Mathematische Behandlung des RSA-Verfahrens und Entwicklung einer JAVA-Anwendung für die Schlüsselpaarerzeugung und (De-) Codierung nach RSA

Nicolas Walz

Facharbeit

Betreuender Lehrer: Herr Faßbender

Jahrgang 11/Q1, LK Informatik

INHALTSVERZEICHNIS

1 EINLEITUNG

Das Thema der Datensicherheit und die damit verbundenen Verschlüsselung von digitalen Daten spielen in unseren heutigen Zeit eine große Rolle. Ein Beispiel hierfür sind die soziale Medien, wie beispielsweise der Messanger WhatsApp, der eine sicherere Verschlüsselung der gesendeten und empfangenen Daten anbietet. Jeder will heutzutage seine Daten sicher und wenn möglich auch verschlüsselt ablegen. Auch Angriffe auf Unternehmen, wegen mangelnder Sicherheit und mangelnder Verschlüsselung der Daten, stehen in unseren heutigen Gesellschaft stark im Vordergrund.

Daher habe ich mich für meine Facharbeit im Fach Informatik für ein Thema aus der Kryptographie entschieden, das RSA-Verfahren. Genauer die mathematische Behandlung des RSA-Verfahrens und Entwicklung einer JAVA-Anwendung für die Schlüsselpaarerzeugung und (De-) Codierung nach RSA. Es ist einerseits ein sehr interessantes Thema, da es hohe mathematische Anteile besitzt und andererseits auch ein aktuelles Thema in Politik und Gesellschaft ist.

Im folgenden wird das RSA-Verfahren näher erläutert hinsichtlich der mathematischen Grundlagen und seiner Geschichte. Zusätzlich wird eine JAVA-Anwendung und eine GUI entwickelt und erläuternd hier dargestellt. Außerdem wird diese Anwendung mit Hilfe der Klassendokumentation des Projektes und eines Implementationsdiagrammes dargestellt.

2 DAS RSA-VERFAHREN

2.1 MATHEMATISCHE ERKLÄRUNG DER SCHLÜSSELGENERIERUNG

Für die spätere Ver- und Entschlüsselung werden zwei Keys, also zwei Schlüssel, benötigt. Diese zwei Keys werden *private Key* und *public Key* benannt. Daher ist das RSA-Verfahren ein Public-Key-Kryptosystem. Ein Public-Key-Kryptosystem, auch asymmetrisches Kryptosystem ist ein Verfahren, wo sich beide Schlüssel, also der *private Key* und der *public Key* zum Ver- und Entschlüsseln eines Klartextes unterscheiden. Nur wer diese beiden Schlüssel besitzt, ist in der Lage, einen Text zu ver- und entschlüsseln. Der *public Key* wird ausschließlich zum chiffrieren eines Klartextes verwendet, der *private Key* ausschließlich für das dechiffrieren des Geheimtextes.[1]

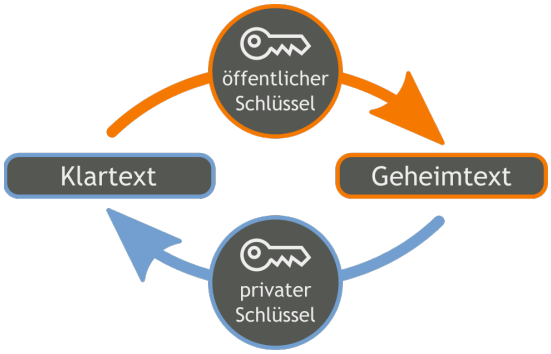


Abbildung 2: System des RSA[2]

Die Teilnehmer der geheimen Konversation können sich entscheiden, ob jeder selbst sein eigenes Schlüsselpaar generiert, oder ob eine gemeinsame Schlüsselpaarerzeugung stattfinden soll. Im weiteren Verlauf des Projektes, wurde sich für die gemeinsame Schlüsselpaargenerierung entschieden. [2]

Allgemein werden verschiedene mathematische Operationen verwendet, um die beiden Schlüssel zu erzeugen.

---

[1] vgl. http://www.cs.uni-potsdam.de/ti/lehre/06-Kryptographie/slides/slides-06.pdf Seite 8 (Angesehen am: 5.März 14:28 Uhr)

[2] Abbildung 2: vgl. https://de.wikipedia.org/wiki/Asymmetrisches_Kryptosystem#/media/File:Orange_blue_public_key_cryptography_de.svg (Eingesehen 06.03.17 18:30 Uhr)

Zunächst werden zwei große Primzahlen *p* und *q* gewählt. Diese Wahl generiert das erste Problem der Schlüsselpaarerzeugung. Das Finden zweier großen Primzahlen, beziehungsweise generell Primzahlen, ist eines der Hauptproblematiken der Mathematik. Im Normalfall werden bei dem RSA-Verfahren zwei Primzahlen mit einer Größe von etwa 100 bis 200 Bit verwendet. Da es keinen generellen Algorithmus zur Bestimmung großer Primzahlen gibt, muss nun mit Hilfe von Primzahlentests, wie beispielsweise dem Miller-Rabin-Test, gearbeitet werden. Hierbei wird eine große Zufallszahl erzeugt und getestet, ob dies eine Primzahl ist, wenn nicht, wird weiter gesucht. Nach der Generierung der Primzahlen *p* und *q*, wird das Produkt $n = p*q$ gebildet. Mit Hilfe dessen, dann die Eulersche Phi-Funktion $phi(n) = (p-1)(q-1)$ berechnet werden kann. [3]

phi(n) stellt die Anzahl der Elemente *x* in *Zn* mit Hilfe des größten gemeinsamen Teilers $ggt(n,x) = 1$ dar. Der letzte Schritt ist die Erzeugung und Berechnung von *d* und *e*. Um die Zahl *d* zu erhalten, wählt man eine zufällige Primzahl der ganzen Zahlen bis *n* mit Hilfe des euklidischen Algorithmus zur Bestimmung des größten gemeinsamen Teilers ggt(*d, phi(n)*) = 1. Um *e* zu berechnen, muss *e* invers zu *d* modulo *phi(n)* sein. Daraus ergibt sich:

$(d*e) \bmod phi(n) = 1$.

Daraus folgt dann das vollständig aufgebaute Public-Key-Kryptosystem RSA, mit dem *public Key* bestehend aus *(n, e)* und dem *private Key* bestehend aus *(p, q, phi(n), d)*. [4]

[3] vgl. Beutelspacher, Albrecht: „Kryptologie“ 10.Auflage, Seite 122 & Seite 129 bis 131

[4] vgl, Freiermuth, Hromkovic, Keller, Steffen „Einführung in die Kryptologie 1.Auflage Seite 326 bis 327

2.2 MATHEMATISCHE ERKLÄRUNG DER VER- UND ENTSCHLÜSSELUNG

Mit Hilfe der oben berechneten und erzeugten Schlüssel kann nun ein Klartext *m* verschlüsselt zu dem Geheimtext *c*, sowie der Geheimtext zu dem Klartext entschlüsselt werden.

Beim Verschlüsseln wird der Klartext *m* in Blocklängen unterteilt. Im weiteren Verlauf des Projektes wurde die Blocklänge „4“ gewählt, die Wahl der Blocklänge wird im Unterpunkt 2.4 näher erläutert.

Zunächst wird der Klartext *m* in Blöcke der Länge vier unterteilt. Wichtig hierbei ist, dass der Text als Zahl vorliegen muss, dies kann mit Hilfe der ASCII-Codierung möglich gemacht werden. Nun wird jeder Block als Binäre Codierung einer Zahl betrachtet. Mit der Formel $c = w^e \bmod n$ kann der passende Kryptotext *c* zum Klartext ermittelt werden. Für die Entschlüsselung des Geheimtextes wird die Formel $w = c^d \bmod n$ verwendet, um den Klartextblock *w* zu erhalten. [5]

2.3 ASPEKTE DER SICHERHEIT UND DER EFFIZIENZ

Grundsätzlich ist das RSA-Verfahren als sicher zu betrachten, wenn keines der Schlüsselelemente des *private Key* an eine dritte Person gelangt. Wenn jedoch Schlüsselelemente, wie *e* oder *n*, und der *public Key* bei einer außenstehenden Person bekannt sind, dann wäre es möglich, einen Geheimtext zu entschlüsseln. Dies ginge durch das Ziehen der „e-ten modularen Wurzel“ aus, dem aus Zahlen vorliegenden Geheimtext. Diese Option ist aber nur vorhanden, wenn dafür *d* ermittelt wird. Eine weitere Möglichkeit den Kryptotext zu entschlüsseln ist, wenn die dritte Person auch noch *phi(n)* kennt. Damit hätte er die Chance, durch das Berechnen des Euklidischen Algorithmus, *d* zu bestimmen. Wenn dieser *n* in dessen Primfaktoren *p* und *q* zerlegen würde, hätte er den kompletten *private Key* vorliegen. Diese Primfaktorzerlegung ist

[5] vgl, Freiermuth, Hromkovic, Keller, Steffen „Einführung in die Kryptologie 1.Auflage Seite 327

jedoch besonders schwer bei dem RSA-Verfahren. Normalerweise werden bei diesem Verfahren besonders große Primzahlen verwendet, wodurch dies eine Herausforderung stellt, diese in ihre Primfaktoren zu zerlegen. Die Mathematik stellt, nach heutiger Forschung, noch keinen effizienten Algorithmus dafür dar, um n zu faktorisieren. Dies ist momentan nur durch sehr ineffizientes systematisches Probieren möglich. Wenn also keine kleinen Primzahlen bei der Schlüsselpaarerzeugung verwendet werden und keine Elemente des *private Key* an die Öffentlichkeit kommen, ist nach heutiger Stand der Forschung nicht möglich einen Geheimtext ohne den *private Key* zu ermitteln. Zudem ist bis heute (Stand: 2015) keine andere Möglichkeit bekannt, um das Public-Key-Kryptosystem zu knacken. [6]

Zur Effizienz ist zu sagen, dass das RSA-Verfahren sehr Rechenintensiv ist, um ihn als Verschlüsselungssystem für Texte, oder deren gleichen, zu verwenden. [7] Daher ist es eher ineffizient.

2.4 WAHL DER BLOCKLÄNGEN

Allgemein gibt die Blocklänge die Anzahl der Zeichen an, die bei jeder RSA-Operation codiert wird. Daher ist es sinnvoll, große Blöcke zu wählen, da dadurch mehr Informationen mit einer Operation codiert werden. Das Verfahren wird dadurch zusätzlich noch effizienter, da so weniger Mathematische Operationen durchgeführt werden muss. Hinzu kommt, dass es bei kleinen Blocklängen möglich wird, den Kryptotext durch eine Substitutionsanalyse zu knacken. Daher wird das RSA-Verfahren auch sicherer, je größer der Nutzer die Blocklängen wählt. [8]

In diesem Projekt wurde die Blocklänge „4“ gewählt worden, da dies auch schon schwieriger wird zu knacken.

---

[6] vgl. Beutelspacher, Albrecht: „Kryptologie“ 10.Auflage, Seite 136 bis 140

[7] vgl. Beutelspacher, Albrecht: „Kryptologie“ 10.Auflage, Seite 140 bis 141

[8] vgl. https://www.cryptoportal.org/data/Asymmetrische%20Kryptologie%20am%20Beispiel%20RSA%20entdecken_v1.1.pdf Seite 24, Eingesehen: 02.04.2017 um 10:00 Uhr

2.5 GESCHICHTE UND ENTSTEHUNG

Das Public-Key-Kryptosystem RSA ist eines der bekanntesten Kryptosysteme. Dieses wurde 1977 von Ronald Rivest, Adi Shamir und Leonard Adleman entwickelt. Nach diesen es dann auch benannt wurde (**R**ivest, **S**hamir und **A**dleman). [9]

Die Grundlage, beziehungsweise die Vorraussetzungen des Durchbruchs für die Entwicklung des RSA von Rivest, Shamir und Adleman, war die Idee des asymmetrischen Kryptographiekonzeptes von Diffie und Hellmann. Dieses System besitzt jegliche Anforderungen einer asymmetrischen kryptographischen Methode. Hiermit wurde es möglich gemacht, nicht nur Texte zu de- und chiffrieren, sondern auch digitale Signaturen zu erstellen. [10]

2.6 VERWENDUNG

Hauptsächlich wird das asymmetrische Krypto-System RSA bei digitalen Signaturen, beziehungsweise Unterschriften verwendet. Jedoch findet man es auch häufig bei (Open-) PGP und S/MIME, welche vor allem beim E-Mail Verkehr verwendet werden. Zudem wird es bei kyprographischen Protokollen, wie SSH verwendet. [11]

[9] vgl. Beutelspacher, Albrecht: „Kryptologie“ 10.Auflage, Seite 121

[10] vgl. http://www.mathematik.uni-kassel.de/~koepf/Diplome/Inka-Tittel.pdf Seite 23, Eingesehen: 06.03.17 um 18:20 Uhr

[11] vgl. https://www.math.uni-bielefeld.de/~baumeist/RSA.pdf Seite 9, Eingesehen: 12.03.17 um 12:37 Uhr

3 IMPLEMENTATION IN JAVA

3.1 UMSETZUNG IN JAVA MIT IMPLEMENTATIONSDIAGRAMM

Grundsätzlich wird bei der Implementation des RSA-Verfahrens die Klasse *BigInteger* verwendet, da man hier außerhalb des Wertebereichs eines normaler Integer gelangt. Um Methoden dieser Klasse nutzen zu können, muss vorher diese importiert werden, durch den import-Befehl *import java.math.BigInteger* . Zudem macht die Klasse BigInteger es möglich, verschiedene mathematische Operationen durchzuführen, wie beispielsweise die Primzahlengenerierung mit Hilfe der Methode *probablePrime(int bitLenght, Random rnd)* oder die Berechnung des euklidischen Algorithmus zweier Zahlen mit *gcd(BigInteger val)*. Daraus folgend eignet sich diese Klasse gut für Kryptographische Anwendungen, die in JAVA umgesetzt werden. Umgesetzt wurde das Projekt mit Hilfe der Software BlueJ für die Implementation und NetBeans für die Erstellung einer GUI.

Das Projekt baut sich in drei Klassen auf, der Schlüsselgenerierung, der Codierung und der Master Klasse. Dies macht eine strukturierte Oberfläche und gute Bearbeitung möglich. Dabei hat jede Klasse eine bestimmte Aufgabe. Die Klasse *Schlüsselgenerierung* ist für die Generierung des *private* und *public Keys* zuständig. Um einen Buchstaben zu de- und codieren existiert die Klasse *Codierung*. Die *Master* Klasse vereint alle anderen Klassen, und kann ganze Texte ver- und entschlüsseln. Unter den drei Klassen bestehen gerichtete Assoziationen. Es existiert immer genau ein assoziiertes Objekt (siehe Implementationsdiagramm, Abbildung 3).

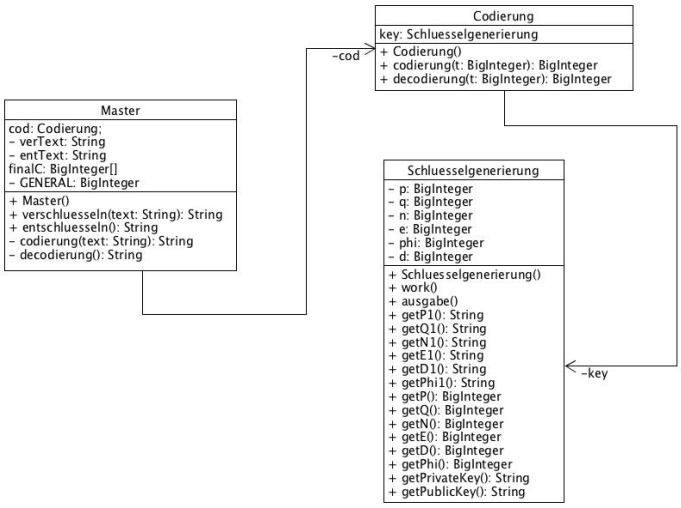


Abbildung 3: Implementationsdiagramm

3.2 METHODENBESCHREIBUNG

Bei der Methodenbeschreibung wird mit der Klasse *Schlüsselgenerierung* begonnen. Der Konstruktor führt die Methode *work()* aus, damit bei der Instanziierung eines neuen Objektes dieser Klasse direkt die Schlüssel generiert werden, ohne, dass der Benutzer eine zusätzliche Methoden eigens ausführen muss. Die Methode *work()* erzeugt den *private* und *public Key* des Kryptosystems. Dabei werden die Primzahlen *p* und *q* mit Hilfe der Methode *probablePrime (int bitLength, Random rnd)* generiert mit einer Bitlänge von 100 Bits, da standardmäßig man bei dem RSA-Verfahrens mit Primzahlen von 100 bis 200 Bits arbeitet, um eine sichere Anwendung garantieren zu können. Nun wird n mittels der Methode *multiply(BigInteger a, BigInteger b)* berechnet. Das Schlüsselelement *phi* wird nach dem Satz von Euler berechnet. Die Erzeugung von *d* ist jedoch aufwändiger, als der Rest der Schlüsselelemente. Grundsätzlich ist es eine zufällige Primzahl, jedoch ist es notwendig bedingt, dass der Euklidische Algorithmus von *d* und *phi* eins ergeben. Dies wird mittels einer if-Anweisung zunächst überprüft. Falls der gewünschte Fall nicht eintritt, wird solange (while-Schleife) eine zufällige

Primzahl generiert, bis der gewünschte Zustand eintritt. Eine weitere Methode von BigInteger wird verwendet, um *e* zu berechnen. Die Methode *modInverse(BigInteger a)* berechnet das inverse Modulo von *d* und *phi*. Daraus ergibt sich das letzte Schlüsselelement *e*. Nachfolgend zu der Methode *work()* ist eine Ausgabe aller Schlüsselelemente auf der JAVA-Konsole. Die restlichen Methoden dieser Klasse, sind Getter-Methoden zur Rückgabe der Schlüssel und der Elemente in Form von *BigIntegern* und *Strings*.

Der Konstruktor der Klasse *Codierung* instanziiert ein neues Objekt der eben beschriebenen Klasse *Schlüsselgenerierung*. Die zwei Methoden dieser Klasse können einzelne *BigInteger* de- und codieren. Zur Codierung wird der Parameter der Methode *codierung(BigInteger t)* mittels der Methode *modPow(BigInteger a, BigInteger b)* codiert, auf der Konsole ausgegeben und zurückgegeben. Die darauf folgende Methode *decodierung(BigInteger t)* decodiert, auch mittels der Methode *modPow(BigInteger a, BigInteger b)* und abschließend zurückgegeben.

Die letzte Klasse *Master* verbindet alle Klassen zu einer, die ganze Klartexte ver- und entschlüsseln kann. Im Konstruktor der Klasse wird ein neues Objekt der Klasse *Codierung* instanziiert, um dort einerseits auf die Schlüsselgenerierung zugreifen zu können, und andererseits, auch die Codierungsmethoden nutzen zu können. Die erste Methode *verschluesseln(String text)* codiert einen *String*, der als Parameter übergeben wird, mit Hilfe der ausgesonderten Methode *codierung(String text)*. Daher wird dort nur die Methode *codierung(String text)* ausgeführt mit *text* als Parameter. Als nächstes folgt die Methode *entschlüsseln()*, welche den Kryptotext *C* decodiert mittels der ebenfalls ausgesonderten Methode d*ecodierung()*. Die Methode, die die eigentlichen Verschlüsselungsoperationen durchführt folgt darauf. Zunächst wird geprüft, ob der als String übergebene Parameter eine *null* Referenz besitzt. Wenn ja wird die Methode beendet, wenn nein folgt der Rest der Methode. Darauf folgt die Unterteilung in die gewählte Blocklänge. Durch eine *for*-Schleife wird nun der Text durchlaufen und alle vier Zeichen den bis dahin durchgelaufenen Text in einem *Array* speichert. Nun wird das *String*-Array durchlaufen und das aktuelle Element in einem *String* gespeichert. Da in diesem Projekt mit *ASCII*-Codes gearbeitet wird und damit später die Möglichkeit besteht, die Klartextbuchstaben zurück zu ermitteln, wird der erste Buchstabe jedes

Blocks mit 256^0, der zweite mit 256^1, der dritte mit 256^2, der vierte mit 256^3 multipliziert. Doch zunächst muss notwendig überprüft werden, ob die Message *M* kleiner ist als *N*, nur wenn dieser Fall existiert, kann das Verfahren fortgeführt werden. Darauf folgend wird das *String-Array*, welches mittlerweile nur noch Zahlenwerte beinhaltet, in ein *BigInteger-Array* übernommen, um mit diesem dann die Mathematischen Operationen der Verschlüsselung durchführen zu können. Diese Codierung folgt dann mit Hilfe der Methode *codierung(BigInteger t)* des bereits initialisiertes Objekt der Klasse *Codierung*. Nun kommt es final nur noch zu einer Ausgabe des verschlüsselten Textes auf der Konsole und die Rückgabe dessen.

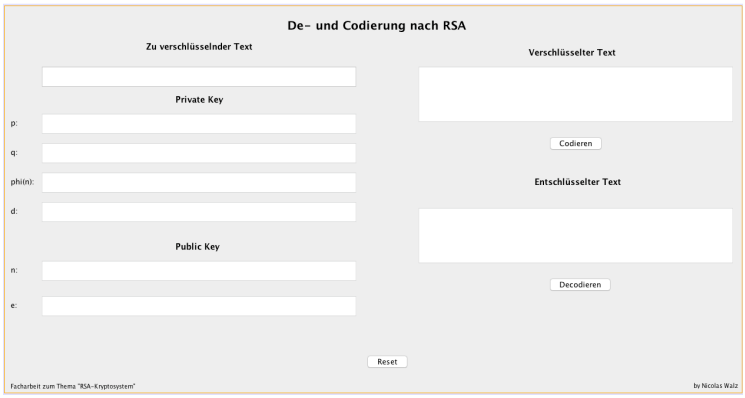
Die letzte Methode dieser Klasse stellt die Decodierung des Kryptotextes *C* dar. *C* wird ein einem *private* deklariertem Array gespeichert, und somit erst in ein neues *Array* übernommen. Nun wird jedes Element mittels der Methode *decodierung(BigInteger t)* der Klasse *codierung*. Um nun wieder auf die Klartextbuchstaben zu kommen, wird die folgende Operation durchgeführt. Zunächst wird das *modulo* (%) des aktuellen Elementes mit *256* berechnet. Dies liefert den ersten Klartextbuchstaben. Der Rest wird nun mittels der Formel *(aktuelles Element - (aktuelles Element % 256)) / 256* bestimmt. Mit diesem Rest wird nun weiter verfahren. Dieses Verfahren wird nun insgesamt dreimal durchgeführt. Somit erhält man numerisch folgend alle Klartextbuchstaben. Diese werden dann einem *String* angefügt, damit man abschließend einen *String* mit allen Klartextzeichen besitzt. Dieser wird final auf der Konsole ausgegeben und zurückgegeben.

3.3 ERSTELLUNG EINES GRAPHICAL USER INTERFACE (GUI)

Die graphische Benutzeroberfläche wird mit Hilfe der Software *NetBeans* erstellt und erzeugt. Um nun den oben beschriebenen Quelltext benutzertauglich zu machen, wird eine GUI benötigt, um eine einfache Benutzung mittels graphischer Benutzung möglich zu machen.

Um eine gute Benutzeroberfläche zu schaffen, gibt es einige Merkmale zu beachten. Zunächst muss diese für jedermann zu nutzen sein. Eine einfache Strukturierung und

eine Input/Output Infrastruktur bilden eine wichtige Grundlage, so dass der Nutzer lediglich einen Klartext eingeben muss, einen Button drückt, alles verschlüsselt wird, ein weiterer Button gedrückt werden muss, und der Text ist wieder decodiert.



In der für dieses Projekt angelegten GUI wird diese Input/Output Infrastruktur genutzt. Zudem macht diese es möglich, alle Schlüsselelemente ausgeben zu lassen. Das *JFrame* besteht aus 9 *JTextFields* (Panels), auf welchen einerseits Inhalte ausgegeben werden können, wie Schlüsselelemente und codierte beziehungsweise decodierte Texte, oder andererseits Inhalte eingegeben werden können, wie der zu verschlüsselnde Text. Jedoch sind alle Output-Panels nicht vom Nutzer editierbar (*setEditable(false)*), um eine Eingabe zu verhindern. Drei *JButtons* (Buttons) machen eine einfache Nutzung möglich, um einen eingegeben Text zu codieren, zu decodieren und um alles zurückzusetzen. 12 *jLabels* machen eine gut sichtbare Beschriftung möglich. Diese werden auch nicht während der Laufzeit der GUI geändert, und somit sind diese auch nicht in der implementativen Einbindung des normalen Quellcodes in die GUI gebraucht.

Nach dem das Design der graphischen Benutzeroberfläche vollzogen wird, muss der eigentliche Quelltextes des Projektes in diese eingebunden werden, damit auch Aktionen ausgeführt werden, wenn ein Button betätig wird. Mit Hilfe der *ActionPerformed(java.awt.event.ActionEvent evt)* Methode jedes Elementes des *JFrame*, kann nun die Einbeziehung durchgeführt werden.

Zunächst muss ein neues Objekt der Klasse Master initialisiert werden, welches im Konstruktor der GUI vollzogen wird. Damit ein eingegebener Klartext codiert werden kann, wird innerhalb der *ActionPerformed* Methode des „Codierungs-Button“ (hier:

jB03) gearbeitet. Zunächst wird sich der eingegebene Text in einem *String krypto* gespeichert. Nun werden die Schlüsselelemente des Kryptosystems in die jeweiligen Panels ausgegeben und so zuletzt der auch der *String krypto*.

Um nun den eben verschlüsselten Kryptotext *C* zu entschlüsseln, muss innerhalb der *ActionPerformed* Methode des „Decodierungs-Button“ (hier: jB02) geschrieben werden. Hierbei wird anfangs der Text entschlüsselt, und als *String* gespeichert. Final wird der erhaltene Klartext ausgegeben. Wenn nun der Nutzer die GUI zurücksetzen möchte, kann er dies mit der Betätigung des „Reset-Buttons“ machen. Daher muss auch hier in der jeweiligen *ActionPerformed* Methode (hier: jB01) gearbeitet werden. Hier wird zunächst ein neues Objekt der Klasse *Master* erzeugt, womit alle Inhalte des Kryptosystems zurückgesetzt wird. Nun müssen noch alle Inhalte der Panels gelöscht werden. Dabei wird die ausgesonderte Methode *work()* genutzt. Diese Methode setzt alle Inhalte aller *JTextFields* der GUI auf einen Leerstring, mittels *setText(String t)*. Somit wird die gesamte graphische Benutzeroberfläche zurückgesetzt.

Damit das Graphical User Interface auch ausgeführt werden kann, wird die main Methode benötigt.

```
public static void main(String args[]) {
    /* Set the Nimbus look and feel */
     Look and feel setting code (optional)

    /* Create and display the form */
    java.awt.EventQueue.invokeLater(() -> {
        new GUI().setVisible(true);
    });
}
```

Diese macht es möglich, dass bei der Ausführung der GUI die Sichtbarkeit des erstellten Objektes auf den booleschen Wert *true* gesetzt wird, damit die GUI sichtbar gemacht wird.

3.4 FINALE UMSETZUNG

Somit arbeiten das Graphical User Interface mit dem vorher bereits implementierten Projektes eng miteinander. So benötigt der Nutzer ausschließlich ein JAVA-Applet, um kryptographische Operationen mit Hilfe des RSA-Kryptosystems durchführen zu können. Also muss er sich nicht näher mit einer Programmiersprache (hier: JAVA) befassen und benötigt auch nicht das dahinter stehende Implementationsprojekt. Hinzu kommt, dass bereits bei der Implementation der Quelltext auf die Benutzeroberfläche angepasst wurde, wie beispielsweise durch getter-Methoden oder ähnliches.

4 ANHANG

4.1 JAVA KLASSENDOKUMENTATION DES PROJEKTES

Class Master

```
java.lang.Object
  └Master
```

```
public class Master
extends java.lang.Object
```

Die Klasse Master ist Bestandteil der Facharbeit zum Thema "Mathematische Behandlung des RSA-Verfahrens und Entwicklung einer JAVA-Anwendung für die Schluesselpaargenerierung und (De-Codierung) nach RSA". Aufgabe dieser Klasse ist es, einen Text zu codieren und zu decodieren. author: Nicolas Walz Version: 22. März 2017

Field Summary	
(package private) Codierung	cod
(package private) java.math.BigInteger[]	finalC

Constructor Summary
Master()

Method Summary	
java.lang.String	entschluesseln()
java.lang.String	verschluesseln(java.lang.String text)

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

cod

```
Codierung cod
```

finalC

```
java.math.BigInteger[] finalC
```

Constructor Detail

Master

```
public Master()
```

Method Detail

entschluesseln

```
public java.lang.String entschluesseln()
```

---

verschluesseln

```
public java.lang.String verschluesseln(java.lang.String text)
```

Class Codierung

```
java.lang.Object
  └Codierung
```

```
public class Codierung
extends java.lang.Object
```

Die Klasse Codierung ist Bestandteil der Facharbeit zum Thema "Mathematische Behandlung des RSA-Verfahrens und Entwicklung einer JAVA-Anwendung für die Schluesselpaargenerierung und (De-Codierung) nach RSA" Aufgabe dieser Klasse ist Codierung und die Decoderierung eines Textes mit Hilfe der Klasse Schluesselgenerierung, die einen public und eine private Key generieren. author: Nicolas Walz Version: 23. Februar 2017

Field Summary

(package private) Schluesselgenerierung	**key**

Constructor Summary

Codierung()

Method Summary

java.math.BigInteger	**codierung**(java.math.BigInteger t)
java.math.BigInteger	**decodierung**(java.math.BigInteger t)

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

key

```
Schluesselgenerierung key
```

Constructor Detail

Codierung

```
public Codierung()
```

Method Detail

codierung

```
public java.math.BigInteger codierung(java.math.BigInteger t)
```

---

decodierung

```
public java.math.BigInteger decodierung(java.math.BigInteger t)
```

Class Schluesselgenerierung

```
java.lang.Object
  └Schluesselgenerierung
```

```
public class Schluesselgenerierung
extends java.lang.Object
```

Die Klasse Schluesselgenerierung ist Bestandteil der Facharbeit zum Thema "Mathematische Behandlung des RSA-Verfahrens und Entwicklung einer JAVA-Anwendung für die Schluesselpaargenerierung und (De-Codierung) nach RSA" Aufgabe dieser Klasse ist die Generierung des public und des private Key nach RSA. Hilfsmethoden werden ausgesondert in die Klasse "Hilfsmethoden". author: Nicolas Walz Version: 23. Februar 2017

Constructor Summary

Schluesselgenerierung()

Method Summary

void	**ausgabe**()
java.math.BigInteger	**getD**()
java.math.BigInteger	**getE**()
java.math.BigInteger	**getN**()
java.math.BigInteger	**getP**()
java.lang.String	**getPrivateKey**()
java.lang.String	**getPublicKey**()
java.math.BigInteger	**getQ**()

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Schluesselgenerierung

```
public Schluesselgenerierung()
```

Method Detail

ausgabe

```
public void ausgabe()
```

---

getD

```
public java.math.BigInteger getD()
```

---

getE

```
public java.math.BigInteger getE()
```

---

getN

```
public java.math.BigInteger getN()
```

---

getP

```
public java.math.BigInteger getP()
```

---

getPrivateKey

```
public java.lang.String getPrivateKey()
```

---

getPublicKey

```
public java.lang.String getPublicKey()
```

---

getQ

```
public java.math.BigInteger getQ()
```

4.2 GLOSSAR

- ASCII-Code: American Standard Code for Information Interchange, Zeichenkodierung [12]
- GUI: Graphical User Interface, graphische Benutzeroberfläche
- PGP/Open-PGP: Pretty Good Privacy, Programm zur Verschlüsselung [13]
- S/MIME Secure/Multipurpose Internet Mail Extensions, Verschlüsselung und Signieren von E-Mails [14]
- SSH: Secure Shell, Programm für verschlüsselte Netzwerkverbindungen [15]

[12] vgl. https://de.wikipedia.org/wiki/American_Standard_Code_for_Information_Interchange, Eingesehen am 04.April 2017 um 17:45 Uhr

[13] https://www.datenschutzzentrum.de/selbstdatenschutz/internet/pgp/wasdas.htm, Eingesehen am 04.April 2017 um17:48 Uhr

[14] https://de.wikipedia.org/wiki/S/MIME, Eingesehen am 04.April 2017 um 17:50 Uhr

[15] https://de.wikipedia.org/wiki/Secure_Shell, Eingesehen am 04.April 2017 um 17:52 Uhr

5 LITERATURVERZEICHNIS

[1] Beutelspacher, A.: Kryptologie, Wiesbaden, 2015, 10. Auflage

[2] Freiermuth, K., Hromkovic, J., Keller, L., Steffen, B.: Einführung in die Kryptologie, Wiesbaden, 2010, 1. Auflage

[3] http://www.cs.uni-potsdam.de/ti/lehre/06-Kryptographie/slides/slides-06.pdf, Zugriff am 05.03.2017 um 14:28 Uhr

[4] http://www.mathematik.uni-kassel.de/~koepf/Diplome/Inka-Tittel.pdf, Zugriff am 06.03.2017 um 18:20 Uhr

[5] https://www.math.uni-bielefeld.de/~baumeist/RSA.pdf, Zugriff am 12.03.2017 um 12:37 Uhr

[6] https://www.cryptoportal.org/data/Asymmetrische%20Kryptologie%20am%20Beispiel%20RSA%20entdecken_v1.1.pdf , Zugriff am 02.04.2017 um 10:00 Uhr

[7] https://de.wikipedia.org/wiki/American_Standard_Code_for_Information_Interchange, Zugriff am 04.04.2017 um 17:45 Uhr

[8] https://www.datenschutzzentrum.de/selbstdatenschutz/internet/pgp/wasdas.htm, Zugriff am 04.04.2017 um 17:48 Uhr

[9] https://de.wikipedia.org/wiki/S/MIME, Zugriff am 04.04.2017 um 17:50 Uhr

[10] https://de.wikipedia.org/wiki/Secure_Shell, Zugriff am 04.04.2017 um 17:52 Uhr

[11] http://www.spektrum.de/fm/912/thumbnails/Datenverschluesselung_fotolia77657014_TomaszZajda.jpg.2161370.jpg, Zugriff am 06.03.2017 um 18:51 Uhr (Abbildung 1 auf Titelseite)

[12] https://de.wikipedia.org/wiki/Asymmetrisches_Kryptosystem#/media/File:Orange_blue_public_key_cryptography_de.svg, Zugriff am 06.03.2017 um 18:30 Uhr (Abbildung 2)

6 ERKLÄRUNG DER EIGENSTÄNDIGEN VERFASSUNG

Versicherung der selbständigen Erarbeitung

Ich versichere, dass ich die vorliegende Arbeit einschließlich evtl. beigefügter Zeichnungen, Kartenskizzen, Darstellungen u. ä. m. selbstständig angefertigt und keine anderen als die angegebenen Hilfsmittel benutzt habe. Alle Stellen, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen sind, habe ich in jedem Fall unter genauer Angabe der Quelle deutlich als Entlehnung kenntlich gemacht.

_____, den _____

(Ort) (Datum)

_____

(Unterschrift)