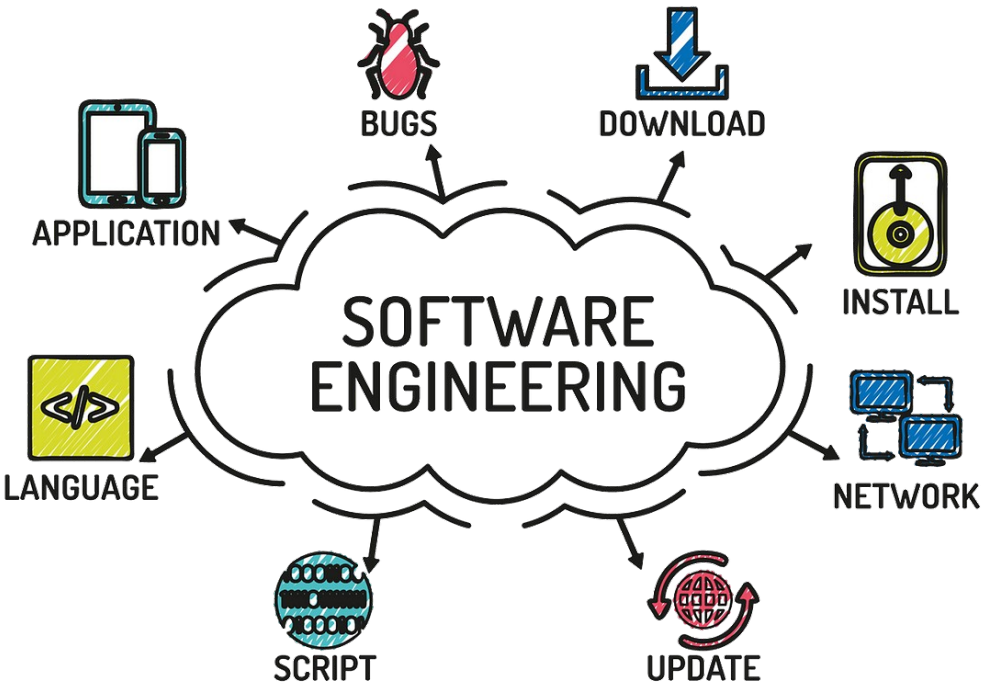
CFGS- Desenvolupament d'Aplicacions Web

Mòdul professional: *Entorns de Desenrotllament*
Codi: *0487*
Durada: *96 hores*



índex del llibre

Tema 1: Programació

---

1. Concepte de programa informàtic.
2. Codi font, codi objecte i codi executable. Màquines virtuals.
3. Classificació dels llenguatges de programació.
4. Paradigmes de programació
5. Característiques dels llenguatges més difosos.
6. Fases del desenrotllament d'una aplicació
7. Estructures d'equips de treball en desenvolupament web
8. Procés d'obtenció de codi executable a partir del codi font.

---

1. Concepte de programa informàtic.

Un programa informàtic o programa d'ordinador és una seqüència d'instruccions, escrites per realitzar una tasca específica en un ordinador. Aquest dispositiu requereix programes per funcionar, en general, executant les instruccions del programa en un processador central. El programa té un format executable que l'ordinador pot utilitzar directament per executar les instruccions. El mateix programa en el seu format de codi font llegible per a humans, del qual es deriven els programes executables (per exemple, compilats), permet a un programador estudiar i desenvolupar els seus algorismes.

Generalment, el codi font l'escriuen professionals coneguts com a programadors. Aquest codi s'escriu en un llenguatge de programació que segueix un dels dos paradigmes següents: imperatiu o declaratiu, i que posteriorment pot ser convertit en un arxiu executable (usualment anomenat programa executable o un binari) per un compilador i més tard executat per una unitat central de processament. D'altra banda, els programes d'ordinador es poden executar amb l'ajuda d'un intèrpret, o poden ser encastats directament en el maquinari.

2. Codi font, codi objecte i codi executable. Màquines virtuals.

El codi font d'un programa informàtic és un conjunt de línies de text amb els passos que ha de seguir l'ordinador per executar un programa.

El codi font d'un programa està escrit per un programador en algun llenguatge de programació llegible per humans, normalment en forma de text pla. No obstant això, el programa escrit en un llenguatge llegible per humans no és directament executable per l'ordinador, sinó que ha de ser traduït a un altre llenguatge o codi binari, així serà més fàcil per a la màquina interpretar-ho (llenguatge màquina o codi objecte que sí que podrà ser executat pel maquinari de l'ordinador). Per a aquesta traducció s'usen els anomenats compiladors, assembladors, intèrprets i altres sistemes de traducció.

Pel que fa al codi objecte, és un pas intermedi ente codi font i l'executable. Li cal encara l'enllaçat amb llibreries (açò ho farà el linker) per a generar l'executable.

El terme codi font també s'usa per a fer referència al codi font d'altres elements del programari, com ara el codi font d'una pàgina web, que està escrit en llenguatge de marcat HTML o Javascript, o altres llenguatges de programació web , i que és posteriorment executat pel navegador web per visualitzar aquesta pàgina quan és visitada.

L'àrea de la informàtica que es dedica a la creació de programes, i per tant a la creació del codi font, és l'enginyeria de programari.

Pel que fa a les màquines virtuals, la més coneguda per tots es la de Java. Una màquina virtual Java (JVM) és una màquina virtual de procés nadiu, és a dir, executable en una plataforma específica, capaç d'interpretar i executar instruccions expressades en un codi binari especial (el bytecode Java), el qual és generat pel compilador del llenguatge Java.

3. Classificació dels llenguatges de programació.

Els llenguatges de programació es poden classificar de diverses maneres segons diferents criteris. A continuació, es presenten algunes de les classificacions més comunes:

1. Segons el nivell d'abstracció

Llenguatges de Baix Nivell

- **Llenguatge Màquina**: Codi binari que la màquina pot executar directament. És específic per a cada tipus de processador.
- **Llenguatge d'Assemblador**: Utilitza mnemònics per representar les instruccions del llenguatge màquina. És més llegible que el codi binari, però encara està molt a prop del maquinari.

Llenguatges d'Alt Nivell

- **Exemples**: C, C++, Java, Python, Ruby, etc.
- **Característiques**: Abstracció més gran del maquinari, més fàcils de llegir i escriure per als humans, independents de la màquina.

2. Segons el paradigma de programació

Llenguatges Imperatius

- **Característiques**: Especifica una seqüència d'operacions per executar.
- **Exemples**: C, C++, Java, Python (en part).

Llenguatges Orientats a Objectes

- **Característiques**: Basats en objectes que encapsulen dades i comportaments.
- **Exemples**: Java, C++, Python, Ruby.

Llenguatges Funcionals

- **Característiques**: Basats en funcions matemàtiques i evitació d'estats mutables.
- **Exemples**: Haskell, Lisp, Erlang, Scala.

Llenguatges Lògics

- **Característiques**: Basats en regles lògiques i inferències.
- **Exemples**: Prolog.

Llenguatges de Scripting

- **Característiques**: Generalment interpretats, utilitzats per automatitzar tasques.
- **Exemples**: Python, JavaScript, Perl, Ruby.

3. Segons la forma d'execució

Llenguatges Compilats

- **Característiques**: El codi font es compila a codi màquina abans de l'execució.

- **Exemples**: C, C++, Rust.

Llenguatges Interpretats

- **Característiques**: El codi es tradueix i s'executa línia per línia durant l'execució.
- **Exemples**: Python, JavaScript, Ruby.

Llenguatges Semi-compilats

- **Característiques**: El codi font es compila a un codi intermedi, que després és interpretat o compilat a codi màquina.
- **Exemples**: Java (compilat a bytecode, executat per la JVM), C# (compilat a IL, executat per .NET runtime).

4. Segons la generació

Llenguatges de Primera Generació (1GL)

- **Exemples**: Llenguatge màquina.

Llenguatges de Segona Generació (2GL)

- **Exemples**: Assemblador.

Llenguatges de Tercera Generació (3GL)

- **Exemples**: C, C++, Java, Fortran, Python.

Llenguatges de Quarta Generació (4GL)

- **Característiques**: Orientats a resultats, sovint utilitzats per a gestió de bases de dades, informes, i aplicacions empresarials.
- **Exemples**: SQL, MATLAB, R.

Llenguatges de Cinquena Generació (5GL)

- **Característiques**: Basats en la resolució de problemes mitjançant restriccions i lògica.
- **Exemples**: Prolog, alguns llenguatges d'intel·ligència artificial.

5. Segons el tipus de tipatge

Llenguatges amb Tipatge Estàtic

- **Característiques**: Els tipus de dades es verifiquen en temps de compilació.
- **Exemples**: Java, C, C++.

Llenguatges amb Tipatge Dinàmic

- **Característiques**: Els tipus de dades es verifiquen en temps d'execució.
- **Exemples**: Python, Ruby, JavaScript.

Llenguatges amb Tipatge Fort

- **Característiques**: Es realitzen poques o cap conversió implícita entre tipus.
- **Exemples**: Haskell, Java, Python (en part).

Llenguatges amb Tipatge Feble

- **Característiques**: Permeten conversions implícites entre tipus.
- **Exemples**: JavaScript, PHP, Perl.

6. Segons l'àmbit d'aplicació

Llenguatges de Propòsit General

- **Exemples**: Python, Java, C++.

Llenguatges de Propòsit Específic

- **Exemples**: SQL (gestió de bases de dades), HTML/CSS (disseny web), MATLAB (computació numèrica).

En resum, els llenguatges de programació es classifiquen segons diversos criteris com el nivell d'abstracció, el paradigma de programació, la forma d'execució, la generació, el tipus de tipatge, i l'àmbit d'aplicació. Aquestes classificacions ajuden a entendre millor les característiques i els usos adequats de cada llenguatge en diferents contextos de desenvolupament.

4. Paradigmes de programació

Els paradigmes de programació són diferents estils o enfocaments per escriure programes informàtics. Aquests paradigmes defineixen les tècniques i metodologies utilitzades per estructurar i executar el codi. A continuació, es presenten els paradigmes de programació més importants, juntament amb les seves característiques i exemples de llenguatges que els utilitzen.

1. Paradigma Imperatiu

El paradigma imperatiu es basa en la noció de donar instruccions explícites a l'ordinador per canviar l'estat del programa a través de seqüències d'operacions.

- **Característiques**:
    - Utilitza declaracions per canviar l'estat del programa.
    - Seqüència clara de passos per arribar a un resultat.
    - Permet un control directe sobre el flux d'execució (bucles, condicions, etc.).
- **Exemples de Llenguatges**: C, C++, Java, Python (en part).

2. Paradigma Funcional

El paradigma funcional es basa en el concepte de funcions matemàtiques. Els programes es construeixen mitjançant la composició de funcions pures, sense estat i sense efectes laterals.

- **Característiques**:
    - Evita l'ús de variables mutables.
    - Enfoc en funcions pures (que no tenen efectes col·laterals).
    - Utilitza recursió en lloc de bucles.
- **Exemples de Llenguatges**: Haskell, Lisp, Erlang, Scala, F#.

3. Paradigma Orientat a Objectes (OOP)

El paradigma orientat a objectes es basa en la noció de "objectes", que són instàncies de classes, i encapsulen dades i comportaments relacionats.

- **Característiques**:
    - Encapsulament: Agrupació de dades i mètodes que operen sobre aquestes dades dins de les classes.
    - Herència: Les classes poden derivar d'altres classes.
    - Polimorfisme: Les operacions poden tenir diferents comportaments segons l'objecte que les executa.
    - Abstracció: Permet treballar amb objectes sense necessitat de conèixer-ne els detalls interns.
- **Exemples de Llenguatges**: Java, C++, Python, Ruby, C#.

4. Paradigma Lògic

El paradigma lògic es basa en la resolució de problemes mitjançant regles lògiques. El programador especifica les relacions i condicions, i el motor lògic dedueix les solucions.

- **Característiques**:
    - Utilitza regles lògiques en lloc d'instruccions imperatives.
    - Basat en fets i regles per inferir noves dades.
    - No hi ha una seqüència específica d'execució; el motor de lògica decideix l'ordre de resolució.
- **Exemples de Llenguatges**: Prolog.

5. Paradigma Declaratiu

El paradigma declaratiu se centra en descriure el que el programa ha de fer, en lloc de com fer-ho. Els detalls de la implementació es deixen al compilador o a l'entorn d'execució.

- **Característiques**:
    - El programador especifica els resultats desitjats sense detallar el control del flux.

    - Molt utilitzat en llenguatges de bases de dades i llenguatges de marca.
- **Exemples de Llenguatges**: SQL (gestió de bases de dades), HTML (disseny web), CSS (estils web).

6. Paradigma de Programació per Esdeveniments

El paradigma de programació per esdeveniments es basa en la resposta a esdeveniments o successos externs, com accions de l'usuari o missatges del sistema.

- **Característiques**:
    - Especifica el comportament en resposta a esdeveniments.
    - Utilitzat sovint en interfícies gràfiques d'usuari (GUIs) i aplicacions web.
- **Exemples de Llenguatges**: JavaScript (per aplicacions web), Visual Basic.

7. Paradigma de Programació Concurrent

El paradigma de programació concurrent es basa en l'execució simultània de múltiples processos o fils per millorar el rendiment i la reactivitat dels programes.

- **Característiques**:
    - Permet l'execució de múltiples tasques al mateix temps.
    - Utilitza fils, processos i comunicació entre processos.
- **Exemples de Llenguatges**: Java (amb concurrència multithread), Erlang (dissenyat per a la concurrència), Go.

En resum, els paradigmes de programació defineixen diferents maneres d'abordar la resolució de problemes i l'estructuració del codi. Els principals paradigmes inclouen l'imperatiu, funcional, orientat a objectes, lògic, declaratiu, per esdeveniments i concurrent. Cada paradigma ofereix avantatges específics i és més adequat per a certs tipus de problemes i aplicacions. Conèixer i comprendre aquests paradigmes permet als programadors seleccionar l'enfocament més adequat per a les seves necessitats i escriure codi més eficient, llegible i mantenible.

5. Característiques dels llenguatges més difosos.

C: és un llenguatge de programació (considerat com un d'allò més importants actualment) amb el qual es desenvolupen tant aplicacions com sistemes operatius alhora que forma la base d'altres llenguatges més actuals com Java, C++ o C#.

Són diverses les característiques de C tal com veiem a continuació.

- Estructura de C - Llenguatge estructurat.
- Programació de nivell mitjà (beneficiant-se dels avantatges de la programació d'alt i baix nivell).
- No depèn del maquinari, per la qual cosa es pot migrar a altres sistemes.
- Objectius generals. No és un llenguatge per a una tasca específica, podent programar tant un sistema operatiu, un full de càlcul o un joc.
- Ofereix un control absolut de tot el que passa a l'ordinador.

- Organització de la feina amb total llibertat. Els programes són produïts de forma ràpida i són molt potents.
- Ric en tipus de dades, operadors i variables a C.
- Com a inconvenients hem de dir que no és un llenguatge senzill d'aprendre, que requereix pràctica i un seriós seguiment si volem tenir el control dels programes.

Java: Per comprendre què és Java, cal definir les característiques que el diferencien d'altres llenguatges de programació.

-És simple Java ofereix la funcionalitat d'un llenguatge, derivat de C i C++, però sense les característiques menys usades i més confuses d'aquests, fent-ho més senzill.
- Orientat a objectes. L'enfocament orientat a objectes (OO) és un dels estils de programació més populars. Permet dissenyar el programari de manera que els diferents tipus de dades que es facen servir estiguen units a les seues operacions.
- És distribuït. Java proporciona una gran biblioteca estàndard i eines perquè els programes puguen ser distribuïts.
- Independent a la plataforma. Això significa que programes escrits en el llenguatge Java es poden executar en qualsevol tipus de maquinari, cosa que el fa portable.
- Recol·lector d'escombraries. Quan no hi ha referències localitzades a un objecte, el recol·lector d'escombraries de Java esborra aquest objecte, alliberant així la memòria que ocupava. Això prevé possibles fugides de memòria.
- És segur i sòlid. Proporcionant una plataforma segura per desenvolupar i executar aplicacions que, administra automàticament la memòria, proveeix canals de comunicació segura protegint la privadesa de les dades i, en tenir una sintaxi rigorosa evita que es trenque el codi, és a dir, no permet la corrupció del mateix .
-És multifil. Java aconsegueix dur a terme diverses tasques simultàniament dins del mateix programa. Això permet millorar el rendiment i la velocitat dexecució.

SQL: La solvència, la versatilitat i la consistència de les bases de dades relacionals i de SQL fan d'aquest llenguatge un dels més importants útils i demandats per analitzar dades.

Aquestes són algunes de les principals característiques de SQL,per qué és tan popular i bàsic dins del món de l'anàlisi de dades:

- Integritat de les dades. Quan parlem d'integritat de les dades, ens referim a assegurar que les dades siguen vàlides, correctes i completes. SQL i les bases de dades relacionals tenen la funció de preservar aquesta integritat.
-Llenguatge estandarditzat. Vol dir que es poden desplegar implementacions del mateix llenguatge SQL a diferents sistemes. D'aquesta manera es pot fer servir el mateix codi per a tots ells.
- Senzillesa i claredat. SQL és un llenguatge integral des del punt de vista conceptual, això vol dir que SQL és un llenguatge unificat, clar i simple, de comprensió fàcil.
- Flexibilitat. Una de les raons per les quals SQL és un llenguatge tan utilitzat és per la seua flexibilitat, atesa la seva versatilitat a l'hora d'implantar solucions i per permetre definir diferents maneres de veure les dades per satisfer les especificacions requerides per part de l'usuari.

6. Fases del desenrotllament d'una aplicació

El desenvolupament d'una aplicació (ja siga web, mòbil o d'escriptori) sol seguir un conjunt de fases estructurades per a assegurar que el producte final siga útil, eficient i

mantingut correctament. Aquest procés pot variar depenent de la metodologia emprada (per exemple, cascada vs àgil), però les fases principals són bastant comunes:

1. Anàlisi i recollida de requisits

Es defineix què necessita l'usuari o el client.

Es recullen funcionalitats clau, característiques desitjades i limitacions tècniques.

Exemple: una app de comerç electrònic necessita registre d'usuaris, catàleg de productes, cistella de compra i passarel·la de pagament.

2. Disseny (arquitectura i interfície)

Arquitectura del sistema: quina tecnologia s'utilitzarà, com es comunicaran els mòduls, base de dades, seguretat, etc.

UI/UX (disseny d'interfície i experiència d'usuari): es creen prototips o wireframes perquè l'usuari puga validar el flux de navegació i l'aspecte visual.

3. Desenvolupament / Implementació

Els programadors construeixen el codi segons el disseny definit.

Es poden fer versions incrementals (en metodologies àgils) o una implementació completa (en cascada).

Es fan revisions de codi i integració contínua per reduir errors.

4. Proves i control de qualitat

Es realitzen tests unitaris, d'integració i d'usuari per a comprovar que l'app funciona com s'esperava.

Es busquen errors (bugs) i es corregeixen.

També s'avaluen aspectes com rendiment, seguretat i usabilitat.

5. Desplegament

L'aplicació es publica en el seu entorn definitiu: App Store / Google Play (si és mòbil), servidor web (si és web) o distribució corporativa.

Es poden fer llançaments graduals per reduir riscos (per exemple, a un percentatge d'usuaris).

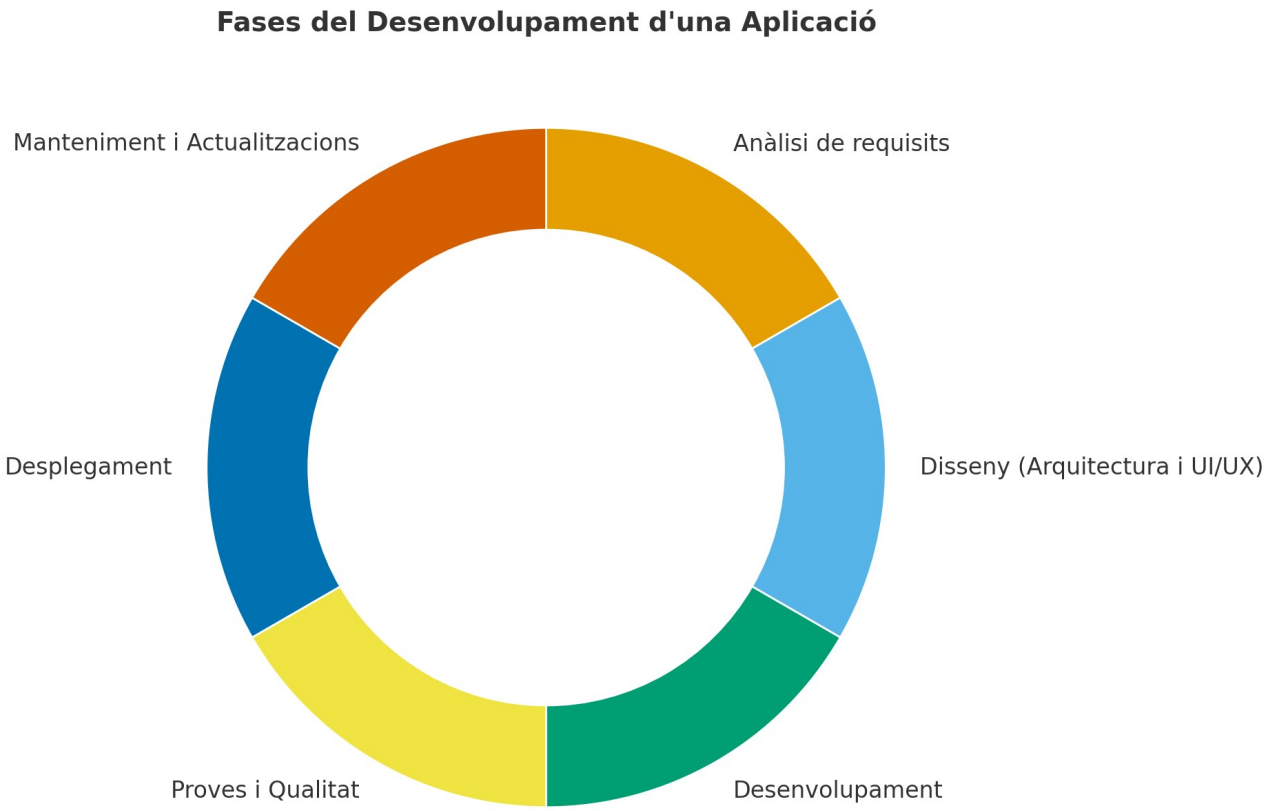
6. Manteniment i actualitzacions

Després del llançament, l'app necessita correccions de bugs, actualitzacions i noves funcionalitats.

També es fa el seguiment de l'ús per a millorar la UX i adaptar-se a nous requisits del mercat.

Cal tenir en compte que en metodologies modernes com Scrum o Kanban, aquestes fases no són estrictament lineals sinó que es repeteixen de forma iterativa, amb cicles curts (sprints) per lliurar valor més ràpid.

Esquema visual tipus diagrama amb aquestes fases perquè siga més fàcil de veure-ho:



7. Estructures d'equips de treball en desenvolupament web

7.1. Equip funcional clàssic (per rols)

Cada persona té una tasca molt definida.

Frontend developers → creen la interfície i la part visible (HTML, CSS, JavaScript, frameworks com React, Angular...).

Backend developers → gestionen la lògica del servidor i la base de dades (Node.js, Django, Laravel...).

UI/UX designers → dissenyen experiència i interfície.

QA testers → comproven qualitat i usabilitat.

DevOps → desplegament, servidors, integració contínua.

Adequat per projectes grans i complexos. Requereix bona coordinació, ja que depenen els uns dels altres.

7.2. Equips multidisciplinaris (model àgil / Scrum)

Típic de startups i empreses àgils. Un equip xicotet (5–9 persones) que inclou frontend, backend, disseny i QA dins el mateix grup.

Rols addicionals als anteriors:

- Product Owner (prioritza funcionalitats).
- Scrum Master (facilita processos).

Molt àgil: cada sprint entrega valor real.

7.3. Estructura per capes (arquitectura web)

Algunes empreses organitzen els equips segons la capa de l'aplicació:

- Equip de presentació (frontend)
- Equip de lògica (backend)
- Equip de dades (DB, analítica, big data)
- Equip d'infraestructura (DevOps, seguretat)

Eficaç en grans corporacions. Risc de crear "silos" de coneixement (aïllament).

7.4. Estructura per funcionalitats o "feature teams" (squads)

- Equips autònoms i xicotets, cadascun treballa en una funcionalitat concreta de l'app web.
- Exemple: un "squad" per autenticació d'usuaris, un altre per compres, un altre per xat en viu.
- Cada squad té frontend, backend i QA.

Molt útil per aplicacions grans i modulars (exemple: Amazon, Spotify). Cal coordinació per evitar duplicacitat d'esforços.

7.5. Equip "full-stack"

- Els membres són desenvolupadors full-stack que poden tocar tant el frontend com el backend.
- Sovint en projectes xicotts o mitjans.

Més flexibilitat i velocitat. Pot mancar profunditat tècnica en aspectes molt específics.

8. Procés d'obtenció de codi executable a partir del codi font

Compilador

Un compilador és un programa que tradueix codi escrit en un llenguatge de programació (anomenat font) a un altre llenguatge (conegut com a objecte). En aquest tipus de traductor el llenguatge font és generalment un llenguatge d'alt nivell i l'objecte un llenguatge de baix nivell, com assembly o codi màquina. Aquest procés de traducció es coneix com a compilació

Intèrprets

Intèrpret és un programa informàtic capaç d'analitzar i executar altres programes. Els intèrprets es diferencien dels compiladors o dels assembladors en què mentre aquests tradueixen un programa des de la seva descripció en un llenguatge de programació al codi de màquina del sistema, els intèrprets només fan la traducció a mesura que siga necessària, típicament, instrucció per instrucció , i normalment no guarden el resultat d'aquesta traducció.

Usant un intèrpret, un sol fitxer font pot produir resultats iguals fins i tot en sistemes summament diferents (exemple. un PC i una PlayStation 5). Usant un compilador, un sol fitxer font pot produir resultats iguals només si és compilat a diferents executables específics a cada sistema.

Maquines virtuals

Com ja hem dit abans, pel que fa a les màquines virtuals, la més coneguda per tots es la de Java. Una màquina virtual Java (JVM) és una màquina virtual de procés nadiu, és a dir, executable en una plataforma específica, capaç d'interpretar i executar instruccions expressades en un codi binari especial (el bytecode Java), el qual és generat pel compilador del llenguatge Java.

Tema 2: Instal·lació i ús d'entorns de desenrotllament integrat IDE's

---

1. IDE

Un **entorn de desenrotllament integrat** (IDE, de l'anglés *Integrated Development Environment*) és una eina que concentra en un sol programa tot el que un desenvolupador necessita per a escriure, provar i mantindre aplicacions.

En el context de la **programació web**, els IDE's són molt útils perquè ajuden a treballar amb múltiples llenguatges (HTML, CSS, JavaScript, PHP, Python, etc.), gestionar projectes complexos i integrar serveis com control de versions o servidors locals.

1.1. Característiques principals d'un IDE

- **Editor de codi avançat**: ressaltat de sintaxi, autocompletat i suggeriments de codi.
- **Depurador (debugger)**: permet executar el programa pas a pas i detectar errors.
- **Gestor de projectes**: organitza carpetes, arxius i recursos del projecte web.
- **Integració amb control de versions**: normalment amb Git.
- **Simulació o servidor local**: per provar aplicacions web abans de desplegar-les.
- **Extensions i plugins**: afegeixen funcionalitats addicionals (suport a nous llenguatges, frameworks, etc.).

1.2. Exemples d'IDE's populars en programació web

- **Visual Studio Code (VS Code)**: molt utilitzat, lleuger però potent, amb gran quantitat d'extensions per a frameworks web.
- **WebStorm**: especialitzat en JavaScript, Node.js, React, Angular, Vue…
- **Eclipse**: tradicional en Java, però amb plugins per a desenvolupament web.
- **NetBeans**: compatible amb PHP, Java i frameworks web.
- **PHPStorm**: centrat en PHP i tecnologies web relacionades.

1.3. Beneficis de s'ús d'IDE's en el desenvolupament web

- **Productivitat**: l'autocompletat i els suggeriments acceleren la codificació.
- **Menys errors**: gràcies al depurador i al ressaltat de sintaxi.
- **Organització**: facilita treballar amb projectes grans.
- **Col·laboració**: integració amb GitHub, GitLab o Bitbucket.

2. Comparativa dels IDE's més populars

Parlarem fonamentalment de programació web i veurem punts forts, febles i el tipus de projecte per al qual són més adequats:

2.1 Visual Studio Code (VS Code)

Punts forts:

- Gratuït i de codi obert.
- Lleuger i molt personalitzable amb extensions.
- Suporta pràcticament tots els llenguatges i frameworks web.
- Integració nativa amb Git i terminals integrats.

Punts febles:

- Necessita extensions per a moltes funcionalitats avançades.
- Pot consumir molta memòria amb massa extensions.

Ideal per a: projectes web de qualsevol grandària (HTML, CSS, JS, React, Vue, Angular, Node, PHP, Python...).

2.2. WebStorm (JetBrains)

Punts forts:

- Especialitzat en JavaScript i frameworks moderns (React, Angular, Vue, Node.js).
- Autocompletat i refactorització molt potents.
- Depurador i integració amb test automàtic.

Punts febles:

- De pagament (amb llicència anual).
- Pesat en comparació amb VS Code.

Ideal per a: equips i projectes professionals centrats en JavaScript/TypeScript i frameworks moderns.

2.3. NetBeans

Punts forts:

- Gratuït i de codi obert.
- Suporta bé PHP, Java i aplicacions web clàssiques.
- Bon entorn per a programadors que combinen web + backend en Java.

Punts febles:

- Interfície menys moderna.
- Menys extensions i comunitat que VS Code.

Ideal per a: aplicacions web amb PHP o Java, especialment en entorns acadèmics.

2.4. Eclipse

Punts forts:

- Gratuït i extensible amb molts plugins.
- Molt potent per a Java i desenvolupament empresarial.

Punts febles:

- Interfície poc intuïtiva.
- Menys adaptat a frameworks web moderns sense plugins addicionals.

Ideal per a: projectes grans en Java + web, entorns corporatius.

2.5. PHPStorm (JetBrains)

Punts forts:
- Especialitzat en PHP i frameworks com Laravel, Symfony, WordPress.
- Integració amb bases de dades i eines de desplegament.
- Depurador avançat i suport complet per a PHP.

Punts febles:
- De pagament.
- No tan flexible per a altres llenguatges fora de l'ecosistema PHP.

Ideal per a: projectes web basats en PHP amb backend complex.

3. Quin triar?

Vols flexibilitat i gratuïtat? --> VS Code.

Treballes amb React/Angular/Vue i vols potència? --> WebStorm.

PHP amb frameworks? --> PHPStorm.

Projectes Java + web? --> Eclipse o NetBeans.

Taula comparativa visual (per a veure quina eina s'adapta millor segons llenguatge, preu i ús)?

IDE	Llenguatges/ Frameworks forts	Preu	Punts forts	Punts febles	Ideal per a...
VS Code	HTML, CSS, JS, React, Vue, Angular, Node, PHP, Python, etc.	Gratuït	Lleuger, extensible, gran comunitat, integració amb Git	Depén molt d'extensions, pot ser pesat amb massa plugins	Projectes web de qualsevol mida, generalista
WebStorm	JavaScript, TypeScript, React, Angular, Vue, Node.js	De pagament (JetBrains)	Potent en JS, refactorització avançada, depurador integrat	Pesat, no gratuït	Projectes professionals centrats en JS i frameworks moderns
NetBeans	PHP, Java, HTML, CSS, JS bàsic	Gratuït	Bo per a PHP i Java, integració backend+frontend	Interfície antiga, menys plugins	Projectes educatius o web + backend Java/PHP
Eclipse	Java, PHP, Python (amb plugins)	Gratuït	Molt potent en Java, altament extensible	Poc intuïtiu, més complex per a web modern	Grans projectes empresarials amb Java + web
PHPStorm	PHP, Laravel, Symfony, WordPress	De pagament (JetBrains)	Especialitzat en PHP, integració amb bases de dades i test	Només realment útil en ecosistema PHP, de pagament	Projectes web centrats en PHP amb backend robust

Aquesta taula et pot servir com a guia ràpida per triar IDE segons el teu projecte.

4. Recomanació

Segons el tipus de projectes web que et puguen interessar:

Si t'interessa el **Front-End modern** (React, Angular, Vue, HTML, CSS, JS)

Recomanació:

- VS Code si vols gratuïtat, rapidesa i flexibilitat.
- WebStorm si vols una eina més completa i professional (sobretot si treballes en equips grans i amb TypeScript).

Si vols fer **Full-Stack (front + back)** amb Node.js, Python, etc.

Recomanació:

- VS Code: perfecte per a projectes híbrids, amb extensions per a Node, Express, Django, Flask, etc.
- WebStorm: si el back és en Node.js i vols depuració avançada.

Si et centres en **Backend amb PHP** (Laravel, Symfony, WordPress...)

Recomanació:

- PHPStorm: el millor entorn, especialitzat i amb totes les eines integrades.
- VS Code també funciona bé, però necessitaràs extensions.

Si treballes amb **Java en aplicacions web empresarials**

Recomanació:

- Eclipse: molt utilitzat en entorns corporatius, amb suport ampli per Java EE i Spring.
- NetBeans: més senzill i amb millor integració nativa entre Java + HTML/CSS/JS.

Si estàs **començant i vols aprendre programació web**

Recomanació:

- VS Code: fàcil d'instal·lar, comunitat enorme, tutorials a dojo.
- NetBeans: si estudies en un entorn acadèmic on Java/PHP siga habitual.

Resum de recomanacions:

VS Code → millor opció general, flexible i gratuïta.

WebStorm / PHPStorm → si treballes professionalment amb JS o PHP i vols potència extra.

Eclipse / NetBeans → si estàs en entorns acadèmics o empresarials amb Java.

5. Extensions de VS Code per a programació web

Ací tens una llista de les extensions essencials de VS Code per a programació web, classificades per tipus:

Per a HTML i CSS

- Live Server → llança un servidor local i actualitza automàticament la web en el navegador quan guardes.
- Auto Rename Tag → renomena automàticament les etiquetes d'obertura/tancament.
- Prettier → formatador de codi per mantindre HTML/CSS/JS ordenat.
- CSS Peek → et deixa navegar fàcilment a les definicions de classes CSS.
- IntelliSense for CSS class names → autocompletat de classes CSS definides als teus arxius.

Per a JavaScript i frameworks

- ESLint → analitza el codi JS/TS i t'avisa d'errors o males pràctiques.
- JavaScript (ES6) code snippets → snippets de codi modern per a JS.
- React Developer Tools (si treballes amb React) → snippets i suport JSX.
- Angular Language Service (si treballes amb Angular) → autocompletat i anàlisi de codi.
- Vue Language Features (Volar) (si treballes amb Vue.js) → suport complet per a Vue.

Per a backend i Full-Stack

- REST Client → et permet provar APIs directament des de VS Code.
- Thunder Client → alternativa senzilla a Postman per test d'APIs.
- PHP Intelephense (si treballes amb PHP) → autocompletat i anàlisi de codi PHP.
- Python (si treballes amb Django/Flask) → suport complet per a Python i entorns virtuals.

Productivitat i utilitats

- GitLens → millora la integració amb Git i mostra qui va escriure cada línia.
- Path Intellisense → autocompletat de rutes d'arxius i carpetes.
- Bracket Pair Colorizer 2 → pinta amb colors diferents els claudàtors, útil en codi llarg.
- Error Lens → mostra errors i advertiments directament al codi.
- Material Icon Theme → icones de carpetes i arxius més visuals.

Amb aquest pack d'extensions, VS Code es transforma en un IDE completíssim per a web.

Kit recomanat o configuració mínima però potent de VS Code

Per a començar amb el desenvolupament web: kit recomanat per configurar VS Code i començar a desenvolupar webs de manera ràpida i eficient:

1. Extensions bàsiques (Instal·la aquestes primer):
   - Live Server → servidor local amb auto-reload.
   - Prettier → formatador de codi automàtic.
   - Auto Rename Tag → renomena etiquetes HTML d'obertura/tancament alhora.
   - Path Intellisense → autocompletat de rutes d'arxius.

- Bracket Pair Colorizer 2 → claudàtors acolorits per a llegibilitat.
- Material Icon Theme → millora visual de carpetes i arxius.

2. Extensions segons llenguatge
   - HTML & CSS: IntelliSense for CSS class names, CSS Peek.
   - JavaScript/TypeScript: ESLint, JavaScript (ES6) code snippets.
   - React: React Developer Tools, ES7+ React/Redux snippets.
   - Vue: Volar (Vue Language Features).
   - Angular: Angular Language Service.
   - PHP: PHP Intelephense.
   - Python: Python (oficial de Microsoft).

Tema 3: Disseny i realització de proves

---

1. Introducció

Parlem de disseny i realització de proves de software (o software testing), que és una part clau en l'enginyeria del programari.

Què són les proves de software? Són un conjunt d'activitats planificades que busquen verificar i validar que un sistema informàtic:

- Funciona d'acord amb els requisits establits.
- No conté errors crítics.
- Complix amb els criteris de qualitat (rendiment, seguretat, usabilitat, etc.).

L'objectiu principal és detectar errors abans que el programari arribe a l'usuari final.

Podem dir que tenim dues parts pel que fa al testing:

1. Disseny = planificar i definir com provar.
2. Realització = executar, analitzar i reportar resultats.

2. Les proves en el cicle de vida d'un projecte

Quan parlem de les proves (testing) dins del cicle de vida d'un projecte, normalment ens referim a les diferents fases en què es realitza la verificació i validació del projecte per a assegurar que compleix els requisits i funciona correctament. A continuació t'explicaré com encaixen les proves dins del cicle de vida típic d'un projecte, especialment en desenvolupament de programari:

1. Planificació del projecte

Objectiu: Definir els objectius, abast, recursos i terminis.

Proves: Encara no hi ha proves actives, però es planifica l'estratègia de testing, es defineixen tipus de proves i criteris d'acceptació.

Exemple: Decidir si es farà prova unitària, integració, sistemes i acceptació.

2. Anàlisi de requisits

Objectiu: Recollir i analitzar els requisits del projecte.

Proves: Es comencen a definir casos de prova basats en requisits. Això ajuda a assegurar que el que es desenvoluparà serà verificable.

Exemple: Si el requisit diu *l'usuari ha de poder registrar-se amb email*, es crea un cas de prova que comprove això.

3. Disseny

Objectiu: Dissenyar l'arquitectura, fluxos i components del sistema.

Proves: Es planifiquen proves més específiques:

- Proves d'integració per veure com es connectaran els diferents components.
- Proves de rendiment inicials per determinar criteris.

4. Implementació / Desenvolupament

Objectiu: Programar o construir els components del projecte.

Proves:

- Proves unitàries: Comproven que cada unitat de codi funciona correctament.
- Proves de component: Validar funcionalitat de cada mòdul individual.

5. Integració i proves del sistema

Objectiu: Comprovar que tots els components treballen junts correctament.

Proves:

- Proves d'integració: Els mòduls interactuen sense errors.
- Proves de sistema: S'avalua el sistema complet segons els requisits funcionals i no funcionals.

6. Prova d'acceptació

Objectiu: Validar que el producte compleix els requisits del client.

Proves:

- Proves d'acceptació per l'usuari (UAT): El client prova el sistema.
- Proves pilots / beta: Petits grups d'usuaris finals utilitzen el sistema abans del llançament total.

7. Desplegament i manteniment

Objectiu: Posar el projecte en funcionament i mantenir-lo.

Proves:

- Proves de regressió: Quan es fan modificacions, s'assegura que no apareguin errors nous.
- Proves de manteniment: Corregir errors detectats després del llançament i validar canvis.

Podem dir que les proves són un procés continu, des de la planificació fins al manteniment. No només serveixen per trobar errors, sinó per garantir qualitat, complir requisits i assegurar satisfacció de l'usuari.

3. Procediments, tipus i casos de proves

En el desenvolupament d'aplicacions web, les proves (o testing) són essencials per garantir que el sistema funcione correctament, siga segur, escalable i fàcil de mantenir. Ací tens un resum dels procediments, tipus i casos de proves més habituals:

3.1. Procediments de proves

Els procediments són el conjunt d'activitats o passos que es segueixen per dur a terme les proves. Inclouen la planificació, execució i seguiment. Són el "com" es fan les proves.

1- Planificació de proves. Es defineixen els objectius, l'abast, els recursos i el calendari de les proves.

2- Disseny de casos de prova. Es creen escenaris específics que descriuen què provar, amb quines dades i quins resultats s'esperen.

3- Execució de proves. Es duen a terme les proves, manualment o automàticament, i es registren els resultats.

4- Anàlisi de resultats. Es comparen els resultats esperats amb els obtinguts per a detectar errors o desviacions.

5- Informe d'errors i correcció. Els defectes es documenten, es corregeixen i es tornen a provar.

3.2. Tipus de proves

Els tipus de proves fan referència a les categories o classes de proves que es poden fer, segons l'objectiu o la part del sistema que es vol validar. Són el "què" es prova.

1. Proves unitàries (Unit Testing). Verifiquen el funcionament de components individuals (funcions, mètodes). Ex: pytest (pytest és una biblioteca de Python molt popular per fer proves unitàries i funcionals) en Python, Jest en JavaScript.

2. Proves d'integració. Comproven que diferents mòduls funcionen correctament junts. Ex: backend + base de dades, API + frontend.

3. Proves funcionals. Validen que el sistema compleixi els requisits funcionals. Ex: provar que un usuari pot iniciar sessió correctament.

4. Proves d'acceptació. Simulen escenaris reals per validar que l'aplicació compleix les expectatives del client.

5. Proves de regressió. Asseguren que els canvis no han introduït nous errors en funcionalitats ja existents.

6. Proves de rendiment. Mesuren la velocitat, escalabilitat i estabilitat sota càrrega. Ex: Locust, JMeter.

7. Proves de seguretat. Detecten vulnerabilitats com injeccions SQL, XSS, etc.

8. Proves d'usabilitat. Avaluen l'experiència d'usuari (UX), accessibilitat i interfície.

3.3. Casos de prova típics en aplicacions web

- Autenticació i autorització: iniciar sessió, permisos d'usuari.
- Formularis: validació de camps, enviament correcte.
- API REST: respostes correctes, codis d'estat HTTP.
- Interacció amb base de dades: inserció, consulta, actualització, esborrat.
- Responsive design: visualització en diferents dispositius.
- Errors i excepcions: gestió d'errors inesperats.
- Internacionalització: suport per a múltiples idiomes.

4. Proves de codi

4.1. Cobriment de codi (Code Coverage)

El cobriment mesura quin percentatge del codi ha estat executat durant les proves. No garanteix que el codi siga correcte, però sí que ha estat testejat.

Tipus de cobriment:

- Cobriment de línies: quantes línies s'han executat.
- Cobriment de branques: quantes condicions (if, else, etc.) s'han provat.
- Cobriment de funcions: quantes funcions s'han cridat.

Eines:

- coverage.py (Python)
- Jest --coverage (JavaScript)
- SonarQube (multi-llenguatge)

4.2. Valors límit (Boundary Value Testing)

Es prova el comportament del sistema en els valors extrems o límit d'un rang d'entrada.

Exemple: Si una funció accepta edats entre 18 i 65. Proves típiques: 17, 18, 65, 66. Això ajuda a detectar errors en condicions com >=, <=, etc.

Avantatge: detecta errors que sovint passen en els límits de validació.

4.3. Classes d'equivalència (Equivalence Partitioning)

Es divideixen les dades d'entrada en grups (classes) que es consideren equivalents, i es prova només un valor per classe.

Exemple: Per una entrada de nota entre 0 i 10:

- Classes vàlides: 0–10 → prova amb 5
- Classes invàlides: <0, >10 → prova amb -1, 11

Avantatge: redueix el nombre de proves necessàries mantenint la cobertura lògica.

4.4. Altres tècniques útils

- Proves de taula de decisions: per sistemes amb múltiples condicions.
- Proves de camins lògics: per assegurar que tots els camins del codi s'han executat.
- Proves d'estat: per sistemes que canvien d'estat (ex: login/logout).

5. Proves unitàries

Les proves unitàries són una part fonamental del testing en programació web. Són proves que verifiquen el funcionament correcte de funcions o mòduls individuals del codi, de forma aïllada. Ara anem a descriure les eines d'automatització més utilitzades per fer proves unitàries segons el llenguatge i l'entorn:

5.1. Eines d'automatització per a proves unitàries

Python

- pytest: molt flexible, fàcil d'utilitzar, compatible amb fixtures i plugins.
- unittest: biblioteca integrada a Python, similar a JUnit.
- nose2: successor de nose, compatible amb unittest.

JavaScript / TypeScript

- Jest: molt popular per React, Node.js i TypeScript. Suporta mocks i cobertura.
- Mocha + Chai: combinació potent per proves unitàries i d'integració.
- Vitest: alternativa moderna a Jest, molt ràpida i compatible amb Vite.

Java

- JUnit: estàndard per proves unitàries en Java.
- TestNG: més flexible que JUnit, ideal per proves complexes.

PHP

- PHPUnit: eina oficial per proves unitàries en PHP.

Go

- testing: biblioteca integrada al llenguatge Go.
- Testify: biblioteca amb assertions més avançades.

Ruby

- RSpec: molt expressiva i llegible.
- MiniTest: més lleugera, integrada a Ruby.

5.2. Automatització amb CI/CD

Aquestes eines es poden integrar en sistemes d'automatització com:

- GitHub Actions
- GitLab CI
- Jenkins
- CircleCI

Això permet que les proves s'executen automàticament cada vegada que es fa un commit, push o merge, garantint que tot vaja bé.

6. Documentació d'Incidències de Proves

Per a documentar les incidències de proves, podem seguir una estructura clara i útil que facilite la comprensió, seguiment i resolució dels problemes detectats durant el procés de test. Ací tens un exemple de plantilla que pots adaptar segons el teu projecte:

Informació general de la prova

- Nom de la funcionalitat provada: (ex. Xat en temps real amb WebSocket)
- Entorn de proves: (ex. Docker local, Azure, Vercel, etc.)
- Data de la prova: (ex. 29/08/2025)
- Responsable de la prova: (ex. Xavier Blanes)

Incidència #1

- Títol: Error de connexió amb WebSocket
- Descripció: En iniciar el frontend, no s'estableix connexió amb el servidor WebSocket. La consola mostra l'error WebSocket connection failed: Error during WebSocket handshake.
- Pasos per reproduir:
  1. Obrir el navegador i accedir al frontend.
  2. Comprovar la consola del navegador.
- Resultat esperat: Connexió establerta amb el servidor WebSocket.
- Resultat obtingut: Error de handshake.
- Gravetat: Alta
- Captura de pantalla / Logs: (adjuntar si escau)
- Estat: Pendent de resolució
- Comentaris: Pot estar relacionat amb configuració de CORS o certificat SSL.

Incidència #2

- Títol: Missatges duplicats al xat
- Descripció: Quan un usuari envia un missatge, aquest apareix duplicat al frontend.
- Pasos per reproduir:
  1. Enviar un missatge des del frontend.
  2. Observar la llista de missatges.
- Resultat esperat: Un sol missatge apareix.
- Resultat obtingut: El missatge apareix dues vegades.
- Gravetat: Mitjana
- Captura de pantalla / Logs: (adjuntar si escau)
- Estat: En investigació
- Comentaris: Pot ser que el frontend estigui afegint el missatge localment i també rebent-lo via WebSocket.

Incidència #3

- Títol: Error de desplegament resolt
- Descripció: El contenidor de FastAPI no arrencava per un error en el fitxer Dockerfile.
- Solució aplicada: S'ha afegit la línia CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"].
- Estat: Resolta

7. Test doubles

Els dobles de prova (en anglès test doubles) són objectes que substitueixen components reals en proves de programari per simular el seu comportament. S'utilitzen per aïllar el sistema que s'està provant i controlar les condicions de prova.

Més clar, quan fem proves de programari, sovint volem provar una part concreta del sistema (per exemple, una funció que envia correus electrònics). Però aquesta funció pot dependre d'altres parts del sistema, com una base de dades o un servidor extern.

Els dobles de prova són com actors que fan veure que són una part del sistema, però en realitat no són reals. Ens ajuden a fer proves més controlades, ràpides i segures.

Suposem que tens una funció que calcula el preu d'un producte amb descompte, però aquest descompte ve d'un servei extern. Per a provar aquesta funció, no vols cridar el servei extern real cada vegada. Així que crees un doble de prova que simula el comportament del servei:

```
class ServeiDescompteFals:
    def obtenir_descompte(self, producte_id):
        return 10  # sempre retorna 10 com a descompte
```

Ací tens un resum dels tipus principals, amb les seves característiques:

Tipus de Dobles de Prova

1. Dummy

Ús: S'utilitza quan un objecte és necessari per complir la signatura d'un mètode, però no s'utilitza realment.

Característiques:

- No té comportament.
- Només serveix per evitar errors de compilació o execució.

Exemple: Un objecte passat com a paràmetre però que no s'utilitza dins del mètode.

2. Stub

Ús: Proporciona dades predefinides a la unitat de prova.

Característiques:

- Retorna valors fixos.
- No verifica comportament.

Exemple: Un mètode que retorna sempre el mateix resultat per a simular una resposta d'una API.

3. Spy

Ús: Registra informació sobre les crides que rep.

Característiques:

- Pot actuar com un stub.
- Permet verificar si s'ha cridat un mètode, quantes vegades, amb quins paràmetres.

Exemple: Verificar que s'ha cridat sendEmail() exactament una vegada.

4. Mock

Ús: Simula el comportament d'un objecte i verifica interaccions.

Característiques:

- S'especifica el comportament esperat.
- Es verifica que les interaccions compleixin les expectatives.

Exemple: Comprovar que s'ha cridat save() amb un objecte concret.

5. Fake

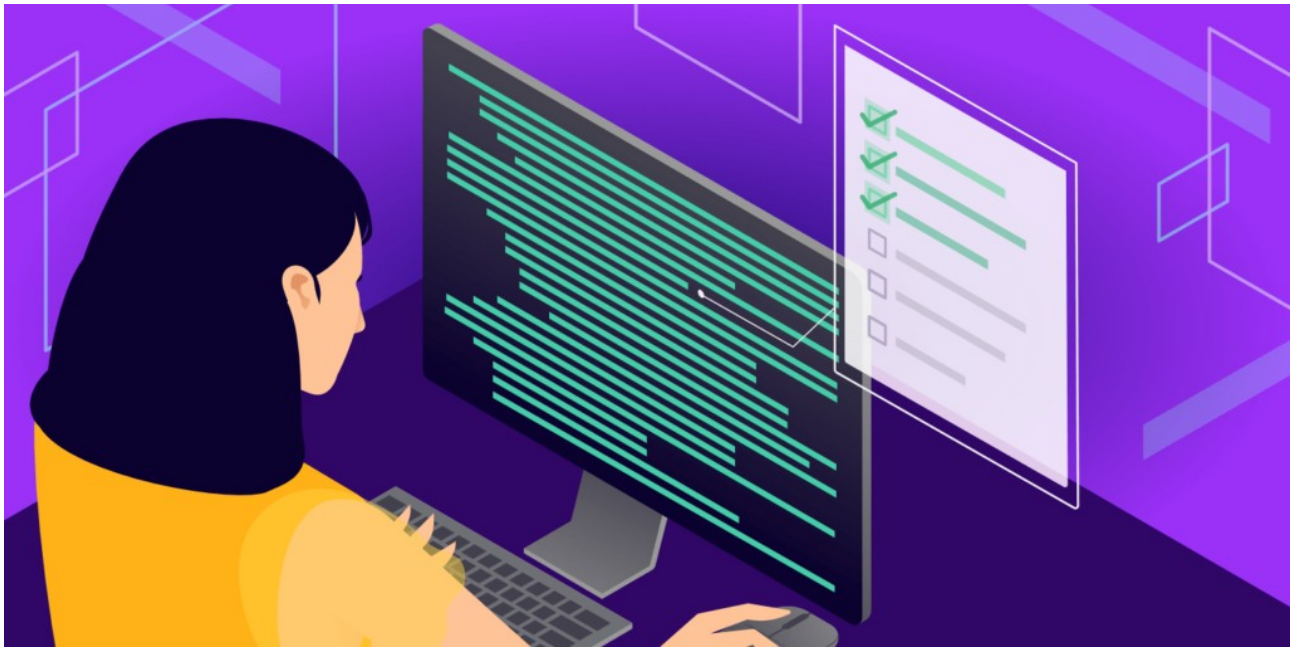
Ús: Implementació funcional però simplificada.

Característiques:

- Pot tenir lògica real però no està pensada per producció.
- S'utilitza per proves més integrades.

Exemple: Una base de dades en memòria per fer proves sense accedir a la real.

Tema 4: Optimització, documentació i control de versions



---

1. Introducció

L'optimització consisteix en millorar el codi perquè siga més eficient, més ràpid, més fàcil de mantenir o consumir menys recursos.

Els objectius principalsseran:

- Millorar el rendiment (menys temps d'execució).
- Reduir l'ús de memòria.
- Simplificar la lògica per fer-la més clara.
- Eliminar duplicacions o codi innecessari.

Pel que fa a la documentació és l'acte d'explicar què fa el codi, com funciona i com s'ha d'utilitzar. Pot estar dins del codi (comentaris) o en documents externs (manuals, fitxers README, etc.).

Tipus de documentació:

- Comentaris en el codi: expliquen parts específiques.
- Docstrings: explicacions dins de funcions o classes.
- README: fitxer que descriu el projecte, com instal·lar-lo i com utilitzar-lo.
- Manuals tècnics o d'usuari: més detallats, per a desenvolupadors o usuaris finals.

Si ho fem bé tindrem:

- Facilita el manteniment del codi.
- Ajuda altres desenvolupadors (o tu mateix en el futur).
- Evita errors i duplicitats.
- Fa que el projecte siga més professional.

2. Refactorització

La refactorització és el procés de modificar el codi intern d'un programa sense canviar el comportament extern. L'objectiu és millorar la qualitat del codi: fer-lo més llegible, mantenible, eficient i menys propens a errors.

És com reorganitzar una habitació: no canvies el que hi ha, però ho col·loques millor perquè siga més fàcil de moure't i trobar les coses.

Limitacions

Tot i els avantatges, la refactorització té algunes limitacions:

- Risc d'introduir errors si no es fan proves adequades.
- Temps invertit: pot ser costós si no es planifica bé.
- No aporta noves funcionalitats: pot semblar que no "avances" en el projecte.

- Dependències externes: pot ser difícil si el codi depèn de sistemes antics o poc documentats.

Patrons de refactorització més usuals

Ací tens alguns patrons habituals:

Patró	Descripció
Extract Method	Separar part d'un codi en una funció nova per fer-lo més llegible.
Rename Variable	Canviar noms de variables per fer-les més descriptives.
Inline Variable	Eliminar variables innecessàries que només fan de pas.
Replace Temp with Query	Substituir variables temporals per funcions que calculen el valor directament.
Encapsulate Field	Fer que l'accés a una propietat passi per mètodes (getters/setters).
Simplify Conditional	Fer condicions més clares i fàcils d'entendre.
Remove Dead Code	Eliminar codi que no s'utilitza.

Refactorització i proves

La refactorització ha d'anar sempre acompanyada de proves per assegurar que el comportament del programa no ha canviat.

- Proves unitàries: són essencials abans i després de refactoritzar.
- Test de regressió: asseguren que no s'han introduït errors.
- Cobertura de codi: ajuda a saber si les proves cobreixen tot el que cal.

Si no fas proves, refactoritzar és com fer cirurgia sense monitoritzar el pacient.

3. Eines d'ajuda a la refacció

Hi ha moltes eines que faciliten la refactorització automàtica o assistida:

Entorns de desenvolupament (IDE)

- Visual Studio Code: té extensions com Refactorix.
- PyCharm / IntelliJ IDEA: ofereixen refactorització automàtica molt potent.
- Eclipse / NetBeans: per Java, amb eines integrades.

Llibreries i eines específiques

- SonarQube: analitza la qualitat del codi i suggereix millores.
- ESLint / Prettier: per JavaScript, ajuden a mantenir estil i detectar problemes.
- Black / Flake8: per Python, milloren la llegibilitat i detecten errors.

4. Control de versions

El control de versions és una pràctica fonamental en el desenvolupament de programari (i també en altres àmbits com la documentació o programació) que permet registrar, gestionar i seguir els canvis que es fan en els fitxers al llarg del temps.

Què és exactament? És com tenir una "història" de cada canvi que s'ha fet en un projecte. Això permet:

- Tornar a versions anteriors si cal.
- Saber qui ha fet cada canvi i per què.
- Treballar en equip sense trepitjar-se els canvis.
- Provar noves funcionalitats en branques separades abans d'integrar-les.

L'eina més popular i més coneguda és Git, i sovint s'utilitza amb plataformes com: GitHub, GitLab o Bitbucket.

Exemple pràctic: imagina que tens un fitxer main.py i el modifiques diverses vegades. Amb Git pots:

- Fer commits: guardar cada canvi amb un missatge explicatiu.
- Crear branques: per provar noves idees sense afectar el codi principal.
- Fer merge: unir els canvis d'una branca amb la principal.
- Fer revert: tornar a una versió anterior si alguna cosa falla.

Conceptes Bàsics del Control de Versions

Repositori (Repository): És el lloc on es guarda tot el codi font i l'historial de canvis. Pot ser local (en el teu ordinador) o remot (en un ser vidor).

Commit: Una acció que guarda els canvis fets als fitxers en el repositori. Cada commit sol portar un missatge descriptiu que explica què s'ha canviat i per què.

Branch (Branca): Una línia paral·lela de desenvolupament dins del repositori. Les branques permeten treballar en noves característiques o corregir errors de manera independent sense afectar a la branca principal (master/main).

Merge (Fusió): La combinació de canvis de diferents branques en una única branca. Això permet integrar el treball fet en branques diferents.

Conflict: Situació que ocorre quan dos canvis diferents afecten la mateixa part del codi i el sistema de control de versions no pot combinar-los automàticament. Els conflictes han de ser resolts manualment pels desenvolupadors.

Beneficis del Control de Versions

Historial de Canvis: Permet veure qui ha fet què i quan, facilitant la traçabilitat dels canvis i la possibilitat de revertir a versions anteriors si es detecten errors.

Col·laboració: Facilita el treball en equip, permetent que múltiples desenvolupadors treballin simultàniament en el mateix projecte sense sobreescriure els canvis dels altres.

Gestió de Versions: Permet mantenir diferents versions del projecte, com ara versions estables, versions en desenvolupament, i branques per a característiques noves.

Reversió de Canvis: Facilita la reversió a una versió anterior si un canvi introdueix un error o problema.

Seguretat: Guardar el codi en repositoris remots proporciona una còpia de seguretat que protegeix contra pèrdues de dades en cas de fallades de maquinari.

5. Eines populars de control de versions

Git

El sistema de control de versions distribuït més utilitzat. Git permet tenir còpies locals completes del repositori i facilita la col·laboració a través de plataformes com GitHub, GitLab o Bitbucket.

Comandes bàsiques:

git init: Inicialitza un nou repositori Git.

git clone: Clona un repositori remot a la màquina local.

git add: Afegeix canvis al staging area.

git commit: Guarda els canvis al repositori amb un missatge descriptiu.

git push: Envia els commits locals al repositori remot.

git pull: Actualitza el repositori local amb els canvis del repositori remot.

git merge: Combina els canvis d'una branca amb una altra.

Subversion (SVN)

Un sistema de control de versions centralitzat on els desenvolupadors treballen amb còpies locals que s'han de sincronitzar amb un servidor central.

Mercurial

Un altre sistema de control de versions distribuït similar a Git, conegut per la seua simplicitat i velocitat.

Perforce

Un sistema de control de versions centralitzat utilitzat principalment en grans entorns corporatius i projectes de gran escala.

6. Bones pràctiques per al Control de Versions

Commits Frequents i xicotets: Fer commits freqüents amb canvis petits per facilitar el seguiment i la reversió de canvis.

Missatges de Commit Descriptius: Escriure missatges de commit clars i descriptius que expliquin el que s'ha canviat i per què.

Ús de Branques: Utilitzar branques per desenvolupar noves característiques, corregir errors i realitzar experiments sense afectar la branca principal.

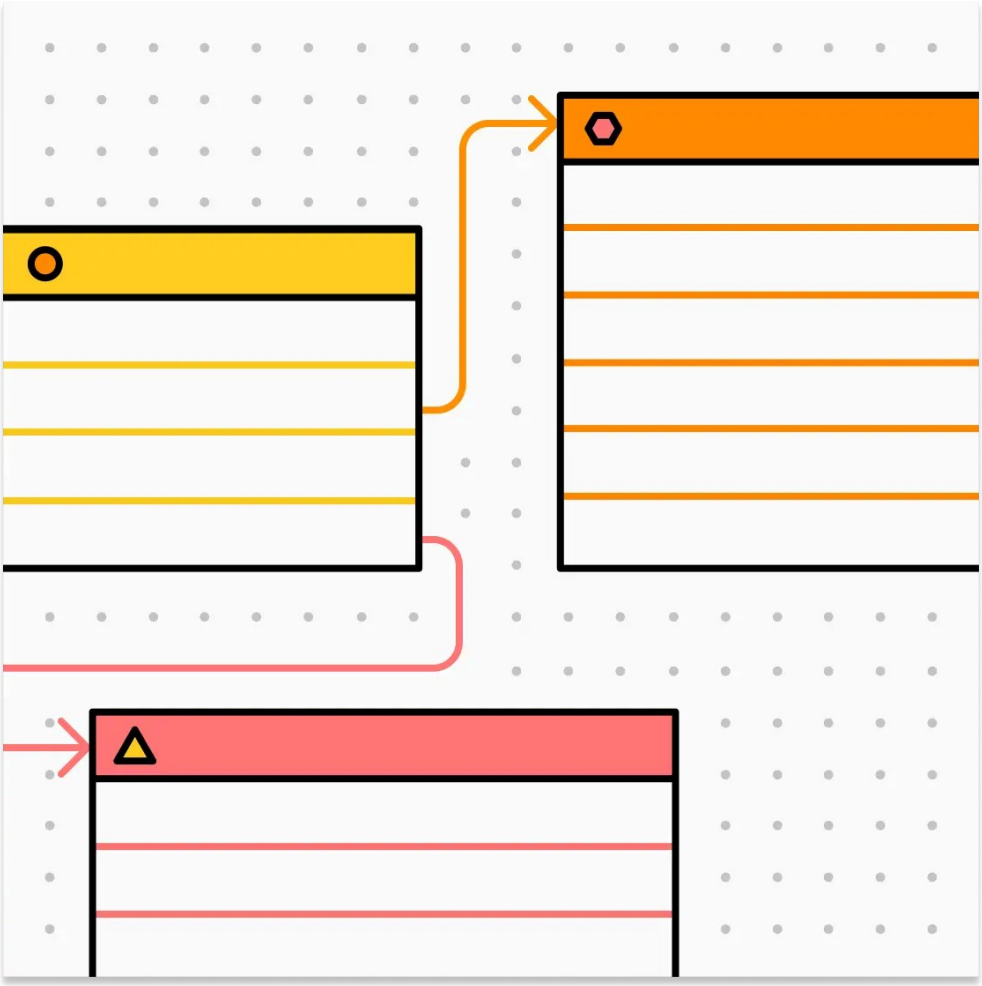
Revisió de Codi: Fer que el codi siga revisat per altres membres de l'equip abans de fusionar-lo a la branca principal per assegurar la qualitat i detectar problemes potencials.

Integració Contínua: Utilitzar pràctiques d'integració contínua per assegurar que els canvis es proven automàticament abans de ser fusionats a la branca principal.

Resolució de Conflictes: Estar preparat per a resoldre conflictes de fusió de manera manual i col·laborativa per assegurar que el codi resultant siga correcte i funcional.

El control de versions és una eina fonamental per a qualsevol projecte de desenvolupament de programari, ja que facilita la col·laboració, la gestió del codi i la seguretat del projecte. Amb una pràctica adequada i l'ús d'eines adequades, els desenvolupadors poden mantenir el codi organitzat, accessible i segur.

Tema 5: Diagrames de classes



1. UML
2. Diagrama de classes. Elements
3. Com dibuixar un diagrama de classes
4. Generació de codi a partir de diagrames de classes i a l'inrevés

1. UML

La UML (Unified Modeling Language) és un llenguatge estàndard per visualitzar, especificar, construir i documentar els components d'un sistema de programari. És molt útil per a dissenyar sistemes orientats a objectes i comunicar idees entre desenvolupadors, analistes i clients.

Tipus principals de diagrames UML

UML inclou diversos tipus de diagrames, agrupats en dues grans categories:

1. Diagrames estructurals (descriuen l'estructura estàtica del sistema)

- Diagrama de classes: mostra les classes, atributs, mètodes i relacions.
- Diagrama d'objectes: instàncies específiques de classes.
- Diagrama de components: com es divideix el sistema en mòduls.
- Diagrama de desplegament: mostra la infraestructura física (servidors, contenidors, etc.).
- Diagrama de paquets: agrupació de classes o components.

2. Diagrames de comportament (descriuen el comportament dinàmic)

- Diagrama de casos d'ús: mostra com els usuaris interactuen amb el sistema.
- Diagrama de seqüència: mostra l'ordre dels missatges entre objectes.
- Diagrama d'activitat: representa fluxos de treball o processos.
- Diagrama d'estats: mostra els estats d'un objecte i com canvien.

2. Diagrama de classes. Elements

Un diagrama de classes és un tipus de diagrama estructural en UML (Unified Modeling Language) que descriu l'estructura del sistema mostrant les classes del sistema, els seus atributs, operacions (mètodes) i les relacions entre les classes. És un dels diagrames més utilitzats en el disseny de sistemes orientats a objectes perquè proporciona una representació estàtica del model de dades del sistema.

Components Principals d'un Diagrama de Classes

1. **Classes**: Representen els objectes del sistema. Cada classe s'indica amb un rectangle dividit en tres parts:
    - **Nom de la classe**: A la part superior.
    - **Atributs**: Al centre, enumerant les propietats o dades de la classe.

    - **Operacions (mètodes)**: A la part inferior, llistant les funcions o mètodes de la classe.
2. **Relacions**: Indiquen com les classes interactuen entre elles. Hi ha diversos tipus de relacions:
    - **Associació**: Una connexió entre dues classes que mostra que existeix una relació entre elles. Pot ser unidireccional o bidireccional.
    - **Agregació**: Una forma d'associació que indica una relació "part-de-tot" on les parts poden existir independentment del tot.
    - **Composició**: Una forma més forta d'agregació on les parts no poden existir independentment del tot.
    - **Generalització (Herència)**: Indica una relació de tipus "és-un" entre una classe base (superclasse) i una classe derivada (subclasse).
    - **Dependència**: Una relació que mostra que un canvi en una classe pot afectar una altra classe que depèn d'ella.
3. **Multiplicitat**: Especifica el nombre d'instàncies d'una classe que poden estar relacionades amb una instància d'una altra classe en una associació. Per exemple, 1..* indica que hi pot haver una o més instàncies.

A continuació es presenta un exemple senzill d'un diagrama de classes per a un sistema de gestió de biblioteca:

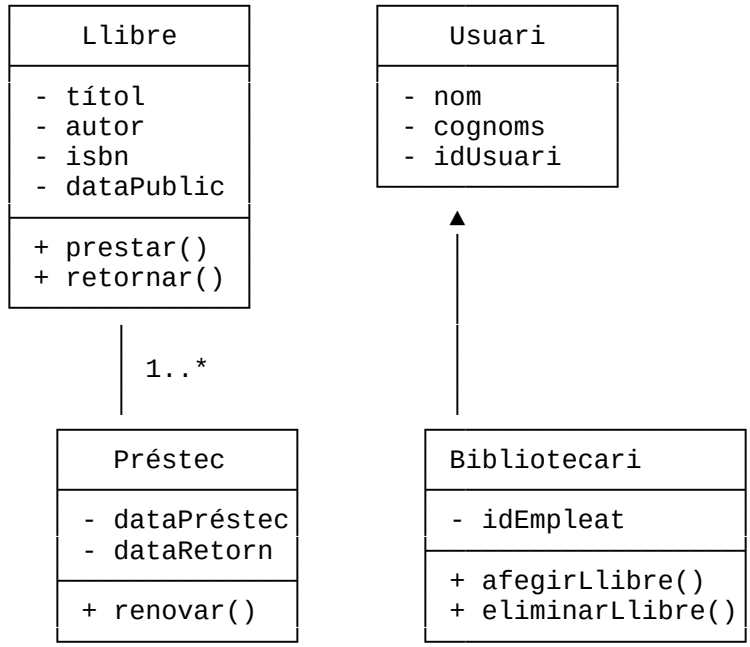
Diagrama de Classes per a un Sistema de Biblioteca

Classes i Relacions

1. **Classe Llibre**
    - **Atributs**:
        - títol: String
        - autor: String
        - isbn: String
        - dataPublicació: Date
    - **Operacions**:
        - prestar()
        - retornar()
2. **Classe Usuari**
    - **Atributs**:
        - nom: String
        - cognoms: String
        - idUsuari: String
    - **Operacions**:
        - registrar()
        - cancelarRegistre()
3. **Classe Bibliotecari** (Hereta de Usuari)
    - **Atributs**:

        - idEmpleat: String
    - **Operacions**:
        - afegirLlibre()
        - eliminarLlibre()
4. **Classe Préstec**
    - **Atributs**:
        - dataPréstec: Date
        - dataRetorn: Date
    - **Operacions**:
        - renovarPréstec()
5. **Relacions**:
    - **Associació** entre Usuari i Préstec: Un usuari pot tenir molts préstecs, però un préstec està associat a un sol usuari (1..*).
    - **Associació** entre Llibre i Préstec: Un llibre pot ser prestat moltes vegades, però un préstec correspon a un sol llibre (1..*).
    - **Herència**: Bibliotecari hereta d'Usuari, indicant que un bibliotecari és un tipus d'usuari.

Diagrama Visual



Explicació del Diagrama

- **Llibre**: La classe Llibre representa un llibre a la biblioteca amb atributs com el títol, l'autor, l'ISBN i la data de publicació. Té operacions per prestar i retornar el llibre.
- **Usuari**: La classe Usuari representa un usuari de la biblioteca amb atributs per al nom, cognoms i un identificador d'usuari. Té operacions per registrar-se i cancel·lar el registre.
- **Bibliotecari**: És una subclasse d'Usuari, amb un atribut addicional idEmpleat i operacions per afegir i eliminar llibres del sistema.

- **Préstec**: La classe Préstec representa el préstec d'un llibre, amb atributs per la data de préstec i la data de retorn. Té una operació per renovar el préstec.
- **Relacions**:
    - Un Usuari pot tenir molts Préstecs (associació 1..*).
    - Un Préstec està associat a un sol Llibre però un Llibre pot estar associat a molts Préstecs (associació 1..*).
    - Bibliotecari és una subclasse d'Usuari, representant una relació d'herència.

Aquest diagrama de classes proporciona una visió clara de la estructura del sistema de gestió de biblioteca, les seves classes principals i les relacions entre aquestes.

Ús del Diagrama de Classes

Els diagrames de classes són útils per a diverses activitats en el desenvolupament de programari:

- **Anàlisi de Requisits**: Per identificar i definir els elements del sistema.
- **Disseny de Sistemes**: Per dissenyar l'arquitectura del sistema.
- **Documentació**: Per documentar la estructura del sistema per a futurs desenvolupadors i mantenidors.
- **Generació de Codi**: Moltes eines de desenvolupament poden generar esborranys de codi a partir de diagrames de classes UML.

Un diagrama de classes és una eina fonamental en el desenvolupament de programari orientat a objectes que permet representar i analitzar la estructura estàtica del sistema.

3. Com dibuixar un diagrama de classes

Per a dibuixar un diagrama de classes UML, pots seguir aquests passos bàsics:

1. Identificar les Classes Principals

Determina les entitats principals del teu sistema. Cada classe hauria de representar un concepte o entitat del teu domini de problemes. Solen ser substantius.

2. Definir els Atributs i Operacions de Cada Classe

Per a cada classe, llista els atributs (propietats) i les operacions (mètodes) que necessitarà. Els atributs són les dades que la classe manté, i les operacions són les funcions que pot realitzar.

3. Establir les Relacions entre Classes

Identifica com les classes estan relacionades entre si. Les relacions poden ser associacions, agregacions, composicions o herències.

4. Representar el Diagrama Visualment

Utilitza una eina de dibuix de diagrames UML o dibuixa a mà si ho prefereixes. Ací tens com representar cada element:

- **Classes**: Representa cada classe amb un rectangle dividit en tres parts: el nom de la classe (part superior), els atributs (part central) i les operacions (part inferior).
- **Relacions**:
    1. **Associació**: Una línia simple que connecta dues classes. Pots afegir una fletxa per indicar la direcció si l'associació és unidireccional.
    2. **Agregació**: Una línia amb un rombe buit en un extrem, indicant la classe que conté (el "tot").
    3. **Composició**: Una línia amb un rombe ple en un extrem, indicant la classe que conté.
    4. **Herència**: Una línia amb un triangle buit apuntant a la superclasse.
    5. **Dependència**: Una línia de punts amb una fletxa que indica la direcció de la dependència.

Eines Recomanades

Hi ha diverses eines que pots utilitzar per a representar diagrames de classes UML visualment, tant en línia com instal·lables.

Eines en línia (sense instal·lació)

PlantUML
- Permet escriure el diagrama en text i genera la imatge automàticament.
- Ideal per integrar amb Git i documentació tècnica.

Lucidchart
- Interfície gràfica intuïtiva.
- Permet col·laboració en temps real.
- Té plantilles UML predefinides.

Draw.io / diagrams.net
- Gratuït i molt flexible.
- Permet guardar els diagrames en local o al núvol.
- Té formes UML disponibles.

Creately
- Interfície visual amb plantilles UML.
- Col·laboració en equip.
- Opció gratuïta amb funcionalitats bàsiques.

UMLetino
- Molt lleuger i ràpid.

- Basat en UMLet, però en versió web.

Eines instal·lables

StarUML
- Potent i professional.
- Compatible amb diversos tipus de diagrames UML.
- Disponible per Windows, macOS i Linux.

Enterprise Architect
- Molt complet, ideal per projectes grans.
- Inclou modelatge de bases de dades, BPMN, etc.

UMLet
- Senzill i ràpid.
- Ideal per fer diagrames ràpids sense complicacions.

Consells

- **Utilitza colors** per distingir diferents tipus de classes o relacions si això ajuda a la claredat.
- **Etiqueta les relacions** amb els seus noms o multiplicitats per fer-les més comprensibles.
- **Mantén el diagrama net i organitzat** per assegurar que siga fàcil de llegir i entendre.

4. Generació de codi a partir de diagrames de classes i a l'inrevés

La generació de codi a partir de diagrames de classes UML (i a l'inrevés) és una funcionalitat molt útil en el desenvolupament de programari, especialment en entorns orientats a objectes. Explicació de com funciona en ambdós sentits:

De diagrama de classes → codi

Aquesta tècnica s'anomena "code generation" o "forward engineering".

Eines que ho permeten:

- StarUML: pots definir classes UML i exportar codi en Java, C++, Python, etc.
- Visual Paradigm: genera codi a partir de models UML.
- Enterprise Architect: molt complet, amb suport per múltiples llenguatges.
- PlantUML + plugins: amb eines addicionals, pot generar esborranys de codi.

Exemple:

```
class Persona:
    def __init__(self, nom, edat):
        self.nom = nom
        self.edat = edat
    def parlar(self):
        pass
    def caminar(self):
        pass
```

Un diagrama amb la classe Persona amb atributs nom i edat, i mètodes parlar() i caminar() pot generar:

De codi → diagrama de classes

Això s'anomena "reverse engineering".

Eines que ho permeten:

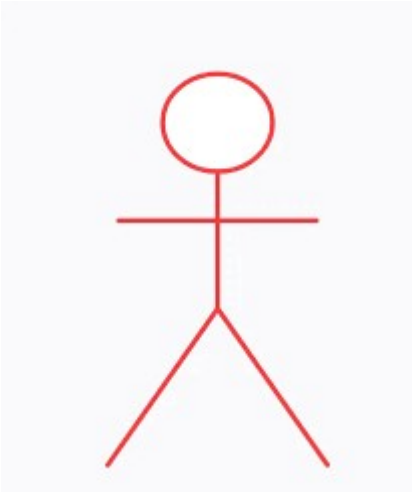
- Pyreverse (inclòs amb pylint): genera diagrames UML a partir de codi Python.
- ObjectAid UML Explorer (plugin per Eclipse): per Java.
- Visual Studio: pot generar diagrames UML a partir de codi C#.
- PlantUML + scripts: pot analitzar codi i generar diagrames textuals.

Exemple: Si tens aquest codi en Python:

```
class Alumne(Persona):
    def __init__(self, nom, edat, curs):
        super().__init__(nom, edat)
        self.curs = curs
    def estudiar(self):
        pass
```

L'eina pot generar un diagrama UML que mostra que Alumne hereta de Persona, amb atributs i mètodes corresponents.

Tema 6: Diagrames de comportament



1. Diagrames de comportaments

Els diagrames de comportament en UML serveixen per descriure com es comporta un sistema, és a dir, com interactuen els seus components, com responen a esdeveniments, i com flueix la lògica del sistema. Són molt útils per entendre el funcionament dinàmic d'una aplicació.

Tipus principals de diagrames de comportament

Diagrama de casos d'ús (Use Case Diagram) Mostra les funcionalitats del sistema des del punt de vista de l'usuari (actors i casos d'ús).

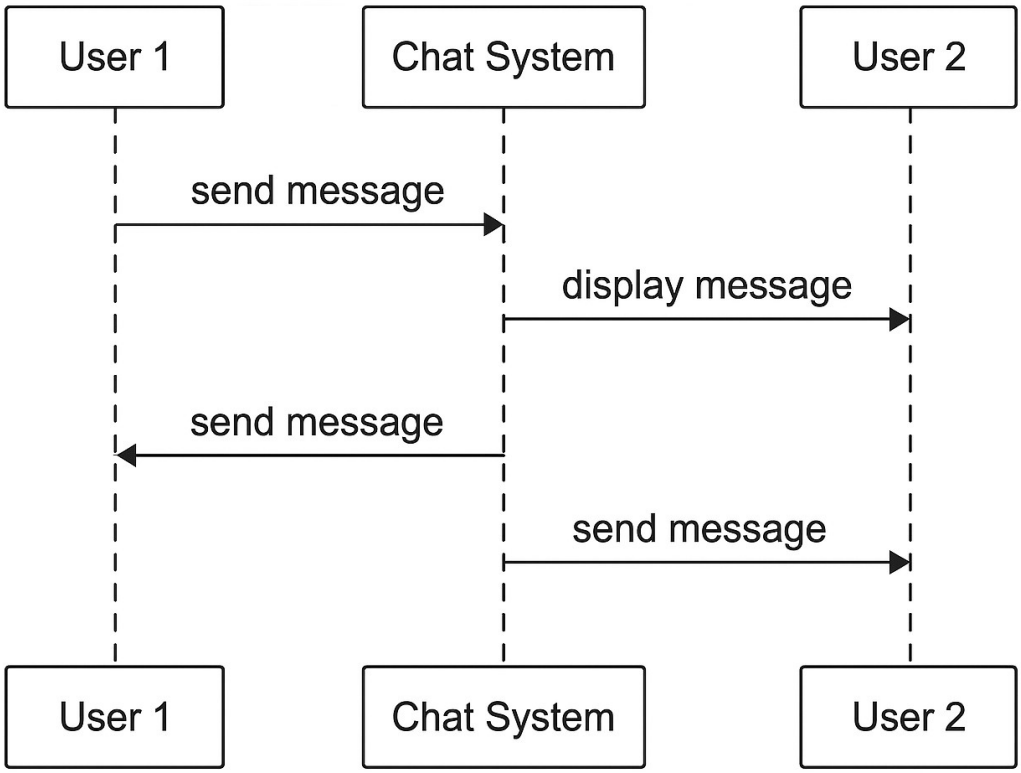
Diagrama de seqüència (Sequence Diagram) Representa la interacció entre objectes al llarg del temps.

Diagrama de comunicació (Communication Diagram) Similar al de seqüència, però s'enfocat en les relacions entre objectes.

Diagrama d'activitat (Activity Diagram) Mostra el flux de treball o processos dins del sistema.

Diagrama d'estats (State Machine Diagram) Representa els estats d'un objecte i les transicions entre ells.

Ací tens un diagrama de seqüència que mostra la interacció entre dos usuaris en un sistema de xat:



- Usuari A envia un missatge.

- Servidor de xat processa i distribueix el missatge.
- Usuari B rep el missatge.

2 Diagrama de casos d'ús

Un diagrama de casos d'ús (use case diagram) és un tipus de diagrama UML que mostra les interaccions entre els usuaris (actors) i el sistema. Aquest tipus de diagrama ajuda a entendre quines funcionalitats ofereix el sistema i com interactuen els usuaris amb aquestes funcionalitats. A continuació, t'explique els elements principals:

Actors: Representen els usuaris o altres sistemes que interactuen amb el sistema. Es dibuixen com a figures de pal.

Casos d'ús: Representen les funcionalitats o serveis que el sistema ofereix als actors. Es dibuixen com a òvals.

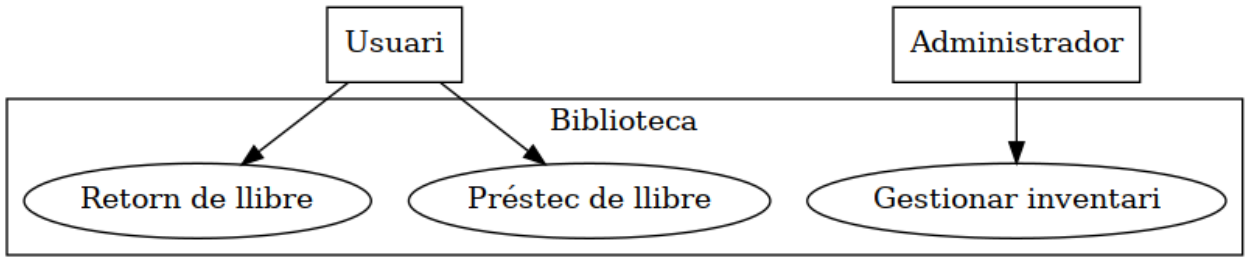
Sistema: És el límit que defineix l'abast del sistema que s'està modelant. Es dibuixa com un rectangle gran que conté els casos d'ús.

Relacions:

- **Associació:** Una línia que connecta un actor amb un cas d'ús, indicant que l'actor participa en el cas d'ús.
- **Inclusió (Include):** Indica que un cas d'ús inclou la funcionalitat d'un altre cas d'ús.
- **Extensió (Extend):** Indica que un cas d'ús pot estendre el comportament d'un altre cas d'ús.
- **Generalització:** Indica una relació de tipus pare-fill entre actors o casos d'ús.

Exemple bàsic d'un diagrama de casos d'ús

Imagina que estem modelant un sistema de biblioteca. Els actors principals podrien ser "Usuari" i "Administrador". Els casos d'ús podrien ser "Préstec de llibre", "Retorn de llibre" i "Gestionar inventari". El diagrama podria semblar-se a això:



Descripció del diagrama

Actors:

"Usuari" representa els usuaris de la biblioteca que poden prendre llibres en préstec i retornar-los.

"Administrador" representa els empleats de la biblioteca que gestionen l'inventari.

Casos d'ús:

Préstec de llibre: Un usuari pot agafar un llibre en préstec.

Retorn de llibre: Un usuari pot retornar un llibre prestat.

Gestionar inventari: Un administrador pot afegir, eliminar o actualitzar informació sobre els llibres disponibles.

Relacions:

L'actor "Usuari" està associat amb els casos d'ús "Préstec de llibre" i "Retorn de llibre".

L'actor "Administrador" està associat amb el cas d'ús "Gestionar inventari".

Aquest és un exemple molt simple, però els diagrames de casos d'ús poden ser molt més complexos, amb múltiples actors i relacions entre els casos d'ús. La clau és mantenir el diagrama clar i fàcil d'entendre per comunicar de manera efectiva les funcionalitats del sistema i les interaccions amb els usuaris.

3 Diagrames d'interacció

Els diagrames d'interacció en UML s'utilitzen per descriure com els objectes interactuen entre ells en termes de l'enviament de missatges. Hi ha diferents tipus de diagrames d'interacció, però els dos més comuns són els ***diagrames de seqüència*** i els ***diagrames de comunicació***.

3.1 Diagrames de seqüència

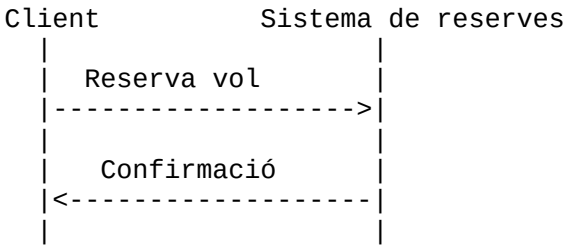
Els diagrames de seqüència mostren la interacció entre els objectes en una seqüència temporal. Aquests diagrames representen els objectes alineats a la part superior i els missatges enviats entre ells al llarg d'una línia de temps vertical.

Elements principals:

- **Actors/Objectes**: Els participants de la interacció, representats per rectangles a la part superior del diagrama.
- **Línia de vida**: Una línia vertical que baixa des de cada objecte, representant el temps que l'objecte existeix durant la interacció.
- **Missatges**: Fletxes que van d'un objecte a un altre, indicant la comunicació entre ells. Les fletxes poden ser sincròniques (amb una punta de fletxa plena) o asincròniques (amb una punta de fletxa oberta).

Exemple bàsic:

Imagina un sistema de reserves de vol. Els actors podrien ser "Client" i "Sistema de reserves". El diagrama podria ser així:



Altre exemple és diagrama de seqüència que mostra la interacció entre dos usuaris en un sistema de xat vist en el punt 1:

3.2 Diagrames de comunicació

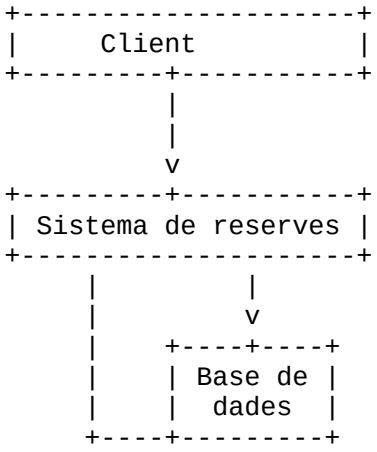
Els diagrames de comunicació (anteriorment coneguts com diagrames de col·laboració) mostren les interaccions entre objectes en una disposició gràfica on es posa èmfasi en la col·laboració entre els objectes per a aconseguir una funcionalitat.

Elements principals:

- **Actors/Objectes:** Representats per rectangles.
- **Enllaços:** Línies que connecten els objectes, indicant que poden enviar-se missatges entre ells.
- **Missatges:** Etiquetes a les línies d'enllaç que descriuen els missatges enviats. Els missatges es numeraran per mostrar l'ordre de l'enviament.

Exemple bàsic:

Seguint amb el sistema de reserves de vol:



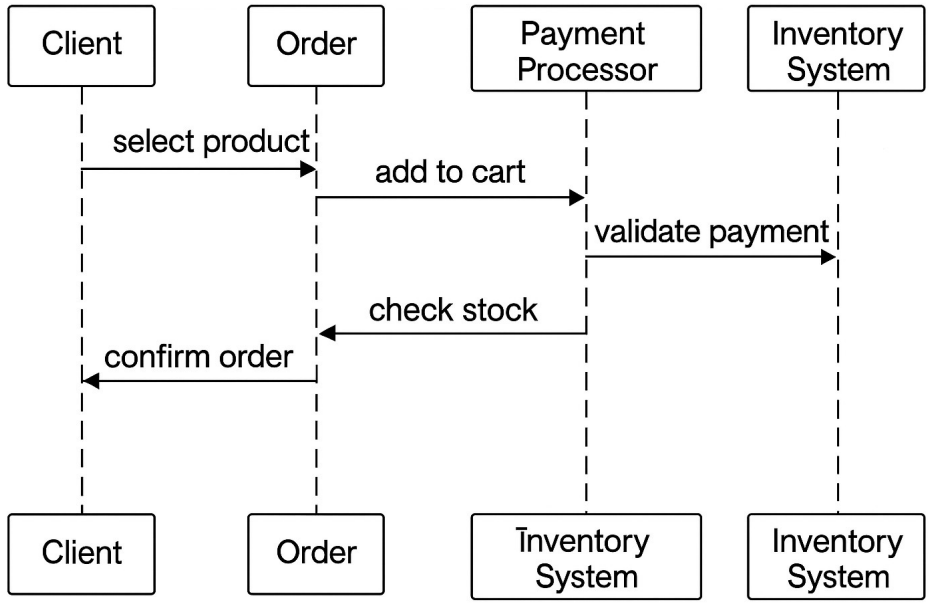
- El "Client" envia un missatge "Reserva vol" al "Sistema de reserves".
- El "Sistema de reserves" pot enviar missatges a la "Base de dades" per obtenir o actualitzar informació.

El iagrama de seqüència: Enfocat en l'ordre temporal dels missatges i el diagrama de comunicació: Enfocat en la col·laboració estructural entre els objectes.

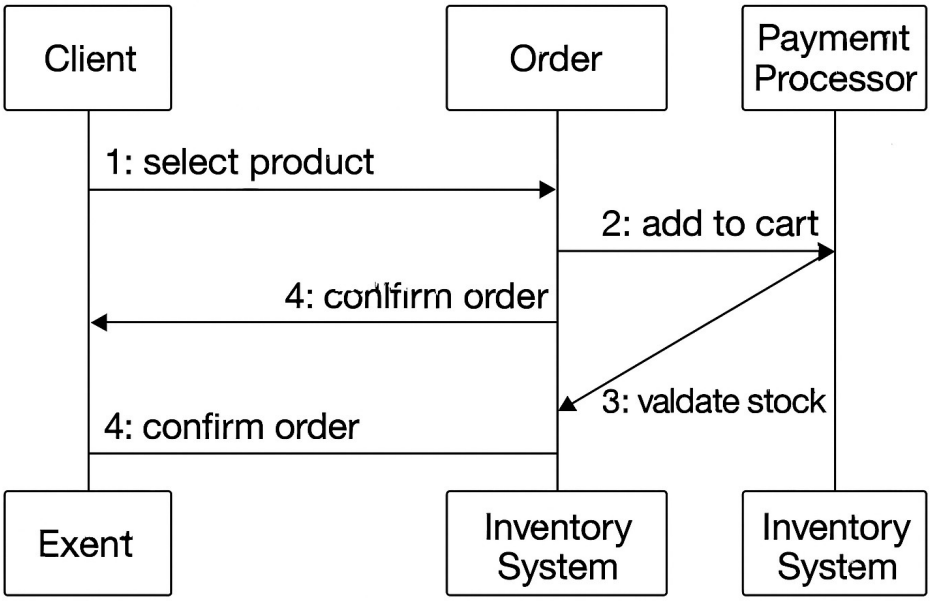
Un cas pràctic

Imagina que estàs dissenyant un sistema de comerç electrònic. Un diagrama de seqüència podria mostrar com un client fa una comanda: des de la selecció d'un producte, passant pel pagament, fins a la confirmació de la comanda. Un diagrama de comunicació podria mostrar com els diferents components del sistema (com el carret de compres, el sistema de pagament, i la base de dades d'inventari) interactuen per completar la comanda.

Diagrama de seqüència:



I diagrama de comunicació:



Aquests diagrames són molt útils per visualitzar i planificar les interaccions complexes dins d'un sistema, ajudant als desenvolupadors i analistes a entendre i millorar el disseny del sistema.

4. Eines per a diagrames de comportament

Hi ha moltes. Eines recomanades:

PlantUML

Basat en text → escrius el diagrama amb codi senzill i es genera automàticament.

Ideal per a documentació versionada (per exemple, dins de repositoris Git).

Compatible amb editors com VS Code, IntelliJ, Eclipse.

StarUML

Eina de pagament però molt potent i intuïtiva.

Dibuix gràfic amb suport complet per a UML.

Lucidchart / Draw.io (diagrams.net)

Lucidchart és online i molt polida, però limitada en la versió gratuïta.

Draw.io (ara diagrams.net)

És completament gratuïta, molt flexible i fàcil d'usar.

Enterprise Architect

Orientada a projectes grans i complexos.

Molt usada en entorns professionals, però és de pagament i més pesada.

Visual Paradigm

Potent, amb versions gratuïtes per a estudiants.

Té molts tipus de diagrames UML i BPMN.

Què busques?

- Si només vols fer esquemes ràpids i compartir-los fàcilment, Draw.io (diagrams.net) és la millor opció gratuïta.
- Si busques integració amb codi i documentació, et recomano PlantUML.

- Si treballes en un projecte gran i professional, probablement StarUML o Enterprise Architect.
