Depanarea programelor si instrumente de debug ...

Alecsandru Soare
Gemini Solutions


:(
Your PC ran into a problem and needs to restart. We're just collecting some error info, and then we'll restart for you. (0% complete)
If you'd like to know more, you can search online later for this error:
IRQL_NOT_LESS_OR_EQUAL

Ce înseammă un produs defect?

- În industrie, un produs este etichetat ca fiind **defect** atunci când acesta nu este (suficient de) sigur pentru a fi utilizat.

- Cauzele pentru care un produs ajunge să fie etichetat ca fiind defect pot fi variate însă acestea pot fi grupate în 3 mari clase:

  - Defect de concepție/proiectare (**design defect**);

  - Defect de fabricație (**manufacturing defect**);

  - Defect din punct de vedere legal (**legal defect**):

Defect de proiectare + defect de execuție

Ce înseamnă un defect software?

- Un **defect software** reprezintă:
  - O eroare (cum ar fi un typo)
  - O omisiune
  - O neînțelegere sau un eșec, etc.

- care poate apare la nivelul unei aplicații software sau al unui sistem software și care are drept consecințe:
  - Producerea unui rezultat incorect sau neașteptat;
  - Producerea unui rezultat corect însă însoțit de o serie de comportamente neașteptate sau neintenționate (blocări, acaparare de resurse cum ar fi memoria, CPU, etc.);

Defectele de proiectare și de execuție

- În industria IT defectul se numește deseori **BUG** și acesta poate fi de tip software sau hardware.

- Fenomenul de propagare a unei greșeli din etapele inițiale de dezvoltare a unei aplicații care se soldează cu unul sau mai multe defecte în etapele finale ale acestuia poartă numele de „mistake metamorphism” (din grecescul meta = ”schimbare” și morph = ”formă”).

Mistake metamorphism

Etimologia cuvintelor „bug” și „debug”

- Termenul „**bug**” e folosit în inginerie cu mult înainte de apariția calculatoarelor.
- Într-o scrisoare a lui Thomas Edison din 1878 cuvântul **bug** apare însoțit de o explicare a sa împreună cu o scurta descriere a procesului de debugging:

*„It has been just so in all of my inventions. The first step is an intuition, and comes with a burst, then difficulties arise — this thing gives out and [it is] then that "**Bugs**" — as such little faults and difficulties are called — show themselves and months of intense watching, study and labor are requisite before commercial success or failure is certainly reached.”*

- Cel mai probabil acest termen vine din arhaismul „***bugge***” folosit cu sensul de monstru în evul mediu mijlociu.

Etimologia cuvintelor „bug” și „debug”

- În anul 1931 jocul mecanic „Baffle Ball” (o variantă de pinball) era promovat într-o reclamă ca fiind „bug free”:

În timpul celui de-al doilea război mondial, orice problemă tehnică apărută în timpul funcționării echipamentelor militare era catalogată ca fiind ori **bug** ori **glitch**.

Termenul de „**debugging**” e menționat pentru prima oară în anul 1945 în rapoartele motoarelor de avion:

Journal of the Royal Aeronautical Society.49, 183/2, 1945
"It ranged ... through the stage of type test and flight test and ‘**debugging**’ ..."”

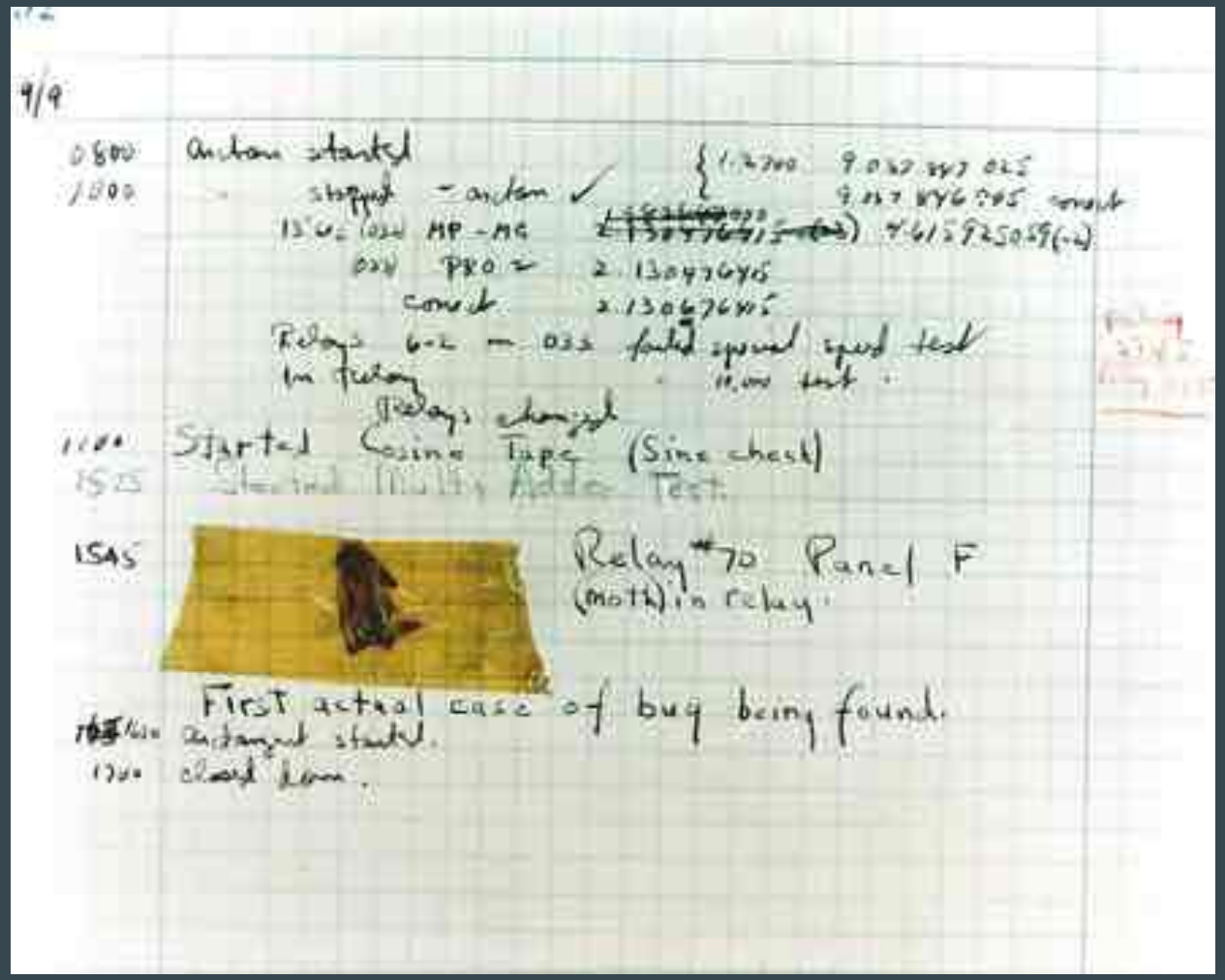
„Bug” și „debug” în IT

Cea care va populariza termenul de bug și debug în industria IT este amiral Grace Brewster Murray Hopper.

În septembrie 1947, în timp ce lucra la realizarea calculatorului Mark II în US Navy cadrul research lab din Dahlgren, Virginiala, unul dintre colegii ei de echipă a descoperit o molie prinsă în lamele unui releu, fapt care împiedica buna funcționare a acestuia.

„Bug” și „debug” în IT

- Grace Hopper va raporta acest eveniment astfel...:



Clase de defecte software

- **Defecte directe**: apar în interiorul aplicațiilor, ca efect al:
  - Erorilor sau omisiunilor din etapele de specificare și de proiectare a aplicației software (**defecte de design**);
  - Greșelilor sau erorilor ce apar în etapa de implementare a aplicației software (**defecte de fabricație**);
- **Defecte indirecte**: apar ca o consecință a funcționării aplicațiilor într-un anumit context. Astfel avem defecte datorate utilizării unor:
  - Biblioteci extene (third party frameworks) instabile;
  - Sisteme de operare care conțin la rândul lor defecte sau incompatibilități;
  - Compilatoare care produc un cod executabil incorect sau incomplet;
  - Dispozitive hardware instabile sau incompatibile;

Efectele defectelor software

- Bug-urile software pot declanșa la rândul lor o serie de efecte:

- **Efecte subtile**: funcționalitatea software-ului pare a fi corectă. Totuși, dacă acestea sunt lăsate să ruleze pentru o perioadă mai lungă de timp (deseori variabilă), atunci aceste efecte subtile se vor acumula ducând în final la efecte vizibile.

- Exemple de defecte cu efecte subtile:
  - Win 9x timer wrap bug at 49.5 days;
  - Win Vista, 7, 2008 a 497 days uptime => TCP/IP killer;
  - Defectul rachetelor Patriot folosite in primul război din Iraq, ce au acumulat un delay de 0.34 s, dupa 100 h de functionare.

Efectele subtile

Efectele defectelor software

- **Efecte tranzitorii**: funcționalitatea software-ului este afectată pe termen scurt după care aceasta revine la normal. Aceste defecte poartă numele de **glitch**-uri și sunt uneori deosebit de greu de reprodus și mai ales de reparat. În această clasă intră

  - Defectele de timing;
  - Defectele de inițializare;
  - Defecte datorate erorilor de comunicație.

Efectele tranzitorii

Efectele defectelor software

- **Efecte vizibile**: funcționalitatea software-ului este afectată în general de o serie de factori externi cum ar fi:

  - O combinație a datelor de intrare;
  - O secvență de comenzi și acțiuni;
  - O configurație hardware particulară

- Efectele vizibile se manifestă prin „blocarea/înghețarea” (**freeze**) sau „crăparea” (**crash**) aplicației.
  În această categorie intră majoritatea bug-urilor software.

Efectele vizibile

Efectele defectelor software

- **Efecte secudare**: funcționalitatea software-ului nu este afectată. Ceea ce este afectat în acest caz este stabilitatea și/sau securitatea sistemului hardware/software pe care aplicația în cauză este executată. În această clasă de bug-uri intră:
  - Aplicațiile care acaparează în mod nejustificat o serie de resurse software (fișiere, porturi de TCP/IP, PID-uri) sau hardware (memorie sau CPU);
  - Aplicații sau componente software a căror execuție conduce la apariția unor „găuri”/breșe de securitate.

Exemple de defecte

Omisiunile la nivelul unei etape se vor traduce deseori prin absența unor functionalități sau prin prezența unor funcționalități incomplete și/sau incorecte în etapa finală:

- În cazul unui editor, omiterea unui mecanism de lock-on-edit poate conduce la pierderi de date atunci când un fișier e editat simultan de mai multe instanțe diferite ale editorului;
- Bug-ul **FDIV** din procesorul Intel Pentium - a apărut în etapa de introducere a unor constante într-un tabel de asociere (tabel folosit în accelerarea operației de împarțire în virgulă mobilă). Un inginer a omis să introducă o parte din aceste constante, ele fiind automat setate pe 0 în etapa de producție => pierderi de milioane de $ pentru Intel;

Exemple de defecte

Typos: Folosirea incorectă a unui simbol/termen/entitate.

- În etapa de design sau de coding, un programator poate omite, poate tasta incorect sau poate tasta de mai multe ori un simbol rezultând o altă funcționalitate.
  - Exemple:
    - x < y sau x > y în loc de x <= y;
    - x < 1 în loc de x << 1;
    - în cazul pointerilor *x în loc de **x sau invers;

```
int average(int a, int b)
{
   return a + b / 2;     /* should be (a + b) / 2 */
}
```

Exemple de defecte

- Folosirea ambiguă a unui simbol/termen/entitate
  - Exemplu:

```
int iNesSaveAs(char* name)
{
  ...
  fp = fopen(name,"wb");
  int x = 0;
  if (!fp)
    int x = 1;
  ...
}
```

Exemple de defecte

Folosirea unor **asumpții incorecte** într-o etapă vor conduce la folosirea incorectă a unor simboluri/termeni/entități în etapa curentă și/sau în etapele ulterioare datorate conversiilor defectuoase de reprezentare dintr-o etapă în cealaltă:

- În etapa de design sau de coding, un algoritm de sortare alfa-numerică presupune că tipul de date asociat caracterelor va fi în format ASCII (un octet) iar atunci când acestea sunt reprezentate în format UNICODE (doi octeți) acest algoritm nu va funcționa corect întotdeauna sau nu va funcționa deloc, el putând bloca aplicația;

Defectele software există. Ce facem cu ele?

Pentru ca un program să fie declarat *produs* el trebuie să îndeplinească mai multe condiții esențiale intre care amintim:

1) **Să îndeplinească funcțiile și cerințele** pentru care a fost proiectat realizat – criteriu fidelității (=produsul respectă fidel cerințele producătorului) și al validității (=produsul îndeplinește cererile utilizatorilor săi);
2) **Să îndeplinească constrângerile de timp și de calcul impuse** în etapa de proiectare (criteriul eficacității derivat din cele doua criterii mentionate mai sus);
3) **Să ofere o ergonomie superioară în utilizare** (criteriul UXD – sau criteriul plăcerii în utilizare - e un criteriu subiectiv însă el cântărește foarte mult în ochii clienților);
4) **Să nu îndeplinească nicio altă funcție în afară de cele pentru care a fost proiectat și realizat** – adică sa nu conțină/fie virus, malware și să nu prezinte (d)efectele sus-menționate.

Un program care, în timpul sau la finalul procesului de dezvoltare, nu îndeplinește criteriul (4) e considerat ca fiind defect (sau buggy). Toate defectele identificate pe parcursul dezvoltării aplicației vor fi catalogate și clasificate folosind: bug reports, defect reports, fault reports, problem reports, trouble reports, change requests, etc.

Bugurile există. Cum le combatem?

Majoritatea erorilor logice pot fi evitatea folosind un stil de programare (**programming style**) sau o serie de tehnici de programare (**programming techniques**) cum ar fi:

Programarea defensivă (**defensive programming**) sau programarea sigură (**secure programming**) = reprezintă proiectarea defensivă a unei rutine/funcții astfel încânt aceasta să continue să funcționeze chiar și în situațiile neprevăzute – inclusiv situațiile în care acea secțiune de cod este folosită necorespunzător (codul devine fool-proof).

Bugurile există. Cum le combatem?

Programarea defensivă - Asumpții de bază:

- **Să nu ai încredere în date**(le de la intrare, de la ieșire și nici în rezultatele intermediare): programatorul va verifica datele și nu va presupune niciodată că ele vor veni bine-formate, vor fi în intervalul sau în domeniul de valori permise, etc.

- **Să nu ai încredere în cod**(ul altuia): programatorul va folosi codul third party într-o manieră în care să-i permită: error recovery și tratarea excepțiilor. El va combina frecvent această asumpție cu prima asumpție verificând mereu rezultatele intermediare (comparând rezultatele obținute cu valorile sau cu domeniile de valori așteptate).

Defensive programming - continuare

Aspectele pro ale programării defensive:

- **Crește calitatea** aplicației deoarece aceasta e proiectată special pentru a reduce problemele și bug-urile;
- **Crește lizibilitatea** codului;
- **Crește stabilitatea** codului în situații de excepție;

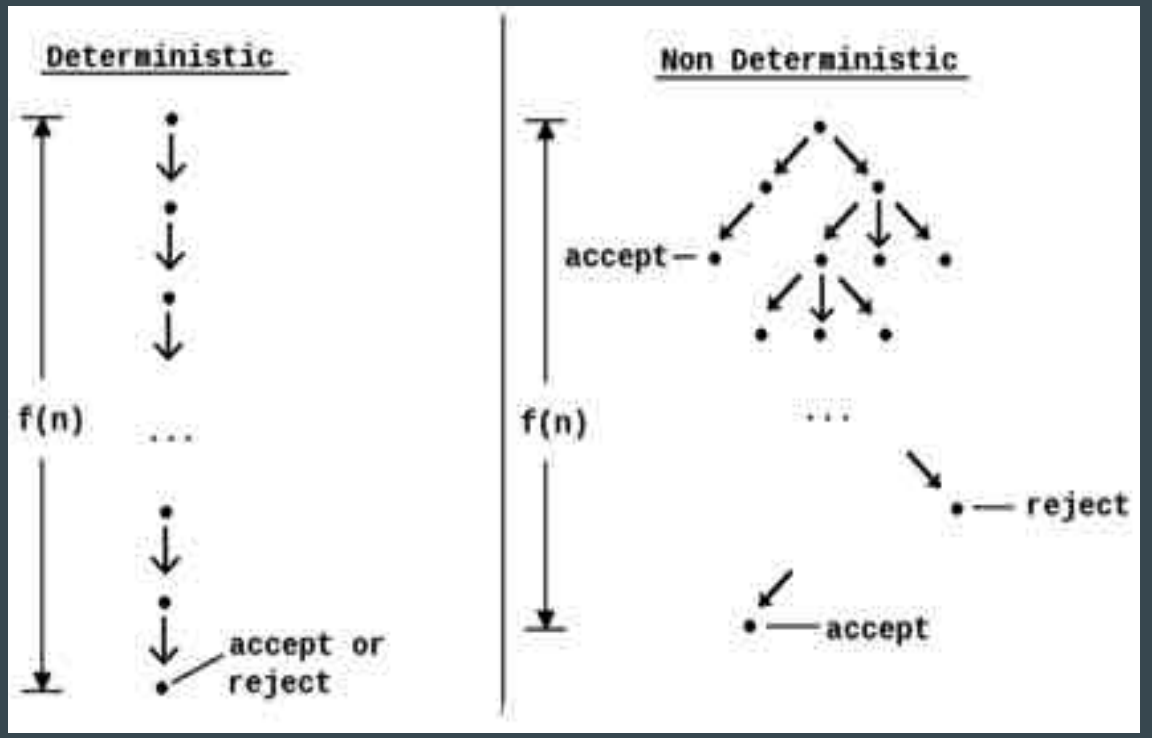
Aspecte contra ale programării defensive:

- **Crește overhead-ul codului** = aplicația va necesita mai multe resurse deoarece crește cantitatea de cod care trebuie executată:
- **Cod total** = **cod care verifică** (pt. situațiile neprevăzute) + **cod util**;
- **Crește costul codului** = deoarece crește cantitatea de cod care trebuie implementată și întreținută;
- **Există riscul de a face „prea mult bine”** în sensul că sunt evitate situațiile în care codul ar trebui să eșueze cu adevărat;

Bugurile există. Cum le combatem?

Metodologiile de dezvoltate (**development methodologies**). În acest caz, în etapele de specificare și design al aplicației se pot folosi **specificații formale complete** (care să conțină comportamentul exact al funcționalităților). În realitate această condiție este aproape imposibil de îndeplinit pentru aplicațiile software complexe datorită:

- **exploziei combinatorice**
- **algoritmilor nedeterministici**



Bugurile și development methodologies

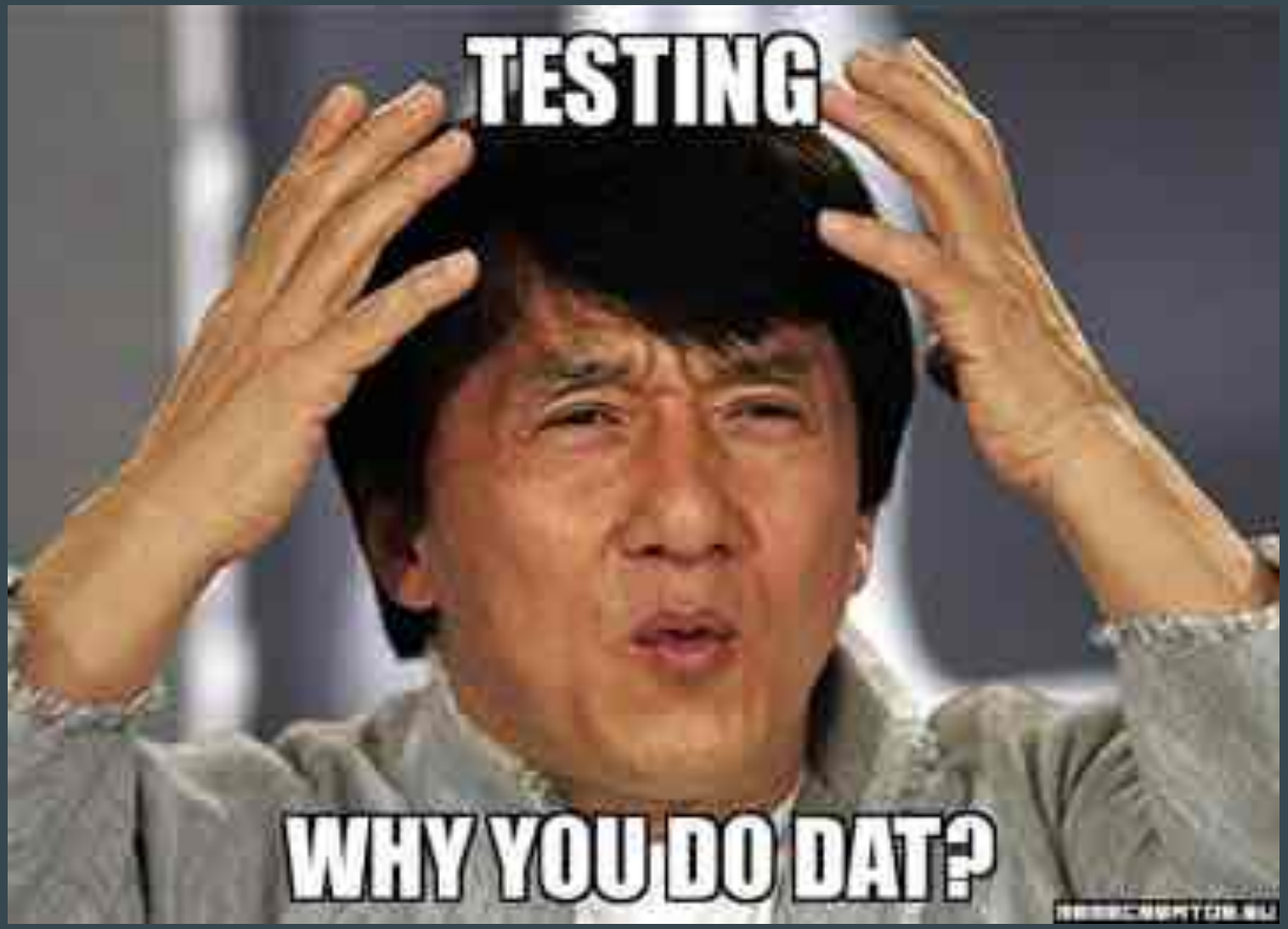
În prezent sunt preferate **metodologiile de lucru de tip agile**, care au la bază automatizarea testării (**test-driven development**) și care contin etape cum ar fi:

- automatizare unit-testing;
- automatizare acceptance testing;

Avantaje:

- **Evidențiază cerințele** care au fost **specificate incomplet sau incorect**;
- Scade timpul și costul ciclului de dezvoltare datorită apariției fenomenului de **continuous-testing** și implicit de **continous-bug-fixing**;

Testing, testing, testing



Bugurile există. Cum le combatem?

Specs, Design and Code review:

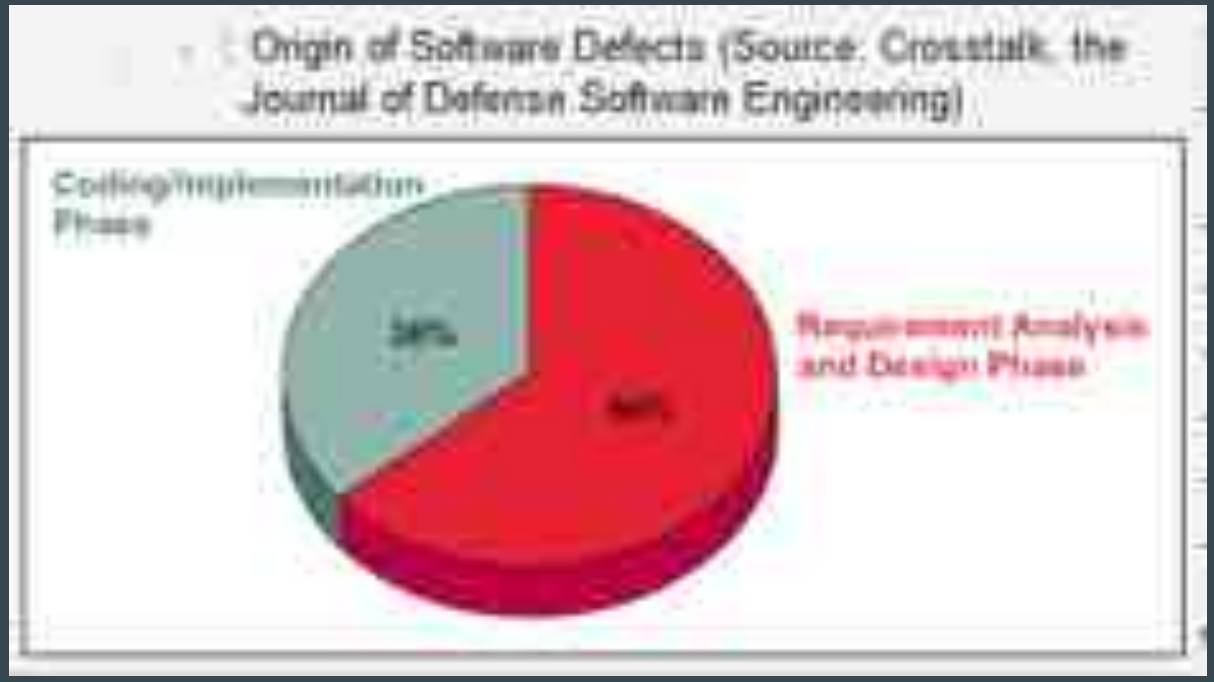
IEEE Software afirmă că **orice formă de revizuire** dar **în special revizuirea cerințelor, a specificațiilor, a designului și a codului** conduc la îndepărtarea a până la 90% dintre erorile unui produs software - acest efect având loc **înaintea executării oricărui test**.

Revizuirea riguroasă a activității softwarelui **este mai eficientă** (atât d.p.d.v. economic cât și practic) **decât orice altă strategie de îndepărtare a codului, inclusiv decât testarea**.

Totuși IEEE subliniază faptul că revizuirea specificatiilor și a codului nu poate și nu trebuie să înlocuiască procesul de testare.

Bugurile există. Cum le combatem?

Dar de ce **specs & design review** și nu direct **code review**?
Potrivit Journal of Defense Software Engineering, majoritatea defectelor din produsele software se datorează greșelilor produse în fazele de requrements and design și ele reprezintă până la 64% din totalul costurilor produse de defectele software (aici nu e vorba de numărul defectelor ci de efectele acestor defecte asupra functionalității produselor):

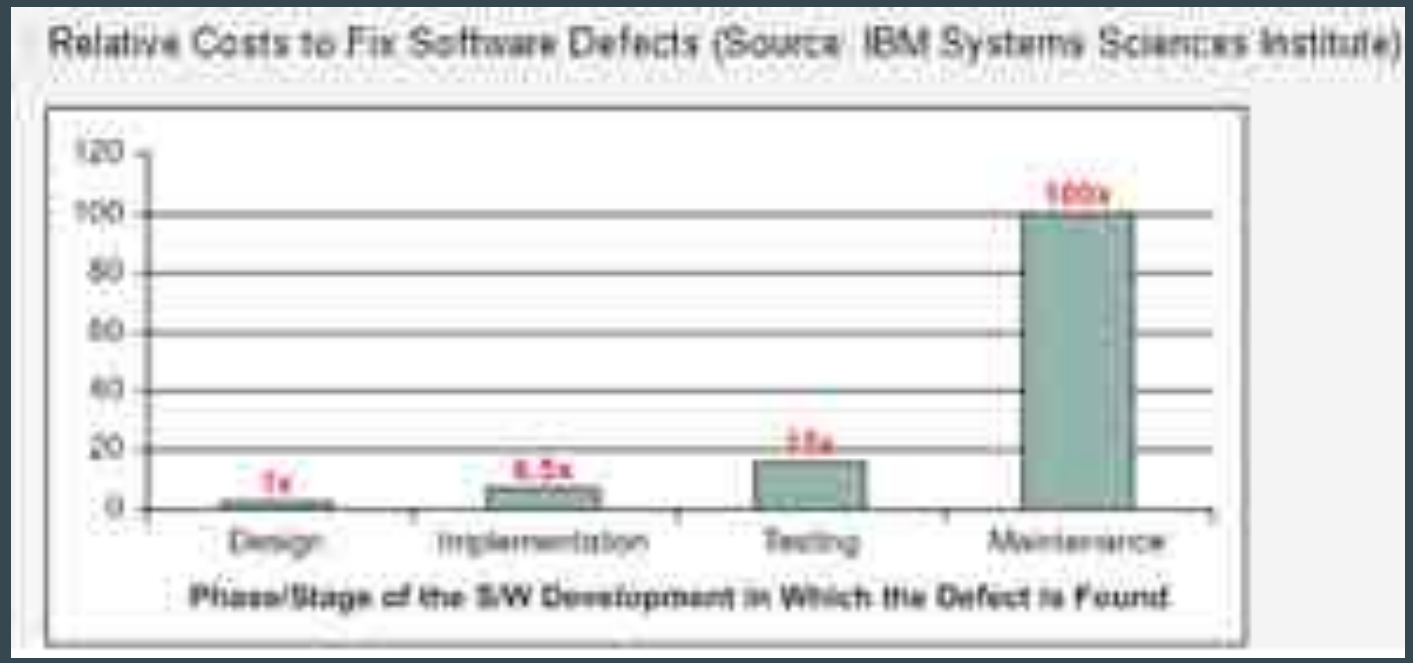


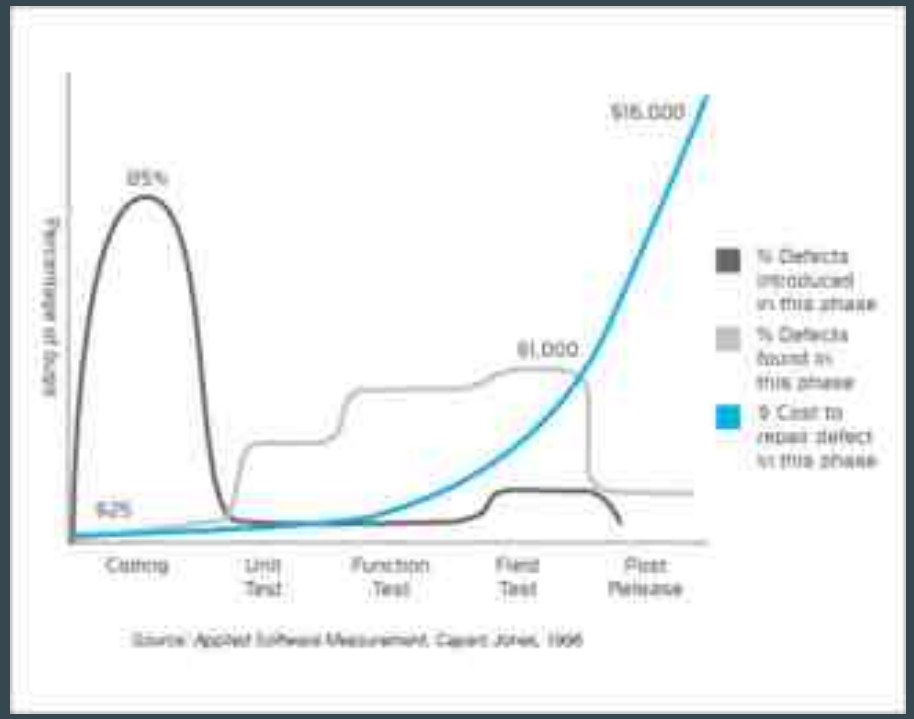
Bugurile există. Cum le combatem?

Detectarea cât mai rapidă a bug-urilor

Costul reparării unui defect detectat în faza de implementare este de 4-6 ori mai mare decât costul reparării acestuia atunci când acesta e detectat în faza de design.

Prin comparație **costul reparării crește exponențial** cu până la 100 ori atunci când bug-ul e detectat în faza de întreținere:





Bugurile există. Cum le combatem?

Limbaje de programare „deștepte”

La nivelul unor **limbaje de programare** există o serie de mecanisme care permit sau împiedică apariția unor erori logice și care cresc organizarea codului (și care scad ambiguitatea).

Exemple de mecanisme de limbaj:

- Static type system (în majoritatea compilatoarelor);
- Namespaces și încapsularea din OOP care rezolvă o serie de ambiguități și conflicte de nume;
- Programarea modulară;
- Managementul automat al memoriei heap (garbage collector);
- Absența aritmeticii pointerilor;
- Implementarea de „bounds checking” pentru vectori (vezi limbajul Pascal și o serie de limbaje de scripting), etc.;

Bugurile există. Cum le combatem?

Instrumente de analiza de cod (**code analysis**) – care se bazează pe **analiza statică a codului** și pe faptul că programatorii fac frecvent anumite tipuri de erori atunci când scriu un program într-un anumit limbaj de programare.

Instrumentarea codului – prin folosirea unor aplicații și/sau a unor biblioteci specializate codul poate fi monitorizat, măsurat și controlat mai bine în vederea identificării „gâtuirilor” (deseori produse de un algoritm sub-optim sau de un algoritm care conține defecte subtile) și pentru a ne asigura de faptul că acesta funcționează corect (**analiză dinamică/la execuție a codului**).

Bugurile există. Cum le combatem?

Identificarea și minimizarea cauzelor apariției defectelor.
Principalele cauze ale apariției defectelor în produsele software sunt:
(EXERCIȚIU: identificați în care din acești factori intervine elementul uman)

- Comunicarea defectuoasă și/sau calitatea cerințelor;
- Alocarea nerealistă a timpului pentru dezvoltarea aplicației;
- Lipsa de experiență în faza de design;
- Lipsa de experiență în ceea ce privește bunele practici în programare;
- Introducerea accidentală de greșeli în faza de implementare;
- Lipsa unor unelte de control al progresului dezvoltării produsului software (SVN, Git, Confluence, Jira, Pivotal etc.);
- Folosirea de biblioteci third-party care conțin defecte;
- Modificări de ultim moment la nivelul cerințelor;
- Lipsa unor abilități de testare;

Debugging / Debuggers

Găsirea unui bug într-o aplicație face parte din procesul de **testare**.

Găsirea unui bug în codul sursă al unei aplicații urmată de fixarea acestuia face parte din procesul de **debugging & bug fixing**.

În procesul de **debugging & bug fixing**, prima parte, cea de **debugging**, este deseori cea mai grea. Deseori etapa de **bug-fixing** e ușoară, odată identificat bug-ul în codul sursă.

Pentru a ușura această sarcină au fost create programe specializate numite **debuggere**.

Debugging / Debuggers

Ce este un debugger?

Un debugger este o aplicație software folosită pentru a testa și a depana o altă aplicație (deseori numită „target”).
Un debugger poate porni target-ul sau se poate atașa la acesta în timp ce target-ul rulează.

Majoritatea debuggerelor sunt aplicații care sunt controlate prin intermediul unei interfețe de tip CLI (command line interface).

Totuși majoritatea programatorilor preferă folosirea lor prin intermediul unor așa-numite **debugger front-ends**, interfețe grafice (uzual din IDE-uri) care permit o serie de efecte vizuale cum ar fi execuția animată a codului, inspecția vizuală a variabilelor, etc.

Debugging / Debuggers

Aplicația target poate rula în mai multe moduri:
(EXERCIȚIU: identificați care sunt avantajele și dezavantajele fiecărei modalități)

- **Într-un mediu/context simulat** de către debugger (în acest caz debugger-ul conține un instruction-set simulator);

- **Pe aceeași platformă hardware/software** cu debugger-ul – acesta este cel mai întâlnit caz (încă);

- **Pe o altă platformă hardware/software** identică sau diferită situată la distanța. În acest caz vorbim despre remote-debugging. Acest mod apare frecvent în embedded development și în mobile development;

Debugging / Debuggers

Atributele unui debugger:

- Permit executarea codului linie cu linie;
- Într-o linie de cod pot permite executarea operațiilor pas cu pas;
- Permit inspectarea și modificarea valorilor variabilelor;
- Permit inspectarea stivei de apeluri;
- Permit inspectarea diferitelor threaduri;
- Permit suspendarea condiționată sau necondiționată a execuției unei aplicații într-un anumit punct;
- Unele debuggere permit modificarea atât a stării target-ului cât și a funcționalității target-ului fără a-l reporni;
- Unele debuggere implementează operația de „reverse debugging”;

Breakpoint-urile

Un **breakpoint** resprezintă un punct în cadrul unui program folosit pentru a opri execuția acestuia în acel loc.

Pe lângă operația de debugging, **breakpointurile** mai pot fi folosite și în operațiile de reverse engineering și/sau cracking.

Atunci când execuția programului este suspendată (**pause**) programatorul poate inspecta:

- mediul/contextul intern al programului: variabile locale și globale, regiștrii, memoria, stiva de apelurii (call stack);
- mediul/contextul extern al programului: log-uri, fișiere (de configurare sau fisiere generate), ce a afișat până în acel punct, etc.

Breakpoint-urile

În funcție de natura lor breakpoint-urile pot fi:

- **Breakpoint-uri necondiționale**;
- **Breakpoint-uri condiționale**;
- Alte tipuri de breakpoint-uri cum ar fi **timing breakpoints**, **event breakpoints**, etc.;

Breakpoint-uri necondiționale = aplicația target se va opri totdeauna la o anumită instrucțiune. Aceste breakpoint-uri se mai numesc și **instruction breakpoints**.

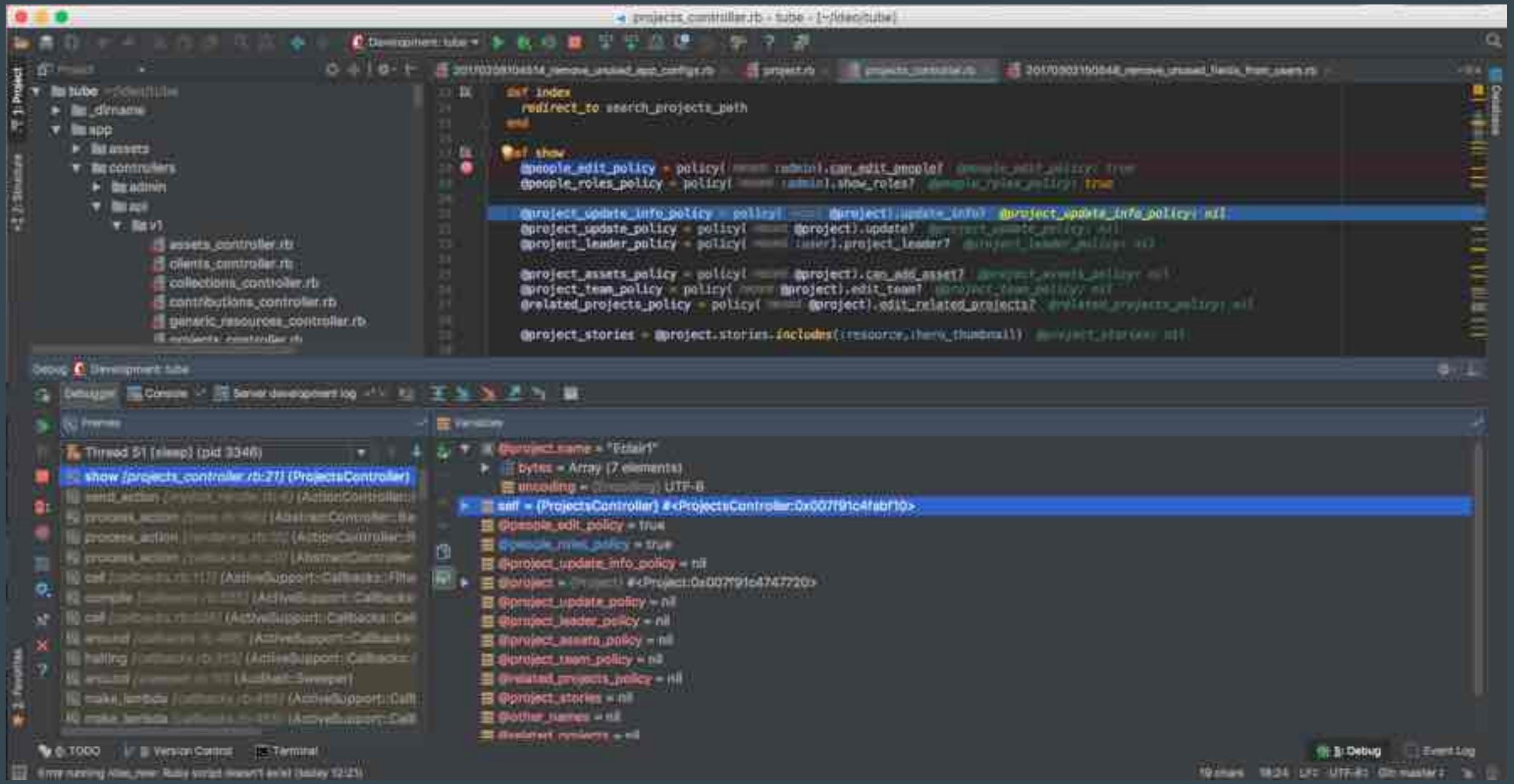
Breakpoint-urile

Breakpoint-uri condiționale: aplicația target se va opri doar dacă o anumita condiție devine adevărată. Condiția vizează o anumită locație de memorie și o anumită operație (de read, write sau modify) asupra acesteia. Astfel programul se poate opri într-un anumit punct ori de câte ori condiția e adevărată (**conditional instruction breakpoint**) sau în toate punctele în care condiția devine adevărată (**data breakpoint** sau **watchpoint**);

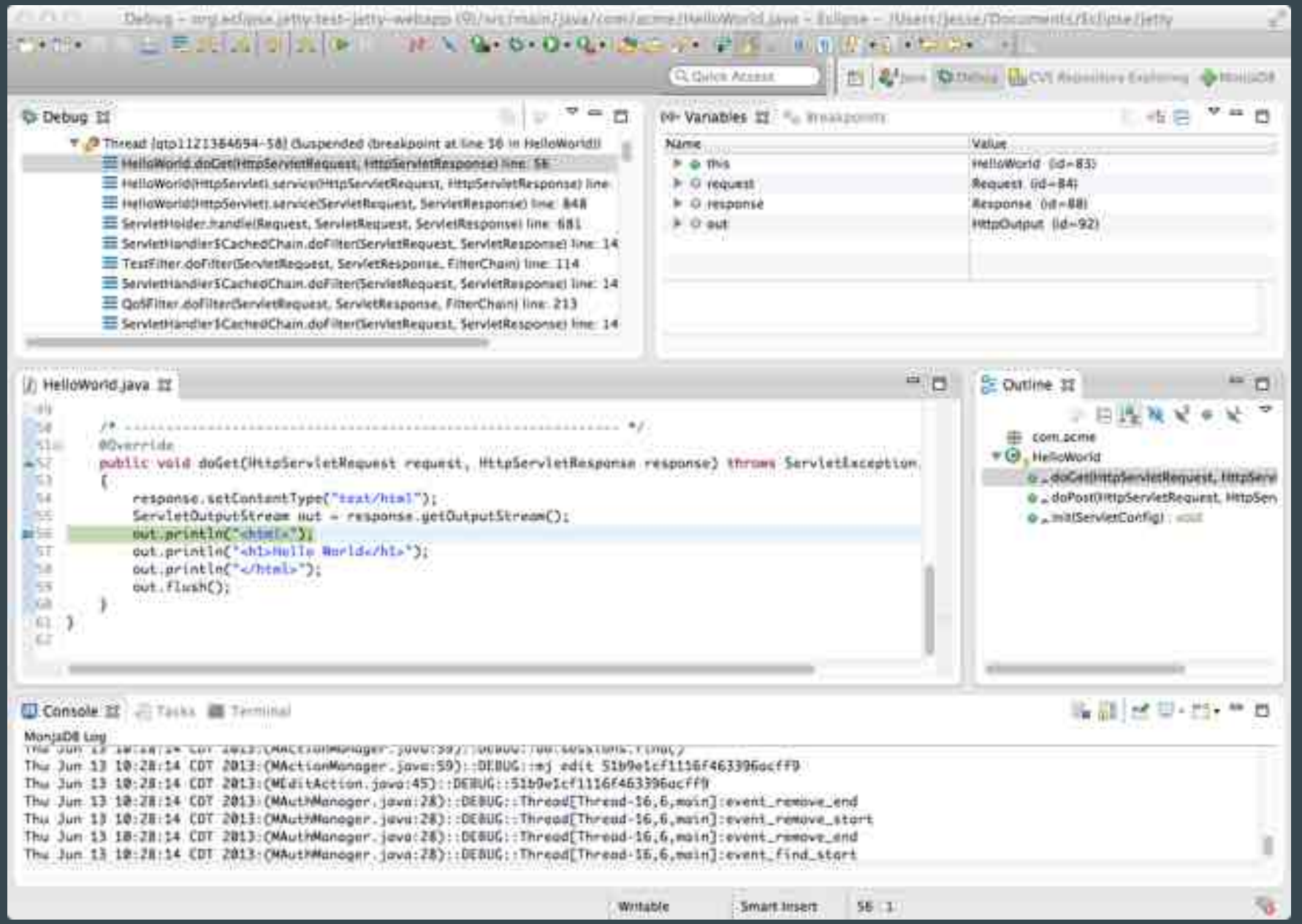
Timing breakpoint: aplicația este întreruptă automat la expirarea unui anumit timer;

Event breakpoint: aplicația este întreruptă la apariția unui eveniment cum ar fi apăsarea unei taste sau a unui mesaj ce provine de la un alt thread, de la o altă aplicație sau de la SO.

Breakpoints



Breakpoints



Breakpoints