Vorstellen
Titel: Von der Idee zum Programm
Strukturiertes Vorgehen

Locator: Bisher Programmiersprache, dann Testen, jetzt alles Zusammenführen
Benutzen um Programme zu schreiben

Bitte Mitschreiben

Warum überhaupt? -> Probleme Lösen:
* sehr kleine Probleme lösbar
* kleine Probleme, schwierig lösbar
* große Probleme -> gar nicht lösbar
-> Das unterschied gute / schlechte Programmierer

```
private void solveAnyProblem() {
    while (isStillUnsolved()) {
        analyzeProblem();
        chooseSuitablePart();
        implementPart();
    }
}
```

Stoff heute -> als Programm notiert (PAUSE)

1. allgemein: Strukturiertes Vorgehen
2. Am Beispiel von Test Driven Development
-> EINE Methode gut/erprobt für funktionierenden Programme
-> Viele Andere, aber verwende selber, daher

Explizit wird NIE gelehrt
* Problem Analysieren
** Module finden / Teile Sehen -> Übungssache
* Teil Auswählen: hart!
** Unabhängig, Zentral, nicht-trivial / muss drüber nachdenken -> wird klarer

* **Implementieren: Simpelst.**

Denn:

„As simple as possible, but no simpler“
-- Albert Einstein

Einstein: „So Einfach wie möglich – aber nicht einfacher“
Oder, unter Programmierern: „Do the simplest thing, that could possibly work!“ (übersetzen)

Wie machen?
* Kleine Programme
** Realen Systeme: Vier Gewinnt, großer Schrank in zentraler Verwaltung (äh: Studentendatenbank)
** Existierende Systeme: Teile finden einfacher, natürliche Unterteilung existiert
* Später größere Programme

Das durchspielen, an kleinem Beispiel: Vier Gewinnt

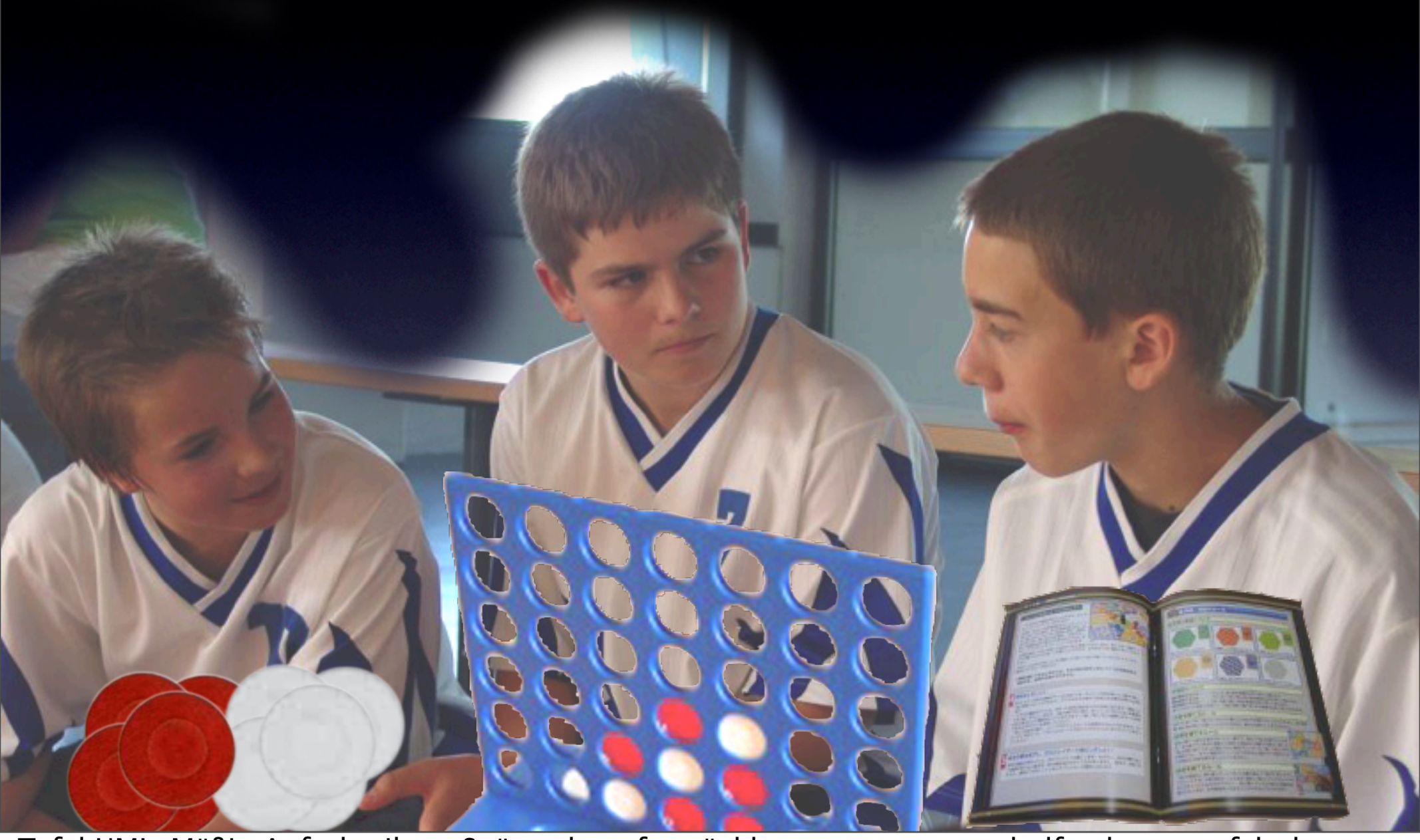
Regeln

Alle wissen Bescheid

-> jetzt zum Vorgehen

1. Teile sehen

Wie: Teile?

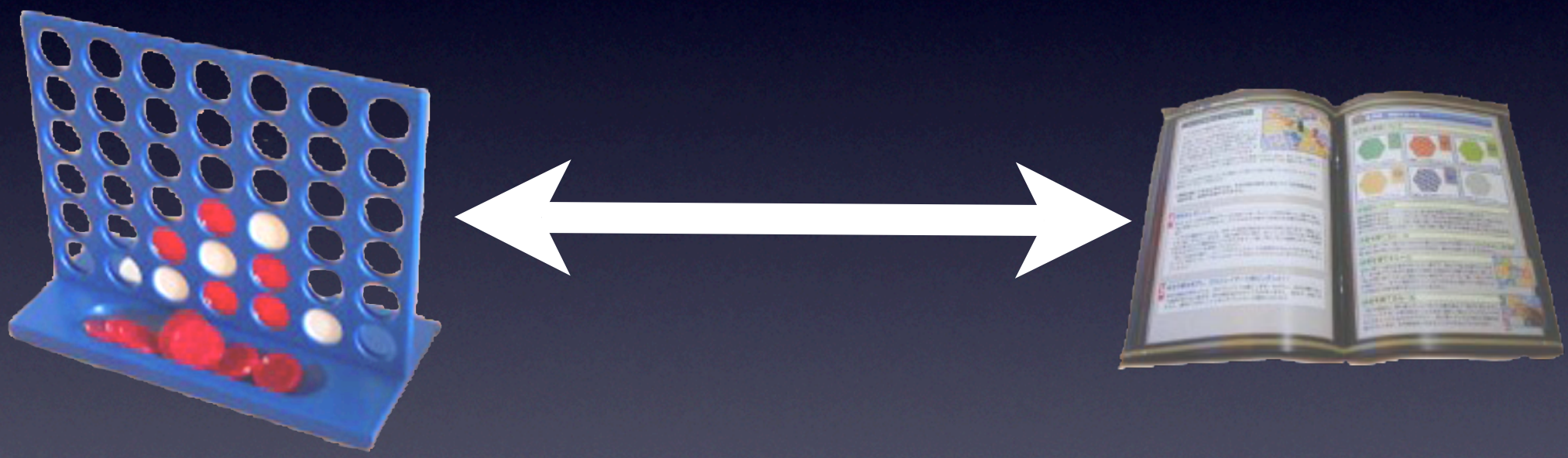


Tafel UML-Mäßig Aufschreiben. Später darauf zurückkommen, wo es geholfen hat, was falsch war
* Wenn Tafel nicht geht Bilder entstehen lassen

Spieler
Spielbrett
Steine
Regeln

Sieht gut aus -> Aber Ausflug Programmiererweißheit

Zwei Arten Objekte



: Visualisierung: Konkret: Spielfeld (links) <-linie-> Abstrakt: Regeln (rechts)

* Konkret: Spieler, Spielbrett, Steine
* Abstrakt: Regeln

Abstrakte Objekte:
* Stehen für Idee, Algorithmus, Regeln
* Nix zum Anfassen, existiert nicht

Wieso Regeln? Anfangs findet keine Abstrakte Objekte -> Erfahrungssache
* PhysikEngine, 3DSoundGenerator, Graph, Scheduler, MemoryManagerPolicy...
-> Meist (!) für größere Projekte unverzichtbar

</ausflug ende>

Bullshit!

* Über Design nachgedacht, kann sogar aufmalen!
* Muss quasi nur noch programmiert werden
* Sorry: Bullshit! (Softwaretechniker vielleicht...)

Idee ohne Umsetzung **immer** zu kompliziert,
oder zu einfach
oder funktioniert nicht
* Mehr Erfahrung == „nur noch“ mäßig, bis viel zu kompliziert
* Routine -> sieht Möglichkeiten für Vereinfachung nicht

Unnötige Komplexität ist der größte Feind!
-> Fehler Direkt
-> Zeit bis Verstanden
-> Versteht falsch -> fehler
Um mit Brian W. Kernighan zu sprechen (C gefunden)

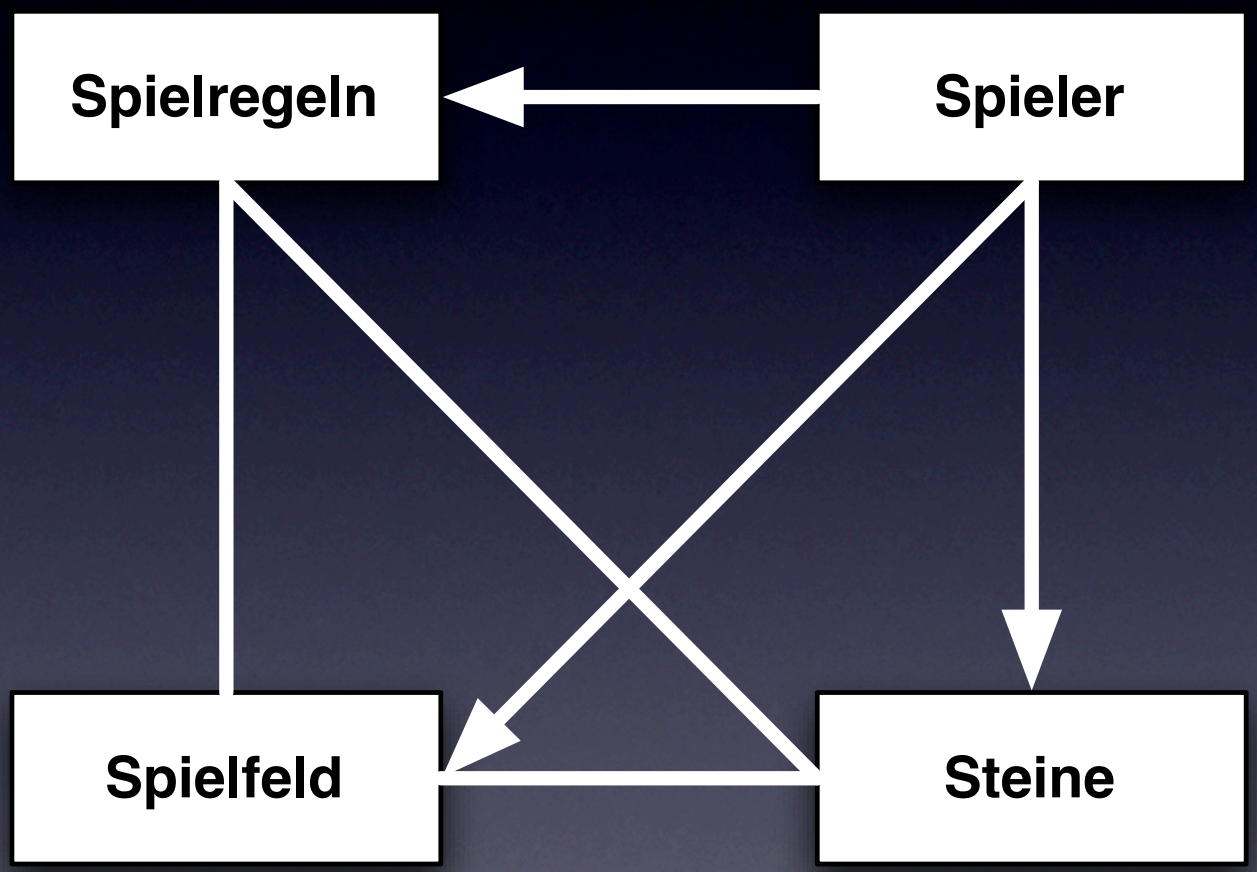
„Debugging is twice as hard
as writing the code in the first place.
Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.“

-- Brian W. Kernighan

übersetzen!

Also -> Wie lösen?

Beispiel



Dinge und Beziehungen – wie mir eingefallen – hier dargestellt

Design absichtlich bevor programmiert

werden mit tatsächlicher Lösung vergleichen

-regeln

- Benutzer-schnittstelle
- Rest

2. Ausflug: Daumenregeln - Regeln die man (fast) immer befolgen sollte

Programme immer Trennen:
* Benutzerschnittstelle <- Keine Funktionalität, nur Eingabe / Ausgabe
* Rest <- Das was wir testen können

Grund:
* Gilt als guter Stil
* Benutzerschnittstellen meist schwerer zu testen
* GUI öfter ausgewechselt (plattform, version2version[word?], konsole/gui/netzwerk/html)

Daher trennt man vom Rest
* UI macht wenig -> klein
* klein -> kann von Hand Testen (muss ja Rest nicht testen)
Mehr Erfahrung -> sogar drei Teile: Model-View-Controller -> Selber lesen

Dieses Wissen auf Beispiel anwenden
</ausflug ende> -> Womit also Anfangen?

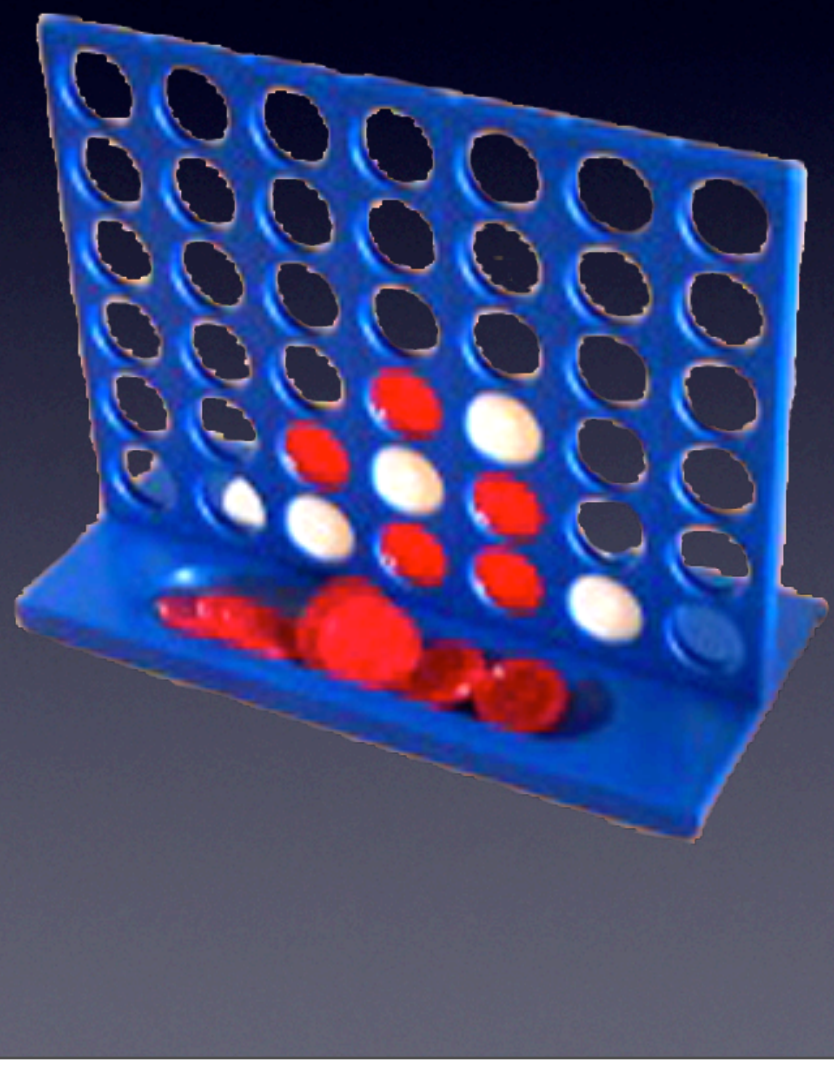
Unabhängig, Zentral?

* Spieler: abhängig
* Regeln: abhängig
* Steine: unabhängig, aber trivial / nicht zentral -> Konstanten?
-> Spielfeld: unabhängig / zentral!

Vielleicht Falsch! Später revidieren? -> Refactoring -> selber lernen (tageswebseite)

Umsetzung: Spielfeld

- Was muss ein Spielfeld können?
- Was ist das einfachste davon?



* Steine einlegen
* Leer herumliegen
** toll! Leer / null / neutral / abbruchkriterium rekursion
*** Gibt es immer! Auch bei jedem Objekt
** Oft ein schwieriges Problem -> muss besonders behandeln
*** Beispiel: Listen: Letztes Element entfernen / erstes Einfügen, Baum: Wurzel Entfernen / Einfügen
** Lösung „Leeres Blatt Problem“ -> IMMMER ein Anfang

Damit anfangen!

Code

Wie Konkret?

2 Dinge: (in verschiedenen Dateien) -> Auf Folien:
* Oben Todo-Liste
* Unten Code

Vorgehen:
Test-Code zeigen, Anfangs auch Programmcode
Später nur Tests, am Schluss nur Test-Ideen
Eigentliche Implementierung nicht -> Übungsmöglichkeit für euch

-> Anfang: Leeres Spielfeld

- Spielfeld leer, wenn nichts passiert ist

```
// File BoardTest.java
class BoardTest extends TestCase {
    public void testEmptyBoard() {

        assertTrue(board.isEmpty());
    }
}
```

Test hinschreiben
-> Wishfull Thinking (Strukturierung)
-> Wenn optimal fertig, dann würde...
mit assert anfangen

dann (DTSTTCPW) einfüllen was man als minimale Vorbereitung braucht

- ~~Spielfeld leer, wenn nichts passiert ist~~

```
// File BoardTest.java
class BoardTest extends TestCase {
    public void testEmptyBoard() {
        Board board = new Board();
        assertTrue(board.isEmpty());
    }
}
// File Board.java
class Board {
    public boolean isEmpty() {
        return true;
    }
}
```

„Ziel“-Klasse soweit wie nötig (DTSTTCPW) implementieren
TODO-Eintrag streichen bzw. nach „erledigt“ verschieben“

DAS war ein Schritt!

Macht man mehr
* Hat man keinen Test
** Weiss nicht wann man fertig ist!
** Ob es Funktioniert
** **nimmt man nicht die einfachste Lösung (weil Verwendung unbekannt)**
*** Programmiert nicht aus Notwendigkeit um nächste Funktion / Meilenstein zu erreichen
** Einfachste Lösung nur durch Benutzen findbar
*** Benutzer Mensch: feedback schwierig
*** Benutzer Test: immer unabhängig von Rest, nie müde, immer schnell, immer akurat
*** TDD daher „Automatisch“ einfach
*** eigentlicher Sinn von TDD

-> nächster schritt? -> Nächst einfaches

- isEmpty() == false, wenn schon Steine gelegt
- Möglichkeit Steine zu legen

```
public void testBoardWithOneStone()
{


    assertFalse(board.isEmpty());
}
```

erst wünschen (assertFalse()) dann Vorbereitungen

oops, steinPlatzieren() gibts nicht

Kann hinschreiben - aber nicht programmieren, da zwei sachen macht.
-> Auskommentieren (dont...) bis Stein platzieren programmiert

- ~~Möglichkeit Steine zu legen~~
- ~~isEmpty() == false, wenn schon Steine gelegt~~
- isEmpty() == false, wenn schon Steine gelegt

```
public void testWhiteStonePlacement() {
    Board board = new Board();
    board.placeWhiteStoneInColumn(5);
    assertTrue(board.hasWhiteStoneAtPosition(5, 0));
    assertFalse(board.hasWhiteStoneAtPosition(5, 1));
}
```

kommt vor - wenn nicht oft, nicht verschachtelt -> Dann ok
Reihenfolge auf TODO-Liste vertauschen, dann

Um Test zu schreiben: erst mit Ziel anfangen
dann Vorbereitungen

dann antitest auf Todo (wenn sowas geht - sinnvoll)
* Natürlich nicht komplett, nur da wo man selbst Fehler macht -> hier um Prinzip zu zeigen

todo-steine-legen erledigt
+ gleich noch rest -> jetzt kann man den isEmpty() == false test Implementieren

```
class Board {
    private int [][] board;
    final static int BOARD_ROWS = 6;
    final static int BOARD_COLUMNS = 7;

    final static int NO_STONE = 0;
    final static int WHITE_STONE = 1;
    final static int RED_STONE = 2;

    public Board() {
        board = new int [BOARD_ROWS][BOARD_COLUMNS];
        // new sets all array-members to 0,
        // wich equals NO_STONE
    }
}
```

Zum programmieren: Representation entscheiden
-> Ich empfehle so, weil einfach

Erklären:
* Wieso Konstanten? -> Beispiel „if (NO_STONE == board[row][column])“
* Mehrdimensionales Array!

Wenn doof: Austauschen
Aber im Zweifel: **Do the simplest thing that could possibly work!** (DTSTTCPW)

- ~~Spielfeld leer, wenn nichts passiert ist~~
- ~~Möglichkeit Steine zu legen~~
- ~~Stein legen, lässt Rest frei~~
- ~~isEmpty() == false, wenn schon Steine gelegt~~
- Rote Steine

An der Stelle: Rückblick -> Ausblick

Gut, jetzt:
* Rote Steine

Fertig

0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0

- Spielfeld anzeigen
- Abwechselnd Spieler nach Reihe fragen

Spielfeld Anzeigen
* System.out.println(spielfeld) -> toString() programmieren
* "Driver" abwechselnd Spieler nach Position fragen

Primitiv, aber man kann schon spielen! und damit auch (von hand) testen
* Ohne UnitTests -> jetzt erst testen (jetzt erst probleme finden)
* kluge programmierer fangen daher damit an - für testen (von hand)
* ist aber nicht mit zentrum
** Kern: Spielfeld mit steinen, nicht wie ausgibt! (netzwerk/html/gui/konsole)

* Problem ist: Lösung Ausgabe schränkt lösung zentrum ein
** Schlecht
** Üblicherweise: wird komplexer
** -> Böse -> mehr fehler -> mehr Debuggen

Daher
* mit Zentrum anfangen
* Feedback bekommen was gut ist

Nun ja...

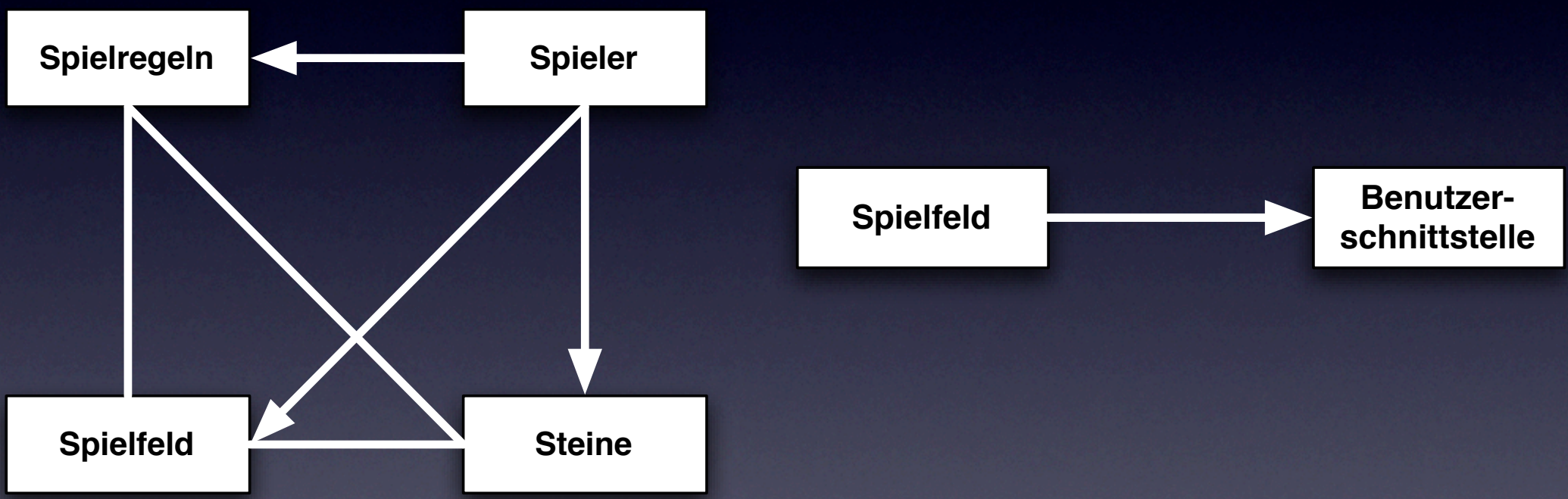
- Stein nur einwerfen wenn Platz
- Ist das Spiel vorbei (vier in einer Reihe)
  - vertikal
  - horizontal
  - diagonal

Begeisterung kurz beiseite:
* Nur einwerfen wenn Platz: rückgabe, testmethode, exception
** Was einfacher ist
** im Spielfeld

* gewonnen
** Regelobjekt
** oder methode auf spielfeld (kann klein sein)

Hat man das Implementiert -> ist man Fertig! Jetzt wars das!

Erstes vs. Endgültiges Design



Vergleich: Erstes Design vs. Endgültiges Design

Was war wichtig, was hat geholfen?
* Erstes Design: Startpunkt gefunden
* Endgültiges Design: Sehr einfach!

Benutzerschnittstelle:
schleife:
	solange noch keiner gewonnen hat
	spielfeld ausgeben -> System.out.println()
	abwechselnd stein einwerfen
-> **EINFACH, wenig möglichkeit für fehler**

-> mit dem Ansatz kann weiter machen -> Beispiel: KI

Ausblick KI

- Aus möglichen Zügen auswählen
  - Lässt Zug mich gewinnen?
  - Lässt Zug Gegner gewinnen?

Wer hat schon eine KI Programmiert?

Gleiche Mittel machen das leicht. Denn: TODOs anschauen:
* KI sagt ob ein Zug...

Damit kriegt man schon interessante Spiele -> Steigerung kann man jederzeit programmieren

Vielleicht doch Regeln extra Objekt?

</Praxisteil>

```
private void solveAnyProblem() {
    while (isStillUnsolved()) {
        analyzeProblem();
        chooseSuitablePart();
        implementPart();
    }
}
```

Was haben wir gemacht?
Wie haben wir‘s gemacht?

Am Schwammigsten / schwierigsten: Auswählen -> Wichtigster Teil
* Risiko / Schwierigsten
* größte Unsicherheit
* oft: unklar wie programmieren am besten (wenn schon 100x gemacht, uninteressant/routine)
* Zentral
* Unabhängig

* Testen gibt feedback was funktioniert
** auch wenn nicht TDD, testmöglichkeiten schaffen
** auch wenn nur main, die ein paar sachen aufruft

wiki.freitagsrunde.org

Übungsmöglichkeiten
(und alte Klausuren, etc.)

Verabschieden: Unsterbliche Worte: aus „Das Literarische Code-Quartett“

Lest mehr Code!

Denn Lesen macht schlau
Was man lesen kann ist gut
was man nicht lesen kann ist schlecht

Programmiert so, das man es lesen kann!