UNIVERSITATEA POLITEHNICA BUCUREŞTI

**Facultatea de Electronică, Telecomunicaţii și Tehnologia Informaţiei**

**Inginerie software**

Lucrarea :

**Model Driven Architecture – MDA**

**Profesor coordonator :**   **Studenți :**

prof. univ. dr. ing. Ştefan Stăncescu Pavel Mihai-Alexandru

Tudor Ștefan-Eugen

Grupa 433 A

Anul universitar

2015 - 2016

**Cuprins**:

**Capitolul 1**: Introducere în conceptul MDA……………….………………2

**Capitolul 2**: MDA Tools…………………………………………………...3

**Capitolul 3:** Concepte de bază……………………………………………..4

 3.1. UML 2.0……………………………………………………………..4

 3.2. MOF…………………………………………………………………5

 3.3. XMI………………………………………………………………….6

 3.4. PIM/PSM/PDM……………………………………………………...7

 3.5. Multi-stage transformation…………………………………………..8

 3.6. Action languages…………………………………………………….8

 3.7. Core models………………………………………………………...11

 3.8. Controlul transformării din PIM în PSM…………………………...12

 3.9. Executable UML……………………………………………………13

**Capitolul 4**: Tipuri de arhitecturi……………………………………………….…14

 4.1 Business Motivation Model (BMM) ………………………………..14

 4.2. Service orientated architecture (SOA) ……………………………...15

4.3. Arhitectura acționată de evenimente (EDA) ………………………..16

**Bibliografie**…………………………………………………………………………....18

**Alocare capitole:**

**Pavel Mihai**: Capitolul 1, Capitolul 2, Capitolul 4

**Tudor Ștefan**: Capitolul 3

**Capitolul 1: Introducere în conceptul MDA**

 **Model-Driven Architecture** sau **Arhitectura Acționată de Modele** reprezintă un model de design software folosit in dezvoltarea de sisteme software. Acest gen de arhitectură separă specificațiile funcționale ale sistemului de specificațiile implementării pe o platformă software specifică, fiind exprimate prin termenul *model*. Noțiunea de *model* face referire la reprezentarea abstractă a unei caracteristici unui sistem dintr-o anumită perspectivă. În cadrul dezvoltării unui produs software, crearea unui model are ca scop afișarea aspectelor importante prin diagrame bine definite.

 Acest concept lansat in anul 2001 de către *Object Management Group* s-a dovedit a fi un pas important in domeniul dezvoltării de produse software. Ideea a pornit de la separarea caracteristicilor referitoare la operațiile unui sistem de detaliile privind modul în care sistemul folosește capabilitățile platformei sale.

 MDA urmărește realizarea câtorva obiective primare: portabilitatea, interoperabilitatea și refolosirea prin separarea arhitecturală a elementelor.

 *Object Management Group* a urmărit stabilirea următorul principiu ce definește MDA: Este necesar ca exprimarea și caracterizarea modelelor sa fie bine definită pentru a putea înțelege sistemul ce se dorește a fi implementat la nivel de întreprinderi. Pentru a realiza acest lucru este necesară existența unei baze de date formale pentru a ușura transformarea modelelor cu ajutorul diferitelor instrumente ce urmează a fi prezentate. [3],[4]

Viziunea *Object Management Group* se caracterizează prin următoarele patru principii care au stat la baza MDA:

* Modelele trebuie să fie bine și clar definite.
* Design-ul unui sistem se face în jurul modelelor.
* Crearea unei baze formale pentru modelele ce definesc sistemul, facilitând funcționarea aplicațiilor automate.
* ****Existența unui standard în implementarea unor astfel de sisteme ar aduce beneficii companiilor. Figura 1. Domenii ce utilizează arhitecturi MDA

**Capitolul 2: Instrumente (tools) MDA**

Un tool MDA reprezintă un instrument folosit în dezvoltarea modelelor (spre exemplu UML) sau metamodelelor (ex: CWM), în general existând două tipuri de modele: modele inițiale și derivate. Modelul inițial este realizat de către utilizator în timp ce modelul derivat se creează automat de către programe. (Exemplu: Un analist de afaceri poate creea un model UML pe baza constatărilor sale. De asemenea, un program poate opera o transformare, în mod automat, de la acest model UML într-unul de tip Java ). [4]

 Instrumentele MDA pot fi de mai multe tipuri:

* **Creation Tool -** Un instrument folosit pentru a obtine modele inițiale și / sau edita modele derivate.
* **Analysis Tool –** Instrument folosit pentru verificarea modelelor: completare, detectare erori, condiții de avertizare. De asemenea poate fi folosit pentru calcularea valorilor modelului.
* **Transformation Tool –** Instrument folosit pentru transformarea unor modele în alte modele, cod sau documentații.
* **Composition Tool –** Instrument folosit în alăturarea mai multor modele de sursă într-un metamodel
* **Test Tool –** Un instrument folosit în testarea modelelor
* **Simulation Tool –** Instrument folosit în simularea execuției unui sistem bazat pe un model anume.
* **Metadata Management Tool –** Instrument creat pentru gestionarea relațiilor generale și metadatelor dintre modele diferite (ex: autorul, data creării/modificării fișierului, metoda de creație, etc.).
* **Reverse Engineering Tool –** Un instrument destinat transformării modelelor ce conțin anumite informații vechi în modele cu drepturi depline.

 **Capitolul 3: Concepte de bază**

În acest capitol vor fi prezentate blocurile care stau la baza MDA. Acestea sunt UML 2.0, Meta Object Facility, XML Metadata Interchange, cele trei tipuri de modele (PIM/PSM/PDM), Multi-Stage formations, Action Languages, core models, model marking si Executable UML. [2]



Figura 2. Concepetele de bază ale MDA

**3.1. UML 2.0**

Din perspectiva MDA, UML este un punct central, deoarece multe unelte sunt sau vor fi bazate pe UML si profile. Pentru a se asigura că va funcționa , OMG a adaptat câteva puncte din contextul UML 2.0.

* Infrastructura: intern, UML nu mai este vag bazat pe MOF, standardele complete UML conțin definiții a constructorilor limbajului UML (metamodelul) via modelele MOF. UML este definit formal. Aceasta este o premisă pentru MDSD, în mod particular pentru transformarea de model și generarea de cod. Acum, OCL folosește MOF ca meta modelul său. UML si OCL sunt acum bazate pe același meta meta model. Astfel, ele sunt compatibile la nucleul lor.
* Extensii, profile, stereotipuri. Definiția și în parte notația profilelor și a stereotipurilor – mecanismul nativ de extindere a UML – a fost refăcut.

Chiar dacă formal toate modelele bazate pe MOF pot fi folosite în contextul MDA, UML și profilele corespondente pentru modeling în domeniul MDA vor predomina. Avantajele acestora sunt:

* Separarea sintaxei concrete de cea abstractă
* Posibilitatea de extensie (prin profile)
* Independența de platformă
* Standardizarea

Pe de altă parte, există numeroase dezavantaje care se aplică în special la UML 1.x. Multe dintre acestea au fost îmbunătățite cu introducerea UML 2:

* UML este mare și prost partiționat
* Nu există suport conceptual pentru viewpoints
* Componentele și șabloanele primesc foarte puțin suport
* Modelul relațional este vag
* Sufera de expresivitate limitată a profilelor, sau, în general, de mijloace limitate de adaptare a metamodelului
* UML și MOF nu sunt fin reglate una față de cealaltă
* Interoperabilitatea diagramelor nu există
* Nu există sintaxă abstractă pentru OCL [2]

**3.2. MOF – Meta Object Facility**

 MOF constituie nucleul MDA. Acesta descrie meta meta modelul pe care uneltele conformante MDA sunt construite. Definiția metamodelelor DSL personalizate ar trebui sa folosească mecanismele MOF, sau să extindă UML prin profile, care constituie o extensie de metamodel ”ușoară”. Cum am spus și mai devreme, UML este definit prin MOF. MOF folosește sintaxa concretă a UML, care poate cauza confuzii. De fiecare dată când extindem metamodelul UML, folosim în mod automat MOF.



Figura 3. O parte din MOF

 MOF nu este important doar ca baza formală a metamodelelor, este de asemenea relevant și pentru construirea uneltelor MDA, cum ar fi repozitoriile, unelte de modelare, generatoare de cod, și așa mai departe. Uneltele generale au nevoie de o bază solidă: aceasta poate fi doar meta modelul. Similar, pentru a garanta portabilitatea datelor folosite pentru unelte, trebuie să se hotărască asupra unui meta model. Deci, este esențial pentru eforturile de standardizare ale OMG și pentru piața de unelte să se defineasca un meta meta model corect și formal.

 MOF are și câteva dezavantaje. De exemplu, MOF nu ofera niciun ajutor pentru definirea unei sintaxe concrete pentru DSL-uri sau pentru probleme de versionare, iar compoziția metamodelelor din metamodele parțiale nu este adresată.

Au fost create diferite implementări ale MOF. Totuși, toate au implementat doar un subset practic relevant a MOF. Poate cea mai influentă implementare a fost EMF, Eclipse Modelling Framework și meta meta modelul eCore. Implementarea EMF a avut ca urmare o influență asupra standardizării MOF 2.0. [2]

**3.3. XMI**

 XMI vine de la XML Metadata Interchange și este o mapare XML pentru MOF – nu doar o schema pentru UML. Momentan, XMI este baza interoperabilității dintre diferite unelte MDA pentru că repozitoarele MOF nu sunt folosite în mare. De la lansarea versiunii 2.0, XMI permite serializarea informației din diagrama de layout, lucru obligatoriu pentru un schimb practic și folositor între unelte de modelare,nu doar pentru generearea de cod.

 La momentul actual încă există multe incompatibilități între formatele XMI a diferitelor unelte, ceea ce complică schimbul de diagrame dintre uneltele de modeling. Totuși, folosirea XMI ca bază pentru generarea de cod nu este o problemă gravă în practică, din moment ce majoritatea generatoarelor popilare suportă parsatoare pentru diferitele dialecte XMI.

 Există două tipuri de XMI. Primul este un tip complet generic care poate stoca toate modelele bazate pe MOF printr-un DTD generic. Structura documentului este definită la nivel de MOF, rezultând documente XMI destul de încărcate. Un lucru pozitiv este ca poate fi aplicat, în general, la toate modelele bazate pe MOF.

 Standardul XMI cuprinde opțiunea de a genera un DTD sau o schema specifică pentru un metamodel dat, bazat pe MOF. Ca o consecință, documentele pot stoca doar instanțe ale acestui metamodel particular, dar fișierul rezultat este mai compact și concret, deoarece structurile sunt mapate la nivelul de metamodel și nu la cel de meta meta model. Majoritatea uneltelor folosesc prima variantă pentru schimburi. [2]

**3.4. PIM/PSM/PDM**

 OMG are un concept concret pentru ce ar trebui MDA să reprezinte. OMG este interesat doar de independența de platformă a logicii de aplicație. Deoarece soluțiile tehnologice – ca și logica de afaceri – continuă să se dezvolte în mod rapid, dar independente una de alta, este necesar din motive de longevitate să se spevifice o logică de aplicație independentă de platforma de implementare. Cu acest scop în minte, OMG consideră MOF – sau modeling bazat pe UML – ca cea mai bună soluție, deoarece permite generarea automată a implementărilor pentru diferite platforme, prin transformatoare. Modelul independent de platformă (PIM – Platform Independent Model) joacă un rol central în descrierea logicii de afacere, neperturbată de motive de îngrijorare tehnice.

 Un model specific unei platforme (PSM – Platform Specific Model) este creat din PIM prin transformarea de model. PSM este specific platformelor, cum ar fi J2EE, .NET sau alte platforme de implementare. Alte transformări pot crea modele mai specifice, până când este generat codul sursă pentru o platformă, care este transformat într-un obiect executabil prin compilare și ambalare (packaging). [2]

 

Figura 4. Clasificarea modelelor în contextul MDA

Un alt tip de model important care există în MDA este modelul de descriere a platformei (PDM). Acesta este un metamode al platformei țintă. Deoarece transformările de model sunt mereu definite ca reguli de transformare între metamodele, este esențial să definim platforma țintă printr-un metamodel. [2]

**3.5. Multi-stage transformation**

 MDA are ca scop obținerea codului sursă prin diferite transformări model-model. Este clar că există avantaje, dacă se folosesc uneltele necesare. Motoarele de transformare sunt definite pe baza MOF. Cât timp aveam de-a face cu metamodele bazate pe MOF, astfel de unelte pot fi folosite. Pentru a folosi aceste motoare de transformare în cazul programării, ar trebui ca limbajele clasice de programare să fie definite prin MOF, ceea ce ar avea ca rezultat că transformările ar deveni foarte complexe comparativ cu șabloanele simple.

 În anumite cazuri este necesar să se configureze manual produsul intermediar pentru a controla următoarele stagii de transformare. OMG numește astfel de configurații model markings. Model markings nu pot fi adnotate direct în PIM, deoarece ar duce la pierderea independenței de platformă. Din motive de consitență, este de asemenea critical să nu se modifice modelele intermediare. [2]

**3.6. Action Languages**

Rămâne nerezolvată problema specificării comportamentului algoritmilor. Pentru a se adresa acestei probleme, OMG a definit semanticile de acțiune (action semantics) cu UML 2.0, care pot fi folosite și cu alte limbaje bazate pe MOF. Semanticile de acțiune permit modelarea comportamentului procedural. OMG definește doar sintaxa abstractă, nu una concretă. Semanticile sunt descrise verbal. Deci, este la latitudinea producătorilor de unelte să își definească notațiile grafice sau prin text, pentru o semantică standardizată. Prin urmare, este posibil să se reprezinte același comportament atât prin text, cât și grafic.

 Semantica de acțiune cuprinde următoarele elemente:

* Variabile (instance handles): atribuirea, citirea, cât și pentru seturi de variabile (sets, bags, sequences)
* Operațiile aritmetice și logice uzuale
* Trăsături tipice ale limbajelor de programare secvențiale, cum ar fi switch, if, for și conceptul de block
* Creare și ștergerea de instanțe
* Extinderea de clase, prin interogări de tip SQL
* Navigarea prin asocieri
* Crearea de legaturi și ștergerea acestora
* Generarea de semnale, inclusiv parametrii
* Definirea funcțiilor cu parametrii de intrare și ieșire, plus metode prin care pot fi apelate
* Cronometre

Semanticile de acțiune nu conțin construcții structurale precum clasele, atributele si relațiile. Acestea sunt deja definite în partea structurală a modelului. Semanticile de acțiune doar definesc ”blocuri de comportament” care au sens doar în asociere cu alte modele parțiale. Ca o consecință, segmentele de semantici de acțiune sunt mereu asociate cu elemente din modelul obișnuit de UML.

În figura de mai jos este prezentată o diagramă de clasă, care este folosită pe post de bază a exemplului.



Figura 5. Model pentru ilustrarea semanticii de acțiune

În primul rând, implementăm un program principal, care lucrează cu instanțe ale claselor din figura de mai sus și crează o instanță a clasei *Vehicle*. Apoi îi atribuim valori pentru atributele make și model.

myVWBus = create Vehicle with plate = ”HDH-GS 142”

myVWBus.make = ”Volkswagen”

myVWBus.model = ”Transporter T4 TDI”

 Clauza with atribuie valori atributelor care oferă identificarea în momentul creării de obiecte, în mod similar cu transmiterea de parametrii constructorilor în limbajele Obiect Orientate. Apoi putem defini o instanță de persoană care va deveni șoferul.

 Putem acum apela metoda drive() pentru a face șoferul să conducă vehiculul.

[actualDriver] = drive[aVehicle] on john

Ceea ce lipsește este implementarea metodei drive(). Această implementare trebuie, cel puțin, să instanțieze o asociere R1, adică să creeze o legătură între două obiecte relevante.

Link this R1 aVehicle

Acest lucru stabilește asocierea bidirecțională dintre *Vehicle* și un șofer. Astfel, se poate interoga mașina pentru a afla cine o conduce. Acesta corespunde cu implementarea metodei drive() pentru clasa *Vehicle*.

theCurrentDriver = this.R1.”driver”

Urmează o scurtă introducere a operațiilor de interogare. Presupunem că vrem să gasim toți indivizii din sitem:

 {allPersons} = find-all Person

 Parantezele specifică că acesta este un set de obiecte, nu doar unul. Este de asemenea posibil să se limiteze o astfel de căutare. De exemplu, toate vehiculele de marcă Audi pot fi căutate:

 {audis} = find Vehicle where make = ”Audi”

 Deși aceste semantici de acțiune reprezintă un alt limbaj de programare, există niște diferențe:

* Nu trebuie să se ia în considerare atribute specifice platformei, cum ar fi managementul de memorie, definirea unei conexiuni dintre doua instanțe EntityBean, sau folosirea de chei relaționale
* Avem de-a face cu blocuri semantice. Sintaxa concretă poate arăta diferit
* Semanticile de acțiune sunt integrate complet în model



Figura 6. Alt exemplu al unui model

 În timpul mapării pe un limbaj de programare, aproximativ următorul cod este generat:

Linia 1: Crearea unei noi instanțe a clasei A, crearea variabilei a

Linia 2: Crearea unei noi instanțe a clasei B, crearea variabilei b

Linia 3: Atributului theB îi este atribuit un pointer la b de către a. Din moment ce asocierea este bidirecțională, b.theA va pointa automat către a. Acesta este un lucru ce nu trebuie specificat explicit de programator – generatorul poate face asta automat folosind informația din diagrama de clase.

Linia 4: Deoarece B este asociat compozițional cu A, generatorul poate crea cod care să șteargă instanța b când a este ștearsă.

Informația pentru generarea de cod este de asemenea luată și din diagrama de structuri. Dezvoltatorul trebuie doar să scrie afirmația link în ASL. [2]

**3.7. Core Models**

 Pentru a beneficia la maxim de MDA, cât mai multe aspecte trebuie să fie standardizate. Aici sunt incluse platformele, lucru deja făcut prin J2EE, .NET, COBRA sau Servicii Web, cel puțin la nivel tehnic, si prin limbaje de transformare, care se întampla în contextul QVT. Pentru a permite utilizatorilor să modeleze logica de aplicație independent de platformă, este necesar să se standardizeze metamodelul pentru anumite domenii. Rezultă că nu devinde posibil doar să se standardizeze limbajele de transformare, dar să se și captureze reguli de transformare refolosibile în biblioteci de transformare. Dezvoltatorul modelează aplicația prin profilul UML standardizat sau prin metamodelul domeniului respectiv, iar module generice comerciale sau open source generează codul specific platformei din ea. [2]

 Metamodelele standardizate sunt numite modele nucleu de OMG în contextul MDA. Numearoase modele nucleu, care sunt în prezent definite ca profile UML, sunt dezvoltate în mod curent. Printre acestea se numără:

* Profile UML pentru COBRA, care definesc maparea PIM-urilor pentru COBRA
* Profile UML pentru CCM, care definesc maparea pentru CCM, componenta model a COBRA
* Profile UML pentru EDOC, care definesc metamodelul pentru definiția PIM-urilor pentru sisteme enterprise distribuite
* Profile UML pentru EAI, care definesc metamodelul pentru PIM-uri pentru aplicații EAI
* Profile UML pentru Quality of Service (QoS) si pentru Fault Tolerance, pentru sistem în timp real și sisteme de securitate
* Profile UML pentru Schedulability, Performance and Time, care definesc metamodel pentru PIM-uri ale căror proprietăți în timp real pot fi analizate cantitativ
* Profile UML pentru testare, care suportă testarea automată pentru sistemele bazate pe MDA

**3.8. Controlul transformării din PIM în PSM**

 În anumite cazuri, transformatorul nu poate transforma un model deoarceve informația din modelul sursă nu este suficient de specifică. Există patru alternative pentru evitarea acestei situații:

* Se poate codifica în transformare că există o alternativă particulară care va fi mereu folosită
* Dezvoltatorul poate specifica manual care dintre alternative poate fi folosită
* Dezvoltatorii pot afirma direct în PIM ce alternative să fie folosite în PSM ca model markings (marcări de model)
* Criteriul de decizie care permite transformatorului să decida ce alternativă să fie folosită poate fi abstractizat în PIM

**Model Markings**

MDA propune conceptul de model markings. Acestea reprezintă informație adițională în modelul sursă al transformatorului, care controlează transformarea. Aceste anotații depind de metamodelul țintă.

****

Figura 7. Exemplu de model markings

Un alt exemplu este reprezentat de J2EE. În PIM definim o *BusinessEntity* numit *Account*. În J2EE, *BusinessEntities* poate fi reprezentată în două feluri: fie ca un *EntityBeans* sau ca un *Stateless Session Beans* care procesează obiecte de transfer de date.

****

Figura 8. Un exemplu de entitate cu marcaje specifice EJB

Este important să se asigure că modelul sursă nu este modificat; în schimb, este definită o copie referință care conține informație suplimentară.



Figura 9. Relația dintre PIM și PIM marcat

PIM marcat conține doar elemente model ale PIM-urilor originale care trebuie marcate. Asta înseamnă că PIM marcat nu trebuie să fie adaptat manual dacă este modificat PIM-ul original. [2]

**3.9. Executable UML**

 Termenul Executable UML este de foarte multe ori întâlnit în contextul MDA. Acesta nu este un standard formal, ci un termen colectiv pentru diferite încercări care au ca scop stabilirea UML ca limbaj de programare. UML trebuie să fie curățat de toate ambiguitățile și redundanțele, rezultând în posibilitatea de executare a diagramelor UML: cu cât metamodelul limbajului de modelare este mai mic (în cazul nostru UML), cu atât este mai ușor să se implementeze un compilator sau un interceptor pentru el. Un alt ingredient necesar pentru UML executabil este un limbaj de acțiune, care este necesar să definească implementări complete ale sistemelor software. [2]

**Capitolul 4: Tipuri de arhitecturi**

**4.1. Business Motivation Model (BMM)** sau **Modelul Motivațional al Proceselor de Lucru**

BMM reprezintă un standard adoptat de OMG (Object Managaement Group). Acesta permite ca noile planuri de afaceri dezvoltate de catre antreprenori sa fie *dezvoltate,comunicate* și *gestionate* respectând strategiile de lucru ale organizației. BMM poate conține informații din orice departament: cerințe de administrare, cerințe de dezvoltare ale aplicațiilor, cerințe nefuncționale etc. [3]

Pentru dezvoltarea unui BMM se dorește urmărirea modulelor:

* **Rezultate:** Ce doreste întreprinderea(organizația) să îndeplinească (obiectivul care se urmărește).
* **Mijloace:** Prin ce modalități intenționează organizația să îndeplinească rezultatele dorite.
* **Directive:** Regulile și politicile care constrâng și/sau guvernează mijloacele disponibile.
* **Evaluare:** Evaluarea mijloacelor de a atinge rezultatele și potențialul impact rezultat.
* **Influențe:** Ce concepte și tehnologii pot influența evaluarea.

Toate procesele de lucru care au loc într-o organizație pot fi reprezentate grafic facilitând întelegerea întregului modul de lucru existent sau planificat. Cele mai utilizate notații grafice pentru reprezentarea proceselor de lucru sunt **BPMN (Business Process Management Notation)** sau **YAWL (Yet Another Workflow Language)**. [3],[4].

Conceptele de bază folosite în dezvoltarea unei arhitecturi de business sunt:

* Actori – utilizatori sau servicii automatizate care operează procesele de lucru prin intermediul interfețelor grafice sau serviciilor web.
* Activități – acțiuni ce pot fi executate de către orice tip de *actor*.
* Evenimente – stare ce poate apărea pe parcursul unui proces
* Reguli de proces – noduri de decizie în care *actorul* este obligat să-și aleagă o opțiune
* Mesaje – formă de comunicare care interconectează secvențele de *activități*

Ciclul de viață al unei structuri **BPM** poate avea următoarea structură:

* Analiza – selectarea obiectivelor și organizarea lor în procese de lucru
* Modelarea proceselor de lucru – alegerea unui model cât mai reprezentativ pentru situațiile din realitate
* Implementarea soluției – integrarea soluției în sistemele informatice ale companiei
* Execuția – implementarea pentru toate instanțele de proces
* Monitorizarea – analizarea proceselor de lucru cu ajutorul indicatorilor de performanță
* Optimizarea – îmbunătățirea performanțelor proceselor de lucru

Figura 10. Modelul Motivațional de Business

**4.2. Oriented Architecture (SOA)** sau **Arhitectura Orientată pe Servicii**

SOA reprezintă un tip de arhitectură software ce presupune o realocare a funcționalității programului în module numite servicii. Acestea se pot distribui în rețea pentru a putea fi accesate concomitent și utilizate împreună pentru rularea unor aplicații destinate mediului de business.

Acest tip de design software introduce posibilitatea refolosirii funcțiilor din procesele de lucru pentru diferite tipuri de platforme. Serviciile reprezintă al treilea nivel în arhitectura SOA față de arhitecturile bazate pe componente. [5]

Figura 11. Structura SOA

SOA este caracterizată ca fiind o arhitectură flexibilă ce contribuie la o bună interconectare a diverselor aplicații ale mediului de business, facilitând schimbul de informații între acestea. Aceste pot fi folosite de diverse grupuri de angajați și/sau parteneri atât din cadrul companiei cât și din afara ei.

Interacțiunile dintre servicii sunt axate pe impotanța mesajului. În schimbul definirii unor anumite tipuri de date, SOA se folosește de structuri (scheme) pentru formarea bazelor de date necesare schimbului de informații dintre module, serviciile devenind astfel interoperabile. [5]

**4.3. Arhitectura acționată de evenimente (EDA – Event Driven Architecture)**

Este o arhitectură software care promovează producția, detecția, consumul și reacția la evenimente.

 Un eveniment poate fi definit ca o schimbare semnificantă a statusului. De exmplu, când o mașină este cumpărată, ea își schimbă statusul de la ”de vânzare” la ”vândută”. O arhitectură de sistem a unei reprezentanțe auto poate trata această schimbare de status ca un eveniment a cărei apariție poate fi făcută cunoscută altor aplicații din cadrul arhitecturii.

 Șablonul arhitectural poate fi aplicat de proiectarea și implementarea aplicațiilor și sistemelor care transmit evenimente între componente și servicii legate slab între ele. Un sistem acționat de evenimente consistă, de obicei, din emițătoare de evenimente (sau agenți), consumatori de evenimente și canale de evenimente. Emițătorii au responsabilitatea să detecteze, adune și transfere evenimente. Un emițător de evenimente nu știe consumatorul de eveniment, nu știe nici măcar dacă acesta există, iar în caz că există, nu știe cum va fi folosit evenimentul în continuare. Consumatorii de evenimente au responsabilitatea de a aplica o reacție imediat ce apare un eveniment.

 Construirea aplicațiilor și sistemelor în jurul unei arhitecturi acționată de evenimente permite aplicațiilor și sistemelor să fie construite într-o manieră care permite reacții rapide, deoarece sistemele acționate de evenimente sunt proiectate pentru medii imprevizibile și asincrone. [1]

**Bibliografie**

[1] http://infospheres.caltech.edu/sites/default/files/Event-Driven%20Applications%20-%20Costs,%20Benefits%20and%20Design%20Approaches.pdf

[2] Model-Driven Software Development – Thomas Stahl, Markus Volter

[3] Modelarea organizației si ingineria software pentru îmbunătățirea sistemelor informatice – Ion Lungu

[4] [www.omg.org](http://www.omg.org)

[5] Model Driven Software Development – Thomas Stahl