**CAPITOLUL 4. Descrierea aplicației**

2.1 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.

2.2 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.

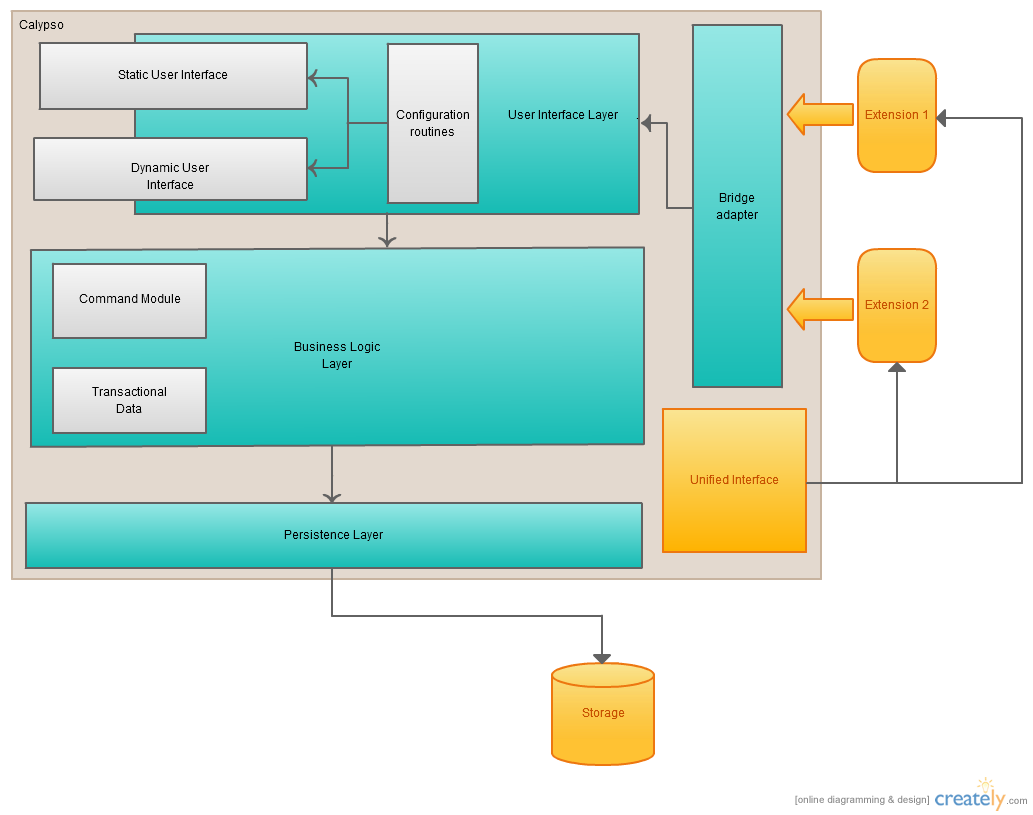


Figura 4.1. Diagrama de nivele a aplicatiei

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 funțional.

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.

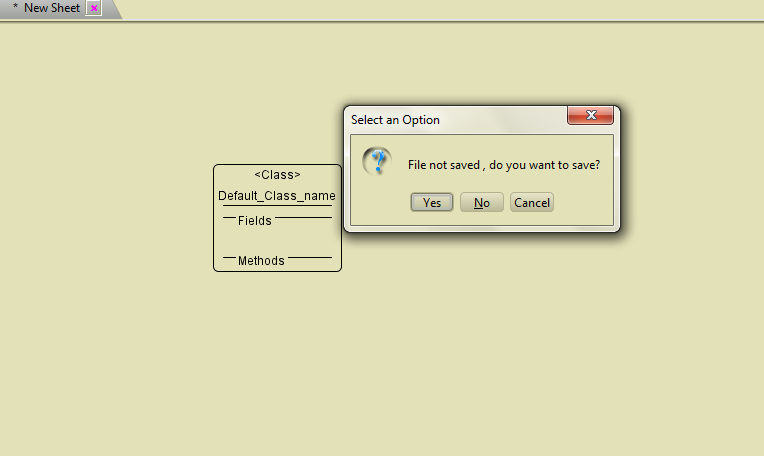


Figura 4.4. Dialog de validare a comenzii

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.

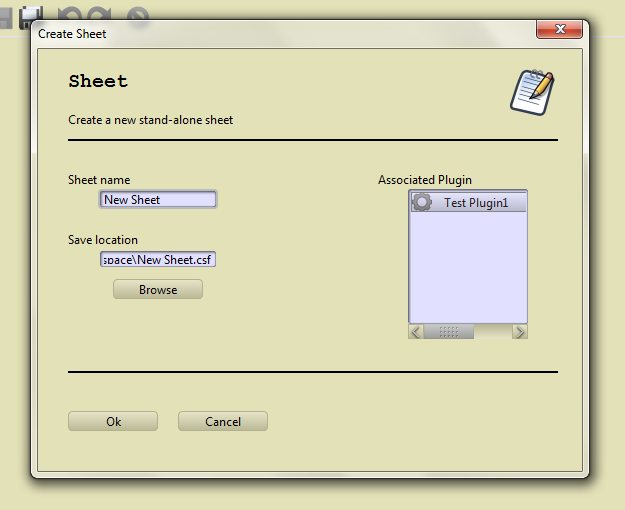
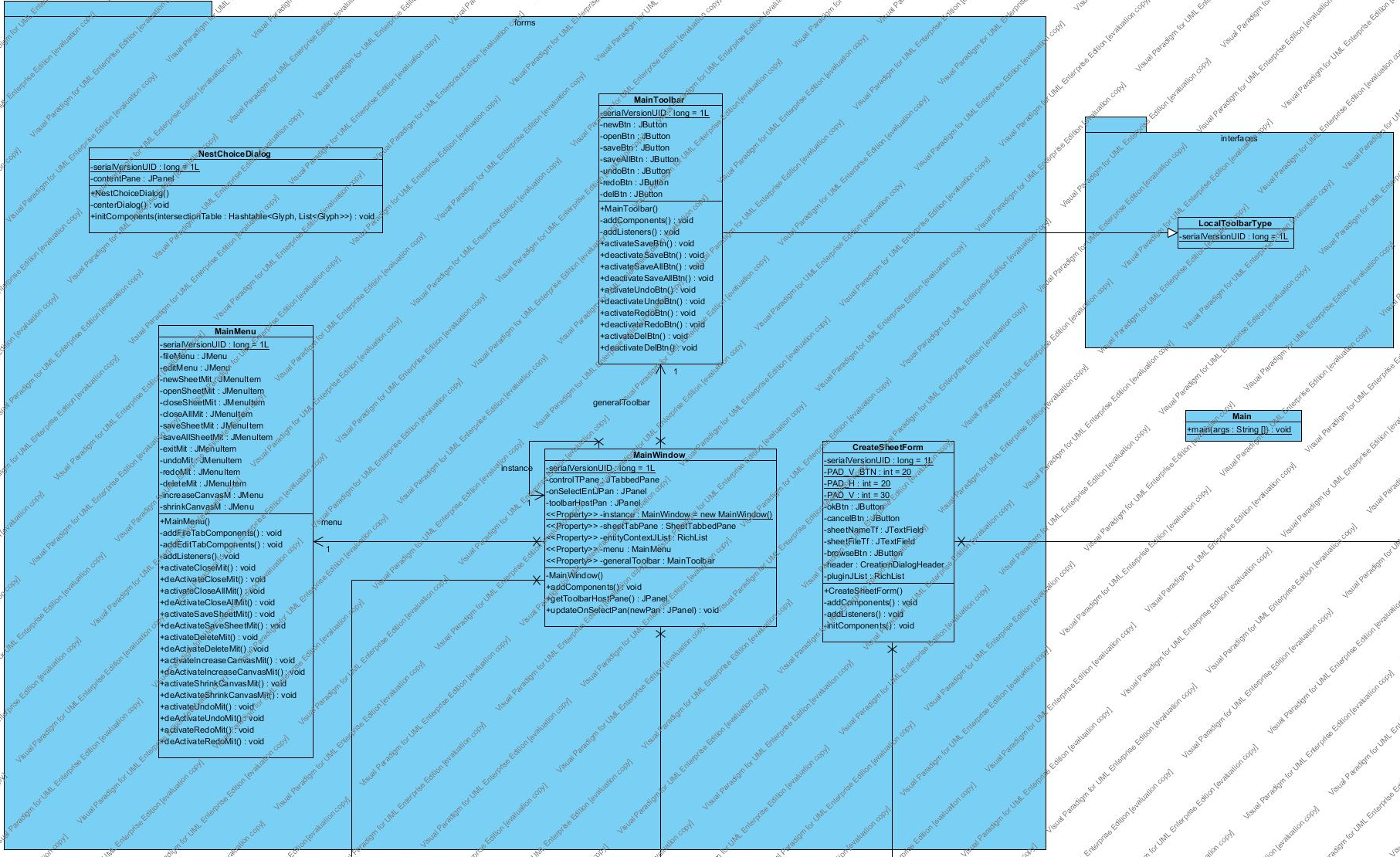


Figura 4.5 Meniu creare de noi documente

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.

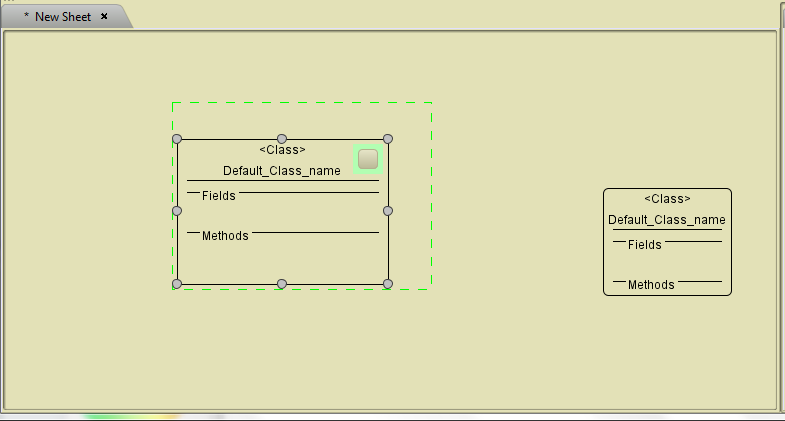


Figura 4.7. Exemplu de comportament de redimensionare



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.

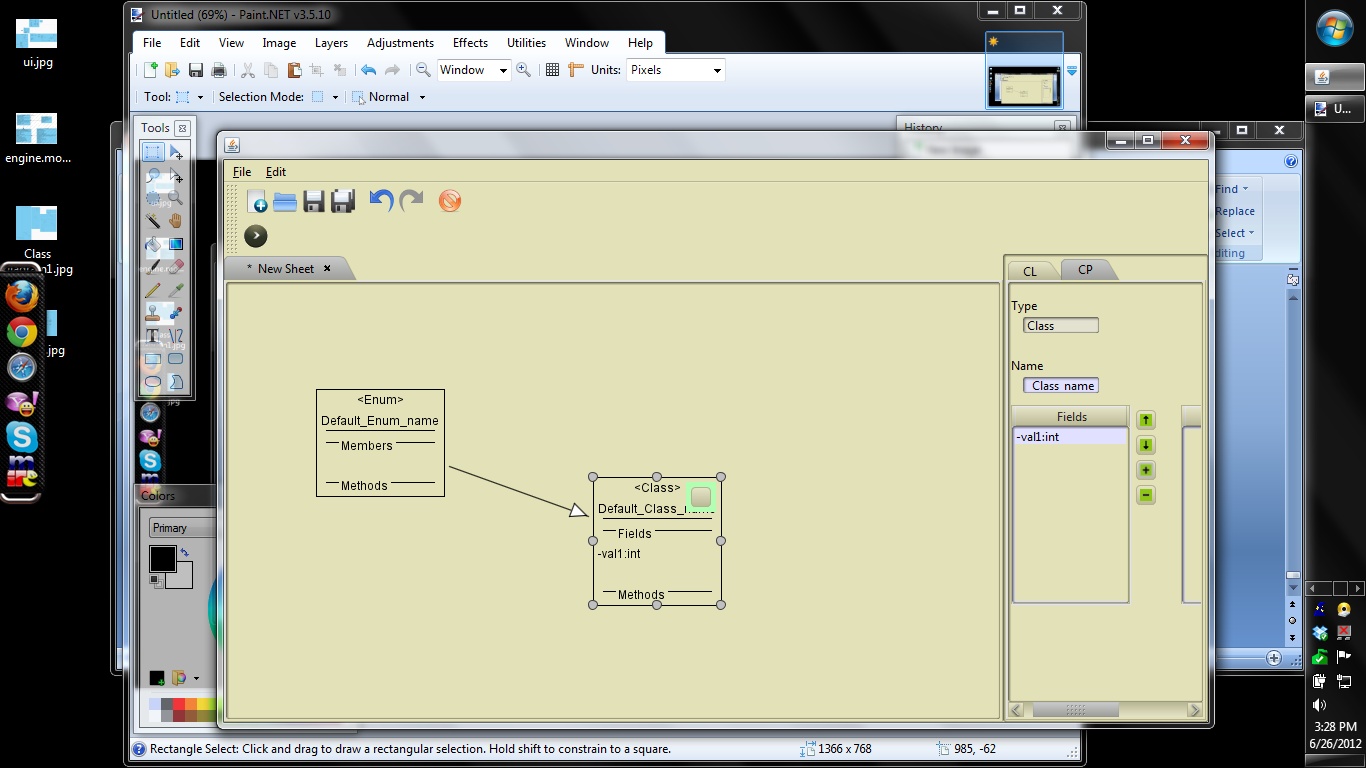


Figura 4.9. Exemplu de meniu dinamic - entitate

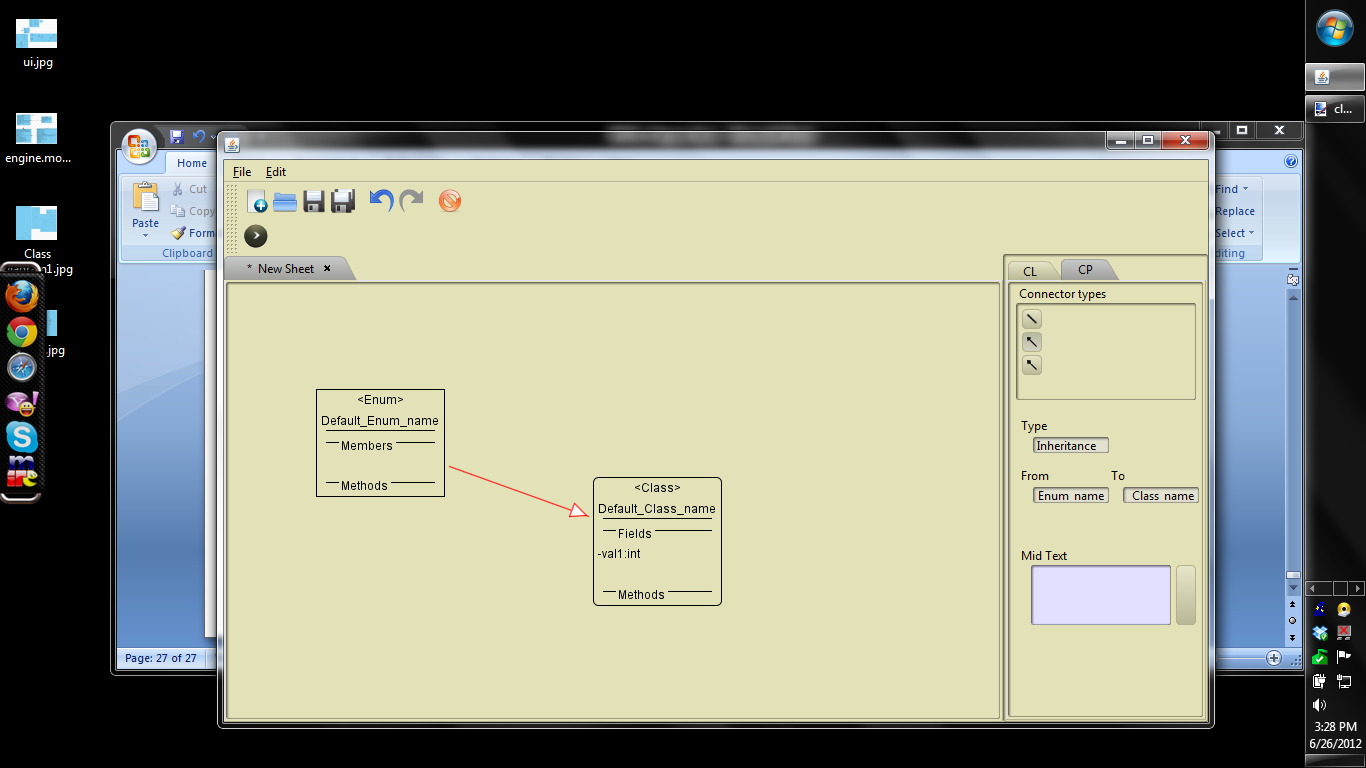


Figura 4.10. Exemplu de meniu dinamic – conector

**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.

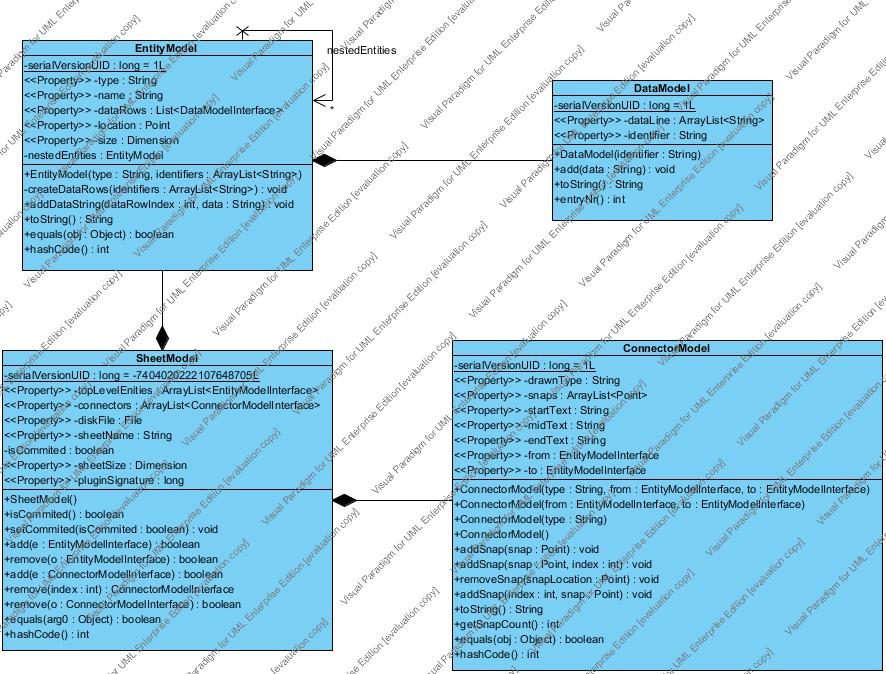


Figura 4.11. Diagrama de clase – date tranzacționale

**Nivelul de 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 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:

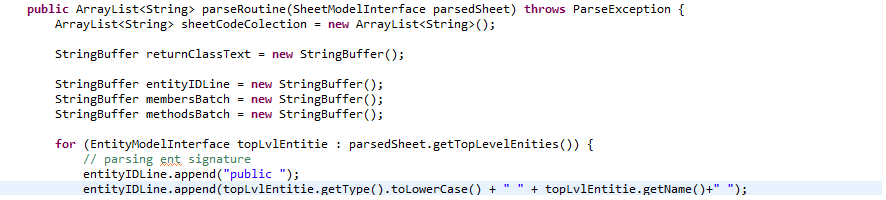


Figura 4.12. Declarația metodei de parsare

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”.

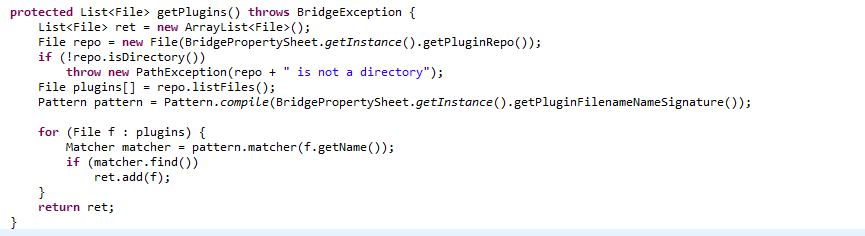


Figura 4.13. Definitia metodei de încarcare a extensiilor

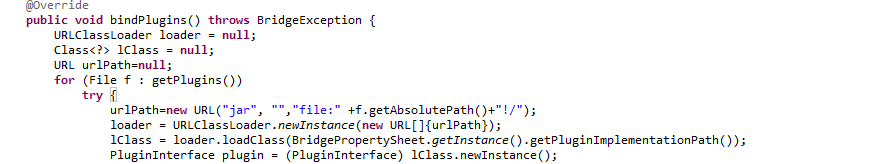


Figura 4.13. Definitia metodei de instanțiere a extensiilor

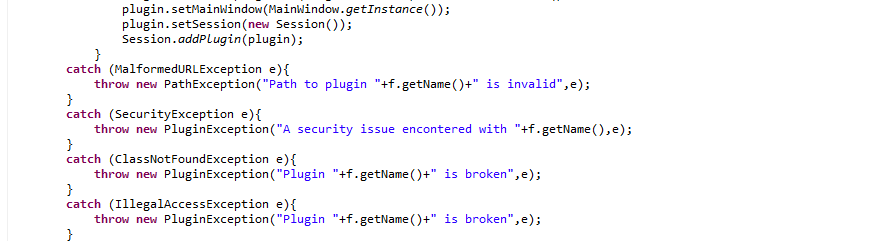


Figura 4.13. Schimbarea statutului punctului de extensie in “extensie curentă”

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.



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.

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.

