

***Sicherheitsbewertung von ChatGPT
4.0-generiertem Programmcode auf Basis
von Extended Static Checking***

**Bachelorarbeit zur Erlangung des akademischen Grades
Bachelor of Science**

**Fachbereich 3: Informatik und Mathematik
Studiengang Informatik**

eingereicht von: Celina Röll
geboren am: 12.07.2001, Bremen
Matrikelnummer: 6039819
E-Mail: celina5@uni-bremen.de

1. Gutachter: Dr. Karsten Sohr
2. Gutachter: Dr. Sabine Kuske

eingereicht am: 11.07.2024

Inhaltsverzeichnis

1. Einleitung	1
2. Grundlagen	2
2.1. Sicherheitsziele	2
2.2. Symmetrische Kryptografie	2
2.3. Asymmetrische Kryptografie	4
2.4. Kryptografische Hashfunktion	5
2.5. Message Authentication Code	5
2.6. Kryptografische API	5
2.6.1. Java Cryptography Architecture und Extension	6
2.6.2. Cryptographic Service Provider	6
2.6.3. Keystore	7
3. Einführung in die verwendeten Tools	8
3.1. ChatGPT	8
3.2. Java Modeling Language	8
3.3. Extended Static Checker	10
4. Forschungsfragen und Ziele	11
5. Methodik	12
6. Fallstudie	15
7. Evaluation der Ergebnisse von ChatGPT	16
7.1. Security Package	17
7.2. Util Package	24
7.3. Model und Storage Package	24
7.4. Factory Package	31
7.5. Business_logic Package	33
7.6. GUI Package	36
8. Diskussion der Ergebnisse	38
9. Ausblick und Perspektiven	42
10. Zusammenfassung	43
A. Abbildungsverzeichnis	44
Literaturverzeichnis	58

Eigenständigkeitserklärung

Ich versichere, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Teile meiner Arbeit, die wortwörtlich oder dem Sinn nach anderen Werken entnommen sind, wurden unter Angabe der Quelle kenntlich gemacht. Gleiches gilt auch für Zeichnungen, Skizzen, bildliche Darstellungen sowie für Quellen aus dem Internet, dazu zählen auch KI-basierte Anwendungen oder Werkzeuge. Die Arbeit wurde in gleicher oder ähnlicher Form noch nicht als Prüfungsleistung eingereicht. Die elektronische Fassung der Arbeit stimmt mit der gedruckten Version überein. Mir ist bewusst, dass wahrheitswidrige Angaben als Täuschung behandelt werden.

Celina Röu

Bremen, den 10.07.2024

Einverständniserklärung zur elektronischen Überprüfung der Arbeit auf Plagiate

Eingereichte Arbeiten können nach § 18 des Allgemeinen Teil der Bachelor- bzw. der Masterprüfungsordnungen der Universität Bremen mit qualifizierter Software auf Plagiatsvorwürfe untersucht werden. Zum Zweck der Überprüfung auf Plagiate erfolgt das Hochladen auf den Server der von der Universität Bremen aktuell genutzten Plagiatssoftware.

Ich bin damit einverstanden, dass die von mir vorgelegte und verfasste Arbeit zum oben genannten Zweck dauerhaft auf dem externen Server der aktuell von der Universität Bremen genutzten Plagiatssoftware, in einer institutionseigenen Bibliothek (Zugriff nur durch die Universität Bremen), gespeichert wird.

Celina Röu

Bremen, den 10.07.2024

1. Einleitung

Large Language Models (LLMs) wie ChatGPT haben sich in den letzten Jahren erheblich weiterentwickelt. ChatGPT, entwickelt von OpenAI und Ende 2022 auf den Markt gebracht, erlangte aufgrund seiner vielfältigen Fähigkeiten ein besonderes öffentliches Interesse. ChatGPT bietet eine Vielzahl möglicher Funktionalitäten im Bereich der Computerprogrammierung, wie zum Beispiel die Vervollständigung, Korrektur, Fehlererkennung, Optimierung, Dokumentation und Generierung von Code an [1].

Die Meinungen über den zukünftigen Einsatz von LLMs in der Programmierung sind geteilt. Welsh, ein ehemaliger Professor an der Harvard University, vertritt die Ansicht, dass in Zukunft alle Programme ausschließlich von künstlichen Intelligenzen entwickelt werden könnten, wobei Menschen höchstens noch eine Aufsichtsfunktion besitzen [2]. Im Gegensatz dazu argumentiert Yellin, der Direktor des IBM Research Software Technology Departments, dass LLMs Programmierer nicht ersetzen können [3]. Immerhin müssen weiterhin die Spezifikationen und Anforderungen der Software definiert und überprüft werden, damit der generierte Code die Anforderungen erfüllt.

Die verstärkte Nutzung von LLMs im beruflichen Alltag wird durch eine kürzlich durchgeführte Studie von Wakefield Research und Github unterstrichen, an der 500 amerikanische Programmcodeentwickler teilnahmen. Dabei gaben 92% der Befragten an, ein AI-Coding-Tool entweder beruflich oder privat zu nutzen, und 70% sahen darin signifikante Vorteile [4].

Angesichts der zunehmenden Verwendung künstlicher Intelligenzen und Large Language Models im Arbeitsalltag stellt sich die Frage, wie zuverlässig und sicher der von ChatGPT, einem der bekanntesten Vertreter der LLMs, generierte Code ist. Zudem wird in dieser Arbeit auch auf eine geschickte Formulierung von Anfragen geachtet, um bessere Ergebnisse von ChatGPT 4.0 zu erhalten. Zur Überprüfung soll ChatGPT eine Stromabrechnungsanwendung mit kryptografischer API in Java implementieren. Der generierte Code wird mittels des ESC-Tools von OpenJML bezüglich der gewünschten Spezifikationen verifiziert und manuell überprüft.

Um die Verständlichkeit der Arbeit zu erhöhen werden grundlegende Begriffe und Definitionen in Kapitel 2 erklärt. Kapitel 3 liefert eine Einführung in die Tools wie OpenJML und ESC, die im Rahmen dieser Arbeit eingesetzt werden. In Kapitel 4 werden die Forschungsfragen und Ziele dieser Arbeit genauer spezifiziert. Die geplante Methodik wird in Kapitel 5 detailliert erläutert, gefolgt von der Beschreibung der zu implementierenden Fallstudie in Kapitel 6. Der Kern der Arbeit beschäftigt sich in Kapitel 7 mit der Evaluation und in Kapitel 8 mit der Diskussion der Ergebnisse von ChatGPT. Abschließend wird die Arbeit mit einem Ausblick auf zukünftige Forschungsarbeiten in Kapitel 9 sowie einer Zusammenfassung in Kapitel 10 abgerundet.

2. Grundlagen

Das folgende Kapitel soll zu bestimmten in dieser Arbeit verwendeten Begriffen kurze Definitionen liefern, um die Verständlichkeit der Arbeit zu erhöhen. Da es sich nicht um eine ausschließlich auf das Thema Kryptografie fokussierte Arbeit handelt, wird es nur eine kurze Beschreibung der wichtigsten Begriffe geben. Im Laufe der Arbeit wird die hier definierte Terminologie von Bedeutung sein und zum Einsatz kommen. Daher ist es empfehlenswert, sich mit den Definitionen vertraut zu machen.

2.1. Sicherheitsziele

Es gibt verschiedene Sicherheitsziele, die in dieser Arbeit in Bezug auf die zu implementierende Fallstudie gewährleistet werden sollen.

Hierzu gehört zum einen die **Vertraulichkeit**, welche durch den Einsatz der Kryptografie erreicht werden soll. Unter **Vertraulichkeit versteht man, dass nur autorisierte Personen lesenden Zugriff auf bestimmte Informationen haben**. Ein weiteres Sicherheitsziel, das zum Beispiel mittels kryptografischen Hashfunktionen erreicht werden soll, ist die **Integrität der Daten** (siehe Kapitel 2.4). **Daten sollen vor unautorisierter und unbemerkter Veränderung geschützt werden**. Außerdem kann durch die Verwendung von Message Authentication Codes auch die **Authentizität** sichergestellt werden (siehe Kapitel 2.5). Authentizität bedeutet, dass Informationen integer und frisch sind und eindeutig einer Identität zugeordnet werden können.

2.2. Symmetrische Kryptografie

Die *symmetrische Kryptografie* basiert darauf, dass Empfänger und Sender einen gemeinsamen geheimen Schlüssel haben mittels dem sie ihre Nachrichten ver- und entschlüsseln können. Wichtig dabei ist, dass keine dritte Person im Besitz des gemeinsamen geheimen Schlüssels kommt. Problem ist deshalb der Austausch dieses Schlüssels und dessen Geheimhaltung. Wichtig ist zudem, dass der Verschlüsselungsalgorithmus stark ist, sodass Angreifer keinerlei Informationen aus dem verschlüsselten Text, auch *Chiffretext* genannt, bezüglich des zugrunde liegenden Klartextes ziehen können [5].

Die Algorithmen zur symmetrischen Kryptografie unterteilt man in *Blockchiffren* und *Stromchiffren*. Bei einer Blockchiffre wird die Nachricht in Blöcke gleicher Größe unterteilt und blockweise mittels des Schlüssels verschlüsselt. Sollte also der letzte Block nicht ganz gefüllt sein, so muss dieser mittels eines Verfahrens, genannt *Padding*, mit zum Beispiel einen festen Wert aufgefüllt werden. Bei der Stromchiffre hingegen wird die Nachricht zeichenweise verschlüsselt. Es wird dabei ein Schlüsselstrom erzeugt, welcher die gleiche Länge wie der Klartext hat. Bei

der Verschlüsselung wird jeweils ein Klartextzeichen mit einem Schlüsselzeichen zu einem Chiffretextzeichen verknüpft.

Bei den Blockchiffren sind *DES* und *AES* zwei bekannte Verschlüsselungsverfahren. Bei *DES*, welches bereits als veraltet gilt und nicht mehr eingesetzt werden sollte, werden 64 Bit große Blöcke mit einem 56 Bit großen Schlüssel verschlüsselt. Bei *AES* werden 128 Bit Blocklängen und 128, 192 oder 256 Bit Schlüssellängen verwendet [6].

In *AES* unterscheidet man zudem zwischen verschiedenen Betriebsmodi, welche sich in ihrer Sicherheit und Umsetzung der Verschlüsselung unterscheiden. Zum einen gibt es da den *Electronic Code Book Mode (ECB-Mode)*. Wie die Abbildung 7 darstellt, verschlüsselt dieser Modus jeden Block einfach direkt mit *AES* und dem Schlüssel, wodurch sich die einzelnen Ciphertextblöcke ergeben. Dies ist natürlich nicht sicher, denn die Verschlüsselung desselben Blocks ergibt auch immer denselben Ciphertextblock. Deshalb sollte dieser Modus auf keinen Fall mehr verwendet werden, da Muster so schnell erkannt werden können.

Eine weitere interessante Betriebsart ist der *Cipher Block Chaining Mode (CBC-Mode)*, welcher in Abbildung 8 noch mal dargestellt ist. Hier werden die einzelnen Datenblöcke mit dem vorherigen verschlüsselten Block über das exklusive Oder (*XOR*) verknüpft und dann erst verschlüsselt. Der erste Block wird dabei mit einem *Initialisierungsvektor (IV)*, also einer zufälligen Zahl, anstelle des vorherigen verschlüsselten Blocks kombiniert. Dieser Prozess verknüpft jeden Block mit dem vorherigen, was dazu führt, dass identische Klartextblöcke unterschiedliche verschlüsselte Ciphertextblöcke ergeben, sofern sie an verschiedenen Stellen im Text stehen. In diesem sowie dem nachfolgenden Modus ist es dabei entscheidend, dass niemals derselbe Schlüssel mit einem bereits verwendeten *IV* für die erneute Verschlüsselung verwendet wird. Denn dies führt aufgrund der einfachen Umkehrfunktion des *XOR*s dazu, dass Nachrichten potenziell entschlüsselt werden können. Der *IV* kann dabei öffentlich sein, solange er zufällig und einmalig pro Nachricht ist.

Der dritte Modus, welcher in dieser Arbeit betrachtet wird, ist der *Galois Counter Mode (GCM)*, der sowohl die Ver- und Entschlüsselung von Nachrichten ermöglicht als auch deren Authentizität und Integrität sicherstellt. Dieser Modus verwendet eine Kombination aus einem *IV* und einem Zähler, um einen Strom von Schlüsselblöcken zu generieren. Diese Schlüsselblöcke werden mit den Daten *XOR*-verknüpft, um den Ciphertext zu erzeugen. Anschließend wird auf diesen Ciphertextblöcken, den zusätzlichen authentisierten Daten (*Additional authenticated data (AAD)*) und ihrer Längeninformationen, ein Message Authentication Code berechnet (siehe Kapitel 2.5). Dies geschieht mittels weiterer *XOR*-Verknüpfungen und der *Galois Field Multiplikation*, die auf den verschlüsselten Nullblock und die verknüpften Datenblöcke angewendet wird. Die folgende Abbildung zeigt die Ver- und Entschlüsselung mit *GCM*. Auf die *Galois Field Multiplikation* wird hier nicht genauer eingegangen. Weitere Informationen dazu und zu *GCM* können im Paper von McGrew und Viega aus dem Jahr 2005 nachgelesen werden [7]. Logischerweise ist es auch hier wichtig, dass niemals ein *IV*-Schlüsselpaar doppelt verwendet wird. Ähnlich wie im *CBC*-Modus kann die Wiederverwendung eines *IV*-Schlüsselpaares im *GCM* dazu führen, dass ein Angreifer Muster in den verschlüsselten Nachrichten erkennt. Dadurch könnte er den Schlüssel für die Verschlüsselung und für die Erstellung des Message Authentication Codes ermitteln und so die Integrität und Vertraulichkeit zukünftiger Nachrichten gefährden.

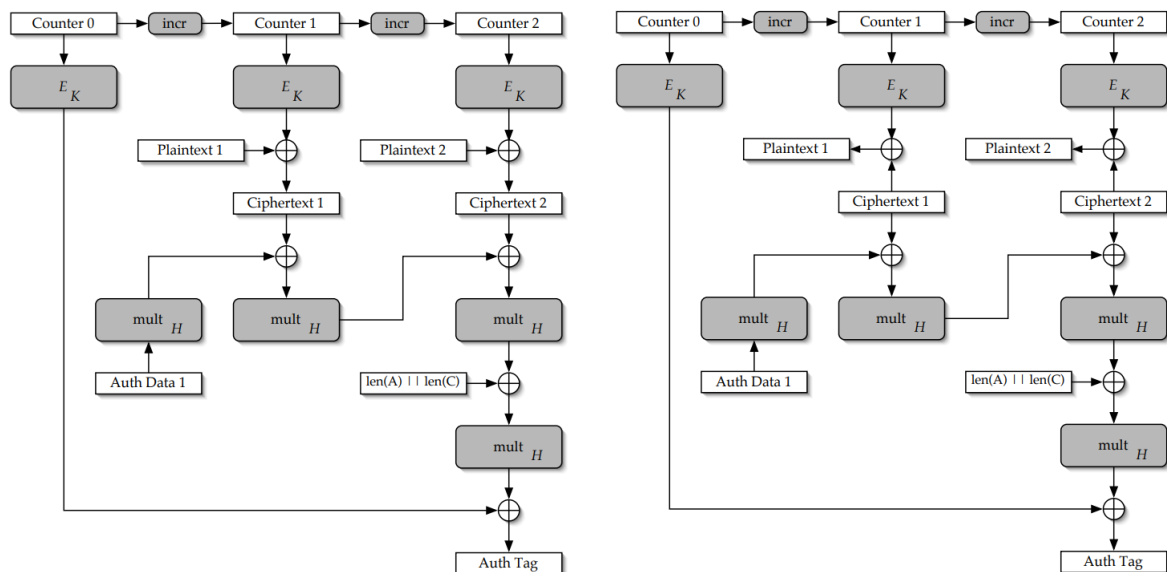


Abbildung 1.: Links die Verschlüsselung und rechts die Entschlüsselung mit GCM [7]

2.3. Asymmetrische Kryptografie

Im Gegensatz zur symmetrischen Verschlüsselung, bei der Sender und Empfänger denselben Schlüssel für das Ver- und Entschlüsseln verwenden, wird bei der *asymmetrischen Kryptografie* ein Schlüsselpaar eingesetzt. Das bedeutet, es gibt zwei unterschiedliche Schlüssel: einen *Public Key* und einen *Private Key*. Der Public Key, auch öffentlicher Schlüssel genannt, kann mit jedem geteilt und öffentlich zugänglich gemacht werden. Der Private Key hingegen muss unter allen Umständen geheim gehalten werden und wird deshalb oft auch als geheimer oder privater Schlüssel bezeichnet.

Der geheime Schlüssel wird zum Entschlüsseln von Nachrichten verwendet, während der öffentliche Schlüssel zum Verschlüsseln eingesetzt wird. Da nur eine Person im Besitz des privaten Schlüssels ist, bedeutet dies, dass Nachrichten oder Daten, die einmal mit dem Public Key verschlüsselt wurden, nur mit Hilfe des Private Keys entschlüsselt werden können [5].

Der besondere Vorteil der asymmetrischen Verschlüsselung liegt darin, dass das Problem des Schlüsselaustauschs nicht wie bei der symmetrischen Kryptografie besteht. Es bedarf keines sicheren Kommunikationskanals, um den Schlüssel an die Kommunikationspartner zu verteilen, da der Public Key allgemein bekannt sein darf. Ein weiterer Vorteil ist, dass nicht für jeden Kommunikationspartner ein neuer Schlüssel benötigt wird, was den Bedarf an einer hohen Anzahl an Schlüsseln, wie bei der symmetrischen Verschlüsselung, verhindert.

Jedoch sind asymmetrische Verschlüsselungsverfahren deutlich rechenintensiver als symmetrische Verfahren, weshalb oft in der Praxis eine Kombination beider eingesetzt wird. Die eigentlichen Daten werden dabei weiterhin symmetrisch verschlüsselt, während der Austausch des symmetrischen Schlüssels zuvor mittels asymmetrischer Verschlüsselung erfolgt [5].

Aufgrund der mathematischen Eigenschaften dieses Schlüsselpaares ist es nahezu unmöglich, aus dem öffentlichen Schlüssel den privaten Schlüssel herzuleiten. Ein Beispiel für ein asymmetrisches Verschlüsselungsverfahren ist der *RSA-Algorithmus* [5].

2.4. Kryptografische Hashfunktion

Eine *kryptografische Hashfunktion* ist eine Funktion, die aus einer Nachricht eine Bitfolge mit fester Länge berechnet und dabei bestimmte Sicherheitseigenschaften wie die *Kollisionsresistenz* erfüllt. Eine kryptografische Hashfunktion ist kollisionsresistent, wenn es so gut wie unmöglich ist, zwei verschiedene Werte zu finden, welche denselben Hashwert erzeugen. Zudem handelt es sich bei kryptografischen Hashfunktionen um *Einwegfunktionen*, was bedeutet, dass sich der Hashwert der Nachricht effizient berechnen lässt, aber es extrem schwierig ist, aus dem Hashwert die ursprüngliche Nachricht zu rekonstruieren. Der berechnete kryptografische Hashwert kann somit als „Fingerabdruck“ der Nachricht betrachtet werden. Schon die kleinste Änderung der Nachricht, beispielsweise ein ausgetauschter Buchstabe, führt zu einem vollkommen anderen kryptografischen Hashwert.

Ein kryptografischer Hashwert stellt somit auch eine Möglichkeit der Integritätsüberprüfung dar. Es gibt verschiedene Verfahren wie *MD5*, *SHA-1*, *SHA-256* und *SHA-512/Keccak*, wobei die beiden Erstgenannten als gebrochen gelten und nicht mehr eingesetzt werden sollten [5].

2.5. Message Authentication Code

Um die Integrität und Authentizität einer Nachricht zu garantieren, gibt es die sogenannte *Prüfsumme*, welche auch als *Message Authentication Code (MAC)* bezeichnet wird. Die Prüfsumme wird normalerweise berechnet, indem eine kryptografische Funktion direkt auf die Nachricht und einen Schlüssel angewendet wird. MACs verwenden hierbei symmetrische Verschlüsselungsverfahren anstelle von asymmetrischen und basieren auf kryptografischen Hashfunktionen oder Blockchiffren.

Ein auf kryptografischen Hashfunktionen basierender MAC wird als *Hash-Based Message Authentication Code (HMAC)* bezeichnet. HMAC verwendet die Nachricht zusammen mit dem symmetrischen Schlüssel als Eingabe für die kryptografische Hashfunktion, anstatt nur den kryptografischen Hashwert der Nachricht zu berechnen. Dabei wird zuerst der innere Hashwert berechnet, indem der Schlüssel und eine Konstante „inner pad“ XOR-verknüpft und mit der Nachricht kombiniert werden. Anschließend wird das Ergebnis gehasht. Danach wird der äußere Hashwert berechnet. Dafür wird der Schlüssel mit einer anderen Konstante, dem „outer pad“, ebenfalls mittels XOR verknüpft. Dieses Ergebnis wird dann mit dem inneren Hashwert kombiniert und erneut gehasht, um den endgültigen HMAC-Wert zu erzeugen [5].

2.6. Kryptografische API

Es folgt nun eine kurze Einleitung, um die Grundkonzepte und den Aufbau der Java Cryptography Architecture (JCA) und der Java Cryptography Extension (JCE) besser zu verstehen. Diese APIs werden für die Implementierung der anvisierten Anwendung benötigt.

2.6.1. Java Cryptography Architecture und Extension

Die Java-Plattform stellt eine umfangreiche kryptografische API bereit, die unter anderem die Realisierung verschiedener kryptografischer Grundelemente wie Hashfunktionen, Verschlüsselungsalgorithmen, Funktionen zur Generierung kryptografischer Schlüssel und Schlüsselverwaltung ermöglicht. Diese Funktionalitäten werden durch die *Java Cryptography Architecture (JCA)*, das offizielle Kryptografie-Framework in Java, unter dem `java.security`-Paket, bereitgestellt [8].

JCA ist darauf ausgelegt, Java-Entwicklern eine einfache Integration von Kryptografie zu ermöglichen, und bietet viele Methoden für verschiedene kryptografische Grundelemente an [9, 10]. JCA trennt dabei die APIs von den zugrunde liegenden Implementierungen, die von unterschiedlichen Providern, wie Bouncy Castle oder Sun, bereitgestellt werden können [8].

JCA weist gewisse Charakteristika auf: Zum einen die *Algorithm Independence*. Das bedeutet, dass die JCA kryptografische Operationen in gut dokumentierte Kategorien, auch als *Engines* bezeichnet, einordnet, um eine einfache Kategorisierung und Nutzung zu ermöglichen. Eine weitere Charakteristik ist die *Algorithm Extensibility*, die das Hinzufügen neuer Algorithmen ermöglicht, welche über eine der unterstützten Engines zugänglich sind. Des Weiteren gibt es die *Implementation Independence*, wodurch Entwickler sofortigen Zugriff auf eine Funktionalität erhalten, ohne provider-spezifische Methoden aufrufen zu müssen. Zudem ist es möglich, dass, ohne dass spezielle Maßnahmen erforderlich sind, unterschiedliche Anbieter für verschiedene Teile einer kryptografischen Operation genutzt werden können. Dies wird als *Implementation Interoperability*-Charakteristik bezeichnet [11].

Die *Java Cryptography Extension (JCE)* ist eine Erweiterung der JCA, die zusätzliche Engines bereitstellt und einen weiteren Provider, SunJCE, integriert. Zudem befindet sich JCE innerhalb eines anderen Pakets, dem `javax.crypto`-Paket [11].

Aufgrund der gebotenen Vielfalt an verschiedenen Algorithmen und der vielen Modi und Konfigurationsoptionen kommt es häufig zur falschen Anwendung kryptografischer APIs. In einer Studie von Krüger et al., in der über 7.000 Java-Anwendungen auf Missbrauch hinsichtlich Kryptografie untersucht wurden, ergab sich, dass 4.929 beziehungsweise 63% der untersuchten Artefakte mindestens eine Warnung erzeugten. Insgesamt gab es 22.664 fehlerhafte Verwendungen, also einen Durchschnitt von 3,1 Missbräuchen pro Artefakt. Etwa 90,5% der 4.929 Missbräuche betrafen die fehlerhafte Verwendung der Klasse `MessageDigest`. Am zweithäufigsten, mit 1.262 fehlerhaften Verwendungen, traten Fehler in der `Cipher`-Klasse auf. Da die `Cipher`-Klasse auch im Laufe dieser Arbeit zum Einsatz kommen wird, wird hier besonders auf deren korrekten Einsatz geachtet [12].

2.6.2. Cryptographic Service Provider

Cryptographic Service Providers (CSP) enthalten eines oder mehrere Pakete. Diese Pakete beinhalten konkrete Implementierungen kryptografischer Algorithmen. Die Basisklasse für alle Security Provider ist die `java.security.Provider`-Klasse. Somit kann ein bestimmter kryptografischer Service, wie die SHA-256 Hashfunktion, verwendet werden, indem die Implementierung von einem installierten Provider übernommen wird [9].

Zudem ist es möglich, eine Implementierung eines speziellen Providers zu verwenden. Wird der

Provider nicht angegeben, so werden die Datenbanken mit den möglichen Sicherheitsdiensten der einzelnen Provider durchsucht und der erste Provider, welcher eine passende Implementierung bietet, verwendet. Welche Provider-Datenbank zuerst überprüft wird, hängt von der bevorzugten Provider-Reihenfolge ab, die in der Laufzeitumgebung des Clients definiert und neu konfiguriert werden kann [9].

Es gibt verschiedene CSPs, wie Bouncy Castle, eine weit verbreitete Open-Source-Kryptografie-API [13], oder die von der JDK bereitgestellten Provider wie Sun, SunJCE, SunJSSE und SunRsaSign [9].

Es ist jedoch wichtig zu beachten, dass insbesondere in Universalanwendungen kryptografische Services nicht von einem speziellen Provider angefordert werden sollten. Das Problem hierbei ist, dass diese Anwendungen dann an diese speziellen Provider gebunden sind, welche jedoch nicht zwangsläufig in jeder Java-Implementierung vorhanden sein müssen. Zudem entfällt der Vorteil, verfügbare optimierte Provider mit höherer Präferenz einzusetzen [9].

2.6.3. Keystore

Logischerweise steht und fällt die Sicherheit des Systems mit der Geheimhaltung der entsprechenden Schlüssel. Deshalb sollten sich auch Gedanken über die sichere Speicherung der generierten Schlüssel gemacht werden. Die JCA bietet hierfür eine Datenbank namens *Keystore* an, die zum Verwalten von Schlüsseln und Zertifikaten dient. Die Schlüssel werden dabei in einer Datei mit der Endung *.keystore* im Benutzerverzeichnis gespeichert. Sowohl Oracle JDK als auch Bouncy Castle bieten eine Vielzahl an unterschiedlichen Typen von Keystores an, die mittels der *getInstance*-Methode der *KeyStore*-Klasse erzeugt werden können.

Diese Keystore-Typen definieren die Speicherung, das Datenformat der Keystore-Informationen sowie die verwendeten Algorithmen zum Schutz der Schlüssel und zur Wahrung der Integrität des Keystores. Der standardmäßige Keystore von Oracle JDK ist *PKCS#12*, ein plattformübergreifender Keystore, der auf dem RSA PKCS#12 Personal Information Exchange Syntax Standard basiert [9].

3. Einführung in die verwendeten Tools

Um zu verstehen, wie ChatGPT seine Ergebnisse generiert, folgt eine kurze Einführung. Zudem werden in diesem Kapitel die Grundlagen der verwendeten Tools vorgestellt, um die Überprüfung der Spezifikationen der von ChatGPT entwickelten Methoden besser nachvollziehen zu können.

3.1. ChatGPT

ChatGPT, ein Generative Pre-trained Transformer, ist ein *Large Language Model*, entwickelt von OpenAI. Verschiedene Techniken wie Natural Language Processing (NLP), maschinelles Lernen und Deep Learning werden in ChatGPT eingesetzt, um Menschen-ähnlichen Text zu generieren. Dafür wird ChatGPT mit einer großen Anzahl von vielfältigen Internettexten trainiert, ohne genau zu wissen, welche Dokumente in das Training eingeflossen sind [14].

Die Kernkomponente von ChatGPT ist das *Transformer-Modell*, ein neuronales Netz, das speziell für Sequenz-zu-Sequenz-Aufgaben entwickelt wurde. Um Ausgaben zu generieren, wird eine Eingabe zunächst in einzelne Wörter unterteilt und in einen Vektorraum überführt. Anschließend wird eine Antwort mithilfe eines Encoders und Decoders Wort für Wort generiert [15].

Bei einem *neuronalen Netz* handelt es sich um ein mathematisches Modell, das aus zahlreichen sogenannten Neuronen besteht, die mittels Verbindungen, bekannt als Gewichte, miteinander vernetzt sind. Das neuronale Netz funktioniert ähnlich wie das menschliche Gehirn: Jedes Neuron empfängt Informationen von außen oder von anderen Neuronen und erzeugt eine Ausgabe, die wiederum zur Eingabe eines anderen Neurons werden kann. Dadurch entsteht ein Netz von neuronalen Verbindungen. Ein neuronales Netz kann mit Daten trainiert werden, indem die zugrunde liegenden Gewichte angepasst werden [14].

GPT-4, das neueste Modell, das ChatGPT zugrunde liegt, ist zudem fähig, sich mit dem Internet zu verbinden, wodurch es im Vergleich zu seinem Vorgänger, GPT-3.5, stets auf dem neuesten Stand des Wissens ist.

3.2. Java Modeling Language

Zur Überprüfung, ob der generierte Programmcode den gewünschten Spezifikationen entspricht, wird *OpenJML* verwendet. OpenJML ist eine Sammlung von Tools, die mit der *Java Modeling Language (JML)* arbeiten können.

JML ist eine Spezifikationssprache für das Verhalten von Java-Programmen, die dazu dient, die Spezifikation von objektorientierter Software zu verifizieren [16]. Als eine *behavioral interface*

specification language (BISL) beschreibt JML zwei wichtige Aspekte eines Java-Moduls: Zum einen seine Interfaces, die aus den Namen und statischen Informationen der Java-Deklarationen bestehen, und zum anderen das Verhalten des Programms, das beschreibt, wie sich die Module bei Verwendung verhalten müssen [17].

JML wurde im Laufe der Zeit auf dem *OpenJDK Compiler*, eine Open-Source-Version von Java, die stets aktualisiert und beibehalten wird, aufgebaut. So entstand das in dieser Arbeit verwendete Tool *OpenJML* [18].

OpenJML kombiniert unter anderem das *Extended Static Checking Tool (ESC)* für statische Analysen (siehe Kapitel 3.3) und einen *Runtime Assertion Checking Compiler (RAC)* für Laufzeitüberprüfungen, um die Korrektheit von Java-Programmen zu gewährleisten.

Um nun genauer zu verstehen, wie (Open-)JML funktioniert, folgt ein kleines Beispiel für dessen Verwendung. Wer sich für eine genauere Erklärung interessiert, wie die Spezifikationen für JML definiert werden, sollte in den dafür vorgesehenen Handbüchern nachschlagen [18, 19].

Zuerst stellt sich die Frage, was unter dem Verhalten einer Methode verstanden wird. Das Verhalten einer Methode ist definiert als die möglichen Zustandsveränderungen, die sie durchführt, wenn sie aufgerufen wird. Dabei wird zwischen verschiedenen Zuständen unterschieden: Die *pre-states* sind die Menge der Zustände, für die der Methodenaufruf erlaubt ist. Die formale Beschreibung dieser Zustände wird als *precondition* bezeichnet. Es gibt eine Menge von Speicherorten, auf die der Zugriff und deren Änderung für die Methode erlaubt ist. Zuletzt unterscheiden wir noch die *post-states*, welche die Menge der Zustände der Methode nach deren Ausführung umfasst. Diese *post-states* können in normale Rückgaben, Rückgabe einer Exception oder keine Rückgabe unterschieden werden. Die formale Beschreibung der ersten beiden *post-states* wird als *normal postcondition* und *exceptional postcondition* bezeichnet [19].

In Listing 1 ist ein Beispielprogramm aus dem Paper „Preliminary Design of JML: A Behavioral Interface Specification Language for Java“ von Leavens, Baker und Ruby abgebildet, anhand dessen die Grundlagen für (Open-)JML erklärt werden [17].

Ein Kommentar, gefolgt von einem At-Zeichen (@), wird als *Annotation* bezeichnet. Diese Annotationen werden nur von JML verarbeitet und vom Java-Compiler selbst nur als Kommentar erkannt. Die Annotationen beschreiben dabei das erwartete Verhalten der Methode. Eine Menge von Annotationen wird auch als *Spezifikation* bezeichnet. Einzelne Annotationen innerhalb einer Spezifikation werden auch *Klauseln* genannt. *Statements* sind Anweisungen innerhalb der Annotationen, die spezifische Bedingungen oder Eigenschaften definieren.

In Zeile 3 wird mit der Annotation `/*@ public normal_behavior` angegeben, dass die Spezifikation für den Anwender gilt und, sofern die precondition erfüllt ist, die Methode eine normale Rückgabe und keine Fehlermeldung zurückgibt.

In Zeile 4 wird die Vorbedingung mit dem Schlüsselwort *requires* definiert. Hier wird vorgegeben, dass der Übergabeparameter *y* einen nicht-negativen Wert haben muss, damit die Wurzel in der Methode berechnet werden kann.

Mit dem Schlüsselwort *assignable* werden die Speicherstellen der Methode angegeben, auf die zugegriffen und die geändert werden dürfen. In diesem Beispiel darf die Methode *isqrt* keine Speicherorte zuweisen.

In Zeile 6 werden mittels *ensures*-Klauseln die Nachbedingungen beschrieben. Diese besagen, dass es ein positives Ergebnis (*\result*) gibt.

Zeile 7 bis 9 definieren, dass das Ergebnis eine Integer-Approximation der Quadratwurzel von

y ist. Das quadrierte Ergebnis darf nicht größer als das Argument y sein. Zudem besagt die Implikation in Zeile 8-9, dass, wenn das quadrierte Ergebnis plus Eins nicht negativ ist, es immer größer als y ist [17].

```

1 public class IntMathOps {
2
3     /*@ public normal_behavior
4         @ requires y >= 0;
5         @ assignable \nothing;
6         @ ensures 0 <= \result
7         @ && \result * \result <= y
8         @ && ((0 <= (\result + 1) * (\result + 1))
9         @ ==> y < (\result + 1) * (\result + 1));
10        @*/
11    public static int isqrt(int y) {
12        return (int) Math.sqrt(y);
13    }
14 }

```

Listing 1: Die Datei IntMathOps.java aus [17]

3.3. Extended Static Checker

Das *Extended Static Checking Tool (ESC)* ist ein statisches Analyseprogramm, das Software hinsichtlich der Korrektheit geforderter Spezifikationen untersucht. Der Begriff *extended* weist darauf hin, dass ESC mehr Fehler erkennt als herkömmliche statische Analyseprogramme. ESC verwendet einen automatischen Theorembeweiser zur Überprüfung der Semantik des Programms und erkennt dabei auch Laufzeitfehler moderner Programmiersprachen sowie Nebenläufigkeitsprobleme und überprüft Designentscheidungen [20].

ESC von OpenJML basiert auf ESC/Java und ESC/Java2, welche *modulare Argumentation* verwenden, um einzelne Programmteile separat, anstelle des gesamten Programms auf einmal, zu überprüfen. Dabei ist zu beachten, dass die statische Erkennung vieler Fehler, wie Array-Grenzfehler oder Null-Dereferenzierungen, unentscheidbar ist. Das Werkzeug kann daher sowohl falsch positive als auch falsch negative Ergebnisse liefern und ist weder korrekt noch vollständig. Da ESC/Java2 auf einem automatischen Theorembeweiser basiert, kann es zudem zum *specification creep*-Problem kommen. Manchmal hängen einfache Eigenschaften von komplexeren ab oder es werden viele Java-Bibliotheken aufgerufen, die ebenfalls annotiert werden müssen. In solchen Fällen könnte die Verifikation dieser einfachen Eigenschaften ebenfalls unentscheidbar sein [20].

Es ist wichtig zu erwähnen, dass die Ziele von ESC/Java und JML unterschiedlich sind. JML soll die vollständige Spezifikation von Programmen ermöglichen, während ESC/Java für leichtgewichtige Spezifikationen einzelner Methoden gedacht ist. Dies führt zu syntaktischen und semantischen Unterschieden zwischen den beiden Sprachen [20]. Im Rahmen dieser Arbeit wird OpenJMLs ESC verwendet, dass die Syntax und Semantik von JML in Form von *leichtgewichtigen Spezifikationen* nutzt. Eine leichtgewichtige Spezifikation verwendet keine Verhaltensschlüsselwörter wie `public normal_behavior` oder `public exceptional_behavior`; andernfalls handelt es sich um eine *schwergewichtige Spezifikation* [17].

4. Forschungsfragen und Ziele

LLMs wie ChatGPT finden immer mehr Anklang im Arbeitsalltag und werden verstärkt zur Codegenerierung herangezogen. Deshalb ist der Kernaspekt dieser Arbeit, die Qualität des generierten Programmcodes zu überprüfen. Im Laufe dieser Arbeit soll mittels OpenJML/ESC und manueller Überprüfung die folgende Frage beantwortet werden:

Entspricht der generierte Programmcodes den gewünschten Spezifikationen?

Insbesondere in Zeiten, in denen die IT-Sicherheit von Systemen, wie deren Integrität, Vertraulichkeit und Verfügbarkeit, immer wichtiger wird, ist es umso entscheidender, auch die Sicherheit des generierten Codes zu analysieren.

Eine repräsentative Umfrage der Bitkom Research im Jahr 2023 von über 1000 Unternehmen aus unterschiedlichen Branchen ergab, dass 82% der befragten Unternehmen eine Zunahme von Cyberangriffen auf ihr eigenes Unternehmen erwarten. 54% gehen von einem starken Anstieg der Angriffe aus [21]. Hinzu kommt, dass der durch IT-Ausrüstungs- und Datendiebstahl sowie Spionage und Sabotage entstandene Schaden 2018/19 bei 103 Milliarden Euro lag. Im Jahr 2021 wurde ein Rekord von 223 Milliarden Euro Schaden erreicht, was einer Verdopplung entspricht [22]. 2023 lag der entstandene Schaden in Deutschland immer noch bei 206 Milliarden Euro [21].

Deshalb ist es wichtig, festzustellen, ob der von ChatGPT generierte Code den Best Practices entspricht und sicher ist. Um dies zu testen, soll ChatGPT JCA und JCE verwenden, um spezifische Sicherheitsfunktionen in der Anwendung zu implementieren, deren Ergebnisse daraufhin untersucht werden. Es stellt sich diesbezüglich also die Frage:

Kann der von ChatGPT 4.0 generierte Code verwendet werden, oder gibt es Sicherheitsbedenken?

Abhängig davon, welcher Prompt übergeben wird, können sich die Ausgaben von LLMs zudem signifikant unterscheiden. Jules White et al. stellten fest, dass

[...] an LLM's output is only as good as the prompt it is given. [23]

Dies wird ein weiterer Aspekt dieser Arbeit sein. Es soll, wenn auch nicht mit Fokus darauf, versucht werden, die Prompts möglichst so zu gestalten, dass ein qualitativ gutes Ergebnis erzielt wird.

5. Methodik

Um die Fragen aus Kapitel 4 zu beantworten, wurde ein systematischer Ansatz gewählt, der im Flussdiagramm der Abbildung 9 visuell dargestellt wird.

Zuerst wurde die Architektur einer Stromabrechnungsanwendung konstruiert und ein PlantUML-Code dazu erstellt. Ein UML-Klassendiagramm beziehungsweise PlantUML-Beschreibung für die Anwendung zu erstellen und sich Gedanken über die Reihenfolge, Abhängigkeiten und Zusammengehörigkeiten der Klassen zu machen, hatte auch den Vorteil, selbst vorzeitig strukturelle und konzeptionelle Mängel in der eigenen Anwendung zu entdecken. Zudem hatten Antal et al. in ihrer Untersuchung herausgefunden, dass ChatGPT 4.0 UML-Klassendiagramme effektiv deuten kann [24]. Die PlantUML-Beschreibung der zu implementierenden Funktionen der Stromabrechnungsanwendung wurde dann an ChatGPT übermittelt und dessen Beschreibung und Bewertung gefordert. ChatGPT sollte dabei nicht nur zeigen, dass es den Inhalt des übergebenen PlantUML-Codes verstand, sondern auch aufzeigen, welche Informationen ihm möglicherweise noch für ein optimales Ergebnis fehlten. Diese zusätzlich angeforderten Informationen wurden dann später in weitere Prompts integriert, um das Ergebnis von ChatGPT zu verbessern.

Im Laufe der Arbeit war auch das *Prompt-Engineering* entscheidend. Prompt-Engineering beschreibt den Prozess, Eingaben beziehungsweise Prompts für künstliche Sprachmodelle so zu erstellen, dass eine möglichst präzise, relevante und kohärente Antwort erzeugt und das gewünschte Ergebnis generiert wird [25]. Deshalb wurde das *Role-Prompt-Pattern*, auch *Persona-Pattern* genannt, eingesetzt. Durch die Vorgabe, eine gewisse Rolle einzunehmen, kann das Verhalten von GPTs beeinflusst und die Ergebnisse dementsprechend verbessert werden [26]. Da für die Stromabrechnungsanwendung sowohl der Sicherheitsaspekt als auch der Einsatz von Best Practices und die Qualität des Codes entscheidend waren, wurden zwei Rollen vorgegeben. Dabei sollte sich ChatGPT in die Rolle eines „certified senior IT security specialist“ und eines „certified senior software engineers“ versetzen. Durch die zusätzliche Angabe, dass es sich um einen zertifizierten Senior handelte, wurde zudem der erwartete Kompetenz- und Erfahrungsstand impliziert.

Des Weiteren wurde durch den Einsatz von „From now on,...“ dem Large Language Model mitgeteilt, sich für den gesamten Chatverlauf in die vorgegebene Rolle zu versetzen [23]. Ein weiterer wichtiger Aspekt des ersten Prompts war die Angabe des grundsätzlichen Ziels und wie dieses erreicht werden sollte. Dadurch wusste ChatGPT, wie der Ablauf der Kommunikation vorgesehen war.

Nachfolgend wurde das Programm Schritt für Schritt zusammen aufgebaut. Wichtig dabei war, dass immer nur einzelne Methoden, mehrere einfache Methoden oder kleine Klassen in einem Schritt von ChatGPT generiert werden sollten. Dies verbessert das Ergebnis, da sich das LLM nur auf einen Aspekt der Aufgabe konzentrieren und nicht mehrere Aufgaben gleichzeitig erledigen musste. Durch klare Nummerierung, Auflistung und effektive Textstrukturierung wurde es erleichtert, Eingaben präziser zu formulieren, was ChatGPT zudem half, relevante und genauere

Antworten zu generieren.

Um das Ergebnis besser nachvollziehen zu können, wurden fünf verschiedene Prompt-Strukturen explorativ entwickelt und jeweils um die entsprechenden Vorgaben ergänzt. Zudem wurde jeder dieser Prompt-Strukturen eine Bezeichnung gegeben, damit die Prompts im Laufe der Arbeit leichter referenziert werden konnten.

Der erste Prompt („Develop“) diente zur Erstellung einer ganz neuen Klasse und deren Methoden oder Attribute. Falls es sich um eine einfache Model-Klasse handelte, wurden nur die gewünschten Attribute vorgegeben. Handelte es sich um eine Klasse, deren Funktionalität im Vordergrund stand, so wurde entsprechend die gewünschte Funktionalität spezifiziert.

[Next, develop/Now, we will focus on developing] the ... enabling This class should encapsulate the following key [functionalities/attributes]:

1. ...
2. ...

Could you please provide a comprehensive outline of the implementation for this [class/service], including any methods or functionalities associated with it?

Wenn das Konstrukt einer Klasse jedoch bereits stand und diese um gewisse Methoden oder Eigenschaften ergänzt werden sollte, wurde der „Enhance“-Prompt übergeben. Dieser Prompt ist dabei folgendermaßen aufgebaut:

Could you enhance the ... class to include ...? Specifically, ensure it:

1. ...
2. ...

Ein weiterer wichtiger Prompt („Remember“) diente dazu, ChatGPT an gewisse Vorgaben oder Ergebnisse zu erinnern. Dies erwies sich insbesondere dann als nützlich, wenn ChatGPT den thematischen Zusammenhang der Diskussion verlor und eine Fortführung der Kommunikation angestrebt wurde. Es war dabei erforderlich, genau zu definieren, welche der entsprechenden Aspekte angepasst werden sollten.

Please remember the given UML diagram at the beginning of our conversation and the definition of the ... class. Improve the code to fit the given UML specification. Specifically, ensure it:

1. ...
2. ...

Der nächste Prompt („Reflect“) diente dazu, dass ChatGPT seine Antwort reflektierte und die genauen Gründe hinter dem gelieferten Ergebnis präsentierte. Dieser Prompt basiert auf den von White et al. 2023 definierten Reflection-Pattern [26]. Außerdem wurde gefordert, den generierten Code gegebenenfalls zu korrigieren, falls es im Programmcode Fehler gab. Dieser Prompt wurde immer dann verwendet, wenn die Antwort von ChatGPT fehlerhaft war. Hiermit sollte überprüft werden, ob der Chatbot selbst in der Lage ist, seine Fehler zu erkennen und zu korrigieren.

Explain the reasoning and your assumptions for Additionally, identify any potential issues or limitations with this approach. Upon reflection, suggest improvements

or alternative strategies to mitigate these issues and provide the modified code.

Sollte ChatGPT den Fehler nicht erkennen, nicht auf die gewünschte Weise korrigieren oder sollte man direkt eine genaue Vorstellung von der notwendigen Lösung haben, so wurde der „Feedback“-Prompt formuliert, um ChatGPT seine Rückmeldung mitzuteilen.

Considering the ... of the ..., would it be prudent to ...? If so, please propose an effective way to implement this, ensuring the ... of the ..., and provide the modified code.

Darüber hinaus war auch die Reihenfolge der Implementierungsschritte entscheidend. Es musste darauf geachtet werden, dass zuerst die Klassen implementiert wurden, die möglichst unabhängig von anderen Klassen waren, und darauf folgend dann die Klassen, die diese benötigten. Dementsprechend war es notwendig, zuerst die Klassen im Security-Package zu implementieren und dann erst die darauf aufbauenden Model-Klassen. Nur so wusste der Chatbot, welche Methoden bereits existierten und wie diese verwendet wurden. Die Reihenfolge hatte somit einen signifikanten Einfluss auf das finale Ergebnis. Um die Wissenschaftlichkeit der Arbeit zu erhöhen und dem Nicht-Determinismus der von ChatGPT generierten Antworten entgegenzuwirken, wurden mehrere Chatverläufe mit nur geringfügigen Änderungen geführt [27].

Nachdem der Programmcode generiert worden war, wurde dieser mittels ESC von OpenJML überprüft. Außerdem erfolgte eine manuelle Überprüfung des Programmcodes auf Fehler, da es bei ESC, aufgrund der Unentscheidbarkeit, sowohl zu false positives als auch false negatives kommen kann [28]. Es galt dabei auch zu bedenken, dass das Verifikationstool in seinen Fähigkeiten, die Korrektheit von Programmcode zu überprüfen, beschränkt ist. Deshalb war es oft notwendig, die Annotationen einzuschränken und zu vereinfachen. Somit konnte die Komplexität der Verifikation reduziert werden, jedoch mit der Einschränkung, dass nicht alles verifiziert werden konnte.

6. Fallstudie

Um die Qualität des von ChatGPT generierten Programmcodes zu überprüfen, soll der Chatbot eine Stromabrechnungsanwendung implementieren. Die Idee dahinter ist, eine Software zu entwickeln, wie sie eventuell auch im realen Berufsalltag erwünscht sein könnte, jedoch ohne die Komplexität einer vollständig produktionsreifen Lösung. Durch bewusste Vereinfachungen soll die Analyse übersichtlicher und handhabbarer gestaltet werden.

Die Stromabrechnungsanwendung, als Fallstudie, wird lokal ausgeführt, ohne die Notwendigkeit einer Server-Client-Architektur. Diese Beschränkung erlaubt es, sich speziell auf die Qualität und Sicherheit, insbesondere der kryptografischen API, zu konzentrieren. Eine übermäßig komplexe Implementierung ist für das Ermitteln der Forschungsziele nicht notwendig und könnte die Analyse erschweren und unübersichtlicher gestalten. Die Benutzeroberfläche der Anwendung ist terminalbasiert und die Daten werden in Hashmaps anstelle von Datenbanken gespeichert, was die Komplexität weiter reduziert.

In Bezug auf die Funktionalität soll die Anwendung die Erstellung von Kunden mit Namen, E-Mail, Wohnadresse und Kundennummer ermöglichen. Des Weiteren wird es Stromzähler geben, für die Informationen wie die Zähler-Endnummer, der Messzeitraum, der alte und neue Zählerstand sowie der jährliche Stromverbrauch in kWh erfasst werden. Eine Assoziationsklasse für die Verträge verbindet jeden Kunden mit einem Stromzähler und beinhaltet Details wie die zugehörige Kunden- und Zähler-Endnummer, eine Vertragsnummer, die monatliche Abschlagsrate, die Gebrauchsrate und den Basispreis.

Wichtig zu beachten ist, dass ein Kunde mehrere Verträge eingehen und mehrere Stromzähler besitzen kann, aber ein Stromzähler gehört immer nur zu höchstens einen Kunden und es kann nur ein Vertrag für diesen abgeschlossen werden.

Die Anwendung bietet darüber hinaus verschiedene Funktionen zur Berechnung finanzieller Aspekte: Die Gesamtnettokosten für den Jahresverbrauch inklusive Grundpreis, den jährlichen Stromverbrauch, die Gesamtbruttokosten inklusive Mehrwertsteuer, die mittels des aktuellen Prozentsatzes von 19% auf den Nettopreis berechnet werden, sowie der neue monatliche Abschlagsbetrag für das kommende Jahr. Zusätzlich gibt es eine Funktion zur Berechnung der Differenz zwischen den tatsächlich entrichteten Beträgen und den berechneten Kosten, um zu bestimmen, ob der Kunde im Laufe des Jahres zu viel oder zu wenig gezahlt hat. Die Ergebnisse all dieser Berechnungen sollen zudem im Terminal ausgegeben werden.

Um die Fähigkeiten von ChatGPT zu überprüfen, sicherheitsrelevante Aspekte korrekt implementieren zu können, soll eine kryptografische API für die Ver- und Entschlüsselung der sensiblen Kunden-, Zähler- und Vertragsinformationen unter Beachtung der DSGVO implementiert werden. Damit neben der Vertraulichkeit auch die Authentizität und Integrität der Daten garantiert werden, wird AES im [Galois Counter Mode](#) zur Verschlüsselung verwendet.

Das Fallbeispiel und dessen Struktur sollen dabei unterstützen, zu evaluieren, wie gut ChatGPT in der Lage ist, nicht nur funktionsfähigen, sondern auch sicheren und robusten Code zu generieren.

7. Evaluation der Ergebnisse von ChatGPT

In den folgenden Unterkapiteln werden die Ergebnisse dieser Arbeit zusammengefasst. Zur besseren Übersichtlichkeit sind die Resultate nach den entsprechenden Packages geordnet und werden in der Reihenfolge präsentiert, in der sie von ChatGPT generiert wurden.

Die Abbildung 10 zeigt das UML-Klassendiagramm des generierten Programmcodes, einschließlich der eventuell notwendigen Verbesserungen, und dient zur Unterstützung des Verständnisses der nachfolgenden Kapitel.

Jedes Unterkapitel erläutert den Chatverlauf, den währenddessen generierten Programmcode sowie die zugehörigen OpenJML Spezifikationen. Die Interfaces wurden erst kurz vor der Erstellung der EBillingApp-Klasse generiert. Sinn dahinter ist, dass ChatGPT zu diesem Zeitpunkt bereits alle wichtigen Klassen implementiert hat und somit besser entscheiden kann, wie die verschiedenen Interfaces aufgebaut sein sollten. In den nachfolgenden Kapiteln werden die entsprechenden Interfaces jeweils direkt in ihrem zugehörigen Package erwähnt.

Darüber hinaus wurden insgesamt drei vollständige und zehn teilweise durchgeführte Chatverläufe mit ChatGPT dokumentiert, die sich nur geringfügig in ihren Prompts und PlantUML-Diagrammbeschreibungen unterschieden. Dies dient dazu, die später erwähnte Problematik des Nicht-Determinismus zu verringern. Der Schwerpunkt liegt jedoch auf dem Ergebnis des letzten Chatverlaufs, da die vorherigen Kommunikationen zusätzlich den Prozess der Annäherung an die letztendlich verwendeten Prompt-Strukturen darstellen. Es ist zu beachten, dass bei einzelnen Chatverläufen ein Problem bei der Speicherung der Antworten von ChatGPT auftrat, sodass nur die beiden Prompts, jedoch nicht die Antworten dazwischen sichtbar sind. Dies ist bedauerlich, konnte jedoch nicht verhindert werden. Der hier dargestellte Chatverlauf ist jedoch vollständig erhalten. Die URLs zu den Chatverläufen sind in der Datei **conversations.pdf** auf dem beiliegenden USB-Stick zu finden. Zusätzlich enthält der USB-Stick auch den gesamten Programmcode und ein javadoc-Verzeichnis. Die folgende URL verweist dabei auf den Chatverlauf, welcher hauptsächlich in dieser Arbeit referenziert wird: <https://chat.openai.com/share/5305a65d-8179-4bbc-9c50-5ecec510559d>

Beginn der Konversation:

Im ersten Prompt in Abbildung 11 wird ChatGPT das grundsätzliche Ziel und seine Rolle erklärt und eine PlantUML-Beschreibung übergeben. Um zu überprüfen, dass der Chatbot seine Aufgabe und die PlantUML-Beschreibung versteht, soll das LLM zu Beginn das UML-Klassendiagramm beschreiben und auf potenzielle Wissenslücken hinweisen. Der Chatbot scheint die Beschreibung des Diagramms grundsätzlich zu verstehen, obwohl die Beziehungen zwischen den einzelnen Klassen anscheinend nicht ganz klar sind. Die in Abbildung 12 zu sehenden zusätzlich angeforderten Informationen werden später in den entsprechenden Prompts integriert, weshalb als Nächstes mit der Implementierung der Klassen im Security Package begonnen wird.

7.1. Security Package

Im Folgenden wird die Implementierung des Security Packages detailliert beschrieben. Diese Implementierung basiert auf den theoretischen Grundlagen, die in Kapitel 2 dargelegt wurden.

Das Security Package besteht aus zwei Klassen und einem Interface. Es gibt die CryptographyAPI-Klasse, die den grundlegenden Programmcode enthält, welcher dann von anderen Klassen über die Service-Klasse SecurityService aufgerufen werden kann. Das Interface soll definieren, wie der SecurityService aufgebaut und verwendet werden soll. Da der SecurityService in den Model-Klassen benötigt wird, startet der Chatverlauf mit der Erstellung der Klassen innerhalb des Security Packages.

Chatverlauf

CryptographyAPI-Klasse: Wie zuvor im Kapitel 5 erklärt, werden die beschriebenen Prompts verwendet. Aufgrund dessen, dass die CryptographyAPI-Klasse umfangreich ist und kompliziertere Methoden enthält, bekommt ChatGPT auch hier erst mal das grundlegende gewünschte Ergebnis der CryptographyAPI-Klasse mittels des [Develop](#)-Prompts vorgelegt und danach werden Schritt für Schritt die einzelnen Methoden implementiert.

Abbildung 13 zeigt den Start der Konversation zur Erstellung der CryptographyAPI. Dabei wird eine Reihe von benötigten Attributen und Funktionalitäten vorgegeben, die ChatGPT im Laufe des Chatverlaufs umsetzen soll. Gewünscht ist ein Keystore- und SecureRandom-Objekt als mindestens notwendige Attribute dieser Klasse. Die genannten Attribute sind auf jeden Fall notwendig für die erwarteten Verschlüsselungsmechanismen, wie zum Beispiel dem Speichern eines symmetrischen Schlüssels im Keystore oder der Erstellung eines Initialisierungsvektors. Die Entscheidung über weitere logischerweise notwendige Attribute soll ChatGPT dabei selbst überlassen werden. Des Weiteren soll die Klasse folgende Funktionalitäten aufweisen:

- Eine Methode, um einen Keystore zu initialisieren oder zu laden, sofern bereits ein Keystore existiert.
- Eine Methode, um existierende symmetrische Schlüssel aus dem Keystore abzurufen oder einen neuen Schlüssel zu generieren und im Keystore zu speichern.
- Eine Methode zum Verschlüsseln von String-Eingaben unter Verwendung des dazugehörigen Schlüssels aus dem Keystore, welcher über seinen keyAlias identifiziert wird.
- Äquivalent dazu eine Methode zum Entschlüsseln von String-Eingaben.

Im ersten Schritt soll der Chatbot erst mal das *Singleton-Design-Pattern*, die grundlegenden Attribute sowie die erste genannte Methode implementieren.

Das Singleton-Design-Pattern ist ein Entwurfsmuster aus der Softwareentwicklung. Es stellt sicher, dass eine Klasse nur eine Instanz hat, und bietet einen globalen Zugriffspunkt auf diese. Es ist besonders nützlich in Situationen, in denen eine zentrale Verwaltung und Konsistenz gewährleistet werden sollen und dafür genau eine Instanz einer Klasse benötigt wird.

Die Antwort von ChatGPT zeigt, dass er die Aufgabe gut verstanden hat, denn der Chatbot erklärt in seiner Antwort zuerst im Detail, was er implementieren wird, und wie der Programmcode funktioniert.

```
1 private final String keystoreType = "PKCS12"; // Or "JKS"
```

```

2 private final String keystorePath = "path/to/keystore.p12"; // Path to
   the keystore
3 private final char[] keystorePassword = "changeit".toCharArray(); //
   Keystore password

```

Listing 2: Generierte Beispielwerte für keystoreType, keystorePath und keystorePassword

Der generierte Programmcode setzt die geforderten Vorgaben korrekt um, wobei die globalen Variablen keystorePath und keystorePassword nur Beispielwerte sind und entsprechend geändert werden sollten. Das Speichern von Geheimnissen direkt im Code oder im Versionskontrollsystem birgt natürlich Sicherheitsrisiken.

ChatGPT verwendet zudem *PKCS#12*; dieser Keystore-Typ gilt unter den verfügbaren Keystore-Typen von JCA und JCE als der weitverbreiteste und ist ab Java 9 der Standard-Keystore-Typ von Oracle JDK. Jedoch wird im Kommentar auch JKS vorgeschlagen, und aus Erfahrung ist klar, dass ChatGPT nicht immer als erste Wahl PKCS#12 wählt (siehe Abbildungen 14 bis 16). JKS und JCEKS sind dabei Java-spezifische Formate, wohingegen PKCS#12 ein offener Standard ist und auch außerhalb von Java verwendet werden kann. Zudem haben Focardi et al. festgestellt, dass PKCS#12 unter den genannten drei Keystores mehr Sicherheit und einen solideren Keystore bietet [13].

Des Weiteren bieten *Umgebungsvariablen* die Möglichkeit, sensible Informationen wie Passwörter zu speichern und somit die Gefahr zu umgehen, diese im Programmcode versehentlich offenzulegen [29].

Deshalb wird in der nächsten Anfrage mittels des *Enhance*-Prompts auch diesbezüglich noch mal nachgefragt. Es werden sich Umgebungsvariablen für den keystorePath und keystorePassword gewünscht, die ausgelesen werden sollen, sowie die Verwendung eines sicheren Keystore-Typs gefordert. ChatGPT ergänzt daraufhin den generierten Programmcode korrekt und begründet seine Entscheidungen.

Die generierte initializeKeyStore-Methode bezieht sich dabei auf den in Kapitel 2.6.3 beschriebenen JCA- und JCE-Keystore. Beim Laden des Keystores in den Speicher kann es im generierten Programmcode vorkommen, dass kein Passwort verwendet wird. Dies ist zwar nicht komplett falsch, da das Passwort in der load-Methode optional ist, es ist jedoch keine gute Praxis. Zudem wird dieses Passwort für eine zusätzliche Integritätsüberprüfung verwendet [13]. Sobald man den Chatbot aber darauf hingewiesen hat, wurde dies entsprechend korrigiert (siehe Abbildung 17).

Nun wird im nächsten Prompt nach demselben Schema gebeten, die Klasse um die zweite Methode retrieveOrGenerateKey zu erweitern. Darüber hinaus werden im Prompt noch weitere Eigenschaften der zweiten Methode spezifiziert. Dementsprechend soll die Methode gewährleisten, dass starke und empfohlene kryptografische Algorithmen für die Schlüsselerzeugung eingesetzt werden sowie die Überprüfung des Keystores auf Existenz des Schlüssels, bevor ein neuer Schlüssel generiert wird.

Auch die zweite Methode wird wie gewünscht umgesetzt.

```

1 // Generate a new AES key
2 KeyGenerator keyGenerator = KeyGenerator.getInstance("AES");
3 keyGenerator.init(256, secureRandom); // Use 256-bit AES for strong
   encryption
4 SecretKey newKey = keyGenerator.generateKey();

```

Listing 3: AES-Schlüsselerzeugung

ChatGPT entscheidet sich für die Verwendung des AES-Algorithmus mit 256 Bit langen

symmetrischen Schlüsseln. Dies entspricht den heutigen Sicherheitsstandards [30].

Im generierten Programmcode fehlt eine `CertificateException`, welche aufgrund der `store`-Methode des Keystores geworfen werden kann. Das Vergessen dieser Exception trat dabei nicht in jedem Chatverlauf mit ChatGPT auf.

```
1 if (keyStore.containsAlias(keyAlias) && keyStore.getKey(keyAlias,
    keystorePassword) != null) {
2     Key key = keyStore.getKey(keyAlias, keystorePassword);
3     if (key instanceof SecretKey) {
4         return (SecretKey) key;
5     } else {
6         throw new IllegalArgumentException("The Key under the
    keyAlias " + keyAlias + " is not SecretKey.");
7     }
}
```

Listing 4: Überprüfung des Rückgabetypes

Darüber hinaus fehlt eine Überprüfung, ob der Rückgabetype ein `SecretKey` ist, denn die `getKey`-Methode des Keystores kann sowohl den Datentyp `Key`, falls es sich um einen geheimen asymmetrischen Schlüssel handeln sollte, oder einen symmetrischen `SecretKey` zurückgeben. Da in diesem Programmcode aber nur ein `SecretKey` zurückgegeben werden sollte, wurde der Code um eine `if`-Abfrage nach dem Typ der erhaltenen Rückgabe und um die Ausgabe einer eventuellen Fehlermeldung erweitert.

Im nachfolgenden Prompt ging es dann um den Aufbau der Verschlüsselungsmethode. Sowie in den vorherigen Prompts werden zusätzliche Vorgaben an ChatGPT übergeben. Hierzu gehören die Verwendung der zweiten erstellten Methode zur Schlüsselwiederherstellung, die Verwendung von Best Practices zur Verschlüsselung, die sichere Erstellung von Initialisierungsvektoren (IV) und die sichere Kombination der verschlüsselten Daten, des IVs und des MACs. Mit der letzten Angabe wird ChatGPT indirekt die Verwendung von AES im Galois Counter Mode nahegelegt. In Kapitel 2.2 wurden die symmetrische Kryptografie und insbesondere der AES-Algorithmus unter Verwendung von GCM erläutert. In seltenen Fällen hat ChatGPT hier AES im CBC-Modus verwendet (siehe Abbildung 18). Das ist in Ordnung und auch bei richtiger Verwendung sicher, gewährleistet jedoch nicht die Verwendung eines zusätzlichen HMACs zur Integritäts- und Authentizitätsüberprüfung.¹ Wie die Abbildungen 14 und 15 eines früheren Chatverlaufs zeigen, wählte ChatGPT dort nur AES ohne Modusangabe aus. Dies ist auf keinen Fall zu empfehlen, da zumeist der ECB-Modus als Standard eingetragen ist. Dieser Fehler trat jedoch nur in einem Chatverlauf auf, in dem die Kombination der verschlüsselten Daten mit dem IV und dem MAC nicht gefordert wurde. Dies verdeutlicht aber dennoch die Relevanz, die gelieferten Ergebnisse zu überprüfen.² Interessant beim Ergebnis von ChatGPT ist auch, dass der Chatbot explizit auf das Risiko der doppelten Verwendung eines IVs zusammen mit demselben Schlüssel hinweist.

Zur Generierung des Initialisierungsvektors verwendet ChatGPT dann `SecureRandom`, einen kryptografisch starken Zufallszahlengenerator (RNG) [31]. Dem IV wird eine Länge von 96 Bit zugewiesen. 96 Bit würden laut den Forschungsergebnissen von Iwata et al. [32] und der NIST [33], das beste Mittelmaß von Interoperabilität, Effizienz, Sicherheit und Einfachheit bieten.

Die generierte Methode `encryptData` weist dabei als Rückgabetype ein `Byte-Array` auf. Dies entspricht nicht den Vorgaben im PlantUML-Code, ist aber ein typischer Rückgabetype für Verschlüsselungsmethoden. Dennoch wird hier mittels des [Feedback](#)-Prompts gefordert, einen

¹Siehe USB-Stick conversations.pdf unter „CBC_Solution“

²Siehe USB-Stick conversations.pdf unter „ECB_Solution“

encodierten String zurückzugeben, da damit im Laufe des Projektes besser gearbeitet werden kann. ChatGPT wählt hier logischerweise die Verwendung von *Base64*, einer Codierungsmethode, die Daten in eine ASCII-Stringdarstellung umwandelt [34]. ChatGPT erwähnt jedoch auch, dass eine potenzielle Gefahr besteht, da Strings in Java unveränderliche Objekte sind. Diese werden in einem *String-Pool* gespeichert, einer speziellen Speicherregion, die darauf abzielt, den Speicherverbrauch zu optimieren, indem identische Strings wiederverwendet werden. Damit hat der Entwickler in Java keinen Einfluss darauf, wie lange unverschlüsselte String-Daten im Speicher verbleiben und wann der Garbage Collector diese entsorgt. Eine Abhilfe könnte hierbei die Verwendung eines character-Arrays anstelle von Strings sein, die nach Verwendung direkt überschrieben werden können. Dies führt aber wiederum zu einer erhöhten Komplexität und erschwerten Wartbarkeit. Letztendlich ist die Wahl zwischen String und char[] ein Balanceakt zwischen Sicherheit und Praktikabilität. Da es sich hierbei nur um eine einfache Stromabrechnungsanwendung handelt, wird deshalb darauf im Folgenden verzichtet.

Zusätzlich gilt zu beachten, dass ChatGPT in der folgenden Programmzeile UTF-8 angibt:

```
byte[] inputData = inputStr.getBytes("UTF-8");
```

Die verwendete Entwicklungsumgebung IntelliJ IDEA 2022.3.1 (Ultimate Edition) weist dabei darauf hin, dass lieber das StandardCharsets.UTF_8 verwendet werden sollte, welches seit Java 7 verfügbar ist und im Gegensatz zur Angabe von UTF-8 die Typsicherheit gewährleistet. Dadurch vermeidet StandardCharsets.UTF_8 die Handhabung einer potenziellen UnsupportedOperationException, die sonst zur Laufzeit entstehen könnte.

Zu guter Letzt wird äquivalent zur encryptData-Methode noch die decryptData-Methode korrekt von ChatGPT implementiert.

Es fällt auf, dass ChatGPT grundsätzlich Probleme hatte, die Sichtbarkeit der Methoden aus der PlantUML-Diagrammbeschreibung korrekt umzusetzen. Deshalb kommt hier der **Remember**-Prompt zum Einsatz. Dieser soll ChatGPT an den anfänglich übergebenen PlantUML-Code erinnern und ihn explizit nach der Sichtbarkeit package-private überprüfen und korrigieren lassen. Die Methoden package-private zu machen, soll zur Kapselung der CryptographyAPI-Klasse beitragen und den Zugriff nur über den dafür vorgesehenen SecurityService ermöglichen. Der Chatbot erkennt, dass mehrere Methoden die falsche Sichtbarkeit gesetzt haben, und korrigiert daraufhin sein Ergebnis. Jedoch unterläuft ihm erneut ein kleiner Fehler: Er setzt die Sichtbarkeit der retrieveOrGenerateKey-Methode (zuvor fälschlicherweise als public deklariert) und der initializeKeyStore-Methode (zuvor richtig deklariert) auf package-private, obwohl beide Methoden als private deklariert sein sollten.

SecurityService-Klasse: Mittels des **Develop**-Prompts wird danach die nächste Klasse der SecurityService erstellt. Wie zuvor erklärt, soll es sich auch hier um eine Singleton-Instanz handeln, welche die CryptographyAPI-Klassenmethoden zum Ver- und Entschlüsseln zur Verfügung stellen soll. Trotz eines Fehlers im PlantUML-Code, bei dem der Rückgabotyp aus Versehen als void angegeben wurde, erkennt ChatGPT dennoch, dass die Rückgabe auch vom Typ String sein muss. Jedoch fehlt der keyAlias als Übergabeparameter beider Methoden. Stattdessen setzt ChatGPT für alle zu verschlüsselnden Daten einen konstanten keyAlias und empfiehlt, diesen oder basierend auf dem Typ der Daten oder des Benutzers einen keyAlias zu bestimmen. Ähnlich wie der zweite Vorschlag von ChatGPT war auch die grundlegende Idee für die Erstellung des keyAlias. Jedoch soll dieser in den Model-Klassen, basierend auf einem festen Präfix und einer zugehörigen individuellen und eindeutigen Nummer zusammengesetzt und dem SecurityService übergeben werden. Die Entscheidung fiel somit auf einen individuellen symmetrischen Schlüssel für jedes Customer-, ElectricityMeter- und CustomerElectricityContract-Objekt. Dies erfordert

zwar mehr Speicherplatz, erhöht aber die Sicherheit im Vergleich zur Verwendung eines symmetrischen Schlüssels für zum Beispiel alle Customer-Objekte zusammen. Des Weiteren weist ChatGPT darauf hin, das Schlüsselmanagement sowie die korrekte Umsetzung von Sicherheitsrichtlinien sicherzustellen.

Im danach folgenden Prompt wird dann noch mal gefordert, die Methode anzupassen und den `keyAlias` als Übergabeparameter in den beiden Methoden zu integrieren. Dies setzt ChatGPT auch korrekt um. Dennoch hat der Chatbot aus der PlantUML-Diagrammbeschreibung nicht erkannt, dass es sich um eine Singleton-Instanz handeln soll. Dies wird von ihm nicht implementiert, aber später im Chatverlauf dennoch verwendet. Im späteren Verlauf der Konversation erzeugt ChatGPT dann noch ein Interface für die `SecurityService`-Klasse.

Verifizierung mittels OpenJML

CryptographyAPI-Klasse: Der von ChatGPT generierte Programmcode wurde im vorherigen Abschnitt manuell untersucht und entsprechend ergänzt. Zudem soll der generierte Programmcode systematisch auf seine Korrektheit hin überprüft werden. Zur formalen Verifikation muss der Programmcode dementsprechend um OpenJML-Annotationen ergänzt werden. Wie in Kapitel 3.3 erwähnt, ist das OpenJML-Tool in seiner Verifikationsfähigkeit beschränkt, was zu gelegentlichen Falschmeldungen (false positives) und bestimmten Einschränkungen bei der Verifikation einzelner Klassen führte.

Wichtige Invarianten für die Verifikation sind die Überprüfung, dass das `secureRandom`-Objekt sowie der `keyStore`, `keystorePath` und `keystorePassword` direkt zu Beginn korrekt initialisiert werden. Das bedeutet, dass das `keyStore`- und das `secureRandom`-Objekt zu keinem Zeitpunkt nach dem Aufruf des Konstruktors null sind und für die restlichen genannten Variablen gilt zusätzlich, dass diese keine Leerstrings sein dürfen. Als Eingangsbedingung für den Konstruktor wird deshalb erwartet, dass der Aufruf von `System.getenv()` für das Passwort und den Pfad nicht null zurückgibt. Als Ausgangsbedingung wird erwartet, dass der `keyStore` sowie `secureRandom` die Invarianten erfüllen. Außerdem wird mittels der *signals*-Klausel vorgegeben, dass der Konstruktor eine `IllegalStateException` wirft, sofern der `keystorePath` oder `keystorePassword` Nullwerte sein sollten. Zu guter Letzt wird noch definiert, dass der Konstruktor *pure* ist. Eine Methode, ein Konstruktor oder eine Klasse werden mit *pure* annotiert, wenn diese keine Seiteneffekte auf das Java-Programm haben. Außerdem sollten Methoden, die mit *pure* annotiert werden, zwei Eigenschaften aufweisen: Sie sollten am besten kein exceptional behavior beinhalten und sie müssen terminieren. Deshalb dürfen auch nur *pure*-Methoden in Spezifikationen verwendet werden. Auch wenn im Konstruktor Variablen initialisiert werden und Seiteneffekte vorliegen, wird der Konstruktor dennoch meistens als *pure* annotiert, um vorzugeben, dass der Konstruktor nur Speicher beim Setzen der eigenen Felder modifiziert [35]. Damit die Verifikation auch mit den als privat deklarierten Instanzvariablen funktioniert, wird die *spec_public*-Annotation vor den Variablen benötigt. Durch diese Annotation wird die Sichtbarkeit der Instanzvariablen in der Verifikation auf public gesetzt. Diese Praxis stellt sicher, dass OpenJML auf die Variablen zugreifen kann und alle relevanten Zustände und Bedingungen der Klasse überprüft werden können.

Des Weiteren werden öfters *assume*-Statements aufgrund der Beschränktheit von OpenJML benötigt. Dadurch wird dem Verifikationstool die Überprüfung der Spezifikationen erleichtert beziehungsweise abgenommen, da bestimmte Annahmen über den Zustand des Programmes explizit gemacht werden. Ohne das *assume*-Statement kann JML nicht erkennen, dass nach dem

Initialisieren von `keystorePath` und der Variable `password` auf null überprüft wird. Somit geht das Tool davon aus, dass diese Überprüfung fehlt, und würde einen Fehler ausgeben.

Danach folgt für die `getInstance`-Methode eine Ausgangsbedingung, dass das Ergebnis nicht null sein soll, also dass das `CryptographyAPI`-Objekt korrekt initialisiert wurde. Des Weiteren wird auch die `initializeKeyStore`-Methode verifiziert. Diese erwartet, dass der übergebene `keystore`-Type korrekt auf PKCS#12 gesetzt ist und dass nach der Ausführung der Methode der `keyStore` initialisiert wurde. Zudem werden auch hier wieder Exceptions geworfen. Jedoch lassen sich hier die Bedingungen oft für OpenJML nicht verifizieren, weshalb das *signals_only* nur unter Aufzählung der möglichen Exceptions verwendet wird. Dies bedeutet, dass die Methode nur die darin genannten Exceptions werfen kann und keine anderen Fehlermeldungen.

Des Weiteren werden im späteren Verlauf in den Methoden auch *assert*-Klauseln eingesetzt. Diese nehmen nicht wie die *assume*-Klausel an, dass eine Bedingung gilt, sondern überprüfen diese an der vorgegebenen Stelle. Diese *assert*-Annotationen sollen es ermöglichen, innerhalb der Methoden die Übergabeparameter und Rückgabewerte der darin aufgerufenen Methoden weitestgehend zu überprüfen. Für einzelne Methoden war es auch möglich, diese in Hilfsmethoden auszulagern und über *requires*-, *assignable*- und *ensures*-Klauseln zu verifizieren. Diese Maßnahmen dienen dazu, die Probleme zu mindern, die durch die Seiteneffekte und die Komplexität vieler kryptografischer Funktionen entstehen. Dadurch kann die effektive Verifikation der Methodenaufrufe innerhalb der generierten Methoden weitgehend ermöglicht werden.

Als Eingangsbedingung für die `retrieveOrGenerateKey`-Methode wird erwartet, dass der Übergabeparameter `keyAlias` weder ein Nullwert noch ein Leerstring ist. Nach Ausführung der Methode sollte das Ergebnis kein Nullwert sein und den entsprechenden Rückgabotyp `SecretKey` haben. Zudem kann überprüft werden, dass der Schlüssel nach Ausführung der Methode auch im `keyStore` existiert. Dies ist besonders dann wichtig, wenn ein neuer symmetrischer Schlüssel erzeugt und im Keystore gespeichert wird. Der Aufruf der `getKey`-Methode erzeugt jedoch eine Warnung, da die Methode im Tool nicht als *pure* annotiert wurde und wurde deshalb wieder entfernt. Da die Methode `retrieveOrGenerateKey` Seiteneffekte, bezogen auf die Variable `keyStore`, hat, wird mittels des *assignable*-Statements dieser Speicherzugriff notiert. Einige der *assert*-Statements werden hier auch benötigt, damit die Nachbedingungen verifiziert werden können.

Die `encryptData`- und die `decryptData`-Methode haben eine ähnliche Spezifikation. Es muss sichergestellt sein, dass die Übergabeparameter `keyAlias` und `inputStr` beziehungsweise `encryptedData` weder Nullwerte noch Leerstrings sind und dass der korrekte Algorithmus verwendet wird. Das Einzige, das sich in der Nachbedingung verifizieren lässt, ist, dass die Rückgabe weder null noch empty ist. Interessant sind dabei zwei *assert*-Statements in den `decryptData`- und `encryptData`-Methoden, welche sicherstellen sollen, dass der `System.arraycopy`-Aufruf erfolgreich ausgeführt wurde.

```
1  //@ assert (\forallall int i; 0 <= i && i < iv.length; combined[i] == iv[i])  
    ;  
2  //@ assert (\forallall int i; iv.length <= i && i < iv.length +  
    encryptedData.length; combined[i] == encryptedData[i - iv.length]);
```

Listing 5: *assert*-Statements der `encryptData`-Methode

Hier wird der *forall*-Quantor verwendet, welcher ähnlich wie eine `for`-Schleife definiert ist. Die dritte Angabe nach dem Semikolon beschreibt dabei die Bedingung, welche gelten soll, also zum

Beispiel, dass `combined[i] == iv[i]` entspricht.

Während der Verifikation der Vorbedingung, dass der korrekte Algorithmus verwendet wird, fällt etwas sehr Interessantes auf, das später noch sehr wichtig wird. Das Tool scheint sehr empfindlich gegenüber den Angaben `final` und `static` zu sein. Wird die Variable `ALGORITHM` als `final` und `static` deklariert, so verläuft die Verifikation der Vorbedingung `//@ requires this.ALGORITHM.equals("AES/GCM/NoPadding");` problemlos. Die Angabe von `static` ist an dieser Stelle natürlich nicht sinnvoll und wurde entfernt. Dies führte jedoch dazu, dass das `requires`-Statement nicht mehr verifiziert werden kann, denn es gibt ein Problem mit dem Aufruf von `equals`. Das `requires` kann dabei durch ein verifizierbares `assert`-Statement innerhalb der Methode ersetzt werden. Wird jetzt auch noch das `final` entfernt, wobei das `final` an dieser Stelle natürlich sinnvoll wäre, so können weder das `assert`-Statement noch das `requires`-Statement erfolgreich verifiziert werden. Die Abbildung 19 zeigt dabei noch mal die beiden möglichen Fehlermeldungen, wenn die Instanzvariable weder `static` noch `final` ist.

Ein weiterer besonders spannender Verifikationsfehler ist der in Abbildung 20 dargestellte „*ArithmeticOperationRange: overflow in int sum*“-Fehler, welcher darauf hinweist, dass die Längen der Byte-Arrays noch mal überprüft werden müssen. Hier handelt es sich um einen tatsächlichen Fehler, welcher von OpenJML entdeckt wurde und eine zusätzliche `if`-Abfrage im Programmcode fordert.

Während der Verifikation tritt jedoch auch ein unerwarteter Fehler auf.

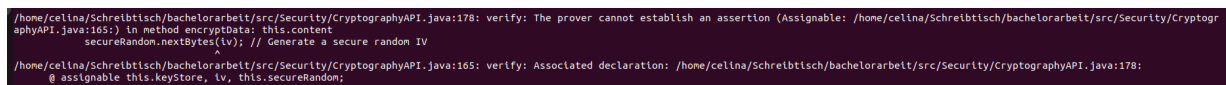


Abbildung 2.: False positive wegen des `nextBytes`-Aufrufs

In dieser Abbildung ist zu sehen, dass OpenJML einen Fehler bezüglich des *assignable*-Statements in der `encryptData`-Methode anzeigt. Der Fehler beschreibt, dass auch auf `this.content` zugegriffen und dessen Speicherwert geändert wird. Jedoch existiert das `this.content` weder in der `CryptographyAPI`-Klasse noch in der `nextBytes`-Methode der `specs/java/security/SecureRandom.jml`-Datei des OpenJML-Tools. Auch die Ergänzungen von `this.secureRandom` und `iv` können den Fehler nicht beheben. Eine Lösung des Problems besteht darin, `\everything` als *assignable*-Statement anzugeben. Das bedeutet, dass der Aufruf der Methode potenziell jeden Zustand im Programm verändern kann. Dieses Statement ist übrigens eigentlich nicht notwendig, denn jede Methode ohne *assignable*-Klausel wird implizit `\everything` als *assignable*-Statement zugewiesen. Die gewählte Annotation ist jedoch ein großer Fehler, wie sich später herausstellte, und wird im nächsten Unterkapitel genauer beleuchtet.

SecurityService-Klasse: Aufgrund der geringen Komplexität des `SecurityServices` sind nur grundlegende Spezifikationen enthalten. Die `encryptData`- und `decryptData`-Methoden benötigen hier keine Annotationen, da diese bereits im Interface `ISecurityService` definiert sind.

Logischerweise handelt es sich bei den Spezifikationen um vereinfachte Spezifikationen der `encryptData`- und `decryptData`-Methoden der `CryptographyAPI`-Klasse. Jedoch müssen hier in einer *assignable*-Klausel auch die veränderten und zugegriffenen Speicherbereiche aus der `CryptographyAPI` angegeben werden. Dies ist erneut begründet durch die implizite Annahme, dass Änderungen jeglicher Zustände im Programm, ohne Angabe einer *assignable*-Klausel, möglich sind. Diese Annahme wird im späteren Verlauf dann ein Problem, weshalb die *assignable*-Klausel an dieser Stelle relevant ist. Auf diese Thematik wird, wie bereits erwähnt,

im nächsten Unterkapitel nochmals genauer eingegangen.

7.2. Util Package

Nachdem das Security-Package vollständig ist, fehlt nun noch der `ValidationService`, welcher grundlegende Methoden zur Validierung von Eingaben enthält und in den verschiedenen Klassen aufgerufen werden kann. Die grundlegende Idee dabei ist, einfache Validierungen direkt auf der Ebene der `EBillingApp`-Klasse und tiefer gehende Validierungen in den Model-Klassen während deren Initialisierung durchzuführen.

Chatverlauf

Mittels des `Develop`-Prompts wurde ChatGPT dazu aufgefordert, einen `ValidationService` zu implementieren, welcher mehrere Methoden zur Validierung von Daten, Strings und `BigDecimal`-Werten enthält. Der `ValidationService` umfasst im Rahmen dieser Arbeit nur wenige Methoden zur Validierung und könnte logischerweise noch ergänzt werden. Dennoch wird diese Klasse im Programmcode integriert, da sie eine zentrale Rolle bei der Sicherstellung der Datenintegrität spielt und die Grundlage für zukünftige Erweiterungen und Verbesserungen bietet. Die Verwendung von `BigDecimal`-Werten anstelle von `float`- oder `double`-Werten ist wichtig, insbesondere beim Einsatz von Preisen und Geldbeträgen. *BigDecimal*-Werte ermöglichen die exakte Darstellung von komplexen Gleitkommazahlen beliebiger Größe und verhindern somit Rundungsfehler, welche insbesondere bei Geldbeträgen von Bedeutung sind.

ChatGPT erstellt auch hier problemlos die gewünschten Methoden, welche später bei der Anwendung in der `EBillingApp`-Klasse der Einfachheit halber umbenannt und invertiert wurden. Das bedeutet zum Beispiel, wenn ChatGPT eine Methode `isSmallerThan` implementiert hat, dann wird diese zu `isGreaterThan` umbenannt und ihr Ergebnis negiert, sodass `true` im Fehlerfall ausgegeben wird. Dadurch sollen spätere `if`-Abfragen vereinfacht werden, indem die Rückgaben nicht immer negiert werden müssen. Im späteren Verlauf der Konversation generiert ChatGPT für den `ValidationService` ein Interface, was jedoch wenig Sinn ergibt, da es sich ausschließlich um `static`-Methoden handelt.

Verifizierung mittels OpenJML

Da `OpenJML` keine `BigDecimal`-Werte verifizieren kann, wurden äquivalente Methoden, die stattdessen `double`-Werte verwenden, implementiert und annotiert [19]. Mit der `skipesc`-Annotation können während der Verifikation zu überspringende Methoden markiert werden. Die `isInvalidDateRange`-Methode kann leider gar nicht verifiziert werden, da die `isBefore`-Methode als nicht *pure* definiert ist und deren Verifikation zu einem *false positive*, dargestellt in Abbildung 21, führt. Mit dem Pfeil wird für gewöhnlich in der Zeile darunter die entsprechende Stelle in der Programmzeile markiert, welche das Verifikationsproblem verursacht hat. Da hier der Pfeil direkt auf die *ensures*-Klausel verweist, deutet dies zumeist auf die Beschränktheit des Tools oder einer fehlerhaften Annotation hin. Das `OpenJML`-Werkzeug kann an dieser Stelle die Nachbedingung nicht überprüfen.

7.3. Model und Storage Package

Nachdem nun die notwendigen Klassen implementiert wurden, kann mit den Model- und Storage-Klassen fortgefahren werden. Hier wäre es zunächst egal, ob mit der Customer- oder ElectricityMeter-Klasse begonnen wird. Die CustomerElectricityContract-Klasse hingegen verwendet Customer- und ElectricityMeter-Objekte und sollte deshalb erst danach mit ChatGPT generiert werden.

Chatverlauf

ElectricityMeter- und ElectricityMeterStorage-Klasse: Zuerst wird wieder mittels des [Develop](#)-Prompts die Erstellung der ElectricityMeter-Klasse gefordert. Da es sich um eine einfache Model-Klasse handelt, wird gleichzeitig auch die Erstellung der dazugehörigen ElectricityMeterStorage-Klasse mit Singleton-Design-Pattern gewünscht. Es wurden dabei eine einzigartige Stromzählernummer für die Identifikation, validierte Messzeiträume, akkurate Messwerte, der Gesamtverbrauch an Strom des letzten Jahres sowie ein SecurityService und ValidationService für die Model-Klasse gefordert.

```
1 public class ElectricityMeter {
2     private String meterNumber;
3     private LocalDate from;
4     private LocalDate to;
5     private BigDecimal oldReading;
6     private BigDecimal newReading;
7     private BigDecimal lastYearAnnualConsumption;
8     // Assume SecurityService and ValidationService are implemented
9
10    public ElectricityMeter(String meterNumber, LocalDate from, LocalDate
        to, BigDecimal oldReading, BigDecimal newReading, BigDecimal
        lastYearAnnualConsumption) {
11        // Validation logic here
12        this.meterNumber = meterNumber;
13        this.from = from;
14        this.to = to;
15        this.oldReading = oldReading;
16        this.newReading = newReading;
17        this.lastYearAnnualConsumption = lastYearAnnualConsumption;
18    }
19
20    // Getters and setters, with encryption/decryption logic for
    sensitive fields
21    // Additional methods for validation and business logic
22 }
```

Listing 6: Generierter Programmcode der ElectricityMeter-Klasse

Der abgebildete Programmcode zeigt, dass ChatGPT zwar die gewünschten Klassen generiert, jedoch sehr oberflächlich. Weder die Ver- und Entschlüsselung noch die Validierung wurden eingebaut. Außerdem werden oldReading, newReading und lastYearAnnualConsumption als BigDecimal-Werte definiert. Diese Werte sollten jedoch nach der DSGVO auf jeden Fall verschlüsselt werden und deshalb den Datentyp String haben. Darüber hinaus wird auch nicht die Einzigartigkeit der meterNumber sichergestellt, weder in der Model-Klasse noch im Storage, obwohl diese der Chatbot in seiner Antwort erwähnt. Des Weiteren gibt ChatGPT manche Imports

nicht mit an, insbesondere die Imports für zuvor generierte Klassen wie den `SecurityService`. Diese Vorgehensweise behält der Chatbot auch in allen nachfolgenden Klassen bei.

Deshalb wird im nächsten Schritt mit dem [Enhance](#)-Prompt nochmals schrittweise die Ergänzungen gewünscht. Zuerst wird nach der Ergänzung der Validierung gefragt. Hier wird die Chance genutzt und direkt definiert, dass nur überprüft werden soll, dass das Startmessdatum vor dem Endmessdatum liegt und dass der alte Messwert kleiner als der neue Messwert ist sowie dass eine Integration von Exceptions erwartet wird. Aufgrund der spezifischeren und gekürzten Aufgabe fällt es ChatGPT leichter, den gewünschten Programmcode korrekt zu generieren.

Da ChatGPT den `ValidationService` mit statischen Methoden initialisiert hat, lässt er hier automatisch den `ValidationService` als Attribut weg und ruft dessen Methoden einfach direkt im Programmcode auf. Der Aufruf des `ValidationServices` wurde nachträglich invertiert, um den invertierten Rückgabewerten der Methoden im `ValidationService` zu entsprechen. Das bedeutet also anstelle die Rückgabewerte des Methodenaufrufes immer zu negieren, wird stattdessen die Fehlermeldung ausgegeben, sobald die Methode `true` zurückgibt.

Da ChatGPT auch noch in seiner ersten Antwort erwähnt hatte, dass die `meterNumber` einzigartig sei, wird mit dem [Reflect](#)-Prompt direkt nachgefragt. Der Chatbot erkennt, dass die Überprüfung fehlt und schlägt verschiedene Lösungen textuell vor. Dann wählt er eine Lösung aus und ergänzt die `ElectricityMeterStorage`-Klasse, um die Überprüfung, ob die `meterNumber` bereits in der `HashMap` als Schlüssel existiert. Sollte dies der Fall sein, wird eine Exception geworfen. Die Lösung ist gut, da die Zählnummer meistens bereits existiert und somit im System passend definiert werden kann, aber dennoch sichergestellt wird, dass es keine zwei Stromzähler mit derselben Zählnummer im Speicher gibt.

Zu guter Letzt fehlt noch die Ver- und Entschlüsselung sensibler und personenbezogener Daten. Diese Anforderung wird wieder mit dem [Enhance](#)-Prompt gestellt. ChatGPT soll dabei als `keyAlias` den Präfix „Meter“ verwenden, kombiniert mit der `meterNumber`. Zudem soll der Chatbot selbst entscheiden, welche Daten nach der DSGVO verschlüsselt werden sollten, und Getter-Methoden implementieren, die diese Daten wieder entschlüsseln. Der generierte Code entspricht den übergebenen Vorgaben. Zudem wurde korrekt erkannt, dass der Datentyp von `oldReading`, `newReading` und `lastYearAnnualConsumption` zu `String` geändert werden muss.

Da überprüft werden sollte, warum ChatGPT gerade diese Werte verschlüsselt und nicht einfach irgendwelche Werte ausgewählt hat, wird im nächsten Prompt nochmals nachgefragt. Hierbei wird eine Kombination aus [Remember](#)- und [Reflect](#)-Prompt eingesetzt, um den Chatbot zum einen daran zu erinnern, dass die DSGVO beachtet werden muss, und zum anderen die Beweggründe hinter seiner Entscheidung erklären zu lassen.

Die Antwort von ChatGPT ist klar: Die Werte müssen auf jeden Fall verschlüsselt werden, obwohl sie nur indirekt personenbezogene Daten betreffen. Aus den Daten könnte ein Angreifer Informationen bezüglich des Verhaltens, der Arbeitszeiten und der Anzahl von Personen im Haushalt ableiten. Wohingegen die `meterNumber` und der Messzeitraum nicht zwingend bei analogen Systemen geschützt werden müssen. Bei heutigen intelligenten Messsystemen wäre hingegen die Verschlüsselung der sekundlichen Messungen auf jeden Fall notwendig [36]. Da im Rahmen dieser Arbeit mit Messzeitraum aber zwei weit auseinanderliegende Daten für analoge Stromzähler gemeint sind, ist die Verschlüsselung nicht zwingend notwendig.

Außerdem werden später weitere notwendige Setter- und Getter-Methoden selbst noch ergänzt. Da diese recht fundamental und unkompliziert sind, ist klar, dass der Chatbot diese auch noch

korrekt generieren hätte können, und sie wurden, um die Konversation einfach zu halten, außen vor gelassen.

Customer- und CustomerStorage-Klasse: Als Nächstes soll ChatGPT dann die Customer-Klasse und dessen zugehörige Storage-Klasse implementieren. Wie zuvor wird auch hier die Ver- und Entschlüsselung sowie eine eindeutige Kundennummer gefordert. In der Antwort des Chatbots ist direkt zu erkennen, dass teils die Erfahrungen und Ergebnisse der ElectricityMeter-Klasse direkt in das jetzige Ergebnis mit einfließen. Die Verschlüsselung des Namens, der E-Mail und der Wohnadresse wird diesmal direkt vorgenommen. Jedoch wird hierfür ein fester Schlüssel für alle Customer-Objekte eingesetzt. Deshalb wird nachfolgend mit dem [Feedback-Prompt](#) nachgefragt, ob es nicht vielleicht logischer wäre, eine UUID für die customerNumber zu verwenden und den keyAlias wieder aus dem Präfix „Customer“ und der customerNumber zusammenzusetzen. Hier soll anstelle der Überprüfung auf Eindeutigkeit im Storage, eine UUID verwendet werden. Eine UUID (*Universally Unique Identifier*) ist eine 128-Bit-Zahl, die üblicherweise im Hexadezimalformat dargestellt wird und durch Bindestriche in mehrere Segmente unterteilt ist. Diese Struktur macht eine Wiederholung derselben UUID äußerst unwahrscheinlich. Dies erscheint an dieser Stelle sinnvoll, da Kundennummern nicht wie die Zählernummern bereits vorliegen, sondern neu vergeben werden. Der Einsatz von UUIDs anstelle einer Überprüfung im Speichersystem führt zu einer verbesserten Leistungsfähigkeit, allerdings auf Kosten der Benutzerfreundlichkeit und Handhabbarkeit. Dieser Kompromiss wird jedoch unter den gegebenen Umständen akzeptiert.

ChatGPT ergänzt daraufhin die Customer-Klasse und implementiert zugleich auch die Getter-Methoden.

CustomerElectricityContract- und CustomerElectricityContractStorage-Klasse: Zuletzt muss noch die CustomerElectricityContract-Klasse und deren Storage-Klasse erstellt werden. Es handelt sich dabei um eine Assoziationsklasse, welche die Customer-Objekte mit den ElectricityMeter-Objekten in Verbindung bringt. Jeder Vertrag soll darüber hinaus eine einzigartige Vertragsnummer, eine monatliche gerundete, ganzzahlige Abschlagsrate, eine positive Gebrauchsrate und einen positiven Basispreis besitzen.

ChatGPT generiert hier erneut nur oberflächlichen Programmcode mit ähnlichen Problemen wie bei der ElectricityMeter-Klasse. Im ersten Schritt wird gefordert, die CustomerElectricityContractStorage-Klasse um zwei weitere Hashmaps zu erweitern. Die contractToMeterNumber-HashMap soll ermöglichen, zu einer Zählernummer den dazugehörigen Vertrag abzurufen. Die contractsToCustomerNumber-HashMap soll alle Verträge zu der angegebenen Kundennummer zurückgeben. Diese beiden Hashmaps sollen die Abfrage von Verträgen erleichtern und die Überprüfung, dass nur ein Vertrag pro Zähler existiert, ermöglichen.

Im zweiten Schritt wird gefragt, ob die Verwendung einer UUID für die Vertragsnummer nicht sinnvoll wäre. Zuletzt wird nochmals die Verschlüsselung der sensiblen und personenbezogenen Daten verlangt. ChatGPT erfüllt die Anforderungen wie in den anderen Model-Klassen auch und empfiehlt, sofern diese sensitive Daten enthalten, die Verschlüsselung der customerNumber und meterNumber in Betracht zu ziehen. Auch wenn diese im aktuellen Programmcode keine personenbezogenen Daten enthalten, wird diese Maßnahme übernommen, um das Risiko zu minimieren, dass bei einem möglichen Sicherheitsvorfall eine Verknüpfung der Zählerdaten mit den Kundendaten möglich ist.

Im späteren Chatverlauf generiert ChatGPT noch das generalisierte IStorage-Interface. Damit alle drei Storage-Klassen dieses Interface nutzen können, setzt der Chatbot generische Typen

ein. Damit das Interface auch genutzt werden kann, mussten die Methodennamen der addT- und getT-Methoden der Storage-Klassen umbenannt werden. Zudem wurden die angegebenen update- und remove-Methoden der Einfachheit halber weggelassen.

Verifizierung mittels OpenJML

Model-Klassen: Da alle drei Model-Klassen ähnlich aufgebaut sind und sich auch ihre Verifikation stark ähnelt, wird im Folgenden die Verifikation der verschiedenen Model- und Storage-Klassen zusammenfassend erläutert.

In der ElectricityMeter- und CustomerElectricityContract-Klasse werden *assume*-Statements benötigt, welche annehmen, dass die zu verschlüsselnden BigDecimal-Werte im String-Format weder null noch empty sind. Dies ist notwendig, da OpenJML BigDecimal-Werte nicht verifizieren kann. Dementsprechend fehlen die Vorbedingungen, welche sicherstellen, dass die Preconditions des SecurityServices eingehalten werden. Diese könnten zwar über Umwege, indem man die BigDecimal-Werte in Strings konvertiert, umgangen werden, jedoch würde spätestens in der später besprochenen Factory-Klasse die *assume*-Statements benötigt werden.

Hier tritt auch das zuvor angesprochene Problem aufgrund der Verwendung von encryptData und decryptData auf. Das alles als zuweisbar angegeben wurde, bedeutet auch, dass die Initialisierung von Instanzvariablen in einer jeden Methode oder einem Konstruktor, welche diese Methode aufrufen, nachträglich verändert werden könnten. Dadurch entstehen für alle Variablen im Konstruktor, welche vor dem Methoden-Aufruf initialisiert wurden, Fehler bei der Überprüfung ihrer Nachbedingungen. Dies beweist auch die Abbildung 22, welche die nicht behebbaren *Postcondition*-Fehler zeigt. Es entstehen dadurch 34 Verifikationsfehler dieser Art, welche schwerlich auf das früher definierte *assignable*-Statement in der CryptographyAPI-Klasse zurückzuführen waren. Nicht nur diese Fehler wurden durch das fehlerhafte *assignable*-Statement und ein fehlendes *assignable*-Statement im SecurityService der decryptData-Methode erzeugt, sondern auch die in Abbildung 23 gezeigten *NullField*-Fehler. Um zu garantieren, dass die Variablen nach der Ausführung weiterhin nicht null sind, wurden *assume*-Annotationen verwendet, die in jedem Konstruktor oder jeder Methode, in denen encryptData oder decryptData aufgerufen wurden, am Ende hinzugefügt werden mussten. Nach der Ergänzung des fehlenden *assignable*-Statements und Korrektur des fehlerhaften Statements, konnten diese Annotationen alle wieder entfernt werden. Dies verdeutlicht, wie empfindlich die Verifikation ist und wie geringfügige Änderungen das Ergebnis stark beeinflussen können.

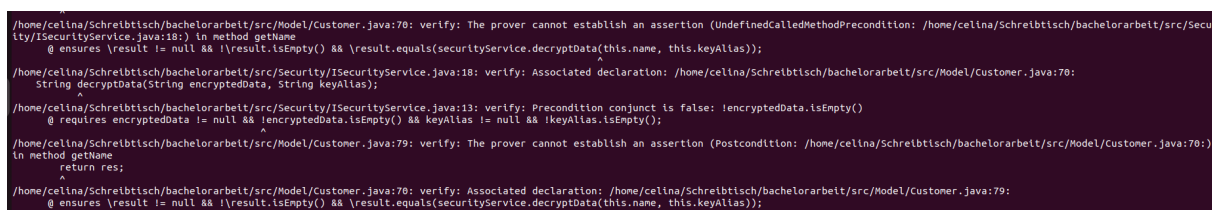
Jedoch bereitet in einer angepassten Lösung der this.content-Fehler der nextBytes-Methode weiterhin Probleme. Liest man aufmerksam die Handbücher für OpenJML, so findet man den Hinweis, dass *q.** bedeutet, dass alle Felder der Expression *q* gemeint sind. Und dies ist die Lösung für das Problem. Auf die Variable this.content kann über den Aufruf *secureRandom.content* zugegriffen werden. Für die Hilfsmethoden in der CryptographyAPI-Klasse, können dann zum Beispiel über *KeyGenerator.** oder über *GCMParameterSpec.** einfach alle Felder als zuweisbar angegeben werden. Diese Strategie wird im Laufe der Arbeit noch öfters benötigt werden.

Es gibt jetzt verschiedene Möglichkeiten, auf die Instanzvariablen der CryptographyAPI aus anderen Klassen heraus zuzugreifen, um diese in den *assignable*-Klauseln zu übernehmen. Zum einen gäbe es da den etwas korrekteren Weg, bei dem eine Klasseninstanz der CryptographyAPI benötigt wird. Diese liegt im aufgerufenen SecurityService auch vor. In diesem Fall könnte man im SecurityService auf dieser Instanz einfach die Variable folgendermaßen referenzieren: `//@ assignable cryptographyAPI.keyStore, cryptographyAPI.secureRandom.content;`

Jedoch wird diese Klausel in nachfolgende Klassen immer verschachtelter. In den Model-Klassen würde der Aufruf dann schon wie folgt aussehen: `//@ assignable ((SecurityService) securityService).cryptographyAPI.keyStore,...;` weshalb die zweite Variante gewählt wurde. Bei dieser Variante werden die benötigten Instanzvariablen als static deklariert. Somit können diese aus allen Klassen ganz einfach aufgerufen werden, sofern der Import der CryptographyAPI-Klasse vorhanden ist. Variablen einfach static zu deklarieren ist natürlich nicht empfehlenswert und sollte in einer echten Anwendung später nicht im Produktionscode übernommen werden. Für die kurzweilige Verifikation ist dies aber akzeptierbar.

Darüber hinaus kann OpenJML nicht mit UUIDs arbeiten. Die Verwendung dieser führt zu vielen false positives, wie die Abbildung 24 zeigt. Die Verifikationsfehler weisen jedoch nicht alle direkt darauf hin, dass die UUID das Problem ist. Deshalb wird für die Verifikation die UUID durch eine Zufallszahl zwischen null und 10000 ersetzt und als String formatiert, um die Verifikation dennoch durchzuführen. Auch in der ElectricityMeter-Klasse kann die einfache Zuweisung und Gleichheitsüberprüfung der meterNumber nicht verifiziert werden und nutzt denselben Workaround.

Die Getter- und Setter-Methoden sind logischerweise alle gleich aufgebaut, weshalb die Verifikation auch nahezu identisch ist. Bei den Gettern und Settern kann leider nur überprüft werden, dass das Ergebnis nicht null oder empty ist, da der Aufruf von decryptData oder encryptData in der Verifikation sonst einen *UndefinedCalledMethodPrecondition*-Fehler wirft.



```

/home/celina/Schreibttisch/bachelorarbeit/src/Model/Customer.java:70: verify: The prover cannot establish an assertion (UndefinedCalledMethodPrecondition: /home/celina/Schreibttisch/bachelorarbeit/src/Security/ISecurityService.java:18:) in method getName
    @ ensures \result != null && !\result.isEmpty() && \result.equals(securityService.decryptData(this.name, this.keyAlias));
    ^
/home/celina/Schreibttisch/bachelorarbeit/src/Security/ISecurityService.java:18: verify: Associated declaration: /home/celina/Schreibttisch/bachelorarbeit/src/Model/Customer.java:70:
    String decryptData(String encryptedData, String keyAlias);
    ^
/home/celina/Schreibttisch/bachelorarbeit/src/Security/ISecurityService.java:13: verify: Precondition conjunct is false: !encryptedData.isEmpty()
    @ requires encryptedData != null && !encryptedData.isEmpty() && keyAlias != null && !keyAlias.isEmpty();
    ^
/home/celina/Schreibttisch/bachelorarbeit/src/Model/Customer.java:79: verify: The prover cannot establish an assertion (Postcondition: /home/celina/Schreibttisch/bachelorarbeit/src/Model/Customer.java:70:)
    in method getName
        return res;
        ^
/home/celina/Schreibttisch/bachelorarbeit/src/Model/Customer.java:70: verify: Associated declaration: /home/celina/Schreibttisch/bachelorarbeit/src/Model/Customer.java:79:
    @ ensures \result != null && !\result.isEmpty() && \result.equals(securityService.decryptData(this.name, this.keyAlias));
    ^

```

Abbildung 3.: Verifikationsfehler bei Verwendung der decryptData-Methode in der Spezifikation

Diese Überprüfung funktioniert vermutlich aufgrund der Einschränkungen von OpenJML oder aufgrund der fehlenden Eigenschaften für die *pure*-Annotation der Methoden nicht. Wie bereits zuvor erwähnt, sollten möglichst nur *pure*-Methoden in Spezifikationen verwendet werden.

Der *UndefinedCalledMethodPrecondition*-Fehler lässt sich beheben, wenn in der *ensures*-Klausel zusätzlich erwartet wird, dass der keyAlias und die zu ver- oder entschlüsselnde Variable korrekt initialisiert wurden. Übrig bleibt dann nur noch der *PostCondition*-Fehler, welcher sich leider nicht ohne Weiteres beheben lässt. Der im Terminal angegebene *PostCondition*-Fehler beschreibt nur, dass das Tool nicht in der Lage ist, die angegebene Nachbedingung auf ihre Richtigkeit hin zu verifizieren. Anscheinend ist die Überprüfung im Zusammenhang mit der SecurityService- und CryptographyAPI-Klasse zu komplex.

Für die erfolgreiche Entschlüsselung oder Verschlüsselung werden *assume*-Annotationen benötigt, welche vorab garantieren, dass die zu entschlüsselnden oder verschlüsselnden Daten und der keyAlias den Vorgaben entsprechen. Diese *assume*-Klauseln werden benötigt, da in den einzelnen Methoden keine zusätzlichen if-Abfragen eingebaut sind, welche dies sicherstellen. Diese if-Abfragen werden hier nicht benötigt, da im Konstruktor der final String keyAlias festgesetzt wird und da es sich um eine Konstante handelt, keinen neuen Wert mehr zugewiesen bekommen kann. Außerdem wird später in der EBillingApp-Klasse noch vor der Objekterzeugung

überprüft, dass gültige Eingaben getätigt wurden, weshalb die Eingaben in der Model-Klasse nicht nochmals überprüft werden. Die Überprüfung der Korrektheit des `keyAlias` und der zu entschlüsselnden Daten können leider nicht als *requires*- oder *assert*-Statements angegeben werden, da später in der *CostCalculation* und im *BillingService* der Aufruf der Getter-Methoden sonst mehrere *InvariantEntrance*-, *Precondition*- und *UndefinedCalledMethodPrecondition*-Fehler wirft. Bedeutet, das Tool kann anscheinend beim Aufruf nicht überprüfen, ob die Instanzvariablen korrekt initialisiert sind (siehe Abbildung 25).

Der Aufruf der richtigen Getter-Methoden in der Verifikation erzeugt zudem Warnungen, da es sich um Methoden handelt, die nicht *pure* sind. Um dieses Problem zu umgehen, gibt es die *ghost fields* und *model methods* in OpenJML. Die *ghost fields* werden mittels `//@ public ghost datatype variablename;` erstellt. Über die *set*-Annotation kann der Wert gesetzt und später in den anderen Klassen aufgerufen werden. *Ghost fields* sind für Spezifikationszwecke gedacht und dienen zur Verfolgung zusätzlicher Zustandsinformationen, die nicht direkt durch konkrete Felder abgeleitet werden können. Die *model methods* sind auch nur für Spezifikationszwecke gedacht und dienen zur Definition von Methoden, die nicht unbedingt eine direkte Entsprechung im Java-Code haben. Sie können zudem eigene Spezifikationen wie normale Methoden besitzen, enthalten aber keinen Methodenrumpf. Die *model methods* werden in der *CustomerElectricityContract*-Klasse benötigt, da die Verwendung von *ghost fields* in der *CustomerElectricityContractStorage*-Klasse nicht ausreichten und *InvariantLeaveCaller*-Verifikationsfehler erzeugten (siehe Abbildung 26).

Außerdem enthalten die *getCustomerNumber*-, *getContractNumber*- und *getMeterNumber*-Methoden sowie die *getInstance*-Methoden der Storage-Klassen ein zusätzliches *signals_only*-Statement, welches als Bedingung *false* angibt. Das bedeutet, die Methode wirft keine Fehlermeldungen, wenn sie durchläuft. Wie die Abbildung 27 zeigt, ist dies notwendig, da die *signals_only*-Statements in späteren Klassen sonst Verifikationsfehler, beim Aufruf von Methoden ohne *signals*-Klauseln, werfen.

Bei den Settern hingegen kann noch mehr verifiziert werden. Es wird überprüft, dass die Übergabeparameter nicht *null* und nicht *empty* sind. Zudem wird mittels einer *ensures*-Klausel, wie zuvor bereits erwähnt, verifiziert, dass die überschriebenen Variablenwerte auch weiterhin korrekt sind. Des Weiteren wird hier mittels der *assignable*-Annotation definiert, auf welche Speicherbereiche zugegriffen und welche geändert werden.

Storage-Klassen: Die Verifikation der Storage-Klassen gestaltet sich unterschiedlich komplex. Die Storage-Klassen der *Customer*- und *ElectricityMeter*-Objekte verifizieren ihre *add*- und *get*-Methode ähnlich. Mittels der *also*-Annotation wird hier angegeben, dass die Spezifikationen aus dem Interface gelten und um weitere Annotationen ergänzt werden sollen. Im Interface aller Storage-Klassen wird gefordert, dass, da nur existierende und sinnvolle Objekte im Storage gespeichert werden sollen, der Übergabeparameter der *add*-Methode nicht *null* sein sollte. Damit die Verifikation fehlerfrei durchgeht, muss zudem überprüft werden, dass die entsprechende Nummer existiert, da diese später im Programmcode verwendet wird. Zudem wird erwartet, dass beim Aufruf von *add*, das neue Objekt logischerweise noch nicht im Speichersystem existiert. Leider lässt sich hier nicht überprüfen, dass nach der Methodenausführung das neue Objekt auch im Storage existiert. Hier kommt es andernfalls, wie die Abbildung 28 zeigt, zu einem *Postcondition*-Fehler, welcher sich nicht mittels *assume*-Statements beheben lässt.

Die *get*-Methode des Interfaces wird ergänzt um die Überprüfung, dass die Rückgabe entweder ein Nullwert ist oder dem Wert der entsprechenden Hashmap gleicht. Der Rückgabewert kann *null*

sein, falls die entsprechende Nummer beziehungsweise das Objekt noch nicht im Speichersystem existiert. Dies ist auch sinnvoll, da so später einfach überprüft werden kann, ob die übergebene meter-, customer- oder contractNumber existiert. Obwohl die Rückgabe null in Ordnung ist, kann OpenJML mit einem null-Rückgabewert die Methode nicht verifizieren und gibt einen *PossiblyNullReturn*-Fehler aus, weshalb hier eine zusätzliche *assume*-Klausel notwendig ist (siehe Abbildung 29).

Interessant wird die Verifikation in der CustomerElectricityContractStorage-Klasse. Neben den zuvor erwähnten Überprüfungen wird in der add-Methode zusätzlich überprüft, dass die meter- und customerNumber korrekt initialisiert sind und noch kein Vertrag für diese Zählernummer existiert. Dies ist wichtig, denn wenn bereits ein Vertrag für diesen Zähler existiert, so gilt dieser Vertrag entweder für einen anderen Kunden, welches ein Verstoß der 1-zu-n-Beziehung darstellt oder aber er gilt für denselben Kunden und sollte dann nicht nochmals hinzugefügt werden. Schließlich wird angenommen, dass jeder Kunde nur einen Vertrag für seinen Stromzähler haben kann.

Die folgende zusätzliche if-Anweisung überprüft die Bedingungen noch bevor die Werte in den Hashmaps gespeichert werden und wurde nachträglich selbstständig hinzugefügt.

```
1  if(contractToMeterNumber.get(contract.getMeterNumber()) != null) {  
2      throw new IllegalArgumentException("Contract for that meter  
    already exists");  
3  }
```

Listing 7: Hinzugefügte Überprüfung, dass es noch keine Vertrag für die meterNumber gibt

Diese Überprüfung würde für gewöhnlich besser in die noch später besprochene ContractService-Klasse passen, noch bevor das Vertragsobjekt überhaupt erzeugt wird. Jedoch kann OpenJML die Verifikation dort nicht ausführen, da der Aufruf der getContractByMeterNumber-Methode für das Programm zu komplex ist. Die Gründe hierfür werden im Unterkapitel 7.5 genauer betrachtet.

In allen add-Methoden kann leider keine Überprüfung des Zustands der Hashmaps durchgeführt werden. Dies erkennt man an dem Verifikationsfehler in Abbildung 30, welcher lediglich verrät, dass eine Nachbedingung nicht überprüft werden konnte.

Darüber hinaus bietet die CustomerElectricityContractStorage-Klasse noch zwei weitere Methoden, getContractByMeterNumber und getContractsByCustomerNumber. Die Verifikation der ersten Methode ist trivial. Die zweite Methode hingegen kann dabei eine Liste von Verträgen zurückgeben, die der Kunde eingegangen ist. Dies ist möglich, wenn der Kunde mehrere Grundstücke und damit auch mehrere Zähler besitzt. Deshalb wird hier überprüft, ob die Rückgabe einfach nur eine leere ArrayList ist, sofern noch kein Vertrag für diese Kundennummer vorliegt, oder aber die Vertragsliste mit dem erhaltenen Ergebnis für die Kundennummer übereinstimmt. Um die Verifikation dieser Methode überhaupt zu ermöglichen, wurde der Programmcode angepasst. Bei der getContractsByCustomerNumber-Methode kann die getOrDefault-Methode nicht richtig verifiziert werden, weshalb die *ensures*-Klausel einen Verifikationsfehler erzeugt (siehe Abbildung 31). Deshalb wurde die getOrDefault-Methode einfach ausgeschrieben, sodass die Methode weiterhin dieselbe Funktionalität hat, aber die getOrDefault-Methode nicht mehr benötigt.

7.4. Factory Package

Die Factory-Klassen sollen das *Factory-Design-Pattern* umsetzen und dienen zur einfachen Instanziierung von Objekten sowie zur gleichzeitigen Speicherung in den entsprechenden Storages. Das Factory-Design-Pattern ist ein Entwurfsmuster, dass die Objekt-Instanziierung kapselt und die Verantwortung für die Erstellung von Objekten an Factory-Methoden delegiert. Dies ermöglicht eine flexible und skalierbare Objekt-Instanziierung.

Da für die Erstellung eines CustomerElectricityContract-Objekts zusätzlich die Existenz und die 1-zu-1-Beziehung vom Stromzähler zum Kunden beziehungsweise zum Vertrag überprüft werden muss, wird für diese Klasse keine Factory-Klasse erstellt. Stattdessen wird eine Service-Klasse namens ContractService implementiert und im Business_logic-Package untergebracht. Die Beschreibung dieser Klasse erfolgt im nächsten Unterkapitel.

Chatverlauf

ElectricityMeterFactory und CustomerFactory: Der Chatbot wird mittels des [Develop](#)-Prompts aufgefordert, die entsprechenden und wenig komplexen Factory-Klassen für Customer- und ElectricityMeter-Objekte zu erstellen. In seiner Antwort erwähnt ChatGPT zudem, dass man die Factory-Klassen nutzen könnte, um die Validierung und Verschlüsselung der Daten noch vor der Objekt-Instanziierung durchzuführen. Dies ist auch möglich, jedoch sieht unsere Programmstruktur vor, dass die Validierung bereits in der EBillingApp-Klasse sowie die Verschlüsselung in den Model-Klassen stattfindet. Daher wird die Struktur der Anwendung beibehalten.

Zudem wird später im Chatverlauf ein Interface IFactory für die Factory-Klassen generiert. Dieses ist jedoch generalisiert und erfordert zusätzliche Überprüfungen, ob die Anzahl und der Datentyp der Übergabeparameter für die entsprechende Factory-Klasse übereinstimmen. Zum Ende der Konversation mit ChatGPT wird dieses Problem nochmals angesprochen. Mittels eines [Remember](#)-ähnlichen Prompts in Kombination mit einem [Reflect](#)-Prompt soll ChatGPT nochmals seine Gründe für die Verwendung des generischen Interfaces erklären und hinterfragen. Der GPT erkennt zwar, dass die Typsicherheit nicht unbedingt gewährleistet ist und schlägt auch die Idee der Verwendung separater Interfaces vor, implementiert jedoch erneut ein parametrisiertes generisches Interface. Daher werden die Interfaces selbst nachträglich ergänzt.

Verifizierung mittels OpenJML

Die formale Verifikation der Komponenten der Factory-Klassen gestaltet sich als verhältnismäßig unkompliziert. Ein wesentlicher Teil der Verifikation ist in den zugehörigen Interfaces angesiedelt. Analog zu den bisherigen Konstruktoren erfolgt auch hier eine Überprüfung der übergebenen Zeichenketten auf Nullwerte und Leere sowie darauf, dass das Ergebnis, der Messzeitraum und BigDecimal-Werte nicht null sind.

Die Factory-Klassen selbst benötigen jetzt nur noch *assume*-Klauseln, um bei der Objekterzeugung zu versichern, dass die übergebenen und bereits in der EBillingApp-Klasse validierten Übergabeparameter weder Nullwerte noch Leerstrings sind.

Jetzt, wo neben den Security-, Util-, Model- und Storage-Klassen auch die Verifizierung der Factory-Klassen hinzukommt, werden nochmals sechs weitere Verifikationsfehler ausgegeben.

```

/home/celina/Schreibtsch/bachelorarbeit/src/Model/Contract.java:41: verify: The prover cannot establish an assertion (InvariantEntrance: /home/celina/Programme/openjml-ubuntu-20.04-0.17.0-alpha-15/specs/
java/lang/CharSequence.jml:30:) in method createCustomer: (Caller: Factory.CustomerFactory.createCustomer(java.lang.String,java.lang.String,java.lang.String), Callee: java.lang.String.isEmpty())
@ ensures this.customerNumber != null && !this.customerNumber.isEmpty();
^
/home/celina/Programme/openjml-ubuntu-20.04-0.17.0-alpha-15/specs/java/lang/CharSequence.jml:30: verify: Associated declaration: /home/celina/Schreibtsch/bachelorarbeit/src/Model/Contract.java:41:
//RAC@ public invariant charArray != null; //FIXME - why this when there is a non_null declaration
^
/home/celina/Schreibtsch/bachelorarbeit/src/Model/Contract.java:39: verify: The prover cannot establish an assertion (InvariantEntrance: /home/celina/Programme/openjml-ubuntu-20.04-0.17.0-alpha-15/specs/
java/lang/CharSequence.jml:30:) in method createCustomer: (Caller: Factory.CustomerFactory.createCustomer(java.lang.String,java.lang.String,java.lang.String), Callee: java.lang.String.isEmpty())
@ ensures this.keyAlias != null && !this.keyAlias.isEmpty();
^
/home/celina/Programme/openjml-ubuntu-20.04-0.17.0-alpha-15/specs/java/lang/CharSequence.jml:30: verify: Associated declaration: /home/celina/Schreibtsch/bachelorarbeit/src/Model/Contract.java:39:
//RAC@ public invariant charArray != null; //FIXME - why this when there is a non_null declaration
^
/home/celina/Schreibtsch/bachelorarbeit/src/Model/ElectricityMeter.java:35: verify: The prover cannot establish an assertion (InvariantEntrance: /home/celina/Programme/openjml-ubuntu-20.04-0.17.0-alpha-1
5/specs/java/lang/CharSequence.jml:30:) in method createElectricityMeter: (Caller: Factory.ElectricityMeterFactory.createElectricityMeter(java.lang.String,java.time.LocalDate,java.time.LocalDate,java.math
.BigDecimal,java.math.BigDecimal,java.math.BigDecimal), Callee: java.lang.String.isEmpty())
@ ensures this.keyAlias != null && !this.keyAlias.isEmpty();
^
/home/celina/Programme/openjml-ubuntu-20.04-0.17.0-alpha-15/specs/java/lang/CharSequence.jml:30: verify: Associated declaration: /home/celina/Schreibtsch/bachelorarbeit/src/Model/ElectricityMeter.java:35:
//RAC@ public invariant charArray != null; //FIXME - why this when there is a non_null declaration
^
Note: /home/celina/Programme/openjml-ubuntu-20.04-0.17.0-alpha-15/specs/java/util/List.jml uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
17 warnings
41 verification failures

```

Abbildung 4.: Verifikationsfehler aufgrund fehlerhafter Annotationen in der *CharSequence.jml*-Datei

Alle diese Verifikationsfehler, wie beispielhaft in Abbildung 4 gezeigt, sind vom selben Typ und entstehen aufgrund einer fehlerhaften Annotation in der *CharSequence.jml*-Datei in Zeile 30. Die Überprüfung auf Leere der Instanzvariablen kann in der *ensures*-Klausel nicht erfolgreich überprüft werden, da in der *CharSequence*-Invariante nicht sichergestellt werden kann, dass die Zeichenkette nicht null ist. Dieser Fehler tritt auf, obwohl zuvor verifiziert wurde, dass dies nicht der Fall ist. Hier kommt die im Unterkapitel 7.1 angesprochene Bedeutung von *final* erneut ins Spiel. Entfernt man das *final* bei den Variablen *keyAlias*, *meterNumber*, *customerNumber* und *contractNumber*, so lösen sich diese Fehler auf. Das Entfernen von *final* ist jedoch nicht ganz korrekt, weshalb die obigen Fehler im Folgenden als false positives akzeptiert werden.

7.5. Business_logic Package

Zunächst wird die Erstellung der *ContractService*-Klasse und dessen Interface vorgestellt, die eine ähnliche Funktion wie die *Factory*-Klassen erfüllt. Wie zuvor erwähnt, wird diese Klasse noch vor den beiden *Factory*-Klassen im Chatverlauf generiert, jedoch erst in diesem Kapitel aus Gründen der Übersichtlichkeit betrachtet.

Anschließend wird die *CostCalculation*- und die *BillingService*-Klasse sowie deren Interfaces behandelt. Die *CostCalculation*-Klasse ist für die Berechnungen der auszustellenden Rechnungen zuständig, während der *BillingService* als Service fungiert, um diese Berechnungen zu initiieren und die Ergebnisse auszugeben.

Chatverlauf

ContractService und IContractService: Mithilfe des *Develop*-Prompts wird ChatGPT angewiesen, den *ContractService* zu erstellen. Es soll geprüft werden, ob Stromzähler und Kunde bereits im System existieren, bevor eine Objekt-Instanziierung und Speicherung in der *Storage*-Klasse erfolgt. Der von ChatGPT generierte Programmcode erfüllt die gestellten Anforderungen. Die Methode *createAndAssignContract* sollte jedoch zusätzlich prüfen, dass für den betreffenden Stromzähler noch kein Vertrag existiert. Diese Überprüfung wurde durch eine zusätzliche *if*-Anweisung im Code ergänzt.

Verifizierung mittels OpenJML

ContractService und IContractService: Überraschenderweise führt diese Überprüfung zu einer Blockierung des Verifikationstools. Dies wird jedoch nicht mit einer Fehlermeldung angezeigt, ganz im Gegenteil: Einige *CharSequence*-Fehler lösen sich auf, und es scheint zunächst so, als hätte diese if-Anweisung einen positiven Effekt. Das ist jedoch nicht der Fall, obwohl die Verifikation ohne Probleme durchläuft. Dass diese if-Anweisung zu einer fehlerhaften Verifikation führt, lässt sich leicht überprüfen.

```
1  /*@
2    @ ensures \result == 6;
3    @*/
4    public int createAndAssignContract(String customerNumber, String
meterNumber, BigDecimal monthlyRate, BigDecimal basePrice, BigDecimal
usageRate) {
5        //@ assume contractStorage.getContractByMeterNumber(meterNumber)
== null;
6        if (contractStorage.getContractByMeterNumber(meterNumber) != null
) {
7            throw new IllegalArgumentException("Contract for that meter
already exists");
8        }
9
10       return 5;
11    }
```

Listing 8: Strategie zur Überprüfung, ob das OpenJML-Tool blockiert ist

Wie im oberen Codeausschnitt dargestellt, ändert man den Rückgabebetyp zur Überprüfung auf `int` und gibt zum Beispiel die Zahl fünf zurück. In der *ensures*-Klausel überprüft man hingegen, dass das Ergebnis gleich sechs ist. Dies sollte auf jeden Fall einen Fehler bei der Verifikation erzeugen, da das Programm offensichtlich nicht formal korrekt ist. In diesem Fall wird dieser Fehler aber nicht erkannt. Dies ist ein eindeutiger Indikator, dass diese zusätzliche if-Anweisung einen blockierenden Effekt auf das Verifikationstool haben muss. Aus diesem Grund wurde die Überprüfung für den Stromzähler in die *CustomerElectricityContractStorage*-Klasse in die *add*-Methode verschoben. Hier kann die formale Korrektheit ohne Probleme überprüft werden. Aufgrund der Angabe von *final* in den Model-Klassen, treten bei der Verifikation jetzt noch mal acht weitere *InvariantEntrance*-Verifikationsfehler auf. Somit gibt es zum aktuellen Zeitpunkt sechs Warnungen und sechzehn Verifikationsfehler.

Chatverlauf

CostCalculation: Im nächsten Schritt wird die *CostCalculation*-Klasse generiert. Diese Klasse ist besonders interessant, denn hier kommen die verschiedenen zuvor generierten Model-Klassen endlich zum Einsatz und ChatGPT muss auf seine vorangegangenen entwickelten Lösungen zurückgreifen. Die Klasse bietet viele Methoden, die einfache Berechnungen durchführen, wie zum Beispiel die Berechnung des Netto- und Brutto-Preises, der Mehrwertsteuer, der neuen Abschlagsrate, des jährlichen Stromverbrauchs und der Zahlungsdifferenz. Das Ergebnis von ChatGPT ist anfänglich teils zufriedenstellend. Die Methoden sind zwar inhaltlich korrekt, jedoch verwendet der Chatbot Übergabeparameter, welche nicht im UML angegeben sind, und vergisst, dass bestimmte Daten noch entschlüsselt werden müssen. Über den [Remember](#)-Prompt soll

ChatGPT daran erinnert werden und den Code noch mal anpassen. Zudem wird als Hilfestellung erwähnt, dass die get-Methoden der Model-Klasse genutzt werden können. Die Korrektur ist angemessen, auch wenn die calculateNet- und consumedElectricity-Methode weiterhin nicht ganz korrekte Übergabeparameter haben. Dennoch handelt es sich hierbei um eine korrekte und nachvollziehbare Lösung, welche so übernommen wird. Jedoch ist die public-Sichtbarkeit der Methoden noch falsch, weshalb ChatGPT auch das noch mal korrigieren soll. Die Sichtbarkeit package-private hilft dabei die Kapselung und Integrität der CostCalculation zu wahren und ist von daher sinnvoll.

Im späteren Chatverlauf wird für die CostCalculation-Klasse jedoch ein Interface generiert. Dies macht wegen den statischen Methoden, wie schon bei dem ValidationService, und der Methoden-Sichtbarkeit keinen Sinn und wird deshalb nicht übernommen. Die Antwort für ChatGPTs Gründe bezieht sich auf die gewünschten Auswirkungen der Interfaces wie Modularität, Skalierbarkeit, Sicherheit und Wartbarkeit des Programms. Jedoch beachtet der Chatbot seine Implementierungsdetails nicht.

BillingService und IBillingService: Für den BillingService, der nach der CostCalculation-Klasse implementiert wurde, wird kein Interface generiert. Dies wäre an dieser Stelle jedoch sinnvoll gewesen, da die Methoden weder statisch noch package-private sind. Das Interface wurde daher nachträglich erstellt. Der BillingService hat zwei Funktionen: Eine Methode calculateBillingDetails, die die CostCalculation zur Berechnung relevanter Ergebnisse nutzt, und eine private Methode, um diese Ergebnisse im Terminal auszugeben. ChatGPT generiert den BillingService gemäß den Anforderungen, integriert jedoch einige der print-Ausgaben in die Methode calculateBillingDetails. Dies steht im Widerspruch zum *Single Responsibility Principle (SRP)*, einem Kernprinzip der Softwareentwicklung, das vorsieht, dass jede Methode beziehungsweise Klasse eine eindeutige Funktionalität haben sollte. Um diesem Prinzip, wie in den bisherigen Klassen auch, gerecht zu werden, sollten diese Ausgaben in die Methode printBillDocumentToTerminal verschoben werden, was auch eine Anpassung der Übergabeparameter dieser Methode erfordert.

Verifizierung mittels OpenJML

CostCalculation: Wie auch im ValidationService kann OpenJML Methoden, die mit BigDecimal-Werten arbeiten, nicht verifizieren. Stattdessen wurden Ersatzmethoden ergänzt, die double-Werte verwenden. Die geringe Komplexität der Methoden macht ihre Annotationen recht einfach.

BillingService und IBillingService: Die Verifikation der calculateBillingDetails-Methode findet vollständig im IBillingService-Interface statt. Hier wird erneut mit der Hilfe von ghost fields überprüft, dass die Messwerte des angegebenen Zählers weiterhin korrekt sind. Dies ist natürlich notwendig, da die Messwerte zur Berechnung der Abrechnung verwendet werden.

Die printBillDocumentToTerminal-Methode kann hingegen nicht immer ohne einen Bug verifiziert werden, da OpenJML Schwierigkeiten mit der Anzahl der print-Ausgaben hat. Dies ist erkenntlich, sobald man als Eingangsbedingung die Übergabeparameter überprüfen möchte, selbst wenn es sich dabei um double-Werte handelt. Die Verifikation löst dabei manchmal wie in Abbildung 32 zusehen, einen *OutOfMemoryError* mit der Meldung „A catastrophic JML internal error occurred“ aus, was auf eine Überlastung des Verifikationsprogramms hindeutet.

7.6. GUI Package

Schlussendlich kann die einfache, terminalbasierte GUI generiert werden. Nun liegen alle notwendigen Klassen vor, sodass ChatGPT die EBillingApp-Klasse problemlos erstellen können sollte. Zudem wird hier auch die main-Methode integriert, damit sich das Programm starten lässt.

Chatverlauf

Wie bei allen anderen Klassen auch, soll ChatGPT die EBillingApp-Klasse erstellen. Diese soll im Rahmen dieser Arbeit lediglich ermöglichen, neue Kunden, Zähler und Verträge zu erstellen sowie die Rechnung eines spezifischen Kunden zu berechnen und auszugeben. Zudem soll das Programm in einer Schleife laufen, sodass es selbst beendet werden kann. ChatGPT erkennt direkt, dass die Factory- und Service-Klassen benötigt werden. Zudem erstellt der Chatbot weitere sinnvolle, separate Methoden, die weder im Prompt noch in der PlantUML-Beschreibung vorgegeben waren. Die run-Methode setzt dabei die gewünschte Schleife und die Interaktion mit dem Benutzer um. Die anderen notwendigen Methoden werden jedoch nicht implementiert, sondern nur vorgegeben. Dies liegt wahrscheinlich an der Größe und Komplexität der Aufgabe, sodass der Chatbot diese nicht vollständig implementiert. Das ist natürlich nicht zufriedenstellend, weshalb eine Nachbesserung gefordert wird. Zudem wird erwähnt, dass auch eine Validierung in der EBillingApp-Klasse durchgeführt werden soll. Die Lösung ist überraschenderweise sehr gut. Selbst die Factory-Klassen werden korrekt eingesetzt, obwohl ChatGPT die create-Methode aus einem nicht verwendeten Interface einsetzt. Da dies dem Chatbot nicht mitgeteilt wurde, ist das an dieser Stelle egal und wird später selbst ergänzt. Auch die Validierung wird wie gewünscht integriert. Jedoch wurden nur die Methoden createCustomer und createElectricityMeter implementiert, mit der Begründung, dass der Kürze halber nicht alle Methoden generiert wurden. Da die createContract-Methode tatsächlich ziemlich ähnlich ist, wird nur noch die Erstellung der calculateAndPrintBilling-Methode gefordert.

Hier ist erkenntlich, dass der Chatbot nicht ganz zufrieden mit dem aktuellen Programmcode ist, denn er verwendet den ContractService anstelle der eigentlich notwendigen Storage-Klasse und denkt sich einen eigenen MeterService dazu aus. Nachträglich betrachtet würde es tatsächlich Sinn ergeben, Methoden wie getContractsByCustomerNumber besser im ContractService anzusiedeln. Dies wäre leicht umzusetzen, indem die Methoden aus der Storage-Klasse in die ContractService-Klasse verschoben werden und stattdessen Getter für die contractsToCustomerNumber- und contractToMeterNumber-Hashmaps integriert werden. Dennoch beeinträchtigt diese vielleicht nicht ganz optimale Umsetzung jetzt nicht das Ergebnis und wird deshalb so gelassen. Daran kann man gut erkennen, dass ChatGPT dabei helfen kann, Logik- oder Strukturfehler aufzudecken. Dies wird oft ersichtlich, insbesondere bei einfacherem Programmcode, wenn der Chatbot Schwierigkeiten hat, den Code zu generieren und dazu neigt, die Lösung jedes Mal anders als vorgegeben zu implementieren.

ChatGPT vereinfacht zudem die Lösung, indem angenommen wird, dass es nur eine 1-zu-1-Beziehung zwischen Kunde und Vertrag geben kann. Dies ist in dieser Anwendung nicht der Fall. Jedoch wird in diesem Chatverlauf das erneute Nachfragen erspart und auf einen Lösungsansatz aus einem anderen Chatverlauf (siehe Abbildung 33) zurückgegriffen.³

³Siehe USB-Stick conversations.pdf unter „Main_Chat2“

Verifizierung mittels OpenJML

Aufgrund der vielen print-Ausgaben konnte in der EBillingApp-Klasse nicht besonders viel, neben ein paar grundsätzlichen Überprüfungen, verifiziert werden. Interessant war hierbei die Notwendigkeit der *assume*-Klausel: `//@ assume \invariant_for(System.out);`. Denn der Aufruf von `System.out` hat intern im OpenJML-Programm zu einem *InvariantLeaveCaller*-Fehler geführt.

```
/home/celina/Schreibtsch/bachelorarbeit/src/GUI/EBillingApp.java:141: verify: The prover cannot establish an assertion (InvariantLeaveCaller: /home/celina/Programme/openjml-ubuntu-20.04-0.17.0-alpha-15/specs/java/io/PrintStream.jml:35) in method createElectricityMeter: (caller: GUI.EBillingApp.createElectricityMeter(), callee: java.io.PrintStream.println(java.lang.String))
System.out.println("Electricity meter created successfully with meter number: " + meter.getMeterNumber());
^
/home/celina/Programme/openjml-ubuntu-20.04-0.17.0-alpha-15/specs/java/io/PrintStream.jml:35: verify: Associated declaration: /home/celina/Schreibtsch/bachelorarbeit/src/GUI/EBillingApp.java:141:
//RAC@ public static invariant \invariant_for(eot) && eot.length() > 0;
```

Abbildung 5.: Verifikationsfehler, sofern das *assume*-Statement für die `System.out`-Invariante fehlt

Dieser Verifikationsfehler hängt mit der Invariante in der Zeile 36 und 42 der *specs/java/io/PrintStream.jml*-Datei zusammen. Diese Meldung konnte nur dadurch behoben werden, dass angenommen wird, dass die Invariante für `System.out` erfüllt ist. Es ist überraschend, dass dieser Fehler nur in der EBillingApp-Klasse, aber nicht in der BillingService-Klasse aufgetreten ist.

Weitere interessante Verifikationsfehler sind die *InvariantEntrance*-Fehler, die diesmal die `isEmpty`-Methode des `ValidationService` als Ursache angeben. Dies ist jedoch nicht der eigentliche Grund. Wie bereits in den vorherigen Unterkapiteln erlernt wurde, liegt das an dem Aufruf der `promptForInput`-Methode, die keine *assignable*-Annotation enthält. Hier tritt ein ähnliches Problem wie in der `CryptographyAPI`-Klasse auf. Die Variablen `istream.streamState` und `outputText` werden anscheinend durch den Aufruf von `System.out.print` und `new Scanner(System.in)` verändert. Auf diese kann natürlich nicht direkt zugegriffen werden, weshalb hier `//@ assignable System.in.*, System.out.*;` angegeben werden muss.

```
/home/celina/Schreibtsch/bachelorarbeit/src/GUI/EBillingApp.java:122: verify: The prover cannot establish an assertion (InvariantEntrance: /home/celina/Programme/openjml-ubuntu-20.04-0.17.0-alpha-15/specs/java/lang/CharSequence.jml:30) in method createElectricityMeter: (caller: GUI.EBillingApp.createElectricityMeter(), callee: GUI.ValidationService.isEmpty(java.lang.String))
boolean invalid = ValidationService.isEmpty(meterNumber) || ValidationService.isNegative(oldReading) || ValidationService.isNegative(newReading);
^
/home/celina/Programme/openjml-ubuntu-20.04-0.17.0-alpha-15/specs/java/lang/CharSequence.jml:30: verify: Associated declaration: /home/celina/Schreibtsch/bachelorarbeit/src/GUI/EBillingApp.java:122:
//RAC@ public invariant charArray != null; //FIXME - why this when there is a non-null declaration
^
/home/celina/Schreibtsch/bachelorarbeit/src/GUI/EBillingApp.java:281: verify: The prover cannot establish an assertion (UndefinedCalledMethodPrecondition: /home/celina/Programme/openjml-ubuntu-20.04-0.17.0-alpha-15/specs/java/lang/String.jml:333) in method promptForInput
Note: Call stack
/home/celina/Programme/openjml-ubuntu-20.04-0.17.0-alpha-15/specs/java/lang/String.jml:985: java.lang.String.charAt
/home/celina/Schreibtsch/bachelorarbeit/src/GUI/EBillingApp.java:63: java.lang.String.trim
/home/celina/Programme/openjml-ubuntu-20.04-0.17.0-alpha-15/specs/java/lang/String.jml:333: verify: Associated declaration: /home/celina/Schreibtsch/bachelorarbeit/src/GUI/EBillingApp.java:281:
public /*@ pure @*/ char charAt(int index)
^
/home/celina/Programme/openjml-ubuntu-20.04-0.17.0-alpha-15/specs/java/lang/CharSequence.jml:79: verify: Precondition conjunct is false: 0 <= index < charArray.length
@
requires 0 <= index < charArray.length;
```

Abbildung 6.: Verifikationsfehler, wegen dem `trim`-Methodenaufruf und fehlender *assignable*-Annotation in der `promptForInput`-Methode

Jetzt bleiben nur noch drei Fehler übrig, alle entstehen aufgrund des Aufrufs von `input.trim()`; Diese *UndefinedCalledMethodPrecondition*- und *Precondition*-Fehler können leider nicht behoben werden und gleichen der Situation beim `getOrDefault`-Aufruf in der `Storage`-Klasse. Deshalb wurde die `trim`-Methode einfach selbst programmiert und durch zwei `while`-Schleifen ersetzt. Nun kann das Programm ohne Probleme vollständig verifiziert werden und es gibt nur noch insgesamt sieben Warnungen und sechzehn Verifikationsfehler.

8. Diskussion der Ergebnisse

Abschließend werden die Ergebnisse jetzt noch mal genauer betrachtet und bewertet. Zudem werden andere Studien herangezogen, um die eigenen Erkenntnisse zu vergleichen und die Ergebnisse im Kontext der bestehenden Forschung zu interpretieren. Auf diese Weise soll die Relevanz und Validität der aktuellen Arbeit unterstrichen und reflektiert werden.

Bewertung der Methodik

Die Methodik für diese Arbeit sah ein systematisches Vorgehen vor. Hierfür musste zuerst selbstständig die Architektur der Stromabrechnungsanwendung sowie die zu verwendenden Software-Design-Patterns geplant werden. Danach wurden explorativ Prompts entwickelt, welche die Reproduzierbarkeit und Optimierung der Ergebnisse erzielen sollten. Erst dann konnte der Chatverlauf mit ChatGPT gestartet und die Ergebnisse daraufhin mit OpenJML/ESC und manuell verifiziert und überprüft werden. Passend zu den Aussagen von Yetiştiren et al. im Jahre 2023, hat der Einsatz einer PlantUML-Diagrammbeschreibung die Arbeit mit dem Chatbot erleichtert, da darauf referenziert werden konnte [37]. Ein UML-Klassendiagramm liegt jedoch nicht immer vor und gehört nicht in jedem Unternehmen zum standardisierten Vorgehen. Jedoch lässt sich davon ausgehen, dass auch andere Diagrammarten, sofern sie gut vom Modell verstanden werden, verwendet werden können. Zusätzlich hat die Erstellung des UML-Klassendiagramms beziehungsweise des PlantUML-Codes geholfen, vorab bereits eigene strukturelle und konzeptionelle Fehler zu beseitigen und sich zu verdeutlichen, wie das erwartete Ergebnis von ChatGPT auszusehen hat.

Die Verwendung der betitelten Prompts erleichterte zudem die Arbeit mit ChatGPT, da sie leicht um die entsprechenden Vorgaben wie Attribute oder Funktionalitäten ergänzt werden konnten, und verbesserten die generierten Ergebnisse.

Zudem bietet die Verifikation mittels OpenJML/ESC die beste Mischung aus Effizienz und Fehlerabdeckung, indem sie mehr Fehler als traditionelle Typsysteme erfasst, jedoch mit wesentlich geringeren Kosten im Vergleich zur vollständigen funktionalen Programmverifikation [20].

Jedoch gibt es auch einige kritische Punkte zu beachten. Die Qualität der Prompts ist entscheidend für die generierten Ergebnisse. Fehlerhafte oder ungenaue Prompts können zu suboptimalen Ergebnissen führen und unterstreichen die Notwendigkeit der sorgfältigen Überprüfung und Anpassung der Prompts. Der vorliegende Nicht-Determinismus bei den Antworten von ChatGPT kann das Ergebnis und die Zuverlässigkeit der generierten Lösungen beeinflussen. Dies sollte berücksichtigt werden, um die Wissenschaftlichkeit dieser Arbeit sicherzustellen, wie später im Abschnitt zu Nicht-Determinismus genauer erläutert wird. Manche Bestandteile der Anwendung konnten nicht vollständig durch OpenJML/ESC verifiziert werden. In solchen Fällen war eine manuelle Analyse unerlässlich, um sicherzustellen, dass alle sicherheitsrelevanten und funktionalen Anforderungen erfüllt sind. Daher sollte die Kombination aus automatisierter Verifikation und manueller Überprüfung sorgfältig abgewogen werden.

Effizienz und Herausforderungen der Codegenerierung

Insgesamt wurden für diese Arbeit 407 Zeilen Programmcode von ChatGPT generiert und 80 weitere Zeilen ergänzt. Jeder vollständige Chatverlauf erforderte dabei etwa acht Stunden Arbeitsaufwand, einschließlich des Lesens aller Antworten und der manuellen Überprüfung des generierten Codes. Dies verdeutlicht, dass ein kleines Programm, das auch Verschlüsselung beinhaltet, innerhalb eines Arbeitstages generiert werden kann. Es ist jedoch anzumerken, dass es sich hierbei um ein relativ einfaches und wenig komplexes Programm handelt, das ein Softwareentwickler in weniger als acht Stunden hätte implementieren können.

Ein großer Vorteil der Verwendung von ChatGPT ist jedoch die Möglichkeit, strukturelle oder logische Unklarheiten aufzudecken. Zudem können zusätzliche Gedankengänge zum generierten Programmcode und zu beachtende Hinweise helfen, auf gewisse Punkte aufmerksam zu werden, welche zuvor nicht bedacht wurden. Jedoch erwies sich das Verfolgen und Führen eines Chatverlaufs mit ChatGPT auf Dauer als mental anstrengend und fordernd. Dies bestätigen auch die Untersuchungen von Sobania et al., 2023 [38]. Diese Erkenntnis erfordert eine sorgfältige Planung und regelmäßige Pausen, um die Qualität der Interaktionen aufrechtzuerhalten.

Herausforderungen für ChatGPT

Außerdem hatte ChatGPT auch vereinzelt Probleme mit spezifischen Situationen. Zum Beispiel hatte der Chatbot Schwierigkeiten, die korrekte Sichtbarkeit der Methoden festzulegen, was durch spezifische Prompts wie den **Remember**- und **Feedback**-Prompt versucht wurde zu korrigieren. Diese Prompts erinnerten den Chatbot an die ursprünglichen Spezifikationen und forderten ihn auf, die Sichtbarkeit anzupassen. Weiterhin bleibt es eine Herausforderung für ChatGPT, bereits implementierte Ergebnisse zu berücksichtigen und korrekt zu integrieren. Hier musste auch nachgeholfen werden. Auffällig war des Weiteren, dass Imports und insbesondere Package-Vorgaben zumeist nicht vollständig vom Chatbot angegeben wurden. Diese mussten selbstständig ergänzt werden. Auch die Generierung der Interfaces bereiteten ChatGPT Probleme. Dem Chatbot war nicht klar, dass für statische Klassen mit Methoden der package-private- oder private-Sichtbarkeit ein Interface weniger sinnvoll ist. Generell entstand der Eindruck, dass ChatGPT willkürlich Klassen wählte, für die ein Interface generiert werden sollte.

Zuverlässigkeit und Optimierung der Codegenerierung

Eine Untersuchung von Yetiştiren et al. dokumentiert, dass ChatGPT für 153 von insgesamt 164 Probleme validen Code erzeugen konnte, wobei 65,2% der Probleme vollständig korrekt gelöst wurden [37]. Diese Ergebnisse werden durch weitere Studien bestätigt, die die Zuverlässigkeit und Fähigkeit von Large Language Models zur Codegenerierung untersuchen [39, 40]. Um die Ergebnisse zu verbessern, ist es besonders wichtig, klare Vorgaben und Erinnerungen in den Prompts zu integrieren. Üblicherweise wird kein vollständig neues Programm implementiert, sondern auf bestehendem Code aufgebaut. Je komplexer der zu generierende Code, desto kleiner und detaillierter müssen die Teilaufgaben definiert werden, um optimale Ergebnisse zu erzielen. Es muss ebenfalls klar sein, was mit dem zu generierenden Code erreicht werden und welche Funktionalitäten dieser haben soll. Zusätzliche Vorgaben, wie der Wunsch nach gut wartbarem Code, können die generierten Ergebnisse beeinflussen [41]. Dies bestätigt auch die Studie von Yetiştiren et al., die feststellten, dass

From this, we argue that the lack of a proper explanation of the problems yields lower validity and correctness scores. [...]

When using code generation tools, it is crucial to provide clear and accurate problem descriptions to obtain valid and correct code. Whenever possible, programmers

should include a comprehensive explanation of the problem, along with sample unit tests in the form of docstrings, comments, or other forms of documentation during the solution generation process. [37]

Nicht-Determinismus der Ergebnisse

Des Weiteren weisen Ouyang et al. auf eine Korrelation zwischen der Länge der Prompts und der Korrektheit/Nicht-Determinismus des generierten Codes hin. Es ist also wichtig, ein Gleichgewicht zwischen der Informationsmenge und dem Nicht-Determinismus des Codes zu finden.

Auffällig war auch, dass die Qualität der Ergebnisse schwankte. An manchen Tagen wurde guter Code generiert, während an anderen Tagen die Qualität und Struktur der Antworten stark variierten. Der Nicht-Determinismus von generierten Ergebnissen von ChatGPT sollte unbedingt berücksichtigt werden. Dies gilt sowohl für Entwickler als auch für die Ergebnisse dieser Bachelorarbeit. Ouyang et al. untersuchten den Nicht-Determinismus von ChatGPT 3.5 und 4.0 und stellten in ihren Ergebnissen zudem fest, dass der Nicht-Determinismus bei ChatGPT 4.0 noch etwas stärker ist [27]. Die Autoren weisen explizit darauf hin, dass

If the code generated from ChatGPT is unstable, it can lead to non-reproducible results and unreliable conclusions. Therefore, researchers should carefully consider the limitations of ChatGPT when designing experiments and interpreting results.[27]

Die Zufälligkeit kann reduziert werden, wenn mehrere Ergebnisse beziehungsweise Chatverläufe betrachtet werden [27]. Wie bereits erwähnt, wurden mehrere Chatverläufe mit ähnlichen Prompts geführt, um den Nicht-Determinismus zu reduzieren. Die Ergebnisse von Chatverläufen, die zu Ende geführt wurden, waren recht ähnlich. Dennoch sollte bei den vorangegangenen Ergebnissen der Nicht-Determinismus berücksichtigt werden.

Bewertung der Qualität des generierten Codes

Zusammenfassend lässt sich sagen, dass der von ChatGPT generierte Code größtenteils den Spezifikationen entsprach und nur geringfügige Anpassungen notwendig waren. Die Implementierung der Verschlüsselung war korrekt, und es wurden viele nützliche Anregungen zur Verbesserung der Sicherheit der Anwendung geliefert. Die Fähigkeit von ChatGPT, validen Code zu generieren, wird auch in verschiedenen Studien bestätigt, wobei auch der Schwierigkeitsgrad des zu generierenden Codes entscheidend ist [39, 40, 42]. Der im Rahmen dieser Arbeit generierte Programmcode lässt sich eher in der Kategorie einfach bis medium bezüglich der Schwierigkeit einordnen.

Es gab geringfügige Fehler in der übergebenen PlantUML-Beschreibung und es wurde vergessen in dem einen Prompt für die Generierung des SecurityServices zu erwähnen, dass das Singleton-Design-Pattern verwendet werden soll. Dennoch entsprach das von ChatGPT generierte Ergebnis den Vorgaben.

Bewertung der Sicherheit des generierten Codes

Wie bereits erwähnt gilt weiterhin, dass der generierte Programmcode immer auch manuell überprüft und getestet werden sollte, um Schwachstellen möglichst zu minimieren. Zusätzlich empfiehlt sogar der Chatbot selbst, aktuelle Sicherheitsstandards noch mal im Internet nachzulesen, bevor einfach seine Lösung übernommen wird. Die Ergebnisse variieren diesbezüglich jedoch je nach Studie. Der von ChatGPT im Rahmen dieser Arbeit generierte Programmcode erfüllt die Sicherheitsstandards und Empfehlungen des BSI [30]. Die Studie von Yetişiren et al. [37] und die von Khan et al. 2023 [41] bestätigen, dass ChatGPT in der Lage ist, sicheren Programmcode zu generieren. Hier wurden jedoch zumeist leichtere Python-Programme

implementiert, welche nicht explizit darauf ausgerichtet waren, potenzielle Schwachstellen zu integrieren. Die Untersuchungen von Khoury et al. im Jahr 2023 zielten dabei genau auf die Generierung solcher Programme in unterschiedlichen Programmiersprachen ab. Die Untersuchungen von Khoury et al. zeigen, dass ChatGPT nur für fünf der 22 Programme sicheren Code generieren konnte, obwohl durch zusätzliche Hinweise sieben weitere Programme verbessert werden konnten. Demnach generiert ChatGPT nicht immer sicheren Programmcode, kann aber Schwachstellen selbst gut erkennen. Jedoch wurde im Rahmen ihrer Arbeit ChatGPT 3.5 verwendet, und manche der Fehler basierten auf einem veralteten Bibliothekeneinsatz. Zudem wurden die Prompts oft aus Sicht eines sicherheitstechnisch unwissenden Programmierers formuliert und nicht direkt vorgegeben, sicheren Programmcode zu generieren [43]. Wie bereits in dieser Arbeit erwähnt wurde, ist das explizite Erwähnen der Relevanz der Sicherheit und einer passenden Rollenbeschreibung mitunter entscheidend, sicheren Programmcode generiert zu bekommen.

Einsatz von OpenJML und Verifikationsprobleme

Eine weitere Herausforderung war die Arbeit mit OpenJML. Während der Verifikation traten einige false positives auf, was die Verifikation erschwerte. Es war oft schwierig zu erkennen, welche Verifikationsfehler tatsächliche Fehler waren und welche false positives darstellten. Besonders häufig traten *Postcondition*- und *InvariantEntrance*-Fehler auf. OpenJML erwies sich als sehr empfindlich gegenüber Schlüsselwörtern wie *static* oder *final*. Zudem war die Angabe, auf welche Speicherbereiche genau zugegriffen wird, entscheidend für die Verifikation. Darüber hinaus ergab sich, dass mit der Angabe $q.*$, alle Felder der Expression q als zuweisbar und veränderbar angegeben werden können [44]. Dies ist besonders dann relevant, wenn bei der Verifikation eine Variable als zuweisbar angegeben wird, auf die man selbst keinen Zugriff hat. Des Weiteren waren manche Bedingungen aufgrund ihrer Komplexität schlichtweg nicht vom Programm verifizierbar. Dazu gehörte auch der auffällige *OutOfMemoryError* in der *BillingService*-Klasse, aufgrund zu vieler *print*-Ausgaben in der *printBillDocumentToTerminal*-Methode. Um diese Herausforderung zu überwinden, half zumeist die Verwendung von *assume*-Statements, um bestimmte Annahmen explizit zu machen und somit die Überprüfung der Spezifikationen zu erleichtern. Auch *assert*-Annotationen halfen zusätzliche Bedingungen innerhalb von Methoden zu überprüfen. Die Verifikation erforderte viel Zeit und Konzentration, da die formale Verifikation der *EBillingApp*-Klasse auch die Verifikation aller direkt oder indirekt verwendeten Klassen verlangte, wie der *Service*-, *Factory*-, *Storage*-, *Model*- und *Business_logic*-Klassen. Eine vollständige Verifikation dauerte durchschnittlich etwa zwanzig Minuten. OpenJML bietet zwar die Möglichkeit, spezifische Methoden oder Klassen anzugeben, die verifiziert werden sollen, jedoch kann dies dazu führen, dass potenzielle Fehler übersehen werden (false negatives), die bei einer vollständigen Verifikation erkannt worden wären. Eine andere Möglichkeit war die Erstellung von Hilfsklassen beziehungsweise Dateien, welche nur die vereinfachten Programmteile enthielten, welche die Verifikationsfehler verursachten und somit die Verifikation beschleunigte. Dennoch lautet die Devise: viel Zeit, Konzentration und Ausdauer bei der Verifikation aufzubringen.

Leider konnte mit OpenJML nicht vollständig untersucht werden, ob die kryptografischen Methoden mit korrekten Parametern aufgerufen wurden. Hier empfiehlt sich der Einsatz von Stubs, um Methoden durch einfache und reine Methoden zu ersetzen. Dies ist sinnvoll, da die genaue Implementierung der JCA- und JCE-Methoden für die Verifikation nicht relevant ist, sondern nur, dass diese mit korrekten Parametern aufgerufen werden. Dieser Ansatz könnte in zukünftigen Arbeiten weiter untersucht werden.

9. Ausblick und Perspektiven

Diese Arbeit überprüfte die Fähigkeit von ChatGPT, sicheren und qualitativ hochwertigen Programmcode zu generieren. Die Prompts wurden schrittweise angepasst, um benutzerfreundlich, übersichtlich und kurz zu sein. Der Einfluss auf das Ergebnis der in dieser Arbeit vorgestellten Prompts, wurde jedoch nicht systematisch untersucht und könnte ein Thema für weiterführende Forschungen sein. Ähnlich den Studien von White et al. [23] und Liu et al. [45] könnten zukünftige Forschungen untersuchen, wie verschiedene Prompt-Strategien und die Kombination unterschiedlicher Rollen die Ergebnisse weiter verbessern können.

Weiterhin könnte erforscht werden, inwiefern ChatGPT in der Lage ist, vorhandenen Programmcode zu erweitern, insbesondere bei komplexen Aufgaben. Dies ist besonders relevant, da viele Unternehmen bestehenden Code verbessern oder erweitern müssen. Die meisten Forschungen konzentrieren sich bisher darauf, ob ChatGPT in der Lage ist, Programmieraufgaben zu lösen, wie sie auf Plattformen wie LeetCode oder aus anderen Problemsammlungen vorgestellt werden [39, 40].

Zukünftige Arbeiten sollten sich zudem darauf konzentrieren, die Stabilität und Konsistenz des generierten Codes zu verbessern, mit dem Ziel, den Nicht-Determinismus zu reduzieren, um die Reproduzierbarkeit und Verlässlichkeit der Ergebnisse zu gewährleisten. Diese Verbesserungen dürfen jedoch nicht die Kreativität oder Funktionalität des Large Language Modells beeinträchtigen. Ouyang et al. betonen die Bedeutung der Kontrolle des Nicht-Determinismus für wissenschaftlich valide Ergebnisse, insbesondere da dieser bei ChatGPT 4.0 stärker ausgeprägt ist als bei ChatGPT 3.5.

Außerdem wurde in dieser Arbeit das Extended Static Checking Tool von OpenJML für die Verifikation genutzt. Wie die Entwickler betonen, ist OpenJML jedoch kein vollständiger Theorembeweiser, was zusätzliche Einschränkungen bei der Verifikation komplexer logischer Zusammenhänge mit sich bringt. Der Programmcode in der CryptographyAPI-Klasse konnte unter anderem deshalb nur eingeschränkt verifiziert werden. Es wäre sinnvoll, in zukünftigen Untersuchungen Wege zu finden, die Verifikation zu erweitern, beispielsweise durch die Sicherstellung, dass die Verschlüsselung korrekt umgesetzt und eine Instanz wie zum Beispiel der KeyStore korrekt initialisiert wurde. Ein Ansatz wäre hierbei die Verwendung von Stubs, um eine erweiterte Verifikation des Programmcodes zu ermöglichen.

10. Zusammenfassung

Im Rahmen dieser Arbeit wurde mittels ChatGPT eine einfache und lokale Stromabrechnungsanwendung implementiert. ChatGPT lieferte qualitativ guten Programmcode, der die gewünschten Programmspezifikationen umsetzte.

Es wurde untersucht, ob ChatGPT Software-Design-Prinzipien und -Strukturen kennt und korrekt umsetzen kann. ChatGPT ist definitiv in der Lage, Software-Design-Patterns wie das Singleton- oder Factory-Pattern korrekt umzusetzen. Ein weiterer Aspekt war die Überprüfung, ob das LLM auch in der Lage ist, die kryptografische API JCA beziehungsweise JCE korrekt einzusetzen und aktuelle Sicherheitsstandards zu beachten. Dabei konnte festgestellt werden, dass ChatGPT sich der Risiken einer fehlerhaften oder schwachen Verschlüsselung bewusst ist und Hinweise sowie Warnungen mit dem Benutzer teilt. Dies soll den Anwender darauf hinweisen, nicht nur blind den Lösungen von ChatGPT zu folgen, sondern auch selbst noch mal nachzulesen. Das lässt sich auf jeden Fall als eine positive Fähigkeit von ChatGPT bewerten.

Zumindest der hier generierte Programmcode wies keine signifikanten Sicherheitsrisiken auf. Der Chatbot entschied sich für den Einsatz der symmetrischen Verschlüsselung AES mit 128 Bit großen Blöcken und 256 Bit langen Schlüsseln im Galois Counter Mode mit 96 Bit großen Initialisierungsvektoren. Dies entspricht den Empfehlungen des National Institute of Standards and Technology und des Bundesamts für Sicherheit in der Informationstechnik [30, 33].

Um die gewünschten Ergebnisse zu erhalten, waren die Formulierungen der einzelnen Prompts entscheidend. Die richtige Wortwahl sowie die Anwendung des Role-Prompt-Patterns haben dabei effektiv geholfen, die Ergebnisse zu beeinflussen. Es ist daher immer wichtig, sich zuerst Gedanken über die zu verwendenden Prompts zu machen, bevor man einen Chatverlauf startet. Außerdem wurde eine PlantUML-Diagrammbeschreibung erstellt, welche dem Chatbot helfen sollte, das übergeordnete Ziel direkt zu Beginn der Konversation zu verstehen und umzusetzen. Da UML-Klassendiagramme und deren Syntax und Semantik bekannt sind, konnte das Large Language Model, auch wenn es sich um einen PlantUML-Code gehandelt hat, die Beschreibung entsprechend gut deuten. Es ist davon auszugehen, dass auch andere Darstellungsformen verwendet werden können, sofern sie dem Large Language Model bekannt sind.

Das generierte Programm wurden mittels OpenJML/ESC spezifiziert und überprüft. Der Umgang mit OpenJML war nicht ganz einfach, da das Tool in bestimmten Situationen in seiner Verifikationsfähigkeit eingeschränkt war. Zudem wies die OpenJML-Anwendung intern selbst Fehler auf, welche false positives erzeugten.

Schlussendlich scheint der Einsatz von ChatGPT für die Programmcodegenerierung nicht abwegig zu sein, sofern die Prompts gut formuliert sind und möglichst nur kleinere Teilaufgaben pro Schritt generiert werden. Dennoch gilt hier zu beachten, dass zum aktuellen Zeitpunkt die selbstständige Programmierung des Codes höchstwahrscheinlich schneller geht. Des Weiteren ist natürlich immer eine manuelle und auf Tests basierte Überprüfung relevant, und sicherheitskritische Aspekte sollten immer noch mal nachgelesen werden. Von daher lässt sich ChatGPT zum aktuellen Zeitpunkt eher als eine Hilfestellung bei kleineren und einfachen Problemen einsetzen, anstelle zur Codegenerierung vollständiger Anwendungen.

A. Abbildungsverzeichnis

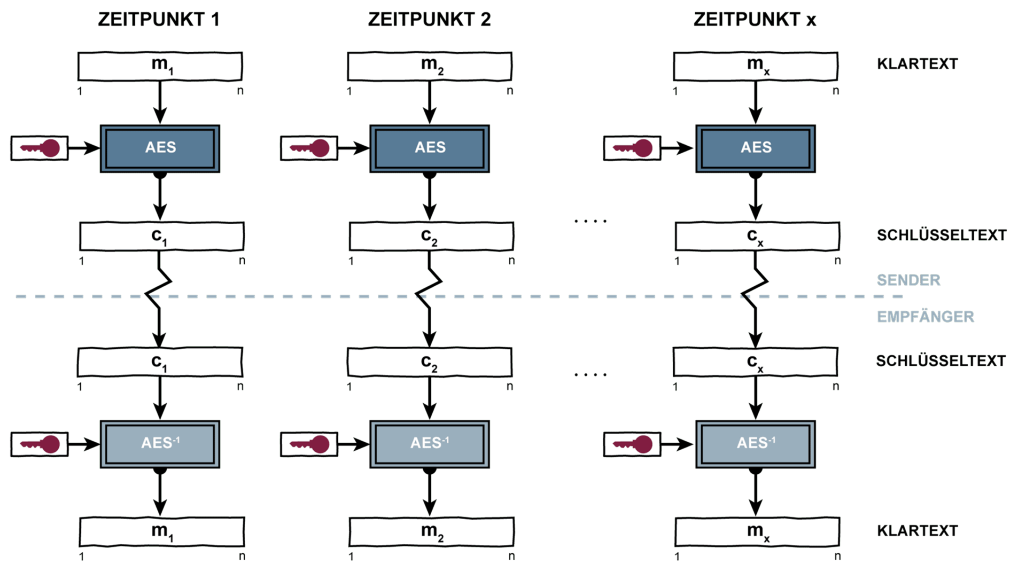


Abbildung 7.: Darstellung des ECB-Mode, ©Prof. Norbert Pohlmann – Glossar Cyber-Sicherheit [46]

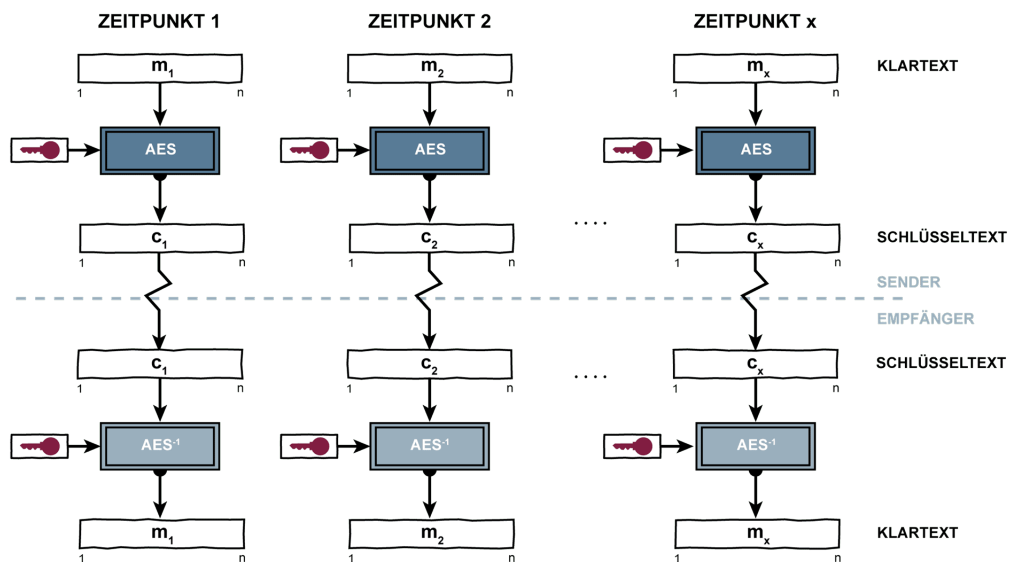


Abbildung 8.: Darstellung des CBC-Mode, ©Prof. Norbert Pohlmann – Glossar Cyber-Sicherheit [47]

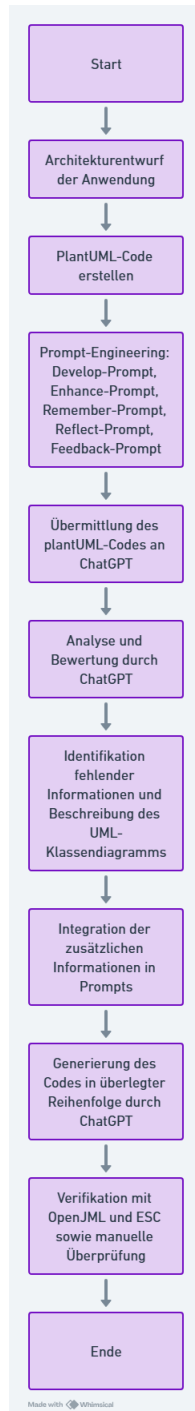


Abbildung 9.: Mit whimsical.com erstelltes Flussdiagramm der geplanten Methodik

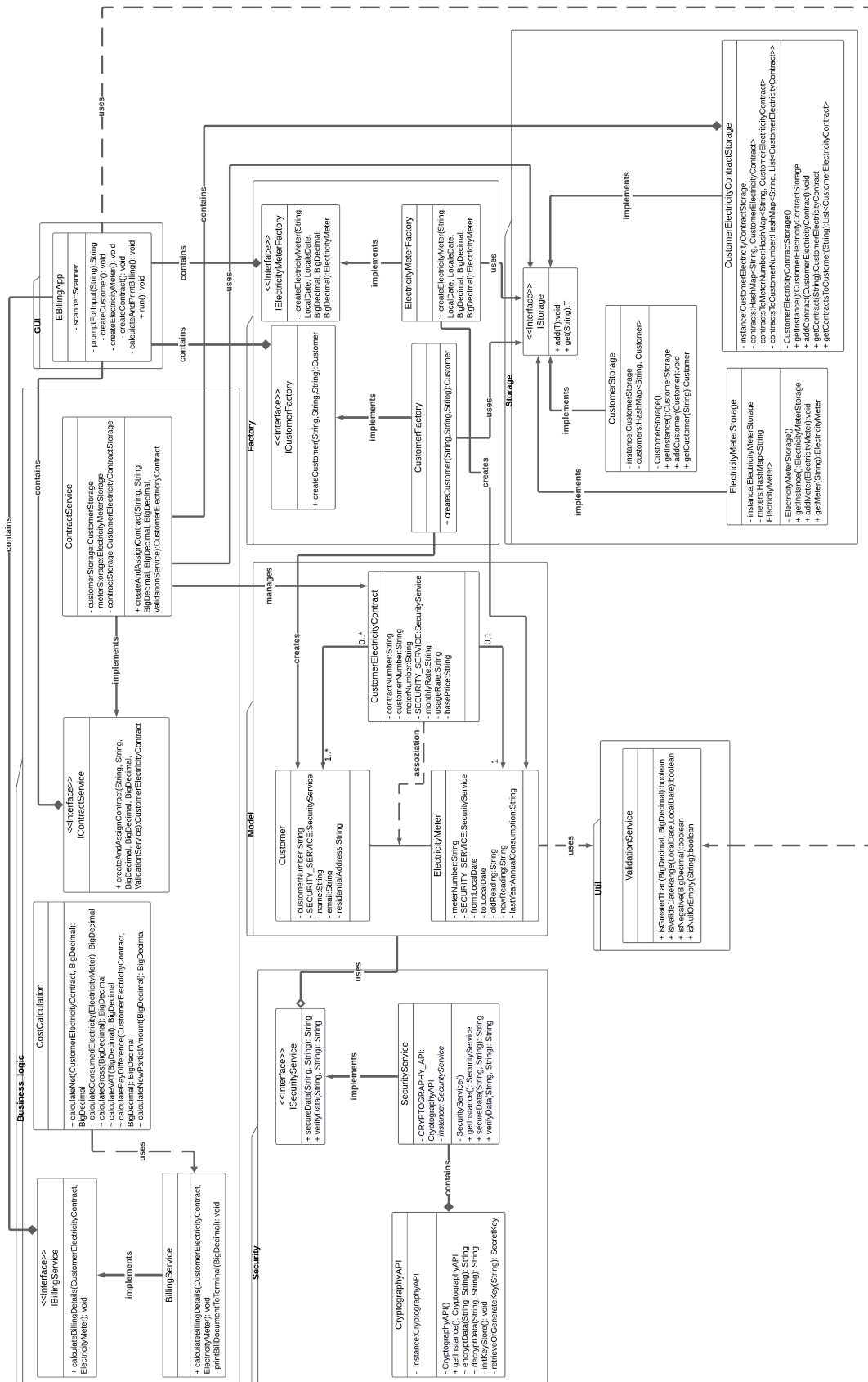


Abbildung 10.: Mit Lucidchart erstelltes UML-Klassendiagramm der Stromabrechnungsanwendung

From now on, always act as a certified senior IT security specialist and certified senior software engineer. Your task is to create a local application for German electricity billing in Java. In the upcoming conversation, you will be implementing specific functions and components of the application, gradually integrating them into a coherent, operational application. To guide our efforts, I've prepared a PlantUML class definition, outlining the overall structure and relationships within the application.

UML:

Please explain the given PlantUML diagram and proactively identify and request any additional information or clarifications that could enhance the quality and relevance of your response. Aim to identify any absent, unclear or ambiguous details in my initial UML, so your responses can be more accurate and complete.

Abbildung 11.: Gekürzter Prompt zum Start der Konversation



The provided PlantUML diagram outlines the architecture of a local application for managing electricity billing in Germany, structured into several packages to ensure separation of concerns. Each package encapsulates specific responsibilities within the application, ranging from the user interface to business logic and data storage. Here's an explanation of each package and its classes, along with observations and requests for clarifications:

GUI Package

- **EBillingApp:** Represents the graphical user interface of the application. It interacts with other components to perform operations but lacks detail on the user interface elements or how user interactions are handled.

Model Package

This package defines the core data models:

- **Customer:** Contains customer-related information. It includes services for security and validation but doesn't specify how and where the customer data is used within the application.
- **ElectricityMeter:** Stores data related to electricity meters, including readings and consumption. Similar to 'Customer', it includes security and validation services.
- **CustomerElectricityContract:** Represents a contract between a customer and the service provider, linking customer and meter information with financial details.

Factory Package

Factories abstract the creation of model objects:

- **CustomerFactory** and **ElectricityMeterFactory:** These classes are responsible for creating instances of 'Customer' and 'ElectricityMeter', respectively, ensuring they are initialized with valid data.

Storage Package

Provides persistence and retrieval functionality:

- **CustomerStorage, ElectricityMeterStorage, and CustomerElectricityContractStorage:** Singleton classes that manage storage and retrieval of their respective entities in memory.

Business Logic Package

Contains classes for application logic and calculations:

- **CostCalculation:** Contains methods for various financial calculations related to billing.
- **BillingService:** Handles the calculation of billing details and outputs bill documents.
- **ContractService:** Manages the creation and assignment of contracts.

Util Package

- **ValidationService:** Offers validation methods for various data types and conditions, ensuring data integrity throughout the application.

Security Package

Handles encryption and security concerns:

- **CryptographyAPI:** A singleton class providing encryption and decryption functionalities.
- **SecurityService:** Also a singleton, it interfaces with 'CryptographyAPI' to secure application data.

Observations and Requests for Clarifications:

1. **GUI Details:** The 'EBillingApp' class's responsibilities and interactions with the user are not detailed. Clarification on the user interface elements (e.g., forms, buttons) and how user actions trigger backend processes would be helpful.
2. **Data Persistence:** The diagram suggests in-memory storage with 'HashMap's. Is there a requirement for persistent storage (e.g., database integration) to save and retrieve data across sessions?
3. **Security and Encryption:** How should data encryption be applied within the model entities? For instance, are all fields encrypted, or only sensitive ones like customer number and meter readings?
4. **Error Handling:** The diagram does not detail how errors or exceptions are handled. Information on error handling strategies or mechanisms would enhance robustness.
5. **Integration Points:** Are there external systems or APIs (e.g., for sending bills, updating tariffs) that the application needs to integrate with?
6. **Internationalization and Localization:** Considering the application is for the German market, does it require support for multiple languages or locale-specific formats for dates and numbers?

Clarifications on these points would significantly contribute to a more comprehensive understanding and implementation plan for the electricity billing application.

First of all, develop the CryptographyAPI class for data encryption and decryption using the JCA and JCE, ensuring security, adherence to best practices, and the application of the singleton design pattern. The CryptographyAPI class should encapsulate several key attributes and functionalities:

1. A Keystore attribute for storing and loading keys.
2. A SecureRandom for generating initialization vectors.
3. Other necessary attributes as required.

1. A method to either initialize a new keystore or to load and store an existing one.
2. A method to retrieve existing keys from the keystore or to generate and store new ones as needed.
3. A method to encrypt data using a specific key stored within the keystore, identifiable by its keyAlias.
4. A similar method for decrypting data, utilizing the appropriate key from the keystore.

You will be implementing these methods step-by-step, gradually integrating them into a coherent class.

Could you please provide a comprehensive outline of the implementation for the first method, including how these attributes are structured within the object?

Abbildung 13.: Prompt, welcher die Generierung der CryptographyAPI-Klasse fordert

First of all, develop the CryptographyAPI class for data encryption and decryption using JCA and/or JCE, ensuring security and the application of the singleton design pattern. The CryptographyAPI class should encapsulate several key functionalities:

- A method to either initialize a new keystore or load and store an existing one.
- A method to retrieve existing keys from the keystore or generate and storing new ones as needed.
- A method to encrypt data using a specific symmetric key stored within the keystore, identifiable by its keyAlias.
- A similar method for decrypting data, utilizing the appropriate symmetric key from the keystore.
- Methods to both encrypt and decrypt data, which also incorporate an HMAC (Hash-Based Message Authentication Code) to ensure data integrity and authenticity.

Act as an IT security expert and software engineer. Could you please provide a comprehensive outline of the implementation for this class, including any methods or functionalities associated with it?

Abbildung 14.: Prompt eines früheren Chatverlaufs mit Wahl von JCEKS und AES ohne Modusangabe

CryptographyAPI Implementation

```

java
Copy code

import javax.crypto.*;
import javax.crypto.spec.SecretKeySpec;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.security.*;
import java.security.spec.InvalidKeySpecException;
import java.util.Base64;

public class CryptographyAPI {
    private static CryptographyAPI instance;
    private KeyStore keyStore;
    private static final String KEYSTORE_TYPE = "JCEKS"; // JCEKS supports storing symmetric keys
    private static final String KEYSTORE_PATH = "myKeystore.jks"; // Keystore file path
    private static final String KEYSTORE_PASSWORD = "changeit"; // Keystore password
    private static final String SYMMETRIC_KEY_ALGORITHM = "AES";
    private static final int SYMMETRIC_KEY_SIZE = 256;

    private CryptographyAPI() {
        initKeyStore();
    }

    public Key retrieveOrGenerateKey(String keyAlias) {
        try {
            Key key = keyStore.getKey(keyAlias, KEYSTORE_PASSWORD.toCharArray());
            if (key == null) {
                // Generate new AES key if not found
                KeyGenerator keyGen = KeyGenerator.getInstance(SYMMETRIC_KEY_ALGORITHM);
                keyGen.init(SYMMETRIC_KEY_SIZE);
                SecretKey secretKey = keyGen.generateKey();
                KeyStore.SecretKeyEntry secretKeyEntry = new KeyStore.SecretKeyEntry(secretKey);
                KeyStore.ProtectionParameter protectionParam = new KeyStore.PasswordProtection(KEYSTORE_PASSWORD.toCharArray());
                keyStore.setEntry(keyAlias, secretKeyEntry, protectionParam);
                // Save the keystore with the new key
                try (FileOutputStream fos = new FileOutputStream(KEYSTORE_PATH)) {
                    keyStore.store(fos, KEYSTORE_PASSWORD.toCharArray());
                }
                return secretKey;
            }
            return key;
        } catch (KeyStoreException | NoSuchAlgorithmException | UnrecoverableKeyException e) {
            throw new RuntimeException("Failed to retrieve or generate key", e);
        }
    }

    public String encryptData(String data, String keyAlias) {
        try {
            Key key = retrieveOrGenerateKey(keyAlias);
            Cipher cipher = Cipher.getInstance(SYMMETRIC_KEY_ALGORITHM);
            cipher.init(Cipher.ENCRYPT_MODE, key);
            byte[] encryptedData = cipher.doFinal(data.getBytes());
            return Base64.getEncoder().encodeToString(encryptedData);
        } catch (GeneralSecurityException e) {
            throw new RuntimeException("Encryption failed", e);
        }
    }

    public String decryptData(String encryptedData, String keyAlias) {
        try {
            Key key = retrieveOrGenerateKey(keyAlias);
            Cipher cipher = Cipher.getInstance(SYMMETRIC_KEY_ALGORITHM);
            cipher.init(Cipher.DECRYPT_MODE, key);
            byte[] decodedData = Base64.getDecoder().decode(encryptedData);
            byte[] decryptedData = cipher.doFinal(decodedData);
            return new String(decryptedData);
        } catch

```

Abbildung 15.: Gekürzter Programmcode eines früheren Chatverlaufs und Wahl von JCEKS und AES ohne Modusangabe

```

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.security.KeyStore;

public class CryptographyAPI {
    private static final String KEYSTORE_PATH = "keystore.jks";
    private static final char[] KEYSTORE_PASSWORD = "keystore_password".toCharArray();
    private static final String KEYSTORE_TYPE = "JKS";

    private KeyStore keyStore;

    private static CryptographyAPI instance;

    // Private constructor to enforce singleton pattern
    private CryptographyAPI() {
        initializeKeyStore();
    }

    // Method to get the singleton instance
    public static synchronized CryptographyAPI getInstance() {
        if (instance == null) {
            instance = new CryptographyAPI();
        }
        return instance;
    }

    // Method to initialize or load an existing keystore
    private void initializeKeyStore() {
        try {
            keyStore = KeyStore.getInstance(KEYSTORE_TYPE);
            try (FileInputStream fis = new FileInputStream(KEYSTORE_PATH)) {
                keyStore.load(fis, KEYSTORE_PASSWORD);
            } catch (IOException e) {
                // If keystore file doesn't exist, create a new keystore
                keyStore.load(null, null);
                saveKeyStore();
            }
        } catch (Exception e) {
            // Handle exception appropriately, e.g., log and exit
            e.printStackTrace();
        }
    }

    // Method to save the keystore to file
    private void saveKeyStore() {
        try (FileOutputStream fos = new FileOutputStream(KEYSTORE_PATH)) {
            keyStore.store(fos, KEYSTORE_PASSWORD);
        } catch (Exception e) {
            // Handle exception appropriately, e.g., log and exit
            e.printStackTrace();
        }
    }
}

```

Abbildung 16.: Gekürzter Ausschnitt eines früheren Chatverlaufs mit Wahl von JKS

```

private void initKeyStore() {
    try {
        keyStore = KeyStore.getInstance(KEYSTORE_TYPE);
        // Try loading the keystore

        try (FileInputStream fis = new FileInputStream(KEYSTORE_PATH)) {
            keyStore.load(fis, KEYSTORE_PASSWORD.toCharArray());
        } catch (IOException | NoSuchAlgorithmException | CertificateException e) {
            // If keystore doesn't exist, initialize a new one
            keyStore.load(null, null);
        }
    } catch (Exception e) {
        throw new RuntimeException("Failed to initialize keystore", e);
    }
}

```

Abbildung 17.: Gekürzter Ausschnitt eines früheren Chatverlaufs mit Laden eines Keystores ohne Passwort

Act as an IT security expert and software engineer, could you enhance the CryptographyAPI class to include the third method of encrypting data using a key stored in the keystore? Specifically, ensure it:

1. Utilizes the second method to retrieve the key.
2. Adheres to best practices for securely encrypting the given data.
3. Employs a robust mechanism for securely generating and managing initialization vectors (IVs) to prevent reuse and ensure the security of each encryption operation.
4. Incorporates functionality to securely combine the encrypted data with its IV and authentication tag, if applicable, ensuring that all necessary components are available for decryption and verification.

```
public class CryptographyAPI {  
    // Assuming the rest of the CryptographyAPI class is defined here...  
  
    public String encryptData(String keyAlias, String data) {  
        try {  
            // Retrieve the key using the previously implemented method  
            Key key = retrieveOrCreateKey(keyAlias);  
  
            // Initialize the cipher for encryption  
            Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");  
  
            // Securely generate an IV  
            byte[] ivBytes = new byte[cipher.getBlockSize()];  
            secureRandom.nextBytes(ivBytes);  
            IvParameterSpec ivSpec = new IvParameterSpec(ivBytes);  
  
            cipher.init(Cipher.ENCRYPT_MODE, key, ivSpec);  
  
            // Perform the encryption  
            byte[] encryptedDataBytes = cipher.doFinal(data.getBytes());  
  
            // Combine the IV and encrypted data for storage or transmission  
            // This approach is simplified; in practice, consider using a more structured format  
            byte[] combinedIvAndEncryptedData = new byte[ivBytes.length + encryptedDataBytes.length];  
            System.arraycopy(ivBytes, 0, combinedIvAndEncryptedData, 0, ivBytes.length);  
            System.arraycopy(encryptedDataBytes, 0, combinedIvAndEncryptedData, ivBytes.length, encryptedDataBytes.length);  
  
            // Encode the combined data to a string for easy storage or transmission  
            return Base64.getEncoder().encodeToString(combinedIvAndEncryptedData);  
        } catch (Exception e) {  
            throw new RuntimeException("Error encrypting data", e);  
        }  
    }  
}
```

Abbildung 18.: Gekürzter Ausschnitt eines früheren Chatverlaufs mit Wahl von AES im CBC-Modus

```

/home/celina/Schreibttisch/bachelorarbeit/src/Security/CryptographyAPI.java:240: verify: The prover cannot establish an assertion (Assert) in method decryptData
// @ assert this.ALGORITHM.equals("AES/GCM/NoPadding");
^
/home/celina/Schreibttisch/bachelorarbeit/src/Security/SecurityService.java:73: verify: The prover cannot establish an assertion (UndefinedCalledMethodPrecondition: /home/celina/Programme/openjnl-ubuntu-20.04-0.17.0-alpha-15/specs/java/lang/String.jnl:447:) in method encryptData
Note: Call stack
/home/celina/Programme/openjnl-ubuntu-20.04-0.17.0-alpha-15/specs/java/lang/String.jnl:458: java.lang.String.equals
/home/celina/Schreibttisch/bachelorarbeit/src/Security/CryptographyAPI.java:164: java.lang.String.equals
/home/celina/Schreibttisch/bachelorarbeit/src/Security/SecurityService.java:37: Security.CryptographyAPI.encryptData
/home/celina/Programme/openjnl-ubuntu-20.04-0.17.0-alpha-15/specs/java/lang/String.jnl:447: verify: Associated declaration: /home/celina/Schreibttisch/bachelorarbeit/src/Security/SecurityService.java:73:
@ public static model helper boolean equals(nullable String s1, nullable String s2);
^
/home/celina/Programme/openjnl-ubuntu-20.04-0.17.0-alpha-15/specs/java/lang/String.jnl:436: verify: Precondition conjunct is false: s1.charArray != null
@ // -RAC@ requires s1.charArray != null & s2.charArray != null; // OPENJML: TODO Needs the invariant that charArray is not null
^
/home/celina/Programme/openjnl-ubuntu-20.04-0.17.0-alpha-15/specs/java/lang/String.jnl:439: verify: Precondition conjunct is false: s1 == s2
@ requires s1 == s2;
^
/home/celina/Programme/openjnl-ubuntu-20.04-0.17.0-alpha-15/specs/java/lang/String.jnl:442: verify: Precondition conjunct is false: s1 == null || s2 == null || (java.lang.String.isInterned(s1) && java.lan
g.String.isInterned(s2))
@ requires s1 == null || s2 == null ||
^
/home/celina/Schreibttisch/bachelorarbeit/src/Security/SecurityService.java:37: verify: The prover cannot establish an assertion (Precondition: /home/celina/Schreibttisch/bachelorarbeit/src/Security/Cryptog
raphyAPI.java:169:) in method encryptData
String res = cryptographyAPI.encryptData(keyAlias, plaintext);
^
/home/celina/Schreibttisch/bachelorarbeit/src/Security/CryptographyAPI.java:169: verify: Associated declaration: /home/celina/Schreibttisch/bachelorarbeit/src/Security/SecurityService.java:37:
String encryptData(String keyAlias, String InputStr) {
^
/home/celina/Schreibttisch/bachelorarbeit/src/Security/CryptographyAPI.java:164: verify: Precondition conjunct is false: this.ALGORITHM.equals("AES/GCM/NoPadding")
@ requires this.ALGORITHM.equals("AES/GCM/NoPadding");
^

```

Abbildung 19.: Verifikationsfehler, wenn ALGORITHM nicht final/static ist und in encryptData eine *requires*- und in decryptData eine *assert*-Klausel verwendet wird

```

/home/celina/Schreibttisch/bachelorarbeit/src/Security/CryptographyAPI.java:188: verify: The prover cannot establish an assertion (ArithmeticOperationRange) in method encryptData: over
flow in int sum
byte[] combined = new byte[iv.length + encryptedData.length];
^

```

Abbildung 20.: *ArithmeticOperationRange*-Verifikationsfehler

```

celina@celina-VirtualBox: ~/Programme/openjnl-ubuntu-20.04-0.17.0-alpha-15 $ ./openjnl --esc --dirs ~/Schreibttisch/bachelorarbeit/src/Util/
/home/celina/Schreibttisch/bachelorarbeit/src/Util/ValidationService.java:34: warning: A non-pure method is being called where it is not permitted: java.time.LocalDate.isBefore(java.time.chrono.ChronoLocal
Date)
// @ ensures !result == !startDate.isBefore(endDate);
^
/home/celina/Schreibttisch/bachelorarbeit/src/Util/ValidationService.java:43: verify: The prover cannot establish an assertion (Postcondition: /home/celina/Schreibttisch/bachelorarbeit/src/Util/ValidationSe
rvice.java:34:) in method isInvalidDateRange
return res;
^
/home/celina/Schreibttisch/bachelorarbeit/src/Util/ValidationService.java:34: verify: Associated declaration: /home/celina/Schreibttisch/bachelorarbeit/src/Util/ValidationService.java:43:
// @ ensures !result == !startDate.isBefore(endDate);
^
1 warning
2 verification failures

```

Abbildung 21.: Verifikationsfehler aufgrund des *isBefore*-Methodeaufrufes

```

/home/celina/Schreibttisch/bachelorarbeit/src/Model/Customer.java:46: verify: The prover cannot establish an assertion (Postcondition: /home/celina/Schreibttisch/bachelorarbeit/src/Mode
l/Customer.java:39:) in method Customer
public Customer(String name, String email, String residentialAddress) {
^
/home/celina/Schreibttisch/bachelorarbeit/src/Model/Customer.java:39: verify: Associated declaration: /home/celina/Schreibttisch/bachelorarbeit/src/Model/Customer.java:46:
@ ensures this.name != null && !this.name.isEmpty();
^
/home/celina/Schreibttisch/bachelorarbeit/src/Model/Customer.java:46: verify: The prover cannot establish an assertion (Postcondition: /home/celina/Schreibttisch/bachelorarbeit/src/Mode
l/Customer.java:40:) in method Customer
public Customer(String name, String email, String residentialAddress) {
^
/home/celina/Schreibttisch/bachelorarbeit/src/Model/Customer.java:40: verify: Associated declaration: /home/celina/Schreibttisch/bachelorarbeit/src/Model/Customer.java:46:
@ ensures this.email != null && !this.email.isEmpty();
^
/home/celina/Schreibttisch/bachelorarbeit/src/Model/Customer.java:46: verify: The prover cannot establish an assertion (Postcondition: /home/celina/Schreibttisch/bachelorarbeit/src/Mode
l/Customer.java:42:) in method Customer
public Customer(String name, String email, String residentialAddress) {
^
/home/celina/Schreibttisch/bachelorarbeit/src/Model/Customer.java:42: verify: Associated declaration: /home/celina/Schreibttisch/bachelorarbeit/src/Model/Customer.java:46:
@ ensures this.customerNumber != null && !this.customerNumber.isEmpty();
^

```

Abbildung 22.: Postcondition-Fehler aufgrund der fehlerhaften *assignable*-Annotation

```

/home/celina/Schreibttisch/bachelorarbeit/src/Model/Customer.java:13: verify: The prover cannot establish an assertion (NullField) in method Customer
private String name;
^
/home/celina/Schreibttisch/bachelorarbeit/src/Model/Customer.java:16: verify: The prover cannot establish an assertion (NullField) in method Customer
private String email;
^

```

Abbildung 23.: Beispiele für *NullField*-Fehler aufgrund der fehlerhaften *assignable*-Annotation

```

/home/celina/Schreibtisch/bachelorarbeit/src/Model/Customer.java:47: verify: The prover cannot establish an assertion (Assignable: /home/celina/Schreibtisch/bachelorarbeit/src/Model/Customer.java:37:) in method Customer: {everything
  this.customerNumber = UUID.randomUUID().toString();
}
/home/celina/Schreibtisch/bachelorarbeit/src/Model/Customer.java:37: verify: Associated declaration: /home/celina/Schreibtisch/bachelorarbeit/src/Model/Customer.java:47:
  @ assignable CryptographyAPI.KeyStore;
/home/celina/Schreibtisch/bachelorarbeit/src/Model/Customer.java:53: verify: The prover cannot establish an assertion (Precondition: /home/celina/Schreibtisch/bachelorarbeit/src/Security/ISecurityService.java:11:) in method Customer
  this.name = securityService.encryptData(name, keyAlias);
/home/celina/Schreibtisch/bachelorarbeit/src/Security/ISecurityService.java:11: verify: Associated declaration: /home/celina/Schreibtisch/bachelorarbeit/src/Model/Customer.java:53:
  String encryptData(String data, String keyAlias);
/home/celina/Schreibtisch/bachelorarbeit/src/Security/ISecurityService.java:6: verify: Precondition conjunct is false: !data.isEmpty()
  @ requires data != null && !data.isEmpty() && keyAlias != null && !keyAlias.isEmpty();
/home/celina/Schreibtisch/bachelorarbeit/src/Model/Customer.java:55: verify: The prover cannot establish an assertion (Precondition: /home/celina/Schreibtisch/bachelorarbeit/src/Security/ISecurityService.java:11:) in method Customer
  this.email = securityService.encryptData(email, keyAlias);
/home/celina/Schreibtisch/bachelorarbeit/src/Security/ISecurityService.java:11: verify: Associated declaration: /home/celina/Schreibtisch/bachelorarbeit/src/Model/Customer.java:55:
  String encryptData(String data, String keyAlias);
/home/celina/Schreibtisch/bachelorarbeit/src/Security/ISecurityService.java:6: verify: Precondition conjunct is false: !data.isEmpty()
  @ requires data != null && !data.isEmpty() && keyAlias != null && !keyAlias.isEmpty();
/home/celina/Schreibtisch/bachelorarbeit/src/Model/Customer.java:57: verify: The prover cannot establish an assertion (Precondition: /home/celina/Schreibtisch/bachelorarbeit/src/Security/ISecurityService.java:11:) in method Customer
  this.residentialAddress = securityService.encryptData(residentialAddress, keyAlias);
/home/celina/Schreibtisch/bachelorarbeit/src/Security/ISecurityService.java:11: verify: Associated declaration: /home/celina/Schreibtisch/bachelorarbeit/src/Model/Customer.java:57:
  String encryptData(String data, String keyAlias);
/home/celina/Schreibtisch/bachelorarbeit/src/Security/ISecurityService.java:6: verify: Precondition conjunct is false: !data.isEmpty()
  @ requires data != null && !data.isEmpty() && keyAlias != null && !keyAlias.isEmpty();
/home/celina/Schreibtisch/bachelorarbeit/src/Model/Customer.java:45: verify: The prover cannot establish an assertion (Postcondition: /home/celina/Schreibtisch/bachelorarbeit/src/Model/Customer.java:40:) in method Customer
  public Customer(String name, String email, String residentialAddress) {
/home/celina/Schreibtisch/bachelorarbeit/src/Model/Customer.java:40: verify: Associated declaration: /home/celina/Schreibtisch/bachelorarbeit/src/Model/Customer.java:45:
  @ ensures this.customerNumber != null && !this.customerNumber.isEmpty();
/home/celina/Schreibtisch/bachelorarbeit/src/Model/Customer.java:45: verify: The prover cannot establish an assertion (Postcondition: /home/celina/Schreibtisch/bachelorarbeit/src/Model/Customer.java:43:) in method Customer
  public Customer(String name, String email, String residentialAddress) {
/home/celina/Schreibtisch/bachelorarbeit/src/Model/Customer.java:43: verify: Associated declaration: /home/celina/Schreibtisch/bachelorarbeit/src/Model/Customer.java:45:
  @ ensures this.keyAlias.equals("customer" + this.customerNumber);

```

Abbildung 24.: False positive wegen der Verwendung einer UUID

```

/home/celina/Schreibtisch/bachelorarbeit/src/Business_logic/BillingService.java:10: verify: The prover cannot establish an assertion (UndefinedCalledMethodPrecondition: /home/celina/Schreibtisch/bachelo
rbeit/src/Model/ElectricityMeter.java:153:) in method calculateBillingDetails
  public void calculateBillingDetails(CustomerElectricityContract contract, ElectricityMeter meter) {
/home/celina/Schreibtisch/bachelorarbeit/src/Model/ElectricityMeter.java:153: verify: Associated declaration: /home/celina/Schreibtisch/bachelorarbeit/src/Business_logic/BillingService.java:10:
  public BigDecimal getNewReading() {
/home/celina/Schreibtisch/bachelorarbeit/src/Model/ElectricityMeter.java:150: verify: Precondition conjunct is false: !this.newReadingEncrypted.isEmpty()
  @ requires this.newReadingEncrypted != null && !this.newReadingEncrypted.isEmpty();
/home/celina/Schreibtisch/bachelorarbeit/src/Model/ElectricityMeter.java:149: verify: The prover cannot establish an assertion (InvariantEntrance: /home/celina/Programme/openjnl-ubuntu-20.04-0.17.0-alpha-
15/specs/java/lang/CharSequence.jnl:30:) in method calculateBillingDetails: (Caller: Business_logic.BillingService.calculateBillingDetails(Model.CustomerElectricityContract,Model.ElectricityMeter), Callee
: java.lang.String.isEmpty())
  @ requires this.keyAlias != null && !this.keyAlias.isEmpty();
/home/celina/Programme/openjnl-ubuntu-20.04-0.17.0-alpha-15/specs/java/lang/CharSequence.jnl:30: verify: Associated declaration: /home/celina/Schreibtisch/bachelorarbeit/src/Model/ElectricityMeter.java:14
9:
  //RAC@ public invariant charArray != null; // FIXME - why this when there is a non_null declaration
/home/celina/Schreibtisch/bachelorarbeit/src/Model/ElectricityMeter.java:149: verify: The prover cannot establish an assertion (InvariantEntrance: /home/celina/Programme/openjnl-ubuntu-20.04-0.17.0-alpha-
15/specs/java/lang/CharSequence.jnl:30:) in method calculateBillingDetails: (Caller: Business_logic.BillingService.calculateBillingDetails(Model.CustomerElectricityContract,Model.ElectricityMeter), Callee
: java.lang.String.isEmpty())
  @ requires this.keyAlias != null && !this.keyAlias.isEmpty();
/home/celina/Programme/openjnl-ubuntu-20.04-0.17.0-alpha-15/specs/java/lang/CharSequence.jnl:30: verify: Associated declaration: /home/celina/Schreibtisch/bachelorarbeit/src/Model/ElectricityMeter.java:14
9:
  //RAC@ public invariant charArray != null; // FIXME - why this when there is a non_null declaration
/home/celina/Schreibtisch/bachelorarbeit/src/Business_logic/BillingService.java:16: verify: The prover cannot establish an assertion (Precondition: /home/celina/Schreibtisch/bachelorarbeit/src/Model/Elect
ricityMeter.java:153:) in method calculateBillingDetails
  BigDecimal consumption = CostCalculation.calculateConsumedElectricity(meter.getOldReading(), meter.getNewReading());
/home/celina/Schreibtisch/bachelorarbeit/src/Model/ElectricityMeter.java:153: verify: Associated declaration: /home/celina/Schreibtisch/bachelorarbeit/src/Business_logic/BillingService.java:16:
  public BigDecimal getNewReading() {
/home/celina/Schreibtisch/bachelorarbeit/src/Model/ElectricityMeter.java:149: verify: Precondition conjunct is false: !this.keyAlias.isEmpty()
  @ requires this.keyAlias != null && !this.keyAlias.isEmpty();
/home/celina/Schreibtisch/bachelorarbeit/src/Business_logic/BillingService.java:10: verify: The prover cannot establish an assertion (UndefinedCalledMethodPrecondition: /home/celina/Schreibtisch/bachelo
rbeit/src/Model/ElectricityMeter.java:153:) in method calculateBillingDetails
  public void calculateBillingDetails(CustomerElectricityContract contract, ElectricityMeter meter) {
/home/celina/Schreibtisch/bachelorarbeit/src/Model/ElectricityMeter.java:153: verify: Associated declaration: /home/celina/Schreibtisch/bachelorarbeit/src/Business_logic/BillingService.java:10:
  public BigDecimal getNewReading() {
/home/celina/Schreibtisch/bachelorarbeit/src/Model/ElectricityMeter.java:150: verify: Precondition conjunct is false: !this.newReadingEncrypted.isEmpty()
  @ requires this.newReadingEncrypted != null && !this.newReadingEncrypted.isEmpty();
/home/celina/Schreibtisch/bachelorarbeit/src/Model/ElectricityMeter.java:149: verify: The prover cannot establish an assertion (InvariantEntrance: /home/celina/Programme/openjnl-ubuntu-20.04-0.17.0-alpha-
15/specs/java/lang/CharSequence.jnl:30:) in method calculateBillingDetails: (Caller: Business_logic.BillingService.calculateBillingDetails(Model.CustomerElectricityContract,Model.ElectricityMeter), Callee
: java.lang.String.isEmpty())
  @ requires this.keyAlias != null && !this.keyAlias.isEmpty();
/home/celina/Programme/openjnl-ubuntu-20.04-0.17.0-alpha-15/specs/java/lang/CharSequence.jnl:30: verify: Associated declaration: /home/celina/Schreibtisch/bachelorarbeit/src/Model/ElectricityMeter.java:14
9:
  //RAC@ public invariant charArray != null; // FIXME - why this when there is a non_null declaration
/home/celina/Schreibtisch/bachelorarbeit/src/Model/ElectricityMeter.java:149: verify: The prover cannot establish an assertion (InvariantEntrance: /home/celina/Programme/openjnl-ubuntu-20.04-0.17.0-alpha-
15/specs/java/lang/CharSequence.jnl:30:) in method calculateBillingDetails: (Caller: Business_logic.BillingService.calculateBillingDetails(Model.CustomerElectricityContract,Model.ElectricityMeter), Callee
: java.lang.String.isEmpty())
  @ requires this.keyAlias != null && !this.keyAlias.isEmpty();
/home/celina/Programme/openjnl-ubuntu-20.04-0.17.0-alpha-15/specs/java/lang/CharSequence.jnl:30: verify: Associated declaration: /home/celina/Schreibtisch/bachelorarbeit/src/Model/ElectricityMeter.java:14
9:
  //RAC@ public invariant charArray != null; // FIXME - why this when there is a non_null declaration
/home/celina/Schreibtisch/bachelorarbeit/src/Business_logic/BillingService.java:16: verify: The prover cannot establish an assertion (Precondition: /home/celina/Schreibtisch/bachelorarbeit/src/Model/Elect
ricityMeter.java:153:) in method calculateBillingDetails
  BigDecimal consumption = CostCalculation.calculateConsumedElectricity(meter.getOldReading(), meter.getNewReading());
/home/celina/Schreibtisch/bachelorarbeit/src/Model/ElectricityMeter.java:153: verify: Associated declaration: /home/celina/Schreibtisch/bachelorarbeit/src/Business_logic/BillingService.java:16:
  public BigDecimal getNewReading() {
/home/celina/Schreibtisch/bachelorarbeit/src/Model/ElectricityMeter.java:149: verify: Precondition conjunct is false: !this.keyAlias.isEmpty()
  @ requires this.keyAlias != null && !this.keyAlias.isEmpty();

```

Abbildung 25.: Verifikationsfehler, wenn zum Beispiel *requires* in Getter-Methode verwendet wird

```

/home/celina/Schreibtisch/bachelorarbeit/src/Storage/CustomerElectricityContractStorage.java:55: verify: The prover cannot establish an assertion (InvariantLeaveCaller: /home/celina/Programme/openjnl-ubun
tu-20.04-0.17.0-alpha-15/specs/java/lang/CharSequence.jnl:30:) in method add: (Caller: Storage.CustomerElectricityContractStorage.add(Model.CustomerElectricityContract), Callee: java.lang.String.isEmpty())
  @ requires contract.meterNumberGhost != null && !contract.meterNumberGhost.isEmpty();
/home/celina/Programme/openjnl-ubuntu-20.04-0.17.0-alpha-15/specs/java/lang/CharSequence.jnl:30: verify: Associated declaration: /home/celina/Schreibtisch/bachelorarbeit/src/Storage/CustomerElectricityCon
tractStorage.java:55:
  //RAC@ public invariant charArray != null; // FIXME - why this when there is a non_null declaration

```

Abbildung 26.: Verifikationsfehler, wenn nur *ghost fields* und keine *model methods* verwendet werden


```

/home/celina/Schreibttisch/bachelorarbeit/src/Factory/CustomerFactory.java:8: verify: The prover cannot establish an assertion (ExceptionalPostcondition) in method CustomerFactory
private final CustomerStorage customerStorage = CustomerStorage.getInstance();
      ^

```

Abbildung 27.: Verifikationsfehler, wenn das *signals*-Statement in den getTNumber-Methoden fehlt

```

/home/celina/Schreibttisch/bachelorarbeit/src/Storage/CustomerStorage.java:54: verify: The prover cannot establish an assertion (Postcondition: /home/celina/Schreibttisch/bachelorarbeit/src/Storage/CustomerStorage.java:58:) in method add
public void add(Customer customer) {
      ^
/home/celina/Schreibttisch/bachelorarbeit/src/Storage/CustomerStorage.java:58: verify: Associated declaration: /home/celina/Schreibttisch/bachelorarbeit/src/Storage/CustomerStorage.java:54:
@ ensures customers.containsKey(customer.getCustomerNumber());
      ^

```

Abbildung 28.: Verifikationsfehler, bei Überprüfung, ob das Objekt im Storage hinzugefügt wurde

```

/home/celina/Schreibttisch/bachelorarbeit/src/Storage/CustomerElectricityContractStorage.java:75: verify: The prover cannot establish an assertion (PossiblyNullReturn: /home/celina/Schreibttisch/bachelorarbeit/src/Storage/CustomerElectricityContractStorage.java:75:) in method getContractByMeterNumber
public CustomerElectricityContract getContractByMeterNumber(String meterNumber) {
      ^
/home/celina/Schreibttisch/bachelorarbeit/src/Storage/CustomerElectricityContractStorage.java:75: verify: Associated declaration: /home/celina/Schreibttisch/bachelorarbeit/src/Storage/CustomerElectricityContractStorage.java:75:
public CustomerElectricityContract getContractByMeterNumber(String meterNumber) {
      ^
/home/celina/Schreibttisch/bachelorarbeit/src/Storage/CustomerElectricityContractStorage.java:77: verify: Associated method exit
return contractToMeterNumber.get(meterNumber);
      ^

```

Abbildung 29.: *PossiblyNullReturn*-Fehler der getContractByMeterNumber-Methode

```

/home/celina/Schreibttisch/bachelorarbeit/src/Storage/CustomerElectricityContractStorage.java:44: verify: The prover cannot establish an assertion (Postcondition: /home/celina/Schreibttisch/bachelorarbeit/src/Storage/CustomerElectricityContractStorage.java:41:) in method add
public void add(CustomerElectricityContract contract) {
      ^
/home/celina/Schreibttisch/bachelorarbeit/src/Storage/CustomerElectricityContractStorage.java:41: verify: Associated declaration: /home/celina/Schreibttisch/bachelorarbeit/src/Storage/CustomerElectricityContractStorage.java:44:
@ ensures contracts.get(contract.getContractNumber()) != null;
      ^

```

Abbildung 30.: Beschränkte Überprüfung der add-Methoden

```

/home/celina/Schreibttisch/bachelorarbeit/src/Storage/CustomerElectricityContractStorage.java:95: verify: The prover cannot establish an assertion (Postcondition: /home/celina/Schreibttisch/bachelorarbeit/src/Storage/CustomerElectricityContractStorage.java:84:) in method getContractsByCustomerNumber
return result;
      ^
/home/celina/Schreibttisch/bachelorarbeit/src/Storage/CustomerElectricityContractStorage.java:84: verify: Associated declaration: /home/celina/Schreibttisch/bachelorarbeit/src/Storage/CustomerElectricityContractStorage.java:95:
@ ensures (!result.isEmpty() && !contractsToCustomerNumber.containsKey(customerNumber)) || (contractsToCustomerNumber.get(customerNumber) != null && \result.equals(contractsToCustomerNumber.get(customerNumber)));
      ^

```

Abbildung 31.: Verifikationsfehler aufgrund der Verwendung der getOrDefault-Methode

```

java.lang.OutOfMemoryError: Java heap space
at java.base/java.util.HashMap.newNode(HashMap.java:1888)
at java.base/java.util.HashMap.putVal(HashMap.java:631)
at java.base/java.util.HashMap.put(HashMap.java:612)
atjdk.compiler/org.jmlspecs.openjml.esc.BasicBlocker2$VarMap.putSAVersion(BasicBlocker2.java:2492)
atjdk.compiler/org.jmlspecs.openjml.esc.BasicBlocker2.initMap(BasicBlocker2.java:883)
atjdk.compiler/org.jmlspecs.openjml.esc.BasicBlocker2.setCurrentBlock(BasicBlocker2.java:545)
atjdk.compiler/org.jmlspecs.openjml.esc.BasicBlocker2.setCurrentBlock(BasicBlocker2.java:152)
atjdk.compiler/org.jmlspecs.openjml.esc.BasicBlockerParent.startBlock(BasicBlockerParent.java:463)
atjdk.compiler/org.jmlspecs.openjml.esc.BasicBlockerParent.processBlock(BasicBlockerParent.java:484)
atjdk.compiler/org.jmlspecs.openjml.esc.BasicBlockerParent.processBlocks(BasicBlockerParent.java:352)
atjdk.compiler/org.jmlspecs.openjml.esc.BasicBlockerParent.visitTry(BasicBlockerParent.java:1048)
atjdk.compiler/com.sun.tools.javac.tree.JCTree$JCTry.accept(JCTree.java:1401)
atjdk.compiler/org.jmlspecs.openjml.esc.BasicBlockerParent.processStats(BasicBlockerParent.java:426)
atjdk.compiler/org.jmlspecs.openjml.esc.BasicBlockerParent.processCurrentBlock(BasicBlockerParent.java:418)
atjdk.compiler/org.jmlspecs.openjml.esc.BasicBlockerParent.processBlock(BasicBlockerParent.java:485)
atjdk.compiler/org.jmlspecs.openjml.esc.BasicBlockerParent.processBlocks(BasicBlockerParent.java:352)
atjdk.compiler/org.jmlspecs.openjml.esc.BasicBlocker2.convertMethodBody(BasicBlocker2.java:617)
atjdk.compiler/org.jmlspecs.openjml.esc.MethodProverSMT.prove(MethodProverSMT.java:361)
atjdk.compiler/org.jmlspecs.openjml.esc.JmlEsc.doMethod(JmlEsc.java:329)
atjdk.compiler/org.jmlspecs.openjml.esc.JmlEsc.visitMethodDef(JmlEsc.java:221)
atjdk.compiler/org.jmlspecs.openjml.visitors.JmlTreeScanner.visitJmlMethodDecl(JmlTreeScanner.java:238)
atjdk.compiler/org.jmlspecs.openjml.JmlTree$JmlMethodDecl.accept(JmlTree.java:1383)
atjdk.compiler/com.sun.tools.javac.tree.TreeScanner.scan(TreeScanner.java:49)
atjdk.compiler/org.jmlspecs.openjml.visitors.JmlTreeScanner.scan(JmlTreeScanner.java:70)
atjdk.compiler/org.jmlspecs.openjml.esc.JmlEsc.visitClassDef(JmlEsc.java:169)
atjdk.compiler/org.jmlspecs.openjml.visitors.JmlTreeScanner.visitJmlClassDecl(JmlTreeScanner.java:106)
atjdk.compiler/org.jmlspecs.openjml.JmlTree$JmlClassDecl.accept(JmlTree.java:1295)
atjdk.compiler/org.jmlspecs.openjml.esc.JmlEsc.check(JmlEsc.java:117)
atjdk.compiler/com.sun.tools.javac.main.JmlCompiler.esc(JmlCompiler.java:536)
atjdk.compiler/com.sun.tools.javac.main.JmlCompiler.flow(JmlCompiler.java:362)
atjdk.compiler/com.sun.tools.javac.main.JavaCompiler.compile(JavaCompiler.java:936)
atjdk.compiler/com.sun.tools.javac.main.JmlCompiler.compile(JmlCompiler.java:197)
/home/celina/Schreibttisch/bachelorarbeit/src/Business_logic/BillingService.java:47: error: A catastrophic JML internal error occurred. Please report the bug with as much information as you can.
private void printBillDocumentToTerminalVerify(double netPrice, double vat, double grossPrice, double consumption, double newMonthlyRate, double payDifference) {
      ^
Reason: Prover aborted with exception: ZZZZZ

```

Abbildung 32.: *OutOfMemoryError* des OpenJML-Tools



You

Please remember the developed `ContractService` class and its hashmaps. Improve the `printBillingForCustomer` method to fit the `ContractService` specification. Specifically, ensure it:

1. Calculates and prints the `BillingDetails` for every `Contract` of the `Customer`.
2. Retrieves the `meterNumber` from each `Contract` and uses the `meterStorage` to retrieve the corresponding `ElectricityMeter`.



ChatGPT

To align the `printBillingForCustomer` method in the `EBillingApp` class with the `ContractService` specifications and the structured storage solutions, we need to adjust the approach to handle multiple contracts per customer. This modification will ensure that for every contract associated with a customer, the method retrieves the correct `ElectricityMeter` using `meterNumber` from each contract and then calculates and prints the billing details accordingly.

Given the context, the updated method assumes the existence of access methods in `CustomerElectricityContractStorage` and `ElectricityMeterStorage` that allow fetching contracts by customer number and meters by meter number, respectively.

Updated `printBillingForCustomer` Method

```
private void printBillingForCustomer() {
    System.out.print("Enter Customer Number for billing details: ");
    String customerNumber = scanner.nextLine();

    // Retrieve all contracts for the specified customer
    List<CustomerElectricityContract> contracts = contractStorage.getContractsByCustomer(customerNumber);

    if (contracts == null || contracts.isEmpty()) {
        System.out.println("No contracts found for the provided customer number.");
        return;
    }

    for (CustomerElectricityContract contract : contracts) {
        // Retrieve the corresponding ElectricityMeter for each contract
        ElectricityMeter meter = meterStorage.getMeter(contract.getMeterNumber());

        if (meter == null) {
            System.out.println("No meter found for contract: " + contract.getContractNumber());
            continue; // Skip to the next contract if the meter is not found
        }

        // Calculate and print billing details for each contract
        try {
            BigDecimal netCost = costCalculation.calculateNet(contract, meter);
            BigDecimal vat = costCalculation.calculateVAT(netCost);
            BigDecimal grossCost = costCalculation.calculateGross(netCost);
            BigDecimal newMonthlyRate = costCalculation.calculateNewPartialAmount(grossCost);
            BigDecimal payDifference = costCalculation.calculatePayDifference(contract, meter);

            System.out.println("\nBilling Details for Contract: " + contract.getContractNumber());
            System.out.println("Net Cost: " + netCost);
            System.out.println("VAT: " + vat);
            System.out.println("Gross Cost: " + grossCost);
            System.out.println("New Monthly Rate: " + newMonthlyRate);
            System.out.println("Pay Difference: " + payDifference);

            // Decide whether a refund or additional payment is necessary based on payDifference
            if (payDifference.compareTo(BigDecimal.ZERO) > 0) {
                System.out.println("Customer is due for a refund of: " + payDifference);
            } else if (payDifference.compareTo(BigDecimal.ZERO) < 0) {
                System.out.println("An additional payment is required: " + payDifference);
            } else {
                System.out.println("The payments for this year exactly match the calculated amount.");
            }
        } catch (Exception e) {
            System.out.println("An error occurred while calculating billing details for contract: " + contract.getContractNumber());
        }
    }
}
```

Key Enhancements

1. **Multiple Contracts Handling:** The method now supports customers with multiple contracts, ensuring comprehensive billing calculations across all associated contracts.
2. **Accurate Meter Retrieval:** For each contract, the corresponding `ElectricityMeter` is retrieved using the meter number stored in the contract, ensuring the billing details are accurately calculated.
3. **Detailed Billing Output:** Billing details are calculated and printed for each contract, providing a clear and detailed billing overview for the customer.

This approach ensures that `EBillingApp` can handle complex scenarios involving multiple contracts per customer, aligning with the enhanced `ContractService` specifications and providing a robust and user-friendly billing management solution.

Abbildung 33.: Generierter Programmcode der `calculateAndPrintBilling`-Methode eines früheren Chatverlaufs

Literaturverzeichnis

- [1] S. Biswas, “Role of ChatGPT in Computer Programming.: ChatGPT in Computer Programming.,” *Mesopotamian Journal of Computer Science*, Jg. 2023, S. 8–16, Feb. 2023. DOI: [10.58496/MJCSC/2023/002](https://doi.org/10.58496/MJCSC/2023/002). Adresse: <https://mesopotamian.press/journals/index.php/%20cs/article/view/51>.
- [2] M. Welsh, “The End of Programming,” *Commun. ACM*, Jg. 66, Nr. 1, S. 34–35, Dez. 2022, ISSN: 0001-0782. DOI: [10.1145/3570220](https://doi.org/10.1145/3570220). Adresse: <https://doi.org/10.1145/3570220>.
- [3] D. M. Yellin, “The Premature Obituary of Programming,” *Commun. ACM*, Jg. 66, Nr. 2, S. 41–44, Jan. 2023, ISSN: 0001-0782. DOI: [10.1145/3555367](https://doi.org/10.1145/3555367). Adresse: <https://doi.org/10.1145/3555367>.
- [4] I. Shani, *Survey reveals AI’s impact on the developer experience*, Letzter Zugriff am 29.12.2023, Juni 2023. Adresse: <https://github.blog/2023-06-13-survey-reveals-ai-impact-on-the-developer-experience/>.
- [5] C. Paar und J. Pelzl, “Message Authentication Codes (MACs),” in *Kryptografie verständlich: Ein Lehrbuch für Studierende und Anwender*. Berlin, Heidelberg: Springer Vieweg, 2016, S. 363–375, ISBN: 978-3-662-49297-0. DOI: [10.1007/978-3-662-49297-0_12](https://doi.org/10.1007/978-3-662-49297-0_12). Adresse: https://doi.org/10.1007/978-3-662-49297-0_12.
- [6] A. Beutelspacher, J. Schwenk und K.-D. Wolfenstetter, *Moderne Verfahren der Kryptographie*. Springer Fachmedien Wiesbaden, 2015, Bd. 8, Zitiert aus Kapitel 2, ISBN: 978-3-8348-1927-7. DOI: [10.1007/978-3-8348-2322-9](https://doi.org/10.1007/978-3-8348-2322-9).
- [7] D. A. McGrew und J. Viega, “The Galois/Counter Mode of Operation (GCM),” Letzter Zugriff am 08.07.2024, 2005. Adresse: <https://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/gcm/gcm-spec.pdf>.
- [8] S. L. Nita und M. I. Mihailescu, “Java Cryptography Architecture,” in *Cryptography and Cryptanalysis in Java: Creating and Programming Advanced Algorithms with Java SE 17 LTS and Jakarta EE 10*. Berkeley, CA: Apress, 2022, S. 29–46, ISBN: 978-1-4842-8105-5. DOI: [10.1007/978-1-4842-8105-5_4](https://doi.org/10.1007/978-1-4842-8105-5_4). Adresse: https://doi.org/10.1007/978-1-4842-8105-5_4.
- [9] Oracle Corporation, *Java Platform, Standard Edition Security Developer’s Guide, Release 21*, Document Number: F79542-02. Letzter Zugriff am 08.07.2024, Okt. 2023. Adresse: <https://docs.oracle.com/en/java/javase/21/security/security-developer-guide.pdf>.
- [10] S. Nadi, S. Krüger, M. Mezini und E. Bodden, ““Jumping Through Hoops”: Why do Java Developers Struggle with Cryptography APIs?” In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, Ser. ICSE ’16, Austin, Texas: Association for Computing Machinery, 2016, S. 935–946, ISBN: 9781450339001. DOI: [10.1145/2884781.2884790](https://doi.org/10.1145/2884781.2884790). Adresse: <https://doi.org/10.1145/2884781.2884790>.

- [11] J. Weiss, “Chapter 1 - Understanding Java’s Cryptographic Architecture,” in *Java Cryptography Extensions*, Ser. The Morgan Kaufmann Series in Data Management Systems, J. Weiss, Hrsg., San Francisco: Morgan Kaufmann, 2004, S. 1–28, ISBN: 978-0-12-742751-5. DOI: [10.1016/B978-0127427515500098](https://doi.org/10.1016/B978-0127427515500098). Adresse: <https://www.sciencedirect.com/science/article/pii/B9780127427515500098>.
- [12] S. Krüger, J. Späth, K. Ali, E. Bodden und M. Mezini, “CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs,” *IEEE Transactions on Software Engineering*, Jg. 47, Nr. 11, S. 2382–2400, 2021. DOI: [10.1109/TSE.2019.2948910](https://doi.org/10.1109/TSE.2019.2948910).
- [13] R. Focardi, F. Palmari, M. Squarcina, G. Steel und M. Tempesta, “Mind Your Keys? A Security Evaluation of Java Keystores,” in *Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, 2018, ISBN: 1-891562-49-5. DOI: [http://dx.doi.org/10.14722/ndss.2018.23083](https://doi.org/10.14722/ndss.2018.23083). Adresse: https://iris.unive.it/retrieve/e4239ddc-92b3-7180-e053-3705fe0a3322/ndss2018_02B-1_Focardi_paper.pdf.
- [14] G. Briganti, “How ChatGPT works: a mini review,” *European Archives of Oto-Rhino-Laryngology*, Jg. 281, S. 1565–1569, 2023, Online published: 22 November 2023. DOI: [10.1007/s00405-023-08337-7](https://doi.org/10.1007/s00405-023-08337-7).
- [15] F. Acheampong, H. Nunoo-Mensah und W. Chen, “Transformer models for text-based emotion detection: a review of BERT-based approaches,” *Artificial Intelligence Review*, Jg. 54, S. 5789–5829, Dez. 2021. DOI: [10.1007/s10462-021-09958-2](https://doi.org/10.1007/s10462-021-09958-2).
- [16] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino und E. Poll, “An overview of JML tools and applications,” *International journal on software tools for technology transfer*, Jg. 7, S. 212–232, 2005. DOI: [10.1007/s10009-004-0167-4](https://doi.org/10.1007/s10009-004-0167-4).
- [17] G. T. Leavens, A. L. Baker und C. Ruby, “Preliminary Design of JML: A Behavioral Interface Specification Language for Java,” Techn. Ber., 1998, Copyright 1998-2005 Iowa State University. Letzter Zugriff am 08.07.2024. Adresse: <https://www.cs.ucf.edu/~leavens/JML/prelimdesign.pdf>.
- [18] D. R. Cok, “Does your software do what it should? User guide to specification and verification with the Java Modeling Language and OpenJML,” Techn. Ber., 2022, Letzter Zugriff am 08.07.2024. Adresse: <https://www.openjml.org/documentation/OpenJMLUserGuide.pdf>.
- [19] D. R. Cok, G. T. Leavens und M. Ulbrich, *Java Modeling Language (JML) Reference Manual*, 2nd, DRAFT July 1, 2022. Letzter Zugriff am 08.07.2024, Juli 2022. Adresse: https://www.openjml.org/documentation/JML_Reference_Manual.pdf.
- [20] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe und R. Stata, “Extended Static Checking for Java,” Ser. PLDI ’02, Berlin, Germany: Association for Computing Machinery, 2002, S. 234–245, ISBN: 1581134630. DOI: [10.1145/512529.512558](https://doi.org/10.1145/512529.512558). Adresse: <https://doi.org/10.1145/512529.512558>.
- [21] Bitkom e.V., *Organisierte Kriminalität greift verstärkt die deutsche Wirtschaft an*, Presseinformation, Letzter Zugriff am 08.07.2024, Sep. 2023. Adresse: <https://www.bitkom.org/Presse/Presseinformation/Organisierte-Kriminalitaet-greift-verstaerkt-deutsche-Wirtschaft-an>.
- [22] Bitkom e.V., *203 Milliarden Euro Schaden pro Jahr durch Angriffe auf deutsche Unternehmen*, Presseinformation, Letzter Zugriff am 08.07.2024, Aug. 2022. Adresse: <https://www.bitkom.org/Presse/Presseinformation/Wirtschaftsschutz-2022>.

- [23] J. White, S. Hays, Q. Fu, J. Spencer-Smith und D. C. Schmidt, “ChatGPT Prompt Patterns for Improving Code Quality, Refactoring, Requirements Elicitation, and Software Design,” in *Generative AI for Effective Software Development*, A. Nguyen-Duc, P. Abrahamsson und F. Khomh, Hrsg. Cham: Springer Nature Switzerland, 2024, S. 71–108, ISBN: 978-3-031-55642-5. DOI: [10.1007/978-3-031-55642-5_4](https://doi.org/10.1007/978-3-031-55642-5_4). Adresse: https://doi.org/10.1007/978-3-031-55642-5_4.
- [24] G. Antal, R. Vozár und R. Ferenc, *Toward a New Area of Rapid Development: Assessing GPT-4-Vision’s Capabilities in UML-Based Code Generation*, arXiv preprint, 2024. DOI: <https://doi.org/10.48550/arXiv.2404.14370>. arXiv: [2404.14370](https://arxiv.org/abs/2404.14370) [cs.SE].
- [25] L. S. Lo, “The CLEAR path: A framework for enhancing information literacy through prompt engineering,” *The Journal of Academic Librarianship*, Jg. 49, Nr. 4, S. 102–120, 2023, ISSN: 0099-1333. DOI: <https://doi.org/10.1016/j.acalib.2023.102720>. Adresse: <https://www.sciencedirect.com/science/article/pii/S0099133323000599>.
- [26] J. White, Q. Fu, S. Hays, M. Sandborn, C. Olea, H. Gilbert, A. Elnashar, J. Spencer-Smith und D. C. Schmidt, *A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT*, arXiv preprint, 2023. DOI: [10.48550/arXiv.2302.11382](https://doi.org/10.48550/arXiv.2302.11382). arXiv: [2302.11382](https://arxiv.org/abs/2302.11382) [cs.SE].
- [27] S. Ouyang, J. M. Zhang, M. Harman und M. Wang, *LLM is Like a Box of Chocolates: the Non-determinism of ChatGPT in Code Generation*, arXiv preprint, 2023. DOI: <https://doi.org/10.48550/arXiv.2308.02828>. arXiv: [2308.02828](https://arxiv.org/abs/2308.02828) [cs.SE].
- [28] D. R. Cok, “OpenJML: Software verification for Java 7 using JML, OpenJDK, and Eclipse,” in *Proceedings of the 1st Workshop on Formal Integrated Development Environment*, C. Dubois, D. Giannakopoulou und D. Méry, Hrsg., Ser. Electronic Proceedings in Theoretical Computer Science, Bd. 149, Grenoble, France: Open Publishing Association, Apr. 2014, S. 79–92. DOI: [10.4204/eptcs.149.8](https://doi.org/10.4204/eptcs.149.8). Adresse: <http://dx.doi.org/10.4204/EPTCS.149.8>.
- [29] Qwiet AI Team, *Keeping Secrets: A Deep Dive into Robust and Secure Environment Variable Management for Developers*, Letzter Zugriff am 06.07.2024, Nov. 2023. Adresse: <https://qwiet.ai/keeping-secrets-a-deep-dive-into-robust-and-secure-environment-variable-management-for-developers/>.
- [30] Bundesamt für Sicherheit in der Informationstechnik (BSI), “Kryptographische Verfahren: Empfehlungen und Schlüssellängen,” Bundesamt für Sicherheit in der Informationstechnik, Bonn, Germany, Techn. Ber. BSI TR-02102-1, Version 2024-01, Feb. 2024, Letzter Zugriff am 08.07.2024. Adresse: <https://www.bsi.bund.de/dok/TR-02102>.
- [31] Oracle, *SecureRandom (Java SE 17 & JDK 17)*, Letzter Zugriff am 08.07.2024, 2024. Adresse: <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/security/SecureRandom.html>.
- [32] T. Iwata, K. Ohashi und K. Minematsu, “Breaking and Repairing GCM Security Proofs,” in *Advances in Cryptology – CRYPTO 2012*, R. Safavi-Naini und R. Canetti, Hrsg., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, S. 31–49, ISBN: 978-3-642-32009-5. DOI: https://doi.org/10.1007/978-3-642-32009-5_3. Adresse: <https://eprint.iacr.org/2012/438>.

- [33] Computer Security Division, Information Technology Laboratory, “NIST Special Publication 800-38D: Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC,” National Institute of Standards und Technology (NIST), Gaithersburg, MD, Techn. Ber., Nov. 2007, U.S. Department of Commerce, Carlos M. Gutierrez, Secretary, National Institute of Standards and Technology, James M. Turner, Acting Director. Adresse: <https://doi.org/10.6028/NIST.SP.800-38D>.
- [34] S. Josefsson, “The Base16, Base32, and Base64 Data Encodings,” The Internet Society, RFC 4648, Okt. 2006, Letzter Zugriff am 08.07.2024. Adresse: <https://www.rfc-editor.org/rfc/inline-errata/rfc4648.html>.
- [35] OpenJML, *JML Tutorial - Calling methods in specifications (pure methods)*, Last Modified: Sat, 27 Jan 2024 21:25:58 GMT. © 2015 - 2024 OpenJML.org. Letzter Zugriff am 08.07.2024, 2024. Adresse: <https://www.openjml.org/tutorial/MethodsInSpecifications>.
- [36] Berliner Beauftragte für Datenschutz und Informationsfreiheit, *Smart Meter*, Letzter Zugriff am 08.07.2024. Adresse: <https://www.datenschutz-berlin.de/themen/wohnen/smart-meter/>.
- [37] B. Yetiştirilen, I. Özsoy, M. Ayerdem und E. Tüzün, *Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT*, arXiv preprint, 2023. DOI: <https://doi.org/10.48550/arXiv.2304.10778>. arXiv: [2304.10778](https://arxiv.org/abs/2304.10778) [cs.SE].
- [38] D. Sobania, M. Briesch, C. Hanna und J. Petke, “An Analysis of the Automatic Bug Fixing Performance of ChatGPT,” in *2023 IEEE/ACM International Workshop on Automated Program Repair (APR)*, 2023, S. 23–30. DOI: [10.1109/APR59189.2023.00012](https://doi.org/10.1109/APR59189.2023.00012).
- [39] M. A. Kuhail, S. S. Mathew, A. Khalil, J. Berengueres und S. J. H. Shah, ““Will I be replaced?” Assessing ChatGPT’s effect on software development and programmer perceptions of AI tools,” *Science of Computer Programming*, Jg. 235, S. 103 111, 2024, ISSN: 0167-6423. DOI: <https://doi.org/10.1016/j.scico.2024.103111>. Adresse: <https://www.sciencedirect.com/science/article/pii/S0167642324000340>.
- [40] F. A. Sakib, S. H. Khan und A. H. M. R. Karim, *Extending the Frontier of ChatGPT: Code Generation and Debugging*, arXiv preprint, 2023. DOI: <https://doi.org/10.48550/arXiv.2307.08260>. arXiv: [2307.08260](https://arxiv.org/abs/2307.08260) [cs.SE].
- [41] M. F. A. Khan, M. Ramsdell, E. Falor und H. Karimi, *Assessing the Promise and Pitfalls of ChatGPT for Automated Code Generation*, arXiv preprint, 2023. DOI: <https://doi.org/10.48550/arXiv.2311.02640>. arXiv: [2311.02640](https://arxiv.org/abs/2311.02640) [cs.SE].
- [42] Y. Liu, T. Le-Cong, R. Widyasari, C. Tantithamthavorn, L. Li, X.-B. D. Le und D. Lo, “Refining ChatGPT-Generated Code: Characterizing and Mitigating Code Quality Issues,” *ACM Trans. Softw. Eng. Methodol.*, Jg. 33, Nr. 5, Juni 2024, ISSN: 1049-331X. DOI: [10.1145/3643674](https://doi.org/10.1145/3643674). Adresse: <https://doi.org/10.1145/3643674>.
- [43] R. Khoury, A. R. Avila, J. Brunelle und B. M. Camara, “How Secure is Code Generated by ChatGPT?” In *2023 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, 2023, S. 2445–2451. DOI: [10.1109/SMC53992.2023.10394237](https://doi.org/10.1109/SMC53992.2023.10394237).
- [44] *JML Tutorial - Frame Conditions*, Last Modified: Sat, 27 Jan 2024 21:25:58 GMT © 2015 - 2024 OpenJML.org. Letzter Zugriff am 08.07.2024, 2015-2024. Adresse: <https://www.openjml.org/tutorial/FrameConditions>.

- [45] C. Liu, X. Bao, H. Zhang, N. Zhang, H. Hu, X. Zhang und M. Yan, *Improving ChatGPT Prompt for Code Generation*, arXiv preprint, 2023. DOI: <https://doi.org/10.48550/arXiv.2305.08360>. arXiv: 2305.08360 [cs.SE].
- [46] P. N. Pohlmann, “Electronic Code Book Mode (ECB-Mode),” Letzter Zugriff am 08.07.2024. Adresse: <https://norbert-pohlmann.com/glossar-cyber-sicherheit/electronic-code-book-mode-ecb-mode/>.
- [47] P. N. Pohlmann, “Cipher Block Chaining Mode (CBC-Mode),” Letzter Zugriff am 08.07.2024. Adresse: <https://norbert-pohlmann.com/glossar-cyber-sicherheit/cipher-block-chaining-mode-cbc-mode/>.