Funktionales TDD

Alles gleich oder alles anders?

Johannes Link

johanneslink.net

Softwaretherapeut

"In Deutschland ist die Bezeichnung Therapeut allein oder ergänzt mit bestimmten Begriffen gesetzlich nicht geschützt und daher kein Hinweis auf ein erfolgreich abgeschlossenes Studium oder auch nur fachliche Kompetenz."

Quelle: Wikipedia

Funktionale Programmierung?

Notwendig. Hilfreich. Zusätzliche Perspektive.

Notwendig

- Pure functions
- Higher order functions
- Immutable data structures

(Sehr) Hilfreich

- Funktionen als Top-Level-Elemente
- Anonyme Funktionen aka Lambdas
- Untypisierte, flexible Datentypen
- Syntax mit wenig Zeremonie
- Pattern Matching
- Tail-Recursion (oder gar Tail-Call)-Optimierung

(Sehr) Hilfreich

- Funktionen als Top-Level-Elemente
- Anonyme Funktionen aka Lambdas
- Untypisierte, flexible Datentypen
- Syntax mit wenig Zeremonie
- Pattern Matching
- Tail-Recursion (oder gar Tail-Call)-Optimierung

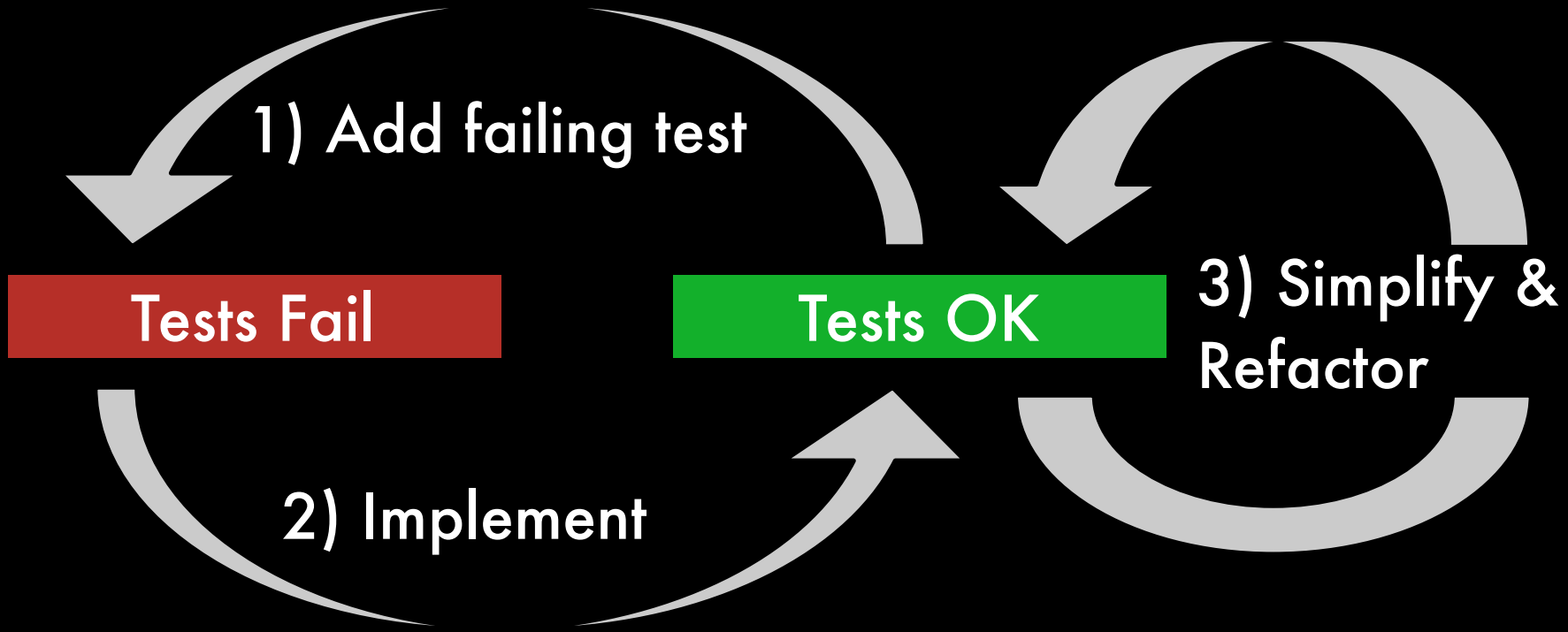
Zusätzliche Perspektiven

- Klare Trennung von Pure und Nicht-Pure
- Algebraisches Typsystem
- Lazy Evaluation

TDD?

- Entwickler schreiben automatisierte Tests während sie programmieren
- Die Tests werden vor dem zugehörigen Produktionscode geschrieben
- Design findet in kleinen Schritten und ständig statt

Test - Code - Refactor



Gut getesteter Code

- Ausreichend Tests für Vertrauen
- Tests sind wartbar und verständlich
- Schwerpunkt liegt bei Micro-Tests
- Entkoppeltes Design

042 : 051

+

A

B

-

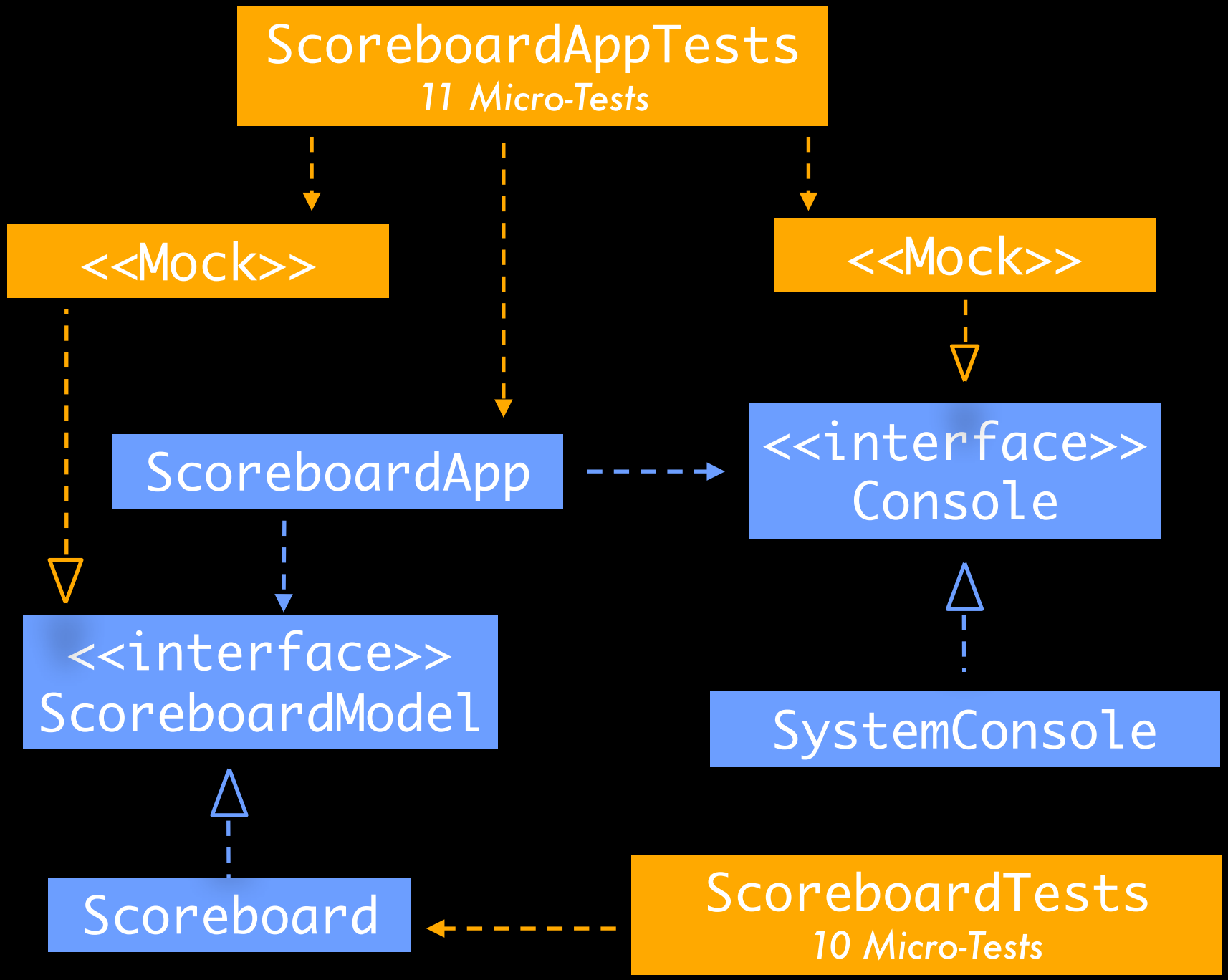
R

```
SCOREBOARD started.
000:000
a
Team A selected
+
001:000
+
002:000
b
Team B selected
+
002:001
-
002:000
c
000:000
x
SCOREBOARD stopped.
```

Java Scoreboard

Objektorientiertes Inside-Out TDD

```
public class ScoreboardAppTests {
   private ScoreboardApp app;
   @Test
   void initialScoreIs000to000() {
     Console console = mock(Console.class);
     app = new ScoreboardApp(
         new Scoreboard(),
         console
     );
     app.run();
     verify(console).println("000:000");
   }
}
```


ScoreboardAppTests
11 Micro-Tests
<<Mock>>
<<Mock>>
ScoreboardApp
<<interface>>
Console
<<interface>>
ScoreboardModel
SystemConsole
Scoreboard
ScoreboardTests
10 Micro-Tests

```
public class ScoreboardAppTests
  @Nested class ScorePrinting
    void initialScoreIsTakenFromScoreboard()
    void scoreIsPrintedIn000Format()
    void moreThan3DigitsAreLeftAlone()
  @Nested class Commands
    void commandASelectsTeamA()
    void commandBSelectsTeamB()
    void commandPlusIncrementsScoreboard()
    void commandMinusDecrementsScoreboard()
    void commandRResetsScoreOnScoreboard()
    void commandsAreTrimmed()
    void commandsAreConvertedToLowercase()
    void unknownCommandsAreIgnored()
```

```
public class ScoreboardTests
  void initialScoreIs000to000()
  void initiallyNoTeamIsSelected()
  void selectingTeamAMakesItSelected()
  void selectingTeamBMakesItSelected()
  void lastSelectCallIsRelevant()
  void incrementIncrementsScoreOfSelectedTeam()
  void decrementDecrementsScoreOfSelectedTeam()
  void whenNoTeamIsSelectedIncrementAndDecrement-
                                        LeaveScoreAsIs()
  void resetScoreSetsScoreTo0to0()
  void noTeamSelectedAfterReset()
```

Typische OO Tests

Zum Verifizieren von Seiteneffekten und Zustand benötigen wir Stubs und Mocks.

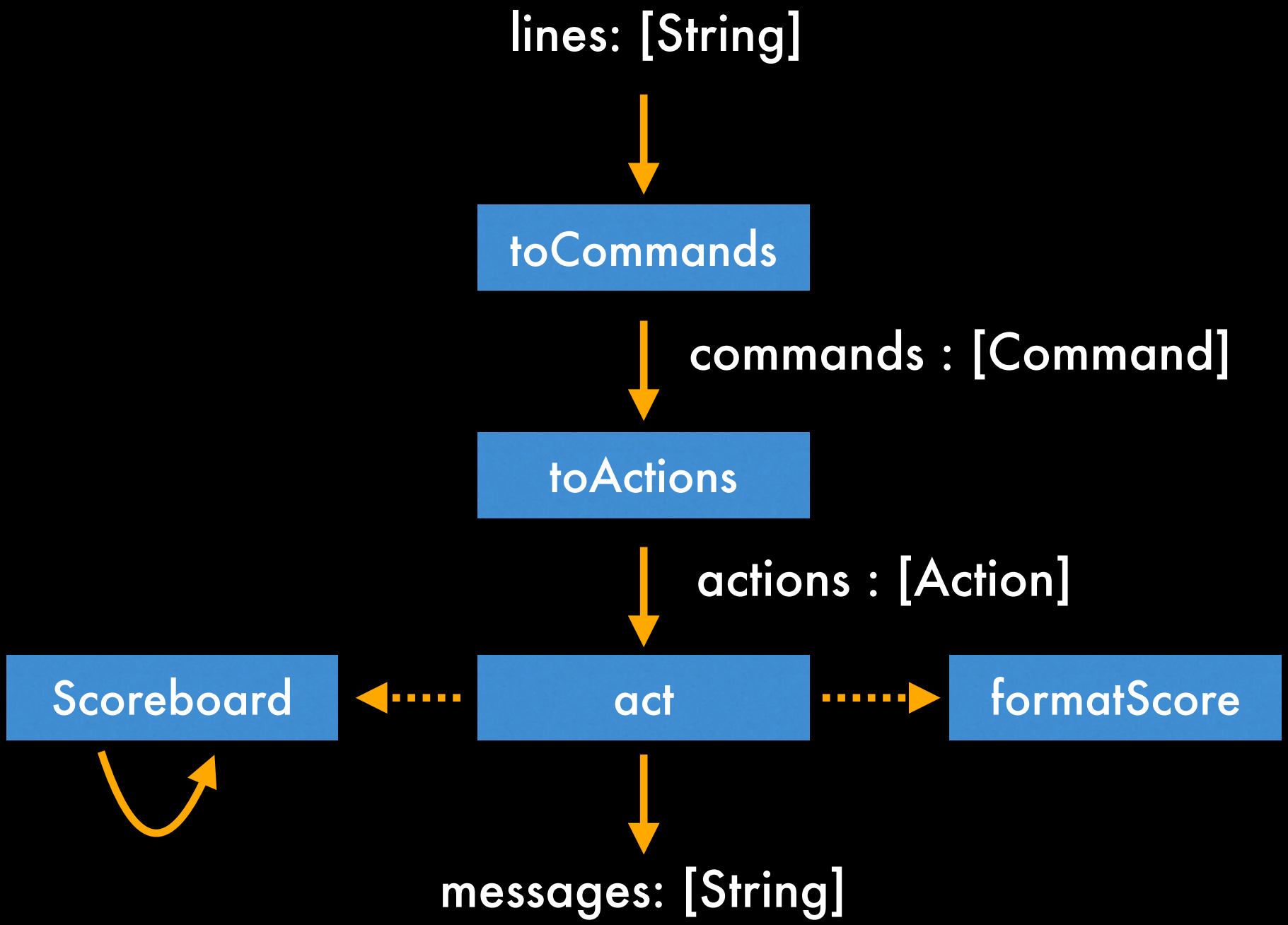
Solche Tests sind oft schwer verständlich und vermitteln das Gefühl, dass man die Implementierung testet.

Haskell Scoreboard

Funktionales Inside-Out TDD

```haskell
import Scoreboard
import ScoreboardApp

spec :: Spec
spec = do

  describe "ScoreboardApp.process" $ do
    it "initial score is 000:000" $ do
      process newScoreboard [] `shouldBe` ["000:000"]

  describe "Scoreboard" $ do
    it "current score of new scoreboard is 0 : 0" $ do
      let scoreboard = newScoreboard
      currentScore scoreboard `shouldBe` (0, 0)
```

```
process :: [String] -> [String]

/**
 * @param commandLines List of entered commands
 * @return List of console messages to print
 */
List<String> process(List<String> commandLines)
```

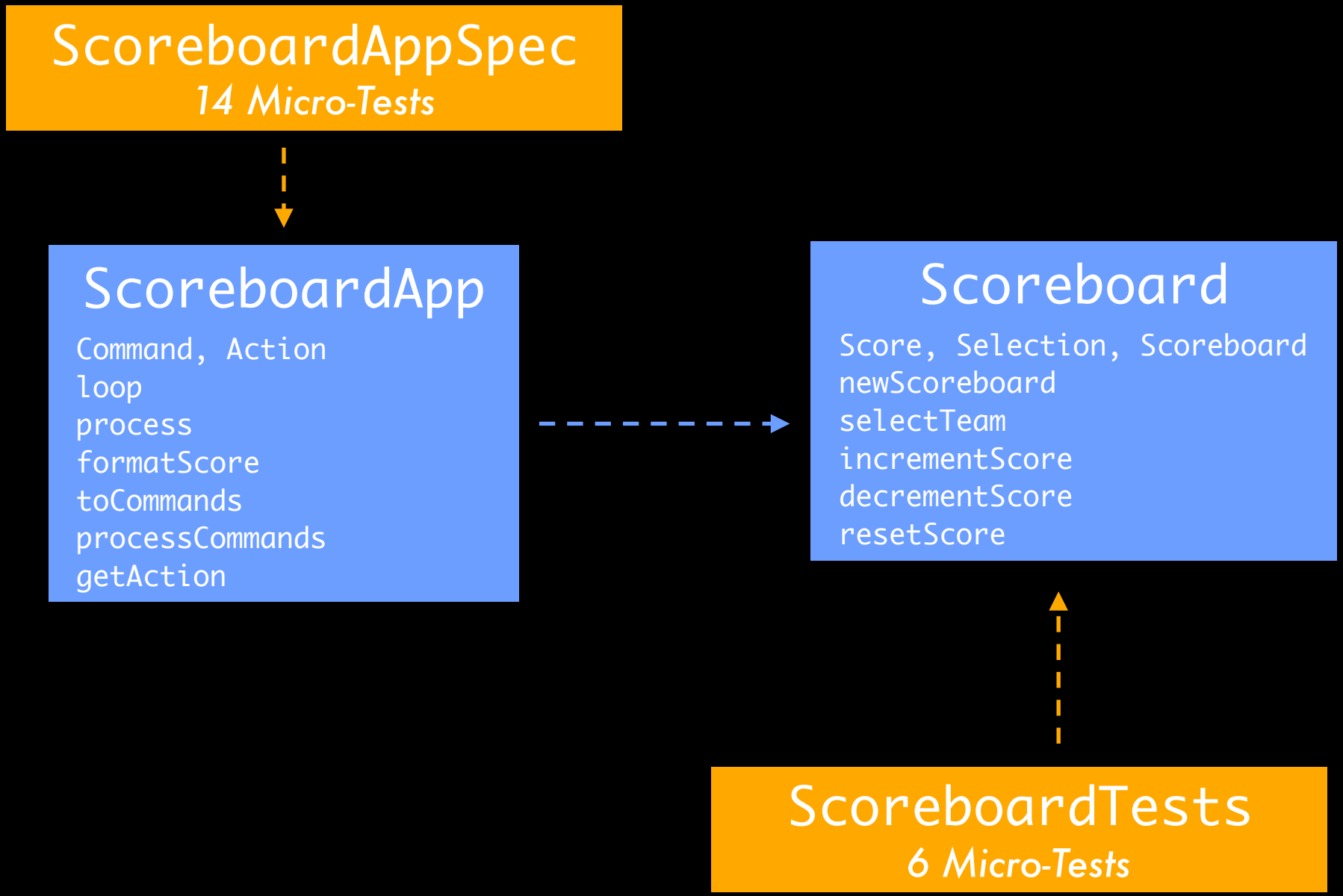

lines: [String]
toCommands
commands : [Command]
toActions
actions : [Action]
Scoreboard
act
formatScore
messages: [String]

Verifiziere Processing Steps

```
describe "ScoreboardApp.toCommands" $ do

    it "lines are converted to commands" $ do
       toCommands ["a", "b", "+", "-", "r", "x"] `shouldBe`
          [SelectA, SelectB, Increment, Decrement, ResetBoard, Exit]

    it "lines are sanitized before conversion" $ do
       toCommands ["   a   ", "B"] `shouldBe` [SelectA, SelectB]

    it "unknown commands are skipped" $ do
       toCommands ["a", "z", "ab", "x"] `shouldBe` [SelectA, Exit]
```

Verifiziere Score-Formatierung

```
describe "ScoreboardApp.formatScore" $ do

   it "single digit scores are filled in with zeros" $ do
      formatScore (1, 9) `shouldBe` "001:009"

   it "multi digit scores are filled in if necessary" $ do
      formatScore (11, 999) `shouldBe` "011:999"

   it "more than 3 digits are left alone" $ do
      formatScore (1234, 98765) `shouldBe` "1234:98765"
```


ScoreboardAppSpec
14 Micro-Tests
ScoreboardApp
Command, Action
loop
process
formatScore
toCommands
processCommands
getAction
Scoreboard
Score, Selection, Scoreboard
newScoreboard
selectTeam
incrementScore
decrementScore
resetScore
ScoreboardTests
6 Micro-Tests

Warum funktioniert process?

```
getContents :: IO String        IO<String> getContents()
putStrLn :: String -> IO ()     IO putStrLn(String)

loop :: IO ()
loop = do
  contents <- getContents
  let commandLines = lines contents
  let messages = process newScoreboard commandLines
  mapM_ putStrLn messages
```

Lazy IO + Infinite Lists:
Der Input und Output erfolgt nach und nach

Typische Tests für funktionalen Code

- Fast alles sind pure Funktionen, die sich direkt testen lassen.
  - Parametrisierung der FuT mit echten Werten und richtigen Funktionen
  - Nur für "äußere" Funktionen mit IO benötigt man eventuell Mocks

```java
@Test
void incrementIncrementsScoreOfSelectedTeam() {
  scoreboard.setScore(1, 2);
  scoreboard.selectTeamA();
  scoreboard.increment();
  assertScore(2, 2);
  assertTrue(scoreboard.isTeamASelected());

  scoreboard.setScore(1, 2);
  scoreboard.selectTeamB();
  scoreboard.increment();
  assertScore(1, 3);
  assertTrue(scoreboard.isTeamBSelected());
}
```

```haskell
it "incrementing score of selected team" $ do
  let scoreboardA = (Scoreboard (1, 2) TeamA)
  incrementScore scoreboardA `shouldBe` (Scoreboard (2, 2) TeamA)

  let scoreboardB = (Scoreboard (1, 2) TeamB)
  incrementScore scoreboardB `shouldBe` (Scoreboard (1, 3) TeamB)
```

Property Testing

```
describe "Scoreboard Properties" $ do
     it "decrementing should always be possible" $ property $
        prop_decrementing

prop_decrementing :: Scoreboard -> Bool
prop_decrementing scoreboard = scoreA >= 0 && scoreB >= 0 where
     decrementedScoreboard = decrementScore scoreboard
     (scoreA, scoreB) = currentScore decrementedScoreboard
```

```
test/ScoreboardSpec.hs:30:
1) Scoreboard, Scoreboard Properties, decrementing is always possible
     Falsifiable (after 1 test):
     Scoreboard (0,0) TeamA
```

```
CheckResult property =
  Property.def("decrementingShould…Scores")
    .forAll(scoreboards())
    .suchThat(scoreboard -> {
      scoreboard.decrement();
      return scoreboard.scoreTeamA() >= 0
          && scoreboard.scoreTeamB() >= 0;
    }
```

```
java.lang.AssertionError:
Expected satisfied check result but was Falsified (
    propertyName =
    decrementingShouldNeverLeadToNegativeScores,
    count = 28,
    sample = (Scoreboard (-1,0) TeamA)
)
at javaslang.test.CheckResult.assertIsSatisfied(…)
```

Property Testing

- Relativ einfach, wenn es um pure Funktionen geht
- Schwierig, wenn
  - ein Test Seiteneffekte hat
  - im Test Objektzustände manipuliert werden

Typisierung und Testen

- Algebraisches Typsystem macht Wert-Erzeugung und -Übergabe sicherer
  - Weniger Tests für Objektinitialisierung und Zustandsübergänge notwendig
- Dependent Types (z.B. in Idris) erzwingen manche Implementierungen
  - Keine Tests für "erzwungene" Implementierung notwendig?

Was können wir für Java lernen?

- Anwendbare funktionale Patterns:
  - So viel "Immutables" wie möglich
  - So viel pure Funktionen wie möglich
  - So viel totale Funktionen wie möglich
  - Property Testing für pure Funktionen
  - Hexagonale Architektur:
    Seiteneffekte finden ausschließlich Außen statt

Könnten wir das Scoreboard in Java funktional nachbauen?

- Immutable Value-Types:
  Ja, aber umständlich zu handhaben
- Pure Funktionen:
  (Statische) Methoden an
  zustandslosen Objekten / Klassen
- Property Testing:
  Javaslang, junit-quickcheck
- Lazy IO:
  Durch Streams bzw. Reactive Streams simulierbar

Fazit

- TDD funktioniert auch bei funktionalen Programmen
- Tests auf pure Funktionen sind einfacher und Mock-arm
- Wert-Objekte ermöglichen mehr pure Funktionen
- Property Testing ist nur bei puren Funktionen wirklich angenehm
- Ein gutes Typsystem macht manche Tests überflüssig
- Und was verändert
  - REPL-based Development?
  - Type-Driven Development?

Der Code ist dort:

http://github.com/jlink/functional-tdd