Einführung

Gelassener reden!!!

Zu Beginn:

* [software@alexander-weber.com](mailto:software@alexander-weber.com) und
* <https://alexander-weber.com>
* Das Bild mit dem Schalter und der Lampe malen.
* UML Wissensstand abfragen ggf. wiederholen
* Den Pfad zu den Übungen aufschreiben

Allgemeine Vorstellung

So guten Morgen zusammen. Ich möchte mich mal kurz vorstellen. (Wie Ihr bereits schon kennt) mein Name ist Alexander Weber. Ich komme ursprünglich aus Kasachstan lebe allerdings schon über 25 Jahren in Deutschland. Ich habe in Nürnberg Elektro- und Informationstechnik Studiert und arbeite schon seit 11 Jahren bei der Siemens (Energy) als Softwareentwickler überwiegend im embedded Bereich.

Key Development Expert

2018 bin ich zum „Key Development Expert“ ernannt worden.

MicroConsult

2019 begann meine nebenberufliche Trainer Kariere bei bzw. für die Fa. MicroConsult. Dort gebe ich diverse C++ Kurse und den „Clean Code“

Embedded Software Engineering Kongress

Kurs. 2021 habe ich, auf dem „Embedded Software Engineering Kongress“, einen Vortrag über Dependancy Injection mit C++ gehalten.

Programmiersprachen

In der Zeit als Softwareentwickler habe ich schon einige Programmiersprachen kennen gelernt. Programmiersprachen wie C, C++, Java, Python und sogar VHDL waren in der Laufbahn drin.

Handy App Programmierung

Darüber hinaus habe ich mich in meiner Freizeit mit der Handy App Programmierung beschäftig und daher auch solche Sprachen wie Kotlin, Swift und Dart ausprobiert.

Erfahrung und Clean Code

Ich kann euch deswegen nur eins sagen, das Thema „Clean Code“ sollte man wirklich nicht aus der Acht lassen. Warum es so Wichtig ist, sauber zu Coden werden wir uns hier gemeinsam anschauen.

Fragen?

Habt ihr noch Fragen zu meiner Person?

Kurze Pause 2 Sekunden

So nun kann ich sogar mit Sicherheit ein paar Dinge über euch sagen:

1. Ihr seid Programmierer
2. Ihr wollt bessere Programmierer werden.

Dann seid ihr bei dem Kurs definitiv richtig.

Ablauf und Zusätzliche Unterlagen

Im Laufe des Kurses, werde ich die bestehenden Unterlagen mit meinen eigenen ergänzen. Ich werde einige Bilder, Kodebeispiele, Zitate usw. zeigen. Um ihnen die Wichtigkeit des Themas „Clean Code“ möglichst gut rüberzubringen, damit ihr die Softwareentwicklung in Zukunft etwas ernster angeht.

Bitte verzeiht mir wenn ich zwischen den PDFs wechsle bzw. in meine Notizen gucke, denn ich möchte euch möglichst viel Wissen weiter geben.

Teilnehmer bekommen alles

Wichtig alles was ich euch hier zeige bekommt ihr am Ende des Kurses mit und sogar noch mehr.

Clean Code Spannend

Das Thema „Clean Code“ ist sehr spannend und eigentlich nicht so kompliziert, wenn man sich an gewisse Regeln hält. Welche Regeln das sind werde ich euch in den nächsten 3. Tagen zeigen.

Duzen, Fragen, Übungsaufgaben und Pausen.

Noch drei Sachen vorweg.

1. Ich hoffe es ist ok, wenn wir uns duzen. Ihr könnt mich also ruhig Alex nennen.
2. Zögert nicht stellt jederzeit Fragen. Bzw. wenn es zu schnell gehen sollte mich ruhig darauf hinweisen.
3. Wir werden jeden Tag Übungsaufgaben erledigen.
4. Wir werden in so ca. 1,5 Stündigen Abständen eine Pause machen.
5. 12:00 Uhr ist die Mittagspause

Das war’s erstmal von meiner Seite. Haben Sie bisher vielleicht schon Fragen?

5 Sekunden Pause!

Teilnehmer Vorstellung.

Vorname was ihr macht und wo, was Ihr vom Kurs erhofft

Nun würde ich mich freuen, wenn ihr euch auch etwas Vorstellt.

Mir reicht der Vorname was ihr macht und wo. So ähnlich wie ich mich vorgestellt hatte. Und zusätzlich noch was Ihr euch von dem Kurs erhofft. Je mehr ich über euch erfahre, desto besser kann ich den Stoff rüberbringen, da ich einiges aus Erfahrung sagen kann.

Wer von euch hat sich schon mal mit dem Thema Clean Code befasst?

* Ein bisschen?
* Gar nicht?

# 1. Lernziele

Folie 2.

Gelassener reden!!!

Was ist guter Kode?

Lasst uns mal zu dem Thema „Was ist guter Code?“ bzw. „Welche Eigenschaften besitzt guter Code?“ etwas Brainstorming machen. Welche Eigenschaften des Kodes werden durch das Säubern verbessert.

1. Lesbarkeit 🡺Leicht verständlich

* Denn der Kode muss viel öfter gelesen werden als er geschrieben wird, entweder durch euch selbst oder durch andere Kollegen.
* Keiner schreibt von Anfang an sauberen Kode. Wenn der Kode erstmal funktioniert, sollte mindesten die gleiche Zeit investiert werden, um diesen Lesbar bzw. sauber zu machen.
* Analogie zur Küche nach dem Kochen wird auch nicht alles stehen gelassen, sondern die Küche sauber gemacht.

Flexibel

1. Wartbarkeit/Erweiterbarkeit 🡺 Einfach änderbar

wartbar

* Wenn die Anforderungen sich ändern muss der Kode angepasst werden. Dabei kommen neue bzw. alte Bugs zum Vorschein.

Produktiv

* Software besteht aus zwei Wörtern Soft = Leicht (veränderbar) und Ware = Produkt.

Wir wollten das unsere Hardware veränderbar wird deshalb wurde Software erfunden.

* Beim Spaghetti Kode ist es manchmal sogar schneller neue Hardware zu designen.

1. Testbarkeit 🡺 Gut testbar

* Klassen sollen einzeln getestet werden.
* Abhängigkeiten zu anderen Klassen sollten auf ein Minimum begrenzt werden.
* Ohne Tests kein fehlerfreies Refactoring möglich

1. Wiederverwendbarkeit

* Module, Klassen und Methoden sollten nur eine Sache erledigen
* Sind dann auch klein und können somit auch leichter wiederverwendet werden.

Es kommen ein paar Zitate von berühmten Informatikern!!!

Hier kommt die Bilder von „cleancode-Robert Martin“ oder „CleanCode.key“ Folie 8-14

Frage: Wer ist Michael Feathers?

Antwort:

Autor von einem bekannten Buch namens: „Effektives Arbeiten mit Legacy Code. Refactoring und Testen bestehender Software“. Das Thema Refactoring werden wir uns am 3. Tag anschauen, dort sind sehr viele Beispiele aus dem Buch.

Frage: Wer ist Ward Cunningham?

Antwort: Erfinder von Wikipedia!

# 2. Softwareengineering

Folie 3.

Gelassener reden!!!

Frage: Für wen schreibt ihr überhaupt Kode? Für den Kunden? Für den Manager? Für Kollegen oder für sich selbst?

Antwort: Ihr schreibt in erster Linie für sich selbst.

Ihr müsst euch später darin zurecht finden. Ihr müsst es debuggen und erweitern.

Mit der Aussage „Was willst du es läuft doch!“ legt ihr euch selber Steine in den Weg.

Ein cooles Zitat besagt:

„Als ich den Kode geschrieben haben wusste nur ich und Gott was ich da tue! Jetzt weiß es nur noch der Gott!“

Da ihr euch nach wenigen Wochen selbst nicht mehr an den Kode erinnern könnt.

Wir Programmierer haben nämlich viel RAM (Flüchtigen Speichen) und wenig nicht Flüchtigen.

Hier noch ein cooler Spruch aus dem Vortrag „Bonus\cleancode-Robert Martin.pdf“ oder „CleanCode.key“ Folie 18

Frage: Was passiert in den meisten Fällen, wenn man den Kode des anderen Kollegen anpassen muss sobald sich die Anforderungen ändern?

Wir sagen oft es wird schneller sein, wenn wir auf den Kollegen warten bis er in 2 Wochen vom Urlaub wieder zurück ist. Weil die Einarbeitung meistens viel Zeit kostest, da man den Kode nicht so leicht verstehen kann.

Es ist viel besser, wenn der Kode nicht funktioniert man ihn aber doch relativ schnell zum Laufen bringen kann, da dieser verständlich ist. Als ein funktionierender Kode den keiner mehr Pflegen kann!

Folie 4.Der Preis schlechten Codes

Gelassener reden!!!

Frage: Ist Kodequalität überhaupt messbar?

Antwort:

Hier kommt das Bild mit den WTFs aus dem Vortrag „cleancode - Rules from Robert C. Martin's book Clean Code“ oder „Diapositiva 1“ Folie 10

Laut Robert Cecil Martin (Oncle Bob) ist Kodequalität anhand von WTFs pro Minute, aus einer Tür wo ein Codereview stattfindet, messbar.

In der Realität würdet ihr WTF sagen, wenn ihr sowas sehen würdet?

Hier kommt das Bild mit der Steckdose aus dem Vortrag „clean-code-vortrag-gearconf“ Folie 4.

Der Kode wird nie perfekt sein aber wir sollten es zumindest versuchen ihn zu verbessern.

Fragen: Wieso sind Programmierer so langsam?

Hat einer vor euch schon mal in einem Green Field Projekt gearbeitet?

Wenn, ja wie schnell habt ihr damals neue Features implementiert?

Antwort: Sehr schnell!

Frage: Und nach einem Jahr?

Antwort: UUUHHHHH wenn wir hier jetzt etwas ändern die ganze Hölle geht in Flammen auf!

Wenn man es dann doch angeht und etwas verzweifelt versucht einzubauen und debugged und debugged und auf einmal funktioniert es, hört man auf zu atmen. Man checkt den Kode vorsichtig, mit den Worten „Gott sei Dank“ ein und geht auf Zehenspitzen vom Arbeitsplatz und Guckt sich am besten nicht mehr um. Am nächsten Tag merkt man verzweifelt, dass etwas Anderes nicht mehr geht.

Fensterheber

Was wäre wenn ihr euer Auto mit defekten Fensterheber in eine Werkstatt bringen würdet. Der Mechaniker sagt „Kein Problem kommen Sie morgen wieder“. Ihr kommt am nächsten Tag vorbei der Meister zeigt euch stolz, dass die Fensterheber wieder einsatzfähig sind. Ihr freut euch steigt ins Auto wollt den Motor anzünden und der Motor springt nicht an. Was denkt ihr von dem Mechaniker „Er ist ein Vollidiot“. Würdet ihr das nächste Mal zu dem Mechaniker wieder kommen. Ich denke nicht. Das gleiche denken die Kunden auch über uns.

Zeit zum saubermachen

Aber das ist der falsche Ansatz. Wenn der Kode funktioniert, kommt die Zeit ihn sauber zu machen. Keiner schreibt von Anfang an sauberen Kode. Das wird wahrscheinlich eine Sauerei. Wir basteln es halt irgendwie zusammen. Verstehen vielleicht nicht einmal richtig wie es funktioniert. Nun sollte man sich am besten Zeit nehmen, um den Kode nochmal durch zu gehen, Unit-Tests zu schreiben, um danach beim „refaktorn“ ein Fangnetz zu haben, falls etwas schief geht und meistens geht es auch schief und danach zu Säubern. Die Hälfte der Zeit sollte für das Refaktorn investiert werden.

Architektur und TDD

Aber noch besser wäre es sich im Vorfeld Gedanken über die Architektur zu machen und mit z.B. TDD zu Entwickeln. Somit hat man Tests, die uns vor späteren Änderungen und möglichen Fehlern schützen sollen.

Pair Programming

Siehe Robert Martins youtube Video: <https://www.youtube.com/watch?v=7EmboKQH8lM&t=4551s&ab_channel=UnityCoin>

Ab 25:40 min

Folie 5. Warum ist guter Code wichtig?

Gelassener reden!!!

Hier kommt das Bild mit der dreckigen Küche und dem Zeit und Produktivität Graphen aus dem Vortrag „vladimirromanov-cleancode“ Folie 5.

Was macht das Management in den Fällen wenn wir langsamer werden und ein Termindruck herrscht?

Antwort: Stellt neue Leute ein in der Hoffnung doppelt so viele sind doppelt so schnell. Jedoch fast jeder weiß, dass es so nicht stimmt. Wer bringt den Neulingen das Programmieren bei, bzw. in welcher Umgebung sich die Leute unterwegs? Die gleichen Leute die diese Sauerei gemacht haben und meistens in einer Spaghetti Code Umgebung. Was wird dann normalerweise aus dem Code der wird mit der Zeit noch schlechter.

# 2.1. Gründe für schlechten Kode

Folie 6.

Gelassener reden!!!

Hier kommt eine Reihe von Bilder Diskussion mit dem Projektleiter aus dem Vortrag „clean-code-vortrag-gearconf“ Folie 9-13.

Folie 7. Problem anderer Leute

Problem anderer Leute.

Frage: Wie viele Prozessoren habt ihr gerade bei euch bzw. benutzt ihr täglich?

Antwort: Bei euch so ca. 10 davon sind die Meisten im Handy und benutzt täglich vielleicht 1000! So ein Handy hat heutzutage mehr Rechenpower als es 1980 auf der ganzen Welt gab.

Und was machen wir damit? Wir spielen Angry Birds aber das ist eine andere Geschichte ☺

Die Menschheit kann heutzutage ohne Software nichts mehr machen. Sei es Einkaufen gehen, Autofahren oder Geldabheben. Wir sind vollständig auf Software angewiesen.

Wie viele Zeilen Kode hat ein Auto heut zu Tage?

Mindestens 100 Mio. Kodezeilen. Sie glauben doch nicht, dass wenn ihr auf das Bremspedal drückt, dass da ein Kabel ist, das die Bremsklotze zusammen drückt?

Dass die Software Menschen umbringen kann haben wir auch schon gesehen siehe Tesla oder Boing.

Politiker denken die regieren die Welt, weit daneben. Programmierer regieren die Welt. Unsere Society ist abhängig von uns. Siehe Volkswagen Abgasskandal.

Also können wir uns die Aussage „Problem anderer Leute“ meiner Meinung nach nicht leisten.

Wir Programmierer sind für all die Menschen da draußen verantwortlich.

Siehe Robert Martins youtube Video: <https://www.youtube.com/watch?v=7EmboKQH8lM&t=4551s&ab_channel=UnityCoin>

Ab 12:00 min

Gelassener reden!!!

Folie 8. Mangelnde Ausbildung, Menschliche Probleme, Termindruck

Frage: Wie viele Programmierer sind wir auf der Erde?

Antwort: Ca. 100 Mio.!

Frage: Wer war der erste, der es auf einem elektronischen Gerät laufen ließ?

Antwort: Alan Turing! Das war 1946. Er hat damals binär mit Lochkarten programmiert.

Von 1946 1 Mann bis heute ca. 100 Mio.

Ich habe mal hochgerechnet wie viele Verdopplungen es waren. Man kann davon ausgehen, dass alle 5 Jahre die Anzahl der Softwareentwickler sich verdoppelt. Das bedeutet wieder rum, dass Sie ständig neue unerfahrene Programmierer ausbilden müssen. Wenn Sie nicht mit einem guten Beispiel vorangehen, werden sich diese Leute schnell an die Gegebenheiten anpassen.

Siehe Robert Martins youtube Video: <https://www.youtube.com/watch?v=Qjywrq2gM8o&t=1626s&ab_channel=UnityCoin>

Ab 9:40 min

Zum Thema fehlende Ressourcen.

Man hört oft wir müssen neue Leute einstellen, um schneller zu werden.

Frage: Sind doppelt so viele Leute doppelt so schnell?

Antwort: In der Einarbeitungsphase verlangsamen diese sogar den Entwicklungsprozess!

Folie 9. Broken Window

Mangelnde Motivation: Broken Window Entwickler: „Ich hab damit nichts zu tun, das ist nicht von mir!“ „Darauf kommt es jetzt auch nicht mehr an!“ „Warum soll ich anfangen, hier sauber zu implementieren?“

Folie 10.

Gelassener reden!!!

**Moment mal!!! Wenn wir uns das oben Genannte durch den Kopf gehen lassen. Bedeutet das etwa, dass wir faul, feige, unfähig und zu nichts zu gebrauchen sind???**

**Pause 2 Sekunden**

**Natürlich nicht! Wir können besser werden! Deshalb seid ihr ja da.**

Folie 11. Unsitten bei der Erstellung von Code

Zum Punkt Keine Kommentare:

Kommentare sind nicht immer gut.

Veraltet

Viele sind schon veraltet und werden auch nie gelöscht, obwohl man schon längst weiß, dass die nicht passen.

Nicht was sondern wieso

Kommentare sollen nicht sagen was der Kode macht, sondern wieso das so gemacht wird.

Aussagekräftiger Quellkode

Was der Kode macht soll mir der Quellkode selbst sagen.

Zum Punkt „Schlechte Namensgebung“ einige Kode Beispiele, da es in dem Skript nicht vorkommt.

Sind zwar Java Kode Beispiele sind aber dennoch verständlich.

Hier kommen die Kode Beispiele aus dem Vortrag „cleancode - Rules from Robert C. Martin's book Clean Code“ oder „Diapositiva 1“ Folie 11-21

# 2.2 Sauberer Code

Folie 13.

Frage: Was ist sauberer Kode?

Antwort: Ich würde es so formulieren: Wenn ein anderer Entwickler sich problemlos in eurem Kode zurecht findet und relativ schnell ein neues Feature selbstständig implementieren kann! Zur Erinnerung den Spruch „Jeder kann Kode schreiben, den die Maschine versteht aber nicht jeder, den ein Mensch versteht.

In der Praxis würde man lieber den Kode neu schreiben, als es zu pflegen.

Hier kommt die Folie „Was heißt clean?“ aus dem Vortrag „clean-code-vortrag-gearconf“ Folie 28.

Und „Was bewirkt Clean?“ aus dem Vortrag „clean-code-vortrag-gearconf“ Folie 33.

Zum Punkt „Die Abhängigkeiten zu anderem Code sind auf ein Minimum begrenzt.“

Frage: Warum sind Switch Case bzw. lange if-else abfragen schlecht?

Antwort:

Abhängigkeiten und Redeployment

Bringen eine Menge an Abhängigkeiten mit sich. Und wenn ein weiterer Case dazu kommt muss sehr viel neu kompiliert und redeployed werden.

„DO ONE THING“ Regel

Sie verletzen auch die „DO ONE THING“ Regel. Switch Cases tuen definitiv mehr als nur eine Sache.

Breaks

Switch Cases sind auch deshalb nicht so gut, da die BREAKs haben. Wenn ein BREAK vergessen wird bzw. gewollt ausgelassen wird, beginnt die Fehlersuche.

Siehe Robert Martins youtube Video: <https://www.youtube.com/watch?v=7EmboKQH8lM&t=4551s&ab_channel=UnityCoin>

Ab 1:20:00 min

Gelassener reden!!!

Folie 14. Guter Code motiviert

Ein cooler Spruch lautet „In a clean code bugs cannot hide“

Folie 15. Zusammenfassung

Die ideale Vorgehensweise wäre:

1. Architektur (UML)
2. TDD

Da keiner von uns von Anfang an Clean schreiben kann, sollte man nach dem kodieren refactorn. Damit man selbst und andere Leute den Code später lesen können.

Wir haben jetzt denke ich verstanden warum guter Kode so wichtig ist.

Bis hierher noch Fragen?

2s Pause

Nun wird es Zeit zu der Praxis über zu gehen. Aber vorher gehen wir in eine 15 Minütige

Kaffee PAUSE!!!

# Kapitel 2 OO in C

2.1. Schutz gegen direkten Zugriff

Folie 10

Zum Punkt „Das Kapseln der Member ermöglicht spätere Änderungen“

Dadurch, dass Membervariablen ausschließlich intern geändert werden, kann man davon ausgehen, dass die Anpassung nur an einer Stelle passiert.

Übung Implementierung von Klassen

Folie 13

In der Header Datei test.h können die Tests per Kompilerschalter aktiviert werden. Dazu müsst ihr nur das Define

ENABLE\_TESTS

Entkommentiert werden.

Meine Lösung zeigen und das const erwähnen.

Kapitel 3 Grundregeln

Gelassener reden!!!

Clean Code Development Windows App + “clean-code-vortrag-gearconf“ Folie 31.

# 2.1. DRY

Folie 5. Aufgabe (Übung DRY)

Dort werden die Protokolle für jeden Kunden individuell z.B. (**customer\_receipt\_0001.txt**) und der Bank als Sammellog im **customer\_receipt.txt** abgelegt.

Die Formatierung soll so aussehen 22.00 € oder 0.05 € und zwar für alle Ausgaben.

Bei der Lösung könnte man eine Report Klasse erstellen, die es noch etwas übersichtlicher macht.

Gelassener reden!!!

Folie 8.

Wiederholungen aus Ungeduld z.B. weil man es schnell zum Laufen kriegen möchte. Kann man machen, allerdings sollte man im Nachhinein Zeit investieren, um es Clean zu machen.

Folie 11.

Nutzt eine gute Entwicklungsumgebung wie Clion oder Visual Studio, diese IDEs können Funktionen ohne viel Aufwand extrahieren.

Zum Thema Regelmäßige Reviews:

Wer von euch praktiziert Pair Programing?

Pair Programing ist effizienter als Code Review, da man eine 5 Stündige Entwicklung nicht mit einer Stunde Code Review abdecken kann. Die Fehler Wahrscheinlichkeit bei Pair Programing ist deutlich geringer. Außerdem ist der Wissenstransfer um einiges besser und zwar in jeder Richtung sei es Architektur, Unittesting geschweige denn (Clean) Coding.

Folie 12.

Sowas wird durch Pair Programing auch nicht mehr so oft passieren.

Folie 13 Lösung.

Das UML Diagramm dazu Zeigen und erwähnen, dass die Klasse groß geworden ist und zusätzlich sehr viele statischen Methoden beinhaltet, die sehr hohe Kopplung mit sich nachziehen. Noch besser wäre es, wenn die Klasse Bank gar keine Ausgaben beinhaltet und die Ausgaben lediglich mittels einer zusätzlichen Klasse „ReportAmount“ gemacht werden siehe Single Responsability Principle.

Bei meiner Lösung sind die Files, die ich erstellt habe Pascal Case.

Zeige meine Implementierung der Bank Klasse. Wichtig Console ist eine statische Klasse. Statische Klassen sollten nach Möglichkeit gemieden werden. In dem Kapitel SOLID werde ich euch eine bessere Implementierung zeigen!

# 2.2 KISS

**Keep it small and simple**

Gelassener reden!!!

Folie 15.

Wir machen es meistens nicht willentlich komplex, es passiert meistens unbewusst während wir eine Aufgabe lösen, dass die Dinge komplex werden. Wenn es dann endlich funktioniert wollen wir es nicht mehr ändern, weil wir Angst haben, das nach dem Umbau was kaputt geht. Wenn wir Unit Tests hätten, hätten wir kein Problem mehr. TDD hilft umso mehr, da hier die Tests zuerst geschrieben werden danach der Source Code. Somit sind Änderungen bzw. Refaktoring-Maßnahmen nicht mehr so tragisch.

Folie 17.

Nachdem Kodereview dieser Folie

Frage:

Wie Groß sollten Methoden/Funktionen sein?

Antwort:

Regel Nummer 1. Funktionen müssen klein sein und Regel 2. Sie müssen kleiner sein als in Regel 1.

Frage:

Wie viele Zeilen soll eine Funktion haben?

Antwort:

Eine Funktion soll nur eine Sache machen.

Robert Martin sagt „Wenn man aus einer Funktion keine weitere mehr extrahieren kann ist die Funktion klein genug und macht dann auch mit Sicherheit nur eine Sache“. Das gleiche gilt übrigens auf für Klassen.

Gelassener reden!!!

Hier kommt die Folie „Theorie Funktionen“ aus dem Vortrag „cleancoding-theorie und praxis guide“ Folien 19-26. Funktionsname Folie 20 ist klein je größer der Scope ist. Private Funktionen haben somit längere Namen. Private Funktionen die aus Privaten Funktionen aufgerufen werden, haben sogar noch längere Namen. Stichwort Journalismus und Ehrlichkeit dem Leser gegenüber.

Wir sind alle Autoren, deshalb sollten wir auch zusehen, dass unser Kode leserlich und verständlich ist. Sie würden sicherlich kein Buch lesen wollen, dass Sie nicht verstehen.

Flag Argumente verletzen do one thing Regel.

class Concert...

public Booking book (Customer aCustomer, boolean isPremium) {...}

Besser: Zwei Methoden draus zu machen.

class Concert...

public Booking regularBook(Customer aCustomer) {...}

public Booking premiumBook(Customer aCustomer) {...}

**Wir müssen jetzt definieren was eine Sache ist.**

Eine Funktion macht nur dann eine Sache, wenn man aus der keine weitere Funktion extrahieren kann. Man wird nicht mit Methoden überschwemmt, denn bei der Extraktion werden neue Klassen, Module, Packages usw. kreiert.

Siehe Robert Martins youtube Video: <https://www.youtube.com/watch?v=7EmboKQH8lM&t=4551s&ab_channel=UnityCoin>

Ab 57:00 min

Übungen:

Bei der Solution wäre folgendes besser gewesen.

Zeile 10 // static const unsigned char day\_info[2][12] = {

// v----- avoid subtracting 1 from `month` parameter

{0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}, // ordinary and

{0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31} // <==LEAP year

};// ^^--------------------- only difference is here (in February)

return day\_info\_leap\_year[is\_leap\_year][month];

Bei der Kiss MyTries Lösung mit Klassen ist cool.

**Folie 20.**

Gelassener reden!!!

Dabei sollten die Public Sachen oben stehen. Stichwort Journalismus. Derjenige der die Klasse verwenden möchte interessiert sich für die Publics.

Möglichst wenig Information preisgeben, wie die Klassen innen drin Ihre Arbeit verrichten.

**Folie 21.**

SetData oder GetData sind keine besonders gute Namen besser wäre zu beschreiben was das Vorhaben der Methode ist “Tell don’t Ask” darüber reden wir aber noch.

# 2.4 Programming to an Interface

**Folie 23.**

Man sollte nur Funktionen im Header prototypisieren, wenn diese wirklich von einer externen Translation Unit gebraucht werden. Alle anderen Funktionen sollten static (private) sein.

Zu diesem Thema hört ihr noch mehr im Kapitel SOLID.

**Folie 24.**

Frage: Was fällt euch bei der Interface Methode hier auf?

Antwort: Es wäre sauberer einen ostream zu übergeben statt ofstream. Das gehört in die konkrete Implementierung. Ein File kann auch mittels fopen geöffnet und beschrieben werden.

Gelassener reden!!!

# 2.6 Vorsicht vor Optimierungen

Frage: Wie könnte man den Kode **Folie 38.** und **39.** noch schöner bzw. einfacher gestalten?

Antwort: Indem man die Methoden NoAccount(), OneAccount() und MultipleAccounts() in separate Kindklassen auslagert. Und die entsprechende Klasse dann in einer Factory erzeugt (siehe (Astract) Factory Pattern)

# 2.7 Principle of least surprise

**Folie 43.**

Oder ein Überladener Operator „+“ der eine „-“ Operation auslöst.

# Kapitel 3 SOLID

**Folie 3**

Gelassener reden!!!

Siehe Bilder Ordner z.B. das Bild „bridge-pattern“ und das SOLID Bild.

Hier ist eine interessante Art und Weise das SRP darzustellen.

# 3.1 Single Responsibility Principle

**Folie 4**

Siehe Bilder zu „SRP“ kommen später.

**Folie 5**

Besprechung nach der Übung:

Im Moment werden die Ausgaben in das logfile.txt gespeichert. Achten Sie speziell drauf wie viele Aufgaben jede Klasse übernimmt. Eine Klasse soll nur einen Grund zur Änderung haben.

Stellt euch vor ein Bankmitarbeiter hätte einen Verbesserungsvorschlag, der die Einwirkung in die Klasse „Account“ nach sich zieht und ein weiterer Vorschlag käme vom Kunden, der eine Verbesserung zum Thema Ausgabe hätte z.B. zur Formatierung. Die Klasse „Account“ müsste somit 2. Mal angepasst werden, da 2. völlig unterschiedliche Anforderungen sich ändern und jede Änderung kann bekanntlich zu Buggs führen. Das Single Responsebility Prinzip verhindert das verschiedene Requirements Eingriffe in die gleiche Klasse bedeuten –> nur eine Verantwortlichkeit.

Zu **Folie 9.**

Erinnert euch daran was ich vorhin gesagt hatte mit den verschiedenen Änderungswünschen verschiedener Personen.

Hier kommen die Folien „5-7“ aus dem Vortrag „solid-design-principles“ oder „Performance Development Review“

Hier kommen die Folien „5-12“ aus dem Vortrag „solid-prinzipien“

Hier kommen die Folien „26-29“ aus dem Vortrag „mit-agilen-praktiken-solide-systeme-bauen“ oder „DWX - solid“

Vielleicht gibt es bessere Beispiele!!!

Zu **Folie 10.**

Nach der Übung:

Warum sollten Klassen mit statischen Methoden gemieden werden (auch Helper Klassen genannt).

* Es herrscht eine starke Kopplung.
* Die sind nicht Thread sicher, da nur statische Attribute verwendet werden können.
* Die Testbarkeit wird erschwert.

In dem Beispiel würde entweder Konsolen-Ausgabe oder ins File schreiben funktionieren aber nicht beides. Wenn der Kunde nun ein anderes Ausgabe Medium wünscht, muss die Klasse Bank und Account erneut angepasst werden. An der Stelle wäre ein Interface besser geeignet. Dieses Interface implementieren dann die gewünschten Ausgabe Medien (Klassen), z.B. Disk, Console usw. Die jeweiligen Klassen Bank und Account haben eine Referenz auf das Interface als Attribut. Der User kann z.B. über die GUI auswählen, ob auf die Konsole oder auf Disk geschrieben werden soll.

Bei MyTries befindet sich auch eine UML und eine Version mit Dependency Injection.

Im Interface IWritable ist auch ein überladener Operator der sogenannte Funktor dabei.

# 3.2 Open Closed Principle

Gelassener reden!!!

Zu **Folie 12.**

Frage: Wie viele Files müssen angepasst werden und wie oft?

Antwort: Es müssen 2 Cpp-Files und 1 h-File angepasst werden und zwar jedes Mal, wenn ein neuer Kontotyp dazu kommt. Und das ist sehr fragil, da bei jeder Änderung die Klasse Account angepasst wird und somit neu kompiliert werden muss. Dies hat zur Folge, dass bei der Änderung unter Umständen etwas kaputt gehen könnte, was vorher funktionierte.

Account Objekt wird mit einem type\_ = kInvalid initialisiert. In der Switch Case Anweisung gibt es den kInvalid Case gar nicht geschweige denn einen Default Case.

Könnt ihr euch noch an die Frage: Warum sind Switch Case bzw. lange if-else abfragen schlecht errinnern?

Antwort: Bringen eine Menge an Abhängigkeiten mit sich. Und wenn ein weiterer Case dazu kommt muss sehr viel neu gebaut werden. Zusätzlich haben diese Breaks und wenn ein Break fehlt muss wieder überlegt werden, ob es so passt oder ein Fehler drin ist.

Nun fragt ihr euch vielleicht irgendwo müssen ja die Entscheidungen getroffen werden. Das stimmt die beste Lösung dafür ist diese in eine Factory oder Builder (siehe Design Patterns) zu vergraben und sogar diese Factory als ein Interface zu übergeben siehe „Abstract Factory Design Pattern“.

Lese langsam damit alle mitkommen die Folien „69-70“ aus dem Buch „Clean Code“ Seite im Buch 38

Zu **Folie 15.**

Hier kommen die Folie „16-24“ aus dem Vortrag „solid-prinzipien“

Hier kommen die Folie „8-11“ aus dem Vortrag „solid-design-principles“ oder „Performance Development Review“

Hier kommen die Folien „30-33“ aus dem Vortrag „mit-agilen-praktiken-solide-systeme-bauen“ oder „DWX - solid“

Das Design Pattern namens Dekoration, Bridge oder Proxy, sind gute Beispiele für OPC.

Zu **Folie 18.**

Besser das UML-Diagramm aus dem solid/solution/OPC Ordner zeigen. Da ist es klarer was zu tun ist.

Nach der Übung kann das Bild zu den weiteren OCP-Lösungskonzepten besprochen werden, das bei den Bildern liegt.

Gelassener reden!!!

# 3.3 Liskov Substitution Principle

Zur Übung **Folie 20.**

Wichtig ist der satz: „Die Kontoklasse ist bereits in einer DLL implementiert“.

Das bedeutet wir können da unter Umständen nicht mehr so einfach rein gucken.

PocketMoneyAccount schmeißt eine Exception. Es gibt auch ein UML-Diagramm.

Nach der **Folie 24.**

Hier kommt die Folie mit der Ente „34“ aus dem Vortrag „mit-agilen-praktiken-solide-systeme-bauen“

Hier kommt noch ein Beispiel Kode aus dem Vortrag „solid-dry-slap“ Folie „61-64“

Hier kommen die Folie „12-14“ aus dem Vortrag „solid-design-principles“ oder „Performance Development Review“

# 3.4 Interface Segregation Principle

Zu **Folie 25.**

Hier kommt die Folie „39“ aus dem Vortrag „mit-agilen-praktiken-solide-systeme-bauen“

Und die Folie „15-18“ aus „solid-design-principles“

Zu **Folie 31.** Eine Lösung bei MyTries

Das ist keine gescheite Lösung, da hier nur die Methoden Deposit und CalculateInterest über das Interface IAccount erreicht werden. Um die Methoden Withdraw und SetOverdraft zu erreichen, muss über das die Klasse SpecialAccount gegangen werden.

# 3.5 Dependency Inversion Principle

Zu **Folie 33.**

Log Methoden sind nicht implementiert!

Zu **Folie 34.**

Übrigens das hier sollte keine Aggregationsbeziehung sein, da weder eine Referenz noch Zeiger vom Logger reingereicht wird, sondern eine Komposition, somit müsste die Raute schwarz ausgefüllt sein.

Hier kommt die Folie „43“ aus dem Vortrag „mit-agilen-praktiken-solide-systeme-bauen“

Schaut euch das Bild genau an. Genauso programmieren nämlich die meisten!!!

Zu **Folie 37.**

Gelassener reden!!!

Hier kommt die Folie „43-49“ aus dem Vortrag „mit-agilen-praktiken-solide-systeme-bauen“

Frage: Gegen welches SOLID Prinzip wird hier verstoßen Folie 48?

Antwort: ISR da das Interface File zwei unterschiedliche Funktionalitäten beinhaltet Lesen und Schreiben und sollte deshalb aufgetrennt werden. Wieso soll die Klasse „Keyboard Reader“ die write Funktion implementieren und der „Printer Writer“ die read?

Hier kommen die Folie „19-21“ aus dem Vortrag „solid-design-principles“

Weitere Beispiele Folien „61-75“ aus dem Vortrag „solid-prinzipien“

Zu **Folie 38.**

Vergessen Sie das mit dem Test ändern Sie den Kode einfach ganz normal.

Gelassener reden!!!

# Kapitel 4 Weitere Regeln

# 4.2 Single Level Of Abstraction

Zu. **Folie 11**

Frage: Welches SOLID Prinzip verletzt die Methode GetTimeInfo()

Antwort

SRP, eine Klasse sollte nur eine Sache machen. Denken Sie an die unterschiedlichen Personen, die Änderungen auf verschiedenen Ebenen Ihrer Software vollziehen möchten.

Die Zeit hat nicht mit dem Logger zu tun.

Sollte deshalb ein eine eigene Klasse z.B. Time wandern.

Der Logger braucht die Zeit also ist es ein Service. Also sollte es per Interface in die Klasse injiziert werden.

Nach der **Folie 11**

kommt noch ein Beispiel Kode aus dem Vortrag „solid-dry-slap“ Folien „26-28“

# 4.3 Tell don’t ask

Nach der **Folie 15**

hier ein gutes Beispiel ab 15:20 min

<https://www.youtube.com/watch?v=biG2rSlth8k&ab_channel=JDDConference>

Stellen Sie sich vor Sie sitzen in der Bar bestellen ein Bier und müssen bezahlen. Sie holen Ihren Geldbeutel raus, geben es dem Kellner, der Kellner nimmt das Geld raus und gibt den Geldbeutel Ihnen wieder zurück. WTF? Das bedeutet wir wussten unter Umständen nicht einmal wieviel Geld wir hatten und ob es für ein Bier reichen würde. Wir haben vorher nicht einmal rein geschaut und nachgezählt.

So läuft es aber meistens wenn wir Programmieren.

Besser wäre wie im richtigen Leben:

Ich möchte ein Bier -> ich gucke, ob ich einen Geldbeutel dabei habe -> zähle das Geld für ein Bier -> wenn es reicht erst dann beträte ich die Bar und bestelle das Bier.

Und wenn ich zahlen muss, hole ich das Geld selbst raus.

Ein Live Coding mit dem DRY Solution Beispiel in der Methode ChangeBalance

ChangeBalance Methode sollte in die Klasse Account wandern.

Zu der **Folie 17**

Wie ich hoffentlich bereits erzählt hatte, sollte eine Methode, die den Zustand eines Objekts verändert, keinen Rückgabewert haben entweder Void + Log oder Void + Exception. Bei der Bank würde auch ein Text erscheinen falls es schief gehen würde.

Gelassener reden!!!

# 4.4 Favor Composition over Inheritance

Zu der **Folie 20**

**Hier wurde einfach nur falsch vererbt richtig wäre siehe „Decorator Pattern“**

Zu der **Folie 22**

Die Verständlichkeit des Codes wird beeinträchtigt, da sich die Implementierung auf mehrere Klassen aufteilt? Das stimmt so nicht siehe SRP und DRY (alle Verdopplungen in der Basis Klasse).

Nach der **Folie 24**

Das UML Diagramm aus WeitereRegeln/Solutions/FCoI zeigen. Hier entsteht genauso ein Diamant. Deshalb wird die Klasse DepositAccount virtual abgeleitet.

Man muss es natürlich nicht so designen. Aber so ein Design könnte durchaus von Ihnen oder einen Kollegen stammen. Auch mit Komposition, kann ein schlechtes Design entstehen, jedoch wird das Refactoring meist einfacher.

**Folie 25**

Das ist zwar keine Komposition was hier dargestellt wird, der Hintergedanke ist der Selbe (Decorator).

**Folie 26**

Das UML Diagramm im WeitereRegeln/Exercises/FCoI\_2 Ordner zeigen.

Hier allerdings würde ich bei den Methoden AddFeature und GetFeature statt der lokalen Variable namens „id“ den Namen „key“ bevorzugen. Id gibt es schon und das ist die Account ID.

Mit map.count(key) kann man auch noch prüfen, ob es vorkommt.

Hier hat man einfach nur gezeigt, dass es wie immer eine andere Lösung gibt, die mit Vererbung und nicht nicht Komposition. Diese ist sogar aus der „Clean Code“ Sicht viel flexibler als eine Komposition.

Gelassener reden!!!

# 4.5 Law of Demeter

Nach der **Folie 33.**

Dieses Bespiel bei Wiki <https://de.wikipedia.org/wiki/Gesetz_von_Demeter>

Frage: Was könnte hier noch verbessert werden? Hinweis DIP!!!

Antwort:

Auto soll den Motor über den Konstruktor injiziert bekommen, da verschiede Modelle andere Motoren haben können. Das gleiche gilt beim Fahrer.

# 4.6 You ain’t gonna need it

Robert Martin meinte aus seiner Erfahrung heraus: „Jedes Mal als ich etwas für die Zukunft vorsah, das was der Kunde möglicherweise brauchen würde, wollte der Kunde in der Realität immer etwas Anderes.“

# 4.7 Nutze source code konventionen

Zu **Folie 41.**

Vor allem sollte darauf geachtet werden, dass File Formatierungen separat zu den Features vorgenommen und eingecheckt werden, da dadurch recht schwer nachvollzogen werden kann was die tatsächliche Feature-Änderung war.

# Kapitel 5 Refactoring

Zu dem Thema Videos <https://www.youtube.com/user/cyber0ne/featured>

# 5.3 Code Smells

Zu **Folie 22. Lange Methoden**

Hier kommen die Kode Beispiele aus dem Vortrag „cleancode - Rules from Robert C. Martin's book Clean Code“ Folien 26-32.

Und noch eine Sache laut Bob Martin eine Funktion ist nur dann klein genug, wenn man aus ihr keine weitere Funktion extrahieren kann. Dann macht sich auch sicherlich nur eine Sache.

Zeige auch die Beispiele Seite 63-66 vom „Clean Code Buch“. Und erwähne nochmal das Thema mit dem Journalismus und Ehrlichkeit dem Leser gegenüber.

Tipp:

Wenn Methoden extrahiert kommt es oft vor das die DIE sagt es sei unmöglich, da mehrere Variablen wo anders benötigt werden.

Macht die Variablen erstmal global, dann funktioniert erstmal die Extraktion. Und im 2. Schritt könnte vielleicht eine Klasse daraus entstehen. Dabei werden die globalen Variablen zu Member.

Zu **Folie 23. Lange Parameterliste**

Wenn Ihr eine Funktion extrahiert kommt es oft vor, dass es zu langen Parameterlisten führt. Wenn aber gemeinsame Argumente in einer Funktion beziehungsweise in der Aufrufhierarchie benötigt werden klingt das nach einem weiteren Objekt. SRP.

Zu **Folie 29. Faule Klassen**

Zu wenig Inhalt ist nicht das Problem denkt an den hohen Zusammenhalt (high cohesion).

Zu **Folie 30. Lösungsausbreitung**

Die Erfahrung zeigt, dass eher zu wenig Klassen vorhanden sind als zu viele. Ich würde eher sagen, dass diese Klassen in sinnvolle Packages verpackt werden sollen. Und ein gemeinsames Interface nach außen anbieten sollen.

Zu **Folie 31. Unpassender Name**

Die 3. Fragen zu Namensgebung

Warum gibt es die Methode? Was tut die Methode? Und Wie die Methode verwendet wird?

Falls nicht schon gezeigt:

Hier kommen die Beispiele aus dem Vortrag „cleancode - Rules from Robert C. Martin's book Clean Code“ oder „Diapositiva 1“ Folien 11-25

Zu **Folie 32. Toter Code**

Seien Sie aber vorsichtig es kann nämlich sein, wenn der Kode in einer gemeinsamen Library liegt, dass Sie ein anderes Projekt damit zerschießen.

Zu **Folie 33. Übung Smells**

Gelassener reden!!!

Smells:

Klasse Address:

* Die Attribute sind public  Information hiding, Geheimnisprinzip
* Die Write Methode wird nicht verwendet  Toter Kode

Abgesehen davon ist Write kein passender Name besser wäre Print  POLS

* In der Read Methode Konsolen Lese- und Schreibzugriffe  SLAP, SRP, SOC

Besser wäre an der Stelle die Konsolenzugriffe in der Main zu machen damit die Klasse Address keine Abhängigkeiten hat und sogar Konstant  imutable gemacht werden kann. Die Attribute werden über den Konstruktor einmal gesetzt und nie mehr verändert.

Oder eine Referenz auf das Intraface ILoggeable über den Konstruktor zu übergeben. Die konkrete Klasse Console die das Interface implementiert wird injiziert.

Oder „Inversion of Control“ Muster einsetzen.

Oder eine statische Klasse Console zu implementieren  schlechter testbar und wartbar.

* Die Read Methode ist zu groß und hat duplizierten Kode.
* Zusätzlich sollte bei den Operatoren als Parameter eine konstante Referenz übergeben werden.  Performance und Sicherheit.

Klasse AddressManagement:

* Sollte keine Logging Abhängigkeit haben.
* ShowAddress Methode sollte Write(print) Methode der Klasse Address ersetzen  Tell don’t ask und die WirteAllAddresses Methode sollte in PrintInfo umbenannt werden ist deutlicher 🡪 POLS, NAMING.
* SortAddresses sollte in SortByZip oder SortByZipAscending umbenannt werden.
* Die Methode SortByZipAscending am besten privat und nicht in der Main aufrufen lassen, sondern lieber zusammen mit der printInfo Methode in eine Wrapper Methode SortByZipAscendingAndPrint aufrufen 🡪 Tell Don´t Ask.
* Der Konstruktor sollte explizit markiert werden.
* Die for loop in der WriteAllAddresses(printInfo) Methode sollte eine const auto & holen.

Die Main:

* Die While Loop in der Main und in der Read Methode der Klasse Address haben duplizierten Kode.  DRY
* Ist zu groß und hat verschiedene Abstraktionsschichten  Verletzung von SLAP.
* Vielleicht eine neue Klasse Application erstellen.
* Die invalid(numberOfAddresses) ist schöner zu lesen als z.B. if(!numberOfAddresses) oder if(numberOfAddresses == 0) vorallem kann man, wenn man den Kode in mehreren stellen verwendet hätte, die Methode einmal anpassen und die Sache wäre erledigt.

Gelassener reden!!!

Zu **Folie 43.** Methoden zusammenstellen

Sprechende Namen kann man auch besser suchen. Src oder Dest lieber ausschreiben.

Zu **Folie 45.** Inline Method

Wenn allerdings so etwas auftaucht „value > maxNumber“ und es auch noch in mehreren Stellen im Kode vorkommt, kann eine Inline Methode durchaus Sinn ergeben.

An der Stelle würde ich auch ein Null Object Pattern bevorzugen um in Zukunft nicht mehr nach ungleich „nullptr“ zu prüfen.

Achtung Fehler IsRefenceNotNull Methode sollte IsNotNull heißen.

Zu **Folie 47.** Entferne Zuweisung an Parametervariable

Deshalb übergebe ich weitestgehend konstante Parameter an eine Funktion. Kann zu Verletzung von POLS führen und zu unangenehmen Bugs aus Unachtsamkeit -> das Beispiel mit der for-Schleife.

Zu **Folie 49.** Entferne Zuweisung an Parametervariable

In der „If Abfrage“ wenn dort aus Versehen balance = 20000 stünde würde der Compiler meckern da die Variable „const“ ist.

Zu **Folie 56.** Inline Temp

Z.B.

bool isLuxeryItem() const

{

auto basePrice = currentOrder.basePrice();

return basePrice > 1000;

}

Manchmal macht es durchaus Sinn eine Zwischenvariable zu haben. Vor allem fürs Debuggen. Das ist jedoch Geschmackssache.

bool isLuxeryItem() const

{

return currentOrder.basePrice() > 1000;

}

Zu **Folie 57.** Übung Composing Methods

Hier geht es vor allen um die Main aber auch ein bisschen in der „counter.cpp“.

Counter:

1. Lösche alles was vom Kompiler automatisch zur Verfügung gestellt wird.
2. Überladen operatoren vereinfachen.
3. Setter, Getter und Count und operatoren inline machen -> counter.cpp löschen.

Main:

1. Lokale variablen nach einander löschen nun sieht man sofort den duplizierten Kode.
2. Die ganzen For-Schleifen in einzelne Methode verschieben.
3. Neue Klasse CounterManagement erstellen?
4. Statt For-Schleifen vielleicht aus der Algorithms Bibliothek etwas anwenden?

Bei der Lösung finde ich die Magic Numbers in der Main nicht gut.

Zu **Folie 70.** Übung Moving Features between Objects

Könnte man skippen da es so ähnlich ist wie die Übung Smells

Address:

1. Die Attribute sind public  Information hiding, Geheimnisprinzip
2. Neue Klasse Console mit statischen Methoden Read, Write  SOC, SRP, SLAP
3. Vielleicht wäre hier das Pattern IoC besser geeignet als statische Klasse Console

Wie bei der SOLID/DIP Übung

AddressManagement:

1. Verschiebe die ReadAddress Methode in die Klasse Address  Move Method
2. Verschiebe die Ausgaben in die Console Klasse.

Gelassener reden!!!

Zu **Folie 77.** Kapsle Attribut

Ich würde an der Stelle keine Getter und Setter verwenden, sondern lieber eine Methode wie CalculateBalance(const double amout)  Tell don’t ask

Zu **Folie 80.** Kapsle eigene Attributzugriffe

Dieses Prinzip sollte man nicht zu ernst nehmen. Hier muss aufgepasst werden, wenn abgeleitete überschriebene Methoden im Einsatz sind, die mittels Polymorphie aufgerufen werden kann es zu Überraschungen kommen. Auf jeden Fall Unit Testen.

Zu **Folie 100.** Geschachtelte Bedingungen durch Wächter ersetzen

Mehrere „returns“ sind vor allem beim Debuggen nervig. Methode Extrahieren von der IDE funktioniert hier oft gar nicht. Hier kann auch das Proxy Pattern eingesetzt werden. Wird in der nächsten Übung in meiner Lösung gezeigt.

Zu **Folie 106. Null-Objekt einführen**

Beim Null Object sollte meiner Meinung nach die Ausgabe "Occasional Customer" kommen.

Zu **Folie 109.** Übung Simplify

Bei MyTries SimpliProxyPattern:

Proxy Pattern wird dazu verwendet, um den „Zugang“ zum original Objekt bei Bedarf zu verhindern.

Proxy ist so eine Art „Türsteher“. In diesem Beispiel wird er dazu benutzt, um zu prüfen, ob der Nenner 0 ist. Bei diesem Beispiel ist so eine Zwischenklasse „fast“ nutzlos. Es könnte aber zukünftig durchaus sein, dass es in der Mathematik die Lösung für die Division durch 0 gefunden wird.

In diesem Fall wäre es nicht nötig das Objekt DivisionOparation anzupassen. Man würde lediglich statt dem Objekt DivisionOparationNullChecked das Objekt DivisionOparation der Calculate Methode übergeben. Und die DivisionOparationNullChecked Klasse löschen. Alles andere wäre gleich geblieben.

Der Unterschied zum Decorator Pattern ist, dass der Proxy das Objekt das es überwacht selbst hält (Siehe UML1 Diagramm)

Der Dekorator kann zur Laufzeit neue Funktionalität den Objekten zuweisen.

Bei MyTries SimplWithOperationFactory:

Eine weitere Möglichkeit wäre eine Referenze auf das Interface IOparationable über den Konstruktor zu übergeben. Dies ermöglicht eine Überwachung von allen Implementierungen des Interfaces inklusive sich selbst (Siehe UML2 Diagramm).

Bei Solution Ordner:

Eine andere Lösung mittels einer Lookup-Tabelle. Das finde ich nicht so schön, da bei jedem weiteren Eintrag in die Map die Calculator Klassen neu gebaut werden muss.  kann brechen.

Zu **Folie 115.**

Wir haben in Team beschlossen keine „Exceptions“ zu benutzen stattdessen haben wir einen Error Klasse mit eindeutigen zentralen Error Kode Enums und Mechanismen.

Zu **Folie 118.**

Hier könnte man auch das Proxy oder Decoratorator Pattern angewendet werden. Dann wäre die Account Klasse ein Komponent der Checking/Saving Account Klasse

Zu **Folie 125.**

Man sollte den Scope einer Variablen so klein halten wie möglich  wird bei Cyber Security geschult, da hier das „Reverse Engineering“ dann schwieriger ist.

Zu **Folie 135.** Split Loops

Der Variablenname „avarage\_mileage“ stimmt erst nach der for-Schleife. Zudem könnte die for-Schleife durch std::accumulate aus der STL im Header algorithms ersetzt werden.

Zu **Folie 139.** Übung Other

Könnte übersprungen werden hier wird nur eine Rekursion mittels for-Schleife ersetzt.

Zeige die MyTries Lösung die Recursive Methode wurde durch eine sort Methode aus der Standart Lib ersetzt. Im header algorithm gibt es schon viele coole Sachen schaut da mal rein.

2 Hinweise

2.1 Pfadfinderregel

**The Boy Scout Rule** aus dem Vortrag „vladimirromanov-cleancode“ Folie 7.