Universitatea din București

Metode de dezvoltare software

Depanarea programelor

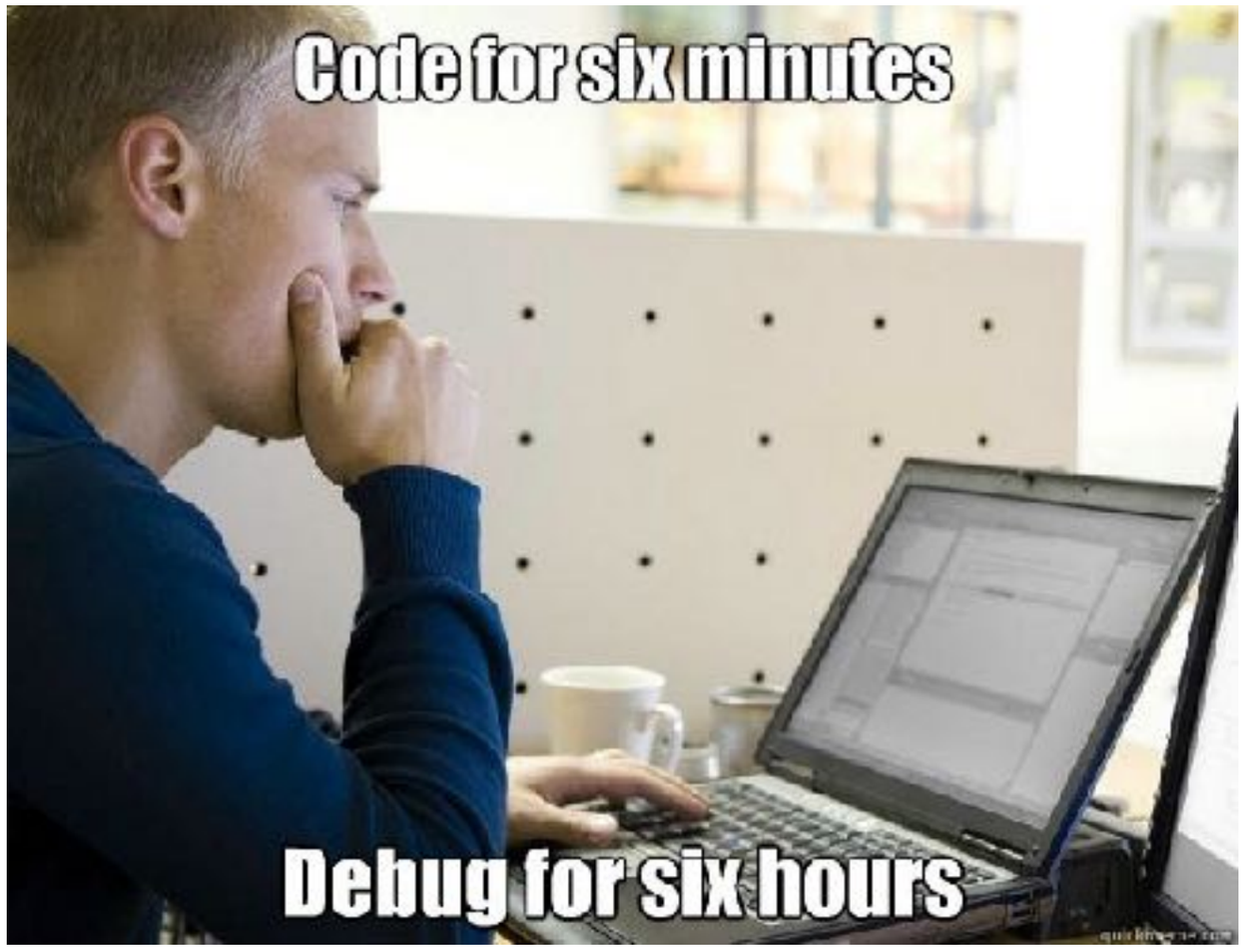
03.04.2017

Alin Ștefănescu

Depanarea programelor

Prezentare bazată pe materiale de
Moa Johansson (Univ. Chalmers) și Peter Thiemann (Univ. Freiburg)

Depanarea… în practică



Depanarea (debugging)

- Pentru **testare**:
  - se caută anumite date de intrare care să genereze un “comportament anormal”
  - un alt scop este și acela de a obține acoperiri cât mai bune
  - ce facem însă în momentul în care observăm într-adevăr o defecțiune (“failure”) în program?

- **Depanarea** programului (debugging)**:**
  - căutarea sursei defectului/problemei detectate
  - folosind de obicei un tool numit “debugger”
  - vom vedea și câteva moduri sistematice de depanare (dependențe în program etc.)

Depanare folosind tipărirea

- varianta cea mai simplă (dar și cea mai simplistă): depanarea de tip “println” (urmărirea diverselor aspecte tipărind în output diverse valori ale variabilelor).
  - d.ex. *System.out.println(“size = "* + *size);*
  - simplu de aplicat; nu necesită alte tool-uri
  - dezavantaje: codul se complică, outputul se complică, performanța uneori scade, ai nevoie de recompilări repetate, excepțiile nu pot fi controlate ușor etc.

Depanare folosind log-urile

- pentru depanare se poate folosi și log-ul programului (istoria execuției programului), dacă e disponibil:
  - în Java: *java.util.logging*
  - fiecare clasă are asociat un obiect *Logger,* care are asociat un nivel (*finest, fine, …, info, warning, severe*) și un handler
  - log-ul poate fi controlat prin program sau proprietăți
  - rezolvă multe din problemele tipului anterior de depanare
  - dezavantaje: codul se complică
  - … altă soluție: un tool dedicat numit “debugger”

Ce este un “debugger”

Funcționalitățile de bază ale unui **debugger** (instrumentul care ne ajută să identificăm problema sau defectul în cod) sunt:

- **controlul execuției**: poate opri execuția la anumite locații numite **breakpoints**
- **interpretorul:** poate executa instrucțiunile **una câte una**
- **inspecția stării programului:** poate **observa** valoarea variabilelor, obiectelor sau a stivei de execuție
- **schimbarea stării**: poate **schimba** starea programului în timpul execuției

Debugger-ul Java în Eclipse

În continuare vom vedea cum funcționează **depanarea** pentru Java în framework-ul **Eclipse**:

- instrucțiuni de instalare și folosire în Eclipse cu Java + JUnit + depanare pot fi consultate aici:
  - http://www.vogella.com/tutorials/Eclipse/article.html
  - http://www.vogella.com/tutorials/JUnit/article.html
  - http://www.vogella.com/tutorials/EclipseDebugging/article.html
- vom demonstra capabilitățile de depanare pentru un mic program

Un mic exemplu

```
public class BinarySearch {
  // presupunem ca vectorul array e ordonat crescator!
  public static int binarySearch(int array[], int target){

    int low = 0;
    int high = array.length;
    int mid;
    while (low <= high) {
        mid = (low + high)/2;
        if ( target < array[ mid ] )
           high = mid - 1;
        else if ( target > array[ mid ])
           low = mid + 1;
        else
           return mid;
    }
    return -1;
  }
}
```

Testare

Rulăm câteva teste pentru funcția de căutare:

- binarySearch({1,2,3}, 1) == 0 **OK**
- binarySearch({1,2,3}, 2) == 1 **OK**
- binarySearch({1,2,3}, 3) == 2 **OK**
- binarySearch({1,2,3}, -1) == -1 **OK**
- binarySearch({1,2,3}, 4) aruncă `ArrayIndexOutOfBoundsException`
- binarySearch({}, 0) aruncă `ArrayIndexOutOfBoundsException`

Testele implementate în JUnit

```
public class BinarySearchTest {

@Test public void test_1() {
   int[] a = {1, 2, 3};
   int x = 1;
   int result = BinarySearch.binarySearch(a,x);
   assertTrue(result == 0);
}

@Test public void test_2() {
   int[] a = {1, 2, 3};
   int x = 2;
   int result = BinarySearch.binarySearch(a,x);
   assertTrue(result == 1);
}

@Test public void test_3() {
   int[] a = {1, 2, 3};
   int x = 3;
   int result = BinarySearch.binarySearch(a,x);
   assertTrue(result == 2);
}
```

```
@Test public void test_4() {
   int[] a = {1, 2, 3};
   int x = -1;
   int result = BinarySearch.binarySearch(a,x);
   assertTrue(result == -1);
}

@Test public void test_5() {
   int[] a = {1, 2, 3};
   int x = 4;
   int result = BinarySearch.binarySearch(a,x);
        assertTrue(result == -1);
}

@Test public void test_6() {
   int[] a = {};
   int x = 0;
   int result = BinarySearch.binarySearch(a,x);
        assertTrue(result == -1);
}
```

Debug – myFirstProject/src/myFirstProject/BinarySearch.java – Eclipse – /Users/alin/Desktop/local/eclipse/workspace

Quick Access | Java | Debug

Debug

```
/System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home/bi
BinarySearch [Java Application]
  myFirstProject.BinarySearch at localhost:61939
    Thread [main] (Suspended)
      BinarySearch.binarySearch(int[], int) line: 12
      BinarySearch.main(String[]) line: 25
/System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home/bi
BinarySearch [Java Application]
  myFirstProject.BinarySearch at localhost:62173
    Thread [main] (Suspended (breakpoint at line 12 in BinarySearch))
      BinarySearch.binarySearch(int[], int) line: 12
      BinarySearch.main(String[]) line: 25
```

Variables | Breakpoints

Name	Value
array	(id=15)
target	4
low	3
high	3
mid	3

BinarySearch.java | BinarySearchTest.java

```
8           int mid;
9
10          while (low <= high) {
11              mid = (low + high)/2;
12              if ( target < array[ mid ] )
13                  high = mid - 1;
14              else if ( target > array[ mid ])
15                  low = mid + 1;
16              else
17                  return mid;
```

Outline

myFirstProject
BinarySearch
binarySearch(int[], int) : int
main(String[]) : void

Console | Tasks | JUnit

Finished after 0.033 seconds

Runs: 6/6 | Errors: 2 | Failures: 0

test_2 (0.001 s)
test_3 (0.000 s)

Failure Trace
java.lang.ArrayIndexOutOfBoundsException: 3

Writable | Smart Insert | 4 : 9

Breakpoints

Un **breakpoint** este o locație în program care, atunci când este atinsă, oprește execuția.

- d.ex. în Eclipse, un breakpoint e setat prin: buton-dreapta și apoi “toggle breakpoint”

- strategie: se pune un breakpoint la ultima linie unde știm că starea e corectă.

- după ce nu mai avem nevoie de un breakpoint, îl dezactivăm

Execuție pas cu pas

Execuția pas cu pas:

- **“step into”** execută instrucțiunea următoare, apoi se oprește
- **“step over”** consideră un apel de metodă ca o instrucțiune
- de obicei metodele din biblioticile Java sunt “sărite” (ele sunt probabil corecte și de multe ori nu avem codul lor sursă)
- pentru a sări peste bucăți mari de cod, se folosesc breakpointuri (folosindu-se comanda “resume”)

Inspecția stării programului

Când execuția este oprită, se poate examina **starea programului**

- d.ex. în Eclipse, se poate folosi fereastra “Variables” unde diverse facilități sunt oferite (variabilele recent schimbate sunt evidențiate, se pot urmări anumite expresii sau variabile cu optiunea “watch” etc.)

- pentru exemplul nostru:
  - putem să oprim execuția înainte de ciclul *while*
  - executăm pas cu pas
  - la a treia execuție a ciclului, *low*==*high*==*3*
  - apoi *mid*==*3*, dar *array[3]* nu există
  - observăm că dacă *target* e mai mare decât toate elementele în *array*, la final avem că *low*==*high*==*array.length*

Schimbarea stării programului

Ipoteza valorii corecte: variabila *high* ar trebui să aibă valoarea *array.length-1*

- d.ex. în Eclipse, se poate schimba starea în timp ce programul este oprit în timpul depanării.

- buton-dreapta pe *high* apoi “Change Value”

- pentru exemplul nostru după al doilea ciclu, setăm *high* la valoarea 2. Continuând execuția obținem rezultatul corect.

Urmărirea anumitor stări

Se poate folosi o expresie booleană pentru a condiționa un breakpoint

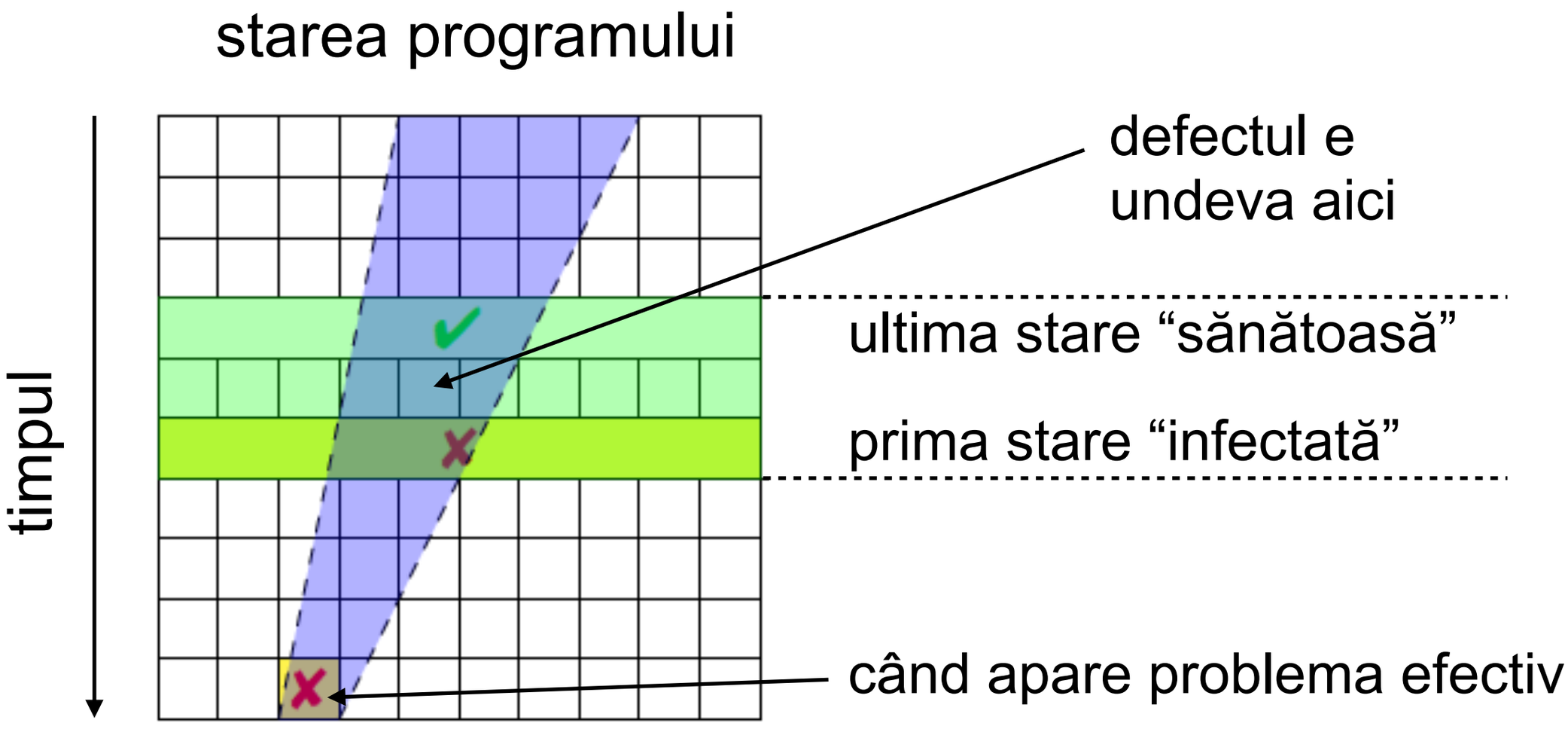
- d.ex. în Eclipse, se selectează un breakpoint și se adaugă condiția
- programul se va opri la acel breakpoint, doar dacă și condiția respectivă este adevărată
- în programul nostru, putem să punem un breakpoint la prima instrucțiune după asignarea lui *mid* și să adăugăm condiția *mid == 3*

Depanare sistematică

Motivație

- Depanarea trebuie realizată într-un mod sistematic deoarece:
  - datele asociate unei probleme pot fi mari
  - programele pot avea mii de locații de memorie
  - programele pot trece prin milioane de stări înainte de a se manifesta problema

Principalii pași în depanarea sistematică



- stările sănătoase trebuie separate de cele infectate
- părțile relevante trebuie separate de cele irelevante

Urmărirea cauzelor și a efectelor

- Dificultatea principală este aceea de a găsi sursa problemei apărute
- În exemplul nostru, problema apărea datorită lui *mid* in *array[mid],* dar problema era generată de inițializarea lui *high*
- Problema fundamentală este faptul că un program este executat “înainte”, pe când noi trebuie să raționăm “înapoi” pentru a descoperi defectul

În exemplul nostru

```
public static int binarySearch(int array[], int target){

  int low = 0;
  int high = array.length ;
  int mid;
  while (low <= high) {
      mid = (low + high)/2;
      if ( target < array[ mid ] )
         high = mid - 1;
      else if ( target > array[ mid ] )
         low = mid + 1;
      else
         return mid;
   }
   return -1;
}
```

Efectele instrucțiunilor

Cum se pot **influența** instrucțiunile între ele

- **scriere** (write): schimbă starea programului; asignează o nouă valoare unei variabile citite în altă parte
  - exemple: asignări; operațiuni in/out; new()
- **control**: schimbă “contorul” programului, determinând care instrucțiune va fi executată la pasul următor
  - condiții, switch-uri
  - instrucțiuni repetitive (*while, for*)
  - apeluri dinamice de metode (prin reflecție)
  - terminare abruptă (break, return, continue)
  - excepții (posibile la orice obiect sau vector)

Dependențe

Dependență de date: instrucțiunea *B* depinde cu ajutorul datelor (***data-dependent***) de instrucțiunea *A* dacă, prin definiție:

1. *A* modifică o variabilă *v* citită de *B* **și**
2. există cel puțin o cale de execuție între *A* și B în care *v* nu este modificată

"Rezultatul lui *A* influențează direct o variabilă citită de *B*"

Dependență de control: instrucțiunea *B* depinde prin control (***control-dependent***) de instrucțiunea *A* dacă, prin definiție:

1. execuția lui *B* poate fi (potențial) controlată de *A*

"Rezultatul lui *A* poate influența faptul că *B* este executată sau nu"

În exemplul nostru

```
public static int binarySearch(int array[], int target){

  int low = 0;
  int high = array.length;
  int mid;
  while (low <= high) {
     mid = (low + high)/2 ;
     if ( target < array[ mid ] )
        high = mid - 1;
     else if ( target > array[ mid ] )
        low = mid + 1;
     else
        return mid;
  }
  return -1;
}
```

`mid` este *data-dependent* de instrucțiunea pe fond galben

În exemplul nostru

```
public static int binarySearch(int array[], int target){

  int low = 0;
  int high = array.length;
  int mid;
  while (low <= high) {
      mid = (low + high)/2 ;
      if ( target < array[ mid ] )
         high = mid - 1;
      else if ( target > array[ mid ] )
         low = mid + 1;
      else
         return mid;
   }
   return -1;
}
```

`mid` este *control-dependent* de instrucțiunea `while`

Dependențe în sens invers

Dependență "înapoi": instrucțiunea *B* **depinde în sens invers** (***backward-dependent***) de instrucțiunea *A* dacă, prin definiție:

- există o secvență de instrucțiuni $A = A_1, A_2, \ldots, A_n = B$ astfel încât:
  1. pentru toți indicii *i*, A_{i+1} este *control-dependent* sau *data-dependent* de A_i și
  2. există cel puțin un indice *i* cu A_{i+1} *data-dependent* de A_i.

"Rezultatul lui *A* poate influența starea programului în *B*"

În exemplul nostru

```
int low = 0;
int high = array.length;
int mid;
while (low <= high) {
    mid = (low + high)/2;
    if ( target < array[ mid ] )
       high = mid - 1;
    else if ( target > array[ mid ] )
       low = mid + 1;
    else
       return mid;
}
return -1;
```

`mid` este *backward-dependent* de instrucțiunile evidențiate la **prima** execuție a corpului ciclului while

În exemplul nostru

```
int low = 0;
int high = array.length;
int mid;
while (low <= high) {
    mid = (low + high)/2;
    if ( target < array[ mid ] )
       high = mid - 1;
    else if ( target > array[ mid ] )
       low = mid + 1;
    else
       return mid;
}
return -1;
```

`mid` este *backward-dependent* de instrucțiunile evidențiate după o execuție **repetată** a corpului ciclului while

Urmărirea defectelor

Algoritm de localizarea sistematică a defectelor

În *I* vom păstra un set de locații infectate (variabilă + contor de program)

În *L* păstrăm locația curentă într-o **execuție care a eșuat**

1. Fie *L* locația infectată raportată de eșec și *I* := {*L*}
2. Calculăm setul de instrucțiuni *S* care ar putea conține originea defectului: un nivel de dependență “înapoi” din *L* pe calea de execuție
3. Inspectăm locațiile $L_1, \ldots, L_n$ scrise în S și dintre ele alegem într-o mulțime $M \subseteq \{ L_1, \ldots, L_n \}$ pe cele infectate
4. În cazul în care $M \neq \emptyset$ (adică cel puțin un L_i este infectat):

   4.1 Fie $I := (I \setminus \{L\}) \cup M$ (înlocuim *L* cu noii candidați din *M*)

   4.2 Alegem ca nouă locație *L,* o locație aleatoare din *I*

   4.3 Ne întoarcem la pasul 2.
5. *L* depinde doar de locații neinfectate, deci aici este locul de infectare!

În exemplul nostru

```
int low = 0 ;
int high = array.length ;
int mid;
while (low <= high) {
    mid = (low + high)/2 ;
    if ( target < array[ mid ] )
       high = mid – 1 ;
    else if ( target > array[ mid ] )
       low = mid + 1 ;
    else
       return mid;
}
return –1;
```

Pentru intrarea **({1,2}, 3)**, `mid` este infectat și avem
mid == low == high == 2

În exemplul nostru

```
int low = 0 ;
int high = array.length ;
int mid;
while (low <= high) {
   mid = (low + high)/2 ;
    if ( target < array[ mid ] )
       high = mid – 1 ;
    else if ( target > array[ mid ] )
       low = mid + 1 ;
    else
       return mid;
}
return –1;
```

Căutăm originile lui *low* și *high*

În exemplul nostru

```
int low = 0 ;
int high = array.length ;
int mid;
while (low <= high) {
    mid = (low + high)/2 ;
     if ( target < array[ mid ] )
        high = mid – 1 ;
    else if ( target > array[ mid ] )
       low = mid + 1 ;
    else
       return mid;
}
return –1;
```

low a fost schimbat în execuția anterioară a ciclului;
dar valuarea sa, *low == 1,* pare în regula

În exemplul nostru

```
int low = 0 ;
int high = array.length ;
int mid;
while (low <= high) {
    mid = (low + high)/2 ;
     if ( target < array[ mid ] )
        high = mid – 1 ;
    else if ( target > array[ mid ] )
       low = mid + 1 ;
    else
       return mid;
}
return –1;
```

high == *2 e* setat la început și a rămas neschimbat (prima ramură *if* unde se poate schimba *high* nu a fost aleasă) și valoarea 2 nu e în regulă

În exemplul nostru

```
int low = 0 ;
int high = array.length ;
int mid;
while (low <= high) {
    mid = (low + high)/2 ;
     if ( target < array[ mid ] )
        high = mid – 1 ;
    else if ( target > array[ mid ] )
        low = mid + 1 ;
    else
        return mid;
}
return –1;
```

high nu depinde de alte locații, deci am găsit defectul!

În exemplul nostru

```
int low = 0 ;
int high = array.length - 1 ;
int mid;
while (low <= high) {
    mid = (low + high)/2 ;
    if ( target < array[ mid ] )
       high = mid - 1 ;
    else if ( target > array[ mid ])
       low = mid + 1 ;
    else
       return mid;
}
return -1;
```

Defectul rezolvat (testele rulează și ele corect acum)

Retestare

După ce corectăm problema găsită, trebuie să testăm

- execuția care a condus la o defecțiune devine un **nou** test
  - folosit la **testarea de regresie**
- în timpul / după rezolvarea problemei, se folosesc **testele unitare existente** pentru a:
  - testa o metodă suspectă în izolare
  - asigura că rezolvarea problemei nu a introdus noi defecte
  - exclude ipoteze greșite despre defect

Simplificarea problemei intrărilor mari

Când depanăm putem avea o problemă cu intrările foarte mari.

Presupunem că avem următorul raport pentru o defecțiune (raportată de un utilizator, de exemplu):

binarySearch({ -256, -30, -2, -1, 5, 5, 16, 22, 44, 45, 69, 90, 90, 111, 113, 154, 200, 206, 298, 301, 305, 333, 354, 364, 376, 410, 524}, 593) care aruncă un **ArrayIndexOutOfBoundsException**

- trece prin mai multe iterații de buclă, înainte de eșec
- posibil nepotrivit pentru o depanare
- **ideal** ar fi să avem un test care eșuează, de **dimensiuni mici**

Simplificarea problemei

Dorim deci un **test mic** care eșuează

O soluție **divide-et-impera**:

1. tăiem o jumătate din intrarea testului
2. verificăm dacă una din jumătăți conduce încă la o problemă
3. continuăm până când obținem un test minim care eșuează

(notă: același principiu ca la căutarea binară!)

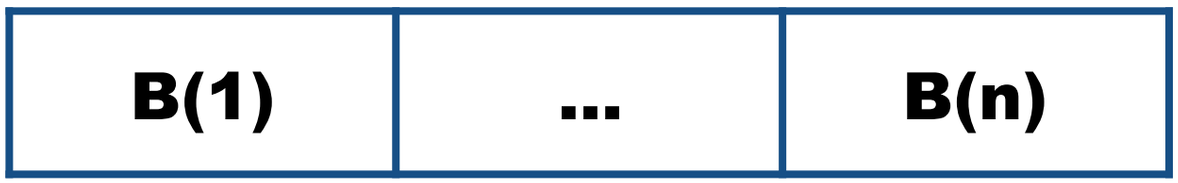
Probleme

- obositor: testele repetate manual (copy-paste, rulează din nou)
- ce facem în cazul în care nici una dintre jumătăți nu conduce la un eșec?

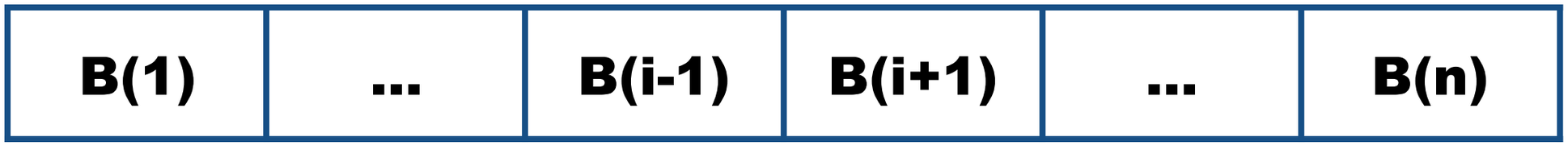
Simplificarea problemei în mod automat

Dorim **automatizarea** soluției propuse anterior

- automatizăm partea de “copy-paste” și rularea repetată de teste
- partiționăm intrarea testului în ***n*** bucăți (generalizare de la 2)



- repetăm, scoțând pe rând câte o bucată și re-testând pe intrarea rămasă



- în caz că nu obținem nici un eșec, creștem granularitatea (mărim numărul de bucăți în care este spartă intrarea)

Exemplu

Presupunem că dorim să testăm metoda următoare

```
public static int checkSum( int array[] )
```

- scopul metodei *checkSum* este de a calcula suma de control a vectorului de intrare *array*
- presupunem ca metoda întoarce un rezultat greșit ori de câte ori vectorul conține două date identice consecutive (dar noi încă nu știm aceasta)
- avem de asemenea următorul test eșuat (dat de un utilizator):

  { 1, 3, 5, 3, 9, 17, 44, 3, 6, 1, 1 , 0, 44, 1, 44, 0 }

Simplificarea intrărilor (*n* = nr. de bucăți)

1	3	5	3	9	17	44	3	6	1	1	0	44	1	44	0	✘

n=2

1	3	5	3	9	17	44	3									✔
								6	1	1	0	44	1	44	0	✘

								6	1	1	0					✘

n=4 crește granularitatea

6	1			✔
		1	0	✔

6	1	1	✘

n=3 ajustăm granularitatea la dimensiunea intrării

6	1		✔
	1	1	✘

Ciclul de viață al defectelor

The 5 Stages of Debugging

At some point in each of our lives, we must face errors in our code. Debugging is a natural healing process to help us through these times. It is important to recognize these common stages and realize that debugging will eventually come to an end.

Denial

This stage is often characterized by such phrases as "What? That's impossible," or "I know this is right." A strong sign of denial is recompiling without changing any code, "just in case."

Bargaining/Self-Blame

Several programming errors are uncovered and the programmer feels stupid and guilty for having made them. Bargaining is common: "If I fix this, will you please compile?" Also, "I only have 14 errors to go!"

Anger

Cryptic error messages send the programmer into a rage. This stage is accompanied by an hours-long and profanity-filled diatribe about the limitations of the language directed at whomever will listen.

Depression

Following the outburst, the programmer becomes aware that hours have gone by unproductively and there is still no solution in sight. The programmer becomes listless. Posture often deteriorates.

Acceptance

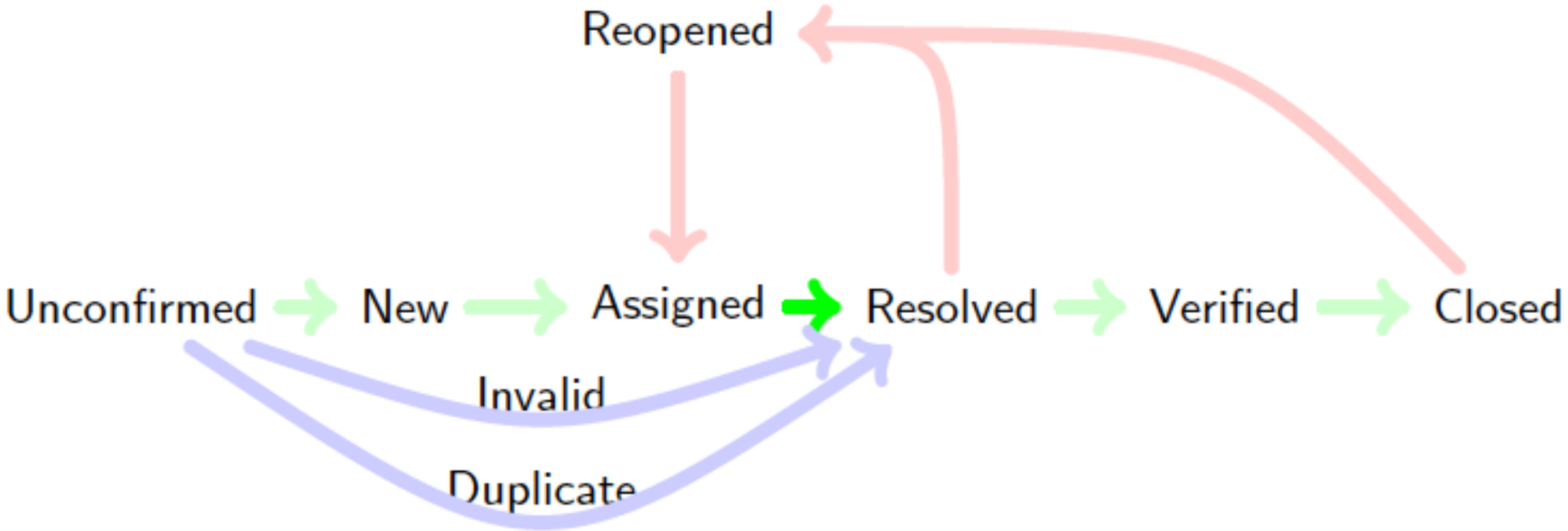
The programmer finally accepts the situation, declares the bug a "feature", and goes to play some Quake.

http://keepcalmandprogram.me/

Stadiul de viață al defectelor

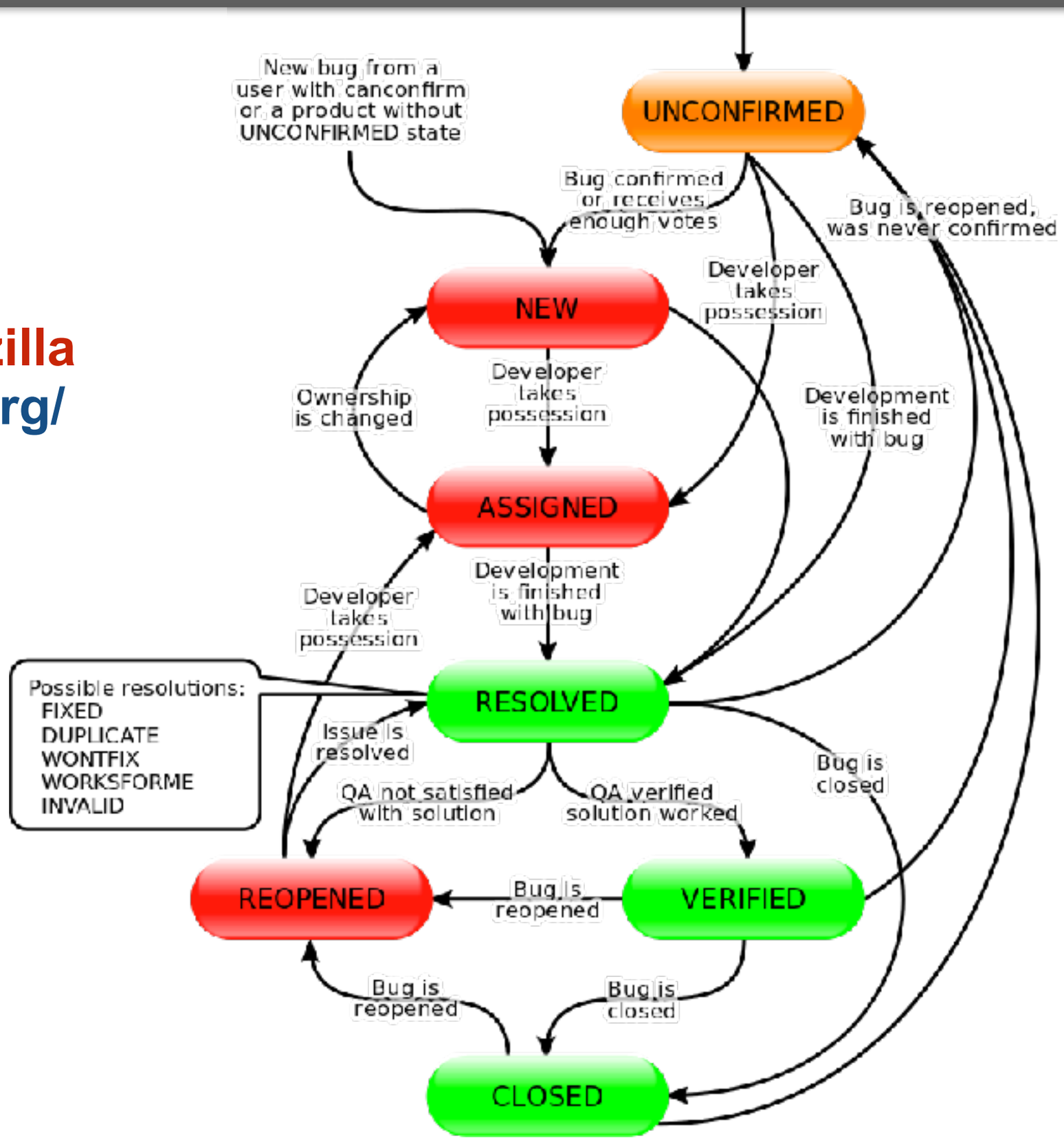
- de cele mai multe ori, persoana care raportează un defect este diferită de cea care îl repară
- un proiect mare poate include mii de defecte
- în astfel de situaţii, raportarea şi repararea nu pot fi făcute informal: un defect ar putea fi uitat şi/sau redescoperit ulterior cu un efort suplimentar
- ciclul de viaţă poate fi mărit sau micşorat în funcţie de natura proiectului
- pentru proiecte mici, un defect poate fi doar „deschis” sau „închis”
- pentru proiecte mari, un defect poate trece prin mai multe faze de urmărire

Stadiul de viață al defectelor - exemplu



Stadiul de viață al defectelor

- Exemplu:
  Ciclul de viață în **Bugzilla**
  http://www.bugzilla.org/



Clasificarea defectelor

În multe organizaţii se foloseşte o clasificare a defectelor pe 4 niveluri:

- defecte **critice**: afectează mulţi utilizatori, pot întârzia proiectul
- defecte **majore**: au un impact major, necesită un volum mare de lucru pentru a le repara, dar nu afectează substanţial graficul de lucru al proiectului
- defecte **minore**: izolate, care se manifestă rar şi au un impact minor asupra proiectului
- defecte **cosmetice**: mici greşeli care nu afectează funcţionarea corectă a produsului software urmărire