Entorns de desenvolupament

CFGS.INF.M05/0.12

Desenvolupament d'aplicacions multiplataforma
Desenvolupament d'aplicacions web

Generalitat de Catalunya
Departament d'Educació

ioc
institut obert
de catalunya

Aquesta col·lecció ha estat dissenyada i coordinada des de l'Institut Obert de Catalunya.

Coordinació de continguts
Cristina Obiols Llopart

Redacció de continguts
Marcel García Vacas

Imatge de coberta
Fred (fr3d.org)

Primera edició: febrer 2013

Material realitzat per Eureca Media, SL
Dipòsit legal: DL B 12713-2016

Introducció

Per a un bon professional del desenvolupament del programari, tan important és conèixer les tècniques bàsiques i avançades de la programació i els principals llenguatges de programació actuals, com les eines i les estratègies que té a la seva disposició per dur a terme una feina al més òptima possible. Per això, en aquest mòdul es mostraran als alumnes els coneixements i es demostraran les eines i les tècniques actuals que són més útils per ser utilitzades en el procés de desenvolupament d'aplicacions per tal d'assolir aplicacions més eficients.

En la unitat "Desenvolupament de programari" s'introduirà un entorn de desenvolupament de programari específic: l'Eclipse. Per mitjà d'aquesta eina concreta es podrà experimentar la creació d'aplicacions de forma professional, fent ús d'utilitats típiques que ofereixen la majoria d'entorns de desenvolupament, però aplicades al cas particular d'aquest entorn (l'Eclipse). Això ha de permetre a l'alumne aplicar els coneixements teòrics a un entorn donat.

Un altre procés importantíssim i moltes vegades menystingut en el desenvolupament del programari és el disseny i execució de proves. S'aprendrà, doncs, en la unitat "Optimització de programari", la varietat de tipus de proves existents per tal d'aprendre'n a dissenyar les que correspongui per a una aplicació definida. En la mateixa unitat es detallarà el conjunt de documentació que hauria de tenir tota aplicació per tal de garantir un òptim funcionament així com un manteniment adequat. Actualment, existeixen multitud d'eines i d'utilitats integrades o no en les eines de desenvolupament del programari que ens faciliten tot aquest procés de prova i documentació d'aplicacions. En farem un repàs de les principals en aquest mateix apartat.

En la unitat "Introducció al disseny orientat a objectes" estudiarem els diagrames UML (diagrames referents a l'Unified Modeling Language), que ens permetran documentar la part d'anàlisi i disseny de les aplicacions. Així doncs, coneixerem les notacions que caldrà emprar per confeccionar cadascun dels principals diagrames, així com la utilitat dels mateixos.

És imprescindible, per assolir els resultats d'aprenentatge esperats, que l'alumne no es limiti a la lectura dels materials teòrics. Sobretot en la unitat 1, cal que l'alumne provi l'entorn de desenvolupament Eclipse que es treballa i que faci totes les activitats que pugui. Això permetrà adquirir les habilitats d'utilització d'entorns concrets i la familiarització amb les eines actuals específiques existents.

Tot el mòdul permetrà a l'estudiant obtenir una visió general del procés de desenvolupament del programari, visió imprescindible per a qualsevol programador d'aplicacions que vulgui dedicar-se professionalment a la creació o al manteniment d'aplicacions informàtiques.

Resultats d'aprenentatge

En finalitzar aquest mòdul l'alumne/a:

Desenvolupament de programari

1. Reconeix els elements i les eines que intervenen en el desenvolupament d'un programa informàtic, analitzant les seves característiques i les fases en què actuen fins arribar a la seva posada en funcionament.
2. Avalua entorns de desenvolupament integrat analitzant les seves característiques per editar codi font i generar executable.

Optimització de programari

1. Verifica el funcionament de programes dissenyant i realitzant proves.
2. Optimitza codi emprant les eines disponibles en l'entorn de desenvolupament.

Introducció al disseny orientat a objectes

1. Genera diagrames de classes valorant la seva importància en el desenvolupament d'aplicacions i emprant les eines disponibles en l'entorn.
2. Genera diagrames de comportament valorant la seva importància en el desenvolupament d'aplicacions i emprant les eines disponibles en l'entorn.

Continguts

Desenvolupament de programari

Unitat 1

Desenvolupament de programari

1. Desenvolupament de programari
2. Instal·lació i ús d'entorns de desenvolupament

Optimització de programari

Unitat 2

Optimització de programari

1. Disseny i realització de proves de programari
2. Eines per al control i documentació de programari

Introducció al disseny orientat a objectes

Unitat 3

Introducció al disseny orientat a objectes

1. Diagrames estàtics
2. Diagrames dinàmics

Desenvolupament de programari

Marcel García Vacas

Entorns de desenvolupament

Índex

Introducció

Cada vegada es pot trobar més i més electrònica, comunicacions i informàtica al nostre voltant, a la nostra vida quotidiana. Anem a un caixer automàtic d'una caixa o banc i caldrà interactuar amb una interfície tàctil per arribar a executar l'operativa que ens interessi. Circulem per determinades ciutats on ens indiquen en cartells lluminosos l'estat de la circulació en temps real o, fins i tot, si hi ha espais lliures per aparcar al propi carrer. I no cal parlar, si ens referim a aplicacions informàtiques, de tots els aparells electrònics que utilitzem diàriament:

- Ordinadors de sobretaula
- Ordinadors portàtils
- Telèfons mòbils
- Agendes electròniques
- Llibres electrònics
- Terminals de punt de venda
- Impressores i fotocopiadores
- Videoconsoles
- Televisors

Però aquests són només un petit exemple de totes les possibilitats que tenim avui dia d'utilitzar dispositius que porten un codi de programació incorporat, que disposen d'interfícies que permeten la interactuació amb els seus usuaris.

Abans que qualsevol d'aquests productes arribi al mercat, s'ha d'haver produït un procés, moltes vegades molt llarg, que comporta fer molts tipus de petites feines entre diverses persones. Una d'aquestes feines, part important de la creació dels productes, serà la programació, el desenvolupament del codi que es trobarà integrat en el producte i que permetrà mostrar aquestes interfícies, interactuar amb els usuaris i processar les informacions i dades vinculades.

Però per a això caldrà que prèviament aquests processos que es veuen automatitzats s'hagin ideat per algú i s'hagin desenvolupat. És en aquest punt que prenen importància els entorns de desenvolupament del programari.

Els entorns de desenvolupament són eines que ofereixen als programadors moltíssimes facilitats a l'hora de crear una aplicació informàtica. El mòdul "Entorns de Desenvolupament" mostra quin és el procés de desenvolupament d'una aplicació informàtica, tot indicant les característiques més importants de les eines que ajuden a aquest procediment.

Al llarg d'aquesta unitat formativa, "Desenvolupament de programari", es treballarà tot allò relacionat amb el desenvolupament del programari. La unitat formativa es troba dividida en dos apartats.

En l'apartat "Desenvolupament de programari" es defineixen tots aquells conceptes relacionats amb aquest àmbit, entrant en detall en cadascuna de les fases de desenvolupament de sistemes informàtics.

En l'apartat "Instal·lació i ús d'entorns de desenvolupament" s'agafa com a exemple un entorn integrat de desenvolupament, l'Eclipse, i s'explica tot allò relacionat amb la seva instal·lació i utilització. Eclipse és un entorn integrat de desenvolupament de codi obert que serveix per poder conèixer les característiques d'aquest tipus d'eines.

Resultats d'aprenentatge

En finalitzar aquesta unitat l'alumne/a:

1. Reconeix els elements i les eines que intervenen en el desenvolupament d'un programa informàtic, analitzant les seves característiques i les fases en què actuen fins arribar a la seva posada en funcionament.

- Identifica la relació dels programes amb els components del sistema informàtic: memòria, processador, perifèrics, entre d'altres.
- Identifica les fases de desenvolupament d'una aplicació informàtica.
- Diferencia els conceptes de codi font, objecte i executable.
- Reconeix les característiques de la generació de codi intermedi per a la seva execució en màquines virtuals.
- Classifica els llenguatges de programació.
- Avalua la funcionalitat oferta per les eines utilitzades en programació.

2. Avalua entorns de desenvolupament integrat analitzant les seves característiques per editar codi font i generar executable.

- Instal·la entorns de desenvolupament, propietaris i lliures.
- Afegeix i elimina mòduls en l'entorn de desenvolupament.
- Personalitza i automatitza l'entorn de desenvolupament.
- Configura el sistema d'actualització de l'entorn de desenvolupament.
- Genera executables a partir de codi font de diferents llenguatges en un mateix entorn de desenvolupament.
- Genera executables a partir d'un mateix codi font amb diversos entorns de desenvolupament.
- Identifica les característiques comunes i específiques de diversos entorns de desenvolupament.

1. Desenvolupament de programari

Tota aplicació informàtica, ja sigui utilitzada en un suport convencional (com un ordinador de sobretaula o un ordinador portàtil) o sigui utilitzada en un suport de nova generació (per exemple, dispositius mòbils com ara un telèfon mòbil de darrera generació o una tauleta tàctil PC), ha seguit un procediment planificat i desenvolupat detall per detall per a la seva creació. Aquest anirà des de la concepció de la idea o de la funcionalitat que haurà de satisfer aquesta aplicació fins a la generació d'un o diversos fitxers que permetin la seva execució exitosa.

Per convertir aquesta concepció d'una idea abstracta en un producte acabat que sigui eficaç i eficient hi haurà molts més passos, moltes tasques a fer. Aquestes tasques caldrà que estiguin ben planificades i que segueixin un guió que pot tenir en compte aspectes com:

- Analitzar les necessitats que tenen les persones que faran servir aquest programari, escoltar com el voldran, atendre a les seves indicacions...
- Dissenyar una solució que tingui en compte totes les necessitats abans analitzades: què haurà de fer el programari, quines interfícies gràfiques tindrà i com seran aquestes, quines dades s'hauran d'emmagatzemar i com es farà...
- Desenvolupar el programari que implementi tot allò analitzat i dissenyat anteriorment, fent-lo d'una forma al més modular possible per facilitar el posterior manteniment o manipulació per part d'altres programadors.
- Dur a terme les proves pertinents, tant de forma individualitzada per a cada mòdul com de forma complerta, per tal de validar que el codi desenvolupat és correcte i que fa el que ha de fer segons l'establert en els requeriments.
- Implantar el programari en l'entorn on els usuaris finals el faran servir.

Aquest apartat se centrarà en el tercer punt, el desenvolupament de programari.

1.1 Concepte de programa informàtic

Un primer pas per poder començar a analitzar com cal fer un programa informàtic és tenir clar què és un programa i què significa aquest concepte. En contrast amb altres termes usats en informàtica, és possible referir-se a un "programa" en el llenguatge col·loquial sense haver d'estar parlant necessàriament d'ordinadors. Es podria estar referint al programa d'un cicle de conferències o de cinema. Però, tot i que no es tracta d'un context informàtic, aquest ús ja aporta una idea general del seu significat.

Un **programa infomàtic** és un conjunt d'esdeveniments ordenats de manera que se succeeixen de forma seqüencial en el temps, un darrere l'altre.

Un altre ús habitual, ara sí vinculat al context de les màquines i els autòmats, podria ser referir-se al programa d'una rentadora o d'un robot de cuina. En aquest cas, però, el que se succeeix són un conjunt, no tant d'esdeveniments, sinó d'ordres que l'electrodomèstic segueix ordenadament. Un cop seleccionat el programa que volem, l'electrodomèstic fa totes les tasques corresponents de manera autònoma.

Per exemple, el programa d'un robot de cuina per fer una crema de pastanaga seria:

1. Espera que introduïu les pastanagues ben netejades, una patata i espècies al gust.
2. Gira durant 1 minut, avançant progressivament fins a la velocitat 5.
3. Espera que introduïu llet i sal.
4. Gira durant 30 segons a velocitat 7.
5. Gira durant 10 minuts a velocitat 3 mentre cou a una temperatura de 90 graus.
6. S'atura. La crema de pastanaga està llesta!

Aquest conjunt d'ordres no és arbitrari, sinó que serveix per dur a terme una tasca de certa complexitat que no es pot fer d'un sol cop. S'ha de fer pas per pas. Totes les ordres estan vinculades entre si per arribar a assolir aquest objectiu i, sobretot, és molt important la disposició en què es duen a terme.

Entrant ja, ara sí, en el món dels ordinadors, la manera com s'estructuren les tasques que han de ser executades és similar als programes d'electrodomèstics anteriorment citats. En aquest cas, però, en lloc de transformar ingredients (o rentar roba bruta, si es tractés d'una rentadora), el que l'ordinador transforma és informació o dades.

Un **programa informàtic** no és més que un seguit d'ordres que es porten a terme seqüencialment, aplicades sobre un conjunt de dades.

Quines dades processa un programa informàtic? Bé, això dependrà del tipus de programa:

- Un editor processa les dades d'un document de text.
- Un full de càlcul processa dades numèriques ubicades en un fitxer.
- Un videojoc processa les dades que fan referència a la forma i ubicació d'enemics i jugadors, les interfícies gràfiques on es trobarà el jugador, els punts aconseguits...

- Un navegador web processa les ordres de l'usuari i les dades que rep des d'un servidor ubicat a internet.
- Un reproductor de vídeo processa els fotogrames emmagatzemats en un arxiu i l'àudio relacionat.

Per tant, la tasca d'un programador informàtic és escollir quines ordres constituiran un programa d'ordinador, en quin ordre s'han de dur a terme i sobre quines dades cal aplicar-les perquè el programa porti a terme la tasca que ha de resoldre.

Executar un programa

Per "executar un programa" s'entén fer que l'ordinador segueixi totes les seves ordres, des de la primera fins a la darrera.

La dificultat de tot plegat serà més o menys gran depenent de la complexitat mateixa d'allò que cal que el programa faci. No és el mateix establir què ha de fer l'ordinador per resoldre una multiplicació de tres nombres que per processar textos o visualitzar pàgines a Internet.

D'altra banda, un cop fet el programa, cada cop que s'executi, l'ordinador complirà totes les ordres del programa.

De fet, un ordinador és incapaç de fer absolutament res per si mateix, sempre cal dir-li què ha de fer. I això se li diu mitjançant l'execució de programes. Tot i que des del punt de vista de l'usuari pot semblar que quan es posa en marxa un ordinador aquest funciona sense executar cap programa concret, cal tenir en compte que el seu sistema operatiu és un programa que està sempre en execució.

1.2 Codi font, codi objecte i codi excutable: màquines virtuals

Editors de text simples

Un editor de text simple és aquell que permet escriure-hi només text sense format. En són exemples el Bloc de Notes (Windows), el Gedit o l'Emacs (Unix).

Per crear un programa el que es farà serà crear un arxiu i escriure a un fitxer el seguit d'instruccions que es vol que l'ordinador executi. Aquestes instruccions hauran de seguir unes pautes determinades en funció del llenguatge de programació escollit. A més, haurien de seguir un ordre determinat que donarà sentit al programa escrit. Per començar n'hi haurà prou amb un editor de text simple.

Un cop s'ha acabat d'escriure el programa, el conjunt de fitxers de text resultants, on es troben les instruccions, es diu que contenen el **codi font**. Aquest codi font pot ser des d'un nivell molt alt, molt a prop del llenguatge humà, fins a un de nivell més baix, més proper al codi de les màquines, com ara el codi assemblador.

En l'apartat "Tipus de llenguatges de programació" es descriuen les característiques dels llenguatges màquina, assemblador, d'alt nivell i de propòsit específic.

El codi objecte de les instruccions té aquest aspecte:
10101001000001101100110
10100101011100001110111

La tendència actual és fer ús de llenguatges d'alt nivell, és a dir, propers al llenguatge humà. Però això fa aparèixer un problema, i és que els fitxers de codi font no contenen el llenguatge màquina que entendrà l'ordinador. Per tant, resulten incomprensibles per al processador. Per poder generar codi màquina cal fer un procés de traducció des dels mnemotècnics que conté cada fitxer a les seqüències binàries que entén el processador.

El procés anomenat **compilació** és la traducció del codi font dels fitxers del programa en fitxers en format binari que contenen les instruccions en un format

que el processador pot entendre. El contingut d'aquests fitxers s'anomena **codi objecte**. El programa que fa aquest procés s'anomena **compilador**.

El **codi objecte** és el codi font traduït (pel compilador) a codi màquina, però aquest codi encara no pot ser executat per l'ordinador.

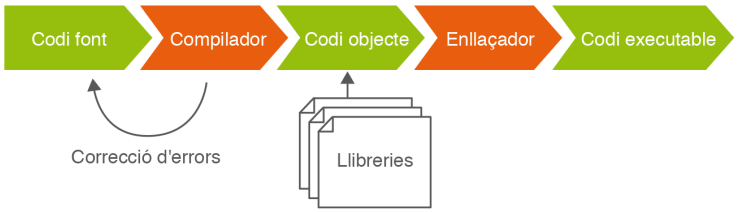
El **codi executable** és la traducció completa a codi màquina, duta a terme per l'enllaçador (en anglès, *linker*). El codi executable és interpretat directament per l'ordinador.

L' **enllaçador** és l'encarregat d'inserir al codi objecte les funcions de les biblioteques que són necessàries per al programa i de dur a terme el procés de muntatge generant un arxiu executable.

Una **biblioteca** (*library* en anglès) és un col·lecció de codi predefinit que facilita la tasca del programador a l'hora de codificar un programa.

A la figura 1.1 es mostra un resum ordenat de tots els conceptes definits. El codi font desenvolupat pels programadors es convertirà en codi objecte amb l'ajuda del compilador. Aquest ajudarà a localitzar els errors de sintaxi o de compilació que es trobin al codi font. Amb l'enllaçador, que recollirà el codi objecte i les biblioteques, es generarà el codi executable.

FIGURA 1.1. Procés de transformació d'un codi font a un codi executable



1.2.1 Màquina virtual

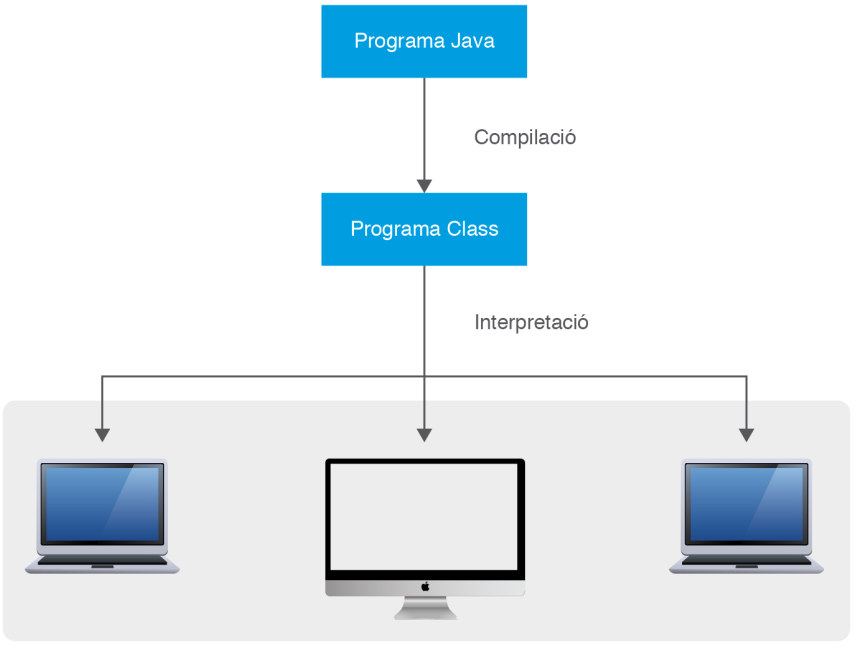
El concepte de màquina virtual sorgeix amb l'objectiu de facilitar el desenvolupament de compiladors que generen codi per a diferents processadors.

La **compilació** consta de dues fases:

- La primera parteix del codi font a un llenguatge intermedi obtenint un programa equivalent amb un menor nivell d'abstracció que l'original i que no pot ser directament executat.
- La segona fase tradueix el llenguatge intermedi a un llenguatge comprensible per la màquina.

Arribat aquest punt es podria plantejar la pregunta: per què dividir la compilació en dues fases? L'objectiu és que el codi de la primera fase, el codi intermedi, sigui comú per a qualsevol processador, i que el codi generat en la segona fase sigui l'específic per a cada processador. De fet, la traducció del llenguatge intermedi al llenguatge màquina no se sol fer mitjançant compilació sinó mitjançant un intèrpret, tal com es mostra en la figura 1.2.

Figura 1.2. Màquina virtual



Màquines amb diversos SO

La màquina virtual Java

La màquina virtual Java (JVM) és l'entorn en què s'executen els programes Java. És un programa natiu, és a dir, executable en una plataforma específica, que és capaç d'interpretar i executar instruccions expressades en un codi de bytes o (el *bytecode* de Java) que és generat pel compilador del llenguatge Java.

Codi de bytes

El codi de bytes no és un llenguatge d'alt nivell, sinó un veritable codi màquina de baix nivell, viable fins i tot com a llenguatge d'entrada per a un microprocessador físic.

La màquina virtual Java és una peça fonamental de la tecnologia Java. Se situa en un nivell superior al maquinari sobre el qual es vol executar l'aplicació i actua com un pont entre el codi de bytes a executar i el sistema. Així, quan un programador escriu una aplicació Java, ho fa pensant en la JVM encarregada d'executar l'aplicació i no hi ha cap motiu per pensar en les característiques de la plataforma física sobre la qual s'ha d'executar l'aplicació. La JVM serà l'encarregada, en executar l'aplicació, de convertir el codi de bytes a codi natiu de la plataforma física.

El gran avantatge de la JVM és que possibilita la portabilitat de l'aplicació a diferents plataformes i, així, un programa Java escrit en un sistema operatiu Windows es pot executar en altres sistemes operatius (Linux, Solaris i Apple OS X) amb l'únic requeriment de disposar de la JVM per al sistema corresponent.

El concepte de màquina virtual Java s'usa en dos àmbits: d'una banda, per fer referència al conjunt d'especificacions que ha de complir qualsevol implementació de la JVM; d'altra banda, per fer referència a les diverses implementacions de la màquina virtual Java existents i de les quals cal utilitzar-ne alguna per executar les aplicacions Java.

Als materials web es pot trobar una explicació, pas a pas, de com descarregar i instal·lar la màquina virtual Java.

L'empresa Sun Microsystems és la propietària de la marca registrada Java, i aquesta s'utilitza per certificar les implementacions de la JVM que s'ajusten i són totalment compatibles amb les especificacions de la JVM, en el prefaci de les quals es diu: "Esperem que aquesta especificació documenti suficientment la màquina virtual de Java per fer possibles implementacions des de zero. Sun proporciona tests que verifiquen que les implementacions de la màquina virtual Java operen correctament."

1.3 Tipus de llenguatges de programació

Establert el concepte de programa informàtic i els conceptes de codi font, codi objecte i codi executable (així com el de màquina virtual), cal ara establir les diferències entre els diversos tipus de codi font existents, a través dels quals s'arriba a obtenir un programa informàtic.

Un **llenguatge de programació** és un llenguatge que permet establir una comunicació entre l'home i la màquina. El llenguatge de programació identificarà el codi font, que el programador desenvoluparà per indicar a la màquina, una vegada aquest codi s'hagi convertit en codi executable, quins passos ha de donar.

Al llarg dels darrers anys ha existit una evolució constant en els llenguatges de programació. S'ha establert una creixent evolució en la qual es van incorporant elements que permeten crear programes cada vegada més sòlids i eficients. Això facilita molt la tasca del programador per al desenvolupament del programari, el seu manteniment i l'adaptació. Avui en dia, existeixen, fins i tot, llenguatges de programació que permeten la creació d'aplicacions informàtiques a persones sense coneixements tècnics d'informàtica, pel fet d'existir una creació pràcticament automàtica del codi a partir d'unes preguntes.

Els diferents tipus de llenguatges són:

- Llenguatge de primera generació o llenguatge màquina.
- Llenguatges de segona generació o llenguatges d'assemblador.
- Llenguatges de tercera generació o llenguatges d'alt nivell.
- Llenguatges de quarta generació o llenguatges de propòsit específic.
- Llenguatges de cinquena generació.

El primer tipus de llenguatge que es va desenvolupar és l'anomenat **llenguatge de primera generació o llenguatge màquina**. És l'únic llenguatge que entén l'ordinador directament.

La seva estructura està totalment adaptada als circuits impresos dels ordinadors o processadors electrònics i molt allunyada de la forma d'expressió i anàlisi dels problemes propis dels humans (les instruccions s'expressen en codi binari). Això fa que la programació en aquest llenguatge resulti tediosa i complicada, ja que es requereix un coneixement profund de l'arquitectura física de l'ordinador. A més, s'ha de valorar que el codi màquina fa possible que el programador utilitzi la totalitat de recursos del maquinari, amb la qual cosa es poden obtenir programes molt eficients.

```
10110000 01100001
Aquesta línia conté una instrucció que mou un valor al registre del processador
    .
```

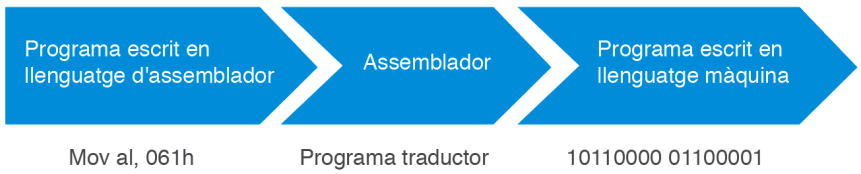
Actualment, a causa de la complexitat del desenvolupament d'aquest tipus de llenguatge, està pràcticament en desús. Només es farà servir en processadors molts concrets o per a funcionalitats molt específiques.

El segon tipus de llenguatge de programació són els **llenguatges de segona generació o llenguatges d'assemblador**. Es tracta del primer llenguatge de programació que utilitza codis mnemotècnics per indicar a la màquina les operacions que ha de dur a terme. Aquestes operacions, molt bàsiques, han estat dissenyades a partir de la coneixença de l'estructura interna de la pròpia màquina.

Cada instrucció en llenguatge d'assemblador correspon a una instrucció en llenguatge màquina. Aquests tipus de llenguatges depenen totalment del processador que utilitzi la màquina, per això es diu que estan orientats a les màquines.

A la figura 1.3 es mostra un esquema del funcionament dels llenguatges de segona generació. A partir del codi escrit en llenguatge d'assemblador, el programa traductor (assemblador) ho converteix en codi de primera generació, que serà interpretat per la màquina.

Figura 1.3. Llenguatge de segona generació



En general s'utilitza aquest tipus de llenguatges per programar controladors (*drivers*) o aplicacions de temps real, ja que requereix un ús molt eficient de la velocitat i de la memòria.

1.3.1 Característiques dels llenguatges de primera i segona generació

Com a avantatges dels llenguatges de primera i segona generació es poden establir:

- Permeten escriure programes molt optimitzats que aprofiten al màxim el maquinari (*hardware*) disponible.
- Permeten al programador especificar exactament quines instruccions vol que s'executin.

Els inconvenients són els següents:

- Els programes escrits en llenguatges de baix nivell estan completament lligats al maquinari on s'executaran i no es poden traslladar fàcilment a altres sistemes amb un maquinari diferent.
- Cal conèixer a fons l'arquitectura del sistema i del processador per escriure bons programes.
- No permeten expressar de forma directa conceptes habituals a nivell d'algorisme.
- Son difícils de codificar, documentar i mantenir.

El següent grup de llenguatges es coneix com a **llenguatges de tercera generació o llenguatges d'alt nivell**. Aquests llenguatges, més evolucionats, utilitzen paraules i frases relativament fàcils d'entendre i proporcionen també facilitats per expressar alteracions del flux de control d'una forma bastant senzilla i intuïtiva.

Els llenguatges de tercera generació o d'alt nivell s'utilitzen quan es vol desenvolupar aplicacions grans i complexes, on es prioritza el fet de facilitar i comprendre com fer les coses (llenguatge humà) per sobre del rendiment del programari o del seu ús de la memòria.

Els esforços encaminats a fer la tasca de programació independent de la màquina on s'executaran van donar com a resultat l'aparició dels llenguatges de programació d'alt nivell.

Els llenguatges d'alt nivell són normalment fàcils d'aprendre perquè estan formats per elements de llenguatges naturals, com ara l'anglès. A continuació es mostra un exemple d'algorisme implementat en un llenguatge d'alt nivell, concretament en Basic. Aquest algorisme calcula el factorial d'un nombre donat a la funció com a paràmetre. Es pot observar com és fàcilment comprensible amb un mínim coneixement de l'anglès.

En Basic, el llenguatge d'alt nivell més conegut, les ordres com `IF comptador = 10 THEN STOP` poden utilitzar-se per demanar a l'ordinador que pari si la variable comptador és igual a deu.

```
' ____________________________________________
' Funció Factorial
' ____________________________________________
Public Function Factorial(num As Integer)As String
Dim i As Integer
    For i = 1 To num - 1
      num = num * i
      Factorial = num
    Next
End Function
```

Com a conseqüència d'aquest allunyament de la màquina i acostament a les persones, els programes escrits en llenguatges de programació de tercera generació no poden ser interpretats directament per l'ordinador, sinó que és necessari dur a terme prèviament la seva traducció a llenguatge màquina. Hi ha dos tipus de traductors: els compiladors i els intèrprets.

Compiladors

Són programes que tradueixen el programa escrit amb un llenguatge d'alt nivell al llenguatge màquina. El compilador detectarà els possibles errors del programa font per aconseguir un programa executable depurat.

Alguns exemples de codis de programació que hauran de passar per un compilador són: Pascal, C, C++, .NET, ...

A la figura 1.4 es pot veure, en un esquema, la funció del compilador entre els dos llenguatges.

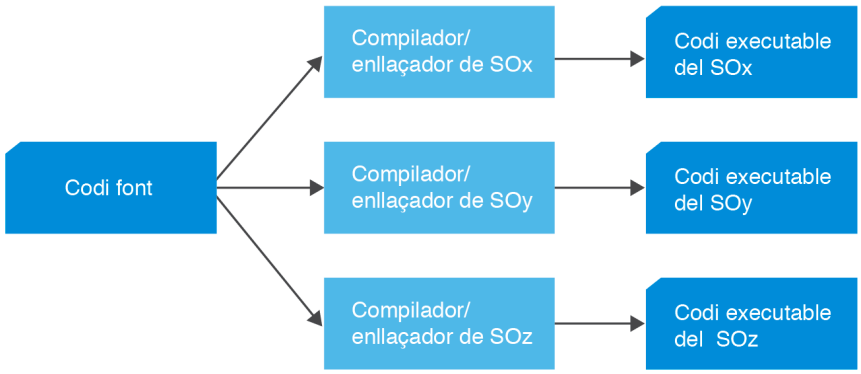
FIGURA 1.4. Codi compilat

El procediment que haurà de seguir un programador és el següent:

- Crear el codi font.
- Crear el codi executable fent ús de compiladors i enllaçadors.
- El codi executable depèn de cada sistema operatiu. Per a cada sistema hi ha un compilador, és a dir, si es vol executar el codi amb un altre sistema operatiu s'ha de recompilar el codi font.
- El programa resultant s'executa directament des del sistema operatiu.

A la figura 1.5 es pot observar un esquema que representa la dependència del sistema operatiu a l'hora d'escollir i utilitzar compilador.

FIGURA 1.5. Codi compilat per SO



Els intèrprets

L'intèrpret també és un programa que tradueix el codi d'alt nivell al llenguatge màquina, però, a diferència del compilador, ho fa en temps d'execució. És a dir, no es fa un procés previ de traducció de tot el programa font a codi de bytes, sinó que es va traduint i executant instrucció per instrucció.

Alguns exemples de codis de programació que hauran de passar per un intèrpret són: JavaScript, PHP, ASP...

Algunes característiques dels llenguatges interpretats són:

- El codi interpretat no és executat directament pel sistema operatiu, sinó que fa ús d'un intèrpret.
- Cada sistema té el seu propi intèrpret.

Compiladors davant intèrprets

L'intèrpret és notablement més lent que el compilador, ja que du a terme la traducció alhora que l'execució. A més, aquesta traducció es fa sempre que s'executa el programa, mentre que el compilador només la du a terme una vegada. L'avantatge dels intèrprets és que fan que els programes siguin més portables. Així, un programa compilat en un ordinador amb sistema operatiu Windows no funcionarà en un Macintosh, o en un ordinador amb sistema operatiu Linux, a menys que es torni a compilar el programa font en el nou sistema.

1.3.2 Característiques dels llenguatges de tercera, quarta i cinquena generació

Els llenguatges de tercera generació són aquells que són capaços de contenir i executar, en una sola instrucció, l'equivalent a diverses instruccions d'un llenguatge de segona generació.

Els avantatges dels llenguatges de tercera generació són:

- El codi dels programes és molt més senzill i comprensible.
- Són independents del maquinari (no hi fan cap referència). Per aquest motiu és possible "portar" el programa entre diferents ordinadors / arquitectures / sistemes operatius (sempre que en el sistema de destinació existeixi un compilador per a aquest llenguatge d'alt nivell).
- És més fàcil i ràpid escriure els programes i més fàcil mantenir-los.

Els inconvenients dels llenguatges de tercera generació són:

- La seva execució en un ordinador pot resultar més lenta que el mateix programa escrit en llenguatge de baix nivell, tot i que això depèn molt de la qualitat del compilador que faci la traducció.

Exemples de llenguatges de programació de tercera generació: C, C++, Java, Pascal...

Els **llenguatges de quarta generació o llenguatges de propòsit específic**. Aporten un nivell molt alt d'abstracció en la programació, permetent desenvolupar aplicacions sofisticades en un espai curt de temps, molt inferior al necessari per als llenguatges de 3a generació.

IDE són les sigles en anglès d'Integrated Development Environment, és a dir, Entorn Integrat de Desenvolupament.

S'automatitzen certs aspectes que abans calia fer a mà. Inclouen eines orientades al desenvolupament d'aplicacions (IDE) que permeten definir i gestionar bases de dades, dur a terme informes (p.ex.: Oracle reports), consultes (p.ex.: informix 4GL), mòduls... , escrivint molt poques línies de codi o cap.

Permeten la creació de prototipus d'una aplicació ràpidament. Els prototipus permeten tenir una idea de l'aspecte i del funcionament de l'aplicació abans que el codi estigui acabat. Això facilita l'obtenció d'un programa que reuneixi les necessitats i expectatives del client.

Alguns dels aspectes positius que mostren aquest tipus de llenguatges de programació són:

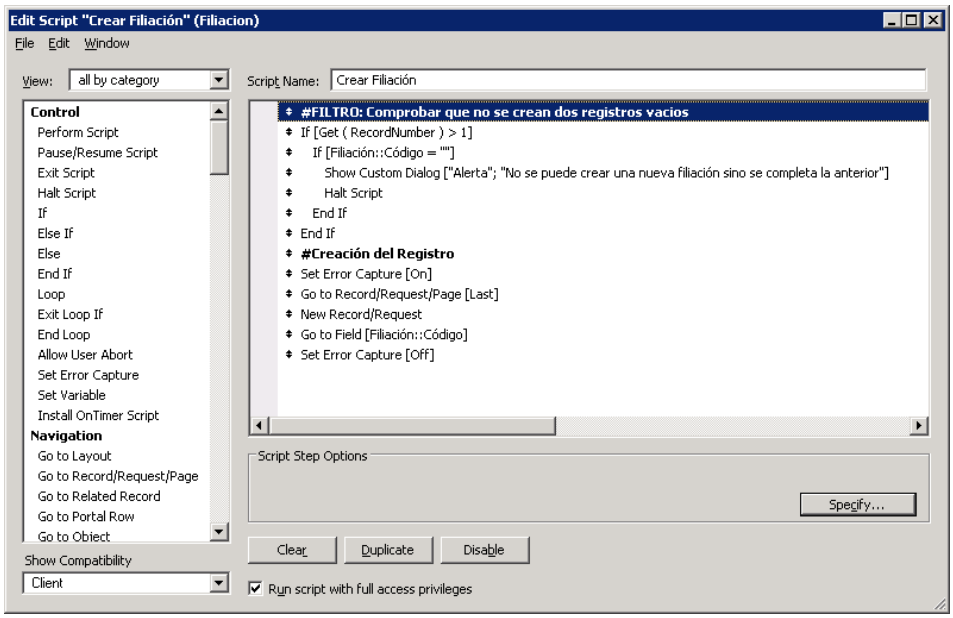
- Major abstracció.
- Menor esforç de programació.
- Menor cost de desenvolupament del programari.
- Basats en generació de codi a partir d'especificacions de nivell molt alt.
- Es poden dur a terme aplicacions sense ser un expert en el llenguatge.
- Solen tenir un conjunt d'instruccions limitat.
- Són específics del producte que els ofereix.

Aquests llenguatges de programació de quarta generació estan orientats, bàsicament, a les aplicacions de negoci i al maneig de bases de dades.

Alguns exemples de llenguatges de quarta generació són Visual Basic, Visual Basic .NET, ABAP de SAP, FileMaker, PHP, ASP, 4D...

A la figura 1.6 es pot veure l'entorn de treball i un exemple de codi font de FileMaker.

FIGURA 1.6. FileMaker: Llenguatge de quarta generació



> Els **llenguatges de cinquena generació** són llenguatges específics per al tractament de problemes relacionats amb la intel·ligència artificial i els sistemes experts.

En lloc d'executar només un conjunt d'ordres, l'objectiu d'aquests sistemes és "pensar" i anticipar les necessitats dels usuaris. Aquests sistemes es troben encara en desenvolupament. Es tractaria del paradigma lògic.

Alguns exemples de llenguatges de cinquena generació són Lisp o Prolog.

1.4 Paradigmes de programació

És difícil establir una classificació general dels llenguatges de programació, ja que existeix un gran nombre de llenguatges i, de vegades, diferents versions d'un mateix llenguatge. Això provocarà que en qualsevol classificació que es faci un mateix llenguatge pertanyi a més d'un dels grups establerts. Una classificació molt estesa, atenent a la forma de treballar dels programes i a la filosofia amb què van ser concebuts, és la següent:

- Paradigma imperatiu/estructurat.
- Paradigma d'objectes.
- Paradigma funcional.
- Paradigma lògic.

El **paradigma imperatiu/estructurat** deu el seu nom al paper dominant que exerceixen les sentències imperatives, és a dir aquelles que indiquen dur a terme una determinada operació que modifica les dades guardades en memòria.

Alguns dels llenguatges imperatius són C, Basic, Pascal, Cobol...

La tècnica seguida en la programació imperativa és la **programació estructurada**. La idea és que qualsevol programa, per complex i gran que sigui, pot ser representat mitjançant tres tipus d'estructures de control:

- Seqüència.
- Selecció.
- Iteració.

D'altra banda, també es proposa desenvolupar el programa amb la tècnica de disseny descendent (*top-down*). És a dir, modular el programa creant porcions més petites de programes amb tasques específiques, que se subdivideixen en altres subprogrames, cada vegada més petits. La idea és que aquests subprogrames típicament anomenats funcions o procediments han de resoldre un únic objectiu o tasca.

Imaginem que hem de fer una aplicació que registri les dades bàsiques del personal d'una escola, dades com poden ser el nom, el DNI, i que calculi el salari dels professors així com el dels administratius, on el salari dels administratius és el sou base (`SOU_BASE`) * 10 mentre que el salari dels professors és el sou base (`SOU_BASE`) + nombre d'hores impartides (`numHores`) * 12.

```
const float SOU_BASE = 1.000;

Struct Administratiu
{
    string nom;
    string DNI;
    float Salari;
}

Struct Professor
{
    string nom;
    string DNI;
    int numHores;
    float salari;
}

void AssignarSalariAdministratiu (Administratiu administratiu1)
```

```
{
    administratiu1. salari = SOU_BASE * 10;
}

void AssignarSalariProfessor (Professor professor1)
{
    professor1. salari = SOU_BASE + (numHores * 12);
}
```

El **paradigma d'objectes**, típicament conegut com a Programació Orientada a Objectes (POO, o OOP en anglès), és un paradigma de construcció de programes basat en una abstracció del món real. En un programa orientat a objectes, l'abstracció no són procediments ni funcions sinó els objectes. Aquests objectes són una representació directa d'alguna cosa del món real, com ara un llibre, una persona, una comanda, un empleat...

Alguns dels llenguatges de programació orientada a objectes són C++, Java, C#...

Un objecte és una combinació de dades (anomenades atributs) i mètodes (funcions i procediments) que ens permeten interactuar amb ell. En aquest tipus de programació, per tant, els programes són conjunts d'objectes que interactuen entre ells a través de missatges (crides a mètodes).

La programació orientada a objectes es basa en la integració de 5 conceptes: abstracció, encapsulació, modularitat, jerarquia i polimorfisme, que és necessari comprendre i seguir de manera absolutament rigorosa. No seguir-los sistemàticament, ometre'ls puntualment per pressa o altres raons fa perdre tot el valor i els beneficis que ens aporta l'orientació a objectes.

```
class Treballador {
   private:
      string nom;
      string DNI;
   protected:
      static const float SOU_BASE = 1.000;
   public:
      string GetNom() {return this.nom;}
      void SetNom (string n) {this.nom = n;}
      string GetDNI() {return this.DNI;}
      void SetDNI (string dni) {this.DNI = dni;}
      virtual float salari() = 0;
}

class Administratiu: public Treballador {
   public:
      float Salari() {return SOU_BASE * 10};
}

class Professor: public Treballador {
   private:
      int numHores;
   public:
      float Salari() {return SOU_BASE + (numHores * 15);}
}
```

El **paradigma funcional** està basat en un model matemàtic. La idea és que el resultat d'un càlcul és l'entrada del següent, i així successivament fins que una composició produeixi el resultat desitjat.

Els creadors dels primers llenguatges funcionals pretenien convertir-los en llenguatges d'ús universal per al processament de dades en tot tipus d'aplicacions, però, amb el pas del temps, s'ha utilitzat principalment en àmbits d'investigació científica i aplicacions matemàtiques.

Un dels llenguatges més típics del paradigma funcional és el Lisp. Vegeu un exemple de programació del factorial amb aquest llenguatge:

```
> (defun factorial (n)
(if (= n 0)
1
(* n (factorial (- n 1)))))
FACTORIAL
> (factorial 3)
6
```

El **paradigma lògic** té com a característica principal l'aplicació de les regles de la lògica per inferir conclusions a partir de dades.

Un programa lògic conté una base de coneixement sobre la que es duen a terme consultes. La base de coneixement està formada per fets, que representen la informació del sistema expressada com a relacions entre les dades i regles lògiques que permeten deduir conseqüències a partir de combinacions entre els fets i, en general, altres regles.

Un dels llenguatges més típics del paradigma lògic és el Prolog.

Exemple de desplegament pràctic del paradigma lògic

Determinarem si hem de prescriure al pacient estar a casa reposant al saber que es compleixen els següents fets: malestar i 39º de temperatura corporal.

Regles de la base de coneixement:

- R1: Si febre, llavors estar a casa en repòs.
- R2: Si malestar, llavors posar-se termòmetre.
- R3: Si termòmetre marca una temperatura > 37º, llavors febre.
- R4: Si diarrea, llavors dieta.

Si seguim un raonament d'encadenament cap endavant, el procediment seria:

```
Indicar el motor d'inferència, els fets: malestar i termòmetre
    marca 39.

<code>Base de fets = { malestar, termòmetre marca º39 }
```

El sistema identifica les regles aplicables: R2 i R3. L'algorisme s'inicia aplicant la regla R2, incorporant en la base de fets "posar-se el termòmetre".

```
Base de fets = { malestar, termòmetre marca º39, posar–se termò
    metre }
```

Com que no s'ha solucionat el problema, continua amb la següent regla R3, afegint a la base de fets "febre".

```
Base de fets = { malestar, termòmetre marca º39, posar–se termò
    metre, febre }
```

Com que no s'ha solucionat el problema, torna a identificar un subconjunt de regles aplicables, excepte les ja utilitzades. El sistema identifica les regles aplicables: R1, tot incorporant a la base de fets "estar a casa en repòs".

```
Base de fets = { malestar, termòmetre marca º39, posar–se termò
    metre, febre, estar a casa en repòs}
```

Com que repòs està a la base de fets, s'ha arribat a una resposta positiva a la pregunta formulada.

El paradigma és àmpliament utilitzat en les aplicacions que tenen a veure amb la Intel·ligència Artificial, particularment en el camp de sistemes experts i processament del llenguatge humà. Un sistema expert és un programa que imita el comportament d'un expert humà. Per tant conté informació (és a dir una base de coneixements) i una eina per comprendre les preguntes i trobar la resposta correcta examinant la base de dades (un motor d'inferència).

També és útil en problemes combinatoris o que requereixin una gran quantitat o amplitud de solucions alternatives, d'acord amb la naturalesa del mecanisme de tornada enrere (*backtracking*).

1.5 Característiques dels llenguatges més difosos

Existeixen molts llenguatges de programació diferents, fins al punt que moltes tecnologies tenen el seu llenguatge propi. Cada un d'aquests llenguatges té un seguit de particularitats que el fan diferent de la resta.

Els llenguatges de programació més difosos són aquells que més es fan servir en cadascun dels diferents àmbits de la informàtica. En l'àmbit educatiu, per exemple, es considera un llenguatge de programació molt difós aquell que es fa servir a moltes universitats o centres educatius per a la docència de la iniciació a la programació.

Els llenguatges de programació més difosos corresponents a diferents àmbits, a diferents tecnologies o a diferents tipus de programació tenen una sèrie de característiques en comú que són les que marquen les similituds entre tots ells.

1.5.1 Característiques de la programació estructurada

La programació estructurada va ser desenvolupada pel neerlandès Edsger W. Dijkstra i es basa en el denominat teorema de l'estructura. Per això utilitza únicament tres estructures: seqüència, selecció i iteració, essent innecessari l'ús de la instrucció o instruccions de transferència incondicional (`GOTO`, `EXIT FUNCTION`, `EXIT SUB` o múltiples `RETURN`).

D'aquesta forma les característiques de la programació estructurada són la claredat, el teorema de l'estructura i el disseny descendent.

Claredat

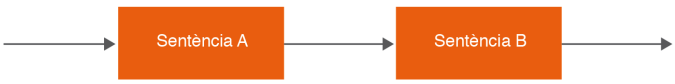
Hi haurà d'haver prou informació al codi per tal que el programa pugui ser entès i verificat: comentaris, noms de variables comprensibles i procediments entenedors... Tot programa estructurat pot ser llegit des del principi a la fi sense interrupcions en la seqüència normal de lectura.

Teorema de l'estructura

Demostra que tot programa es pot escriure utilitzant únicament les tres estructures bàsiques de control:

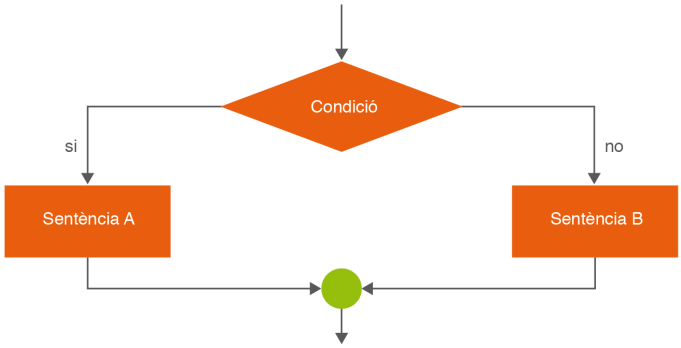
- Seqüència: instruccions executades successivament, una darrere l'altra. A la figura 1.7 es pot observar un exemple de l'estructura bàsica de seqüència, on primer s'executarà la sentència A i, posteriorment, la B.

FIGURA 1.7. Exemple de seqüència



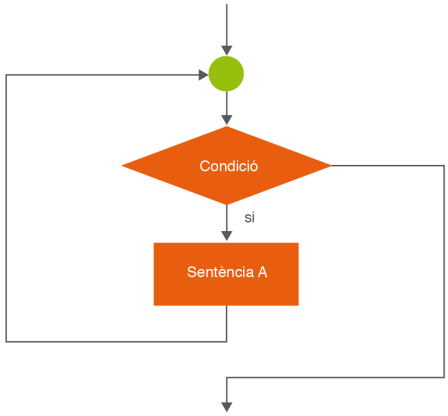
- Selecció: la instrucció condicional amb doble alternativa, de la forma "si condició, llavors SentènciaA, sinó SentènciaB". A la figura 1.8 es pot observar un esquema que exemplifica l'estructura bàsica de selecció.

FIGURA 1.8. Exemple de selecció



- Iteració: el bucle condicional "mentre condició, fes SentènciaA", que executa les instruccions repetidament mentre la condició es compleixi. A la figura 1.9 es pot observar un esquema que exemplifica l'estructura bàsica d'iteració.

FIGURA 1.9. Exemple d'iteració

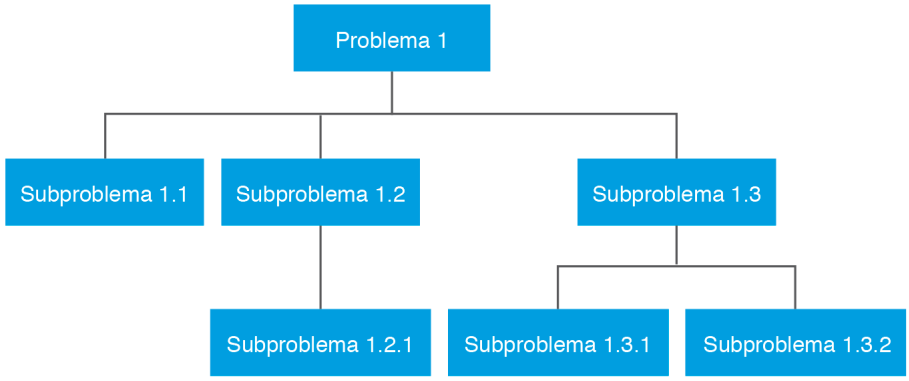


Disseny descendent

El disseny descendent és una tècnica que es basa en el concepte de "divideix i venceràs" per tal de resoldre un problema en l'àmbit de la programació. Es tracta de la resolució del problema al llarg de diferents nivells d'abstracció partint d'un nivell més abstracte i finalitzant en un nivell de detall.

A la figura 1.10 es pot observar un exemple del disseny descendent. A partir del problema1 s'obtenen diversos subproblemes (subproblema 1.1, subproblema 1.2 i subproblema 1.3). La resolució d'aquests subproblemes serà molt més senzilla que la del problema original per tal com se n'ha reduït considerablement l'abast i la mida. De forma iterativa es pot observar com aquests subproblemes es tornen a dividir, a la vegada, en altres subproblemes.

FIGURA 1.10. Disseny descendent



La visió moderna de la programació estructurada introdueix les característiques de programació modular i tipus abstractes de dades (TAD).

Programació modular

La realització d'un programa sense seguir una tècnica de programació modular produeix sovint un conjunt enorme de sentències l'execució de les quals és complexa de seguir, i d'entendre, amb la qual cosa es fa gairebé impossible la depuració d'errors i la introducció de millores. Fins i tot, es pot donar el cas d'haver d'abandonar el codi preexistent perquè resulta més fàcil començar de nou.

Quan es parla de programació modular, ens referim a la divisió d'un programa en parts més manejables i independents. Una regla pràctica per aconseguir aquest propòsit és establir que cada segment del programa no excedeixi, en longitud, d'un pam de codificació.

En la majoria de llenguatges, els mòduls es tradueixen a:

- Procediments: són subprogrames que duen a terme una tasca determinada i retornen 0 o més d'un valor. S'utilitzen per estructurar un programa i millorar la seva claredat.
- Funcions: són subprogrames que duen a terme una determinada tasca i retornen un únic resultat o valor. S'utilitzen per crear operacions noves que no ofereix el llenguatge.

Tipus abstractes de dades (TAD)

En programació, el *tipus de dades* d'una variable és el conjunt de valors que la variable pot assumir. Per exemple, una variable de tipus booleà pot adoptar només dos valors possibles: `vertader` o `fals`. A més, hi ha un conjunt limitat però ben definit d'operacions que tenen sentit sobre els valors d'un tipus de dades; així, operacions típiques sobre el tipus booleà són `AND` o `OR`.

Els llenguatges de programació assumeixen un nombre determinat de tipus de dades, que pot variar d'un llenguatge a un altre; així, en Pascal tenim els *enters*, els *reals*, els *booleans*, els *caràcters*... Aquests tipus de dades són anomenats *tipus de dades bàsics* en el context dels llenguatges de programació.

Fins fa uns anys, tota la programació es basava en aquest concepte de tipus i no eren pocs els problemes que apareixien, lligats molt especialment a la complexitat de les dades que s'havien de definir. Va aparèixer la possibilitat de poder definir *tipus abstractes de dades*, on el programador pot definir un nou tipus de dades i les seves possibles operacions.

Exemple d'implementació d'un tipus abstracte de dades implementat en el llenguatge C

```
struct TADpila
{
        int top;
        int elements[MAX_PILA];
}

void crear(struct TADpila *pila)
{
        Pila.top = -1;
}

void apilar(struct TADpila *pila, int elem)
{
        Pila.elements[pila.top++] = elem;
}

void desapilar(struct TADpila *pila)
{
        Pila.top--;
}
```

1.5.2 Característiques de la programació orientada a objectes

Un dels conceptes importants introduïts per la programació estructurada és l'abstracció de funcionalitats a través de funcions i procediments. Aquesta abstracció permet a un programador utilitzar una funció o procediment coneixent només què fa, però desconeixent el detall de com ho fa.

Aquest fet, però, té diversos inconvenients:

- Les funcions i procediments comparteixen dades del programa, cosa que provoca que canvis en un d'ells afectin a la resta.
- Al moment de dissenyar una aplicació és molt difícil preveure detalladament quines funcions i procediments necessitarem.
- La reutilització del codi és difícil i acaba consistint a copiar i enganxar determinats trossos de codi, i retocar-los. Això és especialment habitual quan el codi no és modular.

L'orientació a objectes, concebut als anys setanta i vuitanta però estesa a partir dels noranta, va permetre superar aquestes limitacions.

L'**orientació a objectes** (en endavant, OO) és un paradigma de construcció de programes basat en una abstracció del món real.

En un programa orientat a objectes, l'abstracció no són els procediments ni les funcions, són els objectes. Aquests objectes són una representació directa d'alguna cosa del món real, com ara un llibre, una persona, una organització, una comanda, un empleat...

Un **objecte** és una combinació de dades (anomenades atributs) i mètodes (funcions i procediments) que ens permeten interactuar amb ell. En OO, doncs, els programes són conjunts d'objectes que interactuen entre ells a través de missatges (crides a mètodes).

Els llenguatges de POO (programació orientada a objectes) són aquells que implementen més o menys fidelment el paradigma OO. La programació orientada a objectes es basa en la integració de 5 conceptes: abstracció, encapsulació, modularitat, jerarquia i polimorfisme, que és necessari comprendre i seguir de manera absolutament rigorosa. No seguir-los sistemàticament o ometre'ls puntualment, per pressa o altres raons, fa perdre tot el valor i els beneficis que aporta l'orientació a objectes.

Abstracció

És el procés en el qual se separen les propietats més importants d'un objecte de les que no ho són. És a dir, per mitjà de l'abstracció es defineixen les característiques essencials d'un objecte del món real, els atributs i comportaments que el defineixen com a tal, per després modelar-lo en un objecte de programari. En el procés d'abstracció no ha de ser preocupant la implementació de cada mètode o atribut, només cal definir-los.

En la tecnologia orientada a objectes l'eina principal per suportar l'abstracció és la **classe**. Es pot definir una classe com una descripció genèrica d'un grup d'objectes que comparteixen característiques comunes, les quals són especificades en els seus atributs i comportaments.

Encapsulació

Permet als objectes triar quina informació és publicada i quina informació és amagada a la resta dels objectes. Per això els objectes solen presentar els seus mètodes com a interfícies públiques i els seus atributs com a dades privades o protegides, essent inaccessibles des d'altres objectes. Les característiques que es poden atorgar són:

- Públic: qualsevol classe pot accedir a qualsevol atribut o mètode declarat com a públic i utilitzar-lo.
- Protegit: qualsevol classe heretada pot accedir a qualsevol atribut o mètode declarat com a protegit a la classe mare i utilitzar-lo.
- Privat: cap classe no pot accedir a un atribut o mètode declarat com a privat i utilitzar-lo.

Modularitat

Permet poder modificar les característiques de cada una de les classes que defineixen un objecte, de forma independent de la resta de classes en l'aplicació. En altres paraules, si una aplicació es pot dividir en mòduls separats, normalment classes, i aquests mòduls es poden compilar i modificar sense afectar els altres, aleshores aquesta aplicació ha estat implementada en un llenguatge de programació que suporta la modularitat.

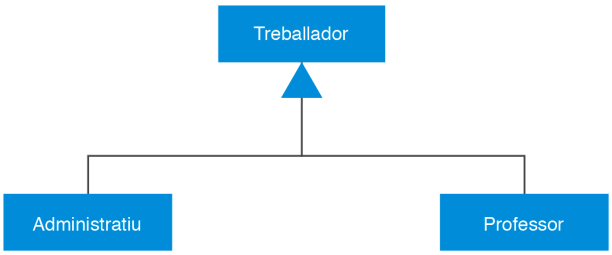
Jerarquia

Permet l'ordenació de les abstraccions. Les dues jerarquies més importants d'un sistema complex són l'herència i l'agregació.

L'herència també es pot veure com una forma de compartir codi, de manera que quan s'utilitza l'herència per definir una nova classe només s'ha d'afegir allò que sigui diferent, és a dir, reaprofita els mètodes i variables, i especialitza el comportament.

Per exemple, es pot identificar una classe *pare* anomenada `treballador` i dues classes *filles*, és a dir dos subtipus de treballadors, `administratiu` i `professor`.

FIGURA 1.11. Exemple d'herència

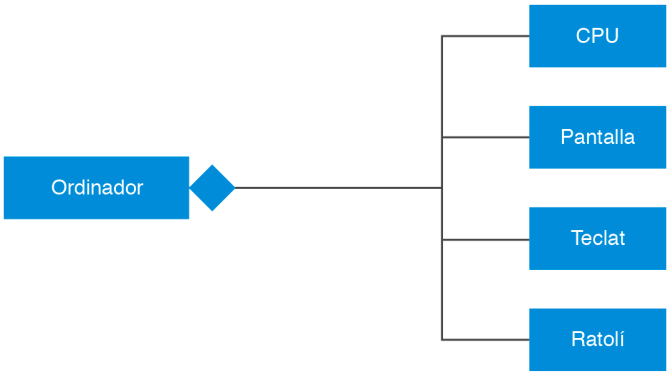


A la figura 1.11 es pot observar la representació en forma de diagrama de l'exemple explicat anteriorment: les classes `administratiu` i `professor` que hereten de la classe `treballador`.

L'agregació és un objecte que està format de la combinació d'altres objectes o components. Així, un ordinador es compon d'una CPU, una pantalla, un teclat i un ratolí, i aquests components no tenen sentit sense l'ordinador. A la figura 1.12

es pot observar un exemple d'agregació en què la classe `ordinador` està composta per les altres quatre classes.

FIGURA 1.12. Exemple d'agregació



El polimorfisme

És una característica que permet donar diferents formes a un mètode, ja sigui en la definició com en la implementació.

La sobrecàrrega (*overload*) de mètodes consisteix a implementar diverses vegades un mateix mètode però amb paràmetres diferents, de manera que, en invocar-lo, el compilador decideix quin dels mètodes s'ha d'executar, en funció dels paràmetres de la crida.

Un exemple de mètode sobrecarregat és aquell que calcula el salari d'un treballador en una empresa. En funció de la posició que ocupa el treballador tindrà més o menys conceptes a la seva nòmina (més o menys incentius, per exemple).

El mateix mètode, que podríem anomenar *CàlculSalari* quedarà implementat de forma diferent en funció de si es calcula el salari d'un operari (amb menys conceptes en la seva nòmina, la qual cosa provoca que el mètode rebi menys variables) o si es calcula el salari d'un directiu.

La sobreescriptura (*override*) de mètodes consisteix a reimplementar un mètode heretat d'una superclasse exactament amb la mateixa definició (incloent nom de mètode, paràmetres i valor de retorn).

Un exemple de sobrecàrrega de mètodes podria ser el del mètode `Area()`. A partir d'una classe `Figura` que conté el mètode `Area()`, existeix una classe derivada per a alguns tipus de figures (per exemple, `Rectangle` o `Quadrat`).

La implementació del mètode `Area()` serà diferent a cada una de les classes derivades; aquestes poden implementar-se de forma diferent (en funció de com es calculi en cada cas l'àrea de la figura) o definir-se de forma diferent.

1.6 Fases del desenvolupament dels sistemes d'informació

Una aplicació informàtica necessitarà moltes petites accions (i no tan petites) per ser creada. S'han desenvolupat moltes metodologies que ofereixen un acompanyament al llarg d'aquest desenvolupament, proporcionant pautes, indicacions, mètodes i documents per ajudar, sobretot, els caps de projecte més inexperts.

Dintre d'aquestes metodologies hi ha Mètrica v3.0. Ha estat desenvolupada pel Ministeri d'Administracions Públiques. Es tracta d'una metodologia per a la planificació, desenvolupament i manteniment dels sistemes d'informació d'una organització. Per al desenvolupament de programari cal fixar-se en la part que fa referència al desenvolupament dels sistemes d'informació (SI), dins la metodologia Mètrica. Divideix el desenvolupament en 5 fases, que se segueixen de forma seqüencial.

També és important tenir clarament identificats els rols dels components de l'equip de projecte que participaran en el desenvolupament de l'aplicació informàtica. A Mètrica aquests perfils són:

- Parts interessades (*stakeholders*)
- Cap de Projecte
- Consultors
- Analistes
- Programadors

A la figura 1.13 hi podeu observar les cinc fases principals de la metodologia Mètrica v3.0.

FIGURA 1.13. Fases de desenvolupament d'una aplicació

1.6.1 Estudi de viabilitat del sistema

El propòsit d'aquest procés és analitzar un conjunt concret de necessitats, amb la idea de proposar una solució a curt termini. Els criteris amb què es fa aquesta proposta no seran estratègics sinó tàctics i relacionats amb aspectes econòmics, tècnics, legals i operatius.

Els **resultats** de l'estudi de viabilitat del sistema constituiran la base per prendre la decisió de seguir endavant o abandonar el projecte.

1.6.2 Anàlisi del sistema d'informació

El propòsit d'aquest procés és aconseguir l'**especificació detallada** del sistema d'informació, per mitjà d'un catàleg de requisits i d'una sèrie de models que cobreixin les necessitats d'informació dels usuaris per als quals es desenvoluparà el sistema d'informació i que seran l'entrada per al procés de Disseny del sistema d'informació.

En primer lloc, es descriu el sistema d'informació, a partir de la informació obtinguda en l'estudi de viabilitat. Es delimita el seu abast, es genera un catàleg de requisits generals i es descriu el sistema mitjançant uns models inicials d'alt nivell.

Es recullen de forma detallada els requisits funcionals que el sistema d'informació ha de cobrir. A més, s'identifiquen els requisits no funcionals del sistema, és a dir, les facilitats que ha de proporcionar el sistema, i les restriccions a què estarà sotmès, quant a rendiment, freqüència de tractament, seguretat...

Normalment, per tal d'efectuar l'anàlisi se sol elaborar els models *de casos d'ús* i *de classes*, en desenvolupaments orientats a objectes, i *de dades i processos* en desenvolupaments estructurats. D'altra banda, s'aconsella dur a terme una definició d'interfícies d'usuari, ja que facilitarà la comunicació amb els usuaris clau.

1.6.3 Disseny del sistema d'informació

El propòsit del **disseny** és obtenir la definició de l'arquitectura del sistema i de l'entorn tecnològic que li donarà suport, juntament amb l'especificació detallada dels components del sistema d'informació. A partir d'aquesta informació, es generen totes les especificacions de construcció relatives al propi sistema, així com l'especificació tècnica del pla de proves, la definició dels requisits d'implantació i el disseny dels procediments de migració i càrrega inicial.

En el disseny es generen les especificacions necessàries per a la construcció del sistema d'informació, com per exemple:

- Els components del sistema (mòduls o classes, segons el cas) i de les estructures de dades.
- Els procediments de migració i els seus components associats.
- La definició i revisió del pla de proves, i el disseny de les verificacions dels nivells de prova establerts.
- El catàleg d'excepcions, que permet establir un conjunt de verificacions relacionades amb el propi disseny o amb l'arquitectura del sistema.
- L'especificació dels requisits d'implantació.

1.6.4 Construcció del sistema d'informació

La **construcció del sistema d'informació** té com a objectiu final la construcció i la prova dels diferents components del sistema d'informació, a partir del seu conjunt d'especificacions lògiques i físiques, obtingut en la fase de disseny. Es desenvolupen els procediments d'operació i de seguretat, i s'elaboren els manuals d'usuari final i d'explotació, aquests últims quan sigui procedent.

Per aconseguir aquest objectiu, es recull la informació elaborada durant la fase de disseny, es prepara l'entorn de construcció, es genera el codi de cada un dels components del sistema d'informació i es van duent a terme, a mesura que es vagi finalitzant la construcció, les proves unitàries de cada un d'ells i les d'integració entre subsistemes. Si fos necessari efectuar una migració de dades, és en aquest procés on es porta a terme la construcció dels components de migració i dels procediments de migració i càrrega inicial de dades.

1.6.5 Implantació i acceptació del sistema

Aquest procés té com a objectiu principal el **lliurament** i l'**acceptació** del sistema en la seva totalitat, que pot comprendre diversos sistemes d'informació desenvolupats de manera independent, i un segon objectiu, que és dur a terme les activitats oportunes per al pas a producció del sistema.

Un cop revisada l'estratègia d'implantació, s'estableix el pla d'implantació i es detalla l'equip que el portarà a terme.

Per a l'inici d'aquest procés es prenen com a punt de partida els components del sistema provats de forma unitària i integrats en el procés de construcció, així com la documentació associada. El sistema s'ha de sotmetre a les proves d'implantació

amb la participació de l'usuari d'operació. La responsabilitat, entre altres aspectes, és comprovar el comportament del sistema sota les condicions més extremes. El sistema també serà sotmès a les proves d'acceptació, que seran dutes a terme per l'usuari final.

En aquest procés s'elabora el pla de manteniment del sistema, de manera que el responsable del manteniment conegui el sistema abans que aquest passi a producció.

També s'estableix l'acord de nivell de servei requerit una vegada que s'iniciï la producció. L'acord de nivell de servei fa referència a serveis de gestió d'operacions, de suport a usuaris i al nivell amb què es prestaran aquests serveis.

2. Instal·lació i ús d'entorns de desenvolupament

Per poder elaborar una aplicació és necessari utilitzar una sèrie d'eines que ens permetin escriure-la, depurar-la, traduir-la i executar-la. Aquest conjunt d'eines es coneix com a entorn de desenvolupament integrat i la seva funció és proporcionar un marc de treball per al llenguatge de programació.

La selecció i una utilització òptima dels entorns de desenvolupament serà una decisió molt important en el procediment de creació de programari. L'entorn de desenvolupament és l'eina amb la qual el programador haurà de treballar durant la major part de temps que dediqui a la creació de noves aplicacions.

Si l'entorn de desenvolupament és el més adient per a un determinat llenguatge de programació i per al desenvolupament d'una aplicació determinada, i si el programador que la faci servir és coneixedor de la majoria de les funcionalitats i sap aprofitar totes les facilitats que ofereixi l'entorn, es podrà optimitzar el temps de desenvolupament de programari i facilitar l'obtenció d'un producte de qualitat.

2.1 Funcions d'un entorn de desenvolupament

IDE vol dir, en anglès, Integrated Development Environment.

Un **entorn de desenvolupament integrat**, o IDE, és un programa compost per una sèrie d'eines que utilitzen els programadors per desenvolupar codi. Aquest programa pot estar pensat per a la seva utilització amb un únic llenguatge de programació o bé pot donar cabuda a diversos.

GUI, en anglès, vol dir Graphical User Interface.

Les eines que normalment componen un entorn de desenvolupament integrat són un sistema d'ajuda per a la construcció d'interfícies gràfiques d'usuari (GUI), un editor de text, un compilador/intèrpret i un depurador.

2.1.1 Interfícies gràfiques d'usuari

Les interfícies gràfiques d'usuari són un conjunt de funcionalitats que permeten incorporar, editar i eliminar components gràfics de forma senzilla en l'aplicació que s'està desenvolupant. Aquests components facilitaran la interacció de l'usuari amb l'ordinador.

2.1.2 Editor de text

Un editor de text és un programa que permet crear i modificar arxius digitals compostos únicament per text sense format, coneguts comunament com arxius de text o text pla.

L'editor de text és l'eina més utilitzada perquè ofereix la possibilitat de crear i modificar els continguts desenvolupats, el codi de programació que farà funcionar adequadament l'aplicació.

Sempre que l'IDE reconegui el llenguatge de programació (o disposi del component adequat) l'editor oferirà:

- Ressaltat de sintaxi (*syntax highlighting*): les paraules clau seran reconegudes amb colors, cosa que facilitarà molt la feina del programador.
- Compleció de codi (*code completion*): es reconeixerà el codi que s'està escrivint i, per exemple en un objecte o classe, oferirà els seus atributs, propietats o mètodes per tal que el programador seleccioni quin vol referenciar.
- Corrector d'errors, normalment des d'un punt de vista de sintaxi.
- Regions plegables: ocultació de certes parts del codi, per tal de facilitar la visualització d'aquells fragments més rellevants.

2.1.3 Compilador

En funció del llenguatge de programació utilitzat, l'IDE podrà oferir la funcionalitat de compilar-lo. Un compilador tradueix un codi de programació en un llenguatge màquina capaç de ser interpretat pels processadors i de ser executat. A l'hora de compilar un codi de programació, els entorns integrats de desenvolupament disposaran de diferents fases d'anàlisi del codi, com són la fase semàntica i la fase lexicogràfica. La compilació mostrarà els errors trobats o generarà codi executable, en el cas de trobar-ne cap.

2.1.4 Intèrpret

L'intèrpret tradueix el codi d'alt nivell a codi de bytes, però, a diferència del compilador, ho fa en temps d'execució.

2.1.5 Depurador

El depurador és un programa que permet provar i depurar el codi font d'un programa, facilitant la detecció d'errors.

Algunes de les funcionalitats típiques dels depuradors són:

- Permetre l'execució línia a línia del codi validant els valors que van adquirint les variables.
- Pausar el programa en una determinada línia de codi, fent ús d'un o diversos punts de ruptura (*breakpoints*).
- Alguns depuradors ofereixen la possibilitat de poder modificar el contingut d'alguna variable mentre s'està executant.

Opcionalment, poden presentar un sistema d'accés a bases de dades i gestió d'arxius, un sistema de control de versions, un sistema de proves, un sistema de refactorització i un generador automàtic de documentació.

2.1.6 Accés a bases de dades i gestió d'arxius

Els llenguatges de quarta generació ofereixen la possibilitat d'integrar codi d'accés a bases de dades (o codi de sentències estructurades). Per facilitar aquesta funcionalitat és molt recomanable disposar de la possibilitat d'accedir directament a la base de dades des del mateix entorn de desenvolupament i no haver-ne d'utilitzar un altre. El mateix succeirà amb la gestió d'arxius.

2.1.7 Control de versions

A mesura que un programador va desenvolupant noves línies de codi, és important tenir un històric de la feina feta o de les versions que s'han donat per bones o s'han modificat. El control de versions permet tornar a una versió anterior que sigui estable o que compleixi unes determinades característiques que els canvis fets no compleixen.

2.1.8 Refactorització

La refactorització (*refactoring*) és una tècnica de l'enginyeria de programari dirigida a reestructurar un codi font, alterant la seva estructura interna sense canviar el seu comportament extern.

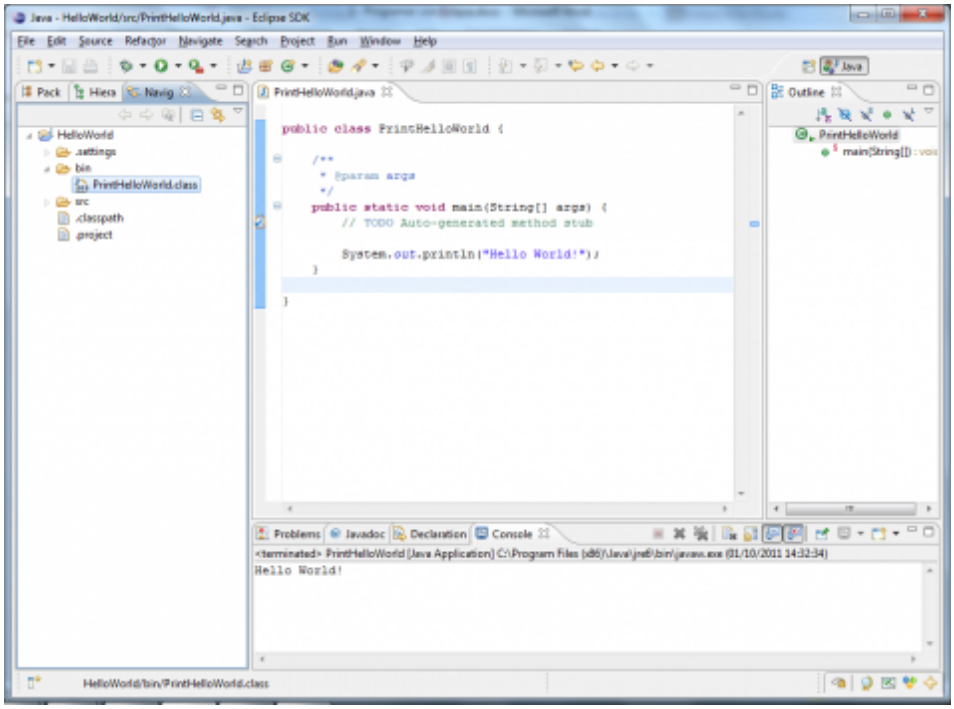
2.1.9 Documentació i ajuda

La documentació i ajuda proveeix accés a documentació, manuals i ajuda contextual sobre les instruccions i la sintaxi dels llenguatges suportats.

2.1.10 Exemples d'entorn integrat de desenvolupament

A la figura 2.1 es mostra un exemple d'un entorn integrat de desenvolupament. Concretament, es tracta de l'eina Eclipse.

FIGURA 2.1. Exemple d'entorn integrat de desenvolupament: Eclipse



Avui dia els entorns de desenvolupament proporcionen un marc de treball per a la majoria dels llenguatges de programació existents en el mercat (per exemple C, C++, C#, Java o Visual Basic, entre d'altres). A més, és possible que un mateix entorn de desenvolupament permeti utilitzar diversos llenguatges de programació, com és el cas d'Eclipse (al que es pot afegir suport de llenguatges addicionals mitjançant connectors *-plugins-*) o Visual Studio (que està pensat per treballar amb els llenguatges VB.Net, C#, C++...).

Com a exemples d'IDE multiplataforma es poden trobar, entre molts d'altres:

- Eclipse, projecte multiplataforma (Windows, Linux, Mac) de codi obert, fundat per IBM el novembre de 2001, desenvolupat en Java.
- Netbeans, projecte multiplataforma (Windows, Linux, Mac, Solaris) de codi obert, fundat per Sun Microsystems el juny de 2000, desenvolupat en Java.

- Anjuta DevStudio, per al GNU/Linux, creat per Naba Kumar el 1999.
- JBuilder, eina multiplataforma (Windows, Linux, Mac), propietat de l'empresa Borland, apareguda el 1995. La versió 2008 incorpora tres edicions (Enterprise –de pagament–, Professional –de pagament– i Turbo –gratuïta-).
- JDeveloper, eina multiplataforma (Windows, Linux, Mac) gratuïta, propietat de l'empresa Oracle, apareguda el 1998, inicialment basada en JBuilder però desenvolupada des de 2001 en Java.

A part dels que hem esmentat, existeixen molts altres IDE, molt coneguts, per a determinades plataformes, com per exemple:

- Visual Studio
- Dev-Pascal
- Dev-C++
- MonoDevelop

2.2 Instal·lació d'un entorn de desenvolupament. Eclipse

Cada programari i cada entorn de desenvolupament té unes característiques i unes funcionalitats específiques. Això quedarà reflectit també en la instal·lació i en la configuració del programari. En funció de la plataforma, entorn o sistema operatiu en què es vulgui instal·lar el programari, es farà servir un paquet d'instal·lació o un altre, i s'hauran de tenir en compte unes opcions o unes altres en la seva configuració.

Tot seguit es mostra com es du a terme la instal·lació d'una eina integrada de desenvolupament de programari, com és l'Eclipse. Però també podreu observar els procediments per instal·lar altres eines necessàries o recomanades per treballar amb el llenguatge de programació JAVA, com l'Apache Tomcat o la Màquina Virtual de Java.

Podeu trobar més documentació referent a l'entorn Eclipse als "Annexos" i "Activitats" dels materials Web d'aquest apartat.

Cal que tingueu presents els següents conceptes:

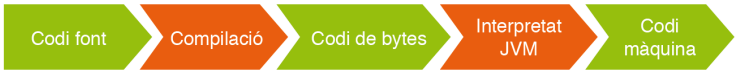
- **JVM** (*Java Virtual Machine*, màquina virtual de Java) s'encarrega d'interpretar el codi de bytes i generar el codi màquina de l'ordinador (o dispositiu) en el qual s'executa l'aplicació. Això significa que ens cal una JVM diferent per a cada entorn.
- **JRE** (*Java Runtime Environment*) és un conjunt d'utilitats de Java que inclou la JVM, llibreries i el conjunt de programari necessari per executar les aplicacions client de Java, així com el connector per tal que els navegadors d'internet puguin executar les *applets*.

Per ampliar el concepte de Màquina Virtual de Java podeu consultar l'apartat "Codi font, codi objecte i codi executable: màquines virtuals" de la unitat "Desenvolupament de programari".

- **JDK** (*Java Development Kit*, *kit* de desenvolupament de Java) és el conjunt d'eines per a desenvolupadors; conté, entre altres coses, el JRE i el conjunt d'eines necessàries per compilar el codi, empaquetar-lo, generar documentació...

A la figura 2.2 es pot observar de forma esquemàtica el procés de conversió del codi Java, des de la creació del seu codi font fins a l'obtenció del codi màquina.

FIGURA 2.2. Procés de conversió del codi



El procés d'instal·lació consisteix en els següents passos:

1. Descarregar, instal·lar i configurar el JDK.
2. Descarregar i instal·lar un servidor web o d'aplicacions.
3. Descarregar, instal·lar i configurar Eclipse.
4. Configurar JDK amb l'IDE d'Eclipse.
5. Configurar el servidor Apache Tomcat en Eclipse.
6. En cas de ser necessari, instal·lació de connectors.
7. En cas de ser necessari, instal·lació de nou programari, com per exemple WindowBuilder.

2.2.1 Instal·lació del 'Java Development Kit'

Per poder executar Eclipse, cal tenir instal·lat el JDK prèviament a l'ordinador. El podeu descarregar a la pàgina bit.ly/1iOZIrD.

Descarregar i instal·lar el JDK

Podem diferenciar entre:

- Java SE (Java Standard Edition): és la versió estàndard de la plataforma, essent aquesta plataforma la base per a tot entorn de desenvolupament en Java pel que fa a aplicacions client, d'escriptori o web.
- Java EE (Java Enterprise Edition): és la versió més gran de Java i s'utilitza en general per crear aplicacions grans de client/servidor i per a desenvolupament de WebServices.

En aquest curs s'utilitzaran les funcionalitats de Java SE.

El fitxer és diferent en funció del sistema operatiu on s'hàgi d'instal·lar. Així:

- Per als sistemes operatius Windows i Mac OS hi ha un fitxer instal·lable.
- Per als sistemes operatius GNU Linux que admeten paquets *.rpm* o *.deb* també hi ha disponibles paquets d'aquests tipus.
- Per a la resta de sistemes operatius GNU Linux hi ha un fitxer comprimit (acabat en *.tar.gz*).

En els dos primers casos, només cal seguir el procediment d'instal·lació habitual al sistema operatiu amb el qual es treballa.

Al darrer cas, però, cal descomprimir el fitxer i copiar-lo a la carpeta on el volem instal·lar. Normalment, tots els usuaris tindran permís de lectura i execució sobre aquesta carpeta.

A partir de la versió 11 del JDK Oracle distribueix el programari amb una **llicència** significativament més restrictiva que la de les versions anteriors. En concret, només pot utilitzar-se per a "desenvolupar, provar, fer prototipus i demostrar les vostres aplicacions". S'exclou explícitament tot ús "per a finalitats comercials, de producció o de negoci interns" diferent de l'esmentat abans.
En cas de necessitar-lo per algun d'aquests usos no permesos a la nova llicència, a més de les versions anteriors del JDK, existeixen versions de referència d'aquestes versions amb llicència "GNU General Public License version 2, with the Classpath Exception", que permet la majoria dels usos habituals. Aquestes versions estan enllaçades a la mateixa pàgina de descàrregues i, també, en l'adreça jdk.java.net.

2.2.2 Configurar les variable d'entorn "JAVA_HOME" i "PATH"

Consulteu a l'annex "Configuració de Java i exemple inicial" per veure pas a pas la configuració de la variable d'entorn PATH.

Una vegada descarregat i instal·lat el JDK, cal configurar algunes variables d'entorn:

- La variable `JAVA_HOME`: indica la carpeta on s'ha instal·lat el JDK. No és obligatori definir-la, però és molt convenient fer-ho, ja que molts programes cerquen en ella la ubicació del JDK. A més, facilita definir les dues varibles següents.
- La variable `PATH`. Ha d'apuntar al directori que conté l'executable de la màquina virtual. Sol ser la subcarpeta *bin* del directori on hem instal·lat el JDK.

Variable CLASSPATH

Una altra variable que té en compte el JDK és la variable `CLASSPATH`. Apunta a les carpetes on són les biblioteques pròpies de l'aplicació que es vol executar amb l'ordre *java*. És preferible, però, indicar la ubicació d'aquestes carpetes amb l'opció *-cp* de la mateixa ordre *java*, ja que cada aplicació pot tenir biblioteques diferents i les variables d'entorn afecten a tot el sistema.

Configurar la variable `PATH` és imprescindible perquè el sistema operatiu trobi les ordres del JDK i pugui executar-les.

2.2.3 Instal·lació del servidor web

Un servidor web és un programa que serveix per atendre i respondre a les diferents peticions dels navegadors, proporcionant els recursos que sol·licitin per mitjà del protocol HTTP o el protocol HTTPS (la versió xifrada i autenticada).

Podeu consultar l'annex "Descàrrega i instal·lació de l'Apache Tomcat" en la secció "Annexos".

Bàsicament, un servidor web consta d'un intèrpret HTTP que es manté a l'espera de peticions de clients i respon amb el contingut sol·licitat. El client, un cop rebut el codi, l'interpreta i el mostra en el navegador.

De fet, un servidor web sol executar de forma infinita el següent bucle:

- Espera peticions al port TCP (Protocol de Control de Transmissió) indicat (l'estàndard per defecte per a HTTP és el 80).
- Rep una petició.
- Cerca el recurs.
- Envia el recurs utilitzant la mateixa connexió mitjançant la qual va rebre petició.

El servidor web en què es basen els exemples d'aquesta unitat és Apache Tomcat.

2.2.4 Instal·lació d'Eclipse

Eclipse és una aplicació de codi obert desenvolupada actualment per la Fundació Eclipse, una organització independent, sense ànim de lucre, que fomenta una comunitat de codi obert i la utilització d'un conjunt de productes, serveis, capacitats i complements per a la divulgació de l'ús de codi obert en el desenvolupament d'aplicacions informàtiques. Eclipse va ser desenvolupat originalment per IBM com el successor de VisualAge.

Aquesta plataforma de programari és independent d'altres plataformes, entorns o sistemes operatius. Aquesta plataforma és utilitzada per desenvolupar entorns integrats de desenvolupament, com l'IDE de Java (Java Development Toolkit JDT).

Com que Eclipse està desenvolupat en Java, és necessari, per a la seva execució, que hi hagi un JRE (Java Runtime Environment) instal·lat prèviament en el sistema. Per saber si es disposa d'aquest JRE instal·lat, es pot fer la prova a la web oficial de Java, a l'apartat ¿Tengo Java?: bit.ly/2P2YOv1. A la figura 2.3 es pot veure un exemple d'aquesta prova. En el cas que no es tingui instal·lat, allà mateix se'n podrà descarregar la darrera versió i instal·lar-la sense cap dificultat.

FIGURA 2.3. Validació de la correcta instal·lació del JRE a l'ordinador

Si desenvoluparem amb Java, com és el nostre cas, cal tenir instal·lat el JDK (recordeu que és un superconjunt del JRE).

Les versions actuals de l'entorn Eclipse s'instal·len amb un instal·lador. Aquest, bàsicament s'encarrega de descomprimir, resoldre algunes dependències i crear els accessos directes.

Aquest instal·lador es pot obtenir baixant-lo directament del lloc web oficial del Projecte Eclipse www.eclipse.org. Podeu trobar les versions per als diferents sistemes operatius en aquest enllaç: bit.ly/2Ppj1OP. En aquesta pàgina, a més, trobareu les instruccions per utilitzar-lo. No són gens complexes.

Al cas de GNU Linux i MAC OS, l'arxiu és un fitxer comprimit. Caldrà, doncs, descomprimir-lo i, a continuació executar l'instal·lador. Aquest és al fitxer *eclipse-inst*, dins la carpeta *eclipse*, que és una subcarpeta del resultat de descomprimir el fitxer anterior.

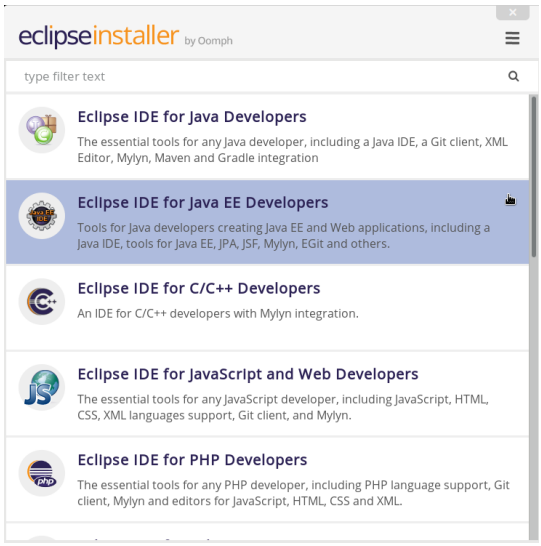
Si només l'usuari actual utilitzarà l'IDE, pot realitzar-se la instal·lació sense utilitzar privilegis d'administrador o *root* i seleccionar, per la instal·lació, una carpeta pròpia d'aquest usuari. Si es dessitgés compartir la instal·lació entre diferents usuaris, caldria indicar a l'instal·lador una carpeta sobre la qual tots aquests usuaris tinguessin permís de lectura i execució.

Un cop executem l'instal·lador, ens apareixerà una pantalla semblant a la de la figura 2.4.

Figura 2.4. Pantalla inicial de l'instal·lador d'Eclipse

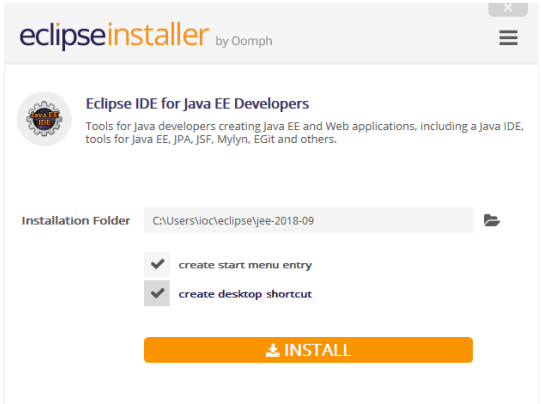
Com es veu a la figura 2.5, l'instal·lador demanarà quina versió volem instal·lar. La versió que utilitzarem és "Eclipse IDE for Java EE Developers".

Figura 2.5. Selecció de la versió d'Eclipse



A continuació, com es veu a la figura figura 2.6, demanarà la carpeta on farem la instal·lació.

Figura 2.6. Selecció de la carpeta on fer la instal·lació

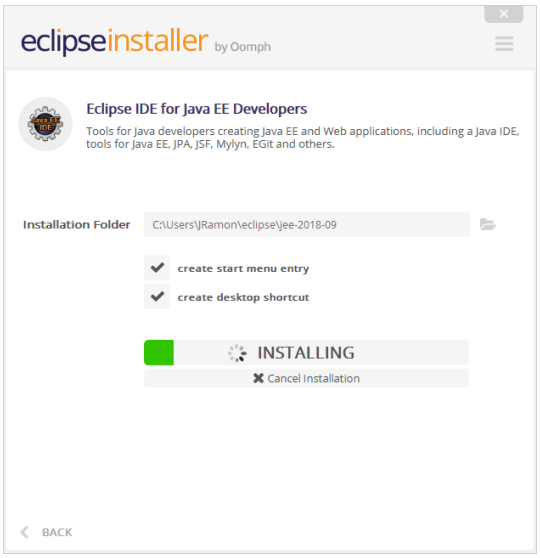


Per seleccionar la carpeta correcta, cal tenir en compte quins usuaris utilitzaran l'entorn. Tots ells han de tenir permís de lectura i execució sobre la carpeta en

qüestió. Un cop introduïda la carpeta, podem seleccionar les opcions dels menús i accessos directes o llençadores que desitgem i clicar el botó *INSTALL* perquè comenci la instal·lació.

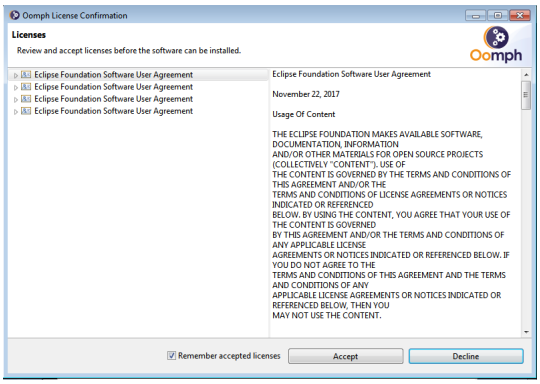
Durant la instal·lació ens apareixerà una pantalla de progrés com la que es veu a la figura 2.7.

FIGURA 2.7. Pantalla de progrés de la instal·lació



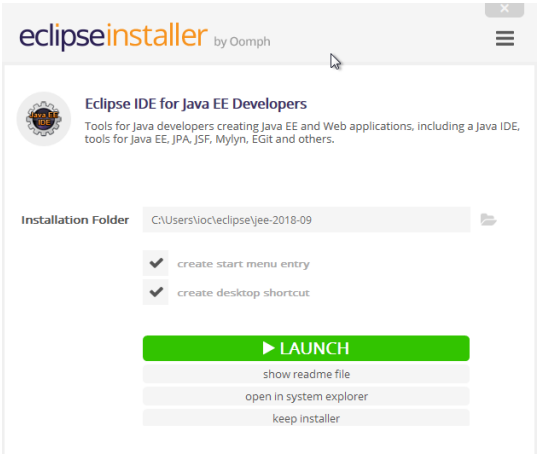
També se'ns demanarà que acceptem les llicències del programari que s'instal·larà, com mostra la figura 2.8.

FIGURA 2.8. Pantalla amb les llicències que cal acceptar



Un cop acabada la instal·lació se'ns mostra una pantalla com la de la figura 2.9 que ens convida a executar directament l'entorn.

FIGURA 2.9. Instal·lació finalitzada



Aquest primer cop podrem executar l'entorn Eclipse fent clic al botó *LAUNCH*. La resta de vegades, caldrà invocar-lo des dels accessos directes o llençadores, si s'han creat o, en cas contrari, invocant directament l'executable. Aquest s'anomena *eclipse* i el trobareu a en una subcarpeta de la carpeta d'instal·lació que s'anomena també *eclipse*. La ruta exacta pot variar d'una versió a una altra.

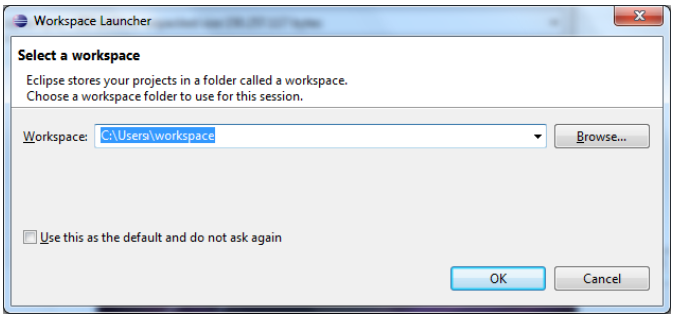
Si en un futur calgués desinstal·lar-lo, només s'hauria d'esborrar la carpeta on s'ha instal·lat ja que la instal·lació d'Eclipse no apareix al respositori de Linux ni al panell de control a Windows.

Quan executem l'entorn, ens apareixerà una pantalla com la que mostra la figura figura 2.10.

FIGURA 2.10. Pantalla d'inici d'Eclipse

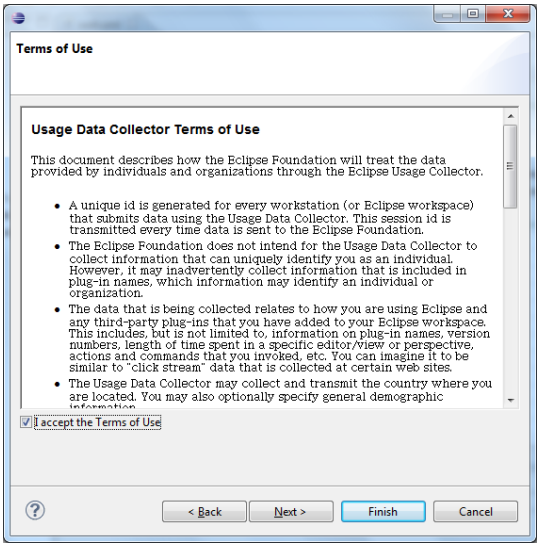
De seguida se'ns demanarà demanarà a quina carpeta caldrà ubicar l'espai de treball, com es veu a la figura 2.11. Podem demanar-li que el recordi per a la resta d'execucions activant l'opció "Use this as the default and do not ask again".

FIGURA 2.11. Escollir la ubicació de l'espai de treball



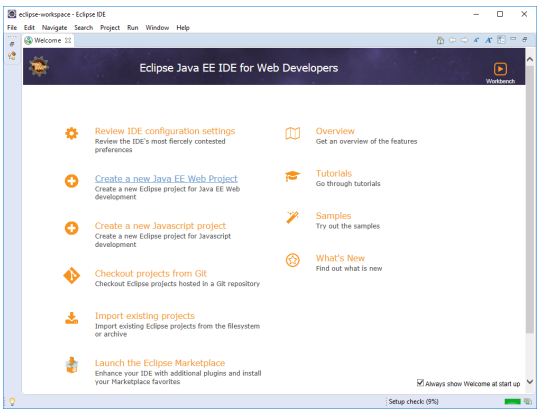
A partir d'aquest moment, ja es podrà utilitzar l'IDE Eclipse, havent d'acceptar en la primera execució els termes d'ús establerts (figura 2.12).

FIGURA 2.12. Acceptació dels termes d'ús de l'aplicació Eclipse



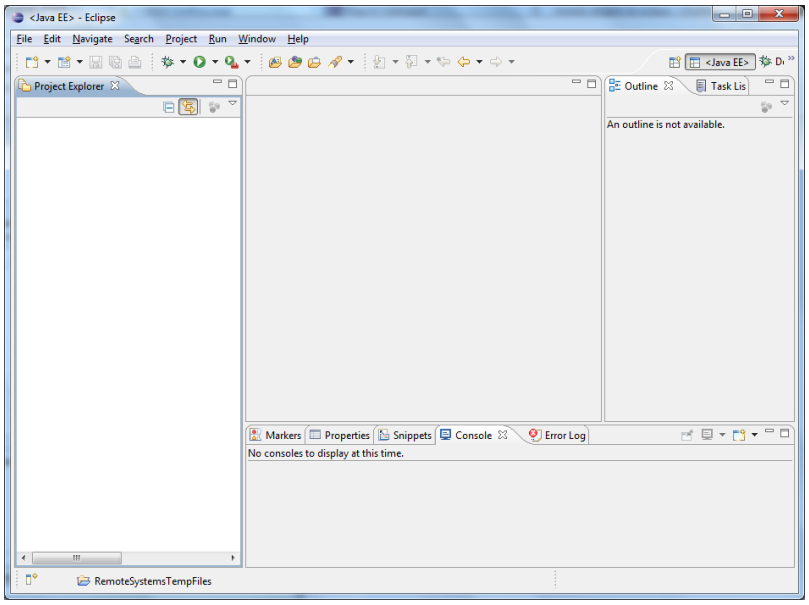
La primera vegada que l'executem, es mostrarà la pestanya de benvinguda, com es veu a la figura 2.13. Podem demanar que no ens la mostri més desactivant l'opció "Always show Welcome at start up".

FIGURA 2.13. Entorn de treball d'Eclipse



Un cop tancada aquesta pestanya, l'entorn de treball serà similar al mostrat per la figura 2.14.

FIGURA 2.14. Entorn de treball d'Eclipse

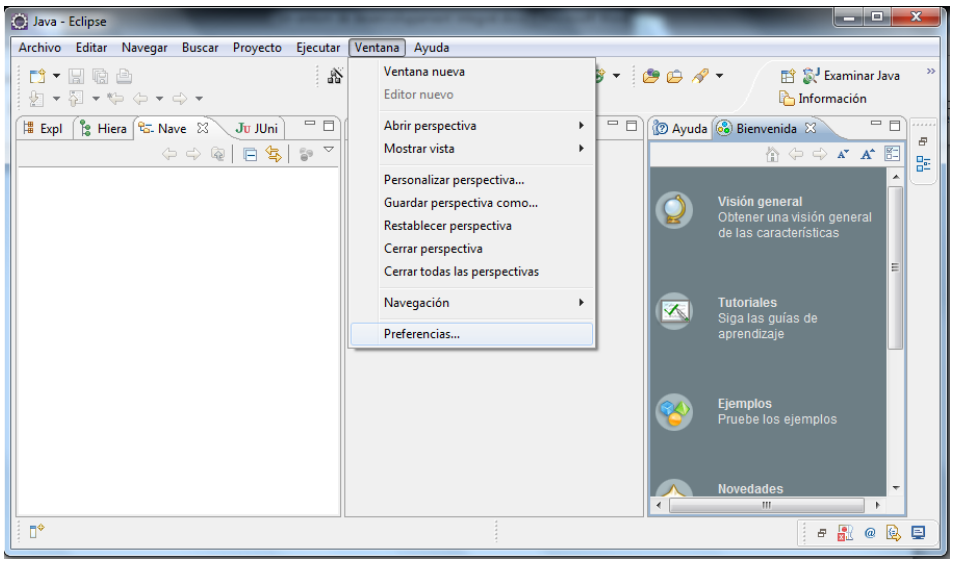


2.2.5 Configuració de JDK amb l'IDE Eclipse

Arribats a aquest punt, es parametritzarà l'entorn d'Eclipse amb el JDK instal·lat. Es pot observar aquest procediment a les figures següents.

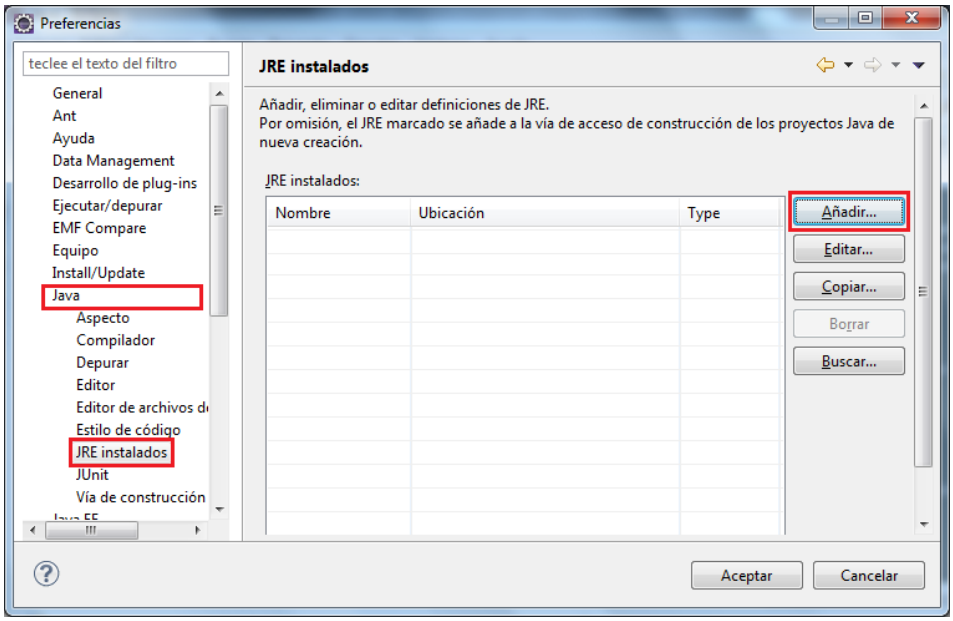
A la figura 2.15 es mostra l'opció de menú que cal seleccionar.

FIGURA 2.15. Parametrització de l'entorn d'Eclipse



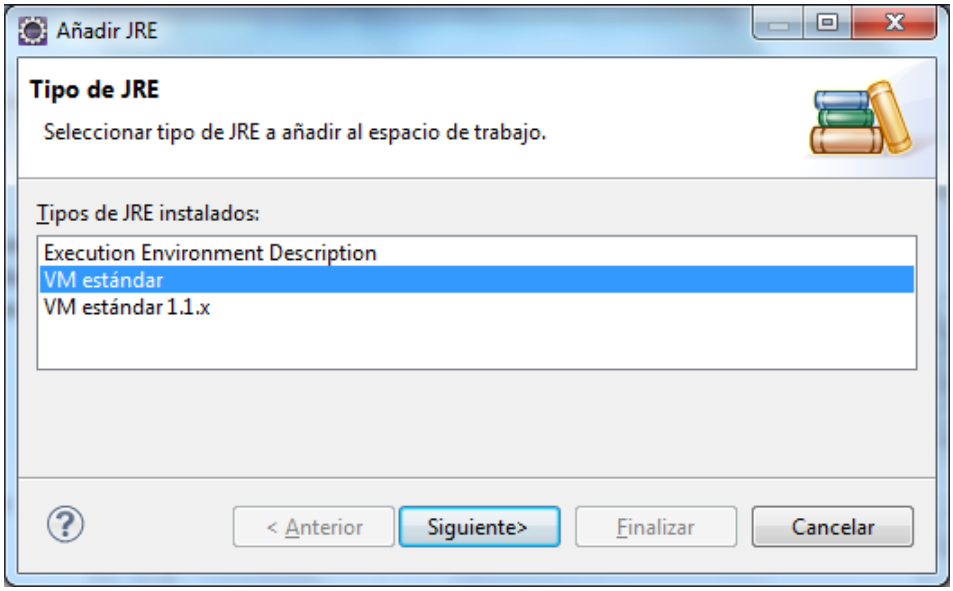
A la figura 2.16 s'arriba a l'apartat "Preferències", on es mostren els JRE instal·lats. Per defecte, no hi ha cap JRE instal·lat, amb la qual cosa s'haurà d'afegir el que s'hagi instal·lat.

Figura 2.16. Especificació del JRE o JDK instal·lat



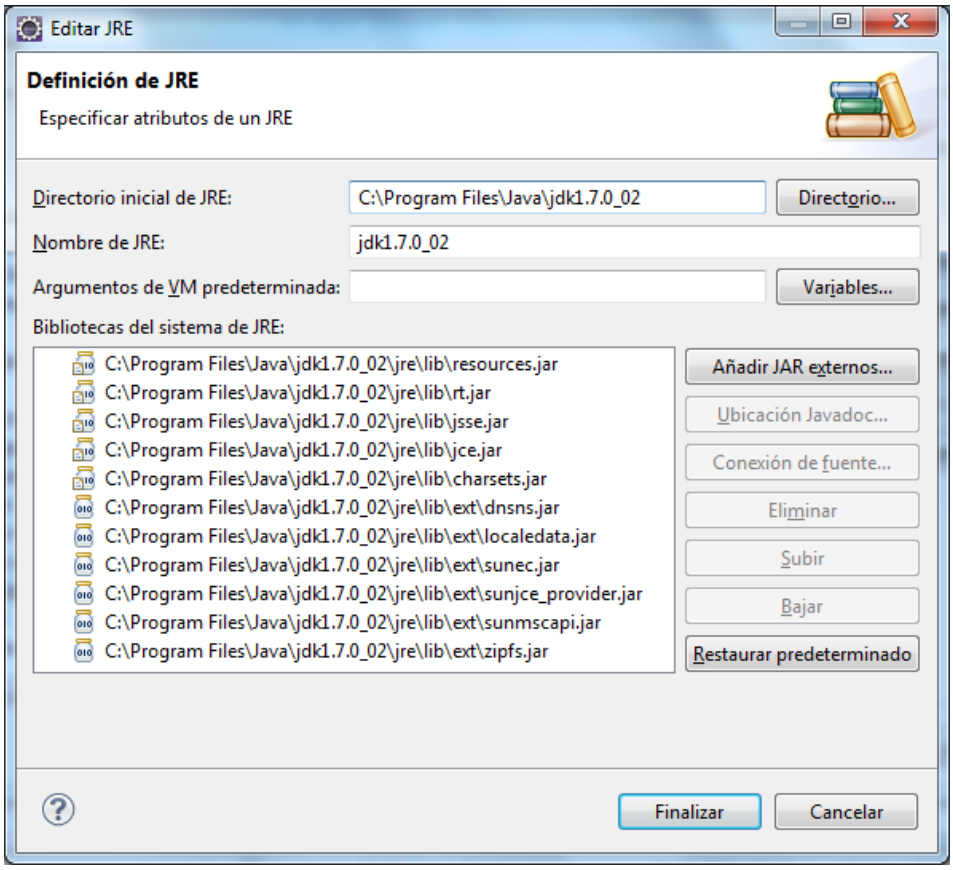
A la figura 2.17 es mostra el diàleg per afegir el JRE seleccionat.

Figura 2.17. Especificació del JRE o JDK instal·lat



Tot seguit caldrà especificar la ruta on s'ha instal·lat el JDK, abans de finalitzar la parametrització (figura 2.18).

FIGURA 2.18. Especificació del JRE o JDK instal·lat



2.2.6 Configuració del servidor web amb l'IDE Eclipse

A la figura 2.19 i figura 2.20 es mostra com seleccionar la pestanya Servers de l'IDE per fer-la visible, si no ho estava prèviament.

FIGURA 2.19. Mostrar la vista de servidors

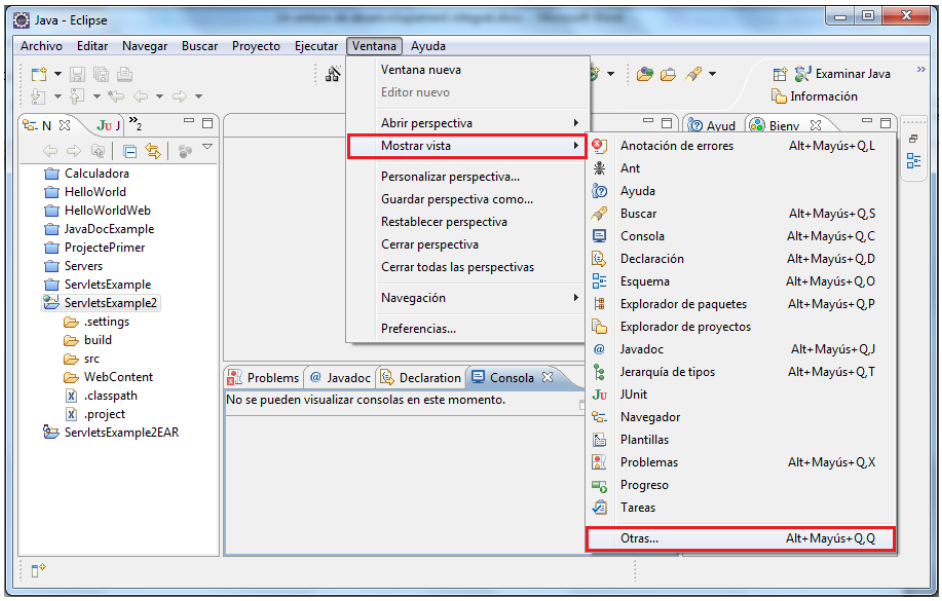
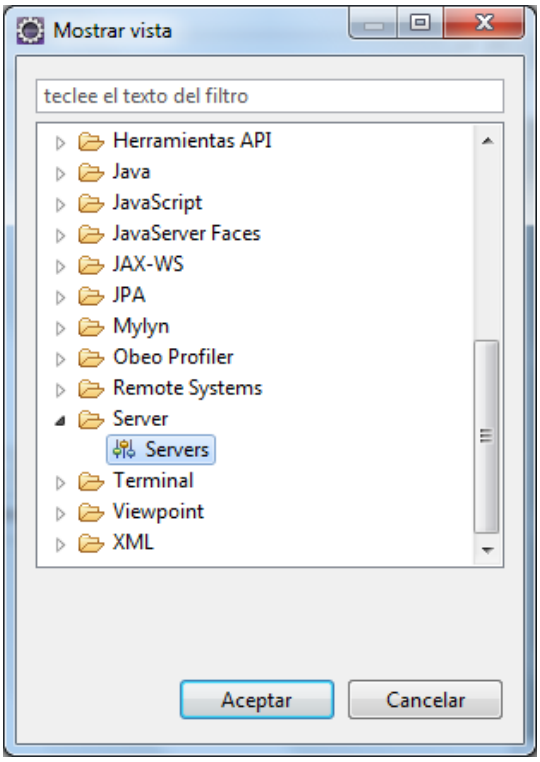
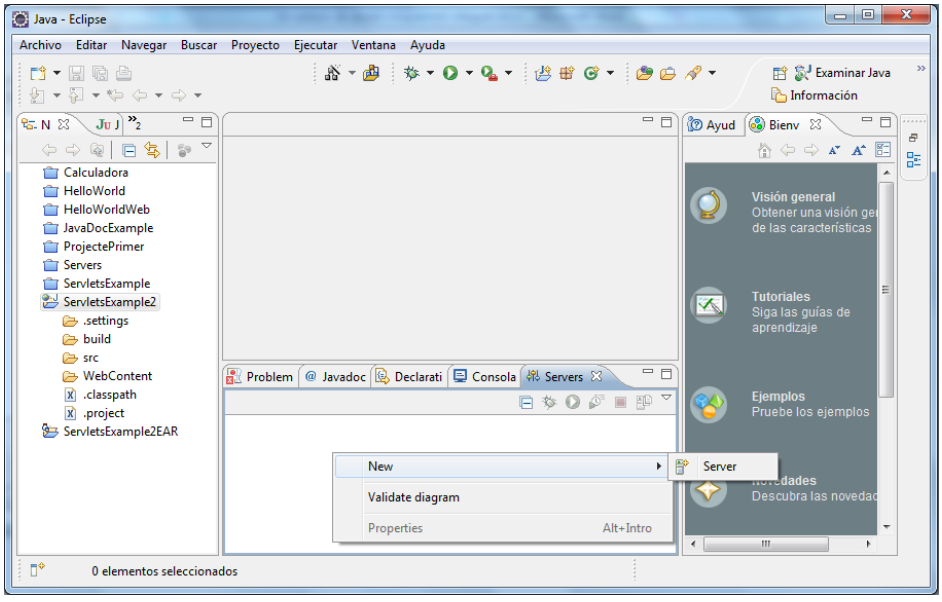


FIGURA 2.20. Selecció de la vista a visualitzar: servidors



En la part inferior de la figura 2.21 es visualitza la vista de servidors buida; s'hi haurà d'indicar el servidor d'aplicacions.

FIGURA 2.21. Afegir un nou servidor



A la figura 2.22 i figura 2.23 queda explicat com indicar quin és el servidor d'aplicacions.

Cal tenir cura de triar la versió que s'ha instal·lat (les captures s'han realitzat amb la versió 7, però la vostra pot ser una altra).

FIGURA 2.22. Selecció del servidor

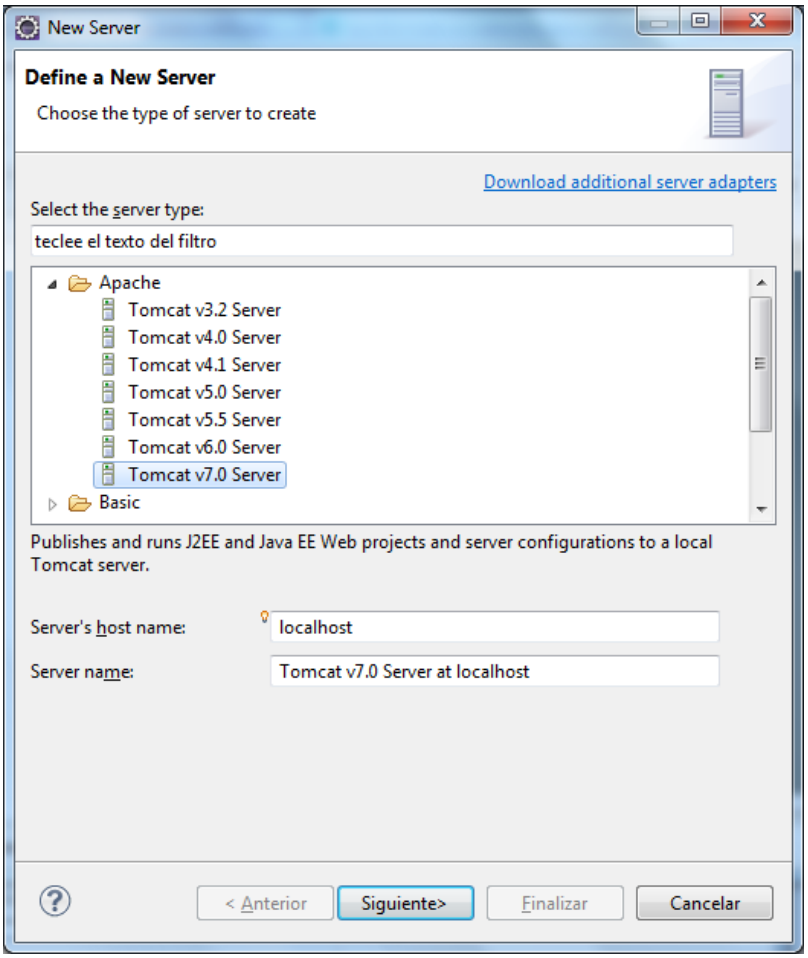
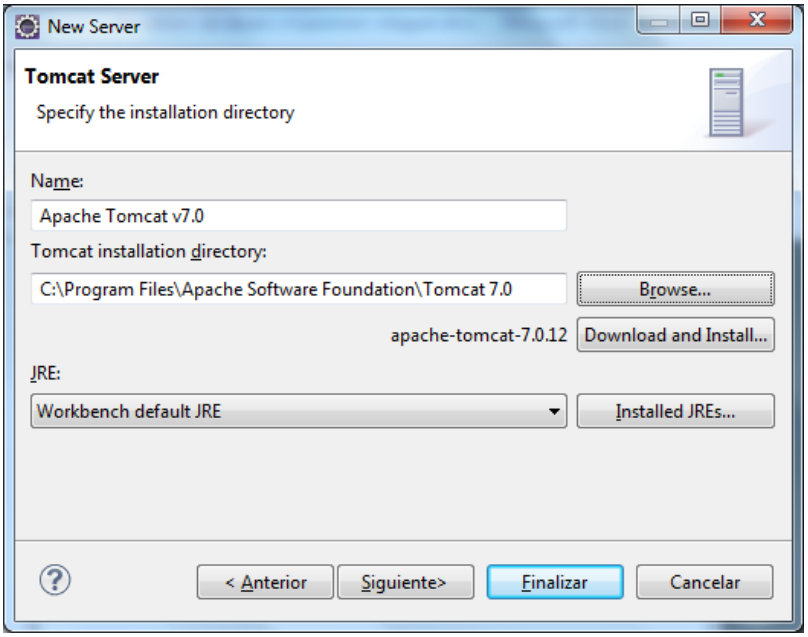


FIGURA 2.23. Especificació de la ruta del servidor



Podeu consultar l'annex "Instal·lació de connectors" en els materials Web.

2.2.7 Instal·lació de connectors

Una vegada instal·lat l'Eclipse, caldrà afegir aquells complements que siguin necessaris per a una correcta execució. La descàrrega bàsica de l'entorn Eclipse inclou algunes de les funcionalitats més bàsiques, però sempre és desitjable obtenir alguna funcionalitat extra. Aquestes funcionalitats es troben en els connectors, que caldrà localitzar, descarregar i instal·lar.

Connectors són un conjunt de components de programari que afegiran funcionalitats noves a les aplicacions instal·lades.

En l'apartat "Community" del lloc web oficial d'Eclipse es poden trobar enllaços a centenars de connectors. Aquests poden haver estat desenvolupats per programadors de la mateixa Fundació o haver-ho estat per usuaris de l'aplicació que volen compartir amb altres usuaris de forma altruista complements que ells mateixos han desenvolupat per solucionar alguna mancança que han trobat.

És molt important una **selecció acurada dels connectors**. Existeixen molts connectors que es poden instal·lar, però a mesura que es vagin afegint, això influirà en el rendiment de l'IDE Eclipse, especialment, en el temps d'arrencada inicial de l'aplicació.

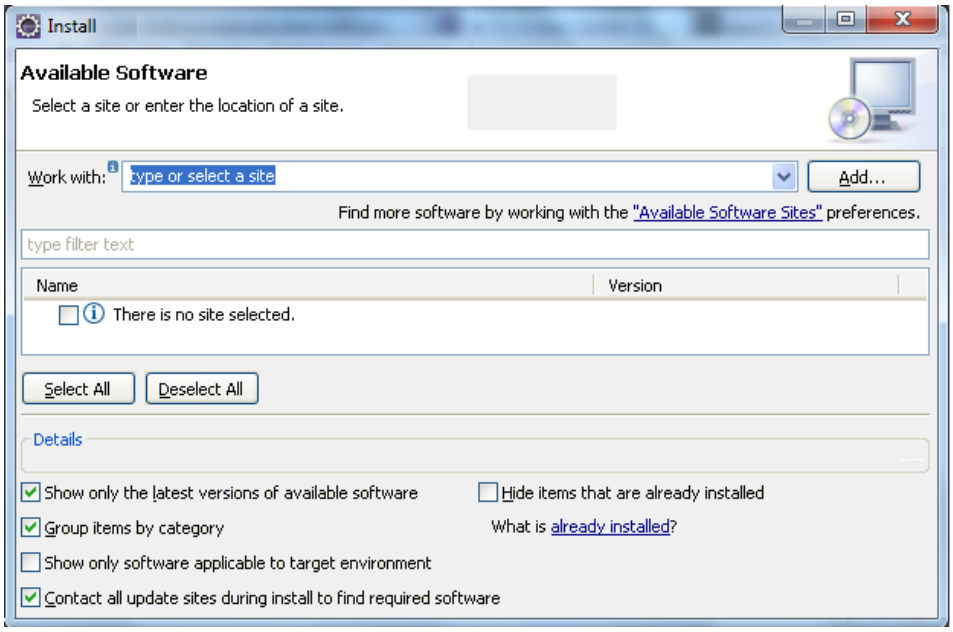
2.2.8 Instal·lació de components per al desenvolupament d'aplicacions GUI

El component que emprarem en aquesta unitat per desenvolupar aplicacions GUI és **WindowsBuilder Pro**.

Instal·lació del component

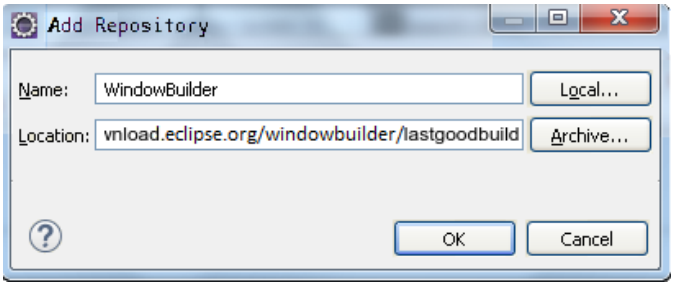
Per tal d'instal·lar-lo s'haurà d'accedir a l'opció del menú *Help* i seleccionar l'opció *Install new software*. Ens apareixerà la finestra de la figura 2.24.

Figura 2.24. Instal·lació de components



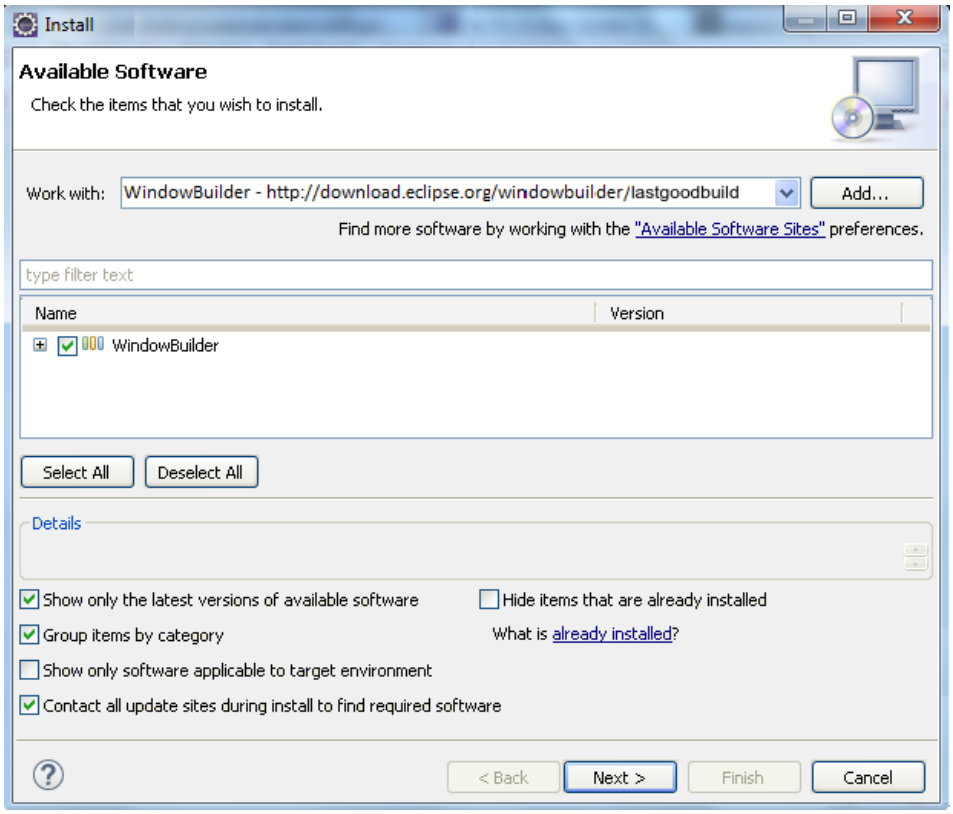
En ella caldrà fer clic al botó *Add...*. Se'ns obre una finestra com la de la figura 2.25. En ella posarem *WindowBuilder* a l'apartat *Name* i, a l'apartat *Location*, l'adreça bit.ly/2X4c31e, que és l'adreça des d'on s'instal·la el connector.

Figura 2.25. Afegir un repositori



Ens apareix una finestra com la de la figura 2.26. En ella cal fer clic al botó *Select All* i, a continuació, novament clic al botó *Next>*. Després, només caldrà seguir les indicacions de l'assistent i, quan aquest ens ho demani, reiniciar l'entorn Eclipse.

Figura 2.26. Instal·lació de WindowsBuilder



Obtenció del Run-time de SWT

Una vegada descarregat el Run-time de SWT (Standard Widget Toolkit), es descomprimeix i s'ubica en un determinat directori.

Podeu descarregar el Run-time de SWT a la pàgina www.eclipse.org/swt.

S'especifica la seva URL com a variable d'entorn.

```
1 CLASSPATH = URL/swt.jar
```

2.3 Ús bàsic d'un entorn de desenvolupament. Eclipse

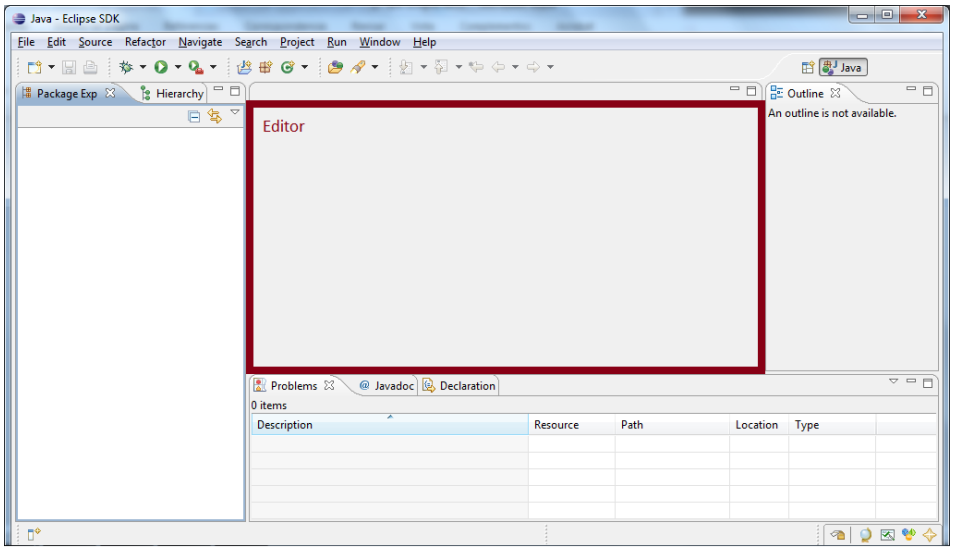
Una vegada instal·lat el programari, cal fer una revisió del seu entorn de treball i revisar els espais que ofereix i les funcions de cadascun d'ells. Dintre dels apartats es poden identificar, entre d'altres, els següents:

- Editor.
- Vistes.
- Barra d'eines principal.
- Barres de perspectives.

2.3.1 Editors

La finestra principal (la més gran a la figura 2.27) es diu Editor. Els editors són el lloc on es podrà desenvolupar el codi de programació, podent-lo afegir, modificar o esborrar. És possible tenir diversos editors oberts al mateix temps, apilats un sobre l'altre. A la part superior de la finestra d'editors, es mostraran solapes que permeten accedir a cada un dels editors oberts (o bé tancar-los).

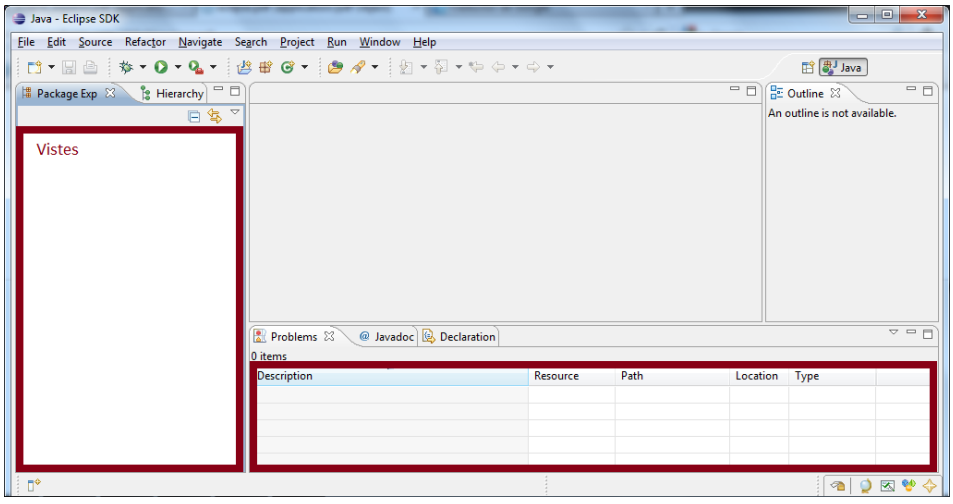
FIGURA 2.27. Identificació de l'espai Editor



2.3.2 Vistes

Les Vistes són un altre tipus de finestres, anomenades finestres secundàries. Serveixen per a qualsevol cosa, des de navegar per un arbre de directoris fins a mostrar el contingut d'una consulta SQL. Són finestres auxiliars destinades a mostrar informació i resultats de compilacions o logs, requerir dades...

FIGURA 2.28. Identificació dels espais Vistes

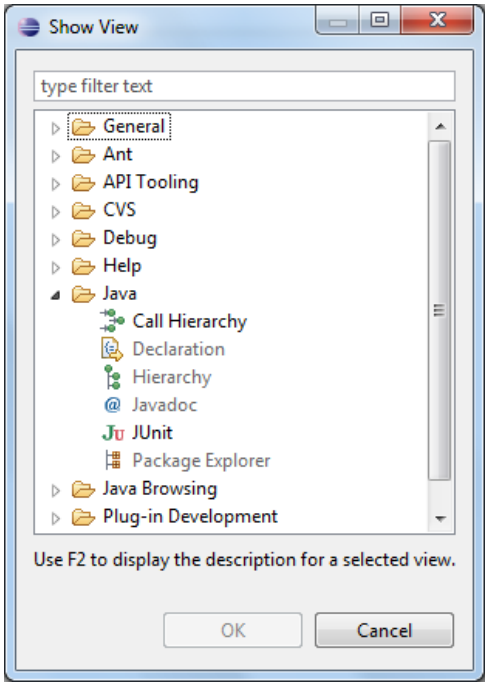


A la figura 2.28 es mostren dos tipus de vistes:

- La vista vertical de l'esquerra mostra la jerarquia dels projectes (quan n'hi hagi).
- La vista horitzontal inferior mostra un petit històric de tasques pendents que pot introduir l'usuari, de forma directa, o Eclipse, en funció de determinats esdeveniments.

Per seleccionar quines són les Vistes que s'han de mostrar, s'utilitza l'opció *Show View* al menú *Window* (vegeu la figura 2.29).

Figura 2.29. 'Show View'



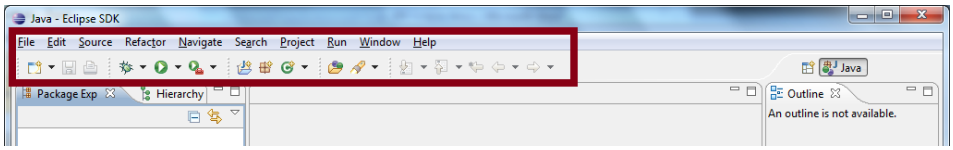
2.3.3 Barres d'eines

Un altre dels components de l'entorn són les Barres d'eines. N'hi ha dues: la Barra d'eines principal i la Barra de perspectives.

Barra d'eines principal

La Barra d'eines principal conté accessos directes a les operacions més usuals. És una barra de menú que tindrà l'accés a totes les opcions del programari (vegeu la figura 2.30).

FIGURA 2.30. Barra d'eines principal



Barra de perspectives

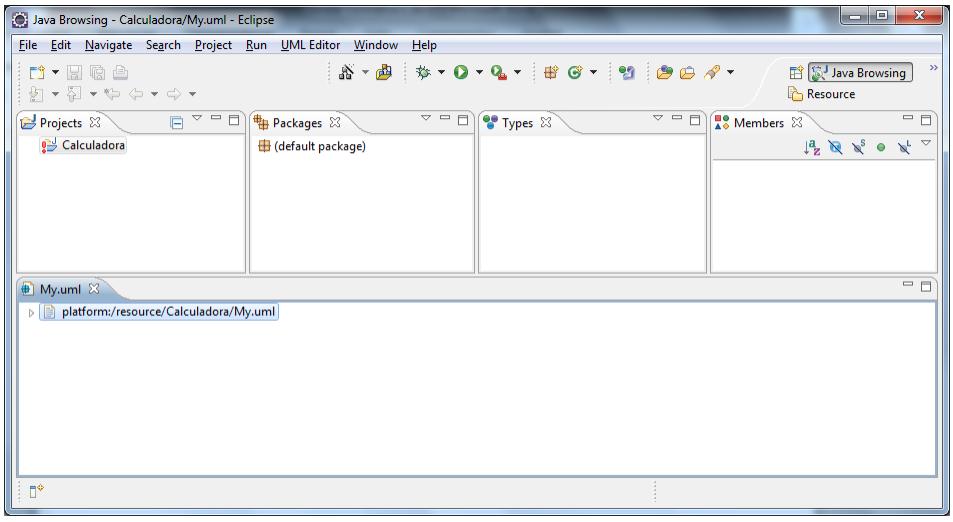
> Una **perspectiva** és un conjunt de finestres (editors i vistes) relacionades entre si.

La Barra de perspectives conté accessos directes a les perspectives que s'estan utilitzant en el projecte. Per exemple, hi ha una perspectiva Java que facilita el desenvolupament d'aplicacions Java i que inclou, a més de l'Editor, Vistes per navegar per les classes, els paquets...

Es pot seleccionar les perspectives actives -les que es mostren a la Barra de perspectives- utilitzant l'opció *Open Perspective* del menú *Window*. Des d'aquest mateix menú també és possible definir perspectives personalitzades. Es pot observar a la figura 2.31.

A més de la Barra d'eines principal, cada vista pot tenir la seva pròpia barra d'eines.

FIGURA 2.31. Barres de perspectives



2.4 Edició de programes

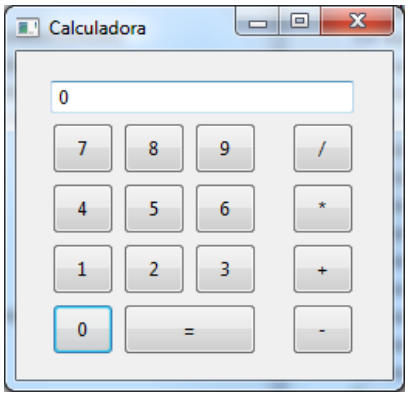
L'edició de programes és la part més important de les que es duran a terme amb Eclipse. Tant la creació com la modificació i el manteniment d'aplicacions necessitaran d'un domini de l'eina en les seves funcionalitats. Una vegada vist com instal·lar l'Eclipse i el seu entorn de treball, el millor per mostrar com editar, crear i mantenir programes és mostrar-ho amb un exemple.

Amb la versió estàndard de l'entorn integrat de desenvolupament Eclipse es distribueix el connector necessari per programar en llenguatge Java, el nom del qual és JDT. És important tenir això en compte, ja que Eclipse és un IDE que no està orientat específicament a cap llenguatge de programació en concret. Si es volgués aprofitar l'eina per desenvolupar en un altre llenguatge de programació, caldria descarregar el connector corresponent per tal que li donés suport.

2.4.1 Exemple d'utilització d'Eclipse: "projecte Calculadora"

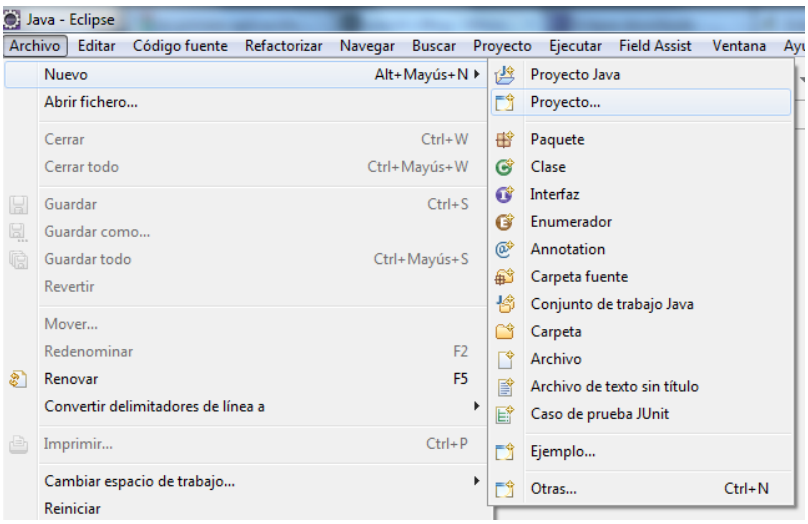
Es tracta d'un exemple senzill d'implementar i, per això precisament, és el més adient per poder entendre com utilitzar de forma bàsica aquesta eina integrada de desenvolupament. A la figura 2.32 es pot observar un exemple de quina seria la interfície gràfica que es vol implementar.

Figura 2.32. Calculadora



Per poder dur a terme un programa nou en Eclipse serà necessari crear un projecte. Per crear un projecte, com es pot veure a la figura 2.33, caldrà escollir l'opció del menú *Arxiu* / *Nou* / *Projecte*, des de la Barra d'eines principal o des de la vista Navigator (obrint el menú contextual amb el botó dret del ratolí i la opció *Nou* / *Project*).

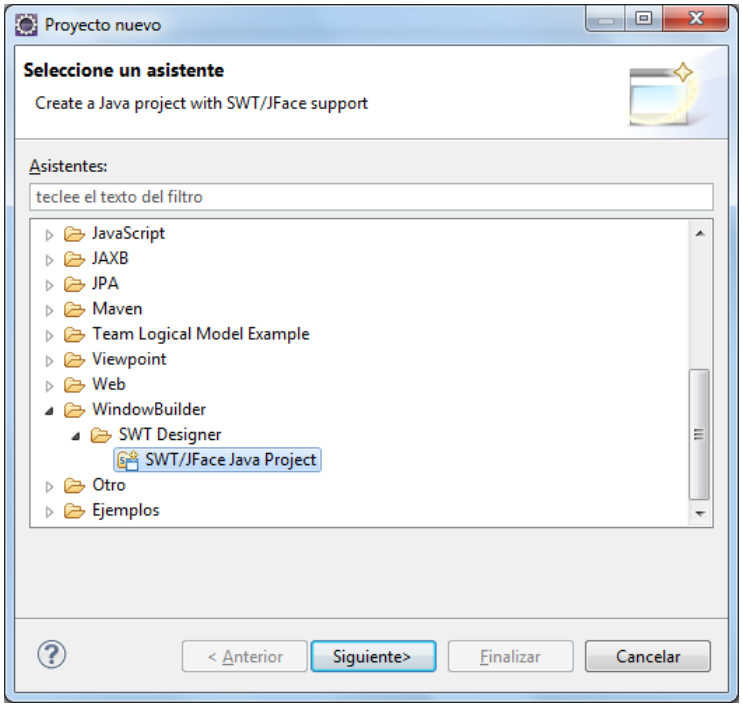
Figura 2.33. Nou projecte



Un projecte agrupa un conjunt de recursos relacionats entre si (codi font, diagrames de classes, documentació...).

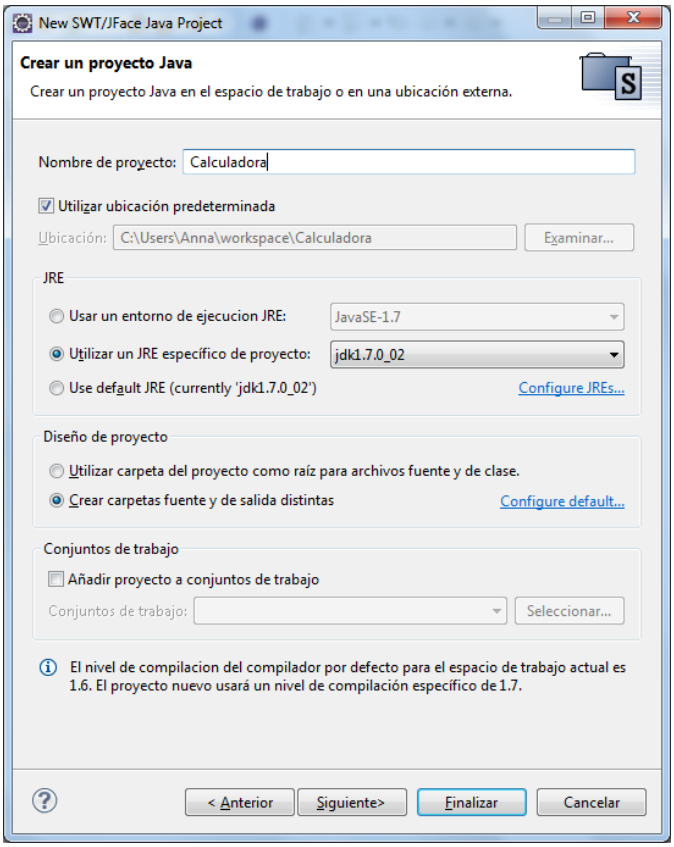
A la figura 2.34 es pot observar la finestra que es mostra al crear un nou projecte.

Figura 2.34. Projecte de tipus WindowBuilder



Qualsevol de les opcions mostrarà l'assistent de creació de projectes que es pot veure a la figura 2.35 i a la figura 2.36.

Figura 2.35. Assistent de creació del projecte

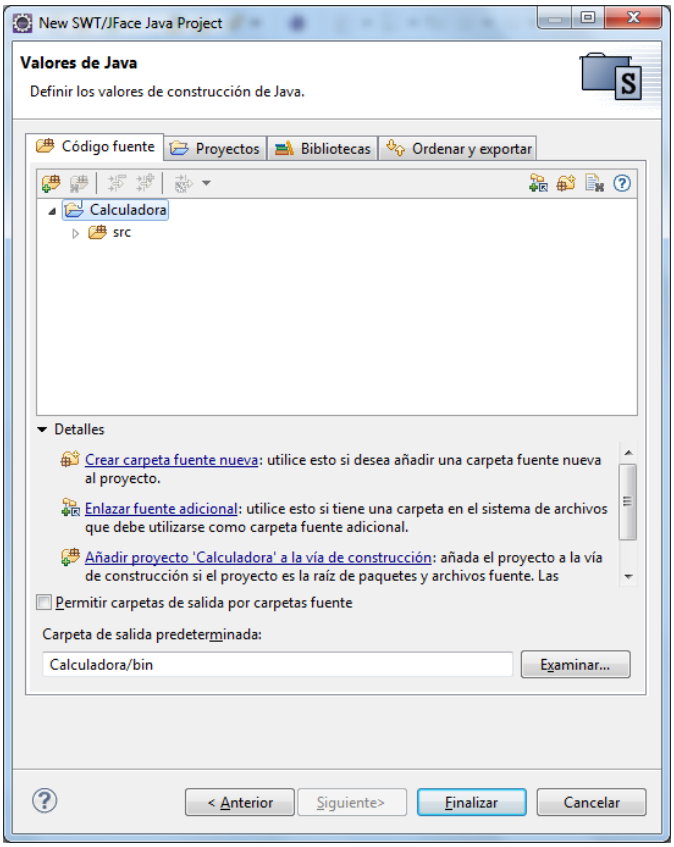


Per iniciar un nou projecte de tipus Window s'ha de seleccionar l'opció *Window-Builder / SWT Designer / SWT / JFace Java Project*.

Aquesta opció demanarà que s'indiqui un nom i una ubicació per al nou projecte. Com es pot observar a la figura 2.36, es podran parametritzar algunes característiques del nou projecte, com poden ser, entre altres:

- Seleccionar la carpeta on s'emmagatzemaran les llibreries (JAR) que necessita el projecte.
- Definir variables d'entorn.
- Indicar si hi haurà dependències del projecte que s'està creant en relació amb projectes ja duts a terme (existents en el mateix espai de treball).
- Crear una carpeta per emmagatzemar el codi de programació i una de diferent per emmagatzemar les classes una vegada ja compilades.

Figura 2.36. Assistent de creació del projecte

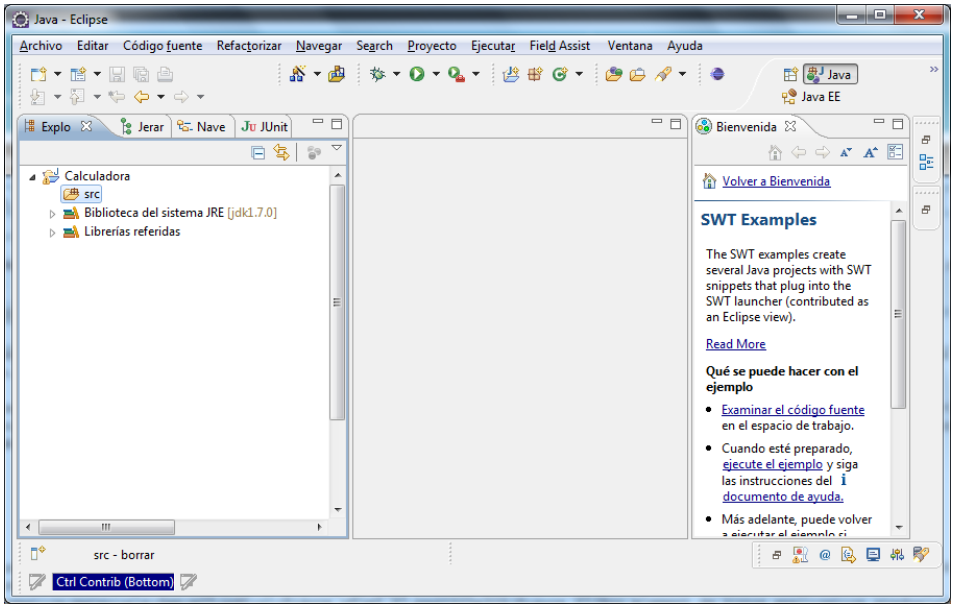


Dins d'aquestes opcions cal fer alguns comentaris. Cal recomanar sempre definir una carpeta per contenir el codi font, amb un nom clar i indicatiu del fet que hi haurà l'origen del projecte, i una altra amb un nom per indicar que contindrà els arxius binaris, on es deixaran els .class generats. També cal indicar que a l'opció de Llibreries es poden afegir tots els JAR que siguin necessaris.

Totes aquestes configuracions poden modificar-se en qualsevol moment a través del menú contextual de la vista Navegador, en l'opció *Propietats*.

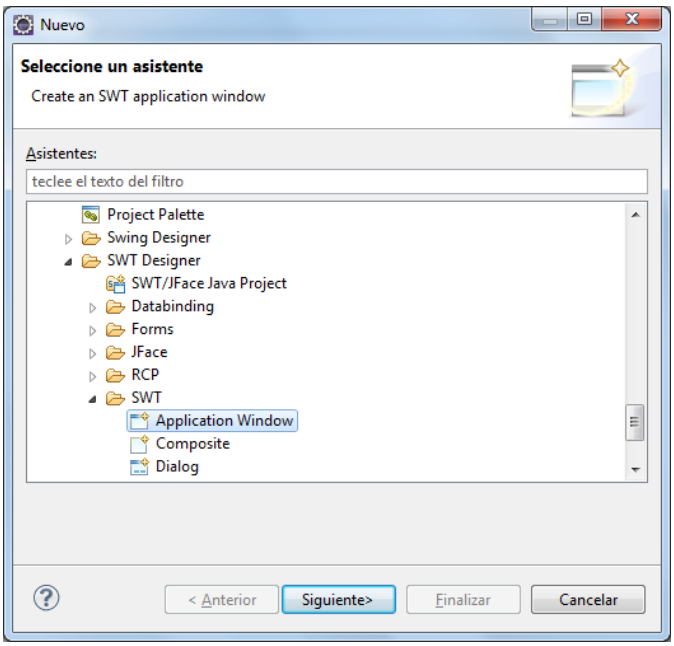
A la figura 2.37 es pot observar, en crear el projecte, com s'obrirà de forma automàtica la perspectiva Java, que és la col·lecció de Vistes que defineix el connector JDT. Si la perspectiva Java està activa, s'afegeixen a la Barra d'eines principal alguns botons extra que permeten accedir amb rapidesa a les funcions més usuals (execució del codi, depuració del codi, creació de noves classes...)

FIGURA 2.37. IDE una vegada creat el projecte



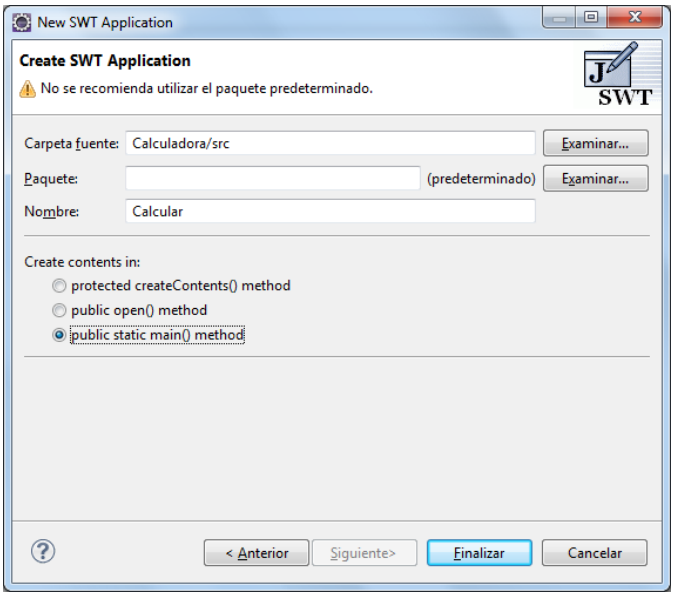
Una vegada creat el projecte, s'haurà de crear una nova classe on es dissenyarà la calculadora. La creació de la classe es du a terme executant, en la Barra d'eines, la ruta *Arxiu* / *Nou* / *Altres*, on s'inicia un assistent que permetrà especificar el tipus d'element a crear. En el cas de la Calculadora s'haurà de seleccionar *WindowBuilder* / *SWT Designer* / *SWT* / *Application Window* (vegeu la figura 2.38).

FIGURA 2.38. Nou element de tipus Application Window



Tal com es pot observar en la figura 2.39, l'assistent permet especificar la ruta on s'emmagatzemarà la classe, el paquet, el nom de la classe i el tipus de mètode que es crearà. En el cas de l'exemple se seleccionarà l'opció *public static main()method*.

FIGURA 2.39. Assistent del nou element

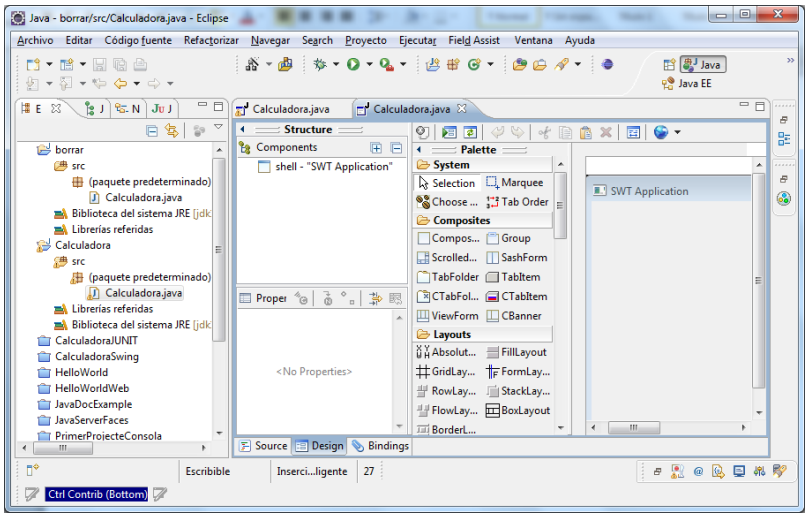


Automàticament, l'assistent ha creat el següent codi:

```
1  import org.eclipse.swt.widgets.Display;
2  import org.eclipse.swt.widgets.Shell;
3
4  public class Calculadora {
5      /**
6      * Launch the application.
7      * @param args
8      */
9      public static void main(String[] args) {
10         Display display = Display.getDefault();
11         Shell shell = new Shell();
12         shell.setSize(450, 300);
13         shell.setText("SWT Application");
14
15         shell.open();
16         shell.layout();
17         while (!shell.isDisposed()) {
18             if (!display.readAndDispatch()) {
19                 display.sleep();
20             }
21         }
22     }
23 }
```

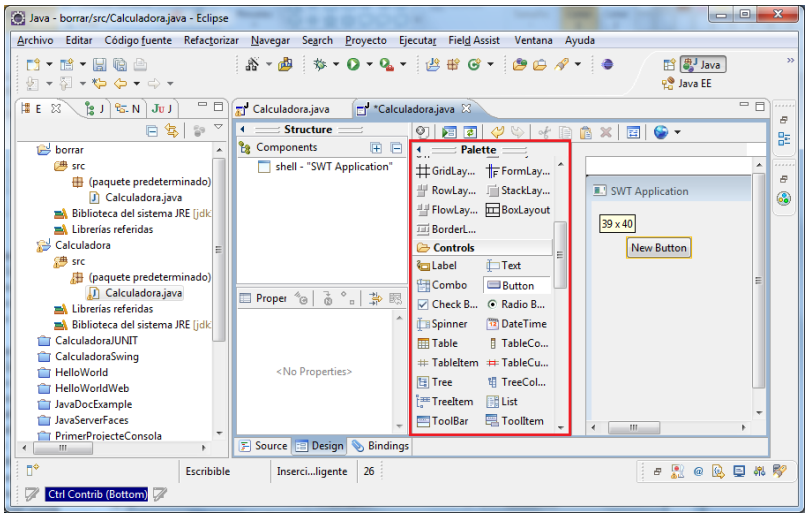
Arribats a aquest punt és quan es podrà començar a dissenyar gràficament el formulari de la calculadora fent ús de la vista Design (vegeu la figura 2.40).

FIGURA 2.40. Vista de disseny



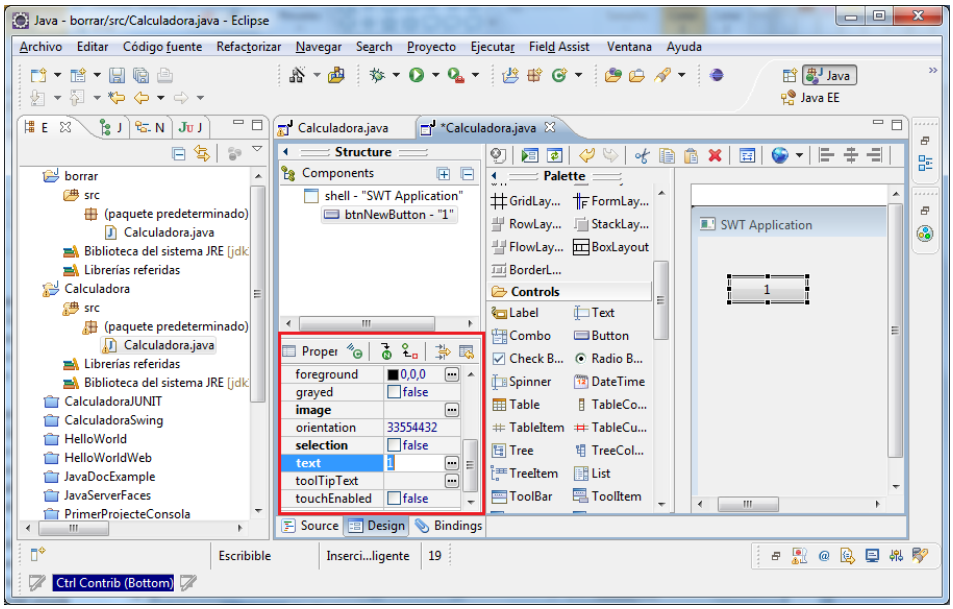
Es visualitza un conjunt de vistes que permet seleccionar un control, per exemple un botó, i desplaçar-lo cap al formulari (figura 2.41).

FIGURA 2.41. Afegir un botó



En la vista de Propietats es podrà modificar les propietats del control seleccionat (figura 2.42).

FIGURA 2.42. Especificar les propietats del botó



El codi que es genera automàticament en afegir el botó corresponent al número 1 de la calculadora és:

```
1 Button btnNewButton = new Button(shell, SWT.NONE);
2         btnNewButton.setBounds(30, 38, 47, 25);
3         btnNewButton.setText("1");
```

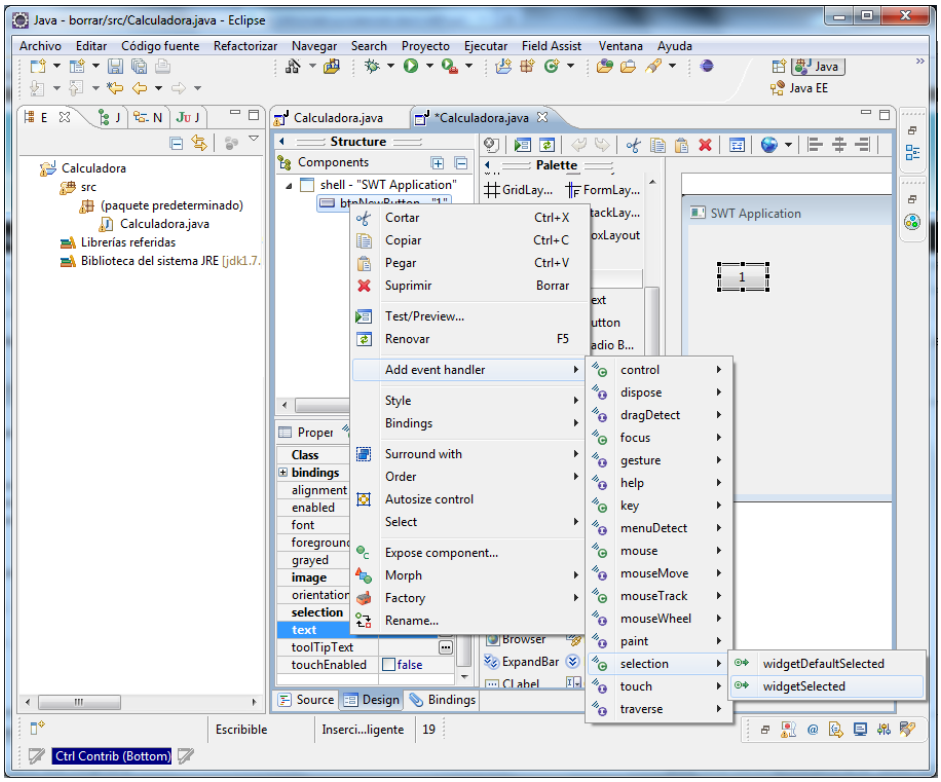
Es repeteix el procés per als 10 números (0,1,2,3,4,5,6,7,8,9), els operands (+, -, /, *) i l'igual (=).

El control utilitzat per mostrar el resultat de l'operació és un text.

El següent pas és activar l'esdeveniment per a cada un dels botons creats per tal de poder especificar l'acció a dur a terme.

En la vista de Components se selecciona el control i amb el botó dret se selecciona *Afegeix controlar d'esdeveniments* / *Seleccionar* / widget *seleccionat*. Es pot observar a la figura 2.43.

FIGURA 2.43. Esdeveniment en clicar el botó.



Automàticament, el codi corresponent al botó queda actualitzat.

```
1  Button btnNewButton = new Button(shell, SWT.NONE);
2       btnNewButton.addSelectionListener(new SelectionAdapter() {
3            @Override
4            public void widgetSelected(SelectionEvent e) {
5            }
6       });
7       btnNewButton.setBounds(30, 38, 47, 25);
8       btnNewButton.setText("1");
```

Es repeteix l'acció per a cadascun dels botons.

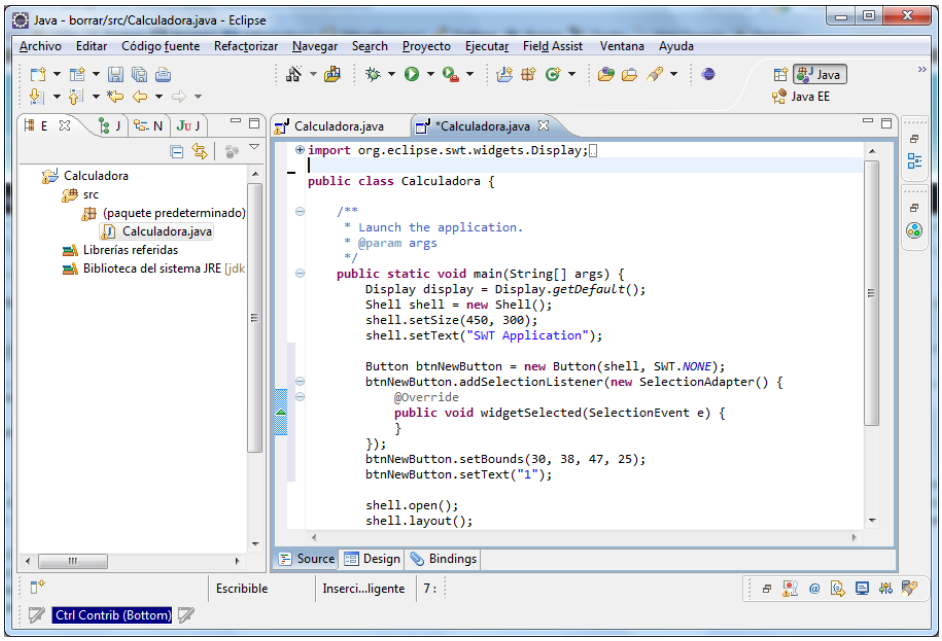
Syntax highlighting és el reconeixement sintàctic de expressions reservades del llenguatge.

Una vegada s'ha generat automàticament el codi dels controls, caldrà començar a programar el codi Java que es vol desenvolupar. Tal com es pot veure en el codi generat de la classe `Calculadora`, algunes paraules mostren un color diferent a la resta. Aquesta forma de marcar amb colors les paraules és a causa que els Editors Java inclouen la capacitat per efectuar el ressaltat de la sintaxi (*syntax highlighting*).

La codificació en colors (figura 2.44) dels termes és:

- Les paraules reservades del llenguatge apareixeran escrites en color bordeus i en negreta.
- Els comentaris apareixeran en verd.
- Els comentaris de documentació (Javadoc) apareixeran en blau.

FIGURA 2.44. Codificació de colors



Es crearan els mètodes necessaris per tal de codificar la funcionalitat de la calculadora. Per exemple, es crea un mètode que executa l'operació de suma, resta, multiplicació o divisió entre dos nombres.

```
int executarOperacio() {
     int resultat = 0;
     int numeroVisualitzar = getResultatInt();

     //Operació: divisió
     if (última_operació.equals("/"))
          resultat = ultim_valor / numeroVisualitzar;

     //Operació: multiplicació
     if (ultima_operacio.equals("*"))
          resultat = ultim_valor * numeroVisualitzar;

     //Operació: resta
     if (ultima_operacio.equals("-"))
          resultat = ultim_valor - numeroVisualitzar;

     //Operació: suma
     if (ultima_operacio.equals("+"))
          resultat = ultim_valor + numeroVisualitzar;

     return resultat;
}
```

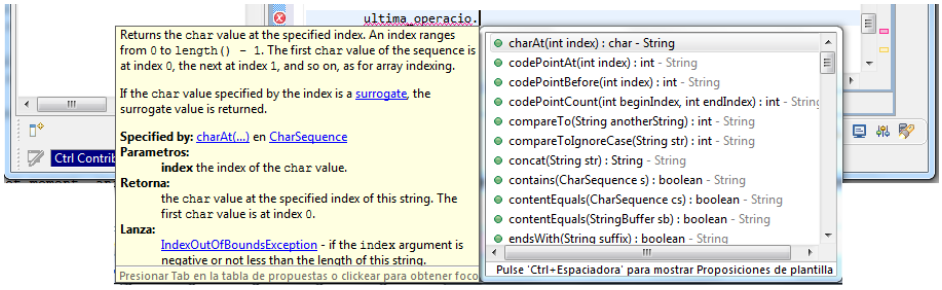
El mecanisme de *code completion* en Eclipse és molt similar al que implementen altres entorns integrats de desenvolupament: quan es deixa d'escriure durant un determinat interval de temps es mostren, si n'hi ha, tots els termes (paraules reservades, noms de funcions, de variables, de camps...) que comencen amb el o els caràcters escrits.

Code Completion vol dir compleció automàtica de les sentències inacabades.

Una altra característica és que escrivint un punt es provoca, sense esperar el temps establert, que s'ofereixin les alternatives o termes proposats. A la figura 2.45 es pot observar un exemple. També és interessant l'assistència a l'escriptura en trucades a funcions. Automàticament, quan s'estan a punt d'escriure els paràmetres que

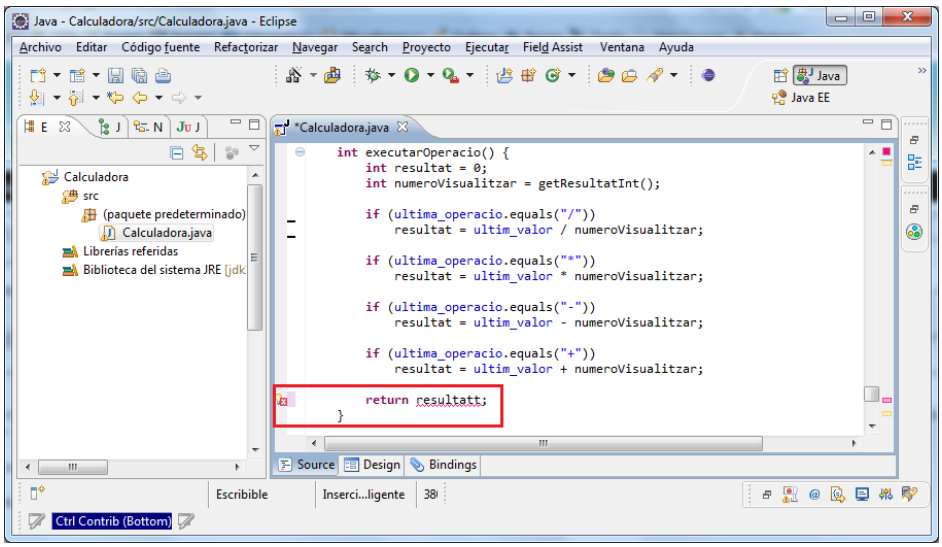
es passen a un mètode, es mostra una caixa de text indicant els tipus que aquests poden tenir.

Figura 2.45. Compleció de codi



D'altra banda, és bastant fàcil que a l'hora de codificar es cometi algun error.

Figura 2.46. Compleció de codi



El corrector d'errors serveix per a la detecció i el marcatge sobre el codi d'errors o avisos.

Com es pot observar a la figura 2.46, el connector JDT pot detectar, i marcar sobre el codi d'un programa, els llocs on es poden produir errors de compilació. Aquesta característica funciona de manera molt semblant als correctors ortogràfics que tenen els processadors de textos. Quan Eclipse detecta un error de compilació, s'ha de marcar la sentència errònia, subratllant amb una línia ondulada vermella (o groga, si en lloc d'un error es tracta d'un avís).

Si el programador posiciona el punter del ratolí sobre la instrucció que va produir la decisió, es mostrarà una breu explicació de per què aquesta instrucció s'ha marcat com a errònia.

Cada línia de codi que contingui un missatge d'error o un avís, també serà marcada amb una icona que apareixerà a la barra de desplaçament esquerra de l'Editor. Si el programador prem un cop sobre aquesta marca, es desplegarà un menú finestra emergent (*pop-up*) mitjançant el qual Eclipse mostrarà possibles solucions per als errors detectats.

L'error detectat en l'exemple de la figura 2.46 es troba en el nom de la variable. S'ha escrit erròniament *resultatt* en lloc de *resultat*.

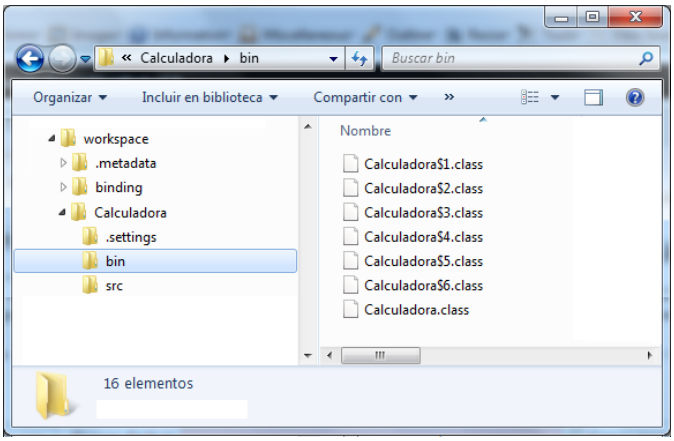
Una vegada desenvolupat el codi, caldrà **compilar-lo** per comprovar que el lèxic, la sintaxi i la semàntica siguin correctes, i generar, així, els fitxers binaris que es puguin executar.

Cal explicar com es compilen els projectes amb Eclipse, perquè ofereix un sistema una mica curiós. A Eclipse no existeix cap botó que permeti compilar individualment un fitxer concret. La compilació és una tasca que es llança automàticament en desar els canvis duts a terme en el codi. Per aquesta raó és pràcticament innecessari controlar manualment la compilació dels projectes.

Existeix una opció a l'entrada Project del menú principal, anomenada *Rebuild Project*, que permet llançar tot el procés de compilació complet, en cas que sigui necessari. Per compilar tots els projectes oberts, també es podrà utilitzar l'opció *Rebuild All*.

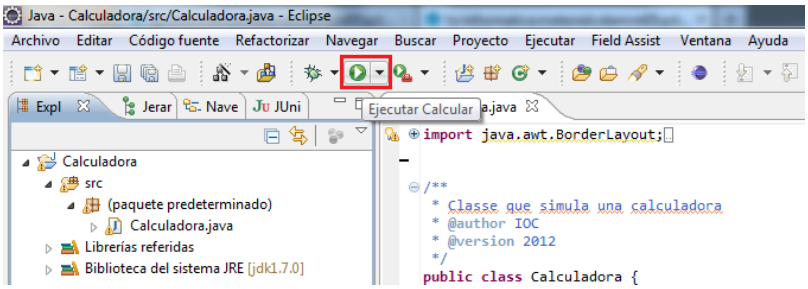
Tal com es mostra en la figura 2.47, en compilar, es genera una carpeta que sol denominar-se *bin* amb els fitxers .class, necessaris per poder executar el projecte.

Figura 2.47. Compilació



Una vegada el codi s'ha compilat, caldrà executar-lo per confirmar que fa el que s'havia planificat que hauria de fer. A la figura 2.48 es pot observar com accedir a la funcionalitat Run, que permetrà tant l'execució com la utilització d'altres opcions, entre les quals hi ha la d'accedir a la depuració per un altre camí.

Figura 2.48. Execució



2.5 Executables

En les aplicacions desenvolupades en el llenguatge C, C++, Visual Basic, Pascal... els fitxers binaris que genera el compilador únicament poden ser executats en la plataforma sobre la qual es va desenvolupar.

En les aplicacions desenvolupades en Java, els fitxers binaris que genera el compilador no és específic d'una màquina física en particular, sinó d'una màquina virtual, la qual cosa permet generar aplicacions compatibles amb diverses plataformes.

Per aquest motiu, l'IDE Eclipse no genera executables sinó codi intermig (codi de bytes) que s'emmagatzema en el directori .bin en forma de fitxers .class. La distribució de les aplicacions i posterior desplegament es pot dur a terme creant algun dels següents tipus d'encapsulaments:

- **JAR** (Java Archive): és un format d'arxiu independent de la plataforma que permet que diversos arxius puguin ser encapsulats dins d'un de sol, de manera que aquest pugui ser una aplicació completa de fàcil mobilitat i execució.
- **WAR** (Web Application archive): és un arxiu JAR (amb l'extensió WAR) usat per distribuir una col·lecció d'arxius JSP, servlets, classes Java, arxius XML i contingut web estàtic (HTML). En conjunt constitueixen una aplicació Web.
- **EAR** (Enterprise Archive File): és un format per empaquetar diversos mòduls en un sol arxiu. Permet desplegar diversos mòduls d'aquests en un servidor d'aplicacions. Conté arxius XML, anomenats descriptors de desplegament, que descriuen com efectuar aquesta operació (EAR = JAR + WAR).

En la figura 2.49, es mostra una representació gràfica dels diferents tipus d'encapsulaments.

FIGURA 2.49. Tipus de fitxers d'encapsulament

Optimització de programari

Marcel García Vacas

Índex

Introducció

Un usuari final d'una aplicació informàtica no és coneixedor de com ha estat implementada ni codificada. El més important per a ell o ella és que aquesta aplicació faci el que hagi de fer, i, a més a més, que ho faci en el menor temps possible. Quina importància tindrà per a un usuari fer servir dos productes que, en definitiva, ofereixen les mateixes característiques? Imaginem el cas de dos televisors. Si els dos reprodueixen els mateixos canals i tenen les mateixes polzades i idèntiques característiques tècniques, què ens farà decidir si fer servir l'un o l'altre? Possiblement, en el cas dels televisors hi haurà intangibles, com la marca o l'estètica o altres característiques subjectives. Però, en el cas de les aplicacions informàtiques o de les pàgines web, quin pot ser l'element que faci que una sigui millor que l'altra si fan exactament el mateix i tenen les mateixes interfícies?

En el cas del programari, tot això és una mica abstracte; hi haurà molts intangibles que poden fer decantar cap a una aplicació o cap a una altra. Però el que segur que serà diferent és la forma d'haver creat i desenvolupat aquestes aplicacions.

En aquesta unitat, "Optimització de programari", veurem aquestes diferències entre un codi de programació normal i un codi optimitzat, i estudiarem les característiques que diferencien un tipus de codi d'un altre i les eines de què es pot disposar per fer-ho.

En l'apartat "Disseny i realització de proves de programari" s'expliquen les diferents formes i tècniques que permeten validar la correctesa del programari desenvolupat i la seva optimització. Hi ha moltes formes de poder crear aplicacions que facin el que han de fer i que acompleixin els requeriments establerts per als futurs usuaris. I moltes vegades, les diferències en el moment de la creació i del desenvolupament de les aplicacions faran que una es pugui considerar molt millor que l'altra, i per tant recomanable.

Què significa haver desenvolupat millor una aplicació informàtica? Significarà que el seu codi de programació sigui òptim, que hagi seguit els patrons d'optimització més recomanats, que s'hagin dut a terme totes les proves que cal fer per validar un funcionament correcte al 100%, que el seu desenvolupament s'hagi documentat, havent-hi un control de versions exhaustiu i molts altres detalls que es veuran en aquesta unitat formativa.

En canvi, una aplicació informàtica, sobretot si és de codi obert o s'ha desenvolupat a mida per una empresa externa o pel propi departament d'informàtica d'una organització, pot ser un producte viu, en contínua evolució. Un dia els informes es voldran veure d'una forma o d'una altra, o caldrà afegir-n'hi de nous; un altre dia les regles de negoci s'hauran modificat i caldrà revisar-les per al seu acompliment. Un altre, potser, caldrà modificar les funcionalitats que ofereix l'aplicació, ampliar-la amb unes de noves i treure'n d'altres.

En l'apartat "Eines per al control i documentació del programari" treballarem algunes tècniques que es fan servir per millorar la qualitat del codi de programari per part dels desenvolupadors. Un exemple és la refacció, però també n'hi ha d'altres com el control de versions o la documentació automàtica del programari.

Resultats d'aprenentatge

En finalitzar aquesta unitat l'alumne/a:

1. Verifica el funcionament de programes dissenyant i realitzant proves.

- Identifica els diferents tipus de proves.
- Defineix casos de prova.
- Identifica les eines de depuració i prova d'aplicacions ofertes per l'entorn de desenvolupament.
- Utilitza eines de depuració per definir punts de ruptura i seguiment.
- Utilitza les eines de depuració per examinar i modificar el comportament d'un programa en temps d'execució.
- Efectua proves unitàries de classes i funcions.
- Element de llista de pics
- Implementa proves automàtiques.
- Documenta les incidències detectades.

2. Optimitza codi emprant les eines disponibles en l'entorn de desenvolupament

- Identifica els patrons de refacció més usuals.
- Elabora les proves associades a la refacció.
- Revisa el codi font utilitzant un analitzador de codi.
- Identifica les possibilitats de configuració d'un analitzador de codi.
- Aplica patrons de refacció amb les eines que proporciona l'entorn de desenvolupament.
- Realitza el control de versions integrat en l'entorn de desenvolupament.
- Utilitza eines de l'entorn de desenvolupament per documentar les classes.

1. Disseny i realització de proves de programari

Les proves són necessàries en la fabricació de qualsevol producte industrial i, de forma anàloga, en el desenvolupament de projectes informàtics. Qui posaria a la venta una aspiradora sense estar segur que aspira correctament? O una ràdio digital sense haver comprovat que pugui sintonitzar els canals?

Una aplicació informàtica no pot arribar a les mans d'un usuari final amb errades, i menys si aquestes són prou visibles i clares com per haver estat detectades pels desenvolupadors. Es donaria una situació de manca de professionalitat i disminuïria la confiança per part dels usuaris, que podria mermar oportunitats futures.

Quan cal dur a terme les proves? Què cal provar? Totes les fases establertes en el desenvolupament del programari són importants. La manca o mala execució d'alguna d'elles pot provocar que la resta del projecte arrossegui un o diversos errors que seran determinants per al seu l'èxit. Com més aviat es detecti un error, menys costós serà de solucionar.

També seran molt importants les proves que es duran a terme una vegada el projecte estigui finalitzat. És per això que la fase de proves del desenvolupament d'un projecte de programari es considera bàsica abans de fer la transferència del projecte a l'usuari final. Qui donaria un cotxe per construït i finalitzat si en intentar arrencar-lo no funcionés?

1.1 Introducció

Qualsevol membre de l'equip de treball d'un projecte informàtic pot cometre errades. Les errades, a més, es podran donar a qualsevol de les fases del projecte (anàlisi, disseny, codificació ...). Algunes d'aquestes errades seran més determinants que d'altres i tindran més o menys implicacions en el desenvolupament futur del projecte.

Les fases de desenvolupament d'un projecte són: presa de requeriments, anàlisi, disseny, desenvolupament, proves, finalització i transferència.

Per exemple, un cap de projecte, a l'hora de planificar les tasques de l'equip, estipula el disseny de les interfícies en 4 hores de feina per a un únic dissenyador. Si, una vegada executada aquesta tasca del projecte, la durada ha estat de 8 hores i s'han necessitat dos dissenyadors, la repercussió en el desenvolupament del projecte serà una desviació en temps i en cost, que potser es podrà compensar utilitzant menys recursos o menys temps en alguna tasca posterior.

En canvi, si l'errada ha estat del dissenyador de la base de dades, que ha obviat un camp clau d'una taula principal i la seva vinculació amb una segona taula, aquesta pot ser molt més determinant en les tasques posteriors. Si es creen les interfícies a

partir d'aquesta base de dades errònia i es comença a desenvolupar el programari sense identificar l'errada, pot succeir que al cap d'unes quantes tasques es detecti l'errada i que, per tant, calgui tornar al punt d'inici per solucionar-la.

> Un **error** no detectat a l'inici del desenvolupament d'un projecte pot arribar a necessitar cinquanta vegades més esforços per ser solucionat que si és detectat a temps.

En un projecte de desenvolupament de programari està estipulat que es dedica entre un 30% i un 50% del cost de tot el projecte a la fase de proves. Amb aquesta dada ens podem adonar de la importància de les proves dins un projecte. Els resultats de les proves podran influir en la percepció que tindrà el client final en relació amb el producte (programari) lliurat i la seva qualitat.

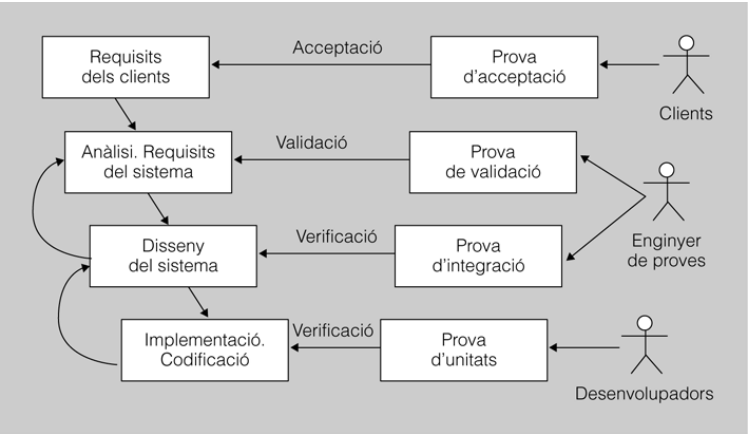
Precisament, és l'objectiu d'aquestes proves: l'avaluació de la qualitat del programari desenvolupat durant tot el seu cicle de vida, validant que fa el que ha de fer i que ho fa tal com es va dissenyar, a partir dels requeriments.

1.2 Les proves en el cicle de vida d'un projecte

A cada una de les fases del cicle de vida d'un projecte, caldrà que el treball dut a terme sigui validat i verificat.

En l'esquema de la figura 1.1 podem veure com encaixen les proves en el cicle de vida del programari.

FIGURA 1.1. Cicle de vida en V: fases de prova



Depuradors

Els depuradors (*debuggers*) són una aplicacions o eines permeten l'execució controlada d'un programa o un codi, seguint el comandament executat i localitzant els errors (*bugs*) que puguin contenir.

Alhora que s'avança en el desenvolupament del programari es van planificant les proves que es faran a cada fase del projecte. Aquesta planificació es concretarà en un pla de proves que s'aplicarà a cada producte desenvolupat. Quan es detecten errors en un producte s'ha de tornar a la fase anterior per depurar-lo i corregir-lo; això s'indica amb les fletxes de tornada de la part esquerra de la figura.

Hi ha eines CASE que ajuden a dur a terme aquests processos de prova; concretament, es coneixen com a depuradors els encarregats de depurar errors en els programes.

Com es pot observar a la figura 1.1, el procés de verificació cobrirà les fases de disseny i implementació del producte. Les persones implicades en la seva execució seran els desenvolupadors o programadors i l'enginyer de proves.

Els desenvolupadors faran proves sobre el codi i els diferents mòduls que l'integren, i l'enginyer, sobre el disseny del sistema.

Validació és el terme que es fa servir per avaluar positivament si el producte desenvolupat compleix els requisits establerts en l'anàlisi. Les persones encarregades de fer les proves de validació són els enginyers de proves.

Finalment, el client ha de donar el vistiplau al producte, raó per la qual es faran les proves d'acceptació en funció de les condicions que es van signar al principi del contracte.

1.3 Procediments, tipus i casos de proves

En cada una de les fases d'un projecte s'haurà de dedicar un temps considerable a desenvolupar les tasques i els procediments referents a les proves. És per això que alguns autors consideren que els procediments relacionats amb les proves són com un petit projecte englobat dins el projecte de desenvolupament.

Aquest projecte de proves requerirà d'una planificació, un disseny del pla de proves, una execució de les mateixes i una avaluació dels resultats, per tal d'analitzar els errors i poder aplicar les accions necessàries. A la figura 1.2 es mostra un esquema amb els procediments que caldrà dur a terme i la documentació que s'haurà d'adjuntar. Aquest esquema servirà com a índex per a aquest apartat.

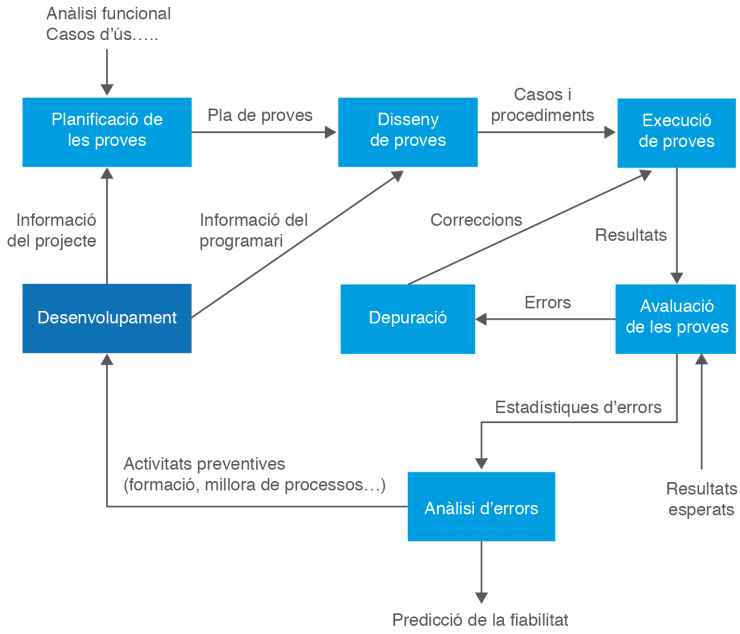
Consulteu l'obra de Presman, R. S. en la secció *Bibliografia bàsica* del web del mòdul (pàgines 419-469).

L'esquema s'inicia amb una planificació de les proves, que té com a punt de partida l'anàlisi funcional, diagrames de casos d'ús... del producte a desenvolupar. En la planificació s'estimaran els recursos necessaris per a l'elaboració de les proves i la posterior validació del programari, i s'obtindrà un pla de proves com a sortida.

Partint del pla de proves i del codi font que s'hagi desenvolupat, es durà a terme el disseny de les proves identificant quin tipus de proves s'efectuarà per a cada una de les funcionalitats, i s'obtindran, com a sortida, els casos de prova i procediments. A partir d'aquest moment, es crea un bucle on s'executaran les proves, s'avaluaran els resultats de les proves efectuades detectant els errors, es depurarà el codi aplicant les correccions pertinents i es tornaran a executar les proves.

En finalitzar el bucle, es farà l'anàlisi de l'estadística d'errors. Aquesta anàlisi permetrà fer prediccions de la fiabilitat del programari, així com detectar les causes més habituals d'error, amb la qual cosa es podran millorar els processos de desenvolupament.

Figura 1.2. Procés de proves



1.3.1 Planificació de les proves

La planificació de les proves és una tasca que cal anar desenvolupant al llarg de totes les fases del projecte informàtic. No cal esperar la fase de programació per crear aquest pla de proves; a la fase d'anàlisi i a la fase de disseny ja es tenen prou dades per tal de poder començar a establir les primeres línies del pla de proves.

És important tenir present que, com més aviat es detecti una **errada** al projecte informàtic, més fàcil serà contrarestar i solucionar aquest error. El cost de la resolució d'un problema creix exponencialment a mesura que avancen les fases del projecte en les quals es detecti.

Però, com succeeix en molts aspectes de la vida, una cosa és la teoria i una altra de molt diferent la realitat. En moltes consultories especialitzades en desenvolupament del programari, així com en altres empreses més petites o, fins i tot, per part de programadors independents que creen el seu propi programari, es consideren les proves com un procés que es tracta al final del projecte, una vegada la major part del codi ha estat desenvolupat.

La **planificació de les proves** té com a objectiu arribar a la creació d'un pla d'actuació que es refereixi a quan i com es duran a terme les proves. Però per a això cal dur a terme una anàlisi minuciosa del sistema i dels seus elements. El pla de proves ha de contenir totes les funcions, les estratègies, les tècniques i els membres de l'equip de treball implicats.

IEEE

IEEE és l'acrònim per a Institute of Electrical and Electronics Engineers, una associació professional sense ànim de lucre que determina la major part dels estàndards en les Enginyeries.

Una bona guia per determinar què contindrà un bon pla de proves es pot obtenir de la normativa IEEE 829-2008 “Standard for Software and System Test Documentation”. Aquest estàndard estableix com haurà de ser la documentació i els procediments que es faran servir en les diferents etapes de les proves del programari. Alguns dels continguts del pla de proves són:

- **Identificador del pla de proves**. És l'identificador que s'assignarà al pla de proves. És important per poder identificar fàcilment quin abast té el pla de proves. Per exemple, si es volen verificar les interfícies i procediments relacionats amb la gestió de clients, el seu pla de proves es podria dir PlaClients.
- **Descripció del pla de proves**. Defineix l'abast del pla de proves, el tipus de prova i les seves propietats, així com els elements del programari que es volen provar.
- **Elements del programari a provar**. Determina els elements del programari que s'han de tenir en compte en el pla de proves, així com les condicions mínimes que s'han de complir per dur-ho terme.
- **Elements del programari que no s'han de provar**. També és important definir els elements que no s'hauran de tenir en compte al pla de proves.
- **Estratègia del pla de proves**. Defineix la tècnica a utilitzar en el disseny dels casos de prova, com per exemple la tècnica de capsa blanca o de capsa negra, així com les eines que s'utilitzaran o, fins i tot, el grau d'automatització de les proves.
- **Definició de la configuració del pla de proves**. Defineix les circumstàncies sota les quals el pla de proves podrà ser alterat, finalitzat, suspès o repetit. Quan s'efectuïn les proves, s'haurà de determinar quin és el punt que provoca que se suspenguin, ja que no tindria gaire sentit continuar provant el programari quan aquest es troba en un estat inestable. Una vegada els errors han estat corregits, es podrà continuar efectuant les proves; és possible que s'iniciïn des del principi del pla o des d'una determinada prova. Finalment, es podrà determinar la finalització de les proves si aquestes han superat un determinat llindar.
- **Documents a lliurar**. Defineix els documents que cal lliurar durant el pla de proves i en finalitzar-lo. Aquesta documentació ha de contenir la informació referent a l'èxit o fracàs de les proves executades amb tot tipus de detall. Alguns d'aquests documents poden ser: resultats dels casos de proves, especificació de les proves, subplans de proves...
- **Tasques especials**. Defineix les taques necessàries per preparar i executar les proves. Però hi ha algunes tasques que tindran un caràcter especial, per la seva importància o per la seva dependència amb d'altres. Per a aquest tipus de tasques, serà necessari efectuar una planificació més detallada i determinar sota quines condicions es duran a terme.
- **Recursos**. Per a cada tasca definida dins el pla de proves, s'haurà d'assignar un o diversos recursos, que seran els encarregats de dur-la a terme.

- **Responsables i Responsabilitats**. Es defineix el responsable de cadascuna de les tasques previstes en el pla.
- **Calendari del pla de proves**. En el calendari queden descrites les tasques que s'hauran d'executar, indicant les seves dependències, els responsables, les dates d'actuació i la durada, així com les fites del pla de proves. Una eina molt utilitzada per representar aquest calendari del pla de proves és el Diagrama de Gantt.

Un error típic és tractar la planificació de proves com una activitat puntual, limitada al període de temps en què s'elabora una llista d'accions per ser desenvolupades durant els propers dies, mesos... La planificació és un procés continu i no puntual; per tant, cal treballar-hi al llarg de tot el projecte, i per a això és necessari:

- **Gestionar els canvis:** no és d'estranyar que, al llarg del cicle de vida del projecte, i amb major probabilitat quan més llarga és la seva durada, es presentin canvis en l'abast del projecte. Serà necessari adaptar el pla de proves a les noves especificacions.
- **Gestionar els riscos**: un risc es podria definir com un conjunt de situacions que poden provocar un impediment o retard en el pla de proves. Els riscos s'hauran d'identificar al més aviat possible i analitzar la probabilitat que hi ha que succeeixin, tot aplicant mesures preventives, si es considera oportú, o disposar d'un pla de contingència en cas que el risc sorgís.

> En la planificació d'un projecte informàtic el temps dedicat a les proves acostuma a ser molt limitat.

D'aquesta manera, es pot afirmar que tota planificació ha de ser viva i s'ha d'anar revisant i controlant de forma periòdica. En cas de desviació, s'han d'analitzar les causes que l'han provocat i com afecta a la resta dels projectes.

1.3.2 Disseny de les proves. Tipus de proves

El disseny de les proves és el pas següent després d'haver dut a terme el pla de proves. Aquest disseny consistirà a establir els casos de prova, identificant, en cada cas, el tipus de prova que s'haurà d'efectuar.

Existeixen molts tipus de proves:

- Estructurals o de capsa blanca
- Funcionals o de capsa negra
- D'integració
- De càrrega i acceptació

- De sistema i de seguretat
- De regressió i de fum

Casos de prova i procediments

A partir del pla de proves s'hauran especificat les parts de codi a tractar, en quin ordre caldrà fer les proves, qui les farà i molta informació més. Ara només falta entrar en detall, especificant el cas de prova per a cada una de les proves que cal fer.

> Un **cas de prova** defineix com es portaran a terme les proves, especificant, entre d'altres: el tipus de proves, les entrades de les proves, els resultats esperats o les condicions sota les quals s'hauran de desenvolupar.

Els casos de proves tenen un objectiu molt marcat: identificar els errors que hi ha al programari per tal que aquests no arribin a l'usuari final. Aquests errors poden trobar-se com a defectes en la interfície d'usuari, en l'execució d'estructures de dades o un determinat requisit funcional.

Als materials Web, tant en la secció d'*Annexos* com en la d'*Activitats*, es poden trobar exemples de casos de proves.

A l'hora de dissenyar els casos de prova, no tan sols s'ha de validar que l'aplicació fa el que s'espera davant entrades correctes, sinó que també s'ha de validar que tingui un comportament estable davant entrades no esperades, tot informant de l'error.

Per desenvolupar i executar els casos de prova en un projecte informàtic, podem identificar dos enfocaments:

- **Proves de capsa negra:** El seu objectiu és validar que el codi compleix la funcionalitat definida.
- **Proves de capsa blanca:** Se centren en la implementació dels programes per escollir els casos de prova..

En aquest apartat es veuran en profunditat tant les proves de capsa negra com les de capsa blanca.

En l'exemple que es mostra a continuació, s'ha implementat en Java un cas de prova que valida el cost d'una matricula.

El mètode `CasProva_CostMatricula` calcularà el preu que haurà de pagar un alumne per matricular-se en diverses assignatures. La prova valida que el càcul de l'operació coincideixi amb el resultat esperat, fent ús de la instrucció `assertTrue`.

```
1  CONST PREU_CREDIT = €100
2  public void CasProva_CostMatricula(){
3      try {
4          int credits = 0;
5          float preu = 0;
6          crèdits = CreditsAssignatura ("Sistemes informàtics"); //12 crèdits
7          crèdits += CreditsAssignatura ("Programació");         //15 crèdits
8          crèdits += CreditsAssigantura ("Accés a dades");       //12 crèdits
9    preu = crèdits * PREU_CREDIT;
10   assertTrue (preu€==00);
11     }catch (Exception e) {Fail ("S'ha produït un error");}
12 }
```

D'aquesta manera, podem observar que els casos de prova ens permeten validar l'aplicació que s'està desenvolupant, essent necessari tenir accessible la documentació generada en l'anàlisi funcional i l'anàlisi tècnica, així com en el disseny (casos d'ús, diagrames de seqüència...).

Els casos de prova segueixen un cicle de vida clàssic:

- Definició dels casos de prova.
- Creació dels casos de prova.
- Selecció dels valors per als tests.
- Execució dels casos de prova.
- Comparació dels resultats obtinguts amb els resultats esperats.

Cada cas de prova haurà de ser independent dels altres, tindrà un començament i un final molt marcat i haurà d'emmagatzemar tota la informació referent a la seva definició, creació, execució i validació final.

Tot seguit s'indiquen algunes informacions que hauria de contemplar qualsevol cas de prova:

- Identificador del cas de prova.
- Mòdul o funció a provar.
- Descripció del cas de prova detallat.
- Entorn que s'haurà de complir abans de l'execució del cas de prova.
- Dades necessàries per al cas, especificant els seus valors.
- Tasques que executarà el pla de proves i la seva seqüència.
- Resultat esperat.
- Resultat obtingut.
- Observacions o comentaris després de l'execució.
- Responsable del cas de prova.
- Data d'execució.
- Estat (finalitzat, pendent, en procés).

Tot cas de prova està associat, com a mínim, a un procediment de prova.

Els **procediments de prova** especifiquen com es podran dur a terme els casos de prova o part d'aquests de forma independent o de forma conjunta, establint les relacions entre ells i l'ordre en què s'hauran d'atendre.

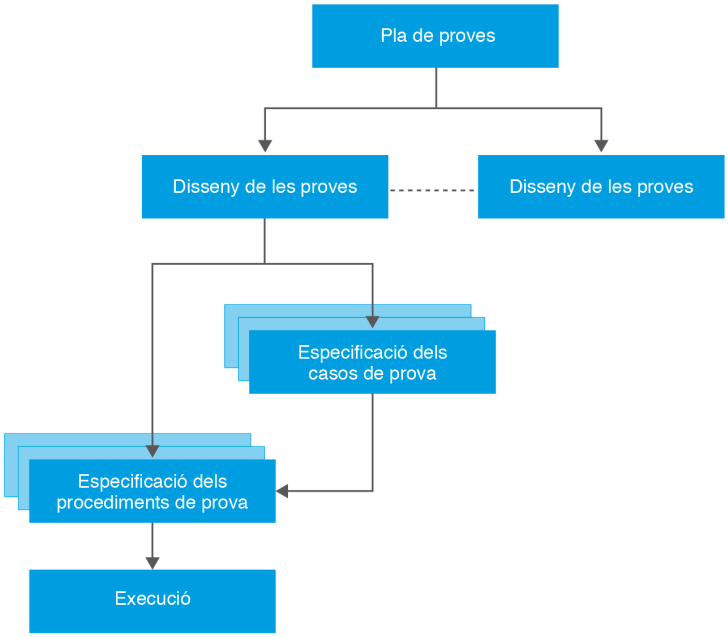
Per exemple, es pot dissenyar un procediment de prova per inserir una nova assignatura en la matrícula d'un alumne; s'elaboren tots els passos necessaris de forma consecutiva per tal d'actualitzar la matrícula de l'alumne. No obstant això, l'assignatura pot ser obligatòria, optativa, amb unes incompatibilitats...; per tant, pel que fa al procediment de prova d'inserir la matrícula serà necessari associar un grup de casos de prova responsables de les diverses entrades de l'usuari.

Atenció deficient a les proves

Els equips de treball han de planificar, dissenyar i fer moltes proves en un curt espai de temps. Aquesta situació comporta una atenció deficient a una part tan important com les proves, deixant camins per atendre, duent a terme proves redundants, no escollint encertadament els casos de prova...

A la figura 1.3 es pot observar un esquema explicatiu de les proves, des de la creació del pla de proves fins a la seva execució. Per a cada part del projecte caldrà crear un disseny de les proves, a partir del qual s'especificaran els casos de prova i els procediments de prova, que estan estretament lligats. Després dels procediments de prova, el darrer pas serà la seva execució.

FIGURA 1.3. Casos de proves i procediments



Tipus de proves

Existeixen molts tipus de proves que han de cobrir les especificacions d'un projecte informàtic a través dels procediments i dels casos de prova.

Tot seguit es presenta un resum d'aquests tipus de proves:

- **Tipus de proves unitàries**. Tenen les característiques següents:
  - Són el tipus de proves de més baix nivell.
  - Es duen a terme a mesura que es va desenvolupant el projecte.
  - Les efectuen els mateixos programadors.
  - Tenen com a objectiu la detecció d'errors en les dades, en els algorismes i en la lògica d'aquests.
  - Les proves unitàries es podran dur a terme segons un enfocament estructural o segons un enfocament funcional.

  - El mètode utilitzat en aquest tipus de proves és el de la capsa blanca o el de capsa negra.

- **Tipus de proves funcionals**. D'aquestes proves cal destacar:
  - Són les encarregades de detectar els errors en la implementació dels requeriments d'usuari.
  - Les duran a terme els verificadors i els analistes, és a dir, persones diferents a aquelles que han programat el codi.
  - S'efectuen durant el desenvolupament del projecte.
  - El tipus de mètode utilitzat és el funcional.

- **Tipus de proves d'integració**. Les seves característiques són les següents:
  - Es duran a terme posteriorment a les proves unitàries.
  - També les efectuen els mateixos programadors.
  - Es duen a terme durant el desenvolupament del projecte.
  - S'encarreguen de detectar errors de les interfícies i en les relacions entre els components.
  - El mètode utilitzat és el de capsa blanca, el de disseny descendent i el de *bottom-up*.

- **Tipus de proves de sistemes**. En destaca:
  - La seva finalitat és detectar errors en l'assoliment dels requeriments.
  - Les duran a terme els verificadors i els analistes, és a dir, persones diferents a aquelles que han programat el codi.
  - S'efectuen en una fase de desenvolupament del programari.
  - El tipus de mètode utilitzat és el funcional.

- **Tipus de proves de càrrega**. Les seves característiques principals són:
  - S'efectuen un cop acabat el desenvolupament, però abans de les proves d'acceptació.
  - També les realitzen analistes i verificadors.
  - Es comprova el rendiment i la integritat de l'aplicació ja acabada amb dades reals i en un entorn que també simula l'entorn real.
  - Es realitzen amb un enfocament funcional.

- **Tipus de proves d'acceptació**. Els aspectes més importants d'aquestes proves són els següents:
  - El seu objectiu és la validació o acceptació de l'aplicació per part dels usuaris.
  - És per això que les duran a terme els clients o els usuaris finals de l'aplicació.
  - Aquestes proves es duran a terme una vegada finalitzada la fase de desenvolupament. És possible fer-ho en la fase prèvia a la finalització i a la transferència o en la fase de producció, mentre els usuaris ja fan servir l'aplicació.

  - El tipus de mètode utilitzat també és el funcional.
  - Inclouen diferents tipus de prova. Entre altres, les proves alfa i les proves beta. En aquestes, el client realitza les proves a l'entorn del desenvolupador, al primer cas, i, al segon cas, en el propi entorn del client.

- **Tipus de proves de sistema**. Les característiques principals d'aquestes proves són:
  - Es realitzen després de les proves d'acceptació i amb el sistema ja integrat a l'entorn de treball.
  - La seva finalitat, precisament, és comprovar que aquesta integració és correcta.
  - L'enfocament utilitzat, lògicament, és el de capsa negra.
  - Les realitzen verificadors i analistes.
  - Inclou diferents tipus de prova. Entre altres, proves de rendiment, de resistència, de robustesa (davant entrades incorrectes), de seguretat, d'usabilitat i d'instal·lació.

- **Tipus de proves de regressió**. Les seves característiques principals són:
  - La seva finalitat és detectar possibles errors introduits en haver realitzat canvis al sistema, bé per millorar-lo, bé per corregir altres errors.
  - Consisteixen bàsicament en repetir proves ja realitzades amb èxit abans de realitzar el canvi. Per tant, inclourà tant proves de capsa blanca com de capsa negra.

- **Tipus de proves "de fum"**. Són proves ràpides de les funcions bàsiques d'un programari que normalment es realitzen després d'un canvi en el codi abans de registrar aquest codi modificat en la documentació del projecte.

Als materials Web, tant en la secció d'*Annexos* com en la d'*Activitats*, hi podeu trobar exemples de casos de proves.

Proves unitàries: enfocament estructural o de capsa blanca

Les proves unitàries, també conegudes com a proves de components, són les proves que es faran a més baix nivell, sobre els mòduls o components més petits del codi font del projecte informàtic.

Aquestes proves poden desenvolupar-se sota dos enfocaments:

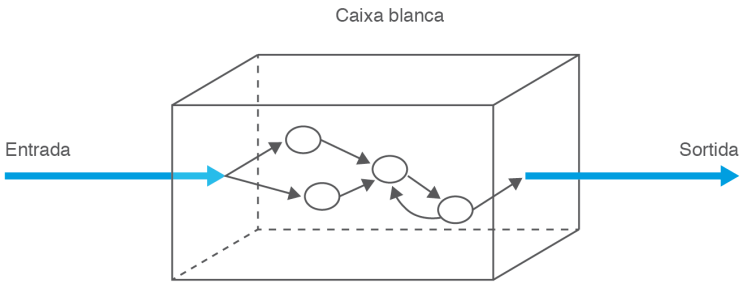
- L'enfocament estructural és la part de les proves unitàries encarregades de l'estructura interna del codi font, des de qual s'analitzen tots els possibles camins.

- L'enfocament funcional (o proves de capsa negra) és la part de les proves unitàries encarregades del funcionament correcte de les funcionalitats del programari.

Les proves de capsa blanca se centren en la implementació dels programes per escollir els casos de prova. L'ideal seria cercar casos de prova que recorreguessin tots els camins possibles del flux de control del programa. Aquestes proves se centren en l'estructura interna del programa, tot analitzant els camins d'execució.

A la figura 1.4 es pot observar un esquema de l'estructura que tenen les proves de capsa blanca. A partir d'unes condicions d'entrada al mòdul o part de codi cal validar que, anant per la bifurcació que es vagi, s'obtindran les condicions desitjades de sortida.

FIGURA 1.4. Estructura de les proves de capsa blanca



Les proves de capsa blanca permetran recórrer tots els possibles camins del codi i veure què succeeix en cada cas possible. Es provarà què ocorre amb les condicions i els bucles que s'executen. Les proves es duran a terme amb dades que garanteixin que han tingut lloc totes les combinacions possibles. Per decidir quins valors hauran de prendre aquestes dades és necessari saber com s'ha desenvolupat el codi, tot cercant que no quedi cap racó sense revisar.

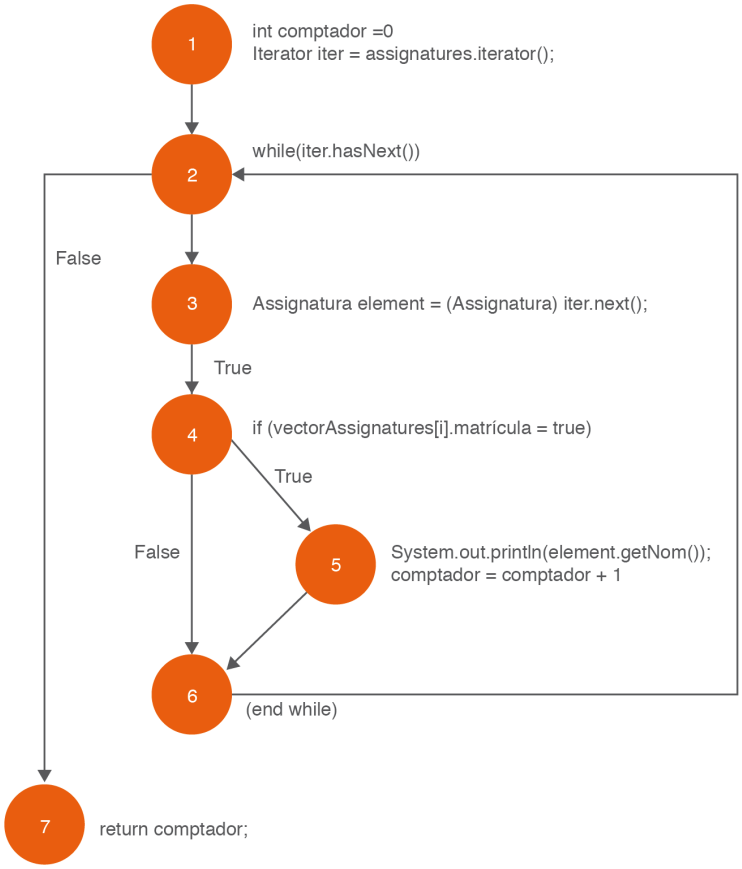
Partint del fet que les proves exhaustives són impracticables, ja que el nombre de combinacions és excessiu, es dissenyen estratègies que ofereixin una seguretat acceptable per descobrir errors. Els mètodes que es veuran dintre de les proves de capsa blanca són el de **cobertura de flux de control** i el de **complexitat ciclomàtica**.

Cobertura de flux de control

El mètode de cobertura de flux de control consisteix a utilitzar l'estructura de control del programa per obtenir els casos de prova, que són dissenyats de manera que garanteixin que almenys es passa una vegada per cada camí del programa.

Una possible tècnica per portar a terme aquest mètode consisteix a obtenir un diagrama de flux de control que representi el codi i provar tots els camins simples, totes les condicions i tots els bucles del programa. Es pot observar un exemple de diagrama de flux a la figura 1.5.

Figura 1.5. Diagrama de flux del llistat d'assignatures.



Pot ser impossible cobrir-ne el 100% si el programa és molt complex, però podem tenir un mínim de garanties d'eficàcia si seguim els suggeriments per dissenyar els casos de prova tenint en compte el següent:

- **Conjunt bàsic de camins independents**: és el conjunt de camins independents que cal cobrir amb el joc de proves.
- **Camí independent**: camí simple amb alguna branca no inclosa encara a cap camí del conjunt bàsic.
- **Camí simple**: camí que no té cap branca repetida.
- **Casos de prova**: un cop determinat el conjunt bàsic, cal dissenyar un cas de prova per a cadascun dels seus camins de manera que, entre tots s'executi almenys una vegada cada sentència.
- **Condicions**: cal assegurar-se que els casos de prova elaborats d'aquesta manera cobreixen totes les condicions del programa que s'avaluen a cert/fals. Cal tenir en compte, però, que, les condicions múltiples, s'han de dividir en expressions simples (una per a cada operand lògic o comparació), de manera que s'ha de provar que es compleixi o no cada part de cada condició. Per tant, en realitzar el gràfic, a una condició múltiple li correspondrà un node per cada condició simple que formi part d'ella i caldrà afegir també al

gràfic les branques necessaris per representar correctament el funcionament d'aquestes condicions.

- **Bucles**: s'han de dissenyar els casos de prova de manera que s'intenti executar un bucle en diferents situacions límit.

Per tal d'explicar el funcionament de les proves unitàries de capsa blanca es planteja l'exemple següent:

```
1  public float LlistatAssignatures(ArrayList assignatures)
2  {
3        int comptador= 0;
4        Iterator iter = assignatures.iterator();
5        while (iter.hasNext())
6        {
7              Assignatura element = (Assignatura) iter.next();
8              if (element.getDisponible() == true)
9              {
10                   System.out.println(element.getNom());
11                   comptador = comptador + 1
12             }
13       }
14       return comptador;
15 }
```

La funció `LlistatAssignatures` mostra, per pantalla, les assignatures que estan disponibles per tal que els alumnes es puguin matricular, i retorna un valor numèric corresponent al nombre d'assignatures disponibles.

En la figura 1.5 es representen gràficament els nodes de la funció, cosa que facilita el càlcul de la complexitat ciclomàtica.

Complexitat ciclomàtica

L'estratègia de cobertura de flux de control requereix dissenyar casos de prova suficients per recórrer tota la lògica del programa. Es pot saber quants casos de prova cal crear i executar? Com es calcula?

El matemàtic Thomas J. McCabe va anomenar complexitat ciclomàtica (CC) al nombre de camins independents d'un diagrama de flux, i va proposar la fórmula següent per calcular-la:

Complexitat ciclomàtica = nombre de branques – nombre de nodes + 2

La complexitat ciclomàtica del graf de l'exemple de la figura 1.5 proporciona el nombre màxim de camins linealment independents.

Els nodes que intervenen són 1, 2, 3, 4, 5, 6 i 7, i les branques són les línies que uneixen els nodes, que són un total de 8.

Complexitat ciclomàtica CC = 8 – 7 + 2 = 3

Això significa que s'hauran de dissenyar tres casos de prova. Així, els tres recorreguts que s'haurien de tenir en compte són:

- Camí 1: 1 – 2 – 7
- Camí 2: 1 – 2 – 3 – 4 – 6 – 2 - 7
- Camí 3: 1 – 2 – 3 – 4 – 5 – 6 – 2 - 7

> El conjunt bàsic depén de l'ordre en què hi afegim els camins independents. Una estratègia possible és triar els camins de més curt a més llarg.

Restarà generar les proves per recórrer els camins anteriors.

A la figura 1.6 es mostren els valors del vector d'assignatures.

FIGURA 1.6. Valors del vector d'assignatures

Camí 1

Assignatura	
Id	
nom	
hores	
crèdits	
disponible	

Camí 2

Assignatura	
Id	123
nom	Formació en centres de treball
hores	317
crèdits	18
disponible	false

Camí 3

Assignatura	
Id	123
nom	Bases de dades
hores	231
crèdits	12
disponible	true

Hi ha tres camins a recórrer. Per a cadascun d'ells serà necessari un vector amb una característiques concretes:

- Per recórrer el **camí 1** serà necessari un vector d'assignatures buit.
- Per recórrer el **camí 2** serà necessari un vector d'assignatures que contingui com a mínim una assignatura que no estigui disponible.
- Per recórrer el **camí 3** serà necessari un vector d'assignatures que contingui com a mínim una assignatura que estigui disponible.

D'aquesta manera, el resultats de les proves serien:

- Entrada (Assignatures1)

  Sortida esperada: 0

  Sortida real: 0

  Camí seguit: 1.

- Entrada (Assignatures2)

  Sortida esperada: 1

  Sortida real: 1

  Camí seguit: 2.

- Entrada (Assignatures3)

  Sortida esperada: 1

  Sortida real: 1

  Camí seguit: 3.

Una vegada executat el joc de proves de capsa blanca, es pot afirmar que han estat superades satisfactòriament.

Proves unitàries: enfocament funcional o proves de capsa negra

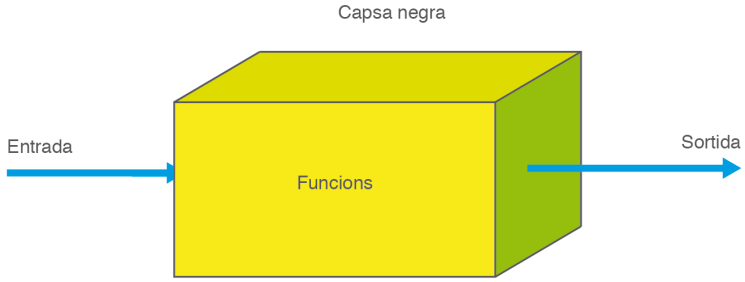
L'enfocament estructural o les proves de capsa blanca, dins les proves unitàries, serveix per analitzar el codi en totes les seves estructures, en tots els seus camins del programari. Però existeix un altre tipus de proves que es basa en un enfocament més funcional, anomenades proves de capsa negra.

> **Les proves de capsa negra** proven la funcionalitat del programa, per al qual es dissenyen casos de prova que comprovin les especificacions del programa.

Les tècniques de prova de capsa negra pretenen trobar errors en funcions incorrectes o absents, errors d'interfície, errors de rendiment, inicialització i finalització. Es centra en les funcions i en les seves entrades i sortides.

A la figura 1.7 es pot observar un esquema de l'estructura que tenen les proves de capsa blanca.

Figura 1.7. Estructura de les proves de capsa negra



Caldrà escollir amb cura els casos de prova, de manera que siguin tan pocs com sigui possible per tal que la prova es pugui executar en un temps raonable i, al mateix temps, que cobreixin la varietat d'entrades i sortides més àmplia possible.

Per aconseguir-ho, s'han dissenyat diferents tècniques:

- **Classes d'equivalència**: es tracta de determinar els diferents tipus d'entrada i sortida, agrupar-los i escollir casos de prova per a cada tipus o conjunt de dades d'entrada i sortida.
- **Anàlisi dels valors límit**: estudien els valors inicials i finals, ja que estadísticament s'ha demostrat que tenen més tendència a detectar errors.
- **Estudi d'errors típics**: l'experiència diu que hi ha una sèrie d'errors que s'acostumen a repetir en molts programes; per això, es tractaria de dissenyar casos de prova que provoquessin les situacions típiques d'aquest tipus d'errors.
- **Maneig d'interfície gràfica**: per provar el funcionament de les interfícies gràfiques, s'han de dissenyar casos de prova que permetin descobrir errors en el maneig de finestres, botons, icones...
- **Dades aleatòries**: es tracta d'utilitzar una eina que automatitzi les proves i que generi d'una manera aleatòria els casos de prova. Aquesta tècnica no optimitza l'elecció dels casos de prova, però si es fa durant prou temps amb moltes dades, podrà arribar a fer una prova bastant completa. Aquesta tècnica es podria utilitzar com a complementària a les anteriors o en casos en què no sigui possible aplicar-ne cap altra.

Un **avantatge de les proves de capsa negra** és que són independents del llenguatge o paradigma de programació utilitzat, de manera que són vàlides tant per a programació estructurada com per a programació orientada a objectes.

Classes d'equivalència

S'han de dissenyar els casos de prova de manera que provin la major funcionalitat possible del programa, però que no incloguin massa valors. Per on començar? Quins valors s'han d'escollir?

Cal seguir els passos següents:

1. **Identificar les condicions**, restriccions o continguts de les entrades i les sortides.
2. **Identificar, a partir de les condicions, les classes d'equivalència de les entrades i les sortides**. Per identificar-ne les classes, el mètode proposa algunes recomanacions:
    - Cada **element de classe** ha de ser tractat de la mateixa manera pel programa, però cada classe ha de ser tractada de manera diferent en

relació amb una altra classe. Això assegura que n'hi ha prou de provar algun element d'una classe per comprovar que el programa funciona correctament per a aquesta classe, i també garanteix que cobrim diferents tipus de dades d'entrada amb cadascuna de les classes.

- Les classes han de recollir tant **dades vàlides com errònies**, ja que el programa ha d'estar preparat i no bloquejar-se sota cap circumstància.
- Si s'especifica un **rang de valors** per a les dades d'entrada, per exemple, si s'admet del 10 al 50, es crearà una classe vàlida ($10 \leq X \leq 50$) i dues classes no vàlides, una per als valors superiors ($X > 50$) i l'altra per als inferiors ($X < 10$).
- Si s'especifica un **valor vàlid** d'entrada i d'altres de no vàlids, per exemple, si l'entrada comença amb majúscula, es crea una classe vàlida (amb la primera lletra majúscula) i una altra de no vàlida (amb la primera lletra minúscula).
- Si s'especifica un **nombre de valors** d'entrada, per exemple, si s'han d'introduir tres nombres seguits, es crearà una classe vàlida (amb tres valors) i dues de no vàlides (una amb menys de dos valors i l'altra amb més de tres valors).
- Si hi ha un conjunt de **dades d'entrada concretes** vàlides, es generarà una classe per cada valor vàlid (per exemple, si l'entrada ha de ser vermell, taronja, verd, es generaran tres classes) i una altra per un valor no vàlid (per exemple, blau).
- Si no s'han recollit ja amb les classes anteriors, s'ha de seleccionar una classe per cada possible classe de **resultat**.

3. **Crear els casos de prova a partir de les classes d'equivalència detectades**. Per a això s'han de seguir els passos següents:
   - Escollir un valor que representi cada classe d'equivalència.
   - Dissenyar casos de prova que incloguin els valors de totes les classes d'equivalència identificades.

L'experiència prèvia de l'equip de proves pot ajudar a escollir els casos que més probabilitats tenen de trobar errors.

Per exemple, es volen definir les proves de capsa negra per a una funció que retorna el nom del mes a partir del seu valor numèric.

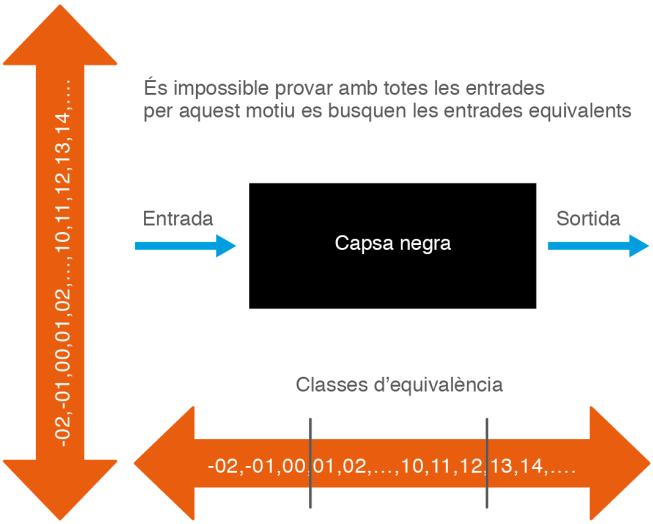
```
String nom;
    nom = NomDelMes(3);
    El valor del nom serà Març.
```

Caldrà identificar tres classes d'equivalències, com es pot observar a la figura 1.8:

```
..., −02, −01, 00 valors invàlids
    01, 02, 03, 04, 05, 06, 07, 08, 09, 10, 11, 12 valors vàlids
    13, 14, 15, ... valors invàlids.
```

A la figura 1.8 es pot observar un esquema de l'exemple exposat.

Figura 1.8. Exemple de capsa negra



Anàlisi de valors límit i errors típics

Hi ha tècniques que serveixen per seleccionar millor les classes d'equivalència. Una és **l'anàlisi dels valors límit**. Per què és una tècnica adequada fixar-se especialment en els valors límit?

S'ha pogut demostrar que els casos de prova que se centren en els valors límit produeixen un millor resultat per a la detecció de defectes.

D'aquesta manera, en escollir l'element representatiu de la classe d'equivalència, en lloc d'agafar-ne un qualsevol, s'escullen els valors al límit i, si es considera oportú, un valor intermedi. A més a més, també s'intenta que els valors a l'entrada provoquin valors límit als resultats.

A l'hora d'escollir els representants de cada classe se seguiran les recomanacions següents:

- En els rangs de valors, agafar els extrems del rang i el valor intermedi.
- Si s'especifiquen una sèrie de valors, agafar el superior, l'inferior, l'anterior a l'inferior i el posterior al superior.
- Si el resultat es mou en un determinat rang, hem d'escollir dades a l'entrada per provocar les sortides mínima, màxima i un valor intermedi.
- Si el programa tria una llista o taula, agafar l'element primer, l'últim i l'intermedi.

També es pot aprofitar l'experiència prèvia. Hi ha una sèrie d'errors que es repeteixen molt en els programes, i podria ser una bona estratègia utilitzar casos de prova que se centrin a buscar aquests errors. D'aquesta manera, es millorarà l'elecció dels representants de les classes d'equivalència:

- El valor zero sol provocar errors, per exemple, una divisió per zero bloqueja el programa. Si es té la possibilitat d'introduir zeros a l'entrada, s'ha d'escollir en els casos de prova.
- Quan s'ha d'introduir una llista de valors, caldrà centrar-se en la possibilitat de no introduir cap valor, o introduir-ne un.
- S'ha de pensar que l'usuari pot introduir entrades que no són normals, per això és recomanable posar-se en el pitjor cas.
- Els desbordaments de memòria són habituals, per això s'ha de provar d'introduir valors tan grans com sigui possible.

Exemple de prova de capsa negra

Tot seguit es mostra un exemple de prova de capsa negra. S'ha de dur a terme el procés de prova del següent procediment:

```
1    Funció Buscar (DNI as string, vectMatricula de Matricules)
         retorna Matricula
```

En concret, en aquesta funció se li proporciona un string que representa el DNI de l'alumne, i un vector anomenat `vectMatricula` que emmagatzema les matrícules dels alumnes. La funció busca en el vector `vectMatricula` la matrícula de l'alumne. La funció retorna la matrícula de l'alumne si la troba o una matrícula buida si no la troba.

Per tal de simplificar el joc de proves, el nombre de matrícules que admet la funció Buscar és 10.

Per a la variable DNI hem de tenir en consideració que està formada de vuit xifres numèriques i una lletra(en aquest exercici no es valida el valor de la lletra):

- 00000001A Prova vàlida.
- Null Prova invàlida, el DNI no té valor.
- 00000001 Prova invàlida, el DNI té un format incorrecte, falta la lletra.
- 00000001AA Prova invàlida, el DNI té un format incorrecte.
- AAAAAAAAA Prova invàlida, el DNI té un format incorrecte.

Pel vector `vectMatricula` hem de tenir en consideració que pot contenir de 0 a 10 matrícules:

- `[]`. Prova vàlida, vector buit.
- `[matrícula1]`. Prova vàlida, vector amb una matrícula.
- `[matrícula1, matrícula2, matrícula3, ... matrícula10]`. Prova vàlida, vector amb un nombre de matrícules entre 0 i 10.
- `[matrícula1, matrícula2, matrícula3, ... matrícula10, matrícula11]`. Prova invàlida, vector amb un nombre de matrícules superior a 10.

Per a la sortida:

- `Alumne.DNI = DNI`. Prova vàlida. Classe amb les dades d'un alumne.
- Classe buida. Prova vàlida. S'ha buscat el DNI d'una persona que no és alumne.
- `Alumne.DNI <> DNI`. Prova invàlida. Classe amb les dades d'un altre alumne.
- Classe buida. Prova invàlida. S'ha buscat el DNI d'un alumne i no s'ha trobat.

Ús d'interfície gràfica

No tan sols s'ha de parlar d'entrades de textos, també cal tenir en compte els entorns gràfics on es duen a terme les entrades de valors o on es visualitzen els resultats.

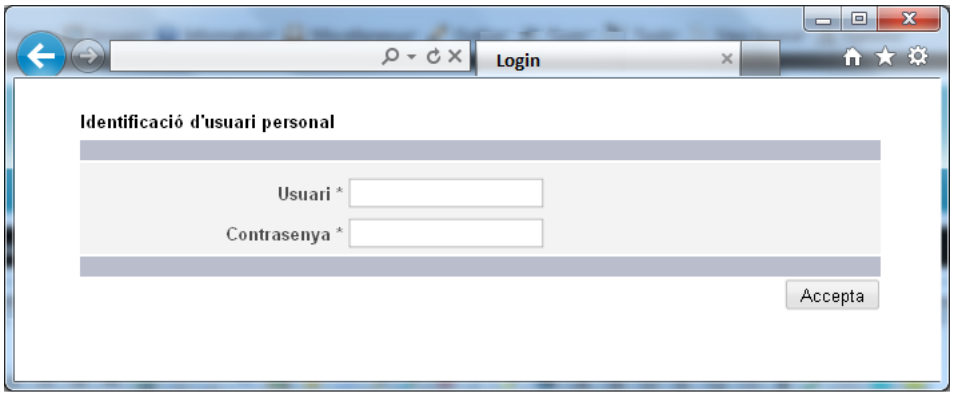
Actualment, la majoria de programes solen interactuar amb l'usuari fent ús de sistemes gràfics que cada vegada són més complexos, amb la qual cosa es poden generar errors.

Les proves d'interfície gràfica d'usuari han d'incloure:

- Proves sobre finestres: icones de tancar, minimitzar...
- Proves sobre menús i ús de ratolí.
- Proves d'entrada de dades: quadre de textos, llistes desplegables...
- Proves de documentació i ajuda del programa.
- Altres.

A la figura 1.9 es mostrauna finestra que controla l'accés al sistema de matriculació dels alumnes mitjançant la introducció d'un nom d'usuari i una clau (*password*). El sistema comprova si hi ha un compte amb el nom i clau especificat i, si és així, es dóna permís per entrar. Si hi ha un compte amb aquest nom i la clau és incorrecta, permet tornar a introduir la clau fins a un màxim de tres vegades.

Figura 1.9. Pantalla per al control d'entrada de l'identificador de l'alumne i de la clau per poder efectuar la matrícula



Tot seguit es mostren alguns dels casos de prova que es podrien utilitzar amb aquest programa:

- **Cas de prova 1:**
  - Entrada: usuari correcte i contrasenya correcta. Prémer botó d'accedir al sistema.
  - Condicions d'execució: en la taula existeix aquest usuari amb la contrasenya i amb un intent fallat (nombre inferior a 3).

  - Resultat esperat: donar accés al sistema i reflectir que el nombre d'intents per a l'usuari correcte és zero en la taula USUARI (compte, contrasenya, nre. intents).

- **Cas de prova 2:**

  - Entrada: usuari incorrecte i contrasenya correcta. Prémer botó d'accedir al sistema.
  - Condicions d'execució: en la taula no existeix aquest usuari amb aquesta contrasenya.
  - Resultat esperat: no donar accés al sistema.

Proves d'integració

Són suficients les proves que es fan a cada part d'una aplicació per assegurar-nos que s'ha validat el funcionament del programari? La resposta és no; és necessari validar també els diferents mòduls combinats.

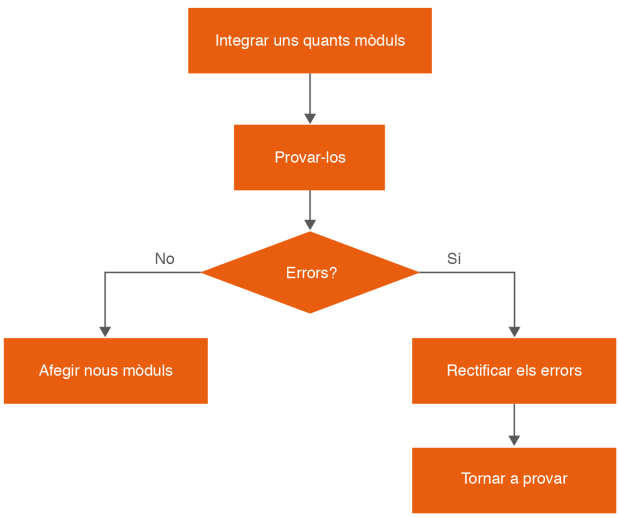
Proves d'integració

Una vegada s'han provat els components individuals del programa i s'ha garantit que no contenen errors, caldrà integrar-los per tal de crear un sistema complet que també haurà de ser provat. Aquest és el nivell de les proves d'integració.

Un **objectiu** important de les proves d'integració és localitzar errors en les interfícies entre les diferents unitats. A més, les proves d'integració serveixen per validar que les parts de codi que ja han estat provades de forma independent continuïn funcionant correctament en ser integrades.

Els elements no s'integren tots al mateix temps, sinó que s'utilitzen diferents estratègies d'integració incremental, que, bàsicament, consisteixen en el que es mostra en el flux de la figura 1.10.

FIGURA 1.10. Esquema d'integració incremental.



Amb aquest procés es facilita la localització de l'error quan es produeixi, perquè se sap quins són els últims mòduls que s'han integrat i quan s'ha produït l'error.

L'organització clàssica dels mòduls és una estructura jeràrquica organitzada per nivells: a la part alta hi haurà el mòdul o mòduls principals (a vegades denominats pares), que fan crides a mòduls subordinats de nivell inferior (fills), i així successivament cada nivell utilitzarà mòduls de nivell inferior fins arribar als mòduls terminals. Els mòduls superiors seran els més propers a l'usuari, és a dir, inclouen la interfície d'usuari (entorn gràfic, menús, ajudes...), i els mòduls inferiors són els més propers a l'estructura física de l'aplicació (bases de dades, maquinari...).

Existeixen diferents estratègies de desenvolupament de proves d'integració, com són les proves d'integració ascendent i les proves d'integració incrementals descendents.

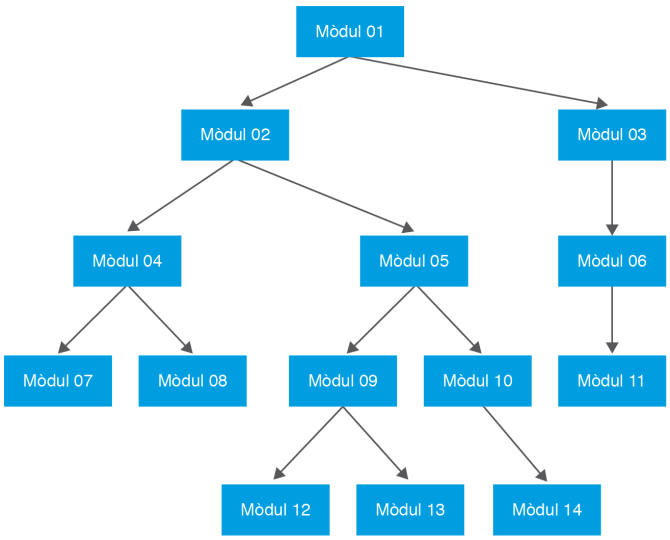
Prova d'integració ascendent

Aquesta estratègia de desenvolupament de les proves d'integració començarà pels mòduls finals, els mòduls de més baix nivell, agrupant-los per les seves funcionalitats. Es crearà un mòdul impulsor que anirà efectuant crides als diferents mòduls a partir de les precondicions indicades i recollint els resultats de cada crida a cada funció.

A mesura que els resultats d'aquestes proves vagin sortint positius, s'anirà escalant per l'arbre de jerarquies amb el mòdul impulsor cap als altres mòduls, fent les crides pertinents de forma recursiva. La darrera prova serà una crida al programari sencer amb els valors d'entrada reals (analitzant els valors també reals de sortida).

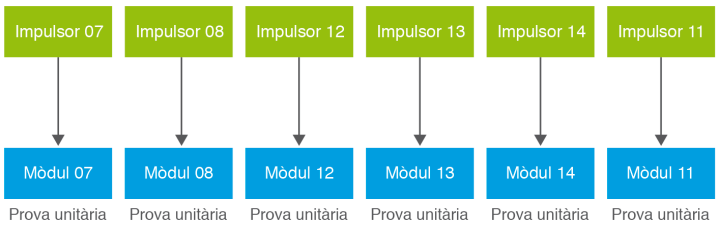
A continuació, es descriu el procés seguit per un sistema d'informació que té una estructura com la que es mostra en la figura 1.11:

Figura 1.11. Esquema de proves d'integració ascendents



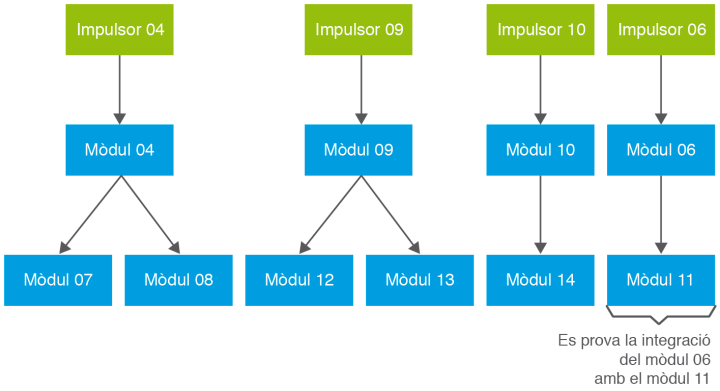
A la figura 1.12 es pot observar la primera fase de les proves d'integració ascendents. Cada mòdul ha de ser provat per separat, per això s'ha de construir un mòdul impulsor independent per provar cada mòdul.

Figura 1.12. Integració incremental ascendent, fase 1



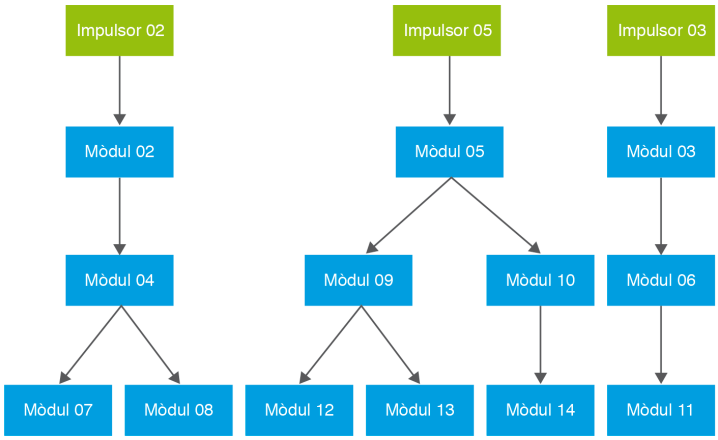
La figura 1.13 mostra la següent fase, una vegada finalitzades les proves sobre els mòduls de nivell més baix, els mòduls (07, 08, 12, 13, 14 i 11). El següent pas és continuar amb els mòduls del següent nivell. Però això implica crear nous mòduls impulsors (04, 09, 10 i 06), que s'aplicaran a aquests mòduls, els quals s'integraran amb els mòduls de nivell més baix anteriorment provats (07, 08, 12, 13, 14 i 11).

Figura 1.13. Integració incremental ascendent, fase 2



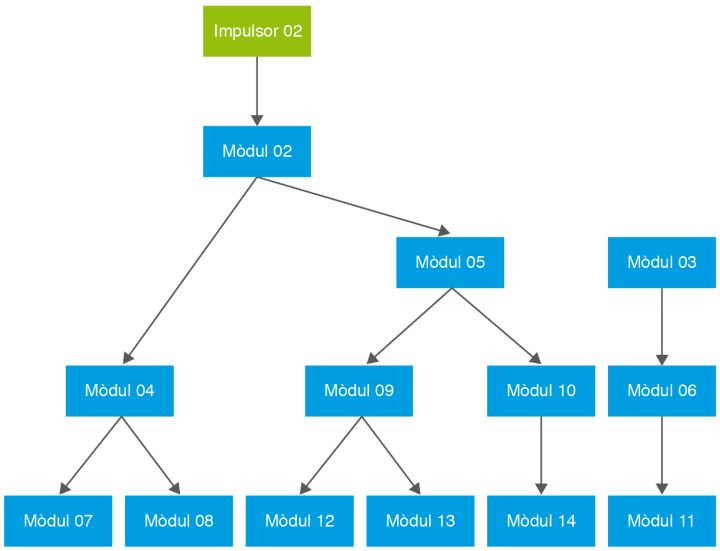
La figura 1.14 mostra un nivell més d'aquesta estratègia, arribant als mòduls 02, 05 i 03.

Figura 1.14. Integració incremental ascendent, fase 3



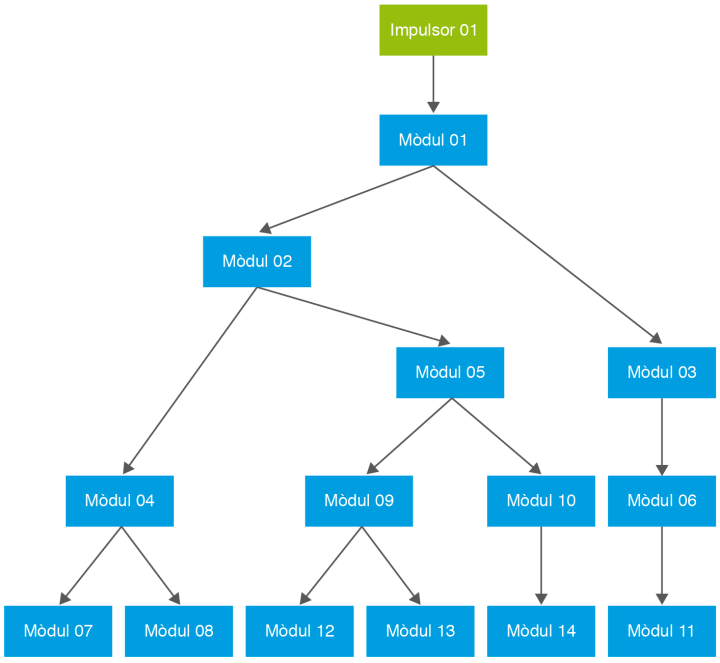
A la figura 1.15 es veu la integració dels mòduls 04 i 05 amb el mòdul 02, per al qual s'haurà de crear l'impulsor 02, que cridarà a aquest mòdul.

Figura 1.15. Integració incremental ascendent, fase 4



A la figura 1.16 es veu com s'arriba al final d'aquest exemple d'integració incremental ascendent, fins al mòdul 01, que integra els mòduls 02 i 03. També caldrà crear un impulsor 01 per a la crida d'aquest mòdul.

Figura 1.16. Integració incremental ascendent, fase 5



Adaptant l'exemple que es va tractant en els punts anteriors a les proves d'integració, si es volguessin efectuar les proves del procés de matriculació dels alumnes en un centre universitari, es podria començar pels mòduls que fan canvis en la

base de dades o en l'XML on s'emmagatzema la informació. Una vegada que cada un d'aquests mòduls funciona correctament, s'inicien les proves dels mòduls de nivell superior, que bàsicament fan crides a aquests mòduls de nivell més baix (mòduls que podrien tenir la lògica de negoci). De forma progressiva, s'aniran incorporant nous mòduls fins a provar tot el sistema.

Avantatges de la integració incremental ascendent
Els avantatges són els següents:

- Ordre adequat: primer s'avaluen els mòduls inferiors, que són els que acostumen a tenir el processament més complex, se'n solucionen els errors, i després es nodreix de dades la resta del sistema.
- Més senzillesa: les entrades per a les proves són més fàcils de crear, ja que els mòduls inferiors solen tenir funcions més específiques.
- Millor observació dels resultats de les proves: com que es comença pels mòduls inferiors, és més fàcil l'observació dels resultats de les proves.

Desavantatges de la integració incremental ascendent
Entre els desavantatges, cal destacar:

- Anàlisi parcial: fins que no es fa la crida al darrer mòdul no es valida el sistema com a tal.
- Alt temps de dedicació: caldrà dedicar molt de temps a implementar cada mòdul impulsor, que poden arribar a ser molts.

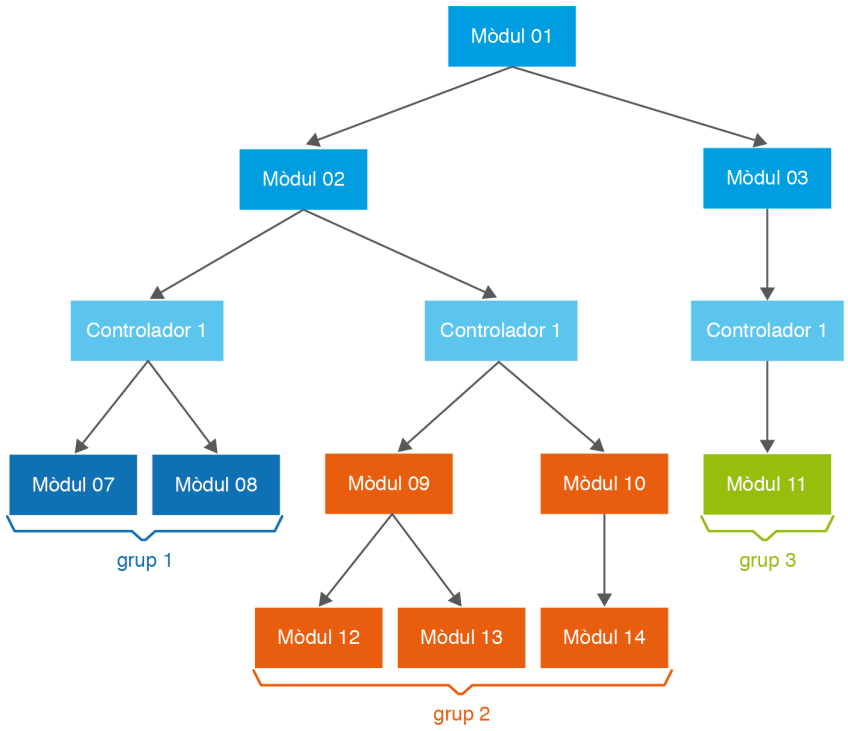
Prova d'integració incremental descendent
Aquesta estratègia de desenvolupament de les proves d'integració començarà pel mòdul de control principal (el més important, el de més nivell). Una vegada validat, s'aniran integrant els altres mòduls que en depenen de forma progressiva, sense seguir una estratègia concreta, només tenint en compte que el nou mòdul incorporat a les proves tindrà ja validats tots els mòduls que el referencien. En funció del tipus de mòduls i del tipus de projecte, s'escollirà una seqüència o una altra a l'hora d'anar integrant mòduls, analitzant el problema concret. Les etapes de la integració descendent són:

- Se selecciona el mòdul més important, el de major nivell. Aquest mòdul farà d'impulsor. Caldrà escriure altres mòduls ficticis que simulin els mòduls que cridarà el principal.
- A mesura que es van integrant mòduls, caldrà provar-los independentment i de forma conjunta amb els altres mòduls ja provats. Una vegada s'ha finalitzat la prova, se substitueix el mòdul fictici creat pel real que s'ha integrat.

- Llavors caldrà escriure els mòduls ficticis subordinats que es necessitin per a la prova del nou mòdul incorporat.

A la figura 1.17 es pot observar un esquema a partir del qual es duran a terme les proves d'integració incremental descendent. En primer lloc, es combinaran els mòduls per formar els grups 1, 2 i 3. Sobre cada grup s'hauran de dur a terme les proves mitjançant un controlador. Els mòduls dels grups 1 i 2 són subordinats del mòdul 02. Igualment, s'haurà d'eliminar el controlador 3 del grup 3 abans de la integració amb el mòdul 03. A continuació, s'integrarà el mòdul 01 amb el mòdul 02 i el mòdul 03. Aquestes accions s'aniran repetint de forma recursiva al llarg de tota l'estructura del projecte.

Figura 1.17. Prova d'integració incremental descendent



Avantatges de la integració descendent

Els avantatges són els següents:

- Identificació de l'estructura: permet veure l'estructura del sistema des d'un principi, facilitant l'elaboració de demostracions del seu funcionament.

- Disseny descendent: primer es defineixen les interfícies dels diferents subsistemes per després seguir amb les funcions específiques de cada un per separat.

- Detecció més ràpida dels errors que es trobin als mòduls superiors pel fet de detectar-se en una etapa inicial.

Desavantatges de la integració descendent
Entre els desavantatges, destaca:

- Cost molt elevat: caldrà implementar molts mòduls addicionals per oferir els mòduls ficticis a fi d'anar efectuant les proves.

- Alta dificultat: en voler fer una distribució de les proves del més genèric al més detallat, les dades que s'hauran d'utilitzar són difícils d'aconseguir, ja que són els mòduls de nivell més baix els que tindran els detalls.

Proves de càrrega i acceptació

El pas següent, una vegada fetes les proves unitàries i les proves d'integració, serà dur a terme primer les proves de càrrega i, posteriorment, les proves d'acceptació.

Les **proves de càrrega** són proves que tenen com a objectiu comprovar el rendiment i la integritat de l'aplicació ja acabada amb dades reals. Es tracta de simular l'entorn d'explotació de l'aplicació.

Amb les proves anteriors (unitàries i d'integració) quedaria provada l'aplicació a escala de "laboratori". Però també es necessita comprovar la resposta de l'aplicació en situacions reals, i fins i tot, en situacions de sobrecàrrega, tant a escala de rendiment com de descontrol de dades.

Per exemple, una aplicació lenta pot ser poc operativa i no útil per a l'usuari. Després de les proves de càrrega, es troben les proves d'acceptació. Aquestes proves les fa el client o usuari final de l'aplicació desenvolupada. Són bàsicament proves funcionals, sobre el sistema complet, i busquen una cobertura de l'especificació de requisits i del manual de l'usuari. Aquestes proves no es fan durant el desenvolupament, ja que seria impresentable de cara al client, sinó una vegada passades totes les proves anteriors per part del desenvolupador o l'equip de tests.

Els programadors acostumen a obtenir sorpreses en les proves d'acceptació, ja que és la primera vegada que es troben amb el programa finalitzat.

L'objectiu de la **prova d'acceptació** és obtenir l'aprovació del client sobre la qualitat de funcionament del sistema desenvolupat i provat.

L'experiència demostra que, encara després del procés més acurat de proves per part del desenvolupador i l'equip de treball, queden una sèrie d'errors que només apareixen quan el client posa en funcionament l'aplicació o el sistema desenvolupat.

Sigui com sigui, el client sempre té la raó. Per aquest motiu, molts desenvolupadors exerciten unes tècniques denominades **proves alfa i proves beta**.

Les proves alfa consisteixen a convidar el client que vingui a l'entorn de desenvolupament a provar el sistema. Es treballa en un entorn controlat i el client sempre té un expert a mà per ajudar-lo a usar el sistema i per analitzar els resultats.

Les proves beta vénen després de les proves alfa, i es desenvolupen en l'entorn del client, un entorn que és fora de control per al desenvolupador i l'equip de treball. Aquí el client es queda tot sol amb el producte i tracta de trobar els errors, dels quals informarà el desenvolupador.

Les proves alfa i beta són habituals en productes que es vendran a molts clients o que faran servir molts usuaris. Alguns dels compradors potencials es presten a aquestes proves, sia per anar entrenant el seu personal amb temps, sia en compensació d'algun avantatge econòmic (millor preu sobre el producte acabat, dret a manteniment gratuït, a noves versions...).

L'experiència mostra que aquestes pràctiques són molt eficaces. En un entorn de desenvolupament de programari tenen sentit quan l'aplicació o sistema per desenvolupar el farà servir un gran nombre d'usuaris finals (empreses grans amb diferents departaments que hauran d'utilitzar aquesta nova eina).

Proves de sistema i de seguretat

Les proves de sistema són aquelles proves que es duran a terme una vegada finalitzades les proves unitàries (cada mòdul per separat), les proves d'integració dels mòduls, les proves de càrrega i les proves d'acceptació per part de l'usuari.

Temporalment, es troben en una situació en la qual l'usuari ha pogut verificar l'aplicació desenvolupada duent a terme les proves d'acceptació. Posteriorment, l'aplicació s'ha integrat al seu nou entorn de treball.

Les **proves de sistema** serviran per validar l'aplicació una vegada aquesta hagi estat integrada amb la resta del sistema de l'usuari. Encara que l'aplicació ja hagi estat validada de forma independent, a les proves de sistema es durà a terme una segona validació amb l'aplicació ja integrada en el seu entorn de treball real.

Tot seguit s'enumeren alguns tipus de proves per desenvolupar durant les proves del sistema:

- **Proves de rendiment:** valoraran els temps de resposta de la nova aplicació, l'espai que ocuparà en disc, el flux de dades que generarà a través d'un canal de comunicació.
- **Proves de resistència:** valoraran la resistència de l'aplicació per a determinades situacions del sistema.
- **Proves de robustesa:** valoraran la capacitat de l'aplicació per suportar diverses entrades no correctes.

- **Proves de seguretat:** ajudaran a determinar els nivells de permisos dels usuaris, les operacions que podran dur a terme i les d'accés al sistema i a les dades.
- **Proves d'usabilitat:** determinaran la qualitat de l'experiència d'un usuari en la manera d'interactuar amb el sistema.
- **Proves d'instal·lació:** indicaran les operacions d'arrencada i d'actualització dels programaris.

Les proves de sistema aglutinen molts tipus de proves que tindran diversos objectius:

- Observar si l'aplicació fa les funcions que ha de fer i si el nou sistema es comporta com ho hauria de fer.
- Observar els temps de resposta per a les diferents proves de rendiment, volum i sobrecàrrega.
- Observar la disponibilitat de les dades en el moment de recuperació d'una errada (a la vegada que la correctesa).
- Observar la usabilitat.
- Observar la instal·lació (assistents, operadors d'arrencada de l'aplicació, actualitzacions del programari...).
- Observar l'entorn una vegada l'aplicació està funcionant a ple rendiment (comunicacions, interaccions amb altres sistemes...).
- Observar el funcionament de tot el sistema a partir de les proves globals fetes.
- Observar la seguretat (el control d'accés i intrusions...).

Les proves a escala global del sistema s'han anat produint a mesura que es tenien funcionalitats perfectament acabades. És el cas, per exemple, de provar el funcionament correcte del desenvolupament d'una partida en un dels tres jocs, o la navegació correcta per les diferents pantalles dels menús.

Les **proves de validació** permeten comprovar si, efectivament, es compleixen els requisits proposats pel nostre sistema.

Proves de regressió i proves de fum

Les **proves de regressió** cerquen detectar possibles nous errors o problemes que puguin sortir en haver introduït canvis o millores en el programari.

Aquests canvis poden haver estat introduïts per solucionar algun problema detectat en la revisió del programari arran d'una prova unitària, d'integració o de sistema. Aquests canvis poden solucionar un problema però provocar-ne d'altres, sense haver-ho previst, en altres llocs del programari. És per aquesta raó que cal dur a terme les proves de regressió en finalitzar la resta de proves.

Un control no convenient dels canvis de versions, o una falta de consideració envers altres mòduls o parts del programari, poden ser causes dels problemes per detectar en les proves de regressió.

Es pot automatitzar la detecció d'aquest tipus d'errors amb l'ajuda d'eines específiques. L'automatització és complementària a la resta de proves, però en facilitarà la repetibilitat. El problema que es pot derivar de l'automatització de les proves de regressió és que demanaran un manteniment complex.

Proves "de fum"

El terme proves de fum sorgeix a partir de la fabricació de maquinari. Si després de reparar un component de maquinari, aquest "no treu fum", és que funcionarà correctament.

Les **proves "de fum"** es fan servir per descriure la validació dels canvis de codi en el programari, abans que els canvis en el codi es registrin en la documentació del projecte.
Són proves d'execució ràpida i comproven les funcions bàsiques del programari.

S'acostuma a dur a terme una revisió del codi abans d'executar les proves de fum. Aquesta revisió del codi es farà, sobretot, pel que fa als canvis que s'hagin produït en el codi.

1.3.3 Execució de les proves

Després de la planificació dels procediments de proves i del disseny dels casos de proves, el següent pas serà el procés d'execució. Aquests procés està representat en el diagrama de flux de la figura 1.18.

L'execució de les proves comportarà seguir els passos següents:

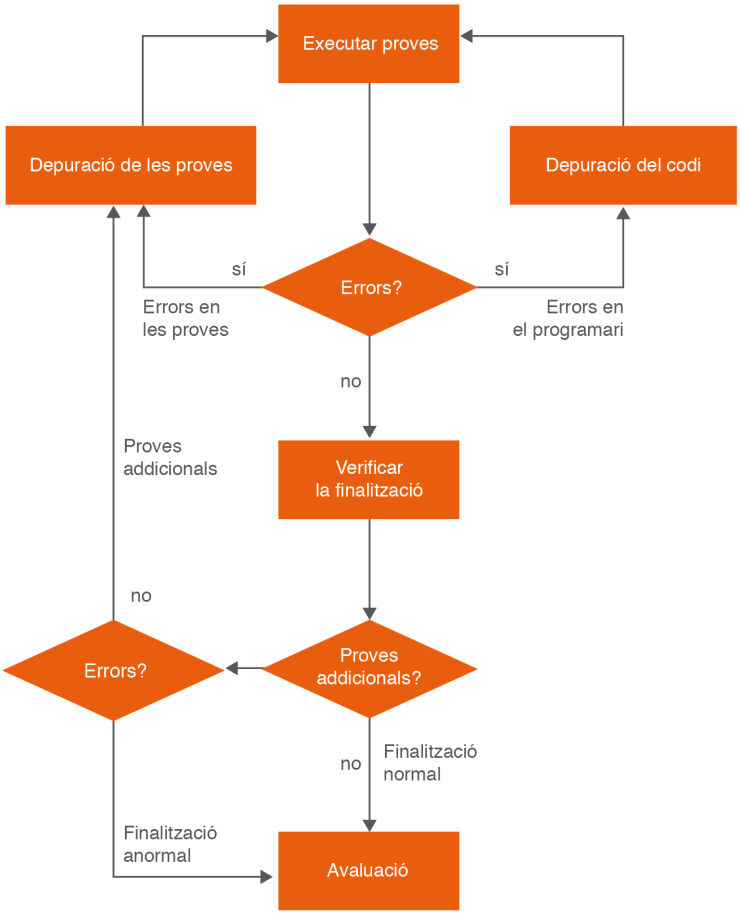
1. Execució de les proves.
2. Comprovació de si s'ha produït algun error en l'execució de les proves.
3. Si no hi ha hagut cap error:
    - Es verifica la finalització de les proves.
    - Es valida si calen proves addicionals.
        - Si es necessiten proves addicionals, cal validar que no existeixin condicions anormals. Si hi ha condicions anormals, es finalitza el procés de proves fent una avaluació del mateix procés; si no, caldrà depurar les proves.

– Si no es necessiten proves addicionals, es durà a terme una finalització del procés de proves fent una avaluació del mateix procés.

4. En el cas d'haver trobat errors en l'execució de les proves, s'haurà de veure si aquests errors han estat deguts a:

   - Un defecte del programari. En aquest cas, l'execució de les proves ha complert el seu objectiu i caldrà depurar el codi de programació, localitzar el o els errors i solucionar-ho per tornar al punt inicial, en què es tornaran a executar les proves i es tornarà a validar si el canvi efectuat ha estat exitós.
   - Un defecte del disseny de les proves. En aquest cas, caldrà revisar les proves que s'han executat, depurant-les, localitzant el o els errors i solucionar-ho, per tenir unes proves correctes, sense errors, llestes per tornar al punt inicial i tornar a executar-les.

FIGURA 1.18. Execució de les proves



Una execució de les proves exitosa no és aquella que troba errors costi el que costi ni aquella que només avalua dues o tres possibilitats, sinó que és aquella que acompleix el que s'ha planificat al pla de proves i que garanteix que allò dissenyat sigui el que es validi.

1.3.4 Finalització: avaluació i anàlisi d'errors

El darrer pas dels procediments de proves serà la finalització del procés. Per poder donar-lo per tancat de forma exitosa, caldrà efectuar una avaluació i una anàlisi dels errors localitzats, tractats, corregits i reavaluats.

Si no es pot donar per finalitzat el procés de proves, caldrà dur a terme una replanificació del procés de proves per establir noves depuracions i noves proves, planificant-les i dissenyant-les de nou.

> En el cas d'haver de refer els **procediments de proves**, és molt important la creació de nous casos de proves i no pas la readaptació dels ja existents. Si es creen nous casos de proves, s'estarà redissenyant els procediments de proves sobre el mateix codi font; si s'intenten readaptar els ja existents o modificar el codi, es corre el risc de fer que el codi font sigui el que s'estigui adaptant als casos de proves.

Finalment, és convenient escriure un informe que ajudi a emmagatzemar l'experiència que s'ha recollit al llarg del procediment de proves. Aquesta informació serà molt important per a futurs projectes, ja que ajudarà a no tornar a repetir els mateixos errors detectats. L'informe haurà de donar resposta a ítems com els que s'indiquen a continuació:

- Nombre de casos de prova generats.
- Nombre d'errors detectats a cada fase del projecte.
- Temps i recursos dedicats als procediments de proves.
- Tipus de proves dutes a terme.
- Tipus de proves que més errors han detectat.
- Nivell de qualitat del programari.
- Mòduls on més errors s'han detectat.
- Errors que han arribat fins als usuaris finals.
- Nombre de casos de prova erronis detectats.

1.3.5 Depuració del codi font

Les proves que s'efectuen sobre tot un projecte informàtic amb tots els processos involucrats (planificació, disseny, execució i avaluació) ajudaran a la detecció i correcció d'errors, intentant trobar-los al més aviat possible en el desenvolupament del projecte. Una tècnica molt important per a l'execució de les proves i, en general, per als programadors, és la depuració del codi font.

La **depuració del codi font** consisteix a anar executant pas a pas el codi font, observant els estats intermedis de les variables i les dades implicades per facilitar la correcció d'errors.

Els procediments que estan vinculats a la depuració del codi font són:

- Identificar la casuística per poder reproduir l'error.
- Diagnosticar el problema.
- Solucionar l'error atacant el problema.
- Verificar la correcció de l'error i verificar que no s'han introduït nous errors a la resta del programari.

La depuració del codi és una eina molt útil si se sap localitzar el mòdul o la part del codi font on es troba un determinat error. Si l'error és difícil de reproduir serà molt complicat trobar una solució per solucionar-lo. Per això, acotant-lo dintre del codi, es pot anar localitzant, fent proves per arribar a les circumstàncies que el reprodueixen.

En la secció *Activitats* del web del mòdul hi podeu trobar exemples de depuració de codi font.

Caldrà anar amb compte amb les solucions aplicades quan es fa servir la tècnica de la depuració del codi. Moltes vegades una modificació en el codi per solucionar un problema pot generar un altre error que hi estigui relacionat o que no hi tingui res a veure. Caldrà tornar a confirmar la correcta execució del codi una vegada finalitzada la correcció.

La depuració de codi permet la creació d'un punt en el codi font fins al qual el programari serà executat de forma directa. Quan l'execució hagi arribat a aquest punt de ruptura, es podrà avançar en l'execució del codi pas a pas fins a trobar l'error que es cerca.

Una altra forma de dur a terme la depuració de codi és anar enrere, des del punt de l'error, fins a trobar el causant. Aquesta tàctica es fa servir en casos més complexos i no és tant habitual, però serà molt útil en el cas de no tenir detectada la ubicació de l'error, que pot trobar-se ocult en alguna part del codi font. En aquests casos, a partir del lloc on es genera l'error, es durà a terme un recorregut cap enrere, anant pas a pas en sentit invers.

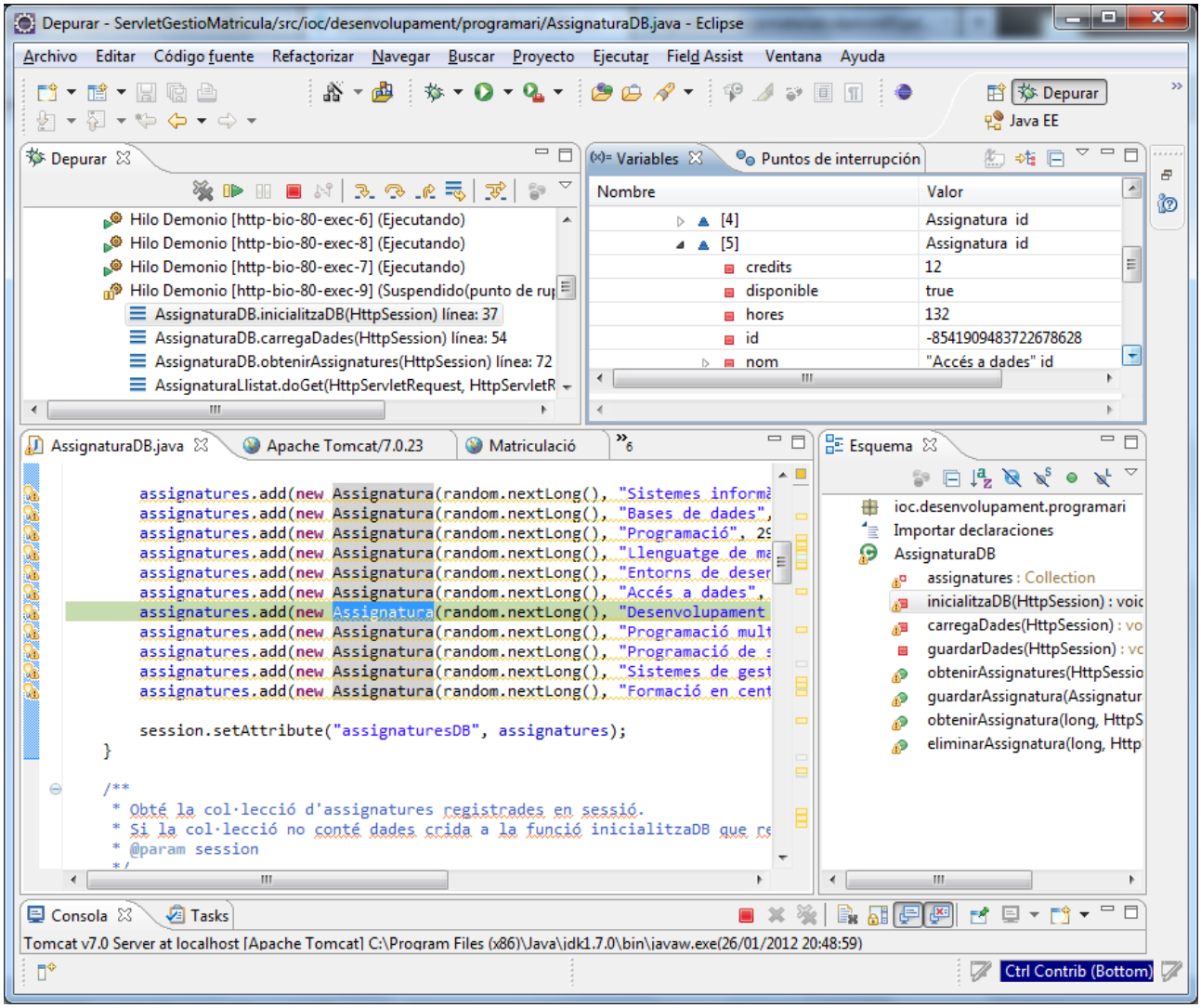
Una eina que ajuda a la depuració del codi font és la utilització de fitxers anomenats registres (*logs*). Aquests fitxers, habitualment de text, enregistren tota la informació vinculada a l'execució d'un programari amb l'objectiu que el programador pugui fer una revisió pas a pas de com ha evolucionat aquesta execució i localitzar errors o mals funcionaments.

La **depuració del codi** és força útil també en l'execució de les proves d'integració, de sistema i d'acceptació.

Eclipse proporciona un entorn de depuració que facilita la detecció d'errors, permetent seguir el codi pas a pas i consultar els valors que van prenent les dades.

A la figura 1.19 es mostra l'entorn de treball de l'eina Eclipse a l'hora de dur a terme les accions de depuració.

FIGURA 1.19. Eclipse: perspectiva de depuració



1.4 Eines per a la realització de proves

En la secció *Activitats* del web del mòdul hi podeu trobar exemples d'eines d'ajuda per a la realització de proves de programari.

Les eines informàtiques per a la realització de proves ajudaran molt a poder automatitzar la tasca d'executar les proves i alguns dels altres processos que cal fer per a implementar-les (planificació, disseny...). Tal com succeeix amb les eines específiques per al desenvolupament de programari, també existeixen eines específiques per a l'ajuda als procediments de proves.

Això sí, com tot, l'ús d'aquestes eines tindrà alguns punts positius, però també comportarà alguns punts dèbils.

1.4.1 Beneficis i problemes de l'ús d'eines de proves

L'ajuda que aporten les eines per a la realització de proves és la d'automatitzar una tasca que moltes vegades és molt repetitiva i pot arribar a reclamar molt de temps als implicats en cas de fer-ho de forma manual. A més, a mesura que es van desenvolupant les proves de forma manual, amb el pas del temps, hi ha més risc que es produeixin errors humans per part dels desenvolupadors o dels verificadors.

La utilització d'una eina que automatitzi les proves ofereix la possibilitat de generar els casos de proves, executar-los i comparar els resultats obtinguts amb els resultats esperats. Moltes vegades, tantes dades tan semblants afavoreixen l'aparició d'errors, que amb les eines informàtiques es poden minimitzar. A més, són especialment útils per confirmar que un error s'ha solucionat.

Les eines d'ajuda a l'elaboració de proves ofereixen, també com a punts forts, una contraposició a la repetició dels errors a l'hora de desenvolupar les proves. Si un mateix programador ha desenvolupat el codi font i és l'encarregat de generar els casos de prova podrà repetir, inconscientment, els errors que ha comès una vegada. Si es fa servir un programari per desenvolupar un projecte, i es volen generar les proves amb el mateix programari, aquest mateix programari les automatitzarà de tal manera que no es repeteixin els possibles errors de programació. Aquesta consistència i garantia a l'hora de dur a terme les proves és més difícil de ser mostrada per un ésser humà.

Un altre punt fort de les eines d'automatització de les proves són les funcionalitats que ofereixen a tall de resum i que permeten tenir més informació tant de les proves com del codi desenvolupat. Serà igual d'important la correcta realització de les proves com la informació que se'n pot extreure i la forma d'accedir-hi. Una eina de gestió de proves pot oferir informes estadístics sobre els resultats de les proves, sobre els diferents mòduls avaluats, sobre els resultats, sobre les parts del codi font més fiables i les que no... Tota aquesta informació, presentada d'una forma adient, facilitarà la presa de decisions i la resolució de problemes.

Com a punts febles en la utilització d'eines d'automatització de les proves es poden trobar:

- Temps que cal dedicar a aprendre a fer servir correctament aquest programari. De vegades es necessari dedicar tant de temps a conèixer a consciència una eina informàtica com el que es dedicaria a efectuar les proves de forma manual. Cada aplicació informàtica té les seves característiques i les seves especificacions a l'hora de ser utilitzada. S'hauran de conèixer bé per treure'n el màxim profit.
- Si no hi ha una bona configuració i una bona selecció de les proves, els resultats obtinguts de les eines en la realització de les proves tampoc no seran fiables i es podrien donar per bons uns resultats que no ho són. Les aplicacions són fiables si es saben utilitzar correctament.
- Dins el projecte, serà recomanable tenir una persona que es dediqui de forma exclusiva a aquesta tasca.

1.4.2 Algunes eines de proves de programari

Les eines de proves del programari es poden classificar segons molts criteris: en funció del o dels llenguatges de programació a què es dóna suport, en funció de si són privatius o de codi obert, o en funció, per exemple, del tipus de proves que permeten dur a terme.

S'ha de considerar que la gran majoria dels entorns integrats de desenvolupament porten integrades eines que permeten la depuració del codi font. Algunes d'elles també permeten el desenvolupament de proves o el fet d'afegir algun mòdul que ho permeti.

En els materials Web, a l'apartat d'Activitats, es pot trobar una activitat de proves unitàries desenvolupat amb JUnit.

Tot seguit s'enumeren algunes eines que permeten el desenvolupament de proves en funció del tipus de proves:

- Proves unitàries:
  - **JUnit**. Automatitza les proves unitàries i d'integració. Proveeix classes i mètodes que faciliten la tasca d'efectuar proves en el sistema per tal d'assegurar la consistència i funcionalitat del programari desenvolupat.
- Proves estàtiques de codi:
  - **PMD**. Pot ser integrat a diverses eines: JDeveloper, Eclipse, jEdit, etc. Permet trobar en el codi errors en el maneig d'excepcions, codi mort, codi sense optimitzar, codi duplicat...
  - **FindBugs**. Pot integrar-se a Eclipse. Efectua un escaneig de codi per mitjà del qual troba errors comuns, males pràctiques de programació, codi vulnerable, rendiment, seguretat...
  - **YASCA**. Permet trobar vulnerabilitats de seguretat, qualitat en el codi, rendiment... Aprofita la funcionalitat dels connectors FindBugs, PMD i Jlint.
- Proves de rendiment:
  - **JMeter**. Permet efectuar proves de rendiment, d'estrès, de càrrega i de volum, sobre recursos estàtics o dinàmics.
  - **OpenSTA**. Permet captar les peticions de l'usuari generades en un navegador web, després guardar-les, i poder-les editar per al seu posterior ús.
  - **WEbLoad**. permet dur a terme proves de rendiment, a través d'un entorn gràfic en el qual es poden desenvolupar, gravar i editar *script* de proves.
  - **Grinder**. és un *framework* (entorn de treball) escrit en Java, amb el qual es poden efectuar proves de rendiment, per mitjà d'*script* escrits en llenguatge Jython. Permet gravar les peticions del client sobre un navegador web per ser després reproduït.

- Proves d'acceptació:
  - **Fitness**. Permet comparar el que ha de fer el programari amb el que realment fa. Es poden efectuar proves d'acceptació i proves de regles de negoci.
  - **Avignon**. Permet als usuaris expressar proves d'acceptació d'una forma no ambigua abans que comenci el desenvolupament. Treballa en conjunt amb JUnit, HTTPUnit...

2. Eines per al control i documentació de programari

Què és més important, dedicar el menor temps possible en el desenvolupament d'una aplicació informàtica (i estalviar-nos tot el cost possible d'un programador), o bé desenvolupar la mateixa aplicació amb un codi font molt més optimitzat i preparat per a futurs canvis (havent-hi dedicat més esforç)?

En el procés de desenvolupament de programari és molt important tenir en compte certes directrius molt recomanables que, d'altra banda, moltes vegades no se segueixen per falta de cultura i per la idea equivocada que el fet de seguir-les elevarà els costos a nivell de temps.

Què costa tenir un programari desenvolupat de forma òptima? Si un programari fa el que ha de fer, és eficaç i és eficient, per què cal que sigui òptim? A més, si el desenvolupament d'un programari fet amb presses pot estalviar temps de programadors, i el temps sempre és valorable en diners, per què s'ha d'invertir a desenvolupar de forma òptima?

Les raons són moltes. Sempre costarà el mateix fer les coses mal fetes que fer-les ben fetes, si t'has acostumat a fer-les ben fetes des d'un principi i ho has après així. En un futur, el manteniment i les possibles ampliacions del programari seran molt més costoses si has intentat estalviar en temps abans.

Què significa programar de forma òptima? Hi ha moltes coses a tenir en compte i es poden trobar molts consells al respecte.

2.1 Refacció

En desenvolupar una aplicació cal tenir molt presents alguns aspectes del codi de programació que s'anirà implementant. Hi ha petits aspectes que permetran que aquest codi sigui considerat més òptim o que facilitaran el seu manteniment. Per exemple, un d'aquests aspectes serà la utilització de constants. Si hi ha un valor que es farà servir diverses vegades al llarg d'un determinat programa, és millor utilitzar una constant que contingui aquest valor, d'aquesta manera, si el valor canvia només s'haurà de modificar la definició de la constant i no caldrà anar-lo cercant per tot el codi desenvolupat ni recordar quantes vegades s'ha fet servir.

A continuació, es mostra un exemple molt senzill per entendre a què es fa referència quan es parla d'optimitzar el codi font:

```
1 Class CalculCostos {
2     Public static double CostTreballadors (double NreTreballadors)
3     {
4   Return 1200 * NreTreballadors;
5     }
6 }
```

Al codi anterior es mostra un exemple de com seria la codificació d'una classe que té com a funció el càlcul dels costos laborals totals d'una empresa. Es pot veure que el cost per treballador no es troba a cap variable ni a cap constant, sinó que el mètode CostTreballadors retorna el valor que ha rebut per paràmetre (`NreTreballadors`) per un nombre que considera el salari brut per treballador (en aquest cas suposat 1200 euros). Probablement, aquest import es farà servir a més classes al llarg del codi de programació desenvolupat o, com a mínim, més vegades dins la mateixa classe.

Com quedaria el codi una vegada aplicada la refacció? Es pot veure a continuació:

```
Class CalculCostos {
    final double SALARI_BRUT= 1200;
    Public static double CostTreballadors (double NreTreballadors)
    {
      Return SALARI_BRUT* NreTreballadors;
    }
}
```

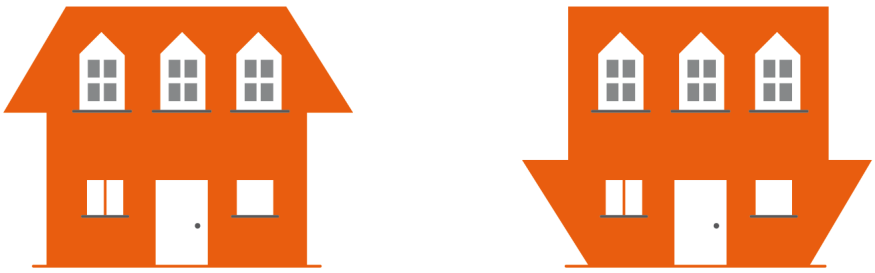
Aquest és un exemple del que s'entén per refacció.

> El terme *refacció* fa referència als canvis efectuats al codi de programació desenvolupat, sense implicar cap canvi en els resultats de la seva execució. És a dir, es transforma el codi font mantenint intacte el seu comportament, aplicant els canvis només en la forma de programar o en l'estructura del codi font, cercant la seva optimització.

El terme refacció va ser utilitzat per primera vegada per William F. Opdyke a la seva tesi doctoral, l'any 1992, a la Universitat d'Illinois.

A la figura 2.1 es pot veure un exemple del que es vol expressar. Quina de les dues cases facilita més la vida dels seus inquilins? Si s'hagués de desenvolupar una aplicació informàtica, a quina de les dues cases s'hauria d'assemblar més, a la de l'esquerra o a la de la dreta?

Figura 2.1. Disseny d'una casa



Possiblement les dues cases compleixen les especificacions inicials, edifici en què es pugui viure, però sembla que la primera casa serà més confortable i més òptima.

El terme refacció prové del terme refactoritzar (*refactoring*). Aquest terme esdevé de la seva similitud amb el concepte de factorització dels nombres o dels polinomis. És el mateix tenir 12 que tenir 3×4, però, en el segon cas, el terme 12 està dividit en els seus factors (encara es podria factoritzar més i arribar al $3 \times 2 \times 2$). Un altre exemple: el polinomi de segon grau $x^2 - 1$ és el mateix que el resultat del producte $(x + 1)(x — 1)$, però en el segon cas s'ha dividit en factors. A l'hora de simplificar o de fer operacions, serà molt més senzill el treball amb el segon cas (amb els termes ja factoritzats) que amb el primer. Amb la factorització apareixen uns termes, uns valors, que inicialment es troben ocults, encara que formen part del concepte inicial.

En el cas de la programació succeeix una situació molt similar. Si bé el codi que es desenvolupa no està factoritzat, és a dir, no se'n veuen a simple vista els factors interns, perquè són estructures que aparentment es troben amagades, quan es du a terme una refacció del codi font sí que es poden veure.

2.1.1 Avantatges i limitacions de la refacció

La utilització de la refacció pot aportar alguns avantatges als desenvolupadors de programari, però cal tenir en compte que té limitacions que cal conèixer abans de prendre la decisió d'utilitzar-la.

Avantatges de la refacció

Hi ha molts avantatges en la utilització de la refacció, tot i que també hi ha inconvenients i algunes limitacions. Per què els programadors dediquen temps a la refacció del codi font? Una de les respostes a aquesta pregunta és l'augment de la qualitat del codi font. Un codi font sobre el qual s'han utilitzat tècniques de refacció es mantindrà en un estat millor que un codi font sobre el qual no s'hagin aplicat. A mesura que un codi font original s'ha anat modificant, ampliant o mantenint, haurà patit modificacions en l'estructura bàsica sobre la qual es va dissenyar, i és cada vegada més difícil efectuar evolutius i augmenta la probabilitat de generar errors.

Alguns dels avantatges o raons per utilitzar la tècnica de refacció del codi font són:

- Prevenció de l'aparició de problemes habituals a partir dels canvis provocats pels manteniments de les aplicacions.
- Ajuda a augmentar la simplicitat del disseny.
- Major enteniment de les estructures de programació.
- Detecció més senzilla d'errors.
- Permet agilitzar la programació.
- Genera satisfacció en els desenvolupadors.

A continuació, es desenvolupen alguns d'aquests punts forts de la utilització de la refacció:

- **Detecció i prevenció més senzilla d'errors**. La refacció millora la robustesa del codi font desenvolupat, fent que sigui més senzill trobar errors en el codi o trobar parts del codi que siguin més propenses a tenir o provocar errors en el conjunt del programari. A partir de la utilització de casos de prova adequats, es podrà millorar molt el codi font.
- **Prevenció de problemes per culpa dels manteniments del programari**. Amb el temps, acostumen a sorgir problemes a mesura que es va aplicant un manteniment evolutiu o un manteniment correctiu de les aplicacions informàtiques. Alguns exemples d'aquests problemes poden ser que el codi font es torni més complex d'entendre del que seria necessari o que hi hagi duplicitat de codi, pel fet que, moltes vegades, són persones diferents les que han desenvolupat el codi de les que estan duent a terme el manteniment.
- **Comprensió del codi font i simplicitat del disseny**. Tornant a la situació en què un equip de programació pot estar compost per un nombre determinat de persones diferents o que el departament de manteniment d'una empresa és diferent a l'equip de desenvolupament de nous projectes, és molt important que el codi font sigui molt fàcil d'entendre i que el disseny de la solució hagi estat creat amb una simplicitat considerable. Cal que el disseny es dugui a terme tenint en compte que es farà una posterior refacció, és a dir, tenint presents possibles necessitats futures de l'aplicació que s'està creant. Aquesta tasca no és gens senzilla, però amb un bon i exhaustiu anàlisi, per mitjà de moltes converses amb els usuaris finals, es podran arribar a entreveure aquestes necessitats futures.
- **Programació més ràpida**. Precisament, si el codi es comprèn d'una forma ràpida i senzilla, l'evolució de la programació serà molt més ràpida i eficaç. El disseny dut a terme a la fase anterior també serà decisiu en el fet que la programació sigui més àgil.

Limitacions de la refacció

En canvi, es poden trobar diverses raons per no considerar adient la seva utilització, ja sigui per les seves limitacions o per les possibles problemàtiques que poden sorgir:

- Personal poc preparat per utilitzar les tècniques de la refacció.
- Excés d'obsessió per aconseguir el millor codi font.
- Excessiva dedicació de temps a la refacció, provocant efectes negatius.
- Repercussions en la resta del programari i de l'equip de desenvolupadors quan un d'ells aplica tècniques de refacció.
- Possibles problemes de comunicació provocats pel punt anterior.
- Limitacions degudes a les bases de dades, interfícies gràfiques...

Alguns d'aquests punts febles relacionats amb la utilització de la refacció queden desenvolupats a continuació:

- **Dedicació de temps**. Una actitud obsessiva amb la refacció podrà portar a un efecte contrari al que es cerca: dedicar molt més temps del que caldria a la creació de codi i augmentar la complexitat del disseny i de la programació innecessàriament.

- **Afectar o generar problemes a la resta de l'equip de programació**. Una refacció d'un programador pot generar problemes a altres components de l'equip de treball, en funció d'on s'hagi dut a terme aquesta refacció. Si només afecta a una classe o a alguns dels seus mètodes, la refacció serà imperceptible a la resta dels seus companys. Però quan afecta a diverses classes, podrà alterar d'altre codi font que hagi estat desenvolupat o s'està desenvolupant per part d'altres companys. Aquest problema només es pot solucionar amb una bona comunicació entre els components de l'equip de treball o amb una refacció sincronitzada des dels responsables del projecte, mantenint informats els afectats.

- **Limitacions degudes a les bases de dades**. La refacció té algunes limitacions o àrees conflictives, com ara les bases de dades o les interfícies gràfiques. En el cas de les bases de dades, és un problema el fet que els programes que es desenvolupen actualment estiguin tan lligats a les seves estructures. En el cas d'haver-hi modificacions relacionades amb la refacció en el disseny de la base de dades vinculada a una aplicació, caldria dur a terme moltes accions que complicarien aquesta actuació: caldria anar a la base de dades, efectuar els canvis estructurals adients, fer una migració de les dades cap al nou sistema i adaptar de nou tot allò de l'aplicació relacionat amb les dades (formularis, informes...).

- **Interfícies gràfiques**. Una segona limitació es troba amb les interfícies gràfiques d'usuari. Les noves tècniques de programació han facilitat la independència entre els diferents mòduls que componen les aplicacions. D'aquesta manera, es podrà modificar el contingut del codi font sense haver de fer modificacions en la resta de mòduls, com per exemple, en les interfícies. El problema amb la refacció vinculat amb les interfícies gràfiques radica en el fet que si aquesta interfície ha estat publicada en molts usuaris clients o si no es té accessibilitat al codi que la genera, serà pràcticament impossible actuar sobre ella.

La **refacció** es considera un aspecte molt important per al desenvolupament d'aplicacions mitjançant programació extrema.

2.1.2 Patrons de refacció més usuals

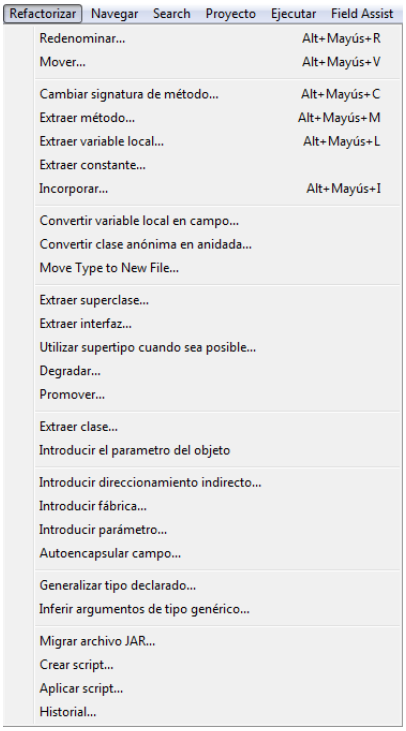
El patrons es poden trobar a totes les versions d'Eclipse per als diferents sistemes operatius (Windows, Linux i MacOS).

Els patrons, en un context de programació, ofereixen una solució durant el procés de desenvolupament de programari a un tipus de problema o de necessitat estàndard que pot donar-se en diferents contextos. El patró donarà una resolució a aquest problema, que ja ha estat acceptada com a solució bona, i que ja ha estat batejada amb un nom.

Per altra banda, ja ha quedat definida la refacció com a adaptacions del codi font sense que això provoqui canvis en les operacions del programari. Si s'uneixen aquests dos conceptes es poden trobar alguns patrons existents.

Com es pot observar a la figura 2.2, tots els patrons es poden dur a terme per mitjà de l'assistent d'Eclipse.

FIGURA 2.2. Eclipse: refactorització



Els patrons més habituals són els següents:

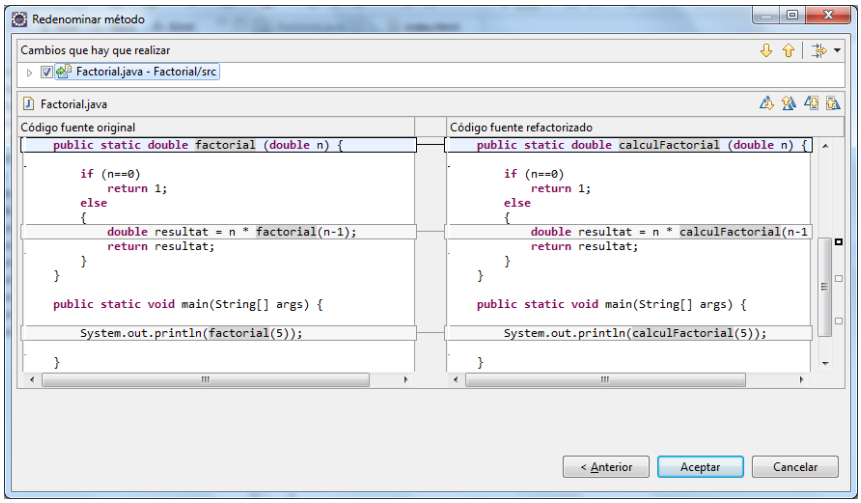
- Reanomenar
- Moure
- Extreure una variable local
- Extreure una constant
- Convertir una variable local en un camp
- Extreure una interfície
- Extreure el mètode

Reanomenar

Aquest patró canvia el nom de variables, classes, mètodes, paquets... tot tenint en compte les seves referències.

En l'exemple de la figura 2.3, es reanomena el mètode factorial amb el nom `calculFactorial`.

Figura 2.3. Eclipse refactorització: reanomenar

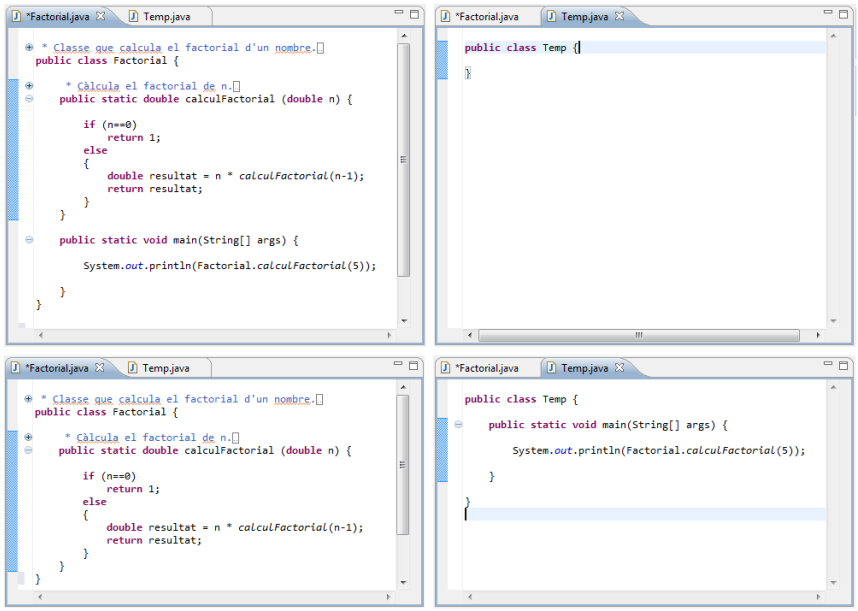


Moure

Aquest patró mou un mètode d'una classe a una altra; mou una classe d'un paquet a un altre... tot tenint en compte les seves referències.

En l'exemple de la figura 2.4, mou la funció principal (`main`), que es troba en la classe `Factorial`, cap a la classe `Temp`.

Figura 2.4. Eclipse refactorització: moure



Extreure una variable local

Donada una expressió, aquest patró li assigna una variable local; qualsevol referència a l'expressió en l'àmbit local serà substituïda per la variable.

En l'exemple, s'assigna l'expressió ‘‘`El factorial de `’’ com el valor de la variable `text`.

```
1  public static void main(String[] args) {
2      int nre = 3;
3      System.out.println("El factorial de " + nre + " és " + calculFactorial(nre)
           );
4      nre = 5;
5      System.out.println("El factorial de " + nre + " és " + calculFactorial(nre)
           );
6    }
```

El codi refactoritzat és el que es mostra a continuació:

```
1  public static void main(String[] args) {
2      int nre = 3;
3      String text = "El factorial de ";
4      System.out.println(text + nre + " és " + calculFactorial(nre));
5      nre = 5;
6      System.out.println(text + nre + " és " + calculFactorial(nre));
7    }
```

Extreure una constant

Donada una cadena de caràcters o un valor numèric, aquest patró el converteix en una constant, i qualsevol referència serà substituida per la constant.

En l'exemple, s'assigna l'expressió ‘‘`El factorial de `’’ com el valor de la constant `TEXT`.

```
1  public static void main(String[] args) {
2      int nre = 3;
3      System.out.println("El factorial de " + nre + " és " + calculFactorial(nre)
           );
4      nre = 5;
5      System.out.println("El factorial de " + nre + " és " + calculFactorial(nre)
           );
6    }
```

El codi refactoritzat és el que es mostra a continuació:

```
1  private static final String TEXT = "El factorial de ";
2
3    public static void main(String[] args) {
4      int nre = 3;
5      System.out.println(TEXT + nre + " és " + calculFactorial(nre));
6      nre = 5;
7      System.out.println(TEXT + nre + " és " + calculFactorial(nre));
8    }
```

Convertir una variable local en un camp

Donada una variable local, aquest patró la converteix en atribut de la classe; qualsevol referència serà substituïda pel nou atribut.

En l'exemple següent es converteix la variable local `nre` en un atribut de la classe `Factorial`.

```
1  public class Factorial {
2    public static double calculFactorial (double n) {
3      if (n==0)
4        return 1;
5      else
6      {
7        double resultat = n * calculFactorial(n-1);
8        return resultat;
9      }
10   }
11   public static void main(String[] args) {
12     int nre = 3;
13     System.out.println("El factorial de " + nre + " és " + calculFactorial(nre)
           );
14     nre = 5;
15     System.out.println("El factorial de " + nre + " és " + calculFactorial(nre)
           );
16   }
17 }
```

El codi refactoritzat és el que es mostra a continuació:

```
1  public class Factorial {
2    private static int nre;
3    public static double calculFactorial (double n) {
4      if (n==0)
5        return 1;
6      else
7      {
8        double resultat = n * calculFactorial(n-1);
9        return resultat;
10     }
11   }
12   public static void main(String[] args) {
13     nre = 3;
14     System.out.println("El factorial de " + nre + " és " + calculFactorial(nre)
           );
15     nre = 5;
16     System.out.println("El factorial de " + nre + " és " + calculFactorial(nre)
           );
17   }
18 }
```

Extreure una interfície

Interfície

Una interfície és un conjunt de mètodes abstractes i de propietats. En les propietats s'especifica què s'ha de fer, però no la seva implementació. Seran les classes que implementin aquestes interfícies les que descriguin la lògica del comportament dels mètodes.

Aquest patró crea una interfície amb els mètodes d'una classe.

En l'exemple es crea la interfície de la classe `Factorial`.

```
1  public class Factorial {
2    public double calculFactorial (double n) {
3      if (n==0)
4        return 1;
5      else
```

```
    {
      double resultat = n * calculFactorial(n-1);
      return resultat;
    }
  }
}
```

El codi refactoritzat és el que es mostra tot seguit:

```
public interface InterficieFactorial {
  public abstract double calculFactorial(double n);
}
```

```
public class Factorial implements InterficieFactorial {
  /* (non-Javadoc)
   * @see InterficieFactorial#calculFactorial(double)
   */
  @Override
  public double calculFactorial (double n) {
    if (n==0)
      return 1;
    else
    {
      double resultat = n * calculFactorial(n-1);
      return resultat;
    }
  }
}
```

Extreure el mètode

Aquest patró converteix un tros de codi en un mètode.

En l'exemple, es converteix la variable local `nre` com un atribut de la classe `Factorial`.

```
public static void main(String[] args) {
      int nre = 3;
      int comptador = 1;
      double resultat = 1;
      while (comptador<=nre){
        resultat = resultat * comptador;
        comptador++;
      }
      System.out.println("Factorial de " + nre + ": " + calculFactorial(nre));
  }
```

El codi refactoritzat és el que es mostra tot seguit:

```
private static void calcularFactorial(int nre) {
      int comptador = 1;
      double resultat = 1;
      while (comptador<=nre){
        resultat = resultat * comptador;
        comptador++;
      }
  }
  public static void main(String[] args) {
      int nre = 5;
      System.out.println("Factorial de " + nre + ": " + calculFactorial(nre));
  }
```

2.2 Proves i refacció. Eines d'ajuda a la refacció

Les eines d'ajuda a la creació de programari s'han de convertir en aliades en la tasca d'oferir solucions i facilitats per a l'aplicació de la refacció sobre el codi font que s'estigui desenvolupant. Un altre ajut molt important l'ofereixen els casos de prova. Les proves que s'hauran efectuat sobre el codi font són bàsiques per validar el correcte funcionament del sistema i per ajudar a decidir si aplicar o no un patró de refacció.

Cal anar amb compte amb els casos de prova escollits. Caldrà que acompleixin algunes característiques que ajudaran a validar la correctesa del codi font:

- **Casos de proves independents entre mòduls, mètodes o classes**. Els casos de prova han de ser independents per aconseguir que els errors en una part del codi font no afectin altres parts del codi. D'aquesta manera, es poden dur a terme proves incrementals que verifiquin si els canvis que s'han produït amb els processos de refacció han comportat canvis a la resta del programari.

- **Els casos de proves han de ser automàtics**. Si hi ha deu casos de proves, no serà viable que es vagin executant de forma manual un a un, sinó que cal que es puguin executar de forma automàtica tots per tal que, posteriorment, es puguin analitzar els resultats tant de forma individual com de forma conjunta.

- **Casos de proves autoverificables**. Una vegada que les proves s'estableixen de forma independent entre els mòduls i que es poden executar de forma automàtica, només cal que la verificació d'aquestes proves sigui també automàtica, és a dir, que la pròpia eina verifiqui si les proves han estat satisfactòries o no. L'eina indicarà per a quins casos de prova hi ha hagut problemes i en quina part del codi font, a fi que el programador pugui prendre decisions.

Per què són tan importants les proves per a una bona execució de la refacció? Les proves i els seus casos de proves són bàsics per indicar si el codi font desenvolupat funcionarà o no funcionarà. Però també ajudaran a saber fins a quin punt es poden aplicar tècniques de refacció sobre un codi font determinat o no.

S'ha d'anar amb compte a l'hora d'utilitzar aquests casos de proves. Si no s'apliquen de forma adient, podran suposar un problema més que una ajuda. Cal diferenciar molt el codi font implementat del cas de prova i que el lligam entre ells sigui al més petit possible. És important verificar que el codi que implementa la prova tingui una execució exitosa, independentment de quina hagi estat la seva implementació. Moltes vegades, una refacció en un codi font obliga a modificar els casos de prova. El que és recomanable és aplicar refaccions de mica en mica, de forma més contínua, però que siguin petits canvis o que els canvis afectin parts petites de codi.

Com caldrà implementar la refacció? Una proposta de metodologia és la que es descriu a continuació:

- Desenvolupar el codi font.
- Analitzar el codi font.
- Dissenyar les proves unitàries i funcionals.
- Implementar les proves.
- Executar les proves.
- Analitzar canvis a efectuar.
- Definir una estratègia d'aplicació dels canvis.
- Modificar el codi font.
- Execució de les proves.

Com es pot observar, aquesta metodologia sembla una petita gestió de projectes dins el propi projecte de desenvolupament de programari. Caldrà fer una bona anàlisi del codi font sobre el que es volen dur a terme les proves, un disseny dels casos de prova que s'implementaran, així com una correcta execució i una anàlisi dels resultats obtinguts.

A continuació es desenvolupa, més detalladament, aquesta proposta de metodologia:

- **Desenvolupament del codi font:** aquesta part no esdevé pròpiament part de la metodologia per implementar la refacció. Abans de dur a terme aquesta tècnica, caldrà tenir desenvolupat el codi que es voldrà analitzar.
- **Analitzar el codi font:** una vegada el programari, o una part d'aquest, ha estat desenvolupat, caldrà dur a terme una anàlisi exhaustiva d'aquest codi per comprovar si es detecten trossos de codi on es podrà dur a terme la refacció. Per determinar-ho, la gran majoria de vegades caldrà dissenyar i aplicar casos de prova.
- **Dissenyar les proves unitàries i funcionals:** abans d'executar-les, cal escriure les proves. Però, abans d'això, caldrà haver analitzat el codi font (pas anterior) i dissenyat els casos de prova. Dur a terme la refacció sense haver executat abans proves unitàries i proves funcionals pot arribar a resultar molt costós i pot implicar un risc molt important per al codi font desenvolupat.
- **Implementar i executar les proves:** una vegada dissenyats els casos de proves, cal implementar-los i executar-los. La raó d'aquesta execució és obtenir més informació referent a com es comportarà el programari en les diferents situacions preparades. Aquestes situacions han de tenir en compte diversos escenaris, en situacions extremes i en situacions normals. El comportament actual del sistema, abans d'efectuar qualsevol canvi, haurà de ser el mateix comportament que una vegada efectuada la refacció.

- **Analitzar canvis a efectuar:** els casos de prova han permès veure quin és el comportament del programari desenvolupat, però també ajuden a mostrar els canvis que es podran efectuar en aquest programari. Els resultats de les proves ofereixen informació sobre els patrons de refacció i de disseny que es podran dur a terme. A més de trobar llocs en el codi que ofereixen indicacions directes de millora amb la refacció, els casos de prova també ofereixen informacions sobre altres refaccions no tan comunes, de les quals els programadors no tenen tan clara la seva necessitat de millora, però que faran que el codi es vagi deteriorant de forma progressiva en el cas de no actuar a temps.

- **Definir una estratègia d'aplicació dels canvis:** aquesta estratègia d'actuació haurà d'aplicar-se de forma progressiva. Cal aplicar primer un conjunt de canvis per confirmar, a continuació, l'estabilitat del sistema, és a dir, confirmar que els canvis no hagin provocat cap altre problema o error. A continuació, es durà a terme un altre conjunt de canvis, i així successivament, fins a finalitzar-los tots. Hi ha actuacions de refacció més senzilles d'efectuar que d'altres, i altres canvis que ataquen directament els errors de disseny de l'aplicació. Per exemple, resoldre problemes com el codi duplicat o les classes llargues, amb canvis petits i senzills, permet que se solucionin problemes importants de disseny; això significarà haver aportat millores grans i importants al codi font.

- **Modificar el codi font i executar les proves:** els canvis que s'hagin produït al codi desenvolupat moltes vegades són molts, però petits. Si es fan manualment, es corre el risc d'equivocar-se. En canvi, utilitzant eines específiques per aplicar els canvis de refacció, es poden efectuar de forma automàtica sense cap por a equivocar-se. Una vegada modificat el codi, caldrà dur a terme l'execució de les proves. Aquesta vegada l'execució validarà que els casos de prova executats en aquest moment coincideixin amb els casos de prova obtinguts anteriorment. D'aquesta manera, es confirmarà que els canvis efectuats no han afectat els resultats esperats.

2.2.1 Eines per a l'ajuda a la refacció

IDE, de l'anglès Integrated Development Environment, és un entorn de desenvolupament integrat.

Actualment, moltes IDE ofereixen eines que ajuden a la refacció del codi font. Aquestes eines solen estar integrades o permeten la descàrrega de mòduls externs o connectors. Amb aquests complements es poden dur a terme molts dels patrons de refacció de forma automàtica o semiautomàtica.

La classificació d'aquestes eines es pot fer a partir de molts criteris diferents, com ara el tipus d'eina de refacció (si és privativa o programari lliure), per les funcionalitats que ofereix (mirar codi duplicat, anàlisi de la qualitat del programari, proposta d'ubicacions al codi font on es poden aplicar accions de refacció...) o a partir dels llenguatges de programació que permeten analitzar.

A continuació, s'enumeren algunes d'aquestes eines seguint la darrera classificació:

- **Java**: RefactorIt, Condenser, JCosmo, Xrefactory, jFactor, IntelliJ IDEA.
- **Visual C++, Visual C#, Visual Basic .NET, ASP.Net, ...**: Visual Studio.
- **C++**: CppRefactory, Xrefactory.
- **C#**: C# Refactoring Tool, C# Refactory.
- **SQL**: SQL Enlight.
- **Delphi**: Modelmaker tool, Castalia.
- **Smalltalk**: RefactoringBrowser.

En la secció *Activitats* del web del mòdul hi podeu trobar activitats de refacció amb Eclipse.

L'IDE Eclipse serveix com a exemple d'eina que permet dur a terme la refacció, com s'ha mostrat en apartats anteriors. Eclipse ja porta integrades diverses eines de refacció en la seva instal·lació estàndard. Es poden trobar al Menú principal, com un apartat propi anomenat *Refactor*, o bé utilitzant el menú contextualitzat mentre es treballa amb l'editor.

2.3 Control de versions

Per a una feina que no comença i acaba dins un període curt de temps, o per a feines que han de ser desenvolupades per un equip de persones, és recomanable tenir un control de les tasques que s'han fet, quan s'han dut a terme, qui les ha fet...

Es parla de gestió de projectes quan s'ha de desenvolupar una feina com la que s'acaba de definir i, a més a més, es planifica. Què té a veure la gestió de projectes amb el control de versions d'un programari que s'està desenvolupant?

Hi tindrà molt a veure l'analogia que tenen els dos procediments. Si la feina demanada és el canvi de la font d'alimentació d'un ordinador, no caldrà ni planificar-ho ni, probablement, en cas que es deixi la feina a mitges, caldrà deixar documentada la situació en què es troba la feina en qüestió. En canvi, si un únic programador ha d'implementar una calculadora, serà discutible decidir si necessitarà una planificació i una gestió d'aquesta feina com si fos un projecte (planificant, analitzant, dissenyant i desenvolupant) o si necessitarà una eina que gestioni les versions desenvolupades fins al moment. Potser portar un control de versions del programari desenvolupat li servirà per poder tornar enrere en cas d'arribar a un punt de no retorn.

ERP

Un ERP (*Enterprise Resource Planning*) és un sistema informàtic que abraça tota una empresa, i que s'utilitza per gestionar tots els seus recursos i compartir la informació necessària entre els diferents departaments en una única base de dades.

Quan és molt recomanable utilitzar un control de versions (i, anàlogament, una gestió del projecte)? En casos en què el desenvolupament de programari comporta una feina de moltes hores, de molts fitxers diferents i (encara que no necessàriament) la presència de diverses persones treballant sobre el mateix projecte, com per exemple en el desenvolupament d'un ERP (*Enterprise Resource Planning*).

Un **sistema de control de versions** és una eina d'ajuda al desenvolupament de programari que anirà emmagatzemant, segons els paràmetres indicats, la situació del codi font en moments determinats. És com una eina que va fent fotos de forma regular, cada cert temps, sobre l'estat del codi.

En un entorn on només treballarà un programador, sols caldrà guardar la informació del codi cada cert temps o cada vegada que ell desi la informació, juntament amb les dades principals, com ara el número de la versió o la data i l'hora en què s'ha emmagatzemat. En un entorn multiusuari, on molts programadors diferents poden manipular els fitxers i on, fins i tot, es podrà oferir la possibilitat de modificar a la vegada un mateix document, en aquests casos cal emmagatzemar molta més informació, com ara quin usuari ha implementat els canvis, les referències de la màquina des d'on s'han fet els canvis, si s'està produint algun tipus de conflicte...

Aquesta funcionalitat no només serà important en els casos de desenvolupament de programari, sinó també en molt altres àmbits. Per posar uns exemples, es pot trobar la importància del control de versions en el cas de treballar en la creació d'un llibre, col·laborant diversos autors en la seva redacció. Si queda enregistrat en quin moment ha fet cada canvi qualsevol col·laborador diferent, i, a més, queda enregistrada una còpia de cada modificació feta, això permetrà a la resta de col·laboradors tenir una informació molt important per no repetir continguts ni duplicar feines. Un exemple real per al que s'acaba d'explicar es pot trobar en la redacció de continguts de la wiquipèdia, on molts redactors creen continguts mitjançant una eina que permet el treball en equip i el control dels continguts creats i penjats. Un altre cas es pot trobar en una gestoria o un bufet d'advocats que han de redactar un contracte o un document en col·laboració amb un o diversos clients. En aquest cas, no serà necessari fer servir una eina com una wiki, potser serà suficient fer servir un bon editor de textos que permeti la funcionalitat de gestió de canvis, on cada vegada que un usuari diferent hagi de fer un canvi, quedi indicat en el document amb un color diferent en funció de l'usuari que l'hagi creat.

En el desenvolupament de programari, els sistemes de control de versions són eines que poden facilitar molt la feina dels programadors i augmentar la productivitat de forma considerable, sempre que aquestes eines siguin utilitzades de forma correcta. Algunes funcionalitats dels sistemes de control de versions poden ser:

- **Comparar canvis en els diferents arxius al llarg del temps**, podent veure qui ha modificat per darrera vegada un determinat arxiu o tros de codi font.
- **Reducció de problemes de coordinació que pot haver-hi entre els diferents programadors**. Amb els sistemes de control de versions podran compartir la seva feina, oferint les darreres versions del codi o dels documents, i treballar, fins i tot, de forma simultània sense por a trobar-se amb conflictes en el resultat d'aquesta col·laboració.
- **Possibilitat d'accedir a versions anteriors dels documents o codi font**. De forma programada es podrà automatitzar la generació de còpies de

seguretat o, fins i tot, emmagatzemar tot canvi efectuat. En el cas d'haver-se equivocat de forma puntual, o durant un període llarg de temps, el programador podrà tenir accés a versions anteriors del codi o desfer, pas a pas, tot allò desenvolupat durant els darrers dies.

- **Veure quin programador ha estat el darrer a modificar un determinat tros de codi** que pot estar causant un problema.
- **Accés a l'historial de canvis sobre tots els arxius a mesura que avança el projecte**. També pot servir per al cap de projectes o per a qualsevol altra part interessada (*stakeholder*), amb permisos per accedir a aquest historial, per veure l'evolució del projecte.
- **Tornar un arxiu determinat** o tot el projecte sencer a un o a diversos estats anteriors.

Els sistemes de control de versions ofereixen, a més, algunes funcionalitats per poder gestionar un projecte informàtic i per poder-ne fer el seguiment. Entre aquestes funcionalitats es poden trobar:

- **Control històric detallat de cada arxiu.** Permet emmagatzemar tota la informació del que ha succeït en un arxiu, com ara tots els canvis que s'han efectuat, per qui, el motiu dels canvis, emmagatzemar totes les versions que hi ha hagut des de la seva creació...
- **Control d'usuaris amb permisos per accedir als arxius.** Cada usuari tindrà un tipus d'accés determinat als arxius per poder consultar-los o modificar-los o, fins i tot, esborrar-los o crear-ne de nous. Aquest control ha de ser gestionat per l'eina de control de versions, emmagatzemant tots els usuaris possibles i els permisos que tenen assignats.
- **Creació de branques d'un mateix projecte:** en el desenvolupament d'un projecte hi ha moments en què cal ramificar-lo, és a dir, a partir d'un determinat moment, d'un determinat punt, cal crear dues branques del projecte que es podran continuar desenvolupant per separat. Aquest cas es pot donar en el moment de finalitzar una primera versió d'una aplicació que es lliura als clients, però que cal continuar evolucionant.
- **Fusionar dues versions d'un mateix arxiu:** permetent fusionar-les, agafant, de cada part de l'arxiu, el codi que més interessi als desenvolupadors. Aquesta funcionalitat s'haurà de validar manualment per part d'una persona.

2.3.1 Components d'un sistema de control de versions

Un sistema de control de versions es compondrà de diversos elements o components que fan servir una terminologia una mica específica. Cal tenir en compte que no tots els sistemes de control de versions utilitzen els mateixos termes per referir-se als mateixos conceptes. A continuació, trobareu una llista amb el nom en català dels termes més comuns i el nom o noms en anglès relacionats:

- **Repositori** (*repository* o *depot*): conjunt de dades emmagatzemades, també referit a versions o còpies de seguretat. És el lloc on aquestes dades queden emmagatzemades. Es podrà referir a moltes versions d'un únic projecte o de diversos projectes.
- **Mòdul** (*module*): es refereix a una carpeta o directori específic del repositori. Un mòdul podrà fer referència a tot el projecte sencer o només a una part del projecte, és a dir, a un conjunt d'arxius.
- **Tronc** (*trunk* o *master*): estat principal del projecte. És l'estat del projecte destinat, en acabar el seu desenvolupament, a ser passat a producció.
- **Branca** (*branch*): és una bifurcació del tronc o branca mestra de l'aplicació que conté una versió independent de l'aplicació i a la qual poden aplicar-se canvis sense que afectin ni el tronc ni altres branques. Aquests canvis, en un futur, poden incorporar-se al tronc.
- **Versió** o **Revisió** (*version* o *revision*): és l'estat del projecte o d'una de les seves branques en un moment determinat. Es crea una versió cada vegada que s'afegeixen canvis a un repositori.
- **Etiqueta** (*tag*, *label* o *baseline*): informació que s'afegirà a una versió. Sovint indica alguna característica específica que el fa especial. Aquesta informació serà textual i, moltes vegades, es generarà de forma manual. Per exemple, es pot etiquetar la primera versió d'un programari (1.0) o una versió en la qual s'ha solucionat un error important.
- **Cap** (*head* o *tip*): fa referència a la versió més recent d'una determinada branca o del tronc. El tronc i cada branca tenen el seu propi cap, però, per referir-se al cap del tronc, de vegades s'utilitza el terme HEAD, en majúscules.
- **Clonar** (*clone*): consisteix a crear un nou repositori, que és una còpia idèntica d'un altre, ja que conté les mateixes revisions.
- **Bifurcació** (*fork*): creació d'un nou repositori a partir d'un altre. Aquest nou repositori, al contrari que en el cas de la clonació, no està lligat al repositori original i es tracta com un repositori diferent.
- ***Pull***: és l'acció que copia els canvis d'un repositori (habitualment remot) en el repositori local. Aquesta acció pot provocar conflictes.
- ***Push*** o ***fetch***: són accions utilitzades per afegir els canvis del repositori local a un altre repositori (habitualment remot). Aquesta acció pot provocar conflictes.
- **Canvi** (*change* o *diff*): representa una modificació concreta d'un document sota el control de versions.
- **Sincronització** (*update* o *sync*): és l'acció de combinar els canvis fets al repositori amb la còpia de treball local.
- **Conflicte** (*conflict*): es produeix quan s'intenten afegir canvis a un fitxer que ha estat modificat prèviament per un altre usuari. Abans de poder combinar els canvis amb el repositori s'haurà de resoldre el conflicte.

- **Fusionar** (*merge* o *integration*): és l'acció que es produeix quan es volen combinar els canvis d'un repositori local amb un remot i es detecten canvis al mateix fitxer en tots dos repositoris i es produeix un conflicte. Per resoldre aquest conflicte s'han de fusionar els canvis abans de poder actualitzar els repositoris. Aquesta fusió pot consistir a descartar els canvis d'un dels dos repositoris o editar el codi per incloure els canvis del fitxer a totes dues bandes. Cal destacar que és possible que un mateix fitxer presenti canvis en molts punts diferents que s'hauran de resoldre per poder donar la fusió per finalitzada.

- **Bloqueig** (*lock*): alguns sistemes de control de versions en lloc d'utilitzar el sistema de fusions el que fan és bloquejar els fitxers en ús, de manera que només pot haver-hi un sol usuari modificant un fitxer en un moment donat.

- **Directori de treball** (*working directory*): directori al qual el programador treballarà a partir d'una còpia que haurà fet del repositori en el seu ordinador local.

- **Còpia de treball** (*working copy*): fa referència a la còpia local dels fitxers que s'han copiat del repositori, que és sobre la qual es fan els canvis (és a dir, s'hi treballa, d'aquí el nom) abans d'afegir aquests canvis al repositori. És emmagatzemada al directori de treball.

- **Tornar a la versió anterior** (*revert*): descarta tots els canvis produïts a la còpia de treball des de l'última pujada al repositori local.

2.3.2 Classificació dels sistemes de control de versions

Es pot generalitzar l'estructura de les eines de control de versions tenint en compte la classificació dels sistemes de control de versions, que poden ser locals, centralitzats o distribuïts.

Sistemes locals

Els sistemes de control de versions locals són sistemes que permeten dur a terme les accions necessàries de forma local. Si no es fa servir cap sistema de control de versions concret, un programador que treballi en el seu propi ordinador podrà anar fent còpies de seguretat, de tant en tant, dels arxius o de les carpetes amb què treballi. Aquest sistema implica dues característiques específiques:

- El mateix programador serà la persona que s'haurà de recordar d'anar fent les còpies de seguretat cada cert temps, el que ell hagi establert.

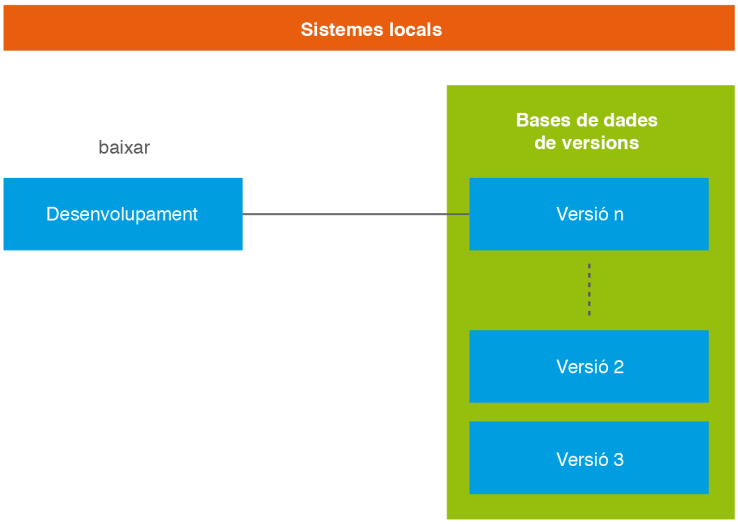
- El mateix programador haurà de decidir on durà a terme aquestes còpies de seguretat, molt probablement en local, en una altra ubicació del disc dur intern, o en un dispositiu d'emmagatzematge extern.

Aquest sistema té un alt risc perquè és un sistema altament dependent del programador, però és un sistema extremadament senzill de planificar i d'executar. El més habitual, en aquests casos, és crear còpies de seguretat (que es poden considerar versions del projecte); els fitxers s'emmagatzemen en carpetes que solen tenir un nom representatiu, com per exemple: la data i l'hora en què s'efectua la còpia. Però, tal com s'ha comentat anteriorment, es tracta d'un sistema propens a errors per diversos motius, com ara que es produeixin oblits en la realització de la còpia de seguretat, o confusions que portin al programador a continuar el desenvolupament en una de les còpies del codi, i anar avançant amb el projecte en diferents ubicacions diferents dies.

Per fer front a aquests problemes, van aparèixer els primers repositoris de versions que contenien una petita base de dades on es podien enregistrar tots els canvis efectuats, sobre quins arxius, qui els havia fet, quan... Efectuant les còpies de seguretat de forma automatitzada.

A la figura 2.5 es pot observar la representació del sistema local, amb els arxius i les seves còpies o versions.

Figura 2.5. Sistema de control de versions locals



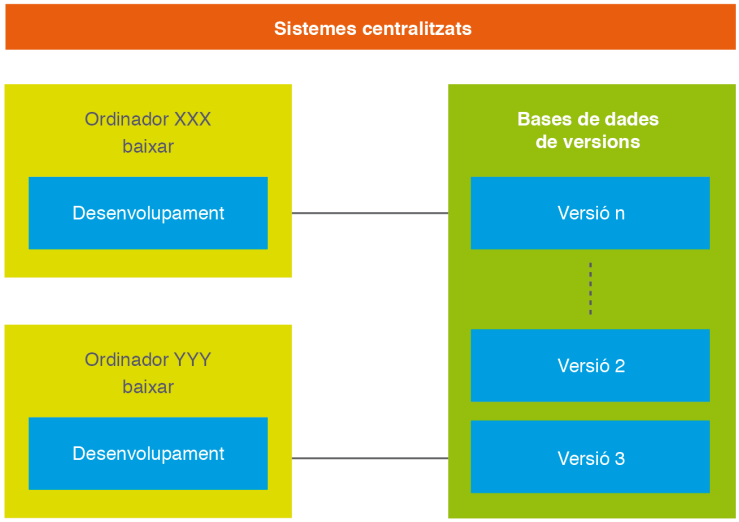
Sistemes centralitzats

Un sistema de control de versions que permeti desenvolupar un projecte informàtic amb més d'un ordinador és el sistema de control de versions centralitzat que es contraposa a un altre sistema de control de versions, també per a més d'un ordinador, com és el distribuït.

En els sistemes centralitzats hi haurà més d'un programador desenvolupant un projecte en més d'un ordinador. Des dels dos ordinadors es podrà accedir als mateixos arxius de treball i sobre les mateixes versions emmagatzemades. Aquesta és una situació més pròxima a les situacions actuals reals en el desenvolupament de projectes informàtics.

El sistema de control de versions centralitzat és un sistema on les còpies de seguretat o versions emmagatzemades es troben de forma centralitzada a un servidor que serà accessible des de qualsevol ordinador que treballi en el projecte. A la figura 2.6 es pot observar com es representa aquest sistema de control de versions centralitzat.

FIGURA 2.6. Sistema de control de versions centralitzat



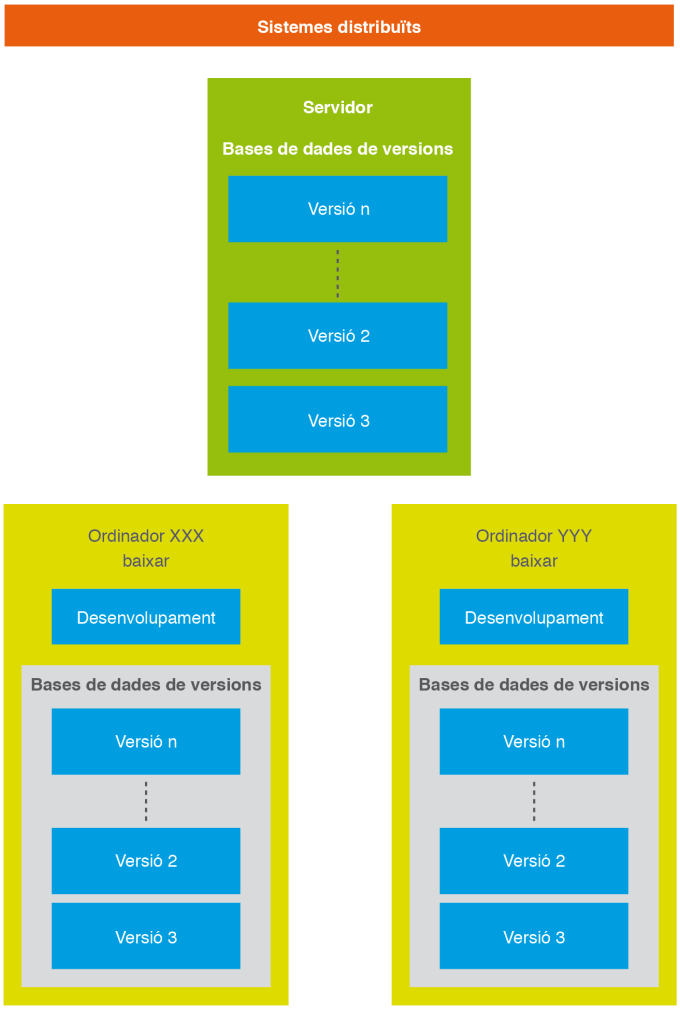
Però aquest sistema també tindrà els seus inconvenients, de vegades molt difícils de salvar. Què succeeix si falla el servidor central? Si és una fallada temporal no s'hi podrà accedir durant un petit període temps. Si és una fallada del sistema, caldrà tenir preparat un sistema de recuperació o un sistema alternatiu (una còpia). Durant aquest temps, els diferents desenvolupadors no podran treballar de forma col·laborativa ni accedir a les versions emmagatzemades ni emmagatzemar-ne de noves. Aquest és un desavantatge difícil de contrarestar.

Com que el repositori es troba centralitzat en un únic ordinador i una única ubicació, la creació d'una brancaa es durà a terme a la mateixa ubicació que la resta del repositori, utilitzant una nova carpeta per crear la duplicitat. Els punts febles amb el treball de les branques seran els mateixos que s'han exposat en general per als sistemes de control de versions centralitzats.

Sistemes distribuïts

Els sistemes de control de versions distribuïts ofereixen una solució a aquest desavantatge ofert pels sistemes de control de versions centralitzats. Com es pot veure a la figura 2.7, la solució que ofereixen els sistemes distribuïts és disposar cada ordinador de treball, així com el servidor, d'una còpia de les versions emmagatzemades. Aquesta duplicitat de les versions ofereix una disponibilitat que disminueix moltíssim les possibilitats de no tenir accessibilitat als arxius i a les seves versions.

FIGURA 2.7. Sistema de control de versions distribuït



Quina és la forma de treballar d'aquest sistema? Cada vegada que un ordinador client accedeix al servidor per tenir accés a una versió anterior, no només copien aquell arxiu, sinó que fan una descàrrega completa dels arxius emmagatzemats.

Si el servidor té una fallada del sistema, la resta d'ordinadors clients podran continuar treballant i qualsevol dels ordinadors clients podrà efectuar una còpia sencera de tot el sistema de versions cap al servidor per tal de restaurar-lo. La idea és que el servidor és el que gestiona el sistema de versions, però en cas de necessitat pot accedir a les còpies locals que es troben en els ordinadors clients.

2.3.3 Operacions bàsiques d'un sistema de control de versions

Entre les operacions més habituals d'un sistema de control de versions (tant en els sistemes centralitzats com en els distribuïts) es poden diferenciar dos tipus: aquelles que permeten l'entrada de dades al repositori i aquelles que permeten obtenir dades del repositori. A la figura 2.8 es mostra un resum d'aquestes operacions.

Entre les operacions d'introducció de dades al repositori es poden trobar:

- **Importació de dades:** aquesta operació permet dur a terme la primera còpia de seguretat o versionat dels arxius amb els quals es treballarà en local cap al repositori.

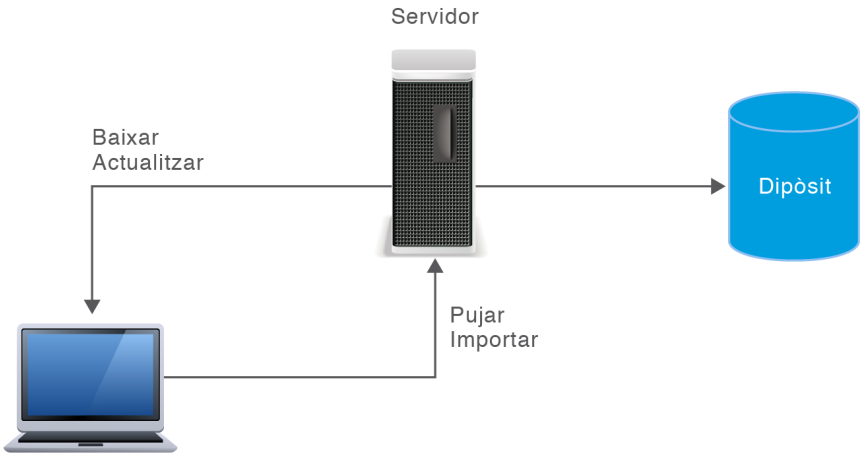
- **Pujar (*commit* o *check in*):** aquesta operació permet enviar al repositori les dades corresponents als canvis que s'han produït en el servidor local. No es farà una còpia sencera de tota la informació, sinó que només es treballarà amb els arxius que s'hagin modificat en el darrer espai de temps. Cal destacar que no els envia al servidor; els canvis queden emmagatzemats al repositori local, que s'ha de sincronitzar.

Entre les operacions d'exportació de dades del repositori es poden trobar:

- **Baixar (*check-out*):** amb aquesta operació es podrà tenir accés i descarregar a l'àrea de treball local una versió des d'un repositori local, un repositori remot o una branca diferent.

- **Actualització (*update*):** aquesta operació permet dur a terme una còpia de seguretat de totes les dades del repositori a l'ordinador client amb què treballarà el programador. Serà una operació que es podrà efectuar de forma manual, quan el programador ho estimi oportú, o de forma automàtica, com en els sistemes distribuïts, on cada vegada que un client accedeix al repositori es fa una còpia completa en local.

Figura 2.8. Operacions que es poden fer en un sistema de control de versions



Actualment, l'opció més popular és una mescla entre els dos sistemes: utilitzar un servidor central i fer servir als clients un sistema distribuït. Convé triar com a servidor central una plataforma amb un alt nivell de seguretat.

2.4 Eines de control de versions

Per dur a terme un control de versions automatitzat existeixen diverses eines que faciliten molt aquesta feina. Aquestes eines poden ser independents a la resta de les eines que es faran servir per a la gestió i el desenvolupament del projecte informàtic o es poden trobar integrades amb altres eines, com ara Visual Studio o Eclipse, facilitant així la feina dels membres de l'equip del projecte. Les funcionalitats que ofereixen poden anar des del control del codi font, la configuració i gestió del canvi, l'administració de versions d'arxius i de directoris d'un projecte, fins a la integració completa dels projectes, analitzant, planificant, compilant, executant i provant de forma automàtica els projectes.

Algunes eines de control de versions són:

- **Team Foundation Server** (TFS): és un sistema que pot utilitzar les arquitectures centralitzades i distribuïdes. És gratuït per a equips petits i projectes de codi lliure; a la resta de supòsits, cal pagar una subscripció. Aquesta eina ha estat desenvolupada per Microsoft, cosa que implica que sigui una eina privativa. És una eina preparada per treballar amb col·laboració amb Visual Studio. Permet accions de control de codi, administració del projecte, seguiment dels elements de treball i gestió dels arxius a partir d'un portal web del projecte. Es tracta d'una evolució de l'eina Visual Source Safe.
- **CVS - Concurrent Versions System.** Aquesta eina ofereix un sistema de control de versions centralitzat amb una sèrie de funcionalitats que ajuden el programador:
  - Manteniment del registre de tot el treball per part dels membres de l'equip del projecte.
  - Enregistrament de tots els canvis en els fitxers.
  - Permet el treball en equip en col·laboració per part de desenvolupadors a gran distància.
- **Subversion** (SVN): eina de codi obert, gratuït, independent del sistema operatiu de la màquina en què s'utilitzi. És un sistema centralitzat. Es va desenvolupar l'any 2000 amb l'objectiu de substituir CVS. Afegeix noves funcionalitats i en millora algunes altres que no acabaven de funcionar adequadament amb l'eina CVS.
- **Mercurial**: és un sistema distribuït gratuït.
- **Git**: Eina de Gestió de Versions desenvolupada per a programadors del nucli de Linux. Desenvolupada a partir d'evolucions de l'eina CVS, ja que Subversion no cobria les necessitats dels desenvolupadors del nucli de Linux. És un sistema distribuït i gratuït. Actualment és el programari més popular de control de version amb diferència.

Popularitat i ús dels sistemes de control de versions.

- **Popularitat dels sistemes de control de versions**: al següent enllaç podeu trobar la gràfica de *Google Trends* amb la popularitat de diferents sistemes de control de versions: goo.gl/JDfg9r.
- **Control de versions al món laboral**: segons un informe d'IT Jobs Watch (www.itjobswatch.co.uk) de l'any 2016 sobre ofertes de treball al Regne Unit, el percentatge d'ofertes de feina que inclouen entre els seus requisits el coneixement de sistems de control de versions és:
  - 29,27% Git
  - 12,17% Microsoft Team Foundation Server
  - 10,60% Subversion
  - 1,30% Mercurial

2.4.1 Altres eines

Existeixen moltes altres eines per a l'ajuda del control de versions. Algunes es poden classificar en funció del llenguatge de programació al que assisteixen i d'altres es poden classificar en funció del o dels sistemes per a què van ser desenvolupades. Podeu veure tot seguit altres eines de sistema de gestió de versions, classificades en funció de si pertanyen a programari lliure o privatiu:

- Programari lliure:
  - GNU Arch
  - RedMine
  - Mercurial (ALSA, MoinMoin, Mutt, Xen)
  - PHP Collab
  - Git
  - CVS
  - Subversion
- Programari privatiu:
  - Clear Case
  - Darcs
  - Team Fundation Server

2.5 Dipòsits de les eines de control de versions

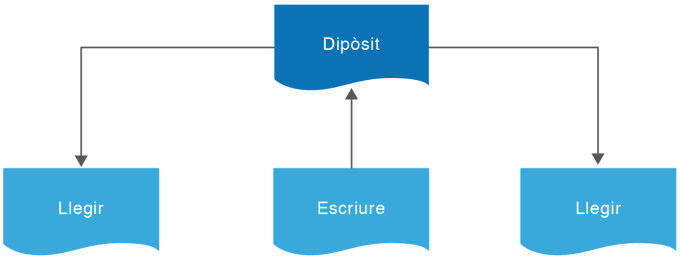
Un **dipòsit** és un magatzem de dades on es guardarà tot allò relacionat amb una aplicació informàtica o unes dades determinades d'un determinat projecte.

L'ús de dipòsits no és una tècnica exclusiva de les eines de control de versions. Les aplicacions i els projectes informàtics solen fer servir dipòsits que contenen informació. Per exemple, un dipòsit amb informació referent a una base de dades contindrà tot el referent a com està implementada aquesta base de dades: quines taules tindrà, quins camps, com seran aquests camps, les seves característiques, els seus valors límits...

En l'àmbit del control de versions, tenir aquest dipòsit suposa poder comptar amb un magatzem central de dades que guarda tota la informació en forma de fitxers i de directoris, i que permet dur a terme una gestió d'aquesta informació.

Tota eina de control de versions té un repositori que és utilitzat com un magatzem central de dades. La informació emmagatzemada serà tota la referent al projecte informàtic que s'estarà desenvolupant. Els clients es connectaran a aquest repositori per llegir o escriure aquesta informació, accedint a informació d'altres clients o fent pública la seva pròpia informació. A la figura 2.9 es mostra un exemple conceptual d'un repositori centralitzat on accedeixen diversos clients per escriure o llegir arxius.

Figura 2.9. Esquema conceptual d'un repositori



Un **repositori** és la part principal d'un sistema de control de versions. Són sistemes dissenyats per enregistrar, guardar i gestionar tots els canvis i informacions sobre totes les dades d'un projecte al llarg del temps.

Gràcies a la informació enregistrada en el repositori es podrà:

- Consultar la darrera versió dels arxius que s'hagin emmagatzemat.
- Accedir a la versió d'un determinat dia i comparar-la amb l'actual.
- Consultar qui ha modificat un determinat tros de codi i quan va ser modificat.

2.5.1 Problemàtiques dels sistemes de control de versions

Els sistemes de control de versions han de resoldre certes problemàtiques, com per exemple, com evitar que les accions d'un programador se sobreposin a les d'altres programadors.

Comptem amb diferents alternatives per resoldre aquestes problemàtiques:

- Compartir arxius per part de diferents programadors
- Bloquejar els arxius utilitzats
- Fusionar els arxius modificats

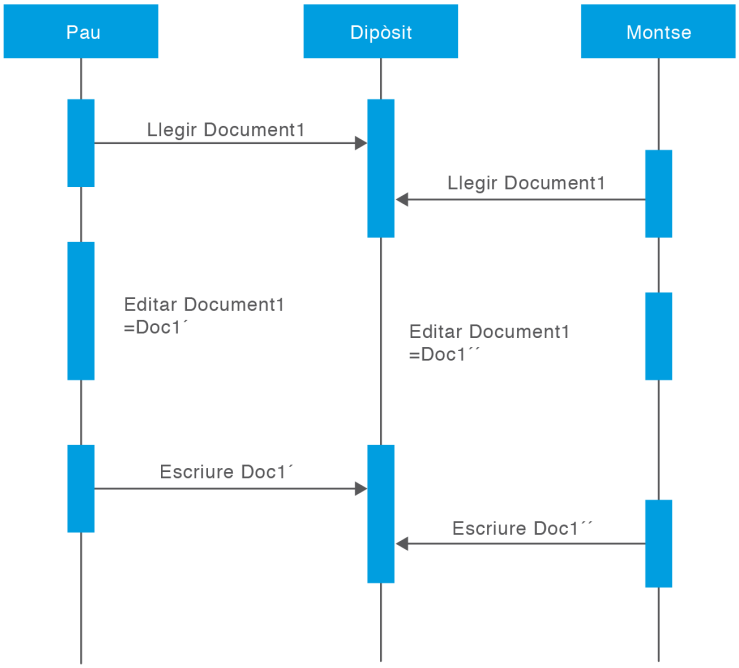
Compartir arxius per part de diferents programadors

Dos companys que treballen en el mateix projecte, desenvolupant una aplicació informàtica, decideixen editar el mateix fitxer del repositori a la vegada. Per exemple:

- En Pau i la Montse estan editant un mateix fitxer.
- En Pau grava els seus canvis al repositori.
- Com que en Pau i la Montse hi han accedit a la vegada, la versió sobre la qual la Montse treballa no contindrà els canvis efectuats per en Pau.
- La Montse podrà, accidentalment, sobreescriure'ls amb la seva nova versió de l'arxiu, pel fet de ser la darrera persona que els guarda. Els canvis de'n Pau no es troben en la nova versió de la Montse.
- La versió de l'arxiu de'n Pau no s'ha perdut per sempre (perquè el sistema recorda cada canvi).
- La resta de programadors que accedeixin al repositori veuran la nova versió de la Montse, però no podran veure els canvis de'n Pau a no ser que indaguin en l'històric de l'arxiu.

A la figura 2.10 es pot observar aquesta seqüència de passos que es poden donar. El treball de'n Pau s'haurà obviat, cosa que s'ha d'evitar que es produeixi. Per això existeixen diverses solucions que ofereixen les eines de control de versions.

FIGURA 2.10. Cas d'accés a la vegada al mateix arxiu per part de dos usuaris



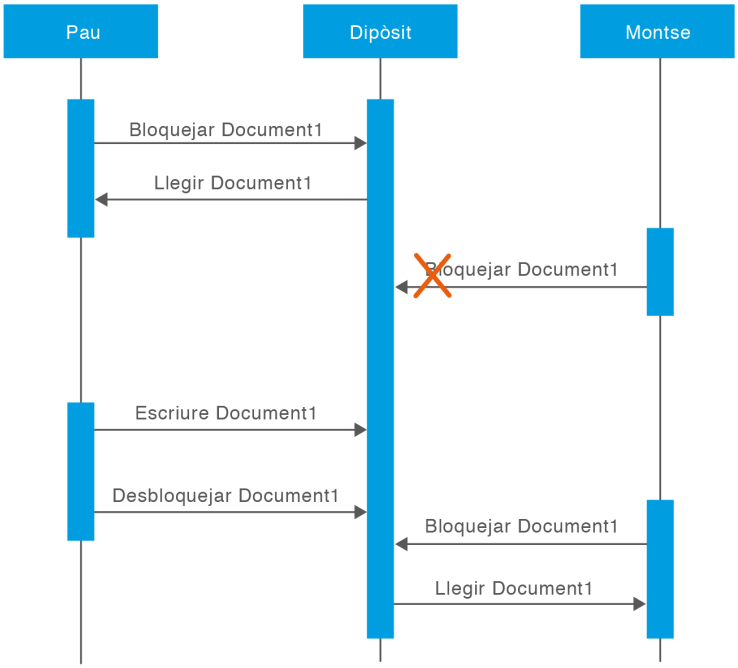
Bloquejar els arxius utilitzats

El cas exposat anteriorment es pot prevenir utilitzant alguna de les tècniques que ofereixen les eines de control de versions. Molts d'aquests sistemes utilitzen un model de solució que consisteix a bloquejar els arxius afectats quan un usuari hi està accedint en mode modificació. La seqüència seria:

- Bloquejar l'arxiu a què es vol accedir.
- Modificar l'arxiu per part de l'usuari.
- Desbloquejar l'arxiu una vegada modificat i actualitzat al repositori.

Es tracta d'una solució senzilla tant conceptualment com d'implementació. Però té alguns punts no tan positius. En primer lloc, només permet accedir a la modificació d'un arxiu a un únic usuari. En el cas plantejat, en Pau podrà accedir a l'arxiu per consultar i modificar, però la Montse no hi podrà accedir, ja que en Pau el tindrà bloquejat. S'haurà d'esperar que en Pau l'acabi de modificar per desbloquejar-lo i, llavors, podrà accedir-hi ella. Es tracta d'una solució molt restrictiva que pot afectar negativament la feina dels desenvolupadors. A la figura 2.11 es poden veure aquests procediments.

FIGURA 2.11. Bloquejar els arxius utilitzats



Alguns problemes d'aquesta solució de bloqueig d'arxius són:

- Possible creació d'inconsistències. Pot semblar que utilitzant el sistema dels bloquejos hi hagi més seguretat a l'hora de manipular arxius, però pot donar-se el cas contrari. Si en Pau bloqueja i edita l'arxiu A, mentre la Montse simultàniament bloqueja i edita l'arxiu B, però els canvis que fan no tenen en compte els canvis de l'altre desenvolupador, es pot donar una situació en què, en deixar de bloquejar els arxius, els canvis fets a cada un siguin semànticament incompatibles. Tot d'una, A i B ja no funcionen junts.

- Mentre un usuari, per exemple en Pau, està bloquejant un arxiu, l'accés per part de la resta de desenvolupadors no és viable. Si en Pau oblida desbloquejar l'arxiu o el demora durant un temps llarg, la resta de companys amb necessitat d'accedir-hi quedaran a l'espera de poder continuar la seva feina, bloquejats per en Pau. Això pot causar molts problemes de caire administratiu dins l'equip de treball.

- De vegades, les necessitats d'accés a un mateix arxiu són independents, és a dir, es pot requerir l'accés a diferents parts del codi. En Pau pot necessitar editar l'inici d'un arxiu i la Montse simplement vol canviar la part final d'aquest arxiu. Aquests canvis no se superposen en absolut. Ells podrien fàcilment editar el fitxer de forma simultània, i no hi hauria cap problema sempre que s'assumís que els canvis es fusionen correctament. En aquesta situació no és necessari seguir una política de bloquejos.

Precisament, aquest darrer problema del sistema de bloquejos és el que suggereix una altra solució per a l'accés simultani al repositori.

Fusionar els arxius modificats

Un altre sistema per solucionar la problemàtica d'accés simultani a un mateix arxiu, com a alternativa al sistema de bloqueig d'arxius, és el que es descriu a continuació:

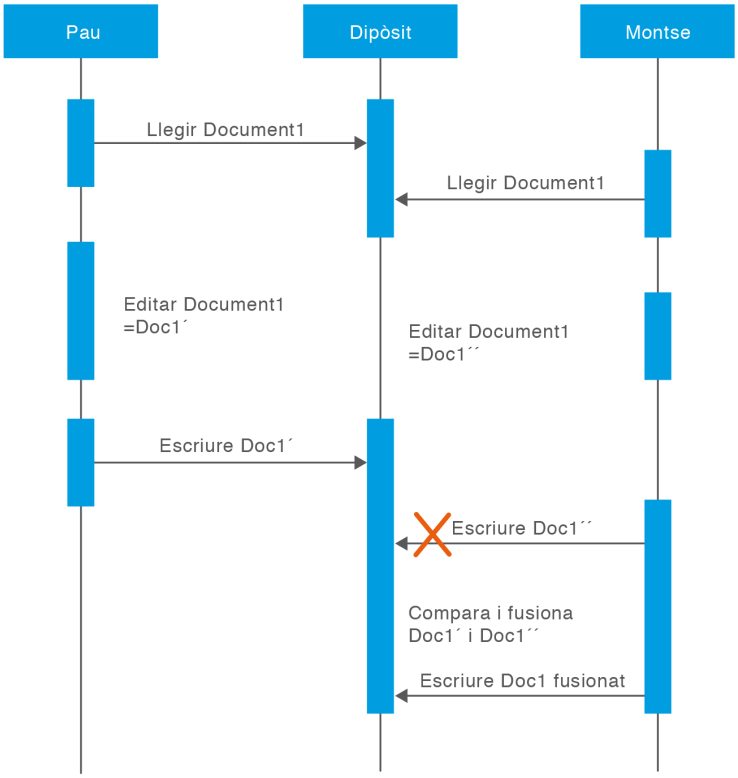
- Cada un dels desenvolupadors que necessiti editar un arxiu s'efectuarà una còpia privada.
- Cada desenvolupador editarà la seva còpia privada de l'arxiu.
- Finalment, es fusionaran les diverses còpies privades de l'arxiu modificat. Aquesta fusió serà automàtica per part del sistema, encara que, finalment, els desenvolupadors són els responsables de donar el vistiplau si la fusió és correcta.

Per entendre bé aquest sistema es pot observar la figura 2.12, on hi ha una explicació gràfica, que es basa en el següent exemple:

En Pau i la Montse creen còpies de treball del mateix projecte, obtingut del repositori. Els dos desenvolupadors treballen simultàniament efectuant canvis al mateix fitxer A en les seves còpies locals:

- La Montse és la primera a enregistrar els seus canvis al repositori.
- Quan en Pau intenta desar els seus canvis, el repositori l'informa que el seu arxiu A no està actualitzat.
- En Pau pot sol·licitar a l'assistent que efectuï una proposta de superposició dels canvis.
- El més segur és que les modificacions efectuades per la Montse i per en Pau no se superposin, per la qual cosa una vegada que tots dos conjunts de canvis s'han integrat, s'enregistra la nova versió de l'arxiu en el repositori.

FIGURA 2.12. Exemple de solució amb fusió



Aquest sistema podrà funcionar de forma senzilla si els canvis de cada un dels dos desenvolupadors no afecten la feina de l'altre. Però què succeeix si els canvis del primer desenvolupador actuen exactament a la mateixa línia de codi font que els canvis del segon? En aquest cas es podria crear un conflicte. Per evitar aquest conflicte, caldrà que el sistema vagi amb molta cura a l'hora de fusionar els dos arxius. Una possible forma d'actuar seria:

- Quan en Pau demana a l'assistent que fusioni els darrers canvis del repositori a la seva còpia de treball, la seva còpia de l'arxiu A marca d'alguna manera que està en un estat de conflicte.
- En Pau serà capaç de veure dos conjunts de canvis conflictius i, manualment, podrà triar entre tots dos o modificar el codi perquè tingui en compte tots dos.
- Una vegada resolts els canvis que se superposaven de forma manual, podrà guardar de forma segura l'arxiu fusionat al repositori.

Aquesta situació s'hauria de donar poques vegades, ja que la majoria dels canvis concurrents no se superposen en absolut i els conflictes no són freqüents. A més a més, moltes vegades el temps que porta resoldre conflictes és molt menor que el temps perdut per un sistema bloquejant.

El sistema de fusió d'arxius és l'utilitzat per algunes eines de control de versions com, per exemple, CVS o Subversion.

D'aquesta forma, concluïm que el punt fort d'aquest sistema és que permet el treball en paral·lel de més d'un desenvolupador o membre de l'equip de treball del projecte, i el punt feble és que no pot ser utilitzat per arxius no fusionables, com podrien ser les imatges gràfiques on cada desenvolupador modifica la imatge per una altra. La bona serà o la primera o la segona, però no es podran fusionar.

2.6 Utilització de Git

Git va ser creat per Linus Torvalds per al desenvolupament del nucli de Linux l'any 2005.

Git és d'un programari de control de versions que utilitza un sistema distribuït i, per consegüent, cada usuari que clona un repositori obté una còpia completa dels fitxers i l'historial de canvis.

Tot i que Git és un sistema distribuït, pot utilitzar-se una còpia del repositori en un servidor de manera que tots els usuaris pugin i baixin els canvis d'aquest repositori per facilitar la sincronització.

Cal tenir en compte que Git és un programari força complex i inclou moltes característiques avançades per a la gestió de revisions i la visualització dels canvis. En aquests materials només es mostren les accions més habituals per poder treballar amb aquesta eina.

2.6.1 Instal·lació

Git acostuma a estar instal·lat a totes les distribucions de Linux.

Git es troba disponible a la majoria de plataformes, incloent Linux, Windows, macOS i Solaris. A la pàgina oficial podeu trobar l'enllaç per descarregar-lo per als diferents sistemes operatius: goo.gl/BGGTMT.

Tot i que Git ofereix una interfície gràfica, en aquests materials s'utilitza mitjançant la línia d'ordres.

El procés d'instal·lació en qualsevol sistema operatiu és molt senzill. En aquesta secció podeu llegir com instal·lar-lo a Windows:

1. Descarregueu l'instal·lador des de l'enllaç següent: goo.gl/pykTra.
2. Un cop finalitzada la descàrrega, feu doble clic sobre l'instal·lador.
3. Apareix la pantalla que mostra la llicència del programari. Feu clic a `Continuar`.
4. Apareix la pantalla que mostra les opcions d'instal·lació. No cal que canvieu res.
5. La següent pantalla ofereix tres opcions. Deixeu marcada l'opció per defecte: utilitzar Git des de la línia d'ordres de Windows (vegeu la figura 2.13).

6. La següent pantalla ofereix dues opcions. Deixeu marcada l'opció per defecte: utilitzar la biblioteca OpenSSL (vegeu la figura 2.14).

7. La següent pantalla mostra tres opcions. Deixeu marcada l'opció per defecte, ja que és l'opció recomanada per Windows per a projectes multiplataforma (vegeu la figura 2.15).

8. La següent pantalla mostra dues opcions: utilitzar un emulador de terminal o fer servir la consola de Windows. Podeu marcar qualsevol de les dues opcions.

9. La darrera pantalla mostra opcions per configurar opcions extres. Deixeu les opcions per defecte marcades (activar el sistema de cau i activar el gestor de credencials de Git) i feu clic a `Instal·lar`.

Figura 2.13. Opcions d'ajustament de la variable PATH de l'entorn

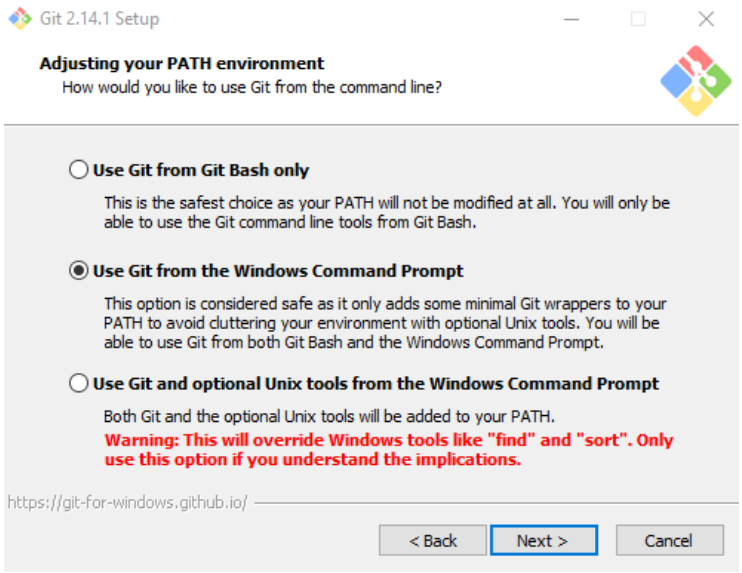


Figura 2.14. Selecció de la biblioteca de transport HTTPS

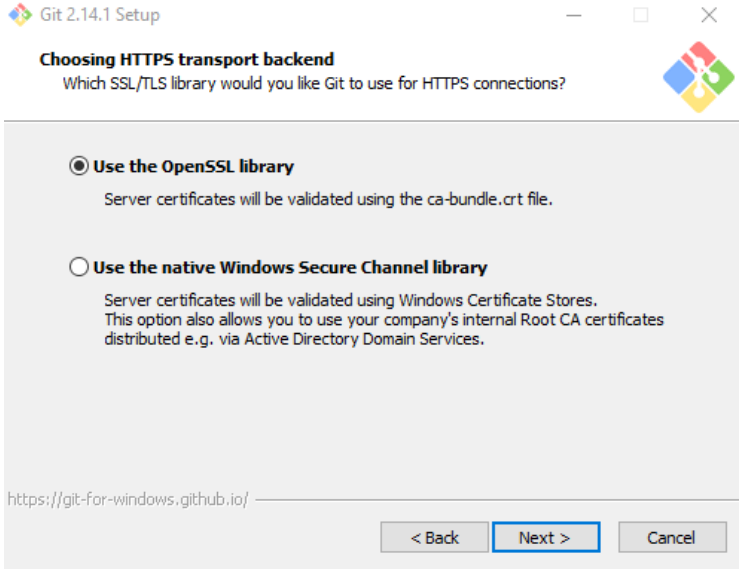
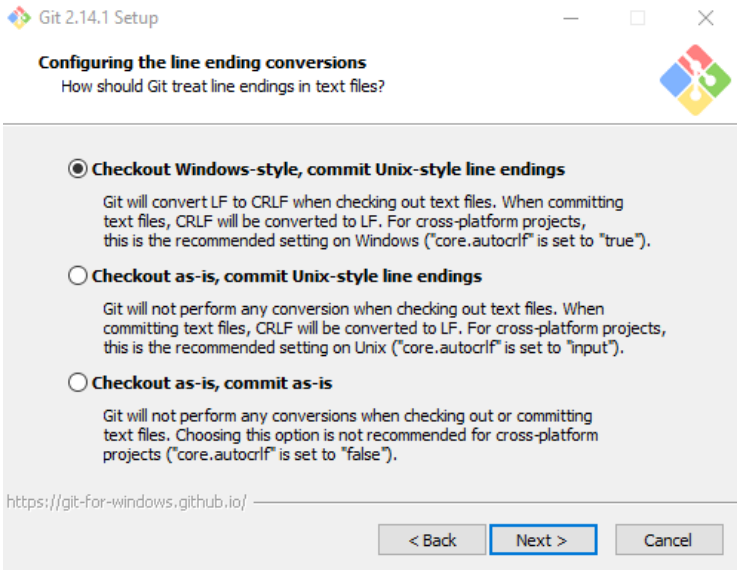


FIGURA 2.15. Configuració del caràcter de final de línia



Un cop instal·lat, obriu la finestra del símbol del sistema (o la terminal, segons el sistema operatiu) i escriviu:

```
git --version
```

El resultat ha de ser similar al següent:

```
git version 2.14.1.windows.1
```

En cas que no funcioni correctament, proveu de reinstal·lar-lo i assegureu-vos que a la pantalla d'opcions per ajustar la variable d'entorn `PATH` heu seleccionat la segona opció: *Use Git from the Windows Command Prompt*.

2.6.2 Operacions bàsiques

El primer pas per començar a treballar amb Git un cop instal·lat és clonar un repositori ja existent o crear-ne un de nou. En aquesta secció es descriu pas a pas com crear un nou repositori i com treballar amb el repositori local i la còpia de treball amb Windows, però en altres sistemes operatius les ordres són les mateixes.

Primer heu d'obrir una finestra del símbol del sistema (o terminal) i crear un directori on s'inicialitzarà el repositori anomenat `proves-git`:

```
md proves-git
```

A Linux i macOS l'ordre serà:

```
mkdir proves-git
```

Entreu dintre del directori amb l'ordre següent:

```
cd proves-git
```

Inicialitzeu el repositori amb l'opció `init` de Git:

```
git init
```

El resultat ha de ser semblant al següent:

```
Initialized empty Git repository in D:/Users/Xavier Garcia/proves-git/.git/
```

Aquesta ordre inicialitza el sistema de control de versions i crea una carpeta oculta (anomenada *.git*) on hi ha tota la informació del repositori. Podeu entrar dintre de la carpeta amb l'ordre:

```
cd .git
```

El contingut de la carpeta serà similar al següent:

```
El volumen de la unidad D es Disco local
 El número de serie del volumen es: 9222-DEE6

 Directorio de D:\Users\Xavier Garcia\proves-git\.git

18/08/2017  18:05               130 config
18/08/2017  18:05                73 description
18/08/2017  18:05                23 HEAD
18/08/2017  18:05    <DIR>          hooks
18/08/2017  18:05    <DIR>          info
18/08/2017  18:05    <DIR>          objects
18/08/2017  18:05    <DIR>          refs
               3 archivos            226 bytes
               4 dirs  1.668.150.059.008 bytes libres
```

Una manera d'eliminar el control de versions d'un projecte és esborrar la carpeta *.git*.

Els fitxers d'aquesta carpeta no s'han de tocar mai, a excepció del fitxer config, que conté la informació del repositori i de vegades cal fer alguna modificació (per exemple, canviar el repositori remot).

Un cop inicialitzat el repositori, el contingut del directori *proves-git* es considera la còpia de treball. Creeu un fitxer dintre d'aquest directori anomenat index.html, amb un editor de text pla amb el següent contingut:

```
<h1>Proves Git</h1>
<ul>
  <li>Pas 1: Inicialitzar el repositori: git init</li>
</ul>
```

Escriviu a la línia d'ordres:

```
git status
```

Aquesta ordre mostra l'estat actual de la còpia de treball. Així, l'ordre anterior us mostrarà un resultat similar al següent:

```
On branch master

No commits yet
```

```
4
5  Untracked files:
6    (use "git add <file>..." to include in what will be committed)
7
8          index.html
9
10 nothing added to commit but untracked files present (use "git add" to track)
```

El nom del fitxer index.html es mostra ressaltat per destacar que aquest fitxer no es troba sota el control de versions. Per afegir un o més fitxers (o directoris) al control de versions es fa servir l'ordre `git add`, indicant com a paràmetre la ruta (admet comodins) dels elements que cal afegir. Per exemple, per afegir tots els fitxers al sistema de control de versions es fa servir l'ordre:

```
1 git add .
```

Quan s'afegeixen fitxers al control de canvis amb `git add`, però encara no s'han pujat els canvis al repositori local es diu que els canvis es troben a l'àrea de *staging*.

Si només voleu afegir els fitxers amb extensió .html, l'ordre ha de ser:

```
1 git add *.html
```

Un cop executada qualsevol de les ordres anteriors, si comproveu l'estat de la còpia de treball el resultat serà similar al següent:

```
1  On branch master
2
3  No commits yet
4
5  Changes to be committed:
6    (use "git rm --cached <file>..." to unstage)
7
8          new file:   index.html
```

Fixeu-vos que el fitxer s'ha afegit al sistema de control de canvis, però encara no s'ha pujat al repositori local.

En cas d'afegir un fitxer per error, podeu eliminar-lo amb l'ordre `git rm -cache`. Per exemple, per eliminar el fitxer anterior del control de versions podeu utilitzar l'ordre:

```
1 git rm index.html --cache
```

Quan s'inicialitza un projecte és habitual fer una primera pujada amb un comentari similar a "Pujada inicial".

Cal tenir en compte que afegir un fitxer al control de versions no el puja al repositori; així, els canvis que es produeixin al fitxer encara no quedaran enregistrats a l'historial de canvis. Per pujar els fitxers al repositori i començar a enregistrar aquests canvis s'ha d'utilitzar l'ordre `commit`:

```
1 git commit -m "Pujada inicial"
```

El paràmetre `-m` serveix per afegir un comentari i s'utilitza per afegir informació extra sobre els canvis que inclou la versió (el conjunt de canvis realitzats). El resultat d'executar l'ordre anterior serà similar al següent:

```
1 [master (root-commit) 33f2ea7] Pujada inicial
2  1 file changed, 4 insertions(+)
3  create mode 100644 index.html
```

Si a continuació comproveu l'estat de la còpia de treball amb l'ordre `git status`, podeu veure que no es detecta res perquè tots els canvis es troben ja a la còpia de treball:

```
On branch master
nothing to commit, working tree clean
```

Per comprovar que es detecten els canvis correctament canvieu el contingut del fitxer index.html pel següent:

```
<h1>Proves Git</h1>
<ul>
  <li>Pas 1: Inicialitzar el repositori: git init</li>
  <li>Pas 2: Afegir tots els fitxers: git add .</li>
</ul>
```

Afegiu un nou fitxer buit anomenat llegeix.me al directori `proves-git`, i comproveu l'estat de la còpia de treball amb `git status`. El resultat serà similar al següent:

```
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   index.html

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        llegeix.me

no changes added to commit (use "git add" and/or "git commit -a")
```

Com podeu apreciar, el missatge indica que s'han detectat canvis al fitxer index.html i que s'ha trobat un fitxer (llegeix.me) que no es troba sota el control de versions.

Afegiu el nou fitxer al control de versions amb l'ordre `git add`:

```
git add llegeix.me
```

Pugeu els canvis al repositori:

```
git commit . -m "Afegit pas 2 i nou fitxer"
```

Fixeu-vos que per pujar els canvis del fitxer index.html s'ha d'afegir un punt (`.`) a les opcions. Això indica que es volen pujar tots els fitxers modificats sota el control de versions, i no només els fitxers afegits. En cas contrari, la versió pujada només inclou el nou fitxer creat i no pas els canvis al fitxer index.html.

Un cop pujada aquesta versió, podeu veure la llista de versions pujades al repositori amb l'ordre `git log`. El resultat serà similar al següent:

```
commit 3f9365181b055c0b956e3bdf97a6e3a37085927f (HEAD -> master)
Author: Xavier Garcia <xaviergaro.dev@gmail.com>
Date:   Fri Aug 18 19:17:53 2017 +0200
```

```
4
5       Afegit pas 2 i nou fitxer.
6
7   commit 33f2ea78a9657bc6ad8660a6e0edea3b68ae02cf
8   Author: Xavier Garcia <xaviergaro.dev@gmail.com>
9   Date:   Fri Aug 18 18:41:18 2017 +0200
10
11      Pujada inicial
```

Configuració de l'adreça de correu a Git

Podeu configurar l'autor i l'adreça de correu utilitzada per Git amb les ordres: `git config -global user.name "Autor"` i `git config -global user.email email@exemple.cat`.

Com podeu apreciar, apareixen les dues versions pujades on s'indica l'identificador de la versió (`commit`), l'indicador de quin és l'últim que s'ha pujat (`HEAD> master`), l'autor i la data de la pujada, seguit del text introduït com a comentari amb l'opció `-m`.

Es recomana pujar els canvis al repositori local de manera freqüent, preferiblement incloent canvis relacionats. Per exemple, si es detecta un error al programari i se soluciona el problema seria adient pujar el canvi abans de continuar afegint una altra característica o solucionant altres problemes. D'aquesta manera en el futur és possible tornar a la versió concreta en la qual es va fer el canvi.

Cal recordar que en cas que la informació ocupi més d'una pantalla no es mostra tot de cop, sinó que el programa en mostra només una part i podreu desplaçar-vos per veure la resta amb les fletxes del teclat. Per sortir, premeu la tecla **q**.

Una altra opció a l'hora de crear repositoris és clonar un repositori existent. Per fer-ho s'utilitza l'ordre `git clone`, indicant l'adreça del repositori que es vol clonar. Per exemple, per clonar el repositori que es troba a l'URL https://github.com/XavierGaro/client-servidor-xat.git l'ordre seria la següent:

```
1   git clone https://github.com/XavierGaro/client-servidor-xat.git
```

Aquesta ordre crea el directori *client-servidor-xat* i descarrega els continguts del repositori mostrant per pantalla els missatges següents:

```
1   Cloning into 'client-servidor-xat'...
2   remote: Counting objects: 45, done.
3   remote: Total 45 (delta 0), reused 0 (delta 0), pack-reused 45
4   Unpacking objects: 100% (45/45), done.
```

En resum, les ordres bàsiques de Git per treballar amb repositoris locals són les següents:

- **git init**: inicialitza un repositori.
- **git add**: afegeix elements de la còpia de treball al control de versions.
- **git rm –cache**: elimina elements del control de versions.
- **git status**: mostra l'estat de la còpia de treball.

- **git commit**: puja els canvis de la còpia de treball sota el control de versions al repositori local.
- **git log**: mostra la llista de versions pujades al repositori local.
- **git clone**: copia un repositori remot, no cal inicialitzar-lo.

2.6.3 Operacions avançades

És molt habitual quan es treballa en control de versions haver de crear noves branques, de manera que al tronc es troba la versió actual del programari i a les branques es desenvolupen noves funcionalitats, es fa el manteniment de versions anteriors o se solucionen errors.

En alguns casos, com la implementació de noves funcionalitats i la solució d'errors, aquestes branques es fusionen amb el tronc un cop s'ha finalitzat amb èxit la tasca.

Per crear una nova branca es fa servir l'ordre `git branch`. Per exemple, dintre del directori *proves-git* (on s'ha inicialitzat prèviament un repositori) escriviu l'ordre següent i es crearà una nova branca amb el nom `nova-branca`:

```
git branch nova_branca
```

No es visualitza cap canvi, però si entreu l'ordre `git branch` veureu la llista de branques al repositori:

```
* master
  nova_branca
```

Com podeu apreciar, es mostra un asterisc al costat de la branca activa (`master`).

Per canviar de branca s'utilitza l'ordre `git checkout`. Així, per canviar a la branca `nova-branca` heu d'utilitzar la següent ordre:

```
git checkout nova_branca
```

Si a continuació proveu d'entrar l'ordre `git branch`, veureu que l'asterisc es mostra al costat de la branca `nova-branca`:

```
master
* nova_branca
```

Per comprovar que els canvis són independents, assegureu-vos de situar-vos a la branca `nova-branca` i editeu el fitxer index.html (o creeu-ne un de nou amb aquest nom) de manera que contingui el codi següent:

```
<h1>Proves Git</h1>
<ul>
  <li>Pas 1: Inicialitzar el repositori: git init</li>
  <li>Pas 2: Afegir tots els fitxers: git add .</li>
```

```
  <li>Pas 4: Crear una branca: git branch nova-branca</li>
</ul>
```

Seguidament pugeu els canvis al repositori amb l'ordre:

```
git commit -m . "Afegit pas 4"
```

Apareix un missatge similar al següent:

```
[nova-branca b4483e3] Afegit pas 4
 1 file changed, 1 insertion(+)
```

Si torneu a la branca `master` amb l'ordre `git checkout master` i editeu el fitxer index.html, podeu veure que el contingut del fitxer index.html és diferent. En canviar de branca, el fitxer a la còpia de treball correspon al de la branca `master` i no al que s'ha modificat anteriorment.

Assegureu-vos que us trobeu a la branca `master` i modifiqueu el fitxer index.html, amb el següent contingut:

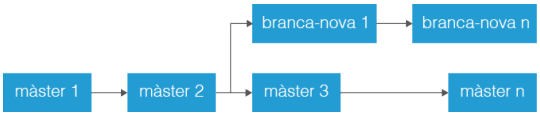
```
<h1>Proves Git</h1>
<ul>
  <li>Pas 1: Inicialitzar el repositori: git init</li>
  <li>Pas 2: Afegir tots els fitxers: git add .</li>
  <li>Pas 3: Pujar els canvis al repositori: git commit . -m "Pujar canvis"</li
      >
</ul>
```

Seguidament pugeu els canvis al repositori amb l'ordre:

```
git commit . -m "Afegit pas 3"
```

Arribats a aquest punt, els continguts de totes dues branques són diferents i a mesura que es pugin més canvis al repositori en una o altra branca s'afegiran al registre propi de cada branca, com podeu veure a la figura 2.16.

FIGURA 2.16. Representació de branques a Git



En el cas que vulgueu integrar els canvis fets en una branca amb el tronc (per exemple, perquè s'ha finalitzat la implementació de la nova funcionalitat o s'ha corregit l'error pel qual es va crear la branca), heu de fer servir l'ordre `git merge`. Per exemple, per fusionar els canvis de la branca `nova-branca` amb la branca `master` s'ha d'utilitzar la següent ordre (essent `master` la branca activa):

```
git merge nova-branca
```

En cas que els canvis de cada branca afectin diferents fitxers no cal fer res més, però com que en aquest cas s'ha modificat el mateix fitxer i les mateixes línies a totes dues branques es produeix un conflicte que cal solucionar abans de continuar.

Per veure on es troba el problema podeu utilitzar l'ordre `git diff`, que mostra un text similar al següent:

```
diff --cc index.html
index ae389e4,e2e54dd..0000000
--- a/index.html
+++ b/index.html
@@@ -2,5 -2,5 +2,9 @@@
  <ul>
    <li>Pas 1: Inicialitzar el repositori: git init</li>
    <li>Pas 2: Afegir tots els fitxers: git add .</li>
++<<<<<<< HEAD
 +  <li>Pas 3: Pujar els canvis al repositori: git commit . -m "Pujar canvis"</
     li>
++=======
+   <li>Pas 4: Crear una branca: git branch nova-branca</li>
++>>>>>>> nova-branca
  </ul>
```

Per solucionar el conflicte heu d'editar el fitxer index.html, on s'hauran afegit les marques `<<<<<<< HEAD` on comença el conflicte i `>>>>>>> nova-branca` on acaba, fer els canvis necessaris i desar el fitxer. El contingut del fitxer index.html una vegada resolt el conflicte ha de ser el següent:

```
<h1>Proves Git</h1>
<ul>
  <li>Pas 1: Inicialitzar el repositori: git init</li>
  <li>Pas 2: Afegir tots els fitxers: git add .</li>
  <li>Pas 3: Pujar els canvis al repositori: git commit . -m "Pujar canvis"</li
      >
  <li>Pas 4: Crear una branca: git branch nova-branca</li>
</ul>
```

Una vegada desats els canvis, pugeu-lo al repositori amb l'ordre:

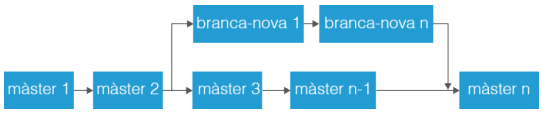
```
git commit -a -m "Conflicte resolt"
```

Fixeu-vos que s'ha fet servir el paràmetre `-a`. Si no es fa així, es considera una pujada parcial i no permet continuar. El missatge d'error que es mostra és el següent:

```
fatal: cannot do a partial commit during a merge.
```

Arribats a aquest punt, s'ha fusionat la branca amb el tronc i la representació del repositori és la que es mostra a la figura 2.17.

Figura 2.17. Branca fusionada amb el tronc a Git



De vegades interessa tornar a una versió anterior del control de canvis. Per fer-ho es pot utilitzar l'ordre `checkout`, però en lloc d'indicar una branca s'indica l'identificador de la versió. Per exemple, si voleu tornar a la versió on es va afegir el pas 2 i el seu identificador és `commit`

`3f9365181b055c0b956e3bdf97a6e3a37085927f`, l'ordre que haureu d'utilitzar és:

```
git checkout 3f93651
```

Com podeu apreciar, no cal entrar l'identificador complet, només el nombre suficient de caràcters, perquè no coincideix amb cap altre identificador de versió. Recordeu que es pot trobar l'identificador corresponent a cada versió amb l'ordre `git log`.

En canviar a una versió anterior es mostra a la terminal un missatge indicant que la còpia de treball es troba en l'estat `detached HEAD` i indica que es pot crear una nova branca a partir d'aquesta versió amb l'ordre `git checkout -b nom-de-la-branca`.

No es recomana fer canvis en aquest estat, ja que és molt fàcil que es produeixin errors que requereixen coneixements avançats de Git per poder-se solucionar. Per evitar aquests problemes, si es volen fer canvis, és millor crear una nova branca a partir de la versió i fer els canvis a la branca.

Per tornar a la versió més actual només cal entrar l'ordre `git checkout`, especificant una branca (per exemple, `master`):

```
git checkout master
```

Git permet afegir etiquetes a les versions de manera que es pot distingir entre les versions habituals i les que representen algun fet especial, com per exemple una versió estable del producte o alguna fita concreta (la inclusió d'alguna funcionalitat important).

Per afegir una etiqueta a l'última versió afegida al repositori s'utilitza l'ordre `git tag -a`. Per exemple:

```
git tag –a v1.0 –m "Primera versió"
```

L'opció `-a` indica el text de l'etiqueta i l'opció `-m` permet afegir informació addicional.

Per llistar totes les etiquetes d'un repositori s'utilitza l'ordre `git tag` sense cap opció. A partir d'aquesta llista, es pot fer servir l'ordre `git show` per mostrar la informació referent a la revisió corresponent a l'etiqueta. Per exemple, l'ordre `git show v1.0` mostra un text similar al següent:

```
tag v1.0
Tagger: Xavier Garcia <xaviergaro.dev@gmail.com>
Date:   Fri Aug 18 21:06:17 2017 +0200

Primera versió

commit 7a7edead12a3f7f33e1bebc692cc868fa534f18f (HEAD -> master, tag: v1.0)
Merge: 3f2e774 b4483e3
Author: Xavier Garcia <xaviergaro.dev@gmail.com>
Date:   Fri Aug 18 20:39:31 2017 +0200

    Conflicte resolt
```

```
13
14 diff --cc index.html
15 index ae389e4,e2e54dd..31d9d62
16 --- a/index.html
17 +++ b/index.html
18 @@@ -2,5 -2,5 +2,6 @@@
19   <ul>
20     <li>Pas 1: Inicialitzar el repositori: git init</li>
21     <li>Pas 2: Afegir tots els fitxers: git add .</li>
22  +  <li>Pas 3: Pujar els canvis al repositori: git commit . -m "Pujar canvis"</
       li>
23 +   <li>Pas 4: Crear una branca: git branch nova-branca</li>
24   </ul>
```

Cal destacar que l'ordre `git show` es pot utilitzar directament amb un identificador de versió i mostraria pràcticament la mateixa informació (sense les dades referents a l'etiqueta). Per exemple:

```
1 git show 7a7ed
```

S'ha de tenir en compte que un repositori pot comptar amb centenars de revisions; això fa que trobar una revisió concreta només pel nombre de revisió sigui força complicat. En canvi, si la revisió que cerqueu està etiquetada trobar-la és molt ràpid. De vegades pot interessar descartar tots els canvis realitzats a la còpia de treball i tornar a la revisió actual. En aquest cas es pot fer servir l'ordre:

```
1 git reset --hard
```

Aquesta ordre descarta tots els canvis i tots els fitxers sota el control de canvis són restablerts. Aquesta acció és coneguda com a "revertir els canvis" en alguns entorns.

Quan es treballa amb repositoris remots (per exemple, quan es clona un repositori) es poden descarregar els canvis que s'han portat a terme en el repositori remot amb l'ordre `git pull`. Aquesta ordre descarrega els canvis i fa una fusió automàtica amb la còpia de treball. Aquesta acció pot provocar conflictes que s'han de resoldre abans de poder pujar els canvis al servidor.

En el cas de voler pujar els canvis del repositori local al repositori remot l'ordre que s'ha d'utilitzar és `git push`. Per executar aquesta ordre primer heu de fer un *pull* per fusionar els canvis al servidor remot amb la còpia de treball. En cas contrari, si s'han produït canvis al repositori remot, l'ordre *push* és rebutjada.

Per afegir un repositori remot heu de fer servir l'ordre `git remote add`, indicant el nom del repositori remot i l'URL corresponent. Per exemple:

```
1 git add remote add xat https://github.com/xaviergaro/client-servidor-xat
```

Aquesta ordre afegeix com a repositori remot l'URL `https://github.com/xaviergaro/client-servidor-xat` amb *xat* com a nom curt. Cal destacar que és possible configurar múltiples repositoris remots, però s'ha de tenir molt de compte a l'hora de sincronitzar-los, ja que si no s'automatitza aquesta tasca és possible oblidar fer la sincronització de tots els repositoris quan es produeixen canvis.

Per fer un *push* al servidor remot heu d'indicar el repositori d'origen (per al repositori local és `origin`) i el repositori de destí. Per exemple, per fer un *push* des del repositori local al repositori remot *xat* l'ordre seria la següent:

```
git push origin xat
```

Per fer un *pull* només cal indicar el nom curt del repositori remot:

```
git pull xat
```

Algunes aplicacions inclouen plantilles per generar els fitxers .gitignore amb la selecció de fitxers i directoris més habituals segons el llenguatge utilitzat.

Habitualment cal ignorar determinats fitxers per evitar que s'afegeixin al sistema de control de versions. Per exemple: fitxers del sistema operatiu, fitxers generats per l'entorn de desenvolupament, fitxers de configuració amb contrasenyes, etc.

Per no haver de preocupar-vos quan feu servir l'ordre `git add .`, podeu afegir aquestes exclusions al fitxer .gitignore. El format d'aquest fitxer es molt senzill: només cal indicar el nom dels fitxers o directoris que es volen ignorar. Per exemple:

Podeu trobar la documentació completa sobre ignorar fitxers o directoris al següent enllaç: goo.gl/CTkwC7.

```
# Diaris
logs
*.log

# Dependències
node_modules
```

Com podeu apreciar, el fitxer .gitignore amb el contingut anterior ignoraria els directoris *logs*, *node_modules* i tots els fitxers amb extensió log. El símbol # s'utilitza per afegir comentaris i són ignorats per Git.

En resum, les ordres avançades que s'han vist en aquesta secció són:

- **git branch**: crea noves branques o llista les branques del repositori.
- **git checkout**: canvia la còpia de treball a la branca o versió indicada.
- **git diff**: mostra els canvis que s'han afegit en una versió.
- **git log**: mostra la llista de versions per la branca activa.
- **git merge**: fusiona els canvis entre dues branques.
- **git tag**: afegeix una etiqueta a una versió.
- **git show**: mostra informació sobre la versió indicada.
- **git reset –hard**: reverteix els canvis a la còpia de treball.
- **git pull**: puja els canvis del repositori local a un repositori remot.
- **git push**: baixa els canvis d'un repositori remot al repositori local.
- **git remote**: afegeix un repositori remot o llista els repositoris remots enllaçats amb el repositori local.

- **fitxer .gitignore**: permet afegir una llista de fitxers i directoris per excloure del sistema de control de versions.

Hi ha clients gràfics que ens permeten treballar amb Git sense necessitat de teclejar les ordres, com SmartGit, GitKraken o SourceTree, que és gratuït.

2.6.4 Integració amb entorns de desenvolupament integrats: Eclipse

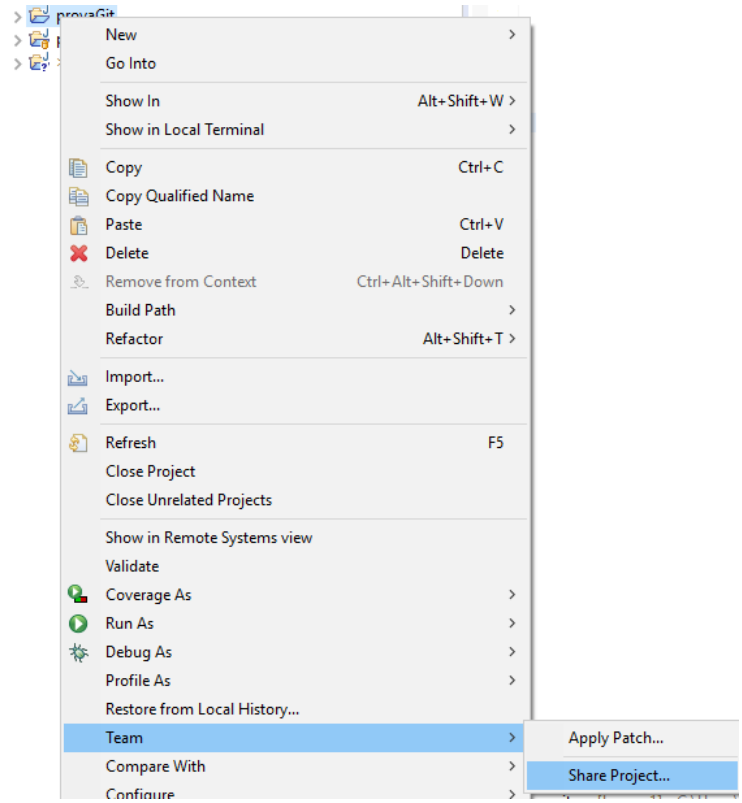
Tots els entorns de desenvolupament integrats (i alguns editors de text per a desenvolupadors) inclouen l'opció de gestionar el control de versions dintre del mateix programa. En aquesta secció es descriu com integra Eclipse el control de versions amb Git.

A Eclipse, la gestió de versions amb Git va incorporada amb el connector EGit. Aquest connector queda instal·lat per defecte en instal·lar Eclipse.

Creació del repositori

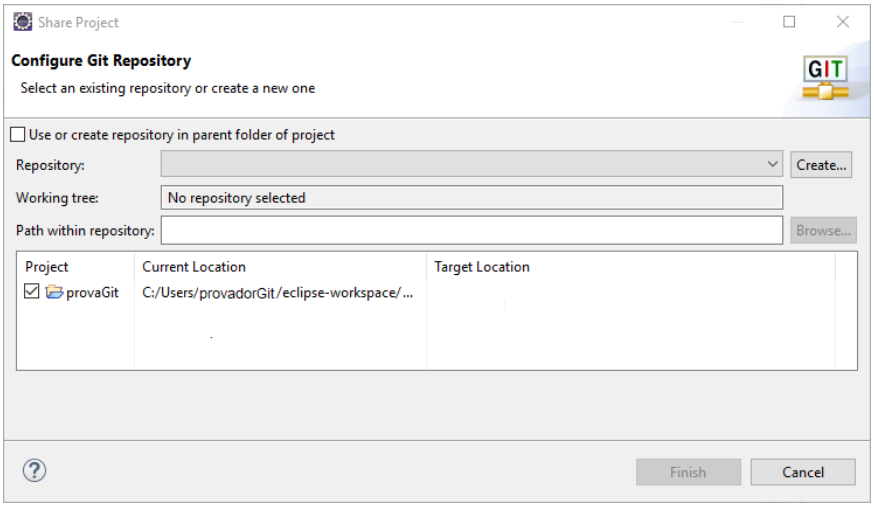
A Eclipse (amb el connector EGit), per inicialitzar un repositori cal fer clic amb el botó secundari a sobre de la carpeta que representa el projecte que volem introduir al repositori i seleccionar les opcions *Team* / *Share Project...*, com es veu a la figura 2.18.

FIGURA 2.18. Opció 'Share'



Ens apareixerà la finestra que es mostra a la figura 2.19.

FIGURA 2.19. Configuració del repositori

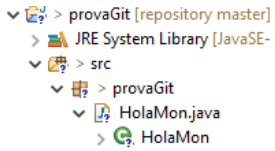


En aquest cas no hi ha cap repositori ja definit i n'hem de crear un de nou. Si ja n'existís un, podriem seleccionar-lo. Per crear el nou repositori, clicarem al botó *Create...* . Ens sortirà una finestra per demanar el directori o carpeta on anirà el repositori que estem creant. Caldrà completar el camí d'aquesta carpeta i, a continuació, clicar al botó *Finish* d'aquesta finestra. Tornarem novament a la pantalla que ens mostra la figura 2.19, però amb les dades *Repository* i *Working tree* omplertes. En clicar novament al botó *Finish*, haurem creat el repositori i hi haurem afegit el projecte actual.

L'opció "Use or create repository in parent folder of project" que apareix a la finestra de la figura 2.19 i que no hem seleccionat força la creació del repositori a la mateixa carpeta del nostre projecte, però té l'inconvenient que ja no es podrien afegir més projectes al repositori així creat.

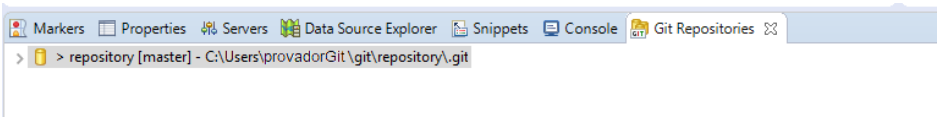
Un cop creat el nou repositori amb el nostre projecte a dins, ens apareixen interrogants a tot allò que està pendent de confirmar, tal com es mostra a la :figura 2.20, que és una captura de l'explorador dels projectes:

FIGURA 2.20. Explorador amb elements pendents de confirmar



També ens ha d'aparèixer la finestra amb els repositoris Git (habitualment a la zona inferior dreta de la finestra), tal com es mostra a la figura 2.21.

FIGURA 2.21. Finestra amb els repositoris Git

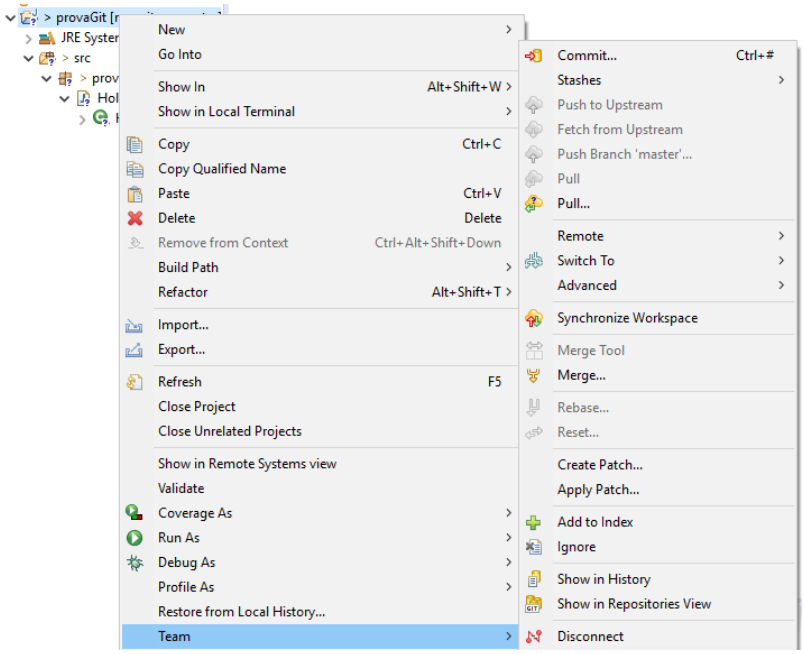


Si no apareix aquesta finestra, es pot obrir amb l'opció del menú *Window* / *Show view* / *Other...* / *Git* / *Git repositories*.

Gestió bàsica del repositori

Un cop creat el repositori, des de l'explorador dels projectes, es pot accedir a totes les accions relacionades amb Git clicant amb el botó secundari del ratolí a sobre de qualsevol dels elements del projecte o, també, a sobre del propi projecte i seleccionant l'opció *Team* del menú contextual que apareix. Aquest menú es veu a la :figura 2.22.

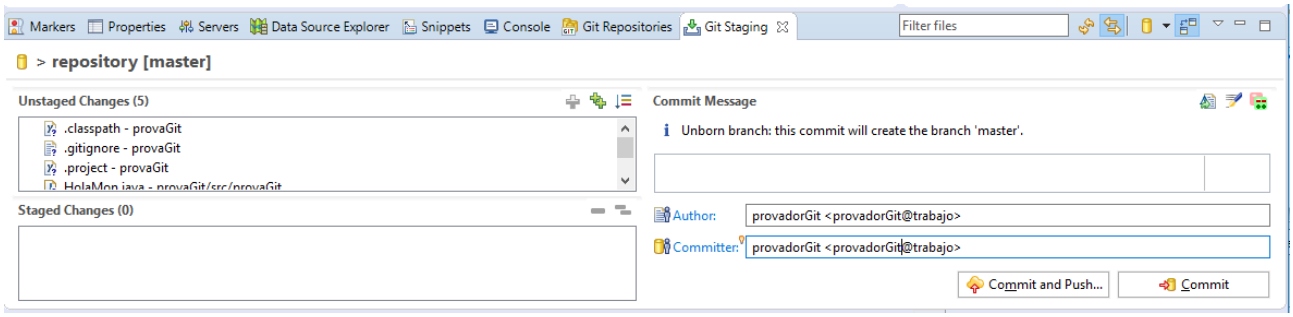
FIGURA 2.22. Opció Team del menú contextual



També apareix un menú contextual similar si es clica amb el botó secundari a sobre del repositori que apareix a la finestra *Git repositories*.

Si seleccionem l'opció *commit...* (que fa el que el seu nom indica), ens apareix la finestra *Git Staging*, com es veu a la figura 2.23.

FIGURA 2.23. Finestra 'Staging'



En aquesta finestra, a la llista *Unstaged Changes*, apareixen tots els fitxers que no volem confirmar. Inicialment hi són tots els fitxers modificats des del darrer *commit* o tots els fitxers si no hem fet encara cap *commit* des de la creació del repositori. D'altra banda, a la llista *Staged Changes* apareixen els fitxers que volem confirmar. Inicialment, no n'hi ha cap. Caldrà, doncs, moure tots els fitxers que vulguem confirmar des de la primera llista a la segona. Això podem fer-ho seleccionant-los i, a continuació, fent clic a la icona amb forma de creu que apareix a sobre de la llista *Unstaged Changes*, al costat dret. També podem moure'ls tots de cop si fem directament clic a la icona que representa dues creus. Si volguéssim fer tornar algun fitxer cap a la llista superior, a sobre de la llista *Staged Changes* també hi ha dues icones que permetran fer-ho: una representa un signe menys i una segona que representa dos signes menys; serveixen, respectivament, per passar els elements seleccionats o tots els elements a la llista *Unstaged Changes*.

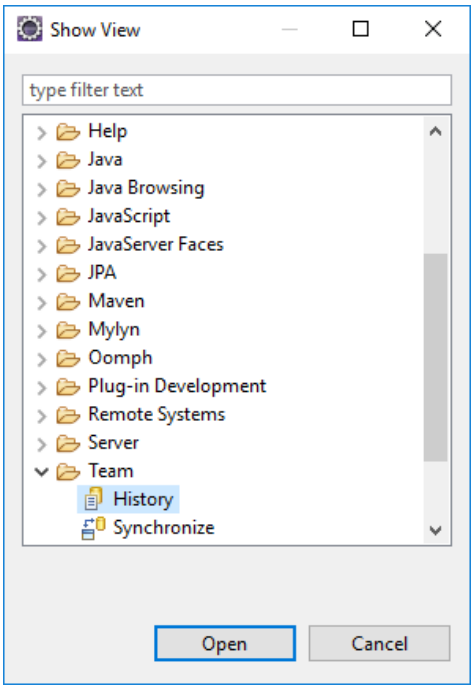
Es pot observar també que els elements de la llista *Unstaged Changes* mantenen un interrogant a la icona que els representa, mentre que als de la llista *Stagged changes* aquest interrogant ha estat substituit per una creu.

En aquest punt, abans de fer el *commit*, ja només resta escriure el comentari associat a aquesta operació. S'escriu al quadre de text de sota de l'etiqueta *Commit message*. Per exemple, podem escriure "versió inicial" com a comentari.

Ara podem clicar al botó *Commit* per acabar de realitzar aquesta operació. Després del *commit* veureu, a l'explorador dels projectes, que s'han substituit totes les creus per cilindres. Aquests cilindres són el símbol habitual de les bases de dades i, en aquest cas, el significat és que Git ha indexat aquests elements. És a dir, en certa manera els ha inclòs a la seva base de dades.

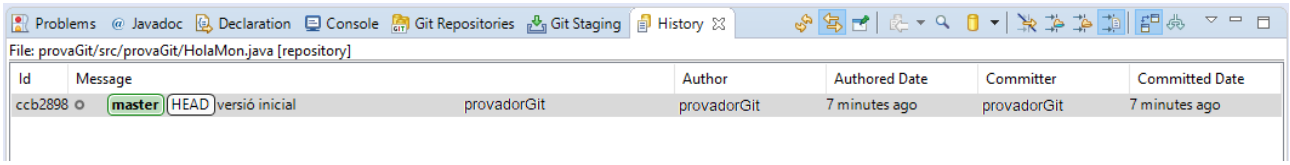
Una altra operació de Git que ens permet realitzar Eclipse és consultar l'historial de les versions que hem confirmat del nostre projecte. Per fer-ho, cal seleccionar *History* a la finestra que s'obre amb l'opció del menú *Window* / *Show view* / *Other...*, tal com es veu a la figura 2.24.

Figura 2.24. Opció 'History'



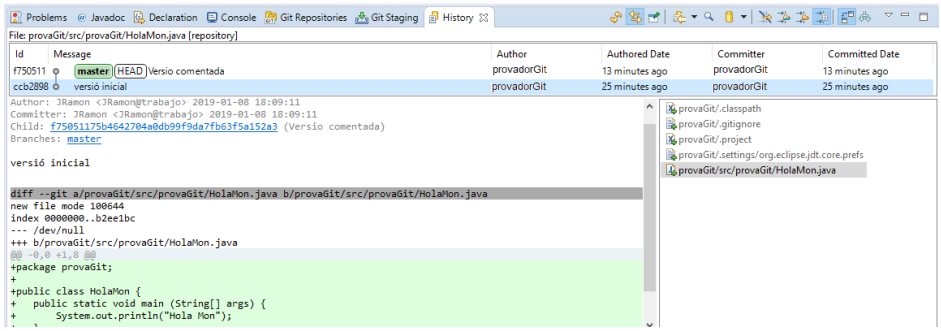
S'obrirà la finestra *History*, que té l'aspecte que apareix a la figura 2.25.

Figura 2.25. Finestra 'History'



En aquesta finestra, si fem un canvi al codi i tornem a confirmar -per exemple, posant-hi el missatge "Versió comentada"-, la finestra ens mostrarà l'historial de les versions de la branca actual. A més, podrem explorar els fitxers de cada versió i se'ns mostraran les diferències, com es veu a la figura 2.26.

Figura 2.26. Exploració de versions



Si cliquem amb el botó secundari a sobre de la representació del nostre projecte a l'explorador dels projectes, trobarem l'opció *Team*, que permet triar i executar altres opcions de Git. Per exemple:

- *Switch to:* permet canviar-nos de branca i crear-ne una de nova.

- *Advanced:* permet esborrar o canviar el nom a les branques.

- *Merge...:* permet fusionar branques.

- *Rebase... :* permet fusionar branques i, a la vegada, gestionar l'historial de les branques que es fusionen.

Operacions amb un repositori remot

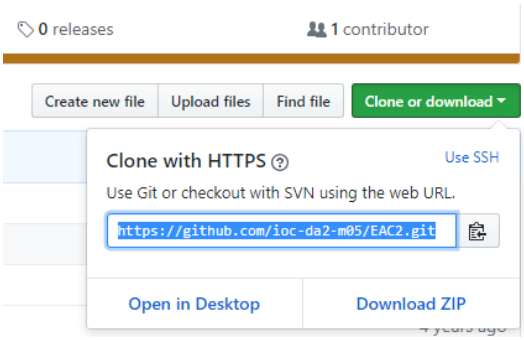
Habitualment voldrem treballar des d'Eclipse amb un repositori remot. Les operacions principals a fer amb ell són *pull* i *push*. Poden realitzar-se en qualsevol ordre, segons convingui al desenvolupador.

All nostre cas utilitzarem GitHub com a repositori remot. GitHub (github.com) és el servei d'allotjaments de repositoris Git més popular. Ofereix integració amb múltiples eines de desenvolupament.

Com trobar la URI del repositori GitHub

Per treballar amb el repositori remot és necessari conèixer el seu URI. Al cas de GitHub, s'obté seleccionant el repositori i clicant el botó *Clone or download*, que apareix a la part superior dreta de la finestra. A la figura 2.27 s'ha accedit a l'URI i s'ha seleccionat, per exemple, per copiar-la al porta-retalls.

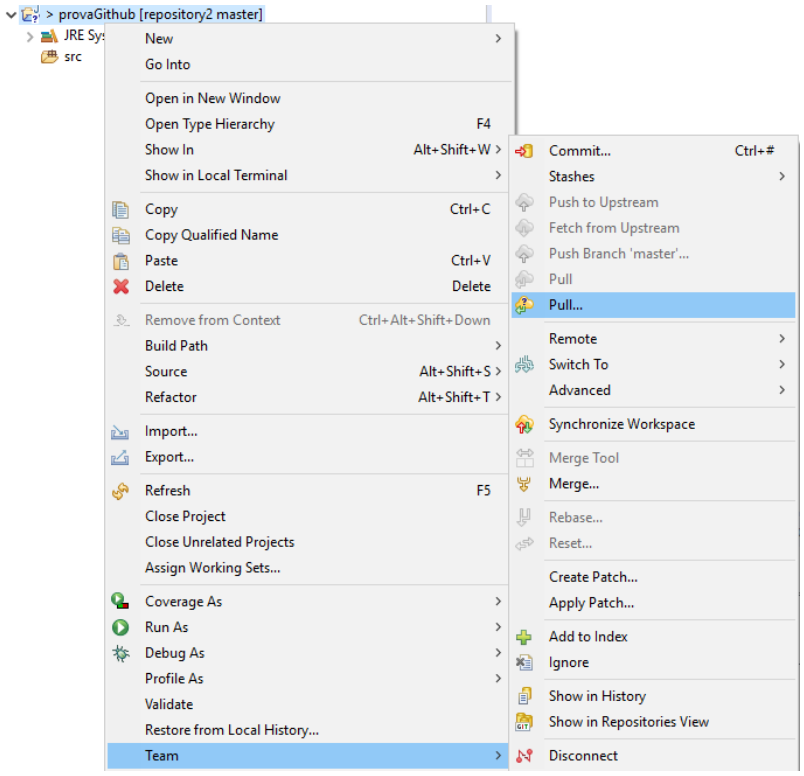
FIGURA 2.27. URI de GitHub



Operació pull

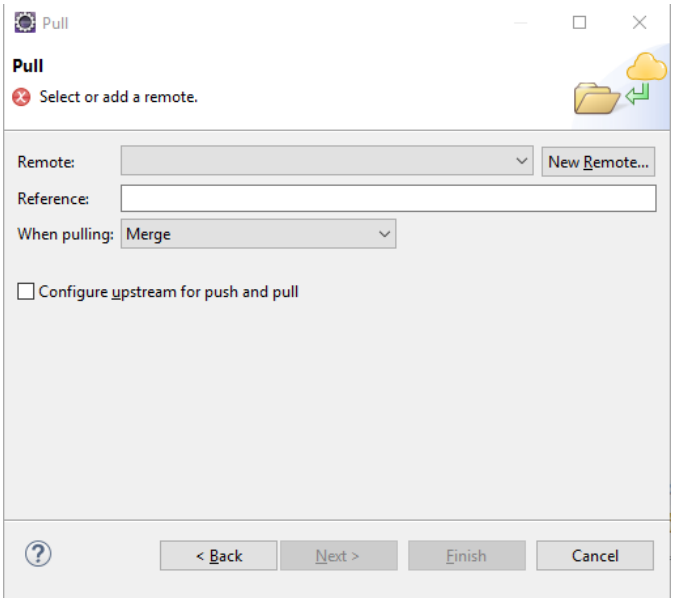
Si volem fer l'operació *pull* per fusionar un repositori remot amb el repositori local, haurem de clicar amb el botó secundari a sobre del nostre repositori local i seleccionar l'opció *Team* / *Pull...* com es veu en la figura 2.28.

FIGURA 2.28. Menu 'Pull'



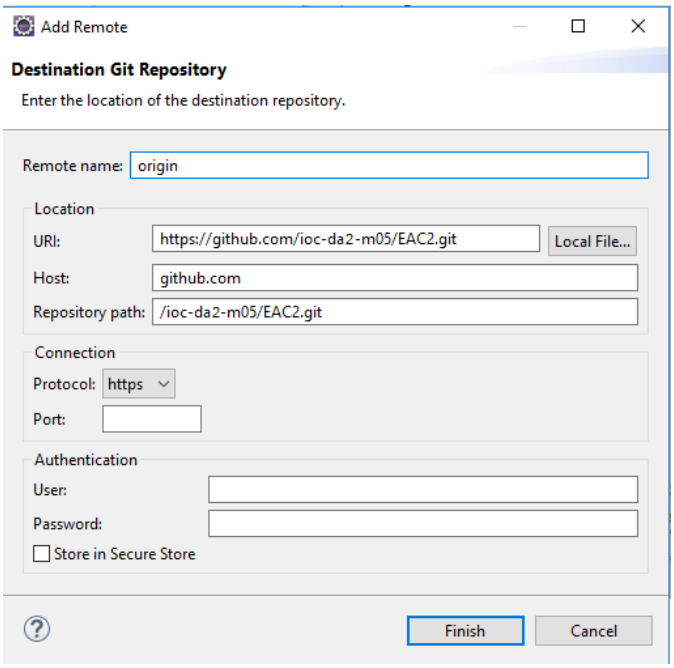
Ens apareixerà la pantalla mostrada a la figura 2.29.

FIGURA 2.29. Opció 'Add' del menú 'Pull'



En ella, cal clicar el botó *New Remote...* per definir el repositori remot a la pantalla com es veu a la figura 2.30.

Figura 2.30. Definició del repositori remot



Cal omplir:

- *Remote name*: nom amb el qual ens referirem, des del nostre entorn, al repositori remot.

- *URI*:identificador del repositori a Internet.

Al nostre cas no cal posar la contrasenya perquè el repositori que utilitzem de GitHub és públic (ho són tots els repositoris gratuïts d'aquest servidor).

Un cop introduïda aquesta informació, s'ompliran automàticament les dades corresponents al *host* i el *Repository path* (camí del repositori).

Un cop cliquem el botó *Finish* tornarem a la pantalla anterior, on caldrà omplir, com a mínim, la dada *Reference*. Aquesta dada conté la branca que volem sincronitzar. Pot seleccionar-se directament clicant abans i de manera simultània les tecles *Ctrl* i espai, com es veu a figura 2.31 i figura 2.32.

FIGURA 2.31. Camp 'Reference' a la finestra 'Pull'

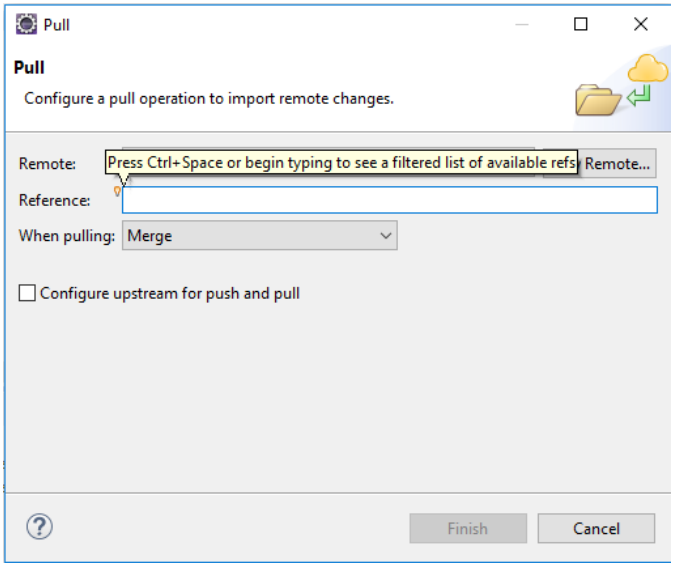
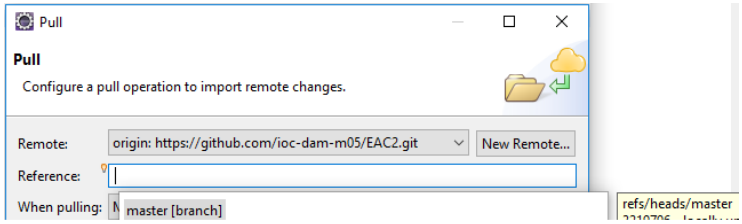


FIGURA 2.32. Desplegable de 'Reference' a la finestra 'Pull'



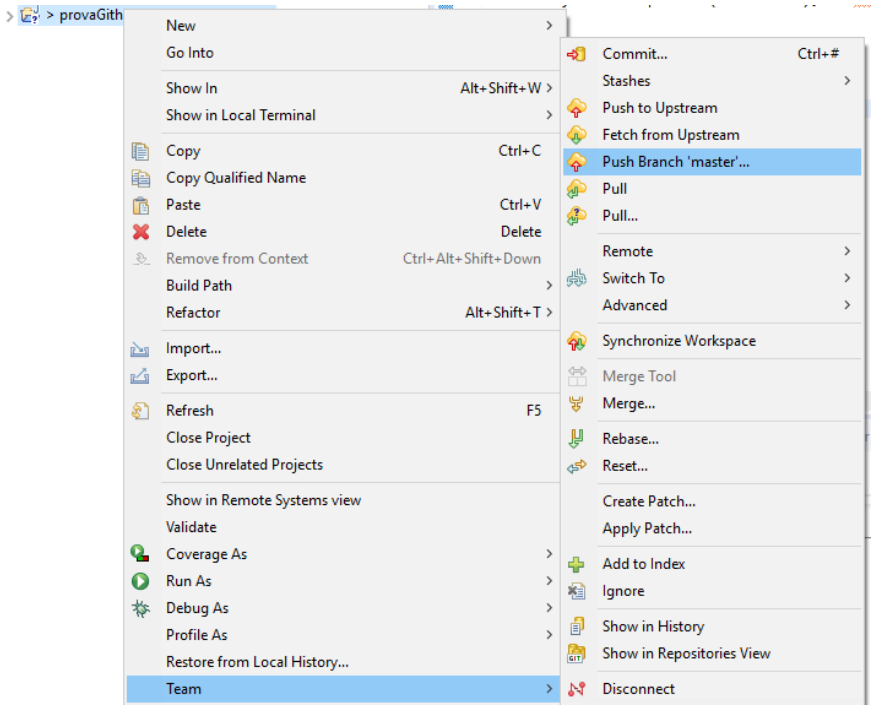
Un cop seleccionada la branca, s'activa el botó *Finish*. En clicar aquest botó, es realitzarà l'operació *push*. Abans de clicar aquest botó, podem triar al quadre de text *When pulling* com volem que es fusionin les branques remota i local .

Operació push

L'operació *push* es realitza de manera semblant. Requereix, però, que a la branca que es tindrà en compte per fer-lo s'hagi fet algun *commit* en local des que es va fer el *pull* des del servidor.

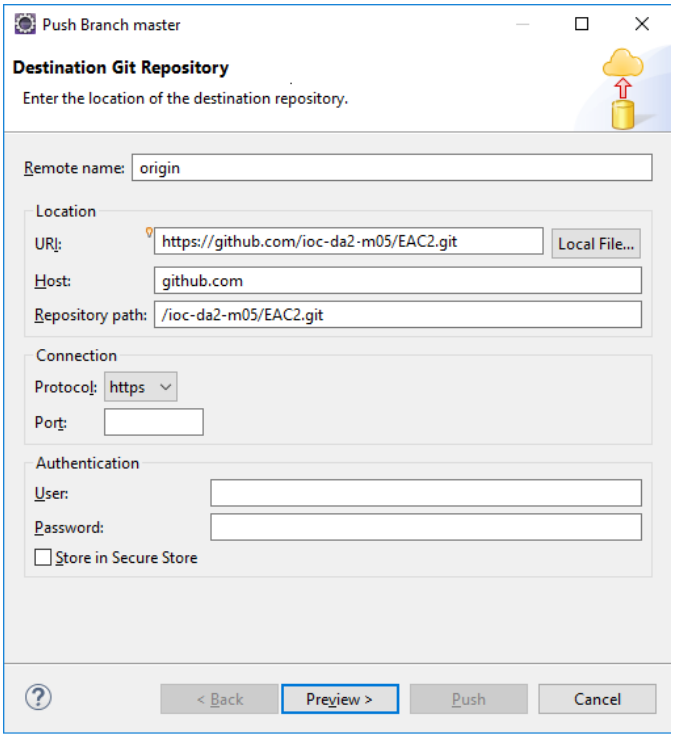
En primer lloc cal seleccionar del menú contextual l'opció *Push Branch*, que conté, al final, el nom de la branca actual entre comentes simples, com es veu a la figura 2.33. Aquesta branca és la que es tindrà en compte per fer el *push*.

Figura 2.33. Menú 'Push'



S'obrirà una finestra semblant a la que es mostra a la figura 2.34.

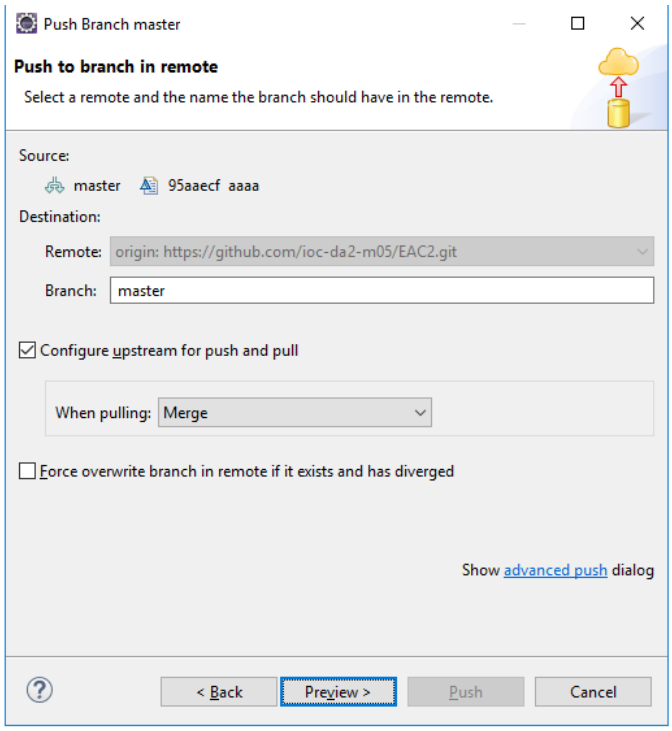
Figura 2.34. Finestra amb el repositori destí



Aquesta finestra només surt el primer cop que utilitzem el repositori. Cal omplir-hi les dades *Remote name* i *URI*. *Host* i *Repository Path* s'ompliran automàticament.

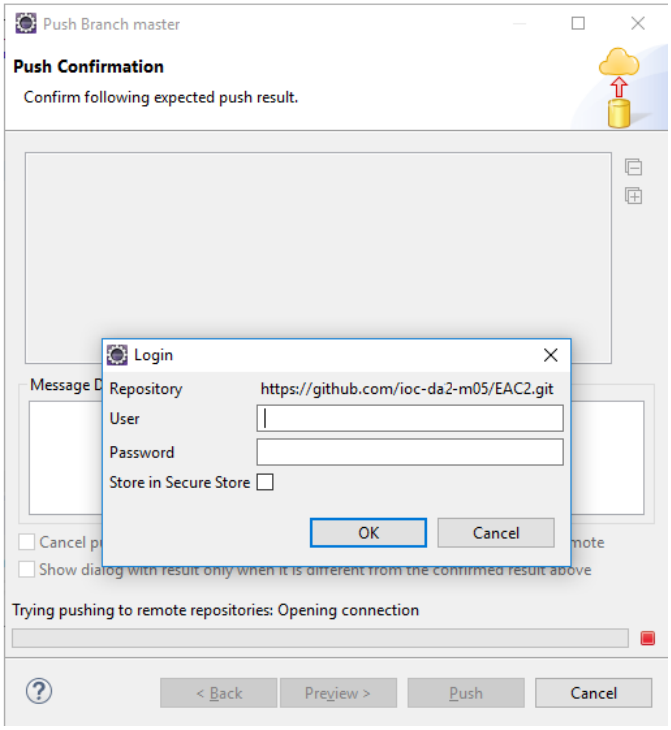
Per passar a la següent pantalla, cal clicar el botó *Preview >*. Ens apareixerà una finestra com la mostrada a figura 2.35.

FIGURA 2.35. Finestra de selecció de la branca



Podem deixar els valors per defecte i fer clic novament al botó *Preview >*. Ens apareixerà una pantalla com la de la figura 2.36.

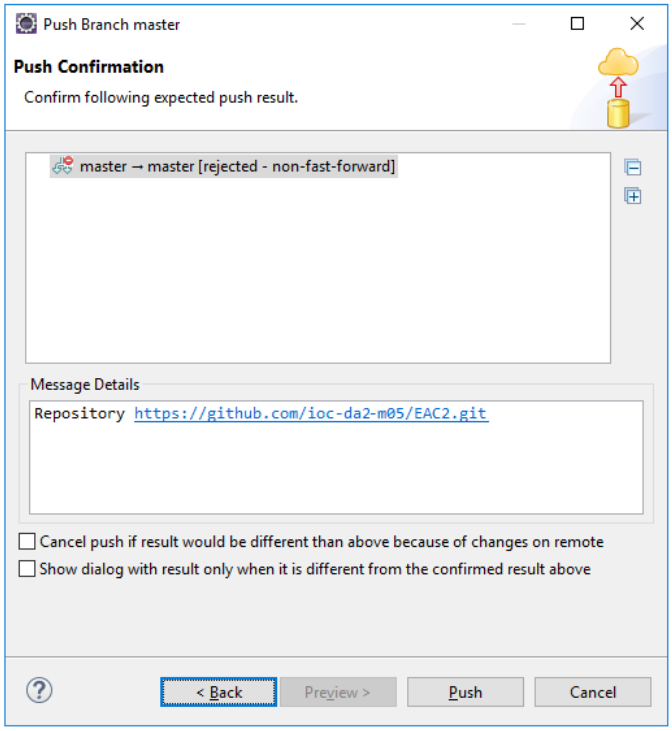
FIGURA 2.36. Usuari i contrasenya del repositori remot



Cal que entrem l'usuari del GitHub i la seva contrasenya i que cliquem, a continuació, el botó *Ok*.

Ens apareix una finestra semblant a la de la figura 2.37, on se'ns mostra la branca a fusionar i l'URI del repositori.

FIGURA 2.37. Confirmació del 'push'

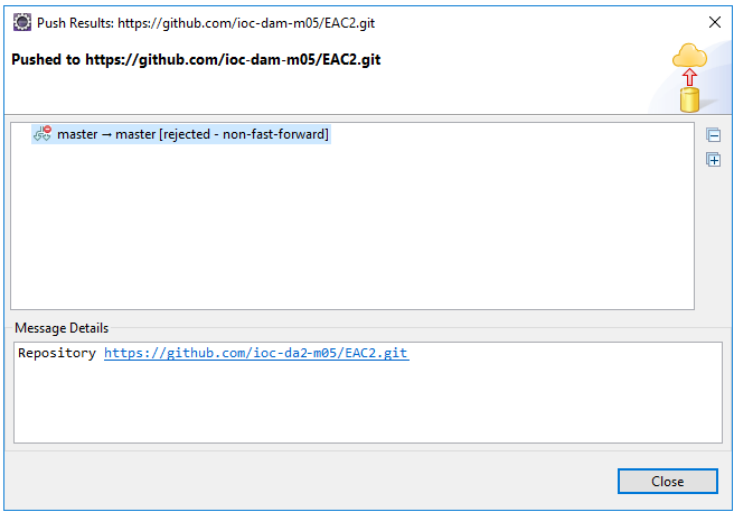


Ens tornarà a demanar usuari i contrasenya. Un cop entrada, ens apareix ja l'última pantalla on se'ns confirma l'operació i que es mostra a la figura 2.38.

Emmagatzematge de l'usuari i la contrasenya

Per evitar tantes peticions d'usuari i contrasenya podem marcar l'opció *Store in Secure Store*. D'aquesta manera, l'entorn emmagatzemarà aquestes dades i les subministrarà automàticament al servidor quan calgui. A més i de manera opcional proporciona mecanismes de recuperació de la contrasenya.

FIGURA 2.38. Confirmació del 'push'



Clicant al botó *Finish* es tanca la finestra.

2.7 Utilització de Github

GitHub (github.com) és un servei d'allotjament de repositoris Git que compta amb més de 10 milions d'usuaris. Ofereix tota la funcionalitat de Git, a més d'oferir

serveis propis com són l'edició de fitxer en línia, la gestió d'errors, possibilitat de documentar els projectes mitjançant una wiki inclosa al repositori o la gestió d'usuaris.

Hi ha dos tipus de repositoris a GitHub:

- Públics: tothom pot visualitzar-los i descarregar-los, sense necessitat de crear un compte a GitHub. Aquests repositoris són gratuïts, i qualsevol usuari registrat pot crear-los.
- Privats: només els membres de l'equip i els usuaris amb permisos poden visualitzar, baixar i pujar canvis al repositori. Aquests repositoris estan limitats als comptes de pagament o d'estudiant (requereixen una adreça de correu universitari vàlida).

Les wikis incloses als repositoris de GitHub compten amb el seu propi control de versions i poden clonar-se mitjançant Git.

GitHub inclou característiques de xarxa social com ara notificacions, llistes de seguidors, opció de subscriure's als repositoris per fer un seguiment dels canvis o marcar repositoris com a favorits. Tot i que la plataforma no proporciona cap sistema de missatgeria entre usuaris, alguns usuaris afegeixen la seva adreça de correu electrònic al seu perfil i és possible comunicar-se amb els administradors d'un repositori mitjançant el sistema de gestió d'errors (*issues*) o la wiki que es pot incloure al repositori.

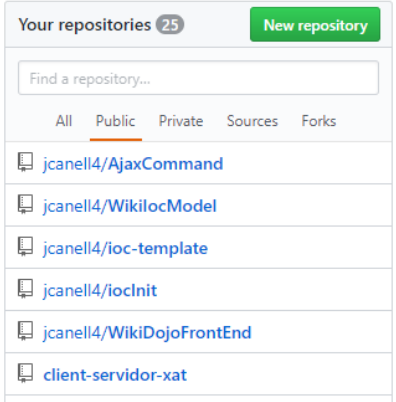
Per treballar amb GitHub es pot utilitzar la línia d'ordres de Git, es pot descarregar algun client gràfic com SourceTree o Github Desktop (desktop.github.com) o es pot treballar directament des d'un IDE integrat amb Git.

2.7.1 Gestió de repositoris privats i públics

Per començar a treballar amb els repositoris de GitHub com a repositoris remots cal crear un compte a la plataforma. En cas contrari, es poden clonar els repositoris públics però no es poden pujar els canvis.

Un cop creat un compte, podeu crear nous repositoris des de la pàgina fent clic al botó *New repositori* (vegeu la figura 2.39).

Figura 2.39. Llistat de repositoris propis

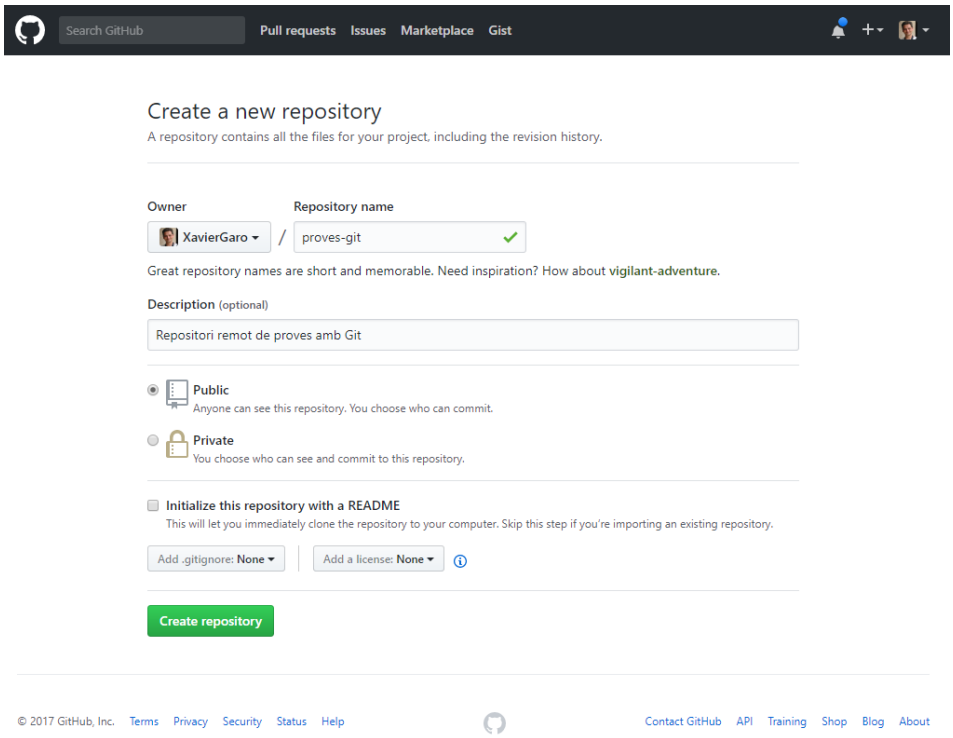


Seguidament heu d'indicar el nom del repositori, la descripció i el tipus (públic o privat), com en la figura 2.40. Addicionalment podeu inicialitzar amb un fitxer README, que es mostra a la primera pàgina del repositori.

L'opció de crear repositoris privats no es troba disponible als comptes gratuïts.

El fitxer README admet el format Markdown i acostuma a incloure informació detallada sobre el repositori i instruccions d'instal·lació. En cas de crear un repositori a GitHub, per sincronitzar-lo amb un repositori local ja existent no marqueu la casella per afegir el fitxer README, ja que en sincronitzar els repositoris es produeix un conflicte.

Figura 2.40. Creació d'un repositori a GitHub



Un cop creat el repositori GitHub, apareix una pàgina on s'indica com sincronitzar el repositori de GitHub amb el vostre repositori local. Per una banda, indica com crear un nou repositori si comenceu un projecte de nou i com sincronitzar-lo amb GitHub:

```
echo "# proves-git" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin https://github.com/nom-usuari/nom-repositori.git
git push -u origin master
```

En cas que vulgueu pujar un repositori ja existent, indica com afegir el repositori remot i com pujar els canvis:

```
git remote add origin https://github.com/nom-usuari/nom-repositori.git
git push -u origin master
```

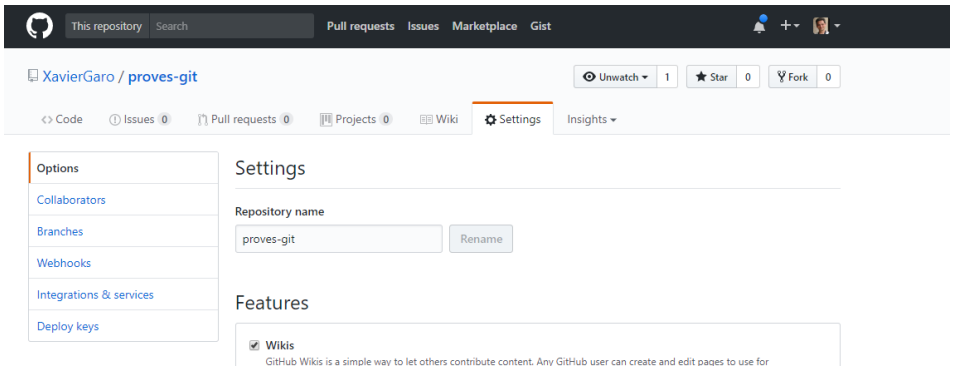
Fixeu-vos que l'URL del repositori que mostra GitHub és la del repositori que heu creat al vostre compte, així que podeu copiar directament el codi a la línia d'ordres.

Un cop s'ha afegit el repositori remot com a origen, no cal que especifiqueu el nom en les ordres `git push` o `git pull`, ja que automàticament fa la sincronització amb aquest repositori.

2.7.2 Configuració d'un repositori de GitHub

Cada repositori de GitHub té la seva pròpia configuració individual (botó *Settings*). Vegeu-ho en la figura 2.41:

FIGURA 2.41. Creació d'un repositori a GitHub



A la secció d'opcions generals hi ha els següents apartats:

- **Quines característiques es volen activar**: activar la wiki, restriccions d'editors, gestió d'errors i gestió de projectes.
- **Com es volen fusionar els canvis**: llista diferents mètodes de fusió de canvis.
- **Limitació temporal d'interacció**: permet imposar una limitació d'interacció temporal amb el repositori per diferents tipus d'usuaris.
- **GitHub Pages**: configura un servei que permet crear un lloc web per al repositori o el projecte.
- **Zona de perill**: permet canviar el tipus de repositori (públic o privat), transferir la propietat o eliminar el repositori.

Respecte a la gestió d'usuaris, GitHub permet afegir **col·laboradors** a un repositori fent clic a l'opció *Collaborators* de la barra esquerra de la secció de configuració. En aquesta secció es mostra la llista de col·laboradors actuals i un botó per enviar una invitació a altres usuaris mitjançant el seu nom d'usuari o adreça de correu electrònic. Aquests col·laboradors podran visualitzar el repositori encara que sigui privat i podran pujar canvis al repositori remot. Cal destacar que l'administrador del repositori no té cap control sobre què pugen els col·laboradors, així que cal tenir-ho en compte a l'hora d'afegir col·laboradors a un repositori.

Gestió d'usuaris a GitHub Enterprise

GitHub Enterprise és una versió per a empreses de GitHub que inclou entre altres característiques una gestió d'usuaris més completa. Entre les característiques addicionals hi ha l'autenticació mitjançant LDAP i altres sistemes segurs, la creació d'organitzacions, la creació d'equips i l'administració d'usuaris per assignar-los a diferents equips per limitar les accions que pot portar a terme cada usuari.

Un altre tipus d'usuari són els **contribuïdors**. Aquests són usuaris de GitHub que no tenen cap relació amb el repositori, però poden fer peticions de pujada fent servir l'opció *Pull requests* que es troba a tots els repositoris. Aquesta opció és molt utilitzada en els projectes de programari lliure on qualsevol usuari interessat pot fer una petició de pujada amb algun canvi per millorar el projecte (per exemple, solucionar algun error). Aquests usuaris són considerats contribuïdors del projecte si s'accepta alguna de les seves peticions de pujada.

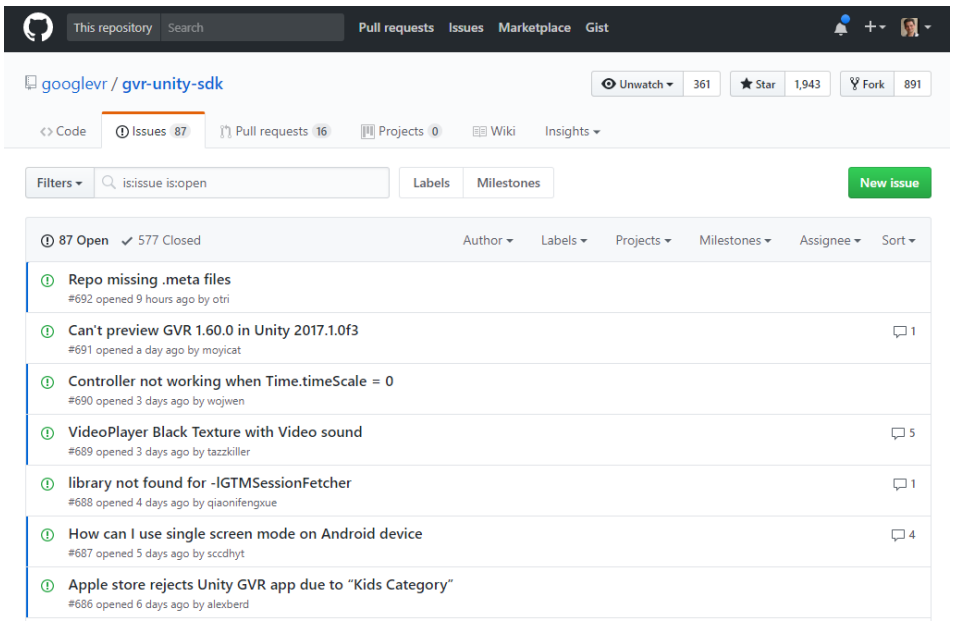
2.7.3 Gestió d'errors ('issues')

El sistema de gestió d'errors i petició de característiques de GitHub s'anomena *issues* i es pot accedir al sistema de qualsevol repositori fent clic al botó ***Issues*** del panell central.

Aquest sistema permet crear noves entrades a qualsevol usuari, en el cas dels repositoris públics, i als col·laboradors, en el cas dels repositoris privats, per enregistrar que hi ha algun error. Aquestes entrades permeten mantenir una conversa amb altres usuaris que tenen el mateix problema i els desenvolupadors afegint informació sobre el problema detectat.

Les entrades poden incloure etiquetes, poden ser assignades a diferents col·laboradors (el responsable de solucionar l'error) i poden filtrar-se: per autor, etiquetes, projectes, etc. (vegeu la figura 2.42).

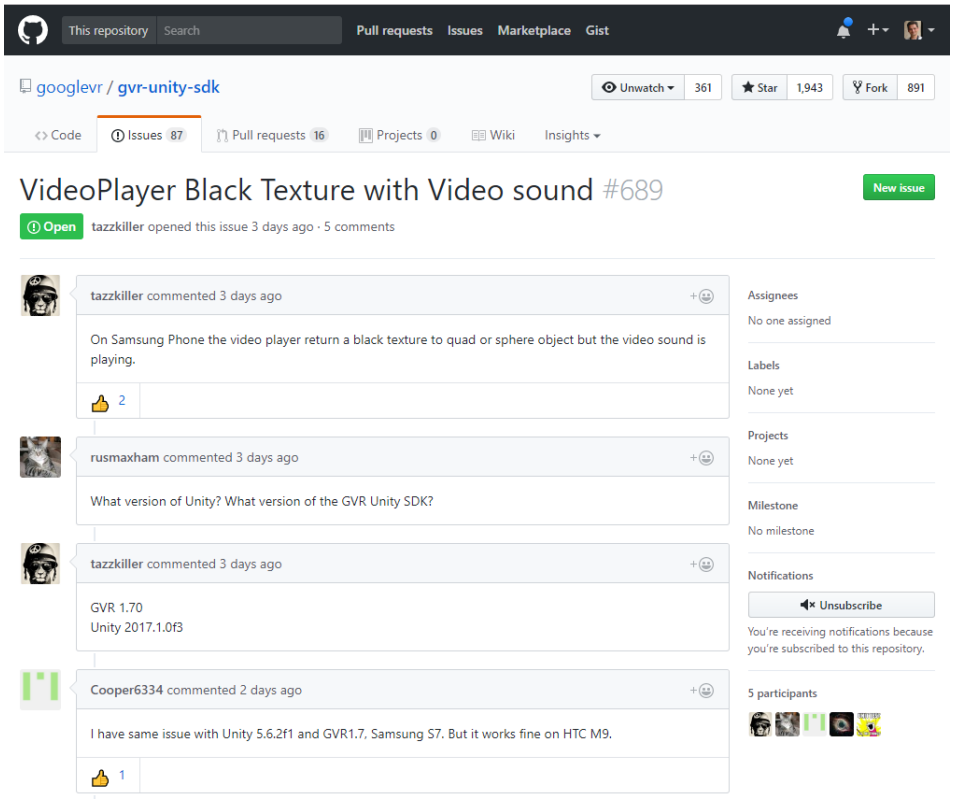
Figura 2.42. Llista d'errors i característiques d'un repositori de GitHub



Per redactar una entrada, podeu fer servir text enriquit, afegir imatges, mencionar usuaris (això fa que aparegui l'entrada a les seves notificacions) o enllaçar amb altres errors o peticions de pujada (per exemple, si un contribuïdor ha enviat la solució a l'error).

Per afegir un nou error o petició de característiques a un repositori, només cal que feu clic al botó *New issue* i redacteu l'entrada (vegeu un exemple d'entrades a la figura 2.43). Aquesta s'afegeix a la llista d'errors del projecte i és visualitzada per tots els usuaris.

FIGURA 2.43. Exemple d'entrada al sistema de gestió d'errors de GitHub



Un dels avantatges d'aquest sistema és que es troba al mateix repositori i no cal fer servir aplicacions externes per enregistrar un nou error o demanar més informació sobre un problema, fet que agilitza la gestió. A més a més, aquest sistema forma part de l'API que proporciona GitHub als desenvolupadors d'aplicacions per connectar amb els seus serveis.

2.8 Comentaris i documentació del programari

Un complement molt útil per als desenvolupadors és la documentació del codi font que s'està implementant. Existeixen moltes formes diferents de documentar amb molts nivells diferents d'aprofundiment. L'opció més senzilla és la primera que s'aprèn quan es comença a programar, que són els comentaris al llarg del codi font.

Documentar el codi d'un programa és dotar el programari de tota la informació que sigui necessària per explicar el que fa. Els desenvolupadors que duen a terme el programari (i la resta de l'equip de treball) han d'entendre què està fent el codi i el perquè.

Els comentaris es troben intercalats amb les sentències de programació, de fet es consideren part del codi font. Són petites frases que expliquen petites parts de les sentències de programació implementades.

Els comentaris no són tinguts en compte per part dels compiladors a l'hora de convertir el codi font en codi objecte i, posteriorment, en codi executable. Això dóna llibertat al programador per poder escriure qualsevol cosa en aquell espai que ell haurà indicat que és un comentari, sense haver de complir cap sintaxi específica.

Existeixen diferents formes d'implementar comentaris, anirà en funció del llenguatge de programació que es faci servir. Tot seguit es mostra la forma d'implementar comentaris amb Java. Amb aquest llenguatge hi ha dues formes d'expressar comentaris, internament i externament:

- **Comentaris interns**: igual que en el llenguatge de programació C i altres derivats, la forma de mostrar comentaris intercalats amb el codi font és afegint els caràcters `//`. Això indicarà que a partir d'aquell punt tot el que s'escrigui fins al final de la línia estarà considerat com a comentari i el compilador no ho tindrà en compte. Una altra forma de mostrar comentaris és afegint els caràcters `/* ... */`, on el comentari es trobaria ubicat en el lloc dels punts suspensius i, a diferència de l'anterior, es poden escriure comentaris de més d'una línia. A continuació, es mostra un exemple de com es pot documentar un petit tros de codi que contindrà els tipus de comentaris treballats. El codi està implementat en Javascript, ja que en un entorn web les validacions se solen fer en la màquina client.

```
1  /* Funció que efectua una validació del DNI, retornant cert si el DNI
       especificat té 8 caràcters i fals en cas contrari */
2  function validarDNI (DNI){
3      if (DNI.value != ""){                    //Valida que la variable DNI
           tingui algun valor.
4          if (DNI.value.length < 8 ) {         //Valida la longitud del DNI.
5              alert ("El DNI no és correcte");  //Mostra per pantalla un
                   missatge d'error.
6              DNI.focus();                     //Posiciona el cursor a la
                   variable DNI.
7              return false;                    //La funció retorna un false
                   indicant que el DNI no és vàlid.
8          }
9      }
10     return true;                             //La funció retorna un true
           indicant que el DNI és vàlid.
11 }
```

API

De l'anglès Application Programming Interface (Interfície de programació d'aplicacions). En la programació orientada a objectes, les API ofereixen funcions i procediments per ser utilitzats per altres aplicacions.

- **Comentaris o documentació a partir de la utilitat JavaDoc**: aquest tipus d'utilitat de creació de documentació a partir de comentaris ha estat desenvolupada per Oracle. JavaDoc permet la creació de documentació

d'API en format HTML. Actualment, és l'estàndard per crear comentaris de classes desenvolupades en Java. La sintaxi utilitzada per a aquest tipus de comentaris és començar per `/**` i finalitzar amb `*/`, on s'incorporarà el caràcter `*` per a cada línia, tal com es mostra tot seguit:

```
/** Comentari de JavaDoc
* De forma automàtica es generarà una pàgina HTML amb les comentaris
    especificats en el codi.
* @etiqueta1 text específic de l'etiqueta1
* @etiqueta2 text específic de l'etiqueta2
*/
```

La diferència entre aquest tipus de comentaris i els comentaris interns és que els comentaris JavaDoc sí que tenen una estructura específica que cal seguir per ser escrits. En canvi, els comentaris interns donen completa llibertat per ser implementats. Una altra diferència significativa és l'objectiu dels comentaris. Els comentaris interns es fan sevir arreu del codi font, mentre que els comentaris JavaDoc estan pensats per ser utilitzats al principi de cada classe i de cada mètode. Finalment, els comentaris JavaDoc generen de forma automàtica la documentació tècnica del programari, en format HTML.

2.8.1 Estructura dels comentaris tipus JavaDoc

Els comentaris tipus JavaDoc segueixen uns estàndards en la seva creació:

- El primer que cal indicar és la descripció principal del comentari. Aquesta descripció és el que es pot trobar des de la indicació de començament del comentari amb els caràcters `/**` fins que s'arriba a la secció on es troben les etiquetes, que queden indicades amb el caràcter `@`. La descripció general és un espai on es podrà escriure el que es voldrà, amb la llargària que es cregui convenient; és el més proper als comentaris estàndards.

- A continuació de la descripció principal, es troba una zona d'etiquetes (*tags*) que comencen pel caràcter `@`. Les etiquetes són *case-sensitive*, que vol dir que el seu significat canviarà si el seu nom s'escriu en minúscules o en majúscules. Una vegada ha començat la secció d'etiquetes, no es podrà continuar amb la descripció general del comentari JavaDoc. Les etiquetes s'han de trobar sempre al principi de les línies, amb el seu identificador, el caràcter `@`. Com a molt es podrà deixar un espai i un asterisc abans de l'etiqueta. Si no se segueix aquesta nomenclatura, l'etiqueta serà tractada com un text descriptiu.

Algunes de les etiquetes són:

- `@author nom`. Especifica l'autor del codi; en el cas que hagi estat escrit per diversos autors, es podrà replicar l'etiqueta tantes vegades com autors

hi hagi o escriure-ho amb una única etiqueta amb els diferents noms dels autors separats amb una coma.

- `@version versió`. Especifica la versió del codi.
- `@param paràmetre descripció`. Per a cada un dels paràmetres d'entrada, s'afegeix el paràmetre i la seva descripció.
- `@return descripció`. Indica el tipus de valor que retornarà la funció.
- `@exception nom descripció`. Afegeix una entrada "Throws", que conté el nom de l'excepció que pot ser llançada pel mètode.
- `@see referència`. Associa amb un altre mètode o classe.
- `@deprecated comentari`. Informa, en forma d'advertència, que una determinada funcionalitat no s'hauria d'utilitzar perquè ha quedat obsoleta.
- Els comentaris JavaDoc poden afegir codi HTML en la seva descripció, cosa que permetrà poder donar format als comentaris escrits. Si, per exemple, es volgués subratllar una part de la descripció principal, només caldria utilitzar l'etiqueta <u> i </u> abans i després del fragment de comentari a subratllar. Això permetrà donar un èmfasi especial a determinats comentaris. Per exemple:

```
/**
*Exemple de descripció general d'un comentari <u>JavaDoc.</u>
@ Author Programador primer del projecte
*/
```

- Amb Java es pot declarar més d'un atribut en una mateixa sentència. Si es vol comentar aquesta sentència per indicar a què correspon cada atribut, es necessitarà més d'una línia de comentaris. Per aquesta raó, si es vol tenir comentada la declaració d'atributs, un a un, serà necessari utilitzar una línia de codi per a cada declaració.

No és el mateix els comentaris del codi font que la documentació del programari. Els comentaris són una forma de documentar, de donar informació referent al codi font, però estan integrats en el mateix codi. El que es coneix com la documentació del programari és la creació de documents externs al codi (arxius), com poden ser manuals tècnics o manuals d'usuari. Aquests documents externs hauran de ser molt més explicatius i, a la vegada, més fàcils de comprendre que els comentaris del codi.

Un usuari tindrà accés als manuals funcionals o d'usuari, però mai accedirà al codi font, així que no podrà veure els comentaris del codi intercalats en el programari. Un programador que hagi de fer el manteniment o hagi d'ampliar l'aplicació informàtica, a banda de tenir accés a la documentació tècnica i funcional, sí que podrà accedir a aquests comentaris del codi font.

2.8.2 Exemple de comentaris tipus JavaDoc

En el següent exemple s'efectua una proposta dels comentaris tipus JavaDoc per a la classe factorial, classe que calcula el factorial d'un nombre.

```
/**
 * Classe que calcula el factorial d'un nombre.
 * @author IOC
 * @version 2012
 */
public class Factorial {

    /**
     * Calcula el factorial de n.
     * n! = n * (n-1) * (n-2) * (n-3) * ... * 1
     * @param n és el número al que es calcularà el factorial.
     * @return n! és el resultat del factorial de n
     */
    public static double factorial (double n) {

       if (n==0)
            return 1;
       else
       {
            double resultat = n * factorial(n-1);
            return resultat;
       }
    }
}
```

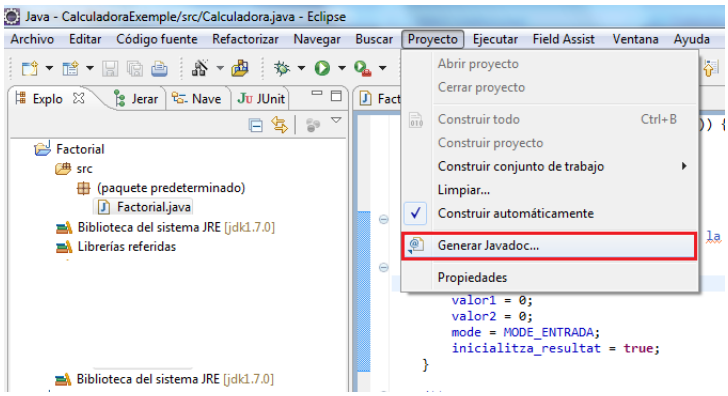
2.8.3 Exemple d'utilització de JavaDoc amb Eclipse

Moltes de les eines integrades de desenvolupament de programari que permeten programar en Java ofereixen funcionalitats per poder crear documentació a partir de JavaDoc. Eclipse no n'és una excepció i també ofereix aquesta possibilitat.

JDK, Java Development Kit, és un programari que aporta eines de desenvolupament per a la creació de programari en Java.

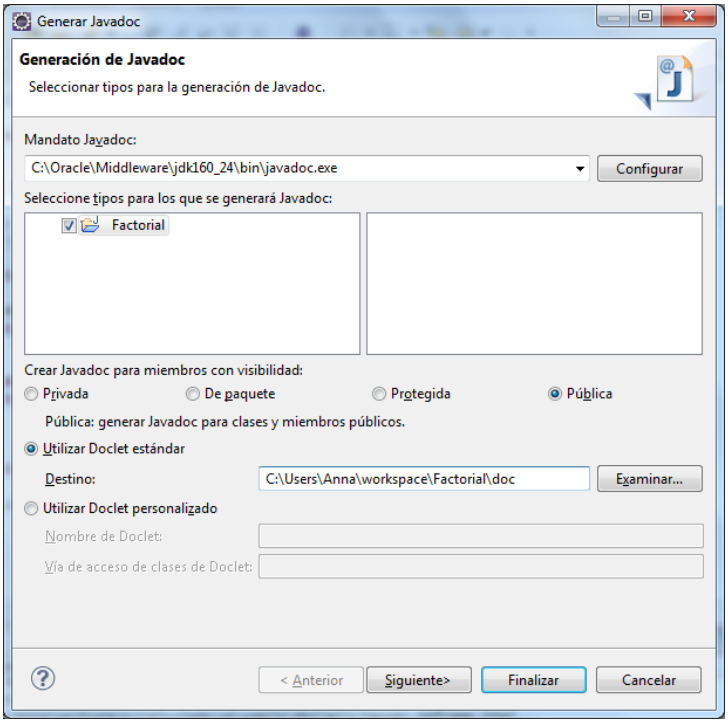
Una vegada desenvolupat un projecte en Java i afegits els comentaris amb les estructures i estàndards JavaDoc, es podrà generar la documentació de forma automàtica utilitzant la funcionalitat *Generar Javadoc* del menú *Projecte*, com es pot veure a la figura 2.44. Aquesta utilitat és part del Java Development Kit.

FIGURA 2.44. Menú 'Projecte' - 'Generar JavaDoc'



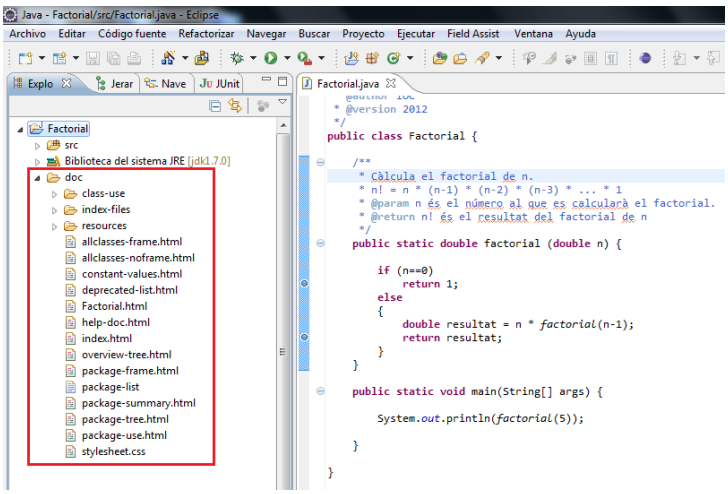
A continuació, obtindrà un diàleg que demanarà la ubicació de l'arxiu Javadoc.exe i una sèrie d'opcions de la creació de la documentació, com la carpeta destinació dels arxius que es crearan o el tipus de classes de les quals s'haurà de crear la documentació. Aquest diàleg es pot observar a la figura 2.45.

Figura 2.45. Diàleg per a la creació de la documentació a partir de JavaDoc



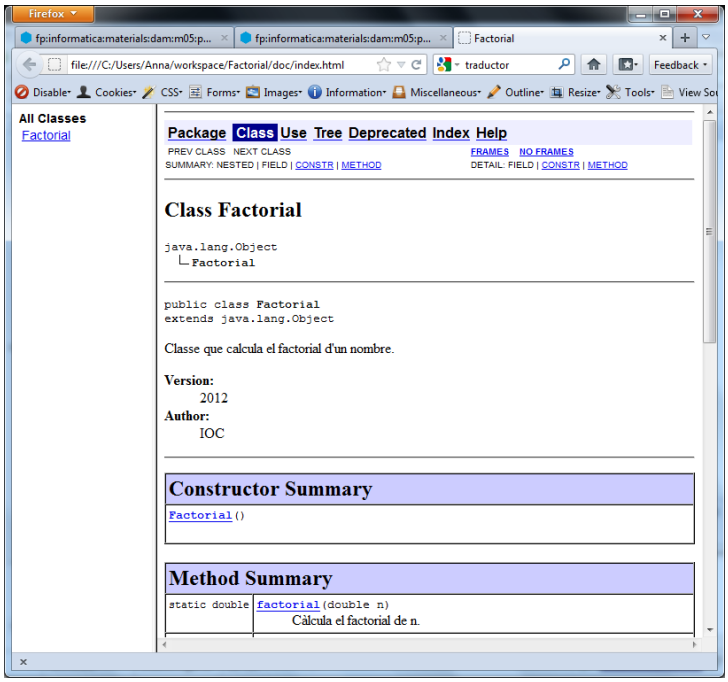
En fer clic sobre el botó *Finalitzar*, automàticament s'ha generat la documentació en format HTML, com es pot observar a la figura 2.46.

Figura 2.46. Documentació generada pel JavaDoc



Si obrim en el navegador l'arxiu index.html, es pot veure el document generat. Aquesta documentació generada es pot observar a la figura 2.47 i figura 2.48.

FIGURA 2.47. Documentació generada pel JavaDoc



A la figura 2.47 es pot observar la descripció general de tota la classe, juntament amb la seva estructura. També es pot observar el resum del seu constructor i els seus mètodes.

JavaDOC es pot trobar a les versions d'Eclipse per als diferents sistemes operatius (Windows, Linux i MacOS).

A la figura 2.48 es pot observar el detall del constructor i el detall dels mètodes, amb la informació que s'ha anat introduint com a comentaris al llarg del codi font.

FIGURA 2.48. Ampliació de la documentació generada

Introducció al disseny orientat a objectes

Marcel García Vacas

Entorns de desenvolupament

Índex

Introducció

La fase de disseny, dins un projecte de desenvolupament de programari informàtic, es durà a terme després d'efectuar l'anàlisi de requeriments, i serà la fase que establirà les bases sobre com s'han de fer les coses a l'hora de desenvolupar l'aplicació. Un projecte ben planificat oferirà una fase de disseny que haurà de ser més costosa (en temps i recursos) que la pròpia fase de desenvolupament del programari. Si es fa ben fet, amb una codificació de programari automàtica o semiautomàtica, un bon disseny estalviarà molt temps d'haver de picar codi.

Aquesta fase de disseny s'haurà de dur a terme en concordança amb la metodologia escollida per al desenvolupament del projecte. Si s'ha apostat, per exemple, per una metodologia àgil, caldrà fer el disseny en funció d'aquest tipus de programació. Durant molts anys s'ha desenvolupat un disseny estructurat que dirigia cap a un tipus de programació estructurada i modular. De fet, en alguns projectes encara es fa servir aquest tipus de disseny. Igual que la programació estructurada va evolucionar cap a la programació orientada a objectes, l'evolució natural ha estat anar cap a una codificació de programari orientada a objectes, seguint, també, una anàlisi i un disseny orientats a objectes. UML permet elaborar un model del sistema que es vol automatitzar.

L'actual unitat formativa s'ha dividit en dos apartats. L'apartat "Diagrames estàtics" està enfocat a la introducció de tot allò relacionat amb el disseny de desenvolupament de programari, entrant en detall en el disseny orientat a objectes i, especialment, en els diagrames estàtics, que són els que identifiquen els elements de la modelització del sistema.

En l'apartat "Diagrames dinàmics", per la seva banda, es treballen els diagrames que indiquen el comportament del sistema i les seves interaccions. Aquests diagrames complementen els diagrames estàtics per oferir, conjuntament, un disseny complert de la solució que s'ofereix a un projecte informàtic.

Resultats d'aprenentatge

En finalitzar aquesta unitat l'alumne/a:

1. Genera diagrames de classes valorant la seva importància en el desenvolupament d'aplicacions i emprant les eines disponibles en l'entorn.

- Identifica els conceptes bàsics de la programació orientada a objectes.
- Instal·la el mòdul de l'entorn de desenvolupament integrat que permet la utilització de diagrames de classes.
- Identifica les eines per a l'elaboració de diagrames de classes.
- Interpreta el significat de diagrames de classes.
- Traça diagrames de classes a partir de les especificacions de les mateixes.
- Genera codi a partir d'un diagrama de classes.
- Genera un diagrama de classes mitjançant enginyeria inversa.

2. Genera diagrames de comportament valorant la seva importància en el desenvolupament d'aplicacions i emprant les eines disponibles en l'entorn.

- Identifica els diferents tipus de diagrames de comportament.
- Reconeix el significat dels diagrames de casos d'ús.
- Interpreta diagrames d'interacció.
- Elabora diagrames d'interacció senzills.
- Interpreta el significat de diagrames d'activitats.
- Elabora diagrames d'activitats senzills.
- Interpreta diagrames d'estats.
- Planteja diagrames d'estats senzills.

1. Diagrames estàtics

El llenguatge unificat de modelització (UML) es basa en diagrames que estableixen les especificacions i documentació de qualsevol sistema. Aquests diagrames es fan servir en el desenvolupament de projectes informàtics per analitzar i dissenyar les necessitats dels usuaris finals de les aplicacions, amb la qual cosa es documenten els seus requeriments. Però també es podran utilitzar en molts més àmbits, com ara la modelització del funcionament d'una empresa, d'un departament o d'un sistema de qualsevol altre entorn.

Els diagrames que es poden fer servir per modelitzar sistemes es poden agrupar o classificar de moltes formes. Una d'aquestes classificacions diferencia entre els diagrames que corresponen a l'UML estàtic i els que corresponen a l'UML dinàmic. Els diagrames inclosos en els considerats de UML estàtic, també anomenats diagrames d'estructura, fan referència als elements que s'hauran de trobar en el sistema de modelització. Aquests diagrames fan una descripció del sistema sense tenir en compte les interaccions amb altres elements o el comportament que tenen els elements del sistema, només identifiquen aquests models i estableixen les seves característiques.

Dins aquest tipus de diagrames, el més important és el **diagrama de classes**, que especificarà totes les classes estimades al disseny orientat a objectes i que seran instanciades al codi de programació que es desenvoluparà a continuació.

En les pàgines corresponents a aquest apartat es treballaran els diagrames corresponents a l'UML estàtic, fent especial referència al diagrama de classes. També es mostra un exemple de com poder treballar amb un IDE (Eclipse) amb els diagrames UML.

1.1 Anàlisi i disseny orientat a objectes

La creació d'una aplicació informàtica es pot considerar una obra d'enginyeria. De fet, fa molts anys que es coneix amb el nom **enginyeria del programari** (o **enginyeria del *software***), considerat com una de les àrees més importants de la informàtica. L'enginyeria del programari engloba tota una sèrie de metodologies, tècniques i eines que faciliten la feina que comporten totes i cada una de les fases d'un projecte informàtic.

L'enginyeria del programari ha evolucionat de forma contínua (i ho continua fent a data d'avui), adaptant-se a les noves tecnologies i a les noves formes de desenvolupar programari. A la dècada dels 70, amb l'inici de la programació d'aplicacions com a feina generalitzada en les organitzacions, i l'aparició de la figura del programador, només hi havia un objectiu: que les aplicacions fessin el

que havien de fer optimitzant la velocitat i l'ús de memòria. En aquells primers programes ningú no es preocupava pel futur manteniment de les aplicacions per part d'altres programadors, ni tampoc que el codi fos fàcil d'entendre.

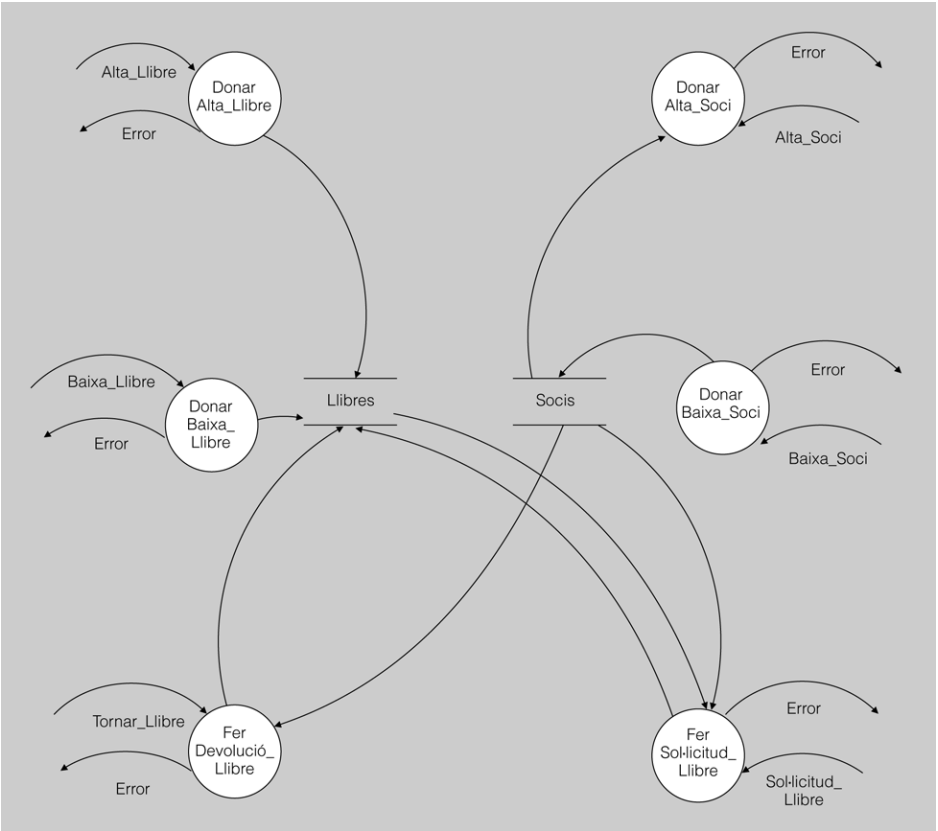
Amb els anys, les aplicacions demanen més funcionalitats i el desenvolupament d'aplicacions es fa més complex a mesura que millora el maquinari i el programari que utilitzen els programadors. La programació passa a ser programació estructurada, implementada en mòduls i més senzilla d'entendre i de mantenir.

A mitjan-finals dels anys 70 comencen a aparèixer les primeres metodologies. El fet s'elaborar el codi de programació a partir de les improvisacions o de les intuïcions del programador, sotmet el projecte a uns riscos, com ara la poca eficàcia o els errors en les funcionalitats que han de solucionar els requeriments de les especificacions.

Aquestes metodologies recullen l'experiència de molts projectes, oferint una sèrie de passos a efectuar per al correcte i exitós desenvolupament de les aplicacions informàtiques. Entre aquests passos es troben les fases en què es divideix un projecte, entre les quals hi ha la fase de disseny.

El disseny estructurat, o també anomenat disseny orientat al flux de dades, permet establir la transició del Diagrama de Flux de Dades (DFD) a una descripció de l'estructura del programa, que es representa mitjançant un diagrama d'estructures. Les eines que faciliten i automatitzen aquest tipus de disseny es coneixen com a eines CASE. A la figura 1.1 es pot observar un exemple de Diagrama de Flux de Dades.

FIGURA 1.1. Exemple DFD



L'evolució en l'enginyeria del programari sorgeix a partir dels anys 80, quan comença a aparèixer la programació orientada a objectes. La forma de programar ja no es basarà en les estructures i en la programació modular, sinó que la base seran les classes i els objectes. Igual que hi ha una evolució en la programació, aquesta evolució també es veu reflectida en les metodologies de gestió de projectes. Al cap d'uns anys d'agafar força la programació orientada a objectes i de fer-se un lloc entre els desenvolupadors d'aplicacions, van aparèixer:

- Modelització de sistemes orientats a objectes.
- Disseny orientat a objectes.
- Generació automàtica de Codi.
- Metodologies àgils.

En l'actualitat existeixen diferents tipus de tecnologies que s'utilitzen en el desenvolupament de programari, i les més utilitzades són:

- Metodologies estructurades.
- Metodologies orientades a objectes.
- Metodologies àgils.

Deixant la tercera de banda, en les altres dues (estructurades i orientades a objectes) es pot considerar que el procés de desenvolupament de programari té les mateixes fases. Segons diversos autors, pot haver-hi grans divergències entre les propostes de fases d'un projecte, però, en general, les proposades són aquestes:

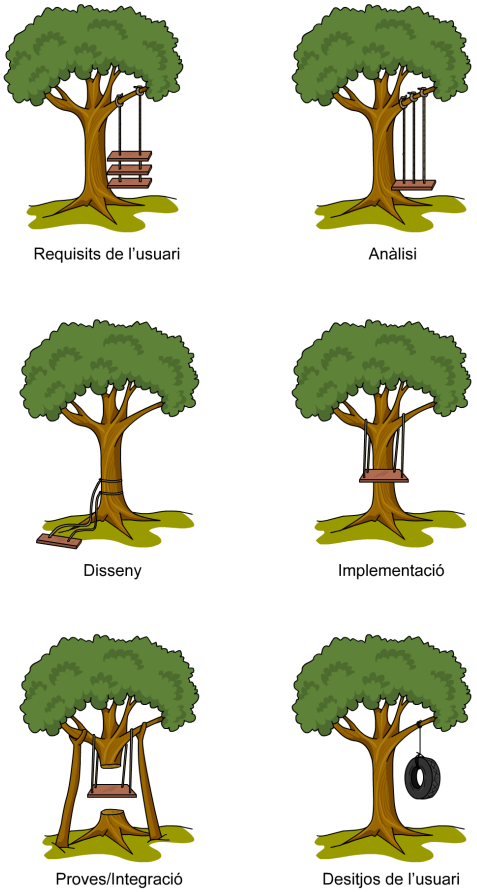
- **Estudi previ**, on es delimita l'àmbit del projecte de desenvolupament d'un programari i se'n determina la viabilitat, els riscos, els costos i el calendari.
- **Anàlisi de requeriments**, on es descriuen de manera adient els requisits del programari, que són les necessitats d'informació que el programari ha de satisfer. L'anàlisi ha de deixar molt clar el que es vol fer a partir de l'estudi d'un problema determinat. Els requeriments que cal recollir hauran ser funcionals i no funcionals. Caldrà que sigui independent de la tecnologia i del tipus de programació escollits per al desenvolupament del projecte.
- **Disseny**, on es projecta el programari amb vista a implementar-lo en un entorn tecnològic concret, definit per aspectes com ara el llenguatge de programació, l'entorn de desenvolupament i el gestor de bases de dades, de tots els quals en pot intervenir més d'un dins un sol projecte. A partir del que caldrà fer, definit a la fase anterior al disseny, es determina el com es farà. Caldrà tenir en compte les possibles conseqüències de les decisions presses en aquesta fase. El detall haurà de ser suficientment baix com perquè a partir d'aquestes indicacions es pugui desenvolupar el projecte.

- **Desenvolupament i proves**, on es porta a terme el desenvolupament del codi font que satisfaci les funcionalitats establertes a les fases anteriors. Aquesta fase no només es limita a la codificació manual, sinó que pot incloure la programació gràfica, la generació automàtica de codi i altres tècniques. En aquesta fase també s'inclouran les proves del programari. Algunes parts de la programació poden començar abans que el disseny estigui enllestit del tot, i també es poden provar parts del programari mentre encara se n'estan implementant d'altres.

- **Finalització i transferència**, on es prepara el programari per tal que estigui en explotació. Durant aquesta fase, és objecte de manteniment per eliminar errors no corregits durant la prova, introduir-hi millores i adaptar-lo a canvis tecnològics i organitzatius que es produeixin en l'entorn de treball dels usuaris.

Un model adequat evita trobar-se amb situacions com la que es mostra en la figura 1.2, on es poden observar, a mode d'exemple, les diferències d'enteniment entre les diferents parts i fases implicades en la gestió d'un projecte informàtic.

Figura 1.2. Errors a evitar en el desenvolupament d'un programari



1.2 Llenguatge unificat de modelització

UML és l'acrònim, en anglès, de *Unified Modeling Language*, és a dir, Llenguatge unificat de modelització. UML són un conjunt de notacions gràfiques que serveixen per especificar, dissenyar, elaborar i documentar models de sistemes i, en particular, d'aplicacions informàtiques. A partir d'aquestes notacions gràfiques o diagrames, l'analista i el dissenyador podran recrear les característiques que caldrà que tingui l'aplicació informàtica que els desenvolupadors hauran de crear posteriorment.

En l'actualitat és el llenguatge de modelització de sistemes més conegut i utilitzat. Gaudeix d'una acceptació gairebé universal i, entre altres beneficis, ha tingut l'efecte d'impulsar el desenvolupament d'eines de modelització gràfica del programari orientat a l'objecte. A més està reconegut i és recomanat per l'OMG.

OMG

Prové de l'anglès, Object Management Group, que és un consorci fundat l'any 1989 amb l'objectiu de fomentar l'ús de la tecnologia orientada a l'objecte mitjançant l'establiment d'estàndards.

Els avantatges de la notació UML són els següents:

- Està recolzada per la OMG (Object Management Group) com la notació estàndard per al desenvolupament de projectes informàtics.
- Es basa en una notació gràfica concreta i fàcil d'interpretar, essent completada amb explicacions escrites.
- A l'analista i/o al dissenyador els permet fer ús dels diagrames que considerin oportuns i amb el grau de detall que considerin en funció de les característiques del sistema.
- Permet tenir una visió global del sistema a implementar.
- Promou la reutilització.

Entre els desavantatges destaquen:

En la secció *Annexos* del web del mòdul hi podeu trobar més informació referent a la història i característiques de l'UML.

- UML no és una metodologia, és una notació.
- UML no és un llenguatge de programació.
- Pot resultar complex obtenir un coneixement complet de les possibilitats del llenguatge.

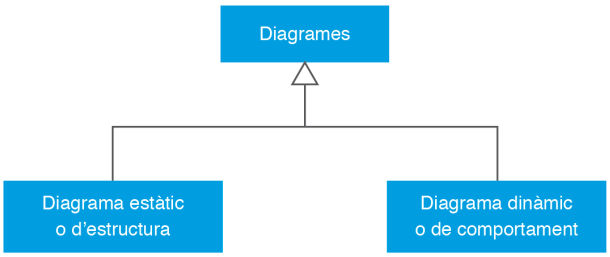
1.2.1 Diagrames

En el llenguatge de modelització UML, en la seva versió 2.4.1, existeixen un total de 14 tipus diferents de diagrames.

Com es pot observar a la figura 1.3, una possible classificació dels diagrames es fa en funció de la visió del model del sistema que ofereixen. Les dues visions diferents de model de sistema que poden representar els diagrames UML són:

- **Visió estàtica (o estructural)**: per oferir aquest tipus de visió s'utilitzen objectes, atributs, operacions i relacions. La visió estàtica d'un sistema dóna més valor als elements que es trobaran en el model del sistema des d'un punt de vista de l'estructura del sistema. Descriuen aspectes del sistema que són estructurals i, per tant, permanents (allò que el sistema té).
- **Visió dinàmica (o de comportament)**: per oferir aquest tipus de visió es dóna més valor al comportament dinàmic del sistema, és a dir, a allò que ha de passar en el sistema. Amb els diagrames de comportament es mostra com es modela la col·laboració entre els elements del sistema i els seus canvis d'estat. Representen allò que pot fer el sistema modelitzat.

FIGURA 1.3. Diagrames UML

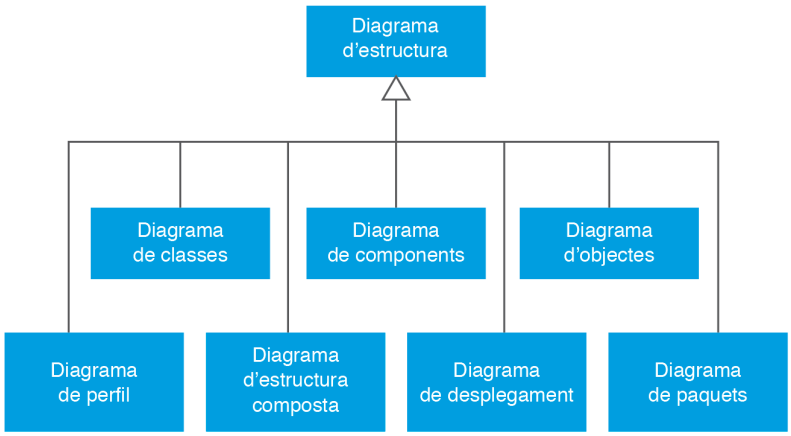


Dins els diagrames estàtics o diagrames d'estructura es poden trobar 7 tipus de diagrames, que són:

- **Diagrama de paquets**, que representa essencialment les relacions de diferents menes entre els continguts de diferents paquets d'un model.
- **Diagrama de classes**, que probablement molts consideren el diagrama principal, atès que descriu classificadors de tota mena i diferents tipus de relacions entre ells abans de fer-los servir en altres diagrames.
- **Diagrama d'objectes**, que representa instàncies de classificadors definits en un diagrama de classes previ i relacions entre elles.
- **Diagrama d'estructures compostes**, que descriu casos en què, o bé les instàncies d'un classificador tenen com a parts instàncies d'altres, o bé en el comportament executant d'un classificador participen instàncies d'altres.
- **Diagrama de components**, que és un diagrama de classes i alhora d'estructures compostes simplificat i més adient per a determinades tecnologies de programació.
- **Diagrama de desplegament**, que descriu la configuració en temps d'execució d'un programari especificat, normalment, per un diagrama de components.
- **Diagrama de perfil**, permet adaptar o personalitzar el model amb construccions que són específiques d'un domini en particular, d'una determinada plataforma, o d'un mètode de desenvolupament de programari...

A la figura 1.4 es pot observar un resum dels diagrames d'estructura.

Figura 1.4. Diagrames d'estructura o estàtics



Dins els diagrames dinàmics o diagrames de comportament es poden trobar 7 tipus de diagrames, que són:

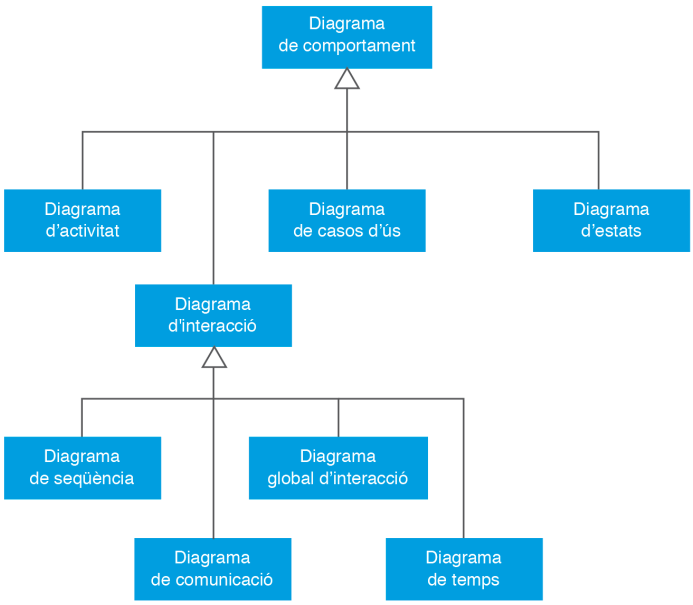
- **Diagrama de casos d'ús**, que considera els comportaments d'un sistema principalment des del punt de vista de les interaccions que té amb el món exterior.
- **Diagrama d'estats**, en el qual es descriuen les diferents situacions -estats- des del punt de vista dels seus comportaments, de les instàncies d'un classificador, alhora que les causes, condicions i conseqüències dels canvis d'una situació a una altra.
- **Diagrama d'activitats**, en què es descriu un comportament complex en forma de seqüències condicionals d'activitats components.

Diagrama d'interacció, que descriu comportaments emergents i té les variants següents:

- **Diagrama de seqüències**, que fa èmfasi en la seqüència temporal de les participacions de les diferents instàncies.
- **Diagrama de comunicacions**, que es basa directament en una estructura composta.
- **Diagrama de visió general de la interacció**, que dóna una visió resumida del comportament emergent.
- **Diagrama temporal**, que es basa en un diagrama d'estats previ o més d'un i alhora posa èmfasi en els canvis d'estat al llarg del temps.

A la figura 1.5 es pot observar un resum dels diagrames dinàmics, també anomenats diagrames de comportament.

FIGURA 1.5. UML: Diagrama de comportament o dinàmic



1.2.2 Diagrama de classes

Un dels diagrames referents de UML, classificat dins els diagrames de tipus estàtic, és el diagrama de classes. És un dels diagrames més utilitzats a les metodologies d'anàlisi i de disseny que es basen en UML.

Un diagrama de classes representa les classes que seran utilitzades dins el sistema i les relacions que existeixen entre elles.

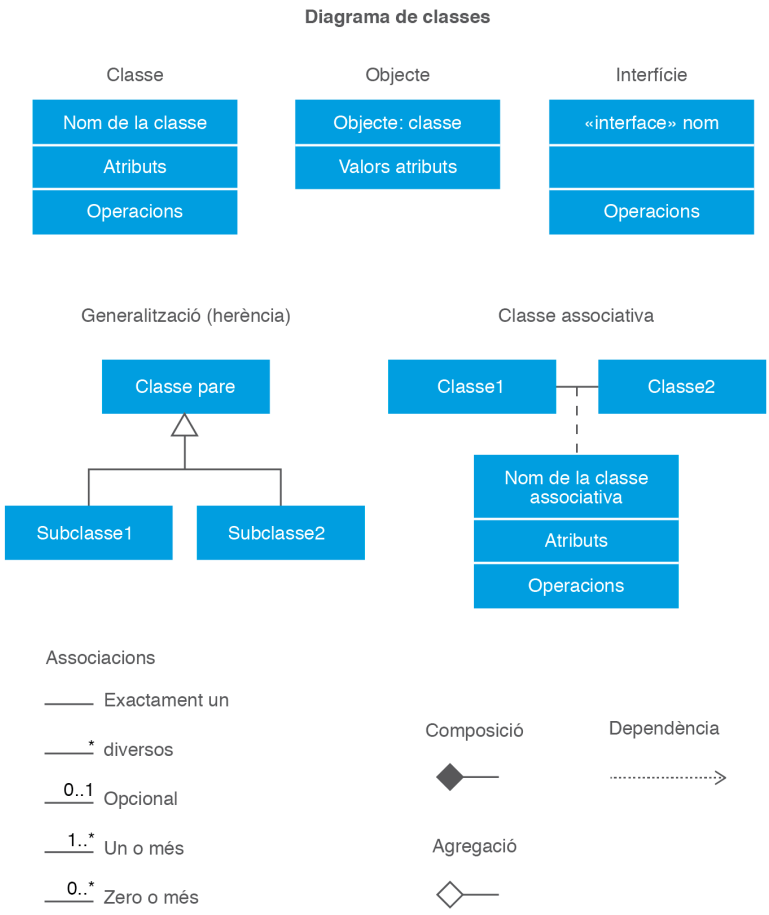
Aquest tipus de diagrames són utilitzats durant les fases d'anàlisi i de disseny dels projectes de desenvolupament de programari. És en aquest moment en què es comença a crear el model conceptual de les dades que farà servir el sistema. Per això s'identifiquen els components (amb els seus atributs i funcionalitats) que prendran part en els processos i es defineixen les relacions que hi haurà entre ells.

Un diagrama de classes porta vinculats alguns conceptes que ajudaran a entendre'n la creació i el funcionament en la seva totalitat. Aquests conceptes són:

- Classe, atribut i mètode (operacions).
- Visibilitat.
- Objecte. Instanciació.
- Relacions. Herència, composició i agregació.
- Classe associativa.
- Interfícies.

A la figura 1.6 es mostren els elements que poden pertànyer a un diagrama de classes.

Figura 1.6. Elements del diagrama de classes



Classes. Atributs i operacions

Una **classe** descriu un conjunt d'objectes que comparteixen els mateixos **atributs**, que representen característiques estables de les classes, i les **operacions**, que representen les accions de les classes.

Els atributs (també anomenats *propietats* o *característiques*) són les dades detallades que contenen els objectes. Aquests valors corresponen a l'objecte que instancia la classe i fa que tots els objectes siguin diferents entre si.

Tot atribut té assignat un tipus i pot tenir una llista formada per un valor o més d'aquest tipus, d'acord amb la multiplicitat corresponent, que han de pertànyer a aquest tipus i poden variar al llarg del temps.

Per exemple, un possible atribut de la classe persona podria ser el seu nom, atribut de tipus *string*.

Les **operacions** (també anomenades mètodes o funcionalitats) implementen les accions que es podran dur a terme sobre els atributs. Ofereixen la possibilitat d'aplicar canvis sobre els atributs, però també moltes altres accions relacionades amb l'objecte, com obrir-lo o tancar-lo, carregar-lo...

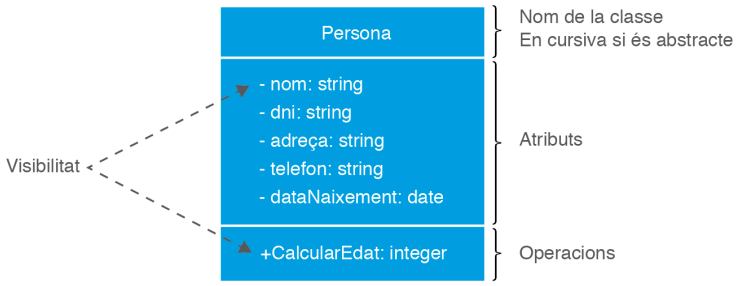
Cada **operació** té una signatura que en descriu els eventuals paràmetres i el seu valor de retorn. Els paràmetres tenen nom, tipus, multiplicitat i direcció (que indica si el paràmetre és d'entrada, de sortida, o d'entrada i sortida alhora). El valor de retorn, si n'hi ha, només té tipus i multiplicitat.

Les operacions poden tornar excepcions durant la seva execució. Una **excepció** és una instància d'un classificador que és el seu tipus. El fet que una operació torni una excepció normalment denota que s'ha produït una anomalia en l'execució.

Per exemple, un possible mètode de la classe `Persona` podria ser `CalcularEdat`, que retorna un enter.

A la figura 1.7, es mostra un breu exemple d'una classe i com es representaria.

Figura 1.7. Exemple de representació d'una classe



Aquesta representació mostra un rectangle que està dividit en tres files:

- A la primera s'indica el nom de la classe.
- A la segona s'indiquen els atributs que donen les característiques a la classe. Caldrà també indicar la seva visibilitat.
- A la tercera s'indiquen els mètodes o les operacions amb la seva declaració i visibilitat.

El codi que es generaria a partir de la classe `Persona` de la figura 1.7 seria:

```
1  Class Persona{
2  {
3       // atributs
4       private String nom;
5       private String dni;
6       private String adreca;
7       private String telefon;
8       private date dataNaixement;
9       // constructor
10      public Persona(String nom, String dni, string adreca, string telefon, date
            dataNaixement{
```

```
11        this.nom = nom;
12        this.dni = dni;
13        this.adreca= adreca;
14        this.telefon = telefon;
15        this.dataNaixement = dataNaixement;
16      }
17      // mètodes o operacions
18      public integer CalcularEdat() {
19           Date dataActual = new Date();
20           SimpleDateFormat format = new SimpleDateFormat("dd/MM/yyyy");
21           String _dataActual = formato.format(dataActual);
22           String _dataNaixement = formato.format(dataNaixement);
23
24           String[] dataInici = _dataNaixement.split("/");
25           String[] dataFi = _dataActual.split("/");
26
27           int anys = Integer.parseInt(dataFi[2]) - Integer.parseInt(dataInici
                 [2]);
28           int mes = Integer.parseInt(dataFi[1]) - Integer.parseInt(dataInici
                 [1]);
29           if (mes < 0) {
30                anys = anys - 1;
31           } else if (mes == 0) {
32                int dia = Integer.parseInt(dataFi[0]) - Integer.parseInt(
                       dataInici[0]);
33                if (dia > 0) anys = anys - 1;
34           }
35           return anys;
36      }
37 }
```

La visibilitat

La **visibilitat** d'un atribut o d'una operació definirà l'àmbit des del qual podran ser utilitzats aquests elements. Aquesta característica està directament relacionada amb el concepte d'orientació a objectes anomenat *encapsulació*, mitjançant el qual es permet als objectes decidir quina de la seva informació serà més o menys pública per a la resta d'objectes.

Exemple de visibilitat

Una classe és un espai de noms per a les seves variables i mètodes; per exemple, un mètode de visibilitat # només pot ser invocat des de mètodes que formin part de la mateixa classe o d'alguna de les seves especialitzacions eventuals.

Les possibilitats per a la visibilitat, tant d'atributs com de mètodes, són:

- + en UML, o `public`, que vol dir que l'element és accessible per tots els altres elements del sistema.
- - en UML, o `private`, que significa que l'element només és accessible pels elements continguts dins el mateix objecte.
- # en UML, o `protected`, que vol dir que l'element només és visible per als elements del seu mateix objecte i per als elements que pertanyen a objectes que són especialitzacions.
- ~ en UML, o `package`, que només es pot aplicar quan l'objecte no és un paquet, i aleshores vol dir que l'element només és visible per als elements continguts directament o indirectament dins el paquet que conté directament l'objecte.

Per mostrar un exemple de la visibilitat, es mostra a continuació el codi de la classe `Habitació`, la classe `Persona` i la classe `Test`. Cadascuna de les classes té els seus atributs i els seus mètodes. Alguns d'aquests elements seran públics o privats, oferint les característiques que s'indiquen en finalitzar el codi.

```
1  public class Habitacio {
2      private String dataEntrada;
3      private String dataSortida;
4      private Persona client;
5
6      public Habitacio (String dataEntrada, String dataSortida, String nom) {
7         this.dataEntrada = dataEntrada;
8         this.dataSortida = dataSortida;
9         client = new Persona(nom);
10     }
11     public String getDataEntrada() {
12        return dataEntrada;
13     }
14     public String getDataSortida() {
15        return dataSortida;
16     }
17     public Persona getClient() {
18        return client;
19     }
20  }
21 class Persona{
22     private String nom;
23
24     public Persona(String nom) {
25        this.nom = nom;
26     }
27     public String getNom() {
28        return nom;
29     }
30     public void setNom(String nom) {
31        this.nom = nom;
32     }
33 }
34 public class Test {
35     public static void main(String[] args) {
36        Habitacio reserva = new Habitacio ("24/011/2012", "28/11/2012", "Joan
              Garcia");
37        Persona client = reserva.getClient();
38        String nom = reserva.getClient().getNom();
39     }
40 }
```

Caldria fixar-se en el mètode `reserva.getClient()`, que únicament té visibilitat als atributs i als mètodes públics de la classe `Persona`, és a dir, pot consultar el nom de la persona a través del mètode públic `getNom` però no pot accedir a l'atribut privat `nom`.

D'altra banda, els mètodes `getNom` i `setNom` de la classe `Persona` tenen visibilitat als atributs i als mètodes privats de la classe, amb la qual cosa, poden accedir a l'atribut privat `nom`.

Objectes. Instanciació

Un objecte és una instanciació d'una classe. El concepte *instanciació* indica l'acció de crear una instància d'una classe.

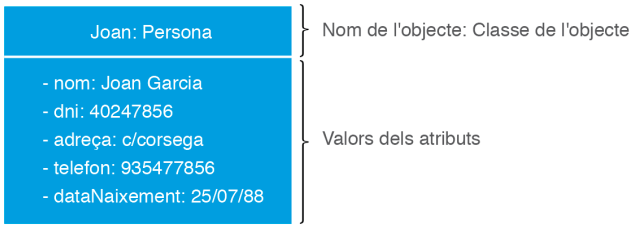
La creació d'una instància d'una classe es refereix a fer una crida al mètode constructor d'una classe en temps d'execució d'un programari.

> Un **objecte** es pot definir, llavors, com una unitat de memòria relacionada que, en temps d'execució, du a terme accions dintre un programari. És quelcom que es pot distingir i que té una existència pròpia, ja sigui de forma conceptual o de forma física.

Un objecte pot pertànyer a més d'una classe, però només d'una d'elles es poden crear objectes directament: la resta de classes representen només papers que pot exercir l'objecte.

Un possible objecte de la classe `Persona` podria ser el que es mostra a la figura 1.8.

Figura 1.8. Exemple d'un objecte de la classe "Persona"



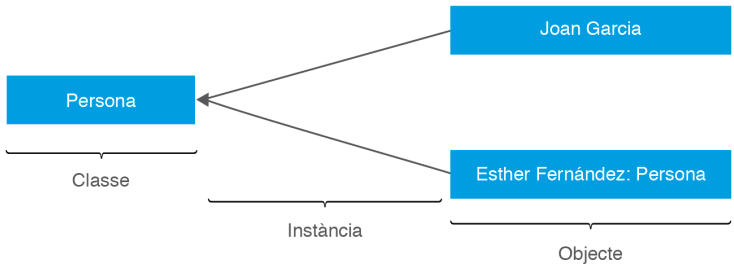
El codi per crear un objecte o instància es mostra tot seguit:

```
1 public static void main(String[] args) {
2       Persona Joan = new Persona("Joan Garcia","40782949","C/Casanoves 130",
            "93 2983924", 25/03/1977);
3 }
```

Un aspecte característic dels objectes és que tenen identitat, cosa que significa que dos objectes creats per separat sempre es consideren diferents, encara que tinguin els mateixos valors a les característiques estructurals.

En canvi, les instàncies de segons quins classificadors no tenen identitat i, alesho-res, dues instàncies amb els mateixos valors a les característiques estructurals es consideren com una mateixa instància. Aquest fet es representa a la figura 1.9.

Figura 1.9. Instanciació d'objectes de la classe "Persona"

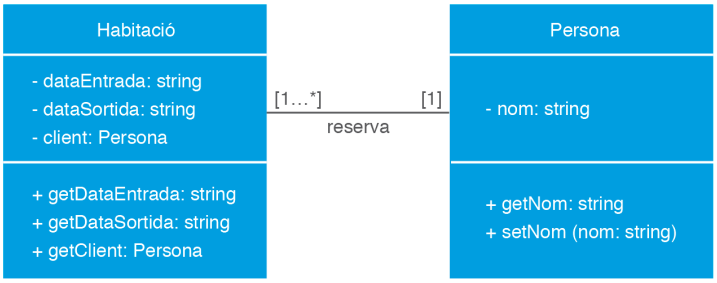


Relacions. Herència, composició, agregació

Les **relacions** són elements imprescindibles en un diagrama de classes. Per relació s'entén que un objecte `obj1` demani a un altre `obj2`, mitjançant un missatge, que executi una operació de les definides en la classe de l'`obj2`.

En l'exemple d'un client que reserva una habitació d'hotel es pot observar que intervenen dues classes, `Persona` i `Habitació`, que estan relacionades, on la classe `Habitació` consulta el nom del client corresponent a la classe `Persona`. Es veu representat a la figura 1.10.

FIGURA 1.10. Relació entre classes



Les relacions que existeixen entre les diferents classes s'anomenen de forma genèrica **associacions**.

Una **associació** és un classificador que defineix una relació entre diversos classificadors, que estableix connexions amb un cert significat entre les instàncies respectives.

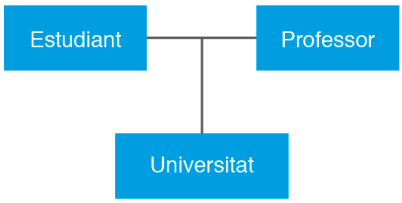
Classificador

Un classificador és un tipus els valors del qual tenen en comú característiques estructurals i de comportament, que són elements que es defineixen a nivell del classificador. Cadascun d'aquests valors és una instància del classificador.

Una associació té diversos **extrems d'associació**, a cadascun dels quals hi ha un classificador que té un cert paper en l'associació; un classificador pot ser en més d'un extrem de la mateixa associació amb papers diferents.

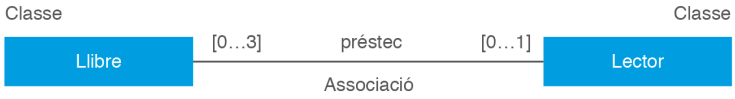
Una associació amb dos extrems es diu que és binària; seria el cas de l'exemple d'un client que reserva una habitació d'hotel. Una associació amb tres extrems es diu que és ternària. Es pot veure un exemple d'una associació ternària a la figura 1.11.

FIGURA 1.11. Associació ternària



La **multiplicitat**, representada per uns valors `<min....max>`, indica el nombre màxim d'enllaços possibles que es podran donar en una relació. Aquesta multiplicitat no podrà ser mai negativa, essent, a més a més, el valor màxim sempre més gran o igual al valor mínim. A la figura 1.12 es pot veure un exemple de la representació d'una associació entre dues classes amb la indicació de la seva multiplicitat.

FIGURA 1.12. Associació amb multiplicitat



Concretament, indica que, en l'associació `Préstec`, per a cada objecte de la classe `Lector` podrà haver-hi, com a mínim 0 objectes de la classe `Llibre` i, com a màxim, 3 objectes. En canvi per a cada objecte de la classe `Llibre` podrà haver-hi, com a mínim 0 i com a màxim 1 objecte de la classe `Lector`.

Es poden trobar diferents tipus de relacions entre classes. Aquestes es poden classificar de moltes formes. A continuació, es mostra una classificació de les relacions:

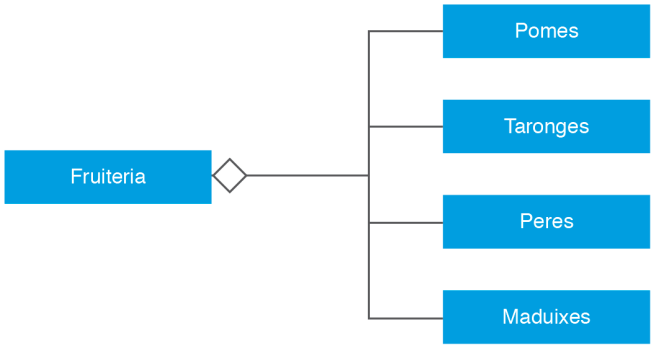
- Relació d'associació
- Relació d'associació d'agregació
- Relació d'associació de composició
- Relacions de dependència
- Relació de generalització

1) La **relació d'associació** es representa mitjançant una línia contínua sense fletxes ni cap altre símbol als extrems. És el tipus de relació (anomenada de forma genèrica associació) que s'ha estat explicant fins al moment. És un tipus de relació estructural que defineix les connexions entre dos o més objectes, la qual cosa permet associar objectes que instancien classes que col·laboren entre si.

2) Una **relació d'associació d'agregació** és un cas especial d'associació entre dos o més objectes. Es tracta d'una relació del tipus tot-part. Aquest tipus de relació implica dos tipus d'objectes, l'objecte anomenat base i l'objecte que estarà inclòs a l'objecte base. La relació indica que l'objecte base necessita de l'objecte inclòs per poder existir i fer les seves funcionalitats. Si desapareix l'objecte base, el o els objectes que es troben inclosos en l'objecte base no desapareixeran i podran continuar existint amb les seves funcionalitats pròpies. La relació d'associació d'agregació es representa mitjançant una línia contínua que finalitza en un dels extrems per un rombe buit, sense omplir. El rombe buit s'ubicarà a la part de l'objecte base. A la figura 1.13 es pot observar un exemple de relació d'associació d'agregació. L'objecte base és l'objecte anomenat `Fruiteria`. Els objectes inclosos a la fruiteria són: `Pomes`, `Taronges`, `Peres` i `Maduixes`. S'estableix

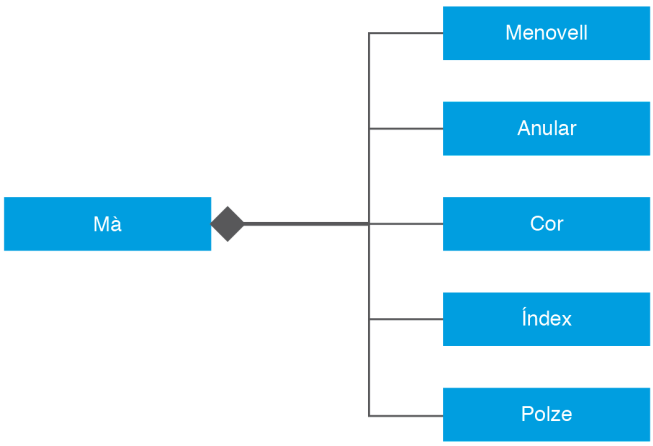
una relació entre aquestes classes del tipus tot-part, on les fruites són part de la fruiteria. Això sí, si la fruiteria deixa d'existir, les fruites continuen existint.

Figura 1.13. Relació d'associació d'agregació



3) Una **relació d'associació de composició** és també un cas especial d'associació entre dos o més objectes. És una relació del tipus tot-part. És una relació molt semblant a la relació d'associació d'agregació, amb la diferència que hi ha una dependència d'existència entre l'objecte base i l'objecte (o els objectes) que hi està inclòs. És a dir, si deixa d'existir l'objecte base, deixarà d'existir també el o els objectes inclosos. El temps de vida de l'objecte inclòs depèn del temps de vida de l'objecte base. La relació d'associació de composició es representa mitjançant una línia contínua finalitzada en un dels extrems per un rombe pintat, omplert. El rombe pintat s'ubicarà a la part de l'objecte base. A la figura 1.14 es mostra un exemple d'una relació d'agregació. L'objecte base `Mà` es compon dels objectes inclosos `Menovell`, `Anular`, `Cor`, `Índex` i `Polze`. Sense l'objecte `Mà` la resta d'objectes deixaran d'existir.

Figura 1.14. Relació d'associació de composició



4) Un altre tipus de relació entre classes és la **relació de dependència**. Aquest tipus de relació es representa mitjançant una fletxa discontínua entre dos elements. L'objecte del qual surt la fletxa es considera un objecte dependent. L'objecte al

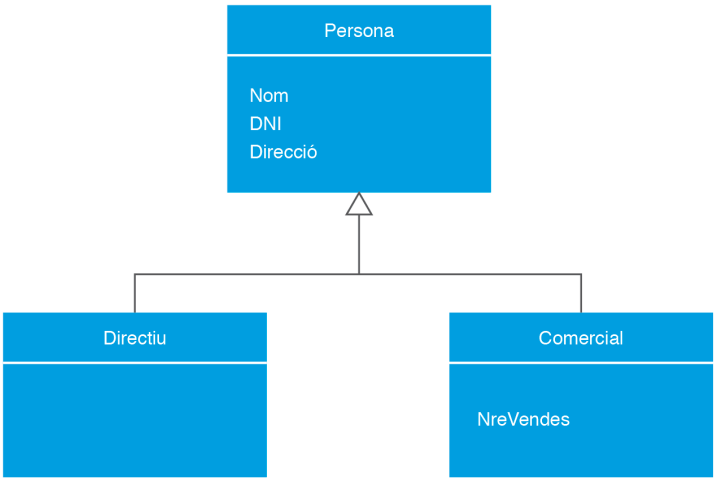
qual arriba la fletxa es considera un objecte independent. Es tracta d'una relació semàntica. Si hi ha un canvi en l'objecte independent, l'objecte dependent es veurà afectat.

5) La relació de generalització es dóna entre dues classes on hi ha un vincle que es pot considerar d'herència. Una classe és anomenada classe *mare* (o superclasse). L'altra (o les altres) són les anomenades classes *filles* o subclasses, que hereten els atributs i els mètodes i comportament de la classe *mare*. Aquest tipus de relació queda especificat mitjançant una fletxa que surt de la classe *filla* i que acaba a la classe *mare*. A la figura 1.15 es pot veure un exemple on l'element `actor` és independent. Hi ha una dependència normal de l'element `pel·lícula` en relació amb l'element `actor`, ja que `actor` es fa servir com a paràmetre als mètodes `afegir` i `eliminar` de `pel·lícula`. Si hi ha canvis a `actor`, l'objecte `pel·lícula` es veurà afectat.

Figura 1.15. Relació de dependència

L'herència es dóna a partir d'aquestes relacions de dependència. Ofereixen com a punt fort la possibilitat de reutilitzar part del contingut d'un objecte (el considerat superclasse), estenent els seus atributs i els seus mètodes a l'objecte *fill*. A la figura 1.16 es pot veure un exemple d'herència.

Figura 1.16. Relació d'herència

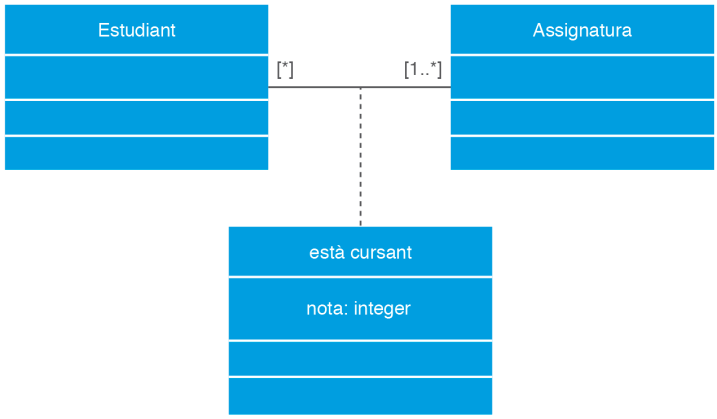


Classe associativa

Quan una associació té propietats o mètodes propis es representa com una classe unida a la línia de l'associació per mitjà d'una línia discontínua. Tant la línia com el rectangle de classe representen el mateix element conceptual: l'associació.

En el cas de l'exemple de la figura 1.17 ens trobem que la nota està directament relacionada amb les classes `Estudiant` i `Assignatura`. Cada un dels alumnes de l'assignatura tindrà una determinada nota. La manera de modelitzar l'UML aquesta situació és amb les classes associades.

Figura 1.17. Exemple de classe associativa



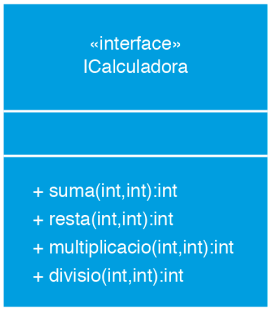
Interfície

Una **interfície** conté la declaració de les operacions sense la seva implementació, que hauran de ser implementades per una classe o component.

Arribats a aquest punt, ens podríem preguntar: Quina diferència hi ha entre una interfície i una classe abstracta?

Una interfície és simplement una llista de mètodes no implementats, així com la declaració de possibles constants. Una classe abstracta, a diferència de les interfícies, pot incloure mètodes implementats i no implementats. A la figura 1.18 es mostra un exemple d'una interfície representada en UML.

Figura 1.18. Interfície en UML



Aquesta interfície s'anomena ICalculadora, i conté els mètodes `suma`, `resta`, `multiplicacio` i `divisio`.

En el codi següent es mostra la definició d'aquesta interfície a partir de la figura 1.18.

```
interface ICalculadora {
    public abstract int suma (int x, int y);
    public abstract int resta (int x, int y);
    public abstract int multiplicacio (int x, int y);
    public abstract int divisio (int x, int y);
}
```

Una vegada definida es mosta com serà la utilització de la interfície definida en el codi següent.

```
public class Calculadora implements ICalculadora {
        public int suma (int x, int y){
            return x + y;
        }
        public int resta (int x, int y){
            return x - y;
        }
        public int multiplicacio (int x, int y){
            return x * y;
        }
        public int divisio (int x, int y){
            return x / y;
        }
}
```

1.2.3 Diagrama d'objectes

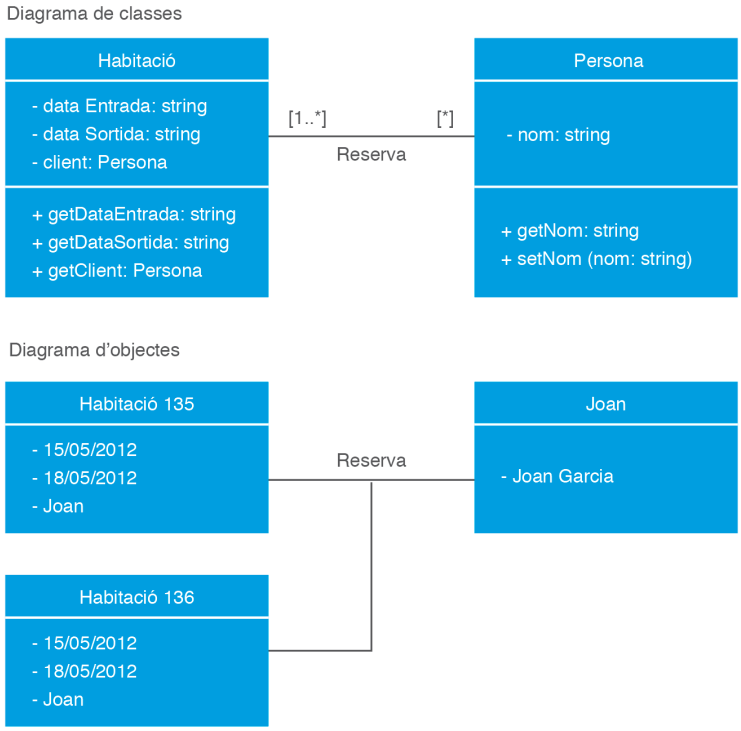
Un **objecte** és una instància d'una classe, per la qual cosa es pot considerar el diagrama d'objectes com una instància del diagrama de classes.

> El **diagrama d'objectes** només pot contenir instàncies i relacions entre objectes: enllaços i dependències que tinguin lloc entre instàncies. Els classificadors i les associacions respectives han d'haver estat definits prèviament en un diagrama de classes.

Sovint té la funció de simple exemple d'un diagrama de classes o d'una part d'ell.

A la figura 1.19 es mostra un diagrama de classes consistent en dues classes, la classe `Habitació` i la classe `Persona`. A partir d'aquest diagrama es crea i es mostra el diagrama d'objectes a la mateixa figura 1.19.

FIGURA 1.19. Diagrama d'objectes



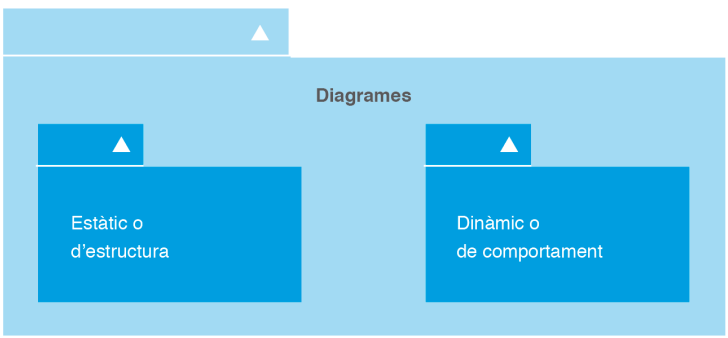
1.2.4 Diagrama de paquets

Els **paquets** permeten organitzar els elements del model en grups relacionats semànticament; un paquet no té un significat per si mateix.

> El **diagrama de paquets** serveix per descriure l'estructura d'un model en termes de paquets interrelacionats.

A la figura 1.20 es mostra un exemple de diagrama de paquets. S'hi ha representat un paquet, `Diagrames`, que conté dos paquets: `estàtic o d'estructura` i `dinàmic o de comportament`.

FIGURA 1.20. Diagrama de paquets



1.2.5 Diagrama d'estructures compostes

Una **estructura composta** és un conjunt d'elements interconnectats que col·laboren en temps d'execució per aconseguir algun propòsit.

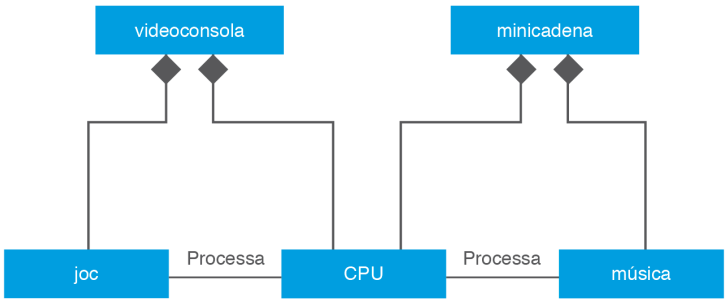
La finalitat del diagrama d'estructures compostes és descriure objectes compostos amb la màxima precisió possible. Es tracta d'un diagrama que complementa el diagrama de classes.

A continuació es descriurà, fent ús d'un exemple, el concepte del diagrama d'estructures compostes.

Si es vol modelitzar un sistema que té una videoconsola i una minicadena, ambdues estan compostes per una CPU (Unitat Central de Procés), però en la videoconsola la CPU processa el joc que serà visualitzat en algun dispositiu de sortida (com, per exemple, un televisor) i, en el segon cas, la CPU processa la música perquè pugui ser escoltada.

Una possible forma de modelitzar el sistema podria ser la que mostra la figura 1.21.

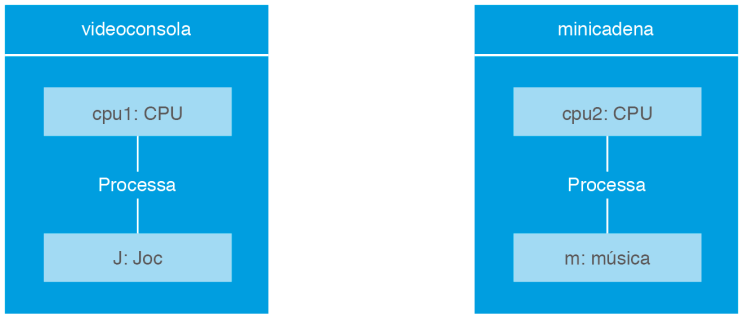
FIGURA 1.21. Diagrama de classes-composició



Però si es revisa amb detall el diagrama de la figura 1.21 es pot observar que té deficiències, ja que fàcilment es pot interpretar que la CPU de la minicadena pot processar música i jocs.

UML 2.0 ha introduït un nou diagrama, anomenat *diagrama d'estructura composta*, que permet concretar les parts que componen una classe. Aquest permetrà representar el sistema amb el diagrama que es mostra en la figura 1.22, que delimita l'abast de la videoconsola i de la minicadena.

FIGURA 1.22. Diagrama d'estructura composta



1.2.6 Diagrama de components

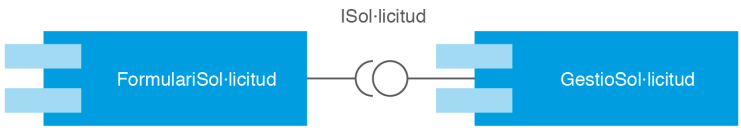
Interfície
Una interfície conté la declaració de les operacions sense la seva implementació, les quals hauran de ser implementades per una classe o component.

Un **component** és una peça del programari que conforma el sistema, peça que pot ser reutilitzable i fàcilment substituïble.

Un component sol fer ús d'una interfície que defineix les operacions que el component implementarà.

El **diagrama de components** mostra els components que conformen el sistema i com es relacionen entre si. A la figura 1.23 es pot veure un exemple d'un diagrama de components.

FIGURA 1.23. Diagrama de components



1.2.7 Diagrama de desplegament

El **diagrama de desplegament** descriu la distribució de les parts d'una aplicació i les seves interrelacions, tot en temps d'execució.

El diagrama de desplegament descriu la configuració d'un sistema de programari en temps d'execució, en termes de recursos de maquinari i dels components de programari, processos i objectes (en memòria o emmagatzemats en bases de dades, per exemple) que s'hi hostatgen.

Tot seguit es mostren diversos exemples de diagrames de desplegament.

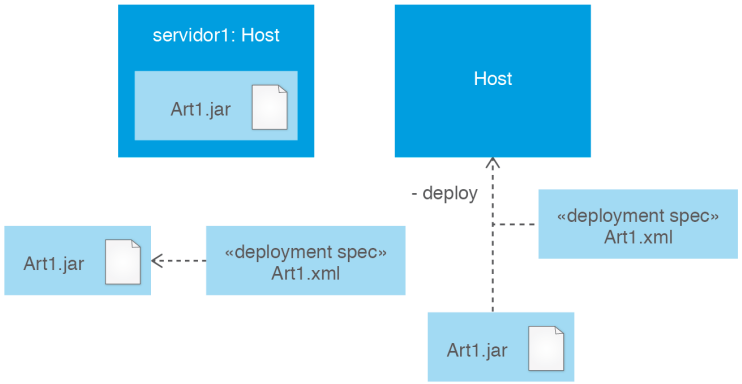
En la figura 1.24 hi ha els nodes `Servidor` i `Client`, considerats a nivell de classificador, i una línia de comunicacions. En aquesta, cada instància de Client

pot bescanviar informació només amb una instància de `Servidor`, mentre que una instància de `Servidor` en pot bescanviar informació almenys amb una instància de `Client`, però podrà fer-ho amb més.

Figura 1.24. Nodes i línia de comunicacions

Que un artefacte es desplegui dins d'un node es pot representar, o bé incloent el símbol de l'artefacte dins el del node, o bé amb el desplegament de l'artefacte cap al node. La figura 1.25 mostra un exemple de cada opció, la primera amb instàncies i la segona amb classificadors.

Figura 1.25. Un desplaçament i els seus elements



1.3 Modelio i UML: notació dels diagrames de classes

Per poder entendre millor els diagrames de classes i la seva notació, es mostra a continuació un exemple complet. A més a més, s'ha fet servir l'entorn de modelatge Modelio per a la creació d'aquests diagrames UML. Aquest entorn està especialitzat en el modelatge amb UML i altres llenguatges gràfics. També permet tant generar codi a partir dels diagrames com generar diagrames a partir de codi font. L'entorn Modelio ha estat implementat a partir de l'IDE Eclipse.

La pàgina web de l'entorn Modelio és www.modelio.org. Podeu descarregar-vos l'instal·lable des de l'apartat *Downloads*. Veureu que els requisits (que s'enllacen a la mateixa pàgina) no són gaire exigents. L'entorn Modelio pot utilitzar-se des dels sistemes operatius Windows, Linux i MacOS X. En aquest darrer cas, no existeix programa d'instal·lació i aquesta es limita a descomprimir els fitxers. Es farà servir el format d'exercici per oferir les indicacions, pas a pas, de com crear un diagrama de classes a partir dels requeriments d'un enunciat.

Enunciat de l'exemple

Les factures dels proveïdors d'una empresa poden ser de proveïdors de serveis o de proveïdors de productes o materials. Cadascuna de les factures, de qualsevol dels dos tipus, tenen en comú:

- El número de la factura.
- La data de la factura.
- L'import total –que es calcula de manera diferent en les unes que en les altres-.
- Les dades del client, que són el NIF i el nom, i que s'hauran de trobar en una classe a part.
- El detall de la factura (tant si és de materials com si és de serveis).

Tant per a les factures de serveis com per a les factures de productes o materials cal tenir guardada la darrera factura emesa, per tenir present el número de factura. Hi ha una llista de serveis amb la descripció i el preu per hora de cadascun.

En el detall de les factures haurà d'haver-hi:

- A les factures de serveis, a cada factura hi ha una llista de serveis amb la data de prestació, el nombre d'hores dedicades i el preu/hora per servei. En una factura hi pot haver més d'una prestació del mateix servei.
- A les factures de productes, per a cada producte (a cada factura n'hi ha almenys un) haurà d'haver-hi la descripció, el preu unitari i la quantitat.

1.3.1 Creació del projecte

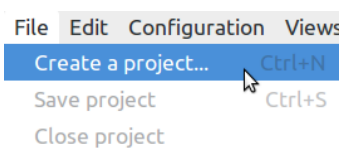
El primer cop que obriu el programa, us apareixerà una finestra de benvinguda. Podeu tancar-la després de, opcionalment, explorar les opcions que ofereix.

Un cop tancada aquesta finestra, cal seguir les següents passes:

- Seleccioneu l'opció del menú *File* / *Create a Project...* tal com es mostra a la figura 1.26.

De manera alternativa, també es pot crear un projecte fent clic a la icona de Crear Projecte

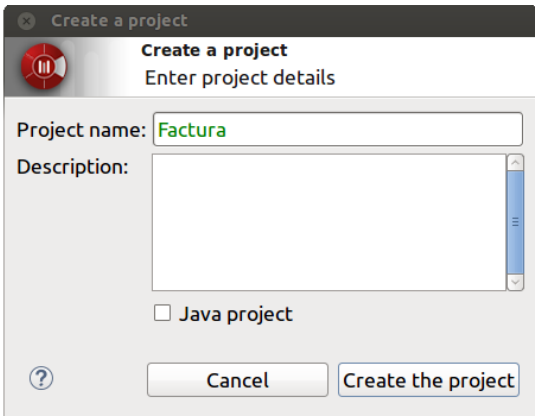
FIGURA 1.26. Opció Create a Project...



L'opció *Java project* activa el dissenyador Java. La seva finalitat és tenir sincronitzat el codi Java amb els diagrames UML. En aquest exemple no utilitzarem aquesta opció.

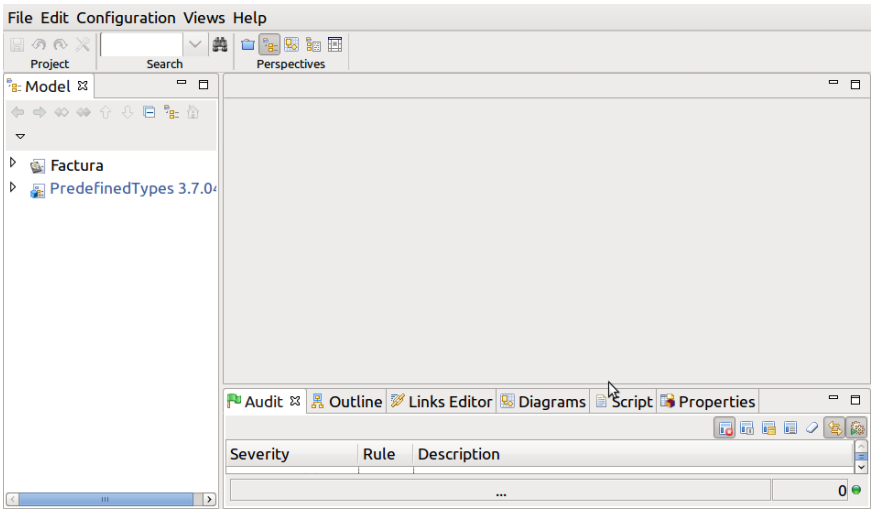
- Ens apareixerà una finestra com la que ens mostra la figura 1.27. Entreu-hi el nom, si voleu, una descripció i feu clic al botó *Create the projecte*

FIGURA 1.27. Definició del projecte



La pantalla resultant es mostra a la figura 1.28

FIGURA 1.28. Projecte nou

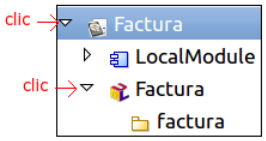


1.3.2 Creació del diagrama de classes

Un cop creat el projecte *Factura*, a la mateixa pantalla que mostra la figura 1.28, podem crear un diagrama de classes seguint les següents passes:

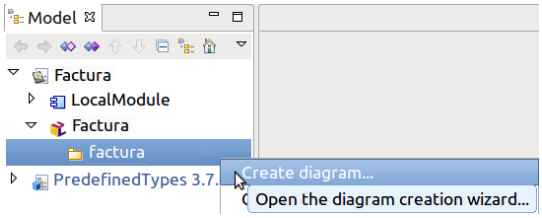
- Fer clic als triangles que hi ha a l'esquerra de les icones, de manera que aparegui la icona amb forma de carpeta que representa el projecte, tal com es mostra a la figura 1.29.

FIGURA 1.29. Accés a la carpeta que representa el projecte



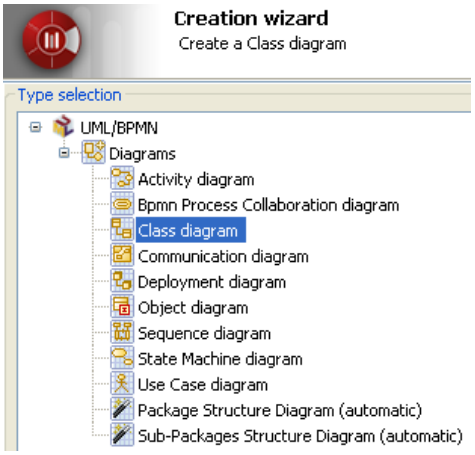
- Fer clic amb el botó secundari a sobre de la icona que representa el projecte (*Factura*, al nostre cas) i seleccionar *Create diagram...*, tal com es mostra a la figura 1.30

FIGURA 1.30. Creació d'un diagrama



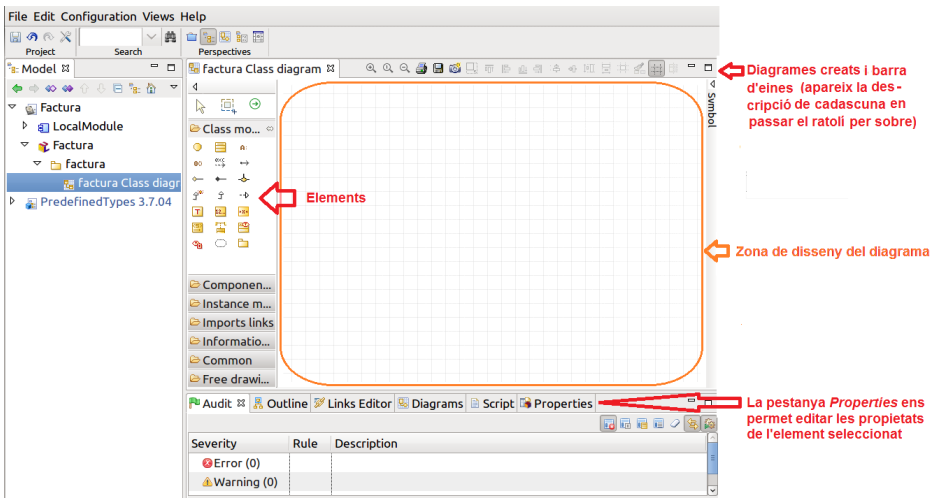
- Triar el tipus de diagrama que vulguem -*Class diagram*, al nostre cas- i fer clic a OK, tal com es mostra a la figura 1.31

FIGURA 1.31. Selecció del tipus de diagrama a crear



Ens apareix a la finestra principal una pestanya amb el nou diagrama. La figura 1.32 mostra les diferents seccions de la pantalla, un cop creat aquest nou diagrama.

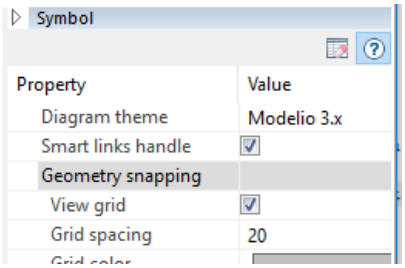
FIGURA 1.32. Seccions d'un diagrama



Per últim, és recomanable deshabilitar la característica *Smart links handle*, ja que no la utilitzarem i, a més, ens pot dificultar les accions. Es fa de la següent manera:

- Fer clic a una zona del diagrama on no hi hagi cap element.
- Fer clic a la columna *Symbol*, que es troba a la part dreta de la finestra, perquè es desplegui. La figura 1.33 ens mostra les propietats d'aquesta columna.
- Deshabilitar la propietat *Smart links handle* i, si volem, tornar a fer clic a la barra superior de la columna *Symbol* perquè es torni a plegar.

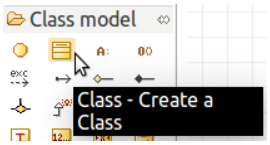
FIGURA 1.33. Propietats de la columna Symbol



1.3.3 Creació de classes. Relació d'herència

L'enunciat especifica: "cadascuna de les factures d'una empresa és, o bé una factura de serveis o bé una factura de productes". Podem observar que es tracta d'una generalització o herència, on la classe pare correspon a `Factura` i les classes *filles* a `Factura de serveis` i `Factura de productes`.

En primer lloc, cal crear les classes. Això es fa:

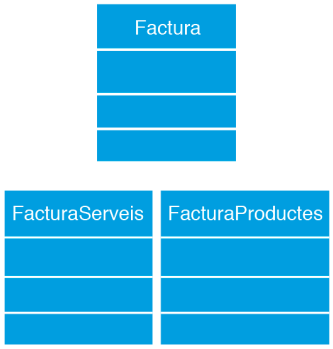


Icona Class

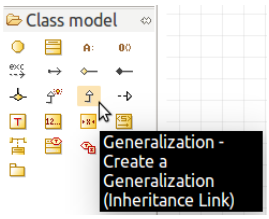
- Fent clic a sobre de la icona *Class*.
- Fent clic, a continuació, al punt del diagrama on volem inserir la classe.

Això cal fer-ho per a cadascuna de les tres classes. A la figura 1.34 es mostren aquestes classes.

FIGURA 1.34. Classes Factura, FacturaServeis i FacturaProducte



Queda pendent crear la relació d'herència entre les classes. Es fa de la següent manera:

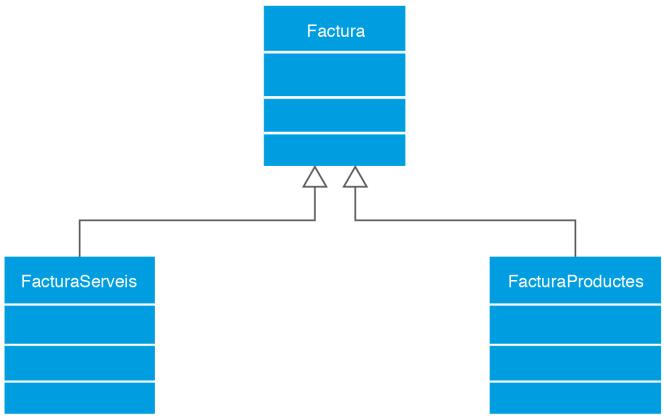


Icona Generalization

- Fent clic a la icona *Generalization*.
- A continuació, fent clic, **en aquest ordre** a una de les subclasses, per exemple a `FacturaServeis`, i, després, a la superclasse, `Factura`. Ens apareixerà una fletxa.
- Després, fent clic novament a la icona *Generalization*.
- Per últim, fent clic a l'altra subclasse, `FacturaProductes` i, a continuació, a la fletxa que ens ha aparegut al segon pas. Quan passem el ratolí per sobre, veureu que es posa de color verd; això significa que s'hi pot fer clic.
- Si hi hagués més subclasses, caldria repetir els dos passos anteriors per a cada subclasse.

El diagrama resultant és el que es mostra en la figura 1.35.

FIGURA 1.35. Relació d'herència entre classes



Per facilitar la realització dels diagrames, pot ser útil conèixer algunes operacions bàsiques que podem realitzar sobre els seus elements:

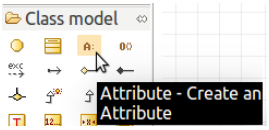
- **Seleccionar un element:** n'hi ha prou amb fer-hi clic a sobre o, si volem seleccionar-ne més d'un a la vegada, podem bé fer clic a sobre d'ells, un rera l'altre, mentre mantenim la tecla *Ctrl* polsada, bé assenyalar, tot arrosegant el ratolí, una àrea que inclogui a tots els elements a seleccionar.

- **Canviar el nom d'un element:** seleccionarem l'element al qual volem canviar el nom i, a continuació, farem clic a sobre del seu nom; una altra forma de fer-ho és seleccionar l'element i fer clic novament al mateix element; s'obrirà una finestra que ens permetrà canviar les propietats de l'element, entre elles, el nom.

1.3.4 Creació dels atributs d'una classe

Si continuem interpretant l'enunciat, ens indica: "les unes i les altres tenen en comú el número, la data i l'import". D'aquesta manera, els atributs de la classe `Factura` són: `número`, `data` i `import`.

Modelio permet afegir els atributs a les classes seguint els següents passos:



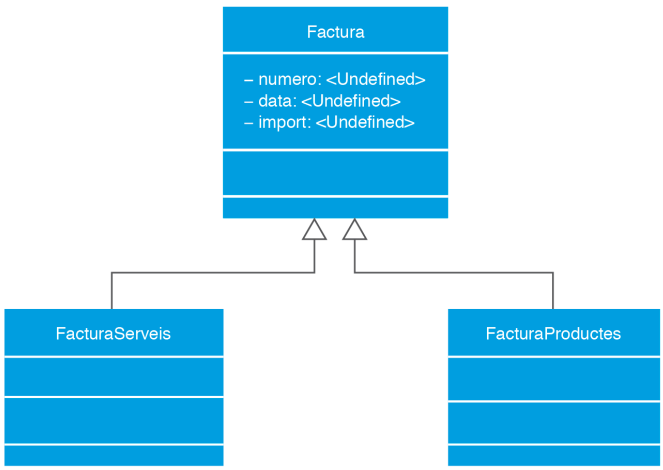
Icona Attribute

- Fer clic a sobre de la icona *Attribute*.

- A continuació, fer clic a sobre de la **classe** on volem afegir aquesta propietat.

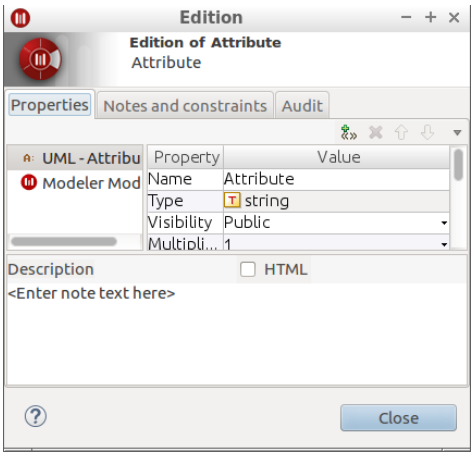
En la figura 1.36 es pot observar com ha de quedar el diagrama de classes.

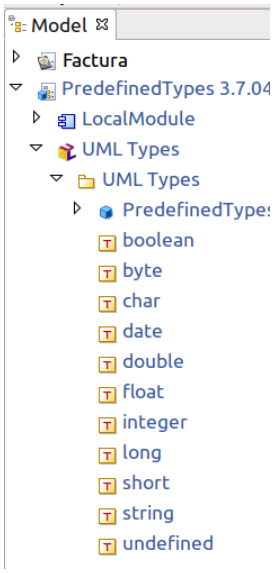
FIGURA 1.36. Atributs de la classe Factura



Ara cal especificar, per a cada atribut, el seu tipus i la seva visibilitat. Aquesta darrera, en principi, serà *private* a tots els casos. Per aconseguir-ho, cal fer doble clic a l'atribut que volem editar. Sortirà una finestra com la que mostra la figura 1.37.

FIGURA 1.37. Edició d'un Attribute





Tipus UML predefinits

A la pestanya *Properties* podrem editar el tipus, la visibilitat i altres característiques, que deixarem amb els seus valors per defecte. En fer-ho, cal tenir en compte el següent:

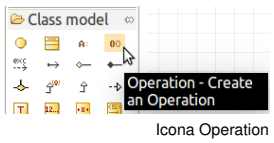
- **Tipus (Type):** podem escriure qualsevol dels tipus UML predefinits. Els podeu veure a la pestanya Model de la part esquerra de la pantalla.
- Després d'entrar el nom del tipus, cal picar la tecla d'introducció per a que el sistema validi el tipus i, a més, enregistri el canvi realitzat. També pot entrar-se el nom del tipus deixant el quadre en blanc o amb la o les primeres lletres del tipus i, a continuació, picar la combinació de les tecles *Ctrl* i *barra espaiadora*; això farà apareixer un desplegable amb els tipus que comencen amb les mateixes lletres que hem entrat.

1.3.5 Creació dels mètodes d'una classe

Modelio anomena *operations* al que per a nosaltres són mètodes.

L'enunciat ens indica que l'import es calcula de manera diferent en les unes que en les altres, és a dir el càlcul de l'import serà diferent per a la classe `FacturaServeis` i per a la classe `FacturaProductes`.

D'aquesta manera, el mètode `CalculImport()` de la classe `Factura` es definirà com un mètode abstracte que serà definit en cada una de les classes filles.



Icona Operation

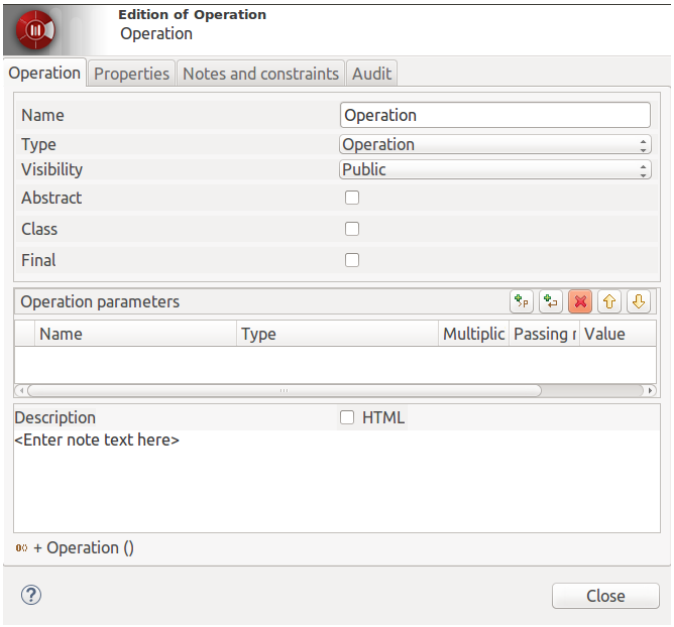
La definició dels mètodos es fa de manera similar a la dels atributs:

- Es fa clic a sobre de la icona *Operation*.
- Es fa clic a sobre de la classe on volem posar aquest mètode.

Ara cal indicar el nom del mètode, els paràmetres i el valor de retorn. Es fa des de la finestra d'edició. S'accedeix a ella fent doble clic a sobre del seu nom.

Ens apareixerà una finestra com la que indica la figura figura 1.38, amb la pestanya *Operation* oberta.

Figura 1.38. Edició d'un mètode

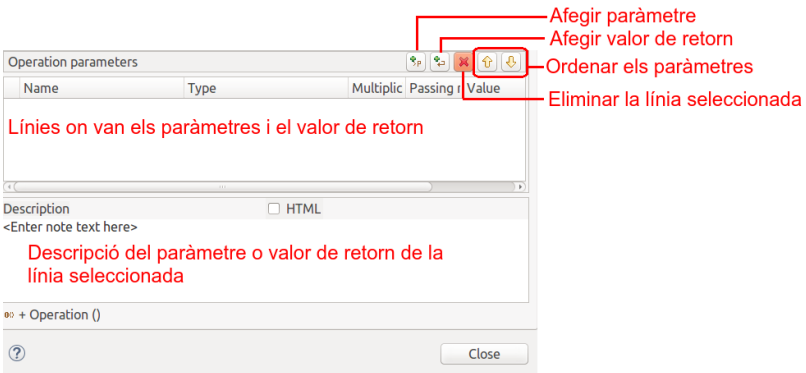


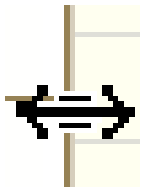
En ella podrem especificar:

- el nom
- la visibilitat
- si és abstracta
- si és final
- els paràmetres
- el valor de retorn

Les icones que apareixen a l'apartat *Operation parameters* ens permeten editar els paràmetres i el valor de retorn. Cadascun té la funció que es mostra a la figura 1.39.

Figura 1.39. Edició dels paràmetres i valor de retorn d'un mètode



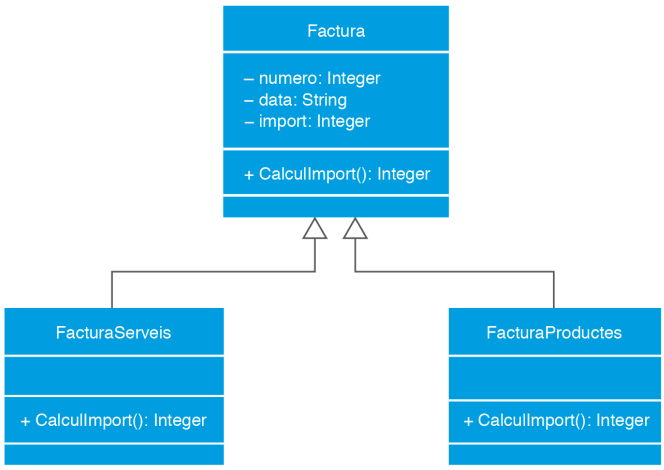


Forma del cursor del ratolí quan està a sobre d'un d'aquests requadres negres i indica que es pot reajustar la classe.

Després d'afegir un mètode a una classe, cal assegurar-se que aquesta té prou amplada per a mostrar tots els seus paràmetres; en cas contrari, cal reajustar la seva mida. Això es fa seleccionant la classe i arrossegant amb el ratolí un dels petits quadres negres que hi apareixen.

D'aquesta manera, el diagrama resultant analitzat fins al moment és el que es mostra en la figura 1.40.

FIGURA 1.40. Diagrama de classes

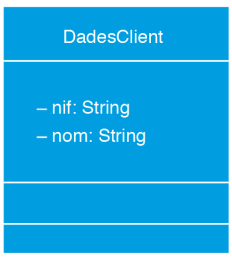


1.3.6 Agregació

Seguint amb l'enunciat, es diu: "les dades del client, que són el NIF i el nom, i són en una classe a part".

És necessari crear una nova classe anomenada `DadesClient`, que contindrà els atributs `NIF` i `nom`. Es pot observar el diagrama d'aquesta classe en la figura 1.41.

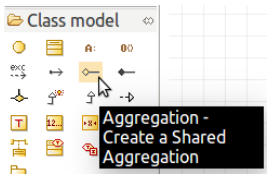
FIGURA 1.41. Classe DadesClient



El tipus d'associació entre la classe `Factura` i `DadesClient` és d'agregació, ja que la classe `DadesClient` és un objecte que únicament podrà ser creat si existeix l'objecte agregat `Factura` del qual ha de formar part.

Amb Modelio es crea seguint els següents passos:

- Fer clic a la icona *Aggregation* de la paleta.



Icona Aggregation

- Fer clic, en primer lloc, a la classe `Factura` i, després, a la classe `DadesClient`.

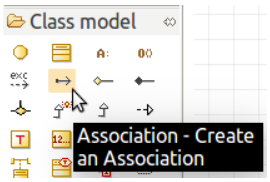
- Fer doble clic a la línia que representa l'agregació i editar, segons necessitem, les propietats de l'agregació (especialment, les propietats *navegable* i les cardinalitats).

- Si s'hagués d'incloure més classes a l'agregació, caldria:

  – Fer clic a la icona que representa una associació (*Association*).

  – Fer clic a la classe que volem afegir a l'agregació.

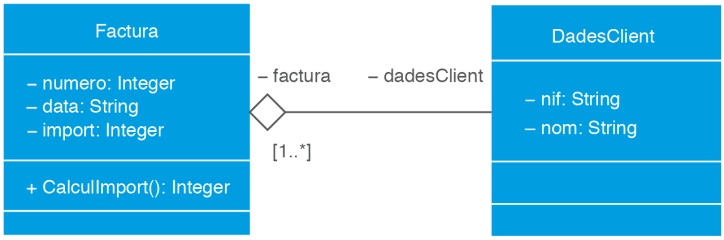
  – Fer clic a la línia que representa l'agregació.



Icona Association

El diagrama resultant analitzat fins al moment és el que es mostra en la figura 1.42.

Figura 1.42. Diagrama de classes

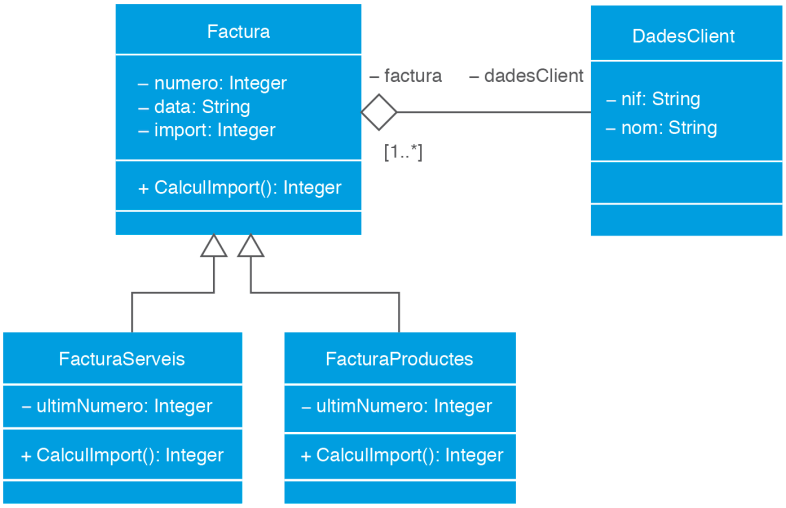


D'altra banda, l'enunciat especifica: "es guarda, d'una banda, l'últim número de factura de serveis i, de l'altra, l'últim número de factura de productes".

És necessari crear un nou atribut ultimNumero en la classe `FacturaServeis` i en la classe `FacturaProductes`.

El diagrama resultant analitzat fins al moment és el que es mostra en la figura 1.43.

FIGURA 1.43. Diagrama de classes

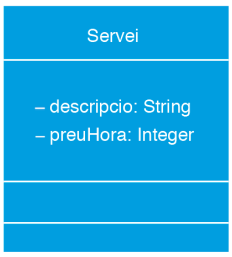


1.3.7 Classe associada

L'enunciat continua explicant: "Hi ha una llista de serveis amb la descripció i el preu per hora de cadascun".

Per tant, és necessari crear una nova classe anomenada `Servei` que contindrà dos atributs: `descripcio` i `preuHora` (preu per hora), com es pot observar en la figura 1.44.

FIGURA 1.44. Classe Servei



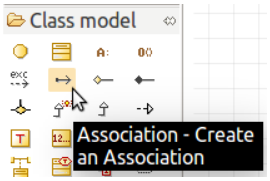
L'enunciat continua: "cada factura de serveis té una llista de serveis amb la data de prestació i el nombre d'hores, i en una factura hi pot haver més d'una prestació del mateix servei".

Per tant, és necessari crear una classe associada anomenada `Hores` que contindrà dos atributs: `dataPrestació` (data de prestació) i `nombre` (nombre d'hores), com es pot observar a la figura 1.45.

Figura 1.45. Classe Hores

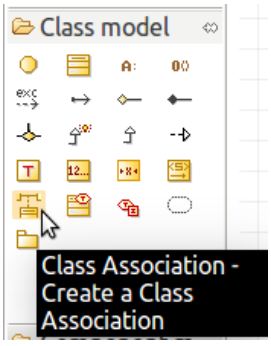
Ara cal crear l'associació entre les classes:

- Crearem una associació entre les classes `Servei` i `FacturaServeis` així:
  - Farem clic a la icona *Association*.
  - A continuació, farem clic a les dues classes consecutivament.



Icona Association

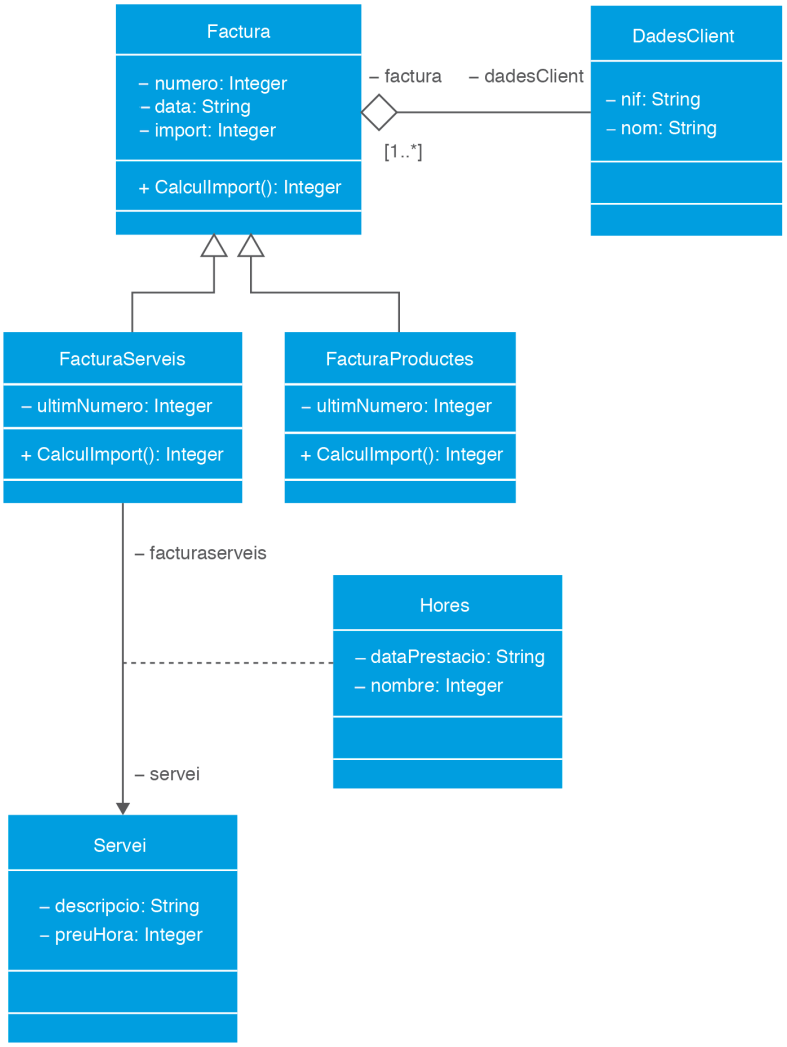
- Indicarem que la classe `Hores` és la classe associada:
  - Farem clic a la icona *classe associativa*.
  - Farem clic a la associació que acabem de crear.
  - Farem clic a la classe `Hores`.



Icona Classe associativa

El diagrama resultant és el que es mostra en la figura 1.46.

FIGURA 1.46. Diagrama de classes

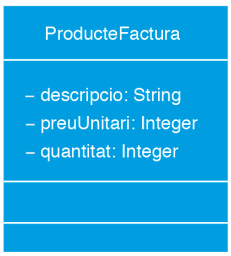


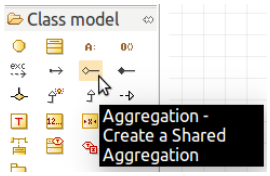
1.3.8 Composició

Continuant amb l'enunciat, aquest ens especifica: "Les factures de productes, per a cada producte (a cada factura n'hi ha almenys un) tenen la descripció, el preu unitari i la quantitat".

Per tant, és necessari crear una nova classe anomenada `ProducteFactura`, que contindrà els atributs `descripcio`, `preuUnitat` (preu unitari) i `quantitat`, com es pot observar en la figura 1.47.

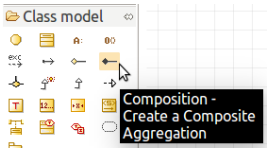
Figura 1.47. Classe ProducteFactura





Icona Aggregation

En aquest cas es tracta d'una composició, ja que és una relació més forta que l'agregació. `FacturaProductes` té sentit per ell mateix, però està compost de `ProducteFactura` (amb la descripció, preu i quantitat de la factura), aquesta relació és més forta que l'associació d'una agregació.

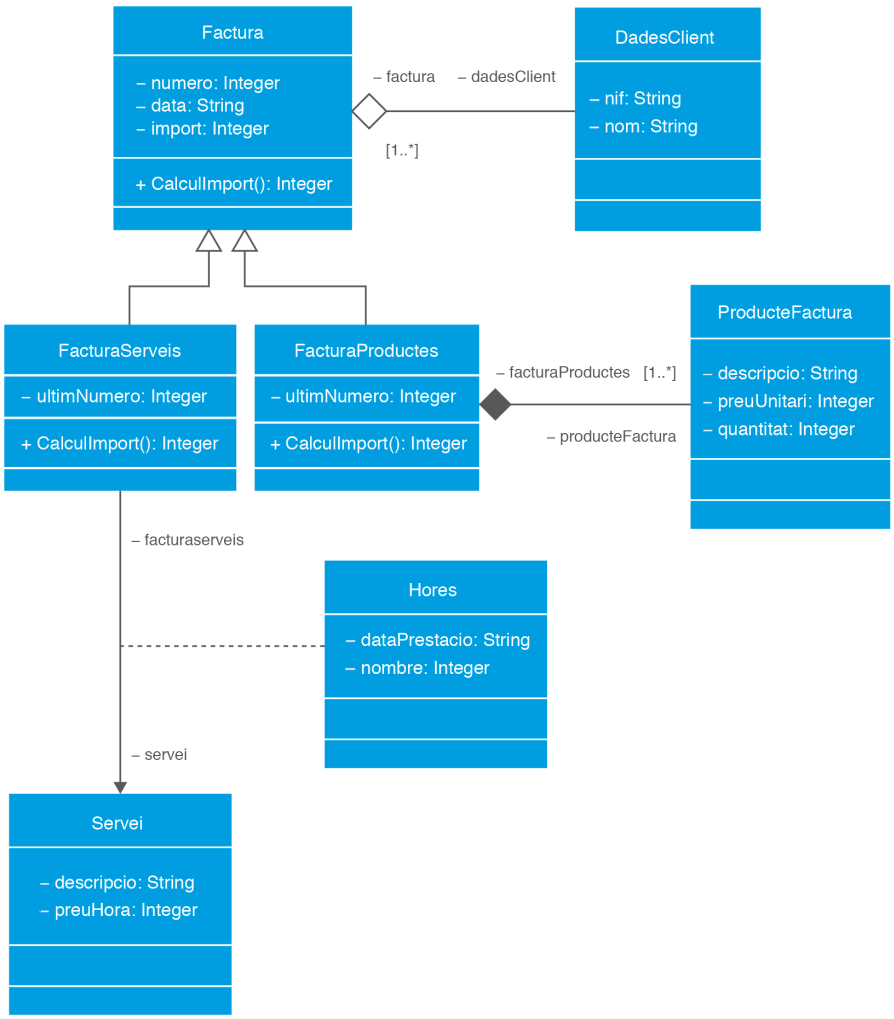


Icona Composition

Es crea igual que l'agregació, amb l'única diferència que, en lloc d'utilitzar la icona *Aggregation*, cal utilitzar la icona *Composition*.

El diagrama resultant de l'enunciat és el que es mostra en la figura 1.48.

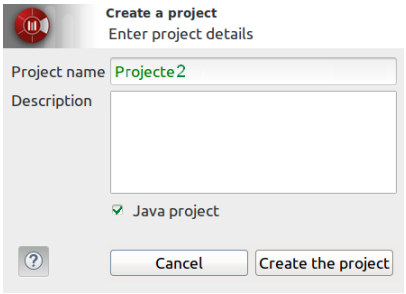
Figura 1.48. Diagrama de classes



1.3.9 Generació de codi a partir d'un diagrama de classes

Si volem poder generar codi a partir del diagrama de classes, en primer lloc, quan creem el projecte cal seleccionar l'opció *Java Project*, com es veu a la figura 1.49.

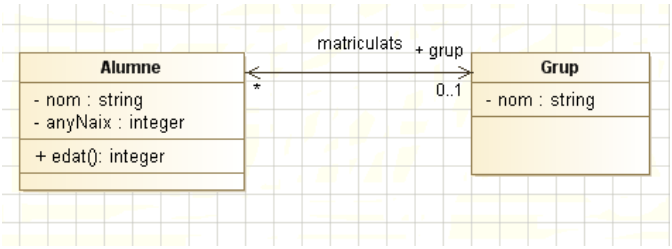
FIGURA 1.49. Creació d'un projecte que permet generar codi en Java



Es farà servir el format d'exemple per oferir les indicacions, pas a pas, de com generar codi a partir d'un diagrama de classes.

El diagrama de classes de partida serà el mostrat a la figura 1.50.

FIGURA 1.50. Diagrama de classes



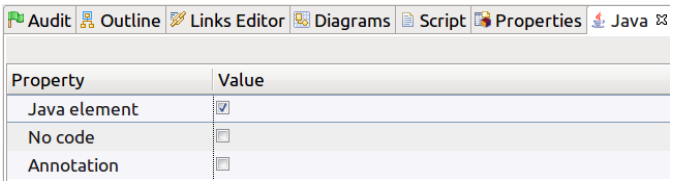
Atenció a les relacions

En fer aquest exemple, cal parar atenció a les cardinalitats. Cal fixar-se també que s'ha assenyalat l'associació com a navegable pels dos costats perquè es generin els membres que permetin aquesta doble navegació.

Un cop creat el diagrama, cal fer que tots els elements que apareixen es considerin com a *elements Java*. Es fa de la següent manera:

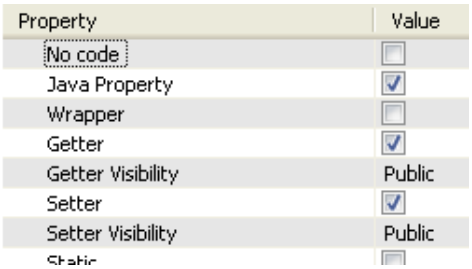
- **Classes**: cal seleccionar la classe i, a la pestanya *Java* de la part inferior, activar la casella *Java element*, com es veu a la figura 1.51.

FIGURA 1.51. Configuració d'una classe com a element Java



- **Propietats**: també cal seleccionar-les i configurar les caselles *Java Property*, *Getter*, *Setter*, *Getter visibility* i *Setter visibilty* de manera que quedi com a la figura 1.52.

Figura 1.52. Configuració d'una propietat com a element Java

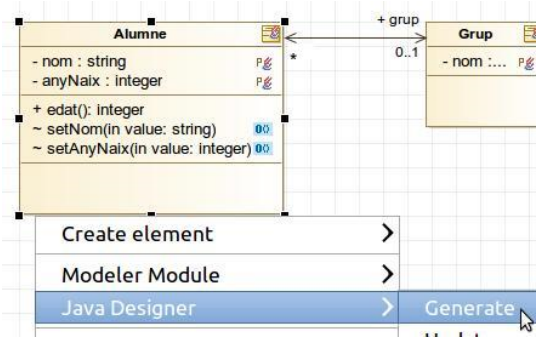


Després de definir les classes i les propietats com a *elements Java*, ho reflectiran, respectivament, les icones següents a la dreta del seu nom:

- **Associacions**: cal assenyalar-les com a *Java Property* dins de les seves propietats, de manera similar a com es fa amb els altres elements.

Per a **generar el codi** cal anar a **cada classe** i seleccionar, des del menú contextual (que apareix amb el botó secundari), l'opció *Java Designer* / *Generate*, com es veu a la figura figura 1.53.

Figura 1.53. Configuració d'una propietat com a element Java



Es pot veure el resultat tot seleccionant, al mateix menú contextual, *Java Desginer* / *Edit*. S'obrirà una pestanya amb el codi corresponent a la classe des de la qual hem triat l'opció.

A l'exemple, el resultat ha estat aquest el que es mostra a la figura 1.54.

Es pot veure que, tot i que el resultat és raonablement bo, cal fer algun retoc a mà a la classe `Grup`; concretament, l' `ArrayList` de la classe `Grup` no té nom, és públic i li falta el *getter*. A més, com és lògic, a la classe `Alumne` cal completar el mètode edat perquè calculi aquesta a partir d' `AnyNaix`.

Cal notar, també, que Modelio utilitza anotacions pròpies per incloure informació que necessita per gestionar internament els diferents elements.

FIGURA 1.54. Codi generat a partir del diagrama

```
import com.modeliosoft.modelio.javadesigner.annotations.mdl;
import com.modeliosoft.modelio.javadesigner.annotations.objid;

@objid ("ea148390-1b93-4ee6-b5ea-96e2e70982d2")
public class Alumne {
   @mdl.prop
   @objid ("b619ad4f-e045-4d54-90ec-a1e25a4a1662")
   private String nom;

   @mdl.propgetter
   public String getNom() {
      // Automatically generated method. Please do not modify this code.
      return this.nom;
   }

   @mdl.propsetter
   public void setNom(String value) {
      // Automatically generated method. Please do not modify this code.
      this.nom = value;
   }

   @mdl.prop
   @objid ("5f7344ef-9cbf-4474-a520-9c2760267bc5")
   private int anyNaix;

   @mdl.propgetter
   public int getAnyNaix() {
      // Automatically generated method. Please do not modify this code.
      return this.anyNaix;
   }

   @mdl.propsetter
   public void setAnyNaix(int value) {
      // Automatically generated method. Please do not modify this code.
      this.anyNaix = value;
   }

   @mdl.prop
   @objid ("13c5bbad-eb82-437f-8376-83a2fca94135")
   private Grup grup;

   @mdl.propgetter
   public Grup getGrup() {
      // Automatically generated method. Please do not modify this code.
      return this.grup;
   }

   @mdl.propsetter
   public void setGrup(Grup value) {
      // Automatically generated method. Please do not modify this code.
      this.grup = value;
   }

   @objid ("28f1391c-5152-4b82-890c-beb8c32fcae3")
   public int edat() {
   }

}
```

```
import java.util.ArrayList;
import java.util.List;
import com.modeliosoft.modelio.javadesigner.annotations.mdl;
import com.modeliosoft.modelio.javadesigner.annotations.objid;

@objid ("24a5fbd8-dccd-40e2-9826-8209eb4a06e6")
public class Grup {
   @mdl.prop
   @objid ("338ab93b-e6f7-400e-bf3a-f8196f24e84a")
   private String nom;

   @mdl.propgetter
   public String getNom() {
      // Automatically generated method. Please do not modify this code.
      return this.nom;
   }

   @mdl.propsetter
   public void setNom(String value) {
      // Automatically generated method. Please do not modify this code.
      this.nom = value;
   }

   @objid ("75030486-5d6f-4c37-b3ec-5cb131c6d884")
   public List<Alumne> = new ArrayList<Alumne> ();

}
```

1.3.10 Inserció de classes en un diagrama a partir del codi

En l'apartat *Generació de codi a partir d'un diagrama de classes* podeu trobar com crear un projecte definit com a *Java Project*.

Aquest apartat també seguirà el format d'exemple. En concret, afegirem dues classes a un projecte creat com a *Java Project*.

En concret, les classes que afegirem són les següents:

```
1 import java.util.ArrayList;
2 import java.util.List;
3
4 public class Botiga {
5
```

```
  private String nom;

  private String adreca;

  private List<ClientHabitual>
 clients = new ArrayList<ClientHabitual>();

  public String getNom(){
    return nom;
  }

  public void setNom(String nom){
    this.nom=nom;
  }

  public String getAdreca(){
    return adreca;
  }

  public void setAdreca(String
 adreca){
    this.adreca=adreca;
  }

  public List<ClientHabitual>
 getClients(){
    return this.clients;
  }
}
```

```
public class ClientHabitual {

  private String nom;

  private String adreca;

  private Botiga botiga;

  public String getNom(){
    return nom;
  }

  public void setNom(String nom){
    this.nom=nom;
  }

  public String getAdreca(){
    return adreca;
  }

  public void setAdreca(String
 adreca){
    this.adreca=adreca;
  }

  public Botiga getBotiga(){
    return botiga;
  }

  public void setBotiga(Botiga
botiga){
    this.botiga=botiga;
  }

}
```

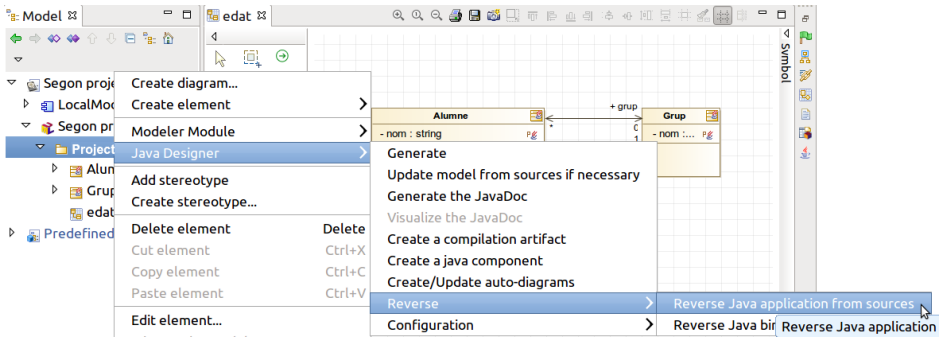
Per fer-ho, en primer lloc cal crear a la subcarpeta *src* del projecte un fitxer *.java* per a cadascuna de les classes, amb el contingut corresponent. La carpeta arrel del

projecte es troba seleccionant el projecte a l'explorador i clicant la icona *Project configuration*.

Icona Project configuration

A continuació, cal seleccionar l'opció *Java Designer* / *Reverse* / *Reverse Java application from sources* com es mostra a la figura 1.55.

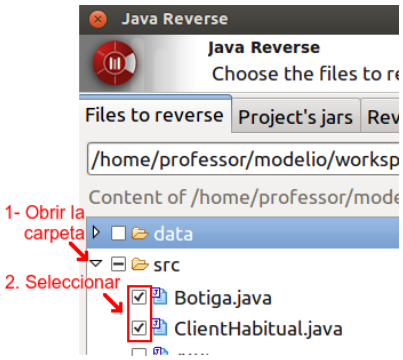
FIGURA 1.55. Opció per generar classes UML a partir de codi font



També poden importar-se fitxers *.class* o *.jar*.

A la pantalla que ens apareix, cal seleccionar els fitxers que contenen les classes que volem afegir al nostre projecte, com es mostra a la figura 1.56.

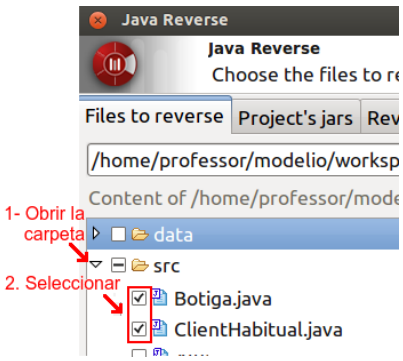
FIGURA 1.56. Selecció dels fitxers de les classes a incorporar al diagrama



Després, cal seguir l'assistent clicant al botó *Next* un parell de vegades, tot deixant les opcions que surten per defecte.

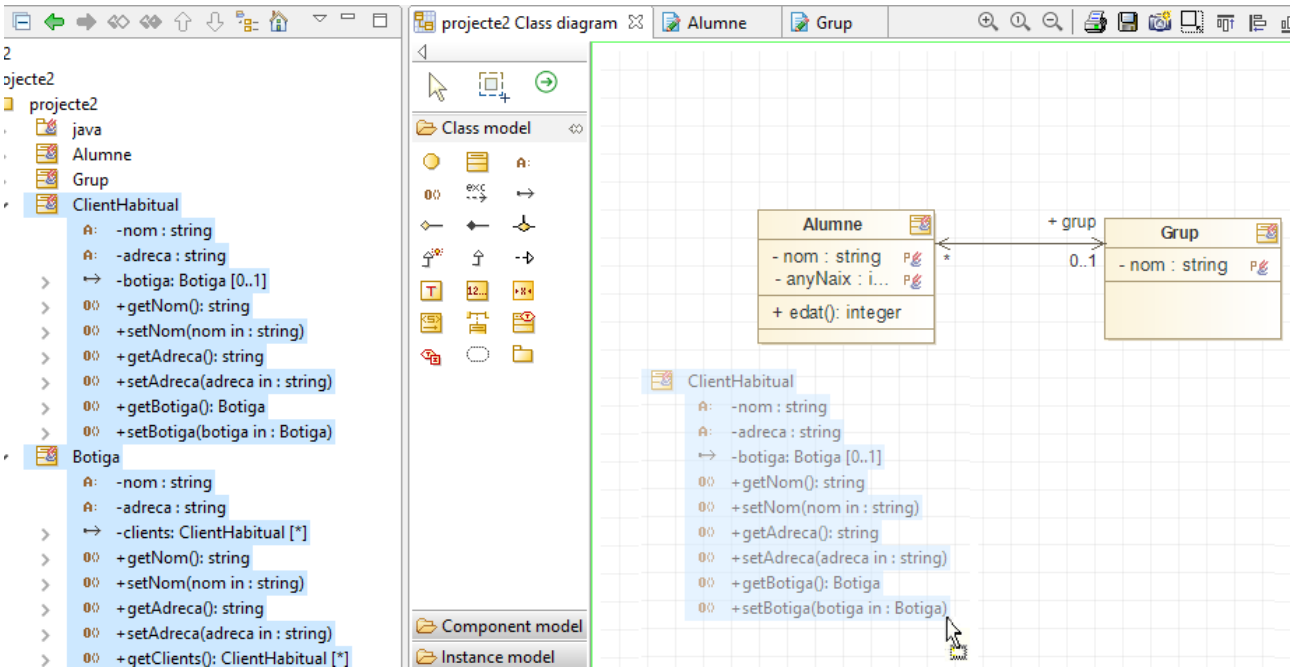
Per últim, clicant al botó *Reverse* ens sortirà una pantalla similar a la de la figura 1.57, que indica que tot ha anat bé:

FIGURA 1.57. Selecció dels fitxers de les classes a incorporar al diagrama



Es pot comprovar també que a la pestanya *Model* (a la part esquerra de la pantalla) es veuen les noves classes afegides al projecte, a més de les que ja tenia anteriorment. Ara només resta incorporar-les al diagrama de classes. Es pot fer desplegant-les, seleccionant-les totes (amb els seus membres inclosos) i arrossegant-les cap al diagrama. A la figura 1.58 es veu un resum del procés.

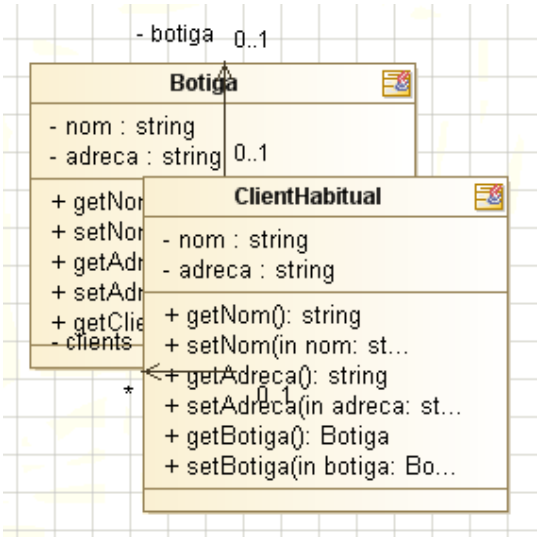
Figura 1.58. Incorporació de les classes al diagrama



Per seleccionar totes les classes que volem incorporar, n'hi ha prou amb fer clic a la primera, desplegar-les totes i, mantenint la tecla de majúscules premuda, fer clic al darrer element que volem incorporar.

Les noves classes presenten un aspecte similar al que es mostra a la figura 1.59.

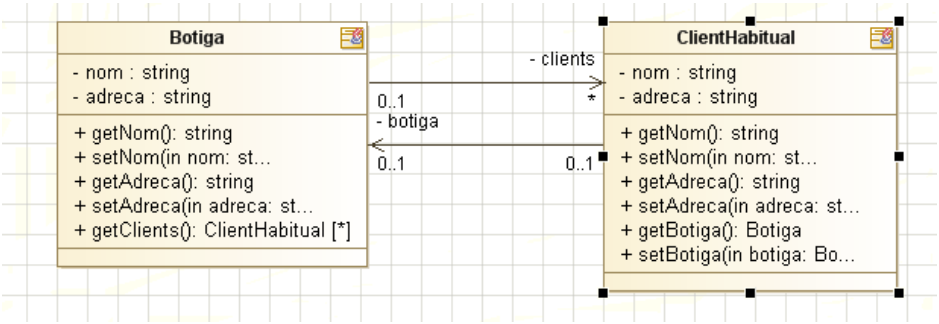
Figura 1.59. Classes incorporades al diagrama



Com es pot veure, les classes surten amuntegades. Podem moure-les i obtenir un resultat similar al mostrat a la figura 1.60.

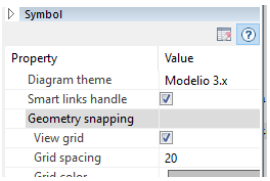
En aquest mateix apartat podeu trobar com fer les operacions bàsiques (moure, redimensionar...) amb els elements dels diagrames.

FIGURA 1.60. Resultat d'endreçar les classes.



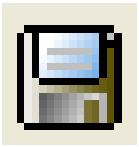
1.3.11 Operacions d'edició bàsiques

El diagrama de classes és el principal dels diagrames estàtiques. Modelio suporta també altres diagrames tant estàtics com dinàmics. Aquests poden tenir propòsits bastant diferents. No obstant, hi ha tota una sèrie d'operacions d'edició que són comunes a tots ells:



Propietats de la columna Symbol

- En primer lloc, a tots els diagrames és recomanable deshabilitar la característica *Smart links handle*, ja que no la utilitzarem i, a més, ens pot dificultar la resta de les accions. Es fa de la següent manera:
  - Fer clic a una zona del diagrama on no hi hagi cap element.
  - Fer clic a la columna *Symbol*, que es troba a la part dreta de la finestra, perquè es desplegui.
  - Deshabilitar la propietat *Smart links handle* i, si volem, tornar a fer clic a la barra superior de la columna *Symbol* perquè es torni a plegar.
- **Seleccionar un element**: n'hi ha prou amb fer-hi clic a sobre o, si volem seleccionar-ne més d'un a la vegada, podem:
  - Clicar a sobre d'ells, un rera l'altre, i mantenir a la vegada la tecla *Ctrl* polsada.
  - Assenyalar amb el ratolí una àrea que els inclogui, tot arrossegant el ratolí (de manera similar a com es seleccionen les icones d'una àrea a l'escriptori).
- **Esborrar un element**: cal seleccionar-lo i, a continuació, fer clic a la tecla *Supr*.


Icona Save diagrama in a file

- **Gravar el diagrama com un gràfic**: cal fer clic a la icona *Save diagram in a file*, que es troba a la barra d'eines de sobre del diagrama (al costat de la icona que representa una impressora i que permet imprimir el diagrama), i respondre al diàleg del sistema per desar fitxers.

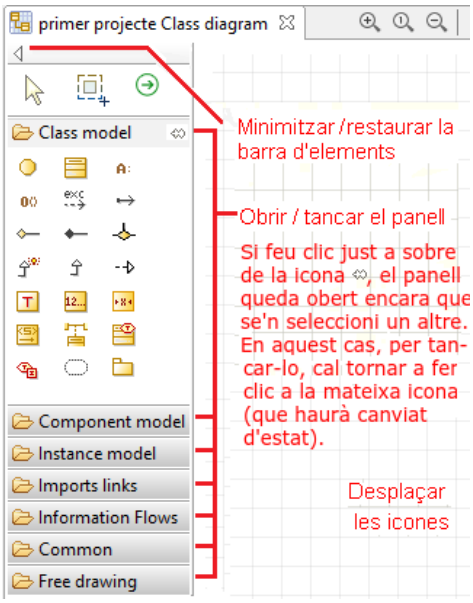

Icona Copy diagrama as grafic

- **Copiar el diagrama al porta-retalls com un gràfic**: cal fer clic a la icona *Copy diagrama as graphic*, que es troba a la barra d'eines de sobre del

diagrama i la imatge gràfica es copia automàticament al porta-retalls; a continuació pot enganxar-se en qualsevol programa d'edició.

- **Moure un element**: cal posar el ratolí a sobre de l'element i arrossegar-lo fins a portar-lo a la posició desitjada.
- **Canviar la forma d'un element**: cal seleccionar l'element, posar el ratolí a sobre d'un dels quadres negres que apareixen després d'haver fet la selecció i arrosegar-lo fins aconseguir la forma desitjada.
- **Connectar dos elements amb un enllaç**: normalment només cal seleccionar la icona corresponent a l'enllaç a la barra d'elements i, després, fer clic als elements que volem unir en l'ordre adient; quan s'està fent aquest segon pas, l'element sobre el que tenim el ratolí apareix de color verd si podem fer-hi clic i de color vermell en cas contrari.
- **Gestió de la barra d'elements**: podeu veure les operacions bàsiques a la figura 1.61. A tots els diagrames el funcionament és el mateix.

Figura 1.61. Elements per a gestionar la barra dels elements d'un diagrama



2. Diagrames dinàmics

El llenguatge unificat de modelització serveix per a la creació, especificació, construcció i documentació de models que representen tot tipus de sistemes. Aquesta modelització ha d'aportar una representació completa del sistema i es fa en termes d'orientació a objectes. Alguns dels sistemes que es poden representar mitjançant UML poden ser sistemes d'informació, sistemes de negocis, sistemes transaccionals (de gestió d'informació), sistemes distribuïts, sistemes estratègics, sistemes en temps real...

Alguns dels aspectes positius de fer servir la modelització de sistemes orientada a objectes són:

- Permet representar de forma més fidedigna el món real.
- Aquests són més fàcils d'entendre i de fer-hi ampliacions o modificacions.
- Tots els implicats parlen el mateix idioma (analistes, dissenyadors, desenvolupadors, verificadors...)
- Aporta més estabilitat pel fet que es poden fer petites ampliacions o modificacions sense haver de canviar tota la modelització.
- Permet la possibilitat de reutilitzar codi de forma senzilla.

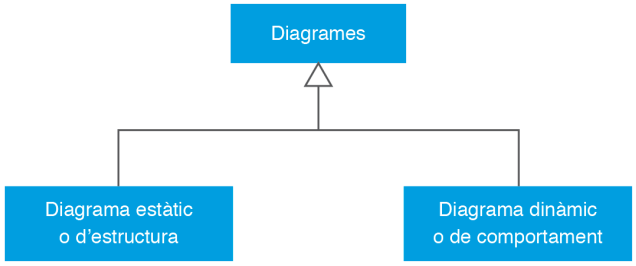
El llenguatge UML, en la seva versió 2.0, està compost de 13 diagrames (14 diagrames a partir de la versió 2.2), que es poden classificar seguint diversos criteris. Un d'ells pot ser en funció de les perspectives concretes o vistes des d'on es representen els sistemes. Per exemple, el diagrama de casos d'ús, que es considera un diagrama de la vista de requeriments o la vista de processos, engloba els diagrames de comportament.

Com es pot observar a la figura 2.1, una altra forma de classificació, que es fa servir en aquests materials, és la d'agrupar els diagrames en:

- Diagrames estàtics o diagrames d'estructura.
- Diagrames dinàmics o diagrames de comportament.

En la versió UML 2.2, i posteriors, es poden trobar set diagrames que pertanyen a cada una de les agrupacions de diagrames.

FIGURA 2.1. Classificació dels diagrames UML



2.1 Diagrames de comportament

Els diagrames considerats dinàmics o de comportament són els que representen el comportament dinàmic del sistema que s'està modelant. És a dir, indiquen les accions i processos que es duran a terme entre els elements del sistema, fixant-se en els seus moviments i en els efectes que tenen aquestes accions i activitats sobre els elements.

Els diferents diagrames de comportament descriuen el comportament de classificadors o, més sovint, de les seves instàncies, des de diferents punts de vista:

- Per una banda, representen o bé components executants o bé comportaments emergents.
- Per una altra, es destaca un d'aquests aspectes o un altre: interaccions amb l'exterior, o les situacions –anomenades estats– per les quals passa una instància durant la seva existència, o la seqüència temporal de les diferents parts d'un comportament...

Alguns d'aquests diagrames dinàmics es poden tornar a agrupar en funció de les seves funcionalitats. Concretament, dels set diagrames dinàmics n'hi ha tres que no s'agrupen, que tenen una entitat pròpia, i quatre més que es troben agrupats.

Els tres diagrames que no estaran agrupats són:

En l'apartat *Diagrama de casos d'ús* d'aquesta unitat es tracta amb detall el diagrama de casos d'ús.

- **Diagrama de casos d'ús**. Aquest diagrama identifica els diferents comportaments d'un sistema des del punt de vista de les seves interaccions amb el món exterior i descriu determinades relacions entre aquests comportaments.
- **Diagrama d'activitats** és el que descompon un comportament en activitats i representa els fluxos d'execució i d'informació entre aquestes activitats.
- **Diagrama d'estats** (en anglès State Machine Diagram). Aquest diagrama mostra els possibles canvis d'una situació a una altra de les instàncies del classificador de context i indica les causes i els comportaments que engeguen aquests canvis.

Els altres quatre diagrames estan agrupats en els anomenats diagrames d'interacció. Aquesta agrupació conté diagrames que representen un comportament emergent per mitjà de missatges entre les instàncies de classificadors d'una estructura interna o col·laboració i pot especificar restriccions temporals relatives a aquests missatges.

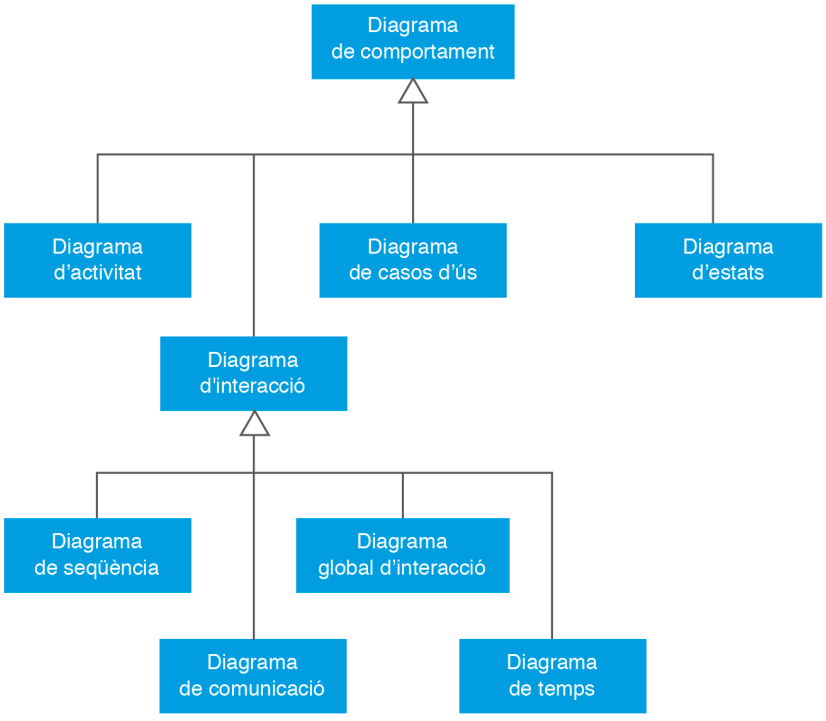
A l'apartat *2.3 Diagrames d'interacció* es tractaran amb detall els diagrames de seqüència i de col·laboració.

Els diagrames d'interacció són:

- **Diagrama de comunicacions**, que representa els missatges damunt els connectors d'una estructura interna o col·laboració
- **Diagrama de seqüència**, que posa èmfasi en l'ordre temporal dels missatges.
- **Diagrama de temps**, que representa una possible seqüència temporal de canvis d'estat d'una instància o de diverses instàncies que interactuen d'acord amb els diagrames d'estats respectius.
- **Diagrama general d'interacció**, que és un diagrama resumit que combina notacions dels diagrames de seqüències i dels d'activitats.

A la figura 2.2 es poden observar els diagrames de comportament pertanyents a l'UML Dinàmic.

Figura 2.2. Diagrama de comportament



2.1.1 Conceptes

Igual que en els diagrames d'estructura, cadascun dels diagrames de comportament de l'UML fa servir alguns tipus d'elements propis, a més d'altres de compartits per diversos diagrames de comportament. Els tipus d'elements compartits principals són:

- Senyal, que és una instància que es transmet d'un objecte a un altre.
- Missatge, que és una comunicació entre instàncies.
- Esdeveniment, que és un succés que pot tenir efectes damunt algun comportament.
- Activitat i acció.

Un **senyal** és una instància que un objecte o1 envia a un altre o2 i, com a conseqüència d'aquest enviament, s'executa un comportament asíncron que té o2 com a objecte de context.

Una **recepció de senyal** és una operació d'estereotip *signal* que indica que les instàncies del classificador dins el qual és definida (generalment, una classe o una interfície) reaccionen al senyal executant un comportament que és asíncron i, per tant, una recepció de senyal no pot tornar cap valor.

Un **missatge** és una comunicació entre instàncies de classificador, per la qual una (l'emissor) envia un senyal a l'altra (el destinatari) o li demana l'execució d'una operació.

Un **missatge síncron** és aquell que, quan s'emet, l'execució del comportament que l'ha emès roman aturada fins que rep un **missatge de resposta** del receptor; en canvi, quan s'emet un **missatge asíncron**, l'operació que l'ha emès es continua executant i no hi ha missatge de resposta. Un missatge asíncron pot consistir en l'enviament d'un senyal o la crida d'una operació; un de síncron només pot consistir en una crida d'una operació.

Un **esdeveniment** és un succés que es pot produir dins el sistema o el seu entorn, i quan té lloc pot provocar l'execució d'un comportament.

El fet que es produeixi un cert esdeveniment en un instant concret es diu **ocurrència d'esdeveniment**, i un conjunt de possibles ocurrències d'esdeveniment que tindrien el mateix significat i els mateixos efectes és un **tipus d'esdeveniment**. Un **disparador** és un element que especifica que una ocurrència d'esdeveniment del tipus indicat pot engegar un cert comportament. I inversament, tant l'engegada com la fi d'un comportament generen esdeveniments que poden engegar altres comportaments per mitjà dels disparadors corresponents.

Una **activitat** és una forma del comportament que es caracteritza per ser jeràrquica, en el sentit que pot ser constituïda per altres activitats; una activitat que no es pot descompondre és una **acció**.

2.1.2 Diagrama d'activitats

El **diagrama d'activitats** descriu les activitats que s'han de dur a terme en un cas d'ús, així com la manera de relacionar-se les activitats entre si per tal d'aconseguir un determinat objectiu. En altres paraules, el diagrama d'activitats descriu com un sistema implementa la seva funcionalitat.

Els diagrames d'activitats hereten conceptes dels diagrames de flux, essent un diagrama bastant fàcil d'interpretar.

Aquest diagrama està estretament vinculat a altres diagrames, com són el diagrama de classes (diagrama estàtic), el diagrama d'estats (diagrama dinàmic) i el diagrama de casos d'ús (diagrama dinàmic). Un diagrama d'activitats explica què està succeint entre diversos objectes o quin és el comportament d'un objecte.

Els diagrames d'activitats fan servir una sèrie d'elements, com ara:

- **Estats inicials**, representats mitjançant un cercle omplert de color negre. Marquen l'inici de l'execució dels processos o activitats.
- **Estats finals**, representats mitjançant un cercle omplert de color negre amb una altra circumferència per sobre amb una petita distància sense omplir. Els estats finals indiquen el final de l'execució d'un procés o activitat.
- **Activitats o accions**, representades mitjançant un rectangle de cantonades arrodonides. Indiquen l'arribada a un node una vegada efectuada una acció o activitat.
- **Transicions o fluxos de control**, representats mitjançant fletxes. La seva direcció indica el node des del qual s'inicia l'activitat o l'acció fins a l'altre node, al qual s'arribarà una vegada finalitzada.

Diagrames previs

Els classificadors que s'esmentin en un diagrama d'activitats s'han d'haver descrit prèviament en un diagrama de classes, i si les instàncies d'algun d'ells tenen diferents estats possibles cal haver-ne fet el diagrama d'estats corresponent.

A la figura 2.3 es poden observar totes les representacions gràfiques dels elements que participen en els diagrames d'activitats.

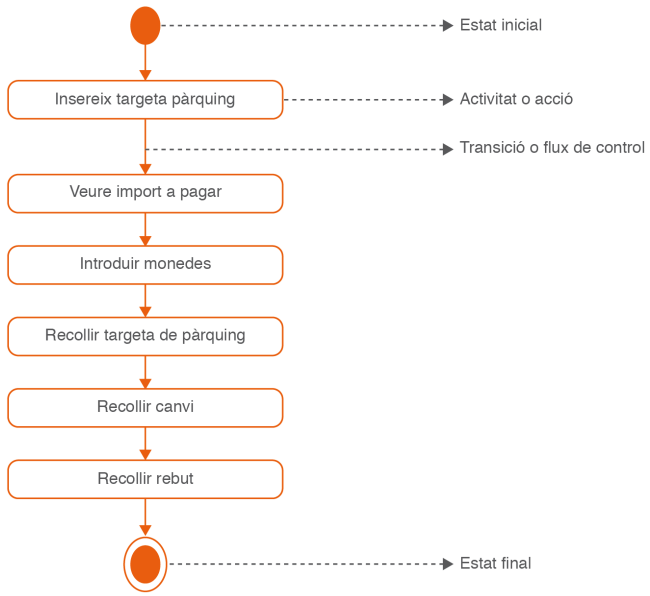
Figura 2.3. Elements del diagrama d'activitats

A la figura 2.4 es mostra un exemple de diagrames d'activitats. Es vol modelitzar el procediment de pagament per part d'un usuari de l'estada del seu cotxe a un pàrquing. Abans d'enretirar el cotxe del pàrquing, haurà d'abonar l'import que se li calcularà automàticament, en funció del temps que hagi estat estacionat el seu cotxe. Es mostra un diagrama d'activitats molt senzill en la figura 2.4, on totes les activitats es mostren a partir de transicions o fluxos de control seqüencials. A partir de l'estat inicial, i fins arribar a l'estat final, es passarà per sis activitats o accions:

- Inserir targeta pàrquing.
- Veure import a pagar.
- Introduir monedes.
- Recollir targeta pàrquing.
- Recollir canvi.
- Recollir rebut.

Es pot veure com les activitats defineixen, de forma molt clara, quina funcionalitat executaran.

Figura 2.4. Exemple de diagrama d'estats - activitats seqüencials



Altres elements importants en els diagrames d'activitats són els que fan referència a la possibilitat de donar alternatives als fluxos de control i a les activitats o accions. Alguns d'aquests elements són:

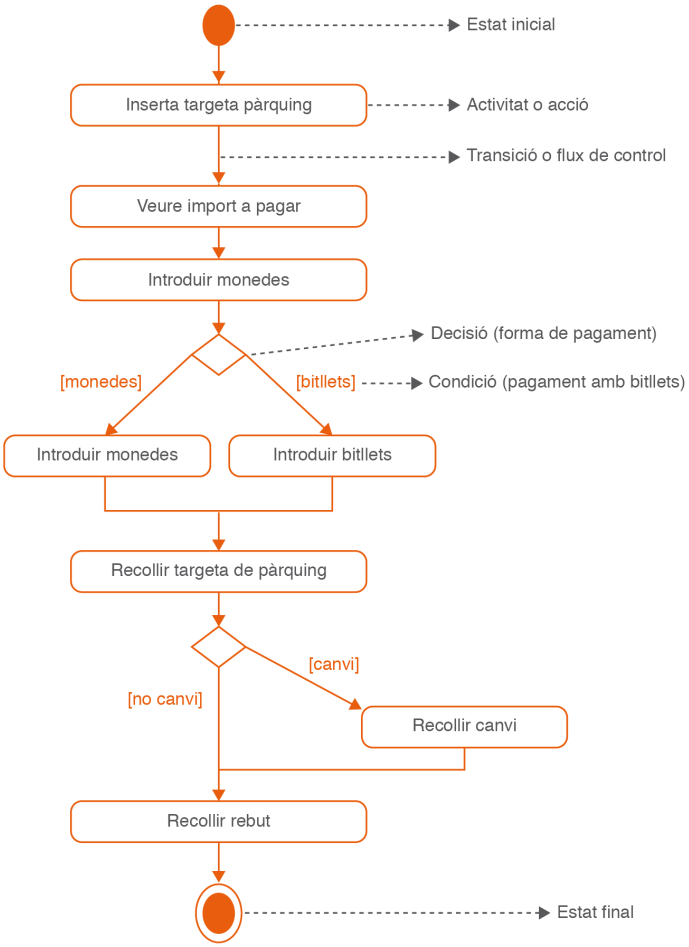
- **Decisió/fusió**, representat mitjançant un rombe regular. A partir d'un flux d'entrada, hi haurà dos o més fluxos de sortida, en funció de la condició marcada. Cada flux de sortida haurà d'estar indicat per una condició. En

el cas de la fusió, serveix per agrupar dos o més fluxos que arriben a un mateix punt de diverses decisions donades anteriorment a aquest punt en el diagrama d'activitats. La fusió tindrà dos o més fluxos d'entrada, però un únic flux de sortida.

- **Condició**, representat per un text que s'escriu entre els símbols `[ ... ]`. Aquesta condició representa la pregunta o preguntes que es farà el flux de control per, una vegada presa la decisió, saber per quin camí caldrà que continuï.

A la figura 2.5 es pot observar un exemple de condicions i bifurcacions a partir d'una ampliació de l'exemple anterior, el mostrat a la figura 2.4.

Figura 2.5. Exemple de diagrama d'activitats - alternativa



Què succeeix si l'usuari vol pagar la tarifa del seu pàrquing amb bitllets en comptes de fer-ho amb monedes? La modelització del sistema haurà de tenir en compte aquestes dues opcions. Què succeeix si l'usuari ha introduït l'import exacte a pagar i no ha de recollir canvi? Potser no és necessari passar per l'activitat Recollir canvi.

En aquests dos exemples es poden veure els elements de condició i de decisió/fusió.

Uns altres elements també importants en els diagrames d'activitats són els anomenats *fork* i *join*. *Fork* significa 'bifurcació' i *join* 'agrupació o unió'. Són dos elements estretament vinculats amb les bifurcacions, podent ser complementaris o substitutius d'aquestes. Les seves característiques són les següents:

- ***Fork***, representat per una línia negra sòlida, perpendicular a les línies de transició. Un *fork* representa una necessitat de ramificar una transició en més d'una possibilitat. La diferència amb una ramificació és que en el *fork* és obligatori passar per les diferents transicions, mentre que en la decisió es pot passar per una transició o per una altra. En altres paraules, les transicions de sortida d'un *fork* representen transicions que podran ser executades de forma concurrent.

- ***Join***, representat per una línia negra sòlida, perpendicular a les línies de transició. Un *join* fusiona dues o més transicions provinents d'un *fork*, és a dir, se sincronitzen en una única línia de flux. Totes les accions de les línies de flux prèvies al *join* han de completar-se abans que s'executi la primera acció de la línia posterior al *join*.

A la figura 2.6 es pot veure una altra evolució de l'exemple del pagament del pàrquing, afegint-hi aquesta vegada els elements *join* i *fork*. Concretament, si es parteix de la figura 2.5, es pot observar que a l'activitat *Recollir targeta del pàrquing* s'ha afegit un element de tipus *fork* que especifica que a partir d'aquella activitat s'haurà de passar per dues activitats, la de recollir el canvi, en el cas de ser necessari, i la de recollir el rebut. L'ordre en què es desenvoluparan serà indiferent.

Una vegada acabades aquestes dues activitats, s'ha afegit un element de tipus *join*, que recull la finalització de les dues activitats per, una vegada acabades, passar a la següent activitat, en aquest cas l'estat final.

Cal tenir en compte alguns condicionants que s'hauran d'acomplir a fi de crear un diagrama d'activitats correcte:

- No és obligatori que hi hagi un estat final. Per exemple, un procés que es desenvolupi de forma contínua mai acabarà.

- Podrà haver-hi diversos estats finals.

- Les activitats es podran descompondre en altres activitats.

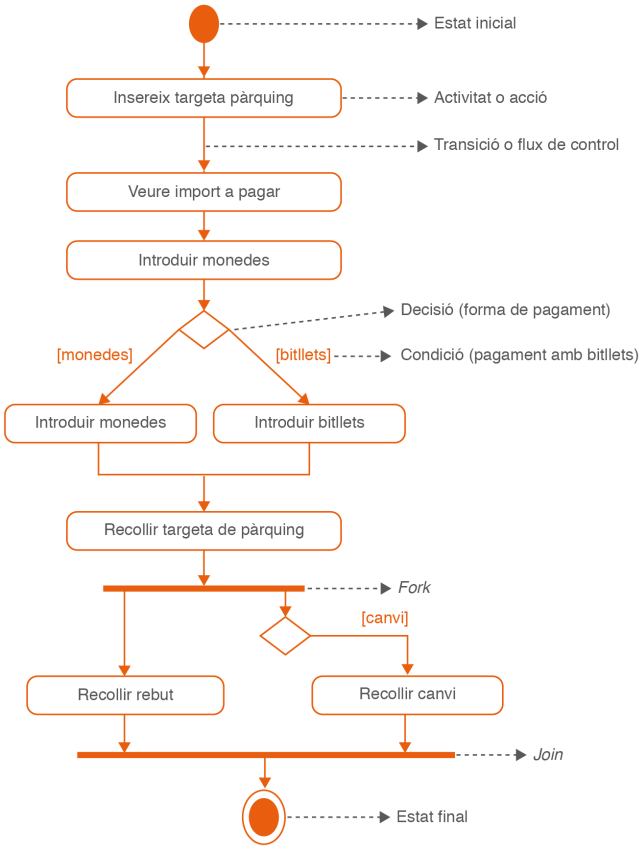
- Una acció és una activitat que no es pot descompondre en altres activitats.

- Un flux de control no podrà finalitzar mai en l'estat inicial.

- Cada activitat o acció ha de tenir, com a mínim, un flux de control d'entrada i un flux de control de sortida.

- Les condicions dels fluxos de sortida d'una mateixa decisió hauran de ser complertes i disjuntes.

FIGURA 2.6. Exemple diagrama de classes - 'fork' i 'join'



2.1.3 Diagrama d'estat

El **diagrama d'estat** representa el conjunt d'estats pels quals passa un objecte durant la seva vida en una aplicació en resposta a esdeveniments, juntament amb les seves respostes i accions. Un esdeveniment podria ser un missatge rebut, un error, un temps d'espera superior al planificat...

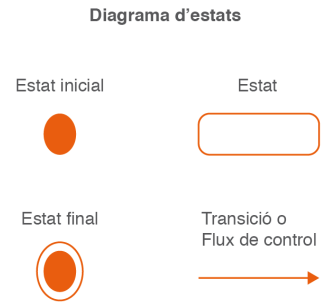
En la major part de les tècniques Orientades a Objectes, els diagrames d'estats es dibuixen per a una sola classe, mostrant el comportament d'un sol objecte durant tot el seu cicle de vida.

Com es pot veure a la figura 2.7, els elements d'un diagrama d'estats són molt similars al dels diagrama d'activitats. Resumint, podem trobar:

- **Estats inicials**, representats mitjançant un cercle omplert de color negre. Marquen l'inici del diagrama d'estats.
- **Estats finals**, representats mitjançant un cercle omplert de color negre amb una altra circumferència per sobre, amb una petita distància sense omplir. Els estats finals indiquen el final del diagrama d'estats. En un diagrama d'estats és possible que no hi hagi estats finals.

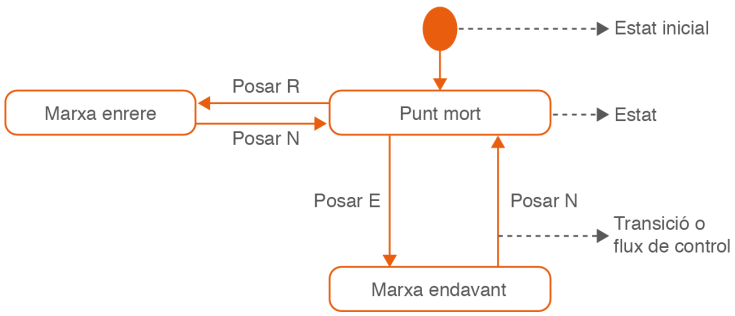
- **Estats**, representats mitjançant un rectangle amb les cantonades arrodonides. Un estat identifica una condició o una situació en la vida d'un objecte, durant la qual satisfà alguna condició, executa alguna activitat o espera que passi algun esdeveniment.
- **Transicions o fluxos de control**, representats amb una fletxa amb direcció que tindrà superposat el nom de l'esdeveniment que provocarà aquesta transacció entre estats. Tindrà sempre un inici i un final. Indica el pas d'un objecte d'un estat a un altre estat.

FIGURA 2.7. Elements del diagrama d'estats



A la figura 2.8 es mostra un petit exemple d'un diagrama d'estats. Es vol modelitzar, mitjançant un diagrama d'estats, el funcionament de les marxes d'un cotxe automàtic. Aquest tipus de cotxe podrà trobar-se en dos possibles estats: o està funcionant marxa endavant o està funcionat marxa enrere.

FIGURA 2.8. Diagrama d'estats - canvi de marxes d'un cotxe automàtic



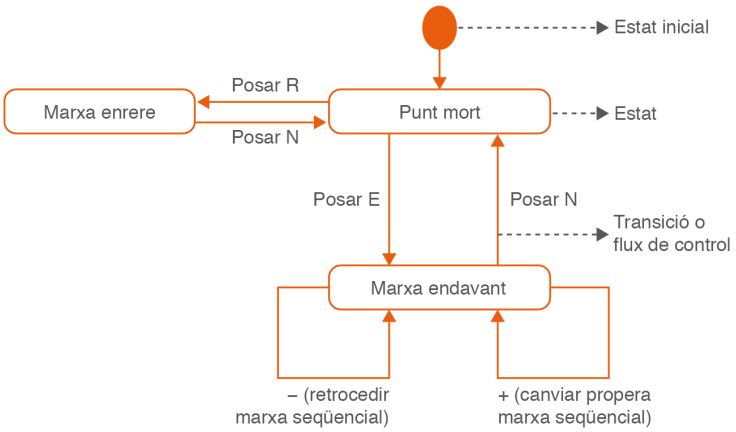
En l'exemple de la figura 2.8 es parteix d'un estat inicial i s'arriba a un estat anomenat *Punt Mort*. En aquest estat el cotxe no es mourà. A partir d'aquí es podrà posar la marxa enrere (mitjançant el flux de control anomenat *Posar R*) i es podrà tornar a l'estat *Punt Mort* amb el flux de control: *Posar N*. El mateix succeirà amb l'estat *Marxa endavant*.

Les transaccions es podran crear unint un mateix estat, com es pot veure a la figura 2.9.

En els cotxes automàtics el conductor pot deixar l'estat *Marxa endavant* activat.

En aquesta situació el cotxe va canviant automàticament de marxes sense l'acció del conductor. Però també s'accepta el poder augmentar o disminuir una marxa de forma seqüencial per part del conductor. En l'exemple es veu com l'estat *Marxa endavant*, a més de poder tornar a través de la transacció *Posar N* a l'estat *Punt Mort*, podrà evolucionar a partir de dues transaccions més (retrocedir marxa o canviar a la propera marxa) al mateix estat, amb la qual cosa modificarà les seves propietats.

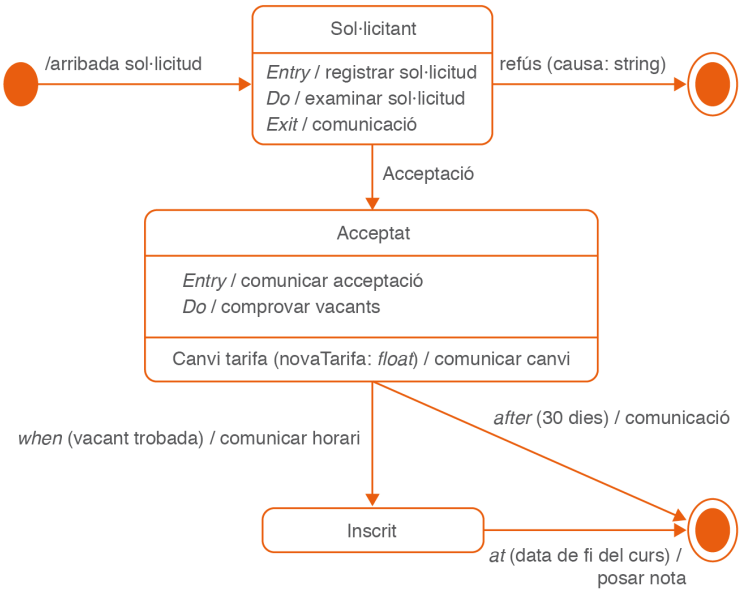
Figura 2.9. Diagrama d'estats - canvi de marxes d'un cotxe automàtic



Arribats a aquest punt, es proposarà un diagrama d'estats una mica més complicat, en el qual es mostra el comportament de l'estat davant un esdeveniment i les accions que efectua l'objecte.

La figura 2.10 és un diagrama d'estats que descriu la gestió de les inscripcions dels alumnes d'una acadèmia.

Figura 2.10. Diagrama d'estats - procés de matriculació



El diagrama comença amb l'estat inicial, representat amb la rodoneta; quan un alumne envia una sol·licitud d'inscripció, es produeix l'esdeveniment de senyal *arribada sol·licitud*, que provoca una transició cap a l'estat *Sol·licitant* i, com a conseqüència de l'arribada a aquest estat, s'executa l'activitat d'entrada *registrar sol·licitud* i, a continuació, l'activitat durant l'estat *examinar sol·licitud*, l'execució de la qual dura fins que es produeix algun dels esdeveniments que fan sortir d'aquest estat: si es produeix l'esdeveniment de senyal *refús*, que té com a atribut la seva *causa*, s'acaba tot el procés pel que fa a aquesta sol·licitud d'inscripció, mentre que si es produeix l'esdeveniment de senyal *acceptació*, l'estat de destinació és *Acceptat*; en tots dos casos, s'executa l'activitat de sortida *comunicació*.

Un cop a l'estat *Acceptat*, s'executa l'activitat d'entrada *comunicar acceptació* i, acte seguit, l'activitat a l'estat *comprovar vacants*, que s'interromp quan es produeix un dels dos esdeveniments següents:

- O bé l'esdeveniment de temps -que han passat 30 dies des de l'arribada a aquest estat-, i aleshores s'executa l'activitat *comunicació* i l'estat de destinació de la transició és l'estat final, que es representa per una rodona buida i una de plena concèntriques.
- O bé l'esdeveniment de canvi, que consisteix que passi a complir-se la condició *vacant trobada*.

En qualsevol cas, si abans de sortir de l'estat *Acceptat* es produeix l'esdeveniment de senyal *canvi tarifa*, que té com a atribut la *nova tarifa*, s'executa l'activitat *comunicar canvi*, d'acord amb la transició interna especificada.

Quan se surt de l'estat *Acceptat*, com a conseqüència d'aquest esdeveniment de canvi s'executa l'activitat *comunicar horari*.

Si un alumne és en l'estat *Inscrit* quan es produeix l'esdeveniment de temps d'arribada de la *data de la fi del curs*, es produeix una transició cap a l'estat final, alhora que s'executa l'activitat *posar nota*.

2.2 Diagrama de casos d'ús

Aquest diagrama és un dels més representatius i més utilitzats de l'anàlisi i disseny orientat a objectes. Mitjançant els diagrames de casos d'ús s'aconsegueix mostrar el comportament del sistema i com aquest es relacionarà amb el seu entorn.

El **diagrama de casos d'ús** identifica els comportaments executants d'un classificador generalment complex (per exemple, un programari) i especifica amb quins usuaris i altres entitats exteriors tenen interacció (en el sentit que en reben informació o els en donen o són engegats per aquestes entitats), i també certes relacions entre aquests comportaments.

El punt de vista que es fa servir per elaborar els diagrames de casos d'ús és el punt de vista de l'usuari final. Això implica que es tracta d'un diagrama molt convenient i utilitzat en els intercanvis inicials d'opinions amb els usuaris finals del sistema a modelitzar i, posteriorment, a automatitzar. Els mateixos usuaris el poden entendre i participar en la seva correcció. Amb aquest tipus de diagrama es podrà crear una documentació adequada per a les necessitats de la presa de requeriments.

Els diagrames de casos d'ús representen els diferents requeriments que fan servir els usuaris finals d'un determinat sistema. A partir de la utilització d'actors i de casos d'ús, aquest tipus de diagrames modelen les diferents funcionalitats que podrà oferir un sistema.

Un **cas d'ús** és una funcionalitat o un servei que ofereix el sistema a modelitzar als seus usuaris finals. És un conjunt d'interaccions seqüenciades que es desenvolupen entre els actors i el sistema per donar resposta a un esdeveniment que inicia un actor denominat principal.

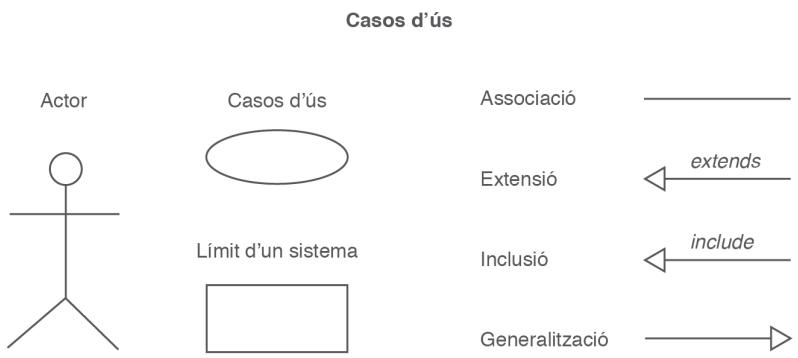
En la modelització d'un sistema podrà haver-hi molts casos d'ús. Això es deu al fet que cada cas d'ús haurà de donar resposta només a una característica concreta del sistema. Així, amb aquesta representació del sistema, l'usuari final podrà validar fàcilment que l'analista ha entès correctament el funcionament del sistema que ha de representar.

Els elements fonamentals del diagrama de casos d'ús són els següents:

- Escenari
- Actor
- Subjecte
- Cas d'ús o comportament
- Associació o relació

A la figura 2.11 es mostren les representacions gràfiques dels elements més importants d'un diagrama de casos d'ús.

FIGURA 2.11. Diagrama de casos d'ús



2.2.1 Escenari

Una entitat externa i els actors

A una sola entitat externa li poden correspondre diversos actors, si les interaccions que té amb el sistema de maquinari i programari són prou diverses per considerar que fa diferents grups de papers en relació amb ell.

Un escenari és un camí representat per un o més actors i un o més casos d'ús i les seves associacions. Es tracta d'un camí que podrà prendre un cas d'ús.

Un cas d'ús podrà tenir diversos escenaris possibles. Aquests escenaris podran representar casos d'èxit i casos on no s'acompleixen les expectatives.

2.2.2 Actor

Un actor és un conjunt de papers que fa una entitat física o virtual externa al subjecte en relació amb els seus casos d'ús; a partir d'aquests papers l'actor interactua amb el subjecte i cada paper té a veure amb un dels casos d'ús del subjecte. Tot allò que inicia un cas d'ús o respon a un cas d'ús també es considera un actor.

Els actors són classificadors instanciables. Un actor pot ser:

- Una persona (un usuari) que interactuï directament amb el subjecte (per tant, si un usuari interactua amb el subjecte per compte d'un altre, l'actor serà el primer usuari, no pas el segon).
- Un sistema informàtic extern en temps d'execució que rebi informació del subjecte o li'n doni.
- Un dispositiu físic que tingui un cert comportament propi i autònom en relació amb el subjecte i hi interactuï directament.
- *Temps*, *Rellotge*... quan un cas d'ús s'engega automàticament en una hora determinada.

Un actor també es defineix com un rol que farà un usuari en utilitzar un sistema. Amb el concepte de rol es vol determinar el fet que una mateixa persona física (un usuari final del sistema) podrà interpretar el paper d'actors diferents en diferents casos d'ús o, fins i tot, en un mateix cas d'ús. Un actor no serà una determinada persona, sinó que serà el paper que juga (la persona que sigui) quan utilitza el sistema.

2.2.3 Subjecte

El subjecte d'un diagrama de casos d'ús és el classificador al qual està associat el diagrama, que en representa els comportaments executants.

Pot ser qualsevol cosa que tingui comportaments: un programari, un sistema informàtic o físic en general, un subsistema, un component, una classe...

2.2.4 Cas d'ús

Un cas d'ús és un comportament executant del subjecte; és engegat d'una manera directa o indirecta per un actor i lliura uns resultats concrets a un actor o a diversos, o a un altre cas d'ús.

Un cas d'ús es representa mitjançant un verb que indica una acció, operació o tasca concreta. Aquesta operació s'activarà a partir d'una ordre donada per un agent extern al cas d'ús. Podrà ser un actor que faci una petició i generi el cas d'ús, o bé un altre cas d'ús que l'invoqui.

2.2.5 Una associació o una relació

Una associació o una relació és un vincle que es dóna entre un cas d'ús i un actor o entre dos casos d'ús.

Entre dos casos d'ús es podran establir diferents tipus de relacions:

- **Associació**: una associació és un camí de comunicació entre un actor i un cas d'ús. Aquesta comunicació implica que l'actor participa en el cas d'ús.
- **Generalització/especialització:** una generalització indica que un cas d'ús és una variant d'un altre, és a dir, podrà trobar-se en una forma especialitzada d'un altre cas d'ús existent. Tant la representació com el sentit poden semblar similars al concepte de subclasses en l'orientació a objecte. Aquest tipus de relació serveix per identificar comportaments semblants per part d'un o diversos casos d'ús. A partir d'una primera descripció més genèrica, l'especialització detalla els comportaments més específics a cada cas d'ús especialitzat. Això es deu al fet que el cas d'ús especialitzat pot variar qualsevol aspecte del cas d'ús base.

  La generalització és el mateix tipus de relació vist des del punt de vista de les subclasses que tenen elements comuns i que, a partir de les seves semblances, permeten crear una superclasse que contindrà aquests elements comuns.
- **Dependència d'inclusió**: el tipus de relació d'inclusió és un cas de dependència entre casos d'ús. Concretament, indica que un cas d'ús està inclòs en un altre. Això succeeix quan els casos d'ús comparteixen uns determinats elements. En aquest cas, el cas d'ús que està inclòs és el que tindrà tots els comportaments compartits. Aquest tipus de relació també es coneix com *use*, perquè hereta moltes de les característiques de l'antiga

relació *use* o utilitza. Es tracta d'una relació útil en casos en què es volen extraure comportaments comuns des de molts casos d'ús a una descripció individual. En aquest tipus de relació no hi ha paràmetres o valors de retorn. Quan un cas d'ús A és inclòs per un altre cas d'ús B, el cas d'ús que ha incorporat el comportament de l'altre, en aquest cas el cas d'ús B, haurà de ser utilitzat per si mateix. En cas contrari, es coneix com a cas d'ús abstracte.

- **Dependència d'extensió:** aquesta relació és un altre tipus de dependència. Ofereix un tipus d'extensió diferent a la relació de tipus generalització, d'una forma més controlada. Quan un cas d'ús s'estén a un altre, això significa que el primer pot incloure part del comportament de l'altre cas d'ús, aquell al qual s'està estenent. A més, podrà afegir-hi accions o comportaments. La forma de funcionar de la dependència serà la creació d'una sèrie de punts d'extensió per part del cas d'ús base. Si hi ha més d'un punt d'extensió, caldrà definir molt bé quin és el punt que s'ha estès. A partir d'aquests punts, el cas d'ús especialitzat només podrà modificar el comportament dels punts d'extensió que s'hagin creat. Aquest tipus de relació és una alternativa a l'ús de casos d'ús complexos. Amb les dependències d'extensió es podran controlar millor les diferents bifurcacions i els errors.

Un cas d'ús es podrà representar de dues formes diferents:

- Mitjançant un diagrama de casos d'UML, seguint la notació de la resta de diagrames UML.

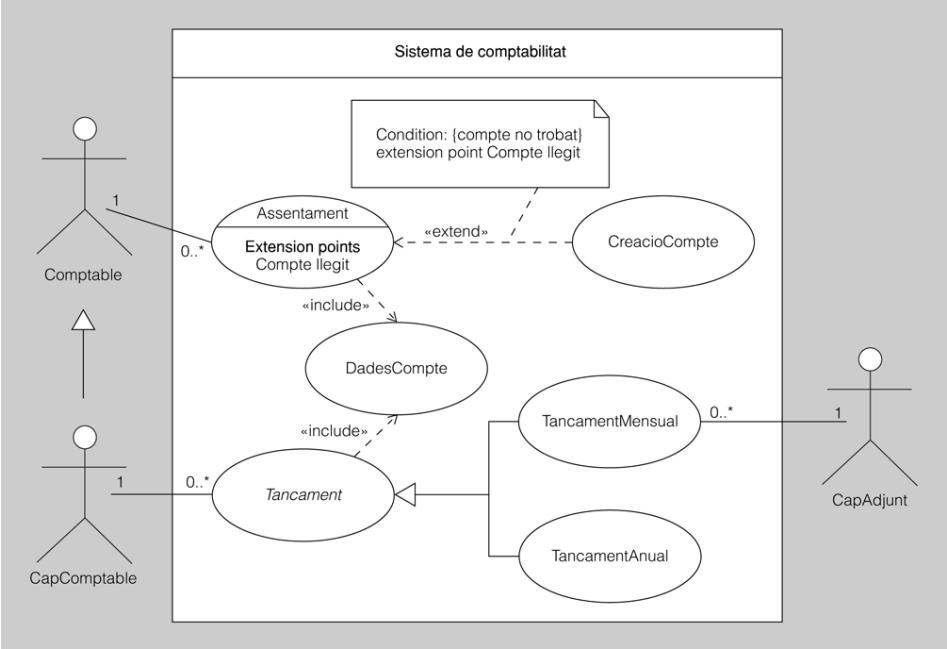
- Mitjançant un document detallat.

Diagrama de casos d'ús en UML

Cadascun dels elements que componen un diagrama de casos d'ús té una notació gràfica que permet la representació de cada cas d'ús de forma esquemàtica i fàcil d'entendre i d'interpretar per als usuaris finals.

A la figura 2.12 veiem el diagrama de casos d'ús d'una suposada aplicació de comptabilitat, representada pel subjecte anomenat *Sistema de comptabilitat*.

FIGURA 2.12. Cas d'ús - sistema de comptabilitat



L'actor *CapComptable* és una especialització de *Comptable*; *Comptable* és l'actor primari del cas d'ús *Assentament*, ja que no té cap més actor, i *CapComptable* és actor primari del cas d'ús *Tancament* i també, per herència entre actors, d'*Assentament*. El cas d'ús *CreacioCompte* estén el cas d'ús *Assentament* perquè quan s'intenta fer un assentament amb un compte inexistent cal crear aquest compte; s'ha indicat el punt d'extensió, *Compte llegit*, dins del cas d'ús estès, assentament, i la condició corresponent, *compte no trobat*. *CreacioCompte* no és executable independentment perquè no té cap actor ni, doncs, actor primari.

El cas d'ús *DadesCompte* consisteix en l'accés a les dades d'un compte, accés que és una part comuna als casos d'ús *Assentament* i *Tancament*, d'aquí la dependència include. *Tancament* és abstracte i *Tancament-Mensual* i *TancamentAnual* en són especialitzacions i, en conseqüència, n'hereten l'actor *CapComptable* i no cal associar-los-el explícitament. *TancamentMensual* té, a més de l'actor heretat, l'actor *CapAdjunt* (que pel diagrama no sé sap si també és primari o no). Les multiplicitats de les associacions són innecessàries per trivials: en cada execució del cas d'ús intervé una sola instància de l'actor i cada instància de l'actor pot participar en qualsevol nombre d'execucions del cas d'ús.

Document detallat de casos d'ús

La segona forma de representar els casos d'ús és presentar un document detallat. Aquest document podrà ser un substitutiu dels diagrames gràfics de casos d'ús, però també podrà ser un complement a aquells diagrames.

En el document detallat es mostrarà tot tipus d'informació referent al cas d'ús, aquesta vegada, però explicada amb text. Tant les comunicacions i les interaccions dels elements com els escenaris i actors involucrats queden descrits en aquest document detallat.

Es presenta en forma de taula completa amb dues columnes i tantes files com informacions es vulguin mostrar. Algunes d'aquestes informacions poden ser:

- **Cas d'ús**, que indicarà el nom del cas d'ús; serà com el títol del document detallat.
- **Actors**, que descriurà tots els actors, tant primaris com secundaris, que prendran part en el diagrama de casos d'ús.
- **Tipus**, que indica el tipus de cas d'ús que s'està detallant. Pot ser un tipus de flux bàsic o bé basat en algun altre cas d'ús (a partir d'una inclusió, una extensió o una generalització).
- **Propòsit**: caldrà indicar-hi l'objectiu, la raó de ser, del cas d'ús.
- **Resum del cas d'ús**: es durà a terme una descripció resumida del cas d'ús.
- **Seqüència normal o flux principal**: s'hi indicarà, seqüencialment, cada pas que es durà a terme juntament amb la seva acció corresponent i la seva descripció detallada. En funció de les accions dels actors, s'escollirà un subflux o un altre per a la continuació del cas d'ús.
- **Subfluxos**: són els considerats fluxos secundaris en les accions seqüencials d'un cas d'ús.
- **Precondicions**: igual que en programació, les precondicions estableixen els requisits que haurà de complir el cas d'ús per poder-se executar.
- **Excepcions**: són les situacions especials que hi pot haver durant l'execució d'un cas d'ús.
- **Urgència**: indicarà la celeritat amb què el cas d'ús s'haurà de desenvolupar i d'executar.
- **Freqüència**: indicarà el nombre de vegades que està prevista l'execució d'aquest cas d'ús.
- **Rendiment**: indicarà en quant de temps s'haurà d'executar el cas d'ús com les accions seqüencials.
- **Comentaris**: espai on es podran recollir altres consideracions referents al cas d'ús que es considerin oportunes.

A la figura 2.13 es pot veure com quedaria un document detallat d'un cas d'ús amb alguns dels camps citats anteriorment.

Figura 2.13. Document detallat d'un cas d'ús

Identificador	Codi identificatiu del cas d'ús		
Nom	Nom descriptiu		
Descripció	Breu descripció de l'objectiu del cas d'ús		
Actors	Identificació dels actors que intervenen en el cas d'ús		
Precondicions	Condicions que s'han de produir abans d'accedir al cas d'ús		
Seqüència normal	Pas	Acció	
	1	Descripció acció 1 (seqüencial)	
	2	Descripció acció 2 (seqüencial)	
	3	Descripció acció 3 (alternativa)	
		3a	Si condició acció 3a
		3b	Si condició acció 3b
	4	Descripció acció 4	
	...	...	
Seqüència alternativa o excepcions	Pas	Acció	
	e1	En el cas que es produeixi l'excepció1, s'haurà d'efectuar l'acció e2	
	e2	En el cas que es produeixi l'excepció2, s'haurà d'efectuar l'acció e3	
	...	...	
Postcondicions	Condicions que ha de complir una vegada executat el cas d'ús		
Notes	Comentaris		

Avantatges/inconvenients dels diagrames de casos d'ús

Els diagrames de casos d'ús ofereixen alguns punts forts que els converteixen en els més populars dels diagrames d'UML:

- Són molt adequats per comunicar-se amb els usuaris finals i, en definitiva, amb els clients.
- Ajuden a documentar les funcionalitats del que es coneix com a capses negres, que és com es consideren alguns dels casos d'ús d'un sistema.
- Ajuden a dividir i gestionar els projectes d'una mida molt extensa.
- Complementen la documentació del projecte informàtic, oferint una explicació senzilla del funcionament als usuaris.
- Permeten fer un seguiment objectiu del projecte.
- Ajuden en la tasca de verificació i validació que duen a terme els verificadors del programari desenvolupat en el projecte.

2.3 Diagrames d'interacció

Els anomenats diagrames d'interacció agrupen un conjunt dels diagrames classificats dintre els diagrames de comportament. De fet, els diagrames d'interacció es consideren un subtipus dels diagrames de comportament.

Cal recordar que els diagrames de comportament indicaran, dintre de la simulació o modelització del sistema representat, què haurà de succeir entre els elements definits als diagrames estàtics o d'estructura.

En els diagrames d'interacció es donarà més èmfasi a les relacions entre els elements dels sistema, concretament en el que té a veure amb el flux de dades i amb el flux de control.

Una **interacció** és un comportament emergent descrit en termes d'intercanvis de missatges entre els elements connectables d'una estructura composta.

Els diagrames d'interacció representen un comportament emergent per mitjà de missatges entre les instàncies de classificadors d'una estructura interna o col·laboració. Aquests diagrames poden especificar restriccions temporals relatives a aquests missatges.

Els diagrames d'interacció són:

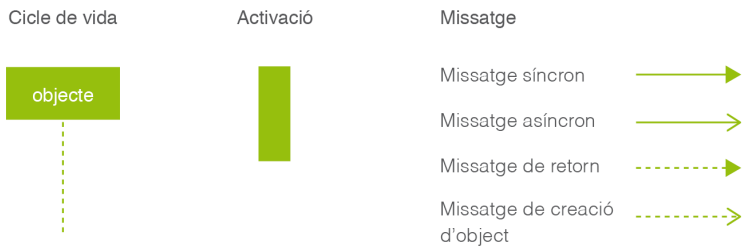
- **Els diagrames de seqüència**. Posen l'èmfasi en l'ordre temporal dels missatges entre els diferents elements de la modelització del sistema. Aquests missatges es coneixen com a missatges d'intercanvi entre objectes, el que realment són crides a mètodes de les classes. Aquests intercanvis es donen en un escenari concret en un temps delimitat.
- **Els diagrames de comunicació** (diagrames de col·laboració). Representen els missatges damunt els connectors d'una estructura interna o de col·laboració. Es tracta d'un tipus de diagrama molt similar als diagrames de seqüència pel que es refereix als missatges que s'intercanviaran els diferents objectes. La diferència rau en el fet que en els diagrames de seqüència l'important és l'ordre temporal d'aquests missatges, mentre que en els diagrames de comunicació es fa més èmfasi en la relació entre els objectes i el tipus de relació que hi ha.
- **Els diagrames de temps**. Representen els canvis d'estat que tindrà un objecte al llarg del temps. Aquests canvis d'estat dels objectes es produiran a partir d'esdeveniments que es produeixin al llarg del temps. Aquests diagrames estan estretament lligats als diagrames d'estats i als diagrames de seqüència.
- **Els diagrames de visió general de la interacció**. Defineixen les interaccions a partir d'una variant dels diagrames d'activitat. Ofereixen una visió general dels fluxos de control utilitzant interaccions en lloc d'activitats.

2.3.1 Diagrama de seqüència

El **diagrama de seqüència** descriu les interaccions entre un grup d'objectes mostrant de forma seqüencial les trameses de missatges entre objectes.

Els objectes interactuen entre si amb l'enviament de missatges. La recepció d'un missatge sol significar l'execució d'un mètode que s'especifica en el missatge, i es torna actiu l'objecte mentre dura l'execució del mètode.

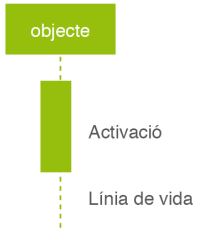
Figura 2.14. Elements del diagrama d'activitats



Els elements són els següents:

- Una **línia de vida** representa un element connectable que participa en una interacció enviant i rebent missatges (figura 2.15). Una línia de vida representa l'interval de temps en què existeix la instància o instàncies que formen part de l'element connectable, des de la seva creació fins a la seva destrucció, tot i que en general només durant una part o diverses parts d'aquest interval participen en la interacció que conté la línia de vida; aquestes parts s'anomenen **activacions** i representen els intervals de temps durant els quals s'està executant alguna operació que té com a objecte de context alguna de les seves instàncies.

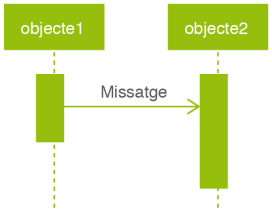
Figura 2.15. Línia de vida



- Un **missatge** és l'especificació de la comunicació entre objectes (figura 2.16). Un missatge es representa amb una fletxa de l'emissor cap al receptor; diferents tipus de missatge es representen amb diferents tipus de fletxa:
  - Un **missatge asíncron** s'indica amb una fletxa de línia contínua i punta oberta; és quan l'objecte no espera la resposta a aquest missatge abans de continuar.
  - Un **missatge síncron** es representa amb una fletxa de línia contínua i punta plena; és quan l'objecte espera la resposta a aquest missatge abans de continuar amb el seu treball.
  - Un **missatge de resposta d'un missatge síncron** s'indica amb una fletxa de línia discontínua i punta plena.

– Un **missatge que provoca la creació d'un objecte** s'indica amb una fletxa de línia discontínua i punta oberta; a més, l'extrem del missatge coincideix amb el començament de la línia de vida del receptor.

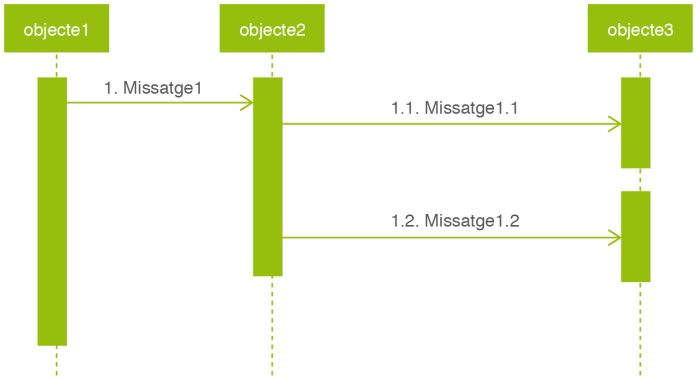
Figura 2.16. Missatge



Els missatges es poden numerar en format de llista multinivell. Si un missatge s'envia una vegada finalitzat el seu precedent es comptabilitza seqüencialment: 1,2,3... Però si el missatge s'envia abans que hagi finalitzat el precedent, s'incrementarà el nivell: 3.1, 3.2,...

A la figura 2.17 se'n pot veure un exemple representatiu.

Figura 2.17. Numeració dels missatges



A continuació, es mostra un exemple complet per al desenvolupament d'un diagrama de seqüència. Concretament, es proposa crear un diagrama de seqüència per a la modelització de l'elaboració d'una *vichyssoise*, sopa freda de puré de patata i porros. Aquesta recepta es durà a terme amb l'ajuda d'un robot de cuina.

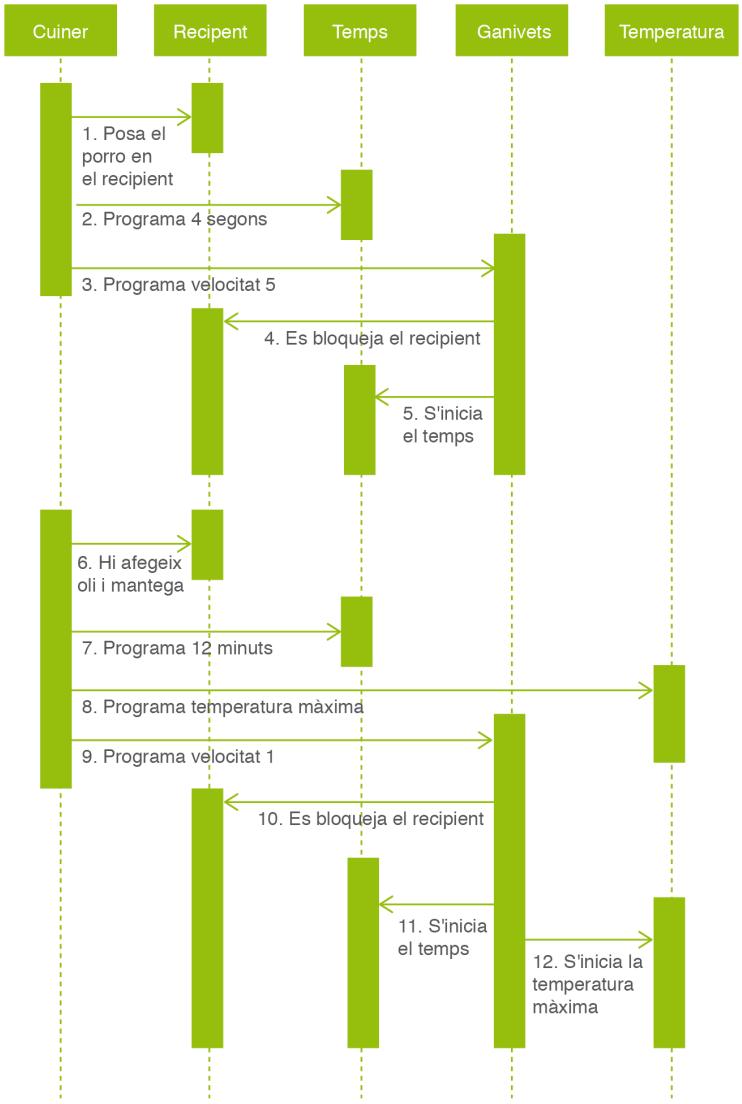
Inicialment, es mostren els passos que s'han de fer per a l'elaboració de la sopa:

El cuiner posa el porro en el recipient i programa 4 segons a velocitat 5. En especificar la velocitat dels ganivets, es bloqueja la tapa del recipient i el cronòmetre del robot comença a comptabilitzar el temps.

Un cop transcorregut el temps, es desbloqueja la tapa, i el cuiner hi pot col·locar l'oli i la mantega. Aquest programa velocitat 1 i 12 minuts a temperatura màxima. En especificar la velocitat dels ganivets, es bloqueja la tapa del recipient, el cronòmetre del robot comença a comptabilitzar el temps i s'activa la resistència

per proporcionar calor.

Figura 2.18. Exemple de diagrama de seqüència - robot de cuina



La recepta continuaria, però es demana el diagrama de seqüència fins a aquest punt.

A la figura 2.18 es pot veure un exemple de diagrama de seqüència que podria solucionar l'enunciat anterior.

2.3.2 Diagrama de comunicació

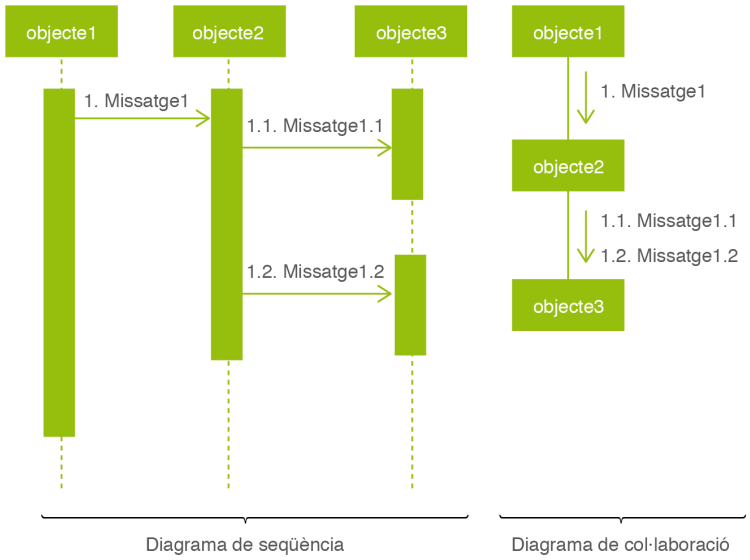
Aquest diagrama, en versions anteriors a la versió 2.0 de l'UML, es coneixia com a diagrama de col·laboració. Des de les darreres versions es coneix amb aquest nou nom: diagrama de comunicació.

La finalitat del **diagrama de comunicació** és representar la comunicació entre els elements del sistema que es vol modelitzar. Treballant en conjunt els elements podran acomplir els objectius del sistema.

El diagrama de comunicacions té l'aspecte general d'una col·laboració en la qual, al costat de cada connector, s'han afegit les fletxes corresponents als missatges que hi circulen durant la interacció. Aquestes fletxes són les mateixes que hi hauria en un diagrama de seqüències de la mateixa interacció pel que fa al tipus de línia i de punta, però són molt més curtes. La disposició vertical dels elements connectables és lliure i, per tant, no representa cap ordenació temporal; per mitjà de la numeració s'indica l'ordre d'emissió dels missatges i si aquesta emissió és seqüencial o en paral·lel.

A la figura 2.19 es mostra un diagrama de comunicació (antigament anomenat diagrama de col·laboració). També es mostra el diagrama de seqüència que hi està vinculat, per poder veure, d'aquesta manera, la seva integració.

FIGURA 2.19. Diagrama de seqüència vs. diagrama de col·laboració



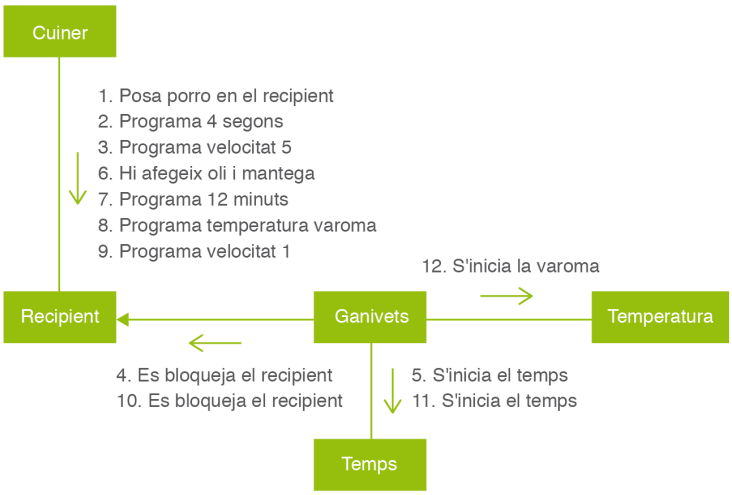
El diagrama de comunicacions, d'una banda, no permet representar els usos d'interacció i, de l'altra, només té representació per a alguns dels operadors d'interacció; en conseqüència, a la pràctica només és adequat per a la descripció d'interaccions relativament senzilles.

Fins a cert punt, un diagrama de comunicacions és semblant a un diagrama de seqüències vist des de dalt: de les línies de vida només se'n veuria la capçalera i, com que si hi hagués diversos missatges entre els mateixos dos elements connectables es veurien sobreposats. També hi ha autors que consideren que és un diagrama molt semblant al diagrama d'objectes, perquè mostra el context necessari per a una col·laboració entre objectes.

A la figura 2.20 es mostra el diagrama de comunicacions corresponent a l'exemple que s'ha exposat anteriorment: l'elaboració d'una sopa freda Vichyssoise. Es

recomana comparar el diagrama de seqüències anterior corresponent a aquest exemple, figura 2.18, amb el diagrama de comunicacions de la figura 2.20.

FIGURA 2.20. Diagrama de comunicacions - robot de cuina



2.3.3 Diagrama de temps

Els **diagrames de temps** (o diagrames temporals) representen una interacció entre diferents estats i en destaquen els aspectes temporals i, en especial, els canvis d'estat o de valor d'una línia de vida (és a dir, de les instàncies que conté) o de diverses línies al llarg del temps. Els diagrames de temps mostren els canvis de l'estat d'un objecte al llarg del temps com a conseqüència d'esdeveniments.

Generalment, en una màquina d'estats hi ha més d'un camí possible (és a dir, més d'una seqüència de transicions) entre l'estat inicial i l'estat final; un diagrama temporal correspon només a un d'aquests camins i, en conseqüència, en un diagrama temporal no hi pot haver parts opcionals o alternatives.

El diagrama de temps no té conceptes propis i les seves notacions pròpies són molt senzilles. En la seva forma més usual i detallada és un diagrama bidimensional, en el qual es representa una escala horitzontal de temps i els diferents estats o valors d'una línia de vida corresponent a diferents altures dins de la dimensió vertical; l'ordenació vertical dels estats no té cap significat. Un diagrama pot tenir diverses franges horitzontals que corresponen a diferents línies de vida que comparteixen una sola escala de temps.

La successió dels diferents estats d'una línia de vida al llarg del temps és una línia contínua feta de segments de recta, dels quals:

- Els segments horitzontals representen intervals de temps durant els quals la línia de vida roman en un cert estat.

- La resta de segments representa les transicions entre aquests estats.

Un **segment vertical** representa una transició que dura un temps negligible, mentre que un segment inclinat representa una transició que dura el temps indicat per la diferència de les abscisses dels seus extrems inicial i final.

Diagrames temporal i de seqüències

Atès que tots dos diagrames tenen una escala de temps i línies de vida, podríem dir que el diagrama temporal ve a ser un diagrama de seqüències sense fragments combinats en el qual damunt cada línia de vida es representen els canvis d'estat.

Tot seguit es mostra un exemple de diagrama de temps. En primer lloc, es mostren, a la figures figura 2.21 i figura 2.22, els diagrames d'estats de dues classes, la `Classe1` i la `Classe2`. La `Classe1` ofereix dos estats (`Estat A` i `Estat B`), mentre que la `Classe2` ofereix tres possibles estats (`Estat 1`, `Estat 2` i `Estat 3`).

Figura 2.21. Diagrama d'estats de "Classe1"

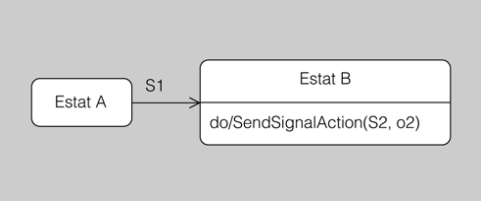
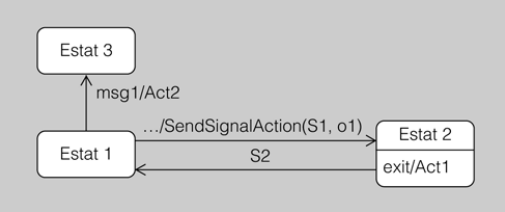


Figura 2.22. Diagrama d'estats de "Classe2"

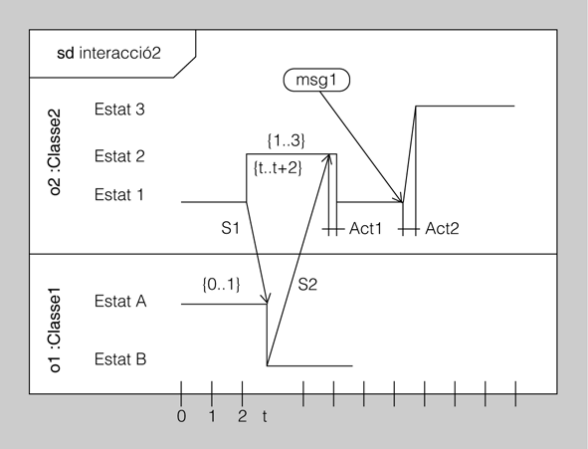


El diagrama de temps mostra una interacció entre l'objecte `o1` de la `Classe1` i l'objecte `o2` de la `Classe2`.

A la figura 2.23 es mostren les línies de vida de cadascun dels objectes, concretament dues franges temporals dins les quals es representen sengles línies de vida interrelacionades.

A la primera línia de vida hi ha dos estats i a la segona tres, l'ordre dels quals no té cap significat. Hi ha una escala de temps a la part de sota. Dels tres missatges que provoquen transicions el primer és conseqüència de la transició d'`Estat1` a `Estat2` (que provoca l'execució d'una activitat, l'acció estàndard `SendSignalAction(S1, o1)` la qual, d'acord amb els seus paràmetres, envia el senyal `S1` a l'objecte `o1`), el segon és conseqüència de l'entrada a `EstatB` i el tercer ve de l'exterior. El temps que passa entre l'arribada del segon missatge i la consegüent sortida d'`Estat2` és la durada de l'execució de l'activitat `act1`, i la durada de la transició d'`Estat1` a `Estat3` és la durada de l'execució de `act2`.

Figura 2.23. Diagrama temporal



Hi ha tres restriccions temporals:

- El missatge que transmet el senyal `S1` ha de durar com a màxim una unitat de temps.
- L'objecte `o1` ha de romandre a `Estat2` entre una i tres unitats de temps.
- El tercer missatge ha d'arribar no més tard de dues unitats de temps després de l'emissió del segon missatge.

La figura 2.24 és una versió més compacta del diagrama dels estats. En comptes de representar-hi els estats a diferents altures, se'n posen els noms damunt la línia de vida, dins dels intervals corresponents; aquest diagrama representa la primera línia de vida del de la figura 2.23.

Figura 2.24. Diagrama temporal simplificat

2.3.4 Diagrama de visió general de la interacció

El **diagrama de visió general de la interacció** aporta una visió global del flux de control de les interaccions.

Alhora de representar els models d'interacció d'un sistema pot sorgir el problema que siguin molt grans, amb la qual cosa és bastant immanejable el seu tractament. Amb els diagrames de visió general de la interacció, se segueix la filosofia "divideix i conqueriràs", ja que permeten modularitzar les interaccions, descomposant-

les en fragments més petits, i representar en un diagrama la relació entre els fragments.

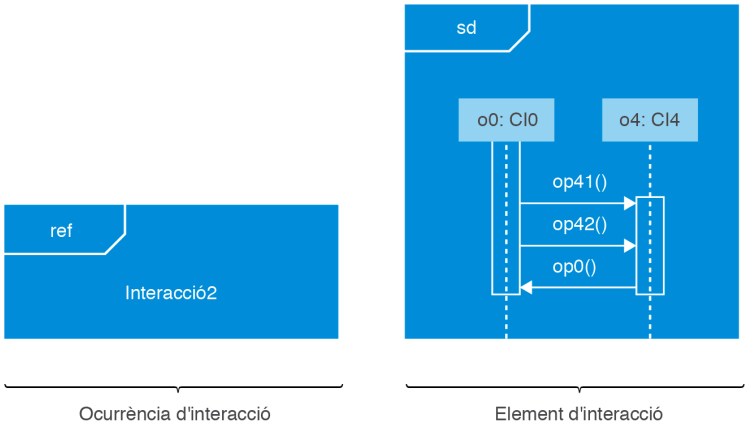
La representació del diagrama de visió general de la interacció fa ús de la nomenclatura del diagrama d'activitat incorporant dos conceptes nous:

- **Ocurrència d'interacció**: fa referència als diagrames d'interacció existents.
- **Element d'interacció**: es du a terme una representació de diagrames d'interacció existents dins un marc rectangular on se n'especifica el contingut.

En la secció *Annexos* del web del mòdul hi podeu trobar un exemple complert de diagrames de visió general de la interacció.

A la figura 2.25 es mostra la representació gràfica de cadascun dels dos conceptes.

FIGURA 2.25. Ocurrència d'interacció i element d'interacció



2.4 Modelio i UML: diagrames de comportament

UML és un llenguatge de modelatge de sistemes que es compon de molts diagrames. Cal comprendre quin és el significat de cadascun dels diagrames individualment, quins són els seus elements i com aquests es relacionen entre ells. Però per entendre d'una forma més global el que pot arribar a oferir UML cal observar un exemple complet, sobre el qual es desenvolupin tots els diagrames. Per fer aquest exemple global s'ha escollit un exemple clàssic, que es basa en el rentat de roba per part d'una rentadora.

Per a la creació d'aquest exemple de rentat de roba es fa servir l'entorn Modelio. Aquest entorn està especialitzat en el modelatge amb UML i altres llenguatges gràfics. També permet tant generar codi a partir dels diagrames com generar diagrames a partir de codi font. L'entorn Modelio ha estat implementat a partir de l'IDE Eclipse. La pàgina web de l'entorn Modelio és https://www.modelio.org/. Podeu descarregar-vos l'instal·lable des de l'apartat *Downloads*.

2.4.1 Diagrama de casos d'ús

A l'apartat "Diagrames estàtics" podeu trobar detallat el procediment per crear un projecte, un diagrama de classes i, també, com realitzar les operacions bàsiques amb **qualsevol tipus de diagrama**.

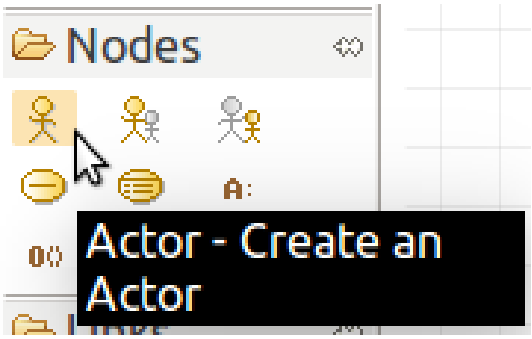
El cas d'ús descriu les accions d'un sistema des del punt de vista de l'usuari. En aquest cas, en Joan posa una rentadora amb l'objectiu que es renti la roba.

El diagrama de casos d'ús es crea de manera anàloga a com es creen els diagrames de classe, però seleccionant a l'últim pas *Use Case diagram* en lloc de *Class diagram*.

L'actor (en Joan) s'afegeix al diagrama:

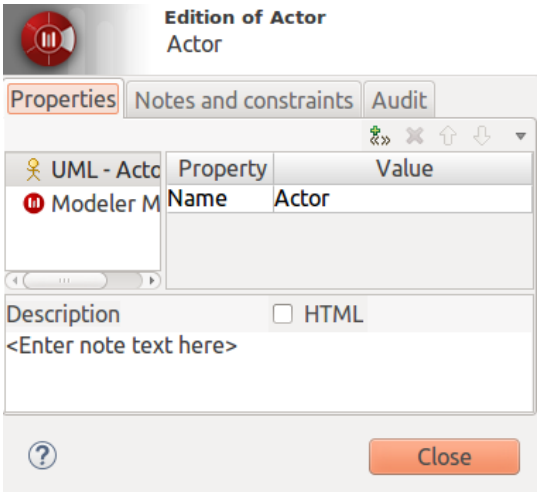
1. Fent clic a la icona *Actor*, a la secció *Nodes*, com es mostra a figura **??**.
2. Fent clic a continuació al punt del diagrama on volem inserir l'actor.

FIGURA 2.26. Icona //Actor//



Podem editar les seves propietats fent doble clic a sobre de la icona que representa l'actor. Ens apareixerà una finestra similar a la mostrada a la figura 2.27.

FIGURA 2.27. Propietats d'un //Actor//



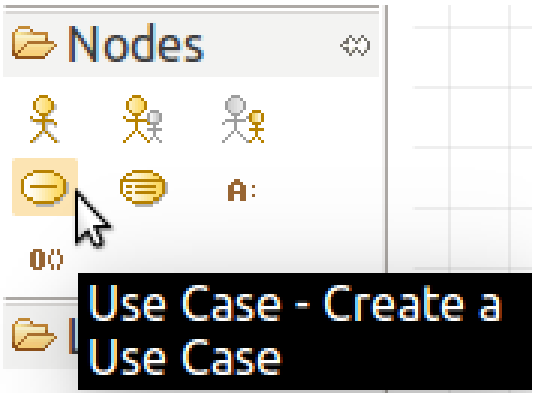
En aquest cas, només podem canviar el nom (*Name*). Hi posarem *Joan*.

Per especificar el cas d'ús *Rentar roba* cal:

1. Fer clic a la icona *Use Case*, a la secció *Nodes*, com es mostra a figura 2.28.

2. Fer clic a continuació al punt del diagrama on volem inserir el cas d'ús.

Figura 2.28. Icona //Use Case//



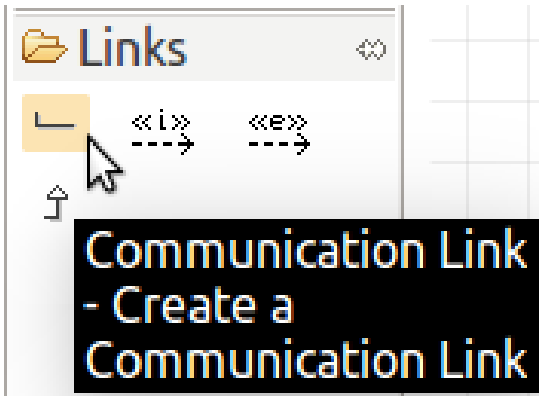
Igual que passava amb els actors, fent-hi doble clic ens apareixerà la finestra de les propietats on podem especificar el nom.

Ara cal enllaçar l'actor (en *Joan*) amb el cas d'ús (*Rentar roba*). Ho farem:

1. Fent clic a la *Communication Link*, a la secció *Links*, com es mostra a figura **??**.

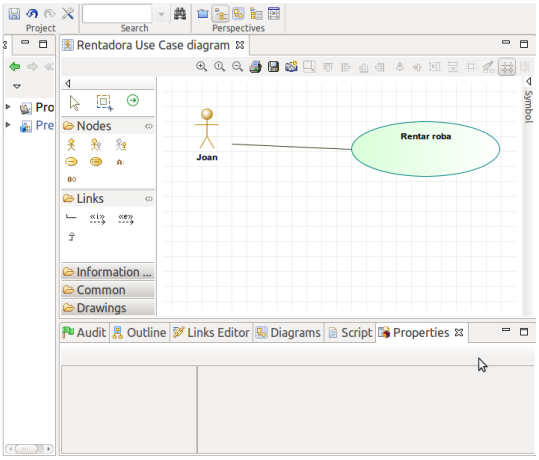
2. Fent clic a continuació a sobre de l'actor i, després, a sobre del cas d'ús.

Figura 2.29. Icona //Communication Link//



A la figura 2.30 es mostra el diagrama del cas d'ús resultant

FIGURA 2.30. Diagrama de cas d'ús: Rentar roba



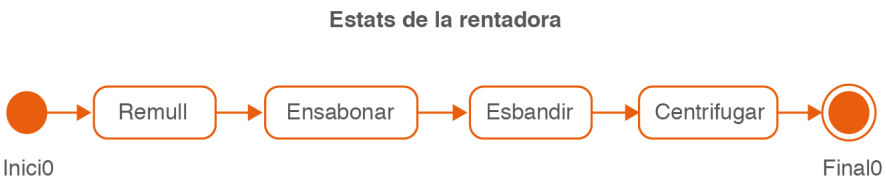
Normalment, cada cas d'ús va associat amb una explicació que sol contenir els següents apartats:

- **Nom**: `RentarRoba`.
- **Descripció**: Posar una rentadora amb l'objectiu que la roba bruta quedi neta.
- **Precondicions**: Roba bruta
- **Flux normal**: La rentadora neteja la roba.
- **Flux alternatiu**: No.
- **Postcondicions**: Roba neta.

2.4.2 Diagrama de transició d'estat

En qualsevol moment, un objecte es troba en un estat en particular. Una rentadora pot estar en la fase de remull, rentat o ensabonat, esbandida, centrifugat i apagada, i canviarà d'una a una altra, d'acord amb el diagrama d'estat que es mostra a la figura 2.31.

FIGURA 2.31. Diagrama de transició d'estat



El símbol de la part superior indica l'estat inicial, i el de la part inferior, el final.

Totes les icones que cal utilitzar en el diagrama d'estats són a l'apartat ***States***.

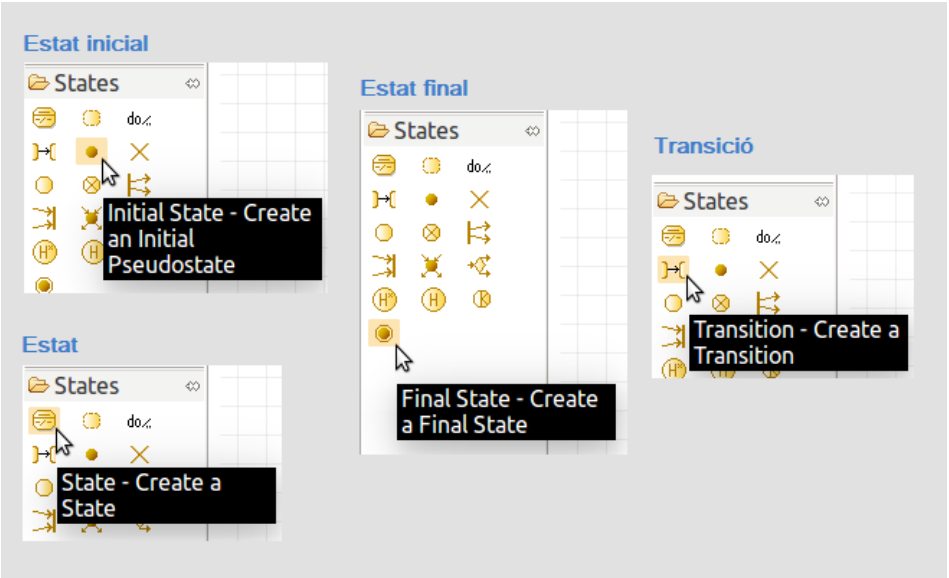
A Modelio, el diagrama de transició d'estats es crea de manera anàloga a com es crea la resta de diagrames, però seleccionant a l'últim pas *State Machine diagram* com a tipus de diagrama.

Un cop creat el diagrama, els estats s'hi afegeixen clicant la icona de la barra d'elements que representa el tipus d'estat que volem afegir (inicial, final o -simplement- estat) i, a continuació, clicant al lloc del diagrama on el volem situar.

Per afegir una transició, també es clica la icona que la representa i, a continuació, als dos estats del diagrama que volen unir-se. El nom de cada transició s'entra en la seva propietat *Guard*.

Podeu veure les icones a la figura 2.32.

FIGURA 2.32. Icones del diagrama de transició d'estats.

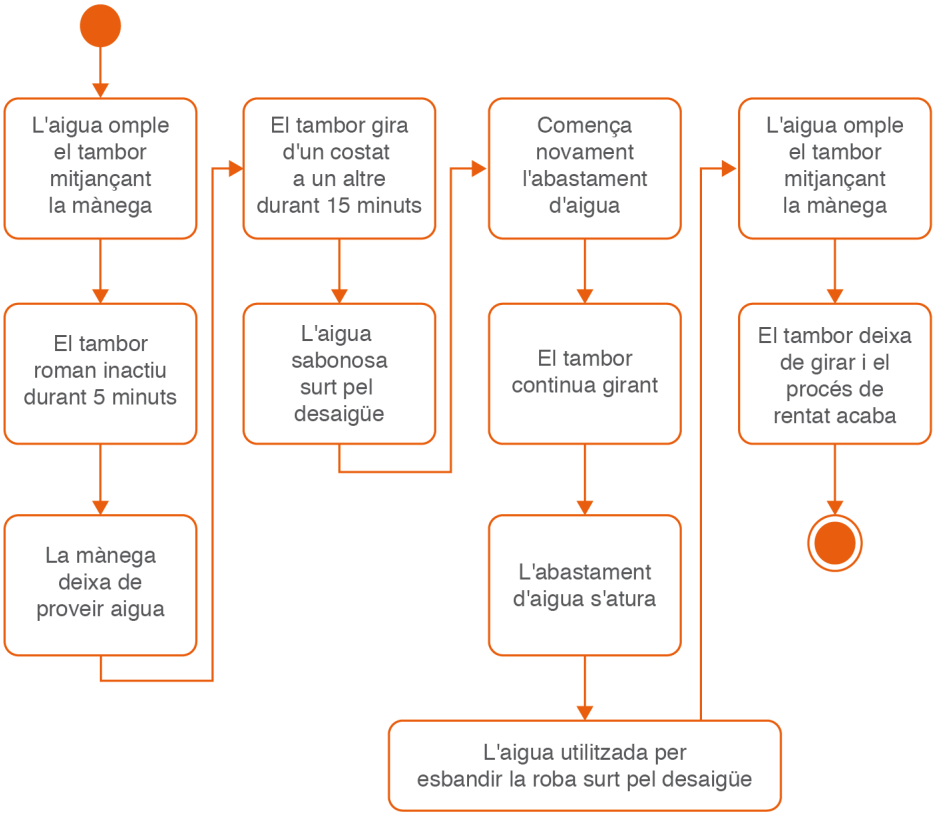


2.4.3 Diagrama d'activitats

Un diagrama d'activitats és una variació del diagrama d'estats UML. El diagrama d'activitats representa les activitats que passen dins un cas d'ús o dins el comportament d'un objecte.

A la figura 2.33 es mostra un possible diagrama d'activitats.

Figura 2.33. Diagrama d'activitats



Fixem-nos que cada una de les columnes podria correspondre a cada un dels estats del procés de rentat: remull, ensabonat, esbandit i centrifugat.

- **Estat de remull:**
  - L'aigua omple el tambor mitjançant la mànega.
  - El tambor roman inactiu durant cinc minuts.
  - La mànega deixa de proveir aigua.
- **Estat d'ensabonat o de rentat:**
  - El tambor gira d'un costat a un altre durant quinze minuts.
  - L'aigua ensabonada surt pel desaigüe.
- **Estat d'esbandit:**
  - Comença novament l'abastament d'aigua.
  - El tambor continua girant.
  - L'abastament d'aigua s'atura.
  - L'aigua utilitzada per esbandir la roba surt pel desaigüe.
- **Estat de centrifugat:**
  - El tambor gira en una sola direcció.
  - El tambor deixarà de girar i el procés de rentat acaba.

A Modelio, el diagrama d'activitats es crea de manera anàloga a com es crea la resta de diagrames, però seleccionant a l'últim pas *Activity diagram* com a tipus de diagrama.

Un cop creat el diagrama, els estats inicial i final i les accions s'afegeixen clicant a la icona de la barra d'elements corresponent al tipus d'element que volem afegir i, a continuació, clicant al lloc del diagrama on el volem situar-lo. Aquestes icones són a l'apartat *Control nodes*.

Per afegir un flux de control, també es clica en primer lloc la icona que el representa, que és a l'apartat *Flows*. A continuació, cal clicar consecutivament els dos elements del diagrama que volen unir-se. Els elements a unir tant poden ser estats com accions com *Forks*, *Joins* o nodes de decisió. Un cop afegit un flux de control al diagrama, es pot canviar la seva forma seleccionant-lo i arrossegant els quadres negres que apareixen en seleccionar-lo.

Per afegir un element *Fork* o *Join*, cal fer clic a la icona *Fork/Join*, que és dins de l'apartat *Control nodes*. Aquesta icona pot representar tant un *Fork* com un *Join* depenent dels fluxos que hi arriben i en surten. Un cop afegit aquest node al diagrama, es pot moure tot arrossegant-lo amb el ratolí. També es pot canviar la seva mida arrossegant novament amb el ratolí de la següent manera:

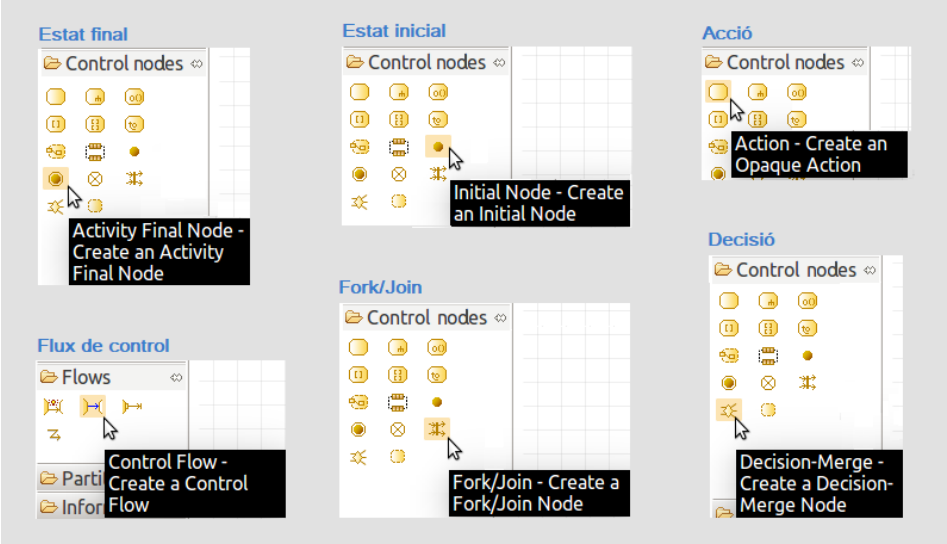
- Seleccionant l'element *Fork-Join*.

- Situant el ratolí en l'extrem que volem allargar o escurçar, entre els dos quadres negres que indiquen que l'element s'ha seleccionant, i arrossegant. El ratolí és a la posició correcta per arrossegar quan té forma de doble fletxa horitzontal. Si la doble fletxa no és horitzontal o el ratolí té una altra forma, no serà possible realitzar l'arrossegament.

Per últim, podeu afegir un node de decisió fent clic a la icona que representa aquests nodes i que es troba a l'apartat *Control nodes* i, a continuació, clicant el punt del diagrama on voleu situar-lo. Podem escriure la condició associada a cadascun dels fluxos de sortida a la seva propietat *Guard*.

Podeu veure les icones d'aquest tipus de diagrama a la figura 2.34.

Figura 2.34. Icones del diagrama d'activitats.



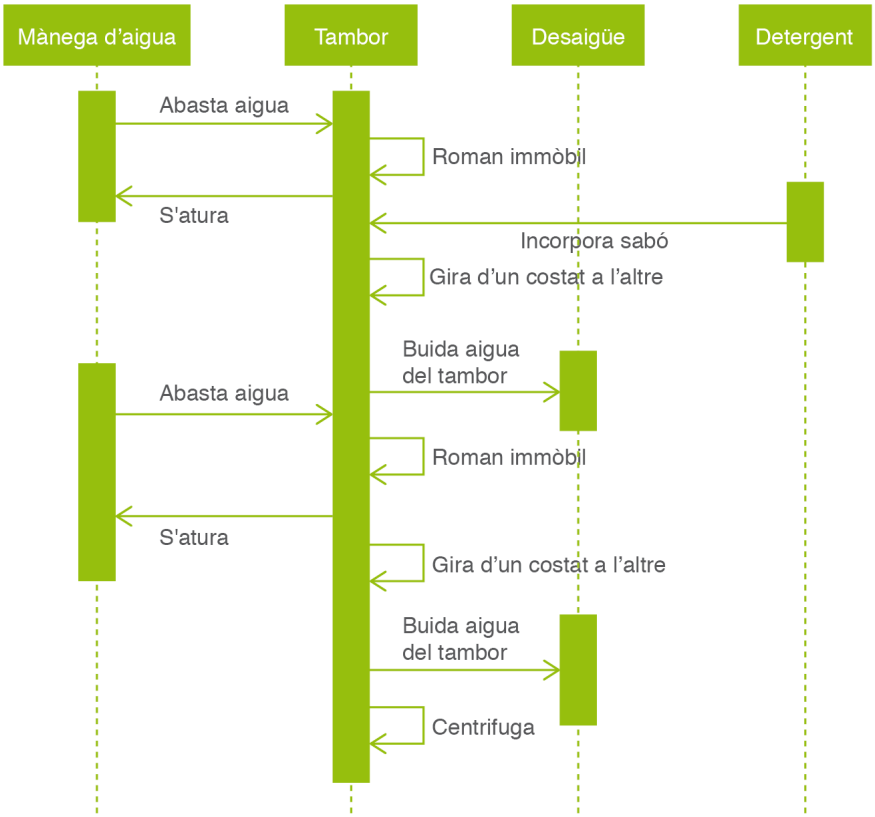
2.4.4 Diagrama de seqüències

En un sistema funcional els objectes interactuen entre si, i aquestes interaccions succeeixen amb el temps. El diagrama de seqüències UML mostra la mecànica de la interacció sobre la base del temps.

Entre els components de la rentadora es troben: una mànega d'aigua (per obtenir aigua fresca), un tambor (on es posa la roba), un sistema de desaigüe i el dispensador de detergent.

Què passarà quan invoqui al cas d'ús *Rentar roba*? Si donem per fet que s'han completat les operacions *afegir roba*, *afegir detergent* i *activar*, la seqüència seria més o menys la que mostra la figura 2.35.

Figura 2.35. Diagrama de seqüència



La seqüència de passos que segueix el procés de rentat és:

1. El sistema d'abastament d'aigua omple el tambor.
2. Durant 3 minuts no hi ha activitat al tambor.
3. El sistema d'abastament deixa de proporcionar aigua.
4. El sistema d'abastament proporciona sabó.
5. Durant 1 minut no hi ha activitat al tambor.
6. El sistema d'abastament deixa de proporcionar sabó.
7. Durant 12 minuts el tambor gira.
8. Comença a sortir aigua pel desaigüe fins que no deixa sabó.
9. El sistema d'abastament d'aigua omple el tambor.
10. Durant 10 minuts el tambor gira.
11. El sistema d'abastament deixa de proporcionar aigua.
12. El desaigüe rep l'aigua que es fa servir per esbandir.
13. Durant 4 minuts el tambor gira.
14. Es finalitza el procés. El tambor deixa de girar.

A Modelio, el diagrama de seqüències es crea de manera anàloga a com es crea la resta de diagrames, però seleccionant a l'últim pas *Sequence diagram* com a tipus de diagrama.

Les línies de vida s'afegeixen clicant la icona corresponent, que és a l'apartat *Nodes*, i, a continuació, fent clic al punt del diagrama on volem afegir-la. Podem donar-les nom tot modificant la propietat *Name*.

Els missatges s'afegeixen:

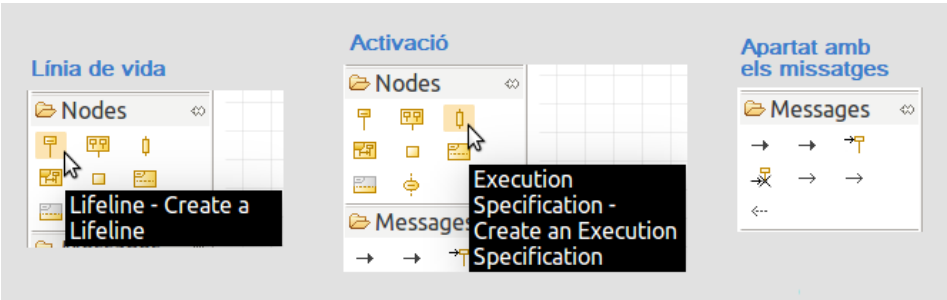
1. Fent clic a la icona corresponent al tipus de missatge que volem afegir, que serà a l'apartat *Messages*.
2. Fent clic a la línia de vida origen del missatge.
3. Fent clic a la línia de vida destinatària del missatge. Veureu que en la línia de vida destinatària s'afegeix automàticament un rectangle, que respresenta una activació. A més, si el missatge és síncron, es crea automàticament el missatge de resposta.

Les activacions també poden afegir-se directament fent clic primer a la icona que les representa, dins l'apartat *Nodes*, i, a continuació, fent clic a la línia de vida a la qual desitgem associar-la.

Pot canviar-se la mida d'una activació seleccionant-la i, a continuació arrosegant el missatge que arriba o surt de l'extrem que volem pujar o baixar. En cas que l'extrem no tingui cap missatge, cal arrossegar el costat superior o inferior de l'activació. El ratolí és a la posició correcta per arrossegar quan té forma de doble fletxa vertical. Si la doble fletxa no és vertical o el ratolí té una altra forma, no serà possible realitzar l'arrossegament.

Podeu veure les icones d'aquest tipus de diagrama a la figura 2.36.

Figura 2.36. Icones del diagrama de seqüències.

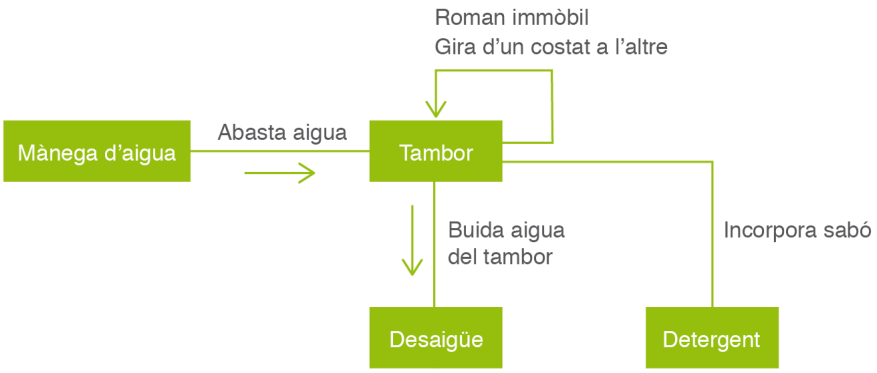


2.4.5 Diagrama de comunicació

Els elements d'un sistema treballen en conjunt per complir amb els objectius del sistema, i un llenguatge de modelització haurà de comptar amb una forma de representar això. El diagrama de col·laboracions UML està dissenyat amb aquesta finalitat.

A la figura 2.37 es pot observar un possible diagrama de comunicació.

Figura 2.37. Diagrama de comunicació



A Modelio, el diagrama de comunicació es crea de manera anàloga a com es crea la resta de diagrames, però seleccionant a l'últim pas *Communication diagram* com a tipus de diagrama.

Els objectes s'afegeixen fent clic en primer lloc a sobre de la icona *Communication Node* i, a continuació, al punt del diagrama on volem inserir-lo. Cada objecte quedarà representat com un rectangle. Podem modificar el seu nom des de la pestanya *Properties*.

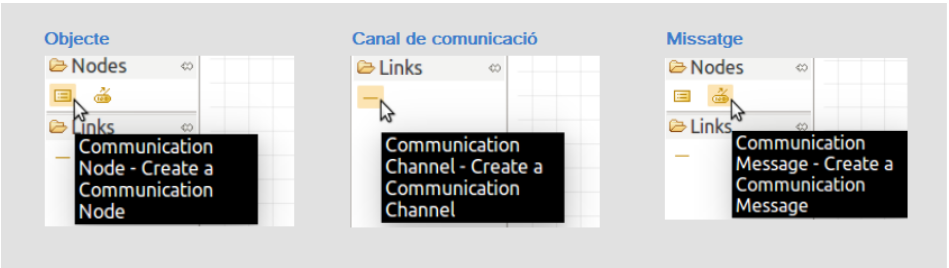
La inserció d'un missatge entre dos objectes es fa en dos passos:

1. Insertar un *canal de comunicació* entre tots dos objectes. Això es fa clicant la icona *Communication Channel* i, a continuació, clicant en els dos objectes que es comunicaran amb aquest missatge. L'ordre és indiferent.

2. Clicar la icona *Communication Message* i, a continuació, el canal que uneix els dos objectes que es comuniquen amb aquest missatge. El punt on es faci el clic és important, ja que determinarà el **sentit** del missatge: el missatge anirà des de l'objecte més proper al clic cap a l'objecte més llunyà al clic. El missatge resultant tindrà forma de fletxa, que indicarà el sentit del missatge, i apareixerà més propera a l'objecte emissor que no pas a l'objecte receptor.

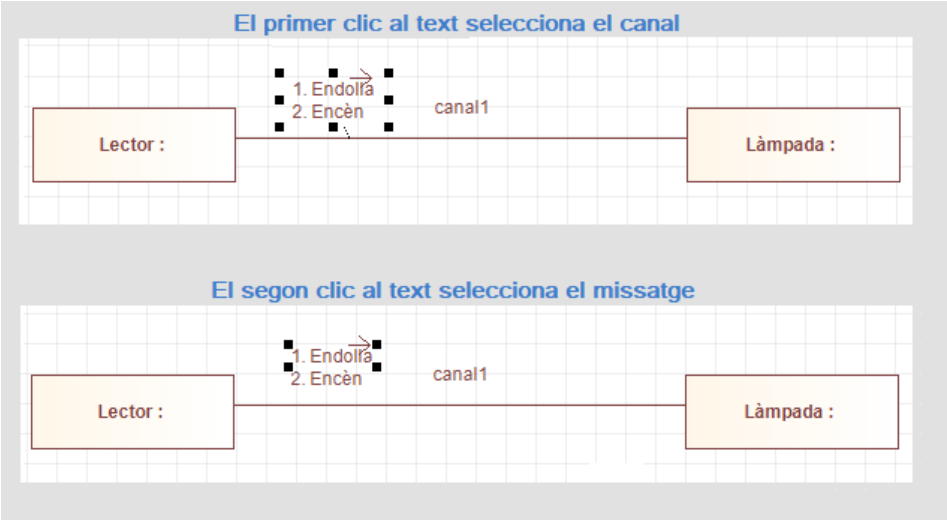
Podeu veure les icones d'aquest tipus de diagrama a la figura **??**.

Figura 2.38. Icones del diagrama de comunicació



Un canal admet més d'un missatge. Per representar-ho al diagrama cal repetir el pas *2* tantes vegades com ens calgui. Els missatges que van en el mateix sentit es representaran al voltant de la fletxa corresponent segons l'ordre d'inserció. Cadascun ve representat per una línia de text. Podem canviar el text que descriu el missatge seleccionant amb el ratolí la línia corresponent i modificant la seva propietat *Name*. Normalment, en clicar-lo el primer cop es seleccionarà tot el canal i els petits quadres negres que indiquen que l'element s'ha seleccionat apareixeran al voltant de tots els missatges que van en el mateix sentit. Per seleccionar només un missatge cal tornar a fer clic a sobre d'aquest; llavors els quadres negres només envoltaran el missatge triat per indicar que ara l'únic missatge seleccionat és aquest. A la figura 2.39 podeu observar la seqüència de clics que cal fer.

Figura 2.39. Selecció d'un missatge



A la figura 2.40 podeu observar un exemple de diagrama de comunicació realitzat amb Modelio:

Figura 2.40. Diagrama de comunicació elaborat amb Modelio