Unterklasse *SecretChat*. Dann muss in der Unterklasse sichergestellt werden, dass man kompatibel im Kontext bleibt, in dem man verwendet werden kann. Die überschriebene Methode *senden* muss also im Kontext ihrer Oberklasse *Chat* verwendet werden können, ohne dass es zu Sicherheitsverletzungen kommt. Wird die überschreibende Methode *senden* von *SecretChat* seitens Benutzer aufgerufen, so darf diese Methode in diesem Kontext keine Sicherheitseigenschaften anfordern, die nicht bereits in ihrer Oberklasse *Chat* gefordert werden. Bei einer direkten call-Abhängigkeit auf *SecretChat* sollte sie aber auch ihre eigenen zusätzlich definierten Sicherheitseigenschaften fordern.

Der erste Lösungsansatz hat den Nachteil, dass es restriktiv ist. Eine überschreibende Methode der Unterklasse hat eine andere Implementierung ihrer Funktionalität und könnte damit auch zusätzliche Sicherheitsanforderungen fordern. Mit diesem Ansatz würde dies verhindert werden. Dafür werden aber zuvor genannte Problemfälle (call-Abhängigkeiten zu Oberklassen) verhindert. Bei einer call-Abhängigkeit zu einer Oberklasse ergeben sich keine Sicherheitslücken in Bezug auf Unterklassen, die Methoden ihrer Oberklassen überschreiben. Der zweite Lösungsansatz ist flexibler. Eine Klasse darf weitere Sicherheitseigenschaften für überschreibende Methoden definieren. Dabei muss jedoch darauf geachtet werden, dass bei der Modellierung und Implementierung von Unterklassen mit Methodenüberschreibungen diese Methoden im Kontext ihrer Oberklassen aufgerufen werden können. Eine Entscheidung zwischen diesen beiden Lösungsansätzen ist eine Entscheidung zwischen mehr Sicherheit und mehr Flexibilität. In dieser Arbeit wird die Sicherheit priorisiert und dafür ein leichter Verlust an Flexibilität akzeptiert. Es wird nicht erlaubt, dass Unterklassen bei überschreibenden Methoden Sicherheitseigenschaften hinzufügen können.

Analog gilt dasselbe für Überschreibungen von Attributen. Auch Attribute können mit dem gleichen Typen oder einem Untertypen in Unterklassen überschrieben werden und auch hier verbieten wir aus den selben Gründen ein Hinzufügen von Sicherheitseigenschaften in den Klasse der überschreibenden Methode.

In unserem Modell haben wir bisher das Überschreiben von Attributen und Methoden abstrahiert. Mit den in diesem Kapitel beschriebenen Beobachtungen werden wir unser Modell im nächsten Kapitel folgenderweise erweitern: Sicherheitseigenschaften, die sich auf überschriebene Attribute und Methoden beziehen, werden weiterhin vererbt. Eine Unterklasse darf jedoch bei überschreibenden Methoden keine weiteren Sicherheitseigenschaften definieren.

4.4.2 Erweiterung des formalen Modells mit Memberüberschreibungen

In Kapitel 4 definiert die Menge $ME(x) = \{(t_v, s_v), ..., (t_w, s_w)\}$ die explizit angegebenen Sicherheitseigenschaften einer Klasse x, wobei (t,s) ein Tag-Signatur-Paar annotiert. Nun wird diese Methodendefinition so verändert, dass überschreibende Methoden nicht in dieser Menge enthalten sind. Dadurch wird verhindert, dass überschreibende Methode Sicherheitseigenschaften hinzufügen, welche die überschriebenen Methoden nicht verfügen.

$$ME(u) = \{(t_v, s_w), ..., (t_w, s_w)\} \setminus \{(t_v, s_{mov}), ..., t_w, s_{mow}\} \text{ mit } v, w \in \mathbb{N}_0$$

Mit mo wird dabei eine überschreibende Methode bezeichnet und s_{mo} entspricht einer Signatur, die gleich der Methode mo ist. Hiermit werden überschreibende Methoden von der Menge ME(x) ausgeschlossen.

Analog kann das selbe Verfahren für überschriebene Attribute angewendet werden:

$$ME(u) = \{(t_v, s_v), ..., (t_w, s_w)\} \setminus \{(t_v, s_{aov}), ..., (t_w, s_{aow})\} \text{ mit } v, w \in \mathbb{N}_0$$

Mit ao wird dabei eine überschreibendes Attribut bezeichnet und s_{ao} entspricht erneut einer Signatur, die gleich dem Attribut ao ist. Überschreibende Attribute werden also von der Menge ME(x) ebenfalls ausgeschlossen.

5 Implementierung

Im vorherigen Kapitel wurde ein Modell entworfen, dass den *Secure Dependency Check* UMLsecs mit Vererbung von Sicherheitseigenschaften erweitert. Klassen vererben ihre Sicherheitseigenschaften an ihre Unterklassen, die dann auch in diesen gültig sind. Bei Methoden- und Attributüberschreibungen gilt ein Sonderfall, der besagt, dass überschreibende Methoden und Attribute keine weiteren Sicherheitseigenschaften hinzufügen dürfen. In diesem Kapitel wird vorgestellt, wie diese beiden Funktionalitäten in CARiSMAs *Secure Dependency Check* hinzugefügt wurden. Der Quellcode des CARiSMA Projekts ist auf dem Onlinedienst GitHub zu finden [CAR]. Für die Implementation der Funktionalitäten in dieser Arbeit wurde ein Branch mit dem Namen *VTopcu_Thesis_SecureDependencyInheritance* innerhalb des Projekts angelegt. Bei der Implementierung handelt es sich um ein eigenes Plugin namens *carisma.check.securedependency.inheritance*. Sie erweitert das Plugin *carisma.check.staticcheck* um ihre Funktionalität mit der Vererbung von Sicherheitseigenschaften. Sie beinhaltet die folgenden drei Klassen:

- SecureDependenciesInheritanceCheck.java
- SecureDependencyInheritanceChecks.java
- SecureDependencyInheritanceViolation.java

Die Implementierung lässt sich in zwei Funktionalitäten gruppieren: Erstens, die Vererbung von Sicherheitseigenschaften entlang einer Vererbungskette von hoher zu niedriger Hierarchie, und zweitens, die Restriktion das Hinzufügen von Sicherheitseigenschaften bei überschreibenden Methoden und Attributen zu verhindern. Im ersten folgenden Unterkapitel 5.1 wird die Implementierung der ersten und im zweiten Unterkapitel 5.2 die Implementierung der zweiten Funktionalität vorgestellt. Anschließend wird der Programmcode in Unterkapitel 5.3 auf Qualität untersucht.

5.1 Vererbung von Sicherheitseigenschaften

Die Funktionalität, Sicherheitseigenschaften an alle Unterklassen zu vererben, wird in der Klasse *SecureDependencyInheritanceChecks.java* realisiert. Die Methode *getCriticalTags* innerhalb dieser Klasse wird für Clients und Supplier von Dependencies aufgerufen. Sie fügt für diese Klassen alle Sicherheitseigenschaften (fresh, high, integrity, privacy und secrecy) in separate Listen hinzu, die im Nachhinein verglichen werden, um festzustellen, ob Sicherheitsverletzungen vorliegen. Statt wie im staticcheck nur die Sicherheitseigenschaften innerhalb der Klasse hinzuzufügen, wird nun mittels einer zweiten for-Schleife auch alle Oberklassen (direkte und indirekte)

auf Sicherheitseigenschaften (secrecy, integrity) überprüft. Falls eine Oberklasse eine Sicherheitseigenschaft hat und über das entsprechende Attribut oder die Methode als Klassenmember verfügt, wird diese Sicherheitseigenschaft in die jeweilige Liste hinzugefügt. Dadurch wird umgesetzt, dass eine Klasse nicht nur ihre explizit angegebenen Sicherheitseigenschaften enthält, sondern auch die ihrer Oberklassen:

Listing 5.1: Vererbung von Sicherheitseigenschaften

```
347 private void getCriticalTags(Classifier classifier, List<String> fresh, List<String>
348  high, List<String> integrity, List<String> privacy, List<String> secrecy) {
349
350 char bracket = '(', colon = ':';
351
352 //get stereotypes inside the class
353 for (EObject stereotype : classifier.getStereotypeApplications()) {
354         if (stereotype instanceof critical) {
355                 critical critical = (critical) stereotype;
356                 fresh.addAll(critical.getFresh());
357                 high.addAll(critical.getHigh());
358                 integrity.addAll(critical.getIntegrity());
359                 privacy.addAll(critical.getPrivacy());
360                 secrecy.addAll(critical.getSecrecy());
361         }
362 }
363 //get stereotypes inside of all parents, if the parent also has the corresponding
364 member
365 for (Classifier par : classifier.allParents()){
366         for (EObject stereotype : par.getStereotypeApplications()){
367                 if (stereotype instanceof critical){
368                         critical critical = (critical) stereotype;
369                         for (Property property : par.getAllAttributes()){
370                                 for (String sec : critical.getSecrecy()){
371                                         if (sec.indexOf(colon) != -1) {
372                                                 if (property.getName().equals
373                                                 (sec.substring(0,sec.indexOf
374                                                 (colon)))) {
375                                                 secrecy.add(sec);
376                                                 break;
377                                         }
378                                 }
379                                 }
380                                 for (String inte : critical.getIntegrity()) {
381                                         if (inte.indexOf(colon) != -1) {
382                                                 if (property.getName().equals
383                                                 (inte.substring(0,      inte.indexOf
384                                                 (colon)))) {
385                                                 integrity.add(inte);
386                                                 break;
387                                         }
388                                 }
389                                 }
390                         }
391                         for (Operation operation : par.getAllOperations()) {
392                                 for (String sec : critical.getSecrecy()){
393                                         if (sec.indexOf(bracket) != -1) {
394                                                 if (operation.getName().equals
395                                                 (sec.substring(0,sec.indexOf
396                                                 (bracket)))) {
397                                                 secrecy.add(sec);
398                                                 break;
399                                         }
400                                 }
401                                 }
402                                 for (String inte : critical.getIntegrity()) {
403                                         if (inte.indexOf(bracket) != -1) {
```

```
404                                                     if (operation.getName().equals
405                                                     (inte.substring(0, inte.indexOf
406                                                     (bracket)))) {
407                                                             integrity.add(inte);
408                                                             break;
409                                                     }
410                                             }
411                                     }
412                             }
413                     }
414             }
415 }
```

5.2 Methoden- und Attributüberschreibungen

Die Überprüfung, ob eine Klasse überschreibende Attribute und Methoden beinhaltet, erfordert einen etwas modulareren Ansatz: Eine Abhängigkeit ist bisher stets in der Form eines Clients und eines Suppliers und es wird überprüft, ob Client und Supplier gegenseitig ihre Sicherheitsanforderungen respektieren. Bei Überschreibungen von Methoden und Attributen ist es allerdings möglich, dass ein überschreibendes Attribut oder eine überschreibende Methode Sicherheitseigenschaften hinzufügt, ohne dass es sich bei dieser Klasse um einen Client oder einen Supplier handelt. Dennoch müssen auch diese Klassen auf Richtigkeit seitens des Secure Dependency Checks überprüft werden. Um dies zu realisieren, wird zu Beginn der Methode *checkSecureDependency* alle Klassen des Modells an die Methode *analyzeClassifier* übergeben, um diese auf überschreibende Attribute und Methoden zu überprüfen:

Listing 5.2: Auf überschreibende Member zu überprüfende Klassen

```
89 public int checkSecureDependency(final Package model) {
90
91         List<Classifier> classifiersToCheck = new ArrayList<>();
92         classifiersToCheck.addAll(UMLHelper.getAllElementsOfType(model,
93         Classifier.class));
94         for (Classifier clas : classifiersToCheck) {
95                 analyzeClassifier(clas, classifiersToCheck);
96         }
```

Die Methode *analyzeClassifier* überprüft dann für jede überschreibende Methode, ob die jeweilige Klasse eine Sicherheitssignatur besitzt, die mit dieser übereinstimmt. Um dies zu realisieren, müssen der Name sowie alle Parameter und Typen der Sicherheitssignatur mit dem Namen sowie alle Parametern und Typen der Methode auf Übereinstimmung analysiert werden. Dafür wird die Methode *haveSameParametersAndTypes* aufgerufen. Diese extrahiert Informationen wie Typ und Parameter aus der gegebenen Signatur mittels regulärer Ausdrücke, wie in 5.3 abgebildet, und fügt diese zwei separaten Listen hinzu.

Listing 5.3: Reguläre Ausdrücke für Typen und Parameter

```
193 public boolean haveSameParameterAndTypes(List<Parameter> parameters,
194 String signature, List<Classifier> classifiersToCheck) {
195
196         String type ;
197         List<String> names = new ArrayList<>();
198         List<String> types = new ArrayList<>();
```

```
199         List<String> s_names = new ArrayList<>();
200         List<String> s_types = new ArrayList<>();
201
202         Pattern pattern = Pattern.compile("in\\s(.*?):");
203         Pattern pattern2 = Pattern.compile(":\\s(.*?)(,|\\))");
204         Matcher matcher = pattern.matcher(signature);
205         Matcher matcher2 = pattern2.matcher(signature);
206         Pattern pattern3 = Pattern.compile("\\):\\s(.*?)$");
207         Matcher matcher3 = pattern3.matcher(signature);
208
209         while (matcher.find()) {
210                 s_names.add(matcher.group(1));
211         }
212         while (matcher2.find()) {
213                 s_types.add(matcher2.group(1));
214         }
215         if (matcher3.find()) {
216                 s_types.add(matcher3.group(1));
217         }
```

Auch für die Parameter und Typen der Methode werden zwei Listen angelegt. Die Listen der Methode und der Signatur werden anschließend auf Gleichheit verglichen. Sind sie gleich, so bezieht sich die Signatur auf die überschreibende Methode und es wird eine Sicherheitsmeldung ausgegeben und eine neue *SecureDependencyViolation* hinzugefügt:

Listing 5.4: Hinzufügen einer Sicherheitsverletzung

```
136 if (haveSameParameterAndTypes(redefined_operation.getOwnedParameters(),
137     sec, classifiersToCheck)) {
138         description =  "Class␣\"" + clas.getName() + "\"␣overrides␣the␣method
139 ␣␣␣␣␣␣␣␣\"" + redefined_operation.getName() + "()\"␣and␣adds␣the␣security
140 ␣␣␣␣␣␣␣␣property␣{secrecy}!";
141         this.secureDependencyViolations.add
142         (new SecureDependencyInheritanceViolation(description,
143          clas, redefined_operation));
144         this.analysisHost.appendLineToReport(description);
145 }
```

Dabei gilt es allerdings auch zu beachten, dass bei Methodenüberschreibungen die Parameter durch Subtypen ersetzt werden können. Daher wird bei der Sicherheitssignatur für alle Parametertypen überprüft, ob diese Subtypen von anderen Typen sind. Falls dies der Fall ist, so wird der Subtyp durch ihren Supertypen ersetzt, damit bei der Überprüfung Signatur und Methode dennoch als übereinstimmend erkannt werden.

Nachdem alle überschreibenden Methoden einer Klasse überprüft sind, werden die überschreibenden Attribute auf Sicherheitsverletzungen analysiert. Dabei ist der Vorgang sehr ähnlich wie bei Methodenüberschreibungen. Da nur der Name des Attributs und sein Typ überprüft werden muss, wird jedoch eine Methode namens *compareType* aufgerufen. Diese vergleicht lediglich den Typen des Attributs mit der aus der Signatur mittels regulärem Ausdrucks extrahierten Typen. Stimmen diese überein, so fügt die Klasse dieser überschreibenden Methode eine Sicherheitseigenschaft hinzu, was einer Sicherheitsverletzung gleichkommt. Erneut wird dann eine Sicherheitsmeldung ausgegeben und eine neue *SecureDependencyViolation* hinzugefügt.

5.3 Untersuchung auf Qualität

Der entwickelte Programmcode wurde mittels Javadoc kommentiert. Dabei wurde der von Oracle vorgegebene Style Guide [OR1] als Richtlinie genommen.

Weiterhin wurde der Eclipse Plugin *Checkstyle* verwendet, um zu überprüfen, ob der geschriebene Programmcode den Programmierstandards entspricht. Mittels Checkstyle gefundene Standardabweichungen wurden, falls möglich, korrigiert. Als Einrückungslevel werden statt zwei Leerzeichen ein Tabulatorzeichen verwendet, wofür von Checkstyle standardmäßig eine Warnung ausgegeben wird. Weiterhin überschreiten einige Methodendeklarationen die maximale Anzahl an Zeichen von 100. Der Grund hierfür ist, dass diese Methoden viele Parameter besitzen und auf kurze Parameternamen zu Gunsten der Leseverständlichkeit des Programmcodes verzichtet wurde. Abgesehen von diesen zwei Punkten zeigt der Checkstyle für den Programmcode keine weiteren Warnungen.

Im Vergleich zum alten Plugin Staticcheck hat der in dieser Arbeit entwickelte Plugin folgende LoC (Lines of Code, Anzahl der Programmzeilen):

- SecureDependenciesInheritanceCheck.java verfügt über 99 Zeilen, 2 Zeilen weniger als SecureDependenciesCheck.java aus dem Staticcheck (101 Zeilen).
- SecureDependencyInheritanceChecks.java verfügt über 746 Zeilen, 264 Zeilen mehr als SecureDependencyChecks.java aus dem Staticcheck (482 Zeilen).
- SecureDependencyInheritanceViolation.java verfügt über 73 Zeilen, Zeilen mehr als SecureDependencyViolation.java aus dem Staticcheck (46 Zeilen).

Insgesamt bedeutet dies, dass der neue Plugin für seine neue Funktionalitäten den alten Plugin um 289 Programmzeilen erweitert, was eine Erhöhung der Anzahl an Programmzeilen um 46% bedeutet.

6 Validierung

Die Implementierung aus dem vorherigen Kapitel wurde mittels testgetriebene Entwicklung auf Korrektheit geprüft. Die einzelnen Testfälle wurden bereits vor der Implementierung mittels Papyrus [Pap], einer Modellierumgebung von Eclipse, entworfen und im Anschluss das Programm auf diesen ausgeführt. Allerdings wurden einige Testfälle nach der Implementierung hinzugefügt, da man während der Implementierung auf sieben noch fehlende Testfälle aufmerksam wurde und diese Lücken geschlossen werden mussten. Das Ziel dieser Testfälle ist es gewesen, alle möglichen Szenarien mittels möglichst einfacher Modellierung zu testen. Die Testfälle wurden einmal auf dem vorhandenen Staticcheck und einmal auf dem für diese Arbeit entwickelten Check mit Vererbung ausgeführt. Insgesamt wurden 29 Testfälle entworfen. Diese lassen sich in die folgenden drei Kategorien einordnen:

1. Fünf **Basisfälle**: Hierbei handelt es sich um Fälle, bei denen eine call-Dependency zwischen einem Client und einem Supplier besteht. Das Konzept der Vererbung ist in diesen Fällen nicht vorhanden. Das Ziel ist es, mittels dieser Fälle zu untersuchen, ob der neue Check ohne Vererbungen in den Modellen die gleichen Ergebnisse liefert wie der alte Check. Die Ergebnisse beider Checks sollten in diesen Testfällen übereinstimmen, denn der Check mit Vererbung soll für Fälle ohne Vererbung die gleichen Fehler finden wie der Staticcheck und keine unerwünschte Nebeneffekte aufweisen.

2. 16 **Vererbungsfälle**: In diesen Fällen ist immer ein Vererbungsmechanismus vorhanden. Sicherheitseigenschaften werden von Oberklassen an den Client und/oder an den Supplier vererbt. Je nach Szenario wird erwartet, dass der Check mit Vererbung mehr, weniger oder die gleiche Anzahl an Sicherheitsverletzungen findet wie der Staticcheck. Dies ist abhängig davon, ob und auf welche Art der Client und/oder der Supplier Sicherheitseigenschaften von ihren Oberklassen erben.

3. Acht **Überschreibungsfälle**: In diesen Fällen werden Methoden und Attribute in Unterklassen überschrieben. Es wird überprüft, ob diese Sicherheitsverletzungen, falls vorhanden, vom neuen Check gefunden werden. Es wird erwartet, dass der Staticcheck keine Sicherheitsverletzungen in diesen Fällen findet, der Check mit Vererbung allerdings möglicherweise Fehler finden kann.

In Tabelle 6.1 sind die Ergebnisse der einzelnen Testfälle einzusehen. Die Tabelle besteht aus vier Spalten: Die erste Spalte *Testname* gibt den Namen des Tests an. Die zweite Spalte *Anzahl Fehler* zeigt die tatsächliche Anzahl der Fehler im

Testname	Anzahl Fehler	Check mit Vererbung	Staticcheck
BasicCases01	1	1	1
BasicCases02	0	0	0
BasicCases03	2	2	2
BasicCases04	0	0	0
BasicCases05	1	1	1
InheritanceCases01	0	0	0
InheritanceCases02	1	1	1
InheritanceCases03	0	0	0
InheritanceCases04	2	2	2
InheritanceCases05	0	0	0
InheritanceCases06	1	1	0
InheritanceCases07	0	0	0
InheritanceCases08	1	1	1
InheritanceCases09	0	0	0
InheritanceCases10	1	1	0
InheritanceCases11	0	0	1
InheritanceCases12	1	1	0
InheritanceCases13	2	2	0
InheritanceCases14	1	1	0
InheritanceCases15	2	2	0
InheritanceCases16	0	0	0
OverridingCases01	1	1	0
OverridingCases02	0	0	0
OverridingCases03	1	1	0
OverridingCases04	1	1	0
OverridingCases05	0	0	0
OverridingCases06	1	1	0
OverridingCases07	1	1	0
OverridingCases08	2	2	0

Abbildung 6.1: Anzahl der Fehler beim Check mit Vererbung und beim Staticcheck im Vergleich: Gleiche Anzahl an Fehler sind mit Grün markiert, eine unterschiedliche Anzahl mit Rot.

Modell an. Die dritte Spalte *Check mit Vererbung* beschreibt, wie viele Fehler der neu entwickelte Plugin findet und die vierte Zeile *Staticcheck* gibt an, wie viele Fehler der alte Plugin findet. Ist die Anzahl der gefundenen Fehler eines Plugins wie erwartet, so ist diese Zelle mit Grün markiert. Hat ein Plugin zu viele oder zu wenige Fehler gefunden, so ist die jeweilige Zelle mit Rot markiert.

Es ist zu sehen, dass die Testfälle in Sachen Fehleranzahl sich in Relation so verhalten wie erwartet: Für die Basisfälle (BasicCases01, ..., BasicCases05) sind die Anzahl der Fehlermeldungen stets gleich. Der Check mit Vererbung verhält sich ohne einen Vererbungsmechanismus im Modell also exakt wie der Staticcheck. Für die Vererbungsfälle (InheritanceCases01, ..., InheritanceCases16) liegt wie erwartet neunmal die gleiche Fehleranzahl und siebenmal eine unterschiedliche Anzahl an Fehlern vor. Dabei findet der Check mit Vererbung sechsmal Fehler, die der Staticcheck nicht findet. Ein solcher Fall (InheritanceCases10) ist in Abbildung 6.2 zu sehen: Für den Staticcheck ist diese Dependency in Ordnung, da weder der Client C noch der Supplier S über Sicherheitseigenschaften verfügen. Der Check mit Vererbung erkennt hingegen, dass der Supplier S die Sicherheitseigenschaft *secrecy* für die Methode m() von seiner Oberklasse O erbt. Da der Client C über diese Sicherheitseigenschaft nicht verfügt, gibt der Secure Dependency Check eine Fehlermeldung aus.

Einmal findet der Staticcheck fälschlicherweise eine Sicherheitsverletzung, die der Check mit Vererbung richtigerweise als legitim anerkennt. Ein solcher Fall ist in Abbildung 6.3 abgebildet. Der Client C definiert in seiner Klasse *integrity* für die Methode m(). Dieses verfügt der Supplier S nicht unmittelbar in seiner Klasse, jedoch erbt sie von seiner Oberklasse O. Dies wird vom Staticcheck nicht erkannt und eine Fehlermeldung ausgegeben. Der Check mit Vererbung hingegen prüft für den Supplier S Sicherheitseigenschaften entlang seiner Vererbungskante und erkennt, dass S über diese Sicherheitseigenschaft verfügt. Für die Überschreibungsfälle (OverridingCases01, ..., OverridingCases08) findet der Staticcheck wie erwartet keine Fehler, wohingegen der Check mit Vererbung bei sechs Fällen, in denen Sicherheitsverletzungen vorliegen, die richtige Fehleranzahl angibt.

Insgesamt sind die Testfälle erfolgreich ausgeführt worden. Der Check mit Vererbung hat, wie erwartet, alle vorhandenen Sicherheitsverletzungen gefunden und für diese passende Fehlermeldungen ausgegeben. Der Staticcheck hat hingegen einige Fehler nicht finden können und für ein korrektes Modell fälschlicherweise eine Fehlermeldung ausgegeben. Mit dem neu implementierten Check mit Vererbungen können, der Definition dieser Arbeit folgend, Sicherheitseigenschaften vererbt werden und der Secure Dependency Check mit diesen Vererbungen umgehen und die Modelle, mit oder ohne Vererbungsmechanismus, auf ihre Sicherheitsdefinitionen prüfen. Der Staticcheck hingegen ist nicht in der Lage alle Sicherheitsverletzungen zu finden und findet in einigen Fällen auch in richtigen Modellen fälschlicherweise Sicherheitsverletzungen. Damit erweist sich der neue Check mit Vererbung als eine wichtige Erweiterung zum bisher vorhandenen Check.

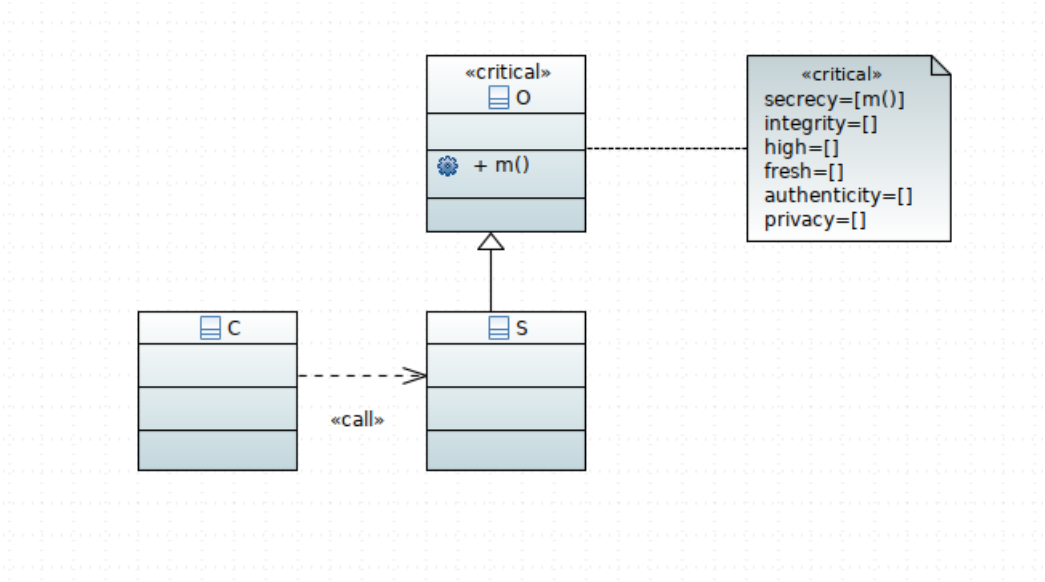

Abbildung 6.2: Client C hat eine Abhängigkeit zu Supplier S. Klasse S erbt Sicherheitseigenschaft von seiner Oberklasse O. Eine Sicherheitsverletzung liegt vor, der Secure Dependency Check schlägt fehl.

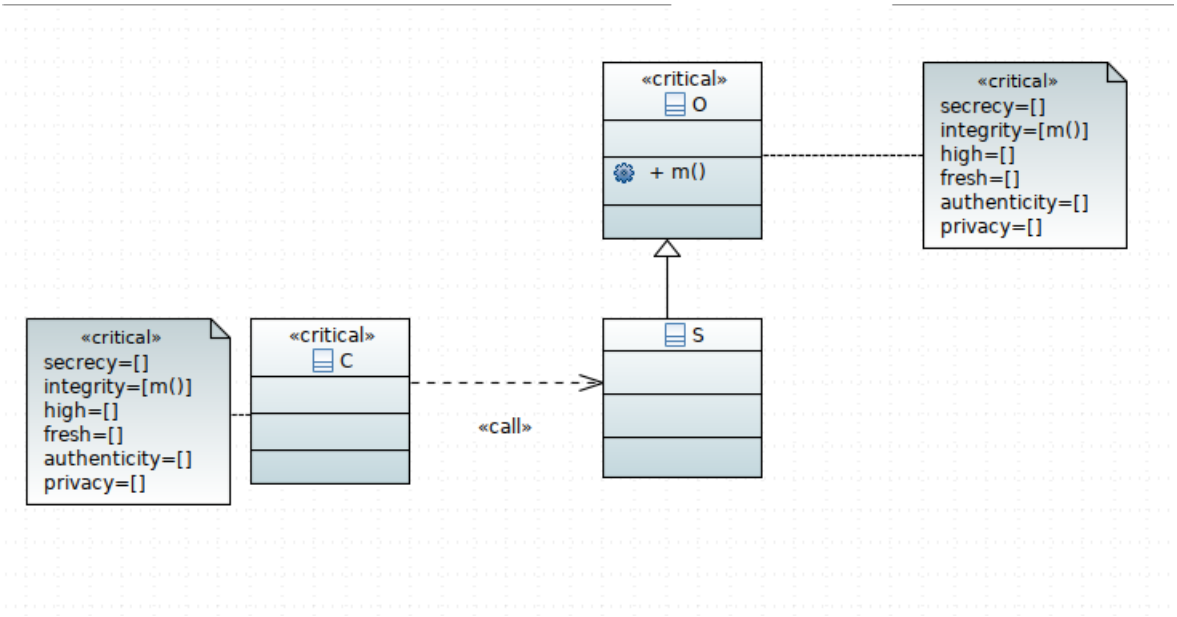

Abbildung 6.3: Client C mit Sicherheitseigenschaften hat eine Abhängigkeit zu Supplier S, welcher Sicherheitseigenschaften von seiner Oberklasse O erbt. Es liegen keine Sicherheitsverletzungen vor.

7 Related Work

In diesem Kapitel werden zu UMLsec ähnliche Modelliersprachen und Methoden aufgeführt, die sich als Ziel gesetzt haben, Sicherheitsanforderungen zu modellieren oder eine stärkere Sicherheit durchzusetzen und dabei, ähnlich zur in dieser Arbeit entwickelten Erweiterung, das Konzept der Vererbung in irgendeiner Form miteinbeziehen. Es wird erläutert, ob und auf welche Weise das Konzept der Vererbung in diesen Modelliersprachen vorhanden ist.

7.1 Misuse cases

Misuse cases ist eine Modelliersprache, welche die UML Diagrammart Anwendungsfalldiagramm (use case diagram) erweitert [SO05]. Das Ziel ist es, Sicherheitsanforderungen innerhalb von Anwendungsfalldiagrammen festzulegen, aber es kann auch für andere funktionale Anforderungen außerhalb der Sicherheit angewendet werden. Es werden positive Anwendungsfälle mit negativen Anwendungsfällen erweitert, um unerwünschtes Verhalten zu modellieren, um wiederum von diesen Sicherheitsanforderungen zu definieren. Negative Anwendungsfälle können Spezialisierungen von generellen Anwendungsfällen sein. Beispielsweise kann ein negativer Anwendungsfall *Erhalte Passwort* speziellere negative Anwendungsfälle haben wie *Erhalte Passwort durch Sniffing* oder *Erhalte Passwort durch Social Engineering* [SOB02].

7.2 SecureUML

SecureUML ist eine auf UML basierende Modelliersprache für modellgetriebene Softwareentwicklung von sicheren, verteilten Systemen [LBD02]. Sie nutzt RBAC und *Enterprise JavaBeans* (EJB) als Beispiel einer Kompenentenarchitektur. Da es in EJB keine direkte Repräsentation einer Rollenhierarchie gibt, erhält eine Subrolle in SecureUML alle Berechtigungen einer Superrolle ohne Ausnahmen [LBD02].

7.3 Ownership Types

Ownership Types bieten eine stärkere Sicherung von objektorientierten Programmiersprachen [CÖSW13]. Dort werden nicht nur Attribute von Objekten vor externen Zugriff beschützt, sondern auch Objekte, die als Attribute gespeichert sind. Dies hat zur Folge, dass Objekte andere Objekte besitzen und auf diese zugreifen können. Ursprünglich waren Ownership Types für kleine Sprachen entworfen worden, sodass keine Angabe darübergemacht wurde, wie mit Vererbung umgegangen wird. Diese Lücke wurde jedoch in den folgenden Jahren geschlossen. Um die Invariante

owners-as-dominators auch beim Subtyping durchzusetzen, wird die Schachtelungsbeziehung zwischen den Ownern in das Typsystem aufgenommen.

7.4 CORAS

CORAS ist eine von der SINTEF Gruppe entwickelte Methode zur Durchführung von Sicherheitsrisikoanalysen dar [DBHL+07]. Auch CORAS verwendet UML um das Ziel der Analyse zu modellieren. Allerdings hat CORAS keine eigene Definition in ihrer Sprache über das Konzept der Vererbung wie es in UML verwendet wird.

8 Fazit

In diesem Kapitel werden in 8.1 die wichtigsten Aspekte dieser Arbeit zusammengefasst, um einen kurzen Überblick über die in dieser Arbeit geschaffenen Leistungen zu bieten. Anschließend wird in 8.2 ein Ausblick auf zukünftige Arbeiten gegeben, welche die Erkenntnisse und Ergebnisse dieser Arbeit mit neuen Aspekten und Funktionalitäten erweitern können.

8.1 Zusammenfassung

Bislang wurde das Konzept der Vererbung in UMLsec abstrahiert. Um das Konzept der Vererbung in UMLsec miteinzubeziehen, wurde in dieser Bachelorarbeit die Vererbung von Sicherheitseigenschaften eingeführt. Dazu wurden die Ideen der beiden Eigenschaften des Bell-LaPadula Modells und der Rollenhierarchie des RBAC in das Klassendiagramm übertragen.

Die Simple Security-Eigenschaft wurde in dieser Arbeit mit dem *secrecy*-Tag in Verbindung gebracht. Sie gilt von nun an als eine Sicherheitsstufe für ihre Klasse, in der sie definiert ist. Jede andere Klasse mit einer call-Abhängigkeit zu dieser Klasse ohne diesen secrecy-Tag gilt als eine Klasse mit einer niedrigeren Sicherheitsstufe. Dies führt zu einem Read up und der Secure Dependency Check schlägt fehl.

In ähnlicher Form wurde die *-Eigenschaft des Bell-LaPadula Modells mit dem *integrity*-Tag in Verbindung gebracht. Auch sie gilt als eine Sicherheitsstufe für ihre Klasse. Jede andere Klasse mit einer call-Abhängigkeit zu dieser Klasse ohne diesen integrity-Tag gilt als eine Klasse mit einer höheren Sicherheitsstufe. Dies führt zu einem Write down und der Secure Dependency Check schlägt fehl.

Um die Vererbung von Sicherheitseigenschaften zu erlauben, wurde die Rollenhierarchie des RBAC als Motivation genommen. Bei dieser erben höherrangige Rollen Berechtigungen von niederen Rollen. Auf ähnliche Weise vererbten wir Sicherheitseigenschaften von Oberklassen zu ihren Unterklassen. Dies führt dazu, dass eine Unterklasse niemals eine niedrigere Sicherheitsstufe haben kann als ihre Oberklassen.

Nachdem diese Ideen zur Vererbung von Sicherheitseigenschaften feststanden, wurden diese in ein formales Modell überführt, das allgemeingültig ist. In dieser wurde die Simple Security-Eigenschaft, die *-Eigenschaft und die Vererbung von Sicherheitseigenschaften definiert. Eine besondere Achtung galt dabei der Überschreibung von Klassenmembern (Attribute oder Methoden), wie sie in UML erlaubt ist. Da es möglich ist, dass eine Unterklasse, die potentiell über mehr Sicherheitseigenschaften verfügt als ihre Oberklasse und damit sicherheitskritischer ist, im Kontext ihrer Oberklasse aufgerufen wird, machte hier eine spezielle Regelung erforderlich. Es wurde beschlossen, dass überschreibende Attribute und Methoden keine weiteren

Sicherheitseigenschaften definieren dürfen, um diese Art von Sicherheitslekagen zu vermeiden.

Nachdem die formale Definition vollendet war, wurde das entworfene Modell in Form eines CARiSMA Plugins in der Programmiersprache Java implementiert. Der neue Check erweitert den vorhandenen Check um die Funktion, Sicherheitseigenschaften zu vererben. Der neue Plugin wurde nach der Entwicklung auf seine Qualität untersucht und anschließend, mittels Testfällen, auf Korrektheit geprüft. Es zeigte sich, dass der neue Plugin bei Klassendiagrammen, in denen das Konzept der Vererbung vorhanden ist, mehr Sicherheitsverletzungen findet als der alte Plugin. Somit zeigt sich der in dieser Arbeit entwickelte CARiSMA-Plugin als eine wichtige Erweiterung zur bisherigen Funktionalität von CARiSMA.

8.2 Ausblick

Das in dieser Arbeit entworfene formale Modell zur Vererbung von Sicherheitseigenschaften sowie ihre Implementation kann in zukünftigen Arbeiten erweitert werden. In dieser Arbeit wurden nur die beiden Tags *secrecy* und *integrity* verwendet. UMLsecs *critical*-Stereotyp verfügt über weitere Tags wie *authencity*, *fresh* und *high*. Diese drei zusätzlichen Tags können grundsätzlich ebenfalls als Sicherheitsstufen für ihre jeweiligen Klassen gelten und vererbt werden. Außerdem ist eine Vererbungshierarchie zwischen einzelnen Tags vorstellbar. So kann ein Stereotyp wie *high* sowohl secrecy als auch integrity bereitstellen und diese einfordern.

In dieser Arbeit haben alle Methoden und Attribute in unserem Modell grundsätzlich die Sichtbarkeit *public*. Eine andere Sichtbarkeit wie *protected* oder *private* kann Auswirkungen auf die Vererbung ihrer korrespondierenden Sicherheitseigenschaften haben. Auch dies kann in einer zukünftigen Arbeit untersucht werden.

Außerdem wurde in dieser Arbeit nicht auf Mehrfachvererbungen von Klassen eingegangen, bei der eine Klasse mehr als nur eine direkte Oberklasse hat. Das Vorhandensein von mehr als nur einer direkten Oberklasse wirkt sich auch auf die Vererbung von Sicherheitseigenschaften aus und kann Thema einer zukünftigen Arbeit sein.

Literaturverzeichnis

[Anh] Gail-Joon Anh. Discretionary access control.

[Bel05] David Elliott Bell. Looking back at the bell-la padula model. In *21st Annual Computer Security Applications Conference (ACSAC'05)*, pages 15–pp. IEEE, 2005.

[BG04] Fabian Büttner and Martin Gogolla. On generalization and overriding in uml 2.0. In *OCL and Model Driven Engineering, UML 2004 Conf. Workshop, O. Patrascoiu, Ed. Univ. Kent*, pages 1–15. Citeseer, 2004.

[BL73] D Elliott Bell and Leonard J LaPadula. Secure computer systems: Mathematical foundations. Technical report, MITRE CORP BEDFORD MA, 1973.

[CAR] Carisma tool. `https://github.com/CARiSMA-Tool/carisma-tool`. Accessed: 2010-06-16.

[Chr16] Deccan Chronicle. Global cyber-crime costs to reach $2 trillion by 2019. 2016.

[CÖSW13] Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. Ownership types: A survey. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, pages 15–58. Springer, 2013.

[Cra03] Jason Crampton. On permissions, inheritance and role hierarchies. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 85–92. ACM, 2003.

[DBHL+07] Folker Den Braber, Ida Hogganvik, M Soldal Lund, Ketik Stølen, and Fredrik Vraalsen. Model-based security analysis in seven steps—a guided tour to the coras method. *BT Technology Journal*, 25(1):101–117, 2007.

[DY06] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Trans. Inf. Theor.*, 29(2):198–208, September 2006.

[FCK95] David Ferraiolo, Janet Cugini, and D Richard Kuhn. Role-based access control (rbac): Features and motivations. In *Proceedings of 11th annual computer security application conference*, pages 241–48, 1995.

[GB98] Cheh Goh and Adrian Baldwin. *Towards a more complete model of role*. Hewlett Packard Laboratories, 1998.

[JaD] The java language documentation. `https://docs.oracle.com/javase/specs/jls/se12/html/index.html`. Accessed: 2019-03-26.

[Jür05] Jan Jürjens. *Secure systems development with UML*. Springer Science & Business Media, 2005.

[Kuh97] D Richard Kuhn. Mutual exclusion of roles as a means of implementing separation of duty in role-based access control systems. In *ACM workshop on Role-based access control*, pages 23–30. Citeseer, 1997.

[LBD02] Torsten Lodderstedt, David Basin, and Jürgen Doser. Secureuml: A uml-based modeling language for model-driven security. In *International Conference on the Unified Modeling Language*, pages 426–441. Springer, 2002.

[Lin06] Hakan Lindqvist. Mandatory access control. *Master's Thesis in Computing Science, Umea University, Department of Computing Science, SE-901*, 87, 2006.

[Moa17] James Moar. The feature of cybercrime & security. 2017.

[OMG17] OMG. OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.5.1, December 2017.

[OR1] How to write doc comments for the javadoc tool. `https://www.oracle.com/technetwork/articles/java/index-137868.html`. Accessed: 2010-06-17.

[OSM00] Sylvia Osborn, Ravi Sandhu, and Qamar Munawer. Configuring role-based access control to enforce mandatory and discretionary access control policies. *ACM Transactions on Information and System Security (TISSEC)*, 3(2):85–106, 2000.

[Pap] Eclipse papyrus modeling environment. `https://www.eclipse.org/papyrus/`. Accessed: 2010-06-17.

[Per08] Chad Perrin. The cia triad. 2008.

[RSS94] Pierangela Samarati Ravi S. Sandhu. Access control: Principles and practice. 1994.

[SCFY96] Ravi S Sandhu, Edward J Coyne, Hal L Feinstein, and Charles E Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996.

[SO05] Guttorm Sindre and Andreas L Opdahl. Eliciting security requirements with misuse cases. *Requirements engineering*, 10(1):34–44, 2005.

[SOB02] Guttorm Sindre, Andreas L Opdahl, and Gøran F Brevik. Generalization/specialization as a structuring mechanism for misuse cases. In *Proceedings of the 2nd symposium on requirements engineering for information security (SREIS'02), Raleigh, North Carolina*, 2002.

[USMDAS14] A Ubale Swapnaja, G Modani Dattatray, and S Apte Sulabha. Analysis of dac mac rbac access control based models for security. *Analysis*, 104(5), 2014.

[ZC08] Gansen Zhao and David W Chadwick. On the modeling of bell-lapadula security policies using rbac. In *2008 IEEE 17th Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 257–262. IEEE, 2008.