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 solle 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 [1]. 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.
Neben dem Untersuchen des Verhaltens und der Struktur im Code, 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.

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.*"[1]
Das heißt also, dass Code verschiedenste Gegebenheiten aus der Realität abbildet. 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.

2.1 Verschiedene Arten von Strukturen

Es gibt verschiedene Bereiche in der Softwareentwicklung, in denen Struktur eine Rolle spielt. Es ist möglich diese Bereiche an zwei Achsen einzuteilen. Zum einen gibt es einige Faktoren welche den Menschen, also die Softwareentwickler betreffen. Auf der anderen Seite befindet sich der tatsächliche Code. Um die Einteilung sinnvoll zu gestalten, teilt die andere Achsen anhand des Wirkungsbereiches der Strukturen ein. [1] 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 [1] (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 Systemen gibt. [2] Diese Aussage wurde von Harvard Wissenschaftlern bestätigt. [3]

2.2 Weitere Einschränkungen

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.[1]

3 Arten, wie Code Verhalten spiegelt

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, einen Sonderfall darstellen. [1]

Beispielverhalten

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.[1]

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. [1] In folgender Abbildung werden dafür jeweils einfache Beispiele gezeigt. (3.1)

```
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 [1]

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.

3.1.1 Eigenes Beispiel

Folgende Abbildung zeigt das Beispiel-Verhalten 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 3.4 lässt sich erkennen, dass diese Aufrufe durch die Kontrolloperatoren *for* und *if* gesteuert werden.

3.1.2 Vor und Nachteile

Da das Programmieren im Kontrollfluss schnell und einfach funktioniert, eignet sich diese Art gut, um neues Vehalten initial abzubilden.

3.2 Verhalten in der Struktur der Daten

3.2.1 Eigenes Beispiel

3.3 Verhalten in den Daten

3.3.1 Eigenes Beispiel

```
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 3.3: Beispiel in einer Datenstruktur

```
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 3.4: Beispiel in einer Datenstruktur

4 Ist Refactoring immer sinnvoll?

4.1 Beobachten statt vorhersagen

5 Sicherheit gewinnen, ohne den Code zu verstehen

Wenn Änderungen an einer Software durchgeführt werden, ist es von hoher Bedeutung, die Sicherheit zu haben, dass danach weiterhin alles funktioniert. Auch die erfahrensten Entwickler machen hin und wieder Fehler. Fehler sind menschlich.
Im folgenden werden einige Maßnahmen vorgestellt, die die Funktionstüchtigkeit von Softwareprojekten gewährleisten können. Insbesondere im Kontext von Refactoring spielt dies eine wichtige Rolle, da dort oft umfassende Änderungen vorgenommen werden. vgl. [1, S. 323]

5.1 Sicherheit durch 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 [5, 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 Prozeduren in einem Programm getestet werden. Genauer gesagt wird der Test nicht zunächst auf das gesamte Programm konzentriert, sondern zuerst auf die kleineren Bausteine des Programms [5, S. 85].
- **Integrations Tests** sollen das Zusammenspiel von einzelnen Modulen, aus dem vorherigen Schritt, testen. Oft ist es der Fall, dass Integrations Tests nicht seperat durchgeführt werden, sondern die Stellen durch umfangreichere Unit-Tests beretis abgedeckt werden [5, 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. [5, 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 ordnugnsgemäß, 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.“* [1, S. 323]

Um den Zeitaufwand des Testens auf lange Sicht geringer zu halten, gibt es wie in [1, S. 323] die Möglichkeit einige dieser Tests zu automatisieren.

In der modernen Softwareentwicklung sind automatiserte Tests oft Teil einer „Continous Integration“. Dabei muss jeder Commit oder jeder Pull Request vorher definierte Aufgaben erfolgreich absolvieren. Dabei können Unit-Tests, Buildvorgänge [4]

5.2 Sicherheit durch Werkzeuge

Gerade für Refactorings bieten die meisten modernen Entwicklungsumgebungen verschiedenste Werkzeuge, die den Entwickler unterstützen sollen. In der nebenliegenden Abbildung 5.1 ist ein Auszug aus den Refactoring-Tools

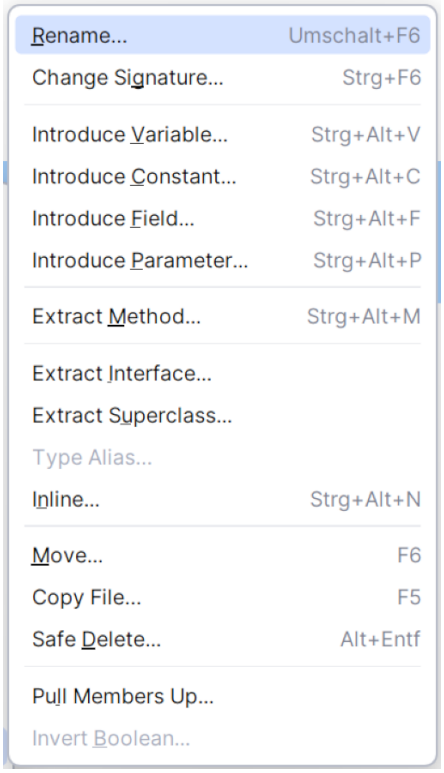

Abbildung 5.1: Screenshot WebStorm Refactoring-Tools [6]

5.3 Sicherheit durch formale Verifikation

5.4 Sicherheit durch Fehlertoleranz

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 zusammengehörigen 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. [1, 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 annschließende 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 [1, S. 325] in eigene Methoden gezogen werden. Jediglich passende Funktions Namen müssen 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

Eine 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 [1, 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 timestamp: Date;
    //..
}
```

Abbildung 6.4: Loginmethode mit einer Leerzeile

6.2 Doppelten 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.[1, S. 326 f.]
Das folgende Beispiel zeigt zwei Klassen, die jeweils einen Fromatierer 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-Formattierers

```
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-Formattierers

Abbildung 6.5: Ähnliche Formattierer Klassen [1, 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 lassen, da dort nur unterschiedliche Zeichenketten eingefügt werden müssen. Die ausschlaggebende Problematik stellt, in diesem Beispiel, das trennende Komma zwischen einzelnen Werten im JSON Format dar. Dies verhindert eine einfache Zusammenführung der beiden Klassen. Im folgenden wird ein möglicher Lösungsansatz gezeigt, solche Probleme trotzdem zu lösen. blabla

6.3 Gemeinsame Affixe nutzen

6.4 Den Laufzeittyp bearbeiten

Literatur

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

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

[3] Alan MacCormack, Carliss Baldwin und John Rusnak. „Exploring the duality between product and organizational architectures: A test of the “mirroring” hypothesis“. In: *Research policy* 41.8 (2012), S. 1309–1324.

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

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

[6] 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 27. Juni 2023 .....

Unterschrift des Verfassers