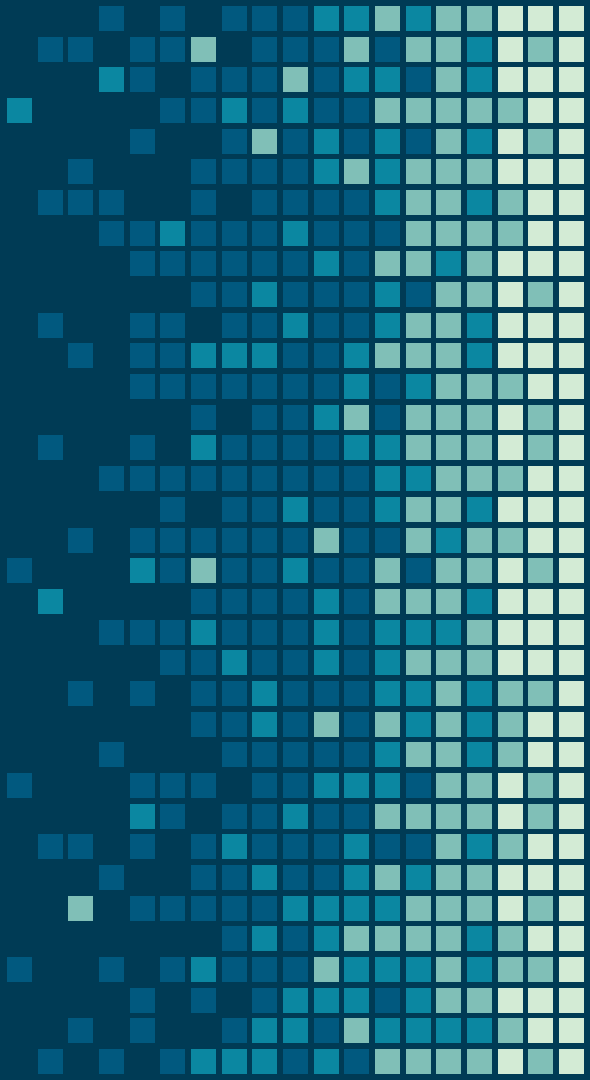
Refactoring mit GitHub Copilot



Hi!

Ich bin Caglar Özdemir

im digitalen Dschungel &
in Aachen zuhause

Eisbrecher

- Wer bist du?
- Was machst du?
- Was sind deine Erwartungen?
- Was ist deine Lieblings-IDE?

1.

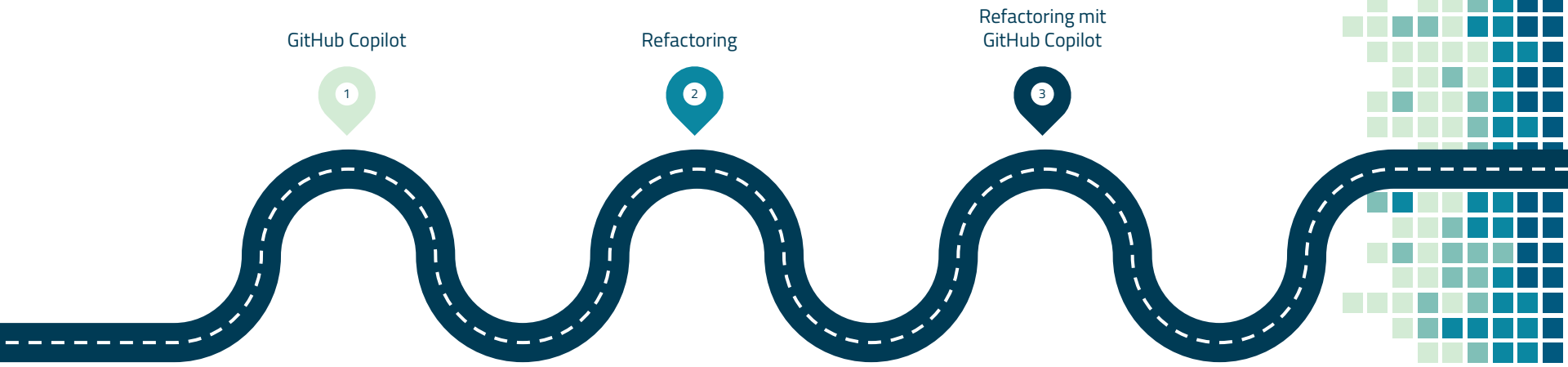
Einleitung

Wichtigkeit von sauberem Code und kontinuierlicher Verbesserung

Gliederung

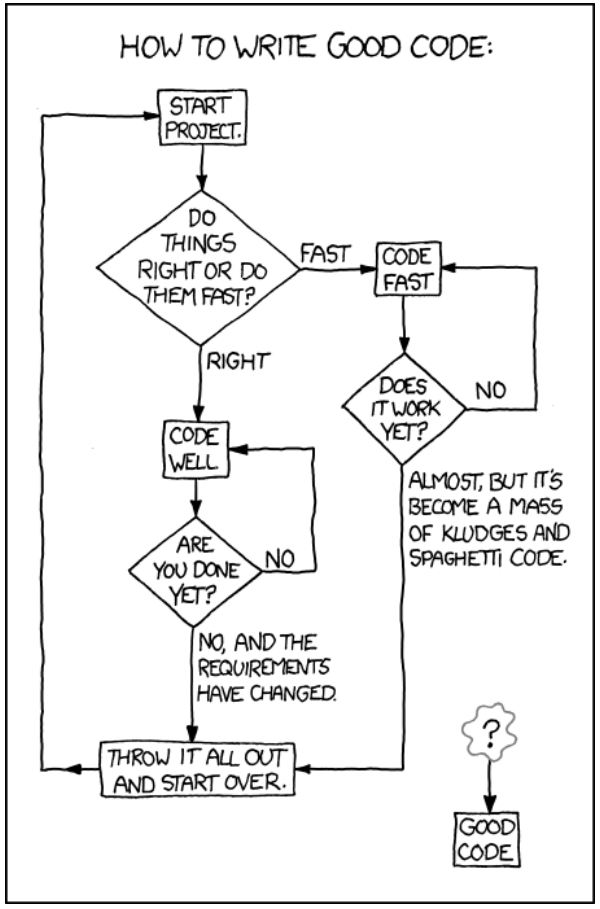
- Was ist GitHub Copilot?
- Was sind Prompts ?
- Wie arbeite ich mit GitHub Copilot auf Visual Studio?
- Prompten ein bisschen
- Was ist Refactoring?
- Warum ist Refactoring notwendig?
- Wann ist Refactoring notwendig?
- Die "schlechten Gerüche" im Code
- Praxis Beispiele
- Refactoring mit Github Copilot
- Aufräumen von einigen Repos

ROADMAP


GitHub Copilot
1
Refactoring
2
Refactoring mit
GitHub Copilot
3

Schulungs Zeitplan

	MONDAY	TUESDAY	WEDNESDAY
9:00 – 10:00	Einleitung	Wiederholung	-
10:00 - 10:45	Was ist GitHub Copilot?	Die Schlechten Gerüche im Code	-
11:00 – 12:00	Was ist GitHub Copilot?	Refactoring mit GitHub Copilot	-
12:00 - 13:00	✓ Pause	✓ Pause	-
13:00 - 14:15	Was ist Refactoring ?	Refactoringsaufgaben	-
14:30 - 16:00	Die schlechten Gerüche im Code	Q&A	-


HOW TO WRITE GOOD CODE:
START PROJECT.
DO THINGS RIGHT OR DO THEM FAST?
FAST
CODE FAST
DOES IT WORK YET?
NO
ALMOST, BUT IT'S BECOME A MASS OF KLUDGES AND SPAGHETTI CODE.
RIGHT
CODE WELL
ARE YOU DONE YET?
NO
NO, AND THE REQUIREMENTS HAVE CHANGED.
THROW IT ALL OUT AND START OVER.
?
GOOD CODE

1.

Was ist GitHub Copilot?

AI Model und unterschied zu anderen Modellen

GitHub Copilot

- **Beschreibung**: GitHub Copilot ist ein KI-gestütztes Coding-Tool, das Entwicklern Vorschläge und Codevervollständigungen in Echtzeit anbietet.
- **Funktion**: Es erkennt den Kontext des Codes, bietet relevante Code-Snippets und erleichtert repetitive Aufgaben.
- **Unterstützte Sprachen**: Copilot unterstützt mehrere Programmiersprachen wie Python, JavaScript, TypeScript, Ruby, Go, und C#.

Technologie hinter GitHub Copilot

- **Art der KI**: GitHub Copilot ist eine generative KI, die Inhalte — in diesem Fall Code — eigenständig erstellt, basierend auf einer riesigen Datenbasis und der Analyse des aktuellen Codes.
- **KI-Modell**: Basierend auf OpenAIs Codex, einer spezialisierten Variante von GPT-3, die auf Codeverständnis und -generierung ausgelegt ist.
- **Trainingsdaten**: Codex wurde anhand von Milliarden Codezeilen aus öffentlich zugänglichen Repositories auf GitHub und anderen Quellen trainiert.
- **Funktionsweise**: Das Modell analysiert den Code, den ein Entwickler schreibt, und nutzt seine Datenbasis, um passende nächste Schritte vorzuschlagen.

Weiterentwicklung und Verfügbarkeit

- **Erweiterung durch Feedback**: GitHub Copilot verbessert sich stetig durch Nutzer-Feedback, um passendere Vorschläge zu liefern.
- **Verfügbare Entwicklungsumgebungen (IDEs)**: Copilot ist in Visual Studio Code, Visual Studio, JetBrains IDEs und Neovim integriert.
- **Einschränkungen und Anpassungen**: Copilot kann durch Unternehmensrichtlinien angepasst werden, um nur bestimmten Code zu verwenden

Fallstudie

- Eine Studie **der Agentur** Harness Software Engineering Insights SEI über die Auswirkungen von GitHub Copilot auf die Produktivität von Entwicklern, die sich auf die Anzahl der Pull Requests (PRs) und die Zykluszeit konzentrierte, ergab, dass die Integration von GitHub Copilot zu **einem Anstieg der Pull Requests um 10,6 % und einer Verringerung der Zykluszeit um 3,5 Stunden** führt. Copilot rationalisiert also tatsächlich die Arbeitsabläufe von Entwicklern und verbessert die Zusammenarbeit zwischen Teams.

Potenzielle Risiken und Qualitätsprobleme

- **Nicht immer perfekte Lösungen**:
  - Copilot generiert Vorschläge, die nicht immer korrekt oder optimal sind.
  - Qualität hängt stark vom Code-Kontext und den verwendeten Trainingsdaten ab.
- **Risiko der "Selbstverstärkung"**:
  - Copilot-generierte Vorschläge können ungeprüft in Repositories eingecheckt werden
  - Diese Vorschläge könnten später wieder als Trainingsdaten dienen und langfristig die Qualität der KI beeinträchtigen.
- **Verantwortungsbewusste Nutzung**:
  - Copilot als unterstützendes Werkzeug nutzen
  - generierte Codevorschläge kritisch zu prüfen

Was sind Prompts ?

- **Definition**: Ein "Prompt" ist eine Eingabe oder Anweisung, die dem KI-Modell eine Richtung gibt, um darauf basierend eine Antwort oder einen Codevorschlag zu generieren.

- **Einsatz in GitHub Copilot**: Prompts können aus Kommentaren, Fragen oder kurzen Anweisungen bestehen und helfen Copilot, den gewünschten Code zu verstehen und zu generieren.

- **Beispiele**:
- Anweisung: "Erstelle eine Funktion, die die Fibonacci-Zahlen berechnet."
- Frage: "Wie implementiere ich eine Schleife in C#?"

Wie funktionieren Prompts ?

- **Funktionsweise**: Copilot interpretiert Prompts, erkennt Muster und generiert Codevorschläge, die zur Eingabe passen.
- **Kontextverständnis**: Der Prompt gibt Copilot den Kontext, um den richtigen Code zu erstellen. Wenn der Kontext klar ist, liefert Copilot oft präzisere Vorschläge.
- **Interaktive Anpassung**: Wenn ein Vorschlag nicht ideal ist, kann der Prompt angepasst werden, um eine bessere Lösung zu erhalten.

Gute & schlechte Prompts

- **Schlechter Prompt**:
  - „Schreibe eine Schleife in C#, die über eine Liste geht“
- **Guter Prompt**:
  - „Schreibe in C# eine **foreach**-Schleife, die über eine Liste von **Person**-Objekten iteriert und den Namen jedes Objekts nur dann in der Konsole ausgibt, wenn das Alter größer als 18 ist.“

Gute & schlechte Prompts

- **Schlechter Prompt**:
  - „Erstelle eine Funktion, die die größten Zahlen filtert.“
- **Guter Prompt**:
  - „Schreibe in C# eine Funktion GetTopElements, die eine Liste von Integern akzeptiert. Die Funktion soll das Array nach Größe sortieren und die **count** größten Elemente als neues Array zurückgeben.“

GitHub Copilot in Visual Studio einrichten

- **Installation**: Copilot kann über den Visual Studio Marketplace als Erweiterung installiert werden.
- **Aktivierung**: Nach der Installation muss Copilot in den Visual Studio-Einstellungen aktiviert und die GitHub-Anmeldung abgeschlossen werden.
- **Anpassung der Einstellungen**: Die Copilot-Einstellungen können angepasst werden, um Vorschlagsdichte, Sprache und Nutzungshäufigkeit zu steuern.

Interaktionsarten mit GitHub Copilot

- **Inline-Vorschläge**: Copilot bietet kontextbasierte Inline-Vervollständigungen direkt im Editor an, basierend auf dem aktuell geschriebenen Code.
- **Mehrere Vorschläge durchblättern**: Mit Alt +] oder Alt + [kann man durch alternative Vorschläge blättern, wenn mehrere Lösungen verfügbar sind.
- **Prompts in Kommentaren**: Kommentare, die als Prompts formuliert sind, können verwendet werden, um Copilot gezielt anzuweisen, z. B.
- // Schreibe eine Funktion zur Berechnung der Faktoriale.

Features von GitHub Copilot in Visual Studio

- **Automatische Vervollständigung**: Copilot schlägt komplette Codezeilen oder -blöcke vor, die den geschriebenen Code ergänzen.
- **Funktionserstellung aus Kommentaren**: Kommentare können als Anweisungen dienen, um spezifische Funktionen, wie z. B. Algorithmen oder Datenverarbeitungsfunktionen, zu erstellen.
- **Testgenerierung**: Copilot kann basierend auf bestehenden Funktionen passende Unit-Tests generieren und damit die Testabdeckung erhöhen.
- **Codeerklärung**: Copilot kann Code kommentieren und Erklärungen hinzufügen, um den Zweck von komplexem Code zu erläutern.

Übung 1

- **Erstelle ein Schere-Stein-Papier spiel auf der Konsole**
- **Versuche ausschließlich nur Prompts zu verwenden**

Übung 2

- **Erstelle Geldautomat simulator auf der Konsole**
- **Versuche ausschließlich nur Prompts zu verwenden**

Übung 3

- **Bremswegberechnung**
- **Versuche ausschließlich nur Prompts zu verwenden**

2.

Was ist Refactoring?

Unterschied zwischen Refactoring und anderen Aktivitäten

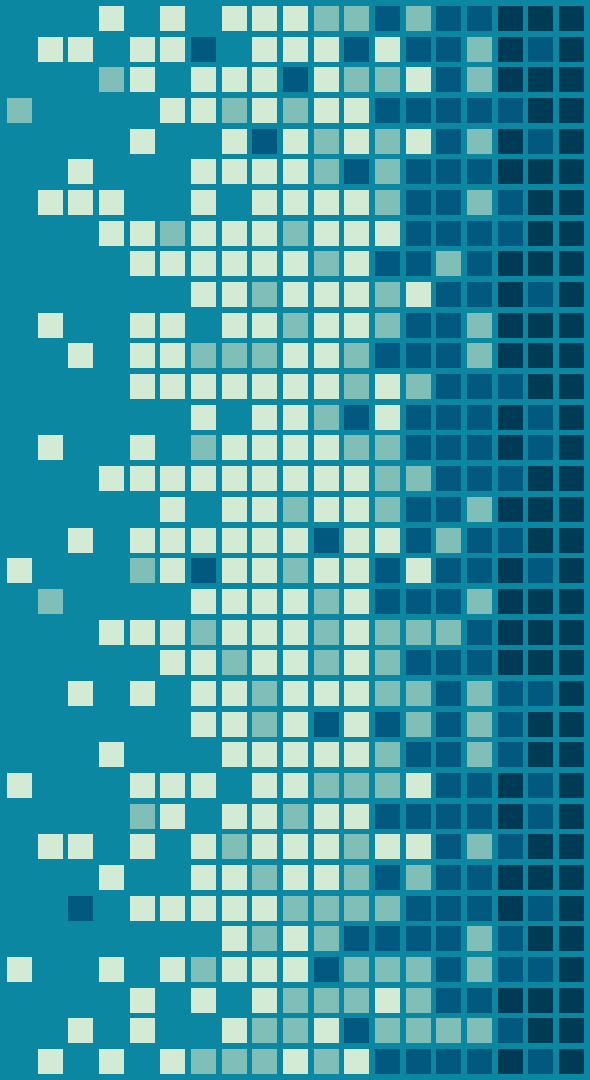
Martin Fowler

Softwareentwickler / Autor

Experte für
- Softwarearchitektur
- Design Patterns
- Agile Methoden

Mitbegründer des Manifests für Agile Softwareentwicklung

> *Jeder (Narr) kann Code schreiben, den ein Computer versteht.*
>
> *Gute Programmierer schreiben Code, den Menschen verstehen können.*



> *Jeder Narr kann Code schreiben, den ein Computer versteht. Gute Programmierer schreiben Code, den Menschen verstehen können.*

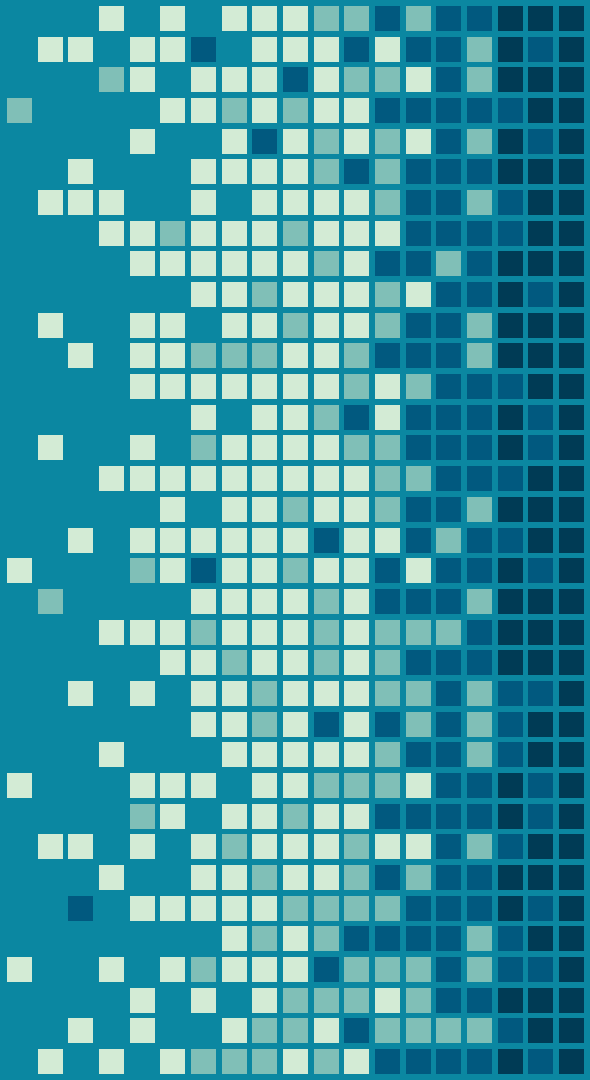
```
public double CalculateRectangleArea(double w, double h)
{
        double a = 0;

        for (int i = 0; i < h; i++)
        {
            a += h;
        }
        return a;
}
```

REFACTORING DEFINIEREN

- Definition:
  Strukturänderung von Software, um sie verständlicher und modifizierbarer zu machen, ohne das beobachtbare Verhalten zu ändern

- Prozess von kleinen, Verhalten-erhaltenden Schritten

> *Beim Refactoring werden die Programme in kleinen Schritten geändert, so dass man im Falle eines Fehlers leicht herausfinden kann, wo der Fehler liegt.*



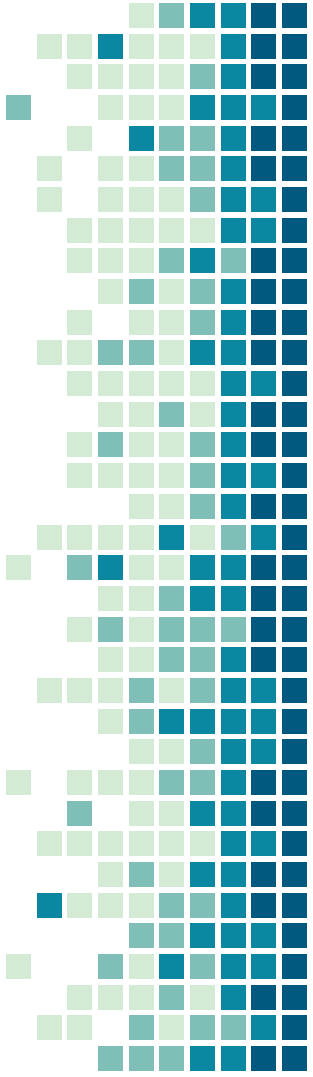
ZWEI HÜTE

Hinzufügen von Funktionalität

- Keine Änderungen am bestehenden Code
- Nur Hinzufügen von neuen Fähigkeiten und Tests

Refactoring

- Nur Code Restrukturierung
- Keine Neue Funktionalität

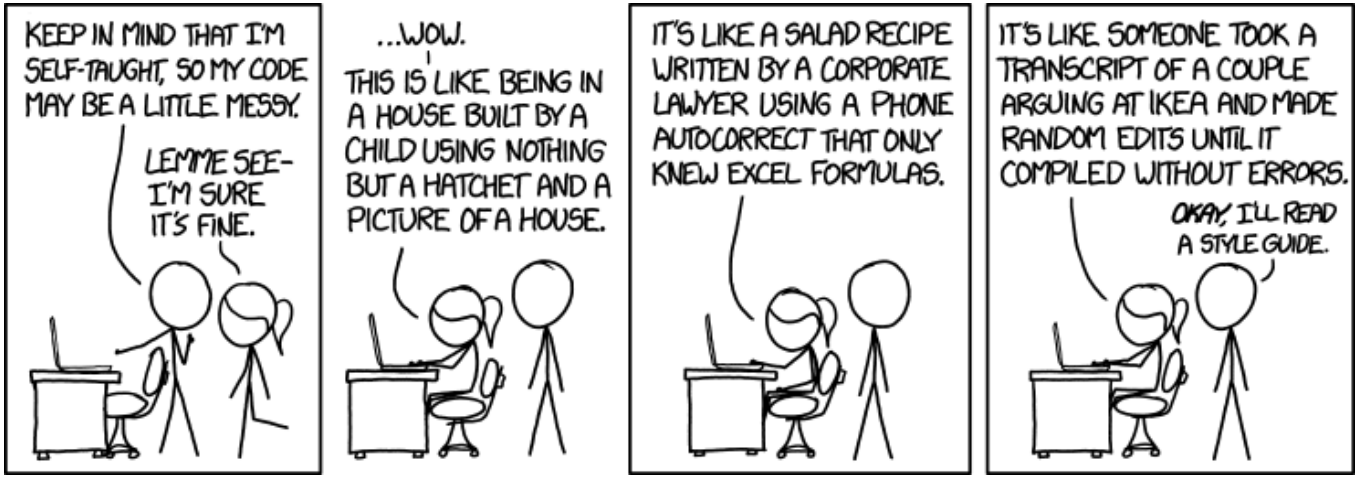


> *Wenn Sie einem Programm eine Funktion hinzufügen müssen, der Code aber nicht zweckmäßig strukturiert ist, sollten Sie das Programm zunächst überarbeiten, um das Hinzufügen der Funktion zu erleichtern, und dann die Funktion hinzufügen.*

3.

Warum ist Refactoring notwendig?

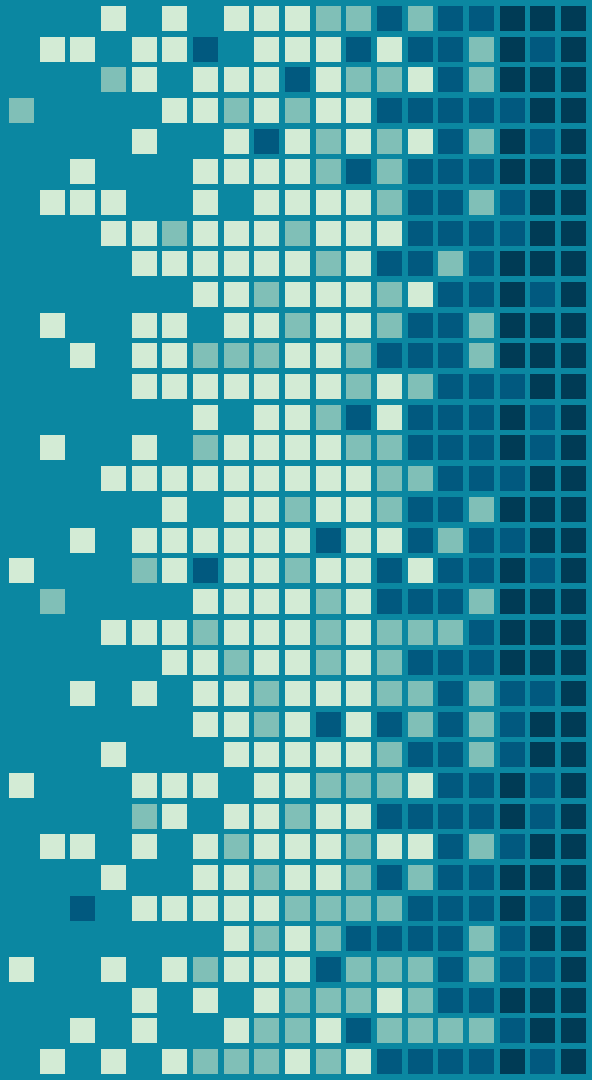
Zunahme von Komplexität & Langsamerer Entwicklungsgeschwindigkeit


KEEP IN MIND THAT I'M SELF-TAUGHT, SO MY CODE MAY BE A LITTLE MESSY.
LEMME SEE- I'M SURE IT'S FINE.
...WOW.
THIS IS LIKE BEING IN A HOUSE BUILT BY A CHILD USING NOTHING BUT A HATCHET AND A PICTURE OF A HOUSE.
IT'S LIKE A SALAD RECIPE WRITTEN BY A CORPORATE LAWYER USING A PHONE AUTOCORRECT THAT ONLY KNEW EXCEL FORMULAS.
IT'S LIKE SOMEONE TOOK A TRANSCRIPT OF A COUPLE ARGUING AT IKEA AND MADE RANDOM EDITS UNTIL IT COMPILED WITHOUT ERRORS.
OKAY, I'LL READ A STYLE GUIDE.

Refactoring verbessert das Design von Software

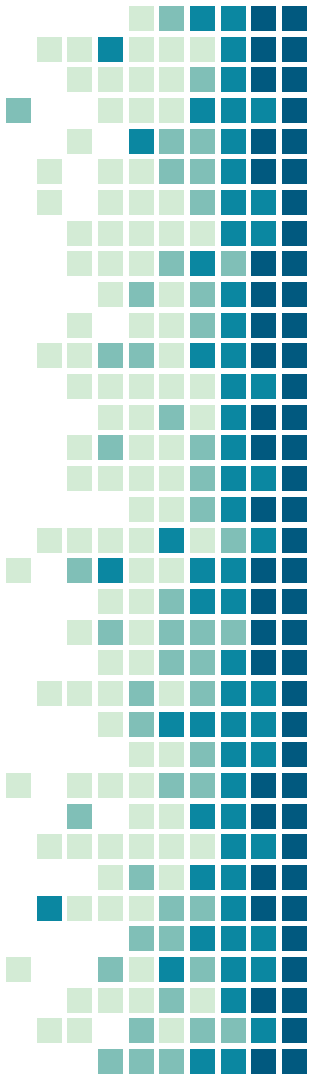
- Ohne Refactoring verfällt die interne Softwarearchitektur
- Kurzfristige Code-Änderungen ohne Verständnis führen zum Verlust der Struktur
- Regelmäßiges Refactoring erhält die Code-Qualität

> *Befolgen Sie beim Programmieren die Camping-Regel: Verlassen Sie die Codebasis immer ordentlicher, als Sie sie vorgefunden haben.*



Refactoring macht Software leichter verständlich

- Programmieren ist ein Dialog mit dem Computer, aber auch mit zukünftigen Entwicklern
- Während des Programmierens geht es darum, genau das zu sagen, was ich will
- Refactoring macht den Code lesbarer und verständlicher



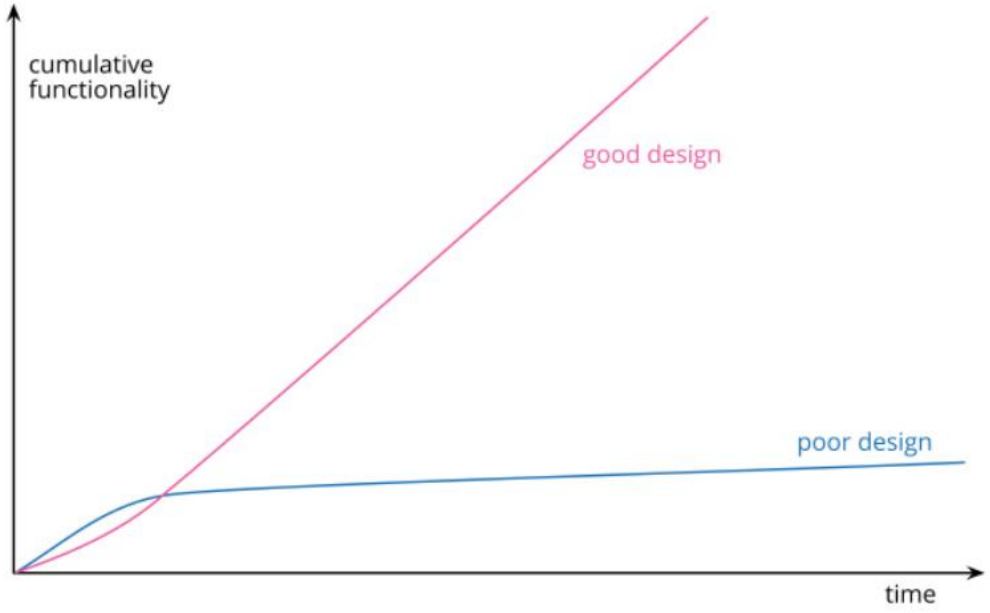
Refactoring hilft mir bei der Fehlersuche

- Verständnis des Codes hilft auch beim Erkennen von Fehlern.
- Refactoring fördert ein tieferes Verständnis des Codes und hilft beim Aufdecken von Bugs.
- Refactoring unterstützt gute Programmiergewohnheiten

Refactoring hilft mir, schneller zu programmieren

- Refactoring ermöglicht schnellere Code-Entwicklung
- Bessere interne Gestaltung, Lesbarkeit und Reduzierung von Fehlern verbessern die Qualität

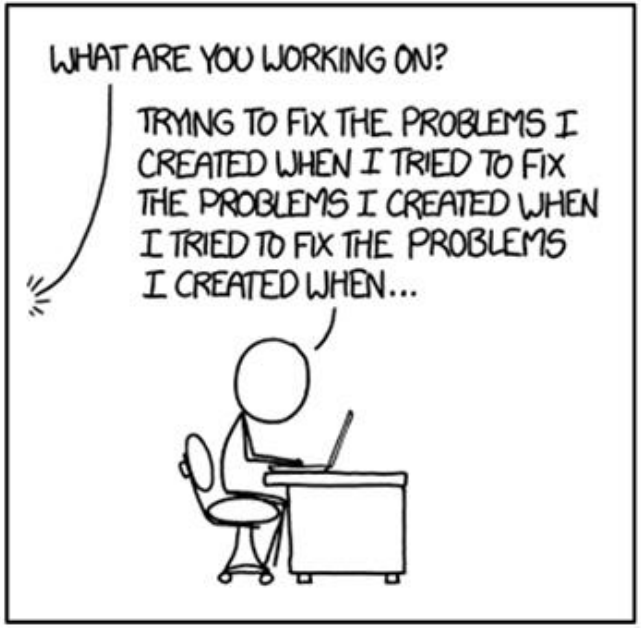
Design-Ausdauer-Hypothese



4.

Wann ist Refactoring notwendig?

Zunahme von Komplexität &
Langsamerer Entwicklungsgeschwindigkeit


WHAT ARE YOU WORKING ON?
TRYING TO FIX THE PROBLEMS I CREATED WHEN I TRIED TO FIX THE PROBLEMS I CREATED WHEN I TRIED TO FIX THE PROBLEMS I CREATED WHEN...

Vorbereitendes Refactoring

- Die beste Zeit zum Refactoring ist kurz vor dem Hinzufügen einer neuen Funktion zum Code
- Durch Refactoring kann ich vorhandenen Code strukturieren, um zukünftige Arbeit zu erleichtern

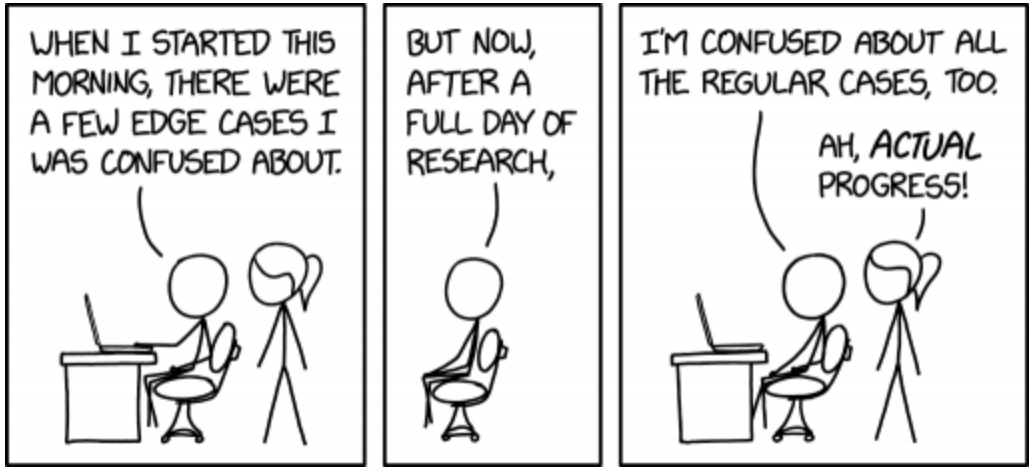
Umfassendes Refactoring

- Bevor ich Code ändere, muss ich verstehen, was er tut
- Refactoring hilft, das Verständnis des Codes sofort offensichtlich zu machen

Verständnis Refactoring

- Unnötig komplexe Logik oder fast identische Funktionen können durch eine parameterisierte Funktion ersetzt werden

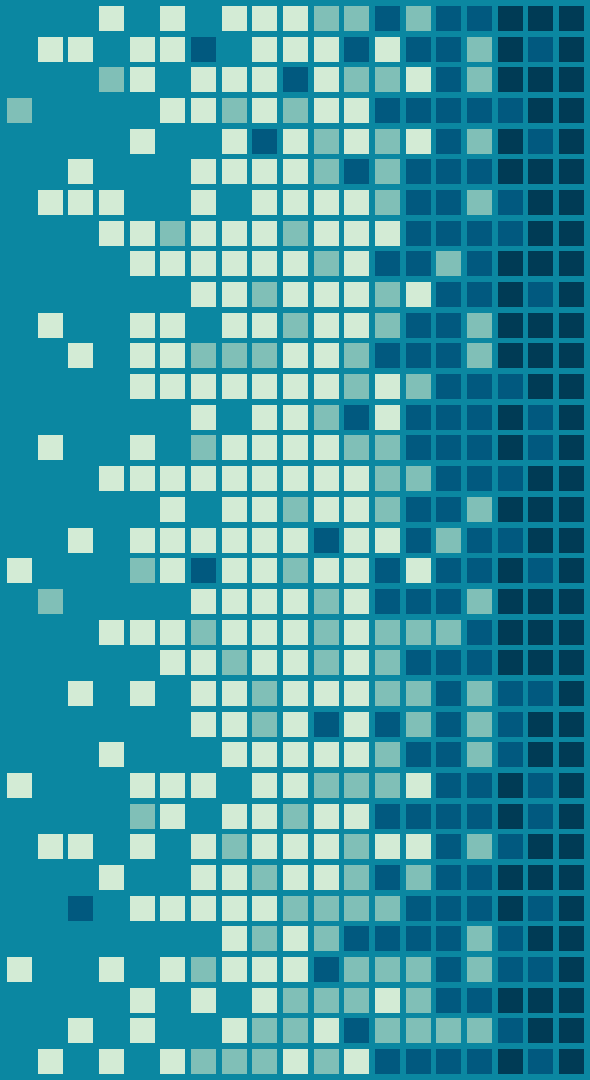
- Wenn möglich, verbessere ich den Code sofort. Andernfalls mache ich eine Notiz und behebe es später


WHEN I STARTED THIS MORNING, THERE WERE A FEW EDGE CASES I WAS CONFUSED ABOUT.
BUT NOW, AFTER A FULL DAY OF RESEARCH,
I'M CONFUSED ABOUT ALL THE REGULAR CASES, TOO.
AH, ACTUAL PROGRESS!

Geplantes und opportunistisches Refactoring

- Guter Code erfordert kontinuierliches Refactoring, nicht nur Fehlerkorrekturen
- Softwareentwicklung ist iterativ, der Code wird kontinuierlich angepasst

> *Man muss refaktorisieren, wenn man auf hässlichen Code stößt - aber auch exzellenter Code braucht eine Menge Refaktorierung.*

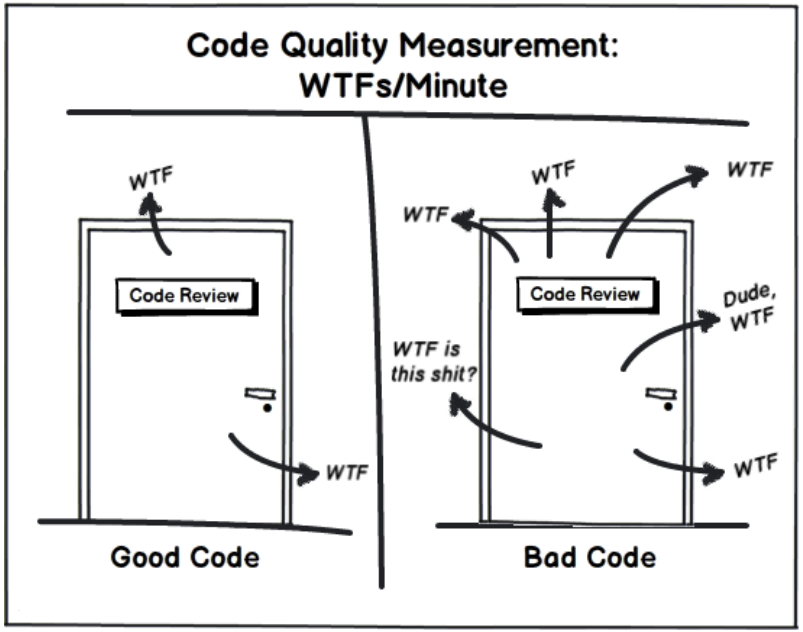


Langfristiges Refactoring

- Meisten Refactoring-Aufgaben dauern nur wenige Minuten bis Stunden
- Größere Refactoring-Aufwände können Wochen dauern
- Kleine Schritte im Refactoring-Prozess ermöglichen es, das System schrittweise zu verbessern, ohne es zu brechen

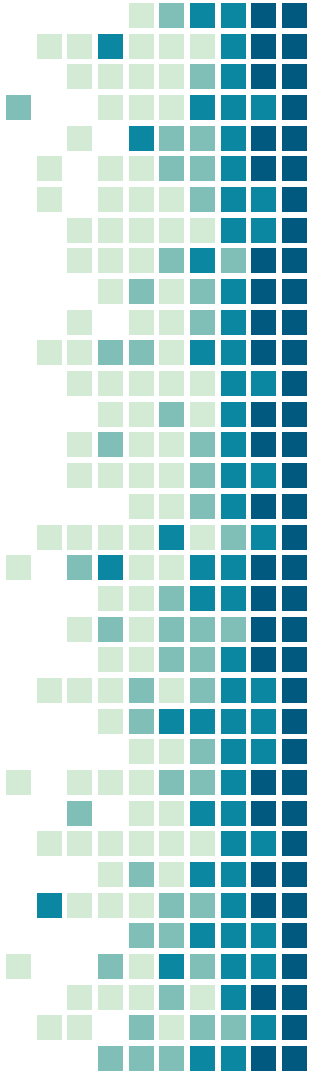
Refactoring im Rahmen eines Code-Review

- Regelmäßige Code-Reviews sind essentiell für Wissenstransfer und klaren Code
- Code-Reviews ermöglichen erfahrenen Entwicklern, ihr Wissen an weniger Erfahrene weiterzugeben
- Direkte Zusammenarbeit zwischen Autor und Reviewer ist oft effektiver als isolierte Begutachtung


Code Quality Measurement:
WTFs/Minute
WTF
Code Review
WTF
Good Code
WTF
WTF
WTF
WTF
Code Review
Dude,
WTF
WTF is
this shit?
WTF
Bad Code

Was sage ich meinem Vorgesetzten?

- Fehlerkorrektur oder unproduktive Arbeit
- Refactoring nicht explizit erwähnen
- Design-Ausdauer-Hypothese



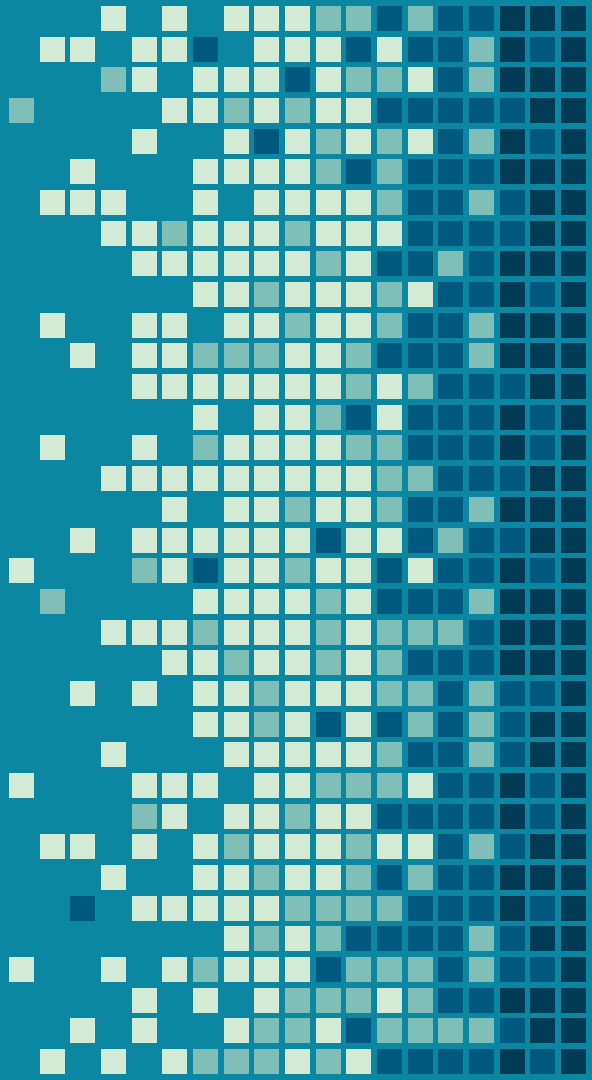
Wann sollte ich nicht refaktorisieren?

- Unordentlichen Code welcher nicht geändert werden muss
- Entscheidung zwischen Refactoring und Neuschreiben

Verlangsamung neuer Funktionen

- wirtschaftlicher Nutzen
- zu wenig Refactoring oft problematischer

> *Der Zweck des Refactoring besteht darin, schneller zu programmieren und mit weniger Aufwand mehr Wert zu schaffen.*



Code Eigentum

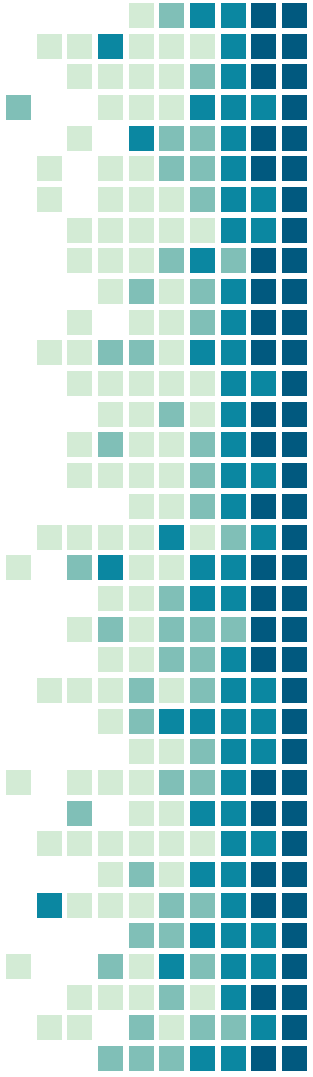
- Beziehungen zu anderen Teilen des Systems
- Änderungen an der Schnittstelle das Verhalten von Clients beeinflussen

Branches

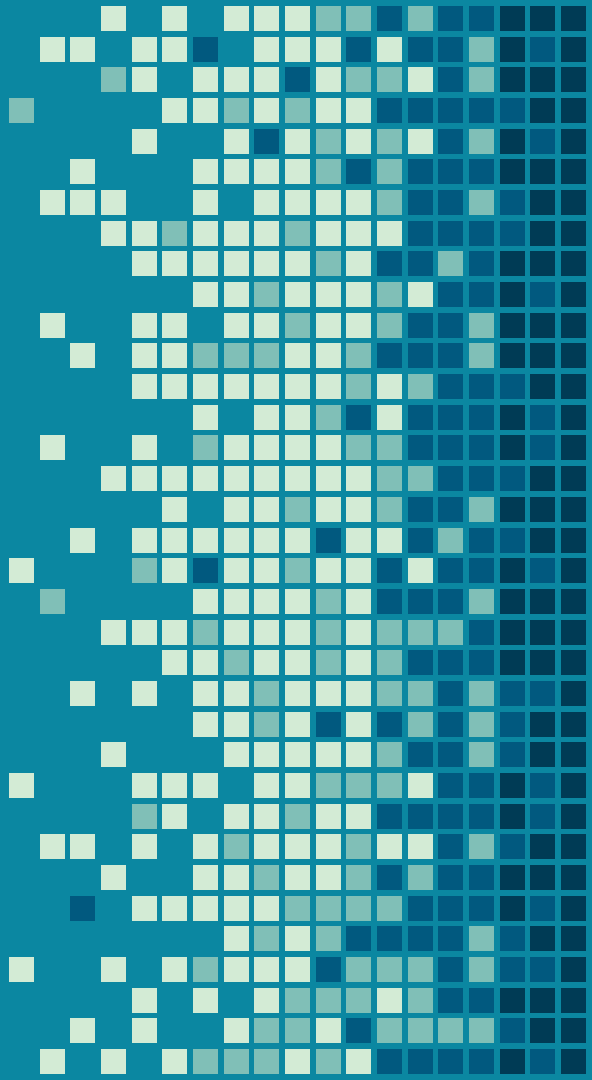
- Feature Branches
- Continuous Integration (CI)
- Merges

Testing

- Schnelles erkennen von Fehlern
- Selbsttestender Code



> *Wenn jemand sagt, dass sein Code ein paar Tage lang kaputt war, während er Refactoring betreibt, kann man ziemlich sicher sein, dass er nicht refactoriert hat.*



Legacy Code

- Ohne Tests kann Legacy-Code nicht sicher refaktorisiert werden
- Refactoring kann ein fantastisches Werkzeug sein, um ein Legacy-System zu verstehen

5.

Schlechte Gerüche im Code

Erläuterung der "schlechten Gerüche"


“Wenn es stinkt, muss man wechseln.”
— Oma Özdemir

Wonach riecht es hier?

- MYSTERIÖSER NAME
- DUPLIZIERTER CODE
- LANGE FUNKTIONEN
- LANGE PARAMETERLISTE
- GLOBALE DATEN
- VERÄNDERBARE DATEN
- FEATURE NEID
- GROSSE KLASSEN
- KOMMENTARE

MYSTERIÖSER NAME

- Aussagekräftigen Namen für Funktionen, Module, Variablen und Klassen

- Refactorings
  - Change Function Declaration
  - Rename Variable
  - Rename Field

DUPLIZIERTER CODE

- Beeinträchtigt Lesbarkeit und Wartbarkeit des Programms

- Refactorings
  - Extract Function
  - Slide Statements
  - Pull Up Method

LANGE FUNKTIONEN

- Suche nach Kommentaren, Bedingungen und Schleifen

- Refactorings
  - Extract Function
  - Replace Temp with Query
  - Introduce Parameter Object

LANGE PARAMETERLISTE

- Lange Parameterlisten können verwirrend sein
- Refactorings
  - Introduce Parameter Object
  - Combine Functions into Class

GLOBALE DATEN

- Umfang globaler Daten so weit wie möglich zu begrenzen, um ihre Auswirkungen zu kontrollieren

- Refactorings
  - Encapsulate Variable

VERÄNDERBARE DATEN

- Datenänderungen können unerwartete Folgen und Fehler verursachen.
- Funktional programmierende Ansätze bevorzugen unveränderliche Datenstrukturen.

- Refactorings
  - Encapsulate Variable
  - Extract Function

FEATURE NEID

- Fasse Änderungen zusammen, indem du Dinge zusammenfügst, die sich gemeinsam ändern. In Ausnahmefällen wird das Verhalten verschoben, um Änderungen an einem Ort zu halten

- Refactorings
  - Move Function
  - Extract Function

GROSSE KLASSEN

- Klassen mit übermäßigem Code sind Anzeichen für Chaos und wiederholten Code
- Refactorings
  - Extract Class

Mittelsmann

- Klassen, die lediglich Aufgaben an andere Klassen delegieren, bieten keinen Mehrwert und sind oft überflüssig.
- Der Middle Man fungiert als "Durchgangsstation", ohne eigene Logik oder Verantwortung.

- Refactorings
  - Remove Middle Man
  - Inline Methods

Spaces, New Lines

- Kein Smell nach Fowler aber dennoch tragen sie ein Beitrag zur Code-Qualittät

Magic Numbers

- Zahlenwerte im Code ohne Erklärung und Kontext
- Der Middle Man fungiert als "Durchgangsstation", ohne eigene Logik oder Verantwortung.

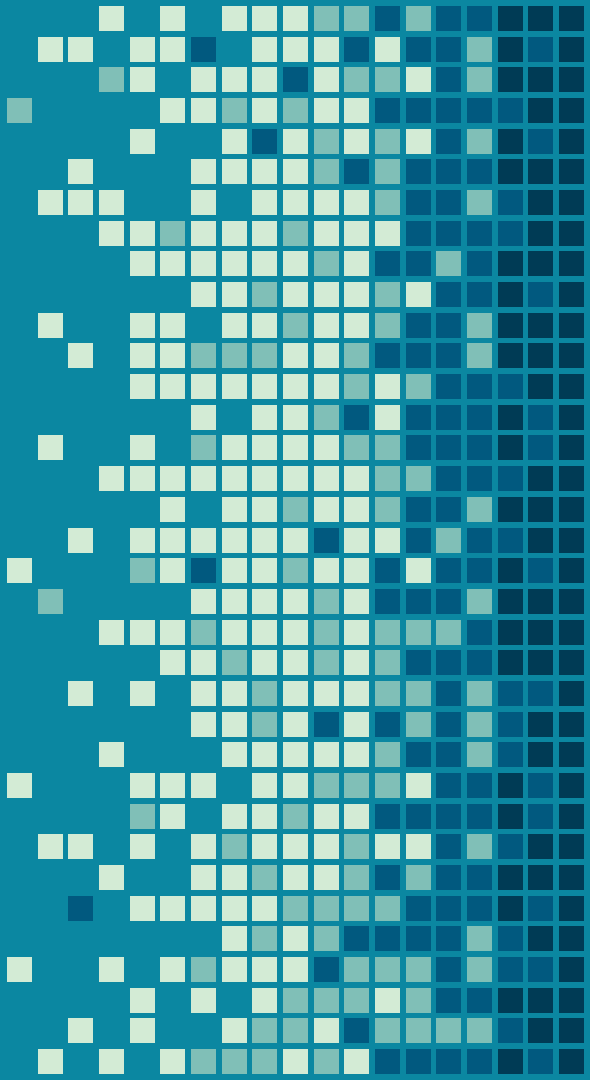
- Refactorings
  - Constants
  - Enums

KOMMENTARE

- Kommentare können auf schlechten Code hinweisen und sind oft überflüssig nach dem Refactoring

- Refactorings
  - Extract Function

> *Wenn Sie das Bedürfnis haben, einen Kommentar zu schreiben, versuchen Sie zunächst, den Code so umzugestalten, dass jeder Kommentar überflüssig wird.*



XML Dokumentation

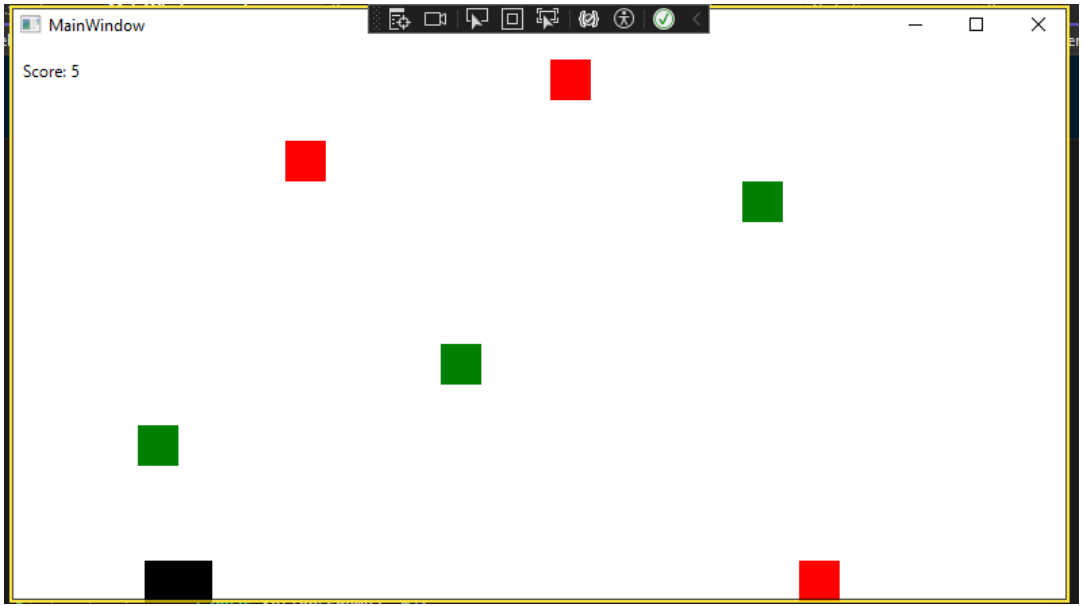
- Header Kommentar an puplic geteilten Methoden (Außnahme internal)

```
/// <summary>
/// Calculates the area of a rectangle.
/// </summary>
/// <param name="width">The width of the rectangle.</param>
/// <param name="height">The height of the rectangle.</param>
/// <returns>The area of the rectangle.</returns>
0 references
public double CalculateRectangleArea(double width, double height)
```

Prompt: /doc

Refacoring Simple Game

- Erkenne die möglichen Smells
- Refactor



Switch Statements

- **Schwierige Wartbarkeit**
- **Erhöhte Fehleranfälligkeit**:
- **Verletzung des Open-Closed-Prinzips**

- Refactoring: Polymorphism, Interfaces

Vorteile der Lösung durch Polymorphismus oder Interfaces

- **Wartbarkeit**: Neue Formen können durch das Erstellen einer neuen Unterklasse hinzugefügt werden, ohne dass bestehender Code geändert werden muss.
- **Klarheit**: Jeder Switch Typ hat ihre eigene Klasse und ist für ihre eigene Methodenausführung verantwortlich.

Messaging Chains

Definition:

- Eine **Message Chain** ist eine Verkettung von Methodenaufrufen über mehrere Objekte hinweg, z.B. obj.getA().getB().getC().

Problem:

- Erhöht die **Abhängigkeit** zwischen Klassen, da der aufrufende Code die internen Details der Objekte kennen muss.
- Verletzt das Prinzip der **Kapselung** und macht den Code schwer wartbar.

Refactor: Create Delegating Method from ObjA

Inappropriate Intimacy

- Klassen oder Module zu viele Details anderer Klassen kennen und verwenden.
- Typischerweise, wenn eine Klasse die internen Details einer anderen ausnutzt, statt auf ihre öffentliche API zuzugreifen.

- Refactor: Encapsulation

- **1. Bloaters (Aufgeblähte Strukturen)**
  - **Long Method**: Methoden sind zu lang und übernehmen zu viele Aufgaben.
  - **Large Class**: Klassen haben zu viele Verantwortlichkeiten und sind übermäßig groß.
  - **Primitive Obsession**: Übermäßige Verwendung primitiver Datentypen anstelle von eigenen Objekten, z.B. für Telefonnummern oder Währungen.
  - **Long Parameter List**: Methoden haben zu viele Parameter, was die Verständlichkeit erschwert.
  - **Data Clumps**: Mehrere Variablen, die immer gemeinsam auftreten, sollten zu einer eigenen Klasse zusammengefasst werden.

- **2. Object-Orientation Abusers (Missbrauch objektorientierter Prinzipien)**
  - **Switch Statements**: Übermäßige Verwendung von `switch`-Anweisungen oder `if-else`-Ketten statt Polymorphismus.
  - **Temporary Field**: Variablen, die nur unter bestimmten Bedingungen verwendet werden, sollten ausgelagert werden.
  - **Refused Bequest**: Eine Unterklasse erbt Methoden oder Felder, die sie nicht benötigt, was auf eine falsche Vererbung hinweist.
  - **Alternative Classes with Different Interfaces**: Ähnliche Funktionalität in Klassen, aber mit unterschiedlichen Schnittstellen.

- **3. Change Preventers (Änderungen erschwerende Strukturen)**
  - **Divergent Change**: Häufige Änderungen an einer Klasse, die verschiedene, unabhängige Aspekte betreffen, erfordern ständige Anpassungen.
  - **Shotgun Surgery**: Eine kleine Änderung erfordert Anpassungen an vielen Stellen im Code.
  - **Parallel Inheritance Hierarchies**: Eine Klasse in einer Vererbungshierarchie erfordert das Hinzufügen einer Klasse in einer anderen Hierarchie, was zu einer engen Kopplung führt.

- **4. Dispensables (Überflüssiges)**
  - **Duplicate Code**: Gleicher oder ähnlicher Code ist an mehreren Stellen im Programm vorhanden und sollte zusammengeführt werden.
  - **Lazy Class**: Klassen, die kaum Funktionalität besitzen und überflüssig erscheinen.
  - **Data Class**: Klassen, die nur Daten enthalten und kein Verhalten besitzen.
  - **Dead Code**: Nicht genutzter oder veralteter Code, der entfernt werden sollte.
  - **Speculative Generality**: Nicht genutzte Funktionen oder Parameter, die „für die Zukunft“ erstellt wurden, aber keine Anwendung finden.

- **5. Couplers (Kopplungen)**
  - **Feature Envy**: Eine Methode greift häufig auf Daten oder Methoden einer anderen Klasse zu, was darauf hinweist, dass sie in die andere Klasse gehört.
  - **Inappropriate Intimacy**: Klassen kennen zu viele Details voneinander und sind stark miteinander verflochten.
  - **Message Chains**: Längere Aufrufketten wie `obj.getA().getB().getC()` weisen auf enge Kopplung hin.
  - **Middle Man**: Eine Klasse delegiert ihre Aufgaben fast ausschließlich an eine andere Klasse und bietet keinen eigenen Mehrwert.

- **6. Others (Sonstige)**
  - **Incomplete Library Class**: Eine Klassenbibliothek fehlt notwendige Funktionen, was oft zu Workarounds führt.
  - **Comments**: Zu viele Kommentare, die zur Erklärung des Codes dienen, deuten darauf hin, dass der Code selbst nicht klar genug ist.

7.

Katalog von nützlichen Refactorings

gemäß Martin Fowler

Katalog Übersicht

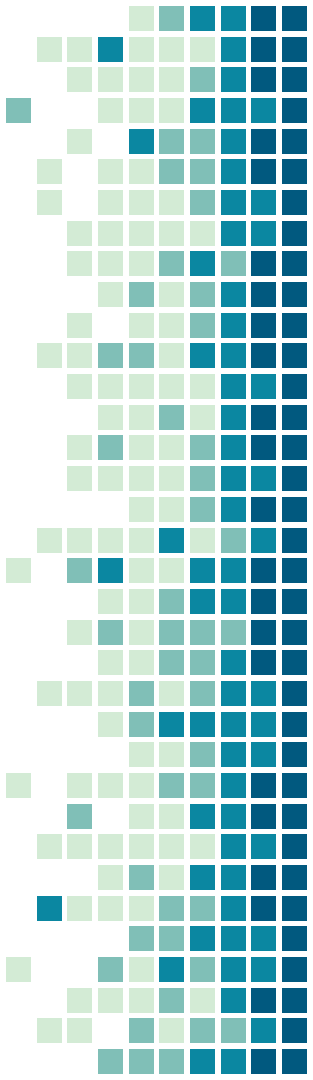
- EXTRACT FUNCTION
- INLINE FUNCTION
- EXTRACT VARIABLE
- INLINE VARIABLE
- ENCAPSULATE VARIABLE
- RENAME VARIABLE
- INTRODUCE PARAMETER OBJECT
- COMBINE FUNCTIONS INTO CLASS

8.
Katalog von Encapsulate Refactorings

gemäß Martin Fowler

Encapsulate Katalog Übersicht

- REPLACE PRIMITIVE WITH OBJECT
- REPLACE TEMP WITH QUERY
- EXTRACT CLASS
- INLINE CLASS
- HIDE DELEGATE
- REMOVE MIDDLE MAN
- SUBSTITUTE ALGORITHM



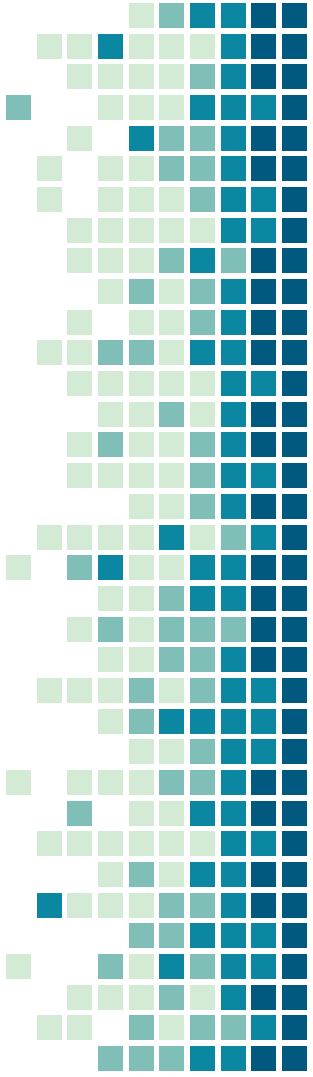
9.

Katalog von Move Refactorings

gemäß Martin Fowler

Move Katalog Übersicht

- MOVE FUNCTION
- MOVE FIELD
- MOVE STATEMENTS INTO FUNCTION
- SLIDE STATEMENTS
- SPLIT LOOP
- REMOVE DEAD CODE



10.

Katalog von Organize Refactorings

gemäß Martin Fowler

Organize Katalog Übersicht

- SPLIT VARIABLE
- RENAME FIELD
- CHANGE REFERENCE TO VALUE
- CHANGE VALUE TO REFERENCE

11.

Katalog von Conditional Refactorings

gemäß Martin Fowler

Conditional Katalog Übersicht

- CONSOLIDATE CONDITIONAL EXPRESSION
- REPLACE NESTED CONDITIONAL WITH GUARD CLAUSES
- REPLACE CONDITIONAL WITH POLYMORPHISM
- INTRODUCE ASSERTION

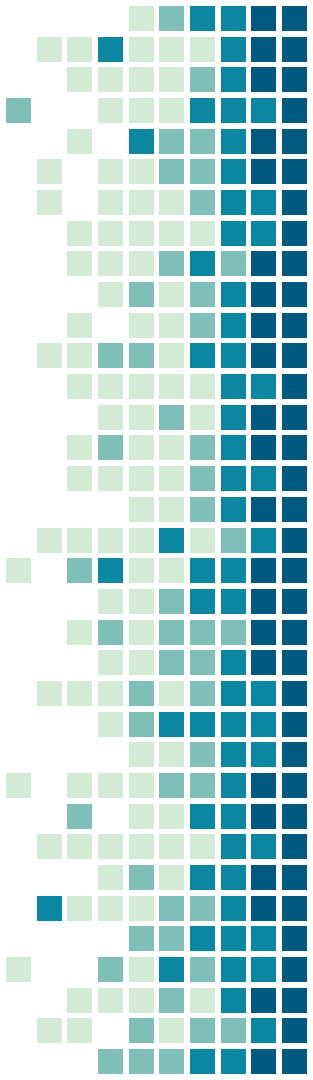
12.

Katalog von API Refactorings

gemäß Martin Fowler

API Katalog Übersicht

- PARAMETERIZE FUNCTION
- REMOVE FLAG ARGUMENT
- PRESERVE WHOLE OBJECT
- REPLACE PARAMETER WITH QUERY
- REPLACE QUERY WITH PARAMETER
- REMOVE SETTING METHOD
- REPLACE CONSTRUCTOR WITH FACTORY FUNCTION



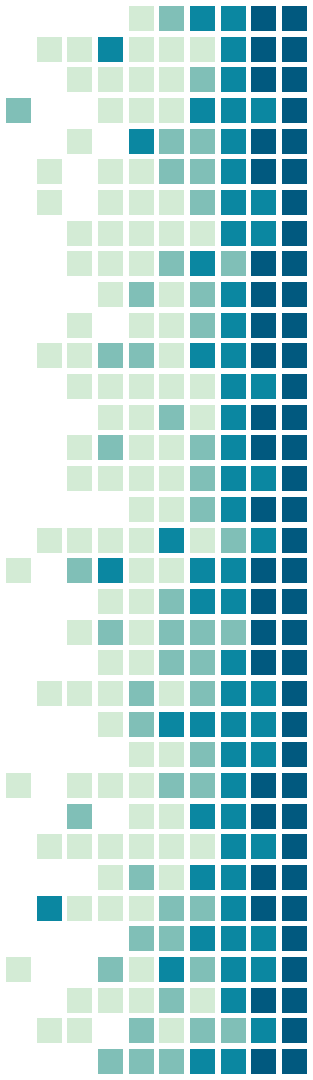
13.

Katalog von Inheritance Refactorings

gemäß Martin Fowler

Inheritance Katalog Übersicht

- PULL UP METHOD
- PULL UP FIELD
- PUSH DOWN METHOD
- PUSH DOWN FIELD
- REPLACE TYPE CODE WITH SUBCLASSES
- REMOVE SUBCLASS
- EXTRACT SUPERCLASS
- COLLAPSE HIERARCHY



DANKE!

Noch Fragen?