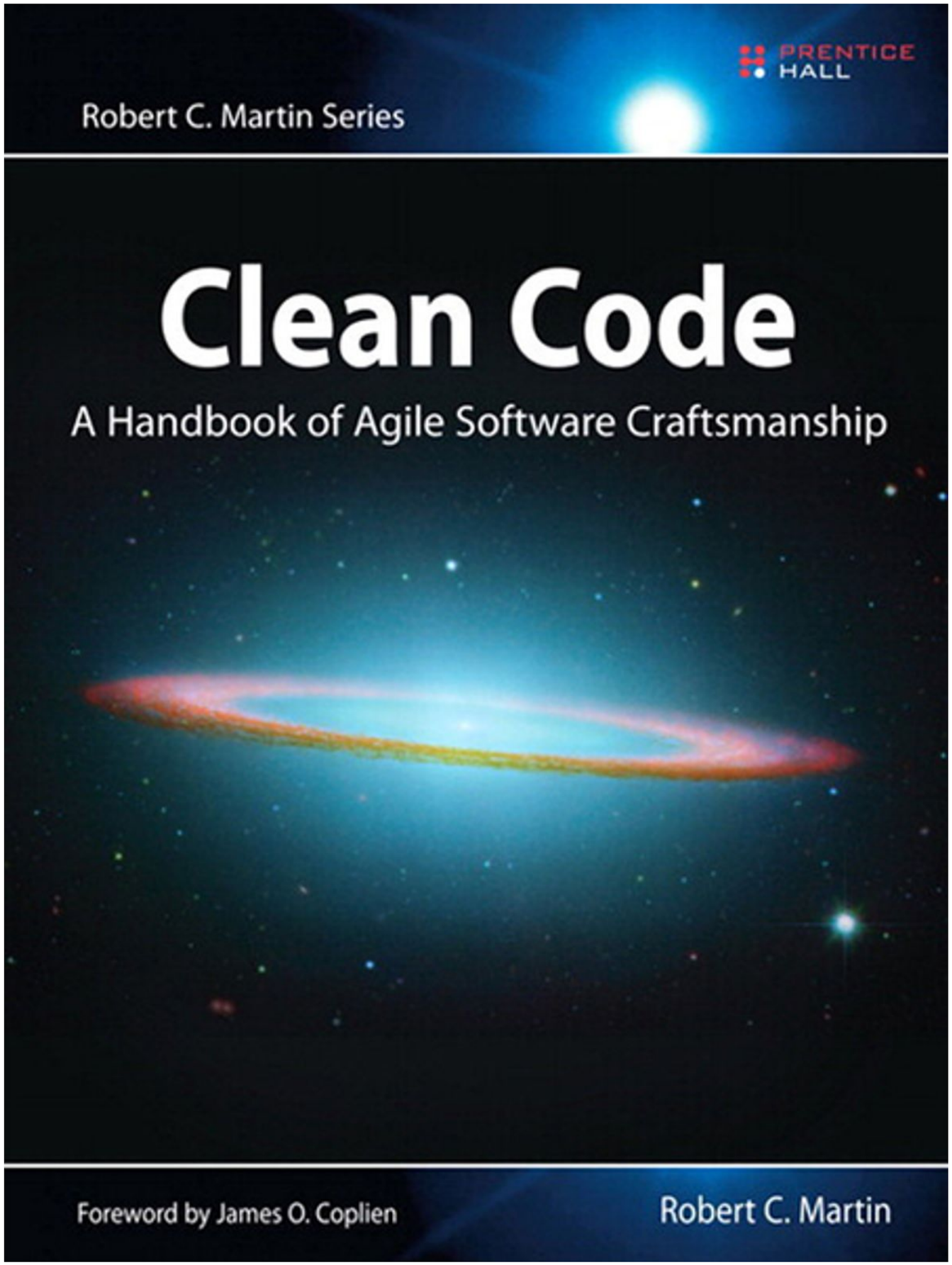
<CleanCode>

Wie man bessere Software schreibt

// von Hannes Dröse



Clean Code

A Handbook of Agile Software Craftsmanship

Robert C. Martin

„Even bad code can function. But if code isn't clean, it can bring a development organization to its knees. Every year, countless hours and significant resources are lost because of poorly written code. But it doesn't have to be that way.“

Was macht guten Code aus?

Funktionalität

Lesbarkeit // Effizienz

Erweiterbarkeit // Sicherheit

Vorsicht vor Optimierungen

Rules of Optimization:
Rule 1: Don’t do it.
Rule 2 (for experts only): Don’t do it yet.
// Michael Anthony Jackson

- moderne Compiler optimieren sehr viel, wenn der Code gut strukturiert ist
- hoch optimierter (effizienter) Code ist schwer zu lesen
- Optimierungen stark von der Programmiersprache abhängig
- ist es überhaupt nötig?
- Besser:
  - erstmal Clean Code
  - dann den Bottleneck identifizieren und (nur) diesen optimieren

Grundprinzipien

- KISS – keep it simple, stupid
- DRY – Don’t Repeat Yourself
- Use Meaningful Names
- No Magic Numbers
- About Comments
- Abide by the Code Style
- Use User-defined Data Structures
- SRP – Single Responsibility Principle
- Favor Pure Functions*
- SoC – Separation of Concerns
- TDD – Test Driven Development

KISS – keep it simple, stupid

- einfache Lösungen sind komplizierten und komplexen vorzuziehen
- kein praktisches Prinzip, mehr ein Mantra

DRY – Don’t Repeat Yourself

- Dopplungen im Code sind unter allen Umständen zu vermeiden
- macht Code sehr lang und schwer zu lesen
- erschwert spätere Änderungen

```cpp
#include <iostream>

int main() {
    int x,y,z;

    std::cout << "Bitte x eingeben:" << std::endl;
    std::cin  >> x;

    std::cout << "Bitte y eingeben:" << std::endl;
    std::cin  >> y;

    std::cout << "Bitte z eingeben:" << std::endl;
    std::cin  >> z;

    return 0;
}
```

```cpp
#include <iostream>

int askForInt(const char msg[]) {
    int result;
    std::cout << msg << std::endl;
    std::cin  >> result;
    return result;
}

int main() {
    int x = askForInt("Bitte x eingeben:");
    int y = askForInt("Bitte y eingeben:");
    int z = askForInt("Bitte z eingeben:");

    return 0;
}
```

Use Meaningful Names

- aussagekräftige, beschreibende Namen für Funktionen, Variablen, Parameter, Structs, Klassen usw. verwenden
- Namen dürfen ruhig so lang sein, wie benötigt
- Analogie: 2. Überschrift in der Zeitung
- Für den gleichen Kontext immer den gleichen Begriff verwenden z.B. get, set, calc, create

```
void doMagic(int param1, float param2) {
    // ...
}

float doMoreMagic(int v[], int n) {
    // ...
}

SHouse getHouse() {
    // ...
}

SCustomer fetchCustomer() {
    // ...
}
```

```
void printToScreen(int index, float value) {
    // ...
}

float calcAverage(int values[], int numOfValues) {
    // ...
}

SHouse getHouseFromDatabase() {
    // ...
}

SCustomer getCustomerFromDatabase() {
    // ...
}
```

No Magic Numbers

- hart-kodierte Zahlen im Code vermeiden!
- ist schwer zu verstehen und erschwert spätere Änderungen
- stattdessen: Parametrisieren oder Konstanten nutzen

```
struct SShip {
    int length;
};

struct SPlayer {
    SShip ships[5];
};

int sumUpShipLengths(SShip ships[]) {
    int result = 0;
    for (int i = 0; i < 5; i++) {
        result += ships[i].length;
    }
    return result;
}
```

```
int sumUpShipLengths(SShip ships[], int numOfShips) {
    int result = 0;
    for (int i = 0; i < numOfShips; i++) {
        result += ships[i].length;
    }
    return result;
}
```

```
const int NUM_OF_SHIPS_PER_PLAYER = 5;

struct SPlayer {
    SShip ships[NUM_OF_SHIPS_PER_PLAYER];
};

int sumUpShipLengths(const SShip ships[]) {
    int result = 0;
    for (int i = 0; i < NUM_OF_SHIPS_PER_PLAYER; i++) {
        result += ships[i].length;
    }
    return result;
}
```

Noch besser: beides zusammen! (DRY)

```
int sumUpShipLengths(const SPlayer &player) {
    return sumUpShipLengths(player.ships, NUM_OF_SHIPS_PER_PLAYER);
}
```

About Comments

“Every time you write a comment, you should grimace and feel the failure of your ability of expression.“

// Robert C. Martin

- sind bei konsequenter Verwendung von Clean Code meistens unnötig, der Code spricht für sich
- legitime Kommentare:
  - legal comments `// v1.0.1 © Hannes Dröse | office.hd@gmx.net`
  - informative comments `// DEPRECATED: is removed in next release`
  - explanations of intent `// explain the why, not the what`
  - warning of consequences `// WARNING: returns null on failure`
  - TODO `// TODO: make function handle negative input as well`
  - marking as important `// IMPORTANT: prefer use of this function`
  - documenting public API

```
/**
 * Multiplies two matrizes and returns the result as a new matrix.
 * If the matrices can't be multiplied, an empty matrix will be
 * returned.
 * @param: matrix1 – the left matrix
 * @param: matrix2 – the right matrix
 * @returns SMatrix
 */
SMatrix multiplyMatrizes(SMatrix matrix1, SMatrix matrix2) {
```

Abide by the Code Style

- für jede Programmiersprache gibt es Richtlinien zur Formatierung (Coding Conventions)
- Festlegungen zu Einrückungen, Klammern, Leerzeichen, Zeilenlängen und -umbrüchen etc.
- Tools nutzen: z.B. Beautify (HTML,CSS,Sass,JS,JSON), phpfmt (PHP), Clang-Format (C,C++,C#,Java,Objective-C), gofmt (GoLang), shell-format (bash,dockerfile,gitignore)
- Gute Basis, reicht aber nicht!
- Typische Regeln:
  - Codezeilen sollten möglichst nicht länger als 80 und allerhöchstens 120 Zeichen lang sein!!!
  - Dateien sollten nicht mehr als 200 Zeilen lang sein
  - Wichtiges steht oben in der Datei, unwichtiges unten
    - z.B. Konstanten, komplexe Datentypen, Funktionen (in absteigender Wichtigkeit)
    - OOP: Klassenkonstanten, -variablen, -funktionen
      -> immer: public vor protected vor private

Abide by the Code Style (2)

- meistens gibt es in einer Firma einen sog. **Code Style:** ein Dokument welches beschreibt wie Code zu formatieren ist, wie Variablen, Klassen, Funktionen zu benennen sind, etc. **-- beliebig umfangreich!**
- Bsp:
  - Google C++ Code Style Guide: https://google.github.io/styleguide/cppguide.html

Names and Order of Includes

In *dir/foo.cc* or *dir/foo_test.cc*, whose main purpose is to implement or test the stuff in *dir2/foo2.h*, order your includes as follows:

1. *dir2/foo2.h*.
2. A blank line
3. C system headers (more precisely: headers in angle brackets with the `.h` extension), e.g. `<unistd.h>`, `<stdlib.h>`.
4. A blank line
5. C++ standard library headers (without file extension), e.g. `<algorithm>`, `<cstddef>`.
6. A blank line
7. Other libraries' `.h` files.
8. Your project's `.h` files.

Separate each non-empty group with one blank line.

- Herr Sahm: „Gibt es in ihrer Firma keinen Code Style, kündigen Sie!“

Use User-defined Data Structures

- in jeder Sprache gibt es Nutzer-definierte Datentypen
  z.B. in C++: enum, struct, union, typedef, class
- Gliederung des Programms in logische, strukturierte, hierarchisch Einheiten
- Orientierung an der realen Welt, daraus werden die Einheiten abgeleitet (Grundprinzip der OOP)

```
enum EProductType { PIZZA, BEVERAGE, SALAD, OTHER };

struct SProduct {
    const char* name;
    const char* description;
    float price;
    EProductType type;
};

struct SCartItem {
    SProduct* product;
    int amount;
};

struct SCart {
    SCartItem items[];
    int numOfItems;
};
```

```
float calcCartPrice(const SCart &cart) {
    float result = 0;
    for (int i = 0; i < cart.numOfItems; i++) {
        result += cart.items[i].amount * cart.items[i].product->price;
    }
    return result;
}
```

Use User-defined Data Structures (2)

- macht Code schlanker (DRY)
- ist besser zu lesen, da viele zusammenhängende Informationen in ein Objekt mit einem Namen zusammengefasst werden

```cpp
int main() {
    int numOfProducts = 5;

    const char* productNames[numOfProducts];
    const char* productDescriptions[numOfProducts];
    float productPrices[numOfProducts];
    // ...

    return 0;
}
```

```cpp
int main() {
    int numOfProducts = 5;

    SProduct products[numOfProducts];

    return 0;
}
```

```cpp
void updateProduct(const char* &name, const char* &description, float &price, EProductType &type) {
    // ...
}
```

```cpp
void updateProduct(SProduct &product) {
    // ...
}
```

- Tipp: Beim Programmieren immer mit den Data-structures beginnen! Sind die sauber definiert, ist der Rest easy.

SRP — Single Responsibility Principle

- Funktionen, Structs und Klassen haben genau EINE Aufgabe
- vermeidet unerwartete Überraschungen
- Name beschreibt genau, was passiert `// keine Kommentare nötig`
- gut zu lesen und zu verstehen
- dadurch bekommt man viele kleine Funktionen statt wenige große; das ist gut -> Wiederverwendbarkeit (DRY)
- Faustregel:
  - Funktionen sollten maximal 20 Zeilen lang sein
  - Klassen maximal 200 Zeilen

```cpp
bool isNumberEven(int number) {
    bool result = (number % 2 == 0);

    if (result) {
        std::cout << number << " ist eine gerade Zahl"   << std::endl;
    } else {
        std::cout << number << " ist eine ungerade Zahl" << std::endl;
    }

    return result;
}
```

```cpp
bool isNumberEven(int number) {
    return number % 2 == 0;
}

void println(int number, const char msg[]) {
    std::cout << number << msg << std::endl;
}

void tellUserIfNumberIsEven(int number) {
    if (isNumberEven(number)) {
        println(number, " ist eine gerade Zahl");
    } else {
        println(number, " ist eine ungerade Zahl");
    }
}
```

Favor Pure Functions*

- pure Funktionen haben keine Nebeneffekte (Side-Effects)
  - keine Änderung der übergebenen Parameter oder an globalen Variablen
  - kein Schreiben oder Lesen in Dateien oder Datenbanken
  - keine Bildschirm-Ausgaben, Nutzereingaben etc.
- haben immer einen return-Wert
- gleicher Input gibt immer den gleichen Return-Wert
- sind leicht zu verstehen, zuverlässig und leicht zu testen
- impure Functions lassen sich nicht komplett vermeiden, sollten aber sparsam verwendet werden
- Tipp: von den puren Funktionen trennen und kennzeichnen

```
// * kein Clean Code Prinzip,
//   trotzdem weit verbreitet
//   und von HD wärmstens empfohlen ;-)
```

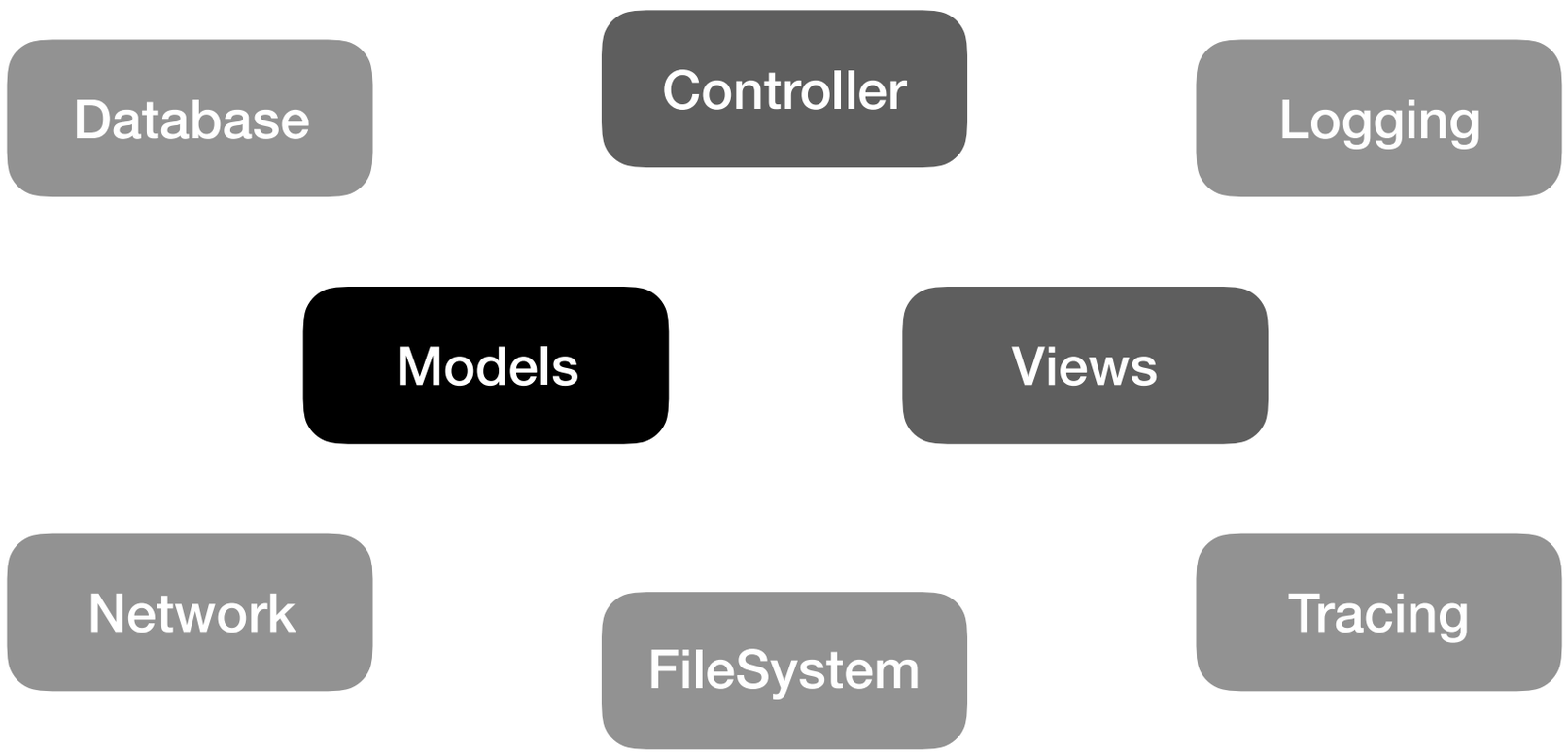
Favor Pure Functions* (2)

```cpp
struct S2DVector {
    float x, y;
};

void printVector(const S2DVector &vec) {
    std::cout << vec.x << ", " << vec.y << std::endl;
}

// return typ void marks an impure function
// same for non-const pointers as parameters
void add3(S2DVector &vector) {
    vector.x += 3;
    vector.y += 3;
}

int main() {
    // vector: 1.2, 7.5
    S2DVector vector = {1.2, 7.5};

    add3(vector);

    // vector has been changed
    // vector: 4.2, 10.5
    printVector(vector);

    return 0;
}
```

```cpp
// this is still impure!
void printVector(const S2DVector &vec) {
    std::cout << vec.x << ", " << vec.y << std::endl;
}

// this is pure now
S2DVector add3(const S2DVector &vector) {
    S2DVector result;
    result.x = vector.x + 3;
    result.y = vector.y + 3;
    return result;
}

int main() {
    // vector: 1.2, 7.5
    S2DVector vector = {1.2, 7.5};

    S2DVector newVector = add3(vector);

    // vector was not changed
    // vector: 1.2, 7.5
    printVector(vector);

    // instead newVector holds the result
    // newVector: 4.2, 10.5
    printVector(newVector);

    return 0;
```

SoC — Separation of Concerns

- wir haben ja nun viele kleine Funktionen nach SRP, aber wie organisieren wir die innerhalb des Projektes?
- Organisation des Codes in Dateien und Ordner nach ihren Zuständigkeiten
- Bekannte Pattern: MVC-Pattern, MV-VM-Pattern



- z.B. Für jeden Bereich einen Ordner
- Models gibt es eigentlich in jedem Programm, alle anderen Bereichen nur nach Bedarf

TDD – Test Driven Development

“Why do most developers fear to make continuous changes to their code? They are afraid they’ll break it! Why are they afraid they’ll break it? Because they don’t have tests.”

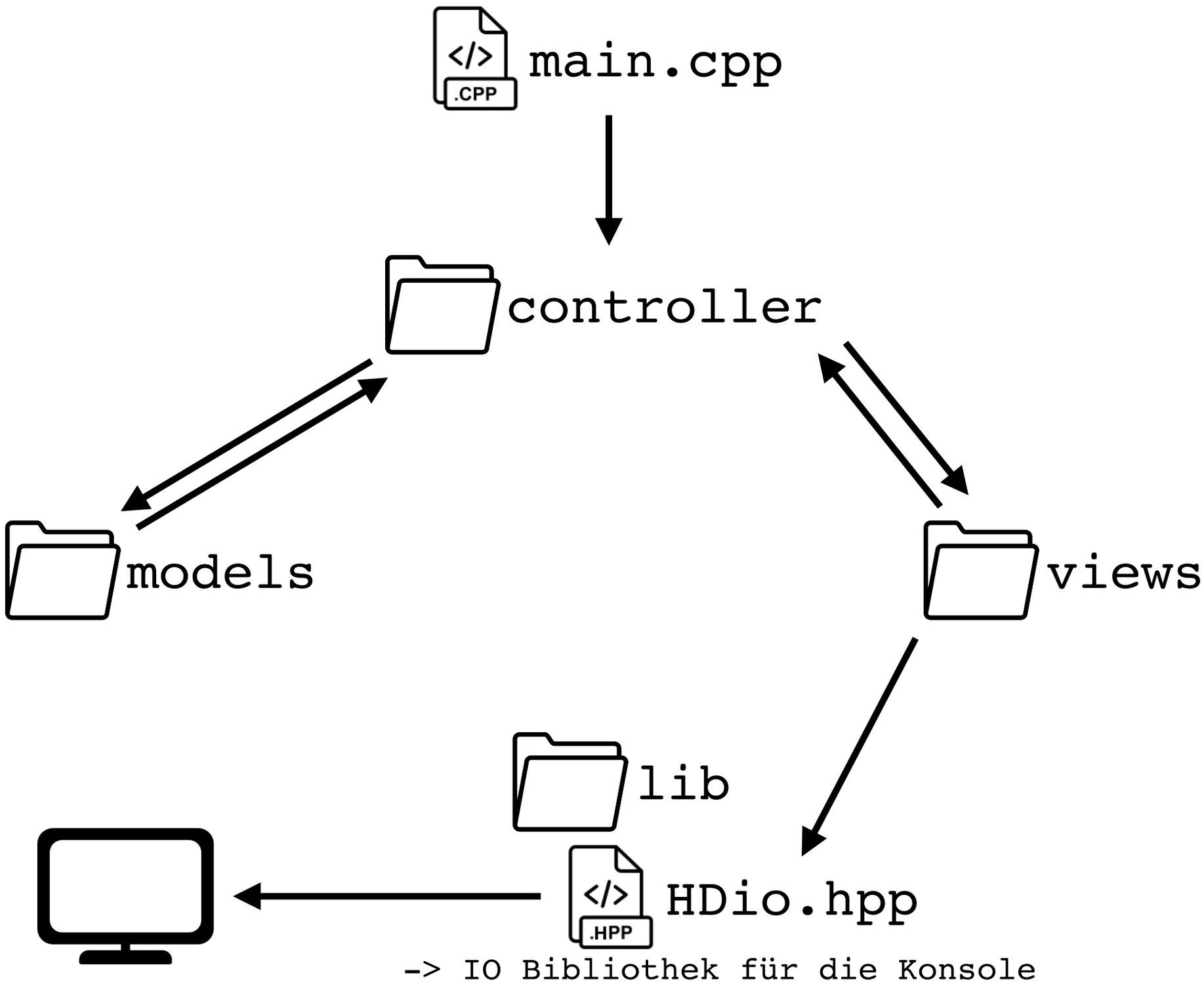
// Robert C. Martin

- am besten parallel zum Code, gleich automatisierte Tests (Unit Tests) schreiben
- 3 Rules of TDD:
  1. You are not allowed to write any production code unless it is to make a failing unit test pass.
  2. You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failures.
  3. You are not allowed to write any more production code than is sufficient to pass the one failing unit test.
- „Tests sind wichtig, aber ob es TDD sein muss, kann man diskutieren!“ // Hannes Dröse

Code-Beispiel

TicTacToe für die Konsole in C++

Architektur



Models

```
enum EPlayer { ONE, TWO, NUM_OF_PLAYERS };

struct SGame {
    SBoard board;
    SPlayer players[NUM_OF_PLAYERS];
    EPlayer playerOnTurn;
};
```

```
const int BOARD_ARRAY_SIZE = NUM_OF_ROWS_COLS * NUM_OF_ROWS_COLS;

struct SBoard {
    SPlayer* fields[BOARD_ARRAY_SIZE]; // nullptr means field is empty
};
```

```
struct SPlayer {
    char sign;
};
```

```
const int MIN_ROW_COL = 0;
const int MAX_ROW_COL = 2;

const int NUM_OF_ROWS_COLS = MAX_ROW_COL - MIN_ROW_COL + 1;

struct SPosition {
    int row;
    int col;
};
```

Controller – Hauptprogramm

```
void playTicTacToe() {
    SGame game;

    beginGame(game);

    while (!isGameOver(game)) {
        makeTurn(game);
    }

    endGame(game);
}
```

-> zum Code

https://source.ai.fh-erfurt.de/ha9384dr/TicTacToe.git

https://source.ai.fh-erfurt.de/ha9384dr/Battleship.git

Quellen

- http://www.inf.fu-berlin.de/inst/ag-se/teaching/K-CCD-2014/Clean-Code-summary.pdf
- https://clean-code-developer.de/
- https://de.slideshare.net/JandV/clean-code-summary
- https://hackernoon.com/few-simple-rules-for-good-coding-my-15-years-experience-96cb29d4acd9
- https://www.goodreads.com/author/quotes/45372.Robert_C_Martin