

Optimització de programari

Marcel García Vacas

Entorns de desenvolupament

Índex

Introducció	5
Resultats d'aprenentatge	7
1 Disseny i realització de proves de programari	9
1.1 Introducció	9
1.2 Les proves en el cicle de vida d'un projecte	10
1.3 Procediments, tipus i casos de proves	11
1.3.1 Planificació de les proves	12
1.3.2 Disseny de les proves. Tipus de proves	14
1.3.3 Execució de les proves	39
1.3.4 Finalització: avaluació i anàlisi d'errors	41
1.3.5 Depuració del codi font	41
1.4 Eines per a la realització de proves	43
1.4.1 Beneficis i problemes de l'ús d'eines de proves	44
1.4.2 Algunes eines de proves de programari	45
2 Eines per al control i documentació de programari	47
2.1 Refacció	47
2.1.1 Avantatges i limitacions de la refacció	49
2.1.2 Patrons de refacció més usuals	52
2.2 Proves i refacció. Eines d'ajuda a la refacció	57
2.2.1 Eines per a l'ajuda a la refacció	59
2.3 Control de versions	60
2.3.1 Components d'un sistema de control de versions	62
2.3.2 Classificació dels sistemes de control de versions	64
2.3.3 Operacions bàsiques d'un sistema de control de versions	67
2.4 Eines de control de versions	69
2.4.1 Altres eines	70
2.5 Dipòsits de les eines de control de versions	70
2.5.1 Problemàtiques dels sistemes de control de versions	71
2.6 Utilització de Git	77
2.6.1 Instal·lació	77
2.6.2 Operacions bàsiques	79
2.6.3 Operacions avançades	84
2.6.4 Integració amb entorns de desenvolupament integrats: Eclipse	90
2.7 Utilització de Github	101
2.7.1 Gestió de repositoris privats i públics	102
2.7.2 Configuració d'un repositori de GitHub	104
2.7.3 Gestió d'errors ('issues')	105
2.8 Comentaris i documentació del programari	106
2.8.1 Estructura dels comentaris tipus JavaDoc	108
2.8.2 Exemple de comentaris tipus JavaDoc	110

2.8.3 Exemple d'utilització de JavaDoc amb Eclipse 110

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 reproduïen 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 compleixin 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 compliment. 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 venda 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 disminuiria 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 èxit. Com més aviat es detecti un error, menys costos serà de solucionar.

També seran molt importants les proves que es duren 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.

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

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

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.

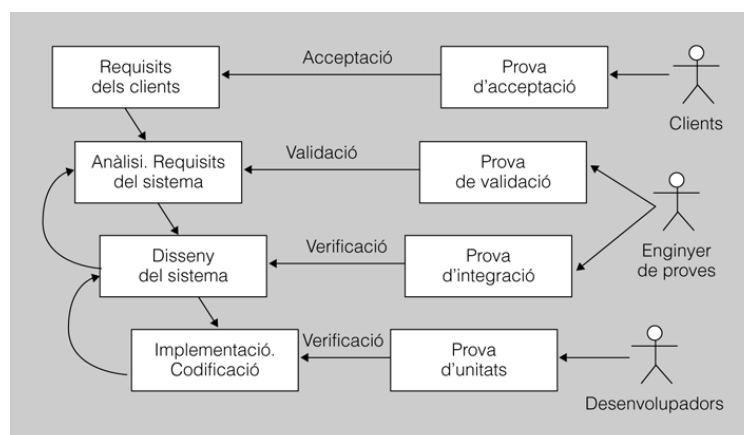
Precisament, és l'objectiu d'aquestes proves: l'avaluació de la qualitat del programari desenvolupat durant tot el seu cicle de vida, validant que fa el que ha de fer i que ho fa tal com es va dissenyar, a partir dels requeriments.

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.

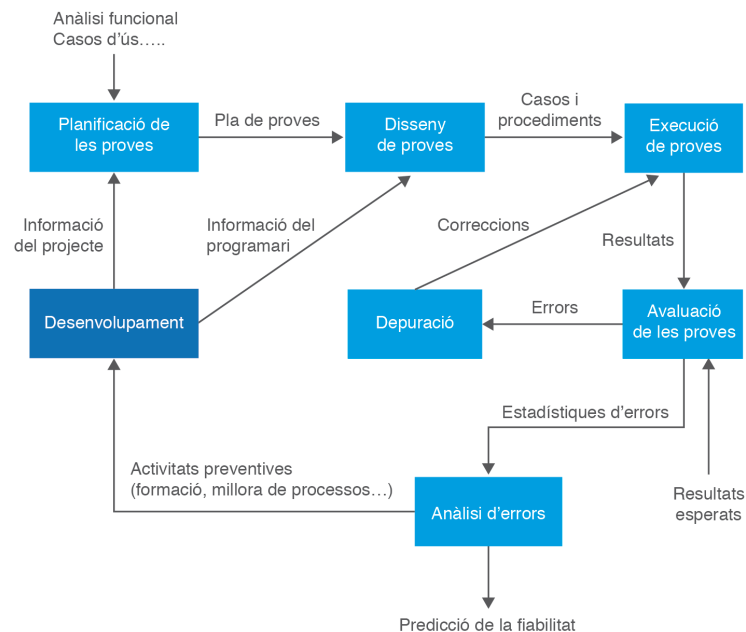
Aquest projecte de proves requerirà d'una planificació, un disseny del pla de proves, una execució de les mateixes i una avaluació dels resultats, per tal d'analitzar els errors i poder aplicar les accions necessàries. A la figura 1.2 es mostra un esquema amb els procediments que caldrà dur a terme i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

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.

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

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 duren 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 tasques 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.

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

El mètode `CasProva_CostMatricula` calcularà el preu que haurà de pagar un alumne per matricular-se en diverses assignatures. La prova valida que el càlcul 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 += CreditsAssignatura ("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 }
```

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

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

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).

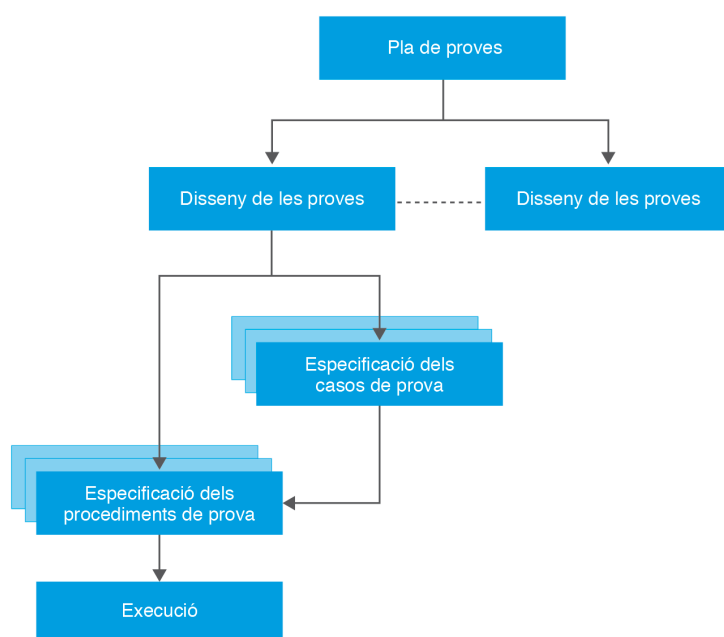
Tot cas de prova està associat, com a mínim, a un procediment de prova.

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.

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



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...

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 duren 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 duren 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 duren 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 duren a terme els clients o els usuaris finals de l'aplicació.
 - Aquestes proves es duren 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 introduïts 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.

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:

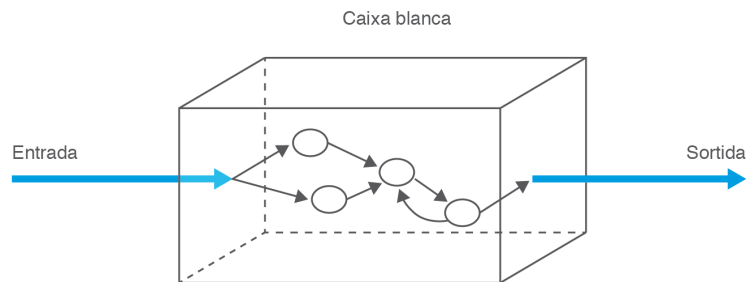
- L'enfocament estructural és la part de les proves unitàries encarregades de l'estructura interna del codi font, des de qual s'analitzen tots els possibles camins.
- L'enfocament funcional (o proves de capsa negra) és la part de les proves unitàries encarregades del funcionament correcte de les funcionalitats del programari.

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

Les proves de capsula 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 capsula 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'obtingran les condicions desitjades de sortida.

FIGURA 1.4. Estructura de les proves de capsula blanca



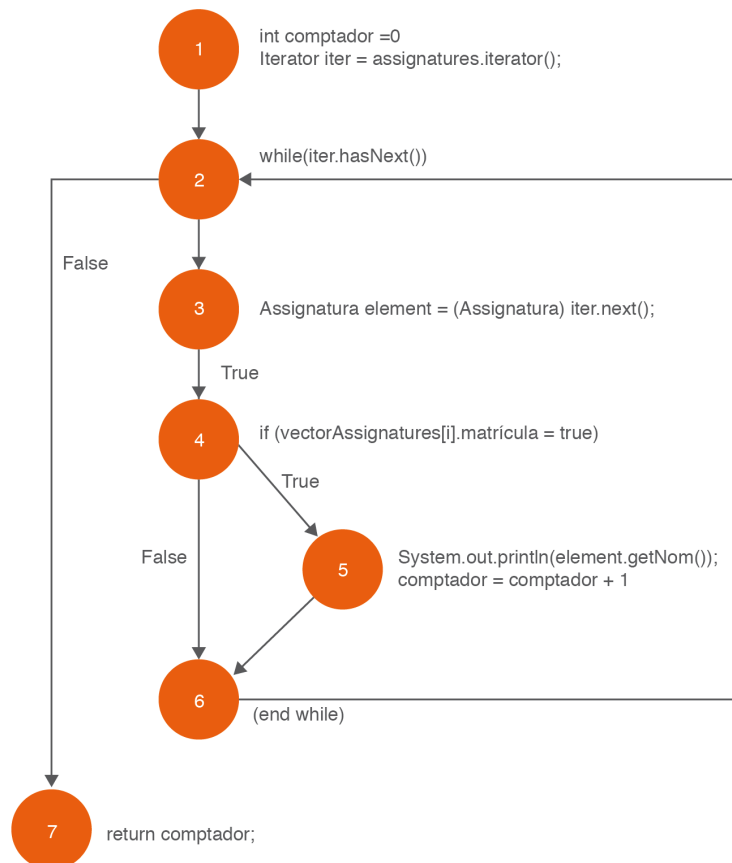
Les proves de capsula blanca permetran recórrer tots els possibles camins del codi i veure què succeeix en cada cas possible. Es provarà què ocorre amb les condicions i els bucles que s'executen. Les proves es duren a terme amb dades que garanteixin que han tingut lloc totes les combinacions possibles. 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 capsula 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 caps 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:

$$\text{Complexitat ciclomàtica} = \text{nombre de branques} - \text{nombre de nodes} + 2$$

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.

$$\text{Complexitat ciclomàtica CC} = 8 - 7 + 2 = 3$$

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 depèn 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 capsula blanca, es pot afirmar que han estat superades satisfactòriament.

Proves unitàries: enfocament funcional o proves de capsula negra

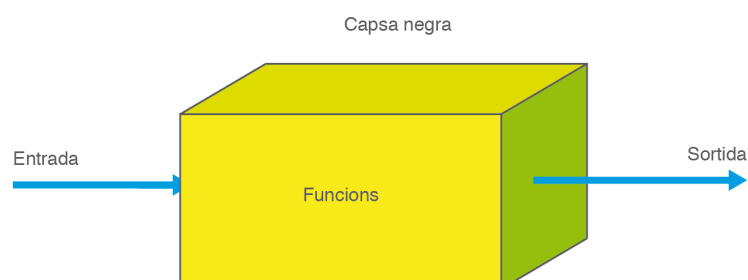
L'enfocament estructural o les proves de capsula blanca, dins les proves unitàries, serveix per analitzar el codi en totes les seves estructures, en tots els seus camins del programari. Però existeix un altre tipus de proves que es basa en un enfocament més funcional, anomenades proves de capsula negra.

Les proves de capsula 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 capsula 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 capsula blanca.

FIGURA 1.7. Estructura de les proves de capsula 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.

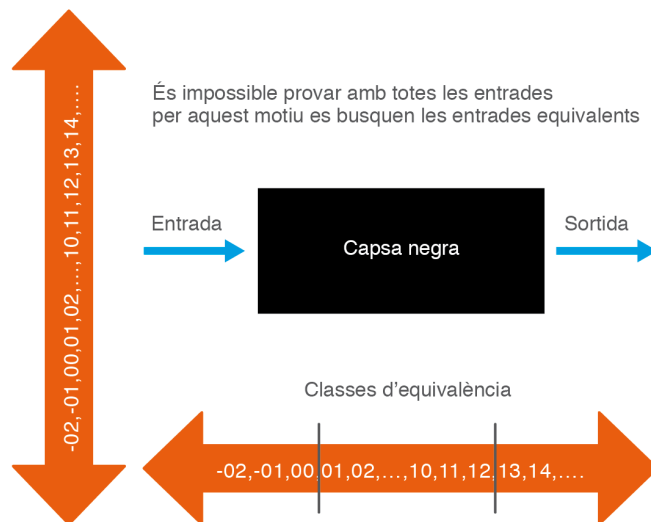
Per exemple, es volen definir les proves de capsa negra per a una funció que retorna el nom del mes a partir del seu valor numèric.

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

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

```
1 ..., -02, -01, 00 valors invàlids
2 01, 02, 03, 04, 05, 06, 07, 08, 09, 10, 11, 12 valors vàlids
3 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.

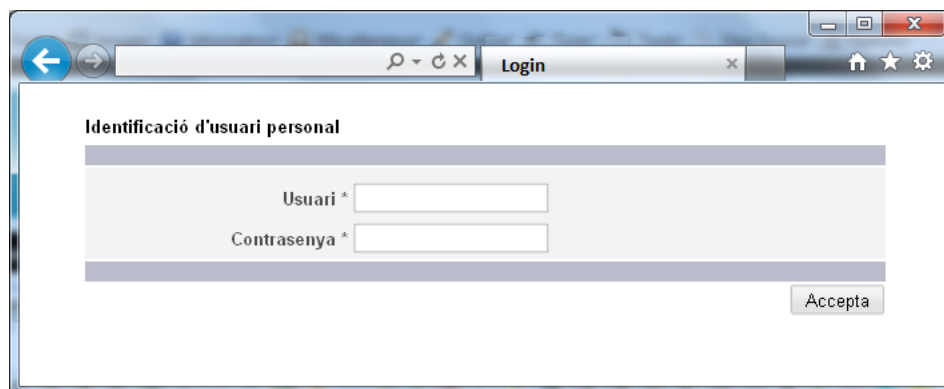
Actualment, la majoria de programes solen interactuar amb l'usuari fent ús de sistemes gràfics que cada vegada són més complexos, amb la qual cosa es poden generar errors.

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 desplegable...
- Proves de documentació i ajuda del programa.
- Altres.

A la figura 1.9 es mostra una 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.

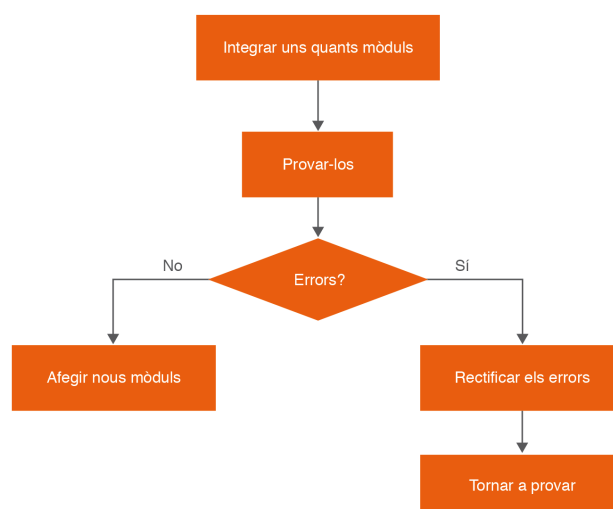
Proves d'integració

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.

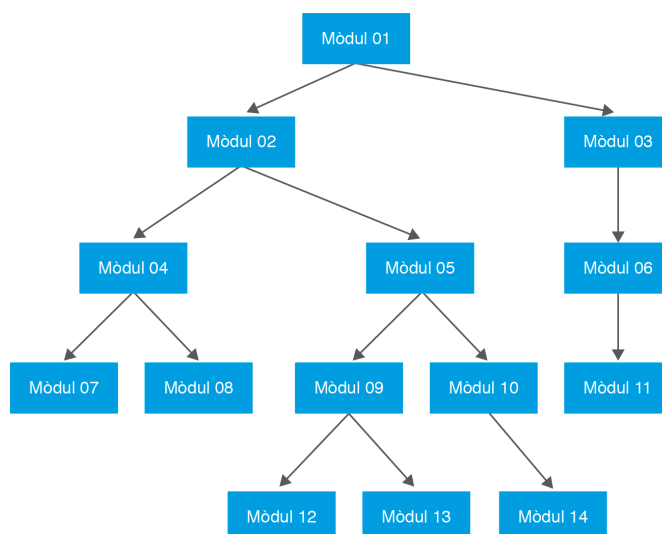
Prova d'integració ascendent

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.

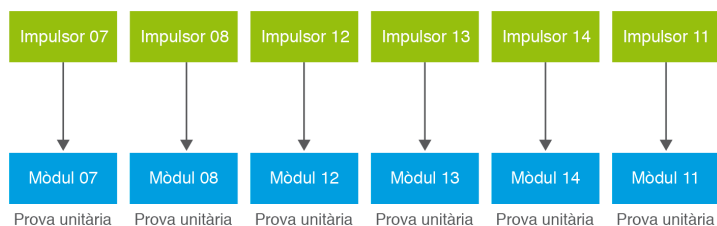
A mesura que els resultats d'aquestes proves vagin sortint positius, s'anirà escalant per l'arbre de jerarquies amb el mòdul impulsor cap als altres mòduls, fent les crides pertinents de forma recursiva. La darrera prova serà una crida al programari sencer amb els valors d'entrada reals (analitzant els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

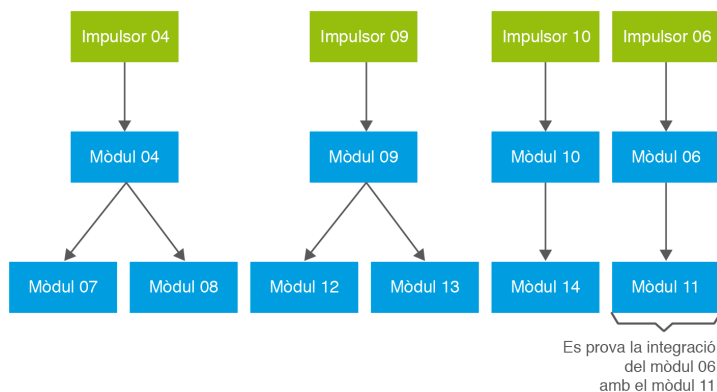
FIGURA 1.11. Esquema de proves d'integració ascendents



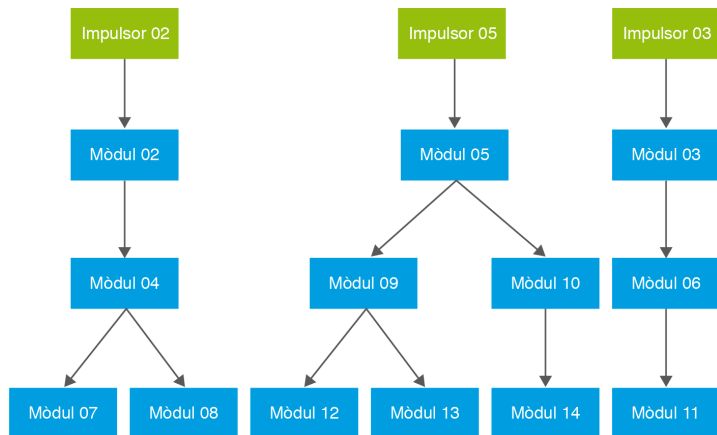
A la figura 1.12 es pot observar la primera fase de les proves d'integració ascendents. Cada mòdul ha de ser provat per separat, per això s'ha de construir un mòdul impulsor independent per provar cada mòdul.

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).

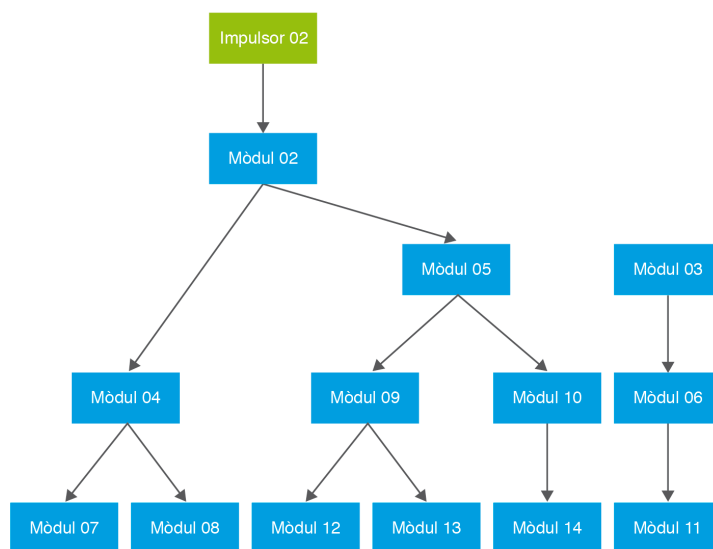
FIGURA 1.13. Integració incremental ascendent, fase 2

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

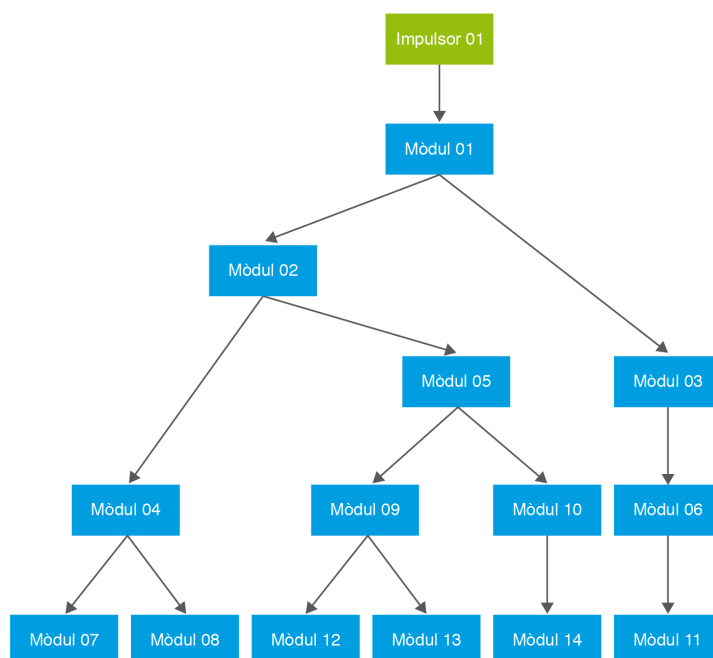
A la figura 1.15 es veu la integració dels mòduls 04 i 05 amb el mòdul 02, per al qual s'haurà de crear l'impulsor 02, que cridarà a aquest mòdul.

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

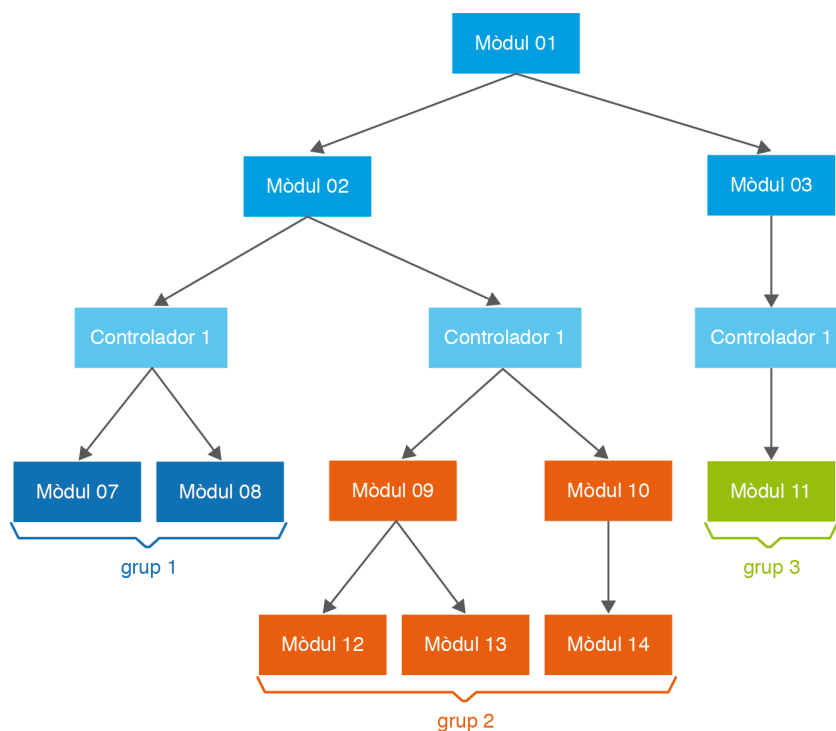
Aquesta estratègia de desenvolupament de les proves d'integració començarà pel mòdul de control principal (el més important, el de més nivell). Una vegada validat, s'aniran integrant els altres mòduls que en depenen de forma progressiva, sense seguir una estratègia concreta, només tenint en compte que 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 informará 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 duren 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.

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.

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.

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

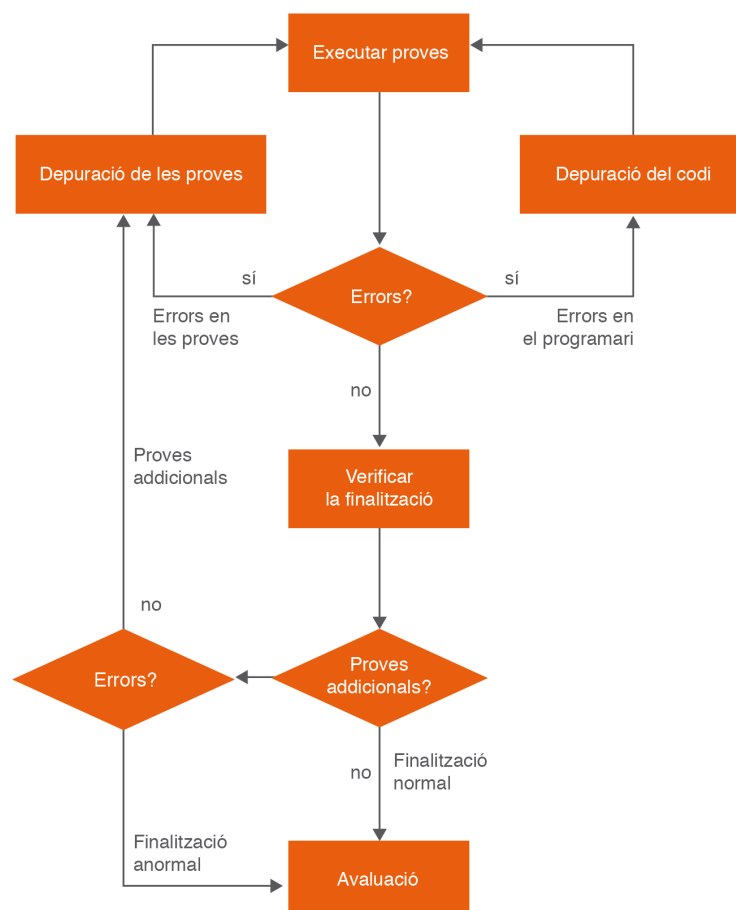
Després de la planificació dels procediments de proves i del disseny dels casos de proves, el següent pas serà el procés d'execució. Aquests procés està representat en el diagrama de flux de la figura 1.18.

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 aconsegueix 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 reproduïxen.

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.

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

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.

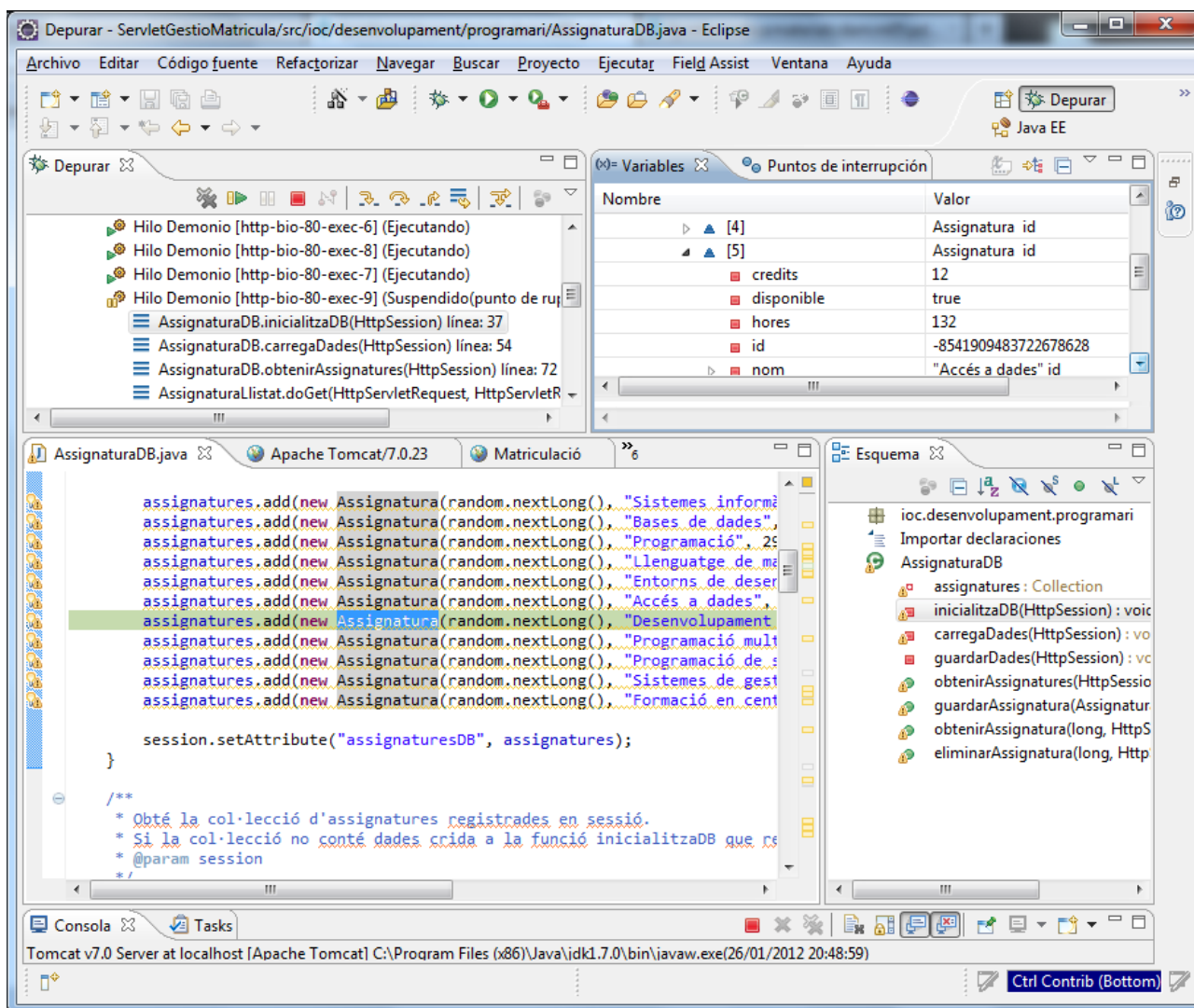
Una eina que ajuda a la depuració del codi font és la utilització de fitxers anomenats registres (*logs*). Aquests fitxers, habitualment de text, enregistren tota la informació vinculada a l'execució d'un programari amb l'objectiu que el programador pugui fer una revisió pas a pas de com ha evolucionat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

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.

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

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 privatis 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.

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

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

- 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:

```
1 Class CalculCostos {  
2     final double SALARI_BRUT= 1200;  
3     Public static double CostTreballadors (double NreTreballadors)  
4     {  
5         Return SALARI_BRUT* NreTreballadors;  
6     }  
7 }
```

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 duplictat 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

Refactorizar	Navegar	Search	Proyecto	Ejecutar	Field Assist
Redenominar...					Alt+Mayús+R
Mover...					Alt+Mayús+V
Cambiar signature de método...					Alt+Mayús+C
Extraer método...					Alt+Mayús+M
Extraer variable local...					Alt+Mayús+L
Extraer constante...					
Incorporar...					Alt+Mayús+I
Convertir variable local en campo...					
Convertir clase anónima en anidada...					
Move Type to New File...					
Extraer superclase...					
Extraer interfaz...					
Utilizar supertipo cuando sea posible...					
Degradar...					
Promover...					
Extraer clase...					
Introducir el parametro del objeto					
Introducir direccionamiento indirecto...					
Introducir fábrica...					
Introducir parámetro...					
Autoencapsular campo...					
Generalizar tipo declarado...					
Inferir argumentos de tipo genérico...					
Migrar archivo JAR...					
Crear script...					
Aplicar script...					
Historial...					

Els patrons més habituals són els següents:

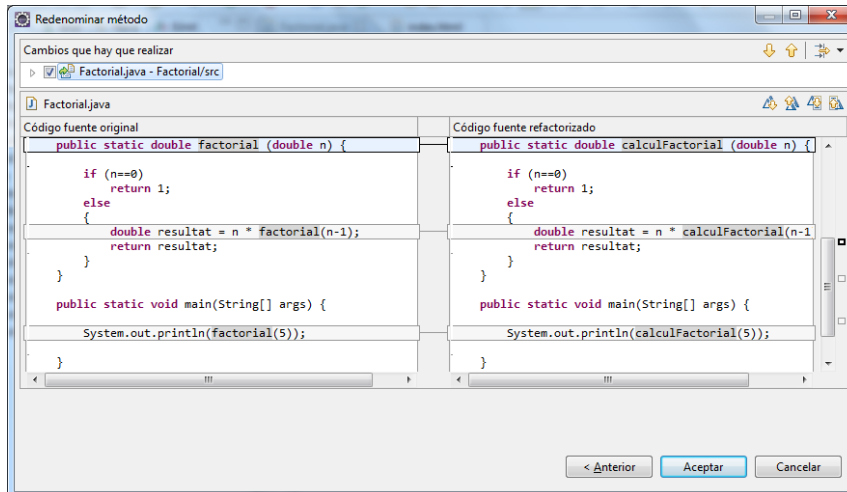
- Reanomenar
- Moure
- Extreure una variable local
- Extreure una constant
- Convertir una variable local en un camp
- Extreure una interfície
- Extreure el mètode

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

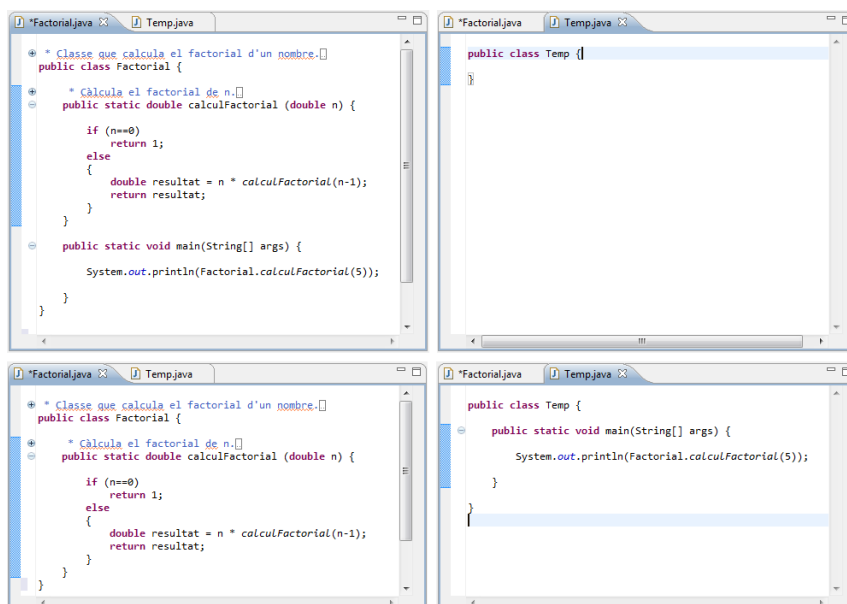


Moure

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         );  
5     nre = 5;  
6     System.out.println("El factorial de " + nre + " és " + calculFactorial(nre)  
7         );  
8 }
```

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 substituïda 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         );  
5     nre = 5;  
6     System.out.println("El factorial de " + nre + " és " + calculFactorial(nre)  
7         );  
8 }
```

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        );
15        nre = 5;
16        System.out.println("El factorial de " + nre + " és " + calculFactorial(nre)
17        );
18    }
19 }
```

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        );
16        nre = 5;
17        System.out.println("El factorial de " + nre + " és " + calculFactorial(nre)
18        );
19    }
20 }
```

Extreure una interfície

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
```

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.

```
6      {
7          double resultat = n * calculFactorial(n-1);
8          return resultat;
9      }
10     }
11 }
```

El codi refactoritzat és el que es mostra tot seguit:

```
1 public interface InterficieFactorial {
2     public abstract double calculFactorial(double n);
3 }
```

```
1 public class Factorial implements InterficieFactorial {
2     /* (non-Javadoc)
3      * @see InterficieFactorial#calculFactorial(double)
4      */
5     @Override
6     public double calculFactorial (double n) {
7         if (n==0)
8             return 1;
9         else
10            {
11                double resultat = n * calculFactorial(n-1);
12                return resultat;
13            }
14     }
15 }
```

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`.

```
1 public static void main(String[] args) {
2     int nre = 3;
3     int comptador = 1;
4     double resultat = 1;
5     while (comptador<=nre){
6         resultat = resultat * comptador;
7         comptador++;
8     }
9     System.out.println("Factorial de " + nre + ": " + calculFactorial(nre));
10 }
```

El codi refactoritzat és el que es mostra tot seguit:

```
1 private static void calcularFactorial(int nre) {
2     int comptador = 1;
3     double resultat = 1;
4     while (comptador<=nre){
5         resultat = resultat * comptador;
6         comptador++;
7     }
8 }
9 public static void main(String[] args) {
10     int nre = 5;
11     System.out.println("Factorial de " + nre + ": " + calcularFactorial(nre));
12 }
```


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 compleixin 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

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:

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

- **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 wikipèdia, 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:

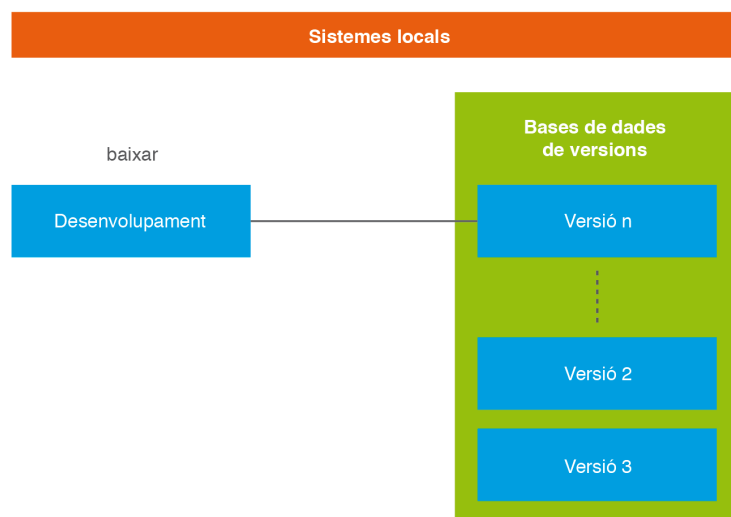
- El mateix programador serà la persona que s'haurà de recordar d'anar fent les còpies de seguretat cada cert temps, el que ell hagi establert.
- El mateix programador haurà de decidir on durà a terme aquestes còpies de seguretat, molt probablement en local, en una altra ubicació del disc dur intern, 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 oblitls 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



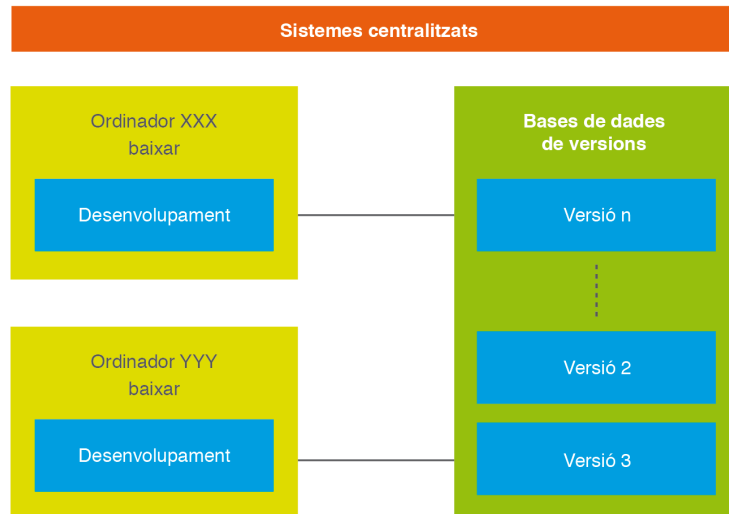
Sistemes centralitzats

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

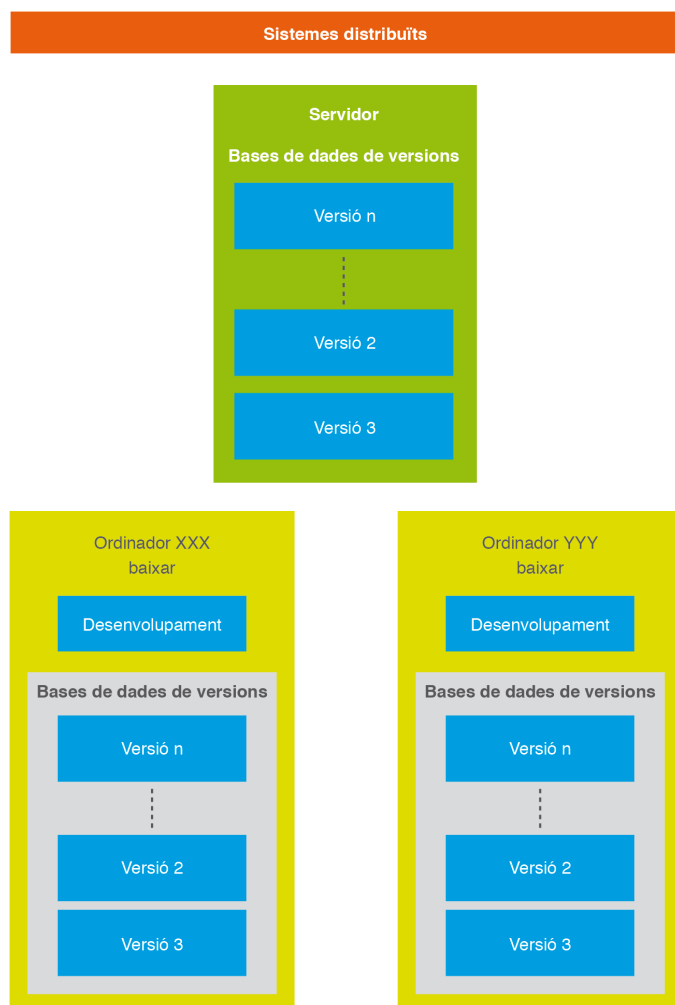


Però aquest sistema també tindrà els seus inconvenients, de vegades molt difícils de salvar. Què succeeix si falla el servidor central? Si és una fallada temporal no s'hi podrà accedir durant un petit període temps. Si és una fallada del sistema, caldrà tenir preparat un sistema de recuperació o un sistema 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 branca es durà a terme a la mateixa ubicació que la resta del repositori, utilitzant una nova carpeta per crear la duplicat. 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 duplicat 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.

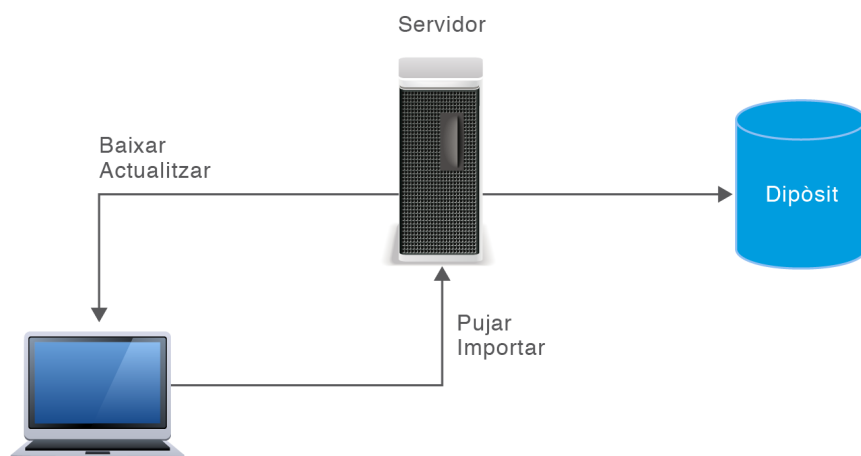
Entre les operacions d'introducció de dades al repositori es poden trobar:

- **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 sistemes 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 Foundation Server

2.5 Dipòsits de les eines de control de versions

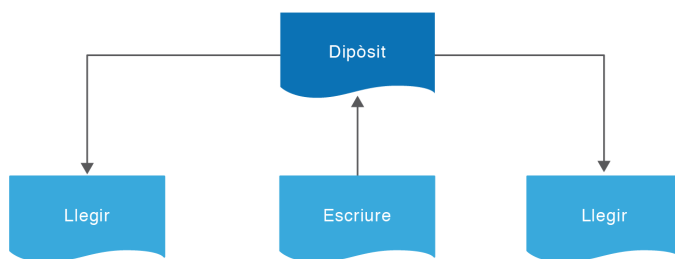
Un **dipòsit** és un magatzem de dades on es guardarà tot allò relacionat amb una aplicació informàtica o unes dades determinades d'un determinat projecte.

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:

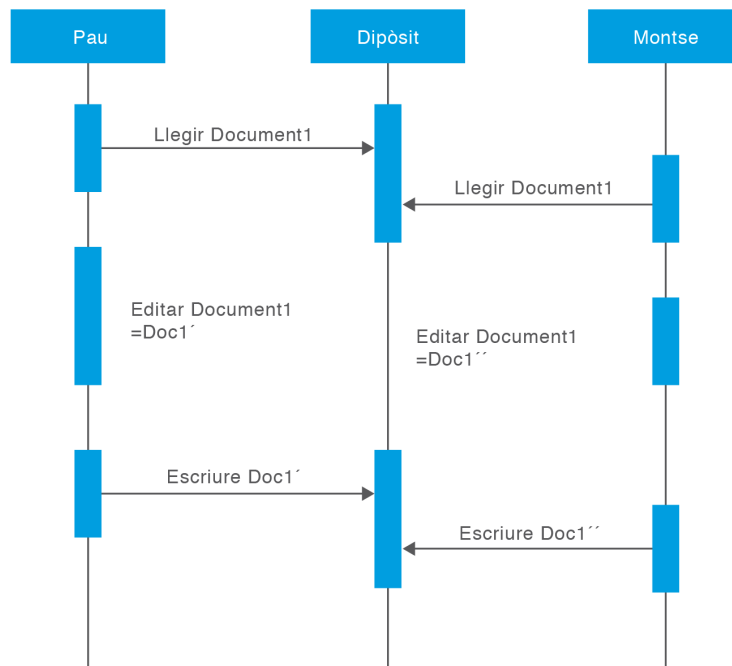
- Compartir arxius per part de diferents programadors
- Bloquejar els arxius utilitzats
- Fusionar els arxius modificats

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, sobre escriure'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.

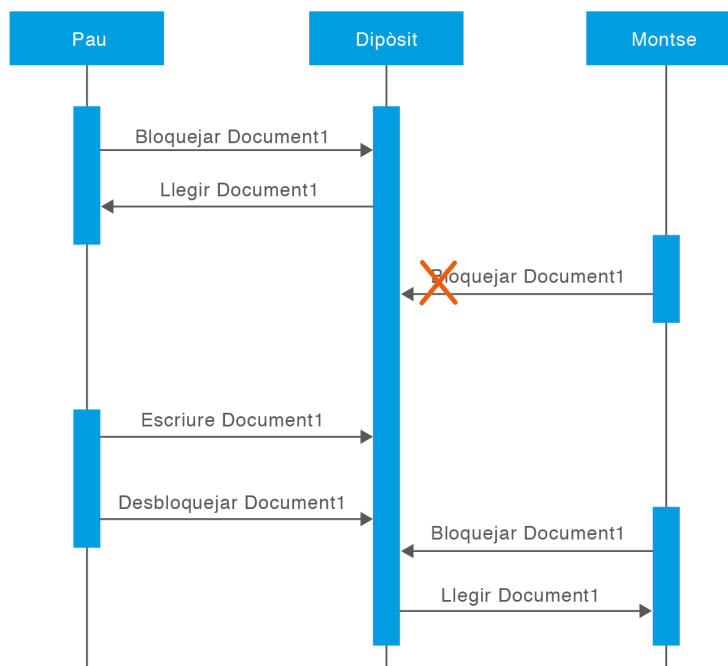
FIGURA 2.10. Cas d'accés a la vegada al mateix arxiu per part de dos usuaris

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

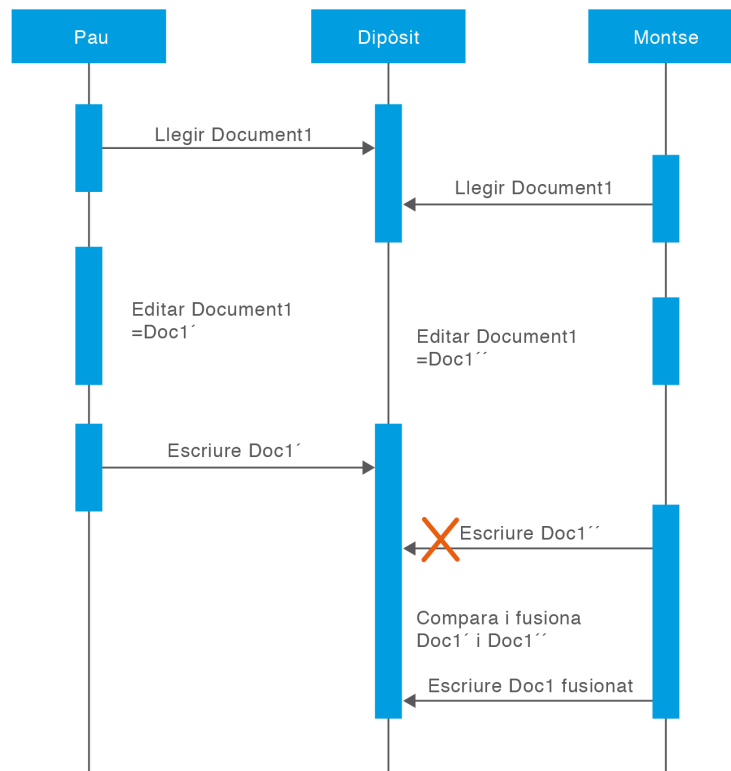
Un altre sistema per solucionar la problemàtica d'accés simultani a un mateix arxiu, com a alternativa al sistema de bloqueig d'arxius, és el que es descriu a continuació:

- 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 efectui 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 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.

Git va ser creat per Linus Torvalds per al desenvolupament del nucli de Linux l'any 2005.

2.6.1 Instal·lació

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:

Git acostuma a estar instal·lat a totes les distribucions de Linux.

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 multi-plataforma (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

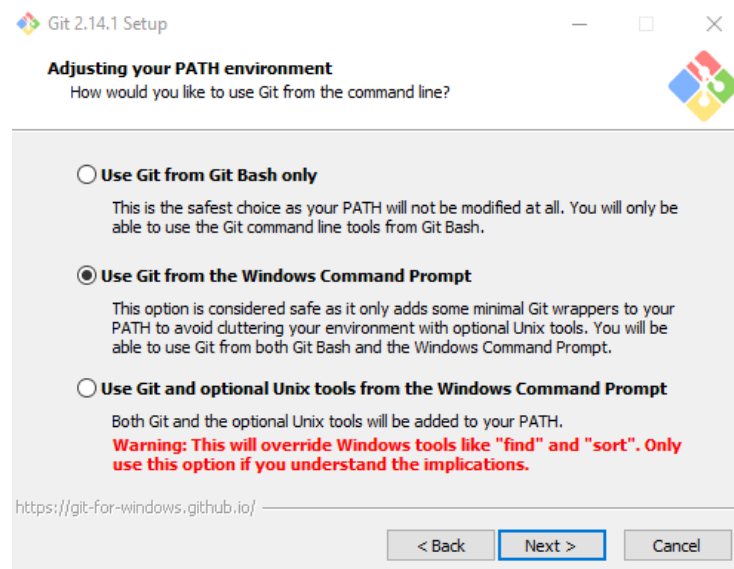


FIGURA 2.14. Selecció de la biblioteca de transport HTTPS

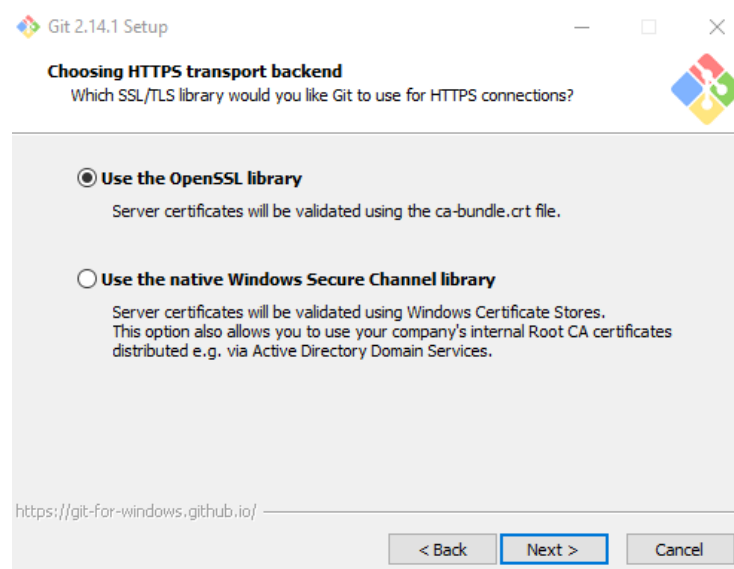
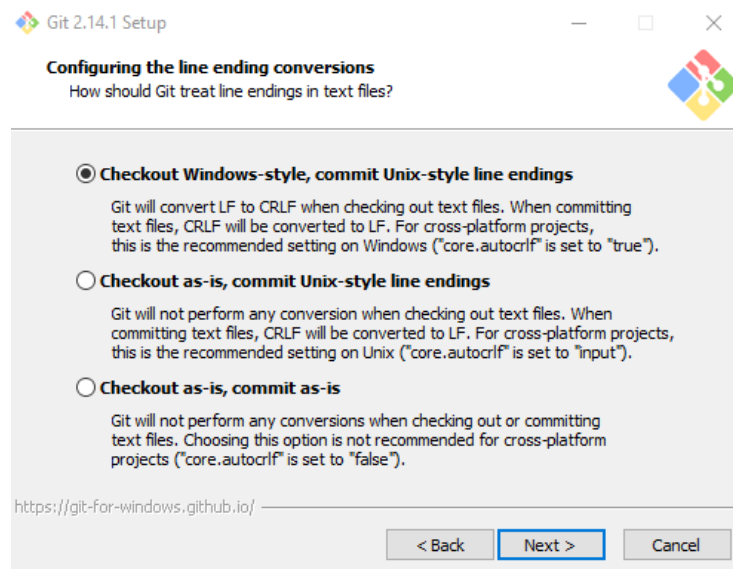


FIGURA 2.15. Configuració del caràcter de final de línia

Un cop instal·lat, obriu la finestra del símbol del sistema (o la terminal, segons el sistema operatiu) i escriviu:

```
1 git --version
```

El resultat ha de ser similar al següent:

```
1 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`:

```
1 md proves-git
```

A Linux i macOS l'ordre serà:

```
1 mkdir proves-git
```

Entreu dintre del directori amb l'ordre següent:

```
1 cd proves-git
```

Inicialitzeu el repositori amb l'opció `init` de Git:

```
1 git init
```

El resultat ha de ser semblant al següent:

```
1 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:

```
1 cd .git
```

El contingut de la carpeta serà similar al següent:

```
1 El volumen de la unidad D es Disco local
2 El número de serie del volumen es: 9222-DEE6
3
4 Directorio de D:\Users\Xavier Garcia\proves-git\.git
5
6 18/08/2017 18:05          130 config
7 18/08/2017 18:05           73 description
8 18/08/2017 18:05          23 HEAD
9 18/08/2017 18:05    <DIR>      hooks
10 18/08/2017 18:05    <DIR>      info
11 18/08/2017 18:05    <DIR>      objects
12 18/08/2017 18:05    <DIR>      refs
13           3 archivos          226 bytes
14           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:

```
1 <h1>Proves Git</h1>
2 <ul>
3   <li>Pas 1: Inicialitzar el repositori: git init</li>
4 </ul>
```

Escriviu a la línia d'ordres:

```
1 git status
```

Aquesta ordre mostra l'estat actual de la còpia de treball. Així, l'ordre anterior us mostrarà un resultat similar al següent:

```
1 On branch master
2
3 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 .
```

Si només voleu afegir els fitxers amb extensió `.html`, l'ordre ha de ser:

```
1 git add *.html
```

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*.

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
```

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"
```

Quan s'inicialitza un projecte és habitual fer una primera pujada amb un comentari similar a "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:

```
1 On branch master
2 nothing to commit, working tree clean
```

Per comprovar que es detecten els canvis correctament canvieu el contingut del fitxer `index.html` pel següent:

```
1 <h1>Proves Git</h1>
2 <ul>
3   <li>Pas 1: Inicialitzar el repositori: git init</li>
4   <li>Pas 2: Afegir tots els fitxers: git add .</li>
5 </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:

```
1 On branch master
2 Changes not staged for commit:
3   (use "git add <file>..." to update what will be committed)
4   (use "git checkout -- <file>..." to discard changes in working directory)
5
6       modified:   index.html
7
8 Untracked files:
9   (use "git add <file>..." to include in what will be committed)
10
11       llegeix.me
12
13 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`:

```
1 git add llegeix.me
```

Pugeu els canvis al repositori:

```
1 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:

```
1 commit 3f9365181b055c0b956e3bdf97a6e3a37085927f (HEAD -> master)
2 Author: Xavier Garcia <xaviergaro.dev@gmail.com>
3 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*:

```
1 git branch nova-branca
```

No es visualitza cap canvi, però si entreu l'ordre `git branch` veureu la llista de branques al repositori:

```
1 * master
2   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:

```
1 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*:

```
1 master
2 * 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:

```
1 <h1>Proves Git</h1>
2 <ul>
3   <li>Pas 1: Inicialitzar el repositori: git init</li>
4   <li>Pas 2: Afegir tots els fitxers: git add .</li>
```

```

5 <li>Pas 4: Crear una branca: git branch nova-branca</li>
6 </ul>

```

Seguidament pugeu els canvis al repositori amb l'ordre:

```

1 git commit -m . "Afegit pas 4"

```

Apareix un missatge similar al següent:

```

1 [nova-branca b4483e3] Afegit pas 4
2 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:

```

1 <h1>Proves Git</h1>
2 <ul>
3   <li>Pas 1: Inicialitzar el repositori: git init</li>
4   <li>Pas 2: Afegir tots els fitxers: git add .</li>
5   <li>Pas 3: Pujar els canvis al repositori: git commit . -m "Pujar canvis"</li>
6 </ul>

```

Seguidament pugeu els canvis al repositori amb l'ordre:

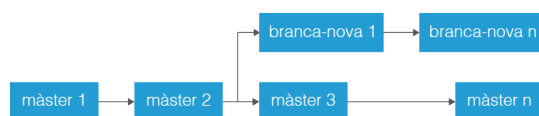
```

1 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):

```

1 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:

```

1 diff --cc index.html
2 index ae389e4,e2e54dd..0000000
3 --- a/index.html
4 +++ b/index.html
5 @@@ -2,5 -2,5 +2,9 @@@
6     <ul>
7         <li>Pas 1: Inicialitzar el repositori: git init</li>
8         <li>Pas 2: Afegir tots els fitxers: git add .</li>
9     +++<<<<<< HEAD
10 +   <li>Pas 3: Pujar els canvis al repositori: git commit . -m "Pujar canvis"</li>
11     ++++++
12 +   <li>Pas 4: Crear una branca: git branch nova-branca</li>
13     +++>>>>>> nova-branca
14     </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:

```

1 <h1>Proves Git</h1>
2 <ul>
3     <li>Pas 1: Inicialitzar el repositori: git init</li>
4     <li>Pas 2: Afegir tots els fitxers: git add .</li>
5     <li>Pas 3: Pujar els canvis al repositori: git commit . -m "Pujar canvis"</li>
6     <li>Pas 4: Crear una branca: git branch nova-branca</li>
7 </ul>

```

Una vegada desats els canvis, pugeu-lo al repositori amb l'ordre:

```

1 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:

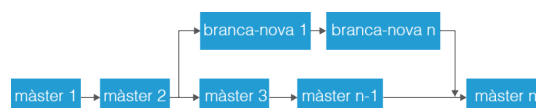
```

1 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:

```
1 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`):

```
1 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:

```
1 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:

```
1 tag v1.0
2 Tagger: Xavier Garcia <xaviergaro.dev@gmail.com>
3 Date:   Fri Aug 18 21:06:17 2017 +0200
4
5 Primera versió
6
7 commit 7a7edead12a3f7f33e1bebc692cc868fa534f18f (HEAD -> master, tag: v1.0)
8 Merge: 3f2e774 b4483e3
9 Author: Xavier Garcia <xaviergaro.dev@gmail.com>
10 Date:   Fri Aug 18 20:39:31 2017 +0200
11
12 Conflicte resolt
```

```
13 diff --cc index.html
14 index ae389e4,e2e54dd..31d9d62
15 --- a/index.html
16 +++ b/index.html
17 @@@ -2,5 -2,5 +2,6 @@@
18 <ul>
19 <li>Pas 1: Inicialitzar el repositori: git init</li>
20 <li>Pas 2: Afegir tots els fitxers: git add .</li>
21 + <li>Pas 3: Pujar els canvis al repositori: git commit . -m "Pujar canvis"</li>
22 + <li>Pas 4: Crear una branca: git branch nova-branca</li>
23 </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:

```
1 git push origin xat
```

Per fer un *pull* només cal indicar el nom curt del repositori remot:

```
1 git pull xat
```

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:

```
1 # Diaris
2 logs
3 *.log
4
5 # Dependències
6 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.

Algunes aplicacions inclouen plantilles per generar els fitxers `.gitignore` amb la selecció de fitxers i directoris més habituals segons el llenguatge utilitzat.

Podeu trobar la documentació completa sobre ignorar fitxers o directoris al següent enllaç: goo.gl/CTkwC7.

- **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

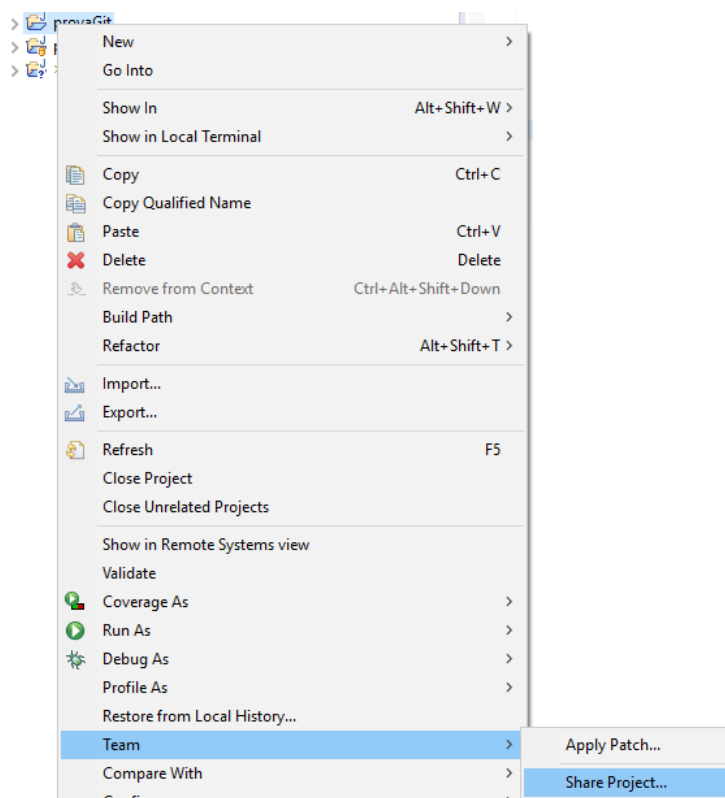
Tots els entorns de desenvolupament integrats (i alguns editors de text per a desenvolupadors) inclouen l'opció de gestionar el control de versions dintre del mateix programa. En aquesta secció es descriu com integra Eclipse el control de versions amb Git.

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

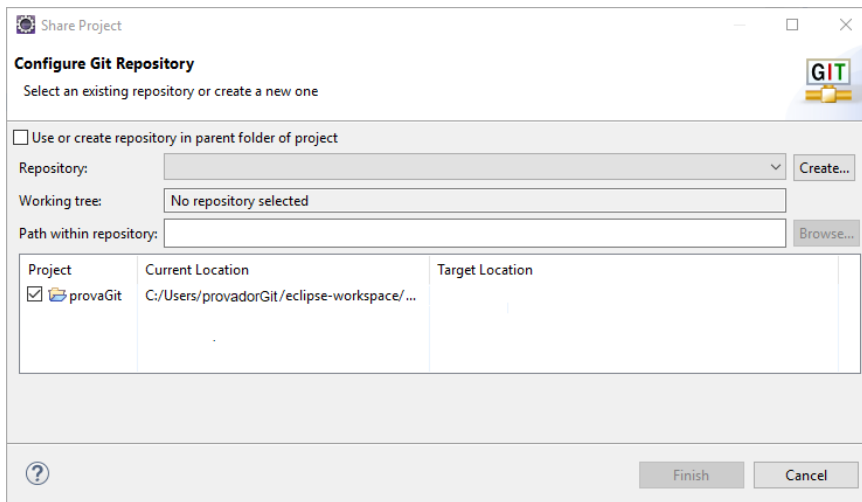
A Eclipse (amb el connector EGit), per inicialitzar un repositori cal fer clic amb el botó secundari a sobre de la carpeta que representa el projecte que volem introduir al repositori i seleccionar les opcions *Team / Share Project...*, com es veu a la figura 2.18.

FIGURA 2.18. Opció 'Share'



Ens apareixerà la finestra que es mostra a la figura 2.19.

FIGURA 2.19. Configuració del repositori

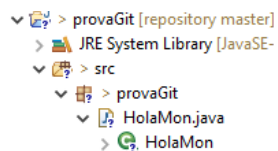


En aquest cas no hi ha cap repositori ja definit i n'hem de crear un de nou. Si ja n'existís un, podríem seleccionar-lo. Per crear el nou repositori, clicarem al botó *Create...*. Ens sortirà una finestra per demanar el directori o carpeta on anirà el repositori que estem creant. Caldrà completar el 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.

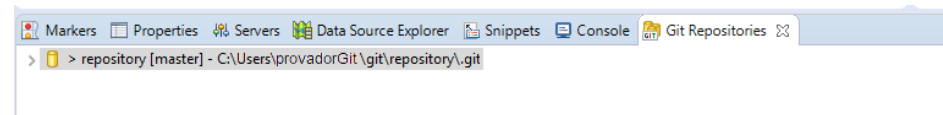
L'opció "Use or create repository in parent folder of project" que apareix a la finestra de la figura 2.19 i que no hem seleccionat força la creació del repositori a la mateixa carpeta del nostre projecte, però té l'inconvenient que ja no es podrien afegir més projectes al repositori així creat.

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



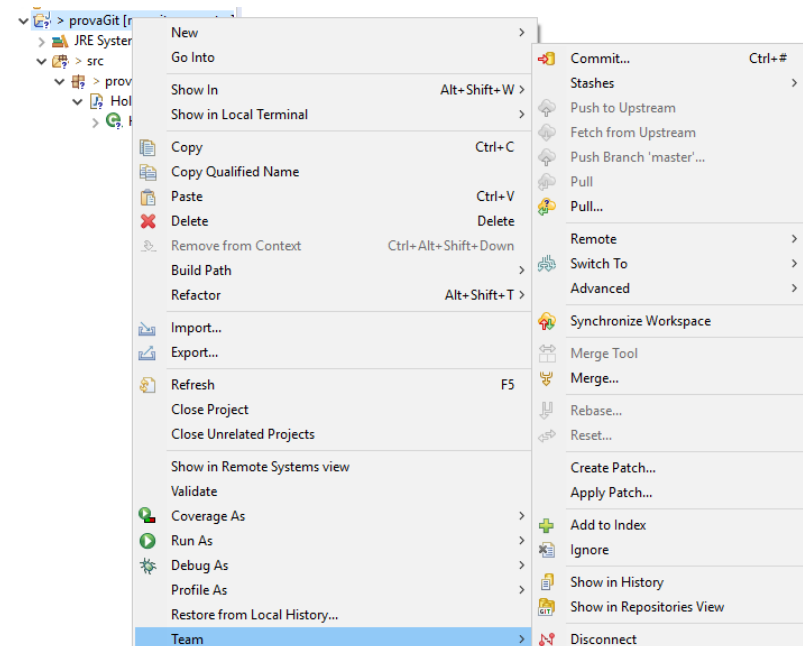
També ens ha d'aparèixer la finestra amb els repositoris Git (habitualment a la zona inferior dreta de la finestra), tal com es mostra a la figura 2.21.

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*.

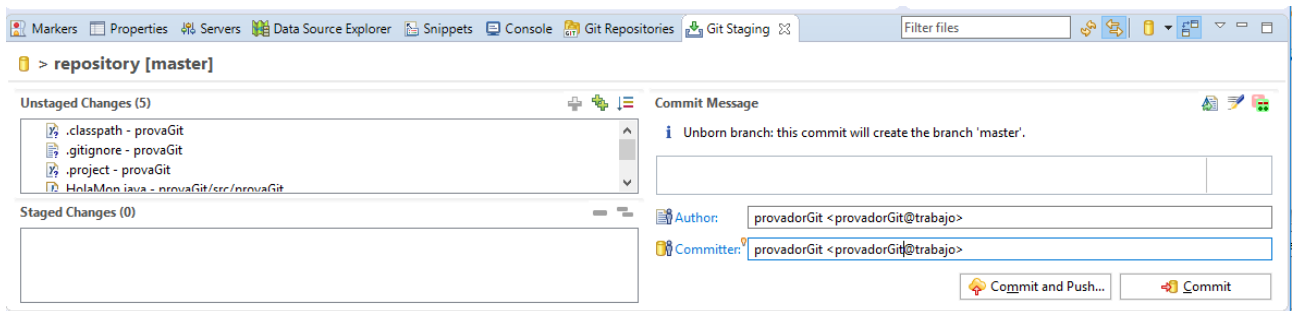
Gestió bàsica del repositori

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'

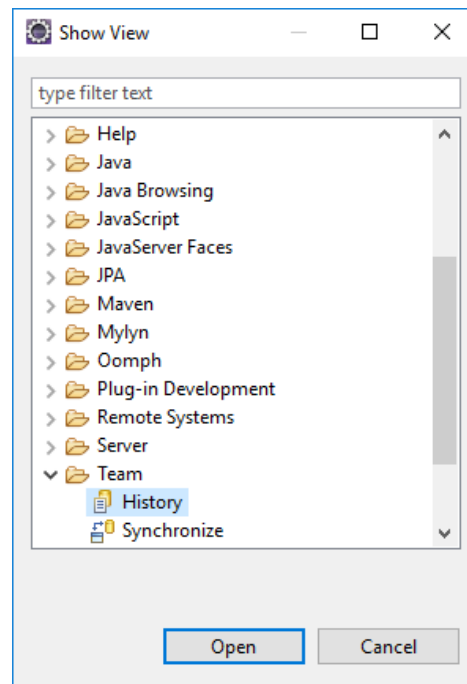
En aquesta finestra, a la llista *Unstaged Changes*, apareixen tots els fitxers que no volem confirmar. Inicialment hi són tots els fitxers modificats des del darrer *commit* o tots els fitxers si no hem fet encara cap *commit* des de la creació del repositori. D'altra banda, a la llista *Staged Changes* 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 *Staged changes* aquest interrogant ha estat substituït per una creu.

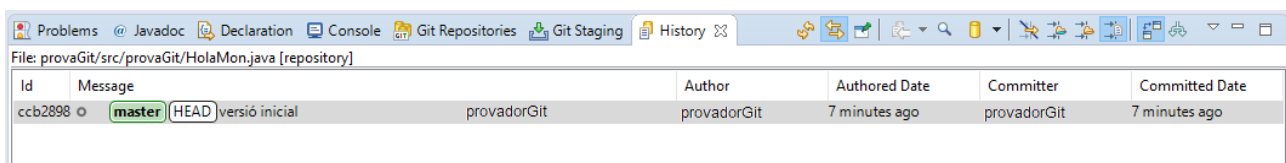
En aquest punt, abans de fer el *commit*, ja només resta escriure el comentari associat a aquesta operació. S'escriu al quadre de text de sota de l'etiqueta *Commit message*. Per exemple, podem escriure "versió inicial" com a comentari.

Ara podem clicar al botó *Commit* per acabar de realitzar aquesta operació. Després del *commit* veureu, a l'explorador dels projectes, que s'han substituït 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.

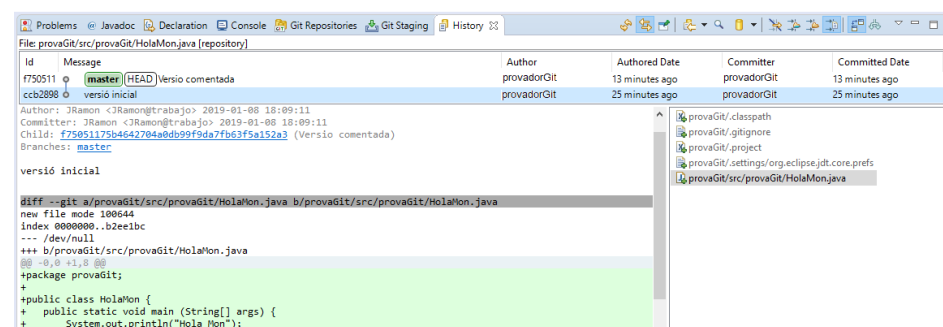
Una altra operació de Git que ens permet realitzar Eclipse és consultar l'historial de les versions que hem confirmat del nostre projecte. Per fer-ho, cal seleccionar *History* a la finestra que s'obre amb l'opció del menú *Window / Show view / Other...*, tal com es veu a la figura 2.24.

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’n 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

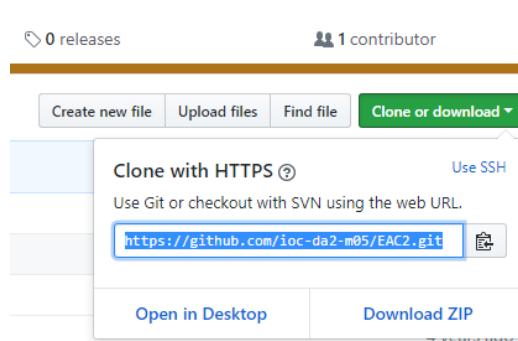
Habitualment voldrem treballar des d'Eclipse amb un repositori remot. Les operacions principals a fer amb ell són *pull* i *push*. Poden realitzar-se en qualsevol ordre, segons convingui al desenvolupador.

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

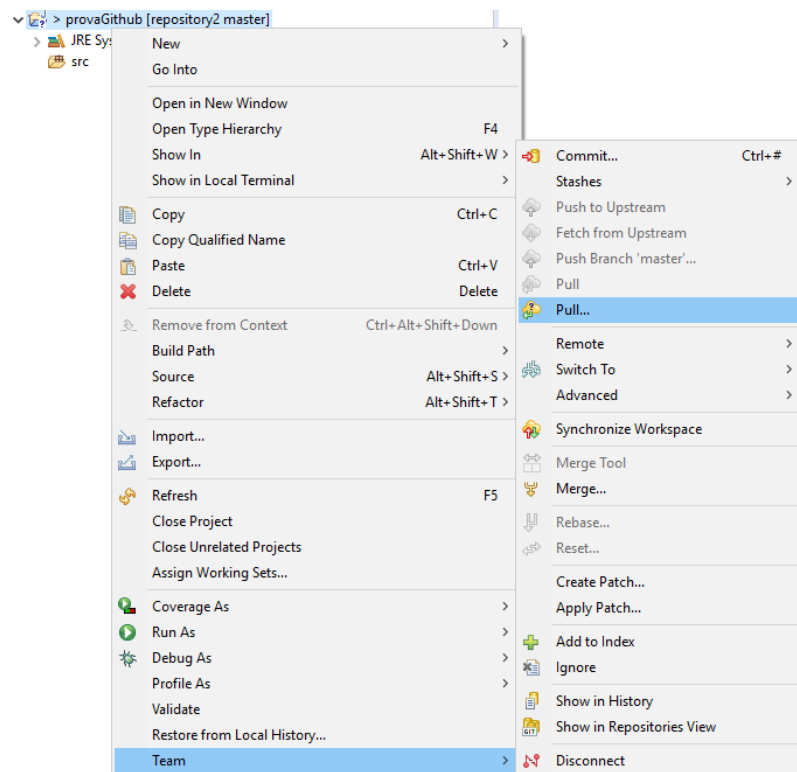
Per treballar amb el repositori remot és necessari conèixer el seu URI. Al cas de GitHub, s'obté seleccionant el repositori i clicant el botó *Clone or download*, que apareix a la part superior dreta de la finestra. A la figura 2.27 s'ha accedit a l'URI i s'ha seleccionat, per exemple, per copiar-la al porta-retalls.

FIGURA 2.27. URI de GitHub

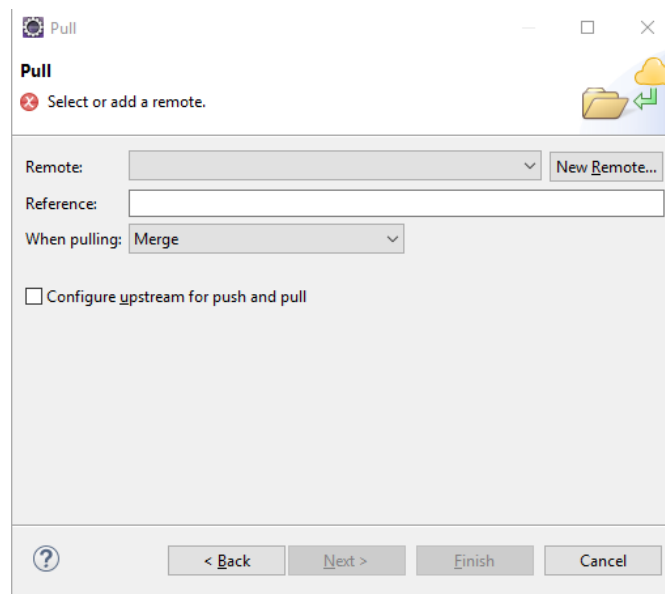


Operació pull

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

Add Remote

Destination Git Repository

Enter the location of the destination repository.

Remote name:

Location

URI:

Host:

Repository path:

Connection

Protocol:

Port:

Authentication

User:

Password:

☐ Store in Secure Store

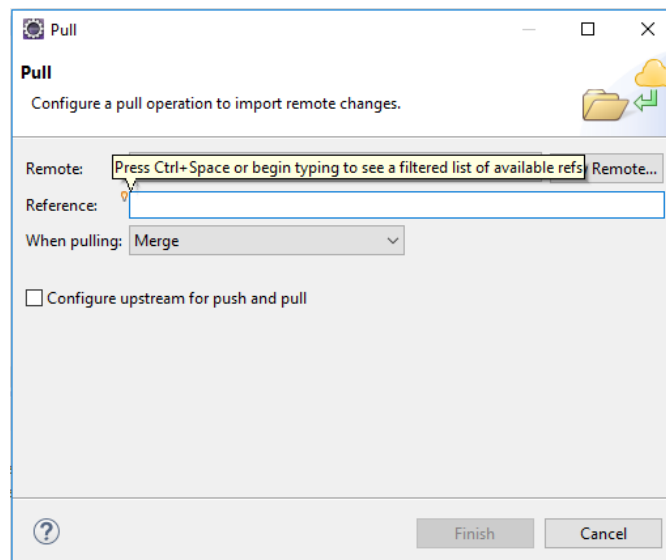
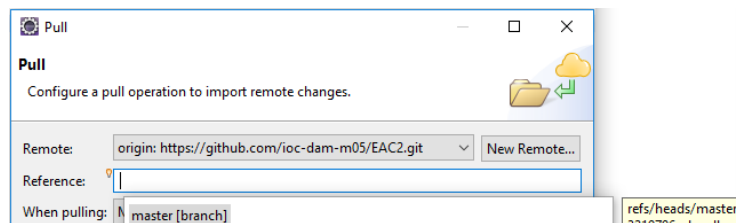
Cal omplir:

- *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.

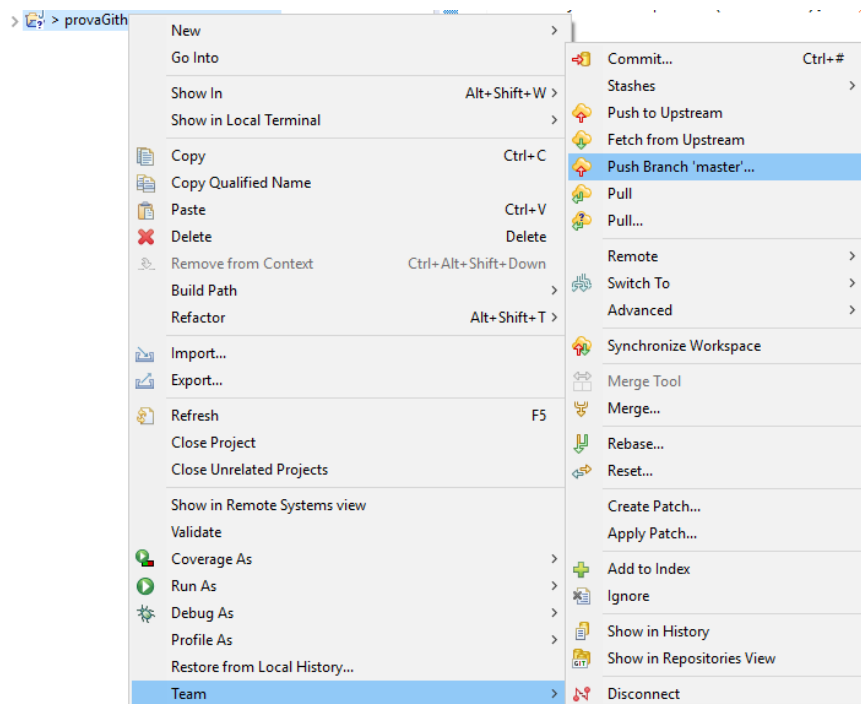
FIGURA 2.31. Camp 'Reference' a la finestra 'Pull'**FIGURA 2.32.** Desplegable de 'Reference' a la finestra 'Pull'

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 .

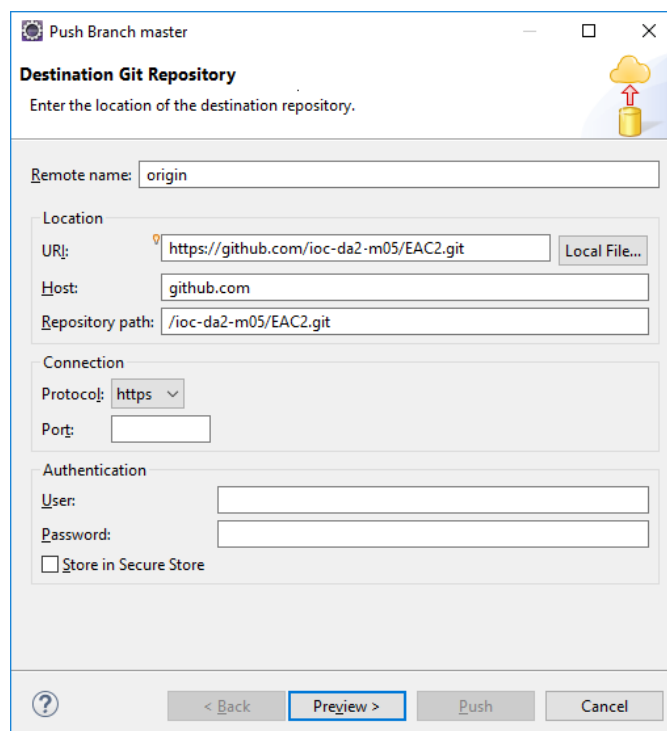
Operació push

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 comentos simples, com es veu a la figura 2.33. Aquesta branca és la que es tindrà en compte per fer el *push*.

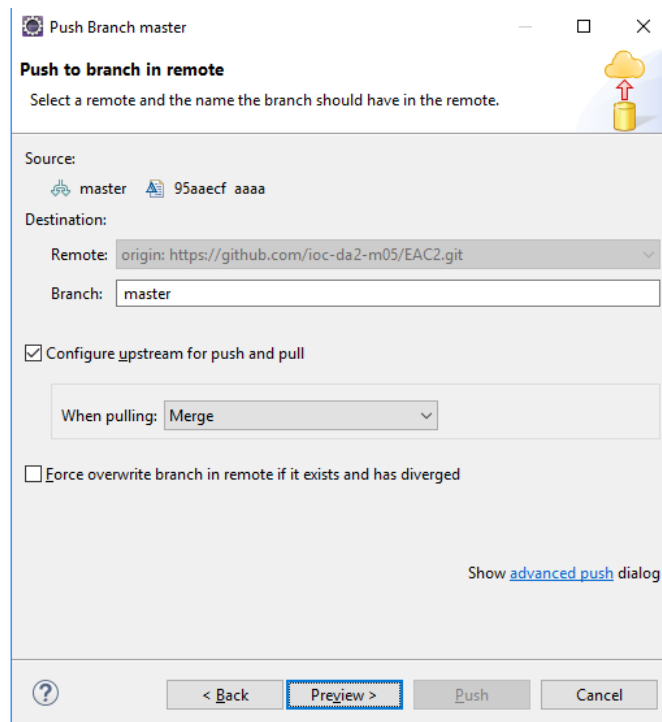
FIGURA 2.33. Menú 'Push'

S'obrirà una finestra semblant a la que es mostra a la figura 2.34.

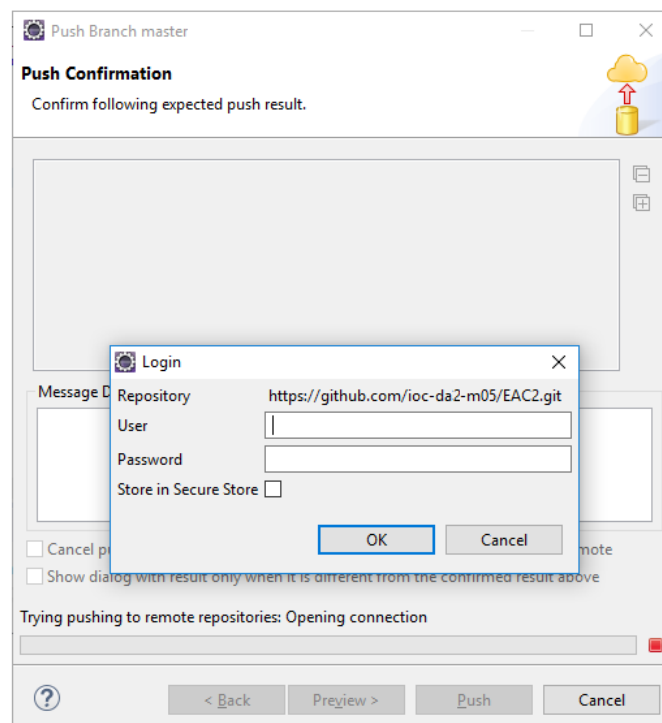
FIGURA 2.34. Finestra amb el repositori destí

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.

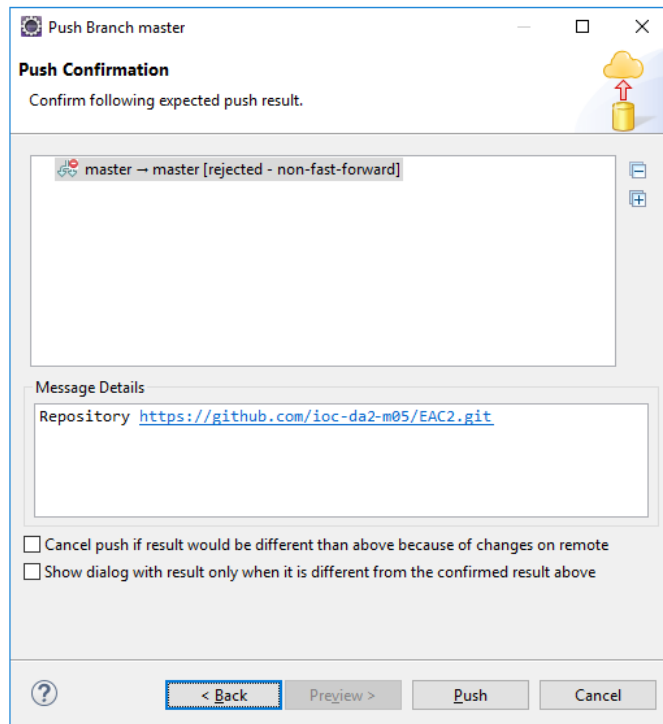
FIGURA 2.35. Finestra de selecció de la branca

Podem deixar els valors per defecte i fer clic novament al botó *Preview >*. Ens apareixerà una pantalla com la de la figura 2.36.

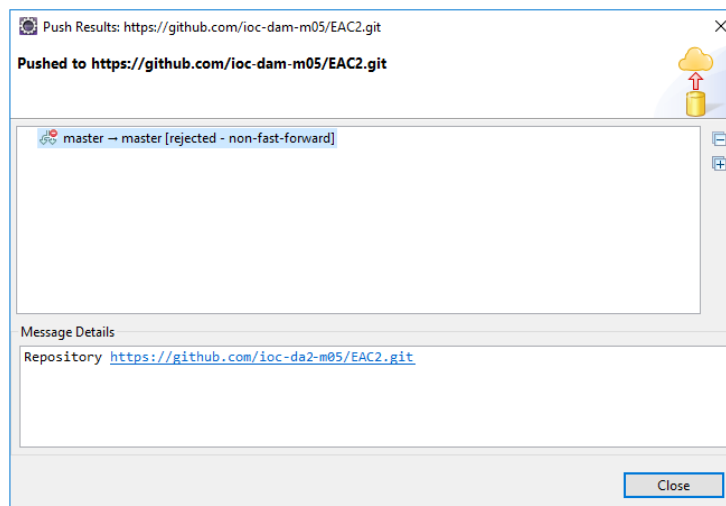
FIGURA 2.36. Usuari i contrasenya del repositori remot

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.

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

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.

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.

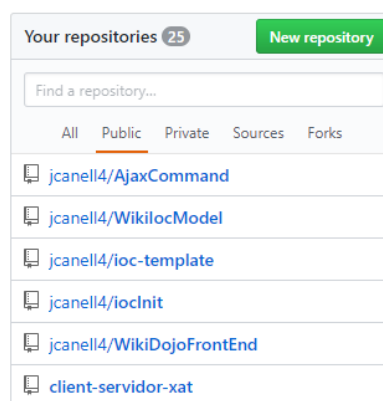
Per treballar amb GitHub es pot utilitzar la línia d'ordres de Git, es pot descarregar algun client gràfic com SourceTree o Github Desktop (desktop.github.com) o es pot treballar directament des d'un IDE integrat amb Git.

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 repository* (vegeu la figura 2.39).

FIGURA 2.39. Llistat de repositoris propis

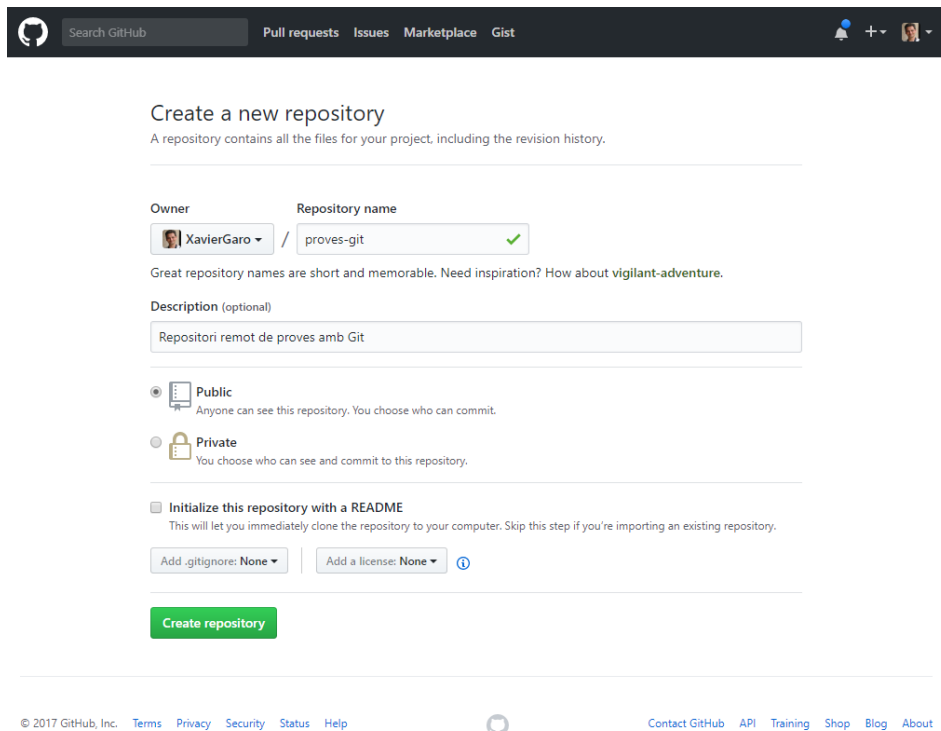


Seguidament heu d'indicar el nom del repositori, la descripció i el tipus (públic o privat), com en la figura 2.40. Addicionalment podeu inicialitzar amb un fitxer README, que es mostra a la primera pàgina del repositori.

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:

```
1 echo "# proves-git" >> README.md
2 git init
3 git add README.md
4 git commit -m "first commit"
5 git remote add origin https://github.com/nom-usuari/nom-repositori.git
6 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:

```
1 git remote add origin https://github.com/nom-usuari/nom-repositori.git
2 git push -u origin master
```

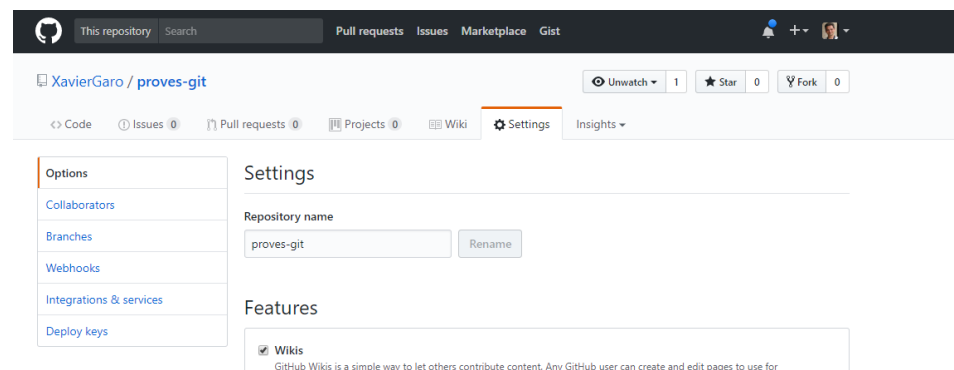
Fixeu-vos que l'URL del repositori que mostra GitHub és la del repositori que heu creat al vostre compte, així que podeu copiar directament el codi a la línia d'ordres.

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.

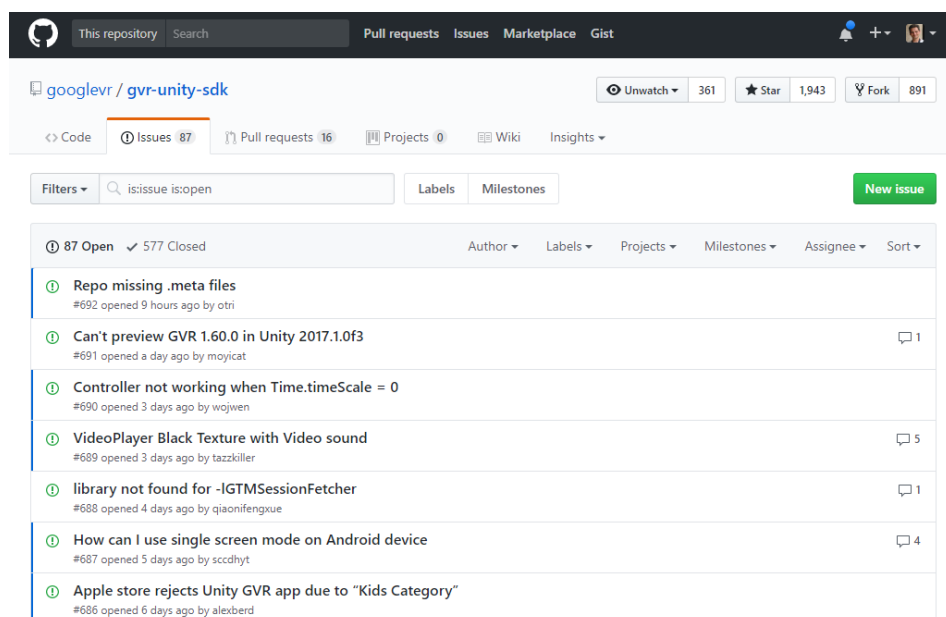
2.7.3 Gestió d'errors ('issues')

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).

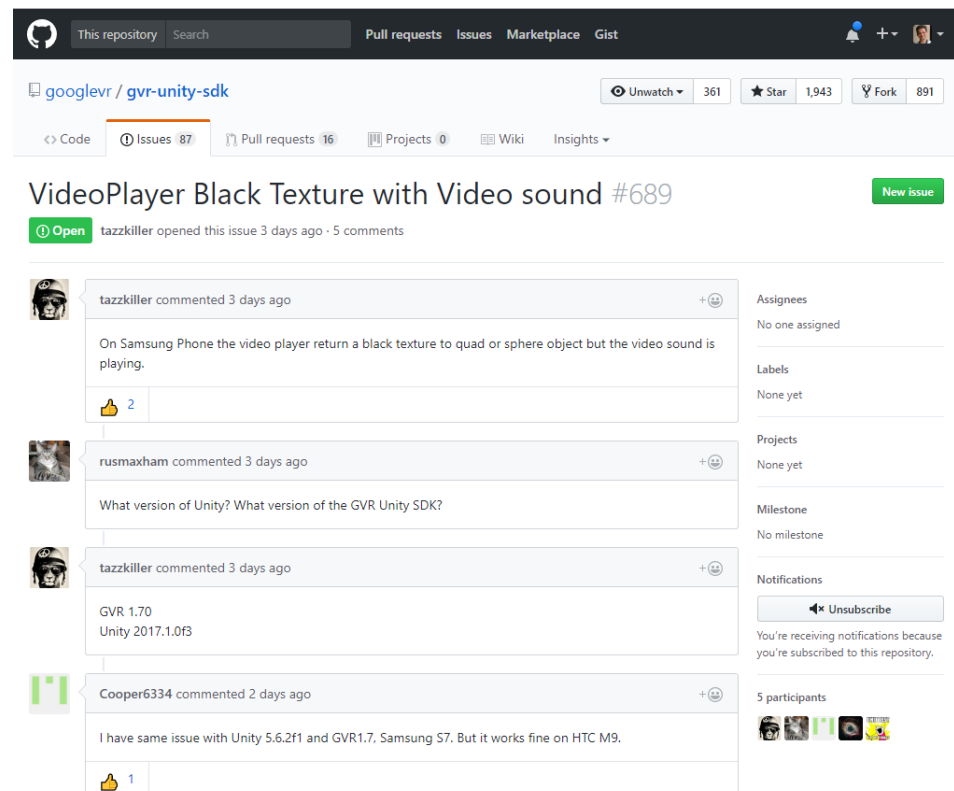
FIGURA 2.42. Llista d'errors i característiques d'un repositori de GitHub



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'escriu 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
2     especificat té 8 caràcters i fals en cas contrari */
3  function validarDNI (DNI){
4      if (DNI.value != ""){                                //Valida que la variable DNI
5                                                             tingui algun valor.
6      if (DNI.value.length < 8 ) {                          //Valida la longitud del DNI.
7          alert ("El DNI no és correcte");                 //Mostra per pantalla un
8                                                             missatge d'error.
9          DNI.focus();                                     //Posiciona el cursor a la
10                                                             variable DNI.
11          return false;                                    //La funció retorna un false
12                                                             indicant que el DNI no és vàlid.
13      }
14  }
15  return true;                                             //La funció retorna un true
16                                                             indicant que el DNI és vàlid.
17 }
```

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:

```
1  /** Comentari de JavaDoc
2  * De forma automàtica es generarà una pàgina HTML amb les comentaris
   especificats en el codi.
3  * @etiqueta1 text específic de l'etiqueta1
4  * @etiqueta2 text específic de l'etiqueta2
5  */
```

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 servir 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:

```
1  /**
2   *Exemple de descripció general d'un comentari <u>JavaDoc.</u>
3   * @ Author Programador primer del projecte
4   */
```

- 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.

```
1  /**
2   * Classe que calcula el factorial d'un nombre.
3   * @author IOC
4   * @version 2012
5   */
6  public class Factorial {
7
8      /**
9       * Calcula el factorial de n.
10      *  $n! = n * (n-1) * (n-2) * (n-3) * \dots * 1$ 
11      * @param n és el número al que es calcularà el factorial.
12      * @return n! és el resultat del factorial de n
13      */
14      public static double factorial (double n) {
15
16          if (n==0)
17              return 1;
18          else
19          {
20              double resultat = n * factorial(n-1);
21              return resultat;
22          }
23      }
24  }
```

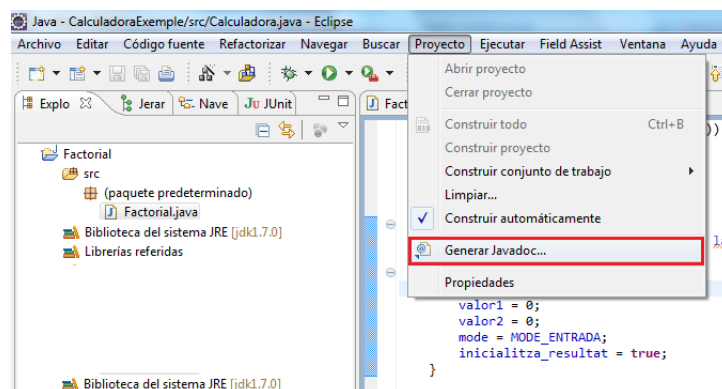
2.8.3 Exemple d'utilització de JavaDoc amb Eclipse

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.

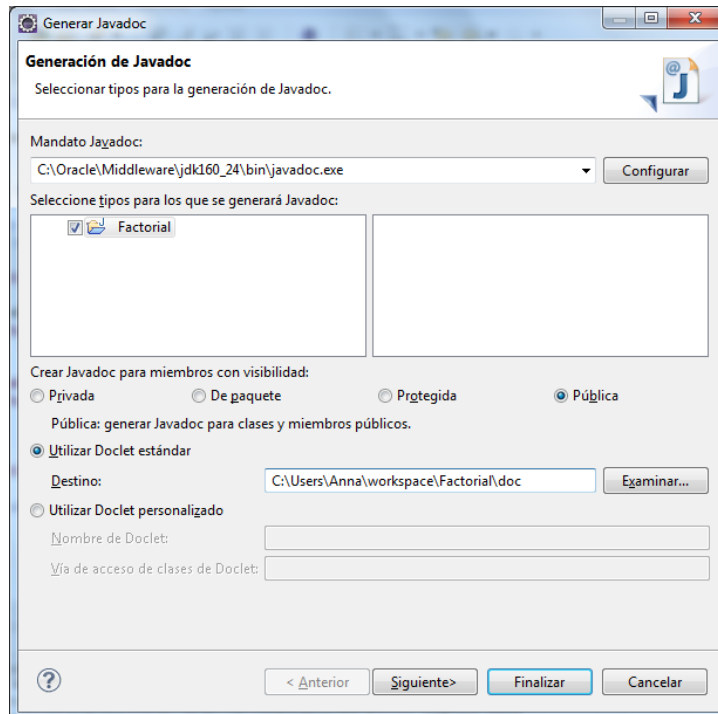
Una vegada desenvolupat un projecte en Java i afegits els comentaris amb les estructures i estàndards JavaDoc, es podrà generar la documentació de forma automàtica utilitzant la funcionalitat *Generar Javadoc* del menú *Projecte*, com es pot veure a la figura 2.44. Aquesta utilitat és part del Java Development Kit.

FIGURA 2.44. Menú 'Projecte' - 'Generar JavaDoc'



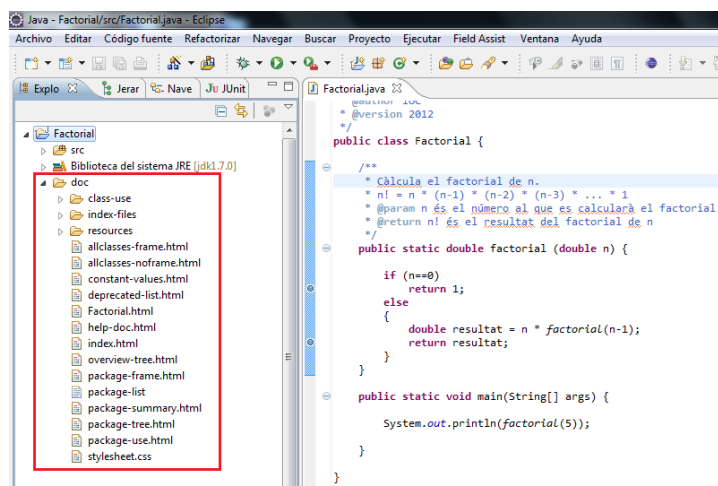
A continuació, obtindrà un diàleg que demanarà la ubicació de l'arxiu Javadoc.exe i una sèrie d'opcions de la creació de la documentació, com la carpeta destinació dels arxius que es crearan o el tipus de classes de les quals s'haurà de crear la documentació. Aquest diàleg es pot observar a la figura 2.45.

FIGURA 2.45. Diàleg per a la creació de la documentació a partir de Javadoc

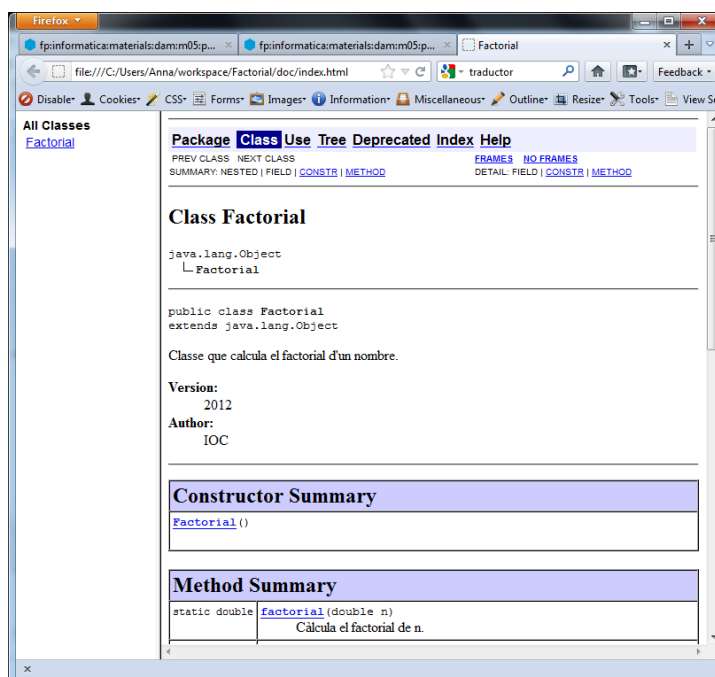


En fer clic sobre el botó *Finalitzar*, automàticament s'ha generat la documentació en format HTML, com es pot observar a la figura 2.46.

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