Facultatea de Electronică, Telecomunicații
și Tehnologia Informației

Cercetare științifică

Arhitecturi software orientate pe servicii

Andrei Mihăescu

Coordonator științific
Conf. Dr. Eduard Popovici

Cuprins

Lista figurilor

Lista tabelelor

Lista acronimelor

BDFU - Big Design Upfront
COBRA - Common Object Request Broker Architecture
COM - Common Object Model
DCOM - Distributed Common Object Model
DRY - Don't repeat yourself
EJB - Enterprise Java Beans
ESB - Enterprise Service Bus
ISB - Internet Service Bus
JIT - Just in time
LOB - Line-of-business
MVC - Model View Controller
P2P - Peer-to-Peer
QA - Quality Assurance
SOA - Service Oriented Architecture
UML - Unified Modelling Language
URI - Uniform Resource Identifiers
YAGNI - You ain't gonna need it

Capitolul 1
Introducere

1.1 Ce este o arhitectura software?

Arhitectura software reprezintă procesul de definire a unei soluții structurate care îndeplinește toate cerințele tehnice și operaționale, totodata optimizând metrici comune de calitate precum performanța, securitatea si facila gestiune. Aceasta presupune o serie de decizii bazate pe o gamă largă de factori fiecare din aceștia având un impact considerabil asupra calității, performanței, gestionabilitații și bunei funcționarii a aplicației.

Philippe Kruchten, Grady Booch, Kurt Bittner și Rich Reitman au derivat și rafinat definiția arhitecturii software bazându-se pe munca lui Mark Shaw si David Garlan (Shawn and Garlan 1996). Definiția lor este următoarea:

"Arhitectura software înglobează setul deciziilor semnificative legate de organizarea unui sistem software ce includ selectarea elementelor structurale și a interfețelor din care sistemul este compus; comportamentul așa cum reiese din interacțiunea acestor elemente; compunerea acestor elemente structurale și comportamentale în subsisteme mai mari; și un stil arhitectural care guvernează această organizare. Arhitectura software implică constrângeri și compromisuri legate de funcționalitate, utilitate, robustețe, performanță, reutilizare, inteligibilitate, economie, tehnice și estetice."

În cartea *"Patterns of Enterprise Application Architecture"*, Martin Fowler evidențiază câteva teme recurente explicând conceptul de arhitectură. El identifică aceste teme dupa cum urmează: "Descompunerea de nivel înalt a unui sistem în parți componente; deciziile care sunt dificil de schimbat; există multe arhitecturi intr-un sistem; ceea ce este arhitectural important se poate schimba de-a lungul ciclului de viata al sistemului; și, la final, arhitectura se rezumă la lucrurile importante."

În cartea *"Software Architecture in Practice (2nd edition)"* Bass, Clements, and Kazman definesc arhitectura astfel: "Arhitectura software a unui program sau a unui sistem de calcul reprezintă structura sau structurile, ce înglobează elementele software, proprietățile lor vizibile către exterior și relația între acestea. Arhitectura se preocupă cu partea public a interfețelor; detaliile private ale elementelor - cele ce sunt strict legate de implementarea internă - nu sunt legate de arhitectură."

1.2 De ce este arhitectura importantă ?

Ca orice structură complexă, software-ul trebuie construit pe o bază solidă. Neluarea în considerare a anumitor scenarii cheie, cât și ignorarea anumitor probleme comune de design pot pune aplicația în pericol. Uneltele și platformele moderne ajută la construirea aplicațiilor, însă nu pot înlocui nevoia unuei proiectării atente a aplicației, bazată pe scenarii și cerințe. Riscurile pe care le presupune o arhitectură slab gândită sunt instabilitatea, inabilitatea de a susține cerințele actuale și viitoare de business sau dificultatea de instalare și gestiune într-un mediu de producție.

Un sistem ar trebui proiectat luând în considerare utilizatorul, infrastructura și obiectivele de business. Pentru fiecare din aceste arii, trebuie gândite scenarii cheie, identificate proprietațile relevante și arii cheie de satisfacție și desatisfacție. Unde este posibil, este indicat sa se stabilească metrici precise care vor putea fii evaluate pentru a măsura succesul fiecărei arii.

Cel mai probabil vor exista compromisuri și un echilibru trebuie găsit între cerințele concurente din aceste trei arii. De exemplu experiența utilizatorului, este adesea o funcție de business și infrastructură și schimbările într-una din zone o poate drastic afecta. În mod similar, schimbările la nivelul experienței de utilizare pot avea impact la nivelul infrastructurii și business. Performanța ar putea fi importantă din punct de vedere business și al utilizatorului, dar administratorul de sistem poate nu avea mijloacele financiare pentru a atinge obiectivele în 100% din timp. Un compromis ar fi să atingă obiectivele 80% din timp.

Rolul arhitecturii este acela de a defini modul în care elemente majore și componente din cadrul aplicației sunt folosite sau interacționează între ele. Selectarea structurilor de date și a algoritmilor, cât și detaliile de implementare specifice fiecărei componente nu sunt de interes pentru proiectare. Problemele de arhitectură și proiectare de multe ori se intercalează. In unele cazuri, deciziile țin mai mult de arhitectură. In altele, în schimb țin mai mult de proiectare și de cum acestea contribuie la realizarea arhitecturii.

1.3 Obiectivele arhitecturii software

Arhitectura software urmărește să găsească un compromis între specificațiile de business și cele tehnice înțelegând cazurile de utilizare și apoi găsind căi de implementare. Obiectivele arhitecturii sunt acelea de a găsi cerințele care influențează structura aplicației. O arhitectură bine realizată reduce riscurile de business asociate cu construirea unei soluții tehnice. Un design bun este suficient de flexibil ca să poate gestiona devierile de la tehnologiile software si hardwre care pot apărea în timp, cât și cerințele utilizatorilor. Un arhitect trebuie să ia in considerare efectul global al deciziilor de proiectare, cât și compromisurile inerente între factorii de perfomanță, dar și cele cu privire la utilizator, infrastructură și cerințele de business.

1.4 Principii arhitecturale cheie

Pentru proiectarea unei arhitecturii următoarele principii cheie ar trebui luate în calcul:

- **Arhitectura trebuie sa fie concepută pentru a suporta schimbări, nu pentru a rămâne neschimbătă.** Întotdeauna trebuie luat în calcul cum aplicația s-ar putea modifica de-a lungul timpului pentru a putea răspunde noilor cerințe și provocări.
- **Modelarea trebuie facută pentru reduce riscul.** Este indicată folosirea uneltelor de proiectare, a sistemelor de modelare precum **UML -** *Unified Modelling Language* și a vizualizare unde este cazul pentru a putea evidenția cerințele și a deciziile arhitecturale și de proiectare, analizându-le impactul.
- **Folosirea modelelor și a uneltelor de vizualizare pentru comunicare si colaborare.** Comunicarea eficientă a designului, a deciziilor luate și schimbărilor necesare este necesară unei arhitecturi bune. Este recomandată folosirea modelelor, a vederilor și a altor mijloace de vizualire a arhitecturii pentru a comunica și impărtăși eficient ideile cu clienții, rezultând astfel într-o comunicare rapidă a schimbărilor de proiectare.
- **Identificarea deciziilor tehnice cheie.** Este esențiala înțelegerea deciziilor tehnice cheie pentru evitarea greșelilor comune. Investirea în luarea acestor decizii bine de prima dată este importantă pentru a obține un design flexibil și foarte puțin expus riscului de a fi afectat de viitoare schimbări.

Pentru rafinarea arhitecturii este recomandată o abordare incrementală și iterativă. Se pornește de la o arhitectură de bază pentru a avea o imagine de ansamblu, după care crează arhitecturii derivate pe măsură ce aceasta este testată și îmbunătățită. Modelul incipient nu trebuie să acopere toate nevoiele, ci ar trebui să reprezinte un prim design pe care să poată fii testate cerințele de business. In mod iterativ vor fi adăugate detaliile care vor putea o primă imagine de ansamblu corectă, pentru ca mai târziu să fie adăugate și detaliile de finețe. O greșeală foarte des întâlnită este aceea de a se concentra pe detaliile neesențiale încă din faze incipiente și de a urma direcții greșite făcând presupuneri incorecte sau eșuând în a evalua eficient arhitectura.

Capitolul 2
Principii fundamentale ale arhitecturii software

Acest capitol va aborda principiile fundamentale de proiectare ale unei arhitecturii software. Aceasta este adesea descrisă ca organizarea sau structura unui sistem, care reprezintă o colecție de componente ce îndeplinesc o funcție sau un set de funcții specifice. Cu alte cuvinte, arhitectura se concentrează pe organizarea componenteleor ce vor oferi o anumită funcționalitate. Organizarea funcțională a componentelor implică grupare acestora în "arii de interes", după cum se poate vedea in schemă de mai jos.

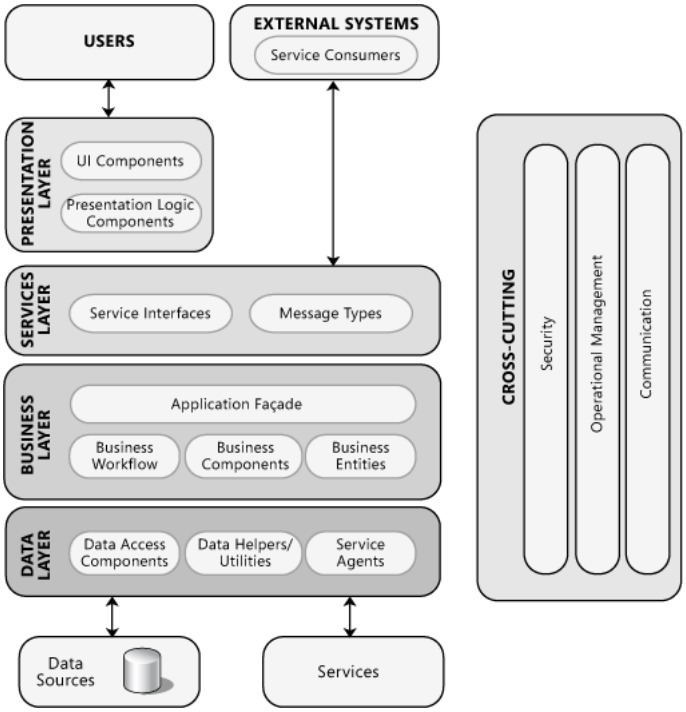

Figura 2.1: Organizarea pe "arii de responsabilitate"

Pe lângă gruparea pe componente, alte arii de interes se concentrează pe interacțiunea între acestea și pe cum ele funcționează împreună.

2.1 Principii de proiectare

La începerea procesului de proiectare trebuie avute în vedere principiile care vor contribui la crearea unei arhitecturii care aderă la practici consacrate, reduce costurile și care încurajează ușurința de folosire și posibilitățile de extindere. Acestea sunt:

- **Separarea atribuțiilor.** Aplicația trebuie divizată în funcționalități distincte cu o suprapunere cât mai mică între acestea. Factori importanți pentru a obține aceasta o reprezintă un nivel mare de coeziune și o cuplare slabă între componente, însă separarea greșită funcționalităților poate duce la o cuplare strânsă între acestea chiar dacă fiecare atribuțiile fiecărei componente nu se suprapun.

- **Principiul responsabilității unice.** Fiecare componentă sau modul ar trebui să îndeplinească o singură funcționalitate sau o agregare coerentă de funcționalități.

- **Principiul cunoașterii minime (cunoscut și sub numele de Legea lui Demeter).** O componentă sau un obiect nu ar trebui să cunoască detaliile interne de implementare ale altor componente sau obiecte.

- **Nu te repeta (eng. Don't repeat yourself - DRY)** Funcționalitatea ar trebui sa fie implementată într-un singur loc. De exemplu, în cadrul proiectării unei aplicații, funcționalitate ar trebui implementată într-o singură componentă și nu ar mai trebui duplicată și în alta.

- **Minimizarea proiectării anticipate.** Proiectarea ar trebui să se rezume doar la ceea ce este necesar. În unele cazuri însă, atunci când costurile de dezvoltare sau al uni eșec în design sunt foarte mari, proiectarea anticipată este necesară. În altele, în special în cadrul dezvoltării agile, aceasta (eng. **BDUF - big design upfront**) poate fii evitată. Dacă cerințele aplicației sunt neclare sau există posibilitate unor evoluții a designului în viitorul apropiat, evitarea eforturilor de proiectare prematură este indicată. Acest principiu se mai numește si **YAGNI (eng. "You ain't gonna need it").**

În cadrul proiectării unuei aplicații sau a unui sistem, obiectivul unui arhitect este să minimizeze complexitatea separând designul în mai multe arii de responsabilitate. De exemplu, interfața grafică, procesele de business și accesul la date reprezintă arii diferinte. În cadrul fiecăreia, componentele proiectate ar trebui să fie focusate pe responsabilități diferite, fără a conține cod din cadrul alteia. De exemplu, interfața grafică nu va conține cod care accesează direct sursele de date; în schimb va folosi componente specializate pentru obținerea datelor.

O evaluare cost/beneficiu va fi folosită pentru determinarea investiției necesare. In unele cazuri poate fii necesară o simplificare a structurii, permițând legarea interfeței grafice la un set de date. În general, separarea funcționalităților va fii făcută ținând cont și de partea de business. Cele ce urmează vor prezenta factori care afecteaza ușurința de design, implementare, testare și mentenanță a aplicației.

2.1.1 Standarde de proiectare

La nivelul fiecărui nivel logic, acolo unde este posibil, proiectare componentelor ar trebui sa fie consitentă în cadrul unei operații. De exemplu, dacă se folosește tiparul "Table Data Gateway" pentru a crea un obiect care să reprezintă punctul de acces al tabelelor dintr-o bază de date, nu ar mai trebui folosit un altul (ex. "Repository") care folosește o altă paradigmă pentru accesarea datelor și inițializarea entităților de business. Însă, este posibilă folosirea altor tipare pentru operațiuni al unui alt modul care are o varietate mai large de cerințe, cum ar fi o aplicație care necesită tranzacții business și rapoarte.

Funcționalitatea nu trebuie duplicată în cadrul aceleași aplicații; o singură componentă va oferi această funcționalitate, care nu va mai exista într-o alta. Aceasta va duce la componente coerente și va ușura optimizarea acestora dacă o anumită funcționalitate va suferi modificări. În caz contrar, duplicarea funcționalității va îngreuna implementarea schimbărilor, scădea claritatea și va introduce potențiale inconsistențe.

Atunci cand este posibil, compoziția este preferată în defavoarea moștenirii pentru refolosirea funcționalității deoarecere moștenirea crește depedența între părinte și clase care moștenesc. Aceasta de asemenea reduce ierarhiile de moștenire, ceea ce poate deveni dificil de gestionat.

Stabilirea unui stil de codare și a unei convenții de denumire pentru dezvoltare va oferi un model consisten care va facilita revizuirea codului de către membrii ai echipei care nu l-au scris, ceea ce duce la o administrare imbunătățită.

Menținerea calității sistemului se poate face folosind tehnici automatizate de QA pe parcursul dezvoltării, precum scrierea de teste unitare, analiza de dependențe și analiza statică a codului pe parcursul dezvoltării. Define clear behavioral and performance metrics for components and sub-systems, and use automated QA tools during the build process to ensure that local design or implementation decisions do not adversely affect the overall system quality.

Proiectarea componentelor aplicației și a sub-sistemelor cu o înțelegere clară a nevoilor operaționale specifice fiecăreia va ușura seminificativ instalarea și mentenanța acesteia. În vederea eficientizării acestor două operațiuni metrici și date operaționale sunt necesare echipei responsabile de infrastructură. Pentru aceasta folosirea unetelor automitizate pentru asigurarea calității pot fi folosite.

2.1.2 Nivelele aplicației

Pentru a pune în practica principiul "ariilor de responsabilitate" aplicația este împărțită în blocuri funcționale a căror funcționalitate se suprapune cât mai puțin. Mare avantaj al acestei abordări este că fiecare bloc funcțional va putea fi optimizat independent față de restul aplicației. În plus, dacă unul încetează să funcționeze, acesta nu va cauza și defectarea altora, iar procesele pot rula independent unul față de celălalt. Această abordare contribuie și la facilitarea înțelegerii și proiectării aplicațiilor și simplifică administrarea sistemelor complexe interdependente.

Permițând fiecărui nivel să comunice cu celălalte sau să fie dependent de alte nivele va crește gradul de complexitate al aplicației și o va face mai greu de înțeles și administrat, așadar regulile de interacțiune între acestea trebuie foarte bine explicitate, astfel încât fluxul datelor în aplicație să fie foarte limpede.

Implementarea cuplării slabe între nivele se poate implementa prin folosirea abstractizării, definindu-se astfel componente-interfața precum "façade" cu intrări și ieșiri bine cunoscute care se traduc prin cereri într-un format cunoscut înțeles de nivelul respectiv. În plus, se pot folosi tipurile Interface sau clase de bază abstracte pentru a implement o interfață comună.

Separarea tipurilor de component în cadrul aceluiași nivel logic se poate obține prin identificarea diferitelor arii de responsabilitate și apoi prin gruparea componentelor asociate fiecăruia

pe nivele logice. De exemplu, interfața grafică nu ar trebui să conțină componente legate de procesare business, însă ar trebui sa conțină elemente care permit utilizatorului să introducă date.

Amestecând formatele datelor în cadrul unui nivel sau a unei component va face ca aplicația să devină mai dificl de implementat, extins și administrat, așadar formatul trebuie să fie consistent. În caz contrar, de fiecare dată când este necesară trecerea de la un format la atul, o operație de traducere trebuie efectuată ceea ce atrage după sine efort suplimentar.

2.1.3 Componente, module și funcții

O componentă sau un obiect nu trebuie să depindă de detaliile interne ale unie alte componente sau obiect. Fiecare componentă sau obiect ar trebuie sa apeleze o metodă a unui alt obiect sau component, iar aceasta să aibaă informații despre cum să proceseze cererea și eventual cum să o redirecționeze către altele.

Funcționalitatea unei componente nu trebuie supraincărcată. De exemplu, o componentă a interfeței grafice nu va conține cod specific accesului datelor și nici nu va încerca să ofere o funcționalitate suplimentară. Adesea componentele supraîncărcate au multe funcțiișiși proprietăți care furnizeaza funcționalități de business amestecate cu functionalități transverse precum logare și tratarea excepțiilor. Resultatul este un design foarte predispus erorilor și dificil de menținut. Aplicarea principiului responsabilității unice și a separării ariilor de responsabilitate contribuie la evitarea acestui lucru.

Înțelegerea felului în care componentele comunică între ele presupune înțelegerea cazurilor de utilizare pentru care aplicația a fost concepută. Trebuie stabilit dacă toate componentele vor rula în cadrul aceluiași proces sau dacă comunicarea dincolo de barierele fizice sau de proces este necesară, implementând interfețe de comunicare.

Codul functionalităților auxiliare trebuie separat de logica de business. Acest cod este cel responsabil pentru securitate, comunicare și manangement operațional precum logare și instrumentație. Amestecarea codului care implementează aceste funcții cu logica de business poate duce la un design dificil de extins și menținut. Schimbările acestui cod necesită implică alterarea logicii de business. Pentru a rezolva această problemă se recomandă folosirea librăriilor și tehnicilor, precum programarea orientate pe aspecte.

Componentele, modulele și funcțiile ar trebui să defineăscă un contratct sau o interfață care descrie în mod explicit utilizare și comportamentul acestora. Un contract conține o descriere explicând cum celălalte component funcționalitățile componentei, modulului sau funcției alături de comportamentul acesteia înainte de apelare, după apelare, efecte adverse ales acesteia, excepții, caracteristici de perfomanță și alți factori.

Capitolul 3
Tipare și stiluri arhitecturale

În acest capitol voi prezenta tipare și principii de nivel înalt des întâlnit în aplicațille. Acestea sunt adesea numite stiluri arhitecturale și includ tipare cum ar fi client/server, arhitectura stratificată, arhitectură bazată pe componente, arhitectură cu bus de mesaje și arhitectura orientată pe obiecte (SOA). Pentru fiecare stil, voi prezenta o imagine de ansamblu, caractersiticiile principale, advantajele și informații legate de ce stil se potrivește cărei aplicații. Este important de înțeles că stilurile descriu diferite aspecte ale aplicațiilor. De exemplu, unele stiluri ahitecturale descriu tipare de instalare, altele descriu probleme legate de structuri și design, iar altele descriu soluții de comunicare. Prin urmare o aplicație tipică va folosi o combinație a mai mult de un stil din cele descrise.

3.1 Ce este un stil arhitectural?

Un stil arhitectural, adesea numit și tipar arhitectural, reprezintă un set de principii care crează un cadru abstract pentru o familie de sisteme. Un stil arhiteectural îmbunătățește partiționarea și încurajează reutilizarea designului oferind soluții la probleme recurente. Tiparele și stiluri ahitecturale pot fi privite ca și principii care conturează forma unei aplicații. Garlan și Shaw definesc un stil arhitectural : *"...o familie de sisteme în termenii unui tipar de organizare structurală. Mai exact, un stil arhitecetural determină vocabularul de componente și conectori care pot fi utilizați în cadrul său, împreuna cu un set de constrângeri care explică cum acestea pot fi combinate. Acestea pot include constrângeri legate de topologie (ex. fără cicluri). Alte constrângeri ar putea face parte din definiția stilului."*

Înțelegerea stilurilor arhitecturale oferă mai multe avantaje. Cel mai important ar fi că oferă un limbaj comun. Oferă de asemenea posibilitate unor discuții agnostice cu privire la tehnologie, care facilitează discuții la nivel înalt incluzând tipare și principii, fară a intra în detalii. De exemplu, folosind stiluri arhitecturale, se poate discuta despre client/server versus n-niveluri. Stilurile arhitecturale pot fi organizate în funcție de aria lor cheie. Următoarea lista enumeră zonele majore de interes alături de stilul lor corespunzător.

Categorie	Stil arhitectural
Comunicare	Arhitectură orientată pe servicii, Bus de mesaje
Instalare	Client/Server, N-Niveluri
Structură	Arhitectură stratificată

Tabela 3.1: Categorii și stiluri arhitecturale

3.1.1 Sumar al stilurilor arhitecturale

Următorul table enumeră stilurile arhitecturale descrise în cadrul acestei lucrări, împreună cu o scurtă descriere a fiecăruia.

Stil arhitectural	Descriere
Client/server	Împarte sistemul în două aplicații, în care clientu lansează cereri către server. În multe cazuri, serverul este o bază de date cu logică implementată în proceduri stocate.
Arhitectură bazată pe componente	Împarte aplicația în componente funcționale sau logice reutilizabile care expun interfețe de comunicare bine cunoscute.
Arhitectură stratificată	Partiționează aplicația în grupuri funcționale numite niveluri.
Bus de mesaje	Un stil arhitectural care implică folosirea unui sistem software care poate primi și trimite mesaje folosind unu sau mai multe canale de comunicare, astfel încât applicațiile să poată interacționa fără a cunoaște detalii specifice.
N-niveluri/3-niveluri	Împarte aplicația în blocuri funcționale la fel ca arhitectura stratificată însă fiecare segment este localizat pe o mașină fizică diferită.
Obiect orientată	O paradigmă bazată pe împărțirea responsabilităților unei aplicații sau sistem în componente individuale reutilizabile și obiecte, fiecare conținând datele și comportamentul relevante.
Arhitectură orientată pe servicii (SOA)	Se referă la aplicații care expun și consumă functionalități prin intermediul serviciilor folosind contracte și mesaje.

Tabela 3.2: Sumar al stiluri arhitecturale

3.1.2 Combinarea stilurilor arhitecturale

Arhitectura unui sistem software nu este aproape niciodată limitată la doar un stil arhitectural, însă este adesea o combinație a mai multor stiluri care formează un sistem complet. De exemplu am putea avea un sistem cu design SOA a cărui servicii au fost dezvoltate folosind o arhitectura stratificată și una orientată pe obiecte.

O combinație de stiluri este de asemenea utilă dacă se dorește construirea unei aplicații web, unde se poate obține o separare eficientă a funcționalității folosind arhitectura stratificată. Aceasta va separa logica de afișare de logica de business și de cea de acces la date. Din motive de securitate poate fi impusă o instalare a aplicației pe 3 niveluri sau chiar pe mai multe. Nivelul de prezentare poate fi instalat în partea demilitarizată a rețelei companiei. In cadrul acestui nivel se poate folosi un tipar separat pentru presentare, cum ar fi Model-View-Contoller (MVC), pentru modelul de interacțiune. Se poate folosi și SOA pentru implementarea unei comunicații orientată pe mesaje între serverul web și serverul aplicativ.

In cazul unei aplicații desktop, clientul va trimite cereri către server. Ceea ce se pretează în acest caz este arhitectura client/server împreună cu abordarea orientată pe componente pentru

a descompune mai departe designul în module care expun intefețele adecvate de comunicare. Folosirea designul orientat pe obiecte ar îmbunătății reutilizabilitate, testarea și flexibilitatea.

Mulți factori pot influența alegerea stilului. Aceștia includ și capacitatea de design și implementare a organizației; capacitatea și experiența dezvoltatorilor; infrastructura și constrângerile organizaționale.

3.2 Arhitectura client/server

Arhitectura client/server descrie sistemele distribuite care implică separarea clientului și serverului și conectarea acestora prin rețea. Cea mai simplă implementare a unui sistem client/server implică un server care este accesat direct de mai mulți client, adesea numit stil arhitectural pe 2 niveluri.

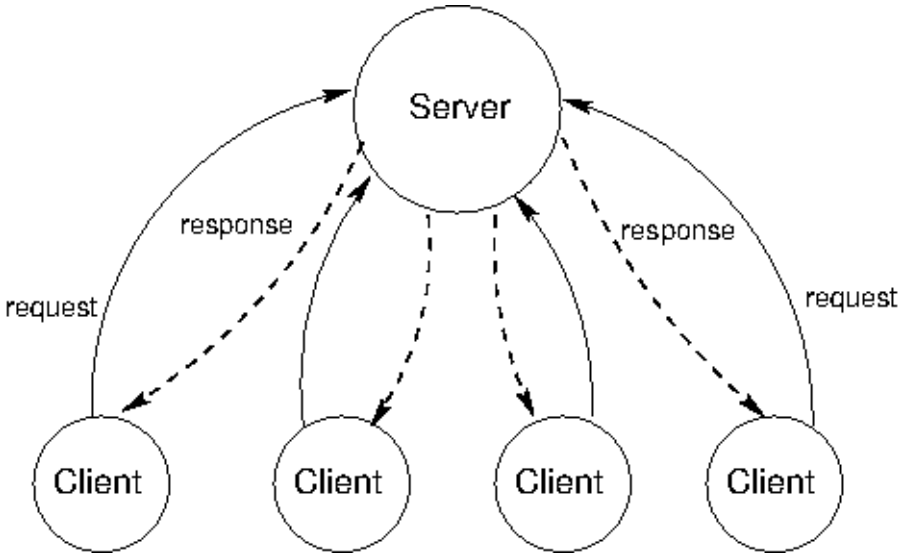

Figura 3.1: Client/server

In mod tradițional, arhitectura client/server era implementată printr-o aplicație desktop cu interfață grafică ce comunica cu un server de baze date conținândn majoritatea logicii de business în proceduri stocate. Mai generic, însă, arhitectura client/server descrie relația între un client și unul sau mai multe server, clientul inițiând una sau mai multe cereri folosind interfața grafică, asteaptă raspunsuri și execută procesarile la recepție. Serverul în mod tipic autorizează utilizatorul și apoi declanșează procesele necesare pentru generarea rezultatului. Serverul poate sa trimită răspunsurile folosind o varietate de protocoale și formate de date pentru a comunica informația clientului.

Astăzi exemple ale arhitecturii client/server includ programele bazate browserele Web ce rulează pe Internet sau intranet; aplicații ale sistemului de operare care accesează servicii de date prin intermediul rețelei; aplicații care accesează surse de date distante; unelte si utilitare pentru manipularea sistemelor la distanță.

Alte variațiuni ale stilului client/server includ:

- Sisteme client-coadă-client. Această abordare permite clienților să comunice cu alți clienți printr-o coadă aflată pe un server.Clienții pot citi și pot trimite date către un server care se comportă precum o simplă coadă pentru stocarea informațiilor. Asta permite clienților sa distribuie și să sincronizeze fișiere și informații. Aceasta se mai numește uneori și arhitectura cozii pasive.

- Aplicații peer-to-peer (P2P). Dezolvat plecând de la stilul precedent, P2P permite clientului și serverului să-și inverseze rolurile cu scopul de a distribui și sincroniza informații și fisiere pe mai mulți clienți. Extinde stilul client/server prin generarea mai multor răspunsuri la cereri, date partajate, descoperirea resurselor și redundanță față de pierderea unor noduri.

- Servere aplicative. Un stil arhtectural în care serverul găzduiește și execută aplicații și servicii pe care un client lejer le accesează prin intermediul unui browser sau a unui program specializat. Un astfel de exemplu este un client care execută o aplicație care rulează bazându-se pe servicii terminal.

Principalele beneficii ale acestui stil sunt:

- Securitate sporită. Toate datele sunt stocate pe server, ceea ce în general oferă un control al securității mai bun decât mașinile client.

- Acces centralizat la date. Deoarece datele sunt stocate numai pe server, accesul și actualizările datelor sunt mult mai ușor de administrat decât în orice alt stil arhitectural.

- Mentenanță facilă. Rolurile și responsabilitățile unui sistem de calcul sunt distribuite între mai multe servere care sunt cunoscute prin intermediul rețelei. Aceasta presupune trasparență pentru client vizavi de posibile reparații, actualizări sau relocări ale unui server.

Acest stil se pretează foarte bine daca aplicația ce se dorește a fi dezvoltată este: un server care va deservi mulți client, o aplicație Web, aplicația conține procese de business care vor fi utilizate în interiorul organizației sau se dorește crearea unor servicii care vor fi consumate de alte aplicații. Stilul arhitectural client/server este deasemenea adecvat atunci când se dorește centralizarea datelor, crearea redundanței și a funcțiilor de administrare sau când aplicația trebuie să deservească diverse tipuri de clienți și dispozitive.

Totuși, tradiționalul stil client/server pe 2-Niveluri are numeroase dezavantaje cum ar fi tendința de corelarea strânsă a datelor aplicației și a logicii de business pe server, ceea ce poate impacta în mod negativ scalabilitatea și dependența de serverul central, ceea ce afectează și fiabilitatea sistemului. Pentru a rezolva această problemă stilul arhitectural a evoluat într-unul ceva mai generic pe 3-Nivele (N-Nivele), descris în cele ce urmează și care reușește să înlăture dezavantajele inerente ale modelului pe 2 nivele, dar păstrează avantajele acestuia.

3.3 Arhitectura bazată pe componente

Ahitectura bazată pe componente descrie o abordare a inginerii software pentru proiectare și dezvoltarea sistemelor. Se concentrează pe descompunerea designul pe componente funcțional individuale sau logice care expun interfețe de comunicare bine definite conținând metode, eveniment și proprietăți. Aceasta crează un nivel mai înalt de abstractizare decât principiile obiect orientării și nu se axează pe probleme cum ar fi protocoale de comunicare și partajarea stărilor.

Principalele trăsături ale acestui stil sunt:

- Reutilizarea. Componentele sunt deobicei proiectate pentru a fi reutilizate în diferite scenarii în aplicații diferite. Totuși, unele sunt proiectate pentru sarcini specifice.

- Substituția. Componentele poate fi subsituite cu componente similare.

- Nespecifice contextului. Componentele sunt proiectate pentru a funcționa în medii și contexte diferite. Informații specifice, cum ar fi datele de stare, ar trebui transmise componentei în loc sa fi incluse sau accesate de către aceasta.
- Extensibilitatea. O componente poate fi extinsă pornind de la componente existente pentru a oferi noi funcționalități.
- Encapsularea. Componentele expun diferinte interfețe care permit apelantului să-i acceseze funcționalitatea și care nu divulga detalii legate de procesele sau variabilele interne.
- Independența. Componentele sunt proiectate pentru a avea dependențe minime față de alte componente. Prin urmare acestea pot fi instalate în orice mediu fără a afecta alte componente sau sisteme.

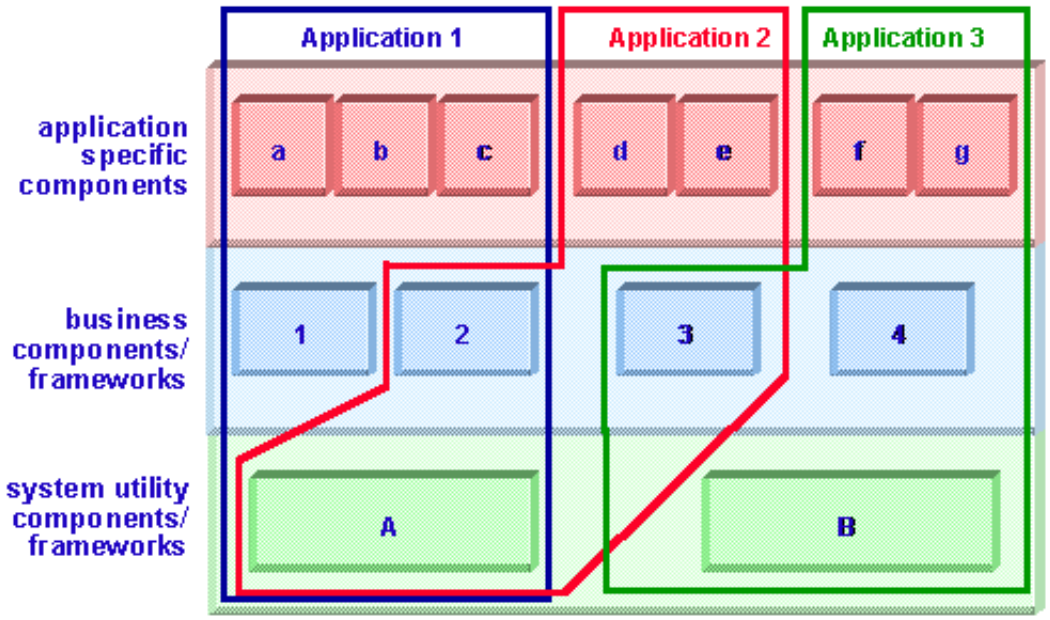

Figura 3.2: Arhitectură bazată pe componente

Tipuri comune de componente utilizate în aplicații includ componente grafice cum ar fi butoane și grile și componente ajutătoare sau utilitare care expun un set de funcții specifice folosite de către alte componente. Alte tipuri de componente sunt acelea care sunt strâns legate de resurse, nu așa de des accesate și care trebui activate folosind abordarea *just-in-time (JIT)* - des întâlnită în cadrul componentelor comandate la distanță ; și componente bazate pe cozi ale căror apeluri de metode pot fi executate asincron folosind mesaje de asteptare.

Componentele depind de un mecanism din cadrul platformei care oferă un mediu în care ele să poate fi executate. Exemplele includ : *component object model (COM)* și *distributed component object model (DCOM)* pentru Windows; și *Common Object Request Broker Architecture (CORBA)* și *Enterprise Java Beans (EJB)* pentru alte platforme. Aceste mecanisme administrează localizarea componentelor și a interfețelor acestora, transmiterea mesajelor și a comenzilor între componente și în unele cazuri menținerea stării.

Principalele avantaje ale acestui stil sunt:

- Ușurința instalării. Pe măsură ce noi versiuni devin disponibile, acestea pot înlocui versiunile existent fără a impacta vreo alta componentă sau sistemul ca un întreg.
- Cost redus. Utilizare componentelor terțe perminte reducerea costurilor de dezvoltare și administrare.

- Ușurința dezvoltării. Componentele implementează interfețe bine cunoscute pentru a oferit funcționalitate definită, permițând dezvoltarea fără a impacta alte parți ale sistemului.
- Reutilizarea. Folosirea componentelor reutilzabile înseamnă că acestea pot împarți costul de dezvoltare pe mai multe aplicații și sisteme.
- Atenuarea complexității tehnice. Componentele reduc complexitatea prin folosirea cadrului componentei și a serviciilor. Exmple de servicii includ activarea, gestionare timpului de viață, metode de așteptare, eveniment și tranzacții.

Tipare de proiectare cum ar fi Dependency Injection sau Service Locator pot fi utilizate pentru gestionare dependințelor între componente și pentru a încuraja slaba cuplare și reutilizarea. Aceste tipare sunt adesea folosite pentru a construi aplicații compozite ce combină și reutilizează componente în cadrul mai multor aplicații.

Se dorește folosirea acestei arhitecturii dacă: există deja componente adecvate sua dacă există acces la componente terțe; aplicația va conține predominant funcții procedurale, sau poate puține date; se folosesc mai multe limbaje de programare. Deasemenea, acest stil se pretează a fi folosit în cazul în care se dorește crearea unei arhitecturii modulare sau compozite care să permită schimbarea și actualizarea facila a componentelor individuale.

3.4 Arhitectura stratificată

Arhitectura straficată se concentrează pe gruparea funcționalităților conexe din cadrul unei aplicații pe niveluri diferite care sunt grupate orizontal unele deasupra celorlalte.Funcționalitate în cadrul fiecarui nivel este dominată de un rol sau o responsabilitate comună. Comunicarea între niveluri este explicită și slab corelată. Stratificarea corectă ajută la separarea responsabilității care sporește flexibilitate și ușurința administrării.

Acest stil a fost descris ca fiind o piramidă întoarsă în care fiecare nivel agregă responsabilitățile și abstractizările nivelului imediat inferior. Folosind stratificarea strictă, componentele din cadrul unui nivel pot interacționa doar cu componente din cadrul aceluiași nivel sau cu componente din nivelul imediat inferior. O stratificare mai lejeră permite componentelor dintr-un nivel să interacționeze cu componente din același nivel sau orice componente aparținând nivelurilor inferioare.

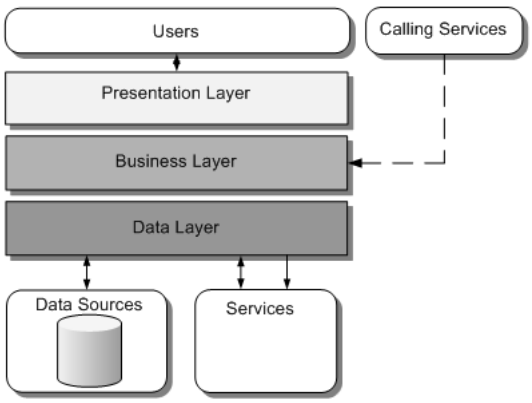

Figura 3.3: Arhitectură stratificată

Straturile aplicației pot fi aceeși mașină fizică (pe acelasi nivel) sau pot fi distribuite pe masini diferite (N-Nivele) și componentele din același nivel comunica cu componente din alte nivele prin interfețe bine definite. De exemplu o aplicație web tipică este alcătuită dintr-un

nivel de prezentare (tot ceea ce ține de interfața grafică), un nivel de logică business și un nivel de acces de date.

Principalele trăsături ale acestui stil includ:

- Abstractizarea. Arhitectura stratificată abstractizează imaginea de ansamblu a sistemului ca și întreg în vreme ce oferă destule detalii pentru a înțelege rolurile și responsabilitățile fiecărui nivel individual și relația între ele.
- Encapsularea. Nici o presupune nu trebuie facută cu privire la tipurile de date, metode și proprietăți sau implementare în timpul proiectării, deoarece acestea nu vor fi expuse public.
- Nivele funcționale clar definite. Separarea funcționalității pe fiecare nivel este clară. Nivele superioare cum ar fi cel de prezentare trimit comenzi celor inferioare, cum ar fi cele de business și acces de date și poate reacționa la evenimente din aceste nivele, permițând fluxuri de date în ambele sensuri.
- Nivel ridicat de coeziune. Limitele bine definite ale responsabilității fiecărui nivel și asigurarea faptului ca fiecare nivel conține funcționalitatea strict legată de sarcinile acelui nivel contribuie la maximizare nivelului de coeziune din acel nivel.
- Reutilizarea. Nivele inferioare nu depinde de cele superioare, permitând o posibilă reutilizare a acestora în cadrul altor scenarii.
- Cuplare slabă. Comunicarea între nivele este bazată pe abstractizare și evenimente pentru a oferi o cuplare slabă între acestea.

Exemple de aplicații stratificate includ aplicații de tip line-of-business (LOB) precum sisteme de contabilitate și administrarea clienților; aplicații web de tip enterprise și site-uri web și aplicații desktop sau agenți inteligenți cu un serer central pentru logica de business.

Tiparele de proiectare care susțin acest stil arhitectural sunt numeroase. De exemplu tiparele "Separated Presentation" conțin a varietate large de tipare pentru manipularea interacțiunilor cu interfața grafică, logica de prezentare și business și datele aplicației cu care utilizatorul lucrează. Acesta permite designerilor sa lucreze la interfața grafica în vreme ce dezvoltatorii lucrează la codul care va orchestra totul. Împărțind funcționalitatea astfel crește sansele de a test comportamentul individual al rolurilor. Principiile cheie ale acestui tipar sunt :

- Separare responsabilității. Acest tipar împarte procesare interfeței grafice în roluri distincte; de exemplu MVC are 3 roluri diferite: modelul, view-ul și controller-ul. Modelul reprezintă datele; View-ul reprezintă interfața grafică; Controller manipulează cererile, modelul și efectuează alte operații.
- Notificării bazate pe evenimente. Tiparul observator este folosit pentru a trimite notificări View-ului când datele administrate de model se schimbă.
- Precesare delegată a evenimentelor. Controller-ul procesează evenimente lansate din interfața grafică.

Principalele beneficii ale acestui stil și tipar sunt :

- Abstractizarea. Nivelele permit schimbări la nivel abstract. Se poate crește sau scădea nivelul de abstractizare al fiecărui nivel din stivă.

- Izolarea. Permite izolarea actualizărilor la nivele individuale pentru a reduce riscul și a minimiza impactul asupra sistemul ca și întreg.
- Administrarea. Separarea responsabilităților cheie ajută la identificarea depedențelor și organiează codul în blocuri mult mai gestionabile.
- Perfomanța. Distribuirea nivelelor pe mai multe mașini fizice poate crește scalabilitatea, fiabilitatea și performanța.
- Reutilizarea. Rolurile promovează reutilizarea. De exemplu, în MVC, Controller-ul poate adesea fii refolosit de către alte View-uri pentru a crea un role specific sau un view personalizat bazat pe aceeași funcționalitate și date.
- Testarea. Posibilitatea sporită de testare rezultă din interfețe bine definite, cât și din posibilitatea de a trece de la o implementare la alta a interfeței nivelului.Tiparele de tip "Separated Presentation" permit construirea obiectelor de test pentru a simula comportamentul obiectelor concrete cum ar fi modelul, controller-ul sau view-ul în timpul testării.

Acest stil este adecvat cazului în care există nivele care poate fi reutilizate în alte aplicații, dacă există deja aplicații care expun procese de business prin interfețe de serviciu sau dacă aplicația este complexă și design-ul de nivel înalt impune o separare astfel încât echipele să se poată concentra pe arii diferite. Acest stil este potrivit și în cazul în care aplicația trebuie să fie diposnibilă pe tipuri diferite de clienți sau dispozitive sau dacă se dorește implementarea unor reguli și procese business complexe și configurabile.

Tiparul prezentat mai sus sporește posibilitatea de testare a aplicației și simplifică mentenanța funcționalității interfeței grafice și oferă posibilitatea separării sarcinilor de concepere a interfeței grafice de dezvoltarea logicii de business.

3.5 Arhitectura bazată pe bus de mesaje

Arhitectura bazată pe bus de mesaje descrie principiul folosirii unui sistem software care primește și trimite mesaje folosind unul sau mai multe canale de comunicare, astfel încât aplicațiile pot interacționa fără a fi nevoite să cunoască detalii. Este un stil de proiectare a aplicațiilor a caror interacțiune se realizează prin transmitere de mesaje (de obice asincron) prin intermediul unui bus comun. Implementările tipice ale acestui stil folosesc fie un ruter de mesaje fie tiparul "Publish/Subscribe" și sunt adesea implementate folosind sisteme de mesagerie de tip "cozi de mesaje". Multe implementări sunt alcătuite din aplicații individuale care comunică folosind o schemă comună și o infrastructură partajată pentru primirea și trimiterea de mesaje. Un bus de mesaje oferă posibilitatea de a gestiona :

- Comunicații orientate pe mesaje. Toate comunicațiile între aplicații se bazează pe mesaje care folosesc scheme cunoscute.
- Logică complexă de procesare. Operațiile complexe poate fi executate folosind un set de operațiuni mai mici, fiecare realizează o sarcină specifică, ca parte a unui proces cu mai multe etape.
- Modificări ale logicii de procesare. Deoarece interacțiunea cu bus-ul se bazează pe scheme și comenzi comune, se pot insera și scoate aplicații din bus pentru a schimba logica care este folosită pentru a procesa mesajele.

- Integrarea cu diverse medii. Folosind comunicarea orientată pe mesaje bazate pe standarde comune se pot crea interacțiuni între medii diferite cum ar fii Microsoft .Net și Java.

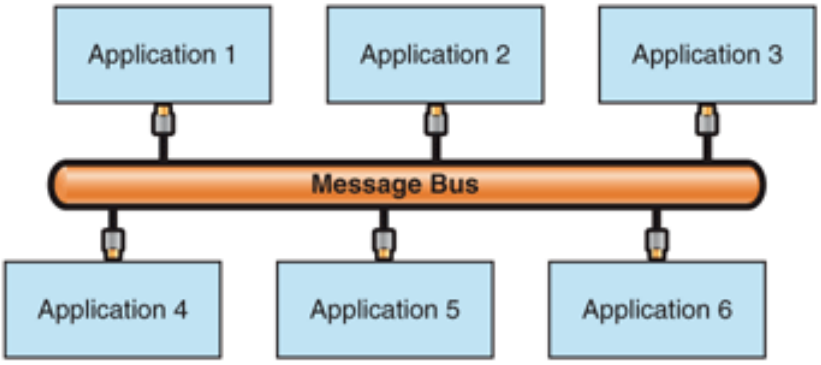

Figura 3.4: Arhitectură bazată pe bus de mesaje

Arhitecturile bazate pe bus de mesaje au fost folosite in cadrul procesărilor complexe. Acest design oferă o arhitectuă ce permite inserarea aplicațiilor în proces și îmbunătățește scalabilitatea prin posibilitatea de a atașa numeroase instanțe alea aceleași aplicații în bus. Variații ale acestui stil sunt:

- Enterprise Service Bus (ESB). Bazată pe design cu bus de mesaje, ESB folosește servicii pentru comunicarea între bus și componentele atașate la bus. De obicei oferă servicii pentru convertirea mesaje între diferite formate, permițând clienților să folosească mesaje incompotabilie pentru a comunica între ei.

- Internet Service Bus (ISB). Similar cu precentul doar că în acest caz aplicațiile se află în cloud și nu pe rețeaua companiei. Un concept esențial al ISB reprezintă folosirea URI-urilor (Uniform Resource Identifiers) și a politicilor pentru controlarea rutării informației din cadrul proceselor aplicațiilor și serviciilor din cloud.

Principalele avantaje ale acestui stil sunt:

- Extensibilitate. Aplicațiile pot fi adaugate sau înlăturate de pe bus fără a avea impact asupra aplicațiilor existente.

- Complexitate redusă. Complexitatea aplicațiilor este redusă, deoarece fiecare aplicație nu trebuie să cunoscă decât modul de interacțiune cu busul.

- Flexibilitate. Setul de aplicații care alcătuiesc un proces complex sau tiparele de comunicare între aplicații, pot fi ușor schimbate pentru a răspunde nevoilor de business, pur și simplu prin modificarea configurării sau a parametrilor care controlează rutarea.

- Cuplare slabă. Atât timp cât aplicațiile expun o interfață adecvată pentru comunicarea cu busul de mesaje, nu există nici o dependență față de aplicația în sine, aceasta permițând schimbări, actualizări și înlocuiri cu aplicații ce expun aceeași intefață.

- Scalabilitate. Mai multe instanțe ale aceeași aplicații pot fi atașate busului pentru a prelucra mai multe cereri în același timp.

- Simplitatea aplicației. Deși implementarea unui bus de mesaje adaugă complexitate infrastrucutrii, fiecare aplicație nu trebuie sa asigure decât o singură conexiune la bus în locul mai multora, către celălalte aplicații.

Acest stil arhitectural este potrivit cazurilor în care deja există aplicații care funcționează împreună pentru a realiza anumite funcții sau care combină mai multe sarcini într-o singură operație. Acest stil este adecvat dacă se dorește implementarea unei sarcini care necesită interacțiune cu aplicații externe sau aplicații găzduite pe medii diverse.

3.6 Arhitectura orientată pe obiecte

Arhitectura orientată pe obiecte este o pardigmă bazată pe divizarea responsabilităților dintr-o aplicație sau sistem pe obiecte individuale reutilizabile, fiecare conținând datele și comportamentul relevant obiectului. Un design orientat pe obiecte privește un sistem ca o serie de obiecte cooperând , în loc de un set de rutine sau instrucțiuni procedurale. Obiectele sunt discrete, independente și slab couplate; ele comunică prin interfețe, apelând metode sau accesând proprietăți ale obiectelor, sau trimițând si primind mesaje. Principiile cheie ale stilui arhitectural obiect orientat sunt următoarele:

- Abstractizare. Aceasta permite reducerea unei operațiuni complexe într-o generalizare ce reține caracteristicile de bază ale acesteia. De exemplu, o interfață abstractă poate fi o definiție bine cunoscută ce expune operațiuni legate de accesul la date folosind metode simple cum ar fi "get" sau "update". O altă formă de abstractizare ar putea fi metadatele folosite pentru a crea o mapare între două formate ce conțin date structurate.

- Compoziție. Obiectele pot asamblate pornind de la alte obiecte și pot ascunde obiectele interne din alte clase sau să le expună ca simple intefețe.

- Moștenire. Obiectele pot moșteni alte obiecte și folosi funcționalitatea din obiectul de bază sau sa o suprascrie pentru a implementa un nou comportament. În plus, moștenirea facilitează mentenanța și actualizările, deoarece schimbările în obiectul de bază propagă schimbările și în obiectele care îl moștenesc.

- Encapsularea. Obiectele expun funcționalitate numai prin metode, proprietăți și eveniment, dar ascund detalii interne cum ar fi starea sau variabile din alte obiecte. Aceasta ușureazp actualizarea și înlocuirea obiectelor, atât timp cât interfețele lor sunt compatibile, fără a afecta celălalte obiecte.

- Polimorfism. Aceasta permite suprascrierea comportamentului din tipul de bază care susține operații din cadrul aplicației implementând tipuri de noi care sunt interschimbabile cu obiectul existent.

- Decuplare. Obiectele pot fi decuplate de la consumator prin definirea unui interfețe abstracte pe care obiectele o pot implementa și pe care consumatorul o poate înțelege. Aceasta permite oferirea implementărilor alternative fără afectare consumatorilor interfeței.

Utilizări tipice ale acestui stil includ definirea unui obiect model care realizează operatii științifice comeplexe sau financiare, și a obiectelor care reprezintă artefacte din lumea reala dintr-un domeniu de business.

Beneficiile majore ale stilului orientat pe obiecte ar fi:

- Inteligibilitatea. Mapează aplicația la obiecte mult mai apropiate de lumea reală, ceea ce o face mai inteligibilă.

- Reutilizarea. Oferă posibilitatea reutilizării codului prin intermediul polimorfismului și a abstractizării.

- Testarea. Posibilități sporite de testare prin intermediul encapsulării.
- Extensibilitatea. Folosirea encapsulării, a polimorfismului și a abstractizării izolează interfețele pe care obiectul le expun de schimbările asupra datelor.
- Nivel înalt de coeziune. Punând în obiect doar metode și funcționalități corelate și folosind diferite obiecte pentru diferite funcții se obține un nivel înalt de coeziune.

Acest stil se potrivește în cazul în care se dorește modelarea unui aplicații bazatî pe obiecte și acțiuni din lumea reală, sau dacă există deja obiectele și acțiunile care să se potrivească cu cerințele de design și operaționale. El mai este adecvat și în cazul în cazre logica de business trebuie encapsulată alături de date în componente reutilizabile sau există logică de business complexă ce necesită abstractizare și comportament dinamic.

3.7 Arhitectura orientată pe servicii

Arhitectura orientată pe servicii crează posibilitatea expunerii funcționalității unei aplicații ca un set de servicii și crearea de aplicații care să folosească aceste servicii. Serviciile sunt slab cuplate deoarecere ele folosesc interfețe standard ce pot fi invocate, publicate și descoperite. Serviciile din cadrul SOA (Service Orientated Architecture) se concentrează pe oferirea unei interacțiuni bazate pe scheme și mesaje cu o aplicație prin interfețe care au scopul restrâns la aceasta și care nu sunt orientate pe componente sau obiecte. Un serviciu care respectă SOA nu ar trebuie sa fie considerat precum o componentă a unui furnizor de servicii.

Stilul SOA poate să includă procese de business în servicii interoperabile folosind o gamă largă de protocoale și formate de date pentru comunicareaa informațiilor. Clienții și alte servicii pot accesa servicii local rulând la același nivel, sau pot accesa serviciile la distanță prin intermediul rețelei.

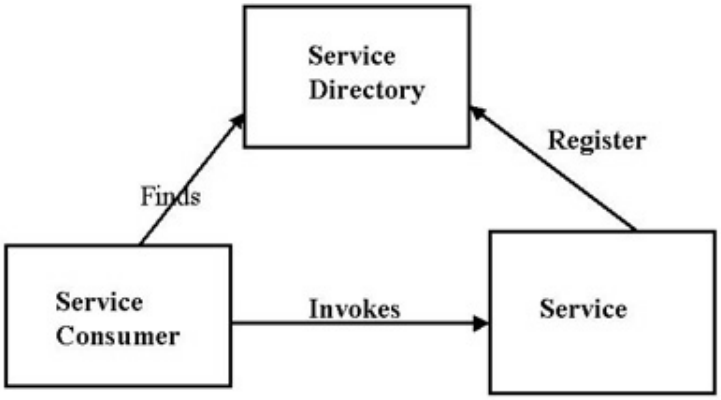

Figura 3.5: Arhitectură orientată pe servicii

Principiile cheie ale stilului SOA sunt:

- Serviciile sunt autonome. Fiecare serviciu este menținut, dezvoltat, instalat și versionat în mod independent.
- Serviciile pot fi distribuite. Serviciile pot fi localizate oriunde pe rețea, local sau la distanță, atât timp cat rețeaua dispune de protocoalele necesare de comunicare.

- Serviciile sunt slab cuplate. Fiecare serviciu este independent față de celălalte și poate fi înlocuit și actualizat fără a întrerupe celălalte aplicații care îl folosesc atât timp cât interfețele sunt compatibile.

- Serviciile împarte schema și contractul, nu clasa. Serviciile partajează contracte și scheme atunci când comunică, și nu clase interne.

- Compatibilitatea se bazează pe politici. Politici în acest caz înseamnă definirea caracteristicilor precum transport, protocol și securitate.

Exemple tipice de aplicații orientate pe servicii includ cele legate de partajarea informației, manipularea proceselor multi-etapă precum sisteme de rezervare sau magazine online, ce expun date sau servicii specifice industriei prin intermediul unei rețele, creând colecții de informații din mai multe surse.

Beneficiile principale ale stilului SOA sunt:

- Alinierea la domeniu. Reutilizarea serviciilor comune cu interfețe standard crește oportunitățile de business și tehnologice și reduce costurile.

- Abstractizarea. Serviciile sunt autonome și accesate printr-un contract formal, ce oferă cuplare slabă și abstactizare.

- Detectabilitatea. Serviciile pot expune descrieri care să permită altor aplicații și servicii să le localizeze și să construiască automat interfața.

- Interoperabilitatea. Deoarece protocoalele și formatele datelor sunt baze pe standarde din industrie, furnizorul și consumatorul servicilui pot fi construiți și instalați pe platforme diferite.

- Raționalizarea. Serviciile pot fi granulare pentru a oferi o funcționalitate specifică, mai degrabă decât să duplice funcționalitatea în cadrul mai multor aplicații.

Acest stil merită luat în considerare atunci când este diposnibil accesul la serviciile care se doresc a fi reutilizate; când serviciile pot fi cumpărate de la o companii terțe; când se dorește construirea aplicațiilor compuse dintr-o serie de servicii cu o singură interfață grafică; sau când se dorește crearea unei aplicații de tipul Software plus Services (S+S), Software as a Service (SaaS), sau bazate pe cloud. Stilul SOA este adecvat atunci trebuie asigurată comunicarea între segmentele unei aplicații și expuse funcționalități într-o manieră independentă de platformă, când se dorește utilizarea serviciilor centralizate precum autentificare sau expunerea serviciilor detectabile prin registre sau care pot fi utilizate de clienți care nu cunosc în prealabil interfețele.