Metodologies de Desenvolupament Dirigides per Models

Ferran Rodenas Barceló
ETIS

Anna Queralt Calafat

18 de Juny de 2007

Resum

En el transcurs de les dos últimes dècades, s'han fet esforços significatius en la recerca i desenvolupament dins l'àmbit de la enginyeria del programari. Fruit d'aquest esforç, l'àrea de la definició de processos i tècniques associades, com per exemple el modelatge de negoci, ha rebut molta atenció tant per part de la industria com del món acadèmic. Així doncs, no es d'estranyar que en els darrers anys hagi sorgit un nou paradigma de desenvolupament de software, anomenat metodologies de desenvolupament dirigides per models, i que, a diferència de les metodologies clàssiques, es catalogat com a centrat en el model. La principal aportació d'aquest nou estil de desenvolupament és la utilització d'eines per transformar automàticament models en altres models més específics, en documentació, en codi font i, inclús, en jocs de proves. D'aquesta manera, es pretén augmentar la productivitat en el desenvolupament alhora que s'escurça el temps de lliurament dels productes finals i es redueix el coneixement tecnològic especialitzat. En aquest treball, s'introduiran dos de les metodologies de desenvolupament dirigides per models més significatives: *Model Driven Architecture (MDA)* i *Domain Specific Modeling (DSM)*. Així mateix, es presentarà un estudi comparatiu d'algunes de les diferents eines existents actualment al mercat que els hi donen suport.

Paraules clau: DSL, DSM, MDA, MDD, Modelatge, OMG, UML

Area TFC: Generació automàtica de programari

Índex de continguts

Índex de figures

1 Introducció

1.1 Justificació del TFC: punt de partida i aportació

Tradicionalment, les metodologies de desenvolupament de software estan basades en una sèrie de passos més o menys universals, com poden ser la recollida de requeriments, l'anàlisi, el disseny, la implementació, les proves i el manteniment. En aquest enfocament, els desenvolupadors recullen uns requeriments que són traslladats en alguns casos a un llenguatge de modelatge determinat per després implementar-los manualment en algun llenguatge de programació concret. La utilització d'aquest enfocament comporta que quasi tot l'esforç del projecte estigui dedicat a la implementació, seguit del disseny i de la recollida de requeriments. Per això, aquest tipus de metodologies són catalogades com centrades en el codi.

En els darrers anys ha sorgit un nou estil de desenvolupament de software a on l'artefacte principal és el model, entès com la especificació d'un sistema des d'una perspectiva particular. Aquest nou estil, anomenat metodologies de desenvolupament dirigides per models, promet proporcionar un procés molt més industrialitzat i que, a diferència de les metodologies clàssiques, es catalogat com a centrat en el model.

Si bé l'ús de models dins del procés de disseny de software és una practica ja coneguda i que s'ha demostrat molt eficient, amb l'ús d'aquestes metodologies s'espera que el major percentatge d'errors aparegui a les fases d'anàlisi i disseny, en comptes de que apareguin en les fases de disseny i construcció dels models de desenvolupament clàssic. Aquesta diferència té un impacte clar en costos, ja que l'esforç de reparació d'un defecte creix exponencialment conforme avancen les fases d'un projecte. Per tant, la detecció primerenca dels defectes és crucial per abaratir el cost de subsanació.

Encara així, la principal aportació d'aquestes metodologies és la utilització d'eines per transformar automàticament aquestos models en altres models més específics, en documentació, en codi font i, inclús, en jocs de proves. D'aquesta manera, es pretén augmentar la productivitat en el desenvolupament alhora que s'escurça el temps de lliurament dels productes finals i es redueix el coneixement tecnològic especialitzat.

Però una estratègia d'aquest tipus requereix d'una inversió important en eines de desenvolupament, ja siguin editors, modeladors, validadors o generadors de codi. En aquest sentit, els principals actors de la industria de desenvolupament de software fa temps que estant intentant proveir d'eines avançades que permetin aplicar completament aquesta nova metodologia. Moltes organitzacions també estan pensant en començar a adoptar aquest nou enfocament mitjançant la utilització d'aquestes eines, però no disposen de la suficient informació sobre les diferents metodologies existents ni de les eines disponibles al mercat actualment.

En aquest treball, per tant, s'introduiran algunes de les metodologies de desenvolupament dirigides per models i es presentarà un estudi comparatiu d'algunes de les diferents eines que els hi donen suport.

1.2 Objectius del TFC

L'objectiu d'aquest TFC és el de determinar fins a quin punt les metodologies de desenvolupament dirigides per models, i en concret, les eines que permeten aplicar aquestes metodologies, estan lo suficientment avançades com per ser capaces de generar codi automàticament a partir de models.

A partir d'aquest objectiu general, es detalla a continuació els objectius específics que es volen assolir:

- **Descriure** en detall les característiques principals de **les metodologies de desenvolupament dirigides per models** per tal de realitzar un estudi comparatiu. En concret, s'analitzaran i compararan les metodologies de *Model Driven Architecture* i *Domain Specific Modeling*.

- **Realitzar un inventari d'eines** existents actualment que permeten aplicar aquestes dues metodologies.

- **Definir uns criteris d'avaluació** per les eines seleccionades que ens permetin:
    - Avaluar les eines independentment de la metodologia emprada.
    - Determinar la capacitat de generació de codi automàticament a partir de models.

- **Analitzar i comparar les eines més representatives** d'acord amb els criteris d'avaluació definits anteriorment.

- **Generar unes conclusions** a partir dels anàlisis anteriors.

1.3 Enfocament i mètode seguit

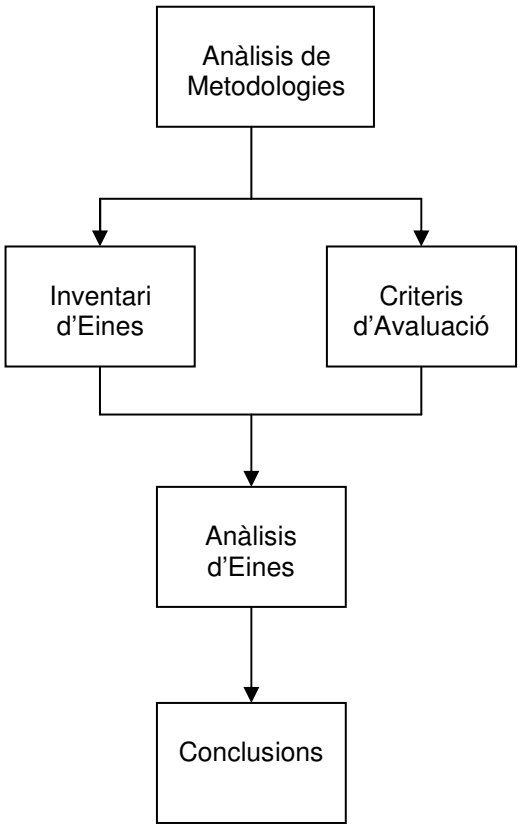
El primer pas d'aquest treball ha consistit en recollir informació sobre les metodologies de desenvolupament dirigides per models per tal de realitzar un anàlisi en detall de les seves característiques principals. A partir d'aquest anàlisi s'ha estat en condicions de realitzar un estudi comparatiu per tal de determinar quin és l'objectiu que pretenen i quin és l'enfocament que apliquen cadascuna d'elles.

En segon lloc, s'han investigat quines eines existeixen actualment que permeten aplicar aquestes metodologies i s'ha realitzat un inventari el més exhaustiu possible.

Acte seguit, s'han definit els criteris que es faran servir per avaluar i comparar les eines, tenint en compte que han de ser independents de la metodologia emprada i que han de determinar les capacitats de generació de codi automàticament a partir de models. Per tal d'agilitzar l'anàlisi i realitzar un comparació objectiva s'ha creat també un exemple que ha estat emprat en l'anàlisi de totes les eines seleccionades.

A continuació, s'han seleccionat les eines més representatives i s'han analitzat i comparat d'acord amb els criteris d'avaluació definits anteriorment.

Finalment, s'han elaborat unes conclusions d'acord amb la informació analitzada durant tot el projecte.



1.4 Planificació del projecte

Tot seguit es presenta un quadre resum de totes les tasques del projecte detallant la seva planificació i l'esforç:

Tasca	Descripció Tasca	Durada (dies)	Data Inici	Data Fi
1	Fase 1: Pla de Projecte	12	01/03/2007	12/03/2007
1.1	Definició dels objectius del TFC	2	01/03/2007	02/03/2007
1.2	Elaboració del pla de projecte	9	03/03/2007	11/03/2007
1.3	Lliurament de la PAC1	1	12/03/2007	12/03/2007
2	Fase 2: Anàlisi de Metodologies de Desenvolupament Dirigides per Models	19	13/03/2007	31/03/2007
2.1	Anàlisi de *Model Driven Architecture* (MDA)	7	13/03/2007	19/03/2007
2.2	Anàlisi de *Domain Specific Modeling* (DSM)	7	20/03/2007	26/03/2007
2.3	Comparativa de les metodologies analitzades	5	27/03/2007	31/03/2007
3	Fase 3: Anàlisi d'Eines de Modelatge	51	01/04/2007	21/05/2007
3.1	Inventari d'eines de modelatge	6	01/04/2007	06/04/2007
3.2	Elaboració d'uns criteris d'avaluació	9	07/04/2007	15/04/2007
3.3	Lliurament PAC2	1	16/04/2007	16/04/2007
3.4	Avaluació de les eines de modelatge	28	17/04/2007	14/05/2007
3.5	Comparativa de les eines analitzades	6	15/05/2007	20/05/2007
3.6	Lliurament PAC3	1	21/05/2007	21/05/2007
4	Fase 4: Memòria i Presentació Virtual	28	22/05/2007	18/06/2007
4.1	Elaboració dels resultats i conclusions del projecte	6	22/05/2007	27/05/2007
4.2	Elaboració de la memòria del TFC	14	28/05/2007	10/06/2007
4.3	Elaboració de la presentació virtual del TFC	7	11/06/2007	17/06/2007
4.4	Lliurament Memòria i Presentació Virtual del TFC	1	18/06/2007	18/06/2007

La planificació del projecte representada en un diagrama de Gantt es pot trobar a l'apartat "Annex I - Planificació del projecte" d'aquest mateix document.

1.5 Productes obtinguts

Els productes obtinguts durant el desenvolupament d'aquest treball són els següents:

- Un pla de projecte;
- Un resum de les principals característiques de les metodologies de desenvolupament dirigides per models juntament amb un quadre comparatiu de les mateixes;
- Un inventari d'eines de modelatge que permeten aplicar aquestes metodologies;
- Uns criteris d'avaluació que serviran per analitzar i comparar les eines de modelatge;
- Un quadre comparatiu de les diferents eines de modelatge analitzades;
- Una conclusió del estudi, que es correspondrà amb la memòria del TFC;
- Una presentació de síntesi de l'estudi.

1.6 Breu descripció dels altres capítols de la memòria

En el capítol 2, “Anàlisi de metodologies de desenvolupament dirigides per models”, es descriuen i comparen les característiques principals de les dos metodologies més representatives: *Model Driven Architecture* i *Domain Specific Modeling*.

El capítol 3, “Anàlisi d’eines de desenvolupament dirigides per models”, està dedicat a les eines que donen suport a aquestes metodologies, realitzant un inventari, definint uns criteris d’avaluació i comparant algunes d’elles mitjançant l’aplicació d’un exemple comú.

I finalment, en el capítol 4, “Conclusions”, s’extreuen unes conclusions a partir de les dades analitzades en els capítols anteriors.

2 Anàlisi de metodologies de desenvolupament dirigides per models

2.1 Model Driven Architecture (MDA)

2.1.1 Introducció

Model Driven Architecture (MDA) és una iniciativa liderada pel *Object Management Group* (OMG) encaminada a la definició d'uns estàndards que permetin la utilització de models per dirigir tot el cicle complet de desenvolupament de software, és a dir, que puguin ser emprats en l'anàlisi i disseny, la programació, les proves, l'ensamblat així com en la instal·lació i el manteniment dels diferents components que conformen un sistema de software.

La finalitat de MDA es facilitar la creació d'uns models que puguin ser llegits per màquines amb un objectiu de flexibilitat a llarg termini en termes de:

- Obsolescència tecnològica: per tal de que les implementacions en noves infraestructures puguin ser suportades i integrades més fàcilment a partir dels dissenys ja existents;
- Portabilitat: de manera que les funcionalitats ja existents puguin ser migrades més ràpidament a nous entorns i plataformes;
- Productivitat: automatitzant les tasques més tedioses del desenvolupament i així permetent que els desenvolupadors es focalitzin en la lògica principal del sistema;
- Qualitat: contribuint a la millora de qualitat de tot el sistema mitjançant la consistència i formalitat dels models;
- Integració: facilitant la producció de sistemes ben integrats a través de les seves especificacions;
- Manteniment: la disponibilitat dels dissenys en un format que pugui ser llegit per una màquina permet que els desenvolupadors accedeixen directament a les especificacions dels sistemes, simplificant les tasques de manteniment;
- Proves i simulació: per tal de que els models puguin ser validats directament contra els requeriments així com ser provats i simulats en diverses infraestructures;
- Retorn de la inversió: de manera que el negoci sigui capaç d'extreure major valor de les inversions en eines.

Per tal d'aconseguir aquestos objectius, MDA parteix d'una idea ben coneguda que consisteix en la separació arquitectural dels conceptes d'un sistema, es a dir, en la separació de la especificació d'una operació d'un sistema dels detalls en que aquesta operació es implementada en el sistema d'acord amb les capacitats de la plataforma on s'executarà. D'aquesta manera es pretén aconseguir tres fites principals: portabilitat, interoperatibilitat i reusabilitat del models.

En aquest sentit, MDA determina que:

- La construcció dels sistemes es realitzi mitjançant models, organitzats a través d'un marc arquitectural de capes i transformacions;
- Els models han de disposar d'una notació i una semàntica ben definida, com a pedra angular per tal d'entendre de forma universal els sistemes;
- Es disposi de metamodels formals que facilitin la integració i la transformació dels models i que siguin la base per la automatització a través d'eines;
- Els estàndards siguin oberts per tal de que siguin acceptats pels diferents fabricants de software, ja que l'intercanvi d'informació entre les eines es considera vital.

Per tal de cobrir aquestos requeriments, MDA es basa en un conjunt d'estàndards definits pel mateix OMG:

- *Unified Modeling Language* (UML): per descriure el domini del problema i la solució arquitectònica;
- *Meta Object Facility* (MOF): per descriure i manipular els models i les metadades;
- *Common Warehouse Metamodel* (CWM): per poder intercanviar i emmagatzemar dades en bases de dades estàndards;
- *Queries/Views/Transformations* (QVT): per realitzar consultes i transformacions en els models;
- *Object Constraint Language* (OCL): per descriure les regles, generalment restriccions, que s'apliquen en models UML.

Tal i com s'il·lustra en el següent diagrama, OMG preveu que MDA es sustenti sobre tot un seguit de serveis omnipresents, que es troben generalment en les aplicacions distribuïdes modernes:

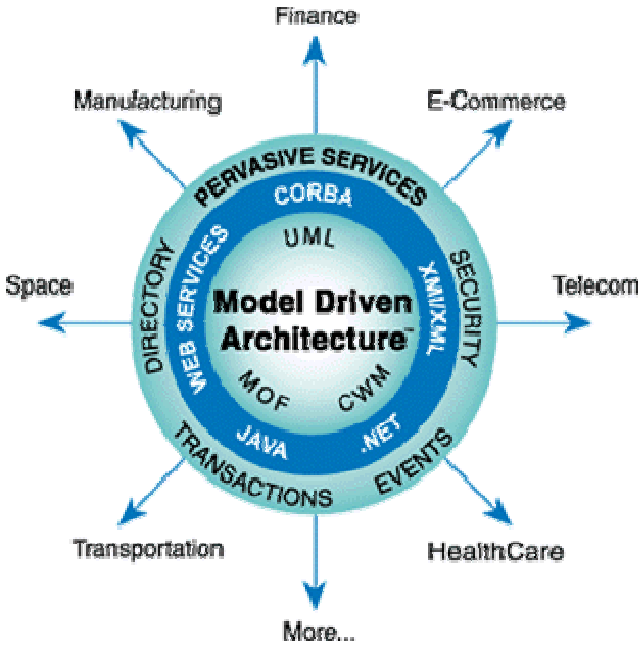


Figura 1 - OMG Model Driven Architecture (MDA)

2.1.2 Els conceptes bàsics

MDA defineix una sèrie de conceptes bàsics que es detallen a continuació:

- *Sistema*: en el context de MDA es refereix a un sistema de software, ja sigui nou o preexistent. Aquest sistema pot incloure: un programa, un ordinador, una combinació de parts de diferents sistemes, una federació de sistemes, persones, una organització, etc.

- *Model*: és la descripció o especificació formal de la funció, estructura i comportament d'un sistema amb un determinat propòsit. Una especificació es considera formal quan està basada en un llenguatge que té un significat semàntic ben definit. Un model es presentat generalment com una combinació de dibuixos i text. El text pot pertànyer a un llenguatge de modelatge o a un llenguatge natural.

- *Arquitectura*: és una especificació de les parts i connectors del sistema i les regles d'interacció de les parts mitjançant els connectors. En el context de MDA, aquestes parts, connectors i regles s'expressen mitjançant una sèrie de models interrelacionats.

- *Viewpoint*: és una tècnica d'abstracció que es focalitza en una determinada sèrie d'aspectes d'un sistema alhora que es suprimeixen els detalls irrellevants. En aquest context, el concepte abstracció es utilitzat per definir el procés d'eliminació de certs detalls per tal d'establir un model més simplificat. Un *viewpoint* pot ser representat mitjançant un o més models.

- *Plataforma*: és un conjunt de sistemes i tecnologies que proporcionen una sèrie de funcionalitats coherents mitjançant interfases i patrons d'ús. Exemples de plataformes son els sistemes operatius, llenguatges de programació, bases de dades, etc.

- *Independència de plataforma*: és una qualitat que un model ha de demostrar quan s'expressa independentment de les característiques de qualsevol plataforma. Independència és un indicador relatiu en termes de mesurar el grau d'abstracció que separa una plataforma d'un altra.

- *Model de plataforma*: descriu una sèrie de conceptes tècnics que representen els elements constituents i els serveis que proporciona una plataforma. També especifica les restriccions en l'ús d'aquestos elements i serveis per part d'altres parts del sistema.

- *Transformació del model*: és el procés de conversió d'un model a un altre dins del mateix sistema. La transformació combina el model independent de la plataforma amb informació addicional per produir un model específic d'una plataforma.

- *Serveis omnipresents*: són serveis que es troben disponibles en un ampli rang de plataformes.

- *Implementació*: és la especificació que proporciona tota la informació requerida per construir un sistema i per posar-lo en operació. Ha de proporcionar tota la informació necessària per construir un objecte i ha de permetre que l'objecte participi en la provisió dels serveis apropiats del sistema.

A partir d'aquestos conceptes, MDA estableix tres models de sistema bàsics que es corresponen cadascun d'ells amb un *viewpoint* diferent. Aquestos models poden ser descrits com capes d'abstracció, ja que en cadascuna de aquestes capes es poden construir tota una sèrie de models, els quals es corresponen a un *viewpoint* del sistema determinat. Aquestos tres models són:

- *Computation Independent Model* (CIM) és la vista del sistema que es centra en l'entorn i els requeriments del sistema, és a dir, sense el detall de la estructura del sistema. CIM també es conegut com model del domini o de negoci, degut a que utilitza un vocabulari que és familiar als experts en la matèria. Presenta exactament el que s'espera que un sistema faci, però amaga tota la informació tecnològica relacionada amb les especificacions per tal de romandre independent de com el sistema és o serà implementat.

  El CIM juga un paper molt important en cobrir el forat que existeix típicament entre els experts en el domini i els responsables d'implementar el sistema. A més, en una especificació MDA, els requeriments del CIM han de poder ser localitzats dins del PIM i del PSM que l'implementen (i vice-versa).

- *Platform Independent Model* (PIM) és la vista del sistema que es focalitza en les operacions pròpies del sistema però amagant els detalls específics de la plataforma d'execució, és a dir, exhibeix el suficient grau d'independència per tal d'habilitar la seva implementació en una o més plataformes.

- *Platform Specific Model* (PSM) és la vista del sistema que detalla l'ús del sistema en una plataforma determinada, és a dir, que combina les especificacions del PIM amb els detalls necessaris per tal d'establir com un sistema usa un tipus de plataforma especifica. Si el PSM no inclou tots els detalls necessaris per tal de produir una implementació d'aquesta plataforma, llavors es considera abstracte (significant que delega en altres models implícits o explícits que contenen els detalls necessaris).

2.1.3 El procés de desenvolupament

El procés de desenvolupament establert per MDA consisteix en la construcció d'un conjunt de models abstractes que són refinats incrementalment mitjançant transformacions entre aquestos models, començant per un model descriptiu del domini del problema i finalitzant amb la generació d'uns components de software adreçats a un plataforma especifica.

Així doncs, el primer pas d'aquest procés consistirà en modelar els requeriments del sistema en un CIM, descrivint la situació exacta en la qual el sistema serà usat així com el que s'espera que faci. Aquestos requeriments seran útils tant per entendre el domini del problema com per establir un vocabulari comú que serà usat en altres models. Els requeriments, a més, han de poder ser localitzats dins del PIM i del PSM que l'implementen (i vice-versa). El CIM pot incloure més d'un model, per exemple, per proporcionar més detall o per focalitzar-ne un aspecte en concret del sistema.

El següent pas serà la construcció del PIM, que descriurà el sistema però no la plataforma d'execució. El model inicial del PIM ha d'heretat els conceptes i requeriments del CIM aplicant les transformacions corresponents. A partir d'aquest model inicial es faran els refinaments necessaris fins aconseguir el resultat desitjat. En aquest refinament es pot aplicar algun tipus d'estil arquitectural, ja sigui de forma manual o a través d'esquemes UML emmagatzemats en algun magatzem de dades que compleixi l'estàndard CWM.

Acte seguit, cal construir el PSM mitjançant la transformació del PIM i posteriorment aplicant els refinaments que calgui. El PSM pot incloure més o menys detall, depenent del seu propòsit, però ha d'especificar com el sistema farà ús de la plataforma escollida. Aquest PSM podrà ser implementat directament si conté tota la informació necessària per construir el sistema i per posar-lo en producció o pot actuar com un PIM el qual caldrà refinar en altres models per tal de poder ser implementat. Igual que en el cas del PIM, en aquest refinament es pot aplicar algun tipus d'adaptacions especifiques de la plataforma, com per exemple, els *UML Profiles*.

El pas final en el procés de transformació serà la generació del codi d'implementació així com altres tipus de components de suport, com per exemple, els arxius de configuració, els arxius descriptors dels components, els guions d'instal·lació, etc. Per a poder generar el màxim nombre de components, és necessari que s'inclogui al PSM la semàntica complerta i el comportament de la plataforma d'execució. I aquí es a on entren en joc la maduresa i qualitat de les eines MDA emprades.

Com a resum gràfic del que s'ha explicat anteriorment, a la següent figura es representa el procés complert de desenvolupament de software mitjançant MDA:

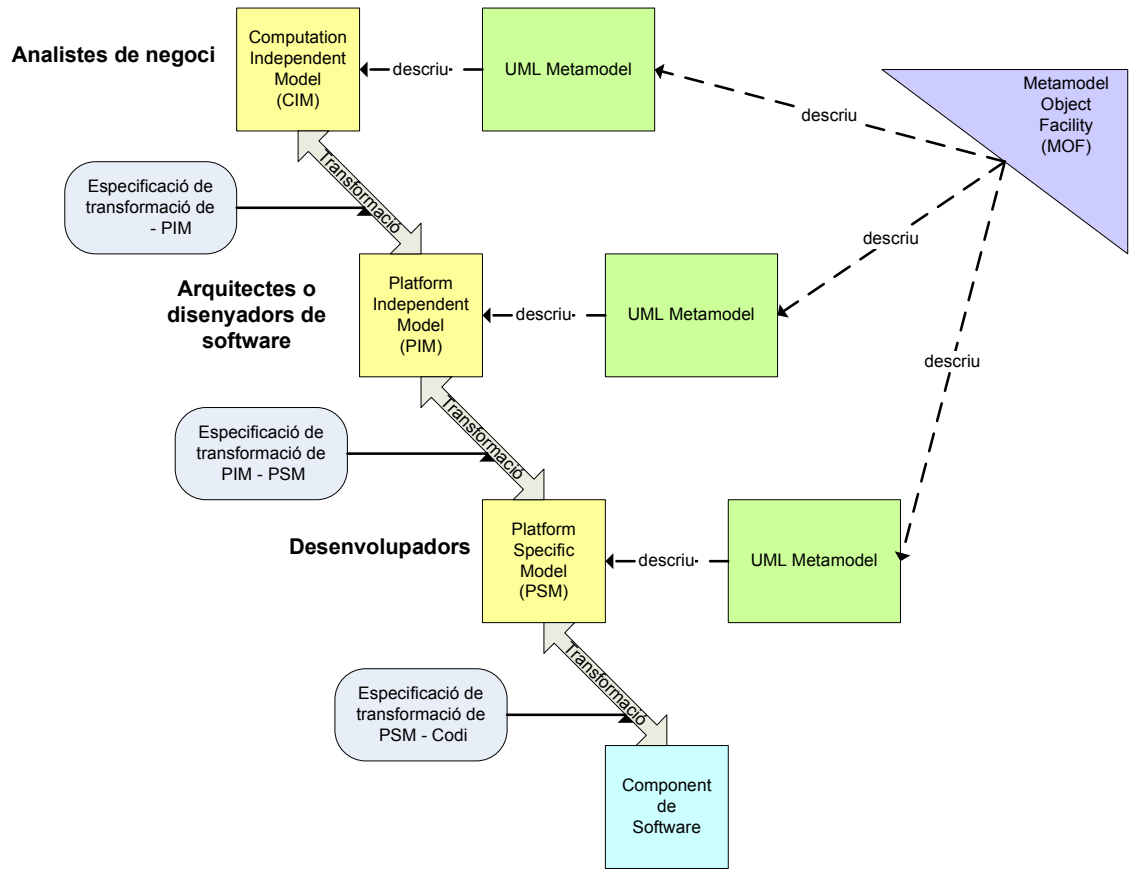


Figura 2 - El procés de desenvolupament mitjançant MDA

En tot aquest procés de desenvolupament mitjançant MDA existeixen dos elements clau: els models i les transformacions. Aquestos elements formaran part d'un patró general que serà emprat repetidament durant tot el procés de desenvolupament:

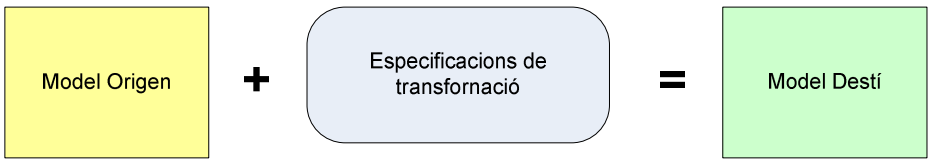


Figura 3 - Patró general de desenvolupament en MDA

Aquest patró determina un escenari a on un model origen és transformat en un model destí a partir d'unes especificacions de transformació i d'acord amb la naturalesa del model destí. Encara que aquesta transformació pot ser resultat d'un procés manual, MDA aconsella que es faci mitjançant un procés automàtic. Però per a que això sigui possible, és necessari que les especificacions detallin les correspondències entre els elements que conformen el model origen i els elements que pertanyen el model destí.

Aquestes especificacions es poden generar d'una manera simple a partir de la tipologia d'un model, com per exemple, les classes, les associacions, els rols o els estereotips, o, per contra, poden ser molt complicades de generar en cas d'emprar llenguatges de modelatge diferents. En aquest últim cas, es farà necessari l'ús de metamodels, és a dir, la definició de la sintaxi abstracte de la notació de modelatge. Es en aquesta definició a on juga un paper clau

l'estàndard MOF, que ens permetrà definir els llenguatges de modelatge i gestionar les metadades associades.

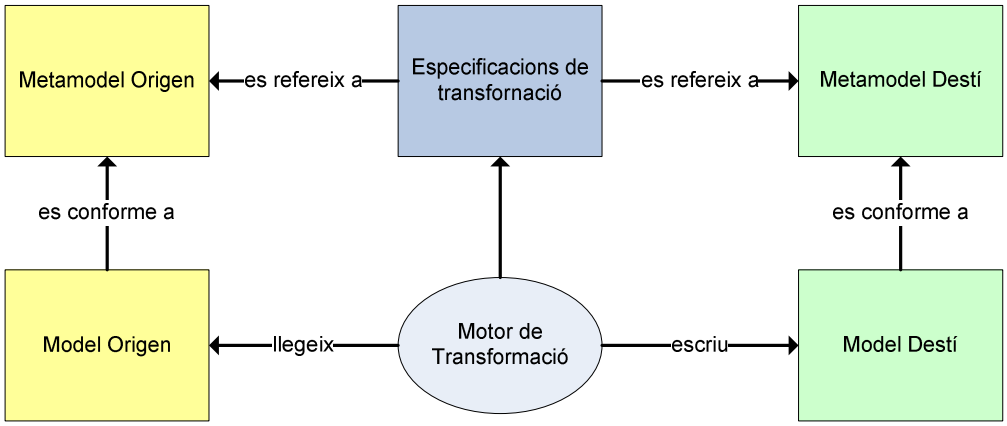


Figura 4 - Transformacions en MDA

De igual manera, el detall de les correspondències entre els elements pot ser molt complicat, ja que generalment només es podrà emprar la tipologia de l'element, i aquesta dada en molts casos serà insuficient per especificar una transformació complerta. Es per això que farà falta unes regles addicionals per tal de produir el model destí correcte. En aquest sentit, MDA proporciona uns elements anomenats marques, que representen un concepte del PSM per tal d'indicar com un element ha de ser transformat a partir d'un element del PIM.

Les marques són considerades part del PSM, ja que són especifiques d'una plataforma en concret, encara que son aplicades dins del PIM. Així doncs, el PIM més totes les marques constituiran l'entrada per la transformació del model cap al PSM.

2.2 Domain Specific Modeling (DSM)

2.2.1 Introducció

Domain-Specific Modeling (DSM) és una metodologia de desenvolupament de software que està basada en la utilització d'uns models que contenen unes especificacions representades en forma de conceptes de domini. La principal característica d'aquesta metodologia és que aquestos conceptes pertanyen a un sol domini que, generalment, és propi d'una sola organització o d'una arquitectura determinada. Es a dir, que en comptes d'emprar un llenguatge de modelatge generalista, cada organització o arquitectura ha de definir una notació pròpia per tal de modelar el sistema. Per aconseguir aquesta fita, aquesta metodologia s'ajuda dels anomenats *Domain-Specific Languages* (DSL) que, com es veurà més endavant, no són més que una notació per representar conceptes d'un domini de problema concret.

En aquesta metodologia, per tant, s'augmenta el nivell d'abstracció més enllà de la simple programació. En aquest sentit, els elements del model representen

components del món del domini, no del món del codi, és a dir, de la implementació. Així doncs, s'utilitza un llenguatge de modelatge propi de cada organització que segueix la semàntica i les abstraccions d'un domini concret, permetent que els modeladors percebin per si mateixos que estan treballant directament amb uns conceptes que coneixen perfectament. Aquesta especificació d'alt nivell permetrà, a més, incloure les restriccions pertinents dins del llenguatge de modelatge i generar els productes finals automàticament a partir dels models.

A diferència d'altres metodologies de desenvolupament dirigides per models, a DSM no hi ha cap organització que intenti liderar els seus principis ni, per tant, cap estàndard que concreti la tecnologia que s'ha d'emprar. Per contra, aquest terme va néixer dins del marc de OOPSLA (*Object-Oriented Programming, Systems, Languages and Applications*), una conferència anual, principalment de caràcter acadèmic, que cobreix temes relacionats amb l'enginyeria de software i, en concret, amb el món de la orientació a objectes. Encara així, la companyia Microsoft es la principal impulsora de DSM, al basar la seva iniciativa de desenvolupament de software anomenada "*Software Factories*" en la utilització, entre altres, d'aquesta metodologia.

Els beneficis que ofereix DSM són:

- Millor comprensió: es redueix la necessitat de aprendre noves semàntiques ja que els conceptes del domini són coneguts i, per tant, son naturals pels desenvolupadors, tasca que ajuda a la lectura, comprensió, validació i comunicació dels mateixos.

- Major agilitat: els canvis de requeriments són conseqüència, generalment, d'un canvi en el domini del problema, no d'un canvi en el domini de la implementació. Per aquesta raó, aplicar aquestos canvis en el llenguatge de modelatge resulta més fàcil que fer-los en la programació, ja que estan orientats al domini del problema.

- Major productivitat: el problema es soluciona només una vegada, en un nivell alt d'abstracció, ja que el codi final es generat directament des d'aquesta solució.

- Reduir complexitat: mantenir les especificacions en un nivell d'abstracció significativament més alt que el tradicional codi font requereix de menys especificacions. A més, com que el llenguatge de modelatge pertany a un únic domini concret, no cal que inclogui les tècniques i estructures que afegeixen complexitat i treball innecessari pels desenvolupadors.

- Major qualitat: la inclusió de les regles de domini, es a dir, les restriccions d'ús dels conceptes fa més difícil, per no dir impossible, que es produeixin especificacions no permeses. La consistència del producte també es veu millorada gràcies a que no hi ha canvis entre la fase de disseny i la d'implementació.

Però per contra, aquesta metodologia té un cost alt d'implementació inicial. Cal definir abans de res un llenguatge específic de modelatge així com els generadors corresponents, i això es una tasca complicada. De totes maneres, això no vol dir que es requereixi d'un inversió més alta que la que s'aplica en un procés de desenvolupament tradicional. Si es té en compte l'esforç total que s'inverteix en el desenvolupament d'un projecte tradicional, es pot comprovar que DSM estalvia molts recursos de desenvolupament, al automatitzar gran part de les tasques que es fan manualment.

2.2.2 Els *Domain-Specific Languages (DSL)*

Un *Domain-Specific Language* (DSL) és un llenguatge de programació que ofereix, mitjançant les apropiades notacions i abstraccions, una potencia expressiva focalitzada, i usualment restrictiva, en un problema de domini particular. Dit d'un altre manera, són llenguatges que en comptes d'estar focalitzats en una problema de tecnologia concret, com pot ser la programació, l'intercanvi de dades o la configuració, estan dissenyats per representar més directament el domini del problema que es vol atacar, es a dir, llenguatges amb un propòsit especial.

Aquestos llenguatges són coneguts també com llenguatges de modelatge, entenent-se aquest concepte com la definició de símbols i relacions que s'utilitzen per construir un model. Generalment, encara que no es obligatori, un llenguatge de modelatge es diagramàtic, i representa un model concret mitjançant grafs compostos de nodes connectats via línies.

Però mentre que els llenguatges de modelatge tradicionals pretenen ser el més genèrics possible, com és el cas de UML, un llenguatge de modelatge DSL pretén ser el més específic possible. Això incrementa el nivell d'abstracció dels models, redueix la quantitat d'informació necessària per ser modelada, redueix les conversions al moure el llenguatge de domini més a prop del domini tal i com el perceben els dissenyadors i millora la qualitat i l'abast del generadors de codi. D'aquesta manera quan més específic sigui el llenguatge, més alts seran els beneficis de productivitat d'aplicar DSM.

La idea dels DSL, però, no es nova. Actualment es poden trobar molts exemples de llenguatges específics, com el *VHDL* per descriure components electrònics de hardware, *lex* and *yacc* per l'anàlisi lèxic de programes, *HTML* per representar la composició de pàgines web, o *SQL* per consultar i actualitzar bases de dades. Aquestos exemples contrasten amb altres llenguatges més generalistes, com per exemple, el *XML* o el *Java*, que no tenen un propòsit específic, sinó que permeten configurar i desenvolupar qualsevol tipus d'aplicació.

Per tal de ser més concrets, s'il·lustrarà l'ús dels DSL agafant com exemple una funcionalitat concreta d'un rellotge. Imaginem que volem desenvolupar una aplicació que permeti posar en hora un rellotge, però amb l'objectiu de que els desenvolupadors d'aquesta funcionalitat es focalitzin no en la implementació, sinó en la essència de la aplicació, es a dir, en el comportament.

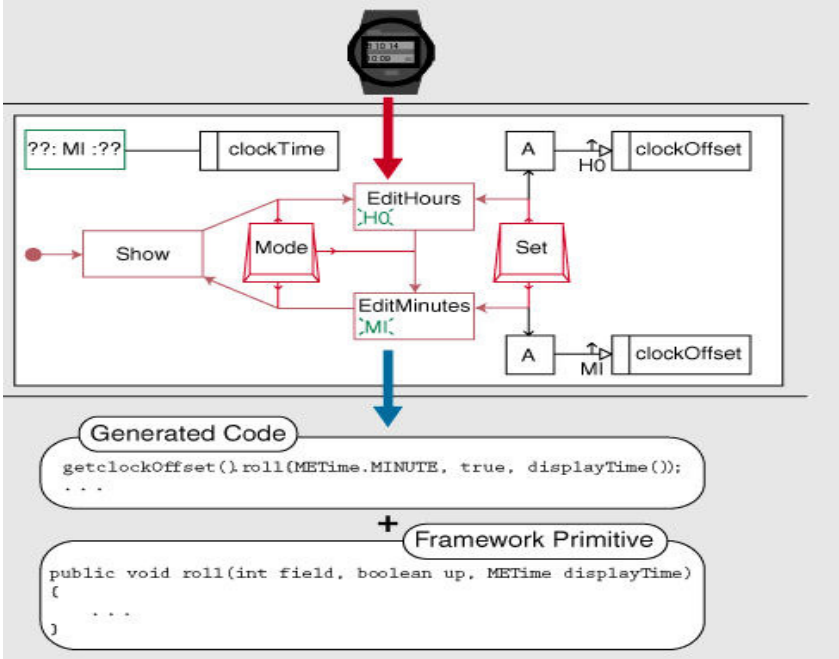


Figura 5 - Exemple de DSL

El model de la figura anterior representa el comportament de la funcionalitat mitjançant conceptes de domini expressats amb un llenguatges específic de domini: accions d'usuari que s'activen polsant botons, elements que parpellegen mentre es van editant, o operacions finals que incrementen i decrement l'hora. Com es pot veure, la implementació dels detalls (en aquest cas, en llenguatge Java), son ocultats als desenvolupadors, però encara així, com que el model captura tots els requeriments de l'aplicació i es disposa d'una arquitectura coneguda, es possible generar a partir d'aquest model el 100% del codi de la funcionalitat.

2.2.3 Com implementar DSM

Per obtenir tots els beneficis de DSM en termes de productivitat, qualitat i reducció de complexitat són necessaris dos components: un llenguatge específic de domini i un generador de codi. Opcionalment, pot ser convenient crear un tercer component, la llibreria de components del domini o *framework*. Antigament, calia, a més, implementar tot un conjunt d'eines per tal de donar-hi suport, i aquesta tasca, apart de ser molt complicada, sol recaure fora de la competència de moltes organitzacions. Avui en dia, però, ja existeixen al mercat eines de modelatge i generadors de codi que d'una manera oberta habiliten la consecució d'aquesta tasca, permetent disposar d'un entorn de desenvolupament basat en DSM en molts pocs dies.

Així doncs, el primer pas serà la elecció d'alguna de les eines de modelatge existents al mercat. Una vegada es disposi d'aquesta eina, caldrà definir un DSL i això comporta definir tres aspectes principals: els conceptes de domini, la

notació a emprar per representar-los en models textuals o gràfics, i les regles que guien el procés de modelatge.

La clau per definir els conceptes de domini serà trobar els experts en el domini del problema, generalment, desenvolupadors experimentats que ja han implementat algun producte en aquest domini o que han dissenyat l'arquitectura del sistema. Aquestos experts podran identificar fàcilment els conceptes a partir de la terminologia, els descriptors i els components dels sistemes ja existents. Sobre aquestos conceptes caldrà proporcionar una notació, es a dir, s'han de definir els símbols que es convertiran en la representació textual o gràfica dels conceptes en el model.

De totes maneres, amb els conceptes de domini i la notació, per si sols, no generarem el llenguatge de modelatge complert. Es necessari entendre, a més, com poden ser posats en comú, es a dir, com interaccionen els components entre sí mateixos. Per tant, caldrà enriquir aquestos conceptes bàsics amb les regles del domini, que, generalment, restringiran l'ús del llenguatge definint quins tipus de connexions entre els conceptes estan permeses. Poden, a més, especificar cóm certs components poden ser reusats i com els models han de ser organitzats. Aquestes regles han de ser identificades i implementades des del principi, ja que la generació de codi i la implementació final ha de ser perfectament vàlida.

Finalment, per tal de cobrir l'espai entre el món del model i el món del codi caldrà definir el generador, que especificarà cóm la informació es extreta dels models i cóm serà transformada en codi. En el cas de generació de codi font, ha de poder ser compilat directament sense que requereixi de cap modificació ni esforç manual per part dels desenvolupadors, de manera que aquest sigui només un simple producte intermediari entre el model i l'executable final.

Per tal de crear aquest generador, es necessari disposar primer d'un codi complert i provat en la plataforma destí. Aquest codi haurà de ser analitzat per determinar quines parts poden ser derivades dels elements del model i quin tipus de patró ha de ser aplicat en les derivacions. Algunes vegades, aquesta tasca requerirà d'una refinació del codi amb l'objectiu de simplificar o clarificar aquestos patrons. En aquest cas, cal disposar d'un joc de proves complert per tal de comprovar si amb la refinació es segueixen generant els components correctament.

En alguns casos, però, pot ser de molta utilitat afegir algun tipus de *framework* que ens proporcioni la interfase entre el codi que es generarà i la plataforma d'execució. Generalment aquest *framework* ja ens el proporcionarà la mateixa plataforma, però, sovint, i per tal de que la generació de codi sigui més simple, serà convenient definir algun component o utilitat extra. Algun exemple de *framework* pot ser la utilització de *Aspect Oriented Programming* (AOP).

A partir d'aquest punt, els desenvolupadors disposaran d'una eina de modelatge específic que contindrà una notació i una semàntica que farà referència a uns conceptes de domini ben coneguts. Amb aquesta eina, estaran

en disposició d'utilitzar els components construïts anteriorment de forma industrialitzada:

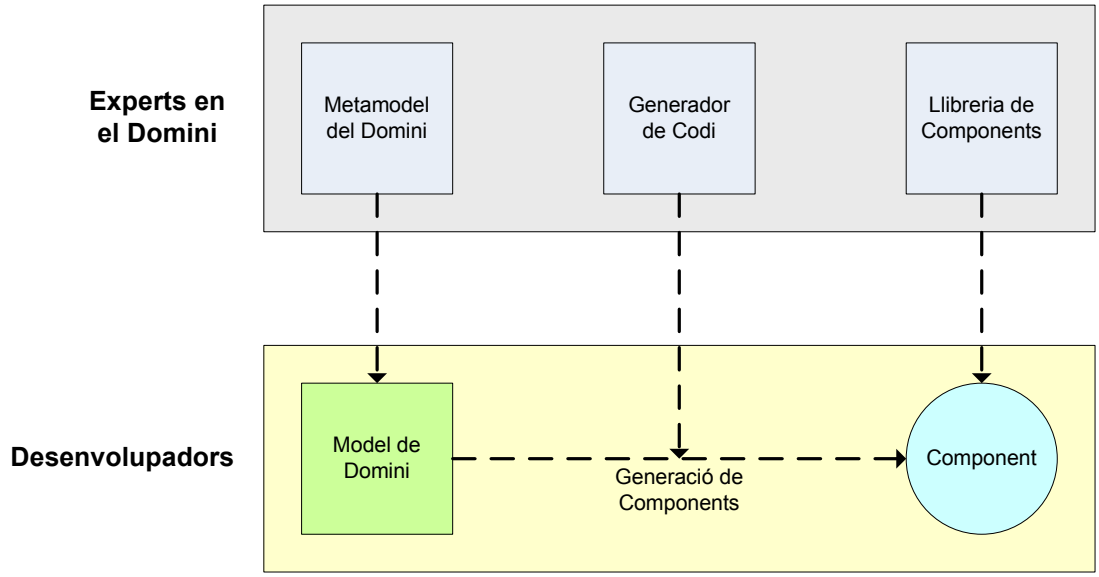


Figura 6 - El procés de desenvolupament mitjançant DSM

2.3 Resum comparatiu de les metodologies analitzades

Les dos metodologies analitzades anteriorment comparteixen l'objectiu comú de guiar el desenvolupament de software mitjançant models, amb els quals es pretén dissenyar un sistema completament i ser capaços de generar el codi automàticament. L'enfocament que fan servir, però, es sensiblement diferent.

Aquest diferencia d'enfocament respon a dos escoles de pensament que es van posar de manifest durant una sessió de debat anomenada "*Translation: Myth or Reality*" celebrada durant el transcurs de OOPSLA'96:

- Un enfocament, anomenat "translacional", liderat per S. Shlaer i S. J. Mellor: Un model per l'aplicació i un altre per l'arquitectura que es composen per generar el codi.
- Un altre enfocament liderat per G. Booch, J. Rumbaugh, I. Jacobson i R. C. Martin: Diferents nivells d'abstracció fins generar el codi.

En efecte, en el cas de MDA, és transformen models amb un alt nivell d'abstracció (PIM) en altres models amb menor nivell d'abstracció (PSM), que a la vegada s'acaben transformant en codi. Es a dir, a cada pas del cicle de desenvolupament s'edita el model per tal de refinar-lo i dotar-lo de més detall per finalment generar el codi a partir del model final. L'objectiu que es pretén aconseguir amb aquest procés és habilitar l'ús del mateix PIM en diferents plataformes de software i estandarditzar totes les transformacions i formats de models mitjançant UML per tal de que aquestos siguin portables entre eines de diferents empreses.

Per contra, a DSM es requereix de l'ús d'un DSL creat específicament dins l'àmbit d'un domini d'un problema d'una aplicació. La combinació d'aquest DSL

amb les capacitats del *framework* i de la plataforma, es a dir, de la capa arquitectural, formaran l'executable final mitjançant l'aplicació d'un generador de codi. En aquesta metodologia la independència de la plataforma no es una necessitat urgent, encara que pot ser fàcilment aconseguida mitjançant la utilització de diferents generadors de codi. En el seu lloc, el principal objectiu es millorar significativament la productivitat del desenvolupador al treballar en un domini que li és familiar.

Aquestos enfocaments defineixen clarament la diferencia entre MDA i DSM i responen clarament a la qüestió de quin procés s'ha d'emprar en cada cas.

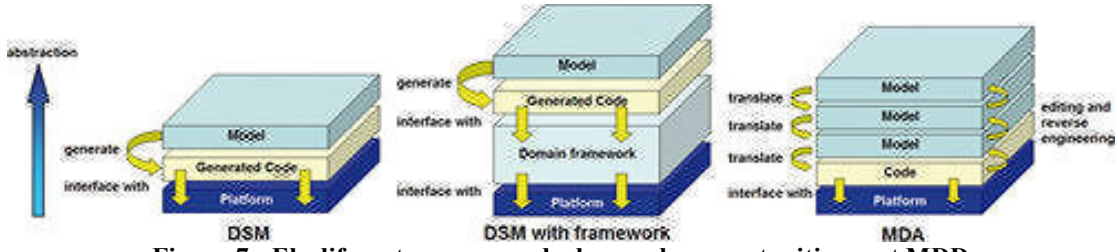


Figura 7 - Els diferents processos de desenvolupament mitjançant MDD

Amb MDA, no hi ha un focus especial en utilitzar un DSL, sinó que s'estableix UML com a llenguatge universal de modelatge, degut a que es el estàndard de OMG. Aquesta situació comporta que hi hagi un monopoli de conceptes, tècniques i mètodes al voltant de UML i OMG. Tampoc busca encapsular el coneixement de domini que pot existir en una companyia, sinó que assumeix que aquest coneixement no existeix o simplement es irrellevant. Per contra, requereix d'un profund coneixement de UML, i, encara que aquest és el llenguatge de modelatge orientat a objectes predominant actualment, en alguns casos pot ser extern a la companyia i, per tant, cal que sigui assimilat pels desenvolupadors. A més, UML no cobreix totes les necessitats d'especificació d'un projecte de software, per exemple, no defineix els documents textuals, els esquemes físics d'una base de dades o les interfícies d'usuari. Sembla llavors que MDA pot ser apropiat per projectes d'integració d'aplicacions o sistemes.

Amb DSM, s'assumeix que el coneixement de domini necessari ja esta disponible a les organitzacions. Això comporta que es requereixi d'una competència en un domini concret, una capacitat que una companyia només pot assolir quan treballa contínuament amb el mateix domini del problema. De qualsevol forma, la implementació d'un DSL es una tasca difícil que en alguns casos pot fer dubtar del guany i del retorn de la inversió que s'obtindrà. En efecte, encara que la primera generació d'eines comercials ja estan disponibles en el mercat, la desavantatge principal d'aquesta aproximació és el temps necessari que cal invertir per tal de dissenyar, implementar i mantenir el DSL. Encara així, l'avantatge principal es que els desenvolupadors poder aprendre, aplicar i estendre el DSL ràpidament sense que hi hagi una corba d'aprenentatge elevada. Juntament amb la capacitat d'abstraure al desenvolupador de les construccions i detalls que no estan relacionades amb el domini del problema fan que DSM sigui útil en el cas de projectes a on el domini del problema es manté estable i, per tant, la solució arquitectònica es comuna.

Totes aquestes característiques es poden resumir en la següent taula comparativa:

Característica	Metodologia	
	MDA	**DSM**
Promotor	OMG	No hi ha promotor, encara que Microsoft és la principal impulsora
Focus principal	Portabilitat, interoperatibilitat i reusabilitat dels models	Millorar la productivitat dels desenvolupadors mitjançant el treball en un domini de problema conegut
Enfocament	Diferents nivells d'abstracció fins generar el codi	Un model per l'aplicació i un altre per l'arquitectura que es composen per generar el codi
Basat en estàndards	Si (UML, MOF, CWM, QVT)	No
Llenguatge de modelatge	UML	DSL
Independència de plataforma	Si, mitjançant un model independent de plataforma (PIM)	Parcial, encara que el modelatge es específic d'una plataforma, es poden construir generadors per diverses plataformes
Adreçat a un domini de problema particular	No, enfocament generalista	Si
Eines disponibles	Si (disponibilitat alta)	Si (disponibilitat baixa)
Cost d'implantació	Mig, cal dominar UML i construir les transformacions	Alt, cal definir el llenguatge de modelatge i els generadors
Tipus de projectes idonis	Integració d'aplicacions o sistemes	Domini del problema estable amb solució arquitectònica ben definida

3 Anàlisi d'eines de desenvolupament dirigides per models

3.1 Inventari d'eines de modelatge

A continuació es relacionen algunes de les eines de modelatge i generació automàtica de codi que existeixen actualment al mercat.

Metodologia	Producte	Fabricant
MDA	Acceleo	Obeo
MDA	Ameos	Aonix
MDA	AndroMDA	Open Source
MDA	ArcStyler	Interactive Objects
MDA	AtoM^3	Open Source
MDA	Codagen Architect	Manyeta
MDA	CodeGenie	Domain Solutions
MDA	Constructor MDARAD	i3 Design
MDA	iQgen	innoQ
MDA	MCC	InferData
MDA	MDE	Open Source
MDA	MDWorkbench	Sodius
MDA	Mia Studio	Mia-Software
MDA	ModFact	Open Source
MDA	Objecteering/UML	Objecteering Software
MDA	OlivaNOVA	CARE Technologies
MDA	OpenMDX	Open Source
MDA	OptimalJ	Compuware
MDA	Rational Software Architect	IBM
MDA	Select Solution for MDA	Select Business Solutions
MDA	Together	Borland
MDA	UCL MDA Tools	Open Source
MDA	XMFMosaic	Xactium
DSM	DSL Tools	Microsoft
DSM	Eclipse EMF	Open Source
DSM	MetaEdit+	MetaCase
DSM	Mia DSL	Mia-Software
DSM	Sculptor	Open Source
MDA/DSM	openArchitectureWare	Open Source

3.2 Selecció d'eines de modelatge per l'anàlisi

Algunes de les eines anteriorment relacionades no compleixen estrictament tots els principis de les metodologies MDA i DSM. En alguns casos concrets, estan focalitzades a resoldre una àrea especifica, com pot ser només el modelatge o només la generació de codi per una arquitectura determinada. En aquest sentit, podem catalogar les eines en la següent tipologia:

- Solució complerta;
- Eines de modelatge;
- Eines de transformació;
- Eines de generació de codi;
- Eines de suport (magatzems de models, etc.).

Els fabricants de solucions complertes diuen que tenir un sol paquet és més sensat perquè així els desenvolupadors no han d'anar alternant entre diferents eines, es a dir, tenen un entorn únic de desenvolupament. En canvi, els fabricants de solucions parcials, diuen que comprar un paquet complert és molt car i que es pot habilitar un entorn de desenvolupament dirigit per models complert per menys diners emprant diferents eines.

En aquest estudi, però, seleccionarem aquelles eines que son complertes funcionalment, es a dir, que cobreixen tant el modelatge d'un sistema com la generació de codi, ja que d'aquesta manera podrem realitzar un anàlisi més complert. Ens interessa, a més, seleccionar una eina de cadascuna de les dos metodologies analitzades anteriorment.

Després de realitzar una recerca en Internet per determinar quines són les eines més representatives, s'han seleccionat per ser analitzades les següents:

- *Microsoft Domain-Specific Language Tools*: que segueix la metodologia *Domain-Specific Modeling*.
- *Eclipse EMF + GMF + oAW*: que utilitza un híbrid entre la metodologia *Domain-Specific Modeling* i *Model-Driven Architecture*.
- *Borland Together 2006*: que segueix la metodologia *Model-Driven Architecture*.

3.3 Criteris d'avaluació

Com que les metodologies analitzades tenen característiques diferents, ens caldrà buscar uns criteris que puguin ser avaluats independentment de la metodologia emprada. Els aspectes que més interessen avaluar són els llenguatges de modelatges suportats, la possibilitat de transformació dels models en altres models, les capacitats de generació de codi automàticament i la integració amb altres eines.

Així doncs, els criteris que es faran servir per avaluar les eines seleccionades seran els següent:

- El tipus de plataforma en la qual es pot executar l'eina.
- Quins són els llenguatges de modelatge suportats.
- Si existeix la capacitat de transformar models en altres models i, en cas afirmatiu, en quin llenguatge es realitza la transformació.
- En el cas de modelar amb UML, si existeix suport per aplicar perfils UML.
- Si existeix la capacitat d'aplicar restriccions en els models i, en cas afirmatiu, en quin llenguatge es defineixen les restriccions.

- El tipus de format dels generadors de codi, per exemple, si estan basats en plantilles.
- Si incorpora generadors de codi ja predefinits.
- Si existeix la possibilitat de modificar els generadors de codi i, en cas afirmatiu, en quin llenguatge es poden definir els generadors.
- Si està habilitada la opció d'importar i exportar models i, en cas afirmatiu, en quin format es realitza.
- Si existeix integració amb altres eines de captura de requisits i de control de versions.

3.4 Exemple de problema: ISA

Per tal de poder analitzar les eines en profunditat i d'un manera homogènia, definirem un exemple comú basat en un problema que està acotat en un domini d'una organització en concret. Aquesta organització i el domini del problema són reals, encara que a efectes d'aquest treball els simplificarem, degut a la seva complexitat juntament amb el poc temps disponible per realitzar l'anàlisi.

El domini del problema pertany a una entitat financera amb seu a Catalunya que executa els seus processos de negoci en un entorn transaccional IBM IMS i utilitza PL/I com a llenguatge de programació. La arquitectura d'execució que sustenta aquest domini, anomenada *Integració de Serveis d'Aplicació* (ISA), s'acaba de crear per tal de proporcionar un marc de treball comú per a totes les aplicacions que es desenvolupen en aquesta entitat. Els objectius principals que es pretenen assolir amb aquesta nova arquitectura són:

- Enfocament cap a una *Service Oriented Architecture* (SOA), de manera que es treballi amb serveis desacoblats i sense capa de presentació.
- Definició d'un flux pels diferents processos de negoci d'una manera explicita i documentada.
- Possibilitar la multicanalitat dels diferents processos de negoci.

En aquest moment s'està buscant una eina que permeti als desenvolupadors augmentar la seva productivitat en el desenvolupament d'aplicacions en aquest nou entorn d'execució. L'enfocament que es pretén seguir és el de desenvolupament dirigit per models, es a dir, que els desenvolupadors modelin els fluxos de les seves aplicacions i que es generi codi automàticament.

Els conceptes generals en que estaran basades totes les aplicacions que s'executin en aquesta arquitectura seran els següents:

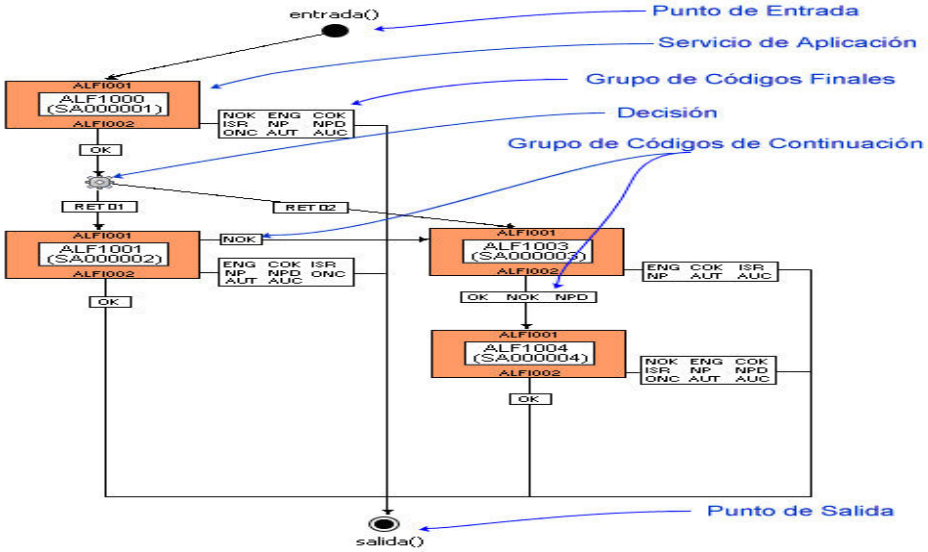


Figura 8 - ISA: model d'aplicacions

En aquest diagrama es poden veure els diferents elements que conformen el que s'anomena Servei de Negoci (un procés de negoci d'una aplicació):

- Un punt d'entrada: aquest node és l'inici del procés i rep una missatgeria d'entrada, que en aquest entorn es correspon a una *include* en llenguatge PL/I.
- Uns serveis d'aplicació: que són unes peces de codi que contenen una lògica de negoci concreta (com per exemple, la consulta de les dades d'una persona, un ingrés en compte, un càrrec de comissions, etc.). Aquestos serveis reben una missatgeria d'entrada i produeixen una missatgeria de sortida, que en tots dos casos es corresponen amb una *include* en llenguatge PL/I. El codi que resol aquesta funcionalitat de negoci està externalitat del flux i, per tant, no es modela.
- Uns mòduls de decisió: que permeten bifurcar el flux del procés en funció de l'anàlisi d'unes dades d'entrada, que en aquest entorn es correspon a una *include* en llenguatge PL/I. El codi que resol aquesta funcionalitat està externalitat del flux i, per tant, no es modela.
- Un punt de sortida: aquest node és el final de la transacció i ha d'exposar una missatgeria de sortida, que en aquest entorn es correspon a una *include* en llenguatge PL/I.
- Codis de sortida: cada servei d'aplicació i mòdul de decisió retorna un o varis codis de sortida, que són els que determinaran la bifurcació del flux. Per exemple, un servei d'aplicació pot retornar un codi "*OK*" que bifurcarà a un altre servei d'aplicació, i pot retorna també un codi "*NOK*", que bifurcarà al final del procés.

A partir dels models creats pels desenvolupadors d'acord amb aquestos conceptes, s'ha de generar codi automàticament en llenguatge PL/I amb el següent format:

```
<Nom_Servei_Negoci>: PROC OPTIONS(MAIN);

/*****************************/
/*- Declaració de Components -*/
/*****************************/
```

```
/*- Include d'Entrada -*/
IE(<Include_Entrada_Servei_Negoci>);

/*- Servei d'Aplicació <Nom_Servei_Aplicacio_1>:
<Descripció_Servei_Aplicacio_1>  -*/
SN(<Nom_Servei_Aplicacio_1>)
INCLUDEIN(<Include_Entrada_Servei_Aplicacio_1>)
INCLUDEOUT(<Include_Sortida_Servei_Aplicacio_1>);
...
/*- Servei d'Aplicació <Nom_Servei_Aplicacio_n>:
<Descripció_Servei_Aplicacio_n> -*/
SN(<Nom_Servei_Aplicacio_n>)
INCLUDEIN(<Include_Entrada_Servei_Aplicacio_n>)
INCLUDEOUT(<Include_Sortida_Servei_Aplicacio_n>);

/*- Mòdul de Decisió <Nom_Modul_Decisio_1>:
<Descripció_Modul_Decisio_1>  -*/
MD(<Nom_Modul_Decisio_1>)
INCLUDEIN(<Include_Entrada_Modul_Decisio_1>);
...
/*- Mòdul de Decisió <Nom_Modul_Decisio_n>:
<Descripció_Modul_Decisio_n>  -*/
MD(<Nom_Modul_Decisio_n>)
INCLUDEIN(<Include_Entrada_Modul_Decisio_n>);

/*- Include de Sortida -*/
IS(<Include_Sortida_Servei_Negoci>);

/************************/
/*- Declaració del Flux -*/
/************************/
FLUXEINICI TO(<Nom_Servei_Aplicacio_x>);
FLUXE FROM(<Nom_Servei_Aplicacio_x>) TO(<Nom_Servei_Aplicacio_y>)
CODI(<Codi_Sortida>);
...
FLUXE FROM(<Nom_Servei_Aplicacio_x>) TO(<Nom_Modul_Decisio_y>)
CODI(<Codi_Sortida>);
...
FLUXE FROM(<Nom_Modul_Decisio_x>) TO(<Nom_Servei_Aplicacio_y>)
CODI(<Codi_Sortida>);
...
FLUXEFINAL FROM(<Nom_Servei_Aplicacio_x>) CODI(<Codi_Sortida>);
...
FLUXEFINAL FROM(<Nom_Modul_Decisio_x>) CODI(<Codi_Sortida>);
...

END <Nom_Servei_Negoci>;
```

Els codis identificats amb els símbols “< ... >” amb lletra cursiva es corresponen amb les diferents entitats, atributs i relacions del model.

Una vegada introduïdes les característiques d’aquest entorn d’execució, ja podem definir l’exemple comú que es farà servir per tal d’analitzar les diferents eines:

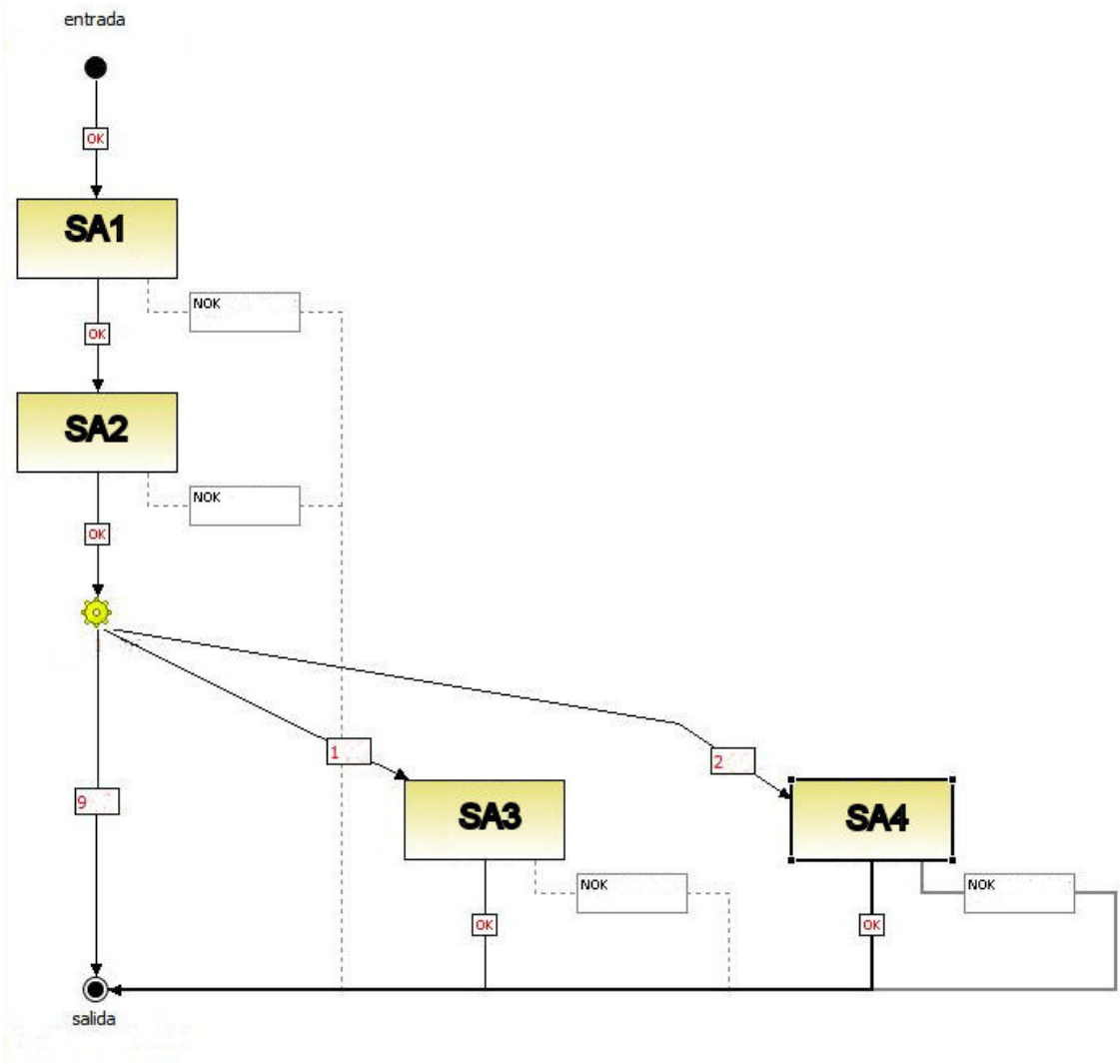


Figura 9- ISA: exemple de flux

En aquest flux es pot veure com a partir de l'entrada del servei de negoci s'invoca al servei d'aplicació "*SA1*", que bifurcarà cap al servei d'aplicació "*SA2*" en cas de retornar un codi "*OK*". El servei d'aplicació "*SA2*", en cas de retornar un codi "*OK*", bifurcarà a un mòdul de decisió (icona d'una roda dentada), que a la seva vegada, en funció del codi de retorn bifurcarà al servei d'aplicació "*SA3*" (en cas de que el codi sigui "*1*"), al servei d'aplicació "*SA4*" (en cas de que el codi sigui "*2*"), o a la sortida del servei de negoci (en cas de que el codi sigui "*9*"). Tots els serveis d'aplicació bifurcaran a la sortida del servei de negoci en cas de retornar un codi "*NOK*".

Una de les primeres dificultats a la qual ens enfrontem és la de trobar una eina que permeti modelar i generar codi per aquest tipus de solucions, ja que el llenguatge del codi a generar, PL/I, es un llenguatge poc conegut i molt orientat a entorns *IBM Mainframe*, entorn en el que no existeixen moltes eines de desenvolupament. De igual manera, al no ser un llenguatge orientat a objectes, es fa difícil encaixar aquest tipus de processos en els llenguatges de modelatge tradicionals.

Per tant, amb les característiques peculiars d'aquest exemple, es pretén analitzar algunes les eines seleccionades anteriorment per comprovar si són capaces de modelar un flux ISA i de generar automàticament tot el codi PL/I necessari.

3.5 Microsoft Domain-Specific Language Tools

La proposta que Microsoft ofereix dins de l'àmbit de desenvolupament dirigit per models es una eina anomenada *DSL Tools*, que es troba integrada dins de l'entorn de desenvolupament de *Microsoft Visual Studio 2005*, i que permet la creació de llenguatges específics de domini (DSL).

L'objectiu principal d'aquesta eina es permetre la creació per part dels experts del domini d'un altra eina que pugui ser emprada directament pels usuaris finals i que farà la funció de modelador i generador de codi. Aquesta nova eina incorporarà conceptes, notacions i restriccions que són fàcilment reconeguts per aquestos usuaris, ja que formen part d'un domini d'un problema que coneixen perfectament, al ser propi de la seva organització.

L'eina DSL Tools consisteix, a grans trets, en:

- Un assistent per crear un projecte de definició d'un llenguatge específic de domini.
- Una dissenyador gràfic per tal de definir el model (o conceptes) del domini i la notació a emprar.
- Unes llibreries d'execució per tal d'establir les regles del domini, basades en els llenguatges C# o Visual Basic.
- Un conjunt de generadors que, a partir de la definició del model del domini, la notació, i les regles del domini, obtenen un altra eina amb una interfície d'usuari, que es la que farà servir l'usuari final.
- Un marc d'execució per tal de definir els generadors de codi, els quals agafen les dades modelades per l'usuari i generen una sortida en format text d'acord amb unes plantilles definides amb els llenguatges C# o Visual Basic.

La següent il·lustració proporciona una perspectiva d'alt nivell de com es dissenya, personalitza, prova i desplega un DSL amb aquesta eina:

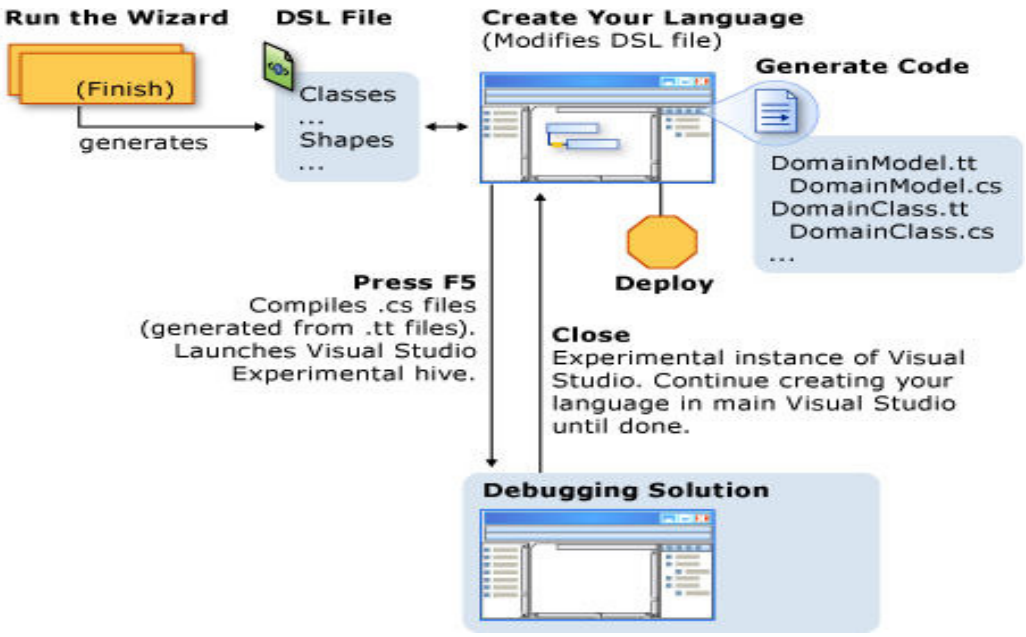


Figura 10 - Microsoft DSL Tools: procés de definició d'un DSL

Així doncs, per tal de definir el DSL del nostre problema es començarà executant l'assistent de creació d'un nou projecte mitjançant les opcions del menú "*File → New → Project*", escollint com a tipus de projecte "*Other Project Types → Extensibility → Domain-Specific Language Designer*" i anomenant "*UOC-Test*" al projecte. A continuació s'escollirà la plantilla "*Minimal Language*" i s'anomenarà "*ISA*" al nou llenguatge. L'assistent demanarà, a més, algunes dades referents al producte final a generar.

A continuació ens apareixerà la pantalla del "*DSL Designer*", que ens permetrà dissenyar el nostre llenguatge específic de modelatge, tal i com es pot veure a la figura següent:

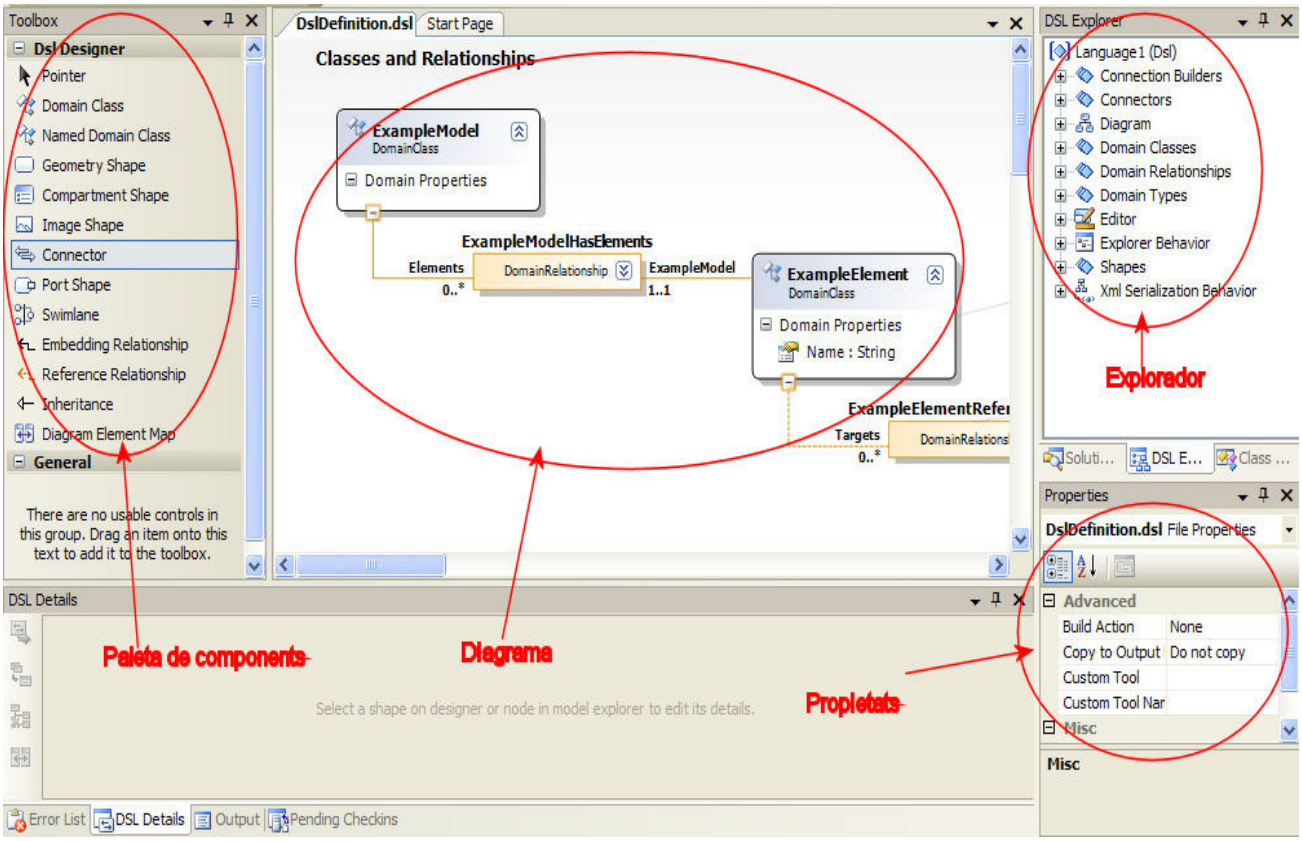


Figura 11- Microsoft DSL Tools: DSL Designer

En aquest dissenyador trobarem quatre elements fonamentals:

- La paleta de components: que ens permetrà arrastrar els següents tipus de components cap al diagrama:
  - Classe de domini: que representen els diferents tipus d'objectes del domini.
  - Relacions de domini: que representen la informació referent a la relació existent entre dos classes de domini. Aquestes relacions poden ser de tipus referència, es a dir, que un objecte fa referència a un altre, o de tipus compost, es a dir, que un objecte conté un altre objecte.
  - Figures: que són la representació gràfica dels diferents artefactes modelats, es a dir, la notació del modelatge.
- El diagrama: a on es representa el model del domini que estem definint.
- L'explorador: que ens mostra en un arbre els diferents elements inclosos al diagrama.
- La vista de propietats: a on es poden consultar i modificar les propietats de l'element seleccionat al diagrama, per exemple, per definir nous atributs a les classes i relacions del domini.

Una vegada introduïts aquestos conceptes, es pot passar a dissenyar el model del domini. En el cas de l'exemple ISA, el model resultant serà el següent:

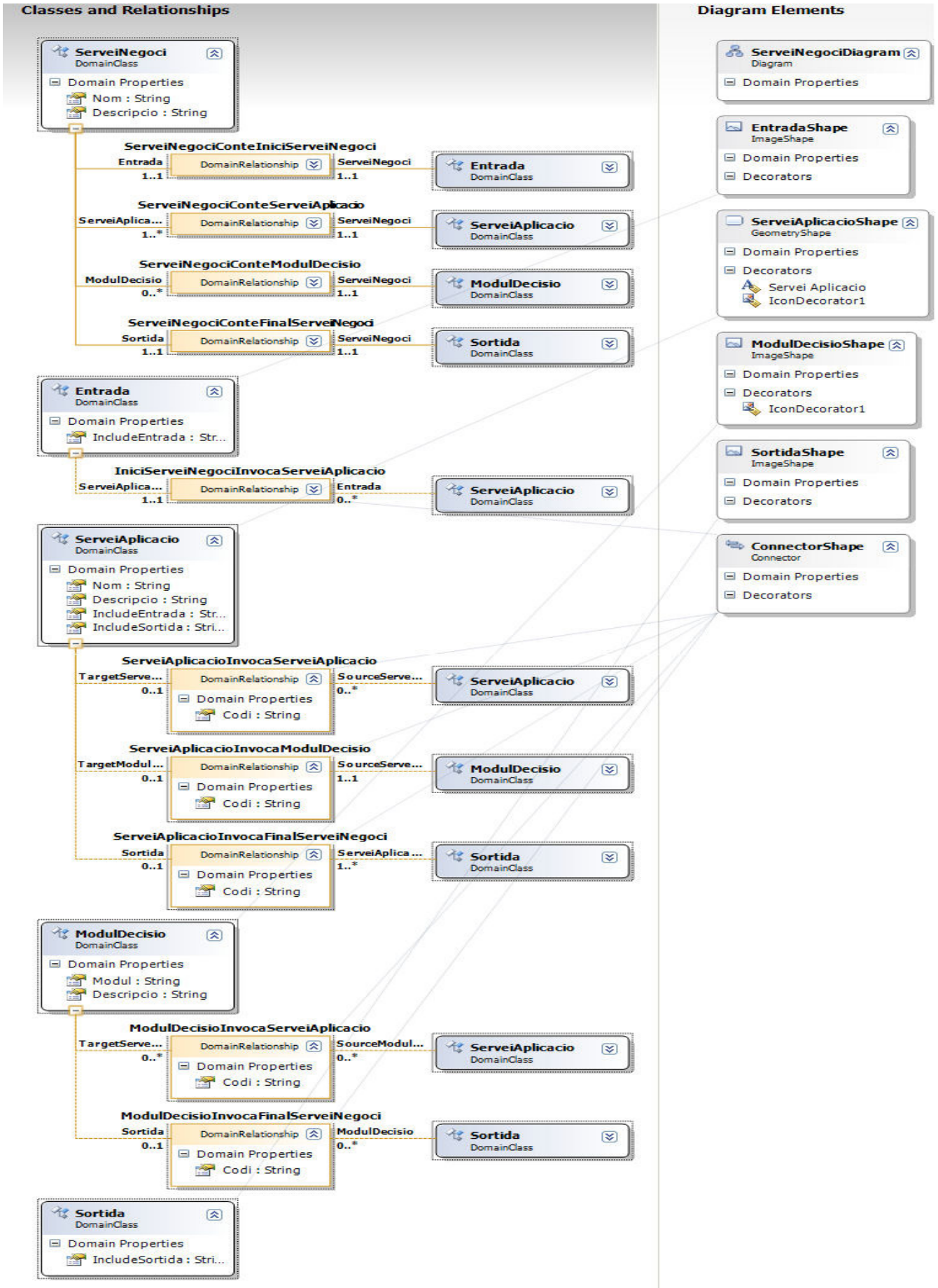


Figura 12 - Microsoft DSL Tools: el model del domini

A la part esquerra del diagrama (*Classes and Relationships*) anterior trobem les diferent classes del domini així com els seus atributs i les relaciones entre les classes; i a la part dreta (*Diagram Elements*) trobem la notació que faran servir els desenvolupadors per tal de modelar els seus sistemes, que poden incorporar alguna figura geomètrica (rectangles, cercle, ...) o, per contra, incorporar alguna imatge predissenyada per nosaltres.

A continuació, definirem les regles del domini, es a dir, les restriccions que cal aplicar a les diferents classes i relacions de domini, mitjançant la codificació manual d'una classe en llenguatge C# o Visual Basic. En aquest cas, s'han codificat restriccions referents als diferents atributs de les classes mitjançant el llenguatge C#. A tall d'exemple, el codi que ve a continuació valida que estiguin informats els atributs de la classe *ServeiAplicació*:

```
using Microsoft.VisualStudio.Modeling;
using Microsoft.VisualStudio.Modeling.Validation;

namespace UOC.UOCTest.ISA {
    [ValidationState(ValidationState.Enabled)]

    partial class ServeiAplicacio {
        [ValidationMethod(ValidationCategories.Save)]
        public void ComprovarAtributs(ValidationContext context) {
            if (this.Nom == null) {
                context.LogError("S'ha d'informar el nom del Servei
d'Aplicació", "ServeiAplicacio",  this);
            }
            if (this.IncludeEntrada.Length == 0) {
                context.LogError("S'ha d'informar la Include d'Entrada
del Servei d'Aplicació", "ServeiAplicacio",  this);
            }
            if (this.IncludeSortida.Length == 0) {
                context.LogError("S'ha d'informar la Include de
Sortida del Servei d'Aplicació", "ServeiAplicacio",  this);
            }
        }
    }
}
```

Com es pot veure en aquest exemple, cadascuna dels conceptes que formen part del domini tenen la seva equivalència en una classe, de manera que accedir als seus atributs i relacions es converteix en una tasca realment senzilla. Destaquem, a més, la possibilitat d'invocar a aquestes restriccions només en determinades circumstàncies, com per exemple, al obrir el diagrama o al emmagatzemar-lo. Aquesta característica s'aconsegueix invocant al mètode:

```
        [ValidationMethod(ValidationCategories.[Open|Save|Menu])]
```

El següent pas consistirà en provar el DSL dissenyat. Primerament executarem un procés que generarà automàticament tot el codi necessari per la interfície d'usuari final. Aquest procés s'invoca mitjançant la utilitat "*Transform All Templates*". A continuació, es compila el codi generat mitjançant el menú "*Build → Build Solution*", i finalment s'executa mitjançant les opcions de menú "*Debug*

→ *Start without debugging*". Ens apareixerà llavors una interfície d'usuari a on el desenvolupador podrà modelar el sistema. En el nostre cas, modelarem el nostre exemple ISA:

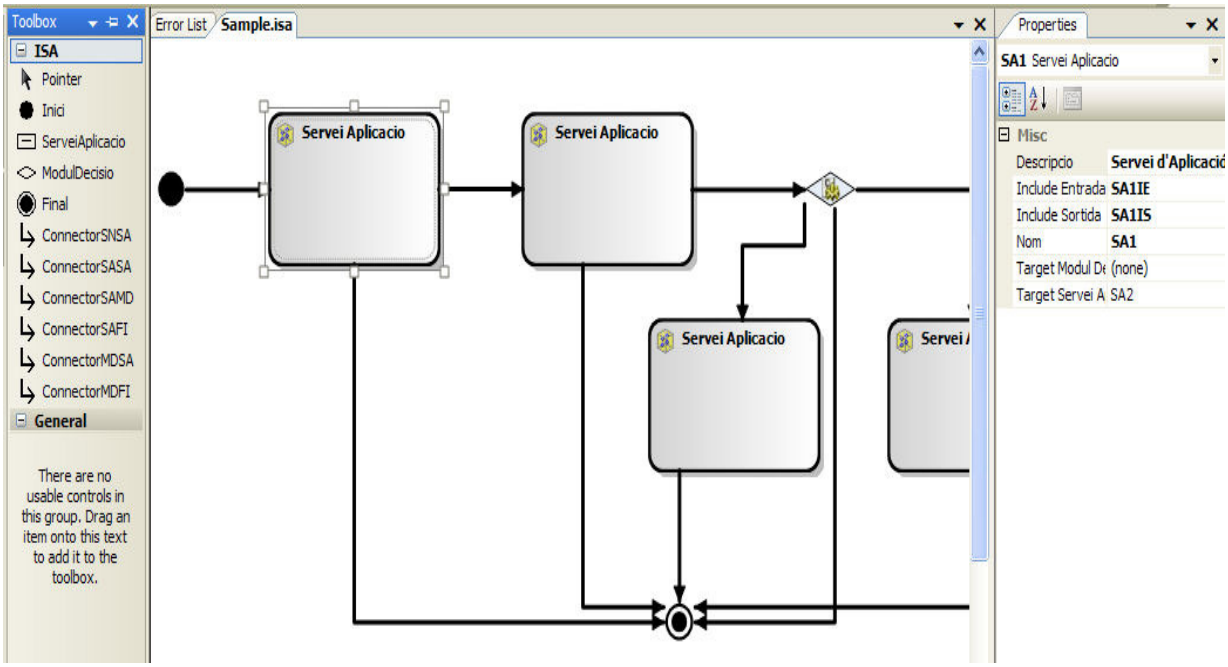


Figura 13 - Microsoft DSL Tools: el nostre editor

La figura anterior és l'eina o interfície que veurà i utilitzarà l'usuari final, es a dir, el desenvolupador. A la part esquerra apareix la paleta de components amb les diferents figures definides anteriorment. A la part central apareix el diagramador per tal de que l'usuari dissenyi la seva solució. I a la part dreta, apareixen les propietats del component que està seleccionat al diagrama. Cal destacar, però, que en aquest exemple no apareixen ni les etiquetes identificatives dels serveis d'aplicació, ni les dels mòduls i connectors. Això es degut a que la versió analitzada de l'eina no permet relacionar atributs del model amb els atributs de les figures mitjançant el dissenyador gràfic, sinó que cal programar-ho manualment retocant el codi generat per l'assistent. Així doncs, si es volen veure els atributs d'un element en concret, caldrà seleccionar-lo al diagrama per tal de que apareguin els atributs a la finestra de la part dreta de l'eina.

Una vegada comprovat que el modelador i les restriccions són formalment valides, només caldrà definir la conversió del model a codi font, es a dir, caldrà definir com es realitzarà la generació automàtica de codi. Per realitzar aquesta tasca farem servir una plantilla que caldrà codificar manualment i que serà processada per un motor de transformació per tal de generar automàticament arxius de text, que poden contenir qualsevol tipus de codi o documentació. La definició de les plantilles es realitzarà mitjançant els llenguatges C# o Visual Basic. Per exemple, per generar el codi amb la declaració de components del nostre exemple, caldrà emprar el següent codi codificat en C#:

```
<#@ template
inherits="Microsoft.VisualStudio.TextTemplating.VSHost.ModelingTextTra
nsformation" debug="true"#>
<#@ output extension=".cs" #>
```

```
<#@ ISA processor="ISADirectiveProcessor"
requires="fileName='Sample.isa'" #>
<# DateTime objToday =  DateTime.Now; #>

/************************************************/
/* Servei de negoci <#= this.ServeiNegoci.Nom #>
/* Generat automàticament amb Microsoft DSL Tools el <#=
objToday.ToString("g") #> */
/************************************************/
<#= this.ServeiNegoci.Nom #>: PROC OPTIONS(MAIN);

/*****************************/
/*- Declaració de Components -*/
/*****************************/
/*- Include d'Entrada -*/
IE(<#= this.ServeiNegoci.Entrada.IncludeEntrada #>);
<# foreach (ServeiAplicacio eachSA in
this.ServeiNegoci.ServeiAplicacio) { #>
/*- Servei d'Aplicació <#= eachSA.Nom #>: <#= eachSA.Descripcio #> -*/
SN(<#= eachSA.Nom #>) INCLUDEIN(<#= eachSA.IncludeEntrada #>)
INCLUDEOUT(<#= eachSA.IncludeSortida #>);
<# } #>
<# foreach (ModulDecisio eachMD in this.ServeiNegoci.ModulDecisio) {
#>
/*- Mòdul de Decisió <#= eachMD.Modul #>: <#= eachMD.Descripcio #> -*/
MD(<#= eachMD.Modul #>);
<# } #>
/*- Include de Sortida -*/
IS(<#= this.ServeiNegoci.Sortida.IncludeSortida #>);

END <#= this.ServeiNegoci.Nom #>;
```

Els codis identificats amb els símbols “<# ... #>” es corresponen a codi en llenguatge C#, i la resta, a codi que s’escriu directament en l’arxiu de sortida. Com es pot comprovar, i al igual que en el cas de les restriccions, en les plantilles es pot accedir als atributs i relacions de cadascun dels components del model mitjançant objectes.

Quan completem la plantilla, podrem executar-la mitjançant el motor de transformació, que ens generarà el codi final:

```
************************************************/
/* Servei de negoci SNTest
/* Generat automàticament amb Microsoft DSL Tools el 09/05/2007 2:09
*/
/************************************************/
SNTest: PROC OPTIONS(MAIN);

/*****************************/
/*- Declaració de Components -*/
/*****************************/
/*- Include d'Entrada -*/
IE(SN-IE);
/*- Servei d'Aplicació SA1: Servei d'Aplicació 1 -*/
SN(SA1) INCLUDEIN(SA1IE) INCLUDEOUT(SA1IS);
/*- Servei d'Aplicació SA2: Servei d'Aplicació 2 -*/
SN(SA2) INCLUDEIN(SA2IE) INCLUDEOUT(SA2IS);
/*- Servei d'Aplicació SA3: Servei d'Aplicació 2 -*/
SN(SA3) INCLUDEIN(SA3IE) INCLUDEOUT(SA3IS);
```

```
/*- Servei d'Aplicació SA4: Servei d'Aplicació 2 -*/
SN(SA4) INCLUDEIN(SA4IE) INCLUDEOUT(SA4IS);
/*- Mòdul de Decisió MD1: Mòdul de Decisió 1 -*/
MD(MD1);
/*- Include de Sortida -*/
IS(SN-IS);

/*************************/
/*- Declaració del Fluxe -*/
/*************************/
FLUXEINICI TO(SA1);

FLUXE FROM(SA1) TO(SA2) CODI(OK);
FLUXE FROM(SA2) TO(MD1) CODI(OK);
FLUXE FROM(MD1) TO(SA3) CODI(1);
FLUXE FROM(MD1) TO(SA4) CODI(2);

FLUXEFINAL FROM(SA1) CODI(NOK);
FLUXEFINAL FROM(SA2) CODI(NOK);
FLUXEFINAL FROM(SA3) CODI(*);
FLUXEFINAL FROM(SA4) CODI(*);
FLUXEFINAL FROM(MD1) CODI(9);

END SNTest;
```

Una vegada validat que tot el procés es correcte, i com a últim pas, caldrà desplegar la solució complerta. Aquest procés crearà un arxiu auto instal·lable, que es el que s'entregarà als usuaris finals. Aquestos, mitjançant aquesta solució, estaran en disposició de modelar el seu projecte d'acord amb una notació fàcil d'entendre i generar automàticament el codi de la solució final.

El codi complert de l'exemple ISA desenvolupat amb l'eina *Microsoft DSL Tools* es pot trobar a l'apartat "Annex II – Exemple amb Microsoft DSL Tools" d'aquest mateix document.

3.6 Eclipse EMF + GMF + oaW

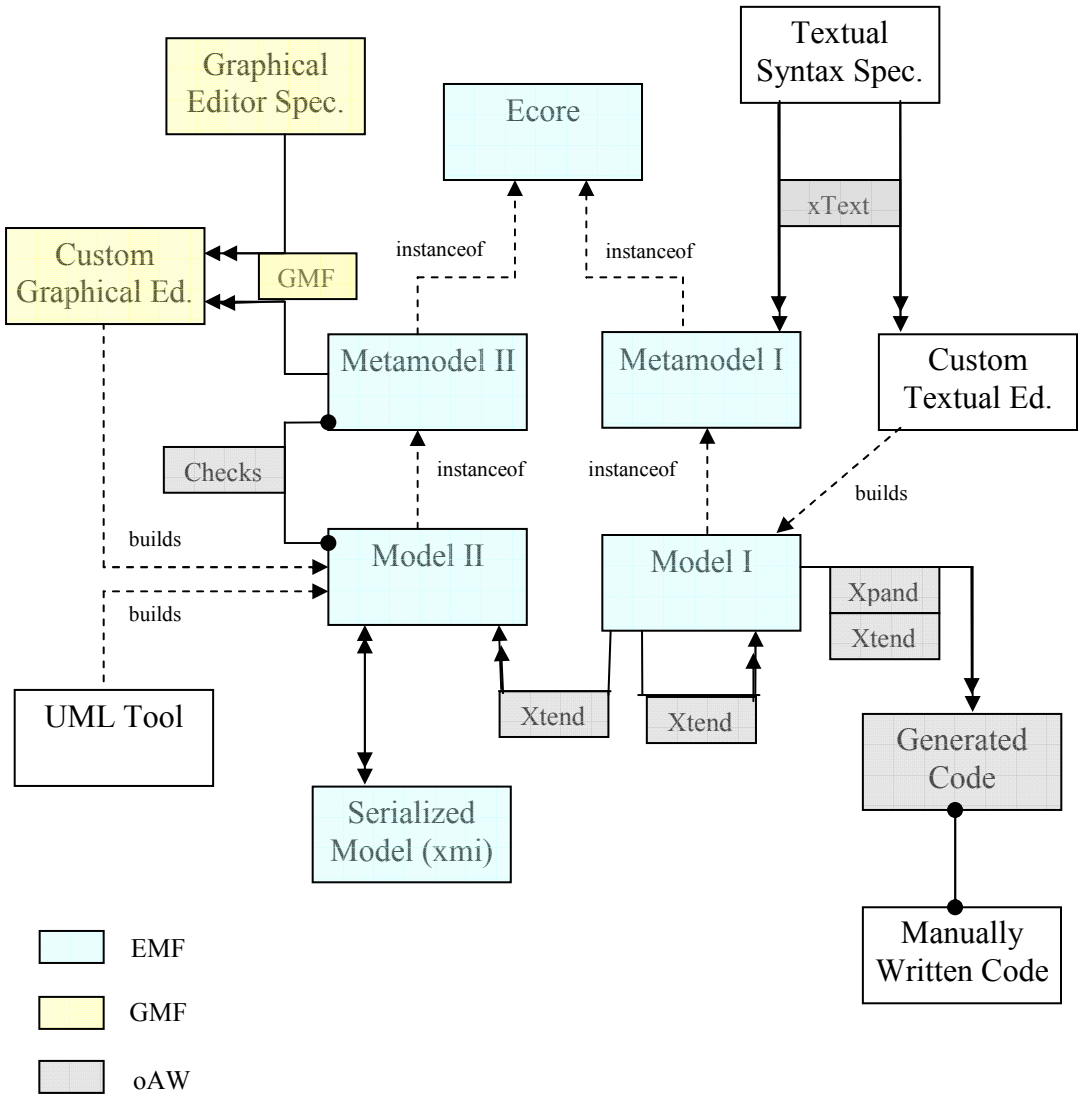
Aquesta proposta *open source* s'emmarca dins del projecte de *Eclipse Modeling*, que té com a objectiu principal proporcionar una sèrie de marcs de treball, eines i implementacions estàndard focalitzades en la evolució i promoció de tecnologies de desenvolupament basades en models. Aquesta combinació d'elements proporciona un marc de treball pel desenvolupament dirigit per models híbrid, ja que utilitza aspectes de MDA i DSM alhora.

En aquest sentit, aquesta solució és una combinació d'una sèrie d'eines basades en Eclipse:

- *Eclipse Modeling Framework* (EMF): un marc de treball que permet definir metamodels i que proveeix d'una sèrie d'adaptadors per tal facilitar a les aplicacions l'accés a les dades d'un model conforme a aquest metamodel.
- *Graphical Modeling Framework* (GMF): un marc de treball que proveeix d'un component de generació i una infraestructura d'execució per tal de desenvolupar editors gràfics de models basats en EMF.

- *openArchitectureWare* (oAW): una col·lecció d'utilitats per assistir en el desenvolupament dirigit per models, es a dir, que a partir de metamodels i models permeti la transformació de models i la generació automàtica de codi.

L'objectiu principal d'aquesta combinació es la de permetre la creació per part dels experts del domini d'un altra eina que pugui ser emprada directament pels usuaris finals per tal de generar codi automàticament a partir de models. Però a diferència de les eines DSM pures, a més de permetre la definició d'un DSL així com l'ús d'una eina gràfica per tal de modelar aquest DSL, en aquesta combinació es permet també importar models definits amb l'estàndard UML. Aquest mode de funcionament es pot veure més clar en el següent diagrama:



Per una banda tenim el component EMF, que ens permetrà definir el metamodel amb els conceptes del domini del problema i que es sustentarà sobre la tecnologia *ecore*, propietària de Eclipse. Sobre aquest metamodel, es podrà construir el model de la solució, ja sigui mitjançant una eina externa de UML, tal com estableix MDA, com mitjançant un DSL, definit de forma textual o

amb la ajuda d'un editor gràfic específic. En aquest últim cas, entra en joc el component GMF, que ens permetrà definir la notació dels diferents elements del metamodel i generar una eina gràfica. En tots els casos, i amb la ajuda del component oAW, es permet definir restriccions al model mitjançant l'ús de OCL, i es podran definir transformacions entre models amb la ajuda de Xtend. Finalment, el component oAW també ens permetrà generar codi automàticament mitjançant l'ús del Xpand i Xtend.

Així doncs, per tal de poder construir el nostre exemple, ens caldrà la següent relació de *plugins*:

- Eclipse 3.3M4
- EMF 2.3.0M5
- EMFT OCL 1.1M4
- EMFT Query 1.1M4
- EMFT Transaction 1.1M4
- EMFT Validation 1.1M4
- GEF 3.3M4
- GMF 2.0M4
- oAW 4.1.2

Una vegada instal·lat Eclipse i tots els *plugins* necessaris, l'executarem. El primer que caldrà fer es crear un projecte GMF mitjançant les opcions de menú "*File → New → Project*" i escollint com a tipus de projecte "*Graphical Modeling Framework → New GMF Project*". Anomenarem al projecte "*edu.uoc.isa*" i activarem la opció de "*Show dashboard view for created project*". Aquesta opció obrirà una vista amb un tauler que ens indicarà l'estat del projecte així com els passos que hem de seguir per completar-ho:

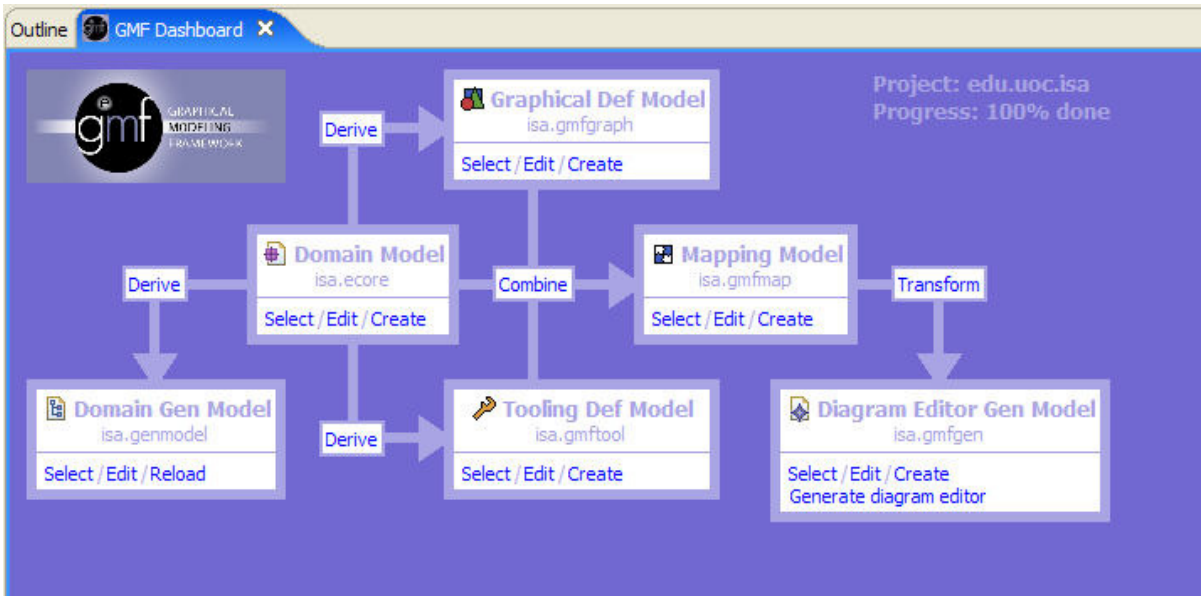


Figura 14 - Eclipse GMF Dashboard

El primer pas consistirà en definir el model del domini, o metamodel. Per crear aquest metamodel només caldrà seleccionar la opció "*Create*" dins la capça anomenada "*Domain Model*" del tauler del projecte. Aquest model del domini el

crearem sota un nou directori anomenat "*model*" dins del projecte abans creat i l'anomenarem "*isa.ecore*".

A continuació apareixerà l'editor de models de EMF, sobre el qual s'hauran de definir els diferents elements del domini. En aquest editor trobarem tres elements fonamentals:

- *EClass*: son les diferents classes o tipus d'objectes del domini.
- *EAttribute*: son els atributs de les classes del domini.
- *EReference*: són les relacions existents entre dos classes de domini. Aquestes relacions poden ser de tipus referència, es a dir, que un objecte fa referència a un altre, o de tipus compost, es a dir, que un objecte conté un altre objecte.

Una vegada assimilats aquestos conceptes, es pot passar a dissenyar el model del domini, que en aquest cas serà el següent:

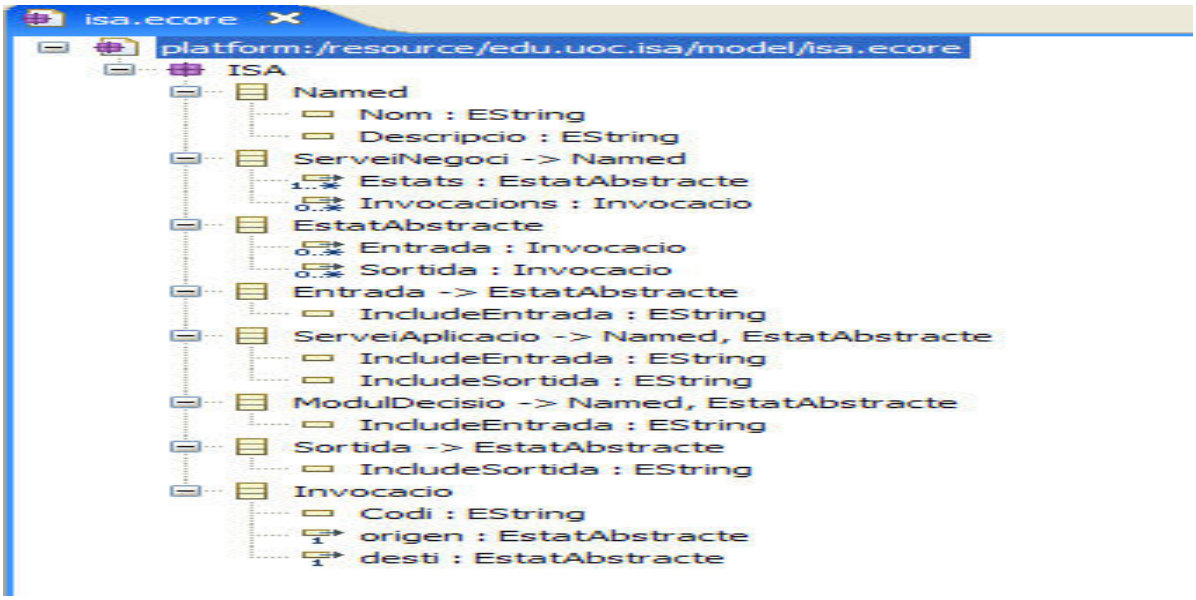


Figura 15 - Eclipse EMF: el model ecore

Com es pot observar a la figura anterior, a l'arbre apareixen els quatre conceptes principals de l'exemple ISA (*Entrada*, *ServeiAplicacio*, *ModulDecisio* i *Sortida*), juntament amb totes les relacions existents entre aquestos conceptes. En aquest cas, a més, s'ha dotat al model d'unes classes auxiliars (*EstatAbstracte* i *Invocacio*) per tal de poder definir atributs dins de les relacions (cas del *Codi de Sortida*).

A partir d'aquest metamodel, cal generar el codi que permetrà accedir a les dades del model. Per realitzar aquesta acció, caldrà seleccionar la opció "*Derive*" situada just damunt de la capça anomenada "*Domain Gen Model*" del tauler del projecte. Una vegada realitzada aquesta acció, obrirem l'arxiu resultant, anomenat "*isa.genmodel*", i situant-nos damunt del node principal, seleccionem l'opció "*Generate All*". Aquesta acció generarà automàticament tot un seguit de classes Java que serviran per poder accedir als models que estiguin basats en el metamodel definit.

Però amb les accions realitzades fins ara, només s'ha construït el metamodel i les classes d'accés a les dades mitjançant el *plugin* de EMF. Ara caldrà construir l'eina editora, que es la que utilitzaran els usuaris finals, mitjançant el *plugin* GMF. Val a dir que també podríem emprar alguna eina UML, però en el nostre cas, i a efectes d'analitzar la solució complerta, hem preferit crear-nos un editor gràfic propi.

En primer lloc, construirem la notació del model seleccionant la opció "*Derive*" situada just a la esquerra la capça anomenada "*Graphical Def Model*" del tauler del projecte. Aquesta acció ens generarà un arxiu anomenat "*isa.gmfgraph*", el qual haurem d'editar per tal de definir la notació dels diferents conceptes del domini. En aquest cas, l'arxiu resultant serà el següent:

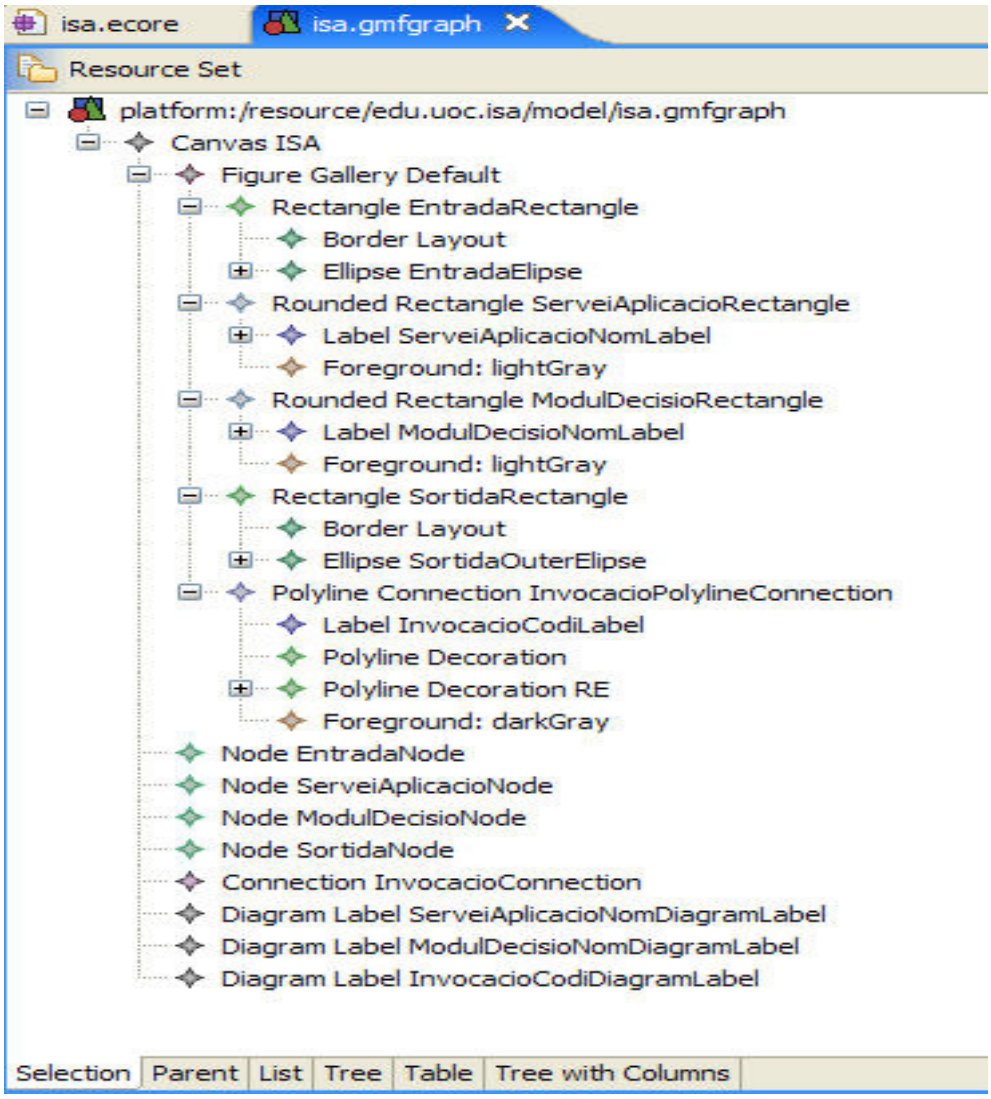


Figura 16 – Eclipse GMF: la notació dels conceptes del domini

A la figura anterior es pot veure com s'han definit figures geomètriques per tal de representar els diferents conceptes del domini.

A continuació definirem la paleta de components de l'editor del model seleccionant la opció "*Derive*" situada just a la esquerra de la capça anomenada "*Tooling Def Model*" del tauler del projecte. Aquesta acció ens

generarà un arxiu anomenat "*isa.gmftool*", el qual haurem d'editar per tal de definir els diferents components gràfics de la paleta. En aquest cas, l'arxiu resultant serà el següent:

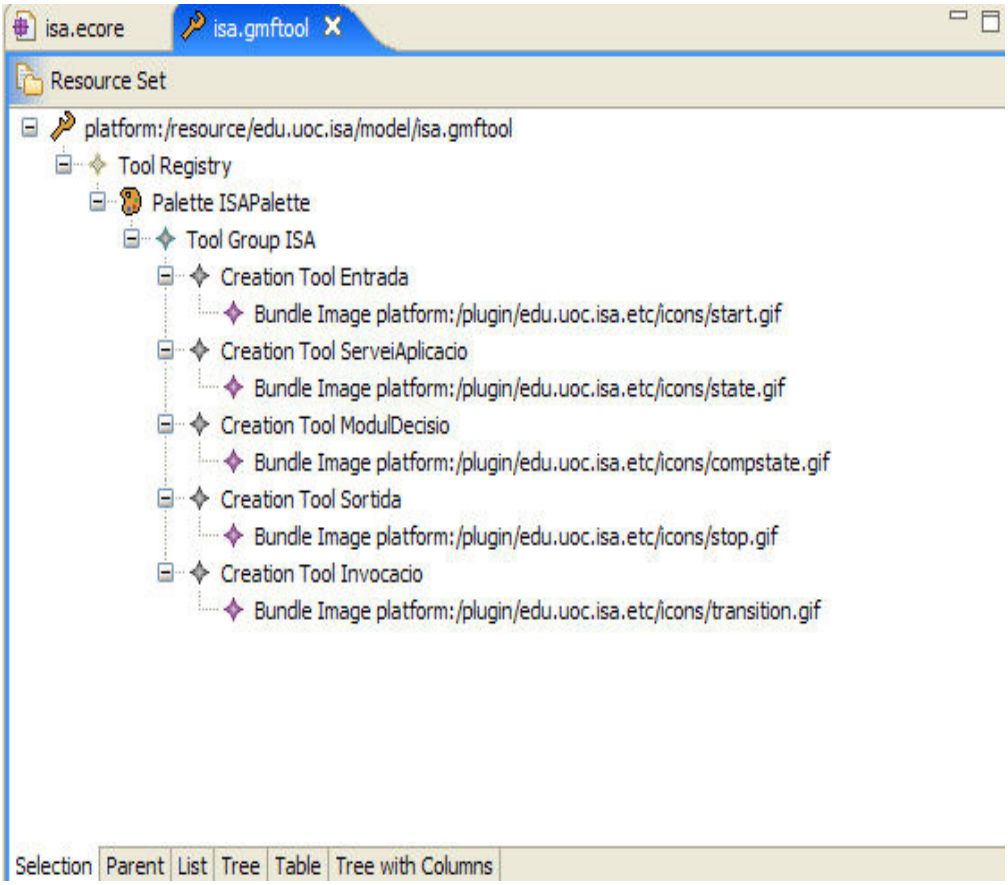


Figura 17 – Eclipse GMF: la paleta de components

Acte seguit, caldrà combinar els diferents arxius creats anteriorment per tal de relacionar els components del model amb la notació gràfica i la paleta de components. Aquesta acció la realitzarem seleccionant la opció "*Combine*" situada just a la esquerra la capça anomenada "*Mapping Model*" del tauler del projecte, que ens generarà un arxiu anomenat "*isa.gmfmap*".

Finalment caldrà generar el codi d'un nou *plugin* Eclipse, que es el que contindrà la interfície gràfica que faran servir els usuaris finals. Seleccionarem l'opció "*Transform*" situada just damunt de la capça anomenada "*Diagram Editor Gen Model*" del tauler del projecte i que ens generarà un arxiu anomenat "*isa.gmfgen*", i després l'opció "*Generate Diagram Editor*" que es troba a la mateixa capça i que ens generarà automàticament tot el codi necessari.

A partir d'aquest moment, ja estem en disposició de provar l'editor del diagrama. Seleccionarem l'opció del menú "*Run → Run*" i escollirem "*Eclipse Application*" com el tipus d'aplicació a executar. Aquesta acció ens obrirà una nova instància de Eclipse sobre la qual podrem crear un projecte de tipus "*Example → ISA Diagram*". Ens apareixerà llavors un editor gràfic que és el que el desenvolupador utilitzarà per modelar el sistema. En el nostre cas, modelarem el nostre exemple ISA:

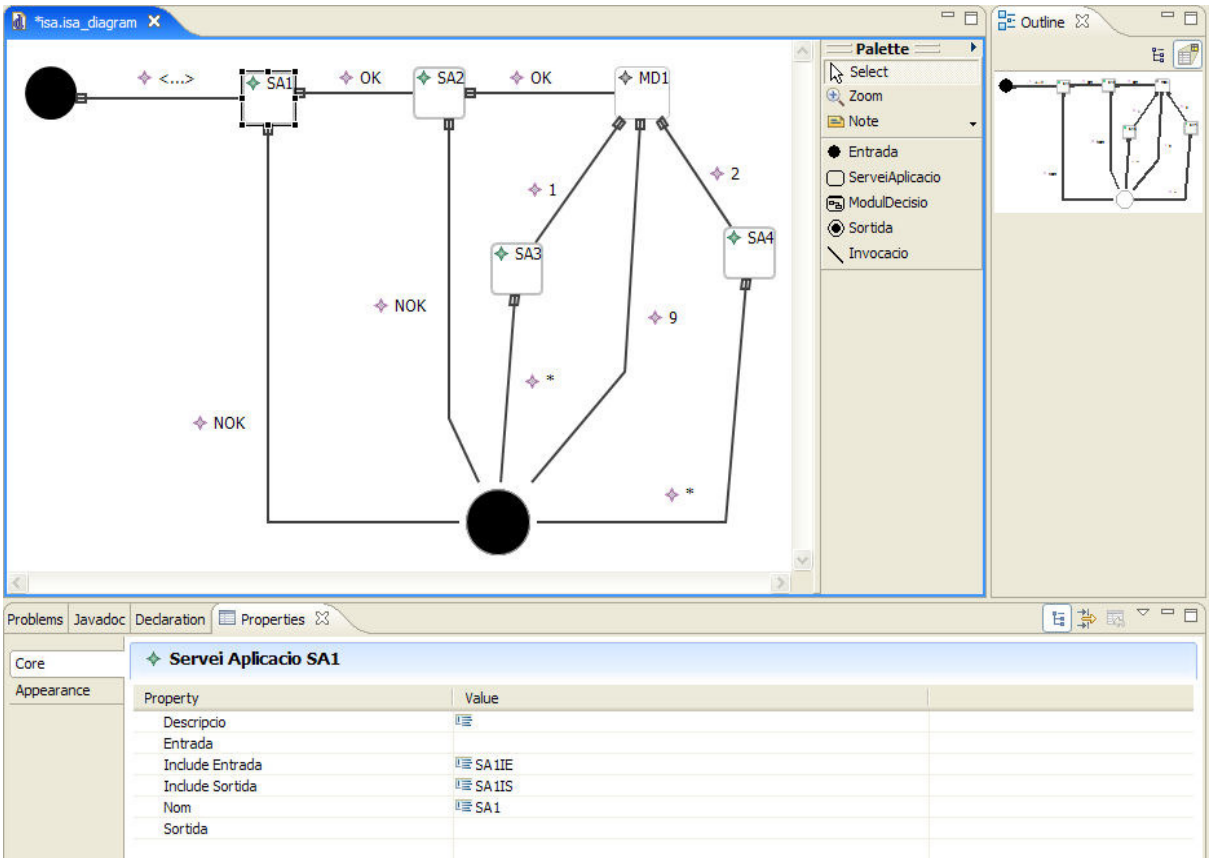


Figura 18 – Eclipse GMF: el nostre editor

La figura anterior és l'eina o interfície que veurà i utilitzarà l'usuari final, es a dir, el desenvolupador. A la part central apareix el diagramador per tal de que l'usuari dissenyi la seva solució, en aquest cas, l'exemple ISA. A la part dreta del diagramador apareix la paleta de components amb les diferents classes i figures definides anteriorment. I a la part inferior, apareixen les propietats del component que està seleccionat al diagrama.

Una vegada comprovat que l'editor es correcte, passarem a definir les regles del domini, es a dir, les restriccions que cal aplicar a les diferents classes i relacions del domini. Aquestes restriccions es defineixen manualment mitjançant el llenguatge OCL. A tall d'exemple, el codi que ve a continuació valida que estiguin informats els atributs de la classe *ServeiAplicació*:

```
import ISA;

context Named
ERROR "El nom ha de contenir un identificador vàlid" :
      Nom.matches("[a-zA-Z]+[a-zA-Z_0-9]*");

context ServeiAplicacio
ERROR "S'ha d'informar la Include d'Entrada del Servei d'Aplicacio":
      this.IncludeEntrada != null;

context ServeiAplicacio
ERROR "S'ha d'informar la Include de Sortida del Servei d'Aplicacio":
      this.IncludeSortida != null;
```

Una vegada comprovades que les restriccions funcionen correctament, només caldrà definir la conversió del model a codi font, es a dir, com es realitzarà la generació automàtica de codi. Aquesta definició es realitzarà mitjançant plantilles codificades en un llenguatge propietari de oAW anomenat xPand. En el nostre cas, la plantilla a codificar serà la següent:

```
«EXTENSION templates::GeneratorUtil»

«DEFINE Root FOR ISA::ServeiNegoci»
      «FILE Nom.toFirstUpper()+".pli"»
/*************************************************/
/* Servei de negoci «Nom» */
/* Generat automàticament amb openArchitectureWare */
/*************************************************/
«Nom»: PROC OPTIONS(MAIN);

/*****************************/
/*- Declaració de Components -*/
/*****************************/
/*- Include d'Entrada -*/
IE(«estatEntrada(this).IncludeEntrada»);
«FOREACH llistaServeiAplicacio(this) AS sa-»
/*- Servei d'Aplicació «sa.Nom»: «sa.Descripcio» -*/
SN(«sa.Nom») INCLUDEIN(«sa.IncludeEntrada»)
INCLUDEOUT(«sa.IncludeSortida»);
«ENDFOREACH-»
«FOREACH llistaModulDecisio(this) AS md-»
/*- Mòdul de Decisió «md.Nom»: «md.Descripcio» -*/
MD(«md.Nom») INCLUDEIN(«md.IncludeEntrada»);
«ENDFOREACH-»
/*- Include de Sortida -*/
IS(«estatSortida(this).IncludeSortida»);

/*************************/
/*- Declaració del Fluxe -*/
/*************************/
«FOREACH Invocacions AS in-»
«IF in.origen.metaType == ISA::Entrada-»
FLUXEINICI TO(«in.desti.Nom»);
«ENDIF-»
«IF in.desti.metaType == ISA::ServeiAplicacio && in.origen.metaType !=
ISA::Entrada-»
FLUXE FROM(«in.origen.Nom») TO(«in.desti.Nom») CODI(«in.Codi»);
«ENDIF-»
«IF in.desti.metaType == ISA::ModulDecisio && in.origen.metaType !=
ISA::Entrada-»
FLUXE FROM(«in.origen.Nom») TO(«in.desti.Nom») CODI(«in.Codi»);
«ENDIF-»
«IF in.desti.metaType == ISA::Sortida-»
FLUXEFINAL FROM(«in.origen.Nom») CODI(«in.Codi»);
«ENDIF-»
«ENDFOREACH-»

END «Nom»;
      «ENDFILE»
«ENDDEFINE»
```

Els codis identificats amb els símbols “<< ... >>” es corresponen a codi en llenguatge Xpand, i la resta, a codi que s’escriu directament en l’arxiu de sortida. Aquest llenguatge permet interactuar amb les dades del model com si

fossin objectes, permeten accedir al seus atributs i, inclús, definir nous objectes derivats dels que podem trobar al model. Per exemple, en el nostre cas hem creat els següents objectes derivats:

```
import isa;

ISA::Entrada estatEntrada(ISA::ServeiNegoci sn):
      sn.Estats.typeSelect(ISA::Entrada).get(0);

Collection llistaServeiAplicacio(ISA::ServeiNegoci sn):
      sn.Estats.typeSelect(ISA::ServeiAplicacio);

Collection llistaModulDecisio(ISA::ServeiNegoci sn):
      sn.Estats.typeSelect(ISA::ModulDecisio);

ISA::Sortida estatSortida(ISA::ServeiNegoci sn):
      sn.Estats.typeSelect(ISA::Sortida).get(0);
```

I ja només ens queda definir el procés que realitzarà la transformació d'aquesta plantilla al codi. Aquest procés es defineix mitjançant XML d'acord amb una sintaxi pròpia de oAW con si fos un flux de processos:

```
<workflow>
      <property file="workflow.properties"/>

      <component id="xmiParser"
class="org.openarchitectureware.emf.XmiReader">
            <metaModelPackage value="ISA.ISAPackage"/>
            <modelFile value="${modelFile}"/>
            <outputSlot value="model"/>
            <firstElementOnly value="true"/>
      </component>

      <component id="dirCleaner"
class="org.openarchitectureware.workflow.common.DirectoryCleaner" >
            <directories value="${srcGenPath}"/>
      </component>

      <component id="batchErrors"
class="org.openarchitectureware.check.CheckComponent">
            <metaModel id="mm"
class="org.openarchitectureware.type.emf.EmfMetaModel">
              <metaModelPackage value="ISA.ISAPackage"/>
            </metaModel>
            <checkFile
value="edu::uoc::isa::model::isa::constraints::ISABatchErrors"/>
            <expression value="model.eAllContents.union( {model} )"/>
      </component>

      <component id="liveErrors"
class="org.openarchitectureware.check.CheckComponent">
            <metaModel id="mm"
class="org.openarchitectureware.type.emf.EmfMetaModel">
              <metaModelPackage value="ISA.ISAPackage"/>
            </metaModel>
            <checkFile
value="edu::uoc::isa::model::isa::constraints::ISALiveErrors"/>
            <expression value="model.eAllContents.union( {model} )"/>
      </component>
```

```
      <component id="generator"
class="org.openarchitectureware.xpand2.Generator" skipOnErrors="true">
            <metaModel id="mm"
class="org.openarchitectureware.type.emf.EmfMetaModel">
                  <metaModelPackage value="ISA.ISAPackage"/>
            </metaModel>
            <expand value="templates::Root::Root FOR model"/>
            <genPath value="${srcGenPath}"/>
            <srcPath value="${srcGenPath}"/>
      </component>

</workflow>
```

En l'exemple anterior podem veure que el procés de generació conté 5 passos:

- *xmiParser*: aquest pas es l'encarregat de definir quin és el model a llegir i a quin metamodel es correspon.
- *dirCleaner*: aquest pas neteja el directori de sortida.
- *bathErrors*: aquest pas comprova que les restriccions del model es compleixen.
- *liveErrors*: aquest pas també comprova que les restriccions del model es compleixen.
- *generator*: aquest pas és l'encarregat de generar el codi.

Si finalment executem aquest procés mitjançant les opcions del menú "*Run → As oAW workflow*", el codi resultant serà el següent:

```
/************************************************/
/* Servei de negoci SNTest */
/* Generat automàticament amb openArchitectureWare */
/************************************************/
SNTest: PROC OPTIONS(MAIN);

/*****************************/
/*- Declaració de Components -*/
/*****************************/
/*- Include d'Entrada -*/
IE(SN-IE);
/*- Servei d'Aplicació SA1:  -*/
SN(SA1) INCLUDEIN(SA1IE) INCLUDEOUT(SA1IS);
/*- Servei d'Aplicació SA2:  -*/
SN(SA2) INCLUDEIN(SA2IE) INCLUDEOUT(SA2IS);
/*- Servei d'Aplicació SA3:  -*/
SN(SA3) INCLUDEIN(SA3IE) INCLUDEOUT(SA3IS);
/*- Servei d'Aplicació SA4:  -*/
SN(SA4) INCLUDEIN(SA4IE) INCLUDEOUT(SA4IS);
/*- Mòdul de Decisió MD1:  -*/
MD(MD1) INCLUDEIN(MD1IE);
/*- Include de Sortida -*/
IS(SN-IS);

/*************************/
/*- Declaració del Fluxe -*/
/*************************/
FLUXEINICI TO(SA1);
FLUXE FROM(SA1) TO(SA2) CODI(OK);
FLUXE FROM(SA2) TO(MD1) CODI(OK);
FLUXE FROM(MD1) TO(SA3) CODI(1);
```

```
FLUXE FROM(MD1) TO(SA4) CODI(2);
FLUXEFINAL FROM(SA1) CODI(NOK);
FLUXEFINAL FROM(SA2) CODI(NOK);
FLUXEFINAL FROM(SA3) CODI(*);
FLUXEFINAL FROM(SA4) CODI(*);
FLUXEFINAL FROM(MD1) CODI(9);

END SNTest;
```

Com a últim pas, caldrà desplegar la solució complerta en forma de *plugins* Eclipse, que seran entregats als usuaris finals.

El codi complert de l'exemple ISA desenvolupat amb l'eina Eclipse més els components EMF, GMF i oAW es pot trobar a l'apartat "Annex III – Exemple amb Eclipse EMF + GMF + oAW" d'aquest mateix document.

3.7 Borland Together 2006

La proposta de Borland dins de l'àmbit de desenvolupament dirigit per models es una eina anomenada Together 2006. Aquesta eina es una plataforma visual de modelatge dissenyada per donar suport als arquitectes, desenvolupadors, dissenyadors, analistes de processos i modeladors de dades en el disseny i construcció d'aplicacions de software.

La tecnologia que proporciona aquesta eina es conforme als estàndards de MDA: *Unified Modeling Language* (UML), *XML Metadata Interchange* (XMI), *Query View Transformation* (QVT) i *Object Constraint Language* (OCL). Així doncs, el llenguatge de modelatge que s'utilitzarà és el UML, encara que, al estar basada en Eclipse, ens permetrà importar models de tipus *ecore*. A més, l'eina disposa de tota una sèrie d'assistents que permetran realitzar transformacions de model a model (com per exemple, de PIM a PSM), així com generació automàtica de codi.

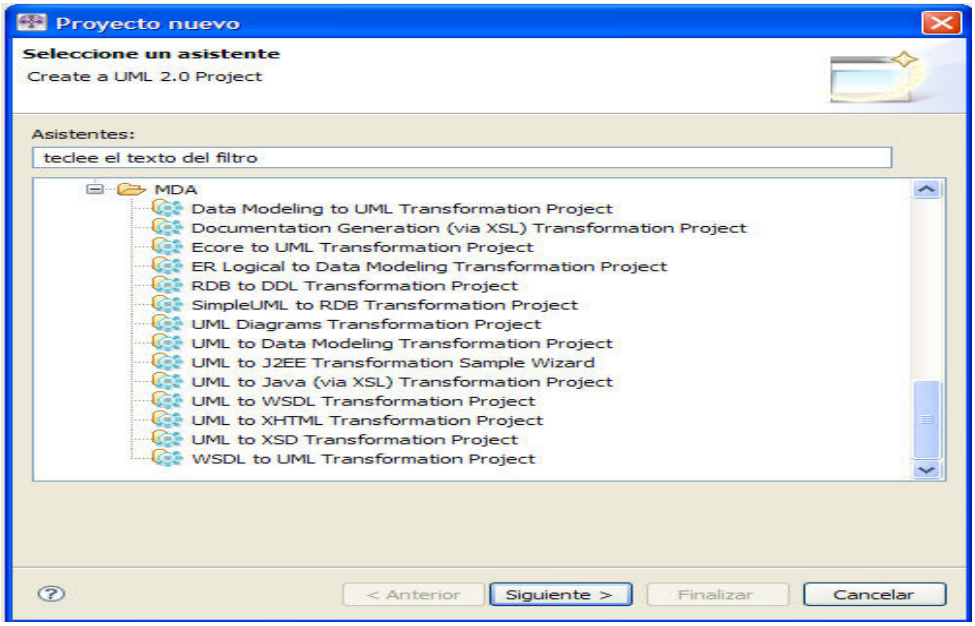


Figura 19 - Borland Together: assistent per la creació de projectes MDA

Abans de començar a desenvolupar el nostre exemple, hem de resoldre un primer problema. L'exemple ISA que cal modelar es basa en un entorn procedural, es a dir, no orientats a objectes. Aquesta situació fa que tingui un difícil encaix dintre dels diagrames UML. Després d'analitzar la situació, es determina que el diagrama a utilitzar serà el de màquines d'estat, ja que si analitzem a fons l'exemple, veurem que el flux, en realitat, es tracta d'una màquina d'aquest tipus.

Així doncs, el primer que farem es dissenyar el nostra problema directament sobre un diagrama de màquina d'estats. Per realitzar aquesta acció crearem un nou projecte mitjançant les opcions del menú "*Archivo → Nuevo → Proyecto*" i escollint com a tipus de projecte "*Modeling → UML 2.0 Project*". A aquest nou projecte l'anomenarem "*uoc.edu.isa.model"*. A la pantalla següent, especificarem "*State Machine*" com el tipus de diagrama a crear per defecte i l'anomenarem "*SNTest"*. L'acció d'escollir el tipus de diagrama comporta també definir, en aquest cas implícitament, el *CIM* que emprarà la solució, es a dir, el model amb els conceptes del domini sobre el que treballarem. En el nostre cas, els conceptes seran els d'una màquina d'estats.

Acte seguit ens apareixerà un diagrama de màquina d'estats. A diferencia de les eines analitzades anteriorment, i al tractar-se d'una eina enfocada a MDA, el desenvolupador dissenyarà directament el seu problema sobre aquest diagrama sense necessitat de dissenyar prèviament un llenguatge i una notació especifica. Així doncs, ja podem modelar el nostre exemple ISA:

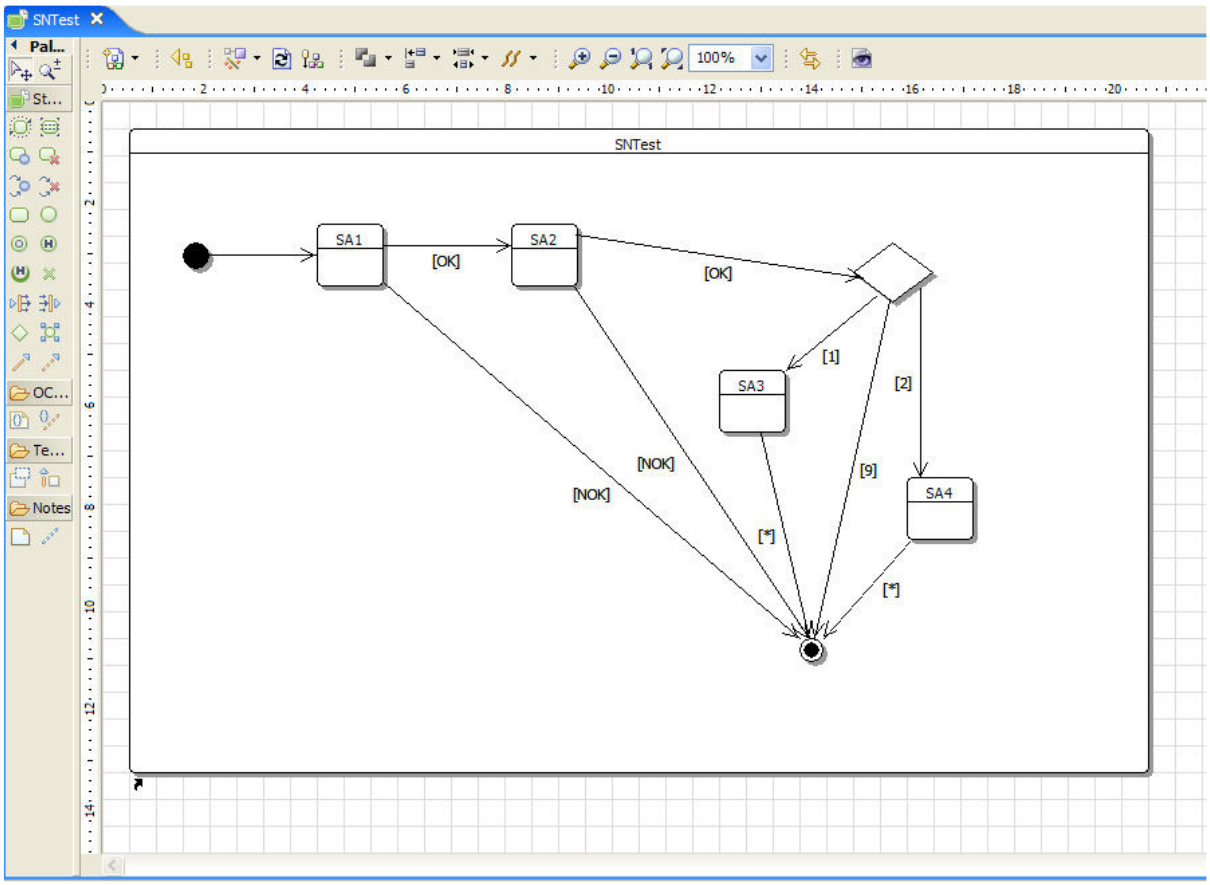


Figura 20 - Borland Together: la màquina d'estats

Com podem veure a la figura anterior, i tal com s'ha comentat anteriorment, al diagrama no apareix cap dels conceptes *ISA*, sinó que es treballa directament sobre els conceptes d'una màquina d'estats. Segons els estàndards de MDA, aquest primer diagrama es correspon al model independent de plataforma (*PIM*).

Acte seguit, caldrà transformar aquest model en un model específic de plataforma (*PSM*). En el nostre cas, això ho aconseguirem definint atributs específics dins de cadascun dels estats, com per exemple:

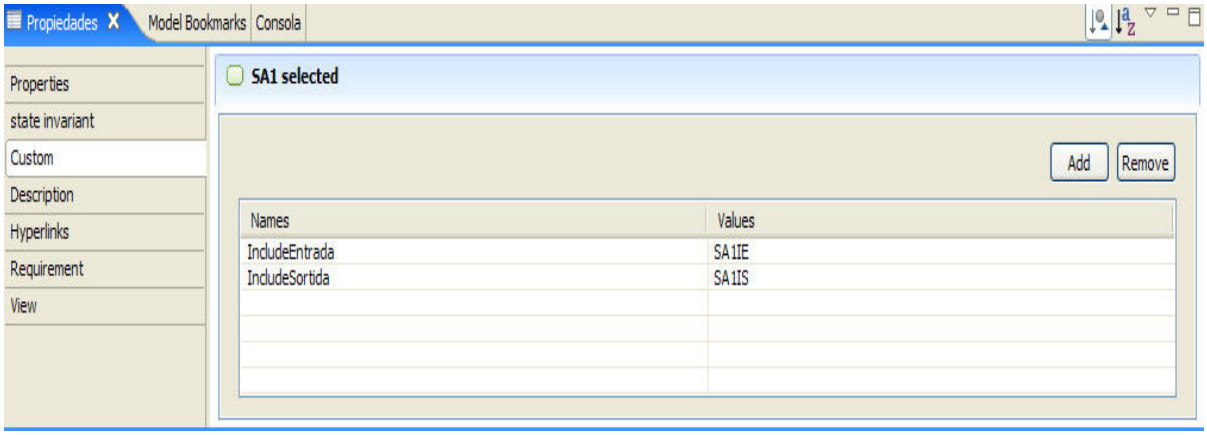


Figura 21 - Borland Together: definició d'atributs

Una vegada construït aquest model, ja podem passar a definir la generació de codi. Per realitzar aquesta acció, caldrà crear un projecte de transformació mitjançant les opcions del menú "*Archivo → Nuevo → Proyecto*" i escollint com a tipus de projecte "*Modeling → MDA Transformation Project*". A aquest nou projecte l'anomenarem "*uoc.edu.isa.uml2text*". A la pantalla anomenada "*Transformations*" caldrà indicar que es vol crear un "*Model to Text Transformation*" i donar un nom a la classe que s'encarregarà de realitzar aquesta transformació, que en el nostre cas, s'anomenarà "*GeneradorISA.java*". Per últim, caldrà indicar el tipus de model d'entrada (el *CIM*), que en aquest cas, serà "*Together → UML 2.0 Project → uml20 → statemachines → StateMachine*".

Aquest assistent ens crearà una classe Java buida que caldrà completar manualment. Cal dir que aquesta tasca es bastant laboriosa, sobretot amb models complicats, ja que el codi a completar no està basat en plantilles, sinó que cal codificar-ho tot en llenguatge Java des d'inici. A més, l'accés al model també es força complicat, entre altres coses perquè es basa en metamodels propis de Borland i la documentació no es prou extensa.

A tall d'exemple, el codi següent pertany a la generació d'una part de la definició dels components:

```
public void
generate(com.borland.tg.emfapi.uml20.statemachines.StateMachine model,
String outputPath) {
      writer = new PrintWriter(new FileWriter(file));
```

```
        writer.println("/*********************************************
        ***/");
        writer.println("/* Servei de negoci " + model.getName() + "
        */");
        writer.println("/* Generat automàticament amb Borland Together
        */");
        writer.println("/*********************************************
        ****/");
        writer.println(model.getName() + ": PROC OPTIONS(MAIN);");
        writer.println("");
        EList regions = model.getRegions();
        for (Iterator iter_regions = regions.iterator();
        iter_regions.hasNext();) {
              Region region_element = (Region) iter_regions.next();
              writer.println("/*****************************/");
              writer.println("/*- Declaració de Components -*/");
              writer.println("/*****************************/");
              EList vertexs = region_element.getSubvertexs();
              for (Iterator iter_vertexs = vertexs.iterator();
        iter_vertexs.hasNext();) {
                    vertex_element = (Vertex) iter_vertexs.next();
                    if (vertex_element.getClass().getName() ==
              "com.borland.tg.emfapi.uml20.statemachines.impl.StateImpl")
              {
                          st = (State) vertex_element;
                          writer.println("/*- Servei d'Aplicació " +
                    st.getName() + ": " + st.getDescription() + " -*/");
                          writer.println("SN(" + st.getName() + ")
                    INCLUDEIN(" + st.getPropertyValue("IncludeEntrada") +
                    ") INCLUDEOUT(" +
                    st.getPropertyValue("IncludeSortida") + ");");
                    } else if (vertex_element.getClass().getName() ==
"com.borland.tg.emfapi.uml20.statemachines.impl.PseudoStateImpl") {
                          ps = (PseudoState) vertex_element;
                          if
                    ps.getKind().equals(PseudoStateKind.get(PseudoStateKi
                    nd.INITIAL))) {
                          writer.println("/*- Include d'Entrada -*/");
                          writer.println("IE(" +
                    ps.getPropertyValue("IncludeEntrada") + ");");
                          }
                    }
              }
        }
}
```

Com es pot veure a l'exemple anterior, l'entrada al mètode de generació es un objecte de tipus "*com.borland.tg.emfapi.uml20.statemachines.StateMachine*". Dins d'aquest objecte trobarem la relació de nodes (vèrtexs) que conté la màquina d'estats i transicions entre aquestos nodes. A partir d'aquí, caldrà anar consultant els atributs dels nodes per tal de determinar la seva topologia (entrada, sortida, decisió, ...).

Finalment, una vegada completada la classe de generació, ens situarem damunt la màquina d'estats que hem creat anteriorment amb el nom *SNTest* i seleccionarem el menú "*Model → Apply Transformation → Model to Text → From Workspace*". A l'assistent que ens apareixerà, caldrà seleccionar el component *GeneradorISA.java* que es troba al directori *src_generated* del

projecte *edu.uoc.isa.uml2text* i com a destí del codi a generar indicar el projecte *edu.uoc.isa.uml2text*. El codi que se'ns generarà serà el següent:

```
/************************************************/
/* Servei de negoci SNTest */
/* Generat automàticament amb Borland Together     */
/************************************************/
SNTest: PROC OPTIONS(MAIN);

/*****************************/
/*- Declaració de Components -*/
/*****************************/
/*- Include d'Entrada -*/
IE(SN-IE);
/*- Include de Sortida -*/
IS(SN-IS);
/*- Servei d'Aplicació SA1: Servei d'Aplicació 1 -*/
SN(SA1) INCLUDEIN(SA1IE) INCLUDEOUT(SA1IS);
/*- Servei d'Aplicació SA2: Servei d'Aplicació 2 -*/
SN(SA2) INCLUDEIN(SA2IE) INCLUDEOUT(SA2IS);
/*- Servei d'Aplicació SA3: Servei d'Aplicació 3 -*/
SN(SA3) INCLUDEIN(SA3IE) INCLUDEOUT(SA3IS);
/*- Servei d'Aplicació SA4: Servei d'Aplicació 4 -*/
SN(SA4) INCLUDEIN(SA4IE) INCLUDEOUT(SA4IS);
/*- Mòdul de Decisió MD1: Mòdul de Decisió 1 -*/
MD(MD1) INCLUDEIN(MD1E);

/************************/
/*- Declaració del Fluxe -*/
/************************/
FLUXEINICI TO(SA1)
FLUXE FROM(SA1) TO(SA2) CODI(OK);
FLUXEFINAL FROM(SA1) CODI(NOK);
FLUXEFINAL FROM(SA2) CODI(NOK);
FLUXE FROM(SA2) TO(MD1) CODI(OK);
FLUXEFINAL FROM(SA3) CODI(*);
FLUXEFINAL FROM(SA4) CODI(*);
FLUXE FROM(MD1) TO(SA3) CODI(1);
FLUXE FROM(MD1) TO(SA4) CODI(2);
FLUXEFINAL FROM(MD1) CODI(9);

END SNTest;
```

Malauradament, aquesta eina té limitada la capacitat de definició de regles del domini dins dels diagrames de màquines d'estat, ja que només permet definir restriccions a nivell de les condicions de guarda de les transicions. Per aquest motiu, ha estat impossible introduir dins d'aquest projecte algun exemple de restricció. De totes maneres, l'eina si que permet introduir restriccions més complexes mitjançant l'estàndard *Object Constraint Language* (OCL) en altres tipus de diagrames.

El codi complert de l'exemple ISA desenvolupat amb l'eina Borland Together 2006 es pot trobar a l'apartat "Annex IV – Exemple amb Borland Together 2006" d'aquest mateix document.

3.8 Resum comparatiu de les eines analitzades

Amb les tres eines analitzades en aquest treball s'ha aconseguit modelar l'exemple ISA i generar el codi automàticament. L'enfocament i la dificultat, però, ha estat sensiblement diferent.

En el cas de l'eina Microsoft DSL Tools, crear un llenguatge de domini específic per l'exemple ISA ha estat una tasca realment senzilla. Destaca, però, la impossibilitat de mostrar la informació de les etiquetes, tant dels diferents conceptes com de les relacions. Segons el manual de l'eina, aquesta tasca es pot realitzar mitjançant programació, però no s'ha localitzat cap referència de com es pot implementar. Per contra, la codificació de les restriccions al model així com la codificació de les plantilles per generar codi automàticament no ha representat cap problema. Cal destacar també, com a punt negatiu, la no alineació amb cap estàndard, situació que les empreses solen evitar ja que comporta lligar-se a un sol fabricant.

En quant a la combinació d'eines Eclipse EMF + GMF + oAW, la creació del metamodel, la definició de restriccions i la codificació de la plantilla per generar codi automàticament, també ha estat una tasca senzilla. A diferència de l'eina Microsoft DSL Tools, el disseny de l'eina DSL amb el component GMF ha estat una mica més complicat, sobretot al no disposar d'un assistent gràfic. Encara així, aquesta tasca no ha representat cap problema. Es destaca positivament el fet de que admeti com a models d'entrada tant el llenguatge de modelatge UML com un llenguatge específic de domini. Aquesta característica habilitarà l'ús d'aquesta eina en moltes més situacions, per exemple, en el desenvolupament d'una lògica de negoci d'una aplicació orientada a objectes, com pot ser J2EE, l'ús de UML té un clar avantatge sobre la creació d'un DSL.

Pel que fa a l'eina Borland Together 2006, s'ha trobat la dificultat d'encaixar dins de UML un model que no pertany a l'àmbit de l'orientació a objectes. En aquest cas, s'ha resolt emprant un diagrama d'estats, però segurament hi ha situacions en les que no es podrà modelar el problema mitjançant UML. Es destaca també la dificultat per codificar el generador de codi. La utilització de plantilles es molt més senzilla de codificar que no pas la programació sencera d'una classe específica. L'ús de metamodels propietaris tampoc ha ajudat a alleugerar la situació, sobretot tenint en compte la documentació tant escassa que aporta el producte. De igual manera es destaca la impossibilitat de definir les restriccions mitjançant OCL, degut a una mancança del producte.

I finalment, mostren la taula comparativa amb les característiques de cadascuna de les eines analitzades, d'acord amb els criteris d'avaluació definits anteriorment:

Característica	Eina		
	DSL Tools	**Eclipse EMF + GMF + oAW**	**Together 2006**
Metodologia	DSM	DSM i MDA	MDA
Plataforma d'execució	Windows	Windows, Linux, Solaris, AIX, HP-UX, Mac OSX	Windows, Linux, Solaris, Mac OSX
Llenguatges de modelatge suportats	DSL gràfic	UML 1.1, DSL textual, DSL gràfic	UML 2.0
Possibilitat de transformar models en altres models	No	Si	Si
Llenguatge per definir transformacions entre models	N/A	XTend	Java
Suport de perfils UML	No	No	Si
Possibilitat de definir restriccions en els models	Si	Si	Si
Llenguatge per definir restriccions en els models	C#, VisualBasic	OCL	OCL
Formats dels generadors de codi	Plantilles	Plantilles	Codi
Incorpora generadors de codi predefinits	No	No	No
Possibilitat de modificar els generadors de codi	Si	Si	Si
Llenguatge per definir els generadors de codi	C#, VisualBasic	XPand	Java
Importació de models	No	ecore, XMI	XMI 2.0, ecore, Rational Rose, Rational XDE
Exportació de models	No	ecore, XMI	XMI 2.0
Integració amb eina de requisits	No	No	Borland CaliberRM, Rational Requisite Pro
Integració amb eina de control de versions	Visual SourceSafe	CVS, SubVersion	Borland StarTeam, CVS, Subversion

4 Conclusions

En aquest treball s'han descrit les principals característiques de les dos metodologies de desenvolupament dirigides per models més significatives i s'ha mostrat com aquestes són implementades en algunes de les eines existents al mercat actualment. Però si bé s'ha aconseguit modelar un exemple d'un problema concret i generar el codi automàticament a partir del model, cap de les aproximacions es perfecta.

Mentre que la visió de MDA és impecable en la teoria, a la pràctica pocs desenvolupadors han fet l'esforç d'aprendre UML en tota la seva globalitat. El motiu es que UML és extremadament complicat en algunes àrees mentre que en altres no està prou definida. Per posar un cas, en el exemple desenvolupat amb l'eina Borland Together 2006 s'ha vist com intentar encaixar dins de UML un model d'un problema que no pertany a l'àmbit de la orientació a objectes es una tasca complicada i no sempre possible. Cal tenir en compte, a més, que el consens general dins la comunitat de modelatge es que la comunitat de sistemes en temps real i, en alguns casos, els proponents de MDA, es van apropiar de l'esforç de definició de UML 2.0, i que, per tant, la majoria de desenvolupadors es van sentit apartats de les discussions sobre com millorar UML. Per això, hi ha algunes veus que diuen que UML no es tan estàndard com se'ns vol fer creure.

De igual manera, hi ha qui pensa que els estàndards definits per OMG, en concret MOF i XMI, són extremadament complicats i que, per tant, no es poden implementar d'una forma eficient amb la tecnologia actual ni són tant interoperables. En aquest sentit, els metamodels emprats per l'eina Borland Together 2006 han estat complicats d'entendre, i les restriccions mitjançant el llenguatge OCL no s'han pogut implementar degut a una mancança del producte.

Per altre banda, hem vist com implementar l'exemple ISA mitjançant la metodologia DSM amb l'eina Microsoft DSL Tools ha resultat una tasca senzilla. Però això no sempre serà així, ja que en molts casos serà necessari adoptar tot un conjunt de models per tal de modelar una aplicació complerta. Per exemple, per modelar una sistema amb una arquitectura de tres capes en J2EE es necessari un model per la lògica de negoci de l'aplicació, un altre model per la interfície gràfica o un model pels esquemes de bases de dades. En aquest sentit, el modelatge de la lògica de negoci que s'implementa mitjançat un llenguatge orientat a objectes com Java no té gaire sentit sense emprar UML i, per tant, en aquest cas, és un cost inútil dedicar esforços en definir un llenguatge de domini específic.

A més, la visió de DSM no sempre encaixa en organitzacions que o bé no s'ho poden permetre o bé volen minimitzar l'esforç dedicat a les eines de desenvolupament, tant en inversió en les mateixes eines com en la educació que els desenvolupadors han de rebre. En el mateix sentit, el fet de que DSM no tingui estandarditzada la tecnologia a emprar, pot suposar un fre en la seva

adopció per part de les empreses, degut principalment, al temor a adquirir una tecnologia propietària.

D'igual manera, la generació complerta de codi ha estat un objectiu de la industria del software durant molts anys i que, desafortunadament, ofereix actualment poques possibilitat d'incrementar l'abstracció del disseny degut principalment a que molts dels llenguatges de modelatge estan basats en el món del codi. Com a conseqüència, els desenvolupadors es troben que els hi és més fàcil programar el codi directament que no pas modelar la funcionalitat i comportament del sistema. A més, les limitacions actuals en la generació automàtica de codi provoquen que els desenvolupadors hagin de completar el codi generat i que, per tant, tinguin que mantenir la mateixa informació en dos llocs, en el codi i en el model. En el mateix sentit, una de les queixes més freqüents dels desenvolupadors es que el procés de generació acaba produint codi poc òptim i, en els casos en que cal completar-ho, de difícil enteniment i, per tant, de difícil manteniment.

L'exemple ISA desenvolupat en aquest treball és un clar exemple de la dificultat per modelar el codi complert. El modelatge d'un flux de processos no ha representat gaires problemes en cap de les eines analitzades, però no passa el mateix si en l'exemple s'hagués tingut que modelar la lògica de negoci corresponent al serveis d'aplicació o als mòduls de decisió. En el cas de MDA, ens hagués estat impossible modelar aquesta lògica mitjançant UML, bàsicament perquè l'entorn destí no està orientat a objectes. I amb DSM, s'hauria tingut que fer un esforç molt important en definir un llenguatge de modelatge específic per representar els diferents conceptes que poden aparèixer en llenguatge procedural com PL/I.

En conclusió, DSM pot semblar una estratègia més encertada que no pas MDA en el cas de grans organitzacions, ja que està més orientada a la resolució d'un problema en un domini concret i no és tant generalista. Per contra, MDA segurament és més encertat en petites companyies que no poden assumir la inversió necessària per desenvolupar i mantenir un DSL o en el cas d'emprar una solució arquitectònica molt establerta en el mercat. En qualsevol cas, tenir una notació comuna ofereix clarament un potencial per millorar la comunicació entre els desenvolupadors, sobretot si tenim en compte l'alt nivell de rotació entre els desenvolupadors que afronten les empreses d'avui en dia.

En aquest sentit, la combinació dels principis de MDA i DSM està arrelant cada vegada més entre la comunitat de modelatge i entre els diferents fabricants d'eines de desenvolupament. Un exemple el tenim en una de les eines analitzades en aquest treball, Eclipse EMF + GMF + oAW, que, tal com s'ha vist en aquest treball, es un híbrid que combina les dues metodologies. De igual manera, en el cas de MDA s'estan fent esforços importants amb la utilització i definició de perfils UML, que pretenen dotar a UML d'un llenguatge més específic per tal de modelar i representar els conceptes específics d'un domini.

Deixem com a propostes de treball futurs una avaluació més exhaustiva de les eines per veure com es pot implementar una generació total de codi així com un anàlisi en més detall del perfils UML.

5 Glossari

Common Warehouse Metamodel (CWM): especificació de OMG per la integració de magatzems de dades.

Computation Independent Model (CIM): vista d'un sistema que es centra en l'entorn i els requeriments del sistema.

Domain-Specific Language (DSL): un llenguatge de programació que ofereix, mitjançant les apropiades notacions i abstraccions, una potencia expressiva focalitzada, i usualment restrictiva, en un problema de domini particular.

Domain Specific Modeling (DSM): desenvolupament de software basat en la utilització de models representats mitjançant *Domain-Specific Languages*.

Domini: tema o àrea de coneixement adreçat a maximitzar la satisfacció dels requeriments dels accionistes d'un sistema.

Extensible Markup Language (XML): llenguatge d'etiquetatge i propòsit general que permet representar i intercanviar informació entre ordinadors o programari.

Metadada: dada que representa models.

Metamodel: model de models.

Meta-Object Facility (MOF): estàndard de OMG que permet la gestió de metadades i la definició de llenguatges de modelatge.

Model: especificació formal de la funció, estructura i comportament d'una aplicació o sistema.

Model Driven Architecture (MDA): aproximació en la especificació de sistemes que separa la especificació d'una funcionalitat de la especificació de la implementació d'aquesta funcionalitat en una plataforma tecnològica determinada.

Object Constraint Language (OCL): llenguatge que s'utilitza per escriure expressions sobre models de forma que estén la potencia expressiva de UML i permet crear models més precisos i més complerts.

Object Management Group (OMG): consorci internacional sense ànim de lucre d'empreses d'informàtica que es dedica al desenvolupament d'estàndards d'integració en un ampli ventall de tecnologies i industries.

Object-Oriented Programming, Systems, Languages and Applications (OOPSLA): conferència anual, principalment de caràcter acadèmic, que cobreix temes relacionats amb l'enginyeria de software i, en concret, amb el món de la orientació a objectes.

Platform Independent Model (PIM): model d'un sistema que no conté informació especifica d'una plataforma.

Platform Specific Model (PSM): model d'un sistema que conté informació relacionada amb una plataforma determinada.

Queries/Views/Transformations (QVT): estàndard de OMG que permet la transformació de models.

Software Factories: aproximació de Microsoft pel desenvolupament de software que es basa en un alt grau de automatització aplicant patrons d'ús i llenguatges visuals.

UML Profile: mecanisme d'extensió genèric per construir models UML adreçats a un domini concret.

Unified Modeling Language (UML): llenguatge estàndard de OMG per especificar la estructura i comportament d'un sistema.

XML Metadata Interchange (XMI): estàndard de OMG per l'intercanvi de metadades mitjançant XML.

6 Bibliografia

A. van Deursen, P. Klint, J. Visser. *Domain-Specific Languages: An Annotated Bibliography*. ACM SIGPLAN Notices, 2000, Volume 35, Issue 6, p. 26-36.

Borland Together 2006 Data Sheet. (http://www.borland.com/resources/en/pdf/products/together/together_datasheet.pdf)

D. Spinellis. *Notable design patterns for domain-specific languages*. The Journal of Systems and Software 56 (2001) 91-99.

Domain-Specific Languages Tools. (http://msdn.microsoft.com/vstudio/DSLTools/)

DSM Forum. *How to get started with DSM*. (http://www.dsmforum.org/how.html)

Eclipse Modeling Framework Project (EMF). (http://www.eclipse.org/modeling/emf/)

F. Truyen. *The Fast Guide to Model Driven Architecture. The Basics Of Model Driven Architecture (MDA)*. Cephas Consulting Corp, 2006.

G. Booch, A. Brown, S. Iyengar, J. Rumbaugh, B. Selic. *An MDA Manifesto*. MDA Journal, May 2004.

G. Booch, S. Fraser, R. C. Martin, S. J. Mellor, M. Lee, S. Garone, M. Fowler, D. C. Schmidt, M. Lenzi. *Translation: Myth or Reality? (panel)*. ACM SIGPLAN Notices, 1996, Volume 21, Issue 10, p. 441-443.

G. Cernosek. *Next-generation model-driven development*. Rational Software White Paper, December 2004. (ftp://ftp.software.ibm.com/software/rational/web/whitepapers/rsa-cernosek-wp.pdf)

ITarchitect. *Domain-Specific Modeling for Generative Software Development*. (http://www.itarchitect.co.uk/articles/display.asp?id=161)

J. Miller, J. Mukerji. *MDA Guide Version 1.0.1*. Object Management Group (OMG), 2003.

J. Tolvanen. *Domain-Specific Modeling for Full Code Generation*. Methods and Tools, Fall 2005, Volume 13, Number 3, p.14-23.

J.J. Garcia, P. Gomez, O. Sanchez. *Herramientas de Metamodelado: Microsoft DSL Tools vs MetaEdit+*. Departamento de Informática y Sistemas, Facultad de Informática, Universidad de Murcia.

K. Czarnecki, S. Helsen. *Feature based survey of model transformation approaches*. IBM Systems Journal, 2006, Volume 45, Number 3, p.621-645.

L. Fuentes, A.Vallecillo. *Una introducción a los perfiles UML*. Novatica, Nº 168, Marzo-Abril 2006, p.6-17.

M. Björkander. *Desarrollo basado en modelos y UML 2.0: ¿el fin de la programación tal como la conocemos?*. Novatica, Nº 168, Marzo-Abril 2006, p.21-24.

M. Fowler. *Domain Specific Language*. (http://www.martinfowler.com/bliki/DomainSpecificLanguage.html)

M. Kontio. *Architectural Manifesto: Choosing MDA tools*. IBM developerWorks (http://www-128.ibm.com/developerworks/wireless/library/wi-arch18.html)

M. Voelter, B. Kolb, S. Efftinge, A. Hasse. *From Front End To Code – MDSD in Practice*. June 15, 2006. (http://www.eclipse.org/articles/Article-FromFrontendToCode-MDSDInPractice/article.html)

N. Ahsan Tariq, N. Akhter. *Comparision of Model Driven Architecture (MDA) based tools (A Thesis document)*. Department of Computer and Systems Sciences, Royal Institute of Technology (KTH), 2005.

openArchitectureWare Fact Sheet. (http://www.eclipse.org/gmt/oaw/oAWFactSheet.pdf)

P. Harmon. *The OMG Model Driven Architecture and BPM*. Business Process Trends Newsletter, May 2004, Volume 2, No. 5.

R. Pohjonen, S. Kelly. *Domain-Specific Modeling: Improving productivity and time to market*. Dr. Dobb's Journal, August 2002.

S. Ambler. *Unified or Domain-Specific Modeling Languages?*. Software Development Agile Modeling Newsletter, March 2006. (http://www.metacase.com/news/AgileModelingMarch2006.html)

S. Cook. *Domain-Specific Modeling*. The Architecture Journal, 2006, Journal 9, p.10-17.

S. Cook. *Domain-Specific Modeling and Model Driven Architecture*. MDA Journal, January 2004.

SD Asia. *MDA Tools Comparison Matrix: A Guided Tool Adoption Roadmap for MDA Adopters*. (http://www.sda-asia.com/sda/specialcolumn/psecom,id,7,_language,Singapore,w,4,nodeid,1,xv_query,mda,xv_numresults,35,xv_sortvalue,0.html)

SD Asia. *MDA Tools Classification Approach: An Evaluation Template*. (http://architecture.sda-asia.com/sda/specialcolumn/psecom,id,2,nodeid,39,_language,Singapore.html)

Visual Modeling Forum. *Domain-Specific Modeling*. (http://www.visualmodeling.com/DSM.htm)

7 Annexos

7.1 Annex I - Planificació del projecte

A continuació s'annexa la planificació del projecte en format *Microsoft® Office Project Professional 2003*:

TFC-Planificació.mpp

7.2 Annex II – Exemple amb Microsoft DSL Tools

A continuació s'adjunta el codi corresponent a l'exemple ISA desenvolupat amb l'eina *Microsoft DSL Tools*:

UOC-ISA-DSLTools.zip

7.3 Annex III – Exemple amb Eclipse EMF + GMF + oAW

A continuació s'adjunta el codi corresponent a l'exemple ISA desenvolupat amb les eines *Eclipse EMF + GMF + oAW*:

UOC-ISA-Eclipse.ZIP

7.4 Annex IV – Exemple amb Borland Together 2006

A continuació s'adjunta el codi corresponent a l'exemple ISA desenvolupat amb l'eina *Borland Together 2006*:

UOC-ISA-Together.ZIP