Generació automàtica de codi Java amb ArgoUML

Ismael Jiménez Gallardo
ETIS

Anna Queralt Calafat

18 juny 2007

Resum

Aquesta memòria presenta un estudi de la generació automàtica de codi Java a partir de diagrames UML amb l'eina ArgoUML. El cicle de vida tradicional del software presenta alguns problemes com la manca de sincronització entre codi i documentació, poca portabilitat i problemes d'interoperabilitat. El paradigma MDA vol solucionar en part aquests problemes fent dels models el centre del desenvolupament del software i apostant per la generació automàtica de models i codi. D'aquesta manera, UML, Java i la generació de codi poden jugar un paper molt important en MDA.

Aquesta memòria presenta un estudi al voltant d'ArgoUML, una eina CASE per la creació de models UML que permet generar codi Java automàticament a partir de diagrames de Classes. Primer s'analitza aquesta eina des del punt de vista d'un usuari (què es pot fer, com es fa) i desprès des del punt de vista d'un programador: quina va ser la filosofia del disseny, com s'estructura el codi i, amb més detall, com es genera codi Java a partir de models UML.

Finalment, aquesta memòria presenta els punts on ArgoUML podria millorar la generació de codi, tant en errors que es podrien reparar com en noves funcionalitats que es podrien afegir. Tots aquests canvis han estat implementats satisfactòriament en el codi font d'ArgoUML.

Paraules clau

Generació automàtica de codi
UML
Java
MDA
ArgoUML

Índex de continguts

Índex de continguts 5
Índex de figures 7
1 Introducció 9
1.1 Justificació del TFC 9
1.1.1 Model-driven Architecture 9
1.1.2 Unified Modeling Language 11
1.1.3 Java 14
1.1.4 Generació automàtica de codi 14
1.2 Objectius del TFC 15
1.3 Enfocament i mètodes seguits 16
1.3.1 Escollir el marc de treball: UML, Java i ArgoUML 16
1.3.2 Estudi de l'eina ArgoUML 16
1.3.3 Millores en la generació de codi amb ArgoUML 17
1.3.4 Recerca bibliogràfica per emmarcar el treball 17
1.4 Planificació del projecte 17
1.5 Productes obtinguts 17
1.6 Descripció de la resta de capítols 17
2. ArgoUML per usuaris i programadors 19
2.1 ArgoUML des del punt de vista d'un usuari 19
2.1.1 Instal·lació 19
2.1.2 Guia ràpida d'ús 19
Panell Explorador 19
Panell Editor 19
Panell de Detalls 20
Panell *To Do* 20
2.1.3 Generació de codi i enginyeria inversa 21
2.1.4 Conclusions 23
2.2 ArgoUML des del punt de vista d’un programador 23
2.2.1 Filosofia del desenvolupament d'ArgoUML 23
Reflexió en acció (*Reflection-in-action*) 24
Disseny oportunista 24
Comprensió i solució de problemes 24
2.2.2 Programari lliure, Codi obert i Java 24
2.2.3 Descàrrega i desenvolupament 25
2.2.4 Estructura del codi 25
Kernel 26
Application 26
UI (User Interface) 27
UML 27
Cognitive 29
2.2.5 Generació automàtica de programari 30
3. Millores en la generació de codi Java a partir de diagrames de classes UML amb ArgoUML: descripció, discussió i implementació 33
3.1 Correcció d’errors en la generació de codi d'ArgoUML 33
3.1.1 Cardinalitat dels atributs 34
Descripció 34
Implementació 35

3.1.2 Associacions múltiples sense paper (*rolename*)..36
Descripció..36
Implementació..36
3.2 Noves funcionalitats afegides a ArgoUML...36
3.2.1 Mètodes get i set..36
Descripció..36
Implementació..41
3.2.2 Header 42
Descripció..42
Implementació..42
4. Conclusions 43
5. Glossari...44
6. Bibliografia..45

Índex de figures

Figura 1 Esquema tradicional de desenvolupament de software 9
Figura 2 Cicle de vida del software en MDA........ 9
Figura 3 Els tres estadis principals de MDA........ 10
Figura 4 Diagrames UML 2.0 11
Figura 5 Exemple d'associació 12
Figura 6 Exemple d'agregacions en UML 12
Figura 7 Exemple de composició 12
Figura 8 Exemple d'herència 13
Figura 9 Vista típica d'ArgoUML 19
Figura 10 Diagrama amb una sola classe 20
Figura 11 Principals components del paquet Kernel........ 25
Figura 12 Detall de la classe Main, component principal de Application 26
Figura 13 Principals components del paquet UI (User Interface) 26
Figura 14 Subpaquets principals que composen el paquet UML........ 27
Figura 15 Components principals del subpaquet Diagram (package UML)........ 27
Figura 16 Detall de la interfície CodeGenerator, la qual implementaran el generador de codi Java (GeneratorJava) 28
Figura 17 Components principals del subpaquet Reveng 28
Figura 18 Components principals del paquet Cognitive 28
Figura 19 Diagrama de classes que mostra la classe GeneratorJava i les seves associacions 30
Figura 20 Diagrama de seqüència que representa el recorregut típic de la generació de codi dins de GeneratorJava 31
Figura 21 Diagrama de classes amb associacions de cardinalitat múltiple........ 33
Figura 22 Exemple d'un classe amb atributs primitius 36
Figura 23 Exemple d'una associació un-a-un entre classes 37
Figura 24 Exemple d'associació un-a-molts 37
Figura 25 Exemple de relació molts-a-molts 40

1 Introducció

1.1 Justificació del TFC

Aquest TFC es troba dins l'àrea de Generació Automàtica de Programari. En concret, es centra en la generació de codi Java a partir de diagrames de *Unified Modeling Language* (UML) amb l'eina ArgoUML. En un sentit general, la generació automàtica de codi es pot emmarcar com un pas dins el procés més general anomenat *Model-driven Architect*ure (MDA). Com veurem en el següent apartat, MDA pretén solucionar alguns problemes que presenta el cicle de vida 'tradicional' de producció de software. En MDA, els models (que es poden expressar en UML) són el punt central i la generació de codi es fa automàticament. Per tant, en aquest apartat es donaran algunes idees bàsiques sobre MDA, UML, Java i la importància de la generació automàtica de codi en general. Mentre que la secció de MDA és relativament autocontinguda, les seccions sobre UML i Java suposen certa familiaritat amb els conceptes d'Orientació a Objectes (OO).

1.1.1 Model-driven Architecture

En la producció de software existeixen diferents cicles de vida (maneres d'afrontar el procés de producció de software) que contenen una sèrie de passes a seguir. Tant si parlem del tradicional cicle de vida en cascada (*waterfall*) com si parlem de processos iteratius o incrementals, les següents fases sempre es troben presents[1]:

1. Recollida de requisits
2. Anàlisi i descripció funcional
3. Disseny
4. Programació
5. Test
6. Manteniment

Les fases 1 a 3 produeixen una gran quantitat de documentació que, molt sovint, queda obsoleta un cop es comença la fase de programació. Si cal fer una modificació en temps de programació, la documentació anterior ja no s'actualitza (per manca de temps o simplement perquè tampoc té molt sentit, ja que el canvi ha d'ocórrer en el codi en qualsevol cas). Això porta molt sovint a una situació en què el codi i la documentació ja no descriuen el mateix, i per tant la feina feta a les fases 1 a 3 ja deixa de tenir utilitat. Altres mètodes com *Extreme Programming* (XP) han volgut evitar això concentrant-se només en les fases de programació i test, acceptant que finalment és el codi el que determina el producte final. Aquesta solució acostuma a donar bons resultats sempre que l'equip programador es mantingui, ja que sense documentació esdevé molt difícil poder transmetre la informació necessària a altres programadors.

Apart d'aquest problema de manteniment de la documentació d'alt nivell (distingim aquí la documentació de disseny o alt nivell de la típica documentació generada per *Javadoc*, per exemple, que seria documentació de baix nivell), la generació de software tradicional presenta dos altres grans problemes: la portabilitat i la interoperabilitat.

El problema de la portabilitat apareix perquè el mercat ofereix contínuament eines que donen més prestacions i qualsevol empresa sempre vol adaptar-se a aquestes noves eines perquè ofereixen més serveis, perquè les eines antigues deixen de tenir suport o simplement perquè el

client ho demana. De fet, fins i tot les eines tenen una evolució que no sempre les fan compatibles amb versions anteriors, de manera que la necessitat d'adaptar-se és inevitable.

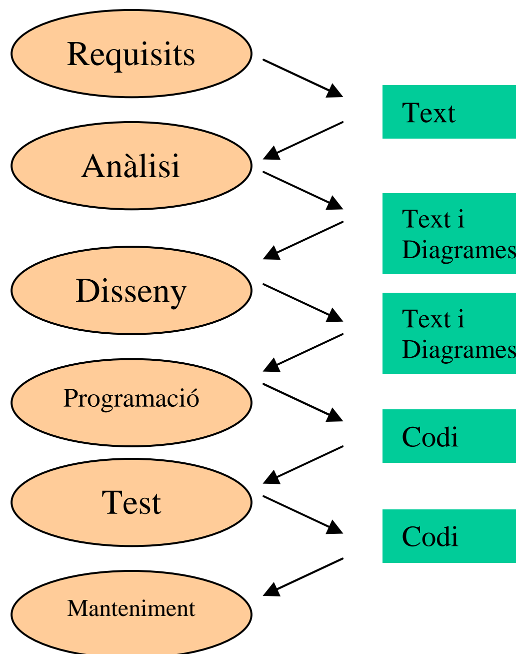

***Figura 1** Esquema tradicional de desenvolupament de software*

Pel que fa a la interoperabilitat, sempre hem de tenir en compte que qualsevol producte informàtic haurà d'interaccionar amb d'altres productes i que per tant s'ha d'assegurar la compatibilitat amb el món exterior. A més a més, és molt probable que fins i tot un producte no sigui monolític, sinó que tingui diversos components desenvolupats amb plataformes diferents.

El *Object Management Group* (OMG), un consorci per promoure la programació orientada a objectes i el modelatge, va llençar en 2001 una iniciativa per intentar solucionar aquests problemes: el *Model-driven Architecture* (MDA), de manera que el els models esdevinguessin el protagonista principal i no el codi[2].

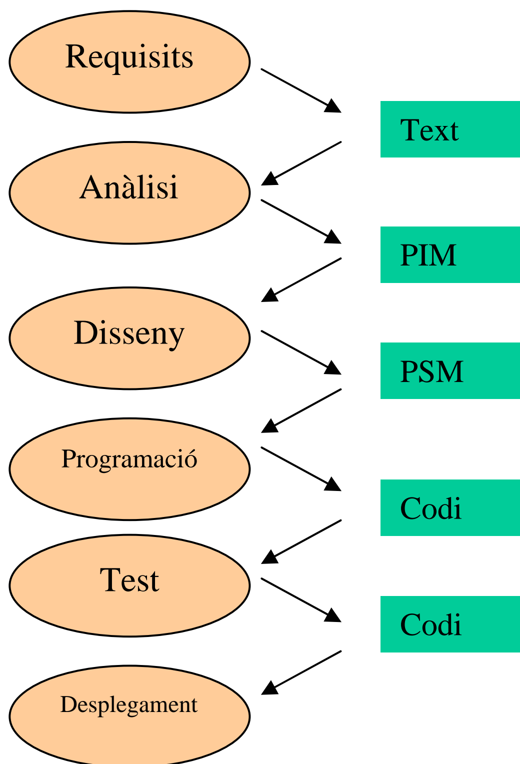

***Figura 2** Cicle de vida del software en MDA*

Simplificant, es podria dir que un dels objectius principals de MDA és separar el disseny de l'arquitectura[3]. D'aquesta manera, el disseny recolliria els requeriments funcionals, és a dir, els escenaris que es poden presentar en l'ús del software per part dels usuaris. En canvi, l'arquitectura s'ocuparia més aviat de proveir la infraestructura necessària per portar a terme el desenvolupament. Expressat en llenguatge MDA, el disseny es faria amb *Platform Independent Models* (PIM) que fan servir llenguatges d'ús general (UML, per exemple). L'arquitectura es basaria llavors en *Platform Specific Models* (PSM) que farien servir models amb característiques específiques de la plataforma de desenvolupament (UML amb extensions, per exemple). De fet, com que possiblement haurem de donar suport a diverses plataformes, és habitual que a un PIM li corresponguin diversos PSM. En qualsevol cas, sempre tindriem models en aquests dos nivells. Finalment, tindriem el codi que es derivaria del corresponent PSM. Immediatament es veu que el fet de contar amb aquests tres nivells (PIM, PSM i Codi) significa que el programador ha d'elevar el seu

nivell d'abstracció i que per tant pot enfrontar-se a més sistemes amb menys esforç.

En qualsevol cas, aquest procés no sembla molt diferent dels estadis tradicionals de desenvolupament de software. On es troba llavors la gran aportació de MDA? Doncs en el fet que les transformacions entre PIM i PSM i entre PSM i codi es fan automàticament amb eines CASE (*Computer-aided Software Engineering*).

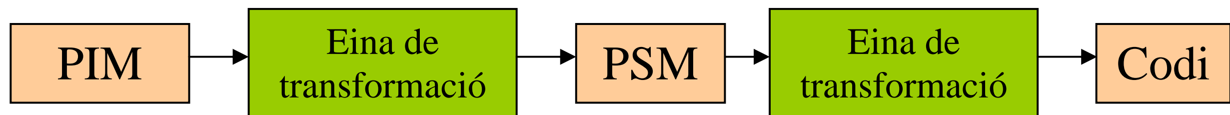

Figura 3 *Els tres estadis principals de MDA*

En definitiva, MDA pot ajudar a solucionar els típics problemes descrits anteriorment de la següent manera:

Productivitat

- Els programadors es poden concentrar en les PIM, deixant els detalls tècnics a la transformació entre PIM i PSM. Aconseguir una eina que faci aquesta transformació pot ser una tasca complexa i costosa però només caldrà fer-ho un cop.
- El fet de concentrar-se en nivells d'abstracció més alts permet afrontar amb més temps els veritables problemes del negoci.

Portabilitat

- Qualsevol disseny basat en una PIM és portable per definició. El grau de portabilitat dependrà de l'eina que faci la conversió automàtica entre PIM i PSM.
- Quan una nova tecnologia arribi (una PSM diferent), haurà de venir acompanyada de la seva corresponent transformació amb la PIM, de manera que el nostre model abstracte sempre estarà vigent

Interoperabilitat

- MDA no només genera diferents PSM basades en un mateix PIM per diferent plataformes, sinó que també proveeix ponts entre les diferents PSM per mantenir la interoperabilitat.

Manteniment i Documentació

- Els canvis que calgui fer en el sistema es faran a nivell de PIM, de manera que la documentació estarà al dia i sincronitzada amb el codi.

En definitiva, MDA vol fer dels models el centre de la programació i aconseguir que algun dia escriure codi sigui tant absurd com avui dia és programar directament en condi màquina.

1.1.2 Unified Modeling Language

Com hem dit abans, MDA necessita llenguatges per poder modelitzar el software i aquí és on el *Unified Modeling Language* (UML) juga un paper molt important. UML és un llenguatge de proposit general (és a dir, no restringit a una determinada plataforma) que inclou una notació gràfica per tal de crear un model abstracte d'un sistema[4]. De fet, UML va ser un

catalitzador de les tecnologies basades en models. UML va néixer en 1997 reunint característiques pròpies d'altres lleguntages formals anteriors que es basaven en mètodes OO. Tot i que UML va néixer per modelitzar software, avui en dia s'ha extés a altres branques i pot modelitzar tant enginyeries de sistemes en general com processos de negocis o administració d'empreses, per exemple.

UML distingeix tres parts fonamentals en el model d'un sistema:

Model funcional

- Explica la funcionalitat del sistema des del punt de vista de l'usuari
- Típicament representat pels diagrames de Casos d'Ús

Model Objecte (Estàtic)

- Descriu l'estructura fonamental del sistema (objectes, atributs, operacions, relacions) que és vàlida en qualsevol moment (per oposició al Model Dinàmic que citarem desprès)
- Típicament representat pels diagrames de Classes i Objectes

Model Dinàmic

- Representa el comportament intern del sistema, descrivint el comportament en el temps
- Típicament representat pels diagrames d'Estats, d'Activitats i de Seqüència.

Tot i que els models UML també tenen una part de text (la recollida dels casos d'ús, per exemple), el més notable d'aquest llenguatge són els diagrames que descriuen els diferents models d'una manera visual. La figura 4 mostra tots els diagrames que l'especificació UML 2.0 recull:

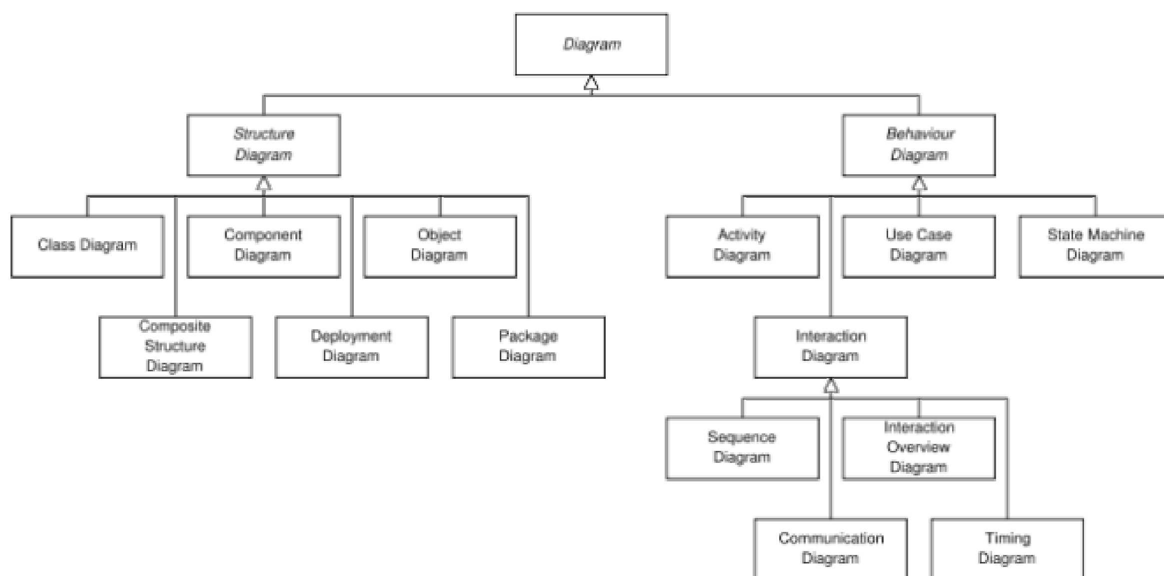

Figura 4 *Diagrames UML 2.0*[5]

De tots aquests diagrames, explicarem breument el diagrama de Classes, ja que és el que farem servir per generar el codi Java. Com el seu nom indica, aquest diagrama descriu les classes (en el sentit OO) i les relacions entre elles. Segons UML, una classe és una mena de classificador que conté tant atributs com operacions, de manera que els atributs poden ser

alhora altres classes i per tant tenir també mètodes. No cal dir que aquesta definició de classe coincideix amb la idea de classe segons OO.

Les classes poden presentar diverses relacions entre si, de les quals citarem només les següents[6]:

Associacions

Si dues classes estan associades en UML, vol dir que aquestes classes tenen una connexió i que es poden enviar missatges (saben de l'existència de l'altra). L'associació s'expressa per una línia que uneix ambdues classes. Si aquesta línia porta una fletxa, vol dir que la classe origen de la fletxa sap de l'existència de l'altre mentre que l'inrevés no és cert. Les associacions poden estar qualificades (un nom amb que una classe coneix una instància de l'altra classe) i poden tenir també una cardinalitat que indica el nombre d'instàncies que formen part de la relació.

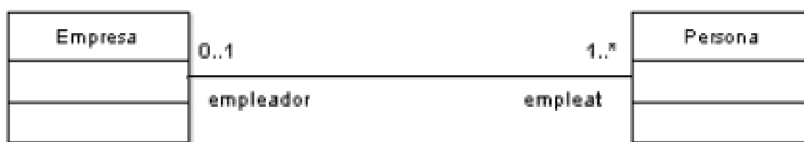

Figura 5 *Exemple d'associació*

Agregacions

Les agregacions són un cas particular d'associació on una classe fa el paper de 'part' i l'altre, el de 'tot'. Representa les relacions tipus acoblament-peces, continent-contingut o col-lectiu-membres. D'aquesta manera un objecte compost pot tenir objectes components que pertanyin a diverses classes. A més, entre dues classes podem tenir més d'una agregació.

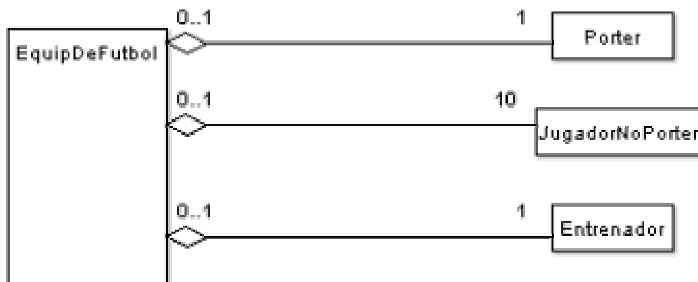

Figura 6 *Exemple d'agregacions en UML*

Composicions

En una composició, els objectes components no tenen vida pròpia, sinó que, quan es destrueix l'objecte compost del qual formen part, també es destrueixen.

Figura 7 *Exemple de composició*

Herència

Diem que una subclasse hereta d'una superclasse quan la primera conté tots els atributs i operacions de la segona, apart d'aquells que són específics de la subclasse.

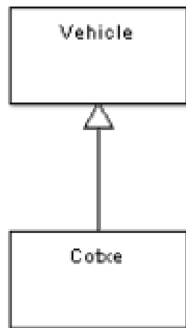

Figura 8 *Exemple d'herència*

1.1.3 Java

Java és un llenguatge de programació OO que va ser creat a principi dels anys 90 per *Sun Microsystems*. La sintaxi de Java és força similar a la de C/C++ però presenta una gran diferència amb aquests llenguatges: no es compila a codi màquina sinó que s'obté codi de bits (*bytecode*) que després és interpretat per una màquina virtual (*Java Virtual Machine*, JVM) en temps d'execució (cal dir que actualment Java permet també la compilació a codi màquina i que fins i tot hi ha processadors Java que permeten execució directa per part del hardware). Respecte a C++ en particular (també un llenguatge OO), Java ha simplificat alguns aspectes com ara la gestió de la memòria o la interacció amb els recursos de baix nivell del sistema[i].

Pel que respecte al contingut d'aquest TFC, podriem destacar els següents punts que fan interesant l'ús de Java dins el paradigma MDA[7]:

- MDA té una clara relació amb OO, tant per ser un producte de OMG com que per la estreta relació que existeix entre OO i els models en general (amb UML com el llenguatge de modelatge més popular). Per tant, sembla quasi automàtica l'elecció d'un llenguatge OO per la implementació de MDA. Tot i això, cal dir que MDA no pressuposa ni molt menys haver d'escollir un llenguatge de programació ni OO ni tan sols 'tradicional'. Idealment, MDA contempla que un compilador pugui traduir directament un model en codi de bits que no caldria inspeccionar. Tot i que existeixen eines d'aquest tipus, encara no són prou madures i fer servir un llenguatge de programació sembla avui en dia encara inevitable
- Dins els llenguatges de programació OO, Java és possiblement el llenguatge més popular, potser per una combinació de simplicitat (especialment respecte a C++ i a d'altres llenguatges OO 'purs' com Eiffel), portabilitat (l'ús de la JVM permet executar el mateix programa en diferent plataformes i això és potser el més interessant des del punt de vista de MDA) i bon màrqueting per part de *Sun Microsystems*.

1.1.4 Generació automàtica de codi

Aquesta secció vol mostrar quins són els avantatges de la generació automàtica de codi com una 'fi' en si mateix, és a dir, més enllà de MDA però sempre amb la idea de generar el codi a

[i] En contrapartida, la combinació de la JVM i la autogestió de memòria fa que les aplicacions en Java puguin executar-se més lentament que en C/C++.

partir d'un model. Per tant, alguns d'aquests avantatges poden ser comuns als que MDA presenta.
Podem agrupar els avantatges de la generació automàtica de codi des dels punts de vista dels programadors i dels *mànagers*[8,9]:

Programadors

- *Qualitat:* el codi escrit manualment acostuma a tenir una qualitat irregular en el sentit que el programador va millorant el seu codi a mesura que va implementant el disseny. La generació automàtica de codi no només evita això, sinó que fàcilment pot reparar un error (*bug*) en parts del codi que es repeteixin.
- *Consistència*: no cal dir que el codi generat automàticament és consistent en els noms fets servir per variables o funcions, per exemple, la qual cosa resulta en interfícies que són més fàcils de fer servir.
- *Un únic focus de coneixement*: un canvi en el model es propagaria amb totes les conseqüències per tot el codi. L'alternativa amb la programació manual és haver de buscar tot els casos que s'han d'actualitzar.
- *Més temps pel disseny:* No cal dir que la programació 'manual' requereix molt de temps emprat en escriure codi repetitiu. Aquest temps es podria aprofitar en millorar el disseny del software.

Mànagers

- *Consistència arquitectònica:* els programadors han de pensar dins el marc del generador de codi i per tant dins de la 'filosofia arquitectònica' del projecte. Si sembla que el generador fa difícil fer una certa tasca, potser és una indicació de què no s'hauria de fer.
- *Abstracció:* els generadors de codi (almenys en l'estadi PIM de MDA) haurien de ser independents de la plataforma d'aplicació. Aquesta abstracció obliga als programadors a escriure software que pot ser portable a d'altres plataformes i els analistes de negoci poden revisar i validar el model fàcilment
- *Suport moral*: els projectes llargs poden ser difícils de portar a terme i pot ser pitjor si a més inclouen escriure un codi tediós. La generació automàtica de codi possibilita que els programadors puguin treballar més temps en la part interessant

Finalment, cal dir que la generació automàtica de codi tampoc és la panacea que solucionarà tots els problemes. En general podem dir que els problemes més importants són:

- La generació de codi pot implicar una gran quantitat de feina per posar a punt l'eina de generació, la qual cosa serà pràcticament inevitable en els estadis més propers a la implementació ja que es tractarà de generar codi per una plataforma específica. Per tant, la generació automàtica de codi pot ser realment rentable si ens trobem amb un projecte gran (o si l'eina és ja un estàndard).
- Si l'entorn del projecte encara no està totalment definit o pot canviar ràpidament, cal preveure espai per la programació manual. En qualsevol cas, cal respectar la programació manual ja que sempre es flexible per incloure casos especials.
- Una única eina és possiblement insuficient per generar codi automàticament durant tot el cicle de vida d'un producte de software.

1.2 Objectius del TFC

Aquest TFC té dos objectius principals:

- Estudiar l'eina ArgoUML tant des del punt de vista d'un usuari (què permet fer i com es fa) com des del punt de vista d'un programador (quins components té el software i com estan relacionats)
- Estudiar amb detall com ArgoUML genera codi Java a partir de diagrames de Classes UML, proposar millores a aquesta generació i noves funcionalitats i implementar-les en el codi font d'ArgoUML

Altre objectius secundaris són:

- Donar a conèixer més l'eina ArgoUML a l'entorn de la UOC
- A nivell personal, entendre com s'estructura i s'implementa un software d'una certa complexitat i veure d'aprop els punts forts i febles de la generació automàtica de codi.

1.3 Enfocament i mètodes seguits

Per la realització d'aquest TFC s'han seguit les següents fasesi mètodes que tot seguit es detallen.

1.3.1 Escollir el marc de treball: UML, Java i ArgoUML

La idea inicial va ser estudiar la generació de codi Java a partir de UML, mirant de combinar tant l'anàlisi d'una eina existent com la implementació de codi per millorar i extendre aquesta eina. Per tant, la primera tasca va consistir en escollir quina eina es faria servir. Els criteris per seleccionar l'eina van ser:

- Havia de ser de llicència gratuïta
- Havia de ser de codi obert (*open source*), preferiblement Java, per poder afegir-hi extensions
- Era interessant que tingués certa documentació per programadors

No cal dir que ArgoUML compleix aquestes característiques. ArgoUML és una eina CASE pel disseny i anàlisi de sistemes de software OO[10]. Com veurem en el capítol 2, ArgoUML pretén ser quelcom més que una eina per fer diagrames UML, sinó que vol ajudar el programador en el procés de cicle de vida del software. ArgoUML va ser l'origen de l'eina comercial *Poseidon for UML*[11] i encara es continua desenvolupant.

1.3.2 Estudi de l'eina ArgoUML

El següent pas va ser un estudi detallat de l´eina ArgoUML (capítol 2 d'aquest treball). Aquesta tasca es va dividir en dues parts:

- *ArgoUML per usuaris*: l'objectiu era experimentar com usuari què podia fer ArgoUML, com s'havia de treballar amb aquesta eina i, especialment, com era la generació de codi i quins punts febles presentava. El mètode va consistir en simplement provar com usuari les diferents funcionalitats descrites en la guia d'usuari.
- *ArgoUML per programadors*: l'objectiu aquí era estudiar com estava organitzat el codi font d'ArgoUML, com es relacionaven els diferents components i, sobre tot, com aconseguia implementar la generació de codi automàticament. El mètodes van consistir en inspeccionar el codi font i l'API d'ArgoUML. Per entendre la generació de codi, un mètode seguit va ser inserir una mena de *warnings*, és a dir, un parell de línies de codi que fessin alguna cosa (com escriure per pantalla o en un fitxer) avisant que el flux del programa havia passat per allà en executar tal o qual operació.

1.3.3 Millores en la generació de codi amb ArgoUML

Aquesta part (corresponent al capítol 3) també va tenir dues tasques diferenciades:

- *Correcció d'errors:* a la fase de inspecció d'ArgoUML com a usuari es van detectar alguns errors en la generació de codi. El mètode seguit va ser simplement entendre com es generava el codi i on es produïa l'error, reparar-ho i testejar.
- *Afegit de noves funcionalitats*: en aquest cas, la consulta de bibliografia sobre generadors de codi i la experiència amb software similar (*Umbrello*[12]) va donar idees per afegir noves funcionalitats. El mètode seguit va ser proposar i argumentar una nova funcionalitat, implementar-la i testejar-la.

1.3.4 Recerca bibliogràfica per emmarcar el treball

Aquesta va ser l'última fase del projecte i es va basar en la recerca bibliogràfica per emmarcar la generació de codi en el marc més ampli de MDA. El mètode seguit va ser primer fer una recerca a Internet (*Wikipedia* és sempre un bon començament per trobar informació concisa i entenedora). A partir d'aquí es van trobar webs especialitzades i des d'aquí bibliografia recomanada També es va provar fer una recerca més general amb Google, d'on també es van trobar altres treballs interessants.

1.4 Planificació del projecte

El projecte es va planificar segons els quatre estadis descrits anteriorment. Els tres primers estadis es van fer coincidir amb les tres primeres Proves d'Avaluació Continuades. En concrete la temporització va ser la següent:

Dates	Secció - Capítols de la memòria
28 de febrer - 12 de març	Escollir el marc de treball
13 de març - 16 d'abril	Anàlisi d'ArgoUML - *Capítol 2*
17 d'abril - 21 de maig	Millores en la generació de codi - *Capítol 3*
22 de maig - 18 de Juny	Recerca bibliogràfica , redacció final del TFC i preparació de la presentació - Capítols 1, 4-6

1.5 Productes obtinguts

Apart del propi TFC i de la presentació, aquest treball adjunta dos documents més:

- El codi font de la classe *GeneratorJava* amb les modificacions fetes per incloure els canvis afegits a la generació de codi Java a partir de diagrames de classes UML. Per fer servir aquesta funcionalitat cal baixar el codi font d'ArgoUML (veure capítol 2) i substituir l'arxiu original per aquest.
- Documentació API generada amb javadoc amb una petita explicació de les noves funcionalitats afegides al codi.

1.6 Descripció de la resta de capítols

El contingut bàsic dels capítols 2 i 3 és el següent:

Capítol 2: Aquest capítol descriu ArgoUML primer des del punt de vist d'un usuari. Es descriu com obtenir i instal·lar el programa, les característiques bàsiques i com fer-les servir. Es donen exemples de com és el codi generat pel programa a partir de diagrames UML. La segona part d'aquest capítol descriu ArgoUML des el punt de vist d'un programador. Es comenta quina va ser la filosofia seguida en la seva concepció, quins són els components i subcomponents principals del software i com interaccionen. Finalment, es dona més èmfasi al component del software responsable de la generació de codi.

Capítol 3: Aquest capítol descriu primer alguns errors en la generació de codi per part d'UML i com s'haurien de corregir. També explica breument com s'ha solucionat això en el codi font. La segona part del capítol proposa millores en la generació de codi per part de UML. Es dona una visió de com s'hauria de traduir els diagrames UML a codi Java i després s'explica com s'ha implementat en el codi.

2. ArgoUML per usuaris i programadors

L'objectiu d'aquesta secció és donar una petita introducció a ArgoUML tant des del punt de vista de l'usuari (secció 2.1) com des del punt de vista del desenvolupador (secció 2.2), amb èmfasi en la part del codi referent a la generació automàtica de programari.

2.1 ArgoUML des del punt de vista d'un usuari

ArgoUML és una eina CASE pel disseny i desenvolupament de software seguint la metodologia d'Orientació a Objectes. ArgoUML implementa l'estàndard UML en la seva versió 1.4[i], de manera que no només es poden dissenyar els diagrames UML, sinó que també permet la generació automàtica de codi i l'enginyeria inversa entre UML i Java.
Actualment hi ha moltes eines que tenen característiques similars a ArgoUML. Tanmateix, aquesta eina té algunes peculiaritats que la fan atractiva des del punt de vista d'un usuari:

- És una eina gratuïta
- És fàcil d'instal·lar i de fer servir
- No només permet dissenyar els diagrames UML, sinó que guia el programador en el procés de desenvolupament per mitjà de *checklists*, *to do lists*, etc.
- Es pot fer servir en qualsevol plataforma que tingui Java disponible

La següent secció descriurà aquestes i d'altres característiques amb més detall. No cal dir que a la web de l'eina (http://argouml.tigris.org/) podem trobar manuals, seccions de *FAQ* i demés ajuda.

2.1.1 Instal·lació

Des de http://argouml.tigris.org/ es poden descarregar els executables per poder fer servir l'eina ArgoUML. Cal tenir instal·lat el Java Run time Environment (JRE) per tal d'executar l'aplicació.

2.1.2 Guia ràpida d'ús

Un cop obrim ArgoUML, ens trobem amb una interfície gràfica que mostra bàsicament quatre panells diferents i una barra de menú a la part superior (Figura 9). Aquests panells s'anomenen Explorador, Editor, de Detalls i *To Do* ('coses encara per fer'). En aquesta secció explicarem per a què serveixen els panells i, en general, què podem fer amb ArgoUML[13].

Panell Explorador

El trobem a la part superior esquerra i bàsicament mostra tot el que el projecte conté: quins diagrames hem generat, quines classes i associacions hem creat, tipus de dades fetes servir. Des d'aquí podem donar una ullada general a tots els components del nostre diagrama.

Panell Editor

Aquest és el panell principal, on podem dissenyar els diagrames UML. ArgoUML suporta els següents digrames: Classes, Casos d'Ús, Seqüència, Col·laboració, Estats i Desplegament. Damunt aquest panell editor hi ha les icones corresponents al diagrama en concret que fem servir, de manera que només hem de seleccionar l'element que volem inserir i clicar sobre la

[i] Actualment UML es troba en la versió 2.0

el panell. En aquesta mateixa finestra podem clicar i escriure les característiques més importants dels elements que inserim, tot i que és recomanable fer servir el panell de Detalls.

Panell de Detalls

En aquest panell podem escriure els detalls de cada element del diagrama: nom, atributs, operacions, cardinalitat, etc. També li podem afegir documentació (informació addicional sobre l'element en qüestió) i fins i tot tenir una visió immediata del codi que se generarà (més sobre aquest punt a la pròxima secció).

Un dels detalls que diferencien ArgoUML de moltes altres eines CASE es troba en una de les pestanyes d'aquest panell: llista de control. Aquesta secció fa una sèrie de preguntes sobre cada element per tal d'ajudar el programador a millorar el seu disseny. Per exemple, pregunta si el nom defineix realment bé la classe, si es podria afegir com atribut d'una altra classe, si pot tenir herència, ... En definitiva, mira de fer una sèrie de preguntes estàndard per tal que el programador no deixi passar aquest tipus de detalls. Aquesta característica enllaça amb la funcionalitat del següent panell.

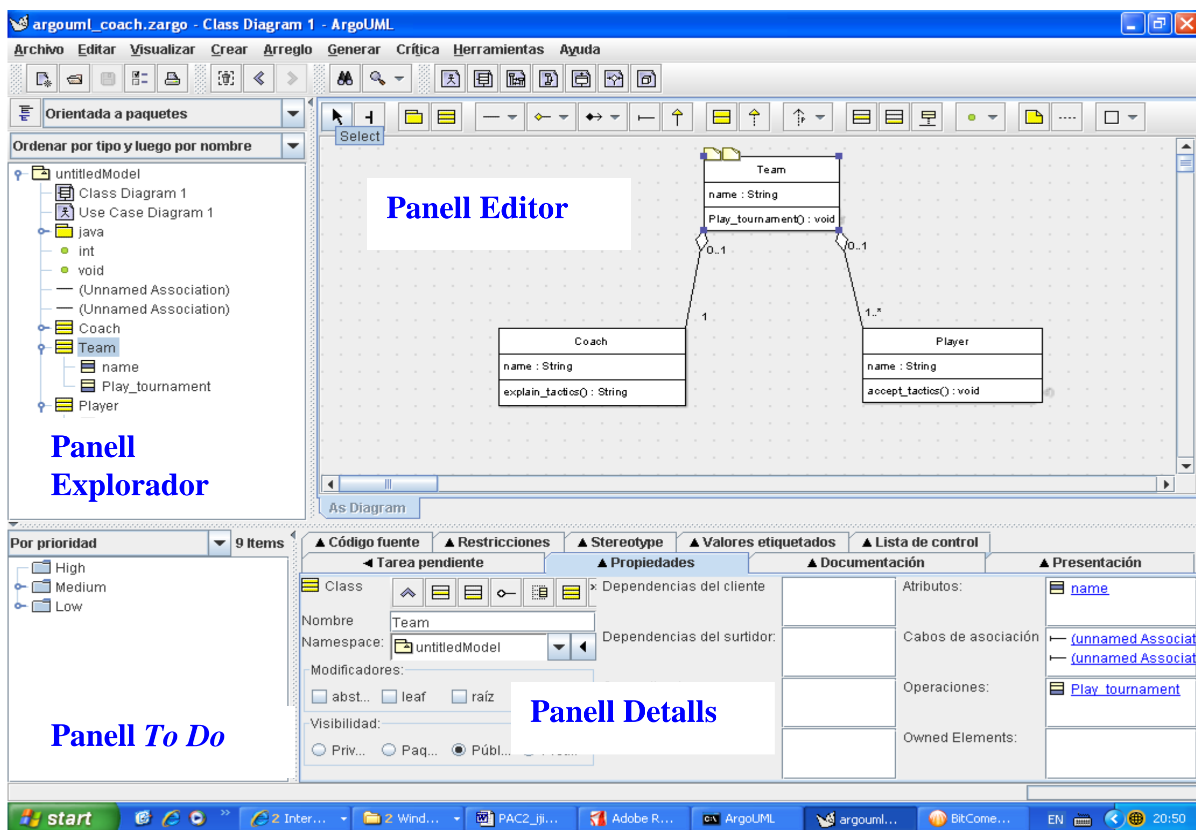

Figura 9 Vista típca d'ArgoUML

Panell *To Do*

Aquest panell es genera automàticament i ordena en tres graus de prioritat tasques que encara no s'han fet. Per exemple, avisa si una classe no té constructor, o si un atribut no té encara mètodes per donar-li valor, etc. En general, aquest panell ens recorda tot allò que encara no hem definit del model. Com veurem a la secció 2.2 (punt de vista del programador), aquest

panell es conseqüència directa de l'èmfasi en els aspectes cognitius que aquesta eina va tenir en el moment del seu desenvolupament.

2.1.3 Generació de codi i enginyeria inversa

ArgoUML només suporta la generació de codi a partir del diagrama (estàtic) de classes. Bàsicament fa una traducció automàtica de les característiques UML cap als equivalents en Java:

- Classes UML, juntament amb els seus atributs i mètodes, es converteixen en classes en Java
- Les associacions entre classes UML es tradueixen a codi Java incloent la classe associada com un atribut
- Les relacions de herència es codifiquen també en codi Java

Il·lustrarem aquests punts donant exemples de com és el codi que retorna ArgoUML amb diferents exemples de classes. Per exemple, la figura 10 mostra una classe que representa un nombre complex i com és el codi Java generat:

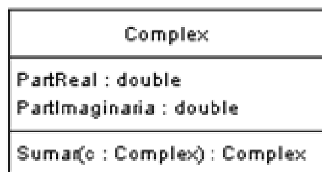

***Figura 10** Diagrama amb una sola classe*

Complex.java

```
public class Complex {
    private double PartReal;
    private double PartImaginaria;
    public Complex Sumar(Complex c) {
    return null;
    }
}
```

Podem veure que el codi generat és una *traducció* directa dels atributs i mètodes presentats.

El següent exemple (Figura 11) mostra la generació de codi a partir d'un diagrama que presenta diverses classes relacionades amb relacions d'agregació i d'herència.

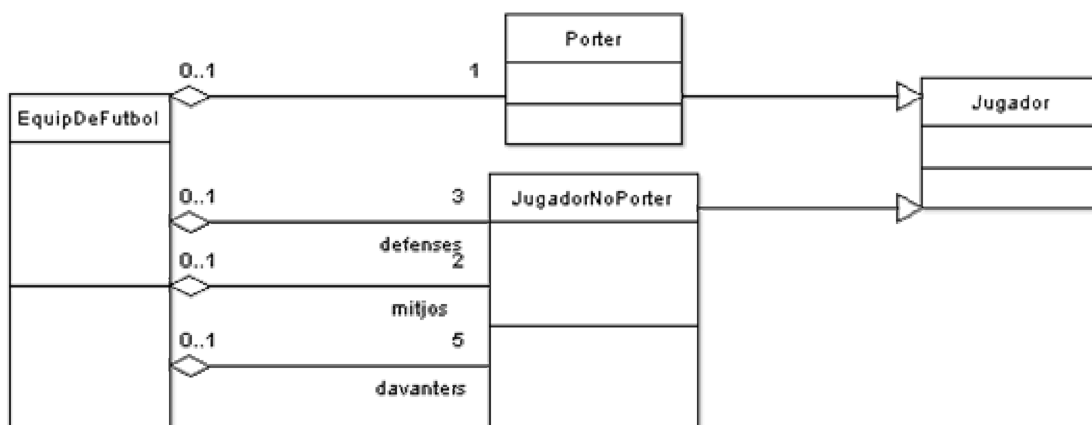

***Figura 11** Diagrama de classes amb agregacions i herència*

A continuació presentem el codi Java generat de les dues classes més interessants (*EquipDeFutbol* i *JugadorNoPorter*):

EquipDeFutbol.java

```
import java.util.Vector;
public class EquipDeFutbol {
    public Porter myPorter;
    /**
    *
    * @element-type JugadorNoPorter
    */
  public Vector  mitjos;
    /**
    *
    * @element-type JugadorNoPorter
    */
  public Vector  davanters;
    /**
    *
    * @element-type JugadorNoPorter
    */
  public Vector  defenses;
}
```

JugadorNoPorter.java

```
public class JugadorNoPorter extends Jugador {
    public EquipDeFutbol myEquipDeFutbol;
    public EquipDeFutbol myEquipDeFutbol;
    public EquipDeFutbol myEquipDeFutbol;
}
```

Veiem que la agregació amb cardinalitat 1 es resol afegint la classe amb el nom d'instància *myClass* (*myPorter* en el nostre cas). Pel que fa a les agregacions amb múltiples cardinalitats, es resol important la classe Vector i creant un Vector amb el nom del paper (*rolename*) que hem assignat. Com a comentari, es fa veure de quin tipus és l'element. Finalment, l'herència es resol amb un *extend*.

El següent exemple mostra un diagrama amb una classe associativa i el codi generat per una de les classes (Figura 12), que com podem veure no inclou aquesta classe associativa.

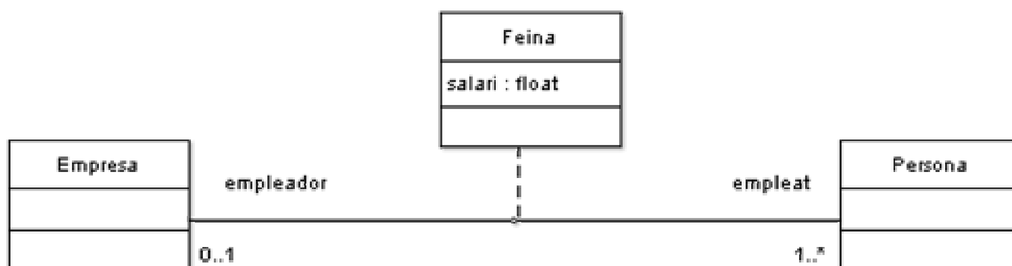

Figura 12 *Exemple de diagrama amb classe associativa*

Empresa.java

```
import java.util.Vector;
public class Empresa {

   /**
   *
   * @element-type Persona
   */
  public Vector  empleat;
}
```

Per generar aquest codi, només cal seleccionar la opció corresponent en el menú superior i seleccionar les classes que volem generar. D'altra banda, el panell de Detalls de cada classe també dóna directament el codi que es generarà en aquell moment.

Respecte a l'enginyeria inversa, cal dir que ArgoUML pot llegir classes Java i mostrar la classe en un diagrama UML (amb els seus atributs i mètodes). També pot identificar associacions d'herència o d'implementació però no representa aquestes relacions.

2.1.4 Conclusions

Des del punt de vista d'un usuari, podem resumir les principals característiques d'ArgoUML en els següents punts:

- ArgoUML és un programa fàcil d'instal·lar i de fer servir
- Suporta el disseny de tot tipus de diagrames UML, amb la característica diferenciadora respecte altres programes de l'ajuda al desenvolupador mitjançant llistes *to do*
- Permet la generació automàtica de codi Java a partir de diagrames de classes estàtics
- No permet la generació de codi a partir de diagrames dinàmics
- És possible generar classes UML a partir de classes Java però la generació de diagrames d'aquestes classes es limita a les associacions d'herència i implementació.

2.2 ArgoUML des del punt de vista d'un programador

Aquesta secció pretén donar una breu visió d'ArgoUML des del punt de vista del desenvolupador, ressaltant la filosofia que es va seguir en el disseny d'aquesta eina (processos cognitius tinguts en compte, ús de codi obert i Java) i quines eines cal tenir per desenvolupar el codi. També es mencionen els mòduls que composen essencialment aquest programari, fent èmfasi en el mòdul encarregat de la generació automàtica de programari. No cal dir que es pot trobar informació molt detallada a la web citada anteriorment i en els documents indicats en ella.

2.2.1 Filosofia del desenvolupament d'ArgoUML

Com s'ha indicat a la secció anterior, un dels punts que diferencien ArgoUML d'altres eines CASE és l'èmfasi que fa aquesta eina en guiar el desenvolupador durant el procés de disseny dels models. ArgoUML pretén ser quelcom més que una pàgina en blanc per dibuixar diagrames. Per aconseguir aquest propòsit, els dissenyadors d'ArgoUML van estudiar els diferents processos que segueix un desenvolupador i van arribar a la conclusió que es poden

dividir bàsicament en tres: Reflexió en acció, Disseny oportunista i Comprensió i solució de problemes[14]. Seguidament descriurem breument aquests processos cognitius i com s'han plasmat en ArgoUML.

Reflexió en acció (*Reflection-in-action*)

Aquesta teoria cognitiva indica que, davant un sistema complex, els dissenyadors normalment afronten el problema començant per dissenyar esquemes més aviat grollers que es van refinant. D'aquesta manera, es van combinant fases de síntesi i anàlisi del problema, de manera que des del principi hi hagi un context en el qual es puguin prendre decisions (evitar el 'pànic al full en blanc').

ArgoUML ha tractat de plasmar aquesta teoria en el fet que proporciona les ja mencionades *to do lists* i *check lists*, les quals es van actualitzant a mesura que el disseny avança. Això permet el desenvolupador anar combinant fases de disseny amb fases d'anàlisi del disseny.

Disseny oportunista

Aquesta teoria descriu que els dissenyadors acostumen a planificar les tasques a fer per ordre d'importància però que, al final, les tasques es fan seguint un ordre de menys cost cognitiu. Això vol dir que si una tasca requereix cerca d'informació o un estudi en profunditat, segurament es relegarà per més tard, atacant primer una tasca que sigui més fàcil. A més a més, és també molt habitual que, durant el procés de resoldre un problema, certa informació pugui donar 'pistes' al programador per resoldre un altre problema, de manera que abandoni el problema inicial per fer una 'excursió' cap el segon problema i desprès retornar. Aquesta conducta no és possiblement la manera òptima de treballar, però aparentment és dóna en molts casos.

ArgoUML tracta d'ajudar els desenvolupadors que segueixen aquest patró cognitiu per mitjà de les llistes *to do*, de manera que es pugui, deixar una activitat i retornar després sense haver de recordar en quin punt exactament s'havia deixat aquesta activitat.

Comprensió i solució de problemes

Aquesta teoria descriu que els dissenyadors han de recórrer el camí que hi ha entre el model mental que tenen d'un problema i la solució formal d'aquest problema (com es fa palès aquest model en codi). Això s'aconsegueix mitjançant la representació de diversos aspectes del model mental i fent successius refinaments de la relació entre aquest model mental i el formal. Això va donar pas, de fet, a la implementació de llenguatges com el UML, que permetien especificar un mateix problema des de diversos punts de vista (transició entre estats, flux del control, flux de les dades).
En aquest sentit, ArgoUML pretén donar accés a diferents punts de vista del disseny mitjançant els diversos diagrames UML disponibles. En aquest punt, doncs, ja és el propi llenguatge UML el que implementa aquesta funcionalitat.

2.2.2 Programari lliure, Codi obert i Java

ArgoUML és un programari de codi obert, per tant qualsevol pot descarregar-se el codi font (en Java) i contribuir amb millores o simplement personalitzar-lo. La següent secció donarà els detalls pràctics de com fer això. El fet que sigui un programari lliure evidentment facilita la seva difusió i el poder tenir *feedback* de molts usuaris. Com a molts altres casos, aquest

programari ha derivat en una versió comercial (*Poseidon for UML*), com vam citar a al capítol 1.

Pel que fa al codi obert, l'objectiu és que molts desenvolupadors puguin contribuir al projecte sense cap cost. A més a més, com més gent col·labori. més fàcil serà trobar errors. D'altra banda, per aquells que contribueixen, és una manera d'aprendre i de donar-se a conèixer ('fer currículum') en un projecte que es farà servir per molta gent[15].

Finalment, l'ús de Java com a llenguatge es basa en el fet que escriure en Java significa arribar a molts més usuaris amb menys esforç. Com hem comentat al capítol 1, el fet que Java sigui un llenguatge interpretat (i no compilat) permet que es pugui utilitzar en qualsevol plataforma que disposi de Java Virtual Machine (JVM). Evidentment, el cost davant un llenguatge compilat és, com a mínim , una menor velocitat d'execució però, considerant els ordinadors actuals i la complexitat d'ArgoUML, l'elecció de Java és totalment encertada.

2.2.3 Descàrrega i desenvolupament

Tota la informació necessària es troba al document *Cookbook for Developers of ArgoUML*[16]. Bàsicament, cal tenir un client SVN i el JDK (a partir de la versió 1.4). El client SVN serveix per descarregar el fitxers fonts, de manera que es pot especificar si volem tots els arxius o només algun dels mòduls. D'aquesta manera, podem optar per descarregar una part i només fer canvis a aquesta part, fent servir els arxius compilats de la resta de mòduls. Un possible client SVN és Rapid-svn (http://rapidsvn.tigris.org/).

Pel que fa al JDK, és necessari per poder compilar el codi Java. ArgoUML té el seu propi constructor (equivalent a fer *javac*) basat en la utilitat *ant* de Apache[17]. Aquest programari permet compilar de manera estructurada i per mòduls, de manera que només calgui compilar el mòdul que s'ha canviat però alhora es tinguin en compte les interfícies amb els altres mòduls. Aquesta utilitat per tant no només fa una crida a javac, sinó que permet mantenir una estructura coherent del programari. Aquesta eina ja ve amb la descàrrega d'ArgoUML.

2.2.4 Estructura del codi

ArgoUML es troba dividit en paquets (*packages*) on s'agrupen les diferents funcionalitats del programa. Dins de cada paquet, trobarem les classes que formen l'aplicació. En aquesta secció presentarem els paquets i les seves classes més importants. La relació complerta de paquets i llibreries es pot trobar a la API (http://argouml-stats.tigris.org/nonav/reports-java5/javadocs/).

La següent figura mostra els paquets més importants que composen ArgoUML: *Kernel, Application, UI, UML, Cognitive i Language* (Figura 10). Aquesta secció comentarà breument les característiques mes interessants d' aquests paquets excepte del paquet Language, al qual se li dedicarà la secció 2.2.5.

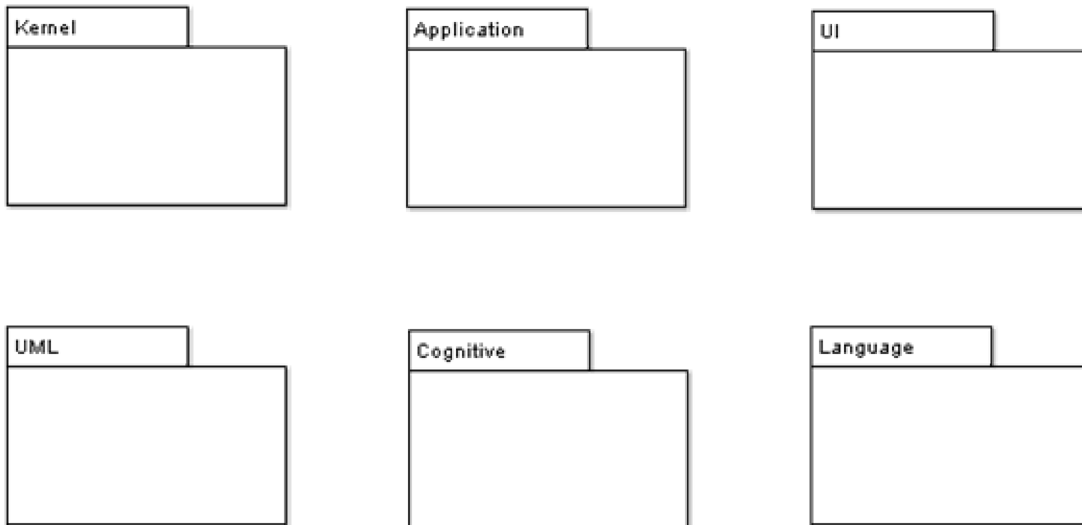

Figura 13 *Paquets principals que composen ArgoUML*

Kernel

Aquest package conté el cor de l'aplicació: la classe *Project*. Aquesta classe guarda tots els elements que formaran el nostre projecte (classes, diagrames). Per altra banda, tenim la interfície ProjectMember que representa un element (abstracte) que pot formar part d'un projecte. Una implementació és *AbstractProjectMember*, del qual derivaran classes com *ProjectMemberDiagram*, *ProjectMemberModel* o *ProjectMemberToDoList*. Finalment, destaquem la classe *ProjectManager*, classe que dirigeix el projecte i que només es pot instanciar un cop (singleton).

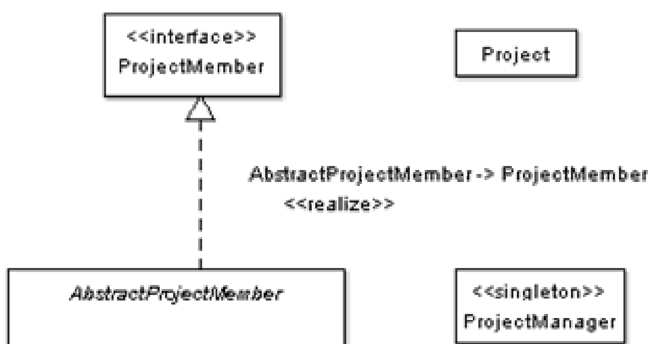

Figura 14 *Components principals del paquet Kernel*

Application

El paquet *Application* té com a classe principal la classe *Main* (representada amb detall a la figura 15). Com el seu nom indica, aquesta és la classe que primer s'executa a l'iniciar l'aplicació. Fa tasques com inicialitzar la interfície gràfica, comprovar que la versió de la

JVM és la correcta i inicialitzar els subsistemes. És interessant indicar que en aquest model s'inicialitza el generador de codi (*GeneratorJava.getInstance()*).

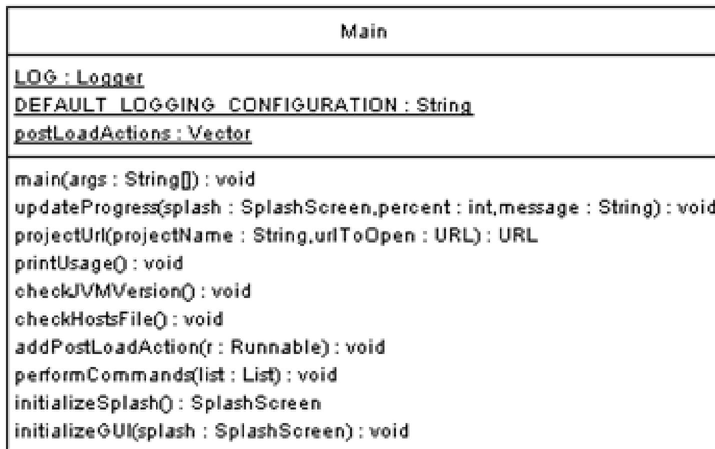

Figura 15 *Detall de la classe Main, component principal de Application*

UI (User Interface)

La finestra principal d'ArgoUML és implementada per *ProjectBrowser* (que és una classe *final* en Java, o sigui, *Singleton*). Aquesta classe té com atributs els altres panells que composen la interfície gràfica (i que hem citat ja en l'apartat sobre el punt de vist d'un usuari): la barra de situació, el panell de detalls, el panell navegador.

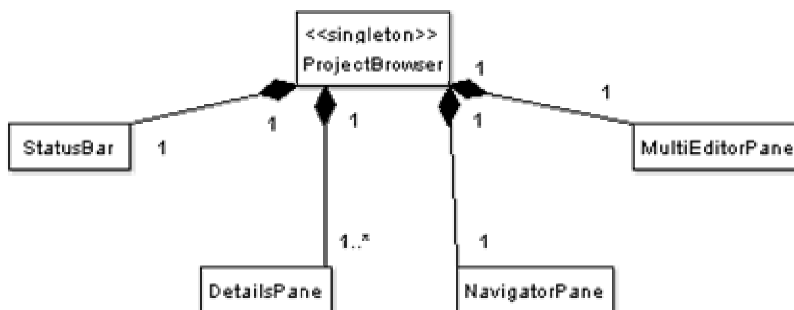

Figura 16 *Principals components del paquet UI (User Interface)*

UML

Aquest paquet conté les classes que implementen els elements UML i és, respecte a funcionalitat, el paquet central d'ArgoUML. Aquest paquet es divideix en altres subpaquets, tres dels quals, els més importants, es representen a la figura següent:

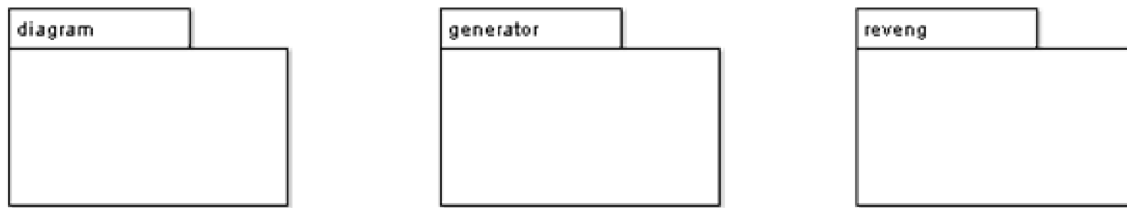

Figura 17 *Subpaquets principals que composen el paquet UML*

El paquet *Diagram* es troba a la seva vegada dividit en paquets que representen cadascun des diagrames UML disponibles (Figura 18). Dins aquests paquets trobem la classe que implementa aquest paquet i d'altre classes auxiliars que implementen la interfície gràfica.

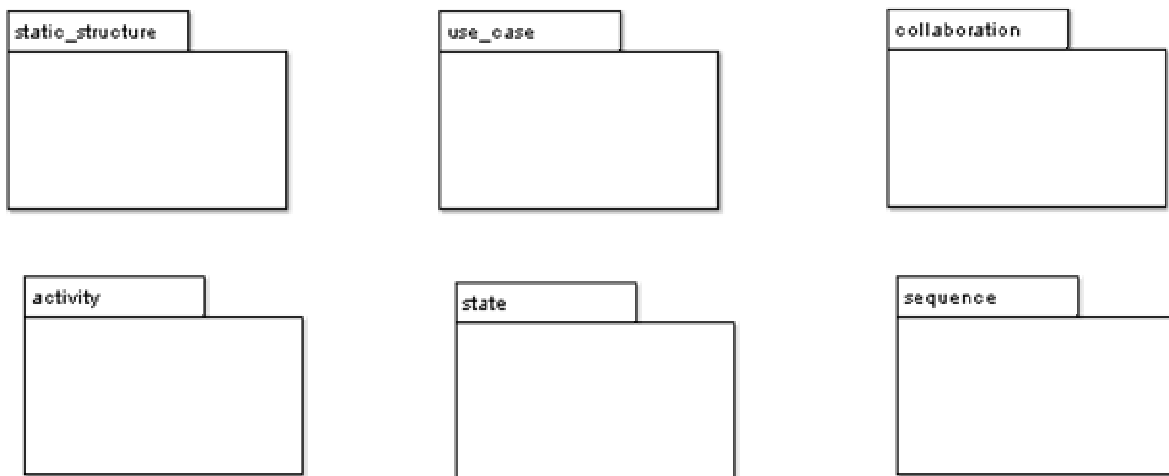

Figura 18 *Components principals del subpaquet Diagram (package UML)*

Respecte al subpaquet *Generator*, l'element més important que conté és la interfície *CodeGenerator*. Aquesta interfície porta els mètodes generals que qualsevol generador de codi hauria d'implementar i és la base de *GeneratorJava*, classe que tractarem amb detall a la pròxima secció.

Aquesta interfície especifica tres mètodes:

- *generate*: genera codi a partir de una entrada de tipus Collection. Té també un paràmetre d'entrada de tipus booleà (*deps*) que indica si cal generar codi pels elements continguts.
- *generateFiles* i *generateFileList*: generen fitxers i llistes de fitxers on es guardarà el codi. Tenen els mateixos paràmetres d'entrada que *generate*.

Figura 19 *Detall de la interfície CodeGenerator, la qual implementaran el generador de codi Java (GeneratorJava)*

Aquests tres mètodes tenen com paràmetres d'entrada un *Collection* (elements) i un *boolean* (deps). El paràmetre de tipus Collection fa referència a l'element UML en qüestió, sense especificar quin ja que es tracta d'una interfície. Com veurem a la secció 2.2.5, la implementació que fa *GeneratorJava* restringeix aquest *Collection* a classes o interfícies (i ho especifica com de tipus *Object*). Respecte al paràmetre booleà, representa si cal generar recursivament els elements que puguin dependre d'aquest. En la implementació que fa *GeneratorJava* veurem que aquest paràmetre no es fa servir.

Finalment, el paquet *reveng* conté un subpaquet anomenat *java* amb les classes que faciliten l'enginyeria inversa entre Java i UML La classe *JavaImport* és la classe principal i fa servir les classes *JavaLexer, JavaRecogn*izer per analitzar el fitxer java (*parse*) i la classe *Modeller* per transformar el resultat de l'anàlisi en classes corresponents al model UML (Figura 20).

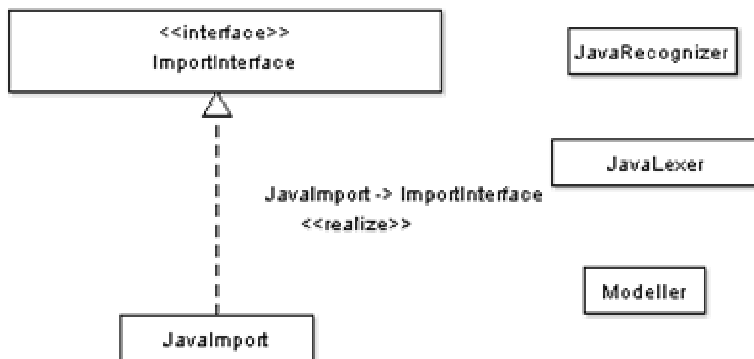

Figura 20 *Components principals del subpaquet Reveng*

Cognitive

El paquet *Cognitive* conté les classes que implementen els elements cognitius d'ArgoUML: les *To Do lists* i les *Check lists*. Seguint la metodologia d'orientació a objectes, hi ha una classe que representa el desenvolupador (*Designer*) i que implementa els mètodes que el desenvolupador farà servir per aprofitar els elements cognitius d'ajuda al disseny. Les altres classes representades a la figura representen altres dels elements cognitius més importants (Figura 21).

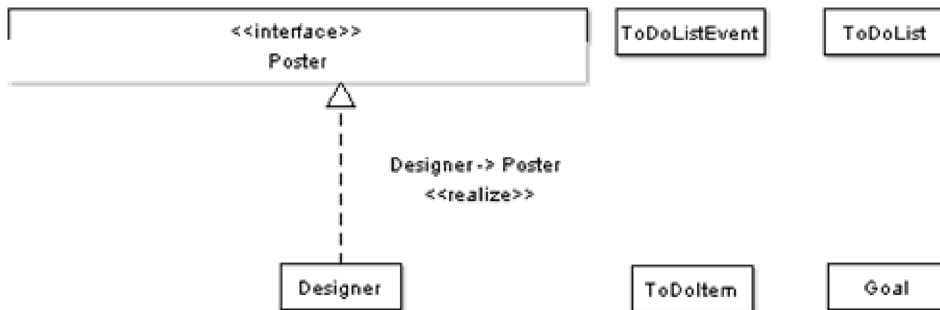

Figura 21 *Components principals del paquet Cognitive*

2.2.5 Generació automàtica de programari

Com es va indicar, el paquet Language és el que conté les classes necessàries per fer la generació automàtica de codi a partir del diagrama de classes. Aquest paquet conté el subpaquet *Java* i aquest a la seva vegada el subpaquet *Generator*. D'aquest paquet, la classe més important és *GeneratorJava*, ja que conté tots els mètodes per generar codi a partir de diagrames.

La figura 22 mostra la classe *GeneratorJava* i les seves relacions més importants amb d'altres classes. *GeneratorJava* implementa la interfície *CodeGenerator*, citada anteriorment. La classe *ClassGeneratioDialog* (que pertany al paquet UML) fa la crida a *GeneratorJava* i li passa com a argument un element del model UML. Aquest element que *GeneratorJava* rep és de la classe *Object* i per tant li queda la tasca d'esbrinar quin tipus d'objecte és i quins components té. Això ho aconsegueix fent servir la classe *Model*. Aquesta classe disposa del mètode *static getFacade* que retorna en realitat un *ModelImplementation.getFacade* (això serveix per amagar la veritable implementació del model UML que fa *MDRModelImplementation* de l'especificació més general que té Model). Aquest mètode retorna un objecte *FacadeMDRImpl* (que és una implementació de la interfície *Facade*). Aquesta classe té els mètodes necessaris per reconèixer els elements que composen l'objecte inicial (pot dir si és una classe o interfície, si té atributs o operacions i de quin tipus són, etc.).

En definitiva, la classe GeneratorJava rep un objecte del model UML i comença a cridar a Facade per demanar-li tota la informació que pugui tenir sobre l'element en qüestió. En la implementació actual d'ArgoUML, només es genera codi si aquest element és un classe o una interfície.

La figura 23 mostra les operacions que *GeneratorJava* executa. *ClassGenerationDialog* crida el mètode *GeneretorJava.generateFile,* que genera un fitxer amb el nom de la classe o interfície i comença a cridar els altres mètodes. L'estratègia que segueix és la següent: anar seguint l'estructura que té un fitxer Java (primer el *header,* després els *imports*, atributs, mètodes), de manera que cada mètode fa les 'preguntes' adients a l'objecte Facade per tal de saber si té propietats que es puguin traduir en codi Java. Cada mètode retorna un *String* ja amb el format de codi Java i finalment tots aquests strings s'afegeixen al fitxer creat inicialment.

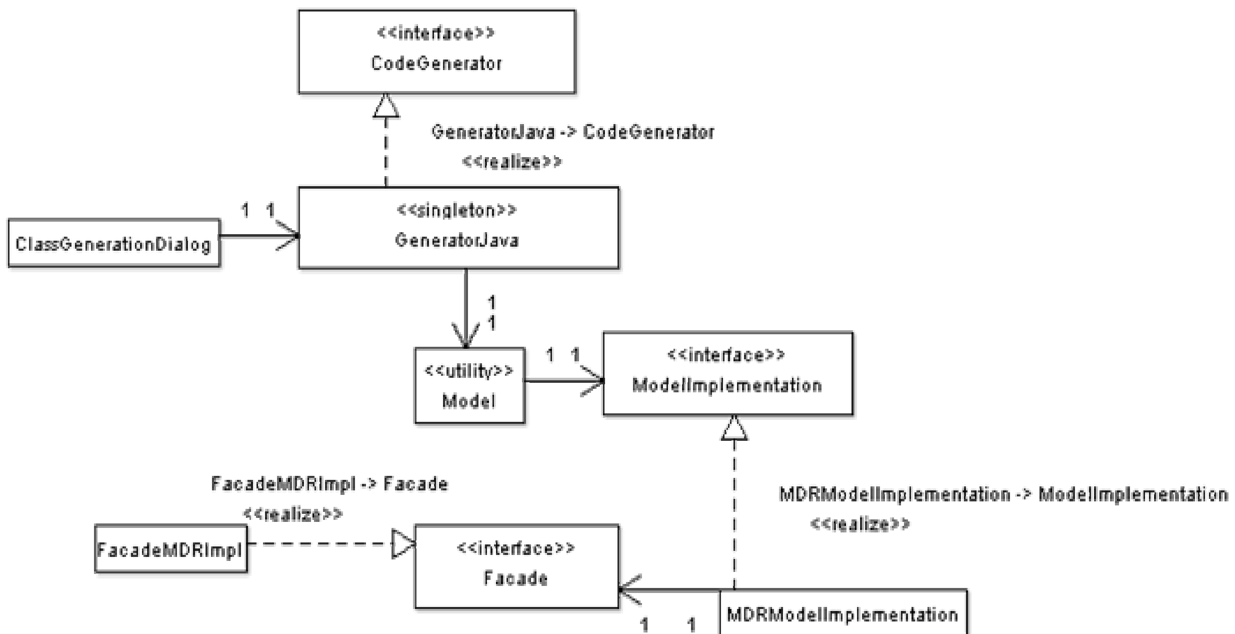

Figura 22 *Diagrama de classes que mostra la classe GeneratorJava i les seves associacions*

La Taula 1 mostra de manera resumida de quins elements s'encarrega cada mètode de *GeneratorJava* (la qual cosa es pot endevinar fàcilment del nom del mètodes, en qualsevol cas). Tots aquests mètodes tenen com a paràmetre d'entrada un *Object* i com a sortida un *String* o *StringBuffer*

Taula 1 *Mètodes de GeneratorJava que intervenen en la generació de codi a partir del diagrama de classes*

Mètodes de GeneratorJava	***Elements de codi Java que genera***
GenerateFile	nom del fitxer
getPackageName	extreu el nom del *package*
generateHeader	escriu el nom del package i crida a generateImports
generateImports	genera els *imports*
generateClassifier	genera l'estructura principal del fitxer i crida els altres mètodes 'Classifier'
generateClassifierStart	Genera la primera línia que defineix el fitxer: [class|interface][nom][visibility][extends|implements]
generateClassifierBody	Afegeix atributs i mètodes
generateClassifierEnd	Afegeix l'últim '}'

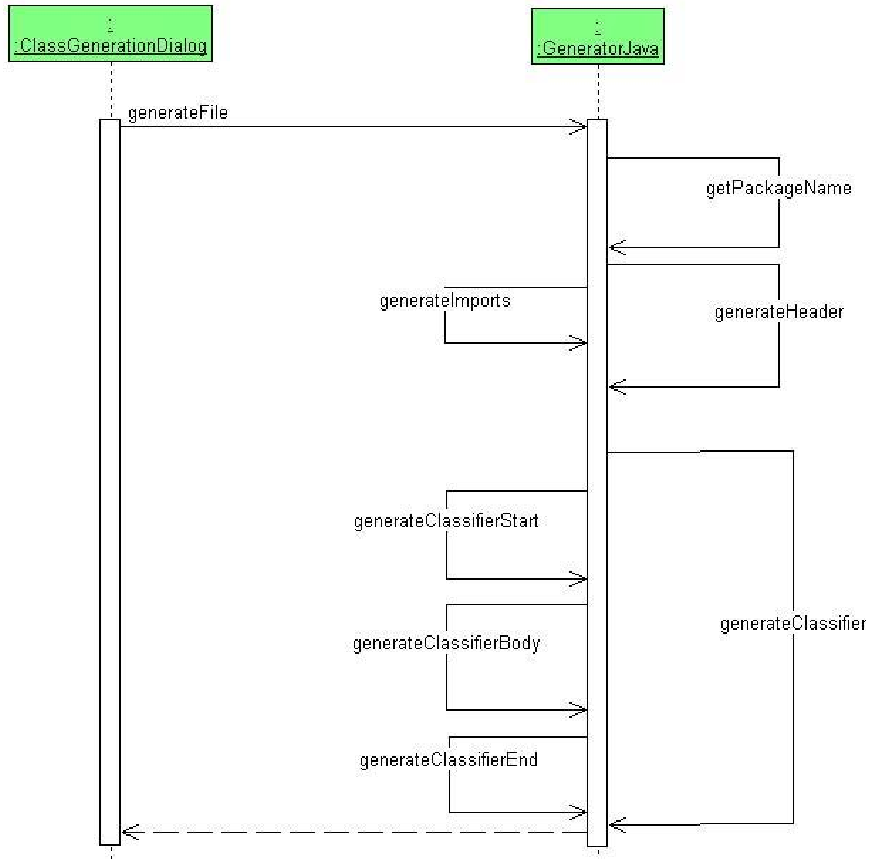

***Figura 23** Diagrama de seqüència que representa el recorregut típic de la generació de codi dins de GeneratorJava*

3. Millores en la generació de codi Java a partir de diagrames de classes UML amb ArgoUML: descripció, discussió i implementació

En general, un primer objectiu de la generació de codi a partir de diagrames UML hauria ser assolir la mateixa feina que fan els actuals precompiladors i altres eines de generació de codi: aconseguir implementar l'esquelet del codi d'una classe de manera precisa i a prova d'errades. Ningú es planteja avui en dia revisar si el codi generat per un precompilador de C ha interpretat bé una instrucció *#def*, per exemple. Els generadors de codi a partir de diagrames UML encara no han assolit la reputació suficient i per tant el primer pas hauria de ser aconseguir aquest nivell. Pel futur quedaria llavors la implementació del codi sencer (a partir de diagrames d'estats de cada classe, potser?).

L'objectiu d'aquest capítol és justificar els canvis que es podrien introduir en ArgoUML per millorar la generació de l'esquelet del codi d'una classe a partir del diagrama de classes i mostrar com es poden implementar en el codi font d'ArgoUML. Es tracta de canvis certament no crítics però que poden ajudar a millorar la qualitat del codi generat. Les millores introduïdes les hem dividit en dues classes:

- Correcció d'errors que presenta actualment ArgoUML: cardinalitat d'atributs i associacions múltiples entre classes sense *rolename* especificat
- Introducció de funcionalitats extra: mètodes *get* i *set* (tant per atributs primitius com per atributs derivats d'associacions), i *headers* amb informació addicional.

Amb aquesta TFC s'adjunta el codi Java generat (tot a la classe original *GeneratorJava*) i el document generat amb *javadoc* amb comentaris addicionals. Els detalls d'implementació no tenen en si mateix cap valor extraordinari i per tant només es citarà en el text en quines funcions s'han afegit els canvis i alguna característica general de l'estratègia que s'ha seguit. En qualsevol cas, els exemples de codi generat que es presentin hauran estat per la versió original o adaptada d'ArgoUML.

3.1 Correcció d'errors en la generació de codi d'ArgoUML

Al capítol 2 ja vam esmentar que hi havia un parell d'errors en la generació de codi d'ArgoUML que es podien corregir. Cap d'ells és realment crític (el codi generat compila) però es podria millorar el codi generat reparant-los. En concret, els dos errors són:

- Cardinalitat dels atributs, tant de tipus primitiu (*int*, *char*, etc ...) com d'atributs provinents d'associacions
- Associacions múltiples entre classes quan no hi ha *rolenames*

3.1.1 Cardinalitat dels atributs

Descripció

UML permet mencionar la cardinalitat d'un atribut (tant primitiu com derivat d'una associació) amb la següent notació:

Taula 2 *Notació de la cardinalitat en UML*

Notació	***Significat***
1	un valor
0..1	un o cap valor
n	n valors
0..*	cap o algun valor
1..*	un valor com a mínim

Apart del casos trivials de *1* o *0..1*, el cas que inclou * és implementat per ArgoUML amb la classe *Vector*. Aquesta classe implementa les interfícies *Collection* i *List* i extend la classe *AbstractList*. Bàsicament, permet mantenir una llista d'elements (no necessàriament tots del mateix tipus) de longitud variable, de manera que no cal especificar un límit. Conté mètodes per afegir elements (al final o a una certa posició), retornar elements, longitud de la llista, etc. És per tant una elecció molt encertada (altres possibilitats com *HashList* també serien bones però tampoc aportarien cap funcionalitat extra).

En canvi, si s'especifica amb ArgoUML una cardinalitat definida, el codi generat no mostra el que s'espera. En l'exemple ja esmentat al capítol 2 veiem:

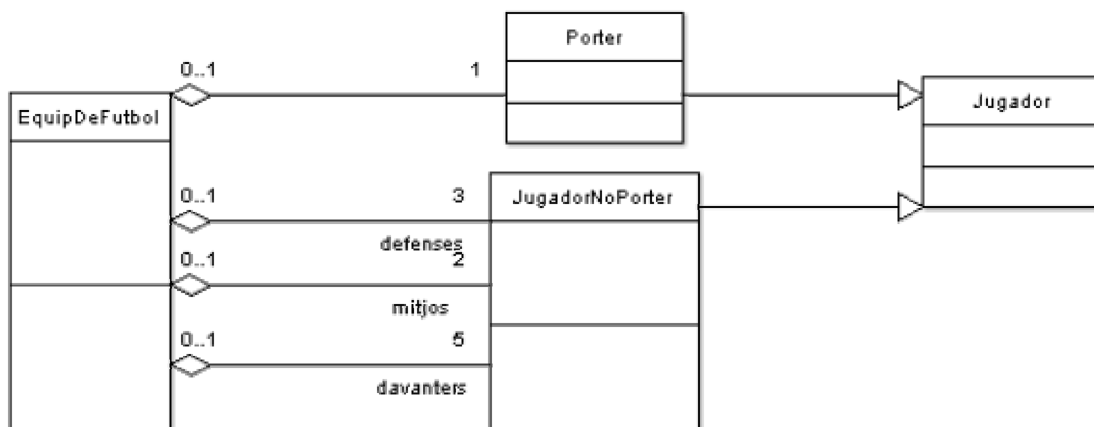

Figura 24 *Diagrama de classes amb associacions de cardinalitat múltiple*

Repetim el codi generat per la classe *EquipDeFutbol*:

EquipDeFutbol.java

```
import java.util.Vector;
public class EquipDeFutbol {
   public Porter myPorter;
   /**
   *
   * @element-type JugadorNoPorter
```

```
  */
 public Vector  mitjos;
  /**
  *
  * @element-type JugadorNoPorter
  */
 public Vector  davanters;
  /**
  *
  * @element-type JugadorNoPorter
  */
 public Vector  defenses;
}
```

Els atributs de tipus JugadorNoPorter porten una cardinalitat concreta però el codi generat no ho mostra. Per implementar això s'ha optat per substituir la classe Vector per un array en aquells casos de cardinalitat definida (també s'han eliminat els comentaris @*element*), de manera que ara el codi generat desprès de les modificacions és:

```
public class EquipDeFutbol {
          public Porter myPorter;
         public JugadorNoPorter[] mitjos = new JugadorNoPorter[2];
         public JugadorNoPorter[] davanters = new JugadorNoPorter[5];
         public JugadorNoPorter[] defenses = new JugadorNoPorter[3];
}
```

L'elecció d'un *array* es justifica pel fet que la longitud de l'*array* ja ha estat definida al diagrama UML i per tant el codi hauria d'evitar que aquest *array* pugui créixer. Evidentment, un dels inconvenients de fer servir un array és que no podem fer servir els mètodes de la classe Vector, però així també impedim que es puguin afegir més valors dels esperats. En canvi, per cardinalitat indeterminada, fer servir un *array* no és una bona solució. En principi podríem declarar la variable com *public JugadorNoPorter[] mitjos* però, abans de fer servir aquest atribut haurem de determinarà quants elements ha de tenir l'*array* i per tant ens trobarem amb un problema. En definitiva, deixem la classe *Vector* com implementació de cardinalitats indefinides i fem servir un *array* si la cardinalitat és definida.

Aquests canvis també s'han afegit pel cas d'atributs primitius (on tampoc s'implementava la cardinalitat).

Implementació

La implementació d'aquests canvis es pot trobar en les funcions *generateCoreAttribute* (pels atributs primitius) i en *generateCoreAssociationEnd* (pels atributs provinents d'associacions). Bàsicament consisteix en llegir la multiplicitat i decidir si ens trobem en un cas de cardinalitat definida, indefinida o simple (*1*).

3.1.2 Associacions múltiples sense paper (*rolename*)

Descripció

Seguint amb l'exemple anterior, el codi de la classe *JugadorNoPorter* en la versió original d'ArgoUML és el següent:

JugadorNoPorter.java

```
public class JugadorNoPorter extends Jugador {
   public EquipDeFutbol myEquipDeFutbol;
   public EquipDeFutbol myEquipDeFutbol;
   public EquipDeFutbol myEquipDeFutbol;
}
```

Veiem que aquesta classe incorpora tres cops *EquipDeFutbol* com atribut. Això és innecessari ja que, des del punt de vista de JugadorNoPorter, només pot estar relacionat amb un EquipDeFutbol. Això no és més que un error en ArgoUML, que recorre totes les associacions i les assigna un atribut, sense mirar si aquesta associació té realment un paper (*rolename*).

De totes maneres, es podria també raonar que no té sentit que es tingui una relació sense *rolenames* i per tant, podríem dir que ArgoUML no hauria de permetre aquestes associacions.

Implementació

Respecte la implementació, els canvis s'han afegit a *generateClassfierBody*. El codi modificat conté una nova classe *Vector* on es van guardant els *Strings* de codi provinents d'associacions, de manera que es comprova si ja s'ha afegit i en tal cas es descarta.

3.2 Noves funcionalitats afegides a ArgoUML

En aquest apartat descriurem les noves funcionalitats que s'han afegit a ArgoUML que poden millorar el codi generat per aquest programa. Bàsicament es tracta d'afegir mètodes *set* i *get* tant per atributs primitius com per atributs provinents d'associacions i afegir un encapçalament amb informació addicional.

3.2.1 Mètodes get i set

Descripció

UML permet definir els atributs com a *public*, *private* o *protected* però no defineix què signifiquen aquests modificadors sinó que ho deixa obert a la interpretació que en facin els llenguatges de programació (de fet, si un llenguatge en té d'altres modificadors, també es poden mencionar en un diagrama UML). Si prenem el llenguatge Java, tenim que el significat és:

- *public*: accessible des de qualsevol altre classe
- *protected*: només accessible des de la pròpia classe o des de subclasses
- *private*: només accessible des de la pròpia classe

És una bona pràctica definir els atributs com a *protected* i després definir mètodes per accedir a aquestes dades (mètodes *get* i *set*). Així aquestes dades estan protegides i podem implementar condicions addicionals per poder obtenir o canviar un atribut.

En el cas d'atributs primitius (com *int* o *String*) aquests mètodes són força simples. Per exemple, per l'exemple de la classe *Complex*, tindríem que el codi generat amb la nova versió d'ArgoUML seria:

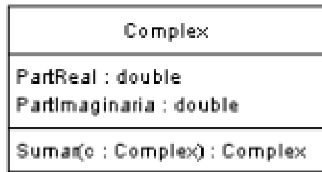

Figura 25 *Exemple d'un classe amb atributs primitius*

```
public class Complex {
  private double PartReal;
  private double PartImaginaria;
  public void set_PartReal(double  element) {
    PartReal = element;
  }
  public double  get_PartReal(){
    return PartReal;
  }
  public void set_PartImaginaria(double  element) {
    PartImaginaria = element;
  }
  public double  get_PartImaginaria(){
    return PartImaginaria;
  }


  public Complex Sumar(Complex c) {
  return null;
  }
}
```

En canvi, si volem fer el mateix amb un atribut provinent d'una associació, hem de tenir en compte algunes particularitats de les associacions. Una associació en UML representa una referència que un element posseeix d'un altre. En aquest sentit, és similar al concepte de punter però amb la diferència que tots dos elements són conscients de la relació (excepte si només és navegable en un sentit). Aquest concepte ha estat comparat amb una relació de matrimoni[18]. Les associacions UML es descriuen en aquesta referència com *ABACUS*, el que significa:

- ***A****wareness* (Consciència): els dos objectes són conscients de la relació.
- ***B****oolean existence* (Existència booleana): si tots dos estan d'acord en terminar la relació, l'associació desapareix.
- ***A****greement* (Acord): tots dos objectes han d'acceptar la relació.
- ***C****ascaded deletion* (Supressió en cascada): si un dels objectes desapareix, la relació també ho fa.

- *USe of rolenames* (Ús de papers): un objecte es pot referir a l'altra mitjançant el seu paper.

En els següents apartats veurem com aquestes característiques s'haurien de plasmar en el codi Java generat. A la figura 26 veiem un exemple d'associació un-a-un amb els mètodes get i set.

***Figura 26** Exemple d'una associació un-a-un entre classes*

La implementació del codi del mètode get no és gaire sorprenent:

```
public Dona getDona( ) {
        return myDona;
    }
```

Tanmateix, la implementació del mètode *set* no és tan òbvia. En el cas del mètode *setDona*, primer hem de veure si *Home* ja tenia una dona. En aquest cas, haurem d'informar a *l'ex-dona* que ha d'eliminar a *Home* de la seva relació[i]. Això requerirà una operació especial (`internalRemoveFromMyHome`) que només pot ser cridada des de *setDona*. Després haurem d'assignar la nova *Dona* a *Home* i fer saber a la nova Dona que aquest és el seu *Home*. En resum, el codi que s'hauria d'implementar seria:

```
public void setDona(Dona element) {
        if ( this.myDona != element ) {     // prevenir un bucle infinit
            if ( this.myDona != null ) {     // ja té una dona!!
                // eliminar el link de la ex-dona
                this.myDona.internalRemoveFromMyHome( (Home)this );
            }
            this.myDona = element;        // nou valor de myDona
            if ( element != null ) {
                // la nova Dona ha de tenir un link a Home
                element.setMyHome( (Home)this );
            }
        }
    }

public void internalRemoveFromMyHome(Home element) {
        this.myHome = null;
    }
```

Si, en canvi, tenim cardinalitats diferents de 1, la situació canvia. Agafem per exemple la situació descrita a la figura 27:

***Figura 27** Exemple d'associació un-a-molts*

[i] També es podria escollir rebutjar formar una nova relació si ja se'n té una de prèvia.

El codi per la classe *Fill* no serà molt diferent del codi que hem mostrat abans per *Home/Dona*. El mètode *get* serà idèntic, i el cos del mètode set serà també semblant, amb l'única variació en el mètode corresponent a la *Mare* per afegir el *Fill*. En concret, tindrem:

```
public void setMare(Mare element) {
          if ( this.myMare != element ) {     // prevenir un bucle infinit
               if ( this.myMare != null ) {     // ja té una Mare!!
                    // eliminar el link de la ex-Mare
                    this.myMare.internalRemoveFromMyFill( (Fill)this );
               }
               this.myMare = element;        // nou valor de myMare
               if ( element != null ) {
                    // la nova Mare ha de tenir un link a Fill
                    element.internalAddToMyFill( (Fill)this );
               }
          }
     }
```

Veiem que la diferència es troba en què *Mare* ha de fer servir una operació especial (*internalAddToMyFill*). Aquesta operació només pot ser cridada des de l'operació *set* de la part de cardinalitat múltiple i és diferent la simple operació set que es va mostrar en el cas un-a-un. A més a més, la classe *Mare* ha de tenir una sèrie d'operacions apart de set per tal de tenir en compte diferents situacions degudes a la cardinalitat. En general, *Mare* tindrà les següents operacions:

- una operació get que retorna la col·lecció
- una operació *set* que tingui com a paràmetre d'entrada una col·lecció
- una operació *add* amb un únic paràmetre, tal i com hem vist abans
- una operació *add* amb una col·lecció de paràmetres
- tres operacions *remove:* una amb un paràmetre, una altra amb una col·lecció com a paràmetre i la tercera sense paràmetres i que esborraria tota la col·lecció
- una operació interna per afegir un element, com hem vist abans

A continuació detallem el codi de cada operació amb comentaris per entendre el propòsit:

```
public List getMyFill() {
            if ( myFill != null ) {
                  //retornem una còpia 'read-only' de la col·lecció
                  return Collections.unmodifiableList(f_myFill);
} else {
                  return null;
            }
      }



public void setMyFill(Vector elements) {
            if ( this.myFill != elements ) {
                  //eliminem el link entre els Fills i les seves mares
                  Iterator it = this.myFill.iterator();
                  while ( it.hasNext() ) {
                        Fill x = (Fill) it.next();
                        x.z_internalRemoveFromMyMare( (Mare)this );
                  }
                  //Assignem els nous fills a la Mare
                  this.myFill = elements;
                  if ( myFill != null ) {
                        //Creem links entre els nous fills i aquesta Mare
```

```
                        it = myFill.iterator();
                        while ( it.hasNext() ) {
                              Fill x = (Fill) it.next();
                              x.z_internalAddToMyMare( (Mare)this );
                        }
                  }
            }
      }


public void addToMyFill(Man element) {//Afegim un únic element
                if ( element == null ) {
                        return;
                }
                //Si ja té aquest element, no cal afegir-lo
                if ( this.myFill.contains(element) ) {
                        return;
                }
                this.myFill.add(element);
                if ( element.getMyMare() != null ) {
        element.getMyMare().internalRemoveFromMyFill(element);
                }
                element.internalAddToMyMare( (Mare)this );
        }



public void addToMyFill(Collection newElems) { //Afegim un conjunt de Fills
                Iterator it = newElems.iterator();
                while ( (it.hasNext()) ) {
                        Object item = it.next();
                        //Comprovem que realment l'argument es un Fill
                        if ( item instanceof Fill ) {
                                this.addToMyFill((Fill)item);
                        }
                }
        }




public void removeFromMyFill(Fill element) {
                if ( element == null ) {
                        return;
                }
                this.myFill.remove(element);
                element.internalRemoveFromMyMare( (Mare)this );
        }

public void removeFromMyFill(Collection oldElems) {
                Iterator it = oldElems.iterator();
                while ( (it.hasNext()) ) {
                        Object item = it.next();
                        if ( item instanceof Fill ) {
                                this.removeFromMyFill((Man)item);
                        }
                }
        }

public void removeAllFromMyFill() {
```

```
                //Ens fem una còpia per evitar una Exception
                Iterator it = new Vector(getMyFill()).iterator();
                while ( (it.hasNext()) ) {
                        Object item = it.next();
                        if ( item instanceof Fill ) {
                                this.removeFromMyFill((Fill)item);
                        }
                }
        }

public void z_internalAddToMyFill(Fill element) {
          this.myFill.add(element);
     }
```

Finalment hem de tenir en compte el cas en d'una associació en què totes dues classes tinguin cardinalitat més gran que 1. La majoria dels mètodes són idèntics als que hem vist anteriorment pel cas *Mare-Fill* (associació una-a-molts). En realitat, l'única diferències troba en el mètode d'afegir serà lleugerament diferent ja que no caldrà trencar el link ja que es poden tenir més d'una relació.

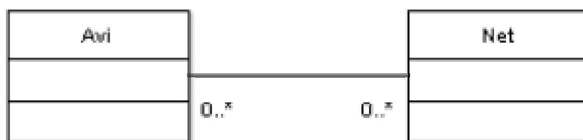

Figura 28 *Exemple de relació molts-a-molts*

```
public void addToMyNet(Net element) {
          if ( element == null ) {
                return;
          }
          if ( this.Net.contains(element) ) {
                return;
          }
          this.Net.add(element);
          element.internalAddToMyAvi( (Avi)this );
     }
```

Implementació

Per la generació de codi corresponent a atributs primitius, un nou mètode ha estat creat: *generateSetGetOperations*. Aquesta funció rep una *Collection* amb els atributs de la classe. A partir d'aquí, només cal esbrinar quin nom té l'atribut i quin tipus de dada és. Aquest mètode retorna un *StringBuffer* que després s'afegeix al codi.

La implementació del codi pels atributs derivats s'ha implementat amb el mètode *generateSetGetOperationsAssociations*. Aquest mètode rep com a paràmetres una *Collection* dels atributs i la pròpia classe com un *Object*. Amb els paràmetres podem crear un iterador i recórrer tots els atributs. Llavors, segons la cardinalitat dels atributs i de la pròpia classe, podem decidir quin codi generarem.

En aquesta implementació, hem optat però no diferenciar entre atributs amb cardinalitat determinada més gran que 1 i cardinalitat indeterminada per no complicar el codi generat, de manera que tots aquests casos s'implementen amb la classe *Vector*.

3.2.2 Header

Descripció

A l'encapçalament del fitxer podem afegir informació extra, com per exemple la data en que es va generar:

```
/*This file was generated on 14-may-2007 */
```

Implementació

En el mètode generateHeader, hem generat un String amb la data a partir de les classes *Date* i *DateFormat*.

4. Conclusions

Les conclusions principals d'aquest treball són:

- El cicle de vida del software tradicional presenta sovint grans problemes en quant a la sincronització de codi i documentació, portabilitat i interoperabilitat.
- MDA intenta superar aquests problemes fent dels models el centre del desenvolupament de software. Aquests models inclouen models que no depenen de cap plataforma en particular (PIM) i models que recullen les característiques peculiars de la plataforma on el software es farà servir (PSM). MDA preveu una generació automàtica entre PIM i PSM i entre PSM i codi
- Les característiques de MDA fan que UML, Java puguin jugar un paper molt important. UML és el llenguatge de modelatge per excel·lència, mentre que Java implementa el paradigma d'Orientació a Objectes i millora la portabilitat amb la JVM.
- La generació de codi, tant dins MDA com en si mateixa, presenta una sèrie d'avantatges com la millora en la qualitat del codi, la consistència arquitectònica i el fet d'estalviar temps que es pot emprar en el disseny.
- ArgoUML és una eina CASE que permet crear models UML i la generació de codi Java a partir de diagrames de Classes. Aquesta eina presenta una sèrie d'avantatges que la fan força atractiva pels usuaris: és de llicència gratuïta, fàcil de fer servir i amb bones prestacions.
- Pels programadors, ArgoUML és una eina interessant ja que incorpora en el seu disseny un estudi cognitiu de com es desenvolupa programari. Aquesta és una eina de codi obert, de manera que es pot estudiar com es composta i, sobre tot, permet canviar el codi font per tal d'afegir-li extensions.
- El codi Java generat per ArgoUML es podria millorar en alguns punts (sobre tot pel que fa al tractament de les cardinalitats). També se li podrien afegir més funcionalitats, com la implementació de mètodes *set* i *get* per atributs tan primitius com derivats d'associacions.
- Aquesta memòria mostra que afegir aquests mètodes *set* i *get* per atributs provinents d'associacions, especialment amb cardinalitat múltiple, suposa generar codi que ha de tractar casos especials en quant a l'establiment de relacions entre Classes.
- Finalment, s'explica com implementar aquests canvis en el codi font d'ArgoUML i demostra per tant la versatilitat d'aquesta eina i els reptes que suposa la generació automàtica de codi amb certa qualitat.

5. Glossari

ArgoUML
Eina de software lliure i codi obert que permet crear models UML i generar codi automàticament

JVM
Java Virtual Machine: conjunt de software i estructura de dades que interpreta Java bytecode (el codi generat en compilar un programa *.java*) i que executa una aplicació en Java.

MDA
Model Driven Architecture: mètode de desenvolupament de software on l'objecte principal són els models i que pressuposa la generació automàtica tant de models derivats com de codi.

OMG
Object Management Group: organització d'empreses informàtiques que fomenta l'utilització de programació Orientada a Objecte, UML i MDA, entre d'altres.

OO
Object Oriented: paradigma de programació que fa servir objectes per tal d'aconseguir encapsulació, herència, modularitat i polimorfisme.

PIM
Platform Independent Model: en MDA, model que representa la funcionalitat d'un software en qüestió sense dependre d'una plataforma (llenguatge de programació, base de dades, sistema operatiu) específica.

PSM
Platform Específic Model: en MDA, model que representa la funcionalitat d'un software en qüestió per una determinada plataforma.

TFC
Treball de Fi de Carrera

UML
Unified Modeling Language: Llenguatge per descriure software en forma de models, típicament associat al paradigma d'orientació a Objectes.

6. Bibliografia

[1] **A. Kleppe, J. Warner, W. Bast**. (2003) *MDA Explained*. Addison-Wesley

[2] http://www.omg.org/mda/

[3] **J. Miller, J. Mukerji**. (2003) *MDA Guide version 1.0.1*. http://www.omg.org/docs/omg/03-06-01.pdf

[4] **M. Fowler, K. Scott**. (1999) *UML Distilled*, Addison-Wesley

[5] http://www.uml.org/

[6] **Booch G., J, Rumbaugh, I. Jacobson**. (1999) *El lenguaje unificado de modelado*. Addison-Wesley.

[7] **Flannagan, D**. (2002) *Java in a Nutshell*. O'Reilly & Associates, Inc.

[8] **Herrington, J**. (2003) *Code generation in action*. Manning.

[9] http://www.codegeneration.net/

[10] http://argouml.tigris.org/

[11] http://www.gentleware.com/

[12] http://uml.sourceforge.net/index.php

[13] **M. van der Wulp**. (2007) *ArgoUML User Manual*. http://argouml-stats.tigris.org/documentation/manual-0.24/

[14] **J. E. Robbins**, *Cognitive Support Features for Software Development Tools*. (1999). Doctoral Dissertation, University of California, Irvine. http://argouml.tigris.org/docs/robbins_dissertation/diss.pdf

[15] http://en.wikipedia.org/wiki/Open_source_versus_closed_source

[16] **L. Tolke, M. Klink, M. van der Wolpe**. (2007) *Cookbook for Developers of ArgoUML*. http://argouml-stats.tigris.org/documentation/defaulthtml/cookbook/

[17] http://ant.apache.org/

[18] **A. Kleppe, J. Warner**. (2005) *Wed Yourself to UML with the Power of Associations* http://www.devx.com/enterprise/Article/28528