Project Software Engineering

Cursus
1ste Bachelor Informatica
Academiejaar 2019-2020

Universiteit Antwerpen

1.Praktisch

- Doel
- Contactpersonen
- Inhoud
- Opbouw
- Mijlpalen
- Tijdsbesteding
- Eindbeoordeling
- Spelregels (⇒ Fraude)
- Cursusmateriaal

http://ansymore.uantwerpen.be/courses/SE1BAC

Doel

Een eerste ervaring verwerven ...
- doen, niet kijken hoe het gedaan wordt

in het zelf realiseren ...
- zelf doen ... weliswaar in groepjes van 2

van een software oplossing ...
- informatica doen: jullie interesse

voor een niet-triviaal probleem ...
- een onderdeel van een groter geheel

Contactpersonen

Docent
- Serge Demeyer
  (Middelheim - G106b)
- serge.demeyer@uantwerpen.be
  GEEN bijlagen in e-mail

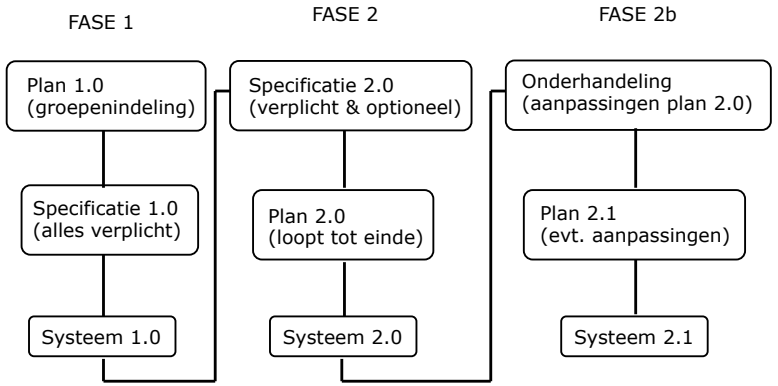
Assistenten
(Middelheim – G 3de verdieping)
- Vragen tijdens uren practicum
  ⇒ practica zalen gereserveerd !

Inhoud

Project
- Zelfwerkzaamheid
- Duo: in groepjes van 2
  + Groepen zijn vrij te kiezen ...
    ... binnen categorieën bepaald door "Programmeren I"
  + Elk krijgt dezelfde punten

... op basis van theorie
- 3-tal lessen
- Minimum toe te passen technieken
  + testen + contracten
  + objectgericht ontwerpen
  + plannen

Opbouw



Semester Indeling (ideale omstandigheden)

Week	Theorie	Opdracht	Mijlpaal	
1	Intro / Betrouwbaarheid			
2	Objectgericht ontwerpen	Specificatie 1.0	Plan 1.0	
3	Planning			
4				
5				
6	* Projectverdediging *	Specificatie 2.0	Systeem 1.0	
7				Plan 2.0
		-- paas vakantie		
8				
9				
10	* Projectverdediging *		Systeem 2.0	
11			Plan 2.1	
12		• *Calculus*		
13		• *Computer Graphics* • *Talen & Automaten*		
…	*Projectverdediging (finaal)*		Systeem 2.1	

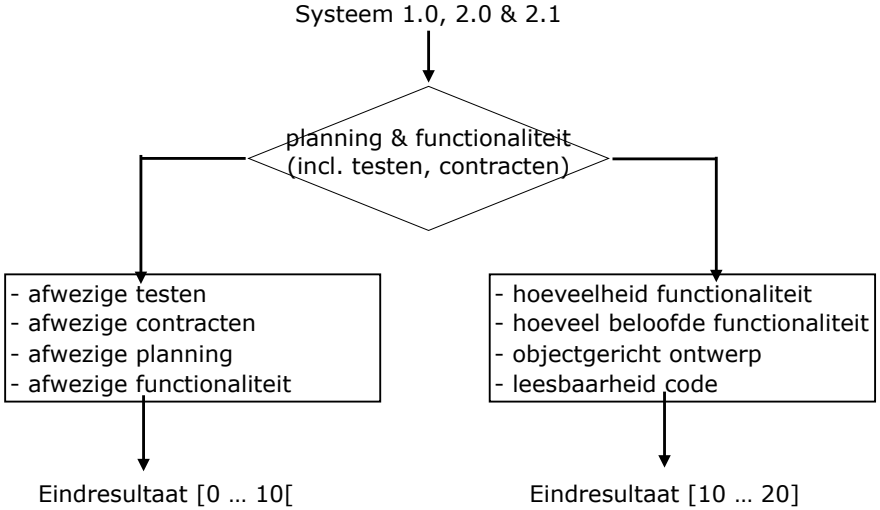
Tijdsbesteding

15u theorie	x 2	⇒ 30u extra studietijd	30
60u oefeningen	x 1.5	⇒ 90u extra studietijd	150
TOTAAL		⇒ "werktijd"	180
TOTAAL per week		⇒ 180 u ÷ (13 – 1) weken	⇒ 15 u per week (*)

Log de tijd besteed aan je project op speciale tijdsbladen.
⇒ zelfcontrole

(*) Dat is wat wij ongeveer verwachten; niet noodzakelijk wat jullie effectief besteden

Eindbeoordeling



Spelregels

- werkt in computerlabo
  + "op de PC thuis werkt het" :(
- stipt opleveren
  + te laat = niet opgeleverd
- accurate tijdsbladen
  + plannen / onderhandelen :)
- maak reservekopieën
  + "gisteren werkte het nog" :(
- werk nauw samen
  + per twee voor één scherm :)
    - 1 programmeur / 1 denker
    - verwissel de rollen

- meld problemen tijdig
  + partner werkt niet mee
  + partner haakt af
- verboden werk te kopiëren
  + betrapt ⇒ fraude
  + bibliotheken (o.a. graphics) zijn toegelaten

Fraude ?

Examenvorm = Project
  + programma code kopiëren / bekijken ≈ Afkijken
  + programma code doorgeven ≈ laten Afkijken

Controle
  + na elke evaluatie vergelijken we alle projecten
    - we houden de historiek bij

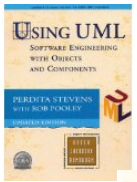
Sanctie
  + uitgesloten voor (een gedeelte van) het vak
  + uitgesloten voor de zittijd
  + uitgesloten voor het academiejaar

Cursusmateriaal

Kopies van de transparanten + alle andere informatie
- http://ansymore.uantwerpen.be/courses/SE1BAC

Zie ook Blackboard
- volg de aankondigingen + lees je UA e-mail

Achtergrondinformatie
- [Stev99a] Using UML: Software Engineering with Objects and Components, P. Stevens, R. Pooley, Addison-Wesley, 1999.
  Engelstalige versie
- [Stev00a] Toepassing van UML: Software-engineering met objecten en componenten, P. Stevens, R. Pooley, Addison-Wesley, 2000.
  Nederlandstalige versie



Ontdek de 7 verschillen

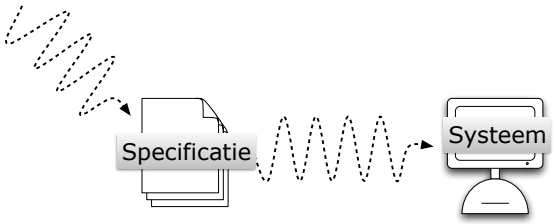
Een oplossing met een foutje ...

2. Betrouwbaarheid

Software Engineering
- Wat?
- Bouw vs. Ontwikkel
- Vereisten
  + Betrouwbaarheid
  + (Aanpasbaarheid & Planning)

Betrouwbaarheid: TicTacToe
- specificatie
- TicTacToe10: ontwerp, implementatie & test
- TicTacToe11: ontwerp, implementatie
  met slechte test (manuele controle)
- TicTacToe12: contracten & eerste demonstratie

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
- apps

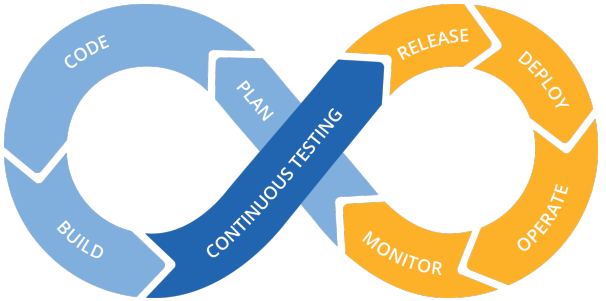
Programmeerploeg
- losse functie-opdeling
- Analist + programmeur, ...

Systeem in functie van organisatie
- Systeem past zich aan

Vereisten

Vereisten	Technieken
• Betrouwbaarheid	• Testen + Contracten
• Aanpasbaarheid	• Objectgericht ontwerp
• Planning	• Tijdsschatting
Werktuigen	
• "Build"	• Make, CMake
• Editor, Debugger	• CLion
• Documentatie	• Doxygen
• Version control	• Git / Github / Gitlab

State-of-the-art Software Construction



Continuous Integration

Tesla
"over-the-air" updates
± once every month

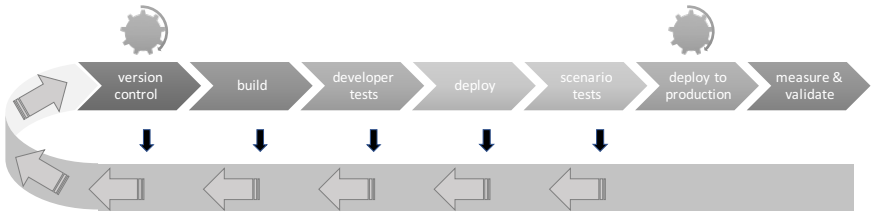
Continuous Delivery
Continuous Deployment

Amazon deploys to production
± every 11,6 seconds

DevOps

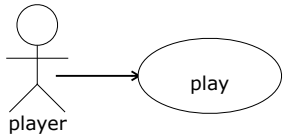
September 2015
Amazon Web Services suffered major disruption.
NetFlix recovers quickly!
(Chaos Monkey)

Continuous Integration / Deployment



Vereisten	Technieken
• Betrouwbaarheid • Aanpasbaarheid • Planning	• Testen + Contracten • Objectgericht ontwerp • Tijdsschatting
Werktuigen	
• "Build" • Editor, Debugger • Documentatie • Version control	• Make, CMake • CLion • Doxygen • Git / Github / Gitlab

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. Announce winner

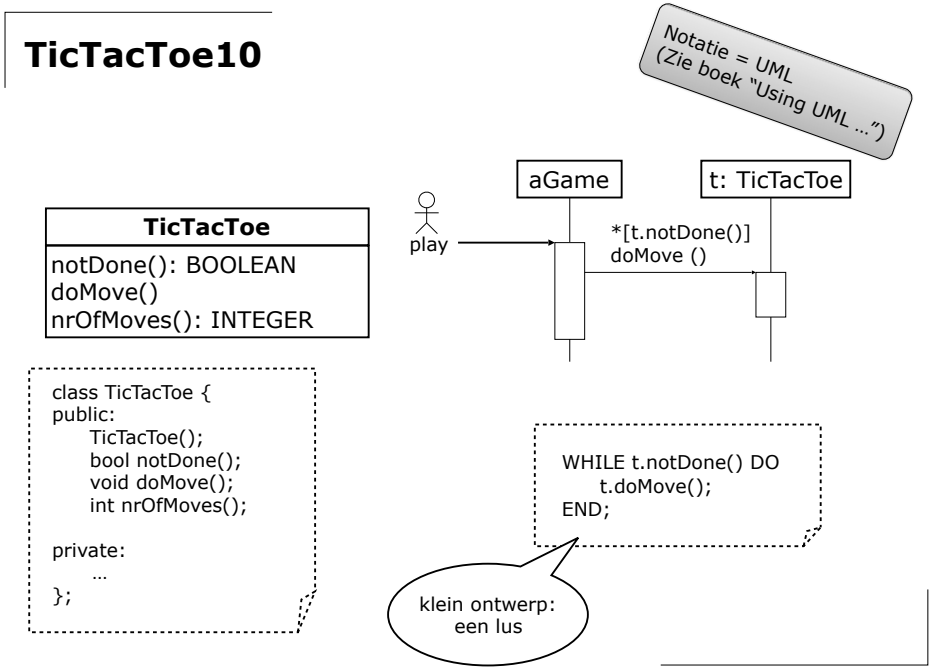
Vuistregel

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

TicTacToe10



TicTacToe10 in C++

```
class TicTacToe {
public:
  TicTacToe();
  bool notDone();
  void doMove();
  int nrOfMoves();

private:
  int _nrOfMoves;
};
```

simpele implementatie
= een teller

```
TicTacToe::TicTacToe() {
  _nrOfMoves = 0;
}

bool TicTacToe::notDone() {
  return _nrOfMoves < 9;
}

void TicTacToe::doMove() {
  _nrOfMoves++;
}

int TicTacToe::nrOfMoves() {
  return _nrOfMoves;
}
```

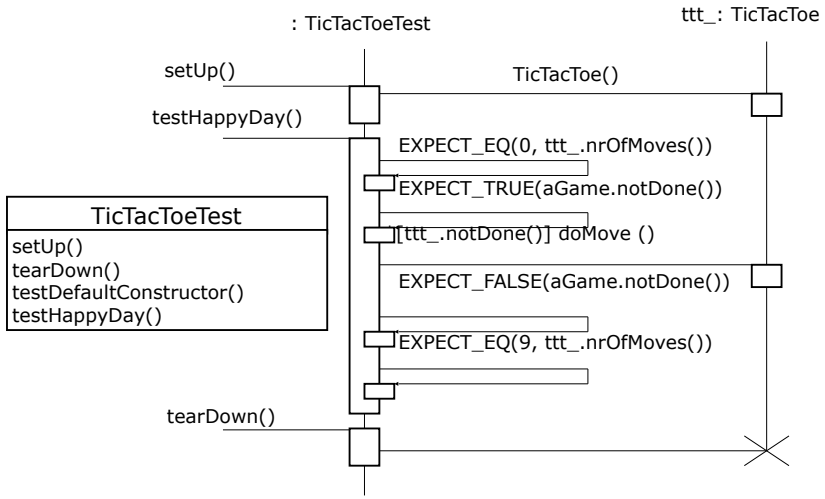

Vuistregel

Minstens één test per "normaal" scenario

Waarom ?
- Minimum veiligheidsnorm

Hoe ?
- Controleer de tussentijdse toestand via ”EXPECT_…”

TicTacToe10: Test (Happy Day)



Test the “happy day” scenario

```
TEST_F(TicTactToeTest, HappyDay) {
   EXPECT_EQ(0, ttt_.nrOfMoves());
   EXPECT_TRUE(ttt_.notDone());

   while (ttt_.notDone()) {
      ttt_.doMove();
   };

   EXPECT_FALSE(ttt_.notDone());
   EXPECT_EQ(9, ttt_.nrOfMoves());
}
```

Controleer tussentijdse toestand

Controleer tussentijdse toestand

Infrastructuur (GoogleTest)

```
class TicTactToeTest: public ::testing::Test {
protected:
  virtual void SetUp() {
  }
  virtual void TearDown() {
  }
  TicTacToe ttt_;
};

... Tests here by invoking macro TEST_F ...

// the main function ... there can only be one :)
int main(int argc, char **argv) {
  ::testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```


Testresultaat

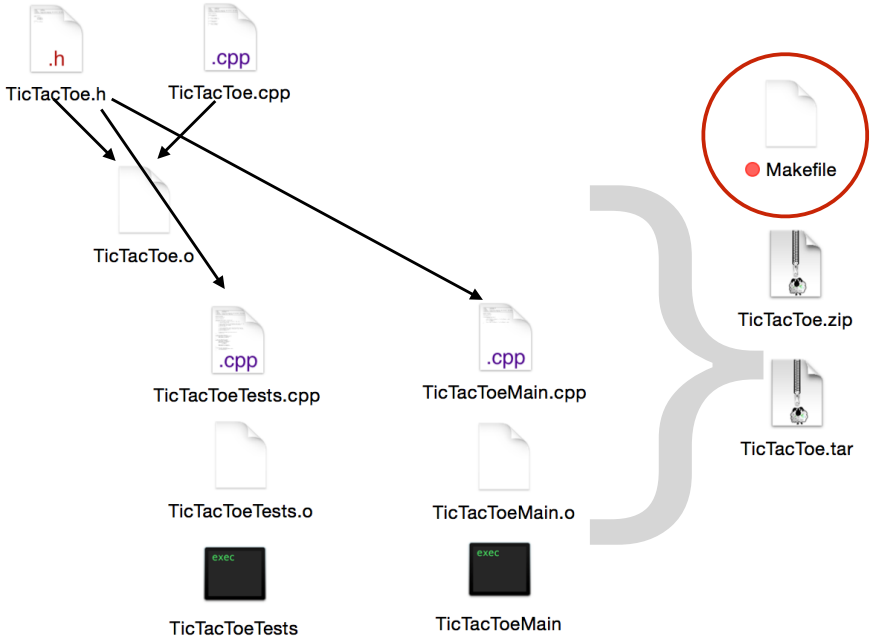
```
[==========] Running 2 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 2 tests from TicTactToeTest
[ RUN      ] TicTactToeTest.DefaultConstructor
[       OK ] TicTactToeTest.DefaultConstructor (0 ms)
[ RUN      ] TicTactToeTest.HappyDay
[       OK ] TicTactToeTest.HappyDay (0 ms)
[----------] 2 tests from TicTactToeTest (0 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test case ran. (1 ms total)
[  PASSED  ] 2 tests.
```

Spelregel: werkt in computerlabo
Zorg *zelf* voor de nodige bibliotheken
Test het vooraf in de computerklas
zorg voor makefiles

Bewust Falende test

```
[  FAILED  ] TicTactToeTest.HappyDay (1 ms)
[----------] 2 tests from TicTactToeTest (1 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test case ran. (1 ms total)
[  PASSED  ] 1 test.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] TicTactToeTest.HappyDay
```

Projectverdediging: Quiz
Ik verander één van de tests
Slaagt de test ? Of faalt de test ?



Build Rules

```
target : depends upon
     build rule(s)
```

een make file is een opeenvolging van "regels"

```
%.o : %.cpp %.h
   g++ -I../gtest/include  -c -o $@ $<
```

een.o file is afhankelijk van .h en.cpp file compileren met de juiste parameters

```
TicTacToeMain : TicTacToe.o TicTacToeMain.o
   g++ -o $@ TicTacToe.o TicTacToeMain.o
```

De main file is afhankelijk van alle .o files linken met de juiste parameters

Build Rules (2)

```
.PHONY: non file targets
     build rule(s)
```

.PHONY is een conventie om build targets te markeren die *geen* bestanden zijn
De rule zal *altijd* afvuren

```
.PHONY : clean
clean :
   rm -f *.o TicTacToeMain TicTacToeTests
```

Vaak gebruikte .PHONY targets
"all" "clean" "install"

```
.PHONY : sourceszip
sourceszip :
   zip -q -r TicTacToe.zip *.h *.cpp Makefile
```

Maak een reservekopie !!!

```
.PHONY : depend
depend :
        g++ -MM -I $(INCL) $(SRCS)
```

Genereer de dependencies

Build Rules (3)

```
varname = variable contents
$(varname)
```

We kunnen ook variabelen definiëren
… en dan later gebruiken

```
CXXFLAGS =   -O2 -g -Wall -fmessage-length=0 -I$(INCL)
OBJS =    TicTacToe.o
SRCS =    TicTacToe.cpp \
       TicTacToeMain.cpp \
       TicTacToeTests.cpp
```

\ om te splitsen over meerdere lijnen

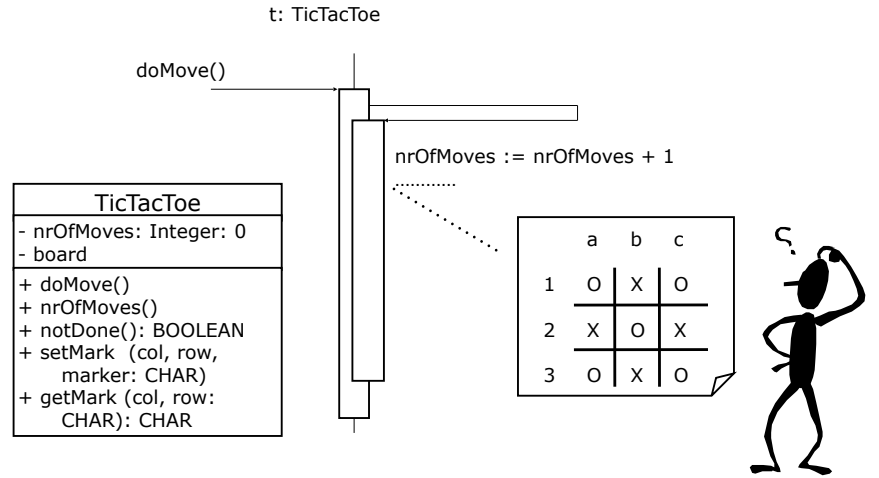
```
TicTacToeMain : $(OBJS) TicTacToeMain.o
   $(CXX) $(CXXFLAGS) -o $@ $(OBJS) TicTacToeMain.o
```

Een build regel met variabelen

```
.PHONY : echo
echo :
        @echo CXXFLAGS = $(CXXFLAGS)
        @echo CXX = $(CXX)
        …
```

echo variabelen (debug makefile)

TicTacToe11 doMove



Incrementeel Ontwerp

```
void TicTacToe::doMove() {
  char col, row, marker;
  col = (char) (_nrOfMoves % 3) + 'a';
  row = (char) (_nrOfMoves / 3) + '0';
  if (_nrOfMoves % 2) marker = 'X'; else marker = 'O';
      // when _nrOfMoves is odd assign 'X'
  this->setMark(col, row, marker);
  _nrOfMoves++;
}
```

Ontwerp & Implementatie
groeien langzaam mee !

Vuistregel

Als de functionaliteit
groeit,
dan groeit de test mee

Waarom ?
- Veiligheidsmaatregel: gecontroleerde groei

Hoe ?
- Pas bestaande tests aan
- ⇒ extra tussentijds toestanden controleren

Test groeit mee

```
TEST_F(TicTactToeTest, HappyDay) {
  ...
  while (ttt_.notDone()) {
    ttt_.doMove();
  };
  char col, row;
  bool markIsX = false; // start with 'O'
  for (col = 'a'; col < 'd'; col++)
    for (row = '0'; row < '3'; row++) {
      if (markIsX)
        EXPECT_EQ('X', ttt_.getMark(col, row));
      else
        EXPECT_EQ('O', ttt_.getMark(col, row));
      markIsX = not markIsX;
    }
...
```

... ook TestDefaultConstructor

Vuistregel

Een test produceert zo
weinig mogelijk uitvoer

”PASSED” ☺

"FAILED" ☹

Waarom ?
- Veel en frequent testen ⇒ onmogelijk om uitvoer te controleren

Hoe dan wel ?
- Tests antwoorden
  - “PASSED” = alle tests zijn goed verlopen (welke ?)
  - “FAILED” = minstens één test heeft gefaald (welke ?)

Een slechte test …

```
TEST_F(TicTactToeTest, ManualBADHappyDayTest) {
   cout << "Start of game: ttt.nrOfMoves() = "
        << ttt_.nrOfMoves() << endl;
   cout << "Start of game: ttt.notDone() = "
        << ttt_.notDone() << endl;
   while (ttt_.notDone()) {
      ttt_.doMove();
   }
   char col, row;
   bool markIsX = false; // start with 'O'
   for (col = 'a'; col < 'd'; col++)
      for (row = '0'; row < '3'; row++) {
         cout << col << " - " << row << ": "
              << ttt_.getMark(col, row) << " =? ";
         if (markIsX) cout << "X" << endl; else cout << "O" << endl;
         markIsX = not markIsX;
      }
   cout << "End of game: ttt.nrOfMoves() = " << ttt_.nrOfMoves() << endl;
   cout << "End of game: ttt.notDone() = " << ttt_.notDone() << endl;
}
```


Manuele Verificatie van Tests

Hoe weet ik of deze uitvoer correct is ?

```
Start of game: ttt.nrOfMoves() = 0
Start of game: ttt.notDone() = 1
a - 0: O =? O
a - 1: X =? X
a - 2: O =? O
b - 0: X =? X
b - 1: O =? O
b - 2: X =? X
c - 0: O =? O
c - 1: X =? X
c - 2: O =? O
End of game: ttt.nrOfMoves() = 9
End of game: ttt.notDone() = 0
```

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

Evaluatiecriteria

Inleiding Software Engineering (20?? - 20??) **Tweede Evaluatie**

Student1: student1 - Student2: student2 - Student3: student3

Commentaar:

Tests

O Nt aanwezig (= geen tests)	O Vrkrd. gebruik (= manuele test)	O Beperkt (= nt. alles getest)	O Voldoende (= geen uitvoer, verkeerde invoer)	O Goed (= normale test)	O Excellent (= testcode goed)

Manuele tests = *onvoldoende*

Nt. alle functionaliteit getest= *onvoldoende*

TicTacToe11: Klasse Interface

TicTacToe
- nrOfMoves: Integer: 0 - board
+ doMove() + nrOfMoves() + 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"

Een klein foutje …

TicTacToe.h

```
//The state of the game is represented as 3x3 array
//of chars marked ' ', 'X', or 'O'.
//We index the state using chess notation, i.e.,
//column is 'a' through 'c' and row is '1' through '3'.*)
void setMark(char col, char row, char marker);
char getMark(char col, char row);
```

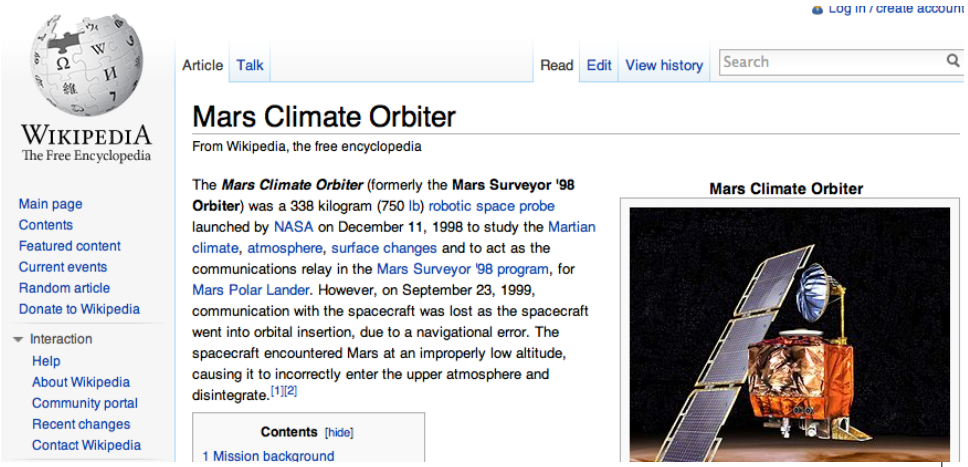
TicTactests.cpp

```
TEST_F(TicTactToeTest, DefaultConstructor) {
   char col, row;
   for (col = 'a'; col < 'd'; col++)
     for (row = '0'; row < '3'; row++) {
          EXPECT_EQ(' ', ttt_.getMark(col, row));
};
```

for (row = '1'; row < '4'; row++) {

Een klein foutje … grote gevolgen (1/2)

[bron: Wikipedia]



Een klein foutje … grote gevolgen (2/2)

[bron: History's Most (In)Famous Software Failures]

1988 — Buffer overflow in Berkeley Unix finger daemon.
he first internet worm (the so-called Morris Worm) infects between 2,000 and 6,000 computers in less than a day by taking advantage of a buffer overflow. The specific code is a function in the standard input/output library routine called gets() designed to get a line of text over the network. Unfortunately, gets() has no provision to limit its input, and an overly large input allows the worm to take over any machine to which it can connect. Programmers respond by attempting to stamp out the gets() function in working code, but they refuse to remove it from the C programming language's standard input/output library, where it remains to this day.

1998 — E-mail buffer overflow
Several E-mail systems suffer from a "buffer overflow error", when extremely long e-mail addresses are received. The internal buffers receiving the addresses do not check for length and allow their buffers to overflow causing the applications to crash. Hostile hackers use this fault to trick the computer into running a malicious program in its place.

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
⇒ PRE: REQUIRE(<boolse expressie>, <boodschap>)
⇒ POST: ENSURE(<boolse expressie>, <boodschap>)

Design By Contract

```
#include "DesignByContract.h"
...
char TicTacToe::getMark(char col, char row) {
  char result;
  REQUIRE(('a' <= col) && (col <= 'c'),
       "col must be between 'a' and 'c'");
  REQUIRE(('1' <= row) && (row <= '3'),
       "row must be between '1' and '3'");
  result = _board[(int) col - 'a'][(int) row - '1'];
  ENSURE(('X' == result) || ('O' == result) || (' ' == result),
       "getMark returns 'X', 'O' or ' '");
  return result;
}
```

#include met
REQUIRE en ENSURE

Gebruik een "assertion"
aborteert het programma
⇒ fout terug te vinden

naamconventie: gebruik
"result" in post conditie

Contracten = Interface

```
Bestand: TicTacToe.h

char getMark(char col, char row);
   // REQUIRE(('a' <= col) && (col <= 'c'), "col must be between 'a' …
   // REQUIRE(('1' <= row) && (row <= '3'), "row must be between '1' …
   // ENSURE(('X' == result) || ('O' == result) || (' ' == result),
   //       "getMark returns 'X', 'O' or ' '");
```

KOPIEER CONTRACTEN IN .h file

- deel van de klasse-interface
  ⇒ CONTRACT met de buitenwereld
- (soms automatisch gegenereerd
  ⇒ steeds synchroon met de code)

Vuistregel

Contracten gebruiken alleen publieke (geëxporteerde) declaraties

Waarom ?
  contract verifiëerbaar door externe partijen (modules, …)
Hoe ?
  Private (nt. geëxporteerde) declaraties extern niet toegankelijk
  ⇒ Denk goed na over wat je pre- & postcondities nodig hebben

Private declaraties

```
void TicTacToe::setMark(char col, char row, char marker) {
   REQUIRE(('a' <= col) && (col <= 'c'), "col must be between 'a' and 'c'");
   REQUIRE(('1' <= row) && (row <= '3'), "row must be between '1' and '3'");
   REQUIRE(('X' == marker) || ('O' == marker) || (' ' == marker),
         "marker must be 'X', 'O' or ' '");
   _board[(int) col - 'a'][(int) row - '1'] = marker;
   ENSURE(_board[(int) col - 'a'][(int) row - '1'] == marker),
      "setMark postcondition failure");
}
```

_board is *niet* 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

Volgorde van oproepen

```
void TicTacToe::setMark(char col, char row, char marker) {
   REQUIRE(('a' <= col) && (col <= 'c'), "col must be between 'a' and 'c'");
   REQUIRE(('1' <= row) && (row <= '3'), "row must be between '1' and '3'");
   REQUIRE(('X' == marker) || ('O' == marker) || (' ' == marker),
         "marker must be 'X', 'O' or ' '");
   _board[(int) col - 'a'][(int) row - '1'] = marker;
   ENSURE((getMark(col, row) == marker),
         "setMark postcondition failure");
}
```

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

"properlyInitialized" via self pointer

```
class TicTacToe {
 …
private:
 TicTacToe * _initCheck;
 …
};

TicTacToe::TicTacToe() {
 _initCheck = this;
 …
}
bool TicTacToe::properlyInitialized() {
 return _initCheck == this;
}
```

Object heeft een extra pointer …

Wijst normaal naar zichzelf
na initializatie

En kan achteraf
geverifieerd worden

properlyInitialized pre- and postconditie

```
class TicTacToe {
public:
 TicTacToe();
// ENSURE(properlyInitialized(), "constructor must end …");

…

 bool notDone();
// REQUIRE(this->properlyInitialized(),
// "TicTacToe wasn't initialized when calling notDone");

 void doMove();
// REQUIRE(this->properlyInitialized(),
// "TicTacToe wasn't initialized when calling doMove");
 ...
```

"properlyInitialized"
⇒ postconditie alle constructors
⇒ verifiëer in testDefaultConstructor

"properlyInitialized"
⇒ preconditie andere procedures

Universiteit Antwerpen *Betrouwbaarheid*

TicTacToe12: Contracten

Zie TicTacToe.h

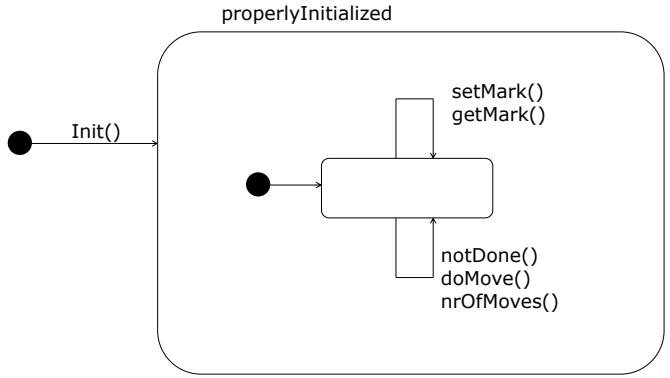
TicTacToe
properlyInitialized (): BOOLEAN notDone() doMove() nrOfMoves() getMark (col, row: CHAR): CHAR setMark (col, row, marker: CHAR)

```
<<precondition>>
properlyInitialized() &
("a"<=col) & (col<="c") &
     ("1"<=row) & (row<="3")

<<postcondition>>
(result = "O") OR (result = "X")
OR (result = " ")
```

```
<<precondition>>
properlyInitialized() &
("a"<=col) & (col<="c") &
     ("1"<=row) & (row<="3")
& ((marker = "O") OR (marker = "X")
OR (marker = " "))

<<postcondition>>
getMark(col, row) = marker
```


Interpretatie Interface

Notatie = UML
(Zie boek "Using UML ...")



Dit is een eindige automaat (cfr. "Talen en Automaten")

Evaluatiecriteria

Inleiding Software Engineering (20?? - 20??) **Tweede Evaluatie**

Student1: student1 - Student2: student2 - Student3: student3

Commentaar:

Contracten

O Nt aanwezig (= geen assert)	O Vrkrd. gebruik (=geen pre-post)	O Beperkt (= alleen NIL)	O Voldoende (= alleen pre-)	O Goed (= soms pre & post)	O Excellent (= vaak pre & post)

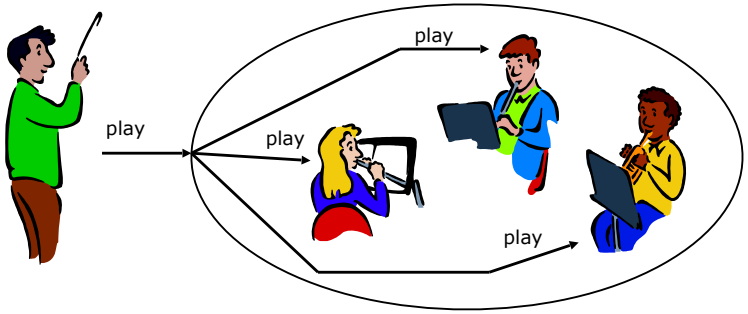
Commentaar:

Beperkt tot initializatie = *onvoldoende*

Gebruik van private/protected attributed
Niet zichtbaar in .h file
= verkeerd gebruik = *onvoldoende*

Complexe Interacties

Een optimale werkverdeling



3. Aanpasbaarheid

versie 13 (displayGame)
- 0, 1, infinity principle
- say things once and only once

versie 14 (displayGame)
- coupling / cohesion
- [testen] ASCII uitvoer

versie 15 (Player)
- domeinmodel

versie 16 (Player.moves)
- vermijd gebruikersinvoer

versie 17 (Player. winner())
- Basisfunctionaliteit

Conclusie

Vereisten

Vereisten	Technieken
• Betrouwbaarheid	• Testen + Contracten
• Aanpasbaarheid	• Objectgericht ontwerp
• Planning	• Tijdsschatting
Werktuigen	
• "Build"	• Make, CMake
• Editor, Debugger	• CLion
• Documentatie	• Doxygen
• Version control	• Git / Github / Gitlab

Magic Numbers

```
void displayGame(TicTacToe& ttt) {
    ...
    for (row = '1'; row <= '3'; row++) {
        cout << row;
        for (col = 'a'; col <= 'c'; col++) {
    ...
}
```

TicTacToe12

```
void TicTacToe::doMove() {
    ...
    col = (char) (_nrOfMoves % 3) + 'a';
    row = (char) (_nrOfMoves / 3) + '1';
    ...
}
```

```
TEST_F(TicTactToeTest, HappyDay) {
...
    for (col = 'a'; col <= 'c'; col++)
        for (row = '1'; row <= '3'; row++) {
...
}
```


Vuistregel

"0, 1, infinity" principle
Allow - none of foo,
- one of foo, or
- any number of foo.

Waarom ?
zet geen arbitraire limieten
⇒ Vroeg of laat zullen deze limieten aangepast moeten worden

Refactor:
creëer een constante die de arbitraire waarde voorstelt

Vuistregel

Variant op "Keep It Stupidly Simple"

say things
once and only once

Waarom ?
duplicatie en redundantie
⇒ veranderen moet op twee, drie … plaatsen
(eentje wordt vroeg of laat vergeten)

Refactor:
creëer abstractie via naam, functie, klasse, …

Y2K

kalenderjaar = laatste twee cijfers

'98 '99 ?? '00 '01

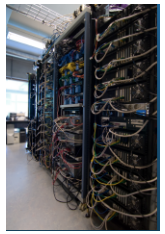
Y2K Cost in Perspective

- (Sources: Gartner Group and Congressional Research Service)
- World War II: $4200 billion
- Millennium bug (Y2K): $600 billion
- Vietnam conflict: $500 billion
- Kobe earthquake: $100 billion
- Los Angeles earthquake: $60 billion

IPv4 address space

Internet Protocol (version 4)
address ruimte = 2^{32}
≈ 4 × 109 ≈ 4 miljard
3 februari 2011

- laatste nummer toegewezen door Internet Assigned Numbers Authority (IANA)



Internet Protocol (version 6)
address ruimte = 2^{128}
≈ 3,4 × 1038
∀ aardbewoner
± 50 quadriljard adressen

Magic Numbers ⇒ Constantes

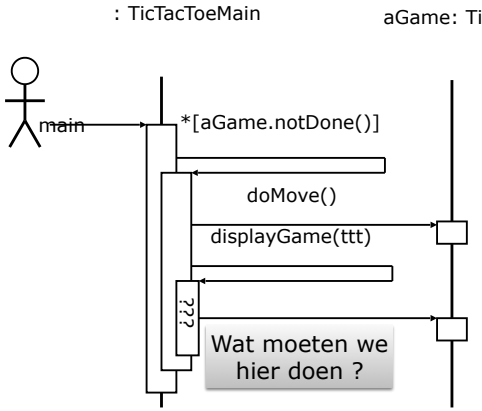
```
void displayGame(TicTacToe& ttt) {
    ...
    for (row = minRow; row <= maxRow; row++) {
        cout << row;
        for (col = minCol; col <= maxCol; col++) {  ...
}
```

TicTacToe13

```
void TicTacToe::doMove() {
    ...
    col = (char) (_nrOfMoves % 3) + minCol;
    row = (char) (_nrOfMoves / 3) + minRow;
    ...
}
```

```
TEST_F(TicTactToeTest, HappyDay) {
...
    for (col = minCol; col <= maxCol; col++)
        for (row = minRow; row <= maxRow; row++) {
...
}
```


TicTacToe13




```
int main(int argc, char **argv) {
    TicTacToe ttt;

    while (ttt.notDone()) {
        ttt.doMove();
        cout << endl << endl;
        displayGame(ttt);
    };
}
```


Pass by Reference vs. Pass by Value (1/2)

TicTacToe13

```
void displayGame(TicTacToe& ttt) {
    ...
}
```

```
TicTacToe numberOfMoves = 1
   a   b   c
 -------------
1 | O |   |   |
2 |   |   |   |
3 |   |   |   |
 -------------
```

```
TicTacToe numberOfMoves = 2
   a   b   c
 -------------
1 | O | X |   |
2 |   |   |   |
3 |   |   |   |
 -------------
```

```
void displayGame(TicTacToe ttt) {
    ...
}
```

```
TicTacToe.cpp:44: failed assertion
    `TicTacToe wasn't initialized …
TicTacToe numberOfMoves = Abort trap: 6
```

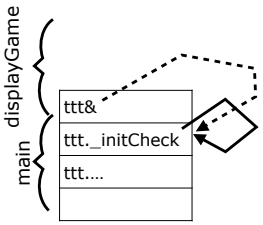
Pass by Reference vs. Pass by Value (2/2)

properlyInitialized ⇒ defensieve programmeerstijl

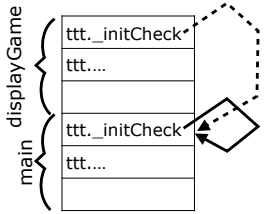
```
void displayGame(TicTacToe& ttt) {
    …

int main(…) {
    TicTacToe ttt;
    …
```

```
void displayGame(TicTacToe ttt) {
    …

int main(…) {
    TicTacToe ttt;
    …
```

Pass by Reference



Pass by Value



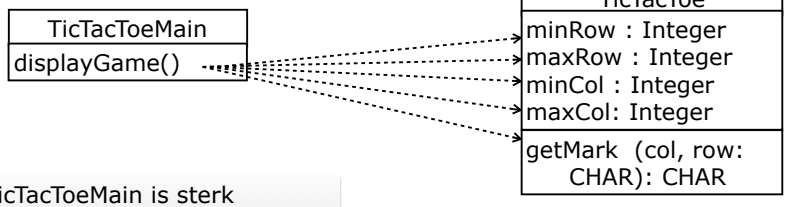
Sterke Koppeling (code)

```
void displayGame(TicTacToe& ttt) {
   char col, row;
   cout << "TicTacToe numberOfMoves = " << ttt.nrOfMoves() << endl;
   cout << "    a   b   c   " << endl;
   cout << "  ------------- " << endl;
   for (row = minRow; row <= maxRow; row++) {
      cout << row;
      for (col = minCol; col <= maxCol; col++) {
         cout << " | " << ttt.getMark(col, row);
      }
      cout << " |" << endl;
   };
   cout << "  ------------- " << endl;
}
```

Afhankelijk van 4 constantes & 1 operatie

Sterke Koppeling

TicTacToe13



TicTacToeMain is sterk gekoppeld aan TicTacToe
- Er zijn veel afhankelijkheden van TicTacToeMain naar TicTacToe
- Een verandering aan TicTacToe betekent meestal dat we ook TicTacToeMain moeten aanpassen

⇒ Aanpasbaarheid :(

Vuistregel

Plaats het gedrag dicht bij de data

Waarom ?
- veranderingen zullen eerder lokaal effect hebben ⇒ zwakke koppeling

Refactor:
- Routines die veel "get" operaties oproepen ⇒ verplaats naar de corresponderende klasse.

Zwakke Koppeling

TicTacToe13 → TicTacToe14

TicTacToeMain
displayGame(ttt TicTacToe)

```
void displayGame(TicTacToe& ttt) {
   ...
   for (row = minRow;
      row <= maxRow; row++) {
      ... ttt.getMark(col, row);
   ...
}
```

TicTacToe
displayGame ()

```
void TicTacToe::displayGame() {
   ...
   for (row = minRow;
      row <= maxRow; row++) {
      ... this->getMark(col, row);
   ...
}
```

TicTacToeMain is *zwak* gekoppeld aan TicTacToe
- Er is maar één afhankelijkheid van TicTacToeMain naar TicTacToe
- Een verandering aan TicTacToe heeft zelden een effect op TicTacToeMain

⇒ Aanpasbaarheid :)

Koppeling via `cout`

```
void TicTacToe::displayGame() {
   char col, row;
   REQUIRE(this->properlyInitialized(),
      "TicTacToe wasn't initialized when calling displayGame");
   std::cout << "TicTacToe numberOfMoves = " << this->nrOfMoves()
      << std::endl;
   std::cout << "    a   b   c   " << std::endl;
   std::cout << "  ------------ " << std::endl;
   for (row = minRow; row <= maxRow; row++) {
      std::cout << row;
      for (col = minCol; col <= maxCol; col++) {
         std::cout << " | " << this->getMark(col, row);
      }
      std::cout << " |" << std::endl;
   };
   std::cout << "  ------------ " << std::endl;
}
```

TicTacToe is afhankelijk van `cout`
⇒ alleen uitvoer op console ☹

Vuistregel

Nooit user-interface code in de basisklassen

Waarom ?
- Minder kans op veranderingen in basisklassen (user-interface wordt vaak veranderd)

Refactor:
- Extra parameter als in/uitvoerkanaal naar de buitenwereld

Koppeling via `ostream`

```
void TicTacToe::writeOn(std::ostream& onStream) {
   char col, row;
   REQUIRE(this->properlyInitialized(),
         "TicTacToe wasn't initialized when calling displayGame");
   onStream << "TicTacToe numberOfMoves = " << this->nrOfMoves() <<
std::endl;
   onStream << "    a   b   c   " << std::endl;
   onStream << "  ------------ " << std::endl;
   for (row = minRow; row <= maxRow; row++) {
      onStream << row;
      for (col = minCol; col <= maxCol; col++) {
         onStream << " | " << this->getMark(col, row);
      }
      onStream << " |" << std::endl;
   };
   onStream << "  ------------ " << std::endl;
}
```

Extra parameter als uitvoerkanaal
+ betere naamgeving

Cohesie

TicTacToe14

TicTacToe
TicTacToe(); getMark (col, row: CHAR): CHAR setMark (col, row, marker: CHAR): CHAR notDone (): BOOLEAN doMove () writeOn (o: ostream)

```
<<precondition>>
properlyInitialized() &
("a"<=col) & (col<="c") &
    ("1"<=row) &
(row<="3")

<<postcondition>>
(result = "O") OR (result
= "X")
OR (result = " ")
```

```
<<precondition>>
…

<<postcondition>>
getMark(col, row) =
marker
```

TicTacToe is redelijk cohesief
- 1 operatie gebruiken impliceert ook het gebruik van alle andere
- alle operaties zijn nodig/nuttig

⇒ veranderingen aan de interface zijn weinig waarschijnlijk

⇒ Aanpasbaarheid ☺

Koppeling vs. Cohesie

Koppeling
= mate waarin een component afhankelijk is van andere componenten

te MINIMALISEREN
⇒ veranderingen hebben minder impact

Cohesie
= mate waarin de onderdelen van een component afhankelijk zijn van elkaar

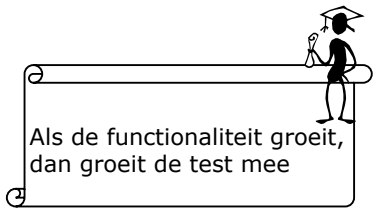
te MAXIMALISEREN
⇒ veranderingen zijn minder waarschijnlijk

?? Ideaal ??
= een component die niks doet
⇒ perfectie is niet haalbaar

Testen van ASCII uitvoer (TicTacToe14)

TicTacToe
TicTacToe(); getMark (col, row: CHAR): CHAR setMark (col, row, marker: CHAR): CHAR notDone (): BOOLEAN doMove () writeOn (o: ostream)

Hoe testen we de uitvoer ?



Vuistregel

Test lange uitvoer door het vergelijken van files.

Waarom ?
- Veel & frequent testen ⇒ makkelijk om steeds uitvoer te testen

Hoe ?
- Verwachte uitvoerdata wordt één maal manueel gecreëerd
- vergelijk verwachte uitvoer met gegenereerde uitvoer

Testen van uitvoer

```
TEST_F(TicTactToeTest, OutputHappyDay) {
   ASSERT_TRUE(DirectoryExists("testOutput"));
   //if directory doesn't exist then no need in proceeding with the test

   ofstream myfile;
   myfile.open("testOutput/happyDayOut.txt");
   while (ttt_.notDone()) {
      ttt_.doMove();
      ttt_.writeOn(myfile);
   };
   myfile.close();
   EXPECT_TRUE(
        FileCompare("testOutput/happyDayExpectedOut.txt",
          "testOutput/happyDayOut.txt"));
}
```

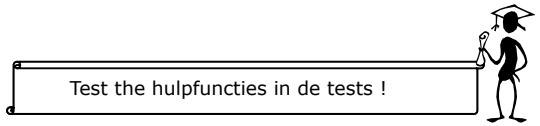
Schrijf alles op een bestand …

… en vergelijk met wat je verwacht

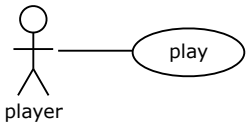
code voor "DirectoryExists" en "FileCompare" ⇒ Zie TicTacToe14

Testing of `FileCompare`

```
TEST_F(TicTactToeTest, FileCompare) {
   ASSERT_TRUE(DirectoryExists("testOutput"));
   ...
   //compare 2 empty files
   EXPECT_TRUE(FileCompare("testOutput/file1.txt", "testOutput/file2.txt"));
   ...
   //compare an empty and a non-empty files
   ...
   //compare two equal files
   ...
   //compare 2 non-empty files which are off by a character in the middle
   ...
   //compare 2 non-empty files where one is one character shorter than the
other
   ...
   //compare existing against non existing file
   }
```



TicTacToe15



Waar voegen we dit bij ?

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

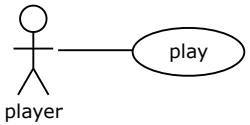
Steps
1. Two players start up a game (First is "O"; other is "X")
2. WHILE game not done
   2.1 Current player makes move
   2.2 Switch current player
3. Announce winner

Vuistregel

Maak een model van het probleemdomein = het DOMEINMODEL

Tips:
- Zelfstandige naamwoord als indicator voor object / klasse.
- Werkwoord als indicator voor operatie
- Naamgeving in basisklassen = naamgeving in probleemdomein

Domeinmodel

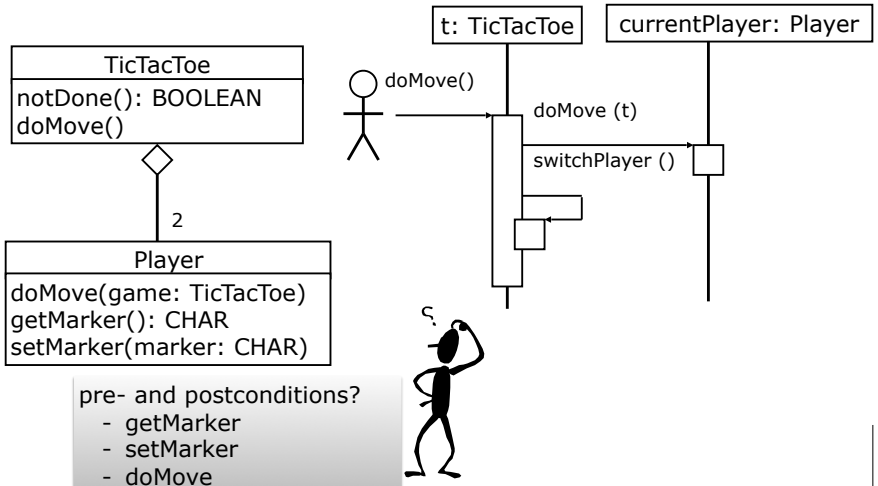


Legende
- Substantief
- *Werkwoord*

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. *Announce* winner

TTT15: Player



pre- and postconditions?
- getMarker
- setMarker
- doMove

Forward Declaration

```
class TicTacToe; // forward declaration

class TicTacToePlayer {
   ...
   void doMove(TicTacToe& game);
   ...
private:
   TicTacToePlayer * _initCheck; //use pointer to myself ...
   char _marker;
};

class TicTacToe {
   ...
private:
   ...
   TicTacToePlayer _players [2];
};
```

TicTacToe & TicTacToePlayer circulair afhankelijk !
⇒ forward declaration

Opgelet: Magic Number

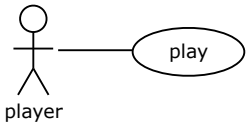
Tests blijven lopen …

```
void TicTacToePlayer::doMove(TicTacToe& game) {
   REQUIRE(this->properlyInitialized(),
         "TicTacToePlayer wasn't initialized when calling getMarker");
   REQUIRE(game.properlyInitialized(),
         "game wasn't initialized when passed to Player->doMove");
   char col, row;
   col = (char) (game.nrOfMoves() % 3) + minCol;
   row = (char) (game.nrOfMoves() / 3) + minRow;
   game.setMark(col, row, _marker);
}
```

Opgelet: Magic Number

Verplaatst van TicTacToe naar Player
Tests blijven lopen ☺

TicTacToe16



Hoe verkrijgen we gebruikersinvoer ?

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. Announce winner

Vuistregel

Een test verwacht géén gebruikersinvoer

Waarom ?
- Veel & frequent testen ⇒ onmogelijk om steeds invoer te geven

Hoe dan wel ?
- Invoerdata wordt gecreëerd door testprogramma (testbestand ?)
- Ontwerp basisklassen onafhankelijk van het data invoer kanaal

TTT16: Reeks van zetten

```
int main(int argc, char **argv) {
   TicTacToe ttt;
   ttt.setMoves("a1c1b2a3c3", "b1a2c2b3");

   while (ttt.notDone()) {
      ttt.doMove();
      cout << endl << endl;
      ttt.writeOn(cout);
   };
}
```

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

een reeks zetten
⇒invoerdata door (test)programma

…

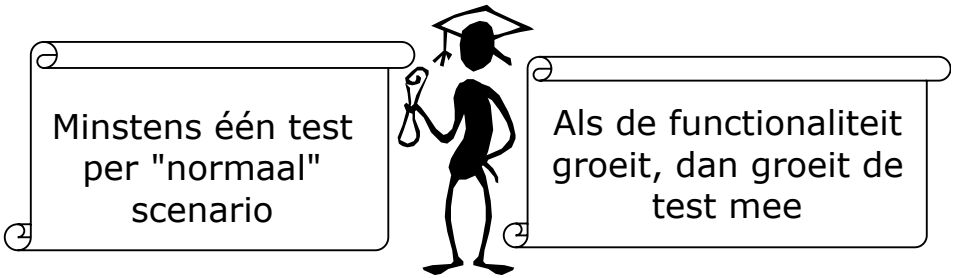
```
void TicTacToe::setMoves(const string oMoves, const string xMoves) {
// REQUIRE(this->properlyInitialized(), "TicTacToe ...
// REQUIRE(legalMoves(oMoves), "TicTacToe::setMoves requires ...
// REQUIRE(legalMoves(xMoves), "TicTacToe::setMoves requires ...
```


Geen gebruikersinvoer in Tests

```
// Tests the "happy day" scenario
TEST_F(TicTactToeTest, HappyDay) {
   EXPECT_EQ(0, ttt_.nrOfMoves());
   EXPECT_TRUE(ttt_.notDone());
   ttt_.setMoves("a1c1b2a3c3", "b1a2c2b3");
   while (ttt_.notDone()) {
      ttt_.doMove();
   };
   char col, row;
   bool markIsX = false; // start with 'O'
   for (col = minCol; col <= maxCol; col++)
      for (row = minRow; row <= maxRow; row++) {
         if (markIsX)
            EXPECT_EQ('X', ttt_.getMark(col, row));
         ...
}
```

Geen gebruikersinvoer

Vuistregels (vorige week)



Waarom ?
- Minimum veiligheidsnorm

Hoe ?
- Controleer de tussentijdse toestand via ”EXPECT_…”

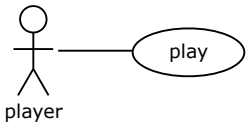
Waarom ?
- Veiligheidsmaatregel: gecontroleerde groei

Hoe ?
- Pas bestaande tests aan

Tests groeien mee …

```
// Tests whether a given string of moves is legal
TEST_F(TicTactToeTest, LegalMoves) {
   EXPECT_TRUE(tttPlayer_.properlyInitialized());
   EXPECT_TRUE(legalMoves(""));
   EXPECT_TRUE(legalMoves("a1"));
   …
   EXPECT_TRUE(legalMoves("a1a2a3b1b2b3c1c2c3"));
   //illegal moves
   EXPECT_FALSE(legalMoves(" "));
   …
   EXPECT_FALSE(legalMoves("b1c4a1"));
}

// Tests the "happy day" scenario for a player
TEST_F(TicTactToeTest, HappyDayPlayer) {
   EXPECT_TRUE(tttPlayer_.properlyInitialized());
   tttPlayer_.setMarker('O');
   EXPECT_EQ('O', tttPlayer_.getMarker());
   …
}
```


TicTacToe17



Announce winner
Waar voegen we dit bij ?

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. Announce winner

Conclusie

Eerst moet je het doen
- KISS principle: klein beginnen en langzaam groeien
- tests lopen + contracten worden nageleefd

Daarna moet je het *goed* doen
- lage koppeling, hoge cohesie
- model van het probleemdomein

} goed ontwerp

- en ... tests blijven lopen + contracten blijven nageleefd

Vuistregels

Ontwerpen
- "0, 1, infinity" principle
- Say things once and only once
- Plaats het gedrag dicht bij de data
- Nooit user-interface code in de basisklassen
  - *Zeker* niet voor invoer — frequent testen
- Maak een model van het probleemdomein (= het domeinmodel)
  - zelfstandige naamwoorden & werkwoorden als indicators

Testen
- Test lange uitvoer door het vergelijken van files
  - Test the hulpfuncties in de tests !
- Een test verwacht géén gebruikersinvoer

Evaluatie Criteria (Tests)

Inleiding Software Engineering (20?? - 20??) **Tweede Evaluatie**

Student1: student1 - Student2: student2 - Student3: student3

Commentaar:

Tests

O Nt aanwezig (= geen tests)	O Vrkrd. gebruik (= manuele test)	O Beperkt (= nt. alles getest)	O Voldoende (= geen uitvoer, verkeerde invoer)	O Goed (= normale test)	O Excellent (= testcode goed)

Een test verwacht géén gebruikersinvoer

Test lange uitvoer door het vergelijken van files

Evaluatie Criteria (Ontwerp)

Objectgericht Ontwerp *(Alles aankruisen!)*

O Geen	O Inheritance	O Reuse Lijsten	O Goeie ADT	O Obj. collaboratie	O Controle Obj. printobjecten/iterators

Data encapsulatie

O geen O aparte files O rechtstreekse access O getters/sette…ly

Smelly code

O Long method bv.:
O Long parameter list bv.:
O Inappropriate intimacy bv.:

Gefundeerde beslissingen

O Geen reden O intuïtie O doordacht

Nooit user-interface code in de basisklassen

Maak een model van het probleemdomein

Plaats het gedrag dicht bij de data

"0, 1, infinity" principle
Say things once and only once

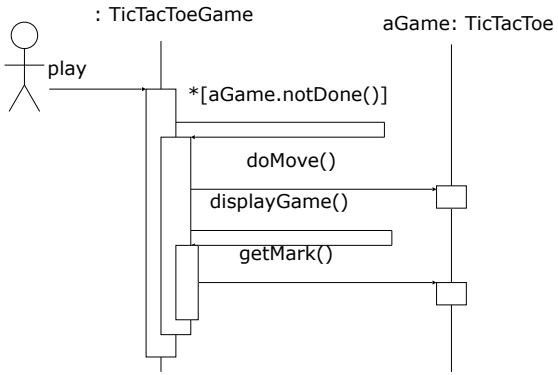
Vuistregel

schrijf ook een test voor ContractViolations

Waarom ?
kijk na of de ABNORMAL END wel degelijk gebeurt
Hoe ?
schrijf tests met oproep illegale waardes
gebruik EXPECT_DEATH ter verificatie

```
EXPECT_DEATH(ttt_.getMark('a' - 1, '1'), "failed assertion");
```


TicTacToe12



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
- Kopieer contracten in .h file
- Contracten gebruiken alleen publieke (geëxporteerde) declaraties
- Contracten bepalen de volgorde van oproepen.
- Gebruik pre-conditie om de initializatie van objecten te controleren
- schrijf ook een test voor ContractViolations (commentaar)

Hoe snel loopt iemand de 100 meter ?

4. Planning

- Tijdsschatting
  + Analogie & Decompositie
  + Empirische schatting
    - Plan 2.0 & Plan 2.1
- Conclusie
- TicTacToe17 en TicTacToe18
    - Player. winner()
- Enkele vuistregels
  + Hollywood principe
  + 3-lagen architectuur — TicTacToe19 & TicTacToe20
    - ⇒ invoer tests + uitvoer tests + domein test
  + Hollywood principe - revisited (TicTacToe21)
    - Template Method and Hook Method
  + Contracten & Interface

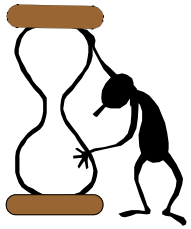
- Continuous Integration
  + Werktuigen

Vereisten

Vereisten	Technieken
• Betrouwbaarheid	• Testen + Contracten
• Aanpasbaarheid	• Objectgericht ontwerp
• Planning	• Tijdsschatting
Werktuigen	
• "Build"	• Make, CMake
• Editor, Debugger	• CLion
• Documentatie	• Doxygen
• Version control	• Git / Github / Gitlab

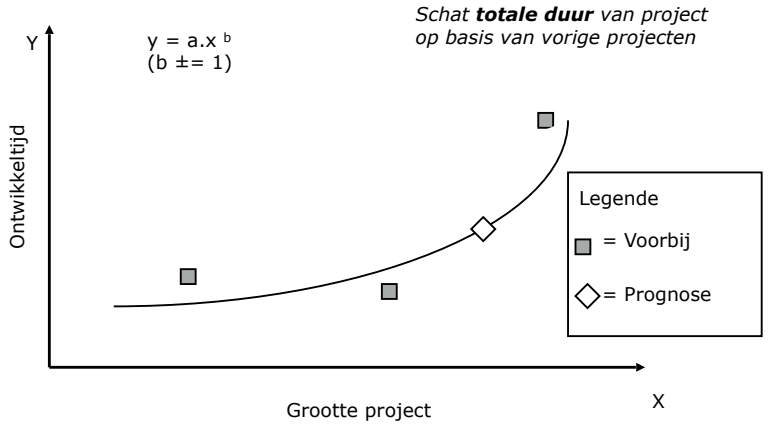
Schatting door analogie

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



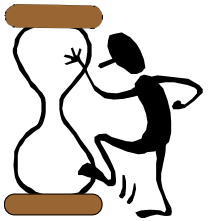
Empirisch
gespendeerde tijd voorbije projecten dient als basis voor schatting volgende

Empirische schatting (project)



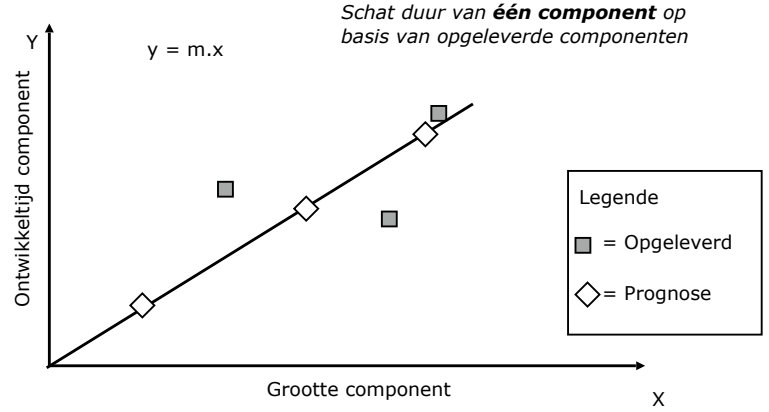
Schatting door analogie

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)



Grootte en Tijd

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

- y = Ontwikkellingstijd ?
  (Zie tijdsbladen)

vergelijking benaderende rechte
$y = m.x$

- Na oplevering n componenten: (x_n, y_n) $\Rightarrow m = \Sigma y_n / \Sigma x_n$
- Schatting y_{n+1} voor grootte x_{n+1} $\Rightarrow y_{n+1} = m.x_{n+1}$

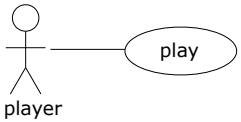
Vuistregel

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



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

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

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

TicTacToe17

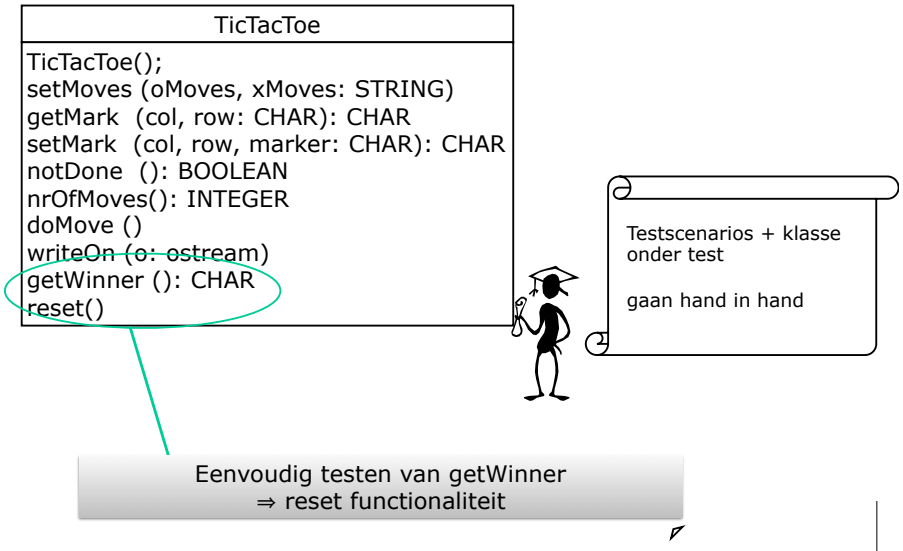
```
class TicTacToe {
    ...
private:
    …
    char _winner;

};

TicTacToe::TicTacToe() {
…
    _winner = ' ';
    ENSURE(properlyInitialized(), "constructor …");
}

void TicTacToe::writeOn(std::ostream& onStream) {
…
    onStream << "TicTacToe numberOfMoves = " << this->nrOfMoves()
             << " - winner = '" << this->getWinner() << "'" <<std::endl;
    …
}
```

winnaar toevoegen impliceert
- - aanpassingen aan de constructor
- - aanpassingen aan "writeOn"

⇒ Testen en HappyDayOutput

- + nieuwe testen voor getWinner

TicTacToe18



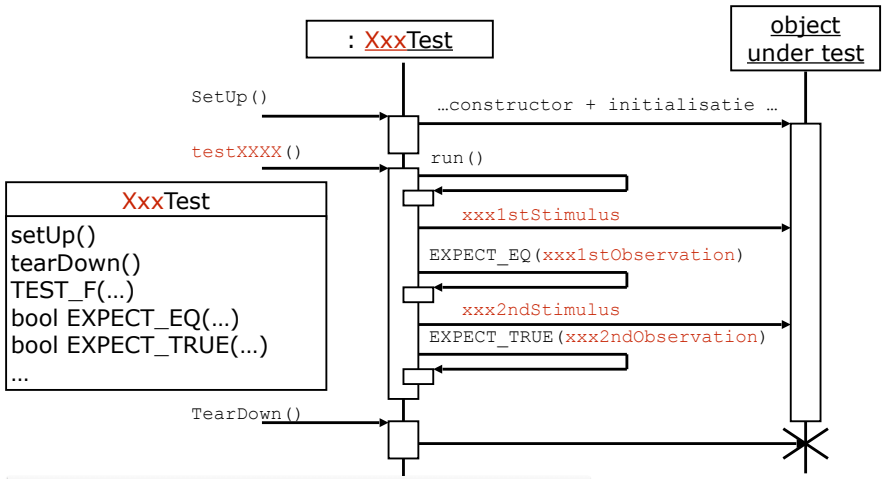
Vuistregel

"Hollywood Principe"
don't call us, we'll call you !

Waarom ?
- Uitbreiding van bibliotheken via subklassen

Hoe ?
- Superklasse in bibliotheek legt protocol van oproepen vast
- Subklassen kunnen gedrag uitbreiden
  + Voorbeeld: UnitTest (mindere mate TinyXML)

Generisch Unittest Protocol



Er zijn veel test-methoden (testcode ≥ basiscode)
Hoe organiseren ?

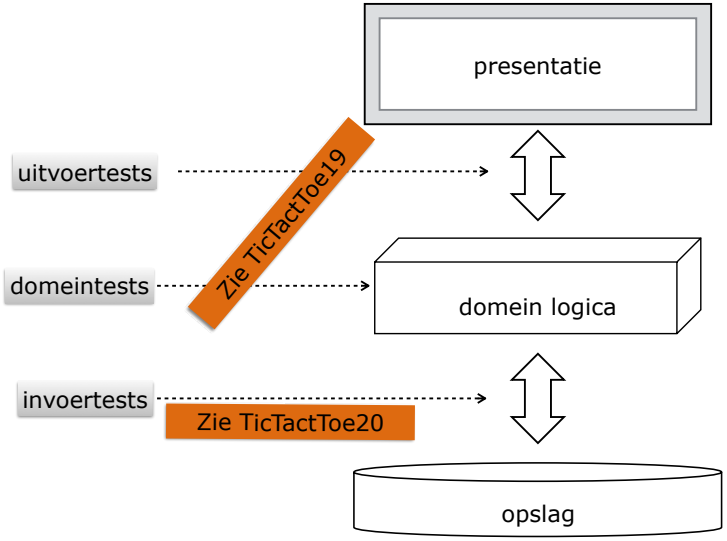
Vuistregel

"3 lagen architectuur"
aparte componenten (en tests !) voor
(a) presentatie,
(b) domein logica,
(c) data-opslag

Waarom ?
- lokaal effect van veranderingen

Hoe ?
- Domeinmodel hangt niet af van databank, noch user-interface
  ⇒ Aparte tests voor invoer / uitvoer / domein

3-lagen architectuur



TicTacToe19

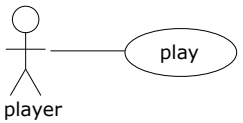
TicTacToeUtils
- FileCompare
- DirectoryExists

TicTacToeTest

TicTacToeDomainTest
- HappyDay
- LegalMoves
- TestWinningColumns
- TestWinningRows
- TestWinningDiagonals
- TestNoWinner
- HappyDayPlayer
- ContractViolations

TicTacToeOutputTest
- OutputHappyDay
- OutputNoWinner

Invoertests



tijdens 2.1 kan "Illegal Move" voorkomen

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

Vuistregel

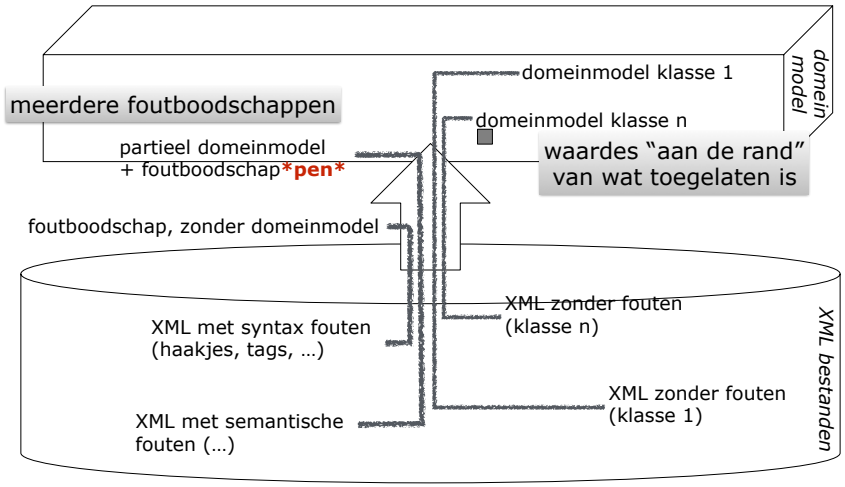
Invoertests:
Minstens één test per
soort "foute" invoer

Waarom ?
- Alle scenarios in de specificaties moeten getest worden
- Hoe ?

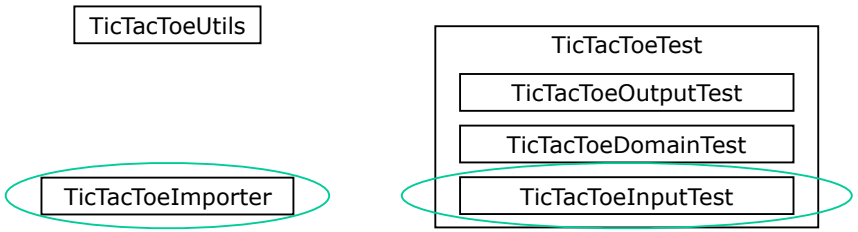
Controleer resultaat uitzondering via "EXPECT_EQ" …
⇒ denk eraan: " Een test produceert zo weinig mogelijk uitvoer"

invoertests

Zie TicTactToe20



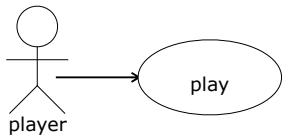
TicTacToe20




```
enum SuccessEnum {ImportAborted,
      PartialImport, Success};
…
static SuccessEnum importTicTacToeGame (
      const char * inputfilename,
      std::ostream& errStream,
      TicTacToe& game);
```

- InputHappyDay
- InputLegalGames
- InputXMLSyntaxErrors
- InputIllegalGames

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. Announce winner

Varianten: Uitvoer op
- simpele tekst
- HTML zonder <TABLE>
- HTML met <TABLE>

Three Variants of Output

```
TicTacToe (5 moves)
   And the winner is 'O'
    a   b   c
  -------------
1 | O |   |   |
2 |   | O |   |
3 |   |   | O |
  -------------
```

TicTacToe (5 moves)

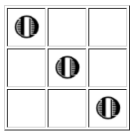
And the *winner* is 'O'.

```
    a   b   c
  -------------
1 | O |   |   |
2 |   | O |   |
3 |   |   | O |
  -------------
```

W3C HTML 4.0

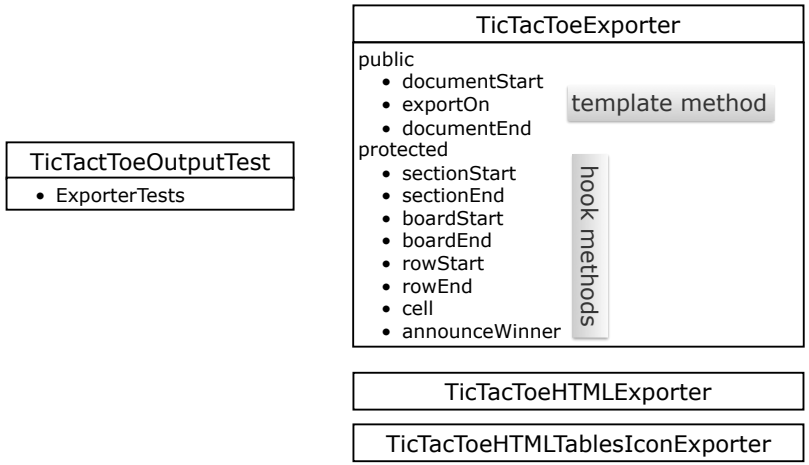
TicTacToe (5 moves)

And the *winner* is  .



W3C HTML 4.0

HollywoodPrinciple Revisited — TicTacToe21



exportOn = Template Method

```
void TicTacToeExporter::exportOn (std::ostream& onStream, TicTacToe &game) {
    char col, row;
    REQUIRE(this->properlyInitialized(), "…
    REQUIRE(game.properlyInitialized(), "…
    REQUIRE(this->documentStarted(), "…

    if (game.nrOfMoves() == 1) {
        this->sectionStart(onStream, "TicTacToe (1 move)");
    } else {
        this->sectionStart(onStream, "TicTacToe ("
            + std::to_string(game.nrOfMoves()) + " moves)");
    };
    if (game.getWinner() != ' ') {this->announceWinner(onStream, game.getWinner())

    this->boardStart(onStream);
    for (row = minRow; row <= maxRow; row++) {
        this->rowStart(onStream, row - minRow + 1);
        for (col = minCol; col <= maxCol; col++) {
            this->cell(onStream, game.getMark(col, row));
        }
        this->rowEnd(onStream);
    };
    this->boardEnd(onStream);
    this->sectionEnd(onStream);
}
```

hook methods

cell = Hook Method

```
void TicTacToeExporter::cell
        (std::ostream& onStream, char const cellMarker) {
    onStream << " | " << cellMarker;}


void TicTacToeHTMLExporter::cell
       (std::ostream& onStream, char const cellMarker) {
    onStream << " | " << cellMarker;}


void TicTacToeHTMLTablesIconExporter::cell
        (std::ostream& onStream, char const cellMarker) {
    onStream    <<  "      <td>";
    exportCellMarkerAsImgRef(onStream, cellMarker);
    onStream << "</td>" << std::endl;
}
```

Vuistregel

Klassen die vaak gebruikt worden
hebben precieze contracten

… en veel tests !

Waarom ?
- Betere betrouwbaarheid door precieze beschrijving interface

Hoe ?
- Leg de “normale” volgorde van oproepen vast
- Specificeer volgorde via de respectievelijke pre- en postcondities
- Schrijf tests die de volgorde (pre- en post-conditions) verifiëren

Contracten & Tests “List” (Double Linked) ?

```
class node
{
public:
int value;
node *next;
node *prev;
};

class list
{
public:
node *front;
node *back;
```

```
list()

void insertFront(int value);
void insertBack(int value);
void removeFront();
void removeBack();
void insertBefore(int value,
   node *nodeB);
void insertAfter(int value,
   node *nodeA);
void removeBefore(node *nodeB);
void removeAfter(node *nodeA);
void removeNode(node *newNode);
void printDListFront();
void printDListBack();
};
```

Contracten ? Tests ?
Moeilijk want interface zonder predicaten

“Lijst” met predicaten

```
//ENSURE(properlyInitialized(), "constructor must end in properlyInitialized state");
list()

//REQUIRE(this->properlyInitialized(), "list wasn't initialized when calling includes");
includes (int value): BOOLEAN;

//REQUIRE(this->properlyInitialized(), "list wasn't initialized when calling insert");
//REQUIRE(~includes(int), "before insert the list should NOT include the inserted value");
//ENSURE(includes(int), "after insert the list should include the inserted value");
insert (int value);

//REQUIRE(this->properlyInitialized(), "list wasn't initialized when calling delete");
//REQUIRE(includes(int), "before delete the list should include the inserted value");
//ENSURE(~includes(int), "after insert the list should NOT include the inserted value”);
delete (int value);
```

Contracten ? Tests ?
Makkelijker want interface met veel predicaten

Vuistregel

Prefereer een interface met predicaten

Waarom ?
- Betere betrouwbaarheid door eenvoudige contracten

Hoe ?
- Specificeer predicaten voor hoofdfunctionaliteit component
- Roep predicaten op in pre- en post-conditions

Vuistregels

Betrouwbaarheid
- Invoertests: Minstens één test per soort "foute" invoer
- Klassen die vaak gebruikt worden: precieze contracten + veel tests
- Prefereer een interface met predicaten

Ontwerpen
- "Hollywood Principle" – wij roepen jouw op als we je nodig hebben
  + Template & Hook Methods
- "3 lagen architectuur"
  + aparte componenten voor (a) presentatie,
    (b) domein logica, (c) data-opslag

Plannen
- Empirische schatten is de basis voor een realistische planning

Evaluatie Criteria (Betrouwbaarheid)

Inleiding Software Engineering (20?? - 20??) **Tweede Evaluatie**

Student1: student1 - Student2: student2 - Student3: student3

Commentaar:

Tests

O Nt aanwezig (= geen tests)	O Vrkrd. gebruik (= manuele test)	O Beperkt (= nt. alles getest)	O Voldoende (= geen uitvoer, verkeerde invoer)	O Excellent (= testcode goed)

Commentaar:

Contracten

O Nt aanwezig (= geen assert)	O Vrkrd. gebruik (=geen pre-post)	O Beperkt (= alleen NIL)	O Voldoende (= alleen pre-)	Goed ore & post)	O Excellent (= vaak pre & post)

Invoertests: Minstens één test per soort "foute" invoer

Klassen die vaak herbruikt worden (o.a. lijsten): contracten & tests

Prefereer een interface met predicaten

Evaluatie Criteria (Planning & Ontwerp)

Plan

O Niet aanwezig O Verkeerd gebruikt O Beperkt O Voldoende O Goed O Excellent

Functionaliteit: O Onvoldoende	O Minder/meer dan beloofd	O Zoals beloofd	
1. Invoer		*2. Uitvoer*	
1.1. Inlezen …	VERPLICHT	2.1. … wegschrijven	VERPLICHT
1.2. Inlezen …	VERPLICHT	2.1. … wegschrijven	BELANGRIJK
1.3. Inlezen …	BELANGRIJK	2.2. … acties wegschrijven	VERPLICHT

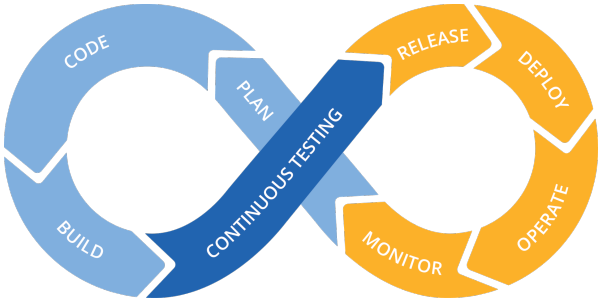
Objectgericht Ontwerp *(Alles aankruisen!)*

O Geen O Inheritance O Reuse Lijsten O Goeie ADT O Obj. collaboratie O Controle Obj. printobjecten/iterators

Empirische schatten is de basis voor een realistische planning

Organisatie van componenten & tests
- "Hollywood Principle"
- "3 lagen architectuur"
- template methods

State-of-the-art Software Construction



Continuous Integration

Tesla
"over-the-air" updates
± once every month

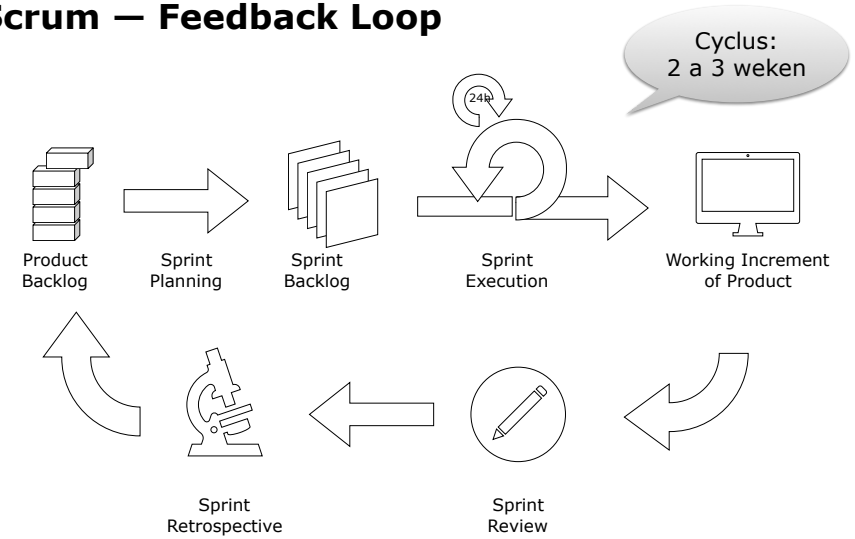
Continuous Delivery
Continuous Deployment

Amazon deploys to production
± every 11,6 seconds

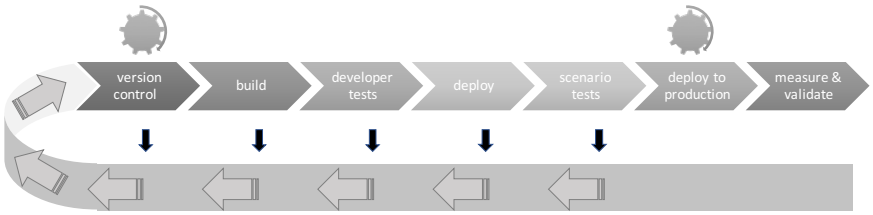
DevOps

September 2015
Amazon Web Services suffered major disruption.
NetFlix recovers quickly!
(Chaos Monkey)

Scrum — Feedback Loop

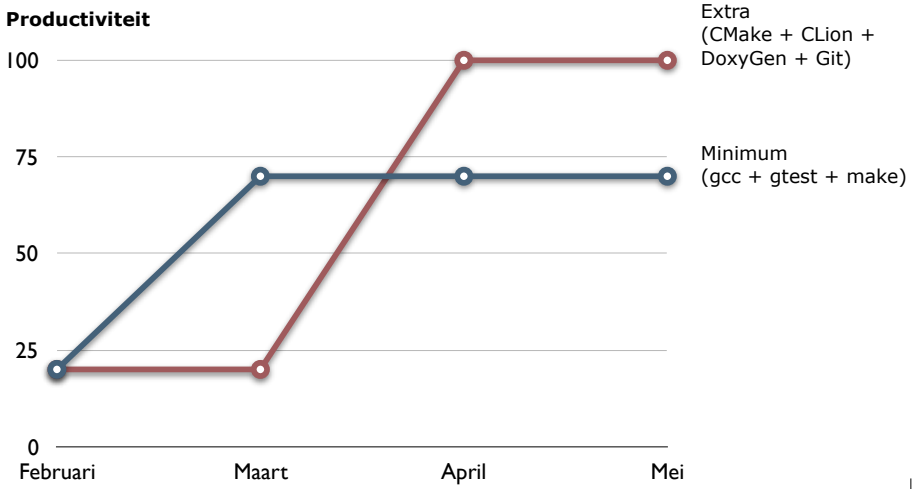


Continuous Integration / Deployment



Vereisten	Technieken
• Betrouwbaarheid • Aanpasbaarheid • Planning	• Testen + Contracten • Objectgericht ontwerp • Tijdsschatting
Werktuigen	
• "Build" • Editor, Debugger • Documentatie • Version control	• Make, CMake • CLion • Doxygen • Git / Github / Gitlab

Leercurve



Evaluatie Criteria (werktuigen)

Project Software Engineering **2DE EVALUATIE**

Student1: ..... - Student2: .....- Student3: .....
Commentaar:

Algemeen

Samenwerking: ☐ blijven samenwerken
Beheersing werktuigen: ☐ makefiles ☐ gtest ☐ tinyxml ☐ cmake ☐ clion ☐ doxygen ☐ fileserver ☐ github
Commentaar:

Minimum	Extra (*)
• gcc • make • gtest • tinyxml	• CMake • CLion • Doxygen • Git / Github / Gitlab

(*) Je krijgt geen extra punten voor het gebruik van werktuigen. Maar eens je het leergeld hebt betaald zou je productiviteit wel moeten stijgen ….

C++ Historiek

Je gebruikt alleen de mogelijkheden van C++98

c++98	c++11	c++14	c++17
1989	2011	2014	2017
• templates • STL • Strings • I/O Stream	• ~~"Move" Semantics~~ • Unified Initialization • ~~"`auto`" keyword~~ • `decltype` • Lambda functions • `constexp` • Multithreading • ~~Smart pointers~~ • Hash tables • `std::array`	• reader writer locks • generalized lambda functions	• fold expressions • constexpr if • structured binding declarations • `string_view` • parallel STL • filesystem library • `std::…`