O abordare orientată pe componente generice pentru crearea dinamică a interfețelor cu utilizatorul

Frăsinaru Cristian
Facultatea de Informatică Iaşi
General Berthelot 16, IAŞI 700483, ROMANIA
acf@infoiasi.ro

Ungureanu Vlad Costel
Facultatea de Informatică Iaşi
General Berthelot 16, IAŞI 700483, ROMANIA
ungureanu_vlad_costel@yahoo.com

REZUMAT
În acest articol vom descrie o nouă tehnică pentru crearea dinamică a interfețelor cu utilizatorul dedicate prezentării şi editării datelor unei aplicații. Pornind de la specificațiile arhitecturilor bazate pe modele (MDA), propunem o soluție compusă dintr-un cadru de lucru abstract care permite atât integrarea dintre entitățile de la nivelul de persistență şi modelele vizuale dedicate acestora, cât şi implementări multiple folosind diverse tehnologii specifice platformei de programare Java, desktop sau web. Rezultatele concrete obținute în diferite proiecte ce utilizează cadrul nostru de lucru ne îndreptățesc să considerăm că acesta reduce considerabil ciclul de dezvoltare şi asigură o mentenanță mult mai uşoară a aplicațiilor.

Cuvinte cheie
Dezvoltare bazată pe componente (CBD), interfață grafică, arhitectură bazată pe modele (MDA).

Clasificare ACM
H5.2. [User Interfaces: Graphical User Interface (GUI)] D2.3. [Design Tools and Techniques: User Interfaces]

INTRODUCERE

Dezvoltarea de aplicații software este un proces complex, mare consumator de timp. În acest proces sunt rare situațiile în care programatorul este nevoit să se folosească de cunoştințele sale, în cea mai mare parte a timpului fiind ocupat cu sarcini simple şi repetitive care nu aduc valoare sistemului software. Acest proces este îngreunat de faptul că specificațiile se schimbă constant, fiind nevoie de modificări repetate care duc pe termen lung la degradarea sistemului.

Analiza şi proiectarea orientate obiect (OOA&D) încearcă să prevină sau să înlesnească acest proces. Utilizarea UML ("*Unified Modelling Language"*) pentru specificații, proiectare, implementare, testare şi "*deployment*" (specifică OOA&D) se poate dovedi utilă la începtul procesului de dezvoltare, dar devine ineficientă în stadiile avansate din cauza schimbării rapide a specificațiilor şi tehnologiilor folosite. Reintegrarea de tehnologii noi într-o aplicație deja existentă este o activitate complexă şi predispusă apariției erorilor.

Din punctul de vedere al productivității, în afara muncii repetitive, procesul de dezvoltare al sistemelor soft se confruntă cu schimbarea constantă a specificațiilor şi integrarea de module noi în aplicație. Astfel, inginerii software trebuie să modifice diagramele UML, iar programatorii trebuie să înțeleagă noile modificări şi să le implementeze. Astfel, în stadiile avansate ale proiectului productivitatea este afectată de repetiția acestor sarcini şi de faptul că noile modificări pot declanşa în cascadă schimbări şi în celelalte module.

Soluția

În aceste condiții, este nevoie de o nouă abordare pentru a face față complexității ridicate a procesului de dezvoltare a sistemelor soft. O astfel de abordare a fost propusă în 2001 de "*Object Management Group*" (OMG): paradigma arhitecturii bazată pe modele (MDA - *Model Driven Architecture*) [6].

Unul din scopurile principale ale MDA este separarea design-ului de arhitectură, tehnologiile pentru acesta evoluând asincron, permițând astfel dezvoltatorilor soft să aleagă acele tehnologii care se pretează cel mai bine necesităților sistemului soft.

Arhitectura bazată pe modele poată fi descrisă de patru principii [5]:

- modelele trebuie să fie exprimate prin notații bine definite, facilitând înțelegerea sistemelor soft de dimensiuni considerabile;
- dezvoltarea de aplicații poate fi structurată în jurul unui set de modele pe baza unei serii de transformări într-o architectură formată din niveluri ("*layers*") şi cadre de lucru ("*frameworks*");
- metamodelele sunt folosite pentru a descrie modele, facilitând transformările între modele şi procesul de automatizare al instrumentelor MDA;
- abordarea bazată pe modele ar trebui să respecte reguli bine definite, asigurând independența de alte cadre de lucru MDA.

Pentru a sprijini aceste principii OMG a definit o serie de niveluri şi transformări care oferă un cadru de lucru conceptual şi un "vocabular" generic pentru MDA.

OMG identifică patru tipuri de modele: "*computation independent model* (CIM)", "*platform independent model* (PIM)", "*platform specific model* (PSM)" şi "*implementation specific model* (ISM)". Într-un sens strict, MDA se referă la modele ca instanțe ale metamodelului "*Meta-Object Facility*" (MOF)[1,9] fiind astfel formate din modele şi legăturile dintre ele.

Modelele trebuie să fie independente de limbajul de programare, sistemul de operare şi detaliile specifice tehnologiilor folosite, în esență ar trebui să fie reprezentate cât mai abstract posibil. Deşi modelele pot avea reprezentări variate (UML, clase, XML), MDA propune utilizarea UML pentru transformări *model – la – text* sau *model – la – model*, încercând astfel facilitarea utilizării modelelor ca resurse în dezvoltarea de aplicații [6].

Utilizarea MDA a luat amploare în ultimii ani, pe baza specificaţiilor sale fiind create o varietate de cadre de lucru şi instrumente software, fiecare cu avantajele şi dezavantajele lor.

ABORDAREA GENERALĂ

Implementările MDA aduc propriile contribuţii şi viziuni asupra procesului de dezvoltare de aplicaţii, păstrând totuşi o legatură puternică cu abordarea generală a arhitecturii bazate pe modele.

O implementare MDA generică porneşte de la reprezentarea modelului. Deoarece definirea unui model este deschisă la intrepretări, acesta poate fi descris în mai multe moduri:

- **Diagrame UML** – UML este cea mai utilizată metodă de reprezentare a modelelor, fiind considerat standard şi bucurându-se de o utilizare puternică. Dezavantajul utilizării UML constă în faptul că modelul se schimbă constant în timpul procesului de dezvoltare, generând modificări în cascadă asupra a numeroase diagrame. Un alt inconvenient al utilizarii UML este formatul interschimbabil XMI, acesta fiind uneori dificil de procesat de unele unelte MDA [10].
- **Documente XML** - Fiind un format independent de platforma şi limbajul de programare, XML oferă o modalitate facilă de reprezentare a modelului. Modificările aduse modelului nu sunt mari consumatoare de timp şi sunt reduse din punctul de vedere al complexităţii. Totuşi, XML nu oferă o viziune de ansamblu a arhitecturii şi nici o reprezentare grafică, făcând astfel modelele complexe greu de inţeles şi de urmărit. XML nu presupune un standard comun, astfel uneltele care folosesc acest tip de reprezentare presupun o structură fixă, uneori rigidă şi incompletă.
- **Scheme SQL** - Schemele SQL pentru bazele de date oferă de asemenea o reprezentare simplă a modelului. În acest caz există o corespondenţă bijectivă între relaţiile bazei de date şi entităţi. Această reprezentare se potriveşte aplicaţiilor care aderă la şablonul arhitectural "*Naked Objects*" [13]. În cazul în care entităţile trebuie să conţină câmpuri fără corespundent în coloanele unui tabel, această reprezentare devine ineficientă.
- **Entităţi** - Entităţile sunt instanţe adnotate ale unor clase ce descriu datele gestionate de aplicaţie, fiind cea mai simplă modalitate de descriere a modelului. Avantajul principal este reprezentat de faptul că entităţile pot fi înţelese cu uşurintă de orice programator indiferent de experinţa în domeniu. Similar cu schmele SQL şi formatul XML, entităţile nu oferă însă o viziune de ansamblu [3,10,11].

După definirea modelului într-un format corespunzător urmează o altă fază a implemntării MDA: *transformările*. Pentru aceasta este necesar un cadru de lucru MDA. Putem defini un cadru de lucru pentru transformări MDA ca fiind o unealtă care poate să înţeleagă modelul, îl poate interpreta şi poate aplica o serie de transformări peste acesta [6].

De obicei, aceste cadre de lucru folosesc un mod de reprezentare specific şi pot efectua un set limitat de transformări. Spre exemplu, un cadru de lucru va folosi doar modele UML prin formatul intermediar XMI, va intepreta doar diagramele de clasă şi la final va aplica doar acele transformări necesare pentru generarea de clase pe baza modelului [9,10].

Un ultim element al unei implementări MDA îl reprezintă *artefactele MDA* (fişiere obţinute pe baza modelului, care pot fi folosite în cadrul unei aplicaţii). Considerând celelalte aspecte ale unei implementări MDA putem spune că artefactele sunt rezultatul final al prelucrării modelului de către un cadru de lucru. În funcţie de modelul oferit şi de capacităţile cadrului de lucru, artefactele pot fi de mai multe tipuri dintre care cele mai uzuale sunt: entităţi, scheme SQL, unităţi de persistenţă, fişiere de configurare şi interfaţa grafică cu utilizatorul [1,3,9,10,11].

Astfel, o implementare MDA trebuie să ofere următoarele funcţionalităţi:

- prelucrarea modelului pentru a fi cât mai simplu de înţeles şi utilizat;
- procesarea modelului şi aplicarea de transformări peste acesta;
- obţinerea de artefacte utilizabile în cadrul unei aplicaţii.

Generarea statică

Cea mai uzuală implementare a specificaţiilor MDA se referă la generarea statică de artefacte. Majoritatea cadrelor de lucru adoptă această aboradare datorită productivităţii crescute, cel puţin în stadiile de început ale procesului de dezvoltare.

Scopul principal al acestui tip de implementare este generarea unei porţiuni cât mai mari din aplicaţie pe baza modelului oferit.

Acest gen de implementări poartă denumirea de *generatoare* deoarece rolul lor constă în aplicarea de transformări peste model şi obţinerea de artefacte. Intepretarea modelulului nu este un element defintoriu pentru aceste cadre de lucru, deşi majoritatea oferă şi o implementare pentru această funcţionalitate.

Cele mai frecvente metode de reprezentare a modelului sunt diagramele UML, schemele SQL şi entităţile. Cele mai frecvente artefacte generate sunt:

- *Entităţile* - În unele cazuri entităţile au asociate şi clase de tip DAO ("*Data Access Objects*") generate automat [1,10,11].
- *Schemele SQL* pentru diverse tipuri de baze de date [3,10].
- *Unităţile de persistenţă* sunt mai rar întalnite, dar reprezintă un tip de artefacte foarte important. Unele implementări vor genera şi artefacte adiţionale cum ar fi fişiere de configurare sau de mapare între modelul relaţional şi obiectual.

Cadrele de lucru avansate oferă implementări şi pentru un manager de persistență [3,10,11,17].

- *Interfață grafică cu utilizatorul* (GUI). Majoritatea implementărilor generează fişiere GUI pentru aplicații web, specifice unor anumite tehnologii (JSP, ASP, PHP, etc.). Există şi unele implementări care oferă soluții de creare a interfeței cu utilizatorul pentru aplicații desktop, cum ar fi JMatter [11] sau NakedObjects [13]. Implementările avansate oferă integrarea cu tehnologii de tip RIA ("*Rich Internet Application*") bazate pe AJAX, cum ar fi IceFaces sau RichFaces [3,6]. Funcționalitatea oferită la nivelul GUI se referă la operațiile de manipulare a datelor gestionate de aplicație, cum ar fi cele de creare, citire, modificare şi stergere (CRUD).

Aceasta abordare este cel mai eficient folosită în concordanță cu utilizarea diagramelor UML, deoarece inginerii software vor crea diagramele înainte de implementarea propriu-zisă, dezvoltatorii beneficiind din plin de artefactele generate direct din model. Generarea de entități şi a nivelului de persistență oferă un progress semnificativ încă de la începutul dezvoltării aplicației, fişierele de configurare generate automat asigură integrarea corectă a diverselor tehnologii iar generarea schemei SQL asigură şi crearea automată a bazei de date.

Un avantaj major îl consituie generarea interfeței grafice pentru operațiile CRUD. Majoritatea aplicațiilor trebuie să ofere un număr mare de componente GUI care să cuprindă toată funcționalitatea necesară acestui nivel. Având în vedere că logica complexă specifică aplicației trebuie decuplată de aspectele legate de reprezentarea vizuală iar aceasta din urmă trebuie să ofere o imagine şi un comportament unitar pentru întreaga aplicație, se obține un câştig semnificativ prin generarea automată pe baza unui model a componentelor ce țin de prezentare şi integrarea ulterioară a acestora cu aspectele ce țin de logica aplicației.

Dezavantajele acestei abordări devin clare în stadiile avansate ale dezvoltării, când modelul se schimbă constant pentru a se adapta la noile cerințe iar regenerarea de artefacte ar duce la pierderea metodelor implementate de programator şi la pierderea de date membre care nu erau incluse în model. De asemenea, în stadiile avansate de dezvoltare fişierele GUI sunt modificate din ce în ce mai des pentru a adăuga noi functionalități sau chiar pentru a face interfața grafică mai uşor de folosit. În condițiile în care generarea statică a interfeței grafice se face într-un mod standard şi fişierele GUI sunt modificate constant, generarea statică devine foarte nepractică.

Generarea dinamică

O altă abordare pentru implementările MDA o constituie generarea dinamică a artefactelor. În locul generării statice la începutul procesului de dezvoltare şi repetarea (după posibilități) a generării cu fiecare modificare a modelului, abordarea dinamică propune generarea unor anumite artefacte în momentul execuției. Deşi entitățile, schemele SQL şi unitățile de persistență sunt mai eficient de generat la început, componentele corespunzătoare interfeței grafice se pretează perfect pentru această abordare.

Entitățile, schemele SQL şi unitățile de persistență suferă schimbări numeroase la început dar tind să se stabilizeze pe parcusrul procesului de dezvoltare. Fişierele UI şi logica asociată tind să se modifice pe tot parcusul dezvoltării aplicației fie pentru a surprinde modificările entităților, dar în general pentru a se preta mai bine la necesitățile utilizatorului.

Similar cu generarea statică, această abordare se pretează pentru operațiile CRUD, lăsând logica de complexitate ridicată şi fişierele GUI associate pe seama programatorului. Din punctul de vedere al implementării MDA generarea dinamică poate fi definită astfel:

- modelul este reprezentat de entități. Deoarece entitățile pot conține câmpuri care nu trebuie să apară în interfață, necesitatea unei metode de "mapare" pentru a specifica generatorului ce câmpuri trebuie incluse în GUI este evidentă.
- generatorul propriu-zis este inclus în componente grafice specializate. Acestea includ şi o parte din logica aplicației pentru a asigura operațiile CRUD.
- implementarea trebuie să fie cât mai abstractă posibil pentru a se putea preta la orice tip de interfață grafică.

Astfel, generarea dinamică presupune existența unor componente specializate, incluse în aplicație, care vor genera la momentul rulării, pe baza modelului oferit, diferite părți ale interfeței grafice.

Chiar dacă dezvoltarea aplicației nu beneficiază de avansul câştigat la început folosind generarea statică (deşi amblele tipuri de abordări pot fi folosite cu success pentru aceeaşi aplicație), generarea dinamică a GUI aduce beneficii pe tot parcursul procesului de dezvoltare. Inițial, acest câştig de performanță este evidențiat de faptul că pentru orice entitate a aplicației se poate genera GUI pentru operațiile CRUD, câştig atât la nivelul de dezvoltare cât şi la nivelul testării funcționale. În fazele avansate ale procesului de dezvoltare avantajul major îl constitue faptul că orice schimbare adusă modului de prezentare a datelor se face prin modificarea într-un singur loc pentru toată aplicația. De asemenea, pentru o entitate pot exista mai multe componente de vizualizare sau editare, în funcție de necesități.

Existența unor componente specializate folosite în toată aplicația asigură continuitate în utilizare şi design. Un avantaj major îl constituie faptul că schimbările din model vor fi reflectate de interfața grafică cu minimum de modificări.

Ca exemplu, putem să considerăm o aplicație pentru care este necesară asigurarea de operații CRUD pentru un număr foarte mare de entități (peste 100 de entități). De asemenea, se doreşte ca unele entități să aibă 2 sau mai multe tipuri de reprezentări sau formulare de introducere a datelor. Prin faptul că definind doar două componente (una de vizualizare/ştergere şi una de creare/editare) se va genera CRUD pentru toate entiățile, generarea dinamică creşte considerabil productivitatea. De asemenea,

utilizarea generării dinamice facilitează migrarea aplicațiilor desktop-web sau web-desktop, fiind necesară doar mutarea claselor care conțin logica.

Funcționalitatea specifică aplicației, cum ar fi acțiunile ce trebuie executate la interacțiunea cu utilizatorul sau regulile de validare, va fi accesată de componentele specializate de la nivelul interfeței grafice printr-un mecanism de conectare declarativ.

Naked Objects

NO("*Naked Objects*") [13] este un șablon arhitectural care presupune generarea dinamică a interfeței grafice pentru orice entitate. Acest șablon stă la baza câtorva implementări de generatoare dinamice pentru aplicații desktop, pe platformele Java și .NET, unul din cele mai reușite proiecte de acest tip fiind JMatter [11].

NO poate fi definit astfel:

- toată logica aplicației este încapsulată în cadrul entităților, care formează așa numitele "domain objects" [2];
- interfața grafică trebuie să fie o reprezentare completă a entităților;
- interfața grafică ar trebui direct generată pe baza definiției entităților.

O implementare a unui generator dimanic de interfețe poate adera la acest șablon doar parțial datorită limitărilor la nivel logic. Astfel, șablonul presupune că orice câmp dintr-o entitate ar trebui să aibă o reprezentare grafică, acest lucru fiind uneori nepotrivit, întrucât pot exista proprietăți redundante, cum ar fi spre exemplu data nașterii și CNP-ul unei persoane. O altă limitare majoră este imposibilitatea de a defini prezentări multiple, orientate pe context, ale datelor încapsulate de un anumit obiect.

ABORDAREA NOASTRĂ

Separarea dintre modelul vizual și tehnologie

Pornind de la tehnicile prezentate anterior, am dezvoltat o arhitectură care să permită definirea abstractă a modelului vizual ce va descrie interfața grafică CRUD precum și cuplarea acestuia cu implementări concrete dezvoltate pe tehnologii consacrate ale platformei de lucru Java SE, cum ar fi *AWT*, *Swing*[14] sau *SWT* sau ale platformei Java EE, cum ar fi *Java Server Faces*[15], *Google Web Toolkit*[16] sau altele.

Implementarea unui generator dinamic aderă la metodologia "*Component-Based Developemnt*" (CBD), scopul fiind acela de a oferi un set de componente grafice specializate și reutilizabile în diverse aplicații. Aceste componente oferă o reprezentare grafică a funcționalității specifice nivelului de gestiune a datelor, definind din punct de vedere arhitectural un nivel standardizat, orientat vizual, situat deasupra nivelului de persistență.

Un element important îl constituie independența de tehnologia utilizată precum și posibilitatea de a genera, pornind de la specificții generice, atât interfața grafică pentru aplicații desktop cât și pentru aplicații web. Componentele vizate de implementare sunt cele de prezentare a entităților precum și cele de editare a acestora.

Un cadru de lucru descriptiv

Pornind de la șablonul de proiectare "*Naked Objects*", putem considera o abordare practică dezvoltarea unei arhitecturi care să adauge nivelului "*domain objects*"[2], un cadru de lucru descriptiv, orientat pe prezentare. Prin aceasta se realiză o *adnotare extinsă* a entităților aplicației, superioară adnotării directe a proprietăților obiectelor întâlnită în diverse framework-uri web, cum ar fi JPA2web [12]. Tehnica noastră va permite polimorfism în modul de interpretare a datelor, din punct de vedere vizual, de către setul de componente.

Astfel, interfața grafică generată va putea fi personalizată în funcție de contextul în care sunt prezentate datele. Spre exemplu, într-o aplicație dedicată gestiunii activității unui hotel, pentru o entitate care reprezintă un client se poate dori o anumită reprezentare a informației la momentul sosirii acestuia și o altă reprezentare la momentul plecării, fiecare având editoare specializate pentru contextul specific. Pe lângă acestea, este necesară și reprezentarea completă a datelor, accesibilă în cadrul nomenclatoarelor aplicației.

ARHITECTURA SISTEMULUI

Dezvoltarea unei interfațe de programare (API) este o abordare practică mult mai avantajoasă decât dezvoltarea unui instrument propriu-zis, oferind flexibilitate și extensibilitate. Astfel, implementarea unui generator dinamic este decuplată de tehnologiile folosite.

Arhitectura API-ului dezvolat de noi are la bază șablonul architectural *Naked Objects* [13] și șabloanele de proiectare *Observer* și *Factory Method* [4].

Similar cu abordările existente deja, API-ul nostru presupune folosirea entităților ca model. Acestora le este atașat un mecanism care să le asocieze cu un conector către sistemul de gestiune a datelor (*DataManager*) precum și cu modalitățile de vizualizare (*DataView*) și editare (*ItemEditor*).

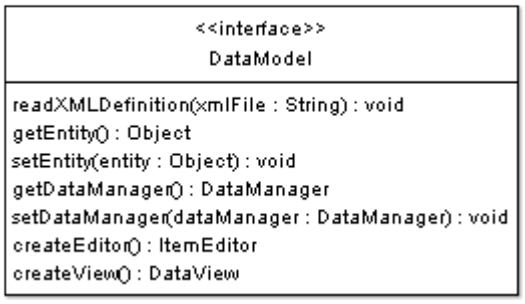

Figura 1. Diagrama de clasă pentru DataModel

Interfața *DataModel* are asociat un nume de clasă ce descrie o anumită entitate și un fișier de mapare XML responsabil cu definirea din punct de vedere vizual a prezentării datelor acesteia. Implementarea *DataModel* va interpreta fișierul XML și pe baza acestuia va crea componente de vizualizare și de editare pentru entitatea asociată. Componentele de vizualizare și editare sunt definite la nivel abstract, fiind independente de tehnologia de reprezentare.

Accesul la date se face printr-un obiect abstract, descris de abstracțiunea *EntityManager*. Obiectele acestei clase vor fi create în mod eficient utilizând o "fabrică de obiecte", specifică tehnologiei de persistență cu care se lucrează.

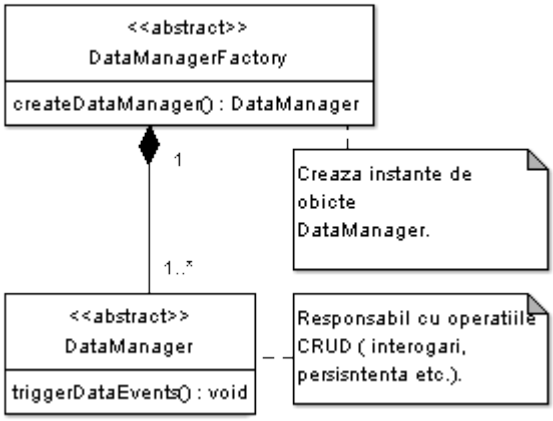

Figura 2. Diagrama de clasă pentru DataManager

Implementările clasei abstracte *DataManagerFactory* aderă la șablonul de proiectare *Factory Method* și au ca scop crearea de instanțe pentru implementările clasei abstracte *DataManager*. Obiectele de tip *Factory* sunt obiecte complexe ("*Heavy*") iar obiectele de tip Manager sunt obiecte de complexitate redusă ("*Light*"), astfel un *DataModel* va avea asociat un obiect de complexitate scăzută.

DataManager este o abstractizare peste nivelul de persistență, orice implementare concretă a acesteia asigurând suportul pentru executarea de interogări și operații specifiece persistenței datelor. Acestea vor fi făcute în funcție de tehnologiile folosite, câteva posibile implementări fiind:

- *EntityManager*: abstractizare peste tehnologii ce aderă la specificațiile JPA, cum ar fi *Hibernate* [17] sau *TopLink*, ce permit interogări orientate obiect utilizând limbaje specializate și persistența entităților pe baza unor reguli de mapare.
- *RelationalManager*: execută interogări SQL și se bazează pe tehnologii clasice de accesare a bazelor de date relaționale, cum ar fi JDBC.
- *ObjectManager*: pentru sisteme de gestiune a datelor pur orientate obiect.
- *TextManager*: pentru date reprezentate în fișiere text, într-un format standard, de exemplu CSV, etc.

Similar, trebuie oferite implementări specifice și pentru clasele *Factory*.

În acest fel, modalitatea de descriere a unei entități din punct de vedere vizual este complet decuplată de modalitatea de persistență a acesteia.

Componentele de vizualizare *DataView* sunt responsabile cu prezentarea datelor pe baza descrierilor din fișierul de mapare XML.

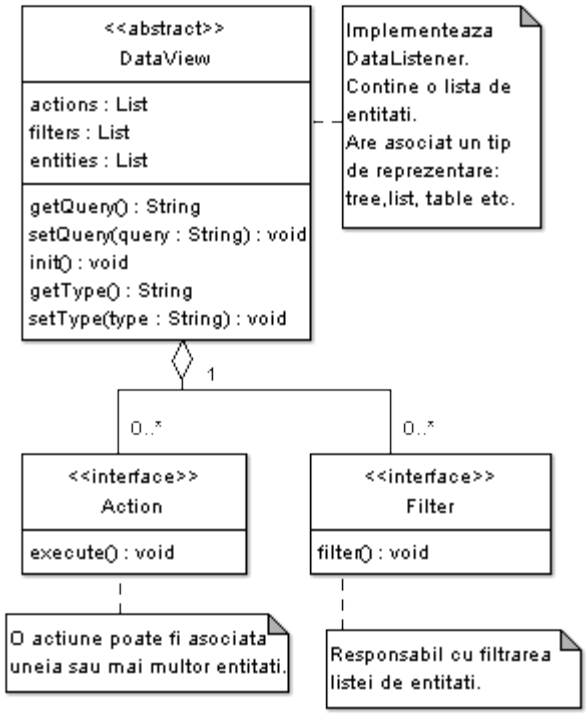

Figura 3. Diagrama de clasa pentru DataView

Implementarea *DataView* este un container de date, acțiuni și filtre.

Pentru reprezentarea datelor va fi necesară specificarea unui tip de reprezentare, acesta putând varia în funcție de necesitate (arbore, listă, table, etc.).

Filtrele vor fi folosite pentru a eficientiza maniera de prezentare, atât din punctul de vedere al vitezei de execuție cât și al volumului de date prezentat utilizatorului. Ele vor fi definite în funcție de tehnologia aleasă pentru persistență, fie prin criterii orientate obiect fie prin fragmente SQL.

Deoarece componenta de vizualizare conține o mulțime de entități, este normal ca asupra ei să poate fi realizate acțiuni cum ar fi adăugarea, modificarea, ștergerea, tipărirea, căutarea, etc. Astfel, componenta va avea asociată și o listă de acțiuni.

Atât acțiunile cât și filtrele sunt externe clasei de tip *DataView*, ele putând fi folosite de alte reprezentări de entități cât și de alte implementări.

DataView trebuie să implementeze interfața *DataListener* ce aderă la șablonul de proiectare *Observer* [12]. Atunci când se vor executa acțiuni de inserare, ștergere sau modificare vor fi generate evenimente, la care componenta de vizualizare va raspunde prin executarea de acțiuni, cum ar fi actualizarea prezentării datelor.

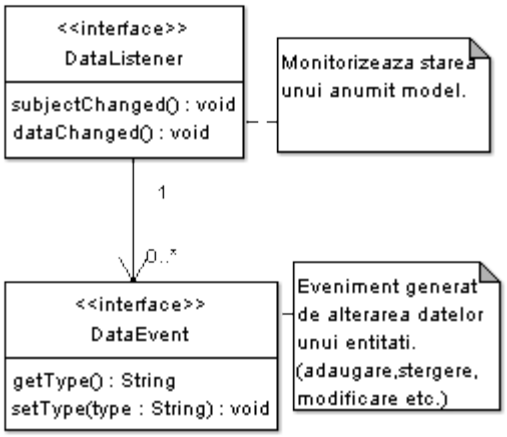

Figura 4. Diagrama de clasă pentru DataListener

DataView implementează interfaţa *DataListener* astfel încât va fi notificată de modificările care au loc în componentele de editare.

Componentele de tip *ItemEditor* sunt responsabile cu generare de interfaţă grafică pentru crearea şi editarea entităţilor.

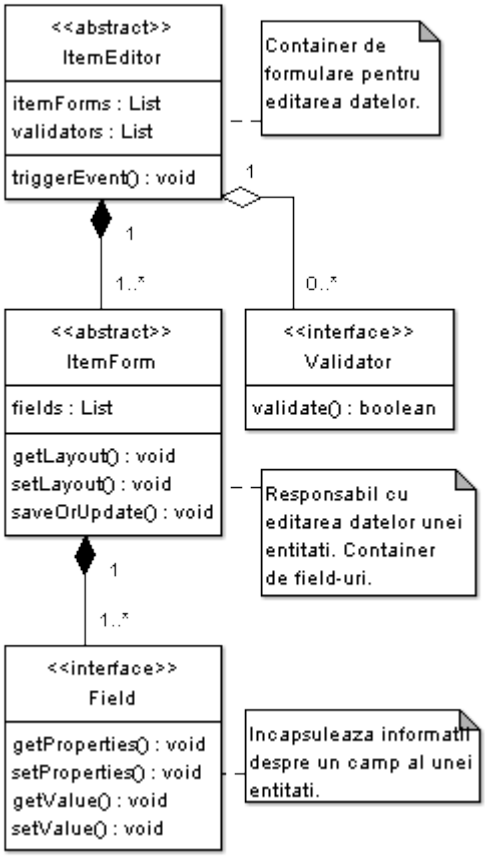

Figura 5. Diagrama de clasă pentru ItemEditor

ItemEditor serveşte drept container pentru formulare. Entitatea poate avea asociate una sau mai multe modalităţi de editare sau creare diferenţiate prin câmpurile care vor fi incluse în GUI. De asemenea, această clasă are asociată şi o listă de validatori. Un validator este responsabil cu verificarea corectitudinii datelor unuia sau mai multor câmpuri dintr-o entitate, fiind acelaşi pentru toate formulare. Obiectele de tip *ItemEditor* sunt generatoare de evenimente, notificând componeta de vizualizare că s-au efectuat anumite operaţii asupra datelor, urmând apoi ca la nivelul vizualizării să se execute acţiunile necesare.

5. ADNOTAREA CU FIŞIERE XML

Interpretarea entităţilor de către componente se face pe baza unui fişier de configurare XML. Acesta va conţine toate informaţiile necesare pentu generarea reprezentării grafice. Pentru o entitate pot exista mai multe componente de vizualizare care vor folosi fişiere de adnotare diferite.

Să descriem informal, printr-un exemplu simplu, structura propusă pentru fişierul XML. Să presupunem că se doreşte crearea unui model pentru reprezentarea unor persoane, având proprietăţile *nume*, *dataNasteri*, *localitate* şi *observaţii*. Documentul de adnotare (parţial) ar putea fi:

```
<data>
  <model name="modelPersoana”>
    <entity>Persoana</entity>
    <manager>EntityManager</manager>
  </model>
  <view name="view1" type="list">
    <query syntax=”hql”>
        select nume from Persoana
    </query>
  </view>
  <view name="view2" type="grid">
    <query syntax=”sql”>
        select a.nume,a.dataNasterii,b.nume from
        persoane a join localitati b on
        a.localitateId=b.id
    </query>
    <column name="nume" header=”Nume Persoana”/>
    <column name="dataNasterii" align=”center”/>
    <column name="localitate"/>
    <filter class="LocalitateFilter"
    column="b.id"/>
  </view>
  <editor name="editor1" layout="tabbed">
    <form name="dateGenerale" layout="flow">
        <field type="text" label="Nume"
        name="nume" required="true"/>
        <field type="date" label="Data nasterii"
        name="dataNasterii"/>
        <field type="query" label="Localitate"
        name="localitate"
        sourceRef=”modelLocalitate”/>
    </form>
    <form name="obs" layout="single">
        <field type="textArea" label="Observatii"
        name="obseratii"/>
    </form>
  <editor/>
  <editor name="">
        . . .
  <editor/>
</data>
```

Fişierul conţine trei secţiuni. Secţiunea “model” va conţine numele complet al entităţii şi numele complet al

implementării pentru tipul de *DataManger* care va fi folosit.

Secțiunea "*view*" conține numele vizualizării și tipul acesteia. De asemenea conține interogarea prin care vor fi aduse entitățile din sursa de date, coloanele care vor fi reprezentate din mulțimea de coloane returnate de interogare și ce filtre vor fi aplicate asupra datelor.

Secțiunea "editor" conține numele și aranajarea în pagină a editorului, numele și aranjarea formularelor și câmpurile care vor fi incluse în formularele de editare.

Pentru o mai bună înțelegere a structurii fișierului de mapare îl putem descrie prin fișierul DTD asociat (descrierea este parțială):

```
<!ELEMENT data(model,view+,editor) ANY>

<!ELEMENT model(entity,manager) ANY>
<!ELEMENT entity (#PCDATA)>
<!ELEMENT manager (#PCDATA)>

<!ATTLIST model name CDATA #REQUIRED>

<!ELEMENT view(query,column+,filter*) ANY>
<!ELEMENT query (#PCDATA)>
<!ELEMENT column (#PCDATA)>
<!ELEMENT filter (#PCDATA)>

<!ATTLIST view type CDATA #REQUIRED>
<!ATTLIST query sintax CDATA #REQUIRED>
<!ATTLIST column name CDATA #REQUIRED>
<!ATTLIST column label CDATA #IMPLIED>
<!ATTLIST editor renderer CDATA #IMPLIED>
<!ATTLIST filter class CDATA #REQUIRED>
<!ATTLIST filter column CDATA #REQUIRED>

<!ELEMENT editor(form+) ANY>
<!ELEMENT form(field+) ANY>
<!ELEMENT field(fieldquery*) ANY>
<!ELEMENT fieldquery (#PCDATA)>

<!ATTLIST editor name CDATA #REQUIRED>
<!ATTLIST editor layout CDATA #IMPLIED>
<!ATTLIST form name CDATA #REQUIRED>
<!ATTLIST form layout CDATA #IMPLIED>
<!ATTLIST field label CDATA #REQUIRED>
<!ATTLIST field name CDATA #REQUIRED>
<!ATTLIST field type CDATA #REQUIRED>
<!ATTLIST field sourceRef CDATA #IMPLIED>
```

Fișierul de mapare astfel definit presupune o intrepretare liberă pentru unele atribute cum ar fi "*type*" sau "*layout*", interpretarea variind în funcție de implementare, fiind posibilă chiar absența lor sau ignorarea lor în cazul în care fișierele de mapare trec de la o tehnologie la alta.

Pentru a trata relațiile între entități (reprezentate la nivelul bazei de date prin chei secundare) unei coloane îi poate fi asociată o interogare care să aducă entițăle necesare, în acest caz fiind necesară și specificarea entității către care există o relație.

6. CONCLUZII ŞI DIRECȚII DE VIITOR

Indiferent de modul în care este realizată, generarea automată a interfețelor grafice ale unei aplicații crește productivitatea în procesul de dezvoltare a aplicațiilor software, eliminând munca repetitivă și de complexitate redusă.

Deşi populară, mai ales în cazul aplicațiilor web, generarea statică impune standarde și convenții greu de înlocuit, făcând dificilă modificarea artefactelor generate, mai ales în stadiile avansate ale procesului de dezvoltare [8]. De multe ori, generarea statică este strâns dependentă de instrumentele de reprezentare a modelului, de convențiile impuse de acestea, de tehnologiile folosite și de platforma pentru care se generează artefactele [8].

Generarea dinamică nu presupune restricții sau convenții fixe, oferind flexibilitate în privința modului de generare a interfeței cu utilizatorul, orice modificare făcută la nivelul modelului fiind reflectată imediat în toate componentele de la nivelul de prezentare. Un alt avantaj constă în faptul că este creată în mod natural o decuplare între nivelul de vizualizare și cel al funcționalității propriu-zie a aplicației.

Inconvenientul acestei soluții este însă necesitatea existenței unui cadru de lucru bine pus la punct cu ajutorul căruia să poată fi integrate aspectele legate de model, persistență și vizualizare, respectiv editare.

In această lucrare am definit o arhitectură abstractă, uşor de utilizat, pe baza careia pot fi dezvoltate în mod independent componente specializate responsabile cu generarea dinamică a interfeței grafice cu ajutorul oricărei tehnologii dedicate, făcând astfel posibilă migrarea logicii de la nivelul vizual de pe o platformă de lucru pe alta fără efort suplimentar.

Modul extins de adnotare a entităților pentru intepretarea lor de către componentele de prezentare sau editare elimină incovenințele șablonului *Naked Objects*, asigurând posibilitatea existenței mai multor reprezentări diferite pentru aceeaşi entitate, lucru extrem de util în majoritatea aplicațiilor în care aspectele legate de manipularea datelor ocupă un loc important.

Abordarea noastră reduce semnificativ dimensiunile codului specific unei anumite tehnologii sau platforme de lucru prin utilizarea unei soluții descriptive bazată pe limnajul XML de definire a modurilor de vizualizare şi editare a datelor. Spre exemplu, o pagină JSP care conține un tabel cu o listă de entități poate ajunge și la 100 de linii de cod, pe când utilizarea unei componente necesită doar o singură linie de cod (utilizarea *tag*-ului componentei).

Schimbarea *design*-ului se face într-un singur loc și este valabilă în toate componentele, indiferent de modalitatea prin care se face reprezentarea (*renderer*, HTML și CSS, compunere de componente – *composite pattern*, etc.).

Soluția descrisă de noi a fost implementată cu succes pentru mai multe aplicații desktop comerciale și open-source, concluzia noastră fiind, că nu este o exagerare să

afirmăm, întocmai ca autorii JMatter [11] că "generea dinamică poate creşte productivitatea de până la 10 ori".

La ora actuală există o implementare aflată într-un stadiu avansat de dezvoltare pentru tehnologia *Swing* şi se lucrează la o implementare pentru cadrul de lucru JSF ("*Java Server Faces*").

In cazul JSF, pentru a eficientiza modul de interacțiune al utilizatorului cu aceste componete specializate se urmăreşte integrarea lor cu un cadru de lucru Ajax, eliminând necesitatea ciclurilor repetitive "cerere-răspuns".

De asemenea, avem în vedere şi o implementare pentru GWT ("*Google Web Toolkit"*) care, deşi dedicată aplicațiilor web, poate fi abordată similar cu implementarea desktop creată pentru Swing.

Din punct de vedere architectural, vom continua dezvoltarea mecanismelor pentru lucrul cu acțiuni şi evenimente, astfel încât şi implementările concrete să beneficieze de o infrastructură comună cât mai complexă.

7. BIBLIOGRAFIE

1. "MDA Explained. The model driven architecture: Practice and Promise", Anneke Kleppe, Jas Warmer şi Wim Bast, Addison-Wesley 2003
2. "Agile Modeling: Effective Practices for EXtreme Programming and the Unified Process", Scott W. Ambler, J. Wiley 2002
3. "Seam in Action", Dan Allen, Manning 2008
4. "The design patterns Java companion", James W. Copper, Addison-Wesley 1998
5. "An introduction to Model Driven Architecture" , Alan Brown, IBM http://www.ibm.com/developerworks/rational/library/3100.html (consultat în 2009)
6. "Model Driven Architecture overcomes limits of traditional object modeling", Eric Linch http://www.devx.com/architect/Article/36130/1954 (consultat în 2009)
7. "Understanding Model Driven Architecture", Sinan Si Alhir http://www.methodsandtools.com/archive/archive.php?id=5 (consultat în 2009)
8. "MDA from a developers perspective", Stefan Tilkov http://www.theserverside.com/tt/articles/article.tss?l=MDA (consultat în 2009)
9. MDA Specifications, www.omg.org/mda/specs.htm#MDAGuide (consultat în 2008-2009)
10. AndroMDA, www.andromda.org (consultat în 2008-2009)
11. JMatter, www.jmatter.org (consultat în 2008-2009)
12. jpa2web, http://jpa2web.sourceforge.net (consultat în 2008-2009)
13. NakedObjects, www.nakedobjects.org (consultat în 2008-2009)
14. Swing, http://java.sun.com/docs/books/tutorial/uiswing/ (consultat în 2008-2009)
15. Java Server Faces, http://java.sun.com/javaee/javaserverfaces/ (consultat în 2008-2009)
16. Google Web Toolkit, http://code.google.com/webtoolkit/ (consultat în 2008-2009)
17. Hibernate, http://www.hibernate.org/ (consultat în 2008-2009)