Arquitectura del programari

Nathalie Moreno Vergara
Antonio Vallecillo Moreno
José Raúl Romero Salguero
Francisco Javier Durán Muñoz

P06/11059/01148

UOC
Universitat Oberta de Catalunya

Índex

Introducció

Tal com s'ha comentat en el mòdul anterior, un aspecte fonamental a l'hora de dissenyar i construir programari és definir-ne l'**arquitectura**. Aquesta arquitectura defineix l'estructura i la funcionalitat de l'aplicació en termes de components i connectors, independentment dels llenguatges de programació i les tecnologies que s'utilitzin finalment per a implementar-la.

L'objectiu principal de l'arquitectura del programari és aportar elements que ajudin a la presa de decisions, al mateix temps que es proporciona un marc comú que permetrà la comunicació entre els diferents equips que participen en un projecte. Per a aconseguir-ho, l'arquitectura programari construeix abstraccions i les materialitza-les en forma de models que es representen en diagrames.

Aquest mòdul introdueix el concepte d'arquitectura programari i justifica la necessitat de documentar aquest aspecte tan rellevant dins del disseny i la implementació d'un sistema programari. Seguidament, es descriuen els **estils arquitectònics** més coneguts i aplicats en l'actualitat com a mecanismes d'abstracció per a estructurar sistemes, i es dedica atenció especial als estils més indicats en el modelatge de sistemes oberts i distribuïts.

Finalment, es descriu amb deteniment el procés de disseny arquitectònic. A tall d'exemple, i usant UML com a llenguatge de descripció d'arquitectures, abordarem el disseny lògic per al cas particular d'arquitectures en tres capes.

Objectius

En aquest mòdul l'alumne aprendrà a fer el disseny arquitectònic d'un sistema programari mitjançant UML. Aquest objectiu es descompon en els subobjectius següents:

1. Entendre el rol del disseny arquitectònic i la seva rellevància en el procés de desenvolupament programari.

2. Entendre la importància de reutilitzar l'experiència i les solucions arquitectòniques existents per a problemes similars, com a mecanisme per a abordar un nou disseny.

3. Saber escollir un patró arquitectònic d'acord amb una col·lecció de requisits crítics funcionals, de qualitat i rendiment.

4. Aprendre a elaborar i documentar el disseny lògic d'una arquitectura de programari.

1. Arquitectures de programari

1.1. Preliminars

Tots hem observat alguna vegada la construcció d'un edifici. En una primera fase, s'aixequen els fonaments, després les columnes i bigues, les diferents plantes, fins a tenir un esquelet de la finca. Després es construeixen terres, parets, portes i finestres, instal·lacions elèctriques i canonades, etc. En definitiva, primer es crea l'estructura que dóna suport a l'edifici i després s'acoblen les diferents parts que aporten les funcionalitats bàsiques de l'immoble. L'estructura té una importància especial, ja que els errors comesos durant la primera etapa podrien degenerar en un edifici no habitable, a més de no ser fàcils de corregir. Al contrari, errors no estructurals com l'absència d'alguna porta o una instal·lació defectuosa de les finestres serien problemes que, encara que importants, podrien ser relativament fàcils de solucionar.

Aquesta manera de procedir és aplicable també a la creació de programari. A diferència de la construcció d'un edifici, el programari no es regeix per lleis físiques ni per procediments coneguts, sinó que és inherentment experimental, creatiu i específic per a un problema concret. Des d'un punt de vista abstracte, **l'arquitectura del programari** –com es coneix l'esquelet o estructura del sistema– es defineix a partir d'un conjunt de requisits funcionals crítics, de rendiment o de qualitat. Analitzant com el programari ha de donar solució a aquests objectius, l'arquitectura construeix el conjunt d'estructures, classes i atributs principals del programari i les seves interfícies de comunicació. Des d'un punt de vista més tangible, l'arquitectura es materialitza en el conjunt de components que implementen aquest esquelet, la qual cosa permet avaluar els requisits fixats inicialment.

L'arquitectura programari també proporciona elements que ajuden a la presa de decisions, al mateix temps que serveix de marc comú que possibilita la comunicació entre els equips que participen en un projecte. Per a aconseguir-ho, l'arquitectura del programari construeix abstraccions, i les materialitza en forma de **models** que es representen en diferents tipus de **diagrames**.

1.2. Objectius de l'arquitectura de programari

Sense pretendre establir una definició completa i definitiva de moment, en un sentit ampli podríem estar d'acord que l'arquitectura és el disseny de més alt nivell de l'estructura d'un sistema, programa o aplicació, i que té les responsabilitats següents:

1) Identificar els **mòduls principals** del sistema, i posposar els detalls d'implementació de cada un en fases posteriors en el disseny.

2) Identificar la **funcionalitat** i les **responsabilitats** que tindrà cada un d'aquests mòduls.

3) Definir les **interaccions** possibles entre els esmentats mòduls; utilitzant per a això mecanismes de control i flux de dades, seqüenciació de la informació, protocols d'interacció i comunicació, etc.

Però quina és la motivació que provoca que sorgeixi l'interès per abordar de manera explícita els aspectes arquitectònics d'un sistema? En aquest sentit, el disseny arquitectònic és important per diverses raons:

a) La descripció de l'arquitectura fa més senzilla la **comprensió de sistemes complexos**, en explicitar les **decisions** de disseny d'alt nivell sobre el sistema programari.

b) Les descripcions arquitectòniques **poden ser reutilitzades**. Així, mentre la reutilització normalment es limita a l'ús de biblioteques de funcions o classes, el disseny arquitectònic potencia la reutilització de grans components programari i, el que és més important, de l'arquitectura de l'aplicació. D'aquesta manera, poden crear-se i catalogar-se patrons arquitectònics d'ús comú. Aquests patrons faciliten el procés de disseny en proporcionar solucions comunes a cert tipus de problemes.

c) Finalment, la descripció arquitectònica **permet al dissenyador raonar** sobre la pròpia estructura del sistema, les seves propietats, etc.

En general, la representació d'una arquitectura programari es fa en termes d'una col·lecció de components i de les interaccions que tenen lloc entre aquests. S'ha de prestar, doncs, atenció especial no solament al disseny dels components, sinó també a com aquests s'organitzen entre si i a la manera com es connecten amb els altres per a dur a terme una tasca determinada. Però vegem quina és la definició d'arquitectura programari que ofereix la comunitat científica.

1.3. Definició d'arquitectura de programari

Malgrat l'interès creixent que els aspectes arquitectònics del desenvolupament programari estan cobrant els últims anys –tant des de la comunitat científica com des de la mateixa indústria–, no hi ha encara un consens clar respecte a què s'entén per l'arquitectura d'un sistema programari.

Llista de definicions

El Software Engineering Institute (SEI) de la Universitat Carnegie Mellon ofereix una recopilació molt completa i interessant de definicions d'arquitectura programari en http://www.sei.cmu.edu/ architecture/definitions.html.

Examinant algunes recopilacions –com la presentada pel Software Engineering Institute (SEI)–, comprovem que, en general, moltes definicions barregen concepcions diferents que resulten de vegades contraposades. Davant aquesta diversitat, dues són les definicions més reconegudes en la literatura científica.

Arquitectura programari (Garlan i Perry, 1995): estructura dels components d'un programa o sistema, les seves interrelacions i els principis i les regles que governen el seu disseny i la seva evolució en el temps.

Arquitectura programari (Bass, Clements i Kazman, 2003): estructura o estructures d'un sistema, cosa que inclou els seus components programari, les propietats observables d'aquests components i les relacions entre aquests.

Encara que no exactament iguals, totes dues definicions suggereixen que:

Concepte de component

En aquestes definicions el concepte de component és genèric i en l'àmbit arquitectònic, i no es refereix en particular a cap dels "components programari" que defineixen els models de components com CORBA Component Model, Java EE (JavaBeans o EJB), ActiveX, COM, COM+ o .NET.

- **L'arquitectura programari d'un sistema defineix elements**: components i connectors que es vinculen en el procés de disseny. L'arquitectura expressa informació sobre la manera com es relacionen els elements entre si i omet, intencionadament, altres aspectes no relacionats amb l'estructura i la funcionalitat detallada de cada un.

- **Queda implícit que tot sistema té una arquitectura**, ja que pot modelar-se com una composició d'elements i les relacions entre aquests.

  En el cas trivial, el sistema constarà d'un únic element, fet que donarà com a resultat una arquitectura poc interessant i probablement poc útil, però una arquitectura a la fi. D'altra banda, que tot sistema tingui una arquitectura no implica necessàriament que aquesta sigui coneguda, fet que demostra la importància de documentar l'arquitectura.

- **Els detalls d'implementació interna de cada element no són rellevants en aquest nivell**. El comportament d'un element forma part de la descripció de l'arquitectura sempre que aquest comportament sigui interessant des del punt de vista global i pugui ser observat o percebut des d'un altre element.

D'altra banda, si examinem amb més detall ambdues definicions, hi trobarem algunes discrepàncies:

- Mentre Garlan i Perry afirmen que cada sistema té una única arquitectura, Bass, Clements i Kazman consideren diverses representacions possibles, atenent a punts de vista diferents. Així, un sistema podria comprendre més d'una estructura, i cap estructura en particular no podria atribuir-se el privilegi que fos anomenada "l'arquitectura".

- Per a Garlan i Perry l'arquitectura no solament conté informació sobre la seva estructura, sinó també sobre els motius i les decisions que s'han pres

a l'hora de construir-la, i que seran clau a l'hora de mantenir el sistema, actualitzar-lo o fer-hi modificacions: "(...) els principis i les regles que en governen el disseny i l'evolució en el temps".

Tenint en compte els aspectes prèviament esmentats, podem definir l'arquitectura programari d'un sistema de la manera següent:

L'**arquitectura programari**, és el conjunt de decisions, principis i regles que regeixen l'organització d'un sistema programari; la selecció dels elements estructurals que componen el sistema, les seves interfícies i els seus protocols d'interacció; les connexions d'aquests elements estructurals per a formar subsistemes de dimensions cada vegada més grans; i l'estil o patró arquitectònic que guia aquesta organització.

Una arquitectura programari serà descrita per una sèrie de models, que són els que componen la descripció arquitectònica del sistema. En el nostre cas utilitzarem UML per a descriure aquests models.

Vegeu també

Veurem amb més detall els models arquitectònics en l'apartat "Representació de l'arquitectura programari" d'aquest mòdul.

1.4. Components i connectors

Les descripcions arquitectòniques són descrites en termes de components i connectors.

Exemples de components:

- clients,
- servidors,
- filtres,
- capes,
- bases de dades, etc.

En el context de l'arquitectura programari, un **component** és una unitat abstracta que encapsula un estat i una funcionalitat, i que interacciona amb el seu entorn a través d'interfícies ben definides.

En altres paraules, un component és definit per les interfícies que defineixen els serveis que proporciona a altres components, sense entrar en detalls ni consideracions sobre com s'implementen aquests serveis.

En el context de l'arquitectura programari, un **connector** és un mecanisme abstracte que fa de mitjancer en la comunicació, coordinació o cooperació entre components.

Els connectors constitueixen els enllaços que acoblen els components i en defineixen l'organització i les connexions. Els connectors poden definir tant els mecanismes i protocols d'interacció entre els components, com alguns requisits de qualitat (temps i capacitat de transmissió, taxes de transferència, taxes d'errors, seguretat, etc.).

Exemples de connectors simples:

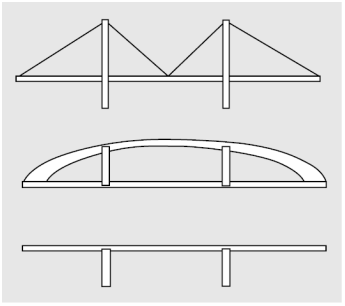
- les crides a procediments,
- el pas de missatges,
- els protocols de comunicació,
- les canonades, etc.

Internament, un connector pot ser una cosa tan simple com una crida a un procediment, o consistir en un nou sistema de components tan complex com el requereixi l'aplicació modelada. Però des del punt de vista arquitectònic, aquesta complexitat romandrà oculta i només serà visible quan s'especifiquin els detalls interns dels components i connectors del sistema en un nivell d'abstracció més baix.

Observen que la descripció de l'arquitectura programari d'un sistema es correspon amb la seva especificació des del punt de vista computacional d'RM-ODP, on els components es corresponen fidelment amb els objectes computacionals i els connectors, amb els objectes d'enllaç.

Encara que, òbviament, cada dissenyador pot definir els components i els connectors de l'arquitectura programari d'un sistema com millor cregui convenient, sempre sol ser millor (menys errors, més eficient, etc.) fer-hi servir algun dels estils arquitectònics existents. En aquest cas, els tipus de components i connectors són determinats pel model o estil arquitectònic seleccionat per al disseny. Per tant, és important conèixer quins són els estils arquitectònics existents i les seves aplicacions, així com els avantatges i inconvenients que presenten. Les seccions següents estan destinades a tractar aquests aspectes.

1.5. Estils arquitectònics



Des dels inicis de l'arquitectura del programari, es va observar que certes regularitats en l'estructura, l'estil i els elements utilitzats apareixien una i altra vegada com a resposta a demandes similars. Per analogia amb l'ús del terme en arquitectura civil, aquests patrons aviat es van anomenar *estils arquitectònics*.

Els **estils arquitectònics** representen maneres diferents d'estructurar un sistema mitjançant components i connectors, d'acord amb decisions essencials sobre els elements arquitectònics i establint restriccions importants sobre aquests elements i les seves relacions possibles.

A l'hora de definir un estil arquitectònic, el camp d'aplicació no és allò principal; és més important tenir en compte el patró d'organització general, els tipus de components habitualment presents en l'estil i les interaccions que s'estableixen entre aquests.

Un **estil arquitectònic** defineix una família de sistemes, en termes d'un conjunt de patrons estructurals.

A més, un estil arquitectònic indicarà quins són els patrons i les restriccions d'interconnexió o composició entre aquests, la qual cosa es denomina **invariants de l'estil**. Aquests invariants són els que diferencien uns estils arquitectònics dels altres.

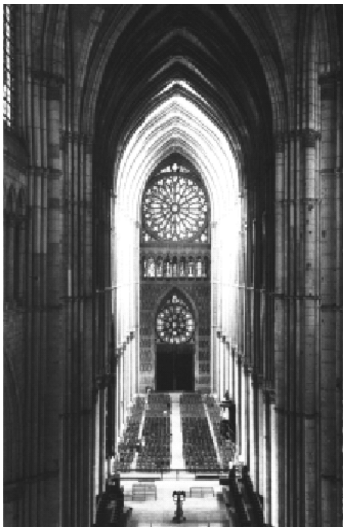
Cada estil caracteritza una **estructura general d'alt nivell**, al costat de **microarquitectures** que contribueixen a l'arquitectura global.

La catedral de Reims

La catedral de Reims (segle XIII, mostrada al marge) pertany a l'estil gòtic, un dels nombrosos estils arquitectònics utilitzats a les catedrals de tot Europa.

L'estil estableix les principals regles que han de guiar la construcció de la catedral al llarg del seu desenvolupament (vuit generacions de constructors hi van treballar), i també determina la manera com pot evolucionar posteriorment. Per a fer això, l'estil caracteritza una estructura general d'alt nivell, al costat de microarquitectures que contribueixen a l'arquitectura global.

En el cas concret del gòtic, els elements essencialment característics d'aquest estil arquitectònic són la volta de creueria i l'arc apuntat; la primera afecta l'estructura i l'altre, més particularment, les formes exteriors. A més, la planta de les catedrals gòtiques són en forma de creu, de tres o cinc naus, amb creuer més curt que el de l'arquitectura romànica, amb la girola o deambulatori a la capçalera i en la qual s'obren les capelles poligonals, totes tancades en un gran mig cercle. També destaquen altes naus laterals en les quals s'obren amplis finestrals i robustos contraforts que envolten el perímetre de la catedral, coronats per pinacles. Totes aquestes característiques són les que constitueixen els **invariants de l'estil** i que permeten distingir clarament una construcció que segueix aquest estil d'una altra.



Finalment, els invariants de cada estil dotaran un sistema d'una sèrie de propietats que en determinaran els avantatges i els inconvenients, i que condicionaran a tria d'un estil o un altre a l'hora de resoldre un problema determinat.

En el cas de les arquitectures del programari, els invariants d'estil seran donats pel conjunt de **regles** i **restriccions** que prescriuen:

Exemples d'estils arquitectònics

El projecte ABLE de la Carnegie Mellor University (CMU) té treballs i exemples molt útils sobre estils arquitectònics de programari. Es pot consultar en http://www.cs.cmu.edu/~able.

1) Quins tipus de components (elements estructurals), interfícies, connectors i patrons poden ser usats en un sistema (possiblement introduint-hi tipus específics per a cada domini d'aplicació concret).

2) Com els components i connectors poden ser combinats (restriccions d'interconnexió o composició entre aquests).

3) Com es comporta el sistema i com certes propietats poden ser determinades depenent de les propietats dels seus components. Cada estil duu associats uns mecanismes d'interacció determinats, que impliquen com es transfereix el control entre els components i com aquests col·laboren per implementar la funcionalitat del sistema.

Un altre aspecte destacable també en qualsevol disseny arquitectònic d'un sistema és el de la seva **integritat conceptual**, un principi pel qual el patró general de disseny d'un sistema es reflecteix en cada part d'aquest. Això implica que les restriccions que defineixen els invariants d'estil han de ser coherents entre si i respectar una certa harmonia.

En el cas de la catedral de Reims, la seva integritat conceptual es deu a l'ús intensiu de l'estil gòtic en tots i cada un dels seus elements arquitectònics, des de l'estructura global als elements interns.

El que opinen els experts

"Conceptual integrity is the most important consideration in system design" (Brooks, *The Mythical Man Month*, 1975)

"The same set of laws determines the structure of a city; a building; or a single room" (Alexander, *A Pattern Language*, 1977)

En general, la integritat conceptual no obliga que hi hagi un únic estil arquitectònic pur, sinó que és possible conjugar-ne més d'un en un mateix sistema. Ara bé, la manera de conjugar els estils no pot ser heterogènia i capritxosa, sinó que ha d'estar subjecta a uns d'invariants de l'estil –per a garantir la integritat conceptual–, que són els que determinen com i quan es poden barrejar els patrons estructurals corresponents.

Per exemple, hi ha molts edificis amb barreges d'estils arquitectònics, però que conjuguen perfectament aquestes barreges de manera harmoniosa (és a dir, respectant-ne la integritat conceptual). Igualment, molts sistemes de programari conjuguen diferents estils arquitectònics, però és molt important determinar també els invariants d'estil que regiran aquestes combinacions. Un exemple molt clar ocorre amb els sistemes web, dissenyats usant un estil client-servidor entre components independents, però que a més s'organitzen en tres capes o nivells. Ara bé, l'assignació de components en capes (que constituiria aquesta barreja d'estils) no es fa de manera capritxosa, sinó que és determinada per uns criteris clars quant a la localització física, el rendiment, l'eficiència i la funcionalitat específica de cada un dels components.

1.6. Classificació dels estils arquitectònics

Hi ha nombrosos estils arquitectònics definits per a sistemes programari que, en general, poden classificar-se com indica la figura 1.

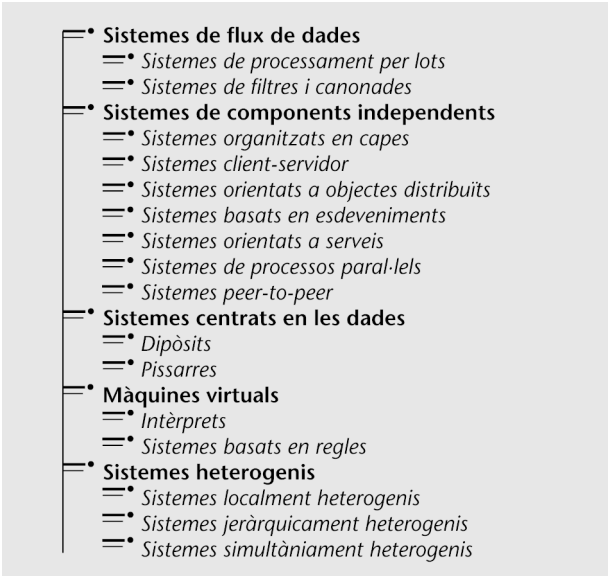


Figura 1. Classificació dels principals estils arquitectònics

Lectures complementàries

Per a veure amb més deteniment els detalls presentats en aquesta classificació i altres de semblants, poden consultar-se les referències següents:

L. Bass; P. Clements; R. Kazman (1998). *Software Architecture in Practice*. Reading, MA: Addison-Wesley.

D. Garlan; M. Shaw (1994). *An introduction to software architecture. Advances in Software Engineering and Knowledge Engineering* (vol. 1, pàg. 1-40). Hackensack, NJ, EUA: World Scientific Publishing Company.

A continuació presentarem molt breument cada una d'aquestes classificacions, i passarem a descriure els estils arquitectònics que són especialment rellevants per als sistemes distribuïts basats en components.

1.6.1. Arquitectures de flux de dades

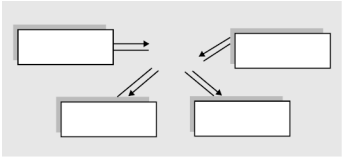
Aquest tipus d'arquitectures són adequades per a aplicacions que s'organitzen en termes de dades que flueixen entre unitats de processament. Cada unitat és dissenyada **independentment** de les altres. Així, les dades tenen una **font** (origen) i acaben en un **clavegueró** (destinació).

Per exemple, les aplicacions que es desenvolupen mitjançant qualsevol de les *shell* dels sistemes Unix són potser els sistemes més representatius de l'estil arquitectònic basat en filtres i canonades. Aquestes aplicacions es construeixen a partir de programes que actuen com a filtres (cada un rep unes dades d'entrada, els transforma i els envia pel seu canal de sortida), que es connecten entre si per canonades (*pipes*). Aquest estil arquitectònic es discutirà amb més detall en el subapartat 1.8.1.: "Sistemes de flux de dades".

Els sistemes que segueixen un estil arquitectònic basat en flux de dades se solen dissenyar al voltant de les dades que manegen; els components actuen com a mers "transformadores" d'aquestes dades. En aquest sentit, les dades cobren més rellevància que el processament.

1.6.2. Arquitectures de components independents

Els models arquitectònics basats en components independents afavoreixen la distribució tant de les **dades** com del **processament**, ja que tots dos estan "encapsulats" en unitats **independents** (els components) que interactuen entre si per aconseguir el seu objectiu. Normalment, l'accés a l'estat que emmagatzema cada component no s'efectua de manera directa, sinó a través d'invocacions a les seves operacions.



Els sistemes programari que segueixen aquests estils arquitectònics consisteixen, en general, en processos o entitats independents que operen (en principi) en paral·lel i es comuniquen a través de missatges. Cada entitat pot enviar missatges a altres entitats, però no accedir a l'estat intern d'altres entitats directament; i per això l'acoblament entre els components és mínim. L'intercanvi

de dades es du a terme mitjançant pas de missatges a components nominats –habitual en els sistemes basats en comunicació entre processos– o propagats mitjançant difusió (en anglès, *broadcast*), típic dels sistemes d'esdeveniments.

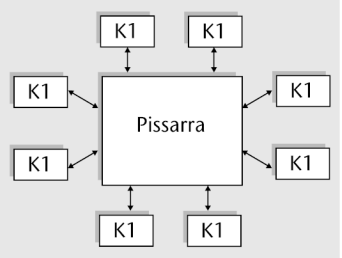
A diferència dels estils arquitectònics basats en fluxos de dades, que se centren en les dades que tracta el sistema i en la manera com aquestes són transformats successivament pels programes, els estils basats en components individuals es concentren més en el processament, és a dir, en la funcionalitat del sistema i en la seva organització en unitats independents. Una característica molt important d'aquests estils arquitectònics és que afavoreixen la reutilització dels seus components i són molt flexibles pel que fa a la seva distribució en màquines diferents.

Vegeu també

Són membres d'aquesta família els estils basats en capes, client-servidor, objectes, esdeveniments o serveis. Aquests subestils seran tractats, respectivament, en els subapartats 1.8.

1.6.3. Arquitectures basades en repositori

En el centre d'aquesta arquitectura hi ha un **magatzem de dades** (per exemple, un document, una "pissarra", una "sopa" d'objectes, o una base de dades) al qual altres components accedeixen sovint per afegir, consultar, esborrar o bé modificar les dades del magatzem.



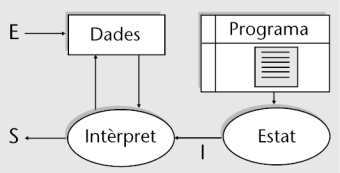
En aquest estil, la comunicació entre els components mai no és directa, sinó que es fa sempre a través del dipòsit (si un component vol enviar alguna cosa a un altre, el diposita al dipòsit i aquest altre ja el recollirà quan pugui). Aquest mode d'interacció indirecta és molt apropiat per a sistemes que requereixen un desacoblament molt elevat entre els seus components, que no necessiten conèixer-se per a interactuar. Les pissarres fins i tot permeten que la comunicació sigui anònima, ja que els components dipositen les dades sense saber qui les consumirà, mentre que els consumidors busquen les seves dades utilitzant-hi "patrons" de cerca.

A més, és fàcil distribuir els seus components. Tanmateix, aquests sistemes poden plantejar alguns problemes a l'hora de distribuir el dipòsit en diferents màquines, ja que es tracta d'una estructura que des d'un punt de vista conceptual és centralitzada clarament.

D'altra banda, el dipòsit o magatzem de dades pot ser passiu o actiu.

- Es diu que un dipòsit és **passiu** quan els components deixen o recullen informació al dipòsit, que actua com un mer magatzem de dades.
- Al contrari, un dipòsit **actiu** envia notificacions als components clients quan canvien les dades que els interessen.

1.6.4. Arquitectures de màquina virtual



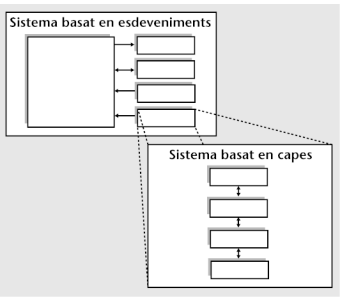
Les aplicacions basades en aquest estil simulen funcionalitats no natives del maquinari i programari en les quals s'implementen, o també capacitats que excedeixen (o que no coincideixen amb) les capacitats del paradigma de programació que s'està implementant. La funcionalitat que es vol s'aconsegueix mitjançant una màquina d'interpretació que inclou tant la definició de l'intèrpret, com l'estat actual de l'execució. L'estil comprèn bàsicament dues formes o subestils, que s'han anomenat *intèrprets* i *sistemes basats en regles*.

1.7. Arquitectures heterogènies

Nombrosos sistemes presenten una arquitectura que pot considerar-se com a pertanyent marginalment a un estil o a un altre, o bé que combina clarament característiques de diversos estils. És normal que una mateixa aplicació usi diferents arquitectures per a resoldre diferents aspectes del mateix problema. Direm llavors que aquests sistemes tenen un estil heterogeni. Aquesta heterogeneïtat pot ser de tres tipus:

- Els sistemes **localment heterogenis** tenen una arquitectura que segueix principalment un estil determinat, però que en unes parts determinades presenta patrons d'altres estils diferents. Per exemple, algunes parts d'un sistema estructurat en objectes distribuïts poden seguir un estil de dipòsit.

- En els sistemes **jeràrquicament heterogenis**, l'arquitectura interna dels components d'un sistema d'acord amb un cert estil segueixen estils arquitectònics diferents. Així, per exemple, cada un dels integrants de les capes d'un sistema estructurat en capes pot seguir un estil arquitectònic basat en components independents.



- Finalment, en els sistemes **simultàniament heterogenis** diversos estils poden aplicar-se per a descriure'n l'arquitectura, depenent del punt de vista, o de quins components considerem més importants. Així, per exemple, molts sistemes de processament per lots poden considerar-se tant sistemes de flux de dades, com sistemes centrats en les dades.

Aquesta heterogeneïtat és fruit, en molts casos, de les similituds existents entre alguns estils arquitectònics. Per exemple, el lector trobarà semblances importants entre:

- Les màquines virtuals i les arquitectures multicapa.
- Els repositori de dades i les arquitectures client-servidor.
- Les canonades i els filtres i els processos paral·lels.

Recordem que, tal com s'havia esmentat en l'apartat d'estils arquitectònics, la manera de conjugar els estils no pot ser capritxosa, sinó que ha d'estar subjecta a uns invariants de l'estil que garanteixin la **integritat conceptual**.

En els subapartats següents estudiarem amb més detall aquells estils arquitectònics rellevants des del punt de vista dels sistemes oberts i distribuïts.

1.8. Principals estils arquitectònics

1.8.1. Sistemes de flux de dades

En els **sistemes de flux de dades**, les dades flueixen d'un lloc a l'altre i es transformen durant aquest moviment. Cada pas s'implementa com una transformació: les dades d'entrada flueixen a través d'aquestes transformacions fins que es converteixen en sortides. Les transformacions es poden executar seqüencialment o en paral·lel i les dades es poden processar, per a cada transformació, element per element o en lots.

Algunes vegades en què les transformacions es presenten com processos separats, aquest model s'anomena **el model de canonades i filtres**, amb referència a la terminologia utilitzada en els sistemes Unix (vegeu la figura 2), on s'utilitza el terme *filtre* perquè una transformació "filtra" les dades que processa dels fluxos de dades d'entrada.

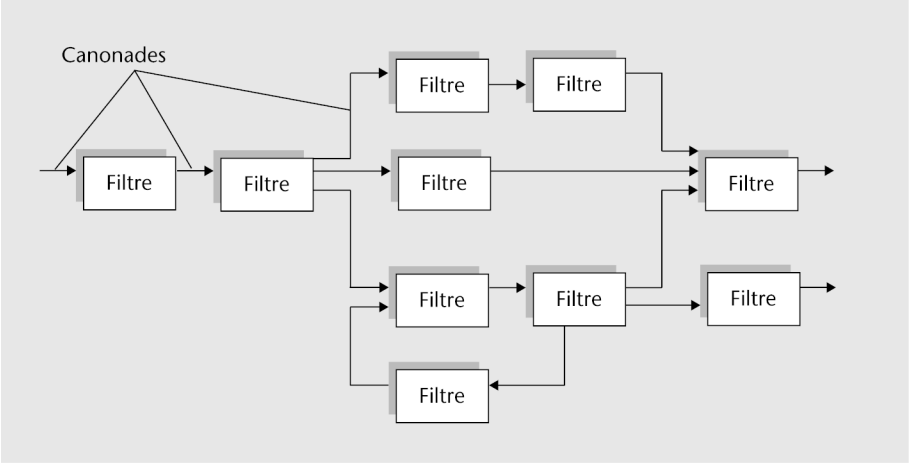


Figura 2. Canonades i filtres

Quan les transformacions són seqüencials, amb les dades processades per lots, aquest model arquitectònic rep el nom de **model seqüencial per lots** (vegeu la figura 3). Aquesta arquitectura és comuna per a algunes classes de sistemes de processament de dades com els sistemes de facturació, que generen gran nombre d'informes de sortida a partir de càlculs senzills sobre diversos registres d'entrada.

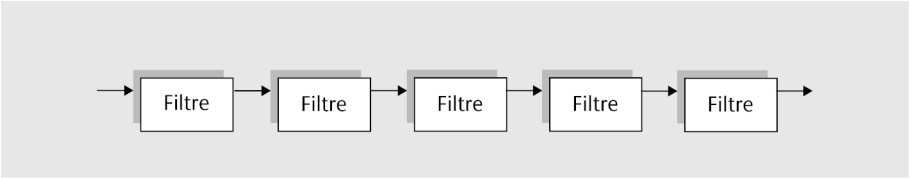


Figura 3. Seqüencial per lots

Un exemple d'arquitectures de fluxos de dades

Suposem una organització que emet factures als seus clients. Una vegada la setmana, els pagaments fets es comparen amb les factures. Per a aquestes factures pagades, s'emet un rebut. Per a les factures que no es paguen dins del temps permès, s'emet un recordatori. El següent és el model d'una part del sistema de processament de factures.

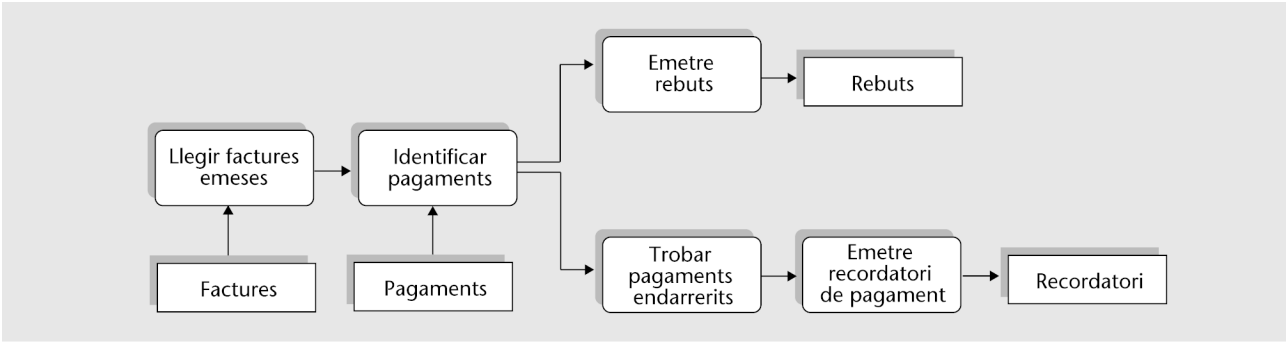


Figura 4. Un model de flux de dades d'un sistema de processament de factures

Es recomana utilitzar aquest estil quan el focus principal de la solució se centra en les dades i, a més, és possible especificar una seqüència d'un nombre conegut de passos o processos per a resoldre el problema inicial. En aquest sentit, els processos són mers elements que van transformant les dades a cada pas. Noteu que és requisit indispensable també que tots els components situats més endavant en la canonada siguin capaços d'inspeccionar i actuar sobre les dades que vénen de més enrere, però no viceversa.

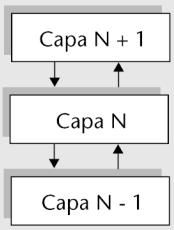
Els **principals avantatges** d'aquest estil arquitectònic són:

- Els filtres són entitats independents fàcilment reutilitzables.
- Encara que en principi forcen un processament seqüencial, moltes vegades els filtres es poden col·locar en paral·lel o fins i tot estar distribuïts.
- Els sistemes caracteritzats per aquest estil arquitectònic són altament modulars i extensibles.
- L'estil és fàcil d'entendre i d'implementar.

El **desavantatge** principal del model prové del fet que força d'alguna manera a un processament lineal de les dades, no apte per a molts tipus de sistemes (com per exemple els sistemes web i aquells en què hi ha moltes interaccions amb els usuaris, que normalment tenen fluxos de control molt complexos).

1.8.2. Sistemes organitzats en capes o nivells

Les **arquitectures en nivells** (*layered architectures*) representen una organització jeràrquica dels elements d'un sistema, de manera que cada capa proporciona serveis als elements de la capa immediatament anterior, i se serveix dels serveis que li brinden els elements de la capa immediatament següent.



En els sistemes organitzats en capes, els components, que poden ser tant procediments com objectes, s'estructuren en **nivells o capes**, cada un dels quals té associat una funcionalitat.

En aquest estil arquitectònic, cada capa pot ser definida com un conjunt de (sub)sistemes amb el mateix grau de generalitat. D'altra banda, en un estil en capes els connectors es defineixen mitjançant els protocols que determinen les vies de la interacció.

La funcionalitat del sistema pot descompondre's en capes horitzontals o verticals.

- A les **capes verticals**, la descomposició sol deure's al nivell d'abstracció dels elements de cada capa: els nivells inferiors implementen funcions simples, lligades al maquinari o a l'entorn, mentre que els nivells superiors implementen funcions més abstractes.

- En els sistemes en **capes horitzontals**, la divisió sol deure's a altres causes, com per exemple el grau de proximitat a l'usuari final (a l'esquerra) davant el processament que duen a terme els *mainframes* i grans gestors de bases de dades (més a la dreta).

- Les **capes intermèdies** solen acollir elements de processament que serveixen de mitjancers entre els clients (a l'esquerra) i els grans servidors, a la dreta. En qualsevol cas, aquesta divisió és molt dependent de la manera com l'arquitecte la dibuixi.

Les restriccions topològiques de l'estil poden incloure una limitació, més o menys rigorosa, que exigeix a cada capa operar només amb capes adjacents, i als elements d'una capa entendre's només amb altres elements d'aquesta; se suposa que si aquesta exigència es relaxa, l'estil deixa de ser pur i perd algunes de les seves propietats. També es perd, naturalment, la possibilitat de reemplaçar una capa sense afectar les restants, amb la qual cosa disminueix la flexibilitat del conjunt i se'n complica el manteniment.

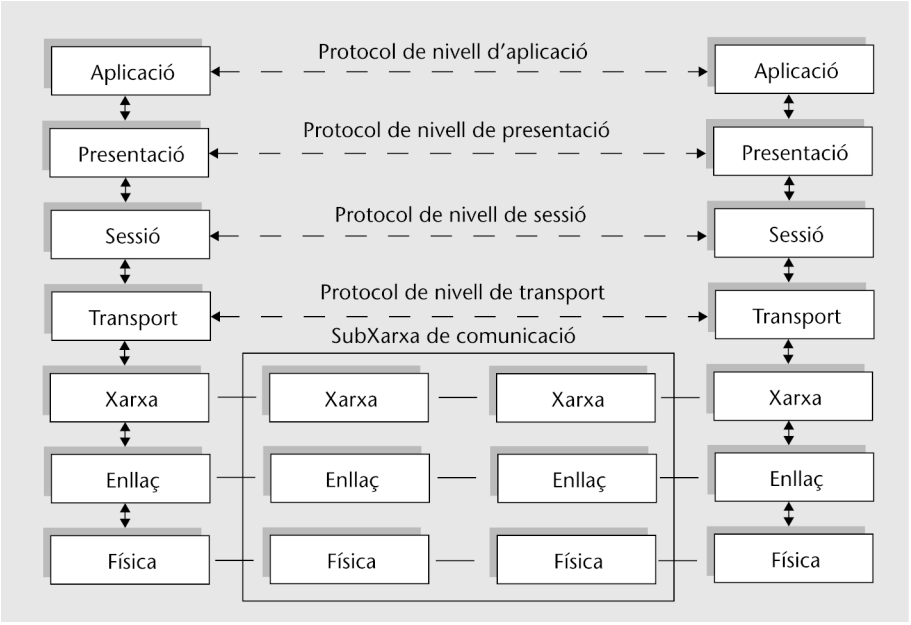


Figura 5. Arquitectura de xarxa basada en el model OSI

Les formes més rígides de les arquitectures en capes no admeten ni tan sols no deixar passar les crides o les dades sense fer-hi res (*pass-through*): cada capa ha de fer alguna cosa, sempre. Moltes vegades, tanmateix, s'utilitzen versions més relaxades d'aquest estil, en què una capa pot accedir directament als serveis de capes no immediatament inferiors per motius de rendiment.

Són casos representatius d'aquest estil molts dels protocols de comunicació; el model OSI (*Open Systems Interconnection*) d'ISO amb els set nivells que tots coneixem (nivell físic, enllaç de dades, xarxa, transport, sessió, presentació i aplicació) és l'exemple més característic d'aquest estil. També es troba en forma més o menys pura en arquitectures de bases de dades i sistemes operatius.

Els **avantatges** són obvis:

- L'estil suporta un disseny basat en **nivells d'abstracció** clarament identificats, la qual cosa, al seu torn, permet als arquitectes la descomposició d'un problema complex en subsistemes independents i complementaris, cada un encarregat d'una missió molt concreta.

- L'estil admet **optimitzacions** i **refinaments** de manera natural.

- També permet una àmplia **reutilització**. Es poden utilitzar diferents implementacions o versions d'una mateixa capa sempre que suportin les mateixes interfícies amb les capes adjacents. Això condueix a la possibilitat de definir interfícies de capa que siguin estàndard, a partir de les quals diferents proveïdors poden construir extensions o prestacions específiques. Això facilita, entre altres coses, la **portabilitat** dels sistemes en els quals l'acoblament amb l'entorn està localitzat a la capa inferior, com succeeix

en els sistemes basats en la torre OSI. Per a portar el sistema a un entorn diferent, n'hi ha prou d'implementar de nou aquest nivell.

D'altra banda, els **desavantatges** més importants d'aquest estil són:

- Molts problemes no admeten ser representats seguint una estructura jeràrquica. Fins i tot quan un sistema es pot establir lògicament en capes, consideracions de rendiment poden requerir acoblaments específics entre capes de nivell alt i baix.

- De vegades és també extremadament difícil trobar el nivell d'abstracció adequat; per exemple, la comunitat de les comunicacions ha trobat complex representar els protocols existents en el model OSI, de manera que molts protocols actuals agrupen en una sola capa diverses capes de les proposades per OSI.

- Els canvis a les capes de baix nivell tendeixen a filtrar-se cap a les d'alt nivell, en especial si s'utilitza una modalitat relaxada d'aquest estil.

- L'arquitectura en capes ajuda a controlar i encapsular aplicacions complexes, però pot complicar les simples.

1.8.3. Arquitectures client-servidor

En una **arquitectura client-servidor**, una aplicació es modela com un conjunt de components servidors, que ofereixen uns serveis i un conjunt de clients que utilitzen aquests serveis.

Així, podríem dir que els integrants principals d'aquest model són:

1) Un conjunt de **servidors** independents que ofereixen serveis a altres components.

2) Un conjunt de **clients** que sol·liciten els serveis oferts pels servidors. En general, pot haver-hi diverses instàncies d'un programa client que s'executa de manera concurrent.

3) Un **mitjà de comunicació**, generalment, una xarxa que permet la comunicació entre clients i servidors, i que permetrà abstreure detalls sobre si la comunicació és local o remota.

En el cas més simple que els clients i els servidors siguin components homogenis i s'executin en una sola màquina, no caldria un mitjà de comunicació especial.

La comunicació entre clients i servidors pot ser **síncrona** o **asíncrona**.

- En el cas **síncron**, el servidor torna el control als clients juntament amb el servei sol·licitat.
- En el cas **asíncron**, el client té el seu propi flux de control i tots dos components intercanvien només dades.

D'altra banda, els rols exercits pels components no són necessàriament fixos. Un servidor, per exemple, podria sol·licitar un servei a un altre servidor a fi de satisfer la petició del seu client, i actuaria per tant en aquell moment al seu torn com a client. Anàlogament, els clients també poden actuar com a servidors per a altres clients en cas que implementin alguns serveis. El cas extrem es dóna en els **sistemes entre parells** (*peer-to-peer*) en què els rols de client i servidor es dilueixen, ja que tot objecte es comporta alhora com a client i servidor.

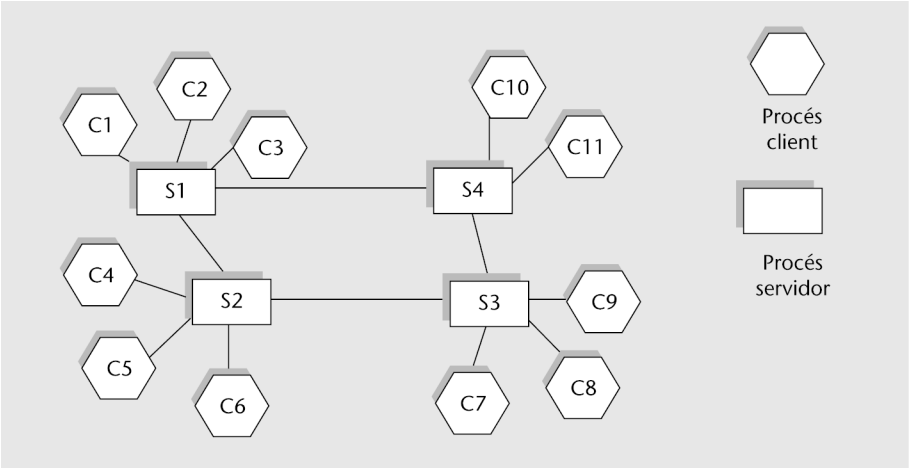


Figura 6. Un sistema client-servidor

Una característica molt important dels sistemes client-servidor és que els clients han de conèixer els noms dels servidors disponibles i els serveis que subministren, encara que en general no han de conèixer l'existència d'altres clients. Tanmateix, els servidors no requereixen conèixer la identitat dels clients o el nombre. En el cas d'objectes distribuïts, els clients accedeixen als serveis subministrats per un servidor a través de crides a procediments remots.

Clients i servidors

En general, quan es parla de clients i servidors ens referim a processos lògics més que a ordinadors físics sobre els quals s'executen aquests processos. Els clients i servidors són processos diferents i poden o no estar allotjats en el mateix node físic.

Un dels **avantatges** més importants del model client-servidor és que es permet de manera natural la distribució dels seus components. A més, aquest tipus de sistemes solen ser fàcilment escalables, la qual cosa permet l'agregació de nous servidors i la seva integració amb la resta del sistema quan cal. De la mateixa manera, és possible actualitzar de manera transparent els servidors sense afectar el funcionament d'altres parts del sistema.

Tanmateix, la falta d'un model compartit de dades entre clients i servidors (i també entre els mateixos servidors) pot suposar un problema. Cada subsistema, en general, organitza i administra les seves dades de manera diferent. Això

significa que els models de dades són específics per a cada servidor i que un client ha d'adaptar els seus models de dades per a treballar amb els diferents servidors que utilitza.

Model mestre-esclau

Una variant interessant del model client-servidor és el conegut com a **mestre-esclau**, que s'utilitza sovint en dominis de càlcul científic. En una arquitectura d'aquest tipus, el client es coneix com a "mestre" i els servidors, com a "esclaus". El mestre és el responsable de resoldre un problema de complexitat elevada i per a fer-ho utilitza algun algoritme divideix-i-venceràs o de ramificació i poda que descompon el problema en nombrosos subproblemes més petits. El mestre encarregarà als esclaus la solució de cada un d'aquests subproblemes i, posteriorment, combinarà les solucions parcials i construirà la solució final. En el cas particular en què tots els subproblemes siguin iguals i, per tant, els esclaus siguin del mateix tipus, estaríem parlant de **granges de processos**.

Aquest tipus de sistemes s'ha començat a utilitzar amb molt èxit amb finalitats científiques en projectes que aprofiten els temps d'inactivitat dels ordinadors personals connectats a Internet, que voluntàriament cedeixen les seves CPU per a fer càlculs. Aquests ordinadors personals actuen per tant com a esclaus per a un procés mestre que tracta de resoldre algun problema d'alta complexitat científica com pot ser la cerca d'intel·ligència extraterrestre analitzant els "sorolls" rebut de l'espai (projecte SETI@home) o processos bioquímics com l'autoassemblatge de proteïnes (projecte folding@home).

1.8.4. Arquitectures heterogènies: sistemes client-servidor organitzats en capes

Habitualment, les aplicacions client-servidor s'estructuren en capes. Depenent de la càrrega i el nivell de processament que facin els components de cada capa, podem distingir diferents estils arquitectònics:

Arquitectures client-servidor de dues capes amb client lleuger (*thin client*)

En un model de client lleuger, tot el processament de l'aplicació i l'administració de dades es fa al servidor. El client només és responsable d'executar el programari de presentació. En casos extrems, un client lleuger només actua com un terminal.

Un gran desavantatge del model de client lleuger és que col·loca una gran càrrega de processament tant al servidor com a la xarxa. El servidor és responsable de tots els càlculs i moltes vegades això implica la generació de bastant trànsit a la xarxa entre el client i el servidor.

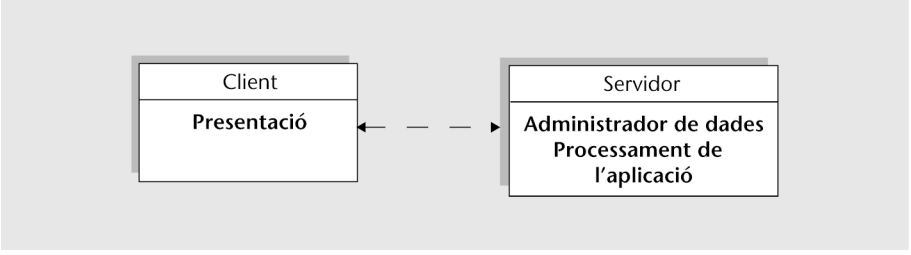


Figura 7. Model de client lleuger

Arquitectures client-servidor de dues capes amb clients pesants (*fat client*)

En aquest model el servidor és només responsable de l'administració de les dades. El programari del client implementa la lògica de l'aplicació i les interaccions amb l'usuari del sistema. Essencialment, el servidor és un servidor de transaccions que administra totes les transaccions de la base de dades.

Encara que el model de client pesant distribueix el processament de manera més efectiva que un de client lleuger, l'administració del sistema resulta més complexa. La funcionalitat de l'aplicació es distribueix a molts ordinadors diferents. Així, quan canvia l'aplicació programari, cal reinstal·lar-la en cada client del sistema.

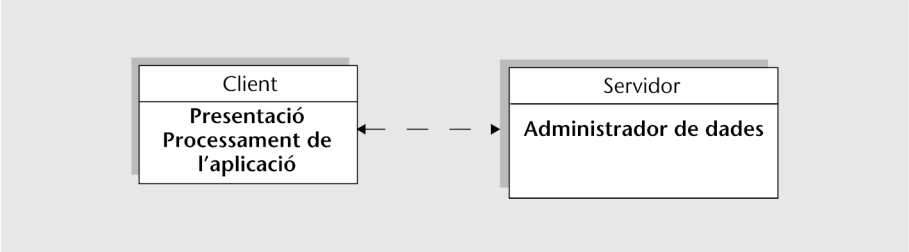


Figura 8. Model de client pesant

Arquitectures client-servidor de tres capes

Per evitar els problemes d'escalabilitat i administració que presenten els dos models anteriors, va sorgir l'arquitectura client-servidor de tres capes. En aquesta arquitectura, la presentació, el processament de l'aplicació i l'administració de les dades són processos separats lògicament.

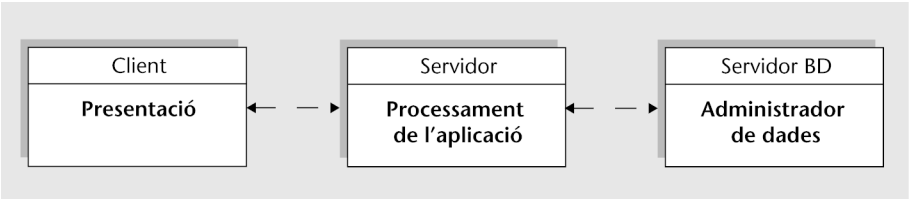


Figura 9. Model de client-servidor de tres capes

Una arquitectura client-servidor de tres capes no implica necessàriament que hi hagi tres sistemes de còmput connectats a la xarxa. Els processos servidors encarregats del processament i de l'administració (com a servidors lògics diferents) podrien executar-se en una mateixa màquina. Tanmateix, si sorgeix la necessitat, és relativament fàcil separar el processament de l'aplicació i la distribució de dades i executar-los en processadors separats.

Variant de múltiples capes

En alguns casos, és apropiat ampliar el model client-servidor de tres capes a una variant de **múltiples capes** (per exemple, quatre) en la qual s'agreguen diferents nivells de servidors al sistema. Els sistemes de múltiples capes

s'utilitzen, per exemple, quan les aplicacions necessiten accedir i utilitzar dades de diferents bases de dades. En aquest cas, un servidor d'integració s'ubica entre el servidor d'aplicació i els servidors de bases de dades als quals s'accedeix. El servidor d'integració recull les dades distribuïdes i els presenta a l'aplicació com si aquestes estiguessin disponibles en una sola base de dades.

Observeu que la capa del servidor per al processament de l'aplicació no implica que hi hagi un sol servidor, sinó que hi pot haver diferents servidors cada un encarregat de processar part del funcionament de l'aplicació.

Altres casos en què s'utilitza una quarta capa estan en els dissenys arquitectònics de certes aplicacions web en les quals es col·loca una capa intermèdia (denominada *proxy invers*) entre la capa del client i del servidor del processament de l'aplicació. El *proxy invers* és l'encarregat de servir al client totes les pàgines i objectes estàtics, mentre que les pàgines dinàmiques les prepara el servidor, que es comunica al seu torn amb els servidors de bases de dades per obtenir la informació apropiada.

Com havíem esmentat en l'apartat d'arquitectures heterogènies, per al disseny global de moltes aplicacions web sol ser comú l'ús d'una arquitectura de tres capes, on la capa intermèdia es construeix com un conjunt d'objectes distribuïts. D'aquesta manera, els sistemes orientats a objectes distribuïts que veurem a continuació poden ser una via que representi internament la capa del processament de l'aplicació amb arquitectura de tres capes.

1.8.5. Sistemes orientats a objectes distribuïts

Una arquitectura orientada a objectes distribuïts estructura el sistema en un conjunt d'objectes feblement acoblats amb interfícies ben definides. Els objectes truquen a serveis oferts per altres objectes. Així, la interacció entre objectes té lloc a través d'invocacions de les seves operacions.

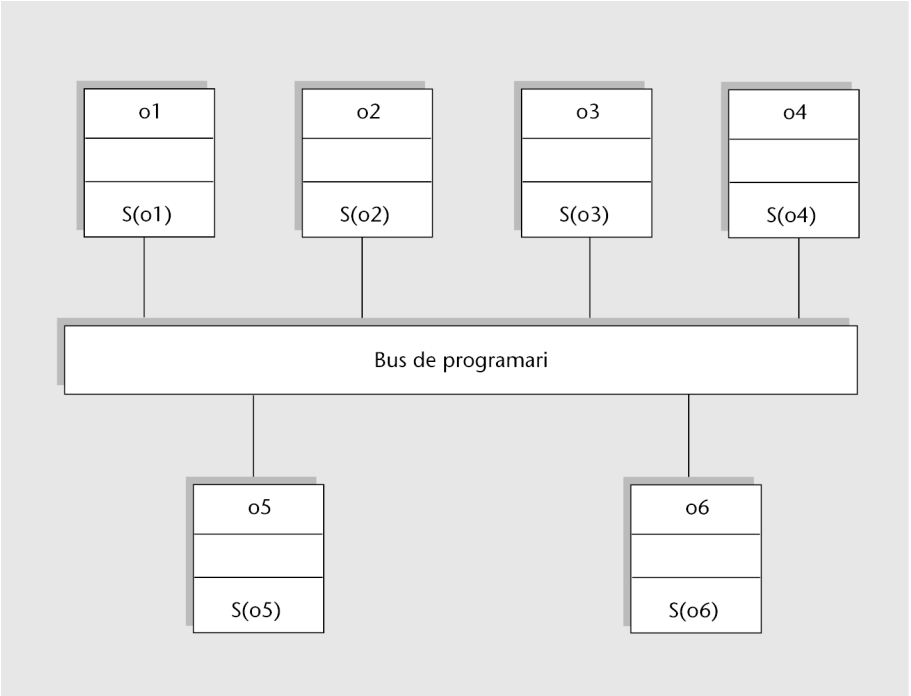


Figura 10. Arquitectura d'objectes distribuïts

Els objectes es distribueixen al llarg de diversos ordinadors sobre una xarxa i es comuniquen a través de plataformes programari intermediari. Per analogia amb un bus de maquinari, el programari intermediari es pot visualitzar com un bus de programari que proporciona un conjunt de serveis que faciliten la comunicació, agregació i distribució dels objectes del sistema. El seu rol és proveir una interfície transparent entre els objectes que els abstregui de detalls sobre la localització, o sobre si els components són heterogenis o no (permetent connectar, per exemple, components desenvolupats amb llenguatges o plataformes diferents: C++, Java, Smalltalk, etc.)

Un exemple de sistemes orientats a objectes distribuïts

En la figura 11 es presenta un exemple d'un sistema construït mitjançant un model client-servidor. Aquest és un sistema d'hipertext multiusuari que proporciona una biblioteca de pel·lícules i fotografies. En aquest sistema hi ha diversos servidors que administren i despleguen els diferents tipus de mitjans. Les pel·lícules s'han de transmetre ràpidament i de manera sincronitzada, però a una resolució relativament baixa. Poden estar comprimides en un magatzem. Tanmateix, les imatges s'han d'enviar en alta resolució. El catàleg ha d'estar disponible per a respondre a una gran varietat de pel·lícules i proveir els sistemes d'informació d'hipertext. El programa client és simplement una interfície d'usuari integrada amb aquests serveis.

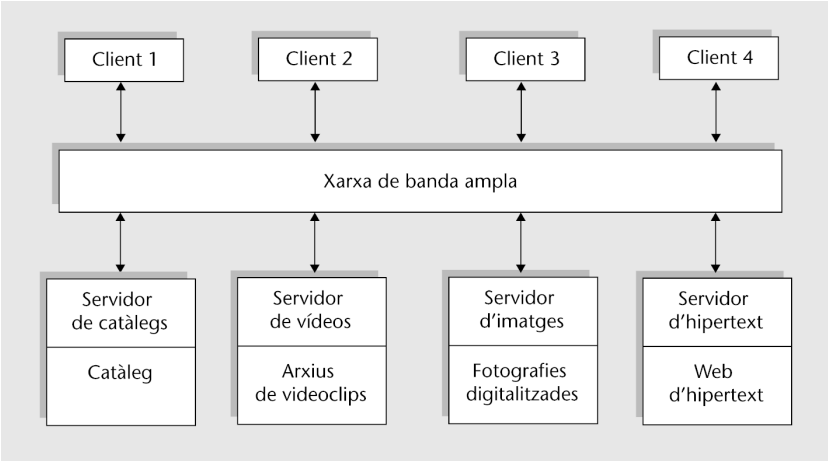


Figura 11. Arquitectura d'alt nivell corresponent a un sistema bibliotecari per a pel·lícules i imatges

Els principals **avantatges** d'aquest estil arquitectònic a l'hora de dissenyar sistemes distribuïts són:

- Permet al dissenyador del sistema **retardar les decisions** sobre on i com s'han de subministrar els serveis. Els objectes proveïdors de serveis es poden executar en qualsevol node de la xarxa. Per tant, la distinció entre els models de client lleuger i pesant és irrellevant, ja que no hi ha necessitat de decidir per avançat on es localitzen els objectes lògics de l'aplicació.

- Aquesta és una arquitectura de sistemes **molt oberta** que permet agregar nous recursos si cal. Les plataformes programari intermediari han estat desenvolupades i han estat implementades per a permetre la comunicació i els serveis entre objectes escrits en diferents llenguatges de programació, així com la integració d'aplicacions i components distribuïts.

- El sistema és **flexible** i **escalable**. Es poden crear diferents instàncies del mateix servei subministrat per objectes diferents o per rèpliques d'objectes per a fer front a sobrecàrregues del sistema. Si aquesta càrrega s'incrementa, es poden activar nous objectes sense pertorbar-ne d'altres.

- És possible **reconfigurar el sistema** de manera dinàmica segons les necessitats, migrant els objectes al llarg de la xarxa. Un objecte proveïdor de serveis es pot traslladar el mateix processador on rauen els objectes sol·licitants del servei.

Tanmateix, el principal **desavantatge** de les arquitectures d'objectes distribuïts és que són més complexes de dissenyar que els sistemes client-servidor. L'arquitectura client-servidor sembla ser la manera més natural de concebre els sistemes, ja que reflecteixen moltes transaccions humanes en les quals la gent sol·licita i rep serveis d'altres persones especialitzades a subministrar aquests serveis. És més difícil pensar en el proveïment de serveis generals si es manca d'experiència en el disseny i desenvolupament d'objectes de gra gruixut.

1.8.6. Arquitectures basades en esdeveniments

La idea dominant en les arquitectures basades en esdeveniments és que, en comptes d'invocar un procediment directament (invocació explícita), com es faria en un estil orientat a objectes, un component pot anunciar, mitjançant difusió, un esdeveniment o més (**invocació implícita**). Vegeu la figura 12.

Invocació implícita

En la literatura relacionada podem trobar diferents denominacions per a aquest estil arquitectònic, com per exemple "arquitectures d'invocació implícita", "d'integració reactiva", o "de difusió (*broadcast*) selectiva".

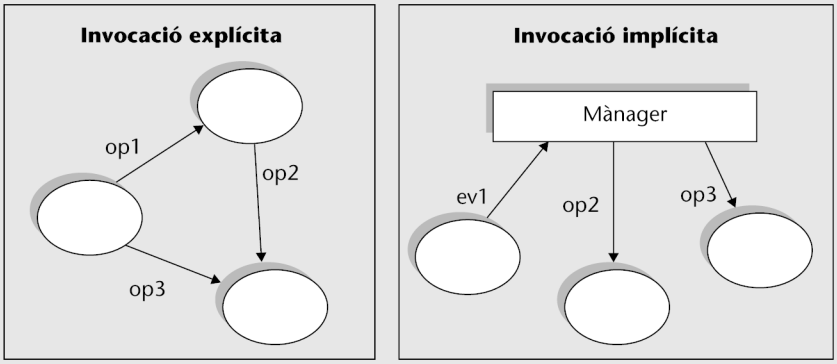


Figura 12. Invocació explícita i invocació implícita

D'aquesta manera, els components d'un sistema interessats en un esdeveniment determinat se **subscriuen** a l'objecte emissor d'aquest esdeveniment, li notifiquen que volen que els avisi quan es produeixi l'esdeveniment i passen a l'objecte emissor la referència d'un procediment que volen que invoqui quan es produeixi aquest esdeveniment. D'aquesta manera, en l'objecte emissor queden vinculats com a **oients** (*listeners*) de l'esdeveniment. Quan l'esdeveniment succeeix, el sistema gestor d'esdeveniments invoca tots els procediments que havien sol·licitat els objectes oients que hi hagi registrats per a l'esdeveniment. D'aquesta manera, l'anunci d'un esdeveniment implícitament ocasiona la invocació de determinats procediments a tots els objectes oients.

Call-backs i el "principi de Hollywood"

Es denomina *call-back* el mecanisme que segueixen els sistemes orientats a esdeveniments, mitjançant el qual l'objecte interessat en un esdeveniment s'hi subscriu i espera que el cridin quan succeeixi l'esdeveniment (d'on prové el terme *call-back*). Es diu que aquest mecanisme segueix l'anomenat principi de Hollywood: "No truqui vostè; nosaltres l'avisarem".

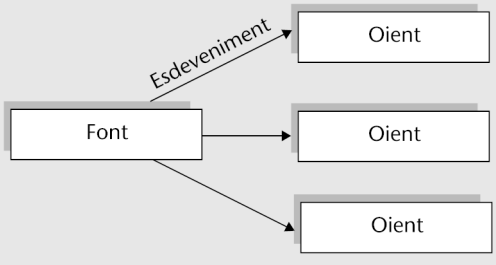


Figura 13. Arquitectures d'esdeveniments

Aquest estil s'ha d'aplicar quan es vol gestionar, de manera aïllada, diverses implementacions d'una "funció" específica. Des del punt de vista arquitectònic, els components d'una arquitectura basada en esdeveniments són objectes o processos les interfícies dels quals proporcionen tant una col·lecció de procediments com un conjunt d'esdeveniments. Els procediments es poden invocar a la manera usual en models orientats a objecte o mitjançant el sistema de subscripció que s'ha descrit.

Els exemples de sistemes que utilitzen aquesta arquitectura són nombrosos. L'estil s'utilitza en entorns d'integració d'eines, en sistemes de gestió de bases de dades per a assegurar les restriccions de consistència sota la forma de disparadors (*triggers*), en interfícies d'usuari per a separar la presentació de les dades dels procediments que les gestionen i en editors sintàcticament orientats per a proporcionar verificació semàntica incremental, etc.

Entre els **avantatges** del model podem destacar:

- Les característiques de l'estil el fan molt apropiat per a la implementació de sistemes reactius, i especialment quan hi ha una gran interacció amb l'usuari.

- L'alt grau de desacoblament entre els components del sistema, que no es coneixen entre si ni fan referència els uns als altres, optimitza el manteniment del sistema.

- El model fomenta el desenvolupament en paral·lel, la qual cosa pot portar millores de rendiment.

- Les arquitectures basades en esdeveniments presenten una alta versatilitat, reutilització, reemplaçabilitat i evolució fàcil, la qual cosa permet el registre, la baixa o el reemplaçament dinàmic de components i esdeveniments.

- És aplicable tant si les implementacions corren sincrònicament com asíncronament perquè no s'espera una resposta.

- Permeten evitar la degradació de les prestacions del sistema que succeiria si, en comptes d'esperar que els avisin que ha succeït l'esdeveniment que esperen, els objectes oients haguessin de preguntar repetidament per la seva ocurrència (fent el que es coneix com a *polling*, un mecanisme força ineficient per la sobrecàrrega de missatges que genera en el sistema).

Entre els **desavantatges** d'aquest estil arquitectònic esmentarem els següents:

- Quan un component anuncia un esdeveniment, no sap quins altres components hi estan interessats, ni l'ordre en què seran invocats, ni el moment en què acaben el que han de fer. Aquest desconeixement pot derivar en

problemes de rendiment, de gestió de recursos quan es comparteix un dipòsit comú o de coordinació.

- Un component no sap com influeixen en la resta del sistema els esdeveniments que emet, de manera que, quan els emet no pot assumir que altres components hi responguin. Encara que sabés que hi ha altres components subscrits a un esdeveniment, tampoc no pot assumir que aquests components s'invoquin en un ordre determinat.

- Finalment, és difícil raonar sobre la correcció o el comportament del sistema, ja que la reacció a un esdeveniment depèn del context en què es dugui a terme.

Com sempre, els avantatges o desavantatges són molt relatius, ja que depenen molt del domini d'aplicació i del sistema concret que pretenem implementar. El que en alguns casos és avantatge es converteix en desavantatge en altres casos.

1.8.7. Arquitectures orientades a serveis

Recentment, les arquitectures orientades a serveis (*service oriented arquitectures*, **SOA**) estan cobrant molt interès per al desenvolupament d'aplicacions en Internet i estan rebent un tractament especial en l'anàlisi dels diferents estils arquitectònics. Més que un subestil de les arquitectures basades en components independents, recentment hom les ha començat a considerar un estil propi.

L'arquitectura SOA està caracteritzada per les propietats següents:

- Una **vista lògica**: els **serveis** són una abstracció (vista lògica) dels programes, bases de dades, processos de negoci, etc., que intervenen en l'aplicació i són definits en termes del que fan. D'aquesta manera, els components bàsics d'aquest estil són els serveis que implementen la lògica del negoci o la funcionalitat bàsica del sistema.

- **Orientació a missatges**: com a part de la descripció dels serveis es defineixen els missatges intercanviats entre proveïdors i sol·licitants. L'estructura interna del servei (llenguatge de programació, processos interns, etc.) romanen ocults a aquest nivell d'abstracció.

- **Orientació a la descripció**: un servei es descriu amb metadades processables. La descripció dóna suport a la naturalesa pública de SOA: només s'inclouen en la descripció els detalls que s'exposen públicament i són importants per a l'ús del servei. Així, un servei web simple queda caracteritzat per quatre estàndards: XML, SOAP, UDDI i WSDL, que treballen segons el model bàsic, de sol·licitud/resposta.

- **Granularitat**: els serveis tendeixen a usar un petit nombre d'operacions amb missatges relativament complexos.

- **Orientació a la xarxa**: els serveis tendeixen a usar-se a través de la xarxa, encara que aquest no és un requisit indispensable.

- **Independent de la plataforma**: els missatges s'envien en un format estàndard i neutral a la plataforma (normalment XML).

Ara bé, encara que SOA és general i (almenys en teoria) es basa en la idea de serveis en general comunicats mitjançant qualsevol tipus de missatges, el normal és que sempre s'utilitzi en entorns web i que es recolzi en estàndards web com HTTP, l'XML, SOAP, WSDL o UDDI. Per tant, en SOA és molt important el concepte de servei web.

Un **servei web** és un sistema programari identificat per una URI, les interfícies públiques i els enllaços del qual es defineixen i es descriuen mitjançant XML. La seva definició pot ser descoberta per altres sistemes programari. Aquests sistemes poden interactuar amb el servei web de la manera prescrita per la seva definició, usant missatges basats en XML a través de protocols estàndards d'Internet.

URI

URI és la sigla d'*uniform resource identifier*, una cadena de caràcters que serveix per a identificar de manera unívoca qualsevol recurs d'Internet (com un ordinador, una pàgina web o un servei). En són exemples http://www.lcc.uma.es/ o ftp://www.uoc.edu. Observeu que també indica el protocol per a accedir-hi (http, https, ftp, mailto, etc.).

En definitiva, un servei web exposa la seva funcionalitat a consumidors possibles en una URI i proporciona mecanismes per a invocar les seves operacions de manera remota (a través d'Internet). El servei web pot implementar-se en qualsevol llenguatge i a qualsevol plataforma, actuant com una caixa negra.

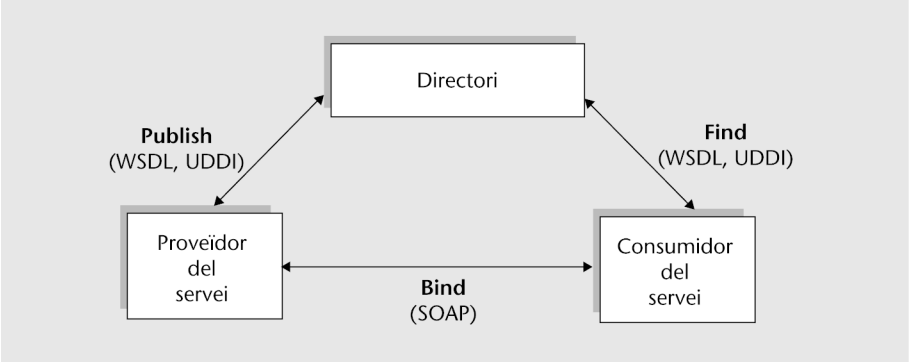


Figura14. Esquema de funcionament dels serveis web

L'esquema de funcionament dels serveis web requereix tres elements fonamentals, i que constitueixen també l'esquema normal de SOA (vegeu la figura 14):

1) Un proveïdor del servei web, que és qui el dissenya, el desenvolupa i l'implementa i el fa disponible per al seu ús, ja sigui dins de la mateixa organització o al públic en general. Les operacions de publicació involucren l'anunci del servei com a tal, la qual cosa correspon a la ubicació del servei en un servidor específic i a l'ús d'un servei de descripció (perquè els clients puguin saber quines funcions té disponibles el servei web i quina informació ha de passar-se a aquestes funcions per a utilitzar-les).

2) Un consumidor del servei, que és qui accedeix al servei web per utilitzar els serveis que aquest presta. Quan un consumidor vol accedir a un servei web, aquest ha de tenir un servidor de descobriment que permeti conèixer la ubicació exacta del servei, és a dir, s'ha de tenir un directori on es tinguin llestes les referències als serveis disponibles. Això s'aconsegueix gràcies a directoris UDDI.

3) Un **agent de servei**, que serveix com a enllaç entre proveïdor i consumidor per a efectes de publicació, cerca i localització del servei.

En general, SOA i els serveis web són apropiats per a aplicacions:

- que han d'operar a través d'Internet,
- on no hi ha la possibilitat de gestionar la instal·lació de manera que tots els clients i proveïdors s'actualitzin alhora,
- on els components d'un sistema distribuït s'executin en diferents plataformes i diferents productes.

1.9. Criteris per a la selecció d'un estil arquitectònic

A l'hora de dissenyar l'arquitectura d'una aplicació, no hauríem de partir des de zero, sinó que hauríem de **reutilitzar** les solucions que han demostrat en el passat que eren bones. Per tant, és molt important identificar quins són els **patrons arquitectònics** més convenients per a la nostra aplicació concreta. Una vegada identificats aquests patrons, hauríem de decidir sota quines condicions i en quins casos poden aplicar-se cada un, com han de ser les interfícies dels components amb els quals s'insti un patró determinat, etc.

L'elecció correcta d'un estil arquitectònic és important, perquè aquest guiarà tot el procés posterior de desenvolupament. En aquest sentit, els dissenyadors han de trobar l'estil més apropiat d'acord amb l'especificació dels requisits concrets del sistema, incloent-hi els funcionals, d'escalabilitat, disponibilitat, seguretat, manteniment i evolució.

Per a un projecte programari donat, hi pot haver diverses arquitectures apropiades. Decidir quina és la millor opció dependrà d'una sèrie de criteris de qualitat, molts dels quals solen ser contradictoris entre si (com ocorre, per

exemple, amb la modularitat davant l'eficiència). Per tant, no només haurem **d'identificar** quins són els aspectes de qualitat que s'han de considerar, sinó també **prioritzar-los**.

- **L'extensibilitat** facilita l'addició de noves característiques, encara que això pot fer més complex el disseny. En general, els sistemes fàcilment ampliables presentaran més grau d'abstracció. Generalitzar (per exemple, decidir quin tipus d'ampliacions poden sorgir, els punts de variabilitat, etc.) requereix invertir bastant temps en el disseny. La distinció entre els requisits opcionals i els desitjables pot ser també molt important, ja que assenyala cap on apuntarà el desenvolupament del sistema.

- La **modificabilitat** té com a objectiu primordial facilitar el canvi de requisits. Observeu que aquesta és diferent de la propietat anterior, encara que requereixi tècniques similars.

- La **simplicitat** tracta de fer fàcil d'entendre i d'implementar l'estil. En contrapartida, aquesta propietat és difícil de compaginar amb les dues anteriors.

- Finalment, **l'eficiència** derivarà en aplicacions de petita mida o alta velocitat. Moltes vegades l'eficiència va en contra de les propietats anteriors, ja que per a augmentar l'eficiència ens podem veure obligats a saltar-nos algunes de les normes de l'arquitectura (com per exemple, accedir a capes inferiors en una arquitectura en capes).

2. Representació de l'arquitectura programari

El disseny arquitectònic se centra en el modelatge de l'aplicació sense abordar la seva distribució física, és a dir, no ens preocupem de moment sobre on seran ubicats físicament els components que formen part d'aquesta arquitectura. Per a fer això, les característiques del sistema han de ser expressades mitjançant algun **llenguatge de descripció d'arquitectures** programari que permeti descriure el sistema en termes de components i connectors.

Com a llenguatge de descripció d'arquitectura utilitzarem UML 2.0, que incorpora els mecanismes adequats per a aquesta finalitat, com veurem en l'apartat següent. A tall d'exemple, ens centrarem en l'estil arquitectònic en tres capes, que és l'utilitzat habitualment per a aplicacions web. En qualsevol cas, aquestes guies són genèriques i poden ser adaptades a qualsevol dels estils arquitectònics descrits en l'apartat anterior.

2.1. UML 2.0 com a llenguatge de descripció d'arquitectures

El llenguatge de modelatge unificat (*unified modeling language*, UML) és un llenguatge estàndard, àmpliament conegut en la indústria del programari, que permet especificar, visualitzar, construir i documentar els diferents elements que concerneixen els sistemes programari, a més de fer models de negoci o altres de sistemes no programari.

Vegeu també

El llenguatge UML s'ha vist en l'assignatura *Enginyeria del programari*.

La versió 1 d'UML no disposava dels conceptes adequats per a representar l'arquitectura programari d'un sistema. Això s'ha solucionat ja en la versió 2.0, amb la redefinició del concepte de component perquè ja no solament sigui físic, i la inclusió dels conceptes de ports i connectors. A més, els mecanismes d'ampliació disponibles en UML (els **perfils**) poden utilitzar-se per a definir altres conceptes no previstos o per a establir restriccions que defineixin de manera més precisa la semàntica d'aquests conceptes.

En concret, les **principals millores** que aporta UML 2.0 per a la descripció arquitectònica dels sistemes programari són:

- Nous conceptes per a descriure l'estructura arquitectònica interna de les classes, components i col·laboracions a partir de la definició de parts (*parts*), connectors (*connectors*) i ports (*ports*).

- Introducció de l'herència de comportament en màquines d'estat i encapsulació de submàquines mitjançant l'ús de punts d'entrada i sortida.

- Millora de l'encapsulació de components mitjançant ports complexos amb màquines d'estat de protocol, que controlen la interacció del component amb el seu entorn.

- Millora d'aspectes d'especificació, realització i connexió als components.

- Millora dels diagrames de comunicació amb conceptes de control més adequats, com ara la composició, les referències, les excepcions, els bucles, les alternatives, etc. A més, s'hi afegeix un nou diagrama, basat en el d'activitats, que permet obtenir una perspectiva més àmplia del conjunt d'interaccions.

En la figura 15 es mostra la taxonomia de diagrames estructurals i de comportament d'UML 2.0.

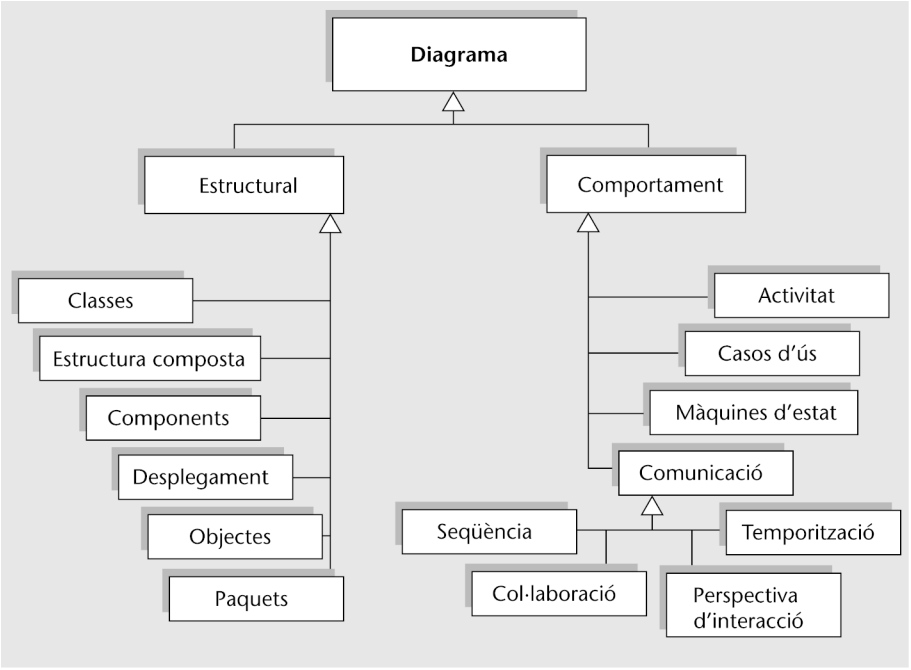


Figura 15. Taxonomia de diagrames d'UML 2.0

Per dur a terme el modelatge i la documentació d'una arquitectura programari, farem ús tant de **diagrames estructurals** (diagrames que mostren l'estructura estàtica dels objectes del sistema sense entrar en aspectes dinàmics) com de **diagrames de comportament** (que permeten descriure els canvis que es produeixen en el sistema amb el pas del temps). Il·lustrarem com fer-ho en el cas concret d'una arquitectura de tres capes.

2.2. Disseny de l'arquitectura de tres capes

L'arquitectura en capes descriu l'organització conceptual dels elements del disseny en grups independentment de l'empaquetament o desplegament físic. Cada capa representa un element gran, sovint compost de diversos paquets o subsistemes.

Habitualment, i en particular en el cas de les aplicacions web, trobarem una divisió en tres nivells atenent aquesta organització:

1) Nivell de presentació

Aquest és el nivell encarregat de generar la interfície d'usuari depenent de les accions dutes a terme per aquest. La capa de presentació conté els components necessaris per a habilitar la interacció de l'usuari amb l'aplicació. Els components de la interfície d'usuari han de mostrar dades a l'usuari, obtenir i validar les dades procedents d'aquest i interpretar les accions d'aquest que indiquen que vol realitzar una operació amb les dades. Així mateix, la interfície ha de filtrar les accions disponibles a fi de permetre a l'usuari fer només les operacions que li siguin permeses en un moment determinat.

2) Nivell de negoci

Conté la lògica que modela els processos de negoci i és on es du a terme tot el processament necessari per a atendre les peticions de l'usuari.

3) Nivell d'administració de dades

És l'encarregat de fer persistent tota la informació, així com de subministrar i emmagatzemar la informació per al nivell de negoci. Gairebé totes les aplicacions i els serveis necessiten emmagatzemar i obtenir accés a un tipus de dades determinat. L'aplicació pot disposar d'un o diversos orígens de dades, que poden ser de tipus diferents. La lògica utilitzada per a obtenir accés a les dades d'un origen de dades s'encapsularà en components lògics d'accés a dades que proporcionin els mètodes necessaris per a la consulta i l'actualització.

Per a representar l'arquitectura d'un sistema programari utilitzarem una col·lecció de diagrames UML: casos d'ús, components, classes, estat, activitat, seqüència i col·laboracions. Cada un il·lustrarà un aspecte concret.

Vegeu també

Els diagrames de classes, activitat, estat i seqüència s'han presentat amb detall en l'assignatura *Enginyeria del programari*, per la qual cosa no hi insistirem aquí.

2.2.1. Diagrames de casos d'ús

Abordar directament el disseny lògic pot ser una tasca complexa per a un dissenyador poc experimentat. Els casos d'ús faciliten aquesta tasca identificant les entitats o els actors que interaccionen amb el sistema, així com la funcionalitat bàsica que es vol implementar.

Establerts els escenaris principals, pot resultar-nos especialment útil la seva agrupació en paquets.

UML defineix els **paquets** com un mecanisme d'agrupació l'objectiu del qual és organitzar els elements de modelatge. Aquesta pràctica bàsica d'aplicar la modularitat dóna suport a la separació d'aspectes que apuntàvem en el mòdul 1.

Nota

Els paquets no són instanciables, és a dir, no es poden tenir instàncies de paquets, per la qual cosa resulten invisibles per al sistema en execució.

Partint d'aquesta representació inicial, convé refinar el diagrama de casos d'ús a fi d'evitar l'acoblament entre paquets, així com les redundàncies possibles. Si un cas d'ús figura a més d'un paquet, haurem de triar el context en què aquest cas d'ús té més pes i establir relacions de dependència amb altres contexts o paquets dels quals s'hagi eliminat el cas d'ús esmentat. Les dependències entre paquets es traduiran en el futur en ineficiència si un canvi en la funcionalitat que implementa un cas d'ús afecta tots els paquets que en depenen.

L'exemple del banc ens servirà com guia per a il·lustrar el procés de disseny lògic i físic de l'arquitectura, com ja havia ocorregut en el mòdul anterior.

La figura 16 il·lustra un extracte del diagrama de casos d'ús per a l'exemple del banc. Com pot observar-s'hi, els casos d'ús s'han agrupat a tres paquets a partir de la funcionalitat del sistema que descriuen. Cada paquet documenta part de la funcionalitat del sistema que pot requerir un component programari o més per a la seva implementació. Els diagrames de components poden ajudar-nos a documentar aquesta relació.

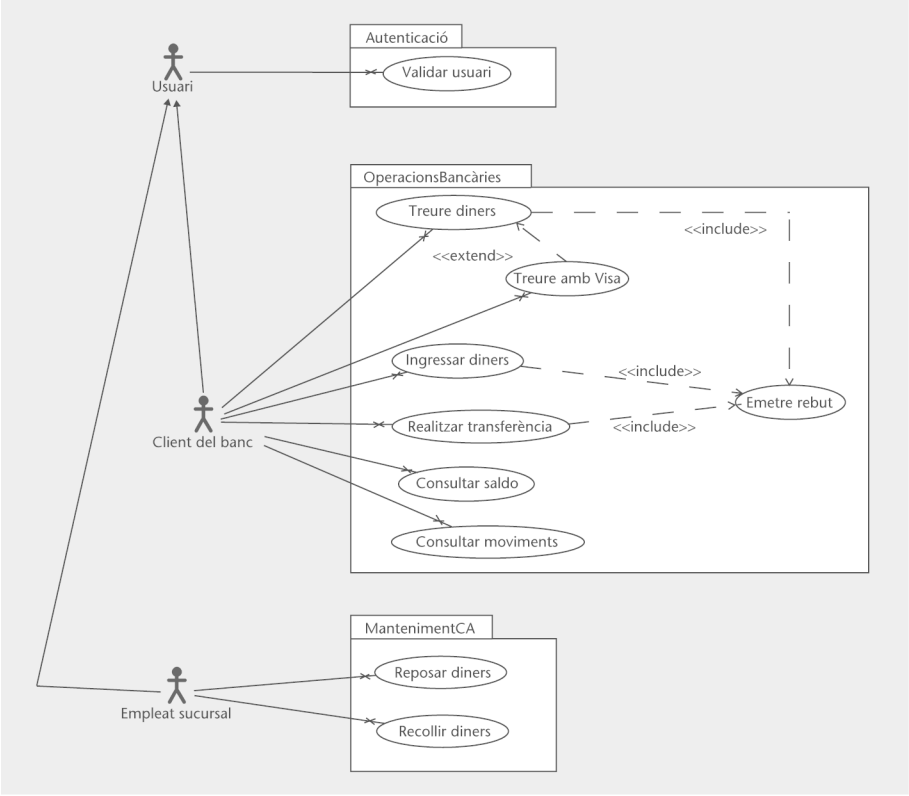


Figura 16. Extracte del diagrama de casos d'ús per a l'exemple del banc

2.2.2. Diagrama de components

Els diagrames de components s'utilitzaran per a il·lustrar la descomposició d'un sistema en components arquitectònics de gra gruixut i les seves interrelacions a través de les interfícies respectives.

En versions anteriors d'UML, els components es consideraven com una representació d'estructures físiques, com ara arxius, DLL, etc. En UML 2.0, els components passen a constituir un potent mecanisme d'especificació en un nivell lògic, ja que només s'utilitzen per a representar i especificar els requisits per a cada element programari.

En UML 2.0, un **component** és una unitat modular del sistema que encapsula una certa funcionalitat, que té interfícies ben definides i que és reemplaçable dins del seu entorn.

Els components es connecten a través d'interfícies implementades (o proporcionades) i interfícies requerides. Per exemple, en la figura 17, el component "Gestor transaccions" implementa una interfície que requereix el component "Sistema comptes".

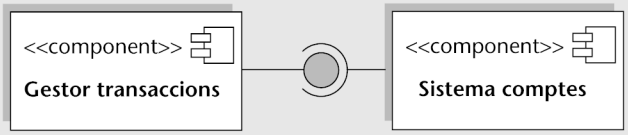


Figura 17. Exemple de components UML 2.0

Els components poden definir comportaments; serà el disseny intern del component el que defineixi la implementació d'aquests comportaments. Per tant, el component ha d'aportar els mitjans perquè altres components puguin requerir els serveis que ofereix.

Una **interfície implementada** (o proporcionada) defineix com s'ha d'accedir als serveis que ofereix un component.

D'altra banda, pot donar-se el cas que un component necessiti suport d'altres components. Aquest haurà de definir, per tant, de manera anàloga què necessita exactament dels altres.

Una **interfície requerida** defineix el tipus de serveis que un component requerirà d'uns altres.

A més de les interfícies, UML 2.0 permet que els components incloguin informació sobre les seves realitzacions[1] i artefactes, i sobre les seves propietats internes. UML 2.0 ho permet per a poder "refinar" aquests components arquitectònics en les fases següents del desenvolupament, incloent-hi els detalls tecnològics sobre com estan **fets** internament. Des del punt de vista arquitectònic, només és rellevant la informació sobre les seves interfícies i interaccions amb altres components. Tanmateix, a l'hora de desenvolupar el sistema cal decidir com s'implementaran internament aquests components. Aquest procés serà tractat amb detall en el mòdul 3.

(1)Una realització es refereix a la implementació d'un requisit.

Un altre aspecte que s'ha de considerar és el cas en què un component ofereixi diferents tipus de serveis als seus congèneres. En general, és possible implementar múltiples interfícies per a expressar aquestes diferències. Tanmateix, seria preferible establir algun criteri d'agrupació, de manera que es permeti aclarir quin tipus de servei dóna una interfície determinada. Per a fer-ho, UML 2.0 fa ús dels ports, en els quals s'agrupen conjunts d'interfícies (tant requerides com implementades), segons criteris de disseny i dels serveis que requereixin o proporcionin.

Un port pot tenir dos tipus d'interfície

Amb les **interfícies proporcionades**, el port exhibeix un conjunt d'operacions al món exterior.

Amb les **interfícies requerides**, el port estableix quin conjunt d'operacions utilitzarà del món exterior.

Un **port** és una característica del component que especifica un punt d'interacció diferenciat entre el component i el seu entorn, o entre el comportament del component i les seves parts internes.

Amb això, UML 2.0 permet especificar punts d'interacció, que exposarà al seu entorn el component, i aportar explícitament la correspondència entre les interfícies publicades i els mecanismes d'implementació interns.

- **Cap a l'exterior**, els ports es connecten amb altres ports mitjançant **connectors**, a través dels quals es fan les peticions al component per a invocar un comportament determinat.

- **Cap a l'interior**, un port connecta els mecanismes interns d'implementació del component (les classes, associacions o altres components que el componen) amb el seu entorn. Això implica que el port serveix al component com a obertura a l'exterior.

Els ports són una aportació important d'UML 2.0, ja que permeten encapsular l'entrada o sortida de les interaccions cap al component. Aquest factor és crucial a l'hora de la reusabilitat, ja que, mantenint el port, és possible modificar la part interna del component sense que afecti la manera com aquest es mostra davant del seu entorn.

En un nivell d'abstracció superior, el concepte de **port** es correspon amb el de **servei**. D'aquesta manera, definirem un port per cada un dels serveis que implementa o requereix un component, i que en defineixen la funcionalitat. Les interfícies UML associades al port defineixen la signatura de les operacions que defineixen el servei.

Utilitzant tots aquests elements estructurals, descriurem l'arquitectura programari d'un sistema centrant l'atenció en l'estructura global d'aquest, i destacant la seva descomposició en paquets, components i interfícies i relacions de dependència entre aquests.

Ús de paquets per representar capes

Cada capa a nivell lògic es representa en UML mitjançant un paquet.

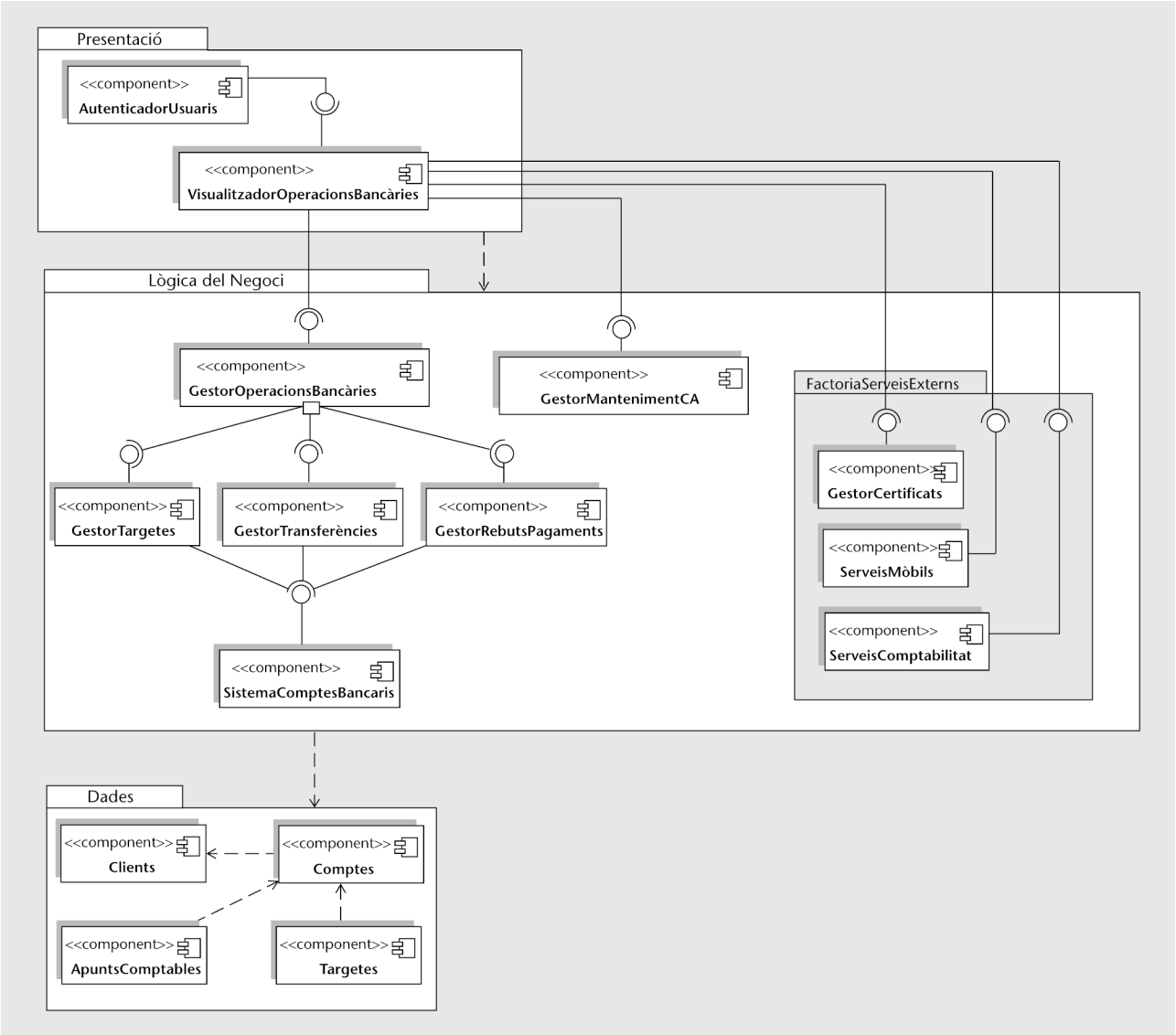


Figura 18. Exemple de representació de l'arquitectura de tres capes per a l'exemple de la banca electrònica

Basant-nos en el model de casos d'ús refinat, organitzarem els paquets en nivells o capes conceptuals independents per al cas concret que ens ocupa: les arquitectures de tres capes. El disseny de cada capa comprendrà dues tasques clarament diferenciades:

- El **disseny intern** té com a finalitat el modelatge dels elements pertanyents a aquest nivell.
- El **disseny extern** defineix la interacció entre aquesta capa i les altres.

Per al disseny intern ens ajudarem dels diagrames de components que calgui: cada un proporcionarà més nivell de detall o descomposició dels paquets i components si cal. A tall d'exemple, la figura 18 il·lustra un diagrama de components corresponent a una distribució o arquitectura en tres capes per a l'exemple del banc en línia. Com veiem, cada capa és modelada com un paquet UML, que proporciona un límit ben definit al voltant d'un conjunt

d'elements relacionats. Al seu torn, cada capa exporta només els elements que altres paquets necessiten veure realment i importa només els elements requerits perquè els elements del paquet puguin dur a terme la seva tasca.

Observeu com en la figura 18 només s'han presentat els elements més significatius. Quan es revela el contingut d'un paquet, s'han de mostrar només els elements necessaris per a transmetre de manera concisa la seva finalitat. Tanmateix, podríem refinar el diagrama anterior amb una descripció més detallada dels components i elements que figuren en cada nivell, com veurem en el mòdul 3. Com més detallat és el disseny, més proper resultarà el model a la implementació.

2.2.3. Diagrames d'interacció

Si bé els diagrames de components només mostren informació estàtica, els diagrames d'interacció s'utilitzen per a documentar i proporcionar informació sobre la dinàmica i la manera com es connecten i es comuniquen els objectes dins d'una capa i entre capes.

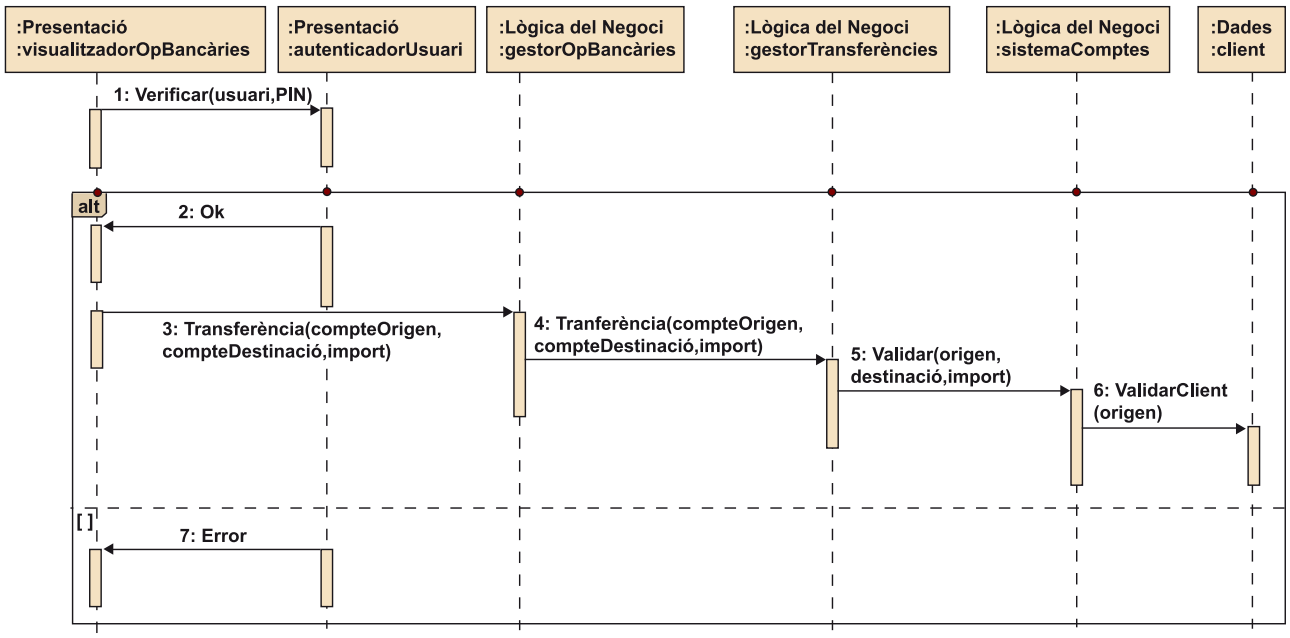


Figura 19. Diagrama d'interacció

En el disseny lògic de l'arquitectura, els **diagrames d'interacció** se centren en les col·laboracions que creuen els límits de les capes i els paquets.

En general, resultarà convenient tenir un conjunt de diagrames d'interacció que il·lustrin els escenaris més significatius des del punt de vista de l'arquitectura.

2.2.4. Col·laboracions

Finalment, completarem el disseny de l'arquitectura programari amb el modelatge de les col·laboracions. En el context de l'arquitectura d'un sistema, una col·laboració permet anomenar un bloc conceptual que inclou tant aspectes estàtics com dinàmics.

Una **col·laboració** denota una societat de classes, interfícies i altres elements que col·laboren per proporcionar algun comportament cooperatiu més gran que la suma dels comportaments dels seus elements.

Les col·laboracions inclouen tant elements **estructurals** com de **comportament**:

- La part **estructural** engloba qualsevol combinació de classes, interfícies, components i nodes que prèviament s'havien declarat en els diagrames estàtics. Una col·laboració és no instable i descriu només els aspectes rellevants de la cooperació d'un conjunt d'instàncies identificades pels rols específics que tenen aquestes instàncies. Per això, una col·laboració dóna nom a un bloc conceptual de l'arquitectura, no a un bloc físic.

- Els diagrames de **comportament**, com ara els diagrames d'interacció, poden afegir-se a una col·laboració per a mostrar més clarament com els rols es relacionen els uns amb els altres a diversos escenaris.

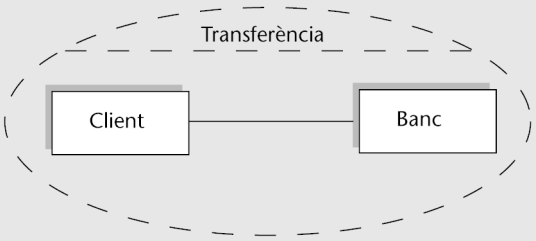


Figura 20. Diagrama de col·laboració

2.3. Modelatge de les dependències entre capes

En UML, si modelem cada capa amb un paquet, usarem relacions de dependència entre els paquets per a mostrar les dependències entre les diferents capes. Una relació de dependència entre dos paquets indica que algun dels elements del paquet "client" invoca o requereix els serveis d'un o més elements del paquet "servidor", però sense especificar aquests elements concrets.

Tanmateix, en moltes arquitectures en capes hi ha un fort acoblament entre els elements de les diferents capes, que es coneix com a arquitectura de "capes transparents". En aquest model, els elements d'una capa col·laboren o es comuniquen amb diversos elements d'altres capes, per la qual cosa d'alguna manera han de ser conscients de l'estructura interna de les capes que utilitzen (observeu el diagrama a l'esquerra de la figura 21). En aquests casos, el que sol fer-se és detallar les relacions de dependència entre els paquets UML, representant les relacions de dependència entre els elements interns dels paquets implicats en les interaccions individuals esmentades.

Nota

L'acoblament i les dependències entre capes es documenten en UML mitjançant relacions de dependència.

A més de les relacions de dependència, una altra via de canalitzar i modelar les dependències entre capes és donada pels patrons de disseny. Com usar-los en una arquitectura per a connectar capes és alguna de les qüestions que tractarem en l'apartat següent.

2.4. Ús de patrons en el disseny arquitectònic

Mentre l'arquitectura en capes guia la definició de les parts importants del sistema, els patrons de disseny poden ser utilitzats per al disseny de les connexions entre capes i paquets UML.

Vegeu també

Els patrons de disseny s'han presentat amb detall en l'assignatura *Enginyeria del programari orientat a objectes*. En aquest apartat repassarem breument alguns conceptes associats a aquest terme tot detallant-ne la utilitat des del punt de vista de l'arquitectura programari.

Els patrons permeten identificar i completar els casos d'ús bàsics exposats pel client, comprendre l'arquitectura del sistema que es construirà i la seva problemàtica, així com buscar components ja desenvolupats que compleixin amb els requisits del tipus de sistema que es construirà. És a dir, ens permeten obtenir d'una manera senzilla l'arquitectura base que busquem durant la fase de disseny arquitectònic.

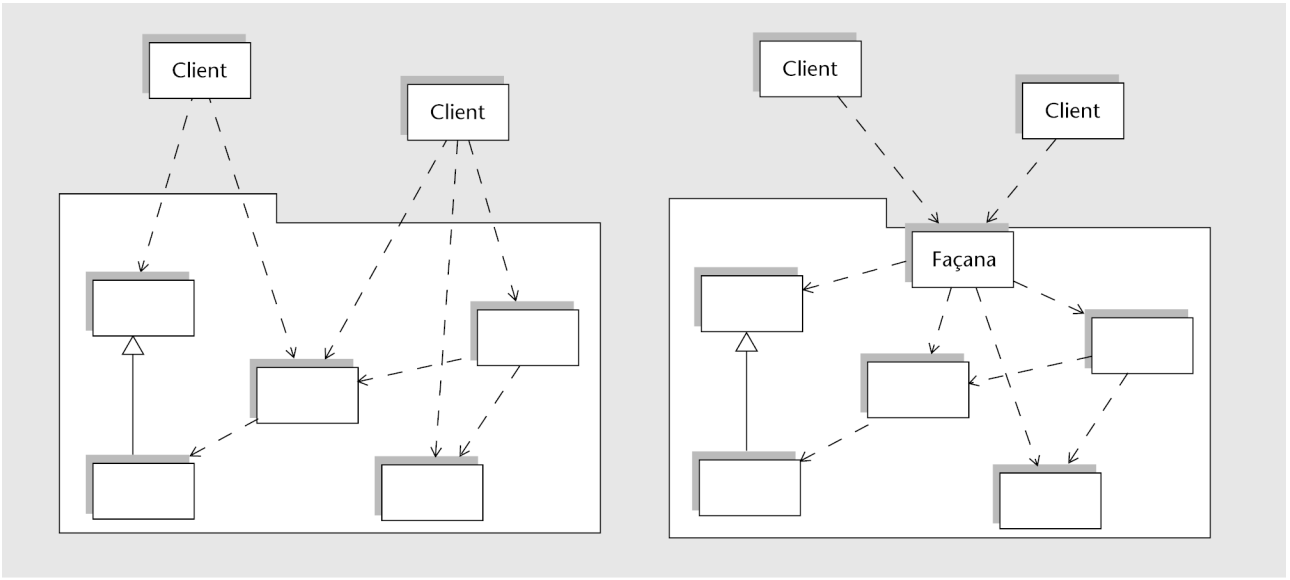


Figura 21. Organització d'una capa sense aplicar el patró façana (esquerra) i aplicant-lo (dreta)

El patró façana

Per exemple, per al cas concret d'una arquitectura en tres capes serà de gran ajuda dotar d'una façana cada nivell de subsistemes i utilitzar-la com a punt d'accés a aquest. El patró **façana** ajuda a controlar o eliminar les dependències complexes o circulars entre objectes i permet fer canvis en els components d'un subsistema sense afectar els clients. D'aquesta manera se simplificarà al màxim el manteniment de les dependències entre nivells.

L'ús de patrons es documenta en UML mitjançant diagrames de col·laboració. Una col·laboració descriu tant un *context* com una *interacció*. El context fa referència als objectes/rols involucrats en la col·laboració, mentre que la interacció documenta la col·laboració que els objectes/rols duen a terme, bé sigui a través de pas de missatges, invocació directa, etc.

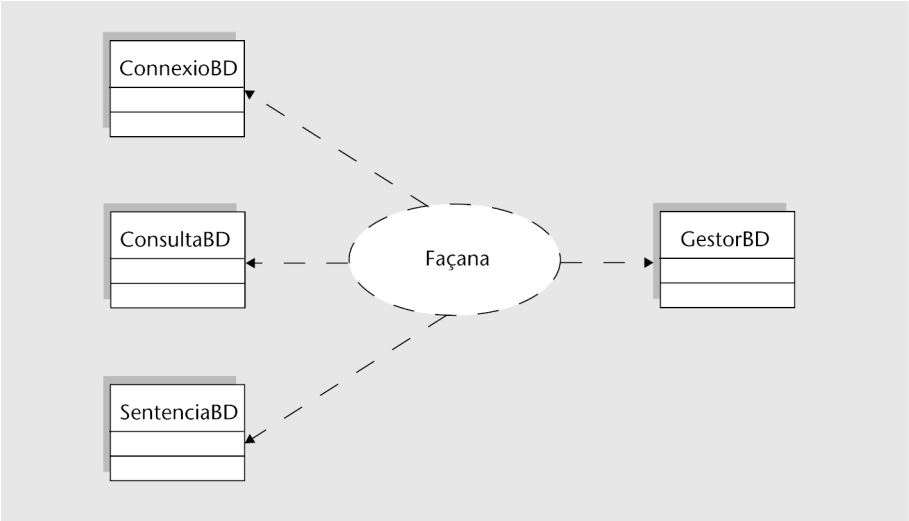


Figura 22. Exemple de modelatge del patró façana en un diagrama UML

En realitat, l'estructura d'un patró es descriu per mitjà d'una **col·laboració parametritzada**. Se sap que en cada disseny en què s'utilitza el patró façana hi ha una classe que actua com tal, però, sens dubte, aquesta classe varia d'un sistema a un altre. La col·laboració parametritzada que representa l'estructura d'un patró pot aparèixer en un diagrama de classes com un oval discontinu, amb línies discontínues unint-lo als rectangles de classe que representen les classes que són els paràmetres.

A més del patró façana, n'hi ha molts altres que són d'interès per al disseny arquitectònic dels sistemes distribuïts. En particular, ens referim als patrons adaptador i observador, que es descriuen a continuació.

2.4.1. El patró adaptador

En primer lloc, el patró **adaptador** (*adapter*) és un patró estructural que s'utilitza per a convertir la interfície d'una classe en una altra, que és l'esperada pels clients. Permet resoldre les diferències i les incompatibilitats entre la interfície que requereix un component i la que proporciona un servidor. L'adaptació pot variar des d'un mer canvi de noms en les operacions fins a haver d'implementar un conjunt totalment nou d'operacions, depenent com de semblants siguin la classe que cal adaptar i la classe objectiu.

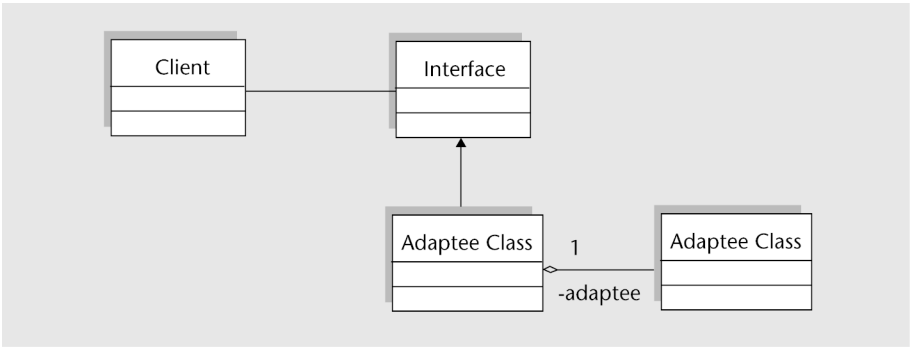


Figura 23. Estructura del patró adaptador

Associats al patró adaptador, hi ha alguns patrons, com ara:

- **Pont (*bridge*)**

Té una estructura similar, però diferent propòsit: està pensat per a separar una interfície de la seva implementació, de manera que totes dues puguin variar independentment. El patró adaptador, per contra, està pensat per a canviar la interfície d'un objecte existent.

- **Decorador (*wrapper*)**

Aquest patró s'utilitza per a afegir noves funcionalitats a un objecte concret sense canviar la seva interfície, això és, no es tracta d'afegir la funcionalitat a la classe completa d'objectes mitjançant el mecanisme d'herència, sinó d'afegir aquesta funcionalitat a un objecte concret, i de manera dinàmica en temps d'execució (a través d'un objecte *wrapper*) i deixar la resta inalterada. D'aquesta manera, el decorador aporta flexibilitat al disseny, davant el determinisme i l'aspecte estàtic d'altres alternatives.

- **Representat (*proxy*)**

De manera general, un representat és un objecte que funciona com a representant d'un altre, la interfície del qual roman invariable, al contrari del que ocorria amb el patró adaptador. L'objectiu del representat d'un objecte és re-

presentar-lo en altres sistemes o entorns, de manera que els clients pensin que estan dialogant amb l'objecte original com si aquest estigués en el seu propi sistema (i oculta, per tant, els aspectes relatius a la localització o reubicació possible). El representat senzillament coneix on és tothora l'objecte original, i hi delega les peticions dels clients.

2.4.2. El patró observador

D'altra banda, el patró **observador** (*observer*) és un patró de comportament que defineix una dependència un-a-molts entre objectes, de manera que, quan un objecte canvia d'estat, tots els altres objectes dependents es modifiquen i s'actualitzen automàticament. Aquest patró és el que normalment s'utilitza en les arquitectures orientades a esdeveniments, ja que és el que s'encarrega d'implementar els mecanismes de publicar i subscriure (*publish and subscribe*).

Aplicarem el patró observador quan un canvi en un objecte requereixi que en canviïn uns altres i es desconeix *a priori* quins i quants són, i fins i tot poden variar dinàmicament. L'objecte *observat* notificarà als seus *observadors* cada vegada que ocorre un canvi a fi que l'estat de tots dos romangui consistent. Després de ser informat d'un canvi en l'objecte observat, cada observador concret pot demanar-li la informació que necessita per a reconciliar el seu estat amb el d'aquell.

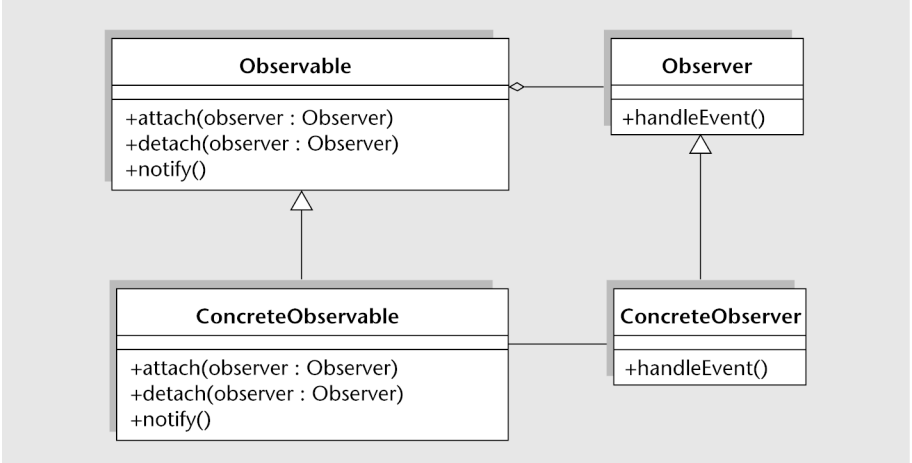


Figura 24. Estructura del patró observador

Com s'observa en el diagrama UML de la figura 24, participen en aquest patró els rols següents:

1) Observable (`Observable`): la interfície en què es defineix com poden interactuar els observadors/clients amb un `Observable`. Aquests mètodes inclouen la introducció i l'eliminació d'observadors, i un o més mètodes de notificació per enviar informació a través d'`Observable` al seu client.

2) Observable concret (`ConcreteObservable`): la classe que proporciona implementacions per a tots els mètodes de la interfície `Observable`. Necessita mantenir una col·lecció d'`Observer`.

3) Els mètodes de notificació copien la llista d'objectes `Observer`, iteren sobre la llista i criden els mètodes oients específics de cada `Observer`.

4) Observador (`Observer`): la interfície utilitzada pels observadors per a comunicar-se amb els clients.

5) Observador concret (`ConcreteObserver`): implementa la interfície Observable i determina en tots els mètodes implementats la manera de respondre als missatges rebuts d'`Observable`.

Per tant, el patró permet variar objectes observats i observadors independentment. Es poden reutilitzar els objectes observats sense els seus observadors i viceversa. I s'hi poden afegir nous observadors sense modificar cap de les classes existents. Atès l'acoblament abstracte establert entre subjectes i observadors (tot el que un objecte sap dels seus observadors és que té una llista d'objectes que satisfan la interfície *Observer*), podrien pertànyer fins i tot a dues capes diferents de l'arquitectura de l'aplicació.

Resum

El procés de disseny comporta una seqüència d'activitats i decisions que redueixen el nivell d'abstracció amb què es representa el programari. Una d'aquestes decisions té a veure amb la tria d'un estil arquitectònic per a abordar l'estructura i organització del nostre sistema. Cada estil descriu una categoria de sistemes, un conjunt de connectors que possibiliten la comunicació, la coordinació i cooperació entre els components i una sèrie de restriccions que defineixen com s'integren els components que conformen el sistema.

Seleccionat l'estil arquitectònic, el pas següent consisteix a traslladar els nostres requisits funcionals a components i connectors arquitectònics tot respectant les restriccions i característiques del patró arquitectònic seleccionat. Per a dur a terme aquesta translació, el modelatge i la documentació de l'arquitectura programari és descrita fent ús tant de diagrames estructurals com de diagrames de comportament seguint la notació d'UML 2.0.

Arribats a aquest punt, l'etapa de disseny següent se centra en el modelatge físic dels components arquitectònics mitjançant components programari implementats utilitzant-hi tecnologies concretes. Per a fer això, els diagrames previs hauran de ser refinats fins a assolir un grau d'especificació i detall més fi. Passarem de components arquitectònics de gra gruixut a la seva implementació en components programari de gra més fi; en definitiva, representacions més properes a la implementació final, com es discuteix en el mòdul següent.

Activitats

1. Dissenyeu l'arquitectura programari del sistema bancari descrit en el mòdul anterior, però on els clients poden usar no solament els serveis bancaris a través de caixers automàtics, sinó també a través d'Internet. Utilitzeu-hi un estil arquitectònic client-servidor i en tres capes. Justifiqueu el criteri utilitzat per a assignar els components a les diferents capes.

2. Dissenyeu l'arquitectura programari d'un sistema de granges de processos com pot ser el SETI@home.

Exercicis d'autoavaluació

1. Quin estil (o estils arquitectònics) triaríeu per a cada un dels supòsits que us presentem a continuació?

a) Si són centrals les dades de l'aplicació, la gestió i la representació.
b) Si el problema pot ser descompost en etapes successives, seguint un procés lineal de transformació de la informació.
c) Si el problema involucra transformacions sobre fluxos continus de dades (com poden ser vídeos) o sobre fluxos molt prolongats.
d) Si les tasques estan dividides entre productors i consumidors.
e) Si ha dissenyat un algoritme de computació abstracte, però no es vol fixar una màquina concreta per a executar-lo.
f) Si té motius per no vincular receptors de senyals amb els seus originadors.

2. Quina relació hi ha entre els conceptes estil arquitectònic, arquitectura del programari i patró de disseny?

3. Quina és la diferència fonamental entre un enfocament de client pesant i un de client lleuger per al desenvolupament de sistemes client-servidor?

Solucionari

Exercicis d'autoavaluació

1. a) L'estil seqüencial per lots o canonades i filtres, o bé una arquitectura de dipòsit.
b) Una arquitectura seqüencial per lots o canonades i filtres.
c) Una arquitectura en canonades i filtres.
d) Una arquitectura client-servidor o d'objectes distribuïts
e) Una arquitectura de màquina virtual.
f) Una arquitectura basada en esdeveniments.

2. Els estils arquitectònics defineixen invariants d'estil, que caracteritzaran grups o famílies d'arquitectures programari (les que satisfan els invariants d'estil). Els patrons de disseny conformen la microarquitectura de l'aplicació i són utilitzats en les descripcions arquitectòniques per a descriure molts dels mecanismes d'interconnexió i organització dels components.

3. En un enfocament de client pesant, algunes de les funcionalitats del sistema s'executen en el mateix client, aprofitant la potència que solen tenir els ordinadors personals, i descarregant així el servidor. D'aquesta manera, s'aconsegueix també que el servidor pugui despreocupar-se de molts aspectes "locals" al client, com el seu navegador web, el seu sistema operatiu, etc.

Glossari

arquitectura client-servidor *f* Estil arquitectònic organitzat com un conjunt de components servidors, que ofereixen uns serveis, i un conjunt de clients que utilitzen aquests serveis.

arquitectura en capes o nivells *f* Estil arquitectònic que representa una organització jeràrquica dels elements d'un sistema, de manera que cada capa proporciona serveis als elements de la capa immediatament superior (o anterior), i se serveix dels serveis que li brinden els elements de la capa immediatament inferior (o següent).

arquitectura programari *f* El conjunt de decisions, principis i regles que regeixen l'organització d'un sistema programari; la selecció dels elements estructurals que componen el sistema, les seves interfícies i els seus protocols d'interacció; les connexions d'aquests elements estructurals per a formar subsistemes cada vegada més grans; i l'estil o patró arquitectònic que guia aquesta organització.

component (arquitectònic) *m* Unitat abstracta que encapsula un estat i una funcionalitat, i que interacciona amb el seu entorn a través d'interfícies ben definides.

connector (arquitectònic) *m* Mecanisme abstracte d'una arquitectura programari que fa de mitjancer en la comunicació, coordinació o cooperació entre components.

estil arquitectònic *m* Conjunt de patrons estructurals que caracteritzen una família de sistemes, d'acord amb una sèrie de decisions essencials sobre l'estructura d'alt nivell i la composició dels sistemes, i establint restriccions important sobre els seus elements i les relacions possibles entre aquests.

Bibliografia

Albin, S. T. (2003). *The Art of Software Architecture: Design Methods and Techniques*. Indianapolis, Indiana: John Wiley & Sons.

Bass, L.; Clements, P.; Kazman, R. (1998). *Software Architecture in Practice*. Reading, MA: Addison-Wesley.

Referències bibliogràfiques

Abowd, G.; Allen, R.; Garlan, D. (1993). "Using Style to Give Meaning to Software Architecture Proceedings of SIGSOFT'93". *Software Engineering Notes* (vol. 3, núm. 118, pàg. 9-20).

Garlan, D.; Allen, R.; Ockerbloom, J. (1994). *Exploiting Style in Architectural Design Environments. Proceedings of SIGSOFT'94: Foundations of Software Engineering* (pàg. 175-188). ACM Press.

Garlan, D.; Perry, D. (1995). *Special Issue on Software Architectures IEEE Transactions on Software Engineering* (vol. 4, núm. 21, pàg. 269-274).

Perdita, S.; Pooley, R. (2002). *Utilización de UML en Ingeniería del Software con Objetos y Componentes*. Madrid: Addison Wesley.

Morris, C.; Fergunson, C. (1993). "How Architecture Wins Techology Wars". *Harvard Business Review* (núm. 71 pàg. 86-96).

Larman, C. (2002). *UML y Patrones. Una introducción al análisis y diseño orientado a objetos y al proceso unificado*. Madrid: Addison Wesley.

Pressman, Roger S. (2001). *Ingeniería del Software. Un enfoque práctico*. Madrid: Mc Graw Hill.

Shaw, M.; Garlan, D. (1996). *Software Architecture: Perspectives on an Emerging Discipline*. New York: Prentice Hall.

Sommerville, I. (2002). *Ingeniería de software*. Madrid: Addison-Wesley.