Curs 20-21

Unitat 1. Desenvolupament de programari.

Entorns de desenvolupament

IES Jaume II El Just
Tavernes de la Valldigna

Continguts

1. Programes informàtics **4**
1.1 Programari i programes informàtics . . . 4
1.2. Algorismes i la seua representació . . . 5

2. Llenguatges de programació **6**
2.1. Concepte de llenguatge de programació . . . 6
Llenguatge de programació i llenguatge informàtic . . . 7
2.2. Classificació dels llenguatges de programació . . . 7
Atenent al nivell d'abstracció . . . 7
Atenent al propòsit . . . 8
Atenent a l'evolució històrica . . . 9
Segons la forma d'executar-se . . . 9
Atenent a com aborden la tasca a realitzar . . . 10
Atenent al paradigma de programació . . . 10
Atenent al lloc d'execució . . . 10
2.3. Quins són els llenguatges més importants? . . . 11

3. Execució de programes **13**
3.1. Traductors: Compiladors i Intèrprets . . . 13
3.2. Codi font, codi objecte i codi executable . . . 13
Màquines virtuals. El cas de Java. . . . 15
Compatibilitat entre sistemes. El concepte de multiplataforma. . . . 15
3.3. Distribució d'aplicacions . . . 16
3.4. Programari lliure i propietari . . . 17

4. Desenvolupament d'aplicacions **18**
4.1. Fases en el desenvolupament d'una aplicació . . . 18
Fase d'Anàlisi . . . 18
Fase de Disseny . . . 19
Fase de Codificació/Implementació . . . 19
Fase d'Explotació o Publicació . . . 20
Fase de manteniment . . . 20
4.2. Models de desenvolupament de programari . . . 20
4.2.1. Cicle de vida en cascada . . . 20
4.2.2. Models de desenvolupament evolutiu . . . 21
4.2.3. Models de desenvolupament àgil . . . 25

4.3. Equips de treball. Funcions i responsabilitats dels membres 27
Equips de treball tradicionals 27
Organització d'equips en Scrum 27

1. Programes informàtics

1.1 Programari i programes informàtics

Segons l'estàndard IEEE 729-1983 de l'*Electrical and Electronics Engineers* (Institut d'Enginyers Elèctrics i Electrònics), dedicat principalment a la creació i difusió d'estàndards, un **programa informàtic** es defineix com:

> Una *seqüència d'instruccions* que poden ser processades per un ordinador. Aquest processament pot incloure l'ús d'un *assemblador*, un *compilador*, un *intèrpret* o un *traductor* per tal de preparar el programa per a la seua *execució* i per tal d'executar-lo.

Per la seua banda, el *Termcat* (Centre de terminologia en català), defineix **programa** com a:

> Conjunt de *dades i d'instruccions codificades* que són l'expressió completa d'un procediment que pot executar un sistema informàtic.

Com veiem, ambdues definicions remarquen que es tracta d'un *conjunt* o més precisament *seqüencia* (ja que l'ordre importa) d'instruccions que indiquen un procediment a realitzar per part d'un sistema informàtic o ordinador. Podríem dir que un programa s'assembla a una recepta de cuina, a la qual s'indiquen els ingredients (*dades*) per una banda i per altra els passos ordenats (*seqüència d'ordres*) que s'han de seguir per treballar els ingredients (*dades*) per obtindre un plat. Més endavant tornarem a utilitzar aquesta comparació.

Per la seua banda, el **programari** (*software* en anglès), és definit per l'IEEE 729 com:

> Programes, procediments, regles i documentació associada pertanyent a l'operació d'un sistema informàtic.

El *Termcat* fa una definició bastant semblant:

> Conjunt dels programes i de les dades que governen el funcionament d'un *processador*.

Com veiem, és un terme un tant més extens que programa, ja que no només fa referència a un sinó a diversos programes més altre contingut, com pot ser la documentació i altres tipus de fitxers. El terme *programari* es complementaria amb el de *maquinari* (*hardware* en anglès) per conformar el que seria un sistema informàtic. El *maquinari* faria referència a tot allò físic, tangible, que es pot *tocar*, mentre que el programari és allò intangible, que *no es pot tocar*, i que fa funcionar el *maquinari*. Com veiem, són termes complementaris, ja que un sense l'altre no té sentit.

Un altre terme relacionat amb els anteriors és el d'**aplicació informàtica**. Segons el Termcat, una **aplicació informàtica** es defineix com:

> Conjunt de programes informàtics que desenvolupen tasques específiques en un ordinador.

Com veiem, es tracta d'una definició bastant genèrica. Què significa això de que *desenvolupen tasques específiques?*. Açò ens porta a veure una xicoteta classificació del *programari*, que podem dividir en dos grans blocs: programari *de sistema* i programari d'*aplicació*. Basant-nos en les definicions de l'IEEE 927:

- **Programari de sistema**: Programari dissenyat per a un sistema informàtic o família de sistemes informàtics en concret, per tal de facilitar el funcionament i manteniment d'un sistema informàtic i els programes associats. Seria programari de sistema, per exemple, els sistemes operatius, els compiladors, les utilitats del sistema, els controladors o *drivers*.
- **Programari d'aplicació (application software):** Programari creat de forma específica per al seu ús en un sistema informàtic. Es tractaria de les *aplicacions* en si, i dins d'aquest entrarien els processador de textos, editors de codi, fulls de càlcul, etc.

1.2. Algorismes i la seua representació

Com hem dit, un programa és una seqüència d'ordres o instruccions que poden ser processades per un ordinador. Aquests programes, generalment tenen la finalitat de facilitar o automatitzar algun procediment que sense ells serien bastant tediosos.

Doncs bé, aquests procediments, s'expressen mitjançant el que es coneix com *algorismes*. Segons l'IEEE, una de les definicions d'*algorisme* és la següent:

> Un conjunt finit de regles ben definides per tal de trobar la solució a un problema en un nombre finit de passos o per tal de realitzar una tasca específica.

Per exemple, l'algorisme per fer una paella seria el següent:

1. Posar l'oli a la paella, encendre el foc i esperar que s'escalfe.
2. Sofregir el pollastre.
3. Sofregir el conill.
4. Sofregir la bajoqueta i el garrofó.
5. Sofregir la tomata rallada.
6. Afegir el pimentó al sofregit.
7. Afegir l'aigua i deixar coure.
8. Afegir l'arròs i esperar que es coga.

Com veiem, es tracta d'una representació de l'algorisme/recepta en llenguatge natural. Com veurem en altres mòduls, els algorismes es poden expressar d'una forma més propera a l'ordinador, que

anomenarem *pseudocodi*, i també de forma gràfica, amb el que es coneixeran com *diagrames de flux*. Veiem un exemple a la figura 1.

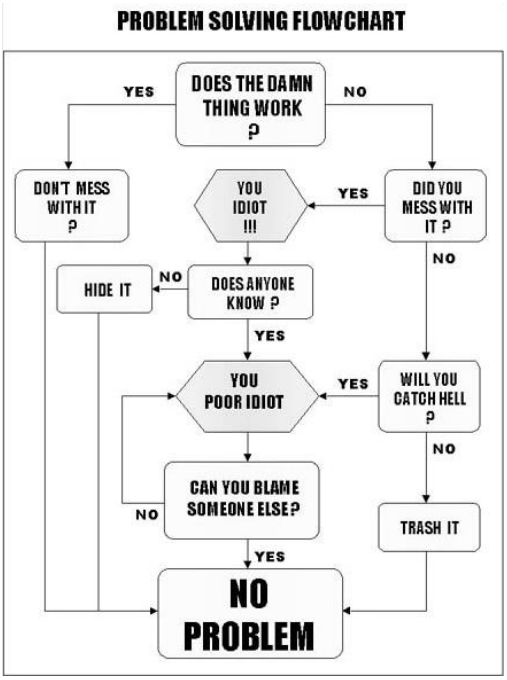

Figura 1: Diagrama de flux per resoldre problemes

Ara que hem vist la definició d'algorisme, podem afegir una nova definició per al terme *programa* com segueix:

> Un *programa* és l'expressió d'un algorisme en determinat *llenguatge de programació*.

Com hem vist, un algorisme pot expressar-se de forma gràfica o textual, en pseudocodi. Per tal de crear doncs un programa a partir d'un algorisme, només haurem de *codificar* aquest en determinat llenguatge, més proper a l'ordinador: un *llenguatge de programació*, concepte que veurem als següents apartats.

2. Llenguatges de programació

2.1. Concepte de llenguatge de programació

Segona la definició de la IEEE 729, un llenguatge de programació és:

> Un **llenguatge artificial** dissenyat per generar o expressar programes.

És a dir, es tracta d'un llenguatge dissenyat per especificar eixes accions que s'han d'executar en un sistema informàtic per resoldre un problema, o generar un resultat a partir d'un conjunt de dades d'entrada.

Llenguatge de programació i llenguatge informàtic

Sovint se sol confondre el concepte de *llenguatge de programació* amb el de *llenguatge informàtic*, però es tracta de conceptes diferents.

Un *llenguatge informàtic* és tot aquell llenguatge utilitzat per o associat amb ordinadors. Un llenguatge de programació compleix aquesta definició, pel que un llenguatge de programació sí que seria un llenguatge informàtic però no tots els llenguatges informàtics han de ser llenguatges de programació.

Per posar algun exemple de llenguatge informàtic que no és un llenguatge de programació tenim tot el ventall de llenguatges de marques, com puguen ser XML o HTML.

2.2. Classificació dels llenguatges de programació

Podem classificar els llenguatges de programació atenent a diversos criteris. En la pràctica, molts llenguatges no podran classificar-se únicament en una categoria, ja que sovint incorporen característiques d'altres llenguatges i filosofies de programació. No obstant això, poder establir una classificació d'aquests ens resulta útil per tal de conéixer els rangs, possibilitats o tipus de llenguatges existents.

Atenent al nivell d'abstracció

Una abstracció és una *visió d'un problema que extrau la informació essencial i rellevant per a determinat propòsit, i ignora la resta d'informació*. El procés d'abstracció (de formar una abstracció) es realitza cada vegada que formulem un problema, té com a ressultat una expressió més o menys específica o propera al llenguatge de la màquina.

Per exemple, per a la recepta de la paella, en lloc d'expressar el pas *2. Sofregir el pollastre*, podríem haver indicat:

- *Incorporar el pollastre a la paella, una vegada l'oli estiga calent*
- *Mentre el pollastre no estiga daurat*

  - *Moure el pollastre per a que es sofrisca per tots els costats*
  - *Esperar*

Doncs bé. Atenent al nivell d'abstracció, podem classificar els llenguatges de programació en:

- **Llenguatges de nivell baix**: Són aquells amb baix nivell d'abstracció, més propers a l'ordinador, i específics per a cada tipus de processador. Un clar exemple és el llenguatge *assemblador (assembly)*.
- **Llenguatges d'alt nivell**: Tenen un major nivell d'abstracció, pel que es troben relativament més a prop del llenguatge humà, i més lluny del de la màquina. Llenguatges d'aquest tipus poden ser *Java* o *python*.
- **Llenguatges de nivell intermedi**: A cavall entre els anteriors, trobem els llenguatges de nivell intermedi, que permeten un major grau d'abstracció que els de baix nivell, però al mateix temps mantenen algunes de les qualitats dels llenguatges de bai nivell. Un llenguatge d'aquest tipus per excel·lència seria *C*. Alguns autors, només distingeixen entre llenguatges d'alt i baix nivell. En aquest cas, el llenguatge *C* es consideraria un llenguatge d'alt nivell.

Atenent al propòsit

Segons el tipus de problemes o camp en què es treballe, podem classificar els llenguatges de programació com:

- **Llenguatges de propòsit general**: Dissenyats per realitzar qualsevol tipus de tasca. Els millors exemples serien C, C++, C# o Java.
- **Llenguatges de propòsit específic**: Dissenyats i optimitzats per a un objectiu concret. Podria ser el cas de *Matlab*, dissenyat específicament per a l'àmbit de les matemàtiques i el càlcul amb matrius.

Tot i que aquesta seria la classificació més habitual en quant a propòsit, també solen incloure's dues categories més, complementàries i no excloents amb les anteriors:

- **Llenguatges de programació de sistemes**: Dissenyats específicament per a la programació de sistemes operatius o drivers. El cas més clar seria el de *C* (que també és de propòsit general).
- **Llenguatges d'script**: Pensats per realitzar tasques de control i auxiliars. Antigament es coneixien com *llenguatges de processament per lots (batch).* Dins aquesta categoria inclouríem scripts en *Shell*/*Bash* per a realitzar tasques del sistema, o scripts *javascript* per gestionar el comportament de pàgines web. Cal dir que *Javascript* ha donat el bot com a entorn de programació fora de la web, gràcies a *nodejs*, com veurem més endavant.

Atenent a l'evolució històrica

A mesura que evoluciona el món de la informàtica, els llenguatges de programació van guanyant en nivell d'abstracció. Podem establir actualment fins a cinc generacions de llenguatges de programació, tenint en compte, que són llenguatges nous que van apareixent, però que no substitueixen a les generacions anteriors.

- **Llenguatges de Primera Generació 1GL:** Codi màquina, interpretable directament pel microprocessador. En aquest temps solien utilitzar-se targetes perforades per emmagatzemar els programes.
- **Llenguatges de Segona Generació 2GL:** S'introdueix el llenguatge *assemblador*, que incorpora *mnemònics* per representar les instruccions del codi màquina.
- **Llenguatges de Tercera Generació 3GL:** Es tracta de llenguatges moderns, dissenyats per tal de facilitar la programació als humans. Són els més utilitzats actualment, com *C* o *Java*.
- **Llenguatges de Quarta Generació 4GL:** Es tracta de llenguatges amb un propòsit més concret, com puguen ser els llenguatges *Natural* (per a bases de dades), *Mathematica* o *Matlab*.
- **Llenguatges de Cinquena Generació 5GL:** Fan referència a llenguatges de programació més propers a la lògica humana, pensats per a que el programador especifique el problema a resoldre i les condicions que aquest ha de reunir, i la màquina el resolga. Un exemple de llenguatge de programació lògica és *Prolog*.

Segons la forma d'executar-se

Tot i que veurem açò amb més detall en apartats posteriors, segons com s'execute un programa, podrem distingir entre:

- **Llenguatges compilats**: Aquells que han de passar per un procés de traducció (compilació) per passar del llenguatge de programació al llenguatge (generalment) màquina. Un clar exemple és *C*.
- **Llenguatges interpretats**: Aquells que no necessiten de cap procés de compilació, però sí un *intèrpret* que *entenga* aquest llenguatge i el vaja traduint instrucció a instrucció a la màquina. Un exemple seria *python*.

Un cas especial, com és el cas de Java, és el dels llenguatges que requereixen d'ambdós procediments. Com veurem més endavant, Java requereix d'un procés de compilació, però no a codi màquina, sinó a un tipus de codi especial anomenat *bytecode*, que és interpretat per la *màquina virtual de java*.

Atenent a com aborden la tasca a realitzar

Segons com els llenguatges de programació plantegen els problemes, podem distingir:

- **Llenguatges imperatius**: Aquells que indiquen *com* cal realitzar una tasca, expressant els passos a realitzar. Ex: *C, Java…*
- **Llenguatges declaratius**: Aquells que indiquen què cal fer, sense preocupar-se del com: *Lisp, Prolog*

Atenent al paradigma de programació

A l'estil de programació que s'utilitza es coneix com *paradigma de programació*. Alguns llenguatges suporten només un paradigma, i d'altres poden suportar-ne diversos. Històricament, els paradigmes han anat apareixent per facilitar la tasca del programador en un moment donat o resoldre determinats tipus de problemes.

Segons el paradigma utilitzat, podem establir els següents tipus de llenguatges:

- **Llenguatges de programació procedurals**: Divideixen el problema en parts més menudes o subprogrames, que s'invoquen unes a altres per ser executades. Ex: *C*, *Pascal*.
- **Llenguatges de programació orientada a objectes**: Segueixen un enfoc més semblant al món real, i modelen objectes que tenen determinat comportament. Ex. *Java*, *C#*. El llenguatge *C++* en aquesta classificació seria un llenguatge híbrid, ja que suporta programació orientada a objectes, però pot seguir un paradigma procedural.
- **Llenguatges de programació funcional**: Realitzen les tasques avaluant funcions (com en matemàtiques) de forma recursiva. Ex. *Lisp*.
- **Llenguatges de programació lògica**: Expressen les tasques a realitzar fent ús de la lògica formal matemàtica. Ex. *Prolog*.

Atenent al lloc d'execució

Quan parlem d'un sistema informàtic aïllat, està clar que els programes s'executen en el propi sistema, però en el cas dels sistemes informàtics distribuïts, on els programes poden executar-se tant en un servidor com en un client, podem distingir dos tipus més de llenguatges:

- **Llenguatges de servidor (*server-side*)**, que s'executen potencialment en un servidor, com és el cas de *PHP* per a servidors web.
- **Llenguatges de client (*client-side*)**, que s'executen potencialment en el client. Al cas del servei web, el client seria el propi navegador web, qui executaria codi *javascript*.

Cal remarcar ací el cas de *Java*, que comprén tots els àmbits, mitjançant *JSP* (*Java Server Pages*) per a la programació del costat del servidor web, o els *servlets* de *java*, xicotetes aplicacions que s'executen en l'entorn del client.

Un altre cas a destacar és el de *javascript* (compte, que no té rea a veure amb java, no es deixeu confondre pel nom). Tot i que *javascript* ha estat tradicionalment lligat a la programació del costat del client, actualment, de la mà del framework *nodejs*, ha donat el bot tant a entorns d'escriptori com de servidor, poden implementar serveis i servidors web d'una forma si cal més eficient que amb els serveis tradicionals.

2.3. Quins són els llenguatges més importants?

Aquesta és una pregunta complicada de contestar. Posats a ensenyar-nos un nou llenguatge, quin agafaríem?

Per a això, caldria vore, primerament per a què volem programar, i una vegada clar, mirar dins dels llenguatges específics quina és la millor opció o ens ofereix més facilitats a l'hora de desenvolupar aplicacions.

Una bona opció també és veure com es mou el mercat. Si donem un cop d'ull a la web *Octoverse* de *Github*, podrem trobar molta informació interessant. Anem per parts. *Github* és el major repositori de projectes basats en programari lliure, i ens ofereix, a banda d'emmagatzemar el codi dels nostres projectes, portar el control de versions, i moltes més coses que veurem més endavant durant el curs. A partir de tots aquests projectes, *Github* elabora aquest *Octoverse*, una espècie d'informe anual sobre els projectes emmagatzemats en ell.

D'entre tota la informació que proporciona *Github* amb *Octoverse* trobem la classificació dels llenguatges més utilitzats als últims anys en els projectes que emmagatzema, així com els llenguatges que més creixement estan tenint durant l'últim any:

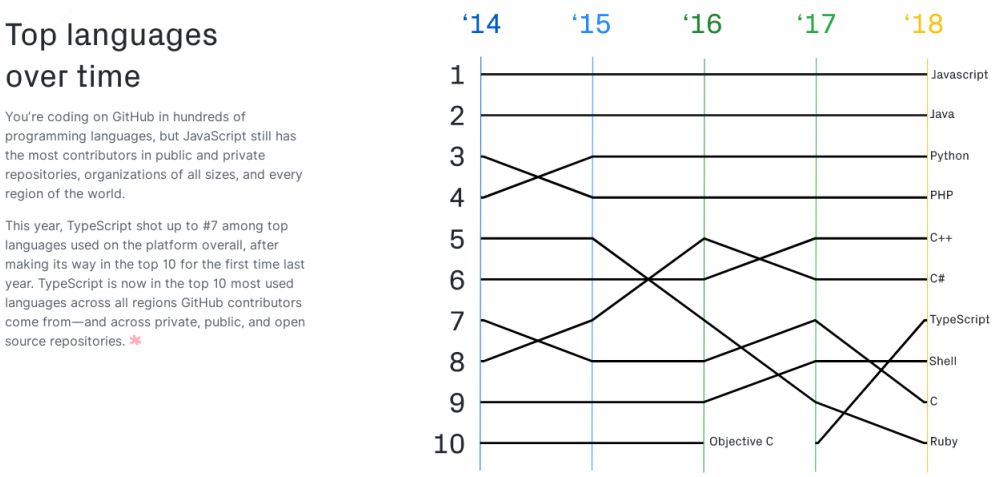

Figura 2: Llenguatges més utilitzats a Github als últims anys

Com veiem, el llenguatge més utilitzat als projectes de Github és *javascript*, seguit de *Java* i *Python*.

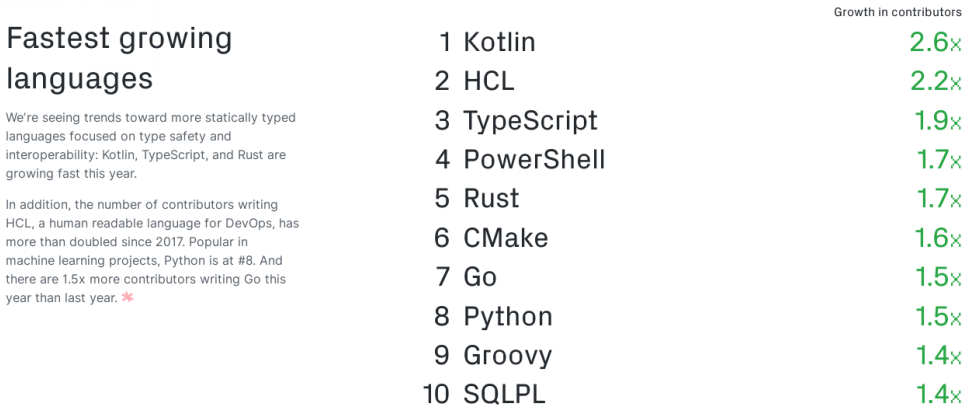

Figura 3: Llenguatges que més creixen a Github

En aquest segon gràfic, veiem com durant el 2017, el llenguatge que major creixement ha experimentat és *Kotlin*, una alternativa a *Java* per a la programació per a *Android*, seguit d'*HCL*, llenguatge per a la computació en el núvol (*cloud computing*) i *Typescript*, una alternativa a *Javascript* amb suport de tipus de dades.

> **Activitat** Investigueu sobre els deu llenguatges més utilitzats segons Octoverse, i feu una xicoteta descripció d'ells, indicant en quines de les categories anteriorment esmentades es trobaria cadascun. Podeu seguir la següent plantilla:

> - Llenguatge: Nom del llenguatge
> - Descripció breu: Descripció del llenguatge
> - Nivell d'abstracció: Alt/Baix/Intermedi
> - Propòsit: General/Específic
> - Generació: 1GL/2GL…
> - Com s'executa: Compilat/Interpretat
> - Com aborda la tasca a executar: Imperatiu/Declaratiu
> - Paradigma de programació: Procedural/O.O./Declaratiu…
> - On sol utilitzar-se: Client/Servidor/Sistema aïllat…

3. Execució de programes

Com hem comentat, un programa expressat en un llenguatge d'alt nivell no pot ser executat directament per l'ordinador, sinò que requereix d'un procés de *traducció* d'aquest codi d'alt nivell a codi màquina, interpretable directament per l'ordinador.

3.1. Traductors: Compiladors i Intèrprets

Els programes que realitzen aquest procés de traducció d'un programa expressat en llenguatge d'alt nivell (*codi o llenguatge font*) s'anomenen *traductors*, i poden ser de dos tipus:

- **Compiladors**: Tradueixen només una vegada el programa original, i creen un nou fitxer *executable* que pot utilitzar-se tantes vegades com es desitge. Durant aquest procés de traducció es poden determinar determinats tipus d'errors de programació. Els llenguatges compilats més coneguts són *C* i tots els seus derivats com *C++* o *C#*.
- **Intèrprets**: Tradueixen i executen cada línia del programa original abans de traduir i executar la següent. No generen per tant cap fitxer executable nou, ja que de per si, mitjançant l'intèrpret el programa en si podem *executar* el programa. En aquest cas, si no hi ha un preprocessat previ, els errors en el programa no es detecten fins que no s'està executant aquest. Alguns dels llenguatges interpretats més coneguts són *Python* o els scripts en *Shell* o *Bash*.

3.2. Codi font, codi objecte i codi executable

En aquest apartat veurem els conceptes de codi font, objecte i executable, centrant-nos principalment en els llenguatges compilats (ja que als intèrprets la traducció és automàtica).

Segons l'IEEE 729, un **programa font** (*source program*) és:

> (1) Un programa que ha de ser compilat, assemblat o interpretat abans de ser executat per un ordinador- (2) Un programa d'ordinador expressat en un llenguatge font. Contrast amb *programa objecte*.

És a dir, quan parlem del **codi font** d'un programa ens referim al conjunt de línies de text que representen les instruccions per a l'ordinador, expressades, com indica l'estàndards IEEE 927 en un *llenguatge font*. Serà, generalment, codi escrit en un llenguatge de programació que requerisca ser compilat, com pot ser C, C++ o Java.

Per la seua banda, l'estàndard defineix com a **programa objecte**:

> Un programa completament compilat o assemblat que està preparat per ser carregat en un ordinador.

És a dir, el **codi objecte** és aquell resultant d'un procés de compilació del codi font. Aquest codi objecte serà expressat generalment en *llenguatge màquina* o *bytecode*, i es distribueix en un o diversos fitxers corresponents al codi font compilat.

Aquest *codi objecte*, no és executable encara per l'ordinador, sinó que requereix d'una fase d'*enllaçat* per combinar tots els fitxers objecte obtinguts en la fase de compilació per tal de generar finalment el **codi executable.**

Gràficament, podríem veure aquest procés de la següent manera:

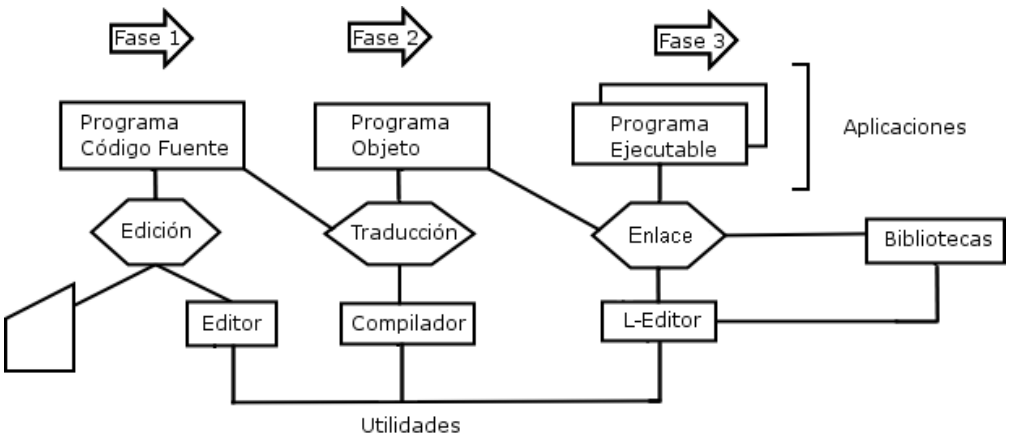

Figura 4: https://es.wikipedia.org/wiki/C%C3%B3digo_objeto

Com veiem, a més, en el gràfic es mostren les tres fases per les que passa el procés de generació de codi executable: *Fase d'edició*, *fase de traducció* i *fase d'enllaçat*. En cadascuna d'elles, disposem d'un tipus de codi (font, objecte o executable), i necessitarem d'un tipus d'eines concretes (editor, compilador i enllaçador).

Si ens fixem en la fase d'enllaçat, a més apareix el concepte de *Biblioteca*. Cal dir, que en aquesta fase, a més d'enllaçar tots els fitxers objectes que ha generat la nostra aplicació, també s'enllacen les llibreries. Com es veurà més endavant, les llibreries són fitxers objecte que ens proporcionen certes funcions d'utilitat per als nostres programes.

Màquines virtuals. El cas de Java.

Quan parlem de **màquina virtual**, solem referir-nos a un programari especial, que emula el hardware d'un ordinador, i que ens permet executar determinat programari (habitualment un sistema operatius o serveis concrets) com si estiguérem executant-lo en l'ordinador emulat.

Aquest mateix concepte s'aplica també a l'àmbit de la programació, i Java, amb la seua *JVM* (*Java Virtual Machine*) n'és l'exemple més clar. El que es pretén amb aquesta idea és compilar els fonts per a una màquina virtual determinada, que no té ni tan sols per què existir físicament. En el procés de compilació, en lloc de generar codi màquina concret, es genera un codi especial, que podríem dir que és el llenguatge màquina d'aquesta *màquina virtual*. Així doncs, és la màquina virtual, qui en aquest cas, **interpreta** aquest codi generat, en funció del sistema operatiu i l'arquitectura que estiga per baix.

Al cas de *Java*, aquest codi intermedi s'anomena ***bytecode***, i és independent de la plataforma: el mateix bytecode ens aprofita per a diferents sistemes. Serà doncs, la *JVM*, específica, ara sí en cada sistema i arquitectura, d'*interpretar* aquest *bytecode* per traduir-lo al llenguatge màquina.

Compatibilitat entre sistemes. El concepte de multiplataforma.

Amb tot el que hem vist fins ara, podem plantejar-nos la següent pregunta: *Si el codi compilat està expressat en llenguatge màquina, és a dir, és executable directament pel processador ¿ens pot aprofitar un executable creat en un sistema operatiu per a un altre?*

L'experiència ens diu, que evidentment, no, però… podríem explicar per què? La resposta està en el mateix procés de compilació. La compilació d'un programa és una tasca molt complexa. S'ha d'analitzar el codi diverses vegades, detectant els diferents elements que el componen, gestionant la memòria que aquest necessitarà, i moltes coses més. Amb tot aquest procés, el compilador, quan el programa necessita accedir als recursos que gestiona el sistema operatiu (com per exemple accedir al disc o escriure per pantalla), incorpora el que es coneixen com *crides al sistema*. Aquestes crides, són específiques de cada sistema operatiu, ja que cada sistema gestiona els recursos d'una manera. Aquest és el motiu pel qual un executable de *Windows* no es pot executar directament en *Linux* o *Mac*.

Concepte de multiplataforma

La *Wikipèdia* ens dóna una bona aproximació a aquest concepte:

> Multiplataforma és un terme informàtic que s'utilitza per definir al programari, ja sigui un sistema operatiu, llenguatge de programació, programa, etc. que pot ésser executat en diverses plataformes. Una aplicació multiplataforma pot executar-se en totes les plataformes més comunes o simplement en més d'una. Per exemple una aplicació multiplataforma seria capaç d'executar-se a Windows, Linux i Mac OS X, ja sigui en un PowerPC o un X86. Alguns exemples de programes multiplataforma són: LibreOffice, Mozilla Firefox, GIMP, Plucker, Skype, Opera, generalment de programari lliure.

> **Activitat**
>
> Investigueu sobre les algunes de les aplicacions multiplataforma comentades a la definició anterior, i esbrineu en quin llenguatge de programació estan desenvolupades. Quin tipus de llenguatges creieu que són més convenients per al desenvolupament d'aplicacions multiplataforma?

3.3. Distribució d'aplicacions

Una vegada som capaços de generar un executable, ens queda un últim pas per fer que els clients o futurs usuaris puguen disposar d'elles: l'empaquetat i distribució de l'aplicació.

Quan parlem d'empaquetat, ens referim a proporcionar forma de *paquet* a les aplicacions (*software o application bundle*). En els paquets de programari, a més dels executables de l'aplicació, s'inclouen les llibreríes de què depèn (bé incorporades o enllaçades), junt amb altres fitxers com contingut multimèdia, fitxers de traducció, entrades de menú, etc. de manera que es proporciona tot com un conjunt.

El fet d'empaquetar aplicacions evita problemes de dependències, tant d'instal·lació com d'execució, ja que els paquets fan referència a tot el programari addicional necessari per poder funcionar, i el sistema que ho gestiona ja s'encarrega d'instal·lar-lo per nosaltres.

Per la seua banda, quan parlem de *sistemes de gestió de paquets*[1] fem referència a les eines que ens serveixen per automatitzar el procés d'instal·lació, actualització, configuració i esborrat de paquets de programari.

Al món de GNU/Linux hi ha tradicionalment dos tipus bàsics de sistemes d'empaquetat: el format *debian (.deb)* de la branca *Debian/Ubuntu* i el format *RPM*, de *Red Hat/Fedora/CentOS*, cadascun

[1] https://es.wikipedia.org/wiki/Sistema_de_gesti%C3%B3n_de_paquetes

amb unes eines concretes per gestionar els paquets. Als últims anys, han aparegut nous formats d'empaquetat *autocontinguts*, com és el cas de *Snap*, *appImage* o *flatpak*, que permeten utilitzar-se en qualsevol sistema GNU/Linux.

Aquests paquets de programari s'organitzen en *repositoris* o *dipòsits* de programari, que poden ubicar-se tant a dispositius d'emmagatzemament com a la xarxa, i ser accessibles des de qualsevol lloc. Als sistemes basats en debian, disposem del fitxer `/etc/apt/sources.list` i el directori `/etc/apt/sources.list.d/` on s'especifiquen tots els repositoris disponibles a l'ordinador (tant de binaris com de fonts), podent incorporar nous repositoris per tal d'afegir programari nou.

3.4. Programari lliure i propietari

Ara que ja sabem diferenciar entre codi font, codi objecte i codi executable, podem fer una xicoteta introducció al programari lliure i propietari.

El concepte de programari lliure, va ser introduït per Richard Stallman, i parla de quatre llibertats que pot exercir qui rep el programari:

1) **Llibertat d'execució**: El programari es pot executar on siga i amb qualsevol propòsit,
2) **Llibertat d'adaptació**: El programari pot estudiar-se i adaptar-se a les necessitats de cadascú, la qual cosa exigeix que el codi font siga accessible.
3) **Llibertat de redistribució**: El programari es pot redistribuir, ajudant la comunitat,
4) **Llibertat de millora**: El programari es pot millorar i publicar aquestes millores, la qual cosa també exigeix l'accés al codi font.

Per tal de garantir aquestes llibertats, el programari es distribueix amb una llicència determinada, on l'autor expressa el seu permís per tal que el receptor del programa puga exercir estes llibertats, i pot incloure algunes restriccions (com donar crèdit als autors originals o que no s'utilitze per a finalitat comercial). Existeixen moltes llicències de programari lliure, però totes elles, han de respectar les quatre llibertats indicades anteriorment.

Sovint el programari lliure sol confondre's amb programari gratuït, però no té per què ser així. El terme *free software* fa referència a les quatre llibertats expressades anteriorment. El terme *free* en aquest cas no té per què significar *gratis*: *free as in freedom, not as free beer*. De tota manera, com que la distribució del programari està garantida, es pot distribuïr, fins i tot, de forma gratuïta.

Són molts els avantatges que proporciona el programari lliure: el fet de disposar del codi fa que el puguem adaptar a les nostres necessitats o models de negoci, i que el codi siga revisat per qualsevol persona, el que aporta major qualitat al programari, així com transparència i seguretat a aquest (no hi ha *portes del darrere* que puguen filtrar informació per exemple).

Front a això, el programari propietari no ofereix els fonts, el que impossibilita correccions o adaptacions per part de tercers, de manera que no sabrem realment mai *què està fent el programari amb les dades que li proporcionem*.

4. Desenvolupament d'aplicacions

El desenvolupament d'una aplicació informàtica requereix d'un procés, també anomenat cicle de vida del programari. Aquest procés passa per diverses fases, i involucra a diferents actors. Hi ha diversos models per a aquest procés, cadascun d'ells amb una visió diferent i una forma concreta d'abordar el propi desenvolupament. En aquest apartat en veurem alguns d'ells.

4.1. Fases en el desenvolupament d'una aplicació

Com hem dit, el desenvolupament de programari passa per diverses fases, que aborden des de la concepció del programari fins el resultat final. Les grans fases en les que tradicionalment s'ha dividit el desenvolupament de programari són: *Anàlisi*, *Disseny*, *Codificació*, *Implantació* i *Manteniment*.

Tot seguit veurem aquestes fases, tenint en compte que, depenent del paradigma de programació en què ens basem (procedural/estructurat, orientat a objectes…), farem ús d'unes o altres tècniques i eines.

Fase d'Anàlisi

En aquesta fase es determina principalment *què* anem a desenvolupar. Es realitza una anàlisi del problema i els *requeriments* que ha de tindre el programari, així com una recerca de possibles solucions en el mercat. En aquesta fase també s'especificaran els models de dades i comportament que tindrà la nostra aplicació.

D'entre les eines més utilitzades en aquesta fase podem trobar:

- **Diagrames per especificar el comportament del sistema**, com puguen ser els *Diagrames de flux de dades*, en la metodologia d'*anàlisi estructurada*, o els *diagrames de casos d'ús*, *diagrames de seqüència*, o *diagrames d'estats*, en metodologies orientades a objectes.
- **Diagrames per especificar models de dades**, per al de conéixer les estructures de dades de l'aplicació i les seues relacions, com poden ser els *diagrames Entitat-Relació*, en l'àmbit de les bases de dades o els *diagrames de classes*, en metodologies orientades a objectes, o bé els *diccionaris de dades* en programació estructurada.

- **Eines per descriure les interfícies d'usuari**, per tal de definir la informació d'entrada i eixida de dades, com puguen ser els *wireframes* (*esquemes de pàgina* o *plànols de pantalla*) per descriure l'esquelet o estructura visual d'un lloc web, o bé *prototipus*, que són com una xicoteta *demo* del que seria el producte final.

En aquesta fase d'anàlisi, també cabria incloure un anàlisi de la *viabilitat* del projecte, per determinar si aquest és factible o no. En cas que el producte siga viable, tant econòmicament com respecte als recursos, cal una fase de *Planificació*, per tal de determinar la direcció del projecte i els terminis i assignació de recursos previstos per a fases posteriors.

Fase de Disseny

Una vegada tenim clars els requeriments, cal anar acostant-nos al que serà el producte final, tenint clars els recursos del sistema, tant físics (tipus d'ordinador, perifèrics, etc.) com lògics (Sistema Operatiu, llenguatge de programació, llibreries, Sistemes Gestors de Bases de Dades).

En aquesta fase, creem la solució al problema a un alt nivell, perfilant els documents que hem elaborat en la fase d'anàlisi, així doncs, distingim:

- **Disseny de les estructures de dades**, on perfilem els models de dades per a la seua persistència en funció del suport que aquestes tindran: fitxers, bases de dades relacionals, orientades a objectes, noSQL… així com les estructures de dades a nivell de codi, i en el cas de la programació orientada a objectes, la definició de les diferents classes que compondran l'aplicació.
- **Disseny del comportament**, on perfilarem el funcionament del sistema a partir de la documentació de la fase d'anàlisi, i com aquest sistema s'organitza. Per a això s'utilitzen tècniques de disseny modular (divisió descendent de l'aplicació en diferents mòduls funcionals), i disseny procedimental, on especifiquem el funcionament de cadascun d'aquests mòduls mitjançant algorismes, bé en diagrames de flux o pseudocodi. Al cas de la programació orientada a objectes, dissenyarem el funcionament dels mètodes que suportaran els objectes que componen la nostra aplicació, que són els qui, al final defineixen el seu comportament.

Fase de Codificació/Implementació

En aquesta fase, es converteixen els algorismes dissenyats en la fase anterior a programes en un llenguatge de programació concret. A més, aquesta fase inclou la realització de proves per tal d'assegurar la qualitat i estabilitat del programari, i consisteixen principalment en comprovar, per una banda que cada mòdul o component realitza la seua tasca correctament (*proves unitàries*), i per altra banda, que els diferents mòduls treballen en conjunt com un únic sistema (*proves d'integració*).

Fase d'Explotació o Publicació

Una vegada codificada i provada l'aplicació, es realitza la fase d'implantació en els sistemes on aquestes han de funcionar, per tal de comprovar el seu funcionament. A més de la instal·lació en sí del programari, es realitzen proves d'acceptació del nou sistema entre els usuaris, migració d'informació entre el sistema anterior (si hi ha) i el nou, i l'eliminació del sistema anterior. A més, caldrà proveir als usuaris de la documentació necessària per tal de realitzar una correcta explotació i bon ús la sistema.

Fase de manteniment

El programari no sol ser perfecte, pel que és fàcil que a mesura que l'aplicació s'està utilitzant, apareguen errades que no s'havien detectat prèviament, es milloren funcionalitats o s'incorporen noves funcionalitats. En general, podem parlar de diferents tipus de manteniment:

- **Manteniment correctiu**: On corregim errors no detectats en les proves que van apareixent amb l'ús.
- **Manteniment adaptatiu**: Al qual cal modificar el programa per adaptar-lo a nous entorns: llibreries, entorn gràfic, actualitzacions del sistema operatiu, etc.
- **Manteniment perfectiu**: On afegim noves funcionalitats a l'aplicació en base a propostes o nous requeriments de funcionalitat per part dels usuaris o clients.

El manteniment correctiu ens torna a la fase de codificació de l'aplicació, on realitzem les modificacions oportunes, tornem a realitzar les proves i implantem la nova versió de l'aplicació. En canvi, el manteniment adaptatiu i perfectiu ens fan tornar més endarrere en el cicle de vida, ja que impliquen modificacions tant en l'anàlisi com en el disseny de l'aplicació.

4.2. Models de desenvolupament de programari

En l'apartat anterior hem vist les diferents fases per les que passa el desenvolupament de programari[2]. Quan parlem de *models de desenvolupament de programari*, farem referència a diferents estratègies per tal d'aplicar estes fases, mitjançant diferents eines.

4.2.1. Cicle de vida en cascada

El cicle de vida en cascada té un punt de vista sistemàtic i seqüencial per al desenvolupament del programari, que comença en un nivell de sistemes i progressa amb l'anàlisi, disseny, codificació,

[2] https://es.wikipedia.org/wiki/Proceso_para_el_desarrollo_de_software#Modelos_de_Desarrollo_de_Software, https://ca.wikipedia.org/wiki/Proc%C3%A9s_de_desenvolupament_de_programari

proves i manteniment. Es basa en el fet que cada fase s'ha d'acabar per tal de continuar amb la següent, i la documentació que es genera a cada fase serveix de punt de partida per a la següent.

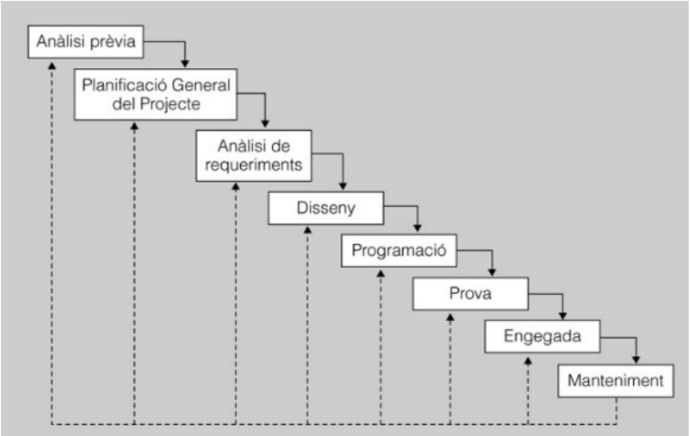

Figura 5: Cicle de vida en cascada

- **Avantatges del cicle en cascada**
  - És el més utilitzat tradicionalment en enginyeria del programari.
  - És metòdic i no deixa res a l'atzar.
  - Proporciona una plantilla en la qual es troben mètodes per a l'anàlisi, disseny, codificació, proves i manteniment.
  - Permeten definir punts de control intermedis per tal de comprovar l'avanç del projecte.
- Inconvenients del cicle en cascada
  - Els projectes reals rarament segueixen al peu de la lletra el model seqüencial que es proposa. Tot i que el model pot incloure interacció, ho fa indirectament. A resultes d'això, els canvis poden crear confusió quan comença l'equip de treball.
  - És complicat que el client expose explícitament tots els requisits. En canvi, el model ho requereix i té dificultats a l'hora d'incloure els nous plantejaments que faça el client.
  - El client no podrà veure els resultats fins que el projecte estiga molt avançat.

4.2.2. Models de desenvolupament evolutiu

En general, es tracta de models on el programari va evolucionant i rebent ajustos i millores contínues per part del client.

Veiem els diferents models que segueixen aquest principi:

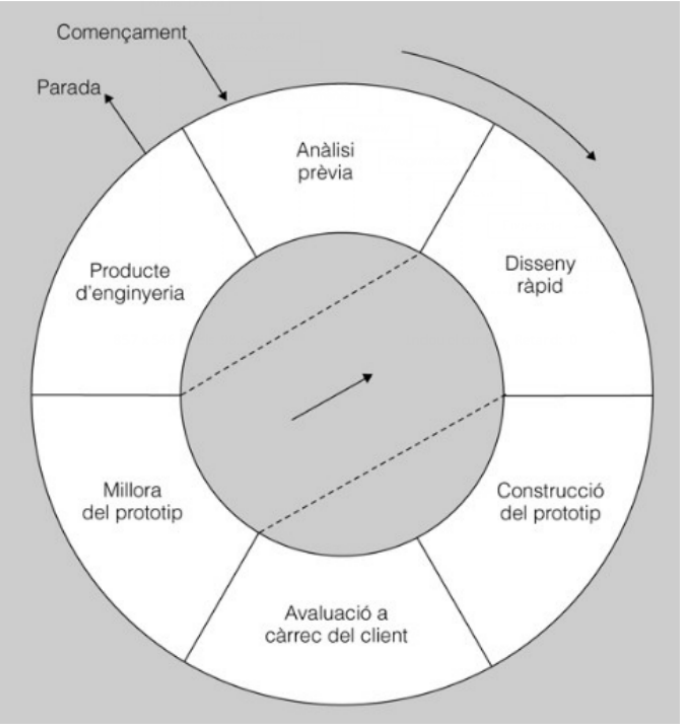

Figura 6: Model de construcció de prototips

Model de construcció de prototips En aquest model, es crea un prototip sobre el que el client pot validar els requeriments. Per tal que el model siga efectiu, cal que el client i el desenvolupador es posen d'acord amb el fet que el prototip és un mitjà per obtenir el producte final, i no és el producte final en si mateix.

Sovint s'utilitzen llenguatges o plataformes de prototipat, per fer aquest de forma ràpida, pel que una vegada validat el què es vol, cal començar el procés amb la plataforma final. Es tracta doncs d'una eina més per a la validació de requeriments que de desenvolupament en sí.

- **Avantatges de la construcció de prototips**
  - Els clients veuen resultats a mitjan procés i poden expressar més fàcilment el que volen.
  - Els desenvolupadors comencen a construir immediatament.
- **Inconvenients de la construcció de prototips**
  - El client veu el que sembla ser una versió operativa del programari i el voldran fer servir de seguida i no entén que cal construir novament el programari. El prototip no té totes les funcionalitats actives, sinó que només sembla que les tinga.
  - Com que el prototip s'ha de descartar i s'ha de desenvolupar ràpidament, l'equip pot

prendre la decisió de fer-lo amb un sistema operatiu o eines que poden no ser adients per al producte final.

Model iteratiu o incremental El model incremental combina elements del model en cascada, aplicat de forma repetida, amb la filosofia interactiva de la construcció de prototips. La idea és que el prototip no siga d'ún sol ús, sinó que aquest sí que siga el que posteriorment serà el producte final, que anirà *incrementant* la seua funcionalitat fins convertir-se en l'aplicació final.

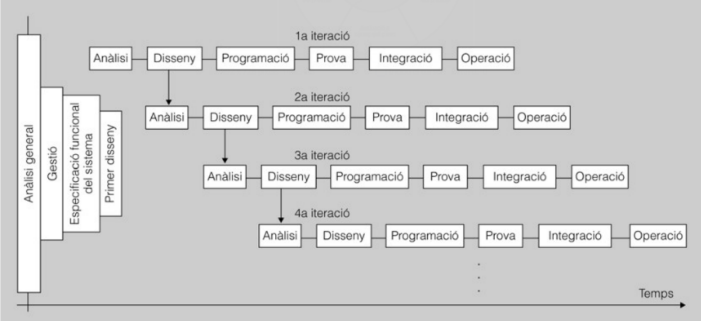

Figura 7: Model iteratiu o incremental

- **Avantatges del model iteratiu**
  - L'usuari pot veure el producte des de l'inici i pot guiar millor l'equip de treball cap al que espera.
  - La tasca d'anàlisi i disseny queda repartida, i no hi ha tanta monotonia en les fases del projecte.
- **Inconvenients del model iteratiu**
  - La interacció amb els usuaris pot alentir la marxa del projecte.
  - No és convenient per a projectes molt grans, ja que el temps per fer-lo es pot incrementar exponencialment, per la gran quantitat de funcionalitats que s'haurien de provar.
  - Una part de l'equip de treball queda inactiu durant els períodes de prova del programari per part dels clients.

Model en espiral Es tracta d'un model molt semblant al model iteratiu. Va ser proposat originalment per Boehm, i barreja la naturalesa iterativa de la construcció de prototips amb les fases sistemàtiques i controlades del model en cascada.

Figura 8: Model en espiral

Les principals diferències amb el model iteratiu són que es tracta d'un procés continu durant tota la vida del programari, fins que en una fase d'anàlisi de riscos es decideix no continuar, mentre que en el model iteratiu es coneixen prèviament quantes iteracions es faran.

En el model en espiral, en una iteració, es poden canviar les fases estàndards per unes altres a discreció de l'equip de treball. Per exemple, es podria fer una iteració amb el model de prototips.

- **Avantatges del model en espiral**
  - És un model adaptable, en que cada iteració es pot planificar amb el model més adient.
  - No cal planificar-ho tot al començament. Es pot anar adaptant segons el ritme de treball i les expectatives del client.
  - Quan s'inicia un canvi, ens situem al punt adequat de l'espiral i se segueix a partir d'ací.
  - Incorpora objectius de qualitat (que s'afegeixen en l'anàlisi de riscos).
- **Inconvenients del model en espiral**
  - El procés és lent i, per tant, poc adient per a projectes molt grans que estiguen ben definits.
  - El cost sol ser molt elevat, per la regeneració continua del codi i les etapes de planificació. Es requereix un equip experimentat en la identificació de riscos.

4.2.3. Models de desenvolupament àgil

Les metodologies de desenvolupament[3] àgil són un conjunt de metodologies modernes basades en el desenvolupament iteratiu, on els requeriments i les solucions evolucionen a través de col·laboracions entre diferents equips organitzats. Es tracta de metodologies no tan rígides com les tradicionals, i centrades en les persones, on el principal mecanisme de control és la retroalimentació, en lloc de la planificació. Aquesta retroalimentació es realitza a través de proves periòdiques i versions freqüents del programari.

Scrum La metodologia SCRUM[4] representa un marc de treball per a la gestió de projectes molt estés en el món del desenvolupament. El seu objectiu és desenvolupar un producte en un període determinat al que treballen diferents persones conjuntament per arribar a un objectiu comú.

Un dels principals *hàndicaps* en el desenvolupament de programari sol ser sovint la manca d'enteniment entre el què l'usuari realment vol i el que li s'està oferint, el que fa que els requeriments varien contínuament i provoquen canvis en tot un cicle de desenvolupament, amb els costos que això implica. Si el client no és capaç d'expressar el que realment desitja, pot que estiguem mesos desenvolupant una aplicació que no li va a ser útil. Açò perjudica aquells models que tenen una estructura més estricta.

> **?**Caminar sobre el agua y desarrollar software en base a una especificación es fácil, si ambos elementos están congelados.**?** (Edward V. Berard)

Scrum té en compte aquest canvi constant d'opinió entre les opinions sobre el què volen o necessiten els clients. Els imprevistos no han de suposar una gran dificultat per al desenvolupament dels productes. *Scrum* parteix de la base que el problema no està completament entès i definit, i es centra en maximitzar la capacitat de l'equip per entregar ràpidament dins dels terminis establerts i respondre a les necessitats d'última hora.

El funcionament d'aquest marc de treball és el següent:

1. En primer lloc, es defineixen els objectius del projecte, separant-los per tasques a realitzar, i assignant-los un temps necessari per realitzar aquestes tasques.
2. A partir de les tasques definides, el *Product Owner* (generalment el client), les ordena per importància i preferències, creant una llista de tasques ordenada, anomenada *Product Backlog*. Aquesta llista, representaria el conjunt de requeriments a alt nivell prioritzats, i que defineixen la feina en un període determinat.

[3] https://es.wikipedia.org/wiki/Metodolog%C3%ADa_de_desarrollo_de_software
[4] https://ca.wikipedia.org/wiki/Scrum

3. A partir del *Backlog*, l'equip fa una reunió per planificar les tasques, en forma de *sprints*. Aquests *sprints* són períodes de temps d'una durada definida (generalment entre una i quatre setmanes), on l'equip realitza les tasques assignades al *Backlog*. Aquest procés s'anomena *Sprint Planning*.
4. Cada *sprint* s'inicia amb una reunió de planificació d'aquest, on es defineix un *Sprint Backlog*, al que s'identifica el treball a realitzar durant l'sprint i el temps necessari per entregar el producte al final del període. Al final de cada sprint, l'equip de desenvolupament haurà de presentar el què s'ha avançat sobre el producte, que ha de ser potencialment lliureables al client.
5. Generalment, en començar cada jornada, es realitza un *Daily scrum* o *scrum diari*, d'una durada fixa (uns 15 min), i a una hora determinada, on cada membre de l'equip contesta tres preguntes: *Que has fet des d'ahir?*, *Què faràs hui?* i *Quins problemes t'impedeixen arribar als teus objectius del Sprint?*.
6. Després de cada sprint, es realitza una *Retrospectiva* d'aquest, on tot l'equip opina sobre com ha funcionat l'equip durant l'*sprint*, amb l'objectiu d'identificar els aspectes positius i els que s'han de millorar per optimitzar el rendiment de l'equip.

Durant cada l'sprint, l'única part del *Backlog* que pot canviar és l'equip de desenvolupament, si es veu que aquest no pot completar les tasques en el període establert. Aquest sistema, permet la creació d'equipa autoorganitzats impulsant la co-localització de tots els membres de l'equip, i la comunicació verbal entre tots els membres i disciplines involucrades en el projecte. El fet de no modificar els objectius en l'sprint llevat de canvis imprescindibles permet augmentar la concentració i la productivitat de l'equip de desenvolupament.

Els principals beneficis d'aquesta metodologia són:

- És flexible als canvis, ja que aporta gran capacitat de reacció en cas que es produïsquen canvis als requeriments generats per les necessitats del client o l'evolució del mercat.
- El client pot començar a utilitzar les característiques més importants del projecte abans que aquest estiga acabat per complet,
- El programari té major qualitat, ja que tenim la necessitat d'obtenir una versió funcional després de cada iteració.
- Millora la productivitat, ja que s'elimina gran part de burocràcia,
- Genera millores en la predicció del temps, ja que es coneix la velocitat mitjana de l'equip per sprint,
- Es redueixen els riscos, ja que coneixem com treballa l'equip, i les funcionalitats de més valor es realitzen en primer lloc.

4.3. Equips de treball. Funcions i responsabilitats dels membres

Durant el desenvolupament de programari, com hem vist, hi ha tasques diverses i molt diferents, pel que anem a tindre rols específics per a cadascuna d'elles. No es tracta de rols rígids, i és habitual que participen en diverses etapes del procés de desenvolupament.

Equips de treball tradicionals

Els principals rols en les metodologies tradicionals són:

- **Analista de sistemes**: El seu paper és el de realitzar un estudi del sistema per dirigir el projecte en una direcció que garantisca les expectatives del client, i determina el comportament del programari. Com el seu nom avança, el seu paper es porta a terme durant l'etapa d'anàlisi.
- **Dissenyador del programari**: És una evolució de l'analista, i realitza el disseny de la solució que cal desenvolupar. El seu paper s'ubica en la fase de disseny.
- **Analista programador**: O comunament anomenat *desenvolupador*, que té una visió més ampla de la programació i aporta una visió general del projecte, més detallada, dissenyant una solució més amigable per a la codificació i participant activament en ella. El seu paper es porta a terme en les fases de disseny i codificació.
- **Programador**: S'encarrega exclusivament de crear el resultat, escrivint el codi font a partir dels dissenys fets pel dissenyador i l'analista programador.
- **Arquitecte de programari**: És qui cohesiona el procés de desenvolupament. Coneix i investiga els frameworks i les tecnologies, tot revisant que tot el procediment es porte a terme de la millor forma i amb els recursos més apropiats. Participa en totes les fases del cicle de vida.

Organització d'equips en Scrum

A la metodologia Scrum, de la que hem parlat anteriorment, podem diferenciar tres rols principals:

- **Product Owner**: Representa *la veu* del client. És qui s'assegura que el resultat de l'equip s'adequa des de la perspectiva del negoci. És qui escriu les històries d'usuari, les prioritza i les col·loca al *Product Backlog*. Es tracta d'un rol més comercial, i que no interfereix en aspectes tècnics amb l'equip de desenvolupament.
- **Equip de desenvolupament**: Té la responsabilitat de lliurar les diferents parts del producte en cada sprint, així com el producte final. Solen ser treballs menuts, entre 3 i 9 persones, amb capacitats transversals d'anàlisi, disseny, programació, proves i documentació.
- **Scrum Master**: És qui organitza l'equip durant els sprints i s'assegura que el procés Scrum es porta a terme correctament. No es tracta d'un líder de l'equip, ja que aquest és autoorganitzat, sinó que actua com a protecció entre l'equip i qualsevol influència que el distraga.