Universitat Oberta de Catalunya
Enginyeria Informàtica

Memòria Projecte Final de Carrera

Implementació del Disseny per contractes mitjançant la Tecnologia Orientada a Aspectes

Autor: Oscar Escudero Sanchez
Consultor: Jordi Àlvarez
Gener del 2005

Sumari

L’objectiu d’aquest projecte és la realització d’un prototipus capaç de generar l’entorn necessari per implementar els contractes de les classes que formen part de la llibreria TADs de l’assignatura de Estructura de la Informació, mitjançant la tecnologia orientada a aspectes i el llenguatge java.

Índex de Continguts

Annexos

Capítol 1 . Introducció

En aquest capítol explicarem el plantejament inicial del projecte, els seus objectius principals i la metodologia que hem seguit per dur-lo a terme.

1.1 Descripció del PFC.

Un dels principals objectius de la informàtica es la fiablitat dels seus programes. Un programa és correcte si compleix amb les especificacions, i és robust si és capaç d'afrontar les situacions que no estan cobertes a les especificacions. Un camí per assegurar que un programa pot fer front a aquests imprevistos és l'ús d'un paradigma d'enginyeria del software anomenat Disseny per Contractes (Design by Contract DBC).

El Disseny per Contractes es basa en la premissa que les classes i els clients tenen un acord entre ells: el client ha de garantir certes condicions abans de cridar a un mètode i la classe ha de garantir certes propietats al finalitzar la crida. Anomenem a les condicions a priori pre-condicions. Les propietats a posteriori s'anomenen post-condicions. Si aquestes pre i post condicions s'inclouen al compilador, aquest les pot analitzar i qualsevol violació del contracte entre el client i la classe, es pot detectar immediatament.

El nostre projecte pretén donar suport i facilitar les eines per tal de poder fer ús dels contractes en la especificació d'una classe. Bàsicament consisteix en generar la infrastructura necessària per tal de que durant l'execució d'una classe es comprovin les pre i post condicions especificades, tant a nivell global de classe com a nivell de mètode. L'objectiu final és generar un prototipus per tal que es puguin especificar contractes a la llibreria TADs de l'assignatura d'Estructures de la informació, i que quan s'executi la llibreria es comprovin els contractes.

Per generar la infrastructura necessària per l'ús del Disseny per Contractes s'utilitzarà la Programació Orientada a Aspectes (AOP). La Programació Orientada a Aspectes és un nou paradigma de programació que bàsicament és una extensió de la Programació Orientada a Objectes. A nivell introductori podem definir un aspecte com un bloc de codi que permet l'ampliació del comportament d'objectes existents. La idea és agrupar comportaments que no podem capturar de manera jeràrquica i que es repeteixen en diverses classes. Aquesta capacitat d'ampliar el comportament d'un objecte és el que fa adient l'ús d'aquesta tecnologia per la implementació que farem del Disseny per Contractes.

1.2 Anàlisis de requeriments .

El projecte haurà de donar suport al disseny per contractes. A continuació detallem els requeriments del sistema:

- Selecció de les classes / package origen:
  Es proporcionaran els mecanismes per tal que l'usuari pugui determinar de quines classes es generaran els aspectes associats per tal de implementar les especificacions dels contractes. Les classes origen hauran d'especificar la informació dels contractes seguint una sintaxis coneguda.

- Reconeixement dels contractes:
  L'especificació del contracte es trobarà dintre de les capçaleres de les classes. Aquestes capçaleres poden aparèixer tan a nivell de classe com a nivell de mètode. Els tags que denoten informació de contracte son els següents:
  @ pre <condició>: Denota pre condició. (precondition)
  @ post <condició>: Denota post condició.(postcondition)
  @ inv <condició>: Denota condició de la clase. (invariant)

- Obtenció de les dades del contracte de la classe font:
  El sistema haurà de obtenir les dades dels contractes de la classe font i generar els aspectes adients.

- Generació aspectes:
  Per cada classe origen es generarà un aspecte que sigui la representació/implementació del contracte. L'aspecte resultant tindrà els Pointcuts i Advices adients per tal de garantir la consistència de l'aspecte amb el contracte de la classe origen.

- Sintaxis de les condicions:
  El sistema haurà de reconèixer com a condicions qualsevol sentència java que tingui com a resultat de la seva avaluació un valor booleà.

- Funció OLD:
  El sistema implementarà la funció OLD() per tal de que es puguin comparar objectes abans i després de l'execució d'un mètode. Aquesta funció es podrà utilitzar a l'especificació dels contractes.

- Entorn d'execució:
  El sistema haurà de proporcionar l'entorn adient per tal que quan s'executin les classes origen es comprovin les condicions del seus contractes i es generin els missatges d'error adients que informin al client de la violació del contracte.

1.3 Planificació del projecte.

1.3.1 Metodologia.

Per dur a terme el projecte aplicarem un cicle de vida evolutiu amb prototipatge. Amb aquest cicle de vida pretenem minimitzar els riscos, ja que degut al desconeixement d'algunes de les tecnologies que utilitzarem, existeix un risc molt alt de desviacions de l'estimació inicial.

Utilitzant el cicle de vida evolutiu podem fer front a les desviacions desde els lliuraments inicials, i així modificar el planning o l'abast del projecte. S'han identificat diferents tasques principals que descompondrem en subtasques per tal de tenir una visió més acurada de l'abast del projecte. La descomposició en subtasques és el que s'anomena mètode de fites en miniatura i juntament amb el cicle de vida evolutiu ens permetrà tenir un prototipus des de l'inici del projecte i identificar els riscos potencials desde un estadi molt primari del projecte, permetent-nos prendre les mesures correctores adients.

1.3.2 Tasques i fites.

A continuació detallem les principals tasques del projecte, així com una descomposició en subtasques.

Tasca 1: Familiarització amb el problema.

1. Estudi de la llibreria nova de TADs de l'assignatura de Estructura de la Informació.
2. Estudi del disseny per contractes.
3. Especificar el contracte per a la classe PilaVectorImpl fent servir tags tipus javadoc: @pre, @post, @inv

Tasca 2: Estudi de tecnologia per el parseig de la classe base (Doclet).

1. Estudi de la Doclet API.
2. Implementar un doclet que accepti @pre, @post i @inv i generi un fitxer amb la informació que contenen aquest tags, de la mateixa forma que fa el Javadoc amb els tags propis.

Tasca 3: Estudi de tecnologia d'aspectes (AOP, Aspect J)

1. Estudi de AOP i d'AspectJ
2. Instal·lació del software requerit: AspectJ i Eclipse.
3. Execució dels exemples continguts a les llibreries AspectJ.

Tasca 4: Generació manual d'un aspecte per la classe/contracte PilaVector.

1. Generació de l'aspecte pel contracte PilaVector (tasca 1).
2. Generació jocs de prova per comprovar el funcionament correcte de l'aspecte.

Tasca 5: Arquitectura i Disseny del sistema.

1. Disseny de l'arquitectura del sistema utilitzant especificació UML: diagrames de classes i seqüencials.

Tasca 6: Prototipus creació automàtica d'un aspecte concret.

1. Convertir la sortida del Doclet (tasca 2) per tal de generar automàticament un Aspecte que s'apliqui sobre la classe Java pel contracte PilaVector (tasca 1).
2. Generació jocs de prova per comprovar el funcionament correcte del aspecte.

Tasca 7: Creació d'un prototipus genèric.

1. Parseig dels tags de contracte per adaptar-los a la sintaxis del aspecte.
2. Aplicar el disseny acordat a las tasca 5, al prototipus de la tasca 6, amb l'objectiu d'aconseguir un prototipus que permeti la creació automàtica per al conjunt de classes de la llibreria TAD.
3. Extensió del prototipus mitjançant una aplicació GUI o el pas de paràmetres per tal que es puguin especificar la llibreria/classes origen per la generació dels aspectes.
4. Jocs de proves per tal de comprovar el correcte funcionament de la generació d'aspectes automàtica.

Tasca 8: Ampliació de les funcionalitats.

1. Afegir la funcionalitat Old:
    a. Definir sintaxi (per exemple: $old(attribute))
    b. Això tindrà repercussió després en dos llocs diferents:
        i. Abans d'executar el mètode s'han de guardar els valors antics que es consultaran (únicament aquests). Tenir en compte que s'han de clonar els objectes.
        ii. A la postcondició s'haurà de substituïr l'ús de $old(...) per el valor de l'objecte.
        iii. Al final s'hauran de descartar les còpies.

1.3.3 Anàlisi de riscos.

L'anàlisi de riscos de les tasques i fites del projecte ens a permès identificar els següents riscos:

- Implementació del doclet:

S'ha de implementar un Doclet que parsegi la classe base, per tal d'obtenir la informació del contracte. A més de la informació dels tags del contracte, el Doclet ens haurà de permetre disposar de l'informació de la classe base necessària per tal de crear l'aspecte associat: nom del package, nom de les variables de la classe, constructors, mètodes i els paràmetres tant dels constructors com dels mètodes.

> **Pla de contingència:**
> En el cas que el Doclet no ens permeti accedir a totes les dades de la classe base que necessitem per tal de construir l'aspecte/contracte, haurem de fer un parseig manual d'aquesta classe. Un parseig manual pot implicar la necessitat d'algoritmes de certa complexitat i per tant un augment del temps destinat per la tasca.

- Parseig dels tags de contracte:

Els tags dels contractes poden tenir condicions complexes a mes d'utilitzar la funcionalitat OLD. Aquests tags s'hauran de parsejar i customitzar per tal de construir els Pointcuts i Advices dels aspectes. La customització pot implicar un alt nivell de dificultat per tal d'adaptar el tag.

> **Pla de contingència:**
> Es pot simplificar el parseig i la customització, simplificant la sintaxis de l'especificació dels contractes. S'haurà de restringir la sintaxis i/o introduir certs tokens que simplifiquin el parseig.

- Funcionalitat OLD:

La funcionalitat OLD permet comparar objectes abans i després, de l'execució d'una condició d'un contracte. La funcionalitat OLD requereix una infrastructura per la seva execució. Aquesta infrastructura, similar a una taula de símbols, pot implicar certa dificultat ja que estarà lligada al nivell de parseig que aconseguim.

> **Pla de contingència:**
> En aquest cas que no hi ha cap solució alternativa. En tot cas podem preveure una desviació major que per la resta de riscos. A més, aquesta funcionalitat serà part de l'última evolució del prototipus per tant, en cas que no es pugui implementar, la versió anterior del prototipus seria operativa.

1.3.4 Planificació del Projecte.

A continuació detallem el calendari previst de treball. A més de les dades de planificació , també s'especifica una desviació en les tasques que presenten cert riscs ja sigui per la complexitat o per l'ús d'una tecnologia nova en la que no tenim experiència.

Descripció tasca	Duració Estimada	Data Inici	Data Final	Desviació (risc)
Tasca 1: Familiarització amb el problema, Anàlisi dels requeriments i elaboració de la planificació.	20	3/10/2004	6/10/2004	0%
Tasca 2: Estudi de tecnologia per el parseig de la classe base (Doclet). • *Risc: Implementació del doclet*	30	6/10/2004	9/10/04	20%
Tasca 3: Conceptes bàsics. Estudi de tecnologia d'aspectes (AOP, Aspect J)	30	9/10/2004	16/11/04	0%
Tasca 4: Generació manual d'un aspecte per la classe/contracte PilaVector.	30	16/11/2004	24/11/2004	0%
Tasca 5: Anàlisi i Disseny del sistema. Descripció de l'arquitectura.	40	24/11/2004	30/11/2004	0%
Tasca 6: Prototipus creació automàtica d'un aspecte concret.	30	30/11/2004	07/12/2004	0%
Tasca 7: Creació de un prototipus genèric. • *Parseig dels tags de contracte per adaptar-los a la sintaxis del aspecte.*	55	07/12/2004	20/12/2004	40%
Tasca 8: Ampliació de les funcionalitats. • *Afegir la funcionalitat Old.*	30	20/11/2004	23/12/2004	20%
Finalització de la documentació i elaboració de la memòria.	40	23/11/2004	27/12/2004	
Total Hores	**305**			

Capítol 2 .Conceptes Bàsics

En el següent capítol descriurem en què es basen les diferents tecnologies que intervenen de forma directa en la realització d'aquest projecte. Analitzarem breument el Disseny per Contractes centrant-nos en la seva sintaxis. A continuació estudiarem la tecnologia Doclet de Java i com aquesta ens pot permetre parsejar classes per tal d'obtenir la informació de la pròpia classe i dels tags especials. Finalment, examinarem el concepte de AOP, farem una breu explicació d'aquest nou paradigma i investigarem una de les seves implementacións, el JAspect.

2.1 Disseny per contractes.

Diem que un software és fiable si és: correcte (fa allò que especifica que fa, ni més ni menys) i robust (reacciona adequadament davant situacions inesperades, no especificades).

El Disseny per Contracte es un tècnica d'enginyeria del software que ens ajuda a construir software correcte mitjançant l'ús d'una especificació. Aquesta especificació pot permetre minimitzar la redundància deguda a les comprovacions de correctessa de les dades o els estats d'execució. L'especificació, no és res més que un acord formal, el contracte, que regula la relació entre una classe/mètode i els seus clients (altres classes/mètodes), és a dir, estableix quins són els drets i obligacions de cada part. Aquest drets i obligacions es el que anomenarem pre i post condicions. Així doncs, les especificacions a les que ens referim es situen entre el disseny detallat i la codificació.

2.1.1 Parts principals d'un contracte.

Les parts principals d'un contracte són:

- Les pre-condicions, que a d'assegurar al client d'una funció abans de fer la crida.

- Les post-condicions que les funcions cridades han d'assegurar al client que les crida.

- Les condicions invariants que una classe ha de garantir sobre si mateixa abans i després de l'execució de qualsevol dels seus mètodes. De fet l'invariant d'una classe és pre-condició i post-condició de tots els seus mètodes.

Podem interpretar doncs, un contracte de la següent manera:

– Si les pre-condicions d'una funció es compleixen al moment de ser cridada, llavors la funció garanteix al client que, en acabar la crida, complirà les post-condicions establertes i, pel cas d'una funció mètode d'una classe, l'invariant de la classe es mantindrà cert .

Veiem a continuació un exemple de contracte aplicat al mon real, per tal de facilitar la seva comprensió.

Contracte de viatge	Obligacions	Drets (beneficis)
Passatger (client)	• Ésser a l'aeroport 1 hora abans de la sortida del vol. • Portar equipatge per sota del pes màxim permès. • Portar el bitllet.	• Volar de l'origen al destí. • Recuperar l'equipatge.
Companyia aèria (proveïdor)	• Portar el client al seu destí. • Portar el seu equipatge també.	• No cal que admeti a l'avió un passatger que: – no té bitllet, o – ha arribat tard. • No cal que admeti equipatge per sobre del pes màxim.

El mateix concepte es por aplicar a una classe que implementa una pila de valors. El següent quadre especifica el contracte per el mètode empilar().

Pila::empilar()	Obligacions	Drets (beneficis)
Client	• Només es pot cridar si la pila no és buida.	•Obtenir l'element del "top". • Actualització de la pila.
Classe Pila (proveïdor)	• Retornar l'element de dalt de tot. • Actualitzar la pila : – treure l'element de dalt de tot.	• La funció és més simple, ja que pot assumir que la pila no està buida. **-més fàcil de provar.** **-menys probabilitat d'error.**

2.1.2 Principi de no redundància.

L'objectiu que persegueix el disseny per contractes és minimitzar els errors i facilitar les tasques de comprovació, això és el que anomenem principi de no redundància: més fiabilitat comprovant menys. El cos d'una funció mai no ha de comprovar les pre-post condicions, sassumeix que són certes.

Fins ara per evitar els errors aplicàvem una programació defensiva:

- Cada funció "es protegeix" a sí mateixa tant com pot.
- Millor comprovar de més que de menys.
- Les comprovacions redundants potser no ajudaran més, però no "faran mal".
- En tractar amb estranys, totes les precaucions són poques.

```
Element Pila::Pop()
{
 if(Empty())
 {
      // gestió de l'error
 }
 else
 {
   --m_count;
   return   m_elements[m_count-1];
 }
}
```

El Disseny per Contractes canvia el paradigma anterior. És simplifica el codi, és més fàcil de mantenir, menys errors, més velocitat i menys inconsistències.

```
@ pre !Empty()
Element Pila::Pop()
{
   --m_count;
   return m_elements[m_count-
1];
}
```

2.1.3 Verificació del contracte.

Ens queda pendent la qüestió de com verificar en temps d'execució el contracte:

- Per una banda ho hem de fer sense acabar en programació defensiva.
- Però, d'altra banda ens hem de protegir contra violacions del contracte que, per exemple, es poden donar per error (sobretot en les primeres versions del codi). A més hem de notificar que el contracte no es compleix.

La majoria de llenguatges de programació no suporten directament el concepte de contracte (alguns com l'Eiffel sí ho fan).

Alguns llenguatges proporcionen mecanismes que ens poden servir per "codificar" les restriccions del contracte. Per exemple, en C i C++ podem utilitzar el mecanisme d'assercions (#include <assert.h>): assert(restriccio), on la restricció és una expressió que s'avalua a cert o fals. En cas de ser falsa apareix un missatge d'error que informa de la violació de l'asserció: programa, fitxer, línia del fitxer on està l'asserció i la restricció violada. Que el programa avorti o no dependrà de què implica per a la resta de codi que s'hagi violat la restricció.

2.1.4 Resum.

- Tothom vol fer codi fiable (correcte i robust) i, per tant, es tendeix a fer programació defensiva (protegir les funcions que es fan del seu ús inadequat).

- La programació defensiva dóna lloc a molta redundància de comprovacions i no explicita quines són les entrades bones que espera una funció i quines de les que es protegeix (s'ha d'endevinar a partir de codi tipus if-then-else on tot està mesclat).

- El disseny per contracte, desenvolupat principalment al context de l'OO però aplicable a d'altres paradigmes, proposa establir contractes entre funcions client i funcions proveïdores que permetin minimitzar la sobre-carrega de comprovacions i fer codi correcte. Això farà que els mètodes proveïdor tendeixen a ser més simples i fàcils de provar.

- Aquests contractes es basen en pre- i post-condicions, i en invariants, que poden ser expressats en llenguatge natural : el proveïdor assumeix que la pre-condició es compleix, i el client que es compleix la post-condició, el proveïdor assumeix que l'invariant de la classe propietària del mètode és cert a l'entrada a la funció i s'encarrega de mantenir-ho a cert després de la seva execució.

- Un gran problema és que la majoria de llenguatges de programació no suporten el concepte de contracte.

2.2 Programació Orientada a Aspectes (AOP)

2.2.1 Introducció a la AOP

L'enginyeria del programari ha experimentat importants avenços paral·lelament amb el desenvolupament de noves arquitectures i l'evolució dels llenguatges de programació.

Durant l'evolució de les metodologies de programació l'objectiu ha estat sempre aconseguir una divisió clara dels problemes que calia tractar en el desenvolupament d'una aplicació. Així hem pogut veure com passàvem de programes amb milers de línies escrits en codi màquina a llenguatges de més alt nivell que permetien la descomposició de problemes en petits procediments estructurats.

A mesura que la complexitat de les aplicacions ha augmentat, també ho ha fet la necessitat de noves tècniques. La Programació Orientada a Objectes (POO) ha pogut resoldre en part aquestes necessitats proporcionant estructures de dades amb un comportament propi que permet modelar objectes de la vida real que poden col·laborar entre ells.

Tot i així aquest paradigma no dóna una solució a un tipus de problema que ens trobem habitualment en el desenvolupament del software actual, on podem trobar sovint el mateix comportament afectant a diferents objectes que es troben en diferents mòduls i que no tenen res més a veure entre ells.

La Programació Orientada a Aspectes (AOP) sorgeix l'any 1996 en Xerox PARC, dins el grup liderat per Gregor Kiczales.

Aquest nou paradigma "presentat com una extensió de la OO" intenta fer possible la separació de comportaments, dins de l'AOP, mitjançant la construcció de noves estructures que s'anomenaran aspectes.

Els principals avantatges que aporta l'AOP són:

- Permet una disseny modular més pur al capturar de forma explícita els interessos transversals en un component que anomenarem aspectes.
- Permet una fàcil evolució. La modificació en la implementació d'aquests aspectes no afecta la codificació de l'aplicació principal i les modificacions en les aplicacions no impliquen necessàriament la reimplementació dels aspectes.
- Reciclatge de les classes. Amb aquesta filosofia, tant les classes com els aspectes poden ser fàcilment reciclats per altres aplicacions. Les classes no implementen comportaments aliens al seu concepte i per tant les podem portar a qualsevol aplicació que les requereixi i els *aspectes* poden incorporar comportaments totalment funcionals en sistemes heterogenis.
- Per últim, podem destacar que la separació d'interessos en mòduls ens permet carregar i descarregar aquestes funcionalitats de forma dinàmica sense haver de modificar cap línia de codi.

L'AOP no pretén substituir la POO sinó més aviat completar-la. L'AOP pretén la implementació d'interessos transversals que puguin ser aplicats a un conjunt ben diferenciat de models amb objectes.

Mentre que la POO representa un desenvolupament de les aplicacions jeràrquic, de dalt a baix, la AOP representa un desenvolupament ortogonal, de dreta a esquerra, que permet que aquests dos paradigmes s'acoblin perfectament.

Els camps en els quals s'aplica aquesta tecnologia actualment són el disseny de components Reciclables, *la implementació de patrons*, la Reenginyeria i molt especialment el disseny de components middleware i aplicacions distribuïdes.

2.2.2 Implementacions de l'AOP

En l'actualitat disposem d'un conjunt força variat d'eines que ens permeten el desenvolupament d'aplicacions que utilitzen l'AOP.

La diferència principal entre aquestes eines radica bàsicament en la plataforma per a la que han estat dissenyades. Tenint per exemple:

- **AspectC++**: una extensió de C++ per a l'AOP
- **AspectJ** i **AspectWerkz**, extensions de Java per a l'AOP.
- **JAC** que ofereix una solució middleware per a sistemes distribuïts.
- **JBOSS**, que ofereix una implementació per l'implantació de l'AOP en aplicacions funcionant sobre el servidor d'aplicacions JBOSS.

De entre totes aquestes eines una de les més utilitzades avui en dia és el ASPECTJ . La raó principal és que funciona sobre una plataforma Java, per a la qual avui en dia s'estan desenvolupant milers de solucions. L' AspectJ és una extensió natural de Java que podem aplicar en qualsevol entorn Java. El fet de ser una extensió natural ha permès la seva ràpida extensió.

2.3 Introducció a ASPECTJ

AspectJ és una extensió senzilla de JAVA, creada al 1998 en Xerox PARC, que implementa l'AOP.

El codi executable obtingut amb AspectJ és totalment compatible amb el de Java i per tant permet heretar els avantatges de Java, la possibilitat de ser utilitzat amb qualsevol programa Java i executat sense problemes en qualsevol JVM.

El gran nombre de desenvolupaments i arquitectures que utilitzen actualment el llenguatge Java i el fet que AspectJ sigui una extensió de Java fa que l'aprenentatge d'aquest nou paradigma sigui senzill per molts programadors experimentats en aquest llenguatge i per tant que s'estengui ràpidament la seva utilització.

Aquesta senzilla extensió afegeix nous conceptes al llenguatge Java, el de joinpoint, i un conjunt de noves construccions: pointcuts, advice, inter-type declarations i aspects.

Un joinpoint és un punt ben definit en el flux del programa, i un pointcut defineix un o varis joinpoints. Un advice defineix el codi que cal executar quan durant l'execució d'un programa ens trobem en les condicions definides per un pointcut.

Aquestes noves construccions afecten dinàmicament l'aplicació, afegint nous comportaments en temps d'execució depenent de diferents condicions en el context de l'execució

Un aspecte el podem definir com el contenidor de tots aquests nous conceptes. Un aspecte implementa un interès transversal, defineix els punts d'interès dins un programa on volem afegir nous comportaments, ens permet codificar el nou comportament mitjançant l'ús dels elements abans esmentats (joinpoint, pointcuts, advices, ...), i ens proporciona accés a la informació referent al context en que s'està executant l'aplicació, tan pel que fa al punt d'execució com a les classes a les quals es té accés, les seves propietats i els seus mètodes.

Podem dir doncs, que la utilització d'AspectJ permet una implementació modular, clara i senzilla dels interessos transversals.

Aquesta propietat que permet la implementació d'interesos transversals , és el que ens permetrà l'ús del AspectJ per tal de construir el nostre prototipus, en el qual hem utilitzat la capacitat de modificar de forma dinàmica el comportament de qualsevol aplicació Java per tal d'executar la informació associada als contractes sense tenir que modificar el codi de les classes base.

2.2.3 Breu especificació del llenguatge i definicions

Aquest apartat pretén mostrar una breu especificació del llenguatge AspectJ que inclogui aquelles parts que hem utilitzat en el nostre treball.

Joinpoint

Un joinpoint és un punt ben definit en l'execució del programa. Aquests inclouen les crides a mètodes, crides a constructors, accessos a camps de les classes i molts altres. Els joinpoints que hem utilitzat principalment són:

Crida a un mètode (method call): indica un punt on es crida a un mètode sense incloure el mètode constructor de la superclasse, *super()*.
Execució d'un mètode (method execution): punt just després de la crida a un mètode, en el moment en que es comença a executar.
Crida a un constructor (constructor call): Punt en que es cridat el constructor de qualsevol objecte, excepte super() i this().

Pointcut

Un pointcut és la definició d'aquells joinpoints que volem capturar. Aquest són utilitzats pels advice() per assignar codi a una determinat punt de l'aplicació, que vindrà definit per un o una combinació de pointcuts. El que hem utilitzat principalment en el nostre treball són:

- **call(..)**: Permet definir el format de la crida o constructor que ha de complir el joinpoint que volem capturar. Això vol dir, el tipus de mètode (privat, public, protected, etc), el tipus de valor que retorna la crida, el paquet i la classe al qual pertany i fins i tot el tipus de paràmetres.

Exemples	Descripció
public void Account.*()	Tots el mètodes públics de la classe *Account* que retornen *void* i no tenen arguments
public * Account.*(..)	Mètodes públics de la classe *Account* sigui el que sigui que retornin i amb qualsevol tipus i nombre d'arguments.
public Account+.new(..)	Qualsevol constructor públic sobre la classe *Account* o les seves subclasses.

- **execution(..)**: Identifica el format del mètode o constructor en execució que volem capturar. Igual que l'anterior però en aquest cas l'*advice* s'executa un cop iniciat el mètode, a diferència de **call** que s'executa abans de l'execució del mètode.

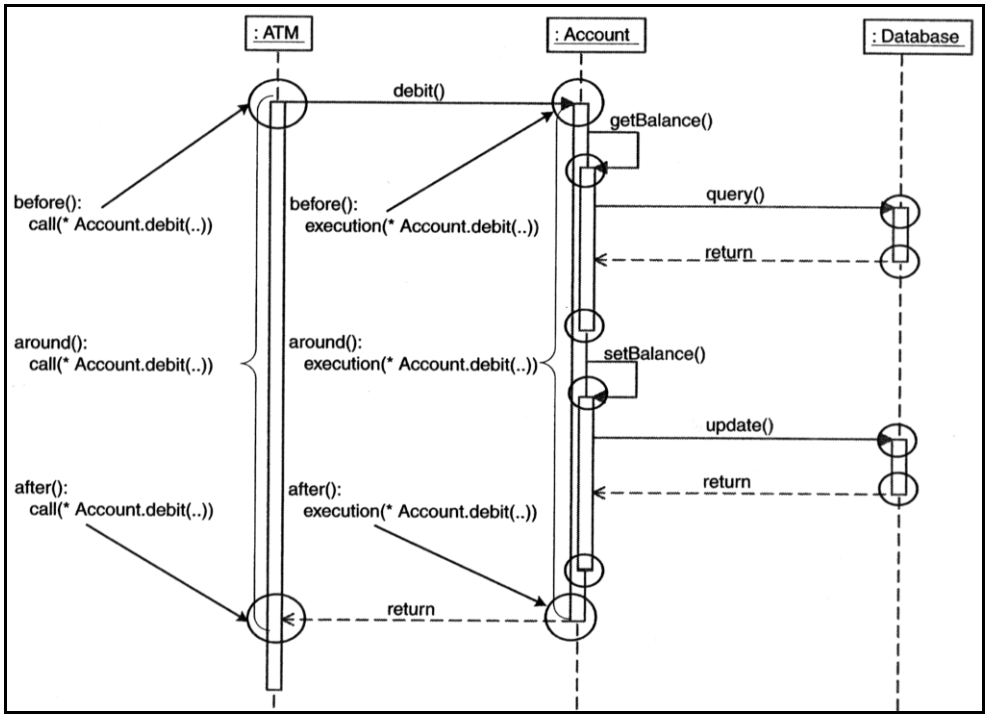
Advice

Un advice defineix el comportament que volem incorporar davant un pointcut. AspectJ suporta tres tipus de advice:

- **before()**: Definició del codi que volem executar abans de l'execució del codi de l'aplicació definit pel pointcut.
- **after()**: Definició del codi que volem executar un cop finalitza l'execució del joinpoint definit pel poincut. Hi ha diferents codis de retorn que l'**after** pot tractar:
    - **returning**: Sempre i quan torni correctament de l'execució del mètode. Es preferible fer-lo servi per defecte.
    - **thowing**: Que ens permet captura si s'ha produït una excepció i el tipus d'execepció.

.

El següent diagrama mostra l'exemple de una aplicació el moment en que s'executarà cada tipus de *advice*.



Aspect

L'Aspect és l'estructura de dades que utilitza AspectJ per definir un nou comportament o un interès transversal. Podem definir propietats i mètodes propis, així com d'altres que ens permeten l'accés a propietats i mètodes privats d'altres classes. Un aspecte també pot estendre classes i interfícies i pot estendre altres aspectes sempre i quan aquests siguin abstractes.

Un aspecte no pot ser instanciat directament amb un new. Els aspectes són creats automàticament per la JVM en el moment en que s'executa l'aplicació i és disposa d'una única instància que afectarà a tot el programa.

2.4 JavaDoc : Doclets Java.

Dintre de la suite de java SDK trobem l'aplicació JavaDoc, que bàsicament és una eina de documentació. El JavaDoc permet generar documentació API en format HTML per un paquet concret o per fitxers fonts individuals. Genera un fitxer .html per cada fitxer .java, on es troba la seva informació API, també genera la jerarquia de classes i un índex amb tots els membres que ha trobat. Per tal que es generi aquesta informació la classe base es parsejada reconeixent els tags de documentació standard i que permeten generar la documentació. Aquest s'efectua mitjançant un Doclet.

Un Doclet és una classe escrita en Java utilitzada pel JavaDoc, on s'especifiquen: els tags que es reconeixeran durant el parseig i com es generarà la sortida. Tot i que el JavaDoc proporciona un Doclet per defecte, que és el que permet generar l'API, es poden escriure Doclets per generar qualsevol tipus de sortida i que reconeguin qualsevol tipus de tags.

2.4.1 Funcionament del JavaDoc.

El JavaDoc comprova automàticament les classes, interfícies, mètodes i declaracions de variables. Habitualment els programes Java incorporen comentaris dintre d'unes marques: **/**** (inici de comentari) i ****/** (fi de comentari), aquestes marques permeten al Doclet per defecte de JavaDoc generar la documentació api.

JavaDoc permet reconèixer altres marques especials que permeten completar la generació de la documentació API de les font Java. Les marques comencen sempre amb el signe @. Aquestes marques han d'estar incloses dintre d'un bloc comentari.

Algunes de les marques per defecte que implementa el JavaDoc són les següents:

- Marques de la documentació de classes e interfícies:
    @ see <nom de la classe>
    @ version <versió del text>
    @ author <autor del text>

- Marques de la documentació dels mètodes:
    @ params <nom paràmetre>
    @ return <tipus de retorn>

Per tal de dur a terme la generació de la documentació, la classe Doclet implementa un conjunt de funcionalitats que permeten recórrer les classes font per tal de parsejar els seus comentaris. Aquests conjunt de funcionalitats permeten "navegar" per la classe font com si d'un XML es tractes.

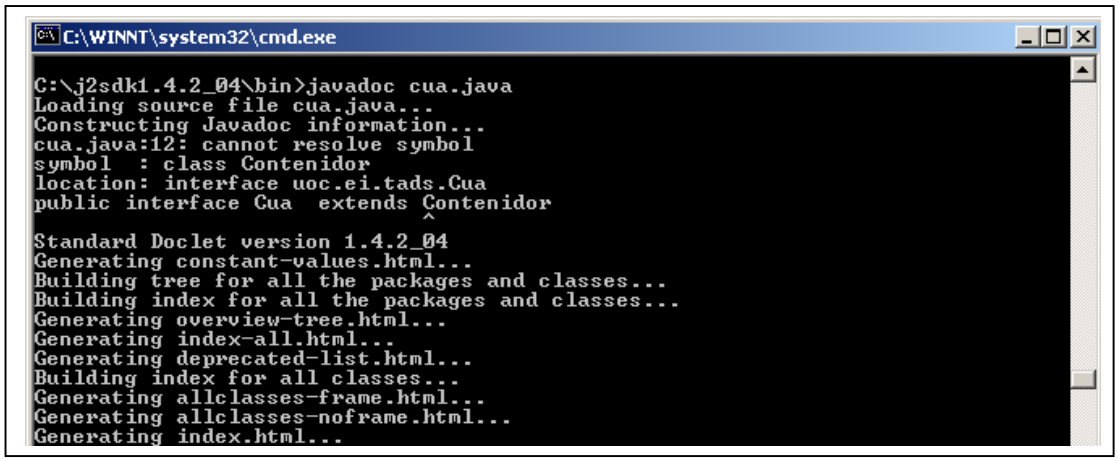
2.4.2 Doclet.

Un Doclet és una classe Java que permet customitzar l'execució del JavaDoc. Amb aquesta customització podem parsejar les classes Java per tal d'obtenir informació especifica de la classe: nom de mètodes, paràmetres, variables etc. A més també podem obtenir la informació de tots els tags de comentari que tinguin el caràcter especial @. Aquestes característiques fan adient aquesta tecnologia pel nostre projecte. Utilitzarem les marques especials: @pre, @post i @invariant per tal d'obtenir la informació dels contractes.

2.4.2.1 Instanciació d'un Doclet.

Com em dit un Doclet és una classe Java que pertany al package com.sun.javadoc. Aquesta classe s'instacia durant l'execució de la aplicació JavaDoc.

L'aplicació JavaDoc pot ser executada des de la línia de comandes:



o bé es pot executar una classe JavaDoc dins d'un programa Java

```
public class ExecutionJavaDoc
{
  String[] javadocargs = { "-doclet",
                           "aspectGen.generation.DocletAspectes",
                           "-docletpath",
                           "D:\\uoc\\project\\aspectGen\\generation",
                           "-classpath",
                           "c:\\j2sdk1.4.2_04\\bin\\tools.jar",
                           "-sourcepath",
                           "d:\\uoc\\project",
                           "-package",
                           "uoc.ei.tads"
                         };

  public static void main(String[] args)
  {
     com.sun.tools.javadoc.Main.execute(javadocargs);
  }
}
```

Un cop s'inicia l'execució de l'aplicació JavaDoc, aquesta instància el Doclet, en el cas que no s'hagi especificat cap Doclet, s'instancia el Doclet per defecte que permet generar la documentació standard.

2.4.2.2 Classe Doclet.

La classe Doclet és una classe abstracta. Per tal de implementar un Doclet haurem d'implementar la funció Start(), que s'instancia al executar el JavaDoc:

```
public class DocletAspectes
{
    public static boolean start(RootDoc root)
    {
       /** Root: objecte que permet accedir a la informació de la
       clase font */
      .....
    }
}
```

La classe RootDoc contindrà informació de les classes java parsejades durant l'execució de JavaDoc. Mitjançant aquesta classe i els seus mètodes es pot accedir a tota la informació de les classes parsejades:

ClassDoc[]	classes() Retorna les classes parsejades.

La classe ClassDoc ens permetrà obtenir informació concreta d'una classe:

ConstructorDoc[]	constructors() Retorna els constructors de la classe.
FieldDoc[]	fields() Retorna les variables de la classe.
MethodDoc[]	methods() Retorna els mètodes d'una classe.
ClassDoc[]	importedClasses() Retorna una llista dels imports de la classe base.

2.5 Implementacions disponibles pel Disseny per contractes.

En l'actualitat i hi ha diversos productes que permeten l' ús de Disseny per Contractes. A continuació passarem a analitzar algunes de les implementacions disponibles en l'entorn Java i les compararem amb la nostra solució:

- **iContract:**

  - Extensió del doclet de Sun per tal d'expressar precondicions, postcondicions i invariants d'una classe.
  - Permet l'ús de funcionalitats com Forall() and Exists() dintre de l'especificació d'una condició.
  - En l'actualitat només està disponible la seva especificació encara que comencen a aparèixer les primeres implementacions.
  - Afegeix el codi de les condicions dintre de les classes base durant el temps d'execució.

  Conclusió:
  La nostra implementació segueix el mateix principi, utilitzat un doclet per tal d'obtenir la informació del contracte. La diferencia es que nosaltres hem optat per utilitzar Aspectes per la implementació del runtime mentre que iContract crear el codi de la comprovació durant l'execució.

- **dbcproxy – jspec:**

  - Extensió del doclet com iContract.
  - Funciona com un preprocessador. El preprocessador genera unes classes addicionals que s'han de compilar junt amb l'aplicació. Posteriorment un proxy comprova els contractes.
  - Esta implementat utilitzant Dynamic proxies de Java. Això obliga a treballar utilitzant proxies a l'hora de utilitzar les classes base en lloc de utilitzar-les directament i que es comprovin els contractes.

  Conclusió:
  Com hem vist el seu ús potser complicar-se pel fet d'utilitzar proxie. Respecte a la part del preprocessador, és similar al que el nostre prototipus farà l'eina de generació d'Aspectes. L'avantatge del nostre prototipus és que un cop creats els aspectes es poden utilitzar les classes base directament.

- **Jcontractor:**
  - Codi obert.
  - No hi ha precompilació.
  - Les pre/post condicions i els invariants es programen com a mètodes java. Aquesta programació es pot fer a la pròpia classe que especifica el contracte com en una altre classe.

  Conclusió:
  És una implementació que de fet s'acosta molt a la programació defensiva. A més és el propi usuari qui ha d'implementar el codi.

- **JMSAssert:**
  - Es una implementació propietaria.
  - Utilitza la filosofia del iContract, d'utilitzar doclet.
  - Igual que el dbcProxy fa un preprocessament generant fitxer que s'utilitzen després en temps d'execució.
  - Per tal de comprovar els contractes s'ha de executar dintre d'un entorn propi.

  Conclusió:
  És una implementació propietària i per tant de pagament. Un altre desavantatge és que necessita crear un entorn d'execució.

Capítol 3. Anàlisi

En aquest capítol explicarem com aplicarem la Programació Orientada a Aspectes per la implementació del Disseny per Contractes. Analitzarem el mapeig entre els tags de contracte: pre, post i invariant amb els tipus d'advices: before i after. Finalment farem la especificació dels casos d'ús del nostre projecte.

3.1 Aplicació d'Aspectes per la implementació de contractes.

La idea bàsica de la que partim és que disposem d'una llibreria de TADs (classes Java) de l'assignatura d'Estructura de la Informació a la que es vol afegir les funcionalitats propies del Disseny per Contractes. Un dels objectius és no modificar cap classe de la llibreria, únicament afegir a aquestes classes les informacions pròpies dels contractes. Una de les funcionalitats de la tecnologia orientada a aspectes ens permetrà complir aquest objectiu: ens permetrà estendre les funcionalitats de les classes TAD mitjançant Pointcuts per tal de comprovar els contractes transparentment. És a dir, utilitzarem aspectes per tal de convertir la informació associada als contractes en Joinpoints i Advices.

Per tal de crear els aspectes associats a cada classe aplicarem els següents criteris:

- Per cada classe base es crearà un Aspecte que tindrà el mateix nom de la classe base més la cadena de caràcters "Aspect".

- Per cada Mètode/Constructor de la classe base que contingui l'especificació d'un contracte es generarà un Pointcut. El Pointcut tindrà el mateix nom que el mètode/constructor més la cadena de caràcters "PointCut".

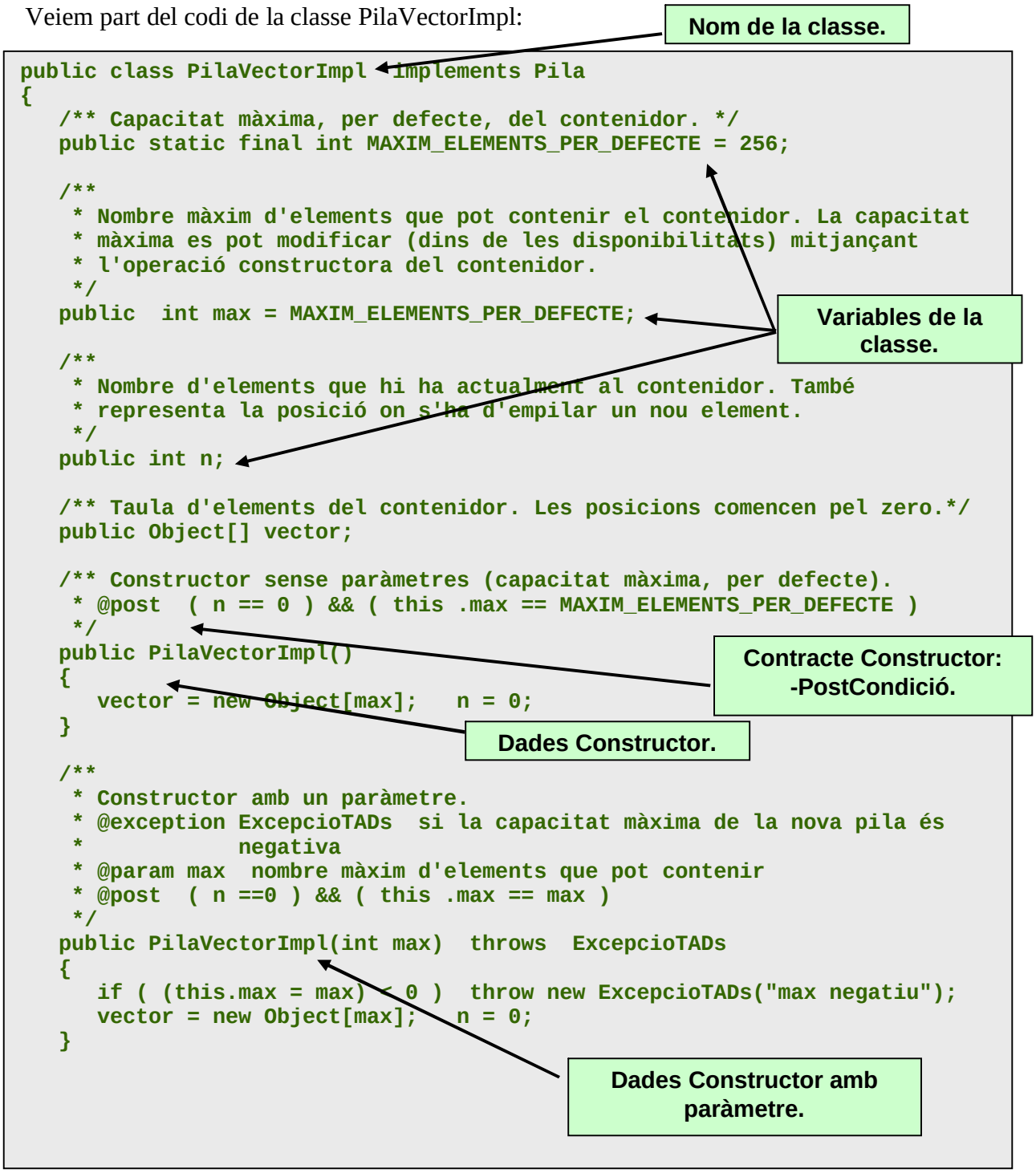
Respecte als Advices dels Pointcuts seguirem el següent mapeig:

Tag de Contracte a la classe Base	Tipus de Advice.
@pre	before():
@post	after():
@invariants	after() (Nivell de classe).

3.1.1 Exemple d'Aspecte Contracte.

Per tal de familiaritzar-nos amb la creació d'aspectes crearem manualment l'aspecte per el contracte de la classe PilaVectorImpl del package uoc.ei.tads de la llibreria TADs.
La creació d'aquest aspecte manualment ens proporcionarà un prototipus que aprofitarem per tal de conèixer la informació que el Doclet ens haurà de facilitar per tal de fer la creació automàtica d'aspectes.

Veiem part del codi de la classe PilaVectorImpl:




```
public class PilaVectorImpl implements Pila
{
   /** Capacitat màxima, per defecte, del contenidor. */
   public static final int MAXIM_ELEMENTS_PER_DEFECTE = 256;

   /**
    * Nombre màxim d'elements que pot contenir el contenidor. La capacitat
    * màxima es pot modificar (dins de les disponibilitats) mitjançant
    * l'operació constructora del contenidor.
    */
   public  int max = MAXIM_ELEMENTS_PER_DEFECTE;

   /**
    * Nombre d'elements que hi ha actualment al contenidor. També
    * representa la posició on s'ha d'empilar un nou element.
    */
   public int n;

   /** Taula d'elements del contenidor. Les posicions comencen pel zero.*/
   public Object[] vector;

   /** Constructor sense paràmetres (capacitat màxima, per defecte).
    * @post  ( n == 0 ) && ( this .max == MAXIM_ELEMENTS_PER_DEFECTE )
    */
   public PilaVectorImpl()
   {
      vector = new Object[max];   n = 0;
   }

   /**
    * Constructor amb un paràmetre.
    * @exception ExcepcioTADs  si la capacitat màxima de la nova pila és
    *            negativa
    * @param max  nombre màxim d'elements que pot contenir
    * @post  ( n ==0 ) && ( this .max == max )
    */
   public PilaVectorImpl(int max)  throws  ExcepcioTADs
   {
      if ( (this.max = max) < 0 )  throw new ExcepcioTADs("max negatiu");
      vector = new Object[max];   n = 0;
   }
```

Veiem a continuació com quedaria l'aspecte associat:

```
public privileged aspect PilaVectorImplAspect
{
  pointcut PilaVectorImpl_PointCut(): call (public PilaVectorImpl.new());

  after(PilaVectorImpl p): target(p) && PilaVectorImpl_PointCut()
  {
    if (!( p.n == 0  &&  p .max == p.MAXIM_ELEMENTS_PER_DEFECTE )
    {
      throw new ContractException(" Violació del contracte");
    }
  }

  pointcut PilaVectorImpl_PointCut(int max):call (public
                                    PilaVectorImpl.new( int)) && args( max);

  after(PilaVectorImpl p, int max): target(p) &&
                                    PilaVectorImpl_PointCut(int) && args (
                                                                    max)
  {
    if (! (p.n == 0  &&  p .max == max ))
    {
      throw new ContractException(" Violació del contracte");
    }
  }
```

3.2 Creació d'un Doclet per generar la informació de l'aspecte.

Veiem ara un bloc de codi que representa el contracte d'un mètode i com quedarà el PointCut associat:

Mètode: nom del mètode,parametres, modificadors (public, private..)

```
/**
* Afegeix un element a la pila, si hi cap.
* @exception ExcepcioContenidorPle  si la pila està plena
* @param elem  element que es vol afegir a la pila
* @pre  (! estaPle() )
* @post ( vector [ n -1] == elem )
*/
public void empilar(Object elem)  throws  ExcepcioContenidorPle
{
  if ( estaPle() )  throw  new ExcepcioContenidorPle();
     vector[n++] = elem;      // empila i incrementa el nombre d'elements
}
```

```
pointcut empilar_PointCut( java.lang.Object elem): execution (public void
                                PilaVectorImpl.empilar(..)) && args( elem);

before(PilaVectorImpl p,  java.lang.Object elem): target(p) &&
                         empilar_PointCut( java.lang.Object) && args ( elem)
{
    if (p.estaPle( ))
      throw new throw new ContractException(" Violació del contracte”);
}

after(PilaVectorImpl p, java.lang.Object elem): target(p)
                    &&empilar_PointCut( java.lang.Object) && args ( elem)
{
    if (!( p.vector [ p.n -1] == elem ))
      throw new throw new ContractException(" Violació del contracte”);
}
```

Amb aquest primer exemple ja em pogut veure quina informació de la classe base necessitarem per tal d'implementar l'aspecte:

Dades	Comentari
Nom classe Base	Donarà lloc al nom de l'aspecte.
Nom constructors/mètodes	Donarà lloc als Poincuts.
Constructors: ❑ Nom ❑ Paràmetres.	La sintaxis dels Pointcuts dels constructors i dels mètodes tindran una sintaxis diferent, per tant requeriran un tractament diferent.
Mètodes: ❑ Nom ❑ Paràmetres. ❑ Retorn. ❑ Modificadors àmbit(public, private...)	La sintaxis dels PointCuts dels constructors i dels mètodes tindran una sintaxis diferent, per tant requeriran un tractament diferent.
Tags contractes: @ pre <condició> @ post<condició>	Segons el tag del contracte es generarà l'advice. La condició de l'advice serà la negació de la condició del contracte.
<condició>	Les condicions s'hauran de modificar per tal de modificar les crides a variables o mètodes del propi objecte base, ja que s'haurà d'afegir davant d'aquestes crides el nom de la instància de l'objecte (veure target p, a l'exemple).

Com veiem al analitzar el quadre anterior els punts de mes risc seran:

- L'obtenció de total la informació dels mètodes i constructors que ens haurà de proporcionar el Doclet.

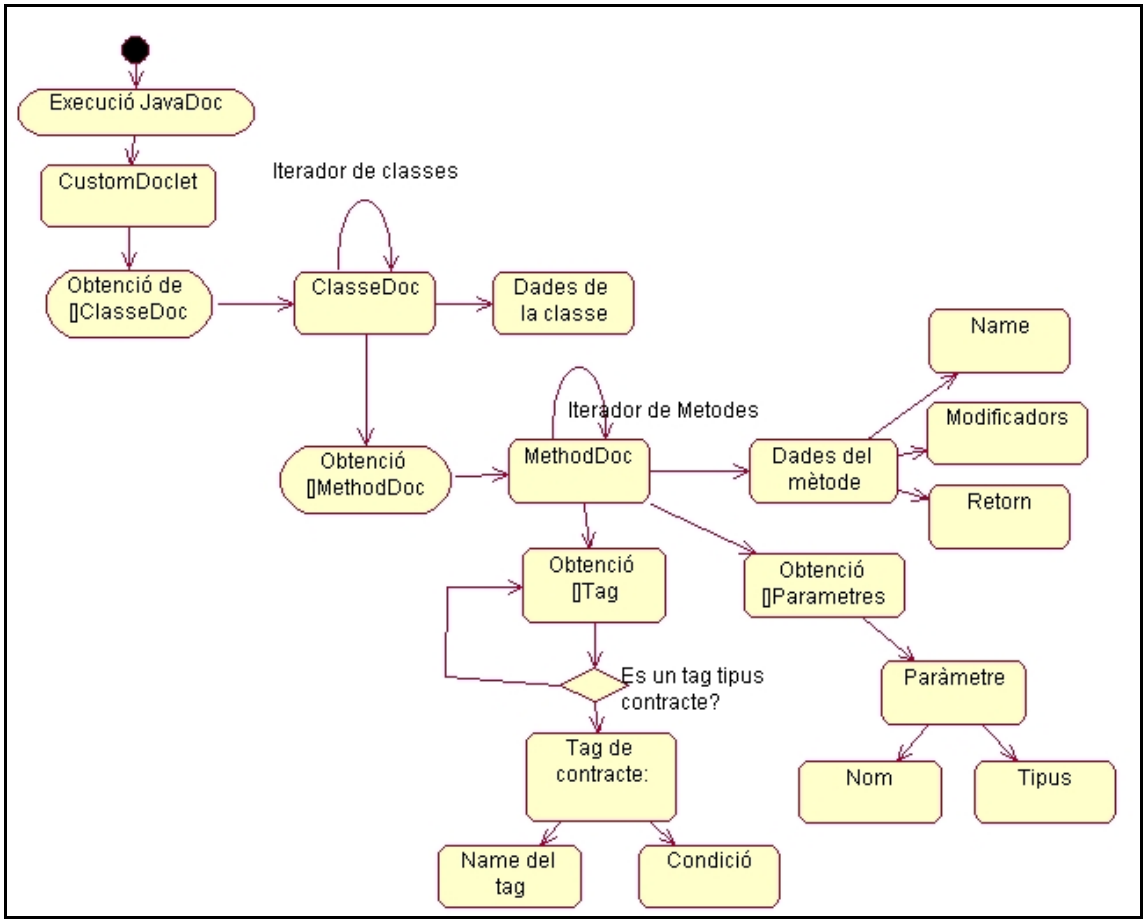
- El parseig de les condicions per tal d'afegir la informació de la instància.

3.3 Ús d'un Doclet per l'obtenció de les dades de la classe base.

JavaDoc quan inicia la seva execució instancia un Doclet. En el nostre cas el Doclet haurà de:

- Parsejar la informació de les classes per obtenir la informació dels mètodes, variables i altres informacions de les classes.
- Reconèixer els tags de tipus contracte.

L'API Doclet conté una jerarquia de classes que ens permet obtenir la informació d'una classe desde la informació general fins la informació particular. Veiem a continuació un esquema de com el nostre Doclet obtindrà aquesta informació:



3.4 El mètode OLD.

El mètode OLD a la definició d'un contracte permet la comparació d'un objecte abans i després d'executar un mètode.

Imaginem el següent escenari: tenim una classe Pila i volem implementar un contracte per assegurar que un cop s'executi el mètode empilar, el valor de la variable que determina el nombre d'elements de la pila, sigui n+1:

```
    /**
     * Afegeix un element a la pila, si hi cap.
     * @exception ExcepcioContenidorPle  si la pila està plena
     * @param elem  element que es vol afegir a la pila
     * @pre  (! estaPle() )
     * @post (n == $old(n)+1)
     */
    public void empilar(Object elem)  throws  ExcepcioContenidorPle
    {
            ......
```

El sistema haurà de permetre emmagatzemar el valor de n, per tal que quan acabi l'execució de la funció es pugui comprovar la post condició.
Per tal de dur a terme aquesta funcionalitat es crearà un objecte runtime, a mena de taula de símbols on s'emmagatzemaran els objectes OLD.

3.4.1 L'emmagatzemament de l'objecte.

Com hem vist a l'exemple anterior l'objecte OLD s'utilitza a la post condició, és a dir, quan el codi del mètode ja s'ha executat. Per tal de dur a terme la comparació de la condició del contracte, l'objecte s'haurà d'emmagatzemar abans de l'execució del mètode. Això vol dir que quan es detecti que un contracte utilitza la funcionalitat OLD, s'haurà d'afegir un Advice before al Pointcut del mètode per tal d'emmagatzemar aquest objecte. Per tant un Pointcut podrà tenir Advices before deguts a pre-condicions del contracte i deguts a l'ús de la funcionalitat OLD a les post-condicions.

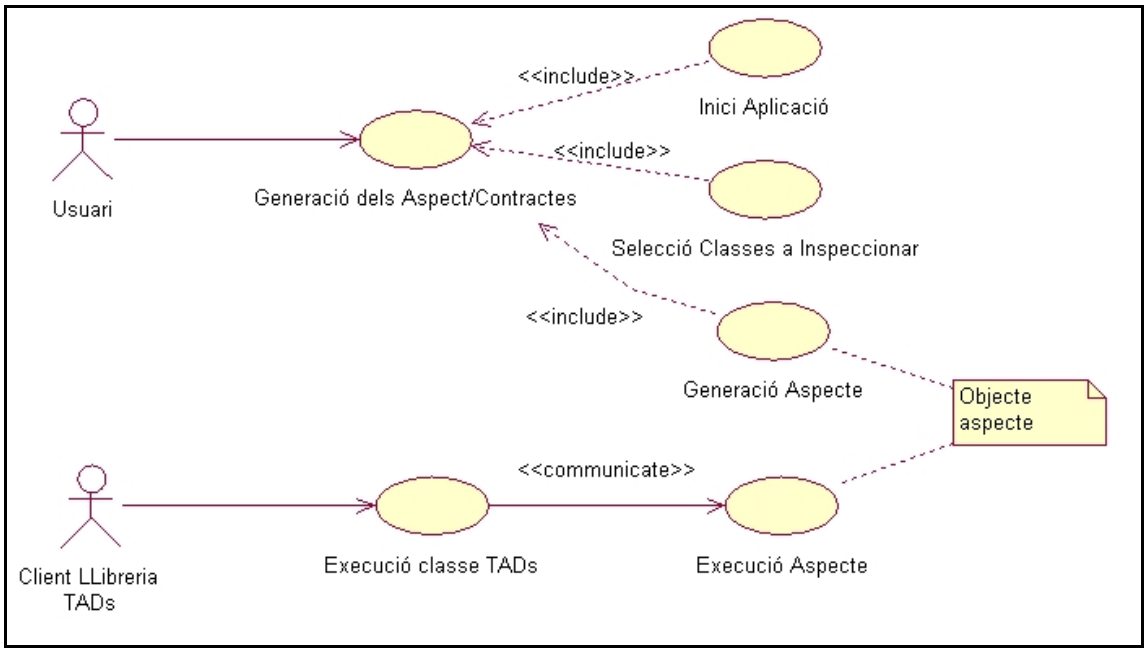
Seguin l'exemple anterior, tindríem el següent Pointcut:

```
pointcut empilar_PointCut():execution(public void
                                  PilaVectorImpl.empilar(..));

 before(PilaVectorImpl p): target(p) && empilar_PointCut( )
  {
    if (estaPle())
      throw new ContractException(" Contracte violat");
    $old(n) = p.n; // Enamagatzemem el valor n
  }
 after(PilaVectorImpl p): target(p) &&empilar_PointCut( java.lang.Object)
  {
    if (! p.n == $old(n) )
      throw new ContractException(" Contracte violat");
  }
```

3.5 Casos d'ús.

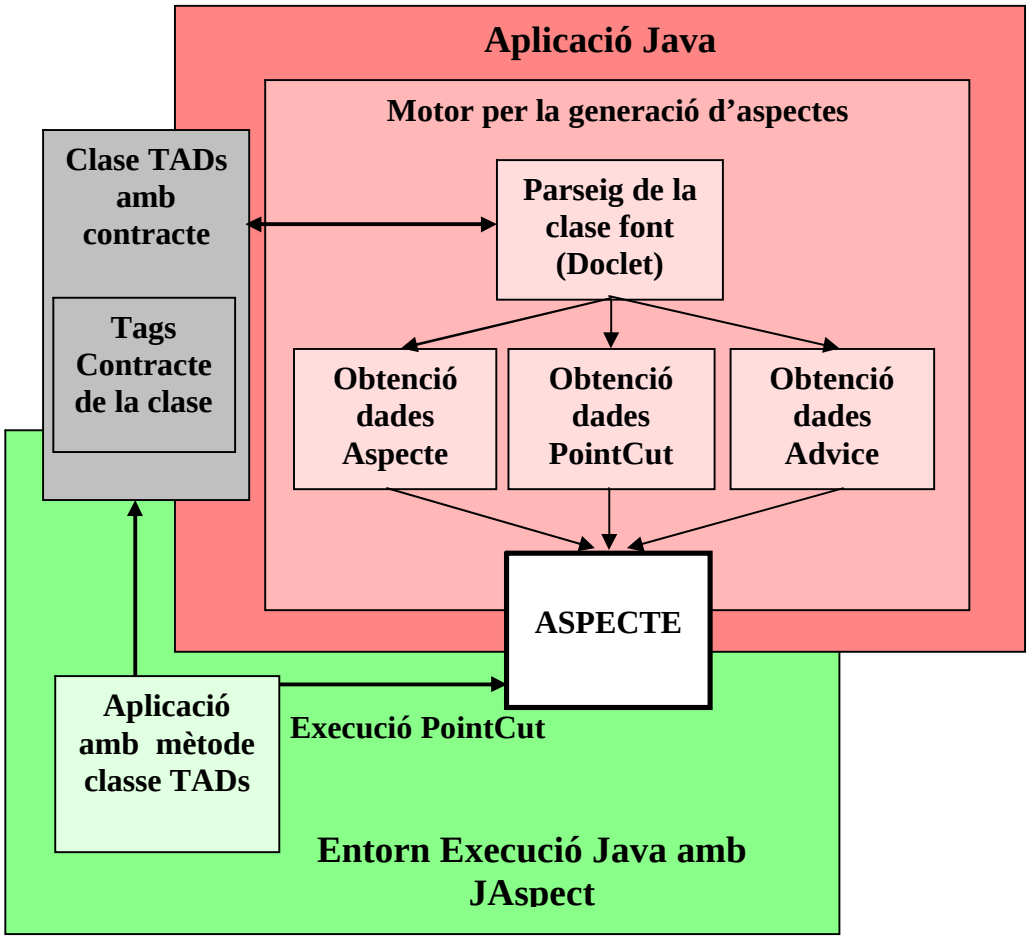
A continuació presentem el diagrama general de casos d'ús.

Diferenciem 2 casos d'ús: un per la generació dels aspectes i un altre per l'execució dels contractes , és a dir, per l'execució dels aspectes.



3.6 Arquitectura de l'eina

Com ja hem vist tenim dos entorns d'execució diferenciats: per una part la generació d'aspectes i per una altre part l'execució d'aquests aspectes. La següent figura representa com es relacionen cadascun dels components que intervenen en el nostre sistema.



Capítol 4. Disseny

En aquest capítol veurem el disseny del nostre sistema així com les diferents solucions que hem aplicat al prototipus.

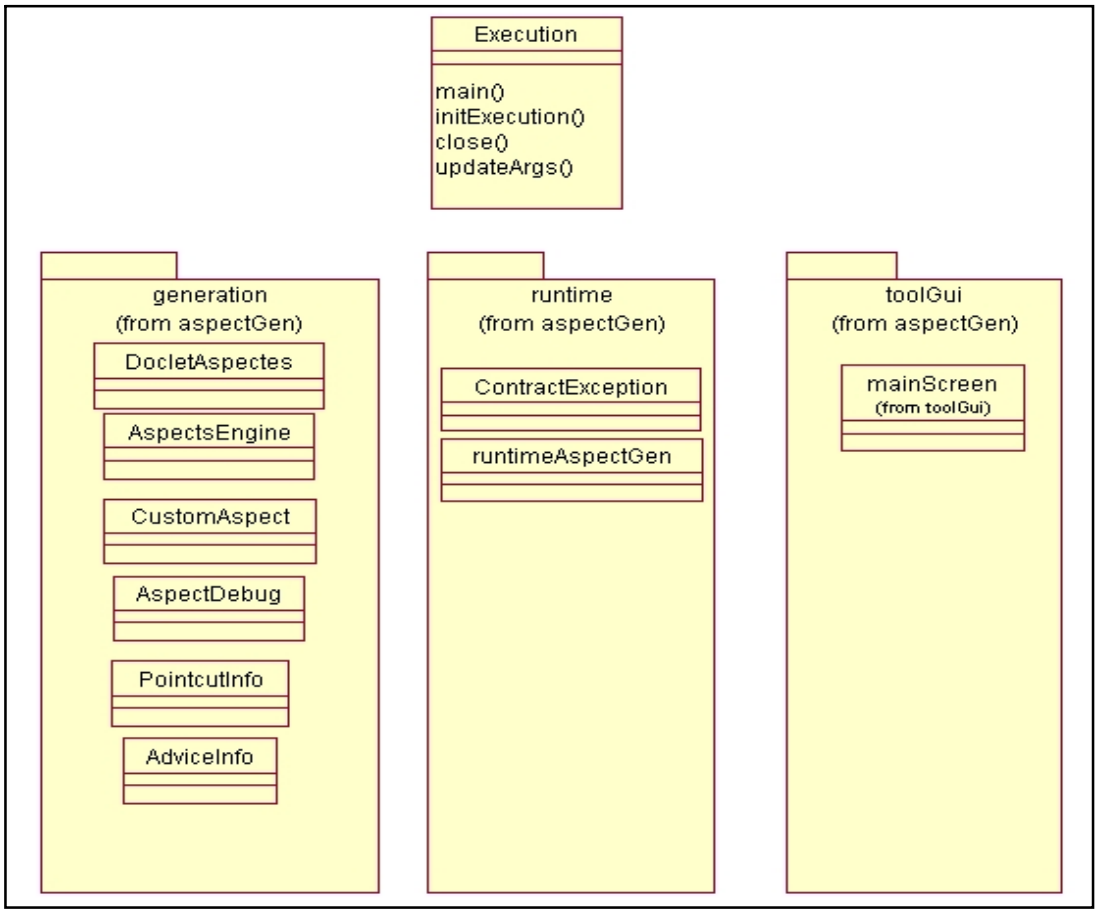
4.1 Arquitectura de paquets.

Diferenciem tres paquets principals, segons les funcionalitats que agrupen:

- Entorn GUI: Representa el mòdul de interfície de l'aplicació, contindrà totes les classes implicades en la generació d'elements de la interfície gràfica de l'aplicació.

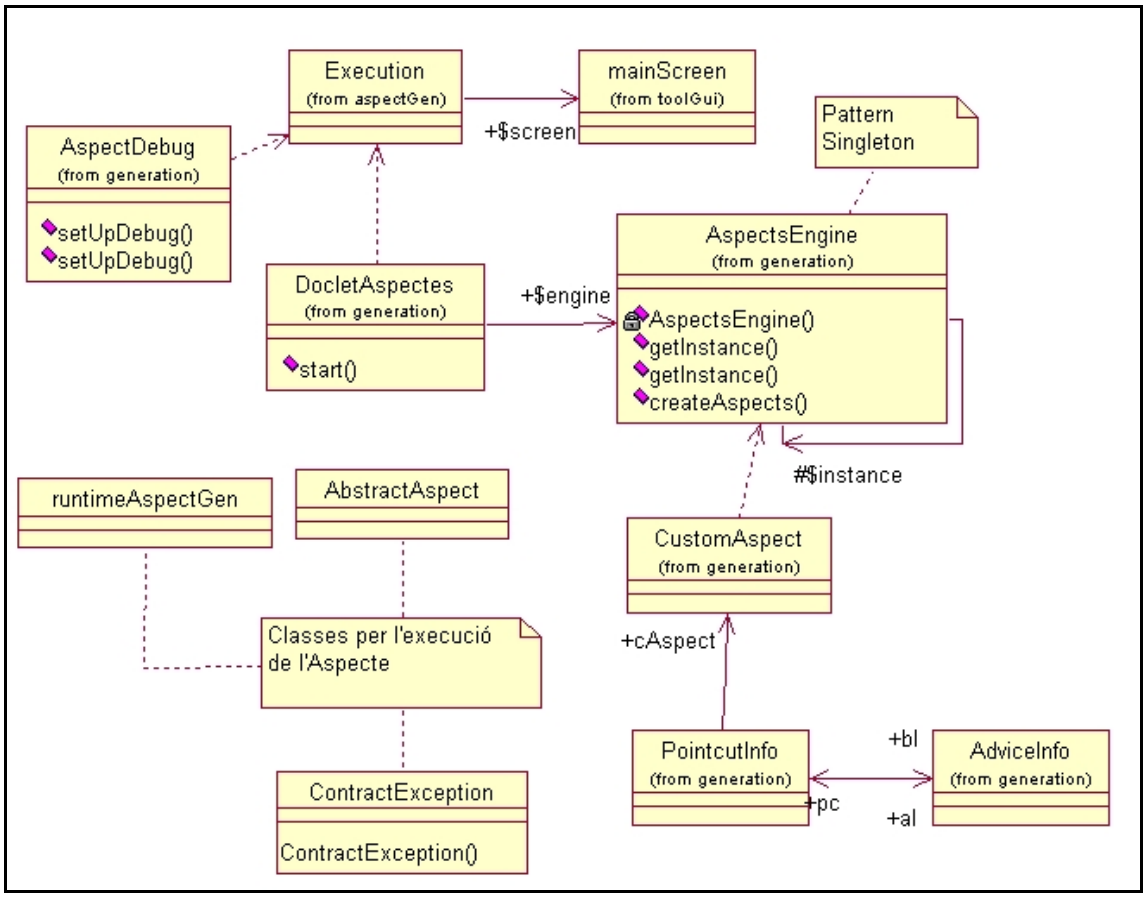
- Entorn de Generació: Contindrà totes les classes implicades en la generació dels l'aspectes en un fitxers persistents.

- Entorn de Execució o Runtime: Contindrà les classes implicades en l'execució d'un aspecte quan un contracte de la classe base s'hagi d'executar.



4.2 Diagrama de classes.

El següent diagrama de classes inclou totes les classes necessàries per l'execució de l'aplicació de generació d'Aspectes així com les classes implicades a l'execució dels aspectes.



Descripció de les classes principals:

- Execution: conté el entry point de l'aplicació (Main). Inicia l'execució del JavaDoc amb el Doclet -> DocletAspectes.

- DocletAspectes: és el Doclet encarregat d'obtenir la informació de la classe base.

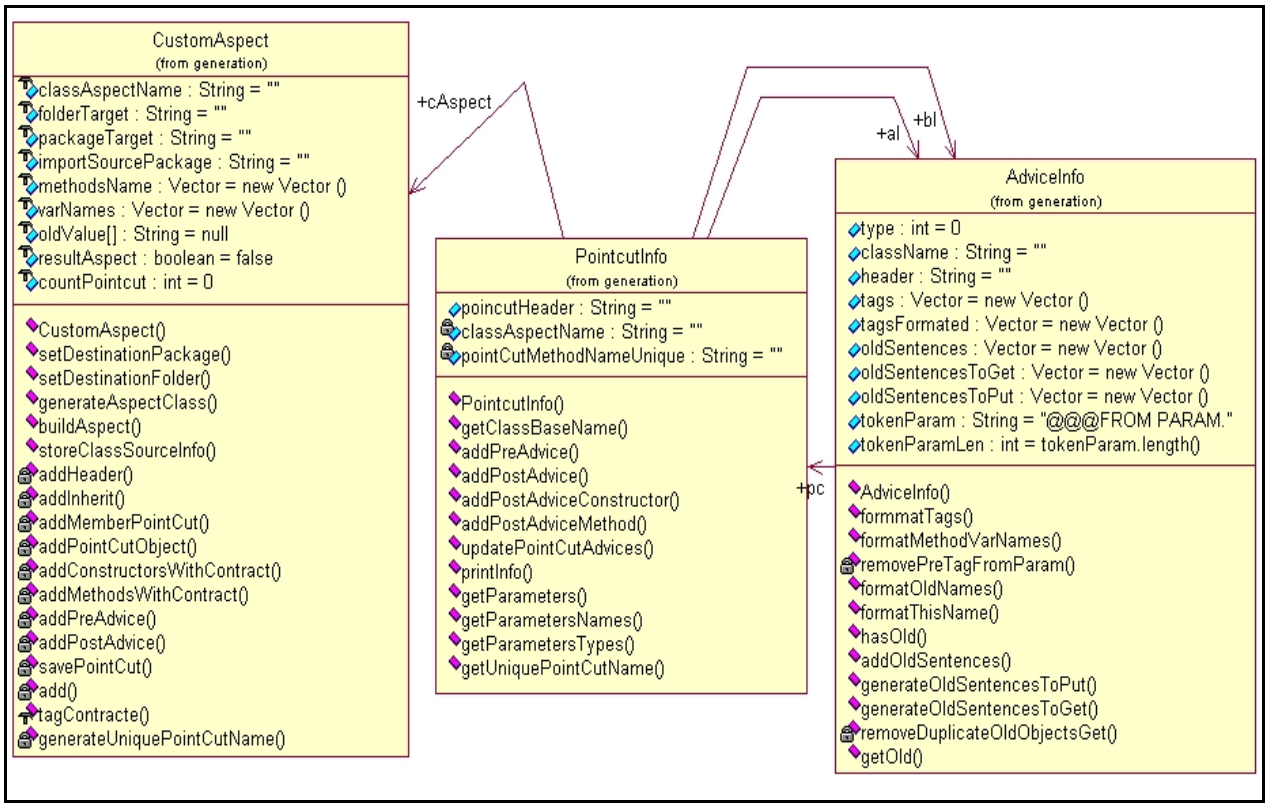
- AspectEngine: és la classe encarregada de parsejar la informació del Doclet i generar el CustomAspect. Al crear la instància de CustomAspect, aquest contindrà la informació de les variables, dels mètodes i de la informació general per crear l'aspecte.

- CustomAspect: centralitza la creació del aspecte. Per una part emmagatzema la informació general de l'aspecte i per l'altre s'encarrega de parsejar tots els mètodes i constructors de la classe base així com els tags de contractes. Amb aquesta informació generarà els Pointcuts adients.

- PointCutInfo: Conté la informació per genrar la capçalera d'un Pointcut. S'encarrega de generar els Advices adients. Instancia un Advice before i Advice after.

4.2.1 Diagrama de classes de la generació d'aspectes.

El següent diagrama es focalitza en les classes implicades en la generació d'aspectes, un cop el Doclet ja ha parsejat la classe base.



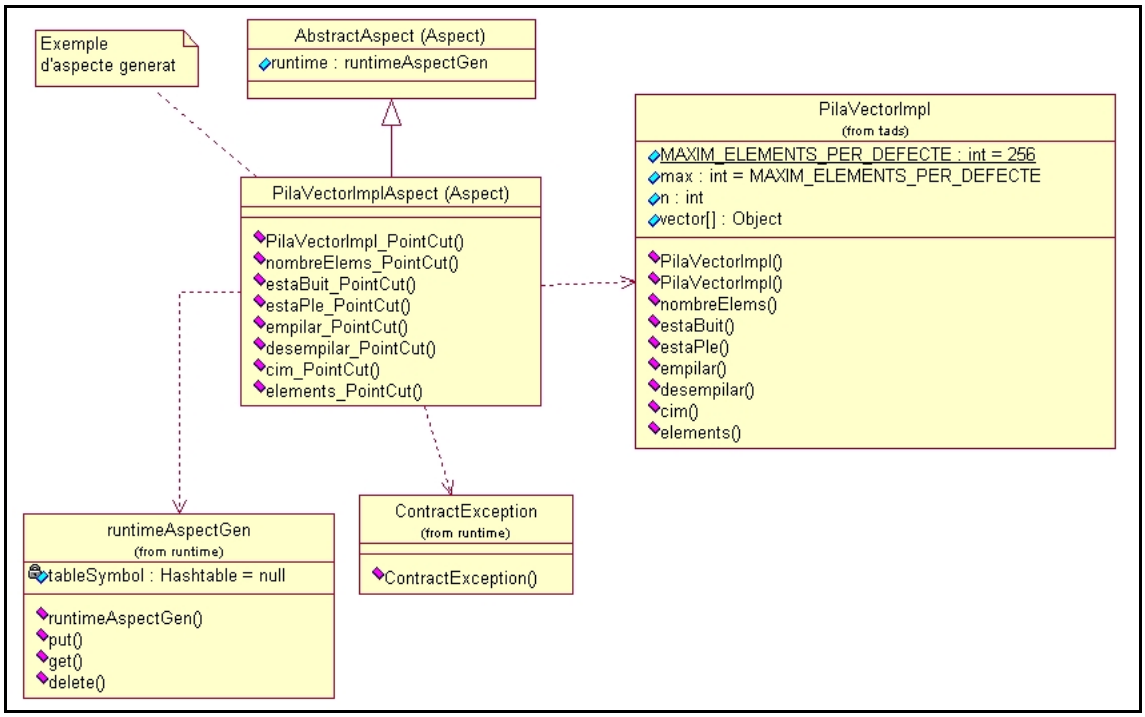
El mètode CustomAspect.buildAspect() és l'encarregat de:

- Emmagatzemar la informació del Doclet: nom de la classe, nom variables, nom mètodes, paràmetres, etc.
- Modificar la informació generada per el Doclet per generar els vectors:
  - OldValue: Emmagatzema les referències a crides OLD de les condicions..
  - VarNames: Emmagatzema el nom de les variables de la classe base.
  - VarNames: Emmagatzema el nom des mètodes de la classe base.

Amb la informació emmagatzemada a les variables del CustomAspect() es generen els Pointcuts i els Advices associats a cada mètode.

Un cop tots els components d'un aspecte han estat generats, la funció savePointCut() permet generar el fitxer .java de l'aspecte.

4.2.2 Diagrama de classes de l'execució d'un Aspecte/Contracte.



Quan s'invoca un mètode de la classe que conté el contracte (per exemple: PilaVectorImpl.cim()), s'executarà l'aspecte PilaVectorAspect, ja que aquest conté un Pointcut pel mètode cim. Si el contracte del mètode conté crides a la funcionalitat $OLD s'haurà d'actualitzar l'objecte runtimeAspectGen.

RuntimeAspectGen, es una classe que a mode de taula de símbols permet gestionar els objectes OLD. Cada aspecte té associat un runtimeObject. Els objectes que s'emmagatzemin a aquesta classe s'haurà de identificar pel fil d'execució ja que un aspecte només té una instància, independentment que la classe amb contracte hagi estat instanciada x vegades.

Per tant els objectes que s'emmagatzemin s'hauran de identificar pel nom de l'objecte i pel fil d'execució. Aquestes dades clau seran les que s'utilitzin per tal de recuperar l'objecte.

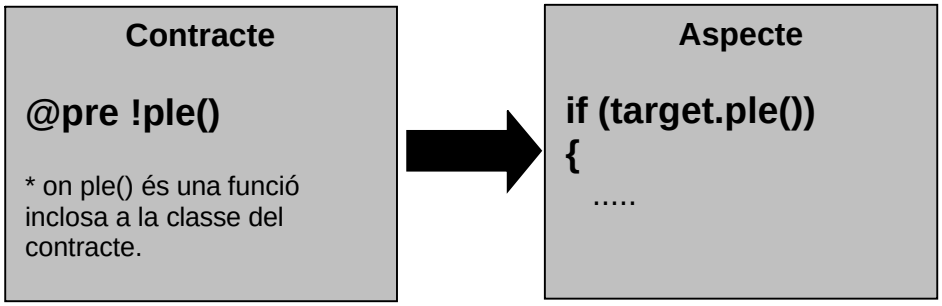
4.3 Implementació del parseig de contractes.

Com ja em vist a l'etapa d'anàlisis per tal de generar l'aspecte no només haurem de parsejar la classe base per tal d'obtenir la informació general de l'aspecte, també haurem de parsejar la informació del contracte. A continuació passarem a analitzar les tasques que haurà de dur a terme el parseig.

La condició d'un contracte pot ser qualsevol sentencia Java que un cop s'evaluï doni com a resultat un valor booleà. Com hem vist a l'exemple de l'aspecte manual, s'hauran de fer modificacions per tal que es pugui utilitzar la condició dintre d'un aspecte.
S'haurà de tenir en compte:

- Crides a mètodes i a variables de la classe: per tal que un aspecte pugui utilitzar les crides a la pròpia classe base, la classe s'haurà de passar com a paràmetre del Pointcut, i aquesta instància s'utilitzarà per fer la crida als membres de la classe contracte.

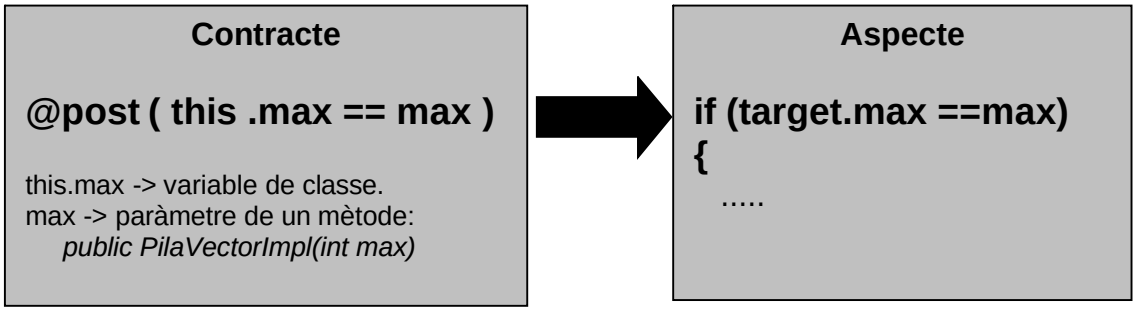
  Veiem un exemple:



  Aquesta transformació implica que s'hauran de diferenciar els mètodes que formen part de la classe, dels mètodes que son d'alguna llibreria Java, ja que en aquest cas no s'haurà de fer la transformació.

- Referències THIS: aquest cas és similar al anterior. Els contractes utilitzen el THIS per tal de diferencia els paràmetres d'un mètode, de les variables de la classe.

  Veiem un exemple:



- Funcionalitat OLD: La implementació de la funcionalitat OLD a l'aspecte obligarà al parseig de la cridada per tal d'obtenir l'objecte a emmagatzemar a la taula de símbols. Per tant s'haurà d'adaptar la informació aportada per la crida OLD, per tal de fer els set() i get() a l'aspecte.

  Veiem un exemple:

Contracte

@post ($old(n) == n+1)

* old(n) -> representa el valor abans d'executar el codi del mètode.

Aspecte

```
before():
{
 AspectGenRuntime.put("$old(n)",n);
 .....
}

after():
{
    if (AspectGenRuntime.get("$old(n)" != target.n +1 )
        .....
}
```

- Comparacions d'objectes (no tipus primitius): Les condicions poden implicar la comparació d'objectes complexos. Aquestes comparacions no admeten els operadors bàsics: == o != , ja que si l'objecte està format per objectes no primitius només es compararan les adreces.

  Veiem un exemple:

```
class Pila
{
   int n =0;
   Vector elements =new Vector();
}
```

  En aquest cas si comparéssim dos objectes Pila amb l'operador == , es compararien les seves adreces, però no es compara els casos que:

  - El valor de n fos el mateix.
  - El contingut de l'array fos el mateix.

Per tal que les comparacions siguin correctes s'hauria d'utilitzar el operador Equals. Això implica que la classe que volem comparar haurà de tenir implementat aquest mètode. Com hem vist els contractes poden incloure condicions que impliquin comparar objectes complexos:

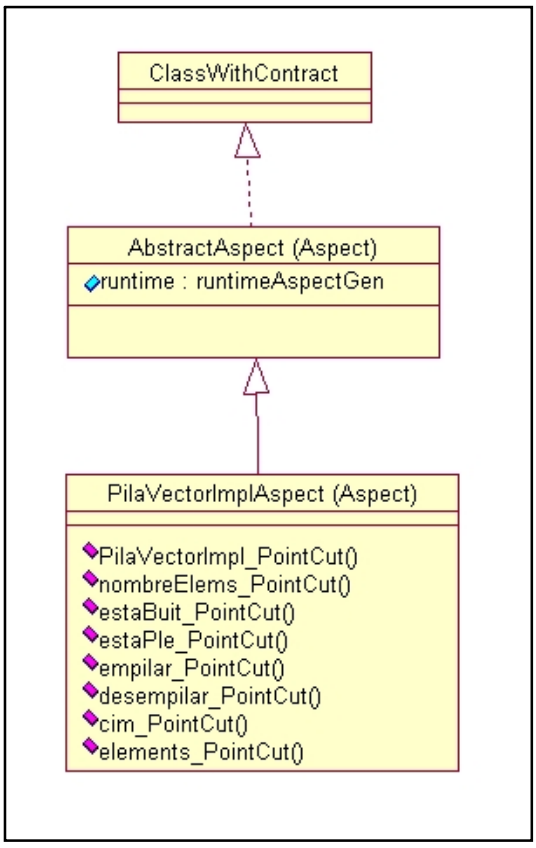
Contracte
@post (this == old(this))

En el cas que aquest contracte pertanyés a las classe PilaVectorImpl, es compararien dos objectes PilaVectorImpl, i el resultat de la comparació seria "false" sempre.

4.3.1 Comparació d'objectes complexos.

La solució a la comparació d'objectes complexos serà l'implementació d'una interfície que formarà part de l'aspecte AspectAbstract i que implementarà un comparador d'objectes. En aquest punt aprofitarem una de les qualitats de la programació orientada a aspectes: la AOP ens permet, mitjançant l'aspecte, modificar la classe base. Modificarem la classe base per tal que implementi una interfície que permeti la comparació d'objectes complexos.

4.3.1.1 Modificació Diagrama de classes.



4.3.1.2 AbstractAspect class.

El codi de la classe abstracta contindrà la interfície ClassWithContract:

```
public abstract aspect AbstractAspect
{
  interface ClassWithContract extends Cloneable
  {
      Object clone();
      boolean equals(Object toCmp);
      boolean fieldEquals(Field field,Object o1,Object o2)throws Exception;
      public boolean arrayEquals(Field field,Object o1,Object o2);
  }


 public Object ClassWithContract.clone()
  {
      ....
  }

  public boolean ClassWithContract.equals(Object toCmp)
  {
      ....
  }
```

L'aspecte concret de la classe ens permetrà implementar la interfície a la classe contracte.

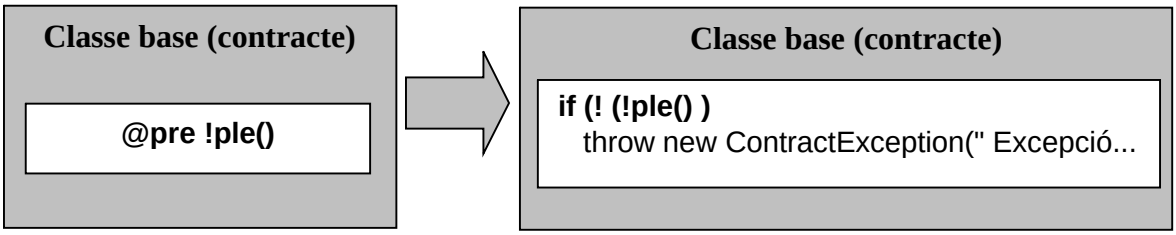
```
public privileged aspect PilaVectorImplAspect extends AbstractAspect
{

  declare parents: PilaVectorImpl implements ClassWithContract;
    .....
}
```

4.3.2 Modificacions de les condicions.

En l'apartat anterior em donat una primera visió d'alguns canvis que haurem de fer per tal d'adaptar les condicions dels tags de contracte per la seva execució dintre d'un aspecte. A continuació detallarem aquest canvis:

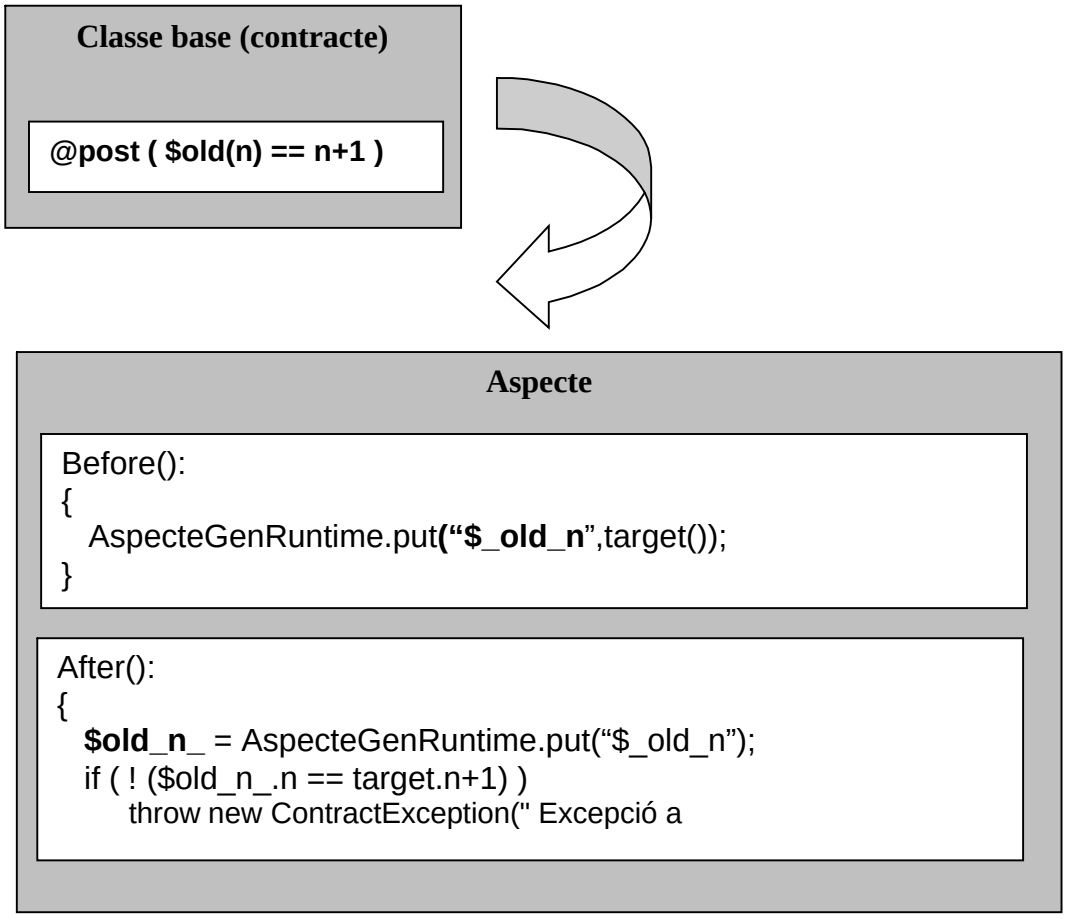
1.- Negació de les condicions:

Les condicions obtingudes dels contractes s'hauran de negar per tal que quan no es compleixin es generi una excepció.



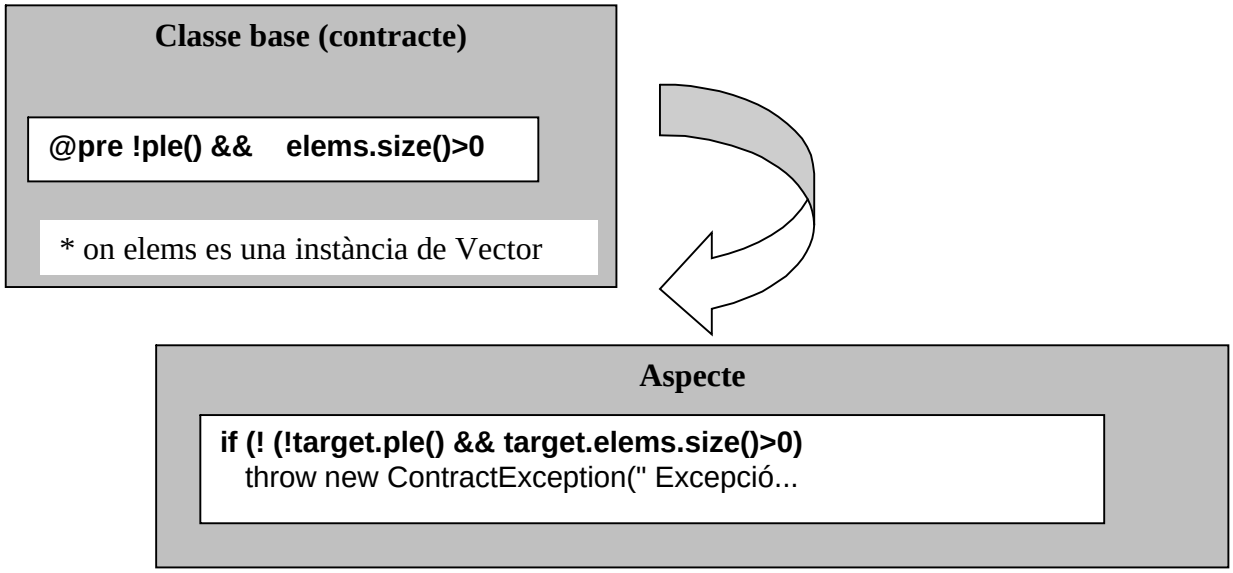
2.- Afegir OLD al before :

En cas que @post utilitzi una crida OLD afegirem una crida AspecteGenRuntime.put() al before. Modificarem el nom del objecte per tal de que al fer el get() poguem crear una variable amb el mateix nom. S'haurà de modificar el nom ja que els caràcters '(' i ')' son caràcters especials en Java



3.- Modificació de les crides:

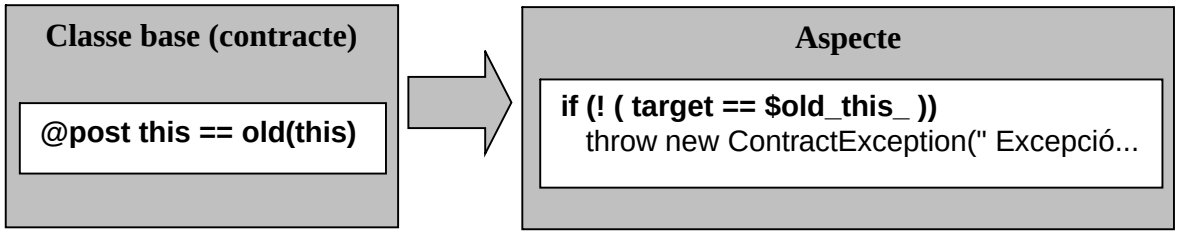
Les condicions que utilitzin crides a mètodes i variables de la classe origen , s'hauran de modificar per tal d'afegir la instància a la crida. Per tal de diferenciar els mètodes que siguin de la classe dels mètodes de Java, haurem de comprovar totes les crides a mètodes.



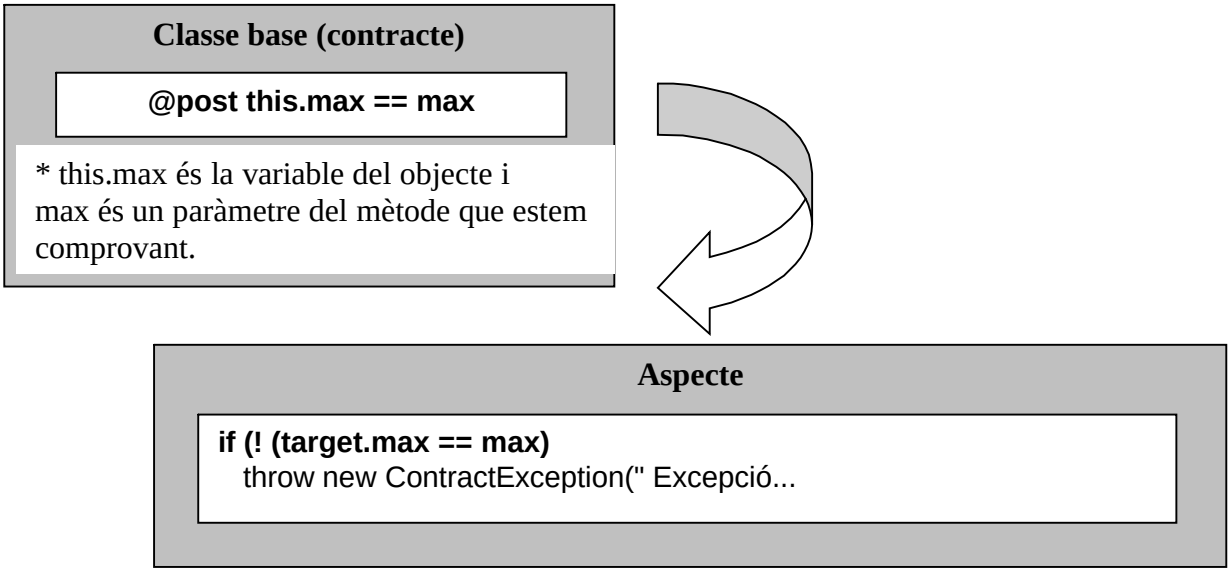
4.- Modificació THIS:

En cas que una condició utilitzi el this, haurem de fer les següents modificacions depenent de la situació:

a) El this fa referència a tot l'objecte: haurem de comprovar que el this no vagi acompanyat de la crida a un mètode o variable de la classe:



b) El this pertany a una crida d'un mètode o variable:



4.3.3 Limitacions sintaxis contracte.

Com hem vist, les modificacions de les condicions per tal que es puguin utilitzar dintre d'un aspecte, requereixen un parseig prou complicat. Aquesta tasca ja havia estat identificada com un risc durant la planificació del projecte. Per tal de minimitzar el risc durant la implementació aplicarem el pla de contingència que teniem previst: introduirem certes limitacions a l'hora d'especificar els contracte.
La limitació de la sintaxis bàsicament es traduirà en deixar espais en blanc entre els diferents items que composen les condicions. Aquest espais en blanc seran utilitzats per minimitzar el cost del parseig.
Regles de sintaxis:

- Totes les condicions hauran d'estar incloses dintre de parèntesis. En cas que les condicions siguin compostes, totes hauran d'estar incloses en un parèntesis superior.

Incorrecte	***Correcte***
`@pre !estaPle()`	`@pre ( !estaPle() )`
`@pre  n>max && max>1`	`@pre ( (n>max) && max>1))`

- S'haurà de deixar un espai en blanc entre el nom d'una variable, mètode o paraula reservada (this) i el següent item , ja sigui : operador de condició, de negació, parèntesis, operador operació...etc

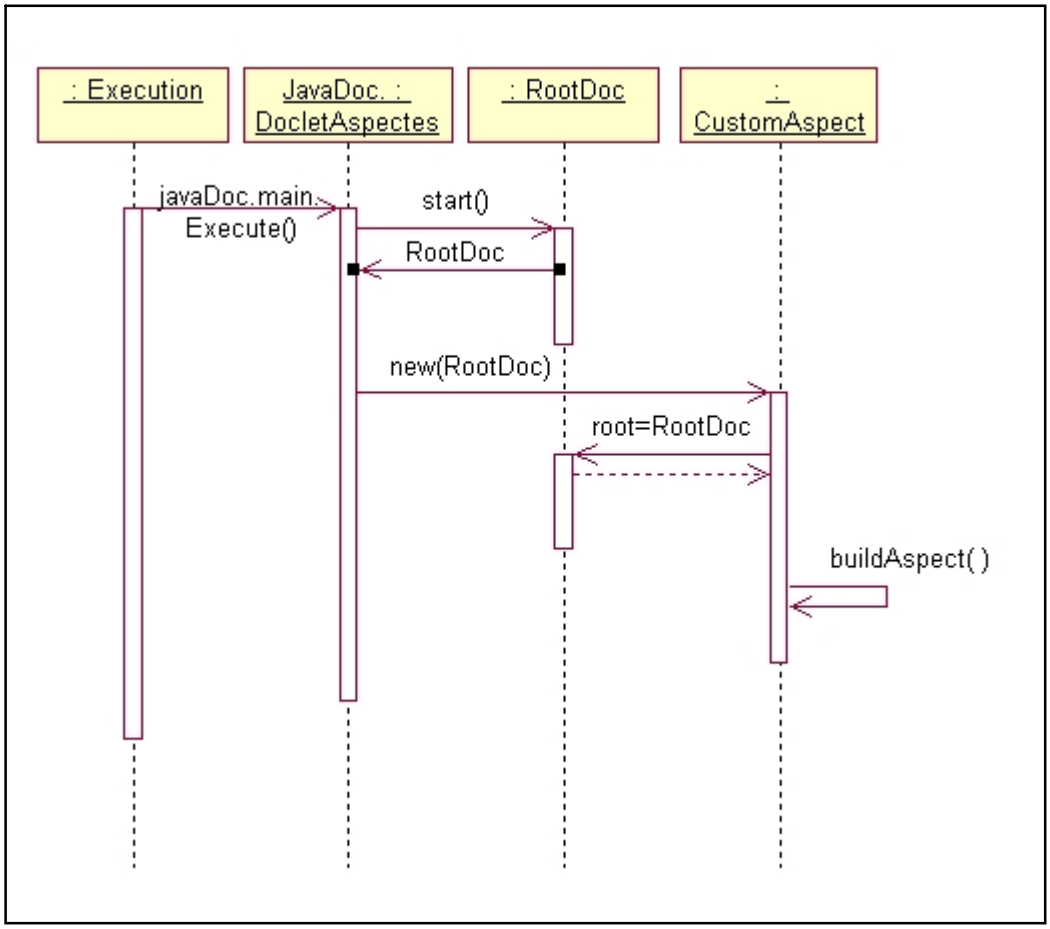
Incorrecte	***Correcte***
`@pre ( !estaPle() )`	`@pre ( ! estaPle() )`
`@pre ( n>max)`	`@pre ( n > max )`
`@post (vector[n-1]==elem )`	`@pre ( vector [ n -1 ] == elem )`

- Les comparacions d'objectes complexos (no primitius: int,real,float) hauran de utilitzar el mètode de comparació Equals.

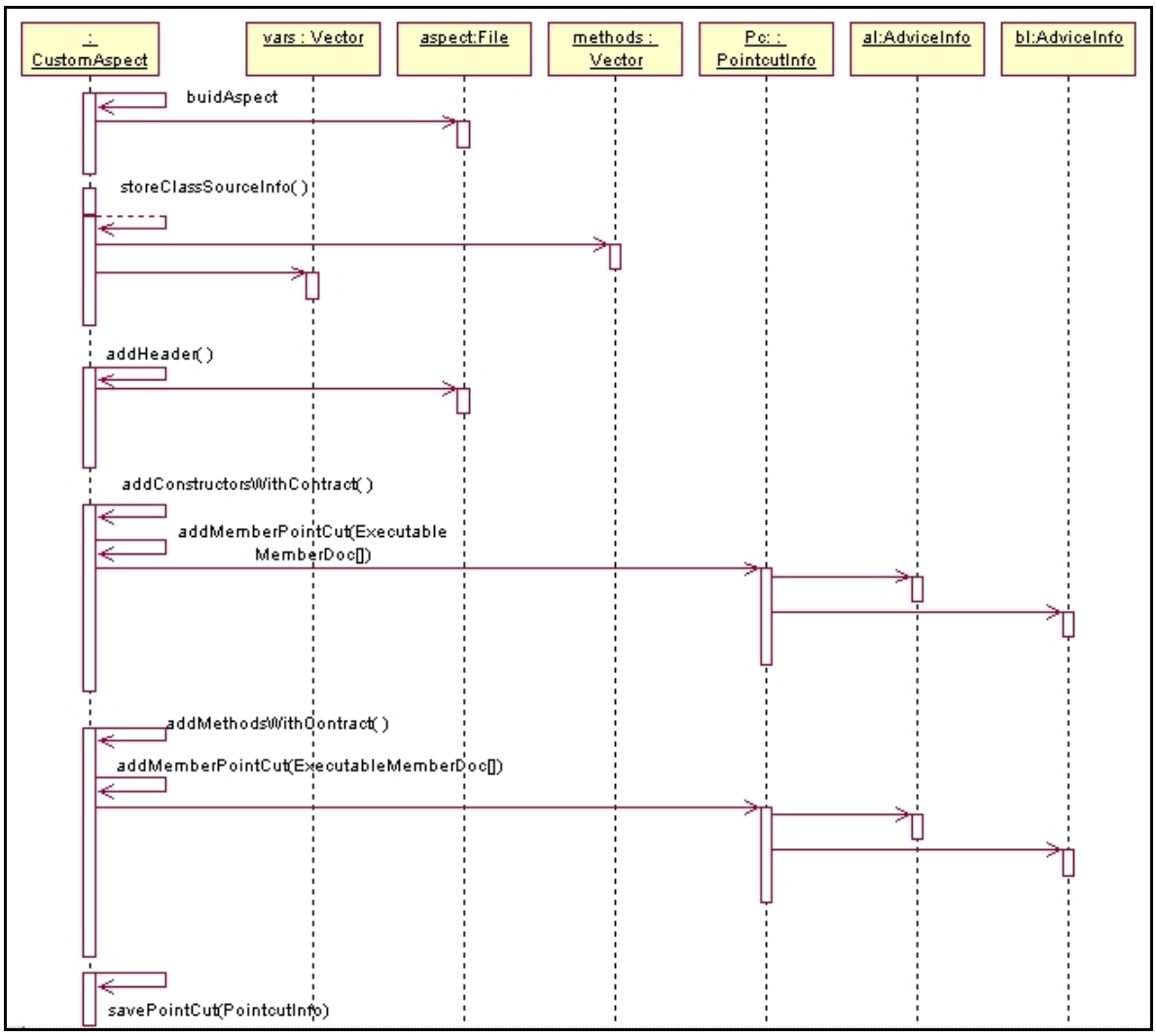
Incorrecte	***Correcte***
`@post (vector == vector2 )`	`@pre ( vector .equals(vector2 )`
`@post (vector 1= vector2 )`	`@pre (!( vector .equals(vector2 ))`

4.4 Diagrames de seqüència.

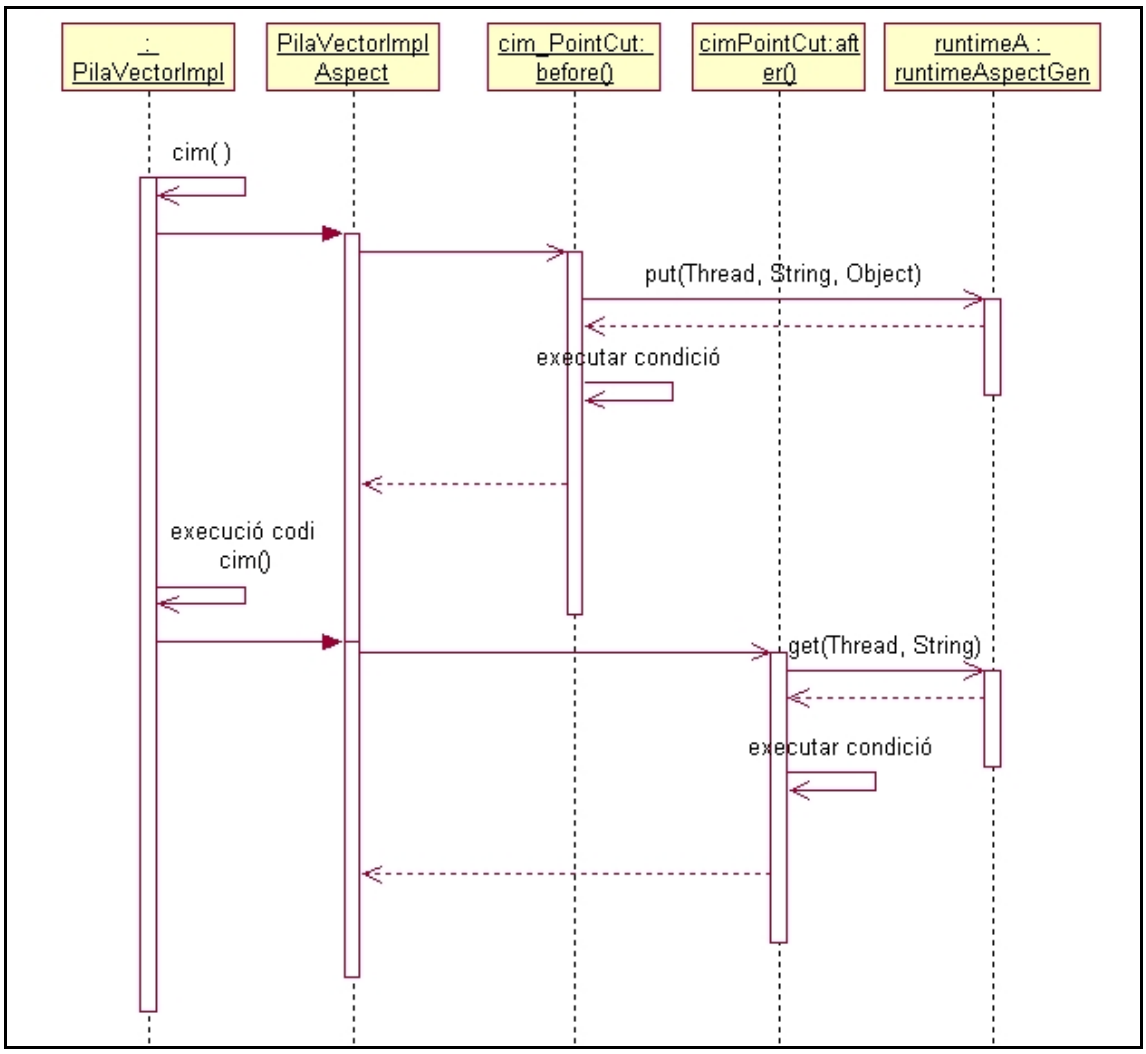
4.4.1 Execució Doclet.



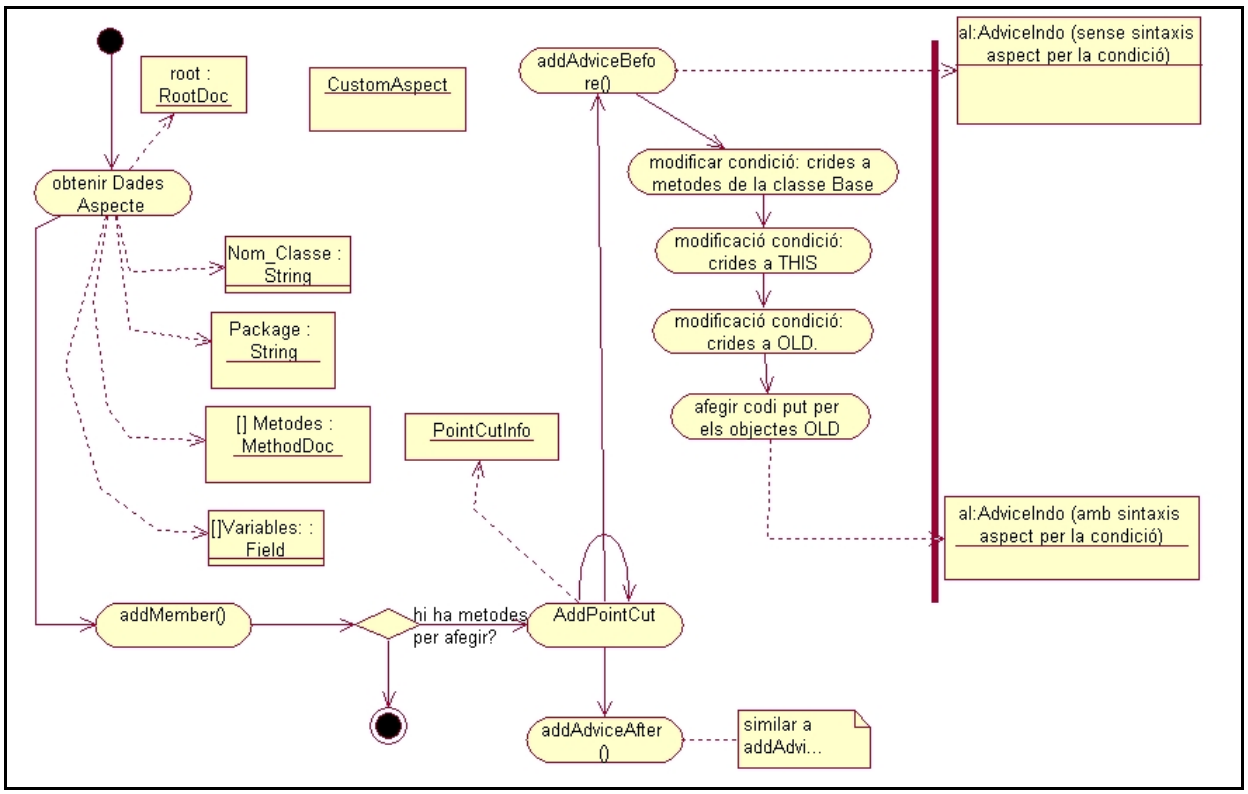
4.4.2 Creació de l'aspecte.



4.4.3 Execució mètode amb contracte i funcionalitat OLD.



4.4.4 Diagrama de estats: Parseig de les condicions.



Capítol 5. Implementació

En el següent capítol veurem el resultat del prototipus obtingut. S'han generat els aspectes que representen els contractes de dos de les classes de la llibreria TADs: PilaVectorImpl i CuaVectorImpl.

En tots dos casos s'han generat programes de prova per tal de comprovar com una violació dels contractes generava un missatge d'error.

5.1 Generació d'aspectes.

A continuació presentem el codi de l'Aspecte generat per PilaVectorImpl Per dur a terme els test també s'ha generat l'Aspecte per CuaVectorImpl.

```
package uoc.ei.dbc.contracts.tad;
import uoc.ei.tads.*;

import aspectGen.runtime.*;
import java.lang.Thread;
public privileged aspect PilaVectorImplAspect extends AbstractAspect
{

  declare parents: PilaVectorImpl implements ClassWithContract;


  pointcut PilaVectorImpl_1PointCut(): execution (public
  PilaVectorImpl.new());

  before(PilaVectorImpl p): target(p) && PilaVectorImpl_1PointCut()
  {
  }

  after(PilaVectorImpl p): target(p) && PilaVectorImpl_1PointCut()
  {
      if (!(( p.n == 0 ) && ( p .max == p.MAXIM_ELEMENTS_PER_DEFECTE
      )))
      {
            throw new ContractException(" Excepció a
            PilaVectorImpl.PilaVectorImpl() no es compleix el
            contracte de la condició. ");
      }

      if (!(( p.n >= 0 )))
      {
            throw new ContractException(" Excepció a
            PilaVectorImpl.PilaVectorImpl() no es compleix el
            contracte de la condició. ");
      }
  }

  pointcut PilaVectorImpl_2PointCut( int max): execution (public
  PilaVectorImpl.new( int)) && args( max);

  before(PilaVectorImpl p,  int max): target(p) &&
  PilaVectorImpl_2PointCut( int) && args ( max)
```

```
{
  if (!(( max >= 0 )))
  {
    throw new ContractException(" Excepció a
    PilaVectorImpl.PilaVectorImpl( int) no es compleix el contracte
    de la condició. ");
  }
}

after(PilaVectorImpl p, int max): target(p) &&
PilaVectorImpl_2PointCut( int) && args ( max)
{
  if (!(( p.n ==0 ) && ( p .max == max )))
  {
    throw new ContractException(" Excepció a
    PilaVectorImpl.PilaVectorImpl( int) no es compleix el contracte
    de la condició. ");
  }

  if (!(( p.n >= 0 )))
  {
    throw new ContractException(" Excepció a
    PilaVectorImpl.PilaVectorImpl( int) no es compleix el contracte
    de la condició. ");
  }
}

pointcut nombreElems_3PointCut(): execution (public int
PilaVectorImpl.nombreElems(..));

before(PilaVectorImpl p): target(p) && nombreElems_3PointCut()
{
    runtimeObject.put(Thread.currentThread(),"$old_this_",((ClassWi
    thContract)p).clone());
}

after(PilaVectorImpl p) returning (int $return): target(p) &&
nombreElems_3PointCut()
{
    PilaVectorImpl $old_this_=(PilaVectorImpl)
    (runtimeObject.get(Thread.currentThread(),"$old_this_"));
    if (!(( $return == p.n ) && ( p .equals( $old_this_ ) )))
    {
          throw new ContractException(" Excepció a
          PilaVectorImpl.nombreElems() no es compleix el contracte
          de la condició. ");
    }
    if (!(( p.n >= 0 )))
    {
          throw new ContractException(" Excepció a
          PilaVectorImpl.nombreElems() no es compleix el contracte
          de la condició. ");
    }
}

pointcut estaBuit_4PointCut(): execution (public boolean
```

```
PilaVectorImpl.estaBuit(..));

before(PilaVectorImpl p): target(p) && estaBuit_4PointCut()
{
    runtimeObject.put(Thread.currentThread(),"$old_this_",((ClassWi
    thContract)p).clone());
}

after(PilaVectorImpl p) returning (boolean $return): target(p) &&
estaBuit_4PointCut()
{
    PilaVectorImpl $old_this_=(PilaVectorImpl)
    (runtimeObject.get(Thread.currentThread(),"$old_this_"));
      if (!(( $return == ( p.n == 0 ) && ( p .equals( $old_this_ )
      ) )))
    {
          throw new ContractException(" Excepció a
          PilaVectorImpl.estaBuit() no es compleix el contracte de
          la condició. ");
    }

    if (!(( p.n >= 0 )))
    {
          throw new ContractException(" Excepció a
          PilaVectorImpl.estaBuit() no es compleix el contracte de
          la condició. ");
    }
}

pointcut estaPle_5PointCut(): execution (public boolean
PilaVectorImpl.estaPle(..));

before(PilaVectorImpl p): target(p) && estaPle_5PointCut()
{
    runtimeObject.put(Thread.currentThread(),"$old_this_",((ClassWi
    thContract)p).clone());
}

after(PilaVectorImpl p) returning (boolean $return): target(p) &&
estaPle_5PointCut()
{
    PilaVectorImpl $old_this_=(PilaVectorImpl)
    (runtimeObject.get(Thread.currentThread(),"$old_this_"));
    if (!(( $return ==( p.n == p.max ) && ( p .equals( $old_this_ )
    ) )))
    {
          throw new ContractException(" Excepció a
          PilaVectorImpl.estaPle() no es compleix el contracte de la
          condició. ");
    }

    if (!(( p.n >= 0 )))
    {
          throw new ContractException(" Excepció a
          PilaVectorImpl.estaPle() no es compleix el contracte de la
          condició. ");
    }
```

```
}

pointcut empilar_6PointCut( java.lang.Object elem): execution
(public void PilaVectorImpl.empilar(..)) && args( elem);

before(PilaVectorImpl p,  java.lang.Object elem): target(p) &&
empilar_6PointCut( java.lang.Object) && args ( elem)
{
  if (!((! p.estaPle( ) )))
  {
    throw new ContractException(" Excepció a
    PilaVectorImpl.empilar( java.lang.Object) no es compleix el
    contracte de la condició. ");
  }

    runtimeObject.put(Thread.currentThread(),"$old_n_",((ClassWithC
    ontract)p).clone());
}

after(PilaVectorImpl p, java.lang.Object elem): target(p)
&&empilar_6PointCut( java.lang.Object) && args ( elem)
{
    PilaVectorImpl $old_n_=(PilaVectorImpl)
    (runtimeObject.get(Thread.currentThread(),"$old_n_"));
    if (!(( p.n == $old_n_.n +1) &&  ( p.vector [ p.n -1] .equals(
    elem ) ) ))
    {
          throw new ContractException(" Excepció a
          PilaVectorImpl.empilar( java.lang.Object) no es compleix
          el contracte de la condició. ");
    }
    if (!(( p.n >= 0 )))
    {
          throw new ContractException(" Excepció a
          PilaVectorImpl.empilar( java.lang.Object) no es compleix
          el contracte de la condició. ");
    }
}

pointcut desempilar_7PointCut(): execution (public java.lang.Object
PilaVectorImpl.desempilar(..));

before(PilaVectorImpl p): target(p) && desempilar_7PointCut()
{
  if (!((! p.estaBuit( ) )))
  {
    throw new ContractException(" Excepció a
    PilaVectorImpl.desempilar() no es compleix el contracte de la
    condició. ");
  }
    runtimeObject.put(Thread.currentThread(),"$old_n_",((ClassWithC
    ontract)p).clone());
}

after(PilaVectorImpl p) returning (java.lang.Object $return):
target(p) && desempilar_7PointCut()
{
```

```
    PilaVectorImpl $old_n_=(PilaVectorImpl)
    (runtimeObject.get(Thread.currentThread(),"$old_n_"));
    if (!(( p.n == $old_n_.n -1 )))
    {
          throw new ContractException(" Excepció a
          PilaVectorImpl.desempilar() no es compleix el contracte de
          la condició. ");
    }

    if (!(( p.n >= 0 )))
    {
          throw new ContractException(" Excepció a
          PilaVectorImpl.desempilar() no es compleix el contracte de
          la condició. ");
  }
}

pointcut cim_8PointCut(): execution (public java.lang.Object
PilaVectorImpl.cim(..));

before(PilaVectorImpl p): target(p) && cim_8PointCut()
{
  if (!((! p.estaBuit( ) )))
  {
    throw new ContractException(" Excepció a  PilaVectorImpl.cim()
    no es compleix el contracte de la condició. ");
  }

    runtimeObject.put(Thread.currentThread(),"$old_this_",((ClassWi
    thContract)p).clone());
}

after(PilaVectorImpl p) returning (java.lang.Object $return):
target(p) && cim_8PointCut()
{
    PilaVectorImpl $old_this_=(PilaVectorImpl)
    (runtimeObject.get(Thread.currentThread(),"$old_this_"));
    if (!(( ( $return .equals( p.vector [ p.n -1] ) ) && ( p
    .equals ( $old_this_ ) ) )))
    {
          throw new ContractException(" Excepció a
          PilaVectorImpl.cim() no es compleix el contracte de la
          condició. ");
    }

    if (!(( p.n >= 0 )))
    {
          throw new ContractException(" Excepció a
          PilaVectorImpl.cim() no es compleix el contracte de la
          condició. ");
    }
}
public Object PilaVectorImpl.clone()
{
try
 {
```

```
      PilaVectorImpl obj = (PilaVectorImpl)(super.clone());
      obj.vector=new Object[vector.length];
      for (int i=0;i<vector.length;i++)
      {
        obj.vector[i]=this.vector[i];
      }
      return obj;
  }
  catch (Exception ex)
  {
     System.out.println(" Excepció al clonar objecte base");
     return null;
  }
 }
}
```

5.2 Test.

Per tal de testejar el prototipus hem generat els contracte de les classes PilaVectorImpl i CuaVectorImpl. Amb els aspectes generats hem testejat l'execució de contractes.
S'han utilitzat les següents classes inclosses al projecte:

- Test_PilaVector_T1: Conté les proves pels contractes de la classe PilaVectorImpl.

- Test_PilaVector_T2: Conté les proves pels contractes de la classe PilaVectorImpl_ToCrashTest. Aquesta classe és una copia de PilaVectorImpl, amb el codi modificat per tal de forçar violaments dels contractes modificant el codi d'execució del mètodes.

- Test_CuaVector_T1: Conté les proves pels contractes de la classe CuaVectorImpl.

- Test_CuaVector_T2: Conté les proves pels contractes de la classe PilaCuaImpl_ToCrashTest. Aquesta classe és una copia de CuaVectorImpl, amb el codi modificat per tal de forçar violaments dels contractes modificant el codi d'execució del mètodes.

```
Resultat Execució: Test_PilaVector_T1

---------- TEST CONSTRUCTORS:
TEST: pilaTest = new pilaVector():
  Es compleix contracte. Resultat -> Test OK
TEST: pilaTest2 = new pilaVector(1):
  Es compleix contracte. Resultat -> Test OK
TEST: pilaTest3 = new pilaVector(-1): (violacio de contracte @pre)
  aspectGen.runtime.ContractException: Excepció a PilaVectorImpl.PilaVectorImpl(
int) no es compleix el contracte de la condició.
  S'ha executat el contracte. Resultat -> Test OK

---------- TEST METODES:
---------- TEST nombreElems():
TEST: nombreElems() -> 0
  No es viola el contracte Resultat -> Test Ok
---------- TEST estaBuit():
TEST: estaBuit() -> true
  No es viola el contracte Resultat -> Test Ok
---------- TEST estaPle():
TEST: estaPle() -> false
  No es viola el contracte Resultat -> Test Ok
---------- TEST empilar():
TEST: test1.empilar('valor1') ->
  No es viola el contracte Resultat -> Test Ok
TEST: test2.empilar('valor1') ->
  No es viola el contracte Resultat -> Test Ok
TEST: test2.empilar('valor2') -> (violacio de contracte @pre)
  aspectGen.runtime.ContractException: Excepció a PilaVectorImpl.empilar(
java.lang.Object) no es compleix el contracte de la condició.
  Violacio del contracte. Resultat -> Test Ok
---------- TEST desempilar():
TEST: test1.desempilar() -> valor1
  No es viola el contracte Resultat -> Test Ok
TEST: test1.desempilar() ->
  aspectGen.runtime.ContractException: Excepció a PilaVectorImpl.desempilar() no
es compleix el contracte de la condició.
  Violacio del contracte. Resultat -> Test Ok
---------- TEST cim():
TEST: cim() -> (violacio de contracte @pre)
  aspectGen.runtime.ContractException: Excepció a PilaVectorImpl.cim() no es
compleix el contracte de la condició.
  Violacio del contracte. Resultat -> Test Ok
```

Capítol 6. Conclusions

Arribats a aquest punt en podem extreure les següents conclusions:

- S'ha assolit l'objectiu principal del treball. Amb l'estudi de l'AOP i una de les seves implementacions, AspectJ, hem aconseguit implementar els mecanismes per tal de donar suporta al disseny per contractes.

- Hem aconseguit el desenvolupament d'un prototipus capaç de generar Aspectes associats als contractes de una classe base de manera automàtica.

- La utilització de l'AOP ens ha permès crear tota la extensió de funcionalitats requerides per l'ús del disseny per contractes sense tocar les classes base. Únicament han estat modificades per afegir la informació del contracte.

- A nivell personal el treball ha estat força enriquidor. El conjunt de tecnologies amb que s'ha hagut de treballar estan a primer ordre del dia en l'actualitat. El treball m'ha permès fer un recorregut per totes elles que m'ha ajudat a comprendre millor el seu funcionament, a tenir una visió força més completa de com s'integren totes aquestes tecnologies en una plataforma Java.

6.1 Línies futures de treball

El treball pretenia crear un prototipus per la generació d'aspectes de la llibreria TADs de l'assignatura de Estructures de la Informació. Hem testejat la generació i execució de dues de les classes de la llibreria amb resultats satisfactoris. Tot i que la generació automàtica ens permetria generar aspectes per totes les classes s'hauran de implementar els contractes de totes elles i després fer la generació. La sintaxis de contracte que hem provat ens hauria de permetre generar els contractes per totes elles. Tot i això el disseny per contractes permet l'ús d'algunes funcionalitats de tipus old, les quals no hem implementat com el cas de all().

Per tant les línies futures de treball haurien de ser:

- Crear contractes per totes les classe de la llibreria.

- Provar la generació d'aspectes de totes elles.

- Afegir noves funcionalitats a la sintaxis dels contractes:
  - $forAll i $exists: son funcionalitats que permeten fer comprovacions sobre taules i col·leccions i que habitualment apareixen al definir pre i postcondicions.

- Augmentar la flexibilitat de la sintaxis en la definició dels contractes: com hem vist en algun capítol, hem limitat la sintaxis de la definició de contractes. S'haurien de crear els mecanismes de parseig per tal que es pogues flexibilitzar la definició. Una possible solució seria crear un parseig més complex en el que no calgués l'ús

d’espais en blanc per detectar els possibles tags clau. Una altre solució seria l’ús d’una eina per l’anàlisi lèxic de les condicions. Per exemple es podria utilitzar el Jlex per tal de crear un lèxic per la sintaxis de les condicions.

Bibliografía

[1] Ramnivas Laddad. *"I want my AOP!". Separate concerns with aspect-oriented programming. http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect.html. Gener 2002.*

[2] http://aosd.net/index.php. *Aspect-Oriented Software development site.*

[3]http://www.revista.unam.mx/vol.4/num5/art11/art11-5.htm*Revista digital Universitaria.*

Annexos

Annex A Posada en marxa de la nostra aplicació

En aquest apartat explicarem les eines necessàries per posar en marxa la nostra aplicació i com dur a terme la seva instal·lació.

Per poder utilitzar AspectJ necessiteu disposar de la versió 1.4.2 del JDK o superior, que la trobareu a:

- JDK 1.4.2 http://java.sun.com/j2se/1.4.2/downloadl.html

També haureu de disposar dels paquets i llibreries necessaris per utilitzar AspectJ. En la següent adreça trobareu un .jar que us realitzarà la instal·lació automàtica del compilador per AspectJ, un Entorn visual per AspectJ massa simple, scripts de compilació per *ant* i documentació.

- AspectJ http://www.eclipse.org/aspectj/index.html

A.1. Instal·lació d'AspectJ sobre Eclipse

Eclipse és un entorn integrat (IDE) de desenvolupament per a Java que permet la integració de AspectJ de forma fàcil.

Per posar en marxa el sistema haureu d'obtenir l'Eclipse i el plug-in d'AspectJ

- Eclipse 2.1.2. http://www.eclipse.org/downloads/index.php.

Per baixar-se el plugin, posar-lo en marxa i provar el seu funcionament, la següent adreça us proporciona una guia pas a pas de la instal·lació

- Breu manual de instal·lació del plug-in de AspectJ per a Eclipse. http://eclipse.org/ajdt/

A.2. Compilació amb Eclipse

La compilació de la nostra la nostra eina junt amb la lliberia de la qual volem obtenir els Aspectes associats als contractes és imprescindible per tal d'obtenir els resultats esperats. Aquesta compilació és especialment senzilla dins Eclipse.
Junt amb la memòria s'han adjuntat dues carpetes *codi_font* i *jar*.

- *project* conté el codi font del prototipus juntament amb els exemples que hem utilitzat per fer les proves
- *jar* conté el mateix però amb el codi de prototipus amb format *jar*.

En el primer dels casos la compilació és molt senzilla, només cal crear un nou projecte i importar el codi font tan de les classes de l'aplicació d'exemple com les classes de la nostra eina i l'aspecte concret, que podem modificar per definir un altre cas d'ús.

En el segon cas l'única diferència és que enlloc d'importar els codis font cal indicar en les propietats de compilació que utilitzarem llibreries externes.

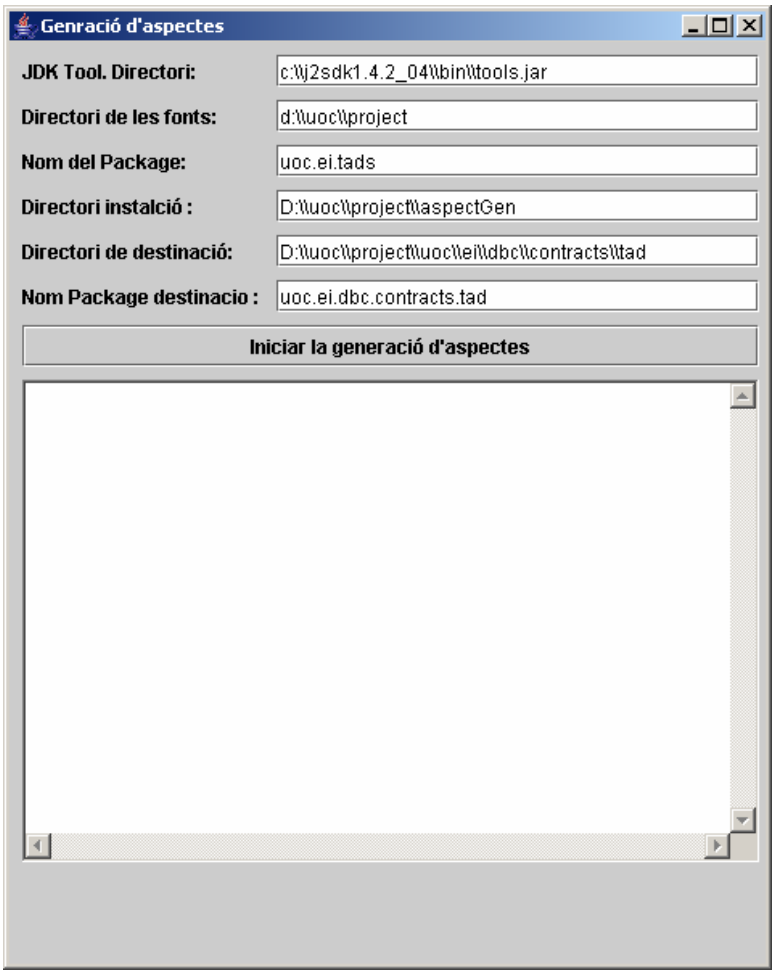
A.3. Ús de llibreries externes .JAR

A més de les llibreries ja esmentades hem utilitzat una llibreria freeware per tal de generar la GUI. Aquesta llibreria està inclosa tant al zip de les classes font com al zip amb els jar. En cas d'executar el prototipus en l'entorn Eclipse, s'haurà d'afegir aquesta llibreria al projecte.

Annex B Manual d'usuari.

B1.Especificació de les paràmetres de generació.

Al iniciar l'execució de l'aplicació apareix una finestra amb uns camps on es pot especificar la següent informació:



JDKToolDirectori: s'ha d'especificar la localització de la instal·lació del JDK i del fitxer tools.jar. Aquest jar permet iniciar l'execució del JavaDoc.

Directori de les fonts: permet especificar el directori de les classes que seran parsejades per tal de generar els aspectes/contracte.

Nom del package: determina el package de les classes que seran parsejades per tal de generar els aspectes/contracte.

Directori instal·lació: Directori on s'ha instal·lat el prototipus.

Directori Destinació: determina on s'emmagatzemaran els aspectes generats.

Package Destinació: permet especificar el package al que perteneixeran els aspectes generats.

Un cop s'hagi comprovat els paràmetres d'execució es pot iniciar la generació prement el botó. Al panell de l'aplicació apareixeran els missatges de l'execució, on s'informarà del aspectes/contractes que s'han creat amb èxit així com les classes que no tenen cap contracte especificat i de les quals no ha estat possible generar aspectes.