Introducció a l'enginyeria del programari

Jordi Pradel Miquel
Jose Raya Martos

PID_00171144

Universitat Oberta de Catalunya
www.uoc.edu

Cap part d'aquesta publicació, incloent-hi el disseny general i la coberta, no pot ser copiada, reproduïda, emmagatzemada o transmesa de cap manera ni per cap mitjà, tant si és elèctric com químic, mecànic, òptic, de gravació, de fotocòpia o per altres mètodes, sense l'autorització prèvia per escrit dels titulars del copyright.

Índex

Introducció 5

Objectius 7

1. Què és l'enginyeria del programari? 9
1.1. Programari i maquinari 9
1.2. El desenvolupament de programari 9
1.3. L'àmbit de l'enginyeria del programari 11
1.4. Què és l'enginyeria del programari? 13
1.5. Història de l'enginyeria del programari 14
1.6. L'enginyeria del programari comparada amb les altres enginyeries 16

2. Organització de l'enginyeria del programari 19
2.1. Organització del desenvolupament, operació i manteniment 19
2.2. Organització dels projectes de desenvolupament 20
2.3. Activitats de l'enginyeria del programari 21
2.3.1. Gestió del projecte 21
2.3.2. Identificació i gestió dels requisits 23
2.3.3. Modelització 24
2.3.4. Construcció i proves 25
2.3.5. Qualitat 25
2.3.6. Manteniment i reenginyeria 26
2.3.7. Activitats des del punt de vista del cicle de vida del projecte 26
2.4. Rols en l'enginyeria de programari 27
2.4.1. El cap de projectes 28
2.4.2. Els experts del domini 28
2.4.3. L'analista funcional 28
2.4.4. L'arquitecte 29
2.4.5. L'analista orgànic o analista tècnic 29
2.4.6. Els programadors 29
2.4.7. L'expert de qualitat (provador) 30
2.4.8. L'encarregat del desplegament 30
2.4.9. El responsable de producte 30
2.4.10. Altres rols 31

3. Mètodes de desenvolupament de programari 32
3.1. Història dels mètodes de desenvolupament 32
3.2. Classificació dels mètodes de desenvolupament 34
3.2.1. Cicle de vida clàssic o en cascada 35

3.2.2. Cicle de vida iteratiu i incremental 37
3.2.3. Desenvolupament *lean* i àgil 39
3.3. Exemples de mètodes de desenvolupament 42
3.3.1. Mètrica 3 43
3.3.2. Procés unificat 46
3.3.3. Scrum 51

4. Tècniques i eines de l'enginyeria del programari.................... 54
4.1. Tècniques basades en la reutilització 54
4.1.1. Desenvolupament orientat a objectes 56
4.1.2. Bastiments 58
4.1.3. Components 58
4.1.4. Desenvolupament orientat a serveis 59
4.1.5. Patrons 60
4.1.6. Línies de producte 62
4.2. Tècniques basades en l'abstracció 63
4.2.1. Arquitectura dirigida per models 64
4.2.2. Llenguatges específics del domini 65
4.3. Eines de suport a l'enginyeria del programari (CASE) 65

5. Estàndards de l'enginyeria del programari.............................. 69
5.1. Llenguatge unificat de modelització (UML) 69
5.2. *Software engineering body of knowledge* (SWEBOK) 70
5.3. *Capability maturity model integration* (CMMI) 71
5.4. *Project management body of knowledge* (PMBOK) 72

Resum.. 73

Activitats... 75

Exercicis d'autoavaluació.. 77

Solucionari.. 78

Glossari... 82

Bibliografia... 84

Introducció

Aquest mòdul ha de servir per a introduir els futurs enginyers de programari a la seva disciplina. Per això, el que volem és donar una visió general de les diferents parts que formen l'enginyeria del programari, situar-les en context i relacionar-les les unes amb les altres.

Més endavant, quan s'estudiïn els detalls de cadascuna de les parts (en aquests mateixos materials o en futures assignatures, podrem tornar enrere a aquest mòdul i veure on encaixa allò que s'està estudiant dins el context de l'enginyeria del programari.

Hem tingut en compte una visió bastant àmplia de l'àmbit de l'enginyeria del programari, no solament des del punt de vista estrictament tècnic sinó també tenint en compte els aspectes organitzatius, ja que és habitual que s'esperi de l'enginyer de programari que sàpiga organitzar el desenvolupament i manteniment del programari.

Començarem estudiant l'enginyeria del programari com a disciplina de l'enginyeria. És important que l'enginyer de programari sàpiga quines són les particularitats de la seva enginyeria i també quines són les similituds amb les altres enginyeries i que sigui capaç d'establir paral·lelismes i veure les diferències. També veurem com es va arribar a la conclusió que, tot i les diferències, l'enginyeria era l'enfocament més adequat per al desenvolupament de programari.

Des del punt de vista organitzatiu, veurem quines són les activitats que s'han de dur a terme per a desenvolupar un producte de programari i de quina manera les podem organitzar en funció de les característiques del producte per desenvolupar.

En concret, veurem tres maneres diferents d'organitzar un projecte de desenvolupament:

1) seguint el cicle de vida en cascada (Mètrica 3),

2) seguint el cicle de vida iteratiu i incremental (Open UP) i

3) seguint els principis àgils (Scrum).

Des del punt de vista tècnic, veurem (sense estudiar-ho en detall) algunes de les eines i tècniques que es fan servir avui dia dins de la professió: l'orientació a objectes, els bastiments, el desenvolupament basat en components, el desenvolupament orientat a serveis, els patrons, les línies de producte, l'arquitectura dirigida per models i els llenguatges específics del domini.

Finalment, un enginyer de programari ha de conèixer els estàndards establerts en la seva professió. Per això veurem alguns estàndards relacionats amb els continguts del mòdul: UML, SWEBOK, CMMI i PMBOK.

Objectius

Els objectius que l'estudiant ha d'haver assolit una vegada treballats els continguts d'aquest mòdul són:

1. Entendre què és l'enginyeria del programari i situar-la en context.

2. Entendre les peculiaritats de l'enginyeria del programari comparada amb altres enginyeries.

3. Saber identificar els diferents rols i activitats que participen en un projecte de desenvolupament de programari.

4. Conèixer alguns dels mètodes de desenvolupament més utilitzats.

5. Conèixer algunes de les tècniques pròpies de l'enginyeria del programari.

6. Conèixer els principals estàndards de l'enginyeria del programari.

1. Què és l'enginyeria del programari?

Abans de començar amb l'estudi de l'enginyeria del programari cal que ens posem en context i definim què és el programari, quines de les activitats relacionades amb el programari cobreix l'enginyeria del programari i com hem arribat, com a indústria, a la conclusió que l'enginyeria és l'enfocament adequat per a aquestes activitats.

1.1. Programari i maquinari

Anomenem *programari* tot allò intangible (no físic) que hi ha en un ordinador, incloent-hi el conjunt de programes informàtics que indiquen la seqüència d'instruccions que un ordinador ha d'executar durant el seu funcionament (també anomenat *codi*) i les altres dades que aquest ordinador manipula i emmagatzema.

Codi i programari

En general, tret que indiquem el contrari, en aquesta assignatura abusarem del llenguatge fent servir la paraula programari per a referir-nos al codi (excloent-ne, per tant, les dades).

Més formalment, l'IEEE defineix programari com: "El conjunt dels programes de computació, procediments, regles, documentació i dades associades que formen part de les operacions d'un sistema de còmput".

Referència bibliogràfica

IEEE Std (1993). *IEEE Software Engineering Standard: glossary of Software Engineering Terminology*. IEEE Computer Society Press.

Per oposició, anomenem *maquinari* el conjunt de components físics d'un ordinador. Aquest maquinari ofereix un seguit d'instruccions que l'ordinador és capaç d'executar quan executa un programa.

1.2. El desenvolupament de programari

Com acabem de veure, per a programar un ordinador cal escriure una seqüència d'instruccions entre les que l'ordinador ofereix. El programari té una forma executable, que és la llista d'instruccions en un format que l'ordinador és capaç d'entendre i executar directament, el codi màquina.

Però el codi màquina no és llegible per a les persones. A més, el joc d'instruccions possibles ofert pel maquinari d'un ordinador és relativament reduït i senzill, la qual cosa fa que sigui força complicat escriure programes complexos fent servir directament aquest joc d'instruccions. Per aquesta raó, per a programar els ordinadors es fan servir llenguatges de programació i biblioteques de programari, que permeten escriure les instruccions a un nivell d'abstracció més elevat, fent servir instruccions compostes, més complexes.

De la manera llegible per les persones en què s'escriu el programari en diem **codi font**.

Exemple de comparació de dos nombres

Suposem que el nostre computador no permet comparar dos nombres directament i que només permet restar-los i saltar a una instrucció concreta en funció de si el resultat era més gran, menor o igual a 0.

Si volem comparar dos nombres (*num1* i *num2*) entre si per saber si són iguals, el codi màquina ha de moure els nombres a un dels registres de la CPU (operació MOV), restar-los (operació CMP) i saltar a l'etiqueta que toqui (operació JG i JMP). Per tant, cada vegada que el programador vulgui comparar *a* i *b* haurà d'escriure dues instruccions: la resta i la comparació.

Els llenguatges de programació, en canvi, permeten escriure una única instrucció *if*(*num1>num2*) i fer la traducció a la sèrie d'instruccions de codi màquina de manera automàtica. D'aquesta manera, el codi font és més fàcil d'escriure i de llegir.

En llenguatge assemblador x86:

```
 mov ax, num1
 mov bx, num2
 cmp ax, bx
 jg num2MesGran
 ;num1>num2
 jmp final
num2MesGran:
 ;num2>=num1
final:
```

En llenguatge C (i derivats):

```
if (num1>num2) {
 //num1 > num2
} else {
 //num2 >= num1
}
```

Típicament, el programari es desenvolupa amb l'objectiu de cobrir les necessitats d'un client o organització concrets, de satisfer les necessitats d'un determinat grup d'usuaris (i vendre'ls o donar-los el programari perquè el facin servir) o per a ús personal.

El desenvolupament de programari és l'acte de produir o crear programari.

Tot i que el desenvolupament de programari inclou la programació (la creació del codi font), fem servir el terme *desenvolupament de programari* d'una manera més àmplia per a referir-nos al conjunt d'activitats que ens porten des d'una determinada idea sobre el que volem fins al resultat final del programari.

Entre aquestes activitats podem trobar exemples com ara el recull, estudi i documentació de les necessitats dels usuaris, el manteniment del programari un cop es comença a fer servir, la coordinació del treball en equip de les diferents persones que intervenen en el desenvolupament, la redacció de manuals i ajudes d'ús per als usuaris, etc.

Com ha passat en altres àrees, quan la qualitat obtinguda i el cost del desenvolupament són importants, les organitzacions i empreses que desenvolupen programari han convertit aquestes activitats en una enginyeria.

Tot i que encara hi ha força debat sobre si el desenvolupament de programari és un art, una artesania o una disciplina d'enginyeria, el que és cert és que desenvolupar programari de qualitat amb el mínim cost possible ha resultat ser una activitat, en general, força complexa. En aquests materials estudiarem el desenvolupament de programari com a enginyeria i no tindrem en compte els altres enfocaments possibles.

1.3. L'àmbit de l'enginyeria del programari

Els primers computadors electrònics i els primers programes informàtics estaven orientats a la realització de càlculs matemàtics (per això tenim el terme *computadora*), però avui dia podem trobar programari pràcticament a tot arreu, des dels sistemes d'informació de qualsevol organització fins un rellotge de polsera, una motocicleta o les xarxes socials a Internet.

Les enormes diferències entre els diferents tipus de programari fan que la manera de desenvolupar uns i altres sigui totalment diferent. Així doncs, per exemple, una xarxa social a Internet pot actualitzar l'aplicació que fan servir els seus usuaris amb relativa facilitat (només ha d'actualitzar els seus servidors), mentre que actualitzar el programari que controla la centraleta electrònica de tots els cotxes d'un determinat model pot tenir un cost enorme per al fabricant.

Una de les primeres tasques, per tant, de l'enginyer de programari, és situar l'àmbit o àrea on s'aplicarà el programari per desenvolupar. En aquest sentit, podem fer una classificació de les àrees potencials d'aplicació de l'enginyeria del programari basada en la que fa Roger Pressman (2005):

- **Programari de sistemes**. Són programes escrits per a donar servei a altres programes, com ara els sistemes operatius o els compiladors. Aquest tipus de programes acostumen a interactuar directament amb el maquinari, de manera que els seus usuaris no són els usuaris finals que fan servir l'ordinador sinó altres programadors.

- **Programari d'aplicació**. Són programes independents que resolen una necessitat específica, normalment d'una organització, com ara el programari de gestió de vendes d'una organització concreta. Poden ser desenvolupats a mida (per a un únic client) o bé com a programari de propòsit ge-

neral (s'intenten cobrir les necessitats de diversos clients i és habitual que aquests utilitzin només un subconjunt de la funcionalitat total).

- **Programari científic i d'enginyeria**. Molt enfocats al càlcul i a la simulació, es caracteritzen per la utilització d'algorismes i models matemàtics complexos.

- **Programari encastat**. És el programari que forma part d'un aparell, des del control d'un forn fins a l'ordinador de bord d'un automòbil. Es caracteritza per les limitacions quant a recursos computacionals i per estar molt adaptat al producte concret que controla.

- **Programari de línies de productes**. És programari dissenyat per a proporcionar una capacitat específica però orientat a una gran varietat de clients. Pot estar enfocat a un mercat molt limitat (com ara la gestió d'inventaris) o molt ampli (com, per exemple, un full de càlcul).

- **Aplicacions web**. Les aplicacions web, independentment que siguin un paquet o a mida, tenen una sèrie de característiques que les fan diferents de la resta del programari. Es caracteritzen per unificar fonts de dades i serveis diversos en entorns altament distribuïts.

- **Programari d'intel·ligència artificial**. Aquests programes fan servir tècniques, eines i algorismes molt diferents de la resta de sistemes i, per tant, tenen una problemàtica pròpia. Pertanyen a aquesta categoria els sistemes experts, les xarxes neuronals i el programari de reconeixement de la parla.

Aquestes categories no són necessàriament excloents, de manera que ens podem trobar amb un programari d'aplicació desenvolupat com una aplicació web o un programari encastat desenvolupat com a línia de productes. El que sí que és cert és que cada categoria de les esmentades té la seva problemàtica específica.

Com que no podem estudiar totes aquestes problemàtiques, en aquests materials ens centrarem en un tipus concret de programari: el programari d'aplicació desenvolupat a mida, concretament, el programari per a sistemes d'informació.

> Un sistema d'informació és qualsevol combinació de tecnologia de la informació i activitats humanes que utilitzen aquesta tecnologia per a donar suport a l'operació, gestió o presa de decisions.

Sistema d'informació i sistema informàtic

Cal no confondre un sistema d'informació amb un sistema informàtic. Un sistema d'informació pot estar format per cap, un o més sistemes informàtics i també per persones i altres suports d'informació.

Per exemple, el sistema d'informació de gestió de la logística d'una empresa pot estar format per un sistema informàtic de gestió de comandes, un albarà en paper i diverses persones. D'altra banda, aquest albarà en paper es podria substituir per un altre sistema informàtic o pel mateix sistema informàtic de gestió de comandes.

El programari per a sistemes d'informació és un tipus de programari que gestiona una certa informació mitjançant un sistema gestor de bases de dades i dóna suport a una sèrie d'activitats humanes dins del context d'un sistema d'informació.

1.4. Què és l'enginyeria del programari?

L'IEEE defineix l'enginyeria com "l'aplicació d'un enfocament sistemàtic, disciplinat i quantificable a les estructures, màquines, productes, sistemes o processos per a obtenir un resultat esperat" i, més concretament, l'enginyeria del programari com "(1) L'aplicació d'un enfocament sistemàtic, disciplinat i quantificable al desenvolupament, operació i manteniment del programari; és a dir, l'aplicació de l'enginyeria al programari. (2) L'estudi d'enfocaments com en (1)".

Referència bibliogràfica

IEEE Std (1993). *IEEE Software Engineering Standard: glossary of Software Engineering Terminology*. IEEE Computer Society Press.

El **desenvolupament** és, com hem dit anteriorment, el procés que porta a la producció o creació del producte de programari; **l'operació** consisteix a executar el producte de programari dins del seu entorn d'execució per tal de dur a terme la seva funció; finalment, el **manteniment** comprèn la modificació posterior del producte de programari per tal de corregir-ne els errors o bé adaptar-lo a noves necessitats.

Per tant, quan parlem d'enginyeria del programari no solament estem parlant d'una manera de desenvolupar programari sinó que hem de tenir en compte la vida posterior del producte un cop creat. L'enginyeria del programari consisteix a dur a terme totes aquestes activitats de manera que puguem mesurar, quantificar i analitzar els diferents processos relacionats amb la vida del producte de programari.

Aquest enfocament ens permet extreure conclusions aplicables a futures activitats relacionades amb el producte de programari i respondre a preguntes com ara quins recursos són necessaris per a desenvolupar un nou producte, quant temps serà necessari per a afegir una nova funcionalitat a un producte ja existent o quins riscos podem trobar.

Tot això és, doncs, molt difícil d'aconseguir sense un enfocament sistemàtic i quantificable i apropa l'enginyeria del programari a les altres enginyeries, mentre que l'allunya de la creació artística.

1.5. Història de l'enginyeria del programari

Inicialment, el programari era un producte gratuït que s'incloïa en comprar maquinari i era desenvolupat, principalment, per les companyies fabricants del maquinari. Algunes poques companyies desenvolupaven programari a mida però no hi havia el concepte de programari empaquetat, com a producte.

El desenvolupament de programari no es gestionava segons una planificació i era pràcticament impossible predir-ne els costos i el temps de desenvolupament. Però ja s'aplicaven algunes tècniques de reutilització, com la programació modular.

El terme *enginyeria del programari* es va començar a fer servir cap al final dels cinquanta, però el punt d'inflexió que el va convertir en un terme usat globalment va ser una conferència del comitè científic de l'OTAN, feta l'octubre de 1968, que va definir formalment el terme *enginyeria del programari*.

La conferència de l'OTAN es va fer perquè el programari estava adquirint cada vegada més importància econòmica i social. Tanmateix, en la mateixa conferència es va detectar que el desenvolupament de programari estava molt lluny d'assolir els nivells de qualitat, productivitat i cost previstos, cosa que es va anomenar la *crisi del programari*. La crisi consistia en la dificultat d'escriure programari correcte, entenedor i verificable, que causava que els projectes tinguessin costos molt més elevats del previst, no s'acabessin en els terminis esperats, tinguessin una qualitat baixa, no complissin els requisits, etc.

La primera conferència sobre enginyeria del programari es va organitzar a la ciutat de Garmish, Alemanya, entre els dies 7 i 11 d'octubre de 1968, i va ser esponsoritzada per l'OTAN. Les actes de la conferència es van publicar el gener del 1969 i es poden trobar en l'enllaç següent: http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF

Així, per exemple, l'informe Chaos, elaborat per Standish Group el 1995, feia un estudi sobre el desenvolupament de programari als Estats Units i detectava que el 90% dels projectes estudiats no complien els objectius de temps, cost o qualitat. Algunes xifres alarmants d'aquest mateix informe mostraven que el 31% dels projectes de programari eren cancel·lats abans de completar-se i que només el 16% acabaven en el temps, pressupost i abast planificats.

Informe Chaos

L'informe Chaos ha rebut crítiques respecte a la seva negativitat, ja que considera reeixits només als projectes en què es compleixen els tres requisits (temps, pressupost i abast).

Independentment del que considerem com a projecte reeixit, hi ha xifres que són indiscutiblement significatives com, per exemple, el fet que es cancel·lés gairebé un terç dels projectes.

Durant dècades, empreses i investigadors es van centrar a superar la crisi del programari i van desenvolupar noves eines, tecnologies i pràctiques, buscant infructuosament solucionar de manera definitiva tots els problemes existents; en el millor dels casos, es van anar introduint paulatinament millores incrementals.

El 1986, Fred Brooks va publicar un article (Brooks, 1987) titulat "No Silver Bullet" (ho podríem traduir com "No hi ha solucions màgiques"), en què argumentava que no hi havia una solució única per al problema. Cap desenvolupament en tecnologia o gestió, deia, introduirà millores ni tan sols d'un ordre de magnitud en productivitat, fiabilitat o simplicitat, almenys no durant una dècada. I també argumentava que no podem esperar millores de dos ordres de magnitud cada dos anys com les que es produïen en el maquinari.

"No Silver Bullet"

El títol de l'article de Fred Brooks, "No Silver Bullet", literalment, "No hi ha bales de plata", fa servir una metàfora en què la solució definitiva mencionada és una bala de plata per a matar el monstre dels problemes de l'enginyeria del programari, que és l'home llop de les llegendes.

La causa d'aquestes afirmacions, segons Brooks, era que les millores que s'estaven introduint podien pal·liar la complexitat accidental, però no la complexitat essencial, inherent al desenvolupament de programari. Així, les solucions ja aplicades aleshores –com ara els llenguatges de programació d'alt nivell o les eines integrades de desenvolupament de programari– i les solucions que es consideraven de futur –com la programació orientada a objectes, la intel·ligència artificial o el prototipatge– solucionen complexitats accidentals i, tot i que introdueixen millores, no ho arriben a fer en ordres de magnitud.

Els anys noranta, el naixement i creixement exponencial d'Internet van provocar un creixement molt ràpid de la demanda de programari especialitzat, sobretot, en la Web. Aquesta demanda creixent, i el fet que moltes organitzacions petites requerissin també desenvolupament de programari, va introduir la necessitat de solucions de programari més simples, ràpides de desenvolupar i barates. Així, tot i que els grans projectes de programari continuaven fent servir les metodologies desenvolupades anteriorment, en els projectes més petits s'aplicaven metodologies més lleugeres.

Al començament de la dècada de 2010, es continuen buscant i aplicant solucions per a millorar l'enginyeria del programari. L'informe Chaos de 2009 mostra que els projectes considerats 100% reeixits han passat del 16% el 1994 al 32% el 2009, mentre que els cancel·lats han baixat del 31% al 24%. Aquestes xifres, tot i haver millorat, es consideren encara força negatives.

La gran majoria d'enginyers estan d'acord amb Brooks que no hi ha solucions màgiques i busquen solucions incrementals que vagin millorant els resultats obtinguts en l'enginyeria del programari. Algunes d'aquestes solucions són les línies de producte, el desenvolupament guiat per models, els patrons o les metodologies àgils de desenvolupament. D'aquestes i altres solucions parlarem al llarg d'aquests materials.

1.6. L'enginyeria del programari comparada amb les altres enginyeries

Un dels avantatges d'aplicar l'enginyeria a les activitats relacionades amb el programari és que ens permet aprofitar el coneixement generat en altres disciplines de l'enginyeria per a millorar la manera en què gestionem aquestes activitats.

Així doncs, al llarg del temps, s'han anat aplicant diferents metàfores (amb més o menys èxit) per tal d'extrapolar el coneixement adquirit en els diferents àmbits de l'enginyeria al món del programari i els sistemes d'informació.

La metàfora de la construcció

Un cas molt comú és el de la construcció. L'arquitecte crea uns plànols que han d'estar enllestits abans de començar la construcció de l'edifici, els paletes són fàcilment intercambiables, ja que les instruccions que han de seguir són molt clares i un cop finalitzada la feina el client es pot fer càrrec del manteniment de l'edifici sense necessitat de contactar amb els paletes que el van construir.

En aplicar aquesta metàfora, però, sovint no es tenen en compte factors com ara que l'edifici es construeix al lloc on es fa servir, mentre que el programari es desenvolupa en un entorn diferent d'aquell on es fa servir; o que el programari de qualitat s'ha de poder utilitzar en entorns diferents; o que el cost de fer còpies del programari és negligible (gairebé nul) en comparació del cost de dissenyar-lo.

Un dels perills de les metàfores és que, sovint, s'apliquen d'acord amb un coneixement incomplet de l'àmbit de partida, la qual cosa portarà a errors a l'hora de traslladar el coneixement i les pràctiques d'una enginyeria a l'altra.

Construcció sense plànols

En l'exemple anterior, hem suposat que l'edifici està totalment dissenyat abans de començar la construcció i que, per tant, no cal modificar el disseny però, en la pràctica, moltes vegades els constructors es troben amb problemes no previstos que els obliguen a modificar el disseny original.

Per exemple, el temple de la Sagrada Família de Barcelona es va començar a construir el 1882, però Gaudí el va replantejar totalment el 1883. L'arquitecte va modificar la idea original a mesura que avançava la construcció, fins que va morir el 1926 sense deixar plànols ni directrius sobre com calia continuar l'obra, que es va continuar construint durant tot el segle XX i part del XXI. Aquesta és una història ben diferent del que tenim en ment quan parlem de com funciona la construcció.

Per tant, és molt important conèixer bé les característiques inherents al programari que el diferencien dels altres productes industrials per tal de poder aplicar amb èxit el coneixement generat a altres enginyeries. En podríem destacar les següents:

1) El programari és intangible. El programari és un producte intangible la producció del qual no consumeix cap matèria primera física: podríem dir que la matèria primera del programari és el coneixement. Com a conseqüència, la gestió dels processos relacionats amb el seu desenvolupament i manteniment és diferent de la de molts productes industrials.

2) El programari no es manufactura. Com passa amb qualsevol producte digital, un cop desenvolupat el programari, crear-ne còpies té un cost molt baix i les còpies creades són idèntiques a l'original. Així doncs, no hi ha un procés de manufactura de la còpia, i, per tant, el coneixement derivat de la manufactura de productes industrials no és traslladable al programari. En canvi, el coneixement relacionat amb el desenvolupament del producte sí que ho serà.

3) El programari no es desgasta. A diferència dels productes tangibles, el programari no es desgasta, la qual cosa fa que el seu manteniment sigui força diferent del manteniment d'un producte industrial tangible. Per exemple, si un programari falla, com que totes les còpies són idèntiques a l'original, totes tindran el mateix error; no hi ha peces de recanvi per canviar. Per aquest motiu, ens caldrà revisar tot el procés de desenvolupament del programari per tal de detectar en quin punt es va introduir l'error i corregir-lo.

4) El programari queda obsolet ràpidament. El ràpid canvi tecnològic fa que el programari quedi obsolet amb relativa rapidesa. Això incrementa la pressió per aconseguir desenvolupar-lo de manera ràpida i amb poc cost, ja que el seu cicle de vida és molt curt si el comparem amb altres productes industrials com ara els cotxes, els avions o les carreteres.

Un altre tret distintiu de l'enginyeria del programari és que és una indústria relativament nova. Com s'ha comentat, la primera menció reconeguda del terme *enginyeria del programari* va ser el 1968 i els primers ordinadors electrònics van aparèixer a la dècada dels quaranta. Per tant, l'experiència acumulada és relativament poca si la comparem amb altres enginyeries. Aquest fet es veu agreujat per la ràpida evolució de la tecnologia, fonamentalment en dos eixos:

- Per una banda, les noves possibilitats tecnològiques fan que canviï totalment la naturalesa dels productes que estem desenvolupant; per exemple, l'aparició de les interfícies gràfiques als anys vuitanta, la consolidació de l'accés a Internet als anys noranta o la popularització de l'accés a Internet des del mòbil a la primera dècada del segle XXI van donar lloc a nous tipus de productes i noves necessitats pràcticament inimaginables deu anys enrere.

- Per altra banda, l'augment de la potència de càlcul dels ordinadors també ha provocat canvis fonamentals en les eines que es fan servir per a desenvolupar programari, tant a escala de llenguatges i paradigmes de programació (estructurats, orientats a objectes, dinàmics, etc.) com de les eines mateixes (entorns de desenvolupament, eines CASE, etc.).

Eines CASE

Les eines CASE (*computer aided software engineering*) són les eines que ens ajuden a dur a terme les diferents activitats de l'enginyeria del programari com ara la creació de models, generació de documentació, etc.

Tot plegat fa que molt del coneixement acumulat al llarg dels anys hagi quedat obsolet a causa de les importants diferències entre el context en què aquest coneixement es va generar i el nou context en què s'ha d'aplicar.

Finalment, no podem deixar de mencionar un altre tret distintiu de l'enginyeria del programari, com és l'aparició del programari lliure. El programari lliure ha transformat enormement la manera de desenvolupar i mantenir qualsevol tipus de programari i ha promogut la creació d'estàndards *de facto*, la reutilització de programari i a la difusió del coneixement respecte a com ha estat desenvolupat. Aquest nivell de transparència i col·laboració no es troba, a hores d'ara, en cap altra enginyeria.

2. Organització de l'enginyeria del programari

Hem dit anteriorment que l'enginyeria del programari implica ser sistemàtics en el desenvolupament, operació i manteniment del programari, per a la qual cosa és necessari definir quin és el procés que cal seguir per a dur a terme aquestes tasques; aquesta és la finalitat dels mètodes de desenvolupament.

L'IEEE defineix un mètode dient que "descriu les característiques del procés o procediment disciplinat utilitzat en l'enginyeria d'un producte o en la prestació d'un servei".

Per tant, per a aplicar enginyeria a les activitats relacionades amb el programari ho hem de fer per mitjà de l'aplicació d'un mètode. Aquest mètode pot ser aplicable al desenvolupament, a l'operació o al manteniment del programari o bé a qualsevol combinació de tots tres.

2.1. Organització del desenvolupament, operació i manteniment

Hi ha diferències importants sobre com s'han d'organitzar les diferents activitats relatives al desenvolupament, operació i manteniment del programari.

Les activitats d'operació acostumen a ser contínues, mentre que el desenvolupament acostuma a ser temporal: el desenvolupament té un inici clar (el moment en què es planteja la possibilitat de desenvolupar el producte) i un final també força clar, sigui amb èxit o fracàs.

Una altra diferència important és que les activitats d'operació acostumen a ser repetitives, mentre que el desenvolupament proporciona un resultat únic (dos processos de desenvolupament donaran lloc a dos productes diferents).

Per això, tot i compartir certes similituds, l'organització del desenvolupament és molt diferent de l'organització de l'operació del programari.

Mentre que el desenvolupament s'acostuma a organitzar en forma de projectes, l'operació s'organitza d'acord amb serveis.

Lectures recomanades

Si voleu aprofundir en la gestió de projectes, podeu consultar la guia PMBOK publicada pel Project Management Institute (PMI), mentre que si us interessa la gestió de l'operació del programari podeu consultar l'estàndard ITIL.

El manteniment es pot organitzar de manera contínua, especialment en el cas del manteniment correctiu (aquell que es fa per a corregir errors al programari) o bé de manera temporal (en forma de projecte) com seria el cas del manteniment evolutiu (aquell que es fa per a modificar el producte d'acord amb les noves necessitats detectades amb posterioritat al seu desenvolupament).

A causa d'aquestes diferències, tot i haver-hi mètodes que defineixen processos per a totes tres activitats (desenvolupament, operació i manteniment), molts se centren en una única d'aquestes. Així, per exemple, hi ha molts mètodes de desenvolupament que no defineixen el procés que cal seguir per a operar el programari.

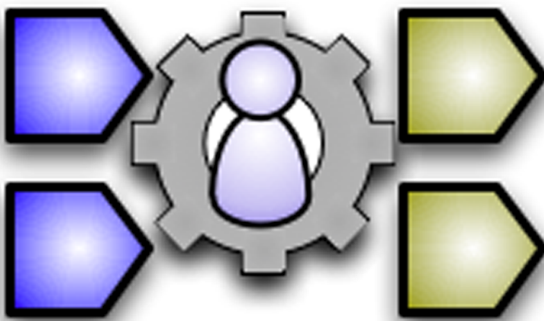
En endavant, ens centrarem en l'estudi dels projectes de desenvolupament de programari i de manteniment.

Un projecte de desenvolupament de programari té, com hem dit anteriorment, un inici i un final ben determinats. Aquest final pot estar determinat per l'èxit o fracàs del projecte o bé per altres circumstancies com ara la desaparició de la necessitat per cobrir.

2.2. Organització dels projectes de desenvolupament

Al llarg del temps s'han anat definint diferents mètodes per al desenvolupament de programari, molts basats en els mètodes aplicats en altres disciplines de l'enginyeria. Cada un d'aquests mètodes defineix un o més processos de desenvolupament. La descripció d'un procés inclou, típicament:

- Quines **tasques** i en quin ordre s'han de dur a terme.
- Quins **rols** han de tenir les diferents persones que participen en el desenvolupament, quina és la responsabilitat de cada rol i quines tasques ha de dur a terme.
- Quins **artefactes** (documents, programes, etc.) s'han de fer servir com a punt de partida per a cada tasca i quins s'han de generar com a resultat.



La descripció d'un procés indica els artefactes d'entrada d'una tasca, el rol responsable de dur-la a terme i els artefactes de sortida (que serviran d'entrada per a la tasca següent).

Els processos definits per un mètode es basen en criteris tan diversos com ara el seu abast (hi ha mètodes que només defineixen el procés de desenvolupament de programari, mentre que altres tenen en compte altres aspectes com, per exemple, el procés que portarà a una organització a decidir si desenvolupa el programari o no), el tipus d'organització al qual va dirigit (petites empreses, Administració pública, l'Exèrcit, etc.) o el tipus de producte que es vol generar (una aplicació de gestió de les vacances dels empleats d'una empresa, el portal d'Hisenda per a presentar la declaració de la renda, etc.).

2.3. Activitats de l'enginyeria del programari

Tal com hem vist, els mètodes defineixen quines tasques es duran a terme en cada procés. Tot i que cada mètode té les seves particularitats, és cert que hi ha un seguit de tasques que s'han de dur a terme a tot projecte de desenvolupament amb independència de com s'organitzi.

Al llarg d'aquest mòdul farem servir el terme *activitat* per a referir-nos a un conjunt de tasques relacionades entre si. Distingim activitats de tasques en el sentit que un procés defineix tasques concretes (generar el model conceptual, crear el diagrama de Gantt amb la planificació del projecte) que poden diferir d'un mètode a un altre, mentre que l'activitat (modelització, planificació) seria la mateixa.

Cada mètode pot donar més o menys importància a cadascuna de les activitats que descrivim a continuació, però no les pot obviar si el volem considerar con un mètode complet de desenvolupament de programari.

2.3.1. Gestió del projecte

La gestió de projectes és una disciplina general, comuna per a tota mena de projectes de les tecnologies de la informació i comunicació (TIC) i d'altres àmbits, com la construcció, l'enginyeria i la gestió d'empreses, entre d'altres, i inclou tots els processos necessaris per a la direcció, la gestió i l'administració de qualsevol projecte, amb independència de quin producte concret s'estigui construint. L'objectiu principal és assegurar l'èxit del projecte.

En el cas dels projectes de desenvolupament de programari, el mètode general de gestió de projectes s'ha de complementar amb els mètodes, les tècniques i les eines pròpies dels projectes de desenvolupament de programari.

Hi ha qui considera que la gestió del projecte, en no ser una activitat específica del desenvolupament de programari, no forma part de l'àmbit de l'enginyeria del programari. En qualsevol cas, forma part del projecte de desenvolupament i, per tant, els mètodes de desenvolupament de programari l'han de tenir en compte.

A continuació s'indiquen algunes de les tasques relacionades amb la gestió del projecte:

- **Estudi de viabilitat**. Abans de començar pròpiament el projecte, caldrà detectar una necessitat en l'organització que l'encarrega i fer un estudi d'alternatives i costos per tal de decidir si el projecte de desenvolupament és o no la millor opció per a satisfer les necessitats esmentades. Per a fer-ho, caldrà anticipar el cost del projecte (quins recursos s'han d'invertir per al desenvolupament) i quant de temps serà necessari per a desenvolupar-lo.

- **Estimació**. Tal com hem dit abans, cal fer una estimació del cost del projecte i també del temps necessari per a dur-lo a terme i de la relació entre recursos i temps: com es modifica l'estimació de temps si afegim o traiem recursos? Aquesta relació no acostuma a ser lineal. L'exemple clàssic és que, mentre que una dona pot gestar una criatura en nou mesos, nou dones no la poden gestar en un mes.

- **Definir clarament els objectius del projecte, que en determinaran l'èxit o fracàs**. Per exemple, un projecte pot fracassar perquè, un cop iniciat, es descobreixi que és impossible complir els seus objectius amb els recursos o el temps disponible (i, per tant, calgui cancel·lar el projecte).

- **Formar l'equip de desenvolupament tenint en compte l'estimació de recursos feta inicialment**. Aquest equip pot estar format per persones amb dedicació completa al projecte o bé amb dedicació parcial. Com més va és més habitual que aquests equips tinguin, a més dels desenvolupadors, persones que representen els *stakeholders*.

Stakeholder

El terme anglès *stakeholder* fa referència a qualsevol persona o organització interessada, afectada o implicada en el funcionament del programari que es desenvolupa.

- **Establir fites** que ens permetin dur a terme les activitats de seguiment i control del projecte per tal de verificar-ne la bona marxa.

- **Identificar riscos que puguin posar en perill l'èxit del projecte**. No solament des del punt de vista tecnològic (haver d'utilitzar tecnologies noves i poc provades, etc.) sinó de tots (manca de suport per part de l'organització, problemes legals, etc.).

Un cop s'ha decidit iniciar el projecte i s'ha fet la planificació inicial, la gestió del projecte continua, ja que se n'ha d'observar el progrés i comparar-lo amb la previsió inicial per tal de validar les suposicions inicials i fer les correccions oportunes en cas d'estar equivocades.

La gestió del projecte està molt influenciada pel mètode de desenvolupament emprat, ja que aquest determinarà les tasques que s'han de dur a terme i també l'ordre en què s'han de dur a terme. El mètode de desenvolupament també indicarà quins artefactes es generen durant la planificació i com es fa el seguiment i control del projecte.

Nota

Pel que fa a la gestió de projectes, no la tractarem en aquesta assignatura sinó que en parlarem àmpliament en una assignatura posterior.

Finalment, podem dir que, en última instància, la gestió del projecte consisteix a equilibrar tres variables:

1) l'abast del projecte (què s'ha d'incloure i què no),

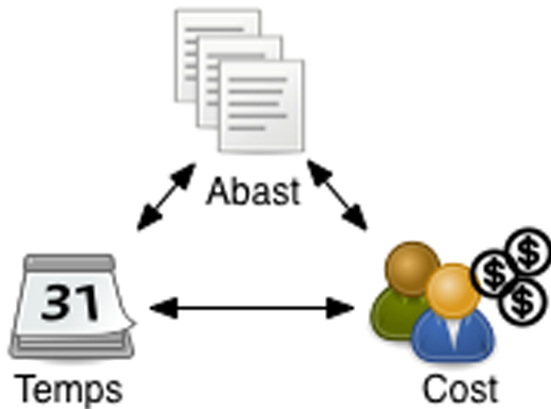
2) el temps (quan es finalitzarà l'execució del projecte), i

3) el cost (en recursos humans i recursos materials).

Altres variables

A més a més de l'abast, el temps i el cost, altres autors com Wysocki (2009) afegeix la qualitat com a variable i distingeix entre el cost (pressupost) i els recursos (principalment, les persones).

Les tres variables de la gestió de projectes: temps, abast i cost



Un canvi en qualsevol d'aquestes tres variables s'ha de compensar modificant, com a mínim, una de les altres dues (és similar al que en teoria de jocs s'anomena *joc de suma zero*).

Joc de suma zero

En teoria de jocs, un joc de suma zero descriu una situació en què el guany o la pèrdua en què incorre un dels participants es compensa exactament per les pèrdues o guanys de la resta de participants. Així doncs, si sumem totes les pèrdues i tots els guanys, el resultat final és 0. Per exemple, una partida de pòquer seria un joc de suma zero, ja que els diners que un jugador guanya els ha de perdre un altre jugador (i viceversa).

2.3.2. Identificació i gestió dels requisits

Aquesta activitat implica la comunicació i col·laboració amb els *stakeholders*, fonamentalment per tal de trobar quins són els requisits del producte que cal desenvolupar.

Vegeu també

Parlarem extensivament de la gestió de requisits al mòdul "Requisits".

> Segons la guia SWEBOK (2004) els requisits "expressen les necessitats i restriccions que afecten un producte de programari que contribueix a la solució d'un problema del món real".

Vegeu també

Podeu trobar més informació sobre SWEBOK en el subapartat 5.2 d'aquest mòdul didàctic.

Els requisits, per tant, ens serveixen per a delimitar quines de les possibles solucions al problema són adequades (les que compleixen els requisits) i quines no.

La identificació i gestió de requisits està molt relacionada amb l'abast del projecte, ja que són els requisits els que determinen aquest abast. Les principals problemàtiques que cal vèncer en aquesta tasca són les pròpies de qualsevol activitat de comunicació:

- **Diferències respecte a la informació amb què treballen les diferents parts**. Els *stakeholders* tenen informació sobre el producte que els desenvolupadors no tenen, mentre que els desenvolupadors tenen informació sobre la tecnologia que els *stakeholders* no tenen. Això pot condicionar la visió del problema que tenen uns i altres i pot afectar negativament la transferència d'informació.

- **Limitacions del canal utilitzat**. Qualsevol canal de comunicació imposa limitacions. Per exemple, si la comunicació és escrita, es perd el llenguatge no verbal, mentre que, si és verbal, es perd la possibilitat de revisió que dóna la documentació escrita.

- **Limitacions del llenguatge utilitzat**. El llenguatge natural és propens a l'ambigüitat, raó per la qual s'han desenvolupat els llenguatges formals d'especificació; aquests llenguatges, però, acostumen a ser menys expressius que el llenguatge natural. A més, només serveixen per a la comunicació si totes les parts els entenen correctament.

- **Dificultat de definir el millor sistema possible**. Un altre problema associat a aquesta activitat és aconseguir que els *stakeholders* comuniquin exactament els requisits del millor sistema possible, ja que és habitual que, en descriure el sistema que volen, estiguin condicionats pel coneixement que tenen sobre sistemes semblants o, senzillament, que no se'ls acudeixin algunes possibilitats.

Per a solucionar aquests problemes és molt habitual l'ús de tècniques basades en retroalimentació durant l'anàlisi de requisits (com ara el prototipatge, que consisteix a ensenyar una versió amb aspecte similar al producte final però que no implementa la funcionalitat real) quan això és possible.

Una altra tècnica molt habitual és l'ús de models, tot i que, en aquest cas, la seva utilitat està condicionada al fet que els usuaris/*stakeholders* entenguin la notació del model.

2.3.3. Modelització

Inclou la creació de models del sistema per desenvolupar: aquests models facilitaran la comprensió dels requisits i el disseny del sistema. En certa manera, aquesta tècnica és equivalent a la construcció de maquetes o models en altres

Vegeu també

Parlarem en detall de la modelització en el mòdul "Anàlisi UML".

disciplines de l'enginyeria, amb la diferència que, en ésser el programari un producte intangible, aquests models no acostumen a tenir una naturalesa física sinó que són intangibles com el programari mateix.

Actualment, el llenguatge més utilitzat per a la creació de models de programari és el llenguatge UML[1], que és un llenguatge publicat per l'Object Management Group (OMG) i que defineix tot un seguit de diagrames a partir dels quals podem elaborar els nostres models.

[(1)] *UML* són les sigles de *unified modeling language*.

Un dels motius de l'èxit del llenguatge UML és que no imposa cap mètode de desenvolupament i, per tant, ha estat adoptat per la majoria de mètodes de desenvolupament actuals. Un altre avantatge important és que només defineix la notació que s'ha de fer servir però no quins artefactes s'han de generar, de manera que un mateix tipus de diagrama es pot fer servir en diversos artefactes (model del domini, diagrama de classes del disseny, etc.).

2.3.4. Construcció i proves

La construcció inclou l'escriptura del codi font (implementació) i la realització de les proves necessàries per a garantir, en la mesura del possible, l'absència d'errors en els programes i l'adequació del producte als requisits.

Vegeu també

Les activitats de construcció i prova i les següents no les tractarem en aquesta assignatura; les veurem en assignatures posteriors dins l'itinerari d'Enginyeria del Programari.

Un cop creat el codi, caldrà crear els arxius executables i posar-los a disposició dels usuaris finals (el que s'anomena *desplegament*).

Com a part d'aquesta activitat també cal tenir en compte les tasques relacionades amb la gestió de la configuració (quins arxius formen part de cada versió del sistema), i la gestió dels canvis (com apliquem els canvis al codi font de manera que puguem generar noves versions del sistema de manera controlada).

2.3.5. Qualitat

La gestió de la qualitat és una activitat que inclou totes les etapes del desenvolupament. Com a mínim, caldrà definir els criteris d'acceptació del sistema: en quin moment decidirem que el projecte està finalitzat satisfactòriament.

Típicament, la gestió de la qualitat implica la recol·lecció de mètriques que ens ajudin a determinar si el programari compleix o no els criteris de qualitat marcats i també la documentació formal dels processos de desenvolupament i la verificació del seu compliment.

2.3.6. Manteniment i reenginyeria

Tal com hem dit anteriorment, un cop tancat el projecte de desenvolupament, caldrà engegar noves activitats d'operació i manteniment que quedaran fora de l'àmbit d'aquest. Mentre que l'operació no implica canvis al programari, el manteniment sí que ho fa.

El manteniment correctiu consisteix a corregir els errors detectats en el programari, i està molt relacionat amb la gestió de la configuració i del canvi, ja que, un cop detectat un error, caldrà corregir-lo sobre el codi font i generar una nova versió del programa, que s'haurà de posar a disposició dels usuaris finals.

El manteniment evolutiu, en canvi, és més semblant a un projecte de desenvolupament, ja que consisteix a afegir noves funcionalitats o adaptar-ne les ja existents per tal d'acomodar noves necessitats detectades després de la fi del desenvolupament inicial.

2.3.7. Activitats des del punt de vista del cicle de vida del projecte

El Project Management Institute fa una classificació diferent, ja que es fixa en la gestió del projecte i en quin punt del cicle de vida es du a terme cada tasca:

Project Management Institute

El Project Management Institute (PMI) és una associació professional de gestors de projectes que es dedica, entre altres activitats, a publicar estàndards de bones pràctiques en la gestió de projectes.

- **Tasques d'iniciació**. Són aquelles relatives a la presa de la decisió sobre si es començarà o no el projecte (o la nova fase del projecte). Per exemple, una organització es pot plantejar, com a alternativa a un desenvolupament, l'adquisició d'un producte ja desenvolupat. Com a resultat de les activitats d'iniciació, obtindrem la definició de l'abast del projecte (allò que es farà i allò que no es farà com a part del projecte), l'estimació de la durada (límit temporal) del projecte i del cost (recursos humans i materials que caldrà invertir per a dur a terme el projecte).

- **Tasques de planificació**. Ajuden a organitzar el treball, definint quines tasques caldrà dur a terme i en quin ordre. Per exemple, es pot decidir planificar el projecte en funció dels riscos (dur a terme primer les tasques que poden posar en perill la viabilitat del projecte), o bé per funcionalitats (desenvolupar una funcionalitat de manera completa abans de treballar en les altres) o per tipus d'activitats (primer fer totes les tasques d'un tipus, després passar a les del tipus següent, etc.).

- **Tasques d'execució**. Són aquelles destinades a completar la feina requerida per a desenvolupar el producte. Per exemple, escriure els programes.

- **Tasques de seguiment i control**. Són les destinades a observar l'execució del projecte per tal de detectar els possibles problemes i adoptar les acci-

ons correctives que calgui. També ens ajuden a validar la planificació (per exemple, validant que l'estimació inicial de cost i temps era correcta).

- **Tasques de tancament**. Un cop finalitzat el projecte, cal tancar-lo adequadament. Per exemple, caldrà fer l'acceptació del producte (que consisteix a determinar que, efectivament, el producte satisfà els objectius establerts), resoldre el contracte si s'ha contractat una organització externa per a fer el desenvolupament o bé traspassar el programari a l'equip d'operacions.

Tot i que hi ha una evident relació temporal entre els diferents tipus de tasques, aquestes no tenen lloc necessàriament de manera seqüencial. De fet, els diferents mètodes de desenvolupament aconsellaran maneres diferents de combinar-les segons quin sigui el seu cicle de vida.

Mentre que un mètode pot aconsellar, per exemple, no començar cap tasca d'execució fins que no s'ha completat en gran mesura la planificació, d'altres ens poden aconsellar que no dediquem massa esforç a la planificació fins que no hàgim executat una part del projecte i hàgim assolit un cert nivell de seguiment i control.

2.4. Rols en l'enginyeria de programari

Tal com hem dit anteriorment, tot mètode de desenvolupament ha de definir una sèrie de rols, i també quines són les seves responsabilitats (quines tasques ha de dur a terme cada rol). Aquesta definició de rols és molt important, ja que el desenvolupament de programari és una activitat en la qual intervé una gran varietat de persones, i és la interacció entre aquestes persones la que determinarà, finalment, l'èxit o fracàs del projecte.

Els rols que defineixi un mètode de desenvolupament dependran, en part, de l'abast del mètode (si només se centra en la creació del programari o si inclou també la part més organitzativa) i, en part, també dels principis i valors que es vulguin transmetre. Així doncs, l'organització en rols dels mètodes que segueixen el cicle de vida en cascada és molt diferent de la del mètodes àgils.

Lectura recomanada

Tom DeMarco; Timothy Lister (1999). *Peopleware: productive Projects and Teams* (2a. ed.). Dorset House Publishing Company.

A continuació estudiarem alguns dels rols típics que podem trobar (potser amb un nom diferent) a qualsevol mètode de desenvolupament.

2.4.1. El cap de projectes

El PMBOK defineix la gestió de projectes com "l'aplicació del coneixement, habilitats, eines i tècniques a les activitats del projecte per tal de complir els requisits del projecte". La gestió del projecte inclou, doncs, tasques d'iniciació, planificació, execució, seguiment, control i tancament.

PMBOK

El PMBOK és l'estàndard publicat pel PMI que recull el cos de coneixement àmpliament acceptat pels seus membres, és a dir, recull aquelles idees amb les quals la majoria dels seus membres està d'acord.

> El cap de projectes és la persona responsable d'aconseguir els objectius del projecte.

Vegeu també

Podeu trobar més informació sobre el PMBOK al subapartat 5.4.

Per tant, el cap de projectes és un rol no tècnic encarregat d'organitzar el projecte, coordinar la relació entre l'equip de desenvolupament i la resta de l'organització i vetllar pel compliment dels seus objectius tant en relació amb els costos com el valor generat.

2.4.2. Els experts del domini

Els experts del domini són les persones que coneixen el domini del sistema que s'està desenvolupant i, per tant, són els principals coneixedors dels requisits. El cas més habitual és que no siguin tècnics sinó que aportin el seu coneixement sobre altres àrees com ara vendes, finances, producció, etc.

Un cop més, els diferents mètodes de desenvolupament demandaran diferents nivells d'implicació per als experts del domini. Mentre que seguint el cicle de vida clàssic aquests només són necessaris durant les fases inicials, els mètodes àgils recomanen que els experts del domini formin part de l'equip de desenvolupament i, fins i tot, es trobin físicament a la mateixa sala que els desenvolupadors.

De vegades, però, no podrem tenir experts del domini per al desenvolupament i s'haurà de recórrer a altres tècniques per a l'obtenció dels requisits com ara les dinàmiques de grup, les entrevistes, etc.

Aquest rol també és conegut com a analista de negoci[2], especialment en entorns empresarials.

(2)En anglès, *business analyst.*

2.4.3. L'analista funcional

L'analista funcional és el responsable d'unificar les diferents visions del domini en un únic model que sigui clar, concís i consistent. Tot i que la seva tasca no és tecnològica, sí que és un perfil tècnic en el sentit que ha de conèixer les notacions i estàndards de l'enginyeria del programari per tal de generar un model del sistema que sigui adequat per a l'ús per part de la resta de tècnics.

Una diferència important respecte als experts del domini és que coneix les limitacions de la tecnologia i també les diferents maneres d'aplicar-la de manera constructiva a la resolució dels problemes del negoci. Així doncs, encara que potser no conegui els detalls de la implementació del sistema, sí que té una idea bastant clara de les possibilitats tecnològiques.

2.4.4. L'arquitecte

L'arquitecte és el responsable de definir les línies mestres del disseny del sistema. Entre altres responsabilitats, té la d'escollir la tecnologia adequada per a la implementació del projecte, per a la qual cosa ha de tenir en compte el tipus de producte, els coneixements dels membres de l'equip i altres requisits de l'organització.

L'arquitecte, doncs, és un rol tècnic amb un bon coneixement de les tecnologies d'implementació. A partir dels requisits recollits per l'analista, crearà un conjunt de documents d'arquitectura i disseny que la resta de desenvolupadors farà servir com a base per al seu treball.

En alguns mètodes (especialment en els incrementals), l'arquitecte també és responsable d'implementar l'arquitectura i de validar-ne la viabilitat i la idoneïtat.

2.4.5. L'analista orgànic o analista tècnic

L'analista orgànic s'encarrega del disseny detallat del sistema respectant l'arquitectura definida per l'arquitecte. El destinatari de la seva feina és el programador.

L'analista orgànic prendrà com a punt de partida un subconjunt dels requisits de l'analista i la definició de l'arquitectura i dissenyarà les parts del sistema que implementen els requisits esmentats fins a un nivell de detall suficient per a la implementació.

En molts mètodes de desenvolupament, especialment en els que tendeixen menys a l'especialització, no hi ha un rol diferenciat per a l'analista orgànic, sinó que els programadors, mencionats a continuació, fan també aquest rol.

2.4.6. Els programadors

Els programadors són els responsables d'escriure el codi font a partir del qual s'han de generar els programes. Per tant, són experts en la tecnologia d'implementació. Per a dur a terme la seva tasca, partiran dels dissenys detallats creats per l'analista orgànic.

Tot i que el terme *programador* només s'aplica a les persones que generen codi font en un llenguatge de programació, també podríem incloure en aquest grup altres especialistes com, per exemple, els experts en bases de dades que s'encarregaran de definir l'estructura de les taules i també d'escriure les consultes a la base de dades.

2.4.7. L'expert de qualitat (provador)

La seva feina és vetllar per la qualitat del producte desenvolupat. Tot i que els desenvolupadors (entenent com a desenvolupadors als arquitectes, analistes orgànics i programadors) han de vetllar per la qualitat del codi font i han de crear proves automatitzades, l'expert de qualitat complementa aquesta tasca aportant un punt de vista extern.

Per tant, l'expert de qualitat ha de partir dels requisits i dels programes i ha de verificar que, efectivament, els programes compleixen els requisits establerts. La seva responsabilitat és molt important: és qui ha de decidir si el sistema es pot posar o no en mans dels usuaris finals.

2.4.8. L'encarregat del desplegament

Un cop creats i validats els programes, l'encarregat del desplegament els ha d'empaquetar i enviar-los als entorns adequats per tal que arribin a mans dels usuaris finals.

La seva feina, doncs, està molt relacionada amb les activitats de gestió de la configuració i dels canvis.

2.4.9. El responsable de producte

Un altre rol molt important en alguns àmbits és el del responsable de producte. El responsable de producte és la persona que té la visió global del producte que es vol desenvolupar i vetlla pel seu desenvolupament correcte.

La tasca del responsable de producte no és fer un seguiment detallat del projecte (això ho fa el cap de projecte), sinó aportar una visió global del producte i assegurar-se que el projecte (o projectes, si n'hi ha més d'un de relacionat) encaixa perfectament amb els objectius i l'estratègia de l'organització.

El responsable de producte no ha de ser necessàriament un expert del domini, però en canvi sí que ha de conèixer perfectament l'estratègia de l'organització i els motius pels quals s'està desenvolupant el programari.

2.4.10. Altres rols

La llista de rols que hem esmentat no és exhaustiva, ja que, com s'ha dit abans, els rols dependran, en gran mesura, de com organitzi el desenvolupament el mètode concret que vulguem aplicar.

També hem de tenir en compte que hi haurà moltes persones que tindran una certa influència sobre el projecte encara que no formin part pròpiament de l'equip responsable del desenvolupament, com podria ser el cas del cap de desenvolupament, el director general de l'organització, etc.

3. Mètodes de desenvolupament de programari

Hem vist fins ara que, dins de l'àmbit de l'enginyeria del programari, l'activitat de desenvolupament s'acostuma a organitzar en forma de projectes i que la manera de gestionar aquests projectes estarà determinada pel mètode de desenvolupament que vulguem aplicar.

El mètode de desenvolupament definirà un cicle de vida (quines etapes formen part del projecte de desenvolupament), quins processos, activitats i tasques tenen lloc a les diferents etapes del cicle de vida, qui s'encarregarà de dur a terme cadascuna de les tasques i també la interacció entre tasques, rols i persones.

En aquest apartat veurem una classificació dels mètodes de desenvolupament que ens ajudarà a situar-los en context i a triar un mètode o altre segons les característiques del producte per desenvolupar. També estudiarem algunes de les famílies de mètodes més difoses actualment: el cicle de vida clàssic o en cascada, el desenvolupament iteratiu i incremental i el desenvolupament àgil/*lean*.

3.1. Història dels mètodes de desenvolupament

La gestió de projectes és una disciplina que s'ha estudiat àmpliament al llarg dels anys (especialment a partir dels anys cinquanta), la qual cosa ha donat lloc a tot un seguit de models que ens indiquen com hem de dur a terme la gestió dels projectes. Probablement, el model de gestió de projectes més difós avui dia és el del Project Management Institute (PMI).

Una de les primeres metàfores que es van aplicar al desenvolupament de programari és la gestió científica (*scientific management*, desenvolupada per Frederik Taylor al final del segle XIX). A grans trets, la gestió científica consisteix a descompondre el procés industrial en un conjunt de petites tasques el més repetitives possible que puguin ser executades per un treballador altament especialitzat, el rendiment del qual ha de ser fàcilment mesurable pels encarregats de gestionar el procés.

L'exemple clàssic de gestió científica és la cadena de producció, que va permetre reduir espectacularment el cost de la producció massiva en sèrie d'infinitat de productes industrials. És natural, doncs, que fos un dels primers mètodes industrials que es van intentar aplicar al desenvolupament de programari i que va donar lloc al cicle de vida en cascada (que veurem més endavant).

Amb el pas del temps, però, es va anar abandonant aquest enfocament en favor d'altres més flexibles tant en la indústria del desenvolupament de programari com en la de manufactura. Un dels mètodes de gestió amb més èxit ha estat el mètode de producció Toyota, que ha donat lloc al que s'anomena *lean-manufacturing*.

Enllaç d'interès

Toyota Production System http://www2.toyota.co.jp/en/vision/production_system/index.html.

El mètode de producció Toyota es basa en dos principis:

- *Jidoka*: evitar produir productes defectuosos aturant, si cal, la línia de producció;
- i la producció *just-in-time*: produir només aquells productes que són necessaris en la fase següent i no acumular estocs.

L'aplicació d'aquesta filosofia al desenvolupament de programari és el que es coneix com a *lean software development*.

Hi ha estudis (Cusumano, 2003) que han demostrat que, si bé una feina prèvia de planificació i descomposició del problema és una bona manera d'aconseguir un sistema amb pocs defectes, altres mètodes que posen èmfasi en l'obtenció d'informació basada en observacions fetes sobre un sistema real i en l'adaptació als canvis poden aconseguir una taxa de defectes similar, i permeten, a més, adaptar-se millor als canvis requerits pels usuaris.

La diversitat de mètodes de desenvolupament és tan gran que, a finals del 2009, un grup d'especialistes en enginyeria del programari format, entre altres, per gent tan diversa com Erich Gamma, Ivar Jacobson, Ken Schwaber o Edward Yourdon, van formar el grup SEMAT[3] amb la finalitat (entre altres) d'unificar el camp de l'enginyeria del programari.

(3) *SEMAT* són les sigles de *software engineering method and theory.*

El manifest del SEMAT diu: "L'enginyeria del programari està greument dificultada avui dia per pràctiques immadures. Els problemes específics inclouen:

- La prevalença de modes passatgeres més típiques de la indústria de la moda que d'una disciplina de l'enginyeria.
- La manca d'una base teòrica sòlida i àmpliament acceptada.
- L'enorme nombre de mètodes i variants de mètodes, amb diferències pobrament compreses i magnificades artificialment.
- La manca d'avaluació i validació experimental creïble.
- La separació entre la pràctica en la indústria i la recerca científica."

3.2. Classificació dels mètodes de desenvolupament

Tot i que les activitats per dur a terme són, a grans trets, les mateixes independentment del mètode, les diferències respecte a com i quan s'han de dur a terme donaran lloc a mètodes absolutament diferents, amb característiques molt diferenciades.

Així, per exemple, hi ha mètodes que aconsellaran fer un anàlisi exhaustiu i formal dels requisits abans de començar cap activitat de construcció, mentre que altres aconsellaran passar a la construcció tan bon punt es tinguin els requisits sense dur a terme cap mena de model.

Una de les primeres tasques de l'enginyer de programari serà, doncs, escollir el mètode de desenvolupament més adequat a la naturalesa del projecte que s'hagi de dur a terme; alguns dels factors que s'acostumen a fer servir per a classificar els projectes són (Wysocki, 2009):

- risc
- valor de negoci
- durada (menys de 3 mesos, 3 a 6 mesos, més de 6 mesos, etc.)
- complexitat
- tecnologia utilitzada
- nombre de departaments afectats
- cost

Així doncs, no seguirem el mateix procés per a desenvolupar un projecte curt (uns 3 mesos) amb poc risc (és a dir, les circumstàncies en què s'ha de desenvolupar el projecte són previsibles), que aporti un valor relativament petit i poc complex, que per a desenvolupar un projecte de 3 anys, en què hi ha molta incertesa i que ha de ser l'eina fonamental de treball per a la nostra organització.

Simplificant aquesta classificació, podem situar cadascun dels nostres projectes en un d'aquests quatre grups (Wysocki, 2009) segons si tenim clara la necessitat que volem cobrir (objectiu) i si coneixem o no els detalls de com serà la solució (requisits, tecnologia, etc.).

	Solució coneguda	Solució poc coneguda
Objectiu clar	1	2
Objectiu poc clar	3	4

Al grup 1 trobem els projectes per als quals està clar què volem fer i com ho farem. En aquest cas, podrem triar un mètode amb poca tolerància al canvi però que, en canvi, sigui senzill d'aplicar.

En canvi, per als projectes del grup 2 necessitarem un mètode que ens permeti canviar les idees inicials a mesura que el projecte avança i que faciliti el descobriment de la solució mitjançant cicles de retroalimentació. Avui dia, la majoria de projectes pertanyen a aquest grup.

Al grup 3 hi trobem un tipus de projecte bastant peculiar, ja que es tractaria de projectes en què tenim la solució però encara hem de buscar el problema.

Exemple del grup 3

Un exemple del grup 3 seria un projecte en què volem avaluar un producte existent per veure si cobreix alguna de les necessitats de l'organització. Així, per exemple, volem promoure la introducció del programari lliure a la nostra organització però no sabem en quines àrees traurem el màxim profit.

Finalment, al grup 4 hi trobaríem, per exemple, els projectes de recerca i desenvolupament, en què hem de ser flexibles tant respecte a la solució final que trobem com respecte al problema que solucionem, ja que, moltes vegades, s'acaba solucionant un problema diferent.

Post It

Un exemple de projecte de tipus 4 no relacionat amb el programari és el de les notes adhesives Post It. El 1968 els científics de 3M van desenvolupar un nou tipus d'adhesiu poc potent però reutilitzable, però no van tenir gaire èxit a promocionar-lo fins que, el 1974 un altre equip va decidir fer-lo servir per a enganxar notes adhesives i va crear un producte totalment diferent del que l'equip inicial tenia en ment.

Els projectes de desenvolupament acostumen a tenir un objectiu força clar i, per tant, a pertànyer als grups 1 o 2. Com es pot veure, la manera de gestionar uns i altres projectes variarà enormement, i per això és important que els enginyers de programari coneguin diferents mètodes de desenvolupament i puguin triar, en cada moment, el més adient.

Nosaltres classificarem els mètodes de desenvolupament ens tres grans famílies:

- Els que segueixen el cicle de vida en cascada (adequat per als projectes de tipus 1) per un costat,
- els mètodes iteratius i incrementals, i
- els àgils, per l'altre costat (més adequats per als de tipus 2).

Mètodes àgils

Tot i que la majoria dels mètodes àgils segueixen el cicle de vida iteratiu i incremental, els hem considerat una família diferent a causa de les seves peculiaritats.

3.2.1. Cicle de vida clàssic o en cascada

El cicle de vida clàssic o en cascada és ideal per a projectes del grup 1, ja que és molt senzill d'aplicar però, en canvi, és poc tolerant als canvis. La manera d'organitzar el desenvolupament és molt similar a una cadena de producció: tenim treballadors altament especialitzats (analista funcional, analista orgànic, programador, especialista en proves, arquitecte, etc.) que produeixen artefactes que són consumits per altres treballadors de la cadena com a part

del procés global de desenvolupament (l'analista funcional produeix requisits que l'analista orgànic transforma en dissenys que el programador transforma en codi, etc.).

El producte passa, progressivament, per les etapes següents:

- **Requisits**. Definir quin ha de ser el producte per desenvolupar. Com que el cicle de vida en cascada és poc flexible als canvis, aquesta etapa és crítica per a l'èxit del projecte, ja que un defecte als requisits es propagaria per la resta d'etapes i n'amplificaria els efectes nocius. Per a evitar aquest problema els diferents mètodes disposaran d'eines com les revisions de requisits o els llenguatges formals d'especificació com l'OCL[4].

(4) *OCL* són les sigles d'*object constraint language*.

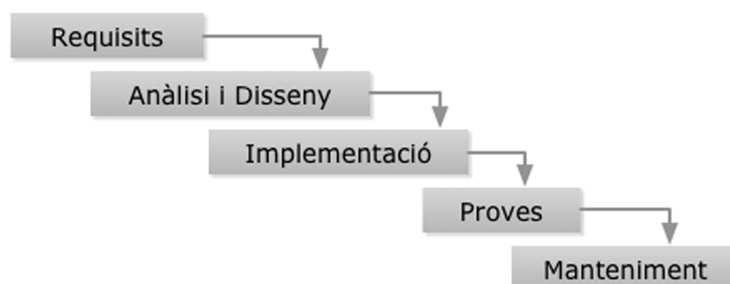
- **Anàlisi i disseny**. Definir com ha de ser el producte tant des del punt de vista extern com intern. L'anàlisi dóna un punt de vista extern documentant mitjançant models de què fa el sistema, mentre que el disseny dóna el punt de vista intern documentant com ho fa (quins components en formen part, com es relacionen entre ells, etc.).

- **Implementació**. Escriure el codi, els manuals i generar el producte executable. Aquest codi s'ha d'escriure segons les indicacions efectuades en la fase d'anàlisi i disseny.

- **Proves**. Es verifica que el producte desenvolupat es correspongui amb els requisits. En aquest punt es mostra el producte als usuaris finals per tal que en validin el resultat.

- **Manteniment**. Es posa el sistema a disposició de tots els usuaris i se'n corregeixen els defectes que es vagin trobant.

Cicle de vida en cascada



Per a millorar la tolerància als canvis i l'adaptabilitat d'aquests mètodes, es poden establir cicles de retroalimentació al final de cada etapa. Així doncs, per exemple, en finalitzar l'etapa d'anàlisi i disseny, podem revisar els requisits per tal d'incorporar les correccions als errors que hàgim anat trobant durant l'anàlisi i disseny.

En qualsevol cas, el cicle de vida en cascada es caracteritza pel seu caràcter seqüencial. Aquesta naturalesa afavoreix, tal com s'ha dit anteriorment, l'especialització dels membres de l'equip de desenvolupament en una sola activitat concreta, la qual cosa facilita, al seu temps, la possibilitat de tenir equips diferents especialitzats en tasques diferents. Així, per exemple, s'afavoreix tenir analistes que s'han especialitzat en modelització i programadors que s'han especialitzat en construcció.

3.2.2. Cicle de vida iteratiu i incremental

El cicle de vida en cascada té l'inconvenient (especialment per a projectes grans) que no tenim cap tipus d'informació empírica sobre el producte final fins que no estem a punt de finalitzar el projecte. Encara que s'hagin fet revisions de tots els artefactes, tot el coneixement que tenim sobre la marxa del projecte i sobre el producte final és teòric.

Quan arriba el moment en què tenim una versió del sistema que podem provar per a obtenir dades empíriques, ja és molt tard per a solucionar els possibles errors introduïts en les primeres fases del desenvolupament. Per tant, si seguim aquest cicle de vida, el cost dels errors en les primeres etapes és molt més gran que el cost dels errors en les etapes finals. Per això es considera poc adequat per als projectes del grup 2 en què no sabem, *a priori*, com serà el resultat final del desenvolupament i, per tant, és molt fàcil que cometem errors durant les fases inicials.

Per a aquest tipus de projectes, necessitarem un mètode que ens permeti canviar les idees inicials a mesura que el projecte avança i que faciliti el descobriment de la solució mitjançant l'obtenció d'informació empírica el més aviat possible: és el cas dels mètodes iteratius i incrementals com ara UP o els mètodes àgils.

Vegeu també

Veurem el procés unificat (UP) al subapartat 3.3.2 d'aquest mòdul didàctic.

Un altre problema del cicle de vida en cascada que intenten solucionar els mètodes iteratius és l'obtenció de resultats parcials utilitzables. Com que ens hem d'esperar a tenir tots els requisits, l'anàlisi i el disseny abans de començar la implementació, podem estar molt de temps (especialment en projectes llargs) invertint en el desenvolupament sense recuperar, ni que sigui parcialment, la inversió feta.

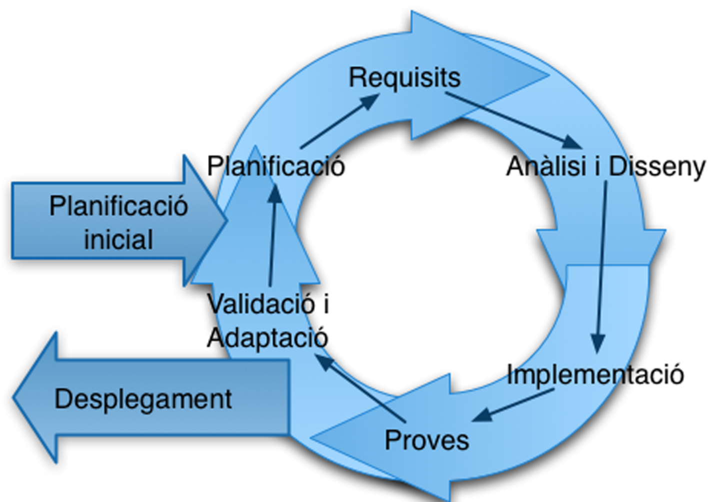
Un mètode iteratiu organitza el desenvolupament en una sèrie d'iteracions, cadascuna de les quals és un miniprojecte autocontingut que amplia el resultat final de la iteració anterior. D'altra banda, un mètode incremental ens assegura que al final de la iteració tenim un resultat final utilitzable. Així doncs, el cicle de vida iteratiu i incremental evita els dos problemes mencionats anteriorment:

- **Accelera la retroalimentació.** Com que cada iteració cobreix totes les activitats del desenvolupament (requisits, anàlisi, implementació i proves)

tenim informació sobre tots els àmbits del producte des del principi i, per exemple, si l'arquitectura que hem definit no es pot implementar, ho veurem al principi i no al final del projecte. També ens permet obtenir informació empírica basada en el producte real ja que la part que està desenvolupada és ja definitiva.

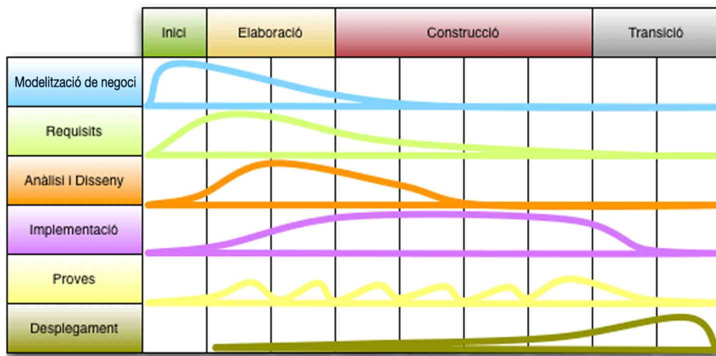
- **En tot moment es té un producte operatiu**. Com que es van afegint funcionalitats sobre un producte operatiu, en qualsevol moment es pot decidir fer servir el sistema desenvolupat, la qual cosa permet accelerar el retorn de la inversió o, fins i tot, finalitzar el projecte abans de consumir tots els recursos assignats si s'arriba a un punt en què el producte és "prou bo".

Cicle de vida iteratiu



Típicament, una iteració durarà entre una i sis setmanes. Per facilitar l'extrapolació de resultats (per exemple, en volum de feina efectuat en cada iteració), totes les iteracions acostumen a tenir la mateixa longitud (és el que s'anomena *time-boxing*). En aquest temps, cal treballar en totes les activitats del desenvolupament (planificació, requisits, anàlisi i disseny, implementació, proves, validació), tot i que, al llarg del temps, l'èmfasi que donem a cadascuna de les activitats anirà variant (les primeres iteracions faran més èmfasi en els requisits i l'anàlisi, mentre que les darreres el faran en les proves). Al final de cada iteració caldrà adaptar el procés i la planificació en funció del coneixement generat durant la iteració.

Èmfasi sobre les diferents activitats en el procés unificat (UP)



Cal tenir en compte, però, que cada iteració ha de ser un projecte complet i, per tant, ha d'incloure totes les etapes del cicle de vida en cascada (menys el manteniment). Un dels errors habituals en l'aplicació del cicle de vida iteratiu i incremental consisteix a disfressar d'iteratiu el cicle de vida en cascada i considerar cadascuna de les seves etapes com una iteració (fent, doncs, una iteració –o diverses– de requisits, després iteracions d'anàlisi, iteracions de disseny, etc.).

El desenvolupament iteratiu és incremental perquè al final de cada iteració es produeix un increment en el volum de funcionalitat implementada al sistema. Per tant, el producte final no es construeix de cop al final del projecte sinó que es va construint a mesura que avancen les iteracions.

El desenvolupament iteratiu i incremental també facilita la planificació centrada en el client i en els riscos, i avança la implementació d'aquelles funcionalitats que impliquen més riscos (per exemple, integrar una tecnologia desconeguda) i aquelles que ofereixen més valor per al client. Com a part del procés d'adaptació, els *stakeholders* acostumen a tenir la possibilitat de triar quines funcionalitats s'implementaran a cada iteració.

Vegeu també

Estudiareu més en detall el procés unificat al subapartat 3.3.2.

3.2.3. Desenvolupament *lean* i àgil

El desenvolupament àgil[5] és un conjunt de mètodes de desenvolupament iteratius que comparteixen uns principis que es van recollir al Manifest àgil el 2001.

(5)En anglès, *agile software development*.

Manifest per al desenvolupament àgil de programari

Estem posant al descobert millors maneres de desenvolupar programari fent-ho i ajudant d'altres a fer-ho.

Mitjançant aquesta feina, hem acabat valorant:

- **Individus i interaccions** per sobre de processos i eines.
- **Programari que funciona** per sobre de documentació exhaustiva.
- **Col·laboració amb el client** per sobre de negociació de contractes.
- **Resposta al canvi** per sobre de cenyir-se a una planificació.

És a dir, encara que els elements de la dreta tenen valor, nosaltres valorem més els de l'esquerra.

Font: http://www.agilemanifesto.org/iso/ca/

Cal tenir en compte que, a la frase final, es reconeix el valor dels elements de la dreta: els mètodes àgils no estan en contra d'aquests elements sinó que els consideren secundaris en relació amb els altres.

Exemple

Per exemple, els mètodes àgils consideren que les persones que participen en un projecte (i la manera com interactuen) són més decisives per a l'èxit que els processos que s'apliquen o les eines que fan servir. Per això, acostumen a ser mètodes poc estrictes pel que fa als processos que s'han de seguir i els artefactes que s'han de generar, per la qual cosa s'anomenen també *mètodes lleugers*.

Des del punt de vista de l'enginyeria del programari, aquest principi semblaria anar en contra de l'enfocament sistemàtic, ja que assumeix que un procés, per ben definit que estigui, sempre dependrà de les persones i, per tant, limita la transferibilitat del coneixement entre diferents execucions del mateix procés; en la pràctica, però, el que passa és que el procés definit en els mètodes àgils se centra més en les persones i les seves interaccions que no pas en els rols i en les tasques que assumeixen. No es tracta, doncs, de no tenir cap procés definit, sinó que el procés se centri en la interacció entre persones.

Aquests principis encaixen perfectament amb la filosofia del desenvolupament iteratiu i incremental, ja que els mètodes iteratius i incrementals donen prioritat al programari operatiu (el producte final de cada iteració és una versió operativa del programari) i faciliten la col·laboració amb el client i la resposta al canvi, i a més permeten canvis en la planificació de les diferents iteracions (es pot decidir prioritzar una funcionalitat o canviar-la per una altra sense afectar significativament la planificació general del projecte). De fet, tot i que el manifest àgil no fa cap referència al cicle de vida, la majoria dels mètodes àgils són iteratius i incrementals.

Una altra diferència important respecte al cicle de vida en cascada és que, com que amb els mètodes àgils el cost d'un canvi en la implementació és molt menor que en el cicle de vida en cascada, es dóna menys importància a la feina prèvia d'anàlisi i es potencia el canvi i l'adaptació.

El desenvolupament *lean*, com s'ha dit anteriorment, fa referència a les tècniques de manufactura *lean* derivades del sistema de producció de Toyota (el *Toyota production system*), i es va popularitzar en l'àmbit de l'enginyeria de programari arran de la publicació del llibre *Lean Software Development: An Agile Toolkit*, de Mary Poppendieck i Tom Poppendieck (2003). En aquest llibre, els autors expliquen la filosofia de manufactura *lean* i, al mateix temps, extrapolen els principis i pràctiques *lean* al desenvolupament de programari relacionant-lo amb el desenvolupament àgil. La filosofia *lean* es basa en set principis:

1) Evitar la producció innecessària. En la línia del concepte de producció *just-in-time*, es tracta de produir només allò que es necessita a l'etapa següent. Així, en comptes de produir tots els requisits abans de fer l'anàlisi, es produeixen només els necessaris per a poder fer l'anàlisi d'aquesta iteració. En aquesta mateixa línia, també ens hem de concentrar a produir només els artefactes que realment aporten valor; a grans trets, ho podríem resumir amb la pregunta: "hi ha algú esperant aquest artefacte?" o "algú es queixarà si aquest artefacte no està actualitzat?" Sorprenentment, moltes vegades la resposta a aquesta pregunta és que no.

2) Amplificar l'aprenentatge. Cal recollir tota la informació que es va generant sobre el producte i la seva producció (retroalimentació, iteracions curtes, etc.) per tal d'entrar en un cicle de millora contínua de la producció.

3) Decidir el més tard possible. Intentar prendre totes les decisions amb el màxim d'informació possible. Això significa endarrerir qualsevol decisió fins al punt en què no prendre la decisió és més car que prendre-la.

> Per exemple, si un estudiant pot decidir anul·lar la convocatòria d'un examen fins a tres dies abans de l'examen però no sap si tindrà temps d'estudiar hauria de prendre la decisió tres dies abans de l'examen, ja que si la pren abans no tindrà tanta informació sobre si el pot aprovar o no com en aquell moment, però si no la pren en aquell moment ja serà massa tard per a decidir.

4) Lliurar el producte com més aviat millor. Cal que el procés de desenvolupament ens permeti implementar ràpidament els canvis i funcionalitats que vulguem afegir al sistema d'informació.

5) Donar poder a l'equip. És important que els equips de persones que formen part del desenvolupament se sentin responsables del producte i, per tant, cal evitar que hi hagi un gestor que prengui totes les decisions. És l'equip de desenvolupament qui decideix sobre les qüestions tècniques i qui es responsabilitza de complir els terminis.

6) Incorporar la integritat. El programari s'ha de mantenir útil al llarg del temps, adaptant-se si cal a nous requisits, per la qual cosa cal desenvolupar-lo de manera que es pugui canviar fàcilment.

7) Visió global. Cal evitar les optimitzacions parcials del procés que poden causar problemes en altres etapes i adoptar una visió global del sistema. Per exemple, si decidim eliminar una funcionalitat per motius tècnics, cal tenir en compte com pot afectar això el valor del producte des del punt de vista de l'organització. Aquest principi va en contra dels treballadors hiperespecialitzats que proposava el cicle de vida en cascada.

Els principis *lean* encaixen perfectament amb els principis del Manifest àgil (de fet, els podríem considerar un superconjunt dels principis àgils) tot i que, a diferència del Manifest àgil, molt centrat en el punt de vista dels desenvolupadors, la filosofia *lean* intenta integrar el desenvolupament de programari dins un marc conceptual i un conjunt de pràctiques aplicats a altres indústries i que afecten tots els aspectes de l'organització (no solament el desenvolupament).

3.3. Exemples de mètodes de desenvolupament

Una de les conclusions de Cusumano (2003) és que les pràctiques per a organitzar el desenvolupament no es podien adoptar de manera aïllada sinó que han de formar part d'un tot coherent que ens ajudi a neutralitzar els efectes negatius d'algunes pràctiques. Per exemple, si no fem un model de requisits molt detallat abans de començar el desenvolupament, és important que mostrem un prototip o una versió beta del sistema als usuaris finals per tal de poder assegurar-nos de què el sistema desenvolupat satisfà correctament les seves necessitats i que podrem fer canvis amb poc cost si el producte no satisfà els requisits reals dels usuaris.

Per a aconseguir aquesta coherència entre pràctiques es van desenvolupar els mètodes de desenvolupament que, tal com hem dit anteriorment, no solament inclouen el cicle de vida sinó que donen indicacions més detallades sobre els rols, les activitats, els artefactes i les pràctiques que s'han d'aplicar en el desenvolupament de programari.

A continuació en descrivim tres:

1) Un que segueix el cicle de vida en cascada (Mètrica 3),
2) un d'iteratiu i incremental (el procés unificat o UP),
3) un d'àgil (Scrum).

La finalitat d'aquest subapartat no és donar una descripció en profunditat dels mètodes, sinó donar una visió general que permeti fer una anàlisi comparativa entre aquests.

3.3.1. Mètrica 3

El mètode Mètrica 3 el va desenvolupar el Ministeri d'Administracions Públiques de l'Estat espanyol amb la finalitat de servir de referència a totes les administracions públiques de l'Estat.

> **Nota**
>
> Mètrica 3 es refereix a la versió 3 de Mètrica que es va publicar l'any 2000. La primera versió es va publicar el 1989.

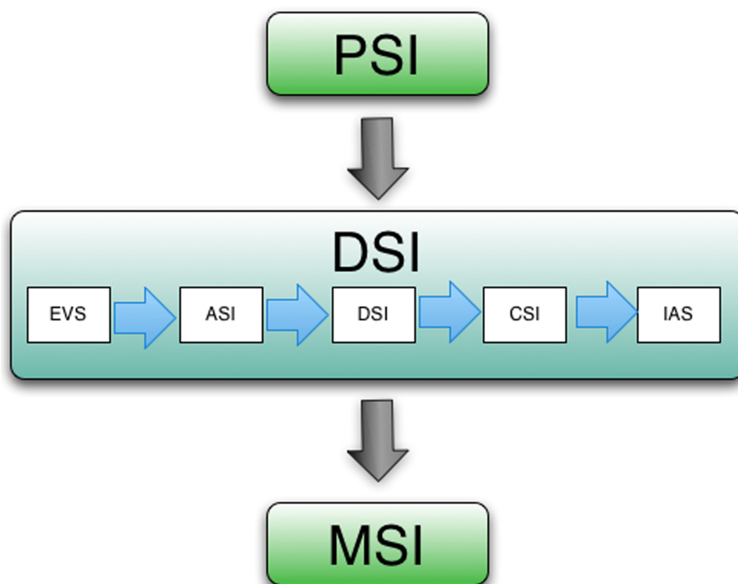
Mètrica 3 distingeix tres processos, el segon dels quals inclou cinc subprocessos:

1) planificació de sistemes d'informació (PSI)

2) desenvolupament de sistemes d'informació (DSI)
- **a)** estudi de viabilitat del sistema (EVS)
- **b)** anàlisi del sistema d'informació (ASI)
- **c)** disseny del sistema d'informació (DSI)
- **d)** construcció del sistema d'informació (CSI)
- **e)** implantació i acceptació del sistema (IAS)

3) manteniment de sistemes d'informació (MSI)

Els processos de Mètrica 3



Com podem veure, Mètrica 3 inclou processos més enllà del desenvolupament pròpiament dit i regula tant el procés previ de planificació com el procés posterior de manteniment.

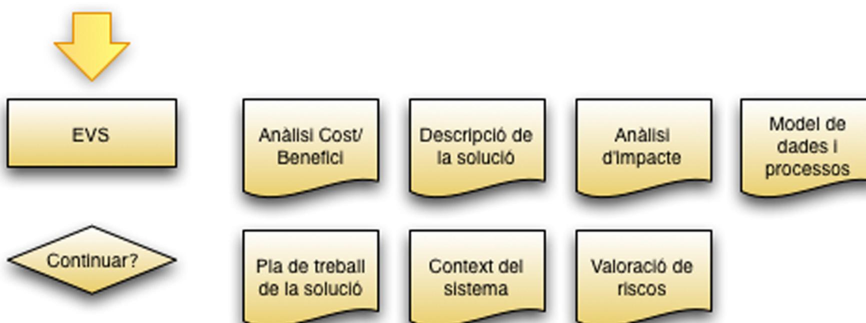
El pla de sistemes d'informació té com a finalitat assegurar que el desenvolupament dels sistemes d'informació es fa de manera coherent amb l'estratègia corporativa de l'organització. El pla de sistemes d'informació, per tant, és la guia principal que han de seguir tots els projectes de desenvolupament que comenci l'organització. Com a tal, incorpora un catàleg de requisits que han de complir tots els sistemes d'informació, una arquitectura tecnològica, un model de sistemes d'informació, etc.

Mètrica 3: artefactes i tasques del procés PSI

El procés de desenvolupament del sistema d'informació pròpiament dit comença amb l'estudi de viabilitat del sistema (EVS). Aquest estudi respon la pregunta de si cal o no continuar amb el desenvolupament del sistema. Per a donar aquesta resposta, tindrà en compte criteris econòmics, legals, tècnics i operatius.

Com a resultat de l'estudi de viabilitat del sistema obtindrem, per exemple, una anàlisi cost/benefici de la solució, una planificació, una descripció de la solució, un model de descomposició en subsistemes, etc. Segons el tipus de projecte (desenvolupament o implantació d'un producte estàndard) tindrem altres artefactes com el model de dades, el model de processos o la descripció del producte, un anàlisi de costos del producte, etc.

Mètrica 3: estudi de viabilitat del sistema



Etapes del cicle de vida en cascada

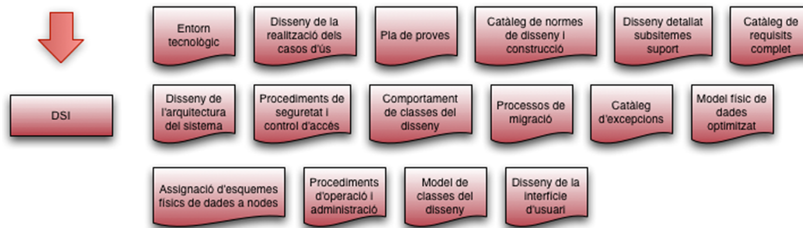
A continuació es portaran a terme les etapes típiques del cicle de vida en cascada: anàlisi del sistema d'informació (ASI), disseny del sistema d'informació (DSI), construcció del sistema d'informació (CSI) i implantació i acceptació del sistema d'informació (IAS).

L'anàlisi del sistema d'informació (ASI) té la finalitat d'aconseguir l'especificació detallada del sistema d'informació, mitjançant un catàleg de requisits i una sèrie de models que cobreixin les necessitats d'informació dels usuaris per als quals es desenvoluparà el sistema d'informació. En aquest procés s'inicia, també, l'especificació del pla de proves que es completarà a la fase següent (DSI).

Mètrica 3: anàlisi del sistema d'informació

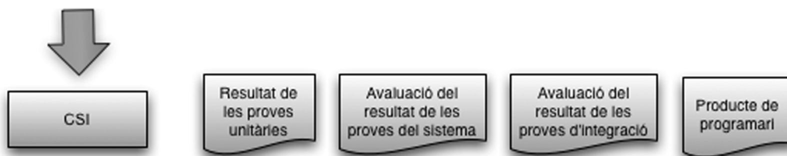
El propòsit del disseny del sistema d'informació és obtenir la definició de l'arquitectura del sistema i de l'entorn tecnològic que li ha de donar suport, juntament amb l'especificació detallada dels components del sistema d'informació.

Mètrica 3: disseny del sistema d'informació



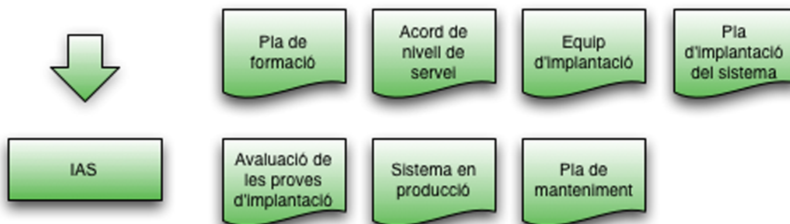
La construcció del sistema d'informació (CSI) té com a objectiu final la construcció i prova dels diferents components del sistema d'informació, a partir del conjunt d'especificacions lògiques i físiques d'aquest, obtingut en el procés de disseny del sistema d'informació (DSI). Es desenvolupen els procediments d'operació i seguretat i s'el·laboren els manuals d'usuari final i d'explotació, aquests darrers quan sigui procedent.

Mètrica 3: construcció del sistema d'informació



El procés d'implantació i acceptació del sistema (IAS) té com a objectiu principal el lliurament i l'acceptació del sistema en la seva totalitat, que pot comprendre diversos sistemes d'informació desenvolupats de manera independent, segons s'hagi establert al procés d'estudi de viabilitat del sistema (EVS), i un segon objectiu és portar a terme les activitats oportunes per al pas a producció del sistema.

Mètrica 3: implantació i acceptació del sistema



A partir d'aquest moment es considera que el desenvolupament de l'SI està finalitzat i comença l'última etapa: el procés de manteniment del sistema d'informació (MSI).

Mètrica 3: manteniment del sistema d'informació

3.3.2. Procés unificat

El procés unificat (UP, *unified process*) va ser proposat per l'empresa Rational Software (actualment part d'IBM) amb la intenció de ser als mètodes de desenvolupament de programari el que l'UML[6] va ser als llenguatges de modelització: una versió unificada dels diferents processos existents que incorporés les millors pràctiques existents. Intenta millorar respecte al cicle de vida en cascada per la via de millorar la productivitat dels equips, l'ús extensiu de models i l'adopció d'un cicle de vida iteratiu i incremental.

(6) *UML* són les sigles de *unified modeling language*.

Nota

El primer llibre sobre UP es va publicar el 1999 amb el nom *The Unified Software Development Process*. Els autors (Ivar Jacobson, Grady Booch i James Rumbaugh) són considerats els pares del procés unificat.

UP és un bastiment de processos configurable que es pot adaptar a diferents contextos, de manera que no s'aplica igual en el desenvolupament d'una Intranet d'una empresa que en el desenvolupament de programari militar. En el nostre cas, ens centrarem en la variant OpenUP per a la descripció de tasques i rols, ja que és una de les més senzilles i, a més, està disponible públicament. A diferència d'altres variants com RUP[7], OpenUP assumeix els valors del desenvolupament àgil i els té en compte a l'hora de personalitzar el procés unificat.

(7) *RUP* són les sigles de *rational unified process*.

En termes generals, però, totes les variants del procés unificat es basen a potenciar sis pràctiques fonamentals (segons el *rational unified process: best practices for software development teams*):

1) Desenvolupament iteratiu i incremental

2) Gestió dels requisits (mitjançant casos d'ús)

3) Arquitectures basades en components (mòduls o subsistemes amb una funció clara)

4) Utilització de models visuals (mitjançant el llenguatge UML)

5) Verificació de la qualitat del programari (al llarg de totes les etapes del procés, no solament al final)

6) Control dels canvis al programari

A més a més, UP promou una gestió proactiva dels riscos, i suggereix que es construeixin abans les parts que incloguin més riscos, i també fomentan la construcció d'una arquitectura executable durant les primeres etapes del projecte.

El procés es pot descompondre de dues maneres: en funció del temps (en fases) i en funció de les activitats o contingut (activitats). A més, tal com hem dit que ha de fer qualsevol mètode, definirà els rols que tenen les diferents persones i quines tasques ha de dur a terme cada rol.

Fases del projecte

Al llarg del temps el projecte passarà (de manera seqüencial) per quatre fases:

1) Inici (*inception*). S'analitza el projecte des del punt de vista de l'organització i se'n determina l'àmbit, l'estimació dels recursos necessaris, la identificació de riscos i de casos d'ús.

2) Elaboració (*elaboration*). S'analitza el domini del sistema per desenvolupar fins a tenir un conjunt de requisits estable, s'eliminen els riscos principals i es construeix la base de l'arquitectura (aquesta serà executable).

3) Construcció (*construction*). Es desenvolupa la resta de la funcionalitat (de manera seqüencial o en paral·lel) i s'obté, com a resultat, el producte i la documentació.

4) Transició (*transition*). Es posa el producte a disposició de la comunitat d'usuaris. Això inclou les proves beta, la migració d'entorns, etc.

Fases i fites del procés unificat

Al final de cada fase s'arriba a una fita en què s'han de poder respondre una sèrie de preguntes i decidir si es continua o no amb el projecte.

La fita d'objectius analitza el cost i els beneficis esperats del projecte; hem de ser capaços de respondre dues preguntes: estem d'acord sobre l'àmbit (què ha de fer i què no ha de fer el sistema) i els objectius del projecte?; estem d'acord sobre si val la pena continuar?

Per a poder prendre aquestes decisions necessitarem haver explorat prèviament quina serà la funcionalitat per desenvolupar, a qui pot afectar el nou sistema i qui hi haurà d'interactuar. També necessitarem tenir una idea apro-

ximada de quina serà la solució final i la seva viabilitat. Finalment, caldrà tenir una estimació inicial del cost i del calendari, i també dels riscos que poden afectar el desenvolupament correcte del projecte.

La segona fita (d'arquitectura) arriba quan ja tenim una versió més o menys estable i final de l'arquitectura i ja hem implementat les funcionalitats més importants i eliminat els principals riscos. És el moment de plantejar-nos si l'arquitectura que hem definit ens servirà per a dur a terme tot el projecte, si considerem que els riscos que queden estan sota control i també si creiem que estem afegint valor a bon ritme.

Per això, haurem hagut d'explorar els requisits en més detall, de manera que tinguem una estimació clara del cost pendent del projecte, i també dels problemes que ens podem trobar. També necessitarem haver implementat un nombre significatiu de funcionalitats per tal de poder estar segurs sobre la idoneïtat de l'arquitectura escollida.

La tercera fita (capacitat operacional inicial) arribarà en el moment en què considerem que el sistema està prou desenvolupat per a lliurar-lo a l'equip de transició. La principal pregunta que ens hem de fer arribats a aquesta fita és si podem dir que ja hem implementat tota la funcionalitat que calia implementar.

La darrera fita (lliurament del producte) és quan hem de decidir si hem aconseguit els objectius inicials del projecte. En aquest cas, és important que els clients acceptin el projecte que hem lliurat.

Amb el procés unificat, cadascuna de les fases inclou activitats que abasten tot el cicle de desenvolupament: des dels requisits fins a la implementació i les proves. Per exemple, en la fase d'inici, la definició de l'arquitectura inclou la creació d'una versió executable d'aquesta.

L'avantatge principal d'aquest canvi de filosofia respecte al cicle de vida en cascada és que, amb el procés unificat, podem prendre les decisions basant-nos en informació empírica i no solament en informació teòrica.

Malauradament, és molt habitual veure equips que creuen estar seguint el procés unificat però que en realitat estan seguint una variant del cicle de vida en cascada ja que, *a priori*, podria semblar que les fases d'UP no són gaire diferents de les etapes definides, per exemple, en Mètrica 3 (*inception* podria ser el PSI, *elaboration* seria EVA + ASI, *construction* DSI + CSI i *transition* IAS + MSI), però, com hem dit, les diferències són molt grans i molt importants: encara que els objectius són similars, la manera d'assolir-los és molt diferent.

Activitats

Pel que fa a les activitats, UP defineix les activitats principals següents:

- **Modelització de negoci (*business modelling*)**. Aquesta activitat intenta solucionar el problema de comunicació entre els experts en el domini i els especialistes en tecnologia que, habitualment, fan servir llenguatges diferents. Es generen els casos d'ús "de negoci" (*business use cases*), que descriuen els processos principals de l'organització i que serviran com a llenguatge comú per a tots els implicats en el procés de desenvolupament. Aquesta activitat és molt important, ja que és la que ens assegurarà que els desenvolupadors entenguin quin és l'encaix del producte que estan desenvolupant dins del context de l'organització per al qual l'estan desenvolupant.

- **Requisits (*requirements*)**. Descriure **què** ha de fer el sistema (i què no ha de fer). El model que es fa servir per a descriure la funcionalitat del sistema és l'anomenat *model de casos d'ús*, que consisteix a descriure escenaris d'ús del sistema mitjançant una seqüència d'esdeveniments (l'usuari fa *X*, el sistema fa *Y*, etc.).

- **Anàlisi i disseny (*analysis & design*)**. Descriure **com** s'implementarà el sistema. Es creen models detallats dels components que formaran el sistema. Aquests models serveixen com a guia per als implementadors a l'hora d'escriure el codi font dels programes. Aquests models han d'estar relacionats amb els casos d'ús i han de permetre introduir canvis en el cas que els requisits funcionals canviïn.

- **Implementació (*implementation*)**. Definir l'organització del codi, escriure'l i verificar que cada component compleix els requisits de manera unitària (és a dir, de manera aïllada de la resta de components) i generar un sistema executable. El procés també preveu la reutilització de components que existeixin prèviament.

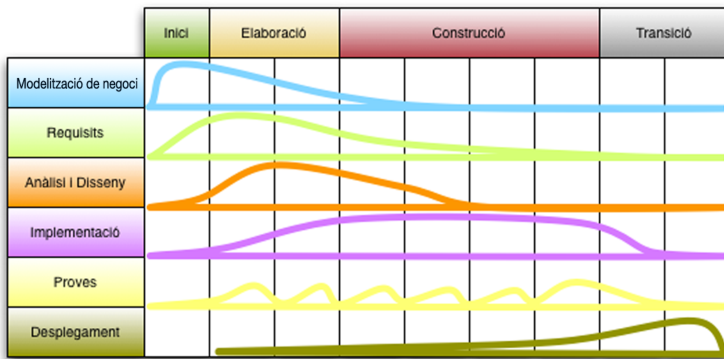
- **Proves (*test*)**. Verificar la interacció entre els objectes i components que formen el sistema. Com que les proves s'executen durant totes les iteracions, és més fàcil detectar errors abans que es propaguin pel sistema. Les proves han d'incloure la fiabilitat, la funcionalitat i el rendiment.

- ***Deployment***. Produir els lliuraments del sistema i lliurar-los als usuaris finals. Inclou les tasques relatives a la gestió de la configuració i de les versions i, sobretot, té lloc durant la fase de transició.

A més a més de les activitats aquí descrites, el procés unificat també preveu altres activitats, com ara la personalització del procés mateix (recordem que UP és un bastiment de processos i no pas un procés concret), la gestió de la configuració i del canvi i també la planificació i gestió del projecte.

És important recordar que totes les activitats tenen lloc durant totes les etapes, tot i que amb diferent èmfasi. Així doncs, durant l'etapa d'elaboració, es dedicarà més temps a la modelització de negoci que no pas a la implementació, mentre que durant l'etapa de construcció, la proporció es veurà invertida. Aquesta variabilitat en l'èmfasi que es dedica a cada activitat s'acostuma a representar mitjançant el diagrama següent.

Èmfasi de les diferents activitats en les fases del procés unificat



Rols

OpenUP defineix els rols següents:

- ***Stakeholder***. Qualsevol persona que tingui un interès en el resultat final del projecte.

- **Cap de projecte**. Encarregat de la planificació i de coordinar la interacció entre els *stakehoders*. També és responsable de mantenir els membres de l'equip centrats a aconseguir complir els objectius del projecte.

- **Analista**. Recull la informació dels *stakeholders* en forma de requisits, els classifica i prioritza. És el que, en la classificació que hem fet al subapartat 2.4, hem anomenat *analista funcional*.

- **Arquitecte**. Defineix l'arquitectura del programari, la qual cosa inclou prendre les decisions clau que afecten tot el disseny i implementació del sistema.

- **Desenvolupador**. Desenvolupa una part del sistema, la qual cosa inclou dissenyar-la d'acord amb l'arquitectura, prototipar (si cal) la interfície gràfica d'usuari, implementar-la i provar-la tant aïllada de la resta del sistema con integrada amb la resta de components. Aquest rol inclou els rols d'analista orgànic i el de programador, tal com els hem definit al subapartat 2.4.

- **Expert en proves**. Identifica, defineix, implementa i duu a terme les proves aportant un punt de vista complementari al del desenvolupador. El

més habitual és que treballi sobre el sistema construït i no sobre components aïllats.

3.3.3. Scrum

Tot i ser contemporani d'UP, Scrum es va popularitzar posteriorment com a part del moviment del desenvolupament àgil. És un mètode iteratiu i incremental per a desenvolupar programari elaborat per Ken Schwaber i Jeff Shuterland. Es tracta d'un mètode molt lleuger que, seguint el principi *lean* de generar només els artefactes que aporten un valor important, minimitza el conjunt de pràctiques i artefactes, i també les tasques i els rols.

Nota

La descripció del mètode Scrum el trobareu a *Scrum Guide*: http://www.scrum.org/scrumguideenglish/ (darrera visita: setembre 2010).

Rols

Scrum distingeix, d'entrada, entre dos grans grups de participants d'un projecte: els que estan compromesos en el desenvolupament (l'equip) i els que hi estan involucrats però no formen part de l'equip (el que altres mètodes anomenen *stakeholders*).

Pollastres i porcs

Scrum anomena *pollastres* els *stakeholders* i *porcs* l'equip. Aquests noms vénen de la història següent: un pollastre i un porc es troben i el pollastre diu: "Muntem un restaurant". El porc s'ho repensa i li demana: "Com l'anomenarem?". El pollastre contesta: "Ous amb cansalada". I el porc diu: "No, gràcies, perquè jo estaria compromès però tu només estaries involucrat".

Entre els membres de l'equip, Scrum defineix tres rols molt particulars:

- ***Scrum Master***. És responsable d'assegurar que l'equip segueix els valors, pràctiques i regles de Scrum. També s'encarrega d'eliminar impediments que l'equip es pugui trobar dins l'organització.

- ***Product owner***. És la persona responsable de vetllar pels interessos de tots els *stakeholders* i portar el projecte a bon terme. És, per tant, l'encarregat de decidir què s'implementa i què és més prioritari.

- ***Team***. La resta de membres de l'equip són els desenvolupadors. No s'especialitzen sinó que tots han de tenir, en més o menys grau, habilitats en totes les activitats implicades. Els membres de l'equip decideixen ells mateixos com s'organitzen la feina i ningú (ni tan sols l'*Scrum Master*) els pot dir com ho han de fer.

Artefactes

Scrum només defineix quatre artefactes:

- ***Product backlog***. És la llista de requisits pendents d'implementar al producte. Cada entrada (normalment una "història d'usuari") té associada una

estimació del valor que aporta a l'organització i també del cost del seu desenvolupament.

- ***Sprint backlog***. El *backlog* per a una iteració concreta (Scrum anomena *sprint* les iteracions); més detallat que el *product backlog* i en què cada història d'usuari està descomposta en tasques entre 4 i 16 hores de durada.

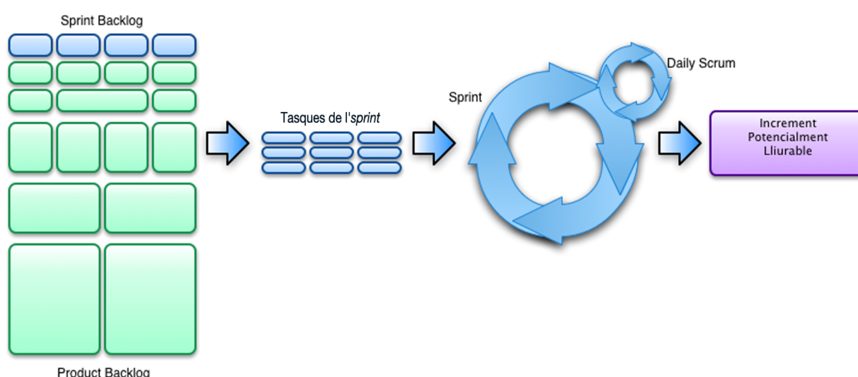
- ***Release burndown chart***. Gràfic que mostra el progrés actual de l'equip en funció del nombre d'històries d'usuari que falten per implementar.

- ***Sprint burndown***. El *burndown* per a una iteració concreta, en què el progrés es pot mesurar en tasques finalitzades encara que no siguin històries d'usuari completes.

Pràctiques

Pel que fa a les pràctiques, es basa en dos cicles d'iteracions: una iteració més llarga (com les que podem trobar en mètodes com UP) i una "iteració diària".

Procés de Scrum



- ***Sprint planning meeting***. Reunió que es fa abans de començar un *sprint* en què es decideixen quines històries d'usuari s'implementaran en aquest *sprint* i, per tant, es crea l'*sprint backlog*.

- ***Daily scrum***. Reunió diària on tots els membres de l'equip responen tres preguntes: què van fer ahir, què pensen fer avui i quins impediments s'han trobat que els han impedit avançar. La finalitat d'aquesta reunió és que tots els membres de l'equip estiguin al corrent de què està fent la resta de membres i així s'identifiquin oportunitats d'ajudar-se els uns als altres.

- ***Sprint review meeting***. En finalitzar un *sprint* es revisa la feina feta i s'ensenya a qui estigui interessat el resultat implementat (la *demo*).

- ***Sprint retrospective***. Serveix per a reflexionar sobre el que hagi passat durant l'*sprint* i per a identificar oportunitats de millora en el procés de desenvolupament.

4. Tècniques i eines de l'enginyeria del programari

L'enginyeria del programari no consisteix només en una manera d'organitzar el desenvolupament, sinó que també ens proporciona un conjunt de tècniques i eines que ens han d'ajudar a posar en pràctica els mètodes que hàgim triat per a desenvolupar el nostre projecte de programari.

De tot el ventall de tècniques de l'enginyeria del programari, en veurem només unes quantes, que hem separat en dos grans grups: les tècniques basades en la reutilització i les basades en l'abstracció.

També parlarem de les eines de suport a l'enginyeria del programari, ja que aquestes eines ens ajuden a aplicar amb èxit les tècniques esmentades.

4.1. Tècniques basades en la reutilització

Un dels principals objectius de l'enginyeria del programari és afavorir la reutilització. En desenvolupar un nou producte, la situació ideal és no partir de zero, sinó tenir parts ja desenvolupades i aprofitar-les en el nou context. Això ens aportarà tota una sèrie d'avantatges:

Lectura recomanada

Bertrand Meyer (1999). *Construcción de Software orientado a objetos*. Madrid: Prentice Hall.

- **Oportunitat**. Com que hem de desenvolupar menys programari, es pot desenvolupar amb més rapidesa i, per tant, tenir-lo enllestit abans.

- **Disminució dels costos**. Com que el component és compartit, el cost del seu desenvolupament i manteniment també pot ser compartit.

- **Fiabilitat**. Els components reutilitzats ja han estat provats i utilitzats i, per tant, podem confiar en el seu bon funcionament ja que, si tenen errors, algú altre els pot haver trobat abans.

- **Eficiència**. El desenvolupador del component reutilitzable es pot especialitzar en un problema molt concret (el que resol el component) i, per tant, trobar les solucions més eficients.

Tal com diu Meyer (1999), la reutilització té una gran influència sobre tots els altres aspectes de la qualitat del programari, ja que en resoldre el problema de la reutilització s'haurà d'escriure menys programari i, en conseqüència, es podran dedicar més esforços (pel mateix cost total) a millorar els altres factors com ara la correcció i la robustesa.

Exemple de reutilització

Un exemple senzill de reutilització són els controladors dels dispositius. En els temps de l'MS-DOS (anys vuitanta - començaments del noranta) els desenvolupadors d'aplicacions havien d'escriure els seus propis controladors dels dispositius (per exemple, de la impressora o de la targeta de so) i, per tant, havien de dedicar part dels recursos a suportar una gama el més àmplia possible de dispositius. Un dels motius de l'èxit del sistema Windows va ser, precisament, la incorporació d'un sistema de controladors genèric que va permetre als desenvolupadors centrar-se en altres àrees del programa que aportessin més valor afegit.

La reutilització no és pas un concepte nou, sinó que ha estat present al llarg de la història del desenvolupament i, de fet, és anterior a l'enginyeria del programari. Al llarg del temps s'ha anat augmentat progressivament el nivell de reutilització, però com que el programari ha anat guanyant en complexitat durant aquest període (en part, gràcies a la reutilització), la necessitat de reutilització, en comptes de minvar, ha crescut.

Avui dia, exemples com el que hem comentat anteriorment ja no es consideren exemples de reutilització, sinó que es donen per suposats i s'espera dels enginyers del programari que assoleixen noves fites en aquest sentit i reutilitzin components cada vegada més grans i més complexos, i també que apliquin la reutilització a totes les activitats possibles: des de la presa de requisits fins a la implementació i el desplegament.

La reutilització, però, no està exempta de costos i problemes. Per a afavorir la reutilització cal desenvolupar els diferents components del programari amb aquest objectiu ja que, en cas de no fer-ho, serà molt difícil aprofitar-los en un context diferent del context en què es van crear.

A més, haurem de documentar els diferents components de manera molt més exhaustiva, ja que, idealment, els equips que reutilitzin un component ho haurien de poder fer sense necessitat de veure ni modificar el seu codi font.

D'altra banda, caldrà anar amb especial compte a l'hora de desenvolupar i mantenir un component reutilitzat, ja que qualsevol defecte que aquest tingui es propagarà a tots aquells sistemes que el fan servir.

Finalment, també caldrà fer una gestió exhaustiva de les diferents versions del component i, a l'hora d'evolucionar-lo, tenir en compte tots els contextos en què s'està fent servir. Un mateix component pot ser utilitzat en diversos sistemes i, segurament, cada sistema en farà servir una versió diferent. Quan calgui corregir errades o afegir funcionalitat caldrà anar amb molt de compte amb la compatibilitat entre versions.

Imaginem que desenvolupem un component C per a un determinat sistema S1. En desenvolupar un segon sistema S2 ens adonem que podem reutilitzar el component C, i llavors ho fem. Per a això ens cal registrar que la versió actual del component és la 1.0 i que està en ús en els sistemes S1 i S2.

Més endavant, el client ens demana un canvi al sistema S1 que, en implementar-lo, afecta el component C, i origina la versió 2.0, incompatible amb la 1.0. Ara tenim el sistema S1, que fa servir la versió 2.0, i el sistema S2 amb la 1.0.

Si, més endavant, detectem una errada al component C que afecta la versió 1.0, l'haurem de corregir en aquesta versió (i crear, per exemple, la versió 1.1, que farà servir el sistema S2) però també a la versió 2.0 (creant la versió 2.1, que es farà servir en el sistema S1).

A una escala no tecnològica, la reutilització també genera qüestions com ara com podem assignar el cost del desenvolupament d'un component reutilitzable, què passa si després ningú no el reutilitza o qui se n'ha de fer càrrec del manteniment.

Una possibilitat és el desenvolupament, *a priori*, de components reutilitzables, però com diu Meyer (1999), "hem de trobar primer les maneres de resoldre un problema abans de preocupar-nos per aplicar la solució a altres problemes".

Dit d'una altra manera: és molt difícil crear una solució genèrica si no s'ha creat abans una solució específica que ens permeti explorar tots els aspectes importants del problema.

Finalment, l'estandardització dels components en la indústria requereix l'elaboració d'una sèrie de normes i acords que han de ser acceptats per tots els participants, i això no és sempre fàcil.

La història de la reutilització ha estat bastant complicada i no han faltat els èxits ni els fracassos. Avui dia, però, encara es continua investigant en noves maneres d'incrementar el grau de reutilització i portar-lo al nivell que hi ha en altres enginyeries, en què el grau d'estandardització i reutilització dels components és encara bastant més alt que en el cas de l'enginyeria del programari.

A continuació estudiarem algunes de les tècniques que s'han anat desenvolupant al llarg dels anys per a millorar els nivells de reutilització.

4.1.1. Desenvolupament orientat a objectes

La programació orientada a objectes es basa en dues tècniques bàsiques que faciliten la reutilització: l'ocultació d'informació i l'abstracció.

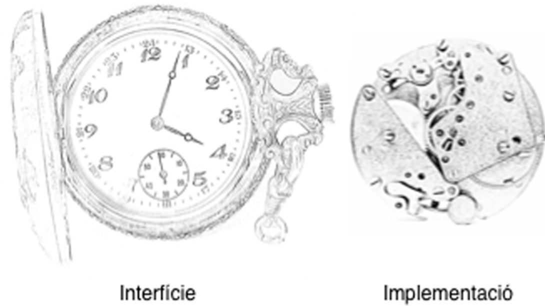
Vegeu també

Parlarem de la programació orientada a objectes en més detall al mòdul "Orientació a objectes".

L'ocultació d'informació consisteix a amagar els detalls sobre l'estructura interna d'un mòdul, de manera que podem definir clarament quins aspectes dels objectes són visibles públicament (i, per tant, utilitzables pels reutilitzadors) i quins no.

Anomenem *interfície* aquella part d'un component de programari que és visible als programadors que el fan servir. Així, quan un programador reutilitza un objecte, només n'ha de conèixer la interfície i, en tot cas, un contracte que li indiqui com funciona la interfície. Per oposició, la implementació d'un component és aquella part del component que queda oculta dels seus usuaris.

Interfície i implementació



En aquesta figura, hem distingit la interfície del component rellotge de la seva implementació. Com a usuaris, en tenim prou de saber que el rellotge ens ofereix una única operació ("Quina hora és?"), i que ens dóna aquesta informació mitjançant les manilles. Això permet que fem servir el rellotge amb independència del mecanisme (a corda, a pila, etc.) amb què estigui implementat.

L'ocultació d'informació facilita la reutilització perquè si un manteniment correctiu o evolutiu d'un component es pot fer modificant-ne només la implementació, sense modificar-ne la interfície, aleshores els usuaris d'aquest component no es veuran afectats.

L'abstracció consisteix a identificar els aspectes rellevants d'un problema de manera que ens puguem concentrar només en aquests.

En el cas de l'orientació a objectes, l'abstracció ens ajuda a agrupar els aspectes comuns a una sèrie d'objectes de manera que només els hem de definir una vegada (és el que s'anomena *classe*) i a crear noves classes d'objectes indicant només aquelles característiques que els siguin específiques.

La manera habitual de reutilitzar classes en orientació a objectes, però, no és reutilitzant classes aïllades sinó creant biblioteques de classes especialitzades que després els desenvolupadors poden fer servir sense necessitat de conèixer com estan implementades.

Exemples de reutilització mitjançant orientació a objectes

Un dels motius de l'exit del llenguatge Java en entorns empresarials és l'extensa biblioteca de classes que incorpora el llenguatge. El programadors Java, per exemple, poden reutilitzar classes que implementen conceptes matemàtics, estructures de dades, dates amb diversos sistemes de calendari o algorismes de criptografia, per la qual cosa es poden centrar a desenvolupar allò que és específic de la seva aplicació.

Un altre gran exemple de reutilització mitjançant orientació a objectes és la biblioteca de classes Standard Template Library. Es tracta d'una biblioteca de classes per al llenguatge C++, estandarditzada l'any 1994, que facilita el treball amb col·leccions d'objectes.

4.1.2. Bastiments

Un bastiment és un conjunt de classes predefinides que hem d'especialitzar per a implementar una aplicació o subsistema. De la mateixa manera que les biblioteques de classes, el bastiment ens ofereix una funcionalitat genèrica ja implementada i provada que ens permet centrar-nos en allò que és específic de la nostra aplicació.

A diferència, però, de les biblioteques de classes, un bastiment defineix el disseny de l'aplicació que el farà servir. És a dir, no solament estem reutilitzant la implementació de les classes sinó que, de manera indirecta, també estem reutilitzant el disseny del sistema. Típicament, són les classes del bastiment les que cridaran les nostres classes i no a la inversa. Per tant, el nivell de dependència del nostre sistema envers el bastiment és més alt que en el cas de fer servir una biblioteca de classes.

En canvi, un bon bastiment ens ofereix un nivell de reutilització més alt i, habitualment, ens facilita el seguiment d'una sèrie de pràctiques i principis ja provats, amb la qual cosa millora significativament la qualitat de la solució final.

Exemple de bastiment

Un exemple de bastiment serien els bastiments per al desenvolupament d'aplicacions web com ara Ruby on Rails. Aquest bastiment s'encarrega d'un munt de tasques de baix nivell per tal que, quan un usuari visita una determinada pàgina web, el bastiment acabi cridant un cert codi per a generar la pàgina web de manera dinàmica. El bastiment no solament ens permet reutilitzar classes sinó que ens ajuda a reutilitzar el disseny sencer del sistema final. D'altra banda, el programador no solament fa servir les classes del bastiment, sinó que les classes del bastiment fan crides a les del programador a l'hora de dur a terme tasques complexes.

4.1.3. Components

Les tècniques vistes fins ara només ens permeten reutilitzar classes individuals o, en tot cas, agrupacions de classes en forma de biblioteques i bastiments. Sovint, però, volem aprofitar més funcionalitat de la que ens dóna una sola classe. Per això va sorgir la programació orientada a components: amb la programació orientada a components, la unitat de reutilització (el component) pot estar formada per una o més classes.

A diferència d'una biblioteca de classes, el conjunt de classes que forma un component es reutilitza com un tot; no podem reutilitzar, per tant, una de les classes del component de manera individual.

Un dels avantatges dels components és que estan desenvolupats amb la intenció de ser reutilitzables i, per tant, acostumen a ser més estrictes quant a l'ocultació d'informació: cada component representa una unitat amb una interfície i un contracte que indica com es pot utilitzar, però que no dóna cap detall sobre com està desenvolupat per dintre.

interfícies gràfiques d'usuari

Un cas d'èxit dels components és el de les interfícies gràfiques d'usuari. En entorns com .Net i Java, no solament tenim disponibles un gran nombre de component de sèrie sinó que hi ha un gran nombre de companyies que es dediquen al desenvolupament de components d'interfície gràfica d'usuari compatibles amb la plataforma. D'aquesta manera si, per exemple, volem crear un gràfic avançat a partir d'unes dades i els components de sèrie no són prou sofisticats, podem comprar un component de gràfics a un tercer i estalviar-nos-en el desenvolupament.

Per a garantir l'èxit d'un sistema basat en components és important definir clarament el component: quina és la seva interfície (quines operacions es poden invocar i amb quins paràmetres) i quin és el seu contracte. Aquest ens indica quins són els efectes d'invocar una operació (també anomenats *postcondicions*), i quines són les condicions que cal verificar per a evitar que es produeixi un error (també anomenades *precondicions*).

En alguns sistemes, la interfície es defineix en un llenguatge específic per a les interfícies (és el que s'anomena *IDL*[8]), de manera que es facilita la utilització del component encara que estiguem implementant el nostre sistema en un llenguatge de programació diferent del llenguatge en què es va programar el component.

(8)*IDL* són les sigles d'*interface definition language*.

El desenvolupament de components reutilitzables és una mica diferent del desenvolupament de components no reutilitzables en el sentit que tenim molt menys control sobre les condicions en què s'utilitzarà el component i, per tant, és més important assegurar-ne la qualitat: per una banda, el component ha de suportar el mal ús sense que aquest tingui conseqüències greus i, per l'altra, hem d'evitar al màxim els errors, ja que es contagiarien als sistemes en què es facin servir.

4.1.4. Desenvolupament orientat a serveis

L'arquitectura orientada a serveis té com a objectiu incrementar el nivell de granularitat de la unitat d'abstracció: en comptes de reutilitzar una part del codi del sistema que volem desenvolupar, el que proposen les arquitectures orientades a serveis és reutilitzar el servei complet.

Una aplicació amb arquitectura orientada a serveis, per tant, estarà formada per una sèrie de serveis, que són unitats aïllades entre si que col·laboren a través d'una xarxa (i, per tant, habitualment, de manera remota) per a implementar la funcionalitat del sistema.

A diferència dels components, que formaran part del nostre sistema i que, per tant, es desplegaran juntament amb aquest, els serveis tenen el seu cicle de vida independent propi i, per tant, es desenvolupen i es despleguen de manera separada dels seus clients.

Pagament amb targeta de crèdit

Moltes entitats bancàries ofereixen, per exemple, la possibilitat de fer pagament amb targeta de crèdit des de les aplicacions desenvolupades per terceres organitzacions. De vegades, aquesta possibilitat s'ofereix com un component (que cal incloure al programari desenvolupat per a poder fer el pagament), però tot sovint es tracta d'un servei. En aquest cas, el servei s'executa en un dels ordinadors de l'entitat bancària i aquesta s'encarrega del seu cicle de vida (desenvolupar i desplegar noves versions per a corregir errors o afegir funcionalitat); el programari que el fa servir, en lloc d'incloure un component que pot fer el pagament, fa crides remotes al servei de l'entitat bancària per a demanar les operacions necessàries per a fer aquest mateix pagament.

4.1.5. Patrons

Fins ara hem vist maneres de reutilitzar la implementació. El problema de les tècniques de reutilització de la implementació és que, moltes vegades, tot i haver-hi certs paral·lelismes entre les diferents situacions, no podem aplicar exactament la mateixa solució a dos contextos diferents.

Per exemple, si estiguéssim desenvolupant una aplicació amb una base de dades remota, ens trobaríem amb el problema que la connexió a la base de dades pot estar oberta o tancada i que, per tant, hem de fer que la connexió es comporti de manera diferent segons estigui oberta o tancada. D'altra banda, a la interfície gràfica, ens trobem que certs elements poden estar actius o inactius i que, de nou, es comportaran de manera diferent segons estiguin o no actius.

En aquests dos casos, tot i haver-hi un cert paral·lelisme (un element del programa es comporta de manera diferent segons quin sigui el seu estat), no podem crear una classe genèrica que puguem reutilitzar en tots dos casos, com tampoc no podem crear un component que ens implementi aquest comportament.

Necessitem, doncs, eines de reutilització que ens permetin reutilitzar idees de manera independent de la implementació: és el que fan els patrons.

L'ús de patrons es va originar al món de l'arquitectura i l'urbanisme al final dels anys setanta. Des d'aleshores molts autors han treballat en la manera de traslladar aquesta pràctica al desenvolupament del programari, però va ser a mitjan anys noranta que es va publicar el llibre que va popularitzar definitivament l'ús de patrons: *Design Patterns: Elements of Reusable Object-Oriented Software*, per part d'Erich Gamma, Richard Helm, Ralph Johnson i John Vlissides.

> Erich Gamma, Richard Helm, Ralph Johnson i John Vlissides defineixen un patró de disseny de la manera següent: un patró de disseny dóna nom, motiva i explica de manera sistemàtica un disseny general que permet solucionar un problema recurrent de disseny en sistemes orientats a objectes. El patró descriu el problema, la solució, quan s'ha d'aplicar la solució i també les seves conseqüències. També dóna algunes pistes sobre com cal implementar-lo i exemples [...]. La solució s'ha de personalitzar i implementar per tal de solucionar el problema en un context determinat.

A partir d'aquí, l'ús de patrons es va anar estenent cap a altres activitats com ara l'anàlisi o l'arquitectura del programari, ja que són una eina molt útil per a reutilitzar idees en aquests contextos. Els principals avantatges dels patrons són:

- Reutilitzar les solucions i aprofitar l'experiència prèvia d'altres persones que han dedicat més esforç a entendre els contextos, les solucions i les conseqüències del que nosaltres volem o podem dedicar.
- Beneficiar-nos del coneixement i l'experiència d'aquestes persones mitjançant un enfocament metòdic.
- Comunicar i transmetre la nostra experiència a altres persones (si en definim de nous).
- Establir un vocabulari comú per a millorar i agilitzar la comunicació.
- Encapsular coneixement detallat sobre un tipus de problema i les seves solucions assignant-li un nom per tal que en puguem fer referència fàcilment.
- No haver de reinventar una solució al problema.

Per a aconseguir aquests beneficis, haurem de documentar, com a mínim els elements del patró següents:

- El nom ens permet fer referència al patró quan documentem el nostre disseny o quan el comentem amb altres desenvolupadors. També ens permet augmentar el nivell d'abstracció del procés de disseny, ja que podem pensar en la solució al problema com un tot.
- El problema ens indica què resol el patró. Aquest problema podria ser un problema concret o la identificació d'una estructura problemàtica (per exemple, una estructura de classes poc flexible).

- La solució descriu els elements que formen part del patró, i també les seves relacions, responsabilitats i col·laboracions. La solució no és una estructura concreta sinó que, com hem indicat anteriorment, és com una plantilla que podem aplicar al nostre problema. Alguns patrons presenten diverses variants d'una mateixa solució.

- Les conseqüències són els resultats i els compromisos derivats de l'aplicació del patró. És molt important que les tinguem en compte a l'hora d'avaluar el nostre disseny, ja que és el que ens permet entendre realment el cost i els beneficis de l'aplicació del patró.

Patró *estat*

Per exemple, el patró *estat* ens pot ajudar a solucionar el problema esmentat anteriorment de classes que es comporten de manera diferent en funció del seu estat. La definició (simplificada) que donen Gamma, Helm, Johnson i Vlissides (1994) del patró *estat* és la següent:

- **Nom**: estat (també conegut com a *objectes per estats*).
- **Problema**: el comportament d'un objecte depèn del seu estat i ha de canviar el seu comportament en temps d'execució en funció d'aquest. Les operacions tenen sentències condicionals llargues que depenen de l'estat de l'objecte. Sovint diverses operacions tenen les mateixes estructures condicionals (és a dir, preveuen les mateixes condicions, ja que depenen dels possibles estats de l'objecte).
- **Solució**: introduir una nova classe que representa l'estat amb una subclasse per cada estat que tingui un comportament diferent. Cada branca de les estructures condicionals es correspondrà ara a una de les subclasses de l'estat. Això ens permet tractar l'estat de l'objecte com un objecte en si mateix que pot canviar independentment de la resta d'objectes.
- **Conseqüències**: localitza el comportament específic de l'estat, particiona el comportament dels diferents estats i fa explícites les transicions entre estats.

Observació

Encara no hem explicat alguns dels conceptes que s'utilitzen en la descripció d'un patró (tingueu en compte que es tracta d'un patró de disseny de programari), però això no ens ha de preocupar, ja que, ara mateix, el que ens interessa és veure quina estructura té un patró i fer-nos una idea aproximada de com el faríem servir.

4.1.6. Línies de producte

Les línies de producte són, en origen, una estratègia de màrqueting que consisteix a oferir, de manera individual, una sèrie de productes relacionats entre si. La línia de producte permet mantenir una identitat base comuna a tots els productes de la línia i, al mateix temps, oferir un producte adaptat als gustos del consumidor.

Un exemple típic de línies de producte és el cas de l'automoció, en què podem triar, per a un determinat model, diferents nivells d'acabats de l'interior, de color, motor, carrosseria, etc.

En indústries com la de l'automoció, un dels avantatges de les línies de producte és que faciliten la reutilització de components comuns entre els diferents productes de la mateixa línia, raó per la qual no han mancat esforços per traslladar aquest concepte al món del programari.

Línies de productes de programari

De fet, podem considerar que, actualment, hi ha moltes línies de productes de programari. Per exemple, de la distribució de Linux Ubuntu hi ha, el 2010, tres edicions oficials

(Desktop, Server i Netbook) i infinitat de derivades, totes basades en el nucli de la distribució Debian, però configurant paquets de programari diferents (entorn d'escriptori, navegador, utilitats, etc.) per tal de personalitzar el producte segons el mercat a què es dirigeixi la distribució.

Des d'un punt de vista més relacionat amb el desenvolupament del programari, hi ha tècniques i tecnologies específiques que faciliten la creació de línies de productes de programari, tot i que el seu estudi queda fora de l'àmbit d'aquests materials.

4.2. Tècniques basades en l'abstracció

Un altre dels reptes de l'enginyeria del programari és facilitar la creació d'abstraccions que permetin desenvolupar sistemes fent servir llenguatges més propers als llenguatges naturals que no pas als llenguatges de les computadores.

De fet, la majoria de les tècniques de reutilització que acabem de veure es basen en la creació d'abstraccions. També hem vist com, al llarg del temps, s'han anat creant llenguatges cada vegada més rics per a comunicar-nos amb les computadores: des dels llenguatges assemblador i el codi màquina fins als llenguatges visuals de modelització.

Aquests nous llenguatges tenen en comú la característica que sempre sabem traduir-los a altres llenguatges "de més baix nivell" i així successivament fins a arribar al llenguatge màquina, ja sigui mitjançant un compilador (que és un programa que genera un programa en un llenguatge destinació a partir d'un programa en un llenguatge origen) o un intèrpret (que és un programa que llegeix, interpreta i executa un programa escrit en un cert llenguatge).

Tecnologia JSP

Per exemple, la tecnologia de pàgines de servidor JSP ens permet escriure un programa en un llenguatge similar a l'HTML (el llenguatge JSP), que llavors serà traduït al llenguatge Java, des del qual serà traduït a a un llenguatge anomenat *Java Bytecode*, que serà interpretat i executat. Totes aquestes traduccions tenen lloc de manera totalment automatitzada i transparents al programador de la pàgina JSP.

Anomenem *enginyeria dirigida per models*[9] un conjunt de tècniques basades en l'abstracció que veuen el desenvolupament de programari com una activitat en què, principalment, es desenvolupen models (en el llenguatge que sigui) que són l'artefacte principal amb independència de l'existència o no de codi font (sigui generat automàticament o manualment).

(9) En angles, *model-driven engineering* (MDE).

A continuació veurem dues tècniques d'enginyeria dirigida per models: l'arquitectura dirigida per models i els llenguatges específics del domini.

4.2.1. Arquitectura dirigida per models

El terme *arquitectura dirigida per models*[10] fa referència a un estàndard publicat per l'OMG que té com a objectiu separar la lògica de negoci i d'aplicació (és a dir, la funcionalitat del sistema per desenvolupar) de la tecnologia en què s'implementarà el sistema.

(10)En anglès, *model driven architecture* (MDA).

Enllaç d'interès

Pàgina oficial de l'OMG sobre MDA:
http://www.omg.org/mda

En MDA, un model és una descripció o especificació d'un sistema i del seu entorn per a un cert propòsit. Típicament el model es presentarà com una combinació de notació gràfica i textual. El text podria estar en un llenguatge de modelització o en llenguatge natural.

La filosofia d'MDA consisteix a definir, a més dels models, una sèrie de regles de transformació entre models que ens permetin generar, de la manera més automatitzada possible, programes a partir dels models, estalviant-nos així la fase d'implementació.

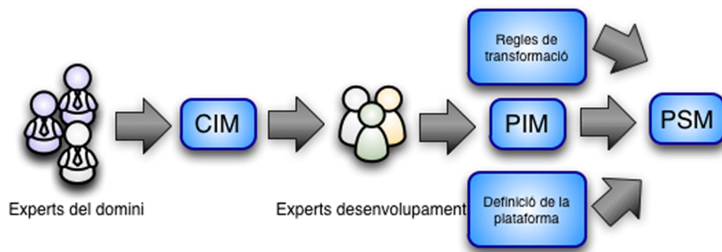
El punt de partida en un desenvolupament MDA és un model que anomenem *model independent de la computació*[11], que és el que serveix per a comunicar els requisits dels experts en el domini als experts en disseny i construcció de programari (els desenvolupadors). Aquest model també es coneix com a model del domini o model del negoci.

(11)En anglès, *computation independent model* (CIM).

El CIM no té en compte la tecnologia amb la qual s'implementarà el sistema i, per tant, no és prou detallat per a ser executat directament. Per això el CIM ha de ser transformat en un segon model, més proper a la tecnologia, anomenat *platform independent model* (PIM).

El PIM, tot i ser més proper a la tecnologia, és independent de la plataforma final d'execució en el sentit que un mateix PIM pot donar lloc a diferents implementacions sobre diferents plataformes. Per exemple, un mateix PIM podria donar lloc a una implementació per a la plataforma Java o a una implementació per a la plataforma Microsoft .Net. D'aquesta manera aconseguim un dels objectius d'MDA: separar l'evolució dels models de l'evolució de la tecnologia.

Acompanyant el PIM tindrem un conjunt de regles de transformació envers diferents plataformes. Aquestes regles ens serviran per a aconseguir l'anomenat *platform specific model* (PSM), que és aquell que pot ser compilat i executat directament sobre la plataforma.



MDA, però, també preveu altres possibilitats, com ara generar directament codi executable a partir del PIM o generar un segon PSM a partir d'un PSM inicial per a afegir-hi nous detalls d'implementació. El que és important, doncs, és el procés de refinament successiu de models.

4.2.2. Llenguatges específics del domini

Mentre que MDA està històricament lligat a l'ús de models gràfics, també podríem crear els nostres models mitjançant llenguatges de caire més textual, que anomenem *llenguatges específics del domini* (DSL, *domain-specific languages*).

Lectura recomanada

Martin Fowler. Language Workbenches: The Killer-App for Domain Specific Languages?

El desenvolupament basat en DSL consisteix a crear llenguatges nous adaptats al domini del sistema que volem desenvolupar de manera que estiguin optimitzats per a expressar els requisits del sistema que estem desenvolupant.

A diferència, doncs, del llenguatge utilitzat en MDA (típicament, UML), que és un llenguatge de propòsit general, els llenguatges específics del domini només serveixen per a solucionar un problema molt concret (entendre un format d'intercanvi de dades, descriure un procés de negoci, etc.). Això facilita que siguin molt senzills però, al mateix temps, obliga a definir-ne més d'un per tal de cobrir la funcionalitat completa d'un sistema.

4.3. Eines de suport a l'enginyeria del programari (CASE)

L'enginyeria del programari, com qualsevol altra activitat humana, requereix unes eines que ens ajudin amb les diferents tasques. Quan aquestes eines consisteixen en aplicacions informàtiques, es coneixen com a eines CASE[12].

(12) *CASE* són les sigles de *computer aided software engineering.*

La finalitat de les eines CASE és ajudar l'enginyer de programari a aplicar els principis de l'enginyeria al desenvolupament de programari. Així, no totes les eines que es fan servir durant el desenvolupament del programari entrarien en aquesta categoria. Per exemple, les eines de programació no es consideren, habitualment, eines CASE, però les eines de modelització si.

El principal avantatge d'aquest tipus d'eines és que faciliten l'automatització d'algunes de les tasques de l'enginyer de programari, i també el tractament informàtic dels productes del seu treball. Per exemple, una eina de modelització ens pot ajudar a mantenir consistents tots els models mitjançant una sèrie de validacions i comprovacions automatitzades.

Eines de modelització optimitzades per a fer diagrames UML

Per a fer un diagrama UML podríem fer servir qualsevol eina de dibuix, però el fet que les eines de modelització estiguin optimitzades per a fer diagrames UML fa que sigui molt més còmode i senzill fer els diagrames amb aquestes eines que no pas amb els programes clàssics de dibuix. Aquestes eines, a més, poden fer validacions sobre quins "dibuixos" són models vàlids i quins no tenen sentit.

Una eina CASE pot ser quelcom tan senzill com una eina que ens ajudi a dibuixar els diagrames o bé quelcom tan complicat com una família d'eines que ens ajudin a seguir estrictament un procés determinat de desenvolupament. En tots dos casos, és important que les diferents eines es comuniquin entre si.

Aquestes eines ens ajuden a gestionar el projecte ja que, com hem dit anteriorment, ens poden ajudar a implementar un procés de desenvolupament i, per tant, a fer el seguiment de les diferents activitats i tasques.

Dins de l'àmbit més específic de l'enginyeria del programari, les eines CASE ens poden ajudar molt en la gestió dels requisits. Les eines de gestió dels requisits ens ajuden a mantenir un repositori centralitzat de requisits i també a consultar la història d'un requisit des de la petició de l'*stakeholder* fins a la implementació en codi (segons l'eina).

Una altra activitat que es pot beneficiar molt de l'ús d'una eina CASE és, com hem dit anteriorment, la modelització en particular i l'anàlisi en general. Les eines CASE ens poden ajudar, fins i tot, a transformar els models en codi executable (com és el cas del desenvolupament basat en models) o a generar models automàticament a partir de codi (és el que s'anomena *enginyeria inversa*). En una situació ideal, l'eina CASE hauria de poder actualitzar el codi a partir dels models i viceversa (és el que s'anomena *round-trip engineering*).

Finalment, la gestió de la qualitat també es pot veure molt beneficiada per l'ús d'eines que ens generin mètriques de manera automatitzada o ens ajudin a gestionar els errors detectats i a analitzar-ne les causes per tal de trobar processos de millora que ens ajudin a reduir el volum d'errors (per exemple, si detectem que el 30% d'errors s'introdueixen durant la construcció del sistema, sabem que hem de millorar les proves que fem d'aquesta activitat).

Les eines CASE, finalment, també ens poden ajudar en les activitats de manteniment i reenginyeria:

- **Eines de gestió del procés**. Ajuden a la definició, modelització, execució o gestió del procés de desenvolupament mateix.

- **Eines de gestió de projectes**. Eines que ajuden en tasques de gestió del projecte com ara l'estimació, la planificació o l'anàlisi del risc.

- **Eines de gestió de requisits**. Eines de suport per a la recollida, documentació, gestió i validació de requisits. Poden ser genèriques, en què els requisits es documenten textualment, o específiques, com ara les basades en històries d'usuari o en casos d'ús.

- **Eines de modelització**. Eines que faciliten la creació de models. En el cas específic d'UML, permeten la creació de diagrames UML tot validant en més o menys grau la correctesa de la sintaxi utilitzada.

- **Eines d'enginyeria inversa**. Es tracta d'eines que permeten analitzar un programari existent per a poder començar a aplicar els principis d'enginyeria del programari; típicament elaboren models d'anàlisi i disseny, per exemple, en UML, a partir del codi, d'una base de dades existent, etc.

- **Entorn integrat de desenvolupament**. Eines per a la codificació del programari, però també per al disseny, l'execució de proves i la depuració.

- **Eines de construcció del programari**. Eines que construeixen l'executable final a partir dels diversos arxius de codi font. Fan la compilació dels arxius, però també l'empaquetament en el format adequat per a l'entrega.

- **Eines de proves**. Eines per a la definició d'una estratègia de proves i per al seguiment al llarg del projecte. Poden ajudar en la definició de la prova, l'execució, la gestió de les proves i dels resultats obtinguts, etc.

- **Eines de desenvolupament d'interfícies gràfiques d'usuari**. Són eines que permeten desenvolupar les interfícies gràfiques d'usuari de manera gràfica, de tal manera que, en lloc d'haver de programar el 100% de la interfície, el desenvolupador pot arrossegar components a les pantalles de manera visual i programar-ne només el comportament dinàmic.

- **Eines de mesura de mètriques**. Eines que permeten mesurar una sèrie de mètriques de manera automatitzada per a donar-nos criteris objectius a partir dels quals puguem valorar la qualitat del programari desenvolupat pel que fa a arquitectura, disseny, codi, proves, etc.

- **Eines de gestió de la configuració i del canvi**. Eines que gestionen el canvi del programari al llarg de la seva vida. Poden gestionar les diverses versions que es van creant de cada artefacte, les diverses versions del pro-

ducte final, la integració de les noves versions de cada part del producte final, etc.

Perquè l'entorn de treball de l'enginyer de programari sigui el més adequat possible, és necessari que les eines que fa servir estiguin correctament integrades i es coordinin correctament entre si.

Entorns integrats de desenvolupament

Com el seu nom indica, els entorns integrats de desenvolupament solen integrar i aglutinar tot un conjunt d'eines que resultin útils per al disseny i codificació de programari. Solen incorporar, per exemple, eines de depuració, d'execució de proves, de mesura de mètriques i eines de desenvolupament de la interfície gràfica d'usuari. D'altra banda, solen coordinar la feina amb les eines de gestió del canvi i de la configuració, i permeten a l'usuari fer servir el control de versions de manera integrada.

Un altre cas típic d'integració és el de les eines de modelització. Aquestes eines solen incorporar eines de mesura de mètriques i eines d'enginyeria inversa. Algunes, a més, s'integren amb l'entorn integrat de desenvolupament de tal manera que faciliten el pas de model a codi o, fins i tot, mantenen tots dos (codi i models) actualitzats de tal manera que un canvi al codi es reflecteixi en els models i viceversa.

Encara un altre exemple: un entorn d'integració contínua és un entorn que integra una eina de control de les versions, eines de proves i eines de construcció de programari perquè, cada cop que es desenvolupa una nova versió d'un arxiu de codi font, es construeixi l'entregable final i s'executin proves de manera automatitzada.

5. Estàndards de l'enginyeria del programari

L'enginyeria del programari, com a disciplina de l'enginyeria que és, requereix l'existència d'estàndards àmpliament acceptats que facilitin la comunicació i la interacció entre enginyers.

De vegades, però, els esforços d'estandardització es troben amb problemes a causa de la relativa joventut de la nostra disciplina i a la rapidesa amb què avança la tecnologia i, per tant, el domini d'aplicació de l'enginyeria del programari.

A causa d'això, la majoria d'estàndards en l'enginyeria del programari neixen rodejats de controvèrsia, tot i que, a mesura que avança el temps, es va reconeixent el seu valor.

Un dels organismes més actius en la definició d'estàndards per a l'enginyeria del programari és l'IEEE; més concretament, la IEEE Computer Society. A l'apèndix C de la guia SWEBOK (2004) podem trobar una llista amb més de 30 estàndards publicats per l'IEEE relacionats amb l'enginyeria del programari.

Un altre organisme molt actiu en aquesta àrea és l'Object Management Group (OMG), que és un consorci d'empreses dedicat a la creació d'estàndards en tecnologia orientada a objectes, com per exemple UML o CORBA.

A continuació, fem un recull dels estàndards que hem considerat més importants. Aquesta llista no pretén ser exhaustiva sinó, simplement, indicativa de la varietat d'estàndards i organismes que hi ha dedicats a l'enginyeria del programari.

5.1. Llenguatge unificat de modelització (UML)

El llenguatge unificat de modelització (UML) és un estàndard publicat per l'OMG que defineix la notació acceptada universalment per a la creació de models del programari. Gràcies a aquest estàndard, qualsevol enginyer de programari pot entendre els models creats per un altre enginyer.

Vegeu també

Estudiarem el llenguatge UML amb més detall al mòdul "Anàlisi UML".

La finalitat principal del llenguatge UML (i un dels motius del seu èxit) és unificar la notació gràfica que fan servir els diferents mètodes de desenvolupament amb independència del mètode emprat. D'aquesta manera, encara que un enginyer no conegui el mètode de desenvolupament amb què s'ha arribat a crear un model, és perfectament capaç d'entendre'n el significat.

Històricament, UML es va desenvolupar dins de l'empresa Rational, creadors de la família de mètodes *unified process*, tot i que, actualment, l'OMG és qui controla totalment l'evolució del llenguatge.

Una característica interessant de l'UML és que (en part a causa de la seva independència del mètode) no indica quins models o artefactes es poden crear, sinó que ofereix una sèrie de diagrames que els diferents mètodes poden fer servir per als seus artefactes. Així doncs, un mateix tipus de diagrama es pot fer servir en diferents contextos per a generar artefactes diferents.

5.2. *Software engineering body of knowledge* (SWEBOK)

El *software engineering body of knowledge* (SWEBOK) és un intent de recollir quin és el coneixement àmpliament acceptat per la comunitat d'enginyers del programari.

Com a part d'aquesta iniciativa, es va publicar una guia que descriu quin és aquest cos de coneixement i on el podem trobar.

Els objectius de la guia de l'SWEBOK són:

- Promoure una visió consistent del que és l'enginyeria del programari a escala mundial.
- Clarificar la situació (i les fronteres) de l'enginyeria del programari envers altres disciplines com les ciències de la computació (*computer science*), la gestió de projectes, l'enginyeria de computadors (*computer engineering*) i les matemàtiques.
- Caracteritzar els continguts de la disciplina de l'enginyeria del programari.
- Proporcionar un accés organitzat al cos de coneixement de l'enginyeria del programari.

La guia de l'SWEBOK organitza el coneixement en 10 àrees clau:

1) requisits del programari

2) disseny del programari

3) construcció del programari

4) proves del programari

5) manteniment del programari

6) gestió de la configuració del programari

7) gestió de l'enginyeria del programari

8) procés de l'enginyeria del programari

9) eines i mètodes de l'enginyeria del programari

10) qualitat del programari

5.3. *Capability maturity model integration* (CMMI)

La *capability maturity model integration* (CMMI), publicada pel Software Engineering Institute, "es pot utilitzar per a guiar el procés de millora aplicat a un projecte, una divisió o l'organització sencera". Als enginyers del programari, doncs, CMMI ens proporciona una guia per a la millora dels processos implicats en el desenvolupament de programari.

CMMI preveu, principalment, tres àrees d'activitat:

- **Gestió de serveis (CMMI-SVC)**, destinat a proveïdors de serveis amb la finalitat d'ajudar-los a reduir costos, millorar la qualitat i la predictibilitat. Inclou bones pràctiques sobre quins serveis s'han d'oferir, assegurar-se que s'està en condicions d'oferir un servei, i establir acords i activitats relacionades, en general, amb la provisió de serveis

- **Adquisició (CMM-ACQ)**, enfocat a ajudar organitzacions que contracten proveïdors de productes o serveis a millorar les relacions amb els proveïdors, els processos de contractació i de control, i també l'adequació del producte o servei adquirit a les necessitats de l'organització.

- **Desenvolupament (CMMI-DEV)**, dirigit a organitzacions que desenvolupen programari i que vulguin millorar la satisfacció dels clients, la qualitat del producte obtingut i el procés mateix de desenvolupament.

Com a part del procés de millora, CMMI defineix una sèrie de nivells de maduresa, en funció del nivell d'implantació de les pràctiques suggerides.

5.4. ***Project management body of knowledge*** **(PMBOK)**

PMBOK (publicat pel Project Management Institute) és una guia de bones pràctiques per a la gestió de projectes. Com a tal, quedaria fora de l'àmbit de l'enginyeria del programari, tal com la defineix la guia SWEBOK (2004) però, tal com hem indicat al llarg del mòdul, la majoria de desenvolupaments de programari es fan en forma de projecte i, per tant, la gestió del projecte és clau per a l'èxit del desenvolupament del programari.

El PMBOK identifica les àrees de coneixement següents:

- gestió de la integració
- gestió de l'abast
- gestió del temps
- gestió de la qualitat
- gestió dels costos
- gestió del risc
- gestió de recursos humans
- gestió de la comunicació
- gestió de les compres i adquisicions

Resum

Hem vist que l'enginyeria del programari és l'aplicació d'un enfocament sistemàtic, disciplinat i quantificable al desenvolupament, operació i manteniment de programari. Aquest enfocament es va adoptar amb la finalitat d'assolir els nivells de qualitat, productivitat i control de costos de la resta d'enginyeries.

El programari, però, a diferència d'altres productes, és intangible i, per tant, la seva producció no consumeix matèries primeres. Per altra banda, no es manufactura (totes les còpies són idèntiques i el cost de crear-ne una és pràcticament nul) ni es desgasta i, a més, queda obsolet ràpidament.

L'aplicació de l'enginyeria del programari implica que hem de seguir un mètode, que descriurà les característiques del procés disciplinat que utilitzarem. Més concretament, ens dirà quines tasques s'han de dur a terme, qui les ha de dur a terme (els rols) i quins artefactes seran les entrades i resultats de cada tasca.

D'aquests mètodes de desenvolupament, hem vist que és important escollir el més adequat a la naturalesa del projecte que s'hagi de dur a terme i n'hem presentat tres grans famílies –els mètodes que segueixen el cicle de vida en cascada, els mètodes iteratius i incrementals i els mètodes àgils– i hem vist un exemple representatiu de cadascuna –Mètrica 3, el procés unificat i Scrum.

Les tasques definides per un mètode poden pertànyer a àmbits diversos com ara la gestió del projecte, la identificació i gestió de requisits, la modelització, la construcció del programari, les proves, la gestió de la qualitat, el manteniment o la reenginyeria.

Pel que fa als rols, cada mètode pot definir rols diferents, tot i que hi ha certs rols que acostumen a aparèixer amb un o altre nom en la majoria de mètodes. És el cas del cap de projectes, l'expert del domini, l'analista funcional, l'arquitecte, l'analista orgànic, el programador, l'expert de qualitat, l'encarregat de desplegament o el responsable de producte.

També hem vist un resum d'algunes tècniques i eines pròpies de l'enginyeria del programari: l'orientació a objectes, els bastiments, el desenvolupament basat en components, el desenvolupament orientat a serveis, els patrons, les línies de producte, l'arquitectura dirigida per models i els llenguatges específics del domini i també alguns estàndards relacionats amb les activitats esmentades anteriorment.

Activitats

1. Doneu dos exemples de cadascun dels tipus de programari mencionats al subapartat 1.3.

2. Donats els exemples següents de programari, indiqueu, per a cadascun, a quins dels tipus de programari indicats al subapartat 1.3 pertany:

a) El nucli de l'SO Linux
b) Un gestor de finestres en un entorn d'escriptori (per exemple, MetaCity)
c) Un paquet ofimàtic com, per exemple, OpenOffice
d) El programari de planificació de recursos empresarials SAP
e) Un sistema de gestió del correu electrònic via Web (com, per exemple, Hotmail)
f) L'aplicació de campus virtual de la UOC
g) El programari Mathematica
h) AutoCAD
i) El programari que gestiona el GPS d'un cotxe
j) El programari de reconeixement de la parla

3. Doneu dos arguments a favor i dos arguments en contra del fet de tractar el desenvolupament del programari com una disciplina de l'enginyeria.

4. Trobeu tres punts en comú i tres diferències entre l'enginyeria del programari i l'alpinisme.

5. A quin tipus d'activitat del subapartat 2.3 pertanyerien les tasques següents:

a) Estudiar el producte ofert per la companyia *X* per veure si satisfà les nostres necessitats o si, al contrari, haurem de buscar un altre producte o fer un desenvolupament a mesura.
b) Determinar si podem acabar el projecte en una data concreta si la Maria deixa el projecte.
c) Establir el calendari de reunions de seguiment del projecte.
d) Estudiar com ens afecta la LOPD.
e) Crear un model de la interfície d'usuari per a mostrar als usuaris finals com serà l'aplicació.
f) Crear un diagrama UML amb el model conceptual del domini.
g) Determinar i documentar quin volum màxim d'usuaris ha de suportar el sistema per desenvolupar.
h) Detectar quines són les necessitats dels administradors del sistema.
i) Fer un prototip de la interfície gràfica d'usuari per tal de provar diferents combinacions de colors i diferents disposicions de la informació.
j) Dissenyar un component del sistema.
k) Comprovar que el sistema suporta el volum màxim d'usuaris que es va determinar.
l) Verificar que s'ha seguit el procés de desenvolupament tal com està definit.
m) Recollir informació sobre el nombre de requisits implementats cada setmana.
n) Corregir un error en un sistema ja en producció.

6. Classifiqueu, segons la classificació vista a l'apartat 3.2, els projectes següents:

a) Desenvolupar una aplicació per a substituir el full de càlcul on apuntem les vacances.
b) Desenvolupar un sistema de comerç electrònic que permeti incrementar les vendes en línia.
c) Avaluar l'OpenOffice com a alternativa a les eines ofimàtiques que s'estan fent servir actualment a la nostra organització.
d) Identificar quins dels nostres sistemes actuals podem substituir per aplicacions de programari lliure.
e) Desenvolupar una aplicació que incorpori dades d'uns fitxers en un format conegut i documentat i les desi en una base de dades existent.
f) Desenvolupar una aplicació per a automatitzar el procés de compra de mercaderies a proveïdors en una organització on, actualment, tot el procés és manual.

7. Suposeu que estem treballant en un petit projecte amb els requisits (per ordre de prioritat) i estimacions següents:

Requisit	Anàlisi i disseny	Implementació	Proves
a	1	1	1
b	3	5	2
c	2	4	1
d	1	3	1

Requisit	Anàlisi i disseny	Implementació	Proves
e	2	1	2
f	1	3	2
g	1	1	1

a) Planifiqueu el projecte suposant que l'estimació està en dies de feina (per exemple, el requisit *a* necessita un dia d'anàlisi i disseny, un dia d'implementació i un dia de proves) fent servir el cicle de vida en cascada, indicant quina feina es farà cada setmana.
b) Feu el mateix amb el cicle de vida iteratiu i incremental suposant iteracions de 10 dies.

8. Indiqueu, per a cadascuna de les situacions següents, si creieu que compleixen o no els principis del manifest àgil:

a) Un cop aclarida la funcionalitat en una conversa amb el client, el cost de documentar-la i implementar-la és més o menys el doble que el d'implementar-la directament i decidim no documentar-la.
b) El client ha de signar el document de requisits abans de començar el desenvolupament per tal d'assegurar-nos que l'ha entès bé i que no caldrà introduir canvis.
c) Els desenvolupadors han d'omplir un informe al final del dia on indiquin, de les seves hores de feina, quantes han dedicat a cadascuna de les tasques.
d) El cap de projectes no assigna tasques als desenvolupadors sinó que aquests es presenten voluntaris.
e) El client ha d'estar disponible en tot moment per a resoldre dubtes als desenvolupadors (i no solament al cap de projecte).
f) S'ha planificat en detall i s'ha documentat en un diagrama de Gantt tota la feina dels propers sis mesos.
g) Hi ha un procés de gestió del canvi que permet introduir canvis als requisits un cop iniciat el desenvolupament.
h) Totes les persones implicades en el desenvolupament del projecte treballen a la mateixa sala.
i) Tota la comunicació entre desenvolupadors s'ha de fer per mitjà d'un fòrum perquè quedi registrada i es pugui reutilitzar més endavant.
j) A l'hora de mesurar l'avanç del projecte no es té en compte la funcionalitat que està a mig implementar. Per exemple, si tenim l'especificació i l'anàlisi d'una funcionalitat però no l'hem acabada de dissenyar i implementar, considerem que l'avanç és 0 i no 50%. Per tant, considerem que l'avanç d'una funcionalitat és SÍ/NO i no un percentatge.

9. Mètrica 3 és un mètode que es pot aplicar tant al desenvolupament estructurat com al desenvolupament orientat a objectes. Indica un artefacte que sigui específic de cadascun dels casos indicats i un que sigui comú a tots dos casos.

10. Indica quines són les tasques específiques de l'analista en OpenUP.

11. A l'article "Metrics you can bet on", de Mike Cohn, l'autor proposa tres propietats que haurien de complir totes les mètriques. Quines són aquestes propietats? Penseu en una mètrica que hàgiu fet servir (o que podríeu fer servir en un projecte) i indiqueu si compleix o no les propietats esmentades.

12. El mètode Scrum ens proposa dos artefactes amb nom similar: el *product backlog* i l'*sprint backlog*. Indiqueu quina és la finalitat de cadascun i enumereu i justifiqueu tres diferències entre tots dos artefactes.

13. Penseu un altre exemple de separació entre interfície i implementació al món real.

14. Busqueu un exemple de patró d'anàlisi. Com està documentat el patró? Penseu en un cas en què pugui ser d'aplicació.

15. Suposeu que estem desenvolupant un producte i que ens demanen un canvi en un dels requisits. Volem determinar l'impacte del canvi en la nostra planificació. Com ens afecta en el cas que estiguem seguint el cicle de vida en cascada? I en el cas del cicle de vida iteratiu i incremental? Canvia molt l'impacte segons si ja està implementat o no?

16. Partint de l'exemple següent de programa escrit en un llenguatge específic del domini digueu a quin domini creieu que s'aplica i quins avantatges i inconvenients hi trobeu:

```
class
   NewTest def eBay_auto_feedback_generator
```

```
      open "http://my.ebay.com/ws/eBayISAPI.dll?MyeBay"
      assertTitle "My eBay Summary"
      pause "2000"
      clickAndWait "link=Feedback"
      pause "3000"
      assertTitle "My eBay Account: feedback"
      clickAndWait "link=Leave Feedback"
      assertTitle "Feedback Forum: leave Feedback"
      click "//input[@name='which' and @value='positive']"
      type "comment", "Great eBayer AAAA++++"
      clickAndWait "//input[@value='Leave Feedback']"
      assertTitle "Feedback Forum: your feedback has been left"
   end
end
```

Exercicis d'autoavaluació

1. Què és el programari per a sistemes d'informació?

2. Quines són les característiques inherents al programari que el diferencien d'altres productes industrials?

3. Per què el desenvolupament de programari s'organitza en forma de projectes?

4. Quina relació hi ha entre les tasques, els rols i els artefactes en un procés de desenvolupament?

5. Indiqueu quatre tasques relacionades amb l'activitat de gestió del projecte.

6. Quina és la funció dels requisits en un projecte de desenvolupament?

7. Quina és la responsabilitat de l'analista funcional en un projecte de desenvolupament?

8. Quines són les etapes del cicle de vida en cascada?

9. Quins són els principals avantatges del cicle de vida iteratiu i incremental respecte al cicle de vida en cascada?

10. Què és i per a què serveix el pla de sistemes d'informació segons Mètrica 3?

11. Quines són les pràctiques fonamentals del procés unificat?

12. Quins són els artefactes de Scrum?

13. Indiqueu dos avantatges i un inconvenient de la reutilització de programari.

14. Què és l'ocultació d'informació (en el context del programari en general i de l'orientació a objectes en particular)?

15. Indiqueu dos avantatges de l'ús de patrons.

16. Què són les eines CASE?

17. Què és UML?

Solucionari

Activitats

1. Programari de sistemes: el nucli de Linux, el controlador de la targeta de xarxa.

Programari d'aplicació: OpenBravo (ERP), OpenOffice.

Programari científic / d'enginyeria: Mathematica, AutoCAD.

Programari encastat: el microprogramari d'una càmera de fotos, el programari que controla els frens ABS d'un cotxe.

Programari de línies de producte: segons la web Software Product Line Conferences el programari dels televisors Philips (que seria també un cas de programari encastat) es deriva tot d'una mateixa línia de productes. Un altre exemple de la mateixa web és el programari controlador dels productes RAID de l'empresa LSI Logic: en aquest cas ens trobem amb un exemple de programari de sistemes encastat i de línia de producte.

Aplicacions web: Facebook, Gmail.

Programari d'intel·ligència artificial: ELIZA, un programari desenvolupat als anys seixanta i que permetia mantenir converses amb els usuaris. D'altra banda, la majoria de videojocs d'estratègia inclouen un component d'intel·ligència artificial per tal de fer més realista el joc.

2.

a) Programari de sistemes
b) Programari de sistemes
c) Programari d'aplicació, línia de productes
d) Programari d'aplicació, línia de productes
e) Aplicació web
f) Aplicació web
g) Programari científic/d'enginyeria
h) Programari d'enginyeria
i) Programari encastat (també podria formar part d'una línia de productes)
j) Intel·ligència artificial

3. Arguments a favor:

- El principal argument a favor és el gran nombre de projectes de desenvolupament de programari que s'han dut a terme al llarg dels anys aplicant l'enginyeria al desenvolupament del programari i com s'ha millorat respecte a la situació anterior. Al subapartat 1.5, parlem del Chaos Report de l'any 1995 i de l'any 2010 i de com s'ha doblat en 15 anys el nombre de projectes que es consideren 100% reeixits.
- Un altre argument a favor és que l'enfocament sistemàtic (que, com acabem de dir, és viable) ens permet aplicar el coneixement de les altres enginyeries al desenvolupament del programari. És el cas, per exemple, de com els principis del mètode de producció de Toyota s'han traslladat al món del desenvolupament de programari en el que s'ha anomenat *desenvolupament lean*.

Arguments en contra:

- Per una banda, Tom DeMarcos ens diu, a l'article "Software Engineering: An Idea Whose Time Has Come and Gone?", publicat a la revista *IEEE Software* al número de juliol/agost de 2009 que, si bé l'enginyeria se centra en el control del procés, aquest no hauria de ser l'aspecte més important per considerar a l'hora de pensar en com desenvolupem el programari sinó que ens hauríem de centrar en com el programari transforma el món en què vivim o l'organització per al qual el desenvolupem.
- D'altra banda, un altre argument en contra de l'enfocament d'enginyeria és aquell que considera que l'impacte de les persones en el procés de desenvolupament de programari (la seva habilitat, com es relacionen i com es comuniquen, el seu estat d'ànim, la seva motivació, etc.) és tan gran que cap mètode, per ben definit que estigui, pot convertir el desenvolupament de programari en un procés repetible.

4. Aspectes en comú:

- S'acostumen a fer en equip.
- El nivell d'experiènica i l'habilitat natural dels participants impacta molt en el resultat final.
- S'ha d'aconseguir un objectiu amb un temps i uns recursos limitats.

Diferències:

- En l'alpinisme cada ascensió comença de zero, mentre que l'enginyeria del programari pot aprofitar parts de la feina que ja estiguin fetes d'altres projectes i, per tant, hem d'intentar tenir en compte els futurs desenvolupaments durant el projecte.
- L'alpinisme té un final clar i a curt termini: o bé el moment d'arribar al cim o bé el moment de tornar. L'enginyeria del programari, en canvi, ha de tenir en compte el manteniment futur del sistema i també la seva operació.
- En l'alpinisme no és habitual canviar de membres de l'equip durant l'ascensió mentre que, al llarg de la vida del programari, és habitual que es produeixin canvis.

5.

a) Gestió del projecte
b) Gestió del projecte
c) Gestió del projecte
d) Gestió del projecte, requisits
e) Modelització
f) Modelització
g) Requisits
h) Requisits
i) Requisits
j) Construcció i proves
k) Construcció i proves
l) Qualitat
m) Qualitat
n) Manteniment.

6.

a) 1 (Objectiu clar i solució coneguda)
b) 2 (Objectiu clar i solució no coneguda)
c) 1 (Objectiu clar i solució coneguda)
d) 3 (Objectiu poc clar i solució no coneguda)
e) 1 (Objectiu clar i solució coneguda)
f) 2 (Objectiu clar i solució no coneguda)

7.

a) Cada setmana són 5 unitats d'estimació.
Setmana 1: anàlisi i disseny de *a*, *b* i part de *c* (1/2)
Setmana 2: anàlisi i disseny de part de *c* (1/2), *d*, *e* i *f*.
Setmana 3: anàlisi i disseny de *g*. Implementació de *a* i part de *b* (3/5)
Setmana 4: implementació de part de *b* (2/5) i part de *c* (3/4)
Setmana 5: implementació de part de *c* (1/4), *d* i *e*
Setmana 6: implementació de *f* i g. Proves de *a*.
Setmana 7: proves de *b*, *c*, *d* i part de *e* (1/2).
Setmana 8: proves de *e* (1/2), *f* i *g*.
b) En aquest cas no distingim anàlisi i disseny, implementació i proves, sinó que planifiquem per requisits analitzats, implementats i provats.
Iteració 1 (setmanes 1 i 2): requisit *a*, *c* (el *b* no el podem completar en aquesta iteració i, per això, fem el *c* abans).
Iteració 2 (setmanes 3 i 4): requisit *b*
Iteració 3 (setmanes 5 i 6): requisits *d* i *e*
Iteració 4 (setmanes 7 i 8): requisits *f* i *g*.

8.

a) No segueix els principis del manifest àgil perquè aquest, tot i que diu que preferim "Programari que funciona per sobre de documentació exhaustiva", també diu que reconeix el valor dels ítems a la dreta i, per tant, no fer cap documentació va en contra d'aquesta última frase.
b) No compleix el principi "Col·laboració amb el client per sobre de negociació del contracte".
c) En aquest cas dependrà de quant de temps han de dedicar els desenvolupadors a omplir l'informe, ja que, en funció d'això, es pot considerar que no es compleix el principi "Individus i interaccions per sobre de processos i eines".
d) Sí que compleix el principi "Individus i interaccions per sobre de processos i eines".
e) Sí que compleix el principi de "Col·laboració amb el client per sobre de negociació del contracte".
f) No compleix el principi "Respondre al canvi per sobre de seguir un pla"
g) Sí que compleix el principi "Respondre al canvi per sobre de seguir un pla".

h) Sí que compleix el principi "Individus i interaccions abans que processos i eines".
i) No compleix el principi "Individus i interaccions abans que processos i eines".
j) Sí que compleix el principi "Programari que funciona abans que la documentació exhaustiva".

9. En la versió estructurada, hi ha un artefacte anomenat *Modelo lógico de datos normalizado* que no existeix en la versió OO. En canvi, la versió OO té un artefacte anomenat *Modelo de clases de análisis* que no existeix en la versió estructurada. En canvi, el "Catálogo de requisitos" és comú a tots dos enfocaments.

10. En OpenUP, l'analista és responsable del següent:

- Identificar i donar una visió general dels requisits
- Detallar els requisits globals del sistema
- Detallar els casos d'ús
- Desenvolupar la visió tecnològica

11. Les propietats que defineix l'article són:

- Les mètriques han de mesurar coses mesurables.
- Les mètriques s'han de prendre al nivell adequat i en la unitat adequada.
- Només té sentit mesurar si es vol actuar sobre el resultat.

Un exemple típic de mètrica és el nombre de línies de codi d'un programa. Són fàcils de mesurar (compleixen el primer principi) i és fàcil de determinar la unitat (línies de codi o milers de línies de codi, segons com sigui de gran el programa).

S'ha de tenir en compte, però, que els programes més complexos necessitaran, *a priori*, més línies de codi. Per això, si volem seguir el segon principi, a l'hora de mesurar variacions en el nombre de línies de codi hem de tenir en compte la variació en funcionalitat del programa.

Finalment, pel que fa al tercer principi, sabem que un increment en el nombre de línies de codi que no sigui "proporcional" a l'increment en la funcionalitat del programa acostuma a estar associat a un excés de repetició dins del codi i a un descens en la qualitat d'aquest, per la qual cosa, si es detecta aquesta situació, caldrà actuar per tal de reduir el volum de codi per mantenir.

12. Tots dos són llistes de coses que s'han de fer abans d'aconseguir un objectiu concret (treure una nova versió del producte en el cas del *product backlog* o bé finalitzar amb èxit una iteració en el cas de l'*spring backlog*), però, mentre que en el *product backlog* el que tenim són funcionalitats, errors per corregir i millores, en l'*sprint backlog* el que tenim són tasques concretes (és a dir, l'*sprint backlog* conté la descomposició dels ítems del *product backlog*).

13. Un altre exemple de separació entre interfície i implementació podria ser una calculadora: la interfície seria el conjunt de tecles amb què introduïm els nombres i els signes de les operacions i la implementació seria el conjunt de circuits que formen la calculadora, i també el programari que la controla.

14. A la web de Martin Fowler podem trobar el patró *rang*, que serveix per a representar rangs de dades com ara el període de validesa d'una oferta en una aplicació de comerç electrònic.

15. En el cas del cicle de vida en cascada, si ja hem fet l'etapa de requisits, caldrà modificar el document de requisits del sistema i propagar el canvi als documents d'anàlisi, disseny, implementació i proves que pertoqui segons en quina de les etapes del procés ens trobem.

En el cas del cicle de vida iteratiu i incremental dependrà fonamentalment de si hem implementat o no el requisit. Si ja l'hem implementat, el cost serà similar al cas anterior (tornar a fer requisits, anàlisi, disseny, implementació i proves) mentre que, si no l'hem implementat encara, només caldrà revisar l'estimació, ja que no l'haurem de detallar fins que arribem a l'iteració en què s'hagi d'implementar.

16. Es tracta d'un programa escrit en un llenguatge de proves d'aplicacions web. El principal avantatge de fer servir un llenguatge específic del domini, en aquest cas, és que és bastant fàcil fer-se una idea de quin és el comportament que està simulant el programa llegint el codi font.

Exercicis d'autoavaluació

1. El programari per a sistemes d'informació és un tipus de programari que gestiona una certa informació mitjançant un sistema gestor de bases de dades i dóna suport a una sèrie d'activitats humanes dins del context d'un sistema d'informació. (Vegeu el subapartat 1.3.)

2. El programari és intangible, no es manufactura, no es desgasta i, a més, queda obsolet ràpidament. (Vegeu el subapartat 1.6.)

3. El desenvolupament té un inici i un final clars i, a més, proporciona un resultat únic. Per això no s'organitza de manera contínua sinó com a projectes. (Vegeu el subapartat 2.1.)

4. Les persones, en funció del seu **rol**, han de dur a terme una sèrie de **tasques** que tindran com a punt de partida un conjunt d'**artefactes** i en generaran, com a resultat, un altre conjunt d'**artefactes**. (Vegeu el subapartat 2.2.)

5. Estudi de viabilitat, estimació, definició dels objectius, formació de l'equip de treball, establiment de fites i identificació de riscos. (Vegeu el subapartat 2.3.1.)

6. Els requisits expressen les necessitats i restriccions que afecten un producte de programari que contribueix a la solució d'un problema del món real. (Vegeu el subapartat 2.3.2.)

7. L'analista funcional és el responsable d'unificar les diferents visions del domini en un únic model que sigui clar, concís i consistent. (Vegeu el subapartat 2.4.3.)

8. Les etapes del cicle de vida en cascada són: requisits, anàlisi i disseny, implementació, proves i manteniment. (Vegeu el subapartat 3.2.1.)

9. El cicle de vida iteratiu i incremental accelera la retroalimentació, ja que cada iteració cobreix totes les activitats del desenvolupament. A més, en tot moment es té un producte operatiu, amb la qual cosa es pot accelerar el retorn de la inversió. (Vegeu el subapartat 3.2.2.)

10. El pla de sistemes d'informació té com a finalitat assegurar que el desenvolupament dels sistemes d'informació es fa de manera coherent amb l'estratègia corporativa de l'organització. El pla de sistemes d'informació, per tant, és la guia principal que han de seguir tots els projectes de desenvolupament que comenci l'organització. Com a tal, incorpora un catàleg de requisits que han de complir tots els sistemes d'informació, una arquitectura tecnològica, un model de sistemes d'informació, etc. (Vegeu el subapartat 3.3.1.)

11. Les sis pràctiques fonamentals del procés unificat són: desenvolupament iteratiu i incremental, gestió dels requisits, arquitectures basades en components, utilització de models visuals, verificació de la qualitat del programari i control dels canvis al programari. (Vegeu el subapartat 3.3.2.)

12. Scrum defineix tres artefactes: el *product backlog*, l'*sprint backlog* i el *burn down chart*. (Vegeu el subapartat 3.3.3.)

13. Com a avantatges tenim l'oportunitat, disminució de costos, fiabilitat i eficiència. Un dels problemes seria que cal fer una gestió exhaustiva de les diferents versions del component. (Vegeu el subapartat 4.1.)

14. L'ocultació d'informació consisteix a amagar els detalls sobre l'estructura interna d'un mòdul, de manera que podem definir clarament quins aspectes dels objectes són visibles públicament (i, per tant, utilitzables pels reutilitzadors) i quins no. (Vegeu el subapartat 4.1.1.)

15. Els principals avantatges de l'ús de patrons són: reutilitzar les solucions, beneficiar-nos del coneixement previ, comunicar i transmetre la nostra experiència, establir un vocabulari comú, encapsular coneixement detallat i no haver de reinventar una solució al problema. (Vegeu l'apartat 4.1.5.).

16. Les eines CASE són aplicacions informàtiques que ajuden a l'enginyer del programari a aplicar els principis de l'enginyeria al desenvolupament de programari. (Vegeu el subapartat 4.3.)

17. El llenguatge unificat de modelització (UML) és un estàndard publicat per l'OMG que defineix la notació acceptada universalment per a la creació de models del programari. (Vegeu el subapartat 5.1.)

Glossari

abstracció *f* L'abstracció consisteix a identificar els aspectes rellevants d'un problema de manera que ens puguem concentrar només en aquests.

activitat *f* Conjunt de tasques que duen a terme les persones que tenen un determinat rol.

anàlisi *f* L'anàlisi d'un sistema informàtic documenta com ha de ser el producte per desenvolupar des d'un punt de vista extern (és a dir, sense considerar com està fet per dintre). Habitualment, aquesta documentació es fa en forma de models.

artefacte *m* Objecte produït pel treball de l'ésser humà. En el context de l'enginyeria del programari, cada un dels documents, models, programes, etc., que es generen com a resultat del treball de l'enginyer.

cicle de vida *m* El cicle de vida d'un procés defineix quines són les diferents etapes per les quals va passant un procés al llarg de la seva vida. En l'enginyeria del programari fem servir aquest terme per a classificar els mètodes en famílies segons el tipus de cicle de vida.
Vegeu **cicle de vida en cascada**, **cicle de vida iteratiu** i **cicle de vida incremental**.

cicle de vida en cascada *m* El cicle de vida en cascada és un cicle de vida seqüencial que consta de les etapes següents: requisits, anàlisi i disseny, implementació, proves i manteniment. Tot i haver-hi diverses variants del cicle de vida en cascada, el que les caracteritza a totes és la seva naturalesa seqüencial.

cicle de vida incremental *m* Un cicle de vida és incremental quan el producte per desenvolupar es crea mitjançant petits increments de funcionalitat completa que es van agregant al producte desenvolupat.

cicle de vida iteratiu *m* Un cicle de vida és iteratiu si organitza el desenvolupament en iteracions de temps determinat. El principal avantatge del cicle de vida iteratiu és que permet establir punts de control regulars (al final de cada iteració) durant el desenvolupament.

computer aided software engineering Terme que es fa servir habitualment per a fer referència a les eines que tenen com a finalitat ajudar l'enginyer de programari a aplicar els principis d'enginyeria al desenvolupament de programari.
Sigla **CASE**

construcció del programari *f* La construcció del programari és l'activitat que té com a finalitat l'obtenció del producte executable. Això inclou, entre altres, la creació del codi font, la creació dels arxius executables o la gestió de la configuració.

desenvolupament àgil *m* El desenvolupament àgil consisteix a aplicar els quatre principis del Manifest àgil al desenvolupament de programari. També es coneix com a desenvolupament àgil el conjunt de mètodes de desenvolupament que comparteixen els principis esmentats.

desenvolupament *lean* *m* El desenvolupament *lean* intenta aplicar les tècniques de manufactura *lean* al desenvolupament de programari. Aquestes tècniques es deriven, originàriament, del sistema de producció Toyota (TPS).

desenvolupament de programari *m* El desenvolupament de programari és l'acte de produir o crear programari.

disseny de programari *m* El disseny de programari documenta l'estructura i el comportament intern d'un sistema informàtic.

enginyeria *f* L'enginyeria és l'aplicació d'un enfocament sistemàtic, disciplinat i quantificable a les estructures, màquines, productes, sistemes o processos per a obtenir un resultat esperat.

enginyeria del programari L'enginyeria del programari és l'aplicació d'un enfocament sistemàtic, disciplinat i quantificable al desenvolupament, operació i manteniment del programari; és a dir, l'aplicació de l'enginyeria al programari.

gestió del projecte *f* La gestió del projecte és tot allò que no sigui l'execució del projecte. El seu objectiu principal és assegurar l'èxit del projecte.

implementació (com a activitat del desenvolupament) *f* Creació del codi font.

implementació (com a part de l'ocultació d'informació) *f* Part interna d'un objecte o un sistema; inclou tot allò que no és visible a aquells que el fan servir.

interfície (com a part de l'ocultació d'informació) *f* Part externa d'un objecte o un sistema; inclou tot allò que sí que és visible a aquells que el fan servir.

manteniment del programari *m* Comprèn la modificació posterior al desenvolupament del producte de programari per tal de corregir-ne els errors o bé adaptar-lo a noves necessitats.

mètode *m* Definició de les característiques del procés o procediment disciplinat utilitzat en l'enginyeria d'un producte o en la prestació d'un servei.

metodologia *f* Ciència que estudia els mètodes. Conjunt dels mètodes d'una disciplina.

modelització *f* Activitat que consisteix en la creació de models del sistema per desenvolupar: aquests models facilitaran la comprensió dels requisits i el disseny del sistema.

ocultació d'informació *f* L'ocultació d'informació consisteix a amagar els detalls sobre l'estructura interna d'un mòdul, de manera que podem definir clarament quins aspectes dels objectes són visibles públicament (i, per tant, utilitzables pels reutilitzadors) i quins no.

operació del programari *f* L'operació del programari consisteix a executar el producte de programari dins del seu entorn d'execució per tal de dur a terme la seva funció.

procediment *m* Vegeu **procés**.

procés *m* Manera de descabdellar-se una acció progressiva.

programació *f* Típicament, s'anomena programació la creació del codi font.

programari *m* Conjunt de programes. Més formalment, el conjunt dels programes de computació, procediments, regles, documentació i dades associades que formen part de les operacions d'un sistema de còmput.

programa *m* Conjunt d'instruccions codificades i de dades que són l'expressió completa d'un procediment, i en particular d'un algorisme, executable per un sistema informàtic.

requisit *m* Els requisits expressen les necessitats i restriccions que afecten un producte de programari que contribueix a la solució d'un problema del món real.

sistema d'informació *m* Un sistema d'informació és qualsevol combinació de tecnologia de la informació i activitats humanes que utilitzen aquesta tecnologia per a donar suport a l'operació, gestió o presa de decisions.

tasca *f* Com a part de l'execució d'un procés, una tasca l'ha de dur a terme una persona amb un rol concret per tal de crear uns artefactes de sortida a partir d'uns artefactes d'entrada.

unified modeling language Llenguatge unificat de modelització. És un estàndard publicat per l'OMG que defineix la notació acceptada universalment per a la creació de models del programari.
Sigla **UML**

Bibliografia

Bibliografia principal

Pressman, R. S. (2005). *Ingeniería del Software* (6a. ed.). McGraw Hill.

Aquest llibre és una bona introducció a l'estudi de l'enginyeria del programari en general.

Diversos autors (2004). *SWEBOK. Software Engineering Body Of Knowledge Guide*. IEEE Computer Society.

Aquesta obra recull el cos de coneixement (és a dir, aquelles idees que són àmpliament acceptades en la indústria) de l'enginyeria del programari. Es pot consultar a l'adreça:

http://www.computer.org/portal/web/swebok/htmlformat (darrera visita, setembre 2010)

Bibliografia complementària

Wysocki, R. K. (2009). *Effective Project Management: Traditional, Agile, Extreme* (5a. ed.). Wiley.

Per a l'estudi dels diferents mètodes de desenvolupament, i també quan és més convenient aplicar-ne un o altre.

Cockburn, A. (2007). *Agile Software Development: The Cooperative Game* (2a. ed.). Addison-Wesley.

Per a reflexionar sobre la naturalesa del desenvolupament del programari, l'estudi de diverses metàfores i el desenvolupament àgil de programari.

Pàgina oficial del mètode Mètrica 3. http://www.csae.map.es/csi/metrica3/index.html (darrera visita: setembre 2010).

Aquí podeu baixar els documents que descriuen el mètode. És especialment interessant el document d'introducció, ja que ens dóna una visió general del mètode i ens ajuda a situar en context la resta de documents.

Wiki del mètode OpenUP. http://epf.eclipse.org/wikis/openup/ (darrera visita: setembre 2010).

Wiki del mètode OpenUP, variant àgil del procés unificat. Aquest mètode està publicat sota llicència EPL i, per tant, és públicament accessible.

Scrum Guide. http://www.scrum.org/scrumguideenglish/ (darrera visita: setembre 2010)

Aquest petit document escrit per un dels creadors de Scrum ens dóna una visió més detallada del mètode àgil de desenvolupament Scrum.

Pàgina oficial de l'OMG sobre MDA. http://www.omg.org/mda/ (darrera visita: setembre 2010).

Pàgina oficial de l'OMG sobre MDA on podem trobar documentació relacionada amb l'estàndard i també la versió oficial.

Referències bibliogràfiques

Brooks, F. (1987). "Essence and Accidents of Software Engineering". *IEEE Computer Magazine*.

Cusumano, M.; MacCormack, A.; Kemerer, C. F.; Crandall, W. (desembre, 2003). *Software Development Worldwide: the State of the Practice*. IEEE Software.

DeMarco, T.; Lister, T. (1999). *Peopleware: Productive Projects and Teams* (2a. ed.). Dorset House Publishing Company.

Fowler, M. "Language Workbenches: The Killer-App for Domain Specific Languages?". http://martinfowler.com/articles/languageWorkbench.html (darrera visita: setembre 2010).

Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. (1994). *Design Patterns: elements of Reusable Object-Oriented Software*. Addison-Wesley.

Meyer, B. (1999). *Construcción de Software Orientado a Objetos*. Prentice Hall.

Poppendieck, M.; Poppendieck, T. (2003). *Lean Software Development: An Agile Toolkit*. Addison-Wesley.

Rational Unified Process: BEST Practices for Software Development Teams http://www.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251_bestpractices_TP026B.pdf. (darrera visita: setembre 2010).

Toyota Production System. http://www2.toyota.co.jp/en/vision/production_system/index.html (darrera visita: setembre 2010).