Desenvolupament de programari basat en components

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

P06/11059/01149

UOC
Universitat Oberta de Catalunya

Cap part d'aquesta publicació, incloent-hi el disseny general i la coberta, no pot ser copiada, reproduïda, emmagatzemada o transmesa de cap manera ni per cap mitjà, tant si és elèctric com químic, mecànic, òptic, de gravació, de fotocòpia o per altres mètodes, sense l'autorització prèvia per escrit dels titulars del copyright.

Índex

Introducció

En el mòdul anterior hem vist com fer el disseny d'una aplicació distribuïda, descomponent-ne la funcionalitat en diversos components arquitectònics que interactuen a través d'interfícies, i que estan connectats entre si mitjançant connectors. Arriba l'hora d'implementar aquests components, per a la qual cosa cal prendre la decisió sobre la tecnologia i els llenguatges de programació que hi utilitzarem. En aquest punt se'ns ofereixen diverses alternatives, que van des dels llenguatges de programació estructurada convencional (com Ada o C), fins als orientats a objectes (com C++ o Java), els llenguatges funcionals (com Haskell), etc.

La programació orientada a objectes ha estat el suport de l'enginyeria del programari per als sistemes centralitzats durant bastants anys i ha permès als programadors escriure (i dissenyar) els seus programes en termes d'objectes individuals que encapsulen estat (amb els seus atributs) i funcionalitat (amb els mètodes que implementen). Tanmateix, la programació orientada a objectes s'ha mostrat insuficient a l'hora d'abordar el disseny i la implementació d'aplicacions distribuïdes. En particular, s'ha vist que presenta nombroses limitacions a l'hora de tractar de manera natural aspectes propis d'aquest tipus de sistemes, com ara la concurrència, la reutilització, la composició tardana, les comunicacions remotes o l'addició modular d'aspectes com la seguretat o les transaccions. Altres deficiències importants de la programació orientada a objectes estan més relacionades amb els aspectes propis de màrqueting i van sorgir de la necessitat d'empaquetar i distribuir diferents elements programari heterogenis, poder integrar-los en una aplicació per persones diferents als desenvolupadors, poder configurar-los i adaptar-los per a diferents entorns i aplicacions, així com la necessitat d'adquirir nous elements o de reemplaçar els existents per productes d'altres fabricants.

Per tractar de pal·liar aquestes deficiències, van sorgir diferents alternatives, com per exemple la programació orientada a aspectes (http://www.aosd.net) o la programació orientada a components, que, en certa manera, estenen la programació orientada a objectes.

Un d'aquests enfocaments, la programació orientada a components, ofereix tota una sèrie de conceptes i mecanismes molt apropiats per al desenvolupament d'aplicacions distribuïdes basant-se en la reutilització d'elements (components programari) i en el seu assemblatge, per la qual cosa és la tècnica preferida per a la implementació dels components arquitectònics del mòdul anterior. La programació orientada a components és l'objecte del present mòdul.

Objectius

Aquest mòdul introdueix els principals conceptes relatius a la programació orientada a components, una extensió de la programació orientada a objectes molt útil i efectiva a l'hora de construir aplicacions programari distribuïdes d'una manera modular i escalable.

Més concretament, els objectius que persegueix aquest mòdul són els següents:

1. Donar a conèixer el concepte de component programari, les principals característiques i la relació que té amb altres conceptes similars com poden ser els objectes o els serveis web.

2. Introduir els mecanismes que proporciona el llenguatge de modelatge UML per a representar als components programari: la seva especificació, implementació, configuració i desplegament.

3. Presentar els principals processos de desenvolupament que actualment se segueixen per a construir sistemes basats en components programari. Aquests processos de desenvolupament són els que permetran "refinar" l'especificació (abstracta) dels components i connectors arquitectònics del sistema en un conjunt de components programari concrets que l'implementen.

1. El concepte de component programari

1.1. La definició de component programari

El concepte de component programari va sorgir de la necessitat de fer evolucionar el concepte d'objecte per a abordar amb èxit les necessitats plantejades pels sistemes oberts i distribuïts. Tanmateix, arribar a una definició de component programari no ha estat tasca fàcil, perquè cada autor o organització tenien originalment una manera diferent d'entendre aquest terme i de definir les característiques que ha d'exhibir. A més, el terme *component* s'ha utilitzat en el context de l'enginyeria del programari per a expressar diferents conceptes, com veurem en aquest apartat.

A continuació enumerem les principals definicions d'aquest terme ofertes per les principals autoritats en la programació orientada a components.

Component programari

Per a il·lustrar el concepte de component, utilitzarem dos exemples: un component simple de mida petita i un component comercial de mida gran.

- El primer és un component *Buffer* per a inserir i extreure caràcters que poden utilitzar la resta dels components de qualsevol aplicació.
- El segon és un component comercial i molt conegut, com és l'editor de textos MS-Word, compost al seu torn d'altres components, com per exemple un editor de textos i un altre de dibuixos, un diccionari de sinònims o un corrector ortogràfic, entre d'altres.

1.1.1. La definició del SEI

En primer lloc, l'Institut d'Enginyeria del Programari (Software Engineering Institute, SEI) de la Universitat Carnegy Mellon va proposar la definició de component programari següent:

> **Component programari**: fragment d'un sistema programari que proporciona uns serveis i que pot ser acoblat amb altres fragments per a formar peces més grans o aplicacions completes.

Per a definir el terme, es van considerar les tres perspectives des de les quals es pensava que podia observar-se un component:

a) La perspectiva **d'empaquetament**, que considera un component com una unitat d'empaquetament, distribució i lliurament.

b) La perspectiva de **servei**, que considera el component com un proveïdor de serveis.

c) La perspectiva **d'integració**, que considera el component com un element encapsulat que pot ser acoblat amb altres components per a construir una aplicació o altres components.

Microsoft Word

En el cas de l'MS-Word, els principals serveis que proporciona són els d'edició de textos i gràfics, diccionari de sinònims i corrector ortogràfic, entre d'altres. A aquests serveis poden accedir tant els usuaris humans a través de la seva interfície gràfica com altres programes a través de diferents interfícies COM. Aquest component és una unitat d'encapsulació de la funcionalitat esmentada i està compost per una sèrie de fitxers que en permeten la distribució, el lliurament i la instal·lació en qualsevol sistema Windows. Al seu torn, el Word està compost per components individuals que proporcionen cada un dels serveis esmentats abans.

En el cas del component *Buffer*, els seus serveis poden ser utilitzats per altres programes a través de les seves interfícies programàtiques. El *Buffer* proporciona un servei per a inserir i extreure caràcters, i un altre per a administrar un dels paràmetres configurables del *Buffer*, el nombre màxim de caràcters que pot emmagatzemar.

1.1.2. La definició de Clemens Szyperski

Una de les definicions més acceptades avui dia per la comunitat del programari és la proposta que va fer Clemens Szyperski el 1996, que va ser publicada el 1998 al llibre *Component Software: Beyond Object-Oriented Programming* i que va establir les bases de la programació orientada a components.

Lectura recomanada

Clemens Szyperski (2002). *Component Software. Beyond Object-Oriented Programming* (2a. ed.). Cambridge, MA: Addison-Wesley Longman.

> Un **component** (Szyperski, 1998) és una unitat de composició binària de programari, que té un conjunt d'interfícies i un conjunt de requisits i que ha de poder ser desenvolupat, adquirit, incorporat al sistema i compost amb altres components de manera independent, en temps i espai.

Les claus més importants d'aquesta definició són les que permeten establir les característiques pròpies dels components programari i que els diferenciaran d'altres entitats programari, com ara els objectes o els serveis web (vegeu l'apartat "La definició d'UML 2.0").

- En primer lloc, s'afirma que un component és una **unitat de composició**, la qual cosa indica que l'objectiu principal d'un component és integrar-se amb altres components per a formar aplicacions.

- En segon lloc, el fet de ser **binari** abstreu completament els detalls d'implementació del component, de manera que no pot (ni ha) accedir-se al seu contingut intern a l'hora d'instal·lar-lo en el sistema o de compondre'l amb altres components. És a dir, no ens han d'importar els detalls interns d'un component a l'hora d'acoblar-lo o d'interactuar-hi (no

ens ha d'importar si el *Buffer* està implementat en Java o en C++, o de manera monolítica o bé utilitzant-hi altres components més petits).

Com a conseqüència d'això, alguns mecanismes propis de l'orientació a objectes, com per exemple l'herència, deixen de ser essencials a l'hora de construir aplicacions a partir de components. Alhora, altres mecanismes com la delegació cobren més rellevància en el context del desenvolupament de programari basat en components, on la integració de peces existents se superposa al desenvolupament de nous mòduls programari.

Sens dubte, l'herència no és supèrflua en aquest context, ja que normalment molts components estan internament implementats a partir d'objectes, que sí que utilitzen aquest mecanisme; el que ocorre és que precisament la programació orientada a components abstreu aquests detalls d'implementació, que romanen ocults.

El fet que un component sigui una **unitat de composició binària** també permetrà reemplaçar un component en una aplicació per qualsevol altre que satisfaci la seva especificació, independentment del llenguatge de programació en què estigui implementat o del fabricant.

- En tercer lloc, els **requisits** (també anomenats **dependències**) determinen les necessitats del component quant a recursos (com, per exemple, les plataformes necessàries perquè funcioni correctament) o els serveis d'altres components que utilitza per dur a terme la seva funcionalitat. Això és una cosa que també diferencia els components dels objectes de la programació orientada a objectes (que només especifiquen els serveis que implementen, però no els que necessiten altres objectes).

- Finalment, una altra característica pròpia dels components que indica Szyperski és la separació entre el component programari i les aplicacions on finalment serà integrat per a exercir les seves funcions: en paraules de l'autor, un component programari ha de poder ser desenvolupat, adquirit, incorporat al sistema i compost amb altres components de manera independent, en temps i espai. De fet, el que es busca amb els components és afavorir la **reutilització** del programari. D'aquesta manera, es pretén utilitzar un mateix component en diferents aplicacions obtenint així l'estalvi consegüent en costos i esforç.

Reutilitzar

En el context del desenvolupament de programari basat en components, reutilitzar no es refereix només a "usar moltes vegades", sinó a "usar en diferents contextos" (possiblement diferents dels contextos o les aplicacions per als quals el component es va dissenyar originalment).

Per exemple, el component *Buffer* s'ha dissenyat per a poder-se integrar a qualsevol aplicació que necessiti un emmagatzemament temporal de caràcters. La reutilització és possible independentment de molts factors, com per exemple el llenguatge de programació que s'hagi fet servir per a implementar-lo, la localització física, etc.

1.1.3. Els criteris de Bertrand Meyer

Bertrand Meyer va establir el 1999 set criteris que, des del seu punt de vista, ha de complir qualsevol component programari:

1) Pot ser utilitzat per altres elements de programari.

2) Pot ser utilitzat per clients sense la intervenció del desenvolupador.

3) Inclou l'especificació de la funcionalitat que ofereix.

4) Inclou l'especificació de totes les seves dependències.

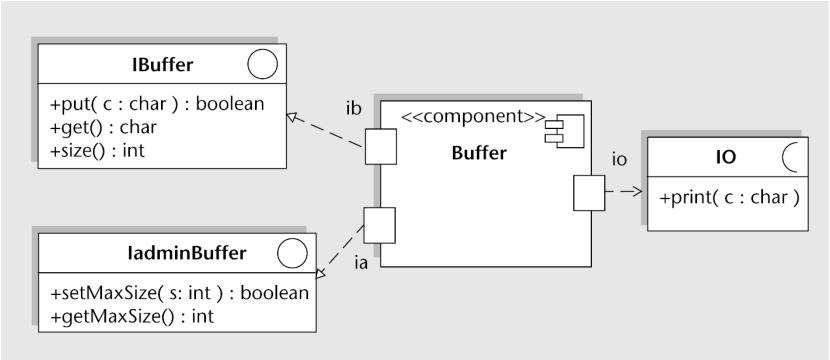
5) És utilitzable sobre l'única base de les seves especificacions.

6) Es pot compondre juntament amb altres components.

7) Pot integrar-se en un sistema ràpidament i fàcilment.

Per a Bertand Meyer, el fet de ser binari no és tan rellevant com subratlla Szyperki. El que per a ell és especialment important és el fet que el component tingui unes especificacions precises que estableixin el seu **contracte d'utilització** i que aquestes hagin de ser satisfetes per qualsevol implementació del component.

Per exemple, l'**especificació** del component *Buffer* pot ser descrita pel diagrama UML següent, que determina de manera precisa les interfícies que componen els seus serveis (**IBuffer** i **IAdminBuffer**). Així mateix, l'especificació també detalla que el component ***Buffer*** fa ús d'un servei extern (una dependència) la signatura del qual és descrita per la interfície **IO**.



UML no és l'únic llenguatge per a descriure interfícies, com veurem en els mòduls següents. Per exemple, també podem utilitzar-hi una descripció textual com la que proporciona el llenguatge de descripció d'interfícies (o IDL, sigla de l'anglès *interface definition language*) del model CCM de components CORBA (CCM és la sigla de l'anglès CORBA *component model*):

```
interface IBuffer {
boolean put(in char c);
char get();
int size();
};
interface IAdminBuffer {
boolean setMaxSize(in int s);
int getMaxSize();
};
interface IO {
void print (in char c);
};
component Buffer {
provides IBuffer ib;
provides IAdminBuffer ia;
uses IO io;
};
// Home per a instar components buffer.
home BufferHome manages Buffer { factory new(in string name); };
```

A més, per a Meyer les especificacions que només descriuen la signatura de les operacions del component no són suficients (per exemple, quina és la política d'inserció i accés als seus elements que segueix el *Buffer*: FIFO o LIFO? Puc reduir en un moment donat la mida màxima del *Buffer* per sota de la seva capacitat actual?). És per això que també cal descriure el comportament de les operacions esmentades, per a la qual cosa Meyer suggereix l'ús de precondicions i postcondicions.

Per exemple, essent **buff** la variable que emmagatzema els caràcters del *Buffer*, i **maxSize** la variable que emmagatzema la mida màxima actual del *Buffer*, les operacions de les interfícies que proporciona el component ***Buffer*** poden especificar-se en OCL com segueix:

```
context IBuffer::put(c:char):boolean
        post:  if buff->size() >= maxSize then result = false
        else buff = buff@pre.concat(c) and result = true
context IBuffer::get(): char
        pre: buff->notEmpty()
        post: result = buff->first()
context IBuffer::size() : int
        post: result = buff->size()
context IAdminBuffer:: setMaxSize( s:int ) : boolean
        post:  if buff->size()> s then result = false
        else maxSize = s and result = true
context IAdminBuffer:: getMaxSize() : int
        post: result = maxSize
```

(Observeu l'ús que es fa de les operacions que ofereix OCL per al maneig de seqüències i col·leccions d'elements, com ara size(), notEmpty(), concat() i first(). A més, el sufix "@pre" en una postcondició serveix per a referir-se al valor d'una variable en el moment de la precondició, mentre que la variable predefinida "result" serveix per a representar el valor que serà tornat per una funció).

Per a Bertrand Meyer, un component pot presentar-se en forma de codi font o codi objecte (és a dir, no importa si el component és "binari" o no), pot estar escrit en un llenguatge funcional, procedimental o en un llenguatge orientat a objectes, i pot ser tan simple com una finestra d'usuari o una aplicació completa (és a dir, no ha de tenir necessàriament una granulositat determinada).

1.1.4. La definició de Cheesman i Daniels

John Cheesman i John Daniels, en el seu llibre *Componentes UML*, van observar que el terme ***component*** pot referir-se a diferents conceptes i d'això prové la confusió regnant entre les persones que intentaven arribar a un consens sobre el seu significat i la seva definició. És a dir, mentre que en la programació orientada a objectes es distingeix entre el concepte de classe i d'objecte, en la programació orientada a components el terme *component* solia emprar-se tant per a referir-se a l'especificació com a la instància d'un component.

Aquests autors identifiquen diferents conceptes associats al terme *component* i que apareixen en les diferents etapes del procés de desenvolupament programari:

- Especificació del component
- Implementació
- Component instal·lat
- Instància de component

Ens podem referir a totes amb el terme *component*, encara que el correcte és referir-se a cada un d'aquests conceptes amb el seu nom específic. Passem a descriure cada un d'aquests conceptes.

1) L'especificació d'un component representa l'especificació d'una unitat de programari i descriu tant els serveis que proporciona, com els que necessita d'altres components, així com el comportament de qualsevol instància del component que respecti l'especificació.

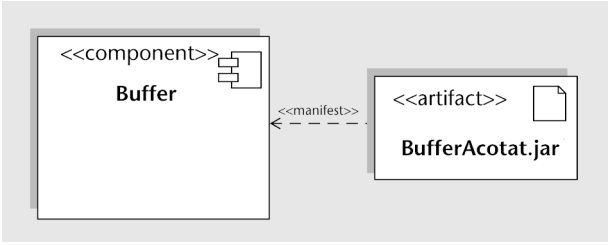
En el cas del *Buffer*, l'especificació del component és la que s'ha descrit anteriorment (mitjançant el diagrama UML anterior o usant-hi l'IDL de CCM), que descriu tant els serveis que proporciona el component com els que necessita per a funcionar. En el cas del component d'MS-Word, l'especificació de tots les seves interfícies (que són més de 70.000) no és pública, fet que en complica la utilització per desenvolupadors externs a Microsoft. Aquest és un dels principals desavantatges dels sistemes no oberts, tal com havíem comentat en el mòdul 1.

El concepte d'especificació de component programari tal com el defineixen Cheesman i Daniels coincidirà amb el concepte de component arquitectònic descrit en el mòdul anterior i que, al seu torn, coincideix amb l'especificació dels objectes computacionals d'RM-ODP.

Cada una d'aquestes "especificacions de component", o components arquitectònics, defineix els components abstractes que formen part del disseny de l'aplicació. Aquests components seran implementats després pels components programari concrets (o artefactes) que formen part de la implementació del sistema, encara que no hi ha d'haver una correspondència un a un entre ells. És a dir, un component arquitectònic pot ser implementat per diversos components concrets interconnectats entre si, que conjuntament proporcionin els serveis que ofereix l'especificació del component esmentat. O dit d'una altra manera, a cada component arquitectònic correspon un component programari (amb les mateixes interfícies que l'arquitectònic), però que internament pot descompondre's en diferents components (vegeu més endavant, per exemple, el diagrama de la figura 1).

2) La **implementació d'un component** (també denominat **artefacte**) és la realització d'una especificació de component que pot ser implantada, instal·lada i reemplaçada de manera independent en un o més arxius i pot dependre d'altres components.

En el cas del *Buffer*, podem disposar d'una implementació en Java mitjançant un vector de caràcters, el codi i els fitxers executables del qual estan inclosos en un fitxer anomenat "BufferAcotat.jar". En UML això es representa mitjançant una associació de dependència amb l'estereotip "manifest", entre l'"artefacte" BufferAcotat.jar (que conté la implementació del component) i l'"especificació de component" *Buffer*.



Una especificació de component ha de ser "feta" per (almenys) una implementació de component. Sens dubte, hi pot haver més d'una "realització" de la mateixa especificació d'un component, en cas de disposar-se de diferents implementacions d'aquest. Les implementacions esmentades poden ser per a plataformes diferents –Java EE, .NET o CCM– o estar implementades mitjançant estructures de dades diferents per qüestions de consum de recursos o eficiència.

Per exemple, en el cas del *Buffer* podem tenir una implementació no delimitada utilitzant llistes enllaçades i una altra de delimitada usant un *array* de caràcters, totes dues en Java.

Així mateix, i tal com hem esmentat abans, una especificació del component pot estar implementada per més d'un artefacte, que conjuntament proporcionen els serveis que defineixen l'especificació del component.

Per exemple, és possible implementar el *Buffer* mitjançant tres artefactes (o implementacions de component) diferents, que tots plegats proporcionen la funcionalitat adequada, tal com es mostra e la figura a continuació.

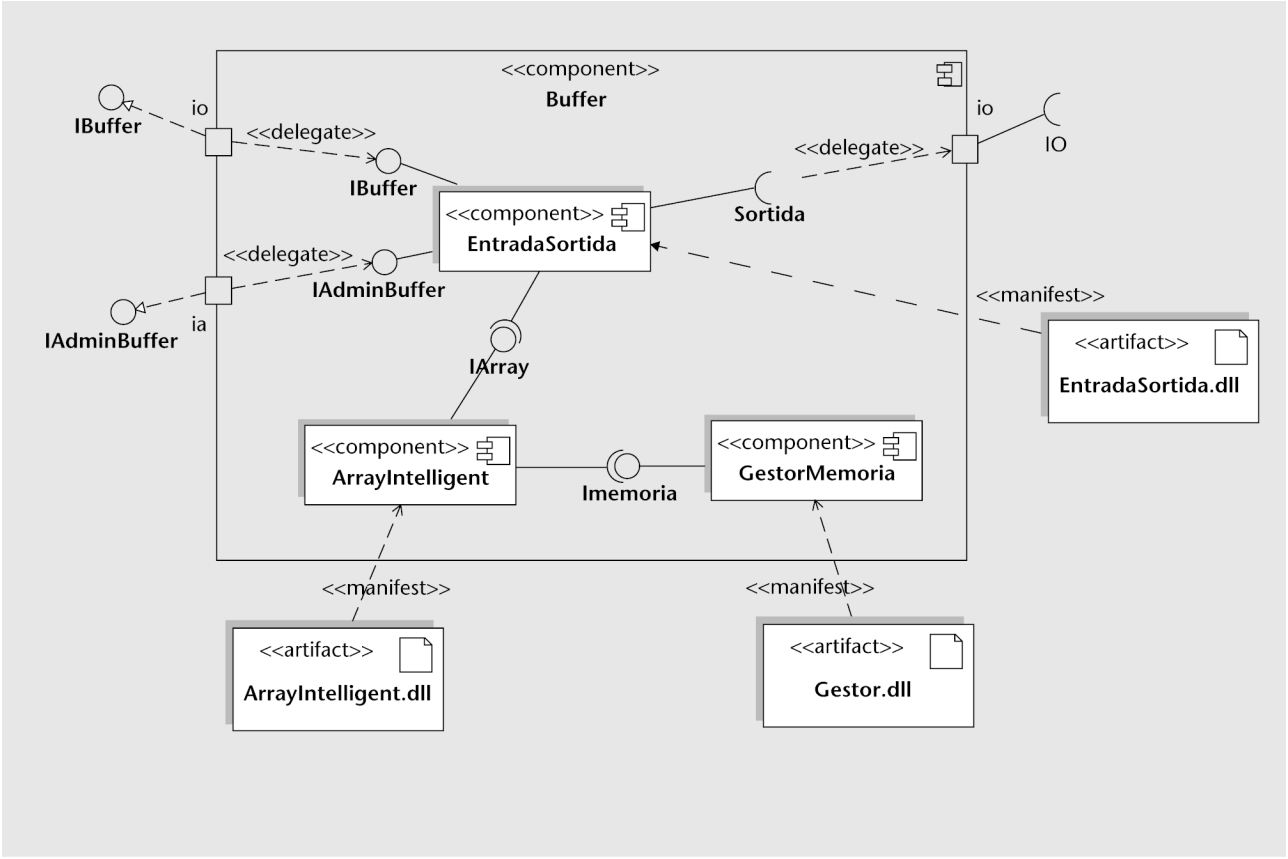


Figura 1. Una implementació possible de l'especificació del component *Buffer* mitjançant tres components programari

En la figura 1 es mostren tant els artefactes (ArrayIntelligent.dll, EntradaSortida.dll i Gestor.dll), com les especificacions de component (representades mitjançant els components UML **EntradaSortida**, **ArrayIntelligent** i **GestorMemoria**), i com aquests s'uneixen i es relacionen per a proporcionar els serveis especificats per al component *Buffer*.

De la mateixa manera, l'especificació del component Word està implementada per un conjunt d'artefactes molt nombrosos, que inclouen, per exemple, l'editor de textos, el corrector ortogràfic, l'editor de dibuixos, l'editor de taules, etc. Cada una d'aquestes "peces" que componen el component Word són, al seu torn, components que poden ser utilitzats per a compondre altres aplicacions (de fet, moltes s'usen en altres components d'MS Office, com Excel o Access).

També cal distingir entre la **implementació** d'un component i la **instal·lació** que se'n fa en una màquina (o tipus de màquina) concret:

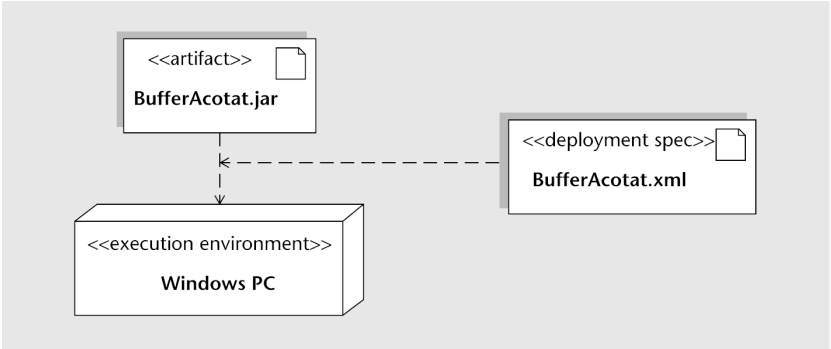
3) Un **component instal·lat** és una còpia instal·lada d'una implementació de component en una màquina concreta.

A l'hora d'instal·lar un component, també cal proporcionar la informació que permet el desplegament dels fitxers corresponents (tant els executables com els de configuració) i, a més, particularitzar la instal·lació segons els requisits, les característiques i les preferències particulars de l'usuari final. Això es du a terme en el que es denomina la fase de **desplegament**.

En el cas del Word, distingirem entre els "artefactes" que vénen al CD d'instal·lació del producte i la còpia del Word, que s'instal·la i es configura a la nostra màquina i que està composta per alguns dels fitxers del CD (però no tots), configurats d'acord amb la nostra màquina, els requisits i les preferències particulars. Per exemple, durant la fase de desplegament del component Word, podem seleccionar-ne la configuració interna, triant entre altres coses els components (artefactes) que formaran part del component instal·lat a la nostra màquina (si volem instal·lar a la nostra màquina el corrector ortogràfic, el separador de paraules amb guions, el corrector de gramàtica, etc.).

En UML s'utilitza un diagrama especial per a representar aquesta informació denominat diagrama de desplegament.

El diagrama següent mostra el diagrama de desplegament en el cas del *Buffer*, on BufferAcotat.xml és un fitxer amb la informació necessària per a instal·lar l'artefacte "BufferAcotat.jar" en un PC amb Windows.



4) Finalment, una instància de component és una instància d'un component instal·lat. Normalment està compost per una col·lecció d'objectes amb el seu propi estat i identitat única que du a terme el comportament implementat.

En una màquina podem tenir diverses instàncies del component *Buffer* o del component Word executant-se alhora. Totes són instàncies del mateix component (l'instal·lat a la nostra màquina), encara que cada una té un estat propi diferent (cada instància de component *Buffer* maneja un conjunt de caràcters diferents; i cada instància del component Word està editant un fitxer diferent, té el cursor en un lloc diferent, té opcions d'edició diferents, etc.).

Observeu que molts autors han utilitzat un sol terme (*component programari*) per a referir-se a tots aquests conceptes, i això és la causa de la confusió. Segons Cheesman i Daniels, el concepte de component no pot ser vist com un únic terme, sinó que ha de ser diferenciat d'acord amb l'etapa del cicle programari a què ens referim.

Una altra distinció molt important que s'ha d'efectuar és entre l'"especificació" i la "instància" del component, diferència que és anàloga a la que hi ha entre una "classe" i un "objecte" en la programació orientada a objectes. Observi's a més que els altres dos conceptes (implementació del component i component instal·lat) no tenen parangó en la programació orientada a objectes, ja que no necessita identificar-los.

1.1.5. La definició d'UML 2.0

En versions anteriors d'UML, els components es consideraven com una representació d'estructures físiques, com ara DLL, executables, etc. Tal com havíem esmentat en el mòdul anterior, en UML 2.0 passen a representar els components en l'àmbit arquitectònic, alhora que la versió 2.0 d'UML incorpora tota una sèrie de conceptes, mecanismes i diagrames per a l'especificació i el disseny de sistemes basats en components programari (connectors, artefactes, realitzacions, nodes, etc.).

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

UML 2.0 posa èmfasi en el fet que els detalls interns del component (pel que fa a la implementació i l'estructura interna) romanen ocults, i que les interaccions amb altres components es realitzen a través d'interfícies que normalment estan associades a ports. UML 2.0 també distingeix entre les interfícies que el component **implementa** i les que **requereix** (o utilitza) d'altres components.

Els conceptes que usa UML 2.0 són molt similars als que van definir Cheesman i Daniels, on un component UML 2.0 correspon a una "especificació de component". A més, UML 2.0 defineix una sèrie de diagrames per a especificar totes les fases diferents que intervenen en el desenvolupament de programari basat en components. Els diagrames de nivell més alt, i que serveixen per a descriure l'arquitectura programari de les aplicacions, ja s'havien explicat en el mòdul 2.

Vegeu també

En l'apartat "Representació de components programari en UML" d'aquest mòdul veurem els diagrames relatius a la implementació i el desplegament de les arquitectures esmentades a partir d'instàncies de components programari concrets.

1.2. Models i plataformes de components

Una vegada disposem del concepte de component, un model de components estableix la manera concreta com un fabricant o una organització defineix la forma de les interfícies dels seus components i els mecanismes que permeten interconnectar-los.

Alguns exemples de models de components inclouen COM, JavaBeans, CORBA o CCM, que seran descrits en els mòduls 4 i 5.

Basada en un model de components concret, una **plataforma de components** és un entorn de desenvolupament i d'execució de components que permet aïllar la major part de les dificultats conceptuals i tècniques que comporta la construcció d'aplicacions basades en els components d'aquest model.

En aquest sentit, podem definir una **plataforma de components** com una implementació dels mecanismes del model, juntament amb una sèrie d'eines associades a aquest.

La figura 2 resumeix les diferències fonamentals entre els conceptes de model i plataforma de components.

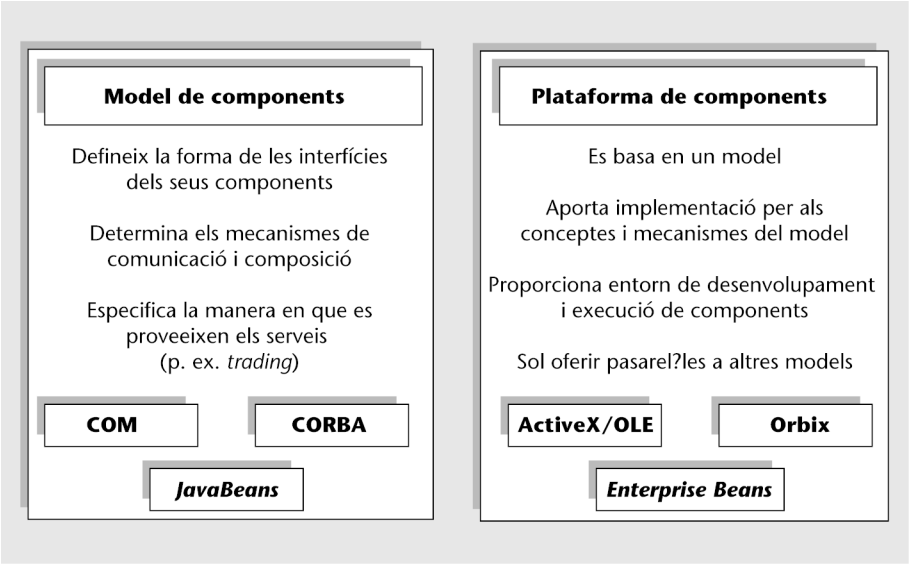


Figura 2. Relació entre models i plataformes de components

Són exemples de plataformes de components ActiveX/OLE, Java EE i Orbix, que es recolza en els models de components COM, JavaBeans i CORBA, respectivament. En l'apartat següent descriurem alguns conceptes bàsics sobre els quals es recolzen les plataformes de components distribuïts (interfícies, contenidors, metainformació, serveis comuns, etc.). Bàsicament, totes les plataformes de components incorporen aquests conceptes i ofereixen una sèrie de serveis als seus components. El que varia entre unes i altres és la manera com els implementen i les prestacions que ofereixen.

En els mòduls 4 i 5 es discutiran detalladament les principals plataformes actuals de components distribuïts.

D'altra banda, un mercat global necessita també estàndards que garanteixin la interoperabilitat dels components a l'hora de construir aplicacions. En el món dels sistemes oberts i distribuïts estem presenciant la clàssica "guerra" d'estàndards que succeeix abans de la maduració de qualsevol mercat. Tots els fabricants i venedors de sistemes tracten de convertir els seus mecanismes en estàndards, alhora que pertanyen a diversos consorcis que també tracten de definir estàndards, però de manera independent. Pensem per exemple en Sun, que intenta imposar JavaBeans com a model de components, mentre que participa juntament amb altres empreses en el desenvolupament dels estàndards de CORBA. Actualment la interoperabilitat es resol mitjançant passarel·les (*bridges*) entre uns models i altres, components especials que s'encarreguen de servir de connexió entre els diferents components heterogenis. La majoria dels models comercials ofereixen passarel·les cap a la resta, ja que reconeixen la necessitat de ser oberts. Així, és possible per exemple integrar un component Java EE en un sistema de components CCM sense problemes.

1.3. Relació amb altres conceptes

Amb vista a aclarir el que s'entén per component programari, és també important tractar de relacionar aquest concepte amb uns altres de similars que es manegen en l'enginyeria del programari, com són els d'objecte i servei web. La taula següent mostra una comparació entre els conceptes d'objecte, instància de components i servei web, atenent diversos criteris.

	Objectes	**Instàncies de components**	**Serveis web**
Especifiquen les seves dependències?	No	Sí, mitjançant les interfícies requerides	Sí, mitjançant les operacions de sortida descrites en WSDL
Perspectiva arquitectònica	Elements **interns** d'un component	Elements **interns** d'un sistema	Elements que es veuen **des de fora** del sistema
Model de desplegament	Juntament amb l'aplicació	Desplegament "físic" ***(install-and-use)***	El programari "existeix" en algun indret ***(connect-and-use)***
Nivells d'intercanvi d'informació	Majoritàriament **dins** d'un component programari	Majoritàriament **dins** de l'empresa	Majoritàriament **entrediverses** empreses
Nivells d'acoblament	Fort	Feble	Molt feble
Comunicació	Directa	Programari intermediari(p.ex., IIOP)	A través del web(SOAP/XML sobre HTTP)
Granularitat	Petita	Mitjana-gran	Mitjana-gran

2. Conceptes i mecanismes bàsics

En els apartats següents detallarem alguns dels conceptes més rellevants relacionats amb el desenvolupament de programari basat en components, entre els quals destaquem l'ús d'interfícies i esdeveniments, els contractes, la vinculació dinàmica, la reflexió, els contenidors i els serveis comuns.

2.1. Interfícies i esdeveniments

Les interfícies d'un component determinen tant les operacions que el component implementa, com les que necessita utilitzar d'altres components durant la seva execució.

En els models de components habituals, cada interfície serà determinada pel conjunt d'atributs i mètodes públics que el component implementa, així com pel conjunt d'esdeveniments que emet.

CCM

CCM (el model de components de CORBA, CORBA *component model*) distingeix entre cinc tipus d'interfícies o punts d'interacció (*features*):

- Els **atributs** són propietats configurables del component.
- Les ***facets*** són interfícies que agrupen les definicions de les operacions que proporciona (implementa) el component.
- Els ***receptacles*** són interfícies que defineixen les operacions que el component requereix d'altres components.
- Els ***event sources*** defineixen esdeveniments que el component pot produir.
- Els ***event sinks*** defineixen esdeveniments que el component pot consumir.

Els components CCM tenen, a més, una interfície especial (***home***) per a crear i gestionar les instàncies corresponents.

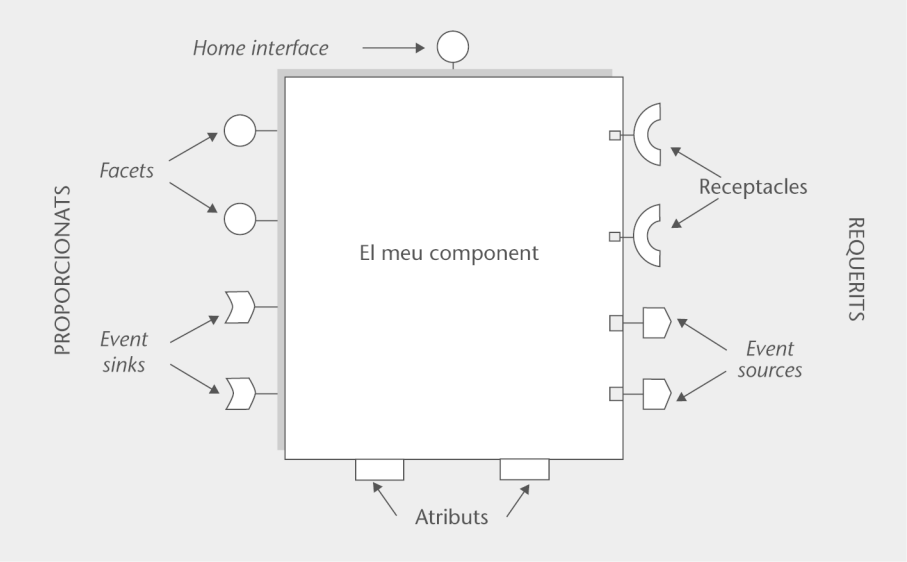


Figura 3. Diferents tipus d'interfícies d'un component CCM

Per a poder reutilitzar els components en diferents contextos, es fa especialment necessària la descripció d'aquestes interfícies, per la qual cosa sorgeixen amb aquest propòsit els **llenguatges de descripció d'interfícies** (*interface description languaje*, IDL). Aquests llenguatges proporcionen la informació que permet manejar els components com a caixes negres i, per tant, sense necessitat de conèixer ni d'accedir als detalls de la seva implementació.

Els **esdeveniments** especifiquen la manera com un component notifica a l'exterior una resposta a un estímul extern o bé un canvi en una condició interna (per exemple, la modificació de l'estat d'una variable). En la interfície d'un component s'especifica tant la signatura de l'esdeveniment com la condició que s'ha de satisfer perquè aquest es produeixi, però sense indicar el consumidor de l'esdeveniment ni la manera com s'ha de tractar, per tal com són detalls que el component no pot ni ha de conèixer.

Un **esdeveniment** és un mecanisme de comunicació pel qual es pot informar de les situacions que ocorren en un sistema de manera asíncrona. Els esdeveniments solen ser emesos pels components per a avisar altres components del seu entorn de canvis en el seu estat o de circumstàncies especials, com per exemple les excepcions.

El mecanisme d'interacció tradicional entre components està basat en **RPC** (*remote procedure call*), per a la invocació dels mètodes públics, i en el model del patró observador per als esdeveniments. En aquest model, la comunicació entre emissors i receptors es pot dur a terme tant de manera directa, com indirecta. En el primer cas, els receptors es registren en l'emissor, que envia els esdeveniments als receptors que tingui registrats en el moment en què es produeixi l'esdeveniment. Però també poden utilitzar-s'hi **distribuïdors**, entitats a què l'emissor envia l'esdeveniment perquè sigui distribuït als receptors

Vegeu també

El patró observador, juntament amb els patrons de disseny més rellevants, s'estudien en l'assignatura *Enginyeria del programari orientat a objectes*.

potencials. Els receptors poden registrar-se amb els distribuïdors, o bé aquests últims poden utilitzar estratègies de difusió total (*broadcast*) o parcial (*multicast*) per a comunicar les ocurrències d'un esdeveniment als receptors.

2.2. Contractes i *trusted components*

Un contracte és una especificació que s'afegeix a la interfície d'un component que estableix les condicions d'ús i implementació que relacionen els clients i els proveïdors del component.

Els contractes cobreixen aspectes tant *funcionals* (semàntica de les operacions de la interfície) com *no funcionals* (per exemple, qualitat de servei, prestacions, fiabilitat o seguretat).

Lectura recomanada

A. Beugnard; J. M. Jezequel; N. Plouzeau; D. Watkins (1999, juliol). "Making components contract-aware". *IEEE Computer*.

És possible establir quatre nivells diferents de contractes d'utilització, cada un més exigent que l'anterior:

1) Contractes bàsics (nivell 1)

Estableixen la signatura de les operacions que du a terme un component, els paràmetres d'entrada i sortida i les excepcions possibles. Es tracta, per tant, de contractes des del punt de vista sintàctic, que no inclouen cap informació sobre el comportament esperat, els protocols d'accés als mètodes, les seves precondicions, etc. Per a descriure aquest tipus de contractes se solen utilitzar els llenguatges de descripció d'interfícies (IDL) o bé notacions gràfiques com UML.

2) Contractes de comportament (nivell 2)

Tracten d'especificar el comportament de les operacions utilitzant assercions booleanes (precondicions i postcondicions) i invariants per a cada servei ofert, de manera similar al que es fa en la tècnica de disseny per contracte (*design by contract*). Així, el client només invocarà operacions del proveïdor en estats en què els invariants i les precondicions d'aquestes operacions es respectin. Com a resultat, el proveïdor assegura que el retorn de l'operació complirà les postcondicions especificades i els invariants de classe. Depenent de com de restrictives siguin les postcondicions, es pot permetre en aquest nivell un cert grau de negociació.

Design by contract

Terme encunyat originalment per Bertrand Meyer en el domini de la programació orientada a objectes (inicialment amb el llenguatge Eiffel). Aquest enfocament requereix que el dissenyador especifiqui de manera precisa cada condició que pogués danyar la consistència del sistema, així com assignar la responsabilitat del seu compliment tant a la rutina que invoca (*client*), com a la responsable de la seva implementació (*proveïdor*).

3) Contractes de sincronització (nivell 3)

Els contractes de comportament assumeixen que els serveis són atòmics o que s'executen com a transaccions. Tanmateix, això no sempre és realista; en aquest nivell s'especifica el comportament global del sistema en termes de sincronització entre crides a mètodes. L'objectiu és descriure l'ordre parcial i les dependències entre els serveis oferts pels components.

4) Contractes de qualitat de servei (nivell 4)

Podem necessitar quantificar el comportament esperat d'un servei o oferir una manera de negociar aquests valors. Per a fer això, es pot especificar un contracte de qualitat de servei (QoS, *quality of service*) estàtic enumerant una a una les característiques (per exemple, temps de retard màxim de resposta, *throughput*[1], etc.) que esperem que respecti el servidor d'un servei. Unes altres vegades, aquest control de qualitat pot implementar-se en manera dinàmica mitjançant la negociació entre client i servidor.

(1) *Throughput*: quantitat de feina que un ordinador és capaç de processar per unitat de temps.

El **disseny per contractes** és una manera de fer el disseny d'un sistema de programari orientat a objectes o a components, que consisteix a assignar a cada objecte o component un contracte (almenys de nivell 2) que n'especifica la funcionalitat. A partir d'aquest punt, la responsabilitat de comprovar les precondicions de qualsevol operació recau en el programa o l'usuari que invoca l'operació (i allibera, per tant, el component de comprovar-les), alhora que qualsevol implementació del component es compromet a respectar les postcondicions imposades pel contracte. D'aquesta manera, els contractes assignen responsabilitats i serveixen per a evitar duplicitats en les comprovacions de les precondicions i postcondicions.

El disseny per contracte en l'àmbit del desenvolupament de programari basat en components dóna lloc al que es coneix com a *trusted component*:

> Un ***trusted component*** és un component programari juntament amb l'especificació dels contractes associats a les seves interfícies, que garanteix que la implementació del component proporciona els serveis especificats en el contracte sempre que l'usuari del component satisfaci les condicions requerides per a utilitzar-lo.

2.3. Reutilització

El desenvolupament de programari basat en components canvia el focus i l'orientació dels processos implicats en la construcció d'aplicacions programari. L'objectiu principal deixa de ser el desenvolupament des de zero, que tracta d'evitar-se mitjançant la **reutilització** de components i l'**assemblatge** i la **in-**

terconnexió posterior per a formar l'aplicació final. Per tant, apareixen mecanismes i operacions pròpies d'aquest tipus de desenvolupament, com són la composició tardana, la vinculació dinàmica i la delegació.

La **reutilització** no solament tracta d'evitar els esforços d'haver d'implementar i provar una cosa ja desenvolupada prèviament per uns altres, sinó que persegueix també el somni anhelat de poder tenir un mercat global de components programari, igual com existeix ja per a altres disciplines (com, per exemple, per al maquinari). Aquest mercat estaria compost per proveïdors i clients; els primers són les empreses que produeixen components programari, els quals són adquirits pels clients per a construir les seves aplicacions. Així, el sistema es construiria com si es tractés de peces d'un puzle.

A manera il·lustrativa, a la figura 4 veiem com el desenvolupament de programari ha anat evolucionant des de la visió monolítica cap a una visió composicional, així com els mecanismes que han permès estructurar i ordenar la interacció del programari amb el seu entorn.

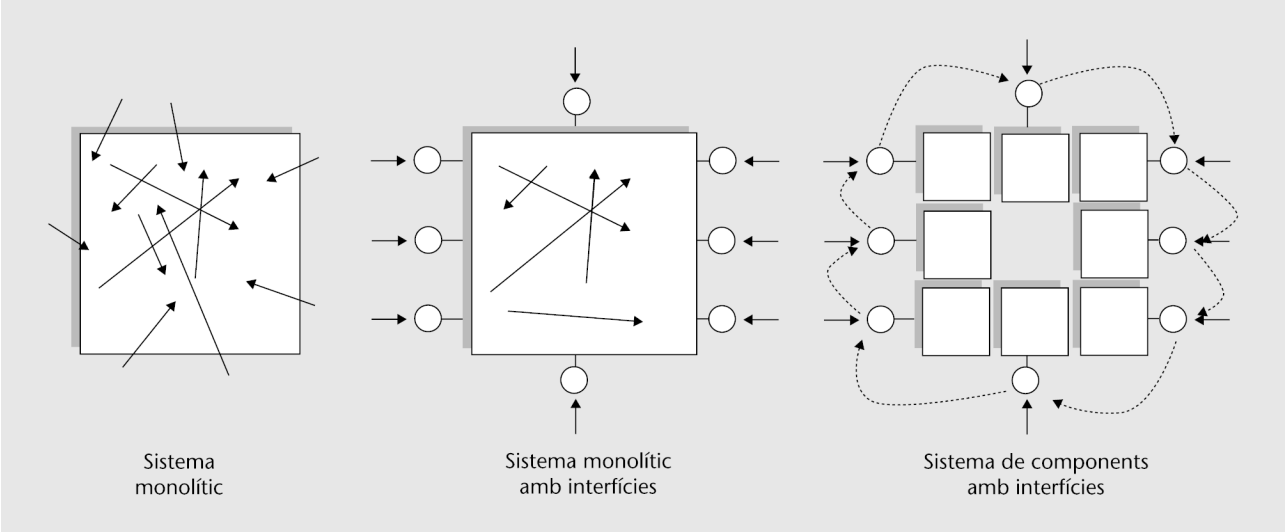


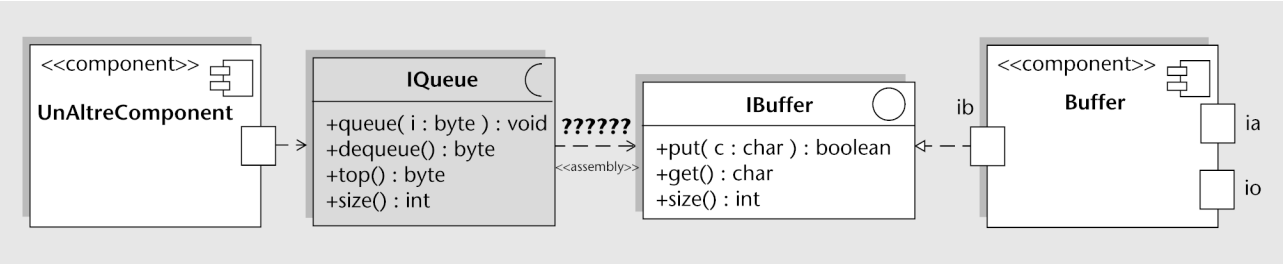
Figura 4. Evolució del programari cap a la visió composicional

El terme *reutilitzar* no es refereix únicament al fet de poder fer ús d'un mateix component tantes vegades com es vulgui. En aquest cas, els components han de ser reutilitzables en diferents contextos, la qual cosa ens permetria fer ús de parts de programari ja desenvolupades i provades en diferents moments, llocs i amb diferents finalitats. Per a poder efectuar aquest tipus de reutilització contextual, es fa especialment important la descripció de l'especificació dels components, i en particular de les seves **interfícies** (tant proporcionades com requerides).

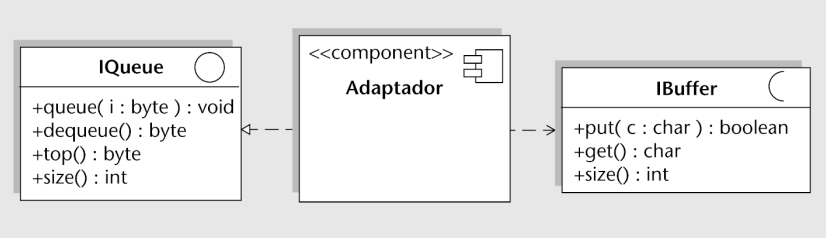
Moltes vegades no és possible reutilitzar el component tal com el trobem al dipòsit o l'adquirim del proveïdor, sinó que cal adaptar-lo lleugerament a les necessitats concretes de la nostra aplicació (per exemple, els noms de les operacions o els tipus dels seus arguments no coincideixen amb els que s'han especificat en l'arquitectura programari del sistema). En aquest cas, cal fer un

procés **d'adaptació**, per al qual normalment s'utilitzen adaptadors (*wrappers*), que són elements programari que resolen aquestes diferències (per exemple, interceptors associats als connectors, o bé altres components programari).

Per exemple, suposem un component (denominat **UnAltreComponent**) que necessita els serveis d'un tipus abstracte de dades Cua per a emmagatzemar *bytes* d'informació, d'acord amb la interfície IQueue mostrada en la figura següent:



El problema és que la interfície requerida per aquest component no coincideix completament amb la interfície ***IBuffer*** proporcionada pel component ***Buffer***. Tanmateix, és possible reutilitzar els serveis d'aquest últim component si definim un petit component Adaptador que serveix per a resoldre aquestes diferències, i intervé entre els components ***UnAltreComponent*** i ***Buffer***.



El component ***Adaptador*** implementa els serveis definits en la interfície ***IQueue*** fent ús dels serveis definits en la interfície ***IBuffer***. Des del punt de vista del component ***UnAltreComponent***, l'adaptador serveix d'"embolcall" al component ***Buffer*** (d'on deriva el nom de *wrapper*), adaptant les seves diferències per a fer-lo compatible amb els seus requisits. A més, la implementació del component ***Adaptador*** és molt simple, ja que davant de les invocacions a les operacions de la interfície IQueue es limita a invocar les operacions de la interfície ***IBuffer***. Això s'anomena **delegació**, un mecanisme molt habitual en la programació orientada a components i que discutirem en el proper apartat.

L'adaptació sol ser fàcil quan l'únic que canvia són els noms de les operacions o els tipus dels seus arguments d'algunes interfícies. Tanmateix, no ho és tant quan es produeixen un altre tipus d'incompatibilitats entre els components que han de ser adaptats.

Per exemple, no seria fàcil fer adaptadors per al *Buffer* si els elements que necessitem emmagatzemar fossin cadenes de caràcters (*strings*) o en comptes d'una Cua necessitéssim una Pila (és a dir, una estructura FIFO en comptes de LIFO).

Reutilitzar enfront de desenvolupar

Els experts solen dir que si cal adaptar més del 20% de la funcionalitat d'un component per a integrar-lo en la nostra aplicació, al final surt més còmode i barat desenvolupar-lo nosaltres mateixos, en comptes de tractar de reutilitzar-lo.

En general, la decisió de reutilitzar un component davant la de desenvolupar-lo nosaltres mateixos des de zero no és una decisió fàcil, ja que els costos d'adaptació poden superar fins i tot els de desenvolupament. Aquesta decisió és una de les que cal prendre en qualsevol procés de desenvolupament de programari basat en components.

2.4. Composició tardana, vinculació dinàmica i delegació

Vegeu també

Podeu veure un exemple de procés de desenvolupament de programari basat en components en el subapartat "Activitats del procés de desenvolupament", on es descriu el procés de desenvolupament de Cheesman i Daniels.

La **composició tardana** és una característica important dels components programari. Aquesta modalitat de composició és la que es fa posteriorment a la compilació del component, com pot ser durant l'enllaçat, la càrrega o l'execució, i per algú aliè al procés de desenvolupament, és a dir, que només coneix el component per la seva interfície o contracte, però no pel seu disseny o implementació.

També cobra importància en el desenvolupament de programari basat en components la fase **d'assemblatge**, que permet compondre l'arquitectura de l'aplicació final establint les connexions entre els components que la formen. Aquesta visió composicional del programari possibilita reutilitzar no solament els components individuals, sinó també les relacions d'interacció (o connectors) que s'estableixin entre aquests.

Vegeu també

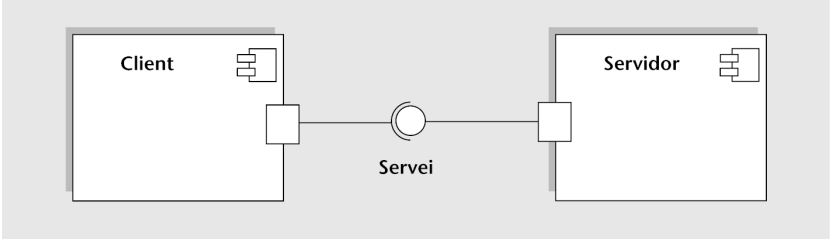
En el punt de vista computacional d'RM-ODP es defineixen les interfícies computacionals, com es pot veure en el mòdul 1, apartat 4.3.3.

En RM-ODP, les interaccions entre interfícies computacionals només són possibles si s'estableix un vincle (*binding*) entre aquestes. En el cas d'interfícies operacionals o de flux, el llenguatge computacional especifica accions per al vincle **explícit**. Aquestes accions poden ser de dos tipus:

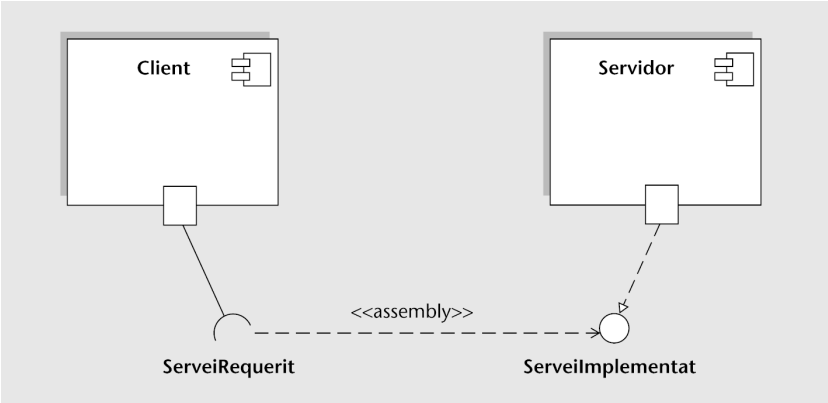
- Les accions de **vincle primitiu** (*primitive binding*) permeten vincular dues interfícies diferents, sempre que ambdues interfícies siguin del mateix tipus i actuïn amb rols complementaris (per exemple, una és client i l'altra, servidor). Aquesta acció la durà a terme un dels objectes involucrats i té com a efecte establir en cada interfície la informació necessària perquè ocorri la interacció (per exemple, identificar quina altra interfície participa en el procés d'interacció).

  RM-ODP considera que, en el cas del vincle primitiu, no cal considerar explícitament una operació de desvinculació (*unbinding*), ja que eliminant la interfície s'eliminarà també el vincle establert.

  En UML 2.0 és possible establir un vincle primitiu entre dos components de dues maneres. En primer lloc, mitjançant una interfície comuna que un dels components implementa (*realization*) i un altre utilitza (*usage*), com es mostra, per exemple, en el diagrama següent, on el component **Servidor** implementa els mètodes descrits en la interfície, **Servei** que utilitza el component **Client**.



  Una altra manera de descriure vincles primitius entre components UML 2.0 que descriuen separadament les interfícies que usen o implementen és mitjançant "connectors", que en UML 2.0 es representen mitjançant associacions de delegació entre les interfícies (que poden ser estereotipades com a *assembly*).



- Les accions de **vincle compost** (*compound binding*) permeten vincular dues o més interfícies del mateix o diferent tipus per mitjà d'un **objecte de vincle** (*binding object*), com el que es mostra en la figura 5. En aquest cas, un dels objectes computacionals involucrats en el vincle (o un objecte independent destinat per a això) crearà l'objecte destinat a establir el vincle. Posteriorment, aquest objecte de vincle instarà un conjunt apropiat d'interfícies i utilitzarà el vincle primitiu per a establir un procés d'interacció entre les interfícies participants.

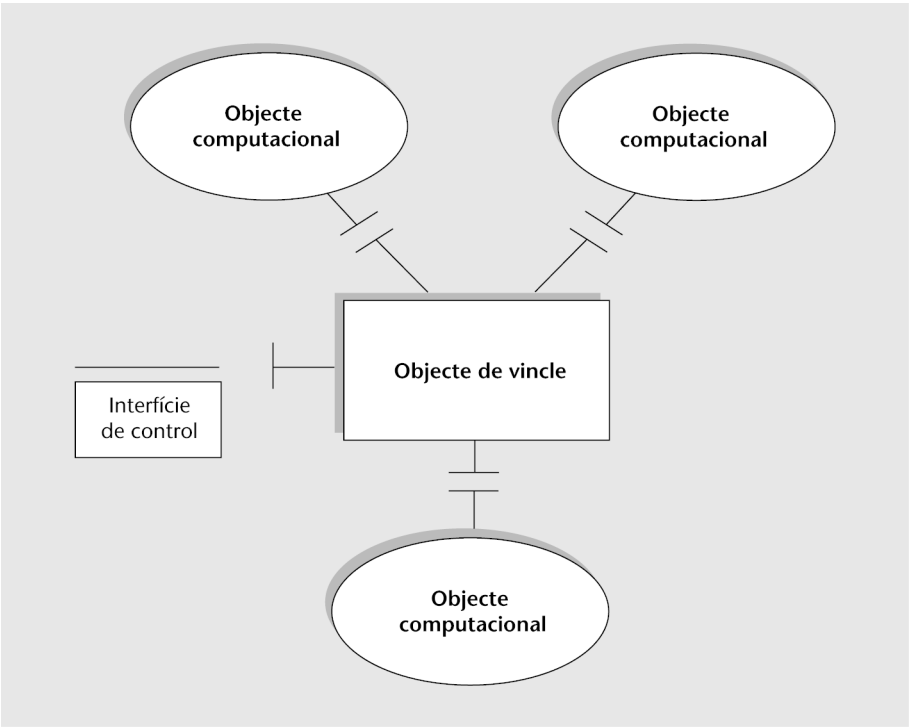


Figura 5. Objecte de vincle

L'objecte de vincle pot opcionalment instal·lar un conjunt d'interfícies de control, a través de les quals pot inspeccionar les operacions dutes a terme o enviar/rebre els identificadors de les interfícies creades. Aquest mecanisme és molt important en el cas dels sistemes multimèdia, per exemple. Considereu el cas en què diversos objectes computacionals emeten i reben fluxos d'informació (per exemple, àudio, imatges o tots dos). En aquest cas, l'existència d'un objecte de vincle és crucial per a la sincronització del flux entre tots els objec-

tes, a més de dur a terme les tasques de *broadcast* o *multicast*. De la mateixa manera, les interfícies de control són útils si es du a terme un monitoratge de la qualitat de servei (per exemple, per a comprovar que el nombre de trames d'àudio enviades per segon no supera un cert llindar).

Finalment, la **delegació** és un altre mecanisme comunament utilitzat en el desenvolupament de programari basat en components, i que afavoreix i possibilita la reutilització. Mitjançant la delegació, un component encarregat d'implementar una operació el que fa és invocar-ne un altre que ja la implementa, reutilitzant els seus serveis.

Vegeu també

Sobre interfícies que han de ser proporcionades per altres components externs, podeu veure la figura que mostra l'arquitectura de l'aplicació en l'apartat 4.3.3 del mòdul 1.

Ja havíem vist en l'apartat anterior com la delegació s'utilitza per a implementar els *wrappers* que s'usen per a adaptar components incompatibles amb els requisits de la nostra aplicació o d'altres components, resolent les diferències.

A part de poder "delegar" la implementació d'una interfície o operació a un component extern, la delegació també s'utilitza per a "delegar" un cert servei en algun dels components interns. Per exemple, el diagrama següent de la figura 6 mostra la composició interna del component **ATM**, un dels components arquitectònics de l'aplicació bancària descrita en els mòduls anteriors. En el diagrama pot veure's com el component **ATM** "abstracte" que es defineix en l'arquitectura del sistema està implementat com un conjunt de components programari interconnectats entre si, i que s'encarreguen d'implementar els seus serveis. En concret, el component ATM està fet mitjançant l'assemblatge de cinc components: **OperacionsBàsiques**, **Seguretat**, **MantenimentCaixer**, **TestMaquinari** i **TestProgramari**. El component **OperacionsBàsiques** implementa una de les interfícies de **l'ATM** (**OperacionsAmbCompte**) i utilitza una altra interfície (**Moviments**). Observeu com la implementació de les interfícies de **l'ATM** es "delega" en les interfícies que implementen altres components. De la mateixa manera, algunes de les interfícies que requereixen els components interns també es deleguen cap a l'exterior del component **ATM**, mitjançant les seves interfícies requerides, i que han de ser proporcionades per altres components externs.

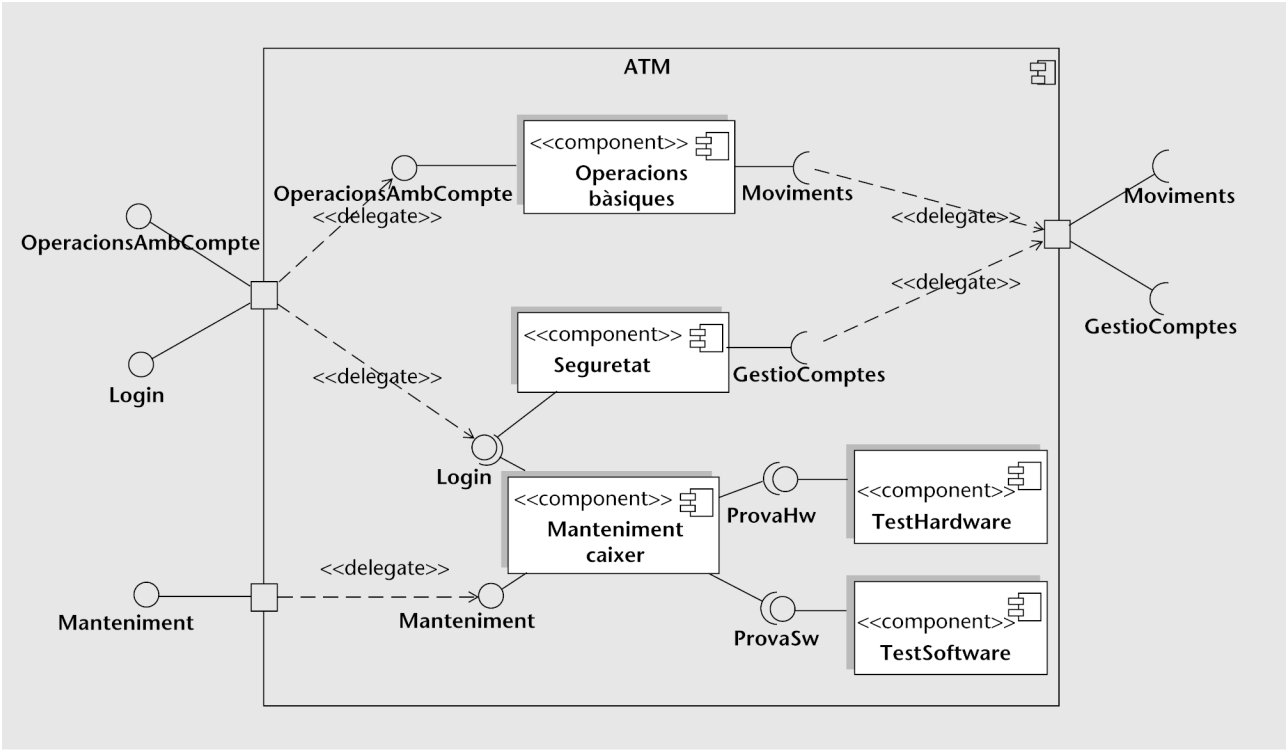


Figura 6. Diagrama de components UML amb la descomposició interna del component arquitectònic ATM en components programari

2.5. Reflexió

Els nous estàndards de components ja especifiquen el tipus d'informació que un component ha de fer pública sobre si mateix i sobre les seves propietats. Aquesta **metainformació** és la que permet als contenidors, entorns i eines de desenvolupament, així com a altres components, descobrir la funcionalitat que ofereix un component i poder manipular-lo.

La manera usual d'implementar la metainformació dels components és mitjançant tècniques reflexives, que cobren cada vegada més rellevància, ja que constitueixen la millor via de descobrir els recursos d'un entorn donat i adaptar-los de manera flexible i modular per a construir aplicacions.

La **reflexió** és un mecanisme que permet representar l'estat dels objectes d'un sistema com a entitats de primera classe i, per tant, poder observar-los, manipular-los i raonar-hi com a elements bàsics.

La programació que utilitza tècniques de reflexió s'anomena **metaprogramació**.

Per exemple, el llenguatge de programació Java proporciona un paquet denominat **Reflection** que conté mètodes que permeten en temps d'execució conèixer, per exemple, la classe a què pertany un objecte, els seus atributs i mètodes públics, construir invocacions per a cridar els mètodes esmentats, i fins i tot crear noves instàncies d'objectes d'una classe.

L'acció d'examinar la metainformació d'un component s'anomena **inspecció**, i pot ser tant estàtica (en temps de disseny) com a dinàmica (en temps d'execució).

Tots els models de components actuals (COM, Java EE, CCM o .NET, per exemple) implementen mecanismes d'inspecció. Per exemple, tots els components COM i CCM disposen d'una interfície (denominada **IUnkown** en COM i **home** en CCM) que permet conèixer la resta de les interfícies que implementa i utilitza el component (que comunament es coneix com a "navegar" per les seves interfícies). D'aquesta manera, és possible conèixer si un component implementa o no una certa funcionalitat (per exemple, serialització o *drag-and-drop*) i invocar-la en aquest cas.

S'anomena **introspecció** l'acció d'examinar la mateixa metainformació quan el model de components suporta reflexió.

La introspecció és un mecanisme molt usual en els components en què el comportament depèn d'alguns valors de les seves metadades, com per exemple els que estableixen la seva configuració actual o la de l'entorn on s'executa en aquell moment donat (penseu en components que admeten mobilitat d'unes màquines a altres).

2.6. Contenidors

Els components solen existir i cooperar dins de **contenidors** (*containers*), que són entitats programari que permeten contenir unes altres entitats, i proporcionen un entorn compartit d'interacció. S'aplica sobretot a objectes i components visuals, que contenen al seu torn altres objectes visuals.

D'aquesta manera, per exemple en el model CCM, els components es creen i són gestionats a través de les seves interfícies **home** i s'executen en **contenidors** que gestionen i accedeixen als serveis del sistema de manera transparent.

Normalment, la relació entre els components i els seus contenidors s'estableix mitjançant esdeveniments.

Com a exemple, considerem un contenidor que es registra en ser creat en un escriptori o una finestra per a ser informat de quan algú diposita un component dins seu (en general, un *drop site*). Així, quan un usuari efectua una operació d'arrossegar i deixar anar (habitualment coneguda per *drag-and-drop*) un component COM i el diposita en el *drop site*, el mecanisme d'esdeveniments del *drop site* ho notifica al contenidor, invocant el procediment que prèviament aquest havia registrat i passant com a paràmetre una referència (*handle*) al component que ha estat arrossegat fins al *drop site*. Davant d'aquesta notificació, el contenidor sol canviar l'aspecte de la icona del component per a indicar que l'operació d'arrossegament ha reeixit, i passar al component una referència als serveis que ofereix el contenidor perquè el component pugui utilitzar-los.

2.7. Serveis i facilitats comunes

La programació d'aplicacions distribuïdes es basa en un conjunt de serveis que proporcionen als components l'accés als recursos compartits d'una manera segura i eficient. Avui dia, gairebé totes les plataformes de components distribuïts implementen diversos d'aquests mecanismes i serveis, que estan normalment basats en els serveis comuns que defineix RM-ODP (encara que no són completament conformes a l'estàndard que defineix ODP, moltes vegades per motius de màrqueting que obliga a definir-los de manera propietària per a assegurar-se així la continuïtat i fidelitat dels clients).

Aquests serveis solen englobar-se en les categories bàsiques següents:

- **Comunicacions remotes**: proporcionen una sèrie de mecanismes per a la comunicació remota entre components, com poden ser els missatges, RPC, canals, etc.

- **Serveis de directori**: proporcionen un esquema de direccionament global per als recursos, serveis i components d'un sistema, incloent-hi l'assignació dels seus noms i la seva organització, localització i accés.

- **Seguretat**: proporcionen l'accés segur i autentificat als recursos i serveis del sistema, així com protecció davant atacs externs o interns.

- **Transaccions**: proporcionen els mecanismes per a coordinar les interaccions dels components quan aquests comparteixen dades crítiques, de manera que sempre se'n pugui garantir la coherència.

- **Gestió**: proporcionen un conjunt de facilitats per al monitoratge, la gestió i l'administració dels components, recursos i serveis del sistema.

Com hem esmentat, no totes les plataformes implementen tots els serveis, ni són compatibles entre si. També és important assenyalar que aquests serveis, normalment, es proporcionen al component a través del contenidor on aquest resideix.

2.8. Deficiències de la programació orientada a components

Sens dubte, el desenvolupament de programari basat en components no està exempt de problemes. Si en la introducció d'aquest mòdul mostràvem quines deficiències del paradigma orientat a objectes van ser determinants per a avançar cap a l'enfocament de la POC, no hem d'ometre ara quina problemàtica podem trobar en tractar amb components. Els següents reptes i problemes amb què actualment s'enfronta aquesta disciplina són:

1) Clarividència

Es refereix a la dificultat amb què es troba el dissenyador d'un component en fer-ne el disseny, ja que no coneix qui l'utilitzarà, ni com, ni en quin entorn, ni per a quina aplicació. Aquest problema està intrínsecament lligat a la composició tardana i a la reusabilitat de components.

2) La paradoxa coneguda com a "**en maximitzar la reutilització es minimitza l'ús**" ("*maximizing reuse minimizes use*").

A l'hora de desenvolupar un component, podem fer-ho per a resoldre únicament el problema en qüestió i només perquè s'utilitzi en el nostre entorn concret. D'aquesta manera serem capaços de fer-ho de manera molt eficient i el component s'ajustarà perfectament al problema que s'ha de resoldre. Però el nombre d'usuaris i de contextos en què podrà ser utilitzat serà molt petit (penseu, per exemple, en un component a mida per a imprimir les dades que produeix el nostre programa de gestió de comptes bancaris). Per a tractar d'augmentar la reusabilitat del component, podem pensar de tractar de fer-ho tan general com es pugui (per exemple, desenvolupant un component general

per a imprimir taules, quadres i figures en diferents formats, i que accepti diferents formats de dades: text pla, fitxers Excel, SSPS, etc.). El problema és que, en augmentar les seves possibilitats i capacitats, sol augmentar tant la seva complexitat com la seva dificultat d'ús (és més difícil d'usar per les nombroses opcions que presenta), i possiblement puguin disminuir les seves prestacions (ja no és tan ràpid com ho era abans, quan no havia de fer conversions de formats d'entrada o sortida, i coneixia per endavant els tipus de dades que havia d'imprimir). D'aquesta manera, disposem d'un component *a priori* més genèric i reutilitzable, però que ja no és tan fàcil d'usar i eficient com abans i, per tant, preferim usar el component original per a la nostra aplicació. En aquest sentit, en desenvolupar un component cal trobar un equilibri entre el seu grau de generalització (reutilització) i el d'ús efectiu i eficient d'aquest.

3) Evolució dels components

La gestió de l'evolució és un problema seriós, ja que en els grans sistemes han de poder coexistir, i molt possiblement coexistiran, diverses versions d'un mateix component. Hi ha diferents enfocaments per a abordar aquest problema (des de la immutabilitat d'interfícies de COM fins a la integració d'interfícies que propugna CORBA), encara que cap no és totalment satisfactori.

4) Falta de suport formal

Els mètodes formals troben dificultats per a treballar amb les peculiaritats dels components, com pot ser la composició tardana, el polimorfisme o l'evolució dels components. Això és un problema per als sistemes crítics en què cal demostrar formalment que els programes funcionen bé pel risc en vides humanes que poden suposar si es produeixen fallades (sistemes de control d'avions o de coets espacials, de centrals nuclears, sistemes d'electricitat o de comunicacions, etc.).

Lectura recomanada

G. Leavens; M. Sitaraman (2000). *Foundations of Component-Based Systems*. Cambridge, UK: Cambridge University Press.

5) Asincronia i curses d'esdeveniments

En els sistemes oberts i distribuïts, els temps de comunicació no estan delimitats inicialment, ni es poden negligir els retards en les comunicacions. Per tant, és molt difícil garantir l'ordre relatiu en què es distribueixen els esdeveniments, sobretot quan els produeixen diferents fonts. Així mateix, el procés de difusió d'esdeveniments és complicat atès que tant els emissors com els receptors poden canviar amb el temps. La situació es complica quan el que es pretén és raonar formalment sobre l'aplicació a partir dels seus components.

6) Interoperabilitat

Actualment, els contractes dels components es redueixen a la definició de les seves interfícies en el nivell sintàctic i la interoperabilitat es redueix a la comprovació dels noms i la signatura dels mètodes. Tanmateix, cal poder referir-nos i buscar els serveis que necessitem per alguna cosa més que els seus

noms, i poder utilitzar els mètodes oferts en una interfície en l'ordre adequat. L'estudi de la interoperabilitat de components en el nivell de protocols (ordre entre els missatges) o semàntic ("significat" de les operacions) és un altre dels problemes oberts.

7) Qualitat

L'avaluació de la qualitat i de les propietats no funcionals, tant dels components individuals com dels sistemes que es construeixen partint d'aquests, és encara un problema no resolt completament. Per exemple, seria molt interessant disposar de mètriques que puguin servir d'ajuda per a seleccionar els components comercials que millor s'adapten a la nostra aplicació, d'acord amb els nostres requisits de qualitat particulars. Tanmateix, actualment hi ha una absència gairebé total de mètriques de qualitat per a components.

Bibliografia

A. Cechich; M. Piattini; A. Vallecillo (2003). *Component-Based Software Quality: Methods and Techniques*. Heidelberg: Springer-Verlag.

3. Representació de components programari en UML

En el mòdul 2 s'havien descrit els mecanismes que proporciona UML 2.0 per a especificar components arquitectònics, mitjançant els conceptes de **component** (unitat d'encapsulació de funcionalitat, amb interfícies ben definides i que pot ser reemplaçable en el seu entorn), **interfície** (tant implementada com requerida), **port** (que representa un servei, la signatura del qual és determinada per un conjunt d'interfícies, i que pot tenir associat un comportament) i **connector** (que permet connectar ports o interfícies entre si).

També s'havien presentat els principals tipus de diagrames UML que s'utilitzen per a representar l'arquitectura programari d'un sistema, els seus components i les seves interaccions: diagrames de components per a l'estructura i diagrames de comunicació i col·laboració per a descriure les interaccions entre els elements arquitectònics.

Per exemple, la figura 6 mostrava un diagrama de components amb l'estructura interna del component ATM, que estava, al seu torn, compost per cinc components acoblats entre si.

Però UML 2.0, a més dels components arquitectònics que defineixen l'estructura programari d'una aplicació, també permet representar els altres tipus de "components" que van definir Cheesman i Daniels, i que s'han descrits anteriorment: les **implementacions** dels components (o **artefactes**), els **components instal·lats** en un sistema, i les **instàncies de components**. Per a representar tots aquests elements, UML defineix els anomenats **diagrames de desplegament**, que s'estudien tot seguit.

3.1. Diagrames de desplegament

Els diagrames de desplegament mostren la distribució física del sistema, i revelen quins elements de programari s'executen en quins elements de maquinari.

Els elements principals d'aquests diagrames són els **nodes**, que es connecten entre si a través de **rutes de comunicació** (*communication paths*). Un node és un element que permet allotjar programari; pot ser de dues maneres:

- Un **dispositiu de** maquinari, com ara un ordinador o algun altre element de maquinari que es connecti al sistema (representat com un node estereotipat "device").

- Un **entorn d'execució** que permeti allotjar o contenir un altre programari, com els sistemes operatius, les màquines virtuals (com la JVM de Java) o

els processos contenidors (representat com a node estereotipat "execution environment").

A més, els nodes poden contenir **artefactes**, considerats com a manifestacions físiques d'elements lògics (programari) com, per exemple, fitxers executables, de dades, de configuració, etc. En general, els artefactes representen una implementació concreta d'un component i poden expressar-se com a classe UML estereotipada amb "artifact", o bé llistant-ne el nom dins dels mateixos nodes.

Vegeu també

Els artefactes s'han descrit en el subapartat "La definició de Cheesman i Daniels" d'aquest mòdul.

La llista d'artefactes d'un node mostra quins artefactes són desplegats en aquest node en temps d'execució del sistema.

És freqüent tenir diversos nodes físics fent la mateixa funció lògica (per exemple, en servidors amb còpies redundants). En aquest cas, en el diagrama es poden dibuixar les còpies i les seves rutes de comunicació o bé incloure-hi un únic element amb un valor etiquetat que determini el nombre de còpies en execució.

Les rutes de comunicació entre nodes indiquen com es comuniquen els diferents elements del diagrama. Per exemple, és habitual etiquetar-les amb informació sobre el protocol de comunicació utilitzat (per exemple, HTTP, TCP/IP, etc.).

En la figura 7 es mostren els principals elements d'un diagrama de desplegament en UML 2.0, en un exemple que presenta la clàssica estructura d'explorador web que accedeix a una aplicació web a través d'un servidor web encarregat de mostrar les pàgines que "construeix" el servidor d'aplicacions després de les sol·licituds del client. A més, també el servidor d'aplicacions accedeix a un conjunt de recursos a què l'usuari té accés a través d'un servidor FTP. En els casos que es consideren més rellevants, els camins de comunicació indiquen així mateix el protocol o mecanisme utilitzat per a dur a terme el procés de comunicació entre els nodes connectats.

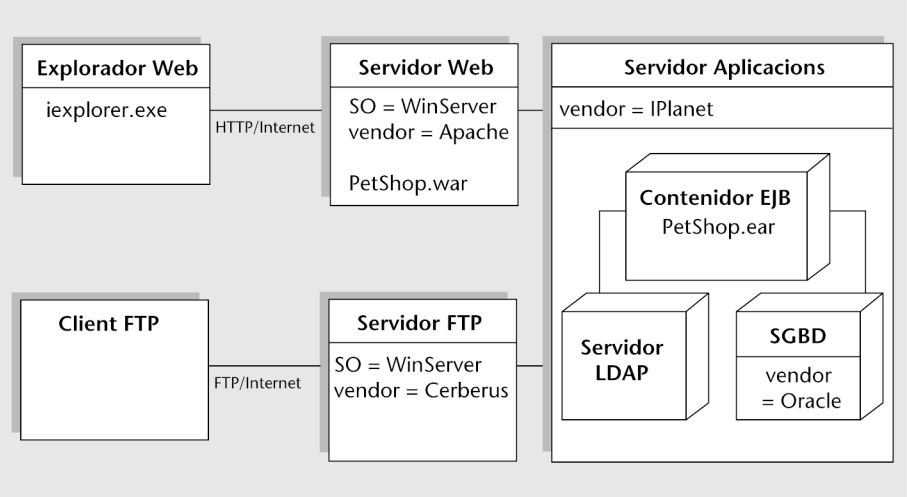


Figura 7. Exemple de diagrama de desplegament en UML 2.0

3.2. Mecanismes d'extensió en UML 2.0

Fins ara hem vist com representar components mitjançant diferents tipus de diagrames UML (de components, de comunicació i de desplegament) i de manera independent de qualsevol tecnologia. Tanmateix, quan es vol detallar més aquests diagrames per a adaptar-los a una plataforma específica o a un domini d'aplicació concret, ens trobem que UML, lògicament, no proporciona elements que ens permetin representar les particularitats de cada plataforma. Per exemple, com es pot indicar que un component és un EJB? O bé com s'ha d'indicar que una invocació a un servei d'un component ha de ser local o remota?

El fet que UML sigui un llenguatge de propòsit general proporciona una gran flexibilitat i expressivitat a l'hora de modelar sistemes. Tanmateix, hi ha nombroses vegades en què cal tenir algun llenguatge més específic per a modelar i representar els conceptes propis de certs dominis particulars. Això succeeix, per exemple, quan la sintaxi o la semàntica d'UML no permeten expressar els conceptes específics del domini, o quan es vol restringir i especialitzar els constructors propis d'UML (que solen ser massa genèrics i nombrosos) i expressar només els "elements" del nostre domini o plataforma.

L'OMG defineix dues possibilitats a l'hora de definir llenguatges específics de domini, i que es corresponen amb les dues situacions esmentades abans: o bé es defineix un nou llenguatge (com a alternativa d'UML), o bé s'amplia el mateix UML, especialitzant-ne alguns dels conceptes i restringint-ne uns altres, però respectant la semàntica original dels elements d'UML (classes, associacions, atributs, operacions, etc.).

La primera opció és l'adoptada per llenguatges com CWM (*common warehouse metamodel*), ja que la semàntica d'alguns dels seus constructors no coincideix amb la d'UML. D'altra banda, hi ha situacions en què n'hi ha prou d'ampliar el llenguatge UML utilitzant una sèrie de mecanismes recollits en el que es denominen **perfils UML** (UML *profiles*).

Cada una d'aquestes dues alternatives presenta avantatges i inconvenients. Així, definir un nou llenguatge *ad hoc* permet més grau d'expressivitat i correspondència amb els conceptes del domini d'aplicació particular, igual com quan ens fem un vestit a mida. Tanmateix, el fet de no respectar el **metamodel** estàndard d'UML impedirà que les eines UML existents al mercat puguin manejar els seus conceptes d'una manera natural. En general, resulta difícil decidir quan hem de crear un nou metamodel i quan, en canvi, és millor definir una ampliació d'UML usant els mecanismes d'extensió estàndards definits amb aquest propòsit.

Metamodel i metaclasse

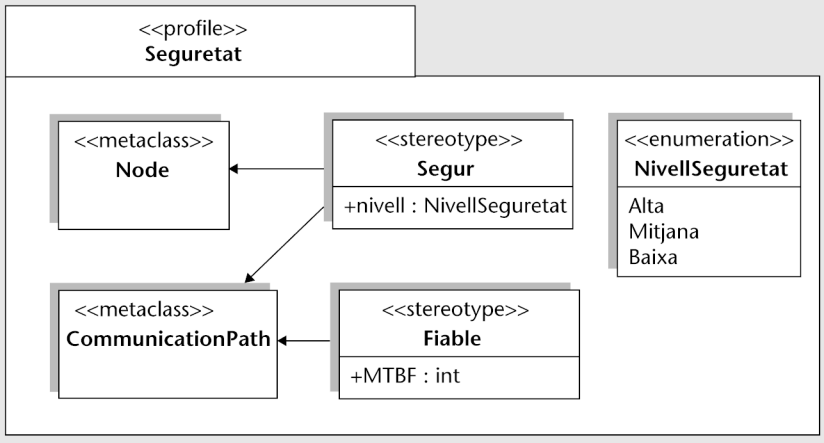
Una **metaclasse** és una classe les instàncies de la qual són classes.

Un **metamodel** defineix el llenguatge per a expressar un model en termes de metaclasses, relacions entre metaclasses i restriccions sobre aquestes.

En aquest apartat ens centrarem en l'anàlisi dels mecanismes d'ampliació que s'utilitzen per a definir un perfil UML.

Un **perfil UML** es defineix en un paquet, esterotipat "profile", que estén a un metamodel o a un altre perfil UML. Tres són els mecanismes que s'utilitzen per a definir perfils: estereotips (*stereotypes*), restriccions (*constraints*) i definicions d'etiquetes (*tag definitions*).

Per a il·lustrar aquests conceptes, utilitzarem un petit exemple de perfil UML, que definirà dos nous elements que poden ser afegits als nodes i als camins entre aquests, per a indicar certes propietats de seguretat dels elements d'un sistema.



1) En primer lloc, un estereotip és definit per un **nom** i per una sèrie **d'elements del metamodel** (**metaclasses**) a què pot associar-se. Gràficament, els estereotips es defineixen com a classes, estereotipades "stereotype". En el

nostre exemple, el perfil UML Seguretat defineix dos estereotips: Segur i Fiable. El primer indica que l'element sobre el qual s'aplica ha de ser segur, així com el grau de seguretat que volem implementar-hi (per simplicitat suposem que pot ser Alta, Mitja o Baixa). El segon indica la fiabilitat de la connexió entre nodes i porta associat el temps mitjà entre fallades (MTBF, per les seves inicials en anglès: *mean time between failures*) de la connexió mesurada en hores. Tal com s'indica en el perfil, només els nodes i els camins de comunicació entre si poden definir-se com a segurs, i només els camins poden tenir associat un grau de fiabilitat. Observeu com el perfil especifica els elements del metamodel d'UML (estereotipats "metaclass") sobre els quals es poden associar els estereotips mitjançant fletxes contínues de punta triangular en negreta.

2) Una **definició d'etiqueta** (*tag definition*) o simplement etiqueta és un metaatribut addicional que s'associa a una metaclasse del metamodel ampliat per un perfil. Tota etiqueta ha de tenir un nom i un tipus, i pertany a un estereotip determinat. D'aquesta manera, l'estereotip "Fiable" té una etiqueta denominada "MTBF", de tipus int, i que indica el temps mitjà entre fallades de la connexió entre nodes, que ha estat estereotipada com a "Fiable". Les etiquetes es representen de manera gràfica com a atributs de la classe que defineix l'estereotip.

3) Finalment, als estereotips, és possible associar-los **restriccions**, que imposen condicions sobre els elements del metamodel que s'ha estereotipat. D'aquesta manera poden descriure's, entre d'altres, les condicions que ha de verificar un model "ben format" d'un sistema en un domini d'aplicació.

Per exemple, suposem que el metamodel del nostre domini d'aplicació imposa la restricció que si dos o més nodes estan units per una connexió segura, els nodes han de ser segurs i el seu nivell de seguretat ha de coincidir amb el de la connexió. La restricció esmentada es tradueix en la següent restricció del perfil UML, en el llenguatge OCL (Object Constraint Language).

```
context Segur inv:
  self.baseCommunicationPath.connection-> -- recorrem els nodes d'aquesta associació
      forAll(n | n.extensionSegur.notEmpty() – i si està estereotipada "Segur"
         implies n.extensionSegur.nivel=self.nivel) – ha de tenir el mateix nivell.
```

En un perfil UML, una metaclasse (M) que és ampliada per un estereotip (E) es coneix com a "base$M", mentre que l'estereotip que l'amplia s'anomena "extension$E". A l'hora d'utilitzar aquestes referències en OCL, s'elimina el caràcter"$".

És per això que, per a saber si un node o un camí està estereotipat "Segur", n'hi ha prou de preguntar si self.extensionSegur.notEmpty().

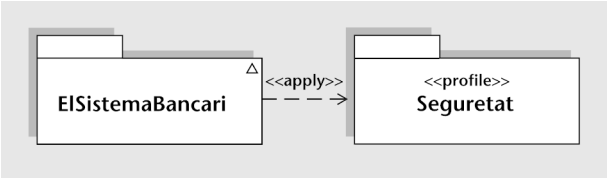
És important assenyalar que aquests tres mecanismes d'extensió no són de primer nivell, és a dir, no permeten modificar metamodels o perfils existents, sinó només afegir-los elements i restriccions, però respectant-ne la sintaxi i semàntica original. Tanmateix, sí que són molt adequats per a particularitzar un metamodel per a un o diversos dominis d'aplicació (com la seguretat en aquest

cas) o plataformes existents (EJB, CCM, .NET). Cada una d'aquestes particularitzacions o adaptacions és definida per un perfil, que agrupa els estereotips, les restriccions i les etiquetes propis d'aquesta adaptació.

Actualment ja hi ha definits diversos perfils UML, alguns dels quals han estat fins i tot estandarditzats per l'OMG: els perfils UML per a CORBA i per a CCM (CORBA *component model*), el perfil UML per a EDOC (*enterprise distributed object computing*), el perfil UML per a EAI (*enterprise application integration*) i el perfil UML per a planificació, prestacions i temps (*scheduling, performance and time*). Molts altres perfils UML estan actualment en procés de definició i estandardització per part de l'OMG i s'espera que vegin la llum molt aviat.

Cada un d'aquests perfils defineix una manera concreta d'usar UML en un entorn particular. Així, per exemple, el perfil UML per a CORBA defineix una manera d'usar UML per a modelar interfícies i artefactes de CORBA, mentre que el perfil UML per a Java defineix una manera concreta que modeli codi Java usant UML. En el mòdul 4 es discutirà el perfil UML per a EJB.

Una vegada s'ha definit un perfil UML, la manera d'utilitzar-lo en una aplicació concreta es representa mitjançant una relació de dependència (estereotipada "apply") entre el paquet UML que conté l'aplicació i els paquets que defineixen els perfils UML que es volen utilitzar. Per exemple, el diagrama següent mostra com s'indica que l'aplicació del sistema bancari fa ús del perfil UML Seguretat.



Utilitzant aquest perfil podem incloure la informació rellevant en els nostres diagrames. Per exemple, la figura 8 mostra els elements del diagrama de desplegament que ja havíem presentat en la figura 7, als quals hem afegit la informació sobre la seguretat que volem especificar. Per a fer això, els hem "marcat" amb estereotips que indiquen si són segurs o fiables, així com el nivell de seguretat de cada un dels elements segurs i el temps mitjà que acceptem entre les fallades dels elements fiables. Observeu com s'especifica el valor de les etiquetes associades als elements estereotipats mitjançant notes que mostren l'estereotip corresponent, el nom de l'etiqueta i el valor assignat (aquests valors se'ls denomina valors etiquetats o *tagged values*):

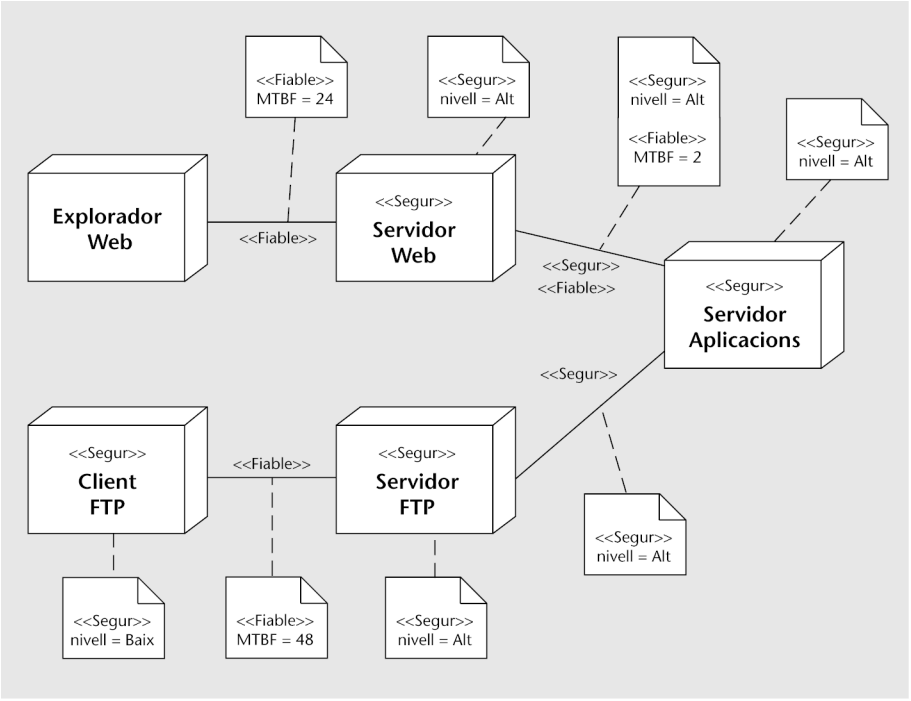


Figura 8. Diagrama del desplegament els elements del qual s'han "marcat" amb informació de seguretat mitjançant un perfil d'UML.

També es poden mostrar valors etiquetats corresponents a estereotips diferents en una mateixa nota UML, com mostra la nota de la connexió estereotipada "Segur" i "Fiable" en el diagrama anterior.

4. Procés de desenvolupament basat en components

Una vegada que coneixem els conceptes i mecanismes propis del desenvolupament de programari basat en components, en aquest apartat discutirem els processos de desenvolupament més apropiats per a construir sistemes d'acord amb aquest enfocament.

Per a fer-ho, sabem que RM-ODP proporciona els conceptes i mecanismes adequats per a dissenyar sistemes de programari, i estableix una clara distinció entre la descripció de l'arquitectura programari (en termes de components arquitectònics i connectors), de la seva distribució i implementació posterior usant, per exemple, components programari. Ara bé, RM-ODP no determina cap procés de desenvolupament concret que defineixi les tasques que ha de dur a terme l'enginyer de programari per a "transformar" els components i connectors arquitectònics en els components programari que proporcionen la implementació del sistema:

- Quines fases cal seguir-hi?
- Com es busquen els components programari que necessito per a implementar el sistema?
- Com selecciono el més adequat, en cas d'haver-hi més d'un component programari que tingui la funcionalitat adequada?
- Què ocorre si no hi ha cap component programari que implementi exactament la funcionalitat que exigeix l'arquitectura programari de la meva aplicació?
- És possible trobar diversos components programari que, combinats adequadament, permetin implementar els requisits d'un únic component arquitectònic?
- I si trobo en un dipòsit de programari un component que implementa només parcialment la funcionalitat d'un component arquitectònic, m'interessa comprar aquest component programari i tractar d'adaptar-lo, o bé desenvolupar-ne un de nou que s'adapti perfectament als requisits del component arquitectònic?

Es tracta, per tant, de definir una sèrie de processos de desenvolupament de programari que permetin implementar els requisits d'una arquitectura programari a partir de components programari. En aquest apartat descriurem alguns dels processos de desenvolupament de programari més utilitzats actualment i com poden aplicar-se al cas concret que ens ocupa.

En primer lloc, podríem pensar que moltes de les pràctiques utilitzades en l'enginyeria del programari tradicional són igualment aplicables al cicle de vida dels sistemes basats en components, com per exemple el procés unificat de desenvolupament de Rational (*rational unified process*, RUP). Ara bé, aquest tipus de processos estan dirigits cap al desenvolupament d'un programari més genèric i, per tant, cal adaptar-los. Per a fer-ho, cal contemplar els requisits específics dels sistemes basats en components, com per exemple el disseny de les interfícies que defineixen els serveis o la cerca i selecció de components prefabricats que puguem reutilitzar en l'aplicació.

Les tècniques de disseny específiques per al desenvolupament de sistemes basats en components més utilitzats són Catalysis (de Desmond D'Souza i Allan Wills) i l'aproximació basada en "components UML", de John Cheesman i John Daniels. Totes dues proposen una adaptació del procés RUP per al cas concret dels sistemes basats en components. A continuació passem a descriure ambdues propostes. Abans d'això s'ofereix una breu descripció dels conceptes de RUP sobre els quals es basen tant Catalysis, com la proposta de Cheesman i Daniels.

4.1. El procés unificat de Rational

El procés unificat de Rational defineix un conjunt de pràctiques que cobreixen les activitats de modelatge de negoci, gestió de requisits, anàlisi i disseny, implementació, proves i desplegament. Totes aquestes pràctiques estan suportades per una sèrie d'eines i aplicacions que ajuden a dur-les a terme. RUP és un dels processos més generals dels que hi ha actualment i són d'aplicació possible en diferents dominis. De fet, encara que RUP no estava originàriament concebut per a això, és possible adaptar-lo per al desenvolupament de programari basat en components.

Rational

Després de l'adquisició de Rational per part d'IBM, aquest és l'encarregat de l'explotació d'aquesta proposta i d'aportar les eines necessàries per a utilitzar-lo, com és el cas *d'IBM rational method componer*.

Pot trobar-se informació més detallada del mètode i de les eines disponibles en:

http://www-306.ibm.com/ software/awdtools/rup.

Qualsevol projecte realitzat seguint RUP es divideix en quatre **fases**:

1) Concepció (posada en marxa).
2) Elaboració (definició, anàlisi, disseny).

3) Construcció (implementació).
4) Transició (final del projecte i posada en producció).

En cada fase es faran una o diverses iteracions (la mida d'aquestes variarà segons les necessitats de cada projecte) i, dins de cada una d'aquestes iteracions, se seguirà un model en cascada per als fluxos de treball que requereixen les activitats anteriorment esmentades.

Com es mostra en la figura 9, la idea clau del procés unificat és la separació en etapes del projecte programari a partir de les **activitats** que tenen lloc en el cicle de vida del programari. Les activitats de més nivell que es distingeixen són les referides a la gestió de requisits, anàlisi, disseny, implementació i prova. A diferència del model en cascada, RUP assumeix que aquestes activitats poden coincidir en el temps. D'altra banda, per a cada una d'aquestes activitats es considera la seqüència de fases descrita anteriorment (concepció, elaboració, construcció i transició). Així, mentre que discorre la seqüència d'activitats pot haver-hi diverses iteracions dins d'aquest conjunt de fases.

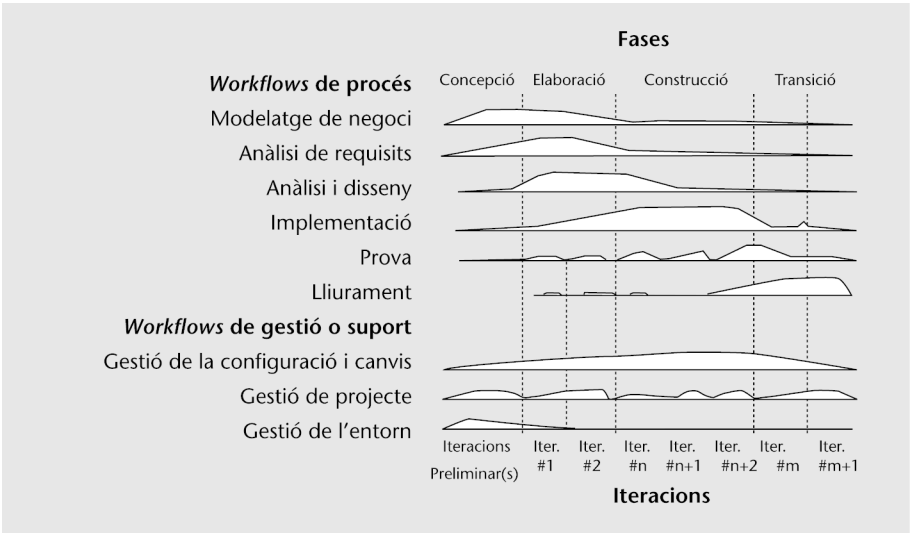


Figura 9. Vista general de RUP

RUP defineix nou **activitats** o *workflows* (fluxos de treball) que s'ha de dur a terme en cada projecte (modelatge de negoci, anàlisi de requisits, anàlisi i disseny, implementació, prova, lliurament, gestió de configuració i canvis, gestió de projecte i gestió de l'entorn). Al seu torn, cada activitat consisteix en una seqüència de **fases** que produeix un resultat de valor observable. Les activitats de RUP s'engloben en dos grups: les de procés i les de gestió o suport. En la figura 10 es mostra de manera genèrica el transcurs d'un projecte programari.

A més, el procés defineix una sèrie de **rols** que es distribueixen entre els membres del projecte i que defineixen les tasques de cada un i el resultat (**artefactes** en l'argot de RUP) que se n'espera.

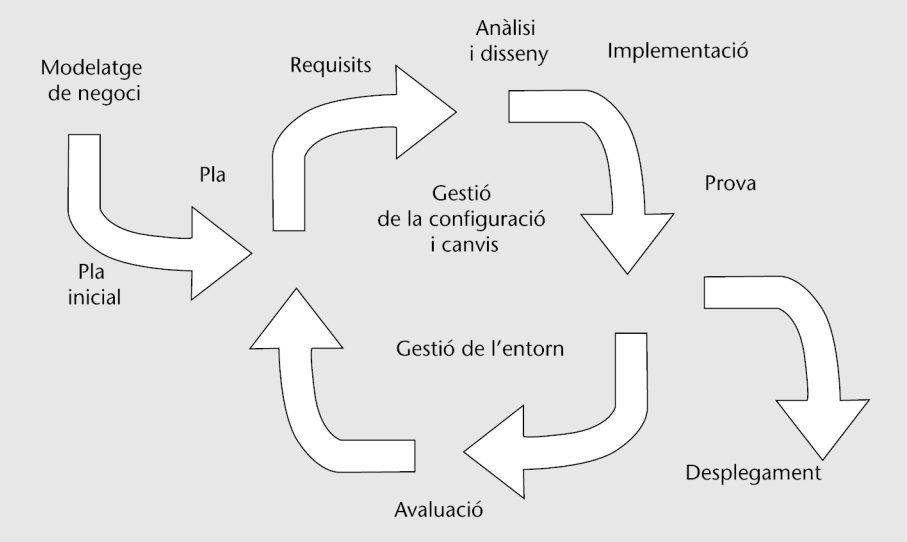


Figura 10. Activitats de RUP

RUP es basa en casos d'ús per a descriure el que s'espera del programari i està molt orientat a l'arquitectura del sistema, documentant-se tan bé com es pugui, basant-se en UML com a eina principal. Com que RUP és un procés molt general i extens, abans d'utilitzar-lo cal adaptar-lo a les característiques de cada projecte en particular o de cada organització. Aquesta adaptació és el que persegueixen les dues propostes que es descriuen tot seguit.

4.2. Aproximació basada en Catalysis

4.2.1. Conceptes de Catalysis

L'aproximació basada en Catalysis es defineix com "una metodologia per al modelatge i la construcció de sistemes oberts basats en components i marcs de treball *frameworks*". Les característiques següents constitueixen els objectius d'aquest mètode:

Catalysis

A http://www.catalysis.org es pot trobar informació extensa sobre Catalysis. En aquest web està disponible també el llibre que exposa aquest enfocament:

F. Desmond; D'Souza; Alan C. Wills (1999). *Objects, Components, and Frameworks with UML: The Catalysis Approach.* Boston: Addison-Wesley.

- Els sistemes es modelen com un conjunt de components que interactuen entre si.
- El comportament del sistema s'analitza i descriu en termes d'interfícies.
- Les especificacions dels components es fan independentment de les seves implementacions, usant una notació precisa i formal (precondicions, postcondicions i invariants).
- Els patrons de les interaccions dels components es modelen de manera explícita i, per tant, poden ser reutilitzats en diferents dissenys.

- Se segueix un procés de refinament iteratiu i rigorós per a anar derivant descripcions cada vegada amb més detall a partir de descripcions més abstractes del comportament del sistema.

- Els conceptes de modelatge es corresponen directament amb representacions concretes en UML.

Catalysis ajuda al modelatge d'un sistema mitjançant models de **tipus** (o especificacions del comportament visible d'un objecte que participa en el sistema exercint un rol). El concepte de "tipus" de Catalysis es correspon amb el concepte de tipus d'ODP i es modela mitjançant classes d'UML. Les interaccions entre els tipus a què pertanyen els objectes del sistema es modelen com a **col·laboracions** UML, les quals permeten descriure les interaccions entre grups d'objectes, on els tipus representen els rols que exerceixen els objectes en les interaccions.

4.2.2. Modelatge de components basat en Catalysis

Catalysis defineix tres activitats principals, que són les que es mostren en la figura 11: a) entendre el context en què ens movem, b) definir l'arquitectura del nostre sistema i c) aportar la solució. Mitjançant aquestes tres activitats s'aconsegueix traslladar les necessitats i els requisits del negoci a una solució operativa.

Aquests tres passos poden ocórrer en qualsevol ordre. De fet, és habitual que un projecte de desenvolupament comenci basant-se en alguns aspectes ja prefixats com, per exemple, l'existència de components o sistemes heretats (*legacy systems*) prèviament existents en l'organització i que cal integrar en la solució final.

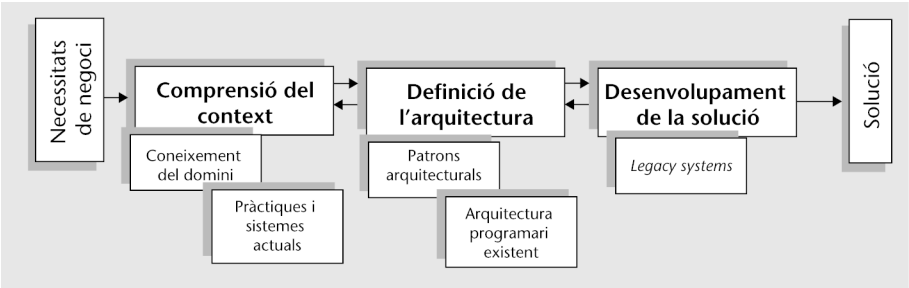


Figura 11. Activitats del procés definit per Catalysis

En els apartats següents s'expliquen cada una d'aquestes activitats.

Comprensió del context

S'ha d'iniciar el modelatge del domini descrivint dos aspectes bàsics del comportament amb un nivell d'abstracció elevat:

- **Comportament estàtic**: conjunt de tipus (rols) relacionats entre si en el domini de l'aplicació. Les relacions estructurals entre els diferents tipus representen les restriccions estàtiques que hi ha entre els elements del domini.

- **Comportament dinàmic**: correspon a les interaccions detectades entre els tipus del domini, recollides com a col·laboracions en les quals els tipus exerceixen uns rols determinats per iniciar o respondre a sol·licituds d'accions.

Nota

En una aproximació inspirada en UML, aquest comportament estàtic es representaria combinant el modelatge de classes, casos d'ús i diagrames de seqüència o col·laboració.

Per a cada tipus, modelat com una classe UML, se n'han de descriure les característiques (operacions i atributs) en detall. A més, resulta particularment important definir la semàntica d'aquestes operacions, especificant l'estat previ a la seva execució i l'estat final després d'obtenir el resultat de l'operació. Aquesta especificació es du a terme mitjançant precondicions i postcondicions.

Els diferents rols que exerceixen els objectes permeten obtenir les diferents interfícies, és a dir, agrupaments de determinats comportaments que exhibirà cada objecte. De fet, aquests comportaments poden ser vistos com a tipus que ofereixen serveis per mitjà d'operacions. Així, en aquesta etapa s'obté el model de tipus d'interfície, que defineix els conceptes i les restriccions referenciats mitjançant precondicions i postcondicions.

Definició de l'arquitectura

Una vegada identificats els aspectes dinàmics i estàtics del sistema abstracte, s'ha de decidir com cal empaquetar el comportament descrit en **unitats funcionals**, cada unes de les quals ha de ser independent, reutilitzable i executable en diferents entorns. És a dir, en aquesta etapa es descriu l'arquitectura programari de l'aplicació, identificant-ne els components, les interfícies i les connexions entre aquestes.

Arquitectures obertes

Com afirmen D'Souza i Wills (2000), en general s'ha demostrat que és bo que els sistemes estiguin basats en components reutilitzables definits sobre una arquitectura que separi les diferents àrees del negoci i les diferents regles que el regeixen, i que hagi estat construïda independentment de la infraestructura tecnològica subjacent. A més, aquesta arquitectura ha de poder acomodar no solament components heterogenis i distribuïts, sinó també els que s'han heretat de les aplicacions ja existents (*legacy systems*).

Com esmentàvem en l'apartat anterior, és possible que el disseny s'iniciï amb certes restriccions imposades, com l'existència d'antigues aplicacions, paquets o programari de tercers. En aquest cas, la decisió de com empaquetar alguna funcionalitat i comportament pot estar ja presa. El que pot ser necessari és el desenvolupament **d'adaptadors** (o *wrappers*) que permetin adaptar la funcionalitat proporcionada per aquest programari existent als requisits concrets de la nostra aplicació (nom de les operacions, nombre i tipus dels paràmetres, etc.) possibilitant així que s'hi integri.

La manera com es fa el repartiment i la separació de la funcionalitat en diferents components requereix un cert nivell d'experiència i ha d'atendre diferents criteris de disseny (requisits de negoci, condicions de desplegament, divisió bàsica de la funcionalitat, rendiment, escalabilitat, etc.).

Desenvolupament de la solució

L'últim dels passos d'aquesta aproximació consisteix a obtenir el codi final de l'aplicació modelada, de manera que pugui ser executada correctament en l'entorn definitiu. Partint de l'arquitectura programari de l'aplicació, necessitarem dur a terme un disseny detallat de cada component que implementarem, així com de la manera com s'acoblaran els uns amb els altres. Tothora hem de considerar la possibilitat que algun d'aquests dissenys ens sigui imposat prèviament (per exemple, per l'existència de components comercials o sistemes heretats), en el qual cas hem de dissenyar de manera detallada i implementar els mecanismes apropiats d'adaptació.

Com a resultat d'aquest disseny detallat obtindrem l'**arquitectura d'implementació** de components, un model que descriu físicament l'organització del sistema, així com la seva realització posterior en una tecnologia determinada.

Vegeu també

Aquest model del sistema sol representar-se mitjançant un conjunt de diagrames de desplegament, tal com es descrivia en l'apartat "Representació de components programari en UML" d'aquest mòdul.

4.3. Aproximació de Cheesman i Daniels

4.3.1. Conceptes bàsics

Per a saber-ne més

Per a saber-ne més es recomana llegir el llibre *UML Components* de Cheesman i Daniels, en el qual els autors descriuen detalladament el seu procés de desenvolupament. També podeu consultar el seu web, que conté molta informació sobre el seu mètode:
http://www.syntropy.co.uk/umlcomponents/.

Cheesman i Daniels van definir l'any 2000 un procés per al desenvolupament de programari basat en components inspirat en aproximacions ja existents, com RUP o Catalysis. D'aquests enfocaments (fonamentalment del primer) prenen nombrosos conceptes, així com part de la seva base metodològica. Tanmateix, el seu objectiu és proporcionar un mètode simple i específic per a sistemes basats en components programari, i per això opten per refinar la major part de les etapes de RUP i eliminen aquelles parts que consideren més restrictives, incomprensibles o complexes. Per exemple, la seva aproximació no pren en consideració cap aspecte relacionat amb els processos de gestió, tal com es defineixen en RUP en l'activitat de suport (des del seu punt de vista, si un procés de desenvolupament es vol fer utilitzable i comprensible, s'ha de deixar al mateix equip de disseny que triï el model més convenient). El fet de ser simple i estar recolzat per una notació com UML és el que l'ha convertit en el mètode de desenvolupament de programari basat en components de més èxit.

És important assenyalar que Cheesman i Daniels van usar originàriament UML 1.4 per a descriure els seus conceptes i artefactes, per la qual cosa van necessitar definir molts elements i estereotips nous per a representar-los (recordeu que UML 1.4 no té prou expressivitat per a modelar components programari). Tanmateix, el fet que UML 2.0 hagi adoptat moltes de les idees proposades per aquests autors (com per exemple els diferents tipus de components) i proporcioni mecanismes per a modelar arquitectures i components programari (com els ports) provoca que la seva proposta pugui ser modelable de manera molt natural i més simple amb UML 2.0. En aquest apartat farem, doncs, ús d'UML 2.0 per a descriure el mètode de Cheesman i Daniels.

4.3.2. Activitats del procés de desenvolupament

La figura 12 mostra les activitats definides en el procés de desenvolupament de Cheesman i Daniels, representades mitjançant blocs. Els requadres ombrejats representen els diferents artefactes (lliurables amb informació significativa) i les línies fines es corresponen amb el flux d'aquests artefactes entre les diferents activitats del procés.

1) Fase de requisits

Les activitats de **requisits**, **prova** i **desplegament** es corresponen exactament amb les definides en RUP, salvant petites diferències com és el cas dels requisits, que estan menys elaborats en aquesta aproximació que en el procés unificat. Així, el principal objectiu de la fase de **requisits** és descriure què hauria de fer el sistema i permetre als desenvolupadors i al client acordar una definició prèvia comuna. Per a fer això s'utilitzen els casos d'ús d'UML, mitjançant els quals s'analitzen, s'organitzen i es documenten la funcionalitat i les restriccions requerides, incloent-hi els requisits no funcionals. Els casos d'ús representen el comportament del sistema basant-nos en els diferents escenaris identificats i permeten modelar els actors, que representen els usuaris, així com qualsevol altre sistema extern que pugui interactuar amb el nostre. En aquesta fase també es defineix el **model conceptual** del sistema, expressat mitjançant un diagrama de classes UML.

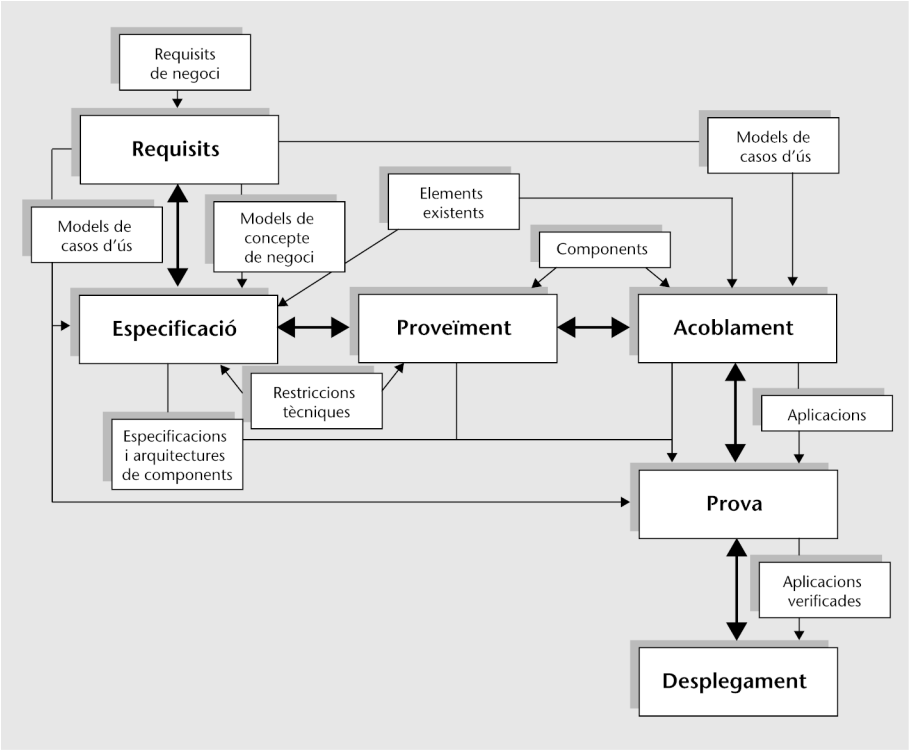


Figura 12. Visió general del procés de desenvolupament de Cheesman i Daniels

2) Fase d'especificació

La fase d'especificació es divideix en tres etapes. Malgrat que les tasques que s'han de fer semblen descrites de manera seqüencial, és important ressenyar que es tracta d'un procés altament iteratiu. Els artefactes resultants són dependents els uns dels altres i, per tant, la seva obtenció i desenvolupament és incremental. La figura 13 mostra un esquema de les tres etapes que constitueixen aquesta activitat:

a) Identificació de components

A partir del model conceptual del sistema i del model de casos d'ús (procedents de la fase de requisits) s'identifica el conjunt inicial de components especificats en el nivell d'aplicació, així com les seves relacions estructurals i dependències de comportament.

b) Interacció de components

Aquesta etapa examina com cada una de les operacions del sistema pot dur-se a terme utilitzant l'arquitectura de components proposada. L'agrupació d'operacions comença a dilucidar-se, per la qual cosa comencen a esbossar-se aspectes importants com el conjunt d'operacions per interfície o quines interfícies seran les que pertanyin a quins components. Aquest procés es du a terme de manera iterativa fins que es coneguin tots els detalls de l'estructura del sistema, amb un clar enteniment de les dependències entre els components.

c) Especificació de components

S'especifiquen de manera precisa les operacions (i restriccions sobre aquestes) que formen part de cada interfície, així com el comportament de cada component. Per a fer això, s'usen màquines d'estat associades als ports de cada component.

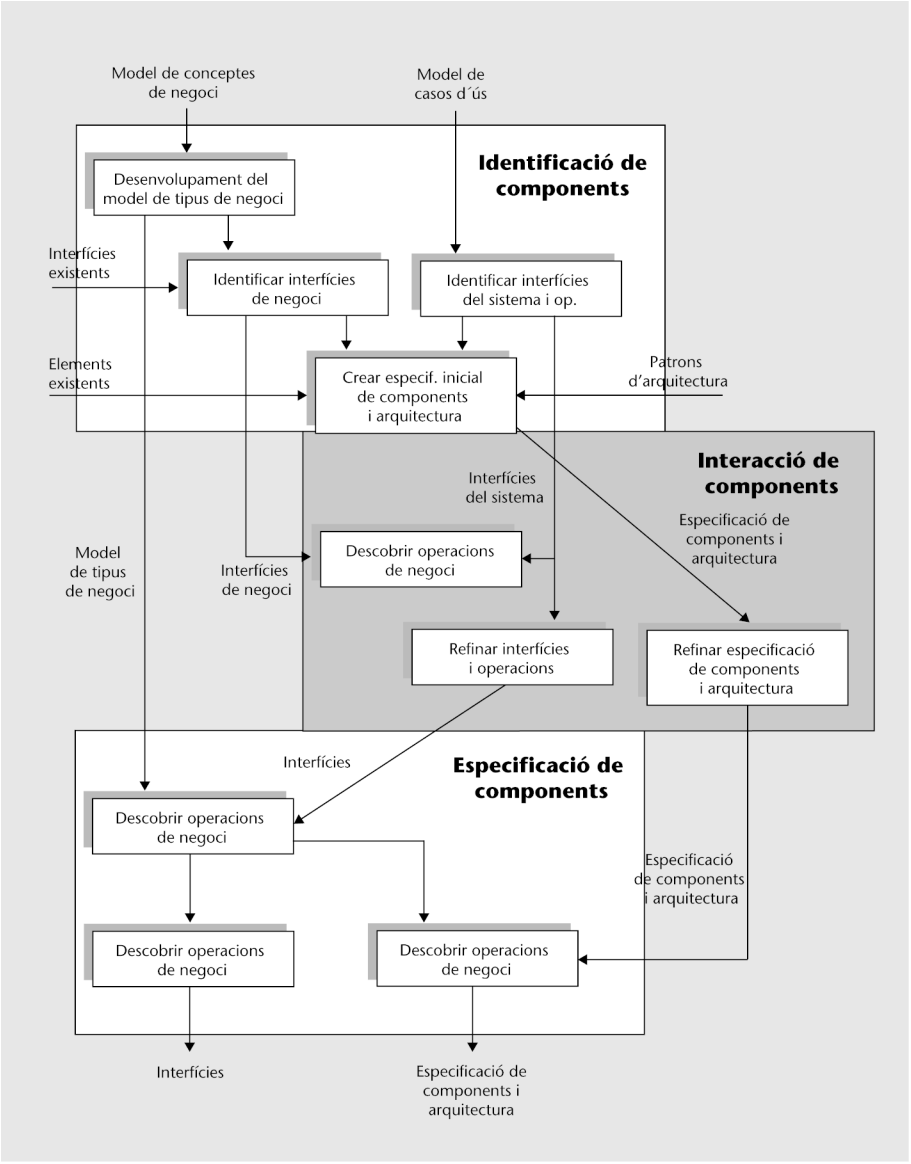


Figura 13. Fase d'especificació

Com a resultat de la fase d'especificació s'obté l'arquitectura programari del sistema.

3) Activitat de proveïment

Posteriorment, l'activitat de proveïment (*procurement*) és la responsable d'aportar el programari d'acord amb l'especificació de components i d'interfícies obtinguda en les activitats anteriors. Obtenir aquest programari pot implicar dos processos ben diferents. D'una banda, pot implicar desenvolupar-lo mitjançant algun dels llenguatges de programació i tecnologies disponibles al mercat. Però també pot tractar de reutilitzar components existents, ja sigui en la mateixa organització o bé adquirir-los d'algun fabricant extern. En aquest cas, la fase de proveïment inclou els processos de cerca i selecció dels components necessaris, la seva adquisició, així com el procés **d'adaptació** corresponent per a resoldre les incompatibilitats possibles amb els requisits (funcionals i no funcionals) imposats per l'arquitectura programari de l'aplicació.

Reutilitzar enfront de desenvolupar

Tal com s'havia indicat en el subapartat 2.3, la decisió de reutilitzar un component enfront de desenvolupar-lo no és senzilla, i bàsicament dependrà de si el procés d'adaptació que necessita és simple o no.

Com a resultat de l'activitat de proveïment s'obté el conjunt de components programari que proporcionen les implementacions dels components arquitectònics descrits en la fase d'especificació.

4) Fase d'assemblatge

Durant la fase d'assemblatge s'ajunten i s'integren els components i els elements programari ja existents (bé perquè hagin estat desenvolupats, bé perquè hagin estat adquirits i possiblement, adaptats), i es dissenya a més la interfície d'usuari del sistema basant-nos en els casos d'ús per a formar l'aplicació final.

5) Fase de prova

En la fase de prova, inspirada en la de RUP, els objectius que es persegueixen inclouen: verificar la interacció entre objectes, verificar la integració de tots els components del sistema, verificar que tots els requisits s'han desenvolupat correctament i identificar i resoldre defectes abans del desplegament de l'aplicació. Per a fer això, igual com en RUP, es proposa una aproximació iterativa, en la qual les diferents proves es duen a terme al llarg de totes les etapes del projecte. Això permet trobar defectes en activitats primerenques, la qual cosa redueix el temps i el cost de desenvolupament de manera dràstica. És important, per tant, triar adequadament les dimensions de qualitat en què es basaran les comprovacions i proves del sistema (per exemple, fiabilitat, funcionalitat, rendiment del sistema, etc.).

6) Fase de desplegament

Al final del procés s'arriba a la fase de desplegament, també basada en la de RUP, per a la qual s'haurà obtingut presumiblement un producte de qualitat i podrem lliurar-lo a l'usuari final. Aquesta fase de treball cobreix un ampli espectre d'activitats, com produir les diferents versions del programari, empaquetar-lo, distribuir-lo, instal·lar-lo, formar els usuaris, etc.

Resum

Al llarg d'aquest mòdul hem introduït els conceptes i mecanismes propis de la programació orientada a components, que es mostra actualment com una alternativa molt vàlida i d'utilització per a implementar sistemes distribuïts d'una certa complexitat.

Els components programari que defineix la programació orientada a components estan pensats per a implementar els components arquitectònics definits en l'arquitectura programari d'un sistema, i permeten explotar amb èxit alguns aspectes molt importants de l'enginyeria programari, com la **reutilització** de peces prèviament desenvolupades i provades amb èxit per altres persones (amb els estalvis en temps i esforç que això suposa), la **reemplaçabilitat** dins d'un context (fet que permet gestionar l'evolució dels components individuals amb el temps) i la millor **especificació** dels serveis que s'ofereixen i de les característiques no funcionals d'aquests serveis.

Els conceptes, mecanismes i processos relatius a la programació orientada a components presentats en aquest mòdul s'han descrit de manera general i independent de qualsevol tecnologia per tal de separar els conceptes propis d'aquesta disciplina de la seva implementació a qualsevol plataforma concreta, ja que les tecnologies comercials canvien molt més ràpidament que els conceptes teòrics. Els dos mòduls següents de l'assignatura descriuen amb detall com diferents tecnologies comercials implementen els conceptes esmentats.

Activitats

1. Basant-nos en les definicions de component programari donades en el primer apartat d'aquest mòdul, atreviu-vos a donar la vostra pròpia definició de component. Hi heu introduït cap factor nou que creieu que és important i que no s'ha inclòs en cap? Heu exclòs algun dels factors o característiques esmentades per algun dels autors perquè creieu que no és massa rellevant?

2. En l'actualitat, el mercat més gran de components programari a punt per a ser reutilitzats està a http://www.componentsource.com, que té milers de components de diferents models (COM, JavaBeans, etc.) organitzats en diferents categories. Visiteu-lo i tracteu de determinar si realment tots els elements programari que s'hi anuncien són components. Per exemple, en ComponentSource hi ha molts programes Java. Podem considerar tots aquests programes Java com a components programari? Quins sí i quins no? Què diferencia els uns dels altres?

3. Imagineu que heu de construir una aplicació programari i que voleu reutilitzar tants components comercials com sigui possible per a evitar els costos de desenvolupament. Com a part de l'aplicació, cal implementar certes funcions d'impressió i vista prèvia (*print and preview*). Visiteu ComponentSource i tracteu de trobar-hi components comercials d'aquest domini d'aplicació que pugueu utilitzar i integrar a la vostra aplicació. Quina informació us ofereix aquest mercat perquè pugueu seleccionar els components adequats per a la vostra aplicació? Quin tipus d'informació trobeu a faltar? Creieu que és possible seleccionar els components a partir de la informació que s'hi mostra o cal obtenir-ne una còpia d'avaluació i provar-los per a poder prendre una decisió correcta?

4. En el procés de desenvolupament basat en components s'intenten substituir els costos i esforços propis de la implementació dels components pels costos i esforços del proveïment de components externs (incloent-hi la cerca, selecció i adquisició de components) i d'integració (tractant de solucionar les incompatibilitats o diferències possibles que presenten els components adquirits). Ja coneixem els costos de desenvolupament i manteniment de qualsevol element programari, que normalment són molt elevats. Tanmateix, la reutilització no és exempta de costos. Observant els components disponibles en ComponentSource per a fer les funcions de *print and preview* que heu identificat en l'activitat anterior, tracteu d'estimar el que us costaria:

a) seleccionar-ne un (per a la qual cosa caldria avaluar-los tots),

b) aprendre a utilitzar-lo per a saber quant caldria adaptar-lo, i

c) fer les adaptacions oportunes mitjançant el desenvolupament dels *wrappers* adequats.

D'altra banda, tracteu d'estimar el que us costaria desenvolupar el component des de zero. Extraieu vosaltres mateixos les conclusions que considereu oportunes i tracteu de discutir-les amb altres companys.

5. Tracteu de buscar la definició adequada dels conceptes següents: classe, objecte, agent, mòdul, paquet, subsistema, recurs i marc de treball (*application framework*). Una vegada definits, tracteu de comparar-los amb el concepte (o conceptes) de component programari, i digueu-ne les diferències possibles.

6. En el mòdul anterior havíeu dissenyat l'arquitectura programari del sistema bancari, en la qual els clients podien usar els serveis bancaris a través de caixers automàtics i d'Internet. Descriviu cada una de les activitats del procés de Cheesman i Daniels per al desenvolupament d'aquest sistema, i analitzeu quines s'han cobert ja fins a arribar al disseny de l'arquitectura programari que havíeu fet i tracteu de cobrir-ne la resta.

Exercicis d'autoavaluació

1. Destaqueu almenys dues diferències entre el concepte de classe (tal com es defineix en la programació orientada a objectes o en UML) i el d'especificació de component (tal com la defineixen Cheesman i Daniels).

2. Definiu els conceptes i mecanismes següents habituals en la programació orientada a components: interfície, esdeveniment, contracte, reutilització, adaptador, composició tardana, vinculació dinàmica, delegació, reflexió, inspecció, introspecció i contenidor.

3. Imaginem que en el nostre domini d'aplicació disposem d'alguns serveis (modelats per interfícies UML) pels quals pretenem cobrar i, per tant, assignar-los un preu. També volem incloure alguna informació sobre el tipus d'invocació que es fa als serveis: local o remota.

Volem que als serveis de pagament només es pugui accedir de manera remota. Per a fer això, es demana:

a) Fer un perfil UML (denominat InterfíciesDePagament) que defineixi tres estereotips: "DePagament", "Local" i "Remota". El primer estén la metaclasse Interface del metamodel d'UML i defineix una etiqueta preu (de tipus int) que indica el preu d'accés a les operacions de la interfície. Els altres dos estereotips estenen la metaclase Usage del metamodel d'UML i serveixen per a indicar si la interfície s'usa de manera local o remota.

b) Definir una restricció en OCL sobre els estereotips "Local" i "Remota" que indiqui que només poden estar estereotipades com a tals les associacions de dependència d'ús l'extrem de les quals (supplier) sigui una interfície UML (indicació: usar la funció predefinida oclIsTypeOf(), que comprova si un element és d'un cert tipus).

c) Definir una restricció en OCL sobre l'estereotip "Local" que impedeixi que la interfície que actua com a *supplier* en l'associació d'ús sigui estereotipada "DePagament". Anàlogament, definir una restricció en OCL sobre l'estereotip "Remota" que obligui que la interfície que actua com a *supplier* en l'associació d'ús sigui estereotipada "DePagament".

4. Enumereu les activitats del procés de desenvolupament de Cheesman i Daniels i destaqueu les que s'han modificat respecte a les originals de RUP. Indiqueu-ne les principals diferències.

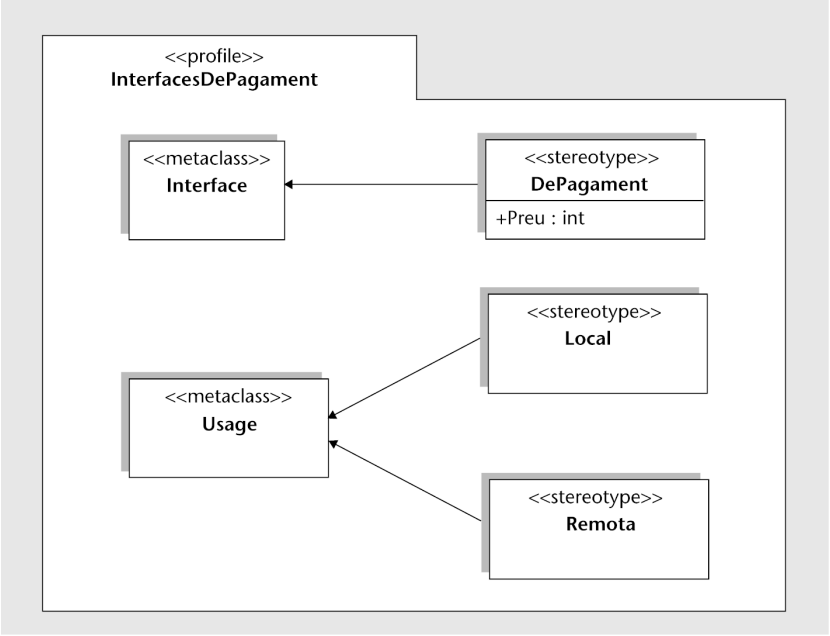
Solucionari

Exercicis d'autoavaluació

1. Esmentarem tres diferències. En primer lloc, les classes no fan referència explícita a les seves dependències i requisits. En segon lloc, les classes no separen l'especificació de la seva funcionalitat de la seva implementació (una classe sol contenir també els cossos dels mètodes que implementa). Finalment, les classes solen construir-se mitjançant herència (d'implementació) i, per tant, no solen permetre una instància i instal·lació separadament de la classe base i de les classes filles.

2. Les definicions dels conceptes esmentats són les que apareixen en l'apartat 2 d'aquest mòdul.

3. a) El perfil demanat és:



b) Les restriccions demanades poden ser expressades sobre els estereotips Local i Remota com segueix:

```
context Local inv:
self.baseUsage.supplier.oclIsTypeOf(Interface)
context Remota inv:
self.baseUsage.supplier.oclIsTypeOf(Interface)
```

La funció oclIsTypeOf() definida en OCL 2.0 comprova si un element és d'un cert tipus (en aquest cas, una interfície).

c) Las restriccions demanades poden ser expressades com segueix:

```
context Local inv:
self.baseUsage.supplier.extensionDePagament->isEmpty()
context Remota inv:
self.baseUsage.supplier.extensionDePagament->notEmpty()
```

4. Les activitats del procés de desenvolupament de Cheesman i Daniels són: anàlisi de requisits, especificació, proveïment, assemblatge, prova i desplegament. Les activitats que defineix RUP són: modelatge de negoci, anàlisi de requisits, anàlisi i disseny, implementació, prova, lliurament, gestió de configuració i canvis, gestió de projecte i gestió de l'entorn.

Els canvis que introdueixen Cheesman i Daniels en el seu procés respecte a RUP són els següents:

- Desapareixen les fases del modelatge de negoci i les de gestió (gestió de configuració i canvis, gestió de projecte i gestió de l'entorn) amb l'objectiu de simplificar el procés

de desenvolupament i concentrar-se fonamentalment en les tasques de construcció de l'aplicació.

- L'activitat RUP de requisits se simplifica bastant i passa a concentrar-se en l'elaboració d'un model conceptual del sistema i d'un model de casos d'usos UML amb els escenaris d'ús que capturen els requisits funcionals i no funcionals del sistema.
- L'activitat RUP d'"anàlisi i disseny" passa a denominar-se "especificació", i se centra en el disseny de l'arquitectura programari del sistema i l'especificació dels components, les seves interfícies i les seves connexions.
- L'activitat RUP d'"implementació" se substitueix per la de "proveïment". Aquesta nova activitat pot incloure tant la cerca, selecció, adquisició i adaptació de components ja existents, com la implementació possible si es decideix desenvolupar-los en comptes d'adquirir-los.
- Apareix l'activitat "assemblatge", que cobra més rellevància en el procés de Cheesman i Daniels a causa de la importància d'aquest procés en el desenvolupament de programari basat en components (en RUP és una part del procés d'implementació).
- Les activitats de "prova" i "lliurament" de RUP es mantenen pràcticament igual, es passen a denominar "prova" i "desplegament" i se centren més en els aspectes propis del desenvolupament basat en components.

Glossari

component programari *m* Unitat de composició d'aplicacions programari que encapsula una certa funcionalitat, té interfícies ben definides i és reemplaçable dins del seu entorn. Depenent del nivell d'abstracció a què ens referim, podem distingir entre els tipus de components següents:
• Especificació de component: especificació d'una unitat de programari, descrivint tant els serveis que proporciona com els que necessita altres components, així com el comportament de qualsevol instància del component que respecti l'especificació.
• Implementació de component (o artefacte): realització d'una especificació de component que pot ser implantada, instal·lada i reemplaçada de manera independent en un o més arxius i pot dependre d'altres components.
• Component instal·lat: còpia instal·lada d'un artefacte en una màquina concreta.
• Instància de component: instància d'un component instal·lat. Normalment està composta per una col·lecció d'objectes amb el seu propi estat i identitat única, que du a terme el comportament implementat.

contenidor *m* Entitat programari que permet allotjar i executar un component programari, proporcionant un entorn compartit d'interacció i facilitant al component l'accés als serveis comuns de la plataforma.

contracte *m* Especificació que s'afegeix a la interfície d'un component i que estableix les condicions d'ús i implementació que lliguen els clients i proveïdors del component.

disseny dirigit per contractes *m* Manera de fer el disseny d'un sistema de programari orientat a objectes o a components que consisteix a assignar a cada objecte o component un contracte (que contingui almenys l'especificació de la signatura i precondicions i postcondicions de les operacions de les seves interfícies) que n'especifica la funcionalitat. A partir d'això, la responsabilitat de comprovar les precondicions de qualsevol operació recau en el programa o usuari que invoca l'operació (i allibera, per tant, el component de comprovar-les), alhora que qualsevol implementació del component es compromet a respectar les postcondicions imposades pel contracte.

model de components *m* Definició que efectua un fabricant o organització de la forma concreta que tenen les interfícies dels seus components i els mecanismes que permeten interconnectar-los. Són exemples de models de components COM, JavaBeans, CORBA i CCM.

perfil UML *m* Paquet UML, estereotipat "profile", que permet ampliar a un metamodel, o a un altre perfil UML, mitjançant l'ús d'estereotips (*stereotypes*), restriccions (*constraints*) i definicions d'etiquetes (*tag definitions*).

plataforma de components *f* Implementació dels mecanismes d'un model de components concret, juntament amb una sèrie d'eines associades. Són exemples de plataformes de components ActiveX/OLE, Java EE, Orbix i OpenCCM.

reflexió *f* Mecanisme que permet representar l'estat dels objectes d'un sistema com a entitats de primera classe i, per tant, poder observar-los, manipular-los i raonar-hi com a elements bàsics.

rusted component *m* Component programari juntament amb l'especificació dels contractes associats a les seves interfícies que garanteix que la implementació del component proporciona els serveis especificats en el contracte sempre que l'usuari del component satisfaci les condicions requerides per a la utilització d'aquest.

Bibliografia

Jacobson, I.; Booch, G.; Rumbarugh, J. (2000). *El proceso unificado de desarrollo de software*. Madrid: Addison-Wesley.

D'Souza, D., Wills, A. C. (1999). *Objects, Components, and Frameworks with UML. The Catalysis Approach*. Boston: Addison-Wesley.

Szyperski, C. (2002). *Component Software. Beyond Object-Oriented Programming* (2a. ed.). Cambridge, MA: Addison-Wesley Longman.

Cheesman, J.; Daniels, J. (2000). *UML Components. A simple process for specifying component-based software*. Boston: Addison-Wesley.