MINISTERUL EDUCAȚIEI, CERCETĂRII, TINERETULUI ȘI SPORTULUI

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

FACULTATEA DE AUTOMATICĂ ŞI CALCULATOARE
CATEDRA CALCULATOARE

DEZVOLTAREA APLICAȚIILOR WEB UTILIZÂND MODEL DRIVEN ARCHITECTURE

LUCRARE DE LICENȚĂ

Absolvent: **Cristian Andrei STAN**

Coordonator ştiinţific: **Șef lucrări ing. Cosmina IVAN**

2011

MINISTERUL EDUCAȚIEI, CERCETĂRII, TINERETULUI ŞI SPORTULUI

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

FACULTATEA DE AUTOMATICĂ ŞI CALCULATOARE
CATEDRA CALCULATOARE

Declaraţie

Subsemnatul *Cristian Andrei STAN*, student al Facultăţii de Automatică şi Calculatoare, Universitatea Tehnică din Cluj-Napoca, declar că ideile, analiza, proiectarea, implementarea, rezultatele şi concluziile cuprinse în această lucrare de licenţă constituie efortul meu propriu, mai puţin acele elemente ce nu îmi aparţin, pe care le indic şi recunosc ca atare.

Declar de asemenea că, după ştiinţa mea, lucrarea în această formă este originală şi nu a mai fost niciodată prezentată sau depusă în alte locuri sau alte instituţii decât cele indicate în mod expres de mine.

Data: 24 Iunie 2011

Absolvent: **Cristian Andrei STAN**
Număr matricol: 21010346
Semnătura:_____

Cuprins.

1. Introducere

Aplicațiile web sunt omniprezente, fie că vorbim despre internet sau despre intranet-uri. Aproape orice tip de business, și aproape orice tip de companie, asociație, instituție dar și inițiative personale au nevoie de o prezență on-line pentru a funționa cu succes în societatea de azi. Dar aceste aplicații web nu trebuie doar să existe, ele se schimbă și evoluează deodată cu nevoile business-ului și ale pieței.

În aceste condiții este vital să existe un ciclu rapid de dezvoltare al aplicațiilor, începând cu specificarea cerințelor și terminând cu livrarea produsului. De asemenea, aceste aplicații devin din ce în ce mai complexe, și cerințele și așteptările sunt mai mari. De aceea dezvoltatorii software caută mereu modalități de a ușura munca și de a construi aplicații complexe cu un minim de efort în timp cât mai scurt.

Unul dintre modurile în care se poate acest lucru este automatizarea sarcinilor. De exemplu, medii de lucru cât mai inteligente care să preia sau să ajute la îndeplinirea a cât mai multor sarcini ale dezvoltatorului, începând de la procesul de compilare până la realizarea documentației. Acest lucru scade timpul procesului de dezvoltare, dar nu este destul. Ceea ce este cu adevărat nevoie este o nouă abordare, un nou mod de a privi lucrurile la un alt nivel de abstractizare.

Pentru a ține pasul cu creșterea vitezei proceselor și intoleranța la defecte, dezvoltatorii nu mai trebuie să piardă timp scriind cod de umplutură, repetitiv, care nu are impact direct asupra funcționalităților sistemului. Acestea ar trebui să constituie focusul, nu legarea între ele a framework-urilor și tehnologiilor și managementul resurselor și a configurațiilor. Toate aceste elemente pot fi și ar trebui să fie cât mai automatizate pentru a permite oamenilor să facă ceea ce știu mai bine, și anume să gândească și să creeze.

Pentru a adresa aceste probleme, diverse soluții au fost propuse. Începând cu programarea orientată obiect, middle-ware-uri, servicii web, Service Oriented Architecture, Software as a Service, Aspect Oriented Architecture, Cloud Computing. Toate acestea sunt diferite straturi de abstractizare care permit dezvoltarea mai rapidă, refolosirea, comunicarea între sisteme eterogene, colaborarea, etc.

Dar oamenii sunt mai mult ființe ce gândesc în imagini decât în linii de cod sau concepte abstracte. De aceea s-a consumat atât de mult timp și resurse pentru ajungerea la un standard de reprezentare grafică a conceptelor abstracte ce țin de un program software, și anume modelul problemei. Pentru înțelegerea în profunzime a particularităților unei aplicații precum și pentru a facilita comunicarea între diferite categorii de stakeholderi, modelele și reprezentarea grafică sunt esențiale. Dar de ce să fie petrecut atâta timp și resurse pentru crearea unor artecte care sunt utile doar până la terminarea aplicației, sau și puțin după, în mentență? De ce nu modele aplicației să devină parte din aceasta?

Așa a luat naștere conceptul de Model Driven Architecture (MDA). MDA este o paradigmă care specifică că modelul care descrie o aplicație într-un limbaj formal de modelare (de exemplu UML) ar trebui să stea la baza arhitecturii acesteia, și să devină o parte vie a sistemului prin generarea automată a codului pe care îl abstractizează.

MDA reprezintă o revoluție a metodologiilor de dezvoltare software și necesită aceeași schimbare în gândire ca și trecerea de la programarea în limbaj mașină la cod de asamblare, sau de la programarea imperativă la programarea orientată obiect. Aplicând acest mod de creare de aplicații, dezvoltatorii ar trebui să fie în sfârșit eliberați de lucrurile care nu țin de funcționalitățile sistemului, și să se poată concentra pe îndeplinirea lor.

2. Obiectivele proiectului

Ținând cont de importanța aplicațiilor web, și de așteptările mari de la dezvoltatorii lor (un ciclu rapid de dezvoltarea, calitate, reactivitate la schimbări, securitate, comportament uniform în diferite medii, etc.) am decis să investighez modul în care conceptele de Model Driven Architecture pot fi aplicate în dezvoltarea aplicațiilor web și cum promisiunile MDA (independență de tehnologii, generarea aplicațiilor din modele, scăderea substanțială a timpului și a costurilor de dezvoltare) sunt indeplinite pentru această categorie de aplicații.

Cel mai mult timp în dezvoltarea aplicațiilor web în momentul actual nu este petrecut pe dezvoltarea cerințelor funcționale ale sistemului și adăugarea de valoare reală pentru utilizatori, ci este ocupat de codul de rutină. Aici mă refer la întreaga stivă a unei aplicații, începînd de la comunicarea cu baza de date, managementul serviciilor, tratarea erorilor, integrarea diferitelor tehnologii și framework-uri, managementul configurațiilor, internaționalizarea, stratul de prezentare.

În mod special pentru a o aplicație web, partea de prezentare pune probleme, deoarece este greu de menținut un design consistent, care să respecte standardele de calitate, în special când mai mulți dezvoltatori lucrează de același proiect. Problemele care pot apărea sunt diverse, dar câteva dintre acestea sunt: prea multe date în sesiune, denumiri confuze ale variabilelor și parametrilor, stiluri inconsistente de programare, cod duplicat, un mod greșit de tratare a erorilor, incompatibilitate între browsere, etc. Toate acestea duc la posibilitatea apariției defectelor și complică inutil munca de mentenanță. De cele mai multe ori aceste probleme sunt date de lipsa unei vederi de ansamblu a aplicației pe partea de prezentare.

Chiar dacă modelarea și analiza rezolvă multe probleme pentru obiectele de business, structura bazei de date, sau interacțiunile intre servicii, încă este dificil de exprimat într-un mod formal dar totuși intuitiv cum o aplicație web funcționează la nivel de prezentare.

Așadar obiectivul lucrării de față îl reprezintă identificarea unui framework care să permită generarea în cât mai mare măsură a unei aplicații web, folosind principiile MDA, începând de la structura bazei de date până la interacțiune și flow-ul paginilor web, și dezvoltarea unei aplicații nu foarte complexe ori complete din punct de vedere al domeniului de business modelat dar care să acopere cât mai mult din capacitățile framework-ului pentru a putea analiza impactului pe care îl are aplicarea MDA în dezvoltarea aplicațiilor web și eventualele elemente care încă nu sunt suficient dezvoltate și care suportă îmbunătățiri.

Cerințele după care am ales framework-ul au fost:

1) **Să fie gratuit**. Deoarece aplicația web care o să o creez nu are ca scop folosirea în mod comercial pentru a genera un profit, ci doar scop demostrativ, dezvoltarea ei nu trebuie să coste nimic. Acest lucru limitează deja foarte mult căutările deoarece sunt multe tool-uri comerciale de calitate ce aderă la standardele MDA folosite

de companii mari pentru a genera aplicații foarte complexe, dar prețul lor este prohibitiv pentru scopul de față. Multe povești de succes despre aceste tool-uri pot fi găsite pe site-ul OMG la categoria povești de succes. Toate acele articole demonstrează încă o dată cât de importantă este MDA pentru industria dezvoltării de software în ziua de azi, și cum aplicații foarte complexe pentru domenii diferite (bancar, aeronautic, transporturi, medicină, construcții de mașini, industrie) au fost create folosind paradigma MDA în timp record la prețuri mult mai mici decât în mod tradițional.

2) **Să fie matur** și folosit cu succes în dezvoltarea aplicațiilor web de o gamă largă de utilizatori.

3) **Să adere cât mai bine la principiile MDA**. În primul rând să ofere posibilitatea creării modelelor de tip PIM care să nu depindă de anumite tehnologii, și apoi să ofere și un suport solid pentru generarea codului de calitate mapat pe tehnologii și limbaje de programare actuale cât mai cunoscute și folosite.

4) **Codul generat să fie de calitate**, și să respecte standardele din industrie. Trebuie să poată oferi posibilitatea creării de sisteme unde logica de business, persistența și prezentarea să fie separate (să adere la modelul Model-View-Controller sau Layered) pentru ca aplicația să poată schimba ulterior anumite elemente fără ca funcționalitatea per ansamblu să fie afectată. Astfel se asigură independența de tehnologie, reactivitatea la schimbări și un nivel redus de cuplare.

5) **Să poată genera în cât mai mare masură codul referitor la partea de prezentare**. Așadar, folosind un limbaj formal dar totodată intuitiv, să se poată exprima modul în care paginile arată și interacționează, adică să se poată modela cerințele de business și scenariile de utilizare și modul de prezentare al acestora. Acest cod generat trebuie să respecte standardele de bună practică din industrie, și să permită utilizatorului să facă eventuale schimbări manuale dacă este cazul (să poată modifica designul și conținutul fără ca regenerarea aplicației să le afecteze).

Conform [15], Androma este ”unul dintre cele mai puternice generatoare MDA open source de pe planetă. Este folosită pentru orice de la cele mai simple aplicații CRUD la aplicații enterprise complexe. Andromda vine cu o gamă largă de cartridge-uri gata implementate pentru Spring, Hibernate, EJB, .Net, Hibernate, Struts, Jsf ”.

Analizând și [16] și [17], precum și documentația framework-ului [18] și prezentarea acestuia [14], am dedus că acest framework respectă toate cerințele impuse, și în plus are o gamă largă de cartridge-uri gata implementate de generare de cod pentru stiva de tehnologii J2EE, inclusiv de prezentare, prin JSF și Struts.

Acestea au fost motivele pentru care am ales Andromda pentru a demonstra modul în care se pot dezvolta rapid aplicații web de calitate aplicând paradigma MDA și gradul de indeplinire al obiectivelor MDA pentru acest tip de aplicații.

3. Studiu bibliografic

Model Driven Architecture a fost propusă ca metodologie de dezvoltare software în 2001 de către consorțiul internațional de standardizare Object Management Group (prescurtat OMG), prin articolul [1]**.** Pentru a înțelege mai bine contextul și bazele acestei metodologii, este necesar să înțelegem ce este OMG, care este activitatea lui, procesul prin care acesta a elaborat-o și care este motivația care a dus la apariția ei.

3.1. OMG și standardele care au dus la apariția MDA

OMG este format din peste 800 de companii din industria calculatoarelor și a tehnologiei informației.

OMG creează doar specificațiile, nu și implementări, dar înainte ca un nou standard să poată fi acceptat ca specificație, membrii echipei care creează specificația trebuie să ofere garanții că în cel mult un an un produs care respectă respectivele specificații va fi lansat pe piață.

Scopul OMG este de a seta standarde care să ajute la reducerea complexității, a costurilor și a dificultăților în crearea de aplicații software. Inițial standardele se axau mai mult pe sistemele distribuite orientate obiect. De atunci focusul grupului s-a schimbat și acum se concentrează mai mult pe modelare (a softului, a sistemelor și a proceselor de business), precum și pe standardele bazate pe model.

Printre cele mai importante produse ale OMG sunt: BPMM, UML, XMI, MOF și MDA.

3.1.1. BPMN

BPMN înseamnă Business Process Model and Notation și este o reprezentare grafică care specifică procese de business în cadrul unui model de proces de business; în momentul actual este standardul pentru modelarea proceselor de business. Această modelare se face folosind diagrame tip flowchart similare diagramelor de activitate din UML. Obiectivul acestui standard este de a permite modelarea proceselor de business într-un mod cât mai intuitiv atât pentru personalul tehnic care se ocupă de dezvoltarea efectivă a aplicațiilor, cât și a personalului non-tehnic care creează aceste procese, le rafinează și monitorizează (business analysts, manageri, alți stakeholderi). Specificația BPMN încurajează ca modelul procesului de business să devină parte integrantă a aplicației prin maparea între reprezentarea grafică și limbajele de execuție (BPEL – Business Process Execution Language).

BPEL sau WS-BPEL este un limbaj standard care permite specificarea acțiunilor în procesele de business utilizând ca interfețe servicii web. Aceste interacțiuni pot fi exprimate în două moduri: procese de business executabile sau abstracte. Cele executabile modelează comportamentul real al unui participant într-un proces; iar cele abstracte au un rol mai mult descriptiv și modelează în mod incomplet procese care nu se dorește a fi executate.

3.1.2. UML

UML (Unified Modeling Language) – este cel mai cunoscut limbaj de modelare în ingineria software. Acest limbaj permite crearea modelelor vizuale a sistemelor sofware orientate obiect. Această modelare se face prin diferite tipuri de diagrame, care oferă informații diverse cu privire la un sistem software. Folosind UML se pot oferi două vederi diferite ale unui sistem:

- **Vedere statică sau structurală**, în care sunt evidențiate obiectele, atributele, operațiile și relațiile între acestea. Reprezentată prin diagrame de clasă și diagrame de structură.

- **Vedere dinamică sau comportamentală**, în care este evidențiat comportamentul dinamic al unui sistem arătând colaborarea între obiecte și schimbarea stării interne a acestora. Câteva exemple: diagrame de secvență, de activitate, state-machine.

Folosind o diagramă UML de clasă se pot prezenta toate aceste tipuri de diagrame și relația între ele:

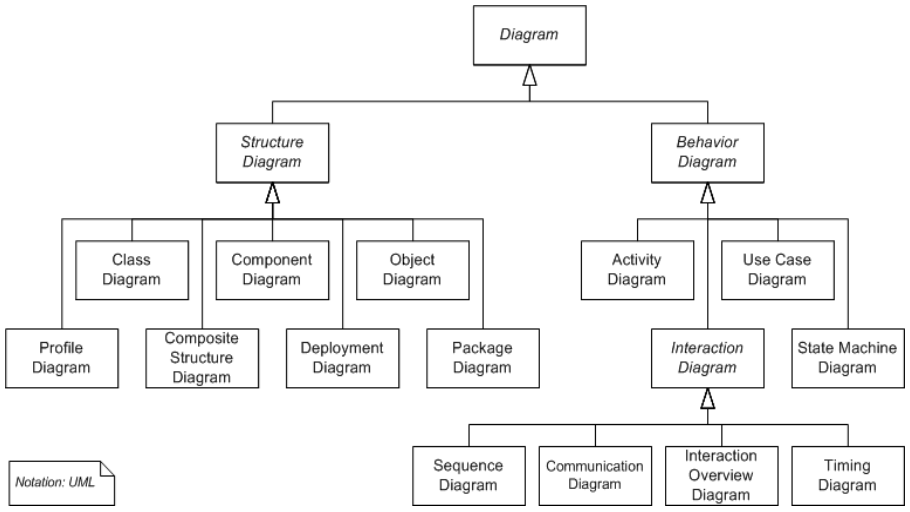

Figura 3.1: Tipuri de diagrame UML.

3.1.3. MOF (Meta-Object Facility)

OMG a creat o arhitectură de meta-modelare pentru definirea standardului UML prescurtată MOF (Meta-Object Facility). MOF reprezintă standardul pentru ingineria condusă de model. Această metodologie de dezvoltare software (Model Driven Engineering - MDE) se concentrează pe crearea și exploatarea modelului domeniului, nu pe calcule sau algoritmi. MOF este o arhitectură de tip layer, cu patru nivele.

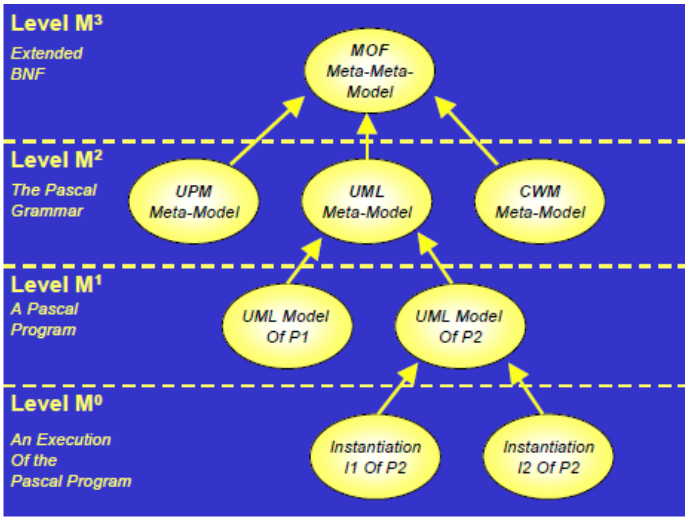

Figura 3.2: Stiva standard de modelare a OMG.

Pe primul nivel se regăsește meta-meta modelul. Acest limbaj este folosit de Meta-Object Facility pentru a construi meta-modelele de pe al doilea nivel. Un exemplu de meta-model este meta-metodelul UML; sau modelul prin care este descris UML. De asemenea, pe acest nivel mai pot fi meta-modelele CWM (Common Warehouse Metamodel – standardizează reprezentarea modelelor bazelor de date (scheme), modele de transformare a schemelor, OLAP și modele de data mining, etc) sau UPM (Unified Process Model). De fapt aceste meta-modele existau independent în lumea dezvoltării de software înainte de specificarea standardului MOF, care a fost creat tocmai pentru că s-a ajuns la concluzia că toate aceste meta-modele pot fi abstractizate și reunite creând un meta-meta model. Pe următorul nivel se găsesc de exemplu modelele create folosind UML sau CWM, iar pe ultimul nivel – numit câteodată nivelul de date sau execuție se regăsesc instanțele efective ale sistemelor implementează un model definit pe nivelul anterior.

Așadar MOF cuprinde următoarele elemente:

- Un model abstract pentru obiecte MOF generice și asocierile lor.
- Un set de reguli pentru maparea oricărui metamodel bazat pe MOF la interfețe independente de limbaj (ex: CORBA IDL).
- Reguli privitoare la ciclul de viață, compoziția și semantica meta-modelelor create.
- O ierahie de interfețe reflective care permit operații generice pentru descoperirea si manipularea modelelor create după meta-modele MOF care nu au o interfață cunoscută.

Puterea MOF constă în faptul că permite folosirea în același mod și cu aceleași unelte a diferite meta-modele care modelează domenii diferite.

3.1.4. XMI (XML Metadata Interchange)

XMI (XML Metadata Interchange) este un alt standard OMG care mapează MOF la standardul eXtensible Markup Language (XML) al W3C. XMI definește tag-urile XML și semnificația acestora pentru a reprezenta modele de tip MOF serializate. Metamodele MOF sunt transformate în DTD (XML Document Type Definition) și modelele sunt transformate în documente XML care sunt consistente cu DTD-ul lor corespunzător. Faptul că XMI se bazează pe XML înseamnă că atât meta-data, cât și instanțele pe care aceste meta-date le descriu (modelele propriu-zise) pot fi puse împreună în același document. Acest lucru permite aplicațiilor să interpreteze instanțele modelelor doar după meta-date pe care modelul le conține, ceea ce face comunicarea autodescriptibilă și deci asincronă. De aceea XMI este foarte important pentru schimbul informațiilor în sisteme distribuite eterogene.

Crearea MOF și XMI a permis evoluții în alte domenii ale Computer Science. Unul dintre acestea este Web-ul semantic și dezvoltarea ontologiilor [9]. OMG a definit, folosind MOF un meta-model de definire a ontologiilor (ODM – Ontology Definition Metamodel). Acest metamodel, la care i se adaugă o extensie de UML (Ontology UML Profile - OUP) permite definirea ontologiilor folosind diagrame UML, precum și reguli de transformări între aceste modele și limbajele web-ului semantic (RDF, OWL). Astfel două mari domenii foarte importante în știința calculatoarelor: ontologiile cu internetul semantic și modelarea UML sunt reunite.

3.2. Model Driven Architecture

3.2.1. Prezentare

Standardele deja create de OMG (UML, CWM, MOF, XMI) au dus și la crearea conceptului de arhitectură condusă de model sau MDA (Model Driven Architecture), care nu este un standard în sine, ci o paradigmă de software development. Specificațiile MDA au fost lansare în 2001 și de atunci au intrat dar sigur în atenția tuturor dezvoltatorilor de software prin potențialul pe care îl promite și gradul de dificultate al problemelor pe care vrea să le rezolve, și anume portabilitate, interoperabilitate și refolosire. Scopul suprem și specificațiile MDA originale presupun crearea unei aplicații funcționale complete dintr-un model UML și folosirea aceluiași model pentru a implementa respectiva aplicație pe diverse platforme.

De la înființarea OMG, scopul acestui consoțiu a fost crearea de standarde care permit abstractizarea limbajelor de programare, platformelor și aplicațiilor specifice și crearea unui sistem global de informație similar cu rețeaua electrică. Așa cum un utilizator se leagă la o priză pentru a obține curent electric, la fel de simplă ar trebui să fie conectarea diferitelor sisteme informatice și schimbul de date între acesta. Un prim pas l-a reprezentat CORBA, urmat de XMI și unificarea conceptului de internet semantic cu modelarea UML. Dar OMG a observat un alt fenomen care complică situația, pe lângă diferitele limbaje și platforme, și anume răspândirea middle-ware-urilor care abstractizează pentru aplicații diverse servicii esențiale: tranzacții, persistență,

comunicare, controlul evenimentelor și mesajelor, etc. Deși toate middle-ware-urile sunt într-o continuă schimbare, evoluție și transformare, este imposibilă impunerea ca standard a acestora; așadar trebuie găsită o soluție care să fie neutră față de ele.

Soluția propusă poartă numele de MDA și este reprezentată în figura următoare:

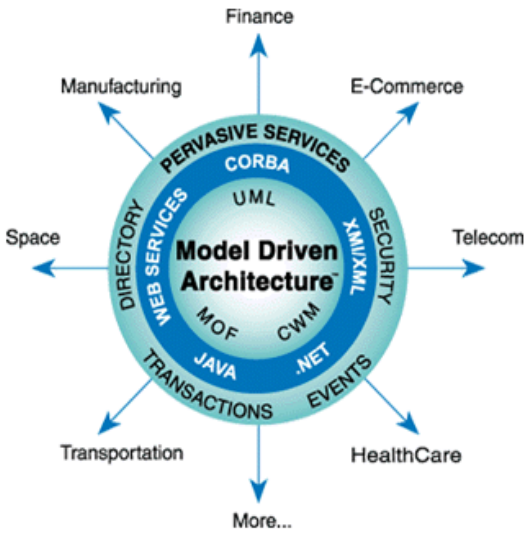

Figura 3.3: Model Driven Architecture în perspectiva OMG. [20]

În centrul arhitecturii se regăsesc standardele de modelare deja specificate de OMG: UML, MOF, CWM și altele. Pot fi multiple tipuri de modele specifice domeniului căruia i se adresează aplicația. De exemplu: financiar, medical, tranporturi, e-commerce, etc. Fiecare acest tip de model este independent de implementare (limbaj, platformă, middle-ware) și numărul lor este relativ mic deoarece ele reprezintă de fapt meta-modele (adică reprezintă toate modelele care vor fi implementate pentru acel domeniu). Concret, ele vor fi profile UML specializate.

Un model este specificarea formală a unei funcționalități, structuri sau comportament într-un sistem având un anumit context și dintr-un anumit punct de vedere. Un model poate fi reprezentat prin diagrame sau text; de obicei se folosește o notație formală (de exemplu UML), care poate fi extensă prin explicații în limbaj natural. O specificație este formală atunci când se bazează pe un limbaj cu o semantică bine definită. Acest formalism permite limbajului UML să fie exprimat în format XML și să adere la o schemă bine-definită (XMI).

3.2.2. Promisiunile MDA

Aplicațiile vor fi dezvoltate inițial folosind modele independente de o platformă specifică, ceea ce separă procesul de modelare a cerințelor de business de dezvoltarea software propriu-zisă. Acest lucru crește gradul de abstractizare, și permite concentrarea pe cerințele sistemului și comportamentul acestuia, nu pe detalii specifice platformei, ceea ce reduce defectele și scade riscurile proiectului. De asemenea, este mult mai ușoară

și ieftină dezvoltarea aplicației în paralel pe diverse platforme, sau portarea de pe o platformă pe alta în funcție de schimbările tehnologiei sau preferințelor utilizatorilor/dezvoltatorilor.

Un alt avantaj este faptul că nu mai trebuie separată documentarea tehnică a produsului și munca software arhitectului sau a business analistului de activitatea dezvoltatorilor. Fie că ne place sau nu, orice proiect software non-trivial are nevoie de un anumit nivel de modelare și documentare pentru ca toți cei implicați în dezvoltarea acestuia să înțeleagă aspectele esențiale. Datorită naturii limbajelor de modelare (diagrame vizuale și intuitive), chiar și personalul non-tehnic le poate înțelege cu ușurință, și reprezintă o modelitate foarte eficientă de comunicare.

Productivitatea crește și timpul necesar lansării unui produs nou este mult redus deoarece cea mai mare parte a codului care reprezintă o muncă de rutină este generată automat; în plus sistemul va avea o calitate superioră datorită consistenței și calității artefactelor produse.

3.2.3. Spectrul modelării software în proiectele actuale

Astăzi dezvoltarea software se poate situa pe diferite nivele ale spectrului modelării, de la lipsa cu desăvârșire a oricărui model până la model driven architeture.

Se pot defini câteva categorii mari în funcție de relația dintre model și codul sursă pe care acesta îl descrie:

- **Doar cod**. Nu se folosesc modele care să descrie sistemul; comportamentul acestuia se deduce direct din codul sursă. Singura ”modelare” care se face este dată de abstractizarea permisă de limbajul de programare ales: interfețe, module, clase, pachete. Orice alt fel de modelare este pur informală, și dacă există se regăsește doar pe tablele de proiectare, prezentări power-point sau caiete. Este potrivită doar pentru proiecte mici sau cu foarte puțini membri, deoarece comunicarea este dificilă, și sunt greu de înțeles caracteristicile cheie ale sistemului printre detaliile specifice de implementare. Această metodă are ca principal dezavantaj dificultatea evoluției și menținerii dacă sistemul crește în complexitate.

- **Vizualizarea codului**. Pe măsură ce sistemul este dezvoltat și crește în dimensiuni și complexitate, apare nevoia de vizualizare a codului prin reprezintări grafice care ajută la înțelegerea structurii și comportamentului acestuia. Există unelte care permit vizualizarea codului simultan cu reprezentarea lui grafică, și chiar modificarea codului în funcție de modificările aduse acestei reprezentări. Acest tip de model se numește model de cod sau de implementare, sau chiar diagrame ori artefacte; cu toate astea reprezintă doar un mod alternativ de a vedea codul existent.

- **Relație bi-direcțională între cod și model**. Modelul este abstractizat și descrie arhitectura și comportamentul sistemului. De regulă se creează un model care descrie sistemul până la un anumit nivel de detaliu, urmând ca echipa care realizează implementarea propriu-zisă să creeze modele specifice cu un grad mult mai mare de detaliu și codul asociat cu acestea. Dacă modelul abstract se schimbă, aceste schimbări

trebuie reflectate și în modelele detaliate și în cod, și invers dacă anumite porțiuni trebuie implementate într-un mod diferit de modelul abstract acesta va trebui modificat pentru a reflecta schimbările. Din păcate acest mod de dezvoltare îngreunează destul de mult dezvoltarea, scade reactivitatea la schimbări și are șanse mari să ducă la inconsistențe după câteva iterații dacă nu se păstrează o disciplină rigidă.

La celălalt capăt al spectrului găsim MDA, care rezolvă toate aceste probleme, deoarece modelul este aplicația, și nu o povară în plus pentru dezvoltatori care trebuiau să aibă grijă să mențină și modelele actualizate doar pentru a avea luxul de a înțelege sistemul mai usor.

3.2.4. Ciclul de dezvoltare software utilizând MDA

Un ciclu complet de dezvoltare software folosind MDA este reprezentat în figura următoare:

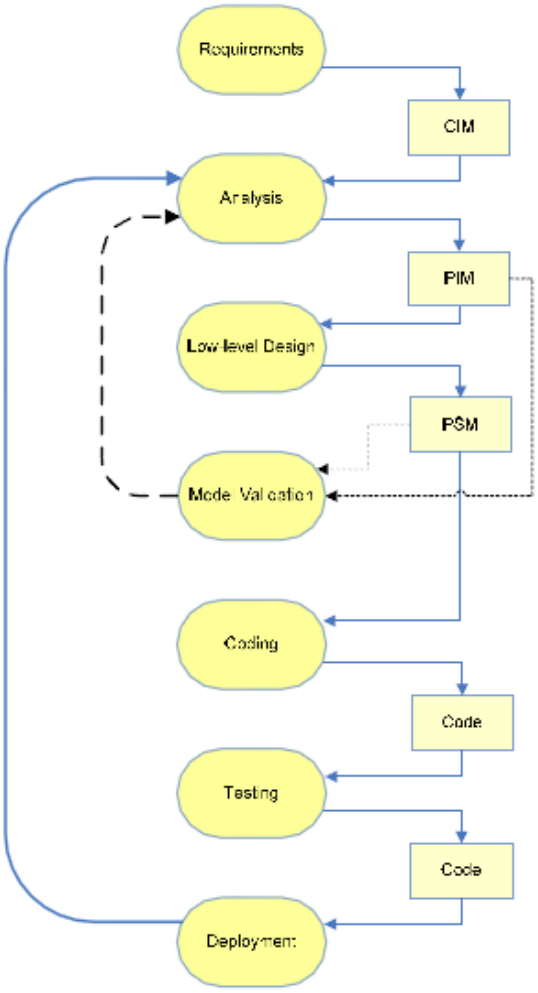

Figura 3.4: Ciclul de dezvoltare folosind principiile MDA.

MDA specifică trei tipuri de modele corespunzătoare diferitelor straturi de abstractizare, fiecare având o vedere din ce în ce mai detaliată a sistemului : CIM, PIM și PSM.

Modelul CIM (Computational Independent Model)

Cerințele funcționale de business ale utilizatorilor sunt de regulă exprimate ca text, și sunt folosite pentru a crea un model independent de calcul (CIM – Computational independent model). Acest model este numit și model de business sau de domeniu deoarece folosește un vocabular familiar experților în domeniu, și are un rol important în reducerea diferențelor între experții domeniului și experții care vor implementa sistemul. Acest model este input-ul pasului de analiză conform specificațiilor OMG, și trebuie să poată fi urmărit în modele următoare, deși în practică de cele mai multe ori analiza se face direct pe baza specificațiilor, și cele mai multe tool-uri comerciale au ca pas de plecare PIM .

Pentru a completa această lipsă, OMG a adoptat standardul Semantics of Business Vocabulary and Business Rules (SBVR) care vrea să fie baza pentru o descriere formală și detaliată a unor entități complexe de business folosind limbaj natural. Folosind MOF, SBVR definește vocabularul și regulile necesare pentru a defini vocabulare și reguli specifice unui business, și o schemă XML care permite inter-schimbarea acestor vocabulare și reguli între organizații și tool-uri software, precum și reguli de mapare XMI. SBVR este o parte integrantă a MDA, deși încă tehnologia este prea puțin maturizată pentru a permite aplicațiilor comerciale să ofere posibilitatea de a crea modele complexe și complete din limbaj natural.

Modelul PIM (Platform Independent Model)

Acesta este modelul creat folosing un limbaj de modelare (de regulă UML) care descrie în mod complet un sistem software sau un sistem de business și nu depinde în nici un fel de o platformă sau o tehnologie specifică. Este vederea abstractă a sistemului specificată într-un tip de model specific domeniului (folosing un profil UML specializat). Pentru a păstra independența de model trebuie definite în mod abstract un set de servicii care să nu țină cont de detaliile tehnice, și în alte modele succesive se vor realiza aceste servicii ca elemente specifice.

Modelul PSM (Platform specific model)

Următorul pas îl reprezintă transformarea modelului PIM într-un model PSM. Datorită standardelor MOF și XMI, această transformare este înlesnită și poate fi automatizată în mare măsură (deși în anumite situații dezvoltatorul are sarcina de a interveni și a face modificări manual; acest pas se mai numește și low-level design deoarece trebuie clarificate și create detalii specifice implementării). Pentru a formaliza tranzițiile între modele precum și a umple alte lipsuri în dezvoltarea și managementul modelelor, a fost creat standardul QVT (Query/View/Transformation). Așa cum spune și numele, acest standard acoperă transformarea, vederile și interogările aplicate modelelor.

QVT este format din 3 limbaje concrete, și este aplicabil modelelor care implementează meta-modelele ce se conformează standardului MOF începând cu versiunea 2.0. QVT integrează și extinde OCL (Object Contraint Language – limbaj declarativ de descriere a regulilor ce se aplică modelelor UML).

Așadar, PIM este modelul care descrie funcționalitatea sistemului (oferă răspunsul la întrebarea ”Ce construim?”), iar PSM este modelul în care se descrie cum această funcționalitate este realizată pe o anumită platformă (răspunde la ”Cum implementăm?”). Concret, modelele PSM vor fi similare modelelor PIM, dar vor implementa niște profile UML specifice unei platforme. Această transformare nu trebuie vazută neapărat ca o transformare într-un singur pas, sau unidirecțională. Pot fi create diverse nivele de abstractizare a platformei, de aceea modelul poate trece prin transformări succesive, fiecare aducând un alt nivel de detaliu și după fiecare transformare modelul poate fi supus unui proces de validare.

Un lucru important de înțeles este faptul că pentru MDA noțiunea de platformă este relativă și are sens doar privită dintr-un anumit punct de vedere, sau cu alte cuvinte același model poate fi văzut ca PIM sau PSM. De exemplu, un model care descrie o aplicație este văzut ca PIM cu privire la comunicarea cu alte sisteme, dacă decizia de a implementa un anumit middle-ware de comunicare nu a fost făcută. Însă dacă se decide că sistemul va comunica folosind tehnologia CORBA sau JMS, modelul va deveni un PSM specific CORBA sau JMS. Însă acest model PSM poate fi PIM cu privire la stratul de persistență (care va putea fi Hibernate, JPA, TopLink, etc.).

Transformări între modele

Conceptul de transformare (sau mapare) este o noțiune esențială a MDA, laolată cu cea de model. Următoarele feluri de transformări se pot defini:

- Rafinări ale PIM. Această transformare este folosită atunci când modelul este specializat, filtrat sau îmbunătățit, fără a se aduce noi informați.

- PIM în PSM. Generarea inițială a modelului PSM folosind diverse mapări predefinite.

- Rafinări ale PSM. Se pot aduce modificări ale modelului PSM pentru a rezolva anumite probleme legate de deployment sau pentru optimizări.

- PSM în PIM. Această transformate este folosită pentru a abstractiza modele deja existente specifice platformei în modele generale dacă se dorește migrarea spre o nouă platformă, sau pur și simplu o viziune de ansamblu a aplicației fără multiplele elemente specifice.

Generarea codului

O dată ce avem un PSM la nivelul de detaliu specific implementării dorit (numit și ISM (Implementation Specific Model)) următorul pas logic este generarea aplicației, mai exact a codului sursă. Din model trebuie generată o gamă largă de fișiere: cod sursă, interfețe, web-servicii, fișiere de configurare, scripturi de build, etc. Cu cât modelul PSM este mai bogat în detalii cu atât va fi mai complet nivelul de generare al codului. Într-un proces de dezvoltare software MDA matur, generarea codului va fi complexă, poate chiar completă.

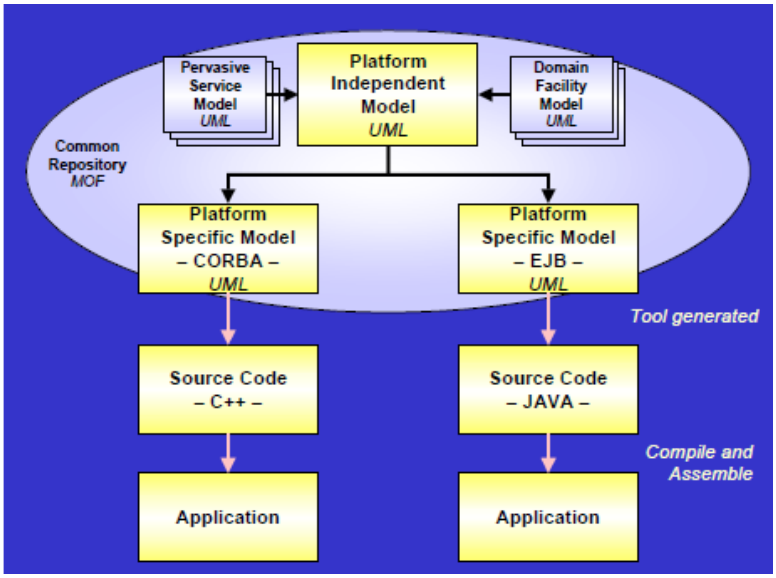

Figura 3.5: Procesul de generare de cod folosind MDA.

În figura 3.4 se arată flow-chart-ul acestui proces de generare. Domain Facility Model – de care este legat PIM se referă la modelele specifice domeniul care sunt standardizate. De exemplu: domeniul financiar, aero-spațial, telecomunicații, medical, etc.

Pervasive Service Model reprezintă seturi esențiale de servicii implementate de diverse middle-ware-uri. De exemplu: managementul evenimentelor, notificări, securitate, tranzacții, etc. De asemenea, atribute ca scalabilitate, tratamentul erorilor și alte caracteristici non-funcționale pot fi modelate dacă se dorește. Toate aceste elemente sunt desigur opționale, dar cum am spus mai sus, cu cât modelul va avea mai multe detalii, cu atât gradul de completitudine al generării va fi mai mare.

O dată generarea completă, cel mai probabil vor exista elemente care trebuie codate manual, cum ar fi eventuale interfețe cu sisteme deja existente, optimizări de performanță, elemente ale interfeței cu utilizatorul, etc. După codare, sistemul trebuie verificat dacă îndeplinește cerințele funcționale și non-funcționale; și dacă toate testele sunt validate, atunci aplicația este terminată.

3.2.5. Viziunea şi viitorul MDA

Viziunea MDA pe termen scurt este cea a unui mediu în care interoperabilitatea eficientă și fără probleme între aplicații, unelte și baze de date este realizată prin schimbul de modele comune. Aceste componente vor fi integrate în mediu prin implementări ale standardelor MDA care le permite să expună și să facă schimb de meta-date ca instanțe de modele predefinite. Aceste servicii vor fi exprimate prin modele standard de programare (API) care vor fi generate din modele independente de platformă.

Meta-data este elementul esențial care permite comunicarea între sisteme eterogene. Chiar dacă această interoperabilitatea se poate realiza prin implementarea de API-uri standardizate, meta-datele comune definesc semantica și capabilitățile sistemului. Aplicațiilor se vor conecta în mediul comun și vor descoperi meta-datele prezente în acesta. De asemenea, noul sistem va trebui să își expună propriile meta-date pentru a fi văzut de restul mediului.

Această abundență de meta-date va permite interoperabilitatea diverselor aplicații software în următoarele moduri:

- Schimbul de date, transformarea, și maparea tipurilor între resurse diferite poate fi exprimat în mod formal prin descriptori independenți de produs.
- Generarea schemelor se poate face pe baza meta-datelor elementelor comune ale acesteia. De exemplu, o aplicație poate folosi diverse tipuri de baze de date care vor avea scheme diferite, dar care aderă la un model standard definit prin meta-date expusă de sistem.
- Se poate crea un strat de vizualizare și inteligență de business care va utiliza meta-datele sistemelor pentru a procesa și formata datele schimbate între ele pentru a fi cu ușurință analizate și înțelese. Astfel se creează un nivel de abstractizare care permite utilizatorilor umani să interpreteze ușor rezultatele obținute.
- Componente diferite care nu aveau cunoștință de capacitățile, interfețele și tipurile de date ale celuilalt vor putea coopera fără probleme o dată ce fac schimb de meta-date care le descriu; astfel sisteme vor putea crea legături noi și să-și extindă rețeaua fără a fi nevoie de intervenție umană pentru a ghida acest proces.

În plus, sistemele MDA nu vor trebui să își schimbe structura internă și reprezentarea proprie a meta-datelor, acestea pot rămâne la fel. Acest lucru se datorează faptului că meta-data expusă mediului exterior este o definiție externalizată, care se supune unor standarde pentru a fi cât mai generică și deci înțeleasă de cât mai mulți participanți. De asemenea, dacă se dorește specializarea meta-datei aceasta poate fi extinsă cu mecanisme de extindere ce fac parte din standardele modelelor generice (de exemplu, pentru UML se pot folosi profile specializate, stereotipuri, tagged values, constrângeri).

Viziunea MDA pe termen lung vede sistemele software ca entități capabile să descopere în mod automat proprietățile mediului în care se află, și să se adapteze la acesta prin modificarea dinamică a comportamentului propriu. Practic constituie o evoluție de la sistemele actuale care comunică pe baza meta-datelor la sisteme ce se vor adapta dinamic la rulare pentru a face față cerințelor mediului.

Sistemelor se vor baza din ce în ce mai mult pe cunoștințe, și vor avea capabilitatea să descopere proprietăți comune între domenii aparent diferite, și să facă

decizii inteligente pe baza acestor descoperiri. De fapt aceste cunostințe se bazează pe aceste meta-date omniprezente, dar la un nivel mai avansat și mai evoluat. Abilitatea de a crea astfel de sisteme va fi o evoluție și va veni din experiența consistentă de creare de sisteme bazate pe meta-date și ontologii care influențează comportamentul și luarea deciziilor. De asemenea este nevoie de crearea straturilor superioare de abstractizare a cunoștințelor specifice domeniului pentru a permite sistemelor să înțeleagă și să folosească eficient aceste informații. Deja mecanismele de reflecție a multor limbaje de programare permit introspecția unui program static; pasul următor este modificarea dinamică a structurii și comportamentului acestora.

Dacă se vor crea sisteme care nu doar mapează modele generice pe baza meta-datelor în modele de implementare (specifice platformei), ci le vor putea interpreta direct, atunci modificarea modelelor se va traduce direct în modificarea comportamentului aplicației. Astfel, o nouă versiune de model publicată în mediu va determina ca toate sistemele care îl implementează să își schimbe automat comportamentul, chiar în timpul funcționării acestora.

Toate acestea vor conduce la apariția sistemelor adaptive dinamice și care se auto-organizează, care vor acționa pe baza cunoștințelor și acționează inteligent fără a fi nevoie să le fie spus cum să o facă. Astfel, sistemele se vor auto-adapta la schimbări neprevăzute fără a fi nevoie de intervenția programatorului, deoarece își vor putea modifica modelul singure. Dacă sistemul trebuie să fie schimbat, acest lucru se va face de expertul în domeniu, fără să fie nevoie să fie neapărat un expert software, ci doar să înțeleagă respectivul domeniu.

Este evident că această viziune este încă departe de a fi atinsă, dar promisiunile pe care le aduce, dacă vor fi îndeplinite, vor revoluționa cu siguranță lumea tehnologiei informației și va aduce sistemele software la un nivel de inteligență care până nu demult părea accesibil doar oamenilor.

4. Analiză şi fundamentare teoretica

4.1. Integrarea MDA în metodologiile de dezvoltare software

Înainte de a folosi conceptele de Model Driven Architecture, este important să înțelegem cum această paradigmă va influența metodologiile actuale de dezvoltare software dacă hotărâm să o includem în procesul propriu. Așa cum am afirmat anterior, MDA nu este un standard sau o metodologie de dezvoltare software. Este o paradigmă care poate fi inclusă în celelalte procese de dezvoltare, oricare ar fi acestea. Așadar, pe orice spectru al metodologiilor de dezvoltare se află un proces, începând cu Waterfall și continuând cu prototip, spirală, Rational Unified Process, metode agile (Scrum, Extreme Programming, Test Driven Development, etc.), acesta nu va avea decât de câștigat prin aderarea la principiile Model Driven Architecture.

În primul rând, MDA, transferă o mare parte din riscuri în etapele incipiente ale proictului, deoarece o dată ce modelul este complet, la fel va fi și o mare parte din aplicație. Un avantaj de care se bucură mai ales metodologiile iterative este faptul că o dată ce nucleul aplicației, care va cuprinde doar funcționalitățile esențiale de cel mai mare risc, va fi modelat la nivel PIM, se pot face teste folosind diverse platforme specifice. Așadar, putem analiza mai multe variante de implementare și în funcție de gradul de îndeplinire al cerințelor non-funcționale putem alege o variantă finală. Și în mod clasic acest lucru este posibil, dar ocupă mult mai timp.

MDA reduce diferențele dintre metodologiile mai formale și cele mai agile. Astfel fiecare dintre cele două lumi primește ce e mai bun în cealaltă. O metodologie formală va beneficia de pe urma MDA prin faptul că toate modelele detaliate și artefactele produse printr-o analiză consistentă și o planificare riguroasă nu sunt doar pași care trebuie făcuți înainte de a se începe efectiv implementarea și nu vor fi aruncate la sfârșit, ci vor deveni parte integrantă a acesteia. Acest lucru permite și reducerea totală a timpului unui proiect. În metodologiile agile MDA aduce un plus de calitate deoarece descurajează crearea de documentație fragmentată pe iterații, care pe urmă vor fi uitate sau desincronizate de alte iterații succesive ce operează pe aceleași părți ale aplicației. Astfel se îmbunătățește și comunicarea în rândul membrilor echipei, care vor folosi modelul pentru a-și exprima sau înțelege ideile mai bine. Per total iterațiilor pot fi mai rapide, deoarece dezvoltatorul nu va mai trebuie să se ocupe de codul de umplutură, ci se va concetra pe logica de business.

Indiferent de metodologia aleasă, etapa care va avea cel mai mult de câștigat este testarea. Dacă framework-ul ales este suficient de matur să producă cod de calitate și lipsit de defecte, atunci putem să realizăm doar testarea funcționalităților, pentru a fi siguri că sistemul face ce și-a propus.

4.2. Descrierea aplicației

Pentru a ilustra utilitatea principiilor MDA în dezvoltarea aplicațiilor web am decis să construiesc o aplicație model utilizând un tool care va genera marea majoritate a codului, pentru a analiza impactul total al acestei paradigme asupra procesului de dezvoltare.

4.2.1. Definirea problemei

Trebuie creată o aplicație care va automatiza procesul de verificare a activităților angajaților unor companii de IT. Firma se ocupă de dezvoltarea de produse software, și plata de către clienți se face în funcție de numărul de ore total petrecut de aceștia pe un anumit proiect. Pentru a putea justifica aceste ore și a taxa corect clienții, munca pe proiect va fi împărțită de project manager pe sarcini, și fiecare sarcină va fi asignată unui dezvoltator. Acesta trebuie să execute sarcina, și la sfârșit să adauge numărul de ore petrecut pe o anumită sarcină. Project managerul pe urmă va analiza aceste entități, numite fișe de timp, și în funcție de numărul de ore total petrecut pe un proiect, va putea emite facturile către clienți.

Procesele ce trebuie să aibe loc în această aplicație sunt următoarele:

1) Project managerul creează un proiect. În urma sedințelor de analiză cu echipa, se vor identifica sarcinile care trebuie îndeplinite pentru a realiza cu succes proiectul. Aceste sarcini vor fi introduse în sistem, și se vor asigna dezvoltatori pentru fiecare.

2) Fiecare dezvoltator va crea fișe de timp pentru sarcină. O fișă de timp este o entitate ce se creează pentru o anumită zi. Dacă sarcina necesită mai multe zile pentru a fi îndeplinită, pentru fiecare zi se va crea o nouă fișă. Fiecare fișă va trebui să consemneze numărul de ore petrecut în ziua pentru care a fost creată pe sarcina respectivă.

3) După ce dezvoltatorul creează fișele, acestea trebuie să fie validate de project manager. Datorită contractelor și discuțiilor încheiate cu clienții, este posibil ca realitatea dezvoltării să nu coincidă cu așteptările acestora. De exemplu, se poate ca pentru un anumit proiect să se negocieze ca o anumită sarcină să dureze o anumită periodă maximă de timp. În realitate, se poate întâmpla ca respectiva sarcină să dureze mai multe ore de muncă efectivă. În acest caz, se va respinge fișa respectivă, care va trebui rezivuită de dezvoltator. Un alt caz care poate duce la respingerea unei fișe este de exemplu decizia ca o anumită sarcină să nu mai fie inclusă în calcularea finală a prețului. De exemplu, oferirea unui discount clientului pentru comandarea unui nou proiect, sau pur și simplu renunțarea la anumite funcționalități.

4) Fișele aprobate de project manager vor putea fi vizualizate de clienți, pentru a avea transparență asupra procesului de dezvoltare și a vedea exact pe ce a fost petrecut timpul pe proiect. Din aceste fișe se vor crea facturile.

4.2.2. Identificarea actorilor și a rolurilor acestora

Aplicația va avea trei actori: dezvoltatori, clienți și project manageri.

- Project managerul este cel care creează sarcinile, și asignează dezvoltatorii acestora. El trebuie să verifice corectitudinea fișelor completate de dezvoltatori și să le aprobe pentru a fi incluse în calculul costului total al proiectului.

- Dezvoltatorii sunt cei care completează efectiv fișele de timp. Aceștia vor introduce numărul de ore petrecut pentru o anumită sarcină. Aceste fișe pot fi salvate temporar înainte de a le trimite spre aprobare project managerului, pentru a putea fi revizuite.

- Clienții vor avea acces la fișele aprobate pentru proiectul sau proiectele lor. Ei nu vor putea aduce modificări stării sistemului, ci doar să consulte aceste fișe pentru a vedea starea și evoluția proiectului.

4.2.3. Cerințele funcționale ale sistemului

1. Crearea de proiecte și sarcini. Project managerul trebuie să poată crea proiecte noi, și să adauge sarcini acestora.
2. Asignarea de clienți proiectului. La fiecare proiect trebuie asignat un client.
3. Asignarea unui dezvoltator fiecărei sarcini.
4. Completarea fișelor de timp. Fiecare dezvoltator trebuie să poată completa fișele personale de timp pentru sarcinile alocate, să aibă posibilitatea de a le salva pentru verificare, și să le trimită spre aprobare project managerului. Un dezvoltator nu trebuie să acceseze fișele de timp ale altor dezvoltatori.
5. Managementul fișelor de timp. Project managerul va avea posibilitatea de a verifica fișele tuturor dezvoltatorilor, cu excepția celor salvate de aceștia pentru verificarea personală. El trebuie să le poată aproba sau respinge.
6. Clienții trebuie să vadă fișele de timp aprobate de project manager pentru proiectele lor.
7. Dezvoltatori vor avea posibilitatea modificării fișelor de timp dacă au fost respinse de project manager, și a le retrimite spre validare.

4.2.4. Identificarea cazurilor de utilizare

Actorii vor avea următoarele cazuri de utilizare:

- Client:
    1. Logarea în sistem.
    2. Vizualizarea fișelor de timp pentru fiecare proiect pe care îl au.

- Dezvoltator:
    1. Logarea în sistem.
    2. Căutarea fișelor proprii de timp.
    3. Crearea unei noi fișe de timp.

4. Modificarea fișelor de timp.

- Project manager:
  1. Logarea în sistem.
  2. Crearea unui proiect nou pentru un client.
  3. Crearea de sarcini pentru fiecare proiect.
  4. Căutarea fișelor de timp pentru un proiect.
  5. Aprobarea sau respingerea unei fișe.

4.3. Andromda

4.3.1. Descriere

Andromda este un framework de generare de cod extensibil care aderă la paradigmele Model Driven Architecture. Andromda transformă modele UML în cod sursă pentru cele mai folosite platforme.

Este un framework care acceptă modele UML în format XMI, dar nu numai. Se poate extinde pentru a recunoaște orice fel de model care este exprimat în format MOF. Andromda folosește plug-in-uri – care pot fi cartridge-uri sau unelte de transformare. Aceste concepte vor fi explicate puțin mai încolo.

Folosind diverse diverse plug-in-uri se poate genera cod în orice limbaj de programare sau pentru orice tehnologie. Andromda suportă momentan Java și C# ca limbaje de programare, dar dacă se dorește se pot scrie cartridge-uri noi pentru a genera cod în orice alt limbaj.

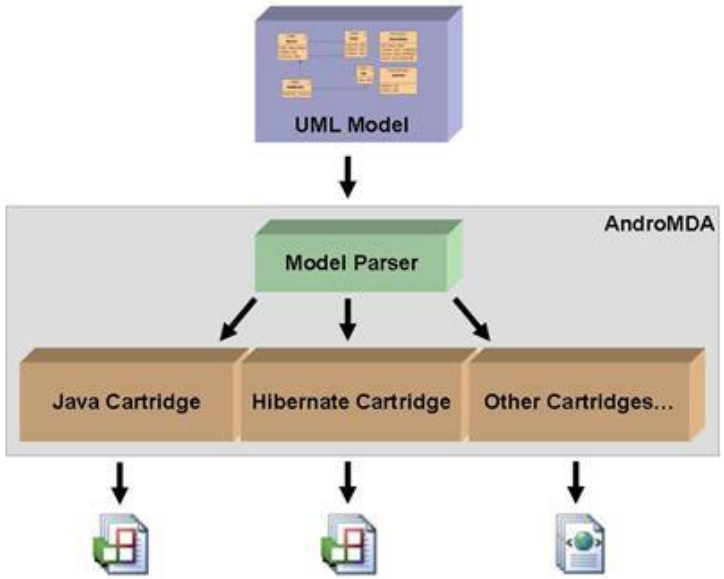

Figura 4. 1: Arhitectura generală a Andromda.

Cel mai des Andromda se folosește pentru a genera proiecte J2EE, deoarece se simplifică foarte mult munca de a lega între ele diverse framework-uri. Pentru J2EE se

poate folosi Hibernate, Spring, Struts, JSF, EJB. Toate aceste cartridge-uri sunt gata făcute; există și suport pentru Web servcii și modelarea proceselor de business (jBPM).

Andromda are două componente importante. Prima este și partea centrală a frameworkului, și anume sistemul de generare de cod. A doua este sistemul de build și management al resurselor proiectului bazat pe Maven. Deși folosirea acestei componentă simplifică enorm atât crearea unei noi aplicații cât și dezvoltarea ulterioară, este opțională. Chiar dacă nucleul de generare poate fi apelat direct din alte unelte, folosirea Maven este puternic recomandată pentru orice proiect J2EE. Folosind plug-in-ul andromdapp:generate Andromda creează automat un nou proiect J2EE ce utilizează Maven, pe baza opțiunilor introduse de utilizator.

O altă unealtă care poate fi de ajutor în migrarea proiectelor existente spre Andromda este schema2XMI, care generează un model XMI pe baza unei scheme de bază de date. Această unealtă creează modele de entități, care vor avea asocierile corecte pe baza foreign key-urilor și a cascadării, și permite alegerea stereotipurilor ce vor fi aplicate acestora.

4.3.2. Aplicarea principiilor MDA în Andromda

În specificațiile MDA apar conceptele de PIM (Platform Independent Model) și PSM (Platform Specific Model). Așadar, arhitecții sau analiștii de business vor crea un model PIM, care va trece apoi prin diverse transformări (manuale sau automate) pentru a deveni un model PSM ce va conține suficiente informații specifice platformei pentru a genera aplicația. În Andromda, modelarea se face la nivel PIM. Pe entitățile modelate sunt aplicate stereotipuri, care vor fi interpretate la generare de cartridge-urile alese. Acestea vor crea pe baza lor modelul PSM, care va fi livrat template-urilor ce fac efectiv procesul de generare de cod.

De exemplu, folosind stereotipul <<Service>> pe o entitate, Andromda va căuta în dicționarul intern de componente care generează cod care dintre aceste componente cunoaște acest stereotip. Dacă este folosit cartridge-ul EJB, acesta va fi apelat, deoarece conține două template-uri corespunzătoare acestui stereotip. Toate template-urile ce corespund unui stereotip vor fi apelate, ceea ce va rezulta în creeare a două clase, una abstractă care definește interfața serviciului și câteva metode utile, și una în care utilizatorul va implementa logica de business propriu-zisă. Așadar, tranziția din modelul PIM în PSM se face chiar la generare, fără ca dezvoltatorul să intervină manual, scurtând astfel procesul cu un pas.

Stereotipurile Andromda sunt independente de limbaj sau platformă. Pentru a păstra această independență, nu se folosesc tipurile standard de date, de exemplu java.lang.Long. Aceste tipuri de date se regăsesc în profilul UML AndroMDADatatypes care trebuie importat în model, și la generare se vor mapa pe tipuri specifice platformei. De exemplu un atribut String va genera un atribut java.lang.String sau Sistem.String, în funcție de limbajul folosit.

4.3.3. Conceptul de cartridge în Andromda

În Andromda, cartridge reprezintă un plug-in fundamental care are abilitatea de a procesa elemente cu stereotipuri specifice (de exemplu <<Entity>>, <<Enumeration>>, << ValueObject > >, etc) sau care anumite proprietăți ce pot fi deduse din model (de exemplu Actorii care sunt legați de un caz de utilizare sau de un serviciu). Fiecare cartridge are un descriptor în care sunt definite aceste elemente, precum și template-urile care le vor procesa.

Andromda conține următoarele cartridge-uri predefinite: BPM4Struts, jBPM, JSF, EJB, EJB3, Hibernate, Java, Meta, Spring, WebService, XmlSchema.

Pentru a le folosi, în cazul proiectelor J2EE ce utilizează Maven este suficientă adăugarea lor ca resurse în fișierul Maven principal al proiectului.

De exemplu:

```
<dependency>
    <groupId>${project.groupId}</groupId>
    <artifactId>andromda-hibernate-cartridge</artifactId>
    <version>3.0</version>
 </dependency>
```

Aceste cartridge-uri pot fi extinse, modificate, sau suprascrise cu ușurință, folosind fișiere xml de configurare.

4.3.4. Librării de translație

Folosind MDA, întotdeauna va exista un echilibru între gradul de generare de cod și nivelul de complexitate al modelului. În Andromda, această problemă aste accentuată de faptul că singurul model pe care se lucrează este cel independent de platformă; de accea pentru a obține o generare cât mai completă a codului acesta trebuie adus la un nivel destul de mare de complexitate. Deși nu este recomandat, Andromda suportă adnotări și tag-uri specifice platformei. Avantajul este că generarea codului va fi mult mai completă, dar astfel modelul va deveni dependent de tehnologia aleasă și nu mai permite flexibilitatea promisă de standardele MDA. Astfel, folosind cartridge-ul Hibernate se pot crea interogări hql direct pe model.

Pentru a compensa această problemă, a fost introdus conceputul de librării de translație. Aceste librării sunt tot plug-in-uri, dar spre deosebire de cartridge-uri, nu pot fi folosite independent, trebuie folosite cu cartridge-urile corespunzătoare. Practic, rolul lor este de a lăsa modelul independent de platformă, dar și a permite generării unei părți mai substanțiale de cod.

De exemplu, atât cartridge-ul Hibernate, cât și cel EJB permit crearea de interogări specifice direct pe model. Dar folosirea acestora nu va mai permite folosirea modelului decât cu respectivele cartridge-uri. Așadar, folosind librăria de translație Query Translation-Library, interogările exprimate prin constrângeri OCL vor fi transformate în interogări specifice (Enterprise Java Bean Query Language sau Hibernate Query Language în funcție de cartridge-ul ales).

OCL înseamnă Object Constraint Language şi este un limbaj declarativ care descrie reguli ce se aplică modelelor UML, și face acum parte din standardul UML. Se folosește pentru a exprima constrângeri și interogări pe obiecte care nu pot fi exprimate prin notații UML standard. În MDA, OCL ajută la generarea unor părți de cod care altfel nu ar fi putut fi modelate (cum sunt aceste interogări în limbaje de interogare obiectuale, sau chiar interogări SQL).

4.3.5. Mecanismul de generare a codului în Andromda

Meta-fațadele sunt obiectele care sunt folosite de orice cartridge pentru a avea acces la model. Aceste meta-fațade reprezintă un strat de abstractizare peste implementarea MOF propriu-zisă a modelului. De exemplu, modelele care implementează modele MOF diferite (UML 1.4, UML 2.0, etc.) sunt tratate în mod similar de mecanismul de generare a codului pe baza meta-fațedelor lor.

Pentru fiecare element al modelului încărcat de Andromda se instanțiază un meta-obiect Java de către implementarea MOF. Clasa acestui obiect este dedusă din meta-modelul UML (sunt meta-clase). Acest graf de meta-obiecte care reprezintă un model este denumite arbore abstract de sintaxă.

Pentru a înțelege mai bine acest mecanism, în documentația Andromda este oferit următorul exemplu:

Dacă se modelează o clasă Person ce are un atribut birthDate de tip Date, următoarele artefacte vor fi generate:

- Un obiect ce va fi o instanță a clasei UMLClass ce va avea ca atribut numele clasei, în acest caz ”Person”.
- Un obiect ce va fi o instanță a clasei Attribute, care va avea ca atribut numele atributului ”birtDate”.
- O refererință de la instanța obiectului de tip UMLClass la obiectul Attribute.
- O referință de la obiectul de tip Attribute la un obiect de tip Datatype ce descrie tipul datei atributului.

Următoarea figură va oferi modelul UML al acestor elemente:

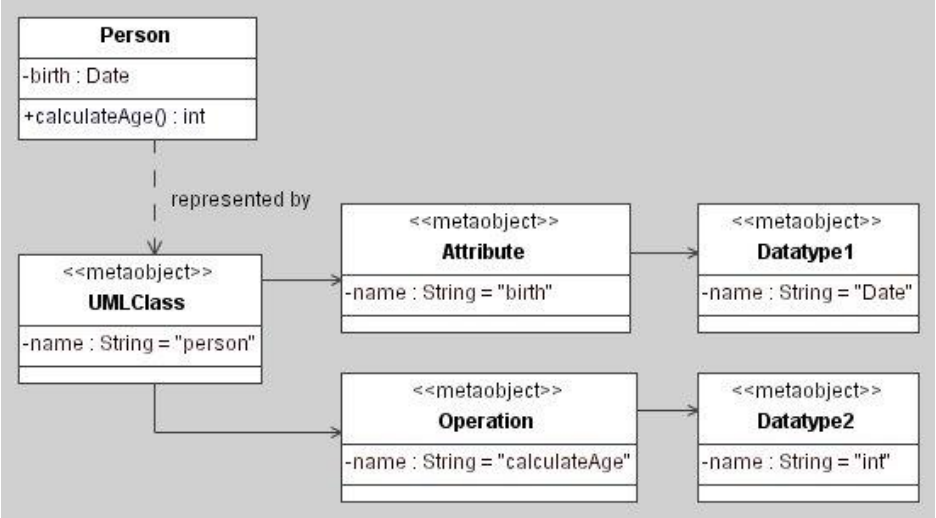

Figura 4.2: Exemplu de arbore abstract de sintaxă.

După cum se observă, aceste meta-obiecte sunt doar reprezentarea meta-informațiilor din model, fără a avea un comportament specific. Dacă în loc de UMLClass am folosi o subclasă a acestuia, numită EntityClass, creată pe bază unui stereotip <<Entity>> care ar putea de exemplu să atașeze automat entității un atribut de tip identificator dacă acesta lipsește, sau să verifice dacă numele entității nu va pune probleme la generarea codului (folosind cuvinte rezervate ale platformei specifice), ori o clasă AccessorAttribute care să atașeze automat get-ere și set-ere pentru atribute, această reprezentare ar fi mult mai inteligentă.

Andromda folosește NetBeans MDR (Metadata Repository). Acesta este [13] un depozit de meta-date care implementează MOF și deci este capabil să încarce orice meta-model MOF și să păstreze instanțe ale acestuia. Dezvoltatorul trebuie să creeze modelul UML al meta-limbajului dorit, iar NetBeans MDR va genera pe baza acestuia modelul MOF care va fi încărcat în depozit (repository). Folosind JMI (Java Metadata Interface – un standard ce specifică reguli pentru generarea API-ului Java corespunzător unui meta-model) se poate genera API-ul care va accesa acest meta-model.

Dar problema cu NetBeans MDR este că generează aceste meta-clase direct în byte-code Java la rulare, ceea ce nu permite extinderea lor pentru a oferi funcționalitatea suplimentară descrisă mai sus. Așadar, conceptul de meta-fațade a fost introdus in Andromda, pentru a compensa aceste lipsuri. Folosind designul pattern-ul Facade, se creează obiecte care abstractizează complexitatea claselor NetBeans MDR, oferind o interfață mult simplificată și independentă de versiunea meta-modelului.

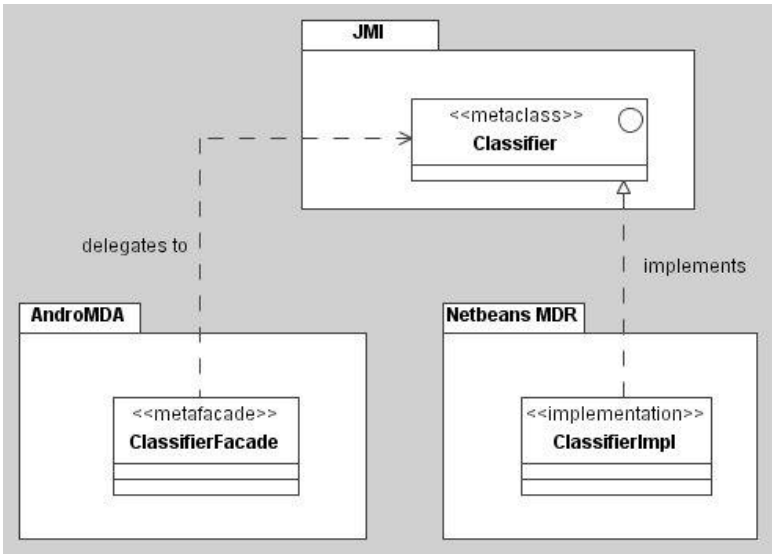

Figura 4.3: Relația între meta-fațadele Andromda cu interfața meta-claselor și implementarea lor în byte-code.

Așadar meta-fațedele ascund complexitatea meta-claselor și permit extinderea comportamentului acestora cu comportamente specifice. Există meta-fațade de bază, în nucleul Andromda, care definesc prin interfețe comportamentul meta-fațadelor specifice diferitelor modele MOF. Aceste meta-fațade de bază vor fi extinse de cartridge-uri pentru a implementa funționalități specifice. Un template dintr-un cartridge poate apela meta-fațadele proprii sau cele de bază.

Următoarea figură detaliază aceste interacțiuni:

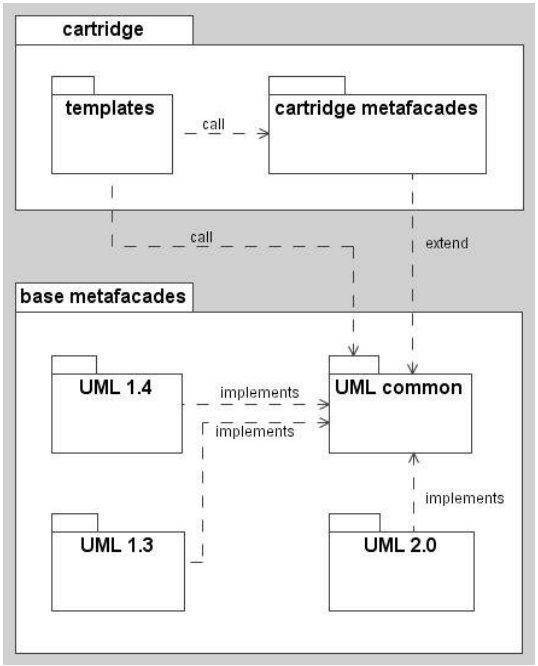

Figura 4.4: Relația dintre template-uri și meta-fațade.

Sumarizând, aceștia sunt pașii necesari pentru generarea codului (totodată pot reprezinta pașii necesari pentru crearea unui nou cartridge):

1) Modelul este parsat într-un arbore independent de sintaxă, creat din meta-obiecte.

2) Aceste meta-obiecte sunt înfășurate de meta-fațade care le abstractizează și le ascund complexitatea.

3) Meta-fațadele creează obiecte specifice platformei care conțin toate datele necesare generării codului.

4) Template-urile folosesc obiectele specifice pentru a genera codul. Andromda suportă template-uri scrise în Velocity sau Freemarker.

4.4. Arhitectura aplicațiilor J2EE create folosite Andromda

Proiectele create de Andromda pentru J2EE au o arhitectură de tip layer. Voi prezenta pe scurt carateristicile acestui arhitecturi și pe urmă modul în care sunt implementate în Andromda.

4.4.1. Arhitectura generală a aplicațiilor de tip layer

Marea majoritatea a proiectelor software enterprise sunt construite prin separarea funcționalității între diferite componente care comunică între ele. Astfel se asigură o cuplare joasă care permite schimbarea sau modificarea acestor componente, fără a afecta restul aplicației, atât timp cât respectul contractul definit între componente. Dacă aceste componente sunt organizate într-o structură de tip stivă, atunci putem vorbi de o arhitectură de tip layer (strat). Caracteristica principală a acesteri arhitecturi este dată de faptul că stratele comunică doar cu vecinii lor.

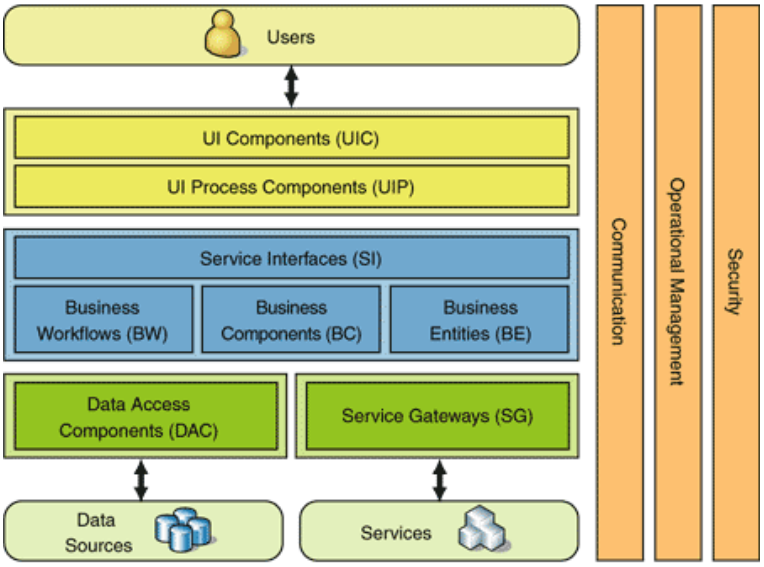

Figura 4.5: Arhitectura de tip layer [19].

Fiecare strat va comunica și va avea acces la funcționalitățile stratului de pe nivelul inferior. Astfel, sistemul păstrează o structură simplă și minimizează dependințele între diferite componente.

În mod normal, această structură arată în felul următor:

1) **Stratul de prezentare**. Acest strat conține o serie de componente de interfață prin care se realizează interacțiunea cu utilizatorul (de exemplu pagini web, formulare de tip rich-client, etc.). Aceste componente de prezentare sunt controlate de alte componente care conțin logica de prezentare și fac legătura cu stratul de servicii. De obicei interacțiunea între componentele de interfață și componentele de proces pentru interfață respectă pattern-ul Model-View-Controller.

2) **Stratul de servicii (sau business layer).** Aici este încapsulată funcționalitatea de business a aplicației și fiecare serviciu îndeplinește a anumită funcționalitate specifică. Acest strat are următoarele componente (opționale): interfața serviciului, workflow-ul procesului, componentele de business și entitățile de business.

Interfața serviciului este stratul din fața componentelor de business care expune doar funcționalitățile esențiale ale acestora, pentru a ascunde complexitatea ne-necesară. Dacă exteriorul comunică doar cu aceste interfețe, putem vorbi de o arhitectură orientată pe serviciu, sau SOA. Aceste interfețe pot fi și servicii web, pentru a oferi respectivele servicii și lumii exterioare.

Componentele de business sunt definite în [10] astfel: ”Implementarea software a unui concept autonom de business sau a unui proces de business. Sunt compuse din toate artifactele software necesare pentru a reprezenta, implementa și oferi un concept dat de business ca un element autonom și reutilizabil al unui sistem distribuit mai complex.” Așadar aceste componente reprezintă realizarea software a conceptelor de business și încapsulează regulile de business.

Workflow-ul procesului reprezintă la un macro-nivel activitățile pe care business-ul le realizează. De exemplu: plasarea unei comenzi on-line, aplicarea la un credit, efectuarea unei tranzacții. Aceste procese sunt implementate prin componente de work-flow care orchestrează una sau mai multe componente de business pentru indeplinirea respectivului proces.

Entitățile de business sunt elemente care incapsulează datele din diferite surse pentru a le oferi un tratament similar în componentele și procesele de business, indiferent de proveniența lor (pot fi oferite de stratul de persistență de pe nivelul inferior, sau pot fi aduse în format XML apelând servicii web ale altei aplicații).

3) **Stratul de access la date.** Acest strat oferă un API simplu pentru a accesa și manipula datele. Aceste componente abstractizează semantica tehnologiilor specifice de păstrare a datelor, ceea ce ajută stratul de servicii să se concentreze pe logica de business, nu pe particularitățile de implementare ale unei tehnologii specifice. De obicei, pentru fiecare entitate de business sunt oferite metode prin care se realizează operațiile de tip CRUD (creare, citire, modificare și ștergere). Astfel, tot codul care se referă la o anumită entitate este concentrat într-un singur loc, ceea ce va ușura mentenanța și eventualele schimbări.

4) **Data stores**. Aici se păstrează efectiv datele; tipuri comune de data store sunt bazele de date relaționale sau sistemul de fișiere.

Pot exista și servicii fundamentale care să nu facă parte dintr-un strat anume, și să fie folosite în toate celelate straturi. Acestea pot fi încadrate în trei categorii: securitate, management operațional (care fac atât managementul resurselor și componentelor, precum și îndeplinirea cerințelor non-funcționale ca scalabilitate și rezistență la defecte) și comunicare (servicii care permit accesul resurselor remote, fie prin apel direct sau mesaje asincrone, fie prin servicii web).

Un mare avantaj al acestei arhitecturi, pe lângă toate cele amintite mai sus, este faptul că permite separarea aplicației pe una sau mai multe mașini fizice în funcție de nevoile de performanță.

4.4.2. Implementarea arhitecturii layer în Andromda

Andromda suportă în momentul de față generarea semnificativă de cod pentru toate straturile prezentate folosind diferite cartridge-uri.

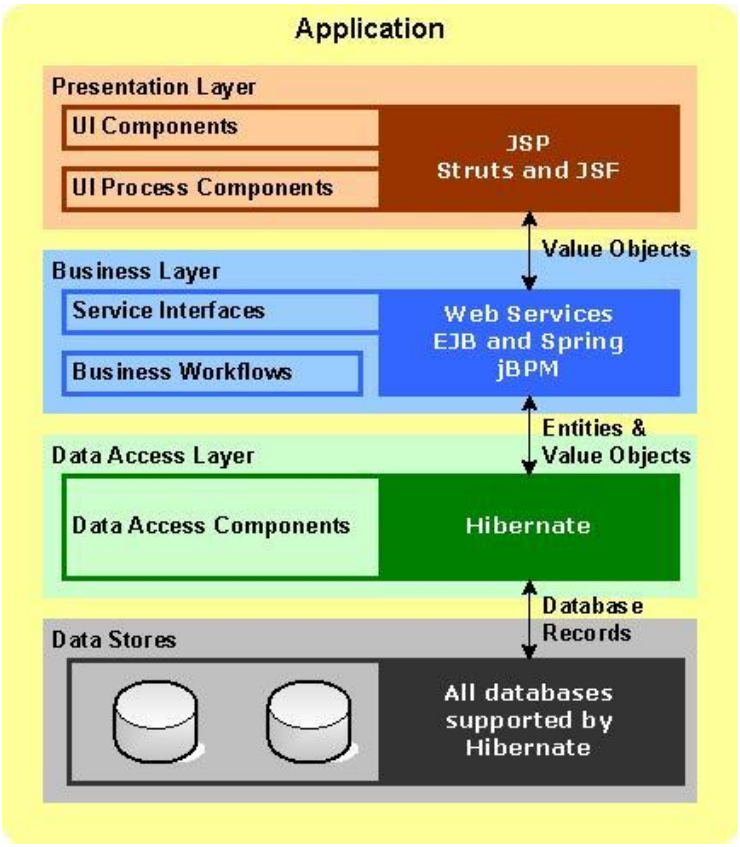

Figura 4.6: Posibile variante de arhitecturi de tip layer folosind Andromda.

Pentru stratul de prezentare Andromda oferă posibilitatea folosirii framework-urilor JSF sau Struts. Pentru generarea codului se folosec diagrame de activitate care

specifică cazurile de utilizare, flow-ul paginilor în acestea, comportamente și componentele de interfață, precum și tipurile de utilizatori și accesul acestora la resurse.

Pentrul stratul de servicii în Andromda se folosește de regulă cartridge-ul Spring. Acesta va realiza managementul tuturor serviciilor și a sesiunilor și va crea, dacă se dorește componente de tip EJB pentru fiecare serviciu, caz în care trebuie să instalăm aplicația într-un container EJB (de exemplu JBoss). De asemenea va oferi pentru fiecare entitate de tip serviciu o clasă ce conține semnătura tuturor metodelor definite pe model în care dezvoltatorul va scrie codul ce realizează respectivele funcționalități. Așadar singurul lucru pe care trebuie să îl facem este să scriem efectiv logica de business, iar Andromda se va ocupa de restul codului structural. Folosind cartridge-ul de web servicii, toate aceste metode vor putea fi expuse ca web-servicii, aplicând un singur stereotip și folosind același cod. Dacă pe un serviciu se aplică stereotipul <<WebService>>, atunci toate metodele definite pe acel serviciu vor fi expuse, sau dacă se dorește expunerea doar anumitor metode, acestea vor fi marcate cu <<WebServiceOperation>>. Andromda oferă și posibilitatea generării de work-flow-uri de business pentru jBPM (tool ce face parte din suita JBoss).

Stratul de acces la date folosește cel mai popular framework ORM (Object Relation Mapping), și anume Hibernate. Folosind cartridge-ul cu același nume se generează obiecte de access la date (DAO – Data Access Objects) pentru fiecare din entitățile de business modelate. În stadiu de dezvoltare este și cartridge-ul pentru Seam. Așadar, orice aplicație care folosește Andromda poate utiliza orice bază de date suportată de Hibernate.

4.4.2.1. Propagarea datelor între straturi

O să urmărim propagarea datelor începând de la nivelul bazelor de date, până în stratul de prezentare. Așadar, datele sunt păstrate în baze de date relaționale, în înregistrări din tabele. Aceste înregistrări sunt preluate de stratul de persistență și tranformate în obiecte ce reprezintă entități din domeniul de business cu ajutorul framework-ului Hibernate. Stratul de persistență va trimite aceste obiecte de business serviciilor unde se efectuează efectiv operațiile și logica de business. O dată ce operațiile și algoritmii necesari au fost încheiați, datele încapsulate în aceste entități vor fi transformate în alte obiecte de tip JavaBean, care vor fi trimise stratului de prezentare. Aceste obiecte se numesc în Andromda ”Value Objects” și sunt doar simple încapsulări de date; nu au nici un fel de logică atașată. Singurele proprietăți ale acestora pot fi tipuri de date primitive sau wrapper-ele acestor tipuri primitive, alte obiecte de tip Value Object, sau colecții de tipuri primitive sau Values Objects. Aceste proprietăți vor fi accesate prin get-ere și set-ere generate. Pentru a crea aceste obiecte este suficientă specificarea stereotipului <<ValueObject>> pe entitățile respective. Pentru a evita posibilitatea transmiterii entităților de business dincolo stratul de servicii, Andromda nu permite serviciilor să returneze decât tipuri primitive sau Value Objects ori colecții ce conțin aceste elemente.

Faptul că entitățile de business nu pot trece dincolo de stratul de servicii are câteva avantaje majore:

- În primul rând, logica de business rămâne astfel conținută strict în stratul de business, altfel ar exista tentația de a manipula aceste obiecte în stratul de prezentare rezultând în fragmentarea logicii în mai multe părți (în special dacă aplicația are mai mulți clienți diferiți de prezentare). Astfel nu crește doar ușurința mentenanței, dar și siguranța ca entitățile nu au fost manipulate și modificate în afară, ceea ce simplifică procesul de dezvoltare și scade posibilitatea apariției defectelor.

- Clientul va primi doar ceea ce are nevoie și nu se vor transmite date inutile. Pentru fiecare element de interfață diferit, chiar dacă se referă la aceeași entitate, se pot defini Value Objects diferite, cu diverse nivele de detaliu. Spre exemplu, într-un tabel se afișează ceilalți utilizatori conectați la aplicație pentru a permite conversația instantă între aceștia. Probabil este nevoie doar de numele lor, numele de utilizator și eventual o poză sau un scurt text despre aceștia. Folosind Value Objects, aplicația va primi strict aceste elemente, ceea ce micșorează considerabil cantitatea de informație, de multe ori inutilă transmisă între straturi. Desigur, pentru pagina de administrator se poate construi un alt Value Object care va include și adresa de email, ultima dată de logare, numărul de accesări ale aplicației pe lună, sau orice altă informație considerată necesară.

- Este și o măsură de siguranță pentru a nu oferi date sensibile clienților nesecurizați.

Pentru a ușura munca suplimentară introdusă de folosirea acestor obiecte, Andromda oferă și suport pentru transformarea între entitățile de business și Value Objects, folosind obiectele de access la date. Astfel, clasa abstractă a unui DAO va mapa automat tipurile primitive ale entității cu cele ale Value Objectului ce au același nume, urmând ca doar celelalte elemente să fie mapate manual de programator.

4.4.2.2. Serviciile și sesiunile Hibernate

Un alt concept important despre aplicațiile generate cu Andromda este faptul că managementul sesiunilor Hibernate este făcut automat de către codul generat; asta eliberează programatorul de obligativitatea de a crea o sesiune la fiecare tranzacție cu baza de date, de a avea grijă de tranzacții, si de a închide sesiunea la sfârșit. Această sesiune Hibernate este un obiect care permite aplicației, atâta timp cât ea este deschisă, să efectueze operații CRUD asupra entităților din baza de date. Starea obiectului este ”atașată”, și asta înseamnă că graful de obiecte poate fi parcurs (adică obiectele de care entitatea depinde sau cu care are o relație pot fi accesate). Dacă respectivele obiecte nu sunt deja în memorie, Hibernate le va încărca automat (lazy loading), pentru a nu ține în memorie un graf complex de obiecte dacă acestea nu vor fi folosite. Cu toate acestea, odată ce sesiunea va fi închisă, starea entității va deveni detașată. Adică obiectul încă va fi prezent în memorie, și i se pot accesa proprietățile, dar modificările nu mai pot fi persistate decât dacă se redeschide sesiunea, și obiectele de care această entitate depinde nu vor mai fi încărcate dacă nu se află deja în memorie. Dacă se încearcă accesarea unor

elemente ale unui obiect detașat care nu se află deja în memorie se va arunca o excepție de tip LazyInitializationException.

Așadar, când se apelează o metodă dintr-un serviciu, se creează automat o sesiune Hibernate, care este închisă la ieșirea din metodă. Cu alte cuvinte, durata de viață a sesiunii ține atâta timp cât se executa respectiva metodă. Așadar entitățile de business vor fi atașate pe durata executării metodei, dar vor deveni detașate la încheierea ei. Acesta este un alt motiv pentru care Andromda nu permite returnarea entităților de business ca urmare a apelului unei metode din stratul de servici, pentru a evita accesarea elementelor ce nu se află în memorie; metoda recomandată de transfer a datelor este cea a creării unui Value Object.

4.5. Maven

Deoarece orice proiect generat în mod standard cu Andromda utilizează Maven, voi face o scurtă prezentare a acestei unelte foarte utile de build și management al resurselor proiectului.

Maven este o unealtă al cărei scop este simplificarea procesului de build al unui proiect Java. Acestea sunt modalitățile prin care Maven încearcă să îndeplinească acest obictiv:

- Procesul de build ar trebui să fie cât mai ușor.
- Oferirea unui sistem de build uniform și similar între proiecte diferite.
- Oferirea de informații calitative despre proiect.
- Oferirea de sfaturi pentu aderarea la principiile de bună practică.
- Posibilitatea de a integra cu ușurință noi funcționalități.

Pentru a oferi un proces ușor de build și uniform între proiecte diverse, a fost introdus conceptul de Project Object Model, sau POM. Fiecare proiect va folosi un fișier xml numit pom.xml ce reprezintă meta-data proiectului. Aceste meta-date includ build-ul, dependințele, și informații generale despre resursele proiectului. Aceste informații sunt folosite de entitățile numite ”goals”. Un ”goal” (traducere: scop, obiectiv, misiune, intenție, etc.) este o funcție executabilă care acționează asupra unui proiect. Sunt scrise într-un limbaj de scripting, și pot fi specifice proiectului sau reutilizabile. Dacă vedem tot sistemul din punct de vedere Orientat Obiect, putem asocia pom-ul și meta-datele din el cu un obiect, iar ”goal”-urile cu metodele care acționează asupra respectivului obiect. Maven are câteva ”goal”-uri standardizate pentru realizarea sarcinilor specifice de build. Acestea sunt:

- **clean**. Curăță proiectul de toate fișierele generate la build-ul anterior.
- **install**. Compilează și asamblează proiectul și îl instalează în repository-ul local. În acest fel poate fi folosit ca dependințe pentru alt proiect.
- **compile**. Compilează codul sursă al proiectului.
- **package**. Ia codul compilat și îl asamblează într-un format distributibil (JAR, WAR, EAR).
- **validate**. Verifică validitatea proiectului și faptul că toate dependințele sunt disponibile.
- **test**. Testează codul sursă folosind un framework de unit test inclus în proiect. Aceste teste nu ar trebui să necesite asamblarea sau instalarea proiectului.

- **deploy**. Copiază pachetul final în repository-ul remote.
- **site**. Generează documentația în format html a proiectului.

Cel mai mare avantaj al folosirii Maven este faptul că odată ce ne familiarizăm cu procesul de build al unui proiect, o să putem aplica aceste cunoștințe pentru orice alt proiect, ceea ce ne salvează foarte mult timp atunci când trecem la un alt proiect ce folosește Maven.

Maven oferă o mulțime de informații utile despre proiect, în parte luate din pom, sau generate direct din sursele proiectului, cum ar fi liste și grafuri complete de dependețe, rapoarte de testare ce includ procentul de acoperire, și altele.

Câteva principii de bună practică pe care Maven le promovează se referă la partea de testare automată. De exemplu, specificarea, execuția și rapoartele unităților de test sunt foarte bine suportate și recomandate de Maven. Maven încurajează păstrarea unităților de testare într-un arbore separat, dar similar cu cel al surselor și utilizarea convenției de nume pentru localizarea și executarea testelor. Dar principiile de bună practică se referă și la structura proiectului, pentru a păstra structuri similare pentru proiecte diferite și a ușura tranziția dezvoltatorilor. De asemenea, se oferă și ajutor în work-flow-ul proiectului, cum ar managementul release-urilor și issue tracking.

Maven are noțiunea de moștenire; astfel un proiect poate fi compus din mai multe module, fiecare modul având propriul fișier pom. Trebuie să existe un pom principal, rădăcină, care va conține lista pom-urilor secundare. Avantajul este că toate fișierele pom copii vor putea accesa proprietățile, dependințele și plug-in-urile definite în pom-ul principal, ceea ce crește gradul de reutilizare.

Maven se bucură de un suport foarte mare al comunității open-source, de accea are acces la un număr foarte mare de librării, meta-date și artefacte în mod standard. De asemenea este ușor extensibil de utilizator, plug-in-uri noi putând fi create utilizând Java sau limbaje de scripting. De asemenea, este capabil să construiască proiectul într-o mare varietate de formate, de exemplu JAR, WAR, EAR, etc. folosind doar meta-datele proiectului. Pe lângă construirea efectivă a proiectului, folosind aceleași meta-date și adnotări pentru javadoc, Maven creează și documentație web în format HTML sau fișiere pdf. Maven încurajează folosirea unui repository, care este un folder în care se vor păstra librăriile de care depinde proiectul, precum și se vor livra artefactele construite. Acest repository poate fi distribuit, și accesat de oricine în rețea, sau păstrat local.

5. Proiectare de detaliu si implementare

5.1. Structura unui proiect generat folosind Andromda

Pentru a genera un nou proiect folosind Andromda, trebuie apelat plug-in-ul de generare pentru Maven, cu următoarea comandă din linie de comandă:
mvn org.andromda.maven.plugins:andromdapp-maven-plugin:3.4-SNAPSHOT:generate.

Proiectul va fi generat după ce utilizatorul îl configurează pe baza opțiunilor posibile. Acestea sunt:

- Versiunea de UML folosită (1.4, 2, emf2.2).
- Tipul aplicației (EAR, WAR).
- Cartridge-ul tranzacțional și de persistență dorit (hibernate, ejb, ejb3, spring, none).
- Limbajul de programare folosit pentru servicii și persistență (Java, Groovy).
- Tipul bazei de date utilizate (h2, hypersonic, mysql, oracle, db2, informix, mssql, pointbase, postgres, sybase, sabdb, progress, derby).
- Includerea capabilităților de workflow (jBPM) – yes, no.
- Folosirea unei interfețe web – yes, no.
- Framework-ul dorit pentru generarea interfeței dacă aceasta există (JSF, Struts).
- Posibilitatea expunerii metodelor din servicii ca web-servicii – yes, no.

Pe baza acestor opțiuni, Andromda va configura noul proiect și va include automat cartridge-urile necesare. Rezultatul va fi un proiect creat din mai multe module, fiecare având un fișier pom propriu, plus unul general.

Așadar structura acestuia va fi următoarea:

1) **Proiectul rădăcină**, ce va conține toate celelate module, controlează procesul de build și conține proprietățile comune.

2) **mda**. Acesta este modulul ce conține modelul UML al aplicației și fișierul de configurare al Andromda, prin care se pot configura atât opțiuni generale, cât și pentru fiecare cartridge inclus în parte. Tot aici vor fi incluse și eventualele xml-uri prin care se extinde comportamentul cartridge-urilor, sunt incluse resurse suplimentare, sau se realizează mapări (de exemplu pentru tipuri de date, servicii web, acțiuni).

3) **common**. Acest modul este suprascris la fiecare regenerare a codului și conține resursele și entitățile comune cu alte sub-module. De exemplu, aici vor fi incluse obiectele de tip Value Object, enumerațiile, interfețele serviciilor și excepțiile create de utilizator.

4) **core**. Aici se vor regăsi toate resursele și clasele controlate de framework-ul Spring, Hibernate și EJB. Vor fi generate atât sursele care necesită implementare manuală de către utilizator, ce sunt create doar o singură dată și nu vor mai rescrise, cum sunt implementarea serviciilor, a entităților și obiectele de acces la date (DAO), cât și

entitățile Hibernate, maparea lor la baza de date, clasele abstracte care vor fi extinse de implementările în care utilizatorul scrie cod, alte clase și resurse folosite de Spring, precum și fișierele prin care se creează și se șterge schema bazei de date. Toate aceastea din urmă sunt recreate la fiecare generare de cod.

5) **web**. Acest modul colectează toate resursele și clasele care vor forma stratul de prezentare. Vor fi atât fișiere pe care utilizatorul le editează și implementează manual, cât și cod generat de framework-ul de prezentare ales (JSF sau Struts).

6) **app**. Aici se vor colecta toate resursele și clasele necesare pentru a construi aplicația finală in format EAR sau WAR. După build, aici se va regăsi aplicația finală, urmând să fie copiată în containerul unde va fi folosită.

7) **webservice**. Acesta este un modul opțional, care se creează dacă se dorește expunerea metodelor din servicii ca servicii web.

Un alt lucru important este faptul că orice modul are 2 foldere: src și target. În folderul src vor fi păstrate toate sursele care trebuie editate de dezvoltator; acestea sunt generate doar o dată și nu vor fi suprascrise. Folderul target este cel în care se păstrează toate artefactele care trebuie recreate la fiecare generare de cod, și unde Maven va copia toate resursele din src după compilare. Arhiva finală ce va conține proiectul deploy-abil în container sau server va fi create doar din aceste foldere target. Dacă se dorește adăugarea resurselor suplimentare în target, de exemplu pagini jsp, fișiere css sau imagini, acestea nu trebuie puse în target deoarece vor fi șterse la regenerare. Trebuie plasate în folderul src, respectând structura folderului target. Astfel vor fi copiate automat de Maven în locul dorit.

5.2. Aspectele Andromda urmărite și testate prin aplicația aleasă

Aplicația va încerca să acopere cât mai mult din capabilitățile Andromda de generare cod pentru stiva J2EE. Pentru aceasta se va analiza impactul Andromda pentru fiecaruia din cele trei straturi ce compun o arhitectură de tip layer, precum și integrarea unui serviciu global de securitate. Așadar, următoarele elemente vor fi create:

- **Un strat de persistență** ce va folosi cartridge-ul Hibernate pentru maparea entităților de business pe tabelele ce vor conține efectiv datele, precum și clase ajutătoare (numite în continuare DAO) care au ca scop oferirea unui API simplu pentru operațiile CRUD asupra entităților de business.

- **Un strat de servicii** ce va utiliza cartridge-ul Spring pentru managementul automat al EJB-urilor care vor interfața serviciile efective, al sesiunilor și obiectelor Hibernate, și va oferi accesul la aceste elemente folosind principiile inversiunii controlul și a injecției dependințelor, principii explicate în detaliu în secțiunea de prezentare a framework-ului Spring. De asemenea, va oferi, prin integrarea framework-ului Spring Security, securitatea globală a aplicației și acces la resurse pe baza cazurilor de utilizare

și a rolurilor definite pentru utilizatori. Comunicarea cu stratul de prezentare se va face pe baza obiectelor de tip Value Object.

- **Ultimul strat va fi cel de prezentare**, care va folosi cartridge-ul JSF pentru a genera din diagrame de activitate UML paginile web efective cu toate elementele acestora, interacțiunea dintre ele (flow-ul paginilor), elementele de control și va comunica cu stratul de servicii pentru accesul la date.

5.3. Detalierea cazurilor de utilizare

Caz de utilizare 1: Logarea în sistem.

Actori principali: Client, dezvoltator, project manager.

Stakholderi și interese:
- Orice utilizator al aplicației care dorește accesul la aplicație.

Precondiții:
- Utilizatorul să aibe creat un cont și acesta să fie activ.

Postcondiții:
- Utilizatorul a intrat în sistem și are acces la resursele acestuia în funcție de tipul său (client, dezvoltator, project manager).

Succesiunea pașilor:
1. Actorul introduce numele de utilizator.
2. Actorul introduce parola.
3. Sistemul verifică aceste date.
4. Utilizatorul intră în sistem.

Succesiunea alternativă a pașilor:

4a. Datele introduse nu sunt valide (nu există numele de utilizator introdus sau parola nu se potrivește).
1. Sistemul afișează un mesaj de eroare.

4b. Contul nu mai este activ.
1. Sistemul afișează un mesaj de eroare.

Caz de utilizare 2: Vizualizarea fișelor de timp pentru un anumit proiect.

Actori principali: Client.

Stakholderi și interese:
- Clientul care dorește verificarea orelor petrecute pe proiect și a progresului acesta.
- Project manager care dorește ca clienții să fie mulțumiți prin transparența activităților pe proiect.

Precondiții:
- Clientul să aibe cel puțin un proiect asignat.

Postcondiții:

- Clientul va avea acces la totalitatea fișelor de timp aprobate de project manager pentru vizualizarea lor.

Succesiunea pașilor:

1. Clientul selectează proiectul pentru care dorește vizualizarea fișelor de timp.
2. Clientul poate filtra rezultatele după dată și sarcină.
3. Sistemul va afișa fișele de timp pentru proiectul selectat în funcție de filtrele selectate.

Caz de utilizare 3: Creare unui proiect nou.

Actori principali: Project manager.

Stakholderi și interese:

- Project manager, care adaugă un proiect în sistem pentru a se putea urmări evoluția timpilor petrecuți de dezvoltatori lucrând la acesta.
- Client, care dorește transparența orele petrecute lucrându-se la proiectul său pentru o facturare corectă.
- Dezvoltatori, care vor trebui să completeze orele lucrate pe proiect.

Precondiții:

- Project managerul să fie logat în sistem.

Postcondiții:

- Un nou proiect a fost creat pentru clientul ales.

Succesiunea pașilor:

1. Project managerul selectează opțiunea de creare a unui proiect nou.
2. Acesta introduce datele necesare (numele proiectului și clientul - selectat dintr-o listă).
3. Sistemul creează noul proiect, care apare în lista proiectelor project managerului.

Succesiunea alternativă a pașilor:

2a. Project managerul se răzgândește și anulează operația.
    1. Starea sistemului rămâne neschimbată.

3a. Project managerul nu completează toate datele necesare.
    1. Sistemul va atenționa project managerul că lipsesc anumite date.
    2. Acesta va completa datele lipsă

3b. Numele proiectului nu este unic pentru acest project manager.
    1. Sistemul va atenționa project managerul că există deja un proiect cu acest nume.
    2. Acesta va schimba numele proiectului.

Caz de utilizare 4: Crearea unei sarcini pentru un proiect.

Actori principali: Project manager.

Stakholderi și interese:

- Project manager care va împărți proiectul în urma analizei cu echipa de dezvoltare în sarcini atomice.
- Client care conștientizează astfel ce sarcini presupune realizarea proiectului pe care îl dorește.
- Dezvoltatorul care va avea de îndeplinit această sarcină.

Precondiții:

- Proiectul să existe în sistem.

Postcondiții:

- Proiectul va avea o nouă sarcină, care va fi asignată unui dezvoltator.

Succesiunea pașilor:

1. Project managerul selectează opțiunea de a crea o sarcină nouă.
2. Project managerul va completa datele necesare pentru a crea noua sarcină (numele sarcinii și dezvoltatorul căreia îi va fi asignată).
3. Sistemul va crea o nouă sarcină care va apărea în lista de sarcini a proiectului și în lista de sarcini pe care dezvoltatorul o are de îndeplinit.

Succesiunea alternativă a pașilor:

2a. Project managerul se răzgândește și anulează operația.
    1. Starea sistemului rămâne neschimbată.

3a. Project managerul nu completează toate datele necesare.
    1. Sistemul va atenționa project managerul că lipsesc anumite date.
    2. Acesta va completa datele lipsă.

Caz de utilizare 5: Căutarea fișelor de timp pentru un proiect.

Actori principali: Project manager.

Stakholderi și interese:

- Project managerul care dorește vizualizarea anumitor informații despre fișele de timp și/sau prelucrarea acestora (cazul de utilizare 6).

Precondiții:

- Project managerul să fie logat în sistem.

Postcondiții:

- Project managerul va putea vizualiza fișele de timp în funcție de opțiunile pe care le alege.

Succesiunea pașilor:

1. Actorul selectează opțiunea de căutare.
2. Actorul poate selecta anumite opțiuni pentru a filtra rezultatele. Acestea sunt:
    - Data creării.
    - Sarcina.
    - Dezvoltator.
    - Status.
3. Sistemul returnează fișele în funcție de filtrele aplicate.

Succesiunea alternativă a pașilor:

2a. Actorul se răzgândește și anulează operația.

1. Starea sistemului rămâne neschimbată.

Caz de utilizare 6: Schimbarea statusului unei fișe de timp.

Actori principali: Project manager.

Stakholderi și interese:
- Project manager care va organiza fișele de timp.
- Clienții care vor vedea noile fișele de timp aprobate.
- Dezvoltatorii care trebuie să modifice fișele de timp respinse.

Precondiții:
- Fișa de timp să existe în sistem.
- Fișa de timp să aibe statusul ”submitted”

Postcondiții:
- Fișa de timp va avea un nou status (”approved” sau ”rejected”).

Succesiunea pașilor:
1. Project managerul selectează o fișă de timp găsită în urma cazului de utilizare 5.
2. Project managerul va schimba statusul acesteia.
3. Sistemul va aplica schimbarea și fișa va avea un nou status (aprobată sau respinsă).

Succesiunea alternativă a pașilor:

2a. Project managerul se răzgândește și anulează operația.

2. Starea sistemului rămâne neschimbată.

Caz de utilizare 7: Crearea unei noi fișe de timp.

Actori principali: Dezvoltator.

Stakholderi și interese:
- Project manager care urmărește activitatea dezvoltatorilor.
- Dezvoltator care dorește justificarea muncii sale.
- Client care dorește urmărirea progresului proiectului.

Precondiții:
- Dezvoltatorul să fie logat în sistem.

Postcondiții:
- Va fi creată o nouă fișă de timp.

Succesiunea pașilor:
1. Dezvoltatorul selectează opțiunea de a crea o nouă fișă de timp.
2. Dezvoltatorul va completa datele necesare pentru creare unei noi fișe de timp:
    - Sarcina.
    - Data.
    - Numărul de ore petrecute.

- Statusul (draft pentru salvarea ce ii permite editarea ulterioară, submitted pentru trimiterea spre validate a project managerului).
- Comentarii dacă este cazul.

3. Sistemul va crea o nouă fișă de timp cu opțiunile selectate.

Succesiunea alternativă a pașilor:

2a. Dezvoltatorul se răzgândește și anulează operația.
1. Starea sistemului rămâne neschimbată.

3a. Dezvoltatorul nu completează toate datele necesare.
1. Sistemul va atenționa dezvoltatorul că lipsesc anumite date.
2. Acesta va completa datele lipsă.

Caz de utilizare 8: Vizualizarea fișelor de timp.

Actori principali: Dezvoltator.

Stakholderi și interese:

- Dezvoltator care dorește vizualizarea fișelor personale de timp.

Precondiții:

- Dezvoltatorul să fie logat în sistem.

Postcondiții:

- Dezvoltatorul va putea vizualiza fișele lui de timp în funcție de filtrele selectate.

Succesiunea pașilor:

1. Dezvoltatorul selectează opțiunea de a vizualiza fișele de timp.
2. Dezvoltatorul poate completa următoarele filtre pentru a restrânge rezultatele:
   - Sarcina.
   - Data creării.
   - Status.
3. Sistemul va afișa fișele de timp ale acestui utilizator în funcție de filtrele aplicate.

Succesiunea alternativă a pașilor:

2a. Dezvoltatorul se răzgândește și anulează operația.
1. Starea sistemului rămâne neschimbată.

Caz de utilizare 9: Modificare unei fișe de timp.

Actori principali: Dezvoltator.

Stakholderi și interese:

- Dezvoltator care dorește modificarea unei fișe personale de timp.

Precondiții:

- Fișa să existe în sistem.
- Fișa să aibe statusul ”draft” sau ”rejected”.

Postcondiții:

- Fișa selectată va fi modificată în funcție de noile opțiuni alese.

Succesiunea pașilor:

1. Dezvoltatorul selectează o fișă de timp găsită în urma cazului de utilizare 8.
2. Dezvoltatorul va putea aduce modificări numărului de ore lucrate, sarcinii de care este legată și a statusului.
3. Sistemul va salva schimbările aduse fișei de timp.

Succesiunea alternativă a pașilor:

2a. Dezvoltatorul se răzgândește și anulează operația.

1. Starea sistemului rămâne neschimbată.

5.4. Modelarea domeniului problemei

Voi începe modelarea aplicației prin crearea modelului de domeniu. Acest lucru va determina crearea tuturor entităților care vor face parte din domeniul problemei, a atributelor acestora, precum și a relațiilor dintre ele. Conform [12] acesta este cel mai important artefact care rezultă în urma analizei orientate obiect. Aceasta se realizează pe cazurile de utilizare, pentru a transforma informațiile cuprinse în acestea, care nu sunt prin definiție orientate obiect, în modelul ce va descrie aplicația din punct de vedere orientat obiect. Acest model nu va cuprinde interacțiunile și operațiile obiectelor, ci doar elementele enumerate mai sus.

Strategiile recomandate pentru a captura aceste clase conceptuale sunt:

- Listarea categoriilor claselor conceptuale.
- Identificarea substantivelor care apar în cazurile de utilizare.

Al doilea pas este identificarea relațiilor dintre acestea, urmată de completarea claselor cu atribute. Aceste atribute trebuie să fie atribute simple sau tipuri de date [12], și nu alte alte clase. Relația dintre clase trebuie modelată ca asociere, nu ca atribut.

Așadar, lista claselor conceptuale pentru aplicația de față este următoarea:

- User
- Timecard
- Task
- Project
- Role
- Status

După stabilirea relațiilor între ele și a atributelor, modelul UML rezultat este următorul:

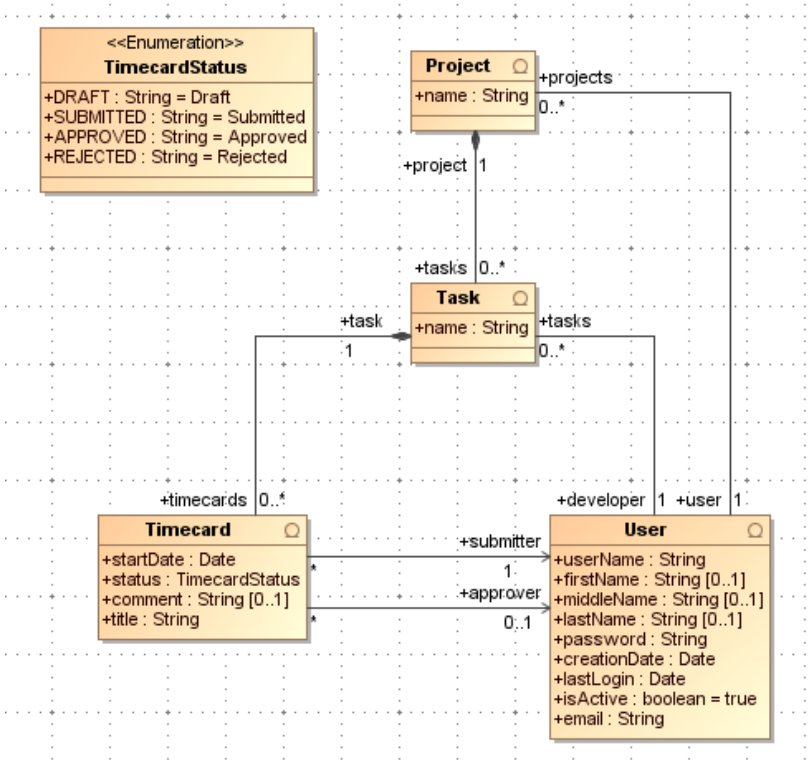

Figura 5.1: Modelul UML al claselor conceptuale.

Deoarece starea unei fișe de timp este un String care face dintr-o mulțime finită și bine cunoscută, nu a mai fost necesară crearea unei noi clase care să aibe ca atribut doar numele stării, de accea am modelat acest atribut ca o enumerare.

Așadar acesta este modelul UML care oricum trebuia făcut pentru a descrie sistemul. Să vedem cum putem să îl folosim pentru a genera codul efectiv utilizând Andromda. În primul rând este evident că aceste obiecte trebuie persistate într-o bază de date. Pentru a realiza acest lucru trebuie să adăugăm claselor stereotipul <<Entity>> (care face parte din profilul UML Andromda). Deoarece Andromda, pe baza acestui stereotip va genera tabele MySql (baza de date aleasă pentru implementare) corespunzătoare tuturor entităților care îl folosesc, va genera și un tabel USER. Dar acesta este un cuvând rezervat în MySql, și va genera o eroare. Pentru a evita acest lucru, putem folosi tag-ul andromda_persistence_table pentru a specifica numele cu care dorim să fie creat tabelul, fără a fi nevoie să modificăm numele obiectului. Folosind alte tag-uri putem specifica de asemenea și numele coloanelor sau al cheilor străine dacă acest lucru va fi necesar.

Un alt lucru de care trebuie să ținem seama dacă utilizăm modele UML 2.0 este faptul că toate atributele sunt implicit create cu atributul UML unique. Acesta va determina ca toate coloanele SQL corespunzătoare să aibă constrângerea UNIQUE aplicată, lucru care nu este de dorit în toate situațiile. Pentru acestea trebuie să marcăm fiecare atribut care nu vrem să unic cu atributul UML nonunique.

Acestea sunt singurele modificări ce trebuie aduse modelului pentru ca Andromda să poată genera stratul de persistență al aplicației.

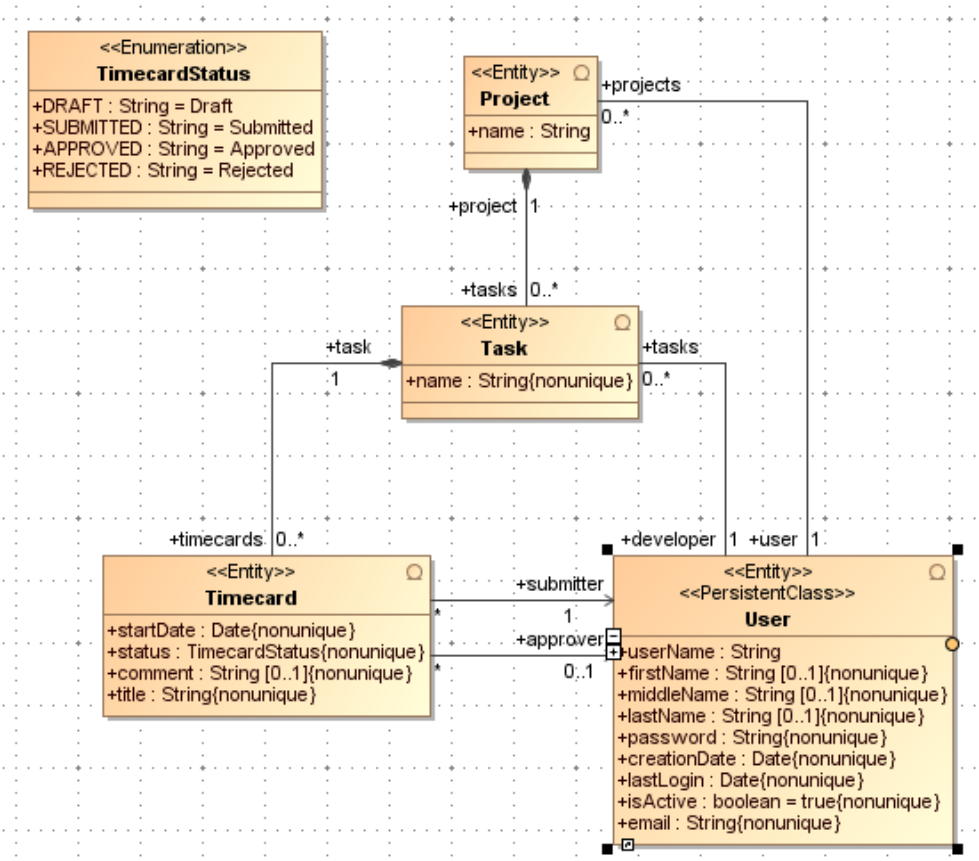

Figura 5.2: Modelul UML al claselor ce vor fi generate folosind Andromda.

Așadar, schimbările sunt foarte puține și ușor de realizat. Rezultatul este că toate aceste clase ce au stereotipul <<Entity>> vor reprezenta entități de business, și vor fi accesibile stratului de servicii pentru a performa operații asupra lor. Graful acestor obiecte va reprezenta structura bazei de date, iar scriptul pentru a crea schema și toate tabelele, constrângerile și cheile străine va fi generat automat. De asemenea, vor fi create fișierele Hibernate care mapează entitățile la baza de date și clase de tip DAO care permit operații CRUD asupra entităților, precum și eventuale transformări ale acestora în obiecte de tip Value Objects. Tot codul care realizează operațiile CRUD, precum și parțial codul care face transformarea entităților va fi generat automat într-o clasă abstractă, iar utilizatorul va trebui doar să completeze în implementarea ei concretă codul de transformare.

După cum se observă, nici unul dintre obiectele modelate nu are o proprietate de identificare unică (de exemplu un id, sau un alt cod numeric). Cu toate acestea, Andromda va genera pentru fiecare un id de tip Long auto-incrementabil care va constitui cheia primară a tabelelor, pe baza cărora se vor face și cheile străine. Aceste id-uri vor fi indexate automate pentru a crește performanțele bazei de date. Dacă totuși dorim să evităm acest lucru si să avem o altă cheie primară (de exemplu userName pentru tabelul USERS), nu trebuie decât să aplicăm stereotipul <<Identifier>> pe respectivul atribut și acesta va deveni cheia primară.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
     Attention: Generated code! Do not modify by hand!
     Generated by: hibernate3/hibernate.hbm.xml.vsl in andromda-hibernate-cartridge.
  -->
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
          "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping default-cascade="none">
    <class name="org.andromda.timetracker.domain.timecard.TimecardImpl"
        table="TIMECARD" dynamic-insert="false" dynamic-update="false">
        <id name="id" type="java.lang.Long" unsaved-value="null">
            <column name="ID" sql-type="BIGINT"/>
            <generator class="native">
                <!-- id-generator merge-point -->
            </generator>
        </id>
        <property name="startDate">
            <column name="START_DATE" not-null="true" unique="false" sql-type="DATETIME"/>
            <type name="java.util.Date"/>
        </property>
```

Figura 5.3: Parte din fișierul Timecard.hbm generat de Andromda care realizează maparea entității Timecard la tabelul TIMECARD.

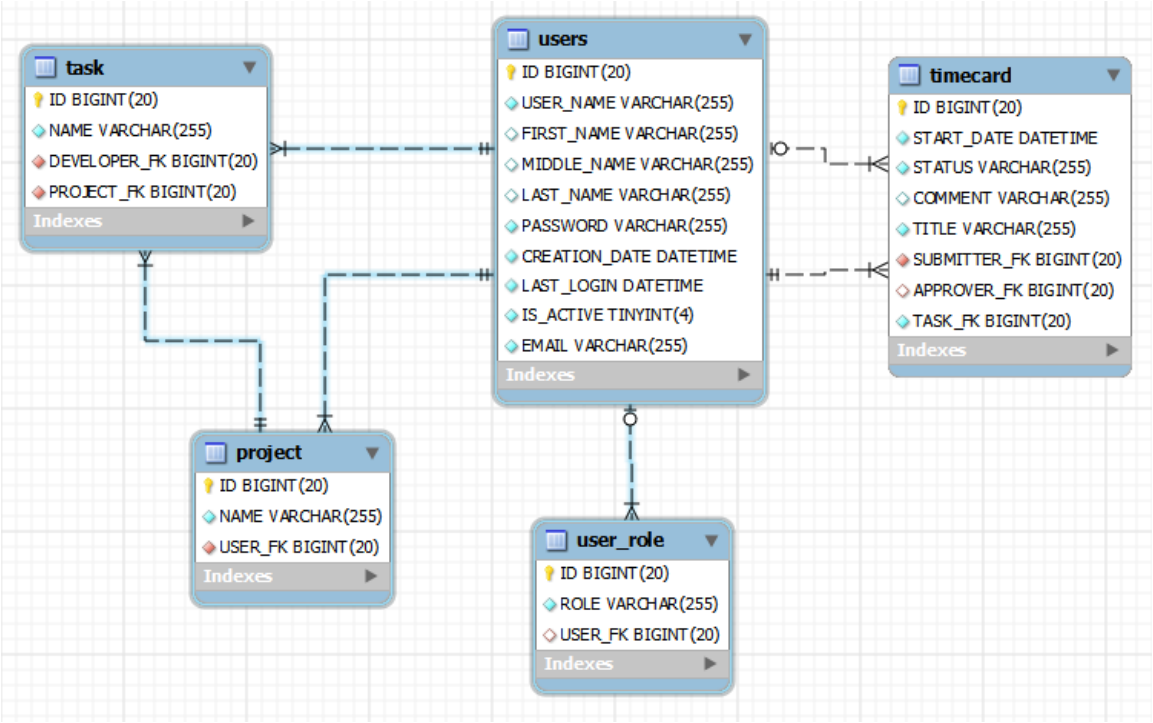

Figura 5.4: Baza de date creată folosind script-ul generat de Andromda.

Așadar, Andromda este de un real folos și salvează enorm de mult timp dacă dorim să folosim Hibernate ca framework de persistență al entităților de business. Partea și mai bună este că tot procesul de generare al acestui strat este realizat cu schimbări

minime în diagrama de model a domeniului, care este un artefact ce ar trebui să existe oricum pentru orice proiect software ce va fi dezvoltat într-o manieră orientată obiect.

5.5. Modelarea serviciilor

Stratul de servicii în Andromda în aplicația de față este generat folosind cartridge-ul Spring. Pentru a înțelege mai bine funcționarea acestuia, precum și modelarea necesară și rezultatul obținute, trebuie să înțelegem prima dată câteva concepte ale acestui framework. Chiar dacă nu este necesar să știm nimic despre Spring pentru a folosi acest cartridge și a face o aplicație funcțională, astfel vom înțelege ce se întâmplă în spatele cortinei. Voi prezenta doar aspecte care sunt folosite în cartridge-ul Spring și în aplicația de față.

5.5.1. Framework-ul Spring

Scop și arhitectură generală

Spring este un framework open-source al cărui scop este de a simplifica dezvoltarea aplicațiilor J2EE. Spre deosebire de alte framework-uri al căror scop este limitat la un singur strat din stiva prezentată în 4.5.1 (de exemplu Hibernate pentru persistență și Struts sau JSF pentru prezentare), Spring ajută la structurarea întregii aplicații într-o manieră consistentă, productivă, adunând laolaltă cele mai bune framework-uri specializate pentru a crea o arhitectură consistentă.

Spring s-a născut din nevoia de a simplifica complexitatea excesivă a celor mai multe aplicații J2EE, care astfel sunt greu de dezvoltat, întreținut și suferă de penalități ale performanței. Câteva din problemele J2EE pe care Sping încearcă să le rezolve:

- Aplicațiile tind să aibe prea mult cod repetitiv și de umplutură care ascunde comportamentul de business și nu permite dezvoltatorilor să se concentreze pe funcționalitățile aplicației.
- Modelul EJB este prea complex și supra-folosit. Chiar dacă scopul inițial al acestei tehnologii a fost de reduce complexitatea codului de business și a crea componente reutilizabile, aceast scop nu a fost atins în totalitate. EJB sunt prin natura lor componente distribuite și tranzacționale. Chiar dacă toate aplicațiile complexe trebuie să facă tranzacții, nu toate au nevoie de natura distribuită a acestor obiecte.
- Pentru aplicațiile J2EE este foarte dificil de aplicat unit testing. Deoarece prin designul EJB-ul nu a fost considerat acest lucru, este foarte dificilă testarea acestor obiecte în afara containerului care le conține. Cu toate acestea, testarea componentelor de business este esențială pentru a obține acoperirea codului și a simula multiple tipuri de defecte.

Chiar dacă J2EE este o platformă de calitate care rezolvă numeroase probleme la un nivel jos, are nevoie de o mai mare simplitate și un ciclu mai rapid de dezvoltare într-o lume din ce în ce mai agilă.

Așadar, Spring oferă servicii pentru întreaga aplicație, și se bazează pe asta pe alte framework-uri verificate (de exemplu Hibernate pentru maparea ORM, sau log4J pentru consemnarea excepțiilor și a atenționărilor). Deci Spring nu încearcă să redefinească tehnologiile existente deja, decât dacă poate aduce un plus de valoare, însă vrea să le lege pe toate într-o manieră consistentă și ușor de folosit. În plus, aduce capacitățile enterprise obiectelor simple de tip POJO. Acest lucru este valoros în mediul J2EE, dar codul astfel scris nu mai depinde de un container enterprise, poate rula într-o gamă mult mai largă de medii (de exemplu aplicații desktop scrise în Swing).

Spring are o arhitectură de tip layer, și este compus din 7 module distincte. Există un nucleu central care definește cum bean-urile sunt create, configurate și controlate, iar celelalte module folosesc acest nucleu, dar sunt separate și pot fi folosite independent sau în conjucție.

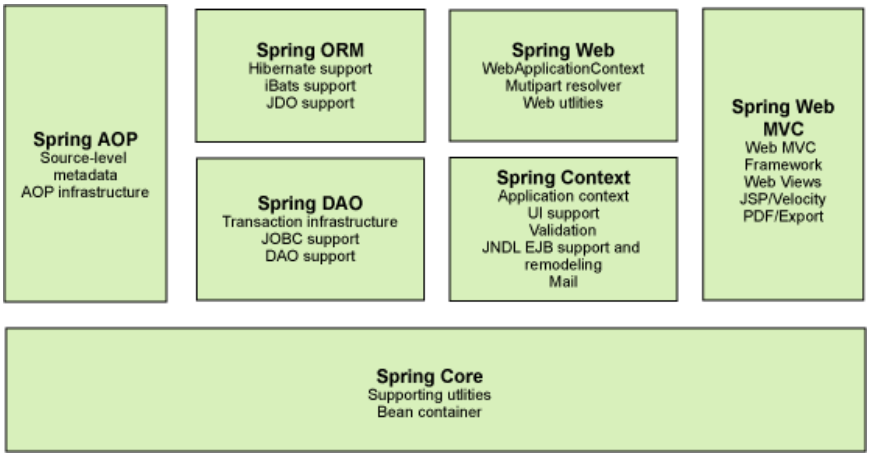

Figura 5.5: Arhitectura generală a framework-ului Spring.

Stratul de abstractizare Spring DAO oferă o ierarhie pentru controlul genral al excepțiilor ce pot fi aruncate de diverse tipuri de baze de date și simplifică codul care trebuie scris pentru utilizarea acestora (de exemplu deschiderea și închiderea conexiunilor la baza de date).

Spring ORM include câteva dintre cele mai folosite framework-uri ORM (Hibernate, iBatis, JDO), pentru a oferi capabilitățile de mapare între entități și baza de date. Toate acestea se conformează modelului abstract al Spring de tranzacționare și management al excepțiilor.

Inversiunea controlului și injectarea dependințelor

Spring este în esență un container de inversiune a controlului. În mod tradițional, o aplicație care folosește o librărie deține controlul secvenței de pași, și face apelurile necesare către librărie atunci când are nevoie de capabilitățile acesteia. Folosind principiul inversiunii controlului, framework-ul deține controlul, urmând flow-ul general, și face apelul către codul aplicației atunci când este cazul. Abordarea acestui concept în

Spring poartă numele de injectarea dependinței, deoarece este mai specifică decât conceptul larg de inversare a controlului.

Injectarea dependințelor se bazează pe elemente specifice ale limbajului Java, nu pe interfețe specifice framework-ului. Astfel framework-ul nu invadează codul aplicației cu cod proprietar; orice aplicație care folosește Spring poate funcționa și fără folosirea lui dacă funcționalitățile oferite de acestea sunt oferite din altă parte.

Aplicația își va expune dependințele prin metode de tip set-er sau prin constructor, iar framework-ul le va injecta la rulare în funcție de necesități și configurație. Așadar, aplicația nu ia obiectele din mediu, ci îi sunt oferite când are nevoie de ele. Acest lucru face ca clasele să se auto-documenteze, deoarece dependințele sunt explicite; permite schimbarea dinamică a sursei obiectului (de exemplu citirea configurației se poate face pe rând dintr-un fișier XML, dintr-un fișier de proprietăți, sau din baza de date) și nu în ultimul rând codul este eliberat de elementele suplimentare care ascund responsabilitățile de business.

BeanFactory

Acest element este o implementare a design pattern-ului Factory și permite crearea și găsirea obiectelor după nume, făcând în plus și managementul obiectelor de tip bean. BeanFactory oferă două tipuri de obiecte:

- **Singleton**. Este tipul standard, și cel mai folosit, fiind ideal pentru servicii stateless. Se va ține o referință comună, care va fi oferită prin căutarea după nume de contextul Spring.
- **Prototype**. Asigură faptul că fiecare căutare va returna o nouă instanță a obiectului care este cerut.

5.5.2. Modelarea serviciilor în Andromda

Pentru a oferi capabilități de businees folosind cartridge-ul Spring, clasele ce vor face acest lucru trebuie să aibă aplicat sterotipul <<Service>>. Astfel se vor genera obiecte de tip POJO care vor fi controlate de Spring și vor putea fi opțional interfațate de Session bean-uri EJB, dacă dorim ca aplicația să ruleze într-un mediu distribuit. Aceste obiecte vor fi de tip singleton.

Avantajul de a folosi Andromda pentru generarea serviciilor este că va crea toate fișierele XML necesare pentru configurarea acestora, incluzând:

- beanRefFactory.xml care va conține BeanFactory-urile ce vor instanția efectiv serviciile, fie ele de tip POJO sau EJB.
- applicationContext.xml care definește proprietățile framework-ului Hibernate, sesiunile entităților de business, interceptorii prin care pot să treacă opțional aceste entități pentru a oferi capabilitățile Aspect Oriented Programming (AOP), precum și definițiile bean-urilor ce vor fi folosite ca servicii.
- applicationContext-localEJB.xml unde sunt definite bean-urile EJB dacă există.
- applicationContext-dataSource.xml unde este definită conexiunea la baza de date și API-ul de tranzacționare.

Chiar dacă Andromda ne scutește de munca necesară pentru a crea aceste fișiere, toate pot fi modificate pentru a îndeplini nevoile speciale ale dezvoltatorilor, folosind MergeMappings, adică fișiere XML în care definim proprietăți ce vor fi copiate și inserate la următoarea generare a codului în aceste fișiere de configurare Spring.

Operațiile modelate pe aceste servicii vor deveni metode ce trebuie implementate de dezvoltator, iar apoi vor fi apelate de bean-urile POJO sau EJB.

O altă facilitate pusă la dispoziție de Andromda este generarea unei clase singleton ServiceLocator, care simplifică procesul de căutare al bean-urilor după nume. Astfel, nu mai trebuie să facem managementul contextului Spring, și vom avea acces la orice serviciu după numele lui în toate straturile aplicației.

Alte elemente generate de cartridge-ul Spring sunt:

- Entități de tip de tip JavaBean denumite Value Objects. Acestea trebuie modelate folosind stereotipul <<ValueObject>> și vor reprezinta încapsularea datelor prin care serviciile comunică cu stratul de prezentare.
- Excepții specifice, care sunt clase modelate cu stereotipul <<ApplicationException>> care pot fi legate de un serviciu printr-o dependință, caz în care toate metodele din acel serviciu vor arunca acea excepție, sau o metodă, caz în care doar metodă respectivă va arunca acea excepție.
- Enumerații, modelate prin stereotipul <<Enumeration>>.
- Securitatea serviciilor și a metodelor. Folosind Spring Security, securitatea se bazează pe rolurile utilizatorilor. Dacă se creează actori pe model ce au același nume cu rolurile definite în fișierele de configurare, actorii vor fi asociați automat cu acele roluri. Astfel, putem securiza serviciile prin simpla adăugare a unor dependințe pe model de la actori la servicii sau metode. Doar dacă utilizatorul este asociat cu rolul respectiv va putea apela repectivul serviciu sau metodă.

Un lucru care mai trebuie menționat este modelarea dependințelor. Dacă un serviciu este modelat ca având dependințe alte servicii sau entități, acestea vor fi injectate automat de Spring folosind principul injectării dependințelor, și deci vor putea fi folosite în serviciul respectiv printr-un simplu apel de metodă, fără a fi nevoiți să creăm instanțe.

5.5.3. Modelarea serviciilor pentru aplicația specificată

Analizând cazurile de utilizare definite în 5.3, și ținând seama de modelul entităților, putem găsi următoarele operații ce trebuie îndeplinite pentru a respecta cerințele funcționale:

- **Crearea unui proiect nou**. Necesită numele proiectului, și clientul căruia îi este destinat.
- **Creare unei noi sarcini**. Necesită numele acesteia și dezvoltatorul care trebuie să o rezolve.
- **Adăugarea unei noi fișe de timp**. Necesită următoarele date: nume, dezvoltator, project manager, data pentru care a fost creată, numărul de ore lucrat, sarcina la care s-a lucrat, statusul și comentarii.
- **Modificarea unei fișe de timp**. Necesită toate datele de care este nevoie la operația de adăugare, și în plus identificatorul sarcinii pentru a putea persista schimbările.

- **Găsirea fișelor de timp**. Necesită criteriile după care se va face filtrarea. Chiar dacă sunt mai multe cazuri diferite în care se poate face căutarea, fiecare cu alte criterii diferite (în funcție de rol sau de scenariul de utilizare), putem abstractiza operația și să oferim un mecanism de căutare care să aplice filtrele doar dacă acestea există. Astfel putem folosi o singură metodă pentru a oferi rezultatele tuturor cazurilor particulare de căutare. Singurul lucru de care trebuie să ținem seama este că aceste criterii pot explicite (completate de utilizator pentru a reduce numărul rezultatelor) sau implicite (deduse din context – de exemplu dacă un client dorește vizualizarea fișelor de lucru pentru un proiect trebuie să filtrăm rezultatele automat după identificarul proiectului pentru a nu-i afișa toate fișele tuturor proiectelor ce respectă criteriile sale de căutare). Dar criteriile implicite vor fi completate în stratul de prezentare, acum trebuie doar să oferim serviciul care realizează această funcționalitate.
- **Găsirea proiectelor pentru un utilizator**. Necesită identificatorul utilizatorului.
- **Găsirea sarcinilor pentru un proiect**. Necesită identificatorul proiectului.
- **Căutarea utilizatorilor**. Necesită rolul după care se va face filtrarea.

Analizând aceste operații, putem observa că diferența între adăugarea unei fișe de timp și modificare ei constă doar în prezența identificatorului fișei la cea din urmă. Așadar, putem rafina lista si să combinăm aceste operații, iar serviciul, pe baza prezenței sau absenței identificatorului va crea o nouă entitate, sau va modifica una existentă folosind restul datelor.

Așadar, putem modela un serviciu al cărui rol este interacțiunea cu entitățile fișe de timp, sarcini și proiect în felul următor:

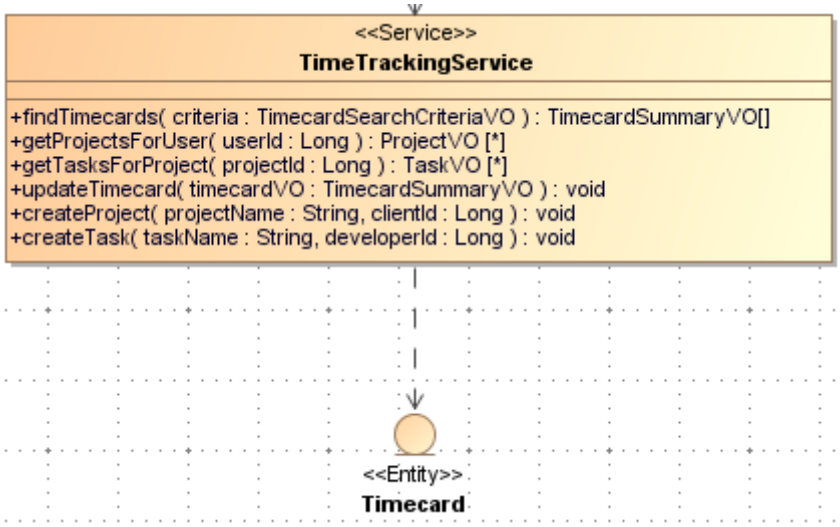

Figura 5.6: Serviciul TimeTracking.

Pentru a simplifica semnătura metodelor de căutare și modificare a fișelor am încapsulat datele în obiect de tip Value Object.

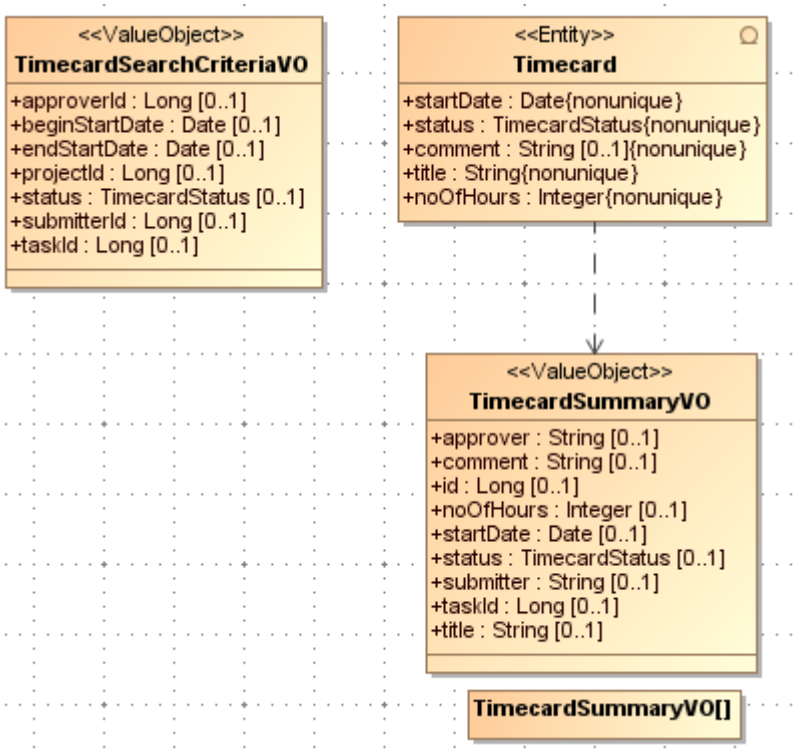

Figura 5.7: Diagrama de obiecte Value Objects referitoare la fișele de timp.

Analizând aceste două figuri, putem observa câteva elemente specifice:

1) Dependința între entitatea Timecard și TimecardSummaryVO. Acest lucru va genera în TimecardDao metode care vor face automat conversia între entitatea de business Timecard și Value Object-ul TimecardSummaryVO.

2) Prezența clasei TimecardSummaryVO[]. Creând o clasă care conține doar numele unui Value Object urmat de caracterele ”[]” și nimic altceva, Andromda va interpreta acest element ca un array ce conține obiecte de tipul Value Object-ului respect, și va genera codul corespunzător peste tot unde acest element este folosit (de exemplu în metoda findTimecards). Acestă construcție nu face parte din specificațiile UML, dar în modelele create în UML 1.4 este singura metodă de a primi șiruri de un tip specificat ca rezultat al apelului unei metode din serviciu. Pentru a respecta specificațiile UML 1.4, putem specifica tipul metodei ca Collection, List sau Array, dar nu putem specifica tipul acestora. În schimb, în UML 2 putem defini multiplicitatea parametrului de ieșire din metodă ca fiind ”many” (selectând *), iar Andromda va genera metode ce vor avea ca parametru de ieșire o colecție de tipul Value Object-ului respectiv (folosind generice).

Din modelul arătat în figura 5.6, Andromda va genera o interfață care definește toate metodele prezente pe model, un bean POJO care folosind principiul injecției dependinței va primit acces la entitatea DAO corespunzătoare entității Timecard, va face verificarea parametrilor și va arunca excepții specifice serviciului, și o clasă în care utilizatorul va implementa efectiv logica de business ce va fi apelată din bean.

```
public abstract class TimeTrackingServiceBase
    implements TimeTrackingService
{
    private TimecardDao timecardDao;

    /**
     * Sets the reference to <code>timecard</code>'s DAO.
     * @param timecardDaoIn
     */
    public void setTimecardDao(TimecardDao timecardDaoIn)
    {
        this.timecardDao = timecardDaoIn;
    }
```

Figura 5.8: Exemplu de bean generat și injecția dependinței.

```
public class TimeTrackingServiceImpl extends TimeTrackingServiceBase {

    @SuppressWarnings("unchecked")
    protected  TimecardSummaryVO[] handleFindTimecards(TimecardSearchCriteriaVO criteria)
            throws Exception {
        return getTimecardDao().toTimecardSummaryVOArray(
                (criteria == null) ? getTimecardDao().loadAll(): getTimecardDao().findByCriteria(criteria)
        );
    }

    protected Collection<ProjectVO> handleGetProjectsForUser(Long userId) {
```

Figura 5.9: Exemplu de implementare al metodelor modelate.

5.6. Modelarea stratului de prezentare

Pentru partea de prezentare, Andromda pune la dispoziția utilizatorilor cartridge-urile JSF sau Struts care vor vor genera o interfață stabilă, flexibilă și ușor de modificat prin modelarea proceselor dinamice de business. Pentru a realiza acest lucru în UML 1.4 se folosesc diagrame de activitate, iar în UML 2.0 diagrame de tip state machine. Ambele cartridge-uri folosesc aceleași elemente pentru generarea codului, ceea ce face ca același model poate fi folosit pentru generarea codului în oricare din cele două framework-uri, în funcție de cartridge. Astfel se folosește conceptul MDA de independență tehnologică, și modelele folosite de Andromda sunt cu adevărat indepedente de implementare (PIM).

Scopul acestor cartridge-uri este de a genera cod care aderă la standardele de bună practică ale industriei, și automatizează toată munca de rutină, lăsându-i și utilizatorului posibilitatea de a adăuga elemente ale logicii de business pentru a completa dezvoltarea. O dată modelul complet Andromda trebuie să fie capabilă să genereze o aplicație gata de rulare, care să funționeze fără a fi nevoie de schimbări ale codului, cu excepția logicii de business.

Pentru aplicația de față am ales cartridge-ul JSF, dar toate elementele de modelare prezentate pot fi folosite și cu cartridge-ul Struts.

Modelarea efectivă se bazează pe cazurile de utilizare ale aplicației. Fiecare caz de utilizare va fi definit pe model, și pentru fiecare va fi creată o diagramă de tip state machine și un controller. Diagrama de tip state machine va controla flow-ul efectiv al logicii de business, și va cuprinde și paginile ce vor constitui interfața propriu-zisă, reprezentate prin stări. Controller-ul va livra logica de business de interfață pentru cazul de utilizare modelat, și va comunica cu stratul de servicii.

Orice aplicație are nevoie de o pagină de pornire (prima pagină care va fi oferită utilizatorul ajunge atunci când folosește aplicația). Pentru a marca acest lucru, pe cazul de utilizare ales va fi aplicat stereotipul <<FrontEndApplication>>. De asemenea, pentru ca Andromda să știe pentru ce cazuri de utilizare să genereze efectiv cod, acestea trebuie marcate cu stereotipul <<FrontEndUseCase>>. Astfel putem avea pe model mai multe cazuri de utilizare, iar generarea se aplică doar celor care implementează acest sterotip.

Figura 5.10: Modelarea cazului de utilizare Search Timecards.

Așa cum am amintit anterior, fiecărui caz de utilizare îi va fi atașată o diagramă de tip state machine și un controller. Pentru cazul de utilizare diagrama va arăta în felul următor:

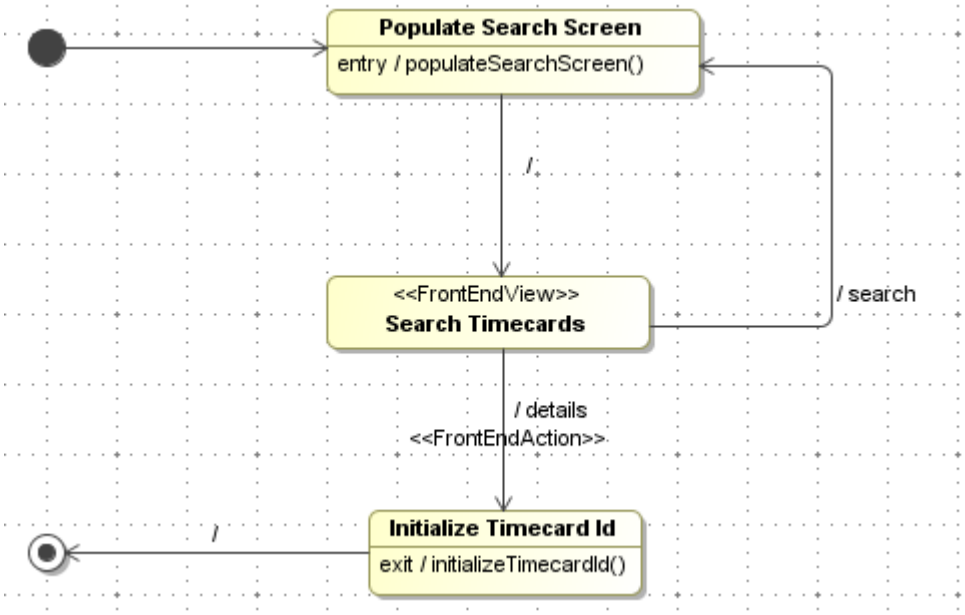

Figura 5.11: Modelarea diagramei state machine pentru cazul de utilizare Search Timecards.

Fiecare caz de utilizare necesită o stare inițială (pe model este un disc negru). Marchează începutul cazului de utilizare, și nu are decât o tranziție exterioară. Pentru fiecare diagramă este obligatorie prezența unei singure stări inițiale.

Stările intermediare pot fi de două tipuri: stări ale serverului (apeluri către controller) și stări ale clientului (de exemplu pagini JSP). Aceste tipuri de stări se diferențiază prin stereotipuri. Pentru a genera dintr-o stare o pagină JSP, aceasta trebuie marcată cu stereotipul <<FrontEndView>>. O stare de acest tip poate avea mai multe tranziții de ieșire, câte una pentru fiecare apel diferit la server.

Celelalte stări ce nu prezintă un stereotip sunt acțiuni ale serverului. Acestea de obicei definesc metode care vor apela metoda controllerului cu aceeași semnătură, și nu pot avea decât o singură tranziție de ieșire.

Tranzițiile sunt elementele care controlează logica de business propriu-zisă. Dacă o tranziție va avea atașată pe model un comportament de tip activate cu nume definit, va genera formularul care va colecta aceste date, în funcție de tag-urile aplicate parametrilor acesteia. În schimb, dacă o tranziție conține un comportament de tip activitate fără nume, parametrii acestei stări vor constitui variabile de pagină transmise următarei pagini. Aceste variabile de pagină vor constitui datele returnate de server ce vor fi doar afișate pe pagină, fără a putea fi modificate de client. Pentru a transmite aceste date controller-ului, acesta trebuie să implementeze o metodă ce va avea parametrii de intrare cu același nume și tip ca parametrii definiți în activitatea tranzițiilor.

În continuare voi exemplifica folosind modelul cazului de utilizare Search Screen.

Acesta are o tranziție exterioară din starea Search Timecards. Pe această tranziție este definit un comportament de tip activitate care are următorii parametrii:

- submitter
- approver
- status
- beginStartDate
- endStartDate

Pentru a indica cartridge-ului că vrem generarea elementelor de interfață ce vor furniza acești parametri, aceștia trebuie marcați acestora cu stereotipul andromda_presentation_web_view_field_type. În funcție de valoare acestui stereotip se vor genera elementele corespunzătoare de interfață. De exemplu: select va determina crearea unui combobox, calendar a unui selector de date, checkbox a unui checkbox, text va genera un textfield, și așa mai departe. De asemenea, dându-i metodei numele search va determina crearea unui buton cu eticheta ”Search” ce va trimite datele corespunzătoare acestui formular către server la apăsarea lui.

Tranziția care intră în Search Screen are definit un comportament de tip activitate ce are un parametru: timecardSummaries de tipul TimecardSummaryVO[] (adică un array de obiecte TimecardSummary – vezi secțiunea 5.5.3.). Deoarece această activitate nu este numită, timecardSummaries va fi o variabilă de pagină care va conține datele ce vor fi afișate (rezultatul căutării). Aplicând stereotipul andromda_presentation_view_table_columns acestui parametru, putem filtra care din câmpurile conținute vor fi afișate utilizatorului, și Andromda va deduce că trebuie creat

un tabel care va conține aceste date. Tabelul rezultat permite sortarea crescătoare sau descrescătoare a rezultatelor după toate câmpurile acestuia și este paginat, mărimea paginii putând fi aleasă tot de pe model.

Pentru a putea transmite datele preluate din formularul de cautare, a fost realizată o activitate pe starea populate search screen ce are definit un nod în care se poate specifica metoda pe care activitatea aceasta o va apela. Astfel, se va face apelul automat al metodei populate search screen din controller atunci cînd aplicația ajunge în starea Populate Search Screen. Metoda din controller va primi un obiect de tip PopulateSearchScreenForm, care va încapsula toate datele care care se referă la pagina respectivă, și anume atât parametrii din formular cât și variabilele de pagină. Astfel, se poate citi formularul pentru a colecta datele care vor fi trimise serviciului pentru a face efectiv filtrarea, și scrie rezultatul obținut în variabila timecardSummaries pentru afișare.

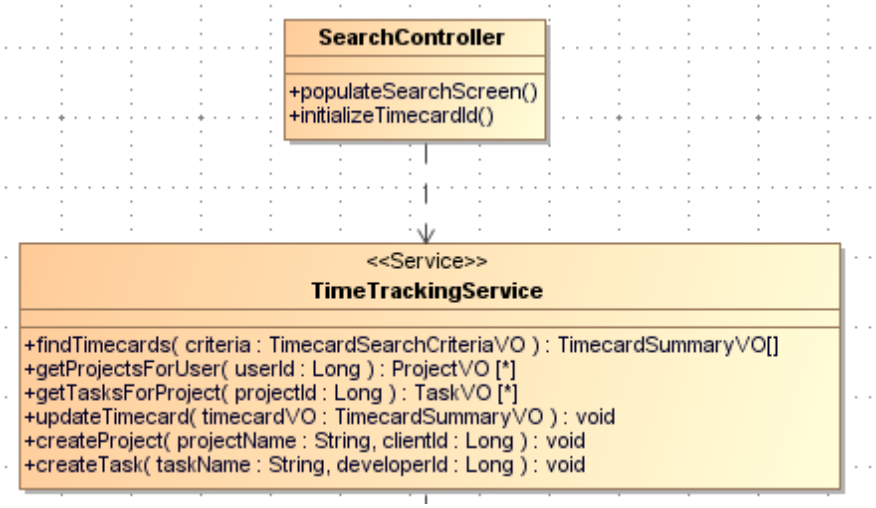

Figura 5.12: Search controller.

Starea finală a cazului de utilizare este modelată printr-un disc cu centrul negru și marginea albă (vezi figura 5.11). Acesată stare finală poate constitui punctul de intrare al următorului caz de utilizare, dacă avem astfel de cazuri legate între ele. Se pot transmite și parametri ca variabile de pagină, și deci putem construi flow-uri mult mai complexe decât cele cuprinse în cadrul unui singur caz de utilizare.

```java
@Override
public void populateSearchScreen(PopulateSearchScreenForm form) {
    // Get list of users and add the "All" option at the top
    Collection<UserVO> users = getUserService().getUsersByRole(Role.DEVELOPER);

    // Add the 'All' option at the top of the list, as a fake user
    List<UserVO> userList = new ArrayList<>(users);

    // Sorts the names
    Collections.sort(userList, userComparator);
    userList.add(0, new UserVO(0L, null, null, null, null, ALL));

    // Populate submitter and approver dropdowns
    form.setSubmitterBackingList(userList, "id", "displayName");
    form.setApproverBackingList(userList, "id", "displayName");

    // Populate status dropdown
    List<String> statusLabels = new ArrayList<>(TimecardStatus.literals());
    List<String> statusValues = new ArrayList<>(TimecardStatus.names());
    statusLabels.add(0, ALL);
    statusValues.add(0, "");
    form.setStatusLabelList(statusLabels.toArray());
    form.setStatusValueList(statusValues.toArray());

    // Populate timecard summaries
    form.setTimecardSummaries(getTimeTrackingService().findTimecards(getTimecardSearchCriteria(form)));
}
```

Figura 5.13: Implementarea metodei populateSearchScreen din SearchController.

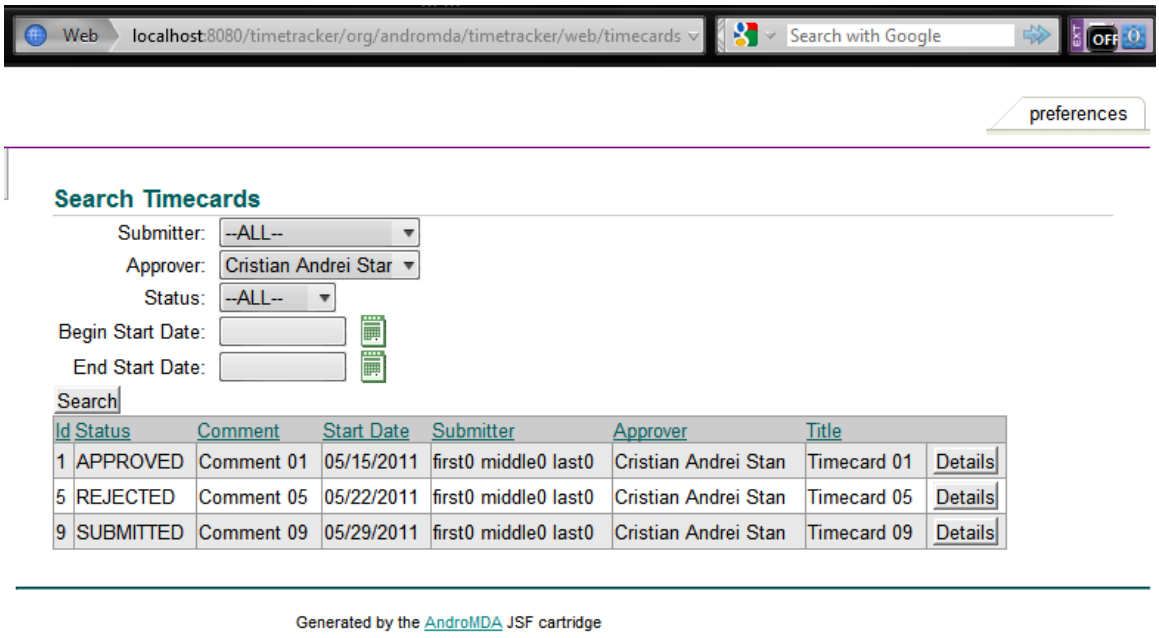

Figura 5.14: Ecranul de căutare generat de cartridge-ul JSF.

Un alt lucru care este foarte mult simplificat de folosirea cartridge-ului JSF este posibilitatea modelării securității direct pe model. Similar modului în care este descrisă

securitatea în modelarea serviciilor (5.5) la fel se aplică securitatea și la nivelul cazurilor de utilizare. Diferența este că în loc să legăm dependințele dintre actori și servicii, acum o să creăm dependințe între actori și cazurile de utilizare. Astfel, un utilizatorul care nu este o instanță a unui actor legat de un caz de utilizare, pur și simplu nu va avea acces la acesta (adică nu îi va fi generat nici un element de interfață care să îl lege de cazul respectiv, și nici nu își va putea forța intrarea prin scrierea directă a link-ului în browser sau salvarea unui bookmark).

6. Testare şi validare

Datorită utilizării framework-ului Spring în aplicație, chiar dacă capacitățile de business sunt expuse și ca bean-uri EJB, structura acesteia permite aplicarea unit testelor pe servicii și metode individuale.

Pentru realizarea testării, am folosit framework-ul TestNG. Acesta este inspirat de JUnit, dar îl extinde cu câteva funcționalități suplimentare, cum ar fi:

- Anotări.
- Testarea aplicației în diverse scenarii de threading (de exemplu fiecare metodă în thread separat, fiecare clasă cu thread separat, etc.).
- Configurații flexibile.
- Data driven testing.

Pentru a include TestNG în proiect, datorită utilizării Maven, singurul lucru care a trebuit făcut a fost trecerea următoarei dependințe în pom-ul principal al proiectului:

```
<dependency>
        <groupId>org.testng</groupId>
        <artifactId>testng</artifactId>
        <version>5.14.9</version>
        <scope>test</scope>
 </dependency>
```

Am folosit TestNG pentru testarea metodelor serviciilor implementate. Prin crearea unui nou beanFactory, acesta va putea datorită injecției dependinței de către Spring să apeleze direct serviciile fără a fi nevoie ca aplicația să fie instalată într-un container.

Voi prezenta ca exemplu testul aplicat metodei getUsersByRole din serviciul UserService.

```
public class UserServiceTest {
    private Log logger = LogFactory.getLog(UserServiceTest.class);

    private UserService userService;

    /**
     * Initialize test suite
     */
    @Configuration(beforeSuite=true)
    public void initializeTestSuite() {

        // Initialize ServiceLocator
        logger.info("Initializing ServiceLocator");
        ServiceLocator locator = ServiceLocator.instance();
        locator.init("testBeanRefFactory.xml", "beanRefFactory");

        // Initialize UserService
        logger.info("Initializing UserService");
        userService = locator.getUserService();
    }

    @Test
    public void testGetAllUsers() {
        logger.info("testGetAllUsers:");
        Collection<UserVO> users = userService.getUsersByRole(Role.DEVELOPER);

        for (UserVO user : users) {
            logger.info(user.getUserName() + " | "+ user.getFirstName() + " " + user.getMid
        }
    }
```

Figura 6.1: Exemplu de unit test folosind TestNG.

După cum se observă, un unit test este compus din două elemente: inițializare și testul efectiv, diferențiate prin anotările corespunzătoare. În initializeTestSuite inițializăm și obținem o referință către serviciul ce va fi apelat, iar în testGetAllUsers, pur și simplu apelăm metoda pe care vrem să o testăm și afișăm rezultatele. Astfel, o să avem afișat în log-ul testului rezultatul și putem verifica corectitudinea metodei ținând cont de datele din baza de date.

Așadar, folosind TestNG în aplicațiile generate cu Andromda putem adera la principiile Test Driven Development, adică să creăm testele chiar înainte de a implementa efectiv metodele, pe baza cerințelor definite pentru fiecare operație, iar apoi să implementăm doar atât cât este necesar pentru a trece testele. Astfel avem asigurată corectitudinea din punct de vedere funcțional al stratului de servicii, urmând ca singurul loc unde să aplicăm testarea manuală să fie partea de prezentare.

7. Manual de instalare și utilizare

Următoarele programe au fost folosite pentru dezvoltarea aplicației:

- Apache Maven 2.2.1.
- JDK 1.6.25.
- IntellijIdea Community Edition 10.5.
- MagicDraw 16.5.
- Jboss 5.1.0.GA.
- MySql Server 5.1.
- MySql Workbench 5.2.
- Firefox 4 cu Firebug 1.7 instalat.

Toate programele care sunt necesare pentru rularea aplicației se vor pe cd-ul care însoțește lucrearea. Pentru a instala și rula aplicația trebuie parcurși următorii pași:

7.1. Instalează Java.

Dacă JDK versiunea 6 este deja instalată pe sistem, puteți sări peste acest pas.

Andromda suportă Java 1.5 și 1.6, dar este recomandată folosirea ultimei versiuni scoase de la http://www.oracle.com/technetwork/java/javase/downloads/index.html.

7.2. Instalează Maven.

Maven poate fi găsit la adresa http://maven.apache.org/download.html, sau pe cd-ul atașat. Se recomandă folosirea versiunii 2.2.1. Maven este sub forma unei arhive; aceasta trebuie doar dezarhivată în locația dorită.

7.3. Instalează JBoss Application Server.

JBoss este un server pentru aplicații Java și un container de EJB open source și foarte popular. Versiunea care trebuie folosită este 5.1.0 deoarece este singura versiune pe care versiunea actuală de Andromda funcționează fără probleme. Aceasta se poate găsi la http://sourceforge.net/projects/jboss/files/JBoss/JBoss-5.1.0.GA/, sau pe cd-ul atașat. La fel ca Maven, aplicația necesită doar dezarhivarea într-un folder ales.

Verificarea funcționării acestui server se face accesând URL-ul http://localhost:8080/ (sau http://127.0.0.1:8080 dacă variabila localhost nu este setată). Aici ar trebui să apară consola de administrare JBoss.

Deoarece JBoss folosește portul 8080, dacă aveți alte aplicații instalate care utilizează acest port (de exemplu o bază de date Oracle), trebuie schimbat portul HTTP al JBoss. Pentru aceasta, trebuie editate fișierele ”\default\conf\bindingservice.beans\META-INF\bindings-jboss-beans.xml” și ”server\default\deploy\jbossweb.sar\server.xml” din folderul unde este instalat JBoss și toate aparițiile numărului 8080 trebuie înlocuite cu noul port (de exemplu 9090). Atunci JBoss va putea fi accesat la adresa http://localhost:9090/.

7.4. Setarea variabilelor de mediu.

Pentru a seta variabilele de mediu, în Linux trebuie adăugate în fișierul /home/.bashrc.

Pentru Windows, aceastea se adaugă în felul următor: Control Panel -> System sau click dreapta pe Computer -> Properties. Se selectează Advanced system settings, tab-ul Advanced, opțiunea Environment Variables. Acolo se pot observa variabilele de sistem existente.

Pentru funcționarea corectă a Andromda, următoarele variabile trebuie setate:

- JAVA_HOME. Valoarea acesteia trebuie să fie folderul în care este instalat JDK-ul Java. Exemplu: C:\Program Files\Java\jdk1.6.0_25. Această variabilă poate fi setată automată la instalarea Java, caz în care nu mai trebuie adăugată. Este obligatorie.
- M2_HOME. Valoarea este folderul în care Maven a fost instalat. Exemplu: D:\APPS\apache-maven-2.2.1. Este obligatorie.
- M2_REPO. Valoarea folderului în care Maven va pastra toate dependințele și librăriile necesare. De exemplu: C:\Users\{Your Username}\.m2\repository. Este obligatorie.
- MAVEN_OPTS. Reprezintă parametrii transmiși mașinii virtuale Java atunci când rulează Maven. Este opțională, dar este recomandat să se adauge cu cel puțin valoarea ”-XX:MaxPermSize=128m -Xmx512m”, altfel s-ar putea ca la rularea unei proiect de dimensiuni mari, cum sunt cele generate de Andromda, mașina virtuală Java să rămână fără memorie. În acest caz se vor arunca excepții java.lang.OutOfMemoryError: PermGen space.
- JAVA_OPTS. Reprezintă parametrii transmiși mașinei virtuale Java. Este opțională, dar adăugându-se valoarea ” -Xmx1024m -XX:MaxPermSize=450m” se pot obține creșteri considerabile de performanță ale serverului JBoss dacă cantitatea de RAM disponibilă în sistem permite acest lucru.
- JBOSS_HOME. Valoarea va fi folderul în care JBoss a fost instalat. Este opțională, dar dacă se adaugă, atunci când se apelează comanda ” mvn -f app/pom.xml -Ddeploy” care construiește efectiv proiectul, acesta va fi instalat automat în JBoss, altfel pentru a rula aplicația aceasta trebuie copiată manual din {project.folder}\app\target în folderul {jboss.folder}\server\default\deploy.
- De asemenea, variabila Path trebuie editată; se adaugă la valoarea conținută ”%JAVA_HOME%\bin;%M2_HOME%\bin;”. Este obligatoriu.

7.5. Instalarea bazei de date MySql.

Se scoate ultima versiune de server MySql de la adresa http://dev.mysql.com/downloads/mysql și se instalează. Versiunea 5.1 se găsește pe cd. Pentru ca aplicația să funcționeze este necesar ca serverul să utilizeze portul 3006. Este recomandată instalarea și a MySQL Workbench, de la adresa http://dev.mysql.com/downloads/workbench/. Versiunea 5.2. se află pe cd. Acest program reprezintă interfața vizuală pentru serverul MySql și face munca de management a bazei de date mult mai ușoară. După instalarea bazei de date, trebuie creată o schemă numită ”timetracker”, și un utilizator ”tt_user” cu parola ”tt_pass”. De asemenea, utilizatorului tt_user trebuie să i se acorde drepturi depline asupra schemei timetracker.

Un singur lucru mai trebuie făcut înainte de instalarea aplicației, și anume adăugarea librăriei mysql-connector-java în librăriile serverului JBoss. Pentru aceasta, se scoate de la adresa http://dev.mysql.com/downloads/connector/j/ arhiva ce conține această librărie. După dezarhivare, se va găsi în interiorul ei un fișier de tip JAR, cu numele ”mysql-connector-java-5{versiune}-bin.jar”. Acest fișier trebuie redenumit în mysql-connector-java-5{versiune}.jar” (adică se șterge ”-bin” de la sfârșitul numelui). Versiunea 5.1.16 denumită corect se poate lua de pe cd. Pe urmă va fi copiat în folderul JBOSS_HOME \server\default\lib.

7.6. Instalarea și rularea aplicației.

Pentru a instala aplicația, trebuie să deschidem o linie de comandă, și să schimbăm calea la folderul în care aplicația a fost instalată. Aici vom scrie comanda ”mvn install”. Acest lucru va determina ca proiectul să fie construit. Trebuie ca la finalul acestui pas să primim un mesaj ”BUILD SUCCESFULL”.

Deoarece comanda ”mvn install” va determina ca toate dependințele Maven, Andromda și ale proiectului să fie download-ate de pe internet în folderul M2_REPO, acest lucru va dura cel puțin 20-30 de minute, chiar și cu o conexiune rapidă la internet. Pentru a evita acest lucru, în folderul mavenrepository din proiect se află o arhivă a repository-ului de Maven cu care a fost construit proiectul. Aceasta trebuie doar dezarhivată în folderul M2_REPO. În acest caz, proiectul se construiește cu comanda ”mvn install -o”. Flag-ul –o indică faptul că maven va folosi repository-ul instalat, și nu va mai incerca să se conecteze la internet pentru a scoate toate acele dependințe.

O dată ce pasul se termină, și primim rezultatul ”BUILD SUCCESFULL”, trebuie să creăm baza de date și să o populăm cu datele de test. Pentru aceasta, trebuie rulate comenzile ”dbcreate” și ”dbupdate”. Acestea sunt scripturi simple care conțin următoarele comenzi:

”mvn -f core/pom.xml org.andromda.maven.plugins:andromdapp-maven-plugin:schema -Dtasks=drop -o && mvn -f core/pom.xml org.andromda.maven.plugins:andromdapp-maven-plugin:schema -Dtasks=create –o”

” mysql --user=tt_user --password=tt_pass < sql\insert.sql”.

După cum se observă, economisesc multe caractere tipărite.

Următorul pas este instalarea aplicației în serverul JBoss. Pentru aceasta se poate folosi scriptul ”deploy”. Acesta conține comanda ” mvn -f app/pom.xml -Ddeploy -o”. Dacă variabila de sistem JBOSS_HOME a fost setată cum am arătat în secțiunea 7.4, proiectul va fi copiat automat în server. Altfel, trebuie copiat manual, după indicațiile din 7.4.

Pentru a accesa aplicația, trebuie să deschidem un browser web și să accesăm adresa http://localhost:8080/timetracker.

8. Concluzii

După procesul de dezvoltare al acestei aplicații, precum și din analiza în detaliu a framework-ului de implementare (Andromda), pot afirma că Model Driven Architecture a ajuns la un nivel de maturitate care îi permite folosirea cu succes în dezvoltarea aplicațiilor web enterprise.

Aplicația generată utilizează cele mai folosite și validate framework-uri din industrie, dar Andromda simplifică foarte mult procesul general și colaborarea între acestea, mult dincolo de capabilitățile fiecărui framework luat în parte.

Chiar dacă curba de înțelegere în detaliu a Andromda este destul de abruptă, oferă în mod standard crearea de aplicații funcționale, care chiar pot rezolva probleme complexe de business. O dată ce trecem de nivelul de gândire al dezvoltării tradiționale și gândim mai degrabă în concepte de business și modele abstracte, observăm că putem livra aplicații de o calitate crescută într-un ritm mult mai rapid decât era posibil până acum.

Probabil că în viitorul apropiat, o dată cu răspândirea pe o scară mult mai largă a folosirii Model Driven Architecture și dezvoltarea unor noi unelte și framework-uri mai performante decât cele existente, nu vom mai nevoiți să scriem deloc fișiere xml de configurare, interfețe Java, să facem managementul configurației și al resurselor sau să pierdem timp cu defecte ale interfeței cu utilizatorul.

Totuși chiar și un tool atât de performant ca Andromda încă mai are nevoie de îmbunătățiri. Folosind cartridge-ul Spring, ar fi mult mai ușor dacă am putea adăuga capabilități Aspect Oriented Programming direct pe model. Un prim pas este făcut prin modelarea securității pe servicii și cazuri de utilizare și a tratamentului excepțiilor, dar încă se mai pot face multe altele: logare, caching, audit, profiling, etc.

Chiar dacă modelarea pentru partea de prezentare rezolvă multe probleme și oferă aplicației un nivel ridicat de calitate, scurtând timpul și necesitatea învățării multor tehnologii, încă se simte mai dificilă decât ar fi cazul, și mult mai greu de deprins decât modelarea serviciilor sau a domeniului problemei. Probabil acest lucru este determinat și de tipul diferit de diagrame pe care aceste categorii le utilizează. Este mult mai ușor de înțeles și modelat o vedere statică decât una dinamică.

Cu toate acestea, per total numărul de linii de cod necesare și timpul dezvoltării a scăzut substanțial. Logica de business este mult mai clar separată de restul codului structural. Procesul este mult mai agil deoarece o dată creat un prim prototip al aplicației și înțelegerea modului în care trebuie modelată problema, noi funcționalități sau schimbări ale celor existente pot fi adăugate foarte ușor. Nu mai este nevoie de repetarea nici unei singure linii de cod de umplutură, tot ceea ce scrie dezvoltatorul se referă strict la domeniul problemei. Aplicația este mult mai clar structurată pe straturi și astfel se crează posibilitatea separării muncii dezvoltatorilor pe diferite nivele. Este de asemenea foarte ușor de realizat testarea elementelor individuale, un avantaj major într-o lume agilă.

De asemenea este de notat și faptul că Andromda oferă posibilitatea extinderii și a particularizării soluțiilor, cartridge-urile pot fi cu ușurință modificate sau extinse pentru a răspunde unor nevoi specifice, și tot codul generat oferă nenumărate mijloace prin care să fie configurat dacă nu se dorește intervenția la nivel de cartridge, începând de la fișierele de configurare Hibernate, Spring, până la modificarea substanțială a aspectului interfeței utilizator.

Așadar, Model Driven Architecture este un pas evolutiv substanțial în dezvoltarea aplicațiilor software, și Andromda reușește să se ridice la înălțimea așteptărilor și a promisiunilor făcute de MDA.

Obiectivele au fost îndeplinite, deoarece am identificat un framework de calitate care folosind paradigma MDA a generat o aplicație ce răspunde tuturor cerințelor funcționale exprimate, și a demonstrat cât de mult poate fi accelerat procesul de dezvoltare software și cât de simplă poate deveni crearea crearea aplicațiilor web enterprise, răspunzând cerințelor din ce în ce mai presante legate de timp, costuri, și calitate.

9. Bibliografie

[1] Richard Soley and the OMG Staff Strategy Group, *Model Driven Architecture*, Object Management Group, White Paper, November 27, 2000.

[2] John D. Poole, *Model-Driven Architecture: Vision, Standards And Emerging Technologies*, ECOOP 2001, April 2001.

[3] Dr. Andreas Tolk, *Avoiding another Green Elephant – A Proposal for the Next Generation HLA based on the Model Driven Architecture*, 2002 Fall Simulation Interoperability Workshop, September 2002.

[4] Milan Karow, Andreas Gehlert, *On the Transition from Computation Independent to Platform Independent Models*, Twelfth Americas Conference on Information Systems, August 2006.

[5] Renas Reda, Yusuf Tözmal, *Model Driven Architecture - Test Methods and Tools*, School of Engineering Blekinge Institute of Technology, Master Thesis, January 2006.

[6] *Behaviour Modelling in Model Driven Architecture*, First European Workshop on Behaviour Modelling in Model Driven Architecture, 2009.

[7] Santiago Meliá, Andreas Krau, and Nora Koch, *MDA Transformations Applied to Web Application Development*, 2005.

[8] Santiago Melia, Cristina Cachero and Jaime Gomez, *Using MDA in Web Software Architectures*, Universidad de Alicante, Spain, 2003.

[9] Dragan Gasevic, Dragan Djuric, Vladan Devedzic, *Model Driven Architecture and Ontology Development*, Springer-Verlag Berlin Heidelberg 2006.

[10] Peter Herzum, Oliver Sims, Business Component Factory: A Comprehensive Overview of Component-Based Development for the Enterprise, Wiley, 1999.

[11] Gabriele Taentzer, Dirk Muller, and Tom Mens, *Specifying Domain-Specific Refactorings for AndroMDA based on Graph Transformation*, Philipps-Universit¨at Marburg, Germany, 2008.

[12] Ovidiu Pop, *Sisteme Informatice*, Curs susținut la Universitatea Tehnică din Cluj-Napoca, Calculatoare An 4, 2010.

[13] Martin Matula, *NetBeans Metadata Repository*, articol, 2003. Disponibil la: http://classfoo.googlecode.com/svn/trunk/doc/metadata/others/MDR-whitepaper.pdf.

[14] Joel Kozikowski, *A Bird's Eye view of AndroMDA*, articol, 2006. Disponibil la http://svn.softwarepublico.gov.br/trac/mdarte/browser/Desenvolvimento/andromda-src-3.1/documentation/xdocs/resources/A-Birds-Eye-View-of-AndroMDA.zip?rev=3&format=raw

[15] Mda CASE Tools, http://case-tools.org/mda.html.

[16] Getabest.net - Software Catalog: Andromda review, http://getabest.net/andromda-download-new-120623.html.

[17] ohloh: Andromda review, https://www.ohloh.net/p/AndroMDA/reviews.

[18] Andromda website, http://www.andromda.org/docs/index.html.

[19] Three-Layered Services Application, MSDN, http://msdn.microsoft.com/en-us/library/ff648105.aspx.

[20] OMG Model Driven Architecture, http://www.omg.org/mda/.

Anexa 1. Lista figurilor

Anexa 2. Abrevieri

AOP	Aspect Oriented Programming
API	Application Programming Interface
BPMN	Business Process Model and Notation
CIM	Computational Independent Model
CRUD	Create, Read, Update, Destroy
CORBA	Common Object Request Broker Architecture
DAO	Data Access Object
EAR	Enterprise Archive
EJB	Enterprise JavaBean
EMF	Eclipse Modeling Framework
JAR	Java Archive
J2EE	Java Platform, Enterprise Edition
JSF	Java Server Faces
JSP	Java Server Pages
MDA	Model Driven Architecture
MDR	Meta Data Repository
MOF	Meta-Object Facility
OCL	Object Constraint Language
OMG	Object Management Group
ORM	Object-Relational Mapping
PIM	Platform Independent Model
POJO	Plain Old Java Object
POM	Project Object Model
PSM	Platform Specific Model
QVT	Query/View/Transformation
UML	Unified Modeling Language
WAR	Web Archive
XMI	XML Metadata Interchange
XML	eXtensible Markup Language
XSD	XML Schema Definition