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

Xavi Blanes 2024

índex del llibre

Tema 1: Desenrotllament de programari

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

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

1. Concepte i aplicació dels entorns de desenrotllament
    - 1.1 Conceptes Clau dels Entorns de Desenvolupament
    - 1.2 Aplicacions dels Entorns de Desenvolupament
2. Classificació i funcions dels IDE
    - 2.1 Classificació dels Entorns de Desenvolupament
    - 2.2 Funcions d'un Entorn de Desenvolupament
    - 2.3 Exemples de Funcions en IDEs Populars
3. Avaluació dels distints IDEs web: Visual Studio Code, Sublime Text, React i Angular

Tema 3: Disseny i realització de proves

1. Introducció
2. Les proves en el cicle de vida d'un projecte
3. Procediments, tipus i casos de proves
4. Eines per a la realització de proves
5. Mètriques de qualitat de programari

Tema 4: Optimització i documentació

1. Introducció
2. Reflacció
3. Eines d'ajuda a la refacció
4. Control de versions
5. Eines populars de control de versions
6. Bones pràctiques per al control de versions
7. Utilització de Git
8. Utilització de Github

Tema 5: Elaboració de diagrames de classes

1. Diagrama de classes
2. Elements d'un diagrama de classes
3. Com dibuixar un diagrama de classes
4. Bones pràctiques en la construcció del diagrama de classes
5. Diagrama d'objectes
6. Diagrama d'objectes Vs Diagrama de classes
7. Eines de disseny de diagrames.
8. Generació de codi a partir de diagrames de classes i a l'inrevés

Tema 6: Elaboració de diagrames de comportament

1 Diagrames de comportament
2 Diagrama de casos d'ús
3 Diagrames d'interacció
    3.1 Diagrames de seqüència
    3.2 Diagrames de comunicació
    3.3 Diferències clau
4 Eclipse Modeling Framework (EMF)

Tema 1: Desenrotllament de programari

---

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

---

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 un 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 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 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.

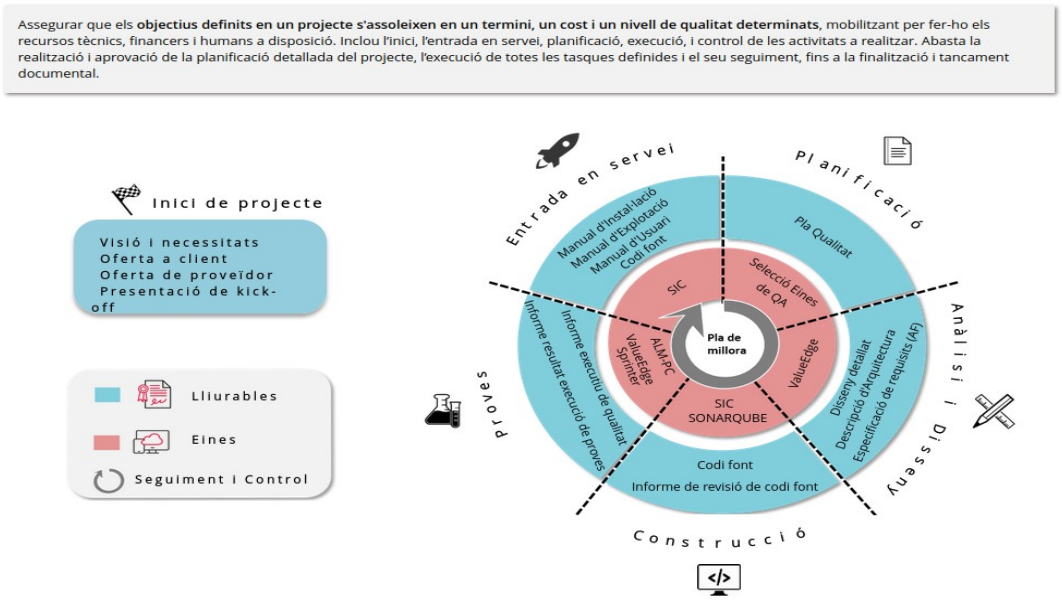
SQL: La solvència, la versatilitat i la consistència de les bases de dades relacionals i de SQL fan d'aquest llenguatge un dels més importants útils i demandats per analitzar dades.

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

Cicle de vida del software



Planificació

Pla de qualitat
El pla de qualitat és un document en el que es detalla com ha de ser el procés que garantitze la qualitat dels projectes.

Anàlisi i Disseny

Anàlisi de Requisits
L'objectiu d'aquesta activitat és fer l'anàlisi dels requisits, transformant les necessitats establertes a alt nivell (en el document de Visió i Necessitats i/o en la pròpia oferta) a requisits del sistema amb major detall.

Elaborar prototip
L'objectiu del prototip és permetre validar que s'han entés correctament els requisits, definint com seran les pantalles, formats de sortida d'informes, la navegació i l'estructura general de l'aplicació. A partir del prototipatge es poden obtenir nous requisits o refinar els existents.

Elaborar la descripció de l'arquitectura
Descriure les relacions, dependències i interaccions entre el sistema i el seu entorn (usuaris, sistemes i entitats externes amb les que interactua).

Elaborar el disseny detallat
El disseny detallat està orientat a descriure el detall intern de la interacció de determinades transaccions de negoci o dels requisits no funcionals, per representar de forma exhaustiva com s'ha d'implementar.

Elaborar el pla mestre de proves
Elaborar el document que recull l'enfoc i estratègia de tots els nivells de proves (unitàries, integració, sistema, acceptació) i tipus que es realitzaran en el projecte/solució.

Construcció

Desenvolupar els elements del programari
Desenvolupar els elements del programari, proves unitàries i proves d'integració entre components
[Informe de revisió del codi font]

Eina de revisió del codi font (Sonarqube)
Solució SaaS que permet analitzar la qualitat del codi. Els resultats extrets estan orientats a mesurar, analitzar i verificar la qualitat i seguretat del nostre codi font, basant-se en una classificació com la definida per la ISO 25000.

Proves

Elaborar el pla mestre de proves
Elaborar el document que recull l'enfoc i estratègia de tots els nivells de proves (unitàries, integració, sistema, acceptació) i tipus que es realitzaran en el projecte/solució.

Especificar els casos de prova
Descriure el detall dels passo, paràmetres, configuracions, i resultats esperats de cadascun dels casos de prova

Automatitzar les proves
L'automatització de les proves ha de considerar de forma més detallada possible com es portarà a terme l'automatització dels casos, usant l'enfoc més adequat i acordat.

Executar les proves
Realitzar la correcta configuració, execució i anàlisi dels resultats, així com registrar els defectes si és necessari.

Eina d'execució de proves manuals
Sprinter (Eina d'execució de proves manuals i exploratòries)

Entrada en servei

Manual explotació
Inclou les seccions dels processos d'explotació, les contingències de les dades, els procediments d'actualització i manteniment i la gestió i administració d'usuaris.

Manual instal·lació
Recull els passos necessaris per la seva paquetització, instal·lació i configuració d'una solució per poder iniciar les proves o per la seva posada en marxa.

Manual usuari
Manuals de suport a l'usuari per la comprensió i ús del programari

7. Estructures d'equips de treball

7.1 Estructura tradicional

Un equip de desenvolupament de programari està format per moltes persones amb funcions diferents i, per tant, amb habilitats diferents. I són precisament aquestes capacitats les que porten al compliment dels objectius.

Per això, com que és un equip molt complex, t'hem portat els 6 principals rols dins del món del desenvolupament programari perquè pugues distingir-los. Quines funcions té cadascú i quines habilitats es busquen.

CAP DE PROJECTE

És la persona que gestiona el bon funcionament del projecte, qui controla i administra els recursos (tant personals com econòmics) per tal de complir el pla i l'objectiu definit. S'encarreguen que tot funcioni segons allò establert, resoldre desviacions al pla, i fer que els diferents equips del projecte se sincronitzen i treballen junts (distribució de tasques, flux d'activitats, tasques administratives, contracte amb el client, direcció i control) . A més, és la cara visible davant del client, que l'informa dels avenços i l'estat del projecte. La seva missió és complir les expectatives del client.

Perfil professional. Experiència prèvia en gestió de projectes del tipus que cerques i gestió dequips. Ha de tenir tant el coneixement tècnic (conèixer la tecnologia i els recursos amb què treballarà), com l'habilitat de gestionar persones i altres recursos. Entre les habilitats que has de cercar a més de la gestió, la capacitat analítica i la de comunicació per desenvolupar un lideratge efectiu. Ha de ser capaç de traduir el projecte en un procés, preveient desviacions i possibles camins fins a arribar a lobjectiu.

ANALISTA DE PROGRAMARI

Intervé a les primeres fases del projecte on es realitzen les especificacions de les necessitats o la problemàtica del client, des del general al detall. Com a expert en el problema del client, l'analista de programari treballa juntament amb aquest per definir correctament les especificacions tècniques del producte. A més, té la missió de traduir aquests problemes del client en especificacions amb sentit per a la resta de l'equip que després desenvoluparà el producte.

Perfil professional. A més a més del background tècnic, hauràs d'avaluar la capacitat d'anàlisi, l'orientació al detall i el focus en resultats. L'analista de programari haurà de tenir una bona capacitat de comunicació per saber traduir els requeriments del client a instruccions per a l'equip i, a més, treballar colze a colze amb el client.

ARQUITECTE DE PROGRAMARI

És la persona o persones amb prou coneixement tècnic del producte o servei per buscar la seva aplicació tècnica a les necessitats del client. Té com a missió crear, durant tot el procés de desenvolupament, la documentació que recull els requisits (juntament amb l'analista de programari), i serà ell qui centralitze les decisions tècniques sobre els problemes que aniran sorgint, assegurar-ne la qualitat, i millorar-ne contínuament la arquitectura.

Un arquitecte de programari tindrà en compte tant els requisits tècnics i funcions com els requisits no funcionals. Els definirà amb l'analista i els prioritzarà. Després pensarà com es resoldran aquests problemes i definirà l'arquitectura. És a dir, la introducció de l'estructura, les directrius, els principis i el lideratge dels aspectes tècnics del projecte.

En aquest procés, a més, haurà de seleccionar la tecnologia que s'utilitzarà, tenint en compte diversos factors com el cost, les llicències, la relació amb els proveïdors, l'estratègia, la política d'actualització…

Com que aquest rol és un dels més importants i menys coneguts, l'arquitecte de programari ha de ser un perfil amb dots de facilitador, formador i líder.

Perfil professional. A més d'un bon coneixement tècnic (mentalitat tècnica, atenció al detall, experiència en desenvolupament de programari…), l'arquitecte de programari ha de tenir una visió global no només del projecte, sinó de l'ecosistema tecnològic. Com diem a dalt, ha de tenir un rol de líder, de facilitador i de suport de l'equip, de control i supervisió (metodologies àgils), així com de millora contínua.

DESENVOLUPADOR DE PROGRAMARI

El desenvolupador de programari serà qui reba la documentació creada per l'arquitecte i l'analista i qui implementee el producte segons aquesta.

Aquest perfil coneix i és capaç de fer totes les tasques de desenvolupament, però se cenyeix a la implementació i delega altres funcions (com la de programació, el testeig, la supervisió o el manteniment) a altres membres de l'equip. La seva responsabilitat és més àmplia, i té com a missió que tots els aspectes de la implementació del projecte funcionen bé.

La seva diferència amb els analistes pot ser molt subtil, ja que el desenvolupador pot participar en la definició del producte, en les especificacions i requeriments, en el disseny

i la millora de prototips, o fins i tot l'anàlisi del cost i beneficis d'escollir un tipus d'arquitectura o una altra.

Ací hi ha un debat històric, i és la diferència entre un desenvolupador i un enginyer de programari. La “regla d'or” quan es tracta de diferenciar: “un enginyer de programari pot ser un desenvolupador de programari, però un desenvolupador no pot ser enginyer de programari”. “Grosso modo”, un desenvolupador és algú que treballa amb un programa, mentre que un enginyer treballa normalment amb la creació del mateix programa. Un enginyer desenvolupa i manté la plataforma on els desenvolupadors després crear el programa mateix.

Perfil professional. Depèn del que desenvoluparàs (si és una web, un SaaS, una aplicació, una aplicació mòbil…). El seu treball és molt ampli, així que tingues molt present el que desenvoluparàs. Primer, el background tècnic (bé enginyeria informàtica o tècnic superior). El Desenvolupador haurà de dominar els llenguatges de programació (un o més), conèixer de Bases de Dades, serveis web, aplicacions orientades a serveis, protocols i llenguatges de comunicació, metodologies àgils, eines de control…

A més, hauràs de cercar una persona amb iniciativa, capacitat d'autoaprenentatge, capacitat d'anàlisi i atenció al detall. També haurà de treballar en equip, de manera que la seva capacitat de col·laboració és important.

PROGRAMADOR

És l'encarregat de traduir l'especificació del sistema en codi. Tot i que el desenvolupador també pot “picar codi”, els programadors es dediquen exclusivament a això. Aquesta persona ha de conèixer els diferents llenguatges de programació. I a més, s'encarrega de depurar els errors, implementar noves funcionalitats o mantenir de manera general les aplicacions quan ho necessiten. Això no vol dir que un programador no puga conèixer pressupostos, planejament o requeriments. Dependrà de l’experiència.

Perfil professional. A més dels requisits formatius o de la capacitat autodidacta, hauràs de cercar una persona flexible, amb capacitat per treballar en equip, proactiu, orientació al detall i la qualitat del codi. Un bon programador coneixerà diversos llenguatges de programació, diversos frameworks, CMS i metodologies de desenvolupament.

TESTER

S'encarregarà d'assegurar que els requisits definits per l'arquitecte de programari es compleixen a la implementació del producte o servei realitzada pels desenvolupadors i/o programadors. Per això, serà responsable d'aplicar diferents mètodes de testeig al costat dels programadors. Informarà de tots els errors trobats durant la fase de proves.

Perfil professional. A més de tenir una enginyeria informàtica o de sistemes, o ser tècnic superior en desenvolupament de programari, el tester haurà de conèixer de metodologies i models de qualitat de programari, així com eines per a l'execució i el seguiment de testing (volum, Smoke Test, performance, funciona, tècnic…). Ha de tenir experiència en automatització i scripting, principalment.

QUALITY ASSURANCE (QA)

Encara que quan parlem de Tester i Quality Assurance (AQ) com el mateix, un i altre perfil poden tenir diferències notables. “Un tester s'encarrega de trobar errors, però un QA no només els troba, sinó que ajuda a prevenir-los”. Per tant, un QA sassegura de la qualitat del programari durant totes les seves fases, no només en la fase de proves com un tester. Es podria dir que és una evolució d'aquest al qual s'han afegit tasques per assegurar la qualitat global del projecte i del producte o servei resultant.

A més, un QA pot participar a la definició del producte, a la definició dels passos de la integració contínua, i/o a la configuració d'eines. No es pot entendre una bona integració contínua sense tests automàtics. Té una visió més horitzontal i pot detectar problemes potencials en la implementació i fins i tot prevenir-los. És recomanable, a més, que conega de metodologies àgils.

Perfil professional. Enginyer informàtic i/o sistemes amb experiència prèvia a testing. No cal que sàpiga de programació, però sí que és altament recomanable. Tant si és un QA Funcional, com Tècnic, necessitarà coneixements en el disseny i l'execució de proves, com en el seguiment i l'optimització dels estàndards de qualitat. A més, el QA participarà de tot el procés, treballant colze a colze amb el Project Manager, per la qual cosa necessitarà saber de metodologies àgils i assegurar-se que compleixen els estàndards tant tècnics com de negoci. Per això, haurem de comprovar la seua capacitat de anàlisi i visió estratègica, així com la seua capacitat per anticipar-se a errors o problemes futurs.

2. Estructura Agile/Scrum

L'estructura Agile es basa en equips petits i autònoms que treballen en iteracions curtes (sprints).

- **Scrum Master**: Facilita el procés Agile i elimina obstacles per a l'equip.
- **Product Owner**: Responsable de la visió del producte i la priorització del treball.
- **Equip de Desenvolupament**: Equips multidisciplinaris autogestionats que inclouen desenvolupadors, testers i altres rols necessaris.
- **Stakeholders**: Usuaris finals i altres parts interessades que proporcionen feedback.

3. Estructura DevOps

El model DevOps combina desenvolupament (Dev) i operacions (Ops) en un sol equip per millorar la col·laboració i la velocitat de desplegament.

- **Desenvolupadors**: Escriuen i mantenen el codi.
- **Enginyers de DevOps**: Automatitzen processos, integren i despleguen el codi.
- **Admins de Sistema/Enginyers d'Operacions**: Gestionen la infraestructura i asseguren la disponibilitat del sistema.
- **Enginyers de Qualitat**: Realitzen proves contínues per assegurar la qualitat del codi.

4. Estructura Cross-funcional

Els equips cross-funcionals inclouen membres amb diferents habilitats que treballen junts en diverses fases del desenvolupament.

- **Equip Cross-funcional**: Inclou desenvolupadors, dissenyadors, enginyers de qualitat, especialistes en UI/UX, etc.

- **Gestor de Projecte**: Coordina les activitats i assegura la comunicació entre els diferents membres de l'equip.
- **Clients/Usuaris Finals**: Proporcionen feedback constant al llarg del desenvolupament.

5. Estructura de Comunitat de Pràctica (CoP)

Aquesta estructura no és per a projectes específics, sinó per fomentar l'aprenentatge i la col·laboració entre persones amb interessos similars dins de l'organització.

- **Comunitats de Pràctica**: Grups informals que es reuneixen per compartir coneixements, eines i tècniques.
- **Facilitador de la CoP**: Organitza les reunions i modera les discussions.
- **Membres de la CoP**: Professionals de diferents equips que participen voluntàriament.

6. Estructura de Desenvolupament Basat en Components

Aquesta estructura se centra en equips que desenvolupen components o mòduls específics del sistema.

- **Gestor de Projecte Global**: Coordina els diferents equips de components.
- **Equips de Components**: Cada equip és responsable d'un component específic del sistema. Pot incloure desenvolupadors, testers, dissenyadors, etc.
- **Integrador de Sistema**: Assegura que els components es combinin correctament en el sistema global.

Estructures Híbrides

Moltes organitzacions utilitzen estructures híbrides, combinant elements de diferents models per adaptar-se millor a les seves necessitats específiques. Per exemple, un equip pot utilitzar metodologies Agile dins de cada equip de components mentre manté una estructura global més tradicional per a la coordinació.

Podem dir que la elecció de l'estructura de l'equip depèn de factors com la mida de l'organització, la naturalesa del projecte, la cultura empresarial i les eines disponibles. És fonamental que l'estructura triada fomenti la col·laboració, la comunicació efectiva i la qualitat del producte final. Adaptar i ajustar les estructures segons les necessitats canviants del projecte i l'equip pot portar a un desenvolupament de programari més eficient i satisfactori.

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. Concepte i aplicació dels entorns de desenrotllament
    - 1.1 Conceptes Clau dels Entorns de Desenvolupament
    - 1.2 Aplicacions dels Entorns de Desenvolupament
2. Classificació i funcions dels IDE
    - 2.1 Classificació dels Entorns de Desenvolupament
    - 2.2 Funcions d'un Entorn de Desenvolupament
    - 2.3 Exemples de Funcions en IDEs Populars
3. Avaluació dels distints IDEs web:
    - Visual Studio Code,
    - Sublime Text,
    - React i
    - Angular

---

1. Concepte i aplicació dels entorns de desenrotllament

Els entorns de desenvolupament, també coneguts com IDEs (Entorns de Desenvolupament Integrats), són eines essencials per a la programació de programari. Aquests entorns proporcionen un conjunt de funcions i eines que ajuden els desenvolupadors a escriure, provar i depurar el codi de manera més eficient. A continuació, es detallen els conceptes clau i aplicacions dels entorns de desenvolupament:

1.1 Conceptes Clau dels Entorns de Desenvolupament

1. **Editor de Codi:**
    - És una part fonamental de qualsevol IDE. Permet escriure i editar el codi font. Els editors de codi solen oferir característiques com ressaltat de sintaxi, autocompletat, i suggeriments de codi per facilitar la programació.
2. **Depurador (Debugger):**
    - Eina que permet als desenvolupadors executar el codi pas a pas per identificar i corregir errors (bugs). Els depuradors solen oferir funcions com punts de ruptura (breakpoints), inspecció de variables, i execució condicional.
3. **Compilador/Intèrpret:**
    - Molts IDEs integren compiladors o intèrprets que converteixen el codi font en codi executable. Això permet als desenvolupadors provar els seus programes directament des de l'IDE.
4. **Control de Versions:**
    - Integració amb sistemes de control de versions com Git, que permeten gestionar les versions del codi i col·laborar amb altres desenvolupadors.
5. **Gestor de Projectes:**
    - Eines per organitzar i gestionar els fitxers i recursos del projecte. Això inclou estructures de carpetes, configuracions de compilació, i altres metadades del projecte.
6. **Extensions i Plugins:**
    - Capacitat per afegir funcionalitats addicionals a l'IDE mitjançant extensions o plugins. Això permet personalitzar l'entorn de desenvolupament segons les necessitats específiques del projecte o del desenvolupador.

1.2 Aplicacions dels Entorns de Desenvolupament

1. **Desenvolupament d'Aplicacions Web:**
    - IDEs com Visual Studio Code, Sublime Text, i Atom (discontinuïtat) són molt populars per al desenvolupament web gràcies a les seves extensions per a HTML, CSS, JavaScript, i frameworks com React o Angular.
2. **Desenvolupament de Programari d'Escriptori:**
    - Entorns com Visual Studio (per a aplicacions .NET) o IntelliJ IDEA (per a aplicacions Java) són àmpliament utilitzats per desenvolupar aplicacions d'escriptori.

3. **Desenvolupament Mòbil:**
    - IDEs com Android Studio (per a aplicacions Android) i Xcode (per a aplicacions iOS) proporcionen eines específiques per al desenvolupament de software mòbil, incloent emuladors i eines de depuració específiques per a dispositius mòbils.
4. **Desenvolupament de Jocs:**
    - Motors de jocs com Unity o Unreal Engine inclouen els seus propis entorns de desenvolupament que permeten crear i depurar jocs, amb eines per al disseny gràfic, la física, i la interacció.
5. **Desenvolupament de Software Empresarial:**
    - Eines com Eclipse són molt populars per al desenvolupament de software empresarial gràcies a la seva capacitat per gestionar projectes grans i complexos, així com per la seva integració amb eines empresarials com servidors d'aplicacions i bases de dades.

Podem dir que els entorns de desenvolupament són eines indispensables que ajuden els desenvolupadors a ser més productius, reduir errors, i gestionar projectes de manera més eficient. La seva selecció depèn sovint del tipus de projecte i del llenguatge de programació utilitzat.

2. Classificació i funcions dels IDE

2.1 Classificació dels Entorns de Desenvolupament

Els entorns de desenvolupament es poden classificar segons diversos criteris:

Segons la Tipologia de Programació:

Generals: IDEs que poden ser utilitzats per a diversos llenguatges de programació. Exemples: Visual Studio Code, Sublime Text, Atom.

Específics: IDEs dissenyats per a un llenguatge de programació específic. Exemples: PyCharm (Python), RubyMine (Ruby), RStudio (R).

Segons la Plataforma de Desenvolupament:

Web: IDEs orientats al desenvolupament d'aplicacions web. Exemples: Brackets, WebStorm.
Mòbil: IDEs específics per al desenvolupament d'aplicacions mòbils. Exemples: Android Studio (Android), Xcode (iOS).
Escriptori: IDEs per a aplicacions d'escriptori. Exemples: Visual Studio (Windows), IntelliJ IDEA (Java).

Segons el Model de Llicència:

Codi Obert: IDEs amb codi font accessible i modificable. Exemples: Eclipse, NetBeans.
Propietari: IDEs amb llicència propietària i generalment de pagament. Exemples: Visual Studio, JetBrains IDEs.

Segons el Mode d'Execució:

Locals: IDEs que s'executen en màquines locals. Exemples: Visual Studio, IntelliJ IDEA.

Basats en el Núvol: IDEs que s'executen en el núvol i poden ser accedits des de qualsevol lloc amb connexió a Internet. Exemples: AWS Cloud9, GitHub Codespaces.

2.2 Funcions d'un Entorn de Desenvolupament

Els entorns de desenvolupament ofereixen una varietat de funcions que faciliten el procés de programació. A continuació, es descriuen les funcions principals:

Editor de Codi: Proporciona eines per escriure i editar codi de manera eficient. Inclou funcions com ressaltat de sintaxi, autocompletat, i suggeriments de codi.

Depuració (Debugging): Permet executar el codi pas a pas per identificar i corregir errors. Inclou punts de ruptura (breakpoints), inspecció de variables, i execució condicional.

Compilació i Execució: Inclou eines per compilar (en el cas de llenguatges compilats) i executar el codi directament des de l'IDE. Això facilita la prova ràpida de canvis.

Control de Versions: Integració amb sistemes de control de versions com Git, que permeten fer seguiment dels canvis en el codi, col·laborar amb altres desenvolupadors, i gestionar branques de desenvolupament.

Gestió de Projectes: Eines per organitzar i gestionar els fitxers del projecte. Això inclou estructures de carpetes, configuracions de compilació, i altres metadades del projecte.

Refactorització: Eines per millorar i reestructurar el codi sense canviar el seu comportament extern. Inclou funcions com renombrar variables, extraure funcions, i canviar la signatura de mètodes.

Integració Contínua: Algunes IDEs integren eines d'integració contínua (CI) que permeten la prova automàtica i la compilació del codi cada vegada que es realitza un canvi.

Plugins i Extensions: Capacitat per afegir funcionalitats addicionals mitjançant plugins o extensions, permetent personalitzar l'IDE segons les necessitats específiques del desenvolupador o del projecte.

Emuladors i Simuladors: Particularment útils en el desenvolupament mòbil, aquests permeten provar aplicacions en entorns simulats de dispositius mòbils sense necessitat de maquinari físic.

Interfície Gràfica d'Usuari (GUI): Algunes IDEs proporcionen eines per al disseny d'interfícies gràfiques d'usuari (GUI), permetent dissenyar visualment components d'interfície per a aplicacions d'escriptori o mòbils.

2.3 Exemples de Funcions en IDEs Populars

Visual Studio Code:

Editor de codi amb ressaltat de sintaxi, autocompletat, i gran quantitat d'extensions.
Integració amb Git i altres sistemes de control de versions.
Depuració integrada per diversos llenguatges de programació.

IntelliJ IDEA:

Eines avançades de refactorització i suggeriments de codi.
Integració amb sistemes de compilació com Maven i Gradle.
Suport per a múltiples llenguatges de programació, especialment Java.

Android Studio:

Emuladors per provar aplicacions Android en diferents dispositius virtuals.
Editor de codi amb suggeriments específics per a Android.
Eines de depuració avançades per aplicacions mòbils.

En resum, els entorns de desenvolupament ofereixen una gamma extensa de funcions que ajuden els desenvolupadors a ser més eficients i productius, proporcionant totes les eines necessàries per al cicle complet de desenvolupament de programari.

3. Avaluació dels distints IDEs web:

Visual Studio Code,
Sublime Text,
React i
Angular

Tema 3: Disseny i realització de proves

---

1. Introducció
2. Les proves en el cicle de vida d'un projecte
3. Procediments, tipus i casos de proves
4. Eines per a la realització de proves

---

1. Introducció

Les proves en el cicle de vida d'un projecte són essencials per diverses raons que contribueixen a l'èxit global del projecte. Ací tens les raons principals per les quals les proves són importants:

1. Assegurament de la Qualitat: Les proves ajuden a garantir que els productes lliurats compleixin amb els estàndards de qualitat establerts. Això inclou assegurar-se que els productes són fiables, funcionen correctament i compleixen amb els requisits especificats.

2. Identificació de Problemes i Errors: Les proves permeten identificar problemes, errors i defectes en les fases inicials del projecte. Això facilita la seua correcció abans que esdevinguen en problemes majors que poden afectar el cost i el temps del projecte.

3. Compliment de Requisits: Les proves asseguren que el projecte compleix amb els requisits dels clients i les parts interessades. Això és fonamental per a la satisfacció dels clients i per garantir que el producte final compleix amb les seues necessitats i expectatives.

4. Reducció de Riscos: Realitzar proves permet identificar i mitigar riscos abans que afecten el projecte. Això inclou riscos tècnics, de negoci i operatius. Les proves de riscos són crucials per minimitzar l'impacte de possibles problemes en el projecte.

5. Avaluació del Rendiment: Les proves de rendiment ajuden a assegurar que el sistema o producte funcione de manera eficient en diferents condicions. Això és important per garantir que el producte puga gestionar la càrrega esperada i proporcionar una bona experiència d'usuari.

6. Verificació i Validació: Les proves proporcionen una manera de verificar que el producte es desenvolupa correctament i validar que el producte final compleix amb els seus objectius i requisits.

7. Compliment Regulatori: En alguns sectors, les proves són necessàries per complir amb les normatives i regulacions legals. Això és especialment important en indústries com la sanitat, l'aeronàutica, i les finances, on la seguretat i la precisió són crítiques.

8. Millora Continua: Les proves permeten identificar àrees de millora en els processos de desenvolupament i implementació. Les lliçons apreses a partir de les proves es poden utilitzar per millorar futurs projectes, augmentant així l'eficiència i l'eficàcia de l'organització.

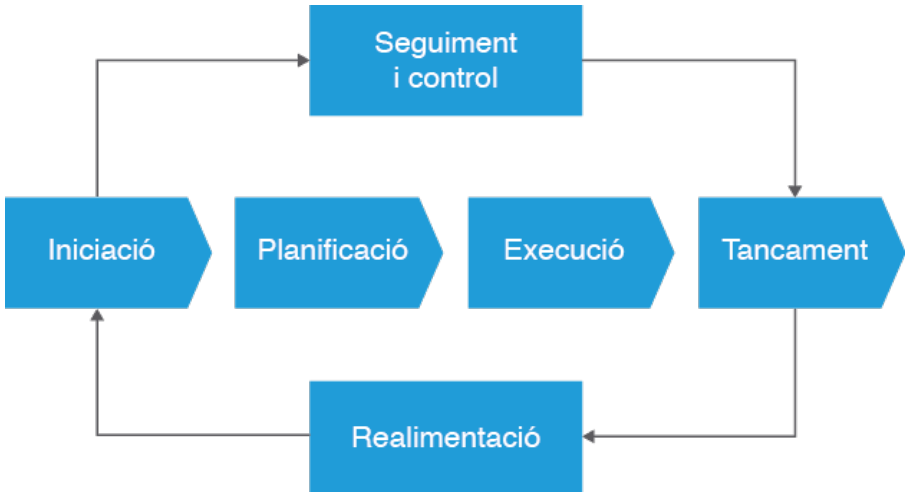
9. Confiança dels Parts Interessades: Realitzar proves rigoroses demostra als clients, usuaris i altres parts interessades que el projecte està sent gestionat de manera professional i que els lliurables seran de qualitat. Això ajuda a construir confiança i a assegurar relacions positives a llarg termini.

10. Reducció de Costos i Temps: Detectar i corregir errors en fases primerenques del projecte és molt més econòmic que fer-ho en fases posteriors o després de la implementació. Això ajuda a mantenir el projecte dins del pressupost i del calendari previst.

En resum, les proves són crucials per assegurar l'èxit d'un projecte, garantint que es compleixen els requisits de qualitat, funcionalitat i rendiment, i que els riscos es gestionen adequadament. Això contribueix a la satisfacció dels clients i a la consecució dels objectius del projecte.

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

El cicle de vida d'un projecte típicament inclou diverses fases: inici, planificació, execució, seguiment i control, i tancament. Dins de cadascuna d'aquestes fases, es poden realitzar diferents tipus de proves per assegurar que el projecte compleix amb els objectius establerts i que els lliurables són de qualitat.



Ací tens una descripció de les proves que es poden realitzar en cada fase del cicle de vida d'un projecte:

1. Inici: En aquesta fase es defineix l'abast del projecte i es realitza un estudi de viabilitat.

Prova de viabilitat: S'avalua si el projecte és factible des d'un punt de vista tècnic, econòmic i operatiu.

Prova d'abast: Es verifica que l'abast del projecte està clarament definit i alineat amb els objectius de l'organització.

2. Planificació: Durant la planificació es desenvolupa un pla detallat per a l'execució i control del projecte.

Prova de planificació: Es revisa el pla de projecte per assegurar que és complet i que cobreix totes les àrees necessàries (temps, cost, qualitat, comunicacions, riscos, etc.).

Prova de riscos: S'analitzen i es proven les estratègies de gestió de riscos per assegurar que són adequades.

3. Execució: Aquesta fase implica la realització de les tasques planificades per produir els lliurables del projecte.

Prova d'integració: Es verifica que els components del sistema treballen correctament junts.

Prova de funcionament: Es realitzen proves per assegurar que els lliurables funcionen segons els requisits establerts.

Prova d'acceptació per part de l'usuari (UAT): Els usuaris finals verifiquen que el producte compleix amb les seves necessitats i requisits.

4. Seguiment i control: Aquesta fase es realitza simultàniament amb l'execució i implica monitoritzar el progrés del projecte.

Prova de rendiment: S'avalua si el projecte està complint amb els objectius de rendiment establerts.

Prova de qualitat: Es verifica que els lliurables compleixen amb els estàndards de qualitat.

Prova de control de canvis: Es revisen i proven els processos de gestió de canvis per assegurar que s'implementen adequadament.

5. Tancament: En aquesta fase es finalitza el projecte i es lliuren els productes finals als clients.

Prova de lliurament: Es comprova que tots els lliurables han estat completats i lliurats segons els requisits.

Prova de post-implementació: Es realitzen proves després del desplegament per assegurar que el producte continua funcionant correctament en l'entorn de producció.

Avaluació de projecte: Es revisen tots els aspectes del projecte per identificar encerts i millores per a futurs projectes.

Aquestes proves ajuden a assegurar que el projecte es desenvolupe de manera ordenada, que es compleixen els objectius, i que els lliurables siguen de qualitat i complisquen les necessitats dels usuaris finals.

3. Procediments, tipus i casos de proves

En l'àmbit de l'enginyeria del software, les proves són essencials per garantir la qualitat i la funcionalitat dels sistemes desenvolupats. Ara detallem els procediments, tipus i casos de proves que es duen a terme en aquest camp:

3.1 Procediments de Proves

Planificació de les proves:

Definició dels objectius de les proves: Què es vol aconseguir amb les proves.

Selecció de tècniques i eines: Quines eines i tècniques s'utilitzaran per realitzar les proves.

Elaboració del pla de proves: Document que detalla com es realitzaran les proves, els recursos necessaris, el calendari, etc.

Disseny de les proves:

Creació de casos de prova: Definició dels escenaris específics que es provaran.

Preparació dels entorns de prova: Configuració dels sistemes necessaris per realitzar les proves.

Definició dels criteris d'acceptació: Estàndards que determinaran si una prova és exitosa o no.

Execució de les proves:

Execució de casos de prova: Realització de les proves planificades segons els casos de prova.

Recol·lecció de dades: Recollida dels resultats i observacions de cada prova.

Avaluació dels resultats:

Anàlisi dels resultats: Avaluació dels resultats per determinar si el software compleix els requisits.

Informe de proves: Documentació dels resultats de les proves, errors trobats, i recomanacions.

Manteniment de les proves:

Actualització de casos de prova: Adaptació dels casos de prova a mesura que el software evoluciona.

Reexecució de proves: Realització de proves regressives per assegurar que els canvis no introdueixin nous errors.

3.2 Tipus de Prova

Prova unitària (Unit Testing): Prova de les unitats més xicotetes del codi (funcions, mètodes) de manera independent.

Normalment es realitza per desenvolupadors utilitzant frameworks de proves unitàries (JUnit, NUnit, etc.).

Prova d'integració (Integration Testing): Prova de la interacció entre diverses unitats o components.

Assegura que els components funcionen correctament quan es combinen.

Prova funcional (Functional Testing): Verificació que el software compleixi els requisits funcionals especificats.

Inclou proves de caixa negra on es prova la funcionalitat sense considerar el codi intern.

Prova de sistema (System Testing): Prova del sistema complet integrat per assegurar-se que compleix les especificacions.

Inclou proves de funcions, rendiment, seguretat, i compatibilitat.

Prova de regressió (Regression Testing): Realització de proves per assegurar que els canvis o actualitzacions no han introduït nous errors.

Es repeteixen proves anteriors per garantir l'estabilitat del sistema.

Prova d'acceptació (Acceptance Testing): Realitzada per l'usuari final o el client per verificar si el software compleix els requisits acordats.

Inclou proves d'acceptació de l'usuari (UAT).

Prova de càrrega (Load Testing): Avaluació del rendiment del sistema sota càrregues específiques.

Determina la capacitat del sistema per manejar un nombre determinat d'usuaris o transaccions.

Prova de seguretat (Security Testing): Identificació de vulnerabilitats i comprovació de les mesures de seguretat del sistema.

Inclou proves de penetració i anàlisi de riscos.

3.3 Casos de Proves

1. Desenvolupament de noves funcionalitats:

Proves unitàries per verificar els mòduls nous.

Proves d'integració per assegurar que els nous mòduls s'integren bé amb els existents.

Proves funcionals per verificar la nova funcionalitat.

2. Correcció d'errors:

Proves unitàries i de regressió per assegurar que els errors corregits no re-apareguin.

Proves d'integració i sistema per verificar que la correcció no ha afectat altres parts del sistema.

3. Actualitzacions de software:

Proves de regressió per assegurar que les actualitzacions no introdueixin nous errors.

Proves de sistema per verificar la funcionalitat i rendiment després de l'actualització.

4. Canvis en els requisits:

Proves funcionals per verificar que els canvis en els requisits s'han implementat correctament.

Proves de sistema per assegurar que els canvis no afectin altres funcionalitats.

5. Llançament de producte:

Proves d'acceptació per garantir que el producte compleix els requisits dels clients.

Proves de càrrega i seguretat per assegurar que el producte pot manejar l'ús en entorns reals.

Aquest enfocament detallat permet assegurar que el software desenvolupat és fiable, segur i compleix les expectatives dels usuaris.

4 Eines per a la realització de proves

A continuació, es presenten diverses eines que s'utilitzen en diferents tipus de proves en enginyeria del software:

Proves Unitàries

1. **JUnit**: Una de les eines més populars per a proves unitàries en Java.
2. **NUnit**: Similar a JUnit, però per a aplicacions .NET.
3. **pytest**: Una eina flexible i potent per a proves unitàries en Python.

Proves d'Integració

1. **TestNG**: Eina de proves avançada per a proves d'integració en Java.
2. **JUnit + Arquillian**: Combinació utilitzada per a proves d'integració i proves d'aplicacions empresarials en Java EE.
3. **Postman**: Utilitzat per a proves d'API RESTful.

Proves de Sistema

1. **Selenium**: Eina per a proves automàtiques de navegadors web.
2. **Cucumber**: Eina de proves basada en comportament (BDD) que pot ser utilitzada per proves de sistema.
3. **Robot Framework**: Eina genèrica d'automatització de proves que suporta diverses biblioteques per a proves de sistema.

Proves de Rendiment

1. **Apache JMeter**: Utilitzat per a proves de rendiment, especialment en aplicacions web.
2. **Gatling**: Eina de codi obert per a proves de rendiment, amb un enfocament en l'automatització i l'escalabilitat.
3. **Locust**: Eina de proves de càrrega fàcilment escalable i distribuïble, escrita en Python.

Proves de Seguretat

1. **OWASP ZAP (Zed Attack Proxy)**: Eina popular per a proves de seguretat en aplicacions web.
2. **Burp Suite**: Eina professional per a proves de seguretat en aplicacions web.
3. **SonarQube**: Eina per a anàlisi contínua de la qualitat del codi i la seguretat.

Proves de Regresió

1. **Selenium**: També utilitzat per a proves de regresió automatitzades.
2. **TestComplete**: Eina comercial per a proves de regressió automatitzades.
3. **Ranorex**: Una altra eina comercial per a proves de regressió i automatització de proves.

Proves d'Acceptació

1. **FitNesse**: Eina de proves d'acceptació col·laborativa.
2. **Cucumber**: A més de les proves de sistema, també és molt utilitzat per a proves d'acceptació gràcies a la seva sintaxi Gherkin.

Eines de Gestió de Proves

1. **TestRail**: Eina de gestió de proves que permet planificar, seguir i gestionar proves.

2. **Zephyr**: Integrada amb Jira, aquesta eina facilita la gestió de proves en equips Agile.
3. **qTest**: Plataforma de gestió de proves que s'integra amb diverses eines de desenvolupament i automatització de proves.

Aquestes eines ajuden a assegurar que el programari siga robust, segur i funcione correctament en totes les situacions previstes. L'elecció de les eines adequades depèn dels requisits específics del projecte i del tipus de proves necessàries.

5. Mètriques de qualitat de programari

Les mètriques de qualitat del programari són essencials per avaluar i assegurar la qualitat dels sistemes de programari. Aquestes mètriques ajuden els equips de desenvolupament i els gestors de projectes a identificar problemes, millorar processos i garantir que el producte final compleixi amb els requisits establerts. A continuació, es presenten algunes de les mètriques de qualitat més importants:

Mètriques de Codi Font

1. **Complexitat Ciclomàtica**: Mesura la complexitat del codi comptant el nombre de camins independents a través del codi. Un valor alt pot indicar que el codi és difícil de mantenir i de provar.
2. **Línies de Codi (LOC)**: Comptatge de les línies de codi. Tot i que no és una mètrica de qualitat directa, pot donar una idea de la mida del projecte.
3. **Densitat de Defectes**: Nombre de defectes per línia de codi. Ajuda a identificar les àrees del codi que són més propenses a errors.

Mètriques de Fiabilitat

1. **Temps Mig Entre Fallades (MTBF)**: Mesura el temps mig entre fallades del sistema. Un valor alt indica una alta fiabilitat.
2. **Temps Mig per a la Reparació (MTTR)**: Mesura el temps mig necessari per reparar un sistema després d'una fallada. Un valor baix indica una alta mantenibilitat.
3. **Taxa de Fallades**: Nombre de fallades per unitat de temps. Ajuda a avaluar la fiabilitat del sistema en funcionament.

Mètriques de Rendiment

1. **Temps de Resposta**: Temps que triga el sistema a respondre a una sol·licitud. És crucial per avaluar el rendiment, especialment en aplicacions en temps real.
2. **Rendiment (Throughput)**: Nombre de transaccions o operacions processades per unitat de temps. Ajuda a determinar la capacitat del sistema.
3. **Ús de Recursos**: Mesura de l'ús de recursos com CPU, memòria i I/O. Un ús eficient dels recursos és indicatiu d'un bon rendiment.

Mètriques de Mantenibilitat

1. **Comprensibilitat**: Mesura de com de fàcil és entendre el codi. Factors com comentaris, noms de variables significatius i estructuració clara contribueixen a aquesta mètrica.

2. **Modularitat**: Mesura de com de ben organitzat està el codi en mòduls independents. Una alta modularitat facilita les modificacions i les proves.
3. **Coupling i Cohesió**: Mesura de la interdependència entre mòduls (coupling) i la relació interna entre els elements d'un mateix mòdul (cohesió). Un baix coupling i una alta cohesió són desitjables per a una millor mantenibilitat.

Mètriques de Seguretat

1. **Nombre de Vulnerabilitats Detectades**: Nombre de vulnerabilitats de seguretat trobades en el codi. Ajuda a avaluar la robustesa del sistema contra atacs.
2. **Temps de Resolució de Vulnerabilitats**: Temps necessari per corregir les vulnerabilitats després de ser detectades. Un temps baix indica una bona capacitat de resposta.
3. **Densitat de Parches de Seguretat**: Nombre de parches de seguretat aplicats per línia de codi o per mòdul. Un valor alt pot indicar un codi menys segur inicialment.

Mètriques de Satisfacció del Client

1. **Nombre de Reclamacions del Client**: Mesura el nombre de reclamacions o queixes dels usuaris finals. Una baixa quantitat és indicativa d'una alta satisfacció.
2. **Temps de Resposta a Incidències**: Mesura el temps que es tarda a respondre i resoldre les incidències reportades pels usuaris. Un temps de resposta ràpid millora la satisfacció del client.
3. **Nivell de Satisfacció del Client**: Mesurat mitjançant enquestes o feedback directe dels usuaris. Proporciona una visió general de com els usuaris perceben la qualitat del programari.

Mètriques de Proves

1. **Cobertura de Codi**: Percentatge de codi que ha estat executat durant les proves. Una alta cobertura indica que moltes parts del codi han estat provades.
2. **Tasa de Defectes Detectats Durant les Proves**: Nombre de defectes trobats durant les proves en comparació amb el nombre total de proves realitzades. Ajuda a avaluar l'efectivitat de les proves.
3. **Nombre de Casos de Prova Executats**: Mesura del nombre de casos de prova que s'han executat en una sessió de proves. Indica l'abast de les proves realitzades.

Aquestes mètriques són fonamentals per assegurar la qualitat del programari i per prendre decisions informades durant el cicle de vida del desenvolupament del programari.

Per avaluar la qualitat del programari, hi ha diversos programes i eines disponibles que ajuden a mesurar diferents mètriques de qualitat. Aquests programes proporcionen anàlisis automàtiques del codi, informes de qualitat, i suggeriments per a la millora.

A continuació, es detallen alguns dels principals programes de mètrica de qualitat de programari:

1. SonarQube

- **Descripció**: Una plataforma d'anàlisi de codi obert que ajuda a mesurar i analitzar la qualitat del codi.
- **Funcionalitats**: Analitza la seguretat, la mantenibilitat, la fiabilitat, la cobertura de codi i les duplicacions de codi.
- **Integració**: Compatible amb diversos llenguatges de programació i s'integra amb eines de CI/CD com Jenkins, GitLab i altres.

2. Coverity

- **Descripció**: Una eina de verificació estàtica de codi que ajuda a detectar errors i vulnerabilitats en el codi font.
- **Funcionalitats**: Identifica defectes relacionats amb la seguretat, el rendiment, la fiabilitat i la mantenibilitat.
- **Integració**: S'integra amb entorns de desenvolupament integrats (IDEs) i eines de CI/CD.

3. CodeClimate

- **Descripció**: Una plataforma basada en el núvol que proporciona anàlisi de qualitat de codi i informes de mantenibilitat.
- **Funcionalitats**: Ofereix mètriques sobre mantenibilitat, seguretat, duplicació de codi i cobertura de proves.
- **Integració**: S'integra amb GitHub, GitLab, Bitbucket i altres eines de gestió de codi font.

4. Kiuwan

- **Descripció**: Una plataforma d'anàlisi de codi i seguretat que ofereix una visió completa de la qualitat i la seguretat del codi.
- **Funcionalitats**: Proporciona mètriques sobre seguretat, qualitat del codi, mantenibilitat, eficiència i compatibilitat.
- **Integració**: S'integra amb entorns de desenvolupament, repositoris de codi i eines de CI/CD.

5. CAST Software

- **Descripció**: Una plataforma d'anàlisi de codi que ofereix una avaluació detallada de la qualitat i la complexitat del codi.
- **Funcionalitats**: Mesura la robustesa, la seguretat, l'eficiència, la mantenibilitat i la transferibilitat.
- **Integració**: Compatible amb diversos llenguatges de programació i s'integra amb eines de gestió de projectes i CI/CD.

6. Pylint

- **Descripció**: Una eina d'anàlisi de codi estàtic per a Python que ajuda a millorar la qualitat del codi.
- **Funcionalitats**: Detecta errors de programació, codis potencialment perillosos, normes de codificació, i proporciona suggeriments per a la millora del codi.

- **Integració**: S'integra fàcilment amb editors de text i entorns de desenvolupament com VSCode, PyCharm, i altres.

7. ESLint

- **Descripció**: Una eina d'anàlisi de codi estàtic per a JavaScript que ajuda a trobar i corregir problemes en el codi.
- **Funcionalitats**: Ofereix regles configurables per a la qualitat del codi, la seguretat i les normes de codificació.
- **Integració**: S'integra amb diversos editors de codi i entorns de desenvolupament.

8. Checkstyle

- **Descripció**: Una eina de verificació de codi per a Java que ajuda a fer complir les normes de codificació.
- **Funcionalitats**: Proporciona informes sobre l'estil del codi, la complexitat ciclomatica, i les possibles millores.
- **Integració**: S'integra amb IDEs com Eclipse i IntelliJ IDEA, i amb sistemes de compilació com Maven i Gradle.

9. PMD

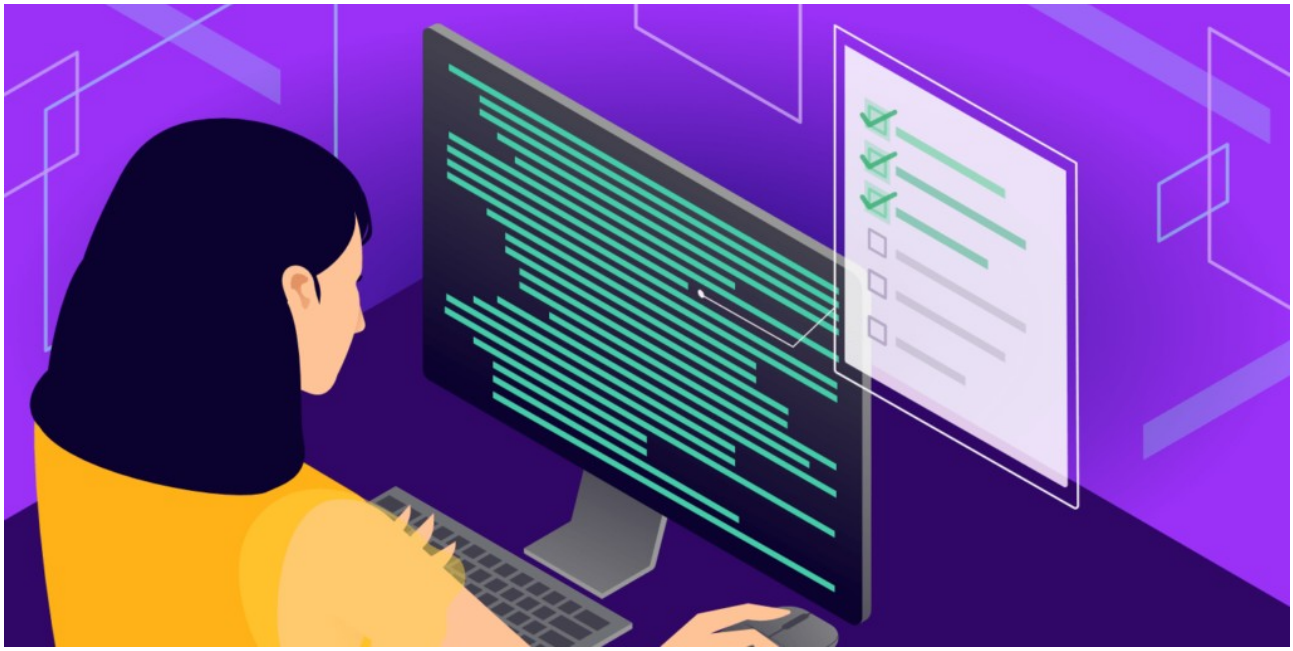
- **Descripció**: Una eina d'anàlisi de codi estàtic per a Java i altres llenguatges, que detecta codi defectuós o potencialment perillós.
- **Funcionalitats**: Identifica errors comuns de programació, codi redundant i altres problemes de mantenibilitat.
- **Integració**: S'integra amb IDEs i eines de construcció com Eclipse, IntelliJ IDEA, Maven i Gradle.

10. FindBugs/SpotBugs

- **Descripció**: Una eina d'anàlisi de codi estàtic per a Java que detecta possibles errors.
- **Funcionalitats**: Analitza el bytecode Java per trobar defectes que poden causar errors en temps d'execució.
- **Integració**: S'integra amb IDEs com Eclipse i NetBeans, i amb sistemes de construcció com Maven i Gradle.

Aquestes eines proporcionen una visió integral de la qualitat del codi, ajudant els equips de desenvolupament a mantenir i millorar contínuament la qualitat del programari.

Tema 4: Optimització i documentació



1. Introducció
2. Reflacció
3. Eines d'ajuda a la refacció
4. Control de versions
5. Eines populars de control de versions
6. Bones pràctiques per al control de versions
7. Utilització de Git
8. Utilització de Github

1. Introducció

L'optimització de codi és un aspecte crucial per millorar el rendiment, la mantenibilitat i l'eficiència de les aplicacions. A continuació, es detallen diverses tècniques, pràctiques i eines específiques per a l'optimització de codi:

Optimització d'Algorismes i Estructures de Dades

Selecció d'algorismes: Utilitzar algorismes més eficients per a tasques específiques. Per exemple, utilitzar HashMap per a cerques ràpides en lloc de ArrayList.

Estructures de dades adequades: Utilitzar estructures de dades com HashMap, TreeSet, o ConcurrentHashMap segons el context d'ús.

Eliminació de Codi Innecessari

Eliminació de codi mort: Utilitzar eines com IntelliJ IDEA o Eclipse per identificar i eliminar codi que mai s'executa.

Simplificació de codi: Reduir la complexitat del codi mitjançant la simplificació de condicions i expressions. Per exemple, en comptes de:

```
if (flag == true) {
      // codi
}
```

Utilitzar:

```
if (flag) {
      // codi
}
```

Reducció de Repeticions de Codi

Refactorització: Utilitzar les capacitats de refactorització d'IDEs com IntelliJ IDEA per reduir la duplicació de codi.

Funcions i mètodes reutilitzables: Crear mètodes generals que es poden reutilitzar en lloc de duplicar el mateix codi en diferents llocs.

Optimització de Llaços

Reducció de la complexitat del bucle: Minimitzar el nombre d'iteracions i moure el codi invariant fora del bucle. Per exemple:

```
for (int i = 0; i < array.length; i++) {
    for (int j = 0; j < array[i].length; j++) {
        // codi
    }
}
```

En lloc de:

```
int length = array.length;
for (int i = 0; i < length; i++) {
```

```
        int innerLength = array[i].length;
        for (int j = 0; j < innerLength; j++) {
            // codi
        }
    }
```

Gestió Eficient de la Memòria

Evitar fuites de memòria: Assegurar-se que tots els recursos al·locats s'alliberin correctament. Utilitzar try-with-resources per gestionar recursos.

Minimització de l'ús de memòria: Utilitzar tipus de dades adequats i evitar al·locacions de memòria innecessàries. Per exemple, utilitzar StringBuilder en lloc de concatenacions de cadenes en bucles.

Paral·lelització i Concurrència

Multithreading: Utilitzar la llibreria java.util.concurrent per gestionar tasques concurrents de manera eficient.

Asynchronous programming: Utilitzar CompletableFuture per millorar la resposta de l'aplicació.

Profiling i Mesura

Profiling: Utilitzar eines de profiling com VisualVM per identificar colls de botella i àrees de millora en el codi.

Benchmarking: Utilitzar biblioteques com JMH (Java Microbenchmark Harness) per mesurar el rendiment del codi abans i després de l'optimització.

Revisió de Codi

Revisions de codi: Fer que el codi siga revisat per altres membres de l'equip per identificar possibles millores i errors.

Pair programming: Col·laborar amb un altre programador per escriure codi de major qualitat i detectar problemes més ràpidament.

Documentació i Comentaris

Comentaris clars: Escriure comentaris que expliquin el propòsit i el funcionament del codi, especialment per a les parts complexes.

Documentació de l'optimització: Documentar les raons i els resultats de les optimitzacions per facilitar la comprensió i el manteniment futur.

2. Reflacció

La refactorització de codi és una pràctica essencial en enginyeria del programari que implica modificar i millorar l'estructura interna del codi sense canviar-ne el comportament extern. Aquesta pràctica millora la llegibilitat, la mantenibilitat, la qualitat del codi i facilita futures extensions o modificacions. A continuació, es descriuen els objectius, les tècniques, les millors pràctiques i les eines per a la refactorització de codi.

Objectius de la Refactorització de Codi

Millorar la llegibilitat: Fer que el codi sigui més fàcil de llegir i comprendre per altres desenvolupadors.

Augmentar la mantenibilitat: Facilitar la detecció i correcció d'errors, així com la introducció de noves funcionalitats.

Reduir la complexitat: Simplificar estructures de codi complicades o innecessàriament complexes.

Eliminar duplicació de codi: Reutilitzar codi existent en lloc de duplicar-lo en diverses parts del programa.

Millorar el rendiment: En alguns casos, la refactorització pot millorar l'eficiència del codi.

Tècniques de Refactorització de Codi

Renombrar Variables, Mètodes i Classes: Canviar noms per fer-los més descriptius i coherents.

```
 // Abans
int d;
d = 100 - y;

// Després
int remainingDays;
remainingDays = 100 - elapsedDays;
```

Extracció de Mètodes: Crear mètodes nous a partir de fragments de codi repetitius o complexos.

```
// Abans
public void printOwing() {
    printBanner();
    // Imprimeix detalls
    System.out.println("name: " + name);
    System.out.println("amount: " + getOutstanding());
}

// Després
public void printOwing() {
    printBanner();
    printDetails();
}
```

```
    private void printDetails() {
        System.out.println("name: " + name);
        System.out.println("amount: " + getOutstanding());
    }
```

Encapsulació de Camps: Fer els camps privats i proporcionar mètodes d'accés (getters i setters).

```
     // Abans
    public String name;

    // Després
    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
```

Substitució de Fragments de Codi amb Mètodes: Substituir codi duplicat amb una crida a un mètode comú.

```
    // Abans
    if (isPrime(num)) {
        System.out.println(num + " is prime");
    }
    if (isPrime(otherNum)) {
        System.out.println(otherNum + " is prime");
    }

    // Després
    checkAndPrintIfPrime(num);
    checkAndPrintIfPrime(otherNum);

    private void checkAndPrintIfPrime(int number) {
        if (isPrime(number)) {
            System.out.println(number + " is prime");
        }
    }
```

Substitució de Codis Màgics amb Constants: Substituir valors literals amb constants descriptives.

```
        // Abans
        if (responseCode == 200) {
            // Codi de resposta OK
        }

        // Després
        private static final int HTTP_OK = 200;

        if (responseCode == HTTP_OK) {
            // Codi de resposta OK
        }
```

Bones pràctiques per a la Refactorització de Codi

Refactoritzar Regularment: Fer de la refactorització una pràctica habitual per mantenir el codi net i ben estructurat.

Proves Automatitzades: Assegurar-se que hi ha proves suficients abans de refactoritzar i executar-les després per verificar que el comportament del codi no ha canviat.

Xicotets Canvis Incrementals: Fer canvis xicotets i freqüents en lloc de grans modificacions de cop. Això facilita la detecció de problemes i fa que el procés siga menys arriscat.

Utilitzar eines de suport: Utilitzar eines i funcions de refactorització integrades en els entorns de desenvolupament (IDEs) per fer canvis de manera segura i eficient.

Comentaris i Documentació: Afegir comentaris i documentació per explicar les raons darrere de les refactoritzacions i els canvis importants en el codi.

Mitjançant l'ús d'aquestes tècniques, pràctiques, els desenvolupadors poden mantenir el codi net, comprensible i fàcilment mantenible, la qual cosa resulta en una major qualitat i eficàcia del programari.

3. Eines d'ajuda a la refacció

IntelliJ IDEA: Potent entorn de desenvolupament integrat (IDE) amb suport extensiu per a la refactorització de codi. Funcions com "Rename", "Extract Method", "Inline Variable" i altres.

Eclipse: Un altre IDE popular per a Java amb diverses eines de refactorització. Suport per a "Rename", "Extract Method", "Change Method Signature", etc.

NetBeans: IDE amb eines de refactorització similars a IntelliJ IDEA i Eclipse. Suport per a "Encapsulate Fields", "Move Class", "Extract Superclass", etc.

ReSharper: Plugin per a Visual Studio que proporciona funcionalitats avançades de refactorització per a diversos llenguatges, incloent Java (a través d'IntelliJ IDEA).

SonarQube: Eina d'anàlisi estàtica de codi que ajuda a identificar àrees que necessiten refactorització i proporciona suggeriments específics.

PMD: Eina d'anàlisi de codi estàtic que pot identificar problemes comuns que es poden solucionar amb refactorització.

4. Control de versions

El control de versions és una pràctica essencial en el desenvolupament de programari que permet gestionar els canvis en el codi font i altres fitxers relacionats al llarg del temps. Aquesta pràctica facilita el seguiment de la història de modificacions, la col·laboració entre desenvolupadors, i la gestió de diferents versions del projecte. A continuació es detallen

els conceptes bàsics, els beneficis, les eines i les millors pràctiques associades al control de versions.

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.

7. 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.

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

Tot i que Git és un sistema distribuït, pot utilitzar-se una còpia del repositori en un servidor de manera que tots els usuaris pugen i baixen els canvis d'aquest repositori per facilitar la sincronització.

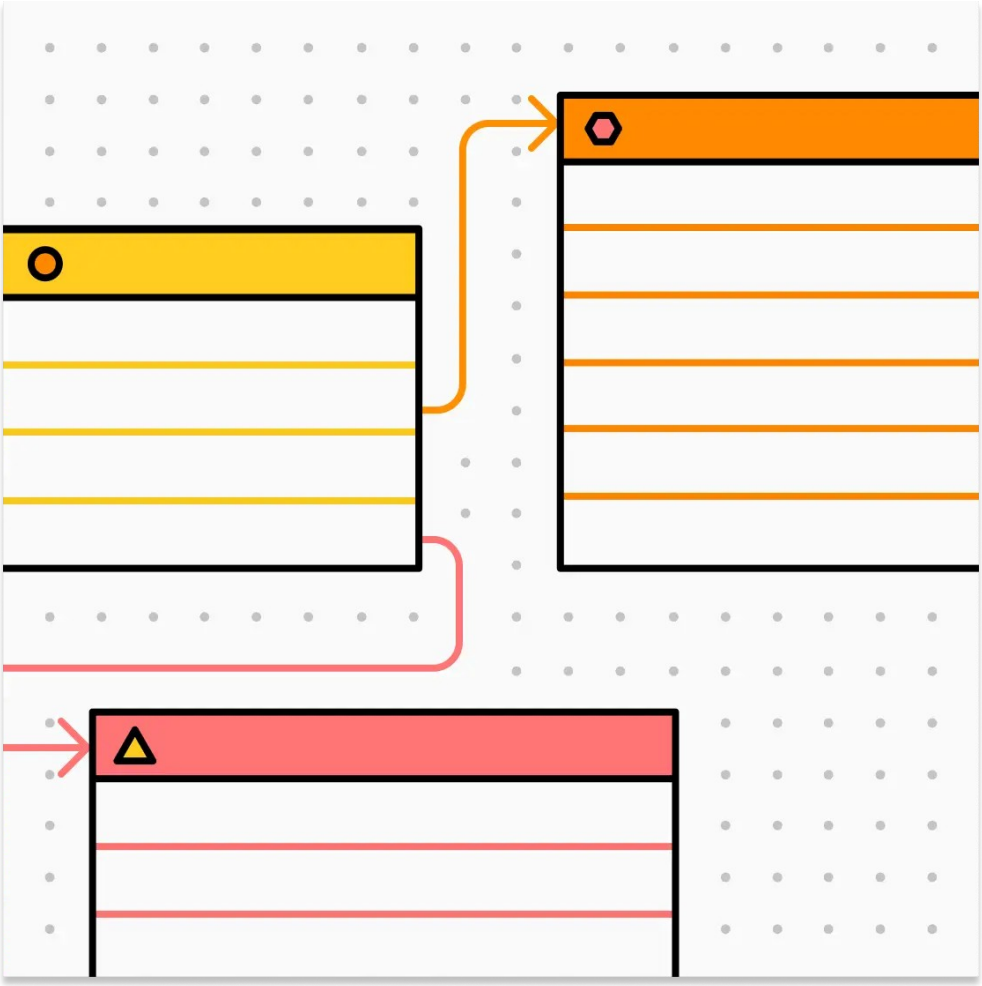
Mirar un tutorial

8. Utilització de Github

GitHub (github.com) és un servei d'allotjament de repositoris Git que compta amb més de 30 milions d'usuaris. Ofereix tota la funcionalitat de Git, a més d'oferir serveis propis com són l'edició de fitxer en línia, la gestió d'errors, possibilitat de documentar els projectes mitjançant una wiki inclosa al repositori o la gestió d'usuaris.

Mirar un tutorial

Tema 5: Elaboració de diagrames de classes



1. UML
2. Diagrama de classes. Elements
3. Com dibuixar un diagrama de classes
4. Exemples de diagrama de classes
5. Diagrama d'objectes
6. Diagrama d'objectes vs Diagrama de classes
7. Eines de disseny de diagrames.
8. Generació de codi a partir de diagrames de classes i a l'inrevés

1. UML

UML (Unified Modeling Language) és un llenguatge de modelatge estàndard en l'enginyeria de programari que s'utilitza per a especificar, visualitzar, construir i documentar els artefactes d'un sistema de programari. Hi ha diversos tipus de diagrames UML que es classifiquen en dues categories principals: diagrames estructurals i diagrames de comportament. A continuació es presenten els tipus principals dins de cadascuna d'aquestes categories:

Diagrames Estructurals

1. **Diagrama de Classes**: Mostra les classes del sistema, els seus atributs, mètodes i les relacions entre elles.
2. **Diagrama d'Objectes**: Representa instàncies específiques de les classes i les relacions entre elles en un moment concret.
3. **Diagrama de Components**: Representa els components físics del sistema i les seues interaccions.
4. **Diagrama de Distribució (Deployment)**: Mostra la configuració del maquinari del sistema i com es despleguen els components de programari en aquest maquinari.
5. **Diagrama de Paquets**: Organitza els elements del model en paquets per mostrar-ne les dependències.
6. **Diagrama de Perfils**: Permet crear extensions personalitzades de UML per adaptar-se a necessitats específiques.

Diagrames de Comportament

1. **Diagrama de Casos d'Ús (Use Case)**: Mostra els casos d'ús del sistema i les interaccions entre els actors i aquests casos d'ús.
2. **Diagrama de Seqüència**: Descriu com els objectes interactuen en una seqüència temporal específica.
3. **Diagrama de Comunicació (o Col·laboració)**: Similar al diagrama de seqüència, però es centra en les relacions entre els objectes.
4. **Diagrama d'Activitats**: Representa el flux de treball o les activitats dins d'un procés o sistema.
5. **Diagrama d'Estats (o Diagrama d'Estat-Transició)**: Mostra els estats pels quals passa un objecte i les transicions entre aquests estats.
6. **Diagrama de Temps**: Representa els canvis d'estat dels objectes en funció del temps.
7. **Diagrama d'Interacció General**: Integra diversos tipus de diagrames d'interacció (seqüència, comunicació, temps) per descriure comportaments complexes.

En resum

Els diagrames UML són eines potents per ajudar a la comprensió i documentació dels sistemes de programari. Cada tipus de diagrama ofereix una perspectiva diferent, ja siga estructural o de comportament, per assegurar que totes les parts interessades tinguen una visió clara del sistema en desenvolupament.

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

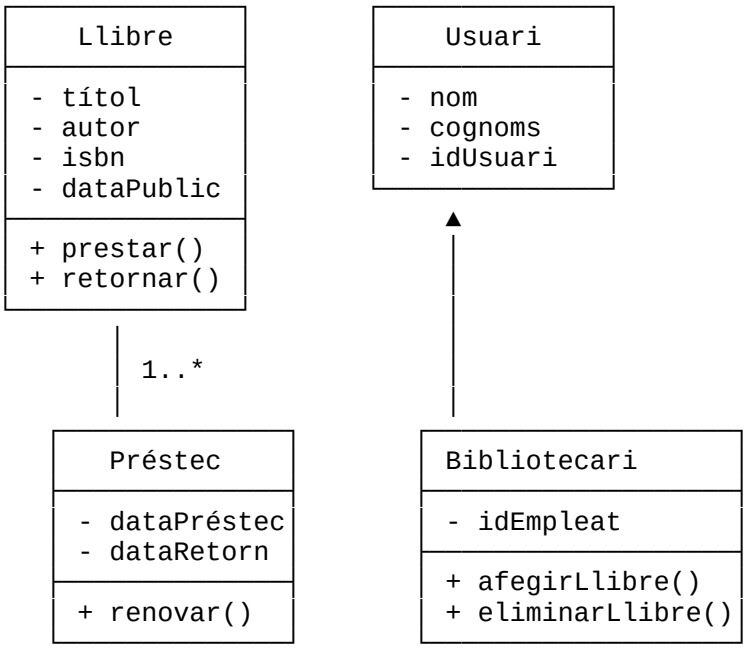
Classes i Relacions

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 és associat a un sol usuari (1..*).
    - **Associació** entre Llibre i Préstec: Un llibre pot ser prestat moltes vegades, però un préstec és per a un sol llibre (1..*).
    - **Herència**: Bibliotecari hereta de 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.

En resum, 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**:
    - **Associació**: Una línia simple que connecta dues classes. Pots afegir una fletxa per indicar la direcció si l'associació és unidireccional.
    - **Agregació**: Una línia amb un rombe buit en un extrem, indicant la classe que conté (el "tot").
    - **Composició**: Una línia amb un rombe ple en un extrem, indicant la classe que conté.
    - **Herència**: Una línia amb un triangle buit apuntant a la superclasse.
    - **Dependència**: Una línia de punts amb una fletxa que indica la direcció de la dependència.

Eines Recomanades

- Lucidchart
- Microsoft Visio
- Draw.io (Gratuïta)
- StarUML
- PlantUML (per a dibuixar diagrames amb codi)

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.

Amb aquests passos, podràs crear un diagrama de classes UML que represente clarament l'estructura del teu sistema.

4. Exemples de diagrama de classes

5. Diagrama d'objectes

El diagrama d'objectes és una representació gràfica d'instàncies de classes en un moment concret del temps. Aquest diagrama és útil per mostrar exemples concrets de com les

classes es relacionen entre si en una situació específica, sovint per ajudar a comprendre millor la dinàmica del sistema.

Components Clau d'un Diagrama d'Objectes

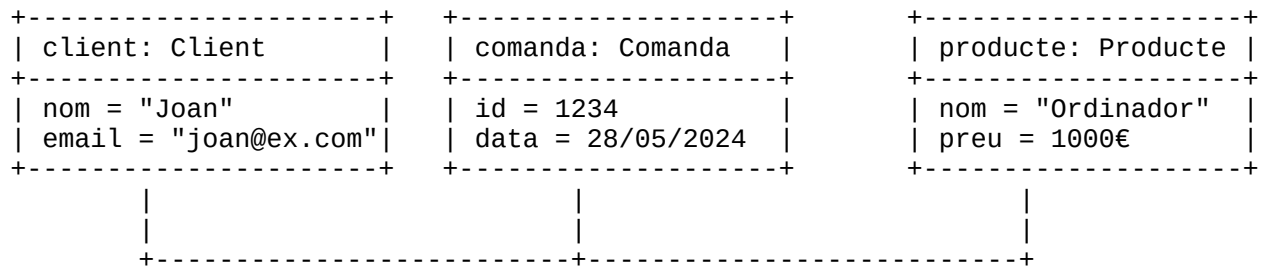
1. **Objectes**
    - **Noms:** Els objectes es representen com rectangles, amb el nom de l'objecte seguit del nom de la classe separats per dos punts (e.g., client: Client). Si l'objecte no té nom, només es mostra el nom de la classe.
    - **Atribucions:** Els atributs de l'objecte es poden mostrar baix del nom, amb els seus valors assignats (ex: nom = "Joan")
2. **Relacions**
    - **Associacions:** Les línies que connecten els objectes representen associacions. Poden incloure rols i multiplicitats, tot i que aquestes es mostren amb menys freqüència en diagrames d'objectes en comparació amb els diagrames de classes.
    - **Agregació/Composició:** Encara que menys comú, també es poden representar per mostrar que un objecte forma part d'un altre.
    - **Enllaços:** A diferència dels diagrames de classes, on es mostren les possibles relacions entre classes, els diagrames d'objectes mostren els enllaços concrets entre instàncies.

Exemples de Diagrama d'Objectes

Per il·lustrar un diagrama d'objectes, considerem un sistema senzill de gestió de comandes:

Classes involucrades: Client, Comanda i Producte

Diagrama d'Objectes:



En aquest exemple:

- client: Client representa una instància de la classe Client amb nom "Joan".
- comanda: Comanda representa una instància de la classe Comanda amb un identificador id de 1234 i una data.
- producte: Producte representa una instància de la classe Producte amb el nom "Ordinador" i un preu de 1000€.
- Les línies que connecten els objectes mostren les relacions concretes entre aquestes instàncies.

Bones Pràctiques per a Diagrames d'Objectes

1. **Claredat i Simlicitat:** Només mostra les instàncies d'objectes i relacions necessàries per a l'exemple concret que estàs tractant d'explicar.
2. **Valors Concreents:** Inclou valors concrets per als atributs dels objectes per fer el diagrama més comprensible.
3. **Context Específic:** Assegura't que el diagrama d'objectes està vinculat a un escenari específic o cas d'ús concret per evitar ambigüitats.
4. **Etiqueta Adequada:** Etiqueta clarament cada objecte amb el seu nom i tipus per evitar confusions.
5. **Us Limitat de Detalls:** No sobrecarreguis el diagrama amb massa detalls. Mantingues-lo el més simple possible per a la situació que vols representar.

Els diagrames d'objectes són especialment útils durant la fase de disseny i anàlisi per validar la dinàmica d'instàncies específiques i ajudar els desenvolupadors i altres parts interessades a entendre millor el comportament del sistema.

6. Diagrama d'objectes vs Diagrama de classes

tenen diferents objectius i utilitzen elements diferents. A continuació, es descriuen les diferències clau entre ambdós tipus de diagrames:

Diagrama de Classes

1. **Objectiu:**
    - Representar l'estructura estàtica del sistema.
    - Mostrar les classes del sistema, els seus atributs, mètodes i les relacions entre elles.
2. **Elements Principals:**
    - **Classes:** Representades com a rectangles dividits en tres parts (nom, atributs i mètodes).
    - **Atributs i Mètodes:** Detallats dins de les classes.
    - **Relacions:** Inclouen associacions, agregacions, composicions i herències.
    - **Visibilitat:** Mostra la visibilitat (pública, privada, protegida) dels atributs i mètodes.
3. **Relacions:**
    - **Associacions:** Línies que connecten classes, poden incloure rols i multiplicitats.
    - **Agregació/Composició:** Representen relacions de part-a-tot (amb diamants buits o plens respectivament).
    - **Herència:** Mostrada amb línies amb un triangle al final, indicant una relació de generalització/especialització.

Diagrama d'Objectes

1. **Objectiu:**
    - Representar l'estat del sistema en un moment específic.
    - Mostrar instàncies concretes de les classes i les relacions entre elles en aquest moment.

2. **Elements Principals:**
    - **Objectes:** Representats com a rectangles amb el nom de l'objecte i la classe (e.g., client1: Client).
    - **Atributs:** Els atributs poden tenir valors concrets.
    - **Relacions:** Mostren enllaços entre objectes específics.
3. **Relacions:**
    - **Enllaços:** Línies que connecten objectes, representant associacions específiques en aquell moment.

Diferències Clau

1. **Temporalitat:**
    - **Diagrama de Classes:** Mostra la definició estàtica i la relació de classes. No canvia amb el temps.
    - **Diagrama d'Objectes:** Mostra l'estat dinàmic en un moment concret del temps. Pot canviar amb diferents escenaris o instàncies.
2. **Nivell d'Abstracció:**
    - **Diagrama de Classes:** Nivell alt d'abstracció, representant conceptes generals.
    - **Diagrama d'Objectes:** Nivell baix d'abstracció, representant casos concrets.
3. **Propòsit:**
    - **Diagrama de Classes:** Dissenyar l'arquitectura del sistema i definir els components principals i les seves relacions.
    - **Diagrama d'Objectes:** Entendre com els objectes interactuen en un escenari específic, sovint utilitzat per proves i validació.
4. **Element Dinàmic vs Estàtic:**
    - **Diagrama de Classes:** Estàtic, no mostra l'estat del sistema en execució.
    - **Diagrama d'Objectes:** Dinàmic, mostra l'estat del sistema en execució.

En resum, els diagrames de classes són útils per planificar i dissenyar l'arquitectura del sistema, mentre que els diagrames d'objectes són útils per comprendre com es comporta el sistema en situacions específiques. Ambdós tipus de diagrames es complementen i s'utilitzen conjuntament en el desenvolupament de programari orientat a objectes.

7. Eines de disseny de diagrames.

1. **Lucidchart**
    - **Descripció:** Una eina basada en el núvol que permet crear diagrames UML de manera col·laborativa.
    - **Funcionalitats:** Diagrames de classes, diagrames d'objectes, col·laboració en temps real, plantilles predefinides.
    - **Plataforma:** Web.
    - **Preu:** Té una versió gratuïta amb funcionalitats limitades i plans de pagament per a funcionalitats avançades.
2. **Microsoft Visio**

- **Descripció:** Una eina de diagrames versàtil de Microsoft, àmpliament utilitzada en entorns corporatius.
- **Funcionalitats:** Suport per a una àmplia varietat de diagrames, inclosos UML, integració amb altres productes de Microsoft.
- **Plataforma:** Windows.
- **Preu:** De pagament, amb opcions de subscripció a Office 365.

3. **StarUML**
    - **Descripció:** Una eina de modelatge UML potent i extensible.
    - **Funcionalitats:** Suport per a múltiples tipus de diagrames UML, extensibilitat amb plugins, exportació a diversos formats.
    - **Plataforma:** Windows, macOS, Linux.
    - **Preu:** De pagament, amb un període de prova gratuït.
4. **Enterprise Architect**
    - **Descripció:** Una eina completa per al modelatge UML i l'enginyeria de sistemes.
    - **Funcionalitats:** Suport per a UML, BPMN, SysML, col·laboració d'equips, generació de codi, enginyeria inversa.
    - **Plataforma:** Windows.
    - **Preu:** De pagament, amb diferents edicions segons les necessitats.
5. **Visual Paradigm**
    - **Descripció:** Una eina de modelatge visual per a UML i altres llenguatges de modelatge.
    - **Funcionalitats:** Suport per a UML, ERD, DFD, col·laboració en equip, generació de codi, integració amb IDEs.
    - **Plataforma:** Windows, macOS, Linux.
    - **Preu:** Té una versió comunitària gratuïta amb funcionalitats limitades i plans de pagament per a funcionalitats avançades.
6. **Draw.io (diagrams.net)**
    - **Descripció:** Una eina gratuïta basada en el núvol per a la creació de diagrames, incloent UML.
    - **Funcionalitats:** Suport per a UML, interfície intuïtiva, integració amb Google Drive i altres plataformes de núvol.
    - **Plataforma:** Web.
    - **Preu:** Gratuïta.
7. **ArgoUML**
    - **Descripció:** Una eina de modelatge UML de codi obert.
    - **Funcionalitats:** Suport per a diagrames UML bàsics, generació de codi.
    - **Plataforma:** Java (multiplataforma).
    - **Preu:** Gratuïta.
8. **PlantUML**
    - **Descripció:** Una eina que permet crear diagrames UML a partir d'un llenguatge de descripció de text.

- **Funcionalitats:** Suport per a diagrames UML diversos, integració amb entorns de desenvolupament com IntelliJ IDEA, Visual Studio Code, exportació a diversos formats.
- **Plataforma:** Multiplataforma (basada en Java).
- **Preu:** Gratuïta.

Consideracions per Escollir una Eina

1. **Funcionalitats Necessàries:** Assegura't que l'eina suporti els tipus de diagrames UML que necessites crear.
2. **Facilitat d'Ús:** Considera la corba d'aprenentatge i la interfície d'usuari.
3. **Col·laboració:** Si treballes en equip, busca eines que permetin col·laboració en temps real.
4. **Integració:** Si necessites integració amb altres eines o IDEs, verifica les opcions disponibles.
5. **Cost:** Tenint en compte el pressupost del teu projecte o organització, escull una eina que s'ajusti a les teves necessitats econòmiques.

Cada eina té les seves pròpies avantatges i limitacions, així que tria la que millor s'adapti a les teves necessitats i al teu flux de treball.

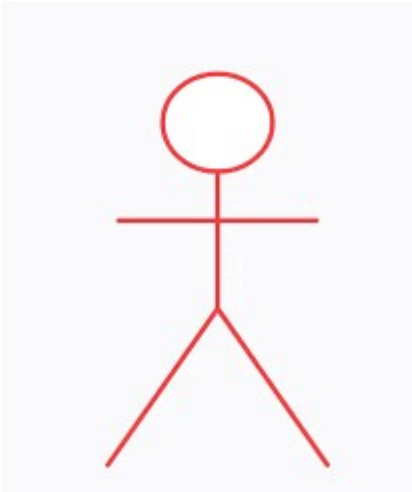
8. Generació de codi a partir de diagrames de classes i a l'inrevés

La Generació Automàtica de Codi consisteix en la creació utilitzant eines CASE de codi font de manera automatitzada. El procés passa per establir una correspondència entre els elements formals dels diagrames i les estructures d'un llenguatge de programació concret. El diagrama de classes és un bon punt de partida perquè permet una traducció força directa de les classes representades gràficament, a classes escrites en un llenguatge de programació específic com Java o C++.

Normalment les eines de generació de diagrames UML inclouen la facilitat de la generació, o actualització automàtica de codi font, a partir dels diagrames creats.

Moltes eines de disseny UML poden generar-los a partir de codi font de C++, Java, Python, IDL, Pascal/Delphi, Ada, o Perl, fent ús d'extensions.

Tema 6: Elaboració de diagrames de comportament



1. Diagrames de comportaments

Els diagrames de comportament en UML (Unified Modeling Language) són utilitzats per descriure el comportament dinàmic del sistema. Aquests diagrames se centren en com els objectes interactuen i canvien al llarg del temps. Els principals diagrames de comportament en UML són:

1. Diagrama de casos d'ús (Use Case Diagram):

Propòsit: Mostra les funcionalitats que el sistema ofereix als usuaris (actors) i les relacions entre els casos d'ús.

Elements clau:

Actors: Representen els usuaris o altres sistemes que interactuen amb el sistema.

Casos d'ús: Representen les funcionalitats o serveis que ofereix el sistema.

Relacions: Inclouen associacions, inclusions (include), extensions (extend) i generalitzacions.

2. Diagrama de seqüència (Sequence Diagram):

Propòsit: Mostra com els objectes interactuen en un flux temporal, destacant l'ordre dels missatges intercanviats.

Elements clau:

Actors i objectes: Els participants de la interacció.

Línies de vida (lifelines): Representen la vida d'un objecte durant la interacció.

Missatges: Representen la comunicació entre els objectes.

Activacions: Períodes durant els quals un objecte està executant una operació.

3. Diagrama de col·laboració o comunicació (Collaboration/Communication Diagram):

Propòsit: Similar al diagrama de seqüència però se centra més en les relacions entre els objectes.

Elements clau:

Actors i objectes: Participants de la interacció.

Línies de comunicació: Connexions entre objectes que indiquen els missatges intercanviats.

Missatges: Numerats per indicar l'ordre d'interacció.

4. Diagrama d'estats (State Machine Diagram):

Propòsit: Mostra els diferents estats d'un objecte i les transicions entre aquests estats.

Elements clau:

Estats: Diferents condicions o situacions de l'objecte.

Transicions: Canvis d'un estat a un altre, generalment causats per esdeveniments.

Esdeveniments: Accions o situacions que provoquen transicions.

Condicions de guarda: Condicions que han de complir-se per activar una transició.

5. Diagrama d'activitats (Activity Diagram):

Propòsit: Mostra el flux de treball o processos, destacant les activitats realitzades i el flux de control entre elles.

Elements clau:

Activitats: Representen operacions o tasques.

Decisions i bifurcacions: Indiquen punts de decisió en el flux.

Nusos inicials i finals: Indiquen el començament i el final del flux de treball.

Flux de control: Fletxes que indiquen l'ordre de les activitats.

6. Diagrama de temps (Timing Diagram):

Propòsit: Representa el comportament dels objectes durant un període de temps específic, similar al diagrama de seqüència però amb un enfocament més gran en el temps.

Elements clau:
Línies de vida: Representen els objectes.
Intervals de temps: Mostren canvis d'estat o valors de propietats durant el temps.
Esdeveniments: Accions que marquen canvis significatius en el temps.

Aquests diagrames proporcionen diferents perspectives del comportament del sistema, permetent una millor comprensió i documentació dels processos i interaccions.

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 ovals.

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.

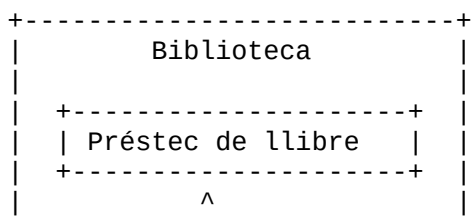
Inclusió (Include): Indica que un cas d'ús inclou la funcionalitat d'un altre cas d'ús.

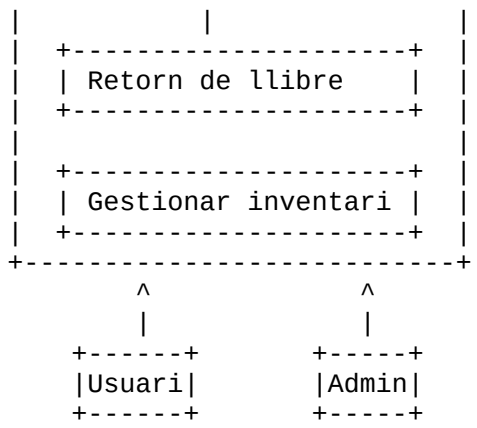
Extensió (Extend): Indica que un cas d'ús pot estendre el comportament d'un altre cas d'ús.

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

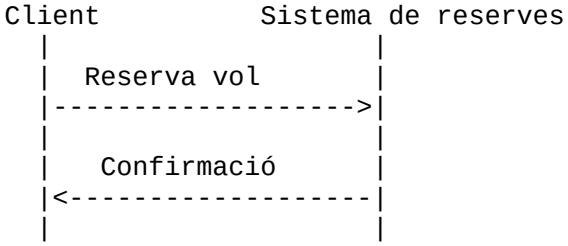
Els diagrames de seqüència mostren la interacció entre els objectes en una seqüència temporal. Aquests diagrames representen els objectes alineats a la part superior i els missatges enviats entre ells al llarg d'una línia de temps vertical.

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:



3.2 Diagrames de comunicació

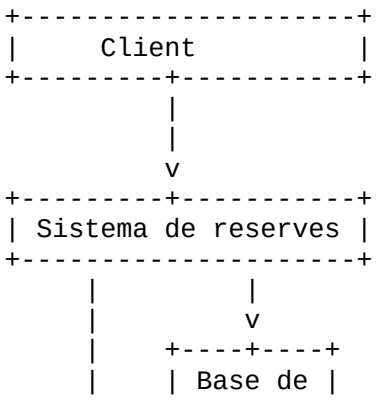
Els diagrames de comunicació (anteriorment coneguts com diagrames de col·laboració) mostren les interaccions entre objectes en una disposició gràfica on es posa èmfasi en la col·laboració entre els objectes per a aconseguir una funcionalitat.

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:



```
|    |  dades  |
+----+---------+
```

- 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.

3.3 Diferències clau

- Diagrama de seqüència: Enfocat en l'ordre temporal dels missatges.
- Diagrama de comunicació: Enfocat en la col·laboració estructural entre els objectes.

Cas d'ús 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.

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. Eclipse Modeling Framework (EMF)

L'Eclipse Modeling Framework (EMF) és una plataforma basada en el marc de treball Eclipse que facilita la construcció d'aplicacions i eines de modelatge de dades. Proporciona un conjunt d'eines i biblioteques per crear, editar i gestionar models de dades estructurats.

Característiques principals d'EMF

1. **Definició del Model**: Permet definir models de dades utilitzant diverses representacions, com ara models UML (Unified Modeling Language), esquemes XML (XSD) o directament a partir de codi Java.
2. **Generació de Codi**: A partir d'un model definit, EMF pot generar automàticament codi Java per a les classes del model, així com per a les eines de suport, com ara editors i validadors.
3. **Persistència**: Proporciona mecanismes per serialitzar i desserialitzar models de dades a partir de diferents formats, incloent XML i XMI (XML Metadata Interchange).
4. **Notificacions de Canvis**: Els objectes generats per EMF tenen suport integrat per a la notificació de canvis, la qual cosa permet que altres components del sistema (com ara editors de models) reaccionin automàticament als canvis en el model.
5. **Edició de Models**: Inclou eines per crear editors gràfics i textuals per als models, que es poden utilitzar dins de l'entorn Eclipse.

Components principals d'EMF

1. **Ecore**: És el meta-model d'EMF, una especificació de com es defineixen els models en EMF. Ecore és semblant a un model UML simplificat i proporciona les estructures bàsiques per definir classes, atributs, referències, etc.
2. **Editor d'Ecore**: Una eina visual dins de l'Eclipse IDE que permet als usuaris crear i editar models Ecore.
3. **Generador de Codi**: Una eina que pren un model Ecore com a entrada i genera el codi Java corresponent per a les classes del model i les eines associades.

Exemple d'ús d'EMF

Imagina que vols crear un sistema de gestió de biblioteques. Pots definir un model de dades que inclou entitats com "Llibre", "Autor" i "Usuari". A continuació, pots utilitzar EMF per:

1. **Definir el Model**: Crear un model Ecore que defineixi les classes i les seves relacions.
2. **Generar Codi**: Utilitzar l'eina de generació de codi d'EMF per crear classes Java per a les entitats del model.
3. **Crear Editors**: Desenvolupar editors gràfics o textuals dins de l'entorn Eclipse per permetre als usuaris crear, editar i gestionar les dades del sistema de biblioteques.

Beneficis d'utilitzar EMF

- **Automatització**: Redueix la necessitat d'escriure manualment codi repetitiu per a la manipulació de dades.
- **Consistència**: Assegura que el codi generat sigui consistent amb el model definit.
- **Eficiència**: Agilitza el procés de desenvolupament en proporcionar eines i estructures predeterminades per a les operacions comunes de manipulació de dades.

Cas d'ús pràctic

Un exemple pràctic podria ser el desenvolupament d'una aplicació de gestió d'inventari. Utilitzant EMF, un desenvolupador podria:

1. **Definir un model Ecore** amb entitats com "Producte", "Categoria" i "Proveïdor".
2. **Generar el codi Java** per a aquestes entitats i les operacions de CRUD (Crear, Llegir, Actualitzar, Eliminar).
3. **Crear una interfície d'usuari** dins de l'Eclipse IDE que permeti als usuaris gestionar l'inventari de manera visual i interactiva.

En resum, l'Eclipse Modeling Framework és una eina potent per al desenvolupament d'aplicacions basades en models, proporcionant una manera eficient i estructurada de gestionar dades complexes.
