Hochschule Kempten
University of Applied Sciences

Seminar

Refactoring in der Softwareentwicklung
Sommersemester 2024
bei Prof. Dr. Georg Hagel

five lines of code (Christian Clausen)

Thema: Fogle der Struktur im Code

Name, Vorname	Heiserer Valentin
Matrikelnummer	453990
Studiengang	Bachelor Informatik
Semester	Sommersemester 2024

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

In dieser Arbeit soll genauer untersucht werden, was Refactoring tatsächlich an Code verändert, welche Möglichkeiten es gibt ein bestimmtes Verhalten überhaupt darzustellen. Als Grundlage dieser Arbeit gilt das 11. Kapitel aus dem Buch *„five lines of code“* von Christian Clausen [2]. Dieses wurde im Rahmen des Seminars „Refactoring“, bei Professor Dr. Georg Hagel, untersucht und für diese Arbeit aufgearbeitet. Das Kapitel des Buchs kann zwar eigenständig gelesen werden, aber ein grundlegendes Verständnis von Refactoring ist trotzdem erforderlich.
Außerdem wird erläutert, in welchen Situationen auf ein Refactoring verzichtet werden sollte und welche Gründe es dafür gibt.
Anschließend sollen einige Maßnahmen vorgestellt werden, mit denen Sicherheit erlangt werden kann, dass Code ordnungsgemäß funktioniert. Gerade nach einem umfassenden Refactoring spielt dies eine große Rolle. Abschließend werden einige Fälle vorgestellt, in denen Refactoring aus verschiedenen Gründen häufig nicht durchgeführt wird und es wird gezeigt wieso dies der Fall sein sollte.

2 Strukturen in der Softwareentwicklung

Bevor man sich mit den Strukturen in der Softwareentwicklung auseinandersetzen kann, ist es wichtig, sich erneut vor Augen zu führen, was Software eigentlich ist.
"*Software modelliert einen Teil der Welt. Die Welt - und unser Verständnis davon - entwickelt sich, und unsere Software muss sich entwickeln, um ein akkurates Modell zu sein.*"[2, S. 311]
Dies heißt außerdem das Software nie fertig ist, da sie sich immer an die Ständig ändernde Welt anpassen.
Code bildet also verschiedenste Gegebenheiten aus der Realität ab. Darunter zählen Informationen, Zusammenhänge und ganze Abläufe. Zusammen ergibt sich dadurch eine Struktur, ein wiedererkennbares Muster, welches sich sowohl in der echten Welt als auch in der Software finden lässt. [2, S. 311]

Verschiedene Arten von Strukturen

Es gibt verschiedene Bereiche in der Softwareentwicklung, in denen Struktur eine Rolle spielt. Es ist möglich diese an zwei Achsen einzuteilen. Zum einen gibt es einige Faktoren welche den Menschen, also die Softwareentwickler direkt betreffen, oder aber den tatsächliche Code. Auf der zweiten Achse wählt Clausen den Wirkungsbereich als Einteilung [2, S. 311]. Die folgende Tabelle zeigt das Ergebnis der Einteilung.

	Teamintern	Teamübergreifend
Code	Daten und Funktionen	externe APIs
Menschen	Hierarchie, Prozesse	Verhalten, Domänenexperten

Tabelle 2.1: Strukturkategorien [2] (korrigierte Form)

Melvin E. Conway stellt bereits 1968 Beobachtungen an, dass es eine gewisse Symmetrie zwischen der Arbeitsweise von Entwicklerteams und den Zusammenhängen der tatsächlichen Systeme gibt. [3]

Auch das Nutzerverhalten kann bestimmte Strukturen vorgeben, da diese ein immer gleiches Verhalten erwarten. Auch wenn einige Abläufe optimiert oder umstrukturiert werden können, ist es nicht immer sinnvoll dies zu tun, da dann ggf. Nutzer neu geschult werden müssen.[2, S. 312]

3 Wie kann Verhalten im Code abgebildet werden?

Im nächsten Teil wird beleuchtet, wie Verhalten im Code abgebildet werden kann. Dabei gibt es grundlegend drei verschiedene Arten. Diese werden im Folgenden anhand eines einfachen Beispiels erläutert. Des Weiteren soll auf die Erzeugung von Endlosschleifen eingegangen werden, wie Clausen feststellt, stellt dies einen Sonderfall dar. Viele Refactorings transformieren Code zwischen diesen Darstellungsformen [2, S. 313].

Beispiel-Verhalten

Es soll bis zu einer bestimmten ganzen Zahl abwechselnd „gerade“ und „ungerade“ in der Konsole ausgegeben werden. „0“ wird hierbei als gerade angesehen.
Dieses Verhalten ist am von Clausen genutzten Beispiel (FizzBuzz) nachempfunden und so weit wie möglich vereinfacht, um weiterhin alle nötigen Besonderheiten zu veranschaulichen.[2]

3.1 Verhalten im Kontrollfluss

Die erste und wohl einfachste Möglichkeit, Verhalten im Code abzubilden, ist der Kontrollfluss. Dieser zeichnet sich durch die Verwendung von Kontrolloperatoren, Methodenaufrufen und der Zeilenabfolge aus. [2, S. 313 f.] In folgender Abbildung 3.1 werden dafür jeweils einfache Beispiele gezeigt.

```
const i = 0;
while (i < 5) {
    foo(i);
    i++;
}
```

(a) Kontrolloperatoren

```
function loop(i: number) {
    if (i < 5) {
        foo(i);
        loop(i + 1);
    }
}
```

(b) Methodenaufrufe

```
foo(0);
foo(1);
foo(2);
foo(3);
foo(4);
```

(c) Zeilenabfolge

Abbildung 3.1: Beispiele für Code im Kontrollfluss [2]

Der Unterschied dieser Unterkategorien wird bei der Betrachtung des Aufrufs „foo(i)“ und dessen Werthereingabe deutlich. Bei 3.1a wird mithilfe des „while“ Operators die Funktion aufgerufen und die Eingabe erhöht.
Das mittlere Beispiel zeigt die Verwendung einer rekursiven Methode, um das Verhalten darzustellen. Der Eingabeparameter dient hier als Wert für den Funktionsaufruf.

Das letzte Beispiel 3.1c zeigt das gleiche Verhalten durch einfache Aufrufe. Hier wird die Funktion mit Wert im Klartext aufgerufen.

Eigenes Beispiel

Der folgende Codeausschnitt 3.2 zeigt das vorher beschriebene Beispiel im Kontrollfluss.

```
function istGerade(n: number) {
    for(let i = 0; i <= n; i++) {
        if(i % 2 == 0) {
            console.log("Gerade");
        } else {
            console.log("Ungerade");
        }
    }
}
```

Abbildung 3.2: Beispiel im Kontrollfluss

Um die Unterschiede der verschiedenen Darstellungsformen zu erkennen, ist es sinnvoll die Aufrufe von *„console.log()“* zu betrachten. Diese stellen bei unserem Beispiel die tatsächlich durchzuführende Aktion dar. In Abbildung 3.2 lässt sich erkennen, dass diese Aufrufe durch die Kontrolloperatoren *for* und *if* gesteuert werden. Dies stellt ein klassisches Beispiel für das Verhalten im Kontrollfluss dar.

Endlosschleife

Im Kontrollfluss gibt es mehrere Möglichkeiten, eine Endlosschleife zu erzeugen. Da sie am weitesten verbreitet sind, sollten sie unter allen Programmierenden bekannt sein.

```
for (;;){}

while (true) {}
```

(a) Endlosschleife mit Kontrolloperatoren

```
function loop() {
    loop();
}
```

(b) Endlosschleife mit rekursiver Methode

Abbildung 3.3: Endlosschleifen im Kontrollfluss [2, S. 314]

Die einzige Besonderheit stellt das *„for(;;)“* dar, da es durch eine leere Bedingung niemals terminiert.

Vor und Nachteile

Da das Programmieren im Kontrollfluss schnell und einfach funktioniert, eignet sich diese Art gut, um neues Verhalten initial abzubilden. Umfassende Änderungen sind sofort möglich, da zum Beispiel einfach die Reihenfolge der Aufrufe geändert werden kann und somit sofort

ein anderes Verhalten entsteht. Dieser Punkt ist aber nicht zwingend als Vorteil zu sehen, da wir in der Softwareentwicklung Stabilität und gute Wartbarkeit bevorzugen [2, S. 313 f.].

3.2 Verhalten in der Struktur der Daten

Die Zweite Möglichkeit Verhalten im Code abzubilden, sind Datenstrukturen. Diese zeichnen sich durch die Verwendung von objektorientierter Programmierung aus. Also durch Klassen, Interfaces, Vererbung, etc. [2, S. 315 f.]
Eines der bekanntesten Beispiele für die Verwendung von Datenstrukturen ist die *„binäre Suche“*. Diese setzt einen sortierten Binärbaum voraus, um zu funktionieren. Suchen wir nun nach einem Wert, wird dieser mit dem der Wurzel verglichen und je nach Ergebnis rekursiv in den linken oder rechten Teilbaum abgestiegen. Das Verhalten wird also maßgeblich durch die Struktur des Baumes bestimmt [2, S. 315 f.].

Eigenes Beispiel

Das folgende Beispiel 3.4 zeigt das vorher beschriebene Beispiel in einer Datenstruktur.

```
interface Zahl{
    istGerade(): void;
}

class GeradeZahl implements Zahl{
    constructor(private count: number) {}
    istGerade() {
        console.log("Gerade");
        if(this.count != 0) {
            new UngeradeZahl(this.count - 1).istGerade();
        }
    }
}

class UngeradeZahl implements Zahl{
    constructor(private count: number) {}
    istGerade() {
        console.log("Ungerade");
        if(this.count != 0) {
            new GeradeZahl(this.count - 1).istGerade();
        }
    }
}
```

Abbildung 3.4: Beispiel in einer Datenstruktur

Um das Verhalten abzubilden, wird auf den Ansatz einer einfach verketteten Liste gesetzt. Es wird ein Interface *„Zahl“* verwendet, welches die Methode *„istGerade()“* vorschreibt. Jede der beiden Klassen *„GeradeZahl“* und *„UngeradeZahl“* implementiert dieses Interface und

ruft die Methode der jeweils anderen Klasse auf, solange der Zähler nicht bei 0 angelangt ist.

Endlosschleife

```
1  class Rec {
2      constructor(public readonly f:(_: Rec) => void) {}
3  }
4
5  function loop() {
6      let helper = (r: Rec) => r.f(r);
7      helper(new Rec(helper));
8  }
```

Abbildung 3.5: Endlossschleife in einer Datenstruktur [2, S. 316]

Der Codeausschnitt 3.5 zeigt eine Endlossschleife in einer Datenstruktur. Dieses Beispiel ist nicht so einfach zu verstehen, da es sich um eine Klasse handelt, die im Konstruktor eine Methode erwartet, die wiederum die selbe Klasse als Parameter erwartet. In der Funktion *„loop()“* wird auf einer Hilfsvariablen eine solche Methode erstellt. Im Inhalt dieser Methode wird auf dem hereingegebenen Objekt die gleiche Methode erneut aufgerufen. Somit ist das Verhalten nun ähnlich zu einem normalen rekursiven Funktionsaufruf, nur dass hier die Klasse als Parameter übergeben wird. Um die Endlosschleife zu starten, muss die eben erstellte Methode mit einem neuen Objekt der Klasse *„Rec“* aufgerufen werden. Da dieser Ansatz nur schwer zu verstehen ist und vor allem schwer zu lesen ist, sollte er nur in Ausnahmefällen verwendet werden [2, S. 316].

Vor und Nachteile

Das Verhalten in Datenstrukturen umzusetzen, hat mehrere positive Effekte. Zum einen erreichen wir durch den objektorientierten Ansatz typensicherheit, was den Code lesbarer und wartbarer macht. Der vorher erwähnte Wunsch nach Stabilität und der Möglichkeit nur kleinere Anpassungen vorzunehmen, wird ebenfalls erfüllt. Auch in der Performance kann es zu einer Verbesserung kommen, da die Datenstrukturen teilweise im Cache abgespeichert werden können. Allerdings ist der Aufwand, um Verhalten in Datenstrukturen abzubilden, höher als im Kontrollfluss [2, S.315 ff.].

3.3 Verhalten in den Daten

Das Verhalten in den tatsächlichen Daten zu kodieren, ist die letzte Möglichkeit, die Clausen in seinem Buch beschreibt. Diese zeichnet sich vor allem durch die Nutzung von Funktionen und Lambdas, die in Arrays gespeichert werden [2, S. 319 f.].

Eigenes Beispiel

```
const daten: (() => void)[] = [
    () => console.log("Gerade"),
    () => console.log("Ungerade")
];

function istGerade(n: number) {
    for(let i = 0; i <= n; i++) {
        daten[i % daten.length]();
    }
}
```

Abbildung 3.6: Beispiel in den Daten selbst

Die Umsetzung des Beispielverhaltens in den Daten selbst ist in Abbildung 3.6 zu sehen. Hier sind die *„console.log()“*-Aufrufe in einem Array gespeichert. Über dieses Array wird in der Funktion *„istGerade()“* iteriert und die Funktionen darin aufgerufen. Die Besonderheit hierbei ist, dass der Modulo-Operator nicht mit einer festen Zahl, sondern mit der Länge des Arrays verwendet wird. Dadurch wird noch deutlicher, dass das Verhalten in den Daten abgebildet wird.

Endlosschleife

```
function loop() {
    let a = [() => { }];
    a[0] = () => a[0]();
    a[0]();
}
```

Abbildung 3.7: Endlossschleife in Daten abgebildet [2, S. 319]

Die Endlosschleife ist in diesem Fall weider einfacher zu verstehen. Es wird ein Array erstellt, welches Funktionen enthält. Anschließend wird auf dem ersten Feld eine Funktion gespeichert, die dann wieder das erste Feld des Arrays aufruft. Auch hier ist es im Endeffekt eine rekursive Funktion, die aber durch das Array abgebildet wird. Um die Schleife zu starten, wird die Funktion im ersten Feld des Arrays aufgerufen [2, S. 319].

Vor und Nachteile

Der einzige Vorteil, den Clausen in seinem Buch nennt ist die in manchen Fällen beste Performance, im Vergleich zu den anderen beiden Darstellungsformen. Jedoch muss man einige negative Aspekt in Kauf nehmen. Durch diese Art der Programmierung erhält man nur eine schlechte Unterstützung durch den Compiler. Es gibt keine Erkennung ob ein gültiges Feld aus dem Array aufgerufen wird oder ob die Funktionen im Array überhaupt die selben Parameter erwarten. Außerdem kann es schnell zu Konmsitentzproblemen kommen, wenn die gespeicherten Daten veränderbar sind. Diese Form der Darstellung sollte also nur in Ausnahmefällen verwendet werden, in denen die Performance eine große Rolle spielt und die anderen Nachteile in Kauf genommen werden können [2, S. 319 f.].

4 Wann ist Refactoring sinnvoll?

Im ersten Moment hört es sich so an, als wäre Refactoring in jedem Fall ein Weg, Qualität, Nutzbarkeit und Wartbarkeit von Software zu verbessern. Jedoch gibt es Situationen, in denen Refactoring keine Verbesserung liefert. Im Gegenteil, es kann sogar zu einer Verschlechterung führen [2, S. 321 f.]. Diese Fälle sollen im folgenden Kapitel beleuchtet werden.

„Durch Refactorings machen wir manche Änderungen leichter und manche schwieriger. Wir führen Refactorings durch, um Änderungen in eine Richtung zu unterstützen, von der wir glauben, dass sich die Software in diese entwickelt. Je mehr Code wir haben, desto sicherer können wir uns über die Richtung und die typische Art der Änderung sein [...]“[2, S. 321]

Gerade bei neu entwickelten Codeteilen ist die Struktur oft noch nicht vollständig ausgereift. Hier ist die oberste Priorität, erst die Korrektheit und Vollständigkeit sicherzustellen. Ist dies nicht der Fall ist es für Entwickler wertvoll, schnell Änderungen vornehmen zu können und verschiedene Ansätze auszuprobieren. Refactoring würde in diesem Fall die Entwicklung verlangsamen und ist daher nicht sinnvoll [2, S. 321].
Trotzdem ist es wichtig durch dieses Vorgehen die Struktur des umliegenden Codes nicht zu vernachlässigen. Bezieht sich die Unsicherheit nur auf einen Teilbereich, sollte dieser vom Rest gekapselt werden, um andere Refactorings nicht zu behindern [2, S. 321].

Auch bei bestehenden Code, oder Code mit genug Sicherheit die Struktur zu festigen, sollte nicht sofort jedes mögliche Refactoring durchgeführt werden, auch wenn es noch so offensichtlich wirkt.

„Versuchen wir, die Entwicklungsrichtung vorherzusagen, können wir der Codebasis langfristig mehr schaden als nutzen. Wie bei den meisten Dingen in unserer Arbeit sollten wir uns nicht auf Vermutungen stützen, sondern auf empirisch erhobene Daten.“[2, S. 322]

Es ist also wichtig Refactoring nur an den Stellen durchzuführen, an denen es bereits Änderungen, in eine bestimmte Richtung, vorgekommen sind. Wenn es keine Anzeichen gibt, dass geplante Änderungen in Zukunft gebraucht werden, wäre er zum einen Zeitverschwendung, zum anderen erhöt es die Komplexität, ohne Nutzen davon zu erhalten [2, S. 322].
Auch wenn Code eine sehr schlechte Qualität aufweist, ist es nicht sinnvoll diesen zu Refacotren, solange keine Änderungen nötig sind. Ist er so gesehen als externe API nutzbar und funktioniert, bringt es keinen Vorteil diesen zu verbessern [4, S. 89].

„Darüber hinaus gibt es noch den Fall, dass es einfacher ist, den Code neu zu schreiben, als ein Refactoring vorzunehmen. Das ist allerdings eine schwierige Entscheidung.“[4, S. 90]

Auch hier sollte auf Grundlage von Erfahrung und Beobachtung gehandelt werden. Es sollte eindeutige Hinweise geben, dass eine Neuentwicklung auf lange sicht mehr Zeit spart und Verbesserungen liefert, als diesen Teil umfassend zu refactoren [4, S. 90].

5 Sicherheit in der Softwareentwicklung

Wird Software neu entwickelt oder werden Änderungen vorgenommen, ist es von hoher Bedeutung, die Sicherheit zu haben, dass alles wie erwartet funktioniert. Egal wie erfahren ein Entwickler sein mag, Fehler sind unausweichlich.
Um mehr Sicherheit zu erlangen, werden im Folgenden Maßnahmen vorgestellt, die Funktionstüchtigkeit von Softwareprojekten gewährleisten. Insbesondere im Kontext von Refactoring spielt dies eine wichtige Rolle, da dort oft umfassende Änderungen vorgenommen werden [2, S. 323].

5.1 Verschiedene Arten von Tests

Die einfachste und verbreitetste Möglichkeit, die Korrektheit von etwas zu überprüfen, ist das Testen. Auch in der Softwareentwicklung ist dies der Fall. Allerdings ist es hier nicht so einfach, wie es sich anhört: Rund die Hälfte der Entwicklungszeit wird in das Testen investiert [7, Introduction].
Es gibt viele verschiedene Arten, um Software zu testen, wobei diese sich auf unterschiedlichen Ebenen bewegen. Die folgende Auflistung soll einen Überblick darüber geben.

- **Unit-Tests** (oder Modultests) sind ein Prozess, bei dem die einzelnen Unterprogramme, Unterroutinen, Klassen oder Funktionen in einem Programm getestet werden. Der Test wird also nicht auf das gesamte Programm gerichtet, sondern nur auf kleinere Bausteine [7, S. 85].
- **Integrations Tests** sollen das Zusammenspiel von einzelnen Modulen, aus dem vorherigen Schritt, testen. Oft ist es der Fall, dass Integrations Tests nicht separat durchgeführt werden, sondern die Stellen durch umfangreichere Unit-Tests beretis abgedeckt werden [7, S. 117 f.].
- **Funktionale Tests** sind ein Prozess, um Abweichungen zwischen dem Programm und der externen Spezifikation zu finden. Eine externe Spezifikation beschreibt das Verhalten des Programms aus der Sicht des Endbenutzers. Diese Tests werden normalerweise als Black-Box-Test durchgeführt, da das interne Verhalten des Codes in den vorherigen Schritten überprüft wurde. [7, S. 119]

Diese Liste ist unvollständig, da nur Test-Arten gelistet sind, die den Code direkt betreffen. Es gibt zusätlich noch **System Tests**, **Akzeptantz Tests** und **Installations Tests**. Diese zielen eine höhere Ebene an und sollen die Software als ganzes Testen um festzustellen, ob die angeforderten Problemstellungen des Kunden richtig erfüllt werden und somit

z.B.: Missverständnisse bei den Anforderungen ausgeschlossen werden können. Es ist wichtig zu verstehen, dass diese Ebenen unabhänig voneinander laufen müssen. Funktionieren alle Unit-Tests ordnungsgemäß, heißt das nicht, dass die Software insgesamt funktioniert, da z.B.: Klassen falsch verwendet werden können.

„[Außerdem bleibt immer] das Risko, dass unsere Tests genau die Stelle, an der ein Fehler auftritt, nicht abdecken oder dass sie etwas anderes testen, als wir glauben.“ [2, S. 323]

Um den Zeitaufwand des Testens auf lange Sicht geringer zu halten, gibt es wie in [2, S. 323] beschrieben, die Möglichkeit das Testen zu automatisieren.
Um noch einen Schritt weiter zu gehen hat sich in der modernen Softwarentwicklung das Konzept der „Continous Integration“ (CI) etabliert. Dabei muss jeder Commit oder jeder Pull Request vorher definierte Aufgaben erfolgreich absolvieren. Unit-Tests, Buildvorgänge und automatisches Deployment sind dabei häufig Bestandteile. Das kontinuierliche Durchlaufen der Tests, ermöglicht eine frühzeitige Erkennung von auftretenden Fehlern [6].

5.2 Wekzeuge in der Entwicklungsumgebung

Softwareentwicklung ist ein zeitaufwendiger Prozess, in dem es schnell zu Fehlern kommen kann. Um diese zu vermeiden, wurden verschiedenste Werkzeuge entwickelt, die die Programmierenden unterstützen sollen. Moderne IDEs (Integrated Development Environments) haben das Ziel die Entwicklungsgeschwindigkeit zu erhöhen und Fehler zu vermeiden. Sie kombinieren eine Vielzahl dieser hilfreichen Werkzeuge in einem einzelnen Programm. Darunter gehören unter anderem Syntax-Highlighting, Autovervollständigung, Debugging-Tools und und andere Überprüfungen, die kontinuierlich im Hintergrund laufen. Auch Werkzeuge, die der Entwickler bewusst einsetzen kann, werden angeboten. Dazu gehören zum Beispiel Versionsverwaltung und Refactoring-Tools [1, S. 275].
Die nebenstehende Abbildung 5.1 zeigt das Kontextmenü für Refactoring Hilfen in der IDE WebStorm. Hier kann der Entwickler verschiedene Aktionen auswählen, die dann automatisiert durchgeführt werden.

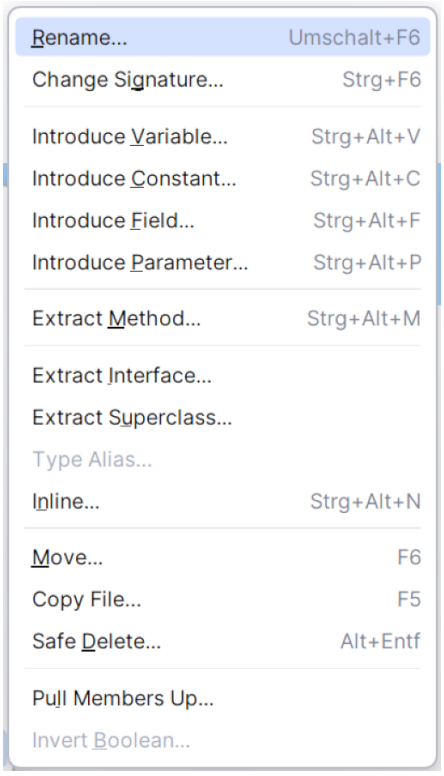

Abbildung 5.1: Screenshot WebStorm Refactoring-Tools [8]

Ein Beispiel dafür ist die Namensänderung eines markierten Feldes. Dabei kann ein neuer Name eingegeben werden und die IDE ändert alle Stellen, an denen das Feld verwendet wird, automatisch ab. Diese Unterstützung wird für die meisten gängigen Refactorings angeboten.[8] Trotzdem ist es wichtig, sich nicht vollständig auf diese Werkzeuge zu verlassen. Vorgenommene Änderungen sollten immer überprüft werden, da auch diese Tools fehlerhaft sein können [2, S. 323 f.].

5.3 Formale Verifikation der Software

Die nächste Thematik spielt vor allem bei besonders kritischer Software eine Rolle. Beispiele hierfür sind Systeme, von denen Menschenleben abhängen, oder bei denen erhebliche Geldsummen im Spiel sind, wie Flugzege oder der nächste Marsrover [2, S. 324].
Formale Verifikation beschreibt das Erheben einer Beweisführung über die Funktionstüchtigkeit von Code. Alle möglichen Ausgänge eines Programms müssen auf mathematischer Ebene bewiesen werden [5]. Da dieser Prozess extrem zeitaufwendig ist und Fehler in der Beweisführung unter allen Umständen vermieden werden sollten, gibt es auch hier spezielle Werkzeuge, die den Entwickler unterstützen und die Berechnungen auf Richtigkeit überprüfen. Auch hier gilt, kommt externe Software zum Einsatz, sind Fehler dieser nicht grundsätzlich auszuschließen [2, S. 324].

5.4 Eine Fehlertoleranz einbauen

Der letzte Ansatz, der untersucht werden soll, bezieht sich auf eine gewisse Fehlertoleranz der Software. Es gibt mehrere Möglichkeiten, diese in Code einzubauen. Eine davon ist das Verwenden von Feature-Schaltern. Diese können im Fehlerfall komplette Funktionaltäten deaktivieren, um die Software weiterhin lauffähig zu halten. Alternativ kann auch ein Mechanismus eingebaut werden, der bei einem Fehler auf eine ältere stabile Version der Software zurückgreift [2, S. 324].
Leider gewährleistet auch dieses Vorgehen keine 100-prozentige Sicherheit, da Fehler falsch erkannt werden können.

„Ein Beispiel hierfür wäre eine Funktion, die im Fehlerfall -1 zurückgibt, anstatt einen Fehler zu werfen. Das System erwartet vielleicht eine ganze Zahl, und -1 passt zu dieser Erwartung.“ [2, S. 324].

Zusammenfassend lässt sich sagen, dass es viele verschiedene Möglichkeiten gibt, die Sicherheit von Software zu gewährleisten. Jedoch ist keine dieser Methoden fehlerfrei. Es ist wichtig, mehrere dieser Ansätze zu kombinieren, um die Wahrscheinlichkeit von Fehlern zu minimieren. Diese Beobachtung speißt die Aussage, dass Software nie fertig ist, da in ihrer Lebenszeit immer wieder neue Fehler entdeckt werden, die behoben werden müssen.

6 Ungenutzte Strukturen im Code

Im letzten Kapitel sollen einige Spezialfälle untersucht werden, bei denen oft aus verschiedenen Gründen auf ein Refactoring verzichtet wird. Gründe dafür können unter anderem Zeitmangel, fehlende Erfahrung oder gar Faulheit der Entwickler sein.

6.1 Leerzeilen nutzen

Wenn im Code Leerzeilen verwendet werden, hat dies zumeist einen Grund. Programmierende grenzen dadurch Gruppierungen von Aufrufen, Variablen oder generell zusammengehörenden Codeteilen voneinander ab. Da Leerzeilen in kürzester Zeit eingefügt werden können und einen großen Effekt zur Lesbarkeit des Codes beitragen, werden diese fast immer benutzt. Dies ist auch ein Grund dafür, dass viele kleinere Refactorings nicht durchgeführt werden, da diese mehr Zeit in Anspruch nehmen würden. [2, S. 325]

```
1  function login(username: string, password: string) {
2      if (username !== "admin" || password !== "123") {
3          throw new Error("Invalid credentials");
4      }
5      const loginUser = getUser(username)
6
7      userHistroy.push(loginUser);
8      currentUser = loginUser;
9  }
```

Abbildung 6.1: Loginmethode mit einer Leerzeile

Das Codebeispiel 6.1 zeigt eine stark vereinfachte Methode, um einen Login durchzuführen. In Zeile Nr. 6 wird die Überprüfung der Anmeldedaten und die anschließende Anfrage des Nutzers, von den weiteren Bestandteilen des Logins räumlich voneinander getrennt. Das durchzuführende Refactoring wirkt in diesem Fall fast trivial. Die beiden Bestandteile vor und nach der Leerzeile können nach [2, S. 325] in eigene Methoden gezogen werden. Es müssen nur noch passende Funktionsnamen gewählt werden. Das folgende Ergebnis 6.2 entsteht dadurch.

```
function updateCurrentUser(loginUser: String) {
    userHistory.push(loginUser)
    currentUser = loginUser;
}

function authenticateUser(username: String, password: string) {
    if (username !== "admin" || password !== "123") {
        throw new Error("Invalid credentials");
    }
    return getUser(username);
}

function login(username: string, password: string) {
    const loginUser = authenticateUser(username, password);
    updateCurrentUser(loginUser);
}
```

Abbildung 6.2: Loginmethode mit extrahierten Untermethoden

Ein weiterer Punkt, an dem oft auf Leerzeilen zurückgegriffen wird, ist das Erstellen von Klassen. Dort werden oft zusammengehörende Felder gruppiert. Das folgende Codebeispiel 6.3 zeigt den Anfang einer Klasse für eine Nutzer Authentifizierung.

```
class Authentication{
    private username: string;
    private password: string;

    private timestamp: Date;
}
```

Abbildung 6.3: Authentication Klasse mit einer Leerzeile

Hier kann an der 4. Zeile eine klare Einteilung zwischen den tatsächlichen Nutzerinformationen und weiteren zur Authentifizierung gehörenden Metadaten erkannt werden. Auch in diesem Fall ist das nach [2, S. 326] durchzuführende Refactoring selbsterklärend. Der folgende Codeausschnitt 6.4 zeigt ein mögliches Ergebnis.

```
class UserCredentials{
    private username: string;
    private password: string;
}

class Authentication{
    private userCredentials: UserCredentials;
    private timestamp: Date;
}
```

Abbildung 6.4: Nutzerinformationen in eigener Klasse

6.2 Ähnlichen Code zusammenführen

Es gibt immer wieder Situationen, in den zwei Methoden oder Klassen nahezu identisch sind, es aber trotzdem einen kleinen Unterschied gibt. In solchen Fällen wird oft auf das Refactoring verzichtet, da der Aufwand zu groß erscheint oder die Entwickler im Programmierfluss keine einfache Lösung finden den Code zusammenzuführen.[2, S. 326 f.]
Das folgende Beispiel zeigt zwei Klassen, die jeweils einen Formatierer umsetzen.

```
class XMLFormatter {
    format(vals: string[]) {
        let result = "";
        for (let i = 0;
              i < vals.length; i++) {
            result +=
              `<value>${vals[i]}</value>`;
        }
        return result;
    }
}
```

(a) Klasse eines XML-Formatierers

```
class JSONFormatter {
    format(vals: string[]) {
        let result = "";
        for (let i = 0;
              i < vals.length; i++) {
            if (i > 0) result += ",";
            result +=
                `{ value: "${vals[i]}" }`;
        }
        return result;
    }
}
```

(b) Klasse eines JSON-Formatierers

Abbildung 6.5: Ähnliche Formatierer Klassen [2, S. 327]

Vergleicht man die beiden Klassen 6.5a und 6.5b, fällt auf, dass diese nahezu identisch sind. Ein Unterschied ist zum Einen die abweichende Formatierung jedes einzelnen Wertes. Bei XML wird der Wert in ein Tag eingebettet, bei JSON wird ein Objekt mit einem Schlüssel-Wert-Paar in geschweiften Klammern erstellt. Dieser Unterschied liese sich einfach vereinen, da dort nur unterschiedliche Zeichenketten eingefügt werden müssen. Die ausschlaggebende Problematik stellt das trennende Komma zwischen einzelnen Werten, im JSON Format, dar. Dies verhindert eine einfache Zusammenführung der beiden Klassen. Da das Refactoring aus mehreren kleinen, aber einfachen Schritten besteht, betrachten wir das in der nachfolgenden Abbildung 6.6 gezeigte Ergebnis.

```
class XMLFormatter {
    format(vals:string[]) {
        return new Formatter(
            new FormatSingle
                ("<value>", "</value>"),
            new None()).format(vals);
    }
}
class JSONFormatter {
    format(vals:string[]) {
        return new Formatter(
            new FormatSingle
                ("{ value: '", "' }"),
            new Comma()).format(vals);
    }
}
class Formatter {
    constructor(
        private single: FormatSingle,
        private sep: Seperator) { }
    format(vals: string[]) {
        let result = "";
        for (let i=0; i < vals.length; i++)
        {
            reuslt =
              this.sep.put(i,result);
            result +=
              this.single.format(vals[i]);
        }
        return result;
    }
}
```

(a) Formattierer Klassen

```
class FormatSingle {
    constructor(
        private before: string,
        private after: string) { }
    format(val: string) {
        return `${prefix}${val}${after}`
    }
}

class Seperator {
    put(i: number, result: string): string;
}
class Comma implements Seperator {
    put(i: number, result: string) {
        if (i > 0) result += ",";
        return result;
    }
}
class None implements Seperator {
    put(i: number, result: string) {
        return result;
    }
}
```

(b) Seperator Klassen

Abbildung 6.6: Ergebnis der Formatierer nach Refactoring [2, S. 329 f.]

Bei der Betrachtung der Klasse *Formatter* fällt auf, dass diese eine große Ähnlichkeit zu den beiden ursprünglichen Klassen aufweist. Der Unterschied besteht darin, dass im Konstruktor zwei weitere Klassen übergeben werden, die das Formatieren der einzelnen Werte und das Einfügen des trennenden Kommas übernehmen.
Die erste Klasse *FormatSingle* übernimmt die genereller Formatierung einer Eingabe. Sie kann mit zwei beliebigen Zeichenketten instanziiert werden, die jeweils vor und nach dem eigentlichen Wert eingefügt werden.
Die Zweite, *Seperator*, übernimmt das Einfügen eines trennenden Zeichens zwischen den einzelnen Werten. Bei der Klasse *Comma* wird ein Komma eingefügt, wenn der Index größer als 0 ist. Die Klasse *None* fügt kein Zeichen ein.
Abschließend müssen noch die jeweiligen Klassen, *XMLFormatter* und *JSONFormatter*, mit den jeweils richtigen Werten instanziiert werden.

Das Ergebnis dieses Refactorings ist eine deutlich verbesserte Struktur, die die Wartbarkeit und Erweiterbarkeit des Codes deutlich erhöht. Es können ohne Probleme weitere Implementierungen von Formatierern erstellt werden, die nach ähnlichen Mustern arbeiten.

6.3 Gemeinsame Affixe nutzen

Ein weiteres eindeutiges Merkmal, für ein klares Refactoring, ist die Verwendung von gemeinsamen Affixen in Methoden- oder Klassennamen. Dies deutet darauf hin, dass diese Codeteile eine klare Zusammengehörigkeit haben und in eine gemeinsame Struktur gekapselt werden können. Laut der Regel ***„Vermeide gemeinsame Affixe“***, können alle Felder und Methoden mit gleichem Präfix oder Suffix in eine eigene Klasse oder ein eigenes Interface extrahiert werden. Das Affix wird dabei zum Namen der neu erstellten Struktur [2, S. 201 ff.].
Das folgende Beispiel zeigt Code, bei dem sofort auffällt, dass diese Regel verletzt wird.

```
interface Protocol {...}
class StringProtocol implements Protocol {...}
class JSONProtocol implements Protocol {...}
class ProtobufProtocol implements Protocol {...}
/// ...
    let p = new StringProtocol();
/// ...
```

Abbildung 6.7: Codeausschnitt mit gemeinsamen Suffix [2, S. 331]

Wird der Code betrachtet, fällt auf, dass *„Protocol“* in allen Klassennamen vorkommt. Der Grund warum Programmierende dieses Refactoring in diesem Fall nicht durchführen, wird klar, wenn das Ergebnis betrachtet wird. Entfernt man bei der Klasse *StringProtocol* das Suffix, bleibt nur noch *„String“* übrig. Dies steht in Konflikt mit einer Basisklasse der Programmiersprache [2, S. 330 f.]. Um diese Codestelle trotzdem zu verbessern, kann das Refactoring in leicht abgeänderter Form durchgeführt werden. Wie im folgenden Codebeispiel 6.8 gezeigt, kann das Problem mithilfe eines *„Namespace“* gelöst werden.

```
namespace ptotocol {
    export interface Protocol {...}
    export class String implements Protocol {...}
    export class JSON implements Protocol {...}
    export class Protobuf implements Protocol {...}
}

/// ...
    let p = new protocol.String();
/// ...
```

Abbildung 6.8: Codeausschnitt mit Namespace [2, S. 331]

Dieser Ansatz ist zwar zielführend, trotzdem sollte hier von Fall zu Fall entschieden werden, ob es eine bessere Lösung gibt, dies zu lösen. Da *„String“* eine häufig verwendete Klasse ist, erschwert das gezeigte Beispiel die Nutzbarkeit des Codes weiter und ist nicht sinnvoll.

6.4 Den Laufzeittyp bearbeiten

Das letzte Beispiel, das betrachtet werden soll, zeigt eine Situation bei der auf den Operator ***„instanceof“*** zurückgegriffen wird, um abhängig davon verschiedene Aufrufe ausgeführt werden sollen. Die Unterscheidung wird mithilfe von ***„if“***-Abfragen durchgeführt. Laut Clausen sollten ***„if“***-Anweisungen in Kombination mit ***„else“*** vermieden werden. Nur Fälle mit ***„instanceof“***, ***„typeof“*** oder Casts werden als Ausnahme genannt. [2, S. 332] Das folgende Beispiel 6.9 zeigt solch eine Situation.

```
1  function foo(obj: any) {
2      if (obj instanceof A) {
3          obj.methodA();
4      } else if (obj instanceof B) {
5          obj.methodB();
6      }
7  }
8  class A {
9      methodA() { ... }
10 }
11 class B {
12     methodB() { ... }
13 }
```

Abbildung 6.9: *if*-Anweisung mit *instanceof* [2, S. 331]

In der Funktion *foo* wird überprüft, ob das übergebene Objekt eine Instanz der Klasse *A* oder *B* ist. Abhängig vom Ergebnis werden dann ihre jeweiligen Methoden aufgerufen. Dieses Verhalten ist ein Paradebeispiel für die Nutzung eines Interfaces.

„Interfaces erlauben es uns, einer Variablen Objekte verschiedener Klassen zuzuweisen. Wenn wir dann eine Methode daran aufrufen, wird der Aufruf zur richtigen Klasse geleitet. Gleichzeitig ist das auch der Weg, wie wir Typenuntersuchungen zur Laufzeit vermeiden können.“ [2, S. 332]

Das folgende Beispiel 6.10 zeigt, wie das Problem mit dem erklärten Ansatz gelöst werden kann.

```
function foo(obj: Foo) {
    obj.foo();
}
class A implements Foo {
    foo() {
        this.methodA();
    }
    methodA() { ... }
}
class B implements Foo {
    foo() {
        this.methodB();
    }
    methodB() { ... }
}
interface Foo {
    foo(): void;
}
```

Abbildung 6.10: Refactoring mit Interface [2, S. 332 f.]

Abschließend lässt sich sagen, dass Refactoring in den gezeigten Situationen oft zu einer besseren Codebasis führt. Auch wenn es im Moment des Programmierens nicht sinnvoll wirkt, sollte der kleine Mehraufwand in Kauf genommen werden. Es ist wichtig, dass Entwickler sich immer wieder bewusst machen, dass Code nicht nur für den Moment geschrieben wird, sondern auch in Zukunft noch wartbar und erweiterbar sein sollte. Müssen andere Entwickler im Nachgang den Code anpassen, nimmt dies mehr Zeit in Anspruch, als wenn der Code von Anfang an sauber strukturiert ist.

7 Fazit

Abschließend lässt sich sagen, dass Refactoring ein wichtiger Bestandteil der Softwareentwicklung ist. Es hilft dabei, die Qualität und Wartbarkeit von Code zu verbessern und somit die Lebensdauer von Software zu verlängern.
Das Seminar hat gezeigt, dass Strukturen im Code eine wichtige Rolle spielen und Verhalten in verschiedenen Formen abgebildet werden kann. Auch das Vezichten auf Refactoring kann in manchen Fällen sinnvoll sein, um die Entwicklung nicht zu verlangsamen, auch wenn es auf den ersten Blick nicht so wirkt. Ein weiterer wichtiger Punkt ist die Sicherheit von Software. Diese kann durch Tests, Werkzeuge, formale Verifikation und Fehlertoleranz gewährleistet werden.

Literatur

[1] Zakieh Alizadehsani u. a. „Modern Integrated Development Environment (IDEs)“. In: *Sustainable Smart Cities and Territories*. Hrsg. von Juan M. Corchado und Saber Trabelsi. Cham: Springer International Publishing, 2022, S. 274–288. ISBN: 978-3-030-78901-5.

[2] Christian Clausen. *five lines of code - Clean Code durch gezieltes Refactoring*. Bonn: Rheinwerk Verlag, 2023, S. 201–203, 311–333. ISBN: 9783836292245.

[3] Melvin E Conway. „How do committees invent“. In: *Datamation* 14.4 (1968), S. 28–31.

[4] Martin Fowler. *Refactoring - Wie Sie das Design bestehender Software verbessern*. 2. Aufl. Frechen: mitp Verlags GmbH & Co. KG, 2020, S. 80–90. ISBN: 9783958459410.

[5] Martin Glinz. *Formale Verifikation*. 2004. URL: `https://files.ifi.uzh.ch/rerg/amadeus/teaching/courses/kvse_ss05/kapitel_04.pdf` (besucht am 01. 07. 2024).

[6] Mathias Meyer. „Continuous Integration and Its Tools“. In: *IEEE Software* 31.3 (2014), S. 14–16. DOI: `10.1109/MS.2014.58`.

[7] Glenford J. Myers, Corey Sandler und Tom Badgett. *The Art of Software Testing*. 3. Aufl. John Wiley & Sons, Inc., 2012. ISBN: 9781118133156.

[8] JetBrains s.r.o. *WebStorm, Version 232.10203.14*. Screenshot. Verfügbar unter: `https://www.jetbrains.com/de-de/webstorm/`. 2024.

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig angefertigt, nicht anderweitig für Prüfungszwecke vorgelegt, alle benutzten Quellen und Hilfsmittel angegeben, sowie wörtliche und sinngemäße Zitate gekennzeichnet habe.

Füssen, den 10. Juli 2024 .....

Unterschrift des Verfassers