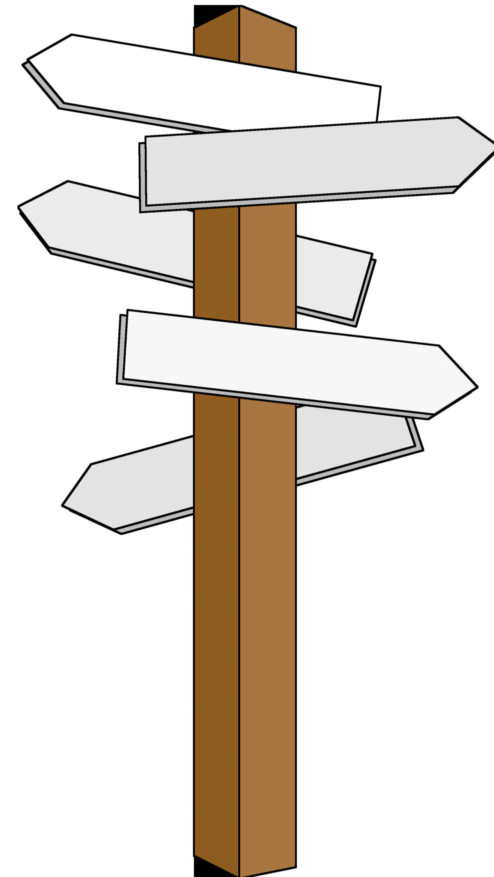
Hoe snel loopt iemand de 100 meter ?

4. Planning

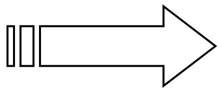
- Tijdsschatting
  - Analogie & Decompostie
  - Empirische schatting
    - Plan 2.0 & Plan 2.1
- Conclusie
- TicTacToe
  - Code Hergebruik
    => TestCase
  - HTML Uitvoer
    => polymorfisme
  - “Undo”
    => Lijsten & polymorfisme



"Ontwikkel" vereisten

Vereisten

- Betrouwbaarheid
- Aanpasbaarheid
- Planning



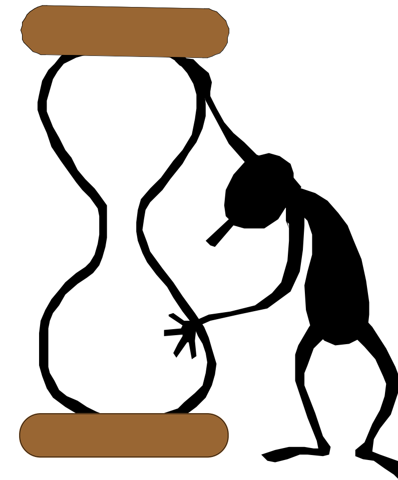
Technieken

- Testen + Contracten
- Objectgericht ontwerp
- Tijdsschatting

Schatting door analogie

Analogie

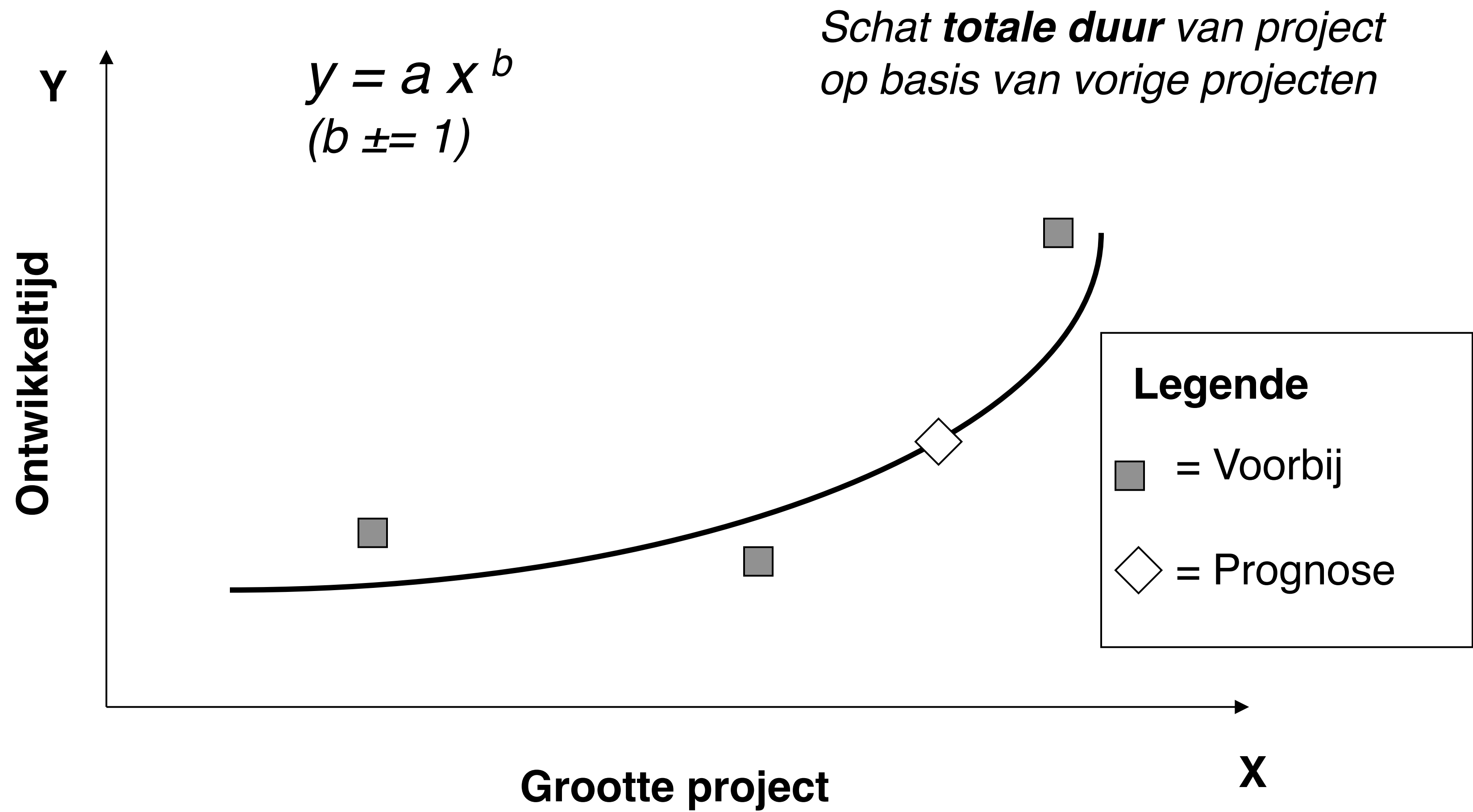
- Schatting = tijd gelijkaardig project
- Wanneer gelijkaardig ?
  - Zelfde probleemdomein
  - Zelfde mensen
  - Zelfde technologie



Empirisch

gespendeerde tijd voorbije projecten dient als basis voor schatting volgende

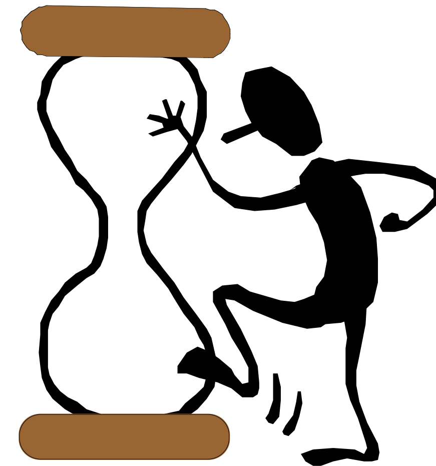
Empirische schatting (project)



Schatting door decompositie

Decompositie

- Schatting = tijd componenten + integratiekost
- Tijd componenten ?
  - cfr. opgeleverde componenten
- Integratiekost ?
  - constant (mits testen en OO)

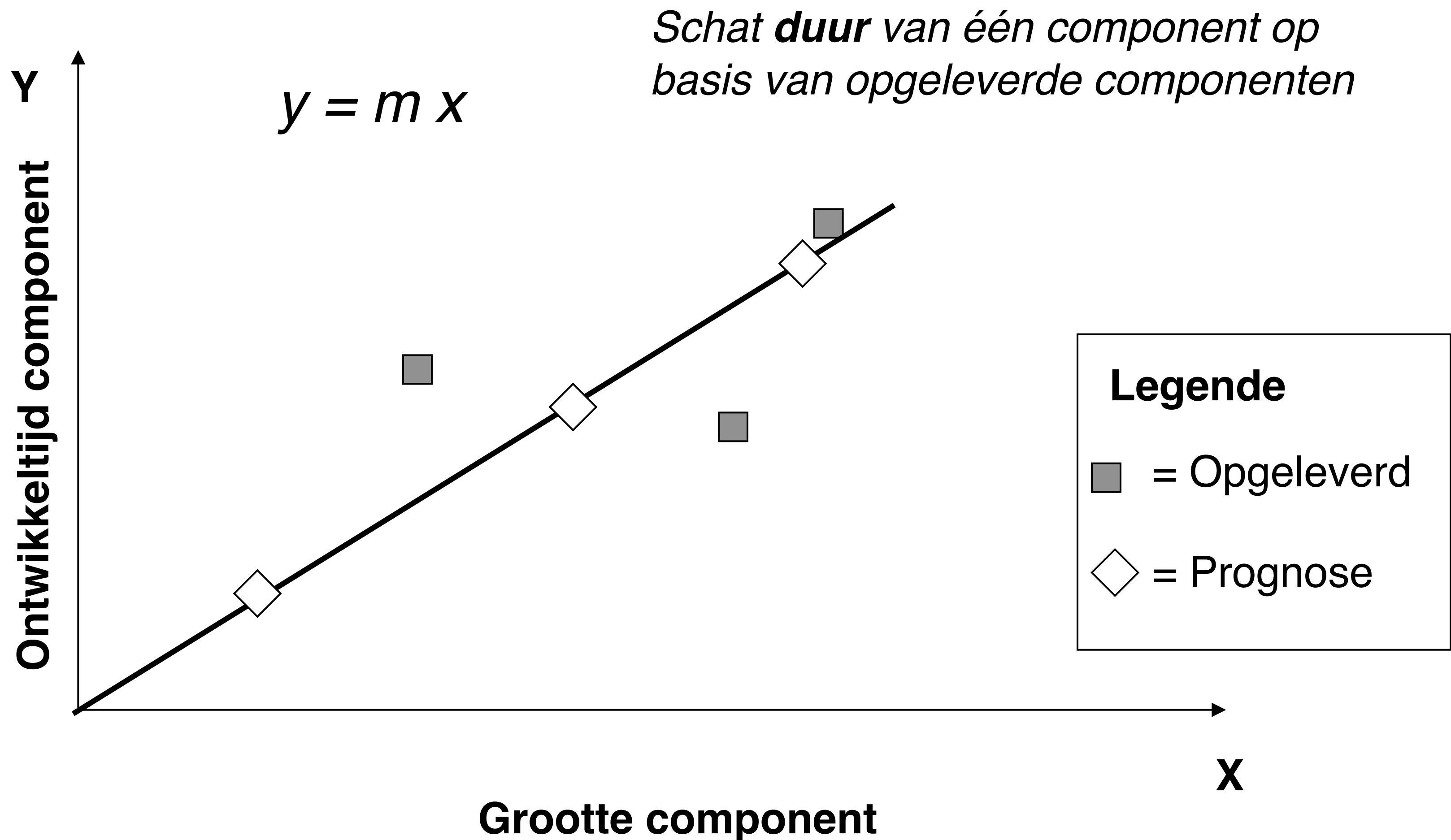


Empirisch

gespendeerde tijd eerste componenten dient als basis voor schatting volgende

Empirische schatting (component)

*Schat **duur** van één component op basis van opgeleverde componenten*



Grootte en Tijd

- x = Grootte Component ?
  # stappen + #uitzonderingen in use case

- y = Ontwikkellingstijd ?
  Zie tijdsbladen

vergelijking benaderende rechte

$$y = mx$$

Na oplevering n componenten: (x_n, y_n) => $m = \sum y_n / \sum x_n$

Schatting y_{n+1} voor grootte x_{n+1} => $y_{n+1} = m.x_{n+1}$

Vuistregel

Empirische schatten is de basis voor een realistische planning.

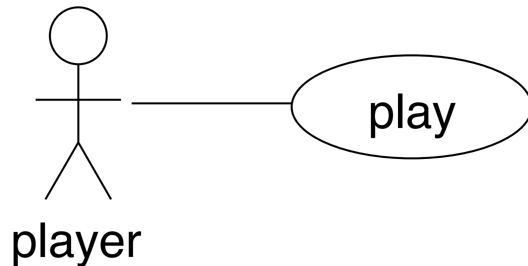
Waarom ?

- Betere controle over aanpassingen aan de planning

Hoe ?

- Hou tijdsbladen nauwkeurig bij
- Maak prognose op basis van gespendeerde werk in het verleden

Use Case Grootte



Use Case 1: play

- …

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

Exceptions

2.1. [Illegal Move] System issues a warning
=> continue from step 2.1

#stappen = 5
#uitzond. = 1
grootte = 6

Voorbeelden

Zie voorbeelden in PlanTmpl20 & PlanTmpl21

Conclusie

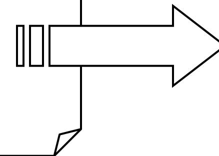
- Betrouwbare prognoses zijn belangrijk
  - Hou tijdsbladen nauwkeurig bij !
- Schatten impliceert fouten
  - Voorzie een redelijke marge
  - Vertrouw niet blindelings op de cijfertjes

Code Hergebruik

TicTacToeTest
setUp() tearDown() init() fail(msg: ARRAY OF CHAR) should(b: BOOLEAN, msg: ARRAY OF CHAR): BOOLEAN shouldNot(b: BOOLEAN, msg: ARRAY OF CHAR): BOOLEAN compareFiles(fileName1, fileName2: ARRAY OF CHAR): BOOLEAN testBasicPlayer(verbose: BOOLEAN): BOOLEAN testLegalMoves(verbose: BOOLEAN): BOOLEAN testRealGame(verbose: BOOLEAN): BOOLEAN testOutputGame(verbose: BOOLEAN): BOOLEAN

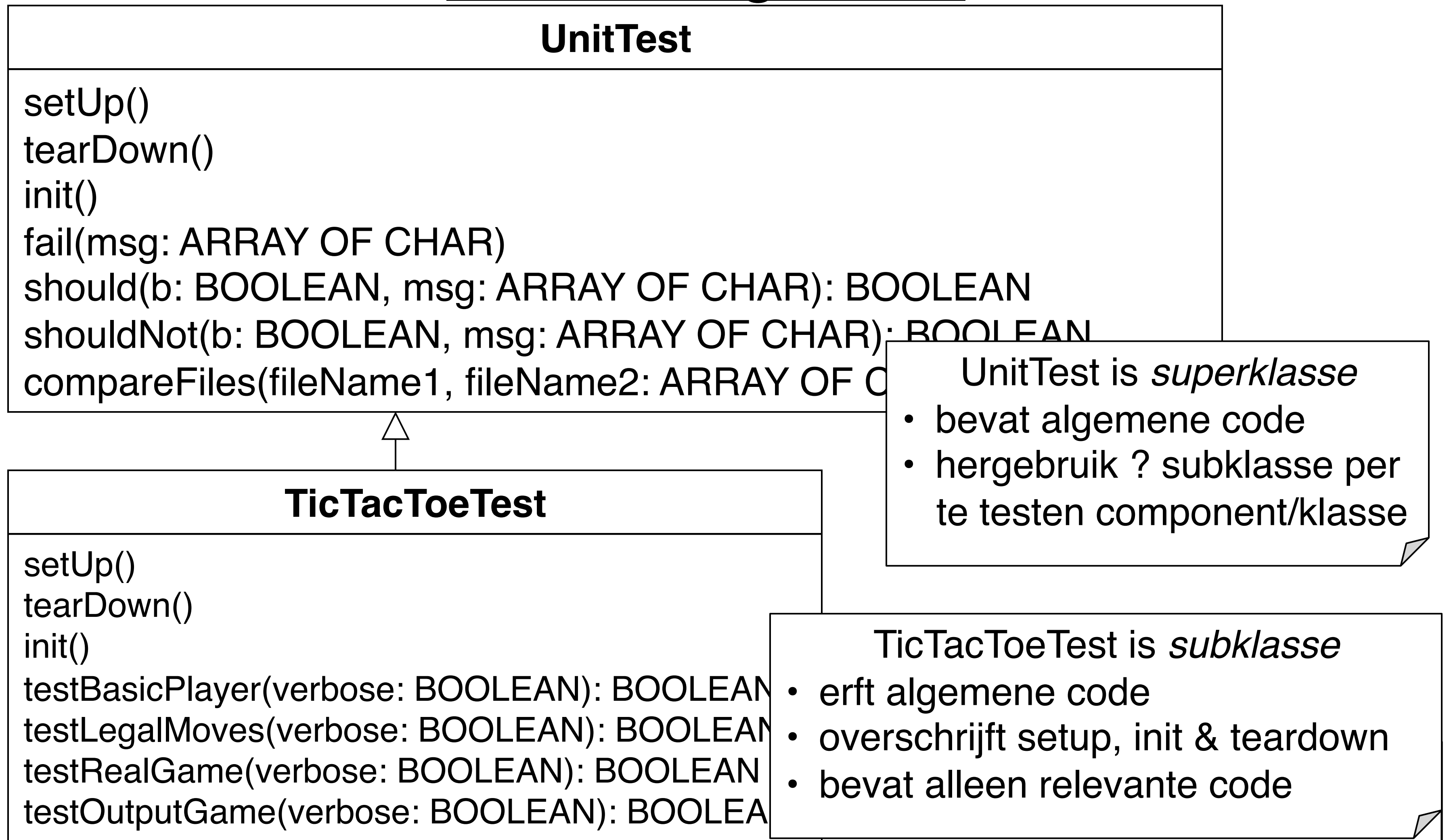
TicTacToeTest is *groot*

- code leesbaarheid
- hergebruik ? copy + delete + insert



Aanpasbaarheid :(

Code Hergebruik



Vuistregel

“Hollywood Principle”

wij roepen jouw op als we je nodig hebben

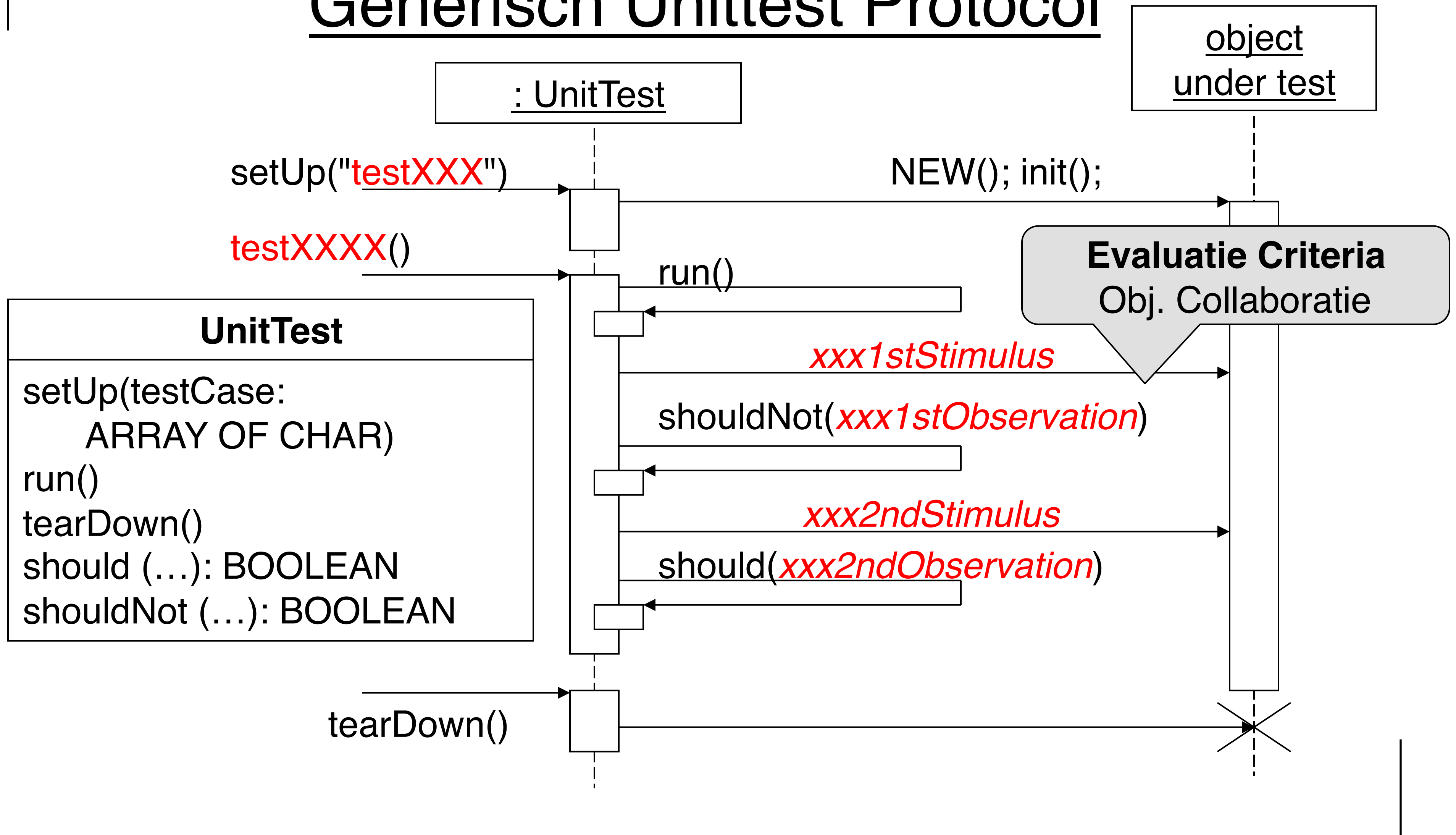
Waarom ?

- Uitbreiding van bibliotheken via subklasses

Hoe ?

- Superklasse in bibliotheek legt protocol van oproepen vast
- Subklassen kunnen gedrag uitbreiden

Generisch Unittest Protocol



Vuistregel

Klassen die vaak herbruikt worden hebben preciese contracten

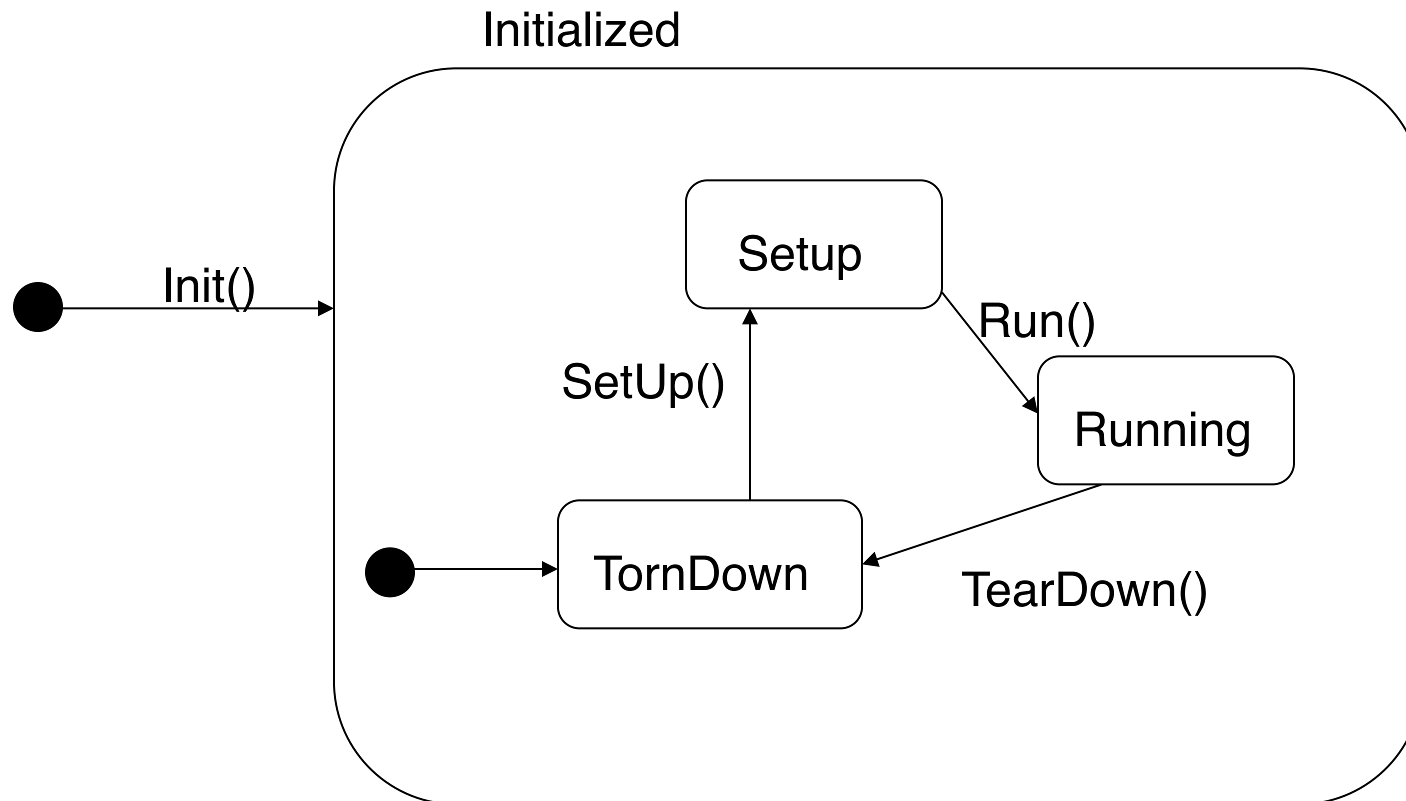
Waarom ?

- Betere betrouwbaarheid door precieze beschrijving interface

Hoe ?

- Leg de “normale” volgorde van oproepen vast
- Specifieer volgorde via de respectievelijke pre- en postcondities

Oproepvolgorde voor “UnitTest”



Contracten voor “UnitTest”

```
PROCEDURE (aTest : UnitTest) Init;
    (* postcondition (120): aTest^.IsInitialized() *)    (* postcondition (120): ~ aTest^.IsSetup() *)
    (* postcondition (120): ~ aTest^.IsRunning() *)      (* postcondition (120): aTest^.IsTornDown() *)

PROCEDURE (aTest : UnitTest) SetUp (testCase: ARRAY OF CHAR);
    (* precondition (100): aTest^.IsInitialized() *)     (* precondition (100): aTest^.IsTornDown() *)
    (* precondition (100): LEN(testCase) < MaxTestCaseLength *)
    (* postcondition (120): aTest^.IsSetup() *)          (* postcondition (120): ~ aTest^.IsTornDown() *)

PROCEDURE (aTest : UnitTest) Run;
    (* precondition (100): aTest^.IsInitialized() *)     (* precondition (100): aTest^.IsSetup() *)
    (* postcondition (120): ~ aTest^.IsSetup() *)        (* postcondition (120): aTest^.IsRunning() *)

PROCEDURE (aTest : UnitTest) TearDown;
    (* precondition (100): aTest^.IsInitialized() *)     (* precondition (100): aTest^.IsRunning() *)
    (* postcondition (120): ~ aTest^.IsRunning() *)      (* postcondition (120): aTest^.IsTornDown() *)
```

Vuistregel

Schrijf testcode als subklasse(n) van "UnitTest"

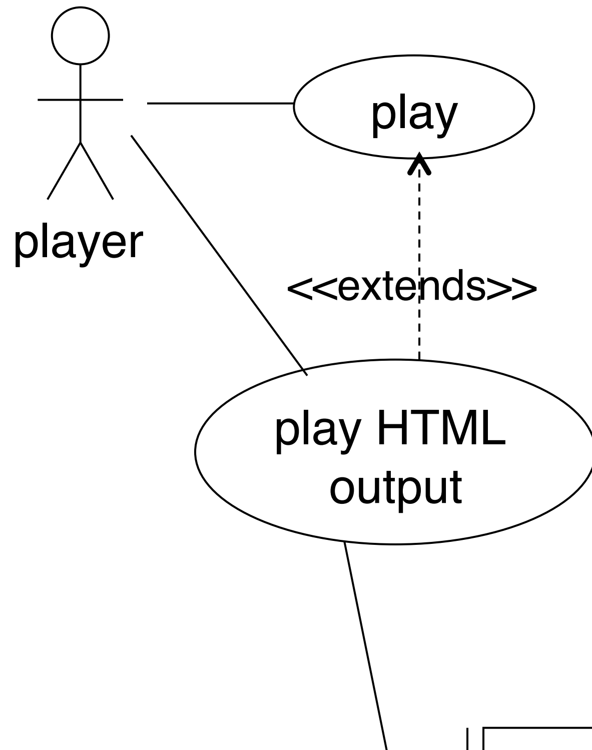
Waarom ?

- Betere onderhoudbaarheid van de testcode

Hoe ?

- Maak een subklasse van "UnitTest" per te testen component
- Overschrijf Init(), SetUp(), TearDown(), Run ()

HTML Uitvoer (TTT16b)



Use Case 3: play HTML output

- Extension of use case 1

Steps

use case 1 + additional steps

afer 2.2

2.3 write game on HTML (table)

during 3

3 write winner on HTML text

(Erg gelijkaardig aan use case 2)
=> inheritance & polymorfisme ?

```
PROCEDURE (aTicTacToe: TicTacToe) writeHTMLOn* (VAR w: Texts.Writer);
…
BEGIN
  Texts.WriteString(w, "<P>");
  Texts.WriteString(w, "TicTacToe game after move ");
  Texts.WriteInt(w, aTicTacToe.nrOfMoves, 0);
  Texts.WriteString(w, "</P>");Texts.WriteLn(w);
  Texts.WriteString(w, "<TABLE BORDER>");Texts.WriteLn(w);
  FOR i := 0 TO 2 DO
    Texts.WriteString(w, "  <TR>");Texts.WriteLn(w);
    FOR j := 0 TO 2 DO
      Texts.WriteString(w, "    <TD>");
      …
      END;
…
```

"Ongeveer" gelijk aan `writeOn` => duplicatie

```
PROCEDURE (aTest: TicTacToeTest) testOutputGame* (…
      writeHTML: BOOLEAN): BOOLEAN;
…
   IF writeHTML THEN
      Texts.WriteString(w, "<HTML>");Texts.WriteLn(w);
      Texts.WriteString(w, "<HEAD>");Texts.WriteLn(w);
      …
   END;
   WHILE aTest.aGame.notDone() DO
      aTest.aGame.doMove();
      IF writeHTML THEN aTest.aGame.writeHTMLOn(w);
         ELSE aTest.aGame.writeOn(w); END;
   END;
   IF writeHTML THEN
      Texts.WriteString(w, "</BODY></HTML>");Texts.WriteLn(w);
   END;
…
```

=> complexe conditionele logica

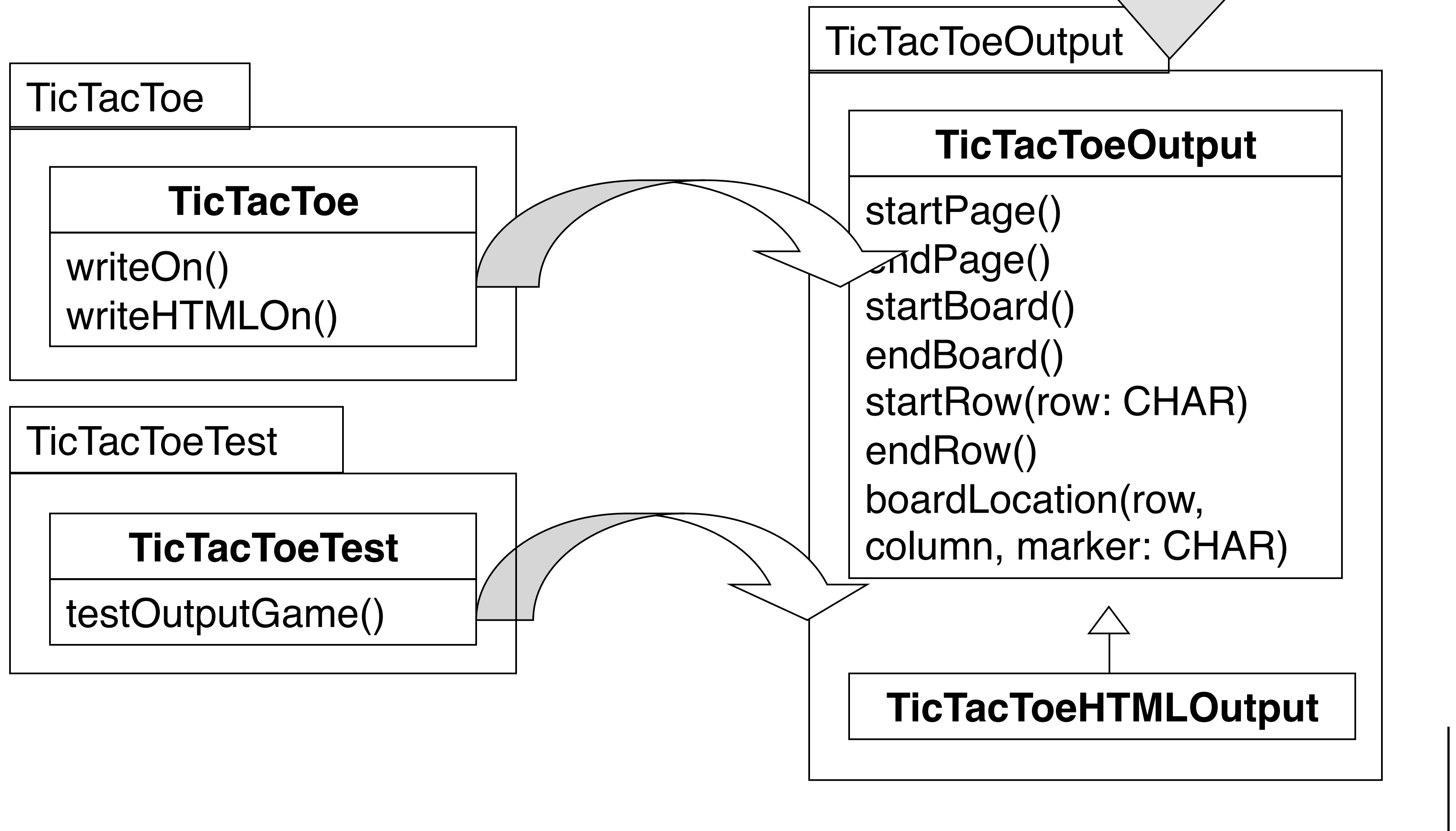
Vuistregel

Vermijd code duplicatie
& complexe logica

Refactor:

- *code duplicatie*: gelijkaardige code in de superklasse; verschillen in subklassen
- *complexe logica*: normaal geval in de superklasse; speciale gevallen in de subklassen

Splits Klassen (TTT16)



```
PROCEDURE (aTest: TicTacToeTest) testOutputGame* (…
      writeHTML: BOOLEAN): BOOLEAN;
…
   Texts.OpenWriter(w);
   output.init ();
   output.startPage(w);
   WHILE aTest.aGame.notDone() DO
      aTest.aGame.doMove();
      aTest.aGame.writeOn(w, output)
   END;
   output.endPage(w);
```

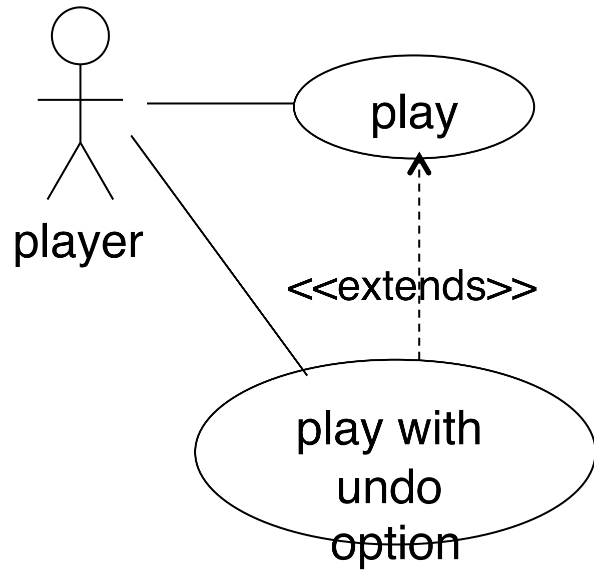
complexe condities => polymorfisme

```
PROCEDURE (aTicTacToe: TicTacToe) writeOn* (VAR w: Texts.Writer;
   output: TicTacToeOutput.TicTacToeOutput);
BEGIN
   output.startSentences(w);
   Texts.WriteString(w, "TicTacToe game after move ");
   Texts.WriteInt(w, aTicTacToe.nrOfMoves, 0);
   output.endSentences(w);
   output.startBoard(w);
   FOR i := 0 TO 2 DO
      output.startRow(w, CHR(ORD("1") + i));
      FOR j := 0 TO 2 DO
         output.boardLocation(...);
      END;
      output.endRow(w);
   END;
output.endBoard(w);
```

extra parameter controleert verschil

verschillen in gedupliceerde code => polymorfisme

“Undo”



Use Case 4: play with undo option

- Extension of use case 1

Steps

use case 1 + additional steps

afer 2.1

2.1.1 current player “undo” move

2.1.2 goto 2 (restart while loop)

"Undo" Algoritme ?

- Hou een lijst bij met verloop spelstatus

```
TicTacToeData* = RECORD (OOLists.NodeDesc)
        nrOfMoves: INTEGER;
        board: ARRAY 3, 3 OF CHAR;
        players: ARRAY 2 OF Player;
        lastCol, lastRow, lastMark : CHAR;
        theWinner: Player;
END;
```

Herbruik bestaande code

Maar …
"Klassen die vaak herbruikt worden hebben preciese contracten"

Contracten voor “OOLists” ?

```
PROCEDURE (l : List) Init ();
PROCEDURE (l : List) LocateFirst ();
PROCEDURE (l : List) LocateLast ();
PROCEDURE (l : List) LocateNode (n: Node);
PROCEDURE (l : List) LocatePrev ();
PROCEDURE (l : List) LocateNext ();
PROCEDURE (l : List) InsertBefore (new: Node);
PROCEDURE (l : List) InsertAfter (new: Node);
PROCEDURE (l : List) Delete ();
PROCEDURE (l : List) GetNode (): Node;
PROCEDURE (l : List) Enumerate (P: NodeProc);
PROCEDURE (n : Node) NodeInfo ();
```

Contracten ? Moeilijk want interface zonder predicaten

"Lijst" met predicaten

```
PROCEDURE (l : List) Init ();
    (* postcondition (120): l^.IsInitialized() *)
PROCEDURE (l : List) Includes (n: Node): BOOLEAN;
    (* precondition (100): l^.IsInitialized() *)
PROCEDURE (l : List) Insert (new: Node);
    (* precondition (100): l^.IsInitialized() *)
    (* postcondition (120): l^.Includes(new) *)
PROCEDURE (l : List) Delete (old: Node);
    (* precondition (100): l^.IsInitialized() *)
    (* postcondition (120): ~ l^.Includes(new) *)
```

Evaluatie Criteria
Reuse (lijsten)

Contracten ? Makkelijker want interface met veel predicaten

Vuistregel

Prefereer een interface met predicaten

Waarom ?

- Betere betrouwbaarheid door eenvoudige contracten

Hoe ?

- Specifieer predicaten voor hoofdfunctionaliteit component
- Roep predicaten op in pre- en post-condities

Vuistregels

Testen

- Schrijf testcode als subklasse(n) van “UnitTest”

Ontwerpen

- Vermijd code duplicatie & complexe logica
- “Hollywood Principle” – wij roepen jouw op als we je nodig hebben
- Klassen die vaak herbruikt worden hebben preciese contracten
- Prefereer een interface met predicaten

Plannen

- Empirische schatten is de basis voor een realistische planning