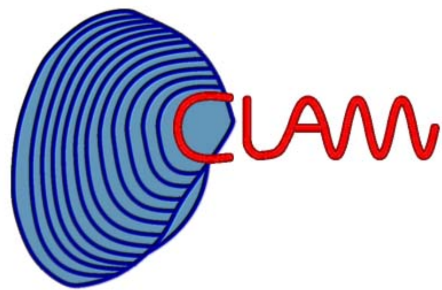

CLAM : C++ Library for Audio and Music

Pau Arumí Albó, Juny del 2002

Index

Index 2

Pròleg 6

PRIMERA PART: CONCEPTES PREVIS I REQUERIMENTS 8

1.1 Introducció 8

1.1.1 Aproximació als objectius de CLAM a través de casos d'us 9

1.1.2 Dos modes de funcionament: supervisat i no supervisat 13

1.1.3 Punt de partida del Projecte 14

1.1.4 CLAM com a part d'una distribució lliure de *Linux* 15

1.2 Definició del Projecte 16

1.2.1 Motivació 16

1.2.2 Objectius 18

1.2.3 Visió General del Sistema 21

1.3 Requeriments sobre el flux de dades. 26

1.3.1 El futur mòdul de flow control 26

1.3.2 Alguns patrons de flux: 27

1.4 Conceptes de disseny del software i eines escollides 32

1.4.1 Orientació a objectes 32

1.4.2 Patrons de disseny 33

1.4.3 Programació genèrica 34

1.4.4 Un llenguatge de programació multiparadigma : C++ 39

1.4.5 Standard Template Library 39

SEGONA PART: REALITZACIÓ DEL PROJECTE 44

2.1 Metodologia i entorn de treball 44

2.1.1 Recursos....................44

2.1.2 Gestió de desenvolupament concurrent 48

2.1.3 Gestió de tasques pendents i correcció d'errors (*bug tracking*) 51

2.1.2 Documentació 53

2.1.3 Cicle de desenvolupament 58

2.2 Disseny i implementació 62

2.2.1 *Dynamic types*....................62

2.2.2 Classes de processament....................75

2.2.3 Classes de dades de processament 85

2.2.4 Controls 90

2.3 Exemples d'aplicacions 98

2.3.1 Spectral delay....................98

2.3.2 SMS Analisis-Síntesis 99

2.3.3 Rappid....................100

2.4 Estudi econòmic 102

2.4.2 Comentari sobre la planificació 103

2.5 Conclusions i línies futures 104

2.5.1 Conclusions respecte CLAM i el treball realitzat 104

2.5.2 Línies futures 106

Bibliografia..108

Apèndix A. Manual d'usuari i desenvolupador. 110

Apèndix B. Nodes (treball en progrés)..112

Pròleg

El projecte final de carrera realitzat s'emmarca dins el projecte CLAM, desenvolupat pel Grup de Tecnologia Musical (MTG) de l'Institut Universitari de l'Audivisual (Universitat Pompeu Fabra).

L'MTG és un grup interdisciplinari el qual destaca per la recerca en l'àrea de l'anàlisi i sínitesi de so per a aplicacions relacionades amb la música i la multimèdia. Durant els últims anys, el grup ha anat creixent a un ritme vertiginós participant en un gran nombre de projectes, tant a nivell estatal com europeu i mundial.

Els orígens del projecte CLAM (que és l'acrònim de *C++ Library for Audio and Music*) es remunten a l'any 2000 quan l'MTG té la necessitat d'implementar diversos sistemes software reaprofitant anteriors desenvolupaments, i donada la complexitat d'aquests projectes, la feina de reaprofitar codi resulta ser molt complexa i costosa. Per tant es va decidir iniciar aquest projecte amb el proposit de dotar d'eines i infrastructura software en forma de llibreria en C++ a la resta de desenvolupadors del grup, posant l`ènfasi en l'aspecte de la qualitat del software de la llibreria.

A la tardor de l'any 2001 el projecte comença a definir-se i és precisament en aquestes dates quan vaig decidir buscar el tema pel meu PFC ja que havia quasi acabat els cursos de la carrera. Donat el meu interés per la música i la producció musical em vaig presentar a l'MTG expressant el meu interés per participar en els projectes que es desenvolupaven dins aquest grup. Així és com vaig entrar a formar part de l'MTG, com a becari de recerca, treballant amb el tot just començat projecte CLAM –aleshores anomenat simplement MTG-Classes a manca d'un nom. Aquest projecte final de carrera presenta, doncs, part del treball que he realitzat durant el primer any del projecte CLAM.

Darrerament, CLAM ha ampliat els seus objectius ja que desde fa dos mesos forma part del projecte europeu IST *Agnula* l'objectiu del qual és produir una distribució Linux, de codi obert, orientada a la producció multimèdia.

Aquest document s'estructura en dos grans apartats: la primera de requeriments i dels conceptes previs que es necessiten per fer i entendre el projecte; i la segona que és pròpiament la realització del projecte, incloent-hi les decisions de disseny més rellevants, la metodologia usada, també un anàlisi enconòmic i la planificació del projecte, i acabant amb les conclusions i línies futures.

Primera part: Conceptes previs i requeriments

1.1 Introducció

Al ser CLAM (*C++ Library for Audio and Music*) una llibreria o framework, i per tant, una eina per construir altres eines, és natural que definir el *què* ha de ser capaç de fer sigui dificultós i alhora s'hagi de respondre d'una manera molt general. A l'apartat d'objectius §1.2.2 començaré explicant quin són aquests objectius generals i es comprovarà que són massa lliures perquè per sí sols duguin a un disseny i implementació.

Aquesta característica no l'entenc com una mancança o com uns requeriments mal refinats o poc investigats. L'entenc com a pròpia de la naturalesa de l'eina. D'altra banda, tot i que els requeriments explícits sí que són generals, teníem molta informació concreta sobre problemes típics del domini, i la manera 'correcta' de com la llibreria els hauria d'afrontar o resoldre. Aquest tipus d'informació provenia sobretot de l'experiència acumulada pels membres del grup d'investigació.

L'obtenció de requeriments s'ha realitzat a partir d'entrevistes o reunions amb els experts en el domini, que alhora havien de ser els futurs usuaris de l'eina. D'altra banda, cal tenir en compte que el cap del projecte CLAM, entra en aqueta categoria, i que tots els membres del projecte tenim coneixements sobre altres eines de tractament de l'audio i la música, i per tant, sovint els requeriments han sortit del propi criteri de l'equip desenvolupador.

Per tant es va optar per un desenvolupament en espiral, en què en cada iteració es faria una investigació dels requeriments i s'intentaria que aquests fóssin prou concrets i acotats com perquè duguessin fàcilment a un disseny i implementació, però que alhora no fóssin intrusius amb les parts encara no tractades, en el sentit de no prendre decisions que en el futur haguéssin de ser revisades, per haver afrontat el problema amb una visió poc general.

De la mateixa manera que desenvolupant aquest projecte he hagut d'extreure requeriments concrets a partir de casos d'us concrets o exemples, crec que una bona manera d'explicar-lo també és començant amb els casos d'us.

1.1.1 Aproximació als objectius de CLAM a través de casos d'us

Els casos d'us desde un punt de vista general són molt senzills: Podem identificar dos tipus actors:

- L'**usuari** de CLAM, en el sentit que reusa components de la llibreria, sense modificar-la.
- L'usuari **desenvolupador** que apart d'usar la llibreria extén els seus components; passin els nous components, a formar part, o no, de la llibreria.

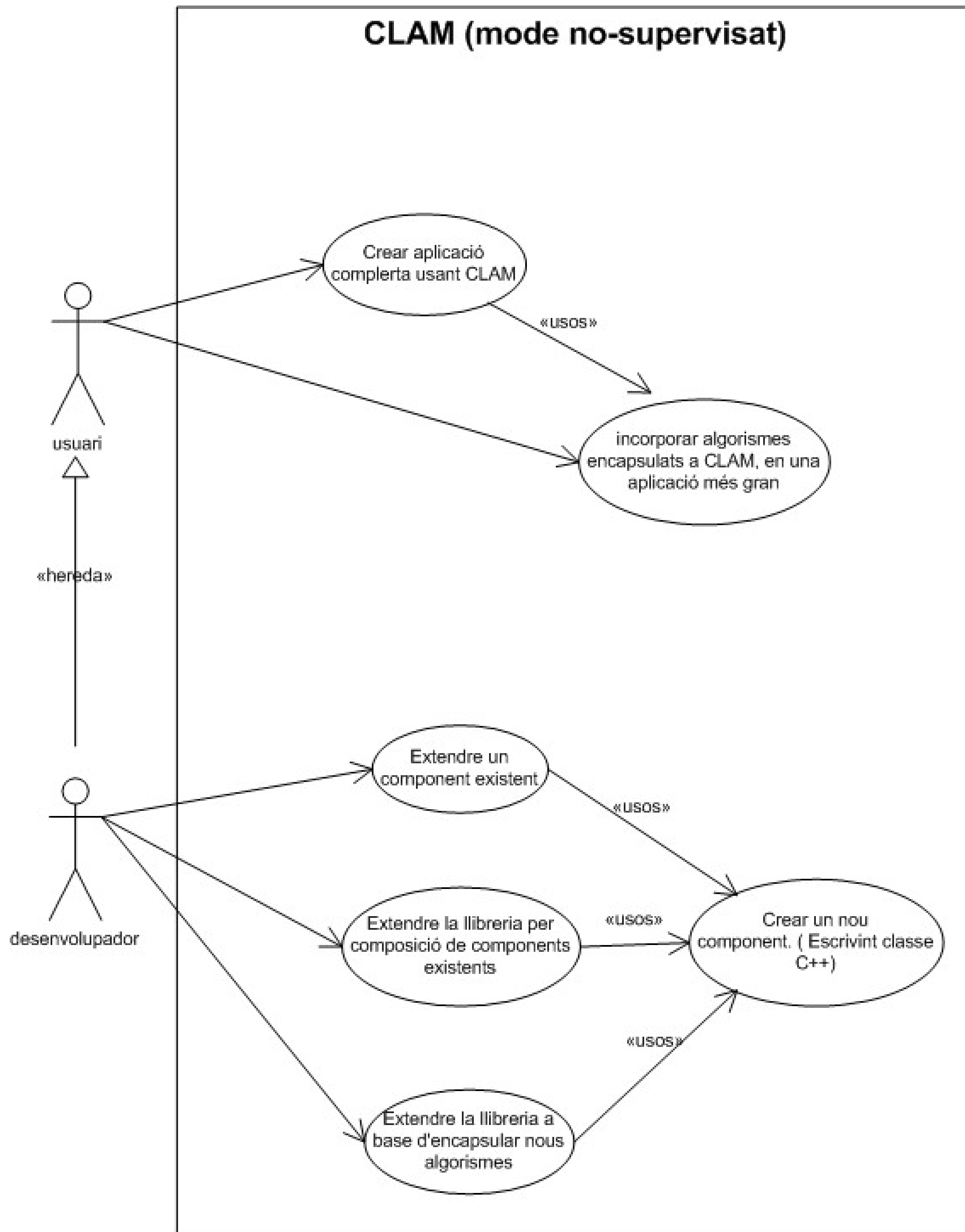

Figura 1Casos d'ús dels actors usuari i desenvolupador

Com s'observa en aquest diagrama d'us l'actor 'usuari' té dues maneres d'utilitzar CLAM: com si fos una llibreria típica de processament del senyal. És a dir, incorporant únicament els algorismes encapsulats en les classes de la llibreria a una aplicació més gran. O bé usant

CLAM com un framework orientat a objectes en el sentit que l'usuari crea l'aplicació sencera utilitzant les facilitats que CLAM ofereix a tals efectes (entrada sortida d'audio, visualització, protocol MIDI, serialització en XML, *multithreading*...)

L'actor 'desenvolupador', en canvi, exten la llibreria a base de crear nous components, entenent per components: algorismes encapsulats, dades que aquests utilitzen i visualitza-cions d'aquestes. Aquesta extensió es pot produir ja sigui creant els components desde zero, extenent-ne d'altres o bé creant-los per composició d'altres ja existents.

1.1.1.1 Un cas d'us concret

Un exemple d'un petit sistema que CLAM ha d'ajudar a implementar-se ràpidament: Ens demanen que d'escriure una petita aplicació en C++ (amb un màxim de 30 línies de codi d'usuari) per fer una anàlisi de l'espectre de l'audio (utilitzant l'algorisme *Fast Fourier Transform*) contingut en un fitxer (per exemple en format WAV), per tal de visualitzar l'audio en domini espectral. I també es demana que la manera de fer-ho sigui molt semblant a definir un graf de processos on les arestes són les dependències de dades entre una entrada d'un procés i la sortida d'un altre (*Data Flow Graph*).

A més, les dades que circulen pel graf han de poder ser de dos tipus diferents: audio en domini temporal i en domini espectral. Tots els processos han de permetre ser configurats amb gran detall, però també han d'oferir configuracions per defecte. Algunes sortides dels processos poden ser de caràcter canviant i altres constants. Per exemple, el procés *WinGen*, és un generador de finestres però genera un sol *token* de sortida i no pas un fluxe de *tokens*.

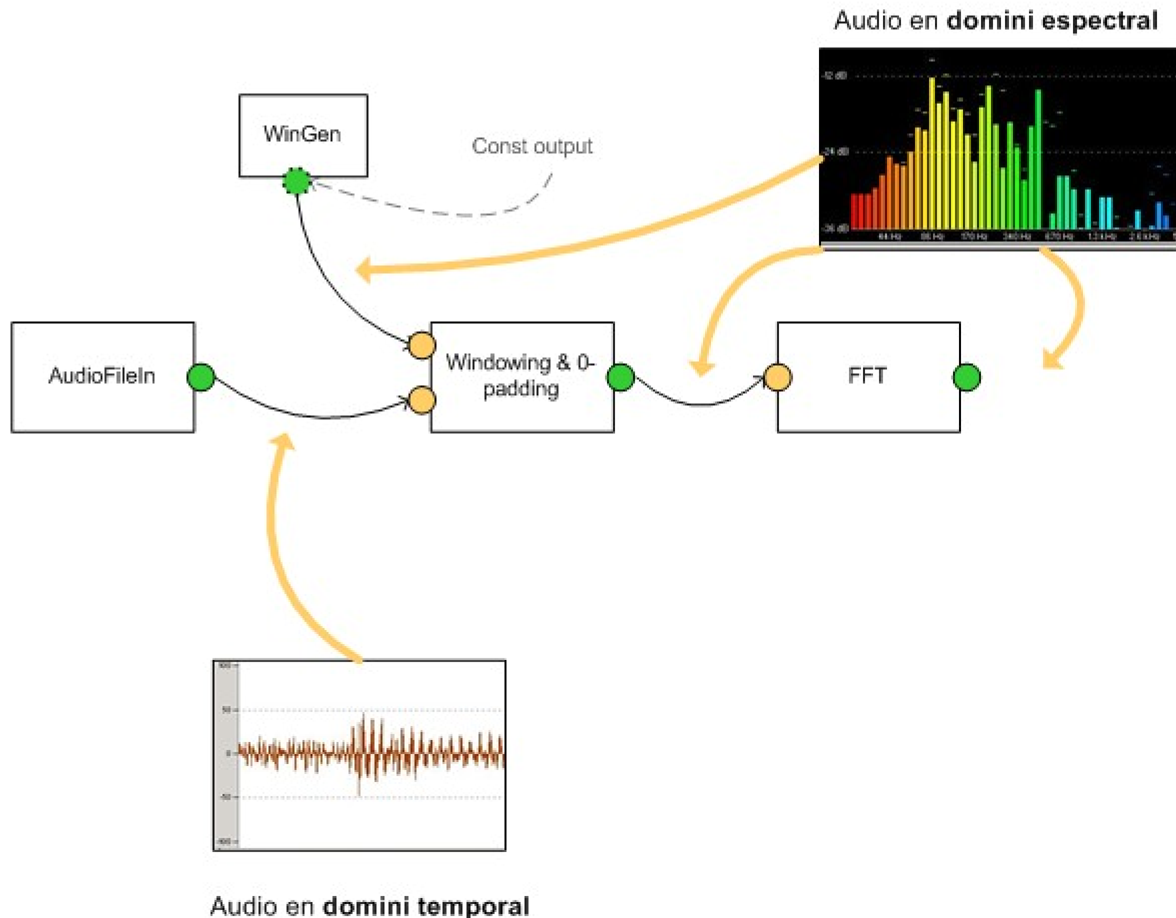

Figura 2. d'us concret: Transformada Ràpida de Fourier (FFT)

Per implementar aquest sistema utilitzant CLAM, tan sols haurem de:

- Derivar d'una classe que proporciona un esquelet d'aplicació, la qual gestiona el multithreading, i ens dóna un mètode a sobreescriure on posar el bucle del processament.
- Definir un sistema que contingui els quatre objectes de processaments de l'anterior figura, com a atributs, i també tants objectes de dades com arestes tinguem: un audio en domini temporal i dos espectres. Utilitzarem les configuracions per defecte, per tant no es preocuparem per configurar-los.
- Definim en un mètode d'inicialització, la connexió entre objectes de processament i les seves dades a través dels ports. I conectem el mòdul GUI adecuat a l'objecte de processament 'FFT'

- Implementem un bucle de processament molt senzill: a cada pas, simplement es fa una execució seqüencial dels mètodes `Do()` de tots els objectes de processament, amb l'ordre esquerra-dreta, segons el dibuix.

1.1.2 Dos modes de funcionament: supervisat i no supervisat

Ja s'intueix, amb l'anterior exemple, que aquest tipus de construccions es podrien fer utilitzant una aplicació gràfica (de l'estil de paquetes de simulació de models com ara *MATLAB-SIMULINK* [1]), i no directament programant amb C++. L'aplicació podria disposar d'una interície gràfica d'usuari (GUI) amb la qual l'usuari definiria xarxes de processos, escollint-los d'un repositori o llibreria.

Fins i tot, aquestes xarxes podrien ser definides de tal forma que poguéssin ser usades i guardades al repositori com si fos un nou procés individual.

L'execució de cada procés individual, així com la gestió de les dades que circulen per les arestes del graf, seria totalment controlat per l'aplicació de forma transparent a l'usuari (gràcies a un mòdul de *Flow Control*).

Aquesta aplicació està pensada, sobretot com una eina de prototipatge ràpid, i està prevista com la futura segona fase del projecte CLAM, i per tant, fora de l'abast del meu projecte final de carrera. En l'entorn de CLAM el mode en que l'usuari es pot despreocupar dels objectes de dades i del control del fluxe (explicitar les execucions dels processos) l'anomenem *mode supervisat,* per aclarar conceptes, cal dir que l'aplicació que he descrit haurà d'incloure tots els elements del *mode supervisat*, però aquest també té sentit d'existir sense l'aplicació.

De totes maneres, tant el meu projecte com la fase actual de CLAM implementen el *mode no-supervisat*. És a dir, cal una gestió manual de les dades entre processos i també un control del fluxe (decidir l'ordre de les execucions) explícit.

Tot i que en el meu projecte només he implementat el mode *no-supervisat* i la implementació del *mode supervisat* no és un requeriment, sí que ho és tenir-lo en compte en totes les decisions

de disseny, de forma que aquestes estiguin orientades a facilitar al màxim la futura incorporació d'aquest mode.

1.1.3 Punt de partida del Projecte

El grup de tecnologia musical de l'IUA-UPF, és conegut per la seva recerca sobre síntesi amb modelatge espectral [2], i altres tecnologies relacionades. Concretament, desde la seva fundació al 1994, s'han dut a terme projectes entre els que s'inclouen: sintetitzador de veu cantada, eina de morphing entre timbres de veu aplicat a *karaokes*, identificació automàtica de cançons radiades i extracció de contingut d'alt nivell (cognitiu) de l'audio.

A la tardor del 2001, pel començament del projecte, ja es comptava amb un bon nombre d'aplicacions software. Però es feia evident que al no haver-se dissenyar per una posterior reusabilitat, era molt complicat adaptar-les per cobrir les necessitats de nous projectes i les noves eines software.

El meu projecte final de carrera, és de caire totalment d'informàtic, en el sentit que es tracta de dotar d'eines i infrastructura sofware (i manera de fer el software) a un grup d'investigadors, però en canvi el domini és el del tractament del senyal i la música.

Per això considero important fixar el punt de partida del projecte: Per una banda, es tracta d'aprofitar l'experiència d'un grup d'expert en tècniques de tractament de l'audio (sobretot en modelatge espectral) i molt motivat a col·laborar en el projecte CLAM, i per l'altra, jo tampoc partia d'una desconeixença del domini: tenia uns coneixements musicals, i estava molt interessat i familiaritzat amb tècniques de gravació i edició de l'audio, així com en usar eines (tant hardware com software) de sínitesi d'audio.

1.1.4 CLAM com a part d'una distribució lliure de *Linux*

El projecte CLAM forma part d'un projecte europeu IST, *Agnula*, una distribució del sistema operatiu *Linux* orientada a estimular el desenvolupament d'aplicacions multimèdia lliures i de codi obert. Aquest fet comporta que la llibreria serà publicada sota la llicència *GPL,* o *General Public License*, el que es coneix informalment, com a *software lliure*.

Es pot dir que actualment (Juny del 2002) la llibreria està prou madura com per ser entregada públicament, cosa que està planejada ser duta a terme el pròxim mes de Novembre.

1.2 Definició del Projecte

Per realitzar l'informe de definició del projecte s'ha usat la metodologia *METHOD-1* [ref GPI, 1999], desenvolupada a l'empresa Andersen Consulting. No s'ha fet un ús estricte i complert de la metodologia sinó que s'ha adaptat a les necessitats del projecte.

METHOD-1 està orientat, principalment, a sistemes d'informació per a l'empresa i insisteix molt en certs aspectes econòmics com ara: l'impacte econòmic a l'organització, els costots i beneficis del sistema a construir. Aquests aspectes econòmics no s'inclouen a l'informe de definició ja que no s'adeqüen a un projecte el qual es troba més proper a l'àmbit de recerca, tot i això, a l'últim capítol §2.4 de la memòria es realitza una anàlisi econòmica del projecte.

1.2.1 Motivació

La raó del projecte està fonamentada en el fet que no existeixi encara cap framework orientat a objectes que compleixi els principals requeriments. En línies generals , els productes similars (deixant de banda les seves condicions comercials, i condicions de plataformes disponibles) o bé pequen per ser poc flexibles, i per tant, no permetren implementar cert tipus de models, o bé pequen per ser poc especialitzats al domini de l'audio, i per tant, no són capaços de tenir un comportament eficient. Per exemple per fer transformacions en temps real.

Típicament, el primer cas (poca flexibilitat) es deu a :

- No permetre diferents tipus de dades. Normalment treballen amb audio en domini temporal (*samples*) i, com a molt poden treballar amb domini espectral (*spectrums*), però no permeten extendre els tipus a , per exemple estructures de dades que representin descriptors d'alt nivell.

- No permetre diferents *granularitat* d'aquestes dades. En el mateix graf de processament no poden conviure processos que produeixin diferent nombre de *tokens*.

- Poca flexibilitat en quant al tipus de processament. No permeten *re-alimentacions* (*feedbacks*), processament inplace . A §1.3.2 es defineixen aquestes conceptes.

- Dónen un suport poc orientat a objectes i de molt baix nivell, la qual cosa impossibilita l'ús com a eina de prototipatge ràpid.

El segon cas (poca eficiència) se sol donar per:

- Al ser una èina general de simulació de models, es fa una simulació d'un model descrit amb funcions matemàtiques, sense treure profit del caràcter discret de l'audio digital, la qual cosa sol comportar operacions simbòliques (derivacions) en comptes d'una execució dels algorismes dels processos.

- No s'obté un codi compilable (o bé aquest és poc eficient).

- No estan pensats per integrar-se en entorns de temps real i d'interaccionar amb protocols com el MIDI.

Així doncs, CLAM ha estat pensat per suplir aquesta mancança d'un sistema d'arquitectura oberta i d'una gran flexibilitat, sobretot en tipus de dades i de flux, però sense comprometre la seva fiabilitat i eficiència.

En l'estat actual del desenvolupament de CLAM, podem dir que ja s'ha aconseguit..

1.2.1.1 Comparació amb sistemes similars

Existeixen moltes lllibreries o sistemes en general que permeten definir i executar processaments sobre l'audio i la música. Al meu entendre, els més reexits són (sense ordre): Simulink, Max/jMax/PD, Siren Music, Open Sound World, Maate, Reaktor, WaveWarp, AudioMulch, OSALP, Overflow, SonicFlow, SoundProcessingKit, SymbolicSound. De tots ells es troba informació molt fàcilment per internet.

D'aquests, n'escullo 4 com a representants de diferents estils de sistemes i els comparo 12 característiques importants, junt amb l'actual versió de CLAM (*mode no-supervisat*) i la futura versió amb *mode supervisat*.

	Simulink	jMax	Sonic Flow	AudioMulch	CLAM no-sup.	CLAM superv
Processat a temps real?	N	S	S	S	S	S
Extensible programant en C++ ?	N	N	S	S	S	S
Té mode aplicació?	S	S	N	S	N	S
Escalable?	S	S	N	N	S	S
OpenSource?	N	S	S	S	S	S
Manega buffers d'audio de mida variable?	S	N	S	N	S	S
Manega audio en domini espectral?	S	N	S	N	S	S
Tipus de dades extensible?	N	N	N	N	S	S
Té control de fluxe per defecte?	S	S	N?	S	N	S
c.f. dependent de la topologia?	N	S	-	N	N	N
Fa simulació amb mètodes simbòlics?	S	N	N	N	N	N
Projecte actualitzat?	S	S	N	S?	S	futur

1.2.2 Objectius

1.2.2.1 De CLAM

Com he explicat a l'introducció, els requeriments explícits de què disposavem els desenvolupadors de CLAM eren de caràcter molt general. La metodologia en espiral que hem seguit ha fet que anéssim investigant els nous requeriments concrets a cada iteració per incorporar noves funcionalitats. A vegades aquests sorgien d'algú de fora del grup i a

vegades de dins el propi grup, i per tant és molt difícil traçar la línia entre aquests requeriments concrets i les decisions de disseny. Podriem dir que es tracta d'una llibreria que ha seguit un model de desenvolupament obert.

Es preveu que encara s'incrementi més la tendència al desenvolupament obert en un pròxim futur, donat que la llibreria passarà a ser Open Source distribuït sota la llicència GPL i els futurs usuaris podran col·laborar tant en la incorporació de nous components de processament a la llibreria, com en aportar idees i necessitats de noves funcionalitats.

En aquest capítol, però em centro en els requeriments explícits i generals. Els classificaré en funcionals i no funcionals. I a l'apartat de conclusions §2.6.1 evaluaré si han estat complerts.

Requeriments funcionals generals.

Desenvolupar una llibreria (o framework) Orientada a Objectes, complerta, flexible, expandible i reusable per satisfer les necessitats de tots els projectes actuals i futurs de l'MTG.

Requeriments no funcionals

Programada en **C++** i **multiplataforma** (Windows, Linux i MacOS)

Que es pugui utilitzar tant com a eina de **prototipatge ràpid**, com també per desenvolupar aplicacions **eficients** (incloent procéssos a temps real).

Que la llibreria permeti desenvolupar sistemes **robustos**, **fiables,** que utilitzin tant el codi com la metodologia de la llibreria i que, del seu desenvolupament, se n'obtinguin nous components **reusables**. A més, que minimitzi el temps de desenvolupament d'aquestes sistemes o aplicacions.

Els requeriments funcionals en més detall:

Considero que són funcionals perquè es tracta d'una llibreria, però en cas d'una aplicació serien vistos com a no-funcionals.

- **Complerta**: que contingui totes les utilitats típiques del processat d'àudio, entrada-sortida (dades d'àudio, MIDI i altres), passivació/activació de la informació que flueix pel sistema (utilitzant formats XML i SDIF, etc.), oferir una capa de visualització (GUI), que sense tacar el model de processament, dóni la possiblitat de sincronitzar amb el model vistes construïdes amb múltiples toolkits gràfics com QT, FLTK, OpenGL...

- Construcció **modular**: que un sistema es pugui construir a base d'escollir mòduls (de processament, de control, etc.) i establir connexions entre ells. Així com també es pugui prescindir de certs mòduls si no són necessaris.

- **Escalable**: que una xarxa de mòduls en puguin formar un de nou.

- Permetre que el fluxe de dades tingui una **gran diversitat de tipus de dades**. (dades en domini temporal, en domini espectral, descriptors de segments d'audio, etc.), com també d'agregacions d'aquestes dades.

1.2.2.2 Del Projecte

Els objectius del meu projecte final de carrera són els mateixos de la part més central de CLAM, deixant fora l'abast del meu projecte tot el què és especific del suport XML, del mòdul de visualització gràfica (GUI), de l'entrada i sortida d'audio, del protocol MIDI, i dels algorismes del processament del senyal.

1.2.3 Visió General del Sistema

1.2.3.1 Model Conceptual

El model conceptual[1] descrit serveix com a primera aproximació de l'especificació del sistema. És, per tant, el punt de partida per obtenir el domini, el qual consta d'un subconjunt d'entitats del model conceptual (la part que es vol informatitzar).

Concretament, el domini tracta sobre el tractament del so i la música, i per tant, les entitats que apareixen al gràfic descriuen les parts que intervenen en la manipulació de la representació d'un so o música feta per una persona.

[1] La notació usada pel Model Conceptual es basa en SYNTHROPY [3]

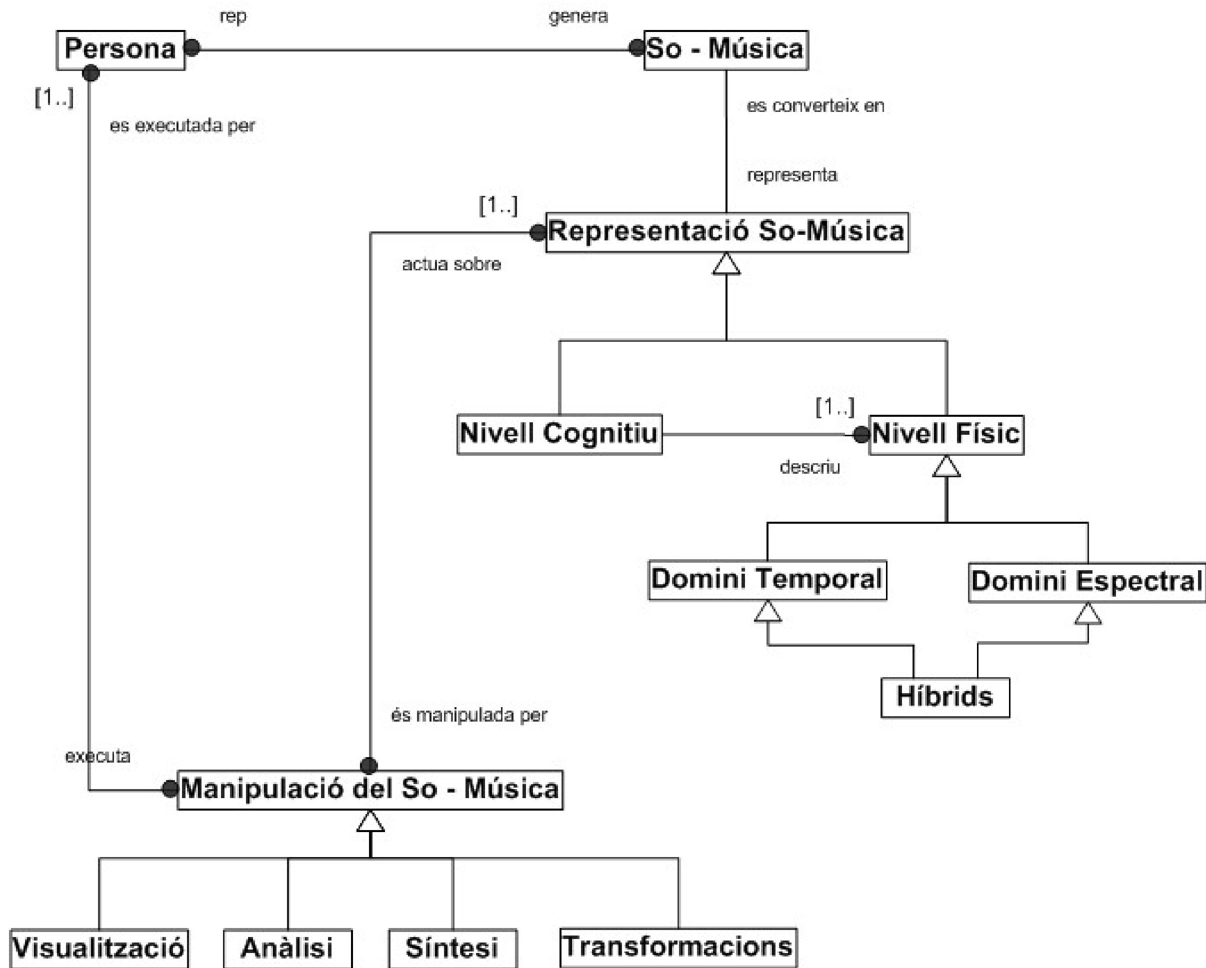

Figura 3.model conceptual del domini. Usant notació SYNTHROPY

La interpretació textual del model conceptual és la següent:

El so o la música generada per alguna persona és convertida a una representació. Aquesta pot ser de caràcter físic (permet la perfecta reproducció original del so) o bé de nivell cognitiu (expressa coses que tenen sentit per les persones). Una representació a nivell cognitiu sempre descriu una o més representació a nivell físic. El so a nivell físic es pot descriure en el domini temporal o en l'espectral, o també pot ser una representació híbrida.

Una o més persones executen (en un sentit abstracte) una manipulació sobre aquesta representació del so-música. Les manipulacions poden ser de diverses espècies: una simple visualització de la representació (sigui aquesta del tipus que sigui), anàlisi i síntesi (les quals

poden generar una representació de nivell cognitiu a partir d'una de nivell físic, i a l'inversa), o bé transformacions en general de les representacions.

Aquesta representació manipulada es torna a convertir en so o música la qual pot arribar a una o moltes persones, tant si aquestes la reben de forma activa o no.

1.2.3.2 Arquitectura

Donat que es vol construir una base comuna per el desenvolupament dels diferents projectes del grup, no només és útil cobrir les necessitats purament de processament sinó que s'intenta donar una solució reusable a altres camps de les aplicacions com ara l'entrada i sortida, la representació gràfica, el *multithreading*, i la interacció amb perifèrics musicals.

Tot i que CLAM preten ser més que una biblioteca de processament, i convertir-se en un framework complert per aplicacions de processat d'audio i música, la part central de processament s'ha mantingut independent de la resta perquè es pugui integrar dins d'una aplicació que ja doni suport a aquests aspectes accessoris.

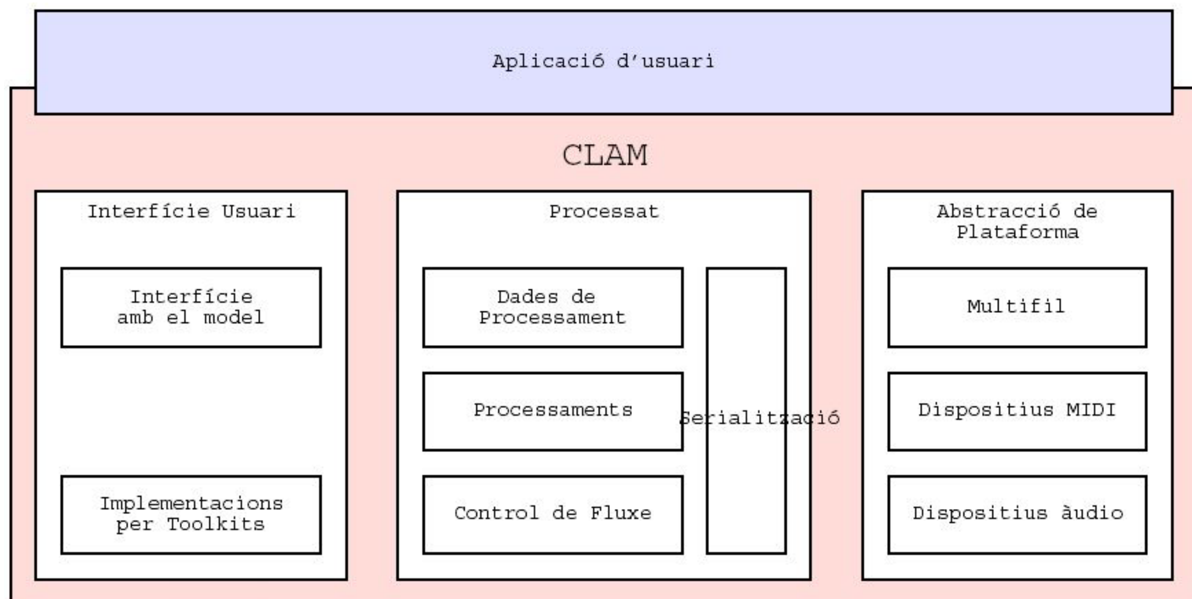

Figura 4. Blocs arquitectònics de CLAM

1.2.3.3 Necessitats de dades

Les dades que necessitarà manegar qualsevol sistema basat en CLAM les podem classificar en dos tipus: d'entrada i sortida, i de configuracions

- Les dades d'entrada i sortida inclouen fitxers d'audio en qualsevol format, dades d'audio en *streaming* originades i finialitzades a una targeta de so, fitxers de dades estructurades del procés (en XML, SDIF...)....

- Les dades de configuració, que no té a veure amb l'audio i música processada, sinó amb informació / paràmetres de configuració del sistema: definicions de xarxes de processos, configuracions de processos. Aquestes dades sempre es quardaran en XML

1.2.3.4 Infrastructura Informàtica

Al ser CLAM multiplataforma, podem implantar el sistema en qualsevol estació de treball amb les plataformes següents: Win9x/2000/XP, Linux (en PC i MAC) i en un futur MacOSX.

Els requeriments mínims de l'estació de treball dependran del tipus d'ús que volguem fer de la llibreria. En termes generals, per processar audio a temps real, necessitem una CPU ràpida, molta memòria RAM (>128Mb), i una tarja de so bona (sobretot per la qualitat dels drivers).

1.3 Requeriments sobre el flux de dades.

El flux en un sistema de processament determinat per un graf de processaments interconnectats, el determina l'ordre d'execució dels elements de processament i la forma en que les dades es generen, passen d'un element a un altre i es modifiquen.

Com s'ha vist abans CLAM permetrà dos modes d'operació:

- Un *mode supervisat* on el fluxe és totalment gestionat per la llibreria o framework
- Un de *no supervisat*, on el programador és qui nodreix als elements de procéssament amb les dades, i determina l'ordre d'execucions.

Si bé el disseny de CLAM pot ser complex degut a l'heterogeneïtat de les dades que ha de gestionar, ho és molt més pels requereiments que té el flux d'aquestes dades.

Cal dir que en *mode supervisat*, com que el control del fluxe és programat específicament per cada sistema i per tant s'és tant flexible com es vulgui (a costa d'un cost molt superior en la programació, apart que sovint cal ser expert en processament del senyal), aquests requeriments sobre el fluxe de dades es poden considerar assolits.

1.3.1 El futur mòdul de flow control

Molt més interessant és pensar en automatismes del control de fluxe que permetin complir tots els patrons de fluxe requerits. El disseny de tal mòdul, anomenat Flow Control no és gens trivial.

Actualment estic treballant en aquest disseny. Començant, pel què anomenem **nodes**, que són les entitats encarregades de manegar la memòria que circula pels arcs del graf de processament (ja es disposa d'una primera implementació que s'aproxima a la soluciò).

Dissortadament no està prou madur per poder ser presentat com un mòdul disponible a CLAM, i tampoc en la present memòria.

La solució que estic adoptant utilitza idees de *models de fluxes de tokens* [16] (també anomenats *fluxes de dades síncrones* [17]). Especialment en l'apartat de mètodes i algorismes per obtenir un *scheduling* per l'execució dels processaments. Pel que he vist fins ara, aquests són perfectament aplicables als grafs de dependències de dades dels sistemes de processament de CLAM, i fins i tot permeten trobar l'*scheduling* òptim (ja sigui optimitzant pel temps d'execució com per les dades (buffers) intermitges usades.)

He decidit posar alguns documents de desenvolupament que hem escrit sobre el tema, al segon apèndix. (Apèndix B). Tot i no estar prou madur com per formar part de la memòria, considero que és una part prou important i a la qual hi he dedicat prous esforços. Potser val la pena fer-hi una ullada.

1.3.2 Alguns patrons de flux:

El flux més simple que cal suportar és aquell en que tots els processos consumeixen i produeixen dades al mateix ritme. En aquest cas, simplement cal executar els processos en ordre tal que cadascun s'executi amb dades disponibles.

Re-alimentacions

El patró anterior necessita algunes modificacions quan introduïm alguna re-alimentació, que són molt comunes als sistemes de processament. Les re-alimentacions no disposen de dades per les primeres execucions i cal donar-ne per defecte.

Les re-alimentacions també compliquen els algorismes per determinar l'ordre d'execució a partir de la topologia. Fins i tot, hi ha topologies que poden arribar a esdevenir ambigües com, per exemple, la que es mostra a la següent figura. Els tres casos es basen en la mateixa

topologia, però, s'obté resultats diferents segons es consideri quina connexió és la realimentació.

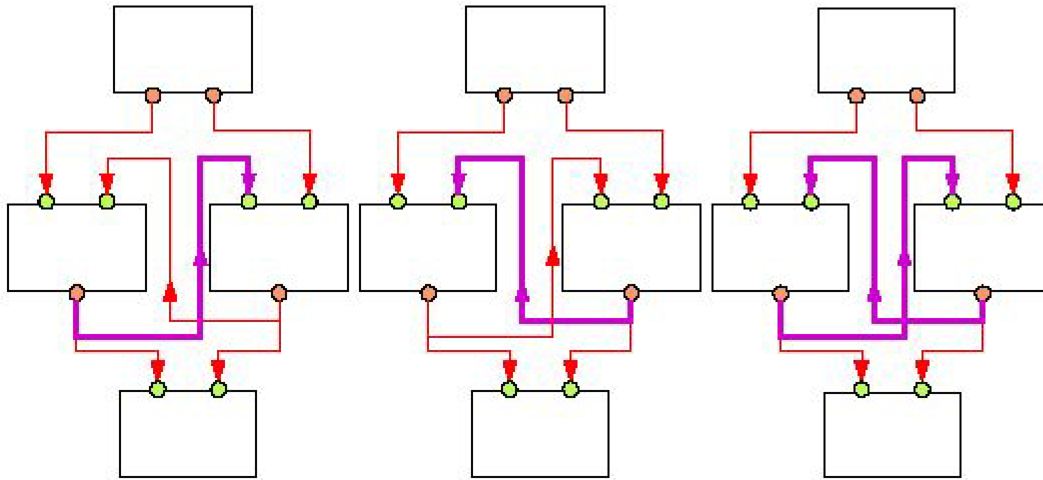

Figura 5. Topologia amb realimentació ambigua

Flux multirate

El següent refinament que ens podem trobar és el fluxe *multirate*. Es dóna quan tots els elements de procés no produeixen i consumeixen dades amb la mateixa proporció a cada pas d'execució. Per assegurar que a cada pas hi hagi dades, cal executar els passos de cada procés amb diferent freqüència.

Flux variable

El cas extrem en aquest sentit és aquell en el que el nombre de dades consumides o produïdes varia d'uns passos d'execució a uns altres. El flux variable obliga a mantenir certa intel·ligència en temps d'execució que, si només haguéssim de fer servir flux fix, seria només necessària en temps de configuració del sistema.

Inplace processing

En els casos anteriors es considerava que les dades d'entrada només es llegeixen i no es modifiquen. Les dades de sortida són dades verges a les que el procés dóna un valor. Això va bé perquè es poden fer moltes suposicions de cara al flux, però no sempre és la millor opció.

Quan les dades són estructures complexes, i el processament consisteix en obtenir l'entrada modificada, aquest tipus de flux esdevé molt ineficient perquè a cada execució de l'algorisme cal tornar a crear l'estructura de la dada.

La solució és fer que la dada d'entrada i sortida sigui la mateixa modificant-la directament. Això ho anomenem *inplace processing.*

L'*inplace processing* complica relativament el flux atès que cal controlar l'accés a la dada i modelar les dependències de dades que es generen, de forma molt semblant a com ho faríem als registres d'una CPU. Hi ha un procés que genera la dada, un que la llegeix, un altre que la modifica i un altre que llegeix la modificació. Cal ordenar l'execució d'aquests processos per no crear inconsistències.

Recàlculs

Els recàlculs es fan servir normalment per corregir les suposicions que es fan durant els transitòris d'un so.

Per exemple, si volem fer un processament sobre el so d'un instrument que es basa en el pitch de la nota, probablement en els transitoris, quan comença una nota, farem suposicions equivocades del pitch i els càlculs seran erronis.

Per obtenir un millor resultat, convé refer els càlculs un cop s'ha detectat l'error.

Els recàlculs són complexos de gestionar amb un control de fluxe genèric però cal implementar-los perquè, en donar bons resultats, són comuns als sistemes que CLAM ha d'implementar.

Flux assíncron (controls)

El flux assíncron és aquell que no es produeix de forma contínua sinó en forma d'events.

Els processos es poden comunicar informació assíncrona entre ells, però no ho fan de la mateixa manera que es comuniquen les dades sinó fent servir el mecanisme de controls que s'explica a §2.3.4

1.4 Conceptes de disseny del software i eines escollides

En el disseny de la llibreria CLAM, diverses tècniques que tenen a veure amb l'enginyeria del software han estat intensivament emprades. En aquest capítol s'expliquen algunes d'aquestes eines, metodologies i tècniques.

1.4.1 Orientació a objectes

Avui en dia ignorar la programació orientada a objectes és com ignorar la programació estructurada o els llenguatges d'alt nivell. Qualsevol projecte d'una certa mida necessita beneficiar-se dels avantatges que ofereix aquest paradigma de programació. [14]

- Millora la comprensió del sistema: La modelització d'una aplicació amb objectes i relacions d'herència, composició i referència entre ells són eines cognitives naturals en els humans.
- Permet gestionar millor la complexitat: L'encapsulació de les dades darrera d'una interfície redueix el nombre de connexions entre les diferents parts de l'aplicació. També facilita el canvi reduint i localitzant els lligams entre les diferents parts del sistema.
- Dona extensibilitat i adaptabilitat al sistema: L'herència permet extendre o modificar els objectes existents i, gràcies al polimorfisme, es poden fer servir els nous objectes on hi havien els antics sense modificacions.
- Permet la reutilització de codi: Per una banda, el disseny orientat a objectes permet mantenir una bona ocultació de les dades i els detalls, així com minimitzar

l'acoblament entre les classes, la qual cosa facilita molt la reutilització de classes ja programades. D'altra banda l'herència permet fer servir codi d'una classe en una altra.

Quan parlem d'orientació a objectes, cal tenir en compte amb quina visió es parla. Cada llenguatge de programació en fa la seva visió. Anomenem idioma a cadascuna de les formes de treballar amb un mateix llenguatge. Fins i tot, en cada idioma la visió de l'orientació a objectes és diferent.

Més que buscar què és concretament l'orientació a objectes, Stroustrup, el pare del C++, dóna una visió més alliberadora [3]. Diu que no és important cercar la orientació a objectes, sinó cercar aquelles eines que siguin útils pels desenvolupadors siguin, no siguin, o no s'estigui segur de si són, orientades a objectes.

1.4.2 Patrons de disseny

Molta de la feina del disseny de CLAM i d'aquest PFC ha vingut reforçada per l'anàlisi de patrons de disseny aplicats als diferents problemes trobats. El concepte de patró de disseny ha estat popularitzat arran del llibgrte d'Erich Gamma et al. [4], on s'estableix formalment el concepte i es defineixen amb exhaustivitat i de forma molt pràctica els patrons de disseny més comuns.

Un patró de disseny és una abstracció d'un problema freqüent que sovint es soluciona de manera similar. En fer l'analogia entre l'abstracció i el problema objectiu, s'identifica ràpidament els actors i les forces, i es pot avaluar si la solució oferta pel patró és acceptable pel cas de disseny.

Avança molt la feina d'avaluació, perquè, a l'especificació dels patrons de disseny, es remarca quins factors clau fan que un patró sigui l'idoni o no, tot i haver aconseguit fer l'analogia.

Els patrons de disseny són eines per transmetre i adquirir experiència en la resolució de problemes de disseny. Evidentment not tots els problemes són iguals, però de fet, el què més

els diferencia és el domini i contexte concret del disseny, en canvi existeixen certes analogies entre ells que en abstreure-les esdevenen reaprofitables. No s'està parlant de reaprofitament de codi, sinó de reaprofitament de disseny.

Els patrons de disseny són també molt útils donat que ofereixen al grup de dissenyadors un llenguatge comú amb que comunicar-se idees de disseny

A el llibre de Gamma et al. [4] es descriuen alguns patrons de disseny en l'àmbit del disseny orientat a objectes. Aquests patrons de disseny els classifiquem en tres grans famílies:

- **Patrons de construcció**: Resolen com es creen els objectes.
- **Patrons d'estructuració**: Resolen com es combinen els objectes.
- **Patrons de comportament**: Resolen la manera de modelar el comportament.

El concepte de patró de disseny també s'ha aplicat a altres àmbits de l'enginyeria del software, que no només al disseny orientat a objectes. Per exemple, hi ha bibliografia sobre patrons de disseny per la programació orientada a components o a agents distribuïts.

1.4.3 Programació genèrica

Si bé, la programació orientada a objectes o la programació orientada a components han promogut la reutilització del codi, aquesta arriba només fins a cert nivell.

El problema amb la programació orientada a objectes a la majoria de llenguatges és que es lliga massa els algorismes amb les dades que maneguen, més del que els conceptes requereixen. És a dir, amb orientació a objectes pura no es pot programar un algorisme independentment del tipus de dada que s'està manegant.

La programació genèrica permet programar algorismes i definir tipus abstractes de dades, centrant-se en els conceptes que representen les dades i no pas en els tipus concrets.

1.4.3.1 Programant amb conceptes

Igual que la programació orientada a objectes respecte a la programació estructurada, la programació genèrica no substitueix pas la programació orientada a objectes sinó que l'expandeix.

La programació genèrica és una forma de programar que es basa només en els conceptes que representen els símbols. Un concepte és una família d'abstracions que acompleixen un seguit de requeriments comuns, entenent per abstraccions tipus de dades o classes concretes.

Per exemple, es pot definir el concepte *ordenable* com aquell que agrupa totes les classes en les que es pot establir un ordre entre les seves instàncies mitjançant un operador <.

Hom pot dissenyar un algorisme tot suposant quelcom sobre aquest concepte. Per exemple, dissenyem un algorisme per escollir el major element de dos donats. Aquest algorisme serà aplicable a un parell d'objectes de qualsevol classe que pertanyi al concepte *ordenable*.

1.4.3.2 Parts d'un concepte

Parlant d'una forma més formal, un concepte és un conjunt de requeriments que un tipus concret pot satisfer. Aquests requeriments s'adrecen a quatre punts de vista:

- Conjunt d'expressions vàlides: Aquestes expressions són les que s'han de poder fer servir amb els objectes del tipus que satisfà aquest concepte.

- Tipus associats: Quan un tipus modela un concepte, hi ha un seguit de tipus associats que adopten rols concrets segons el concepte. El concepte pot exigir que aquest tipus associat també modelitzin altres conceptes.

Per exemple, creem el concepte *contenidor ordenat* amb dos rols associats: *elementBasic* i *iterador* que requereixen respectivament complir els conceptes *ordenable* i *iterador*.

Quan un vector de nombres reals modela el concepte *contenidor ordenat*, automàticament els nombres reals modelen el concepte *ordenable* i l'iterador que ens proporciona el vector modela el concepte *iterador*.

- **Invariant**: Els invariants tenen més a veure amb la semàntica de les expressions vàlides i es modelen amb precondicions i postcondicions. Els invariants necessiten un llenguatge formal per poder-se expressar i verificar.

- **Garanties de complexitat**: de cara a la usabilitat es garanteix per cada operació una complexitat asimptòtica màxima.

Aquest darrer grup de requeriments és difícil de comprovar. De fet només s'afegeix com a part de la documentació per que els usuaris d'una classe coneguin que aquests requeriments es compleixen

1.4.3.3 Implementacions

Els diferents llenguatges de programació han arribat a diferents proximacions per implementar parcialment els conceptes. És a dir, fan servir diferentes estratègies per reaprofitar codi perquè funcioni per a un conjunt de classes que tenen alguna cosa en comú.

A la majoria de llenguatges orientats a objectes, el punt feble és la verificació de que els objectes genèrics entren dins el concepte.

Smalltalk [5] tracta tots els objectes com a igual. Un programador d'Smalltalk pot passar qualsevol objecte com a paràmetre d'un mètode. És en fer servir aquest objecte dins el mètode, quan li enviem un missatge, que es comprova si hi ha cap mètode associat amb el missatge enviat, éa a dir, es comprova si pertany al concepte que s'esperava. Però això es fa en temps d'execució i, si l'objecte no s'escau al concepte, el programa dóna un error d'execució.

En resum, en temps de compilació, Smalltalk no fa cap comprovació de tipus, la qual cosa possibilita la programació genèrica, però, tampoc fa cap comprovació de concepte, la fa en

temps d'execució. Això deriva en errors en temps d'execució no gaire controlables i que es podrien haver detectat en compilació.

C++ sí que té comprovació de tipus en temps de compilació, de fet, aquest fet impossibilitaria la programació genèrica si no fos per les classes abstractes i els *templates*.

Es poden implementar conceptes amb classes abstractes. Qualsevol instància d'una subclasse d'una classe abstracta es pot passar com a paràmetre a qualsevol lloc on es demani un punter o una referència a la classe abstracta. L'ús d'aquesta instància és segur en temps d'execució donat que la instància de la subclasse ha d'implementar la interfície de la classe abstracta. Això es comprova en temps de compilació.

Aquesta aproximaciò, però, té diversos problemes:

- Cal fer la relació del concepte amb la classe de forma explícita. Objectes que s'avinguin al concepte, però que no derivin de la classe abstracta no es poden fer servir.
- La definició de la classe esdevé complexa, donat que cal derivar de tots els conceptes als que pertany,
- Els conceptes poden tenir parts en comú i en quest cas es complica la resolució de l'herència.
- Els tipus bàsics no poden encabir-se en cap concepte donat que no són classes que puguin derivar-se.
- La resolució dels mètodes virtuals es fa en temps d'execució i, encara que ara no provoca errors, si que no es pot resoldre de forma *inline*[1].

Java millora alguns d'aquests punts amb els *interfaces*. Els *interfaces* es fan servir només en el sentit que necessitem: especifiquen uns requeriments d'interfície. No hi ha cap

[1] un mètode inline és aquell que el compilador expandeix codi de baix nivell en el punt on es fa la crida, de forma que es guanya eficiència en temps d'execució, en contra, s'augmenta la mida del codi.

implementació i la resolució de l'herència és més senzilla. Tot i així, també cal explicitar a la declararció de la classe quins conceptes s'hi avenen amb el tipus.

Els *templates* es varen introduir al llenguatge C++ precisament per poder fer programació genèrica. Són una aproximació al problema dels conceptes complementària a l'ús de classes abstractes. Els *templates* són declaracions parametritzades, on els paràmetres de la declaració són tipus o constants sencers (*integers*). En comptes de declarar una classe o una funció, estem declarant un motlle d'on sortiran diverses funcion o classes segons els valors.

En comptes de forçar la subclassificació i la implementació d'un *interface* complert, es pot programar un algorisme o una classe *template* tot suposant que el paràmetre amb el que es farà servir, compleix l'interfície requerida.

Els templates aconsegueixen l'objectiu: implementar algorismes i estructures de dades d'una forma conceptual. Però, d'alguna manera, aquesta implementació té alguns defectes que dificulten tant el desenvolupament d'una lliibreria com el seu ús.

- Cap declaració és instanciada fins que no es fa servir. És a dir, el compilador trobi un troç de codi on es concretitza el paràmetre template. És molt difús saber si una declaració template és correcta fins que no es fa servir directament a nivell de símbol. És a dir, quan fixem el paràmetre template amb un símbol (classe o enter).

- Quan les classes templates pertanyen a una llibreria, els errors de compilació de codi d'usuari, els dóna a dins de la llibreria.

- Els símbols que representen declararcions template són costosos d'entendre.

- Els errors de compilació no són gaire clars si no s'està molt acostumat a fer servir templates.

A això cal afegir-hi el fet de que com que és una funcionalitat del llenguatge que s'ha estandaritzat recentment, la majoria de les implementacions són inconsistents entre elles i moltes tenen greus errors en el seu tractament.

Com que, en l'actualitat, cap implementació de C++ disponible implementa la paraula clau export que possibilitaria separar les declaracion i les implementacions dels símbols templatitzats, els temps de compilació d'una aplicació que faci servir la llibreria són molt llargs i els executables es fan molt grans.

1.4.4 Un llenguatge de programació multiparadigma : C++

La decisió d'utilitzar C++ com a llenguatge d'implementació de la llibreria ja ens va venir donada. Però és fàcilment justificable perquè es tracta d'un llenguatge que permet obtenir solucions optimes en quant a l'eficiència.

Stroustrup considera [3] que el C++ és un llenguatge multiparadigma, en el sentit que no obliga a lligar-se a un sol estil. Permet fer programació a baix nivell, orientació a objectes, i programació genèrica.

Un altre aspecte important del C++ és que es tracta d'un estardard ISO mentre que d'altres com el Java o el C-sharp són propietaris de companyies privades.

1.4.5 Standard Template Library

La programació genèrica ha pres importància a l'escena de la programació en C++ arran de la introducció a l'estàndard de C++ de l'especificació de la STL (*Standard Template Library*). La STL és la llibreria que fa servir programació genèrica amb templates per oferir un seguit d'abstraccions i mecanismes que faciliten la codificació.

A més de fer servir les classes genèriques de STL, en molts casos CLAM ha adoptat la seva forma de fer les coses, donat que les diferents implementacions fan servir algunes solucions molt intel·ligents a alguns problemes que nosaltres mateixos ens hem trobat.

Els elements més utilitzats de la STL són els contenidors genèrics. Un contenidor és una estructura de dades que emmagatzema elements del mateix tipus, per exemple, una llista, un vector, un arbre, una pila, etc.

Aquestes estructures tenen diferent organització interna. Per poder abstraure algorismes que puguin aplicar-se als contenidors d'una forma genèrica, es fa servir entre d'altres coses el concepte d'iterador.

Un iterador és quelcom que ens serveix per accedir als diferents elements d'un contenidor. Els contenidors han de proveir mètodes per obtenir iteradors i els algorismes han de saber manegar-los. D'aquesta manera, es poden desacoblar les dependències d'implementació entre els algorismes i els contenidors.

1.4.5.1 Iteradors

Els iteradors són objectes que es defineixen per a cada contenidor concret però que ofereixen una mateixa interfície estàtica. A més, pot encapsular, cada un, diversos tipus de recorregut sobre una mateixa estructura.

La STL modela els iteradors imitant la interfície que es fa servir per accedir a una estructura d'array amb punters de C, sent els punters els iteradors de l'estructura.

Amb una estructura array de C faríem:

```
void strncpy (char * origen, int n, char * desti)
{
    char * finalOrigen=origen+N;
    while (origen!=finalOrigen)
        *desti++ = *origen++;
}
```

Per uniformitat i per economia de conceptes, les operacions que es fan servir són les mateixes que les dels punters:

- Els iteradors s'incrementen i decrementen amb els operadors ++ i -- de forma prefixa i posfixa per avançar o retrocedir un element.

- També els podem sumar o restar un sencer N per avançar o retrocedir N elements.

- Per accedir al valor actual fem servir l'operador unari `*`, si volem accedir a un camp del valor, també podem fer servir l'operador `->` , tot igual que si fos un punter.

- Amb els operadors `==` i `!=`, podem comparar dos iteradors per veure si apunten al mateix element o no.

De tal forma que ara podem implementar la mateixa funció usant iteradors:

```
template <class iterator>
void copy_n(iterator origen, int N, iterator desti)
{
    iterator finalOrigen = origen + N;
    while (origen != finalOrigen)
         *desti++ = *origen++;
}
```

L'avantatge d'aquest algorisme en front del primer és clar: mentre el primer estava lligat a un array C de caràcters, el segon és genèric i el podem aplicar a dos contenidors qualsevols que tinguin el concepte associat iterador.

Alguns iteradors, però són més restrictius envers les operacions que poden realitzar. Segons aquestes restriccions es defineixen diverses categories d'iteradors. La funció `iterator_category(iterator)` retorna un punter nul del tipus de tag que correspon amb la seva categoria, que passo a anumerar:

- **output_iterator_tag**: Només es pot fer una sola inserció de valor entre increment i increment. No es pot llegir el valor apuntat, només escriure-hi.

- **input_iterator_tag**: Es pot extreure una posició tantes vegades com vulguem mentre no retorni l'element singular que indica que no podem extreure'n més. Es pot incrementar i llegir d'una altra nova posició però, una vegada incrementat, les altres còpies de l'iterador es tornen incosistents. No es pot escriure a la posició apuntada, només llegir-hi.

- **forward_iterator_tag**: Si escrius i després llegeixes, obtens el valor que has escrit. Les còpies d'un forward iterator funcionen de forma consistent independentment.

- **bidirectional_iterator_tag**: Es pot fer servir on es fa servir un forward iterator i, a més, permet decrements.

- **random_access_iterator_tag**: Es pot fer servir on es fa servir un bidirectional iterator i, a més, es pot fer aritmètica sencera de la mateixa manera que fem amb els punters. Podem fer servir l'operador d'indexació o l'operador de suma i resta amb sencers.

Aquests tags es poden fer servir per determinar en temps de compilació com implementar alguns algorismes. Si sobrecarreguem funcions per un paràmetre que sigui un punter al tipus de tag, podem escollir la implementació a partir de l'aparentment inútil valor de retorn de la funció `iterator_category`. És una altra forma de fer un *static dispatch*.[1]

```
template <class iterator>
iterator advance(iterator origen, int N)
{
    //Crida a la func. Sobrecarregada segons un tercer paràmetre
    advance(origen, N, iterator_category(origen));
}
template <class iterator>
iterator advance(iterator origen, int N, random_iterator_tag * foo)
{
    // Com que és un random iterator podem fer la suma
     return origen + N;
}
template <class iterator>
iterator advance(iterator origen, int N, bidirectional_iterator_tag *
foo)
{
    // Com que no és un random ho hem de fer seqüencialment
    if (N>=0)
        while (N--) origen++;
    else
        while (N++) origen--;
}
```

Llistat 1. Fent servir la categoria de l'iterador per triar la funció en temps de compilació.

[1] Static dispatch és la selecció del mètode de forma estàtica, és a dir, en temps de compilació, en contraposició a la selecció dinàmica, que es basa en la virtualitat.

Altres possibilitats dels iteradors que van més enllà del que proporciona la STL, són les que en deriven de construir iteradors alternatius amb diverses finalitats. Baus i Berek [6] proposen alguns iteradors molt interessants que permeten accedir no exactament als elements d'un contenidor:

- **Iteradors amb funcions**: Són iteradors que no retornen un element del contenidor sinó el valor retornat per una funció, que pot ser pròpia de l'iterador, o explicitada amb un *functor* com a paràmetre template.

- **Iteradors de combinació**: és un iterador que encapsula N iteradors i una funció de combinació. Les operacions es propaguen als iteradors encapsulats i, en demanar un contingut, es retorna el resultat de la funció aplicat als elements apuntats per cadascun dels iteradors.

- **Iteradors de derreferència**: Sovint es construeixen contenidors amb tipus polimòrfics. Aquests contenidors necessiten que l'element bàsic sigui un punter. Molts dels algorismes genèrics operen amb funcions que es poden aplicar al tipus dels objectes iterats, però, quan el contenidor conté punters, el tipus dels objectes iterats és un punter i les funcions a aplicar són pròpies del tipus al que apunta el punter.

Segona part: Realització del projecte

2.1 Metodologia i entorn de treball

2.1.1 Recursos

2.1.1.1 Equip humà

Durant el transcurs d'aquest projecte, l'equip de desenvolupament de CLAM ha estat format per 6 persones. Cadascun dels membres de l'equip treballa en una o dues plataformes i està moderadament especialitzat en certes àrees de CLAM. Aquesta especialització és moderada perquè no és exclusiva. De fet, el disseny i el desenvolupament del sistema en sí és molt creuat entre els diferents desenvolupadors.

Els sis desenvolupadors del projecte són:

- Xavier Amatriain (cap de projecte)
- Enrique Robledo (encapsulat de algorismes de processament i control de fluxe)
- Miquel Ramírez (interícies gràfiques)
- David García (XML)
- Marteen Deboer (entrada sortida d'audio i MIDI)
- Pau Arumí (estructures de dades i control de fluxe)

Entre parèntesis s'indica la seva especialitat tot i que com ja s'ha comentat, aquestes especialitats són molt difuses: hi ha molts àmbits que estan coberts per tothom, i tothom ha intervingut en el disseny i implementació de les àrees dels altres.

Darreramant s'han incorporat tres desenvolupadors més (Albert Mora, Xavier Rubio i Sandra Gilabert) que actualment estan en fase de formació.

També cal mencionar una desena d'usuaris que treballen en altres projectes del grup d'investigació. Aquests usuaris col·laboren incorporant a CLAM els nous algorismes desenvolupats en l'àmbit dels seus projectes i, a més, fan de beta-testers de les successives entregues de CLAM.

2.1.1.2 Servidors

L'equip de desenvolupament fa servir dos hosts *Debian GNU/Linux* com a servidors. Un conté serveis de grup d'investigació en general i fa les següents funcions:

- Servidor de serveis de xarxa (*NIS*)
- Servidor de fitxers remots (*NFS* i *Samba*)
- Servidor de correu
- Groupware (*BSCW*)

L'altre servidor està més orientat a tasques pròpies del desenvolupament:

- Repositori de codi (Servidor *CVS*)
- BugTracking (*Mantis*)
- Accés CVS a només consulta (*ViewCVS*)
- Generador de documentació de referència (*Doxygen*)

Ambdós servidors estan assegurats amb un sistema de còpies de resguard.

2.1.1.3 Estacions de treball

Les estacions de treball són PC's. Per tenir representades en el grup de treball el major nombre possible de plataformes suportades, les estacions de treball funcionen amb diversitat de sistemes MS-Windows i GNU/Linux.

La major part dels sistemes MS-Windows amb els que treballem són *Windows2000* i *XP* tot i que també es manté alguna estació *NT4* i la major part dels sistemes GNU/Linux són distribucions *Debian*, tot i que també cal mantenir alguna estació amb *RedHat* per comprovar que CLAM també hi funciona i per generar paquets d'aquesta distribució.

També es disposa d'un *G4* amb *MacOSX* per fer proves i compilacions en aquesta plataforma. Tot i així, la portabilitat sobre Mac s'ha mantingut sobretot a nivell de compilador, donat que el compilador *CodeWarrior* està disponible també per a Windows i Linux. En l'estació *G4* encara no funcionen correctament els mòduls d'audio i MIDI, la qual cosa és natural perquè és la part més dependent de la plataforma, i encara hem fet poques proves amb aquest sistema.

2.1.1.4 Programari

Els compiladors suportats són *VisualC++* versió 6 a Windows i *GCC 2.95* a Linux. També s'han provat sense problemes, tot i que no es manté el software en aquests compiladors:

- *VisualC++* versió 7 per Windows
- *GCC 3.0* per Linux
- *CodeWarrior* versió 7.0 per Windows
- *Intel C++ Compiler* 6.0 per Windows

Cal dir que els compiladors són el punt més dur del disseny i implementació d'aquesta llibreria, donat que la majoria no implementen correctament l'estàndard.

El cas més alarmant és el *VisualC++*. A la pàgina de la llibreria *Boost* [7] hi ha un recull de bugs del *Visual* que fan que compilar porcions de codi estàndard com les que apareixen en el següent llistat sigui impossible.

```
//wrong explicit function template instantiation

template<class T>
void f()
{
    printf("%d\n", sizeof(T));
}

void use()
{
    f<double>();      // output: "1"
    f<char>();        // output: "1"
}

// Partial specialization of class template does not work.
template<class T>
struct A {};

template<class T>
struct B {};

template<class T>
struct A<B<T> > {}; // comp. Error: template was already
                    // defined as a non-template.
```

Llistat 2Codi estàndard que no compila en MS VisualC++

La llibreria 'estàndard' que ofereix el *Visual* està basada en una versió antiga de la llibreria de HP que té moltes falles. Per compilar sense problemes, els desenvolupadors fem servir la *STLPort*, que és una implementació de la llibreria que està pensada per suplir la pròpia del compilador en aquests casos.

Comparat amb el *VisualC++*, el *GCC 2.95* és molt més complidor amb l'estàndard, però encara no ho és totalment. Les versions 3.0 són més fidels però no les fem servir normalment per desenvolupar perquè ha canviat l'*ABI* i el depurador *GDB* encara no està preparat per aquest canvi.

2.1.2 Gestió de desenvolupament concurrent

Quan diversos desenvolupadors estan actualitzant un mateix codi, sorgeix el problema de gestionar les inconsistències entre les diferents modificacions. En aquest sentit, la comunicació entre els desenvolupadors és clau, però, es pot fer més àgil dotant a l'equip d'algunes eines.

2.1.2.1 Software de control de versions del codi de forma concurrent: *CVS*

L'eina més important és un sistema de gestió de les versions concurrents. Pel desenvolupament de CLAM s'ha fet servir CVS (Concurrent *Versions System*) [8]. El CVS és un sistema de gestió de version molt simple, però, al mateix temps molt versàtil i útil. Avui en dia es fa servir a la majoria de projectes on els desenvolupadors estan distribuïts geogràficament.

La funció principal d'aquest programari és evitar els conflictes entre les modificacions dels diversos programadors. Cada desenvolupador té una còpia del codi en local, on fa les seves modificacions i que pot sincronitzar amb l'estat del repositori comú en qualsevol moment. Quan el desenvolupador està satisfet amb les seves modificacions, les confirma al servidor. Aquest no li deixarà fer-ho si les seves modificacions entren en conflicte amb altres que el desenvolupador encara no ha sincronitzat.

En el següent exemple de codi, es mostra un cas de conflicte en un mateix arxiu de codi (`FDFilterGen.cxx`) resultant d'una situació molt comuna: Un desenvolupador l'ha modificat en local i l'ha confirmat al servidor, aquest l'hi ho ha permes ja que ha detectat que la modificació s'havia fet sobre la versió en local que corresponia a la mateixa versió que era al servidor. Un segon desenvolupador, que no estava al cas de les modificacions del primer, també ha modificat la mateixa porció de codi (amb una lleugera diferència respecte el primer). Ara, a l'intentar confirmar aquests canvis el servidor l'informa que ha estat modificant una versió anterior a l'última actualització i per tant no li permet fer. El que ha de fer aquest desenvolupador és l'operació actualitzar el fitxer desde el servidor. I al fer-ho, com és natural apareixen conflictes.

La forma de manegar-los que té el CVS és molt intuïtiva: deixant a l'arxiu actualitzat unes marques a les línies amb conflictes, explicitant-ne les dues versions: la del servidor i la local. D'aquesta manera el desenvolupador pot decidir amb quina versió quedar-se.

```
          SetGain(0);
<<<<<<< FDFilterGen.cxx
          SetHighCutOff(0);
          SetLowCutOff(0);
=======
          SetHighCutoff(0);
          SetLowCutoff(0);
>>>>>>> 1.17.2.12
          SetPassBandSlope(0);
          SetStopBandSlope(0);
```

Llistat 3 Resultat d'un conflicte entre versions del mateix fitxer detectat pel CVS.

Al repositori central, queda, a més, un històric documentat dels canvis de manera que permet revisar tots els canvis i els seus autors, que s'han fet a cada arxiu, fer comparacions entre versions, o desfer canvis concrets.

Una altra funcionalitat addicional del CVS és la capacitat de crear branques de desenvolupament independents sobre el mateix codi i ofereir maneres de manegar escisions i reunificacions de branques.

Aquest sistema és especialment útil quan el codi s'està utilitzant al mateix temps que es desenvolupa, com és el nostre cas. El que es sol fer és tenir una branca inestable i quan arribem a un punt estable per fer un lliurement (*release*), es sincronitza (operació *merge*) la branca estable amb la primera. Aquesta branca estable és la única que es permet 'veure' als usuaris no desenvolupadors ja que és molt important que les inconsistències temporals pròpies del desenvolupament de la llibreria no efectin als seus usuaris (no desenvolupadors).

També hi ha altres casos de sincronització entre les diferents branques. Entre aquests destaquen els *bugfixes*, en que, normalment a petició dels usuaris de la branca estable, actualitzem el seu codi que haurà estat corregit desde la branca de desenvolupament.

2.1.2.2 Reunions

CVS només soluciona la concurrència a nivell de codi font. També cal que els desenvolupadors es sincronitzin a nivell de disseny. Per això, cal un altre tipus de sincronització que es reforça amb reunions formals i informals.

Les reunions formals es fan amb periodicitat setmanal amb l'objectiu de gestionar els esforços de desenvolupament. Les reunions informals es produeixen sovint quan sorgeix la necessitat de dissenyar quelcom de forma conjunta, aclarir algun punt o consensuar alguna decisió de disseny presa unilateralment abans de tirar-la endavant.

2.1.2.3 Discussió online

El problema de les reunions informals era que:

- Interrompien el desenvolupament que cadascú portava a terme.
- Sorgien massa freqüentment.
- No quedava constància escrita del què s'hi discutia .

Per aquests motius, tot i estar tots els desenvolupadors treballant a la mateixa sala, es va considerar raonable establir mecanismes per la discussió en línia mitjançant els fòrums del software de groupware (*BSCW* [9]) que es fa servir a l'MTG i diverses llistes de correu temàtiques que es sincronitzaven amb el forum.

Aquestes discussions en línia varen suavitzar la freqüència de les interrupcions i en deixaven constància tant per a la consulta posterior com per als que no n'havien pres part.

2.1.3 Gestió de tasques pendents i correcció d'errors (*bug tracking*)

A mesura que la llibreria ha esdevingut més complexa, i, ha crescut el nombre d'usuaris, ha calgut adoptar un sistema de gestió d'informes d'error i de propostes de funcionalitats.

Hi ha disponibles diverses eines de gestió d'informes d'error. Per a CLAM s'ha escollit l'eina *Mantis* [10], donat que la gestió del cicle de vida dels informes d'error és molt clara. Es pot establir una discussió sobre aquests bugs, i té una interfície pensada per avaluar el cost del canvi i l'assignació de recursos, tot plegat informant via correu electrònic als desenvolupadors interessats.

A més, *Mantis* és un programari lliure, net, madur i molt fàcil d'instal.lar i administrar, al contrari que d'altres productes no comercials semblants que vam trobar, com ara *Bugzilla* i *phpBT*.

L'altre programari, *phpBT*, és molt semblant al *Mantis*, però, la gestió de correu s'adapta menys a les necessitats que tenim al grup de desenvolupament. A més, ens semblava molt més elegant i flexible la seva interfície i més dinàmic i coherent el seu ritme d'entregues de noves versions. Tot i ser un projecte més jove tenia molt més bona base.

En Mantis els informes d'errors o peticions de funcionalitat, adopten els següents estats al llarg de la seva vida:

- **New**: L'informe ha entrat al sistema.
- **Feedback**: El desenvolupador és conscient d'un error però demana més informació als usuaris.
- **Acknowledged**: Algun desenvolupador l'ha vist però encara no ha pres cap acció concreta.
- **Confirmed**: Un desenvolupador l'ha reproduit i confirma l'error informat.

- **Assigned**: Un desenvolupador ha assumit la tasca de corregir-lo. O ha delegat aquesta tasca a un altre desenvolupador.
- **Resolved**: Les accions correctives corresponents a l'error ja s'han pres.
- **Closed**: L'informador, o algú diferent a qui ha fet la correcció, ha confirmat la desaparició de l'error.

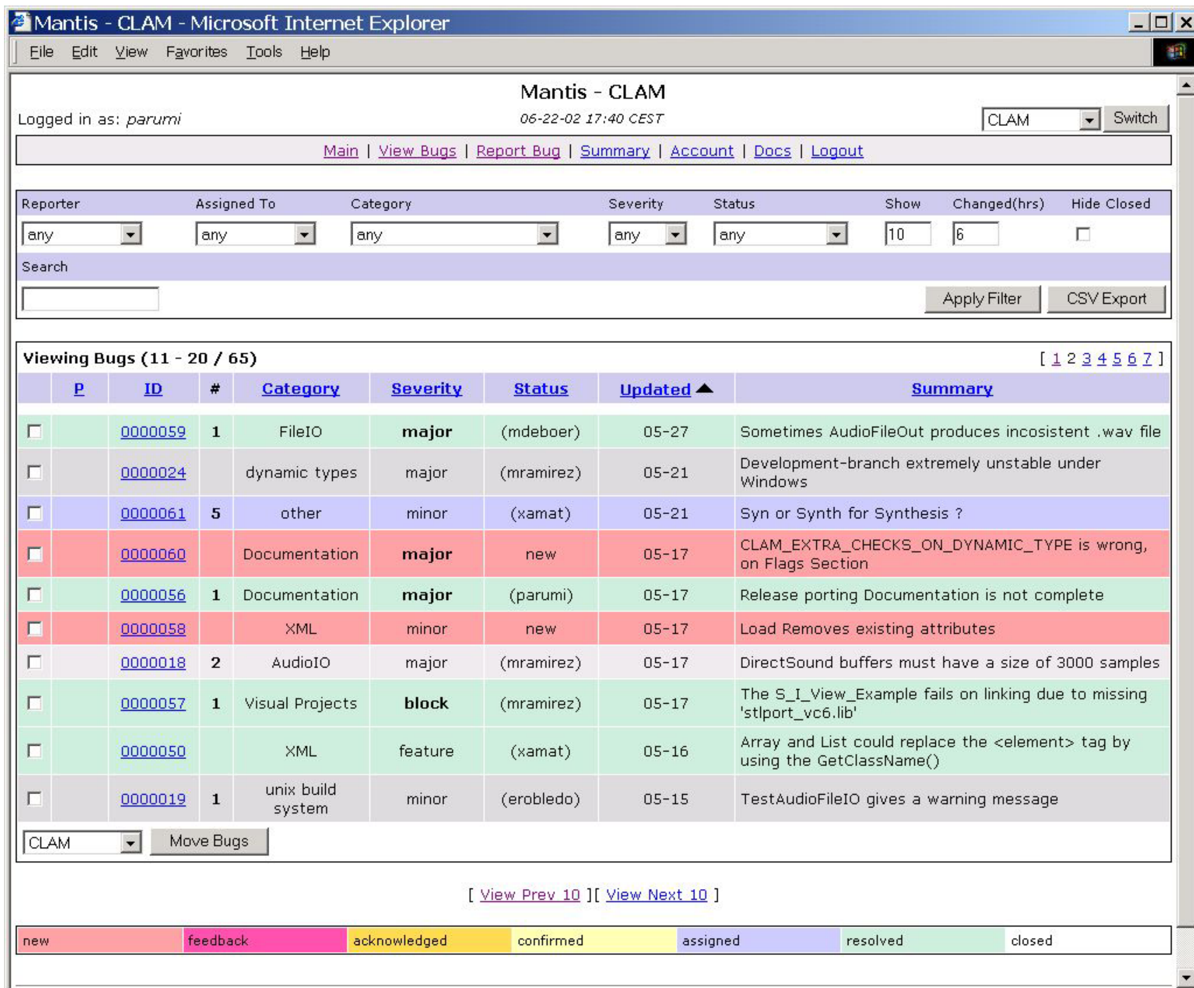

Figura 6: Captura de pantalla de l'aplicació de gestió d'errors Mantis, a través d'un navegador.

2.1.2 Documentació

La documentació és quelcom important a qualsevol projecte software. En el cas d'una llibreria la documentació pren una importància crucial, donat que s'hi juga, no només el seu manteniment sinó la seva usabilitat.

Manuals i tutorials

La documentació de CLAM ha d'anar adreçada a tres actors principals amb diferents nivells de profunditat.

- **Mantenidors**: Són aquells desenvolupadors que mantenen el nucli de CLAM. La documentació destinada només a aquests actors, comprèn descripcions detallades del disseny i la implementació i les justificacions de les decisions de disseny preses. També inclou altra documentació com els anàlisis de requeriments, els esborranys de disseny previs i altra documentació produïda durant el desenvolupament.

- **Desenvolupadors**: Anomenem desenvolupadors, tot i que de fet, els tres actors ho són, a aquells que, sense entrar en les interioritats del disseny de CLAM, implementen nous objectes, de processament o de dades. Aquest nivell de documentació, requereix conèixer els requeriments que aquests objectes tenen.

- **Usuaris**: Són aquells usuaris (de fet són programadors) que faran servir CLAM per construir el seu sistema amb objectes de processament i de dades existents. La documentació per aquest tipus d'usuari es limita a descriure la interfície més externa dels objectes de CLAM.

S'ha compilat un manual d'usuari que cobreix els dos darrers nivells, el qual s'inclou a aquesta memòria com a apèndix (veure Apèndix A).

La documentació adreçada als mantenidors es manté dins del software de groupware per a la seva discussió continuada.

Doxygen

A banda dels manuals d'usuari, també hi ha una guia de referència generada automàticament amb *Doxygen*. *Doxygen* és una eina de documentació de codi semblant a *JavaDoc*. [11].

La documentació per a *Doxygen* es manté dins del codi font amb la qual cosa és més fàcil que el mateix programador la mantingui al dia. A partir d'aquesta documentació dins el codi i de l'anàlisi del propi codi, *Doxygen* genera documentació en diversos formats interactius: *HTML*, *PDF*, *texinfo*, *man*... S'ha inclòs aquesta documentació dins el CD que acompanya la memòria.

Aquesta informació és molt útil per els usuaris de la llibreria donat que els hi dóna una visió molt clara de les interfícies que han de fer servir i els permet navegar gràcies al seu complert sistema de referències.

Doxygen, però, no només té una funció de documentació sinó que és molt útil com a eina d'anàlisi per al desenvolupament de CLAM. Només a partir de l'anàlisi del codi, i, sense cap documentació explícita per part del programador, *Doxygen* ja proporciona, entre altres utilitats:

- Inventari de tots els símbols definits.
- Diagrames d'herències
- Diagrames d'inclusió de capçaleres (fitxers .hxx).
- Informació sobre qui fa servir cadascuna de les funcions.
- Informació sobre les funcions que fa servir cada funció.
- Informació sobre qui implementa una determinada funció virtual

CLAM::AudioDevice class Reference - Microsoft Internet Explorer

File Edit View Favorites Tools Help

CLAM::AudioDevice Class Reference

This is the abstract base class for an audio device. More...

```
#include <AudioIO.hxx>
```

Inheritance diagram for CLAM::AudioDevice:

CLAM::AudioDevice

CLAM::ALSAAudioDevice CLAM::DirectXAudioDevice

[legend]

Collaboration diagram for CLAM::AudioDevice:

int vector< AudioIn * > string vector< AudioOut * >

mNChannels mInputs mName mOutputs

CLAM::AudioDevice

[legend]

List of all members.

Public Methods

AudioDevice (const std::string &name)
virtual ~**AudioDevice** ()
Destructor of class. More...

virtual void **Start** (void)=0 throw (Err)
This method must be called to begin the use of this Device. More...

virtual void **Stop** (void)=0 throw (Err)
This method must be called to end the use of this Device. More...

virtual void **Read** (**Audio** &audio, const int channelID)=0
*Reads the information given by this Device and passes data to an **Audio** chunk.* More...

virtual void **Write** (const **Audio** &audio, const int channelID)=0
*Writes the information given by an **Audio** chunk in the Devices.* More...

virtual void **GetInfo** (**TInfo** &)
Getter for the Info of Device Object attached to this AudioDevices instantiation. More...

Figura 7Captura de pantalla de documentació HTML generada pel Doxygen a partir del codi font

Aquesta documentació de classe ha estat generada a partir del següent codi. S'observa que els comentaris estan en format *JavaDoc* (`/** ... */`) i que contenen alguns tags (`@param`, `@see ...`)

```
/** This is the abstract base class for an audio device. With an audio device
  * we refer to any kind of (multichannel) audio input/output/fullduplex.
  * AudioDevices are usually created by the AudioManager. The interface
  * for the user however, are the AudioIn and AudioOut classes.
  * <p>
  * @see AudioIn, AudioOut, AudioDeviceList, AudioManager
  */
class AudioDevice
{
     friend class AudioIn;
     friend class AudioOut;
     friend class AudioManager;
public:

     std::vector<AudioIn*> mInputs;
     std::vector<AudioOut*> mOutputs;

     std::string mName;
     int mNChannels;

     /** Constructor of the class that sets the name of object to the string passed
      *  by parameter. @name String with the name of object
      */
     AudioDevice(const std::string& name)
     {
            mName = name;
            mNChannels = 0;
     }

     /** Destructor of class*/
     virtual ~AudioDevice() { };

     /** This method must be called to begin the use of this Device.
         Must be implemented by any Device derived from this class*/
     virtual void Start(void) throw(Err) = 0;
     /** This method must be called to end the use of this Device.
         Must be implemented by any Device derived from this class*/
     virtual void Stop(void) throw(Err) = 0;
     /** Reads the information given by this Device and passes data to an Audio chunk.
       * Must be implemented by any Device derived from this class.
      *  @param audio Audio object where data will be stored.
      *  @param channelID Channel to read.
      */
     virtual void Read(Audio& audio,const int channelID) = 0;
     /** Writes the information given by an Audio chunk in the Devices.
      *  Must be implemented by any Device derived from this class.
      *  @param audio Audio object with values that must to be passed to Devices
      *  @param channelID Channel to write
      */
     virtual void Write(const Audio& audio,const int channelID) = 0;

     /** Getter for the Info of Device Object attached to this AudioDevices
       *   instantiation
      *  @param info TInfo object that method will modify with the values of Tinfo
       *  internal object
      */
     virtual void GetInfo(TInfo&);
            ...
```

Llistat 4 Exemple de codi amb comentaris JavaDoc, per ser interpretats pel Doxygen

Rational Purify

Malgrat els tests, de classes i d'integració, a què anavem sotmetent les successives implementacions, i a l'estil de codificació en que ens hem imposat, sovint ha estat inevitable llargues hores dedicades a buscar bugs. Ens vam adonar que els bugs més difícils de trobar tenien l'orígen en un mal managament de la memòria dinàmica, així que varem buscar alguna eina per detectar corrupcions de memòria o almeny els *memory leaks*[1].

Les primeres eines utilitzades van ser *MemProof* per Linux (utilitzada en el desenvopament de *Gnome*) i funcions de llibreria proporcionades pel *VisualC++*, en Windows. Tot i que la primera era clarament superior a la segona, aquestes només abarcaven la detecció de *memory leaks*.

Posteriorment ens vam utilitzar el *Rational Purify*, tot i que només el tenim disponible per Windows, és una eina molt més completa, madura i que s'integra perfectament amb el compilador. Fa una comprovació exhaustiva del *heap* del programa: detecta *memory leaks*, accessos a memòria no actualitzada, a memòria no gestionada per l'aplicació, i ús incorrecte d'allocadors i alliberadors de memòria que poden actuar a diferents *heaps* (per exemple, es pot donar el cas que es faci un *new* al *heap* de l'aplicació i el *delete* del mateix objecte es faci al *heap* d'una *DLL*).

És una eina, que convinada amb la pràctica dels tests, ens ha estalviat un nombre molt important d'hores dedicades a implementació i manteniment.

[1] Els *memory leaks* són recursos de memòria ocupats durant l'execució del programa que no han estat alliberats.

2.1.3 Cicle de desenvolupament

2.1.3.1 Cicle de vida

Tant CLAM com el projecte final de carrera s'han desenvolupat amb un cicle de vida en espiral. Amb un objectiu final en ment, a cada etapa s'estableixen els objectius parcials i es fa un cicle de vida semblant al clàssic per assolir aquests objectius.

La metodologia de desenvolupament s'ha vist molt influenciada per les eines i per la metodologia estesa als desenvolupament *opensource*, la qual es caracteritza per un desenvolupament en espiral orientat a entregues: A cada fase, cada desenvolupador planifica els desenvolupaments que entraran en la següent entrega.

El sistema de gestió de canvis del codi font (CVS veure apartat §2.1.2.1) ens permet que cada desenvolupador, si ha de fer modificacions que puguin alterar el normal desenvolupament dels altres, crei una nova branca de desenvolupament i així integrar els canvis en punts concrets. S'estableixen així petits subprojectes de curta durada amb un o dos desenvolupadors involucrats.

El desenvolupament del projecte final de carrera ha constat de fases que cal veure com petits subprojectes. La planificació havia de ser subprojecte a subprojecte, donat que l'evolució de CLAM també ha estat pas a pas, i, a més, fins que el projecte no ha anat avançant, no es coneixia la viabilitat de cadascuna de les fases. Les fases de desenvolupament han estat:

- Sistema de composició dels elements bàsics dels sistema. (usant patró *Composite*)
- Sistema bàsic de passivació; guardar estructures d'objectes en format XML.
- Primera iteració dels DynamicTypes.
- Gerarquia bàsica d'objectes de processament, dades del processament i objectes de configuració.

- Controls pels objectes de processament.
- Segona iteració dels DynamicTypes.
- Tercera iteració dels DynamicTypes.
- Nodes i control de fluxe.

2.1..3.2 Prototipatge i integració

A CLAM s'han fet servir moltes tècniques de programació poc comunes que, tot i estar contemplades per l'estàndard del llenguatge, sovint superen el límit del que la implementació del compilador concreta és capaç de fer. Un límit que, sobretot en el cas del *VisualC++* està molt per sota del que s'espera d'una implementació comercial tan estesa i amb tants recursos al darrera.

Per fer moltes parts d'aquest projecte, sovint s'ha fet servir una aproximació incremental mitjançant prototipus. Els prototipus anaven afegint característiques que s'anaven provant en les principals plataformes: *GCC*, *VisualC++* i *CodeWarrior*.

Aquestes proves agilitaven la incorporació de noves característiques en no haver de modificar i compilar tota la llibreria i en localitzar en un codi petit problemes que un compilador concret podia donar. Tot i la frustació de veure com moltes bones idees no superaven la prova de la portabilitat, vam aconseguir trobar solucions alternatives per a la majoria d'elles. És trist entendre per portabilitat, haver de considerar bugs i divergències gratuïtes de l'estàndard.

Aquesta tasca d'investigació hagués estat impossible de dur-se a teme integrant aquestes proves amb tot el codi de CLAM. Així quan integravem la solució dins de CLAM estàvem gairebé segurs que funcionaria. Amb prototipus petits es controlen més les interaccions i a més, es compila més ràpid.

2.1.3.3 Proves i detecció d'errors

Cada classe o cada unitat funcional té el seu propi programa d'autoproves. Sense pretendre fer totes les proves exhaustives, la bateria de proves prové del recull de les successives proves que s'han anat fent sobre cada classe o conjunt de classes relacionades.

S'han fet proves unitàries exhaustives de *caixa blanca* i *negra* a les classes més centrals del sistema. En altres classes menys cèntriques s'ha confiat en les proves de funcionalitat inicials que sovint eren també una prova d'integració. En les classes no centrals, només s'ha passat a fer proves unitàries en el cas que haguem necessitat localitzar un vici concret.

S'ha procedit reactivament a les proves de les classes no cèntriques perquè els requeriments de temps que teníem no permetien provar totes les classes de forma exhaustiva.

S'ha seguit la política de, un cop fetes, conservar i mantenir aquestes proves com a una part més del sistema per aprofitar aquest temps invertit en desenvolupar la prova.

També s'ha procurat que la majoria de les proves, sigui quin sigui el seu origen, puguin interactuar amb un sistema automàtic d'auto-proves que no requereixi la intervenció humana per verificar si han estat superades o no. D'aquesta manera podem passar les proves més sovint i detectar més ràpidament quan una modificació ha introduït inestabilitat.

Altres tres eines, han facilitat molt la tasca de detecció d'irregularitats:

- Les **comprovacions d'invariant**, que són uns mètodes escrits pel programador que l´únic que fan és comprovar que els objectes complexes estiguin en estat intern consistent. Així es poden detectar condicions d'error latents que trigarien en manifestar-se o no ho farien pas. Típicament la comprovació d'invariant es fa als programes tests o en codi de llibreria, dins d'assercions en *debug.*

- Les **assercions** que verifiquen algunes precondicions i postcondicions dels mètodes concrets. El seu funcionament s'explica en un apartat més endavant. Juntament amb les assercions hi ha els avisos que podem habilitar per detectar situacions que, tot i no ser incorrectes, sí que són sospitoses.

- La **inspecció** que són mètodes que permeten representar les interioritats d'un objecte mentre el programa s'executa. En aquesta funcionalitat, ha pres part el suport XML un cop ha començat a estar disponible i també disposa de visualització gràfica de les dades cridada desde el debugger.

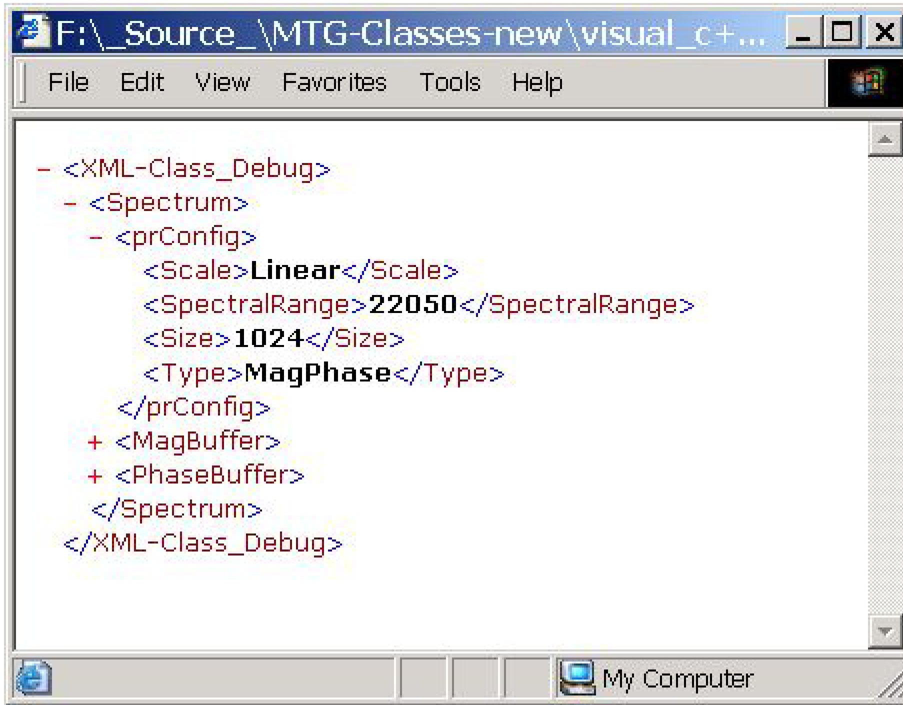

Figura 8. Inspecció d'un objecte (dynamic type) llançada desde el debugger, en format XML.

2.2 Disseny i implementació

En aquesta secció s'expliquen les decisions de disseny més importants dels elements estructurals que formen el nucli de la llibreria. Començaré amb els *dynamic types*, que es poden considerar una extensió del C++ per treballar amb objectes amb atributs no instanciats, i per tant, es tracta d'una eina prou baix nivell com per utilitzar-se en molts altres contextes. A continuació, descriuré els blocs bàsics a partir dels quals es pot construir un processament: les classes *Processing*, les dades que aquests manipulen, i les que els controlen.

Per completar el *framework* necessitem unes altres facilitats que no estan tant directament relacionades amb el processament de les dades: la capacitat de serialització dels objectes del model que permet activar i passivar les dades en el format que desitgem; i el mòdul GUI de visualització gràfica i altres facilitats importants per fer aplicacions d'audio: el MIDI i el multifil (*multithread*). Tot i que he col·laborat en el disseny d'aquestes facilitats; com he dit a l'apartat d'objectius, aquesta part queda fora de l'abast del meu projecte.

2.2.1 *Dynamic types*

2.2.1.1 Concepte

A CLAM ens trobem amb algunes classes del nucli de la llibreria que tenen un gran nombre d'atributs, per exemple classes de dades que descriuen segments d'audio, i en el context d'un processament concret, d'aquests atributs només en necessitem un subconjunt determinat; o per exemple objectes de configuració en els quals s'ofereixen paràmetres per defecte o paràmetres redundants entre ells, la qual cosa representa un malbaratament de memòria, sobretot quan tenim grans col.leccions d'aquests objectes.

Donar solució a aquest problema és l'objectiu principal del desenvolupament de l'estructura de dades (o més aviat, extensió del llenguatge) *dynamic type*,

En canvi els *dynamic types* són una solució que permet optimitzar la memòria utilitzada. Una classe *dynamic type* (que és subclasse de *DynamicType*) defineix un conjunt d'atributs que podran ser instanciats i desinstanciats en temps d'execució, alliberant la seva memòria. Estan definits de tal forma que, a part del protocol d'instanciació, semblen a l'usuari de la classe, atributs privats accedits per mètodes accessors (*GetXXX(), SetXXX(...)*), conservant totes les restriccions del tipatge.

2.2.1.2 Alternatives de disseny estudiades.

Abans de decidir-nos per la solució final, que es basa en l'ús intensiu de *macros*, i per tant és poc d'estil C++, (és que es pot veure com una extenció del llenguatge) varem buscar altres alternatives que no tinguéssin aquest preu en l'estil de programar la llibreria:

La primera solució que ens vam plantejar per el problema de no malbaratar memòria pels atributs no usats va ser la d'utilitzar herència, tenint tantes subclasses com subconjunts d'atributs volguéssim. Però aquesta aproximació no era prou flexible, ja que aquests subconjunts podien tenir interseccions i, a més, es volia tenir total flexibilitat alhora d'instanciar i desinstanciar qualsevol atribut en temps d'execució.

El C++ permet treballar amb objectes dinàmics, per tant la segona aproximació va ser plantejar-nos usar classes fent un ús intensiu de variables de memòria dinàmica. És a dir, declarant els atributs com a punters del tipus adecuat i portant una gestió de la seva memòria.

Aquesta opció també va ser descartada ja que: Per una banda no optimitzava la memòria: l'espai ocupat pels punters era inevitable i en classes que tenien molts atributs de tipus bàsic (de mida inferior a la mida d'un punter), la solució amb atributs normals hauria estat millor que usar punters, fins i tot quan aquests no tinguessin memòria instanciada.

Per altra banda, no solucionava el problema de garantir un accés uniforme (en quant a l'estil dels mètodes accessors) i a més no permetia una gestió de la memòria dels atributs de forma automàtica.

L'alternativa més interessant a l'ús de *macros* estava basada en els *templates* de C++, i s'inspirava en algunes solucions usades a la llibreria *Boost*. [12] La idea principal és la de definir atributs de tipus `dyn_attribute<tipus_base>` templatitzats pel tipus base que volguem, es tracta d'una classe template que actua de *wrapper* d'un tipus o classe qualsevol. Aquest *wrapper* ofereix mètodes per instanciar/desinstanciar l'objecte adaptat i també per accedir-hi. Per exemple, podriem fer:

```
dyn_attribute<int> nSines;

nSines.Add();
nSines.Set(5);
int i = nSines.Get();
```

L'estudi d'aquesta solució va demostrar que era viable una gestió eficient de la memòria i a més oferia les capacitats d'introspecció que ens oferien les macros, i que per tant era adient per implementar la serialització automàtica.

Desgraciadament vam haver de desestimar aquesta opció pel fet que requeria un ús intensiu de macros i alguna tècnica en particular (instanciacions parcials, i altres) que, tot i estar contemplada a l'estàndard, la majoria de compiladors no suporten (en especial *VisualC++*).

2.2.1.3 La solució escollida

Els *dynamic types* implementen, sense intervenció de l'usuari, la interfície de *Component*[1]: això és possible perquè al declarar els atributs dinàmics, permet al *DynamicType* conèixer la

[1] La nostra classe Component te la mateixa semàntica que se li dóna al patró Composite. Veure patrons de disseny §1.4.2

seva estructura de composició, i per tant pot implementar: produir còpies de la composició en profunditat i someres (*deep* i *shallow copies*) i assignacions de composicions.

Ens permeten obtenir certa introspecció del tipus, que ens permet donar les solucions a la serialització que ofereixen per defecte llenguatges com SmallTalk i Java. Concretament hem optat per començar oferint la serialització en XML. Només per haver declarat una classe com a *dynamic type*, n'obtenim la capacititat de guardar-ne les seves dades (de l'objecte i de tota la composició recursiva, en cas que la tingui) i de restaurar-les, sense que l'usuari hagi d'escriure cap línia de codi.

Cal aclarar que *dynamic type* és el nom que reben a CLAM i no té res a veure amb el què alguns autors [13] anomenen tipus dinàmics: definicions de tipus que es poden crear i manipular en temps d'execució. De fet, en el nostre cas, el que és dinàmic són els atributs i no el tipus en sí.

Els *dynamic types* són una part molt central de CLAM perquè s'utilitzen per representar tots els objectes de dades de processament i els de configuració.

Com a conseqüència de ser utilitzats en parts molt sensibles a l'eficiència d'execució, com són les dades processades, els *dynamic types* han patit diversos processos d'optimització, i el seu ús no es considera cap perduda d'eficiència, fins i tot en alguns casos facilita algunes optimitzacions addicionals[1].

Com ja he dit, la complexitat de la implementació dels *dynamic types* queda amagada a l'usuari d'aquestes classes en el sentit que no se li requereix que implementi cap mètode, sinó que simplement declari els atributs dinàmics d'una forma similar a les classes C++ normal. Per tant, la pregunta és: on hem ocultat aquesta complexitat? I la resposta: a dos classes; la classe base

---

[1] Per exemple, al treballar amb objectes dinàmics, la localitat de les dades dinàmiques provoca menys fallades de pàgina a *cache,* que no pas atributs dinàmics 'dispersos'. Per altra banda, si tenim una col·lecció de *dynamic types* amb la mateixa estructura dinàmica podem obtenir un iterador que recorre un mateix atribut dinàmic de tots els elements de la col.lecció, i aquest accés pot ser molt optimitzat, ja que podem crear col·leccions de *dynamic types*, en memòria contígua.

DynamicType de la qual cal derivar, i a la classe concreta a través de macros. Per qüestions de mantenibilitat, el criteri seguit ha estat el d'alleugerir al màxim possible la complexitat de les macros i traslladar-la sempre que era possible a la classe base. Més endavant mostraré exemples d'aquestes macros.

2.2.1.4 Estructura interna

Els atributs dinàmics no es corresponen amb atributs normals de C++ ni tan sols amb variables dinàmiques. Estan situats a un bloc de memòria que gestiona el *dynamic type*. Els objectes que representen els atributs dinàmics es creen i destrueixen mitjançant els operadors de construcció i destrucció (*new* i *delete inplace*) sobre el bloc de memòria disponible, que l'anomenem taula de dades.

El *dynamic type* manté una taula d'instanciacions on marca, per cada atribut dinàmic, si aquest està instanciat o no. De fet, el que es guarda és l'*offset* de la memòria de l'atribut a la taula de dades, o bé un -1 si aquest no està instanciat. Al fer una instanciació o desinstanciació d'atribut, cal demanar explícitament a l'objecte que actualitzi les seves dades per reajustar l'espai disponible a la taula de dades. Així optimitzem les reallocacions de la taula de dades.

En principi, la taula d'instanciacions és pròpia de cada objecte però, s'ha optimitzat de forma que es comparteixi entre diversos objectes que tenen la mateixa forma dinàmica o prototipus. La taula d'instanciacions es comparteix amb el criteri de que quan un objecte intenta modificar-la, llavors ha de construir la seva pròpia taula d'instanciacions.

A més de la taula d'instanciacions, cada tipus de *dynamic type* té una taula compartida per totes les instàncies d'aquest tipus concret on hi ha informació general sobre els atributs disponibles (tipus, nom, característiques, funció constructor, destructor...)

En resum:

- Hi ha una taula compartida entre totes les instàncies amb informació comuna de descripció dels atributs dinàmics: la **taula de descripció de tipus.**

- Hi ha una taula compartida entre les instàncies provinents d'un mateix prototipus que conté informació sobre l'estat d'instanciació de cada atribut: la **taula d'instanciació**.

- A cada objecte hi ha una taula de dades que conté de manera contígua el contingut dels atributs instanciats per cada objecte: la **taula de dades**.

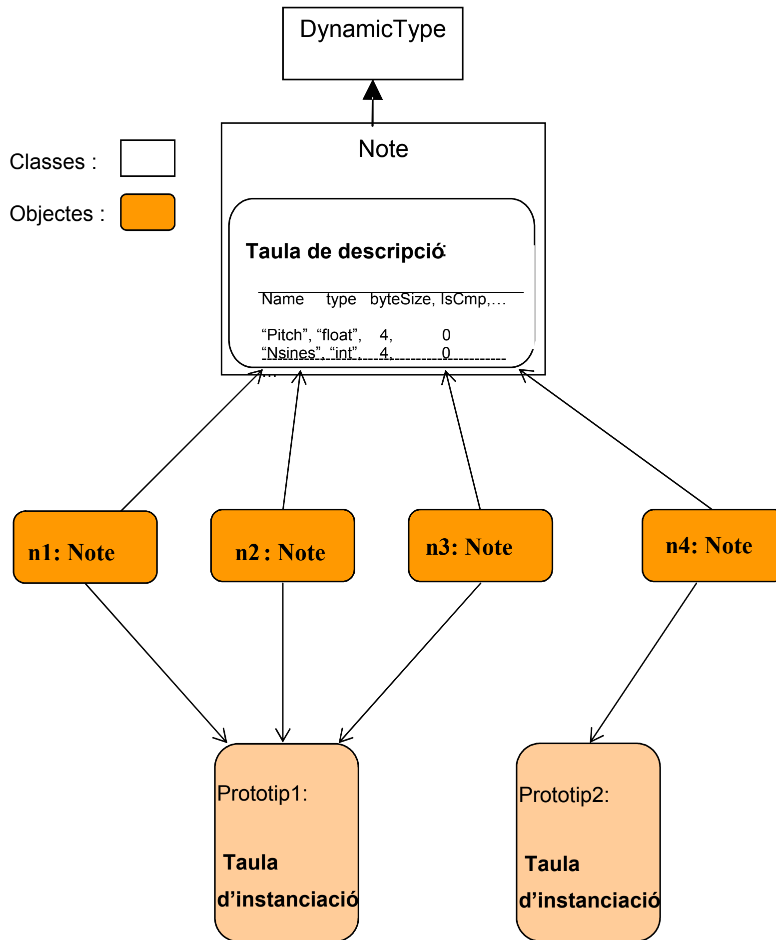

Figura 9. Estructura interna d'uns objectes dynamic type

2.2.1.5 Utilització: un exemple

```
class Note: public DynamicType
{
public:
    DYNAMIC_TYPE      (4, Note)
```

```
    DYN_ATTRIBUTE    (0, public, float, Pitch)
    DYN_ATTRIBUTE    (1, public, unsigned, Nsines)
    DYN_CONTAINER_ATTRIBUTE (2, public, list<Sine>, Sines, harmonic)
    DYN_ATTRIBUTE    (3, private, Audio, Wave)
};
```

Com he dit anteriorment és una declaració basada en macros. S'observa que hi ha tres tipus diferents de macros. La primera `DYNAMIC_TYPE` és l'anomenada macro de classe i expendeix el constructor de la classe i algun altre mètode.

Les altres dues macros són per declarar atributs dinàmics. `DYN_ATTRIBUTE` rep com a paràmetre l'índex de l'atribut, l'accessibilitat (public, private, protected), el tipus i el nom. La segona `DYN_CONTAINER_ATTRIBUTE` permet al *dynamic type* treure profit dels contenidors que compleixen la interfície de la STL (entre ells, és clar, tots els de la STL).

El benefici principal d'aquesta macro és que permet al *dynamic type* fer introspecció per tots els seus sub-elements. S'observa que aquesta macro té un atribut addicional, aquest serveix per etiquetar cadascun dels elements del contenidor quan fem la passivació del *dynamic type* en XML.

Un cop hem declarat la classe, la podem utilitzar d'aquesta manera:

```
Note myNote;  // creem una instància

myNote.AddAll();  // marquem per instanciar tots 4 atributs
myNote.RemoveWave();  // des-marquem per el 4rt atrib.

cout << myNote.HasPitch();  // per pantalla surt: "false"
myNote.UpdateData();  // realitzem les 3 instanciacions.
cout << myNote.HasPitch();  // ara surt: "true"

myNote.SetNSines(10);
myNote.SetPitch(440);
int i=myNote.GetPitch();  // compiler error: GetPitch() returns float
int j=myNote.GetNSines(); // ok

Note anotherNote(myNote);// creem un altre objecte amb la mateixa
                         // matriu d'atributs dinàmics.
```

I podem treure profit de les capacitats de serialització d'una manera molt senzilla:

```
XMLStorage storage("document");
```

```
// Serialitzem myNote en xml a tavés de l'XMLStorage, i volca la
// sortida en un fitxer.
storage.Dump(myNote, "file_name.xml");

// restaurem el contingut de tota l'estructura en un nou objecte:
Note anotherNote;  // inicialment sense cap atr. dinàmic instanciat
Storage.Restore(anotherNote, "file_name.xml");
// Ara myNote=anotherNote, incloent tots els seus fills.
```

2.2.1.6 Primera iteració: Només tipus bàsics i punters

La principal característica de la primera implementació dels *dynamic types*, i també la seva limitació principal, era que només suportava atributs dinàmics que, o bé fossin tipus bàsics de C o bé punters (de qualsevol tipus).

Aquesta mancança es devia a que no es feia ús dels constructors dels tipus, i a la hora d'instanciar un atribut, l'únic que es feia era reservar un espai no informat de la taula de dades. Aquest espai o *slot*, això sí, era posteriorment convertit al tipus especificat a la macro mitjançant un *static cast* dins el codi de l'accessor corresponent.

Però tot i que es fes un *static cast* al tipus adequat, l'objecte retornat no tenia perquè ser vàlid. En particular, qualsevol classe C++ que tingui definides funcions virtuals, guarda dins la memòria dels seus objectes un punter a la *taula de funcions virtuals*. I és evident que aquest tipus de dades internes només es poden crear usant el constructor que ofereix la pròpia classe.

Un altre síntoma del mateix problema era la impossibilitat d'actualitzar el valor d'un atribut dinàmic de tipus no simple, ja que per això cadria usar un operador d'assignació (el qual no es pot garantir que existeixi) o bé un constructor de còpia, que si bé sempre està definit, no teniem la forma d'utilitzar-lo en els atributs dinàmics.

2.2.1.5 Segona iteració: Tot tipus d'objectes i passivació en XML

La segona iteració en el disseny i implementació dels *dynamic types*, va ser motivada per l'anterior problema, i la solució la va aportar l'operador *new inplace* de la llibreria estàndard de

C++. Aquest operador es comporta com un *new* normal i corrent (executa el constructor per defecte de la classe) però no reserva espai de memòria, sinó que utilitza un espai especificat en l'argument de la funció.

L'implementació d'aquesta solució va comportar que a la taula estàtica descriptora del tipus, es guardessin punters a tres funcions: la constructora, la destructora i la constructora per còpia. Això va permetre que els atributs dinàmics fóssin de qualsevol tipus i no només tipus bàsics..

Implementació de la passivació en XML

La primera implementació de la serialització als *dynamic types* només considerava la passivació XML i encara no feia activació. Feia servir la informació en temps d'execució que hi ha disponible a la taula compartida de descripció dels atributs.

A la taula de descripció d'atributs hi havia la següent informació útil de cara a implementar el mètode `StoreOn`: Una cadena amb el nom de l'atribut, una altra cadena amb el nom del tipus i uns quants valors booleans que indicaven, entre altres coses si l'atribut era `Component` o `DynamicType`.

En aquesta primera implementació, la taula de descripció s'omplia a mida que els atributs s'anaven instanciant. Cada cop que s'instanciava un atribut, primer es mirava si aquest havia estat informat a la taula de descripció, i si no, llavors s'informava.

Això era degut a que la taual de descripció havia d'omplir-se des dels mètodes d'instanciació dels atributs concrets (`AddXXX`) perquè la informació pròpia de cada atribut només era accessible a les macros que defineixen cada atribut.

Per fer la detecció de si el tipus és un `Component` i `DynamicType` es va fer servir una tècnica de *static dispatch*, sobrecarregant un paràmetre d'una funció per a un punter a `Component` i punter a `void`. S'enviava en el lloc del paràmetre, un valor nul convertit en punter al tipus de l'atribut.

Per fer la passivació es recorria la taula de descripció del primer a l'últim atribut. Si l'atribut era un `Component`, es feia un *cast* a `Component` de la posició de memòria corresponent, i es

feia servir un `XMLComponentAdapter` que implementava el mètode `Store` adecuat. Si no era `Component`, llavors es comparava la cadena del tipus amb les cadenes d'alguns tipus bàsics comuns i si coincidia es feia un *cast* i s'utilitzava un `XMLAdapter` templatitzat pel tipus concret.

S'observa que la forma de detectar els tipus bàsics és molt feble. No és tolerant a noms de tipus sinònims (*typedefs*) i força a canviar la implementació de l'`StoreOn` cada cop que afegim un nou tipus bàsic.

Quan es va plantejar el fet d'implementar l'activació XML, es va veure que aquesta implementació era molt limitant i es varen introduir tàctiques inspirades en la metaprogramació i en la programació genèrica per reconstruir els tipus dinàmics.

2.2.1.6 Tercera iteració: Optimitzacions, mètodes encadenats i activació XML.

La tercera i última iteració va incloure una forta optimització de la manera com es gestionava la memòria. Ens varem adonar que sovint, les modificacions de la forma dinàmica de l'objecte, requerien massa cost computacional ja que implicaven fer una reallocació de tota la taula de dades, executant el constructor de còpia de cada atribut, i aquests podien ser grans agregacions o estructures geràrquiques.

Es tracta d'un cas on les dues forces: optimitzar la memòria i optimitzar el temps de computació entren en conflicte. La solució que s'ha pres, és que per defecte s'optimitzi la memòria, però si es vol, es pot optar per optimitzar el temps fent-ho de forma explicita (cridant un mètode `PreAllocateAllAttributes)`

Mètodes encadenats

Varem observar que no era necessari deduir el tipus a partir del nom ni guardar enlloc el nom de l'atribut, per fer un mètode que fes la passivació o activació d'un atribut concret, si aquest mètode s'expandia amb la macro d'aquest atribut. Tenim el tipus que podem fer servir

directament al codi, i tenim el nom de l'atribut que podem fer servir per cridar les funcions que ens donen informació sobre ell, sense accedir a cap taula de descripció.

El problema més imporant era com fer que un mètode expandit per la macro comuna, pogués cridar a cadascun dels mètodes expandits per les macros d'atribut. La forma que fèiem servir per què les macros d'atribut no col.lisionessin, era fer que el nom de l'atribut normés part dels noms dels mètodes: `AddMyInt`, `RemoveMyInt`, `GetMyInt`, `SetMyInt...` Així doncs, desde la macro comuna no sabíem quin és el nom del mètode corresponent de cada atribut.

En C++ una altra forma de fer que dos mètodes no col·lisionin encara que tinguin el mateix nom és mitjançant la sobrecàrrega. Si els tipus del paràmetres són diferents, els mètodes són diferents. No és vàlid sobrecarregar pel tipus de l'atribut, donat que aquest es pot repetir.

La solució ve de la mà dels templates i s'inspira en les tècniques de metaprogramació. Cada atribut té associat un enter que indica la posició en la taula de descripció. Donat que els enters serveixen com a paràmetres template podem fer que el tipus que necessitem per la sobrecàrrega sigui un template instanciat per aquest enter.

Això ens deslliga dels noms, però com es pot cridar des del mètode comú tots i cadascun dels mètodes per a cada atribut? Un altre cop es poden fer servir les tècniques de metaprogramació: Des del mètode comú, es crida al mètode sobrecarregat per `AttributePosition<0>`. A partir d'aquí cada mètode de la cadena, després de fer la tasca pel seu atribut, cridarà al mètode següent de la cadena que serà el sobrecarregat per `AttributePosition<Index+1>`. La cadena finalitza quan es crida al mètode sobrecarregat per `AttributePosition<N>` on N és el nombre d'atributs. Aquest darrer mètode es defineix com un mètode buit i s'expandeix a dintre de la macro comuna,

```
#define DYN_ATTRIBUTE(N, ACCESS, TYPE, NAME) \
ACCESS: \
    void Do##NAME() { \
          /* Here we have available all the attribute information */ \
          std::cout << N <<"\t"#TYPE"\t"#NAME << std::endl; \
    } \
private: \
    void DoChainedAttr(AttributePosition<N>* ) { \
          Do##NAME(); \
          DoChainedAttr((AttributePosition<(N)+1>*)NULL); \
```

```
     } \
```

Llistat 5. Mecanismes d'encadenament de mètodes: Macro d'atribut

Totes les crides que es fan amb l'encadenament de mètodes, en el fons no tenen cap penalització en l'eficiència. De fet, aquestes crides són inline i la crida a la funció que inicia la cadena equival a posar les accions corresponents a cada atribut, una darrera de l'altre.

El que sí que era problemàtic amb els mètodes encadenats eren els errors que donava el compilador quan hi havia alguna inconsistència en els índexs que es posaven als atributs.

Vam fer servir un xic més de metaprogramació per generar errors de compilació. Les definicions per defecte dels chained methods es fan servir per detectar quan s'havia produït una inconsistència en els índexs i, amb algunes tècniques de metaprogramació que es veuen al llistat 5 i 6 es pot informar a l'usuari de quin tipus d'error és (un índex repetit, el·lidit o fora de rang) i quin ha estat l'índex erroni.

```
#define DYNAMIC_TYPE(CLASS_NAME,N) \
public: \
     enum { eNumAttr= N }; \
public: \
     /** Do all Dynamic Attributes */ \
     void DoAll () { \
            DoChainedAttr((AttributePosition<0>*)NULL); \
     } \
private: \
     /** Method chain terminator */ \
     void DoChainedAttr (AttributePosition<N>*) { \
     } \
     /** Undefined link for the Do method chain (Do) */ \
     template <unsigned int NAttrib> \
     void DoChainedAttr (AttributePosition<NAttrib>*a) { \
            CheckAttribute ((AttributePosition<NAttrib>::InboundsCheck*)NULL, \
                            (AttributePosition<NAttrib>*)NULL); \
     }\
private: \
     template <unsigned int NAttrib> \
     class AttributePosition : public DynamicType::AttributePositionBase<NAttrib> { \
            public: \
                   typedef StaticBool<!(NAttrib>=N)> InboundsCheck; \
     }; \
     /** Instantiated whenever a Attribute number is out of range.
      * Gives a compilation error message.
      */ \
     template <unsigned int NAttrib> \
     void CheckAttribute (StaticFalse*inRange,AttributePosition<NAttrib>*a) { \
            AttributePosition<(NAttrib)-1>* previous; \
            previous->CompilationError_AttributePositionOutOfBounds(); \
     }\
     /**
      * Instantiated whenever a Attribute number is left.
      * Gives a compilation error message.
      */ \
     template <unsigned int NAttrib> \
```

```
void CheckAttribute (StaticTrue*inRange,AttributePosition<NAttrib>*a) { \
       a->CompilationError_AttributeNotDefined(); \
} \
```

Llistat 6. Mecanisme d'encadenament de mètodes: Macro de classe

D'aquesta manera, es va aconseguir convertir quelcom que era una disfunció a les noves macros respecte les antigues (apareixien errors difícils de llegir) en una funcionalitat nova (comprovació de consistència en els índexs d'atribut en temps de compilació, en comptes d'execució). Els chained methods han aportat altres millores als *dynamic types*, donat que el mateix entrebanc ens haviem trobat per la serialització, ja havia obstaculitzat algunes altres funcionalitats.

Per exemple, ara la taula de descripció s'omple des dels constructors, hi ha mètodes per instanciar i desinstanciar tots els atributs a la vegada (`AddAll()`, `RemoveAll()`) i s'han fet algunes optimitzacions en la forma de reservar espai a la taula de dades.

Els dos llistats anteriors no són implementacions complertes però il·lustren tot el que s'ha explicat abans respecte l'encadenament de mètodes i la detecció d'errors amb una hipotètica cadena de mètodes `DoAll` que crida els corresponents `DoX` on la `X` és el nom de cadascun dels atributs.

Activació de l'objecte en XML

L'activació de l'objecte a partir d'un arbre XML, s'ha fet de forma molt simètrica a la passivació. És a dir, utilitzant el patró de disseny adaptador, que incorpora els mètodes `StoreOn` i `LoadFrom` a l'objecte. El problema principal era trobar una manera general i flexible d'escollir el tipus d'adaptador per cada tipus d'un *dynamic type*.

En la nova versió, la distinció entre `Component` o no es fa de forma molt semblant a com es feia abans: sobrecarregant amb un paràmetre que pot ser punter a `void` o punter a `Component`.

La diferència està en la forma de detectar quins són els tipus bàsics amb els que es pot fer servir un `XMLStaticAdapter`. Ara és molt més flexible: es fa servir una solució semblant a

les interfícies de McNamara [18]: explicitava el fet de que un tipus modelava un concepte per poder, després, fer comprovacions i prendre decissions en temps de compilació.

En la seva solució, l'explicitació de la pertanyença a un concepte, es feia dins de la definició de la classe. En els cas que ens ocupa, és clar que això no es pot fer així perquè molts dels tipus que hauran de modelar el concepte són tipus elementals de C o tipus externs que no es poden tocar.

L'aproximació que s'ha implementat és explicitar que el tipus modela el concepte, fora de la declaració del tipus.

2.2.2 Classes de processament

2.2.2.1 Arquitectura

CLAM fa servir bàsicament una arquitectura de processament modular. El sistema es pot descriure com una xarxa d'elements de procés o mòduls pels quals van passant les dades com a una cadena de muntatge per obtenir el producte final.

Així doncs, la clau per definir l'arquitectura està en definir:

- Un encapsulament pels algorismes
- Un format per les dades, i
- Els patrons de flux.

2.2.2.3 Encapsulament

Desde bon principi va es va veure clar, que per aconseguir l'objectiu de reusabilitat de la llibreria, calia esmerar-se molt en la forma d'encapsular els algorismes de processament i en definir una interfície i comportament de la classe encapsuladora (`Processing`).

Un punt clau va ser decidir que un sistema de processat (deixant de banda el GUI) es composaria només de tres elements conceptuals diferents: Els objectes de processament, les dades que aquests maneguen i la lògica per gestionar aquestes dades i les execucions dels objectes de processent. D'aquesta lògica en diem *flow control* encara que no estigui encapsulada en cap classe sinó que sigui programada *adhoc* per un sistema concret.

Per tant, no s'admet manipular les dades de processament en funcions que no estiguin correctament encapsulades en un objecte de processament. Aquesta restricció facilita molt que un usuari pugui entendre un sistema que ha escrit una altra persona usant CLAM, i per tant que el pugui modificar i reusar.

Un altre aspecte important per assolir una reusabilitat i una manera comuna de fer les coses, és definir la interfície dels objectes de processament el més neta i petita possible.

Bàsicament, en temps d'execució a un objecte de processament el podem 1) **configurar**, 2) li podem dir que **executi** un pas de l'algorisme, 3) li podem **enviar controls** (missatges orientats a l'event. En parlaré a la secció §2.3.4) i 4) el podem fer **canviar d'*estat d'execució*** (executant-se, parat, desabilitat...)

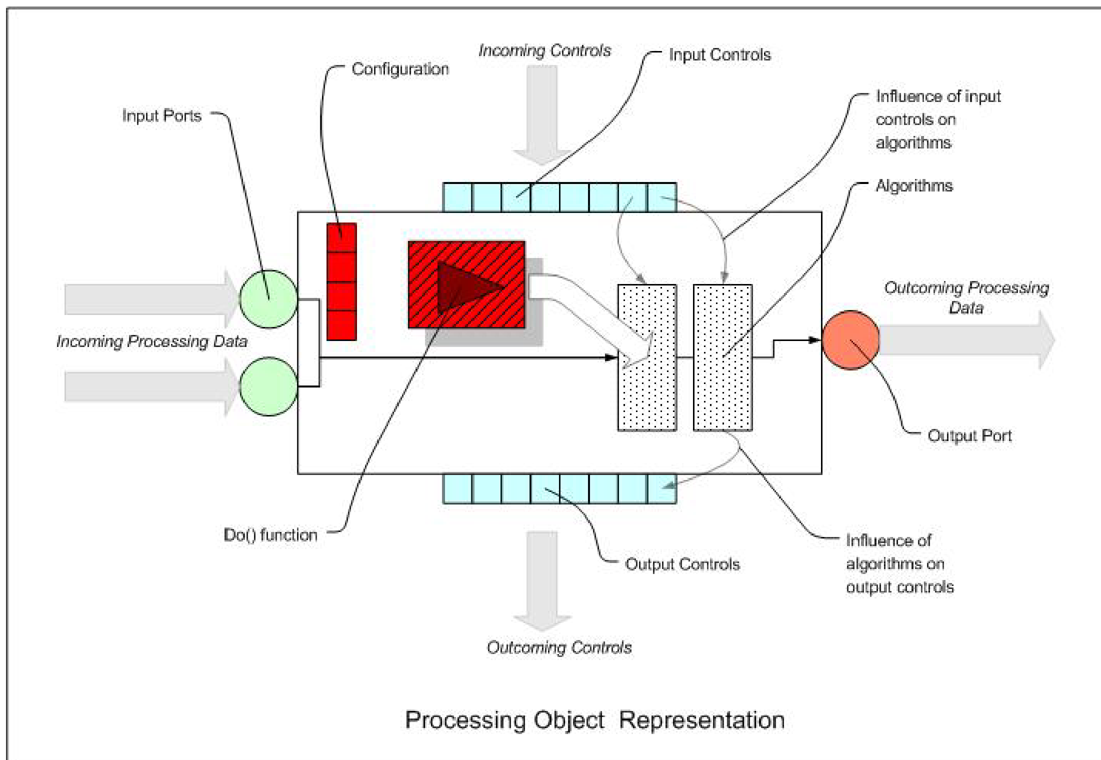

Figura 10. Arquitectura d'un objecte de processament

Per aconseguir els objectius, vam posar ènfasi en que l'interfície fós la mínima possible i alhora donés la suficient flexibilitat. Voliem evitar:

- que hi haguessin vàries maneres de crear aquests objectes (molts constructors)
- que hi haguessin vàries maneres d'executar-lo.
- que hi haguessin vàries maneres de afectar/consultar el seu estat (les seves dades en general)

Com s'observa a la Figura 10 un objecte de processament es composa de:

- Uns ports d'entrada i sortida: que serveixen per accedir a les dades d'entrada i sortida i per comunicacions més avançades:
    - Li serveix a l'objecte de processament, per informar a l'exteriror les seves necessitats de dades. (per exemple el nombre de *tokens* que vol llegir en cada pas d'execució)

  - Li serveix al flow control, per informar del *prototip* de dades que apartir d'aquell moment li arribaran. D'aquesta manera l'objecte de processament, si està preparat així, pot portar a terme optimitzacions[1].

- Un objecte de configuració, el qual encapsula totes les dades necessàries per configurar l'objecte de processament. (veure secció §2.2.2.5)

- Un o varis algorismes de processament.

- Un mètode `Do()` sense paràmetres que executa l'algorisme usant les dades accessibles als ports.

- Uns controls d'entrada i sortida. Els d'entrada permeten canviar paràmetres de l'algorisme desde fora, i els de sortida permeten enviar controls a altres objectes de processament.

[1] Un exemple d'aquestes optimitzacions és el següent: un objecte de processament que es dedica a multiplicar objectes de tipus Spectrum se l'informa a través dels seus dos ports d'entrada que li arribaran sempre Spectrums codificats en complexes. Aleshores, aquest aplica un algorime òptim per aquesta reprentació en comptes del general i sense haver d'inspeccionar el tipus de codificació cada dada Spectrum d'entrada.

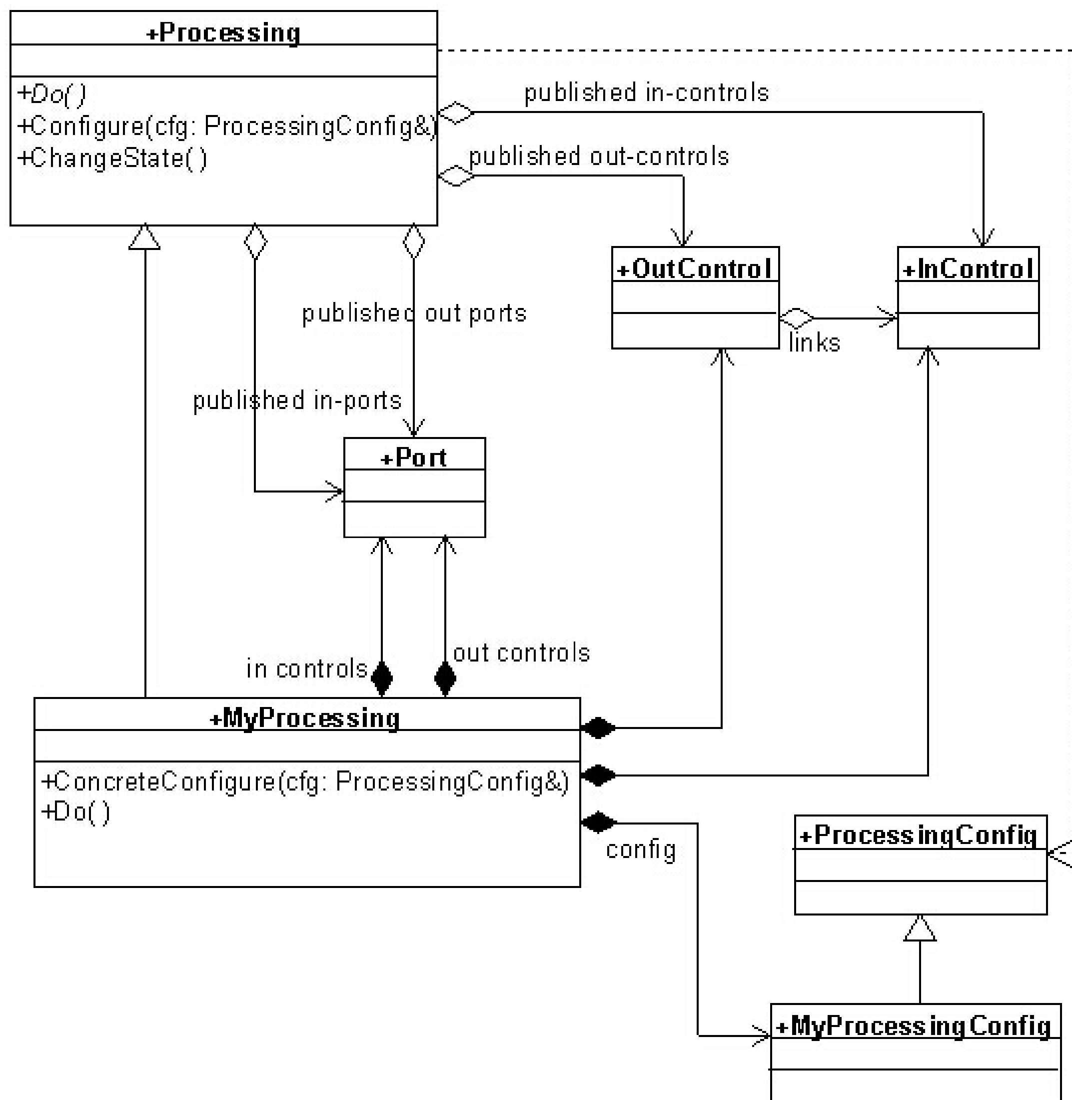

Figura 11 Diagrama de classes dels processaments

Cal dir que en el mode no-supervisat en el qual estem treballant actualment, no és obligatori utilitzar els ports. Alternativament, podem tenir sobrecarregues del `Do` amb paràmetres que usem per passar les dades que altrament haurien estat accedides desde els ports.

Aquesta alternativa és acceptada perquè és una mica més eficient que accedir les dades desde el port (almenys per una indirecció de memòria), i cal que quan sigui necessari, poguem optimitzar un sistema al màxim.

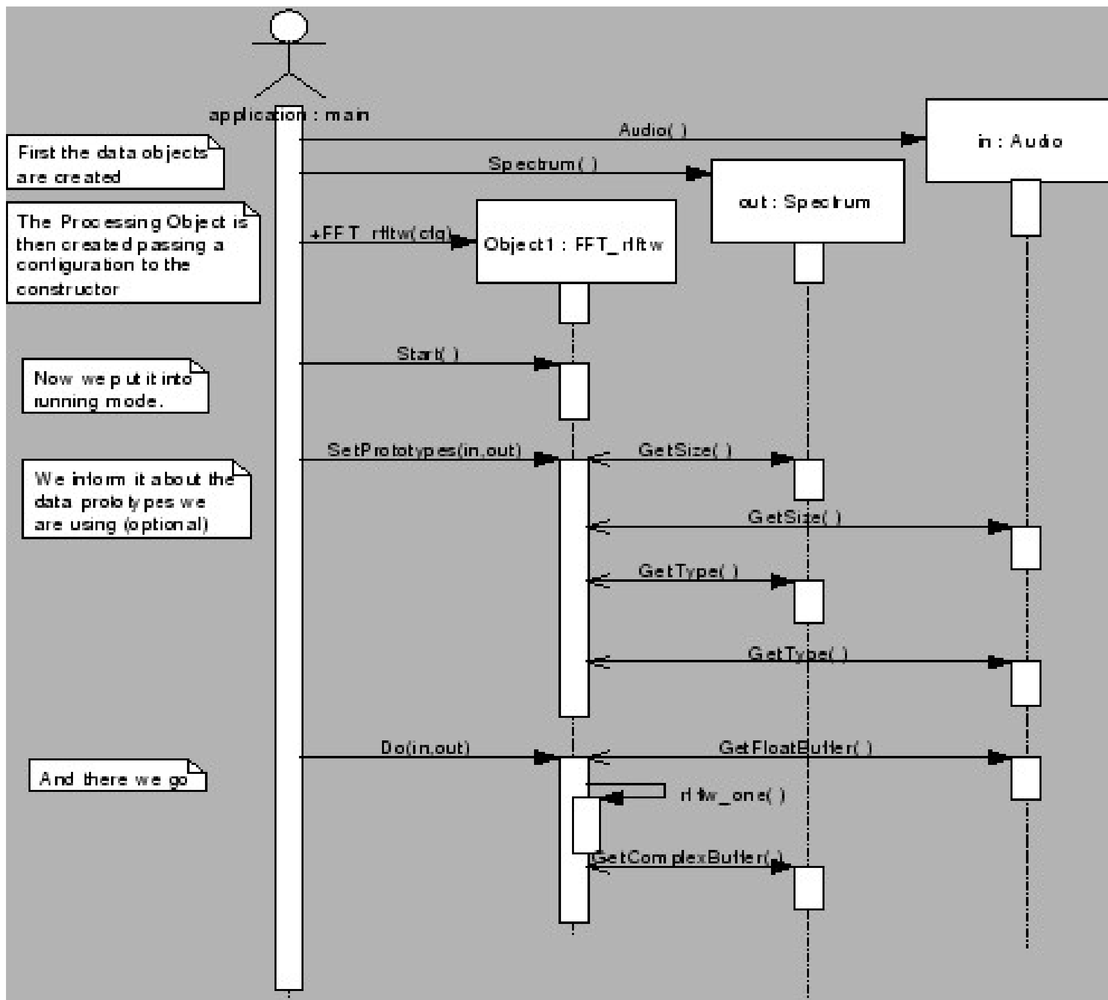

Figura 12. Inicialització i execució d'un processament 'FFT', amb l'estil del mode no-supervisat. (Do(...) amb paràmetres)

Però en canvi, és preferible l'altre (i de fet en l'última etapa de CLAM és obligat proporcionar el `Do` sense paràmetre i declarar els ports) perquè ens permet fer introspecció en els sistema (inspeccionar les dades d'entrada i sortida de qualsevol objecte de processament).

De fet, els ports són elements totalment necessaris dins el disseny del mode supervisat, però també seran molt útils en el mode supervisat quan tinguem funcionant una eina de *debugging*, que actualment està en estat molt avançat, la qual permetrà a qualsevol sistema CLAM, simplement establint uns punts de trencament (*breakpoints*) al bucle d'execucions, controlar l'execució del sistema. Podrem parar-la en qualsevol moment i navegar per l'estructura d'objectes de processament (veure el següent punt sobre l'escalabilitat) i visualitzar les dades o bé en format XML o bé amb vistes del GUI.

2.2.2.4 Escalabilitat

Ja que CLAM està pensada com una eina que ha de ser fàcil d'usar i altament reusable, una eina clau és l'escabilitat, o la capacitat de definir nous objectes de processament per composició: un sistema de processament es pot encapsular en un nou objecte de processament tot definint-li uns ports, uns controls, una configuració i un control del fluxe.

Tot i que el següent diagrama no és de CLAM sinó que són pantalles del Simulink [1], mostra clarament el concepte de l'escalabilitat.

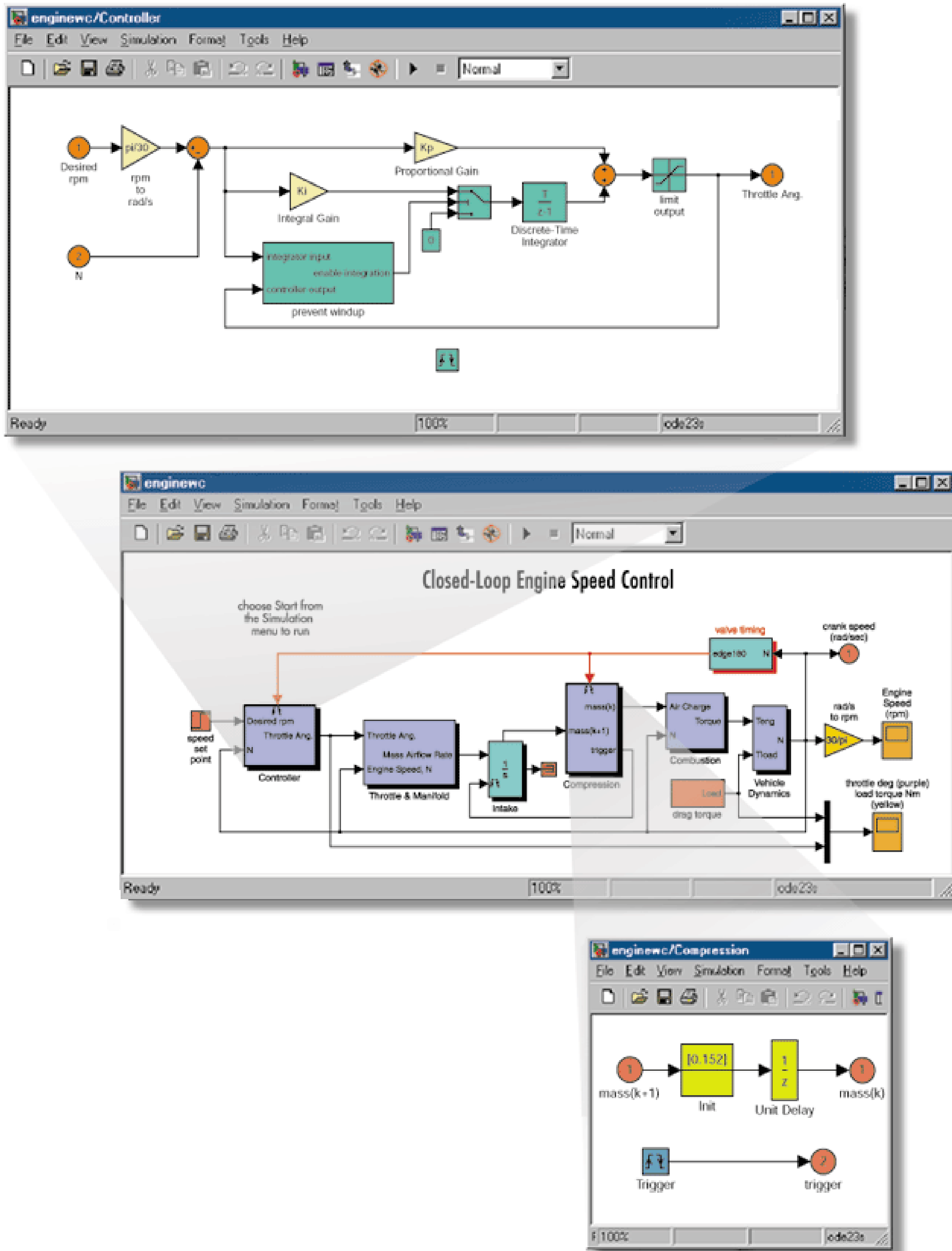

Figura 13.Escalabilitat. Un subsistema es pot veure com un objecte de processament individual.

En el futur mode supervisat tindrem components no compilats, simplement amb una definició de la xarxa i la seva interfície, que podran ser incorporats a un sistema, i vistos de la mateixa

manera que els components compilats. Això permetrà una gran facilitat a l'hora d'extendre els components.

De moment però, ens conformem amb la versió compilada. Disposem d'una classe base `ProcessingObjectComposite` que deriva de `Processing`, i de la qual, al seu torn, cal derivar tots els subsistemes que volguem crear. L'avantage de restringir a usar aquesta classe com a base és que aquesta ja proporciona un bon nombre de facilitats com són: gestió de noms dels objectes de processament que estan a dintre (usem un sistema semblant als noms dels fitxers amb la ruta dels directoris), gestió de quins controls i ports volem publicar a l'exterior i gestió automàtica dels *estats d'execució* dels objectes de processament interns.

En un ProcessingComposite concret, és obligatori programar el mètode `Do`. Ens serveix per encapsular-hi el control de fluxe del subsistema.

El nombre de nivells possibles de la composició d'objectes de processament és arbitrari.

2.2.2.5 Configuració i control

Es va decidir que tot objecte de configuració havia de ser *dynamic type*. Això ens dóna una uniformitat en com accedir a les dades, una manera de treballar amb configuracions amb atributs amb valors per defecte (aquesta és la semàntica que tenen els atributs no instanciats), i el més important: la capacitat , que ens ve de franc amb els *dynamic types*, de guardar i recuperar configuracions en fitxers XML. Característica que en els sistemes implementats s'ha mostrat molt valuosa.

El punt d'entrada per configurar un objecte de processament és el mètode `Configure` que està definit a la classe base `Processing`. El què es fa a la base és comprovar que realment és 'legal' configurar-lo, és a dir, que no estem en estat *running*, manegarà el nom de l'objecte de processament i finalment cridarà un mètode que cal definir-lo a la classe concreta: el `ConcreteConfigure`. Aquí és on s'usarà la informació que duu la configuració.

Per facilitar l'ús, incorporem dos constructors a cada `Processing` concret: un amb una configuració com a paràmetre, i en altre sense paràmetre que simplement configura l'objecte utilitzant la configuració per defecte, la qual tota configuració proporciona.

Com s'ha dit, un objecte de processamant no pot ser reconfigurat mentre està en estat *running*, per tant per certes operatives necessitem uns paràmetres que puguin ser modificats en qualsevol moment: i això s'aconsegueix amb els controls.

2.2.2.6 Estats d'un *processing*

Els objectes de processament estan en qualsevol moment, en un cert *estat d'execució*. La millor manera de mostrar-ho és amb un diagrama d'estats UML.

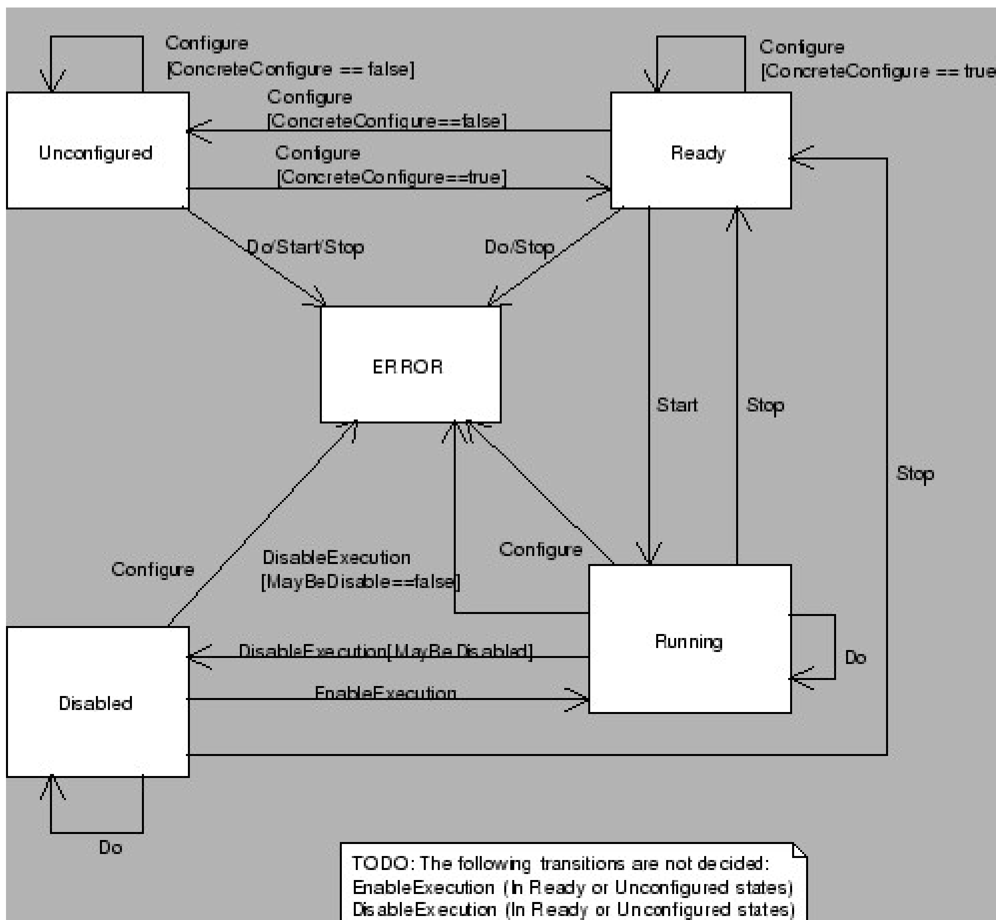

Figura 14. Estats d'execució d'un processing

Detalls del seu funcionament es poden trobar a la secció 30 del manual d'usuari (Apendix A).

Però la raó de ser d'aquests estats és molt senzilla: en general la configuració d'un objecte de processament pot ser una mètode costós; pot implicar creació d'estructures de dades, inicialitzacions d'aquestes, procediments inicials, pre-càlculs...

Per assolir una eficiència que permeti el processament en temps real de l'audio amb garanties, hem de restringir els moments on es poden fer aquestes configuracions costoses. Per això, amb els estats, l'usuari que crea el sistema, ha d'explícitament *aturar* l'execució per fer qualsevol configuració.

Evidentment, l'*aturada* del sistema pot implicar que alguns objectes de processaments, en particular els dels extrems del procés, que obtenen i dipositen l'audio al hardware, hagin de fer re-inicialitzacions de buffers circulars, etc. Però això, és infinitament més desitjable que una 'penjada' causada per no processar les dades en el temps requerit.

2.2.3 Classes de dades de processament

Els projectes que es fan al grup de recerca, i que CLAM ha de suportar, són sobretot projectes que treballen en la transició entre els dominis sintàctics i semàntics de l'audio. És a dir, en un extrem hi ha el processament de senyal, a baix nivell, basat en propietats físiques, i a l'altre extrem tenim un processament de continguts, a alt nivell, que està basat en la interpretació cognitiva d'aquestes propietats.

Per tant, en un mateix sistema de processat CLAM, poden conviure diferents tipus de dades dins un rang molt ampli: desde un objecte d'audio que encapsula un petit bloc de *samples* (mostres d'audio digitalitzat), fins estructures de dades geràrquiques que contenen la melodia d'un segment de música i descripcions de nivell cognitiu sobre aquesta.

La classe base de CLAM per representar el concepte de les dades generades i consumides pels `Processing`, és precisament `ProcessingData`. Posem la restricció que totes les dades manipulades pels algorismes, estiguin encapsulades en una subclasse de `ProcessingData`. És

a dir, els paràmetres dels `Do()` i els objectes referenciats pels ports només poden ser subclasses de `ProcessingData`.

Exemples d'aquestes classes disponibles a la llibreria inclouen: Spectrum, Audio, SpectralPeakArray, Fundamental, Segment, Frame...

Cal dir que la classe base, `ProcessingData`, és molt lleugera, en el sentit que no implementa ni força de derivació de cap mètode com sí que ho fa la classe `Processing`. De totes maneres hi ha tres raons per la seva existènica és doncs: tenir un tipus base per poder categoritzar les classes corresponents a aquest 'concepte', poder manegar referències a qualsevol tipus de dada de processament (i no altres objectes), i forçar que totes elles siguin *dynamic type* (homogeneïtzació de la interfície, XML...)

2.2.3.1 Aspectes estructurals

Qualsevol classe de dades de processament és de fet, un classe concreta *dynamic type*. La major part dels seus atributs són definits amb les macros d'atributs dinàmics. Aconseguim forçar la derivació de *dynamic type*, perquè de fet, la base `ProcessingData` ja deriva de `DynamicType`.

Per una part, això es bo perquè en proporciona uns accessors (Get/Set) de forma automàtica, i així homogeneitzem unes interfícies (les de les classes de dades) que en algunes classes pot ser molt gran. A més, com s'ha comentat en els objectius dels dynamic types, això ens permet treballar amb atributs instanciats i desinstanciats, la qual cosa es pot fer durant el processat.

Un altre motiu perquè tota classe de dades sigui dynamic type, és que tenir una implementació per defecte automàticament implementada de la serialització en XML d'aquestes dades és molt important. Ja sigui per ajudar al desenvolupament, o per aconseguir una entrada i sortida adecuada per un sistema.

2.2.3.2 Consideracions d'eficiència

Sovint les classes de dades ténen atributs dinàmics que són buffers (per exemple l'`Audio` necessita un buffer de mostres en domini temporal, i un `Spectrum` pot contenir un buffer de complexes.).

Aquestes classes ofereixen accessors pels elements individuals d'aquest buffer, els quals poden ser molt útils en fase de desenvolupament, però en canvi comporta alguna ineficiència: cal accedir al buffer cada vegada que es necessita accedir a l'element, la qual cosa comporta un parell d'indireccions addicionals. (ja que per retornar l'atribut, el *dynamic type* ha d'accedir a les seves taules d'instanciacions i de dades).

Això, de fet no és un problema, perquè pot ser fàcilment evitat quan busquem eficiència en un algorisme en que per cada `Do` s'hagi d'accedir multipes vegades al buffer (cas molt habitual) simplement es guarda una referència a l'objecte buffer, i s'accedeix al buffer de manera òptima, tal com es mostra en el següent exemple:

```
void DoubleMagnitude(const Spectrum& spec)
{
    int i;
    Tdata sum=0;
    DataArray& mag = spec.GetMagBuffer();
    int size = spec.GetSize();
    for (i=0; i<size; i++)
        mag[i]*=2;
}
```

2.2.3.3 Constructors i configuracions.

Al ser les classes de dades unes classes molt centrals en el processament, aquestes han de complir el doble objectiu de ser eficients i usables. Això en part s'aconsegueix de les dues maneres que he explicat: homogeneïtzant la interfície i treballant amb referències a buffers si és el cas.

De totes maneres, l'estructura d'aquestes classes s'ha vist modificada darrerament (al mes l'Abril) per aconseguir major usabilitat i eficiència.

Inicialment teniem una aproximació igual que la dels objectes de processament en quant als constructors i configuracions: obligavem que només hi hagués un constructor amb paràmetre, i que aquest fos un objecte de configuració específic per la classe.

D'aquesta manera seguiem una única manera de fer les coses (entre els processaments i les dades), i facilitavem la construcció de dades que necessiten molts paràmetres de configuració.

Addicionalment, de la mateixa manera que els processaments, demanavem implementar a la classe de dades, la interfície de `GetConfig()`, que retornava un objecte de configuració del qual l'objecte de dades n'és propietari, amb les dades de la configuració actualitzada.

Això implicava haver de guardar un objecte de configuració cosa que tenia varis problemes:

- Per objectres de dades petits dels quals necessitem grans quantitats, això representa una gran ineficiència en memòria. Es podia, però, aplicar el patró de disseny *Flyweight* per solventar-ho.

- O bé dupliquem els atributs que es troben a la configuració en altres atributs de la classe de dades, que no és eficient en espai i a més requereix sincronització entre els dos, o bé usem directament els atributs de la configuració la qual cosa no és molt eficient ja que requereix accedir a dos atributs dinàmics: primer a l'objecte configuració i després a l'atribut concret.

La revisió de que parlava, va solucionar aquest problema seguint les següents idees:

- No és obligatori que cada classe de dades tingui la seva classe de configuració. La regla que seguim és: només la tindran aquelles classes que tenen un nombre prou gran de paràmetres de configuració que facin necessària més d'una sobrecàrrega del constructor.

- Per tant, en les classes sense configuració s'usarà un constructor amb paràmetres (i aquest poden tenir valors per defecte, tal com ho permet el C++)

- Tots i només aquells atributs de 'configuració' considerats necessaris seran guardats directament dins de la classe de dades.

- Les classes que tinguin configuració, implementaran un GetConfig(...) amb un paràmetre d'entrada i sortida que serà la configuració concreta que serà sincronitzada amb l'estat de l'objecte de dades. D'aquesta manera, l'objecte de dades no s'ha de fer càrrec de gestionar la memòria de l'objecte de configuració.

2.2.3.4 Un exemple: Segment

A l'apartat 41 del manual (apèndix A), hi ha una descripció de les classes de dades més important. En aquesta memòria no em proposo explicar el disseny concret de cap d'elles, ja que aquestes decisions estan molt guiades per la informació sobre el domini de processament de l'audio i música. De totes maneres, al manual sí que se'n parlar.

Només per donar una idea el tipus de dades amb que ens podem 'topar' en un processament de CLAM, incloc un diagrama de classes UML de la classe de dades de processament `Segment`. Cal tenir en comte que en el gràfic s'obvia que totes les classes que apareixen deriven de la classe base `ProcessingData`.

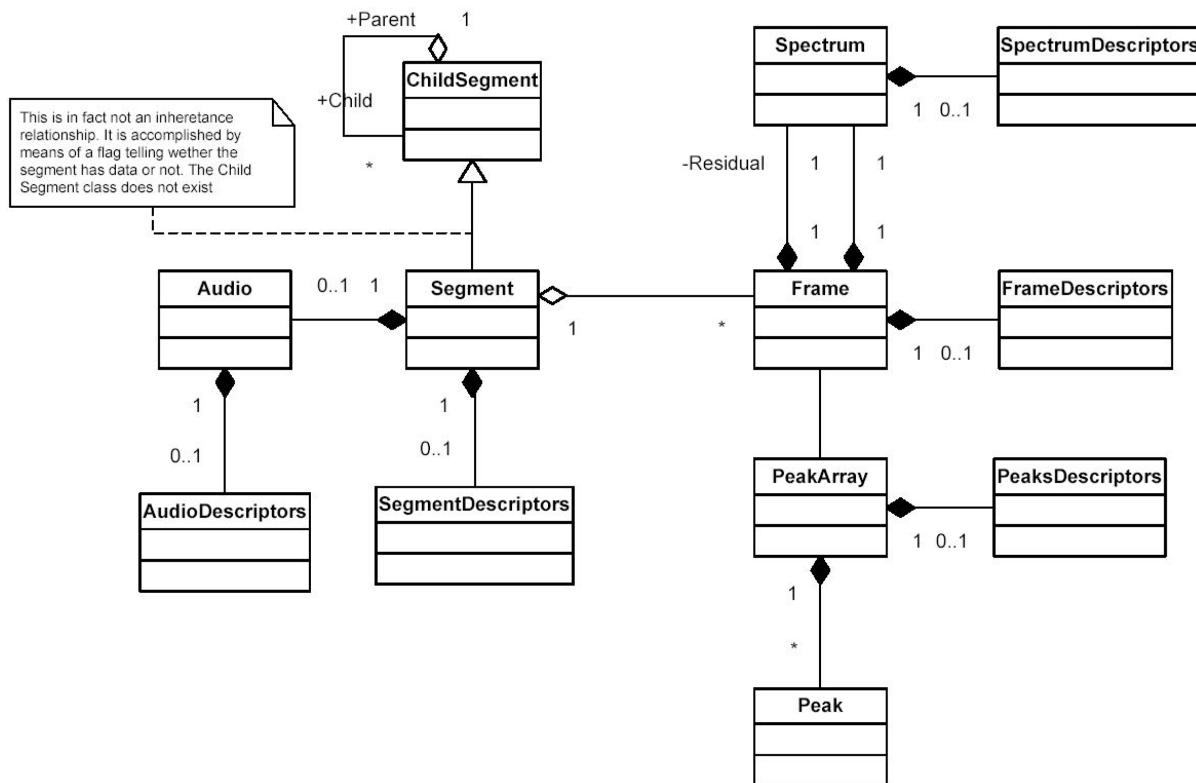

Figura 15. Diagrama de classes de Segment

2.2.4 Controls

Algunes classes de processament necessiten que entitats externes canviin el comportament de l'objecte de forma assíncrona [1] durant la seva execucció. Els controls d'entrada són els mecanismes per realitzar aquest tipus de canvis.

D'altra banda, les classes de processament poden ser utilitzades per detectar algun tipus d'event. Els controls de sortida permeten fer notificacions d'events assíncrons.

[1] *Assíncron*, en aquest context significa que no respecta la freqüència d'execucions; un control de sortida pot emetre un event sense haver d'esperar el torn d'execució de l'objecte de processament al qual pertany.

Per tant, a l'hora de dissenyar un objecte de processament concret, haurem de pensar en les seves necessitats de comunicació assíncrona, i declarar un atribut control d'entrada (`InControl`), per cada tipus de missatge que volguem que el processament rebi. De la mateixa manera, per cada tipus de missatge de sortida, caldrà un atribut de tipus control de sortida (`OutControl`).

Una aplicació pot connectar un controls de sortida amb un o més controls d'entrada. Als controls d'entrada els hi podem associar un mètode de servei o de *callback*, i aquests poden propagar un control entrant enviant-lo per un (o més) control(s) de sortida, que al seu torn poden estar connectats a altres controls d'entrada.

D'aquesta manera obtenim un segon tipus de fluxe: el de controls que té les següents característiques contraposades al fluxe de senyal o dades:

Flux de control	Flux de dades
assíncron	síncron
Orientat a l'event	Orientat a freqüències d'execució. (mes o menys constants)
Propagació automàtica dels events. (el bucle de processament ha d'esperar que acabin totes les crides encadenades)	Un flow control ha d'executar els processos explícitament (en l'ordre correcte)
Topologia poc flexible (no admet realimentacions)	Topologia molt flexible (permet realimentacions, delays...)

Taula 1Característiques dels fluxes de dade i controls

Es va optar per dissenyar i implementar els controls, perquè donava solució a una sèrie de requeriments concrets que havien estat extrets d'exemple d'aplicació.

Per exemple, en graf d'objectes de processaments en que tenim un mòdul d'anàlisi de baix nivell i un altre d'anàlisi d'alt nivell, tot i trobar-se en extrems oposats del graf, volem que tinguin una estreta col.laboració o que 'dialoguin'[1] entre ells, però sense acoblar les interfícies de les dues classes, de tal manera que siguin intercanviables per altres mòduls. I és clar, que aquesta comunicació ha de ser instantànea, si esperem el torn d'execució ja seria massa tard.

Per detalls en l'ús dels controls i exemples de codi, mirar la secció 31 del manual. (Apèndix A)

[1] Un exemple concret d'aquest cas són els processos 'FFT' i 'detector de pitch'. El segon, segons el pitch que detecta, pot voler ajustar algun paràmetre (mida de la finestra) de la 'FFT', per millorar les subseqüents deteccions del pitch.

2.2.4.1 Implementació

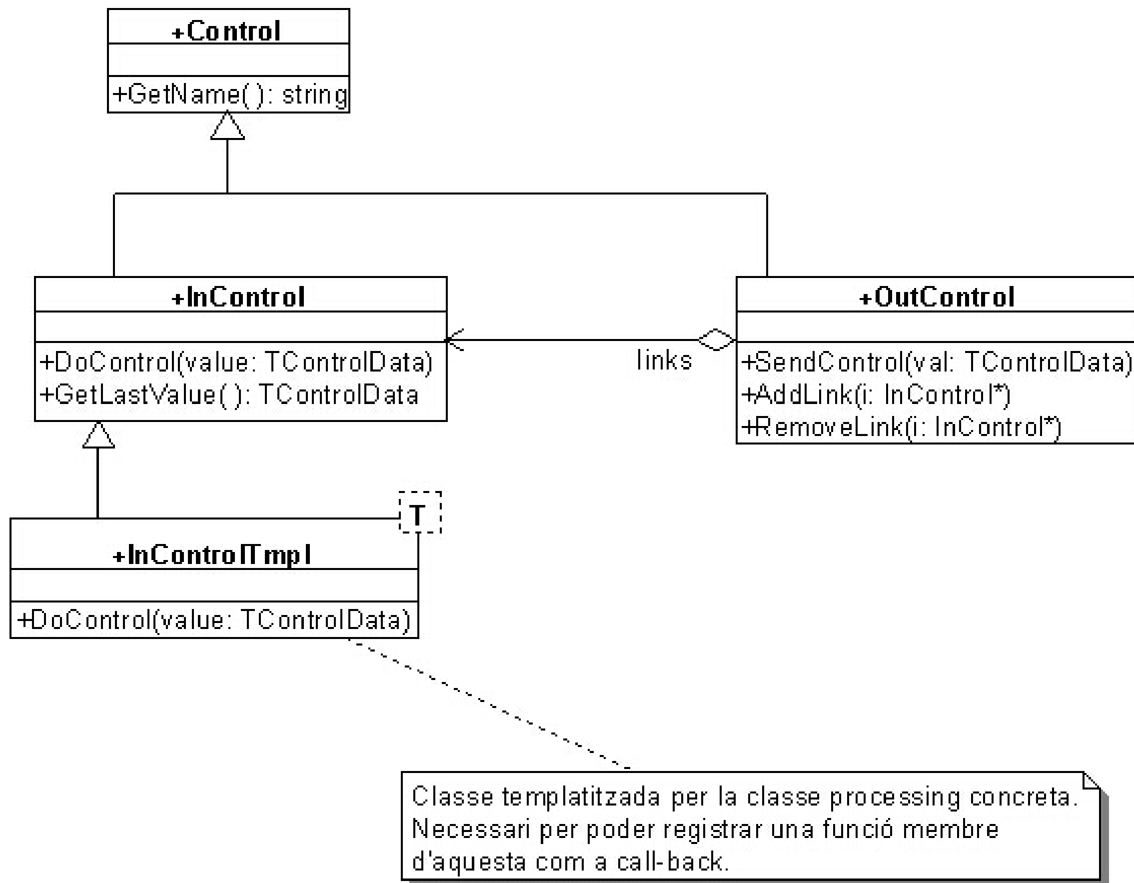

Figura 16Diagrama de classes dels controls

Un aspecte rellevant en el disseny i la implementació dels controls, és com s'ha resolt el comportament que ha de tenir un InControl (l'entitat que rep els events) davant l'arribada d'un control.

Hi ha un comportament per defecte molt senzill: quan es reb un control, mitjançant el missatge `DoControl(val)` enviat a un `InControl`, es guarda aquest valor, i l'objecte de processament 'pare' el podrà consultar amb el mètode `GetLastValue()`.

En el procés d'anar definint els requeriments concrets referents als controls d'un objecte de processament, es va decidir que el següent seria molt desitjable:

- En algunes situacions, quan ens arriba un control d'entrada, voldrem enviar-ne de sortida (sense haver d'esperar el proper temps d'execució del processing)
- Podem voler fer un seguiment dels valors que ens arriben per un control entremig de dues execucións del Do()
- Si tenim molts controls en una classe de processament, volem evitar de fer una comprovació de tots ells per veure si han canviat cada vegada que el mètode Do és cridat.

Controls d'entrada amb rutina de servei (*call-back*)

Aquesta flexibilitat s'ha aconseguit a base de poder associar una rutina de servei definida dins la classe processing concreta, pel seu programador.

En l'implementació d'aquest *call-back*, era important que el codi quedés encapsulat junt amb els altres elements del processament, és a dir, dins la classe processing concreta. A més, també calia que des del *call-back* es pogués accedir a l'estat de l'objecte de processament i que a poder ser fos un mètode virtual que pogués ser redefinit en cas que una classe de processament necessités ser subclassificada.

Per tant es va descartar l'opció de usar com a *call-back* una funció *estàtica* (no membre d'una classe).

Per sort, vaig comprovar que el C++ suportava guardar referències a funcions membres i executar-les en qualsevol instant. L'única dificultat a resoldre era deguda a que el tipus de tal referència es defineix a partir de la classe concreta que declara el mètode, i alhora d'executar-lo es necessitava un punter a l'objecte de processament concret.

La solució escollida, com s'ha vist a la Figura 16, consisteix en tenir una classe `InControl` que ofereix el comportament senzill (guarda l'últim valor), i a més, derivar-li una classe `InControlTmpl` que s'ha d'especialitzar amb la classe de processament 'pare', alhora de declarar el control d'entrada.

Per clarificar, podem veure la part rellevant de la implementació de `InControlTmpl`.

```
template<class ProcObj>
class InControlTmpl : public InControl
{
private:
    typedef int (ProcObj::*TPtrMemberFunc)(TControlData);
    TPtrMemberFunc mFunc;
    ProcObj* mProcObj;

int DoControl(TControlData val)
{
    InControl::DoControl(val);
    if(mFunc)
          return (mProcObj->*mFunc)(val); // executem la funció membr.
    else
          return 0;
}
...
```

I finalment, vegem una declaració de controls en una classe processing d'exemple:

```
class MyProcObj : public Processing
{
// Attributes
    MyProcConf mConfig;
private:
    InControlTmpl<MyProcObj> mInPitch;
    InControlTmpl<MyProcObj> mInAmplitude;

    OutControl mOutNoteOn;
    OutControl mOutNoteOff;
    bool ConcreteConfigure(const ProcessingConfig &cfg) {return true;}
// Constructor/Destructor
public:
    MyProcObj(const MyProcConf &c) :
          mInPitch("Pitch", this, &MyProcObj::DoInPitchControl),
          mInAmplitude("Amplitude",this,
                      &MyProcObj::DoInAmplitudeControl),

          mOutNoteOn("NoteOn", this),
          mOutNoteOff("NoteOff", this)
...
```

Notem que els constructors dels controls, necessiten com a arguments el nom, el punter a l'objecte 'pare', i en cas que sigui un *InControlTmpl*, cal passar una referència al mètode (amb una sintaxi bastant obscura). La raó de passar un punter al 'pare' és doble:

- Permet al control que envii un missatge a l'objecte de processament demanant-li que el 'publiqui' (és a dir: que l'objecte de processament afegeixi al control al contenidor

controls publicats), i que, per tant, sigui accessible desde l'interfície abstracta de `Processing`. A la Figura 11 (Diagrama de classes dels processaments), es pot veure aquesta agregació situada a la base. El motiu principal de tenir un accés als controls desde la classe abstracta, és que ens permet fer fàcilment introspecció del sistema.

- En el cas d'un `InControlTmpl`, aquest objecte es guarda el punter al 'pare' ja que el necessita per executar el `DoControl`(), concretament la funció membre que fa de *call-back*.

Cal dir que el mecanisme de declarar i publicar controls l'aplica també amb els ports.

2.3 Exemples d’aplicacions

Aquest apartat explica alguns casos d’ús de la llibreria. En llibreries amb el nivell de generalitat de CLAM és defícil avaluar quan algun aspecte del seu disseny és útil o necessari. Així, aquests casos d’ús ens serviran per projectar sobre casos reals les abstraccions de disseny reflectides a la memòria.

La majoria de casos d’ús són aplicacions que es varen desenvolupar directament amb CLAM. D’altres són aplicacions existents amb anterioritat que s’hi varen portar. El segon cas ens ha estat molt útil per poder establir comparatives amb les implementacions anteriors.

Tot i que algunes aplicacions es varen fer expressament per testar la llibreria, bona part van ser projectes independents que es varen desenvolupar de forma paral·lela a CLAM.

Aquests projectes paral·les han patit els canvis de CLAM durant la seva maduració, però, tenir unes quantes aplicacions funcionant amb CLAM durant el desenvolupament ens ha estat molt útil: ha enfortit la validesa d’algunes propostes que es feien i ha detectat les febleses d’altres.

També varen servir per centrar el disseny sovint massa general, en casos concrets i donar resultats tangibles ja abans del final del projecte.

2.3.1 Spectral delay

La primera aplicació que vam desenvolupar amb CLAM va ser un retard espectral. Vam escollir aquest efecte perquè, incloïa alguns dels elements més comuns als sistemes objectiu, tot i mantenir certa senzillesa.

Aquests elements són l’enfinestrat (un producte de buffers d’audio en domini temporal), la transformàda ràpida de Fourier i la seva inversa per passar del domini temporal a l’espectral, un producte d’espectres per filtrar cadascuna de les bandes, un generador d’espectres en format

BPf (funcions definides per interpolació de punts) per generar els perfils dels filtres de cadascuna de les bandes, un delay general de qualsevol tipus de dades de processment, un sumador amb ensolapament per sumar les bandes tenint en compte l'enfinestrat i obtenir el resultat.

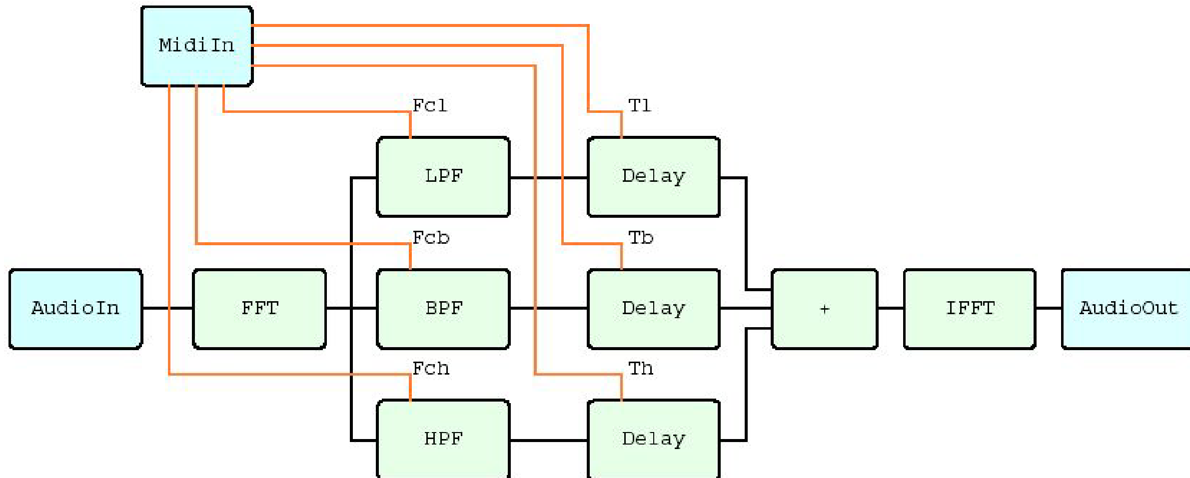

Figura 17. Esquema del Spectral Delay

A més, a aquest senzill esquema se li poden anar afegint moltes de les funcionalitats d'ús general com ara les diferents fonts i destí de dades (formats de diveros fitxers, targeta de so, streaming...), interfície gràfica, manipulació de controls desde l'interfície gràfica o des d'un dispositiu MIDI...

2.3.2 SMS Analisis-Síntesis

Spectral Modeling Synthesis [15] és un algorisme d'anàlisis i resíntesis en el que es basen molts dels desenvolupaments del MTG. L'algorisme extreu del senyal una representació que permet algunes manipulacions més orientades al contingut d'alt nivell, com per exemple el *pitch* o altura d'un so.

La representació alternativa s'extreu mitjançant l'anàlisi espectral del senyal, separant la part harmònica del soroll i, extraient-ne alguns descriptors. Després de manipular aquesta representació es pot tornar a resintetitzar obtenint-ne el resultat final.

2.3.3 Rappid

Rappid és un sistema de temps real basat en CLAM que està pensat per fer-se servir en un espectacle musical en viu. La idea general és que dos intèrprets musicals toquin cadascun un instrument (una viola i una arpa) i un tercer intèrpret, amb l'ordinador generi un instrument virtual a base d'intermodular ambdós senyals amb paràmetres de control extrets dels mateixos dos senyals. El tercer intèrpret controla quins són els paràmetres de control de cada senyal i de quina manera n'afecten al resultat.

És una aplicació real que s'ha fet servir per una obra musical d'en Gabriel Brncic, estrenada a París aquest mes de Juny.

El sistema de processament ha d'extreure paràmetres de control de les senyals. Què són aquests paràmetres de control? Una envolupant de la senyal, les dades del timbre, descriptors de melodia... És clar que aquests paràmetres de control poden tenir molt a veure amb els descriptors que puguem extreure i la capacitat d'aplicar aquests descriptors per modificar l'altre senyal.

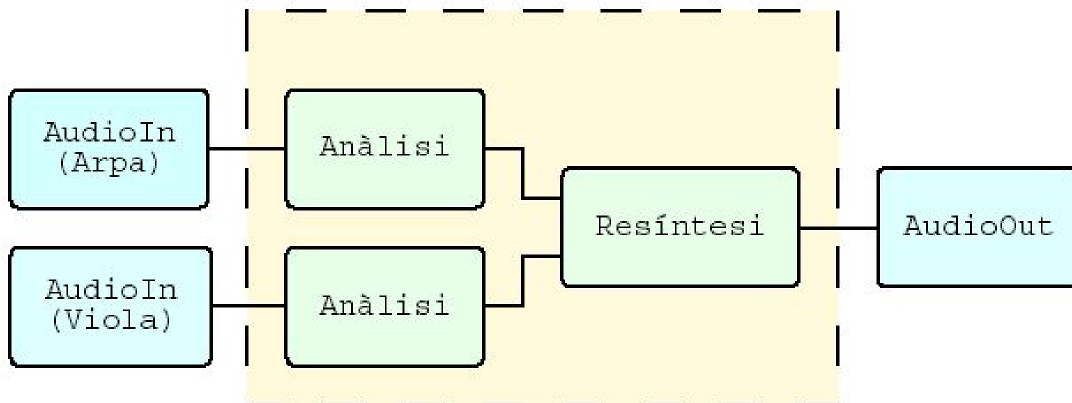

Figura 18. Esquema de Rappid

De fet aquest projecte s'ha fet amb l'ambició d'anar una més enllà d'un sistema concret com el que s'ha descrit. El segon objectiu era el de crear un sistema o aplicació que es pogués aplicar a qualsevol classe de processament CLAM (normalment seran composicions d'aquests), oferint un sistema molt robust i totalment fiable per executar-se en temps real.

Aquest esquelet de sistema gestiona les diferents opcions d'entrada/sortida, els multithreads, la comunicació amb la GUI.. El sistema ha demostrat ser robust i fiable i també molt eficient. Aquest aspecte el comentaré a l'apartat §2.5.1.

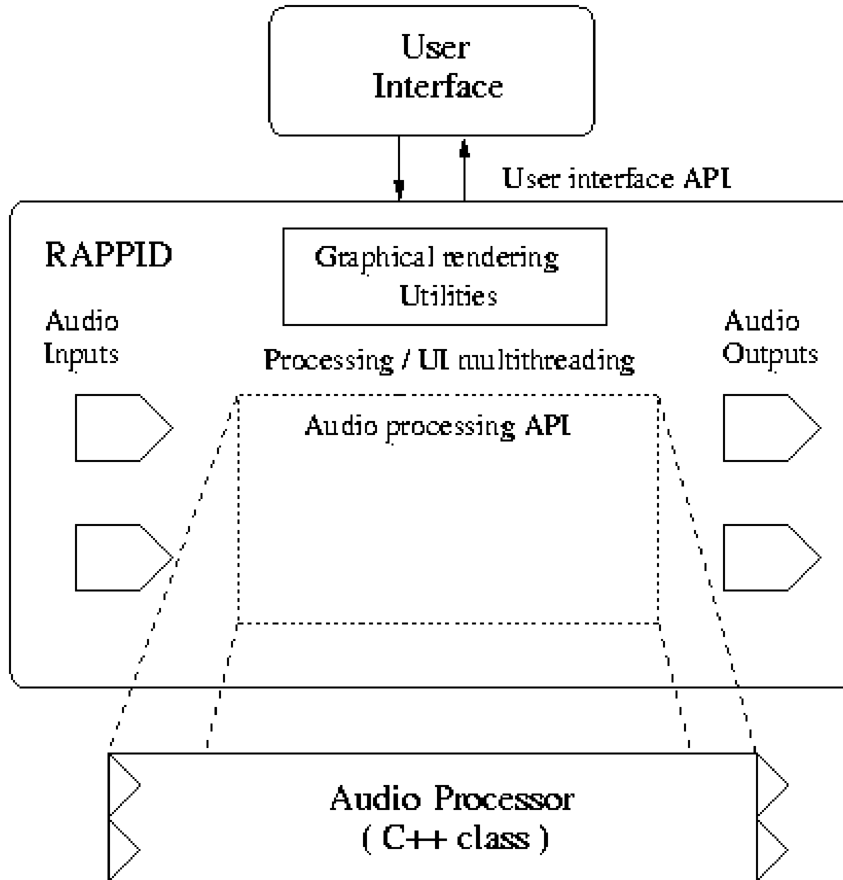

Figura 19. Esquelet d'aplicació per sistemes fiables, robustots i a temps real

.

2.4 Estudi econòmic

El projecte va començar l'Abril del 2001, al mateix temps que es va definir l'equip que desenvolupa CLAM. Des de llavors he estat treballant un mínim de 32 hores setmanals, tot i que, evidentment no totes aquestes hores eren dedicades a quelcom relacionat amb el projecte.

Al llarg d'aquest temps ha calgut realitzar tasques que no estan relacionades de forma directa amb aquest projecte: suport a usuaris, manteniment dels sistemes, resolució d'incidències...

També abans de poder fer alguns desenvolupaments centrats en el projecte ha calgut implementar algunes estructures de dades comunes. Per això, el càlcul del temps de la següent taula ha estat estimat respecte el temps total.

Activitat	Temps (hores)
Recerca	600
Disseny	400
Coordinació	180
Codificació	400
Documentació	200
Memòria	340
Total	2120

2.4.2 Comentari sobre la planificació

Al ser el projecte CLAM, a mig termini, i sobretot al no tenir uns requeriments concrets i complerts, la planificació inicial ha estat molt general, i de fet, molt equivocada[1].

Però un cop finalitzat el projecte ho veig molt natural, ja que era impossible preveure les dificultats d'uns desenvolupaments els quals no en coneixiem uns requeriments concrets.

Al fer un desenvolupament del software en espiral, ha estat a cada nova iteració que he pogut planificar l'etapa que em disposava a obrir. I cal dir que aquestes sub-planificacions sovint també eren errònes ja que la naturalesa del problema obligava fer una fase d'investigació abans de defininir totalment els requeriments concrets del desenvolupament de l'etapa.

Per això desisteixo de presentar en forma de diagrama de Gantt, tant la planificació inicial com la planificació real. En part és degut per aquest garbuix de planificacions per cada etapa (molt diferents les inicials que les reals) però també hi té a veure el fet que, precisament, no pogués cumplir amb la planificació d'aquest darrer mes (dedicada a la memòria, evidentment) a l'haver-me d'absentar dues setmanes.

[1] Per exemple, no preveia que el flow control pogués ser una tant complexe, i calgués tenir tants altres elements prèvis: l'havia planificat per tenir la primera versió pel Gener del 2002 !

2.5 Conclusions i línies futures

2.5.1 Conclusions respecte CLAM i el treball realitzat

Els objectius d'usabilitat, flexibilitat, fiabilitat i eficiència es poden considerar acomplerts en bona mesura, desde la última release interna de la llibreria 0.3 (del mes de Maig) i la posterior adaptació de les aplicacions disponibles a aquesta *release*.

Dic en bona mesura perquè, apart de l'eficiència (que tractaré al proper subapartat), la resta de propietats són d'avaluació molt subjectives. Estic segur que amb la futura la introducció del *flow control* automàtic, els sistemes que usin CLAM guanyaran en fiabilitat i usabilitat, tot mantenint la flexibilitat i eficiència (depenent del tipus de processament, l'eficiència podria reduir-se una mica).

De totes maneres, em puc basar en la pròpia experiència en fer una aplicació d'exemple (SpectralDelay §2.3.1) i en el feedback dels usuaris: s'ha aconseguit que la complexitat estigui ben oculta dins les classes de la llibreria (sobretot a les classes abstractes: `DynamicType`, `Processing`, `Application`...) de manera que les aplicacions que s'han construït fins ara, són molt fàcils d'entendre i de modificar.

Dit d'una altra manera, el codi d'usuari és molt proper a l'esquema mental que aquest té del sistema, evitant detalls superflus, i alhora, dotant de flexibilitat suficient a certs punts. Un exemple d'això són les configuracions per defecte, que permeten no explicitar la configuració d'un objecte de processament, però sempre que es necessita podem modificar-ne els paràmetres.

Valoro molt positivament el fet de desenvolupar una eina que ja té uns usuaris. Si bé implica dedicar bastant de temps al suport, també dóna moltes idees de quins aspectes van mal encaminats, noves necessitats, etc.

Cal tenir en compte que usem la metodologia de desenvolupament en espiral, i per tant, sempre hi ha temes que tenen el disseny obert. És aquí, sobretot, on s'agraeix les aportacions dels usuaris, sobretot si aquests són experts en el domini.

El suport automàtic a l'activació i passivació de les classes de CLAM en XML (en especial els *dynamic type*), s'ha mostrat encara més útil del que en un principi haviem previst, ja que s'usa en contextes diferents (processament, debugging, configuracions...) i no té cap cost per l'usuari.

En quant a la completesa (mòduls de GUI, I/O...) s'ha avançat molt en la última release, tot i que encara falta molta feina a fer, començant per fer que funcioni el que ja tenim, en totes les plataformes, continuant en implementar l'arquitectura de *pluggins* estàndard...

En quant a les valoracions negatives, citaré la dificultat i el temps que es perd en mantenir l'entorn de compilació (usar diferents llibreries...) i mantenir el mateix codi compilable en diferents plataformes.

Especialment negatiu és el temps perdut degut a les desviacions del C++ estàndard que implementen tots compiladors (en diferent mesura, això sí).

El temps de compilació de les aplicacions que usen la llibreria és inacceptable, degut a l'intens ús de templates i de la STL. Actualment estem solventant aquest problema.

Valoració de l'eficiència

Per valorar l'eficiència, a falta d'unes comparacions serioses entre CLAM i altres productes, em basaré amb l'experiència que ens ha donat el projecte *Rappid* (veure §2.3.3)

Es tracta d'una aplicació per tractar l'audio en temps real i està orientat ser molt robust i re-usable. Concretament, s'analitzaven dues entrades d'audio se n'extreien envolvents de l'energia i es realitzava una sèria de modulacions entre les dues entrades, i el resultat sortia per la tarja de so. S'utilitzava *multithreading* i una petita GUI, i s'executava sobre Linux (amb el *real-time kernel patch*).

Amb el sistema compilat sense cap optimització (mode *debug*, sense usar mètodes *inline*...) es podia obtenir una latència entre l'audio d'entrada i el de sortida de només 12 milisegons. La qual cosa és força espectacular, ja que està a l'alçada de les unitats hardware de processament del senyal.

Dit d'una altra manera, aquesta latencia era la menor possible que permetia el sistema operatiu i el driver de la tarja, en el cas que només haguéssim implementat un *by-pass*. El fet d'introduir un processat de càrrega mitjana no va ocasionar cap *click* produït per *underruns,* mantenint aquesta latència mínima.

Per altra banda vam fer un *profiling* de Rappid, en el qual aquest executava un anàlisis-síntesis bàsic per convertir l'audio del domini temporal a l'espectral i al revés. Els resultats també van ser molt bons: un 40% del temps es dedicava a executar conversions entre codificacions de l'espectre, un altre 20% en executar la FFT i la IFFT juntes. Sumant tots els percentatges dels processats inevitables obteniem 98%, només sobrava un 2% que era el temps que el sistema es dedicava a gestió de memòria, controls, GUI, accessos a les dades.... I per tant mostrava que CLAM no introduia cap ineficiencia, en comparació a un processat del senyal adhoc.

De totes maneres, aquesta prova no és ni de bon troç completa. I estic segur que hi ha processaments a la llibreria que són susceptibles d'optimitzacions agresives.

2.5.2 Línies futures

En general, dissenyar i implementar el *mode supervisat*. Els passos a seguir són els següents:

- Acabar de dissenyar i implementar els nodes (veure apèndix B, per la feina en progrés), integrar-ho amb els objectes de processament i els seus ports i testejar-ho.
- El disseny dels nodes s'ha de fer tenint en ment el futur mòdul de *flow control* que serà el principal element del *mode supervisat*.

- Dissenyar i implementar les xarxes, que són els objectes de processament reusables però no compilats. Les xarxes estan molt lligades amb el mode aplicació de CLAM.

Es podria investigar més sobre una implementació usant templates dels *dynamic types* i així desfer-nos de les macros, tot obtenint un disseny més net. I potser noves possibilitats...

Dotar la llibreria amb una aplicació de *debugging*, que permeti controlar l'execució del sistema i fer-ne introspecció, poguent inspeccionar tots els elements involucrats, ja sigui a través de vistes o fent ús de la serialització en XML.

Seria molt desitjable tenir una manera per establir paràmetres generals a tot el sistema d'una forma senzilla, sense haver de modificar les configuracions per defecte. S'ha d'investigar en estructurar les configuracions (potser usant el patró observador: un canvi global notifica totes les configuracions que hi estan subscrites)

Usar més les tècniques de programació genèrica per reduir el nombre de classes de processament, tot reaprofitant més el codi.

Dissenyar un sistema flexible i eficient per fer càlculs estadístics en estructures dinàmiques (compostos de dynamic types) de descriptors. Caldria investigar les aproximacions que en fan els llenguatges funcionals.

Per suposat, reduir el temps de compilació de la llibreria.

Bibliografia

[1] The MathWorks Web site, http://www.mathworks.com. "products" section. User Manual in pdf.

[2] Serra, Xavier. *A system for sound analysis/transformation/synthesis based on deterministic plus stochastic decomposition*. Standford. CCRMA Department of music. Tesi doctoral, 1989.

[3] Stroustrup, Bjarne. *Why C++ is not only an object-oriented programming language*. In *OOPSLA'95 Proceedings* (1995)

[4] Gamma, E., Helm R., Johnson, R., and Vlissides, J. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[5] Goldberg, A., and Robson, D. Smalltalk-80: *The Language and Its Implementation*. Addison-Wesley, 1983.

[6] Baus, C., and Becker, T. *Custom iterators for the STL. In First Workshop on C++ Template Programming*, Erfurt, Germany (October 10, 2000)

[7] Portability hints: Microsoft Visual C++ 6.0 sp4.

http://www.boost.org/more/microsoft_vcpp.html.

[8] Concurrent Version System - the open standard for concurrent control. http://www.cvshome.org

[9] BSCW, Basic Support for Cooperative Work. http://bscw.gmd.de/

[10] Mantis. http://mantisbt.sourceforge.net/

[11] JavaDoc Tool Home Page. http://java.sun.com/j2se/javadoc/

[12] Boost.org - free peer-reviewed portable C++ source libraries http://www.boost.org

[13] Johnson, R. E. *Dynamic object model. Types creation in execution time*.

[14] Czarnecki, K., Eisenecker, U., and Steyaert, P. *Beyond objects: Generative programming*.

[15] Serra, X. and Smith, J. *Spectral Modeling Synthesis*. In Proceedings of International Computer Music Conference 1989.

[16] Buck, J. and Lee, E. *The token flow model. Data Flow Workshop*, Australia, May 1992

[17] Govindarajan, R. and Guang, R. *Rate-Optimal Schedule for Multi-Rate DSP Computations.* In *Journal ov FLSI Signal Processing*

[18] Siek, J., and Lumsdaine, A. *Concept checking: Binding parametric polymorphism in C++*. In *First Workshop on C++ Template Programming*, Germany 2000

Apèndix A. Manual d'usuari i desenvolupador.

Apèndix B. Nodes (treball en progrés)