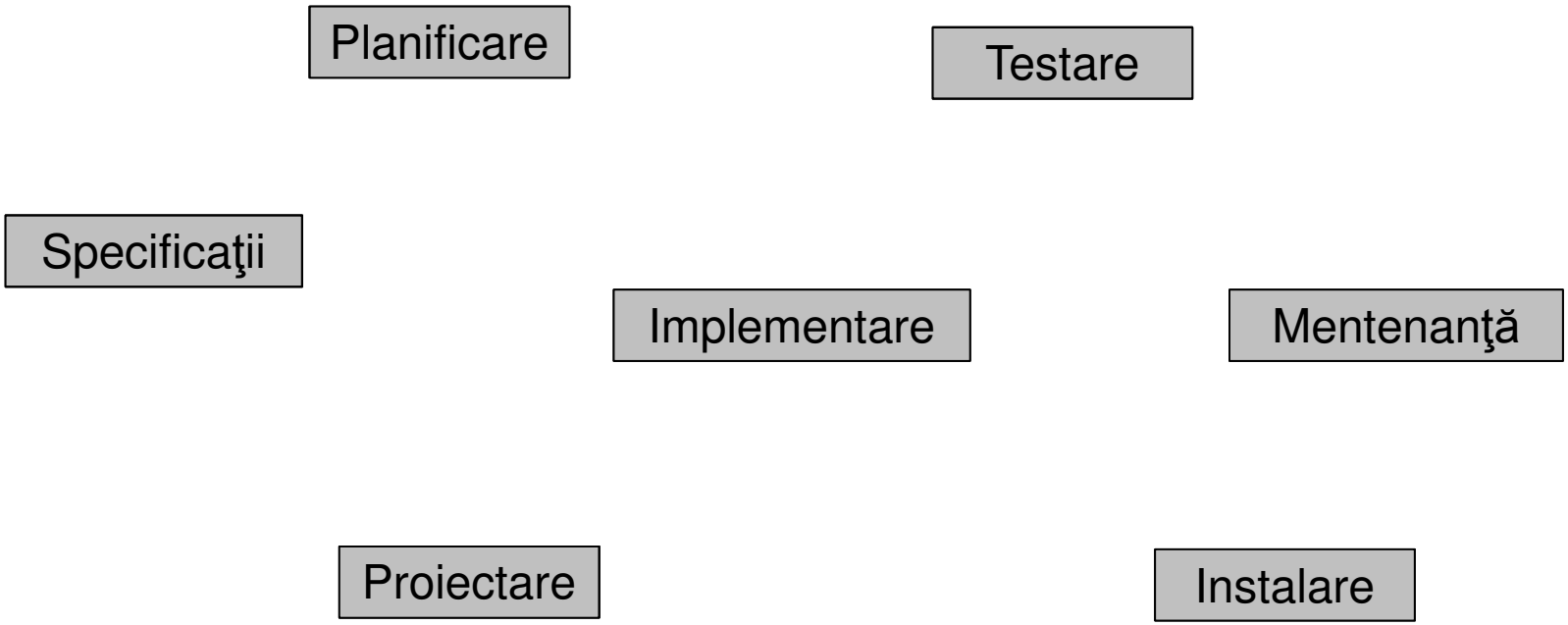
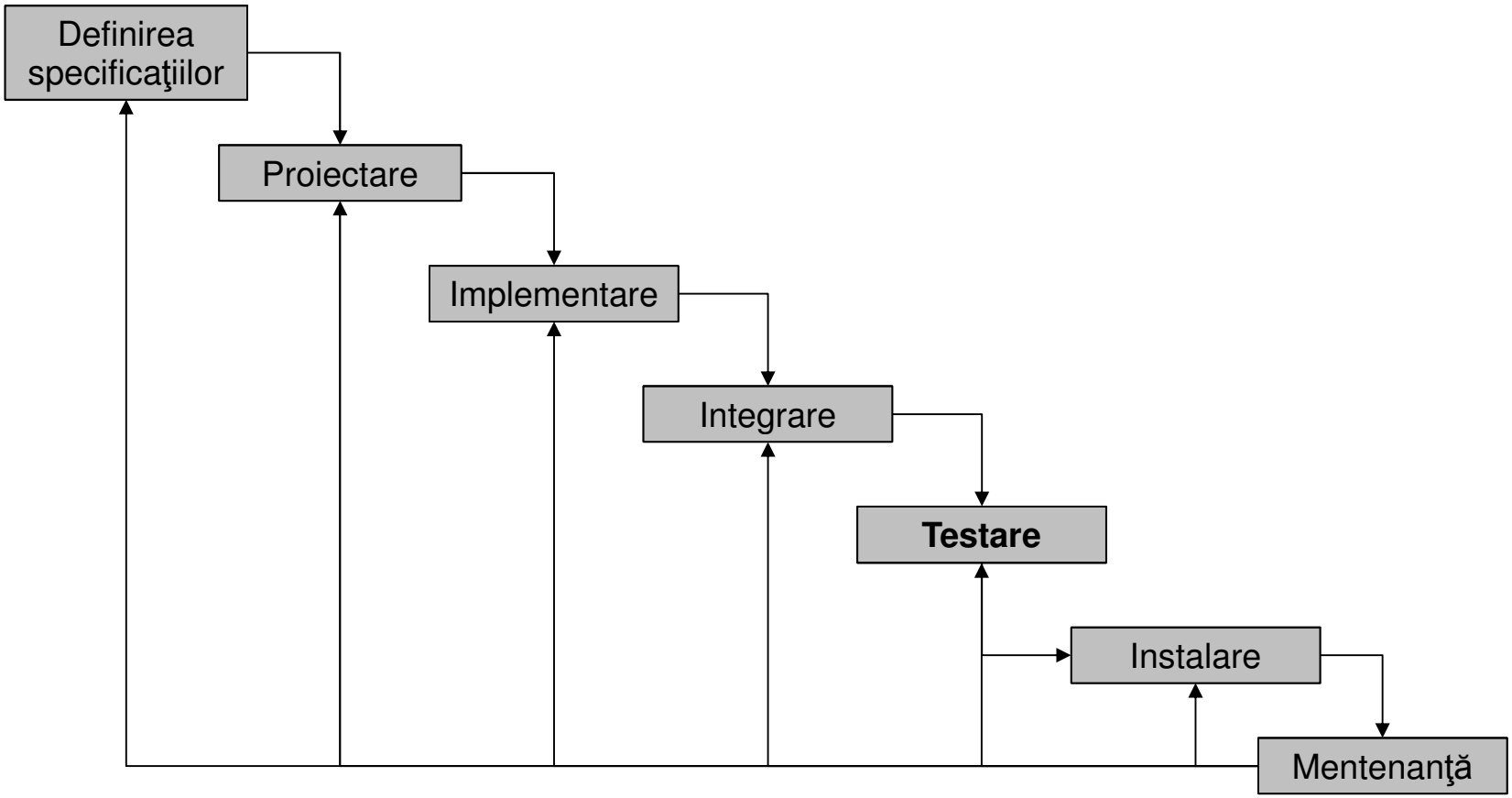
Concepte de Inginerie Software

Etapele unui proiect software



Concepte de Inginerie Software

Modelul Waterfall



Royce, Winston, "Managing the Development of Large Software Systems", Proceedings of IEEE WESCON 26, 1970

Concepte de Inginerie Software

Avantajele/dezavantajele modelului Waterfall

Avantaje:

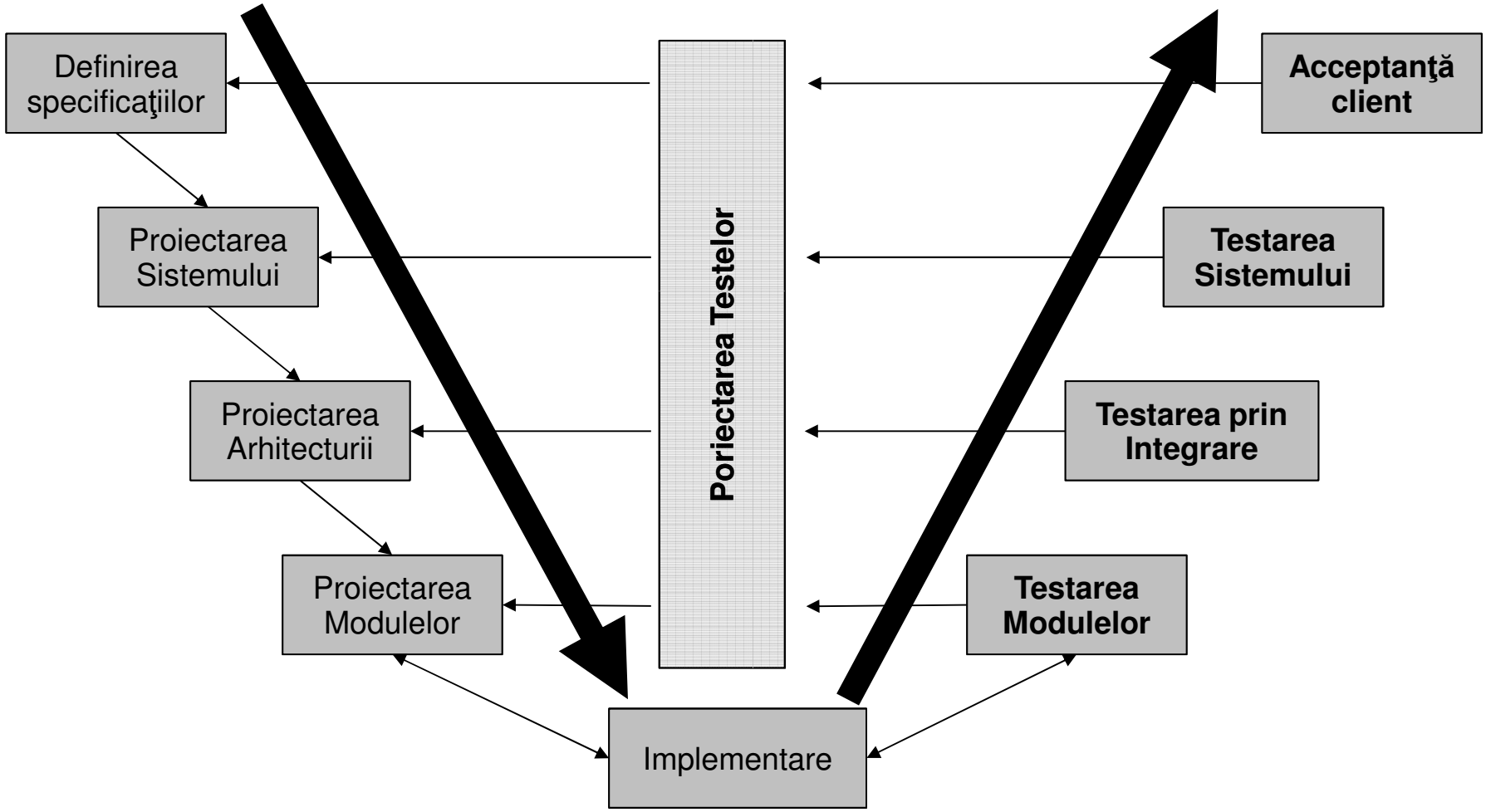
- Abordare secvenţială a proceselor (proiecte mici fără schimbarea pe parcurs a specificaţiilor)
- Impune proiectarea riguroasă a specificaţiilor
- Imagine de ansamblu asupra punctelor critice (milestones)

Dezavantaje:

- Abordare secvenţială a proceselor (proiecte mari)
- Rigiditate la schimbarea cerinţelor clientului
- Modificările pe parcurs implică schimbarea proiectării iniţiale

Concepte de Inginerie Software

Modelul „V”



Concepte de Inginerie Software

Avantajele/dezavantajele modelului „V”

Avantaje:

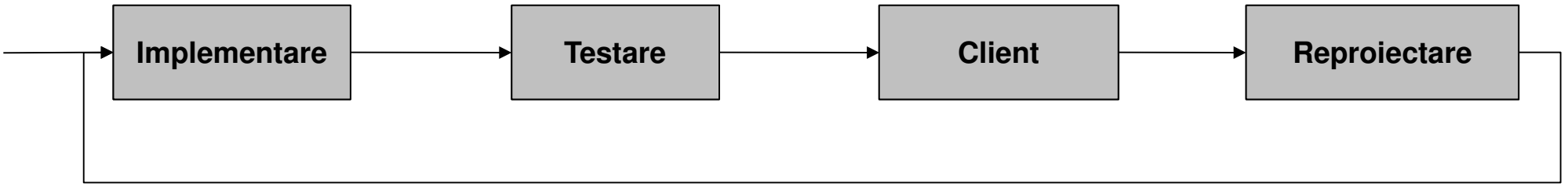
- Îmbunătăţeşte modelul Waterfall
- Conţine cele mai importante etape în dezvoltarea unui produs software
- Conţine principalele etape de testare
- Proiectarea se poate realiza pe parcursul dezvoltării proiectului în funcţie de rezultatele testării

Dezavantaje:

- Aplicabil de obicei în proiectele mari
- Necesită mai multe resurse decât modelul Waterfall

Concepte de Inginerie Software
Extreme programming

- Ţine seama de schimbarea cerinţelor clientului
- Comunicare continuă cu clientul, furnizarea de variante ale produsului la intervale scurte de timp, timeboxing
- Evită implementarea anumitor funcţionalităţi până când este absolut necesar
- Implementarea se porneşte către cea mai simplă soluţie, adăugând funcţionalitate pe parcurs



Ce este Testarea Software ?

„Software testing is any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results.”

Hetzel, 1988

„Software Testing is an empirical investigation conducted to provide stakeholders with information about the quality of the product or service under test.”

Kaner, 2006

“*Testing is questioning a product in order to evaluate it.”*

James Bach

“*Testing can be used very efectively to show the presence of bugs but never to show their absence.”*

Dijkstra

Erori datorate greşelilor în Testarea Software

- NASA Mariner 1 (22 Iulie, 1962)
- un amplificator a cedat în timpul lansării -> distrugerea navei
- codificarea incorectă a unei formule FORTRAN,
- citirea neatentă a specificaţiilor

Erori datorate greşelilor în Testarea Software

- racheta Ariane 5, (4 Iunie 1996), zbor 501 a Agenţiei Spaţiale Europene

- Conversia de la 64-bit float la 16-bit int generează o excepţie de depăşire netratată -> distrugere

- reutilizarea neatentă a codului preluat de la Ariane 4

- cost 500 mil. $

Erori datorate greşelilor în Testarea Software

- Mars Climate Orbiter, 1998
- discrepanţă între unităţi de măsură în sistemele anglo-american şi metric -> dezintegrare
- reutilizarea neatentă a codului preluat de la Ariane 4
- cost aprox. 500 mil. $

Erori datorate greşelilor în Testarea Software

- Mars Pathfinder, 1997

- ajunsă pe Marte, proba spaţială se reseta frecvent, cauza fiind inversiunea de prioritate între procesele cu resurse comune. [Sha, Rajkumar, Lehoczky. Priority Inheritance Protocols]

Descrierea fenomenului :

1. procesul A de prioritate mică obţine resursa R
2. A este întrerupt de C (prioritate mare)
3. C aşteaptă eliberarea lui R, iar A revine în execuţie
4. A este întrerupt de B (prioritate medie, A < B < C) => C aşteaptă terminarea procesului B, fără a fi direct condiţionat de acesta

- soluţie: ridicarea priorităţii unui proces (A) care obţine o resursă, la nivelul celui mai prioritar proces (C) care poate solicita resursa respectivă

Obiectivele Testării Software

- Descoperirea defectelor -> rezolvarea lor
- Verificarea îndeplinirii specificaţiilor
- Verificarea şi validarea produsului software
- Asigurarea calităţii
- Luarea deciziilor de a da drumul pe piaţă produsului software
- Evitarea eliberării premature a produsului pe piaţă
- Minimizarea riscurilor de comercializare
- Predicţie asupra costurilor de suport tehnic
- Prin standardizarea procesului de testare se oferă siguranţă clientului

Probleme/Provocări în Testarea Software

- Timp insuficient pentru procesul de testare
- Prea multe combinații de testat
- Lipsa specificațiilor clare/ modificarea acestora
- Lispa cursurilor specializate
- Lipsa unui proces de testare
- Lipsa instrumentelor software dedicate
- Managementul nu înțelege necesitatea procesului de testare – nu alocă resurse

Testarea Software – mijloc de asigurare a calităţii

- testarea software – factor necesar (nu şi suficient) în procesul de asigurare a calităţii (realizarea specificaţiilor, comunicarea cu clientul, managementul proiectului, ...)
- calitate = „valoare pentru o anumită persoană” [Gerald Weinberg]

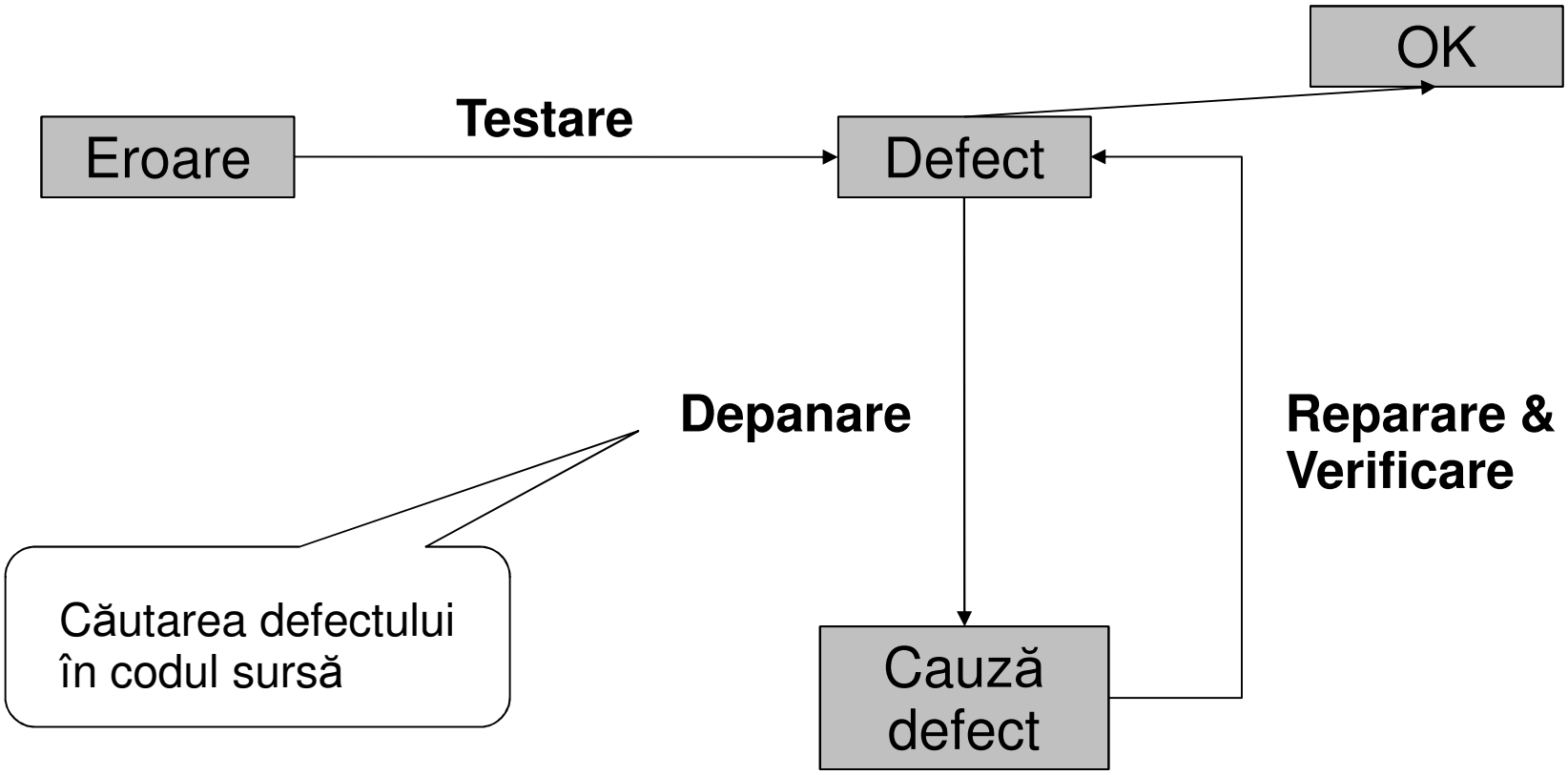
  - diferă de la o persoană la alta

  - calitate = valoare -> mijloace de măsurare a valorii
- calitate (în sensul de acceptanţă client) = motive pro acceptare vs. motive contra acceptare [Joseph Juran]

  - motivele pro acceptare urmărite de către programatori

  - motivele contra acceptare urmărite de către testori – evitarea lor

Testare vs. Depanare



Întrebări esenţiale în Testarea Software [Kaner]

- CE se testează?
- CINE testează?
- Care este STRATEGIA de testare?
- Cum se VERIFICĂ dacă un test a fost trecut cu succes sau nu?
- Când un caz de test este verificat COMPLET?
- Când se consideră FINALIZAT procesul de testare?

Ce se testează ?

- Funcţionalitatea aplicaţiei <- specificaţii
- Comportamentul la rularea concomitentă cu alte aplicaţii
- Comportamentul în diferite configuraţii hardware
- Orice posibile influenţe exterioare (sistem de operare, utilizarea procesorului, procesele concurente,...)
- Lucrul cu memoria

Rolul echipei de testare

- Respectarea procedurilor de testare aferente produsului în cauză
- Realizarea cazurilor de test
- Detecţia defectelor
- Raportarea defectelor
- Rigurozitate şi minuţiozitate pe tot parcursul proiectului

Testarea diferitelor produse software

- pagini web de informare, liste de discuţii
- jocuri pe calculator -> sume mari de bani
- aplicaţii militare -> securitate naţională
- aplicaţii medicale -> viaţa pacienţilor
- aplicaţii mari (x 100 mil Euro)
- aplicaţii medii şi mici

Importanţa testării în diferite etape ale dezvoltări proiectului

- înainte de realizarea primei versiuni (creare de teste, unit testing)
- terminarea primei versiuni – măsurarea riscurilor
- versiunile intermediare
- versiunea pilot -> penalizări
- versiunea finală

Gradele de importanţă a testelor (după risc)

-> diferă de la un proiect la altul

- defecte legate de aspectul vizual (culoare, font, asezare în pagină)
- defecte de GUI (Grapical User Interface) ce pot (nu pot) fi evitate prin anumite metode (workaround)
- defecte funcţionale ce nu respectă specificaţiile
- defecte blocante/critice – esenţiale pentru proiect

Cine face testarea?

-> diferă de la un proiect la altul

- echipa de testare (realizează numai testarea produsului) – uneori face parte din aceeaşi companie ca şi echipa de dezvoltare, alteori este o companie separată

- echipa de dezvoltare (nu este recomandat, deşi este o practică actuală) – procesul de testare este combinat cu cel de depanare

- clientul (în funcţie de caietul de sarcini) – produsul este testat de către client pe parcursul dezvoltării sau doar la final

Testor vs. Programator

- comunicare (în interiorul firmei sau în altă firmă)
- utilitare software de raportare a defectelor
- subiectivitatea programatorului faţă de testor
  - aceeaşi companie – anumite defecte nu sunt raportate/înregistrate
  - companii diferite – programatorii arată mai mult respect echipei de testare – nu au şef comun

Strategii de Testare Software

Black Box	Testarea domeniilor de valori, testarea funcţională, testarea bazată pe specificaţii, testarea axată pe riscuri, testarea la limită, testarea de regresie, testarea bazată pe scenarii, ...
White Box	Testarea acoperirii, testarea API, testarea statică (examinarea codului, verificarea sintaxei), inserarea de defecte
Gray Box	Asemănător cu testarea Black Box, având însă cunoştinţe legate de structura internă a produsului software

Ce este un defect (bug, failure, issue)?

- eroare într-un program software
- abaterea programului software de la specificaţii
- problemă de funcţionalitate,
- nerespectarea aşteptărilor utilizatorilor [Glenford Myers]
- reduce valoarea unei aplicaţii
- „A bug is something that bugs somebody who matters” [James Bach]

Raportarea defectelor-Calitate Software

- diferă de la o companie la alta
- interfaţă testor -> programator -> testor
- aplicaţii de raportare
- înregistrarea etapelor de testare pe parcursul dezvoltării proiectului software
- sistem de evaluare a muncii depuse de către testor -> managerul de proiect
- realizarea statisticilor şi calcularea metricilor de evaluare a stadiului actual în dezvoltarea proiectului

Calitatea unui produs software
[ISO/IEC 9126:1991]

- fiabilitate (frecvenţa defectelor, toleranţa la defecte)
- eficienţă (timp de răspuns, consumarea resurselor)
- mentenanţă (timpul necesar analizei, modificării, testării)
- funcţionalitate (accurateţea respectării specificaţiilor, interconectarea cu alte sisteme, securitate, respectarea standardelor când se impune)
- simplu de folosit (simplu de învăţat, intuitiv, operare rapidă)
- portabilitate (timp necesar instalării, adaptabilitate)

Calitatea unui produs software

Satisfiers vs. Disatisfiers [Joseph Juran]

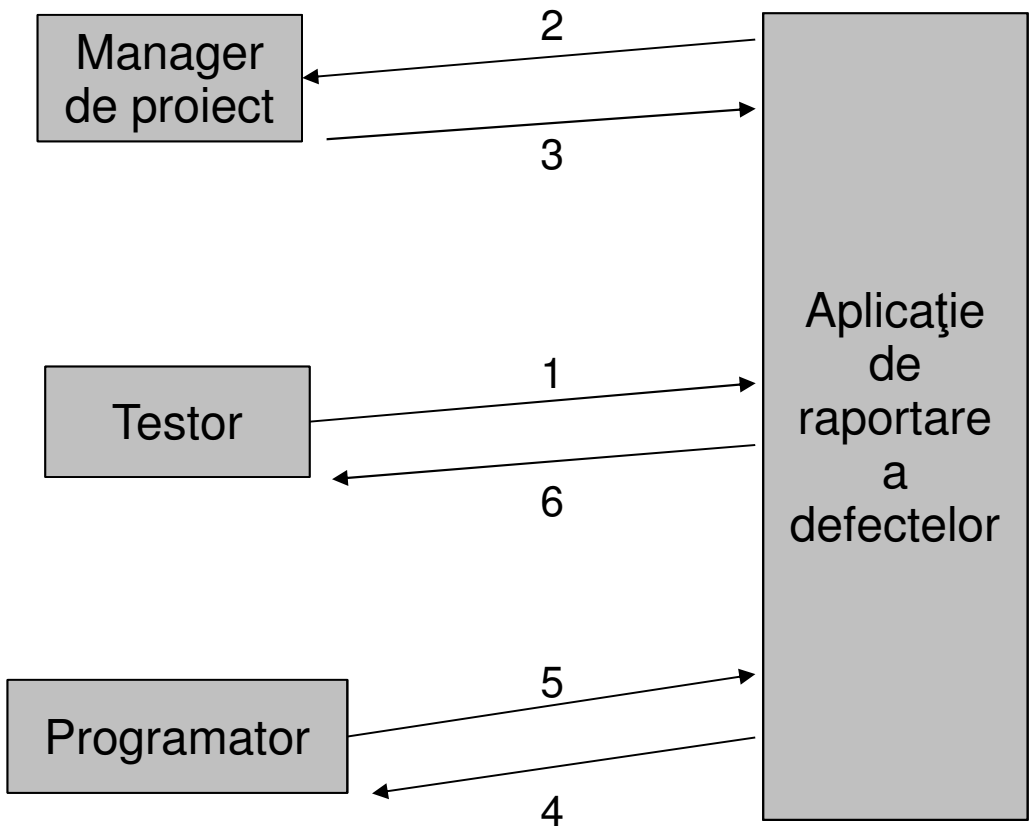
||

Programator vs. Testor

- respectarea specificaţiilor
- funcţionalitate
- dezvoltarea rapidă

- greu de utilizat
- lent în rulare
- nefiabil – erori
- nepotrivit cu mediul de lucru al clientului

Principii de raportare



1. deschiderea unui test şi raportarea unui defect

2,3. atribuirea defectului către un programator

4,5. rezolvarea defectului pentru versiunea indicată de către managerul de proiect

6. testarea prin regresie

- repetarea paşilor dăcă defectul persistă

Tipuri de defecte

- specificaţii greşite – necesită modificare/completare
- cerinţe client greşite (hidden requirements) – necesită completarea/modificarea specificaţiilor
- defecte de implementare – nerespectarea specificaţiilor
- defecte de proiectare – necesită reproiectarea pe baza specificaţiilor
- defecte de documentare

Utilitare de raportare a defectelor

WEB

- Bugzilla – http://www.Bugzilla.org
- Bugaware - http://www.bugaware.com
- TrackStudio - http://www.trackstudio.com
- TaskComplete - http://www.taskcomplete.com
- JIRA - http://www.atlassian.com/software/jira

Bugzilla – principii de raportare

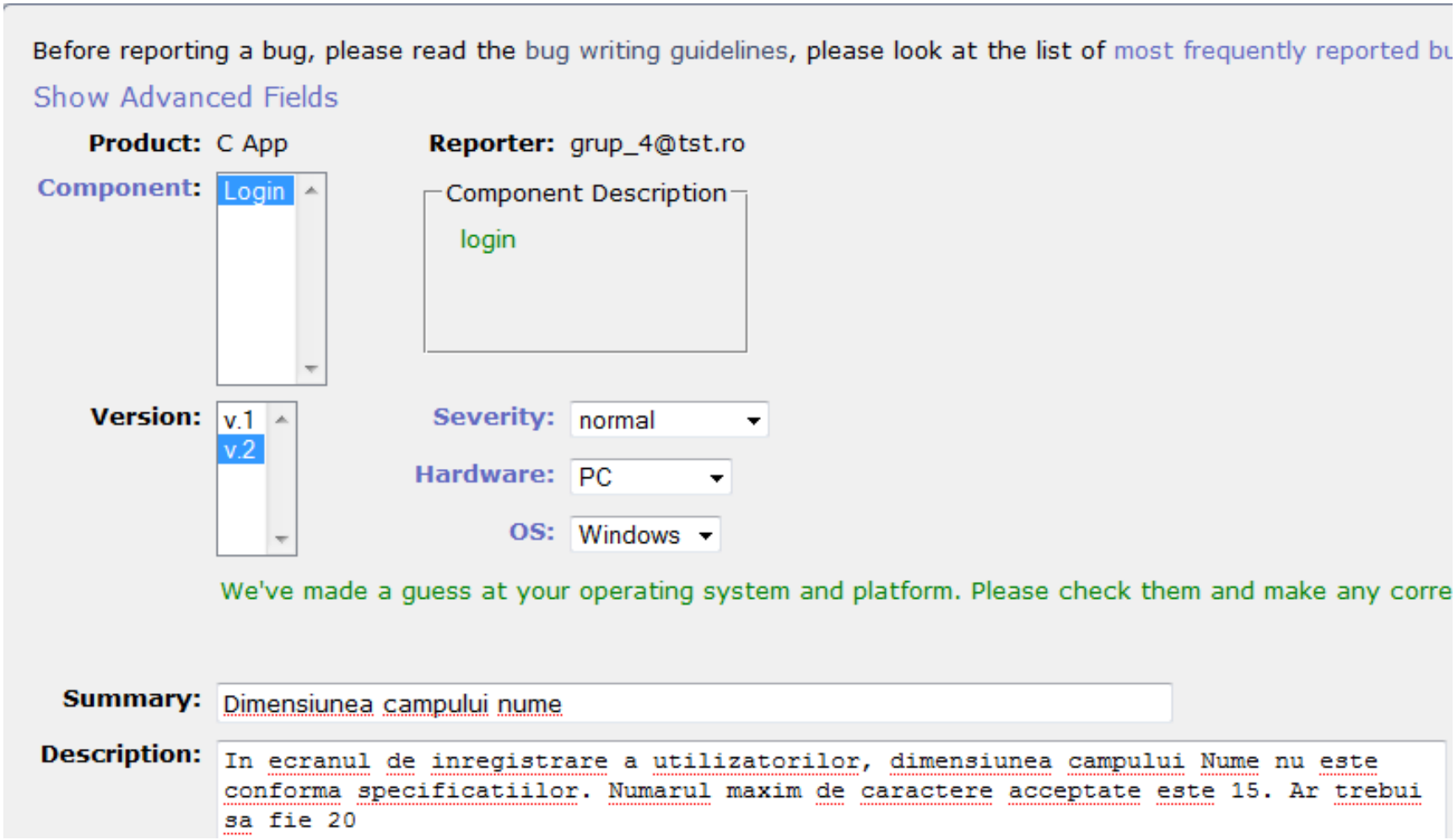
- precizie
- claritate – explicarea paşilor premergători defectului
- un singur defect / raportare
- se raportează orice defect – defect mic -> defect mare
- motivare asupra existenţei defectului

Bugzilla – starea defectelor

- UNCONFIRMED – defectul adăugat în sistem
- NEW – utilizatorii cu drepturi de confirmare (QA/PM) – acceptă defectul
- ASSIGNED – defectul este atribuit unui testor/ grup de testori
- REOPENED – defectul nu a a fost reparat conform indicaţiilor
- RESOLVED – programatorul repară defectul şi consideră procesul terminat
- VERIFIED – procesul de reparare a defectului este verificat
- CLOSED – defectul este considerat inexistent în noua versiune

Bugzilla – Ecran principal

Bugzilla – Crearea unui defect



Bugzilla – importanţa/severitatea unui defect

Stabilită de către testor

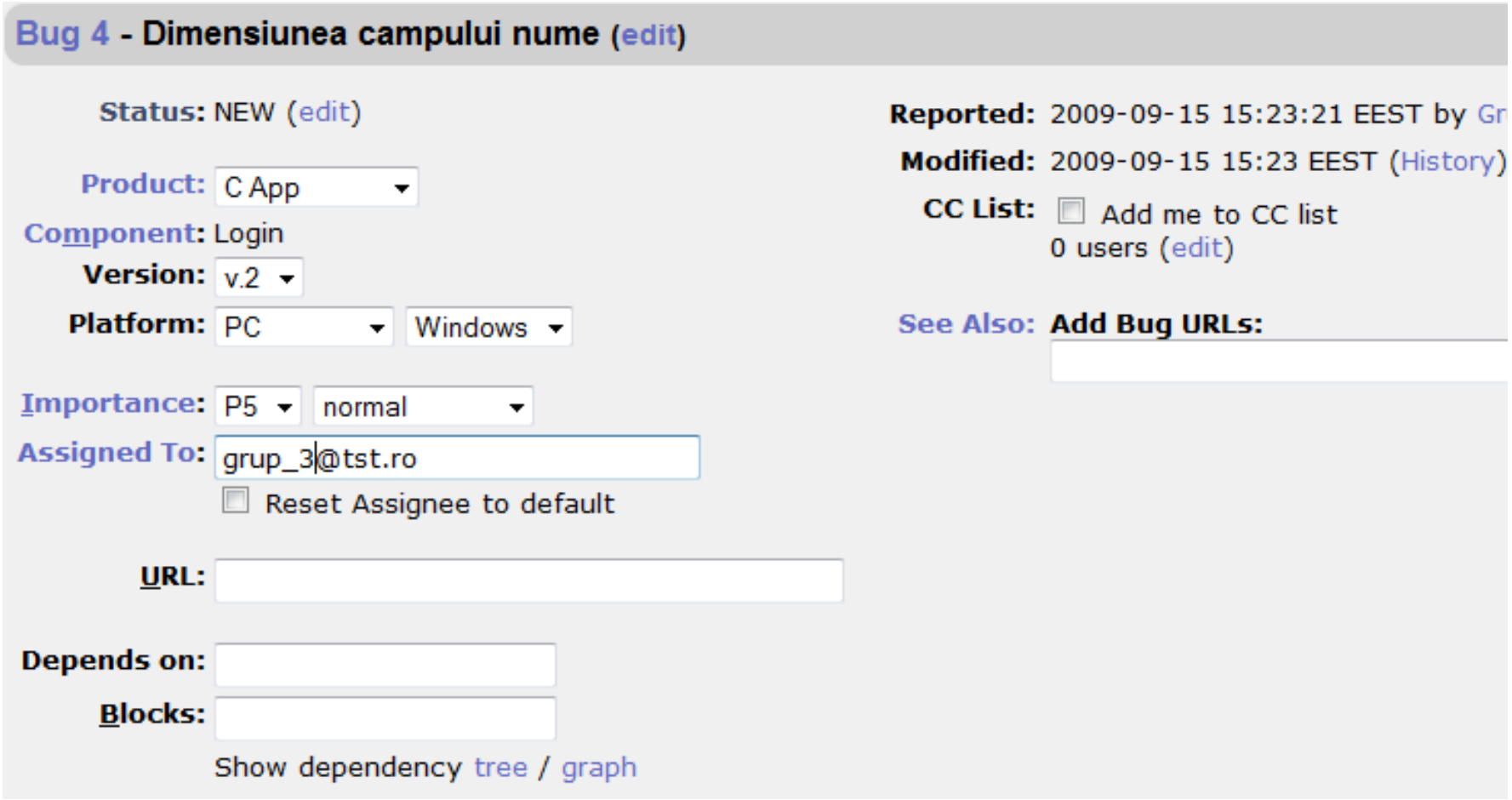
- Blocker – importanţă maximă
- Critical - nu există soluţii de a fi evitat
- Major - există soluţii de a fi evitat
- Normal – nu necesită soluţii speciale de evitare
- Minor - comportament ciudat
- Trivial – problemă de GUI
- Enhancement - îmbunătăţire

Bugzilla – prioritatea unui defect

Stabilită de către PM

- P1 – prioritate maximă
- P2
- P3
- P4
- P5 – prioritate minimă

Bugzilla – atribuirea unui defect către un programator



Bugzilla – vizualizarea defectelor

Status: UNCONFIRMED, NEW, ASSIGNED, REOPENED

ID	Sev	Pri	OS	Assignee	Status Resolution	Summary
1	cri	P5	Wind	grup_1@tst.ro	NEW	domeniu invalid
4	nor	P5	Wind	grup_3@tst.ro	NEW	Dimensiunea campului nume

2 bugs found.

Long Format

XML

Time Summary

CSV | Feed | iCalendar | Change Column

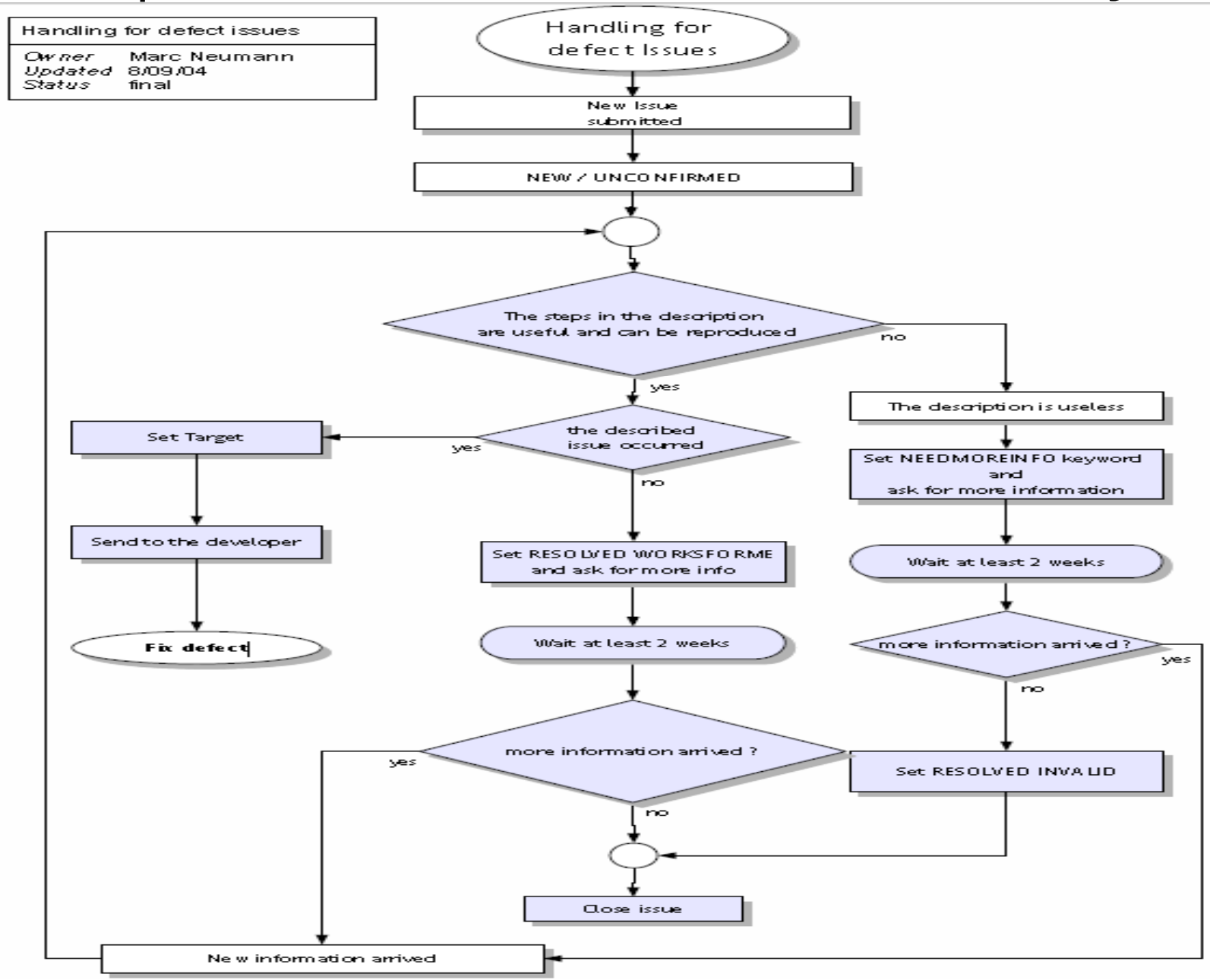
Send Mail to Bug Assignees |

OpenOffice – Issue Tracker

Reguli

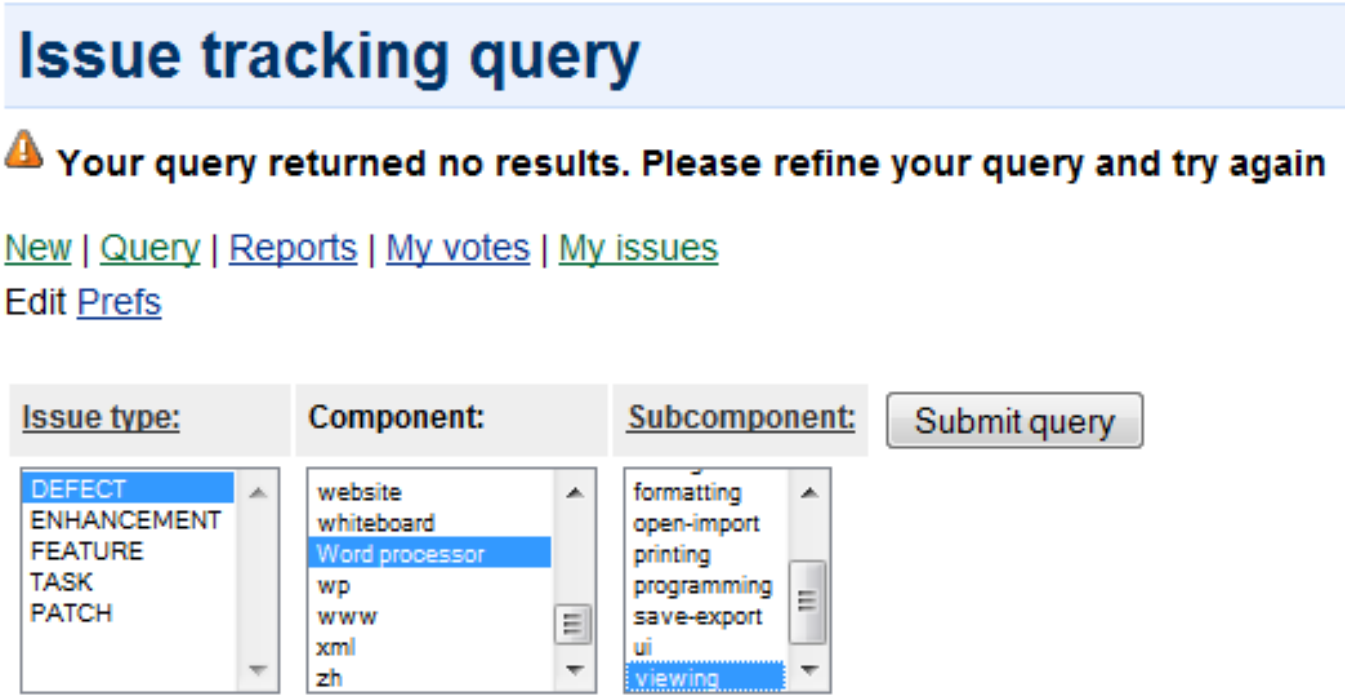
- One problem - One issue
- Provide a meaningful summary
- Provide step-by-step descriptions
- Provide sample documents, if possible
- Use Attachments where possible
- Put all relevant information into the issue

OpenOffice – Issue Tracker. Ciclu de viaţă



OpenOffice – Issue Tracker

Alegerea componentei/subcomponentei OO



OpenOffice – Issue Tracker

Verificarea listei curente de defecte

Issue list

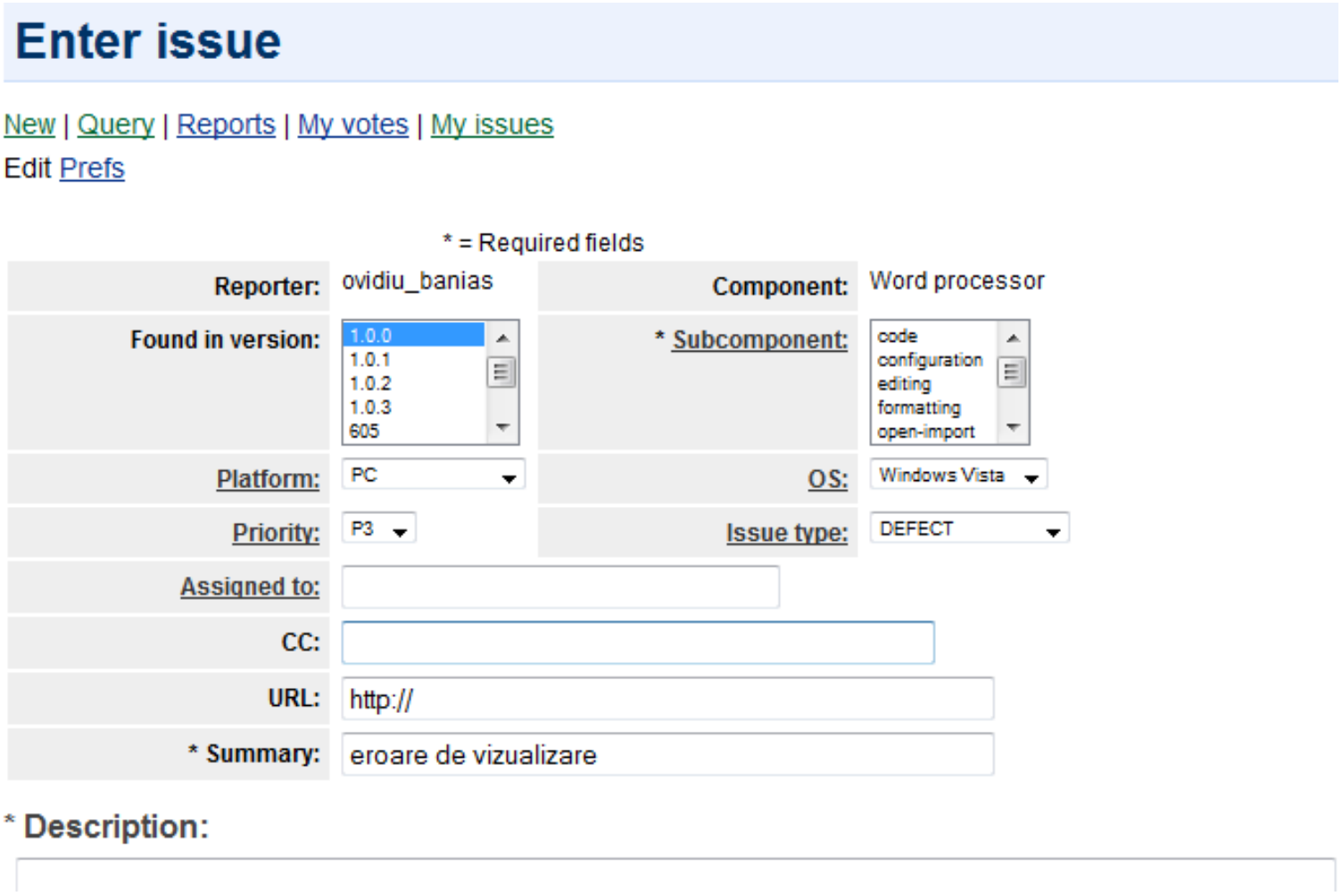
New | Query | Reports | My votes | My issues
Edit Prefs

Tue Sep 15 13:47:45 +0000 2009

ID	Type	Pri	Plat	Owner	State	Resolution
62697	DEFECT	P2	PC	sj	NEW	
82500	DEFECT	P2	All	od	NEW	
90120	DEFECT	P2	PC	hbrinkm	NEW	
91938	DEFECT	P2	Unknown	od	NEW	
103543	DEFECT	P2	All	mav	NEW	
36611	DEFECT	P3	PC	od	NEW	
39136	DEFECT	P3	Other	hbrinkm	NEW	
40538	DEFECT	P3	PC	od	NEW	
43634	DEFECT	P3	PC	cd	STARTE	
44539	DEFECT	P3	Opteron/	fme	NEW	
44572	DEFECT	P3	All	sj	STARTE	

OpenOffice – Issue Tracker

Adăugarea unui defect nou



Elemente cheie în raportare

- Descrierea defectului – cât mai amănunţită
- Motivarea programatorului asupra acceptării şi rezolvării – prezentarea/documentarea amănunţită a defectului.
- Motivarea importanţei/gravităţii alese (Trivial -> Blocking)
- Ataşarea de jurnale (log-uri) şi de capturi de ecran – interesant şi uşor de urmărit/reprodus pentru programator
- Alegerea unei versiuni actuale – programatorul nu va fi încântat să testeze o versiune învechită şi probabil nu va da importanţă sau va cere testarea unei versiuni mai noi
- Indicarea cât mai amănunţită a configuraţiei sistemului testat

Informaţii necesar a fi completate în raportarea defectului

- Nume defect
- Produsul/Componenta/Subcomponenta
- Versiunea produsului
- Platforma
- Sistemul de operare
- Importanţa/gravitatea/severitatea defectului
- Descriere amănunţită

Descrierea amănunţită a defectului

1. Descriere succintă şi la obiect a defectului

2. Enumerarea paşilor realizaţi pentru obţinerea defectului

3. Ataşarea de fişiere ajutătoare

4. Explicarea detaliată a defectului şi dacă este posibil prezentarea cauzei şi a modului de remediere

5. Explicarea posibilelor consecinţe ce pot duce la defect blocking

Obiecţiile programatorilor legate de un raport de defect

- Nu este reproductibil
- Nu se înţelege
- Irealist
- Nu este defect – cel mult cerinţă nouă

Raportarea defectelor nereproductibile

- Foarte important de a fi raportate
- Programatorul va ştii unde să caute
- Programatorul ca ştii cu ce să asocieze dacă defectul este descris amănunţit
- Dacă se ataşează salvări de ecrane sau log-uri, programatorul va estima porţiunea de cod care generează defectul
- Verificarea sistemului de raportare dacă nu conţine alte defecte asemănătoare
- Descrierea amănunţită şi rapidă a condiţiilor în care a survenit defectul. În timp se pot uita paşii urmaţi

Raportarea neinteligibilă a defectelor

- Programatorul nu reuşeşte să refacă defectul
- Programatorul va renunţa să lucreze la rezolvarea defectului
- Paşii necesari reproducerii defectului sunt complecşi şi ciudaţi
- Nu este furnizată suficientă informaţie de reproducere a paşilor
- Programatorul nu înţelege raportul
- Programatorul consideră că este cel mult o cerinţă nouă

Testara componentelor (Unit testing)

In computer programming, unit testing is a software verification and validation method in which a programmer tests if individual units of source code are fit for use.

A software testing methodology in which individual tests (unit tests) are developed for each small part of a program.

In computer programming, a unit test is a procedure used to validate that a particular module of source code is working properly.

The most 'micro' scale of testing; to test particular functions or code modules. Typically done by the programmer and not by testers, as it requires detailed knowledge of the internal program design and code.

Sopul testării pe componente [Farrel-Vinay]

- Restrângerea ariei de testare -> se pot găsi erori manual
- Identificarea şi rezolvarea defectelor înainte de testarea prin integrare
- Măsurarea productivităţii programatorilor legată de componentele pe care le-au dezvoltat

Ce este o componentă ?

Clasă, Funcţie, bucată de cod

Caracteristici

- Nu mai mult de 4 nivele de imbricare
- Complexitatea ciclomatică <= 10 [McCabe] (CC=M-N+1, controlflow graph, ex.)
- Dacă conţine cod comun cu altă componentă -> nu este o componentă
- Este testarea de nivel cel mai jos

Cine face testarea?

Testor vs Programator

(avataje/dezavantaje)

Cum se face testarea?

- Definirea foarte exactă a componentelor
- Testarea dinamică a codului (testare white-box) -> inserare cod afişare mesaje de control în log -> urmărirea logului şi a grafului de control pt. verificare
- „Mock objects” - dezvoltarea de clase ajutătoare (ex. lucrul cu bazele de date)
- „drivers and stubs” - driverele simulează apelul componentelor, iar stub-urile simulează funcţionalitatea unei componente (hard-coded)
- testarea componentelor implică: refactorizare (pt. testabilitate) + design patterns = separarea blocurilor de cod (bussines logic de design)

Mock Objects

- obiecte *ajutătoare*
- simulează comportamentul obiectelor *reale*
- facilitează testarea înainte de implementarea propriu-zisă

(testarea unei componente înainte de implementarea alteia)

- facilitează testarea fără a face legătura cu alte componente

(testarea unei componente fără testare prin integrare cu altă componentă)

- substituie obiectele *reale* prin intermediul interfețelor

Mock Objects - Framework

- jMock
- NMock
- EasyMock

Nmock
[www.nmock.org]

- substituie funcționalitatea prin implementarea interfețelor
- definește așteptările (NUnit(real) vs Nmock(„fals”))
- returnează obiecte = stub
- Detectează defectul mai rapid decât folosirea assert-urilor prin rularea codului
- Mesaje de eroare

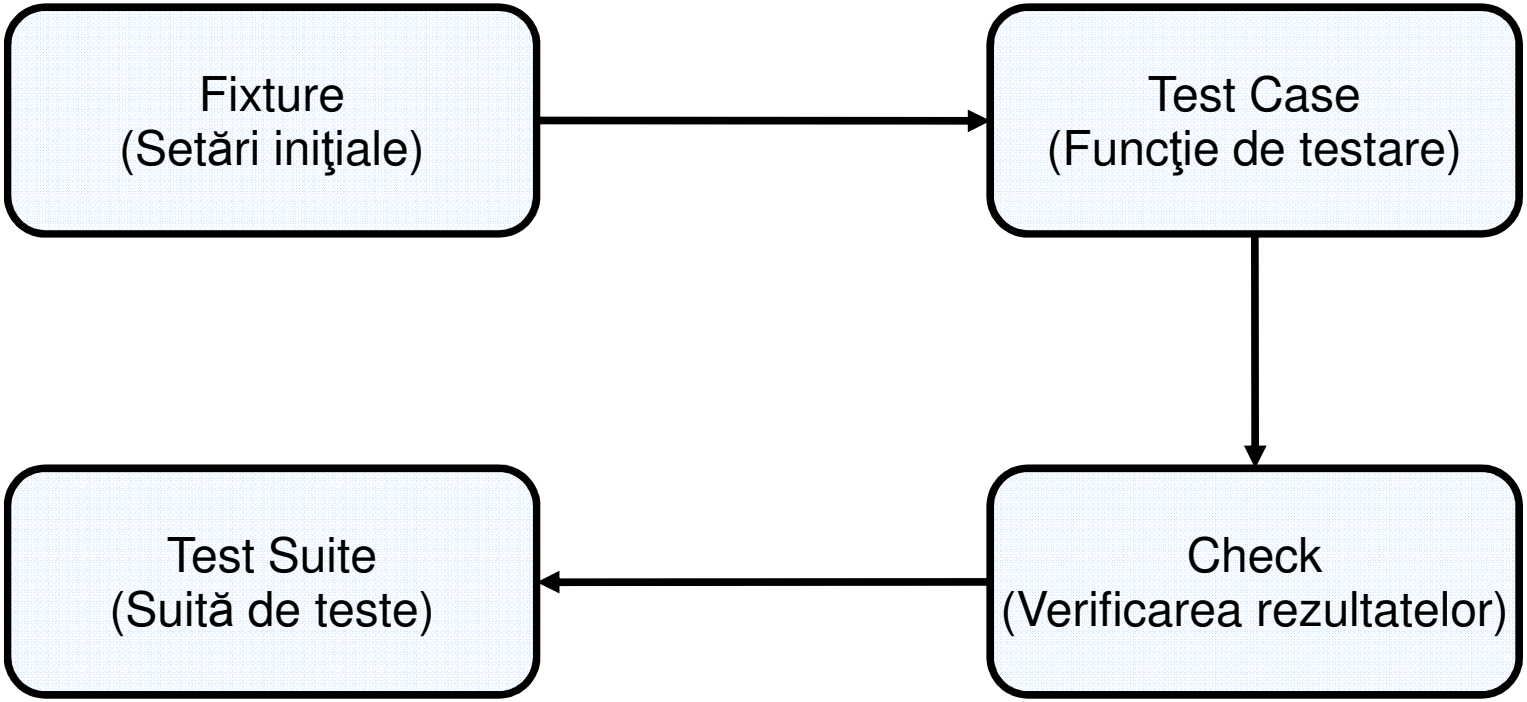
Mock Objects (Test Doubles) [Gerard Meszaros]

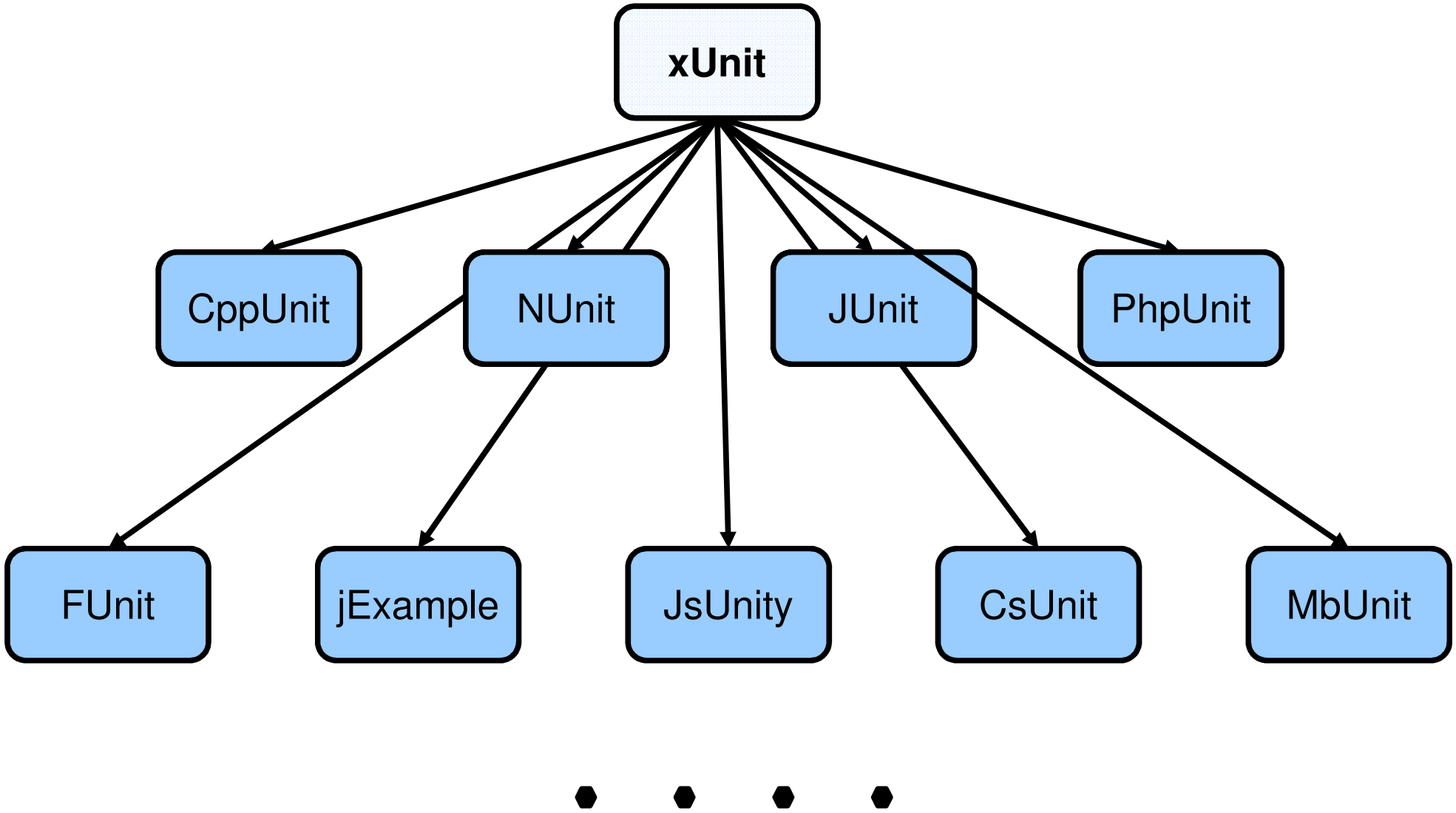
- *Test Doubles <- Stunt Doubles*

Test Doubles Objects:

- *Dummy* – folosite doar ca parametri, dar nu sunt folosite
- *Fake* -
- *Stub* – returnează răspunsuri la apeluri în timpul testării, salveaza informații din timpul rulării
- *Mock* – obiecte **pre-programate** cu așteptări conform specificațiilor

xUnit framework
[Kent Beck]




xUnit
CppUnit
NUnit
JUnit
PhpUnit
FUnit
jExample
JsUnity
CsUnit
MbUnit

NUnit

Caracteristici

- Framework de testare a componentelor
- Nu este un software de testarea automată GUI
- Atribute .NET (adăugare *metadata* în cod, oferă informaţii despre *.NET assembly code*)
- scris în .NET C#, portat din *jUnit*
- Nu permite scrierea de scripturi, doar rularea codului compilat *.dll*

Nunit. Arhitectură

```
namespace NumeNamespace{

    [TestFixture]
    public class NumeClasa{
            //...

        [SetUp]
        public void FunctieInitializare(){
            //...
        }

        [TearDown]
        public void FunctieTerminare(){
            //...
        }

        [Test]
        public void FunctieTestare(){
            //...
        }
    }
}
```

Nunit. Atribute

Atribut: **[Category(NumeCategorie)]** – permite selectarea testelor rulate din NUnit

```
[Test]
[Category("Categoria X")]
```

Atribut: **[Combinatorial]** – generează toate combinaţiile pt. parametri funcţiei de test

```
[Test]
[Combinatorial]
public void FunctieTestare(Values(1,2,3,4)] int x,[Values("A","B","c")] string y)
```

- apelul funcţiei FunctieTestare se va face de 4*3=12 ori

Atribut: **[TestCase(p1,p2,...)]** – atribuie parametri valorile *p1,p2,...*, parametrilor funcţiei de test

```
[TestCase(12,3,4)]
[TestCase(12,2,6)]
[TestCase(12,4,3)]
public void DivideTest(int n, int d, int q)
```

Nunit. Atribute

Atribut: **[Description(descriere)]** – descrierea testului

```
[Test]
[Description("....")]
```

Atribut: **[ExpectedException(typeof(tipExceptie))]** – dacă funcţia de test înregistrează o excepţie *tipExceptie*, atunci testul a trecut cu success

```
[Test]
[ExpectedException( "System.ArgumentException" ) )]
```

Atribut: **[Ignore("mesaj")]** – testul va fi ingnorat cât timp este folosit acest atribut.

Nunit. Atribute

Atribut: **[Setup]** – precedă funcţia de iniţializare a procesului de testare

Atribut: **[Setup]** – precedă funcţia în care se face testarea

Atribut: **[TearDown]** – precedă funcţia de finalizare a procesului de testare

Atribut: **[Timeout (milisecunde]** – dacă funcţia depăşeşte timpul specificat se raportează failure + mesaj

Nunit. Predicate (Assertions)

- Metode statice
- Reprezintă modul de testare Nunit
- Dacă un predicat (assertion) este fals, atunci este raportată o eroare, funcţia ce o conţine ne mai returnând nimic
- Comparaţia între rezultatul obţinut şi cel aşteptat se se realizează prin obiecte de constrangere (constraint object)

Nunit. Predicate (continuare)

Assert.AreEqual(tipdata rezultat_asteptat, tipdata rezultat_obtinut, [string mesaj]);

- tipdata = {int, float, object,...}

Assert.AreSame(object expected, object actual);

- acelaşi obiect este referit de ambii parametri

Assert.Contains(object anObject, IList collection);

- un obiect este conţinut într-o listă sau şir de obiecte

Assert.True(bool condition, string message, object[] parms);

- dacă condiţia este falsă -> Fail + mesaj

Assert.Greater(int arg1, int arg2);

Nunit. Predicate (continuare)

Assert.Pass([[string message], object[] parms]);

Assert.Fail([[string message], object[] parms]);

Assert.Ignore([[string message], object[] parms]);
- test ingnorat la rulare

Assert.Inconclusive([[string message], object[] parms]);
- se consideră ca nu sunt suficiente date

Nunit. Predicate (continuare)

StringAssert.Contains(string expected, string actual);

StringAssert.StartsWith(string expected, string actual);

StringAssert.EndsWith(string expected, string actual);

StringAssert.AreEqualIgnoringCase(string expected, string actual);

StringAssert.IsMatch(string regexPattern, string actual);

Nunit. Exemplu

- adaugare balanta minima
- adaugare noua functie de testare
- adaugare exceptie
- adaugare categorie
- adaugare testcase
- adaugare setup

Testarea Domeniului de valori

“Dacă într-o căsuţă de text se poate introduce valoarea 0, atunci se va găsi cineva care să împartă un număr la acea valoare” [Murphy]

The essence of domain testing is that we partition a domain into subdomains (equivalence classes) and then select representatives of each subdomain for our tests. [Kaner]

Domain testing = equivalence partitioning, boundary analysis,
category partitioning

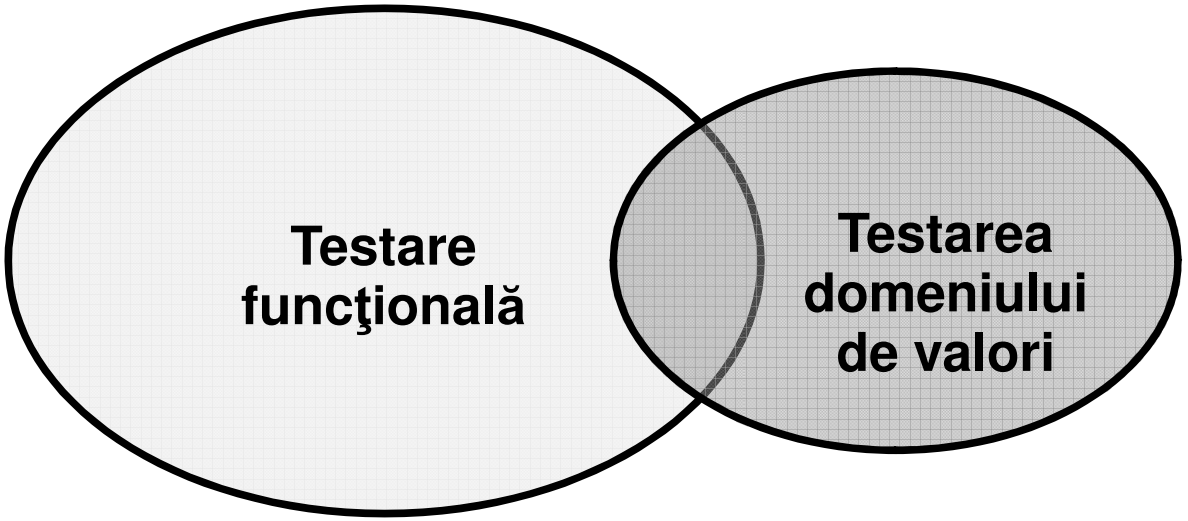
“Equivalence partitioning is … intuitively used by virtually every tester we’ve ever met.” [Richard Craig]

Introducere

Arta testării domeniului de valori:

1. Partiţionarea domeniului de valori

2. Alegerea celor mai importante valori pentru testare



$f : A \rightarrow B$

A – *d*omeniu
B – codomeniu

Ce testam?

Testarea domeniului de valori este un fel de testare funcţională [Kaner]

Clase de echivalenţă

All elements within an equivalence class are essentially the same for the purpose of testing. [Kaner]

- Echivalenţă intuitivă – valori diferite produc acelaşi rezultat/ rezultate asemănătoare – nu există regulă de împărţire – adaptabilitate în funcţie de cerinţe

- Echivalenţă definită prin specificaţii – rareori utilizată – de obicei este sarcina testorului să aleagă clasele de echivalenţă, nu a celor care scriu specificaţii

- Echivalenţă prin analiza variabilelor – white box

- Echivalenţă subiectivă – doi testori vor realiza probabil clase de echivalenţă diferite

- Echivalenţă bazată pe risc – valorile sunt grupate în funcţie de asemănarea în ceea ce priveşte tipul de defect ce s-ar putea produce (s-ar putea sa nu existe limite ale domeniului, doar riscuri!). (ex. Problema triungiului).

Problema triunghiului

Problemă: *Se citesc de la tastatură trei numere, care se presupun a reprezenta laturile unui triunghi. Să se verifice dacă triungiul este scalen, isoscel, echilateral sau nu este triunghi*

- Clasa 1 : numărul prea mic sau prea mare de laturi
- Clasa 2 : 0..9 -> 48..57 (ASCII) – 47 ('/') şi 58 (':') - limite
- Clasa 3: a=b=c
- Clasa 4: a=b, b<>c, a<>c
- Clasa 5: a<>b, b<>c, a<>c
- Clasa 6: a+b>c, a+c>b, c+b>a,

Clase de echivalenţă. Alegerea reprezentanţilor

A best representative of an equivalence class is a value that is at least as likely as any other value in the class to expose an error. [Kaner]

- nu neapărat orice valoare din aceeaşi clasă poate produce un defect
- de obicei se testează valorile limită (ex. 0..99, se testează -1, 0, 99, 100) – valabil pentru spaţiile ordonate
- pentru spaţii neordonate se aleg alte criterii de testare la limitelor (ex. testarea tipăririi la sute de imprimante – se aleg acele imprimante cu memorie, viteză, costuri mici/mari)

Clase de echivalenţă. Combinarea reprezentanţilor

- nu există o regulă anume
- se reduce considerabil timpul/testele
- Ex. 3 variabile de 2, 3, 4 cifre = 90 * 900 * 9000 = 729 * 10^6

Împărţirea în clase de echivalenţă:

-> (9, 10, 34, 99, 100) X (99, 100, 546, 999, 1000) X
(999, 1000, 8635, 9999, 10000)

Greşeli ale testorilor în testarea domeniului de valori

- generalizare slabă – alegerea neadecvată a valorilor
- Lipsa unor valori importante – lipsa manipulării erorilor
- Prea multe valori – se testează inutil valori (testarea valorilor limită 97, 98, 99, 100, pentru un număr de 2 cifre)
- Nedetectarea unei limite
- Nedetectarea dimensiunii – de exemplu dimensiunea unui câmp de text în care se introduc numere de 1,2,3,4 cifre
- Neformularea riscului – orice test porneste de la un risc pe care testorul şi-l imaginează
- Neexplicarea riscului – motivarea legăturii între cazul de test şi risc

Primii paşi în testarea domeniului de valori

1. Primele teste sunt simple şi evidente – dacă rezultă defect, s-a detectat o problemă gravă -> reproiectare sau rezolvarea timpurie evitând efectele secundare ce pot apărea. Modalitate de învăţare a programului, a funcţionalităţii

2. testarea simpatetică – raportarea defectelor se face cunoscând produsul, iar implicaţiile pe care le pot aduce defectele sunt descrise pentru a fi înţelese de programator

3. testarea succintă a tuturor modulelor – înainte de testarea amănunţită

4. alegerea testelor mai puternice – împărţirea valorilor în clase de echivalenţă şi testarea valorilor limită

5. imaginarea posibilelor defecte bazate pe riscul pe care acestea ar putea să îl implice

6. testarea subiectivă în funcţie de presupunerile testorului

Problemă [Kaner]

Specificaţiile problemei:

- *se cere adunarea a două numere*
- *după introducerea fiecărui număr se va apăsa tasta Enter*
- *fiecare număr este format din două cifre*
- *suma va fi afişată*
- *lansarea programului în execuţie - de la linia de comandă*

Specificaţii complete?

Problemă [Kaner]. Paşi

1. Introducerea câtorva perechi de numere şi verificarea rezultatului. Observarea interfeţei cu utilizatorul

2. Împărţirea în clase de echivalenţă şi analiza limitelor

199 de valori pentru fiecare număr:
1..99
0
-99..-1
-> 199 * 199 = 39.601 combinaţii (2 numere)

Întrebare: **Are rost să testăm: 5+9, 5+10, 6+11, 7+4, 84+41, 39+92?**

Răspuns: **E suficient un singur test.**

Întrebare: **Există o situaţie în care doar una din sumele de mai sus poate duce la defect, iar celelalte nu ?**

Problemă [Kaner]. Paşi

2. Împărţirea în clase de echivalenţă şi analiza limitelor (continuare)

Întrebare:**Testăm toate sumele posibile?**

Răspuns: **Este posibil, dar nu are rost. Alegem câteva teste reprezentative.**

Întrebare:**Testăm 100+100? Ce reprezintă aceste valori?**

Răspuns: **Da. Valori limită.**

Întrebare: **De ce testăm 100+100, dacă în specificaţii este scris că numerele au două cifre?**

Răspuns: **Pentru a verifica dacă programatorul a prevăzut programul cu astfel de condiţii**

Problemă [Kaner]. Paşi

2. Împărţirea în clase de echivalenţă şi analiza limitelor (continuare)

- două teste sunt echivalente dacă sunt asemănătoare şi testorul consideră că nu are rost să ruleze ambele teste
- pentru a reduce numărul de teste rulate domeniul se împarte în clase de echivalenţă – o clasă de echivalenţă este formată din teste echivalente
- din fiecare clasă de echivalenţă se alege un reprezentant – test reprezentant
- dacă spaţiul intrărilor poate fi mapat pe axa numerelor, atunci clasele de echivalenţă vor fi delimitate de către valorile limită [Myers]
- clasele de echivalenţă şi valorile limită sunt stabilite de către testor în funcţie de experienţă şi intuiţie

Problemă [Kaner]. Tabelul cu valori limită

Variabilă	Clasă de echivalenţă validă	Clasă de echivalenţă invalidă	Valori limită şi cazuri speciale	Descriere
Primul număr	-99..99	< -99 > 99	-100, -99 99, 100	
Al doilea număr	-99..99	< -99 > 99	-100, -99 99, 100	

Problemă [Kaner]. Tabelul cu valori limită (continuare)

Variabilă	Clasă de echivalenţă validă	Clasă de echivalenţă invalidă	Valori limită şi cazuri speciale	Descriere
Primul număr	-99..99	< -99 > 99	-100, -99 99, 100	
Al doilea număr	-99..99	< -99 > 99	-100, -99 99, 100	
Suma	-198..198	< -198 > 198	-200, -199, -198, -1, 0, 1, 198, 199, 200	

Problemă [Kaner]. Tabelul cu valori limită (continuare)

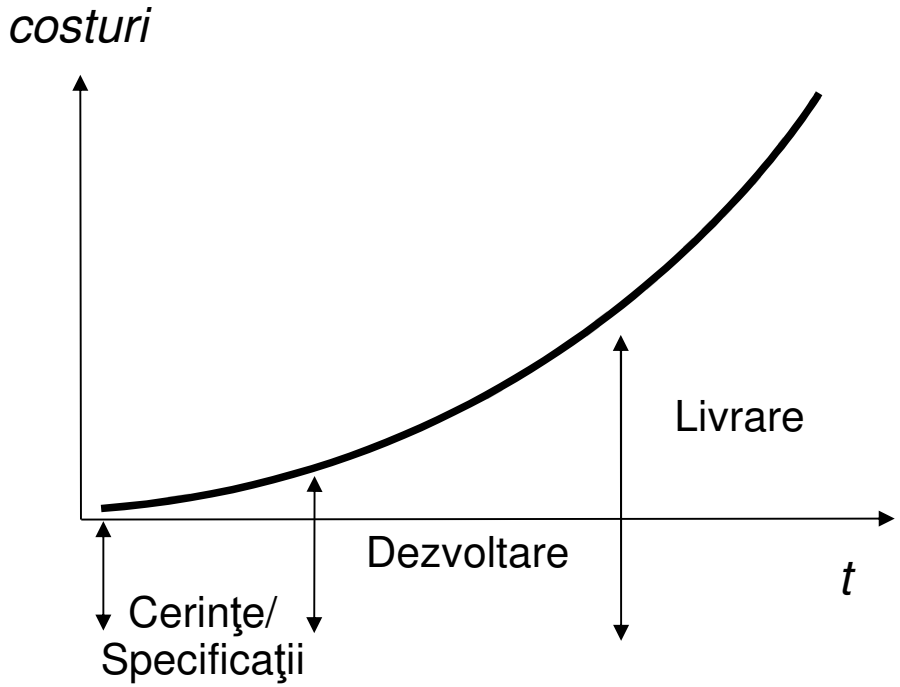
Variabilă	Risc	Clasă de echivalenţă – nu ar trebui să producă defect	Clasă de echivalenţă – s-ar putea să producă defect	Reprezentanţi	Descriere
Primul număr	Depăşirea limitelor (out of range)	-99..99	< -99 > 99	-100, 100	
	Nu face deosebire între (in-range şi out-of range)			-100, -99, 99, 100	
	Clasificare incorectă a cifrelor	Caractere diferite de cifre	0..9	0 (48 ASCII) 9 (57 ASCII)	
	Clasificare incorectă a non-cifrelor	0..9	Caractere diferite de cifre	/ (47 ASCII) ; (58 ASCII)	

Clase de echivalenţă. Analiza limitelor. Exemple

- Limite numerice
- Caractere acceptate/neacceptate
- Numărul maxin de înregistrari dintr-o BD
- Dimensiunea rezultatelor expresiilor matematice (număr de biţi sau de cifre)
- Dimensiunea unui fişier
- Dimensiunea unui număr sau a unui şir de caractere

Costurile pe care le implică detecţia şi rezolvărea defectelor

- costurile cresc odată cu evoluţia în timp a dezvoltării proiectului
- repararea unui defect -> (%) poate produce un alt defect
- contractele între client şi dezvoltator – penalizări legate de livrare
- cu cât sunt detectate/reparate mai multe defecte în perioada critică de livrare, cu atât cresc şansele ca livrarea să fie amânată -> costuri de penalizare
- un defect detectat din timp reduce costurile proiectului de X ori faţă de un defect detectat târziu



Reducerea costurilor

- se caută defecte în cerinţe/specificaţii – cel mai ieftin la momentul proiectării (costuri foarte mici – se evită propagarea erorii)

- detectarea unui defect de către programator înainte testor – reduce costurile considerabil (timp, bani, persoane implicate)

- defecte înainte de livrare influenţează -> divizia de marketing (nu se pot realiza la timp pliante, demo-uri), divizia de documentare (documentaţie, tutoriale), divizia de vânzâri

- după livrare costurile pot creşte exponenţial, dacă nu este specificat altfel în contract. Produsul software este actualizat (client unic vs. milioane de clienţi, e-mail/web vs. mail)

Extreme programming – soluţie de reducere a costurilor ?

- reduce considerabil costurile de rezolvare a unui defect înainte de livrare

- procesul de dezvoltare a proiectului – divizată în cicluri de scurtă durată (programare, testare, verificare client, proiectare) – interacţiune continuă cu clientul

- metodă de dezvoltare software Agile

Costuri legate de calitatea produsului

4 categorii de costuri

- prevenire
- verificare
- defecte interne
- defecte externe

Costuri de prevenire. Exemple

- şcolarizarea celor implicaţi în proiect
- analiza cerinţelor
- proiectare tolerantă la defecte
- claritatea specificaţiilor
- documentarea internă a proiectului
- evaluarea anticipată a fiabilităţii utilitarelor necesare în dezvoltarea produsului

Costuri de verificare. Exemple

- revizuirea proiectării
- inspectarea codului
- testare white box & black box
- şcolarizarea testorilor
- testare beta
- testarea funcţionalităţii din punct de vedere al utilizatorilor propriu-zişi
- testarea produsului de către client, înainte de livrare

Costul defectelor interne. Exemple

- Rezolvarea defectelor
- Testarea prin regresie
- Timp pierdut în interiorul companiei (programator, testor)
- Timp pierdut în marketing şi vânzări
- Timp pierdut în publicitate
- Penalizări legate de livrarea întârziată

Costul defectelor externe. Exemple

- Pierderea bunăvoinţei clientului
- Telefoane pentru suport tehnic
- Scrierea documentaţiei Questions & Answers
- Investigarea problemelor raportate de către client
- Oferirea suportului pentru mai multe versiuni
- Garanţie, restituirea anumitor sume către client
- Livrarea produselor actualizate

Motivarea necesităţii rezolvării defectului alături de părţile interesate pe baza costurilor estimate

Părţi interesate

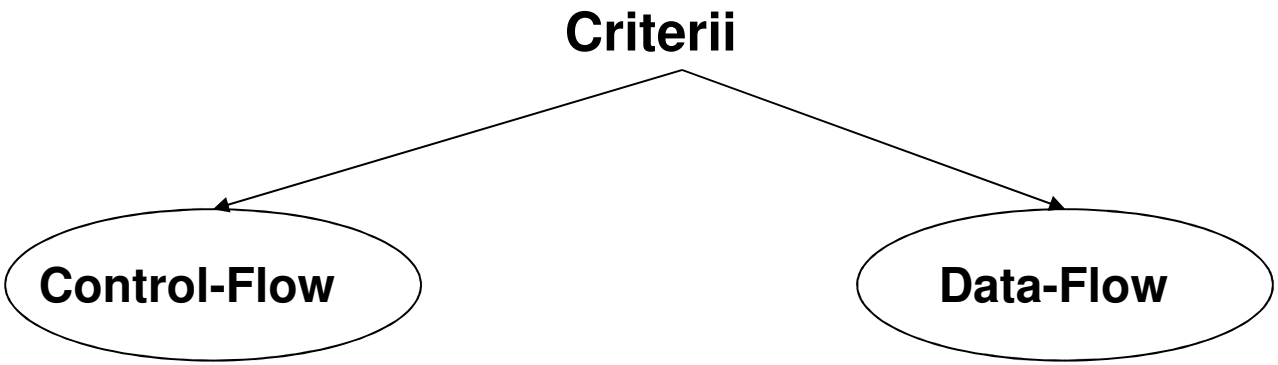
- Echipa de documentare
- Echipa de suport tehnic
- Echipa de marketing
- Echipa de vânzări
- Managerii de proiect

Testare/Analiză/Verificare statică

- îmbunătăţeşte calitatea codului
- defecte software (implementare, editare)
- verifică codul pe anumite şabloane standardizate la nivelul organizaţiei
- Testare statică vs. Testare dinamică
- nu se rulează software-ul -> se verfică bucăţi de cod
- produce avertismente înainte de compilare

Testare/Analiză/Verificare statică

- Nu execută codul sursă
- Metode manuale (la inceput, costuri reduse) & automate - complementare
- Detecţia blocurilor executate în timpul testării
- Examinează codul



Criteriul Control-Flow

- Reprezentarea grafului Control-Flow
  - noduri = operaţii/condiții executate secvenţial
  - muchii = fluxul de control dintre operaţii

- Scopul – acoperirea grafului C-F, prin reducerea cazurilor de test

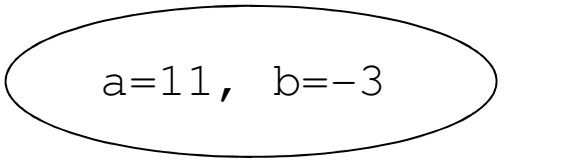
- Tipuri de acoperire:
  - acoperirea operaţiilor executabile
  - acoperirea ramurilor
  - acoperirea deciziilor și a condiţiilor -> independenţa subexpresiilor
  - acoperirea rutelor de execuție -> posibilele rute activate în execuţia codului sursă

- Ex: condiţii, cicluri, ieşire din condiţie/ciclu,...

- Dezavantaje:
  - necesitatea înțelegerii codului de către testor pt. producerea grafului C-F
  - multe linii de cod (module/unități) -> numărul mare de cazuri de test

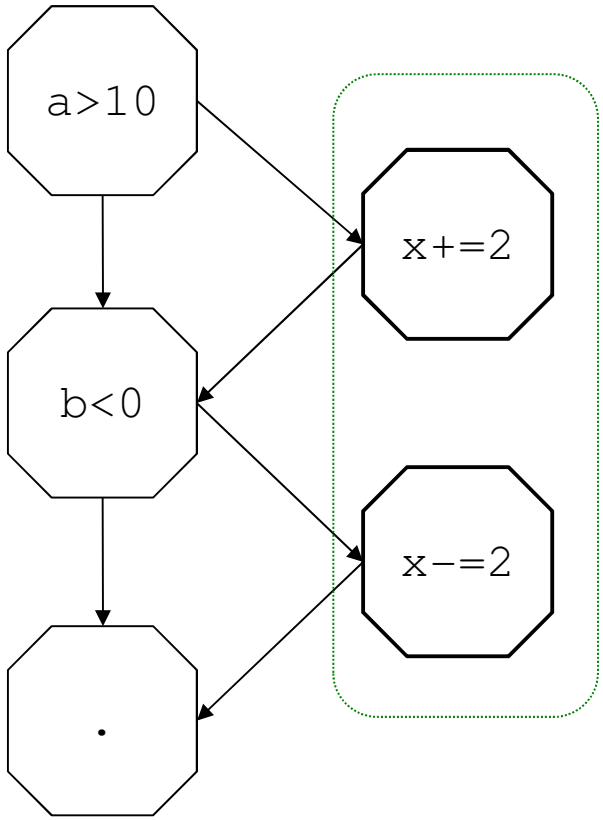
Criteriul Control-Flow. Acoperirea 100% a operațiilor executabile

```
if (a>10) x+=2;
if (b<0) x-=2;
```



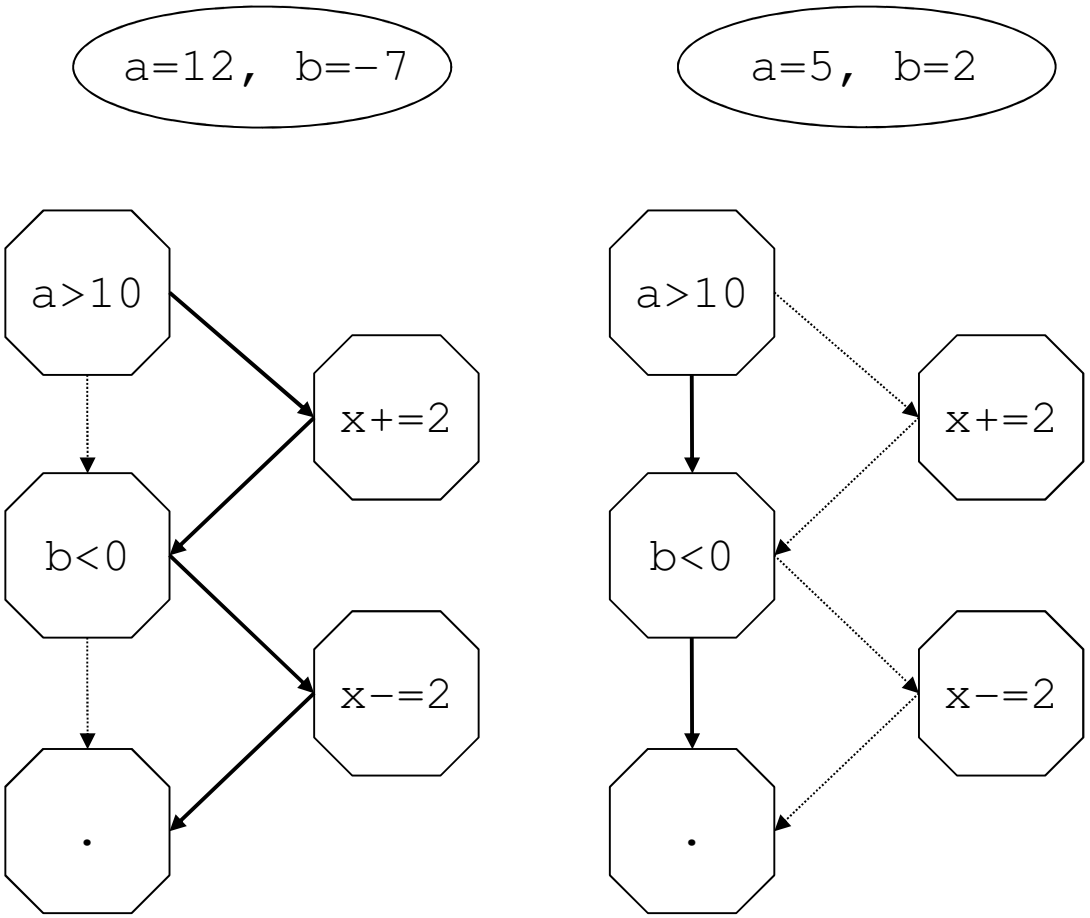
Acoperire completă a operațiilor executabile

Acoperire incompletă a rutelor de execuție. Comportamentul codului poate fi diferit pentru alte perechi (a,b)



Criteriul Control-Flow. Acoperirea 100% a deciziilor

```
if (a>10) x+=2;
if (b<0) x-=2;
```

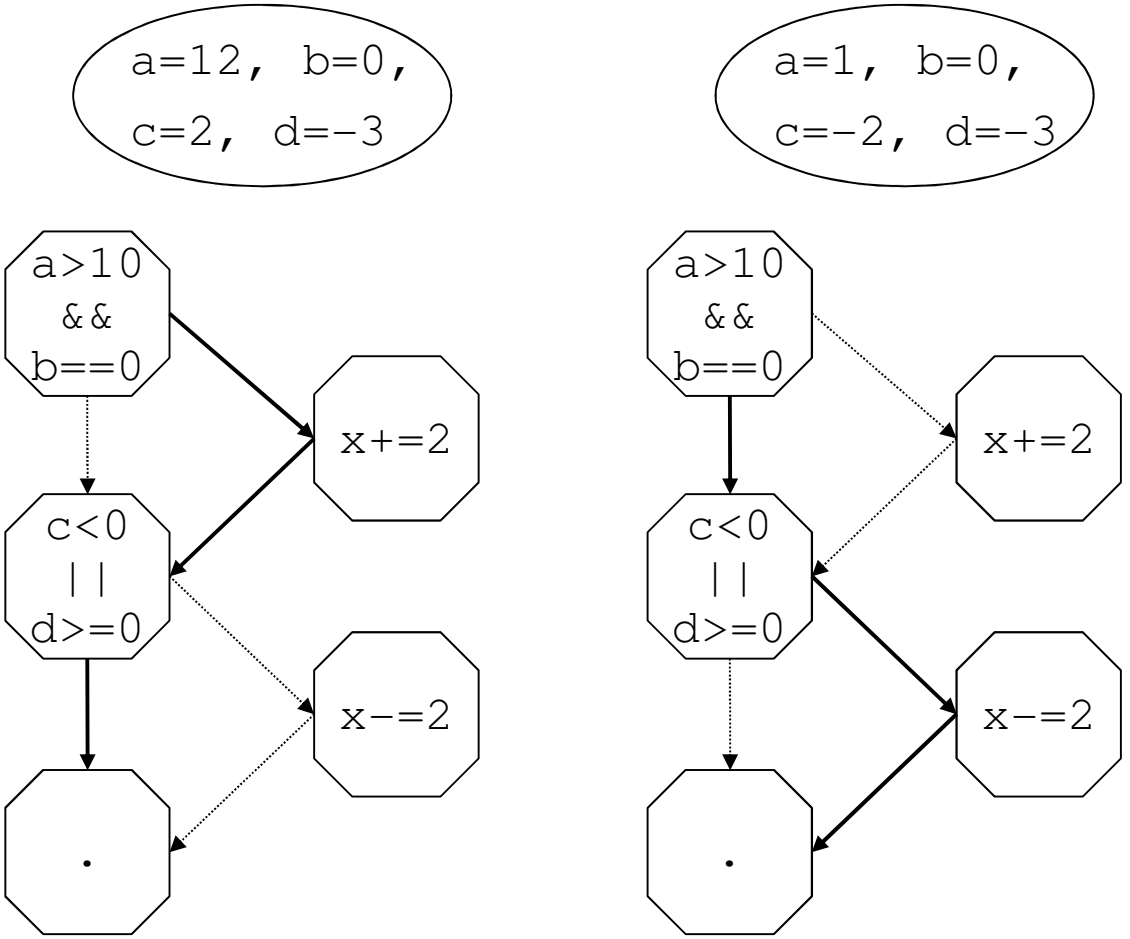


Acoperire incompletă a rutelor de execuție.

Din 4 rute au fost acoperite 2.

Criteriul Control-Flow. Acoperirea 100% a condițiilor

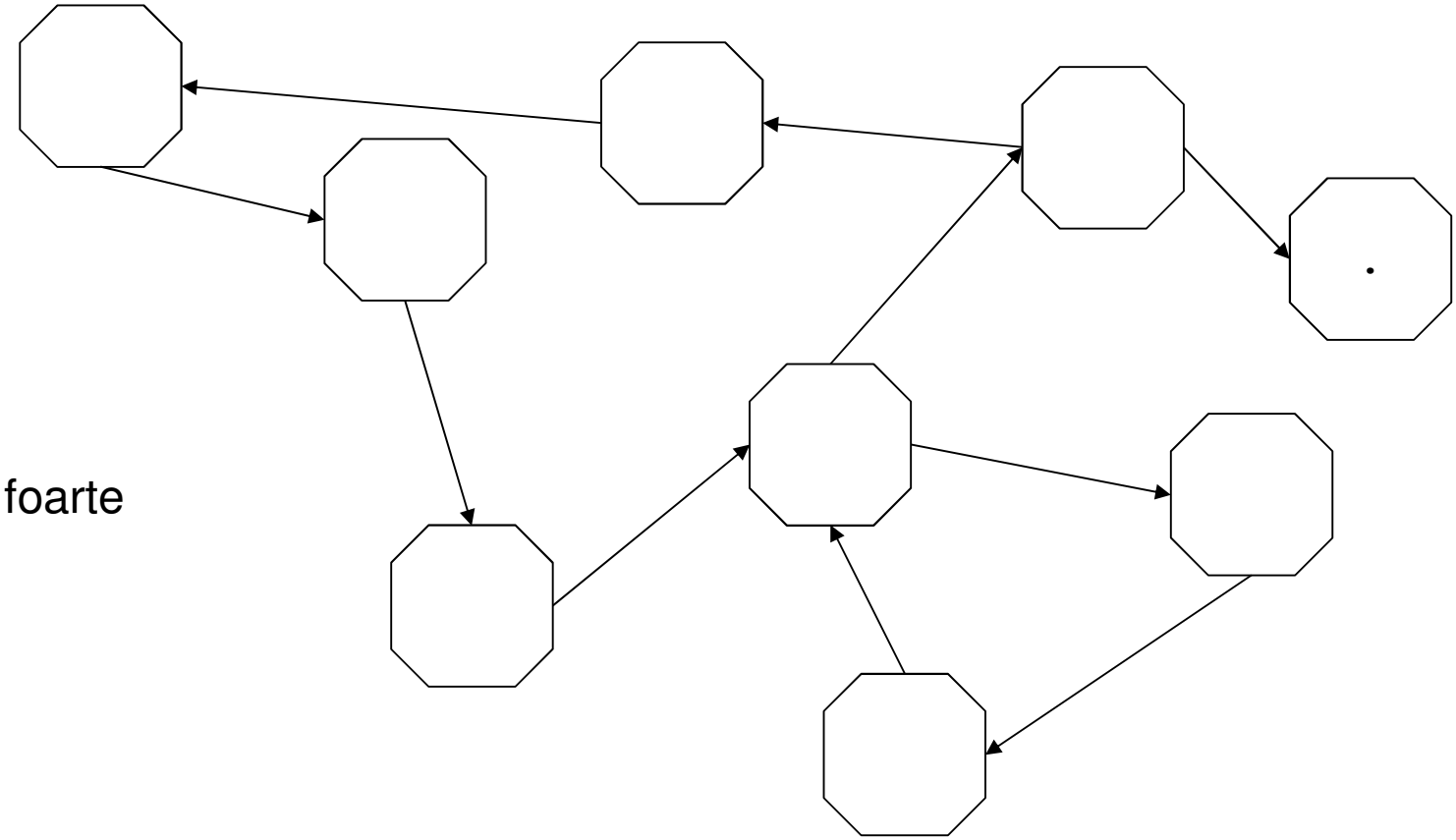
```
if (a>10 && b==0) x+=2;
if (c<0 && d>=0) x-=2;
```



Acoperire incompletă a rutelor de execuție.

Din 4 rute au fost acoperite 2.

Criteriul Control-Flow. Acoperirea 100% a rutelor de execuție

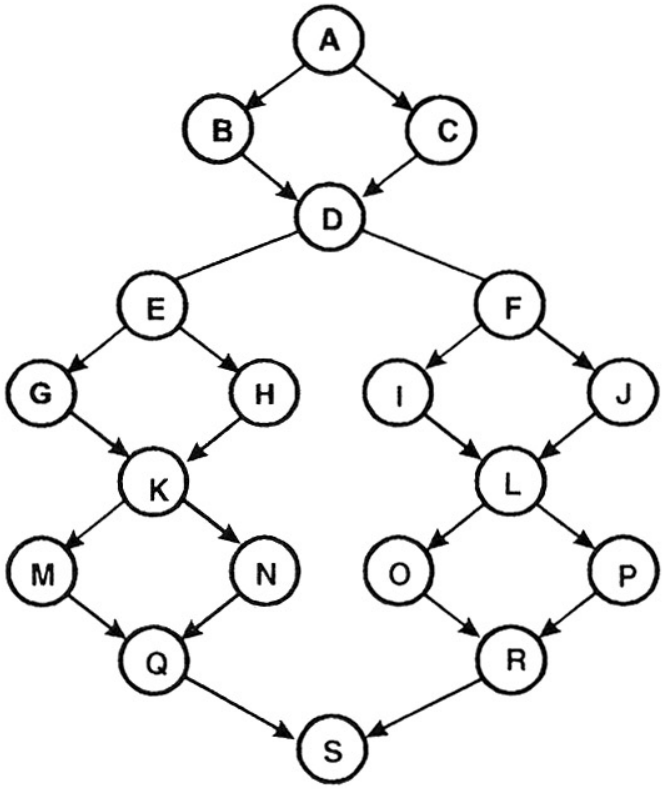


Cicluri – număr foarte mare de teste

Criteriul Control-Flow. Testarea structurată. Complexitate ciclomatică [Tom McCabe]

- obținerea grafului Control-Flow din codul software
- calcularea Complexității Ciclomatice (C)
- selectarea unei mulțimi de rute de bază (început/sfârșit)
- crearea unui caz de test pentru fiecare rută de bază
- executarea testelor

Complexitate ciclomatică (C)



C = muchii – noduri + 2

sau

C = p + 1 / p – numărul de decizii binare (valabil pt. tipuri de decizii exclusiv binare)

C = 24 - 19 + 2 = 7

sau

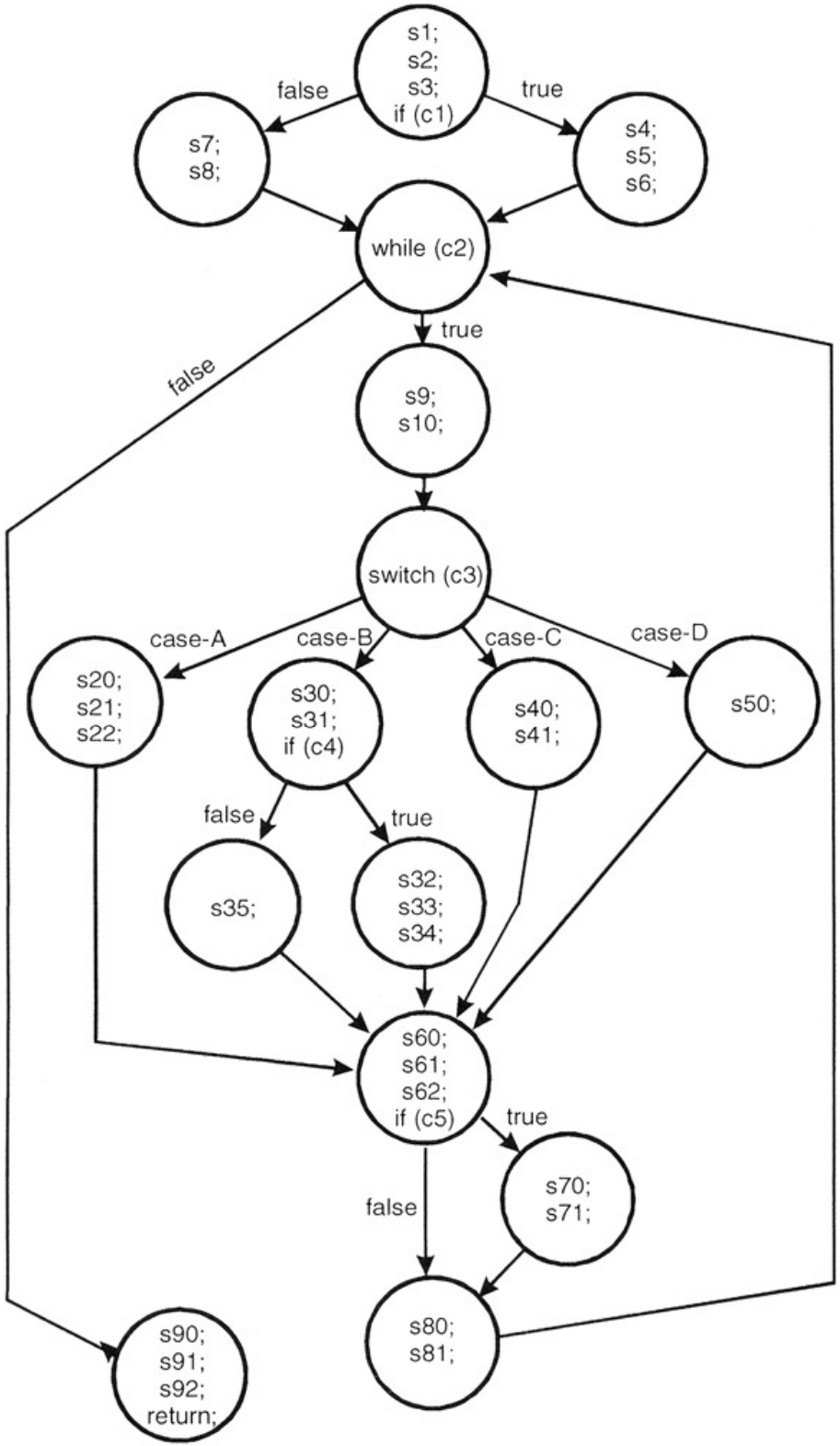
C = 6 + 1 = 7

C = numărul minim de rute de bază independente neciclice, care împreună traversează toate muchiile, având proprietatea că oricare două rute au cel puțin o muchie diferită. Garantează 100% acoperirea operațiilor executabile și a condițiilor

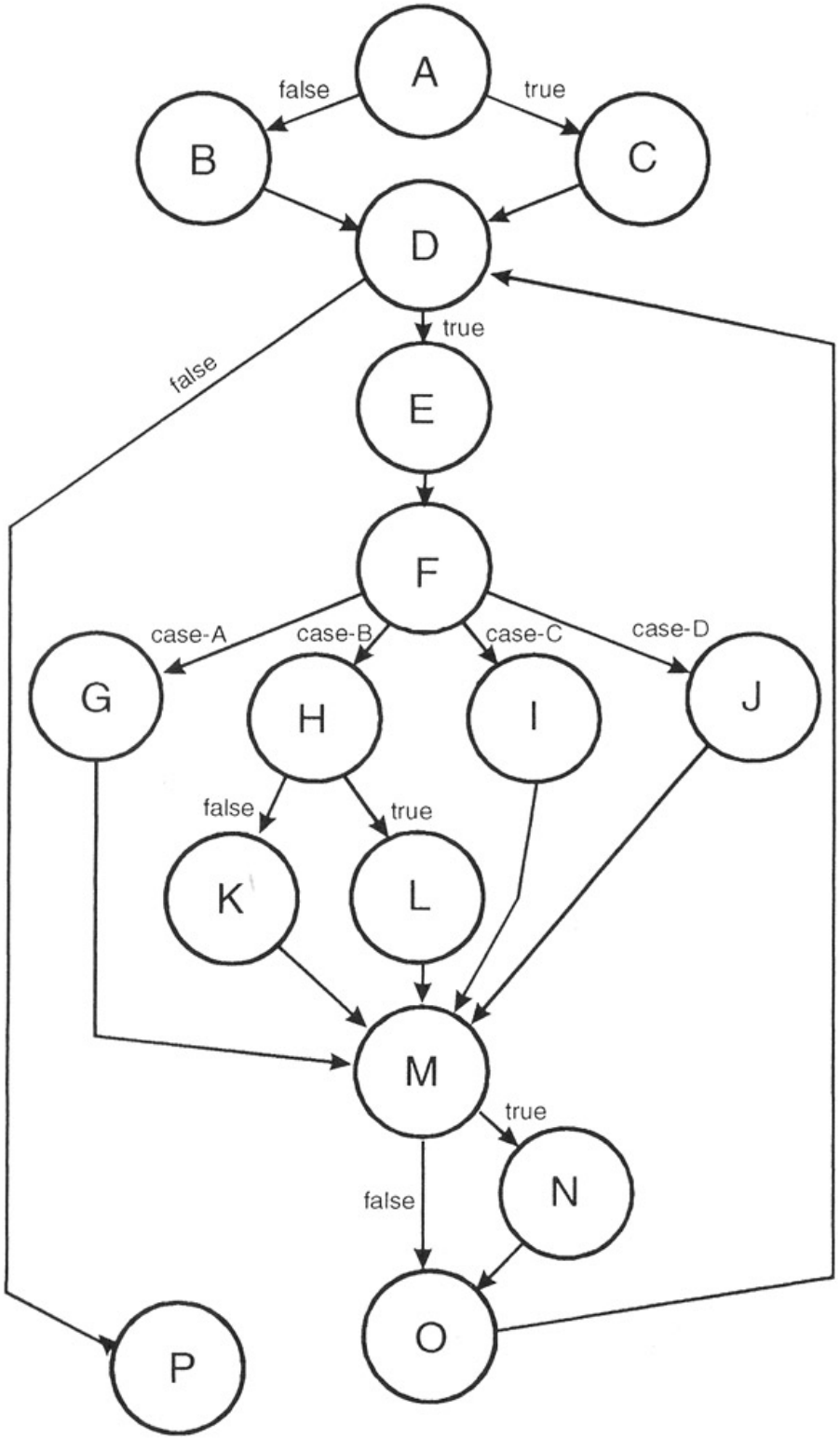
Complexitate ciclomatică (C). Exemplu [Brown & Donaldson]

```
boolean evaluateBuySell (TickerSymbol ts) {
      s1; s2; s3;
      if (c1) {s4; s5; s6;}
      else {s7; s8;}
      while (c2) {
            s9; s10;
            switch (c3) {
                  case-A: s20; s21; s22; break; // End of Case-A
                  case-B: s30; s31;
                        if (c4) { s32; s33; s34; }
                        else { s35; } break; // End of Case-B
                  case-C: s40; s41; break; // End of Case-C
                  case-D: s50; break; // End of Case-D
            } // End Switch
            s60; s61; s62;
            if (c5) {s70; s71; }
            s80; s81;
      } // End While
      s90; s91; s92;
      return result;
}
```

si = operația i
cj = condiția j


s1;
s2;
s3;
if (c1)
false
true
s7;
s8;
s4;
s5;
s6;
while (c2)
false
true
s9;
s10;
switch (c3)
case-A
case-B
case-C
case-D
s20;
s21;
s22;
s30;
s31;
if (c4)
s40;
s41;
s50;
false
true
s35;
s32;
s33;
s34;
s60;
s61;
s62;
if (c5)
true
s70;
s71;
false
s80;
s81;
s90;
s91;
s92;
return;

C = 22 – 16 + 2 = 8



Variantă 8 rute de bază

1. ABDP
2. ACDP
3. ABDEFGMODP
4. ABDEFHKMODP
5. ABDEFIMODP
6. ABDEFJMODP
7. ABDEFHLMODP
8. ABDEFIMNODP

Cazuri de test / condiții

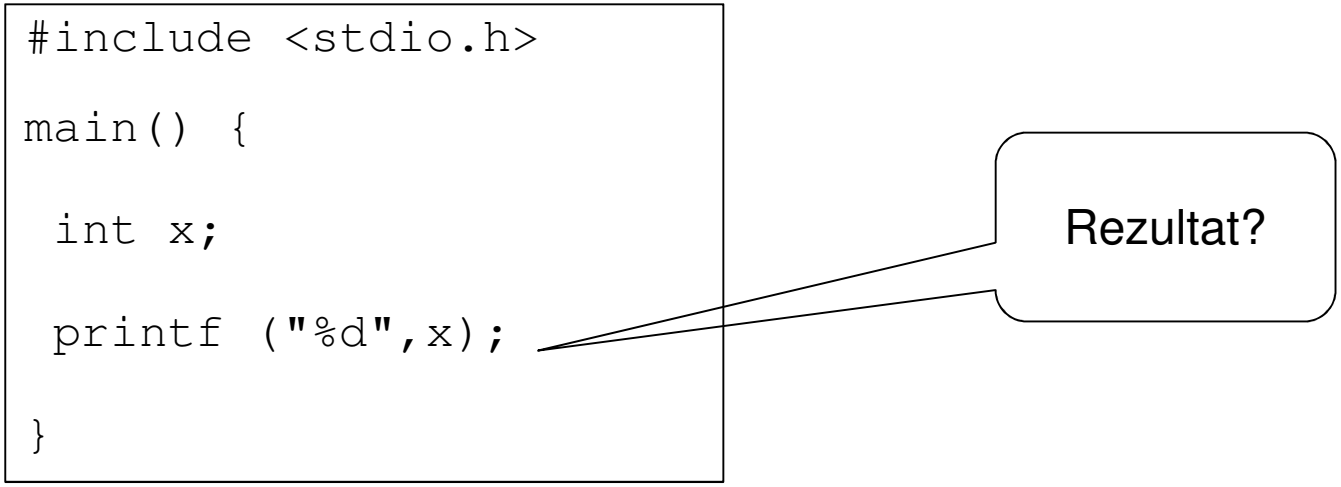
Test Case	C1	C2	C3	C4	C5
1	False	False	N/A	N/A	N/A
2	True	False	N/A	N/A	N/A
3	False	True	A	N/A	False
4	False	True	B	False	False
5	False	True	C	N/A	False
6	False	True	D	N/A	False
7	False	True	B	True	False
8	False	True	C	N/A	True

Complexitate ciclomatică (C). Temă

```
...
     if (c1) {
          while (c2) {
               if (c3) { s1; s2;
                    if (c5) s5;
                    else s6;
                    Break;} // → End while
               else
               if (c4) { }
               else { s3; s4; break;}
          } // End while
     } // End if
     s7;
     if (c6) s8; s9;
     s10;
...
```

```
si = operația i
cj = condiția j
```

Criteriul Data-Flow

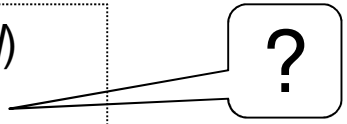


Testarea Data-Flow = detectează utilizarea eronată a variabilelor în codul sursă

Criteriul Data-Flow

Posibilităţi de primă apariţie a unei variabile în cod

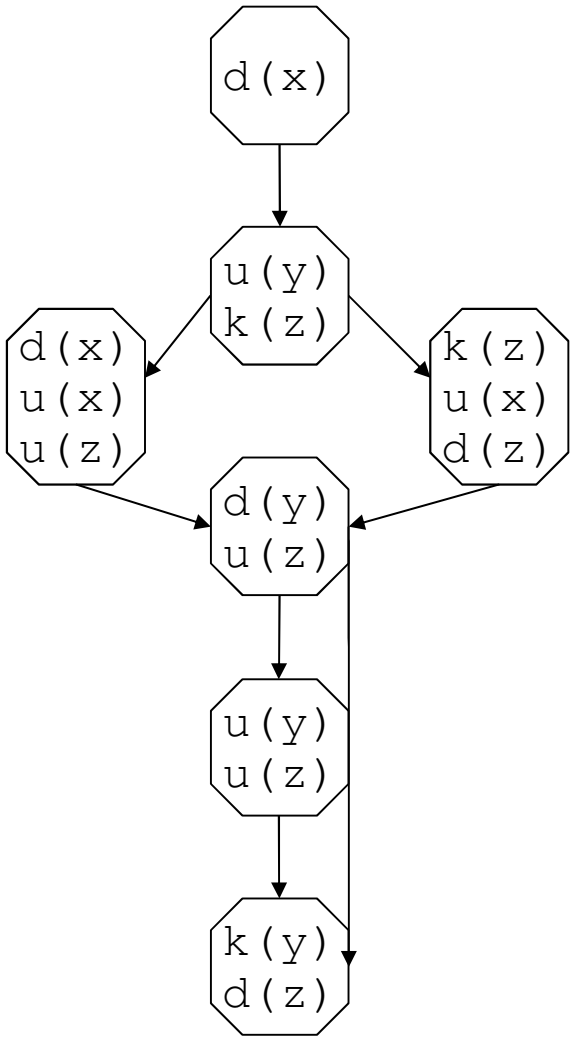
1. ~*d* variabila nu există (notat prin ~), apoi este definită (*d*)
2. ~*u* variabila nu există, apoi este utilizată (*u*)
3. ~*k* variabila nu există, apoi este distrusă (*k*)



Perechi (d,u,k)

dd – definire succesivă -> posibil eroare de codare
du – definire apoi utilizare -> corect
dk – definire apoi distrugere -> posibil eroare de codare
ud – utilizare apoi (re)definire -> ok
uu – utilizare succesivă -> ok
uk – utilizare apoi distrugere -> ok
kd – distrugere apoi (re)definire -> ok
ku – distrugere apoi utilizare -> eroare majoră
kk – distrugere succesivă -> posibil eroare de codare

Criteriul Data-Flow (Static)



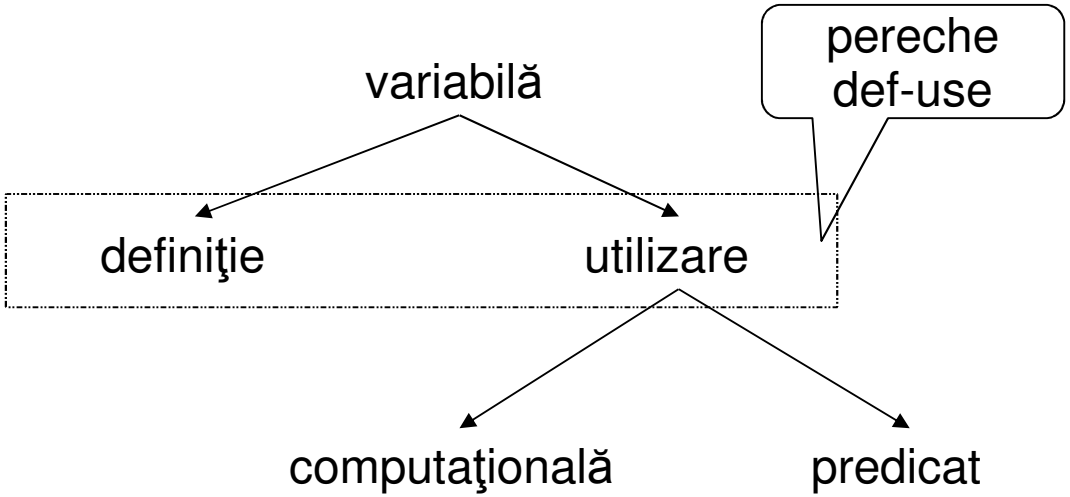
Variabila **x**	Variabila **y**	Variabila **z**
~d – corect *dd – eroare* *du – ok*	*~u* – eroare! *ud – ok* *du – ok* *uk - ok*	*~k* – eroare *ku – eroare!* *uu – ok*
		ud – ok
du – ok	*dk* – eroare	*kk* – eroare

Cazuri particulare:

- utilizarea vectorilor (a[j] – depinde de context)
- anumite rute ce conțin perechi eronate (d,u,k) nu sunt folosite niciodată -> cod nu neaparat incorect

==> Criteriul Data-Flow (Dinamic)

Criteriul Data-Flow (Dinamic)



- Se bazează pe Control-Flow
- **Pentru fiecare variabilă se definește cel puțin un caz de test pentru fiecare pereche „def-use”**

Criteriul Data-Flow. Temă

Creați cazuri de test pe baza criteriului Data-Flow pentru fiecare variabilă din funcția factorial de mai jos. Un caz de test poate acoperi mai multe variabile.

```
int factorial (int n) {
        int answer, counter;
        answer = 1;
        counter = 1;

loop:
        if (counter > n) return answer;
        answer = answer * counter;
        counter = counter + 1;
        goto loop;
}
```

Criteriul Data-Flow. Temă

Creați cazuri de test Data-Flow, pe baza căilor Control Flow:

```
int module( int selector) {
int foo, bar;
switch selector {
case SELECT-1:
      foo = calc_foo_method_1();
      break;
case SELECT-2:
      foo = calc_foo_method_2();
      break;
case SELECT-3:
      foo = calc_foo_method_3();
      break;
      }
switch foo {
case FOO-1:
      bar = calc_bar_method_1();
      break;
case FOO-2:
      bar = calc_bar_method_2();
      break;
      }
return foo/bar;
}
```

Testare statică. Analiză statică

- FindBugs, Checkstyle
- Splint, Frama-C, BLAST
- CppCheck
- Perl::Critic

Exemple de analiză statică

- Recursivitate infinită
- O singură instrucţiune return la nivel de funcţie – depanare uşoară
- Greşeli de editare (=, ==) - typos
- Comparaţii între obiecte diferite
- Evitarea erorii NullPointerException
- Cod redundant

Exemple de analiză statică

Neiniţializarea variabilei

```
int f( bool b )
{
   int i;
   if ( b )
   {
      i = 0;
   }
   return i;    // i este neiniţializată dacă b =
}               false
```

Exemple de analiză statică

Dereferenţierea pointerului NULL

```
#include <malloc.h>

void f( )
{
  char *p = ( char * ) malloc( 10 );
  *p = '\0';

  //  ...
 free( p );
}
```

```
#include <malloc.h>
void f( )
{
  char *p = ( char * )malloc ( 10 );
  if ( p )
  {
    *p = '\0';

    //  ...
    free( p );
  }
}
```

Exemple de analiză statică

Ingnorarea valorii returnate

```
#include <stdio.h>

void f( )
{

  fopen( "test.c", "r" ); // valoarea
returnată ignorată

  // ...
}
```

```
#include <stdio.h>

void f( )
{
  FILE *stream;
  if((stream = fopen( "test.c", "r" )) == NULL )
    return;

  // ...
}
```

Exemple de analiză statică

Lipsă argument

```
#include <string.h>
void f( )
{

  char buff[15];
  sprintf(buff, "%s %s", "Hello, World!");
}
```

```
#include <string.h>
void f( )
{
  char buff[15];
  sprintf(buff, "%s %s ", "Hello","World");
}
```

Exemple de analiză statică

Indice depăşit

```
int buff[14]; // array de 0..13 elemente
void f()
{
   for (int i=0; i<=14;i++)
   {
     buff[i]= 0;

     // ...
   }
}
```

```
int buff[14]; // array of 0..13 elements
void f()
{
   for ( int i=0; i < 14; i++)
   {
     buff[i]= 0;
    // ...
   }
}
```

Exemple de analiză statică

Comparaţie OR cu o constantă>0

```
#define INPUT_TYPE 2
void f(int n)
{
  if(INPUT_TYPE || n)
  {
    puts("întotdeauna");
  }
  else
  {
    puts("niciodată");
  }
}
```

```
#define INPUT_TYPE 2
void f(int n)
{
  if((INPUT_TYPE & n) == 2)
  {
    puts("comparaţie AND pe biţi adevărată");
  }
  else
  {
    puts("comparaţie AND pe biţi falsă");
  }
}
```

Exemple de analiză statică

Operator incorect

```
void f( int i )
{
   while (i = 5)
   {
   // ...
   }
}
```

```
void f( int i )
{
   while (i == 5)
   {
   // ...
   }
}
```

Exemple de analiză statică

Incrementare eronată a contorului

```
void f( )
{
   int i;

   for (i = 100; i >= 0; i++)
   {
      //  ...
   }
}
```

```
void f( )
{
   int i;

   for (i = 100; i >= 0; i--)
   {
      //  ...
   }
}
```

Exemple de analiză statică

Comparaţie între bit field şi tipul boolean

```
struct myBits
{
  short flag : 1;
  short done : 1;
  //...
} bitType;

void f( )
{
  if (bitType.flag == 1)
  {
  // ...
  }
}
```

```
void f ()
{

if(bitType.flag==bitType.done)
  {
    // ...
  }
}
```

Exemple de analiză statică

Utilizare sizeof pentru o expresie

```
void f( )
{
  size_t x;
  char a[10];

  x= sizeof (a - 4);
  // ...
}
```

```
void f( )
{
  size_t x;
  char a[10];

  x= sizeof (a) - 4;
  // ...
}
```

Exemple de analiză statică

Recursivitate infinită

```
void f(int n)
{
  f=n*f(n-1);
}
```

```
void f(int n)
{
  if (n==1) return 1;
          else  f=n*f(n-1);
}
```

Exemple de analiză statică

Ciclu infinit

```
int i=2;
while (i)
{
  // ... i nu este modificat
}
```

```
int i=2;
while (i)
{
  // ... i este modificat
}
```

```
while (true)
{
  // ... nu există condiţie de ieşire break
}
```

```
while (true)
{
  // ...
 if (x==y) break;
}
```

Testare automată

- Testarea automată(automatic) vs. Testare automatizată(automated)
- Automată -> automatizată -> asistată de calculator
- Nu există automatizare care să substituie 100% percepţia umană
- Proces de automatizare (nu 100%) – la nivel de creare a testelor, la execuţie, la verificare

„*Automated testing depends on our ability to programaticaly detect when the software under test fails”.* [Kaner]

„Our ability to automate testing is fundamentaly constrained by out ability to create and use oracles*”.* [Kaner]

„Test automation is the use of software to control the execution of tests, the comparison of actual outcomes to predicted outcomes, the setting up of test preconditions, and other test control and test reporting functions„
[Adam Kolawa]

Produse software

- Mercury Interactive – WinRunner, QTP (QuickTest Professional)
- Rational – TestManager, TeamTest
- Seapine – QAWizard
- Segue – SilkVision, SilkTest
- Jakarta - Cactus

Avantajele automatizării testării [Peter Farrell-Vinay]

- Teste manuale necesită o lungă perioadă de timp pentru fi executate. Caracteristici stabile -> pot fi automatizate
- Testele de încărcare şi cele de performanţă nu pot fi executate fără unele automatizări
- Testele de încredere şi cele de regresie trebuie să fie rulate rapid. Sisteme ce depăşesc 100 KLOC (kilo lines of code) - doar testare automată.
- Testele pe un număr mare de configuraţii -> componentă repetitivă -> avertizare promptă în caz de incompatibilitate
- Unele părţi din testele manuale pot fi repetitive. Mici teste automate pot creşte viteza foarte mult

Avantajele automatizării testării (continuare)

- Testele care implică interacţiuni între intrările utilizatorilor - nu pot acoperii toate combinaţiile posibile, fără automatizare

- Automatizarea testelor poate ajuta la popularea bazelor de date şi la generarea de fişiere

- Sistemul are mai multe versiuni -> teste de regresie -> automatizare

- În cazul în care costul de a menţine testarea este ridicat, atunci se vor automatiza doar caracteristicile esenţiale

Dezavantajele automatizării testării [Peter Farrell-Vinay]

- Testarea automată nu poate înlocui testarea manuală
- Unii testori nu au experienţă în programare -> nu pot automatiza procesul de testare
- Schimbarea interfeţei cu utilizatorul poate duce la incompatibilitate cu testele automate, deja create
- Necesită timp mare de implementare şi documentare
- Uneori testarea automată devine inutilă d.p.d.v. al costurilor

Testarea manuală vs. Testarea automatizată

Problemă la nivel de companie : Luarea deciziei de a automatiza sau nu testele.

Factori:

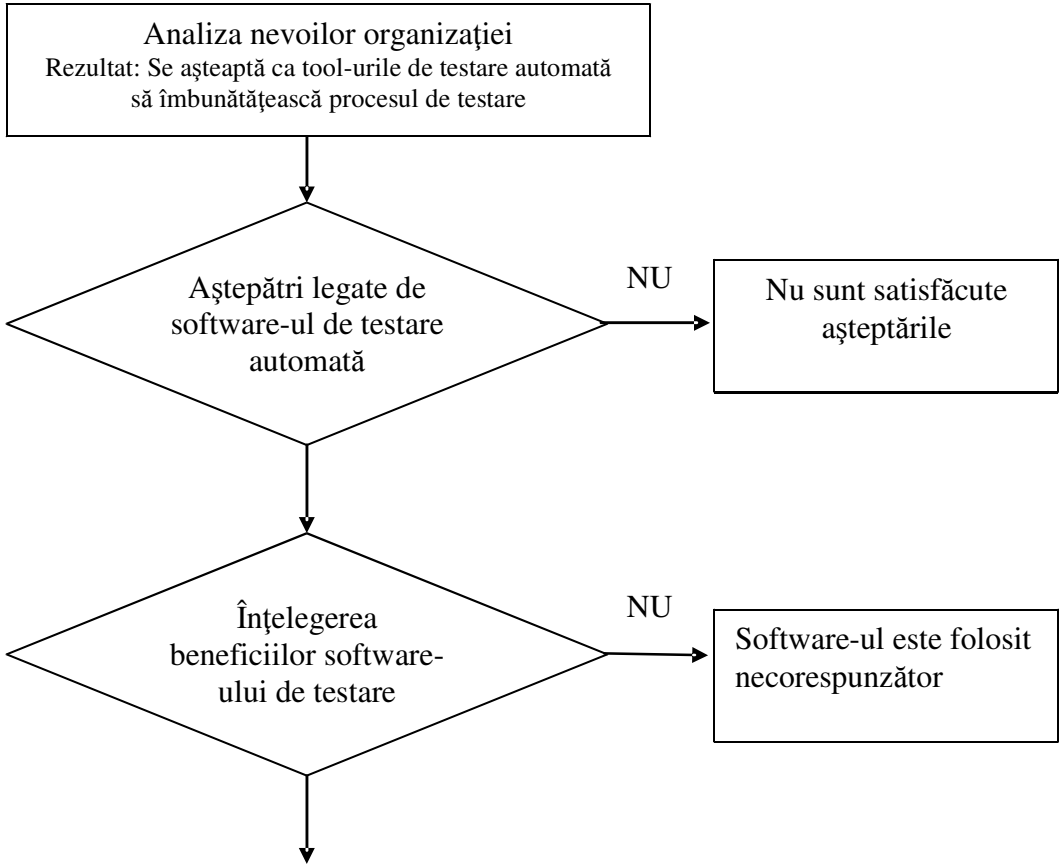
- riscuri,
- costuri,
- calitate,
- timp.

Cu cât un testor petrece mai mult timp verificând un modul cu atât cresc şansele de a găsi mai multe defecte şi posibile greşeli

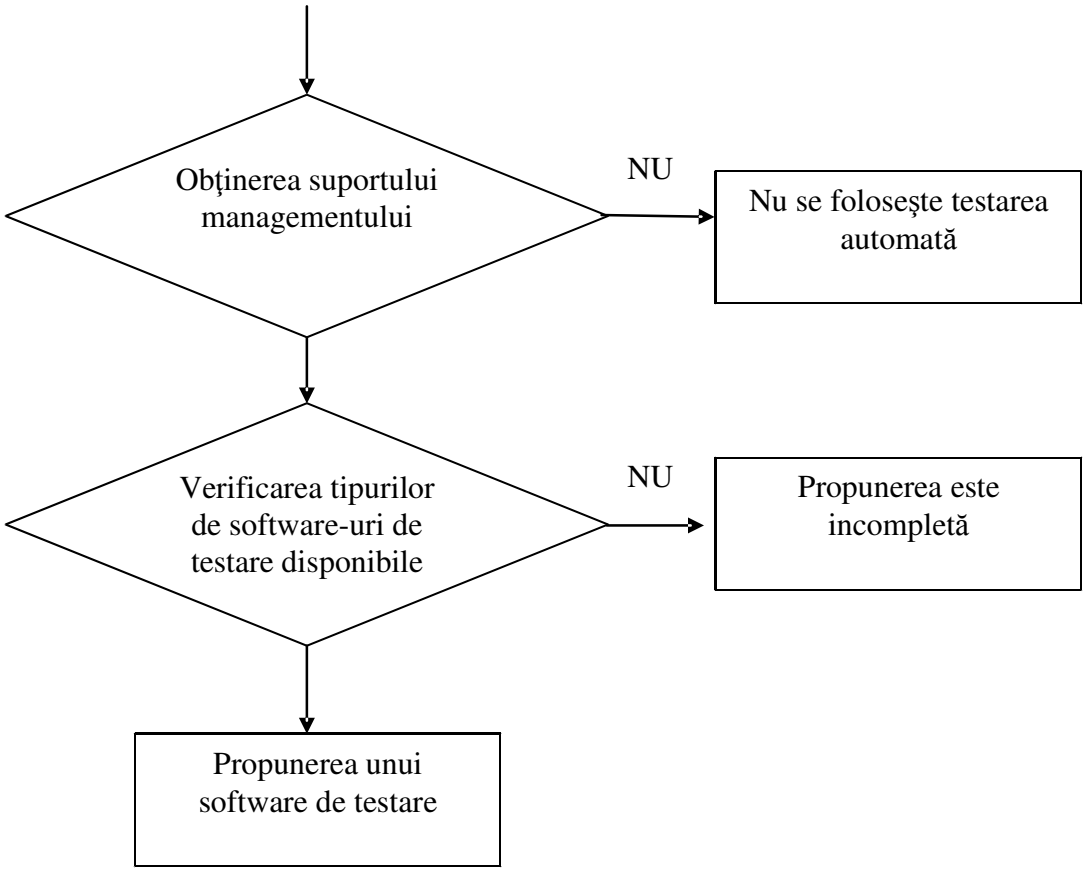
Când se automatizează testele?

- după ce specificaţiile au fost scrise
- în funcţie de specificaţii, testele pot fi dezvoltate înainte de implementarea propriu-zisă
- este necesar ca software-ul de automatizare să poată recunoaşte toate componentele aplicaţiei testate
- este necesară o cunoaştere/familiarizare prealabilă a testorilor cu software-ul de automatizare. Se verifică dacă este potrivit aplicaţiei de testat.

Luarea deciziei de automatizare



Luarea deciziei de automatizare



Costurile automatizării

- costul de evaluare a instrumentelor de pe piaţă
- costul instrumentului
- costul de instruire a personalului pentru a utiliza instrumentul
- costul creării de teste automate
- costul de întreţinere a testelor automate
- cost (posibil) al unui instrument de profilare a codului
- costul de a avea o structură specială pentru verificarea statică a codului
- costul de integrare a instrumentului în structura de rulare/generare a testelor

Metode de testare automatizată

Validarea rezultatelor

- Testarea interfeţei grafice *(GUI - Graphical User Interface)* – generarea de evenimente legate de interfaţa cu utilizatorul: intrări de la tastatură, mouse.
- Testarea codului – Interfeţele publice ale claselor/modulelor/bibliotecilor, sunt testate cu diferiţi parametri de intrare

+ testarea exploratorie (testori experimentaţi)

Testarea GUI

- Înregistrarea interactivă a acţiunilor utilizator
- Recunoaşterea componentelor software-ului (butoane, casete de dialog, imagini, ...)
- Uneori nu necesită scriere de cod
- Aplicabilă oricărui software cu interfaţă grafică
- Modificarea ulterioară a poziţiei/numelui componentelor poate duce la eşuarea testului înregistrat anterior
- Permite adăugarea de cod (scripturi) în vederea realizării automate de cazuri de test pornid de la un test înregistrat manual

Testarea codului

- xUnit framework (jUnit, nUnit, phpUnit) = code driven testing
- Kent Beck, *Simple Smalltalk Testing: With Patterns* – prima soluţie de scriere şi verificare a testelor în Smalltalk, primul framework xUnit
- Testarea GUI -> erori false (schimbarea poziţiei sau numelui obiectelor) -> timp pierdut în verificare acestori posibile erori
- Se poate testa fiecare funcţie în parte

Oracole

Definiţii

- *prophet: an authoritative person who divines the future*
- *a prophecy (usually obscure or allegorical) revealed by a priest or priestess; believed to be infallible*
- *an oracle is a mechanism used by software testers and software engineers for determining whether a test has passed or failed*

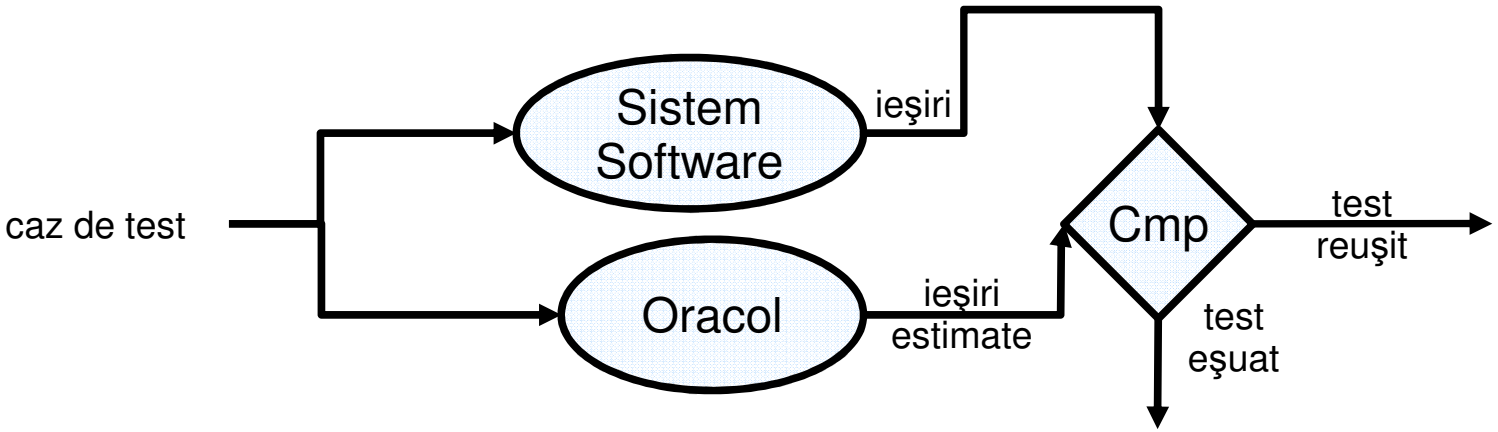
Testarea utilizând oracole

„Our ability to automate testing is fundamentaly constrained by our ability to create and use oracles.”

[Kaner]

Oracol
- mecanism software cu ajutorul căruia se stabileşte dacă o procedură de testare a fost îndeplinită cu succes sau nu.

- compară ieşirile sistemului testat cu ieşirile estimate de către oracol



Oracole. Structură [Kaner]

- *Generator* – rezultate estimate sau aşteptate pt. fiecare test
- *Comparator* - compară rezultatele obţinute cu cele estimate
- *Evaluator* – determină dacă valorile comparate sunt suficient de apropiate pt. a declara că testul a trecut sau nu

Tipuri de oracole [Hoffman]

	Oracole propriu-zise	Oracole euristice	Oracole de verificare a consistenţei	Oracole autoverificabile	Oracol pasiv
Definiţie	Generează toate rezultatele aşteptate	Verifică doar anumite valori, cât şi consistenţa celorlalte	Compară rezultatele curente cu cele ale versiunilor anterioare	Include răspunsul în mesaj	Nu verifică corectitudinea rezultatelor, doar prezenţa lor
Avantaje	- toate erorile sunt detectate	- mai rapide şi mai uşoare decât Oracolele propriu-zise	- metodă rapidă - poate verifica cantităţi mari de date într-un timp scurt	- analiză post-testare - verificarea pe baza mesajelor - generează şi verifică cantităţi mari de date	- pot rula orice cantităţi de date
Dezavantaje	- cost ridicat - necesită timp mare de implementare	- poate „scăpa” erorile sistematice **(funcţie sinus eronată)**	- versiunile anterioare pot ascunde erori nedetectate	- necesită definirea de răspunsuri şi generarea de mesaje care să le conţină	- doar defectele majore sunt interceptate

Douglas Hoffman, „Oracle Strategies for Automated Testing”, 2000

Oracole. Exemple

- Copy/Paste – MS Office Word vs. OO Writer
- Drag & Drop – MS Office Excel vs. OO Calc
- Bara de adresă – Mozilla vs. IE
- Afişare pagini web - Mozilla vs. IE

Test-Driven Development (TDD)

Reguli [Kent Beck]

- *Never write a single line of code unless you have a failing automated test*
- *Eliminate duplication*

Manifesto for Agile Software Development

We are uncovering better ways of developing software
by doing it and helping others do it.
Through this work we have come to value:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on
the right, we value the items on the left more.
[beck-manifesto]
=> Agile ideals are inherently goal-oriented

Tipuri de teste automate în TDD

Testele programatorilor

- *un fel de* testare a componentelor – scop diferit
- de preferat a fi scrise în acelaşi limbaj ca şi codul sursă

Testele clienţilor

- specifică funcţionalitatea de care clientul are nevoie
- numite teste de acceptanţă
- scrise într-un limbaj pe care clientul îl înţelege -> clientul scrie teste - Framework for Integrated Test (FIT) http://fit.c2.com

Red/Green/Refactor [Kent Beck]

Procesul de implementare a fiecărui test

1. Scrierea testului
2. Compilarea testului - obligatoriu trebuie să „pice” testul (pt. că nu este nimic implementat, nu există încă funcţiile apelate)
3. Implementarea minimală a codului sursă – cât să compileze
4. Rularea testului – testul trebuie să „pice”
5. Implementarea minimală a codului sursă – cât să „treacă” testul
6. Rularea testului – testul trebuie să „treacă”
7. Refactorizare – pt. claritatea codului şi eliminarea duplicităţii
8. Repetarea pasului 1

Red/Green/Refactor - utilitate

- paşi cât mai mici
- cu cât pasul e mai mic, cu atât mai uşor de găsit o eroare
- testele produc feedback imediat – nici nu mai e nevoie de debugger (rularea pas cu pas a programului) – se va ştii locul în care s-a produs defectul
- siguranţă în dezvoltare

Exemplu implementare TDD [James Newkirk]

Problemă

Implementarea unei structuri de date de tip stivă, care să permită următoarele operaţii: Push, Pop, Top şi IsEmpty.

Lista testelor

- Crează *Stivă* şi verifică *IsEmpty=true*
- Adaugă un obiect în stivă şi verifică *IsEmpty=false*
- Adaugă un obiect, scoate obiectul din stivă şi verifică *IsEmpty=true*
- Adaugă un obiect în stivă, memorează, scoate obiect şi verifică dacă obiectele sunt identice
- Adaugă trei obiecte în stivă, memorează, apoi scoate trei obiecte din stivă şi verifică dacă sunt scoase în ordinea corectă
- Scoate obiect dintr-o stivă goală
- Adaugă un obiect, apelează *Top* şi verifică dacă *IsEmpty=false*
- Adaugă un obiect, memorează, apelează *Top* şi verifică dacă cele două obiecte sunt identice
- Apelează *Top* pe o stivă goală

Alegerea primului test

- Cel mai simplu
- Cel mai apropiat de esenţa problemei

Test1: Creează unui obiect de tip stivă şi verifică dacă *IsEmpty=true*

PAS 1. Scrierea testului. Nu va trebui să compileze

```
[Test]
    public void Empty()
    {
        Stack stack = new Stack();
        Assert.IsTrue(stack.IsEmpty);
    }
```

PAS 2. Implementarea codului pentru compilare. Nunit → fail(red)

```
public class Stack
{
    private bool isEmpty = false;

    public bool IsEmpty
    {
        get
        {
            return isEmpty;
        }
    }
}
```

PAS 3. Implementarea minimală a codului pentru Nunit → success(green)

```
public class Stack
{
    private bool isEmpty = false;

    public bool IsEmpty
    {
        get
        {
            return true;
        }
    }
}
```

Test2: Adaugă un obiect în stivă şi verifică dacă *IsEmpty=false*

PAS 1. Scrierea testului. Nu va trebui să compileze

```
[Test]
   public void PushOneTest()
   {
      Stack stack = new Stack();
      stack.Push("first element");
      Assert.IsFalse(stack.IsEmpty,
        "After Push, IsEmpty should be
false");
   }
```

PAS 2. Implementarea codului pentru compilare. Nunit → fail(red)

```
public void Push(object element)
  {
  }
```

Nunit: .PushOneTest : After Push, IsEmpty should be false

PAS 3. Implementarea minimală a codului pentru Nunit → success(green)

```
public void Push(object element)
  {
  IsEmpty=false;
  }
```

PAS 4. Refactorizare

```
[TestFixture]
public class StackFixture
{
    private Stack stack;

    [SetUp]
    public void Init()
    {
        stack = new Stack();
    }

    [Test]
    public void EmptyTest()
    {
        Assert.IsTrue(stack.IsEmpty);
    }

    [Test]
    public void PushOneTest()
    {
        stack.Push("first element");
        Assert.IsFalse(stack.IsEmpty,
               "After Push, IsEmpty should be false");
    }
}
```

Test3: Adaugă un obiect în stivă, scoate obiect şi verifică dacă *IsEmpty=true*

PAS 1. Scrierea testului. Nu va trebui să compileze

```
[Test]
    public void PopTest()
    {
        stack.Push("first element");
        stack.Pop();
        Assert.IsTrue(stack.IsEmpty,
            "After Push - Pop, IsEmpty should be
true");
    }
```

PAS 2. Implementarea codului pentru compilare. Nunit → fail(red)

```
public void Pop()
{
}
```

Nunit: .PopTest : After Push - Pop, IsEmpty should be true

PAS 3. Implementarea minimală a codului pentru Nunit → success(green)

```
public void Pop()
    {
        isEmpty = true;
    }
```

Test4: Adaugă un obiect în stivă, memorează obiect, scoate obiect şi verifică dacă obiectele sunt identice

PAS 1. Scrierea testului. Nu va trebui să compileze

```
[Test]
    public void PushPopContentTest()
    {
        int expected = 1234;
        stack.Push(expected);
        int actual = (int)stack.Pop();
        Assert.AreEqual(expected, actual);
    }
```

PAS 2. Implementarea codului pentru compilare. Nunit → fail(red)

```
public object Pop()
{
    isEmpty = true;
    return null;
}
```

Nunit: .PushPopContentTest : System.NullReferenceException :
Object reference not set to an instance of an object.

PAS 3. Implementarea minimală a codului pentru Nunit → success(green)

```
public class Stack
{
    private bool isEmpty = true;
    private object element;

    public bool IsEmpty    {
        get{
            return isEmpty;
        }
    }

    public void Push(object element)    {
        this.element = element;
        isEmpty = false;
    }

    public object Pop()    {
        isEmpty = true;
        object top = element;
        element = null;

        return top;
    }
}
```

PAS 4. Refactorizare

```
public class Stack
{
    private object element;

    public bool IsEmpty
    {
        get
        {
            return (element == null);
        }
    }

    public void Push(object element)
    {
        this.element = element;
    }

    public object Pop()
    {
        object top = element;
        element = null;

        return top;
    }
}
```

Test5: Adaugă trei obiecte în stivă, memorează obiecte, scoate obiecte şi verifică dacă obiectele sunt scoase în ordinea inversă adăugării

PAS 1. Scrierea testului. Nu va trebui să compileze

```
[Test]
    public void PushPopMultipleElementsTest()
    {
        string pushed1 = "1";          stack.Push(pushed1);
        string pushed2 = "2";          stack.Push(pushed2);
        string pushed3 = "3";          stack.Push(pushed3);
        string popped = (string)stack.Pop(); Assert.AreEqual(pushed3, popped);
        popped = (string)stack.Pop();        Assert.AreEqual(pushed2, popped);
        popped = (string)stack.Pop();        Assert.AreEqual(pushed1, popped);
    }
```

PAS 2. Codul compilează. Nunit → fail(red)

```
Nunit: .PushPopMultipleElementsTest :
  expected:<"2">
       but was:<(null)>
```

PAS 3. Implementarea minimală a codului pentru Nunit → success(green)

```
public class Stack
{
    private ArrayList elements = new ArrayList();

    public bool IsEmpty
    {
        get
        {
            return (elements.Count == 0);
        }
    }

    public void Push(object element)
    {
        elements.Insert(0, element);
    }

    public object Pop()
    {
        object top = elements[0];
        elements.RemoveAt(0);
        return top;
    }
}
```

Test6: Scoate obiect dintr-o stivă goală (*IsEmpty=true*)

PAS 1. Scrierea testului. Nu va trebui să compileze

```
[Test]
   [ExpectedException(typeof(InvalidOperationException))]
   public void PopEmptyStackTest()
   {
      stack.Pop();
   }
```

PAS 2. Codul compilează. Nunit → fail(red)

Nunit: .PopEmptyStackTest : Expected: InvalidOperationException
but was ArgumentOutOfRangeException

PAS 3. Implementarea minimală a codului pentru Nunit → success(green)

```
public object Pop()
    {
        if(IsEmpty) throw
new InvalidOperationException("cannot pop an empty
stack");

        object top = elements[0];
        elements.RemoveAt(0);
        return top;
    }
```

Test7: Adaugă un obiect, apelează *Top* şi verifică dacă *IsEmpty=false*

PAS 1. Scrierea testului. Nu va trebui să compileze

```
[Test]
    public void PushTopTest()
    {
        stack.Push("42");
        stack.Top();
        Assert.IsFalse(stack.IsEmpty);
    }
```

PAS 2,3. Implementarea codului pentru compilare. Nunit → success(green)

```
public object Top()
{
      return null;
   }
```

Test8: Adaugă un obiect, memorează, apelează *Top* şi verifică dacă obiectele sunt identice

PAS 1. Scrierea testului. Nu va trebui să compileze

```
[Test]
   public void PushTopContentCheckOneElementTest()
   {
      string pushed = "42";
      stack.Push(pushed);
      string topped = (string)stack.Top();
      Assert.Equals(pushed, topped);
   }
```

PAS 2. Implementarea codului pentru compilare. Nunit → fail(red)

Nunit: PushTopContentCheckOneElement :
expected:<"42">
but was:<(null)>

PAS 3. Implementarea minimală a codului pentru Nunit → success(green)

```
public object Top()
   {
      return elements[0];
   }
```

11 stiluri de testare software [Kaner & Bach]

1. Testarea funcţiilor – *functional testing*
2. Testarea domeniilor de valori – *domain testing*
3. Testarea bazată pe specificaţii – *specification-based testing*
4. Testarea axată pe riscuri – *risk-based testing*
5. Testarea la limită – *stress testing*
6. Testarea de regresie – *regression testing*
7. Testarea de către utilizatori – *user testing*
8. Testarea bazată pe scenarii – *scenario testing*
9. Testarea bazată pe stări – *state-model based testing*
10. Testarea automată de volum ridicat – *high volume automated testing*
11. Testarea de explorare – *exploratory testing*

1. Testarea funcţiilor

- fiecare funcţie este testată separat
- se testează funcţionalitatea de bază
- teste uşoare, cu caracter general, nu foarte precise
- cerinţe -> use cases -> teste funcţionale
- Testare funcţională vs. Testare non-funcţională (ex.)

  portabilitate, securitate, consum de resurse, încărcare

2. Testarea domeniilor de valori

- împărţirea în clase de echivalenţă
- alegerea reprezentanţilor
- valorile limită

3. Testarea bazată pe specificaţii

- un fel de testare funcţională
- parcurge amănunţit specificaţiile -> cazuri de test
- specificaţii amănunţite -> valoare ridicată a testării
- specificaţii vagi -> valoare scăzută a testării

4. Testarea axată pe riscuri

- Testorul imaginează cazuri de test bazate pe risc (Ex.)
- Gruparea testelor în funcţie de risc -> clasificare puternică
- Uşurinţă în crearea testelor, cu focusare pe un anumit risc
- Testorul gândeşte din perspectiva riscurilor, programatorul din perspectiva specificaţiilor

5. Testarea la limită

- se urmăreşte depăşirea limitelor impuse (Ex.)

- foarte importantă pentru aplicaţiile online, multiuser (nr. utilizatori, nr. conexiuni DB)

- orice aplicaţie trebuie testată la limită (analogie cu testarea domeniului de valori) – se testează atât valorile limită specificate, cât şi valorile ce depăşesc aceste limite

- stabileşte stabilitatea sistemului

6. Testarea de regresie

- necesită atenţie deosebită în crearea testelor (manual vs. automat)
- refolosirea cazurilor de test în versiuni ulterioare
- documentaţie foarte atentă pentru mentenanţă
- verifică dacă modificările aduse proiectului au influenţat sau nu funcţionalitatea conform specificaţiilor

7. Testarea de către utilizatori

- utilizatori reali
- testarea: pe baza specificaţiilor sau nu
- testele arată impactul asupra utilizatorilor finali – gradul de acceptanţă a acestora
- majoritatea acestor teste vor fi simpliste

8. Testarea bazată pe scenarii

- Dezvoltă cazuri de test complexe – pe scenarii
- caracteristici: motivaţie, credibilitate, complexitate, uşor de evaluat
- Soap Operas [Hans Buwalda]

9. Testarea bazată pe stări

- *a*utomat cu stări finite
- teste automate bazate pe model
- caracteristici: credibil şi motivant

10. Testarea automată de volum ridicat

- număr mare de teste
- rezultatele vor fi testate cu un *oracol*
- restele sunt executate şi interpretate de către calculator
- Testare automată vs. Testare asistată de calculator

11. Testarea de explorare

- proiectarea de noi teste be baza rezultatelor obţinute de testele existente
- învăţare din experienţa testelor anterioare
- explorare = experienţă + creativitate + învăţare

Metrici software. Avantaje/dezavantaje

Avantaje

- Creşterea cercetării în domeniu -> număr lucrări ştiinţifice, cărţi
- Companiile producătoare de software – adoptă metricile

Dezavantaje

- Anumite metrici sunt aplicabile programelor mici
- Există metrici de măsurare a codului – irelevant pentru industrie

„What theory is doing with respect to measurement of software work and what practice is doing are on two different planes, planes that are shifting in different directions” [Glass 1994]

Definiţii

„Measurement is the assignment of numbers to objects or events according to rule”

[Stevens]

-> „Measurement is the assignment of numbers to objects or events according to a rule derived from a model or theory”

[Kaner]

"Measurement is the process of empirical, objective, assignment of numbers to properties of objects or events of the real world in such a way as to describe them"

[Finkelstein]

"Measurement is the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to characterize them according to clearly defined rules"

[Fenton]

Teoria măsurării

Factori

- atributul ce este măsurat (scara de măsură, variaţiile)
- instrumentul de măsurare a atributului (scara de măsură, variaţiile)
- relaţia dintre atribut şi instrument
- efecte secundare în măsurare
- scopul măsurătorii

Exemple

- nr. de defecte – măsoară eficienţa testorului
- formule de măsurare a codului sursă – complexitatea programului

Metrici

- metrică = funcţie de măsurare

- metrică software =

  "a function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which software possesses a given attribute that affects its quality." [Kaner]

- metrică :

  calitativă

  cantitativă

Metrici. Intrebări

1. Care este motivul ?

2. Care este scopul ? Atenţie mare la utilizare – metricile pot fi folosite greşit

3. Care este atributul măsurat ? Care este scara ?

4. Ce instrument de măsurare ? Care este scara ?

IEEE 1061 standard

- *corelare* liniară între metrică şi factorul de calitate
- *consistenţă* – M:A->B a1<a2 ==> M(a1)<M(a2) – monotonie (a1,a2 fact. calitate)
- *predicţie* – M(a1,t) = cunoscut ==> a1 = estimat
- *discriminare* – calitate ridicată vs. calitate scăzută
- *fiabilitate*

Metrici directe

"A metric that does not depend upon a measure of any other attribute."

[IEEE 1061]

- Metrici directe (o variabilă) vs. indirecte (mai multe variabile)

- MTTF (Mean Time To Failure) [IEEE 1061] = metrică directă asupra fiabilităţii. Corect? (este influenţată de alte variabile ?)

Variabile: utilizatorul

timpul de rulare

Exemple de metrici directe: lungimea codului sursă, durata de testare, numărul defectelor descoperite, timpul alocat de testor

Exemple de metrici indirecte: productivitatea programatorului (linii de cod/timp implementatre), densitatea defectelor (defecte/dimensiunea modulului testat), stabilitatea cerinţelor (nr. iniţiat/nr. final)

Exemplu 1

- *Atribut* = lungimea unei mese
- *Scara atributului* = scară gradată
- *Variaţia atributului* = 120 cm ± 0.5 cm
- *Instrument* = liniar
- *Variaţia instrumentului* = 0.3 cm
- *Relaţie atribut/instrument* = directă (modificarea atributului -> modificare măsurătorii)
- Efecte secundare = măsuratoare incorectă folosind liniar de mică lungime -> prea multe măsurări

Exemplu 2

Concurs de alergare. A – locul 1 (10.000 Ron), B – locul 2 (1.000 Ron),
C – locul 3 (100 Ron).

- *Atribut* = viteza alergătorilor
- *Scara atributului* = m/s
- *Variaţia atributului* = în funcţie de participanţi – la alt concurs poate fi complet diferit
- *Instrument* = numărarea manuală a locurilor pe care au sosit participanţii

Exemplu 2 (continuare)

- *Scara instrumentului* = scară ordinală/poziţională/nominală (1,2,3,...)

  scară gradată - „A este de 2 ori mai rapid decât B şi de 3 ori mai rapid decât C”?

  scară de tip interval - „diferenţa dintre A şi B este egală cu diferenţa dintre B şi C” ?

(ex. scară ordinală : atribut = gravitatea unui defect, scara = blocant, critic, major, mediu, minor, ... Diferenţa dintre „blocant” şi „critic” este egală cu diferenţa dintre „mediu” şi ”minor” ?)

- *Variaţia instrumentului* = mai mulţi arbitri ar putea avea percepţii diferite dacă cursa este strânsă
- *Relaţie atribut/instrument* = legătură între atribut (viteză) şi instrument (loc în clasament) – cu cât viteza este mai mare, cu atât locul este mai bun în clasament
- Efecte secundare = depind doar de concurenţi

Exemplu 3

Contorizarea numărului de defecte detectate de către testor

- *Atribut* = gradul de performanţă al testorului (eficienţă, calitate, îndemânare)
- *Scara atributului* = ..?.. Testorul A găseşte 2 defecte blocante, iar testorul B găseşte 20 de defecte minore. Care este scara ? se stabileşte per proiect
- *Variaţia atributului* = există dar nu poate fi definită. Testorul A are productivitate diferită în zile diferite – natura umană ;) - „knowledge worker”
- *Instrument* = nu există un instrument anume. Se poate folosi numărarea defectelor

Exemplu 3 (continuare)

- *Scara instrumentului* = scară nominală (blocant, critic, major,...). Un defect blocant nu este de două ori mai important decât un defect critic.
- *Variaţia instrumentului* = nu există
- *Relaţie atribut/instrument* = legătură între atribut (performanţa testorului) şi instrument

(nr. de defecte raportate) – cu cât numărul de defecte este mai mic, cu atât ..?.., cu cât numărul de defecte este mai mare, cu atât ..?..

Instrumentul NU poate măsura productivitatea

- Efecte secundare =

dacă testorul ştie că este evaluat pe baza numărului de defecte, va raporta mai multe defecte.
Productiv pt. Companie ?
Influenţează direct evaluarea programatorului
Se vor vâna defecte cu gravitate scăzută – sunt mai uşor de detectat

CONCLUZIE: Este utilă măsurarea numărului de defecte ?
DA. Dar nu pt. evaluarea testorilor ci doar a bunului mers al proiectului.

Măsurarea nivelului de testare a unui proiect software

- 10 testori, testează 30 minute aplicaţia A=B+C. Răspunsuri diferite la intrebarea „Cât de mult aţi testat aplicaţia (0% - 100 %)?”
- testarea completă = ∞ -> xx% din ∞ = ∞
- testarea completă ≈ nivel ridicat de testare al produsului (testare suficientă)

Metrici de acoperire

- Acoperirea liniilor de cod – testarea fiecărei linii de cod
- Acoperirea ramurilor
- Acoperirea căilor (def-use coverage)
- Acoperirea tuturor condiţiilor – toate valorile logice pe care le pot lua operanzii
- Acoperirea ciclurilor
- Acoperirea datelor
- Memorie liberă/resurse disponibile
- Şiruri de caractere hardcodate
- Fiecare meniu/submeniu
- Fiecare dialog

Metrici de planificare

$$M = \frac{realizat}{platificat}$$

- % testelor dezvoltate din cele planificat
- % testelor planificate şi executate
- % specificaţiilor ce s-au transformat în teste
- % testelor executate şi care au trecut
- Ore de testare (sau numărul de teste executate) în funcţie de prioritate

Metrici de proiect

- Timp alocat în comparaţie cu proiectele anterioare
- Numărul defectelor descoperite, deschise, rezolvate (comparativ cu prj. anterioare)
- Numărul defectelor descoperite per săptămână (comparativ cu prj. anterioare)
- Numărul (sau %) modulelor implementate până la momentul curent