**Universitatea de Vest din Timișoara**



**Facultatea de Matematică și Informatică**

**Specializarea Informatică aplicată**

**LUCRARE DE LICENȚĂ**

**GENERAREA DE STUB-URI JAVA**

**Coordonatorul științific, Student,**

**Conf. Dr. Victoria Iordan Dorin Șuletea**

**Timisoara,**

**2012**

**ABSTRACT**

If you did hear about Eclipse you must have heard about it’s Remote Procedure Call architecture, and i am not talking about the IDE, but about the whole Eclipse infrastructure. It is one of it’s most distinct features, that make the whole workbench extremely proficient. This type of architecture consists of many interchangeable modules that can be integrated with each other on compile-time or even runtime. As a result of this feature each developer that works with Eclipse can integrate in his application almost anything that Eclipse can do, may it be a small module that’s related to text coloring for example, either a peer-to-peer communication layer that is build from thousands of classes. Building software in that manner sound like a smart idea.

Another smart and generally used technique is to document properly the code you wrote. And this is when diagrams come in play. Diagrams are an important part of our life, either you are an engineer that is designing an automated warehousing system and you want to design it’s layout, or you are a speaker at a huge conference and you want to create a mind-map to summarize your speech. Especially diagrams are useful when talking about computer science for their ability to store large amounts of data, and are indinspensable when creating software documentation. But there is one issue : these diagrams come in so many shapes, purposes or flavors that there is no way someone will be able to build a tool that can draw all the diagrams you could need in a process of some kind.

This is the part where the two concepts described above will merge. If it is not possible to implement an application that can draw any type of diagram, and there is a way to create software that can accommodate limitless change (like Eclipse) why shouldn’t exist an application that can provide a way for the users to define their own need and integrate it with the existing tool using the same plugin architecture as Eclipse. This is the main purpose of Calypso(the tool tool to be described in the following paper), an application for drawing diagrams from user defined plugins.

**CUPRINS**

1. **Introducere .........................................................................................4**
2. **Baze teoretice.....................................................................................5**

**2.1 Context general........................................................................5**

**2.2 Modelul de dezvoltare software...............................................6**

**2.2.1 Introducere......................................................................6**

**2.2.2 Descrierea procesului.......................................................7**

1. **Specificatiile aplicatiei ......................................................................20**

**3.1 Context general ..........................................................................20**

**3.2 Specificatii și constrângeri...........................................................23**

1. **Descrierea aplicatiei..........................................................................25**

**4.1 Conceptul de decuplare...............................................................25**

**4.2 Nivele functionale și arhitectura aplicatiei...................................27**

**5. Concluzii.................................................................................................45**

**6. Bibliografie.............................................................................................46**

**CAPITOLUL 1. Introducere**

Un sistem distribuit este alcăuit din o serie de mașini anonime ce comunică printr-o retea de calculatoare. Aceste calculatoare comunica unul cu altul pentru a ajunge la un scop comun (efectuarea unei operații). O operației astfel va fi divizată in mai multe sub-operații ce vor rula independent pe mai multe calculatoare, pentru ca rezultatul să fie apoi asamblat pe calculatorul ce a solicitat acea operație. [5]

Functionalitatea descrisa mai sus poate fi obtinuta prin folosirea unei aplicatii RPC (remote procedure call) care permite executarea unei metode aflate pe un calculator aflat la distantă prin folosirea stub-urilor. Un stub este un obiect fictiv aflat pe masina client ce va reprezenta un punct de acces către obiectul concret ce se află în spatiu de memorie a unei mașini aflate la distanță.

Rolul acestor stub-uri este de a mapa adresele parametrilor locali cu adresele parametrilor de la distanta, ce vor avea alte adrese de memorie sau chiar o alta structura de interna în functie de arhitectura sistemului.

În aplicația prezentată s-a încercat implementarea unui sistem distribuit, nu la nivel de mașini diferite ce o execută ci la nivel de distribuire a functionalităților între două sau mai multe aplicatii.

Astfel s-a obtinut un sistem ce își poate schimba functionalitățile la rulare prin importarea acestora din alte aplicații (puncte de extensie) create în conformitate cu standardele impuse de acest sistem.

**CAPITOLUL 2. Baze teoretice**

2.1 Context general

In acest capitol vor fi tratate conceptele de bază ale acestei lucrări și va fi realizată prezentare generală a modelului de dezvoltare software, în continuare referit ca şi SDP (Software Design Process). Acest capitol se adresează in general managerilor de proiect şi arhitecţilor software.

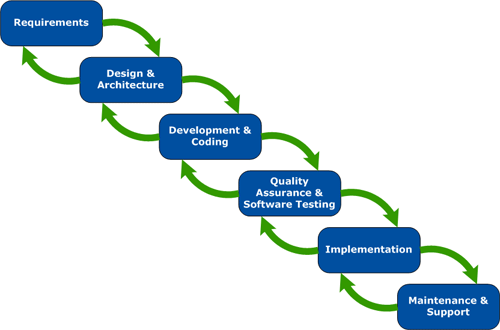


Figura 2.1. Etapele dezvoltării software

2.2 Modelul de dezvoltare software

2.2.1 Introducere

Procesul de dezvoltare software constă dintr-o serie bine definită de pași care odată urmati cu exacticate duc la sporirea vitezei de dezvoltare software si îmbunatatesc calitatea produsului final. Cu toate acestea, de obicei, pentru mulţi observatori, procesul de dezvoltare si proiectare software pare destul de iraţional. Programatorii încep implementarea fără o declaraţie clară a comportamentului dorit sau a constrângerilor functionale sau nefunctionale ale aplicatiei.

Aceasta este o abordare eronată, care poate duce la greșeli ce necesita bugete enorme pentru a fi inlaturate in stadiile finale ale proiectului. Din acest motiv ariile de cercetare în proiectarea software şi metodele de programare, respectiv programare structurată sunt ariile cu evolutia cea mai rapida in ultima perioada. Toate metodologiile care există la ora actuală sunt rezultatul dorintei noastre de a rationaliza sistematic procesul de proiectare software.

Din pacate nu există un mod de proiectare software perfect, ci doar abordari mai reusite sau mai putin reusite. In continuare se va prezenta un mod de proiectare rational, cu avantajele si dezavantajele aferete. Unul dintre marile avantaje al acestui mod de lucru este reducerea drastică a costurilor de dezvoltare si mentenanța pe parcursul ciclului de viată a unui proces software.

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

* In multe cazuri cei responsabili de construirea unui sistem informatic nu stiu exact cerintele din timp, si frecvent nu au cunostințele tehnice să livreze toată informatia pe care o poseda celorlalti.
* Un alt motiv ar fi: chiar daca se cunosc specificatiile exacte exista multi alti factori ce pot influenta procesul de design software. Multe detalii devin cunoscute doar in timpul dezvoltării, printre care multe nu sunt compatibile cu designul deja elaborat si necesita modificarea acestuia.
* Chiar daca toate detaliile ar fi cunoscute in faza incipientă a proiectului, exista studii care demonstrează incapacitatea creierului uman de a lua in considerare o găma atât de larga de detalii.
* Toate proiectele software sunt supuse schimbarilor din exterior. Schimbari care fie au legatura cu necesitațile clientului, fie cu evolutia tehnologiilor existente pe piața.

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 specificații**

Pentru a fi rationali, in faza incipienta a proiectului este necesar sa fie foarte clar specificat cum va arăta produsul final. Acea informatie trebuie sumarizată si documentată in produsul de lucru cu numele “Document de specificații”. Finalizarea acestui document inainte de demararea proiectului va permite proiectarea software-ului având toate specificațiile bine precizate.

1. Necesitate documentului de specificații

Clientul isi poate sumariza functionalitațile relevante care apoi pot fi revizuite si actualizate de catre acesta.

* + - Se elimină luarea deciziilor accidentale in timpul procesului de proiectarea software. Programatorii ce lucrează la sistem de obicei nu au o perspectivă de ansamblu aspura acelui sistem. Prezența unui document de specificații ii va scuti de nevoia de a hotarî ce este mai util pentru client și ce functionalitați nu vor fi compatibile între ele.
    - Se elimina dublurile și inconsistenta. Fara un document de specificatii multe dintre intrebările la care raspunde acest document vor fi puse catre programatori si designeri, de obicei raspunsurile fiind diferite, fapt ce duce la inconsistenta implementarii si a arhitecturii.
    - Un document de specificatii bine intocmit este necesar (dar nu și suficient) pentru a face o estimare buna a volumui de lucru ce necesită a fi executat cât si de resursele necesare pentru construirea acestuia.
    - Documentul de specificații prezinta o baza bună pentru dezvoltarea de teste automate. Fara acesta departamentul de testare nu ar ști ce functionalitați necesita testare.

1. Constrângeri ale documentul de specificații

Definiția unui document ideal de specificații este simplă: acesta trebuie sa contină toată informatia necesară pentru a scrie un software acceptabil pentru client și nimic mai mult.

* Fiecare afirmatie trebuie sa fie independentă de deciziile de implementare.
* Documentul trebuie să fie complet: dacă fiecare criteriu din document este indeplinit produsul va acceptat de către client.
* Dacă informațiile nu sunt disponibile la crearea documentului, aceste informații lipsă trebuie explicit specificate.
* Produsul trebuie organizat ca si document de referintă, nu ca și o introducere in utilizarea sistemului. Merită precizat faptul ca deși intocmirea unui astfel de document necesită un efort considerabil, o sumarizare a cerințelor este mult mai ușor de citit decat o introducere, ceea ce scuteste timp și efort in fazele urmatoare ale ciclului de viată a produsului.

1. Persoane responsabile de întocmirea documentului de specificații

În cazul ideal acest document este întocmit de către client sau reprezentanții acestuia. In realitate clientul rareori posedă cunostintele tehnice necesare pentru a crea un astfel de document. Prin umare dezvoltatorii software vor produce un document preliminar, care va fi validat de către client sau reprezentanții acestuia.

1. Cum este organizat documentul Pentru a fi mentenabil produsul va fi separat in câteva module funcționale care sunt descrise mai jos:

* Specificația sistemului : specificatia mașinilor pe care va rula sistemul.
* Interfețe de intrare ieșire (I/O) pentru definirea modului in care sistemul va comunica cu mediul extern.
* Specificarea valorilor rezultate in contextul starii interne a sistemului si istoria mediului în care ruleaza acesta.
* Constrângeri de timp si acuratețe.
* Modificări ulterioare probabile: un sistem calitativ este usor modificabil și adaptabil noilor cerinte ale pieţei. Acest paragraf al documentului de specificații defineste parțile sistemului care sunt cele mai susceptibile schimbării. Proiectarea unui sistem in care oricare componentă este la fel de ușor modificabilă este fizic imposibila și sporirea extensibilitații și independenței unui modul crește dependințele unui alt modul. Programatorii nu ar trebui sa decidă ce schimbări sunt mai probabile in sistem.
* Tratarea excepțiilor și a erorilor. Documentul de specificații va preciza modul in care sistemul se comportă in cazul aparitiei unui eveniment neprevazut. In marea majoritate a cazurilor această parte este ignorată si deciziile în ce priveste modul de tratare a erorilor revine programatorilor.

**Documentarea structrii de module**

Dezvoltarea eficientă a produselor software necesită ca programatorii implicati in proces sa poata lucra independent. Ghidul modulelor definește responsabilitațile a fiecarui modul dar nu oferă suficientă informatie pentru implementare. O specificatie de interfata a modulelor trebuie scrisa pentru fiecare entitate modular-functională, aceasta fiind scrisă de către un “designer senior” si este revizuită de către implementatori cat si de programatorii ce vor folosi acest modul ca si serviciu. Acesta document este structurat formal si ofera o viziune “black box” a fiecarui modul si contine :

* Lista de aplicații invocabile de către celelalte module (numite aplicații de acces).
* Parametri acestor aplicații.
* Definirea constrângerilor de timp si acuratețe pentru acestea.
* Definirea erorilor si excepțiilor acestora.

**Descrierea ierarhiei de utilizare**

Ierarhia de utilizare poate fi descrisă in etapa in care toate modulele si aplicațiile lor sunt cunoscute. Aceasta este convenient structurată ca o matrice binara unde valoarea de la poziția (AB) este adevarata doar dacă corectitudinea aplicatiei A depinde de prezența in sistem a unei aplicatii B deja implementată. Acest document este util în cazul că aplicația va fi lansata in trepte, datorită faptului ca precizează dependența dintre sisteme intr-un mod matematic formal.

**Descrierea structurii interne a modulelor**

Odata ce interfața unui modul a fost specificată, implementarea acesteia nu poate fi finalizată ca și o sarcina independentă. Inainitea implementării deciziilor de design cruciale acestea vor fi notate in documentul cu denumirea “design-ul intern de modul”. Acest document este conceput pentru a permite reviziurea eficienta a proiectului si pentru a explica echipei de mentenanță intenția și motivația deciziei luate.

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 ale acestuia sunt descrise. Pentru fiecare aplicatie de acces se definește cum aceasta afectează datele interne ale aplicației.

Descompunerea în submodule și design-ul acestora este continuată pană la punctul in care fiecare sarcină este suficient de sumara încat ea poate fi preluată de către un implementator.

**Implementarea efectivă**

După finalizarea design-ului și a documentației se poate trece la pasul de implementare ce constă in scrierea efectivă de cod executabil. Datorită faptului că pașii descriși mai sus au fost urmați, acest proces va decurge fără dificultați și cu o viteză ridicată. În cod nu se vor include comentarii redundante cu documentația deja existentă, astfel se reduc costurile de implementare și mentenanta a sistemului deoarece redundanța în comentariile de cod creste probabilitatea ca acestea vor fi inconsistente cu documentația existenta, fapt ce poate crea confuzie sau chiar incompatibilitati între interfețele modulelor dependente unul de altul.

**Mentenanța**

Mententanța consta în implementarea, reproiectarea și reimplementarea sistemului create. Pentru a mentine consistența codului executabil creat si a documentației existente în cazul apariției unei schimbări, aceasta trebuie specificată în documentație și proprietațile incompatibile cu acea shimbare șterse atât din documentația aferenta cât și din cod. Cu toate că la prima vedere costurile pe termen scurt par sporite, pe termen lung acest procedeu va păstra arhitectura sistemului calitativă ceea ce va scădea rata de distrugere a calitații sistemului.[1]

2.2.3 Tipuri de diagrame folosite in procesul de dezvoltare software

În urmatorul capitol vor fi descrise cele mai des folosite diagrame în procesul de dezvoltare și proiectare software. Rolul acestor diagrame este de a prezenta cititorului o viziune de ansamblu asupra unui aspect al aplicatiei (acesta aspect fiind diferit în funcție de tipul de diagrama utilizată). Utilitatea acestor diagrame este capacitatea de a oferi informația in mod grafic, ceea ce le face o sursa de informatii rapidă și usor de utilizat.[3]

**Diagrame de activitate**



Figura 2.2. Diagrama de activitate

O diagramă de activitate este o reprezentare grafică a fluxului de munca pe pasi indivizibili ce suportă structuri decizionale, iterații și concurența de procese. In UML 2x diagramele de activitate pot fi folosite pentru a descrie procesele de business sau de operare a unui sistem. O diagramă de activitate ofera o privire de ansamblu asupra fluxului de date din aplicatie și controlul acestor fluxuri. Aceste diagrame sunt construite dintr-un numar limitat de figuri geometrice conectate orientat. Cele mai importante și des folosite figuri sunt:

* Dreptunghiul rotunjit - reprezinta activitati;
* Rombul - structuri de decizie;
* Elipsa neagra – starea initiala a sistemului;
* Elipsa neagra incercuita - starea finala.

Sagețile sunt trase de la starea inițiala pană la starea finală și reprezintă ordinea în care fiecare activitate este desfasurata. In timp ce in UML 1.x aceste diagrame erau specializate pe reprezentarea starii interne a datelor, odata cu apariția standardului UML 2.x utilitatea acestora a fost extinsă.

**Arbori de decizie**

Acest tip de diagrame de la aparitia standardului UML 2.x a fost parțial inlocuit de către diagramele de activitate. Cu toate ca cele din urma ofera o informație mai vasta decat diagramele de decizie, acestea sunt in continuare folosite în special in procesul de comunicare cu clientul sau in comunicarea cu un specialist din domenii diferite decat IT datorita simplitatii lor atat de creare cat și de utilizare. Arborii de decizie sunt folosiți pe larg in domeniile informaticii ce au legatură cu inteligența artificială, un exemplu fiind teoria jocurilor sau data mining.

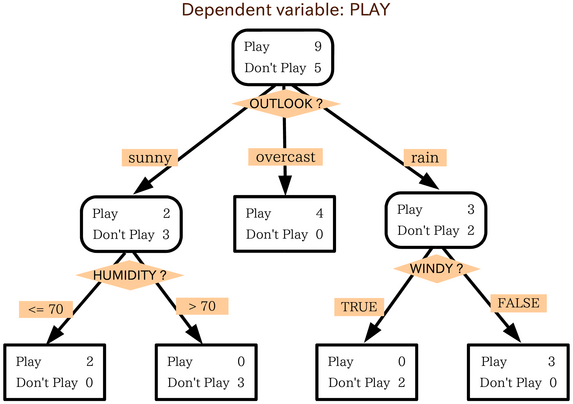


Figura 2.3. Arbore de decizie

**Diagrame de flux**

Diagramele de flux sunt deseori confundate cu diagramele de activitate, fapt ce se datoreaza responsabilitaților asemanatoare cu acestea. Insa acestea au statut de diagrame de tip diferit din cauza faptului ca în timp ce o diagramă de activitate prezintă o secventa de pași descriși la un nivel inalt ce trebuie urmați pentru a obtine un anume rezultat, o diagrama de flux prezinta o secventa de pași formalizați și indivizibili. De obicei diagramele de flux sunt folosite pentru descrierea algoritmilor in aplicație.

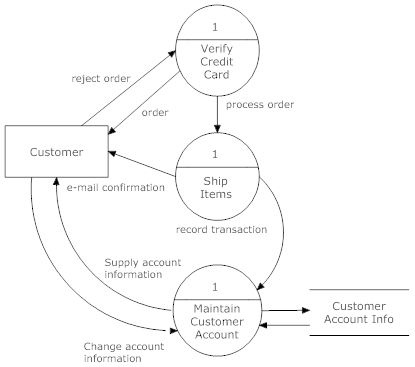


Figura 2.4. Diagrama de flux

**Diagrame de sincronizare**

O diagramă de sincronizare este o reprezentare a unui set de semnale intr-un spatiu de timp. O astfel de diagrama poate conține mai multe rânduri de valori, de obicei unul dintre acestea fiind ceasul intern. Acestea se folosesc pe larg în domeniulul electornicii digitale, depanare hardware si comunicații. În afara faptului ca acest tip de diagrame oferă o viziune de ansamblu asupra relației de sincronizare între procese, ea permite depistarea erorilor logisice posibile. În figura de mai jos se descrie un port serial periferic (SPI). Cele mai importante noduri din acel SPI au capacitatea de setare a polaritații ceasului atât cat și a fazei acestuia.

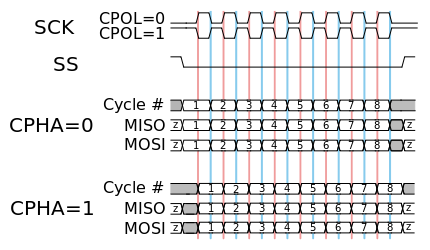


Figura 2.5. Diagrama de sincronizare

**Diagrame de stare**

Acest tip de diagrame este folosit în disciplinele ingineresti legate de predicția comportamentelor unui sistem informatic, matematic sau limbaje formale. Diagramele de stare necesita ca sistemul descris sa fie compus dintr-un numar finit de stări. Formele descrise difera una fața de alta prin semantica lor. Șablonul tranzițiilor între stări este analizat și reprezentat printr-o serie de evenimente care decurg în una dintre multiplele stări posibile. In concluzie fiecare diagramă de acest tip reprezeintă o clasă unitară de obiecte în diferitele stări ale sale pe parcursulul executiei simulării.

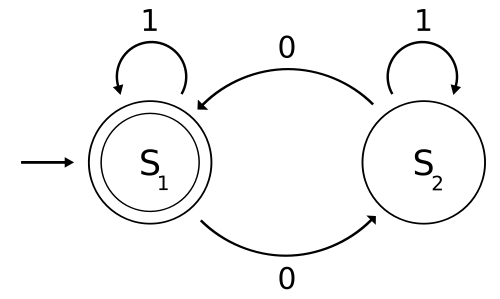


Figura 2.6. Digrama de stare

**Grafuri orientate si neorientate**

Un graf este 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 în domenii ce țin de comunicații la distanța, algoritmi de rutare și inteligență artificială.

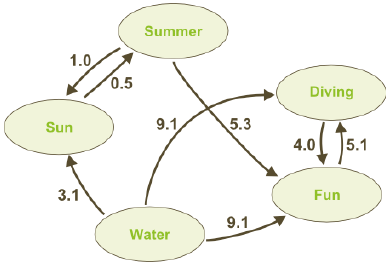


Figura 2.7. Graf orientat

**Constrangeri de integritate a bazelor de date**

O schema a unei baze de date este structura acestei baze de date definită intr-un limbaj formal. Această schemă reprezinta planificarea modului in care baza de date va fi construită, mai specific modul în care tabelele acesteia vor fi impărțite. Definirea formală a unei baze de date constă dintr-un set de formulări numite constrângeri de integritate impuse asupra acesteia. Aceste constrângeri de integritate asigură compatibilitatea între parțile schemei, astfel se poate asigura consistența datelor din modelul schemei.

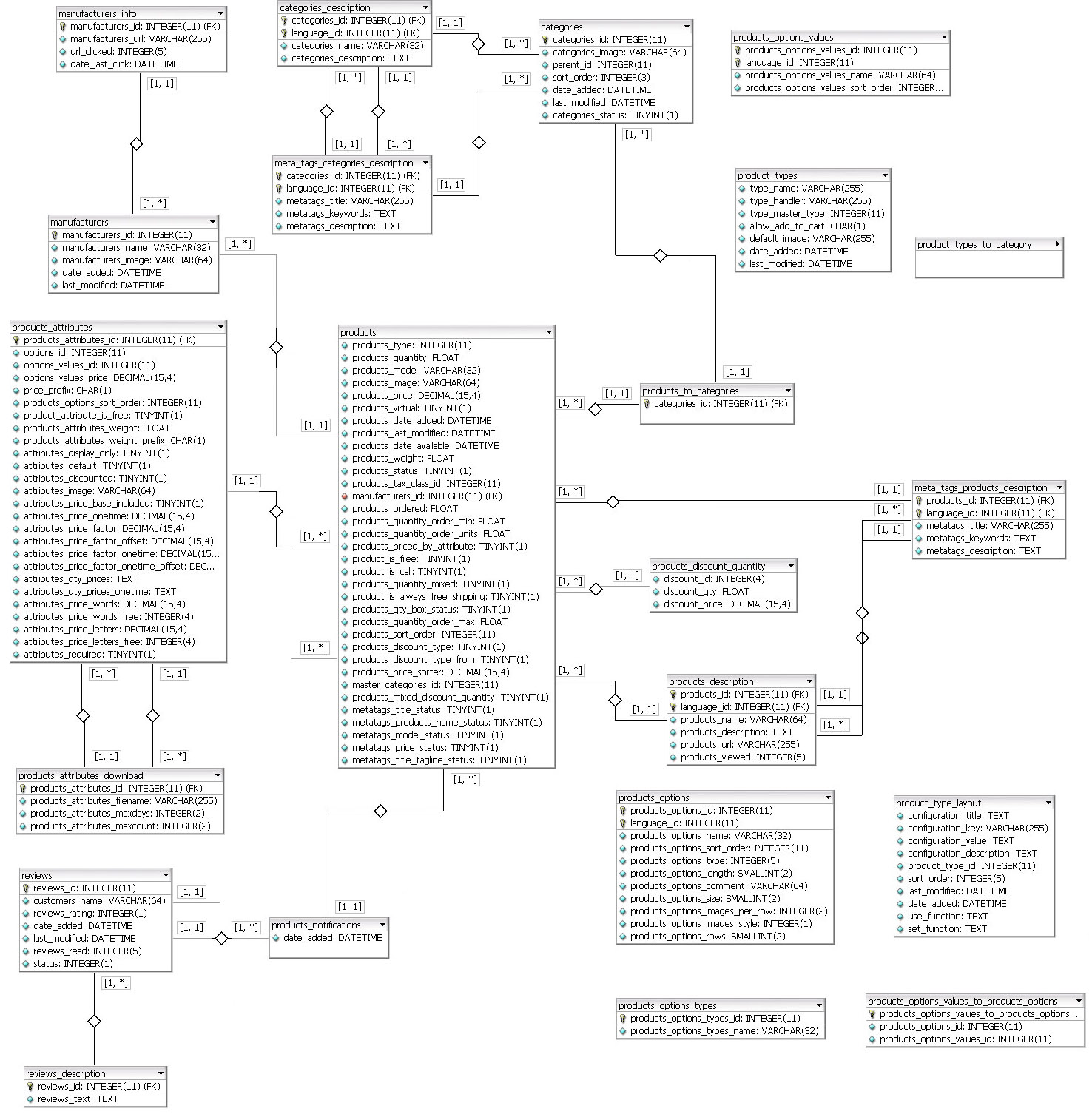


Figura 2.8. Diagrama de baze de date

**CAPITOLUL 3. Specificațiile aplicației**

3.1 Context general

În 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 concepute dupa un standard intern, adaptat domeniului în care activează acea companie. În plus acest tip de prezentare a informației (schematic prin folosirea diagramelor) a devenit extrem de popular și in discipline ce nu au legatură cu disciplinele ingineresti cum ar fi in psihologie, actividați didactice sau chiar în economie.

Putem identifica nevoia clară pe piata a unei aplicații ce poate facilita managementul, crearea si editarea de diagrame. Fara dubii exista o serie de aplicații ce indeplinesc intr-o oarecare masură nevoile unui anume segment de piața, acesta fiind și punctul slab al aplicatiilor de acest tip: incapacitatea de adaptare la grupuri diferite de utilizatori .

Deseori chiar și aplicatiile destinate desenarii de diagrame pentru un domeniu specific, cum ar fi IT-ul, sunt foarte rigide și neadaptate la nevoile utilizatorului. Cel mai bun exemplu fi aplicatia StarUML. Aceasta este o aplicate open source destinată desenării de diagrame de tip UML (Unified Modeling Language) gratuită și relativ simplu de folosit. Este foarte populară în mediul academic, mai specific folosită de catre elevi și studenți. Cu toate că această aplicație este destul de reusită ca și concept și implementare are o problema majora: neadaptarea la limbajul de programare ce urmeaza a fi folosit pentru implementarea sistemului. Prin urmare unele tipuri, constructii sau legaturi intre obiecte (definite in limbajul de programare lipsesc din diagrama de clase si vice-versa).

Cu toate ca limbajele orientate pe obiect au o structura relativ asemanatoare in materie de tipuri si relatii, exista diferențe care la prima vedere ar parea neimportante și nesemnificative insa care pot crea ambiguitați in etapa de implementare. Cu toate că standardul UML este global recunoscut, se permit mici modificări a unei entitați create pentru a se adapta mai bine la domeniul aplicației, ceea ce nu este suportat de marea majoritate a aplicațiilor de desenare a diagramelor de tip UML. Aceste mici modificări, necesare pentru a pastra consistenta intre diagramele de clase (de exemplu) si codul efectiv sunt ignorate, ceea ce va produce erori în pașii urmatori design-ului.

Pentru a demonstra ideea voi reveni la exemplul anterior. Creatorii StarUML afirmă că aplicatia lor suportă designul diagramelor de clase pentru limbajul Java. Cu toate că afirmația este partial adevarată, StarUML nu are specificat un tip de clasă cum ar fi “Clasele abstracte”. Astfel o clasa abstractă de obiecei prin convenție va fi prefixata cu <Abstract> in numele acesteia, convenție care de altfel nu exista in generatorul de STUB-uri prezent în aplicatie si prin urmare toate functionalitațile prezente la nivel de generare de cod sau verificare de model a diagramei, acestea fiind inoperabile.

O altă problemă prezentă la aplicatiile disponibile ar fi multitudinea de functionalitați oferite. Afirmația de mai sus poate fi controversata și poate contraintuitiva. Pentru a fi cat mai explicit voi da un alt exemplu de aplicație foarte des folosită de catre inginerii software: VisualParadigm.

Visual Paradigm este un tool folosit pentru desenarea diagramelor de tip UML având ca și client ținta ingineri ce activează la nivel profesional în domeniul proiectării software. Faptul că această aplicație a primit 5 premii majore in domeniul IT demonstrează faptul ca este o aplicație reusită, însa datorită faptului că publicul ținta al acestei aplicații sunt proiectanții de sistem seniori se oferă un numar atât de mare de funcionalitați încat un utilizator cu un nivel de experiență redus în domeniul proiectării sistemelor s-ar confrunta cu dificultați majore în identificare funcționalităților necesare acestuia. Interfata grafica extrem de încarcata fac identificarea functionalitaților cheie a aplicației extrem de dificilă, astfel un utilizator neinițiat, care ar vrea o solutie rapidă, nu își va putea crea acea soluție fară o pregatire riguroasă in folosirea aplicației. Figura de mai jos prezintă o captură de ecran ce conține interfața grafică oferită de către aplicația VisualParadigm.

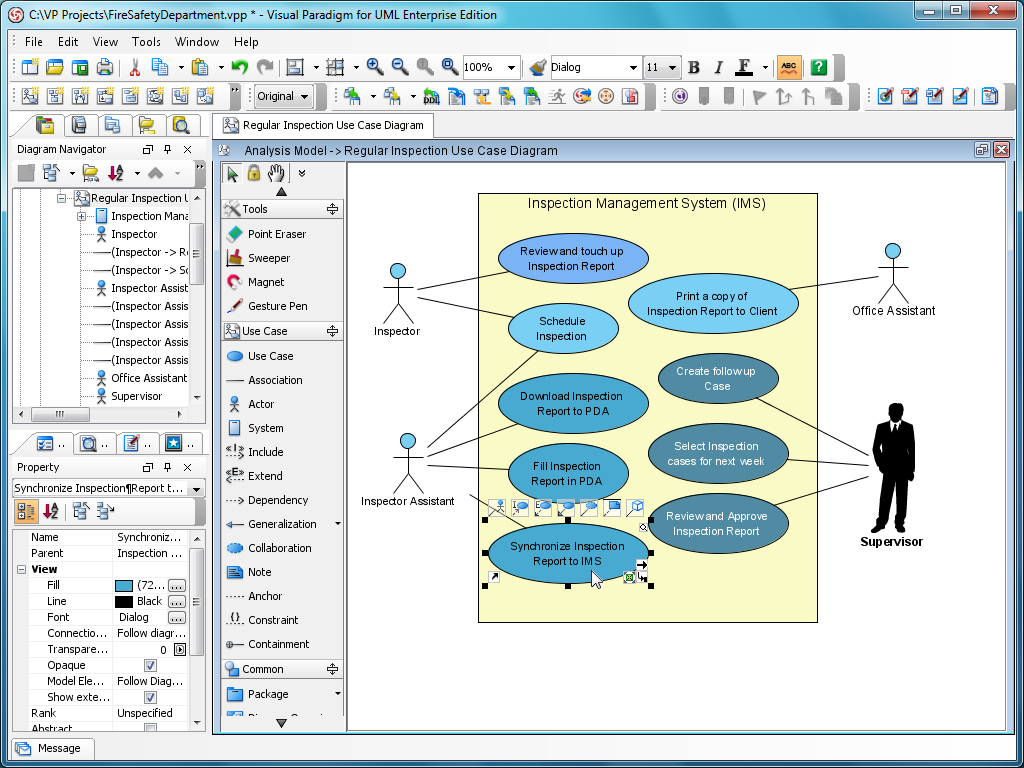


Figura 3.1. Interfață grafică Visual Paradigm

Există exemple numeroase care prezintă avantajele și dezavantajele folosirii aplicațiilor deja existente. În timp ce unele oferă functionalitați avansate dar sunt greu de utilizat, altele oferă functionalitați usor de utilizat dar mult prea simpliste pentru a crea o diagramă bine definită.

Prin aplicatia prezentată în această lucrare s-a încercat implementarea unui instrument ce va adresa aceste probleme. Printr-o separare clară între interfața grafică, controlul aplicației și functionalitațile oferite de catre aceasta a rezultat o aplicație scalabilă la cerințele utilizatorului, independent de domeniul în care activează sau de gradul de experiență al acestuia.

3.2 Specificatii si constrangeri

În aplicația prezentată s-a incercat păstrarea avantajelor altor aplicații de pe piață și eliminarea dezavantajelor acestora, fapt posibil datorită specificațiilor bine definite. În acest capitol se vor descrie specificațiile de nivel înalt ale aplicatiei și motivația implementării acesteia.

Un prim considerent major în construirea aplicației a fost **scalabilitatea** acesteia. Scalabilitatea se referă la comportamentul consistent al aplicatiei în diverse condiții de lucru. Aplicatia livrata trebuie sa fie suficient de scalabilă pentru a rula in parametri normali fie in cazul utilizării de către un utilizator pe un calculator personal, fie in cazul unei intreprinderi in care aplicatia este folosită concomitent de către câteva sute de utilizatori concomitent, folosind un server de baze de date pentru date. Scalabilitatea unui sistem software are la baza gama de tehnologii și biblioteci folosite pentru dezvoltarea acestuia.

Adaptabilitatea redusa a unei aplicatii ii reduce drastic durata de viata, in special in cazul unei aplicatii orientata pe utilizator privat (și nu pe mediu enterprise) cum este aplicația prezentată. Astfel putem deduce un alt set de constrangeri de care va trebui tinut cont: **adaptabilitatea la limbaj și standarde**. Cu fiecare schimbare adusă unui limbaj deja existent, apariție de standard nou în modul de desenare a diagramelor, sau apariția unui limbaj total nou, aplicația trebuie actualizată pentru a permite acomodarea acestor schimbari.

O aplicație care are un proces de instalare și configurare complicat este în fond o aplicatie cu o intretinere dificilă. În cazul aplicațiilor foarte dependente de platformă și cu multe fisiere de proprietați, variabile de sistem sau căi de acces, se poate ajunge la situatia în care marea majoritate a clientilor fie nu vor instala aplicatia și vor căuta o alternativă mai simplu de configurat, fie vor apela la producatorii aplicației pentru asistența, ceea ce va creste costurile de întreținere. În concluzie, este nevoie ca aplicația dezvoltată sa fie **ușor de configurat.**

Cu toate ca liderul pe piața în materie de sisteme de operare ramâne Microsoft cu produse din seria Windows, există un segment de piața destul de extins ce folosește sisteme de operare alternative pentru stațiile personale de lucru, de exemplu Ubuntu sau MacOS. Pentru a nu pierde acel segment de utilizatori se propune ca aplicația sa fie **independenta de platforma si de arhitectura interna a sistemului.**

Studiile arata ca aplicațiile cu interfața grafică prea încarcată sau care nu evidențiază funcționalitațile cele mai relevante pentru utilizatori iși pierd popularitatea. Pentru a nu intra în acea categorie de aplicații se propune crearea unei **interfețe grafice prietenoase și configurabile.**

**CAPITOLUL 4. Descrierea aplicației**

4.1 Conceptul de decuplare

Pentru a obtine un produs software extensibil la nivel funțional cum este descris în documentul de specificații (capitolul anterior) a fost decisă folosirea unor notiuni de inginerie a sistemelor bazată pe componente (Componet-based software engineering), prescurtat CBSE. CBSE este o ramură a ingineriei software care pune accent pe separarea modulelor în funcție cu functionalitațile diversificate puse la dispozitie de către un sistem software. Abordarea bazată pe reutilizarea componentelor permite crearea de module independente și reutilizabile. Ca și mod de lucru dezvoltatorul software privește componentele ca pe entitati de sine statatoare cu scopul de a oferi servicii, nu ca și o componentă a aplicației ce va functiona în paralel cu altă componentă pentru indeplinirea unei sarcini.

Fiecare componentă software individuală este un pachet software, modul sau serviciu web care incapsulează un set de functionalitați sau date înrudite. Fiecare dintre procesele sistemului este separat în componente diferite astfel că toate functionalitațile înrudite semnatic se regasesc într-un modul, astfel se ajunge la cohesiunea integrală a fiecarui modul.

În ce privește coordonarea globala la nivel de sistem software, modulele vor comunica prin interfețele acestora. Cand o componentă oferă servicii sistemului, acesta adoptă o interfața oferita de către sistemul aferent ce specifică ce servicii și componente ale modului pot fi utilizate. Această interfață poate fi vazută ca și semnatura componentei – clientul nu necesită cunoștințe despre funcționalitatea internă (implementarea modulului) pentru a folosi serviciile oferite de catre acesta, principiu ce duce incapsularea la nivel de modul, nu doar de clasă (ca și în proiectarea orientată pe obiecte clasică).

O altă caracteristică importantă estea acea că modulele pot fi substituite unele cu altele chiar și în timpul rulării în caz că noul component adaugat adoptă interfața specificată. Prin consecintă acele componente pot fi înlocuite cu versiuni mai noi ale lor sau cu module alternative fără sa interfereze cu sistemul în care acestea functionează.

Refolosirea componentelor este o altă caracteristică a sistemelor proiectate în acest mod. Programatorii trebuie sa creeze atât o arhitectură cât și o implementare pentru fiecare modul încat sa îi permită acestuia să fie refolosit în diverse proiecte. Pentru a obtine module reutilizabile acestea trebuie sa se supună urmatoarelor reguli:

* documentate în întregime
* sa aibă capacitatea de a pasa erori și coduri de eroare modulelor cu care interactionează
* poiectate în ideea că aceste module vor fi întrebuintate deseori în mod eronat.

Când este necesar ca o componentă sa fie accesată prin contextul execuției sau prin rețea, concepte ca și serializarea sau marshelling sunt utilizate pentru a livra componenta sistemului client. Un sistem ce rulează mai multe componente este numit server de aplicatie. Folosirea unei combinatii de servere de aplicatie și componente software se numeste “Computatie distribuită”. Aplicabilitatea acestei tehnologii se referă în special la domeniul bancar și aplicațiilor business.

Ca și exemplu acest tip de abordare a fost folosit pentru EJB (Enterprise Java Beans), COM+ model (modelul folosit de platforma .NET) și multe altele.[2]

4.2 Nivele functionale și arhitectura aplicatiei

Aplicatia descrisă folosește într-o masură foarte mare abordarea orientata pe modulul, descrisă mai sus. Pentru a păstra implementarea și design-ul într-un mod mentenabil, funcționalitățile aplicației au fost distribuite în cinci subnivele funcționale principale, care la randul lor sunt împarțite în subnivele. În figura de mai jos este prezentată diagrama de nivele a aplicatiei.

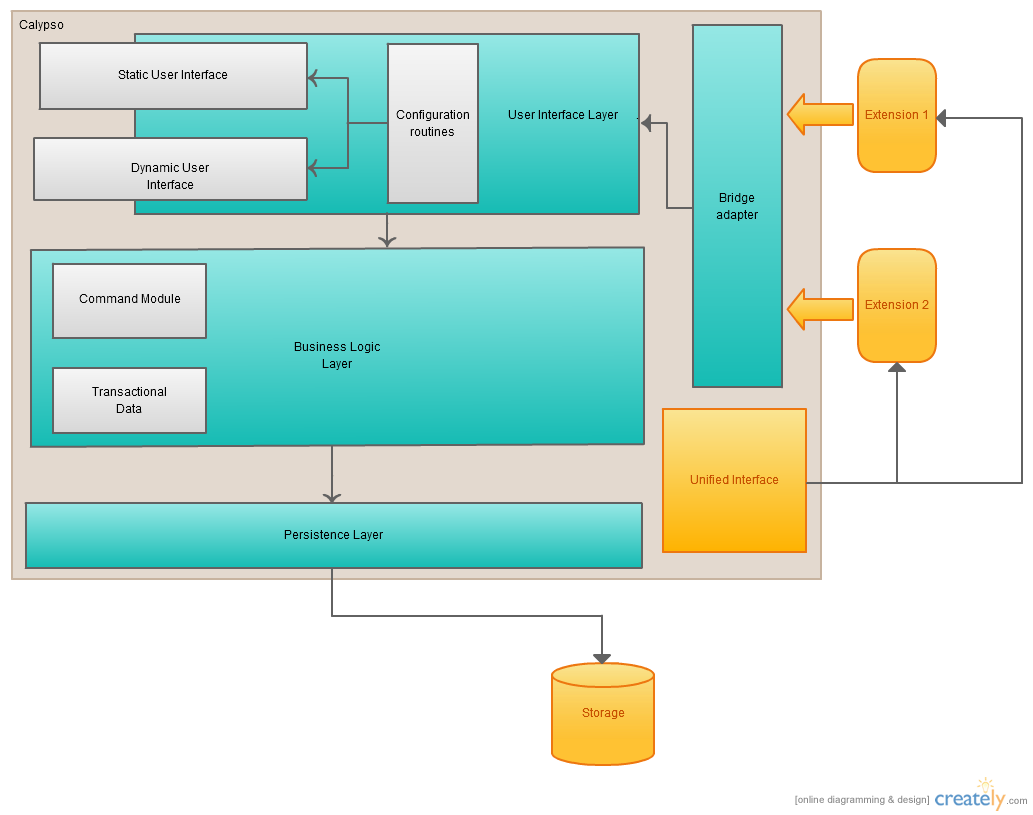


Figura 4.1. Diagrama de nivele a aplicatiei

În continuare va fi descris fiecare subnivel funcțional în detaliu și componentele din care este alcatuit acesta.

**Nivelul interfeței cu utilizatorul**

Acest nivel conține, dupa cum îi sugerează și numele, toate clasele ce au legatura cu interfața cu utilizatorul, scopul acestuia fiind punerea la dispozitie a tuturor funcționalităților aplicației luând în considerare un set de constrângeri pentru executarea acestora. Execuția unor operații depinde de contextul în care acea operație se va desfăsura: de exemplu o operație de “undo” nu poate fi executată dacă nu s-a comis nici o operație asupra documentului curent, la fel cum o operație de “redo” nu se poate executa daca nu s-a comis “undo” în prealabil. Nivelul interfetei cu utilizatorul ține cont de aceste constrângeri și are grija să evidențieze operațiile valide în contextul curent sau sa ascundă operațiile invalide. Această funcționalitate este implementată prin inactivarea controalelor. Astfel o operație invalidă va fi semnalată prin modificarea culorii butonului ce o executa și prin dezactivarea sa la nivel funcțional.



Figura 4.2 Bara de instrumente inactiva



Figura 4.3 Bara de instrumente activa

Prin implementarea interfeței grafice în acest mod, se asigură simplitatea utilizarii (cerință cheie a aplicație). Astfel unui utilizator neexperimentat îi va fi foarte clar ce operații sunt valide și care sunt valide, fară perturbarea acestuia prin mesaje de eroare. Cu toate că s-a încercat implemetarea acestui tip de protecție împotriva erorilor în cat mai multe cazuri, există erori ce nu depind de context, și au ca și cauză parametrii invalizi oferiți aplicației. În acest caz se afișează un dialog ce prezintă eroarea.

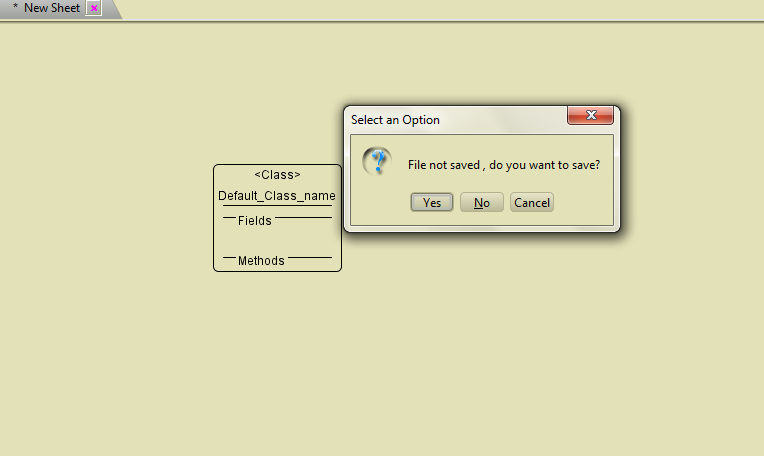


Figura 4.4. Dialog de validare a comenzii

Pentru a oferi o experientă intuitivă s-a implementat un alt mod de control al stării interne a datelor aplicației: controlul activ al componentelor. Astfel aplicatia rezultată are 2 module masive de interfață grafică: Modulul static și modulul dinamic. Acestea vor fi descrise in continuare.

**Modulul static de interfață cu utilizatorul**

Acest modul conține toate ferestrele folosite în aplicație și reprezintă componentele persistente ale acesteia, care nu vor varia în funcție de extensiile aplicației. În modulul static al interfeței cu utilizatorul se pot regasi componente vizuale ca: meniul aplicației, bara de unelte, lista de entitați/conectori disponibili. De altfel aici se vor regasi toate ferestrele de setare petru un nou document sau proiect.

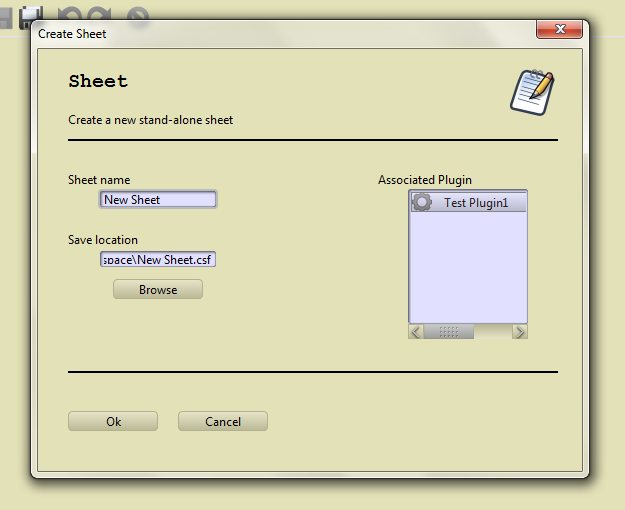
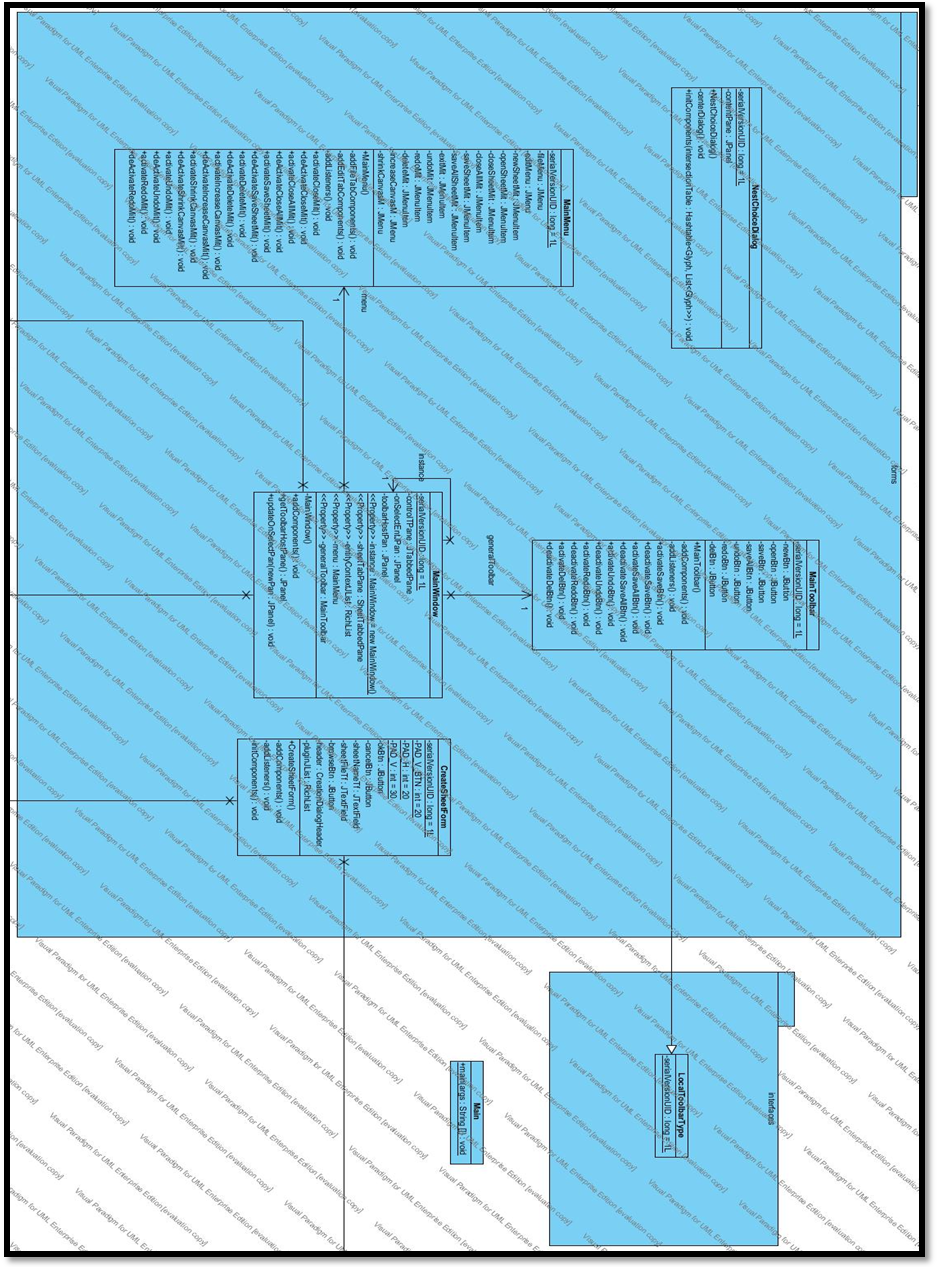


Figura 4.5 Meniu creare de noi documente

Pentru a facilita transferul de referințe catre aceste fereste, în aplicație ferestrele de interes vor avea (la nivel de implementare a codului) interfețe ce le vor defini funționalitățile, și referințele către acestea vor fi transmise ca și tipul interfeței folosind șablonul de proiectare MVC (Model View Controller). Astfel se crează o arhitectură decuplabilă, ceea ce duce la posibilitatea schimbării interfetei grafice statice fara modificari în afara pachetului ce o contine.

În continuare va fi prezentată diagrama de clase a componentei de control și de afișare pentru acest modul:

Figura 4.6. Diagrama de clase a interfeței statice



**Modulul de interfata dinamică cu utilizatorul**

Modulul dinamic de interfață cu utilizatorul constă dintr-un set de componente interactive puse la dispoziția utilizatorului. În principiu modulul de interfața dinamica cu utilizatorul se referă la editorul de documente. Acest modul oferă diverse forme de componente care sunt unice proprietaților de model ale aplicației. Pentru a oferi o utilizare placută și intuitivă a aplicației, fiecare componentă desenată are un set propriu de functionalitați dinamice.

De exemplu, modificarea dimensiunilor unei entitați existente se face prin intermediul unei operații de “drag-and-drop” ce se execută asupra unei componente ce leaga entitatea cu proprietațile sale. Aceste tipuri de functionalitați oferă utilizatorului care le invocă feedback în timp real, astfel acesta își poate da seama care va fi starea obiectului dupa execuția operației și eventual sa întrerupă operația în cazul în care rezultatul nu este cel așteptat.

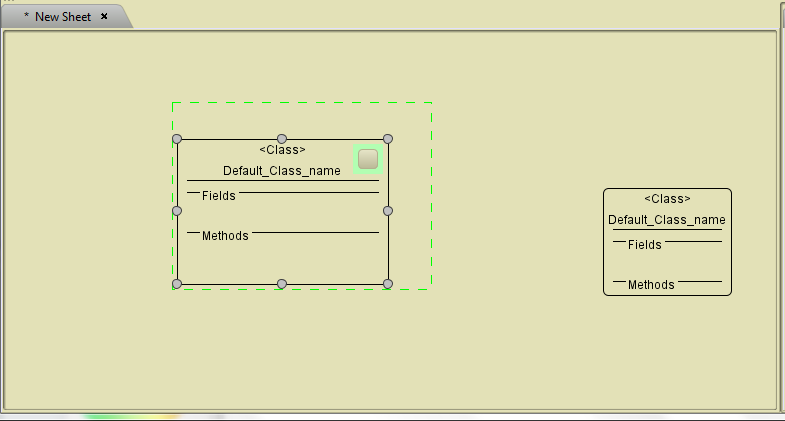


Figura 4.7. Exemplu de comportament de redimensionare



Figura 4.8. Exemplu de comportament de relocare

Acest tip de funcționalitate este implementată cu ajutorul metodei “Glass pane” și “Double buffering”. Glass pane constă în adăugarea unui container asupra unui alt container semitransparent ce desenează pozitiile auxiliare ale componentelor și datorita transparenței sale permite vizualizarea concomintentă a elementelor asupra carora nu se execută operații. Acesta este reactualizat la fiecare mișcare a mouse-ului ce are loc în procesul de executie a operației.

Datorită randării intense descrise mai sus poate aparea efectul de “rupere a imaginii” care constă în deteriorarea imaginii în cazul în care numarul de frame-uri pe secundă este foarte mare (ca în cazul de față). Acest efect a fost înlăturat cu ajutorul tehnicii “Double Buffer”. Aceasta tehnică constă în folosirea unei componente de tip Graphics2D auxiliar care va pastra noua imagine, pentru ca apoi sa fie înlocuit cu componenta ce se află pe ecran.

O altă parte a modulului de interfață dinamica cu utilizatorul este “panoul de lucru”. Acest panou reprezintă funcționalitățile obiectului (entități) desenat și își modifică conținutul la nivel de operații disponibile și aranjamentul în pagină în funcție de următorii factori :

* ce obiect este selectat;
* cărui punct de exentensie aparține obiectul selectat;
* care sunt datele interne ale obiectului;
* ce operații sunt valide/invalide în contextul curent.

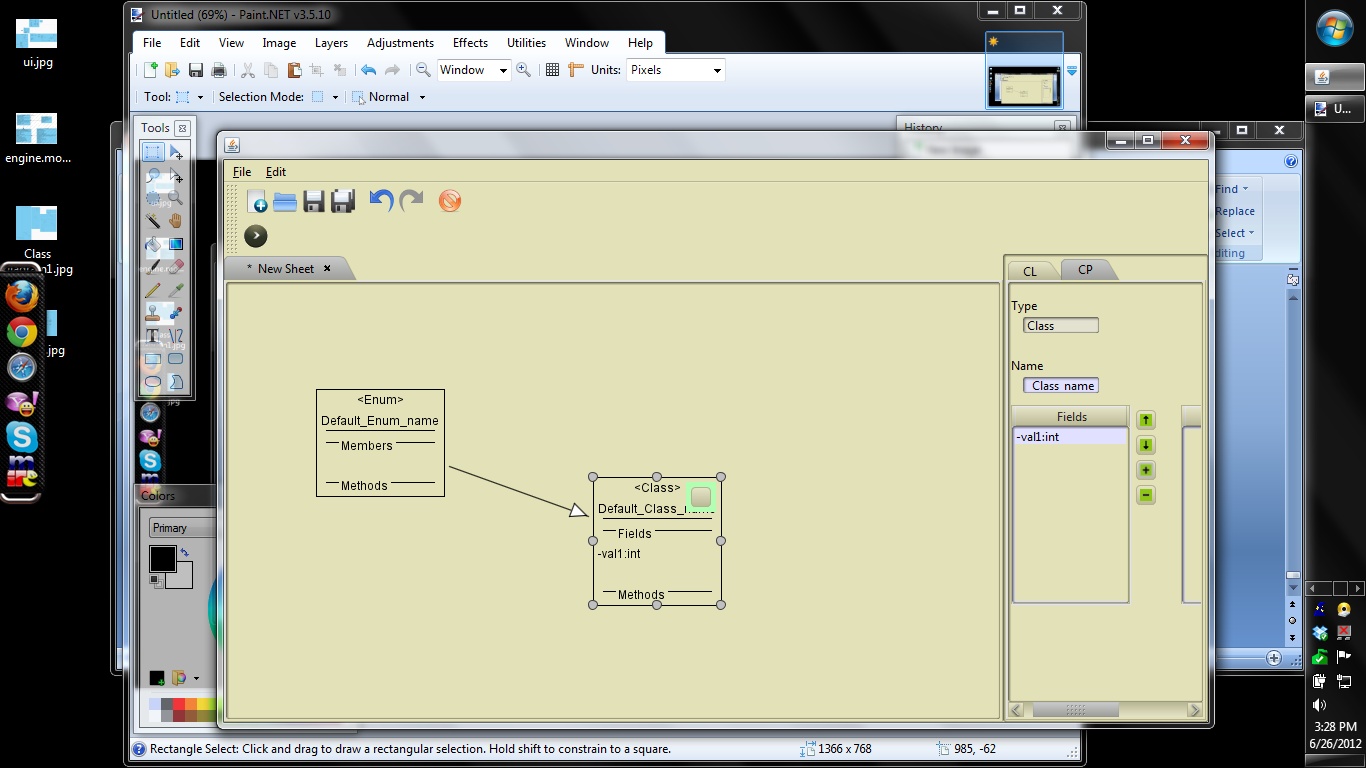


Figura 4.9. Exemplu de meniu dinamic - entitate

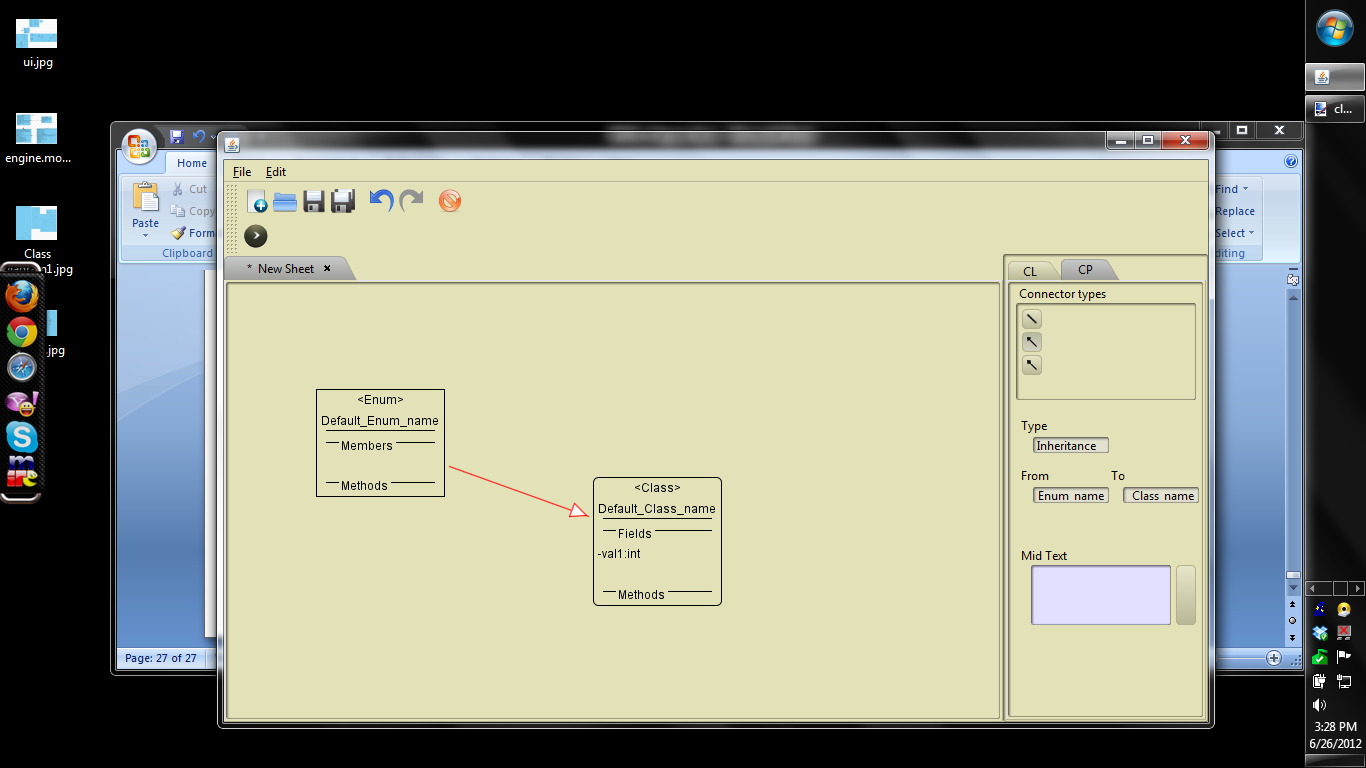


Figura 4.10. Exemplu de meniu dinamic – conector

**Date tranzacționale**

Datele tranzacționale ale aplicației reprezintă starea internă a obiectelor folosite în aplicație. Fiecare entitate sau conector ce poate fi desenat cu ajutorul aplicației are ca și corespondent un obiect ce aparatine grupului de date tranzacționale. Obiectele ce corespund acestui nivel se regăsesc în pachetul engine.model și sunt numite în conformitate cu regula : <componenta corespondentă>Model.java.

Aceste date sunt cele mai uzitate de către aplicație. Ele vor fi salvate către nivelul de persistență pentru utilizarea lor ulterioară, vor fi încarcate de catre acelasi nivel de persistența pentru a fi modificate și vor fi transferate punctelor de extensie pentru ca sa poată fi modificate de către acestea în decursul rulării aplicației.

Prin concluzie pentru a putea corespunde cerințelor aplicației acest nivel a fost supus unei analize riguroase pentru a fi proiectat cat mai independent de alte module și componentele acestuia cat mai independente una de alta. Prin urmare fiecare obiect din nivelul de date tranzacționale va implementa o interfață cu ajutorul căreia va fi transmis către obiectele client.

O funcționalitate ce o prezinta aplicația estea cea de a crea noi entitați copuse (în cazul punctului de extensie livrat cu aplicația: reprezentări de clase nested). Astfel reprezentările grafice ale claselor create vor acomoda în interiorul lor alte reprezentări de clase. Pentru a face posibil acest comportament a fost creata o clasă abstractă Glyph ce oferă o lista de entități interioare. Această clasă este superclasa tuturor obiectelor de model în afară de ConnectorModel (care conceptual nu poate avea entitați interioare). Această clasă nu este prezentă în diagrama de clase de mai jos din cauza faptului ca ea este declarată în pachetul de date transferabile către extensii (descris în urmatoarele capitole).

Pentru a permite extensibilitatea obiectelor de model ale aplicație fara a oferi posibilitatea modificării structurii de date ale acestora, proprietațile ce tin de conținutul acestor obiecte a fost incapsulat în obiecte de tip DataModel, ce constă dintr-o listă de proprietați și o cheie unică pentru identificarea acestora. Astfel din extensiile aplicației pote fi adaugat un numar nelimitat de proprietați de continut, fara a necesita accesul către structura internă a obiectelor asupra caruia se executa operația.

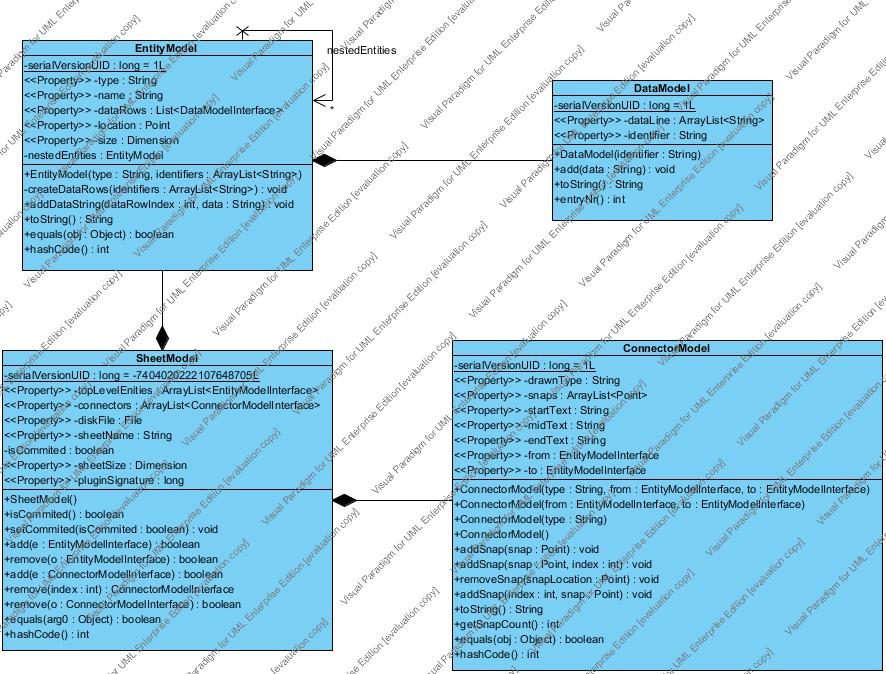


Figura 4.11. Diagrama de clase – date tranzacționale

**Nivelul de persistență**

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).

În 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:[4]

* 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 bridge, descris în acest capitol are ca și scop transferul de date și comenzi de la punctele de extensie către aplicație. Acesta va fi reponsabil de ciclul de viața a obiectelor inițializate în extensii dar folosite în aplicație. O altă repsonsabilitate a sa este de a injecta funcționalităti preulate din extensii în interfața grafică statică a aplicației.

De exemplu dezvoltatorul de extensii vrea să ofere utilizatorului aplicației un mod de a exporta diagrama creată în mod text (de exemplu cod Java din UML). Pentru a obtine acest comportament el trebuie să definească metoda prin care se generează textul:

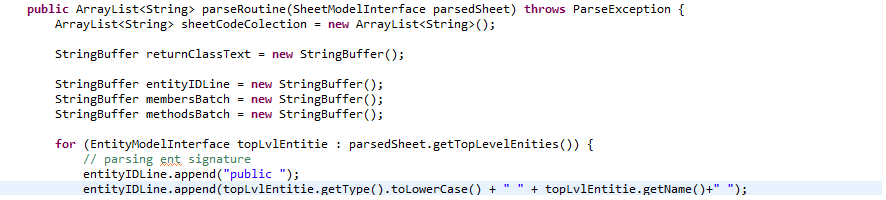


Figura 4.12. Declarația metodei de parsare

În interiorul punctului de extensie se va crea metoda ce executa acea operatie apoi se va oferi această funcționalitate la nivelul interfeței grafice (de exemplu un buton pe panoul de comandă central al aplicației).

Pentru a-și atinge scopul modului bridge este implementat prin modelul RPC (remote procedure call).RPC constă în comunicarea între procese și permite unei subrutine să execute o procedură dintr-un alt spațiu de memorie sau adresa fară ca programatorul să fie nevoit să definească explicit modul de comunicare inter-proces.

În aplicația prezentată funcționalitatea RCP a fost implementată cu ajutorul tehnologiei Reflection oferite de către java. Reflection permite examinarea și modificarea comportamentelor unui obiect în tipul rulării aplicației. În cazul concret transferul de date și functionalități între aplicația centrală și punctele de extensie are loc folosind urmatoarea secvență de acțiuni logice:

* se importă toate punctele de exensie din folderul specificat în fisierul de proprietați aplicației;
* se crează o noua instanță a punctului de acces catre extensie;
* se incarcă datele interne ale aplicației relevante pentru extensie și extensia încarcată își va modifica statutul in “extensia curenta”.

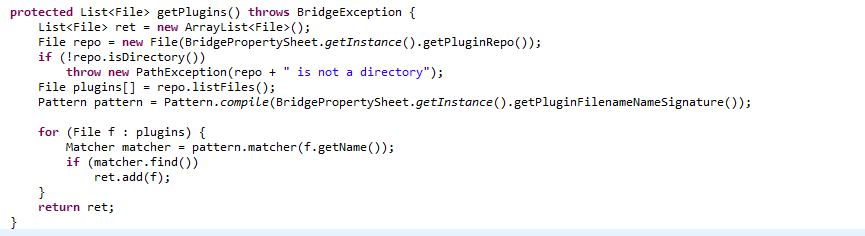


Figura 4.13. Definitia metodei de încarcare a extensiilor

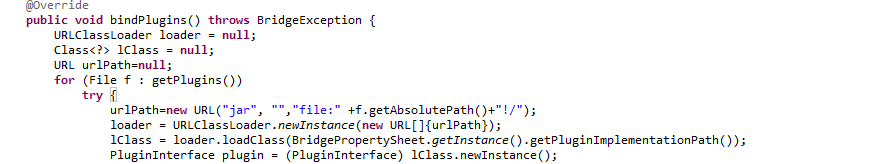


Figura 4.13. Definitia metodei de instanțiere a extensiilor

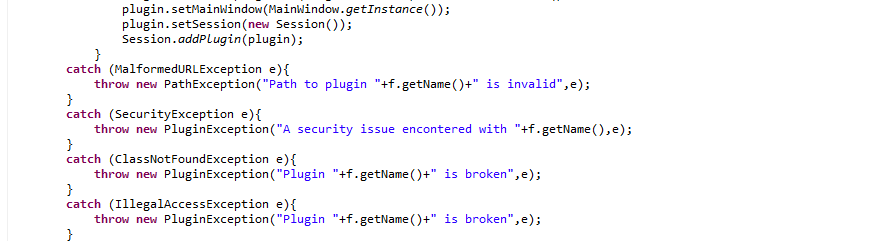


Figura 4.13. Schimbarea statutului punctului de extensie in “extensie curentă”

Pentru a putea executa rutinele descrise mai sus atât cât punctele de extensie cât și aplicatia centrală trebuie să implementeze acelasi protocol de comunicare și o interfață unificată.

Această interfața se poate regasi în sub-pachetul brindge.transferable atât în aplicatie cat și în API-ul oferit pentru dezvoltarea de extensii. Toate obiectele ce au instanțe nepartajate vor fi definite aici în funcție de scopul acestor obiecte.

Pachetul “contexts” reprezintă interfețele claselor cărora li se delega funcționalitățile specifice extensiilor. De exemplu modul de desenare al unei entitați va fi modificabil din extensie, astfel un obiect de tipul EntityView va delega comportamentul de desenare unui obiect agregat de tipul EntityContext obținut din punctul de extensie.



Figura 4.14. Diagrama de clase a pachetului *Contexts*

Aceste contexte vor fi distribuite între aplicații in modul urmator: interfața clasei va fi regasită atât în aplicația de bază cât și în punctul de extensie, în timp ce implementarea se regaseste pe partea extensiei în exclusivitate.

În timp ce funcționalitatea entităților din aplicatie este distribuită între obiectul concret (local) și contextul acestuia (remote) există mai multe tipuri de obiecte ce necesita transferul integral către extensie sau vice-versa. Obiectele ce necesită transferul integral prin bridge își vor avea interfața în subpachetul bridge.transferable.interfaces. De precizat ca nu obiectele concrete sunt localizate aici, ci doar interfețele aceastora în conformitate cu “dependency inversion principle”. Principiu care reduce cuplajul între obiecte (în cazul nostru reduce cuplajul între aplicație și API-ul extensiei sale).

Un exemplu foarte relevant ar putea fi obiectele de tipul CommandInterface. Pentru crearea segmentului de undo-redo a fost folosit șablonul de proiectare command. În cazul execuției unei operații se va crea un obiect de tipul CommandInterface care va memora starea membrilor pe care acesta îi modifică înainte și după operație. Astfel obiectul se poate readuce la starea sa inițiala prin aplicarea valorilor vechi. Ce relevanță are acest segment pentru modulul bridge:

Cum am afirmat în capitolele precedente dezvoltatorul de extensii are capacitatea de a-și înregistra propriile rutine în aplicație. Fara acces la segmentul undo acesta nu își va putea înregistra operatiile, prin concluzie va pierde funcționalitatea de undo-redo. Pentru a înlătura acest comportament nedorit acestuia i s-a oferit acces către interfața CommandInterface. Pentru a-și înregistra o comanda ce se initializează la apasarea unui buton dezvoltatorul de extensii își va crea un obiect ce implementează această interfața unificată și il va înregistra în obiectul de tip UndoSegmentProxy. [4]

În general aceeași funcționare o au obiectele de model ce vor fi transmise prin interfața acestora către extensie. Asfel metodele din extensie vor putea modifica un obiect ce se află în memory pool-ul aplicației centrale prin interfața sa, fără a avea dependinte concrete de tip, ci doar de interfața. În general obiectele ce sunt transmise integral sunt obiecte ce apartin modelului aplicației (transactional data layer) și sunt parte a ierarhiei de dependințe din API-ul oferit pentru dezvoltarea de extensii.

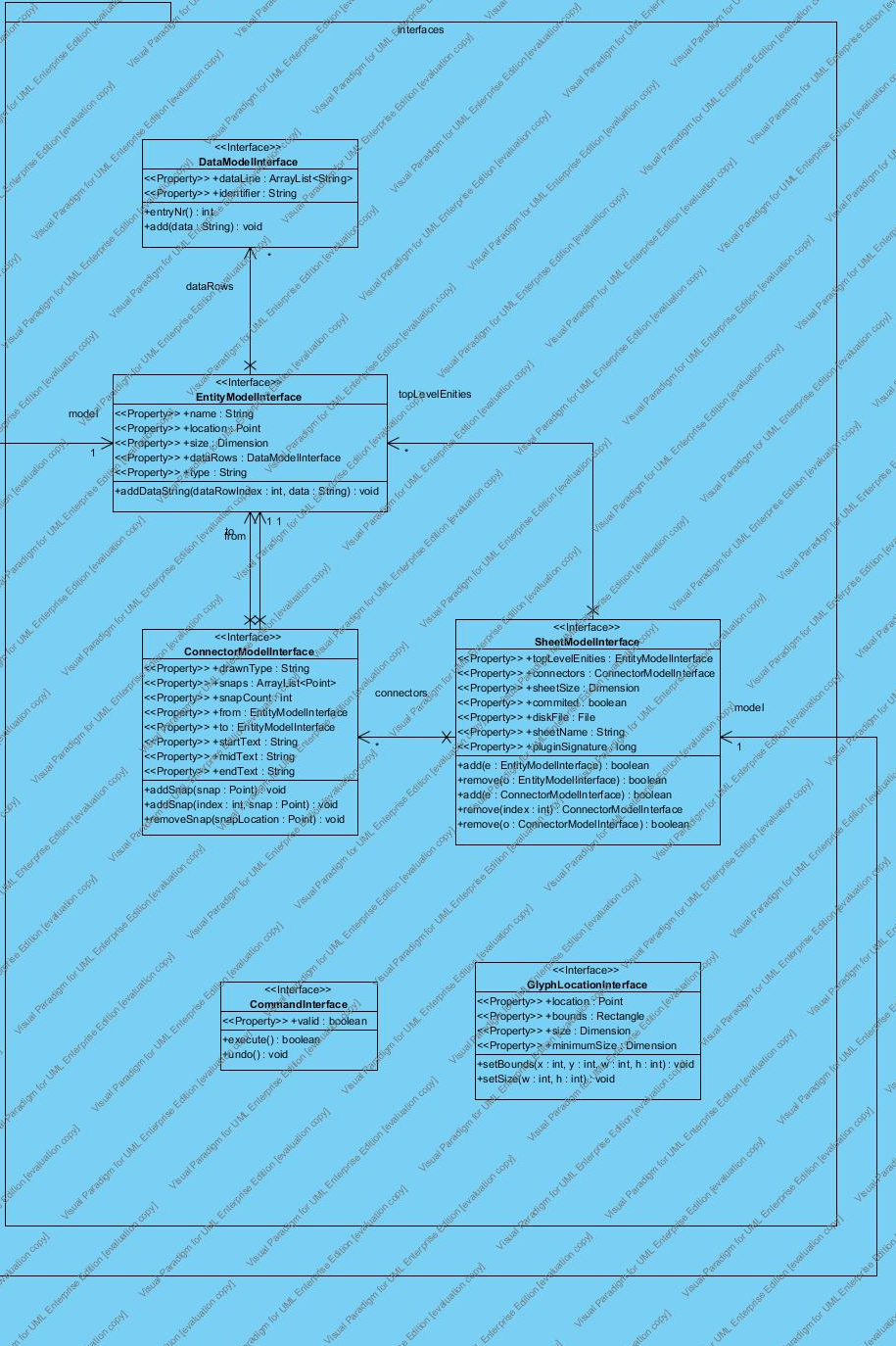


Figura 4.15. Digrama de clase *bridge.transferable.interfaces*

**CAPITOLUL 5. Concluzii**

O diagramă este o reprezentare bidimensiolă simbolică a informațiilor în conformitate cu un protocol de vizualizare și interpretare a datelor. Acestea sunt extrem de utile datorită faptului ca pot prezenta informații complexe într-un mod foarte succint dar explicit în același timp. Diagramele sunt cel mai des utilizate in discipline ingineresti dar în special în cele ce au legatură cu tehnologia informației.

Aceste diagrame pot fi avea o infinitate de tipuri și reprezentări, fapt ce face imposibilă crearea unei aplicații software ce poate viza toate disciplinele si toate tipurile de diagrame existente.

Însa prin crearea unei aplicatii de tip RPC se poate oferi utilizatorilor un mod de a-și personaliza entitațile pe care acesta le va utiliza în crearea diagramelor necesare.

Aplicația prezentată cu numele Calypso face posibila aceasta abordare. Datorită modulului de injectie de cod functionalitați pot fi inserate și extrase din aplicație la timpul rularii. Astfel se poate modifica si tipul diagramelor pe care Calypso le crează. Asfel se obtine un sistem alcatui dintr-un manager de ciclu de viață a obiectelor (ce reprezintă aplicația centrală) și functionalitați distribuite ce se regasesc în extensii.

Prin urmare aplicatia permite crearea oricarui tip de diagrame într-un mod intuitiv adaptat la utilizator.

**BIBLIOGRAFIE**

1. David L. Parnas , Paul C. Clements - A RATIONAL DESIGN PROCESS: HOW AND WHY TO FAKE IT
2. <http://wiki.eclipse.org/index.php/Rich_Client_Platform> - Eclipse Rich Client Platform
3. Martin Fowler - UML Distilled
4. [Eric Freeman, Elisabeth Robson, Bert Bates, Kathy Sierra](http://shop.oreilly.com/product/9780596007126.do#tab_03) - Head First Design Patterns
5. Manish Parashar - Distributed Computing: Introducetion