**UNIVERSITATEA “ALEXANDRU IOAN CUZA” DIN IAȘI**

**FACULTATEA DE INFORMATICĂ**

Domain Driven Design Rich Model

*Cristian Humă*

**Sesiunea:** *iulie, 2018*

**Coordonator științific**

*Drd. Colab. Florin Olariu*

DECLARAŢIE DE CONSIMŢĂMÂNT

Prin prezenta declar că sunt de acord ca Lucrarea de licență cu titlul „*Domain Driven Design Rich Model*”, codul sursă al programelor şi celelalte conţinuturi (grafice, multimedia, date detest etc.) care însoţesc această lucrare să fie utilizate în cadrul Facultăţii de Informatică.

De asemenea, sunt de acord ca Facultatea de Informatică de la Universitatea „Alexandru Ioan Cuza” din Iași, să utilizeze, modifice, reproducă şi să distribuie în scopuri necomerciale programele-calculator, format executabil şi sursă, realizate de mine în cadrul prezentei lucrări de licenţă.

Iaşi, *27.06.2018*

Absolvent *Cristian Humă*

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(semnătura în original)

Cuprins

[CONTRIBUȚII 7](#_Toc517857924)

[Metoda tradițională de dezvoltare 7](#_Toc517857925)

[Metoda DDD folosind un model bogat 7](#_Toc517857926)

[ASPECTE TEORETICE 8](#_Toc517857927)

[Domain Driven Design 8](#_Toc517857928)

[Principalele Concepte 9](#_Toc517857929)

[Harta Contextelor 10](#_Toc517857930)

[Modelul domeniului 11](#_Toc517857931)

[Tipuri De Obiecte 13](#_Toc517857932)

[Servicii de domeniu 14](#_Toc517857933)

[Agregate 15](#_Toc517857934)

[Repositories 18](#_Toc517857935)

[Concluzii 20](#_Toc517857936)

[ORM 21](#_Toc517857937)

[Arhitectura Onion 21](#_Toc517857938)

[Concluzii 23](#_Toc517857939)

[MVC 23](#_Toc517857940)

[Microservicii 24](#_Toc517857941)

[ASPECTE PRACTICE 26](#_Toc517857942)

[Tehnologii 26](#_Toc517857943)

[Controller 27](#_Toc517857944)

[Entitați 29](#_Toc517857945)

[Mecanism unic de validare 30](#_Toc517857946)

[ORM 31](#_Toc517857947)

[Eliminarea logicii din servicii 33](#_Toc517857948)

[ÎMBUNĂTĂȚIRI 35](#_Toc517857949)

[Mecanism unic de tratare a erorilor 35](#_Toc517857950)

**INTRODUCERE**

**Tema Licenței**

Domain Driven Design reprezintă o abordare pentru dezvoltarea software organizată în jurul unui domeniu prin care arhitectura sistemului urmărește încapsularea și vizibilitatea conceptelor precum și prevenirea duplicării codului, toate conducând spre o dezvoltare mai ușoară și mai sigură pe termen lung. Această abordare caută să elimine breșe pe care un programator le-ar putea introduce oferind o arhitectură care să reflecte logica aplicației. În cazul de față se bazează pe un model bogat ce presupune aducerea informațiilor aproape de operațiile care lucrează pe ele.

**Motivația Alegerii Temei**

Ultimii ani industria IT s-a bucurat de o creștere enormă, fapt ce a condus la aplicații din ce în ce mai mari, cu milioane chiar miliarde de utilizatori. La o asemenea scară provocările nu sunt puține iar costurile nu sunt mici, fapt ce a condus spre crearea unei metodologii sigure și eficiente pentru a putea continua dezvoltarea a unor sisteme de-a lungul anilor: domain driven design(DDD). În contextul globalizării secolului XXI, nevoia de aplicații mari, complexe și scalabile este inevitabilă iar DDD-ul își propune să le rezolve oferind un ghid riguros și bine documentat.

Am ales această temă deoarece consider ca un sistem să fie de succes el trebuie să comunice și să lucreze cu alte sisteme externe dar să aducă propria valoare în această relație; nu este suficient ca un sistem să realizeze o acțiune specifică lui, ci trebuie să fie capabil să transmită informațiile sale altor sisteme, pentru a crea o experiență plăcută fără a-l obliga pe utilizator să folosească alte sisteme decât cele utilizate până în acel moment. Din această perspectivă am putea considera că experiența utilizatorului ar trebui să fie asemenea cu dezvoltarea: iterativă. Odată familiarizat cu un sistem, dezvoltatorii ar trebui să profite din plin de acest aspect și să continue experiența utilizatorului cu alte sisteme/funcționalități care să comunice cu sistemele deja prezente.

Alt motiv al alegerii acestei teme constă în faptul că abordarea unor aplicații mari este de cele mai multe ori intimidantă iar foarte multe decizii mărunte, aparent neimportante la momentul discuției, conduc la o dezvoltare haotică și cu multe dezavantaje. Domain driven design nu reprezintă o soluție perfectă pentru astfel de probleme dar oferă argumente și mecanisme, la nivel de concept, ce pot fi implementate în orice sistem pentru a crește calitatea dezvoltării.

O implementare foarte populară a conceptelor acestei metodologii este reprezentată de microservicii, și cum acestea în ultimii ani au câștigat mult teren, ele oferind scalabilitatea căutată de dezvoltatori ce totodată conduce atenția dezvoltării asupra esenței problemei, celelalte aspecte (infrastructură, servicii externe, etc.) putând fi abordate independent unele de altele.

**Concluzii**

În final, această abordare a dezvoltării software caută să producă aplicații mari, scalabile fără a introduce problema complexității și de a oferi posibilitatea paralelizării dezvoltării prin echipe multiple, autonome.

**Pagina codului sursă**

<https://github.com/CristiHuma96/DomainDrivenDesign>

# CONTRIBUȚII

## Metoda tradițională de dezvoltare

Modul în care multe aplicații de dimensiuni mari sunt dezvoltate, cu sau fără voia celor care o dezvoltă, poate produce multe costuri pe termen lung, fie că vorbim de timp sau de resurse financiare. Un mod de dezvoltare creează un model al aplicației iar această lucrare aduce în perspectivă o problemă des întâlnită: folosirea unui model anemic.

Un model anemic constă în separarea datelor de operațiile care lucrează pe ele, abordare ce conduce la încălcarea unor principii ale programării orientate obiect și la apariția unor probleme pe termen lung:

* vizibilitatea operațiilor – un nou membru al echipei nu poate observa foarte ușor ce operații sunt permise pentru un anumit obiect, pentru anumite date;
* duplicarea codului sursă – neavând vizibilitatea asupra operațiilor deja definite, un comportament asemănător poate fi creat din nou, conducând astfel spre probleme de consistență și posibile breșe de securitate;
* lipsa încapsulării – protecția integrității datelor este realizată de încapsulare iar lipsa acesteia conduce la una dintre cele mai mari probleme în dezvoltarea software: complexitatea mare. O complexitate mare produce o dezvoltare încetinită, un număr mai mare de erori și afectează în mod negativ abilitatea de adaptare la nevoile business-ului

## Metoda DDD folosind un model bogat

Un „nou” mod de a dezvolta și a face managementul facil unei aplicații de dimensiuni mari o reprezintă folosirea unui model „bogat”.

Un model bogat constă în centralizarea datelor și a operațiilor într-un singură structură iar aici intervin multiple concepte din domain driven design menite de a obține o arhitectură bine definită și care să ofere un mod clar și bine controlat asupra accesului datelor; acestea vor fi prezentate în următoarele capitole.

**Concluzii**

Documentarea a presupus citirea și înțelegerea mai multor concepte din domain driven design pentru a putea formula o explicație completă asupra modului în care aceasta funcționează și a demonstra valoarea pe care o aduce în cadrul dezvoltării dacă este folosită în cazurile specificate.

# ASPECTE TEORETICE

## Domain Driven Design

Domain Driven Design(DDD) este o metodologie, un set de reguli și bune practici, pentru dezvoltarea software, organizată în jurul unui **domeniu**, a unei sfere de cunoștințe, bazat pe un **model**. Un model reprezintă un sistem de abstracții ce descriu aspecte ale domeniului ales și care pot fi folosite pentru a rezolva probleme legate de acesta. Această abordare presupune o **comunicare strânsă cu experții domeniului** pentru a obține soluții la problemele puse în discuție. Totuși, în cele mai dese cazuri, o aplicație software implică mai multe funcționalități, mai multe domenii, și deci de aici apare principiul de **sub-domenii**, de **contexte**, ce reprezintă sub-problemele și respectiv soluțiile aferente lor. În tot acest proces, în cadrul unui domeniu sau

sub-domeniu, este necesară folosirea **aceluiași limbaj** pentru a elimina confuziile.

Aceasta abordare a fost introdusă pentru prima oara în 2004 de Eric Evans în cartea sa intitulată „Domain Driven Design”. În figura 1 se regăsește „o hartă” a acestei metodologii, descrise de autorul cărții.

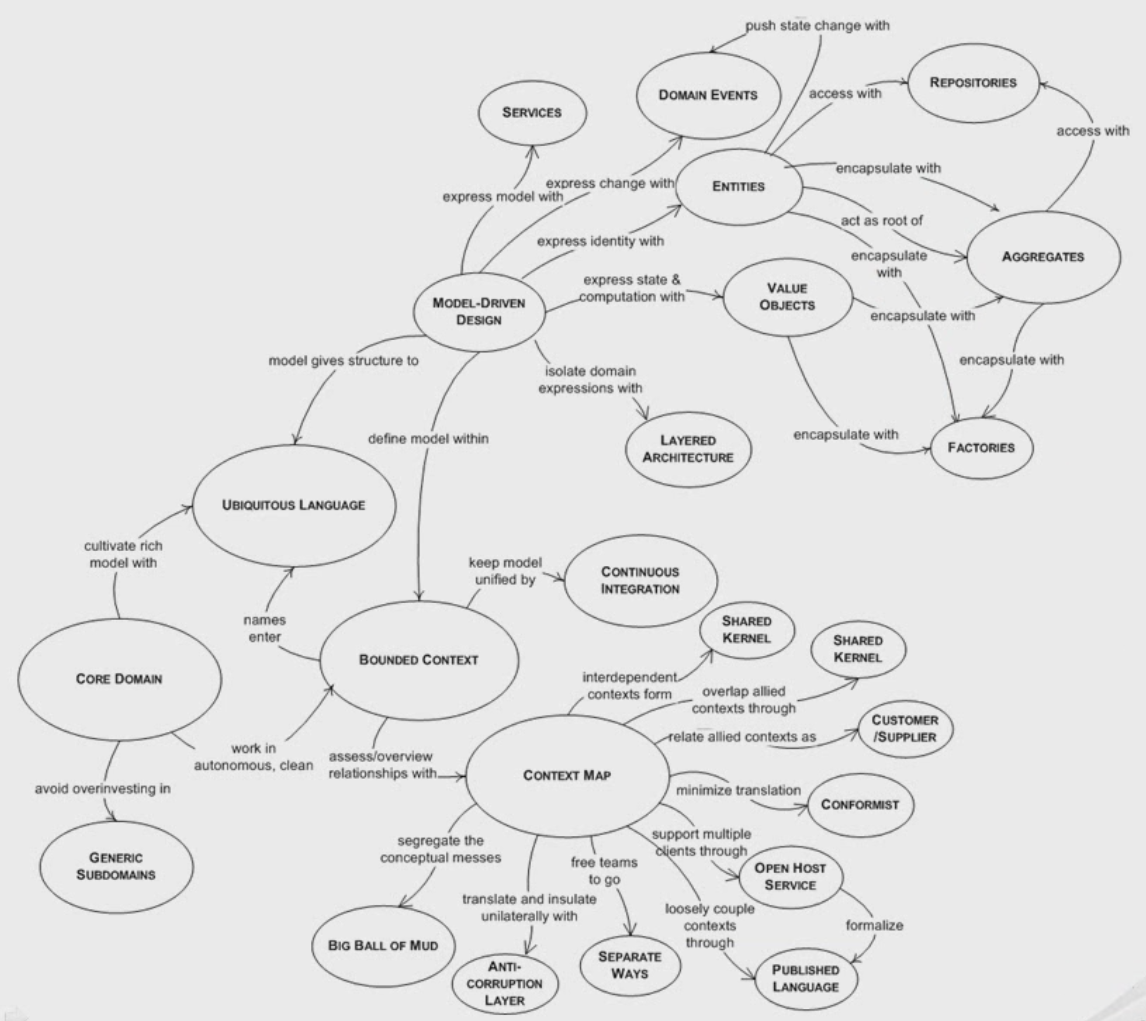


Figura 1. Domain Driven Design Harta [[1]](#_Bibliografie)

### Principalele Concepte

În DDD descoperim 4 concepte mari, acestea fiind:

* **model** **-** un sistem de abstracții ce descrie aspectele domeniului nostru ce pot fi folosite pentru a rezolva problemele vizate legat de acel domeniu;
* **ubiquitous language -** crearea unui model care să reprezinte corect domeniul nostru presupune o comunicare strânsă cu experții domeniului nostru cât și în cadrul echipei de dezvoltare, de unde apare nevoie unui limbaj **comun** si **bine** **definit**, pentru a evita confuziile;
* **contexte delimitate** – odată cu creșterea unei aplicații, mentenanța devine o problemă reală iar anumite elemente de cod ce au sens într-o parte a sistemului nu au relevanță în alte parți; aici intervine noțiunea de context delimitate, ce ne permite delimitarea și implicit împărțirea soluției noastre în diverse componente/module;
* **domeniul principal** – partea esențială a unui sistem este partea ce conține logica business-ului ce tratează aspectele importante pe care aplicația noastră încearcă să le rezolve și pentru care este dezvoltată.

Un prim pas al dezvoltării folosind DDD este constituit din:

* **prioritizarea construirii contextului principal**, funcționalitatea ce este reprezentativă pentru aplicația noastră; acesta **nu ar trebui împărțită** de la bun început **în multiple contexte delimitate**; deși contextele delimitate rezolvă problema complexității, acestea ar trebui folosite doar în cazul în care codul sursa devine suficient de mare/complex;
* Un concept important din POO(programarea orientată obiect) ce este pus în aplicare îl reprezintă **abstractizarea** și se referă la extragerea unei element din realitate într-o structură ce ascunde detalii neimportante pentru funcționalitatea dorită și expune doar esențialul necesar dezvoltării. Această abordare căută simplificarea modelului pentru a lucra cu mai puține concepte la un singur moment dat.

### Harta Contextelor

Odată ce ne gândim la numeroasele sub-sisteme ale unei aplicații ca fiind contexte delimitate ne putem pierde perspectiva globală asupra aplicației. O hartă a contextelor reprezintă această perspectivă ce ține cont de toate contexte delimitate și de comunicarea care o realizează între acestea. Harta contextelor este un concept important din DDD deoarece urmărește evoluția aplicației din perspectiva sub-modulelor dezvoltate de către una sau mai multe echipe precum și comunicarea dintre ele.

O legătură între 2 sub-sisteme definește etapele procesului care afectează acele sisteme: schimbările din primul sistem afectează cel de al doilea sistem sau invers. Totuși această abordare ne oferă o vedere asupra modului în care se desfășoară comunicarea însă nu ne spune nimic legat de cantitatea de comunicare ce are loc între două sau mai multe sisteme. Această cantitate este uneori denumită sugestiv ca fiind „lățimea de bandă”. Lățimea de bandă adaugă o nouă dimensiuni hărții deoarece pune în evidență cele mai importante contexte din sistem, oferind o cale vizuală echipei de dezvoltare de a decide ce sistem poate fi modificat ușor cu impact minim asupra celorlalte.

Diferența între un context delimitat si un sub-domeniu este una clara dar de foarte multe ori confundata deoarece ele corespund unei alteia:

* un sub domeniu reprezintă **spațiul unei sub-probleme** a sistemului nostru;
* un context delimitat corespunzător acel sub-domeniu reprezintă **spațiul soluției** ce are scopul rezolvării problemei sub-domeniului.

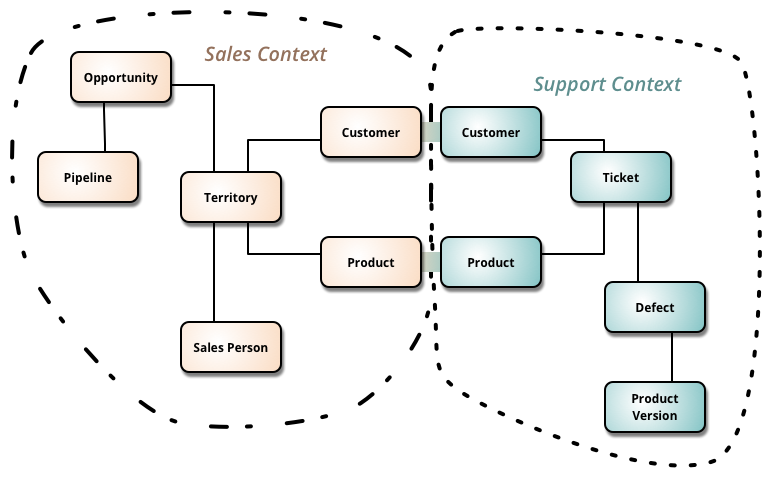


Figura 2. Exemplu Hartă Contexte [[2]](#_Bibliografie)

### Modelul domeniului

Abordarea dezvoltării este reprezentată de modelul domeniului ales, acest model constituind arhitectura sistemului acelui domeniu și al modului în care aceasta funcționează. În domain driven design apar 2 tipuri de modele:

* **anemic** – un model în care obiectele domeniului conțin puțină sau chiar nicio logică a business-ului; aceste modele sunt considerate anti-șabloane deoarece ele nu respectă principiile POO, așa cum afirmă și Martin Fowler: “The fundamental horror of this anti-pattern is that it's so contrary to the basic idea of object-oriented designing; which is to combine data and process them together”. Acesta poate fi caracterizat prin faptul că:
  + este intuitiv;
  + este ușor de implementat;
  + este potrivit pentru operații de tip CRUD;
  + este potrivit pentru aplicații mici și pe termen scurt;
  + nu este sugerat pentru aplicații mari deoarece nu nescalabil odată cu creșterea complexității.
* **bogat** – un model în care obiectele domeniului conțin toată logica business-ului care lucrează cu ele; aceste modele sunt preferate deoarece oferă vizibilitate asupra logicii business-ului și oferă totodată un control mai bun și mai granular al integrității datelor, conducând astfel spre o dezvoltare mai facilă. Acesta poate fi caracterizat prin faptul că:
  + oferă siguranța dezvoltării într-un mod constant;
  + respectă principiile POO;
  + oferă scalabilitate;
  + urmărește principiile DDD;
  + este potrivit pentru proiecte complexe de termen lung;
  + este costisitor;

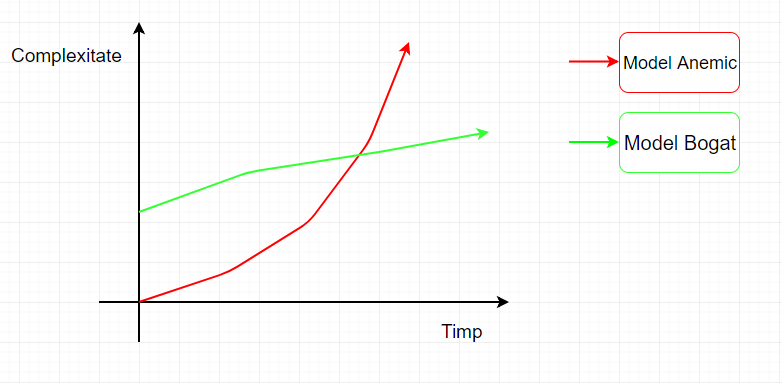


Figura 3. Anemic VS Bogat

### Tipuri De Obiecte

Abstractizarea este un element important în orice aplicație software astfel încât este important de identificat ce fel de obiecte trebuiesc folosite pentru diverse concepte ale domeniului nostru. DDD definește 2 tipuri de obiecte, ce se diferențiază în funcție de modul în care le identificăm: **entități** și **value** **objects**. Entitățile sunt obiecte ce au o identitate proprie, având un identificator unic și fiind mutabile. Value objects sunt obiecte ce se diferențiază unele de altele prin conținutul lor, ele fiind imutabile. Diferența dintre ele se poate observa vizual în figura 4.

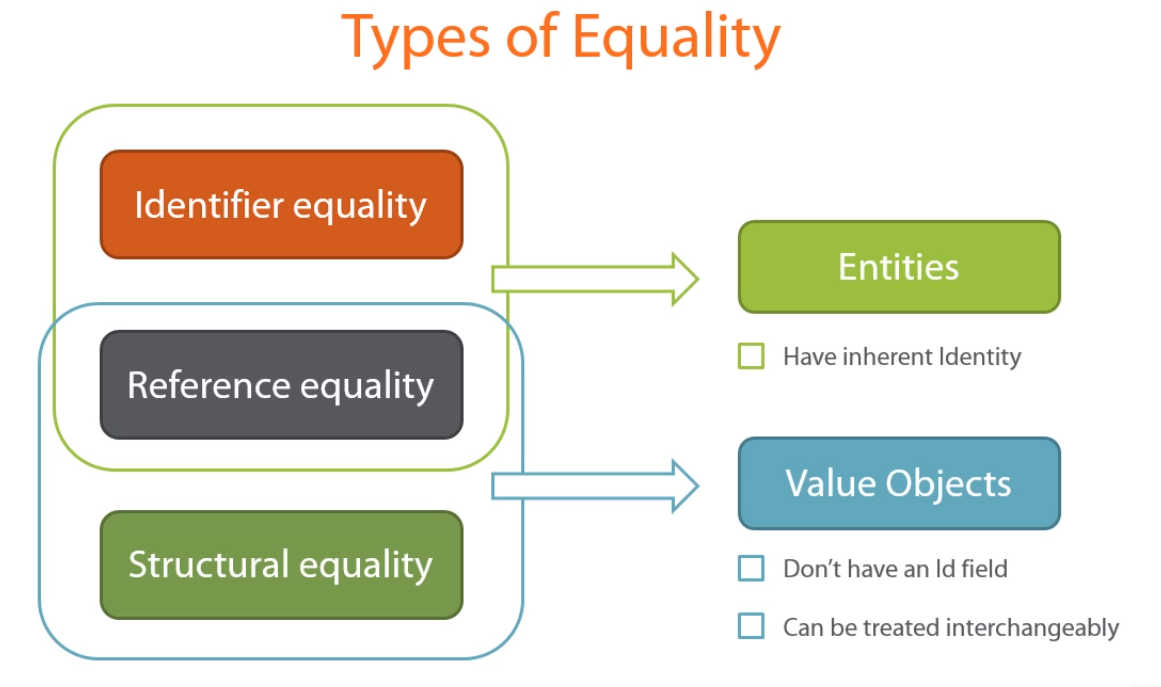


Figura 4. Entitați VS Value Objects [[4]](#_Bibliografie)

În general avem 3 tipuri de identificare/comparare a unor obiecte:

* Prin referință : 2 obiecte sunt egale daca au aceeași adresă în memorie
* Printr-un identificator: 2 obiecte sunt egale daca au același identificator(ID)
* Prin valoare: 2 obiecte sunt egale daca au același conținut(membrii coincid)

Un concept poate fi o entitate într-un domeniu și un value object în altul, însă tot timpul value objects trebuie să aparțină cel puțin unei entitați. Diferența între aceste 2 concepte este una subtilă și depinde de problema domeniului, dar în general decizia daca un obiect este un value object sau nu se rezumă la întrebarea daca acel obiect poate fi înlocuit în siguranța de alt obiect cu același conținut.

#### Importanța Value Objects

E de preferat folosirea value objects în locul entităților deoarece acestea sunt imutabile și mai puțin costisitoare, astfel încât reprezintă un loc ideal pentru logica de business. Totuși entitățile ar trebui sa fie containerele acestor value objects și să ofere un o privire de ansamblu asupra funcționalităților.

Mutarea unor concepte care tind sa fie folosite și modificate împreuna în aceeași structură, constitue un value object. Aceasta abstractizare ajută la simplificarea problemei domeniului deoarece grupăm multiple concepte într-un loc și avem de a face cu mai puține concepte la un moment dat. Totodată această abordare oferă multiple beneficii:

* Identificarea problemelor mai rapid la compilare: folosirea greșită a datelor noastre este identificată la compilare, când sistemul așteaptă un anumit tip de obiect, însă ii este furnizat altul; aceasta problemă ar putea apărea la execuție și ar fi mai greu de identificat;
* Modelul devine mult mai expresiv pentru că structura obiectelor corespunde cu ideile business-ului;
* Evitarea duplicării logicii domeniului, aceasta fiind plasată într-un singur loc; obținem astfel o structura unitară ce nu oferă breșe de integritate sau o interpretare diferită de cea a business-ului de care se poate folosi tot contextul;

### Servicii de domeniu

Când o operație este importantă pentru modelul domeniului, dar nu aparține niciunei entități, această responsabilitate revine unui serviciu de domeniu. Un astfel de serviciu are rolul de a orchestra operații care lucrează cu multiple entitați sau value objects însă această decizie trebuie să fie cântărită bine deoarece un număr mare de servicii conduce la un model anemic, unde comportamentul sistemului nostru se află în servicii.

Un serviciu de domeniu este diferit de alte servicii deoarece acesta conține o parte a logicii business-ului nostru și lucrează doar cu elementele ale acelui domeniu. O definiție oferită de Eric Evans se poate observa în figura 5 iar în figura 6 se pot observa diferite tipuri de servicii în cadrul unei aplicații.

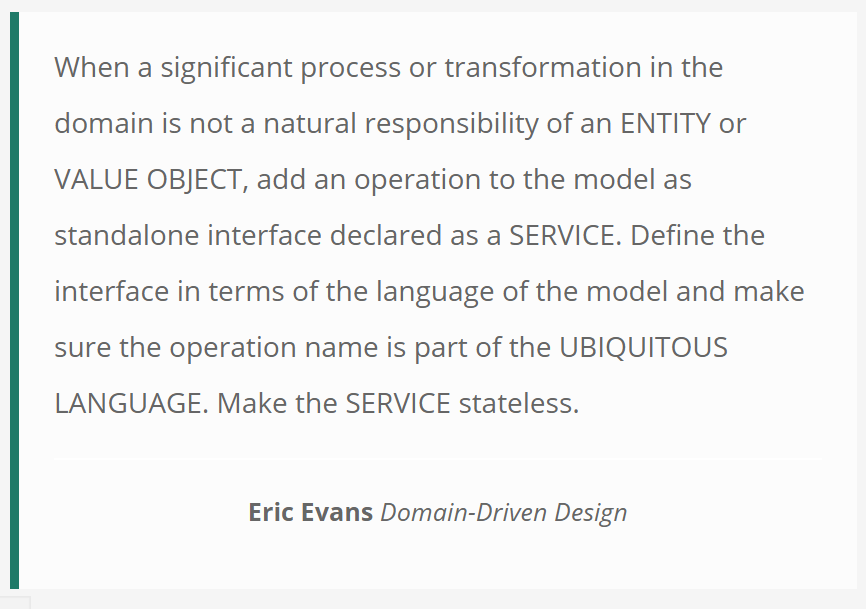
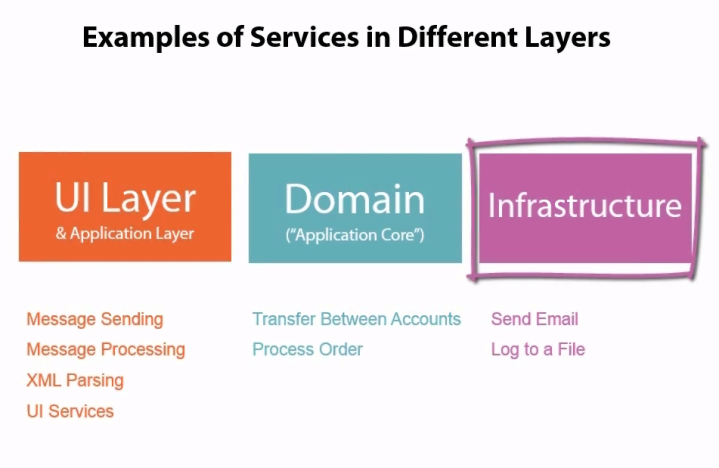
 

Figura 5. Definiție Domain Services [[5]](#_Bibliografie) Figura 6. Exemple de servicii [[6]](#_Bibliografie)

Deși această abordare poate părea ca un simplu schimb de responsabilități ea conduce la un serviciu mai usor de testat având mai puține dependințe externe și la centralizarea logicii domeniului nostru în limitele modelului, centralizare ce oferă o complexitate mai mică și o înțelegere mai ușoară a codului sursă.

În final, un serviciu de domeniu se află la limita dintre impuritatea modelului nostru și complexitatea pe care o deține. Pe de o parte nu putem crea un serviciu complet izolat deoarece lucrează cu logica domeniului nostru însă pe de altă parte nu putem să ii oferim prea multă logică deoarece am ajunge la un model anemic.

### Agregate

Într-un model complex care este dezvoltat dea lungul timpului, dependințele dintre entitățile modelului pot scăpa de sub control iar popularea cu date a unor astfel de entități devine o operație foarte costisitoare și greu de manageriat. Această problemă apare din cauza folosirii unui singur model, a unui singur context pentru a prezenta toată problema domeniului. Una din metodele de a tine sub control aceste dependințele este introducerea agregatelor împreună cu împărțirea sistemului în sub-domenii, pentru a fi mai ușor de manageriat.

Agregatele reprezintă o abordare a dezvoltării ce constă în adunarea sub o singură abstracție a mai multor entități și value objects ce se schimbă împreună atunci când sunt modificate. Agregatele se comportă ca o unitate pentru stratul de servicii al aplicației, însemnând că serviciile ar trebui să le preia din baza de date, să le supună unor operații de business și să le salveze înapoi ca un singur obiect. Acesta caută să obțină consistență tranzacțională ce respectă principiile ACID:

* Atomicitate: schimbările stării unei tranzacţii sunt atomice în sensul că acestea sunt salvate complet, sau nu are loc nicio schimbare;
* Consistenţă: o tranzacţie nu conduce către o stare invalida a bazei de date;
* Izolare: obținem aceeași stare a bazei de date chiar dacă tranzacţiile sunt executate în mod concurent sau în mod secvențial;
* Durabilitate: odată ce o tranzacţia a fost executată cu succes schimbările pe care ea le-a produs în starea bazei de date nu vor putea fi anulate în nici un fel de o posibila eroare.

Fiecare agregat are:

* Un set de invarianți pe care îi menține de-a lungul existenței sale astfel încât în orice moment acesta se află într-o stare validă;
* o rădăcina agregat: o entitate ce reprezintă clasa părinte pentru restul obiectelor din acel agregat; singura cale de acces la conținutul agregatului o reprezintă această clasă, restul claselor neputând fi direct referențiate pentru a restricționa accesul exterior la obiectele interne ale agregatului.

În ciuda beneficiilor care le aduc agregatele, de foarte multe ori acestea sunt adunate în clustere mari si complexe, ce nu respectă constrângerile business-ului și reprezintă o greșeală dacă așteptările includ folosirea lor de către foarte mulți utilizatori, obținerea unor performanțe bune și flexibilitatea scalabilității.

Pentru a obține așteptările mai sus menționate se recomandă crearea unor agregate independente unele de altele pe cât posibil iar în cazul în care există dependințe intre multiple agregate se poate folosi consistența eventuală. Consistența eventuală afirmă că la un moment dat, când un server suferă o modificare și ajunge să aibă o versiunea nouă a unei informații față de celelalte servere, un client poate avea versiunea nouă a informației iar altul versiunea veche. Odată făcută o cerere, nu se mai consultă și celelalte servere deoarece ar spori timpul de așteptare. Această practică este folosită în sistemele distribuite pentru că, în final, toate serverele să aibă aceeași versiune. În practică, această situație reprezintă o ocazie foarte bună de a introduce un serviciu de domeniu, ce comunică mai departe operația care a avut loc.

Există două metode de a crea un agregat:

* folosim două clase: o interfață separată și o clasă care o implementează; această abordare ascunde detaliile implementării însă relevă o problemă:
  + ubiquitous language nu este prezent în această abordare deoarece trebuiesc diferențiate prin nume aceste clase și trebuie menținută această abordare pentru consistență;
  + nu exista un motiv concret de a crea o interfață: de cele mai multe ori vom avea o singură clasă pentru un astfel de obiect; necesitatea unei interfețe provine din nevoia de a avea mai multe implementări.
* folosim obiecte de domeniu susținute de obiecte de stare: avem obiecte bogate, cu comportament și obiectele de stare, ce reprezintă starea acelor obiecte bogate; astfel evităm problemele metodei anterioare și obținem:
  + o complexitate mică;
  + o separare a responsabilităților mai bună;
  + o atenție sporită asupra domeniului nostru;
  + comunicarea cu baza de date este minimă.

În final scopul este ca logica business-ului nostru să fie cât mai separată de cea a lucrului cu baza de date iar comunicarea cu aceasta să fie atent definită și bine controlată.

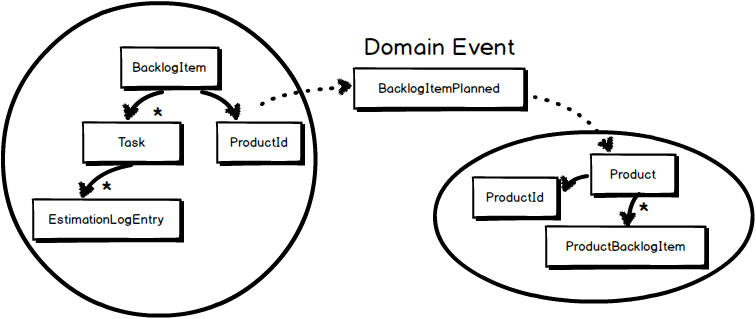


Figura 7: Agregate [[7]](#_Bibliografie)

### Repositories

Repository este un model pentru a încapsula toata comunicarea aplicației cu baza de date, scopul urmărit fiind ușurința programatorului cu care poate modifica informațiile dorite prin apelarea unui singur punct de acces fără niciun alt efort.

Acest respository poate utiliza 2 metode pentru a comunica efectiv cu baza de date:

* un ORM(object relational mapper) – constă în reprezentarea înregistrărilor ca obiecte pentru a obține o relație naturală și transparentă între entitățile modelului și baza de date
* comunicarea directă cu baza de date folosind cereri SQL sau apelând proceduri din cadrul bazei de date pentru obținerea datelor dorite

Există multiple bune practici pentru folosirea acestui model iar acestea sunt:

* un repository trebuie să corespundă unui agregat
  + separarea responsabilităților;
* accesul la repository este valabil doar rădăcinii agregatului corespunzător
  + scade complexitatea;
  + sporește securitatea și impune logica business-ului;
* un repository întoarce întotdeauna o instanță a agregatului corespunzător
  + avem un răspuns constant indiferent de natura cererii;
  + pentru obținerea unor date interne agregatului se pot crea metode pentru a extrage agregate cu informația dorită dar nu informația singulară.

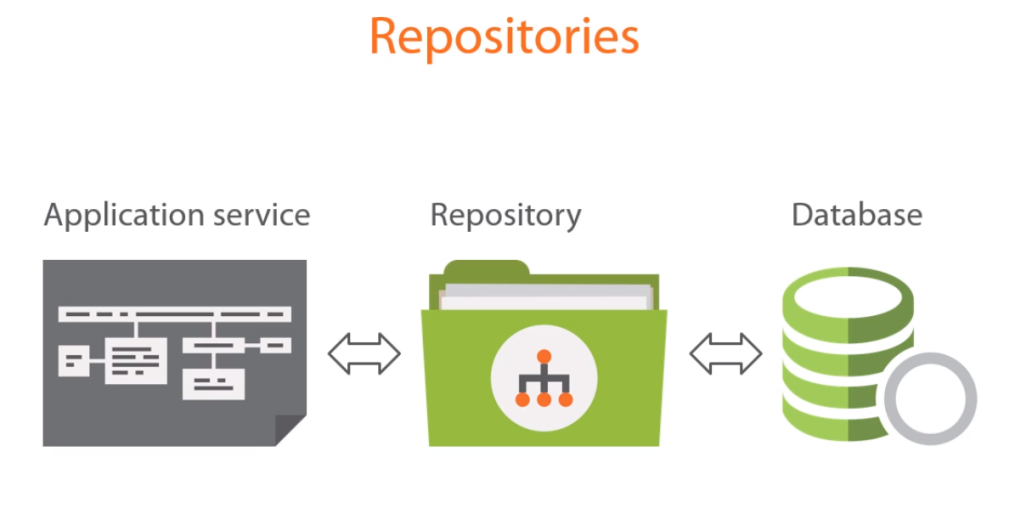


Figura 8. Vizualizare Repository [[8]](#_Bibliografie)

Un alt model ce este folosit împreună cu repository este unitatea de lucru. Unitatea de lucru se referă la faptul că orice cerere externă ar trebui tratată ca fiind o tranzacție: în caz de succes se salvează schimbările iar în caz de eroare nu se vor produce modificări în baza de date. Folosite împreună aceste modele arhitecturale încearcă să creeze un strat de abstracție între stratul de date și cel al logicii aplicației și să ofere un mediu de testare rapid și eficient.

O unitate de lucru va coincide cu durata de viață a unei cereri externe: doar la finalul acesteia, unitatea de lucru va salva schimbările folosind modelul repository pentru contactul cu baza de date. Orice eroare în timpul procesării nu va fi salvată în baza de date iar programatorul care va lucra să adauge funcționalități nu va trebuie să salveze explicit schimbările în cazul în care totul a decurs normal.



Figura 9. Unitatea de lucru [[9]](#_Bibliografie)

### Concluzii

#### Avantaje Domain Driven Design:

* **Ușurința comunicării**: o stabilire clară și de la început a unui limbaj comun legat de modelul nostru, echipele de dezvoltare vor descoperi procesul dezvoltării ca fiind mai ușor deoarece DDD necesită mai puțin jargon tehnic, ubiquitous language folosind termeni simpli pentru a defini aspecte mai tehnice
* **Flexibilitate ridicată**: cum DDD este bazat pe conceptele din programarea orientată obiect arhitectura va obține calitatea de modularitate și de încapsulare; acest fapt permite adăugarea/eliminarea/modificarea modulelor periodic, fără costuri majore
* **Acuratețea implementării**: comunicarea cu experții conduce la un model care reflectă foarte bine conceptele din realitate, astfel că se creează un produs care rezonează cu audiența asociată acelui domeniu
* **Microservicii:** motivul pentru popularitatea microserviciilor îl reprezintă scalabilitatea și flexibilitatea, calități la care DDD excelează prin separarea responsabilităților și prin comunicarea eficientă între modulele/serviciile unui sistem
* **Istoria succesului**: are deja o istorie de proiecte complexe ce au avut succes în trecut dovedindu-și astfel valoarea ca metodă de dezvoltare pentru produse software de mari dimensiuni
* **Calitatea codului**: obținem un cod sursă de calitate fără duplicate și care să poată fi ușor testat și care totodată exprimă mai multă logică cu mai puțin cod, fapt ce conduce la o complexitate mică
* **O înțelegere mai ușoară**: o complexitate mică duce la o înțelegere mai ușoară a programatorilor asupra proiectului dar și a nevoilor clientului

#### Dezavantaje Domain Driven Design

* **Necesită prezența activă a unui expert din domeniu**: crearea unui model reprezentativ domeniului nostru necesită un expert care să cunoască foarte bine domeniului pentru a crea un model ce poate rezolva într-un mod eficient și realist problemele propuse
* **Reprezintă un cost adițional** pentru echipa de dezvoltare deoarece:
  + Constituie un alt mod de a dezvolta un produs software ce implică multe cunoștințe noi pe care le definește dar și elemente din alte arii: programarea orientată obiect, protocoale, baze de date și bune practici aplicate în industrie
  + trebuie să comunice intens cu expertul domeniului
* **Este potrivit pentru proiecte mari**, **complexe** și este cel mai profitabil când este aplicat pe termen lung

## ORM

Un ORM(object relational mapping) reprezintă o tehnică, un mecanism prin care este posibilă manipularea unor obiecte fără a considera modul lor de stocare în baza de date. Acest mecanism oferă echipei de dezvoltare o perspectivă constantă asupra obiectelor chiar dacă mecanismul de persistenta este modificat, el mapând colecțiile de obiecte ale sistemului nostru cu tabelele corespunzătoare lor. Scopul acestuia este de a creste ritmul dezvoltării și de a permite un design extensibil.

Un ORM reprezintă o unealtă de separare a logicii față de baza de date, astfel încât experții pe baze de date lucrează cu ORM iar restul echipei de dezvoltare se poate concentra pe logica aplicației. Este un caz ce relevă autonomia echipelor de dezvoltare.

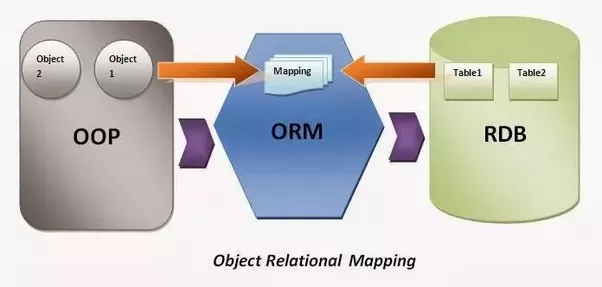


Figura 10. ORM [[10]](#_Bibliografie)

## Arhitectura Onion

Această noțiune a fost pentru prima oară introdusă în 2008 de către Jeffry Palermo într-o serie de bloguri și reprezintă o arhitectură ce are menirea de a rezolva problema arhitecturilor clasice: cuplarea strânsă **ș**isepararea responsabilităților. O privire de ansamblu asupra acestui concept este prezentată în figura 11.

Numele provine din existența mai multor straturi concentrice ce are în centru contextul principal și pornește spre exterior cu celelalte straturi care se bazează pe el. Regula după care se ghidează această arhitectură o reprezintă faptul nu ar trebui să avem dependințe dinspre interior spre exterior, ci doar straturile superioare să depindă de cele interioare; această practică vrea să protejeze soluția problemei domeniului nostru iar orice modul să poată fi modificat sau înlocuit fără să aducă modificări sistemelor „interne”.

Cuplarea slabă dintre straturile arhitecturii este obținută prin dependințe unidirecționale(din exterior spre interior) iar separarea responsabilităților este obținută prin stabilirea clară a scopului fiecărui strat, stratul din mijloc având majoritatea logicii business-ului în el, acesta fiind cel mai valoros deoarece reprezintă conceptele domeniului nostru cu care încercăm să rezolvăm problema acelui domeniu.

Figura 12 pune în evidență conceptele DDD ce se află în stratul central al aplicației și le corelează cu principiile arhitecturii onion: fiecare nivel are o anumită responsabilitate ce nu trebuie încălcată, pentru a separa cât mai clar modulele ce pot fi schimbate de-a lungul timpului de codul sursă ce reprezintă logica business-ului nostru.

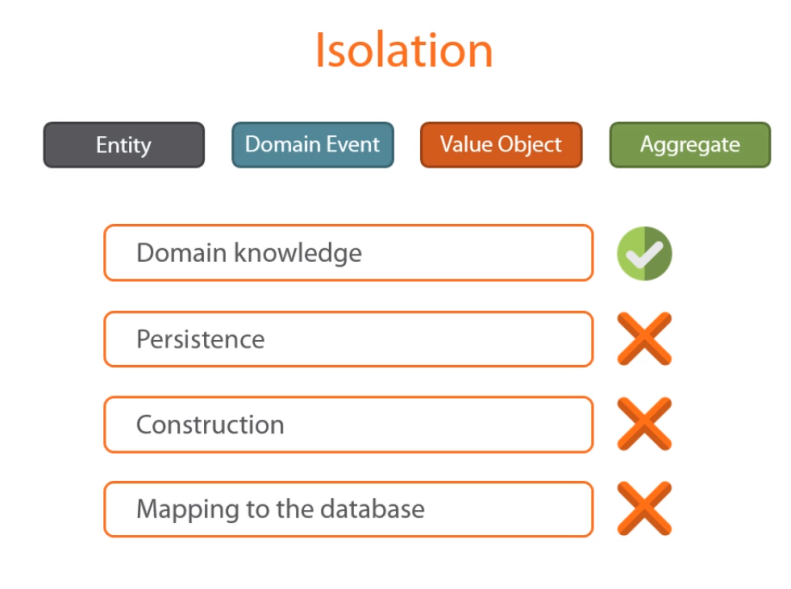
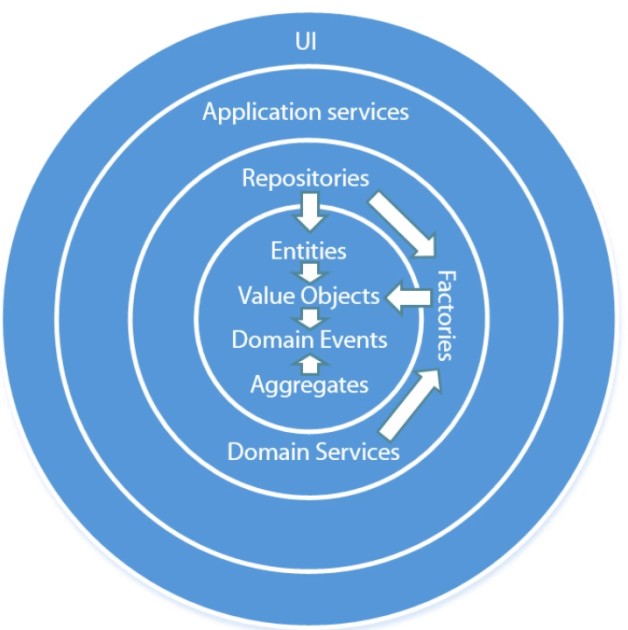


Figura 11. Arhitectura Onion Prezentare [[11]](#_Bibliografie) Figura 12. Izolarea stratului central [[12]](#_Bibliografie)

### Concluzii

Această arhitectură stabilește o ordine clară a importanței straturilor și a relațiilor dintre ele, oferind încă un suport teoretic pentru separarea responsabilităților și a modului în care sistemul nostru ar trebui structurat conceptual pentru a beneficia de scalabilitate și eficiență în dezvoltare.

## MVC

MVC(Model-View-Controller) este un model arhitectural folosit în dezvoltarea software. Scopul acestuia este de a separa logica de business de interfața pentru utilizator, rezultând un sistem unde putem aduce modificări unui nivel fără a le afecta pe celelalte. Acest pattern este format din 3 componente:

* Model – partea din aplicație care se ocupă cu logica datelor și de datele propriu-zise
* View – partea grafică a aplicației, cea care oferă o formă utilă datelor noastră; este totodată legătura cu un utilizator obișnuit care vrea sa comunice cu sistemul nostru
* Controller – partea ce orchestrează cererile primite de la clienți și oferă un răspuns din partea sistemului, rolul lui fiind de intermediar între primele 2 componente

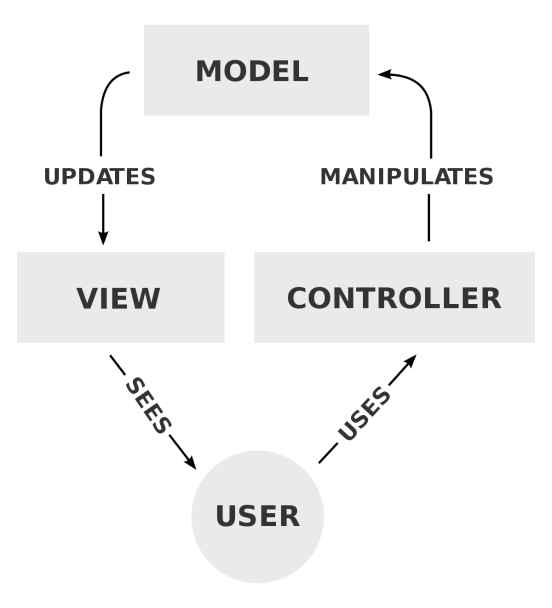


Figura 13: MVC [[13]](#_Bibliografie)

## Microservicii

Arhitectura bazată pe microservicii constă în dezvoltarea unei singure aplicații ca o suită de servicii de mici dimensiuni, fiecare având rolul său, dar care comunică prin mecanisme simple, precum protocolul HTTP.

Această arhitectură vine ca alternativă a arhitecturii monolitice care conducea la aplicații mari, complexe, unitare cu foarte multe dependințe. În contextul în care tehnologiile erau relativ constante la acea vreme, acest lucru era acceptat pentru că nu impunea foarte multe provocări, însă odată cu revoluția internetului acest lucru s-a schimbat. Nevoia permanentă de a se adapta la cerințele vremii, la ultimele tendințe, a condus în final la acest tip de arhitectură: împărțită în mai multe module independente una de alta, astfel încât dacă se dorește o schimbare putem modifica unde avem nevoie fără a produce repercusiuni în întregul sistem.

Popularitatea acestei arhitecturi în ultimii ani este datorată mai multor motive:

* Echipe autonome pot lucra independent unele de altele
* Pune în lumină dezvoltarea haotică a aplicațiilor mari și greutatea cu care acestea sunt dezvoltate
* Oferă o nouă perspectivă asupra modului în care se pot dezvolta aplicații indiferent de natura și dimensiunea lor

Separarea și împărțirea aplicației în module independente care totuși pot comunica unele cu altele se bazează pe concepte din domain driven design și anume soluții separate (contexte delimitate) pentru sub-probleme ale aplicației (sub-domenii), ce pot comunica folosind o metodă cunoscută ambelor părți (ubiquitous language) fără a introduce dependințe sau efecte nedorite (stratul anti-corupție).

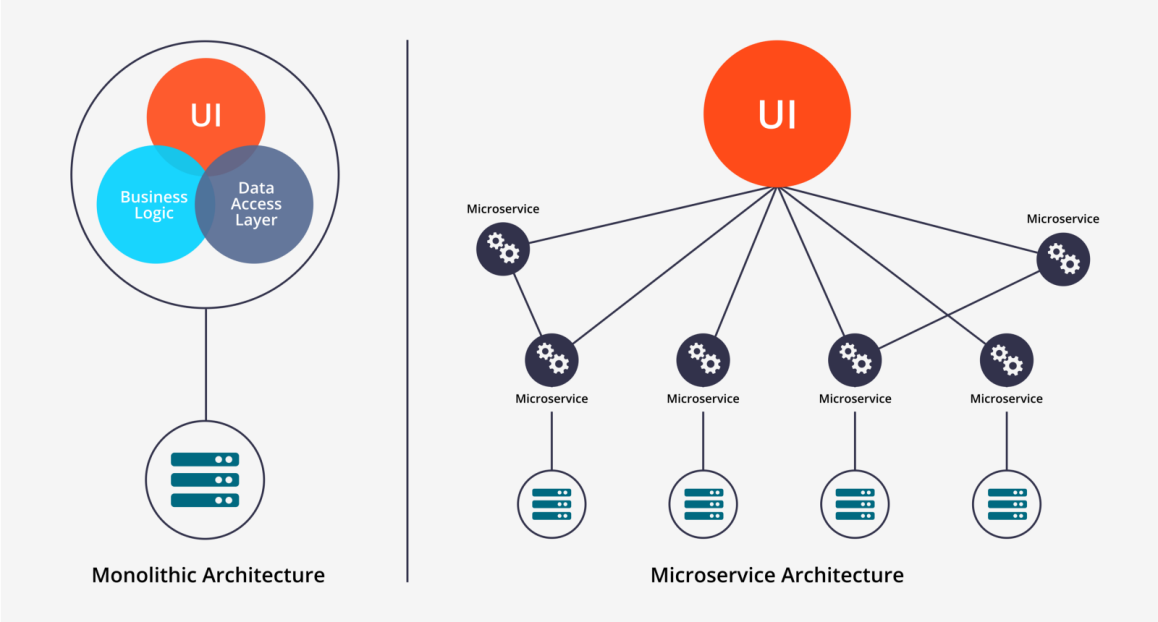
****

Figura 14: Microservicii [[14]](#_Bibliografie)

**Concluzii**

În concluzie microserviciile care au condus spre dezvoltarea calcului în cloud, una dintre cele mai puternic dezvoltate ramuri ale IT-ului în ultimii ani, se bazează pe teoria domain driven design, ea fiind enunțată pentru prima oara în 2004.

# ASPECTE PRACTICE

Partea practica a acestei lucrări constă în crearea unei aplicații folosind metoda tradițională de dezvoltare și anume aplicarea unui model anemic comparată cu un model bogat. Arhitectura va urmări modelul arhitectural MVC.

Aplicația folosită în a ilustra avantajele și dezavantajele aplicării DDD are ca scop managementul unui hotel și a rezervărilor acestuia din partea clienților. Atenția fiind direcționată spre arhitectură, crearea unui UI(user interface) nu ar fi adus nicio valoare lucrării drept concluzie fiind faptul ca exemplele vor fi vizualizate sub forma unor secvențe de cod și o discuție peste fiecare porțiune de cod care sa pună în evidență avantajele și dezavantajele fiecarei abordări.

## Tehnologii

Ca tehnologii pentru partea de server am utilizat:

* .NET Core 2.0 – Framework creat de Microsoft ce facilitează utilizarea MVC dar care totodată se bucură de un mediu open-source fiind deschis la extensii în funcție de nevoia business-ului; pentru mai multe detalii a se vizita <https://docs.microsoft.com/en-us/dotnet/core/>
* NHibernate – un ORM puternic și versatil pentru a face față multor schimbări în decursul dezvoltării; pentru mai multe detalii a se vizita <http://nhibernate.info/>
* SQL Sever – sistem de gestionare de baze de date relaționale creat de Microsoft; pentru mai multe detalii a se vizita <https://docs.microsoft.com/en-us/sql/sql-server/sql-server-technical-documentation?view=sql-server-2017>

Anemic Model VS Rich Domain Model

### Controller

|  |
| --- |
| public IActionResult Create([FromBody] Customer customer)  {  try  {  if (!ModelState.IsValid)  {  return BadRequest();  }  if (\_customerRepository.GetByEmail(customer.Email) != null)  {  return BadRequest("Email is already in use: " + customer.Email);  }  customer.Status = CustomerStatus.**Regular**;  \_customerRepository.Add(customer);  \_customerRepository.SaveChanges();  return StatusCode(201);  }  catch (Exception e)  {  return StatusCode(500, new {error = e.Message});  }  } |

La nivel de controller regăsim mai multe probleme printre care enumerăm:

* Cuplarea domeniului nostru cu un contract de date(o decizie de comun acord cu privire la modul de transmisie a datelor intre server și client) conduce la imposibilitatea refactorizării deoarece ar produce erori clientului odată cu evoluția domeniului nostru și totodată produce o breșă de securitatea deoarece clientul poate introduce valori invalide iar noi să le acceptăm: de exemplu să modifice cererea ca noul client sa fi cheltuit deja 10000 LEI .
  + Soluția: obiecte de transfer de date. Aceste obiecte fac parte din stratul de anti-corupție al aplicației, pentru a oferi un mediu stabil și sigur dezvoltării fără a pune în pericol comunicarea cu mediul extern; cu alte cuvinte, nu expunem structura claselor noastre lumii exterioare, ci oferim o altă clasă care va rămâne stabilă de-a lungul timpului, schimbările în sistem fiind făcute fără a o modifica
* Managementul erorilor nu se face într-un mod unic deoarece nu avem o metodă concretă de a trata răspunsul domeniului nostru și deci nu este clar din punct de vedere al business-ului de ce s-a produs eroarea
  + Soluția: Încapsularea răspunsului domeniului nostru într-o singură structură, indiferent de natura răspunsului, reprezintă un mod expresiv și controlat pentru a rezolva această problemă
* Rezistența slabă la erori: nu impunem regulile de business ci lăsăm la libera alegere a programatorului dacă respectă sau nu această logica. De exemplu, logica aplicației ne spune că un nou client poate avea doar statutul de „REGULAR” și poate avansa doar prin metode bine controlate și atent alese; un programator ar putea introduce o eroare si ar putea adăuga un client nou cu statutul “VIP” deși acesta nu a făcut nicio operație pentru a obține acest privilegiu
  + Soluția: Introducerea agregatelor și a value objects pentru a obține o încapsulare mai mare si mai puține căi de acces ce vor fi bine definite cu acțiuni limitate pentru a putea impune logica business-ului
* Nu este responsabilitatea repository-ului de a salva schimbările. Unitatea de lucru este cea care decide acțiunea fiind una atomică: fie salvează toate schimbările, fie nu salvează nimic.
  + Soluția: Crearea unui controller de bază ca atunci când se trimite un răspuns pozitiv înapoi, se vor salva schimbările automat astfel noi tratând o cerere externă ca fiind o acțiune atomică: dacă totul se află în parametrii normali salvăm schimbările altfel nu salvăm nimic; în acest mod arhitectura aplicației poate impune aceste reguli și riscul ca un programator să facă o astfel de greșeală scade
* Nu avem un răspuns corespunzător fata de cererea clientului.
  + Soluția: Adoptarea standardului HTTP constituie în acest caz o altă ramură a ubiquitous language ce trebuie păstrat la nivel de context

|  |
| --- |
| public IActionResult Create([FromBody] CreateCustomerDto item)  {  Result<CustomerName> customerNameOrError = CustomerName.Create(item.Name);  Result<Email> emailOrError = Email.Create(item.Email);  Result result = Result.Combine(customerNameOrError, emailOrError);  if (result.IsFailure)  return Error(result.Error);  if (\_customerRepository.GetByEmail(emailOrError.Value) != null)  return Error("Email is already in use: " + item.Email);  var customer = new Customer(customerNameOrError.Value, emailOrError.Value);  \_customerRepository.Add(customer);  Return StatusCode(201);  } |

### Entitați

|  |
| --- |
| public class Customer : Entity  {  [Required]  [MaxLength(100, ErrorMessage = "Name is too long")]  public virtual string Name { get; set; }  [Required]  [RegularExpression(@"^(.+)@(.+)$", ErrorMessage = "Email is invalid")]   * + public virtual string Email { get; set; }   [JsonConverter(typeof(StringEnumConverter))]  public virtual CustomerStatus Status { get; set; }  } |

La nivel de entități descoperim următoarele probleme:

* Lipsa încapsulării – nimic nu ne oprește în a altera clasa curenta și a ajunge într-o stare invalidă, de a încălca invarianții obiectului, invarianți definiți de logica business-ului nostru. De exemplu, în situația curentă numele și email-ului utilizatorului pot fi confundate unul cu celalalt, fapt ce ar conduce la o stare ilegală a obiectului, stare ilegală nedetectată de sistemul nostru. Cauza acestei erori este reprezentarea a 2 concepte diferite în același mod în sistemul nostru.
  + Soluția: Reprezentarea conceptelor diferite prin tipuri diferite folosind value objects conduce la un model mai expresiv dar și la o detecție mai rapidă a erorilor la compilare deoarece având tipuri diferite nu mai putem confunda numele și email-ul unui utilizator; expresivitatea constă în faptul că reprezentăm diferit concepte diferite și obținem o imagine mai clară a operațiilor permise peste acele date
* Nu este bine definit scopul entității. Este o clasă fără comportament, conține doar date: este un client, dar ce poate face un client în domeniul nostru? Ce ii este permis, respectiv ce ii este interzis să facă? Ce invarianți are? Aceasta abordare crește probabilitatea duplicării codului, deoarece se pot duplica acțiunile la care este supusă această clasă, acțiuni posibil definite în alt loc, fapt ce va conduce la o mentenanță mai costisitoare.
  + Soluția: Un model bogat ce constă în aducerea datelor și a operațiilor care lucrează pe ele în aceeași structura. Această abordare reprezintă esența refactorizării unei aplicații pentru a permite un comportament flexibil dar sigur ar dezvoltării

|  |
| --- |
| public class Customer : Entity  {  public CustomerName Name;  public Email Email;  public virtual CustomerStatus Status { get; protected set; }  public Customer(CustomerName name, Email email)  {  Name = name ?? throw new ArgumentNullException(nameof(name));  Email = email ?? throw new ArgumentNullException(nameof(email));  Status = CustomerStatus.Regular;  }  public bool AddBooking(Booking booking)[...]  public bool ConfirmBooking(Booking booking)[...]  public virtual void Promote()[...]  } |

### Mecanism unic de validare

|  |
| --- |
| public class CreateCustomerDto  {  [Required]  [MaxLength(100, ErrorMessage = "Name is too long")]  public virtual string Name { get; set; }  } |

|  |
| --- |
| public class UpdateCustomerDto  {  [Required]  [MaxLength(100, ErrorMessage = "Name is too long")]  public string Name { get; set; }  [Required]  [RegularExpression(@"^(.+)@(.+)$", ErrorMessage = "Email is invalid")]  public string Email { get; set; }  } |

Introducerea unui strat de compatibilitate folosind obiecte de transfer de date este o decizie bună însă această abordare nu rezolvă toate problemele:

* clientul care ne apelează API-ul nu poate introduce date invalide dar aceste validări sunt realizate doar în acest loc astfel încât programatorul încă are posibilitatea de a crea stări invalide a obiectelor deoarece validările sistemului nostru sunt făcute doar în momentul sosirii unei cereri externe
* Logica domeniului nostru este duplicată, deci mai greu de menținut și de dezvoltat
  + Soluția: Încapsularea totală a obiectelor domeniului nostru și oferirea unei metode unice de obținere a acelui obiect, metodă ce conține toate validările domeniului nostru pentru a nu încălca invarianții obiectelor și deci asigurarea faptului că nu se va ajunge într-o stare invalidă

|  |
| --- |
| public class Email : ValueObject<Email>  {  public string Value { get; }  private Email(string value)  {  Value = value;  }  public static Result<Email> Create(string email)  {  email = (email ?? string.Empty).Trim();  if (email.Length == 0)  return Result.Fail<Email>("Email should not be empty");  if (!Regex.IsMatch(email, @"^(.+)@(.+)$"))  return Result.Fail<Email>("Email is invalid");  return Result.Ok(new Email(email));  }  } |

### ORM

Un ORM ne poate ajuta să creștem viteza dezvoltării deoarece dezvoltatorii se bazează pe acesta pentru a face legătura între structurile aplicației(entitățile) și baza de date. Din această perspectivă introducerea unor tipuri de date create de dezvoltatori aduce probleme de compatibilitate odată cu evoluția aplicației deoarece ORM-ul ar pierde maparea acestor elemente.

Soluția la această problemă o reprezintă folosirea unui câmp primitiv privat și a unui proxy pentru a-l accesa în clasele sistemului nostru. Această abordare ne asigură ca metodele de acces ale clasei nu se schimbă însă comportamentul intern da, fără însă a aduce schimbări cunoscute extern. Pentru o utilizare mai facilă se recomandă folosirea unor operatori de conversie:

* Implicit: un obiect CustomerName va putea fi întotdeauna tradus într-un string
* Explicit: nu orice string poate reprezenta un nume de client

Se observă o paralelă intre obiectele de transfer de date și această abordare deoarece nu conectăm implementarea sistemului nostru cu lumea exterioară și nici cu baza de date: tot timpul se preferă un strat intermediar pentru 2 sau mai multe straturi care au cu totul alt scop.

|  |
| --- |
| public class Customer : Entity  {  private string \_name;  public virtual CustomerName Name  {  get => (CustomerName)\_name;  set => \_name = value;  }  private readonly string \_email;  public virtual Email Email => (Email)\_email;  public virtual CustomerStatus Status { get; protected set; }  public Customer(CustomerName name, Email email)  {  \_name = name ?? throw new ArgumentNullException(nameof(name));  \_email = email ?? throw new ArgumentNullException(nameof(email));  MoneySpent = Dollars.Of(0);  Status = CustomerStatus.Regular;  //other bussiness rules  }  } |

### Eliminarea logicii din servicii

|  |
| --- |
| public class BookingService  {  public Booking ConfirmBooking(Customer customer, Booking booking)  {  decimal price = CalculatePrice(booking);  booking.Price = price;  booking.IsConfirmed = true;  customer.MoneySpent += price;  return booking;  }  private decimal CalculatePrice(Booking booking)[…]  public void AddBooking(Customer customer, Booking booking)[…]  public bool PromoteCustomer(Customer customer)[…]  } |

Un serviciu are scopul de a realiza o acțiune, o activitate ce nu își are locul natural în domeniul nostru. Totodată, un număr mare de servicii cu multe atribuții de obicei indică un model anemic unde logica business-ului este separată de datele pe care operează. Totuși, exista situații când este nevoie de asemenea servicii și anume când trebuie să ne conectăm la un serviciu extern și să aplicăm o logică peste acele date.

* În situația curentă avem un serviciu ce conține logica domeniului nostru și astfel există riscul de a ajunge în stări ilegale și de a duplica codul. De exemplu, dacă în timpul dezvoltării un programator ar uita să confirme rezervarea? Nimic din codul nostru nu ne spune că este o operație necesară așa ca se pot produce greșeli. Astfel am ajunge în poziția în care clientul a plătit vacanța însă sistemul nostru nu a confirmat-o. În sisteme mari, cu mulți clienți, o greșeală aflată în producție conduce la daune financiare considerabile.
* Soluția: aducerea împreună a operațiilor si a datelor împreună, sub o singură acțiune atomică, pentru ca programatorul să aibă o privire de ansamblu asupra logicii operațiilor noastre și totodată să obținem mai mult control asupra a ceea ce poate să modifice/greșească un posibil dezvoltator.

|  |
| --- |
| public class Customer : Entity  {  private string \_name;  public virtual CustomerName Name[…]  private readonly string \_email;  public virtual Email Email => (Email)\_email;  public virtual CustomerStatus Status { get; protected set; }  public Customer(CustomerName name, Email email)[…]  public virtual Result CanPromote()  {  if (Status.IsAdvanced)  return Result.Fail("The customer already has the Advanced status");  if (PurchasedBookings.Where(x => x.PurchaseDate > DateTime.UtcNow.AddYears(-1)).Sum(x => x.Price) < 100m)  return Result.Fail("The customer has to have at least 100 dollars spent during the last year");  //other bussiness rules and logic  //for ex: the last promotion should be at least 6 months old  return Result.Ok();  }  public virtual void Promote()  {  if (CanPromote().IsFailure)  throw new Exception();  Status = Status.Promote();  }  } |

# ÎMBUNĂTĂȚIRI

### Mecanism unic de tratare a erorilor

Erorile în cadrul unei aplicații pot apărea oriunde iar prinderea și tratarea lor duce la o dezvoltare sigură și mult mai eficientă. În acest caz cel mai adesea se folosesc coduri de stare din protocolul HTTP pentru a informa clientul care folosește aplicația de rezultatul cererii lui. În ciuda tuturor eforturilor întotdeauna vor exista erori netratate care pot provoca costuri majore atât pentru client cât și pentru dezvoltator.

Cel mai adesea apelăm la un bloc try-catch pentru a “prinde” restul tipurilor de erori netratate de sistemul nostru. În clasa controller-ului descoperim o problemă în acest sens: fiecare cerere web trece printr-un astfel de bloc, însa problema constă în duplicarea acestuia.

|  |
| --- |
| public class CustomerController : Controller  {  [HttpGet("{id}")]  public IActionResult Get(int id)  {  try[…]  catch (Exception e)  {  return StatusCode(500, new {error = e.Message});  }  }    [HttpPost]  public IActionResult Create([FromBody] Customer customer)  {  try[…]  }  [HttpPut]  [Route("{id}")]  public IActionResult Update(long id, [FromBody] Customer item)  {  try[…]  }  [HttpPost]  [Route("{id}/promotion")]  public IActionResult PromoteCustomer(long id)  {  try[…]  }  [HttpPost]  [Route("{id}/bookings")]  public IActionResult ConfirmBooking(long id, [FromBody] long bookingId)  {  try[…]  }  } |

O astfel de abordare introduce duplicarea codului și a logicii în cadrul unei singure clase. O alta problemă apare în momentul în care vrem să modificăm tratarea unui singur astfel de bloc: trebuie să fim consecvenți iar nerespectarea acestei bune practice conduce la timp pierdut în momentul unei refactorizări. O soluție este crearea unui mecanism unic de tratarea unor astfel de erori iar .NET Core facilitează această abordare prin libertatea de a adăuga pași intermediari în cadrul procesării unei cereri externe. Odată creată această clasă intermediară care să trateze erorilor nepreluate până în acest punct ea trebuie adăugată în etapele procesării. În final obținem un cod simplificat dar care tratează în mod unic erorile și nu lasă posibilitatea netratării erorilor de alți programatori.

|  |
| --- |
| public class ExceptionHandler  {  private readonly RequestDelegate \_next;  public ExceptionHandler(RequestDelegate next)  {  \_next = next;  }  public async Task Invoke(HttpContext context)  {  try  {  await \_next(context);  }  catch (Exception ex)  {  await HandleExceptionAsync(context, ex);  }  }  private Task HandleExceptionAsync(HttpContext context, Exception exception)  {  string resultJsonConvert.SerializeObject(Envelope.Error(exception.Message));  context.Response.ContentType = "application/json";  context.Response.StatusCode = (int)HttpStatusCode.InternalServerError;  return context.Response.WriteAsync(result);  }  } |

**CONCLUZII**

Domain Driven Design oferă un ghid bine documentat și argumentat pentru a putea dezvolta într-un mod facil un produs software dar nu este foarte popular datorită faptului că este costisitor: toți membrii echipei trebuie să cunoască anumite părți de teorie și să identifice din timp probleme care pot apărea mai târziu pentru a putea implementa o soluție eficientă și elegantă. Un mod practic de a o folosi este a de a crea o echipă mica experimentată care să cunoască aceste principii și să creeze infrastructura aplicației pe care restul programatorilor o vor îmbogăți cu funcționalități. În final DDD urmărește o arhitectura “autonomă” care sa detecteze probleme și probleme de logica, de business pentru a crește viteza și calitatea dezvoltării.

Această practică oferă multe beneficii însă pentru a o stăpâni necesită timp și răbdare, fiind un alt mod de a dezvolta un produs software. În primul rând pune in evidență cunoștințele solide și diverse ale unui programator, fie ca vorbim de programarea orientată obiect, de programarea funcțională, baze de date, protocoale sau bune practici; toate aceste noțiuni se întrepătrund într-o soluție complexă, modulară, unde multiple echipe pot lucra autonom.

Din punctul meu de vedere această metodologie reprezintă baza teoretică peste care multe sisteme complexe (formate din microservicii/module autonome) se bazează iar odată înțeleasă conduce la posibilitatea aplicării ei în mai multe direcții, în mai multe medii de programare. În spațiul public se discută intens despre scalabilitatea și puterea sistemelor informatice din ziua de astăzi dar conceptele care se regăsesc acolo nu sunt evidențiate; deși de multe ori poate părea intimidantă abordarea unor astfel de sisteme, există deja metode bine documentate și aplicate după care să ne ghidăm pentru astfel de provocări.

În final, domain driven design conduce astfel spre crearea unui alt tip de mentalități asupra programării și cum ar trebui abordată: ca fiind soluția unei probleme a unui domeniu care poate nu a avut foarte mult contact cu puterea tehnologiei secolului XXI și ar beneficia major în urma unui sistem personalizat.

**BIBLIOGRAFIE**

* [1] - <https://ict.ken.be/domain-driven-design>
* [2] - <https://martinfowler.com/bliki/BoundedContext.html>
* [4][8][11][12] - <https://app.pluralsight.com/library/courses/domain-driven-design-in-practice>
* [5] - <http://gorodinski.com/blog/2012/04/14/services-in-domain-driven-design-ddd/>
* [6] - <https://app.pluralsight.com/library/courses/domain-driven-design-fundamentals>
* [7] - <https://vaughnvernon.co/?p=879>
* [9] - <https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions/getting-started-with-ef-5-using-mvc-4/implementing-the-repository-and-unit-of-work-patterns-in-an-asp-net-mvc-application>
* [10] - <http://www.nicolaslobbe.com/en/education/Nicola-Slobbe/localdb-in-sql-server-and-use-of-orm>
* [13] - <https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller>
* [14] - <https://hackernoon.com/microservices-are-hard-an-invaluable-guide-to-microservices-2d06bd7bcf5d>
* <https://app.pluralsight.com/library/courses/refactoring-anemic-domain-model>
* <https://enterprisecraftsmanship.com/>
* <https://martinfowler.com/tags/domain%20driven%20design.html>
* <https://airbrake.io/blog/software-design/domain-driven-design>
* „Domain-Driven Design: Tackling Complexity in the Heart of Software” by Eric Evans