

Ingineria sistemelor soft

Curs 11

Testarea sistemelor soft

*Suport de curs bazat pe **B. Bruegge and A.H. Dutoit***

"Object-Oriented Software Engineering using UML, Patterns, and Java"

Testarea sistemelor soft

- *Testarea* = procesul de identificare a diferențelor dintre comportamentul dorit/așteptat al sistemului (specificat cu ajutorul modelelor) și comportamentul observat al acestuia
 - *Testarea unitară* - identifică diferențe dintre specificarea unui obiect și implementarea acestuia ca și componentă
 - *Testarea structurală* - identifică diferențe dintre modelul aferent proiectării de sistem și comportamentul unei grup de subsisteme integrate
 - *Testarea funcțională* - identifică diferențe dintre modelul cazurilor de utilizare și sistem
 - *Testarea performanței* - identifică diferențe dintre cerințele nefuncționale și performanțele sistemului
- Din perspectiva modelării, testarea reprezintă o încercare de a demonstra că implementarea sistemului este inconsistentă cu modelele acestuia
 - Scopul este proiectarea unor teste care să pună în evidență defectele sistemului

Fiabilitatea sistemelor soft

- *Corectitudinea* unui sistem în raport cu specificația - vizează concordanța dintre comportamentul observat al sistemului și specificarea acestuia.
 - *Fiabilitatea softului* = probabilitatea ca acel soft să nu cauzeze eșecul sistemului, pentru o anumită perioadă de timp și în condiții specificate [IEEE Std. 982.2-1988]
- *Eșec* (eng. *failure*) = orice deviere a comportamentului observat de la cel specificat/așteptat
- *Eroare/stare de eroare* (eng. *erroneous state*) = orice stare a sistemului în care procesările ulterioare ar conduce la eșec
- *Defect* (eng. *fault/defect/bug*) = cauza mecanică sau algoritmică a unei stări de eroare
- *Testare* = încercarea sistematică și planificată de a găsi defecte în softul implementat
 - "Testing can only show the presence of bugs, not their absence." (E. Dijkstra)

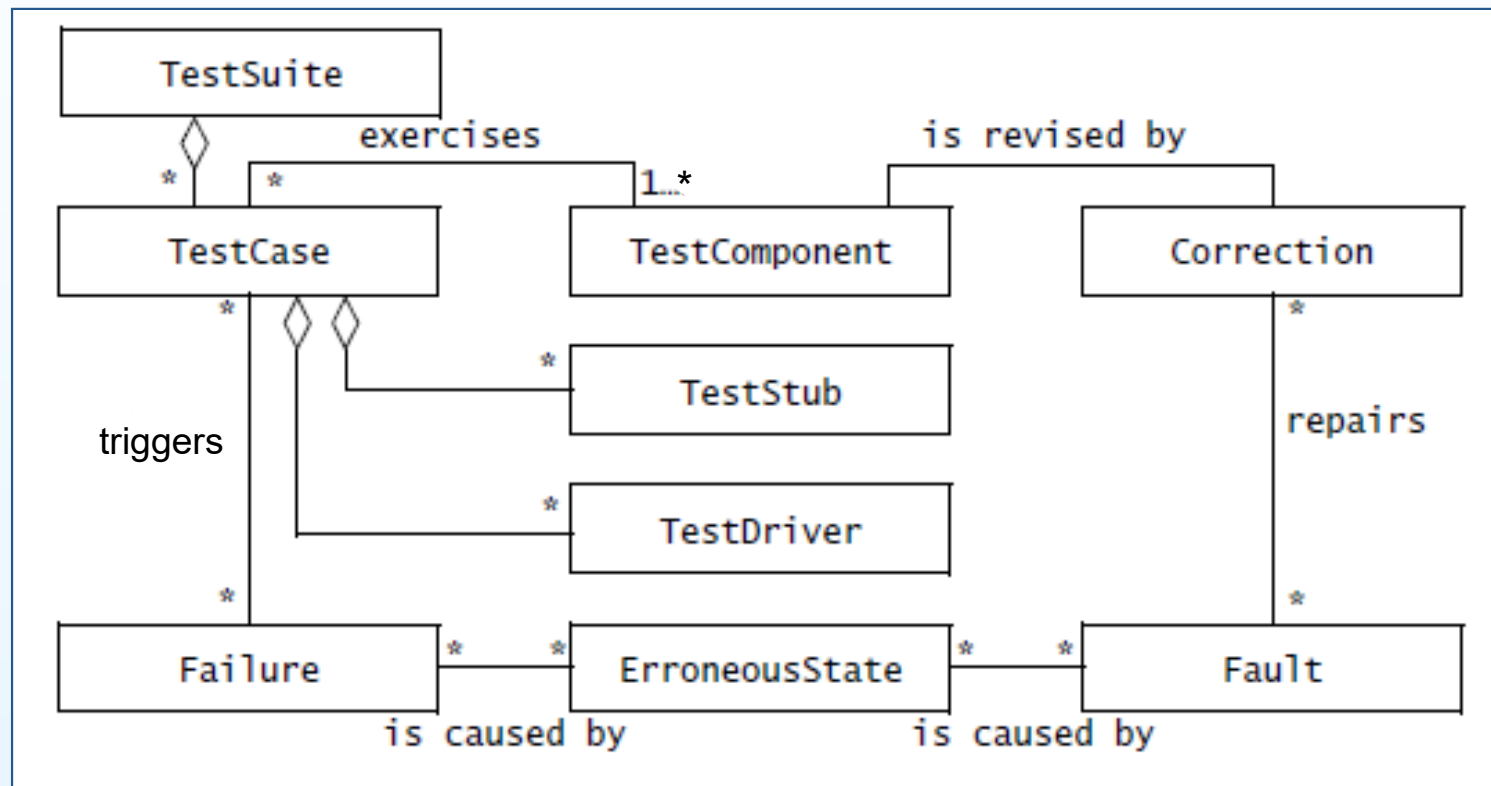
Fiabilitatea sistemelor soft (cont.)

- Tehnici de creștere a fiabilității sistemelor soft
 - *Tehnici privind evitarea defectelor (eng. fault avoidance techniques)*
 - identifică posibilele defecte la nivel static (fără o execuție a modelelor/codului) și încearcă să prevină introducerea acestora în sistem
 - ex.: metodologii formale - Cleanroom, Correctness by Construction
 - *Tehnici privind detectarea defectelor (eng. fault detection techniques)*
 - sunt utilizate în timpul procesului de dezvoltare, pentru a identifica stările de eroare și defectele care le-au provocat, anterior livrării sistemului (ex. review, testare, depanare)
 - nu își propun recuperarea sistemului din stările de eșec induse de defectele identificate
 - *Tehnici privind tolerarea defectelor (eng. fault tolerance techniques)*
 - pornesc de la premisa că sistemul poate fi livrat cu defecte și că eventualele eșecuri pot fi gestionate prin recuperare în urma lor la execuție
 - ex.: sistemele redundante utilizează mai multe calculatoare și softuri diferite pentru realizarea acelorași sarcini

Testarea sistemelor soft - concepte

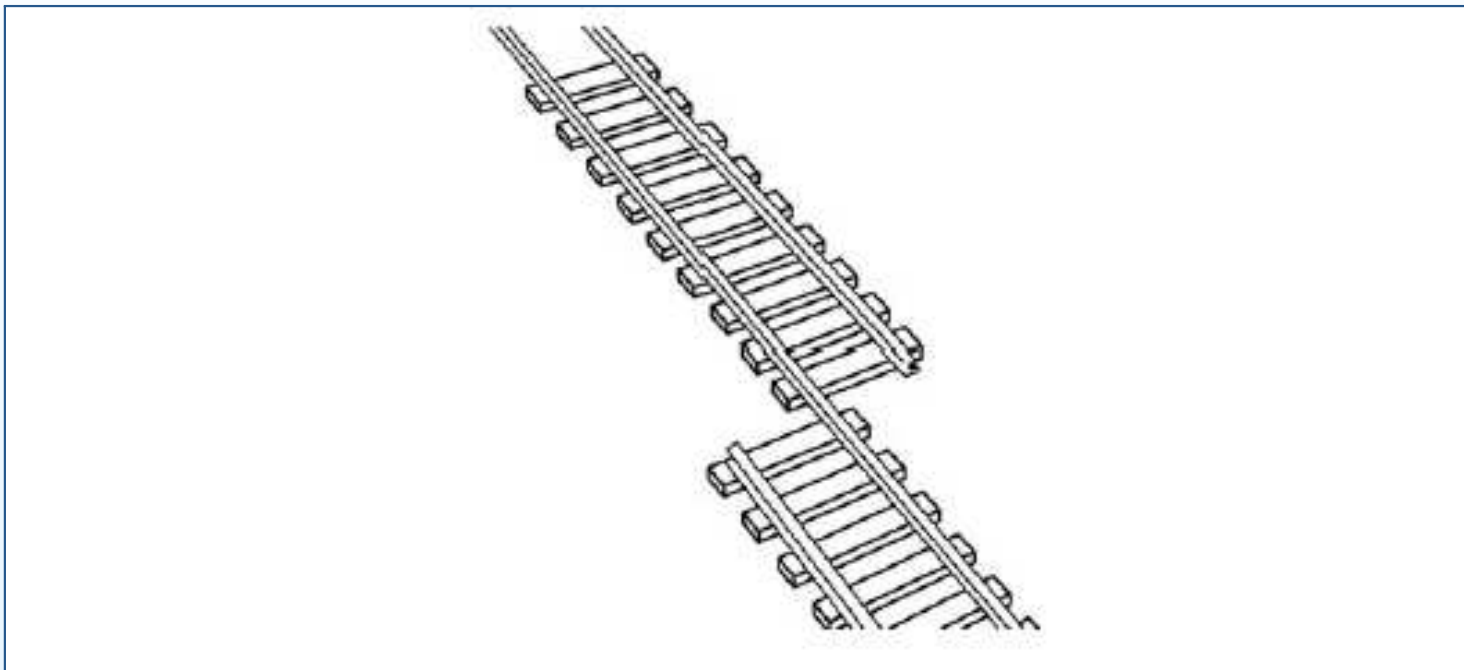
- *Componentă de test* (eng. *test component*) = parte a unui sistem care poate fi izolată pentru testare (obiect/subsistem, grup de obiecte/subsisteme)
- *Defect* (eng. *fault, bug, defect*) = greșeală de proiectare sau codificare ce poate determina un comportament anormal al unei componente
- *Stare de eroare* (eng. *error state*) = manifestare a unui defect în timpul execuției sistemului
- *Eșec* (eng. *failure*) = deviere a comportamentului observat al componentei de la cel specificat
- *Caz de test* (eng. *test case*) = o mulțime de date de intrare și rezultate așteptate, proiectate cu intenția de a provoca eșecul sistemului și a descoperi defecte la nivelul componentelor
- eng. *Test stub* = implementare parțială a unei componente, de care depinde componenta de test
- eng. *Test driver* = implementare parțială a unei componente care depinde de componenta de test (împreună cu stub-urile, permit izolarea unei componente pentru testare)
- *Corectură* (eng. *Correction*) = modificare a unei componente, în scopul de a remedia un defect (poate introduce noi defecte)

Testarea sistemelor soft - concepte (cont.)



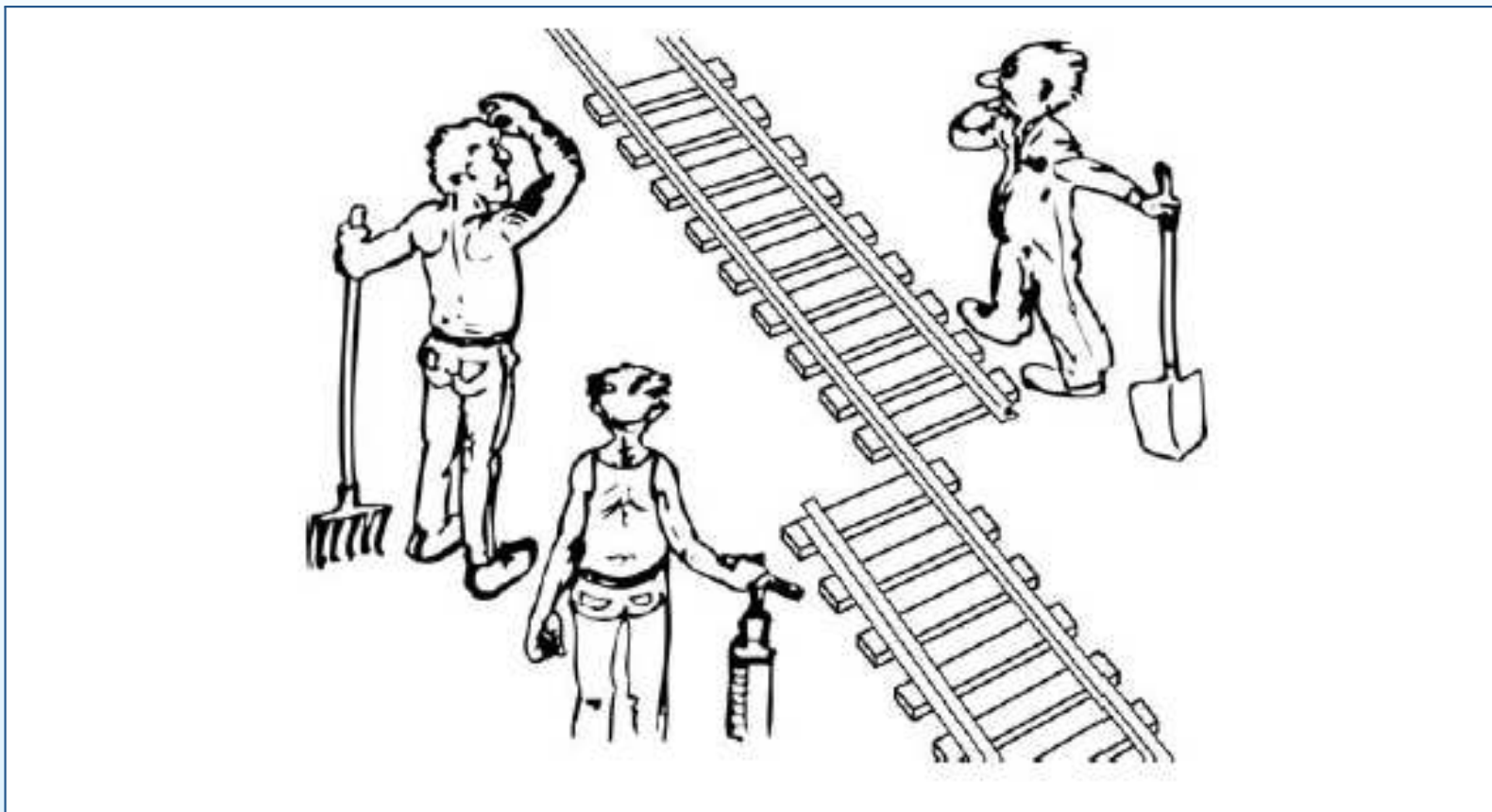
Testarea sistemelor soft - concepte (cont.)

- Eșec, eroare sau defect?



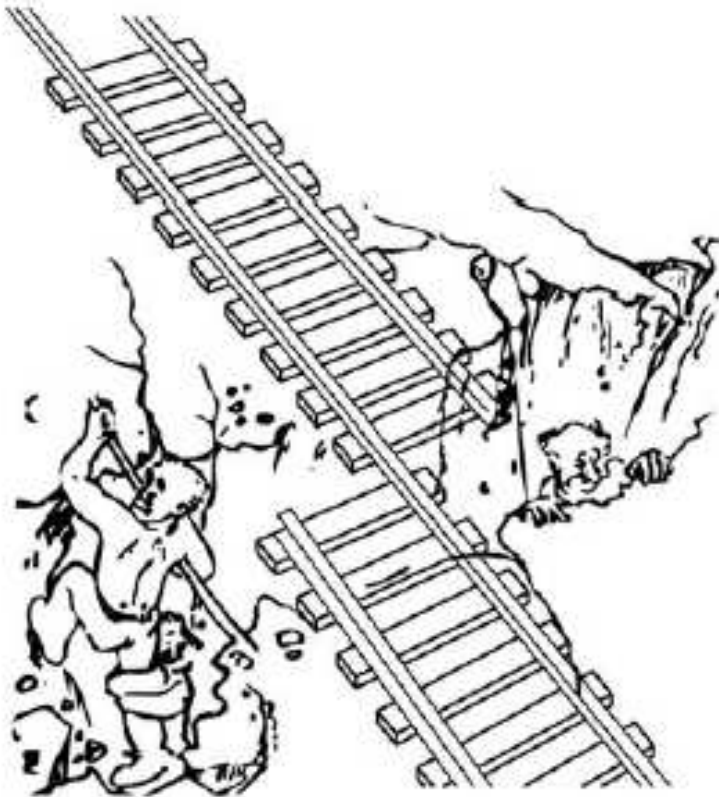
Testarea sistemelor soft - concepte (cont.)

- Defect algoritmic
 - omiterea inițializării unei variabile
 - setarea unei variabile folosite drept index in afara valorilor permise



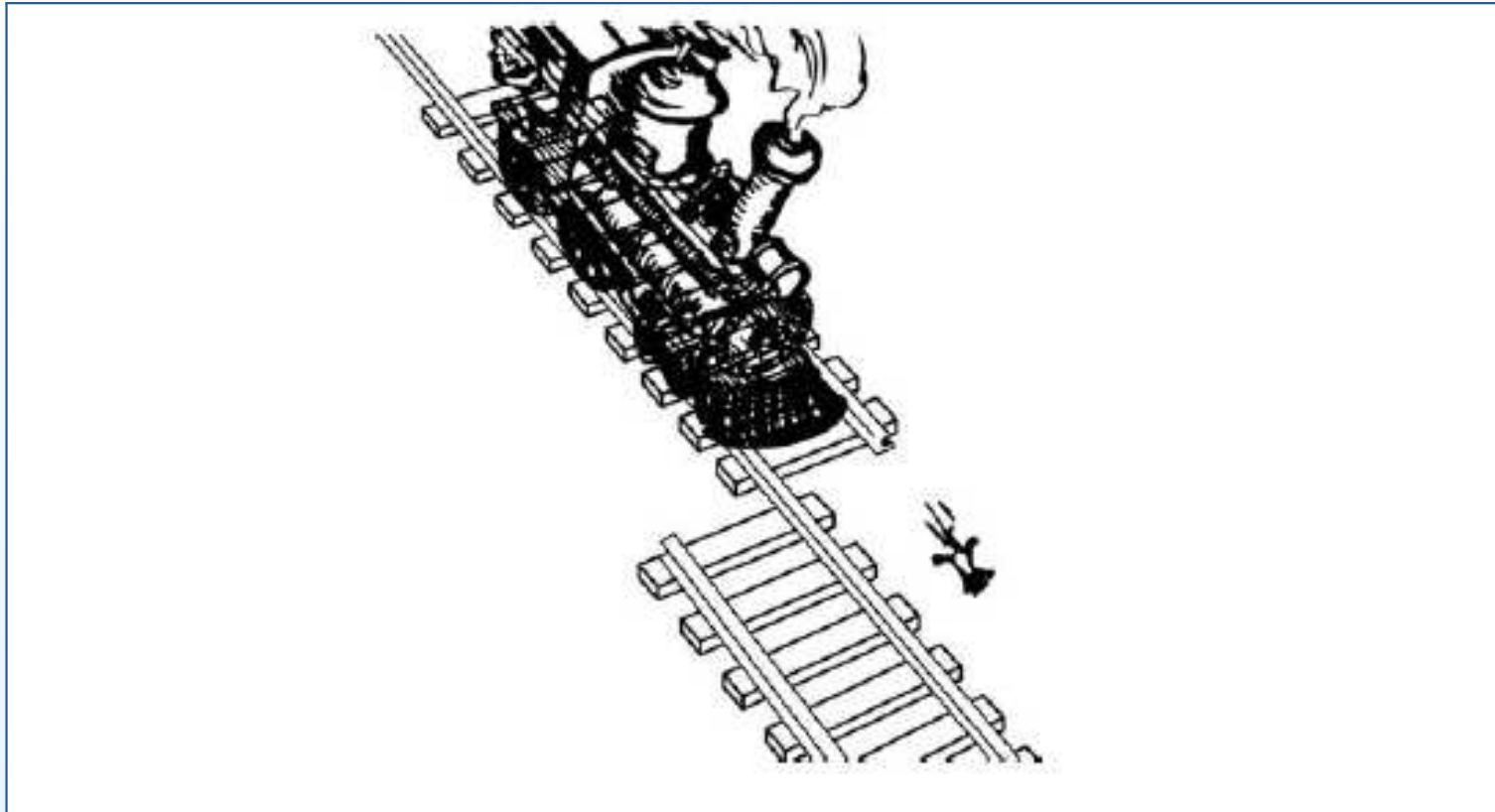
Testarea sistemelor soft - concepte (cont.)

- Defect mecanic
 - defect la nivelul mașinii virtuale
 - pană de curent



Testarea sistemelor soft - concepte (cont.)

- Eroare



Testarea sistemelor soft - activități

- *Planificarea testării* (eng. *test planning*)
 - alocă resurse și programează activitățile de testare
 - această activitate ar trebui să aibă loc devreme în procesul de dezvoltare, astfel încât testării să i se aloce suficient timp și resursă umană calificată (ex. cazurile de test ar trebui proiectate imediat ce modelele aferente devin stabile)
- *Inspectarea componentelor* (eng. *component inspection*)
 - Identifică defecte la nivelul componentelor prin inspectarea manuală a codului sursă al acestora
- *Testarea utilizabilității* (eng. *usability testing*)
 - încearcă să detecteze defecte în proiectarea interfeței utilizator a sistemului
 - uneori, sistemele eșuează din cauză că utilizatorii sunt induși în eroare de interfața grafică și introduc neintenționat date eronate
- *Testarea unitară* (eng. *unit testing*)
 - încearcă să detecteze defecte la nivelul obiectelor sau subsistemelor individuale, raportându-se la specificarea acestora (modelul aferent proiectării obiectuale)

Testarea sistemelor soft - activități (cont.)

- *Testarea de integrare* (eng. *integration testing*)
 - încearcă să identifice defecte prin integrarea diferitor componente, raportându-se la modelul aferent proiectării de sistem
 - *testarea structurală* (eng. *structural testing*) = testare de integrare implicând toate componentele sistemului
- *Testarea de sistem* (eng. *system testing*)
 - testează sistemul integrat în ansamblu
 - *testarea funcțională* (eng. *functional testing*) - realizată de către dezvoltatori, testează sistemul în raport cu modelul său funcțional
 - *testarea performanței* (eng. *performance testing*) - realizată de către dezvoltatori, testează sistemul în raport cu specificarea cerințelor nefuncționale și cu obiectivele adiționale de proiectare
 - *testarea de acceptare și testarea de instalare* (eng. *acceptance testing, installation testing*) - realizate de către clienți în mediul de dezvoltare, respectiv de exploatare a sistemului

Inspectarea componentelor

- Inspectarea identifică defectele unei componente prin analiza codului acesteia, în cadrul unei reuniuni formale
 - Inspecțiile pot avea loc anterior sau ulterior testării unitare
- *Metoda inspectării a lui Fagan* [Fagan, 1976] - primul proces structurat de inspectare
 - Inspecția este condusă de către o echipă de dezvoltatori, incluzând autorul componentei, un moderator și unul sau mai mulți recenzori
 - Pași
 - *Overview* - autorul componentei prezintă, pe scurt, scopul acesteia și obiectivele inspecției
 - *Pregătire* - recenzorii se familiarizează cu codul componentei (fără a se concentra pe identificarea defectelor)
 - *Ședința de inspectare* - unul dintre cei prezenți parafrazează codul sursă al componentei și membrii echipei semnalează probleme cu privire la acesta
 - *Revizuire* - autorul revizuieste codul componentei, conform observațiilor primite
 - *Urmări* - moderatorul verifică varianta revizuită și poate stabili necesitatea de reinspectare

Inspectarea componentelor (cont.)

- Etape critice: pregătirea și ședința de inspectare
- În afara identificării defectelor, recenzorii pot semnala abateri de la standardele de codificare sau ineficiențe
- Eficiența unei inspecții depinde de pregătirea recenzorilor
- *Active Design Review* [Parnas and Weiss, 1985] - proces de inspectare îmbunătățit
 - Elimină ședința de inspectare, recenzorii identifică defecte în faza de pregătire
 - La finalul etapei de pregătire, fiecare recenzor completează un chestionar care atestă gradul de înțelegere a componenetei
 - Autorul colectează feedback asupra componentei în urma unor întâlniri individuale cu fiecare recenzor
- Ambele metode de inspectare s-au dovedit a fi mai eficiente în descoperirea defectelor decât testarea
 - În proiectele critice se recurge atât la inspecții, cât și la testare, întrucât au tendința de a identifica tipuri diferite de erori

Testarea utilizabilității

- Identifică diferențele dintre sistem și așteptările utilizatorilor cu privire la comportamentul acestuia (spre deosebire de celelalte tipuri de testare, nu compară sistemul cu o specificație)
- Tehnica de realizare a testelor privind utilizabilitatea
 - Dezvoltatorii formulează un set de obiective descriind informația pe care se așteaptă să o obțină în urma testelor (ex.: evaluarea layout-ului interfeței grafice, evaluarea impactului pe care timpul de răspuns îl are asupra eficienței utilizatorilor, evaluarea măsurii în care documentația online răspunde nevoilor utilizatorilor)
 - Obiectivele anterioare sunt evaluate într-o serie de experimente în care reprezentanți ai utilizatorilor sunt antrenați să execute anumite sarcini
 - Dezvoltatorii observă participanții și colectează date privind performanțele acestora (timp de îndeplinire a unei sarcini, rata erorilor) și preferințele lor
- Tipuri de teste de utilizabilitate
 - Teste bazate pe scenarii
 - Teste bazate pe prototipuri (verticale sau orizontale)
 - Teste pe baza sistemului real

Testarea utilizabilității (cont.)

- Elemente fundamentale ale testelor de utilizabilitate
 - obiectivele de test
 - reprezentanți ai utilizatorilor
 - mediul de lucru, real sau simulat
 - interogare extensivă a utilizatorilor de către responsabilul cu testele de utilizabilitate
 - colectare și analiză a rezultatelor cantitative și calitative
 - recomandări cu privire la modul de îmbunătățire a sistemului
- Obiective de test uzuale
 - compararea a două stiluri de interacțiune utilizator
 - identificarea celor mai bune/rele funcționalități într-un scenariu/prototip
 - identificarea funcționalităților utile pentru începători/experti
 - identificarea situațiilor care necesită help online, etc.

Testarea unitară

- Se focusează pe componentele elementare ale sistemului soft - obiecte și subsisteme
 - Candidați pentru testarea unitară: toate clasele modelului obiectual și toate subsistemele identificate în proiectarea de sistem
- Avantaje
 - Reducerea complexității activităților de testare, prin focusarea pe componente cu granularitate mică
 - Ușurința identificării și corectării defectelor, ca urmare a numărului mic de componente implicate într-un test
 - Posibilitatea introducerii paralelismului în activitatea de testare (componentele pot fi testate independent și simultan)
- Tehnici de testare unitară
 - *Testarea bazată pe echivalențe (eng. equivalence testing)*
 - *Testarea frontierelor (eng. boundary testing)*
 - *Testarea căilor de execuție (eng. path testing)*
 - *Testarea bazată pe stări (eng. state-based testing)*
 - *Testarea polimorfismului (eng. polymorphism testing)*

Testarea bazată pe echivalențe

- Tehnică de testare blackbox care minimizează numărul de cazuri de test, prin partiționarea intrărilor posibile în clase de echivalență și selectarea unui caz de test pentru fiecare astfel de clasă
 - Se presupune că sistemul se comportă în mod similar pentru toți membrii unei clase de echivalență => testarea comportamentului aferent unei clase de echivalență se poate realiza prin testarea unui singur membru al clasei
- Pași
 - I. Identificarea claselor de echivalență
 - II. Selectarea intrărilor pentru test
- Criterii utilizate în stabilirea claselor de echivalență
 - *Acoperire*: fiecare intrare posibilă trebuie să aparțină uneia dintre clasele de echivalență
 - *Caracter disjunct*: o aceeași intrare nu poate aparține mai multor clase de echivalență
 - *Reprezentare*: Dacă, prin utilizarea ca și intrare a unui anumit membru al unei clase de echivalență, execuția conduce la o stare de eroare, atunci aceeași stare va putea fi detectată utilizând ca și intrare orice alt membru al clasei

Testarea bazată pe echivalențe (cont.)

- Ex.: testarea unei metode care returnează numărul de zile dintr-o lună, date fiind luna și anul (întregi)

```
class MyGregorianCalendar {  
    ...  
    public static int getNumDaysInMonth(int month, int year) {...}  
    ...  
}
```

- I.1 Identificarea claselor de echivalență pentru lună
 - clasa lunilor cu 31 de zile (1,3,5,7,8,10,12)
 - clasa lunilor cu 30 de zile (4,6,9,11)
 - clasa lunilor cu 28/29 de zile (2)
- I.2 Identificarea claselor de echivalență pentru an
 - clasa anilor bisecți
 - clasa anilor non-bisecți
- Valori invalide
 - pentru lună: < 1 , > 12
 - pentru an: < 0

Testarea bazată pe echivalențe (cont.)

- II. Selectarea reprezentanților pentru test
 - II.1 Pentru lună: 2 (February), 6 (June), 7 (July)
 - II.2 Pentru an: 1904, 1901
 - => 6 clase de echivalență prin combinație

Equivalence class	Value for month input	Value for year input
Months with 31 days, non-leap years	7 (July)	1901
Months with 31 days, leap years	7 (July)	1904
Months with 30 days, non-leap years	6 (June)	1901
Month with 30 days, leap year	6 (June)	1904
Month with 28 or 29 days, non-leap year	2 (February)	1901
Month with 28 or 29 days, leap year	2 (February)	1904

Testarea frontierelor

- Caz particular al metodei de testare bazată pe echivalențe, focalizată pe explorarea cazurilor limită
 - În loc să se aleagă un reprezentant arbitrar al clasei de echivalență pentru testare, metoda cere alegerea unui element aflat "la limită" (caz particular)
 - Presupunerea care stă la baza acestui tip de testare este aceea că dezvoltatorii omit adesea cazurile speciale ("frontierele" claselor de echivalență) (ex.: 0, stringuri vide, anul 2000, etc.)
- Ex.: pentru exemplul considerat anterior
 - Luna 2 (February) și anii 1900 și 2000 (un an multiplu de 100 nu este bisect decât dacă este și multiplu de 400)
 - Lunile 0 și 13, aflate la limita clasei de echivalență conținând lunile invalide

Equivalence class	Value for month input	Value for year input
Leap years divisible by 400	2 (February)	2000
Non-leap years divisible by 100	2 (February)	1900
Nonpositive invalid months	0	1291
Positive invalid months	13	1315

Testarea căilor de execuție

- Metodă de tip whitebox, care identifică defecte în implementarea unei componente, prin testarea tuturor căilor de execuție din cod
 - Presupunerea din spatele acestei tehnici este aceea că, prin execuția, cel puțin o dată, a fiecărei căi (eng. *path*) din cod, majoritatea defectelor vor genera eșecuri
 - Punctul de start în aplicarea acestei metode îl constituie construirea unei reprezentări de tip schemă logică (eng. *flow graph*) asociate codului testat
 - nodurile corespund instrucțiunilor
 - arcele corespund fluxului de control
 - Ex.: implementare greșită a metodei *getNumDaysInMonth()*

```
public class MonthOutOfBounds extends Exception {...};
public class YearOutOfBounds extends Exception {...};

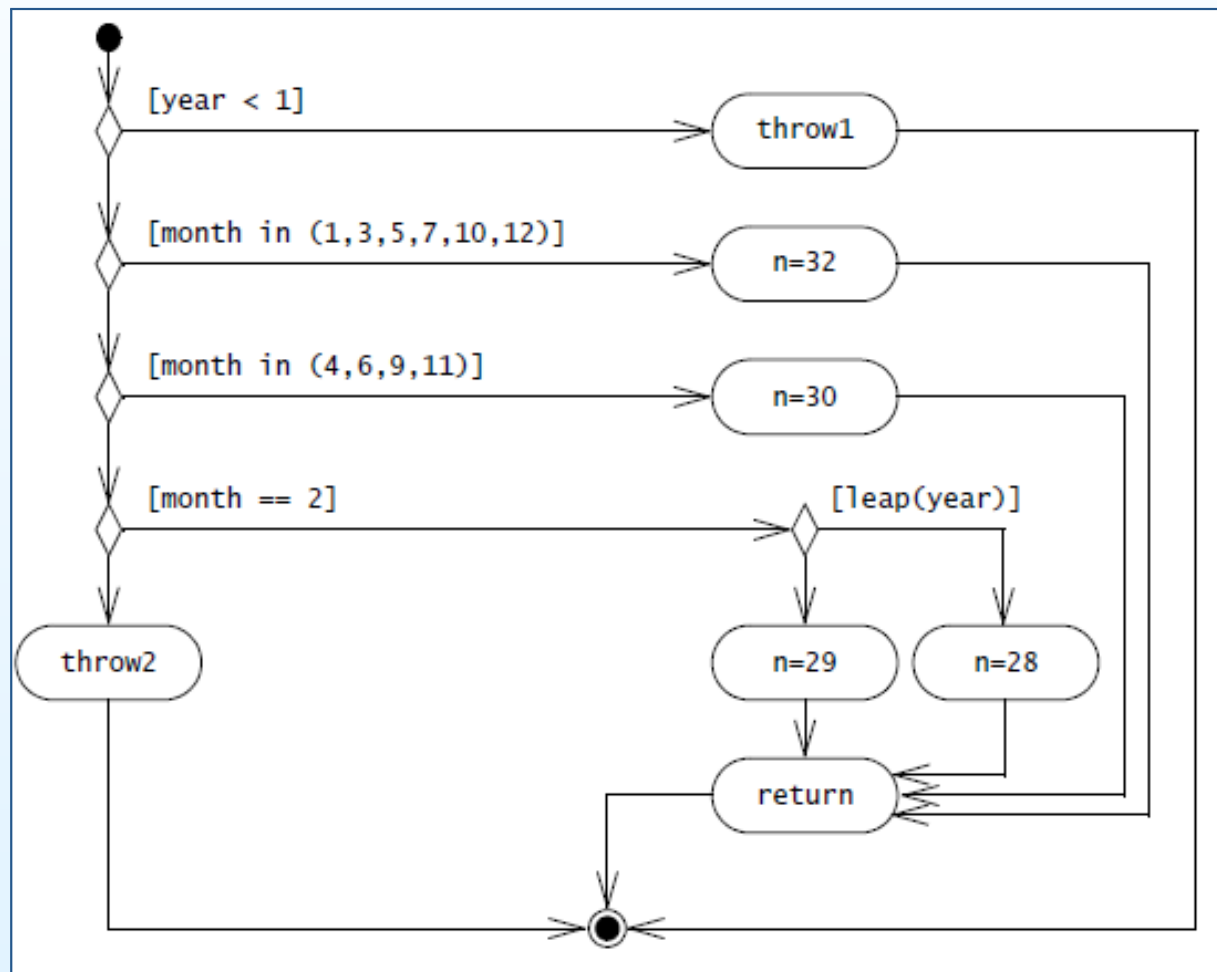
class MyGregorianCalendar {
    public static boolean isLeapYear(int year) {
        boolean leap;
        if ((year%4) == 0){
            leap = true;
        } else {
            leap = false;
        }
        return leap;
    }
}
```

Testarea căilor de execuție (cont.)

```
public static int getNumDaysInMonth(int month, int year)
    throws MonthOutOfBounds, YearOutOfBounds {
    int numDays;
    if (year < 1) {
        throw new YearOutOfBounds(year);
    }
    if (month == 1 || month == 3 || month == 5 || month == 7 ||
        month == 10 || month == 12) {
        numDays = 32;
    } else if (month == 4 || month == 6 || month == 9 || month == 11) {
        numDays = 30;
    } else if (month == 2) {
        if (isLeapYear(year)) {
            numDays = 29;
        } else {
            numDays = 28;
        }
    } else {
        throw new MonthOutOfBounds(month);
    }
    return numDays;
}
```

Testarea căilor de execuție (cont.)

- Schema logică aferentă implementării (greșite a) metodei *getNumDaysInMonth()* (diagramă UML de activități)



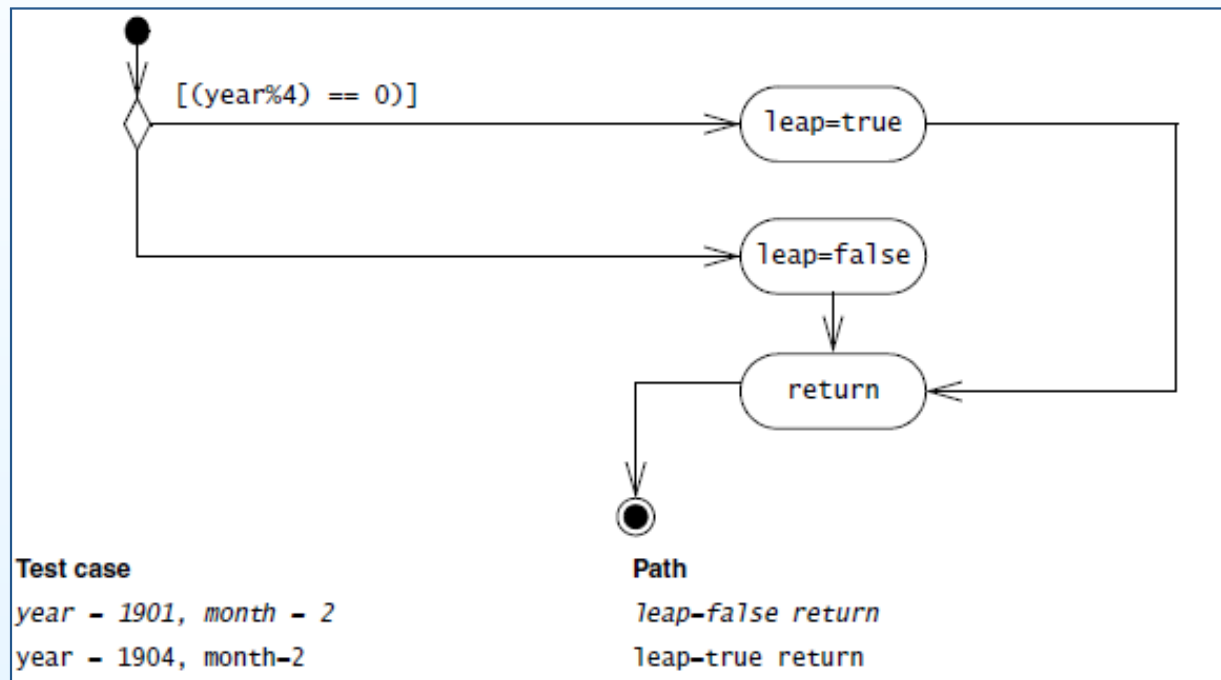
Testarea căilor de execuție (cont.)

- Testarea completă a căilor de execuție presupune proiectarea cazurilor de test astfel încât fiecare arc al diagramei de activități să fie traversat cel puțin o dată
 - Aceasta presupune analiza fiecărui nod decizional și selectarea câte unei intrări pentru fiecare dintre ramurile *true* și *false*
 - combinația (1,1901) identifică defectul $n=32$
 - Ex.: cazurile de test generate pentru metoda *getNumDaysInMonth()*

Test case	Path
(year - 0, month - 1)	{throw1}
(year - 1901, month - 1)	{n-32 return}
(year - 1901, month - 2)	{n-28 return}
(year - 1904, month - 2)	{n-29 return}
(year - 1901, month - 4)	{n-30 return}
(year - 1901, month - 0)	{throw2}

Testarea căilor de execuție (cont.)

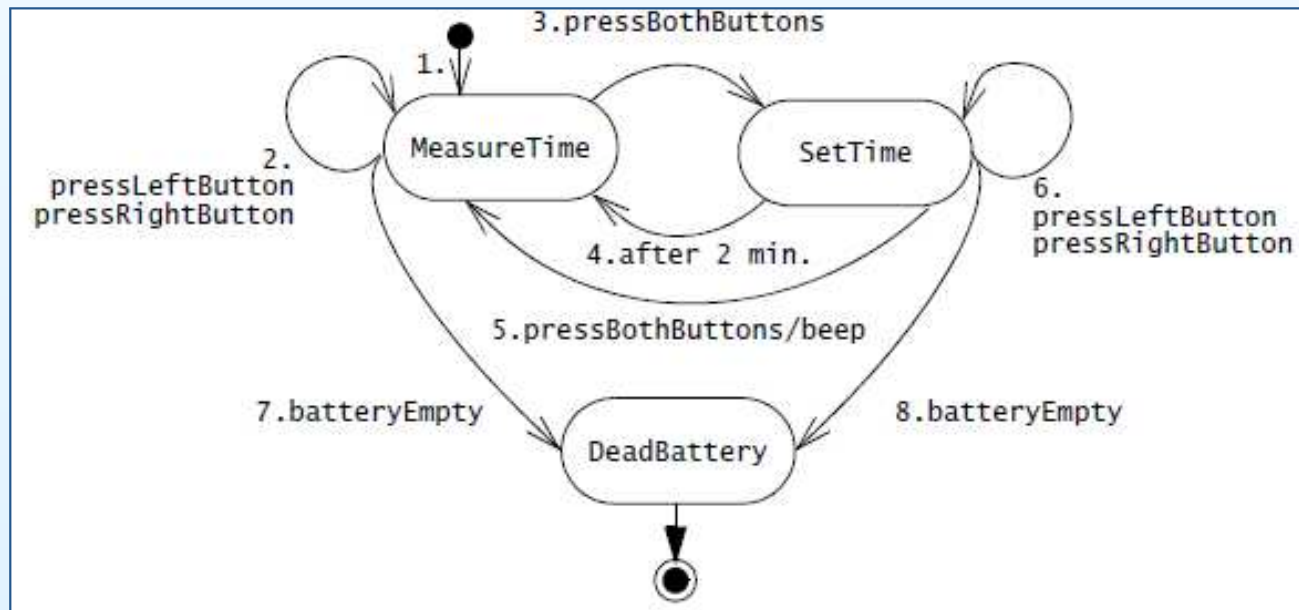
- Ex.: diagrama de activități și cazurile de test generate pentru metoda *isLeapYear()*



- Testarea căilor de execuție vs. testarea bazată pe echivalențe / a frontierelor
 - ambele detectează defectul $n=32$
 - prima e posibil să nu genereze caz de test aferent anilor multiplu de 100
 - e posibil ca nici unul dintre cazurile de test generate de cele două metode să nu identifice eroarea asociată lunii 8

Testarea bazată pe stări

- Tehnică de testare a sistemelor orientate obiect, care generează cazuri de test pentru o clasă pe baza diagramei UML de tranziție a stărilor asociată respectivei clase
 - Pentru fiecare stare, se stabilește un set reprezentativ de stimuli aferenți tranzițiilor posibile din acea stare (similar testării bazate pe echivalențe)
 - După aplicarea fiecărui stimul, se compară starea curentă a componentei cu cea indicată de diagramă, indicându-se eșec în caz de neconcordanță
- Ex.: diagramă de tranziție a stărilor aferentă clasei *2Bwatch*



Testarea bazată pe stări (cont.)

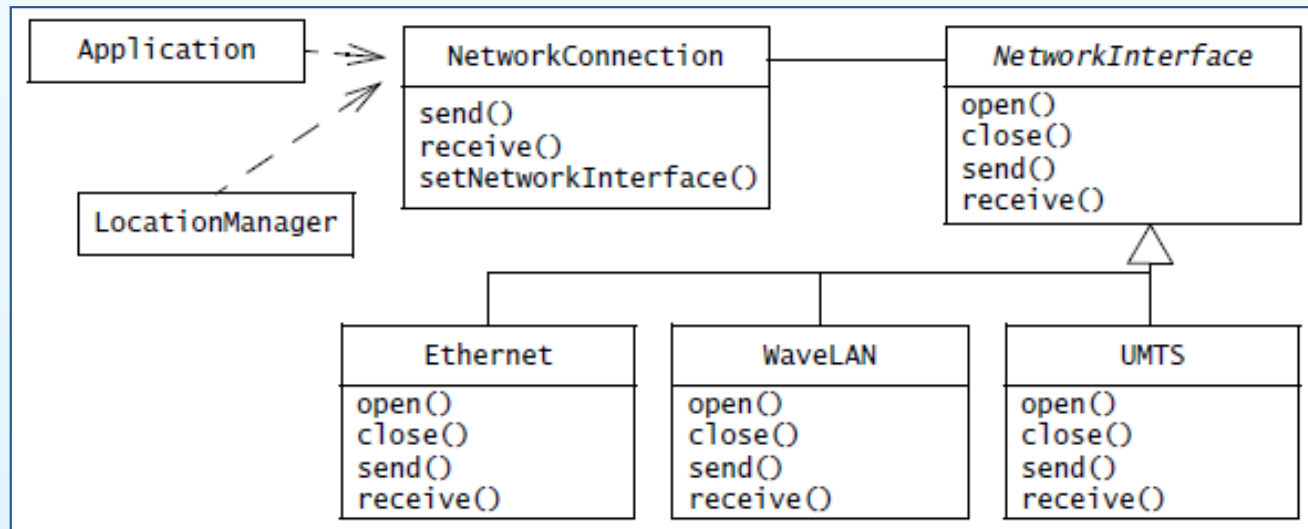
- Ex.: cazuri de test generate pentru sistemul *2Bwatch* (a.î. fiecare tranziție, exceptând 7 și 8, să fie traversată cel puțin o dată)

Stimuli	Transition tested	Predicted resulting state
Empty set	1. <i>Initial transition</i>	MeasureTime
Press left button	2.	MeasureTime
Press both buttons simultaneously	3.	SetTime
Wait 2 minutes	4. <i>Timeout</i>	MeasureTime
Press both buttons simultaneously	3. <i>Put the system into the SetTime state to test the next transition.</i>	SetTime
Press both buttons simultaneously	5.	SetTime->MeasureTime
Press both buttons simultaneously	3. <i>Put the system into the SetTime state to test the next transition.</i>	SetTime
Press left button	6. Loop back onto MeasureTime	MeasureTime SetTime

- Avantaje/dezavantaje ale testării bazate pe stări
 - Starea fiind încapsulată, cazurile de test trebuie să includă aplicarea unor secvențe de stimuli care aduc componenta în starea dorită, înainte de a putea testa o anumită tranziție
 - + Potențial de automatizare

Testarea polimorfismului

- Polimorfismul introduce o nouă provocare în procesul de testare, prin faptul că permite ca un același mesaj să se concretizeze în apeluri de metode diferite, funcție de tipul actual al apelatului
 - Atunci când se realizează testarea căilor de execuție pentru o metodă ce utilizează polimorfism, este necesar să se ia în calcul toate legăturile posibile => necesitatea de a expanda metoda pentru a aplica algoritmul clasic de test
 - Ex.: aplicare a șablonului *Strategy* pentru a încapsula diferite implementări *NetworkInterface*



Testarea polimorfismului (cont.)

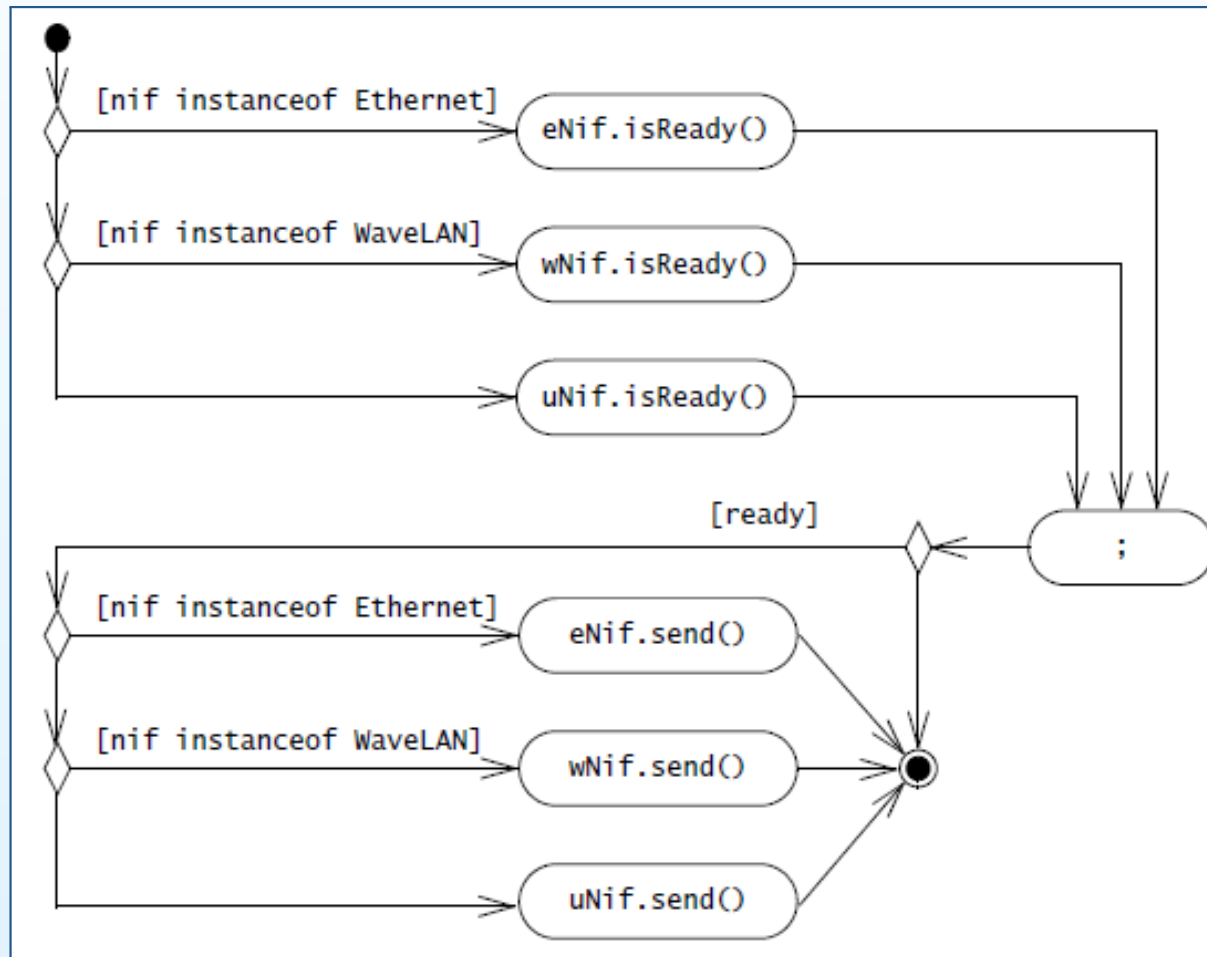
- Ex.: Codul sursă al metodei *NetworkConnection.send()*, cu și fără polimorfism (ultima variantă este cea folosită pentru generarea cazurilor de test)

```
public class NetworkConnection {  
    //...  
    private NetworkInterface nif;  
    void send(byte msg[]) {  
        queue.concat(msg);  
        if (nif.isReady()) {  
            nif.send(queue);  
            queue.setLength(0);  
        }  
    }  
}
```

```
public class NetworkConnection {  
    //...  
    private NetworkInterface nif;  
    void send(byte msg[]) {  
        queue.concat(msg);  
        boolean ready = false;  
        if (nif instanceof Ethernet) {  
            Ethernet eNif = (Ethernet)nif;  
            ready = eNif.isReady();  
        } else if (nif instanceof WaveLAN) {  
            WaveLAN wNif = (WaveLAN)nif;  
            ready = wNif.isReady();  
        } else if (nif instanceof UMTS) {  
            UMTS uNif = (UMTS)nif;  
            ready = uNif.isReady();  
        }  
        if (ready) {  
            if (nif instanceof Ethernet) {  
                Ethernet eNif = (Ethernet)nif;  
                eNif.send(queue);  
            } else if (nif instanceof WaveLAN) {  
                WaveLAN wNif = (WaveLAN)nif;  
                wNif.send(queue);  
            } else if (nif instanceof UMTS) {  
                UMTS uNif = (UMTS)nif;  
                uNif.send(queue);  
            }  
            queue.setLength(0);  
        }  
    }  
}
```

Testarea polimorfismului (cont.)

- Ex.: Diagrama de activități aferentă variantei expandate a codului sursă al metodei *NetworkConnection.send()* (generarea cazurilor de test se face după metoda prezentată la "Testarea căilor de execuție")



Referințe

- [Fagan, 1976] M. E. Fagan, *Design and code inspections to reduce errors in program development*, IBM Systems Journal, Vol. 15, No. 3, 1976.
- [Parnas and Weiss, 1985] D. L. Parnas and D. M. Weiss, *Active design reviews: principles and practice*, Proceedings of the Eighth International Conference on Software Engineering, London, U.K., pp 132-136, August 1985.