**Cuprins**

* Software pentru sisteme embeded
* Aplicatii embedded
* Exemple de sisteme embedded
* Unelte de dezvoltare a software-ului pentru sisteme integrate
* Notiunea de firmware
* Arhitecturi software pentru sisteme embedded
* Modelare UML
* Baze de date relationale
* Tehnologii folosite
* Programarea orientata pe obiecte
* Platforma Netbeans
* Swing in Java
* Modele UML pentru calculatorul de bord
* Diagrama de clase generala
* Diagrame de clase individuale
* Diagrama de cazuri de utilizare
* Diagrama de componente (pachete)
* Descrierea simulatorului
  + Calculul de coordonate pentru cele trei indicatoare pentru controlul manual pornit
  + Calculul vitezei si turatiei
  + Explicarea preluarii mesajelor vocale folosind google si progamul de taiere/editare audio
  + Explicarea preluarii datelor pentru parcurgerea scenariilor din baza de date / Prezentarea scenariilor
  + Interfata cu utilizatorul
  + Folosirea tehnicii Double Buffering
  + Folosirea Apache Derby Network Server

**Aplicatii embedded**

**Ce face aplicatiile embedded sa functioneze**

Sistemele integrate sunt peste tot. Nu poti sa scapi de ele. Intr-o casa obisnuita din America, sunt in jur de 40 de microprocesoare, fara sa tinem cont de calculatoarele personale (care contribuie cu 5 – 10 fiecare), sau masini (care in mod normal contini cateva duzine), iar predictia pentru urmatoarea decada este ca aceste cifre vor creste cu cateva ordine. Este oarecum ironic, pentru ca majoritatea oamenilor din afara domeniului electronicii nu au absolut nici o idee despre ceea ce inseamna de fapt „embedded”.

Persoanele din departamentul de marketing tin la segmentarea pietelor. Teoria in acest sens este ca o asemenea analiza segmentata va duce la obtinerea de produse mult mai bune, prin atingerea tuturor cerintelor din fiecare segment intr-o maniera specifica. Pentru produsele integrate, obtinem astfel segmente ca telecomunicatie, control al procesului, clienti sau auto. Insa de curand, din ce in ce mai multe produse apar ce nu respecta acest model. De exemplu, un telefon cu camera video este un produs orientat mai mult catre client sau catre telecomunicatie. Adevarul este ca nu are prea multa importanta. O consideratie generala ce ar putea fi facuta in aceasta zona, ar fi cu siguranta legata de cat de comune sunt astfel de aplicatii. Comentariul principal ce l-am putea face despre toate ar fi cantitatea de software din cadrul fiecarui dispozitiv, ce creste din ce in ce mai mult.

O data ce microprocesoarele au devenit din ce in ce mai mici si ieftine, din ce in ce mai multe produse au microprocesoare integrate in ele pentru a le face „destepte”. Asemenea produse precum DVD-playerele, ceasurile digitale, lifturile, motoarele de automobile, termostatele, echipamente de control industrial si instrumente stiintifice si medicale, sunt conduse de microprocesoarele lor si de software. Oamenii folosesc termenul de sistem integrat pentru a face referire la orice sistem ascuns din cadrul acestor produse.

Software-ul pentru sistemele integrate trebuie sa rezolve multe probleme noi aparute fata de cele specifice aplicatiilor software pentru calculatoare personale sau servere. Sistemele integrate au de obicei mai multe lucruri de facut la un moment de timp. Ele trebuie sa raspunda la evenimente externe (de exemplu cineva apasa un buton dintr-un lift). Ele trebuie sa faca fata tuturor conditiilor neobisnuite aparute fara interventia oamenilor. Ceea ce fac ele este de asemenea tot timpul constrans de termeni limita.

**Exemple de sisteme embedded**

Pentru a intelege mai bine problema software-ului pentru sisteme integrate, is pentru a face problemele putin mai contrete, sa incepem prin a examina cateva sisteme simple. Vom face tot timpul referire la aceste exemple din cand in cand pentru a face referire la probleme si solutii specifice.

**Telegraful**

Primul sistem pe care il vom analiza este unul care a fost denumit simbolic „Telegraf”, in timpul dezvoltarii sale. Telegraful este ceea ce ne permite sa facem legatura dintre o imprimanta ce are un singur port de mare viteza la o retea. Din afara, telegraful este o cutie mica de plastic, avand 2 sau 3 inci pe o parte, si de grosime de aproape jumatate de inci.Un cablu pe una dintre parti intra in portul serial al imprimantei. Un conector aflat pe cealalta parte a cutiei intra in retea. O schita a Telegrafului este afisata in figura de mai jos :



Conector la retea

Conector la imprimanta

Evident, Telegraful trebuie sa primeasca informatii de la retea si sa le copieze pe portul serial. Cu toate acestea, Telegraful este de fapt mai complicat decat atat. In continuare sunt prezentate cateva dintre lucrurile pe care Telegraful trebuie sa le faca:

* Pe retea, informatiile ajung uneori in dezordine, date sunt cateodata pierdute pe drum, iar anumite date ajung cateodata de doua ori. Telegraful trebuie sa ordoneze acest haos pe retea si sa asigure un flux de date curat catre imprimanta.
* Este posibil ca in retea sa fie multe calculatoare, toate dintre acestea putand realizeze o operatie de printare in acelasi moment. Imprimanta se „asteapta” sa fie in legatura cu un singur calculator. Telegraful trebuie sa ofere imprimantei o singura operatie de imprimare la un moment de timp, si cateodata sa tina in asteptare toate celelalte calculatoare.
* Imprimantele de retea trebuie sa ofere informatii de stare catre orice calculator din retea ce solicita acest lucru, chiar daca ele sunt ocupate in cadrul unei operatii de printare pentru un alt calculator. Imprimanta initiala cu un port serial nu poate face acest lucru. El trebuie fi facut de Telegraf.
* Telegraful trebuie sa poata lucra cu mai multe tipuri de imprimante fara a avea nevoie de configurari speciale. Telegraful trebuie sa recunoasca tipul de imprimanta la care este atasat.
* Telegraful trebuie sa poata raspunda rapid la anumite evenimente. Acestea sunt, de exemplu, diferite tipuri de cadre de retea la care Telegraful trebuie sa raspunda in sub 200ms.
* Telegraful trebuie sa tina cont permanent de timp. De exemplu, daca un calculator care transmitea informatii legate de imprimare catre Telegraf se intrerupe, Telegraful trebuie la un moment dat sa renunte la acea operatie de printare – poate dupa 2 minute – si sa printeze de la alt calculator din retea. Altfel, stricarea unui computer ar face imprimanta indisponibila fata de toate calculatoarele.

**Probleme legate de dezvoltarea Telegrafului**

Pentru a satisface lista de cerinte de mai sus, Telegraful are un microprocesor integrat in el. Software-ul sau este mai extensiv si mai complicat decat ne-ar putea da de inteles aparenta sa exterioara. Ce probleme apar in dezvoltarea unui astfel de software?

Pentru inceput, desigur, software-ul pentru Telegraf trebuie sa fie corect din punct de vedere logic. Nu trebuie niciodata sa „uite” de la care calculator printeaza, sa renunte la informatii sau sa ofere informatii de stare incorecte. Aceasta este o cerinta prezenta in fiecare componenta software atat in sisteme integrate cat si in cadrul aplicatiilor.

Cu toate acestea, scrierea software-ului pentru sisteme de tip Telegraf, sau pentru multe alte sisteme integrate – ofera diferite probleme aditionale, care insa nu vor fi discutate.

**Primirea informatiei**

Imprimanta poate imprima maxim la viteza la care ii sunt transmise date de la Telegraf. Telegraful nu trebuie sa fie supus fenomenului de congestie („bottleneck” ), intre calculatoarele din retea si imprimanta. De cele mai multe ori, problema transmiterii informatiilor catre sistemul integrat, este similara cu incercarea cresterii vitezei de rulare a unei aplicatii obisnuite. Solutia va fi regasita prin programare inteligenta: cautare si sortare mai performanta, algoritmi numerici mai buni, structuri de date ce pot fi manipulate mai usor, si asa mai departe. Desi aceste tehnici depasesc scopul nostru, vom lua in discutie greseile posibile facute folosind sisteme de operare in timp real.

**Raspunsul**

Cand un cadru de retea important este primit, Telegraful trebuie sa raspunda in sub 200ms, chiar daca la momentul de timp respectiv face altceva. Software-ul trebuie sa fie scris pentru a permite acest lucru. Vom pune accent pe discutarea problemei raspunsului, pentru ca reprezinta o problema obisnuita in sistemele integrate, datorata gradului mai mare sau mai mic de compromis prezent in toate solutiile gasite.

Oamenii folosesc des termenul vag de viteza. Cu toate acestea, arhitectii de sisteme integrate trebuie sa intampine doua probleme diferite – primirea informatiei si raspunsul – iar tehnicile de tratare a lor nu sunt deloc aceleasi. De fapt, rezolvarea uneia dintre probleme de multe ori face ca cealalta sa se agraveze. De aceea, in informatiile oferite in continuare vom face referire doar le termenii de „primire a informatiei” si „raspuns”, evitand termenul de „viteza”.

**Capacitatea de testare**

Nu este deloc usor sa determinam daca Telegraful efectiv functioneaza. Problema este ca o mare parte din cod intampina evenimente neobisnuite. Telegraful este in aceasta privinta un sistem integrat tipic, pentru ca aceste sisteme trebuie sa poata intampina cu succes orice eveniment exterior fara interventia omului. De exemplu, o mare parte din codul Telegrafului, este dedicat problemei de pierdere a informatiei in retea. Totodata, informatia nu se pierde in retea foarte des, in special intr-un laborator de testare, unde probabil reteaua este setata foarte bine, este compusa din componente nou noute, si nu are o lungime mai mare de 3 metri. Acest lucru face dificila testarea tuturor acelor linii de cod.

In mod similar, Telegraful trebuie sa se ocupe de evenimente care sunt aproape simultane. Daca doua calculatoare trimit o cerere pentru pornirea procesului de imprimare exact in acelasi moment, de exemplu, poate face fata software-ul? Telegraful contine cod care rezolva aceasta situatie, dar cum putem simula conditiile necesare pentru testarea acestui cod?

**Capacitatea de depanare software**

Ce credeti ca se intampla de obicei cand prin procesul de testare se descopera o problema in software-ul Telegrafului? Telegraful nu are un ecran; nu are o tastatura; nu are boxe; nici macar mici luminite pentru semnalizare. Cand apare o problema, nu suntem atentionati prin icoane dragute sau ferestre in anumite locuri. In scchimb, Telegraful pur si simplu nu mai functioneaza. O problema in sistemul de retea? O problema in codul ce verifica ce calculator efectueaza o operatie de printare? O problema in codul ce ofera informatii de stare? Telegraful pur si simplu nu mai functioneaza.

Din pacate, oprirea functionarii Telegrafului nu ne ofera multe informatii despre problema aparuta. Mai mult, fara o tastatura sau un ecran, nu putem rula un program de depanare pe Telegraf. Trebuie gasite astfel alte metode prin care sa ne dam seama ce s-a intamplat. Tehnici de depanare software pentru sistemele integrate vor fi astfel luate in discutie precum si tehnici de chiar evitare a unor probleme dificile specifice.

**Fiabilitatea**

Precum celelalte sisteme integrate, Telegraful nu trebuie sa se strice. Desi unii utilizatori par sa aiba o anumita toleranta pentru sistemele de tip desktop, ce trebuie repornite din cand in cand, nimeni nu are rabdare cu cutii mici de plastic care se pot bloca din cand in cand. In anumite situatii mai ciudate, software-ul aplicatiei poate afisa un mesaj pe ecran , intreband utilizatorul ce doreste sa faca. Sistemele integrate nu au aceasta optiune; Indiferent ce se intampla, software-ul trebuie sa functioneze fara interventie umana.

**Spatiul de memorie**

Telegraful are doar o cantitate finita de memorie, in mod particular, 32KB de memorie pentru program si 32KB de memorie pentru date. Doar atata memorie poate avea dispozitivul, ca pretul acestuia sa ramana rezonabil. Desi memoria este din ce in ce mai ieftina, aceasta nu este totusi inca gratis. A face software-ul sa incapa in spatiul de memorie disponibil este o capacitate obligatorie pentru multi ingineri software.

**Instalarea programului**

Software-ul din Telegraf nu a ajuns acolo pentru ca cineva a facut click pe un mouse cu cursorul pe o icoana. Uneltele speciale necesare pentru instalarea programelor in dispozitive integrate vor fi discutate in cele ce urmeaza.

**Scanerul de coduri de bare fara fir**

Sa luam in considerare un alt exemplu de sistem integrat, un scaner pentru coduri de bare fara fir. Atunci cand utilizatorul apasa pe butonul acestuia, scanerul isi activeaza laserul pentru a citi codul de bare si apoi trimite acest cod de bare prin unde radio la casa de marcat. Acest lucru poate fi observat in figura de mai jos:



**1.Utilizatorul apasa pe buton**

**2.Laserul citeste codul de bare**

**3.Codul de bare este trimis prin unde radio casei de marcat**

Cum se aseamana problema dezvoltarii software-ului pentru scanerul de coduri de bare fara fir, cu cel pentru Telegraf?

Ca raspuns, am putea spune ca cele doua sunt in mare similare. O problema ce nu o regasim in cazul scanerului este cea a primirii datelor. Pur si simplu nu exista foarte multa informatie intr-un cod de bare, iar utilizatorul nu poate apasa butonul suficient de repede. Pe de alta parte, scanerul fara fir are o problema ce Telegraful nu o are.

**Consumul**

Avand in vedere ca scanerul ete fara fir, bateria este singura sa sursa de energie, iar avnad in vedere ca scanerul este un dispozitiv ce urmeaza a fi manipulat, greutatea acestei baterii este limitata de o greutate medie ce poate fi ridicata confortabil de un utilizator. Cat vrea clientul sa tina bateria? Raspunsul evident – pentru totdeauna – nu este fezabil. Care ar fi urmatorul raspuns bun.

Urmatorul raspuns bun ar fi ca bateria sa tina un schimb de lucru de 8 ore. Dupa aceea, scanerul poate fi plasat intr-un toc aflat la marginea casei de marcat pe parcursul noptii pentru a-si reincarca bateria. Se pare, cu toate acestea, ca nici acest lucru nu este fezabil pentru activarea unui laser, a unui microprocesor, a unei memorii si a unei surse radio pentru 8 ore depinzand doar de energia asigurata de baterie. Asadar, una din principalele probleme a acestui software este gasirea componentelor hardware ce nu sunt necesare la anumite momente de timp, si dezactivarea acestora. Chiar si procesorul poate intra in aceasta categorie, asa cum vom vedea in continuare.

**Imprimanta laser**

Un alt sistem integrat este imprimanta laser. Majoritatea imprimantelor laser au un numar rezonabil de microprocesoare integrate pentru a asigura functionarea diferitelor aspecte legate de imprimare. In mod particular, unul dintre microprocesoare se ocupa cu achizitia datelor de la diferitele porturi de comunicatie de pe imprimanta, pentru recunoasterea momentului de apasare a unui buton de pe panoul de control, pentru afisarea de mesaje catre utilizator pe monitor, pentru recunoasterea blocarii hartiilor in imprimanta si revenirea din aceasta stare, pentru recunoasterea momentului cand imprimanta nu mai are hartie, si asa mai departe.

Dar cea mai mare responsabilitate a microprocesorului este de a se ocupa cu motorul laserului, acea componenta a imprimantei ce este responsabila cu plasarea semnelor negre pe hartie. Singurul lucru facut de motorul laserului fara asistenta microprocesorului este de a plasa un punct negru, sau sa plaseze cate un punct negru in fiecare loc aferent pe o bucata de hartie. Nu are nici o notiune legata de forma literelor, fonturi, marimea fonturilor, caractere italice, subliniere, caractere ingrosate, sau oricare dintre lucrurile luate de bune de catre utilizatorii imprimantei. Microprocesorul trebuie sa citeasca datele de intrare si sa isi dea seama unde trebuie plasat fiecare punct negru. Acest lucru duce la o alta problema gasita in sistemele integrate.

**Blocarea procesorului**

Realizarea unde trebuie plasat fiecare punct negru atunci cand unei imprimante i se cere sa printeze un anumit text, pe o linie inclinata cu un font neobisnuit de dimensiune foarte mica, ia foarte mult timp, chiar si pentru un microprocesor puternic. Utilizatorii se asteapta la un raspuns rapid atunci cand apasa pe un buton, cu toate acestea, faptul ca microprocesorul este ocupat cu gasirea de valori pentru functii trigonometrice pentru a descoperi unde pe pagina trebuie plasate literele dintr-un font serif inclinat nu este problema lor. Calcule ce tin procesorul blocat pentru perioade lungi de timp duc la cresterea problemei timpului de raspuns.

**Monitor pentru rezervor subteran**

Sistemul de monitorizare pentru un rezervor subteran urmareste nivelurile de combustibil din acesta la o statie de alimentare. Scopul sau principal este de a detecta scurgeri inainte ca statia de alimentare sa se transforme intrun depozit de materiale toxice din greseala sau sa activeze o alarma foarte zgomotoasa la detectare. Sistemul are de asemenea un panou de control cu 16 butoane, un afisaj cu cristale lichide pentru 20 de caractere, si o imprimanta termica. Cu ajutorul butoanelor, utilizatorul poate transmite sistemului sa afiseze sau sa printeze diferite informatii cum ar fi nivelurile de combustibil din rezervoare, sau timpul si data, sau informatii generale de stare ale sistemului.

Pentru a afla cat combustibil este in unul din rezervoare, sistemul citeste nivelurile date de doua flotoare din rezervor, unul ce indica nivelul de combustibil si unul ce indica nivelul de apa ce se acumuleaza intotdeaua la baza acestor rezervoare. De asemenea citeste valoarea temperaturii si diferite alte valori ale rezervorului. Benzina se dilata sau se contracta in mod considerabil o data cu schimbarea temperaturii, iar acest lucru trebuie luat in seama. Sistemul nu trebuie sa declanseze alarma doar pentru ca benzina s-a racit si s-a contractat ducand la coborarea flotorului.

Nici una dintre aceste probleme nu ar fi dificila in mod particular, in afara de problema costului ce deseori creste in cazul sistemelor integrate.

**Cost**

Proprietarul unei statii de alimentare cumpara un asemenea dispozitiv doar pentru ca o agentie guvernamentala il sfatuieste in acest sens. De aceea, el urmareste ca dispozitivul in cauza sa fie cat se poate de ieftin. Asadar, sistemul va fi realizat cu un microcontroller ieftin, probabil unul ce abia se descurca cu adunarea numerelor pe 8 biti, nameivorbind de folosirea coeficientului de expandare a benzinei intr-o maniera eficienta. Astfel, microprocesorul din acest sistem va fi foarte ocupat doar cu calcularea cantitatii de benzina reale din rezervor; Acest lucru ar duce de multe ori la blocarea procesorului.

**Monitor pentru un reactor nucear**

In cele din urma, un exemplu foarte simplu din care putem deduce surprinzator de mult, ar fi un sistem ipotetic ce controleaza un reactor nuclear. Sistemul nostru ipotetic trebuie sa faca multe lucruri, dar singurul aspect ce ne va interesa va fi portiunea de cod ce va monitoriza doua temperaturi, ce ar trebui tot timpul sa fie de valori egale. Daca ele difera, se va indica existenta unei probleme in cadrul reactorului.

**Unelte de dezvoltare a software-ului pentru sisteme integrate**

Programatorii de aplicatii de obicei lucreaza pe acelasi tip de calculator pe care va rula aplicatia in cauza. De exemplu, cineva ce scrie un program ce va rula sub Windows, de obicei va scrie programul folosind ca sistem de operare Windows. El sau ea editeaza programuil, il compileaza, il incearca, il depaneaza, totul pe aceeasi masina.

Aceasta abordare trebuie sa se schimbe pentru sisteme integrate. In primul rand, majoritatea sistemelor integrate au componente hardware specializate ce pot fi atasate la senzori speciali sau ce pot conduce controale speciale, iar singura modalitate de a incerca software-ul este pe hardware-ul specializat. In al doilea rand, sistemele integrate folosesc microprocesoare ce nu au mai fost folosite niciodata ca baza a statiilor de lucru (si e improbabil sa fie folosite). Evident, programele nu sunt compilate in mod magic in setul de instructiuni al unui microprocesor oarecare ales pentru sistemul in cauza, iar programele nu sunt introduse in mod magic in memoria sistemului integrat pentru executie. In continuare vom discuta despre uneltele ce fac aceste lucruri posibile.

**Masinile gazda si tinta**

In lumea sistemelor integrate exista o serie de motive pentru a realiza programarea efectiva pe o masina diferita fata de cea pe care va rula programul eventual. Sistemul ce va fi dat spre utilizare poate avea sau nu o tastatura, un ecran, o unitate pentru discuri sau alte periferice necesare pentru programare. S-ar putea sa nu aiba suficienta memorie pentru a rula un editor de programe, sau e pobibil ca un editor de programe pentru microprocesorul respectiv sa nici nu fi fost scris.

Asadar, majoritatea programarii pentru sisteme integrate este realizata pe o masina gazda, un calculator pe care toate uneltele de programare ruleaza. Numai dupa ce programul a fost scris, compilat, ansamblat si mutat pe masina tinta, sistemul este livrat clientilor. Unii oameni folosesc termenul de statie de lucru in loc de gazda; Cuvantul tinta este aproape universal.

**Compilatoare de trecere**

Majoritatea calculatoarelor folosite ca si gazde vin o data cu compilatoare, asambloare si asa mai departe pentru crearea de programe ce vor rula pe sistemul gazda. Aceste unelte sunt numite unelte navite („native tools”). Compilatorul original pe un sistem Windows NT bazat pe un Intel Pentium, de exemplu, creaza programe destinate pentru a rula pe un Intel Pentium. Acest compilator va poate fi folositor daca microprocesorul tinta este un Pentium, dar este complet nefolositor daca microprocesorul tinta este altceva, sa zicem un Motorola 6800 sau un MMIPS sau un Zilog Z80. Aceste procesoare mai noi nu vor intelege instructiunile binare ale unui Pentium, dar instructiunile Pentium sunt ceea ce produc compilatoarele originale. Ceea ce ne trebuie este un compilator ce ruleaza pe masina gazda si produce instructiunile binare ce vor fi intelese de microprocesorul tinta. Un astfel de program este denumit compilator de trecere.

Intr-o lume ideala, daca am scrie un program in C sau C++ ce l-am putea compila pe un compilator original si rula pe masina gazda, vom putea rula acelasi cod sursa prin intermediul compilatorului de trecere si vom obtine un program ce va putea fi rulat pe masina tinta. Din pacate, acest lucru nu este adevarat, nici macar in teorie mai ales in practica. In teorie, un program ce este compilat fara eroare pe compilatorul original, va trebui sa compileze fara eroare si pe compilatorul de trecere. Regulile ce ce fac un program in C sau C++ sa fie format corect sunt bine definite. Cu toate acestea, in practica trebuie sa ne asteptam ca anumite constructii acceptate de un compilator nu vor fi acceptate de altul. Nu vor exista probleme cu instructiuni *if*sau*switch* sau bucle *do*. Vor aparea probleme cu functii folosite fara sa fie declarate, functii declarate folosind stiluri de declarare vechi, si asa mai departe. Cei ce vand compilatoare incearca permanent sa minimizeze aceste probleme, dar ele inca nu au disparut.

Faptul ca programul ruleaza bine pe masina gazda si este compilat fara eroare pe compilatorul de trecere, nu reprezinta o asigurare ca acesta va functiona pe masina tinta. Aceleasi probleme ce apar o data cu incercarea de portare a programelor C de pe o masina pe alta se aplica si aici. Variabilele declarate ca *int* s-ar putea sa fie de o anumita marime pe masina gazda si de o alta marime pe masina tinta. Structurile pot fi impachetate diferit pe cele doua masini. Abilitatea de a accesa entitati de 16 biti si de 32 de biti ce rezida in adrese numerotate intr-o maniera depasita poate fi diferita. Si asa mai departe.

Din aceasta cauza, trebuie sa ne asteptam la o serie diferita de avertizari de la un compilator de trecere. De exemplu, daca codul respectiv creaza un pointer nul la un int, compilatorul original s-ar putea sa stie ca cele doua entitati sunt de aceeasi dimensiune si sa nu afiseze o avertizare. Compilatorul de trecere, pe de alta parte, s-ar putea sa avertizeze ca intregii si pointerii vizi nu au aceeasi dimensiune pe sistemul tinta.

**Asambloare de trecere si lanturi de unelte**

O alta unelta necesara daca vrem sa scriem un program in limbaj de asamblare este in asamblor de trecere. Asa cum va puteti imagina din denumire, un asamblor de trecere este un asamblor ce ruleaza pe masina gazda, dar produce instructiuni binare potrivite pentru masina tinta. Datele de intrare pentru asamblorul de trecere trebuie sa fie limbaj de asamblare potrivit pentru masina tinta (pentru ca acela este singurul limbaj de asamblare ce poate fi transformat in instructiuni binare pentru tinta). Nu are nici un sens sa astaptam ca datele de intrare potrivite pentru un asamblor de legatura, sa aiba vreo legatura cu datele de intrare pentru asamblorul original.

In figura de mai jos este prezentat procesul de realizare de software pentru sisteme integrate. Asa cum putem observa, fisierele de iesire de la fiecare instrument devin fisiere de intrare pentru urmatorul. Din aceasta cauza, instrumentele trebuie sa fie compatibile unul cu altul. Un set de unelte ce sunt compatibile in acest sens sunt numite lant de unelte. Lanturile de unelte ce pot rula pe diferite masinti gazda si ce construiest programe pentru diferite masini tinta sunt disponibile de la multi vanzatori.



Fisierul executabil este cumva copiat pe masina tinta

Fisiere de asamblare

Limbaj de asamblare al tintei

Fisiere C si C++

Fisier executabil (poate fi in diferite formate standard)

Masina tinta

Asamblor de trecere

Compilator de trecere

Linker

Fisier obiect (poate fi in orice format)

Fisier obiect (poate fi in orice format)

Operatii in masina gazda

**Introducerea softwareului integrat in sistemul tinta**

Asamblorul local va construi un fisier ce va reprezenta imaginea softwareului destinat masinii tinta; sa tratam in continuare problema introducerii acestui fisier in sistemul tinta. Exista mai multe posibilitati pentru acest lucru.

**Programatoarele PROM**

Metoda clasica de a introduce software-ul de la fisierul de iesire al asamblorului local in sistemul tinta este folosirea fisierului pentru a crea un ROM sau un PROM. Crearea unui ROM este potrivita doar atunci cand dezvoltarea software-ului s-a incheiat, tinand cont ca pretul constructiei ROM-ului este destul de ridicat.

Introducerea unui program intr-un PROM necesita un dispozitiv numit programator PROM. Acest lucru este indicat daca marimea codului nu este suficient de mare pentru a justifica creerea unui ROM, sau daca planuim sa facem modificari in cod, sau in timpul depanarii. Daca intentionam sa folosim un PROM si un programator PROM in scopul depanarii, este folositor sa construim mai multe versiuni ale sistemului tinta in care PROM-ul este plasat intr-un soclu pe acesta, in loc sa fie lipit direct. Astfel, la gasirea unui bug, se poate inlatura PROM-ul continand software-ul cu bug-ul referit, si stergerea lui (daca esre un PROM ce poate fi sters), sau aruncarea sa la gunoi (daca nu poate fi sters), scrierea unui PROM nou cu un software ce are bug-ul rezolvat, si punerea acestuia in soclu. Pentru aceasta operatie este necesar un mic dispozitiv denumit „chip puller”. Acesta este ieftin si usor de folosit in cele mai multe cazuri. Introducerea PROM-ului in soclu se face de obicei prin simpla apasare.

Daca planuim sa folosim un programator PROM, trebuie sa ne asiguram ca progrramatorul folosit poate intelege fisierul de iesire creat de asamblorul de pe masina gazda. Este posibila si achizitionarea programatorului de la un vanzator si a asamblorului de la altul, asadar trebuie sa ne asiguram ca cele doua sunt compatibile.

Pin

Chip



Soclu lipit

Plasarea cipului in soclu prin simpla apasare

Contacte ale soclului

Placa tinta

Soclu lipit pe placa tinta

Conexiune interna intre conectorii din soclu si cei din placa tinta

**Emulatoare ROM**

Un alt mecanism popular folosit pentru introducerea software-ului in masini tinta pentru depanare, este emulatorul ROM, un dispozitiv ce inlocuieste memoria ROM din sistemul tinta. Din punctul de vedere al celorlalte componente hardware ale sistemului tinta, emulatorul apare exact ca un ROM. Cu toate acestea, emulatorul ROM contine o cutie mare de componente electrice si un port serial sau un conector de retea prin care se poate conecta la masina gazda. Software-ul ce ruleaza in gazda poate transmite fisiere create de asamblor catre emulatorul ROM, care se va comporta apoi exact ca un ROM ce a fost programat cu software-ul tocmai scris.

Precum cu programatoarele PROM, trebuie sa ne asiguram ca software-ul ce copiaza noul cod in emulatorul ROM, intelege formatul fisierului creat de asamblorul local.

O imagine prezentant un emulator ROM poate fi vazuta in figura de mai jos:



**Emulator ROM**

Conexiune seriala sau de retea ce face legatura dintre emulator si masina gazda

Cablu panglica ce ataseaza sonda de emulatorul ROM

Sonda a emulatorului ROM introdusa in soclul cipului de memorie

Placa tinta

Soclul cipului de memorie

**Emulatoarele de circuit interne (IN-Circuit Emulators)**

Daca folosim un astfel de dispozitiv pentru depanarea software-ului, atunci putem folosi suprapunerea memoriei („overlay”), o functionalitate obisnuita a emulatoarelor de circuit interne, ca un mecanism de introducere a fostware-ului in masina tinta in scopul depanarii.

**Flash**

Daca sistemul integrat isi depoziteaza programul in memoria flash, o optiune ce tot timpul ar putea fi plauzibila a fi plasarea flash-ului intr-un soclu si tratarea lui ca un EPROM . Majoritatea programatoarelor PROM pot programa componente de memorie flash. Totodata, daca masina tinta are un port serial, un conector de retea sau un alt mecanism de comunicatie cu lumea exterioara, memoria flash deschide o noua posibilitate: este posibila scrierea unei parti de cod care sa permita primirea de noi programe de la gazda prin intermediul liniei de comunicatie, si scrierea lor in memoria flash. Desi aceasta portiune de cod poate parea dificil de scris – si poate fi intradevar – scrierea ei poate fi folositoare pentru un numar rezonabil de motive:

* Se poate incarca software nou in sistem pentru depanare – fara tragerea de cipuri din socluri si inlocuirea lor, asumandu-ne totodata riscul de a lipi pini intre ei, a rupe fire, sau altfel stricarea unei componente hardware prototip fragile.
* Copierea de software nou in memoria flash prin intermediul portului serial sau a unui port de retea e mult mai rapida decat scoaterea unei componente din soclu, programarea ei in programatorul PROM, si replasarea ei in soclu.
* Daca se doreste permiterea clientilor sa incarce versiuni noi de software pe sistemul integrat in camp, un motiv des intalnit pentru folosirea memoriei flash pentru sistem, atunci acesta este un software ce ar trebui scris oricum.

Angajarea la acest proiect, aduce cu sine intampinarea urmatoarelor probleme:

* Avand in vedere ca microprocesorul nu poate prelua instructiunile din flash in timp ce programeaza memoria flash, software-ul de programare a flash-ului trebuie sa se copieze singur in RAM. Acest lucru va schimba adresa la care software-ul ruleaza. Din moment ce asamblorul local a construit software-ul ca sa ruleze la locatia sa originala in flash, va trebui sa ne dam seama cum sa-l facem sa ruleze in noua locatie.
* Se va dori ca sistemul tinta sa poata copia software nou, chiar daca se intrerupe in mijlocul unei copieri anterioare. Pentru a asigura acest lucru, trebuie sa se aranjeze o modalitate pentru sistem de a accesa software-ul programat in flash, chiar daca acesta este singurul software ce functioneaza in sistemul tinta. De obicei, acest lucru necesita asigurarea ca software-ul de la pornire nu poate fi usor corupt. Modalitatea obisnuita de a realiza acest lucru, este copierea intregului software de inceput prin legatura de comunicatie in RAM, si apoi copierea acestuia in flash ca intr-o singura operatie, pe cat de scurta posibil.
* Din acelsi motiv, de fiecare data cand se modifica software-ul de programare a memoriei flash, se va dori copierea acestuia in RAM iar apoi copierea acestuia in flash. Pentru a face acest lucru fezabil, va trebui sa se asigure ca software-ul de programare a flash-ului se gaseste intr-un singur bloc de adrese, si ca nu depinde de functii de bibliotecasi altele in afara blocului. Deseori, va fi nevoie de plasarea softului de programare a flash-ului in propriile segmente, pentru a transmite asamblorului local sa-l plaseze in alta locatie fata de restul codului.
* In timpul depanarii softului de programare a memoriei flash, va trebui sa se foloseasca una dintre celelalte metode de copiere a acestuia in sistemul tinta.

**Monitoare**

O alta posibilitate avuta cu sistemele ce dispun de un port de comunicatii este folosirea unui *monitor*, un program ce rezida in ROM-ul sistemului tinta si stie sa incarce noi programe in sistem. Un monitor tipic permite transmiterea software-ului printr-un port serial, inmagazinarea acelui soft in RAM-ul sistemului tinta, si apoi rularea acestuia. Cateodata monitorul efectueaza de asemenea si functiile unui asamblor local, iar anumite monitoare ofera si o serie de servicii pentru depanare, cum ar fi setarea de break-points (puncte de oprire), si afisarea memoriei si a valorii registrelor. Se poate scrie chiar propriul program monitor, dar ele sunt disponibile de la multi vanzatori, printre care si vanzatorii de sisteme de operare in timp real.

Evident, doar daca sistemul integrat nu va fi tot timpul atasat la o gazda ce poate copia software in ea, monitoarele sunt folositoare doar pentru depanare, si va fi necesara folosirea unei alte metode de incarcare a softului in sisteme ce vor fi livrate clientilor.

**Concluzii**

* Dezvoltarea de software pentru sisteme integrate este facuta in mod normal pe o masina gazda, diferita fata de masina tinta pe care softul va fi eventual livrat clientilor.
* Un lant de unelte pentru dezvoltarea de software integrat contine in mod normal un compilator de recere, un asamblor de trecere, si o metoda de incarcare a softului in masina tinta.
* Un compilator de trecere intelege acelasi limbaj C ca si compilatorul original (cu cateva exceptii), dar iesirea sa foloseste setul de instructiuni al microprocesorului tinta.
* Un asamblor de trecere intelege limbajul de asamblare specific microprocesorului tinta si transmite mai departe instructiuni pentru acel microprocesor.
* Linker/Locator combina separat module compilate si asamblate intr-o imagine executabila.In plus, el plaseaza cod, date, cod de pornire, siruri de caractere constante, si asa mai departe la adrese convenabile in ROM si RAM.
* Linker/Locator-ii folosesc segmente pentru a decide unde sa plaseze diferitele portiuni de cod si date.
* Linker/Locator-ii produc iesiri in formate diferite; ramane responsabilitatea programatorului sa asigure ca iesirea din linker/locator este compatibila cu uneltele folosite pentru incarcarea softului in tinta.
* Trebuie sa se gaseasca o posibilitate de a incarca softul in sistemul tinta pentru testare. Cele mai obisnuite modalitati includ programatoare PROM, emulatoare ROM, emulatoare de circuit interne, memorie flash si monitoare.

**Notiunea de firmware**

În electronica şi informatica, firmware-ul este un termen folosit adesea pentru a desemna programele fixe, de obicei destul de mici, şi structurile de date ce controlează intern diverse dispozitive electronice. Printre exemplele tipice de dispozitive care conţin produse destinate utilizatorului final din gama firmware, ar fi calculatoare sau telecomenzi, piese de calculator şi dispozitive precum hard-disk-uri, tastaturi, ecrane TFT sau carduri de memorie, pana la instrumente stiintifice şi roboti industriali. De asemenea, produse de consum mai complexe, cum ar fi telefoanele mobile, camerele digitale, sintetizatoarele, etc, conţin firmware pentru a permite efectuarea functiilor de bază, precum şi pentru implementarea funcţiilor de nivel înalt.

Nu există graniţe stricte între firmware şi software, ambii termeni descriptivi fiind destul de vagi. Cu toate acestea, termenul de firmware a fost iniţial introdus în scopul de a contrasta cu software-ul de nivel superior care poate fi schimbat fara înlocuirea unei componente hardware, acesta fiind de obicei implicat in asigurarea functionarii operatiilor de bază cu nivel scăzut de complexitate. fără de care un dispozitiv ar fi complet nefuncţional. Firmware-ul este, de asemenea, un termen relativ, pentru ca cele mai multe dispozitive integrate conţin firmware la mai mult de un nivel. Subsisteme, cum ar fi CPU, memorii flash, regulatoare de comunicare, module LCD, şi aşa mai departe, au propriul lor (de obicei fix) cod de program şi / sau microcod, considerat ca fiind "parte a hardware-ului", de catre firmware-ul de nivel superior.

Firmware-ul simplu de obicei se află în ROM sau OTP /PROM, în timp ce firmware-ul mai complex (de multe ori la graniţa cu software-ul) foloseste de obicei memorie flash pentru a permite actualizări, cel puţin în dispozitivele moderne. Motive comune pentru actualizarea firmware-ul includ rezolvarea bug-urilor sau adăugarea de caracteristici noi pentru dispozitiv. Făcând acest lucru, de obicei implică încărcarea unui fişier imagine binara (furnizat de către producător) în dispozitiv, în conformitate cu o procedură specifică; acest lucru, este uneori dorit (de către producătorul dispozitivului) a fi efectuat de către utilizatorul final.

**Originea termenului**

Ascher Opler a inventat termenul de "firmware" într-un articol din 1967 din revista Datamation. Iniţial, aceasta a însemnat microcod – continutul unui spatiu de depozitare ce poate fi scris (o memorie mica specializata de mare viteză), care a definit si a implementat setul de instructiuni al calculatorului. Daca este necesar, se poate reincarca firmware-ul pentru a specializa sau a modifica instructiunile ce pot fi executate de unitatea centrala de procesare (CPU). Asa cum a fost folosit initial, firmware-ul contrasta cu hardware-ul (chiar cu CPU-ul ) si cu software-ul (instructiunile normale executate de CPU). Nu era compus din instructiuni masina ale unitatii centrale de procesare, ci din microcod de nivel inferior folosit la implementarea instructiunilor masina. A existat la granita dintre hardware si software, de aici si numele de “firmware”.

**Firmware in anul 2010**

Conceptul de “firmware” a evoluat in a insemna aproape orice continut programabil al unui dispozitiv hardware, nu doar codul maisna pentru un procesor, dar si configuratii si date pentru circuite integrate specializate destinate aplicatiilor (“application-specific integrated circuits - ASICs”), dispozitive logice programabile, etc.

**Calculatoare personale**

Sub anumite aspecte, diferitele componente firmware sunt la fel de importante precum sistemul de operare intr-un calculator functional. Totodata, spre deosebire de majoritatea sistemelor de operare moderne, firmware-ul are rareori un mecanism automat evoluat de a se autoactualiza pentru a rezolva orice probleme legate de functionalitate detectate dupa transportul si punerea efectiva in functiune a unitatii.

In momentul de fata, oricine poate actualiza cu usurinta BIOS-ul intr-un calculator personal modern; dispozitive precum placile video sau modemurile deseori se bazeaza pe firmware dinamic, incarcat de un driver al dispozitivului, si pot fi prin urmare actualizate in mod transparent prin mecanismul de actualizare propriu al sistemului de operare. In contast, firmware-ul din dispozitivele de depozitare este actualizate foarte rar, chiar si cand este folosita memorie flash (mai degraba decat ROM); nu existe mecanisme standardizate pentru detectarea si actualizarea versiunilor de firmware. Cu toate acestea, in practica, asemenea dispozitive au o rata de functionalitate scazuta in comparatie cu dispozitivele in care firmware-ul poate fi actualizat.

**Periferice**

Majoritatea perifericelor calculatoarelor sunt in sine mini calculatoare specializate. In timp ce dispozitivele externe au firmware incarcat intern, o data cu anul 2010, placile video moderne si placile pentru dezvoltare (expansion cards (se pot introduce in placa de baza pentru a adauga functionalitate)), au deseori parti de firmware incarcate de sistemul de operare la pornire, acest lucru oferind un grad mai ridicat de flexibilitate. Acest tip de hardware poate fi astfel nefunctionabil pana in momentul in care calculatorul gazda nu a terminat a-I “alimenta” cu firmware-ul necesar, in mod normal printr-un driver specializat (mai exact: printr-un subsistem de start-up dintr-un pachet de drivere ale dispozitivului). Driverele moderne aferente dispozitivelor interne cat si a celor externe, pot contine de asemenea o interfata grafica cu utilizatorul directa, destinata configurarii, ce adesea foloseste pati dintr-un program normal de interfatare cu utilizatorul, in plus fata de call-urile sistemului de operare, si/sau alte interfete destinare driverelor pentru dispozitive.

**Produse destinate consumatorilor**

O data cu 2010, majoritatea dispozitivelor portabile destinate ascultarii de muzica, au suport pentru actualizare de firmware. Anumite companii folosesc actualizari de firmware pentru a adauga noi formate de fisiere ce pot fi ascultate ; iriver a adaugat formatul Vorbis astfel, de exemplu. Alte functionalitati ce pot fi schimbate o data cu actualizarea firmware-ului pot include interfata cu utilizatorul, sau timpul de viata al bateriei. Majoritatea telefoanelor mobile au posibilitatea de actualizare a firmware-ului, in mare parte pentru aceleasi motive; Unele pot fi actualizate chiar pentru a mari calitatea receptiei sau a sunetului, ilustrand faptul ca firmware-ul este folosit la mai mult de un nivel in produse complexe (in microcontrollere gen CPU fata de dispozitive de analiza a semnalului – digital signal processors (DSP) in caz particular).

**Automobile**

Incepand cu anul 1996, majoritatea automobilelor au inceput sa foloseasca un calculator de bord si diversi senzori pentru detectarea problemelor mecanice. Incepand cu 2010, vehiculele moderne include de asemenea sisteme ABS controlate de calculator, precum si sisteme de control a transmisiei (Transmission Control System - TCS). Driverele folosite pot obtine in timp real, in timpul mersului informatii cum ar fi gradul de economisire a combustibilului, sau presiunea rotilor. Distribuitorii locali pot actualiza firmware-ul pentru majoritatea tipurilor de maisni.

**Definitia IEEE**

Institute of Electrical and Electronics Engineers (IEEE) Standard Glossary of Software Engineer Terminology, Std 610. 12-1990 defineste firmware-ul asa cum urmeaza:

*“Combinatia dintre un dispozitiv hardware si instructiuni de calculator si date ce rezida ca software ce poate fi doar citit (read-only) pe acel dispozitiv”.*

*Note: (1) Acest termen este uneori folosit pentru a face referire doar la un dispozitiv hardware, sau doar la instructiunile de calculator sau date, dar aceste definiri sunt depreciate. (2) Confuzia ce inconjoara acest termen a facut unele persoane sa sugereze ca el poate fi evitat in totalitate.*

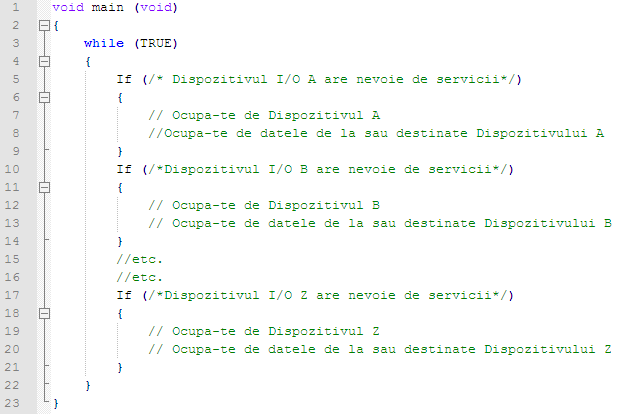
**Arhitecturi software pentru sisteme embedded**

Cel mai important factor ce determina ce arhitectura va fi cea mai optima pentru orice tip de sistem embedded, este gradul de control avut asupra raspunsului sistemului. Dificultatea de a obtine un raspuns bun, depinde nunumai de timpii de raspuns absoluti necesari, dar si de viteza de procesare a microprocesorului, precum si celelalte cerinte de procesare. Un sistem ce nu exceleaza in functionalitate si cu putine cerinte legate de timp de raspuns, poate fi scris cu o arhitectura foarte simpla. Pe de alta parte, un sistem ce trebuie sa raspunda rapid la multe evenimente diferite, si ce are cerinte de procesare variate, toate cu termeni limita si prioritati diferite, va avea nevoie de o arhitectura mult mai complexa.

Vom prezenta 4 tipuri de arhitecturi, incapand cu cea mai simpla dintre ele, ce nu ofera practic nici un control asupra raspunsului si prioritatilor, pana la cele ce ofera mult maimult control, dar cu costul unei complexitati mult mai mari. Cele 4 sunt round-robin, round-robin cu intreruperi, coada de functii programabila (function-queue-scheduling), si sisteme de operare in timp real.

**Round-Robin**

Codul din Fig. 1 este prototipul algoritmului Round-Robin, cea mai simpla arhitectura imaginabila. Nu exista intreruperi. Bucla principala pur si implu verifica fiecare dispozitiv de intrare-iesire la rand, si se ocupa de cele ce au nevoie de servicii.

**Fig. 1**

Aceasta este o arhitectura eminamente simplista – fara intreruperi, fara informatii partajate, fara probleme legate de intarzieri, fiind astfel o arhitectura cu un potential intotdeauna atractiv, atata timp cat o putem folosi.

Asa simplista cum este, arhitectura round-robin poate fi adecvata in anumite instante. Luati in considerare spre exemplu un multimetru digital. Acesta este un dispozitiv ce masoara rezistenta electrica, intensitatea si diferenta de potential in ohmi, amperi si volti, fiecare dintre ele in intervale de masura diferite. Un multimetru obisnuit are doua sonde pe care utilizatorul le atinge de doua puncte ale circuitului ce va fi masurat, un display digital, si un buton ce prin rotire alege ce masuratoare sa se faca si in ce interval de masura. Sistemul face masuratori continue, si modifica afisajul in functie de cea mai recenta masuratoare. O imagine a unui astfel de dispozitiv putem observa in figura de mai jos.



Volti

Sonde

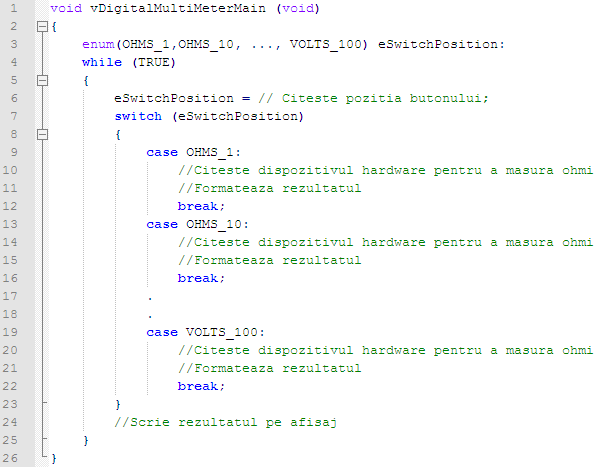
Volti

Amperi

Un posibil pseudocod pentru un multimetru este afisat in Fig. 2. La fiecare trecere prin bucla, se verifica pozitia butonului principal, si se face legatura catre portiunea de cod pentru a face masuratoarea potrivita, pentru a formata rezultatul, si a scrie rezultatul in afisaj. Chiar si un microprocesor modest poate trece prin aceasta bucla de multe ori in fiecare secunda.

Arhitectura round-robin functioneaza foarte bine pentru acest sistem pentru ca sunt doar 3 dispozitive I/O , timpul de procesare nu este unul mare, iar cerintele legate de timpul de raspuns de asemenea nu sunt atat de stricte. Microprocesorul poate citi dispozitivul hardware ce efectiv face masuratoarea in orice moment. Dispozitivul de afisaj poate fi scris la orice viteza convenabila pentru microprocesor. Cand utilizatorul schimba pozitia butonului principal, este putin probabil sa observe putinele fractiuni de secunda necesare procesorului pentru a parcurge bucla principala. (Cel mai probabil acesta este prea distras de repozitionarea sondelor pentru a observa o intarziere putin mai mare.) Asadar, arhitectura round-robin este adecvata pentru a indeplini toate aceste cerinte, iar simplitatea ei o face extrem de atractiva ca alegere pentru acest sistem.

Din pacate, arhitectura round-robin are un singur avantaj asupra celorlalte arhitecturi – simplitatea - in timp ce are si o serie de probleme ce o fac nepotrivita pentru multe ate sisteme.

**Fig. 2**

Daca unul din dispozitive trebuie sa raspunda in mai putin timp decat ii ia microprocesorului sa parcurga bucla principala in cazul cel mai defavorabil, sistemul nu va mai functiona. In Fig. 1, daca dispozitivul Z nu poate astepta mai mult de 7 milisecunde pentru a dispune de un serviciu, si daca partile de cod ce ofera serviciu dispozitivelor A si B dureaza 5 milisecunde fiecare, atunci procesorul nu va putea ajunge la dispozitivul Z suficient de rapid de fiecare data. Totusi, se poate obtine putin mai mult de la arhitectura round-robin prin testarea dispozitivului A, apoi B, apoi Z si asa mai departe, dar aceasta tehnica este totusi limitata. Lumea este plina de dispozitive I/O ce au nevoie de servicii rapide: porturi seriale, porturi de retea, butoane activate prin apasare, etc.

**Round-Robin cu intreruperi**

Acest tip de arhitectura ofera mai mult control asupra prioritatilor intreruperilor hardware. Portiunile de cod destinate intreruperilor implementeaza functii de prioritate mai ridicata (permitand asignarea nivelurilor de prioritate catre dispozitive). Aceste portiuni de cod de asemenea seteaza flaguri, ce sunt asociate codului aferent fiecarui task, permitand continuarea sa dupa terminarea portiunii de cod initiale (e.g, fDeviceX, unde X este dispozitivul A, B sau Z – Fig.3).

Printre avantajele acestui tip de arhitecturi putem mentiona evitarea pericolul de aparitie a informatiei partajate, printr-o serie de tehnici specifice ce vor fi explicate. Ca dezavantaje ar fi setarea si controlul folosind prioritati.

Algoritmul arhitecturii round-robin cu intreruperi este prezentat in Fig. 4.

**Nivelurile de prioritate ale arhitecturii Round-Robin cu intreruperi**

Round Robin cu intreruperi

Procesare de

prioritate ridicata

Round Robin

|  |
| --- |
| Dispozitiv A ISR |
| Dispozitiv B ISR |
| Dispozitiv C ISR |
| Dispozitiv D ISR |
| Dispozitiv … ISR  Tot |
| Dispozitiv Z ISR |
| Codul tuturor taskurilor  Procesare de  prioritate scazuta |

**Fig. 3**

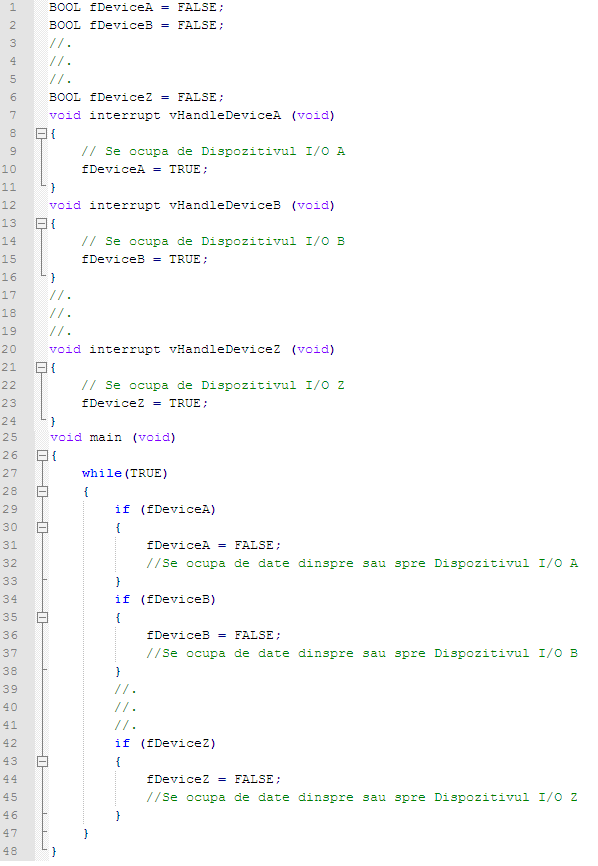
Aceasta arhitectura este potrivita pentru multe tipuri de sisteme, incepand de la cele relativ simple pana la cele surprinzator de complexe. Un exemplu din categoria celor simple este un bridge de comunicare, un dispozitiv cu doua porturi, ce transmite mai departe informatia primita la primul port, catre cel de-al doilea port si vice-versa. Un asemenea dispozitiv este afisat in figura de mai jos.



**Legatura de comunicatie** B

**Legatura de comunicatie A**

**Fig. 4**



**Fig. 4**

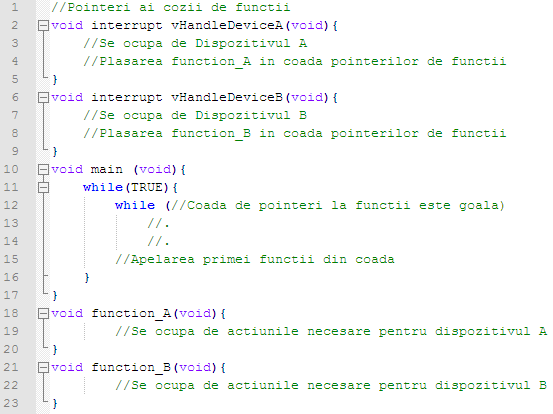
Avantajul evident al arhitecturii round-robin cu intreruperi este ca timpul de raspuns al sarcinilor cu o prioritate mare este imbunatatit, atata timp cat ISR-ul are prioritate fata de bucla principala (bucla principala va fi intotdeauna intrerupta, indiferent de ceea ce face pentru a satisface intreruperea), si cu toate acestea algoritmul ramane relativ simplu. Timpul de raspuns cel mai lung pentru o sarcina cu prioritate scazuta, corespunzator cazului cel mai defavorabil, este suma timpilor de executie a codului din bucla principala si a tuturor rutinelor de tratare a intreruperilor. O data cu introducerea intreruperilor, poate aparea problema informatiei partajate: daca functia intrerupta de prioritate scazuta este in mijlocul calculului folosind date ce sunt asigurate sau modificate de o functie de inrerupere de prioritate ridicata, trebuie sa se aiba grija ca la revenirea de la intrerupere, informatiile functiei de prioooritate scazuta sunt inca valide (prin annnularea intreruperiiilor din jurulll sectiunilor de cod critic, de exemplu).

**Coada de functii programabila (Function Queue Scheduling)**

Aceasta arhitectura ofera o metoda de asignare de prioritati intreruperilor. Aici, rutinele de tratare a intreruperilor (ISR) realizeaza procesari urgente de la dispozitivele de intrerupere, punand de asemenea un pointer catre o functie intr-o coada de asteptare pentru calculele realizate in continuare. Bucla principala verifica doar coada de asteptare a functiilor, iar daca aceasta nu este goala, face apel care prima functie din coada. Prioritatile sunt asignate in functie de ordinea functiilor din coada – nu exista nici un motiv pentru care functiile sa fie plasate in coada in ordinea de aparitie a intreruperilor. Acestea pot fi plasate cu usurinta in ordinea prioritatilor : functiile cu prioritate ridicata la inceputul cozii si functiile cu prioritate scazuta la sfarsitul acesteia. Timpul de asteptare corespunzator cazului cel mai defavorabil pentru functia cu cea mai ridicata prioritate, este timpul de executie al celei mai lungi functii din coada (ganditi-va la cazul in care procesorul abia incepe executia celei mai lungi functii in momentul in care intreruperea plaseaza o sarcina de prioritate ridicata pe prima pozitie a cozii de asteptare). Timpul de asteptare cel mai lung pentru sarcina cu cea mai scazuta prioritate este infinit : aceasta poate sa nu fie executata niciodata daca portiuni de cod cu o prioritate superioara sunt tot timpul introduse la inceputul cozii. Avantajul cozii de functii programabile este ca prioritati pot fi asociate sarcinilor; dejavantajele sunt ca aceasta arhitectura este mai complicata fata de cele de mai inainte, si de asemenea, si in cazul ei pot aparea probleme de partajare a informatiilor.

Cu toate ca arhitectura Function Queue Scheduling reduce timpul de raspuns aferent cazului cel mai defavorabil pentru un task cu prioritate ridicata, acesta poate fi cateodata insufucient de bun, datorita faptului ca daca un task de prioritate scazuta dureaza prea mult, acest lucru va afecta timpul de raspuns pentru taskul de prioritate ridicata. In acnumite cazuri, aceasta problema poate fi evitata, prin rescrierea functiilor lungi in mai multe parti, fiecare dintre aceste parti programand-o pe cea urmatoare prin adaugarea ei in coada de functii, dar acest lucru face ca procedura sa fie prea complicata. Acestea sunt cazurile unde se cere folosirea arhitecturii sistemelor de operare in timp real.

Un exemplu de pseudocod al arhitecturii cu coada de functii programabile este prezentat in figura urmatoare:

****

**Sisteme de operare in timp real (RTOS)**

Site-ul universitatii MIT (Michigan Institute of Technology ) contine urmatorul sfat:

“Daca aptopul tau cu Windows se reseteaza singur si afiseaza un ecran albastru cu un mesaj de eroare , asa numitul “Blue Screen of Death”, iar apoi se reseteaza iar incercand sa se conecteze la o retea wireless, mai mult ca sigu ai o problema…”

Un BSOD (Blue Screen of Death), precum intalnim la Windows, nu este ceva ce am dori sa regasim intr-un sistem de control pentru un dispozitiv embedded (ganditi-va la sistemul de franare al masinii , sau la sistemul strategic de aparare al zborurilor avioanelor de lupta). Sistemele integrate, embedded, pot fi atat de simple incat nici nu este nevoie de un sistem de operare. Cu toate acestea, atunci cand un OS (Operating System) este folosit, trebuie sa se ofere garantia anumitor capacitati de calcul oferite in anumite intervale prestabilite de timp. Asemenea sisteme de operare mai sunt cunoscute si ca sisteme de operare in timp real sau RTOS.

Un sistem de operare in timp real este complicat, potential scump, si consuma memorie pretioasa in sistemul nostru integrat constrans mai totdeauna in privinta costului si memoriei. Asadar, de ce am folosi unul? Sunt doua motive principale: flexibilitate si timp de raspuns. Componenta principala a sistemului de operare in timp real este taskul, iar adaugarea de noi taskuri si stergerea celor depasite este ceva direct, pentru ca nu exista bucla principala. Sistemul de operare in timp real este programat ca fiecare task sa fie rulat pe baza prioritatii sale. Programarea acestor taskuri de catre RTOS, este referita ca multi-tasking. Intr-un sistem multi-tasking preemptiv, sistemul de operare in timp real poate opri un task de prioritate scazuta in orice moment, pentru a executa unul de prioritate mai ridicata, asadar, timpul de asteptare pentru un task de prioritate ridicata este aproape zero (intr-un sistem multi-tasking ne-preemptiv, taskul de prioritate scazuta isi termina executia inainte ca taskul de prioritate zero sa porneasca). In cel mai simplu sistem de operare in timp real, un task se poate afla in una din urmatoarele 3 stari:

**In rulare:** Codul taskului este executat de procesor. Un singur task poate rula la un anumit moment de timp.

**Gata de rulare:** Toate informatiile necesare sunt disponibile iar taskul este pregatit pentru rulare atunci cand procesorul este disponibil. Multe taskuri pot fi gata de rulare la un moment de timp, si vor rula in functie de prioritate.

**Blocate:** Un task poate fi blocat in asteptarea unor date sau a executiei unui eveniment. Un task, daca nu este preemptat, va fi blocat dupa rularea pana la terminare. Mai multe taskuri pot fi blocate la un momnet de timp.

Partea sistemului de operare in timp real, numita programator tine seama de starea fiecarui task, si decide care dintre acestea sa fie lansat in rulare. Programatorul, este un dispozitiv destul de simplist: Pur si ismplu urmareste toate taskurile aflate in starea gata de rulare, si le alege pe cele cu prioritatea cea mai ridicata. Taskurile se pot bloca singure, daca isi termina executia, si se pot debloca si reveni in starea gata de rulare la aparitia unui eveniment, dar este responsabilitatea programatorului sa treaca taskuri din starea de gata de rulare in starea de rulare, bazate pe prioritatile afisate in Fig. 5.

**Deblocare task**

**Gata de rulare**

**Blocat**

**Task cu prioritate**

**Task ce isi termina executia**

**Prioritate mai mare**

**Task preemptat**

**In rulare**

**Alegerea unei arhitecturi**

In continuare vor fi prezentate o serie de sugestii privind alegerea unei arhitecturi pentru un sistem integrat.:

* Alegeti arhitectura cea mai simpla ce poate satisface raspunsurile si cerintele necesare sistemului. Scrierea software-ului pentru sisteme embedded este ceva suficient de complicat chiar si fara a alege o arhitectura complexa fara a fi necesar.
* Daca sistemul are cerinte de raspuns ce ar implica folosirea unui sistem de operare in timp real, cu siguranta ar trebui sa inclinati spre folosirea acelui sistem. Majoritatea sistemelor comerciale sunt vandute cu o serie de functionalitati folositoare ce ar usura foarte mult testarea si rezolvarea problemelor sistemului in cauza.
* Daca poate fi folositor pentru sistemul in cauza, se pot crea si arhitecturi hibride, prin combinarea celor 4 tipuri prezentate mai inainte. De exemplu, daca este folosit un sistem de operare in timp real, se poate folosi un task de prioritate scazuta pentru a se ocupa de componentele hardware ce nu au nevoie de un raspuns rapid. In mod similar, intr-o arhitectura Round\_robin cu intreruperi, bucla principala se poate ocupa de componentele hardware mai incete in mod direct, in loc sa citeasca flagurile setate de rutinele de intrerupere.

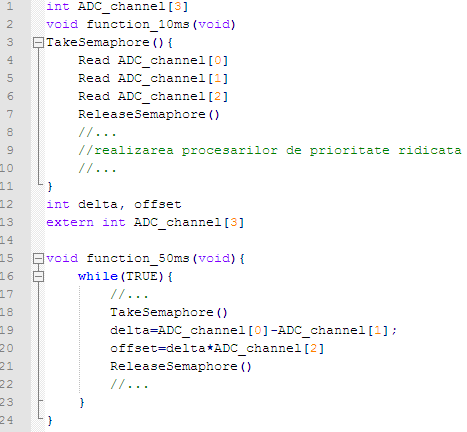
**Problema partajarii datelor**

Am intampinat problema partajarii datelor cand am introdus conceptul de informatii partajate in sisteme in care date generate sau modificate de o intrerupere cu prioritate ridicate, erau folosite intr-un task de prioritate scazuta ce rula la o viteza mai mica. A fost aratat ca sectiunile de cod critice din cadrul taskului de prioritate scazuta trebuie tratate cu mare grija, pentru ca informatiile sa nu fie modificate fara intentie de rutina intreruperii.

Acest lucru a fost realizat prin oprirea intreruperilor la inceputul sectiunilor critice de cod, si repornirea lor la terminarea acestora. Un sistem de operare in timp real trebuie sa aiba tot timpul un mecanism, similar cu pornirea si oprirea intreruperilor, pentru a se ocupa de date partajate intre taskuri cu prioritati diferite. In mod normal, acest mecanism este un *semafor*. Ne putem gandi la un semafor ca la un obiect ce este transmis intre taskuri ce partajeaza date. Un task ce proceseaza un semafor inchide programatorul, nepermitand preemptarea acestuia pana ce semaforul nu este eliberat.

De exemplu, ne putem imagina ca avem de-a face cu citirea unor informatii de la un convertor analog-digital si procesarea acestora la o prioritate ridicata, intr-un timp de 10ms/task, iar la fiecare 50ms, o portiune aleatoare din aceste informatii este folosita intr-un task de prioritate scazuta.

Un astfel de exemplu ar avea urmatorul pseudocod:



**Baze de date**

**Modelul relational**

Modelul relational a fost propus de IBM, si a revolutionat reprezentarea datelor facand trecerea la generatia a doua a bazelor de date.

Modelul este simplu, are o solida fundamentare teoretica fiind bazat pe teoria seturilor (ansamblurilor) si pe logica matematica.Pot fi reprezentate toate tipurile de structuri de date de mare complexitate, din diferite domenii de activitate.

Modelul relaţional este definit prin: structura de date, operatorii care acţionează asupra structurii şi restricţiile de integritate.

1. Conceptele utilizate pentru definirea structurii de date sunt: domeniul, tabela (relaţia), atributul, tuplul, cheia şi schema tabelei.

*Domeniu* este un ansamblu de valori caracterizat printr-un nume. El poate fi explicit sau implicit.

*Tabela/relaţia* este un subansamblu al produsului cartezian al mai multor domenii, caracterizat printr-un nume, prin care se definesc atributele ce aparţin aceleaşi clase de entităţi.

*Atributul* este coloana unei tabele, caracterizată printr-un nume.

*Cheia* este un atribut sau un ansamblu de atribute care au rolul de a identifica un tuplu dintr-o tabelă. Tipuri de chei: primare/alternate, simple/comune, externe.

*Tuplul* este linia dintr-o tabelă şi nu are nume. Ordinea liniilor (tupluri) şi coloanelor (atribute) dintr-o tabelă nu trebuie să prezinte nici-o importanţă.

*Schema tabelei* este formată din numele tabelei, urmat între paranteze rotunde de lista atributelor, iar pentru fiecare atribut se precizează domeniul asociat.

*Schema bazei de date* poate fi reprezentată printr-o diagramă de structură în care sunt puse în evidenţă şi legăturile dintre tabele. Definirea legăturilor dintre tabele se face *logic* construind asocieri între tabele cu ajutorul unor atribute de legătură. Atributele implicate în realizarea legăturilor se găsesc fie în tabelele asociate, fie în tabele distincte construite special pentru legături. Atributul din tabela iniţială se numeşte *cheie externă* iar cel din tabela finală este *cheie primară*. Legăturile posibile sunt *1:1*, *1:m*, *m:n*. Potenţial, orice tabelă se poate lega cu orice tabelă, după orice atribute. Legăturile se stabilesc la momentul descrierii datelor prin limbaje de descriere a datelor (LDD), cu ajutorul restricţiilor de integritate. Practic, se stabilesc şi legături dinamice la momentul execuţiei.

1. Operatorii modelului relaţionalsunt operatorii din algebra relaţională şi operatorii din calculul relaţional.

**Algebra relaţională** este o colecţie de operaţii formale aplicate asupra tabelelor (relaţiilor), şi a fost concepută de E.F.Codd. Operaţiile sunt aplicate în *expresiile algebrice relaţionale* care sunt cereri de regăsire. Acestea sunt compuse din operatorii relaţionali şi operanzi. *Operanzii* sunt întotdeauna tabele (una sau mai multe). *Rezultatul* evaluării unei expresii relaţionale este format dintr-o singură tabelă.

Algebra relaţională are cel puţin puterea de regăsire a calcului relaţional. O expresie din calculul relaţional se poate transforma într-una *echivalentă* din algebra relaţională şi invers.

Codd a introdus şase operatori *de bază* (reuniunea, diferenţa, produsul cartezian, selecţia, proiecţia, joncţiunea) şi doi operatori *derivaţi* (intersecţia şi diviziunea). Ulterior au fost introduşi şi alţi operatori derivaţi (*speciali*). În acest context, o*peratorii* din algebra relaţională pot fi grupaţi în două categorii: pe mulţimi şi speciali.

*Operatori pe mulţimi* (R1, R2, R3 sunt relaţii (tabele)) *sunt:*

* *Reuniunea*. R3 = R1 ∪ R2, unde R3 va conţine tupluri din R1 sau R2 luate o singură dată;
* *Diferenţa.* R3 = R1 \ R2, unde R3 va conţine tupluri din R1 care nu se regăsesc în R2;
* *Produsul cartezian.* R3 = R1 × R2, unde R3 va conţine tupluri construite din perechi (x1x2), cu x1∈R1 şi x2∈R2;
* *Intersecţia.* R3 = R1 ∩ R2, unde R3 va conţine tupluri care se găsesc în R1 şi R2 în acelaşi timp, etc.

*Operatori relaţionali speciali sunt:*

* *Selecţia*. Din R1 se obţine o subtabelă R2, care va conţine o submulţime din tuplurile iniţiale din R1 ce satisfac un predicat (o condiţie). Numărul de atribute din R2 este egal cu numărul de atribute din R1. Numărul de tupluri din R2 este mai mic decât numărul de tupluri din R1.
* *Proiecţia.* Din R1 se obţine o subtabelă R2, care va conţine o submulţime din atributele iniţiale din R1 şi fără tupluri duplicate. Numărul de atribute din R2 este mai mic decât numărul de atribute din R1.
* *Joncţiunea* este o derivaţie a produsului cartezian, ce presupune utilizarea unui calificator care să permită compararea valorilor unor atribute din R1 şi R2, iar rezultatul în R3. R1 şi R2 trebuie să aibă unul sau mai multe atribute comune care au valori comune.

Algebra relaţională este prin definiţie *neprocedurală* (descriptivă), iar calculul relaţional permite o manieră de căutare *mixtă* (procedurală/neprocedurală).

**Calculul relaţional** se bazează pe *calculul predicatelor* de ordinul întâi (domeniu al logicii) şi a fost propus de E.F. Codd. *Predicatul* este o relaţie care se stabileşte între anumite elemente şi care poate fi confirmată sau nu. *Predicatul de ordinul 1* este o relaţie care are drept argumente variabile care nu sunt predicate. *Variabila* poate fi de tip tuplu (valorile sunt dintr-un tuplu al unei tabele) sau domeniu (valorile sunt dintr-un domeniu al unei tabele). *Cuantificatorii* (operatorii) utilizaţi în calculul relaţional sunt: universal (∀) şi existenţial (∃).

*Construcţia de bază* în calculul relaţional este expresia relaţională de calcul tuplu sau domeniu (funcţie de tipul variabilei utilizate).

Expresia relaţională de calcul este formată din: operaţia de efectuat, variabile (tuplu respectiv domeniu), condiţii (de comparaţie, de existenţă), formule bine definite (operanzi-constante, variabile, funcţii, predicate; operatori), cuantificatori.

Pentru implementarea acestor operatori există *comenzi specifice* în limbajele de manipulare a datelor (LMD) din sistemele de gestiune a bazelor de date relaţionale (SGBDR). Aceste comenzi sunt utilizate în operaţii de regăsire (interogare).

După **tehnica folosită** la manipulare, LMD sunt bazate pe:

* calculul relaţional (QUEL în Ingres, ALPHA propus de Codd);
* algebra relaţională (ISBL, RDMS);
* transformare (SQL, SQUARE);
* grafică (QBE, QBF).

*Transformarea* oferă o putere de regăsire *echivalentă* cu cea din calculul şi algebra relaţională. Se bazează pe *transformarea* (mapping) unui atribut sau grup de atribute într-un atribut dorit prin intermediul unor relaţii. *Rezultatul* este o relaţie (tabelă) care se poate utiliza într-o altă transformare.

*Grafica* oferă *interactivitate* mare pentru constrirea cererilor de regăsire. Utilizatorul specifică cerea *alegând* sau completând un ecran structurat grafic. Poate fi *folosit* de către toate categoriile de utilizatori în informatică.

1. Restricţiile de integritate ale modelului relaţional sunt structurale şi comportamentale.

Restricţiile structurale sunt:

* Restricţia de unicitate a cheii. Într-o tabelă nu trebuie să existe mai multe tupluri cu aceeaşi valoare pentru ansamblul cheie;
* Restricţia referenţială. Intr-o tabelă t1 care referă o tabelă t2, valorile cheii externe trebuie să figureze printre valorile cheii primare din t2 sau să ia valoarea null (neprecizat);
* Restricţia entităţii. Intr-o tabelă, atributele din cheia primară nu trebuie să ia valoarea NULL.

Cele trei restricţii de mai sus sunt minimale.

Pe lângă acestea, există o serie de alte restricţii structurale care se referă la dependenţele dintre date: funcţionale, multivaloare, joncţiune etc. (sunt luate în considerare la tehnicile de proiectare a bazelor de date relaţionale - BDR).

Restricţiile de comportament sunt cele care se definesc prin comportamentul datelor şi ţin cont de valorile din BDR:

* Restricţia de domeniu. Domeniul corespunzător unui atribut dintr-o tabelă trebuie să se încadreze între anumite valori;
* Restricţii temporare. Valorile anumitor atribute se compară cu nişte valori temporare (rezultate din calcule etc.).

Restricţiile de comportament fiind foarte generale se gestionează fie la momentul descrierii datelor (de exemplu prin clauza CHECK), fie în afara modelului la momentul execuţiei.

Restricţiile de integritate suportate de Oracle sunt:

* NOT NULL nu permite valori NULL în coloanele unei tabele;
* UNIQUE nu sunt permise valori duplicat în coloanele unei tabele;
* PRIMARY KEY nu permite valori duplicate sau NULL în coloana sau coloanele definite astfel;
* FOREIGN KEY presupune ca fiecare valoare din coloana sau setul de coloane defini astfel să aibă o valoare corespondentă identică în tabela de legătură, tabelă în care coloana corespondentă este definită cu restricţia UNIQUE sau PRIMARY KEY;
* CHECK elimină valorile care nu satisfac anumite cerinţe (condiţii) logice.

Termenul de chei (keys) este folosit pentru definirea câtorva categorii de constrângeri şi sunt: primary key, unique key, foreign key, referenced key.

Se consideră că modelul relaţional are o serie de limite cum ar fi:

* Simplitatea modelului îl face dificil de aplicat pentru noile tipuri de aplicaţii (multimedia, internet etc.);
* Nu asigură o independenţă logică totală a datelor de aplicaţie;
* Poate creşte redundanţa datelor.

**Baze de date relationale**

Bazele de date relaţionale (BDR) utilizează modelul de date relaţional şi noţiunile aferente. BDR au o solidă fundamentare teoretică, în special prin cercetările de la IBM conduse de E.F.Codd.

**BDR** este un ansamblu organizat de *tabele* (relaţii) împreună cu *legăturile* dintre ele.

**Concepte** utilizate la organizarea datelor în BDR şi respectiv fişiere sunt prezentate în tabelul de mai jos:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Fişiere | fişier | înregistrare | câmp | valori |
| BDR | tabelă(relaţie) | tuplu (linie) | atribut(coloană) | domeniu valori |

**Avantajele** BDR faţă de fişiere :

|  |  |  |
| --- | --- | --- |
| CRITERIU | BDR | FIŞIERE |
| Independenţa datelor | logică şi fizică | fizică |
| Niveluri de structurare | conceptual, logic şi fizic | logic şi fizic |
| Deschidere şi portabilitate | mare | mică |
| Reprezentarea şi utilizarea datelor | simplificat prin model | complicat |
| Structura de date se păstrează | în dicţionarul BDR | în programe. |

Atunci când dorim să realizăm o bază de date relaţională trebuie să ştim clar ce avem de făcut, adică să stabilim obiectivele activităţii noastre. În acest sens, câteva dintre cele mai importante **obiective**, le prezentăm în continuare:

* *Partiţionarea* semnifică faptul că aceleaşi date trebuie să poată fi folosite în moduri diferite de către diferiţi utilizatori;
* *Deschiderea* se referă la faptul că datele trebuie să fie uşor adaptabile la schimbările care pot apărea (actualizarea structurii, tipuri noi de date etc.);
* *Eficienţa* are în vedere stocarea şi prelucrarea datelor, care trebuie să se facă la costuri cât mai scăzute, costuri care să fie inferioare beneficiilor obţinute;
* *Reutilizarea* înseamnă faptul că fondul de date existent trebuie să poată fi reutilizat în diferite aplicaţii informatice;
* *Regăsirea* este o actvitate frecventă pe bazele de date şi de aceea cererile de regăsire trebuie să poată fi adresate uşor de către toate categoriile de utilizatori, după diferite criterii;
* *Accesul* înseamnă modul de localizare a datelor şi acest lucru trebuie să poată fi realizat prin diferite moduri de acces, rapid şi uşor;
* *Modularizarea* presupune faptul că realizarea BDR trebuie să se poată face modular pentru generalitate şi posibilitatea lucrului în echipă;
* *Protecţia* bazei de date trebuie asigurată sub ambele aspecte: securitatea şi integritatea datelor;
* *Redundanţa* se asigură în limite acceptabile prin implementarea unui model de date pentru baze de date şi prin utilizarea unei tehnici de proiectare a BDR. Se asigură astfel, o redundanţă minimă şi controlată; *Independenţa* datelor faţă de programe trebuie asigurată atât la nivel logic cât şi şi fizic.

Bazele de date relaţionale au evoluat ca un tip special de aplicaţii informatice, şi anume cele care au organizarea datelor în memoria externă conform unui model de date specific. De aceea, în metodologia de realizare a BDR se parcurg, în cea mai mare parte, cam aceleaşi etape ca la realizarea unei aplicaţii informatice, cu o serie de aspecte specifice. Pe de altă parte, în literatura de specialitate, sunt diferite propuneri de metodologii de realizare a bazelor de date.

Ţinând cont de cele două aspecte de mai sus, sunt propuse câteva actvivităţi care trebuie parcurse la realizarea unei baze de date. Aceste activităţi vor fi regăsite, sub aceeaşi denumire sau sub denumiri diferite, în majoritatea metodologiilor de realizare a bazelor de date, din literatura de specialitate.

**Activităţile** (etapele) parcurse pentru realizarea unei BDR sunt: analiza de sistem, proiectarea noului sistem, realizarea componentelor logice, punerea în funcţiune, dezvoltarea.

1. *Scopul analizei de sistem* este de a evidenţia cerinţele aplicaţiei şi resursele utilizate (studiul), precum şi de a evalua aceste cerinţe prin modelare (analiza).

Studiul situaţiei existente se realizează prin: definirea caracteristicilor generale ale unităţii, identificarea activităţilor desfăşurate, identificarea resurselor existente (informaţionale, umane, energetice, echipamente, financiare etc.), identificarea necesităţilor de prelucrare.

Analiza este o activitate de modelare (conceptuală) şi se realizează sub trei aspecte: structural, dinamic şi funcţional.

1. *Analiza structurală* evidenţiază, la nivel conceptual, modul de structurare a datelor şi a legăturilor dintre ele. Cea mai utilizată tehnică este *entitate-asociere*. Aceasta conţine:

* Identificarea entităţilor: fenomene, procese, obiecte concrete sau abstracte (substantivele din prezentarea activităţii descrise) (exemple de entităţi: Persoane, Produse, Beneficiari).
* Identificarea asocierilor dintre entităţi ca fiind legăturile semnificative de un anumit tip (verbele din prezentarea activităţii descrise).
* Identificarea atributelor ce caracterizează fiecare entitate în parte (exemple de atribute: Marca, Nume, Adresă).
* Stabilirea atributelor de identificare unică a realizărilor entităţii, drept chei.

*Rezultatul* analizei structurale este modelul static (structural) numit şi diagrama entitate-asociere. Diagrama entitate-asociere (Entity-Relationship) poate fi generată cu produse software tip CASE (Computer Aided Software Engineering), ca de exemplu Oracle Designer. Pornind de la o astfel de diagramă, se pot construi, în actvitatea de proiectare, schemele relaţiilor (tabelelor).

1. *Analiza dinamică* evidenţiază comportamentul elementelor sistemului la anumite evenimente. Una din tehnicile utilizate este diagrama *stare-tranziţie*. Aceasta presupune:

* Identificarea stărilor în care se pot afla componentele sistemului.
* Identificarea evenimentelor care determină trecerea unei componente dintr-o stare în alta.
* Stabilirea tranziţiilor admise între stări.
* Construirea diagramei stare-tranziţie.

*Rezultatul* analizei dinamice este modelul dinamic.

1. *Analiza funcţională* evidenţiază modul de asigurare a cerinţelor informaţionale (fluxul prelucrărilor) din cadrul sistemului, prin care intrările sunt transformate în ieşiri. Cea mai utilizată tehnică este *diagrama de flux* al datelor. Conform acestei tehnici se delimitează:

* Aria de cuprindere a sistemului.
* Se identifică sursele de date.
* Se identifică modul de circulaţie şi prelucrare a datelor.
* Se identifică apoi rezultatele obţinute.

*Rezultatul* analizei funcţionale este modelul funcţional.

1. *Proiectarea structurii bazei de date* se face pe baza modelelor realizate în activitatea de analiză. Inainte de proiectarea bazei de date se alege tipul de sistem de gestiune a bazei de date. Alegerea SBGD-ului se face ţinând cont de două aspecte: cerinţele aplicaţiei (utilizatorului) şi performanţele tehnice ale SGBD-ului.

Cerinţele aplicaţiei se referă la: volumul de date estimat a fi memorat şi prelucrat în BDR; complexitatea problemei de rezolvat; ponderea şi frecvenţa operaţiilor de intrare/ieşire; condiţiile privind protecţia datelor; operaţiile necesare (încărcare/validare, actualizare, regăsire etc.); particularităţile activităţii pentru care se realizează baza de date.

*Performanţele tehnice* ale SGBD-ului se referă la: modelul de date pe care-l implementează; ponderea utilizării SGBD-ului pe piaţă şi tendinţa; configuraţia de calcul minimă cerută; limbajele de programare din SGBD; facilităţile de utilizare oferite pentru diferite categorii de utilizatori; limitele SGBD-ului; optimizările realizate de SGBD; facilităţile tehnice; lucrul cu mediul distribuit şi concurenţa de date; elementele multimedia; instrumentele CASE; interfeţele de comunicare; posibilitatea de autodocumentare; instrumentele specifice oferite.

*Proiectarea BDR* se realizează prin proiectarea schemelor BDR şi proiectarea modulelor funcţionale specializate.

*Schemele bazei de date* sunt: conceptuală, externă şi internă.

1. *Proiectarea schemei conceptuale* porneşte de la identificarea setului de date necesar sistemului. Aceste date sunt apoi integrate şi structurate într-o schemă (exemplu: pentru BDR relaţionale cea mai utilizată tehnică este *normalizarea*). Pentru acest lucru se parcurg paşii:

* Stabilirea schemei conceptuale iniţiale care se deduce din modelul entitate-asociere (vezi analiza structurală). Pentru acest lucru, se transformă fiecare entitate din model într-o colecţie de date (fişier), iar pentru fiecare asociere se definesc cheile aferente. Dacă rezultă colecţii izolate, acestea se vor lega de alte colecţii prin chei rezultând asocieri (*1:1, 1:m, m:n)*.
* Ameliorarea progresivă a schemei conceptuale prin eliminarea unor anomalii (exemplu: cele cinci forme normale pentru BDR relaţionale).
* Stabilirea schemei conceptuale finale trebuie să asigure un echilibru între cerinţele de actualizare şi performanţele de exploatare (exemplu: o formă normală superioară asigură performanţe de actualizare, dar timpul de răspuns va fi mai mare).

*Tehnica de normalizare* este utilizată în activitatea de proiectare a structurii BDR şi constă în eliminarea unor anomalii (neajunsuri) de actualizare din structură.

*Anomaliile de actualizare* sunt situaţii nedorite care pot fi generate de anumite tabele în procesul proiectării lor:

* *Anomalia de ştergere* semnifică faptul că stergând un tuplu dintr-o tabelă, pe lângă informaţiile care trebuie şterse, se pierd şi informaţiile utile existente în tuplul respectiv;
* *Anomaliile de adăugare* semnifică faptul că nu pot fi incluse noi informaţii necesare într-o tabelă, deoarece nu se cunosc şi alte informaţii utile (de exemplu valorile pentru cheie);
* *Anomalia de modificare* semnifică faptul că este dificil de modificat o valoare a unui atribut atunci când ea apare în mai multe tupluri.

*Normalizarea* este o teorie construită în jurul conceptului de *forme normale* (FN), care ameliorează structura BDR prin înlăturarea treptată a unor neajunsuri şi prin imprimarea unor facilităţi sporite privind manipularea datelor.

Normalizarea utilizează ca metodă *descompunerea* (top-down) unei tabele în două sau mai multe tabele, păstrând informaţii (atribute) de legătură.

**FN1.** O tabelă este în FN1 dacă toate atributele ei conţin valori elementare (nedecompozabile), adică fiecare tuplu nu trebuie să aibă date la *nivel de grup* sau *repetitiv*. Structurile de tip arborescent şi reţea *se transformă* în tabele cu atribute elemntare.

O tabelă în FN1 prezintă încă o serie de *anomalii* de actualizare datorită eventualelor dependenţe funcţionale incomplete.

*Fiecare* structură repetitivă generează (prin descompunere) o nouă tabelă, iar atributele la nivel de grup se înlătură, rămânând doar cele elemntare.

**FN2.** O tabelă este în FN2 dacă şi numai dacă este în FN1 şi fiecare atribut noncheie al tabelei este *dependent funcţional complet* de cheie. Un atribut B al unei tabele *depinde funcţional* de atributul A al aceleiaşi tabele, dacă fiecărei valori a lui A îi corespunde o singură valoare a lui B, care îi este asociată în tabelă. Un atribut B este *dependent funcţional complet* de un ansamblu de atribute A în cadrul aceleiaşi tabele, dacă B este dependent funcţional de întreg ansamblul A (nu numai de un atribut din ansamblu).

O tabelă în FN2 prezintă încă o serie de *anomalii* de actualizare, datorită eventualelor dependenţe tranzitive.

*Eliminarea* dependenţelor incomplete se face prin descompunerea tabelei iniţiale în două tabele, ambele conţinând atributul intermediar (B).

**FN3**. O tabelă este în FN3 dacă şi numai dacă este în FN2 şi fiecare atribut noncheie depinde în mod *netranzitiv* de cheia tabelei. Într-o tabelă T, fie A,B,C trei atribute cu A cheie. Dacă B depinde de A (A 🡪B) şi C depinde de B (B 🡪C) atunci C depinde de A în mod *tranzitiv*. *Eliminarea* dependenţelor tranzitive se face prin descompunerea tabelei iniţiale în două tabele, ambele conţinând atributul intermediar (B).

O tabelă în FN3 prezintă încă o serie de *anomalii* de actualizare, datorate eventualelor dependenţe multivaloare.

O definiţie mai riguroasă pentru FN3 a fost dată prin forma intermediară **BCNF** (Boyce Codd Normal Form): o tabelă este în BCNF dacă fiecare determinant este un candidat cheie.Determinantul este un atribut elementar sau compus faţă de care alte atribute sunt complet dependente funcţional.

**FN4.** O tabelă este în FN4 dacă şi numai dacă este în FN3 şi nu conţine două sau mai multe dependenţe *multivaloare*. Într-o tabelă T, fie A,B,C trei atribute. În tabela T se menţine dependenţa multivaloare A dacă şi numai dacă mulţimea valorilor lui B ce corespunde unei perechi de date (A,C), depinde numai de o valoare a lui A şi este independentă de valorile lui C.

**FN5.** O tabelă este în FN5 dacă şi numai dacă este în FN4 şi fiecare *dependenţă joncţiune* este generată printr-un candidat cheie al tabelei. În tabela T (A,B,C) se menţine *dependenţa joncţiune* (AB, AC) dacă şi numai dacă T menţine dependenţa multivaloare A -->> B sau C.

Dependenţa multivaloare este caz particular al dependenţei joncţiune. Dependenţa funcţională este caz particular al dependenţei multivaloare.

*b) Proiectare schemei externe* are rolul de a specifica viziunea fiecărui utilizator asupra BDR. Pentru acest lucru, din schema conceptuală se identifică datele necesare fiecărei viziuni. Datele obţinute se structurează logic în subscheme ţinând cont de facilităţile de utilizare şi de cerinţele utilizator. Schema externă devine operaţională prin construirea unor viziuni (view) cu SGBD-ul şi acordarea drepturilor de acces. Datele într-o viziune pot proveni din una sau mai multe colecţii şi nu ocupă spaţiul fizic.

*c) Proiectarea schemei interne* presupune stabilirea structurilor de memorare fizică a datelor şi definirea căilor de acces la date. Acestea sunt specifice fie SGBD-ului (scheme de alocare), fie sistemului de operare. Proiectarea schemei interne înseamnă estimarea *spaţiului fizic* pentru BDR, definirea unui model fizic de alocare (a se vedea dacă SGBD-ul permite explicit acest lucru) şi definirea unor *indecşi* pentru accesul direct, după cheie, la date.

*Proiectarea modulelor funcţionale* ţine cont de concepţia generală a BDR, precum şi de schemele proiectate anterior. În acest sens, se proiectează fluxul informaţional, modulele de încărcare şi manipulare a datelor, interfeţele specializate, integrarea elementelor proiectate cu organizarea şi funcţionarea BDR.

*3) Realizarea componentelor logice.* Componentele logice ale unei BD sunt programele de aplicaţie dezvoltate, în cea mai mare parte, în SGBD-ul ales. Programele se realizează conform modulelor funcţionale proiectate în etapa anterioară. Componentele logice ţin cont de ieşiri, intrări, prelucrări şi colecţiile de date. În paralel cu dezvoltarea programelor de aplicaţii se întocmesc şi documentaţiile diferite (tehnică, de exploatare, de prezentare).

*4) Punerea în funcţiune şi exploatarea.* Se testează funcţiile BDR mai întâi cu date de test, apoi cu date reale. Se încarcă datele în BDR şi se efectuează procedurile de manipulare, de către beneficiar cu asistenţa proiectantului. Se definitivează documentaţiile aplicaţiei. Se intră în exploatare curentă de către beneficiar conform documentaţiiei.

*5) Dezvoltarea sistemului.* Imediat după darea în exploatare a BDR, în mod continuu, pot exista factori perturbatori care generează schimbări în BDR. Factorii pot fi: organizatorici, datoraţi progresului tehnic, rezultaţi din cerinţele noi ale beneficiarului, din schimbarea metodologiilor etc.

**Tehnologii folosite**

**Programarea orientata pe obiecte**

Programarea orientată pe obiecte (POO, uneori şi **Programarea orientată obiect**) este o paradigma de programare. Termenul englez corespunzător este *Object Oriented Programming (OOP)*.

**Clase si obiecte**

Programarea orientată obiect este unul din cei mai importanţi paşi făcuţi în evoluţia limbajelor de programare spre o mai puternică abstractizare în implementarea programelor. Ea a apărut din necesitatea exprimării problemei într-un mod mai natural fiinţei umane. Astfel unitaţile care alcătuiesc un program se apropie mai mult de modul nostru de a gândi decât modul de lucru al calculatorului. Până la apariţia programării orientate pe obiect programele erau implementate în limbaje de programare procedurale(C, Pascal) sau în limbaje care nici măcar nu ofereau o modalitate de grupare a instrucţiunilor în unitaţi logice(funcţii, proceduri) cum este cazul limbajului de asamblare(assembler). Altfel spus o problemă preluată din natură trebuia fragmentată în repetate rânduri astfel încât sa se identifice elementele distincte, implementabile într-un limbaj de programare.

O mare problemă a programării procedurale era separarea datelor de unitaţile care prelucrau datele(subrutinele), ceea ce făcea foarte dificilă extinderea şi întreţinerea unui program. Astfel s-a pus problema ca aceste doua entităţi(date şi subrutine) să fie grupate într-un anumit mod, astfel încât subrutinele sa "ştie" în permanenţă ce date prelucrează şi, mai mult decăt atât, ele sa formeze un modul, adică o unitate care separă implementarea de interfaţă, ceea ce implica posibilitatea refolosirii codului. A apărut astfel conceptul de clasa. Clasa realizează, în speţă, ceea ce am văzut mai înainte: grupează datele şi unitaţile de prelucrare a acestora într-un modul, unindu-le astfel într-o entitate mult mai naturală. Deşi tehnica se numeşte "Programare Orientată Obiect", conceptul de bază al ei este este Clasa. Clasa, pe lângă faptul că abstractizează foarte mult analiza/sinteza problemei are proprietatea de generalitate, ea desemnând o mulţime de obiecte care împart o serie de proprietăţi.

De exemplu: Clasa "floare" desemnează toate plantele care au flori, precum clasa "Fruct" desemnează toate obiectele pe care noi le identificam ca fiind fructe. Bineînţeles, în implementarea efectiva a programului nu se lucrează cu entităţi abstracte, precum clasele ci se lucrează cu obiecte, care sunt "instanţieri" ale claselor. Altfel spus, plecând de la exemplul de mai sus, dacă se construieşte un program care să lucreze cu fructe, el nu va prelucra entitatea "fruct" ci va lucra cu entităţi concrete ale clasei "fruct", adică "măr", "pară", "portocală", etc.

Apare însă următoarea problemă: "cum se trece de la o structură generală la una particulară, mai precis ce înseamnă efectiv procesul de instanţiere?". Instanţierea(trecerea de la clasă la obiect) înseamnă atribuirea unor proprietăţi specifice clasei, astfel încât aceasta să indice un obiect anume, care se diferenţiază de toate celelalte obiecte din clasă printr-o serie de atribute. Dacă vom considera ca "fruct\_exotic" care desemnează clasa tuturor fructelor exotice conţine proprietatea "culoare" atunci atribuind acesteia valoarea "galben" noi vom crea o noua mulţime( clasa fructelor exotice care au culoarea galbenă ) care este o subclasă a clasei "fruct\_exotic", deci realizăm astfel o particularizare. Mai mult decât atât, dacă vom adăuga noi şi noi atribute vom individualiza clasa astfel încât să ajungem la un caz concret, care este Obiectul.

**Obiectele comunica intre ele**

Odată identificate entităţile( în speţa clasele ) ele nu rămân izolate; ele vor fi grupate în module, pachete, programe, etc., care vor stabili legături între ele. Aceste legături reflectă relaţiile care se stabilesc între clasele/obiectele problemei pe care am preluat-o din natura. Extinzând exemplul de mai sus, vom adăuga o nouă clasă: "Raft", care va avea următoarele proprietăţi: "număr" si "conţinut". Vom instanţia(particulariza) clasa "Raft" atribuind atributelor "număr" valoarea "1" şi "conţinut" valoarea "fructe". Aceasta înseamnă că am creat un obiect al clasei "Raft" care: "este primul din magazin şi conţine fructe". Bineînţeles că acest raft va fi în relaţie cu, clasa "Fruct" pe care am exemplificat-o mai devreme. Astfel el conţine obiecte de tip "Fruct".

Relaţia pe care am enunţat-o mai sus se mai numeşte şi relaţie de compunere, o relaţie fundamentală în POO, iar clasa "Raft" se numeşte clasă compusa(eng. Agregate), fiindcă în componenţa ei intră alte clase, în cazul nostru "Fruct", cum se vede în diagrama de mai jos:

1

n

**Fruct**

**Raft**

Un raft contine mai multe fructe(n)

Să considerăm în continuare că în magazin avem şi fructe care trebuie păstrate la temperaturi joase. Pentru ele vom avea nevoie de un raft special. Acest nou raft(să-l numim raft frigorific) este în esenţa tot un raft doar că are în plus proprietatea de răcire. Acest lucru ne duce cu gândirea la faptul că putem reutiliza codul scris pentru "Raft" pentru a implementa o clasă numită "RaftFrigorific". Altfel spus, dorim ca noua noastră clasă să fie o subclasă a clasei "Raft" deoarece ea "are toate proprietăţile clasei "Raft" plus altele particulare, care o diferenţiază".

Acest lucru, în mod intuitiv, este numit moştenire. Moştenirea este o relaţie statică(definită în momentul programării) care pune în legătură o clasă cu alta(sau mai multe) astfel încât clasa rezultată să "preia" toate atributele clasei/claselor pe care o/le moşteneşte. Clasa care moşteneşte atributele altei clase se numeşte "clasă derivată" iar clasa care are moştenitori se numeşte "clasă de bază". Terminologia este, şi în acest caz, intuitivă. Unul din avantajele moştenirii, care se observă direct, este acela al reutilizării codului: clasa derivată nu va mai implementa metodele clasei de baza, ci va implementa numai metodele ei specifice; mai mult clasa derivată va conţine (prin intermediul moştenirii) toate atributele(date şi subrutine sau "metode") ale clasei de bază. Astfel spus şi clasa "RaftFrigorific" va avea atributele "număr" si "conţinut".

Următoarea diagramă ilustrează moştenirea:

**RaftFrigorific**

**Raft**

Clasa RaftFrigorific moşteneşte clasa Raft

Moştenirea este de asemeni o relaţie fundamentala în POO, care este recunoscuta ca fiind un principiu de baza, alături de Încapsulare, şi Polimorfism;

**Clasele se comporta diferit în funcţie de tipul lor**

Am văzut mai sus ca o clasa poate moşteni o alta clasă, ceea ce înseamnă că o entitate preia toate atributele altei entităţi. Putem avea, de asemeni, mai multe clase care moştenesc o clasa. Fie clasa "A" şi clasele "B", "C" şi "D"; sa presupunem ca "B", "C", "D" moştenesc pe "A". În acest caz putem face afirmaţia: "B, C şi D sunt de tip A". Aceasta înseamnă că B, C şi D moştenesc toate caracteristicile clasei A deci pot fi identificate cu o clasa de tip A.

Făcând o analogie cu exemplul anterior, dacă A -> "Fruct", B -> "mar", C -> "nuca", D -> "pruna", afirmaţia de mai sus ar suna în felul următor: "mărul, para şi nuca sunt fructe", lucru care este în mod evident adevărat. Dar sa consideram următoarea situaţie: "Fruct" poate implementa o subrutină pe care o vom numi "mănâncă-mă!". Deoarece, prin moştenire, clasele "Mar", "Par" şi "Nuca" preiau toate atributele clasei "Fruct", deci şi subrutina "mănâncă-mă!" înseamnă că un măr se mănâncă la fel ca o pruna! Bineînţeles, dacă încercăm să muşcăm cu pofta dintr-o nuca... Asta înseamnă ca "mănâncă-mă!" trebuie sa se particularizeze pentru fiecare clasa in parte, reflectând un comportament adecvat cu proprietăţile acesteia. Acest lucru mai este numit şi polimorfism.

El este al treilea şi cel mai important principiu al POO. El asigura faptul ca fiecare clasa se comporta diferit(polimorfic) la un anumit mesaj trimis către obiect(în cazul nostru, apelul subrutinei "mănâncă-mă!"). Se observa că polimorfismul există în strânsă legătură cu moştenirea, fără de care nu ar exista. El aduce însa o nuanţa moştenirii: " Preiau din clasa de baza doar acele proprietăţi care sunt comune şi reflectă un comportament adecvat structurii mele". Altfel spus, prin polimorfism pot face o moştenire selectivă.

**Principii de baza**

* Abstractizarea – Este posibilitatea ca un program să ignore unele aspecte ale informaţiei pe care o manipulează, adică posibilitatea de a se concentra asupra esenţialului. Fiecare obiect în sistem are rolul unui “actor” abstract, care poate executa acţiuni, îşi poate modifica şi comunica starea şi poate comunica cu alte obiecte din sistem fără a dezvălui cum au fost implementate acele facilitaţi. Procesele, funcţiile sau metodele pot fi de asemenea abstracte, şi în acest caz sunt necesare o varietate de tehnici pentru a extinde abstractizarea:
* Încapsularea – numită şi ascunderea de informaţii: Asigură faptul că obiectele nu pot schimba starea internă a altor obiecte în mod direct (ci doar prin metode puse la dispoziţie de obiectul respectiv); doar metodele proprii ale obiectului pot accesa starea acestuia. Fiecare tip de obiect expune o interfaţă pentru celelalte obiecte care specifică modul cum acele obiecte pot interacţiona cu el.
* Polimorfismul – Este abilitatea de a procesa obiectele în mod diferit, în funcţie de tipul sau de clasa lor. Mai exact, este abilitatea de a redefini metode pentru clasele derivate. De exemplu pentru o clasă Figura putem defini o metodă arie. Dacă Cerc, Dreptunghi, etc. vor extinde clasa Figura, acestea pot redefini metoda arie.
* Moştenirea – Organizează şi facilitează polimorfismul şi încapsularea, permiţând definirea şi crearea unor clase specializate plecând de la clase (generale) deja definite - acestea pot împărtăşi (şi extinde) comportamentul lor, fără a fi nevoie de a-l redefini. Aceasta se face de obicei prin gruparea obiectelor în clase şi prin definirea de clase ca extinderi ale unor clase existente. Conceptul de moştenire permite construirea unor clase noi, care păstrează caracteristicile şi comportarea, deci datele şi funcţiile membru, de la una sau mai multe clase definite anterior, numite clase de bază, fiind posibilă redefinirea sau adăugarea unor date şi funcţii noi. Se utilizează ideea: ”Anumite obiecte sunt similare, dar în acelaşi timp diferite”. O clasă moştenitoare a uneia sau mai multor clase de bază se numeşte clasă derivată. Esenţa moştenirii constă în posibilitatea refolosirii lucrurilor care funcţionează.

**Concluzii**

Ideea POO este de a crea programele ca o colecţie de obiecte, unităţi individuale de cod care interacţionează unele cu altele, în loc de simple liste de instrucţiuni sau de apeluri de proceduri (vezi şi programare procedurală).

Obiectele POO sunt de obicei reprezentări ale obiectelor din viaţa reală (*domeniul problemei*), astfel încât programele realizate prin tehnica POO sunt mai uşor de înţeles, de depanat şi de extins decât programele procedurale. Aceasta este adevărată mai ales în cazul proiectelor software complexe şi de dimensiuni mari, care se gestionează făcând apel la ingineria programării.

**Platforma Netbeans**

Ce este un „Rich Client” (client bogat)?

Intr-o arhitectura client – server termenul de „rich - client” este folosit pentru clienti unde procesarea datelor apare in principal pe partea de client. Clientul ofera de asemenea interfata grafica cu utilizatorul. Deseori, clientii bogati sunt aplicatii extensibile prin asa numitele plugin-uri (le-am putea traduce ca adaosuri), sau prin module. Astfel, clientii bogati sunt capabili sa rezolve mai mult de o singura problema. Clientii bogati pot de asemenea sa rezolve probleme asemanatoare, cat si probleme ce nu au absolut nici o legatura cu scopul initial.

Clientii bogati sunt de obicei dezvoltati pe baza unei infrastruncturi („framework”). Un framework ofera un punct de start pe baza caruia utilizatorul poate asambla intr-o maniera logica parti legate ale aplicatiei, numite module. In mod ideal, solutiile fara legatura (cum ar fi cele facute disponibile de diversi producatori) pot functiona impreuna, asa incat toate modulele par a fi create ca un intreg. Dezvoltatorii de software si vanzatorii, pot combina de asemenea distributii de clienti bogati corespunzatori unor module diferite, cu scopul de a le face disponibile anumitor utilizatori.

In plus de asta, clientii bogati au avantajul ca sunt usor de distribuit si actualizat, precum o functie automata online de actualizare din cadrul clientului insasi, sau printr-un mecanism ce permite clientului sa fie pornit prin intermediul internetului (de exemplu, prin Java web Start).

In continuare sunt prezentate o serie de caracteristici ale unui client bogat:

* Arhitectura a aplicatiei flexibila si modulara
* Independenta platformei
* Adaptabilitate la utilizatorul final
* Abilitatea de a functiona online cat si offline
* Distributie simplificata catre utilizatorul final
* Actualizare simplificata a clientului

**Ce este o platforma de „Rich Client”**

O platforma de client bogat este o aplicatie tip mediu ciclu de viata („application lifecycle environment”), o baza pentru aplicatiile desktop. Majoritatea aplicatiilor de tip Desktop au caracteristici similare, cum ar fi meniuri, bare de unelte, bare de stare, vizualizari de progres, afisari de date, setari ce pot fi particularizate, salvari si incarcari de configurari de date pentru utilizatorul final, ecrane de tip „splash”, ferestre de informatii, internationalizare, sisteme de ajutor, si asa mai departe. Pentru acestea, precum si alte caracteristici tipice aplicatiilor client, o platforma rich client ofera o infrastructura cu ajutorul careia functionalitatile pot fi alaturate usor si simplu.

Configurabilitatea si extensibilitatea unei aplicatii sunt cele mai importante caracteristici ale unei infrastructuri de acest tip. Ca rezultat, este posibila, de exemplu, oferirea declarativa a intrarilor din meniul unei aplicatii intr-un fisier text, dupa care meniul va putea fi incarcat automat de infrastructura. Asta inseamna ca, codul sursa devine mult mai concentrat si maleabil, iar dezvoltatorilor li se permite sa puna accent mai mult pe necesitatile efective ale partii de afaceri a aplicatiei, in timp ce meniul este configurabil cat de mult posibil.

Cel mai important aspect al unei platforme client bogat este arhitectura sa. Aplicatiile bazate pe o platforma client bogat sunt scrise sub forma de module, in cadrul carora parti de program cu coerenta logica sunt izolate. Un modul este descris in mod declarativ si incarcat in mod automat de platforma. Ca rezultat, nu exista o legatura explicita necesara intre codul sursa si aplicatie. In aceasta maniera, este de asemenea foate usor sa se asambleze aplicatii specifice utilizatorului sau domeniului din module individuale.

O platforma de client bogat de asemenea elibereaza dezvoltatorul de a fi ingrijorat de sarcini cu functionalitate redusa fata de logica de afaceri a aplicatiei. La sfarsitul ciclului de dezvoltare, se obtine o aplicatie cu o arhitectura moderna bine meritata.

**Avantajele unei platforme „Rich Client”**

In afara de modularitatea oferita de o arhitectura de client bogat, ce implica simultan un grad ridicat de robustete si valoare oferita utilizatorului final, ajutorul continuu al dezvoltarii oferit trebuie de asemenea accentuat. Acestea, precum si alte avantaje ale unei platforme de client bogat sunt descrise pe scurt in continuare:

* Reducerea timpului de dezvoltare
* Consistenta interfetei cu utilizatorul
* Reactualizarea
* Independenta platformei
* Reutilizarea si fiabilitatea

**Reducerea timpului de dezvoltare**

O platforma de client bogat ofera o multitudine de API-uri pentru dezvoltarea aplicatiilor de tip desktop. De exemplu, acestea pot fi folosite de dezvoltatori pentru a manipula ferestre si meniuri sau pentru a asigura afisarea de optiuni ce pot fi particularizate. Prin refolosirea multor componente predefinite, dezvoltatorii se pot concentra foarte mult pe logica de afaceri a aplicatiei in cauza.

**Consistenta interfetei cu utilizatorul**

Usurinta de utilizare a unei aplicatii este tot timpul o problema cruciala, in special cand aplicatia este destinata pentru a fi folosita de profesionisti ai unui domeniu particular. O platforma de client bogat face disponibila o infrastructura pentru afisarea de interfete cu utilizatorul, avand grija in mod particular de consistenta, accesibilitate si usurinta in utilizare.

**Reactualizarea**

Folosing o platforma de client bogat, face posibila distribuirea rapida si eficienta de module noi sau actualizate catre utilizatorii finali. Ca rezultat, nu toti clientii unei aplicatii trebuie sa fie informati de dezvoltatori pentru a trece la o versiune noua. Actualizari pot fi distribuite si instalate sub forma de module, pentru ca diferite caracteristici sa poata fi dezvoltate si livrate de echipe de operare complet independente. Arhitectura modulara a aplicatiei asigura posibilitatea ca module complete sa poata fi distribuite fara a fi nevoie sa se astepte pentru finalizarea altor module.

**Independenta platformei**

Platforme de tip client bogat, sunt bazate pe standarde internationale si componente reutilizabile. Ca rezultat, aplicatiile Java bazate pe ele pot fi pornite in mod automat pe mai multe sisteme diferite, cum ar fi Windows, Linux, atata timp cat o implementare a mediului de dezvoltare Java (Java Runtime Development) este disponibil. Cat timp setul de caracteristici si aplicabilitatea aplicatiilor este in continua schimbare, este foarte important ca acestea sa fie dezvoltate intr-o maniera anume incat sa fie extensibile si sa poata fi pornite pe sisteme tinta diferite. Toate aceste lucruri sunt asigurate de o platforma de client bogat, salvand timp si bani. Aplicatiile bazate pe platforme de client bogat nu au nevoie de librarii aditionale sau alte componente, altele decat cele din mediul de rulare Java (Java Runtime Environment - JRE).

**Reutilizarea si Fiabilitatea**

Platformele de client bogat fac disponibila o gama larga de caracteristici si module, ce pot fi utilizate in propriile aplicatii ale dezvoltatorului. Daca modulul nu se potriveste in totalitate cu cerintele aplicatiei, este posibil ca acesta sa fie folosit ca punct de pornire, extinzandu-l sau schimbandu-l in functie de necesitati. Avand in vedere ca majoritatea platformelor isi fac si codul sursa disponibil, s-ar putea de asemenea, ca in anumite cazuri, sa se merite luarea in considerare a schimbarii sau chiar extinderii insasi platformei. Acesti factori implica un grad mai ridicat de fiabilitate si independenta.

**Caracteristici ale platformei Netbeans**

Platforma Netbeans ofera, pe langa avantajele generice ale unei platforme de client bogat, numeroase alte infrastructuri si alte caracteristici particulare ce pot fi folositoare unei aplicatii. Cele mai importante, ce constituie caracteristicile principale ale platformei Netbeans, sunt prezentate in continuare:

* Infrastructura interfetei cu utilizatorul
* Editorul de date
* Afisajul particularizabil
* Infratructura Wizard
* Sisteme de date
* Internationalizare
* Sistem de ajutor

**Infrastructura interfetei cu utilizatorul**

Ferestrele, meniurile, barele de unelte, precum si alte componente sunt facute disponibile prin intermediul platformei. Ca rezultat, concentrarea utilizatorului se muta pe actiuni particulare, ce duc la condensarea codului., facandu-l mai bun si mai putin predispus la erori. Interfata cu utilizatorul completa oferita de platforma Netbeans este bazata in proportie de 100% pe AWT Swing, si poate fi extinsa cu componente proprii.

**Editorul de date**

Editorul de date Netbeans este o unealta puternica din cadrul sistemului inteligent de dezvoltare Netbeans (Netbeans IDE), si poate fi utilizat de propriile aplicatii ale utilizatorului. Functionalitatea si uneltele editorului pot fi extinse si adaptate cu usurinta la scopul aplicatiei.

**Afisajul ce poate fi particularizat**

Un afisaz particularizat la necesitatile specifice ale utilizatorului este cautat in fiecare aplicatie. Platforma Netbeans face disponibila o infrastructura, facand foarte simpla introducerea de optiuni proprii, permitand utilizatorului sa salveze setari intr-o maniera atractiva din punct de vedere vizual.

**Infrastructura Wizard**

Platforma Netbeans ofera unelte foarte simple pentru creerea si extinderea de functionalitati tip asistent cu mare usurinta ce pot ghida utilizatorul printr-o serie de pasi complecsi in cadrul aplicatiei.

**Sistemele de date**

Din punct de vedere al platformei Netbeans, date pot fi facute disponibile atat local cat si prin intermediul unui server FTP, CVS, a unui fisier XML, sau a unei baze de date. Prin intermediul abstractizarii, accesarea datelor de catre un modul este transparenta celorlalte module. Astfel, autoaccesarea informatiilor nu reprezinta o problema, acest lucru fiind efectuat de platforma Netbeans insasi.

**Internationalizare**

Platforma Netbeans ofera clase si metode ce fac disponibila internationalizarea unor resurse precum JavaHelp, si altele. Constante tip text pot fi depozitate cu usurinta in fisiere de proprietati. Platforma Netbeans de asemenea incarca constante tip text si icoane aplicabile tarii curente si setarilor de limba actuale.

**Sistemul de ajutor**

Prin intermediul sistemului standard JavaHelp, platforma Netbeans ofera un sistem centralizat pentru afisarea si integrarea subiectelor de ajutor, catre utilizatorul final. In plus, module aditionale pot sa contribuie la propriile subiecte de interes, venind astfel in ajutorul utilizatorului. In plus fata de acestea, platforma Netbeans lasa utilizatorul sa ofere si informatii de ajutor sensibile la context.

**Structura platformei Netbeans**

Pentru a oferi o imagine de ansamblu a structurii unei aplicatii de tip client bogat, si pentru a arata relatia dintre aplicatie si procesul de dezvoltare, in continuare vom prezenta arhitectura platformei Netbeans. Vom fi descrise de asemenea si blocurile principale ce construiect platforma Netbeans, precum si responsabilitatile si functionalitatile oferite de catre componenta de rulare.

In cele din ruma, structura sistemului de incarcare a claselor va fi explicata, o data cu rolul jucat in cadrul procesului de construire a aplicatiei de catre platforma Netbeans.

**Arhitectura platformei Netbeans**

Dimensiunea si complexitatea aplicatiilor moderne a crescut continuu o data cu trecerea timpuilui. In acelasi timp, aplicatiile profesionale trebuie sa fie, inainte de orice altceva, flexibile, pentru a putea si extinse cu usurinta si rapiditate. Acest lucru face atractiva impartirea unei aplicatii in mai multe componente distincte.

Ca rezultat, fiecare componenta distincta trebuie sa fie independenta, facand accesibile interfete bine definite ce sunt folosite de alte parti ale aceleiasi aplicatii, cu functionalitati ce pot fi extinse si folosite de alte parti. Impartirea aplicatiei in module, - ca parti independente din punct de vedere logic - , duce la cresterea considerabila a arhitecturii unei aplicatii. Spre deosebire de aplicatiile monolitice, in care fiecare clasa poate folosi cod din orice alta clasa, arhitectura in acest caz este mult mai flexibila, si mai important, mai usor de intretinut. De asemenea este posibila protejarea unei clase de accesul catre aceasta din lumea exterioara, desi o asemenea protectie la nivel de clasa este mult prea in detaliu pentru a fi folositoare majoritatii aplicatiilor. Acesta este exact aspectul central al aplicatiilor client moderne, implementat cu bri de platforma Netbeans. Conceptele si structurile sale sustin dezvoltarea si conceperea de apicatii flexibile si modulare. Blocul de constructie principal al platformei Netbeans este *modulul*. Un modul este o colectie de clase legate ca functionalitate, precum si o descriere a interfetelor expuse de acesta, precum si o descriere a celorlalte module necesare functionarii.

Platforma Netbeans completa, precum si aplicatia contruita pe baza acesteia, este impartita in module. Acestea sunt incarcate de nucleul platformei Netbeans, cunoscut ca si *containerul de rulare Netbeans* (Netbeans runtime container). Acesta incarca modulele aplicatiei intr-o maniera dinamica si automata, fiind ulterior responsabil si cu rularea efectiva a aplicatiei. Astfel, mediul de dezvoltare Netbeans este un exemplu foarte bun de aplicatie de tip client bogat. Functionalitatile si caracteristicile unui IDE (Integrated development environment), cum ar fi suportul pentru limbajul Java, sau editorul de cod, sunt create sub forma de module bazate pe platforma Netbeans, precum este infatisat in figura de mai jos. Acest lucru aduce cu sine marele avantaj ca aplicatia poate fi extinsa de module aditionale, si ca poate fi adaptata necesitatilor particulare ale utilizatorului, permitand dezactivarea sau chiar dezinstalarea modulelor ce nu sunt folosite.

****

**Structura conceptuala a IDE-ului Netebeans**

Pentru a permite aplicatiilor sa atinga acest nivel de modularitate, platforma Netbeans pe de o parte face disponibile mecanisme si concepte ce permit modulelor sa fie extensibile prin intermediul altor module, iar pe de alta parte permite intercomunicarea dintre acestea fara a fi dependente unul de altul. In alte cuvinte, platforma Netbeans suporta asa numitul fenomen de „*loose coupling*”in cadrul unei aplicatii.

Pentru a optimiza incapsularea codului in module, aspect necesar intr-un sistem de tip modular, platforma Netbeans face disponibil propriul sistem de incarcare a claselor. Fiecare modul este incarcat de propriul sistem de incarcare a claselor („*classloader*”), iar, in cadrul procesului, face disponibila o portiune de cod separata. Ca rezultat, un modul isi poate face disponibile pachetele, facandu-si totodata disponibile functionalitatile catre celelalte module. Pentru a folosi functionalitati de la alte module, un modul poate fi declara dependente fata de acestea. Aceste dependente sunt declarate in fisierul manifest al modulului, si rezolvate de containerul Netbeans de rulare, asigurand faptul ca aplicatia tot timpul va fi pornita intr-o stare consistenta. Mai mult decat orice altceva, aceasta legatura lejera joaca un rol important in conceptul declarativ al platformei Netbeans. Prin aceasta spunem ca, in cadrul fisierelor descriptive si de configuratii este declarat pe cat de mult posibil, in scopul de a evita o conexiune fortata a acestor concepte cu codul sursa Java. Un modul este descris de informatiile din fisierul manifest al acestuia, precum si de informatiile oferite de fisierele aferente de tip XML, asadar neavand nevoie sa fie adaugat in mod explicit la platforma Netbeans. Folosind fisiere XML, platforma „stie” ca fisierele ce ii sunt disponibile, precum si locatiile acestora, sau problemele ce trebuie rezolvate pentru a face disponibila incarcarea acestora.

Platforma Netbeans insasi este formata dintr-un grup central de module (precum se poate observa in figura de mai jos), ce sunt necesare pentru pornirea aplicatiei si pentru definirea interfetei cu utilizatorul. In acest sens, platforma Netbeans face disponibile un numar mare de API-uri, simplificand in mod considerabil procesul de dezvoltare. Incluse in acest grup (cel din figura), sunt, de exemplu, Actiunile API, ce fac disponibile de multe ori necesarele clase de actiune; puternicele noduri API, si optiunile SPI, cu ajutorul carora propriile optiuni pot fi integrate cu usurinta in aplicatie. Pe langa acestea, mai sunt si alte componente complete reutilizabile ale platformei Netbeans, cum ar fi ferestrele de Output si ferestrele Favorites.

****

**Arhitectura platformei Netbeans**

**Containerul de rulare Netbeans (Netbeans Runtime Container)**

Baza platformei Netbeans si a arhitecturii sale modulare este containerul de rulare Netbeans. Acesta este format din urmatoarele module:

* Bootstrap: acest modul este executat inainte de toate celelalte. Acesta duce la capat toate toate portiunile de cod aferente comenzilor preinregistrate si pregateste si incarcarea claselor , ce duce la incarcarea modulului de Startup.
* Startup: acest modul porneste aplicatia, in momentul in care initializeaza modulul sistem si sistemul de fisiere.
* Sistemul de module: acest modul este responsabil atat cu definirea modulelor cat si cu setarile si dependentele acestora.
* Sistemul de fisiere: aces modul face disponibile date de sistem virtuale, cu acces indpendent la platforma. Este folosit in principal pentru a incarca resurse de module intr-o aplicatie
* Utilitati: aces modul asigura componentele de baza, cum ar fi comunicatia intermodulara.

Containerul de rulare este un subset de module minim necesar aplicatiilor bazate pe platforma Netbeans. Fara necesitatea de alte module sau setari, o plicatie poate fi pornita continand aceste 5 module. In mod direct, dupa rulare aceasta se va opri, alte taskuri nefiind definite. Cand containerul de rulare este ponit, acesta gaseste toate modulele disponibile si construieste din acestea un registru intern. In mod normal, un modul este incarcat doar atunci cand este nevoie de el. De aceea, el este inregistrat la pornire. Un modul are si abilitatea de a executa sarcini in momentul in care este incarcat de containerul de rulare. Acest lucru are loc prin mijloace ale installer-ului de module. Containerul de rulare permite si incarcarea dinamica, descarcarea, instalarea si dezinstalarea de module, toate avand loc la rulare. Aceasta functionalitate este pentru a actualiza o aplicatie de catre utilizator, sau petru dezactivarea de module ce nu sunt necesare.

**Java Swing**

**Scurt istoric**

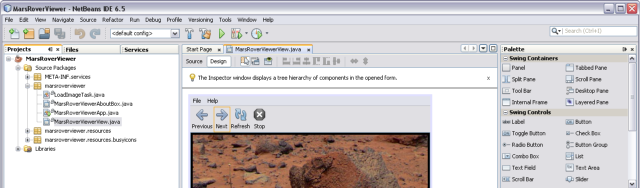
Conform Enciclopediei Britannica, Swing a fost un gen de muzica populara in Statele Unite ale Americii, intre anii 1930 – 1945. Bun, sa zicem ca asta nu ar avea legatura cu Java. In schimb, pe 23 mai 1995, John Gage, directorul departamentului stiintific de la Sun, a facut cunoscut lumii limbajul Java. O data cu nasterea limbajului a aparut ceva numit „Abstract Window Toolkit”, sau AWT. In schimb, AWT a adus cu sine si o serie de mici iregularitati, si o data cu acestea au aparut problemele.

Setul original de componente a venit o data cu platforma Java, AWT fiind dependent de multe aspecte cheie ale platformei initiale. In loc sa ofere un set de componente profesional si matur, Java a pus la dispozitie cea mai simplista versiune posibila. Daca o functionalitate nu era disponibila pe toate platformele Java, nu era disponibila pe nici una dintre platforme. Iar apoi apareau probleme legate de diferentele existente intre browsere/platforme. Fiecare mediu de rulare Java (Java runtime Environment) se baza pe modul in care setul de componente era legat de setul de widgeturi native specifice platformei Java. Daca existau probleme cu aceasta conexiune, in primul rand, ele erau particulare platformei (si/sau browserului), si in al doilea rand codul trebuia scris pentru a ocoli aceste probleme, astfel programele neputand fi scrise dintr-o data, sau rulate oriunde.

O data cu cresterea in popularitate a tehnologiilor Java, utilizatorii au realizat ca AWT era extrem de incet si nefiabil, si nici nu se putea face mare lucru cu componentele oferite. Foarte putine dintre ele erau disponibile, si acestea nu puteau fi folosite intr-un mediu vizual de programare. Astfel, noi tehnologii au fost introduse, cum ar fi „just-in-time” (JIT), compilatoare pentru a imbunatati performanta si, cu ajutorul Borland, JavaBeans pentru o dezvoltare bazata pe componente.

O data cu aceste noi tehnologii au aparut din ce in ce mai multe seturi de widgeturi, in timp ce setul de componente AWT a ramas foarte simplist, de baza. Asadar, timpii de copiere a appleturilor au crescut si au crescut, pentru ca noile seturi de widgeturi nu faceau parte din nucleul platformei Java, si fisiere arhiva Java (JAR) au fost introduse pentru a imbunatati acesti timpi. In cele din urma, toti producatorii principali de browsere si-au adaugat componenta de librarie favorita la propria maisna virtuala – AFC, IFC, si WFC – pentru a numi doar cateva. Cu toate acestea, toate librariile foloseau diferite modele arhitecturale, si nu existau browsere adevarate de legatura standard. In cele din urma, Sun Microsystems a facut echipa cu Netscape Comunication si alti parteneri pentru a crea inca o librarie numita Java Foundation Classes (JFC). Parte a JFC, este asa numitul set de componente Swing.

**Swing GUI Builder**

****

**Constructia profesionala a interfetei cu utilizatorul**

Constructia de interfete profesionale este facuta prin tragerea si pozitionarea componentelor GUI (graphic user interface) dintr-o paleta pe o plansa de lucru. Constructorul GUI se ocupa automat de spatierea si alinierea corecta a componentelor. Prin efectuarea unui simplu click pe componente precum Jlabels, Jbuttons, ButtonGroups, JTrees, JTextFields, ComboBoxes, se pot modifica proprietatile acestora direct in locul in care se afla. Constructorul GUI poate fi folosit pentru realizarea de interfete prototip in fata clientilor.

**Intuitiv si particularizabil**

Se poate alege ca, codul GUI sa fie generat avand nume de clase simple sau nume complet definite. Bara de ajutor afiseaza indicatii sensibile la context legate de ceea ce poate fi facut cu componenta respectiva, precum si scurtaturi ce pot mari viteza de lucru in viitor. Mediul de dezvoltare inteligent Netbeans (IDE), vine si cu suport integrat pentru localizare si accesibilitate. O data cu actualizarea JDK 6 Update 10, toate ferestrele se pot previzualiza in diferite afisaje si modalitati.

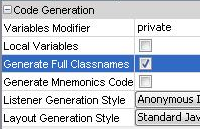
**Componente GUI standard si particulare**

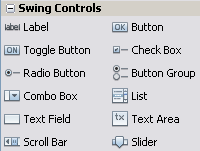
Componenta Pallete extensibila este facuta accesibila cu componente Swing si AWT preinstalate, si include un manager vizual de desenare. Prin folosirea optiunii Inspector se poate vedea o imagine tip arbore a tuturor componentelor precum si a propietatilor acestora.

**Suportul infrastructurii Swing Application (JSR296)**

Infrastructura Swing Application simplifica lucrul cu ciclul de viata al aplicatiilor, a actiunilor si resurselor. Este indicata folosirea uneltelor de generare a codului si a blocurilor de constructie a acestuia, pentru dezvoltarea de aplicatii de dimensiune mica si medie mai repede ca niciodata. Se pot crea meniuri rapide si intuitive prin simpla tragere si plasare a componentelor de meniu pe plansa de lucru; se pot adauga de asemenea comentarii sau redenumi componente de meniu prin simpla selectare a acestora si scrierea in modul de vizualizare Design.

**Suportul tehnologiei Beans Binding (JSR 295)**

 Este indicata de asemenea folosirea cu prisosinta a tehnologiei Beans Binding si a API-urilor Java Persistance pentru crearea de aplicatii desktop cu baze de date mult mai usor. Folosind noul template de proiect Java Desktop Application, se poate crea cu usurinta o fereastra Form folosita pentru a afisa tabelele dintr-o baza de dateRealizarea legaturii dintre un tabel al unei baze de date si o fereastra de timp Form se poate face prin simpla tragere a tabelului din fereastra Runtime peste fereastra Form in cauza.



In continuare vor fi prezentate o serie de componente Swing des folosite:

**Bibliotecile Swing API**

javax.accessibility javax.swing.plaf

javax.swing.text.html javax.swing

javax.swing.plaf.basic javax.swing.text.parser

javax.swing.border javax.swing.plaf.metal

javax.swing.text.rtf javax.swing.colorchooser

javax.swing.plaf.multi javax.swing.tree

javax.swing.event javax.swing.table

javax.swing.undo javax.swing.filechooser

javax.swing.text

javax.swing.plaf

javax.swing

javax.swing.text.parser

javax.swing.plaf.metal

javax.swing.colorchooser

javax.swing.tree

javax.swing.table

javax.swing.filechooser

Cea mai importanta dintre acestea fiind java.swing

**Componentele Swing**

* **Componente atomice**

JLabel, JButton, JCheckBox, JRadioButton,JToggleButton, JScrollBar, JSlider, JProgressBar,

JSeparator

* **Componente complexe**

JTable, JTree, JComboBox, JSpinner, JList, JFileChooser, JColorChooser, JOptionPane

* **Componente pentru editare de text**

JTextField, JFormattedTextField, JPasswordField, JTextArea, JEditorPane, JTextPane

* **Meniuri**

JMenuBar, JMenu, JPopupMenu, JMenuItem, JCheckboxMenuItem, JRadioButtonMenuItem

* **Containere intermediare**

JPanel, JScrollPane, JSplitPane, JTabbedPane, JDesktopPane, JToolBar

* **Containere de nivel inalt**

JFrame, JDialog, JWindow, JInternalFrame, JApplet

**Asemanari si deosebiri cu AWT**

Tehnologia Swing extinde AWT:



**Conventia ”J”**

java.awt.Button - javax.swing.JButton

java.awt.Label - javax.swing.JLabel,etc.

Noi gestionari de pozitionare: BoxLayout si SpringLayout

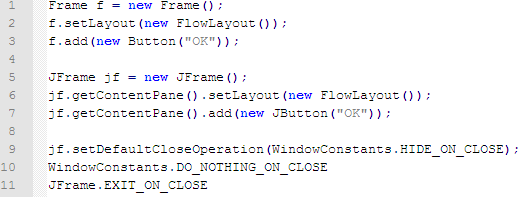
**Folosirea HTML**



**Folsirea ferestrelor**

****

Exemplu de cod pentru initializarea unei ferestre :



**Arhitectura modelului Swing : MVC (Model – View – Controller)**

**Descrierea proiectului**

**Calculul de coordonate pentru cele trei indicatoare pentru controlul manual pornit**

Pentru a putea asigura o imagine dinamica cat mai realista a indicatoarelor pentru viteza, respectiv pentru turatie, a fost necesara impartirea pane-ului (componenta Java Swing) in mai multe zone, exprimate in program prin urmatoarele variabile **:**

* **zona1, zona2** (pentru vitezometru);
* **zona3**, **zona4** (pentru turometru);

Acest lucru a fost necesar in principal datorita faptului ca imaginea indicatorului pentru viteza ciprinde toate cele 4 cadrane ale unui cerc, acul trebuind sa poata realiza o rotatie de 360˚, precum si faptului ca axele de simetrie ale imaginii nu coincid cu axele de coodronate aferente componentei din Swing. Originea spatiului de lucru, fiind astfel in coltul din stanga sus, iar cele doua axe (abscisa si ordonata) avand sensuriledupa cum se poate observa in figura de mai jos :

(0,0)

x



Fig1: spatiul de lucru al pane-ului din Swing

y

Pentru o exemplificare mai detaliata a metodei folosite, vom lua in considerare cazul particular al indicatorului pentru viteza.

Ideea de baza in desenarea dinamica a celor 3 “ace” aferente indicatoarelor, este gasirea a 3 perechi de coordonate, corespunzatoare varfurilor acestora. In cazul vitezometrului, cele doua coordonate calculate sunt reprezentate in program prin variabilele **x** si **y**, asa cum vom vedea intr-o portiune de cod prezentata in continuare.

Asadar, modificarea in mod dinamic a acestor doua coordonate, relativ fata de actiunile utilizatorului, in modul de folosire ce permite controlul manual, sau fata de setul de date citite la rularea unui scenariu de testare, presupune modificarea uneia dintre coordonate (cresterea sau scaderea ei cu o anumita valoare – **pas1**, **pas2**, sau chiar o constanta), si calcularea celei de-a doua coordonate in functie de prima. Asa cum vom vedea, in functie de anumite circumstante aparute datorita spatiului de lucru, coordonata modificata va trebui sa fie abscisa, iar in altele ordonata, pentru obtinerea unui calcul cat mai precis.

 De asemenea, si formula folosita pentru calculul celei de-a doua coordonate va fi diferita, in functie de cele 4 zone in care este impartit pane-ul, asa cum se poate observa in figura de mai jos:

zona de sus

****

****

zona

de mijloc

dreapta

zona

de mijloc

stanga

****

zona de jos

Fig 2 : Impartirea in zone a imaginii indicatorului pentru viteza pentru calculul de coordonate

In program, cele 4 zone sunt exprimate cu ajutorul variabilelor **zona1**, si **zona2**, astfel:

**zona de jos ⇔zona1 = 1 si zona2 = 0 ;**

**zona de sus ⇔zona1 = 0 si zona2 = 1 ;**

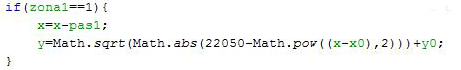
**zona de mijloc stanga ⇔zona1 = 0 si zona2 = 0 si x ≤ 304 ;**

**zona de mijloc dreapta ⇔zona1 = 0 si zona2 = 0 si x ≥ 590 ;**

Formula folosita, pentru calculul in acest caz a coordonatei **y** deriva din ecuatia cercului, ceea ce este un fapt usor de inteles, avand in vedere forma indicatorului, si modul in care acul acestuia trebuie sa se “miste” . Aceasta variaza in functie de cele 4 zone, dupa cum urmeaza :

**daca acul se afla in zona de jos (1)**

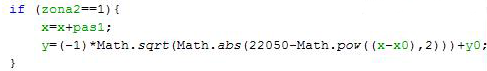
Portiunea de cod unde se realizeaza acest lucru poate fi vazuta in figura de mai jos :



Asa cum se poate observa, variabila este cea care este modificata, iar variabila  este cea calculata. Variabila va varia in functie de valoarea vitezei, determinand ca acul sa isi schimbe pozitia mai repede (la viteze mici) sau mai incet (la viteze mari).

**daca acul se afla in zona de sus (2)**

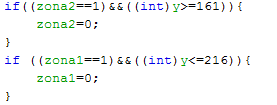
Portiunea de cod unde se realizeaza acest lucru poate fi vazuta in figura de mai jos :



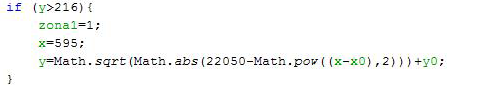
perechea de coordonate (), reprezinta originea cercului descris, sau altfel spus, celalalt capat al acului indicatorului, ce ramane fix pe parcursul functionarii programului.

Pentru cele doua zone de mijloc, coordonata va ramane constanta, iar coordonata **,**  va creste, sau va scade cu o valoare constanta (va scade pentru zona de mijloc stanga si va creste pentru zona de mijloc dreapta). Necesitatea existentei celor doua zone de mijloc, se datoreaza faptului, ca trecerea acului de la la trebuie sa fie una continua, fapt ce nu se intampla, daca pur si simplu se trece de la formula **(1)** la formula **(2).**

Se pune acum problema schimbarii intr-o maniera dinamica a acestor zone, in functie de valorile celor doua coordonate. Acest lucru este realizat in program prin metoda , unde se face trecerea din in zona de mijloc dreapta precum si trecerea din in zona de mijloc stanga, astfel:



Iesirea din cele doua zone de mijloc si intrarea in zonele de sus respectiv jos, este facuta tot in functie de valoarea coordonatei , precum se poate observa in figura de mai jos:



(aceasta portiune de cod face trecerea din zona de mijloc dreapta in zona de jos)

Aflandu-ne in cazul de utilizare ce permite controlul manual, modificarea pozitiei acului este realizata in program la detectarea evenimentului de apasare a sagetii de sus, sau a celei de jos de la tastatura, fiecare dintre cele doua cazuri avand cate o metoda aferenta, , respectiv,  **.** Aceste doua metode,coroborate cu cea dedicata setarii zonelor, asigura practic modificarea in mod dinamic si destul de realist a pozitiei acului de indicare a vitezei la momentul actual de timp.

**** In continuare este prezentata una dintre cele doua metode, cea corespunzatoare apasarii sagetii sus de la tastatura , altfel spus accelerarii manuale, pentru a avea o imagine de ansamblu asupra modalitatii prin care se calculeaza aceste coordonatele necesare.

**Calculul vitezei si turatiei**

Este realizata in cadrul programului prin intermediul metodelor si, fiind bazata pe metoda regresiei polinomiale.

**Regresia polinomiala**

O problema binecunoscuta in multe arii de cercetare este aceea care presupune existenta unui set de date privind doua sau mai multe variabile aleatoare, scopul modelarii fiind descrierea relatiei dintre ele in vederea prognozarii valorii uneia in raport cu valorile celeilalte sau celorlalte.

Aceasta problema se pune atunci cand intre variabilele aleatoare considerate exista o legatura consistenta, bazata pe natura intima a fenomenelor care stau la baza lor. Este posibil ca, din punct de vedere formal, doar pe baza datelor numerice, acestea sa para corelate, de exemplu toate sa aiba tendinta de crestere in acelasi timp, acest fapt fiind insa sustinut de natura fenomenelor in cauza.

In cazul in care legatura intre cele doua variabile statistice nu este liniara, si totusi banuim ca exista, avem o regresie neliniara, de exemplu, regresia polinomiala. Atunci, se gaseste curba respectiva de regresie (corespunzatoare, de exemplu, ecuatiei polinomiale de regresie).

Un exemplu de astfel de regresie polinomiala va furniza o ecuatie de forma:

, unde, de obicei, X este asa numita variabila predictoare iar Y este variabila prognozata, adica variabila de raspuns.

Coeficientii necunoscuti  **,** vor fi calculati folosind metoda CMMP (celor mai mici patrate), rezolvand sistemul de necunoscute:

**1 . .**

**1 . .**

**. = . . . . . . .**

**. . . . . . .**

**1 . .**

In cazul calcularii vitezei si turatiei variabila predictoare este una dintre coordonate (ori x ori y), in functie de zona careia ii corespund aceste coordonate sau chiar de valorile lor.

Vom analiza pentru inceput cazul calcularii vitezei. Metoda in cauza folosita este **,** si asa cum am mentionat anterior, formulele folosite depind de anumite zone in care am considerat a fi impartit spatiul de lucru, in cauza, componenta pane din Swing. Impartirea a fost facuta cum putem observa in Fig 3.

Aceasta impartire este datorata imposibilitatii folosirii aceleiasi coordonate ca variabila predictoare, deoarece, dupa cum se poate observa si din figura, variatia coordonatei pentru **zona de sus** este foarte mica, precum si a coordonatei pentru **zona de jos**,iar pentru obtinerea unui polinom capabil sa realizeze o predictie cat mai precisa, variatia trebuie sa fie semnificativa. Asadar, vor fi gasite 3 polinoame prin metoda de regresie polinomiala pentru a acoperi intregul arc de interes al indicatorului pentru viteza.



zona

din dreapta

zona

din stanga

zona

de sus

****

****

zona

de jos

Fig 3 : Impartirea in zone a imaginii indicatorului pentru viteza pentru calculul vitezei

Primul polinom va corespunde zonei de sus , exprimata in program printr-o limitare a valorii coordonatei **,** setul de date corespunzator acestei zone fiind prezentat in continuare, precum si graficul aferent unei regresii polinomiale de ordin 3. Variabila predictoare este in acest caz coordonata , variatia ei in aceasta portiune fiind mult mai semnificativa decat cea a coordonatei

|  |  |
| --- | --- |
| x | v |
| 358 | 55 |
| 379 | 60 |
| 403 | 65 |
| 428 | 70 |
| 454 | 75 |
| 480 | 80 |
| 505 | 85 |
| 529 | 90 |

Polinomul obtinut astfel este cel de mai jos :

Portiunea de cod din cadrul metodei pentru calculul vitezei aferenta acestui calcul este cea de mai jos:



Cel de-al doilea polinom folosit, va corespunde zonei din stanga jos a indicatorului, asadar o combinatie intre **zona din stanga** si **zona de jos** din **Fig3,** variabila predictoare fiind in acest caz coordonata **.** Setul de date corespunzator acestei zone este cel de mai jos, precum si graficul aferent unei regresii polinomiale de ordin 3.

|  |  |
| --- | --- |
| y | v |
| 349 | 0 |
| 326 | 10 |
| 304 | 15 |
| 279 | 20 |
| 254 | 25 |
| 230 | 30 |
| 208 | 34 |
| 181 | 40 |
| 157 | 45 |
| 136 | 50 |
| 117 | 55 |

Polinomul obtinut astfel este cel de mai jos :

Portiunea de cod din cadrul metodei pentru calculul vitezei aferenta acestui calcul este cea de mai jos:

****

In acest caz, variabila este cea care determina apartenenta in **zona din stanga** a indicatorului.

Cel de-al treilea polinom, va corespunde astfel zonei din dreapta jos a indicatorului, asadar, graficul sau descriind un arc ce va strabate in acelasi timp **zona din dreapta** si **zona de jos** din **Fig3.** Variabila predictoare este si in acest caz tot . Setul de date corespunzator acestei zone este cel de mai jos, precum si graficul aferent unei regresii polinomiale de ordin 3.

|  |  |
| --- | --- |
| y | v |
| 112 | 90 |
| 128 | 95 |
| 145 | 100 |
| 167 | 105 |
| 192 | 110 |
| 218 | 115 |
| 244 | 120 |
| 268 | 125 |
| 293 | 130 |
| 316 | 135 |
| 334 | 140 |

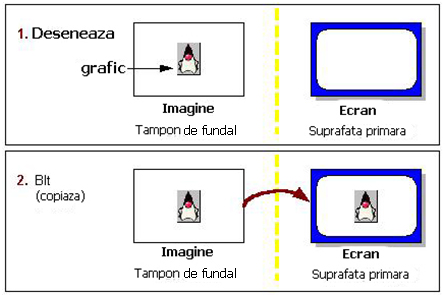
**Folosirea tehnicii Double Buffering**

Aceasta tehnica este folosita pentru desenarea componentelor grafice din cadrul aplicatiei, mai exact desenarea acelor pentru cele 3 indicatoare (viteza, turtie si combustibil). Desi in constructia interfetei sunt folosite componente Swing din cadrul platformei Netbeans, cum ar fi *panel* sau *frame*, care implicit au implementata aceasta tehnica, pentru a evita suparatorul efect de clipire a imaginii la updatare, double-buffering-ul trebuie facut manual.

Sa presupunem ca ar trebui sa desenam o imagine intreaga pe ecran, pixel xu pixel, sau linie cu linie. Daca ar trebui sa desenam asa ceva direct pe ecran, (folosind, spre exemplu, Graphics.drawline), s-ar observa, cel mai probabil cu mare dezamagire, ca acest lucru ar dura destul de mult timp. De asemenea, exista posibilitatea de a observa aparitia a diverse artefacte nedorite o data cu desenarea imaginii. Pentru a evita acest lucru, majoritatea programatorilor folosesc asa numita tehnica *double-buffering*.

Notiunea traditionala de double-buffering din aplicatiile Java este destul de direct explicata asa cum se poate observa in continuare: creaza o imagine „off-screen” (in spatele ecranului), desenarea pe acea imagine folosind obiectul grafic al imaginii, apoi, intr-un pas, apelarea metodei *drawImage* folosing obiectul grafic al imaginii principale, si imaginea off-screen. Asa cum am mai mentionat, Swing foloseste aceasta tehnica in majoritatea din componentele sale, proprietate setata ca adevarata in mod implicit, prin folosirea metodei *setDoubleBuffered.*

Suprafata ecranului este referita de obicei ca „*suprafata principala*” , iar imaginea din spatele acesteia, asa numita *off-screen image* , folosita pentru double-buffering, este referita ca spatiul tampon din fundal („*back buffer*”) . Actinea de copiere a continutului uneia dintre suprafete catre cealalta suprafata, este referita frecvent ca transfer bloc linie („*block line transfer*”), sau „*blitting*”.

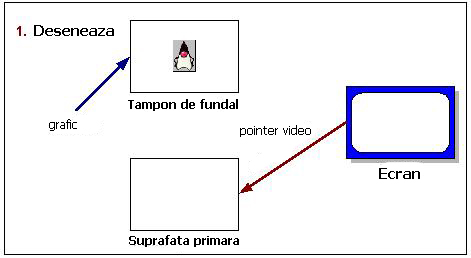


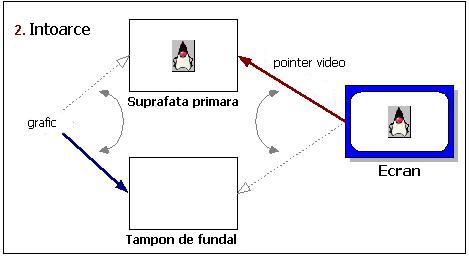
Suprafata principala este de obicei manipulata prin intermediul obiectului grafic al oricarei componente ce este afisata; atunci cand modul *full-screen* este folosit, orice operatie ce foloseste obiectul grafic al imaginii full-screen, reprezinta o manipulare directa a memoriei ecranului. Datorita acestui fapt, se pot folosi in avantajul programatorului si alte capabilitati ale modului full-screen excluziv, ce altfel nu ar fi disponibile datorita suprapunerii sistemului de ferestre. O astfel de tehnica ce este disponibila exclusiv in mod full-screen, este o forma de double-buffering, numita „*page-flipping*” (intoarcerea paginilor).

**Page flipping** (Intoarcerea paginilor)

Multe placi video dispun de notiunea de *pointer video*, ce reprezinta o simpla adresa la memoria video. Acest pointer „spune” placii video unde sa caute continutul video ce urmeaza sa fie afisat la urmatorul ciclu de refresh. In anumite placi video, si in anumite sisteme de operare, acest pointer poate fi chiar manipulat prin programare. Sa presupunem ca ati creat tampon din fundal (back buffer), in memoria video, avand exact latimea, inaltimea si lungimea in biti a ecranului, si desenarea in acel tampon, in aceeasi maniera ca si cand s-ar folosi tehnica double-buffering. Acum, sa ne imaginam ce s-ar intampla daca in loc de a copia imaginea pe ecran ca in double-buffering, s-ar schimba pointerul video catre tamponul referit.Astfel, in timpul ciclului de refresh urmator, placa video va folosi imaginea noastra spre afisare. Aceasta schimbare se numeste „intoarcerea paginilor”, sau *page-flipping*, iar avantajul de performanta fata de simpla copiere din tehnica double-buffering, este ca un singur pointer trebuie mutat in memorie spre deosebire de copierea intregului continut dintr-un tampon in altul.

Cand o intoarcere de pagina are loc, pointerul catre vechiul tampon din fundal, face legatura acum catre suprafata primara, iar pointerul catre vechea suprafata primara , adreseaza acum catre memoria tampon de fundal. Acest lucru pregateste automat urmatoarea operatie de desenare.



****

Cateodata este mai avantajoasa folosirea mai multor tampoane de fundal, intr-un asa numit *lant de intoarceri*. Acest lucru este folositor in mod particular atunci cand timpul folosit pentru a desena este mai mare decat rata de refresh a monitorului. Un lant de intoarcere reprezinta pur si simplu doua sau mai multe tampoane de fundal (cateodata numite tampoane intermediare), plus suprafata principala (aceasta este uneori numita buffering-triplu, buffering-cvadruplu, etc.). Intr-un lant de intoarceri, urmatorul tampon de fundal disponibil devine suprafata primara etc., chiar pana la folosirea ultimului tampon de fundal pentru desenare.

**Avantajele folosirii Double-Buffering si Page-Flipping**

Daca metrica referitoare la performanta este redusa la viteza la care double-buffering si page-flipping ruleaza fata de randarea normala, s-ar putea sa ne lovim de un sentiment de dezamagire. Vom descoperi ca cifrele pentru randarea directa sunt mult mai bune fata de cele ale tehnicii de double –buffering, iar acestea sunt la randul lor mult mai bune fata de cazul folosirii tehnicii page-flipping. Fiecare dintre aceste tehnici este folosita pentru imbunatatirea *perceptiei* performantei, ce este mult mai importanta in aplicatiile grafice decat performanta cifrelor.

Double-buffering este folosita in principal pentru a elimina dezavantaje vizibile ce pot face o aplicatie sa treaca precum una facuta de un amator, in graba, sau pur si simplu sa clipeasca la rulare. Page-flipping este folosita in principal pentru a elimina *ruperea*, un efect de despartire ce apare atunci cand desenarea pe ecran este mai rapida de rata de refresh a monitorului. Desene mai ingrijite presupun o perceptie a performantei mult mai buna, si de asemenea o experienta a utilizatorului superioara.

**Scenarii de testare a aplicatiei**

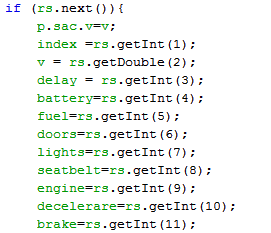
Scenariile de testare a aplicatiei se impart in 3 categorii principale:

* Scenarii pentru testarea cvasicompleta a functionalitatii (scenarii complexe)
* Scenarii pentru testarea comportarii indicatoarelor si a semnalelor de avertizare
* Scenarii pentru testarea comportarii indicatoarelor

**Scenarii complexe**

In cadrul aplicatiei sunt dezvoltate doua scenarii ce testeaza functionalitatea cvasicompleta a computerului de bord virtual, denumite generic „**Scenario1**” si „**Scenario2**”. Fiecaruia dintre ele ii corespunde o clasa separata in program, „**scenario1**” respectiv „**scenario2**”, ce asigura modificarea dinamica a interfetei aferente scenariului in cauza, din timpul rularii programului, astfel, cele doua clase fiind incluse in componenta UI (User Interface), de interfata cu utilizatorul a aplicatiei. Functionalitatea din spatele celor doua scenarii nu este implementata insa in aceste doua clase, ea regasindu-se in clasa „**topClass**”, si fiind explicata in cele ce urmeaza.

In spatele tuturor scenariilor, nunumai a celor complexe, exista cate un tabel corespunzator, parte a bazei de date relationale din cadrul aplicatiei. Valorile fiecarei inregistrari a tabelului sunt preluate in cate o variabila in cadrul unei metode a clasei „**topClass**”, fiind folosita ulterior pentru a dicta comportamentul computerului de bord virtual , asa cum se poate observa in portiunea de cod de mai jos:



Asa cum se poate observa, fiecare inregistrare a tabelului corespunzator unui scenariu ofera 11 valori, corespunzand indexului inregistrarii, vitezei, intarzierii (timpul de asteptare al starii respective), 6 valori booleene aferente activarii/dezactivarii semnalelor de atentionare, si inca 2 valori booleene ce vor semnala inceperea unei decelerari sau a unei franari.

Aceste 11 valori sunt utilizate apoi in cadrul a doua metode, cum a mai fost mentionat inainte, ducand la modificarea dinamica indirecta a simulatorului. Aceste doua metode in cauza sunt „**actualizariScenariu()**” si „**calculCoordonateScenariu()**”.

Testarea functionalitatii cvasicomplete a celor doua scenarii se traduce atat prin testarea simularii a unor semnale de avertizare, cat si a comportarii dinamice a celor 3 indicatoare.

Spre exemplu, primul scenariu complex are urmatorii pasi:

1. Porneste motorul si asteapta 5 secunde
2. Declanseaza semnalul de atentionare pentru usi deschise(semnalul va ramane activ timp de 10 secunde apoi va fi dezactivat);
3. Accelereaza pana la viteza de 50 M/h;
4. Asteapta la viteza de 50M/h timp de 10 sec;
5. Accelereaza pana la viteza de 70 M/h ;
6. Asteapta la viteza de 70M/h timp de 5 secunde;
7. Decelereaza pana la viteza de 20M/h;
8. Declanseaza semnalul de atentionare pentru nivel scazut de combustibil(semnalul va ramane activ timp de 10 secunde apoi va fi dezactivat);
9. Opreste motorul;
10. La 0M/h asteapta 5 secunde.
11. Declanseaza semnalul de atentionare pentru lumini aprinse(semnalul va ramane activ timp de 10 secunde apoi va fi dezactivat);

Incarcarea acestuia este realizata in timpul rularii din cadrul barei de meniu principale, avand ca si conditie necesara, activarea tot din cadrul barei de meniu a optiunii de oprire a controlului manual.

Cel de-al doilea scenariu complex are urmatorii pasi, modalitatea de incarcare a acestuia fiind similara cu cea descrisa anterior:

1. Porneste motorul si asteapta 10 secunde;
2. Declanseaza semnalul de atentionare pentru centura de siguranta (semnalul va ramane activ timp de 10 secunde apoi va fi dezactivat);
3. Accelereaza pana la viteza de 100M/h;
4. Asteapta la viteza de 100M/h timp de 10 secunde;
5. Franeaza pana la viteza de 50M/h;
6. Declanseaza semnalul de avertizare pentru nivel scazut al bateriei (semnalul va ramane activ timp de 10 secunde apoi va fi dezactivat);
7. Decelereaza pana la viteza de 0 M/h;
8. La 0M/h asteapta 5 secunde.
9. Opreste motorul .

Scenarii pentru testarea comportarii indicatoarelor si a semnalelor de avertizare

Scenariu 1**(Medium Functionality Test Scenario 1)**:

1. Porneste motorul si asteapta 5 secunde
2. Declanseaza semnalul de atentionare pentru usi deschise (ramane activ 10 secunde)
3. Opreste motorul

Scenariu 2**(Medium Functionality Test Scenario 2)**:

1. Porneste motorul si asteapta 10 secunde
2. Declanseaza semnaul de atentionare pentru centura de siguranta
3. Opreste motorul

Scenariu 3**(Medium Functionality Test Scenario 3)**:

1. Porneste motorul si asteapta 5 secunde
2. Accelereaza pana la viteza de 50M/h
3. Declanseaza semnalul de atentionare pentru nivel scazut de combustibil (ramane activ 10 secunde)
4. Opreste motorul

**Scenariu 4 (Medium Functionality Test Scenario 4)**:

1. Porneste motoul si asteapta 5 secunde
2. Declanseaza semnalul de atentionare pentru nivel scazut al bateriei (ramane activ 10 secunde)
3. Opreste motorul

**Scenariu 5** **(Medium Functionality Test Scenario 5)**:

1. Porneste motorul
2. Accelereaza pana la 100M/h
3. Franeaza pana la 0M/h
4. Opreste motorul
5. Declanseaza semnalul de atentionare pentru lumini aprinse (ramane activ 10 secunde)

**Scenariu 6 (Medium Functionality Test Scenario 6):**

1. Porneste motorul si asteapta 5 secunde
2. Accelereaza pana la 140M/h
3. Franeaza pana la 100M/h
4. Asteapta la viteza de 100M/h timp de 5 secunde
5. Declanseaza semnalul de atentionare pentru nivel scazut de combustibil(ramane activ 10 secunde)
6. Decelereaza pana la 30M/h
7. Opreste motorul

Scenarii pentru testarea comportarii indicatoarelor

**Scenariu 1 (Indicator Test Scenario 1):**

1. Porneste motorul si asteapta 10 secunde
2. Opreste motorul

**Scenariu 2** **(Indicator Test Scenario 2):**

1. Porneste motorul si asteapta 5 secunde
2. Accelereaza pana la viteza de 60 M/h
3. Asteapta la viteza de 60M/h 5 secunde
4. Opreste motorul

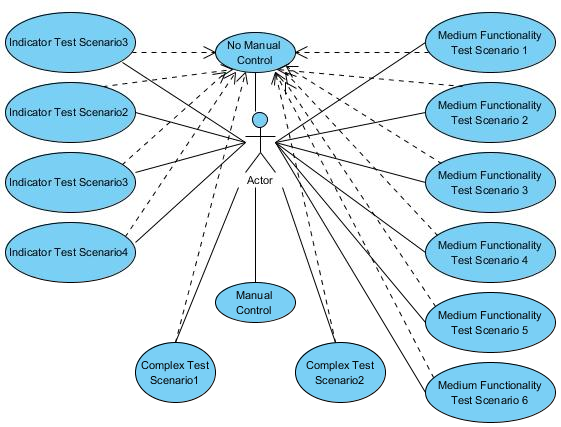
**Scenariu 3** **(Indicator Test Scenario 3):**

1. Porneste motorul
2. Accelereaza pana la 60M/h
3. Franeaza pana la 30M/h unde asteapta 5 secunde
4. Opreste motorul

**Scenariu 4** **(Indicator Test Scenario 4):**

1. Porneste motorul
2. Accelereaza pana la 100M/h unde astapta 10 secunde
3. Decelereaza pana la 10M/h
4. Opreste motorul

**Diagrama de cazuri de utilizare**

****

**Diagrama de clase**

****

****

****

****

****

****

**Bibliografie**

* **Wikipedia**
* **An Embedded Software Primer by David E. Simon**
* **Embedded Systems and Software Validation by Abhik Roychoudhury**
* **The Definitive Guide to Java Swing by John Zukowsky**
* **The Definitive guide to Netbeans Platform by Heiko Bock**