Security in der Java-Grundausbildung

Andrea Baumann · Dieter Pawelczak

Universität der Bundeswehr München
Institut für Software Engineering
{andrea.baumann | dieter.pawelczak}@unibw.de

Zusammenfassung

Das Thema Security ist in der Java-Grundausbildung bisher in der Regel nicht fest verankert. In der Literatur bekommt man zwar Hilfestellungen für die robuste bzw. sichere Programmierung, jedoch sind diese für einen direkten Einsatz in einer einführenden Lehrveranstaltung noch zu komplex. Daher befasst sich dieses Dokument mit der Frage, wie schon zu Beginn eine Integration bzw. eine stärkere Fokussierung auf sicherheitsrelevante Aspekte möglich ist. Dazu werden die von Oracle vorgeschlagenen *Secure Coding Guidelines for Java SE* analysiert und mit den bisherigen Inhalten, wie sie im Studiengang *Technische Informatik und Kommunikationstechnik* an der *Universität der Bundeswehr München* gelehrt werden, abgeglichen. Wir geben konkrete Programmierbeispiele im Hinblick auf sichere Software und klären, wie diese frühzeitig in der Ausbildung angewandt werden können. Gut ein Drittel der von Oracle vorgeschlagen Sicherheitsrichtlinien können anhand der hier vorgestellten Beispiele mit wenig Aufwand adressiert werden. Der Schwerpunkt liegt dabei auf dem Schutzziel der Verfügbarkeit und weniger auf der Informationssicherheit. Zum Abschluss diskutieren wir die gewählten Lehrinhalte und mögliche Alternativen für eine intensivere Beschäftigung mit dem Thema Security.

1 Motivation

Die Bedeutung, sichere Software zu entwickeln, ist nicht erst durch das Bekanntwerden von Angriffen auf die IT-Infrastruktur gewachsen. Die GI (Gesellschaft für Informatik) und die ACM (Association for Computing Machinery) fordern bereits seit der Jahrtausendwende die Ausbildung in der Informatik im Hinblick auf sichere Softwareentwicklung anzupassen. In der Regel erfolgt dies in Aufbaukursen wie *IT-Sicherheit* oder *Sicheres Softwareengineering* [Info16, ACM16].

Für die Informationssicherheit sind für ein System die folgenden drei Schutzziele von zentraler Bedeutung: die Verfügbarkeit (engl. availability), die Integrität (engl. integrity) und die Vertraulichkeit (engl. confidentiality) [Somm12]. Diese Schutzziele müssen unter Beachtung der geltenden Sicherheitsrichtlinien erfüllt werden. Dazu ist z.B. die Authentifizierung und Autorisierung des Akteurs notwendig. Ein Fehler in der Programmierung, der ein System zum Absturz bringt, ist ebenso fatal, wie ein unerlaubter Zugriff auf geschützte Information. Dieser Aspekt ist insbesondere für die Schulung von Programmieranfängern sehr wichtig, da hier der Grundstein für möglichst saubere, fehlerfreie und durchdachte Programmierung gelegt wird.

Viele Studierende beginnen mit der Sprache Java erste Programmierkenntnisse zu sammeln. Dabei wird an Hochschulen und Universitäten zunächst auf die Syntax und die Semantik der Sprachelemente eingegangen, während das Thema Security meist außen vor bleibt. Ebenso wird die Thematik in Lehrwerken für Java in den hinteren Kapiteln [KrHa14, Ulle14] oder überhaupt

P. Schartner · A. Baumann (Hrsg.) · D•A•CH Security 2017 · syssec (2017) 373-387.

nicht behandelt [SeWa11]. Auf der anderen Seite orientieren sich Bücher zum Thema Security oft mehr am Entwicklungsprozess, vgl. [SFBHB+13], oder sind für Anfänger zu komplex [Bloc08]. Das Standardwerk *Building Secure Software* von Viega und McGraw [ViMc11] diskutiert sehr detailliert Technologien und Systeme, die aber zu Beginn des Studiums wegen der Komplexität noch nicht behandelt werden können.

Sicherlich sind für Programmieranfänger nicht alle Aspekte eines sicheren Softwaresystems begreifbar, daher wird die Thematik bisher nur zögerlich in die Grundausbildung einbezogen. Laut [TsCM05] ist es allerdings wichtiger, praktikable Umsetzungsmöglichkeiten zu propagieren, als einen Softwareentwickler mit der gesamten Tiefe des Themas zu konfrontieren. Wir werden darlegen, dass IT-Security schon zu Beginn der Programmierausbildung adressiert werden kann, ohne die Einstiegshürden zu erhöhen. Im Gegenteil hier kann, sollte bzw. muss der Grundstein gelegt werden, damit später möglichst sicherer Programmcode entwickelt wird [WYYB14].

2 Aufbau der Java-Grundausbildung

Das Modul *Grundlagen der Programmierung* wird bei uns im ersten Studienjahr gelesen und umfasst 6 ECTS Leistungspunkte – das entspricht 180 Stunden. Die Leistung wird von den Studierenden über 12 Wochen in einer 5-stündigen Vorlesung und einem 3-stündigen Praktikum mit einer anschließenden 90-minütigen Prüfung erbracht. Inhalte sind primitive Datentypen, Variablen, die funktionale Programmierung mit statischen öffentlichen Methoden und Rekursion, die imperative bzw. prozedurale Programmierung mit Schleifen und Verzweigungen, Sortieralgorithmen, die objektorientierte Programmierung mit Klassen und Schnittstellen, die Implementierung von Warteschlange und Stack mit Hilfe von Feldern und Listen.

Vertiefung der Objektorientierung mit Generics und Collections, Ausnahmebehandlung, Pattern und Matcher, Threads, der Umgang mit Dateien, Serialisierung bzw. Deseralisierung von Objekten, RMI sowie Oberflächen-Programmierung (JavaFX) werden erst in weiterführenden Lehrveranstaltungen unterrichtet.

Sowohl in der Literatur als auch auf Wiki-basierten Webforen, wie z.B. [Bloc08, Orac15, CERT17], bekommt man Hilfestellungen für die robuste bzw. sichere Programmierung. Allerdings gilt für diese Publikationen, dass ein direkter Einsatz für Studierende zu Beginn ihrer Ausbildung ungeeignet ist, da sie Sprachelemente und Bibliotheken enthalten, die erst später gelehrt werden.

3 Konkrete Beispiel zur Integration von Security

Wir analysieren die *Secure Coding Guidelines for Java SE* (SCG) von Oracle [Orac15] für den Einsatz in unserer einführenden Lehrveranstaltung. Diese werden von Oracle direkt gepflegt und weisen im Gegensatz zu Wiki-basierten Webforen eine feste Struktur auf: Die SCG sind in elf Bereiche eingeteilt, siehe Tabelle 1.

Wie erstellen auf Basis der SCG einfache, für Studierende nachvollziehbare Beispiele und zeigen daran potentielle Fallstricke bei der Java Programmierung auf, diskutieren diese und geben Lösungsvorschläge. Zusätzlich verweisen wir auf die Richtlinien aus den SCG[1], die zur Vertiefung der Thematik dienen können.

[1] Die im Text angegebenen Nummern entsprechen den Angaben in den SCG. Die erste Stelle gibt den Bereich an, dem die Richtlinie zugeordnet ist.

Tab. 1: Anzahl der Richtlinien in den elf Bereichen der Oracle Secure Coding Guidelines [Orac15]

Englische Originalbezeichnung	Bereich	Anzahl
0 - Fundamentals	Grundlagen	8
1 - Denial of Service	Nichtverfügbarkeit von Diensten	3
2 - Confidential Information	Vertrauenswürdige Information	3
3 - Injection and Inclusion	Injektion und Inklusion	9
4 - Accessibility and Extensibility	Erreichbarkeit und Erweiterbarkeit	6
5 - Input Validation	Überprüfung der Eingabe	3
6 - Mutability	Veränderbarkeit	12
7 - Object Construction	Objekterstellung	5
8 - Serialization and Deserialization	Serialisierung und Deserialisierung	5
9 - Access Control	Zugriffskontrolle	13
A - Defensive use of the JNI (Java Native Interface)	Defensive Verwendung des JNI	9

3.1 Clean Code

Im Themenbereich *Fundamentals* ist in den SCG eine der wichtigsten Richtlinien aufgelistet, die nicht nur im Hinblick auf die Informationssicherheit relevant ist, sondern jedem Programmieranfänger ans Herz gelegt werden muss: 0-0 *Prefer to have obviously no flaws rather than no obvious flaws*, was soviel heißt, wie: bevorzuge Programmcode, der offensichtlich keine Mängel enthält, gegenüber Programmcode, der keine offensichtlichen Mängel beinhaltet. Diese Richtlinie bezieht sich auf ein Zitat von C. A. R. Hoare. *There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult [Hoar81].* Programmcode muss so gestaltet werden, dass sich keine Mängel verstecken können. Das ist viel schwieriger, als Programmcode zu schreiben, der so kompliziert ist, dass Mängel gar nicht erst entdeckt werden können. Das Ziel schöner Programmierung ist: Schreibe möglichst eleganten Code, der die innere Logik widerspiegelt.

In diesem Zusammenhang diskutiert Robert C. Martin in seinem Buch *Clean Code* [Mart08], wie die Lesbarkeit von Programmcode, z.B. durch eine sinnvolle Benennung der Identifikatoren im Code, erhöht werden kann. Gerade für Programmieranfänger ist die Lesbarkeit von fremdem und eigenem Code enorm wichtig, da so zum einen beim Lesen von fremdem Code das Verständnis für die Programmierung insgesamt gefördert wird, aber auch das Erklären des eigenen Programms beim Code-Review unterstützt wird.

Listing 1: Maximum berechnen (korrekt)

```
1  static int getMaximum(int[] values) {
2     int max = values[0];
3     for (int i = 1;
4              i < values.length;
5              i++) {
6        if (max < values[i]) {
7           max = values[i];
8        }
9     }
10    return max;
11 }
```

Listing 2: Maximum berechnen (fehlerhaft)

```
1  static int max(int[] x) {
2     int i = x[0];
3     for (int j = 1;
4              j < x.length;
5              j++) {
6        i = (x[j] > i)
7              ? x[j]
8              : x[i];
9     }
10    return i;
11 }
```

In beiden Listings 1 und 2 sind zwei verschiedene Vorschläge bzw. Versuche zum Auffinden

des maximalen Integer-Wertes in einem Feld umgesetzt. Die Lesbarkeit der Variante 2 ist deutlich schlechter verglichen mit Variante 1: Sowohl die Verwendung der if-Anweisung als auch die Wahl sinnvoller Namen für die Identifikatoren erhöhen die Lesbarkeit. Zur Speicherung des Maximums wurde im Listing 2 der Name `i` verwendet, der oft einen Index repräsentiert. Dadurch kommt es zu einem Fehler, da `i` sowohl zum Speichern von Werten, als auch einmal als Index in Zeile 8 verwendet wird.

In diesem Kontext lässt sich noch eine weitere Richtlinie besprechen: 0-7 *Document security-related information* beschreibt die Notwendigkeit das Verhalten einer Methode transparent zu machen. Für unsere Umsetzung muss z.B. dokumentiert werden, dass die Methode `getMaximum` nur mit nicht leeren Feldern funktioniert. Für eine spätere Verwendung dieser Methode durch einen anderen Programmierer ist das aber entscheidend, da eine fehlerhafte Anwendung das gesamte Programm zum Absturz bringen kann.

3.2 Matrixmultiplikation

Um den Studierenden den Umgang mit Feldern näher zu bringen, wird die Matrixmultiplikation betrachtet. Zuerst soll dazu eine Klassenmethode geschrieben werden, die überprüft, ob es sich bei der Eingabe eines zweidimensionalen Feldes vom Typ `double[][]` um eine im mathematischen Sinn korrekte Matrix mit endlichen `double`-Werten handelt. Die Methode soll auf alle Eingaben eine gültige Antwort liefern, d.h. es darf keine Ausnahme auftreten. Sie muss nicht-endliche `double`-Werte erkennen und ggf. `false` zurückgeben.

Listing 3: Klassenmethode `isMatrix`

```
1  public static boolean isMatrix(double[][] matrix) {
2     if (matrix == null
3        || matrix.length == 0 || matrix[0] == null || matrix[0].length == 0) {
4        return false;
5     }
6
7     for (int row = 0; row < matrix.length; row++) {
8        if (matrix[row] == null || matrix[row].length != matrix[0].length) {
9           return false;
10       }
11       for (int column = 0; column < matrix[row].length; column++) {
12          if (!Double.isFinite(matrix[row][column])) {
13             return false;
14          }
15       }
16    }
17
18    return true;
19 }
```

Listing 3 zeigt eine Beispiellösung. Es wird überprüft, ob überhaupt eine Referenz übergeben wird (Zeile 2), ob die Matrix mindestens eine sinnvolle Zeile hat (Zeile 3), ob jede Zeile vorhanden und gleich lang ist (Zeile 8) und ob die Matrix ausschließlich endliche `double`-Werte enthält (Zeile 12). An diesem Beispiel lernen die Studierenden nicht nur den Umgang mit sogenannten *Jagged Arrays*, sondern auch wie wichtig es ist, eine Eingabe intensiv zu überprüfen, da dies sonst zu einem Programmabsturz führen kann.

Vielen Programmieranfängern fehlt hier das Verständnis, dass Schnittstellen im Laufe der Softwareentwicklung auch fehlerhaft bedient werden können und dass Angriffe auf Systeme häufig solche Schwachstellen nutzen. Anhand der Funktion `isMatrix()` lässt sich sehr schön zeigen, dass selbst skurrile Eingaben, wie z.B. `null`, `new double[][]{}`, `new double[][]{null}`

oder auch `new double[][]{{Double.NaN}}` den Wert `false` zurück liefern und keine `NullPointerException` oder `IndexOutOfBoundException` werfen.

Die SCG thematisieren die Validierung von Eingaben unter dem Punkt 5-1 *Validate inputs*. Es muss klar definiert sein, welche Eingaben eine Schnittstelle erwarten muss bzw. kann. In Zeile 12 wird überprüft, ob es sich bei den eingegebenen Werten überhaupt um endliche `double`-Werte handelt. Hier lohnt es sich über die folgende scheinbar äquivalente Lösung zu diskutieren:

```
if (value == Double.NaN
        || value == Double.NEGATIVE_INFINITY
        || value == Double.POSITIVE_INFINITY) {
                return false;
}
```

Entgegen der Erwartung der meisten Studierenden liefert die Methode jetzt bei Eingabe `new double[][]{{Double.NaN}}` den Wert `true` zurück. Der Vergleich mit `Double.NaN` liefert in Java immer den Wert `false`, deswegen muss `Double.isFinite()` oder `!Double.isNaN()` verwendet werden. In den SCG wird dies unter dem Punkt 3-9 *Prevent injection of exceptional floating point values* thematisiert. In diesem Zusammenhang sollten auch die Eigenarten, die bei der Typumwandlung von `double`- nach `int`-Werten auftreten, diskutiert werden.

Auch von Benutzereingaben können ungültige Werte, wie `NaN` mit der Klassenmethode `Double.parseDouble(String)` gelesen und verarbeitet werden. Hier kann man die Richtlinie 3-1 *Generate valid formatting* ansprechen. Diese fordert, dass vor jeglicher weiteren Bearbeitung der Benutzereingabe diese zuerst validiert werden muss. Falsche Eingaben sollten abgewiesen und nicht korrigiert werden, wie z.B. `NaN` zu `0`.

Im Anschluss implementieren die Studierenden eine Methode zur Matrixmultiplikation. Gehen wir nun mal davon aus, dass wir diese Methode nicht selbst, sondern von einer nicht vertrauenswürdigen Quelle erhalten haben, dann sollten wir nach Ausführung die Rückgabe mit unserer `isMatrix`-Methode überprüfen und z.B. nicht einfach davon ausgehen, dass die resultierende Matrix im mathematischen Sinn korrekt ist (vgl. 5-2 *Validate output from untrusted objects as input*).

3.3 Feld-basierter Stack

Ein weiteres Thema, das wir in der Programmierausbildung behandeln, ist die Implementierung grundlegender Datenstrukturen. Die Studierenden sollen so ein Verständnis für die Arbeitsweise und die Komplexität der von den Datenstrukturen angebotenen Operationen bekommen und können mit diesem Wissen später die richtige Auswahl aus vorhandener Klassen, wie z.B. `ArrayList` oder `LinkedList`, treffen. Konkret betrachten wir die Implementierung einer Warteschlange und eines Stacks mit einer einfach verketteten Liste und einem Feld. Im Folgenden zeigen wir, welche sicherheitsrelevanten Themen man im Zusammenhang mit der Feldbasierten Stack Implementierung ansprechen kann. Eine erste, noch nicht perfekte Beispiel-Implementierung ist im Listing 4 angegeben.

Das Listing zeigt die Implementierung der grundlegenden Operationen, die ein Stack minimal zur Verfügung stellen muss. Aktuell kann der Stack maximal 10000 Elemente aufnehmen. Wird auf ein Element zugegriffen, obwohl der Stack leer ist, dann erhält man eine `ArrayIndexOutOfBound` Ausnahme. Dies ist auch der Fall, falls man versucht das 10001 Element auf den Stack zu legen.

Listing 4: Stack Implementierung basierend auf Feld

```
1  public class Stack {
2     private int top = 0;
3     private Object[] stack = new Object[10000];
4
5     public boolean isEmpty() {
6        return top == 0;
7     }
8     public void push(Object element) {
9        stack[top++] = element;
10    }
11    public Object pop() {
12       return stack[--top];
13    }
14    public Object first() {
15       return stack[top-1];
16    }
17 }
```

Ausgehend von dieser Implementierung kann man die Richtlinie 1-1 *Beware of activities that may use disproportionate resources* diskutieren. Im obigen Beispiel wird dieser Grundsatz verletzt, da unabhängig von der tatsächlichen Anzahl der referenzierten Elemente schon zu Beginn der gesamte Speicher angefordert wird, obwohl dieser evtl. gar nicht benötigt wird. Verwendet jemand diese Implementierung mehrfach, dann kann das ohne Not zu einem `OutOfMemoryError` führen. Besser ist die folgende Umsetzung der `push`-Methode, die je nach Bedarf den Speicherplatz erhöht (siehe Listung 5).

Listing 5: Verbesserte `push`-Methode der Stack Implementierung

```
1     // ...
2     private Object[] stack = new Object[16];
3     // ...
4     public void push(Object element) {
5        if (top == stack.length)
6           stack = Arrays.copyOf(stack, stack.length*2);
7        stack[top++] = element;
8     }
9     // ...
```

Bleiben wir noch einen Gedanken länger bei der angepassten `push`-Methode und fragen uns: Wie oft geht so eine Erweiterung gut? Entweder ist kein Speicher mehr vorhanden und dadurch kommt es zu einem `OutOfMemoryError` oder es findet ein Überlauf statt. Letzteres fordert eine negative Felderweiterung, die zu einer `NegativeArraySizeException` führt. Möchtet man dies verhindern, dann sollte man dabei Punkt 1-3 *Resource limit checks should not suffer from integer overflow* beachten. Eine Abfrage mit z.B. `stack.length*2` < *max* wird im Fall eines Überlaufs der Berechnung `stack.length*2` nicht greifen. Stattdessen muss man die Abfrage wie folgt formulieren: `stack.length` < *max*`/2`

Betrachtet man jetzt die Methode `pop` aus dem Listing 4, dann fällt auf, dass die Referenzen auf nicht mehr benötigte Objekte nicht freigegeben werden, d.h. der Garbage Collector kann diese nicht freigeben. Außerdem wird das Feld, in dem die Referenzen gehalten werden nicht wieder reduziert, falls nur noch wenig Elemente im Stack sind. Im Listing 6 werden beide Aspekte berücksichtigt, so dass nicht unnötigerweise Speicher blockiert und somit die Richtlinie 1-2 *Release resources in all cases* erfüllt wird.

Listing 6: Verbesserte `pop`-Methode der Stack Implementierung

```
1     // ...
2     public Object pop() {
3        Object element = stack[--top];
4        stack[top] = null;
5        if (top < stack.length / 3)
6           stack = Arrays.copyOf(stack, stack.length / 2);
7        return element;
8     }
9     // ...
```

Das Prinzip der Geheimhaltung der Implementierung, das in 0-6 *Encapsulate* adressiert wird, und die Definition von Schnittstellen muss einem Programmieranfänger unbedingt vermittelt werden. Da wir in der Grundausbildung sowohl die Implementierung des Stacks mit Hilfe des Feldes, als auch die Realisierung durch eine einfach verkettete Liste betrachten, kann man daran sehr gut die Kapselung bzw. das Geheimnisprinzip erklären. Dem Verwender des Stacks bleibt die Implementierung verborgen und der Austausch der Implementierung ist ohne weiteres möglich, solange die öffentliche Schnittstelle der Klasse erhalten bleibt.

3.4 Unsicherer Safe

Ziel der robusten Programmierung ist es zu verhindern, dass eigene Klassen missbraucht werden oder deren Funktionalität ausgehebelt wird. Dazu betrachten wir zunächst die Sichtbarkeiten an der Klasse `Safe`, da das Prinzip eines Safes allen Studierenden vertraut ist. In Listing 7 ist eine erste Variante der Klasse `Safe` gegeben.

Listing 7: Klasse `Safe`

```
1  public class Safe {
2     private String secret = null;
3     private final String password;
4
5     public Safe(String password) {
6        this.password = password;
7     }
8
9     public String getSecret(String password) {
10       return isPasswordValid(password) ? secret : null;
11    }
12
13    public void setSecret(String password, String secret) {
14       if (isPasswordValid(password))
15          this.secret = secret;
16    }
17
18    public boolean isPasswordValid(String password) {
19       return this.password.equals(password);
20    }
21 }
```

Hier sieht man, dass schon einige sicherheitsrelevanten Forderungen umgesetzt sind. Die Richtlinie 4-1 *Limit the accessibility of classes, interfaces, methods, and fields* fordert, dass wir unsere Klassen und ihre Attribute und Methoden möglichst gut nach außen hin schützen sollen. Dies wurde hier durch die Verwendung der Sichtbarkeit `private` für die Attribute umgesetzt. Der Zugriff auf unser `secret` ist nur über die Methoden `getSecret` und `setSecret` möglich, die beide zuerst überprüfen, ob das Passwort korrekt ist.

Als nächstes betrachten wir die Richtlinie 0-2: *Avoid duplication*. Doppelter Programmcode bedeutet immer, dass man evtl. bei Anpassungen oder bei der Behebung von Fehlern im Code, die

Änderung in duplizierten Teilen vergisst. In der Klasse `Safe` wurde z.B. die Überprüfung des Passworts ausgelagert, so dass man hier leicht eine Anpassung vornehmen kann, falls man z.B. den Fall betrachten möchte, dass als Parameter für `password` auch `null` übergeben werden kann (siehe Listing 8).

Listing 8: Methode `isPasswordValid` der Klasse `Safe`

```
1  public class Safe {
2     // ...
3     public boolean isPasswordValid(String password) {
4        if (this.password == null) {
5           return password == null;
6        } else {
7           return this.password.equals(password);
8        }
9     }
10 }
```

Schon initial muss beim Design der Schnittstellen auf die Informationssicherheit geachtet werden: 0-1 *Design APIs to avoid security concerns*. Meistens ist es schwierig diese nachträglich einzubauen. Eine Schnittstelle, die einmal öffentlich war, darf nicht so einfach auf privat geändert werden, da diese ja evtl. schon von einem anderen Programmierer verwendet wird. Schauen wir uns hierzu unsere Klasse `Safe` noch genauer an. Da `Safe` nicht auf final gesetzt wurde, kann diese Klasse geerbt und ihre Methoden können überschrieben werden. Dies kann evtl. ein Sicherheitsrisiko darstellen, wie die Klasse `NoSafe` im Listing 9 demonstriert.

Listing 9: Klasse `NoSafe`

```
1  public class NoSafe extends Safe {
2     public NoSafe(String password) {
3        super(password);
4     }
5
6     @Override
7     public boolean isPasswordValid(String password) {
8        return true;
9     }
10 }
```

Die Methoden `setSecret` und `getSecret` in der Klasse `Safe` stützen sich auf die Methode `isPasswordValid` ab (siehe Zeile 13 und 9 im Listing 7). Die Methode `isPasswordValid` der Klasse `Safe` ist allerdings weder `private` noch `final` und kann somit überschrieben werden. Die Klasse `NoSafe` kann durch das Überschreiben der Methode `isPasswordValid` den Schutz des *Geheimnisses* aushebeln. Die Richtlinie 0-1 steht damit im engen Zusammenhang mit dem Punkt 4-5 *Limit the extensibility of classes and methods*.

Betrachten wir jetzt eine weitere Klasse, die die Funktionalität der Klasse `Safe` erbt (siehe Listing 10). Die Klasse `WasSafe` erlaubt im Konstruktor zusätzlich die Eingabe eines initialen Geheimnisses, das auch gespeichert wird.

Wie verändert sich das Verhalten der Klasse, wenn die Oberklasse die Methode `isPasswordValid` abändert und z.B. immer `true` zurück liefert? Dann ist das initiale Geheimnis nicht mehr geschützt (siehe 4-6 *Understand how a superclass can affect subclass behavior*).

Weiterhin sicherheitskritisch ist die Verwendung der überschreibbaren Methode `setSecret` im Konstruktor. Was passiert, wenn eine erbende Klasse diese Methode überschreibt. In diesem Fall könnten – wie schon beim Überschreiben der Methode `isPasswordValid` – ungewünscht

Nebeneffekte auftreten. Prinzipiell gilt, dass die vollständige Initialisierung eines Objekts so gut wie möglich geschützt wird und darum z.B. keine überschreibbaren Methoden im Konstruktor verwendet werden (siehe 7-4 *Prevent constructors from calling methods that can be overridden*).

Listing 10: Klasse `WasSafe`

```
1  public class WasSafe extends Safe {
2     private final String initialSecret;
3
4     public WasSafe(String password, String initialSecret) {
5        super(password);
6        setSecret(password, initialSecret);
7        this.initialSecret = initialSecret;
8     }
9
10    public String getInitialSecret(String password) {
11       return isPasswordValid(password) ? initialSecret : null;
12    }
13 }
```

Noch stärker kann der Konstruktor geschützt werden, indem der Konstruktur auf `private` gesetzt wird und stattdessen eine statische Fabrikmethode angeboten wird. Dadurch kann man z.B. auch die Parameterwerte, die beim Konstruktoraufruf verwendet werden, vor der Erstellung der Objekte überprüfen oder z.B. die Anzahl der zu erstellenden Objekte einschränken (siehe 7-1 *Avoid exposing constructors of sensitive classes* und Listing 11).

Listing 11: Klasse `Safe`

```
1  public class Safe {
2     // ...
3     private Safe(String password) {
4        this.password = password;
5     }
6
7     public static final Safe createSafe(String password)  {
8        // ... check password
9        return new Safe(password);
10    }
11    // ...
12 }
```

Dass ein Passwort, wie es z.B. in der Klasse `Safe` verwendet wird, eine vertrauenswürdige Information ist, ist jedem Programmieranfänger klar und kann in diesem Abschnitt als Aufhänger verwendet werden, um die Studierenden für den Umgang mit vertrauenswürdiger Information zu sensibilisieren.

Um die Zugriffe auf die Objekte der Klasse `Safe` zu protokollieren hat ein Programmierer in die Methoden `getSecret` und `setSecret` den folgenden Programmcode eingefügt.

```
public String getSecret(String password) {
   System.out.println("getSecret(" + password + ")")
   ...

public void setSecret(String password, String secret) {
   System.out.println("setSecret(" + password + ", " + secret + ")")
   ...
```

Dieses Vorgehen widerspricht der Richtlinie 2-2: *Do not log highly sensitive information*. Auf einmal ist es möglich Passwörter und Geheimnisse auf der Konsole im Klartext zu lesen. Genauso gefährlich ist es, wenn man beim Loggen von Programmabläufen vertrauenswürdige

Informationen in Dateien ablegt. Dies können wie in unserem Beispiel Passwörter sein, aber z.B. auch Informationen darüber, welcher Nutzer wann welche Aktionen ausgeführt hat.

Der Punkt 2-1 *Purge sensitive information from exceptions* kann den Studierenden anhand eines kleinen Beispiels demonstriert werden, in dem auf der Konsole nach einem Passwort gefragt wird. Eigentlich ist eine PIN gefordert und daher wird die Eingabe in einen `Integer`-Wert umgewandelt (siehe Listing 12). Führt man das Programm aus, dann sieht man zwar die Eingabe in der Shell nicht, aber das Passwort taucht in der Nachricht zur Ausnahme auf, falls es sich bei der Eingabe nicht um einen `Integer`-Wert handelt. Auch beim Arbeiten mit Dateien zeigen die Exception Pfadinformationen an, die evtl. auf die Struktur des Dateisystems schließen lässt und somit potenziellen Angreifern Hilfestellung bieten.

Listing 12: Passwort in Exception

```
1 public static void main(String[] args) {
2     System.out.print("Password?␣");
3     char[] pinInput = System.console().readPassword();
4     int pin = Integer.parseInt(String.valueOf(pinInput));
5 }
```

Vergleichen wir das letzte Beispiel, bei dem das Passwort in einem `char`-Array abgelegt wird, mit unserem `Safe`, bei dem wir das Passwort in einen String speichern: Die Richtlinie 2-3 *Consider purging highly sensitive from memory after use* gibt uns hier einen Hinweis, welche Lösung die bessere ist. Daten, die im Arbeitsspeicher eines Programms liegen, können unter bestimmten Umständen eingesehen werden. Beim Debuggen bekommen die Studierenden eine Idee, wie das möglich ist. Daher sollten besonders vertrauenswürdige Informationen möglichst schnell wieder aus dem Speicher verschwinden oder dort nur verschlüsselt auftauchen. In Java haben wir aber keine direkte Kontrolle, wie Objekte im Speicher gelöscht werden. Man kann zwar die Java Garbage Collection aufrufen, weiß aber nicht, was im Hintergrund genau passiert. Bei der Verwendung eines `String` zum Speichern eines Passworts haben wir folglich nicht die letzte Kontrolle über das Löschen des Passworts aus dem Speicher. Verwenden wir dagegen ein `char`-Feld, dann können wir vor der Freigabe den Inhalt des Feldes überschreiben.

3.5 Komplexe Zahlen

In diesem Abschnitt betrachten wir Probleme, die bei Zustandsänderungen von Objekten auftreten anhand der Klassen: `ComplexNumber` und `HarmonicOscillator`. In unserem Fall wird die komplexe Zahl selbst in einem Real- und einem Imaginärteil gespeichert – nach extern besteht aber auch die Möglichkeit die Polarform der komplexen Zahl zu verwenden. Als Anwendung der komplexen Zahl dient dabei die harmonische Schwingung – was besonders gut zu unserem elektrotechnischen Studiengang passt. Für dieses Dokument haben wir beide Klassen stark gekürzt.

Die Klasse `ComplexNumber` in Listing 13 zeigt eine mögliche Implementierung: Neben der Speicherung des Real- und Imaginärteils bietet sie die typischen Operationen für komplexe Zahlen an. Im konstanten Klassenattribut `ONE` wird das neutrale Element bezüglich der Multiplikation gespeichert.

Listing 13: Klasse ComplexNumber (Variante 1)

```
1  public class ComplexNumber {
2
3     public static final ComplexNumber ONE = new ComplexNumber(1, 0);
4
5     private double real; private double imaginary;
6
7     public ComplexNumber(double real, double imaginary) {
8        this.real = real; this.imaginary = imaginary;
9     }
10
11    public ComplexNumber(double distance, double phi, boolean inRadian) {
12       if (inRadian) {
13          real = distance * Math.cos(phi);
14          imaginary = distance * Math.sin(phi);
15       } else {
16          real = distance * Math.cos(Math.toRadians(phi));
17          imaginary = distance * Math.sin(Math.toRadians(phi));
18       }
19    }
20
21    public ComplexNumber mult(ComplexNumber c) {
22       double tmpReal = real * c.real - imaginary * c.imaginary;
23       imaginary = real * c.imaginary + imaginary * c.real;
24       real = tmpReal;
25       return this;
26    }
27
28    // .. more Operations ...
29
30    public double getDistance() { return Math.hypot(real, imaginary); }
31
32    public double getPhi(boolean inRadian) {
33       double radians = Math.atan2(imaginary, real);
34       return inRadian ? radians : Math.toDegrees(radians);
35    }
36
37    @Override
38    public String toString() {
39       return "(" + real + ") + (" + imaginary + ")i";
40    }
41 }
```

Die Richtlinie 6-9 *Make public static fields final* ist erfüllt. Damit ein öffentliches Klassenattribut nicht als globale Variable missbraucht wird, muss dieses mit `final` als Konstant gekennzeichnet werden. Das Klassenattribut `ONE` kann nicht von extern z.B. durch den Aufruf `ComplexNumber.ONE = new ComplexNumber(4,5)` geändert werden.

Schauen wir uns jetzt die folgende Verwendung der Klasse an. Es werden drei Multiplikationen mit dem neutralen Element `ONE` durchgeführt. Am Ende der Zeile steht im Kommentar die Ausgabe, die durch das Programm bei Ausführung erzeugt wird.

```
ComplexNumber z = new ComplexNumber(3, 2);
System.out.println("      z = " + z);                          //       z = (3.0) + (2.0)i
System.out.println("    ONE = " + ComplexNumber.ONE + "\n");   //     ONE = (1.0) + (0.0)i

System.out.println("z * ONE = " + z.mult(ComplexNumber.ONE)); // z * ONE = (3.0) + (2.0)i
System.out.println("ONE * z = " + ComplexNumber.ONE.mult(z)); // ONE * z = (3.0) + (2.0)i

System.out.println("      z = " + z);                          //       z = (3.0) + (2.0)i
System.out.println("    ONE = " + ComplexNumber.ONE);          //     ONE = (3.0) + (2.0)i
```

Das Klassenattribut `ONE` wird durch die Ausführung der Anweisungen verändert. Die Richtlinie 6-10 *Ensure public static final field values are constant* wurde nicht beachtet. Die Methode `mult` kann die Werte des Objekts, auf dem diese Methode aufgerufen wird, verändern. In die-

sem Fall würde es nur helfen das Klassenattribut auf `private` zu setzen – aber Achtung es darf dann nicht einfach eine Methode `getOne()` eingeführt werden, die das Klassenattribut `ONE` zurückgibt. Das führt auf ein vergleichbares Problem: Dieses Sicherheitsrisiko ist in der Richtlinie 6-11 *Do not expose mutable statics* adressiert und bezieht sich nicht nur auf Konstanten, sondern allgemein auf Klassenattribute, die änderbar sind. Dieser letzte Punkt gilt insbesondere für Sammlungen von Objekten sogenannte Collections, wie z.B. Felder aber auch die oben eingeführte Stack-Implementierung. Wird der Zugriff auf eine Sammlung durch ein öffentliches Attribut oder eine entsprechende Getter-Methode ermöglicht, dann kann jeder die Einfüge- und Löschoperationen, die es normalerweise auf solche Sammlungen gibt, verwenden (siehe 6-12 *Do not expose modifiable collections*). 6-2 *Create copies of mutable output values* verallgemeinert diesen Sicherheitshinweis noch dahingehend, dass es notwendig ist bei Rückgabe von Objekten eine Kopie anzufertigen, falls diese in der Klasse weiterhin verwendet werden.

Ist sich der Nutzer der Problematik bewusst, dann könnte er zumindest vor jedem Aufruf der Methode `mult` das Objekt kopieren. Falls man nicht auf die Veränderbarkeit verzichten möchte oder kann, dann sollte man wenigstens eine Kopiermöglichkeit für die Objekte der Klasse zur Verfügung stellen (Punkt 6-4 *Support copy functionality for a mutable class*, siehe Listing 14).

Listing 14: Klasse ComplexNumber (Variante 2)

```
1  public class ComplexNumber {
2     // ...
3     public static ComplexNumber copyInstance(ComplexNumber c) {
4        return new ComplexNumber(c.real, c.imaginary);
5     }
6     // ...
7  }
```

Berücksichtigt man den Punkt 6-1 *Prefer immutability for value types* und deklariert die Klassen von Anfang an als Unveränderliche (siehe Listing 15), dann vermeidet man viele der oben genannten Probleme. Evtl. muss man sich dann über die Effizienz Gedanken machen, da die Wiederverwendung eines Objekts wie z.B. in der Methode `mult` nicht mehr möglich ist.

Listing 15: Klasse ComplexNumber (Variante 3)

```
1  public class ComplexNumber {
2     // ...
3     private final double real;
4     private final double imaginary;
5     // ...
6     public ComplexNumber mult(ComplexNumber c) {
7        return new ComplexNumber(real * c.real - imaginary * c.imaginary,
8               real * c.imaginary + imaginary * c.real);
9     }
10    // ...
11 }
```

Schauen wir uns jetzt zusätzlich die Klasse `HarmonicOscillator` im Listing 16 an, die sich auf die Implementierung der Klasse `ComplexNumber` abstützt.

Die Klasse repräsentiert eine harmonische Schwingung, die sich aus dem Phasor bzw. einer komplexen Amplitude und einer Frequenz zusammensetzt. Die Operation `getVectorAtTime` liefert den Zustand der Schwingung zu einem Zeitpunkt `time` in Form einer komplexen Zahl.

Listing 16: Klasse HarmonicOscillator

```
1  public class HarmonicOscillator {
2
3     private final ComplexNumber phasor;
4     private final double frequence;
5
6     public HarmonicOscillator(ComplexNumber phasor, double frequence) {
7        // this.phasor = phasor;
8        this.phasor = ComplexNumber.copyInstance(phasor);
9        this.frequence = frequence;
10    }
11
12    public HarmonicOscillator(double amplitude, double phase, double frequence) {
13       phasor = new ComplexNumber(amplitude, phase, true);
14       this.frequence = frequence;
15    }
16
17    public ComplexNumber getVectorAtTime(double time) {
18       return phasor.mult(new ComplexNumber(1.0, frequence * time, true));
19    }
20
21    @Override
22    public String toString() {
23       return phasor.getDistance() + "e^i(" + frequence + "t"
24              + (phasor.getPhi(true) >= 0 ? "+":"") + Math.abs(phasor.getPhi(true)) + ")";
25    }
26 }
```

Nachdem ein Objekt unserer Klasse `ComplexNumber` jetzt nicht mehr geändert werden kann, erscheint es ungefährlich, im Konstruktor die Referenz auf den Parameter `phasor` direkt in das gleichnamige Attribut zu übernehmen, vgl. Zeile 8 in Listing 16. Warum trotzdem eine Kopie erzeugt wird, lässt sich einfach an folgendem Gedanken erklären: Was würde passieren, wenn ein Programmierer unsere Klasse `HarmonicOscillator` mit einem Objekt der Klasse `ComplexNumberExtension` in Listing 17 verwendet, ohne dass eine vertrauenswürdige Kopie erzeugt wird.

Listing 17: Klasse ComplexNumberExtension

```
1  public class ComplexNumberExtension extends ComplexNumber {
2
3     // ... see ComplexNumber Constructors
4     @Override
5     public ComplexNumber mult(ComplexNumber c) {
6        return new ComplexNumberExtension(1, 1);
7     }
8  }
```

Schauen wir uns jetzt die folgende Verwendung der Klasse an. Wir erzeugen zwei harmonische Schwingungen mit den gleichen Parametern – einmal mit einem Objekt vom Typ `ComplexNumber` und einmal mit einem Objekt vom Typ `ComplexNumberExtension`. Jetzt lassen wir uns für beide harmonischen Schwingungen den aktuellen Zustand der Schwingung zum gleichen Zeitpunkt `Math.PI/4.0` ausgeben und erhalten unterschiedliche Ergebnisse.

```
HarmonicOscillator h1 = new HarmonicOscillator(new ComplexNumber(1, 0), 1.0);
HarmonicOscillator h2 = new HarmonicOscillator(new ComplexNumberExtension(1, 0), 1.0);

System.out.println(h1.getVectorAtTime(Math.PI/4.0)); // (0.707...) + (0.707...)i
System.out.println(h2.getVectorAtTime(Math.PI/4.0)); // (1.0) + (1.0)i
```

Das liegt daran, dass sich die Berechnung in der Methode `getVectorAtTime` auf die Methode `mult` der Klasse `ComplexNumber` in Fall von `h1` und ohne vertrauenswürdige Kopie auf die Methode `mult` der Klasse `ComplexNumberExtension` in Fall von `h2` abstützt. Stellt man

sich vor, dass es sich bei der Methode `getVectorAtTime` z.B. um eine Berechtigungsfreigabe handelt, die auf einer Methode basiert, die eben genauso wie die Methode `mult` manipuliert werden kann, dann erkennt man die Gefahr, die davon ausgeht. Wie schon in Abschnitt 3.4 gezeigt, muss man dafür sorgen, dass die Funktionalität einer Klasse durch die Vererbung nicht ausgehebelt wird. Daher fordert die Richtlinie 6-3 *Create safe copies of mutable and subclassable input values* für Objekte von Klassen, die erweitert werden können, auch sichere Kopien anzufertigen.

4 Zusammenfassung und Ausblick

Wir zeigen, dass gut ein Drittel der 76 Richtlinien aus den SCG von Oracle schon zu Beginn der Programmierausbildung sinnvoll adressiert werden kann, vgl. Abbildung 1. Anhand der hier vorgestellten Beispiele ist es relativ einfach, diese in die Ausbildung zu integrieren.

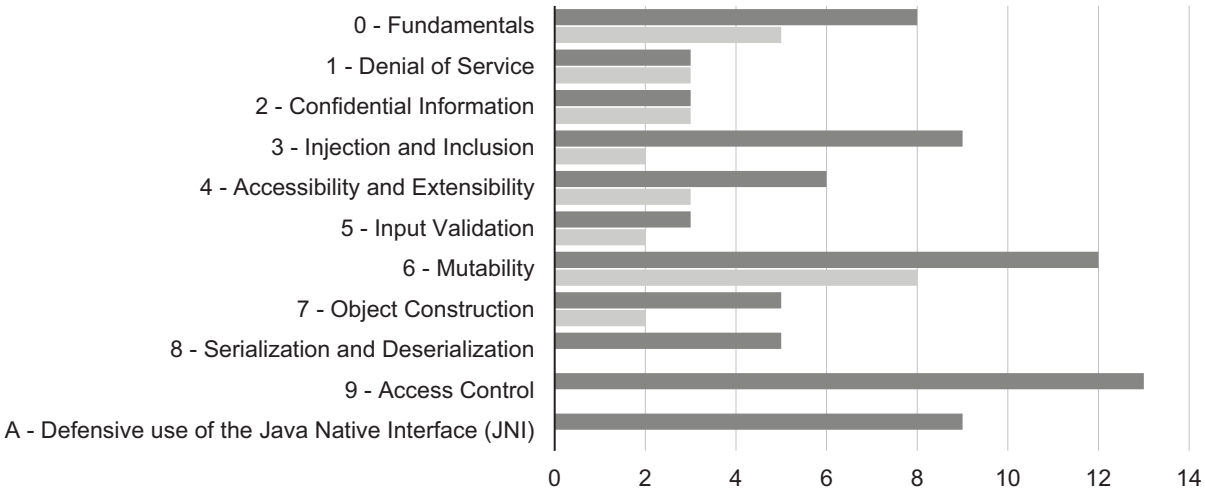

Abb. 1: Anzahl der Richtlinien: gesamt (dunkelgrau) und für Programmieranfänger nutzbare (hellgrau)

Der Schwerpunkt liegt dabei auf dem Schutzziel der Verfügbarkeit bzw. der robusten Programmierung. Die Schutzziele der Integrität und Vertraulichkeit hingegen können meist nicht tiefer behandelt werden, da diese in der Regel den Einsatz des *Java Security Managers* beinhalten. Dieser ist normalerweise nicht Teil der Grundlagenausbildung. Darüber hinaus können einige Richtlinien aufgrund ihrer Komplexität noch nicht angesprochen werden, da diese Kenntnisse zu fortgeschrittenen Themen, wie z.B. dem `ClassLoader`, den `Collections`, dem Umgang mit Dateien oder Datenbanken, erfordern.

Möchte man den Security-Gedanken im Hinblick auf die Verfügbarkeit weiter ausbauen, dann sollten Ausnahmen und ihre Behandlung so früh wie möglich in der Grundausbildung besprochen werden. Gerade bei der Überprüfung von nicht vertrauenswürdigen Eingaben spielen diese eine wichtige Rolle. Geht man noch einen Schritt weiter, sollte man über die Einführung der Pattern- und Matcher-Klasse nachdenken. Damit wäre zusätzlich die Validierung von Strings möglich, die dann z.B. eine intensivere Beschäftigung mit der Richtlinie 5-1 *Validate inputs* ermöglicht. Zusätzlich könnten Richtlinien aus dem Bereich *Injection and Inclusion* abgedeckt werden. Sofern diese neuen Aspekte jedoch zeitneutral behandelt werden sollen, stellt sich die Frage, ob man auf andere Kompetenzen, wie z.B. Sortieralgorithmen verzichten kann.

Die Integration sicherheitsrelevanter Aspekte in der Java-Grundausbildung ist für weiterführende Lehrveranstaltungen auf jeden Fall sinnvoll, da die Studierenden für das Thema Security sensibilisiert werden. Insbesondere durch die konkreten Beispiele bekommen sie ein Gespür für das Thema. Sie lernen den Umgang mit entsprechender Literatur und können bei Bedarf auf diese zurückgreifen.

Literatur

[ACM16] ACM: Computer Engineering Curricula 2016 - Curriculum Guidelines for Undergraduate Degree Programs in Computer Engineering. Tech. Rep., Association for Computing Machinery (ACM) IEEE Computer Society (2016).

[Bloc08] J. Bloch: Effective Java. Java Series, Pearson Education, 2. Aufl. (2008).

[CERT17] CERT: SEI CERT Oracle Coding Standard for Java, Software Engineering Institute, Carnegie Mellon University (2017), online (abgerufen 06.06.2017): https://www.securecoding.cert.org/confluence/display/java/SEI+CERT+Oracle+Coding+Standard+for+Java.

[Hoar81] C. A. R. Hoare: The Emperor's Old Clothes. In: *Commun. ACM*, 24, 2 (1981), 75–83.

[Info16] G. für Informatik: Empfehlungen für Bachelor- und Masterprogramme im Studienfach Informatik an Hochschulen. Tech. Rep., Gesellschaft für Informatik e.V (2016).

[KrHa14] G. Krüger, H. Hanse: Java Programmierung - Das Handbuch zu Java 8. Verlag O'Reilly, 8. Aufl. (2014).

[Mart08] R. C. Martin: Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1 Aufl. (2008).

[Orac15] Oracle: Secure Coding Guidelines for Java SE (2015), online (abgerufen 06.06.2017): http://www.oracle.com/technetwork/java/seccodeguide-139067.html.

[SeWa11] R. Sedgewick, K. Wayne: Einführung in die Programmierung mit Java. Pearson Verlag (2011).

[SFBHB+13] M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, P. Sommerlad: Security Patterns: Integrating Security and Systems Engineering. Wiley Software Patterns Series, Wiley (2013).

[Somm12] I. Sommerville: Software Engineering. Pearson Studium, 9. Aufl. (2012).

[TsCM05] K. Tsipenyuk, B. Chess, G. McGraw: Seven pernicious kingdoms: A taxonomy of software security errors. In: *IEEE Security & Privacy*, 3, 6 (2005), 81–84.

[Ulle14] C. Ullenboom: Java ist auch eine Insel: Programmieren lernen mit dem Standardwerk für Java-Entwickler, aktuell zu Java 8. Galileo Computing (2014).

[ViMc11] J. Viega, G. McGraw: Building Secure Software: How to Avoid Security Problems the Right Way. Addison-Wesley Professional, 1. Aufl. (2011).

[WYYB14] K. A. Williams, X. Yuan, H. Yu, K. Bryant: Teaching Secure Coding for Beginning Programmers. In: *J. Comput. Sci. Coll.*, 29, 5 (2014), 91–99.