



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)

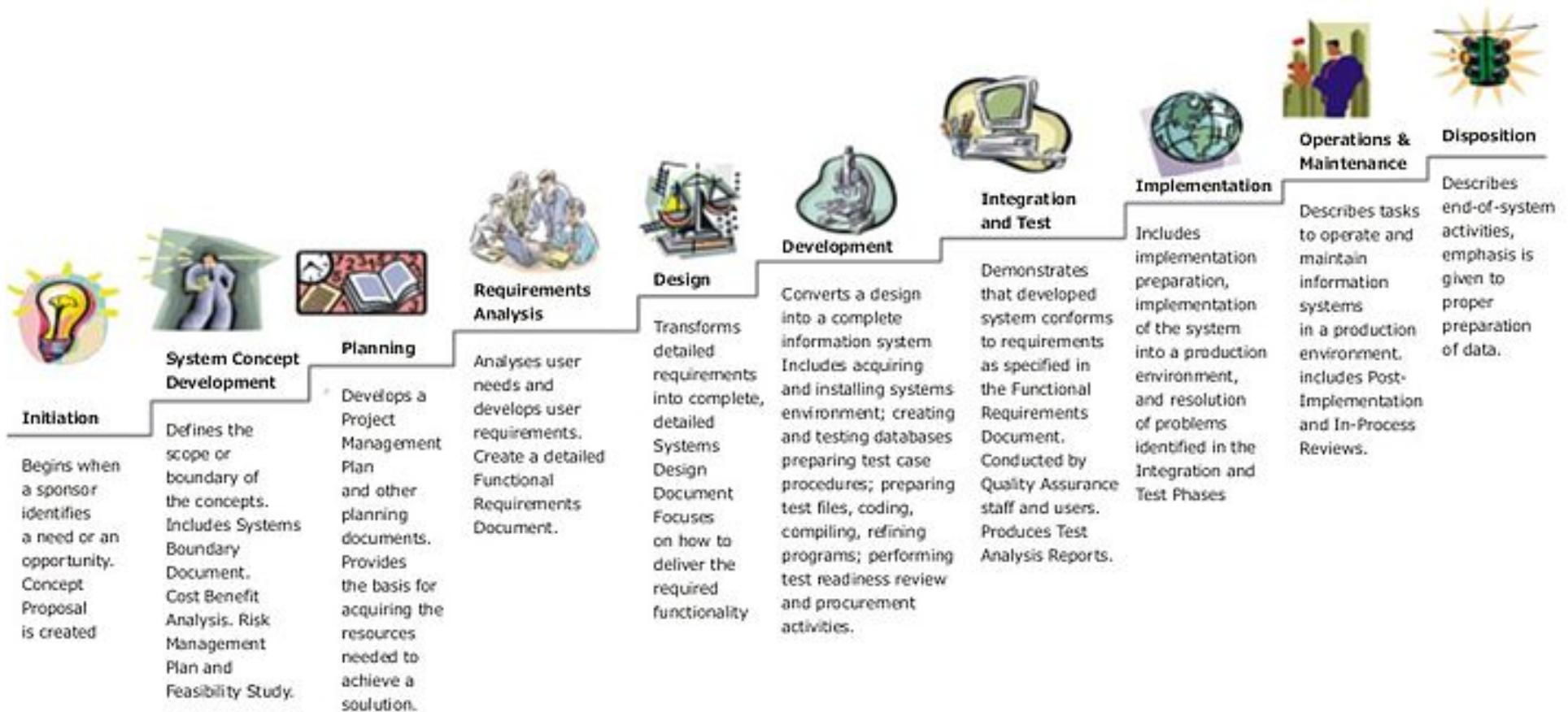
If you'd like to know more, you can search online later for this error:
IRQL NOT LESS OR EQUAL

Instrumente de debug

Soare Alecsandru Florin
Making Software Happen
<http://geminisols.com>

Systems Development Life Cycle (SDLC)

Life-Cycle Phases



Ce înseamnă 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 apărea la nivelul unei aplicații software sau al unui sistem software și care are drept consecințe:

Producerea unui rezultat incorrect sau neașteptat;

Producerea unui rezultat corect însă însotit 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 initiale 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



**99 little bugs in the code.
99 little bugs in the code.
Take one down, patch it around.**

127 little bugs in the code...

Paranteză (Istoricul termenelor de „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 însotit 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.”*
- 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.

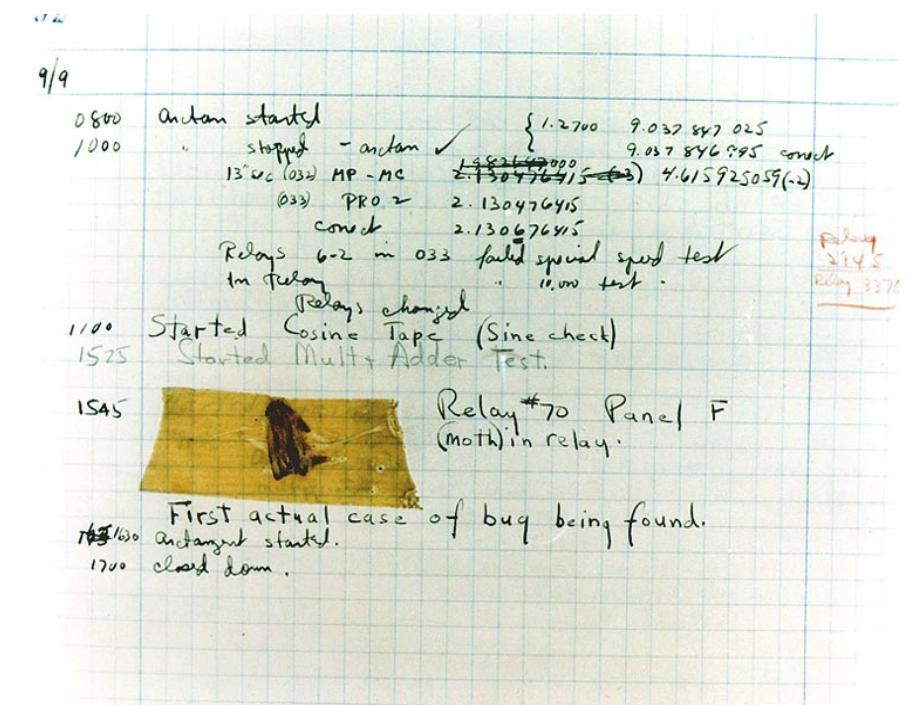
„Bug” și „debug” în IT

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.

Grace Hopper va raporta acest eveniment astfel...:

Sfârșitul parantezei)



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 incorrect sau incomplet;

Dispozitive hardware instabile sau incompatibile;

Etc.

Exemplu de (d)efect de context – rezultat în urma unui JAR hell

- Initial avem acest defect (prioritate p1, bug critic):

regression: "Measure your ingredients" screen if rotating the tablet being on Step1 of make mode

Steps:

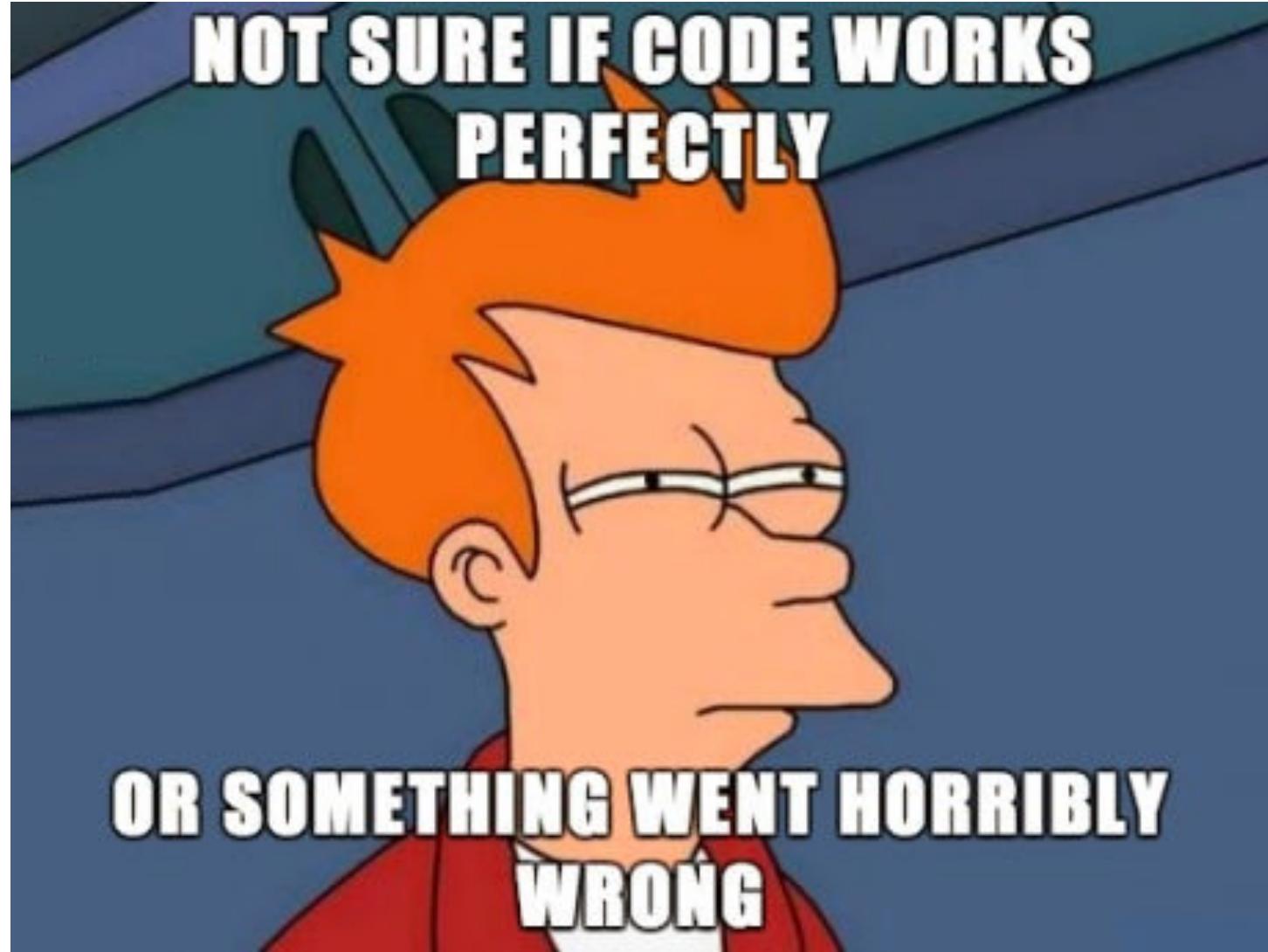
1. Instal Yummly
2. Login with a WHR account
3. Open a guided recipe
4. Tap on Make it
5. Advance to step1
6. Rotate the tablet

Actual result:

"Measure your ingredients" screen is shown. Yummly crashes if closing the screen.

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 în primul război din Irak;
 - Etc.



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:

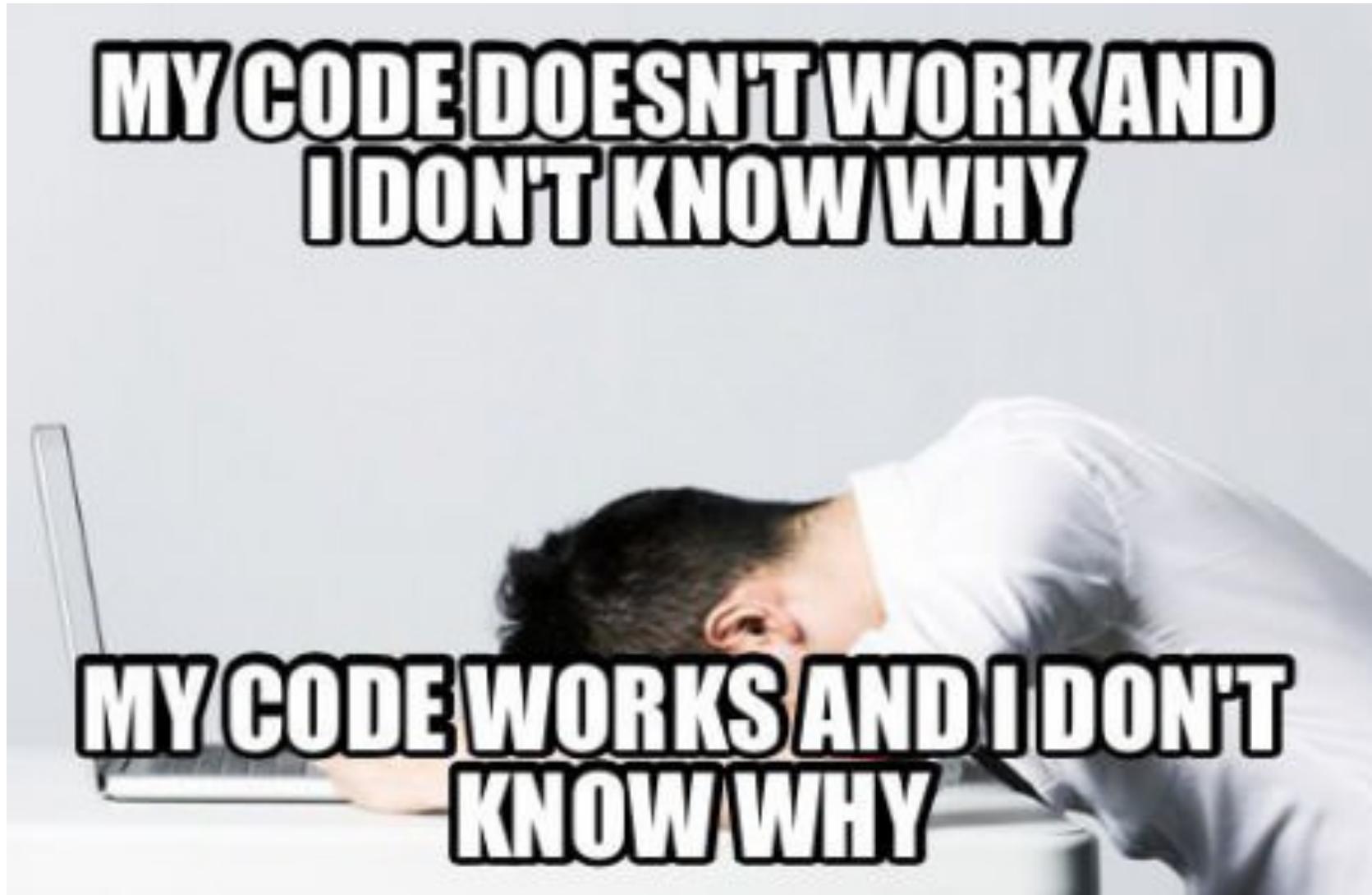
Defectele de timing;

Defectele de inițializare;

Defecte datorate erorilor de comunicație;

Etc.;





Exemplu de glitch si de fixare defectuoasă a acestuia

- Initial avem acest defect (prioritate p3, bug estetic):

The counting button has a small glitch when resuming the app

Steps:

1. Install Yummly and login with a WHR account
2. Open a guided recipe
3. Tap on Make it - start cooking
4. Advance to a step with appliance timer
5. Tap on Start Timer and put the app in background
6. Put the app in foreground

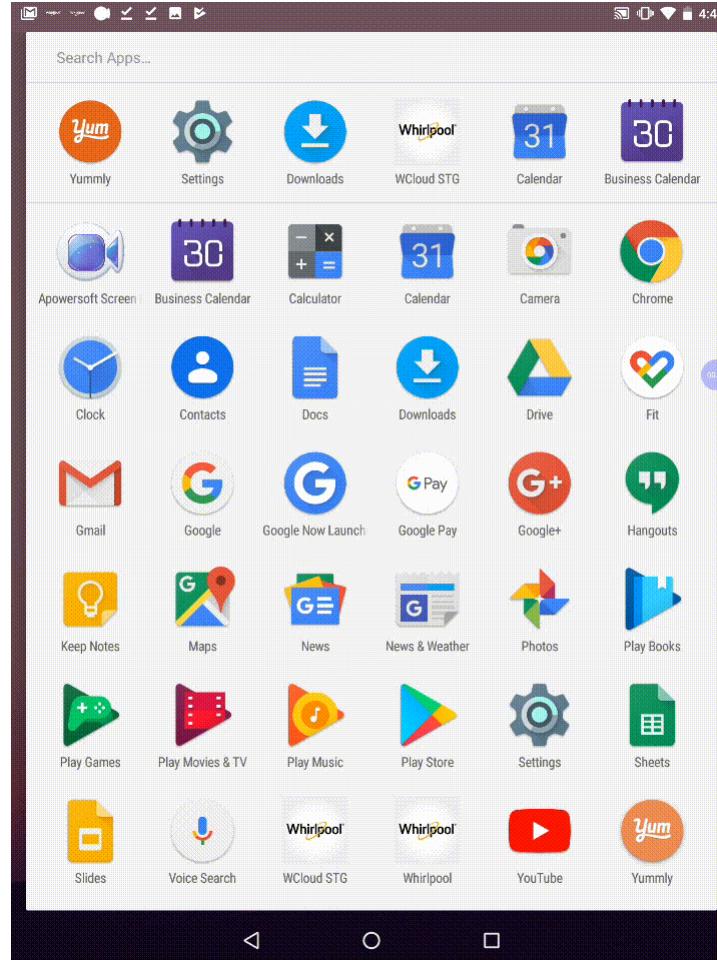
Actual result:

The counting button has a small glitch (see the video attached).

Exemplu de glitch si de fixare defectuoasă a acestuia

- Care, vizual, arată astfel:
- Fixul pare să simplu – undeva, în cod, trebuia făcut un refresh:
...

```
if (isTempButton() || isTimerButton()) {  
  
    refreshText();  
  
}  
...  
...
```



Exemplu de glitch si de fixare defectuoasă a acestuia

- Însă, 7 săptămâni mai târziu, acel fix va produce un alt (d)efect pe un alt scenariu (prioritate p1, bug critic):

Unconnected mode - Yummly crashes when extra time expires

Steps:

1. Install Yummly and login or not.
2. Login with WHR account or not
3. Open a guided recipe
4. Start cooking it in unconnected mode
5. Advance to a step with Add Time button
6. Add some extra time

Actual result:

When the extra time expires a crash occurs

- Problema a fost de la o eroare de sincronizare (un race condition) care conducea, în acest caz, la o buclă infinită.
- Despre erorile de sincronizare vom vorbi în slide-urile următoare....

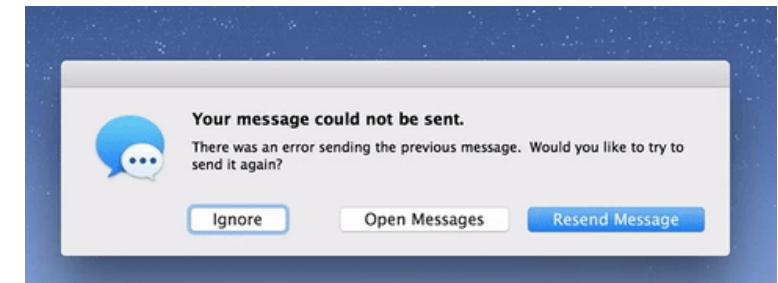
Efectele defectelor software

- **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 secentă de comenzi și acțiuni;
O configurație hardware particulară;
Etc.

- Efectele vizibile se manifestă prin intrarea într-o buclă infinită, prin „blocarea/înghețarea” (**freeze**) sau prin „crăparea” (**crash**) aplicației.

- În această categorie intră majoritatea bug-urilor software.

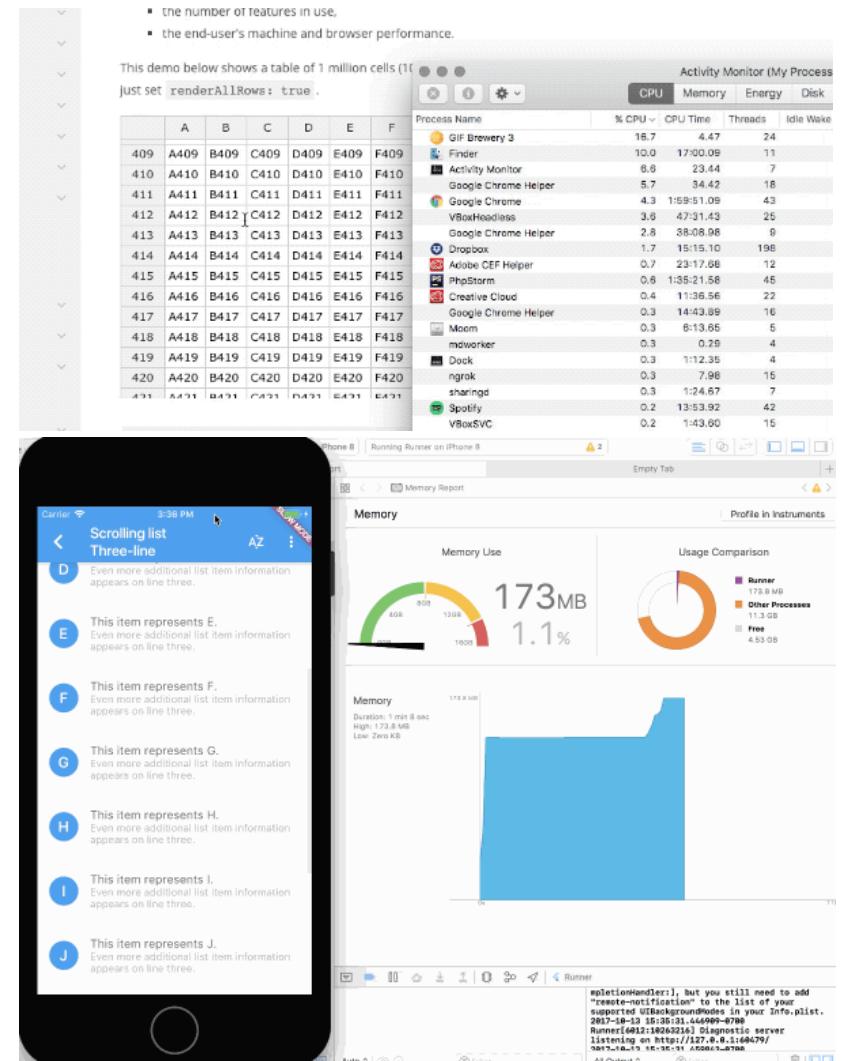




Efectele defectelor software

- **Efecte secundare:** 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 funcționalități 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 împărțire 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 dolari pentru Intel;

Exemple de defecte

Folosirea unor **asumptii incorecte** într-o etapă vor conduce la folosirea incorectă a unor simboluri/termeni/entități în etapa curentă și/sau în etapele ulterioare datorare 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;

Exemple de defecte

Typos: Folosirea incorectă sau omisiunea 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 $x \leq y$;

$x < 1$ în loc de $x \ll 1$;

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 */
}
```

Exemplu de defecte – managementul memoriei

```
struct channel {  
    char *buffer;  
    char *dir_flags;  
};  
  
struct channel alloc_chan(int chan_id, size_t size) {  
    struct channel retval;  
    if (chan_id > 0) {  
        retval.buffer = (char *) malloc(size);  
        char flags[2] = {READ, WRITE};  
        retval.dir_flags = flags;  
    }  
  
    if (chan_id == 1) {  
        retval.buffer = NULL;  
        retval.dir_flags = NULL;  
    }  
    return retval;  
}
```

```
void free_chan(bool cond, struct channel chan) {  
    if (cond) {  
        free(chan.buffer);  
    }  
  
    free(chan.buffer);  
    free(chan.dir_flags);  
}
```

Exemple de defecte – Memory leaks

```
struct channel {  
    char *buffer;  
    char *dir_flags;  
};  
  
struct channel alloc_chan(int chan_id, size_t size) {  
    struct channel retval;  
    if (chan_id > 0) {  
        retval.buffer = (char *) malloc(size);  
        char flags[2] = {READ, WRITE};  
        retval.dir_flags = flags;  
    }  
  
    if (chan_id == 1) {  
        retval.buffer = NULL;  
        retval.dir_flags = NULL;  
    }  
    return retval;  
}
```

```
void free_chan(bool cond, struct channel chan) {  
    if (cond) {  
        free(chan.buffer);  
    }  
  
    free(chan.buffer);  
    free(chan.dir_flags);  
}
```

Exemplu de defecte – „alocarea” pe stivă

```
struct channel {  
    char *buffer;  
    char *dir_flags;  
};  
  
struct channel alloc_chan(int chan_id, size_t size) {  
    struct channel retval;  
    if (chan_id > 0) {  
        retval.buffer = (char *) malloc(size);  
        char flags[2] = {READ, WRITE};  
        retval.dir_flags = flags;  
    }  
  
    if (chan_id == 1) {  
        retval.buffer = NULL;  
        retval.dir_flags = NULL;  
    }  
    return retval;  
}
```

```
void free_chan(bool cond, struct channel chan) {  
    if (cond) {  
        free(chan.buffer);  
    }  
  
    free(chan.buffer);  
    free(chan.dir_flags);  
}
```

Exemplu de defecte – eliberarea multiplă a memoriei

```
struct channel {  
    char *buffer;  
    char *dir_flags;  
};  
  
struct channel alloc_chan(int chan_id, size_t size) {  
    struct channel retval;  
    if (chan_id > 0) {  
        retval.buffer = (char *) malloc(size);  
        char flags[2] = {READ, WRITE};  
        retval.dir_flags = flags;  
    }  
  
    if (chan_id == 1) {  
        retval.buffer = NULL;  
        retval.dir_flags = NULL;  
    }  
    return retval;  
}
```

```
void free_chan(bool cond, struct channel chan) {  
    if (cond) {  
        free(chan.buffer);  
    }  
  
free(chan.buffer);  
    free(chan.dir_flags);  
}
```

Exemple de defecte – NULL dereferencing

```
struct channel {  
    char *buffer;  
    char *dir_flags;  
};  
  
struct channel alloc_chan(int chan_id, size_t size) {  
    struct channel retval;  
    if (chan_id > 0) {  
        retval.buffer = (char *) malloc(size);  
        char flags[2] = {READ, WRITE};  
        retval.dir_flags = flags;  
    }  
  
    if (chan_id == 1) {  
        retval.buffer = NULL;  
        retval.dir_flags = NULL;  
    }  
    return retval;  
}
```

```
void free_chan(bool cond, struct channel chan) {  
    if (cond) {  
        free(chan.buffer);  
    }  
  
free(chan.buffer);  
free(chan.dir_flags);  
}
```

Exemplu de defecte – NULL dereferencing

Acest tip de bug, catalogat ca fiind frecvent și catastrofic, mai poartă numele și de „greșeala de un miliard de dolari”. Sir Charles Anthony Richard Hoare își cere scuze public în anul 2009 pentru inventarea mecanismului de null reference:

„I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language ([ALGOL W](#)). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.”



Exemplu de defecte – memorie neinitializată

```
struct channel {  
    char *buffer;  
    char *dir_flags;  
};  
  
struct channel alloc_chan(int chan_id, size_t size) {  
    struct channel retval;  
    if (chan_id > 0) {  
        retval.buffer = (char *) malloc(size);  
        char flags[2] = {READ, WRITE};  
        retval.dir_flags = flags;  
    }  
  
    if (chan_id == 1) {  
        retval.buffer = NULL;  
        retval.dir_flags = NULL;  
    }  
    return retval;  
}
```

```
void free_chan(bool cond, struct channel chan) {  
    if (cond) {  
        free(chan.buffer);  
    }  
  
    free(chan.buffer);  
    free(chan.dir_flags);  
}
```

Exemple de defecte

- Erori de sincronizare
 - Deadlocks
 - Race Conditions
 - Live lock
- Erori aritmetice și de manipulări de date
 - Valoare în afara domeniului
 - Buffer overflow/underflow
 - Excepții aritmetice
 - Off by one
 - Asumții greșite privind precedența operatorilor
 - Manipularea defectuoasă a stringurilor
 - Erori de comparare (== vs. Equals, equals & hashCode)
 - Truncări de date (în urma unor operații sau în urma unei conversii (cast));
 - Indecși incorecți pentru array-uri
- Buguri de securitate (buffer overflow, SQL injection, exploiting heap, etc.)
- Operații redundante (initializări redundante, dead code, condiții redundante, etc.)

Exemple de defecte - deadlock

Thread 1:

```
synchronized (A) {  
    synchronized (B) {  
        }  
    }
```

Thread 2:

```
synchronized (B) {  
    synchronized (C) {  
        }  
    }
```

Thread 3:

```
synchronized (C) {  
    synchronized (A) {  
        }  
    }
```

Exemplu de defecte – race condition

```
class Reservation {  
    int seatsRemaining ;  
  
    public boolean reserve (int x) {  
        if (x <= seatsRemaining ) {  
            seatsRemaining -=x;  
            return true;  
        }  
  
        return false;  
    }  
}
```

În acest caz, dacă avem 2 sau mai multe thread-uri care apelează metoda `reserve(x)`, `x == 1 && seatsRemaining == 1` și întâmplător majoritatea ajung să execute simultan linia de cod `if` atunci acestea vor întoarce `true` deși în realitate nu mai era decât un singur loc disponibil.

Defectele software există. Cum validăm un produs IT?

Pentru ca un program să fie declarat produs el trebuie să satisfacă următoarele condiții:

1. **Să îndeplinească funcțiile și cerințele** pentru care a fost proiectat realizat – criteriu fidelității (produsul respectă fidel cerințele producătorului) și al validității (produsul îndeplinește cerințele utilizatorilor săi);
1. **Să îndeplinească constrângerile de timp și de calcul impuse** în etapa de proiectare (criteriul eficacității derivat din cele două criterii menționate mai sus);
1. **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 clientilor);
1. **Să nu îndeplinească nicio altă funcție în afara de cele pentru care a fost proiectat și realizat** – adică să nu conțină/fie virus, malware și să prezinte cât mai puține (d)efecte similare cu cele sus menționate (ideal ar fi niciunul).

Un program care, în timpul sau la finalul procesului de dezvoltare, nu îndeplinește criteriul (4) este 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 evitate 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 încât 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 - Asumptii de bază:

Motto (Finagle's Law of Dynamic Negatives also known as Melody's law or Finagle's corollary to Murphy's law):

Anything that can go wrong, will—at the worst possible moment.

- **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-forma(ta)te, 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 asumție cu prima asumție verificând mereu rezultatele intermediare (comparând rezultatele obținute cu valorile sau cu domeniile de valori așteptate).

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** (situațiile neprevăzute) + **cod util**;
- **Crește costul codului** = deoarece crește cantitatea de cod (cod ce care trebuie implementat și apoi întreținut);
- **Există riscul de a face „prea mult bine”** în sensul că sunt evită situațiile în care codul ar trebui să eșueze cu adevărat;

Bugurile există. Cum le combatem?

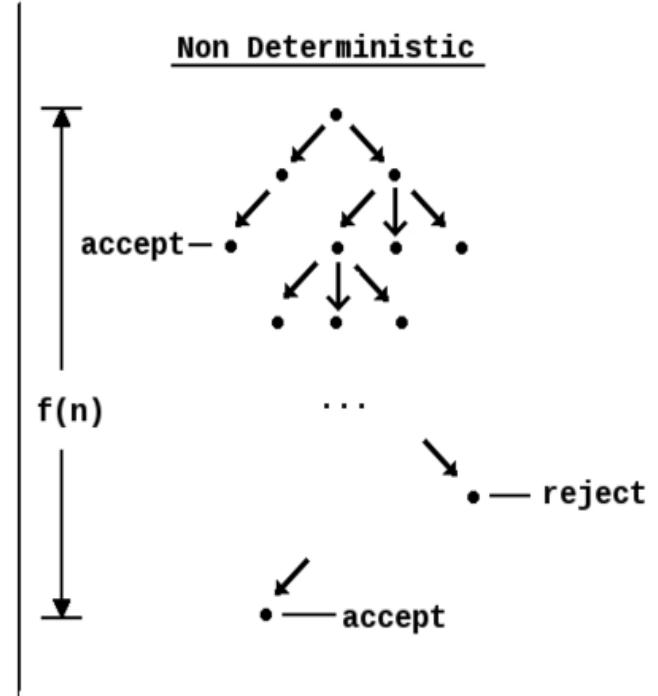
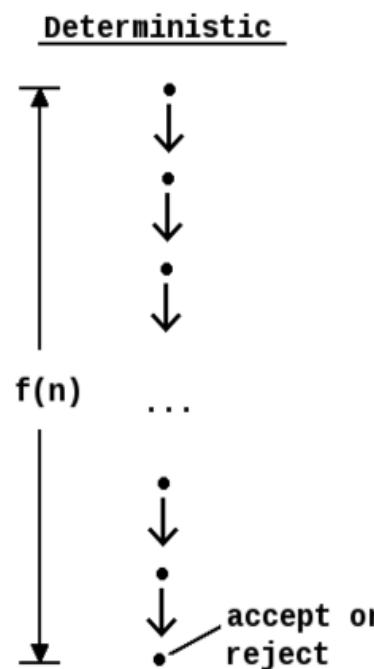
Metodologiile de dezvoltare (**development methodologies**). În acest caz, în etapele de specificare și design al aplicației se pot folosi **specificații formale complete** (care să conțină comportamentul exact al funcționalităților). În realitate această condiție este aproape imposibil de îndeplinit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

exploziei combinatorii

(o aplicație poate ajunge să crească factorial generarea sau procesarea datelor prin iteratie, prin recurență sau folosind programarea multithreading)

algoritmilor nedeterministici

(în acest caz decizia este atât de ramificată încât e imposibil de formalizat într-o manieră simplă și intuitivă).



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

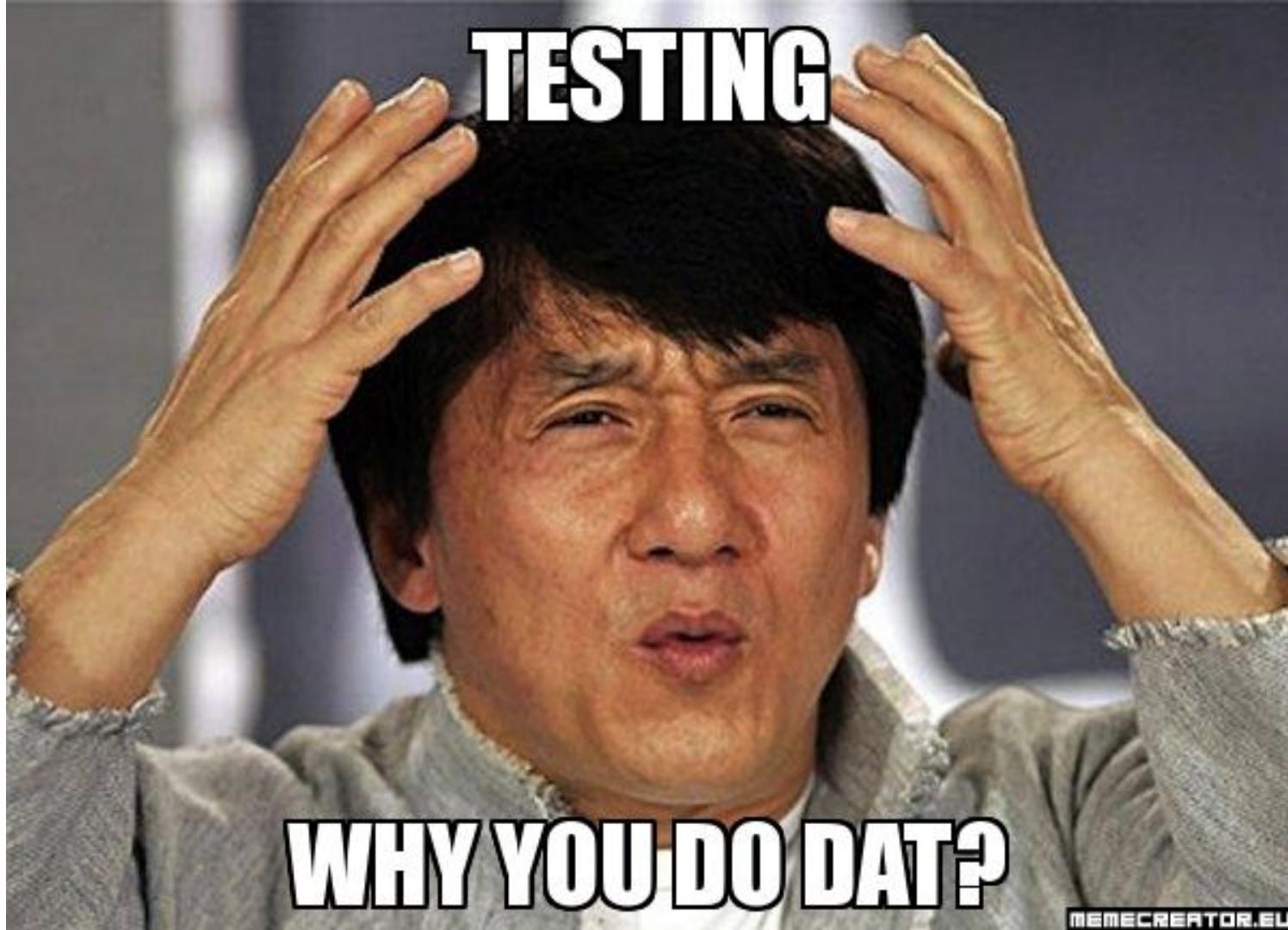
- Automate unit-testing;
- Automate acceptance testing;

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 **continuous-bug-fixing** (etapa de bug-fixing e distribuită acum omogen, pe tot parcursul dezvoltării și nu mai conduce la efectele care apărău în cazul folosirii unei metodologii de tip water-fall – întârzieri, last-minute fixes, etc.);

Testing, testing, testing

GEMINI
SOLUTIONS



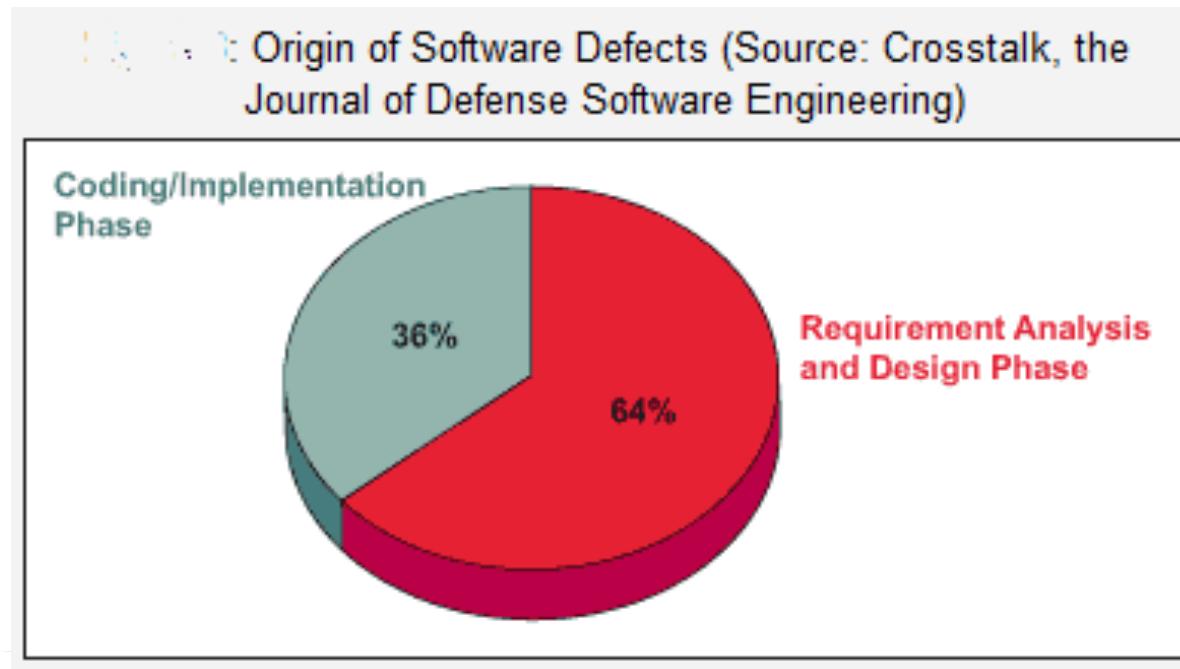
Bugurile există. Cum le combatem?

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 îndepărarea a până la 90% dintre erorile unui produs software - acest efect având loc **înaintea executării oricărui test**. **Revizuirea riguroasă** a activității softwareului **este mai eficientă** (atât d.p.d.v. economic cât și practic) **decât orice altă strategie de îndepărare a codului, inclusiv decât testarea**. Totuși IEEE subliniază faptul că revizuirea specificațiilor 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 requirements 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 funcționalității produselor):

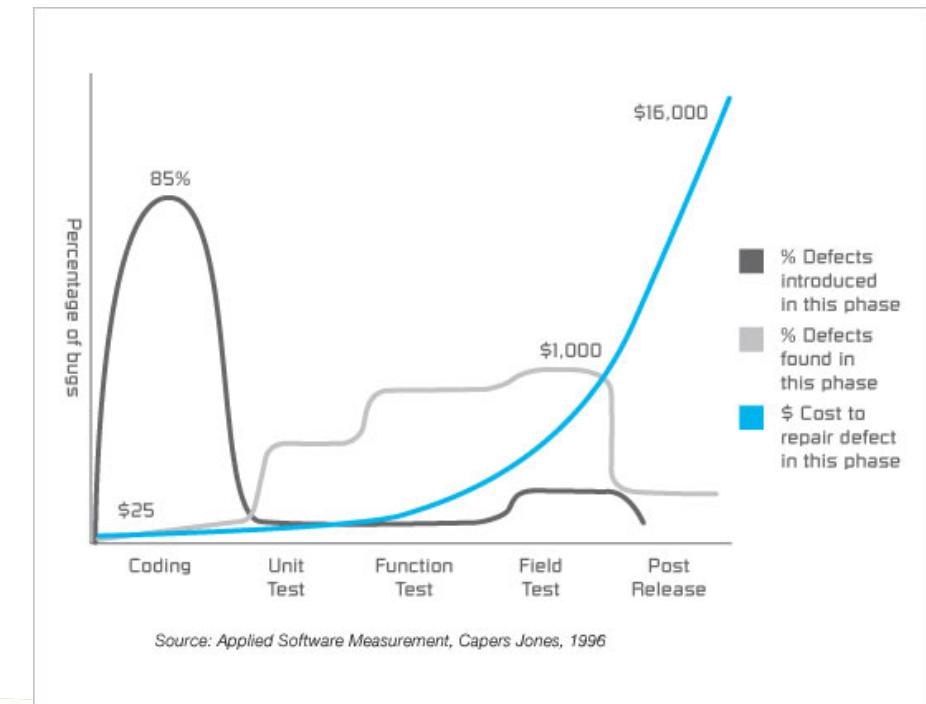
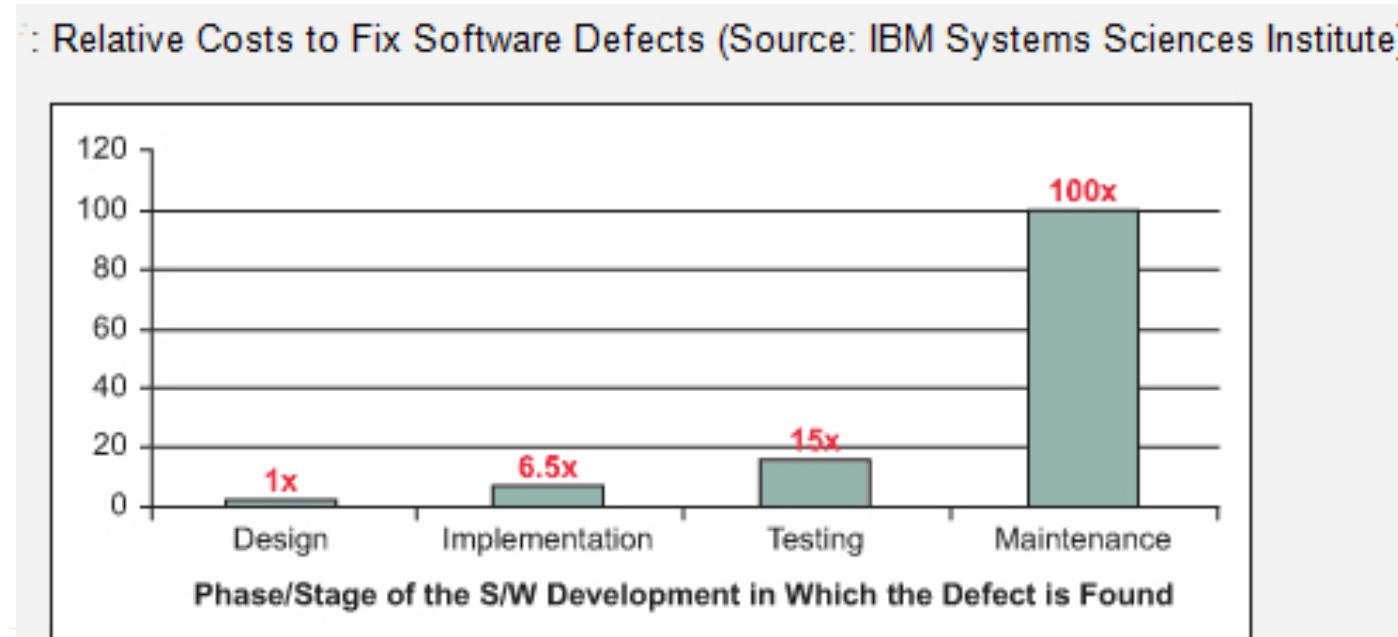


Bugurile există. Cum le combatem?

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.

Costul reparării continuă să crească exponential 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 (scăzând astfel ambiguitatea acestuia).

Exemple de mecanisme de limbaj:

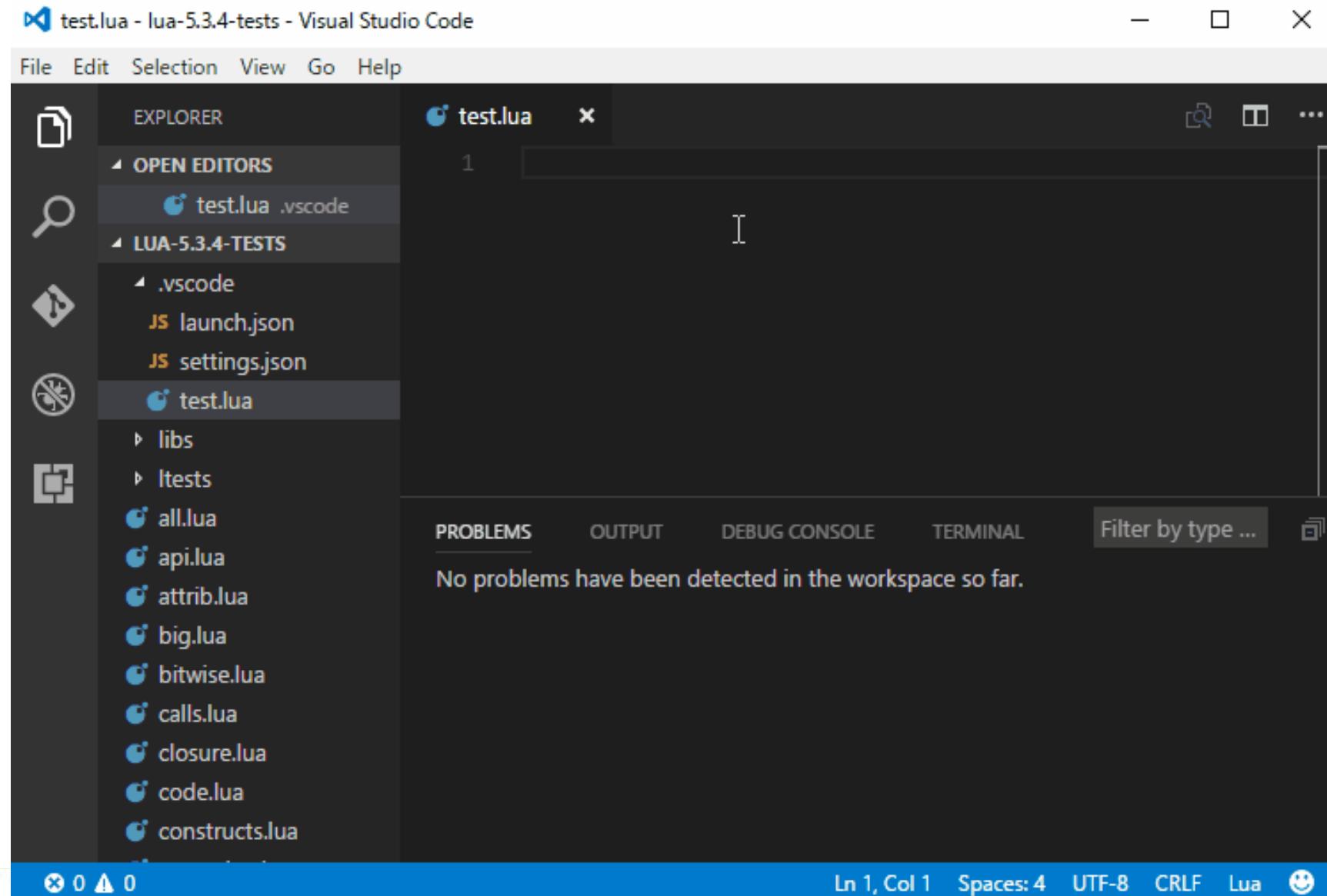
- Static type system (în majoritatea compilatoarelor);
- Namespaces și încapsularea din OOP care rezolvă o serie de ambiguități și conflicte de nume;
- Programarea modulară;
- Managementul automat al memoriei heap (garbage collector);
- Absența aritmeticii pointerilor;
- Null safety (vezi limbajul Kotlin);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

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**).

Exemplu de analiza statică în IDE: Lua și luacheck



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;

Partea a doua

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**.

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 ataşa 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 aşa-numite **debugger front-ends**, interfețe grafice (uzual din IDE-uri) care permit o serie de efecte vizuale cum ar fi execuția animată a codului, inspectia vizuală a variabilelor, etc.

Debugging / Debuggers

Aplicația ce urmează a fi depanată, poate rula în mai multe moduri:
(EXERCIȚIU: identificați care sunt avantajele și dezavantajele fiecărei modalități)

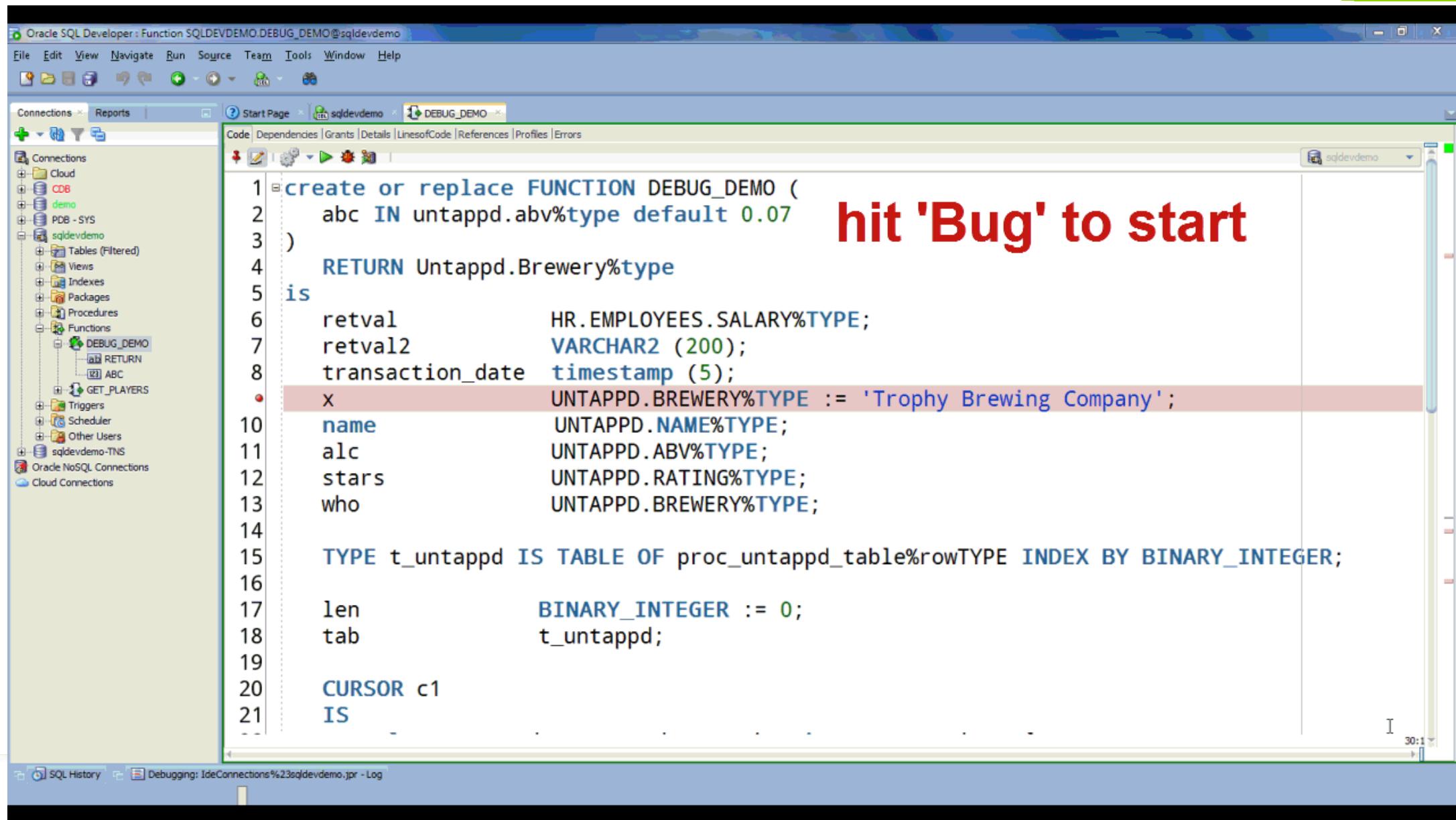
1. **Într-un mediu simulat.** În acest caz debugger-ul conține sau se folosește de un simulator;
1. **Într-un mediu real.** În acest caz debugger-ul și aplicația ce urmează a fi depanată rulează:
 1. **Pe aceeași platformă hardware/software** – acesta este cel mai întâlnit caz (încă);
 1. **Pe o altă platformă hardware/software** identică sau diferită situată la distanță. 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”;

Debugging-ul unei proceduri stocate în PL/SQL



The screenshot shows the Oracle SQL Developer interface. The title bar reads "Oracle SQL Developer : Function SQLDEVDEMO.DEBUG_DEMO@sqldevdemo". The menu bar includes File, Edit, View, Navigate, Run, Source, Team, Tools, Window, Help. The toolbar has various icons for file operations. The left sidebar shows connections to Cloud, CDB, demo, PDB - SYS, and sqpdevdemo, with sqpdevdemo expanded to show Tables (Filtered), Views, Indexes, Packages, Procedures, Functions, Triggers, Scheduler, Other Users, and sqpdevdemo-TNS. The main workspace displays the PL/SQL code for the DEBUG_DEMO function. A red rectangular callout box highlights the line of code "UNTAPPD.BREWERY%TYPE := 'Trophy Brewing Company';". The text in the callout box is bolded and colored red. The code itself is in blue and black font. The bottom status bar shows "SQL History" and "Debugging: IdeConnections%23sqldevdemo.jpr - Log".

```
1 create or replace FUNCTION DEBUG_DEMO (
2     abc IN untappd.abv%type default 0.07
3 )
4     RETURN Untappd.Brewery%type
5
6     retval          HR.EMPLOYEES.SALARY%TYPE;
7     retval2         VARCHAR2 (200);
8     transaction_date timestamp (5);
9
10    x               UNTAPPD.BREWERY%TYPE := 'Trophy Brewing Company';
11    name            UNTAPPD.NAME%TYPE;
12    alc             UNTAPPD.ABV%TYPE;
13    stars           UNTAPPD.RATING%TYPE;
14    who             UNTAPPD.BREWERY%TYPE;
15
16    TYPE t_untappd IS TABLE OF proc_untappd_table%rowTYPE INDEX BY BINARY_INTEGER;
17
18    len              BINARY_INTEGER := 0;
19    tab              t_untappd;
20
21    CURSOR c1
22        IS
```

hit 'Bug' to start

Multithreading Debugging in C#

Any CPU Continue Lifecycle Events Thread: [16328] Worker Thread Stack Frame: NumberUtilities.IsPrime

NumberUtilities.cs

ProjectArchive.SharedLibrary

NumberUtilities

```
31     }
32 }
33 public static bool IsPrime(int candidate)
34 {
35     if ((candidate & 1) == 0)
36     {
37         if (candidate == 2)
38             {
```

candidate 1

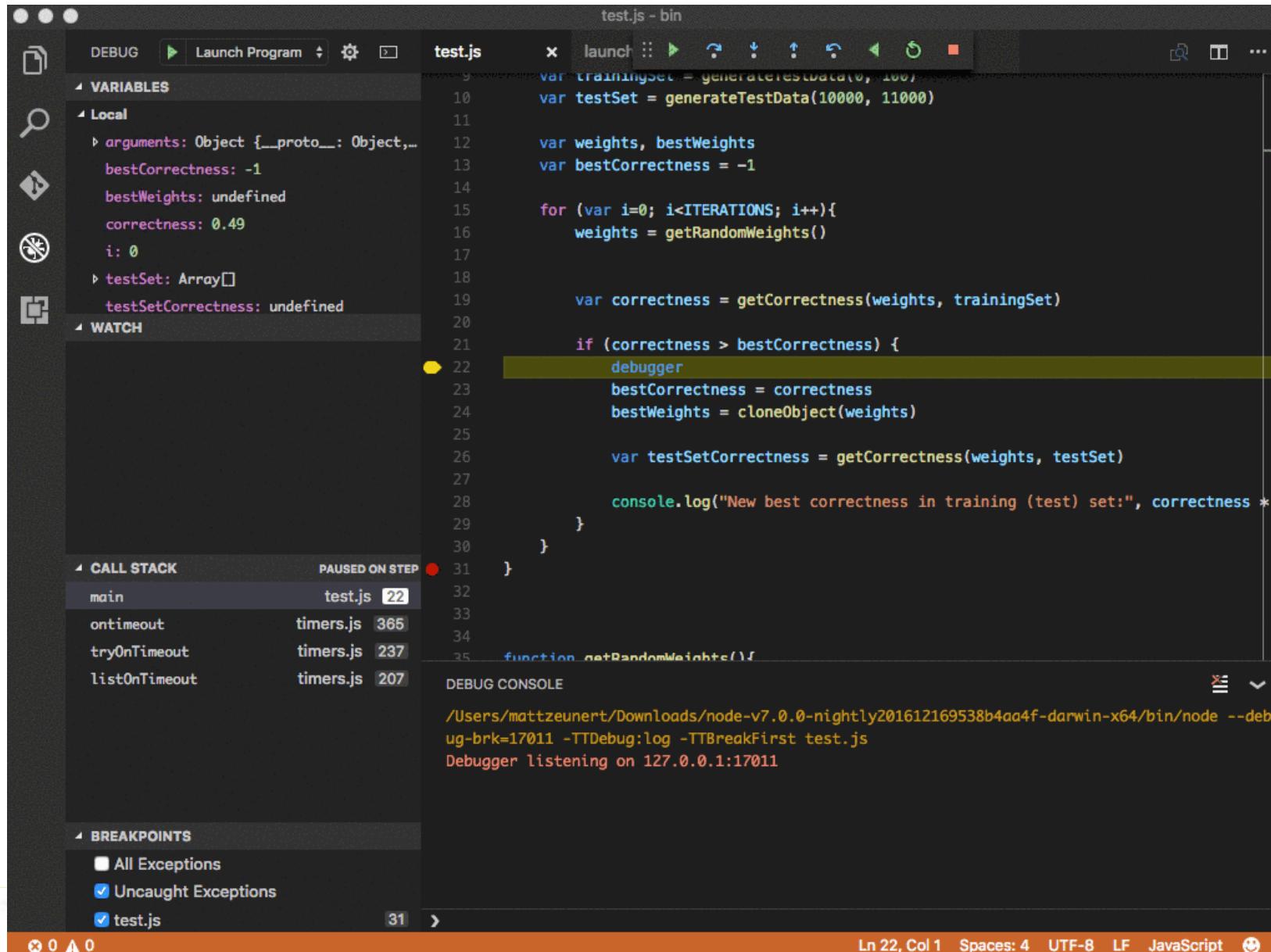
2 references

≤ 0ms elapsed

Debugging: Run to Cursor in Visual Studio 2017

```
{  
    /*----- Let's Look at Run to Click!*/  
    Student s = new Student("ScottGu", 4.0);  
  
    //When I click on the Glyph My Code Runs to that line  
    List<Teacher> MyTeachers = GenerateTeachers();  
    MyTeachers.ForEach(s.AddTeacher);  
    if (s == null)  
    {  
        //Caution: Run to Click can finish your application if you click  
        // a line that will not be hit in your code execution  
        Retreat();  
    }  
    return 0;  
}  
1 reference | 0 changes | 0 authors, 0 changes  
private static List<Teacher> GenerateTeachers()  
{  
    List<Teacher> list = new List<Teacher>();  
  
    //I can even Run to Click in different Methods  
    Teacher a = new Teacher("Scott Hanselman", "Life");  
    Teacher b = new Teacher("Maria Nagagga", "Happiness");  
    Teacher c = new Teacher("John Montgomery", "Business");  
    a.Ranking = 10;  
    b.Ranking = 10;  
    c.Ranking = 6;
```

Reverse/Time-Travel Debugging in Visual Studio Code



The screenshot shows the Visual Studio Code interface with the following details:

- File:** test.js
- Editor:** Code editor showing the following snippet of JavaScript:

```
test.js - bin
var trainingSet = generateTestData(0, 100);
var testSet = generateTestData(10000, 11000);

var weights, bestWeights;
var bestCorrectness = -1;

for (var i=0; i<ITERATIONS; i++){
    weights = getRandomWeights()

    var correctness = getCorrectness(weights, trainingSet)

    if (correctness > bestCorrectness) {
        debugger
        bestCorrectness = correctness
        bestWeights = cloneObject(weights)
    }

    var testSetCorrectness = getCorrectness(weights, testSet)

    console.log("New best correctness in training (test) set:", correctness *
}
```

- Variables:** Local variables shown in the Variables sidebar:
 - arguments: Object {__proto__: Object, ...}
 - bestCorrectness: -1
 - bestWeights: undefined
 - correctness: 0.49
 - i: 0
 - testSet: Array[]
 - testSetCorrectness: undefined
- Call Stack:** Shows the current stack trace:

main	test.js	22
ontimeout	timers.js	365
tryOnTimeout	timers.js	237
listOnTimeout	timers.js	207
- Breakpoints:** Breakpoints section showing "All Exceptions" and "Uncaught Exceptions" selected.
- Console:** DEBUG CONSOLE output:

```
/Users/mattzeunert/Downloads/node-v7.0.0-nightly201612169538b4aa4f-darwin-x64/bin/node --de
bug-brk=17011 -TTDebug:log -TTBreakFirst test.js
Debugger listening on 127.0.0.1:17011
```
- Status Bar:** Shows file path (Ln 22, Col 1), file type (JavaScript), and encoding (UTF-8).

Un **breakpoint** reprezintă 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.

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-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.

Exemplu de breakpoint conditional simplu

The screenshot shows the DevTools Conditional Breakpoint panel. The code in the editor is:

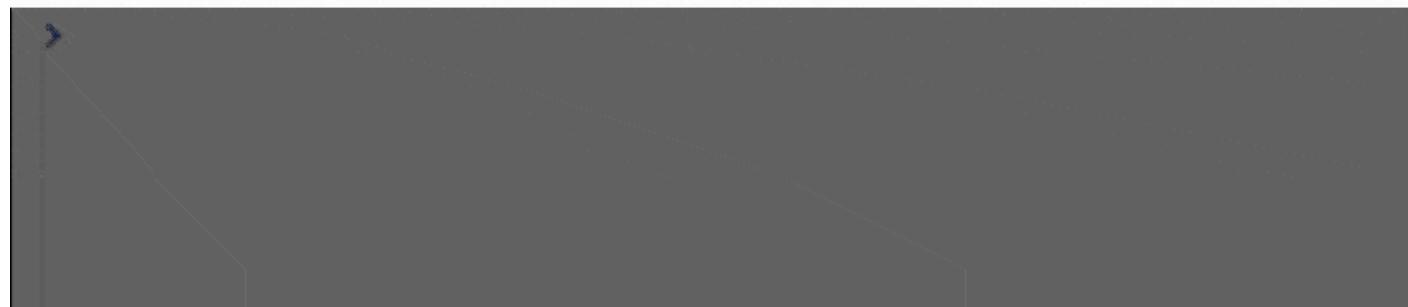
```
1
2 window.foo = function(num) {
3     num;
4 }
```

The breakpoint is set on line 2. The status bar at the bottom right says "Not Paused".

DevTools: Conditional Breakpoints

Dev Tips

umaar.com/dev-tips @umaar



Exemple de events breakpoint-uri:

The screenshot shows the Chrome DevTools interface with the 'Sources' tab selected. On the left, the file tree shows 'questionFormController.js' under 'post/abdulapopoola.com'. The main pane displays the code for this file, with a blue selection bar highlighting the 'answered' function. The right pane contains the 'Breakpoints' section, which is expanded to show a single entry: 'Script snippet #1:44' with the condition 'stack[i].next = null;'. Other sections like 'Call Stack', 'Scope', and 'Event Listeners' are also visible.

```
17 this.optionsKeys = ['A', 'B', 'C', 'D'];
18
19 this.answered = answered;
20 this.nextQuestion = nextQuestion;
21 this.prevQuestion = prevQuestion;
22 this.submitQuestion = submitQuestion;
23
24 function answered (question) {
25     var questionIndex = question.entryPosit
26     PubSub.publish(EVENTS.QUESTION_ANSWERED
27 }
28
29 function nextQuestion () {
30     PubSub.publish(EVENTS.SELECT_NEXT_QUEST
31 }
32
33 function prevQuestion () {
34     PubSub.publish(EVENTS.SELECT_PREV_QUEST
35 }
36
37 function submitQuestion () {
38     PubSub.publish(EVENTS.SUBMIT_EXAM, that
39 }
40 }
```

{ } 2 lines, 35 characters selected

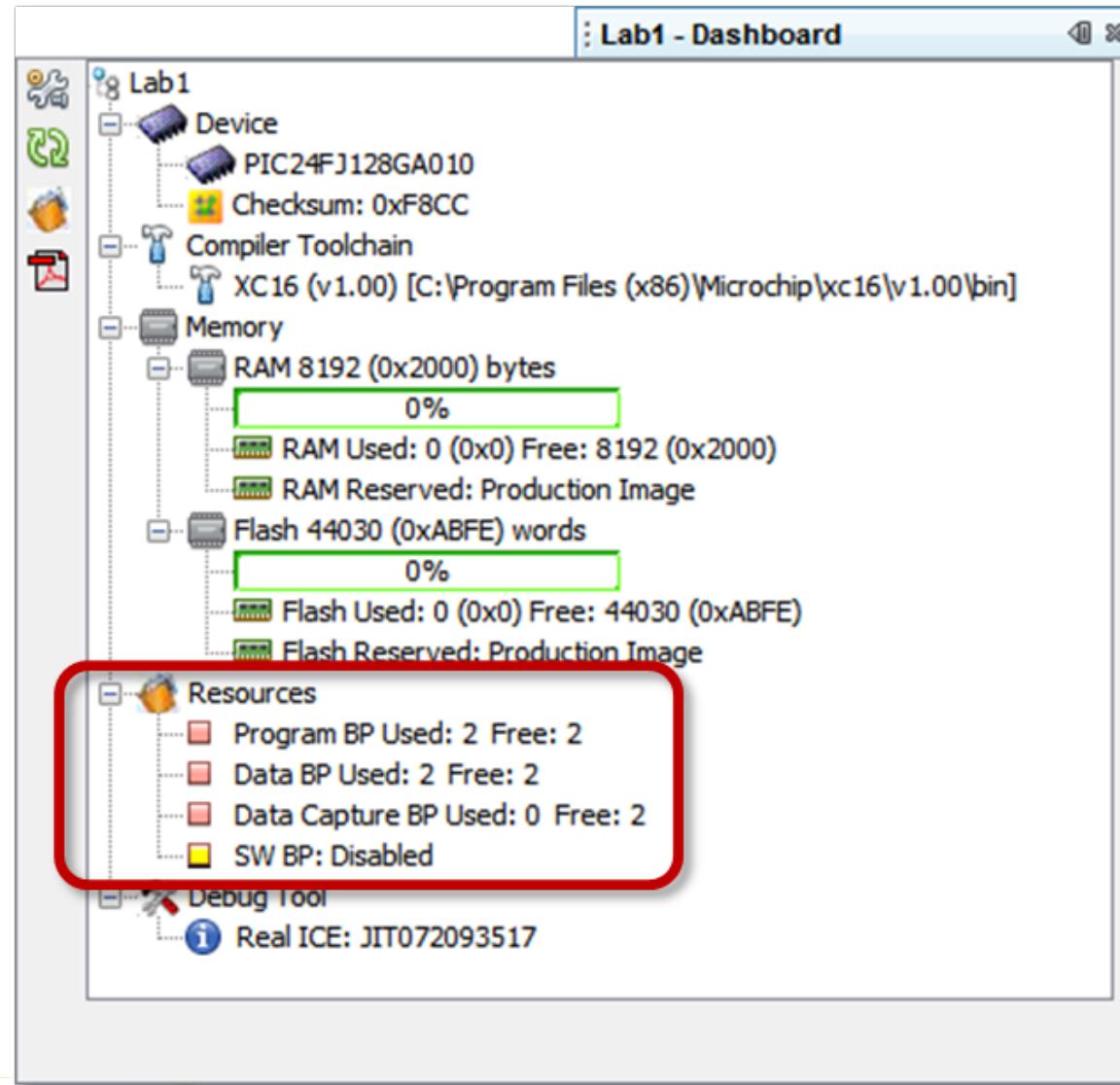
Breakpoint-uri hardware. Diferite procesoare oferă o serie de mecanisme hardware ce permit debugging-ul programului sau al memoriei: registrii de debug, flag-uri în registrii de control care permit alterarea funcționării procesorului (de exemplu: se evită execuția de tip branch delay slot = atunci când două sau mai multe instrucțiuni în asamblare sunt executate simultan deoarece acestea sunt independente între ele). Fiind implementate direct în hardware, aceste tipuri de breakpoint-uri sunt limitate numeric (uzual 4);

Breakpoint-urile software. Acestea sunt implementate simplu prin înlocuirea on-the-fly a locației în care se găsește breakpointul cu:

O instrucțiune care va apela direct debugger-ul (frecvent un syscall);

O instrucțiune invalidă ce va provoca întreruperea intenționată a programului și executarea unei rutine de tratare a acelei întreruperii. Aceasta rutină este la rândul ei interceptată/gestionată de către debugger.

Breakpoint-uri hardware în microcontrollere

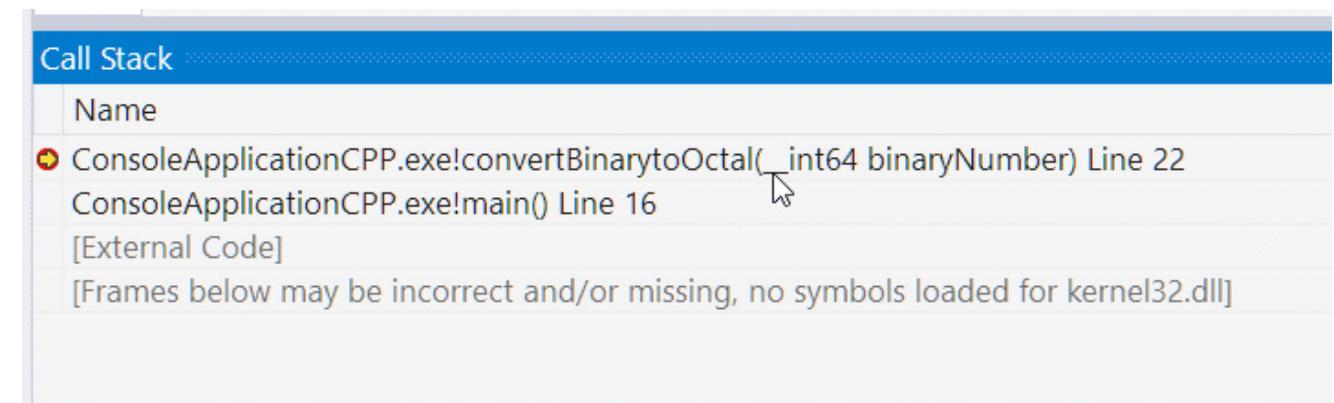
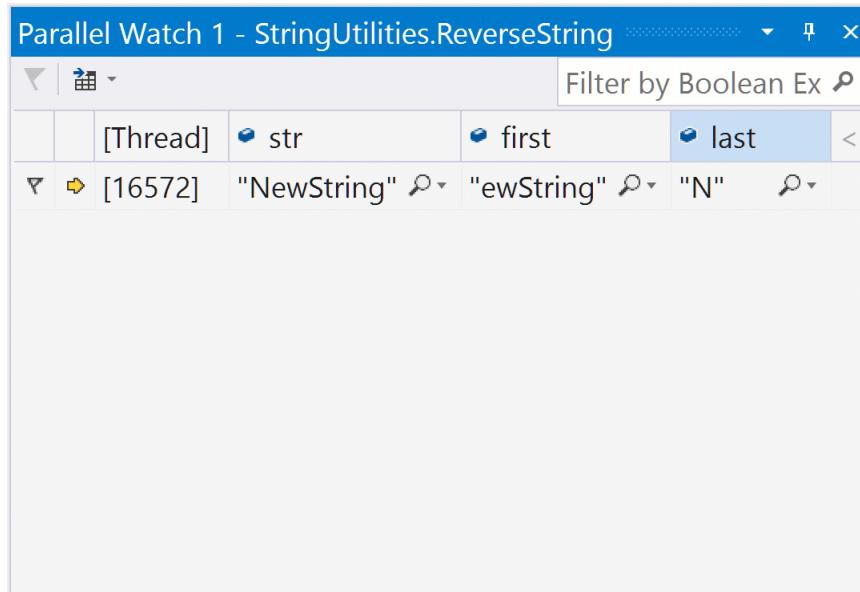


Mecanisme avansate de debugging

Ultimele versiuni de IDE-uri permit o serie de operații avansate de debugging cum ar fi:

- Vizualizarea anumitor variabile locale la apeluri successive ale aceleiași funcții;
- Vizualizarea parametrilor funcțiilor în stiva de apeluri;
- Breakpoint asupra unei metode/excepții folosind doar numele acesteia (atunci când utilizatorul nu are acces la codul sursă al acesteia);
- Breakpoint atunci când o locație de memorie își schimbă valoarea (data breakpoint);
- Breakpoint-uri pentru loop-uri;
- Executarea unor acțiuni în momentul în care un breakpoint este apelat;
- Importul și exportul de breakpointuri dintr-un proiect;
- Partajarea breakpointurilor între membrii unei echipe;

Exemplu: „Parallel Watch” și „Show parameter values in the Call Stack”



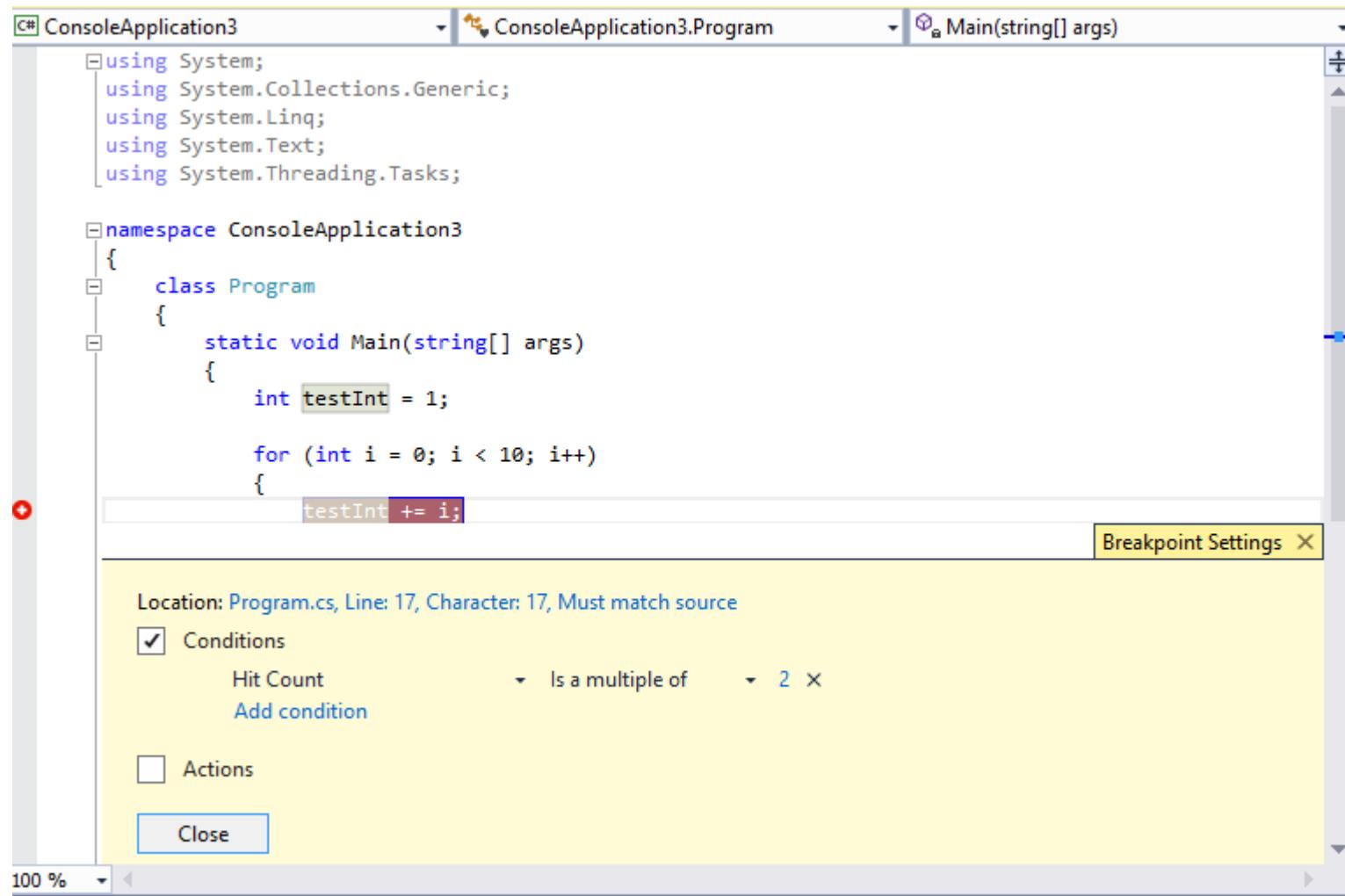
Exemplu: „Function Breakpoint” și „Data Breakpoint”

The screenshot shows the Microsoft Visual Studio IDE interface. The menu bar at the top includes File, Edit, View, Project, Build, Debug (which is highlighted in blue), Team, Tools, Test, Analyze, Window, and Help. Below the menu is a toolbar with various icons. A dropdown menu labeled "Debug" is open, showing options like "Break", "Break All", "Stop", and "Stop All". The main code editor window displays the following C++ code:

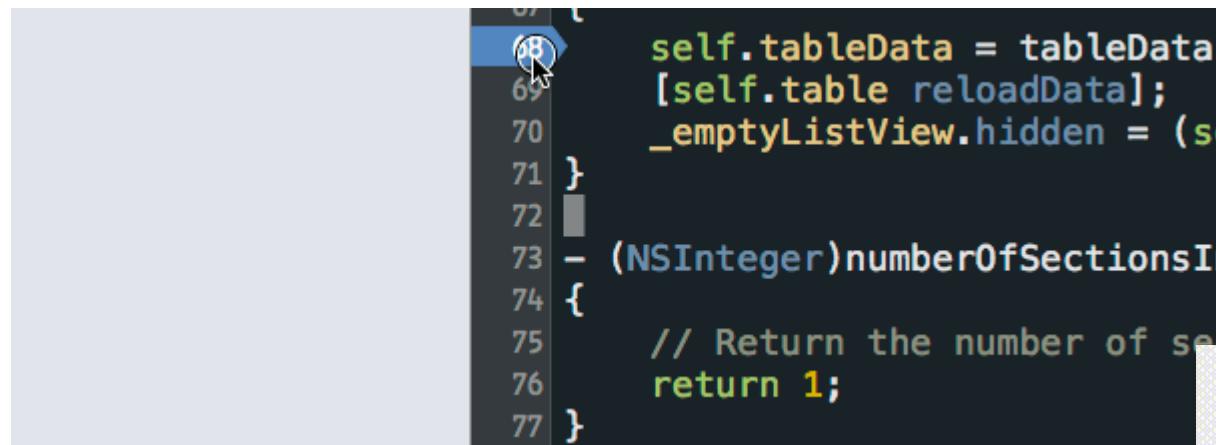
```
19 class CObject1 : public CRefCount
20 {
21     public:
22     CObject1(int myint)
23     {
24         m_myint = myint;
25     }
26 }
```

A yellow arrow icon, representing a breakpoint, is located on the left margin next to line 26. At the bottom of the screen, there is a "Breakpoints" window with a toolbar containing buttons for New, Labels, Condition, Hit Count, and Procedure. The window also has search and filter options.

Exemplu de „Hit count breakpoint”

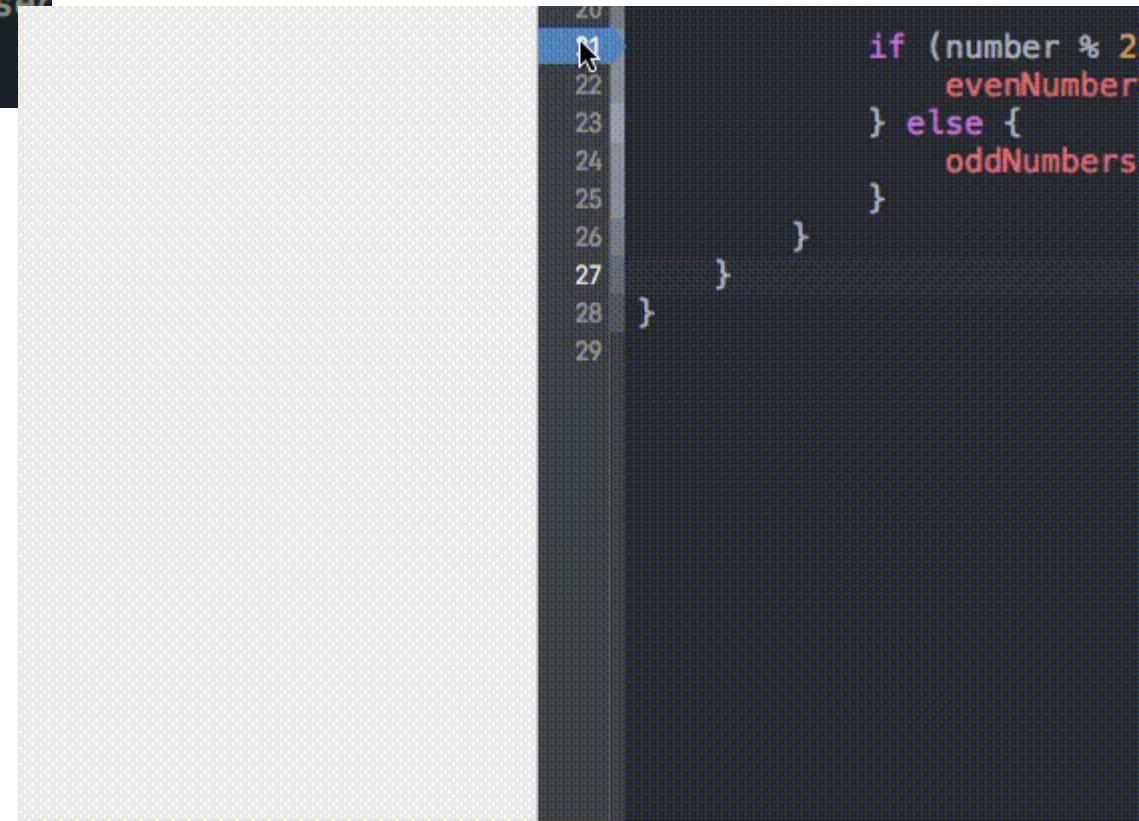


Exemplu de breakpoint care produce acțiune



A screenshot of a debugger interface showing a code editor. A blue horizontal bar indicates a breakpoint at line 68. The code is written in Objective-C:

```
67    self.tableData = tableData;
68    [self.table reloadData];
69    _emptyListView.hidden = (se
70 }
71
72 -
73 - (NSInteger)numberOfSectionsIn
74 {
75     // Return the number of sec
76     return 1;
77 }
```



A screenshot of a debugger interface showing a code editor. A blue horizontal bar indicates a breakpoint at line 20. The code is written in Swift:

```
20 if (number % 2
21     evenNumber
22 } else {
23     oddNumbers
24 }
25
26 }
27 }
28 }
29 }
```