# PFC – Analitzador de models UML

## Memòria

**Autor: Aleix Cerecedo Escriu**
**Titulació: Enginyeria Informàtica**
**Consultor: Javier Ferró Garcia**
**Data: 12 juny 2007**

Aquest projecte descriu la construcció d'una aplicacio encarregada de realitzar l'anàlisis d'un model UML. Està encabit dins el marc d'un aplicatiu de gestió de models en un repositori centralitzat de la àrea de Tècniques Avançades d'Enginyeria de Programari de la carrera d'Enginyeria Informàtica de la UOC.

Aquesta aplicació preten ser un punt de partida per a futures adaptacions i/o ampliacions per conseguir finalment un validador complet que sigui útil per als dissenyadors de sistemes i/o components de software. És per això que per a la seva construcció s'han pres desicions que no han buscat tant la optimització del aplicatiu com la seva organització, buscant poder encavir futures ampliacions i/o modificacions previsibles.

Per a realitzar un anàlisis d'un model UML s'ha partit de la execució de conjunts de mètriques que s'especialitzen en el càlcul d'un paràmetre relacionat amb la qualitat del model, i de la implementació XMI per a la persistència de models UML. En aquesta versió d'aquesta aplicació no s'interpreten els resultats, deixant aquesta qüestió per a un possible futur manteniment.

Aquest document reflecteix la evolució del desenvolupament d'aquest programari intentant detallar totes les qüestions relacionades que han anat apareixent durant el procés de la seva construccio.

# Índex

# 1. Memòria

## 1.1. Introducció

### 1.1.1. Justificació del PFC i context en el qual es desenvolupa

Actualment el nombre de projectes informatics i en concret de software que iniciem a l'any és enorme, i tota eina que faciliti qualsevol de les tasques en el procés de construcció o manteniment que tot aplicatiu comporta no només serà benvinguda, sinó que generalment comportarà una reducció significativa dels marges d'error sempre existents.

En concret, les eines que facilitin la edició, manteniment i coherència dels models UML amb les seves implementacions aportaran robustesa i qualitat als productes desenvolupats, així com també aportaran, en un alt grau de certesa, un estalvi considerable de recursos en planificació de projectes, en temps d'anàlisis dels components, i en general en la formació de nous analistes i programadors que hàgin de mantenir un component de software ben dissenyat.

Tot i que fa temps que s'utilitzen els diversos diagrames UML, encara falta molt per a poder dir que tota peça de software té diagrames UML que la descriguin. Els motius són diversos, i no és objectiu d'aquest projecte entrar en aquesta qüestió. Però si és important desde el punt de vista d'aquest projecte un dels problemes que sovint apareixen: el formalisme dels models.

De fet, molts analístes defineixen diagrames UML i l'únic objectiu que pretenen és que el model sigui entenible i pugui descriure millor que amb les paraules un disseny. Però molt poques vegades aquests analistes validen el formalisme dels seus models, possibilitant la introducció d'errors assumibles -ja que el diagrama, tot i que mal escrit, s'enten-, i per tant dificultant una integració de components de software amb, per exemple, altres departaments o altres empreses.

Aquest projecte preten ser un punt de partida per una peça de validació i anàlisis d'un model UML, encabit dins el marc d'un aplicatiu de gestió de models en un repositori centralitzat, que aporti un valor afegit als models analitzats. No preten, en cap cas, implementar tota validacio i/o anàlisis possible extraible dels diferents tipus de diagrames UML: la idea que persegueix és arrancar una eina que requerirà futures ampliacions de les seves funcionalitats.

En qualsevol cas el plantejament d'aquesta peça seguirà els criteris d'independència necessaris en tot component de software per evitar acoblaments i requisits que puguin interferir en la qualitat resultant. És per aquest motiu que la peça resultant d'aquest projecte s'ha de conceptualitzar com un component aïllat que intentarà assolir els objectius marcats en aquest plà de projecte.

### 1.1.2. Objectius del PFC

Com s'ha entrevist en la introducció d'aquest document, el projecte intentarà definir un marc intern de desenvolupament per a qualsevol extracció i anàlisis d'informació de qualsevol diagrama UML definit en qualsevol format de fitxer.

Degut a que aquesta arquitectura és molt genèrica, i engloba moltes funcionalitats, aquest projecte intentarà

realitzar una implementació concreta per a un format de diagrames UML concret, un o pocs tipus de diagrames UML, i pocs tipus d'extracció i anàlisis d'informació.

En el moment de definir la planificació del projecte no s'ha realitzat un estudi exhaustiu de la materia a tractar, de forma que aquesta haurà de tenir molt en compte la part de recerca d'informació relacionada amb la extracció i anàlisis d'informació dels diagrames UML (evidentment de la documentació trobada es definirà la especificació funcional del component), però es partirà de la suposició que l'estudi serà incomplet i que en un temps serà obsolet. Així el projecte s'aproparà millor als seus principals objectius: definir un marc extensible i adaptable a un gran nombre d'ampliacions.

S'espera que al final de la realització d'aquest projecte es pugui realitzar algun tipus d'anàlisis sobre algun/s model/s UML d'exemple, i que l'usuari pugui veure la informació resultant. Tot i així cal remarcar que no és objectiu d'aquest projecte definir una capa gràfica de presentació dels resultats, ja que implicaria un temps de desenvolupament que podria restar temps, i per tant qualitat, als verdaderament objectius principals d'aquest projecte.

Finalment cal comentar que aquest projecte parteix d'uns requeriments de projecte imposats per la fisiologia del projecte, a saber: és un projecte final de carrera universitaria amb uns terminis i fites definides previament per la facultat, de forma que costituiran l'eix principal de la planificació.

### 1.1.3. Enfocament i mètode seguit

Degut als terminis definits per la facultat en la avaluació continuada de la realització d'aquest projecte s'ha decidit realitzar un desenvolupament tradicional en cascada, de forma que cada part del procés de desenvolupament de la aplicació es pogués encabir en una fase (interval de temps entre dos entregues de la avaluació continuada).

En cap cas podem considerar que degut al redisseny induit per les conclusions extretes en la recerca deixada per a la fase d'implementació (productes existents en el mercat per a realitzar la lectura d'un fitxer XMI) ha acabat sent un desenvolupament en espiral o iteratiu, ja que l'únic que s'ha fet és tornar a començar el procés seqüencial de la cascada.

### 1.1.4. Planificació del projecte

| | Duració | Detall | Comentari |
|---|---|---|---|
| **FASE 1** | | | |
| Recerca mètriques | 1 setmana | 19 - 25 març | |
| Recerca anàlisis semàntica | 1 setmana | 26/març - 1/abril | |
| Generació documentació recerca | 0 setmanes | | Es farà mentre es realitza la recerca. |
| Disseny aplicació | 1 setmana | 2 - 8 abril | |
| | | | **Entrega PAC2 16 abril** |
| **FASE 2** | | | |
| Implementació | 5 setmanes | 16 abril - 20 maig | S'utilizarà junit per a realitzar la implementació, de |
| Tests | 0 setmanes | | forma que cada unitat haurà de tenir el seu test. |
| | | | **Entrega PAC3 21 maig** |
| **FASE 3** | | | |
| Compilació documentació / memòria | 1 setmana | 21 - 27 maig | |
| Analisis de resultats obtinguts | 1 setmana | 28 maig - 3 juny | |
| Realització defensa | 1 setmana | 4 - 10 juny | |
| | | | **Entrega FINAL 18 juny** |

Com es pot comprobar per la planificació, tant abans de la entrega del 16 d'abril, com abans de la entrega del 18 de juny, queda una setmana lliure. Això s'ha fet així ja que sempre acaben sorgint complicacions d'última hora que afecten als plaços d'entrega i aquestes dues setmanes serviran per a evitar contratemps.

En concret, la fase 1 només contempla dues setmanes per a la recerca. Aquest és un risc, ja que en un

projecte que intenta permetre qualsevol extensió és molt important tenir una bona visió global, i per tant potser els plaços resulten massa curts.

Un altre risc clarament visible en aquesta planificació és la poca especificació de la fase 2 -implementació-. Això és degut a que la recerca i el disseny marcarà clarament quin tipus de feina caldrà realitzar en aquesta fase (de totes formes queda clar que la fase d'implementació implica la fase de test degut a l'ús de la filosofia junit: escriure primer el test i després la implementació). També cal comentar que es parteix de la premisa que en la fase 1 -especificació i disseny- s'intentarà marcar objectius resolubles amb els terminis marcats, deixant constància de tot alló que pugui ser deixat fòra del disseny per qüestions de falta de temps.

Finalment cal comentar que tot i haver-hi una tasca en la fase 3 anomenada “Compilació documentació / memòria”, es parteix de la base que a mida que es vagin realitzant les diferents fases s'anirà creant la memòria. Aquesta fase és per a depurar, completar, i fer entenible el text resultant, i és per això que la seva durada és tant curta.

També cal dir que després de la fase 2, analitzar els resultats obtinguts pot ser més o menys trivial, i potser és més útil intercanviar els plaços per a la primera i segona tasca de la fase 3. No s'ha adjuntat un diagrama de dependències entre les diferents tasques ja que aquest plà és senzill, poc detallat, i a més totes les tasques són dependents amb la anterior excepte les dues primeres tasques de la fase 3, que es poden realitzar en paral.lel.

### 1.1.5. Productes obtinguts

Tot i que en el apartat “Sel.leccio programari / tecnologíes”, al final de l'apartat “1.3.2. Redisseny Aplicació”, detalla les tecnologíes escollides per a poder implementar el programari, durant la fase de recerca s'han probat altres tecnologíes per intentar aprofitar implementacions existents per a reduir el cost de desenvolupament del programari, a saber: JMI, com a especificació d'un metamodel; Eclipse Modeling Framework (EMF), per a la implementació dels metamodels XMI i la lectura dels fitxers XMI; i NetBeans i MagicDraw, per a la lectura de fitxers XMI. Les referències a aquestes tecnologíes estan en el apartat “Bibliografia”.

### 1.1.6. Breu descripció dels altres capítols de la memòria

Els següents capítols de la memòria reflexen la evolució del desenvolupament de la aplicació. Així doncs, hi ha dos grans apartats: “Recerca i Disseny”, corresponent a la fase 1; i “Implementació”, corresponent a la fase 2.

En el apartat “Recerca i Disseny” es mostra la documentació entregada al final de la primera fase, coincidint amb la segona entrega de la avaluació continuada marcada per l'itinerari de la facultat. És on es poden trobar la majoria dels conceptes i idees dels quals parteix l'aplicatiu, conjuntament amb el primer disseny realitzat.

En el següent apartat, “Implementació”, es poden trobar les causes del redisseny, el nou redisseny, i la explicació detallada de les diferents peces que integren la solució entregada, així com el seu funcionament. Tot i que l'últim punt d'aquest apartat es mostren les primeres valoracions respecte els resultats obtinguts a partir del nou redisseny, en el apartat “1.5. Conclusions” es mostra una valoració global del projecte i dels objectius marcats en la planificació inicial.

## 1.2. Recerca i disseny

### 1.2.1. Anàlisis d'un model: mètriques i semàntica

Existeix tota una disciplina al voltant de l'anàlisis de software que intenta ponderar la qualitat dels aplicatius. Evidentment els models UML formen part d'un aplicatiu, de forma que també han aparegut diferents estudis sobre l'avaluació de la qualitat d'una aplicació a partir de l'anàlisis del seu model.

Els estudis actuals al voltant de l'anàlisis de models utilitzen bàsicament mètriques extretes d'un model (per exemple la profunditat màxima d'una classe dins el model) per a extreure conclusions sobre el component modelitzat (per exemple alt acoblament entre les peces del component).

D'aquests estudis destaquen alguns que utilitzen grups de mètriques (sets) per a evaluar un model. Alguns exemples són els sets de Genero, Kim and Boldyreff o Chidamber and Kemerer.

En qualsevol cas, però, aquests conjunts de mètriques no són fiables al 100%. Els resultats que aporten sobre un model molts cops requereixen una interpretació humana o una comparativa de resultats entre diferents versions de l'aplicatiu analitzat per a poder evaluar correctament la qualitat del software estudiat.

Així doncs, en aquest projecte es tindrà en compte la gran variabilitat de conjunts de mètriques que es poden utilitzar per a validar un model, i sobretot es tindrà en compte el fet que s'espera, degut a la data d'alguns d'aquests estudis, nous estudis més robustos i més complets en el decurs dels pròxims anys.

També cal recordar que el principal objectiu d'aquest projecte és definir una arquitectura extensible que permeti realitzar anàlisis de models, i per tant no serà objectiu d'aquest projecte entrar en detall en aquests estudis. De fet s'utilitzarà el set de Chidamber and Kemerer com a referència per a realitzar la implementació de la arquitectura.

#### Chidamber and Kemerer: reusabilitat del codi

Chidamber and Kemerer van definir un conjunt de mètriques que permeten analitzar la qualitat d'un component o aplicatiu. En aquest projecte ens centrarem en la interpretació d'aquestes mètriques per a evaluar la reusabilitat del codi modelitzat.

**Mètodes ponderats per classe (MPC)**
Aquesta mètrica es centra en el nombre de mètodes de una classe. La idea és que una classe amb més mètodes és més difícil de reutilitzar que una classe amb pocs mètodes. Aquesta mètrica no té en compte la complexitat de cada mètode i es pot extreure d'un diagrama de classes (es podria utilitzar la mètrica McCabe's Cyclomatic Complexity per a ponderar la complexitat d'un mètode i poder comparar millor dues classes entre si, però requeriria diagrames de seqüència, activitat o d'estats per a poder calcular-la).

**Profunditat de l'arbre d'herències (PAH)**
Aquesta mètrica mesura la màxima profunditat d'una classe del seu arbre d'herències, és a dir, la màxima distància amb la arrel de l'arbre. La idea és que una classe molt profunda heredarà més mètodes i per tant serà més difícil de reusar. Es calcula a partir de tots els diagrames de classe que tingui un model.

**Nombre de fills (NDF)**
Un gran nombre de fills d'una classe podria indicar una incorrecta abstracció de la classe pare, i una classe amb molts fills implica un augment del test que s'ha de fer. També cal dir que les classes en profunditats baixes de l'arbre d'herències (a la part alta) hauríen de tenir més fills que les situades a alta profunditat, fet que relaciona aquesta mètrica amb la mètrica PAH.

**Acoblament entre objectes de les classes (AOC)**
Mesura l'acoblament d'una classe contant el nombre de classes de les quals es considera que està

acoblada. Es considera que alts nivells d'acoblament dificulten i/o impedeixen la reutilització ja que trenca amb el disseny modular. Es calcula a partir d'un diagrama de classes sumant les relacions d'us que té la classe amb els atributs i els paràmetres dels seus mètodes (no considerarem les referències a objectes de la api de desenvolupament -per exemple la api de Java o .Net- ja que es considera que no tindrà impacte, tot i que es podria/s'hauria de considerar).

**Falta de cohesió dels mètodes (FCM)**
Es una mesura de la carència de solapament en el ús dels atributs d'una classe. Si els mètodes utilitzen subconjunts separats d'atributs de la classe es podria fer la hipòtesis de que els mètodes no estan correctament agrupats y que probablement la classe es podria dividir en varies. Classes amb un alt FCM dificulten la reutilització ja que augmenten la dificultat de comprensió. Aquesta mètrica s'hauria de calcular amb diagrames de seqüència (ja que si es calculés amb els mètodes d'un diagrama de classe no sabríem si els mètodes utilitzen o no els atributs).

**Reacció per a una classe (RPC)**
És la mesura del nombre de mètodes que poden ser potencialment invocats per un objecte d'una classe. Es parteix de la idea de que un mètode que realitza moltes crides es més complexe, requereix més test, i és més difícil d'entendre. Per calcular aquesta mètrica cal el codi o diagrames de comportament -per exemple de seqüència-, ja que és la implementació qui determina el nombre d'invocacions que realitzarà el mètode. Hi ha la variant d'aquesta mètrica que calcula el nombre de crides de forma recursiva, és a dir, que té en compte les crides que fan els mètodes cridats.

**Més mètriques**

A part d'altres conjunts de mètriques definits en altres estudis com el de Chidamber and Kemerer, també hi ha altres mètriques que es poden extreure aïlladament tant dels diagrames de classes com d'altres diagrames i que ens aporten informació de la complexitat, acoblament, i altres paràmetres de la qualitat d'un model.

Un dels objectius d'aquest projecte és la creació d'una arquitectura que permeti la execució de varis conjunts de mètriques, o el que és el mateix, la execució de diferents anàlisis sobre un o varis models. Així doncs, es definirà un set de mètriques general que englobarà algunes d'aquestes mètriques, per a poder-se executar alhora que el set de Chidamber and Kemerer. L'objectiu de la creació d'aquest conjunt queda detallada en el apartat posterior: “Anàlisi i disseny de la aplicació”.

**Anàlisis semàntic d'un model UML: OCL**

Es sabut que els models UML no són formals. Per aconseguir un model formal UML permet la definició de regles i restriccions formals a través del Object Constraint Language (OCL), un llenguatge d'especificació que utilitza lògica per especificar les propietats invariants de sistemes que es composen de conjunts i relacions entre conjunts.

A partir del llenguatge OCL sí que es podria realitzar una validació semàntica d'un model, ja que es podria realitzar una validació de les regles OCL per a detectar inconsistències o contradiccions entre propietats invariants.

De totes formes, no formarà part dels objectius d'aquest projecte crear un analitzador semàntic a partir de regles OCL. Simplement es tindrà en compte en la arquitectura que en un futur podrà existir un tipus d'anàlisis de regles OCL, i que per tant ha de poder ser executat.

### 1.2.2. XMI

Fins ara s'han vist els models en abstracte, però la aplicació haurà de rebre com a paràmetres d'entrada un conjunt de models, i evidentment aquests hauran de tenir algun tipus de format que els codifiqui. Per a aquesta qüestió cada implementació d'un editor visual UML pot tenir el seu propi format, però la OMG

defineix en un estàndard la codificació XMI com a sistema d'intercanvi de dades de models entre aplicacions.

XMI és un llenguatge basat en l'XML, que defineix unes etiquetes -tags XML- que permeten definir els tipus d'elements pròpis d'un model. A més a més, l'estàndard també permet extendre el llenguatge XMI per facilitar la vida als generadors de models.

La realitat, però, es que aquesta extensibilitat ha dut a que cada programa de modelatge UML generi el seu pròpi codi XMI, la majoria de vegades inpedint la compatibilitat entre aquests programes i destruint l'objectiu principal d'XMI: permetre l'intercanvi de models entre diferents aplicacions.

De totes formes és de suposar que en el decurs dels següents anys apareguin eines més robustes de modelatge UML que es consolidin en el mercat (com Rational), i que facilitin la integració XMI entre ells. Mentrestant aquest projecte ha de tenir en compte que la persistència XMI dels models podria tenir una codificació diferent en funció de la eina que generés el model, de forma que la arquitectura tindrà en compte que podríen coexistir diferents implementacions XMI.

**Eines de modelatge**

Per a aquest projecte s'ha triat el editor visual UML d'Eclipse, ja que és gratuït i perque quan genera un model directament l'enmagatzema en el espai de treball en format XMI. Ara bé, aquesta decisió no ha tingut en compte problemes d'implementació que poguèssin sorgir en un futur, de forma que no és objectiu d'aquest projecte lligar-se a aquesta implementació d'XMI (és a dir, sempre es pot canviar l'editor visual i la implementació XMI que generi). Així doncs queda pendent per a la fase d'implementació acabar d'escollir la tecnologia sobre la qual es construirà la aplicació.

### 1.2.3. Anàlisi i disseny de la aplicació

La idea principal que es persegueix és que la aplicació executi alguns conjunts de mètriques per a un tipus de diagrama UML concret, de forma que es mostri la extensibilitat per a altres conjunts de mètriques i d'altres implementacions d'un model (per exemple diferents implementacions XMI o altres formats si es dongués el cas).

Si la fase d'implementació ho permet es podria fer la implementació per a dos tipus de diagrames UML (per exemple, a part del diagrama de classes es podria fer la implementació del diagrama de seqüència), però no entra dins els objectius principals fer dues implementacions. Aquest fet podria provocar un problema amb la implementació del set de Chidamber and Kemerer ja que algunes de les seves mètriques requereixen algun altre tipus de diagrama diferent al de classes.

Els conjunts de mètriques que s'han d'implementar han de mostrar com es poden combinar, ja que per qüestions de rendiment s'espera que la aplicació, en cas d'haver més de N conjunts de mètriques a executar, no faci M > N lectures del/s fitxer/s XMI. Així, si hi ha un conjunt general que calcula alguna mètrica que algun set necessita, aquest últim hauria de poder recuperar el resultat de la mètrica evitant calcular-lo. També és important que la arquitectura, mentre faci la única lectura d'un fitxer XMI, vagi avisant als conjunts de mètriques dels elements llegits perque totes les mètriques a executar ho vagin fent mentre es fa la lectura. D'aquesta forma garantitzaríem el rendiment òptim de la arquitectura.

També s'intentarà abstraure el model de la aplicació perque no depengui d'una implementació concreta dels diagrames UML, ja que també podria ser que l'XMI acabés sent obsolet i els conjunts de mètriques no!. De totes formes, la aplicació sí que dependrà de la especificació UML estàndard, assumint que si aquesta tecnologia s'obsoleta la aplicació també ho serà.

La arquitectura de la aplicació, doncs, haurà de satisfer els següents requisits:

- Ha de poder executar diferents conjunts de mètriques per a un conjunt de models d'entrada.
- Els paràmetres d'entrada -models- podran ser fitxers XMI (inicialment només seran XMI).
- Un conjunt de mètriques declararà quins diagrames requereix per a executar-se, de forma que

l'aplicatiu haurà de llençar una excepció en cas de no disposar els diagrames necessaris per a poder executar un cert conjunt de mètriques.
- La arquitectura hauria de poder aguantar diferents implementacions d'XMI, donat el cas.

### Diagrama de classes

Com que després del diagrama de classes s'adjuntarà una breu explicació de cada concepte que hi surti, no es detallarà el glossari de termes relacionats amb la aplicació. Tampoc s'adjuntarà el diagrama de casos d'ús per la simplicitat que té (de fet, l'agent usuari utilitza el cas d'ús anàlisi).

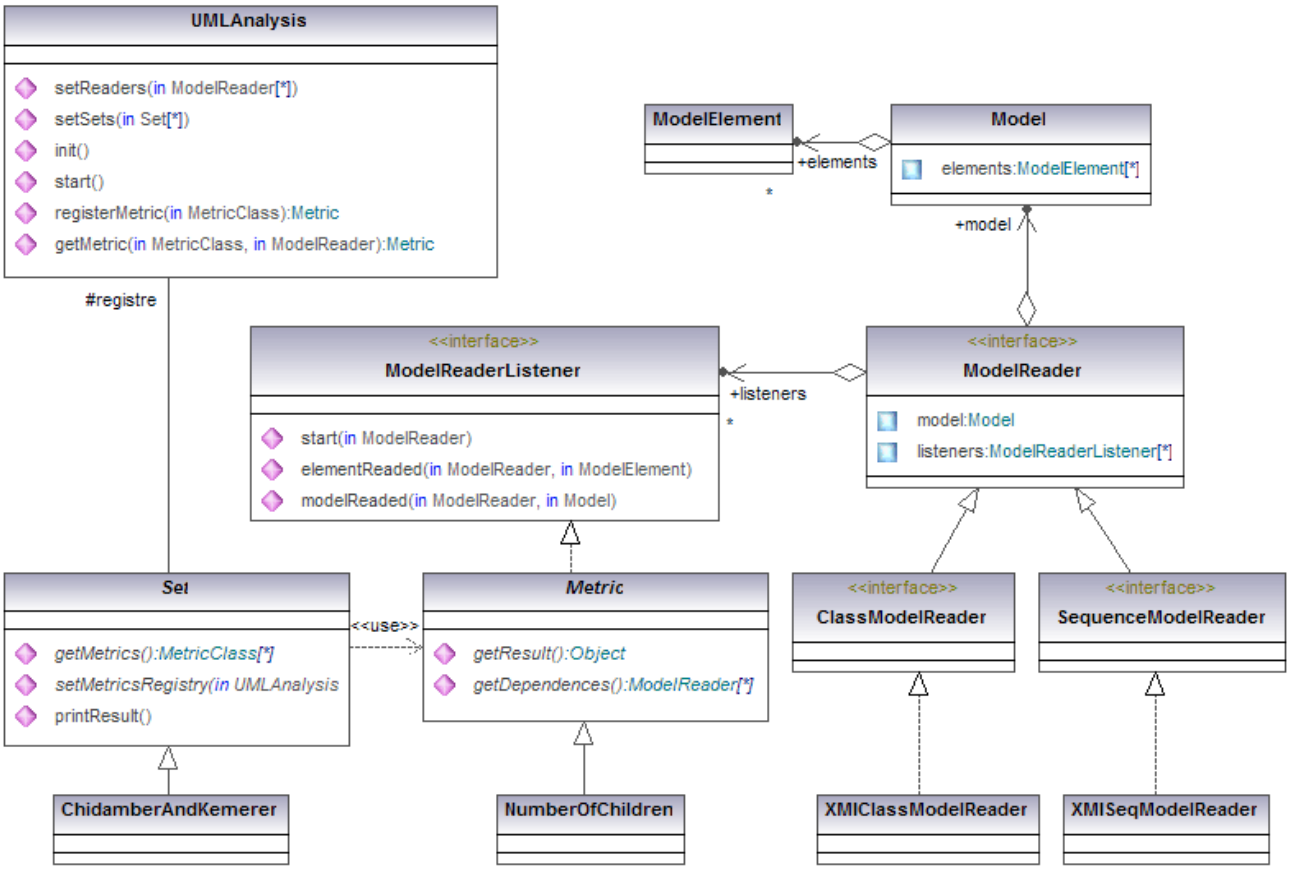



*Fig. 1, Diagrama de classes*

S'enten per Model un conjunt d'un o més diagrames UML que representin el model d'una aplicació. Això és degut a que hi ha diferents implementacions d'XMI que enmagatzemen en un o varis fitxers XMI els diferents diagrames que tingui el model de la aplicació (per exemple Eclipse, en la versió 0.2.5 del plugin VisualUML, genera un fitxer XMI per cada diagrama, i en canvi Altova Umodel 2007 genera un fitxer XMI per a tots els diagrames del model de la aplicació). Així doncs, un diagrama UML correspondria a un model, i un conjunt de diagrames podríen ser un sól model o varis models (un model per diagrama).

El disseny defineix una interfície *ModelReader* per diagrama. En cas que no es vulgui fer una implementació separada per diagrama només cal implementar totes les interfícies filles de *ModelReader* en una sóla implementació. Ja s'encarregarà la implementació de la aplicació d'entendre el *ModelReader* com si fòs molts *ModelReader*'s (veure explicacions de la classe *UMLAnalysis* més endavant).

També es pot observar com s'intenta separar la lògica de les mètriques de la codificació d'un Model a través de la interfície *ModelReaderListener*. Tota mètrica (classe abstracta *Metric*) implementa aquesta interfície ja que li permetrà anar rebent els elements llegits del model (classe *ModelElement*) i per tant anar fent els càlculs mentre es fa una única lectura del model. Aquest sistema de notificació està basat en el patró escoltador de canvis de Java anomenat *PropertyChangeListener* (variant del patró *Publisher-Subscriber*).

Pel costat de la lògica de les mètriques s'ha definit el conjunt de mètriques (classe abstracta *Set*). Aquesta classe agrupa conjunts de mètriques, permetent tenir un conjunt de qualsevol nombre positiu de mètriques; i defineix la signatura *printResult* per qué les classes filles, o el que és el mateix, les classes que realment simbolitzen un conjunt/estudi, implementin la forma de mostrar el resultat. Aquesta decisió s'ha pres així degut a que es té en compte que només l'estudi Chidamber and Kemerer sap quina interpretació fer dels resultats de les mètriques que observa.

La classe més important és *UMLAnalysis*, encarregada de gestionar la lògica del test del model, és a dir, realitza la execució dels diferents *Set*'s relacionant-los amb els *ModelReader*'s que s'hàgin definit a executar. És aquesta classe qui s'ha d'encarregar de gestionar els diferents tipus de *ModelReader* que hi pugui haver i per tant als que una mètrica s'hi pugui registrar (definit a través del mètode *Metric.getDependences()*).

*UMLAnalysis* té la particularitat de ser el registre de les mètriques de l'aplicatiu. Aquesta decisió apareix de la necessitat de reaprofitar, per qüestions de rendiment, la lògica de les mètriques per a diferents conjunts de mètriques. Així, si hi ha dos *Set*'s que necessiten llegir la mateixa mètrica d'un mateix model, no cal que executin dos cops la mètrica (un per cada *Set*), sinó que al registrar la mètrica la classe *UMLAnalysis* s'encarrega de crear només un cop la mètrica a executar i de fer-la disponible a tot *Set* que la vulgui llegir. Així, la factory de les mètriques no és la classe *Set*, com es podria suposar, sinó que és el registre (classe *UMLAnalysis*). Una classe de tipus *Set* només haurà de declarar quines mètriques necessita a través de la signatura *getMetrics():metricClass[*]*, i *UMLAnalysis* utilitzarà aquesta llista de tipus per a crear les instàncies.

Finalment cal comentar que el tipus *MetricClass* (es pot observar en la signatura *Set.getMetrics* i en la classe *UMLAnalysis*) és qualsevol tipus de dada que representi un tipus *Metric*. Es deixa per a la fase d'implementació escollir si s'utilitza una cadena de caràcters, un identificador numèric, o el tipus *Class* de Java; al igual que el paràmetre *ModelReader* de totes les signatures de la classe *ModelReaderListener*, que bàsicament busca informar al listener de quin és el model que llegeix un element i és possible que canvii en la implementació.

**Diagrama de seqüència**

El diagrama de la figura 2 representa l'exemple de la execució d'un únic *Set* (*ChidamberAndKemerer*) amb una única mètrica (*NumberOfChildren*). Això s'ha fet per a simplificar la comprensió del diagrama, però cal tenir en compte que en la fase *init* de l'anàlisis es provocaria el registre de totes les mètriques de tots els *Set*'s; en la fase *start* s'executaríen tots *ModelReader*'s en ordre i seqüencialment; i en la fase *printResult* s'executaria aquesta funció per cada *Set* de l'anàlisis.

És important destacar que és la classe de tipus *Set* qui s'encarrega de mostrar el resultat. Això implica que si tenim dues façanes gràfiques per a mostrar a l'usuari hi haurà dues implementacions de *ChidamberAndKemerer*! Això es sol.luciona creant un paquet per cada façana gràfica, o senzillament creant una implementació del *Set* en que el nom indiqui el tipus de presentació dels resultats que en faci (per exemple crear una classe anomenada *ChidamberAndKemererApplet*). Aquesta última sol.lució permet crear una classe abstracta anomenada *ChidamberAndKemerer* que no implementi la funció *printResult* i ho deixi per a les seves classes filles.

Finalment cal destacar la síncronia entre crides a les signatures de la classe *UMLAnalysis*:

- El mètode *printResult* de un *Set* només té sentit executar-lo després que s'hagi llençat l'*start*, perque sinó les mètriques no tindran els resultats correctes, només els inicials.
- Si no es crida a *init* no es registraran cap de les mètriques i per tant no hi haurà ningú escoltant la lectura del/s model/s. Per tant s'ha de cridar abans de fer l'*start*.
- Els mètodes *setReaders* i *setSets* han de ser cridats abans de realitzar l'*init* degut a que per a fer el registre calen els *Readers* (una mètrica depen d'un *Reader*).

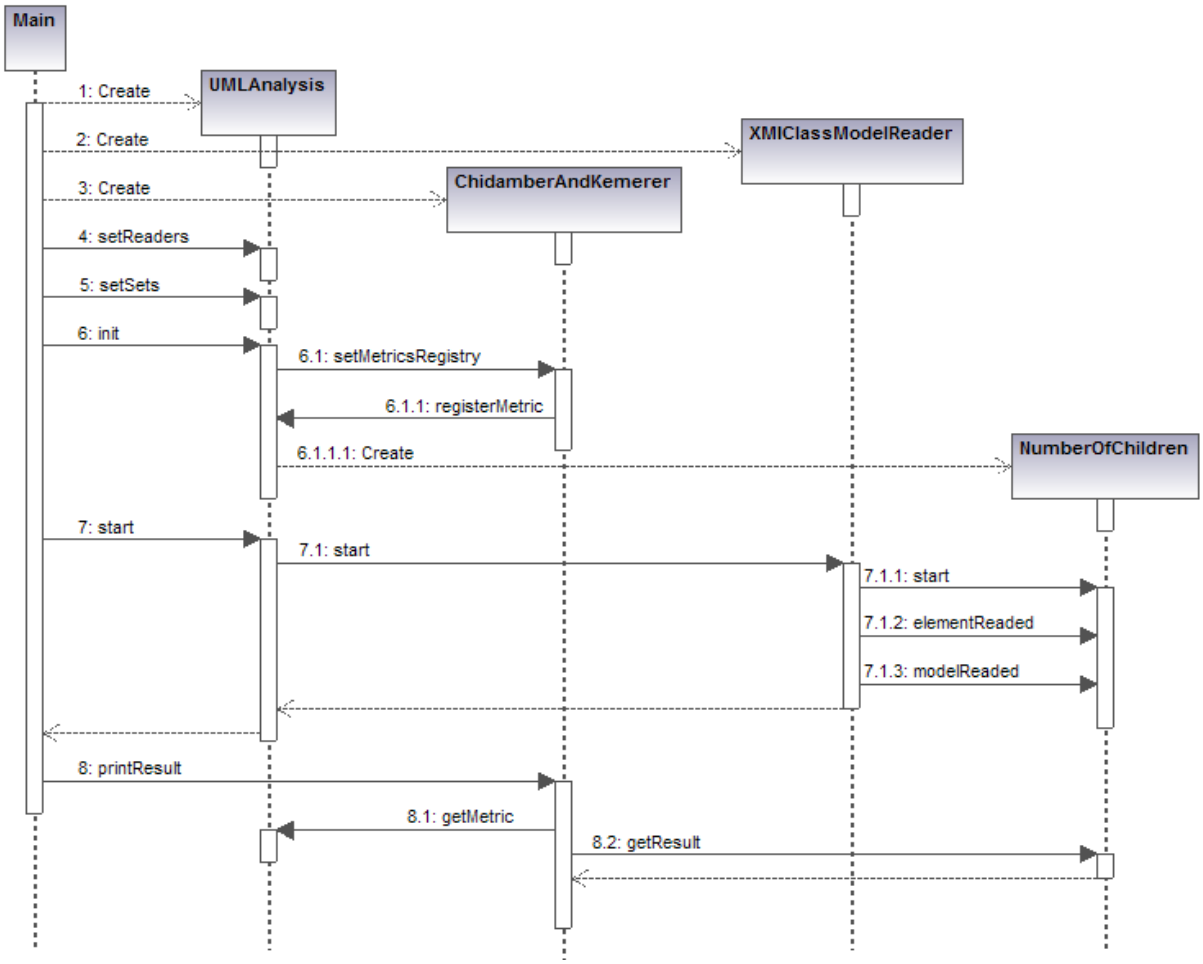



*Fig. 2, Diagrama de seqüència*

## 1.3. Implementació

### 1.3.1. Anàlisis del disseny inicial: Motius del redisseny

La fase de disseny va deixar la tecnologia generadora de fitxers XMI per a la fase d'implementació. Aquesta decisió es va pendre ja que es va separar la implementació de la persistència del model respecte l'implementació de l'analitzador de models, així que es va creure que no era necessari escollir ni la versió XMI ni el producte a utilitzar per a generar els fitxers de model ja que es pretenia que la aplicació fós capaç de ser independent d'aquestes tecnologíes.

Però durant la sel.lecció de la tecnologia XMI en aquesta fase s'ha vist que els diferents productes del mercat útils per a llegir fitxers XMI (NetBeans, Plugin UML d'Eclipse, Magic Draw) estan molt lligats al metamodel (o model dels models UML) que hagi definit el proveïdor, és a dir, a la implementació de l'equivalent de les classes Model i ModelElement definides en el primer disseny d'aquesta aplicació.

Com que la implementació de les mètriques depen totalment del metamodel ja que requereixen fer-hi consultes per a realitzar els seus càlculs, finalment s'ha optat per a redissenyar la aplicació, buscant independitzar les mètriques dels diferents tipus de lectors XMI que es poguèssin utilitzar en un futur.

Un altre motiu per a redissenyar la aplicació ha sigut el fet de que si volem separar les mètriques de la persistència del model aquestes no poden conèixer els detalls de la lectura, ja que el format XMI conté constants referències entre els diferents elements que conté i fins que no s'hagi llegit tot el model la mètrica no pot començar a realitzar les consultes ja que es podria trobar amb informació inclompleta o incoherent (referències a elements no llegits, etc).

Aquest últim fet ha provocat la decisió de canviar la filosofia de la aplicació, ja que abans es centrava exclusivament en que les mètriques escoltaven els elements que anaven sent llegits pel lector i ara s'ha decidit passar a una càrrega previa del model, independenditzant les mètriques dels diferents lectors de model que hi pugui haver disponibles.

Finalment cal indicar l'últim motiu que ha provocat aquesta greu decisió. Les mètriques realitzen un tipus de càlculs on gran part del seu cost computacional resideix en la recerca dels valors un cop ja s'ha carregat el model, de forma que la precàrrega dels valors que requereix una mètrica s'hauria convertit únicament en l'enmagatzematge de les referències dels elements a analitzar les seves propietats, acció que també pot realitzar el nostre metamodel.

### 1.3.2. Redisseny aplicació

Ja en la fase d'implementació, després d'haver intentat utilitzar alguna api existent de lectura de fitxers XMI sense èxit, s'ha optat per crear un lector de fitxers XMI pròpi, utilitzant els fitxers XMI que genera la aplicació Altova Umodel 2007, que són de la versió 2.1 (tot i que aquest producte també és compatible amb XMI 2.0).

Aquesta decisió ha sorgit únicament del fet que aquesta aplicació facilitava la creació dels diferents tipus de diagrames UML d'un model generant un sól fitxer XMI. Altres aplicacions (com per exemple el plugin editor visual UML d'Eclipse), per no ser compatibles amb tots els diagrames, perque les api's sobre les que treballen estan en procés de desenvolupament, o senzillament perque són de propietat i per tant cal pagar per disposar d'elles, han quedat descartades; tot i que és molt possible que un futur manteniment d'aquesta aplicació dugui a la utilització d'aquestes api's.

També cal dir que abans de començar el redisseny s'ha tingut en compte que tornar a implementar el metamodel (ja que els diferents proveïdors de llibreries per a llegir fitxers XMI ofereixen algun tipus de metamodel), tot i ser necessari per a poder separar les nostres mètriques del proveïdor del lector XMI, suposa una feina gens menyspreable i potser en va si es troba alguna api de Java que pugui oferir-ho gratuïtament (per exemple alguna implementació de JMI per a UML).

Així doncs s'ha decidit que la implementació del metamodel, per aquesta primera versió de l'analitzador UML, es reduirà al subconjunt de classes necessaries per a executar les mètriques que s'ha decidit implementar, i s'anirà construint a mida que les mètriques requereixin elements del metamodel. És per aquest motiu que en el diagrama de classes no apareixen les classes del metamodel, ja que en el moment de definir el diagrama no estaven clars quins elements es requeriríen.

### Diagrama de classes

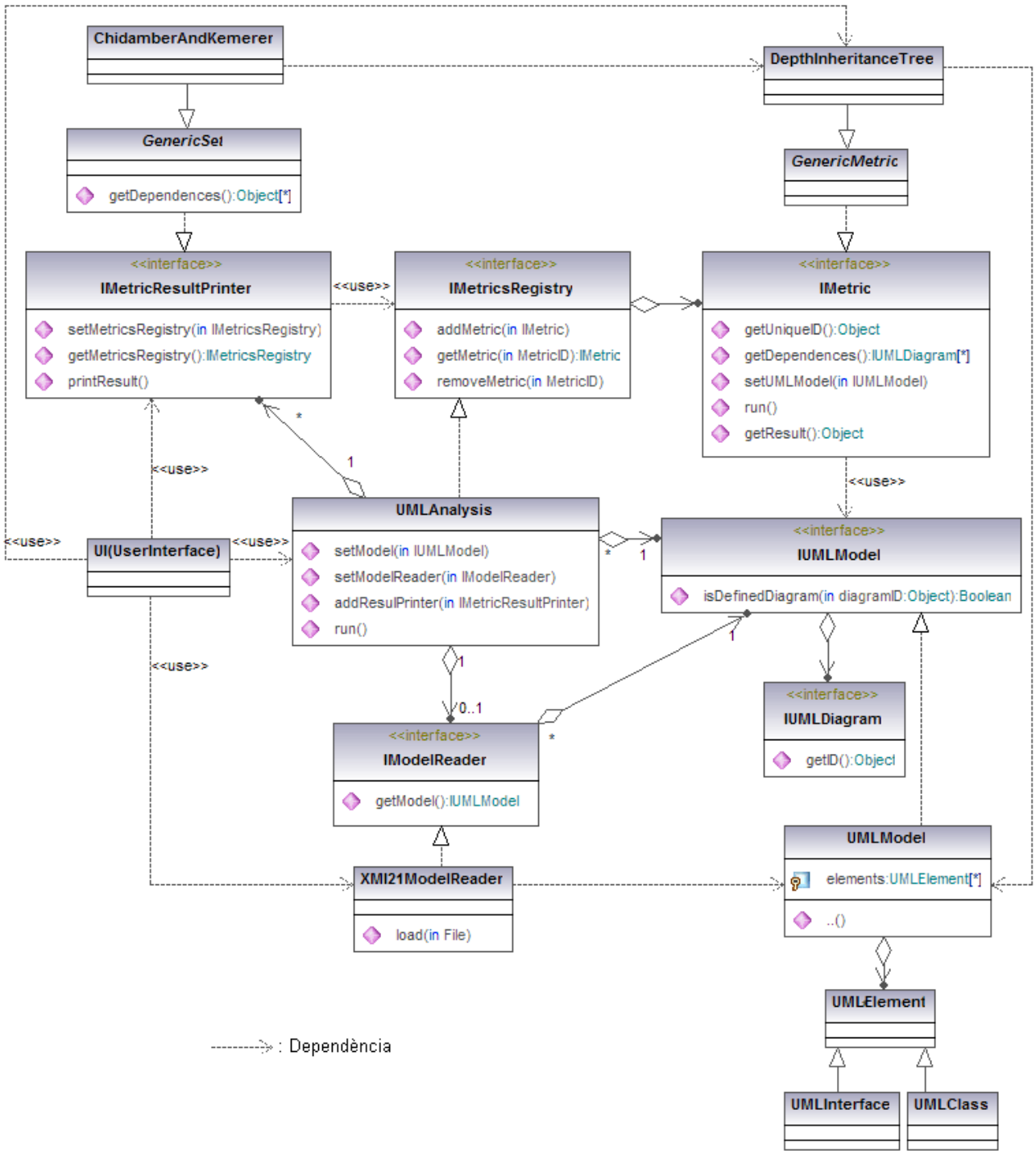



*Fig. 3, Diagrama de classes*

El primer canvi substancial respecte l'anterior disseny és la separació dels conjunts de mètriques -classe Set- de les classses directament relacionades amb l'anàlisi d'un model. La forma de fer-ho és interpretant un Set com un pintador de resultats, de forma que el component principal del analitzador passa a ser la mètrica i no el Set com en el primer disseny.

El segon punt a destacar del nou disseny és la separació del lector de model de l'analitzador. Ara el lector del model és una de les dues formes que l'analitzador pot treballar: o se li passa directament el model o se li passa una referència al lector perque l'analitzador pugui demanar-li el model abans de començar l'anàlisis. Així aconseguim separar del tot la lectura del model del aplicatiu, facilitant un posterior manteniment del procés de lectura.

També cal comentar que la classe UMLAnalysis és l'aglutinador de totes les peces de l'aplicatiu, de forma que la seva feina únicament consisteix en relacionar el model i els pintadors de resultats amb les mètriques, i executar les mètriques (és a dir, executar l'anàlisis).

UI és la classe que rep els paràmetres de la aplicació i configura l'anàlisis. Aquesta feina consisteix bàsicament en definir en el anàlisis (implementat per la classe UMLAnalysis) quines mètriques i quins pintadors de resultats s'han d'executar, juntament amb quin lector de model o model s'ha de tenir en compte. En el diagrama de classes no s'ha especificat la seva signatura -mètodes i propietats- ja que no és rellevant per a la aplicació. De fet, si s'utilitzés l'analitzador de forma integrada amb qualsevol altre aplicatiu, se suposa que aquesta classe deixarà d'existir perque algu altre configuri l'anàlisis. En el següent punt d'aquest document (“1.3.3 Detalls de la implementació”) es detalla el funcionament de la classe UIConsole, que equival a la classe UI).

El sentit de la existència de la interfície IMetricsRegistry és únicament per a crear una classe que centralitzi el registre de les mètriques en el anàlisis, de forma que sigui aquesta la qui controli que no es registrin dues mètriques iguals. De fet, però, aquesta peça era totalment prescindible, entre altres coses per l'acoblament que genera entre la classe UMLAnalysis i els pintadors de resultats, però la decisió s'ha pres tenint en compte que els pintadors de resultats podríen ser una peça externa de la aplicació, i per tant ens servirà d'adaptador de l'anàlisis en cas de futures externalitzacions dels pintadors de resultats (tot i que llavors caldria fer alguna modificació externalitzant el registre de l'anàlisis, o bé modificant la implementació de la classe UMLAnalysis).

Abans de parlar de la part més important de totes, el metamodel, cal destacar que en aquest diagrama només s'han mostrat una mètrica i un pintador de resultats (DepthInheritanceTree i ChidamberAndKemerer). Evidentment totes les altres mètriques com pintadors de resultats són germans d'aquestes classes respectivament.

També cal fer un comentari al respecte de la relació d'ús de la classe UI amb la classe IMetricResultPrinter, que no és del tot correcte: hauria de ser una relació d'ús amb la classe ChidamberAndKemerer. Aquesta relació només preten indicar que és la classe UI qui coneix la existència d'una mètrica concreta, i no UMLAnalysis.

Finalment cal parlar del metamodel. Aquest està definit per les interfícies IUMLModel i IUMLDiagram, bàsicament perque el codi de l'analitzador no depengui de la implementació del metamodel. De fet, hi ha una clara dependència entre les mètriques i el metamodel, així com una gran dependència entre el lector de model i el metamodel, i els pintadors de resultats i les mètriques. És a dir, modificar el metamodel possiblement impactarà en totes les peces de l'aplicatiu, menys a l'analitzador propiament.

Així doncs és important observar que l'impacte d'una modificació en el metamodel afectarà a la implementació de les mètriques (classe GenericMetric, DepthInheritanceTree, etc), i a la implementació del lector (classe XMI21ModelReader), però no a la resta de classes (evidentment, si es modifica la forma de retornar els resultats de la mètrica degut als canvis del metamodel també afectarà a les classes que pintin els resultats -classes GenericSet, ChidamberAndKemerer, etc-).

La interfície IUMLDiagram està pensada per a poder determinar les dependències d'una mètrica. En l'anterior disseny les dependències d'una mètrica s'enteníen com a dependències amb els ModelReader's, però com que amb el nou disseny separem el lector de l'analitzador, la mètrica no té per que saber de l'existència de lectors de model.

La decisió de declarar les dependències d'una mètrica dels diagrames que necessita per a realitzar els seus

càlculs apareix de la consideració que hi ha mètriques que no té sentit executar si no s'ha definit en el model cert/s diagrames (per exemple hi ha mètriques que no tenen sentit si no es defineix el diagrama de seqüència), i per tant només fent una verificació de dependències abans d'executar una mètrica ens pot estalviar el control de resultats incoherents.

Amb les dependències es pot observar que la lògica dels anàlisis (classe UMLAnalysis) està separada de les implementacions de les mètriques, del metamodel, dels possibles lectors de model, i dels pintadors de resultats.

Per aquesta primera versió de l'analitzador es té pensat construir un metamodel pròpi, amb un lector que l'ompli, i unes mètriques que llegeixin aquest metamodel. Les qüestions de manteniment que suposarà aquest disseny es comentaran posteriorment, després d'explicar la organització del codi entregat i com està plantejat.

**Diagrama de seqüència**

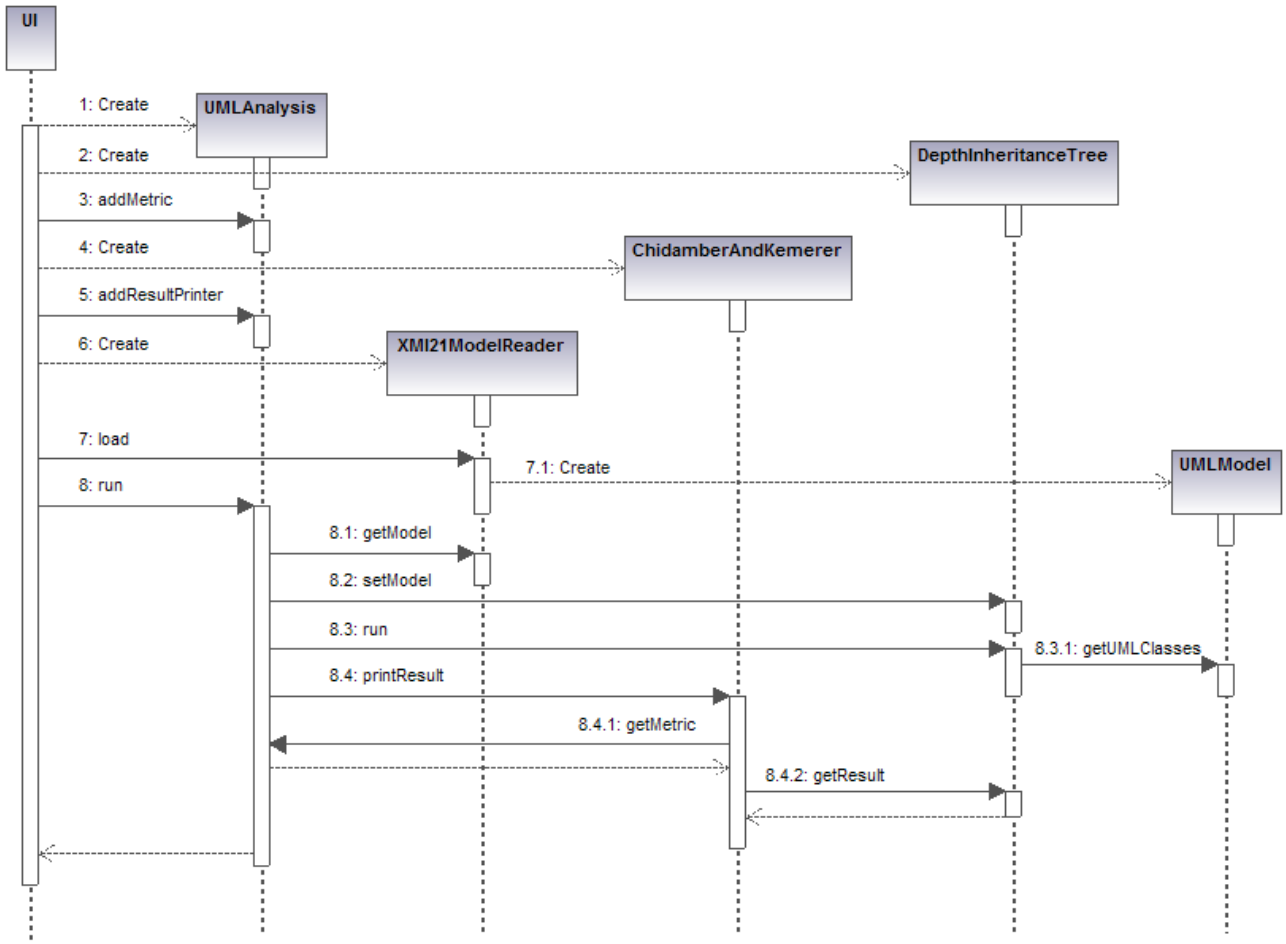



*Fig. 4, Diagrama de seqüència*

El diagrama de la figura 4 representa l'exemple de la execució d'un anàlisis amb una sóla mètrica (DepthInheritanceTree) i un únic pintador de resultats (ChidamberAndKemerer). No s'han declarat els noms de les instancies dels lifeline's ni es mostra el gate d'entrada que cridaria a la classe UI ja que s'ha considerat que no calia per a comprensió de la execució del disseny.

Aquesta execució és una simplificació, ja que no estan definides les crides en cas de produir-se errors, ni les iteracions per la execució de les mètriques i dels pintadors de resultats (quan UMLAnalysis crida al run de la

mètrica, aquesta crida hauria d'estar dins un bucle que fés executar totes les mètriques que estiguin definides). Tampoc es mostra la creació del model, donant per sentat que és una qüestió a realitzar per el lector de model en funció del metamodel. L'objectiu d'aquest diagrama és només aclarir els detalls dels principals passos de la execució de l'analitzador.

Segons el disseny, hi ha varies maneres de realitzar la execució, així que aquest és el diagrama d'una de les possibilitats, però la classe UMLAnalysis es pot configurar de diferents maneres, a saber:

- No cal passar-li els pintadors de resultats, desde la classe UI es podria executar els pintadors de resultats directament.
- No cal passar-li el lector de model, es podria crear i carregar el model desde la classe UI i passar-li a la classe UMLAnalysis el model directament.

En qualsevol cas, però, aquest diagrama mostra la execució del anàlisis amb menys càrrega computacional per l'usuari de UMLAnalysis -classe UI-.

**Sel.lecció de programari / tecnologíes**

Ja desde un primer moment aquest projecte estava pensat per a realitzar en Java, un llenguatge de desenvolupament gratuït i open source. El que quedava per triar era la versió XMI, i en cas de utilitzar alguna api per llegir fitxers XMI, el proveïdor i la versió corresponent.

Com que s'ha decidit implementar el lector de fitxers XMI, i no s'ha utilitzat cap api per la implementació del metamodel, els únics programaris que s'utilitzaran són:

Java (jdk1.5.0_07): Per al codi font de la aplicació.
JUnit (v4.1): Llibreria Java open source per a la compilació i execució dels tests. S'adjuntarà la llibreria amb el programari.
Apache Ant (v1.6.5): Llibreria Java open source per la automatització d'scripts de compilació i execució. Ens servirà per a realitzar la construcció del paquet i executar tant els tests com la aplicació.
Xerces (v2.8.0): Lliberia Java open source que ens permetrà llegir fitxers XML (els XMI). S'adjuntarà la llibreria amb el programari.

La versió de cada llibreria s'ha escollit amb un únic criteri: la que ja estigués instal.lada. És per això que la sel.lecció d'aquestes versions no ha seguit cap criteri d'incompatibilitat entre versions ni cap altre motiu que s'hagi de comentar.

Al respecte dels fitxers XMI s'ha escollit compatibilitzar amb la versió 2.1, ja que el programari que s'ha utilitzat per a generar els fitxers de model ha sigut Altova Umodel 2007, compatible amb aquesta versió.

### 1.3.3. Detalls de la implementació

#### Organització del codi

A partir del nou disseny s'ha pogut comprobar que l'aplicació es pot desglossar en les següents peces: analitzador, metamodel, lector de model, mètriques, i pintador de resultat/s. I tal com ja s'ha comentat, la única peça que queda independent del metamodel és l'analitzador, la resta hi tenen dependències.

Així doncs, la organització del codi queda així (els paquets són packages Java):

Paquet umlmetrics.kernel: Codi de l'analitzador.
Paquet umlmetrics.v1.metamodel: Codi del metamodel.
Paquet umlmetrics.v1.xmi: Codi del lector de fitxers XMI.
Paquet umlmetrics.v1.metrics: Codi de les mètriques.
Paquet umlmetrics.v1.resultprinters: Codi dels pintadors de resultats (hi ha els sets).
Paquet umlmetrics.v1.main: Codi d'entrada a la aplicació, o sigui, el configurador del analitzador.

Cada paquet té un paquet associat anomenat test on hi haurà les classes corresponents als tests. Així els tests de les classes del paquet umlmetrics.kernel estaran ubicats en el paquet umlmetrics.kernel.tests. Dins aquests paquets s'hi poden trobar dos tipus de classes Java: les anomenades TestAuxClassXXX, classes “dummy” (classes inútils) que són d'ús auxiliar per als tests; i les classes anomenades TestNomDeClasse, que són les que proven la classes NomDeClasse.

Tots els paquets menys el del analitzador tenen un v1. Aquest fet és degut a que si apareixen noves versions de lectors de models, o noves mètriques i/o pintadors de resultats es podríen ubicar en els mateixos paquets, però si es canvia la versió del metamodel (v1) tots els altres paquets que tenen l'indicador v1 quedaran afectats. D'aquesta forma s'intenta que les noves versions del metamodel s'ubiquin amb l'indicador de versió fins que es trobi una llibreria de suficient confiança i estabilitat; és a dir, que si s'implementa una segona versió del metamodel, com que segurament caldrà revisar el codi de tots els paquets v1, es preten que s'anomeni v2 als nous paquets amb codi corresponent de totes les peces revisades.

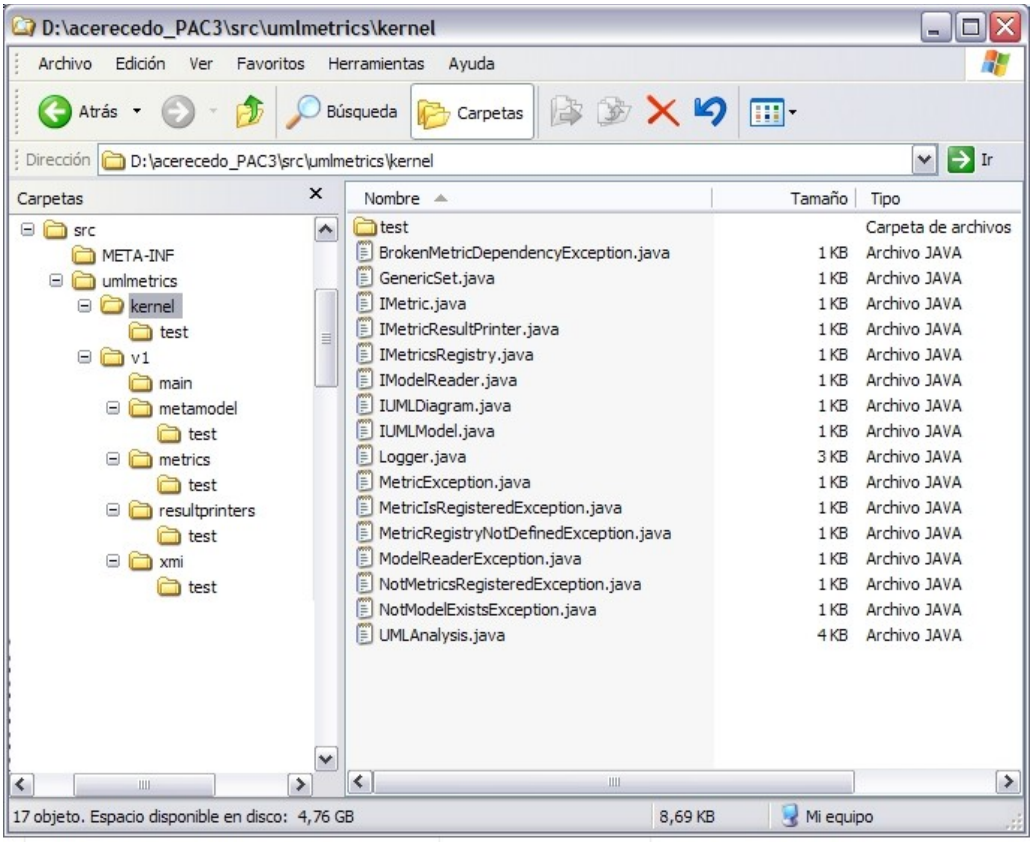



*Fig. 5, Classes del paquet umlmetrics.kernel*

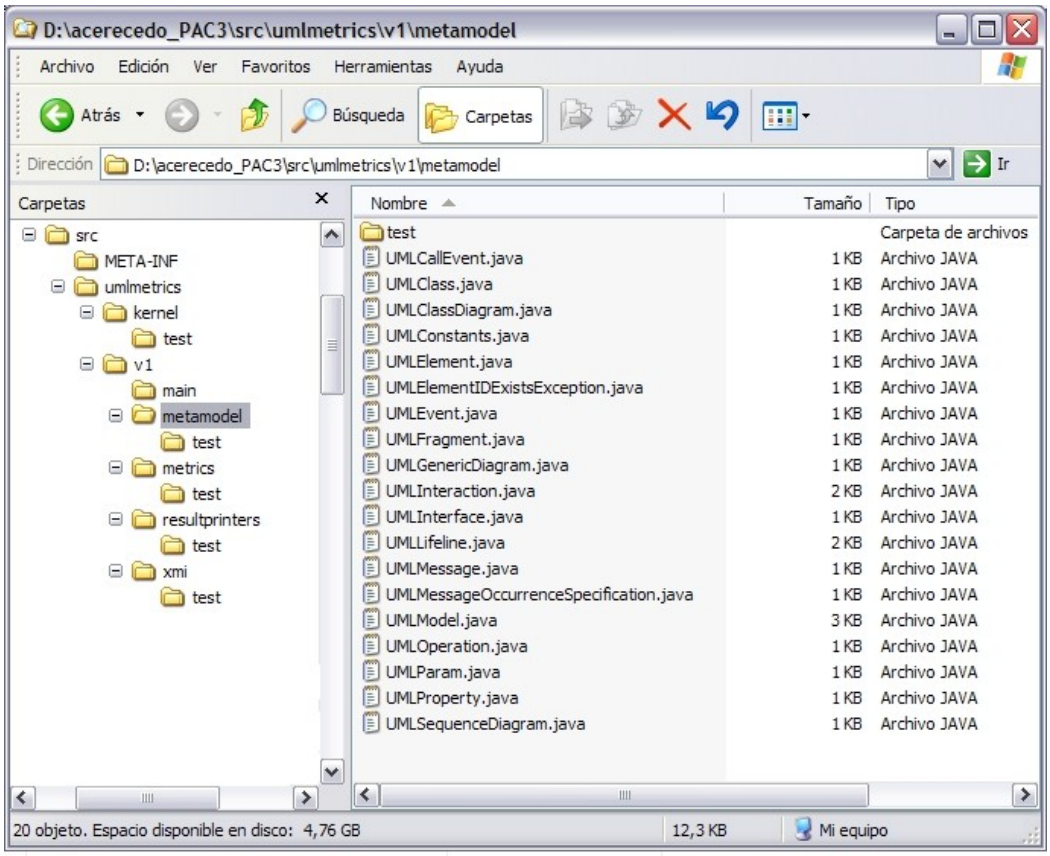



*Fig. 6, Classes del paquet umlmetrics.v1.metamodel*

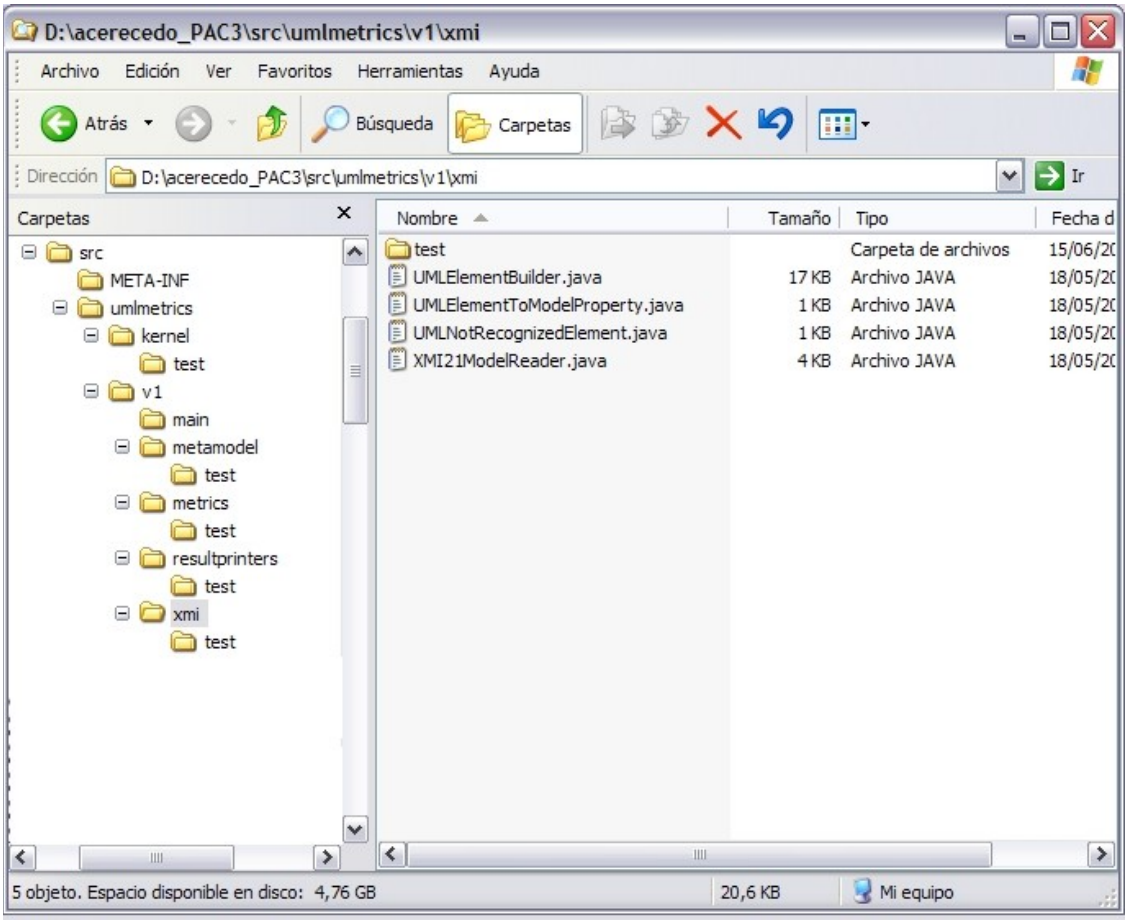



*Fig. 7, Classes del paquet umlmetrics.v1.xmi*

**Organització del programari entregat**

Dins el fitxer zip on es troba aquesta documentació es poden observar una sèrie de fitxers:

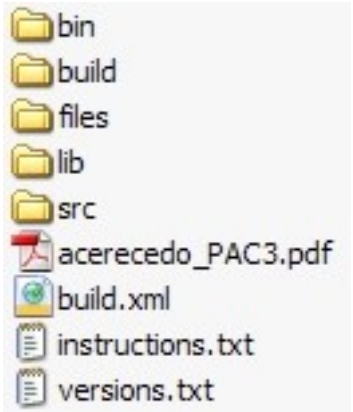



*Fig. 8, Carpeta arrel del programari entregat*

Carpeta bin: Conté el jar generat en la construcció del paquet, amb totes les classes implementades.

Carpeta build: Aquesta carpeta es genera dinàmicament quan es compila i construeix la aplicació. Contindrà les classes que posteriorment s'utilitzaran per a crear el jar de la carpeta bin. Es pot borrar ja que no té cap utilitat excepte la creació del jar.

Carpeta files: Conté els fitxers relacionats amb la aplicació. Això implica tant els fitxers d'entrada umlmetricsInput.xml i els XMI, com els fitxers generats pels tests -carpeta testsreports-, com els fitxers de sortida errors.txt i output.txt.

Carpeta lib: Conté dues de les llibreries necessaries per a construir i executar la aplicació (JUnit i Xerces), les altres dues que es requereixen (Java JDK i ANT) han d'estar instal.lades.

Carpeta src: Conté els fonts de la aplicació, organitzats en les carpetes que defineixen els packages Java.

Fitxer build.xml: És el fitxer que conté el codi ANT per a la construcció i execució de la aplicació, així com els tests.

Fitxer instructions.txt: És el fitxer que especifica com utilitzar el fitxer ANT, indicant quines crides s'ha de fer per a realitzar les diferents accions disponibles.

Fitxer versions.txt: S'adjunta aquest petit fitxer que recorda les versions d'aquest paquet de programari per poder realitzar possibles actualitzacions.

**Consideracions implementació**

umlmetrics.v1.main.UIConsole: Tot i que desafortunat, la intenció del nom era indicar que aquesta aplicació (ja que aquesta classe fa de punt d'entrada) mostra els resultats per consola -sortida estàndard-. La veritat és que aquesta classe només s'encarrega de llegir els paràmetres necessaris per a executar la aplicació del fitxer files/umlmetricsInput.xml, principalment els noms de les classes que implementen tant les mètriques com els pintadors de resultats, instanciar les classes, i passar les referències d'aquests objectes a l'analitzador i executar l'anàlisis del model. Realment aquesta classe s'hauria d'anomenar UIAnalysisConfig -o similar-, ja que permet qualsevol tipus de pintat de resultats ja que es cosa de les classes que implementen IResultPrinter. També es desafortunada la el.lecció del nom del paquet, ja que no quedarà afectada en cas de modificacions del metamodel.

Loggers: En la aplicació hi ha dues classes que implementen la manera de mostrar els missatges de la aplicació i dels pintadors de resultats: umlmetrics.kernel.Logger i umlmetrics.v1.resultprinters.ConsoleMsg, respectivament. La classe Logger s'ha posat en el kernel de l'analitzador degut a que bàsicament permet redirigir els missatges d'error, de forma que és una classe utilititaria que centralitza aquesta acció (redirecció de missatges). La classe ConsoleMsg és també una classe utilititaria però està ubicada en el paquet dels resultPrinters, ja que centralitza la redirecció dels missatges a la consola per a tots els pintadors de resultats, simplificant el codi d'aquests.

Metamodel: Com s'ha comentat en el apartat del redisseny, aquest queda definit pel conjunt de classes del paquet umlmetrics.v1.metamodel. En aquest paquet s'han anat afegint totes les classes que són representatives dels diferents elements UML i que normalment porten associat un tag XMI. El fet d'anar afegint les classes a mida que es necessitaven ha generat que moltes classes s'assemblin molt als elements XMI, empitjorant la possible reutilització del metamodel en versions posteriors. Com que per aquesta primera versió de l'analitzador es permet la interpretació de diagrames de classes i de seqüència (no tots els elements codificats en un XMI d'aquests diagrames estan suportats, només aquells que han sigut necessaris per a implementar les mètriques disponibles), en el metamodel podem trobar-hi tant classes relacionades amb el model en general (classes, etc) com específiques del diagrama de seqüències (classe UMLFragment). La classe principal, i que implementa la interfcíe IUMLModel és la classe UMLModel.

Lector del model, umlmetrics.v1.xmi.XMI21ModelReader i UMLElementBuilder: Realment la lògica de la creació dels elements del metamodel a partir dels tags XMI resideix en la classe UMLElementBuilder, la classe XMI21ModelReader només s'encarrega de governar la lògica XML associada als fitxers XMI. En aquesta primera versió, UMLElementBuilder codifica les constants d'un fitxer XMIv2.1, partint d'un fitxer XMI generat per Altova Umodel 2007. Això vol dir que si hi ha declaracions en els fitxers XMI que són pròpies d'Altova el lector no serà compatible amb altres implementacions XMI. En qualsevol cas és la factory de les instancies del metamodel.

Mètriques (paquet umlmetrics.v1.metrics): En aquest paquet s'hi ubiquen totes les classes que implementen alguna mètrica, les classes de resultats de la mètrica, i la classe genèrica de les mètriques per a la primera versió del metamodel. Les classes de resultats són agregacions de les mètriques i intenten ser el màxim d'independents del metamodel, però és impossible separar-ho del tot i tots els contenidors de resultats d'una mètrica contenen referències a l'identificador únic d'element UML, per si el pintador de resultats vol acedir-hi per a mostrar més informació relacionada amb el resultat obtingut. En aquesta primera versió el control d'errors per part de les mètriques és exclusivament de dependències amb els diagrames definits en el model, de forma que és de suposar que són bastant millorables. Hi ha una forta dependència entre totes les classes d'aquest paquet i el metamodel, així és de preveure que canvis en el metamodel afectin a la lògica de totes les mètriques.

Pintadors de resultats, Sets (paquet umlmetrics.v1.resultprinters): En aquesta primera versió els pintadors de resultats recuperen les mètriques que requereixen i pinten els resultats obtinguts (ChidamberAndKemerer és limita a mostrar el resultat de les mètriques que defineix excepte la mètrica “Falta de cohesió dels mètodes”, que no està implementada). S'ha decidit no implementar valoracions perque excedia del temps disponible per a la realització d'aquest projecte, i perque s'ha pogut observar que la valoració de les mètriques és, en la major part dels casos, comparativa i no declarativa. Tot i que les mètriques retornen informació detallada de les classes analitzades per aquesta primera versió els pintadors de resultats es limiten a mostrar les mitjes del valor que analitzen per la sortida estàndard.

Tests: Per a la implementació dels tests s'han creat classes auxiliars (classes TestAuxClassXXX) i classes genèriques (umlmetrics.kernel.test.TestSet per a tots els pintadors de resultats i classe umlmetrics.v1.metrics.test.TestGenericMetric per a totes les mètriques). Totes les classes relacionades amb els tests s'han ubicat en els packages amb el sufix “.test”. Totes utilitzen la llibreria JUnit ja que facilita molt la implementació i execució dels tests. També s'ha utilitzat la filosofia JUnit, primer s'ha escrit el test i després s'ha implementat la classe que representava. Tot i ser una bona filosofia, és de suposar que alguns tests seran inclomplets i no evaluaran totes les possibilitats i/o estats de les peces que verifiquen.

**Construcció i execució de la aplicació**

Tal com indica el fitxer instructions.txt, el primer que cal per poder utilitzar aquesta aplicació és tenir instal.lada una versió 1.5 del jdk de Java (la aplicació només s'ha provat amb la versió 1.5.0_07, així que tot i que pel tipus de codi utilitzat no sembla que versions posteriors puguin donar problemes, cal recordar que no s'han realitzat les proves. Pel tipus de codi utilitzat, les versions inferiors a la 1.5 no funcionaran).

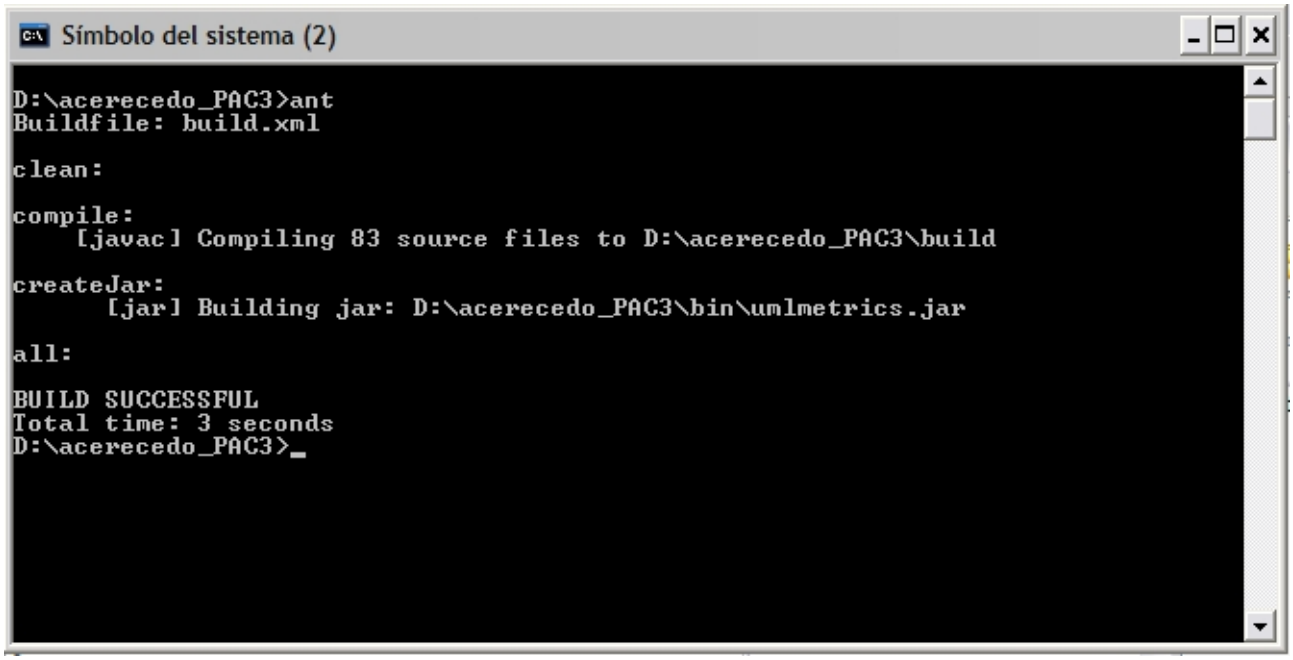
Posteriorment a la instal.lació del jdk de Java cal tenir instal.lat el Apache ANT, un paquet que es pot trobar en la adreça indicada en el apartat de “Enllaços” del final d'aquest document. El paquet es distribueix com un zip que després de descomprimir en una carpeta qualsevol, s'estableixen una sèrie de variables globals i ja es pot utilitzar (de totes formes es recomana llegir les instruccions que adjunta el zip amb els binaris de l'Apache ANT).

Un cop tenim els dos paquets necessaris, ja podrem compilar / construir, executar els tests, i executar la aplicació. El fitxer build.xml conté el codi que realitza aquestes tasques, de forma que si es produeixen canvis en les interfcíes de la aplicació, en la organització del codi, de la classe d'entrada -la que té el mètode main-, etc, caldrà revisar el codi d'aquest fitxer.

En qualsevol cas, però, en el moment d'entregar aquesta primera versió d'aquest paquet de programari les crides que permet fer aquest fitxer són les següents:

Compilació / Construcció: S'executa amb la comanda: *ant* (desde la carpeta on estigui ubicat el build.xml). Aquesta acció crea la carpeta build si no existeix, compila les classes de la aplicació amb les llibreries que requereixi, i un cop les té compilades genera el jar amb l'aplicació en la carpeta bin.

*Fig. 9, Execució de la construcció del programari*




Execució dels tests JUnit que s'ha implementat amb l'aplicació: S'executa després d'assegurar-se que en el classpath està definida la llibreria junit-4.1.jar ja que es requereix per que ANT entengui la comanda “junit”. Un cop tenim aquest requisit podrem executar aquesta acció teclejant la comanda *ant junittest*. Aquesta acció ha de retornar un missatge de tipus “build succesfully” indicant que ha anat bé (com tota comanda ANT), fet que voldrà dir que totes les classes de test JUnit que s'hàgin executat que estiguèssin dins la aplicació tindran un fitxer associat amb els resultats de la seva execució en la carpeta files/testsreports (per exemple, el fitxer resultant de la execució de la classe de test TestUMLAnalysis ubicada en el paquet umlmetrics.kernel.test s'anomenarà TEST-umlmetrics.kernel.test.TestUMLAnalysis.xml). Cal comentar que en el codi ANT del fitxer build.xml s'indica, en un comentari, que caldria crear una classe Java de test que centralitzés tots els tests de la aplicació, i així desde el fitxer ANT només s'hauria de cridar a aquesta classe quan es vulguèssin executar els tests. Ara mateix totes les classes abstractes amb tests (per que altres tests aprofitin codi comú) s'han d'excloure desde la tasca ANT ja que sinó dóna error (i no s'executen els tests). Els tests no requereixen fitxers d'entrada ja que només proven les funcionalitats de les diferents peces de la aplicació, i si alguna classe de test ho requereix, crea un fitxer temporal per poder-lo utilitzar (aquesta classe de test és la umlmetrics.v1.xmi.test.TestXMI21ModelReader).

*Fig. 10, Execució dels tests*


Símbolo del sistema (2)
D:\acererecedo_PAC3>set CLASSPATH=%CLASSPATH%;lib/junit-4.1.jar
D:\acererecedo_PAC3>ant junittest
Buildfile: build.xml
compile:
junittest:
[junit] Running umlmetrics.kernel.test.TestUMLAnalysis
[junit] Tests run: 5, Failures: 0, Errors: 0, Time elapsed: 0,407 sec
[junit] Testsuite: umlmetrics.kernel.test.TestUMLAnalysis
[junit] Tests run: 5, Failures: 0, Errors: 0, Time elapsed: 0,407 sec
[junit] Running umlmetrics.v1.metamodel.test.TestUMLModel
[junit] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 0,266 sec
[junit] Testsuite: umlmetrics.v1.metamodel.test.TestUMLModel
[junit] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 0,266 sec
[junit] Running umlmetrics.v1.metrics.test.TestCouplingBetweenObjectClasses
[junit] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 0,25 sec
[junit] Testsuite: umlmetrics.v1.metrics.test.TestCouplingBetweenObjectClass
es
[junit] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 0,25 sec
[junit] Running umlmetrics.v1.metrics.test.TestDepthInheritanceTree
[junit] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 0,25 sec
[junit] Testsuite: umlmetrics.v1.metrics.test.TestDepthInheritanceTree
[junit] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 0,25 sec
[junit] Running umlmetrics.v1.metrics.test.TestNumPubOps
[junit] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 0,25 sec
[junit] Testsuite: umlmetrics.v1.metrics.test.TestNumPubOps
[junit] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 0,25 sec
[junit] Running umlmetrics.v1.metrics.test.TestNumberOfChildren
[junit] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 0,266 sec
[junit] Testsuite: umlmetrics.v1.metrics.test.TestNumberOfChildren
[junit] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 0,266 sec
[junit] Running umlmetrics.v1.metrics.test.TestResponseForClassFirstStep
[junit] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 0,266 sec
[junit] Testsuite: umlmetrics.v1.metrics.test.TestResponseForClassFirstStep
[junit] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 0,266 sec
[junit] Running umlmetrics.v1.metrics.test.TestWeightedMethodsClass
[junit] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 0,296 sec
[junit] Testsuite: umlmetrics.v1.metrics.test.TestWeightedMethodsClass
[junit] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 0,296 sec
[junit] Running umlmetrics.v1.resultprinters.test.TestChidamberAndKemerer
[junit] Tests run: 6, Failures: 0, Errors: 0, Time elapsed: 0,282 sec
[junit] Testsuite: umlmetrics.v1.resultprinters.test.TestChidamberAndKemerer
[junit] Tests run: 6, Failures: 0, Errors: 0, Time elapsed: 0,282 sec
[junit] Running umlmetrics.v1.resultprinters.test.TestNumPubOpsResultPrinter
[junit] Tests run: 6, Failures: 0, Errors: 0, Time elapsed: 0,266 sec
[junit] Testsuite: umlmetrics.v1.resultprinters.test.TestNumPubOpsResultPrin
ter
[junit] Tests run: 6, Failures: 0, Errors: 0, Time elapsed: 0,266 sec
[junit] Running umlmetrics.v1.xmi.test.TestUMLElementBuilder
[junit] Tests run: 5, Failures: 0, Errors: 0, Time elapsed: 0,391 sec
[junit] Testsuite: umlmetrics.v1.xmi.test.TestUMLElementBuilder
[junit] Tests run: 5, Failures: 0, Errors: 0, Time elapsed: 0,391 sec
[junit] Running umlmetrics.v1.xmi.test.TestXMI21ModelReader
[junit] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 0,328 sec
[junit] Testsuite: umlmetrics.v1.xmi.test.TestXMI21ModelReader
[junit] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 0,328 sec
BUILD SUCCESSFUL
Total time: 7 seconds
D:\acererecedo_PAC3>

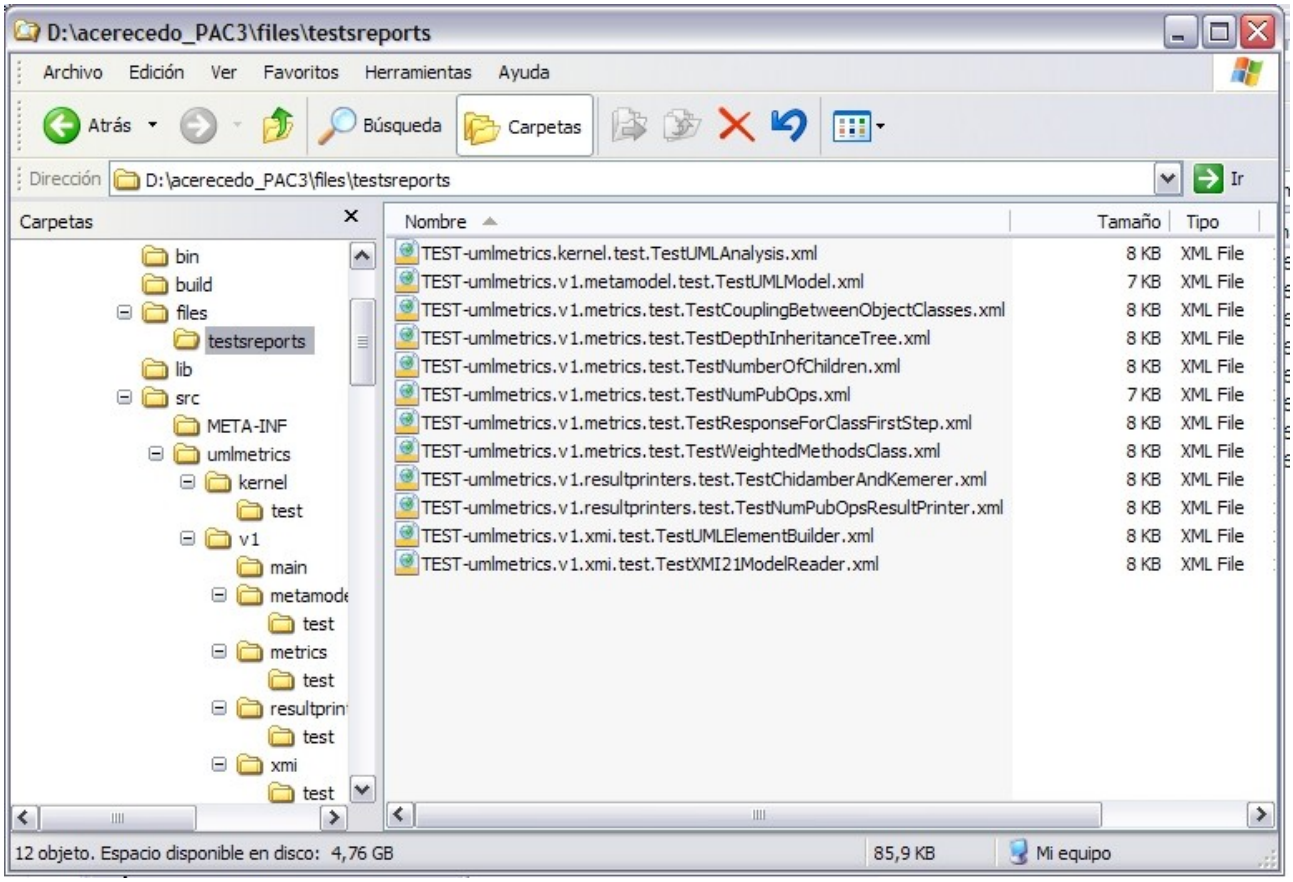



*Fig. 11, Carpeta amb els resultats de tots els tests*


*Fig. 12, Fitxer XML amb el resultat d'una classe de test*

Execució de la aplicació: La aplicació té com a punt d'entrada la classe umlmetrics.v1.main.UIConsole, que és la qui defineix el mètode main. Aquesta classe espera un paràmetre indicat amb el atribut -x que indica el nom del fitxer XML amb els diferents paràmetres d'entrada de la aplicació (fitxer files/umlmetricsInput.xml). Així la acció ANT crida a aquesta classe passant-li com a paràmetres la -x i el nom del fitxer files/umlmetricsInput.xml. Aquest fitxer permet indicar les mètriques (tags mètrics) i els pintadors de resultats

(tags resultPrinter) que s'han d'executar en el anàlisis, juntament amb el fitxer XMI que conté el model, i opcionalment permet redirigir les sortides estàndard i d'error (System.out i System.err) a fitxer indicant-ho amb els tags system.out i system.err.

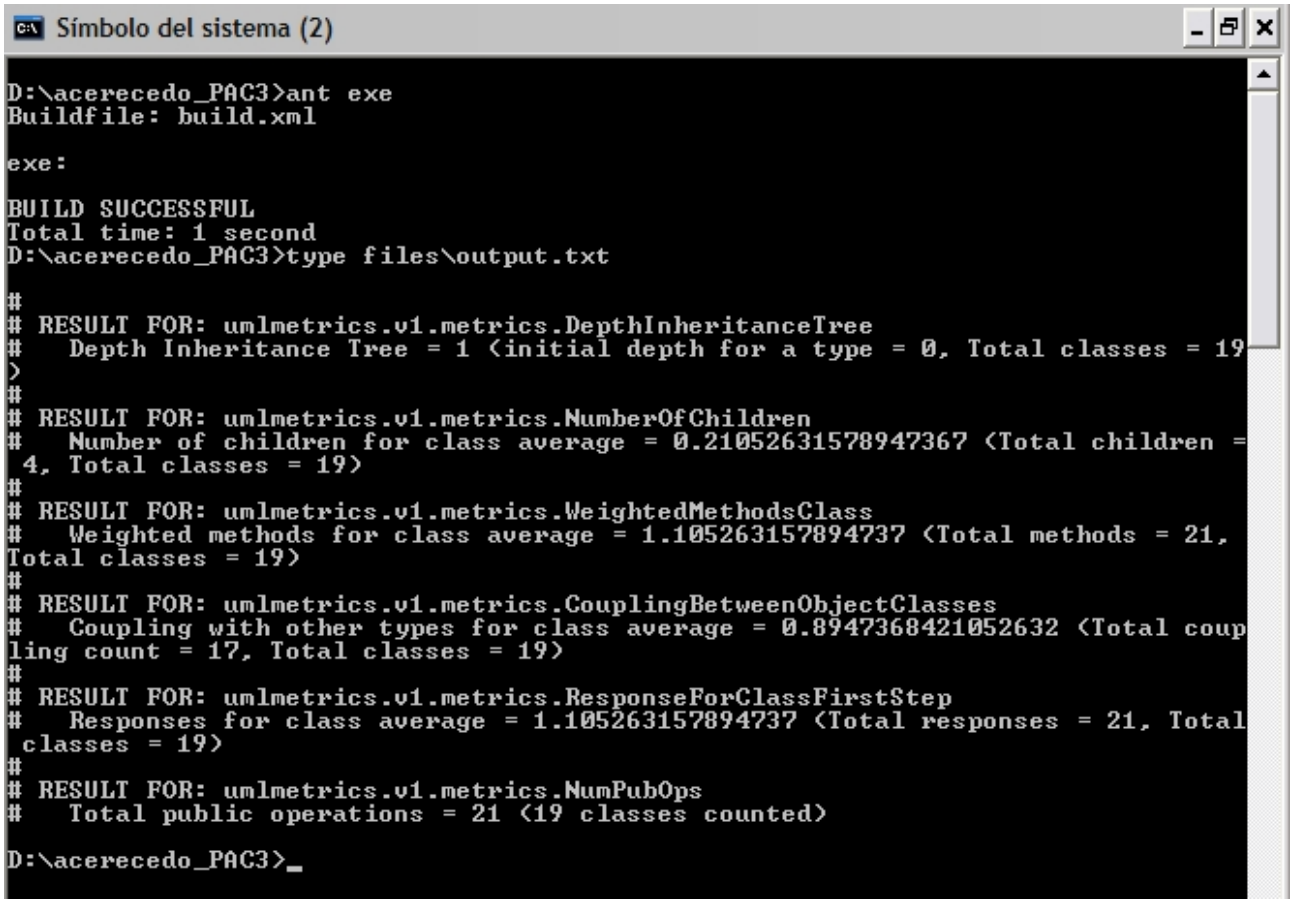



*Fig. 13, Execució de l'analitzador*

```
<?xml version="1.0" encoding="UTF-8"?>
<umlmetrics>
    <xmimodel file="./files/Altova2007MyApp.xmi"/>
    <metric className="umlmetrics.v1.metrics.CouplingBetweenObjectClasses"/>
    <metric className="umlmetrics.v1.metrics.DepthInheritanceTree"/>
    <metric className="umlmetrics.v1.metrics.NumberOfChildren"/>
    <metric className="umlmetrics.v1.metrics.WeightedMethodsClass"/>
    <metric className="umlmetrics.v1.metrics.ResponseForClassFirstStep"/>
    <metric className="umlmetrics.v1.metrics.NumPubOps"/>
    <resultPrinter className="umlmetrics.v1.resultprinters.ChidamberAndKemerer"/>
    <resultPrinter className="umlmetrics.v1.resultprinters.NumPubOpsResultPrinter"/>
    <system.out file="./files/output.txt"/>
    <system.err file="./files/errors.txt"/>
</umlmetrics>
```

*Fig. 14, Fitxer XML amb els paràmetres d'entrada de l'analitzador*

En el moment d'entregar aquesta primera versió de la aplicació s'han realitzat, previament, una construcció (per tant s'entrega amb la carpeta build creada i amb els fitxers .class amb el bytecode dels fonts Java, i amb el jar de la aplicació bin/umlmetrics.jar), la execució dels tests implementats (per tant dins la carpeta files hi podrem veure la carpeta testsreports amb tots els XMLs corresponents a les diferents classes de test de la aplicació), i una execució que redirigeix la sortida estàndard i d'error als fitxers files/output.txt i files/errors.txt. En el moment que es tornin a executar aquestes tasques es sobreescriuran aquests fitxers.

**Manteniment**

Creació de noves mètriques: Tal com està definida i implementada la aplicació es de suposar que una utilització d'aquesta implicaria implementar noves mètriques. Com que per aquesta versió es parteix d'un metamodel en constant evolució, la forma de crear noves mètriques implicarà crear la classe de test de la mètrica (veure classe umlmetrics.v1.metrics.test.TestNumberOfChildren com a exemple), la classe que implementi la mètrica (veure classe umlmetrics.v1.metrics.NumberOfChildren com a exemple), i algun pintador de resultats que mostri els seus resultats (veure classes TestNumPubOpsResultPrinter i NumPubOpsResultPrinter del paquet umlmetrics.v1.resultprinters i el de test com a exemple). Si la mètrica requereix noves dades del metamodel (gairebé segur), també caldrà afegir o modifcar algun test en la classe umlmetrics.v1.xmi.test.TestUMLElementBuilder (aquesta classe implementa els tests de la càrrega dels elements necessitats del metamodel desde un fitxer XMI, de forma que al verificar que aquests es carreguen bé al model també verifiquem que estiguin definits al metamodel), fet que implicarà, per que funcioni el test, modificar el UMLElementBuilder que és qui realment carrega els elements del XMI al metamodel. Si únicament es vol fer que els pintadors de resultats mostrin la informació de forma diferent (per exemple en codi html o utilitzant un panell swing) només cal adaptar les classes del paquet de pintadors (package umlmetrics.v1.resultprinters) o senzillament la classe ConsoleMsg (que és qui redirigeix indirectament els missatges dels pintadors a la consola o sortida estàndard).

Nova configuració / Nous loggers: Si es vol utilitzar alguna api de traces (per exemple log4j), només caldrà modificar la classe Logger. Aquest fet pot impactar amb la classe UIConsole ja que segurament no calgui passar-li com a paràmetres els noms de fitxer on es vol redirigir la sortida estàndard i la d'error. Si es vol modificar la forma de declarar les mètriques o els pintadors de resultats a executar només caldrà adaptar la classe UIConsole per a permetre les noves funcionalitats.

Nova versió XMI / Nou lector: Si apareix un nou producte que genera XMI i utilitza algun tipus de format diferent, la aplicació està pensada per a encavir un nou lector en el mateix paquet on està el XMI21ModelReader, de forma que la classe UIConsole (factory del lector), al rebre els paràmetres d'entrada de l'usuari, haurà de saber quina instancia crear. Si es vol determinar qui pot carregar el model després d'haver llegit el fitxer (per exemple perque segons el fitxer es sabrà de quina versió o proveïdor és i per tant qui és el qui sabrà llegir-ho) el que s'haurà de fer és crear un lector DOM del fitxer XMI, i un parsejador del DOM al pròpi metamodel (és important mantenir la capa intermitja del metamodel perque separarem les mètriques dels diferents tipus de persistències de model que pugui haver-hi, per exemple algun nou format d'algun fabricant com Rational que no sigui compatible amb XMI -això és una suposició-).

Nou metamodel: Com ja s'ha comentat extensament, si es vol utilitzar una nova versió del metamodel per encavir qualsevol definició de model, caldrà crear un nou paquet que s'anomeni umlmetrics.v2.metamodel i que implementi les interfcíes IUMLModel i IUMLDiagram. Aquestes dues interfícies gairebé no defineixen cap signatura, de forma que intenten encavir qualsevol tipus d'implementació d'un metamodel. En cas, però, que el metamodel no pugui tenir en compte els diagrames com a dependències de les mètriques, s'haurà de buscar algun nou tipus de sistema de validació d'integritat de dades del model perque la mètrica no retorni valors incoherents, o senzillament es podria arribar a eliminar el sistema de validació de dependències amb els diagrames (és recomanable, però, que es mantingui fins que UML no deixi de tenir diagrames o fins que els models de les aplicacions no siguin UML). Evidentment caldrà revisar el codi de les mètriques ja que constantment estan accedint al metamodel per a poder recuperar i analitzar els seus valors.

**Valoracions respecte la implementació i el nou disseny**

Després d'haver realitzat el disseny, la implementació i les execucions de test i d'exemple, podem arribar a algunes conclusions sobre la qualitat del disseny de l'aplicació i sobre alguns dels objectius esperats abans de començar amb el disseny.

En primer lloc cal comentar que el disseny presenta massa acoblament entre les diferents peces de la aplicacio, fet que implicarà un esforç en el seu manteniment i que pot provocar introduir errors al modificar funcionalitats.

El conjunt de mètriques de Chidamber and Kemerer no s'ha pogut realitzar. Ni estan completes totes les mètriques que defineix (falta implementar la mètrica “Falta de cohesió dels mètodes”) ni fa cap valoració

sobre els resultats obtinguts (simplement es limita a pintar els resultats). És un objectiu incomplet.

Un altre objectiu incomplet és la reutilització del codi de les mètriques. No s'ha trobat la manera de conseguir que mètriques semblants puguin aprofitar trossos de codi comuns. Evidentment es podria haver creat algunes mètriques generals de les que en depenguèssin les mètriques reals, de forma que el recorregut de totes les operacions de totes les classes d'un model només es fés una vegada. Com que aquesta sol.lució implicaria utilitzar algun tipus de patró d'escoltador de canvis de propietats entre les mètriques, i especificar les dependències entre mètriques (i no amb diagrames UML), suposava massa impacte en el redisseny pel canvi en el rendiment associat (tampoc és una sol.lució que aporti moltes avantatges respecte la actual ja que normalment un model pot tenir desde 10 fins a 150 classes, suficientment pocs elements com per crear tot un sistema que per reduir el cost de recorreguts tant petits incrementi el cost amb més execucions iteratives -per realitzar les notificacions al les diferents mètriques que hi depenguin-), s'ha decidit deixar la implementació tal com quedava plantejada pel redisseny.

Finalment cal tornar a recordar el ja comentat varies vegades en aquest document: aquest disseny té un alt acoblament amb el metamodel, que a més s'ha construït a mida que es requeríen noves entitats del model UML per a realitzar els càlculs de les mètriques, fet que ha degenerat en una excesiva dependència d'una peça segurament mal construïda. En un futur es podrà veure l'impacte que tindrà aquesta mala decisió més acuradament, però segur que afectarà a suficients mètriques com per poder abançar que caldrà estar a l'aguaït de la aparició d'alguna llibreria fiable i accessible que implenti un metamodel que encaveixi els models UML (per exemple alguna implementació de MOF).

## 1.4. Valoració econòmica

Aquest projecte s'ha realitzat únicament amb un recurs humà. Tenint en compte la poca experiència de l'estudiant en aquesta àrea, sumat al fet que encara és un estudiant, la valoració partirà de la atribució de la categoria de becari al recurs humà.

Així que partint d'un salari de becari de 3,75 € la hora, i tenint en compte la següent imputació d'hores:

FASE 1
Recerca mètriques: 48 hores
Recerca anàlisis semàntica: 18 hores
Disseny aplicació: 32 hores

FASE 2
Redisseny: 16 hores
Implementació: 64 hores

FASE 3
Compilació documentació / memòria: 12 hores
Anàlisis de resultats obtinguts: 6 hores
Realització defensa: 32 hores

TOTAL HORES = 228
**TOTAL PROJECTE** = 228 * 3,75 = **855 euros**

## 1.5. Conclusions

Un cop realitzat l'aplicatiu és el moment de fer les valoracions finals. Ara bé, cal remarcar que el fet que s'hagi produit un redisseny ha afectat bastant al desenvolupament general del projecte, introduint un esforç adicional que ha restat atenció de punts que es podríen considerar més importants en la àrea dels anàlisis de models UML.

Els dos principals objectius que no s'han pogut abordar degut a aquesta incidència són: el tema de la formalitat d'un model a partir de les declaracions OCL, així que si algun dia es vol introduir un anàlisis semàntic caldrà implementar-lo en el carregador ja que la arquitectura no el considera; i les valoracions de l'anàlisis, que tampoc s'han abordat i per tant l'únic tipus de resultat que mostra la aplicació és absolutament numèric i dificilment interpretable si no hi ha una experiència previa amb aquest tipus d'anàlisis.

En general la aplicació permet la execució d'un anàlisis d'un model, permetent la codificació d'un model en el format XMI, configurar el anàlisis perque es declarin les mètriques a executar, i permetent la definició dels analitzadors de resultats que permetran mostrar/interpretar els resultats obtinguts. És a dir, la aplicació satisfà un alt grau de configuració que servirà per adaptar-se a una gran quantitat de situacions, fet que facilita la seva utilitat.

El rendiment de la arquitectura és exponencial. Això és degut a que aquest depen directament de dues variables: el nombre d'elements del model i el nombre de mètriques a executar. Però com que el nombre d'elements d'un model és relativament baix i constant entre diferents models (més o menys podem establir que ningun model excedeix de les 100 classes o de 10 diagrames de seqüència amb una mitja de 10 lifelines per cada diagrama), també podríem establir que el cost és linial respecte el nombre de mètriques a executar. Així, si haguèssim conseguit la extensibilitat de les diferents mètriques permetent aprofitament d'execució hauríem pogut reduir el cost computacional de la arquitectura per a futures extensions. Però degut a les particularitats de les mètriques entre si, tampoc hauríem pogut reduir tant el cost de les seves execucions com amb el fet de carregar els diferents elements del model en taules de dispersió (hashtables), fet que optimitza la cerca dels elements requerits per una mètrica i permet obtenir uns temps d'execució acceptables (la execució de les 6 mètriques implementades triga al voltant d'un segon).

Al respecte de la coexistència de diferents versions XMI o de la definició d'un model a partir de diferents fitxers podem dir que s'ha aconseguit indirectament, ja que al plantejar un anàlisis configurable deixem per al agent del anàlisis la forma de determinar aquestes qüestions. Així, si volem compatibilitzar diferents versions XMI només caldrà adaptar el configurador perque determini les versions dels diferents fitxers que pugui rebre com a paràmetres, i instanciar el carregador compatible amb la versió. Él cas de la definició d'un model a partir de diferents fitxers XMI és el mateix cas, ja que només caldrà adaptar el configurador i el carregador de model sense tenir que tocar la arquitectura.

Un dels principals objectius (ja que venia desde abans del primer disseny) ha sigut la separació entre la implementació del model (per exemple XMI) de la implementació de les mètriques. Podem dir que la arquitectura ho ha aconseguit a costa de la excesiva dependència de totes les peces amb el metamodel.

Així que de tots els objectius plantejats durant la realització d'aquest projecte, declarats tant en les etapes de planificació com en les de disseny, podem dir que en general s'han pogut satisfer però no amb èxit total. De totes formes podem considerar que la documentació aportada per aquest projecte pot servir de punt de partida per a futures adaptacions d'aquest aplicatiu, ajudant a conseguir finalment la realització d'un bon analitzador de models UML.

# 2. Glossari

**Acoblament entre elements d'un model**: Interacció entre diferents sistemes o entre els diferents components d'un sistema.
**Agregació**: Part-de, referit a la relació entre dos classes o elements d'un model.
**ANT**: Eina de programació que serveix per la realització de les tasques mecàniques i repetitives, normalment de la fase de compilació.
**API (de software)**: Llibreria que implementa un conjunt de funcions per poder ser utilitzades per un programa de software.
**Bucle**: Cicle d'execució d'una o diverses instruccioons.
**Bytecode Java (fitxer class)**: Fitxer que conté el codi d'una classe Java a interpretar per la Java Virtual Machine.
**Crida asíncrona**: Invocació d'una funció la qual retorna el control quan ha acabat la seva feina.
**Classpath**: Variable global del sistema que requereix Java per a trobar les llibreries que requereixi la execució o compilació.
**Codificació**: Veure 'Implementació'.
**Crida síncrona**: Invocació d'una funció la qual retorna el control just després d'haver sigut cridada i sense perque haver acabat la seva feina.
**Desenvolupament en espiral**: Enfocament metodològic de desenvolupament de software que a cada bucle o iteració (activitat) es planifica la següent activitat.
**Desenvolupament iteratiu**: Enfocament metodològic de desenvolupament de software que consisteix en repetir certes etapes o tasques (inicialització, iteració, i control del projecte) varies vegades.
**Desenvolupament tradicional en cascada**: Enfocament metodològic de desenvolupament de software que consisteix en ordenar rigurosament les diferents etapes del cile de vida del software.
**Diagrama UML**: Sistema gràfic que permet modelitzar un sistema o aplicació.
**Diagrama de classes**: Diagrama UML que modelitza la relació entre les diferents classes del sistema.
**Diagrama de comportament:** Diagrames UML que modelitzen els aspectes dinàmics del sistema.
**Diagrama de casos d'ús**: Diagrama UML que modelitza la interacció externa que ofereix el sistema.
**Diagrama de seqüència**: Diagrama UML que modelitza la interacció entre objectes d'un sistema a través del temps.
**Disseny modular:** Sistema de disseny que busca la divisió de les peces que integren un sistema per a poder optimitzar la seva construcció.
**DOM (Document Object Model):** API que permet accedir, afegir i modificar dinàmicament continguts estructurats (com XML, Html o XMI).
**EMF (Eclipse Modeling Framework)**: Arquitectura d'Eclipse per a modelitzar sistemes.
**Factory**: Patró de disseny que la seva finalitat persegueix centralitzar la creació de les instàncies de les classes.
**Façana gràfica**: Veure 'UI'.
**Gate d'entrada**: Element d'un diagrama de seqüència. Permet declarar una interacció no associada a un lifeline.
**Html (HyperText Markup Language):** Llenguatge estructurat pensat per a textos. És l'estàndard de les pàgines web.
**Implementació**: Realització d'una interfície per part d'una classe.
**Interfície**: Declaració d'un component o peça d'un sistema que busca la separació del codi que la realitza del codi de qui utilitza el component.
**Instancia**: Realització en temps d'execució d'una classe.
**Iteració**: Execució del conjunt d'instruccions d'un bucle.
**Jar**: Fitxer de Java que serveix per a contenir totes les classes executables d'un component o sistema implementat.
**Java**: Llenguatge de programació de Sun Microsistems.
**JMI (Java Metadata Interface):** Estàndard de Java per a la gestió de metadata (dades de les dades).
**JUnit**: API per a Java que ofereix un conjunt d'utilitats per a programar els tests d'un aplicació.
**Lifeline**: Element d'un diagrama de seqüència. Normalment modelitza una instancia.
**Metamodel**: Model dels models UML.
**Mètrica**: Mètode per a mesurar un paràmetre evaluable de la qualitat d'un sistema de software.
**Model UML (Unified Modeling Language)**: Llenguatge gràfic que serveix per a visualitzar, especificar, construir i documentar un sistema de software.
**MOF (Meta-Object Facility)**: Estàndard de la OMG que defineix el metamodel de la arquitectura UML.
**OCL (Object Constraint Language):** Llenguatge d'especificació que utilitza lògica per especificar les propietats invariants de sistemes que es composen de conjunts i relacions entre conjunts.
**OMG (Object Management Group):** Consorci dedicat a l'establiment de diversos estàndards de tecnologíes relacionades amb la orientació a objectes (filosofia de programació).
**Open Source**: Software desenvolupat i distribuit gratuitament que permet la seva modificació al adjuntar sempre el seu codi.

**Package (Java):** Entitat organitzativa de Java que permet separar grups de classes i/o interfícies d'un sistema o component.
**Patró de disseny**: Model que permet la solució d'algun problema en la construcció de software.
**Persistència d'un model**: Sistema de conservació d'un model per a poder-lo utilitzar en diferents execucions.
**Plugin d'Eclipse**: Component pròpi de la arquitectura d'Eclipse (IDE de programació) que permet ampliar la funcionalitat d'Eclipse.
**SAX (Simple API for XML)**: API que permet llegir continguts estructurats.
**Script**: Codi d'un llenguatge interpretat.
**Set (de mètriques)**: Conjunt de mètriques.
**Signatura (d'una classe)**: Conjunt de mètodes i propietats d'una classe.
**Swing**: API gràfica de Java.
**Tag XMI**: Element XML eclusiu del format XMI.
**Tag XML**: Element bàsic de qualsevol document XML.
**UI (User interface)**: Component responsable de mostrar la informació i/o funcionalitat d'una aplicació a l'usuari humà.
**XMI (XML Metadata Interchange)**: Especificació per a l'intercanvi de diagrames entre diferents aplicacions de modelatge de sistemes.
**XML (eXtensible Markup Language)**: Metallenguatge extensible d'etiquetes.

# 3. Bibliografia


POWER, James F. (2006). “Some observations on the application of software metrics to UML models”. Submitted to the Workshop on Model Size Metrics at MoDELS/UML 2006. http://www.cs.nuim.ie/~jpower/Research/Papers/2006/modelsize06.pdf

Genero, M; Miranda, D; Piattini, M. (2002). “Defining and Validating Metrics for UML Statechart Diagrams”. http://alarcos.inf-cr.uclm.es/qaoose2002/docs/QAOOSE-Gen-Mir-Pia.pdf

Tsiolakis, A. (2001). "Semantic Analysis and Consistency Checking of UML Sequence Diagrams1". http://www.informatik.uni-bremen.de/~tsio/papers/AlikiTsiolakisTR2001-06.pdf

Carbone, M; Santucci, G. (2002). "Fast&&Serious: a UML based metric for effort estimation". http://alarcos.inf-cr.uclm.es/qaoose2002/docs/QAOOSE-Car-San.pdf

-, (2003). “UML 2.0 OCL Specification”. OMG Adopted Specification. http://www.omg.org/docs/ptc/03-10-14.pdf

-, (2005). “MOF 2.0/XMI Mapping Specification, v2.1”. OMG formal. http://www.omg.org/docs/formal/05-09-01.pdf

-, (2006). "Unified Modeling Language: Superstructure". Version 2.1. OMG Adopted Specification. http://www.omg.org/docs/ptc/06-04-02.pdf


**Enllaços:**

Java JDK:
http://java.sun.com/

Java JMI:
http://java.sun.com/products/jmi/

Altova 20007 UModel:
http://www.Altova.com/UModel

Eclipse Modeling Framework Project (EMF):
http://www.eclipse.org/modeling/emf/?project=emf
http://dev.eclipse.org/viewcvs/indextools.cgi/emf-home/docs/architecture/architecture-summary.html

NetBeans:
http://www.netbeans.org/

Magic Draw
http://www.magicdraw.com

Apache Ant:
http://ant.apache.org/

JUnit:
http://www.junit.org