Functional TDD

Is TDD redundant with Functional Programming?

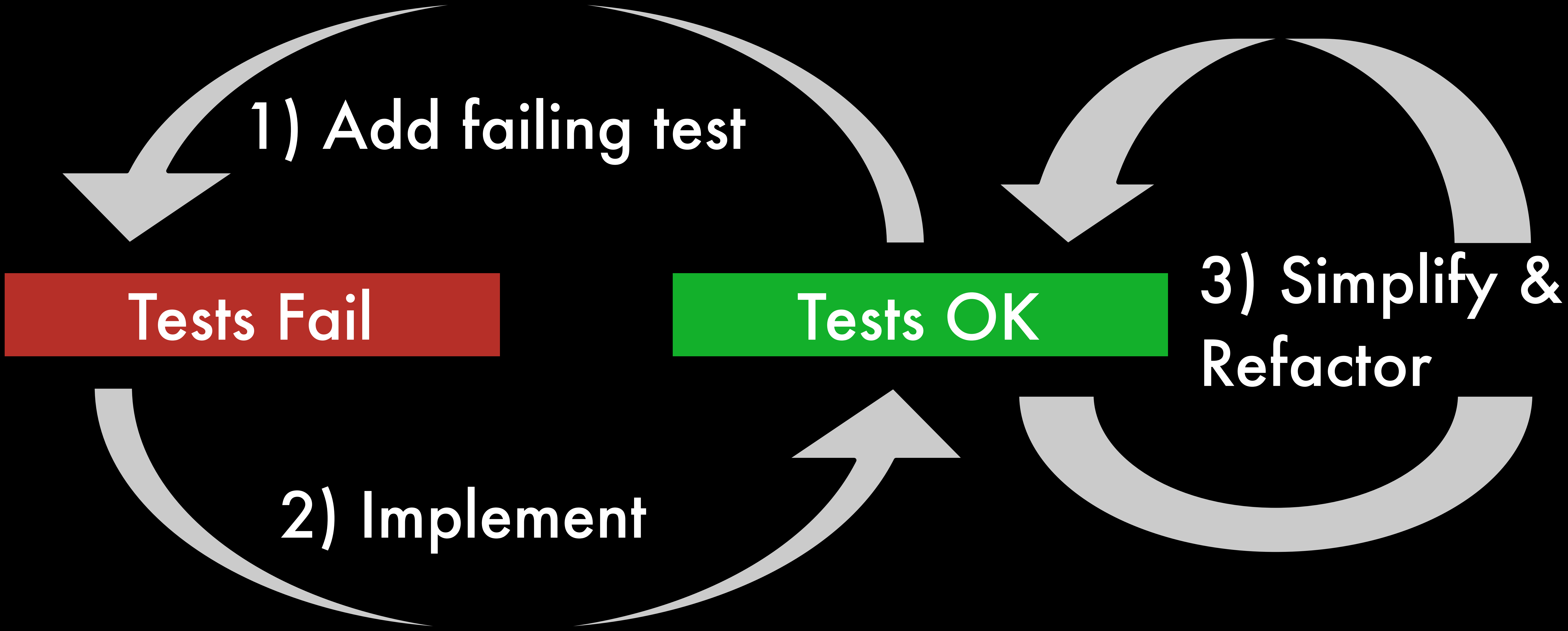
@johanneslink

johanneslink.net

TDD *à la Kent Beck*

- Developers write automated tests while they program
- Tests are written before production code
- Design takes place step by step

Feedback für unser Design:
Test-Code-Refactor



Feedback for functional quality:
Well-tested code

- Enough tests to keep up trust
- Tests are maintainable and understandable
- Most tests are microtests

Functional Programming?

Essential.

Helpful.

Providing additional insights.

Essential

- Pure functions
- Higher order functions
- Immutable data structures

Java?

- Pure functions
- Higher order functions
- Immutable data structures

(Very) helpful

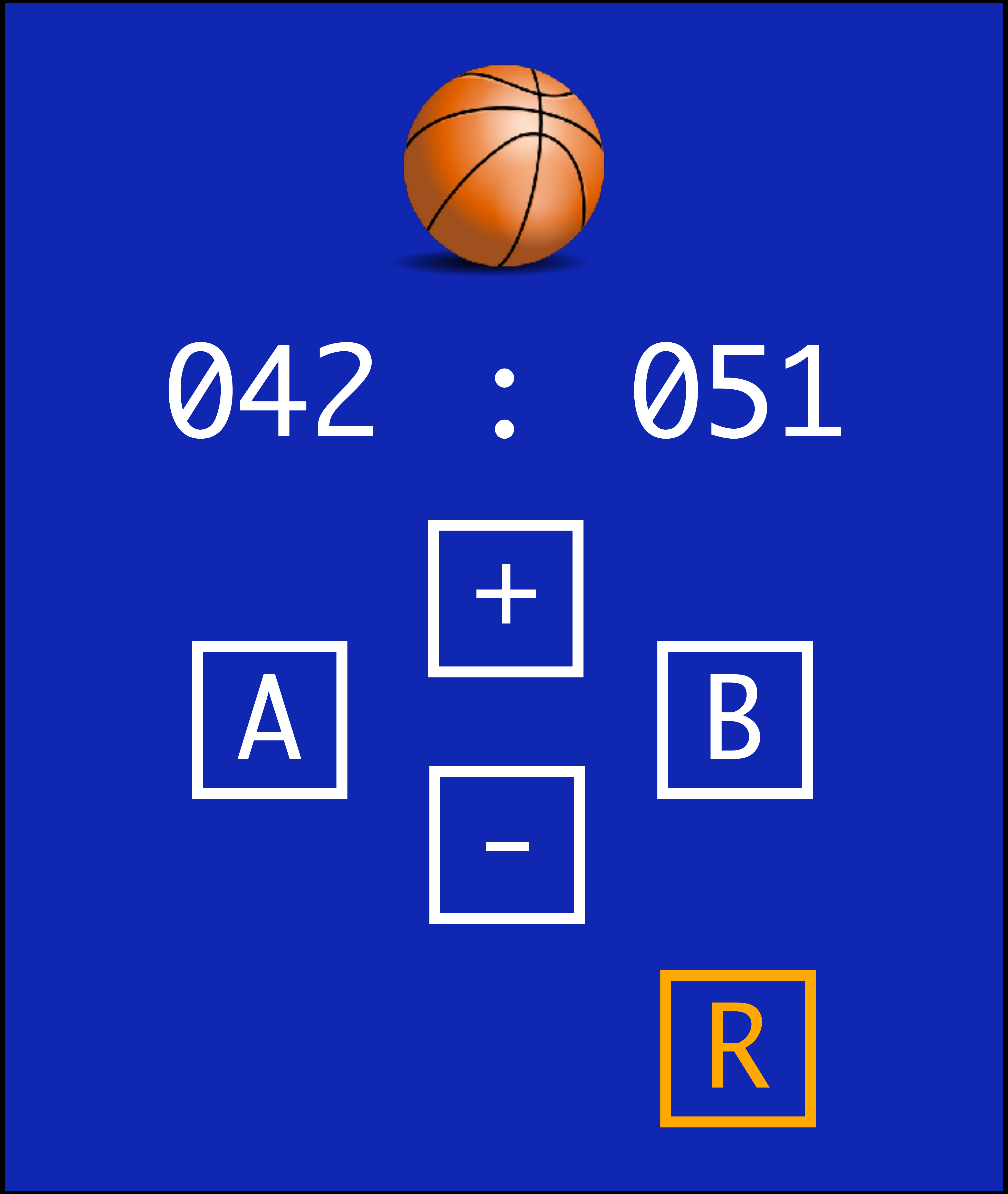
- Functions as top-level constructs
- Anonymous functions aka lambdas
- Type system & syntax friendly to HOFs
- Create flexible data structures on the fly
- Pattern Matching
- Tail recursion optimisation

Java?

- Functions as top-level constructs
- Anonymous functions aka lambdas
- Type system & syntax friendly to HOFs
- Create flexible data structures on the fly
- Pattern Matching
- Tail recursion optimisation

Providing additional insight

- Clear separation of pure and non-pure
- Algebraic type system
- Lazy Evaluation



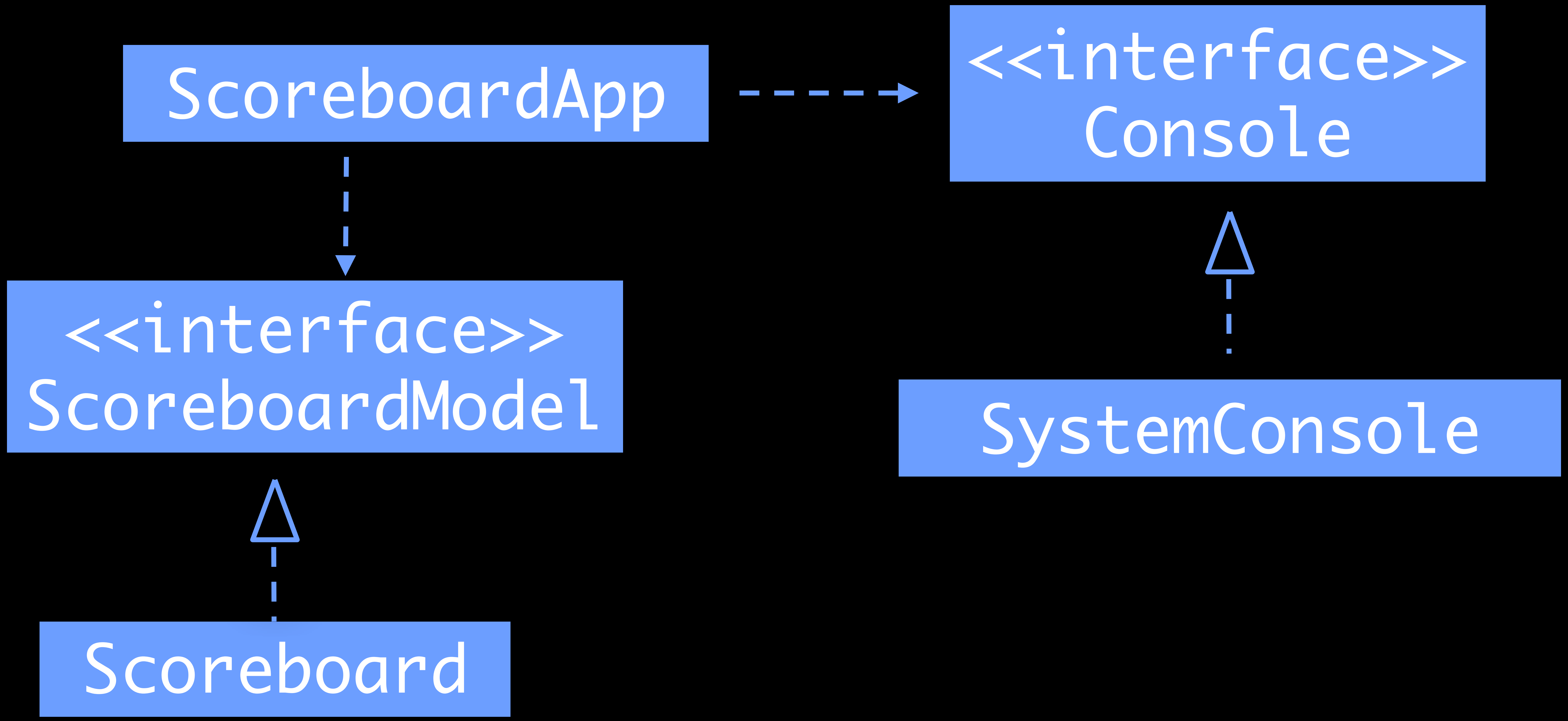
042 : 051

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

Java Scoreboard

Object-oriented inside-out TDD

```java
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");
    }
}
```



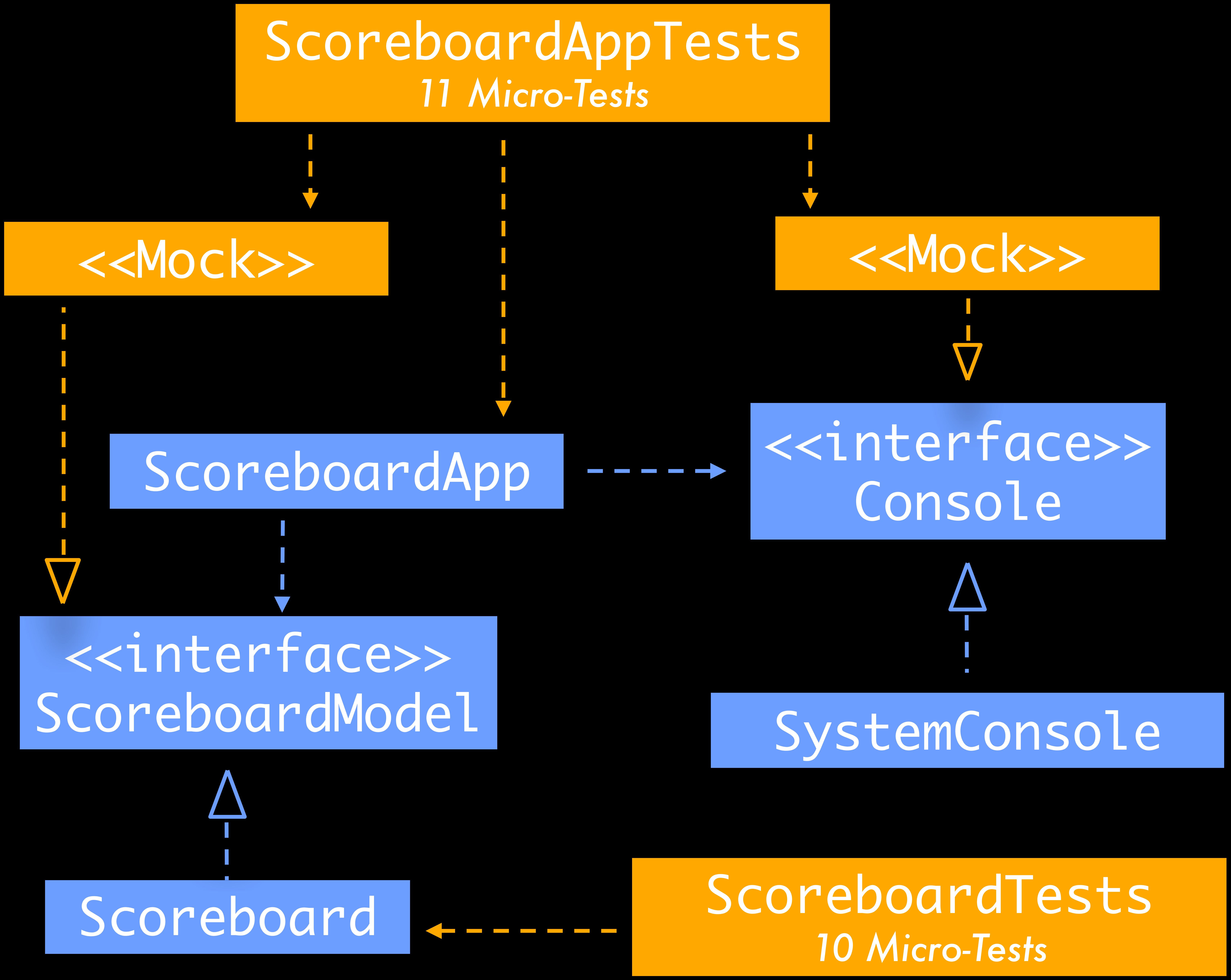
ScoreboardApp

<<interface>>
Console

<<interface>>
ScoreboardModel

SystemConsole

Scoreboard


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()
```

```java
public class ScoreboardTests
 void initialScoreIs000to000()
 void initiallyNoTeamIsSelected()
 void selectingTeamAMakesItSelected()
 void selectingTeamBMakesItSelected()
 void lastSelectCallIsRelevant()
 void incrementIncrementsScoreOfSelectedTeam()
 void decrementDecrementsScoreOfSelectedTeam()
 void whenNoTeamIsSelectedIncrementAndDecrementLeaveScoreAsIs()
 void resetScoreSetsScoreTo0to0()
 void noTeamSelectedAfterReset()
```

Typical OO tests

We need test-doubles to verify state and side-effects.

Such tests are often hard to grasp and give you the feeling of testing the implementation.

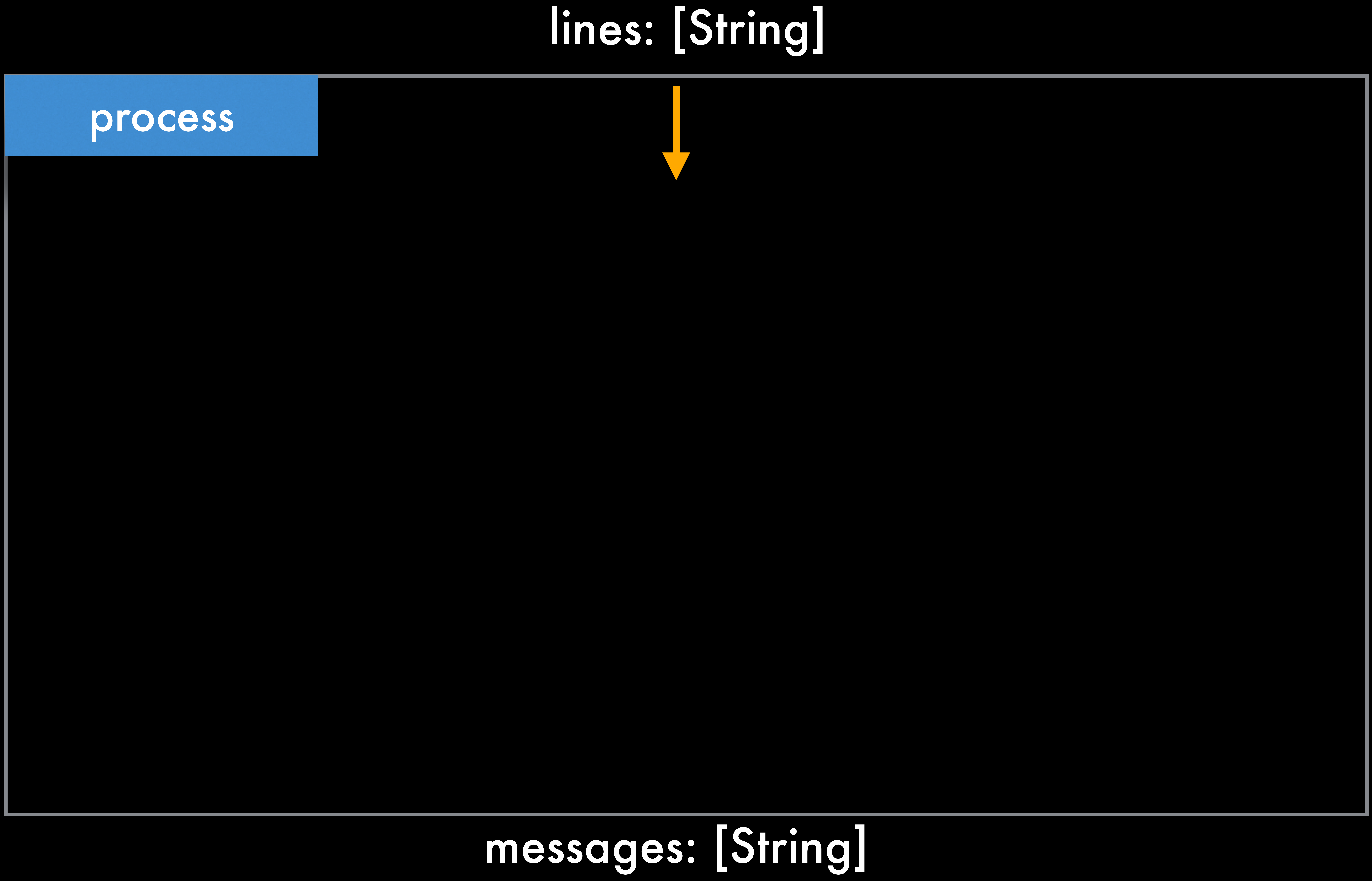
Haskell Scoreboard

Functional 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]

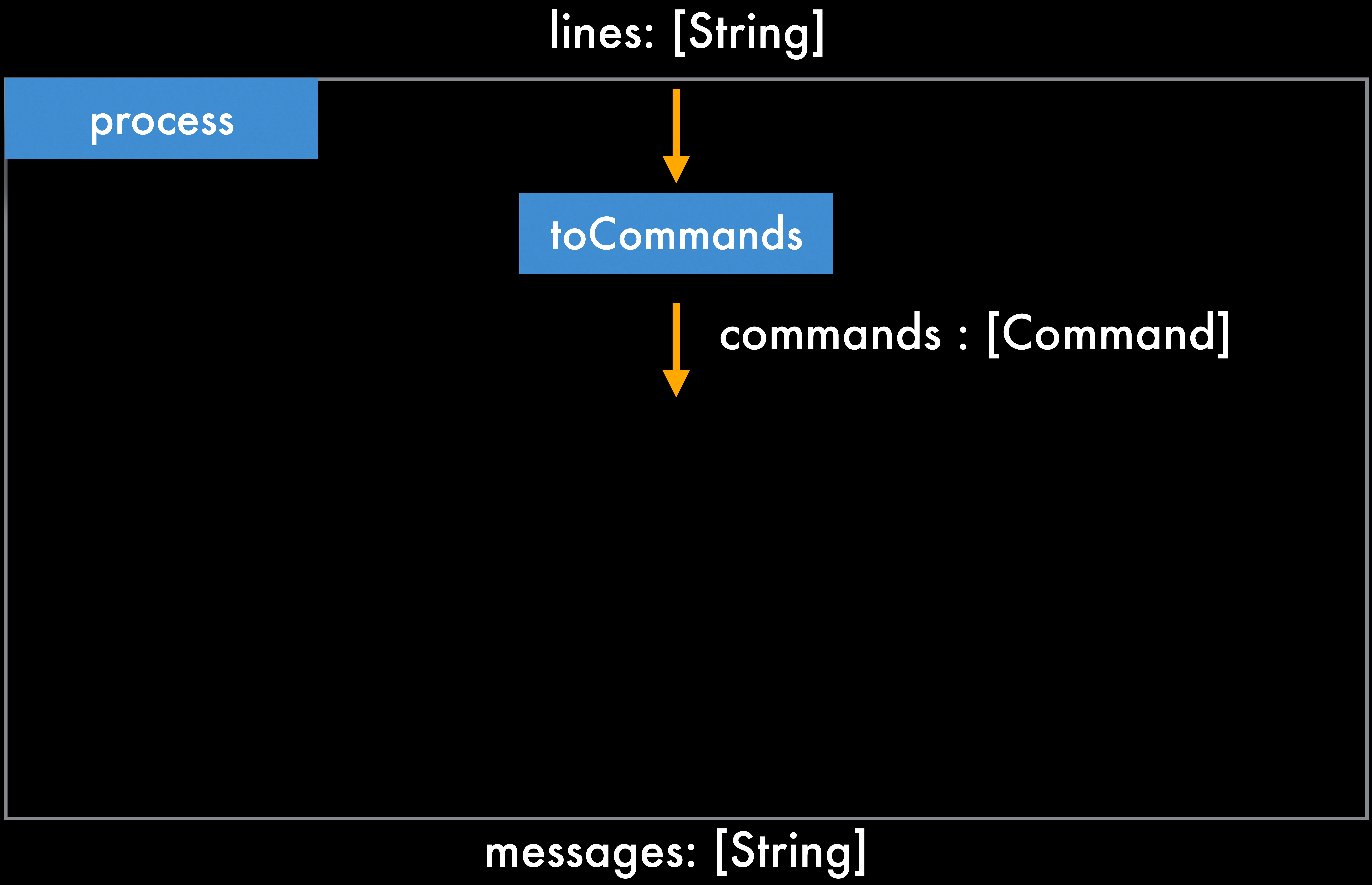
```
/**
 * @param commandLines List of entered commands
 * @return List of console messages to print
 */
List<String> process(List<String> commandLines)
```



lines: [String]

process

messages: [String]



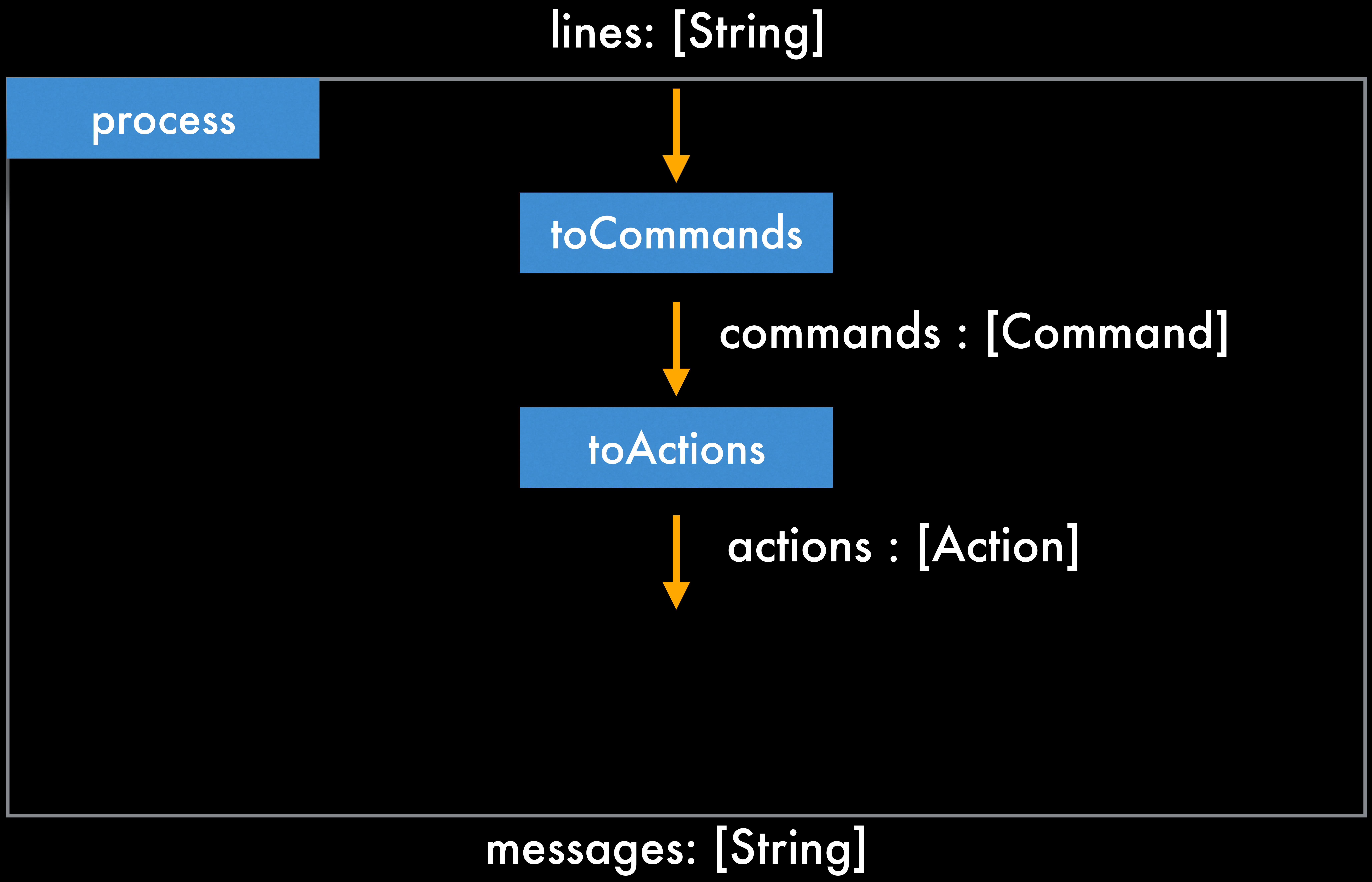
lines: [String]

process

toCommands

commands : [Command]

messages: [String]



lines: [String]

process

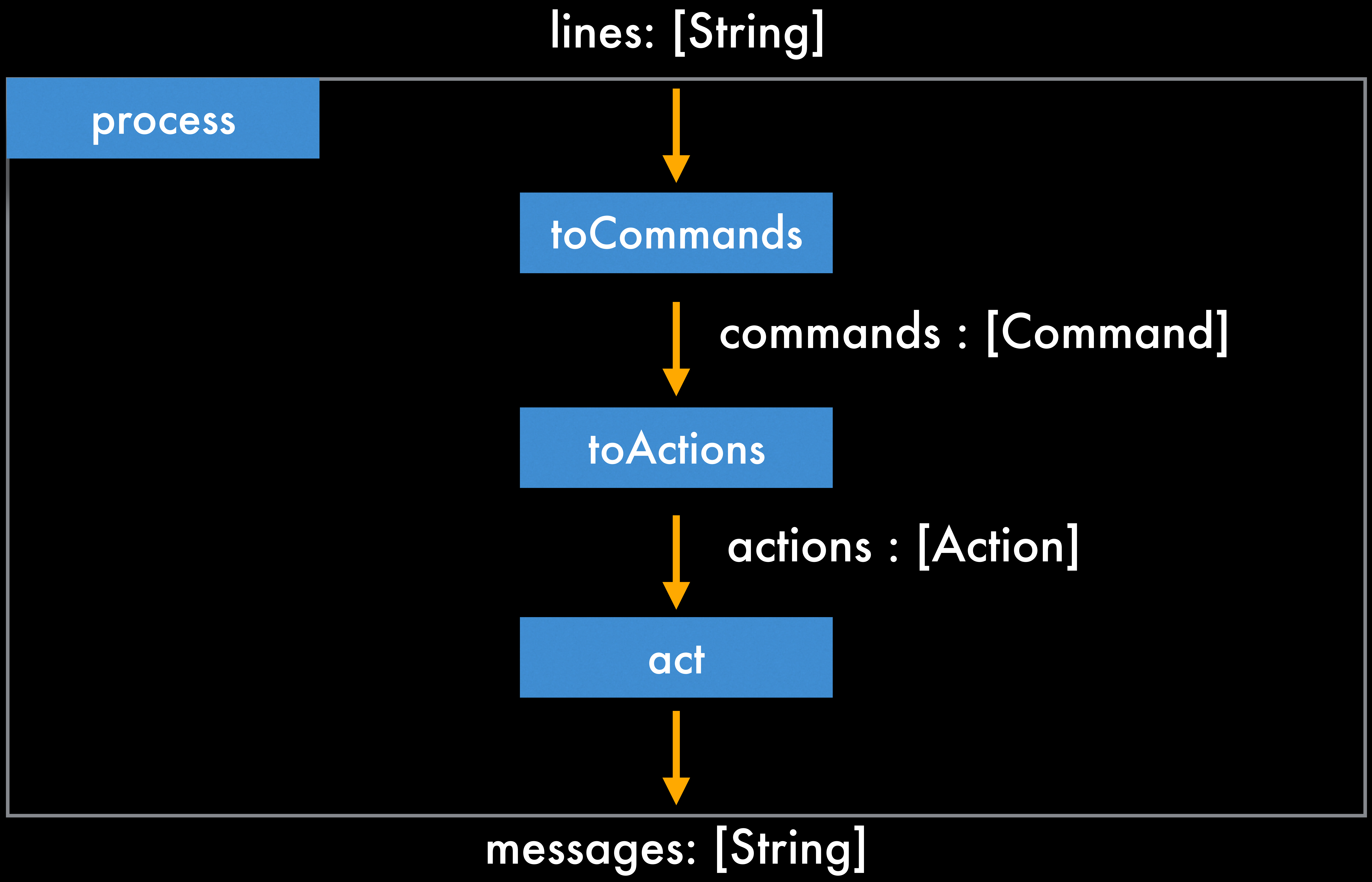
toCommands

commands : [Command]

toActions

actions : [Action]

messages: [String]



lines: [String]

process

toCommands

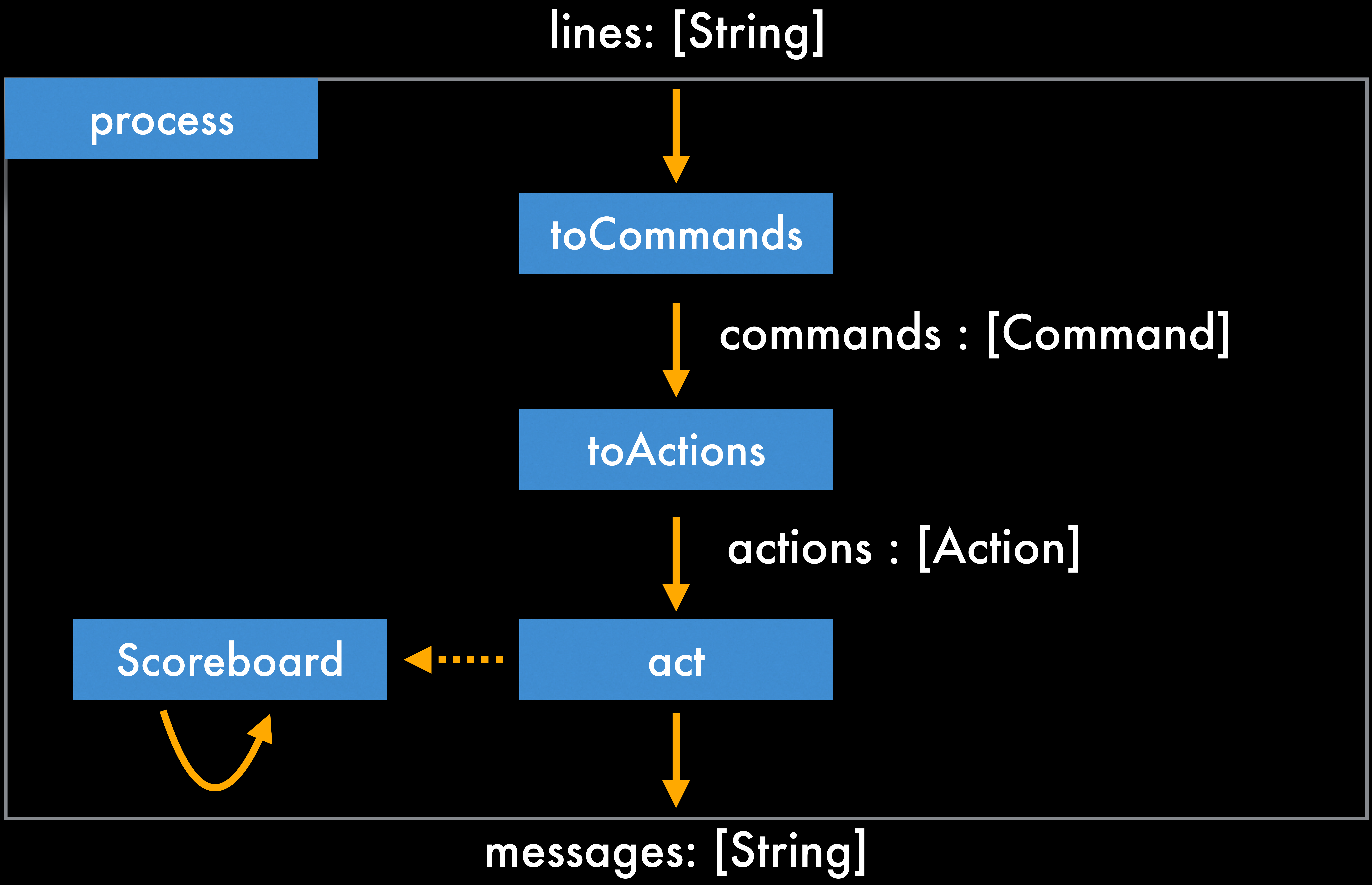
commands : [Command]

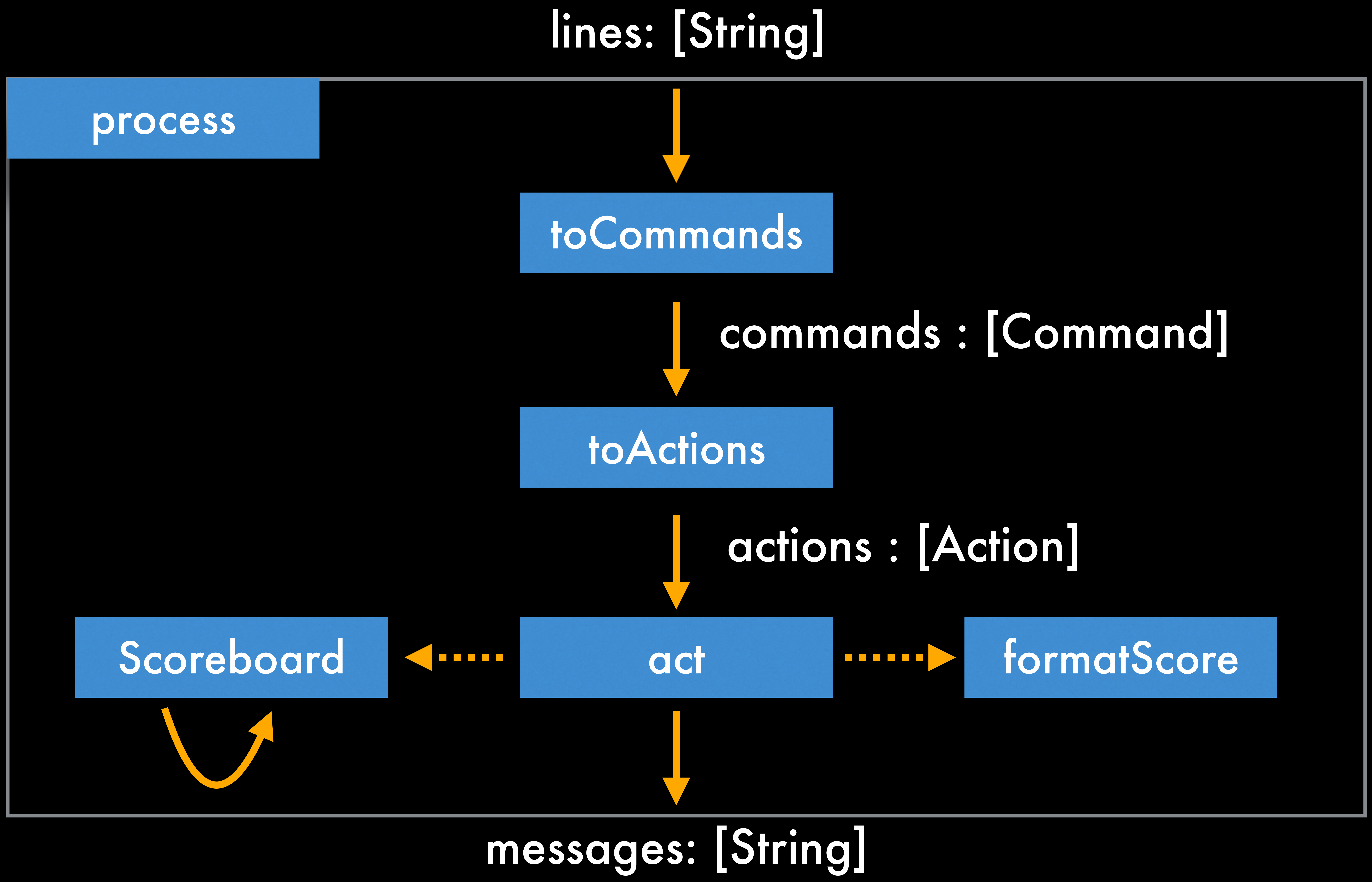
toActions

actions : [Action]

act

messages: [String]





lines: [String]

process

toCommands

commands : [Command]

toActions

actions : [Action]

Scoreboard

act

formatScore

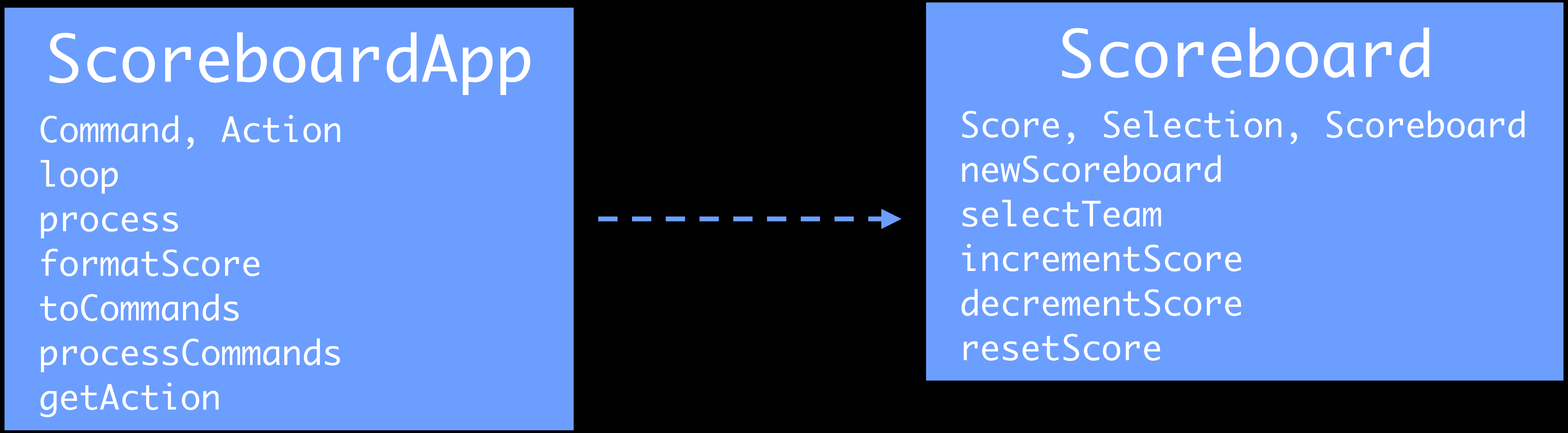
messages: [String]

Verify 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]
```

Verify score formatting

```
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"
```

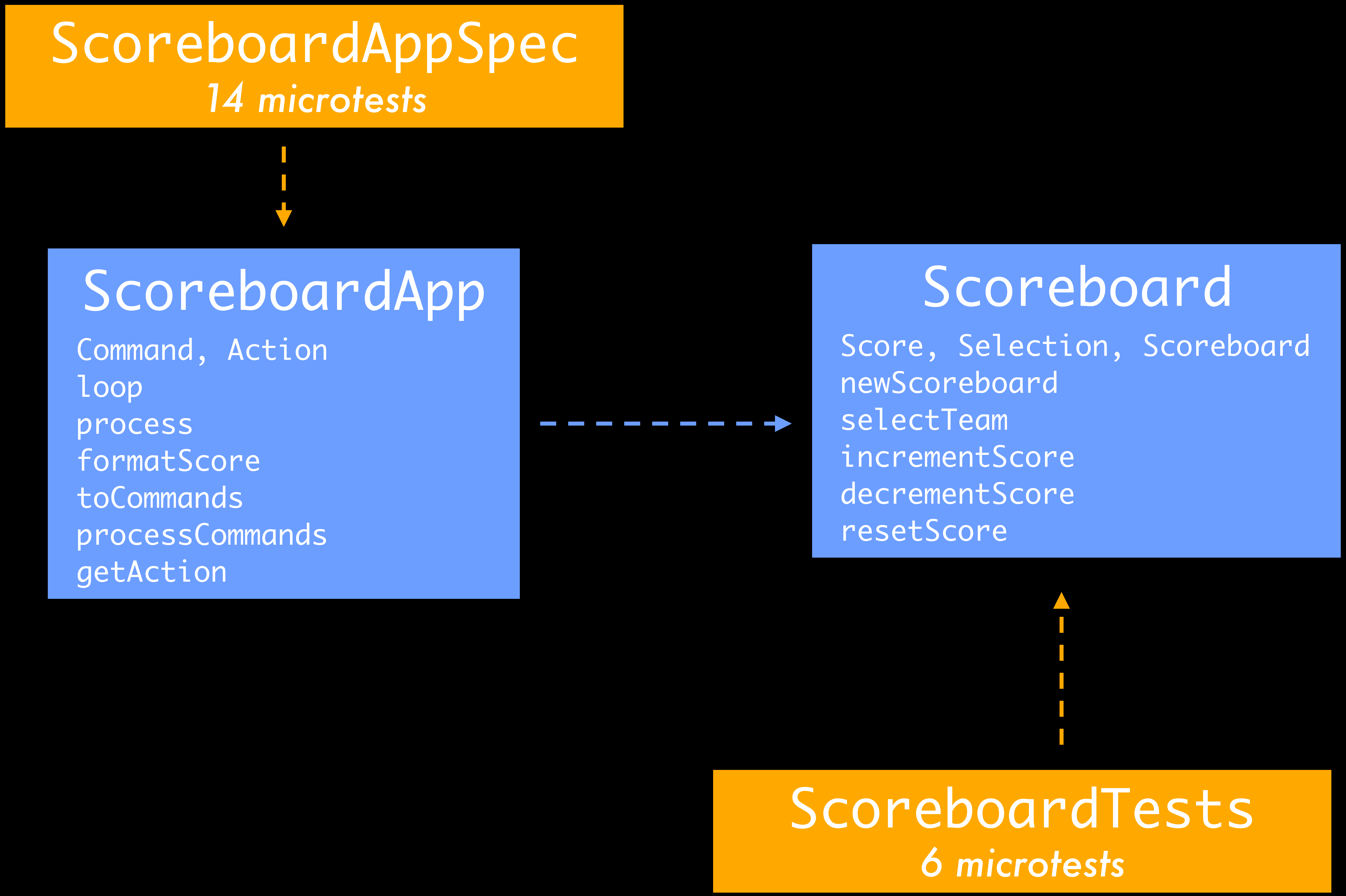


ScoreboardApp

Command, Action
loop
process
formatScore
toCommands
processCommands
getAction

Scoreboard

Score, Selection, Scoreboard
newScoreboard
selectTeam
incrementScore
decrementScore
resetScore



ScoreboardAppSpec

14 microtests

ScoreboardApp

Command, Action
loop
process
formatScore
toCommands
processCommands
getAction

Scoreboard

Score, Selection, Scoreboard
newScoreboard
selectTeam
incrementScore
decrementScore
resetScore

ScoreboardTests

6 microtests

Why does process work?

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

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

Why does process work?

```
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
```

Why does process work?

```
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

Typical tests for functional code

- Mostly everything consists of pure functions, which can be directly tested
  - Parameterise *function under test* with real values and real functions
  - Only for "outer" impure functions you might need 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: Verify universally desired properties of a function

Property Testing: Verify universally desired properties of a function

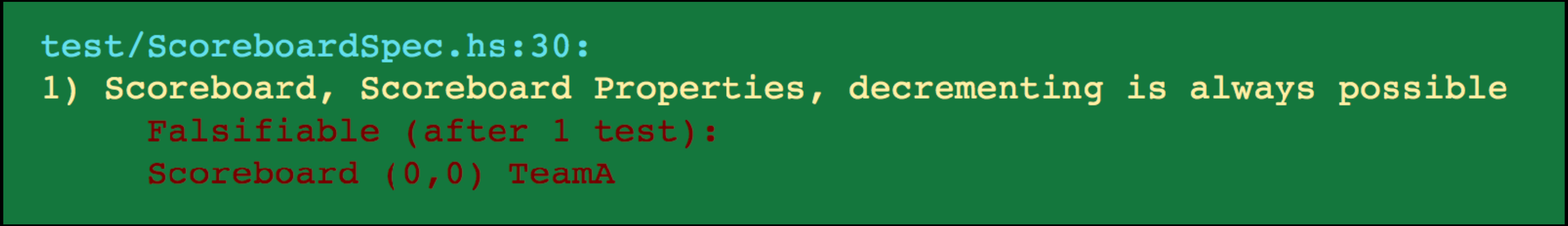
"Trying to decrease a team's score should never result in a negative score"

Quickcheck

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

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

Quickcheck

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

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



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

http://jqwik.net

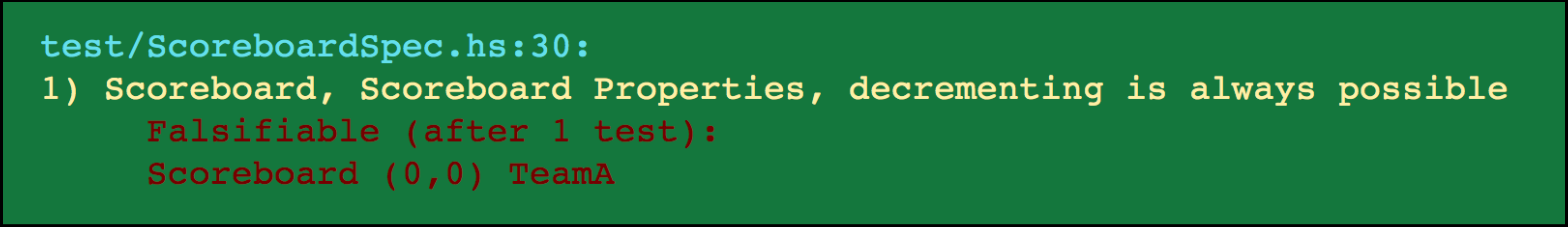
```java
@Property
boolean decrementingIsAlwaysPossible(@ForAll Scoreboard scoreboard) {
   scoreboard.decrement();

   return
      scoreboard.scoreTeamA() >= 0 &&
      scoreboard.scoreTeamB() >= 0;
}
```

http://jqwik.net

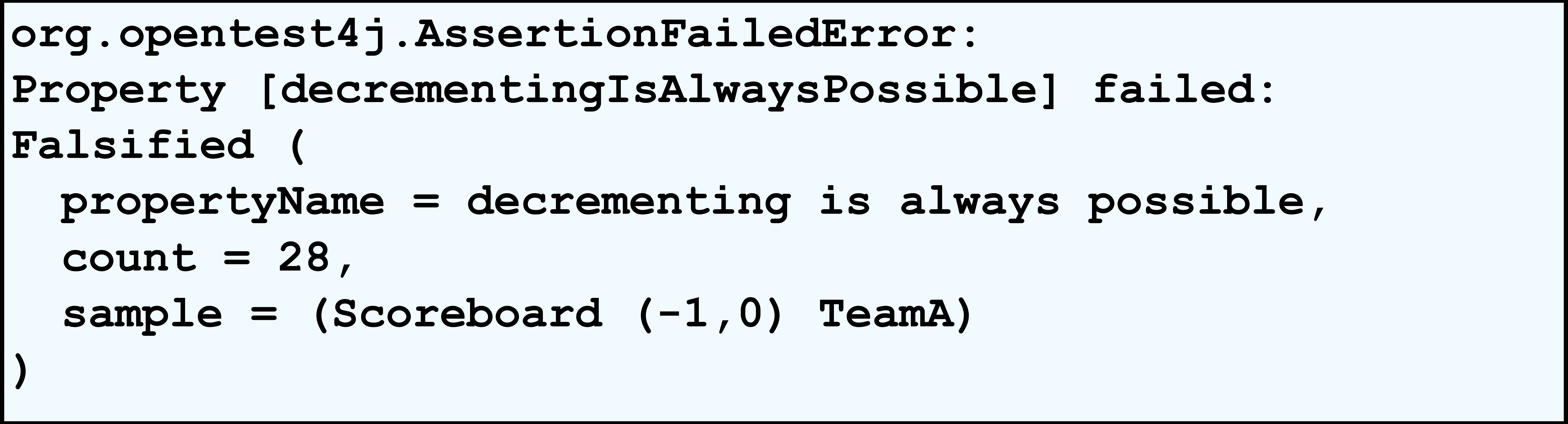
```java
@Property
boolean decrementingIsAlwaysPossible(@ForAll Scoreboard scoreboard) {
   scoreboard.decrement();

   return
      scoreboard.scoreTeamA() >= 0 &&
      scoreboard.scoreTeamB() >= 0;
}
```

```
org.opentest4j.AssertionFailedError:
Property [decrementingIsAlwaysPossible] failed:
Falsified (
    propertyName = decrementingIsAlwaysPossible,
    count = 164,
    sample = (Scoreboard (-1,0) TeamA)
)
```

```
org.opentest4j.AssertionFailedError:
Property [decrementingIsAlwaysPossible] failed:
Falsified (
  propertyName = decrementing is always possible,
  count = 28,
  sample = (Scoreboard (-1,0) TeamA)
)
```



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



```
org.opentest4j.AssertionFailedError:
Property [decrementingIsAlwaysPossible] failed:
Falsified (
  propertyName = decrementing is always possible,
  count = 28,
  sample = (Scoreboard (-1,0) TeamA)
)
```

Types and testing

- Algebraic type system makes creating and passing values safer
  - Fewer tests needed for object initialisation and state changes
- Dependent type (as in Idris) can sometimes enforce the correct implementation
  - No more tests necessary for this "enforced" implementation?

What can we learn for Java?

- Applicable functional patterns:
  - Use "Immutables" whenever possible
  - Use pure functions whenever possible
  - Use total functions whenever possible
  - Use property testing for pure functions
  - Hexagonal architecture
    (aka Ports and Adaptors):
    Impure behaviour only in the outer layers

Could we rebuild the functional solution in Java?

- Immutable value types:
  Possible but involved usage
- Pure functions:
  (Static) methods or variables on stateless objects / classes *with no side-effects*
- Property testing:
  jqwik/javaslang, junit-quickcheck
- Lazy IO:
  Can be simulated through Streams or Reactive Streams

The Code:

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