**Cuprins :**

1. **Introducere**
   1. Cuprins
   2. Obiective
   3. Abrevieri
2. **Baze teoretice**
   1. General
   2. Procesul de dezvoltare software (SDP)
      1. Introducere
      2. Descrierea procesul de dezvoltare software
   3. Tipuri de diagrame folosite in domeniul IT
3. **Definirea problemei**
   1. Motivatie
   2. Specificatii si constrangeri
4. **Descrierea aplicatiei**
   1. Conceptul de decuplare
   2. Nivele functionale si arhitectura
      1. Interfata Grafica
      2. Date tranzactionale
      3. Nivelul de persistenta
      4. Mecanismul de extensie
5. **Puncte de extensie**
   1. Arhitectura
   2. Exemplu punct de extensie
6. **Concluzii**
7. **Bibliografie**

Abrevieri (note)

SDP- software design process

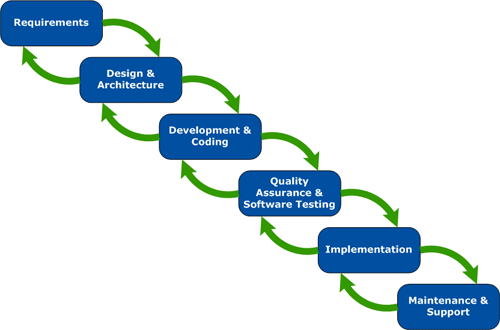
STUB – mock classes

CBSE- component based software engineering

**2.0 Baze teoretice**

2.1 General :

In aces capitol vor fi tratate conceptele de baza pe care se bazeaza aceasta lucrare.Aici nu vor fi descrise notiuni tehnice ci doar o prezentare generala a modelului de dezvoltare software , in continuare precizat ca si SDP (software design process).Acest capitol se adreseaza in general managerilor de proiect si arhitectilor software.



2.2 Modelul de dezvoltare software

2.2.1 Introducere

O persoană perfect raţională este acea persoana care are mereu un motiv bun pentru ceea ce face. Fiecare pas poate fi dovedit a fi cel mai bun mod de a ajunge la un scop bine definit. Cei mai multi dintre noi pot fi considerati profesionisti cu un mod de gandire rational. Cu toate acestea de obicei, pentru mulţi observatori, procesul de proiectare a software-ului apare destul de iraţional.Programatorii începe fără o declaraţie clară a comportamentului dorit şi constrangerile functionale si nefuntionale ale aplicatiei. Aceasta duce la o secventa luna de decizii de dezign cu nici o de declaratie si motivatie clara a acelor decizii.

Aceasta este o abordare clar eronata care poate duce la erori ce necesita bugete enorme pentru a fi inlaturate in stadiile finale ale proiectului. Acesta este motivul pentru care area de cercetare în proiectare de software, metode de programare, programare structurată este una dintre ariile cu evolutia cea mai rapida in ultima perioada . Toate metodologiile care exista actual sunt rezultat al dorintei noastre de a rationaliza sistematic procesul de proiectare software.

Din pacate nu exita un mod de proiectare software perfect si doar abordari mai reusite sau mai putin reusite , in continuare se va prezenta un mod de proiectare rational cu avatajele si dezavantajele aferete.Unul dintre marele avantaje al acestui mod de lucru fiind reducerea drastica a costurilor de dezvoltare si mentenanta pe parcursul ciclului de viata a unui proces software.

Un proces de design software nu poate fi perfect rational din cateva motive mai mult sau mai putin evidente.

1. In multe cazuri cei responsabili de construirea unui sistem informatic nu stiu exact cerintele, si frecvent nu livreaza toata informatia pe care o poseda celorlalti.
2. Un alt motiv ar fi : chiar daca se cunosc specificatiile exact exista multi alti factori ce pot influenta procesul de desing software.Multe detalii devin cunoscutea doar in timpul dezvoltarii , printre care unele nu sunt compatibile cu desingul deja elaborat si necesita modificarea acestuia.
3. Chiar daca toate detaliile ar fi cunoscute in faza incipienta a proiectului, exista studii care demonstreaza incapacitatea creierului uman de a lua in considerare o gama atat de larga de detalii.
4. Toate proiectele software sunt supuse schimbarilor din exterior. Schimbari care fie au legatura cu necesitatile clientului, fie cu evolutia tehnologiilor existente.

2.2.2 Descrierea procesului de proiectare si dezvoltare software

In sectiunea urmatoare se va descrie un proces de design rational idealist care sta la baza oricarui proces de proiectare software reusit. Fiecare pas descris este urmat de o descriere detaliata a produsui asociat acelui pas.

**Crearea Documentului de specificatii**

Pentru a fi rationali, in faza incipienta trebuie sa fie foarte clar specificat cum va arata produsul final.Acea informatie trebuie sumarizata si documentata in produsul de lucru cu numele “Document de specificatii”. Finalizarea acestui document inainte de demararea proiectului va permite proiectarea software’ului cu toate specificatiile bine precizate.

1. Necesitate documentului de specifacatii

Clientul is poate sumariza functionalitatile relevante care apoi pot fi revizuite si actualizate de catre acesta.

Se elimina luarea deciziilor accidentale in timpul procesului de proiectarea software. Programatorii ce lucreaza la sistem de obicei nu au o perspectiva de ansamblu aspura acelui sistem.Prezenta unui document de specificatii ii va scuti de nevoia de a hotara ce este mai util pentru client.

Se elimina dublurile si inconsistenta. Fara un document de specificatii multe dintre intrebarile la care raspunde acest document vor fi puse catre programatori si designeri, de obicei raspunsurile fiind diferite, fapt ce duce la inconsistenta.

Un document de specificatii bine intocmin este necesar (dar nu si suficient) pentru a face i estimare buna a volumui de lucru ce necesita a fi executat si de resursele necesare pentru construirea acestuia.

Documentul de specificatii prezinta o baza buna pentru dezvoltarea de teste automate. Fara acesta departamentul de testare nu ar sti ce functionalitati necesita testare.

1. Constrangeri ale documentul de specificatii

Definirea documentului ideal de specificatii este simpla : acesta trebuie sa contina totata informatia necesara pentru a scrie un software acceptabil pentru client si nimic mai mult.

Fiecare afirmatie trebuie sa fie independenta de deciziile de implementare.

Documentul trebuie sa fie complet in sens :daca fiecare criteriu din document este indeplinit produsul va acceptat de catre client.

Daca informatiile nu sunt disponibile la crearea documentului, aceste informatii lipsa trebuie explicit specificate.

Produsul trebuie organizat ca si document de referinta, nu o introducere in utilizarea sistemului.Merita precizat faptul ca desi intocmirea unui astfel de document necesita un efort considerabil, o sumarizare este mai usor de citit decat o introducere, ceea ce scuteste timp si efort in fazele urmatoare.

1. Cine scrie documentul de specificatii In cazuri ideale acest document este intocmit de catre client sau reprezentantii acestuia. In realitate clientul rareori poseda cunostintele necesare crearii unui astfel de document. Prin umare dezvoltatorii de software vor produce un document preliminar, care va fi validat de catre client sau reprezentantii acestuia.
2. Cum este organizat documentul Pentru a fi mentenabil produsul va fi separat in cateva module funtionale care sunt descrise mai jos:

Specificatia de sistem : specificatia masinilor pe care va rula sistemul.

Interfete de intrare iesire (IO) pentru definirea modului in care sistemul comunica cu mediul extern.

Specificarea valorilor rezultate in contextul starii interne a sistemului si istoria mediului pe care ruleaza acesta

Constrangeri de timp si acuratete

Modificari ulterioare probabile: un sistem bun este usor modificabil si adaptabil la noi cerinte. Acest paragraf al documentului de specificartii defineste ariile sistemului care sunt cele mai susceptibile schimbarii. Proiectarea unui sistem in care oricare componenta este la fel de usor modificabila este fizic imposibila. Programatorii nu ar trebui sa decida ce schimbari sunt cele mai probabile in sistem

Tratarea exceptiilor si a erorilor. Documentul de specificatii va descrie modul in care sistemul se comporta in cazul aparitiei unui eveniment neprevazut de catre acesta. In marea majoritate a cazurilor aceasta parte este ignorata si decizia in ce priveste modul de tratare a erorilor apartine programatorilor

**Documentarea structrii de module**

Dezvoltarea eficienta a produselor software necesita ca programatorii implicati in proces sa poata lucra independent. Ghidul modulelor defineste responsabilitatile acestora dar nu ofera suficienta informatie pentru implementarea lor. O specificatie de interfata a modulelor trebuie scrisa pentru fiecare, aceasta fiind scrisa de un “senior designer” si este revizuita de implementatori cat si de programatorii ce vor folosi acest modul. Acesta document este structurat formal si ofera o viziune “black box” a fiecarui modul. Acest document contine :

1. O lista de aplicatii invokabile de catre celelalte module (numite aplicatii de acces)
2. Parametrii acestor aplicatii
3. Definirea constrangerilor de timp si acuratete pentru acestea
4. Definirea erorilor si exceptiilor acestora

**Descrierea ierarhiei de utilizare**

Ierarhia de utilizare poate fi descrisa in etapa in care toate modulele si aplicatiile lor sunt cunoscute. Aceasta este convenient structurata ca o matrice binara unde valoare de la pozitia (AB) este adevarata doar daca corectitudinea aplicatiei A depinde de prezenta in sistem a unei aplicatii B implementata. Acest document este util in cazul ca aplicatia va fi lansata in trepte, datorita faptului ca precizeaza dependenta dintre sisteme intr-un mod matematic formal.

**Descrierea structurii interne a modulelor**

Odata ce interfata unui modul a fost specificata, implementarea acesteia nu poate fi finalizata ca si o sarcina independenta. Inainitea implementarii deciziilor cruciale de design acestea vor fi notate in documentul cu denumirea “design’ul inter de modul”. Acest document este conceput pentru a permite reviziurea eficienta a proiectului si pentru a explica echipei de mentenanta intentia si motivatia deciziei.

In unele cazuri modulul va fi divizat in submodule, caz in care pentru fiecare submodul procesul va fi reluat de la pasul anterior. In caz contrar structurile de date interne sunt descrise. Pentru fiecare aplicatie de acces se defineste cum aceasta aplicatie afecteaza datele interne.

Descompunerea in submodule si design’ul acestora este continuata pana la punctul in care fiecare sarcina este suficient de sumara incat ea poate fi preluata de catre un implementator.

**Implementarea efectiva**

Dupa finalizarea design’ului si a documentatiei se poate trece la pasul de scriere efectiva de cod executabil. Datorita faptului ca pasii descrisi mai sus au fost urmati acest proces va decurge fara dificultati cu o viteza ridicata. Codul nu va include comentarii redundante cu documentatia deja existenta, astfel se reduc costurile de implementare si mentenanta a sistemului deoarece redundanta in comentariile de cod cresc probabilitatea ca ca acestea vor fi inconsistente cu documentatia existenta.

**Mentenanta**

Mententanta inseamna reproiectarea si reimplementarea sistemului. Pentru a mentine consistenta codului si a documentatiei in cazul aparitiei unei schimbari aceasta trebuie specificata in documentatie si proprietatile incompatibile cu acea shimbare sterse si din documentatia aferenta si din cod. Cu toate ca la prima vedere costurile pe termen scurt par sporite, pe termen lung aces procedeu va pastra arhitectura sistemului calitativa ceea ce va scadea rata de distrugere a calitatii sistemului.

2.2.3 Tipuri de diagrame folosite in procesul de dezvoltare software

In urmatorul capitol vor fi descrise cele mai des folosite diagrame in procesul de proiecesul de proiectare software. Rolul acestor diagrame este de a prezenta cititorului o viziune de ansamblu asupra unui aspect al aplicatiei (acesta fiind diferit in functie de tipul dediagrama utilizata). Utilitatea acestor diagrame este capacitatea de a oferi informatia in mod grafic, ceea ce le face o sursa de informatii rapida si usor de utilizat.

**Diagrame de activitate**

****

O diagrama de activitate este o reprezentare grafica a fluxului de munca pe pasi indivizibili ce suporta structuri decizionale, iteratii si concurenta de procese. In UML diagramele de activitati diagramele de activitate pot fi folosite pentru a descrie procesele de business sau de operare a unui sistem. O diagrama de activitate ofera o privire de ansamblu asupra fluxului de date din aplicatie si controlul acestora.

Acestea sunt construite dintr-un numar limitat de figuri geometrice conectate orientat. Cele mai importante si des folosite figuri sunt:

1)Dreptunghiul rotunjit - reprezinta activitati

2)Rombul - structuri de decizie

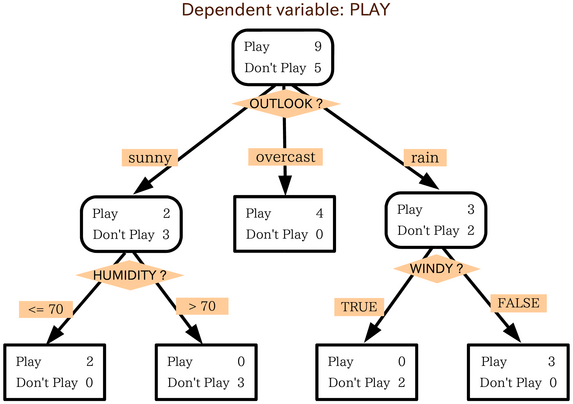
3)Elipsa neagra – starea initiala a sistemului

4)Elipsa neagra iincercuita - starea finala

Sagetile sunt trase de la starea initiala pana la starea finala si reprezinta ordinea in care fiecare activitate este desfasurata. In timp ce in UML 1.x aceste diagrame erau specializate pe reprezentarea starii interne a datelor , odata cu aparitia standardului UML 2.x utilitatea acestora s-a extins.

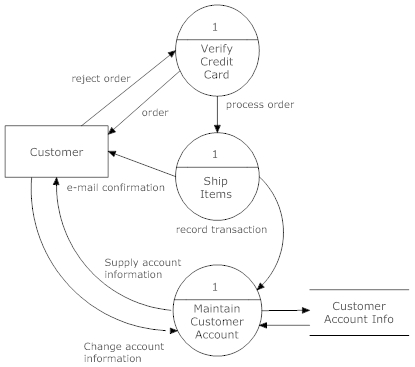
**Arbori de decizie**

Acest tip de diagrame de la aparitia standardului UML 2.x a fost partial inlocuit de diagramele de activitate. Cu toate ca cele din urma ofera o informatie mai vasta decat diagramele de decizie, acestea sunt in continuare folosite in special in procesul de comunicare cu clientul datorita simplitatii lor atat de creare cat si de utilizare. Arborii de decizie sunt folositi in domeniile informaticii ce au legatura cu inteligenta artificiala , un exemplu fiind teoria jocurilor sau data mining.



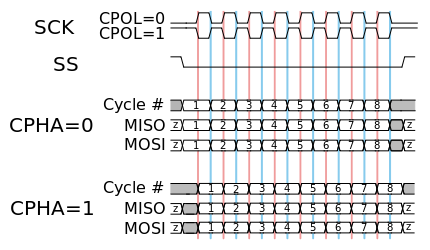
**Diagrame de flux**

Diagramele de flux sunt deseori confundate cu diagramele de activitate, fapt ce se datoreaza responsabilitatilor asemanatoare a acesotora. In timp ce o diagrama de activitate prezinta o secventa de pasi descrisi la nivel inalt ce trebuie urmati pentru a obtine un anume rezultat de nivel inalt, o diagrama de flux prezinta o secventa de pasi la nivel indivizibil. De obicei diagramele de flux sunt folosite pentru descrierea algoritmilor in aplicatie.



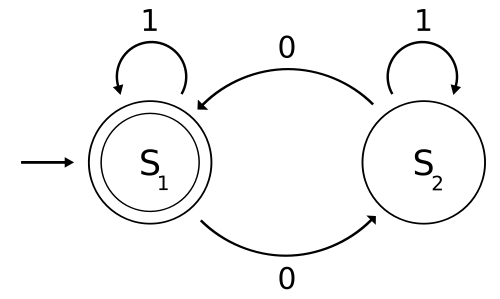
**Diagrame de sincronizare**

O diagrama de sincronizare este o reprezentare a unui set de semnale intr-un spatiu de timp. O astfel de diagrama poate contine mai multe randuri, de obicei unul dintre acestea fiind ceasul intern. Se folosesc pe larg in domeniulul electornicii digitale, depanare hardware si comunicatii. In afara faptului ca acest tip de diagrame ofera o viziune de ansamblu asupra relatiei de sincronizare, ea permite depistarea erorilor logisice posibile. In figura de mai jos se descrie un port serial periferic (SPI). Cele mai importante noduri din acel SPI au capacitatea de setare a polaritatii ceasului si fazei acestuia.



**Diagrame de stare**

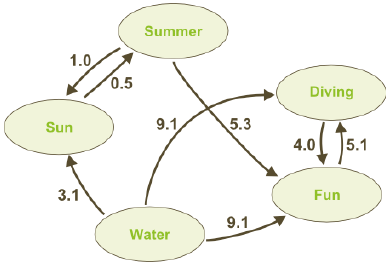
Acest tip de diagrame este folosit in disciplinele ingineresti legate de pedictia de comportamente a unui sistem. Diagramele de stare necesita ca sistemul descris sa fie compus dintr-un numar finit de stari. Formele descrise difere una fata de alta prin semantica lor. Utilitatea principala a diagramelor de stari este sa descrie intr-un mod abstract comportamenul unui sistem. Acest comportament este analizat si preprezentat printr-o serie de evenimente, care decurg in una dintre multiplele stari posibile. In concluzie fiecare diagrama de acest tip reprezeinta o clasa unitara de obiecte in diferitele stari ale sale pe parcursulul executiei sistemului.



**Grafuri orientate si neorientate**

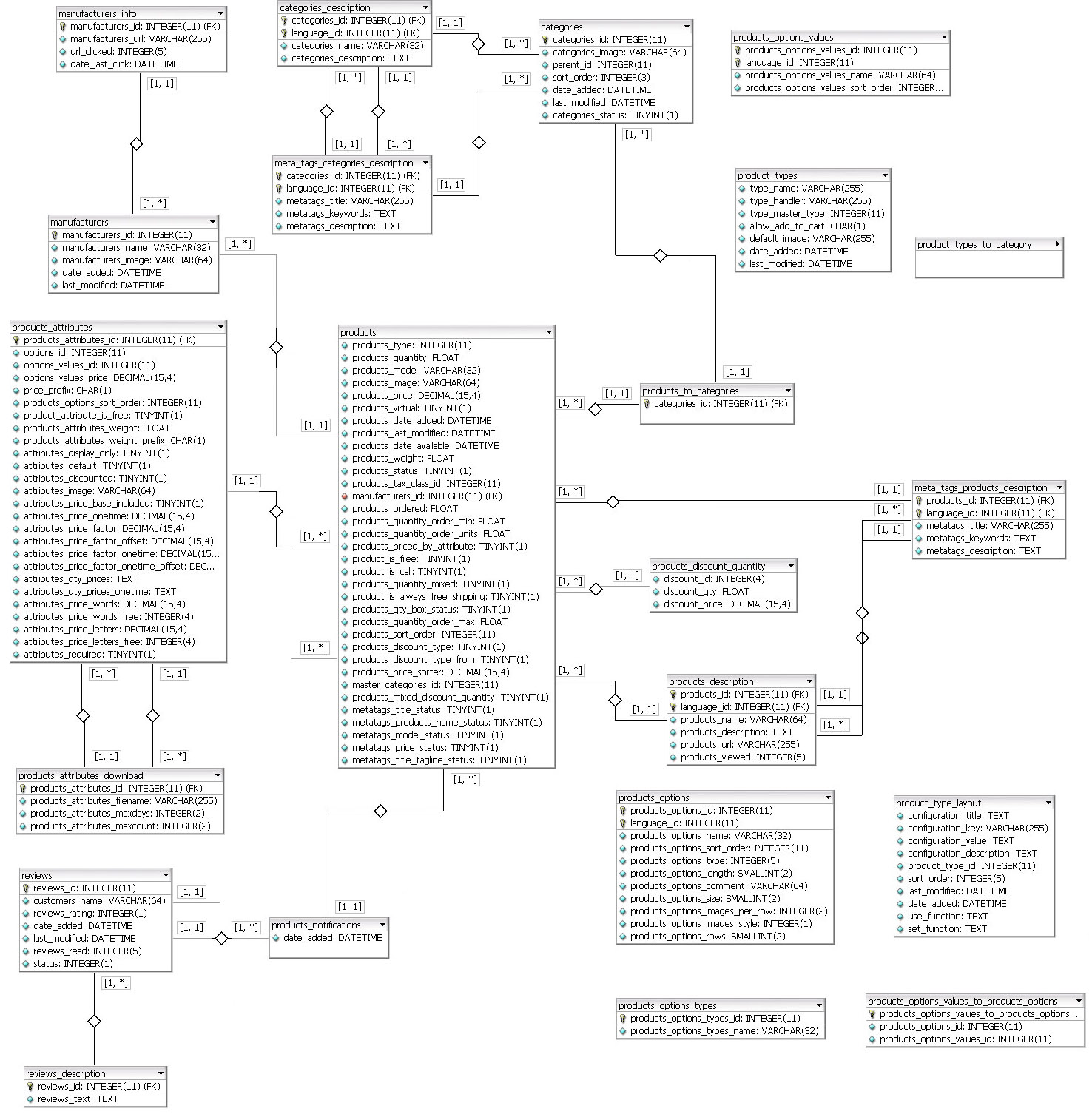
Un graf o pereche ordonată de [mulțimi](http://ro.wikipedia.org/wiki/Mul%C8%9Bime), notată G=(X,U), unde X este o [mulțime](http://ro.wikipedia.org/wiki/Mul%C8%9Bime) finită și nevidă de elemente numite noduri sau vârfuri, iar U este o [mulțime](http://ro.wikipedia.org/wiki/Mul%C8%9Bime) de perechi (ordonate sau neordonate) de elemente din X numite muchii (dacă sunt perechi neordonate) sau arce (dacă sunt perechi ordonate). În primul caz, graful se numește neorientat, altfel acesta este orientat.

Așadar un graf poate fi reprezentat sub forma unei figuri geometrice alcătuite din puncte (care corespund vârfurilor) și din linii drepte sau curbe care unesc aceste puncte (care corespund muchiilor sau arcelor). Grafurile sunt pe larg folosite in domeniul IT in special in docmenii ce tin de comunicatii la distanta, algoritmi de rutare si limbaje formale.

****

**Constrangeri de integritate a bazelor de date**

O schema a unei baze de date este structura acestei baze de date definite intr-un limbaj formal. Aceasta schema reprezinta planificarea modului in care baza de date va fi construita, mai specific modul in care tabelele acesteia vor fi impatite. Definirea formala a unei baze de date consta dintr-un set de formulari numite constrangeri de integritate impuse asupra unei baze de date. Aceste constrangeri de integritate asigura compatibilitatea intre partile schemei, astfel se poate asigura consistenta datelor din modelul schemei.



**3.0 Definirea problemei**

3.1 General :

In capitolul anterior au fost prezentate doar cateva dintre cele mai utilizate tipuri de diagrame folosite in domeniul IT, marea majoritate a diagramelor folosite de catre companiile ce activeaza in domeniu fiind facute dupa un standard intern, adaptat domeniuli in care activeaza acea companie. In plus acest tip de prezentarea a informatiei (schematic) a devenit extrem de popular si in discipline ce nu au legatura cu disciplinele ingineresti cum ar fi in psihologie, actividati didactice sau chiar economie.

Putem identifica nevoia clara pe piata a unui tool ce poate facilita management’ul , crearea si editarea de diagrame. Fara dubii pe piata exista o serie de aplicatii ce indeplinesc intr-o oarecare masura nevoile unui anume segment de piata, acesta fiind si punctul slab al aplicatiilor de acest tip.

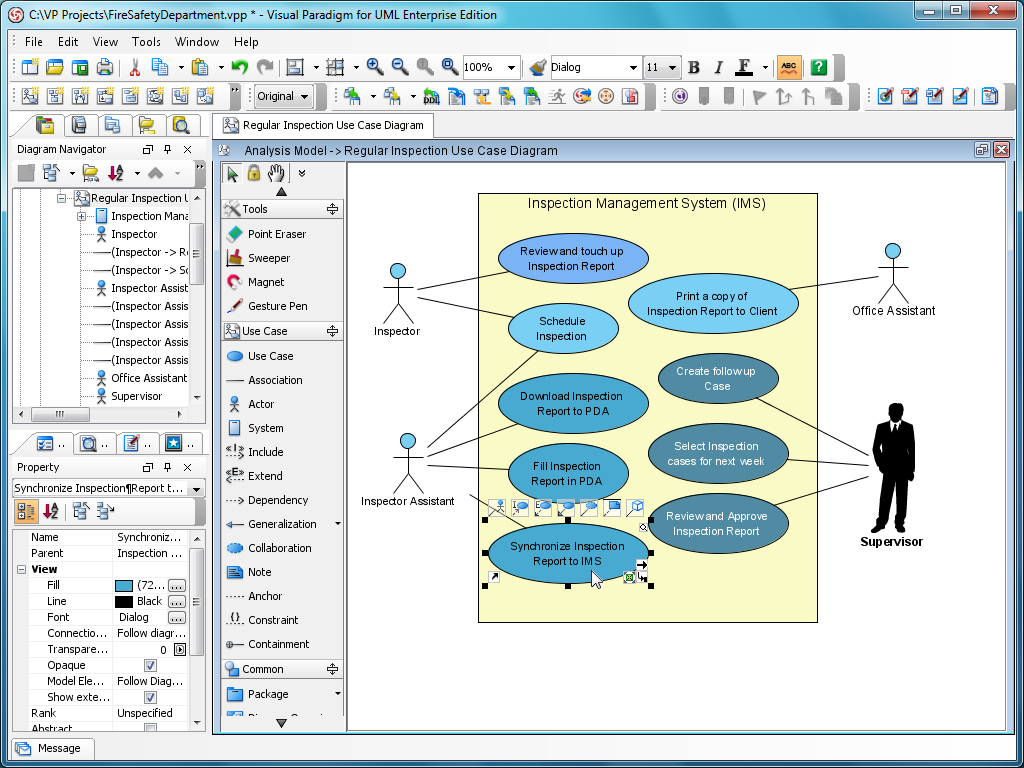
Deseori chiar si aplicatiile destinate desenarii de diagrame pentru un domeniu specific cum ar fi IT’ul sunt foarte rigide si neadaptate la nevoile utilizatorului. Cel mai bun exemplu pe aceasta tema poate fi aplicatia cu numele StarUML. StarUML este o aplicate destinata desenarii de diagrame de tip UML (Unified Modeling Language) de tip open source, datorita faptului ca este gratis si relativ simplu de folosit, este foarte populara in domeniul academic, mai specific folosita de catre elevi si studenti. Cu toate ca aceasta aplicatie este destul de reusita ca si comcept si implementare are o problema majora : neadaptarea la limbaj.

Cu toate ca toate limbajele orientate pe obiect au o structura in ce priveste legaturile dintre clase si tipurile de clase si conexiuni intre acestea relativ asemanatoare, exista diferente care la prima vedere ar parea neimportante si nesemnificative intre 2 limbaje diferite. Cu toate ca standardul UML este global recunoscut se permit mici modificari a unei entitati create pentru a se adapta mai bine la domeniul aplicatiei, ceea ce nu este suportat de marea majoritate a aplicatiilor de desen a diagramelor. Aceste mici modificari, necesare pentru a pastra consistenta intre diagramele de clase (de exemplu) si codul efectiv sunt ignorate.

Pentru a demonstra ideea voi reveni la exemplul anterior. Creatorii StarUML afirma ca aplicatia lor suporta designul diagramelor de clase pentru limbajul Java. Cu toate ca afirmatia este partial adevarata, StarUML nu are specificat un tip de clasa cum ar fi “Clasele absracte”. Astfel o clasa abstracta de obiecei prin conventie va fi prefixata cu <Abstract>. Conventie care de altfel nu exista in generatorul de STUB’uri prezent in aplicatie. In acest mod toate functionalitatile prezente la nivel de generare de cod, sunt inoperabile.

O alta problema prezenta la aplicatiile disponibilie ar fi multitudinea de functionalitati oferite de catre aplicatii. Afirmatia de mai sus poate fi controversata si poate cuntraintuitiva. Pentru a fi cat mai explicit voi da un alt exemplu de aplicatie foarte vast folosita de catre inginerii software: VisualParadigm.

Visual Paradigm este un tool folosit pentru desenarea diagramelor de tip UML avand ca si client tinta ingineri ce activeaza la nivel profesional in domeniul proiectarii software. Faptul ca aceasta aplicatie a primit 5 premii majore in domeniul IT demonstreaza faptul ca este o aplicatie reusita insa datorita faptului ca publicul tinta este al acestei aplicatii sunt proiectantii seniori de sistem se ofera un numar atat de ridicat de funcionalitati incat un utilizator cu un nivel de experienta redus in domeniul proiectarii sistemelor s-ar confrunta cu dificultati majore in identificare functionalitatilor necesare acestuia. Interfata grafica extrem de incarcata fac identificarea functionalitatilor cheie a aplicatiei extrem de dificila, astfel un utilizator neinitiat, care ar vrea o solutie rapida nu isi va putea crea acea solutie fara o pregatire riguroasa in folosirea aplicatiei. Figura adiacenta prezinta o captura de ecran ce contine interfata grafica oferita de VisualParadigm.



Exista exemple numeroase care prezinta avantajele si dezavantajele folosirii aplicatiilor deja existente. In timp ce unele ofera functionalitati avansate dar sunt greu de utilizat, altele ofera funcionalitati usor de utilizat dar mult prea simpliste pentru a crea o diagrama bine definita.

Prin aplicatia prezentata in acest document s-a incercat implementarea unui tool ce va adresa aceste probleme. Printr-o separare clara intre interfata grafica, controlul aplicatiei si functionalitatile oferite de catre aceasta a rezultat o aplicatie ce poate fi scalabila la cerintel utilizatorului independent de domeniul in care activeaza sau de gradul de experienta al acestua.

3.2 Specificatii si constrangeri

In aplicatia prezentata s-a incercat pastrarea avantajelor ale altor aplicatii din domeniu si eliminarea dezavantajelor acestora. Fapt posibil datorita specificatiilor bine definite. In urmatorul capitol se vor descrie specificatiile de nivel inalt ale aplicatei si motivatia in spatele acestora

Un prim considerent major in construirea aplicatiei fost “**Scalabilitatea**” acesteia. Scalabilitatea se refera la comportamentul consistent al aplicatiei in diverse conditii de lucru. Fie cu cerinte de resurse mari, cu un sistem special conceput si configurat pentru utilizarea aplicatei (de exemplu folosirea in paralel a aplicatei in paralel de catre 10 mii de utilizatori, aplicatia fiind disponibila pe un server), fie folosirea aplicatiei pe un sistem personal cu resurse limitate de catre un utilizator. Aplicatia livrata trebuie sa fie suficient de scalabila pentru a rula in parametri normali in ambele cazuri. Scalabilitatea unui sistem software are la baza gama de tehnologii si bibloiteci folosite pentru dezvoltarea acestuia.

Adaptabilitatea redusa a unei aplicatii ii reduce drastci durata acesteia de viata, in special o aplicatie orientata pe client privat, cum este aplicatia prezentata. Astfel putem deduce un alt set de constrangeri de care va trebui tinut cont : “**Adaptabilitatea la limbaj si standarde”**. Cu fiecare schimbare adusa unui limbaj deja existent, aparitie de standard nou in desenarea de diagrame, sau aparitia unui limbaj total nou, aplicatia trebuie actualizata pentru a permite acomodarea acestor schimbari.

O aplicatie ce are un proces de instalare si configurare complicat este in fond o aplicatie cu o intretinere mai dificila. In cazul aplicatiilor foarte dependente de platforma si cu multe fisiere de proprietati, variabile de sistem sau cai de acces se poate ajunge la situatia in care marea majoritate a clientilor fie nu vor instala aplicatia si vor cauta o alternative mai simplu de configurat, fie vor apela la producatorii aplicatiei pentru asistenta, ceea ce va creste costurile de intretinere. Pint concluzie este nevoie ca aplicatia prezentata sa fie **“Usor de configurat”**.

Cu toate ca liderul pe piata in materie de sisteme de operare ramane Microsoft cu produse din seria Windows, exista un segment de piata destul de extins ce folosesc sisteme de operare alternative pentru statiile personale de lucru. Pentru a nu pierde acel segment de utilizatori se propune ca aplicatia sa fie **“Independenta de sistem” .**

Studiile arata ca aplicatiile cu interfata grafica prea incarcata sau ce nu evidentiaza fnctionalitatile cele mai relevante pentru utilizatori isi pierd popularitatea. Pentru a nu intra in acea categorie de aplicatii se propune crearea unei “**Interfete grafice prietenoase si configurabile”**

**2.0 Baze teoretice**

2.1 Conceptul de decuplare

Pentru a obtine un soft extensibil la nivel de functionalitate cum este descris in specificatii (capitolul anterior) a fost decis sa se foloseasca “Ingineria sistemelor bazata pe compobebte” (Componet-based software engineering) , prescurtat CBSE. CBSE este o ramura a ingineriei software care pune accent pe separarea modulelor in legatura cu functionalitatile diversificate puse la dispozitie de un sistem software. Abordarea bazata pe reutilizarea componentelor permite crearea de module independente si reutilizabile. Ca si mod de lucru inginerul software priveste componentele ca entitati cu scopul de a oferi servicii sinestatatoare, nu ca si o componenta a aplicatiei ce va functiona in paralel cu alta cumponenta pentru indeplinirea unei sarcini.

Fiecare component software individual este un pachet soft, modul sau serviciu web care encapsuleaza un set de functionalitati sau date inrudite. Fiecare dintre procesele sistemului esre separat in componente diferite astfel ca toate funtionalitatile inrudite semnatic se regasesc intr-un modul, astfel se ajunge la cohestinea totala a fiecarui modul.

In ce priveste coordonarea globala la nivel de sistem software modulele vor comunica prin interfetele acestora. Cand un component ofera servicii sistemului, acesta adopta o interfata oferita de catre sistemul aferent ce specifica ce servicii si componente modului poate utiliza. Aceasta interfata poate fi vazuta ca si semnatura componentului – clientul nu necesita cunostinte despre functionalitatea interna (implementarea) pentru a folosi serviciile oferite, principiu ce duce incapsularea la nivel de modul, nu doar de clasa (ca si in proiectarea orientata obiect clasica)

O alta caracteristica importanta estea acea ca modulele pot fi substituite unele cu altele chiar si in timpul rularii daca noul component adaugat adopta interfata specificata. Prin consecinta acele componente pot fi inlocuite cu versiuni mai noi ale lor sau cu module alternative fara sa interfereze cu sistemul in care acestea functioneaza.

Refolosirea componentelor este o alta caracteristica al sistemelor proiectate in acest mod. Programatorii trebuie sa creeze o arhitectura si o implementare in modul in care fiecare modul poate fi folosit in diverse proiecte.Pentru a obtine module reutilizabile acestea trebuie sa fie :

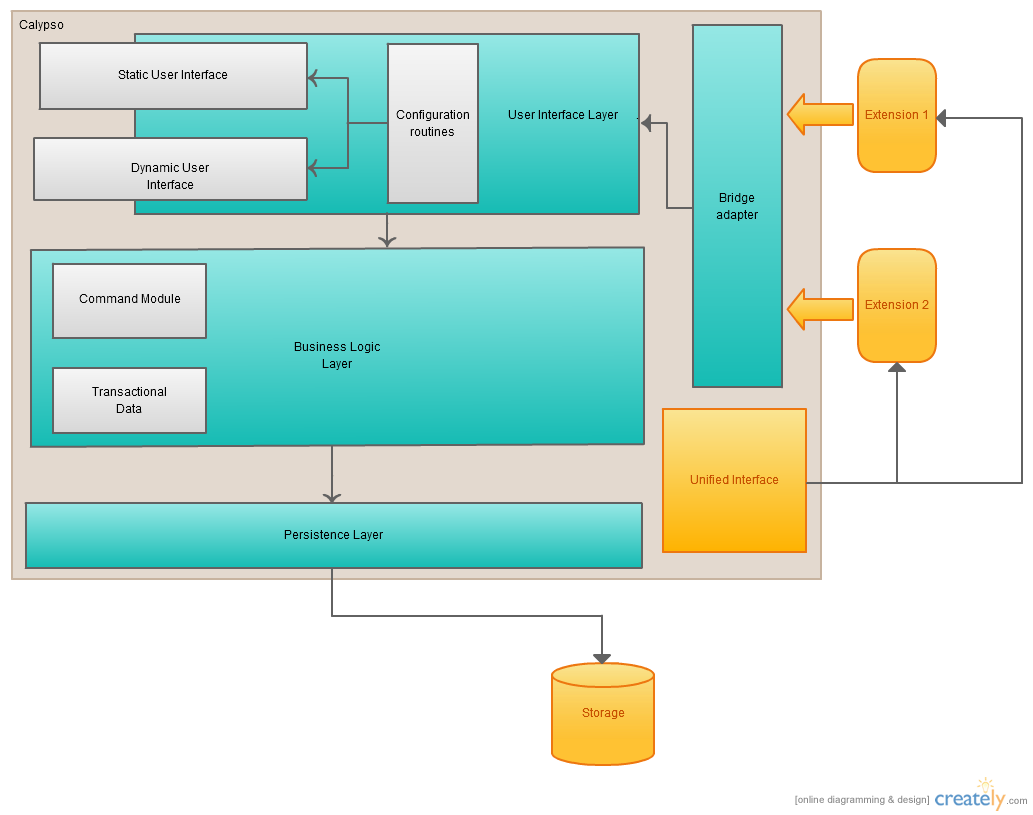
* documentate in intregime
* sa aiba capacitatea de a pasa erori si coduri de eroare modulelor cu care interactioneaza
* poiectate in ideea ca aceste module vor vi intrebuintate deseori in mod eronat.

Cand este necesar ca o componenta sa fie accesata prin contextul executiei sau prin retea, concepte ca si serializarea sau marshelling sunt utilizate pentru a livra componenta la client. Un sistem ce ruleaza mai multe componente este numit server de aplicatie. Folosirea unei combinatie de servere de aplicatie si componente software se numeste “Computatie distribuita”. Aplicabilitatea aceste tehnologii se refera in special la domeniul bancar si aplicatiilor business.

Ca si exemple acest tip de abordare a fost folosit pentru EJB (Enterprise Java Beans) , COM+ model (modelul folosit de platforma .NET) si multe altele.

2.2 Nivele functionale si arhitectura

Aplicatia descrisa foloseste intr-o masura foarte mare abordarea orientata pe modul descrisa mai sus. Pentru a pastra implementarea si design’ul intr-o maniera mentenabila functionalitatile aplicatiei au fost distribuite in 5 subnivele functionale mari , care la randul lor sunt impartite in subnivele. In figura de mai jos est prezentata diagrama de nivele a aplicatiei.



In continuare va fi descris fiecare subnivel functional si componentele din care este alcaturi acesta in detaliu.

**Nivelul interfetei cu utilizatorul**

Acest nivel dupa cum ii sugereaza si numele contine toate clasele ce au legatura cu interfata cu utilizatorul. Scopul acestuia fiind punerea la dispozitie toate functionalitatile aplicatiei luand in considerare un set de constrangeri pentru executarea acestora. Executia unor operatii depinde de contextul de executie al acestora; de exemplu o operatie de “undo” nu poate fi executata daca nu s-a comis nici o operatie asupra documentului curent, la fel cu o operatie de “redo” nu se poate executa daca nu s-a comis “undo” intainte. Nevelul interfetei cu utilizatorul tine cont de aceste constrangeri si are grija sa evidentieze operatiile valide in contextul curent sau sa ascunda operatiile invalide. Aceasta functionalitate este implementata prin inactivarea controalelor. Astfel o operatie invalida va fi semnalata prin modificarea culorii butonului ce o executa si dezactivarea sa la nivel funtional.

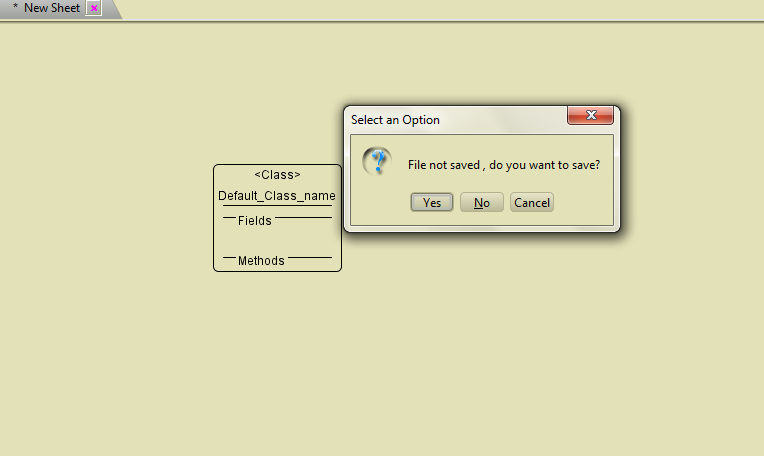
Inactive toolbar



Active toolbar



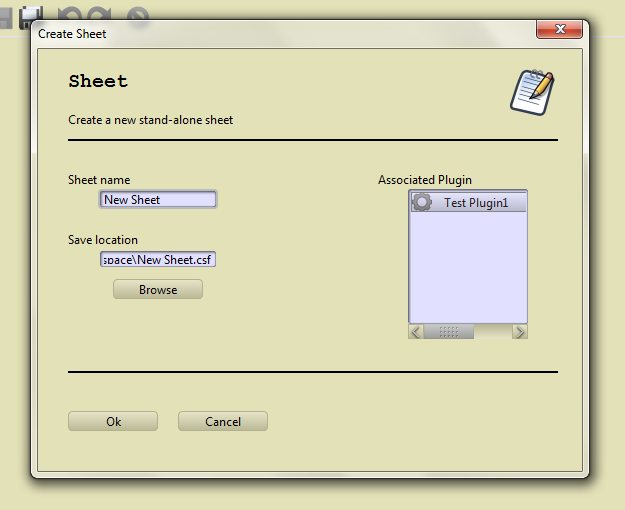
Prin implementarea interfetei grafice in acest mod, se asigura sumplitatea de utilizare (cerinta cheie a aplicatie). Astfel unui utilizator neexperimentat ii este foarte clar ce operatii sunt valide si care sunt valide, fara perturbarea acestuia prin mesaje de eroare. Cu toate ca s-a incercat implemetarea acestui tip de protectie impotriva erorilor in cat mai multe cazuri, exista erori ce nu depind de context , si au ca si cauza parametri oferiti aplicatiei. In aces caz se afiseaza un dialog ce prezinta eroare



Pentru a oferi o experienta intuitiva s-a implementat un alt mod de control al starii intenre a datelor aplicatiei: controlul activ al componentelor. Astfel aplicatia rezultata are 2 module masive de interfata grafica: Modulul static si modulul dinamic. Acestea vor fi descrise in continuare.

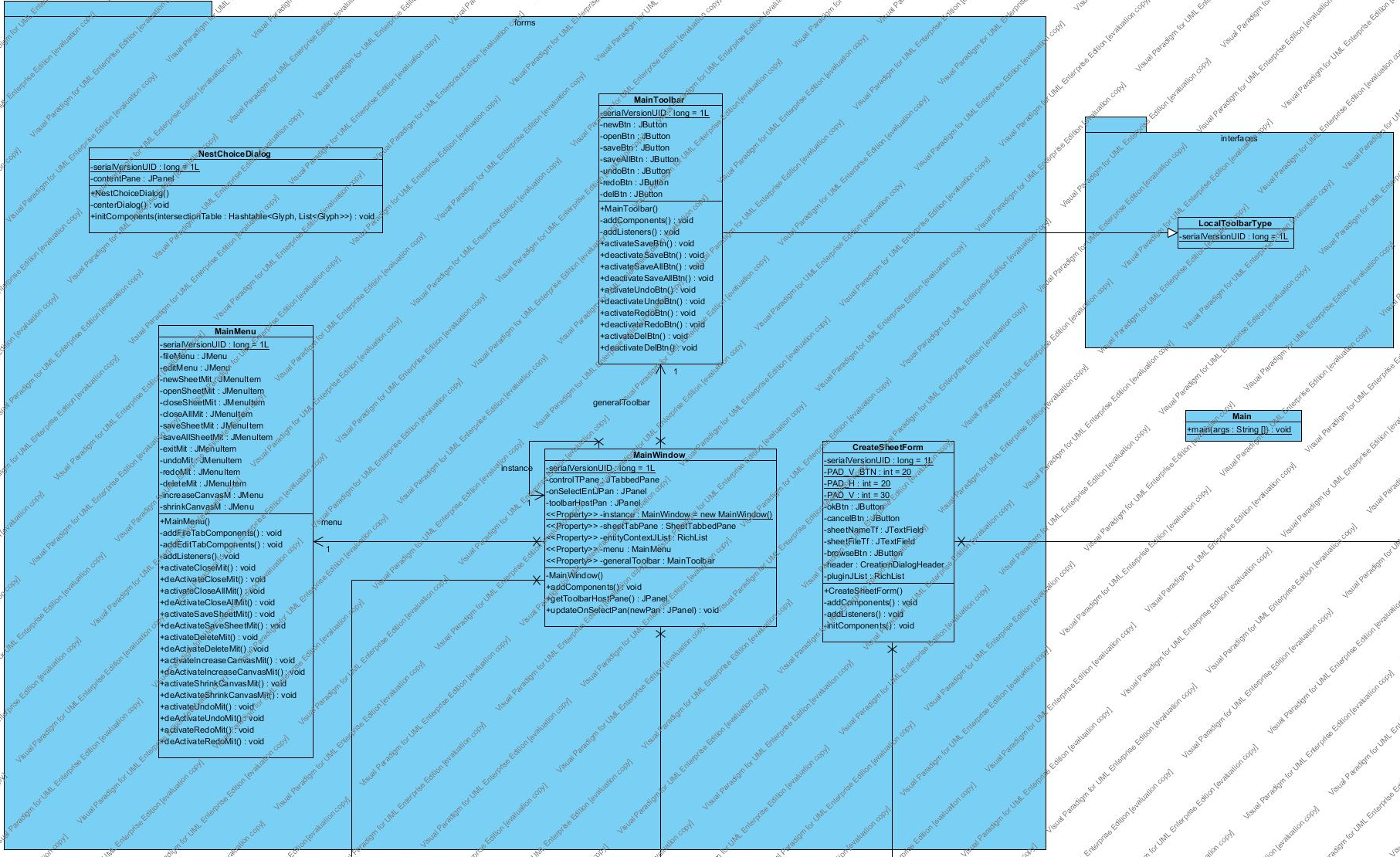
**Modulul static de interfata cu utilizatorul**

Acest modul contine toate ferestrele folosite in aplicatie si reprezinta componentele persistente ale aplicatiei care nu vor varia in funtie de extensiile aplicatiei. In modulul static al interfetei se pot regasi componente vizuale ca meniul aplicatiei, bara de unelte , lista de entitati/conectori disponibilie. De altfel aici se vor regasi toate ferestrele de setare petru un nou document sau proiect.



Pentru a facilita transferul de referinte catre aceste fereste in aplicatie ferestrele de interes vor avea (la nivel de implementare cod) interfete ce le vor defini funtionalitatile, si referintele catre acestea vor fi transmise ca si tipul interfetei folosind sablonul de proiectare MVC (Model View Controller). Astfel se creaza o arhitectura decuplabila, ceea ce duce la posibilitatea schimbarii interfetei grafice statice fara modificari in afara pachetului ce o contine.

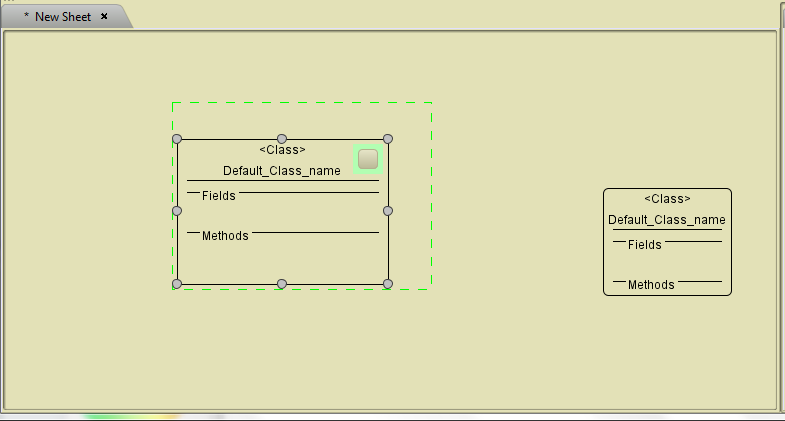
In continuare va fi prezentata diagrama de clase a componentei de control si de afisare pentru acest modul:



**Modulul de interfata dinamica cu utilizatorul**

Modulul dinamic de interfata cu utilizatorul consta din un set de componente interactive puse la dispozitia utilizatorului. In principiu modulul de intefata dimanica cu utilizatorul se refera la editorul de documente.Acest modul ofera diverse forme de componente ce sunt unice proprietatilor de model ale aplicatiei.Pentru a oferi o utilizare placuta si intruitiva a aplicatiei , fiecare component desenat are un set de functionalitati dinamice ale sale.

De exemplu: Pentru a modifica dimnesiunile unei entitati existente utilizatorul prin intermediul unei operatii de “drag-and-drop” ce se executa asupra unui component ce leaga entitatea cu proprietatile sale. Aceste tip de functionalitati ofera utilizatorului ce le invoca feedback in timp real , astfel acesta isi poate da seama care va fi starea obiectului dupa executia operatiei si eventual sa intrerupa operatia in caz ca rezultatul nu este cel asteptat.



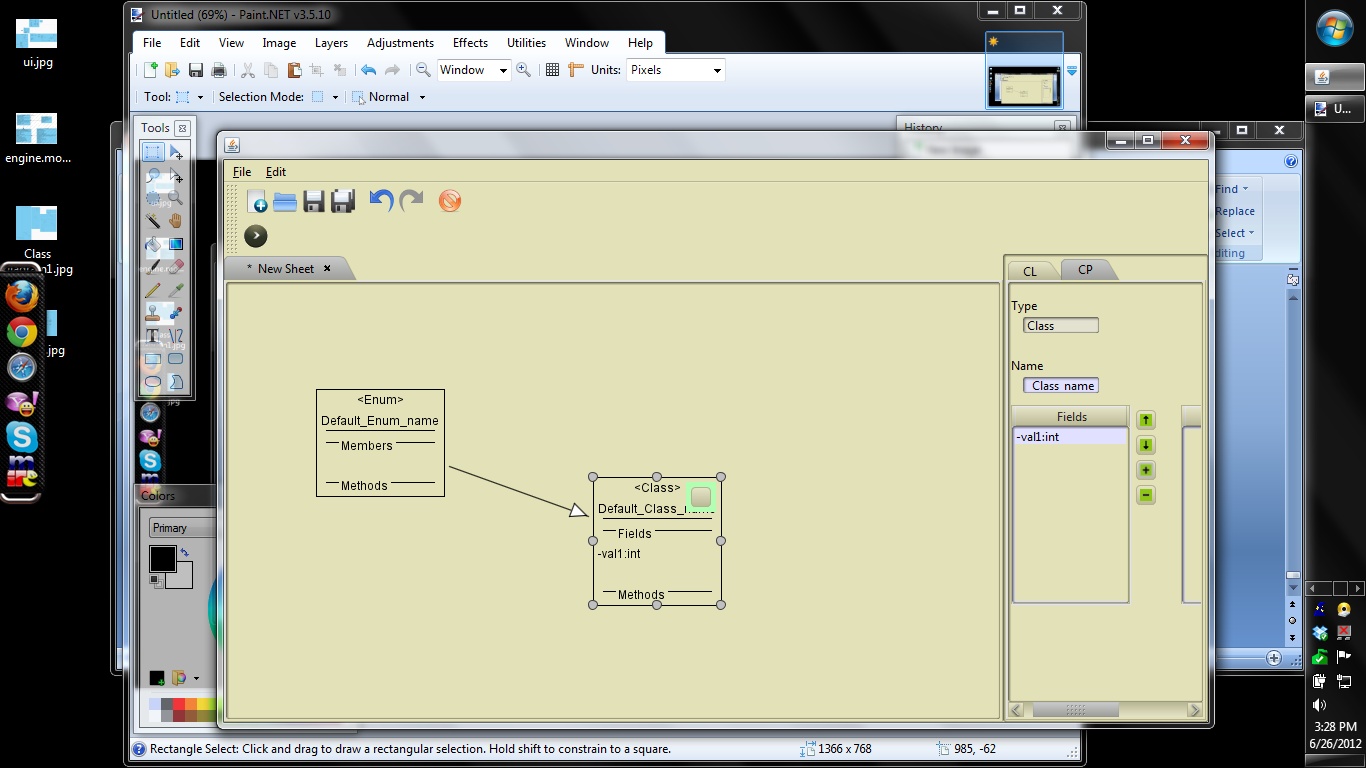


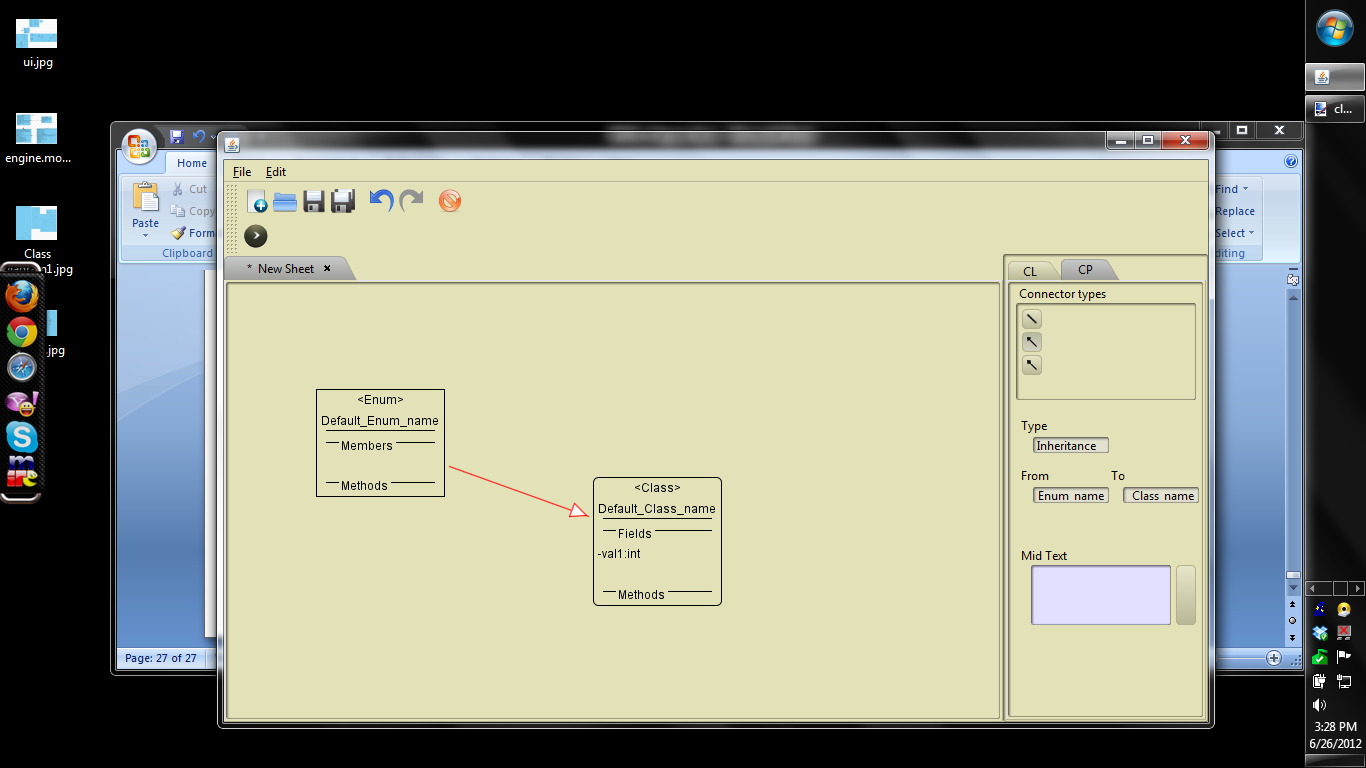
Acest tip de functionalitate este implementat cu ajutorul metodei “Glass pane” si “Double buffering”. Glass pane consta in adaugarea asupra unui container a unui alt container semitransparent ce deseneaza pozitiile auxiliare ale componentelor si datorita transparentei permite visualizarea concomintenta a elementelor asupra carora nu se executa operatii. Acesta este reactualizat la fiecare miscare a mouse-ului.

Datorita randarii intense descrise mai sus poate aparea efectul de “rupere a imaginii” care consta in deteriorarea imagii in cazul in care numarul de frame-uri pe secunda este foarte mare (ca in cazul de fata). Acest efect a fost inlaturat cu ajutorul tehnicii “Double Buffer”. Aceasta tehnica consta in folosirea unui component de tip Graphisc2D auxiliar care va pastra noua imagine, pentru ca apoi sa fie inlocut cu componenta ce se afla pe ecran.

O alta parte a modulului de interfata cu utilizatorul este “panoul de lucru”. Acest panou reprezinta funtionalitatile obiectului (entitatii) desenate si isi modifica continutul operatiile disponibile si aranjamentul in pagina in functie de urmatorii factori :

* ce obiect este selectat
* carui punct de exentensie apartine obiectul selectat
* care sunt datele interne ale obiectului
* ce operatii sunt valide/invalide in contextul curent





**Date tranzactionale**

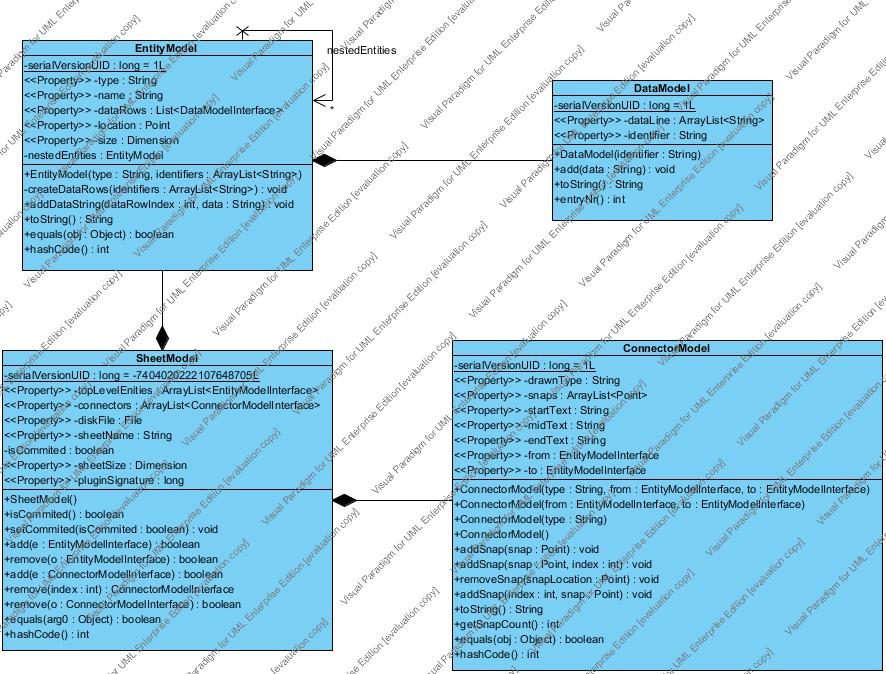
Datele tranzationale ale aplicatiei reprezinta starea interna a obiectelor folosite in aplicatie. Fiecare entitate sau conector ce poate fi desenat cu ajutorul aplicatiei are ca si corespondent un obiect model. Obiectele ce corespund acestui nivel se regasesc in pachetul engine.model si sunt numite in conformitate cu regula : <componenta corespondenta>Model.java.

Aceste date sunt cele mai uzitate de catre aplicatie. Ele vor fi pasate catre nivelul de persistenta pentru utilizarea lor ulterioara, vor fi incarcate de catre acelasi nivel de persistenta pentru a fi modificate si vor fi transferate punctelor de extensie pentru ca sa poata fi modificate de catre acestea in decursul rularii aplicatiei.

Prin concluzie pentru a putea corespunde cerintelor acest nivel a fost supus unei analize riguroase pentru a fi proiectat cat mai independent de altte module si componentele acestuia cat mai independente una de alta. Prin urmare fiecare obiect din nivelul de date tranzactionale va implementa o interfata cu ajutorul careia va fi transmis catre obiectele client.

O functionalitate ce o prezinta aplicatia estea cea de a crea entitati (in cazul punctului de extensie livrat initial cu aplicatia reprezentari de clase) interioare. Astfel reprezentarile grafice ale claselor create vor acomoda in interiorul lor alte reprezentari de clase. Pentru a acomoda acest compoerament a fost crata o clasa abstracta Glyph ce ofera o lista de Glyph’uri interioare. Aceasta clasa este superclasa tuturor obiectelor de model in afara de ConnectorModel. Aceasta clasa nu este prezenta in diagrama de clase de mai jos din cauza faptului ca ea este declara in pachetul de date transferabile catre extensii (descris in urmatoarele capitole).

Pentru a permite extensibilitatea obiectelor de model din aplicatie fara a oferi posibilitatea modificarii structurii acesora proprietatile ce tin de continutul acestor obiecte a fost incapsulat in obiecte de tip DataModel, de consta dintr-o lista de proprietati si o cheie unica pentru identificarea acestora. Astfel din extensiile aplicatiei pote fi adaugat un numar nelimitat de proprietati de continut, fara a necesita accesul catre structura interna a obiectelor asupra caruia se executa operatia.



**Nivelul de persitenta**

Pentru oferi aplicatiei capabilitatea de a salva proiectele deschise pentru editare ulterioara a fost implementat mecanismul de persitenta. Ca si mod de stocare nivelul de persistenta poate fi configurat fie pentru utilizarea cu baze de date neralationale, servere de baze de date relationale locale/la distanta via JDBS, fie prin folosirea tehnologiilor din gama JPA (Java Persistence Api).

In implementarea curenta ca si mediu de stocare au fost alese bazele de date nerelationale (cu toate ca in implementarile anterioare au fost folosite baze de date relationale locale). Acest tip de stocare a fost favorizat datorita urmatoarelor caracteristici:

* portabilitate
* usurinta de utilizare de catre utilizator
* mod de configurare minimalist
* costuri de implementare
* viteza la serializare/deserializare

In cazul folosirii unui alt tip de tehnologie in contextul acestui nivel pot aparea probleme de ordinul scalabilitatii si portabilitatii. In ce priveste utilizatorii ocazionali acestia nu vor avea nevoie de un spatiu de salvare a proiectelor centralizat, insa au nevoie de o aplicatie usor configurabila ce va functiona pe sistemul ce il au la dispozitie. De aceea implementarea standard are la baza fisiere de tip .cly ce vor contine date serializate. Pe de alta parte utilizatorii ce sunt implicati in procesul de dezvoltare software in mediul business pot avea nevoie de un mod de a centraliza datele (pentru a le pastra consistente si actualizate). In cazul acestora nivelul de persistenta este configurabil de catre un administrator de sistem pentru nevoiele lor.

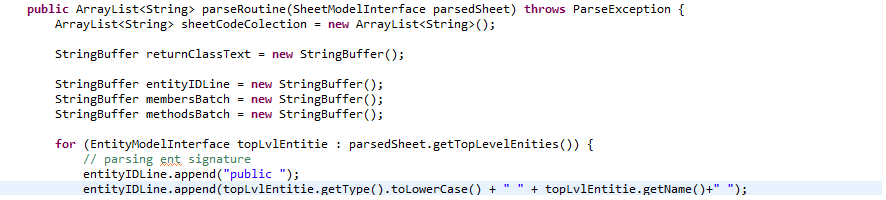
In ce priveste acest nivel, fiecare obiect de model de ordin superior va avea in componenta sa un String care reprezinta locatia acelui component pe disc (ca si cale absoluta). Aceasta cale va fi initializata la creare obiectului si este imutabila. De fiecare data cand documentul este inchis sau salvat datele actualizate vor fi serializate in fisierul corespondent acelui document.

S-a descis folosirea mecanismului de serializare a datelor datoriata criptarii oferite de catre acesta. Astfel fisierele specifice aplicatiei nu vor putea fi modificate din exterior, mod in care se reduce riscul de paraitie a erorilor cauzate de utilizatori.

**Canalul de transfer (Bridge)**

Modulul cu denumirea Brindge, descris in acest capitol are ca si scop transferul de date de la punctele de extensie catre aplicatie. Acesta va fi reponsabil de ciclul de viata a obiectelor initializate in extensii dar folosite in aplicatie. O alta repsonsabilitate a sa este de a injecta functionalitati preulate din extensii in interfata grafica statica a aplicatiei.

De exemplu dezvoltatorul de extensii vrea sa ofere utilizatorului aplicatiei un mod de a-i exporta diagrama creata in mod text (de exemplu cum Java din UML). Pentru a obtine acest comportament el trebuie sa defineasca metoda prin care se genereaza textul

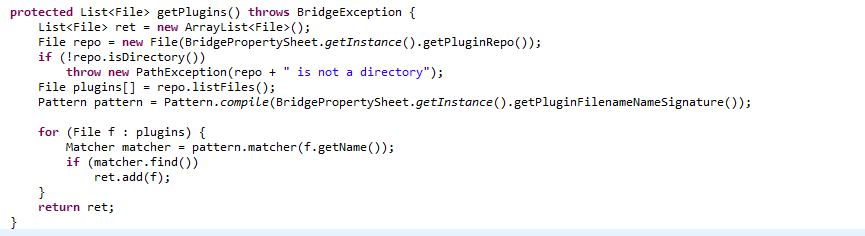


in interiorul punctului de extensie si apoi sa ofere aceasta functionalitate la nivelul interfetei grafice (de exemplu un buton pe panoul de comanda central al aplicatiei).

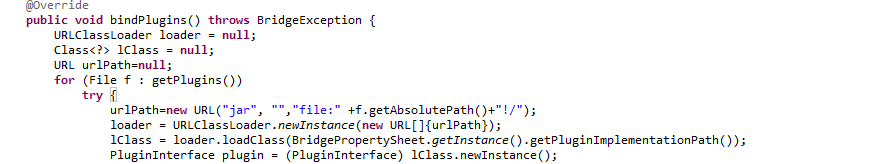
Pentru a-si atinge scopul modului bridge este implementat prin modelul RPC (remote procedure call).RPC consta in comunicarea intre procese ce permite unei subrutine sa execute o procedura din alt spatiu de adrese fara ca programatorul sa fie nevoit sa defineasca explicit modul de comunicare inter-proces.

In aplicatia prezentata functionalitatea RCP a fost implementata cu ajutorul tehnologiei Reflection oferite de catre java. Reflection permite examinarea si modificarea comportamentelor unui obiect in tipul rularii aplicatiei. In cazul concret pentru transferul de date si functionalitati intre aplicatia centrala si puntele de extensie are loc folosind urmatoare secventa de actiuni logice:

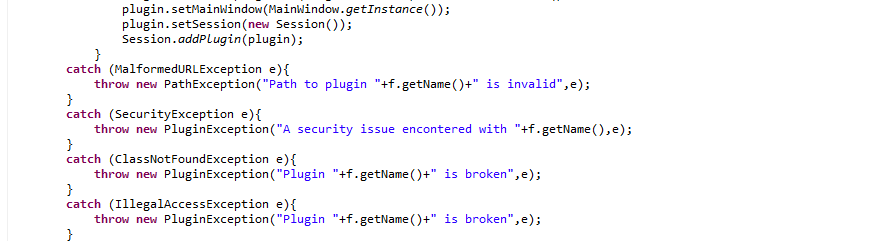
* se importa toate punctele de exensie din folderul pecificat in proprietatile aplicatiei



* se creaza o noua instanta a punctului de intrare in extensie



* se incarca datele interne ale aplicatiei relevante pentru extensie si extensia incarcata isi va modifica statutul in “extensia curenta”



Pentru a putea executa rutinele descrise mai sus atat cat punctele de extensie cat si aplicatia centrala trebuie sa implementeze acelasi protocol de comunicare si o interfata unificata.

Aceasta interfata se poate regasi in sub-pachetul brindge.transferable atat cat in aplicatie cat si in API’ul oferit pentru dezvoltarea de extensii. Toate obiectele ce au instante nepartajata vor fi definite aici in functie de scopul tipului de obiecte.

Pachetul “contexts” reprezinta interfetele claselor carora li se delega functionalitatile specifice extensiilor. De exemplu modul de desenare al unei entitati va fi modificabil din extensie, astfel un obiect de tipul EntityView va delega comportamentul de desenare unui obiect agregat de tipul EntityContext obtinut din punctul de extensie.



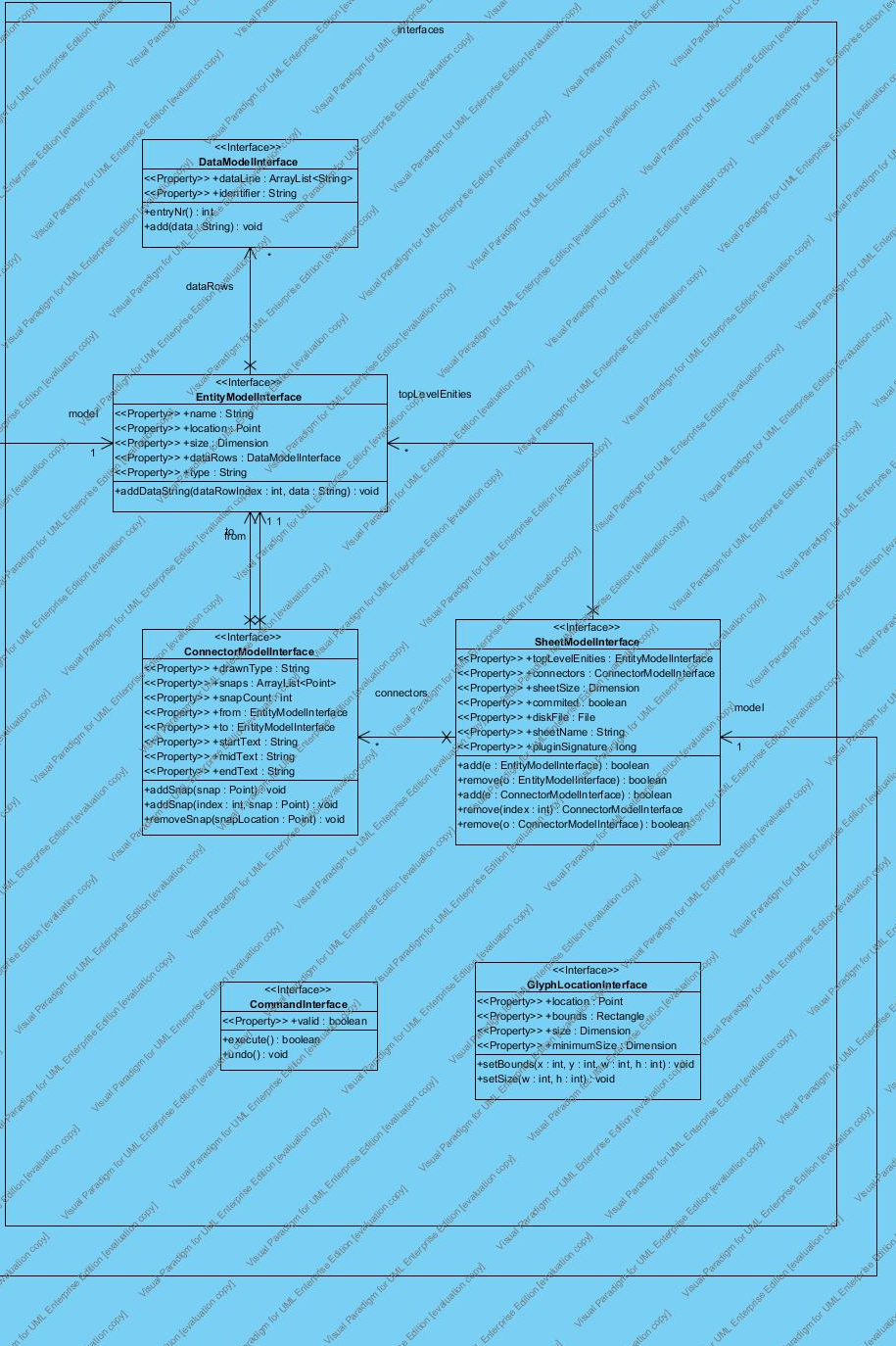
Aceste contexte vor fi distribuite intre aplicatii in modul urmator. Interfata clasei va fi regasita atat cat in aplicatia de baza cat si in extensie, in timp ce implementarea se regaseste pe partea extensiei in exclusivitate.

In timp ce functionalitatea entitatilor din aplicatie este distribuita intre obiectul concret (local) si contextul acestuia (remote) exista mai multe tipuri de obiecte ce necesita transferul integral catre extensie sau vice-versa. Obiectele ce necesita transferul integral prin bridge isi vor avea interfata in subpachetul bridge.transferable.interfaces. De precizat ca nu obiectele concrete sunt localizate aici, ci interfetele aceastora in conformitate cu “dependency inversion principle”. Principiu care reduce cuplajul intre obiecte (in cazul nostru reduce cuplajul intre aplicatie si API-ul extensiei sale).

Un exemplu foarte relevant ar putea fi obiectele de tipul CommandInterface. Pentru crearea segmentului de undo-redo a fost folosit sablonul de proiectare command. In cazul executiei unei operatii se va crea un obiect de tipul CommandInterface care va memora starea membrilor pe care acesta ii modifica inainte si dupa operatie. Astfel obiectul se poate readuce la starea sa initiala prin aplicarea valorilor vechi. Ce relevanta are acest segment pentru modulul bridge:

Cum am afirmat in capitolele precedente dezvoltatorul de extensii are capacitatea de a-si inregistra propriile rutine in aplicatie. Fara acces la segmentul undo acesta nu isi va putea inregistra operatiile, prin concluzie va pierde functionalitatea undo-redo. Pentru a inlatura acest comportament nedorit acestuia i s-a oferit acces catre interfata CommandInterface. Pentru a-si inregistra o comanda ce se initializeaza la apasarea unui buton dezvoltatorul de extensii isi va crea un obiect ce implementeaza aceasta interfata unificata si il va inregistra in obiectul de tip UndoSegmentProxy (tipul de obiecte proxy va fi descris in capitolul urmator).

In general aceeasi functionare o au obiectele de model ce vor fi transmise prn interfata acestora catre extensie. Asfel metodele din extensie vor putea modifica un obiect ce se afla in memory pool-ul aplicatiei centrale prin interfata sa fara a avea dependinta concreta de tip, ci doar de interfata. In general obiectele ce sunt transmise integral sunt obiecte ce apartin modelului aplicatiei (transactional data layer) si sunt parte a ierarhiei de dependinte din API-ul oferit pentru dezvoltarea de extensii.



## E:\Dropbox\Workspace\Calypso\DOC\Raw\webpic.gifUniversitatea de Vest din Timișoara

## Facultatea : Matematică – Informatică

## Specializarea : Informatică Romană

## Lucrare de licență

## Titlul lucrarii : Generarea de ‘stuburi’ în limbajul Java

Coordonator: Student:

Victoria Iordan Șuletea Dorin