Arhitectura sistemelor soft enterprise. Platforma .NET

Curs 4

Prezentarea pe Web. Concurenta

Prezentarea pe Web (1)

- Bibliografie: cap 4 din PoEAA
- Reprezinta punct de cotitura in realizarea aplicatiile enterprise
- Motivatia utilizarii interfetelor utilizator pe Web:
  - Nu este nevoie de drepturi instalare, ~~exceptand cele necesare pentru Flash sau Silverlight~~
  - Interfata utilizator unitara
  - Orice SO mainstream permite instalarea de browser
  - Existenta numeroaselor medii de programare/limbaje/biblioteci/framework-uri care permit crearea de aplicatii Web

Nota: diferenta intre un framework si o biblioteca – Frameworks = Components + Patterns, http://www.inf.ufsc.br/~vilain/framework-thiago/p39-johnson.pdf

Prezentarea pe Web (2)

- Punctul de plecare: aplicatie de tip server web
- De regula, un fisier de configurare care precizeaza ce programe proceseaza anumite URL-uri
- Pe un server pot exista mai multe programe care se adauga in directoare dedicate
- Doua moduri de materializare a aplicatiilor web:
  - script
  - pagina server
- Script: un program care proceseaza apelul HTTP
  - Exemple: scripturi CGI, servlete Java

Prezentarea pe Web

- Script-uri (continuare):
  - Se bazeaza pe examinarea obiectului HTTP Request, care contine un sir de caractere (http://www.google.ro/#hl=ro&source=hp&q=design+patterns&btnG=C%C4%83utare+Google&meta=&aq=f&oq=design+patterns&fp=7512f19741983a9c)
  - Sau: poate lua in considerare ceea ce s-a trimis prin formular ("form") de pe client
  - Segmentarea continutului pe baza de expresii regulate (Perl) sau pe baza de perechi de forma cheie = valoare
  - **Iesirea**: un sir de caractere, reprezentand continutul paginii Web ce va fi afisata in browserul clientului
  - Problema majora: un programator ar trebui sa stie sa scrie HTML, cu tot cu formatare si layout; alternativ, un designer ar trebui sa poata sa programeze

Prezentarea pe Web: exemplu de servlet

```
package hall;


import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;


public class HelloWWW extends HttpServlet
{
 public void doGet(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException {
  response.setContentType("text/html");
  PrintWriter out = response.getWriter();
  out.println("<!DOCTYPE HTML PUBLIC \"-
     //W3C//DTD HTML 4.0 " +

     "Transitional//EN\">\n" +
          "<HTML>\n" +
          "<HEAD><TITLE>Hello
     WWW</TITLE></HEAD>\n" +
          "<BODY>\n" +
          "<H1>Hello WWW</H1>\n" +
          "</BODY></HTML>");
 }
}
```

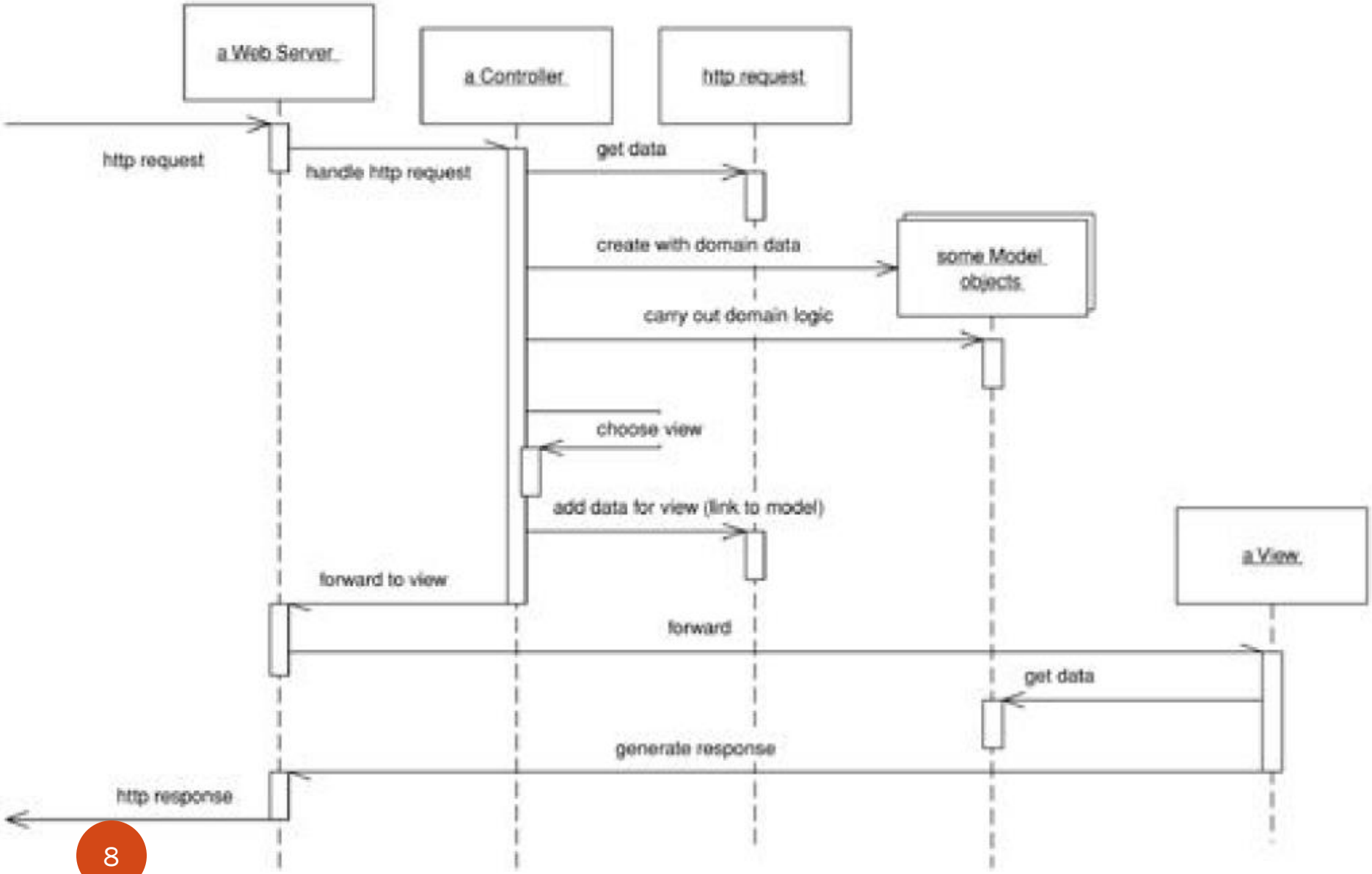
Prezentarea pe Web

- Varianta a 2-a: *pagini* de server
  - Pagini continand majoritar cod HTML, precum si secvente de cod ce se executa de catre o aplicatie pe server
    - Secventele de cod in limbaj de programare = scriptlets
  - Exemplu: PHP, ASP.NET, JSP, JSF,…
  - Se lucreaza usor cu ele daca este putina informatie dinamica de prezentat, sau formatul informatiei dinamice este relativ omogen
  - Daca se doreste realizarea unei prezentari care sa difere in functie de criterii evaluate la runtime, o pagina server devine problematic de mentinut
    - Exemplu: continutul sa fie prezentat diferit in functie de tipul sau (albume muzica rock/clasica/jazz/populara)

Prezentarea pe Web

- Combinatie a celor doua variante
  - un script e bun la interpretare de cerere de la client
  - o pagina server este potrivita pentru prezentarea raspunsului intr-un anumit fel
- Rezultatul combinarii: pattern-ul Model-View-Controller (MVC)
  - View: interfata utilizator din/in care se preia cererea sau care prezinta rezultatul
  - Model: modelarea problemei, reprezentare specifica domeniului; in acest context modelul include si partea de persistare a datelor
  - Controller: poate reactiona la evenimente de pe view/sa comande view/sa interactioneze cu modelul

Prezentarea pe Web



Prezentarea pe Web

- Avantaj al MVC:
  - Modelul (logica de domeniu) este separat de prezentarea web -> modificarea usoara a prezentarii + **testare usoara a modelului**
  - Permite introducerea unui Application Controller (AC) – specifica ce ecrane apar si in ce ordine; modificarile ulterioare (adaugari de ecrane/schimbari de ordine) sunt usor de programat/configurat
  - Modelul de decizie din AC poate fi scris independent de orice tip de prezentare particulara – se poate refolosi pentru GUI-uri realizate cu alte tehnologii
  - AC – e necesar daca alegerea urmatoarei pagini este in sarcina aplicatiei
  - AC – inutil daca alegerea urmatoarei pagini se face de catre utilizator

Pattern-uri legate de View

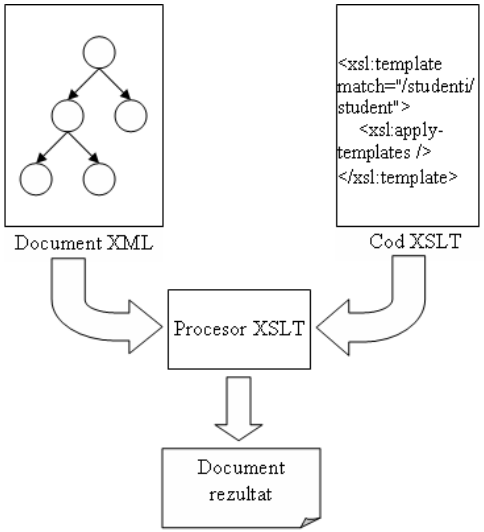
- Pattern-uri legate de View
  - Template View
  - Transform View
  - Two Step View
- Prima decizie: **Template View** vs **Transform View**
- **Template view** permite realizarea paginii (partea vizuala) impreuna cu indicarea locului in care continutul dinamic va fi depus la runtime
- Dezavantaj: cod amestecat, HTML+cod destinat serverului; se poate rezolva partial problema prin utilizarea de obiecte ajutatoare care sa micsoreze cantitatea de cod

Ex. template view (fragment, ASP.NET MVC)

```
@model LoginPart.Models.Person

@{
ViewBag.Title = "Register";
}

<h2>Register</h2>


@using (Html.BeginForm())
{
@Html.AntiForgeryToken()

<div class="form-horizontal">
    <h4>User1</h4>
    <hr />
    @Html.ValidationSummary(true, "", new { @class = "text-danger" })
    <div class="form-group">
        @Html.LabelFor(model => model.userName, htmlAttributes: new { @class = "control-l
        <div class="col-md-10">
            @Html.EditorFor(model => model.userName, new { htmlAttributes = new { @class
            @Html.ValidationMessageFor(model => model.userName, "", new { @class = "text-
        </div>
    </div>

    <div class="form-group">
        @Html.LabelFor(model => model.userEmail, htmlAttributes: new { @class = "control-
        <div class="col-md-10">
            @Html.EditorFor(model => model.userEmail, new { htmlAttributes = new { @class
            @Html.ValidationMessageFor(model => model.userEmail, "", new { @class = "text
        </div>
    </div>
```

Pattern-uri legate de View

- **Transform View** – aplica o transformare asupra unui set de date furnizat de domain logic
- Exemplu clasic: XSLT aplicat unui document XML furnizat de model



- Template View si Transform View se pot mixa, dar de regula oamenii prefera utilizarea consistenta doar a uneia din ele => usurinta in mentenanta

Exemplu XML + XSLT

1: class.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl"
    href="class.xsl"?>
<class>
    <student>Jack</student>
    <student>Harry</student>
    <student>Rebecca</student>
    <teacher>Dr. Smile</teacher>
</class>
```

3: iesire:

```
<html>
<body>
        <p><b>Jack</b></p>
        <p><b>Harry</b></p>
        <p><b>Rebecca</b></p>
        <p><u>Dr. Smile</u></p>
</body>
</html>
```

2: class.xsl:

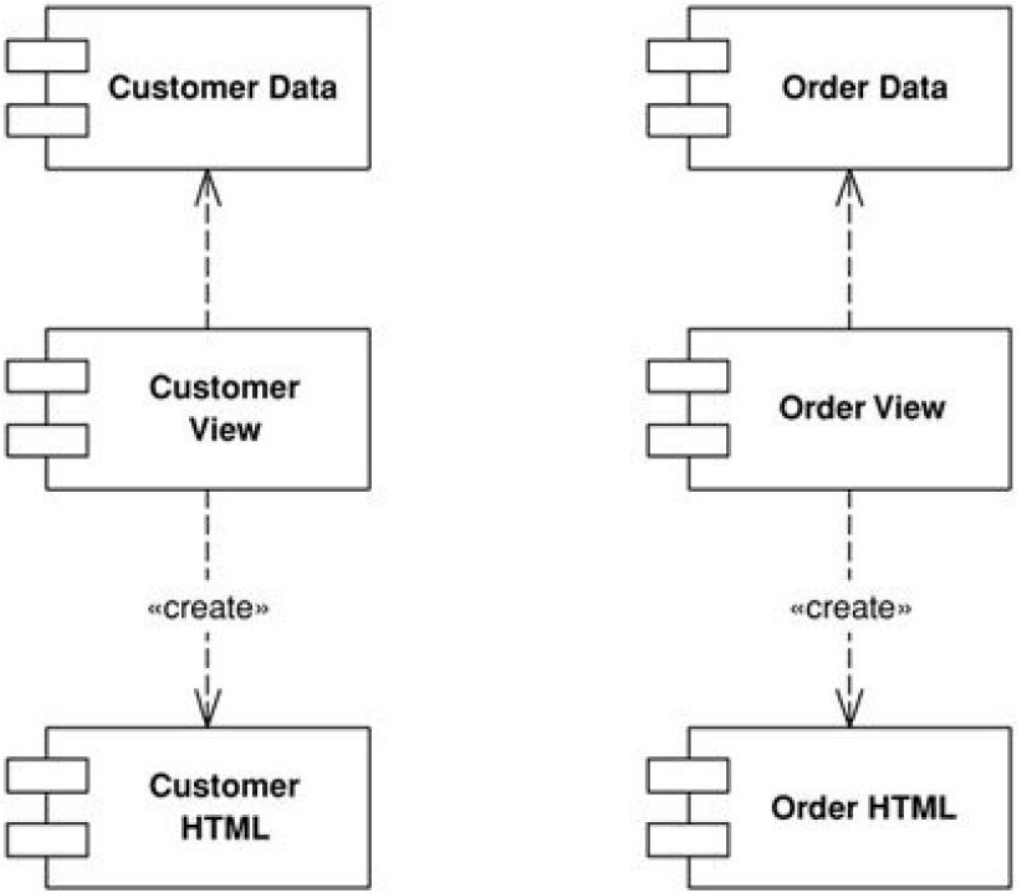
```
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="teacher">
            <p><u><xsl:value-of select="."/></u></p>
    </xsl:template>
    <xsl:template match="student">
            <p><b><xsl:value-of select="."/></b></p>
    </xsl:template>
    <xsl:template match="/">
            <html>
            <body>
            <xsl:apply-templates/>
            </body>
            </html>
    </xsl:template>
</xsl:stylesheet>
```

Exemplu preluat de la: http://www.tizag.com/xmlTutorial/xslttutorial.php

Pattern-uri legate de View

- Urmatoarea decizie: pagina separata pentru fiecare view (eng: Single stage view) sau Two stage view
- Pagina separata/view: cate o pagina pentru fiecare ecran
- Fiecare pagina preia datele din model si le prezinta utilizatorului

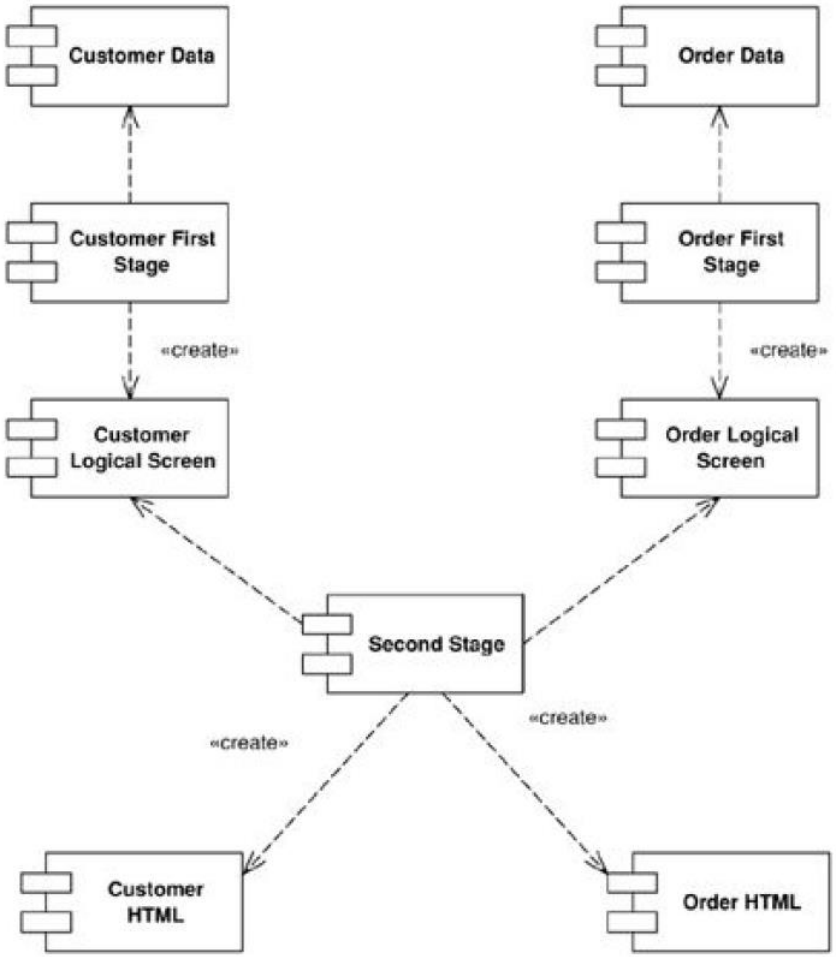
Pattern-uri legate de View



Pattern-uri legate de View

- Two stage view: sparge procesul in doua faze
- In prima faza se produce ecran logic
- In a doua faza se produce HTML

Pattern-uri legate de View



Pattern-uri legate de View

- Two Stage View: avantaj/conditii
  - Decizia legata de care anume HTML se prezinta este luata intr-un singur loc => mentenanta simplificata
  - Utilizabile in cazul in care ecrane diferite partajeaza acelasi layout
  - Se poate folosi acolo unde exista diferite dispozitive pe care se face prezentare (terminale de automate, browsere pe calculatoare, smart phones)

Concurenta

- Bibliografie: cap 5 din PoEAA
- Mai multe procese sau fire de executie care manipuleaza aceleasi date => probleme de concurenta
- Aspect dificil de gestionat, din cauza ca nu se pot enumera toate scenariile posibile
- Greu de testat - tot din motivul prezentat anterior; greu de reprodus bug-urile
- Pentru baze de date: rezolvata de managerul de tranzactii, componenta a serverului de BD
- Dar nu toate accesele concurente sunt pe baza de date! ☹

Concurenta

- Concurenta la accesul la date, pe serverul de baze de date: suport asigurat de serverele de baze de date
- Concurenta **offline**: datele sunt preluate de pe server de catre mai multe fire de executie si procesate in paralel; ce se intampla la update pe serverul de BD, de catre fiecare thread?
- Exemplificare: sistemele de control al surselor (SVN, CVS, Git)
- Cazuri:
  - Modificari pierdute – update peste datele scrise de altcineva
  - Citire de date inconsistente – dupa ce o parte din date sunt citite, ele se modifica; valoarea citita anterior este perimata si asta afecteaza procesul

Concurenta

- Varianta de rezolvare: nu se permite accesul concurent, se serializeaza toate accesele la resurse
  - Dezavantaj: procesare inceata; inacceptabila pentru cazul in care utilizatorii lucreaza lent pe aplicatie
- Procesare concurenta de cereri: prin procese separate sau prin mai multe fire de executie
  - Proces: context de executie cu izolare totala (fiecare proces are memoria proprie alocata), dar consumator de resurse (“heavyweight”) din perspectiva SO;
  - Fire de executie (threads): pot opera mai multe, simultan, in cadrul unui proces; mai usor de gestionat de SO decat procesele (“lightweight”), dar cu partajare de memorie - deci mai multe posibilitati de aparitie a concurentei

Concurenta

- 2 tipuri de tranzactii:
  - De la aplicatie la baza de date = tranzactie sistem
    - Suport din partea serverului de BD
  - De la utilizator catre aplicatie = tranzactie de business
    - De regula exista clase care se pot folosi pentru a defini locul de inceput si de sfarsit al unei tranzactii, dar programatorul trebuie sa se implice mult in scrierea de cod in aceasta directie

Concurenta

- Izolarea si imuabilitatea – 2 solutii pentru aplicatiile enterprise
- Izolarea – detectarea si declararea zonelor in care la un moment dat doar un singur fir de executie poate sa actioneze la un moment dat
- Exemplu: sincronizari in Java

```
public class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment() { c++; }
    public synchronized void decrement() { c--; }
    public synchronized int value() { return c; }
}
```

Concurenta

```
//C#
public class DemoThreading {
    private System.Object locker= new System.Object();
    public void Process()
    {
        lock (locker)
        {
            // Accesare de catre fir de executie a resurselor partajate
        }
    }
}
```

Concurenta

- Lucrul cu obiecte imuabile (eng: “immutable”) – odata ce un astfel de obiect este creat, orice metoda aplicata asupra lui nu ii va modifica starea, ci eventual va produce alt obiect
  - De ce clasele String in Java si .NET sunt create imuabile?
- Imuabilitatea – nu e un panaceu, pentru ca o aplicatia enterprise are ca scop lucrul cu date si *modificarea* acestora
- Insa detectarea corecta a tipurilor/datelor imuabile sau a situatiilor in care se folosete caracterul imuabil al obiectelor relaxeaza conditiile de concurenta

Concurenta

- Cum se mai poate rezolva problema concurentei offline?
- Doua forme de control al concurentei: control optimist si control pesimist
- Blocare optimista – 2 persoane vor sa modifice acelasi fisier, primul care isi comite schimbarile este acceptat, celui de al doilea i se semnaleaza ca datele peste care vrea sa scrie sunt altele decat cele pe care le-a citit – se evita astfel suprascrierea accidentala

Concurenta

- Argument pentru utilizarea blocarii optimiste: blocarea datelor se face doar in faza de salvare a datelor
- Argument contra utilizarii optimiste: trebuie implementat pasul de verificare a versiunii datelor si semnalarea posibilelor neconcordante
- Daca datele nu pot face “auto-merge”, se reia procesul de editare, pe noua versiune
  - auto-merge e deja implementat pe serverele de control al versiunilor
  - uneori insa este nevoie de interventia umana pentru rezolvarea de conflicte (“solve merge conflict”)

Concurenta

- Blocare pesimista: primul care incepe sa opereze pe date blocheaza pe oricine altcineva care vrea sa acceseze datele ulterior
- Blocarea pesimista = prevenirea de la inceput a conflictelor = argument pro
- Argument contra utilizarii pesimiste: se reduce concurenta – poate impiedica chiar si citirea datelor

Concurenta

- Cum alegi intre varianta optimista si pesimista?
  - Frecventa si severitatea conflictelor: infrecvent sau usor de facut unirea versiunilor => blocare optimista
  - Daca rezultatele unui conflict ridica probleme mari de rezolvare => blocare pesimista
- Prevenirea citirilor inconsistente:
  - Se poate utiliza read lock sau write lock
  - Cat timp cineva are un read lock, oricine poate citi acele date, dar nimeni nu poate sa le modifice (scrie/sterge)
  - Cat timp procesul sau thread-ul X are un write lock, nimeni altcineva nu poate citi sau scrie acele date; X este singurul care are dreptul sa citeasca/scrie datele

Concurenta

- Problema legata de tehnicile pesimiste: deadlock
- Cazul in care doua sau mai multe actiuni concurente asteapta una dupa cealalata ca sa elibereze datele blocate
- Asteptarea poate dura la infinit
- Solutii: modul care sa determine blocajele mortale, atunci cand apar; se “omoara” unul din procese pentru ca altele sa poata sa continue
  - Procesul omorat = victima
  - Pentru victima, deadlock-ul este costisitor: necesita reluarea pasilor

Concurenta

- Problema: predictia aparitiei lor este dificila
- Alternativa: pentru fiecare lock se da un timp limita
  - La atingerea timpului limita, detinatorul lock-ului devine victima
- Mai usor de implementat decat detectarea de inter-blocaje
- Alta varianta: evitarea completa a deadlock-urilor: toate lock-urile se achizitioneaza la inceput, fara a mai da posibilitatea de a cere alte lock-uri pe parcurs

Concurenta

- Alta varianta: ordonarea lock-urilor (e.g. pt fisiere: alfabetic dupa numele fisierelor)
  - Daca ai pus lock pe B.txt nu mai poti pune lock pe A.txt; devii automat victima daca vrei lock pe A.txt; ordinea “blochez A, apoi B” este insa acceptata
- Variantele se pot combina: e.g. alocare de la inceput a resurselor + limita de timp impusa
- De regula, se prefera varintele simple, chiar daca prin asta se mareste numarul de victime

Concurenta

Tranzactia:

- Principala unealta pentru managementul concurentei
- Secventa de pasi cu inceput si sfarsit bine delimitate
  - Exemplu: la ATM, tranzactia incepe o data cu introducerea cardului si se termina cand se scoate cardul; intre timp am operatiile: consultarea contului, plati, transfer de bani, preluare de bani
- Sumarizare in acronimul ACID: atomicitate, consistenta, izolare, durabilitate

Concurenta

- Atomicitate: toate instructiunile din tranzactie se indeplinesc in totalitate (commit) sau deloc (rollback)
- Consistenta: resursele trebuie sa fie in stare consistenta, necorupta atat la inceput cat si la sfarsitul tranzactiei
- Izolare: rezultatul unei tranzactii individuale nu trebuie sa fie vizibil altor tranzactii pana cand tranzactia nu se comite
- Durabilitate: orice rezultat al unei tranzactii reusite trebuie sa fie permanent (chiar daca sistemul cedeaza dupa finalizare)

Concurenta

- Resurse tranzactionale: cozi de mesaje, fisiere, imprimante
- Conditie pentru tranzactii: sa fie cat mai scurte, pentru a mari gradul de acces paralel
- Regula: nu ar trebui ca o tranzactie sa “traverseze” mai multe interogari.
- Ideal, o tranzactie ar trebui inceputa la pornirea unei interogari si terminata la sfarsitul ei
- Varianta: sa incepi cat mai tarziu tranzactia; citirile si procesarile care nu pot duce la date inconsistente sa fie facute inainte de deschiderea tranzactiei

Concurenta

- Reducerea izolarii tranzactiilor pentru viteza mai mare de raspuns
- Izolare deplina => tranzactii serializabile; in urma interogarii se pot obtine mai multe raspunsuri si toate sunt considerate corecte
- Scenarii:
  - 2 pachete de clase; pachetul A contine 7 clase, pachetul B contine 5 clase; Martin citeste cate clase sunt in A => 7; este intrerupt din activitate; intre timp, David adauga 2 clase in A si 3 in B; Martin revine si continua contorizarea => in pachetul B vede 8 clase; ca atare, raporteaza 15 clase, ceea ce nu e corect.
  - Prin serializare: Martin ar raporta fie 12 clase (inainte ca David sa fie lasat sa adauge), fie 17 (dupa ce David a facut actualizarile); serializarea nu garanteaza care din cele doua tranzactii se executa prima, dar macar nu da rezultat incorect (“15” de mai sus)

Concurenta – niveluri de izolare

Isolation Level	Dirty Read	Unrepeatable Read	Phantom
Read Uncommitted	Yes	Yes	Yes
Read Committed	No	Yes	Yes
Repeatable Read	No	No	Yes
Serializable	No	No	No

Concurenta

- Pot fi si alte tranzactii decat cele de pe un server de BD
- Business transaction - vezi exemplificarea pentru ATM
- Nu pot fi neaparat incluse intr-o tranzactie sistem din cauza ca pot fi prea lungi
- De preferat sa nu fie tranzactii lungi pentru ca aplicatia nu suporta stres prea mare
- Problema: refactorizarea de la tranzactii lungi la scurte nu e deplin studiata

Concurenta

- Pentru business transactions – cel mai greu de implementat este izolarea – se poate ajunge la inconsistenta a datelor
- Reguli:
  - Bazeaza-te/foloseste tranzactiile de sistem
  - Pe cat posibil: mentine tranzactiile doar pe durata unei interogari la aplicatie
  - Favorizeaza optimistic offline lock
  - Utilizeaza coarse-grained lock – la nivel de grupuri de obiecte ce se folosesc la operatii

Concurenta

- Care este suportul oferit de platformele Java si .NET (Framework, Core) pentru specificare de tranzactii de business? Raspuns la lucian.sasu@yahoo.com