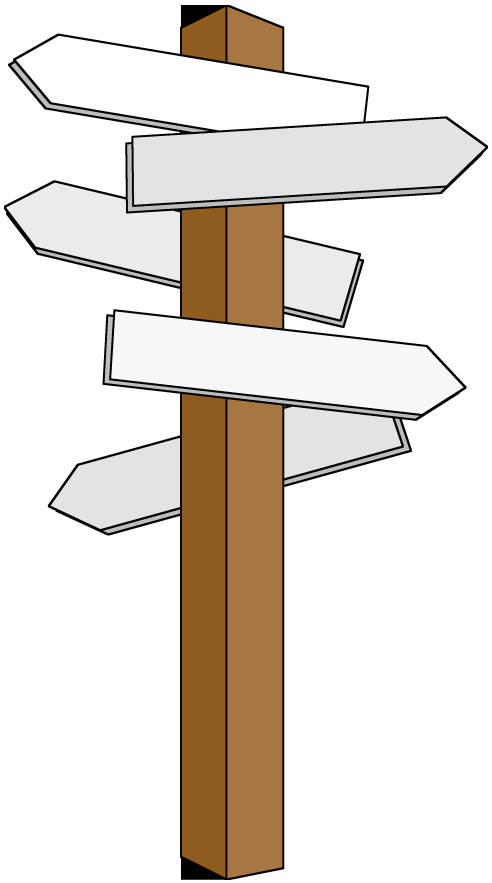
Ontdek de 7 verschillen

© UFS, Inc.

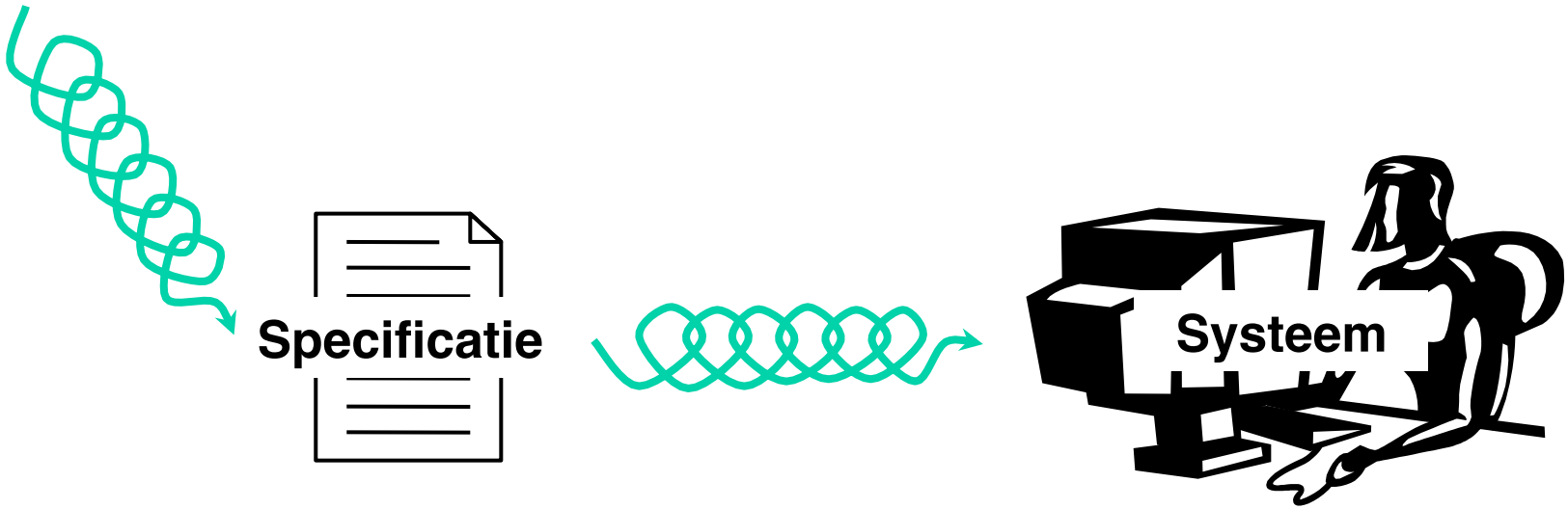
Een oplossing met een foutje …

2. Betrouwbaarheid

- Software Engineering
  - Wat?
  - Bouw vs. Ontwikkel
  - Vereisten
    - Betrouwbaarheid
    - (Aanpasbaarheid & Planning)
- Betrouwbaarheid: TicTacToe
  - specificatie
  - versie 1.0: ontwerp, implementatie & *test*
  - versie 1.1a: ontwerp, implementatie met *slechte test* (manuele controle)
  - versie 1.1a en 1.1: *contracten*



Wat is Software Engineering ?



"Multi-person construction of multi-version software"

- Ploegwerk
- Evolueren of ... uitsterven

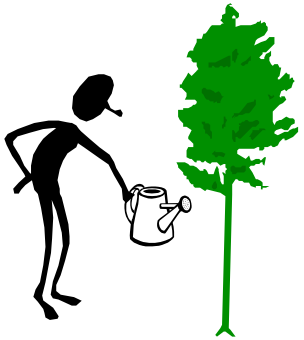
"Bouw" een systeem

- Monolithische systemen
  - banken, verzekeringen, loonberekening
- Programmeerploeg: stricte functie-opdeling
  - Analist, programmeur, ...
- *Organisatie in functie van systeem*
  - Organisatie past zich aan

"Ontwikkel" een systeem

- Modulaire systemen
  - “desktop” systemen, web-applicaties
- Programmeerploeg: losse functie-opdeling
  - Analist + programmeur, ...
- *Systeem in functie van organisatie*
  - Systeem past zich aan

Vereisten



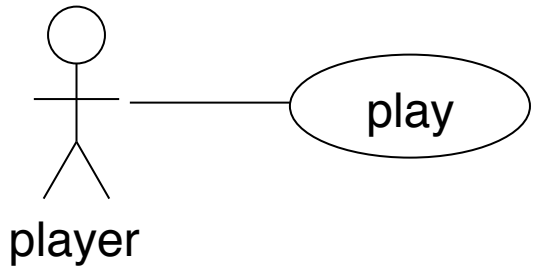
Vereisten

- Betrouwbaarheid
- Aanpasbaarheid
- Planning

Technieken

- Testen + Contracten
- Objectgericht ontwerp
- Tijdsschatting

TicTacToe: Specificatie



Use Case 1: play

- Goal: 2 players play TicTacToe, 1 should win
- Precondition: An empty 3x3 board
- Success end: 1 player is the winner

Steps

1. Two players start up a game
   (First is "O"; other is "X")
2. WHILE game not done
   - 2.1 Current player makes move
   - 2.2 Switch current player
3. Anounce winner

Vuistregel

Keep It Stupidly Simple
(the KISS principle)

Waarom ?

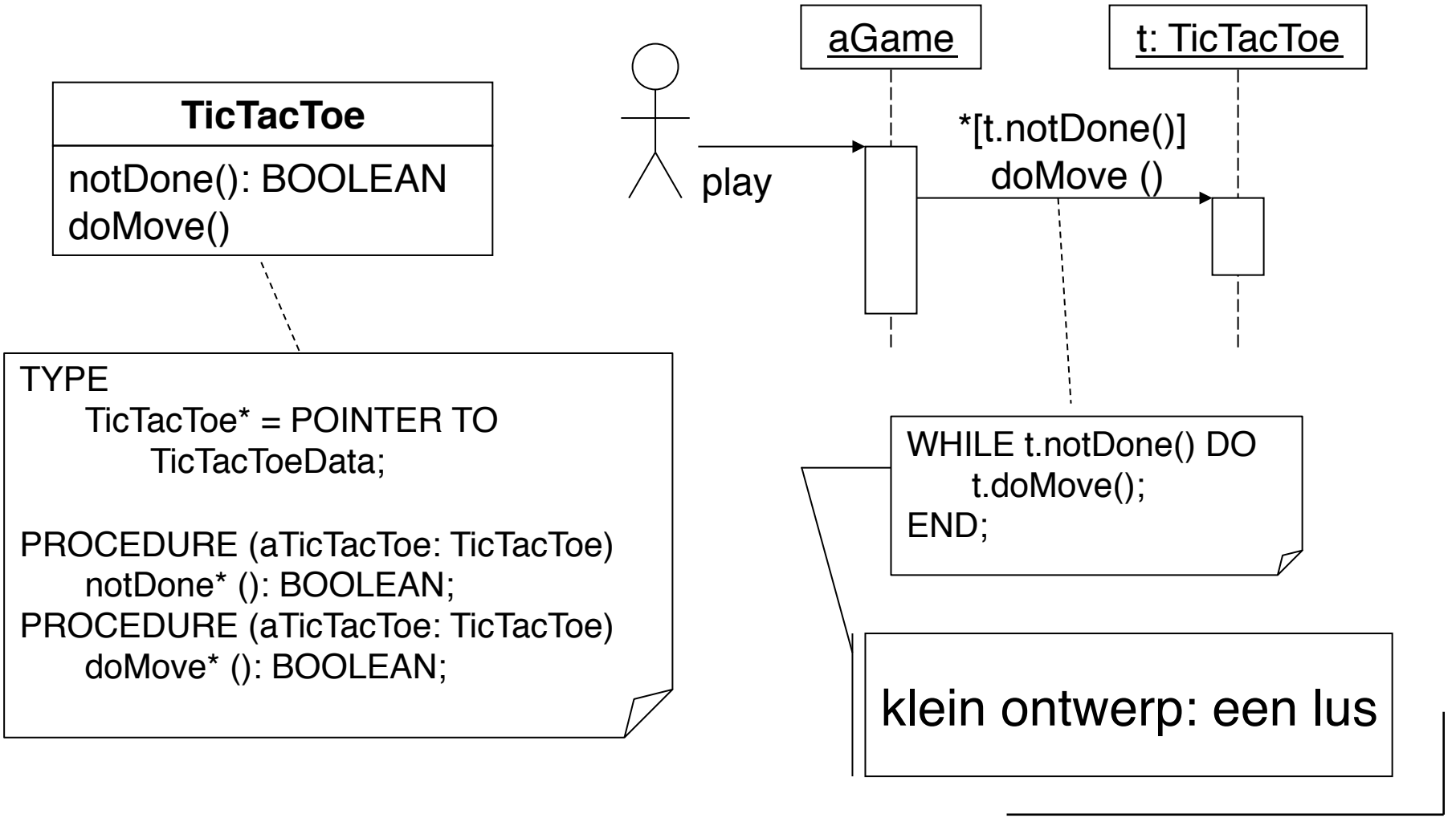
"There are two ways of constructing a software design: one way is to make it so simple that there are obviously no deficiencies, and the other way is make it so complicated that there are no obvious deficiencies."

(C. A. R. Hoare - Turing Award Lecture)

Hoe ?

- Begin zo klein mogelijk
  => laat ontwerp & implementatie langzaam groeien

TTT1.0: Ontwerp



```
TYPE
   TicTacToe* = POINTER TO
   TicTacToeData;
   TicTacToeData* = RECORD
      nrOfMoves: INTEGER;
   END;

PROCEDURE (aTicTacToe:
   TicTacToe) init*;
BEGIN
   aTicTacToe.nrOfMoves
      := 0;
END init;
```

```
PROCEDURE (aTicTacToe: TicTacToe)
   notDone* (): BOOLEAN;
   BEGIN
      RETURN aTicTacToe.nrOfMoves
         < 9;
   END notDone;

PROCEDURE (aTicTacToe: TicTacToe)
   doMove* ();
   BEGIN
      aTicTacToe.nrOfMoves :=
         aTicTacToe.nrOfMoves + 1;
   END doMove;
```

kleine implementatie:
een teller

Vuistregel

Minstens één test per "normaal" scenario

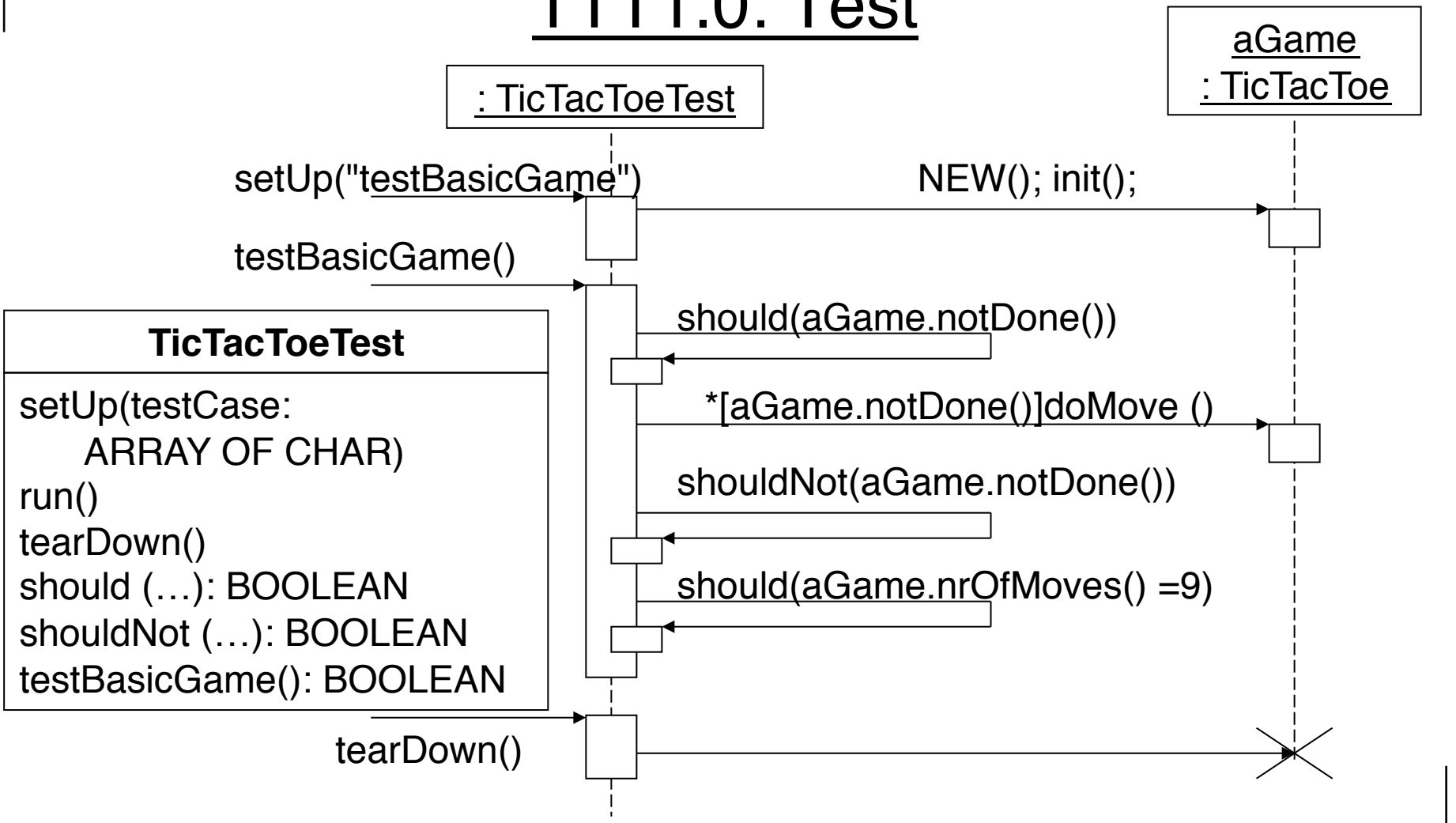
Waarom ?

- Minimum veiligheidsnorm

Hoe ?

- Controleer de tussentijdse toestand via "should" & "shouldNot"
  => onthoud of een test gelukt is of niet

TTT1.0: Test

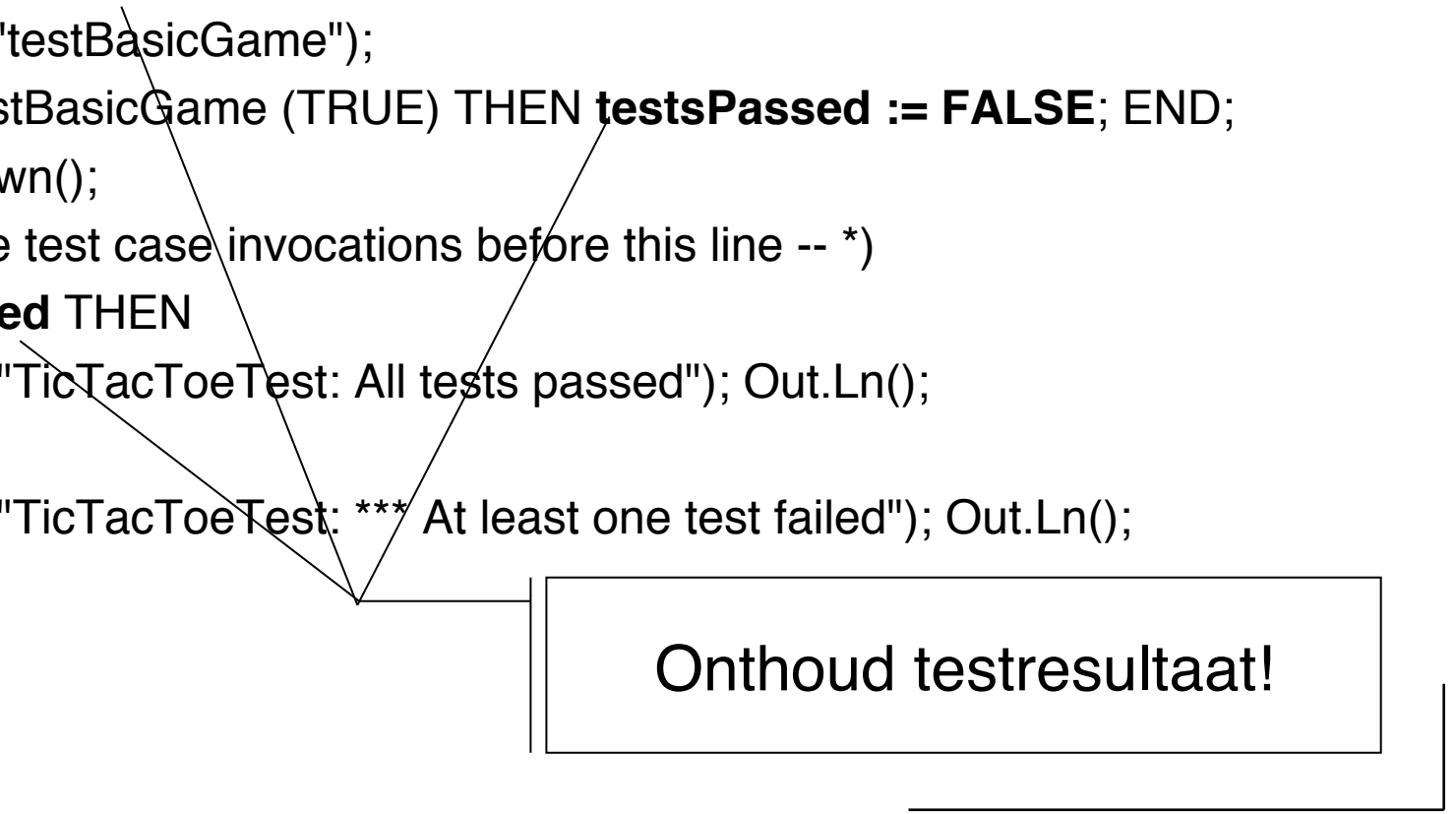

: TicTacToeTest
aGame
: TicTacToe
setUp("testBasicGame")
NEW(); init();
testBasicGame()
should(aGame.notDone())
*[aGame.notDone()]doMove ()
shouldNot(aGame.notDone())
should(aGame.nrOfMoves() =9)
tearDown()
TicTacToeTest
setUp(testCase:
ARRAY OF CHAR)
run()
tearDown()
should (…): BOOLEAN
shouldNot (…): BOOLEAN
testBasicGame(): BOOLEAN

```
PROCEDURE (aTest: TicTacToeTest) testBasicGame*
      (verbose: BOOLEAN): BOOLEAN;
   BEGIN
      IF verbose THEN Out.String(aTest.testCase); Out.Ln; END;
      IF ~ aTest.should(aTest.aGame.notDone(),
         "notDone is FALSE at beginning of game") THEN RETURN FALSE; END;
      WHILE aTest.aGame.notDone() DO
         aTest.aGame.doMove();
      END;
      IF verbose THEN Out.String("... at the end of the loop"); Out.Ln; END;
      IF ~ aTest.shouldNot(aTest.aGame.notDone(),
         "notDone is TRUE at the end of game") THEN RETURN FALSE; END;
      IF ~ aTest.should(aTest.aGame.numberOfMoves() = 9,
         "number of moves at end of game is <> 9") THEN RETURN FALSE;  END;
      RETURN TRUE;
   END testBasicGame;
```

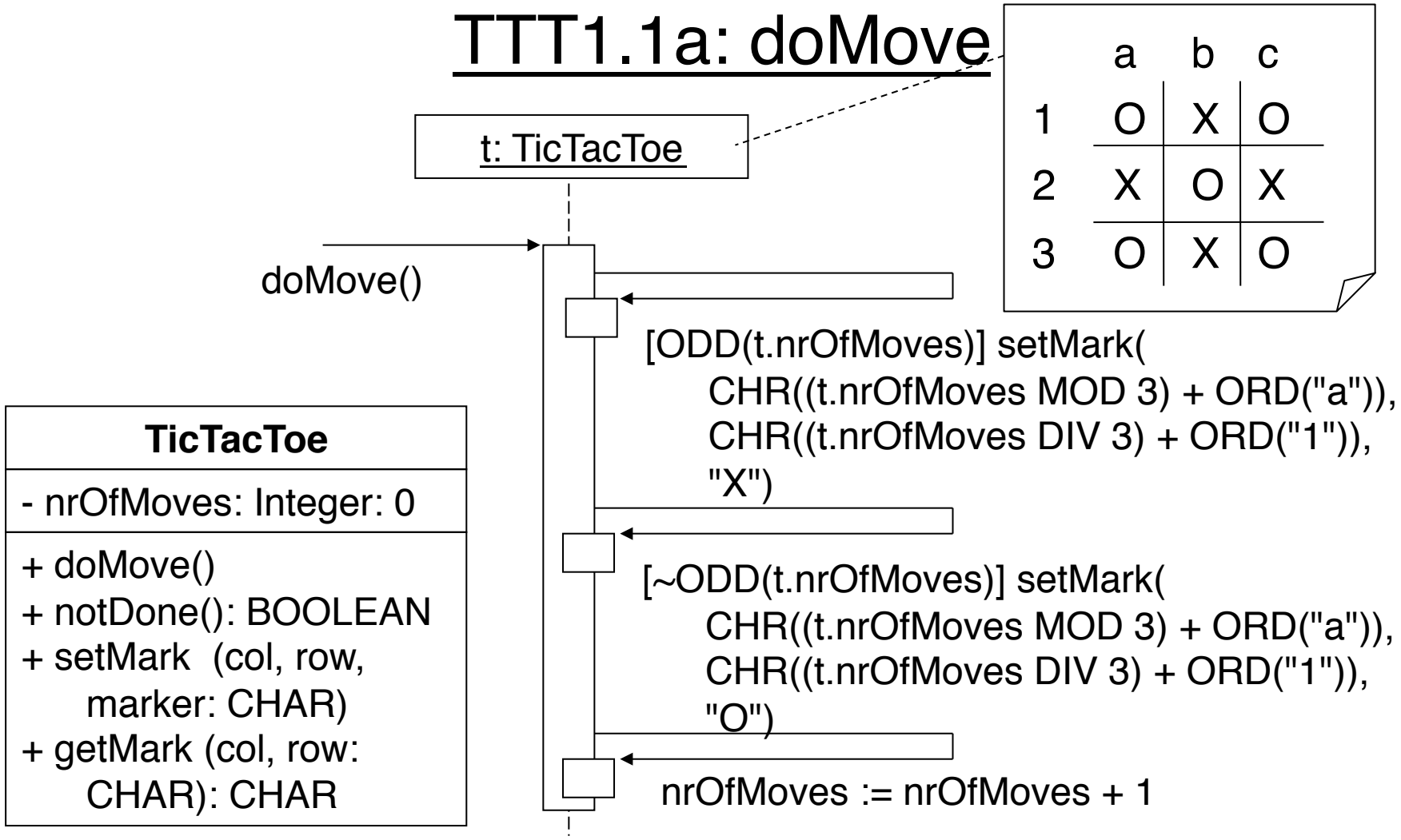
Controleer tussen-tijdse toestand

```
PROCEDURE (aTest: TicTacToeTest) fail (MSG: ARRAY OF CHAR);
   BEGIN
      Out.String("Failure: "); Out.String(aTest.testCase);
      Out.String(" - "); Out.String(MSG); Out.Ln();
   END fail;

PROCEDURE (aTest: TicTacToeTest) should (b : BOOLEAN; MSG: ARRAY OF
   CHAR): BOOLEAN;
   BEGIN
      IF ~ b THEN aTest.fail(MSG); END; RETURN b;
   END should;

PROCEDURE (aTest: TicTacToeTest) shouldNot (b : BOOLEAN; MSG: ARRAY
   OF CHAR): BOOLEAN;
   BEGIN
      IF b THEN aTest.fail(MSG); END; RETURN ~ b;
   END shouldNot;
```

```
PROCEDURE Main*;
   VAR testsPassed: BOOLEAN; aTest: TicTacToeTest;
BEGIN
   NEW(aTest); aTest.init();
   testsPassed := TRUE (* assume that tests will pass *);
   aTest.setUp("testBasicGame");
   IF ~ aTest.testBasicGame (TRUE) THEN testsPassed := FALSE; END;
   aTest.tearDown();
   (* -- add more test case invocations before this line -- *)
   IF testsPassed THEN
      Out.String("TicTacToeTest: All tests passed"); Out.Ln();
   ELSE
      Out.String("TicTacToeTest: *** At least one test failed"); Out.Ln();
   END;
END Main;
```



TTT1.1a: doMove



	a	b	c
1	O	X	O
2	X	O	X
3	O	X	O

TicTacToe

- nrOfMoves: Integer: 0

+ doMove()
+ notDone(): BOOLEAN
+ setMark (col, row, marker: CHAR)
+ getMark (col, row: CHAR): CHAR

```
PROCEDURE (aTicTacToe: TicTacToe) doMove* ();
   VAR row, col, mark: CHAR;
   BEGIN
      col := CHR((aTicTacToe.nrOfMoves MOD 3) + ORD("a"));
      row:= CHR((aTicTacToe.nrOfMoves DIV 3) + ORD("1"));
      IF ODD(aTicTacToe.nrOfMoves) THEN
         mark := "X" ELSE mark := "O"; END;
      aTicTacToe.setMark(col, row, mark);
      aTicTacToe.nrOfMoves := aTicTacToe.nrOfMoves + 1;
END doMove;
```

Ontwerp & Implementatie
groeien langzaam

Vuistregel

Als de functionaliteit groeit,
dan groeit de test mee

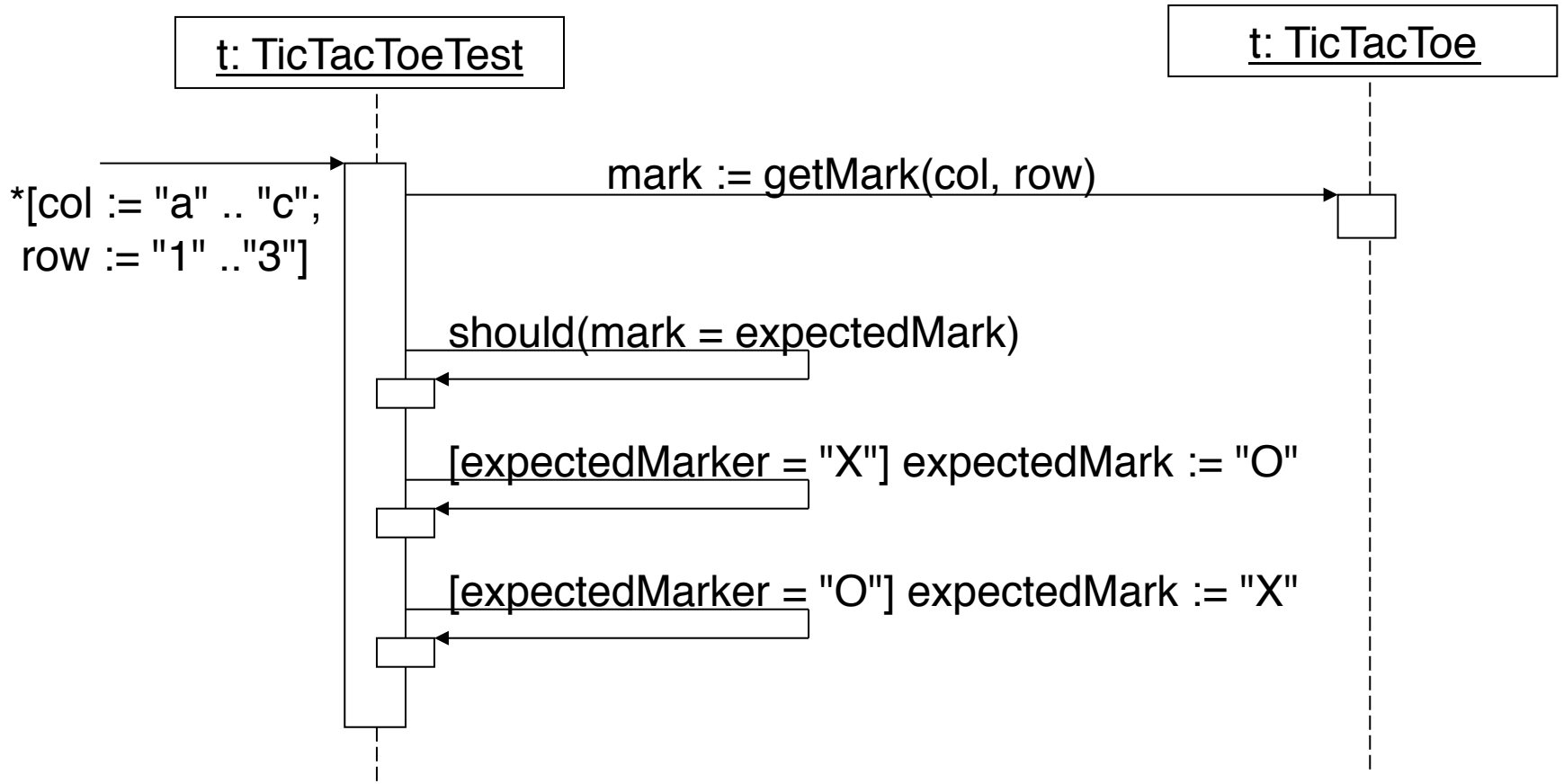
Waarom ?

- Veiligheidsmaatregel: gecontroleerde groei

Hoe ?

- Pas bestaande tests aan
  => extra tussentijds toestanden controleren

TTT1.1a: Test


t: TicTacToeTest
t: TicTacToe
*[col := "a" .. "c";
row := "1" .."3"]
mark := getMark(col, row)
should(mark = expectedMark)
[expectedMarker = "X"] expectedMark := "O"
[expectedMarker = "O"] expectedMark := "X"

```
PROCEDURE (aTest: TicTacToeTest) testBasicGame* (verbose: BOOLEAN):
   BOOLEAN;
...
   WHILE aTest.aGame.notDone() DO aTest.aGame.doMove(); END;
   expectedMark := "O";
   FOR col := ORD("a") TO ORD("c") DO
      FOR row := ORD("1") TO ORD("3") DO
         IF verbose THEN ... END;
         mark := aTest.aGame.getMark(CHR(col), CHR(row));
         IF ~ aTest.should(mark = expectedMark,
            "mark is not what expected") THEN RETURN FALSE; END;
         IF expectedMark = "X"
            THEN expectedMark := "O" ELSE expectedMark := "X"; END;
         END;
      END;
...
```

Test groeit mee

Vuistregel

Een test produceert zo weinig mogelijk uitvoer

- "all tests passed" 
- "at least one test failed" 

Waarom ?

- Veel en frequent testen => onmogelijk om uitvoer te controleren

Hoe dan wel ?

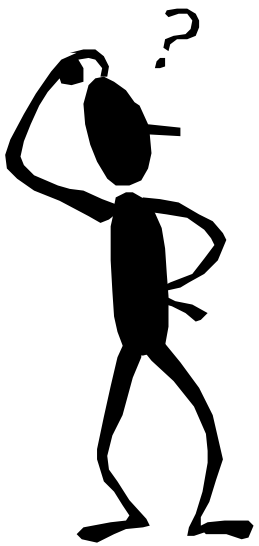
- Tests antwoorden "*all tests passed*" of "*at least one test failed*"
- Tests werken normaal in "silent mode" maar er is ook "verbose"

```
PROCEDURE (aTest: TicTacToeTestBad) testBasicGame*;
…
   Out.String("BEGIN OF GAME: aGame.notDone() = ");
   IF aTest.aGame.notDone() THEN Out.String("TRUE");
      ELSE Out.String("FALSE"); END; Out.Ln;
   WHILE aTest.aGame.notDone() DO aTest.aGame.doMove(); END;
   mark := "O";
   FOR col := ORD("a") TO ORD("c") DO
      FOR row := ORD("1") TO ORD("3") DO
         Out.Char(CHR(col)); Out.Char("-");Out.Char(CHR(row));Out.Char(":
         Out.Char(aTest.aGame.getMark(CHR(col), CHR(row)));
         Out.String(" =? "); Out.Char(mark);Out.Ln;
         IF mark = "X" THEN mark := "O" ELSE mark := "X"; END;
   END; END;
…
   Out.String("END OF GAME: aGame.numberOfMoves() = ");
   Out.Int(aTest.aGame.numberOfMoves(), 0); Out.Ln;
```

Een slechte test

Manuele Verificatie van Tests

Hoe weet ik of deze uitvoer correct is ?



De vorige testcode produceert volgende uitvoer

```
BEGIN OF GAME: aGame.notDone() = TRUE
a-1:O =? O
a-2:X =? X
a-3:O =? O
b-1:X =? X
b-2:O =? O
b-3:X =? X
c-1:O =? O
c-2:X =? X
c-3:O =? O
END OF GAME: aGame.notDone() = FALSE
END OF GAME: aGame.numberOfMoves() = 9
```

Schrijf *nooit* testcode die manuele verificatie van de uitvoer vereist

```
PROCEDURE (aTest: TicTacToeTest) testBasicGame*
      (verbose: BOOLEAN): BOOLEAN;
   BEGIN
      IF verbose THEN Out.String(aTest.testCase); Out.Ln; END;
      IF ~ aTest.should(aTest.aGame.notDone(),
         "notDone is FALSE at beginning of game") THEN RETURN FALSE; END;
      WHILE aTest.aGame.notDone() DO
         aTest.aGame.doMove();
      END;
      IF verbose THEN Out.String("... at the end of the loop"); Out.Ln; END;
      IF ~ aTest.shouldNot(aTest.aGame.notDone(),
         "notDone is TRUE at the end of game") THEN RETURN FALSE; END;
      IF ~ aTest.should(aTest.aGame.numberOfMoves() = 9,
         "number of moves at end of game is <> 9") THEN RETURN FALSE;  END;
      RETURN TRUE;
   END testBasicGame;
```

Alleen uitvoer in geval van "verbose"

TTT1.1a: Klasse Interface

TicTacToe
- nrOfMoves: Integer: 0
+ doMove() + notDone(): BOOLEAN + setMark (col, row, marker: CHAR) + getMark (col, row: CHAR): CHAR

Zijn alle waardes van type CHAR **legale** col & row ?
    => NEE: "a" .. "c" of "1" .. "3"
Is elke waarde van type CHAR een **legale** marker ?
    => NEE: " ", "X" of "O"
Wat is het **verband** tussen "getMark" en "setMark" ?
    => resultaat "getMark" is hetgeen vroeger gezet werd via "setMark"

Vuistregel

- Controleer de waardes van de argumenten die je binnenkrijgt (PRE)
- Controleer de waarde van het resultaat dat je teruggeeft (POST)

Waarom ?

- Meeste fouten door interpretatieproblemen van de interface

Hoe ?

- Schrijf pre- en postcondities bij elke procedure
  => vlak na BEGIN: ASSERT(<boolean expression>, **100**)
  => vlak voor END: ASSERT(<boolean expression>, **120**)

```
PROCEDURE (aTicTacToe: TicTacToe)
   getMark* (col, row: CHAR): CHAR;
(** Answer the mark on the given position on the board. *)
VAR result: CHAR;
BEGIN
   ASSERT( ("a"<=col) & (col<="c") & ("1"<=row) & (row<="3"), 100);
   result := aTicTacToe.board [ORD(col) - ORD("a"),
                               ORD(row) - ORD("1")];
   ASSERT(("X" = result) OR ("O" = result) OR (" " = result), 120);
   RETURN result;
END getMark;
```

- Constantes
  => WatSon

Gebruik een "assertion"
- aborteert het programma
  => fout terug te vinden
- compileeroptie
  => geen redundante controles

```
PROCEDURE (aTicTacToe : TicTacToe) getMark (col, row: CHAR): CHAR;
(* Answer the mark on the given position on the board. *)
   (* precondition (100):   ("a"<=col) & (col<="c") & ("1"<=row) & (row<="3") *)
   (* postcondition (120):("X" = result) OR ("O" = result) OR (" " = result) *)
```

<u>WatSon uitvoer</u>

precondities constante in [100 ... 109]

(& invarianten constante in [110 ... 119])

& postcondities constante in [120 ... 129]

- deel van de klasse-interface
  => CONTRACT met de buitenwereld
- automatisch gegenereerd
  => steeds synchroon met de code

Vuistregel

Contracten gebruiken alleen publieke (geëxporteerde) declaraties

Waarom ?

- contract verifieerbaar door externe modules

Hoe ?

- Private (nt. geëxporteerde) declaraties extern niet toegankelijk
  => Denk goed na over wat je pre- & postcondities nodig hebben

```
PROCEDURE (aTicTacToe: TicTacToe) setMark* (col, row, mark: CHAR);
   (* Mark the given position on the board. *)
   BEGIN
      ASSERT( ("a"<=col) & (col<="c") & ("1"<=row) & (row<="3"), 100);
      ASSERT(("X" =mark) OR ("O" = mark) OR (" " = mark), 100);
      aTicTacToe.board [ORD(col) - ORD("a"), ORD(row) - ORD("1")] := mark;
      ASSERT(aTicTacToe.board [ORD(col) - ORD("a"), ORD(row) - ORD("1")]
         = mark, 120);
   END setMark;
```

board is nt. geëxporteerd
=> contract niet verifieerbaar

Vuistregel

Contracten bepalen de volgorde van oproepen.

Waarom ?

- Belangrijk, maar niet te lezen in een "platte" klasse-interface

Hoe ?

- post-conditie van de voorgaande => pre-conditie van de volgende

```
PROCEDURE (aTicTacToe: TicTacToe) setMark* (col, row, mark: CHAR);
   (* Mark the given position on the board. *)
   BEGIN
      ASSERT( ("a"<=col) & (col<="c") & ("1"<=row) & (row<="3"), 100);
      ASSERT(("X" =mark) OR ("O" = mark) OR (" " = mark), 100);
      aTicTacToe.board [ORD(col) - ORD("a"), ORD(row) - ORD("1")] := mark;
      ASSERT(aTicTacToe.getMark(col, row) = mark, 120);
   END setMark;
```

Postconditie van setMark
gebruikt getMark
=> setMark oproepen voor getMark

Vuistregel

Gebruik pre-conditie om de initializatie van objecten te controleren

Waarom ?

- initialisatie vergeten: veel voorkomende & moeilijk te vinden fout

Hoe ?

- pre-conditie met "properlyInitialized"
  properlyInitialized ? Controleer een self-pointer

```
TYPE
   TicTacToe* = POINTER TO TicTacToeData;
   TicTacToeData* = RECORD
      initCheck: TicTacToe;
      …
   END;

PROCEDURE (t: TicTacToe) properlyInitialized* (): BOOLEAN;
   BEGIN
      RETURN t.initCheck = t;
   END properlyInitialized;
```

Object heeft een extra pointer …

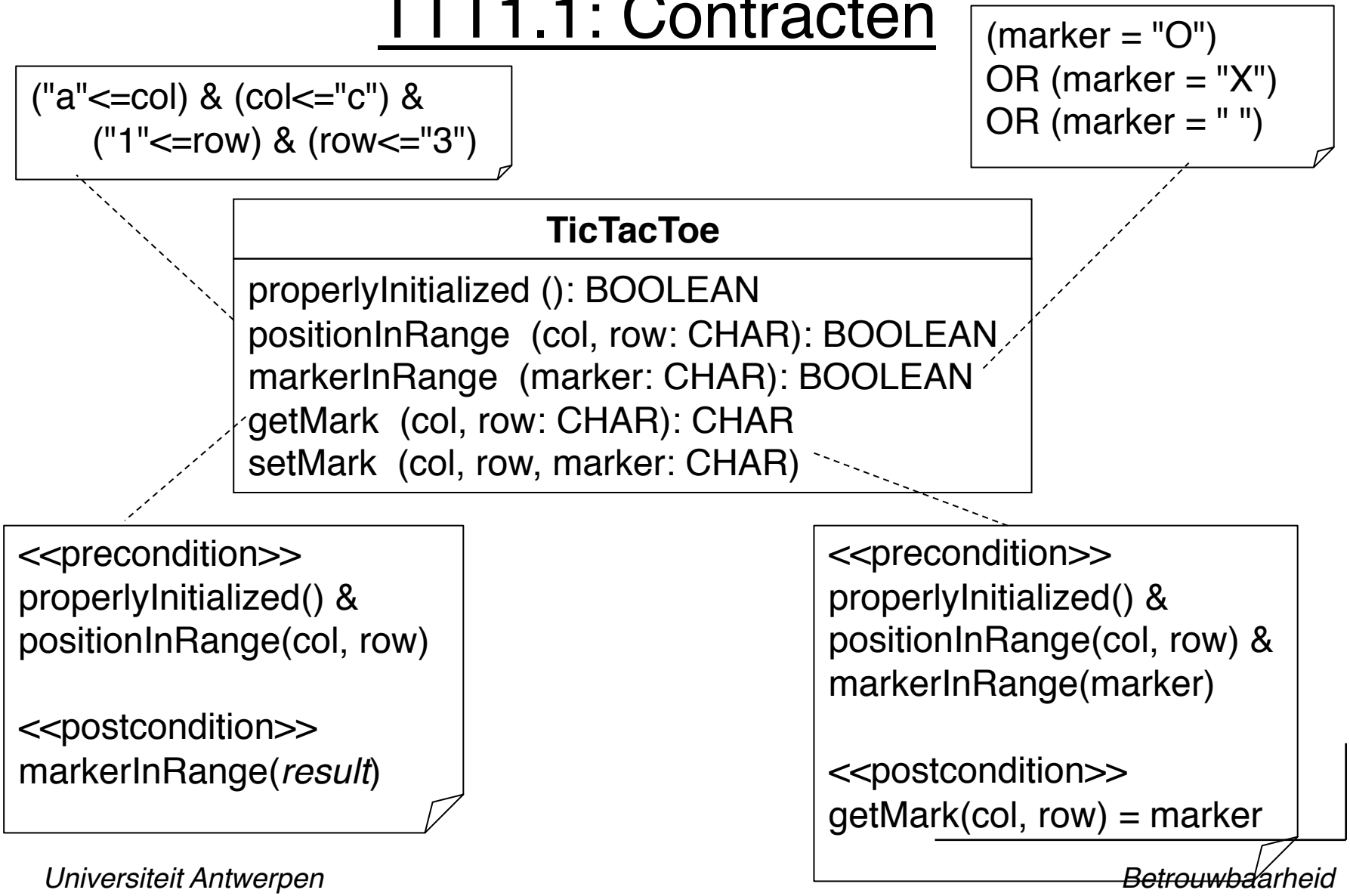
Wijst normaal naar zichzelf

```
PROCEDURE (aTicTacToe: TicTacToe) init*;
   …
      aTicTacToe.initCheck := aTicTacToe;
      ASSERT(aTicTacToe.properlyInitialized(), 120);
   END init;

PROCEDURE (aTicTacToe: TicTacToe) getMark* (col, row: CHAR):
   CHAR;
   (** Answer the mark on the given position on the board. *)
   VAR result: CHAR;
   BEGIN
      ASSERT(aTicTacToe.properlyInitialized(), 100);
      ASSERT(…)
```

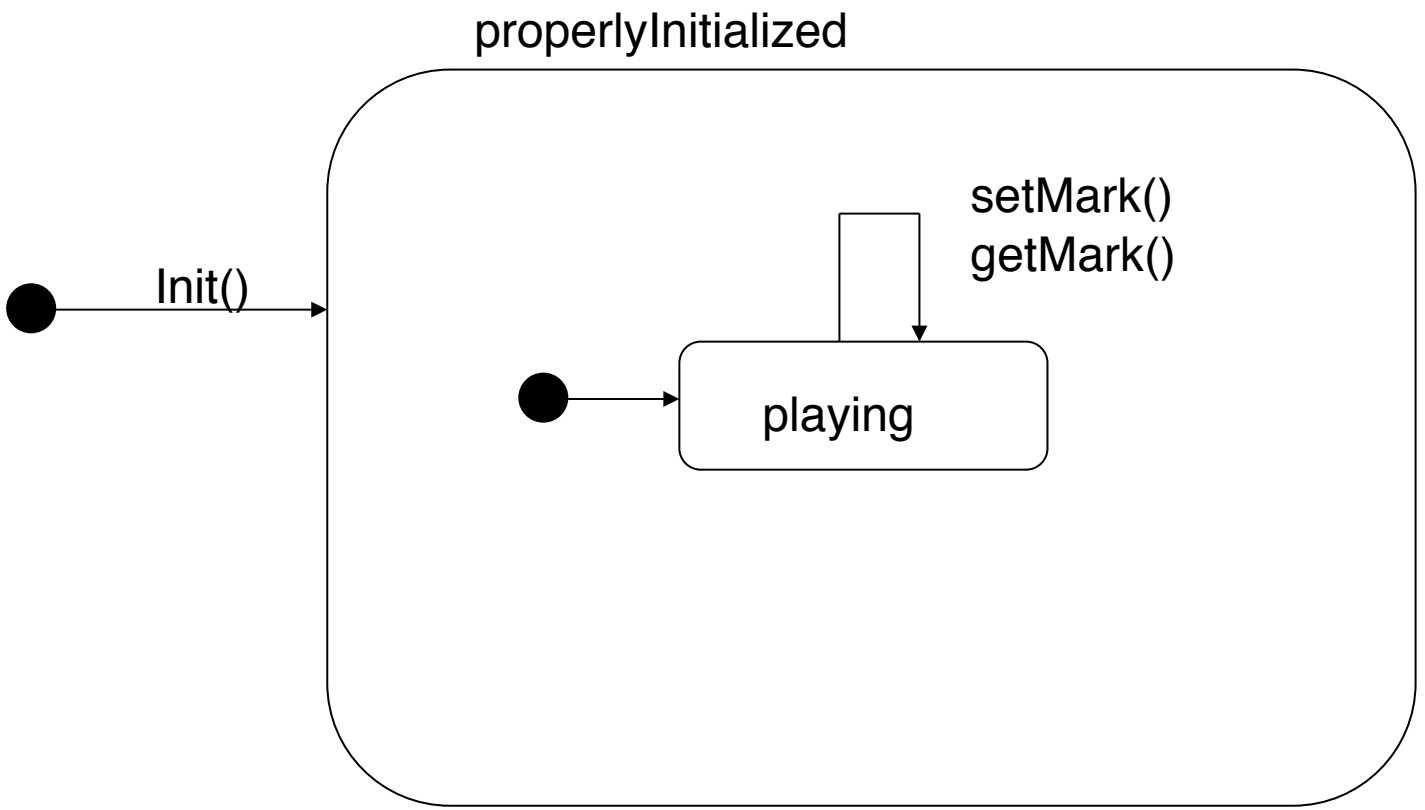
initializatie garandeert "properlyInitialized"

"properlyInitialized"
=> preconditie andere procedures

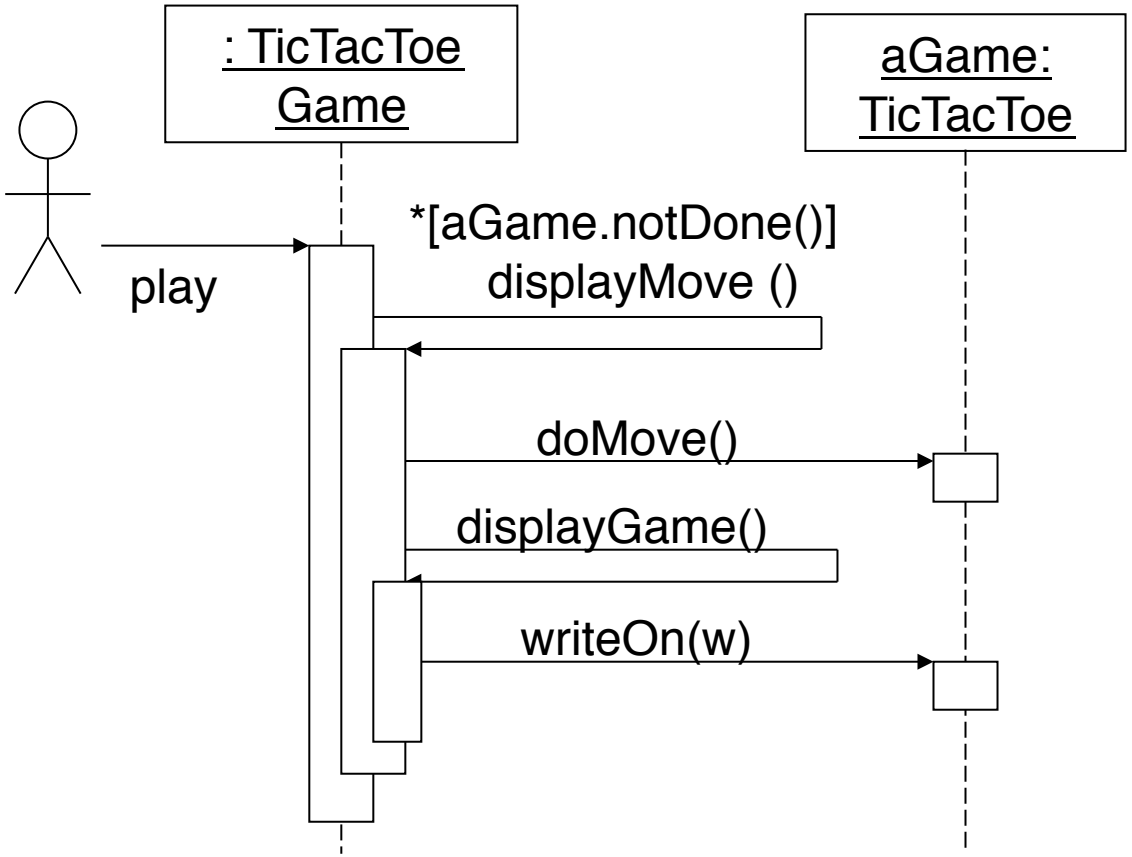
TTT1.1: Contracten



Interpretatie Interface



TicTacToeGame (1.1)



Vuistregels (Samenvatting)

- ONTWERP
  - Keep It Stupidly Simple (the KISS principle)
- TESTEN
  - Minstens één test per "normaal" scenario
  - Als de functionaliteit groeit, dan groeit de test mee
  - Een test produceert zo weinig mogelijk uitvoer
- CONTRACTEN
  - Controleer de waardes van de argumenten die je binnenkrijgt
  - Controleer de waarde van het resultaat dat je teruggeeft
  - Contracten gebruiken alleen publieke (geëxporteerde) declaraties
  - Contracten bepalen de volgorde van oproepen.
  - Gebruik pre-conditie om de initializatie van objecten te controleren