Grundlagen der Informatik 1
Wintersemester 2013/2014
Programmierprojekt

Version 1.1 — 20. Februar 2014

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Inhaltsverzeichnis

1 Einleitung

> *On a day nearly 4,000 years ago, in a town called Menet Khufu bordering the thin ribbon of the Nile, a master scribe sketched out the hieroglyphs that told the story of his lord's life—and in so doing he opened the recorded history of cryptology.* David Kahn: „The Codebreakers" [2]

In diesem Praktikum erhalten Sie einen Einblick in die historische Welt der Kryptologie, die Kunst bzw. Wissenschaft des Verschlüsselns (Kryptographie) und Brechens von Verschlüsselung (Kryptoanalyse). Die Entwicklung der Kryptographie geht nahezu zurück bis zur Entwicklung der Schrift selbst. Mit der Möglichkeit, Botschaften in Textform auszutauschen, kam auch der Wunsch auf, dies in geheimer Art und Weise zu tun – beispielsweise, um kriegswichtige Informationen vor dem Feind geheim zu halten oder um eine Nachricht vertraulich mit einem Geschäftspartner auszutauschen. Zweck der Verschlüsselung war es (und ist es bis heute), eine Nachricht mithilfe einer geheimen Zusatzinformation – dem sogenannten *Schlüssel* – so zu modifizieren (bzw. zu *verschlüsseln*), dass diese nur von jemandem gelesen (*entschlüsselt*) werden kann, der über die gleiche Geheiminformation verfügt[1]. Personen, die die verschlüsselte Nachricht (der sogenannte *Ciphertext* oder *Geheimtext*) während des Austauschs (beispielsweise in Form eines Briefs oder Funkspruchs) mitlesen sollen– ohne Kenntnis des Schlüssel – keinerlei Informationen über den Inhalt der eigentlichen Nachricht (den sogenannten *Plaintext* oder *Klartext*) erfahren können. Zu diesem Zweck wurden im Laufe der Geschichte und werden auch heute noch Verschlüsselungsverfahren (auch Chiffre, englisch *Cipher*) entwickelt, um Nachrichten zu ver- und entschlüsseln.

Sie werden in diesem Praktikum einige der historischen Verschlüsselungsverfahren kennenlernen und implementieren (hierbei wird kein kryptographisches Vorwissen vorausgesetzt). Gleichzeitig werden Sie sich aber auch in die Position der Angreifer begeben und versuchen die implementierten Verfahren mittels sogenannter *Kryptoanalyse* zu brechen, denn: So sicher die historischen Verfahren vor einigen Jahrzehnten, Jahrhunderten oder Jahrtausenden waren – den Angriffen mit einer Rechenkapazität heutiger Computer halten sie allesamt nicht stand.

Tot ist die Kryptographie (und auch die Kryptoanalyse) als Wissenschaft damit allerdings nicht, im Gegenteil: In Zeiten weltweiter Kommunikation über das Internet ist Verschlüsselung von Daten und Informationen heute wichtiger denn je und wird täglich millionenfach eingesetzt. Die Entwicklung und Kryptoanalyse moderner Verschlüsselungsverfahren ist allerdings eine andere Geschichte. . . [2]

[1] Wenn beide Seiten, das heißt Sender und Empfänger, den gleichen Schlüssel verwenden, spricht man auch von einem symmetrischen Verschlüsselungsverfahren. In den 1970ern kamen Verfahren auf, bei denen Empfänger und Sender unterschiedliche Schlüssel verwenden. Hierbei spricht man von asymmetrischen Verfahren. In diesem Praktikum werden wir jedoch ausschließlich symmetrische Verschlüsselungsverfahren betrachten.

[2] Bei weitergehendem Interesse an der Geschichte und modernen Entwicklung der Kryptographie und Kryptoanalyse (oder auch für ein tieferes Verständnis der in diesem Praktikum vorgestellten Verfahren und Analysen) empfehlen wir Ihnen die spannend geschriebenen und unterhaltsamen Bücher von David Kahn [2] und Simon Singh [3] – und im Rahmen Ihres weiteren Studiums natürlich die Vorlesung „Einführung in die Kryptographie" am Fachbereich Informatik. Außerdem bietet die Webseite CrypTool-Online [1] die Möglichkeit, die meisten der in diesem Praktikum besprochenen Verschlüsselungs- und Kryptoanalyse-Verfahren im Browser auszuprobieren.

2 Organisatorisches

- Zur Orientierung sind die Aufgaben in diesem Praktikum in zwei Bereiche unterteilt: Die **grundlegenden Aufgaben** (Teil I) zu Beginn umfassen 40 Punkte sowie 5 Bonuspunkte[1] (mit der Angabe „Bonus: x Punkte“ markiert), die **weiterführenden Aufgaben** (Teil II) weitere 95 Punkte. Welche der gestellten Aufgaben Sie bearbeiten und in welcher Reihenfolge ist Ihnen überlassen, allerdings sind die grundlegenden Aufgaben naturgemäß einfacher als die weiterführenden und seien Ihnen daher als Einstieg empfohlen.
- Im gesamten Praktikum können Sie **maximal 100 Punkte** erzielen, darüber hinausgehende Punkte werden „abgeschnitten“. Zum Bestehen des Praktikums sind **mindestens 40 Punkte** nötig, egal aus welchem Bereich, allerdings müssen die **5 Punkte für Formalia aus Abschnitt 4 auf jeden Fall** erreicht werden, um das Projekt zu bestehen.
- Die **Bearbeitungszeit** des Praktikums beginnt am **Montag, 17.02.2014**, mit Veröffentlichung dieser Aufgabenstellung und endet am **Freitag, 28.02.2014, um 18:00 Uhr**.
- Die **Bearbeitung** des Projekts erfolgt in Gruppen, die im Moodle-Kurs[2] zum Projekt im Lernportal Informatik eingeteilt wurden.
- Die **Abgabe** des Projekts erfolgt im Moodle-Kurs zum Projekt und muss bis spätestens **Freitag, 28.02.2014, um 18:00 Uhr** erfolgen. *Verspätete Abgaben können nicht bewertet werden!* Sie haben die Möglichkeit, in Moodle Zwischenstände Ihrer Arbeit als *Entwurf* hochzuladen. Für Details zum Abgabeformat siehe Abschnitt 4.
- Ihre Abgabe wird von Tutoren bewertet und Ihre Gruppe im Rahmen eines ca. 25-minütigen **Testats** über Ihre Abgabe geprüft, wobei Ihnen Fragen zu allen Teilen Ihrer Implementierung gestellt werden können. Die Testate finden am **Montag, 03.03.2014, und Dienstag, 04.03.2014**, statt. An welchem Tag und um welche Uhrzeit Ihre Gruppe geprüft wird, entnehmen Sie bitte dem Infodokument über die Testatszeiten, das rechtzeitig im Moodle-Kurs veröffentlich wird. Die Ergebnisse des Testats werden in Moodle eingetragen/bekanntgegeben.
- Für das Testat ist die **Anwesenheit der gesamten Gruppe** notwendig, *nicht anwesende Studenten können das Projekt nicht bestehen*! Sollte es Ihnen nicht möglich sein, Ihren Testattermin wahrzunehmen, dann melden Sie sich bitte frühzeitig bei uns, um einen zu Ausweichtermin vereinbaren können. Bitte beachten Sie, dass für die Verschiebung Ihres Termins ein triftiger Grund vorliegen muss.
- Während der Bearbeitungszeit steht Ihnen werktägig der **C-Pool** im Informatikgebäude (S202/C005) zur Verfügung; er ist ausschließlich für das GdI1-Projekt reserviert. Zudem findet im C-Pool während der Bearbeitungszeit werktägig von 9 bis 17 Uhr eine **Betreuung durch Tutoren** statt, die Sie mit Fragen zum Projekt kontaktieren können. Die Tutoren können Hilfestellung zum Verständnis des Projekts und zur Herangehensweise geben, werden aber natürlich keine Schritt-für-Schritt-Anleitungen zur Programmierung liefern. Eine Übersicht, welcher Tutor zu welchem Termin im C-Pool anwesend ist, finden Sie im Moodle-Kurs. Sie können diese Information nutzen, um mit dem Tutor, der Sie testieren wird, vorab über einen Zwischenstand Ihrer Bearbeitung zu sprechen.
- Für Teamarbeit in der Softwarentwicklung bieten sich diverse **Tools zur Unterstützung** an, deren Erläuterung den Rahmen dieses Projekts sprengen würden. Als besonders nützliches Hilfsmittel wollen wir an dieser Stelle aber die Nutzung einer Software zur Versionsverwaltung empfehlen.[3] Die Rechnerbetriebsgruppe (RBG) des Fachbereichs Informatik bietet Ihnen – sofern Sie über einen RBG-Login verfügen – im Rahmen der bereitgestellten Infrastruktur die Möglichkeit, die populären Versionsverwaltungsysteme Subversion/SVN[4] oder git[5] zu nutzen (alle nötigen Erläuterung hierzu finden Sie im Wiki[6] der RBG). Sie können diese Tools allerdings auch ohne Zugriff auf die RBG-Infrastruktur einsetzen.
- Bei **organisatorischen Fragen oder Problemen** wenden Sie sich bitte per E-Mail an `gdi1@informatik.tu-darmstadt.de`.

[1] Bonuspunkte gibt es für einzelne Teilaufgaben die für die Gesamtbearbeitung des Praktikums nicht weiter relevant sind aber eine interessante Vertiefung der Aufgabe darstellen.

[2] `https://moodle.informatik.tu-darmstadt.de/course/view.php?id=322`

[3] Die nötige kurze Einarbeitungszeit in die Nutzung von Versionsverwaltungssoftware sollte sich bereits in diesem kurzen Praktikum von zwei Wochen, in jedem Fall aber im Laufe Ihres Studiums mehr als bezahlt machen.

[4] `http://subversion.apache.org/`

[5] `http://git-scm.com/`

[6] `https://support.rbg.informatik.tu-darmstadt.de/wiki/de/doku/computerhilfe/svn-git`

3 Plagiarismus

Der Fachbereich Informatik misst der Einhaltung der Grundregeln der wissenschaftlichen Ethik großen Wert bei. Zu diesen gehört auch die strikte Verfolgung von Plagiarismus.[1]

Mit der Abgabe Ihrer Projekts bestätigen Sie, dass Ihre Gruppe alleinige Autoren des gesamten Materials sind. Falls Ihnen die Verwendung von Fremdmaterial gestattet war, so müssen Sie dessen Quellen deutlich zitieren.

[1] https://www.informatik.tu-darmstadt.de/de/sonstiges/plagiarismus/

Teil I.

Grundlegende Aufgaben (40 Punkte, Bonus: 5 Punkte)

4 Formale und allgemeine Anforderungen (5 Punkte)

Bevor wir mit der inhaltlichen Beschreibung des Projekts beginnen, zunächst einige Worte zum *Abschluss des Projekts* und den *formalen und allgemeinen Anforderungen* an Ihre Abgabe.

- Komprimieren Sie **am Ende des Projekts** (s. Abschnitt 2) Ihr komplettes Eclipse-Projekt als Zip-Datei: File – Export – General/Archive File. Stellen Sie sicher, dass Ihre Zip-Datei alle projektrelevanten Dateien beinhaltet (zumindest die Dateien `.classpath` und `.project` sowie den Quelltext (Verzeichnis `src`) und alle class-Dateien (Verzeichnis `bin`)). Benennen Sie Ihre Zip-Datei mit dem Namen `Gruppe###.zip` und Ihr Projekt mit dem Namen `Gruppe###`, wobei Sie `###` durch Ihre (mit führenden `0`-en auf drei Stellen aufgefüllte) Gruppennummer ersetzen (z.B. `Gruppe042.zip` bzw. `Gruppe042` für Gruppennummer 42). Laden Sie Ihre Zip-Datei als Abgabe in Moodle hoch.
- In Moodle wird Ihnen ein **Template-Projekt** für Eclipse bereitgestellt (verwendet Java 7), dass als Basis für Ihre Projektimplementierung dient. Importieren Sie die Datei `template.zip` über File – Import – General/Existing Projects into Workspace – archive file. Alle vorgegebenen Interfaces sowie einige Tests, die Sie zur Überprüfung Ihrer Implementierung nutzen können, sind über die Datei `framework.jar` eingebunden. Die Tests für ein von Ihnen zu implementierendes Interface `XYZ` liegen stets im gleichen Package in der Klasse `TemplateXYZTests`.
  Die Datei `framework.jar` kann im Laufe des Praktikums noch aktualisiert werden (z.B. um fehlerhafte Testfälle zu korrigieren). Stellen Sie daher unbedingt sicher, dass Sie die **aktuellste Version aus Moodle** verwenden und beachten Sie diesbezügliche Ankündigungen im Forum. Verändern Sie die vorgegebenen Interfaces nicht, diese werden im Testat durch eine frische Kopie der Datei `framework.jar` ersetzt.
- Nutzen Sie bei der **Benennung des Projekts** sowie von Klassen, Interfaces etc. keine Umlaute. Kommentieren Sie alle Klassen, Felder und Methoden mit JavaDoc mit Ausnahme von Methoden, die eine dokumentierte Interface-Methode implementieren. Im Hinblick auf das Testat sollten Sie zudem im eigenen Interesse sicherstellen, dass der Code ausreichend dokumentiert ist.
- Im Rahmen des Projekts wird neben der Funktionalität Ihrer Implementierung auch die gewählte **Modellierung** bewertet. Achten Sie dafür darauf, Klassen in eine geeignete Hierarchie zu bringen und Redundanzen, bspw. durch Vererbung, zu vermeiden. Lesen Sie sich hierfür zu Beginn des Praktikums die gesamte Aufgabenstellung zumindest grob durch, um einen Überblick über die Zusammenhänge der zu implementierende Klassen zu erhalten.
- Behandeln Sie **auftretende Fehler** (z.B. durch ungültige Parameter) durch das Werfen geeigneter Exceptions mit aussagekräftigen Fehlermeldungen. Nutzen Sie hierfür, sofern vorhanden, die von uns vorgegebenen Fehlerklassen oder ergänzen Sie bei Bedarf eigene.
- Unsere **vorgegebenen Tests** decken nur einige grundlegende Aspekte der zu implementierenden Funktionalität ab. **Testen Sie** daher Ihren Code im eigenen Interesse nicht nur an den von uns geforderten Stellen. Implementieren Sie dazu eigene, separate Testklassen. Verändern Sie *nicht* unsere vorgegebenen Testklassen, diese werden beim Testat durch eine frische Kopie ersetzt.
- Nach Abschluss des Projekts wird Ihre Abgabe von einem Tutor in einem ca. 25-minütigen **Testat** geprüft (siehe Abschnitt 2). Im Rahmen des Testats muss jedes Gruppenmitglied in der Lage sein, Fragen zur Erläuterung jedes beliebigen Teils Ihres abgegebenen Codes (insbesondere nicht nur den von ihm implementierten Code) zu beantworten. Jedes Gruppenmitglied sollte also vorab sicherstellen, dass es alle Aspekte Ihrer Abgabe verstanden hat.

Bepunktung der formalen Anforderungen (müssen zum Bestehen des Projekts in jedem Fall erfüllt sein!):

1. Zip-Datei mit korrektem Namen eingereicht (1 Punkt)
2. Gültiges und vollständiges Eclipse-Projekt mit korrektem Namen enthalten, das importiert werden kann (1 Punkt)
3. Es werden keine Umlaute für Projekt-, Klassen-, Interfacenamen etc. genutzt (1 Punkt)
4. Alle eigenen Klassen, Interfaces etc. sind sinnvoll in (vorgegebene / ergänzte) Packages eingeordnet (1 Punkt)
5. Alle Klassen des Projekts kompilieren (mit den unveränderten vorgegebenen Interfaces und Klassen) (1 Punkt)

5 Grundlagen (5 Punkte)

Um konkrete Verschlüsselungsverfahren implementieren zu können, müssen wir uns erst um die Vereinbarung und Implementierung einiger grundlegender Elemente in diesem Zusammenhang kümmern. In diesem Abschnitt legen wir daher zunächst fest, was ein Verschlüsselungsverfahren (in unserer Java-Modellierung) ist und wie die Objekte aufgebaut sind, die es ver- und entschlüsseln soll.

Anmerkung: Die gesamte Implementierung des Projekts liegt im Package `de.tudarmstadt.gdi1.project`. Bei Verweisen auf Interfaces, Klassen etc. benennen wir diese daher nur relativ zu diesem Pfad. Eine als `sub.Something` referenzierte Klasse liegt also in `de.tudarmstadt.gdi1.project.sub.Something`.

Anmerkung: In den meisten vorgegebenen Aufgabenstellungen fordern wir Sie auf, ein vorgegebenenes Interface zu implementieren. Wie die Methoden des Interfaces in einem solchen Fall zu implementieren sind, entnehmen Sie bitte – sofern nicht weiter erläutert – jeweils dem JavaDoc des Interfaces.

5.1 Das `Cipher`-Interface

In unserer Hierarchie an Verschlüsselungsverfahren befindet sich an oberster Stelle das Interface `cipher.Cipher`:

```
public interface Cipher {
        public String encrypt(String text);
        public String decrypt(String text);
}
```

Dieses Interface soll von allen im Laufe des Projekts vorgestellten Verschlüsselungsverfahren implementiert werden und definiert zwei Methoden `encrypt` und `decrypt` zum Ver- und Entschlüsseln, die jeweils einen String als Eingabe (Plain- bzw. Ciphertext) erhalten und den (ver- bzw. entschlüsselten) String (Cipher- bzw. Plaintext) zurückgeben.

5.2 Alphabete (4 Punkte)

Während ein String in Java eine beliebige Sequenz von UTF-16-kodierten Zeichen ist, arbeiten (historische) Verschlüsselungsverfahren praktisch immer über einem deutlich kleineren Zeichensatz (*Alphabet* genannt) wie z.B. den 26 Kleinbuchstaben `{a, b, c, d, ..., z}`. Dementsprechend interpretieren auch wir unsere Strings nur über einem beschränkten, definierten Alphabet, dessen Funktionalität durch das Interface `alphabet.Alphabet` beschrieben wird. Ein Alphabet definiert dabei eine (geordnete) Liste von Zeichen (in der jedes enthaltene Zeichen nur einmal vorkommt), die als Zeichen in einem Nachrichtenstring (für Ver- oder Entschlüsselung) infrage kommen und erlaubt einige nützliche Operationen auf dieser.

Implementieren Sie das Interface `Alphabet`. Achten Sie dabei (entsprechend den Anforderungen aus Abschnitt 4) darauf, dass Sie ungültige Eingaben bei der Erstellung eines Alphabets abfangen. Bedenken Sie hierbei, dass ein Alphabet keine doppelten Zeichen erlaubt und werfen Sie im Fehlerfall geeignete Exceptions.

Bepunktung: Vollständige und korrekte Implementierung des Interfaces `Alphabet` (4 Punkte).

5.3 Eine Fabrik als technisches Hilfsmittel (1 Punkt)

Aus technischen Gründen benötigen wir im Rahmen der Tests die Möglichkeit, die von Ihnen implementierten Klassen zu instanziieren (ohne deren genaue Position und Namen in der Package-Hierarchie zu kennen). Zu diesem Zweck müssen Sie das Interface `Factory`[1] implementieren, indem Sie nach und nach die dort enthaltenen Methoden zur Instanziierung der zu erstellenden Klassen implementieren.

[1] Der Name leitet sich vom sogenannten *Factory-Pattern* (`https://en.wikipedia.org/wiki/Factory_method_pattern`) ab. Sie müssen dieses allerdings nicht verstehen, um die hier benötigte Funktionalität zu implementieren.

Beginnen Sie nun damit, indem Sie das Interface `Factory` durch eine eigene Klasse `FactoryIM` im Package `de.tudarmstadt.gdi1.project`[2] implementieren und die Methode zum Erstellen eines Alphabets mit Inhalt füllen. Methoden, die Objekte zurückgeben sollen, deren Klassen Sie noch nicht implementiert haben können Sie bis auf weiteres einfach **`null`** zurückgeben lassen.

Bepunktung: Korrekte Implementierung der `Factory`-Methode zur Erstellung eines Alphabets (1 Punkt).

[2] Sollten Sie Ihre `Factory`-implementierende Klasse anders nennen oder an anderer Stelle ablegen wollen, müssen Sie für die bereitgestellten Tests in der Klasse `test.Main` die Zeile `TestCore.FACTORYPATH = "de.tudarmstadt.gdi1.project.FactoryIM";` entsprechend anpassen.

6 Monoalphabetische Substitution (15 Punkte, Bonus: 2 Punkte)

Klassische Verschlüsselungsverfahren bedienen sich vor allem zweier Techniken zum Verschleiern einer Nachricht: der *Substitution* und der *Transposition*.

Bei der Transposition einer Nachricht werden die darin enthaltenen Zeichen umsortiert, um so die Bedeutung der Nachricht zu verwischen. In diesem Praktikum betrachten wir Transpositionschiffren nicht weiter.

Eine Substitutionschiffre ersetzt hingegen die auftretenden Zeichen in einer Nachricht durch andere Zeichen anhand einer Substitutionstabelle die jedem Zeichen des Klartextalphabets ein Zeichen des Chiffretextalphabets zuweist. Zum Entschlüsseln eines Textes wird die Substitutionstabelle dann einfach in umgekehrter Richtung angewandt. Die ersetzen Zeichen stammen dabei meistens aus dem gleichen Alphabet, dies ist aber nicht zwingend so. Beispielsweise würde der Klartext „`hallowelt`“ zum Chiffretext „`xh!!oal!w`“ verschlüsselt, wenn man folgende Substitutionstabelle zugrunde legt:

Klartext-Zeichen	a	e	h	l	o	t	w
Chiffretext-Zeichen	h	l	x	!	o	w	a

Beachten Sie, dass sich die Klartext- und Chiffretext-Alphabete unterscheiden können (hier taucht `!` nicht im Klartext-Alphabet auf, `e` nicht im Chiffretext-Alphabet) und dass Klartext-Buchstaben bei einer Substitution auch auf den gleichen Buchstaben im Chiffretext abgebildet werden können (wie hier im Fall des Zeichens `o`).

In diesem Abschnitt betrachten wir zunächst *monoalphabetische* Substitutionsverfahren, d.h. solche, bei denen es genau eine Substitutionstabelle gibt und jeder Buchstabe einer Nachricht auf die gleiche Art und Weise ersetzt wird. Später im Praktikum werden Sie auch *polyalphabetische* Verfahren (vgl. Abschnitt 8) kennenlernen, bei denen je nach Position eines Buchstabens in einer Nachricht eine andere Substitutionstabelle zum Einsatz kommt.

6.1 Gemeinsame Basisklassen (6 Punkte)

Bevor wir uns konkrete historische monoalphabetische Substitutionschiffren anschauen, versuchen wir zunächst, die Gemeinsamkeiten solcher Verfahren in unserer Klassenhierarchie nachzuvollziehen.

6.1.1 `SubstitutionCipher` (3 Punkte, Bonus: 1 Punkt)

Implementieren Sie zunächst das Interface `cipher.substitution.SubstitutionCipher`, welches das `Cipher`-Interface um Methoden zum Übersetzen von Buchstaben in Abhängigkeit ihrer Position ergänzt. Da eine Substitutionschiffre an sich nicht festlegt, wie diese Übersetzung erfolgen soll, macht es Sinn, das Interface in einer *abstrakten* Klasse zu implementieren, welche die tatsächliche Spezifikation der Übersetzungsmethoden aus dem Interface `SubstitutionCipher` offen lässt. Die Methoden zum Ver- und Entschlüsseln von Nachrichten (aus dem Interface `Cipher`) können Sie jedoch bereits in Ihrer abstrakten Klasse implementieren.

Abstrakte Klassen lassen sich nicht ohne weiteres testen. Eine Möglichkeit ist es eine Testimplementierung der abstrakten Klasse zu erstellen und mit dieser zu testen. Schöner und differenzierter lassen sich abstrakte Klassen mit Mocking Frameworks, wie z.B. Mockito (`https://code.google.com/p/mockito/`) testen. Hierbei übernimmt Mockito die Kontrolle, wann immer Methoden noch nicht implementiert sind und es wird in den Tests spezifiziert, wie sich das Objekt verhalten soll (beispielsweise, welche Wert zurückgegeben werden soll). Ein kleines Beispiel für die Arbeit mit Mockito finden Sie in den Tests zu `SubstitutionCipher`.[1] Wie Sie sehen, müssen Sie in der Factory lediglich das Klassenobjekt zurückgeben. Den Rest regelt Mockito.

Bepunktung:

[1] Beachten Sie, dass die Methoden `encrypt` und `decrypt` in Ihrer `SubstitutionCipher`-Implementierung als **`final`** deklariert sein müssen, damit die Tests mit Mockito funktionieren.

1. korrekte Implementierung der Verschlüsselung (1 Punkt)
2. korrekte Implementierung der Entschlüsselung (1 Punkt)
3. korrekte Implementierung des Interfaces als abstrakte Klasse (1 Punkt)
4. eigene Tests für die Abstrakte Implementierung von `SubstitutionCipher` mit einem Mockingframework (Bonus: 1 Punkt)

6.1.2 `MonoalphabeticCipher` (3 Punkte)

Implementieren Sie nun das Interface `cipher.substitution.monoalphabetic.MonoalphabeticCipher` in einer Klasse, die von Ihrer abstrakten, `SubstitutionCipher`-implementierenden Klasse erbt. Überlegen Sie sich, wie Sie hier die Übersetzungsmethoden (`translate` und `reverseTranslate`) konkret implementieren können und erstellen Sie einen Konstruktor für die Klasse, sodass Sie die entsprechende Methode des `Factory`-Interfaces für einen `MonoalphabeticCipher` ausfüllen können.

Achten Sie in Ihrer Implementierung auf eine geeignete Fehlerüberprüfung und werfen Sie ggf. entsprechende Exceptions.

Bepunktung:

1. korrekte Implementierung der Übersetzungsmethoden (2 Punkte)
2. korrekte Implementierung der `Factory`-Methode inkl. geeignetem Konstruktor (1 Punkt)

6.2 Die Caesar-Chiffre (4 Punkte)

Eine der bekanntesten historischen Verschlüsselungsverfahren ist die *Caesar-Chiffre*, benannt – der Überlieferung nach – nach ihrem Erfinder, dem römischen Feldherrn Gaius Julius Cäsar. Die Caesar-Chiffre stellt das erste dokumentierte Verfahren dar, dass in der Geschichte zur Übertragung militärischer Botschaften eingesetzt wurde.

Das Prinzip dieser monoalphabetischen Substitutionchiffre ist simpel: Man wähle eine beliebige Zahl K zwischen 1 und 26 (für das klassische Alphabet der Länge 26 – für ein beliebiges Alphabet der Länge n wählt man eine ganze Zahl aus $[1, n]$) als Schlüssel. Zum Verschlüsseln einer Nachricht „verschiebt“ man nun jeden Buchstaben um K Buchstaben im Alphabet nach rechts.

Über einem Alphabet `{a, b, ..., z}` und bei einer Verschiebung um $K = 3$ Buchstaben (dem von Cäsar gewählten Schlüssel, abgeleitet vom Buchstaben C als *dritten* im Alphabet) würde also ein `a` durch ein `d`, ein `b` durch ein `e`, usw. und schließlich ein `z` durch ein `c` substituiert werden (vgl. Abbildung 6.1).

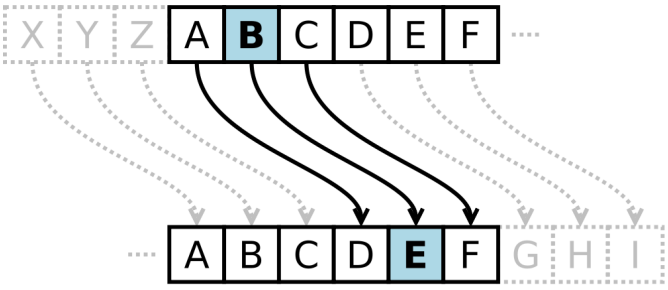

Abbildung 6.1.: Illustration der Caesar-Chiffre.[2]

Implementieren Sie nun das Interface `cipher.substitution.monoalphabetic.Caesar` in einer Klasse, die von Ihrer `MonoalphabeticCipher`-implementierenden Klasse erbt. Nutzen Sie hierbei die bereits vollständig implementierten Ver- und Entschlüsselungsmethoden dieser Klasse und überlegen Sie sich, wie Sie die Caesar-Chiffre allein durch eine geeignete Instanziierung der neuen Klasse in ihrem Konstruktor implementieren können. Implementieren Sie zudem die

[2] Quelle: *Wikimedia Commons*, `https://commons.wikimedia.org/wiki/File:Caesar3.svg` (14.02.2014).

Instanziierungsmethode für die Caesar-Chiffre in Ihrer `Factory`-Implementierung. Achten Sie auch hier auf eine geeignete Fehlerbehandlung.

Zur Unterstützung ihrer Implementierung betrachten Sie das Interface `utils.Utils`. Dieses enthält eine Methode `shiftAlphabet`, die für Ihre Implementierung hilfreich sein sollte. Wir werden die weiteren Methoden in `Utils` im weiteren Verlauf des Praktikums noch verwenden. Implementieren Sie für die Lösung der aktuellen Aufgabe daher lediglich die Methode `shiftAlphabet` und geben Sie für die weiteren Methoden einfach **`null`** zurück. Implementieren Sie auch die zu `utils.Utils` gehörende Factory Methode. Wie immer ist es hilfreich zusätzlich eigene Tests zu schreiben.

Bepunktung:

1. korrekte Implementierung der Caesar-Chiffre als Erbe der `MonoalphabeticCipher`-Instanz (1 Punkt)
2. Benutzung von `utils.Utils.shiftAlphabet` (1 Punkt)
3. überschreiben Sie keine bereits implementierten Methoden (1 Punkt)
4. korrekte Implementierung der `Factory`-Methode (1 Punkt)

6.3 Monoalphabetische Substitution mit Schlüsselwort (5 Punkte)

Da die Caesar-Chiffre nur so viele verschiedene Schlüssel erlaubt, wie Zeichen im Alphabet vorkommen (üblicherweise 26), ist leicht zu sehen, dass eine Durchprobieren aller möglichen Schlüssel schnell den Klartext hinter einer verschlüsselten Nachricht zum Vorschein bringt. (Eine detaillierte Kryptoanalyse der Caesar-Chiffre folgt in Kürze in Abschnitt 7.)

Um den Schlüsselraum (also die Menge der möglichen Schlüssel) zu vergrößern, kann man anstelle der einfachen Verschiebung des Alphabets über ein beliebiges Austauschen von Buchstaben durch andere nachdenken. Da eine komplett zufällige Vertauschung (vgl. Abschnitt 6.4) für Menschen allerdings schwierig zu merken ist, ist der praktikable Zwischenschritt (sowohl historisch als auch in diesem Praktikum), die Vertauschung auf einem *Schlüsselwort* aufzubauen.

Bei der monoalphabetischen Substitution mit Schlüsselwort wählt man sich zunächst ein beliebiges Wort – oder allgemeiner eine (leicht zu merkende) Zeichenkette – und entfernt alle doppelt auftretenden Buchstaben. So wird beispielsweise aus dem Ausgangswort `gdipraktikum` das Schlüsselwort `gdipraktum`. In der Substitutionstabelle ersetzt man nun die ersten Buchstaben des Alphabets durch die Buchstaben des Schlüsselworts:

Klartext-Zeichen	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
Chiffretext-Zeichen	g	d	i	p	r	a	k	t	u	m																

und füllt dann die restlichen Buchstaben des Chiffretext-Alphabets mit den verbleibenden Buchstaben in umgekehrter alphabetischer Reihenfolge auf:

Klartext-Zeichen	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
Chiffretext-Zeichen	g	d	i	p	r	a	k	t	u	m	z	y	x	w	v	s	q	o	n	l	j	h	f	e	c	b

Durch diese Methode der Erstellung der Substitutionstabelle wird eine immens größere Anzahl an möglichen Schlüsseln als bei der Caesar-Chiffre erreicht, was die Kryptoanalyse – wie wir in Abschnitt 9 sehen werden – dieses Verfahrens im Vergleich bereits deutlich schwerer macht.

Implementieren Sie nun das Interface `cipher.substitution.monoalphabetic.KeywordMonoalphabeticCipher` in einer Klasse, die von Ihrer `MonoalphabeticCipher`-implementierenden Klasse erbt. Überlegen Sie sich hierzu, wie Sie die monoalphabetische Substitution mit Schlüsselwort allein durch eine geeignete Instanziierung der neuen Klasse in ihrem Konstruktor implementieren können. Implementieren Sie zudem die Instanziierungsmethode für die Chiffre in Ihrer `Factory`-Implementierung. Achten Sie auch hier auf eine geeignete Fehlerbehandlung.

In dieser Aufgabe sollte Ihnen eine weitere Methode des zuvor beschriebenen `utils.Utils` Interfaces helfen: `utils.Utils.reverseAlphabet`. Diese Methode bekommt als Parameter ein Alphabet und soll ein neues Alphabet in umgekehrter Reihenfolge zurückgeben.

Bepunktung:

1. korrekte Implementierung der monoalphabetischen Substitution mit Keyword als Erbe der `MonoalphabeticCipher`-Instanz (2 Punkte)
2. überschreiben Sie keine bereits implementierten Methoden (1 Punkt)

3. korrekte Implementierung von `utils.Utils.reverseAlphabet` (1 Punkt)
4. korrekte Implementierung der `Factory`-Methode (1 Punkt)

6.4 Zufällige Monoalphabetische Substitution (Bonus: 1 Punkt)

Um die kryptographisch stärkste Form der monoalphabetischen Substitution zu implementieren, die auf einer zufällig vertauschten Substitutionstabelle basiert, benötigen wir keine zusätzliche Klasse mehr – Ihre Implementierung des `MonoalphabeticCipher`-Interfaces erlaubt bereits eine solche Instanziierung mittels eines zufälligen Alphabets. Was hierzu noch fehlt ist die Erzeugung eines solchen.

Um Zufall in Ihr Programm zu bringen können Sie auf die Java Objekte `java.util.Random` (bzw. `java.security.SecureRandom`). Zweitere Klasse ist ein starker Zufallszahlengenerator wie er auch in modernen Kryptographischen Verfahren eingesetzt werden könnte. `java.util.Random` ist ein eher einfacher Zufallszahlengenerator aufbauend auf Kongruenzgeneratoren. Dieser sollte normalerweise nicht für kryptographische Verfahren verwendet werden, ist allerdings performanter als `java.security.SecureRandom`. Weitere Informationen zu den beiden Zufallszahlengeneratoren finden Sie im JavaDoc.

Im `utils.Utils` Interface finden Sie die Method `randomizeAlphabet`. Dieses erhält als Eingabe ein Alphabet und erstellt ein neues Alphabet mit gleichen Buchstaben jedoch in zufälliger Reihenfolge.

Bepunktung:

1. korrekte Implementierung von `randomizeAlphabet` unter Verwendung von `Random` oder `SecureRandom` (Bonus: 1 Punkt)

7 Kryptoanalyse der Caesar-Chiffre (15 Punkte, Bonus: 3 Punkte)

Wir wechseln nun gedanklich die Seiten und betrachten einige grundlegende Ansätze der Kryptoanalyse. Das zu Grunde liegende Szenario ist denkbar simpel: Stellen Sie sich vor, Sie haben eine mit der Caesar-Chiffre verschlüsselte Nachricht abgefangen, von der Sie weder den Klartext noch den genutzten Schlüssel kennen. Sie wollen diese Nachricht nun entschlüsseln, am besten sogar den Schlüssel (für weitere Nachrichten) extrahieren. Was tun Sie?

7.1 Von Known-Ciphertext-Angriffen und Häufigkeitsanalysen

Das soeben geschilderte Szenario nennt man in der Kryptographie bzw. Kryptoanalyse einen *Known-Ciphertext-Angriff*: Sie kennen als Angreifer also lediglich den Chiffretext, haben aber keine Information über den Plaintext oder den benutzten Schlüssel (letzteres ist natürlich immer der Fall, denn mit einem bekannten Schlüssel können Sie natürlich immer entschlüsseln).

In diesem Praktikum bilden wir diese Art von Angriffen im Interface `analysis.KnownCiphertextAnalysis` ab. Ziel einer Kryptoanalyse ist es in diesem Praktikum immer (und häufig auch in der Praxis), den zur Verschlüsselung des anzugreifenden Chiffretext verwendeten Schlüssel zu extrahieren – im Fall der Caesar-Chiffre also den Integer-Wert für die Verschiebung des Alphabets.

Wie Sie an den im Interface `KnownCiphertextAnalysis` deklarierten Methoden erkennen, ist die Ausgangslage für die Kryptoanalyse allerdings nicht ganz so aussichtslos wie zunächst geschildert. In der Regel hat man bei die Kryptoanalyse etwas Hintergrundinformation, auf die man zurückgreifen kann – beispielsweise indem man weiß, in welcher Sprache die Klartextnachricht verfasst ist.

Für unsere ersten Schritte in der Kryptoanalyse wollen wir uns einen konkrete Eigenschaft von Sprachen zunutze machen: In jeder Sprache haben (auf längere Texte betrachtet) die in der Sprache genutzten Buchstaben eine gewisse Häufigkeit, mit der Sie in einem Text vorkommen. In einem deutschen Text beispielsweise kommt der Buchstabe `e` im Allgemeinen viel häufiger vor als der Buchstabe `q`. Für jede Sprache kann man – auf Basis einer größeren Sammlung von Texten in der Sprache – daher eine charakteristische *Häufigkeitsverteilung* der Buchstaben bestimmen. Abbildung 7.1 zeigt beispielhaft die Häufigkeitsverteilung der Buchstaben in der englischen Sprache, Tabelle 7.1 listet die Häufigkeitsverteilung in der deutschen Sprache auf.

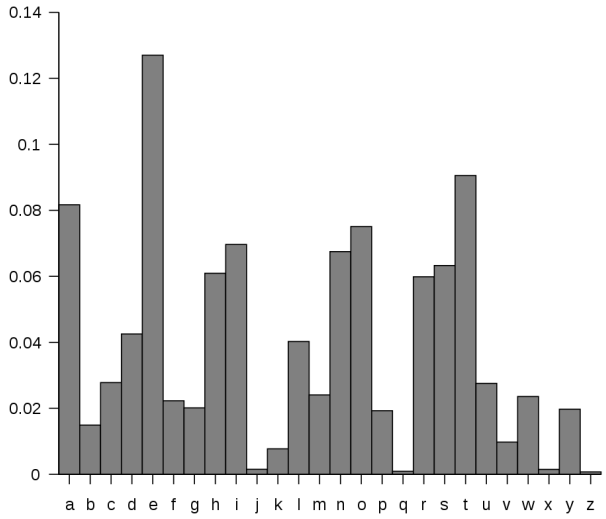

Abbildung 7.1.: Häufigkeitsverteilung der Buchstaben in englischen Texten.[1]

Zeichen	a	b	c	d	e	f	g	h	i	j	k	l	m
Häufigkeit	6,5%	1,9 %	2,7%	5,1%	17,4%	1,7%	3,0%	4,8%	7,6%	0,3%	1,4%	3,4%	2,5%
Zeichen	n	o	p	q	r	s	t	u	v	w	x	y	z
Häufigkeit	9,8%	2,6%	0,7%	0,01%	7,0%	7,3%	6,2%	4,3%	0,8%	1,9%	0,03%	0,04%	1,1%

Tabelle 7.1.: Häufigkeitsverteilung der Buchstaben in deutschen Texten.[2]

Die Idee hinter unserer ersten Kryptoanalyse der Caesar-Chiffre ist nun, diese charakteristische Häufigkeitsverteilung einer Sprache zu nutzen, um einen Chiffretext zu entschlüsseln bzw. den verwendeten Schlüssel zu extrahieren.[3] Als zentrale Erkenntnis hinter diesem auf der *Häufigkeitsanalyse* der Buchstaben im Chiffretext basierenden Angriff ist, dass die Caesar-Chiffre (als monoalphabetische Substitutionschiffre) gleiche Klartextbuchstaben immer zu gleichen Chiffretextbuchstaben verschlüsselt. Dadurch bleibt die charakteristische Häufigkeitsverteilung in ihrer Ausprägung auch auf dem Chiffretext erhalten – wobei nun die entsprechenden Häufigkeiten anderen Buchstaben zugeordnet ist. Diese Tatsache macht man sich bei der Häufigkeitsanalyse zunutze, indem man die Häufigkeitsverteilung der Buchstaben im Chiffretext bestimmt und dann versucht, den häufigsten Buchstaben im Chiffretext auf den häufigsten Buchstaben in der Sprache (im Deutschen bspw. das `e`) abzubilden, den zweithäufigsten auf den zweithäufigsten Buchstaben in der Sprache (im Deutschen das `n`) usw.

Im Fall der Caesar-Chiffre wird dieser Angriff zusätzlich noch sehr stark dadurch vereinfacht, dass die Substitutionstabelle lediglich eine Verschiebung des Alphabets um K Buchstaben ist – K ist dabei der verwendete Schlüssel. Weiß man also, dass der häufigste Buchstabe in der Sprache das `e` ist und berechnet man, dass im Chiffretext das `k` der häufigste Buchstabe ist, so ist es sehr wahrscheinlich, dass der Schlüssel $K = 6$ ist, da genau dieser Schlüssel die Abbildung von `e` auf `k` hervorruft.

7.2 Das `Distribution`-Interface, Teil 1 (8 Punkte)

Bevor wir eine Häufigkeitsanalyse implementieren können müssen wir zunächst modellieren, was wir unter einer Häufigkeitsverteilung über einem Alphabet verstehen. Hierzu dient in der Klassenhierarchie das Interface `alphabet.Distribution`, welches diverse Methoden zur Abfrage und Manipulation von Häufigkeitsverteilungen über einem Alphabet erlaubt. Für unsere simple Häufigkeitsanalyse für die Caesar-Chiffre brauchen wir allerdings nur einen kleinen Teil des Interfaces.[4]

Implementieren Sie nun in einer neuen Klasse das Interface `Distribution`, wobei Sie zunächst nur Folgendes implementieren müssen:

- Der Konstruktor erhält Parameter `Alphabet source`, `String text` und **`int`** `ngramsize` und soll die Häufigkeit der Buchstaben aus dem Alphabet `source` im String `text` zählen (nicht im Alphabet vorkommende Zeichen sollen nicht gezählt werden). Den Parameter `ngramsize` können Sie an dieser Stelle ignorieren.
- Die Methode **`double`** `getFrequency(String)` – hier zunächst nur für Strings der Länge 1, also praktisch: für Zeichen – soll die Häufigkeit dieses Zeichens zurückgeben.
- Die Methode `List<String> getSorted(`**`int`**` length)` soll die Strings der Länge length (hier zunächst nur für die Länge 1) in einer Liste, sortiert nach der Häufigkeit ihres Auftretens, zurückgeben.

Füllen Sie zudem die Instanziierungsmethode des `Factory`-Interfaces für eine `Distribution` aus.

Bepunktung:

1. korrekte Implementierung des Konstruktors (4 Punkte)
2. korrekte Implementierung von `getFrequency` (1 Punkt)
3. korrekte Implementierung von `getSorted` (2 Punkte)

[1] Quelle: *Wikimedia Commons*, `https://commons.wikimedia.org/wiki/File:English_letter_frequency_%28alphabetic%29.svg` (14.02.2014).

[2] Quelle: *Wikipedia*, `https://en.wikipedia.org/wiki/Letter_frequency` (14.02.2014).

[3] Die erste Beschreibung der Häufigkeitsanalyse von Chiffren (und damit auch das erste Dokument zur Kryptoanalyse überhaupt) geht übrigens zurück auf den arabischen Philosoph und Mathematiker al-Kindi aus dem 9. Jahrhundert.

[4] Sie an dieser Stelle auf eine volle Implementierung verzichten und diese in den entsprechenden späteren Aufgabenteilen ergänzen bzw. anpassen. Falls Sie spätere Erweiterungen schon jetzt in Ihrer Implementierung berücksichtigen wollen, finden Sie die entsprechenden Erläuterungen zur weiteren Funktionalität von `Distribution` im Abschnitt 9.2.

4. korrekte Implementierung der `Factory`-Methode (1 Punkt)

7.3 Known-Ciphertext-Angriff auf die Caesar-Chiffre mittels Häufigkeitsanalyse (4 Punkte, Bonus: 3 Punkte)

Mithilfe der Implementierung des `Distribution`-Interfaces können Sie nun die erste Kryptoanalyse für die Caesar-Chiffre angehen. Implementieren Sie dafür das Interface `analysis.caesar.CaesarCryptanalysis`, wobei Sie hier zunächst nur die aus `analysis.KnownCiphertextAnalysis` geerbte Methode `knownCiphertextAttack(String, Distribution)` ausfüllen müssen. Gehen Sie bei deren Implementierung vor wie in Abschnitt 7.1 beschrieben, d.h. ermitteln Sie das häufigste Zeichen im Chiffretext und bestimmen Sie den Schlüssel als Differenz zwischen der Position des häufigsten Zeichens des Chiffretexts und der Position des häufigsten Zeichen der gegebenen `Distribution` im Alphabet. Implementieren Sie zudem die Instanziierungsmethode des `Factory`-Interfaces für `CaesarCryptanalysis`.

Optional können Sie sich überlegen, dass Sie die Genauigkeit Ihrer Analyse (gerade bei kurzen Chiffretexten) steigern, wenn Sie nicht nur das häufigste Zeichen bei der Berechnung mit einbeziehen, sondern auch das zweit- oder sogar dritt-häufigste Zeichen, und das in Ihrer Implementierung berücksichtigen.

Bepunktung:

1. Implementierung von `knownCiphertextAttack(String, Distribution)` durch Häufigkeitsanalyse des häufigsten Zeichens (3 Punkte)
2. korrekte Implementierung der `Factory`-Methode (1 Punkt)
3. Berücksichtigung des zweithäufigsten Zeichens (Bonus: 2 Punkte)
4. Berücksichtigung des dritthäufigsten Zeichens (Bonus: 1 Punkt)

7.4 Wissen ist Macht: Known-Plaintext-Angriffe auf Caesar (3 Punkte)

In den vorangegangenen Abschnitten haben wir gesehen, dass sich die Caesar-Chiffre selbst in einem Known-Ciphertext-Angriff relativ leicht brechen lässt, wenn man die Häufigkeitsverteilung des zugrunde liegenden Plaintexts kennt. Dies ist allerdings eine denkbare schlechte Ausgangssituation für eine Kryptoanalyse. In der Praxis ist es relativ wahrscheinlich, dass man für Teile von verschlüsselten Nachrichten oder sogar ganze Nachrichten den zugehörigen Klartext kennt und nun vor der Aufgabe steht, mithilfe dieses Wissens den benutzten Schlüssel zu extrahieren um in der Folge alle weiteren Nachrichten entschlüsseln zu können. Ein Angriff in einem solchen Szenario nennt man einen *Known-Plaintext-Angriff*.

Im Fall der Caesar-Chiffre ist leicht zu sehen, dass bereits eine bekannte Kombination aus Klartextbuchstabe und verschlüsseltem Chiffretextbuchstabe ausreicht, um den benutzten Schlüssel eindeutig zu bestimmen – er ist schlicht die Differenz der Positionen dieser beiden Buchstaben im Alphabet – und damit die Caesar-Chiffre vollständig zu brechen. Weiß man also beispielsweise, dass der Buchstabe `x` in einem Chiffretext mit dem Buchstaben `p` im Klartext korrespondiert (und arbeitet man über dem üblichen Alphabet mit 26 Buchstaben, in dem `x` an Position 24 und `p` an Position 16 steht), muss der verwendete Caesar-Schlüssel $K = 24 - 16 = 8$ sein.

Implementieren Sie entsprechend der beschriebenen Vorgehensweise die Methode `knownPlaintextAttack(String, String, Alphabet)` des Interfaces `analysis.caesar.CaesarCryptanalysis` in Ihrer Kryptoanalyse-Klasse für Caesar. Wie oben beschrieben sollte diese bereits für Strings ab der Länge 1 stets den korrekten Caesar-Schlüssel zurückgeben.

Bepunktung:

1. korrekte Implementierung von `knownPlaintextAttack(String, String, Alphabet)` (2 Punkte)
2. Known-Plaintext-Angriff ist (für nicht-leere Eingabestrings) immer erfolgreich (1 Punkt)

Teil II.

Weiterführende Aufgaben (95 Punkte)

8 Polyalphabetische Substitution (12 Punkte)

Nachdem wir in Abschnitt 6 *monoalphabetische* Substitutionsverfahren kennengelernt haben, nehmen wir nun eine neue Klasse von Chiffren in den Blick: *Polyalphabetische Substitutionsverfahren* arbeiten (wie monoalphabetische) mit der Substitution von Klartextbuchstaben durch Chiffretextbuchstaben, wobei Sie allerdings nicht nur ein Zielalphabet (Chiffretextalphabet) verwenden, sondern mehrere.

In einer vereinfachten, allgemeinen Beschreibung bildet eine polyalphabetische Chiffre einen Klartextbuchstaben in Abhängigkeit von seiner *Position im Klartext* auf einen Chiffretextbuchstaben (aus einem von mehreren Chiffretextalphabeten) ab. So könnte beispielsweise in einer Klartextfolge `aaaa...` der erste Buchstabe `a` auf ein `c`, der zweite auf ein `x`, der dritte auf ein `s` usw. abgebildet werden. Im Unterschied zu einer monoalphabetischen Substitution, wo einem Klartextzeichen immer in das gleiche Chiffretextzeichen übersetzt wurde, können hier aus dem gleichen Klartextzeichen also verschiedene Chiffretextzeichen hervorgehen.

In der Praxis muss natürlich festgelegt sein, wie die Zeichenposition im Klartext die Substitution beeinflusst. Dies erfolgt in der Regel so, dass eine Reihe der Substitution eine Reihe von n Substitutionstabellen zugrunde liegen und für das erste Zeichen die erste Tabelle, für das zweite die zweite Tabelle usw. verwendet wird, bis man bei der n-ten Tabelle angelangt ist. Für das $n+1$-te Zeichen wird dann für die Substitution wieder die erste Tabelle genommen, die Ersetzungstabellen *wiederholen* sich also (eine Tatsache, die man, wie wir später sehen werden, bei der Kryptoanalyse von polyalphabetischen Chiffren ausnutzen kann).

Bevor wir zur Implementierung übergehen hier noch ein etwas ausführlicheres Beispiel. Wir arbeiten (sowohl im Klar- als auch im Chiffretext) in diesem Beispiel mit dem fünf-elementigen Alphabet `{a,b,c,d,e}` und betrachten die folgenden drei Substitutionstabellen:

Klartext-Zeichen	a	b	c	d	e
Chiffretext-Zeichen (1)	b	a	c	e	d
Chiffretext-Zeichen (2)	e	c	a	d	b
Chiffretext-Zeichen (3)	d	b	e	a	c

In der ersten Substitutionstabelle wird ein `a` also durch ein `b` ersetzt, in der zweiten durch ein `e` und in der dritten durch ein `d`. Verschlüsselt wir nun einen Klartext „`abcdeabcde`“, so wird für die Verschlüsselung des ersten Zeichens die erste Substitutionstabelle angewendet, das `a` also durch ein `b` substituiert. Für das zweite Zeichen `b` wird die zweite Tabelle verwendet, also ein `c` substituiert. Für das dritte Zeichen `c` wird die dritte Tabelle verwendet, also ein `e` substituiert. Für das vierte Zeichen `d` wird wieder die erste Tabelle verwendet, so dass ein `e` substituiert wird. Dieser Prozess wird solange wiederholt, bis alle Zeichen ersetzt sind und sich Chiffretext „`bceebdaaad`“ ergibt (zur Veranschaulichung ist je Zeichen nochmal die verwendete Substitutionstabelle angegeben):

Klartext	a	b	c	d	e	a	b	c	d	e
Chiffretext	b	c	e	e	b	d	a	a	a	d
Verwendete Substitutionstabelle	1	2	3	1	2	3	1	2	3	1

Am erzeugten Chiffretext ist deutlich zu erkennen, dass sowohl gleiche Klartextbuchstaben zu verschiedenen Chiffretextbuchstaben verschlüsselt werden (`a` zu `b` und `d` an Position 1 bzw. 6) als auch verschiedene Klartextbuchstaben zu gleichen Chiffretextbuchstaben (`c` und `d` zu `e` an Position 3 bzw. 4). Ein solches Verhalten ist charakteristisch für eine polyalphabetische Chiffre und wäre im Gegensatz dazu mit einer monoalphabetischen Chiffre nicht möglich.

8.1 Eine allgemeine Implementierung (6 Punkte)

Implementieren Sie nun zunächst das Interface `cipher.substitution.polyalphabetic.PolyalphabeticCipher`. Beachten Sie dabei, dass Sie es weiterhin mit einer Substitutionschiffre zu tun haben und erben Sie dementsprechend von Ihrer abstrakten, `SubstitutionCipher`-implementierenden Klasse. Überlegen Sie sich zudem, dass die Angabe eines Klartextalphabets sowie einer (nicht-leeren) Liste von Chiffretextalphabeten ausreichend, um eine allgemeine polyalphabetische Substitutionschiffre vollständig zu spezifizieren und nutzen Sie dies für den Konstruktor Ihrer neuen Klasse, sodass Sie die entsprechende Methode des `Factory`-Interfaces für einen `PolyalphabeticCipher` ausfüllen können.

Achten Sie in Ihrer Implementierung auf eine geeignete Fehlerüberprüfung und werfen Sie ggf. entsprechende Exceptions.

Bepunktung:

1. korrekte Implementierung der Übersetzungsmethoden (4 Punkte)
2. korrekte Implementierung eines geeigneten Konstruktors (1 Punkt)
3. korrekte Implementierung der `Factory`-Methode (1 Punkt)

8.2 Die Vigenère-Chiffre (6 Punkte)

Nachdem die Entwicklung Kryptoanalyse von monoalphabetischen Substitutionschiffren, insbesondere der Häufigkeitsanalyse (vgl. Abschnitt 7.1), diese Verfahren – zumindest Experten gegenüber – unsicher hat werden lassen, war es vom Standpunkt der Kryptographie aus notwendig geworden, stärkere Verschlüsselungsverfahren zu entwickeln. Obwohl bereits Forscher vor ihm die Vorzüge der Nutzung von mehrerer Alphabete bei der Verschlüsselung (also die polyalphabetische Technik) erkannt hatten, gelang es Blaise de Vigenère als erstem, diesen neuartigen Ansatz in ein praktisch nutzbares Verfahren zu verwandeln. Wir kennen das von ihm in die endgültige Fassung gebrachte Verschlüsselungsverfahren heute daher unter dem Namen *Vigenère-Chiffre*.

Vigenères Idee zur Umsetzung einer polyalphabetischen Substitutionschiffre war konzeptionell simpel, aber genial: In der Vigenère-Chiffre arbeitet man mit 26 Substitutionstabellen[1], welche sich durch die 26 möglichen Verschiebungen des Alphabets ergeben und die man zur Veranschaulichung in einem Quadrat wie in Abbildung 8.1 gezeigt darstellen kann (auch *Vigenère-Quadrat* genannt). Im Grunde genommen umfasst das Vigenère-Quardrat also alle möglichen Caesar-Substitutionstabellen über dem Alphabet mit 26 Buchstaben (vgl. Abschnitt 6.2).

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
B	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
C	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
D	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
E	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
F	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
G	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
H	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
I	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
J	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
K	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
L	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
M	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
N	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
O	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
P	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Q	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
R	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
S	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
T	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
U	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
V	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
W	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
X	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
Y	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Z	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

Abbildung 8.1.: Das Vigenère-Quadrat.[2]

Zur Verschlüsselung wählt man nun ein beliebiges Schlüsselwort, bestehend aus Buchstaben des Alphabets, und schreibt dieses über den Klartext, wobei man das Schlüsselwort so oft wie nötig wiederholt. Nun wird jedes Klartextzeichen mit jener Substitutionszeile aus dem Vigenère-Quadrat verschlüsselt, die mit dem darüber notierten Schlüsselbuchstaben anfängt. Beim Entschlüsseln wiederum geht man genauso vor und übersetzt jedes Zeichen rückwärts entsprechend der zum notierten Schlüsselbuchstaben gehörenden Substitutionszeile.

[1] Allgemeiner sind es n Substitutionstabellen für ein Alphabet der Länge n; wir betrachten hier zur vereinfachten Darstellung allerdings zunächst das übliche Alphabet mit 26 Buchstaben.

[2] Quelle: *Wikimedia Commons*, `https://en.wikipedia.org/wiki/File:Vigen%C3%A8re_square_shading.svg` (14.02.2014).

Für ein ausführliches Beispiel wählen wir uns nun als Schlüsselwort die Zeichenfolge „gdipraktikum“ und wollen die Nachricht „heuteabendtreffenamsee“ verschlüsseln. Dazu schreiben wir zunächst das Schlüsselwort (wiederholend) über den Klartext:

Schlüsselwort	g	d	i	p	r	a	k	t	i	k	u	m	g	d	i	p	r	a	k	t	i	k
Klartext		h	e	u	t	e	a	b	e	n	d	t	r	e	f	f	e	n	a	m	s	e	e

Nun übersetzen wir jeden Klartextbuchstaben, indem wir die Vertauschung entsprechend der Substitutionszeile aus dem Vigenère-Quadrat vornehmen, die mit dem über dem Klartextbuchstaben notierte Schlüsselbuchstaben beginnt. Für den ersten Buchstaben h nutzen wir also die Zeile, die mit dem Schlüsselbuchstaben g beginnt, sodass das h durch n substituiert wird. Für den zweiten Buchstaben e nutzen wir die Substitutionszeile die mit d beginnt, sodass das e durch h substituiert wird. Für den dritten Buchstaben u nutzen wir die Substitutionszeile die mit i beginnt, sodass das u durch c substituiert wird. So fahren wir fort bis alle Zeichen substituiert sind, wodurch sich der Chiffretext „nhcivalxvnndkinteawlmo“ ergibt:

Schlüsselwort	g	d	i	p	r	a	k	t	i	k	u	m	g	d	i	p	r	a	k	t	i	k
Klartext		h	e	u	t	e	a	b	e	n	d	t	r	e	f	f	e	n	a	m	s	e	e
Chiffretext		n	h	c	i	v	a	l	x	v	n	n	d	k	i	n	t	e	a	w	l	m	o

Wie schon bei der allgemeinen polyalphabetischen Verschlüsselung ist am erzeugten Chiffretext deutlich zu erkennen, dass sowohl gleiche Klartextbuchstaben zu verschiedenen Chiffretextbuchstaben verschlüsselt werden als auch verschiedene Klartextbuchstaben zu gleichen Chiffretextbuchstaben. Außerdem sieht man (am Beispiel des Klartextbuchstabens a), dass gleiche Klartextbuchstaben unter dem gleichen Schlüsselbuchstaben natürlich auch zu gleichen Chiffretextbuchstaben verschlüsselt werden.

Implementieren Sie nun das Interface `cipher.substitution.polyalphabetic.Vigenere` in einer Klasse, die von Ihrer `PolyalphabeticCipher`-implementierenden Klasse erbt. Nutzen Sie hierbei die bereits vollständig implementierten Ver- und Entschlüsselungsmethoden dieser Klasse und überlegen Sie sich, wie Sie die Vigenère-Chiffre allein durch eine geeignete Instanziierung der neuen Klasse in ihrem Konstruktor implementieren können. Implementieren Sie zudem die Instanziierungsmethode für die Vigènere-Chiffre in Ihrer `Factory`-Implementierung. Achten Sie auch hier auf eine geeignete Fehlerbehandlung.

Bepunktung:

1. korrekte Implementierung der Vigenère-Chiffre als Erbe der `PolyalphabeticCipher`-Instanz (5 Punkte)
2. korrekte Implementierung der `Factory`-Methode (1 Punkt)

9 Kryptoanalyse beliebiger monoalphabetischer Substitutionschiffren (43 Punkte)

In diesem Kapitel betrachten wir die Kryptoanalyse beliebiger *monoalphabetischer Substitutionsverfahren*. Hierbei wird eine beliebige Permutation des Ausgangsalphabets (vgl. Abschnitt 6.4) als Schlüssel verwendet. Im folgenden ist eine zufällig erstellte Substitutionstablle über unserem Alphabet der Kleinbuchstaben gegeben:

Klartext-Zeichen	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
Chiffretext-Zeichen	q	d	y	x	s	w	t	h	c	a	i	n	g	r	f	u	b	j	p	l	v	z	e	o	m	k

Bei einem Alphabet mit 26 Buchstaben, stehen somit 26! (26 Fakultät)[1], also 403291461126605635584000000 viele verschiedene Schlüssel zur Verfügung. Um die Größe dieser Zahl besser abschätzen zu können, machen wir uns folgenden Zusammenhang bewusst:

$$10^{26} < 26! < 10^{27}$$

Wolframalpha (http://www.wolframalpha.com) gibt folgende Abschätzungen:

- number of cells in human body: $\approx 10^{14}$
- number of cells in human body times number of humans on earth: $\approx 10^{23}$
- number of atoms in a human: $\approx 10^{27}$

Nehmen wir vereinfacht an, dass ein Computer eine Million, also 10^6 Schlüssel in der Sekunde durchprobieren könnte. In diesem Falle würde er immer noch etwa 10^{20} Sekunden, also etwa 10^{13} Jahre benötigen. Es wird geschätzt, dass das Universum etwa 13 Milliarden (also etwa 10^{10}) Jahre alt ist.

Glücklicherweise müssen wir zum Brechen einer monoalphabetischen Chiffre nicht ganz so lange warten. In Abschnitt 7 haben Sie bereits gesehen, wie eine einfache Häufigkeitsanalyse dazu genutzt werden kann Caesar Chiffren automatisiert zu lösen. Mit dem gleichen Trick lassen sich auch einige wenige Zeichen eines Chiffretextes übersetzen, der mit einem zufälligen Schlüssel verschlüsselt wurde. Zählt man die Vorkommnisse der einzelnen Zeichen im Chiffretext, so ist das am häufigsten vorkommende Zeichen – im Deutschen – mit relativ hoher Wahrscheinlichkeit die Verschlüsselung des Buchstaben `e` (siehe auch Tabelle 7.1). Nach Festlegung des ersten Buchstaben, hilft die einfache Häufigkeitsanalyse jedoch nur noch bedingt bei der Entschlüsselung des restlichen Textes. Mit ein bisschen Zufall und dem richtigen Quäntchen Verstand kommt man jedoch weiter.

Angreifermodelle

Wir werden in den folgenden Aufgaben verschiedene Ansätze implementieren, um auch beliebige monoalphabetische Substitutionschiffren zu lösen. Hierbei betrachten wir verschieden starke Angreifermodelle. Ein Angreifermodell beschreibt die Möglichkeiten, die einem Angreifer gegen das Verfahren gegeben werden. In Abschnitt 7.1 haben Sie bereits *Known-Ciphertext Angriffe* kennengelernt, bei denen ein Angreifer nur einen Chiffretext erhält. Dies ist für den Angreifer die wohl schwierigste Situation. Zudem haben Sie *Known-Plaintext Angriffe* kennengelernt, in denen der Angreifer zusätzlich noch den Plaintext zu (zumindest) einem Ciphertext kennt, und gesehen, dass die Caesar-Chiffre damit sofort und fehlerfrei zu brechen ist.

In diesem Abschnitt betrachten wir nun folgende Angreifermodelle, die zusätzlich noch stärkere Angreifer beinhalten:

Chosen-Plaintext Angriff Das *Chosen-Plaintext* Angreifermodell ist ein sehr starkes Angreifermodell, dass es dem Angreifer erlaubt, Verschlüsselungen zu beliebigen (selbst gewählten) Plaintexten zu erhalten. Das Interface `analysis.ChosenPlaintextAnalysis` stellt hierfür eine Methode bereit, der ein Objekt vom Typ `EncryptionOracle` übergeben wird. Wenn Sie später das Interface implementieren und einen Chosen-Plaintext Angriff durchführen, erlaubt Ihnen dieses Objekt Verschlüsselungen zu erzeugen.

[1] Für den ersten Buchstaben gibt es 26 Möglichkeiten, für den zweiten 25, für den dritten noch 23 Möglichkeiten usw: $26! := 26 \cdot 25 \cdot 24 \cdot \ldots \cdot 2 \cdot 1$

Die Möglichkeiten des Angreifers im Chosen-Plaintext Modell mögen vielleicht etwas übertrieben klingen. Sollte ein Verfahren jedoch beweisbar sicher in diesem Modell sein, bedeutet dies auch, dass das Verfahren sicher ist gegen alle Angreifer mit weniger Möglichkeiten. Das Chosen-Plaintext Modell ist daher ein gängiges Modell, um die Sicherheit moderner Verschlüsselungsverfahren zu beurteilen. Antike Verfahren halten solch starken Angreifern, wie wir sehen werden, jedoch nicht stand.

Known-Plaintext Angriff Bei Known-Plaintext Angriffen erhält der Angreifer Chiffretexte sowie zugehörige Klartexte. Im Gegensatz zum Chosen-Plaintext Angriff kann er diese jedoch nicht selbst wählen.

Known-Ciphertext Angriff Der Known-Ciphertext Angriff ist der für Codeknacker wohl schwierigste Angriff. Hierbei kennt der Kryptoanalytiker lediglich den Chiffretext. In unserem Falle gehen wir davon aus, dass zumindest die Sprache der verschlüsselten Nachricht bekannt ist.

Known-Ciphertext Angriff mit Cribs Als Crib[2] bezeichnet man Wörter oder Phrasen von denen der Kryptoanalyst vermutet (bzw. weiß), dass diese in dem verschlüsselten Text vorkommen.

Im folgenden werden wir Angreifer in den unterschiedlichen Modellen implementieren. Wir beginnen dabei mit dem stärksten Angreifer, dem Angreifer im Chosen- und Known-Plaintext Modell.

9.1 Chosen-Plaintext und Known-Plaintext Angriff (5 Punkte)

Als Einstieg sollen Sie Chosen- und Known-Plaintext Angriffe implementieren. Betrachten Sie dazu das Interface `analysis.monoalphabetic.MonoalphabeticCpaNpaCryptanalysis`. Dieses erbt sowohl von dem Interface `analysis.ChosenPlaintextAnalysis` als auch von `analysis.KnownPlaintextAnalysis`. Implementieren Sie nun das Interface `MonoalphabeticCpaNpaCryptanalysis`. Für die Known-Plaintext Angriffe ist es ausreichend die Methode

```
public char[] knownPlaintextAttack(String ciphertext, String plaintext, Alphabet alphabet)
```

zu implementieren. Die anderen beiden Methoden sollten einfach auf diese verweisen, diese also etwa mittels

```
return knownPlaintextAttack(ciphertext, plaintext, distribution.getAlphabet());
```

aufrufen. Ihre Aufgabe ist es, den verwendeten Schlüssel herauszurechnen und diesen als `char[]` array zurückzugeben. Beginnen Sie die Implementierung am besten mit dem Chosen-Plaintext Angriff. Hierbei ist es ausreichend eine einzige Anfrage an das bereitgestellte Orakel `EncryptionOracle` zu stellen. Für Known-Plaintext Angriffe hängt es von dem bereitgestellten Plaintext/Ciphertext Paar ab, in wie weit Sie den Schlüssel rekonstruieren können. Ist z.B. der Text kürzer als das Alphabet, so lässt sich der Schlüssel natürlich nicht vollständig wiederherstellen. In einem solchen Falle setzen Sie die noch unbekannten Buchstaben des Schlüssels auf `' '`, also ein Leerzeichen. Konnten Sie also etwa folgenden Schlüssel bestimmen

Klartext-Zeichen	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
Chiffretext-Zeichen		d		x			t	h	c	a	i	n	g	r	f	u		j		l	v		e	o	m	k

so sollten sie folgendes Array zurückgeben:

```
new char[]{ ' ', 'd', ' ', 'x', ' ', ' ', 't', 'h', 'c', 'a', 'i', 'n', 'g',
            'r', 'f', 'u', ' ', 'j', ' ', 'l', 'v', ' ', 'e', 'o', 'm', 'k' };
```

Wie immer müssen Sie für die Testfälle die entsprechende `Factory`-Methode implementieren und sollten zusätzlich zu den vorgegebenen Testfällen eigene Testfälle ergänzen.

Bepunktung:

1. korrekte Implementierung des Chosen-Plaintext Angriffes (2 Punkte)
2. korrekte Implementierung des Known-Plaintext Angriffes (3 Punkte)

[2] Der Begriff „Crib“ entstand in Bletchley Park, dem Hauptsitz der britischen Kryptoanalysten im zweiten Weltkrieg.

9.2 n-Gramme und Distribution, Teil 2: Vorbereitung auf den Known-Ciphertext Angriff (5 Punkte)

Wie Sie feststellen konnten, halten monoalphabetische Chiffren Chosen- sowie Known-Plaintext Angriffen nicht stand. Im folgenden möchten wir einen schwächeren Angreifer betrachten. Hierbei erhalten Sie lediglich einen Ciphertext sowie eine Liste mit Cribs, also Worten, von denen Sie bereits wissen, dass diese im zugehörigen Klartext vorkommen. An welcher Stelle ein solches Wort auftritt ist hierbei jedoch unbekannt. Um diesen Angriff implementieren zu können müssen wir jedoch noch ein wenig Vorarbeit leisten.

In Abschnitt 7.2 haben Sie bereits eine erste Version des `Distribution` Interfaces implementiert, um eine einfache Häufigkeitsverteilung zu berechnen. Wir werden nun die restlichen Methoden implementieren. Darüber hinaus werden wir die Häufigkeitsanalyse verallgemeinern von einzelnen Buchstaben zu n-Grammen. Ein *n-Gramm* ist das Ergebnis einer Textzerlegung. Wir betrachten n-Gramme auf Buchstabenebene, das heißt, wir zerlegen den Text zunächst in einzelne Buchstaben und fassen dann jeweils n benachbarte Buchstaben zusammen. Das Ergebnis heißt n-Gramm. Insbesondere sind wir an Bi- sowie Trigrammen (also 2- und 3-Grammen) interessiert. Wir werden das Konzept allerdings allgemeiner implementieren.

Betrachten wir für das Verständnis den folgendes Zitat

> *Machines take me by surprise with great frequency.* (Alan Turing)

Dieses enthält die folgenden Bi-Gramme

> [Ma] [ac] [ch] [hi] [in] [ne] [es] [s] [t] [ta] [ak] [ke] [e] [m] [me] [e] [b] [by] [y] [s] [su] [ur] [rp] [pr] [ri] [is] [se] [e] [w] [wi] [it] [th] [h] [g] [gr] [re] [ea] [at] [t] [f] [fr] [re] [eq] [qu] [ue] [en] [nc] [cy] [y.]

Das `Utils` Interface stellt die Methode `ngramize` bereit. Diese erwartet als Parameter den zu zerlegenden Text, sowie die größe der zu erzeugenden n-Gramme (die n-Gramm-Längen werden als varargs bzw. als int-Array an die Methode übergeben). Als Rückgabewert soll die Methode eine `Map<Integer, List<String>>` liefern, wobei ein Key (eine n-Gramm-Länge) in der Map auf die Liste der gefundenen n-Gramme für diese Länge zeigt. Um im obigen Beispiel zu bleiben: Der Aufruf

```
ngramize("Machines take me by surprise with great frequency.", 2, 3 )
```

sollte eine Map mit zwei Einträgen (für 2 und 3) erzeugen:

```
{2=["Ma", "ac", "ch", "hi", "in", "ne", "es", "s ", " t", "ta", "ak", "ke", "e ",
    " m", "me", "e ", " b", "by", "y ", " s", "su", "ur", "rp", "pr", "ri", "is",
    "se", "e ", " w", "wi", "it", "th", "h ", " g", "gr", "re", "ea", "at", "t ",
    " f", "fr", "re", "eq", "qu", "ue", "en", "nc", "cy", "y."],
 3=["Mac", "ach", "chi", "hin", "ine", "nes", "es ", "s t", " ta", "tak", "ake",
    "ke ", "e m", " me", "me ", "e b", " by", "by ", "y s", " su", "sur", "urp",
    "rpr", "pri", "ris", "ise", "se ", "e w", " wi", "wit", "ith", "th ", "h g",
    " gr", "gre", "rea", "eat", "at ", "t f", " fr", "fre", "req", "equ", "que",
    "uen", "enc", "ncy", "cy."]}
```

Implementieren Sie nun die Methode `ngramize` in ihrer `Utils`-Klasse.

Im nächsten Schritt verfeinern wir die `Distribution`-Klasse, so dass diese nicht nur auf Buchstaben (also 1-Grammen) arbeitet, sondern Häufigkeitsverteilungen für beliebige n-Gramm-Längen berechnet. Ihr Konstruktor sollte bereits drei Parameter erwarten: `Alphabet source`, `String text` und **`int`** `ngramsize`. Passen Sie Ihre Klasse nun so an, dass diese für alle n-Gramme mit $n = 1,\ldots,$`ngramsize` die Häufigkeitsverteilung bestimmt. Zusätzlich müssen Sie folgende Anpassungen durchführen

- Die Methode **`double`** `getFrequency(String)` sollte nun passend der Länge des übergebenen Strings die Häufigkeit des entsprechenden n-Gramms zurückgeben.
- Die Methode `List<String> getSorted(`**`int`** `length)` soll die n-Gramme der Länge `length` sortiert nach Häufigkeit zurückgeben
- Implementieren Sie zusätzlich die Methode `String getByRank(`**`int`** `ngramsize, `**`int`** `rank)`. Diese soll das `rank` -häufigste n-Gramm der Länge `length` zurückgeben. Hierbei soll `rank` 1-basiert sein, das heißt das häufigste n-Gramm wird über Rang 1 angesprochen. Um also bspw. das zweithäufigste Trigram abzufragen würde die Methode mit Parametern `(3,2)` aufgerufen werden.

Achten Sie außerdem darauf, dass Sie nur auf im Alphabet vorkommenden Zeichen arbeiten. Hierfür sollte die Methode `alphabet.normalize` hilfreich sein. Achten Sie bei Ihrer Implementierung wie immer auf eine ordentliche Fehlerbehandlung.

Bepunktung:

1. korrekte Implementierung von `Utils.ngramize` (2 Punkte)
2. korrekte Anpassung der `Distribution`-Klasse (3 Punkte)

9.3 Wörterbücher: Vorbereitung auf den Known-Ciphertext Angriff (3 Punkte)

Neben der Betrachtung verfeinerter Häufigkeiten werden wir in den folgenden Analysen auf ein zweites Hilfsmittel zurückgreifen: ein Wörterbuch. Implementieren Sie hierfür das Interface `alphabet.Dictionary`. Als Konstruktor bekommt das Wörterbuch ein Alphabet und einen Eingangstext übergeben, den es in Wörter aufbrechen und analysieren soll. Wörter sollen hierbei an folgenden Zeichen getrennt werden: `' '`, `','`, `'!'`, `'?'` und `'.'` (also an Leerzeichen, Kommata, Ausrufezeichen, Fragezeichen und Punkten). Zusätzlich sollen nur Wörter in das Wörterbuch aufgenommen werden, die vom Alphabet erlaubt werden (siehe `Alphabet.allows`).

Tipp: Die Analyse des Textes kann z.B. relativ einfach mittels `java.util.Scanner` erfolgen. Achten Sie hierbei auf ein korrektes Schließen (`Scanner.close()`) des Scanners nach Benutzung.

Implementieren Sie zusätzlich die entsprechende `Factory`-Methode für Ihre Implementierung von `Dictionary`.

Bepunktung:

1. korrekte Implementierung des Konstruktors und der Methode `getAlphabet` (1 Punkt)
2. korrekte Implementierung des Wörterbuchs (2 Punkte)

9.4 Backtracking: Vollständige Suche bei kleinem Schlüsselraum (8 Punkte)

Wir möchten nun einen Known-Ciphertext Angriff unterstützt mit Cribs (also bekannten Wörtern im Plaintext) implementieren, wobei wir mithilfe von Backtracking vorgehen. Backtracking haben Sie bereits in der Vorlesung (n-Damen-Problem) sowie den Übungen (Reguläre Ausdrücke/Pattern Matching) kennengelernt. Diese Technik erlaubt es einen Lösungsraum Schritt für Schritt abzutasten. In unserem Falle ist der Lösungsraum bei einem zugrundeliegenden Alphabet mit 26 Buchstaben jedoch extrem groß (es existieren 26 Fakultät mögliche Lösungen). Wir werden daher als ersten Zwischenschritt kleinere Alphabete mit maximal 10 Zeichen betrachten. Diese Größenordnung (noch etwa 3,5 Millionen Lösungen) ist auch ohne die Zuhilfenahme von Cribs noch handhabbar.

Erstellen Sie nun eine Klasse, die die Interfaces `MonoalphabeticCribCryptanalysis` sowie `BacktrackingAnalysis` implementiert. Das erste Interface stellt die zu implementierenden Methoden des Known-Ciphertext Angriffes bereit. Das zweite Interface soll bei der Implementierung des Backtracking unterstützen. Denken Sie daran die entsprechende Methode in der `Factory` zu implementieren. Zunächst betrachten wir die Methode

```
char[] knownCiphertextAttack(String ciphertext, Distribution distribution,
    Dictionary dictionary, List<String> cribs, ValidateDecryptionOracle testOracle)
```

Die einzelnen Parameter sind hierbei

ciphertext Der gegebene Schlüsseltext.

distribution Die Häufigkeitsanalyse auf dem Quell-Alphabet.

dictionary Ein Wörterbuch.

cribs Eine Liste mit Cribs, also Wörtern die im zum Schlüsseltext gehörenden Klartext vorkommen. Zunächst werden wir diese einfach ignorieren.

testOracle Ein Hilfsobjekt vom Typ `analysis.ValidateDecryptionOracle`. Dieses erlaubt es zu überprüfen ob richtig entschlüsselt wurde. Wir werden dieses später durch eine Heuristik ersetzen.

In dieser Methode werden wir nun den vereinfachten Angriff implementieren. Zusätzlich sollten Sie die `getState`-Methode implementieren. Diese wird von Testfällen periodisch aufgerufen, um einen Zwischenstand der Berechnung auszugeben. Als Zwischenstand können Sie z.B. den zur Zeit vermuteten Schlüssel, sowie Parameter wie die Anzahl der durchgeführten Iterationen etc. ausgeben. Die `getState`-Methode erhält als Parameter

sourceAlphabet Das Ausgangsalphabet, also das Alphabet des Plaintextes.

targetKey Der zu suchende Schlüssel. Dies ermöglicht z.B. eine Darstellung der Distanz des zur Zeit vermuteten Schlüssels zum echten Schlüssel.[3]

Anmerkung: Die `getState`-Methode ist auch prima geeignet für das Debuggen Ihrer Lösung. Beachten Sie außerdem, dass Ihr Objekt Zustand in Klassenvariablen speichern kann. Ein Objekt wird immer nur für einen Angriff verwendet und insbesondere nicht von mehrere Threads gleichzeitig für eine Analyse verwendet.

Um das eigentliche Backtracking zu implementieren werden wir die unterschiedlichen Methoden des Interfaces `BacktrackingAnalysis` implementieren. Die Methode `isPromisingPath` können Sie für diesen Abschnitt noch unimplementiert lassen bzw. immer **`true`** oder **`false`** zurückgeben. Die Methode `reconstructKey` stellt hier den Beginn der Rekursion dar und wird aus der Methode `knownCiphertextAttack` aus aufgerufen. Als Modellierung einer (Teil-)Lösung verwenden wir eine `Map<Character, Character>`. In dieser Map werden nach und nach Belegungen hinzugefügt wobei der Key der Map ein Zeichen aus dem Quellalphabet entspricht und der Value ein Zeichen aus dem Zielalphabet (dem Alphabet des Ciphertexts). Zu Beginn des Backtrackings ist die Map (die aktuelle Teillösung) leer und in jedem Schritt des Backtrackings wird eine weitere Belegung hinzugefügt.

Gegeben eine aktuelle (Teil-)Lösung in Form von `Map<Character, Character>` müssen Sie also zunächst die noch möglichen Belegungen bestimmen. Betrachten wir dazu Beispielhaft folgendes Alphabet, Schlüssel, sowie eine mögliche Teillösung.

Klartext-Zeichen	b	d	f	h	i	k	m	p	s	u
Schlüssel	f	h	k	p	s	m	i	b	d	u
Aktuelle-Teillösung			k	p			i		d	u

Um Ihr Backtracking zu implementieren müssen Sie im Grunde in Methode `reconstructKey` lediglich drei Operationen durchführen, bevor Sie den Rekursionsschritt (der erneute Aufruf von `reconstructKey`) durchführen.

1. Falls die Teillösung einer kompletten Lösung entspricht, müssen Sie den Ciphertext mit dem gefundenen Schlüssel entschlüsseln und das Ergebnis mit Hilfe des `ValidateDecryptionOracle` überprüfen. Ist der gefundene Plaintext der richtige, so haben wir den richtigen Schlüssel gefunden und können das Backtracking beenden. Der gefundene Schlüssel soll als **`char`**`[]` Array zurückgegeben werden (Siehe Abschnitt 9.1). Ist die Teillösung nicht richtig muss ein Schritt „backgetrackt“ werden.

2. Es muss festgelegt werden für welches noch nicht festgelegte Zeichen aus dem Quellalphabet (`sourceAlphabet`) als nächstes ein Schlüsselzeichen festgelegt werden soll. Dies wird mit Hilfe der Methode `getNextSourceChar` implementiert werden.

3. Es müssen die möglichen Belegungen für das in Schritt 2 festgelegte Zeichen bestimmt werden. Hierfür werden wir die Methode `getPotentialAssignments` implementieren. Anschließend wird der Rekursionsschritt durchgeführt. Das heißt es wird das erste Zeichen c_T der potential Assignments gewählt, es wird eine um $c_S \mapsto c_T$ (wobei c_S das in Schritt 2 festgelegte Zeichen des Quellalphabets beschreibt) erweiterte Teillösung generiert und anschließend wird erneut die Methode `reconstructKey` aufgerufen (der Rekursionsschritt).

Es fehlt die Implementierung der Methoden `getNextSourceChar` sowie `getPotentialAssignments`. Die Methode `getNextSourceChar` kann einfach gehalten werden. Das heißt, es ist für die aktuelle Analyse völlig ausreichend einfach ein noch nicht in der Teillösung vorkommendes Zeichen aus dem Quellalphabet zu wählen. Ebenso kann die Methode `getPotentialAssignments` einfach implementiert werden, indem eine Liste der noch nicht verwendeten Zeichen aus dem Zielalphabet zurückgegeben wird. (Natürlich können Sie auch mit Hilfe von `Distribution` und `Dictionary` versuchen, bessere Reihenfolgen zu generieren.)

Zusätzlich sollten Sie die oben beschrieben Methode `getState` implementieren. Diese kann z.B. auch für Ihr Debugging benutzt werden, indem Sie diese z.B. alle 1000 Iterationsschritte aufrufen. In unserer Beispielimplementierung gibt die `getState` Methode folgendes zurück:

[3] Die Testmethoden lassen Ihre Analyse in einem eigenen Thread laufen und rufen periodisch die getState Methode auf. In den Tutorentestfällen wird hierbei nicht der gesuchte Schlüssel mit eingegeben. Also: nicht schummeln...

```
[b, d, f, h, i, k, m, p, s, u]
[k, p, f, m, u, i, h, b, d, s]
[i, d, h, m, u, f, p, b, s, k, ]
[0, 0, 0, 1, 1, 0, 0, 1, 0, 0, ]
iterations: 2592864 (3628800)
path: ishdumfbkp
correct: 1
```

Hierbei ist die erste Zeile das Quellalphabet und die zweite der gesuchte Schlüssel (Zielalphabet). In der dritten Zeile geben wir die aktuell betrachtete Teillösung an. Die vierte Zeile zeigt welche Zeichen aktuell richtig zugeordnet sind. Trotz einer richtigen Zuteilung von 3 Zeichen geben wir in der letzten Zeile nur 1 als Korrekt an. Dies kommt daher, dass zwei Zeichen zufällig richtig gewählt worden sind, aber nicht in der Backtracking-Reihenfolge. Diese geben wir als Pfad an, das heißt unser Algorithmus hat zunächst „i“, dann „s“, dann „h“ usw. ausgewählt. Für „i“ wurde das richtige Zeichen ausgewählt, für „s“ jedoch noch nicht. Aufgrund des Backtracking-Ansatzes bedeutet dies daher, dass der Algorithmus bis „s“ zurückgehen muss, um die richtige Lösung zu finden.

Bepunktung:

1. korrekte Implementierung von `getNextSourceChar` (1 Punkt)
2. korrekte Implementierung von `getPotentialAssignments` (1 Punkt)
3. korrektes Backtracking (5 Punkte)
4. sinnvolle Ausgabe von Zwischenständen in `getState` (1 Punkt)

9.5 Backtracking: Suche auf großem Schlüsselraum mit Cribs (10 Punkte)

Die in dem vorherigen Abschnitte erstellte Lösung ist für große Alphabete leider nicht geeignet. Wie lange dauert Ihre Analyse auf Alphabeten der Größe 10, 11, 12, 13? Auf dem vollständigen Alphabet mit 26 Buchstaben werden wir das Ergebnis der Analyse auf jeden Fall nicht mehr erfahren, egal, wie performant Sie auch programmieren.

Als zusätzliches Hilfsmittel werden wir nun Cribs verwenden, mit denen wir das Backtracking steuern können. Ziel ist es, Pfade im Backtracking, die nicht zu einer Lösung führen können, möglichst früh zu erkennen und abzuschneiden. Wir betrachten folgenden Klartext

> *„willst du mir wohl sagen, wenn ich bitten darf, welchen weg ich hier nehmen muss?“*
> *„das haengt zum guten theil davon ab, wohin du gehen willst“, sagte die katze.*
> *„es kommt mir nicht darauf an, wohin“, sagte alice.*
> *„dann kommt es auch nicht darauf an, welchen weg du nimmst,“ sagte die katze.*
> *„wenn ich nur irgendwo hinkomme,“ fuegte alice als erklaerung hinzu.*
> *„o, das wirst du ganz gewiss,“ sagte die katze, „wenn du nur lange genug gehest.“*
>
> (Lewis Carroll: „Alice im Wunderland“)

sowie eine Verschlüsselung mit folgendem zufällig erzeugtem Schlüssel

Klartext-Zeichen	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
Chiffretext-Zeichen	v	p	i	r	f	c	d	b	z	l	n	y	t	g	q	k	a	e	w	u	j	x	h	m	s	o

Verschlüsselter Text:

```
hzyywurjtz ehqbywvdfg hfggzibpzu ufgrvechfy ibfghfdzib bzfegfbtfg

tjwwrvwbvf gduojtdjuf gubfzyrvxq gvphqbzgrj dfbfghzyyw uwvdufrzfn

vuoffwnqtt utzegzibur vevjcvghqb zgwvdufvyz ifrvggnqtt ufwvjibgzi

burvevjcvg hfyibfghfd rjgzttwuwv dufrzfnvuo fhfggzibgj ezedfgrhqb

zgnqttfcjf dufvyzifvy wfenyvfejg dbzgojqrvw hzewurjdvg odfhzwwwvd

ufrzfnvuof hfggrjgjey vgdfdfgjdd fbfwu
```

Anmerkung: Wie Sie bereits festgestellt haben, arbeiten wir ohne Leer- und Satzzeichen in unserem Alphabet. Die obige Darstellung Gruppiert den Text einfach in Gruppen à 10 Buchstaben. Hierbei ist die Gruppierung willkürlich gewählt und dient lediglich dazu den Text optisch etwas aufzubereiten. Die hierzu verwendete Methode ist ebenfalls im `Utils`-Interface enthalten und heißt `toDisplay`.

Angenommen wir wissen, dass im Klartext die Worte *alice* und *irgendwo* vorkommen. Wir möchten nun unseren Backtracking-Prozess anhand dieser Cribs steuern. Hierzu ändern wir zunächst die Methode `getNextSourceChar` so ab, dass diese zunächst Buchstaben von bekannten Cribs auswählt. Also, z.B. dass diese in der Reihenfolge `alicergndwo` zurückgegeben werden. Sind alle Buchstaben der Cribs vergeben, können wieder zufällige Buchstaben des Quellalphabets ausgewählt werden – natürlich können Sie auch die zur Verfügung gestellten Objekte `Distribution` nutzen, um z.B. die zu wählenden Buchstaben in einer bestimmten Reihenfolge auszuwählen.

Nun kommt der entscheidende Schritt: In jeder Rekursionsstufe rufen wir zunächst die Methode `isPromisingPath` auf. Gibt diese **`false`** zurück, so brechen wir diesen Pfad ab, und geben direkt **`null`** zurück. Es bleibt zu klären, welche Pfade „promising“ (also: vielversprechend) sind. Hierzu stellen wir zwei Ideen vor; gerne können Sie weitere eigene Ideen mit einbinden:

Exact Crib Match

Sind in einer Teillösung alle Buchstaben eines Cribs bereits gesetzt, so muss, wenn wir den Schlüsseltext mit dem Teilschlüssel entschlüsseln, das Crib in dem Plaintext vorkommen. Im folgenden haben wir einen Schlüssel gewählt der mit dem gesuchten Schlüssel auf den Buchstaben von *alice* übereinstimmt. In dem entschlüsselten Text, haben wir alle noch nicht festgelegten Zeichen durch X ersetzt. Wie Sie sehen, kommt das Crib an zwei Stellen vor:

```
XillXXXXXi XXXXlXaXeX XeXXicXXiX XeXXaXXXel cXeXXeXicX XieXXeXXeX

XXXXXaXXae XXXXXXXXXe XXXeilXaXX XaXXXXiXXX XeXeXXillX XXaXXeXieX

aXXeeXXXXX XXiXXicXXX aXaXXaXXXX iXXaXXeali ceXaXXXXXX XeXaXcXXic

XXXaXaXXaX XelcXeXXeX XXXiXXXXXa XXeXieXaXX eXeXXicXXX XiXXeXXXXX

iXXXXXeXXe XXealiceal XeXXlaeXXX XXiXXXXXaX XiXXXXXXaX XXeXiXXXaX

XeXieXaXXe XeXXXXXXXl aXXeXeXXXX eXeXX
```

Kommt das Crib in der Lösung nicht vor, so kann die aktuelle Teillösung nicht korrekt sein, und wir brauchen den aktuellen Pfad nicht weiter zu erforschen.

Partial Crib Match

Neben der Überprüfung, ob ein bereits in der Teillösung gesetztes Crib in dem zugehörigen (partiellen) Plaintext vorkommt, können Sie auch überprüfen, ob ein Crib noch die Möglichkeit hat aufzutreten. Im folgenden haben wir für die Teillösung

$$\{a \mapsto v, l \mapsto y, i \mapsto z, c \mapsto i, e \mapsto f, r \mapsto e, g \mapsto d\}$$

wieder den halbentschlüsselten Plaintext angegeben. Wir wissen, dass das Wort „irgendwo“ in dem Plaintext vorkommen muss. Wir können also nun ein „Fenster“ von 8 Buchstaben über unseren halb entschlüsselten Text schieben und überprüfen, ob das Wort an einer Stelle noch passen könnte. In diesem Fall hat es geklappt:

```
XillXXXXXi rXXXlXageX XeXXicXXiX XeXXarXXel cXeXXegicX XierXeXXeX

XXXXXaXXae XgXXXXgXXe XXXeilXaXX XaXXXXiXXX geXeXXillX XXagXeXieX

aXXeeXXXXX XXirXicXXX araXXaXXXX iXXagXeali ceXaXXXXXX XeXaXcXXic

XXXaraXXaX XelcXeXXeg XXXiXXXXXa gXeXieXaXX eXeXXicXXX rirgeXXXXX

iXXXXXeXXe gXealiceal XerXlaerXX gXiXXXXXaX XirXXXXgaX XgeXiXXXag

XeXieXaXXe XeXXXXXXrl aXgegeXXgg eXeXX
```

Passt das Wort nicht mehr, so kann die aktuelle Teillösung ebenfalls nicht korrekt sein, und wir brauchen den aktuellen Pfad nicht weiter zu erforschen.

Hinweis: Mit der beschriebenen Methode und ausreichend vielen (und guten) Cribs lassen sich Texte, die mit einer monoalphabetischen Chiffre verschlüsselt worden sind recht gut entschlüsseln. Die Laufzeit kann hierbei allerdings immer noch einige Minuten betragen. Ebenso wird es problematisch, falls zu wenige oder zu kurze Cribs bekannt sind. Eine Möglichkeit hiermit umzugehen ist es, das Wörterbuch zur Hilfe heranzuziehen und Wörter zu suchen, die in die gegebene Teillösung noch passen und das Backtracking zusätzlich hierüber zu steuern. Für dieses Praktikum müssen Sie eine solche Analyse jedoch nicht implementieren. In allen Testfällen stellen wir ausreichend Cribs zur Verfügung und im nächsten Abschnitt sehen wir, wie man mit einem einfachen Trick sogar gänzlich ohne Cribs auskommen kann.

Bepunktung:

1. korrekter Einbau der Methode `isPromisingPath` in die Methode `reconstructKey` (1 Punkt)
2. Implementierung von „Exact Crib Match“ in `isPromisingPath` (2 Punkte)
3. Implementierung von „Partial Crib Match“ in `isPromisingPath` (2 Punkte)
4. Bestehen der unveröffentlichten Testfälle (2 Punkte)

9.5.1 Ein eigenes `ValidateDecryptionOracle` (3 Punkte)

Bis hierhin haben wir zur Überprüfung unserer Lösung das `ValidateDecryptionOracle` verwendet. In der Praxis ist ein solches Orakel natürlich nicht vorhanden. In dieser Aufgabe sollen Sie mit Hilfe von `Distribution` und `Dictionary` eine eigene Implementierung von `ValidateDecryptionOracle` erstellen. Überlegen Sie sich hierzu eine geeignete Methode, um festzustellen, ob ein gegebener Text einen gültigen Text im Alphabet darstellt. Also, warum sieht

```
ponnitxjyo qpfgniedcr pcrroagmot tcrxeqbpcn agcrpcdoag gocqrcgycr

yjiixeigec rdtzjydjtc rtgconxehf rempfgorxj dcgcrponni tiedtcxocv

etzccivfyy tyoqroagtx eqejberpfg oriedtceno acxerrvfyy tciejagroa

gtxeqejber pcnagcrpcd xjroyyitie dtcxocvetz cpcrroagrj qoqdcrxpfg

orvfyycbjc dtcenoacen icqvnecqjr dgorzjfxei poqitxjder zdcpoiiied

tcxocvetzc pcrrxjrjqn erdcdcrjdd cgcit
```

nicht richtig aus wohingegen der folgende Text gültig zu sein scheint:

```
willstdumi rwohlsagen wennichbit tendarfwel chenwegich hiernehmen

mussdashae ngtzumgute ntheildavo nabwohindu gehenwills tsagtediek

atzeeskomm tmirnichtd araufanwoh insagteali cedannkomm tesauchnic

htdaraufan welchenweg dunimmstsa gtediekatz ewennichnu rirgendwoh

inkommefue gtealiceal serklaerun ghinzuodas wirstdugan zgewisssag

tediekatze wenndunurl angegenugg ehest
```

Anmerkung: Mit einer Heuristik wird es eventuell nicht mehr möglich sein den genau richtigen Schlüssel zu bestimmen. Allerdings sollen Sie in der Lage sein einen Schlüssel mit nur wenig Fehlern zu finden.

Implementieren Sie zusätzlich die zugehörige `Factory`-Methode.

Bepunktung:

1. Implementierung einer „sinnvollen“ Heuristik (2 Punkte)
2. Bestehen der unveröffentlichten Testfälle (1 Punkt)

9.6 Zufall, hilf! – Eine allgemeine Lösung mit genetischem Algorithmus (12 Punkte)

Im vergangenen Abschnitt haben wir bereits eine weitgehend automatisierte Lösung erarbeitet, die allerdings auf das Vorhandensein von Cribs angewiesen ist. Im folgenden gehen wir einen neuen Weg und zeigen, wie man mit Hilfe von Zufall monoalphabetische Chiffren komplett lösen kann – auch ohne Cribs! Hierzu werden wir auf einfache genetische bzw. evolutionäre Algorithmen zurückgreifen.

Das Genom: Die Beschreibung der Lösung. Zunächst muss festgelegt werden, wie eine gültige Lösung modelliert wird. In unserem Falle ist eine gültige Lösung ein `Alphabet`.

Fitness: Wie gut ist eine Lösung? Im Laufe des Algorithmus werden immer wieder neue Lösungen (Individuen) aus bestehenden Individuen erzeugt. Um hierbei zielgerichtet voran zu gehen werden normalerweise schlechte Lösungen verworfen und gute Lösungen weiter betrachtet. Hierzu ist es notwendig, einer Lösung einen Punktwert zuzuteilen, der beschreibt, wie gut oder schlecht diese Lösung im Vergleich zu anderen Lösungen ist. Die Spezifizierung dieser Fitnessfunktion ist mit der wichtigste Aspekt bei genetischen Algorithmen.

Die Population: Eine Generation. Als Population oder Generation bezeichnet man die Menge der zu einem Zeitpunkt lebendigen Individuen.

Selektion: Survival of the fittest. Als Selektion bezeichnet man die Auswahl der fittesten Individuen aus der Population. Diese werden verwendet, um die nächste Generation zu erzeugen.

Fortpflanzung: Erzeugen der nächsten Generation. Gegeben eine Menge an Individuen wird über einen zu spezifizierenden Prozess eine neue Population erstellt. Hierbei können unterschiedliche Strategien zur Anwendung kommen. Die einfachste ist die Mutation, bei der aus einem einzigen Ausgangsindividuum durch kleine Veränderungen des Genoms (der Lösung) neue Individuen erzeugt werden. Natürlich sind komplexere Methoden denkbar. So können z.B. auch die Genome zweier Individuen kombiniert werden (sogenanntes „Crossover“), um ein neues Individuum zu erzeugen.

Der Ablauf eines genetischen Algorithmus folgt im Grunde immer folgendem Schema:

1. Es wird eine initiale Population erzeugt. Dies kann zufällig geschehen, oder aber mittels Heuristiken die sich auf Domänenwissen beziehen. Ein Parameter der hierbei festzulegen ist, ist die Größe der Population.
2. Nun werden die folgenden Schritte wiederholt, bis sich eine stabile Lösung ergibt:
   a) Durch Selektion werden die n fittesten Individuen ausgewählt. Hierbei gilt es neben der Fitnessfunktion die Anzahl der auszuwählenden Individuen zu bestimmen.
   b) Durch Fortpflanzung wird ausgehend aus den fittesten Individuen eine neue Generation aufgebaut.

9.6.1 Known-Ciphertext Angriffe mit Genetischen Algorithmen (12 Punkte)

Ein großer Vorteil genetischer Algorithmen – z.B. gegenüber Backtracking – ist, dass diese normalerweise recht einfach implementierbar sind, da sie dem obigen simplen Schema folgen. Demgegenüber steht die Wahl der Parameter, die schnell komplex werden kann. Im folgenden werden wir sehen, dass im Falle von monoalphabetischen Chiffren jedoch bereits ein sehr einfacher genetischer Algorithmus ausreicht um die Chiffre in kurzer Zeit zu brechen.

Wir betrachten dazu die Interfaces `analysis.monoalphabetic.MonoalphabeticKnownCiphertextCryptanalysis` sowie `analysis.monoalphabetic.GeneticAnalysis`. Ersteres stellt wieder die Methoden der Analyse bereit. Das zweite Interface stellt die einzelnen Methoden zur Implementierung des genetischen Algorithmus zur Verfügung. Zunächst müssen Sie das Interface `analysis.monoalphabetic.Individual` implementieren. Ein Individuum besteht aus einem Alphabet und gegebenenfalls einem Fitnesswert.

Erstellen Sie anschließend eine Klasse die `MonoalphabeticKnownCiphertextCryptanalysis` sowie `GeneticAnalysis` implementiert. Beachten Sie, dass sie auch für diese Klasse wieder die entsprechende Methode in der `Factory` implementieren müssen. Nun gilt es den genetischen Algorithmus anhand der folgenden Methoden implementieren:

- Die Methode `prepareInitialGeneration` dient der Erzeugung der initialen Population. Eine Möglichkeit diese zu Erzeugen ist auf die Ergebnisse der Häufigkeitsanalyse zurückzugreifen. Beachten Sie, dass Individuen an dieser Stelle noch keine Fitness zugeordnet werden muss.

- Die Methode `computeFitness` bekommt als Eingabe ein Individuum (als Alphabet) und soll dessen Fitness feststellen. Zusätzlich erwartet die Methode den Schlüsseltext, das Quellalphabet, sowie die Häufigkeitsverteilung und ein Wörterbuch Überlegen Sie sich, wie Sie mittels `Distribution` und `Dictionary` eine geeignete Bewertung eines Individuums vornehmen können.
- Die Methode `computeSurvivors` wählt die besten Individuen (mittels `computeFitness`) aus und gibt diese in einer Liste zurück.
- Die Methode `generateNextGeneration` die Liste der besten Individuen, die Größe der zu erstellenden Population, ein Random Objekt für die Generierung von Zufall, sowie die üblichen Hilfsvariablen (Alphabet, Distribution und Dictionary). Es gibt eine Liste an neuen Individuen zurück.

Wie beim Backtracking sollten Sie zusätzlich die Methode `getState()` implementieren und eine sinnvolle Ausgabe des Zwischenstandes ausgeben. In unserer Implementierung sieht die (hier nicht weiter erläuterte) Anzeige wie folgt aus:

```
[a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z]
[u, n, i, z, o, r, q, h, y, x, g, k, c, j, e, b, s, m, d, f, w, p, a, v, t, l]
[i, a, h, r, o, d, v, u, y, s, c, g, l, j, b, p, n, m, z, q, e, f, x, k, t, w, ]
[0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, ]
fitness: 36.66138983054633
generations: 5110
stable: 1968.0
correct: 5
```

Parameterwahl

In unserer Lösung haben wir mit folgenden Parametern gearbeitet. Populationsgröße 10, Anzahl an Überlebenden 3. Für die Generation der neuen Generation haben wir eine einfache Mutation gewählt. Hierbei nehmen wir einen Überlebenden sowie eine Zahl zufällig im Intervall $s \in \{1,\ldots,$`Alphabet.size()`$\}$. Wir kopieren anschließend das Alphabet des Überlebenden und führen s viele zufällige Vertauschungen durch. Das Ergebnis fügen wir der neuen Generation ein. Als Fitnessfunktion verwenden wir eine gewichtete Häufigkeitsanalyse (kommen n-Gramme im Text vor, die in der Sprache häufig sind, so gibt es mehr Punkte). Zusätzlich untersuchen wir den entschlüsselten Text auf Vorkommnisse von Wörtern im Wörterbuch. Je mehr Wörter und je länger die Wörter, desto mehr Punkte vergeben wir.

Zusätzlich bedienen wir uns eines kleinen Kniffes. In der obigen Ausgabe sehen Sie den Zähler *stable*. Dieser zählt, wie lange keine Verbesserung mehr eingetreten ist. Das heißt, wir speichern uns das bis hierhin beste Individuum und setzen den Zähler jedes mal zurück, wenn wir ein besseres gefunden haben. Sollte sich lange Zeit (bei uns 2000 Generationen) keine Verbesserung einstellen, so versuchen wir die aktuelle Lösung mittels eines brute-force Angriffes zu lösen. Hier könnte man z.B. wieder die Backtracking-Analyse verwenden. Wir betrachten einfach alle möglichen Paare im Alphabet, tauschen diese und betrachten, ob das Ergebnis besser geworden ist. Finden wir die Lösung so, brechen wir ab. Finden wir „nur“ ein besseres Individuum, so setzen wir den Zähler stable zurück und laufen weiter. Finden wir keine bessere Lösung, lassen wir den Algorithmus einfach so weiter laufen, jedoch brechen wir nach 5000 Generationen ohne Verbesserung ab und starten erneut.

Anmerkung: Denken Sie für Ihre Implementierung auch an die Implementierung der entsprechenden `Factory`-Methode. Sie haben, wie beim Backtracking auch, Zugriff auf ein `ValidateDecryptionOracle`. Für die volle Punktzahl sollte Ihre Lösung allerdings auch mit einem selbst geschriebenen Orakel funktionieren.

Bepunktung:

1. korrekte Implementierung von `prepareInitialGeneration` (1 Punkt)
2. korrekte Implementierung von `computeSurvivors` (1 Punkt)
3. sinnvolle Implementierung von `computeFitness` (2 Punkt)
4. sinnvolle Implementierung von `generateNextGeneration` (2 Punkt)
5. sinnvolle Implementierung von `getState` (1 Punkt)
6. korrekte Implementierung des genetischen Algorithmus (2 Punkt)
7. bestehen der Testfälle (2 Punkt)

8. bestehen der Testfälle mit eigenem `ValidateDecryptionOracle`. Hierbei ist es ausreichend den Schlüssel möglichst genau zu bestimmen. (1 Punkt)

10 Kryptoanalyse der Vigenère-Chiffre (20 Punkte)

In diesem Abschnitt wenden wir uns nun nach der Kryptoanalyse monoalphabetischer Substitutionschiffren den polyalphabetischen zu – genauer gesagt, der Vigenère-Chiffre. Obwohl konzeptionell einfach zu verstehen und umzusetzen, erlangte die Vigenère-Chiffre nach Ihrer Erfindung bald den Beinamen „*le chiffre indéchiffrable*“ (französisch für: die nicht-entschlüsselbare Chiffre). In der Tat stellte das polyalphabetische Konzept hinter der Vigenère-Chiffre die Kryptoanalytiker vor enorme Schwierigkeiten. Es brauchte rund dreihundert Jahre, bis es dem deutschen Infanteriemajor und Kryptograph Friedrich Kasiski im 19. Jahrhundert gelang, eine allgemeinen Ansatz zum Brechen der Vigenère-Chiffre zu entwickeln. Seine Analysemethode ist heute unter dem Namen *Kasiski-Test* bekannt.

10.1 Der Kasiski-Test

Die Stärke polyalphabetischer Verfahren wie Vigenère – und damit die Schwierigkeiten ihrer Kryptoanalyse – beruht darauf, dass jedes Zeichen unter wechselnden Schlüsselbuchstaben chiffriert wird (vgl. Abschnitt 8). Die bei monoalphabetischen Chiffren so erfolgreiche Häufigkeitsanalyse wird damit unmöglich gemacht, da gleiche Klartextbuchstaben an verschiedenen Positionen zu unterschiedlichen Chiffrebuchstaben verschlüsselt werden und damit die charakteristische Häufigkeitsverteilung von Buchstaben in einer Sprache (vgl. Abschnitt 7.1) verwischen.

Dennoch weisen auch Vigenère-Chiffretexte eine gewisse Struktur auf, die darauf zurückzuführen ist, dass sich der benutzte Schlüssel wiederholt und somit – in gewissen, von der Schlüssellänge abhängenden Abständen – die gleiche monoalphabetische Substitutionstabelle zur Verschlüsselung von Klartextbuchstaben verwendet wird.[1] Kasiski erkannte diese Struktur und nutzte Sie wie folgt in der als Kasiski-Test bekannt gewordenen Analyse.

Betrachtet man das Zusammenspiel von Schlüsselwort, Klartext und Chiffretext, wie hier am Beispiel vom Klartext „`tobeornottobethatisthequestion`“ unter dem Schlüssel „`praktikum`“, kann man erkennen, dass gleiche Zeichenfolgen im Klartext manchmal gleiche Zeichenfolgen im Chiffretext hervorrufen:

Schlüsselwort	p	r	a	k	t	i	k	u	m	p	r	a	k	t	i	k	u	m	p	r	a	k	t	i	k	u	m	p	r	a
Klartext	t	o	b	e	o	r	n	o	t	t	o	b	e	t	h	a	t	i	s	t	h	e	q	u	e	s	t	i	o	n
Chiffretext	i	f	b	o	h	z	x	i	f	i	f	b	o	m	p	k	n	u	h	k	h	o	j	c	o	m	f	x	f	n

In unserem Beispiel ist das für die Klartextsequenz „`tobe`“ der Fall, welche zweimal im Klartext auftritt und zweimal zur gleichen Chiffretextsequenz „`ifbo`“ übersetzt wird. Da es (zumindest bei Sequenzen ab einer Länge von etwa 3 Zeichen) sehr unwahrscheinlich ist, dass eine solche Wiederholung durch verschiedene Klartextsequenzen hervorgerufen wurde, kann man aus dem Abstand an Zeichen zwischen den beiden Auftritten der Sequenz im Chiffretext Rückschlüsse auf die Länge des verwendeten Schlüssels ziehen. Die brillante Idee Kasiskis hierbei ist: Der Abstand von sich wiederholenden Chiffretextsequenzen ist mit hoher Wahrscheinlichkeit ein Vielfaches der Länge des verwendeten Schlüssels. Sucht man also in einem Chiffretext nach allen Wiederholungen von Zeichensequenzen (ab einer Länge von 3 Zeichen[2]), bestimmt die jeweiligen Abstände zwischen zwei solcher Wiederholungen und berechnet alle gemeinsamen Teiler dieser Abstände, so ist die Schlüssellänge mit hoher Wahrscheinlichkeit einer dieser Teiler.

Im oben gezeigten Beispiel würde man also den Chiffretext

Chiffretext	i	f	b	o	h	z	x	i	f	i	f	b	o	m	p	k	n	u	h	k	h	o	j	c	o	m	f	x	f	n
Position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30

[1] In der Tat verschwindet diese Struktur völlig, wenn sich der Schlüssel bei einer Verschlüsselung *nie* wiederholt. Beim Versuch, die Vigenère-Chiffre wieder sicher zu machen schlug der AT&T Ingenieur Gilbert Vernam Anfang des 20. Jahrhunderts vor, einen Schlüssel zu nutzen, der länger ist als die zu sendende Nachricht, und diesen auch nur einmal zu verwenden. Dieses später als *One-Time-Pad* bekannt gewordene Verfahren bietet in der Tat perfekte Sicherheit gegen jegliche Kryptoanalyse (jeder vermeintliche Klartext hinter einer Verschlüsselung ist gleich wahrscheinlich), wie der Mathematiker und Kryptograph Claude Shannon in einer grundlegenden, 1949 veröffentlichten Arbeit beweisen konnte. Leider ist es in der Praxis wenig einsatzfähig, da – wie Shannon ebenfalls bewies – der benutzte (zufällige, nicht wiederholende und geheime) Schlüssel mindestens die gleiche Länge haben muss wie die geheim zu haltende Nachricht um perfekte Sicherheit zu erreichen.

[2] Bei kürzeren Sequenzen ist die Wahrscheinlichkeit, dass die Wiederholung zufällig von verschiedenen Klartextsequenzen hervorgerufen wurde, zu groß.

auf sich wiederholende Zeichensequenzen absuchen und deren Abstände bestimmen. Diese Analyse (für Sequenzen ab der Länge 3) ergibt folgende Wiederholungen:

Zeichensequenz	Position erstes Auftreten	Position zweites Auftreten	Abstand
`ifb`	1	10	9
`fbo`	2	11	9
`ifbo`	1	10	9

Während in diesem einfachen Beispiel nun alle angetroffenen Abstände gleich sind, ist dies bei längeren Chiffretexten natürlich eher nicht der Fall. Da die gesuchte Schlüsselänge vermutlich ein Teiler *aller* gefundenen Abstände ist, berechnet man nun zunächst den größten gemeinsamen Teiler G aller Abstände (in diesem Fall ist natürlich $G = \text{ggT}(9, 9, 9) = 9$). Nun bestimmt man alle Teiler von G, was in diesem Beispiel die Zahlen 1, 3 und 9 sind. Die Länge des verwendeten Vigenère-Schlüssels ist nun mit hoher Wahrscheinlichkeit eine dieser Teiler – in unserem Beispiel ist es der größte Teiler, die 9.

10.2 Vigenère-Analyse: Bestimmung möglicher Schlüssellängen (8 Punkte)

Implementieren Sie nun als ersten Schritt in der Analyse des Vignenère-Chiffres den Kasiski-Test. Dazu implementieren Sie das Interface `analysis.vigenere.VigenereCryptanalysis` in einer neuen Klasse, wobei Sie zunächst die Interface-Methoden ignorieren (und für diese einfach **`null`**) zurückgeben können.

Schreiben Sie nun die folgende vier Hilfsmethoden für die spätere Kryptoanalyse von Chiffretexten, die den Kasiski-Test umsetzen:

- Eine Methode zum *Berechnen aller Teiler einer Zahl*, die eine natürliche Zahl als Parameter erhält und eine Liste mit allen Teilern dieser Zahl (in aufsteigender Sortierung) zurückgibt.

- Eine Methode zur *Berechnung des größten gemeinsamen Teilers einer Liste von Zahlen*, die eine Liste von natürlichen Zahlen als Parameter erhält und den größten gemeinsamen Teiler aller dieser Zahlen zurückgibt.
  Hinweis: Für den größten gemeinsamen Teiler ggT von n Zahlen $x_1, \ldots, x_n$ gilt $\text{ggT}(x_1, x_2, x_3, \ldots, x_n) = \text{ggT}(\text{ggT}(x_1, x_2), x_3, \ldots, x_n)$. Sie können den größten gemeinsamen Teiler also Schritt für Schritt für je zwei Zahlen berechnen (nutzen Sie zur Implementierung des ggT für zwei Zahlen den aus Vorlesung und Übung bekannten Algorithmus von Euklid).

- Eine Methode zur *Bestimmung aller Distanzen zwischen sich wiederholenden Zeichenketten ab einer bestimmten Länge in einem Text*, die einen String und eine minimale Länge für zu berücksichtigende Sequenzen als Parameter erhält und eine Liste aller Distanzen zwischen sich wiederholenden Zeichenketten zurückgibt, die mindestens die geforderte Länge haben.

- Eine Methode zur *Bestimmung aller vermutlichen Schlüssellängen*, die einen Chiffretext (String) als Parameter erhält und die vermuteten Schlüssellängen entsprechend des Kasiski-Tests als (aufsteigend sortierte) Liste von Zahlen zurückgibt. Die Methode bestimmt mithilfe der zuvor implementierten Hilfsmethoden also zunächst alle Distanzen im Chiffretext, berechnet dann den größten gemeinsamen Teiler dieser Distanzen und gibt schließlich alle Teiler dieses größten gemeinsamen Teilers als mögliche Schlüssellängen zurück.

Bepunktung:

1. Implementierung der Berechnung aller Teiler einer Zahl (1 Punkt)

2. Implementierung der Berechnung des größten gemeinsamen Teilers einer Liste von Zahlen (2 Punkte)

3. Implementierung der Bestimmung aller Distanzen von Zeichensequenzen im Text (3 Punkte)

4. Implementierung der Bestimmung aller vermutlichen Schlüssellängen (2 Punkte)

10.3 Known-Ciphertext-Angriff mit Kasiski-Test (2 Punkte)

Mithilfe der Implementierung des Kasiski-Tests aus dem vorangegangenen Abschnitt 10.2 können wir nun einen ersten einfachen Known-Ciphertext-Angriff auf die Vigenère-Chiffre umsetzen. Implementieren Sie dazu in Ihrer Analyse-Klasse für Vigenère aus dem Interface `analysis.vigenere.VigenereCryptanalysis` die Methode `List<Integer> knownCiphertextAttack(String)`, die lediglich einen Chiffretext zur Analyse erhält (ohne weitere Informationen wie bspw. einer Häufigkeitsverteilung der Buchstaben) und versucht, die möglicherweise genutzten Schlüssellängen für diesen Chiffretext zu ermitteln. Machen Sie sich dabei zunutze, dass die Bestimmung möglicher Schlüssellängen mittels Kasiski-Test allein auf der Basis des Chiffretext arbeitet und geben Sie in Ihrer Implementierung die Liste mit den ermittelten, potentiellen Schlüssellängen (in aufsteigender Sortierung) zurück.

Implementieren Sie zudem die Instanziierungsmethode des `Factory`-Interfaces für `VigenereCryptanalysis`.

Bepunktung:

1. Implementierung von `knownCiphertextAttack(String)` mittels Kasiksi-Test (1 Punkt)
2. korrekte Implementierung der `Factory`-Methode (1 Punkt)

10.4 Known-Plaintext-Angriff zu Bestimmung des Schlüssels (5 Punkte)

Widmen wir uns nun dem Known-Plaintext-Angriff (vgl. Abschnitte 7.4 und 9) auf Vigenère. Ein Chiffretext-Plaintext-Paar gibt offensichtlich direkt die verwendeten Schlüsselzeichen preis, die sich – ähnlich wie beim Known-Plaintext-Angriff auf Caesar – aus der jeweiligen Differenz der Positionen von Chiffrebuchstabe und Plaintextbuchstabe im Alphabet ergeben. Um aus der Sequenz der Schlüsselzeichen den verwendeten Schlüssel abzuleiten, ist allerdings noch ein weiterer Gedankenschritt nötig: Je nach Länge des Plaintextes und des Schlüssels wiederholt sich dieser in der Sequenz der Schlüsselzeichen vollständig, teilweise oder gar nicht, was man durch einen einfachen Vergleich von Wiederholungen in der ermittelten Schlüsselzeichensequenz aber leicht ermittelt.

Implementieren Sie nun in Ihrer Analyse-Klasse für Vigenère aus dem Interface `analysis.vigenere.VigenereCryptanalysis` die Methode `String knownPlaintextAttack(String, String, Alphabet)`, die einen Chiffretext sowie dazugehörigen Klartext sowie das benutzte Alphabet zur Analyse erhält und versucht, den genutzten Schlüssel (oder zumindest ein Präfix davon) zu ermitteln.

Bepunktung:

1. Bestimmung des korrekten Anfangsteils eines Schlüssels länger als der Klartext (Schlüsselzeichensequenz = Schlüssel) (1 Punkt)
2. Bestimmung eines korrekten Schlüssels, der sich (mindestens) zweimal in der Schlüsselzeichensequenz wiederholt (2 Punkte)
3. Bestimmung eines korrekten Schlüssels, der sich teilweise in der Schlüsselzeichensequenz wiederholt (2 Punkte)

10.5 Known-Ciphertext-Angriff mit Cribs (5 Punkte)

In unserem ersten Known-Ciphertext-Angriff (Abschnitt 10.3) haben wir lediglich die möglichen Schlüssellängen aus einem Chiffretext ermittelt. Nun wollen wir einen Known-Ciphertext-Angriff erstellen, der den vollen Schlüssel ermittelt und dafür auf Cribs (also bekannte Wörter im Plaintext, vgl. Abschnitt 9) zurückgreift.

Zunächst gehen wir hierbei wie in Abschnitt 10.3 vor und ermitteln alle möglichen Schlüssellängen. Nun können wir für jede Schlüssellänge (wobei wir sinnvollerweise mit der kürzesten beginnen) alle Zeichen im Chiffretext betrachten, die mit dem gleichen Schlüsselbuchstaben verschlüsselt wurden, und für diese – da sie quasi eine monoalphabetische Verschlüsselung darstellen – ihre Häufigkeitsverteilung bestimmen. Gehen wir nun, wie bei der Caesar-Chiffre, vereinfacht vor und wählen das Schlüsselzeichen so, dass das häufigste Zeichen des Alphabets auf das häufigste derjenigen Zeichen abgebildet wird, die mit dem Schlüsselzeichen verschlüsselt wurden, können wir jedes vermeintliche Schlüsselzeichen für einen Schlüssel der aktuell zu testenden Länge ermitteln.

An dieser Stelle kommen jetzt die Cribs zum Einsatz: Da wir wissen, dass diese im Klartext auftreten müssen, können wir den Chiffretext nun probeweise mit dem ermittelten Schlüssel für die aktuelle Schlüssellänge entschlüsseln und prüfen,

ob alle Cribs im Klartext enthalten sind. Ist dies der Fall, haben wir den richtigen Schlüssel gefunden und sind fertig, wenn nicht, verwerfen wir die aktuelle Schlüssellänge und probieren es mit der nächsten vermuteten Schlüssellänge.

Implementieren Sie nun in Ihrer Analyse-Klasse für Vigenère aus dem Interface `analysis.vigenere.VigenereCryptanalysis` die Methode `String knownCiphertextAttack(String, Distribution, List<String>)`, die einen Chiffretext, eine `Distribution` sowie eine Liste von Cribs zur Analyse erhält und versucht, den genutzten Schlüssel zu ermitteln.

Erstellen Sie für diese Analyse eine Hilfsmethode, die aus einem String für jede Folge von m Zeichen das n-te Zeichen in dieser Folge extrahiert und diese Zeichen zu einem String zusammengefügt zurückgibt. Nutzen Sie diese Hilfsmethode, um einen String mit denjenigen Zeichen aus dem Chiffretext zu extrahieren, die mit dem gleichen Schlüsselzeichen verschlüsselt wurden (n ist hier das n-te Zeichen des Schlüssels, m ist die Länge des Schlüssels). Verwenden Sie den so extrahierten String dann für die Berechnung der Häufigkeitsverteilung für die Chiffrezeichen unter diesem Schlüsselzeichen.

Bepunktung:

1. Implementierung der Hilfsmethode zum Extrahieren der unter einem Schlüsselzeichen verschlüsselten Chiffrezeichen (2 Punkte)
2. Implementierung von `knownCiphertextAttack` mittels Cribs (3 Punkte)

11 Enigma (10 Punkte)

Nach dem ersten Weltkrieg suchte man nach Alternativen zu den bereits bekannten Chiffren, da diese sehr aufwändig in der Durchführung und vor allem nicht mehr sicher waren. Eine der Personen, die sich mit diesem Problem beschäftigte war der promovierte deutsche Elektrotechniker Arthur Scherbius, der als der Erfinder der berühmt gewordenen *Enigma*[1] gilt. Seine Idee lässt sich als Weiterentwicklung der Chiffrierscheibe[2] betrachten. Eine Chiffrierscheibe bestand aus 2 Ringen, in denen das Alphabet aufgeschrieben war und die somit durch Drehung bewirkte, dass sich das komplette Vigenère-Quadrat abbilden lies. Die Idee die Scherbius nun hatte war die neu gewonnene Technik der elektrischen Schreibmaschine in Kombination mit mechanisch rotierenden Scheiben zu verwenden, um die Verschlüsselung von Texten zu automatisieren. Auf Basis dieser Idee entstand die erste Enigma, welche noch als zivile Chiffriermaschine gedacht war. Durch einige Weiterentwicklungen entstand eine Maschine, welche aufgrund ihrer bedeutenden Rolle während des zweiten Weltkrieges zur damals gefürchtetsten Chiffriermaschine avancierte. In diesem Kapitel wollen wir eine vereinfachten Form der Enigma implementieren.

Das Modell, das wir hier betrachten, besaß eine Tastatur (ähnlich der einer Schreibmaschine zur Eingabe der Buchstaben) sowie Lampen, welche genau so angeordnet waren wie die Tasten der Tastatur. Drückte man eine der Tasten, wurde der gedrückte Buchstabe automatisch chiffriert und die Lampe des entsprechend Chiffrebuchstabens leuchtete auf. Die Besonderheit dieser „Schreibmaschine“ war, dass das mehrmalige Drücken der selben Taste immer verschiedene Lampe zum aufleuchten brachte. Diese Funktionalität – also die Erzeugung einer polyalphabetischen Verschlüsselung – wurde durch das Zusammenspiel der folgenden Bauteile bewirkt:

- ein *Steckbrett*, auf dem Buchstaben direkt nach der Eingabe und vor der weiteren Verarbeitung vertauscht werden konnten
- drei, je nach Bauart aber auch mehr, *Walzen* die für jeden der 26 Buchstaben einen elektrischen Eingangs- und Ausgangskontakt hatten und durch die interne Verkabelung eine Substitutionstabelle über die 26 Buchstaben darstellten
- eine *Umkehrwalze*, die als letzte Walze die Kontakte von je zwei Buchstaben verband und damit das Signal wieder zurück durch die anderen Walzen und schließlich zu der Lampe des erzeugten Chiffrebuchstabens leitete

Ein Ausschnitt der schematischen Darstellung des Schaltplans der Enigma und Ihrer Bauteile ist in Abbildung 11 zu sehen.

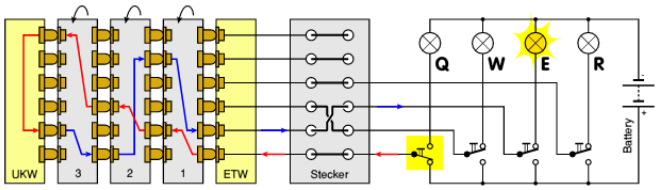

Abbildung 11.1.: Schematische Darstellung der Enigma[3]

Jeder Tastendruck auf der Enigma bewirkt einen Stromfluss zunächst durch das Steckbrett, dann durch die Walzen (Hinrichtung), die Umkehrwalze und anschließend zurück durch die Walzen (Rückrichtung) und das Steckbrett. So sieht man, in Abbildung 11 wie die Taste `Q` gedrückt wird und die Lampe von `E` leuchtet – `Q` wurde also zu `E` verschlüsselt.

Nach jedem Tastendruck rotiert die erste Walze weiter, sobald die erste Walze eine volle Umdrehung vollzogen hat, rotiert die zweite Walze zusätzlich um eine Position, hat auch diese eine volle Umdrehung vollzogen, die dritte Walze etc.

[1] Mit der Beschreibung der Entwicklung der Enigma sowie der Geschichte ihrer Kryptoanalyse lassen sich ganze Kapitel von Büchern füllen (die Bücher von Kahn [2] und Singh [3] sind zwei gute Beispiele dafür), sodass wir im Rahmen dieser Praktikumsaufgabenstellung nur einen flüchtigen Blick auf eine vereinfachte Konstruktion einer Enigma werfen und ihre Kryptoanalyse leider gar nicht behandeln können. Als Einstieg zu einer tiefgründigeren Beschäftigung mit der Enigma und der spannenden Geschichte, wie sie im zweiten Weltkrieg von den Allierten geknackt wurde, eignet sich – neben den genannten Büchern [2, 3] – beispielsweise die entsprechende Seite der deutschen Wikipedia unter `https://de.wikipedia.org/wiki/Enigma_%28Maschine%29`.

[2] siehe z.B. `http://de.wikipedia.org/wiki/Chiffrierscheibe`

[3] Quelle: *Crypto Museum*, mit freundlicher Genehmigung, `http://www.cryptomuseum.com/crypto/enigma/working.htm` (16.02.2014)

Dadurch leuchtet also bei jedem weiteren Drücken der Taste `Q` also ein anderer Buchstabe auf (bis alle Walzenpositionen durchlaufen wurden).

Das Steckbrett besaß für gewöhnlich nicht – wie in der Abbildung dargestellt – die Möglichkeit die Buchstaben frei zu verdrahten, sondern ließ es nur zu, dass Verbindungen zwischen 2 Buchstaben gesteckt werden konnten, um diese beiden Buchstaben zu vertauschen. Nicht gesteckte Buchstaben waren automatisch mit sich selbst verbunden waren und wurden also nicht verändert. Dies bedeutet, dass wenn – wie in der Abbildung – `W` mit `E` verbunden ist, automatisch auch `E` mit `W` verbunden ist.

Während des zweiten Weltkrieges besaß die Enigma für gewöhnlich drei Walzen, welche nach einem Tagesschlüssel aus einem Set von fünf Walzen ausgewählt wurden. Zusätzlich gab es ein sich täglich änderndes Steckmuster und eine tagesaktuelle Ausgangsstellung der Walzen, die die initiale Position für jede Walze festgelegt hat. Dadurch ergab sich eine so hohe kryptographische Komplexität, dass es großen kryptoanalytischen Geschicks bedurfte, um die Enigma zu brechen.

11.1 Die Bauteile (7 Punkte)

Wir beginnen nun mit der Implementierung der einzelnen Bestandteile der Enigma, also des Steckbretts, der Walzen sowie der Umkehrwalze.

Implementieren Sie zunächst das Interface `cipher.substitution.enigma.Rotor`. Dieses Interface repräsentiert eine der Walzen der Enigma. Solch eine Walze übersetzt von einem `Alphabet` in ein anderes, ist jedoch auch in der Lage die selbe Übersetzung rückwärts durchzuführen (siehe Beschreibung von `translate`). Zu beachten hierbei, dass eine Walze rotieren kann und sich dabei die komplette Verdrahtung sich bewegt, was *nicht* durch einen simples Shiften des Alphabets abgebildet werden kann. Schematisch bedeutet dies bspw. bei einem Alphabet `A,B,C,D` z.B.:

```
A +1 -> B           A +1 -> B           A -2 -> C           A +0 -> A           A +1 -> B
B +1 -> C  Rotation B -2 -> D  Rotation B +0 -> B  Rotation B +1 -> C  Rotation B +1 -> C
C -2 -> A  =======> C +0 -> C  =======> C +1 -> D  =======> C +1 -> D  =======> C -2 -> A
D +0 -> D           D +1 -> A           D +1 -> A           D -2 -> B           D +0 -> D
```

Eine solche Rotation wird durch die Methode `rotate` bewirkt. Zusätzlich kann eine Walze auch auf ihre Startposition zurückgesetzt werden, wobei eine Startposition für eine feste, beim Start übergebene Position in der Verdrahtungsliste steht.

Als nächstes sind die Implementierungen von `cipher.substitution.enigma.ReverseRotor` und `cipher.substitution.enigma.PinBoard` an der Reihe. Da sich beide Bauteile nur in einem Punkt unterscheiden, fassen wir diese hier etwas zusammen. Das Interface `PinBoard` repräsentiert das Steckbrett der Enigma. Zu beachten ist, dass dieses Bauteil zwei Buchstaben mit Kabeln verbinden kann, allerdings nicht muss. Dies bedeutet zweierlei Dinge:

- Eingabe und Ausgabealphabet müssen symmetrisch sein, sprich wenn `a` zu `k` übersetzt wird, muss automatisch auch `k` zu `a` übersetzt werden.
- Manche Buchstaben können unverbunden bleiben und bilden damit automatisch auf sich selbst ab. Wird beispielsweise keine Vertauschung für `a` angegeben, ist automatisch `a` mit `a` „verbunden“.

Die Umkehrwalze ist im Gegensatz zum Steckbrett ein festes Bauteil und bietet daher nicht Möglichkeit, Buchstaben unverbunden zu lassen. Dies bedeutet, dass die Übersetzung von Eingabealphabet zu Ausgabealphabet symmetrisch sein muss, allerdings – da es ein Stromkreis ist – nicht reflexiv (`a` darf nicht auf `a` abbilden) sein darf.

Achten Sie bei Ihrer Implementierung darauf, dass die jeweiligen Konstruktoren mit den entsprechenden Instanziierungmethoden aus dem `Factory`-Interface zu erfüllen sind. Achten Sie außerdem darauf, geeignete Fehlermeldungen bei falschen Instanziierungen (insbesondere bei inkompatiblen Alphabeten) zu werfen.

Bepunktung:

1. Implementierung von `Rotor` (3 Punkt)
2. Implementierung von `ReverseRotor` (2 Punkte)
3. Implementierung von `PinBoard` (2 Punkte)

11.2 Die eigentliche Enigma (3 Punkte)

Nun, da wir nötigen die Bauteile zusammen haben, ist es an der Zeit die Enigma „zusammenzubauen“. Hierzu implementieren Sie bitte das Interface `cipher.substitution.enigma.Enigma`. Zu beachten ist, dass hier die korrekte Funktionsweise der Enigma nachempfunden werden soll, sprich: Die Verschlüsselung eines Buchstabens durchläuft zunächst das Steckbrett, dann alle Walzen vorwärts, die Umkehrwalze, alle Walzen rückwärts in umgekehrter Reihenfolge und anschließend noch einmal das Steckbrett. Anschließend rotieren die Rotoren weiter. Die Rotation der Rotoren funktioniert wie bei einer Uhr, der erste Rotor dreht immer einen Schritt weiter, der zweite dreht einen Schritt weiter, sobald der erste Rotor eine volle Umdrehung vollzogen hat (d.h., wieder auf der Ausgangsposition steht), der dritte sobald der zweite eine volle Umdrehung vollzogen hat usw.

Die Entschlüsselung funktioniert auf dieselbe Art und Weise wie die Verschlüsselung (dies wird durch die Bauweise der Umkehrwalze sichergestellt). Beachten Sie hierbei, dass die Enigma nach jedem Text wieder in den Ursprungszustand versetzt werden muss, die Walzen also wieder in ihre Ausgangsstellung gedreht werden müssen.

Implementieren Sie auch die Instanziierungmethode im `Factory`-Interface für `Enigma`.

Bepunktung:

1. Implementierung von `Enigma` (3 Punkte)

12 Eigene Erweiterungen (10 Punkte)

Die Tutoren können kreative Erweiterungen in der Bewertung Ihrer Abgabe berücksichtigen. Es geht in dieser Aufgabe nicht darum, ein Stückchen von einer unserer vorangegangenen Aufgabenstellungen abzuweichen, sondern um eine größere Erweiterung, die Sie eigenständig entwickelt haben. Überraschen Sie Ihren Tutor mit einer interessanten Idee!

Beispiele hierfür könnten sein:

- eine vollständige Kryptoanalyse der Vigenère-Chiffre
- eine Kryptoanalyse der Enigma
- ein steganographisches[1] Verfahren zum Verstecken eines Textes
- eine Steuerung für Ihre Implementierung von der Kommandozeile/dem Terminal aus
- eine Oberfläche zur Generierung von Zufall für die Schlüsselerzeugung (z.B. durch Mausbewegung)
- ...

[1] Steganographie (siehe z.B. `https://de.wikipedia.org/wiki/Steganographie`) ist die Kunst bzw. Wissenschaft, eine Information (z.B. einen Text) innerhalb eines anderen Objekts zu verstecken, sodass diese Information möglichst unbemerkt übertragen werden kann. Beispielsweise könnte man Bilder innerhalb eines Videos verstecken, indem man diese jeweils nach 30 Frames einfügt – das menschliche Auge kann dieses eine aus 30 Bildern nicht mehr als eigenständiges Bild wahrnehmen, ein Computer kann es aber natürlich wieder extrahieren.
An dieser Stelle sei noch gesagt, dass es sich bei Steganographie *nicht* um eine Form der Verschlüsselung sondern um eine eigenständige Disziplin handelt.

Literaturverzeichnis

[1] CrypTool-Online. `http://www.cryptool-online.org`.

[2] D. Kahn. *The Codebreakers: The Comprehensive History of Secret Communication from Ancient Times to the Internet*. Scribner, 1996.

[3] S. Singh. *The Code Book: The Secret History of Codes and Code-Breaking*. Fourth Estate, 1999.

Changelog

Im Verlauf des Praktikums wurden die folgenden Änderungen (nach Versionsnummer gelistet) an dieser Aufgabenstellung sowie der bereitgestellten Datei `framework.jar` vorgenommen.

Version 1.1

- **Abschnitt 6.1.1:** Implementierungsanweisung den Ver- und Entschlüsselungsmethode präzisiert, Hinweis zu Mockito-Tests ergänzt.
- **Abschnitt 7.3/7.4:** Das Package `analysis.caesar` wurde fälschlicherweise als `analysis.caeser` referenziert.
- **Kapitel 8:** Fehler im einleitenden Beispiel korrigiert (die Substitutionszeilen (2) und (3) enthielten fälschlicherweise doppelte Buchstaben).
- `framework.jar`:
  - Beschreibung der Methode `Utils.toDisplay` in `utils/Utils.java` korrigiert (vgl. `https://moodle.informatik.tu-darmstadt.de/mod/forum/discuss.php?d=27369#p97645`).
  - Test der Methode `Utils.toDisplay` in `test.utils.TemplateUtilsTests` korrigiert (Leerzeichen am Ende der ersten Zeile wurde entfernt).
  - Das Interface `CaesarCryptanalysis` erbt nun (wie in Abschnitt 7.4 verwendet) auch vom Interface `KnownPlaintextAnalysis`.
  - Hinweis zu Mockito-Tests in `test.cipher.substitution.TemplateSubstitutionCipherTests` als Kommentar ergänzt.
  - Tippfehler im Test `test.alphabet.TemplateDistributionTests.testDistributionIsNormalized` korrigiert.

Version 1.0

Initiale Version.