Proiectarea Sistemelor Software Complexe

Curs 13 –Etapele Proiectării Arhitecturii Unui Sistem Software Complex

13.1 Context

Responsabilitatea unui arhitect software nu constă doar din activitatea de proiectare. Un arhitect software trebuie să:

- **Lucreze cu echipa care stabilește cerințele aplicației** – în acest context rolul arhitectului este acela de a ajuta la adunarea cerințelor și înțelegerea nevoile sistemului ca ansamblu și de a asigura faptul că indicatorii de calitate sunt expliciți și bine înțeleși.
- **Lucreze cu toți cei care vor interacționa cu aplicația** – în acest context arhitectului are rolul de pivot, el trebuie să se asigure că toate nevoile celor care vor interacționa cu aplicația sunt înțelese și încorporate în proiect. De exemplu, pe lângă cerințele care se referă la funcționalitate, administratorul de rețea poate să ceară ca aplicația să fie ușor de instalat, configurat și actualizat.
- **Conducă echipa tehnică de proiectare** – definirea arhitecturii unei aplicații este o activitate de proiectare. Arhitectul conduce echipa de proiectare, compusă din proiectanți de sistem și conducători de echipe de programatori, pentru a realiza schița arhitecturii.
- **Lucreze cu managerul de proiect** – arhitectul trebuie să ajute la planificarea proiectului, estimarea și alocarea taskurilor.

În Fig. 11.1 este ilustrat procesul de proiectare a arhitecturii unui sistem software complex. Procesul constă din parcurgerea iterativă a trei etape:

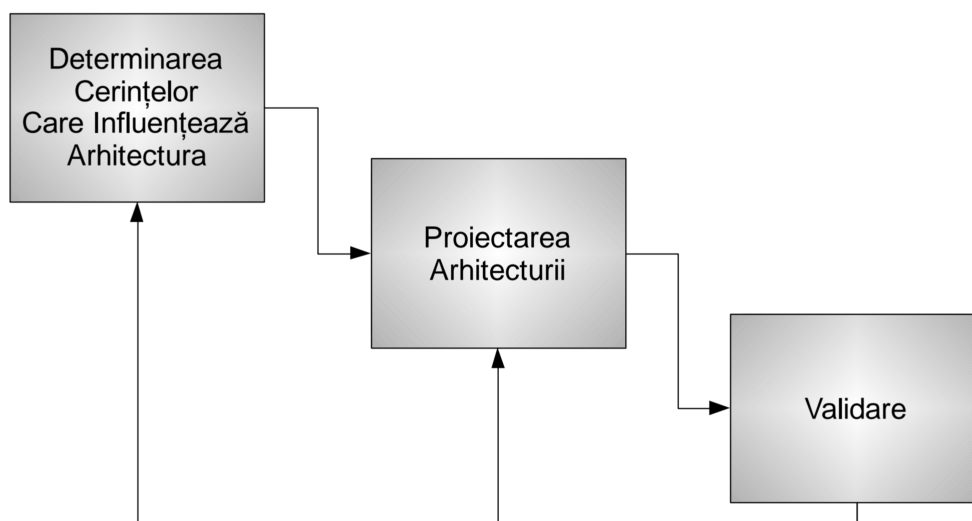

Fig. 11.1.Etapele parcurse în proiectarea unei arhitecturi pentru un sistem software.

- **Determinarea cerințelor care influențează arhitectura** – presupune crearea unui model al cerințelor care va dirija proiectarea arhitecturii.
- **Proiectarea arhitecturii** – definirea structurii și a responsabilităților componentelor care formează arhitectura.
- **Validarea** – testarea arhitecturii;de obicei se realizează parcurgând lista de cerințe și eventualele cerințe ulterioare și verificarea faptului că arhitectura proiectată în etapa anterioară permite implementarea cerințelor.

Etapele enumerate mai sus sunt parcurse iterativ până când se obține o arhitectură validă. În continuare vor fi prezentate în detaliu cele trei etape.

13.2 Determinarea Cerințelor Care Influențează Arhitectura

Înainte de a proiecta arhitectura unui sistem software este important să se obțină o imagine clară asupra cerințelor care influențează arhitectura. De obicei aceste cerințe sunt cerințe non-funcționale care se referă la calitatea sistemului software.

Procesul de identificare a cerințelor care afectează arhitectura are două tipuri de intrări: pe de o parte arhitectul trebuie să analizeze cerințele funcționale, iar pe de altă parte el trebuie să țină cont și de cerințele venite din partea celor care vor interacționa cu aplicația. În urma analizei efectuate asupra celor două tipuri de cerințe rezultă cerințele care influențează arhitectura sistemului software. Procesul de analiză a cerințelor în vederea izolării cerințelor care influențează arhitectura este ilustrat in Fig. 11.2.

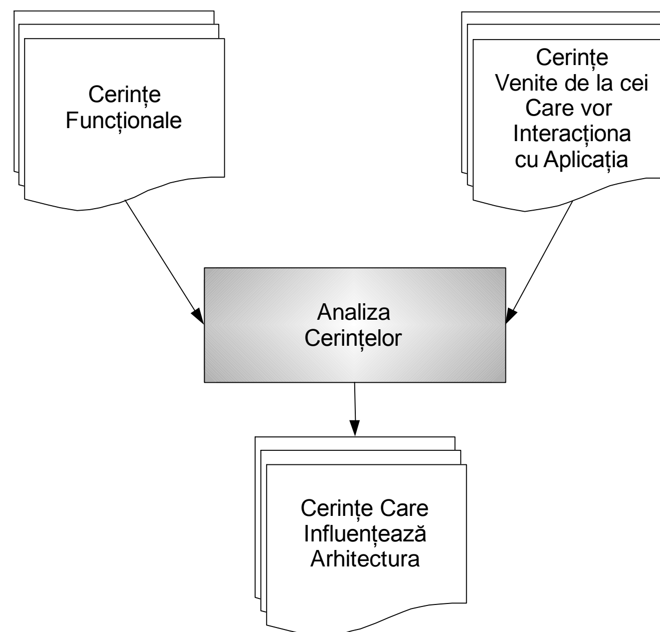

Fig. 11.2. Analiza cerințelor.

Unele cerințe vor acționa ca și constrângeri ele limitând opțiunile arhitectului. În Tabela 11.1 sunt prezentate exemple de cerințe care influențează arhitectura, iar în Tabela 11.2 sunt prezentate exemple de cerințe care impun constrângeri.

Indicator de Calitate	Cerință
Performanță	Aplicația trebuie să aibă un timp de răspuns sub patru secunde pentru 90% din cereri
Securitate	Toate conexiunile trebuie autentificate iar informația transmisă criptată
Fiabilitate	Pierderea mesajelor nu este acceptată
Scalabilitate	Aplicația trebuie să funcționeze corespunzător pentru momente de vârf când sunt conectați simultan un număr de 500 de utilizatori.

Tabela 11.1. Exemple de cerințe care influențează arhitectura.

Constrângere	Cerință
Business	Sistemul trebuie să funcționeze ca și un plug-in pentru MS BizTalk, întrucât se urmărește vânzarea lui către Microsoft
Dezvoltare	Programarea trebuie făcută în Java pentru a se folosi staff-ul de dezvoltare existent
Temporală	Prima versiune trebuie scoasă pe piață în șase luni

Tabela 11.2. Exemple de cerințe care acționează ca si constrângeri.

Un alt aspect care trebuie avut în vedere vis-a-vis de cerințele care influențează arhitectura unui sistem software, este faptul că aceste cerințe nu sunt egale, unele fiind mai importante decât altele. De aceea este necesară o prioritizare a acestor cerințe. De obicei se folosesc trei niveluri de prioritizare:

- **Ridicată** – sistemul software trebuie să implementeze cerințele cu această prioritate. Aceste cerințe au un cuvânt greu de spus în ceea ce privește arhitectura.
- **Medie** – cerințele cu această prioritate vor trebuii implementate la un moment dat, dar nu sunt absolut necesare pentru prima versiune.
- **Scăzută**– funcționalități dorite, dar care se pot implementa în măsura posibilităților.

Prioritizarea cerințelor se complică atunci când apar conflicte între cerințe, de ex.: timp scurt până la scoaterea pe piață a produsului vs. dezvoltarea de componente generice și reutilizabile. De cele mai multe ori rezolvarea acestor conflicte nu este ușoară, dar este sarcina arhitectului să le rezolve.

13.3 Proiectarea Arhitecturii

Reprezintă cea mai importantă acțiune întreprinsă de către arhitect. Un document de cerințe foarte bine structurate, respectiv o comunicare bună cu restul echipelor implicate în proiect nu însemnă nimic dacă se proiectează o arhitectură slabă. Etapa de proiectare a arhitecturii are ca și intrări cerințele obținute în etapa anterioară iar ca rezultat se obține un document care descrie arhitectura sistemului software. Proiectarea arhitecturii se realizează în doi pași: primul pas se referă la alegerea unei strategii globale, iar al doilea constă din specificarea componentelor individuale și a rolului pe care fiecare componentă îl joacă în arhitectura globală.

13.3.1 Alegerea Strategiei Globale

De obicei alegerea strategiei globale se bazează pe un număr restrâns de modele predefinite și bine înțelese. Așadar primul pas din cadrul etapei de proiectare constă în alegerea unui cadru care să satisfacă cerințele cheie ale sistemului. Pentru majoritatea sistemelor alegerea unui singur model predefinit cum este modelul *n-tire client-server*, este suficient, dar pentru sistem complexe se poate să fie nevoie de combinarea mai multor modele. În continuare sunt prezentate principalele model de proiectare folosite.

N-Tire Client-Server

În Fig. 11.3 este ilustrată anatomia unei aplicații web care folosește modelul N-tire client-server. Principalele proprietăți ale acestui model sunt:

- **Separarea responsabilităților** – se realizează o separare clară a nivelului prezentare de nivelul business logic și de nivelul care este responsabil de gestionarea detelor.
- **Comunicarea este sincronă** – comunicare între niveluri (tires) este sincronă de tipul cerere-răspuns. Cererile se transmit într-un singur sens și anume de la clientul web, către serverul web, către serverul de aplicații și în final către serverul de baze de date.
- **Flexibilitate la instalare** – nu există restricții în ceea ce privește modul în care aplicația va fi instalată. Toate nivelurile pot să ruleze pe aceeași mașină sau pe mașini diferite.

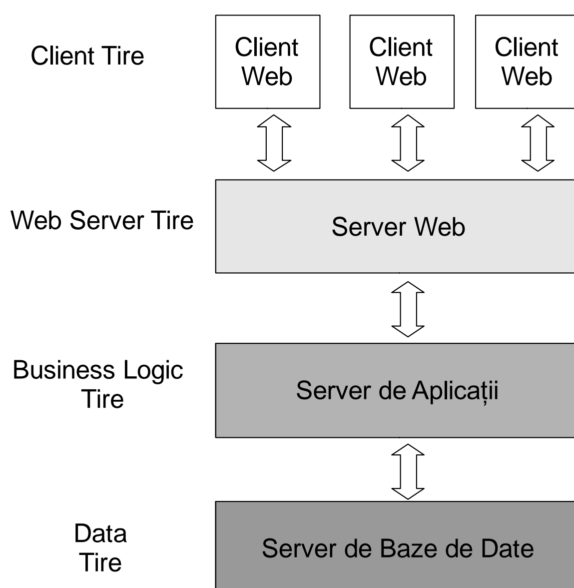

Fig. 11.3.Exemplu de arhitectură N-tire client-server.

În Tabela 11.3 sunt prezentați indicatorii de calitate care pot fi asigurați prin utilizarea modelului N-tire client-server.

Indicator de Calitate	Probleme
Disponibilitatea	Serverele din fiecare nivel pot fi replicate astfel încât dacă unul din ele se defectează celelalte vor rămâne disponibile. Global aplicația va oferi un serviciu de calitate inferioară până când serverul care s-a defectat va fi reparat.
Tratarea erorilor	Dacă un client comunică cu un server care s-a defectat majoritatea serverelor web și a serverelor de aplicații implementează suport pentru transferul transparent către un alt server. Adică, clientul este redirectat fără să își dea seama de acest lucru către o replică a serverului care s-a defectat.
Toleranța la modificări	Separarea responsabilităților îmbunătățește toleranța la modificări. Întrucât nivelurile prezentate, business logic și gestionarea datelor sunt bine încapsulate, se pot face modificări la fiecare nivel fără ca aceste modificări să se propage către celelalte niveluri.
Performanța	Această arhitectură oferă performanțe ridicate. Principalele probleme care

	trebuie avute în vedere din punctul de vedere al performanței sunt: numărul de fire de execuție concurente pentru fiecare server, viteza conexiunilor dintre niveluri și cantitatea de date transferată între acestea. De obicei se urmărește minimizarea numărului de apeluri realizate între niveluri pentru a trata o singură cerere.
Scalabilitatea	Serverul din fiecare nivel poate fi replicat. De obicei acest tip de arhitectură este scalabilă atât în interior cât si în exterior. În realitate însă serverul de baze de date poate să impună anumite limitări.

Tabela 11.3. Indicatorii de calitate controlați printr-o arhitectură de tipul N-tire.

Acest tip de arhitectură se poate folosi atunci când trebuie suport pentru un număr mare de clienți și un număr mare de conexiuni concurente.

Coadă de Mesaje

În Fig.11.4 este prezentat un exemplu de arhitectură care folosește ca și model conceptul de coadă de mesaje. Principalele proprietăți ale unei astfel de arhitecturi sunt:

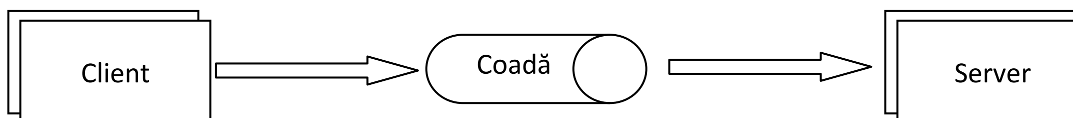

Fig. 11.4. Exemplu de arhitectură bazată pe modelul coadă de mesaje.

- **Comunicare asincronă** – clienții trimit mesaje către coadă, aici mesajele sunt stocate până când o altă aplicație (server) le citește. După ce a scris un mesaj într-o coadă clientul continuă fără să aștepte ca mesajul să fie procesat.
- **Calitatea serviciului (QoS) este configurabilă** – o coadă poate fi configurată pentru performanțe ridicate dar o fiabilitate mai scăzută, respectiv pentru performanțe scăzute, dar fiabilitate ridicată.
- **Asigură o cuplare scăzută** - nu există o legătură directă între aplicațiile client și aplicațiile server. Clientul nu știe ce server va procesa mesajul, iar serverul nu știe de la ce client vine mesajul.

În Tabela 11.4 sunt prezentați indicatorii de calitate care pot fi controlați printr-o arhitectură de tipul coadă de mesaje.

Indicator de Calitate	**Probleme**
Disponibilitatea	Cozile fizice pot fi replicate pe mai multe mașini server, astfel că atunci când o coadă se defectează un client poate trimite mesaje către celelalte replici.
Tratarea erorilor	Dacă un client comunică cu o coadă care se defectează, el poate găsi o altă replică a cozi și să trimită mesajul către acea replică.
Toleranța la modificări	Arhitectura bazată pe cozi de mesaje asigură un grad de decuplare ridicat, ceea ce face ca acest tip de arhitectură să prezinte o toleranță ridicată la modificări, întrucât clientul și serverul nu sunt legați printr-o interfață predefinită. Singura dependență dintre client și server este dată de tipul mesajului folosit, la extrem se pot folosi mesaje al căror format poate fi descoperit dinamic.
Performanța	Tehnologiile bazate pe cozi de mesaje pot procesa mii de mesaje pe secundă. Folosirea unei cozi de mesaje mai puțin fiabilă poate crește performanțele.

Scalabilitatea	Cozile pot fi replicate într-un cluster de servere care pot fi găzduite de aceeași mașină sau de mașini diferite, acest lucru face, ca tehnologiile bazate pe cozi de mesaje să fie foarte scalabile.

Tabela 11.4. Indicatorii de calitate adresați de arhitectura de tip coadă de mesaje.

Acest tip de arhitectură este util atunci când, clientul nu are nevoie imediată de un răspuns. De asemenea această arhitectură este utilă și în cazul când conexiunea la server este sporadică, în această situație mesajul fiind ținut în coadă până când un server se conectează și citește mesajul.

Mesagerie de Tipul Publică-Subscrie

În Fig. 11.5 este prezentat un exemplu de arhitectură bazată pe modelul publică-subscrie. Principalele proprietăți ale acestui tip de arhitectură sunt:

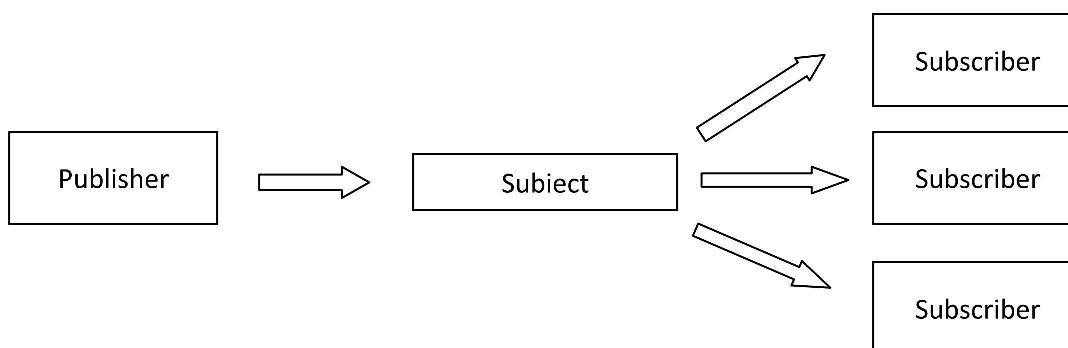

Fig. 11.5. Exemplu de arhitectură bazată pe modelul publică-subscrie.

- **Mesagerie de tipul mai-mulți-la-mai-mulți** – mesajele publicate sunt trimise către toate aplicațiile care au subscris pentru topicul pentru care mesajul a fost transmis. Mai multe aplicații pot publica pentru același topic, respectiv mai multe aplicații pot subscrie pentru același topic.
- **Calitatea serviciului este configurabilă** – în afară de controlul fiabilității mesajelor se poate controla și tipul comunicației care poate fi punct-la-punct sau broadcast/multicast.
- **Cuplare scăzută** – la fel ca și în cazul modelului bazat pe coadă de mesaje și în acest caz nu există o legătură directă între aplicația care publică și cea care a subscris pentru recepționarea mesajelor dintr-un anumit topic.

În Tabela 11.5 sunt prezentați indicatori de calitate care pot fi controlați prin acest tip de arhitectură.

Indicator de Calitate	Probleme
Disponibilitatea	Subiectele cu același nume logic pot fi replicate pe mai multe mașini, astfel atunci când una din mașinile pe care a fost replicat un topic se defectează, aplicațiile care publică pot trimite mesaje către alte replici.
Tratarea erorilor	Dacă un client comunică cu un topic de pe o mașină care se defectează, el poate găsi o altă replică a topicului și să trimită mesajul către acea replică.
Toleranța la modificări	Arhitectura de tipul publică-subscrie asigură un grad de decuplare ridicat, ceea ce face ca acest tip de arhitectură să prezinte o toleranță ridicată la modificări. Noi aplicații care publică respectiv care subscriu pot fi adăugate fără a modifica arhitectura. Singura dependență dintre cele două tipuri de aplicații este dată de

	tipul mesajului folosit, la extrem se pot folosi mesaje al căror format poate fi descoperit dinamic.
Performanța	Tehnologiile bazate pe modelul publică-subscrie pot procesa mii de mesaje pe secundă. Ca și în cazul cozilor de mesaje reducerea fiabilității poate crește performanța. Eficiența este mai ridicată dacă se folosesc mecanisme de comunicare de tipul broadcast sau multicast.
Scalabilitatea	Cozile pot fi replicate într-un cluster de servere care pot fi găzduite de aceeași mașină sau de mașini diferite, acest lucru face, ca tehnologiile bazate pe cozi de mesaje să fie foarte scalabile.

Tabela 11.5 Controlul indicatorilor de calitate în cazul folosirii modelului publică-subscrie.

Arhitecturile de tipul publică-subscrie sunt extrem de flexibile, ele sunt potrivite pentru aplicații care au nevoie de comunicare asincronă de tipul unu-la-mai-mulți, mai-mulți-la-unu sau mai-mulți-la-mai-mulți.

Brocăr

În Fig. 11.6 este prezentată o arhitectură de tip brocăr. Principalele proprietăți ale unei astfel de arhitecturi sunt:

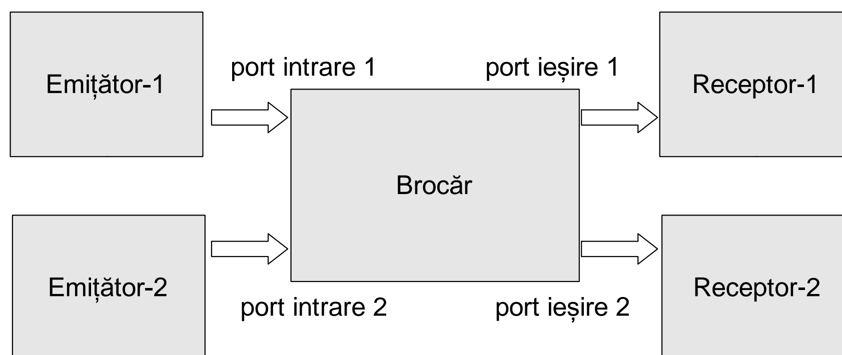

Fig. 11.6.Exemplu de arhitectură de tip brocăr.

- **Comunicare de tip hub** – brocărul acționează ca și un hub la care se conectează aplicațiile care transmit și cele care recepționează. Comunicarea cu brocărul se realizează prin porturi care au asociat un anumit tip de mesaj.
- **Realizează routarea mesajelor** – un brocăr conține funcționaliate care face ca mesajele recepționate la un port să fie redirecționate către altul. Routarea poate fi codificată în cod sau poate să fie dependentă de informația prezentă în mesaj.
- **Realizează transformarea mesajelor** – logica implementată pe brocăr transformă mesajul sursă de la portul de intrare în mesajul care respectă formatul de la portul de ieșire.

În Tabela 11.6 sunt prezentate modalitățile de control asupra indicatorilor de calitate utilizând acestă arhitectură.

Indicator de Calitate	**Probleme**
Disponibilitatea	Pentru a se obține o disponibilitate mare este nevoie ca brocării să fie replicați.
Tratarea erorilor	Întrucât porturile unui brocăr au asociat un anumit format de mesaj, orice mesaj care nu respectă un anumit format este ignorat. În cazul în care se folosește replicarea brocărilor atunci când un emițător comunică cu un brocăr

	defect el poate alege o altă replică a brocărului.
Toleranța la modificări	Un brocăr separă logica de transformare și rutare a mesajelor de emițători și receptori. Acest lucru îmbunătățește toleranța la modificări întrucât modificarea logicii de transformare și rutare nu afectează emițătorul sau receptorul.
Performanța	Brocării pot să devină o limitare în ceea ce privește performanța, mai ales atunci când volumul de mesaje procesate este ridicat, iar logica de procesare complexă.
Scalabilitatea	Utilizarea clusterelor de brocări pot face posibilă construirea unui sistem scalabil.

Tabela 11.6 Controlul indicatorilor de calitate în cazul folosirii arhitecturii de tip brocăr.

Arhitecturile de tipul brocăr sunt utile pentru sistem software care conțin componente ce comunică prin mesaje care necesită foarte multe transformări.

Coordonator de Procese

Arhitectura unui coordonator de procese este prezentată în Fig. 11.7. Principalele elemente ale acestui model sunt:

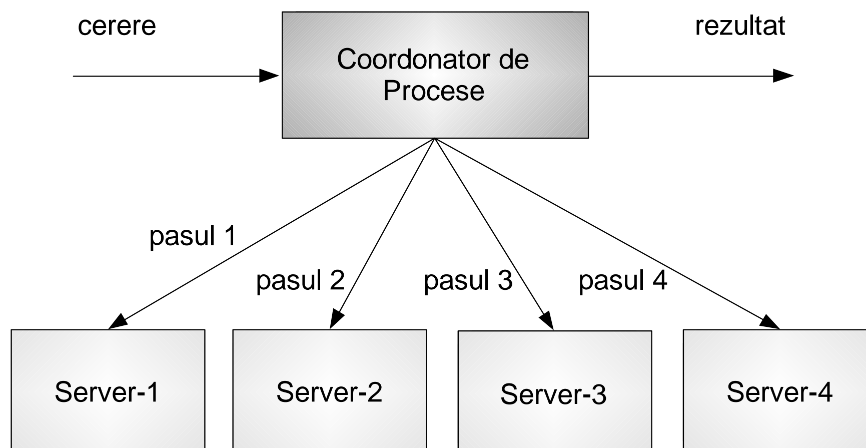

Fig. 11.7.Exemplu de arhitectura de tip coordonator de proces.

- **Încapsularea proceselor**: coordonatorul de procese încapsulează secvența de pași necesară pentru a îndeplinii funcționalitatea dorită. Secvența poate fie oricât de complexă. Coordonatorul reprezintă un punct central pentru definirea procesului business, ceea ce face ca procesul să fie ușor de înțeles și modificat. Acest proces coordonator primește o cerere de inițializare a procesului business, după care trimite cereri către fiecare server în ordinea definită de procesul business, iar în final emite rezultatul.
- **Cuplare redusă**: componentele server nu “cunosc” întregul proces business nici măcar rolul pe care ele îl au în acest proces. Serverele pur și simplu implementează un set de servicii care sunt apelate de procesul coordonator atunci când este nevoie de ele.
- **Comunicare flexibilă**:comunicarea între procesul coordonator și servere poate fi sincronă sau asincronă.

În Tabela 11.7 sunt prezentate modalitățile de control asupra indicatorilor de calitate utilizând acestă arhitectură.

Indicator de Calitate	Probleme
Disponibilitatea	Coordonatorul reprezintă un singur punct de defectare, de aceea pentru a se îmbunătății disponibilitatea acesta trebuie replicat.
Tratarea erorilor	Tratarea erorilor este foarte complexă întrucât defecțiuni pot să apară în orice etapă a procesului. Apariția unei defecțiunii într-un anumit pas poate să însemne anularea unor acțiuni realizate într-un pas anterior. Tratarea erorilor trebuie făcută cu mare atenție pentru a se asigura consistența datelor.
Toleranța la modificări	Arhitectura de tip coordonator de procese prezintă o toleranță mare la modificări întrucât întreg procesul de business este definit într-un singur loc. În plus serverele pot fi reimplementate fără ca acest lucru să necesite modificări ale procesului coordonator sau ale celorlalte servere.
Performanța	Pentru a se asigura o performanță ridicată, procesul coordonator trebuie să poată trata un număr mare de cereri simultane. De asemenea performanța întregului proces va fi limitată de cel mai încet pas.
Scalabilitatea	Procesul coordonator poate fi replicat pentru a se obține scalabilitate atât în interior cât și în exterior.

Tabela 11.7 Controlul indicatorilor de calitate în cazul folosirii arhitecturii de tip coordonator de procese.

13.3.2 Definirea componentelor

După ce a fost definit scheletul arhitecturii prin selectarea unuia sau mai multor modele, următorul pas constă din definirea principalelor componente. Pentru aceasta trebuie să se:

- Identifice principalele componente ale sistemului, precum și modul în care ele sunt integrate în scheletul arhitecturii.
- Identifice interfața și serviciile disponibile pentru fiecare componentă în parte.
- Identifice responsabilitățile componentelor.
- Identifice dependențele între componente.
- Identifice bucățile din arhitectură care pot fi distribuite pe mai multe servere din rețea.

Câteva indicații pentru proiectarea componentelor:

- **Minimizarea dependențelor între componente**. Proiectarea de componente slab cuplate astfel ca modificările realizate într-o componentă să nu se propage la restul componentelor. De fiecare dată când se face o modificare sistemul trebuie retestat.
- **Proiectarea componentelor astfel încât să înglobeze un set de responsabilități bine legate**. Componentele trebuie să utilizeze un set restrâns bine definit de interfețe. Astfel se reduce posibilitatea de a avea modificări care să se propage la mai multe componente, se minimizează efortul de întreținere și testare.
- **Izolarea dependențelor de tehnologii middleware**.Cu cât numărul de componente care depind de tehnologii middlewareeste mai mic cu atât este mai ușor să se treacă la o versiune mai nouă de middleware sau să se schimbe tehnologia middleware.
- **Descompunerea ierarhică în componente.**
- **Minimizarea numărului de apeluri între componente.** Numărul de apeluri poate fi redus prin agregarea unei secvențe de apeluri într-un singur apel.

13.4 Validarea Arhitecturii

Scopul etapei de validare este acela de a verifica faptul că arhitectura proiectată este potrivită pentru sistemul software care urmează să fie dezvoltat. Principala dificultate în ceea ce privește validarea unei arhitecturi constă în faptul că în acest moment nu există un produs software “fizic” care să poată fi executat și testat. Există două metode prin care se poate valida arhitectura unui sistem software: testarea manuală utilizând scenarii, respectiv validare prin construirea unui prototip.

Utilizarea Scenariilor

Utilizarea scenariilor pentru validarea arhitecturii unui sistem software presupune definirea unor stimuli care să aibă efect asupra arhitecturii. După care se face o analiză pentru a se determina care va fi răspunsul arhitecturii la un astfel de scenariu. Dacă răspunsul este cel dorit atunci se consideră că scenariul este satisfăcut de arhitectură. Dacă răspunsul nu este cel dorit sau este greu de calificat atunci s-a descoperit o zonă de risc în arhitectură. Se pot imagina scenarii care să evalueze oricare din cerințele sistemului.

Crearea Unui Prototip

Deși scenariile sunt tehnici foarte utile în vederea testării unei arhitecturi, nu întotdeauna verificarea unui scenariu se poate face doar prin analiza arhitecturii, de aceea în anumite situații se recurge la construirea unui prototip care să permită verificarea anumitor scenarii. Un prototip se poate construi din două motive:

- **Proof-of-concept**: verifică dacă se poate implementa arhitectura proiectată astfel încât să satisfacă cerințele.
- **Proof-of-technology**: verifică faptul că tehnologia middleware selectată se comportă așa cum se așteaptă.

Odată ce prototipul a fost implementat și testat răspunsul arhitecturii la stimulii prevăzuți în scenarii se poate obține cu un înalt grad de certitudine.

Deși un prototip poate fi foarte util în ceea ce privește validarea unei arhitecturi, trebuie avut grijă ca dezvoltarea unui astfel de prototip să nu dureze prea mult. De obicei un prototip este abandonat după ce arhitectura a fost validată, de aceea dezvoltarea unui prototip nu trebuie să dureze mai mult de câteva zile, cel mult 1-2 săptămâni.

Bibliografie

[1] Ian Gorton. Essential Software Architecture. Editura Springer. 2006.