Universitatea POLITEHNICA din București
Facultatea de Electronică, Telecomunicații și Tehnologia Informației

Arhitecturi software orientate pe servicii

Lucrare de dizertație

Prezentată ca cerință parțială pentru obținerea
titlului de *Master*
în domeniul *Electronică, Telecomunicații și Tehnologia Informației*
programul de studii *Tehnologii Software Avansate pentru Comunicatii*

Conducător științific
Eduard Popivici

Absolvent
Andrei Mihaescu

Anul 2016

Declaraţie de onestitate academică

Prin prezenta declar că lucrarea cu titlul *Arhitecturi software orientate pe servicii*, prezentată în cadrul Facultăţii de Electronică, Telecomunicaţii şi Tehnologia Informaţiei a Universităţii "Politehnica" din Bucureşti ca cerinţă parţială pentru obţinerea titlului de *Master* în domeniul Inginerie Electronică şi Telecomunicaţii/ Calculatoare şi Tehnologia Informaţiei, programul de studii *Tehnologii Software Avansate pentru Comunicatii* este scrisă de mine şi nu a mai fost prezentată niciodată la o facultate sau instituţie de învăţământ superior din ţară sau străinătate. Declar că toate sursele utilizate, inclusiv cele de pe Internet, sunt indicate în lucrare, ca referinţe bibliografice. Fragmentele de text din alte surse, reproduse exact, chiar şi în traducere proprie din altă limbă, sunt scrise între ghilimele şi fac referinţă la sursă. Reformularea în cuvinte proprii a textelor scrise de către alţi autori face referinţă la sursă. Înţeleg că plagiatul constituie infracţiune şi se sancţionează conform legilor în vigoare. Declar că toate rezultatele simulărilor, experimentelor şi măsurătorilor pe care le prezint ca fiind făcute de mine, precum şi metodele prin care au fost obţinute, sunt reale şi provin din respectivele simulări, experimente şi măsurători. Înţeleg că falsificarea datelor şi rezultatelor constituie fraudă şi se sancţionează conform regulamentelor în vigoare.

Bucureşti, Iunie 2016.
Absolvent: Andrei Mihaescu

.....

Cuprins

Lista figurilor

Lista tabelelor

Lista acronimelor

BDFU - Big Design Upfront DRY - Don't repeat yourself
SOA - Service Oriented Architecture
QA - Quality Assurance
YAGNI - You ain't gonna need it
UML - Unified Modelling Language

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.

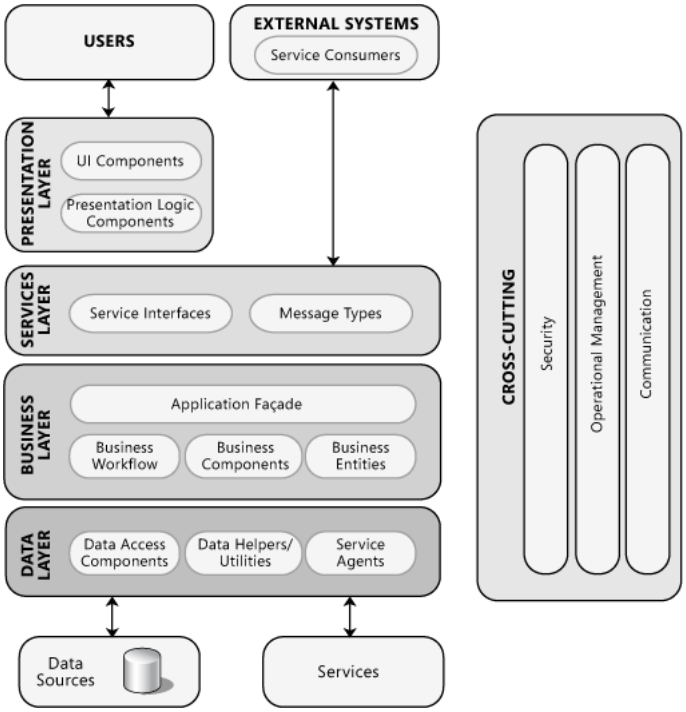

Figura 2.1: Organizarea pe "arii de responsabilitate"

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

3.2 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

3.3 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.