Università degli Studi di Napoli
"Federico II"

Facoltà di Scienze Matematiche Fisiche e Naturali

Tesi di Laurea in Scienze Informatiche

Progetto e Sviluppo di un Framework orientato agli oggetti per la ricostruzione parziale di eventi nell'esperimento CMS al CERN

Relatore:

Dott. Luca Lista

Laureando:

Umberto Di Meglio
matr.: 408/386

Anno Accademico
2003 - 2004

Gli esseri umani
si dividono in due categorie:
i geni e quelli che dicono di esserlo...
io sono un genio
(A. Einstein)

Indice

Ringraziamenti

Con la presente tesi voglio ringraziare tutte quelle persone che, con incoraggiamenti e ammonizioni, hanno contribuito alla mia crescita e mi hanno spronato anche nei momenti più difficili.
Il percorso scelto non è stato sicuramente facile ma, con la mia caparbietà e con l'aiuto di persone veramente speciali sono riuscito ad arrivare a costruire tutto questo. Non si tratta, ovviamente, di un punto d'arrivo, bensì di un punto di partenza per mettere in pratica tutto quello che questa università e le persone incontrate mi hanno dato.
Un ringraziamento particolare va al mio relatore, dott. Luca Lista, che, con l'aiuto del dott. Pierluigi Paolucci, ha reso possibile l'avventura iniziata questa estate presso il CERN (Svizzera) e continuata presso la sezione dell'I.N.F.N. di Napoli. Di certo il loro supporto e la loro disponibilità sono stati di fondamentale importanza. Un ringraziamento va anche al dott. Francesco Fabozzi per la disponibilità e la gentilezza dimostrata.
Non posso di certo dimenticare gli zii e i cugini che mi sono stati vicini e mi hanno aiutato nel periodo trascorso in Svizzera insieme a tutte le persone fantastiche che ho avuto modo di conoscere e con cui ho potuto collaborare.
Colgo l'occasione per ringraziare di cuore il dott. Vincenzo Innocente (ricercatore del CERN) che, con la sua esperienza e con le sue conoscenze, mi ha guidato e spronato ad imparare nuovi concetti e ad adoperare strutture e modelli che durante il corso di studi ero sicuro che esistessero solo per uno scopo didattico.
Un ultimo ringraziamento, ma non per importanza, va senza dubbio alla mia famiglia che mi ha dato i mezzi e la forza per intraprendere questo cammino e a Roberta che ha saputo sopportarmi, incoraggiarmi e soprattutto ascoltarmi in quelle lunghe giornate trascorse ripetendo prima di un esame.
Come non menzionare, infine, tutti i miei amici, vecchi e nuovi che di sicuro sono le prime persone che hanno contribuito a farmi diventare la persona che

sono oggi.
Posso ritenermi soddisfatto dell'esperienza fatta e del tesoro culturale che ho messo da parte e che formerà le basi del mio futuro.

Grazie a tutti!!!

con affetto... ***Umberto***

Capitolo 1

Introduzione

Questo lavoro di tesi, svolto questa estate come studente del "Summer Student Program 2004[7]" presso il CERN di Ginevra e in collaborazione con l'I.N.F.N.[13] di Napoli, è rivolto all'analisi e allo sviluppo di un framework orientato agli oggetti scritto in C++ per la "ricostruzione parziale" di eventi, ossia la possibilità di poter ricostruire o elaborare le informazioni di una parte della collisione da analizzare. Ciò consente un notevole risparmio di risorse di calcolo per applicazioni che sono critiche in termini di velocità. Tra queste va menzionato il trigger di alto livello che è quella parte di elaborazione dell'evento che viene processata in tempo reale, e che prende la decisione di registrare o meno un evento di collisione. L'introduzione di un tempo morto nell'elaborazione delle informazioni di trigger causerebbe una perdita degli eventi fisici.

Il CERN (European Organization for Nuclear Research) è uno dei più grandi laboratori di ricerca nel mondo. Fondato nel 1954 e situato al confine tra Francia e Svizzera è finanziato da venti nazioni europee e vede la collaborazione di migliaia scienziati provenienti da tutto il mondo.

Il principale progetto attuale del CERN è la realizzazione di un nuovo acceleratore, il Large Hadron Collider (LHC), dove fasci di protoni ad alta intensità verranno fatti collidere con un valore di energia mai raggiunto prima, pari a 14 TeV[1] nel centro di massa. Queste condizioni estreme offriranno nuove possibilità di ricerca, quali la verifica di fenomeni previsti teoricamente ma non ancora dimostrati sperimentalmente e la scoperta di nuovi costituenti della materia. L'esperimento CMS (Compact Muon Solenoid) è uno dei

[1] 10^{12} electron-Volt

quattro esperimenti che si svolgeranno presso l'acceleratore LHC, che è previsto iniziare la presa dati alla fine del 2007.
L'esperimento raccoglie i dati ad una frequenza di circa 150Hz, per una memoria occupata di circa 1MByte per *evento* di collisione e corrispondenti a circa 1000 TByte l'anno. La quantità di dati raccolta, nonché la complessità delle molteplici analisi dei dati che sono in programma, richiede al software, che sarà utilizzato per accedere ed analizzare i dati, notevoli prestazioni in termini di robustezza, affidabilità e flessibilità.
A questo scopo l'esperimento necessita di una struttura che permetta un uso flessibile dei dati e che si avvarrà di un complesso framework implementato tramite un linguaggio di programmazione di alto livello orientato agli oggetti (C++).
L'analisi dati parte dalle informazioni raccolte dai rivelatori dell'esperimento per ciascuna collisione in forma di segnali elettronici codificati ("raw data") che sono ricostruiti tramite complessi algoritmi di ricostruzione e riconoscimento delle particelle, le cui proprietà permettono di effettuare misure fisiche.

Mentre l'acceleratore (LHC) e l'esperimento (CMS) sono in fase di istallazione, già da tempo si lavora allo sviluppo del software di simulazione e ricostruzione. Tra gli istituti che collaborano al progetto CMS è presente anche la sezione di Napoli dell'Istituto Nazionale di Fisica Nucleare (I.N.F.N.), che in particolare si occupa della costruzione di alcune parti del rivelatore e dello sviluppo del software.

Capitolo 2

LHC e l'esperimento CMS

CMS (Compact Muon Solenoid) [1] è un esperimento di Fisica delle particelle elementari che opererà sull'acceleratore LHC (Large Hadron Collider)[2] in fase di costruzione presso il CERN (European Organization for Nuclear Research) [3] di Ginevra.
CMS è un rivelatore di grandi dimensioni realizzato per studiare le particelle (fotoni, elettroni, muoni, protoni, ecc.) generate dalle collisioni di protoni che avvengono all'interno dell'acceleratore LHC, identificandone la tipologia e misurandone i parametri (energia, impulso, direzione, ecc.). Lo scopo principale di questo studio è quello di studiare i costituenti fondamentali della materia e le loro interazioni fondamentali e ricercare nuove particelle.

2.1 L'acceleratore LHC

Prima di raggiungere l'acceleratore LHC, i fasci di protoni saranno accelerati sfruttando l'azione combinata di acceleratori già esistenti al CERN; in particolare, un acceleratore lineare, il ProtoSincrotone (PS) e un SuperProtoSincrotone (SPS). Dopo essere stati iniettati in LHC, i protoni saranno accelerati fino ad un'energia di circa 14TeV.
I fasci di protoni andranno a collidere lungo l'acceleratore nei quattro punti di intersezione in cui sono situati i quattro esperimenti (Figura 2.1): ALICE (A Large Hadron Collider) [2], LHCB (Large Hadron Collider Beauty experiment) [5], ATLAS (A Toroidal LHC Apparatus) e CMS (Compact Muon Solenoid) [6].

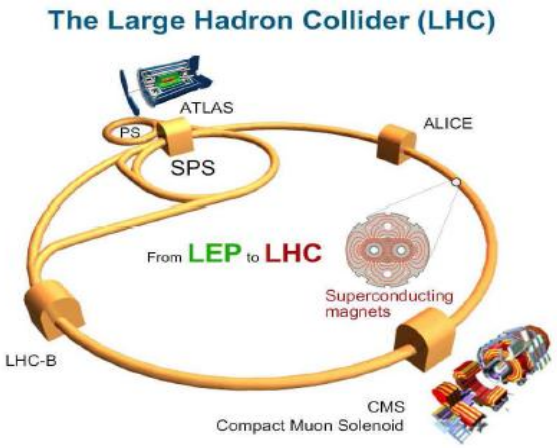

Figura 2.1: l'anello LHC e i suoi esperimenti

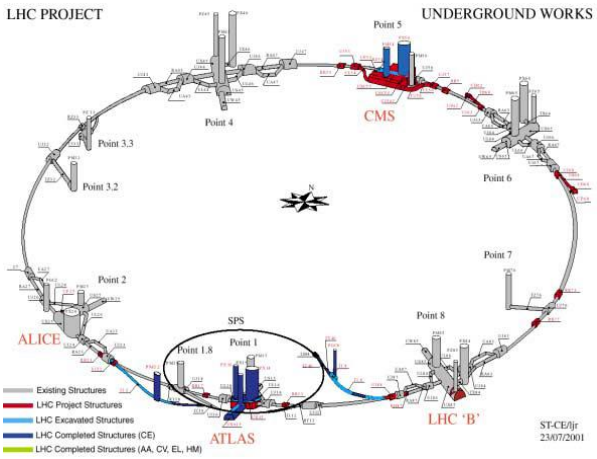

Figura 2.2: vista del progetto sotterraneo di LHC

L'incrocio tra fasci (bunch crossing) avverrà ogni 25 ns e comporterà per gli esperimenti un'elettronica di lettura veloce. In queste interazioni verranno prodotte centinaia di particelle di cui solo una parte avrà un interesse per studiare i processi fisici; la restante costituirà la parte di eventi di scarso interesse che dovrà essere scartata.

2.2 L'esperimento CMS

Per garantire prestazioni ottimali ai rilevatori è richiesta un'alta granuralità, ossia la suddivisione del rivelatore in un gran numero di elementi sensibili, allo scopo di ottenere un'efficiente ricostruzione delle tracce e un'elevata precisione per il riconoscimento della posizione del punto di passaggio di una particella. Ciò comporta anche un gran numero di canali di rivelazione, e quindi una grossa memoria occupata dalle informazioni di un singolo evento di collisione. Un'elettronica di lettura veloce, è indispensabile per evitare la sovrapposizione (pile up) dei segnali relativi a bunch crossing diversi. È anche necessaria infine una buona resistenza alle radiazioni affinché il rivelatore possa resistere ad un ambiente con elevate dosi di radioattività.

L'apparato CMS (Figura 2.3), lungo 22 m, con diametro di 14.6 m e del peso di 14500 tonnellate, si sviluppa secondo una simmetria approssimativamente cilindrica intorno al punto di collisione dei fasci ed è costituito da diversi rivelatori alloggiati l'uno all'interno dell'altro secondo uno schema a "cipolla". La parte del rivelatore parallela all'asse del fascio è detta "barrel" ed è costituita da superfici cilindriche, mentre quelle perpendicolari al fascio, "endcap" (tappi), sono costituite da superfici piane a forma di disco.
Dall'interno verso l'esterno, si distinguono:

- un sistema di tracciamento (Tracker) ad alta precisione che circonda il tubo intorno alla zona di interazione;
- un calorimetro elettromagnetico (E-CAL);
- un calorimetro Andronico (H-CAL);
- un magnete solenoidale superconduttore;
- un sistema di rivelazione dei muoni.

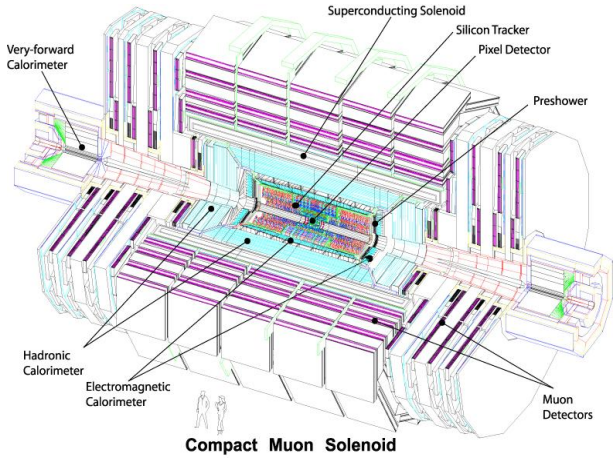

Figura 2.3: struttura dell'apparato di CMS

Il Tracker è il sistema di rivelazione più vicino al punto di interazione ed ha come scopi principali la ricostruzione delle tracce di particelle cariche e la ricostruzione dei vertici primari e secondari di decadimento; il Tracker contribuisce anche all'identificazione dei muoni ed elettroni insieme al sistema di rivelatori di muoni e ai calorimetri.
Il sistema calorimetrico identifica elettroni, fotoni, particelle adroniche, misurandone energia e posizione. Ha un tempo di risposta veloce e possiede un'elevata granularità allo scopo di minimizzare la sovrapposizione delle tracce. È costituito da un calorimetro elettromagnetico (ECAL) ed uno adronico (HCAL) che assorbono completamente le particelle che interagiscono in essi e ne misurano l'energia. Il calorimetro elettromagnetico è costituito da cristalli di tungstanato di piombo che offrono le migliori prestazioni per l'identificazione e la misura dell'energia di fotoni ed elettroni anche in condizioni di intenso campo magnetico ed elevate radiazioni e con tempi di risposta brevi (circa 10 ns). I calorimetri adronici sono realizzati alternando strati di materiale assorbitore in ottone a strati di scintillatori (4mm).

Sia il Tracker che il sistema calorimetrico sono alloggiati in un magnete costituito da un solenoide superconduttore lungo 13m con un diametro interno di 2.95 m che sarà capace di produrre un campo magnetico di 4 T. Il campo magnetico sarà parallelo al fascio di protoni e spingerà le particelle cariche su traiettorie elicoidali per consentire misure precise del loro momento.

Il rivelatore di muoni è posto esternamente ai calorimetri e alla bobina

del solenoide ed è costituito da quattro strati di rilevatori sia nel barrel che nell'endcaps che hanno una risoluzione spaziale di almeno 100 μm.

2.2.1 Il *Trigger* dell'esperimento

La grande quantità di eventi prodotti (circa 40 milioni al secondo) rende indispensabile un sistema di presa dati in grado di effettuare in tempo reale (*on-line*) una selezione degli gli eventi di interesse che necessitano di essere memorizzati in maniera permanente su memorie magnetiche e una reiezione degli eventi di scarso interesse. Il sistema in grado di selezionare gli eventi prende il nome di "*trigger*" (Figura 2.4). Questo presenta un'architettura a due livelli. Il primo livello di trigger è progettato in modo da selezionare

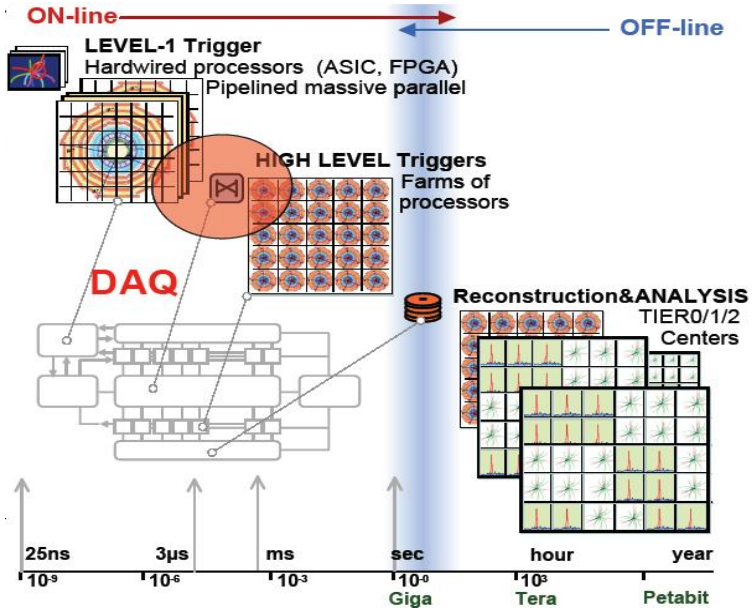

Figura 2.4: i diversi ivelli di trigger

al massimo 10^5 eventi al secondo (100 KHz) di tasso di uscita. L'elevata frequenza di *bunch-crossing* rende impossibile prendere una decisione tra un un incrocio dei fasci e il successivo. Per questo motivo il sistema di acquisizione deve essere in grado di memorizzare temporaneamente i dati in una coda (pipeline) in attesa delle decisioni provenienti dal trigger di primo livello. L'attesa è detta latenza ed è fissata a 3.2 μm, pari a 120 bunch-crossing.

L'elaborazione dei dati è completamente hardware, ovvero gli algoritmi impiegati per la selezione degli eventi sono realizzati mediante reti logiche integrate sui componenti del sistema.
Il secondo livello di trigger, detto di alto livello (HLT) riduce il tasso di 100 KHz generato dal trigger di primo livello ad una frequenza di circa 150 Hz. Il sistema di lettura (*read-out*) può memorizzare gli eventi in uscita dal primo livello di trigger, mentre quello di alto livello (HLT) decide se memorizzare gli eventi in maniera permanente sul disco o rigettarli. Il trigger di alto livello è realizzato tramite calcolatori programmati con il software dell'esperimento. Il software di Trigger verrà istallato in una farm di calcolatori con migliaia di nodi per garantire un tempo di processamento inferiore a 10^{-2}s. Il framework del software che sarà utilizzato per il Trigger di alto livello è lo stesso attualmente in fase di sviluppo per il processamento *off-line* degli eventi. Il lavoro di tesi, incentrato su estensioni del framework, ha, tra le applicazioni possibili, l'utilizzo nel Trigger di alto livello per consentire una riduzione dei tempi di esecuzione.

Capitolo 3

Il software dell'esperimento CMS

Gli esperimenti di fisica delle alte energie posti su LHC fanno largo uso di strumenti di ingegneria del software e dei linguaggi orientati agli oggetti per soddisfare le nuove esigenze che, sotto diversi aspetti, renderebbero problematico l'uso di soluzioni adottate nella precedente generazione di esperimenti come in particolare l'uso della programmazione procedurale e l'uso del linguaggio Fortran77.
L'esperimento CMS richiede, sia per la complessità dei rilevatori che per la lunga durata, un uso di strumenti Hardware e Software costantemente aggiornati e una strategia flessibile in grado di rendere possibili tutti gli aggiornamenti che con il passare del tempo si renderanno necessari. Per queste ragioni si ha la necessità di un'architettura ben definita ed omogenea, di applicazioni ripartite in entità gestibili indipendentemente (componenti/moduli); di interfacce e protocolli definiti in modo univoco allo scopo di implementare una efficiente comunicazione tra le parti operanti. Il linguaggio di programmazione orientato agli oggetti adottato in CMS per l'implementazione del software è il C++[9].

Il funzionamento del software dell'esperimento può essere riassunto, partendo dall'acquisizione dei dati prodotti in seguito alle collisioni, fino all'analisi fisica come descritto di seguito. I fasci di protoni vengono fatti collidere e producono particelle secondarie che interagiscono con i sottorilevatori dell'apparato CMS. Tutte le informazioni provenienti dai sottorilevatori vengono raccolte dall'elettronica di lettura e scritte su disco se il sistema di Trigger da un segnale positivo. Le informazioni memorizzate su disco in fase di ac-

quisizione sono identificate con il nome di "Raw Data" e vengono utilizzate per "ricostruire" gli eventi di interesse adoperando appositi algoritmi di ricostruzione. I prodotti della ricostruzione sono organizzati in oggetti secondo un opportuno modello; gli oggetti rappresentano la base di partenza per l'analisi fisica dei dati che ha come scopo di effettuare misure e osservazione dei fenomeni di interesse.

3.1 Architettura del software CMS

Attualmente è in fase di sviluppo un framework orientato agli oggetti che verrà usato per costruire applicazioni per processare dati reali o simulati che possono avere diverse applicazioni, che vanno dal trigger di alto livello all'analisi. Nel Framework potranno essere incorporati diversi moduli di software scritti dai fisici. La persistenza[1] degli oggetti è implementata usando un software sviluppato al CERN detto POOL[11], che può essere interfacciato con un sistema di memoria di massa con alte prestazioni, ad esempio HPSS[12], per una memorizzazione su disco e nastro.

3.2 Design dell'architettura

Il disegno generale dell'architettura software di CMS è motivato da requisiti quali: *ambienti multipli*, *sviluppo distribuito del codice*, *flessibilità* e *facilità d'uso*. I moduli software, infatti, devono essere in grado di operare in una varietà di ambienti come la fase di acquisizione dati, di trigger, di ricostruzione e di analisi finale. Il software, inoltre, deve essere flessibile, ossia facilmente adattabile alle diverse esigenze che si incontreranno durante l'esperimento, poiché non tutte le richieste del software saranno conosciute a priori. Le tecnologie hardware e software, infatti, cambieranno durante la durata dell'esperimento e una migrazione ad una nuova tecnologia potrebbe richiedere solo uno sforzo localizzato ad una piccola porzione del sistema. L'intero sistema software, infine, deve essere facile da usare da parte dei collaboratori fisici che non sono esperti di programmazione e che non possono dedicare molto tempo ad imparare complesse tecniche di programmazione e ai dettagli implementativi del software.

[1] memorizzazione permanente su disco o altro suporto

Questi requisiti implicano che il software debba essere sviluppato tenendo ben presente non solo le *performance* in termini di velocità di esecuzione, ma anche la modularità, la flessibilità, la facilità di manutenzione, la qualità e la documentazione.
Tutti i requisiti dell'architettura software per il progetto CMS portano alla seguente struttura generale:

- **un framework**: CARF (CMS Analysis & Reconstruction Framework);
- **moduli di software fisico**: moduli definiti con una chiara interfaccia che possono essere aggiunti al framework;

Il framework[8] definisce le astrazioni di livello più alto, il loro comportamento e i modelli (*pattern*) di collaborazione.
I moduli fisici sono scritti dai fisici e dai gruppi di sviluppo dei rilevatori. I moduli possono essere posti all'interno di un'applicazione del framework a run-time (a tempo di esecuzione), indipendentemente dall'ambiente di sviluppo. I moduli fisici non comunicano direttamente gli uni con gli altri ma solo attraverso i protocolli di accesso ai dati che sono parte del framework stesso.

3.2.1 Il Framework in generale

Uno dei concetti a cui lo sviluppo orientato agli oggetti tende è sicuramente quella di riuso: riuso del codice grazie all'ereditarietà, riuso dei design grazie ai pattern[8]. Un framework è costituito da un insieme di classi riusabili nel quale viene specificato il modo in cui le loro istanze interagiscono. È, inoltre, lo scheletro di un'applicazione che può essere personalizzato da uno sviluppatore. Un framework rende possibile ottenere sia riuso del codice che di design. È rappresentato da un'insieme di classi astratte e da come queste interagiscono. Quest'ultimo aspetto implica la definizione del framework come modello collaborativo riferito alle componenti del framework stesso. Lo sviluppatore non scrive codice per coordinare le componenti, bensì determina le componenti che, aderendo alla logica collaborativa del framework, verranno coordinate da quest'ultimo. In generale i framework assumono il controllo della applicazione e non il contrario.

3.3 Il processo di Ricostruzione

La ricostruzione rappresenta una delle applicazioni fondamentali del framework. Per il momento viene effettuata e testata su dati simulati nell'attesa che l'esperimento abbia inizio.
La fase di ricostruzione provvede alla creazione, a partire dalle informazioni fornite dai sottorilevatori, di oggetti che sono utilizzabili per l'analisi fisica.

Per ogni collezione di oggetti che si tenta di ricostruire esiste spesso più di un algoritmo che interviene nel processo di ricostruzione. La possibilità di una ricostruzione combinata equivale alla necessità di dover eseguire consecutivamente una catena di moduli di ricostruzione. Le sequenze di moduli della ricostruzione possono diventare estremamente complesse in fase di presa dati. Un possibile approccio potrebbe essere quello di specificare la sequenza di moduli di ricostruzione da eseguire esplicitamente. Un approccio del genere implica una serie di problematiche come l'ordine di esecuzione dei moduli e le difficoltà di gestione di sequenze molto complesse. Queste problematiche hanno portato alla scelta di un metodo di ricostruzione differente, detto ricostruzione "*on demand*" (vedi sez. 3.3.1).

Il processo di ricostruzione avviene in maniera implicita (vedi sez. 3.3.2) in cui l'utente dovrà solo conoscere la collezione di oggetti che intende analizzare, innescando una sequenza di richieste fino ad arrivare ai dati digitalizzati, i Raw Data. In tal modo solo gli oggetti che prenderanno effettivamente parte alla ricostruzione saranno ricostruiti.

3.3.1 L'azione on-demand

Negli anni passati molti studi sono stati effettuati per valutare l'architettura di esperimenti usata nelle simulazioni, nei programmi per la ricostruzione e quelli per l'analisi. Questi studi hanno portato al convincimento che la tradizionale architettura composta da un programma principale e da una serie di "subroutines" (sotto-programmi) e anche una più elaborata architettura "pipelined" (catena di montaggio) risultano non abbastanza flessibili per soddisfare le richieste dell'esperimento CMS.
Il framework CARF (CMS Analysis & Reconstruction Framework)[14], per permettere una maggiore flessibilità, implementa una "architettura ad invocazione implicita". I moduli si registrano al momento della creazione e sono invocati solo quando necessario. In questo modo solo i moduli che realmente

sono necessari vengono caricati ed eseguiti (da qui il nome "ricostruzione su richiesta").

3.3.2 L'invocazione implicita

I moduli che cambiano stato dipendono da alcuni "eventi" esterni che possono registrarsi come elementi detti "observer"[8] all'interno di una struttura detta "dispatcher".

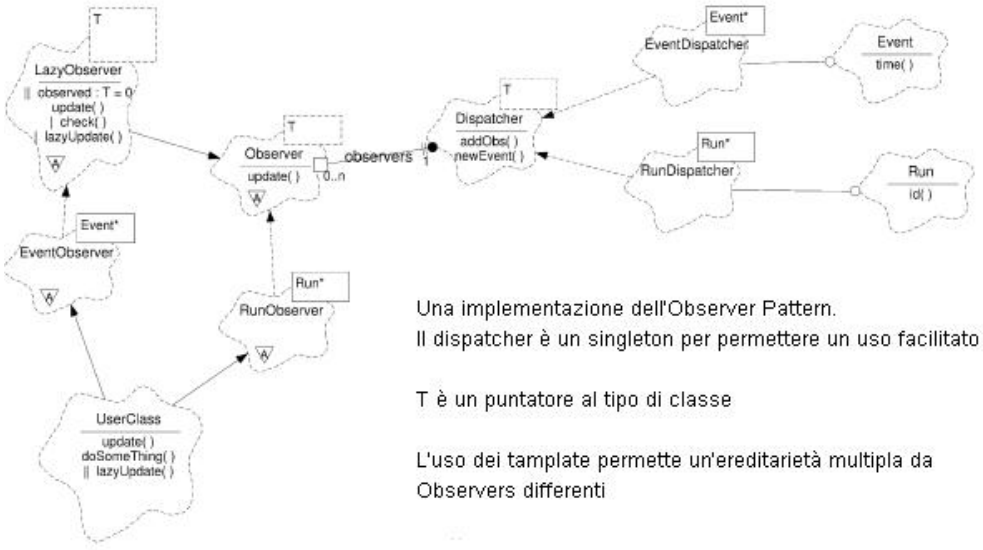

Figura 3.1: Il diagramma delle classi mostra le classi che collaborano nel meccanismo di invocazione implicita implementata in CARF

Un tipico "evento" esterno, nell'applicazione CARF, è l'inizio dell'elaborazione di nuovi eventi fisici o di nuovi eventi simulati. Quando è presente un nuovo evento il dispatcher informa tutti gli observers registrati di aggiornare il loro stato ed eventualmente di eseguire una specifica azione. Un'azione, per evitare di avere computazioni non usate, può essere "congelata" fino a quando viene richiesto ad un observer uno specifico servizio. Questo meccanismo di "registrazione" e "notifica" fa uso del "Observer Pattern"[8] e in CARF è implementato usando classi parametriche (template[2]) che tengono traccia

[2]una classe templata è una classe parametrica che rende possibile l'utilizzo della stessa per tipi differenti di oggetti

dell'evento da inviare all'analisi. Un esempio del diagramma delle classi (Figura 3.1) mostra l'uso della classe che "osserva" gli eventi fisici e l'esecuzione degli stessi. In questo esempio la classe utente è un "observer standard" che attuerà un'azione immediata quando cambia un evento in esecuzione e un "lazy observer" (letteralmente un observer pigro) dell'evento che attuerà un'azione solo quando un utente chiederà un servizio al framework.

3.3.3 Le fasi della Ricostruzione

Il processo di ricostruzione è visto come una collezione di elementi indipendenti l'uno dall'altro, ognuno dei quali produce un corrispondente insieme di oggetti ricostruiti (***RecObj***). Prima di arrivare agli oggetti ricostruiti, i dati registrati dai rilevatori detti *Hits*[3], vengno memorizzati per poi passare alla fase di digitalizzazione. Questa fase tiene conto del modo in cui viene acquisita l'informazione e produce i cosidetti "*DIGI*" o "*Raw Data*". Ottenuti i *DIGI* si passa alla fase di ricostruzione successiva. Ad esempio, nel rivelatore di tracciamento i Raw Data sono costituiti da una codifica degli indirizzi delle celle che hanno riportato un segnale. Un primo modulo del framework decodifica le informazioni dei Raw Data; un secondo modulo raggruppa in un singolo Hit i segnali di celle adiacenti; un ulteriore modulo, infine, associa tra di loro gli Hit prodotti dalla stessa particella per la quale viene costruito un oggetto "traccia" che conterrà, oltre alla collezione degli Hits che lo costituiscono, anche delle stime dei parametri cinematici (momento, direzione). Questa catena di elaborazione viene invocata automaticamente appena un modulo di analisi richiede l'accesso alla collezione di tracce.

3.3.4 I moduli del Framework

Il software CMS è suddiviso in pacchetti (packages). Ciascun pacchetto software compilato, produce una libreria condivisa. CARF supporta completamente un caricamento dinamico a tempo di esecuzione (*run-time*) di queste librerie condivise e rende possibile personalizzare i criteri e parametri della ricostruzione. Questi potranno essere specificati perché l'utente può sovrascrivere le impostazioni di default e attribuire ai propri moduli la configurazione preferita. Quando si accede a degli oggetti da un database viene innescato il

[3]Un Hit rappresenta il segnale identificato su di una cella del rilevatore; l'insieme degli Hit costituisce i Raw Data

caricamento dinamico dei pacchetti richiesti per utilizzarli. Un'applicazione CARF può iniziare con un insieme minimo di software caricato: il suo software di base verrà espanso con ulteriori funzionalità durante l'accesso agli oggetti richiesti dall'utente e contenuti nel database. Nello stesso tempo il software non richiesto non verrà caricato, mantenendo, in questo modo, lo spazio utilizzato dell'applicazione ragionevolmente "piccolo".

La parte del framework che permette di utilizzare e interfacciare il software per la ricostruzione è una componente fondamentale delle librerie condivise di CARF che prende il nome di "Reco library" in cui vengono forniti tutti gli strumenti necessari per includere e configurare i vari algoritmi e registrare i dati ricostruiti in una memoria di massa. Questa libreria sarà discussa più in dettaglio nel prossimo capitolo.

Capitolo 4

La libreria Reco e la "*Ricostruzione Parziale*"

4.1 Introduzione

L'analisi fisica è il passo finale della catena di processamento degli eventi e si basa su dati ricostruiti a partire dalle informazioni fornite dai sottorilevatori. La fase di ricostruzione rappresenta la parte più delicata sia in termini di efficienza che di affidabilità per le applicazioni, in particolare per quelle più critiche come il trigger di alto livello.

Lo scopo di questa tesi è quello di rendere possibile la "*ricostruzione parziale*", ossia la possibilità di poter ottenere una ricostruzione di una parte di un evento seguendo i criteri di ricostruzione on-demand e di invocazione implicita richiesti dall'esperimento. In particolare, è stato realizzato un prototipo della libreria Reco in grado di offrire questo tipo ricostruzione e un test applicato ad un prototipo di *Analysis Object Data* (AOD)[15] sviluppato presso la Sezione dell'I.N.F.N. di Napoli. Il lavoro di tesi è iniziato partendo da un prototipo preesistente della libreria Reco ed è stato articolato effettuando:

- analisi dell'architettura esistente;
- revisione di tale architettura per permettere la ricostruzione regionale;
- implementazione e aggiornamento delle classi individate nella fase di revisione;

- test del prototipo.

4.2 Analisi del Framework preesistente

Per analizzare le caratteristiche della libreria Reco, sarà necessario capire il funzionamento della struttura per evidenziarne i difetti e proporre i cambiamenti necessari.

4.2.1 Le principali classi

Le principali classi utilizzate nella libreria Reco sono:

- **RecObj**: una classe astratta[1] che necessita di essere specializzata negli oggetti che rappresentano i dati ricostruiti per l'analisi fisica (tracce, muoni, elettroni, ecc.);
- **RecUnit**: è la classe astratta di base per gli algoritmi di ricostruzione (uno per ogni RecUnit).
- **Reconstructor**: è la classe astratta che gestisce la collezione ricostruita. Le sottoclassi derivanti implementano in diversi modi il meccanismo di ricostruzione on-demand che viene innescato, nel caso in cui non sia presente la collezione richiesta, delegandone il compito alla RecUnit a cui associato;
- **RecCollection**: rappresenta le collezioni di oggetti ricostruiti. È una classe parametrica che viene fornita dal Reconstructor e la cui richiesta innesca il meccanismo di "Ricostruzione *On-Demand*"(sez 3.3.1)

Queste classi rappresentano il più alto livello di astrazione.

Ogni classe definita dall'utente in cui vengono definiti degli algoritmi da utilizzare durante la fase di ricostruzione deve necessariamente ereditare dalla classe RecUnit per potersi interfacciare al framework e ai servizio che questo mette a disposizione. La *RecUnit*, inoltre, può usufruire delle collezioni di oggetti ricostruiti messe a disposizione dalla classe ***"RecCollection"*** e diverse combinazioni *RecUnit/RecCollection* possono fornire differenti versioni

[1] Una classe astratta è una classe generica che contiene metodi e attributi comuni a più classi le quali ne specializzeranno le caratteristiche ereditando dalla classe astratta

dello stesso tipo ***"RecObj"***. La comunicazione tra RecCollection e RecUnit avviene tramite la classe ***Reconstructor***. Questa classe comunica con la RecCollection implementando un modello detto *Visitor Pattern*[2][8] e implementa, insieme alla RecUnit, un modello detto *Strategy Pattern*[3][8].
Per capire meglio il funzionamento e l'interazione delle classi principali del framework, suppuniamo (Figura 4.1) che venga richiesta una collezione di

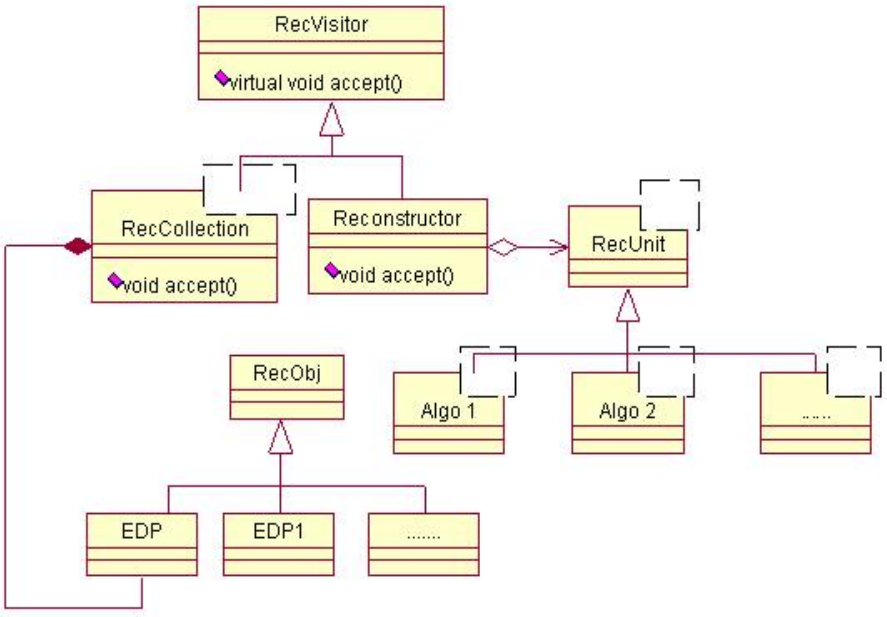

Figura 4.1: un esempio delle classi principali del framework

oggetti di un ipotetico tipo *EDP* (Event Data Product), al momento della richiesta verrà controllato che la collezione non sia stata ancora ricostruita; se così non è, la libreria Reco la ricostruisce utilizzando la *RecUnit* che implementa l'algoritmo di ricostruzione di tipo *EDP* ("$RecUnit^{EDP}$"). Nel caso

[2]il *Visitor Pattern* permette di definire un'operazione da eseguire sugli elementi di una struttura ad oggetti. Tale rende possibile ridefinire questa operazione senza cambiare le classi di elementi in cui opera

[3]lo *Strategy Pattern* permette di definire una famiglia di algoritmi, incapsularne ognuno in una classe e renderli intercambiabili. Questo pattern permette agli algoritmi di variare in maniera indipendente dal client che li utilizza. Nel nostro caso il *client* è la classe Reconstructor e lo *strategy*, ovvero la classe che incapsula gli algoritmi è la RecUnit

in cui questo algoritmo abbia bisogno di altre collezioni verranno invocate a catena le RecUnit corrispondenti alle collezioni richieste.

Le misure fisiche che possono essere effettuate sugli oggetti ricostruiti sono molteplici e variano in base al tipo di analisi che si vuole effettuare. Un algoritmo di ricostruzione, infatti, può usare differenti calibrazioni, allineamenti o parametri. È possibile, quindi, che ci siano algoritmi con diverse configurazioni per ricostruire lo stesso tipo di oggetto. In questo modo si viene a creare un'ambiguità che viene risolta introducendo una chiave identificativa per ogni algoritmo utilizzato che può essere una semplice stringa o un oggetto più complesso (***RecQuery***).

Un algoritmo, come detto in precedenza, altro non è che una specializzazione della classe RecUnit e può essere configurato dall'utente o assumere una configurazione di default che lo rende differente da tutti gli altri. Una configurazione completa di una RecUnit è definita da:

- **Nome**: il nome della RecUnit;
- **Versione**: la versione dell'algoritmo corrispondente;
- **Parameter Set** (insieme di parametri): una classe che contiene l'insieme completo dei parametri della RecUnit che sono configurabili dall'utente;
- **Component Set** (insieme delle componenti): è un insieme formato da una coppia (nome, RecQuery) in cui vengono specificate le componenti che verranno utilizzate per la ricostruzione;
- **Calibration Set** (insieme di calibrazione): l'insieme in cui vengono dichiarati i parametri per la calibrazione;

L'intera configurazione è incapsulata all'interno di una classe chiamata RecConfig che viene definita al momento della creazione dell'algoritmo.

4.2.2 Limiti del Framework attuale

La fase di ricostruzione è un processo che, come abbiamo visto precedentemente, può richiedere molte dipendenze e che produce in output una collezione di oggetti ricostruiti derivanti dalla combinazione dalle informazioni contenute in altre collezioni. Analizzando il modo in cui comunicano le vari classi si possono individuare due contesti in cui una collezione può operare: un primo contesto che definiremo **locale** e un secondo che definiremo **globale**. Il primo rappresenta l'ambito in cui la collezione è definita, con i propri oggetti ricostruiti, i propri parametri di configurazione, le proprie componenti e, ovviamente, la propria RecUnit. Il secondo, invece, rappresenta l'ambito in cui sono definite tutte le collezioni disponibili. Una collezione, al momento della creazione, verrà identificata in maniera univoca e l'accesso alle informazioni in essa contenute avviene attraverso una ricerca tra tutte le collezioni disponibili.
Al momento della ricostruzione, se esiste una dipendenza, l'algoritmo chiederà alle RecUnit, registrate nel contesto globale, le collezioni di cui ha bisogno. Si consideri la Figura 4.2, in questo esempio per la ricostruzione del bosone di Higgs[18] che decade in una coppia ZZ sono richieste sette collezioni (escludendo quelle che vengono usate la ricostruzione dei muoni e degli elettroni) e, per ognuna di queste, si dovrà passare da un contesto locale, quello in cui si vuole ricostruire esattamente H$\longrightarrow$ZZ, ad un contesto globale in cui sono contenute le collezioni che contribuiscono alla ricostruzione come ad esempio H$\longrightarrow$eeee, H$\longrightarrow \mu\mu\mu\mu$, H$\longrightarrow \mu\mu$ee, ecc. Lo stesso avviene per B_s [17] che decade in J/$\psi\phi$ come mostrato in Figura 4.3 in cui sono presenti cinque collezioni escludendo quelle che servono per la costruzione di μ e di K.
Un algoritmo di ricostruzione che ha bisogno di altre collezioni, quindi, deve necessariamente passare da un contesto locale, in cui è definita la sua RecUnit, ad un contesto globale in cui sono contenute le collezioni di cui necessita. Le collezioni esterne forniranno gli oggetti ricostruiti che saranno usati come informazioni di partenza per la ricostruzione dell'algoritmo. Come si è potuto constatare negli esempi precedenti un processo di ricostruzione può generare molte dipendenze con le collezioni di cui ha bisogno e, le molte richieste al database, comportano un dispendio di tempo che risulta significativo ai fini dell'analisi. Questo dispendio di tempo risulta particolarmente rilevante per le applicazioni che operano in tempo reale come il Trigger di alto livello (vedi sez 2.2) che decide se registrare o rigettare un evento prodotto.
Migliorare questo aspetto significa evitare l'introduzione di un tempo morto.

Un ulteriore e più importante vantaggio per l'applicazione nel Trigger è la possibilità con un analogo meccanismo di definire degli ambiti parziali di ricostruzione che permettono, ove richiesto, di effettuare solo parzialmente la ricostruzione delle informazioni.

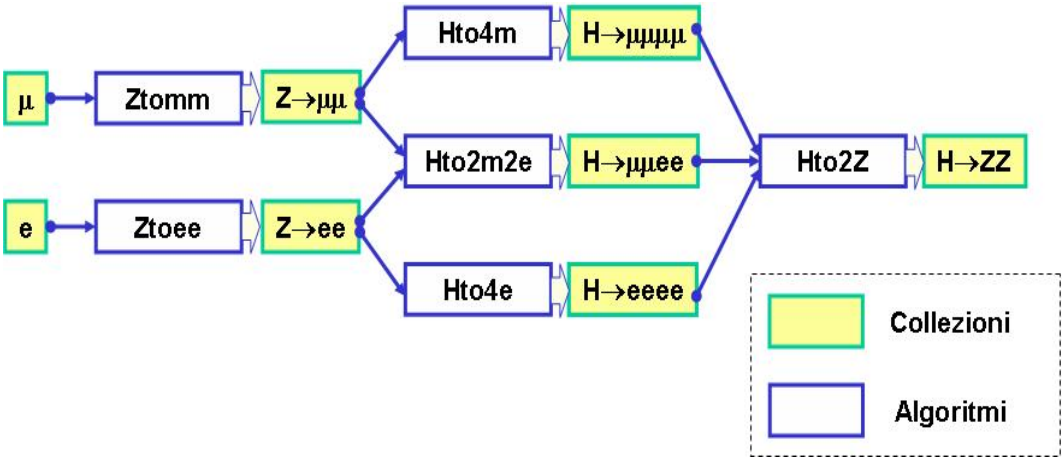

Figura 4.2: ricostruzione di H⟶ZZ

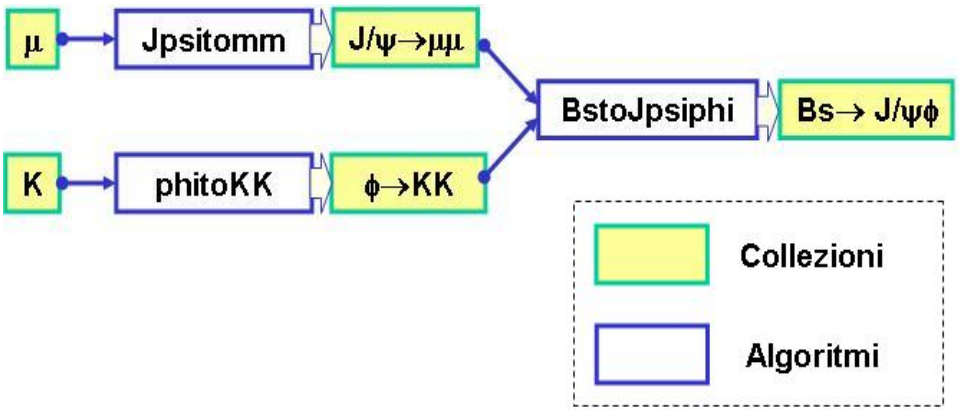

Figura 4.3: ricostruzione di $B_s \longrightarrow J/\psi\phi$

4.3 Estensione delle funzionalità del Framework

Prima di procedere ad analizzare le effettive funzionalità che devono essere incluse nell'estensione del framework è opportuno tener presente alcuni aspetti fondamentali.
Nel software del progetto CMS, infatti, sono presenti molte applicazioni che utilizzano le componenti del framework esistente, quindi mantenere lo standard è un vincolo da tener presente che può essere rispettato utilizzando, laddove sia possibile, la stessa interfaccia del framework e mantenedo inalterate le funzionalità delle applicazioni senza sconvolgerne la struttura.
L'obbiettivo principale è ottimizzare il meccanismo di ricostruzione della libreria Reco migliorandone le prestazioni introducendo il concetto di "*ricostruzione parziale*". A questo è importante capire cosa si intende per ricostruzione parziale e quali vantaggi comporta.
Si definisce ricostruzione *regionale* o *parziale* di un evento, la possibilità di ricostruire su richiesta (*on demand*) anche solo una parte delle collezioni che interagiscono durante un processo di ricostruzione di un evento di interesse. Con l'utilizzo di questo meccanismo si evita di interrograre un database per ogni collezione che fa parte del processo; si rende inoltre possibile l'analisi e la ricostruzione di una parte dell'evento senza doverlo necessariamente ricostruire per intero. Si consideri, ad esempio, la ricostruzione del bosone di Higgs visto nella sez. 4.2.2. La ricostruzione regionale permetterà di ricostruire su richiesta solo la collezione $Z \longrightarrow \mu\mu$ senza dover necessariamente ricostruire anche la collezione $H \longrightarrow ZZ$ che comunque potrà essere effettuata restando nello stesso contesto (locale) in cui è definita insieme alle altre collezioni che appartengono al processo. L'unico accesso al contesto globale si avrà solo per le collezioni di interesse generale come i dati ottenuti dai rilevatori che costituiranno le informazioni di partenza per la ricostruzione di diversi algoritmi.

4.3.1 Analisi e Design; considerazione di possibili Use-Case

Analizzando il framework attuale si è giunti alla conclusione che la *ricostruzione parziale*, per essere attuata, necessita di un meccanismo che contenga gli algoritmi utilizzati per la ricostruzione di un evento e che offra la possi-

bilità di poter richiedere la ricostruzione di una collezione intermedia[4].
Per poter costruire un meccanismo di questo tipo, dobbiamo focalizzare la nostra attenzione su come sia possibile implementare un modello che riesca a separare le componenti (gli algoritmi) dalle collezioni e che conosca lo stato di una collezione richiesta, ossia se è stata ricostruita o no.
Tra le diverse architetture software esistenti, un modello che ben si presta al nostro scopo è la ***Blackboard System Architecture***[16]. Prima di capire come possa essere incorporata nel contesto della libreria Reco, cerchiamo di capire cos'è e come funziona.

La Blackboard System Architecture in generale

La Blackboard System Architecture[16] fu sviluppata negli anni '70 per la necessità di trovare un sistema di collaborazione tra moduli per la risoluzione di problemi in larga scala.

Un approccio tradizionale per combinare un insieme di moduli software differenti è quello di connetterli seguendo il loro flusso di connessione (Figura 4.4(a)). I moduli possono apparire più volte nel diagramma della comunicazione, ma la connessione è predeterminata e diretta. Questo tipo di approccio può andare bene nel caso in cui sia i moduli che le connessioni sono statiche. Nel momento in cui i moduli sono soggetti a cambiamenti, oppure nel caso in cui l'ordine dei moduli non può essere determinato prima che si verifichi un dato evento a tempo di esecuzione (*run-time*), la non flessibilità del sistema diviene inaccettabile.

Un approccio differente consiste nell'usare una comunicazione tra moduli *indiretta*[5] e *anonima*[6] attraverso una sorta di contenitore dei dati: la blackboard (Figura 4.4(b)). Con questo approccio è possibile ottenere tutte le connessioni possibili tra i moduli e la scelta di una connessione può essere effettuata in maniera dinamica attraverso un meccanismo che controlli le scelte effettuate. Le informazioni messe nella Blackboard sono pubbliche[7] e l'utilizzo della comunicazione indiretta dei moduli riduce il numero di interfacce necessarie che devono essere supportate da moduli collaboranti.

---

[4] definiamo collezione intermedia una collezione che partecipa al processo di ricostruzione

[5] la comunicazione tra moduli avviene solo tramite Blackboard

[6] ogni modulo, non essendo a conoscenza degli altri moduli presenti nella Blackboard, si limita a chiedere un servizio alla Blackboard che lo smisterà al modulo adatto

[7] con pubblico si intende la visibilità dei moduli e dei dati disponibili

La Blackboard System Architecture consiste in tre componenti principali(Figura 4.5): Knowledge Sources, Blackboard e Control Shell.

Le Knowledge Sources (KSs) sono dei moduli di calcolo indipendenti che insieme contengono le informazioni necessarie per risolvere un problema.

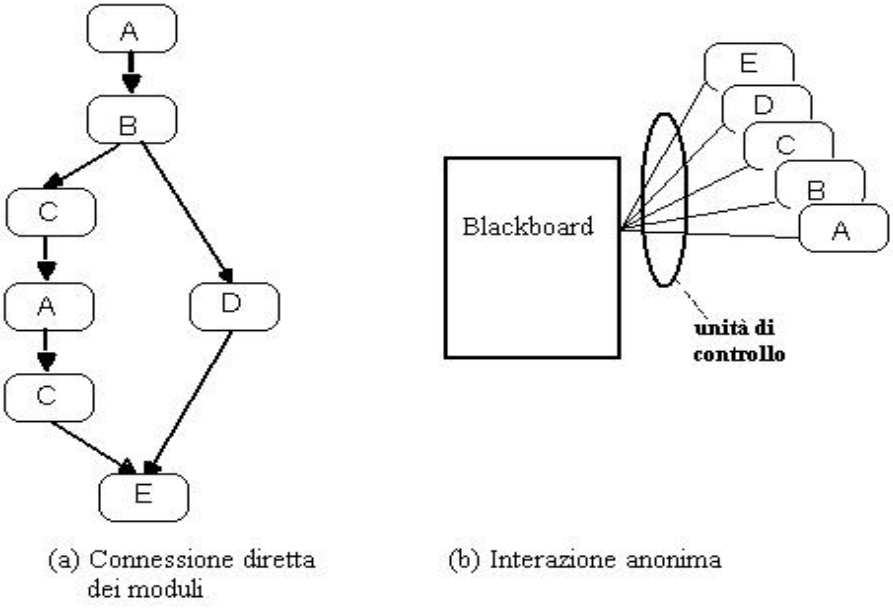

Figura 4.4: connessione dei moduli

Questi sono moduli anonimi, il che significa che non interagiscono direttamente con gli altri KSs e non sono a conoscenza di quali altri KSs sono presenti nel sistema.

La Blackboard è il cuore dell'omonima architettura, è un contenitore di: dati di input, soluzioni parziali e altri dati che si trovano in diversi stati della soluzione del problema. Questa componente fondamentale, inoltre, pubblica i risultati ottenuti dalle KSs in una struttura dati condivisa che è disponibile a tutte le componenti del sistema.

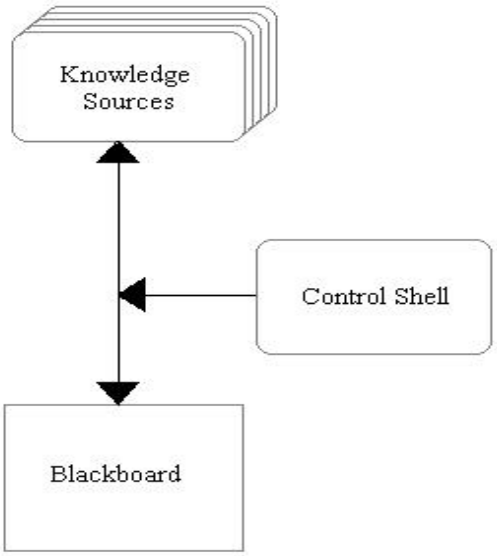

Figura 4.5: componenti di una Blackboard System

La Control Shell, infine, prende decisioni a run-time (tempo di esecuzione) durante il processo di risoluzione del problema e controlla che le richieste fatte siano coerenti con le funzionalità messe a disposizione dalla Blackboard.

Questo tipo di architettura, in conclusione, permette di creare un sistema:

- *flessibile*: perché l'aggiunta di una nuova KSs non comporta nessun cambiamento dell'architettura in quanto le KSs presenti non si conoscono tra di loro;

- *condiviso*: perché tutti i dati vengono pubblicati nella Blackboard e sono visibili a tutti moduli che partecipano alle risoluzione di un problema;

- *coerente*: perché grazie alla Control Shell è possibile garantire che le operazioni da effettuare siano coerenti con il problema da risolvere.

La fase di Design

Per poter effettuare una buona fase di design si inizierà a sviluppare il framework identificando di volta in volta le componenti della Blackboard System

che si creano con le principali classi coinvolte per poi ottenere la struttura completa.

Iniziamo la fase di design prendendo in considerazione la RecUnit attuale e le sue caratteristiche che possono essere utilizzate nel progetto del nuovo framework. La RecUnit attuale (vedi sez. 4.2.1) rappresenta la classe da cui ogni algoritmo di ricostruzione deve ereditare per poter essere utilizzato nel framework. Per ogni algoritmo utilizzato si crea un contesto *locale* in cui è definito l'algoritmo stesso e genera in output solo la collezione che tenta di ricostruire. Ovviamente siamo interessati a permettere la ricostruzione su richiesta anche di una delle collezioni che fanno parte del processo di ricostruzione e che possono essere diverse da quella finale. Per poter attuare questo tipo di ricostruzione si ha la necessità di cambiare la struttura della RecUnit facendola diventare una unità di ricostruzione capace di contenere più algoritmi (Figura 4.6). In questo modo si può richiedere, utilizzando una sola

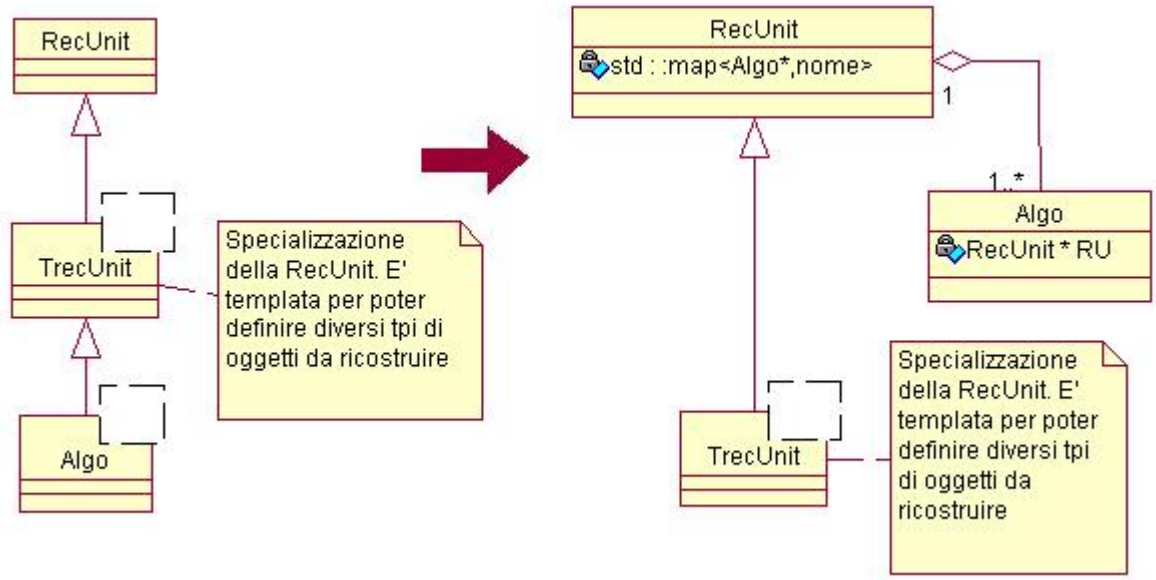

Figura 4.6: trasformazione della classe RecUnit

RecUnit, l'esecuzione di più algoritmi. Con questa nuova forma, ovviamente, c'è la necessità di utilizzare un'unica interfaccia per gli algoritmi. Introdurremo, quindi, una nuova classe che chiameremo ***SubRecAlgo***. Questa classe astratta rappresenterà l'interfaccia di base da utilizzare per ogni algoritmo

che farà parte della ricostruzione. Il design risultante della classe RecUnit sarà come in Figura 4.7.

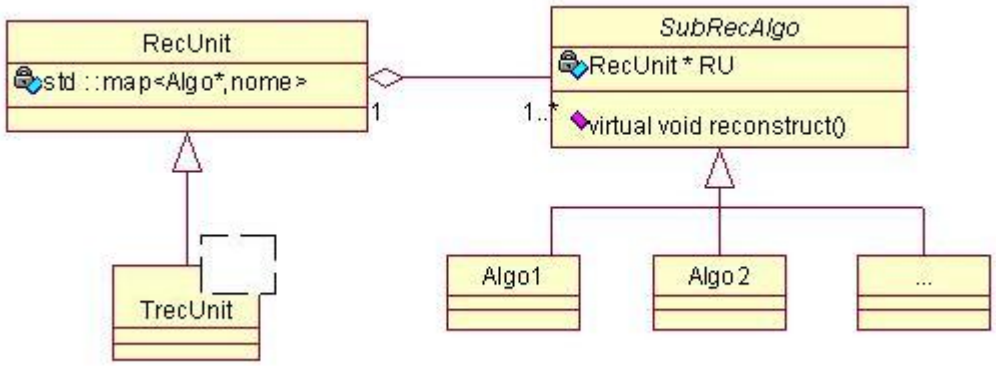

Figura 4.7: RecUnit risultante e inserimento della SubRecAlgo

In questa struttura possiamo individuare due delle tre componenti fondamentali della Blackboard System Architecture (vedi Figura 4.8): la Control Shell e le Knowledge Sources.
Le KSs ovviamente sono gli algoritmi che saranno utilizzati durante il processo di ricostruzione. Le classi che ereditano da SubRecAlgo non interagiscono direttamente, come per definizione delle KSs, ma solo tramite la Blackboard che troveremo fra poco proseguendo nel design.
La Control Shell, invece, è la RecUnit in quanto rappresenta effettivamente il meccanismo di controllo di esecuzione degli algoritmi. Il suo scopo, inoltre, è quello di far eseguire l'algoritmo corrispondente alla collezione da ricostruire richiesta.

A questo punto non ci resta che individuare la Blackboard vera e propria e cioè la struttura in cui pubblicare le collezioni ricostruite che potranno essere utilizzate dagli algoritmi registrati nella RecUnit. Il meccanismo di ricostruzione preesistente memorizza gli oggetti ricostruiti all'interno della classe Reconstructor.

Nel nuovo design del framework basterà generalizzare il numero di collezioni ricostruite basandosi sugli algoritmi che vengono eseguiti ad ogni richiesta. Sembra naturale, a questo punto, che la classe Reconstructor faccia parte sia della Blackboard che della Control Shell perché, oltre a contenere le collezioni ricostruite, potrà essere utilizzato per coordinare il processo di ricostruzione richiedendo l'esecuzione dell'algoritmo associato alla collezione non

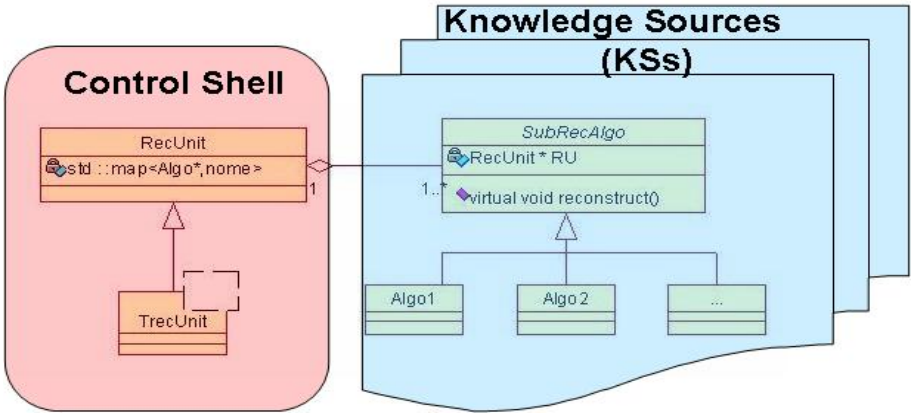

Figura 4.8: identificazione della Control Shell e delle KSs all'interno della nuova struttura

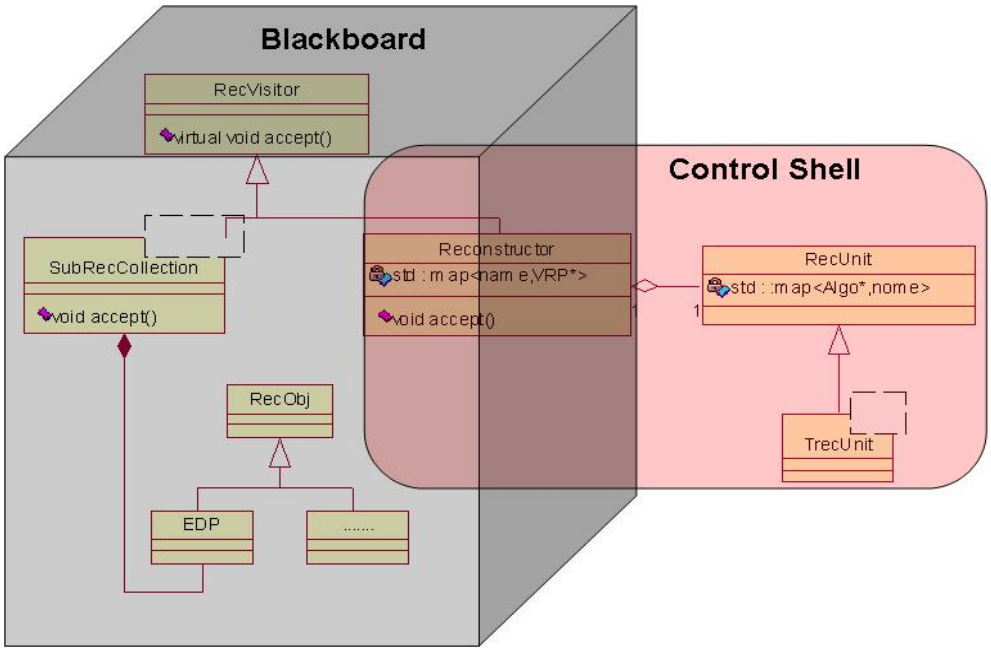

Figura 4.9: la blackboard e l'ampliamento della Control Shell

ancora ricostruita. Faranno parte della Blackboard, oltre alla classe Reconstructor, gli oggetti ricostruiti, ovvero le classi ereditanti dalla classe ***RecObj***, che sono le componenti delle collezioni e la classe ***SubRecCollection*** che permette di restituire una collezione ricostruita (vedi Figura 4.9). Non bisogna dimenticare, infatti, che le collezioni contenute nel Reconstructor sono accessibili da un programma utente solo attraverso quest'ultima classe che, ad ogni esplicita richiesta, restituisce la collezione ricostruita. La SubRecCollection è, inoltre, una classe parametrica dove il parametro rappresenta il tipo di oggetti di cui deve essere formata la collezione richiesta. Di seguito in Figura 4.10 mostreremo il diagramma UML[19] delle classi principali che fanno parte della nuova libreria.

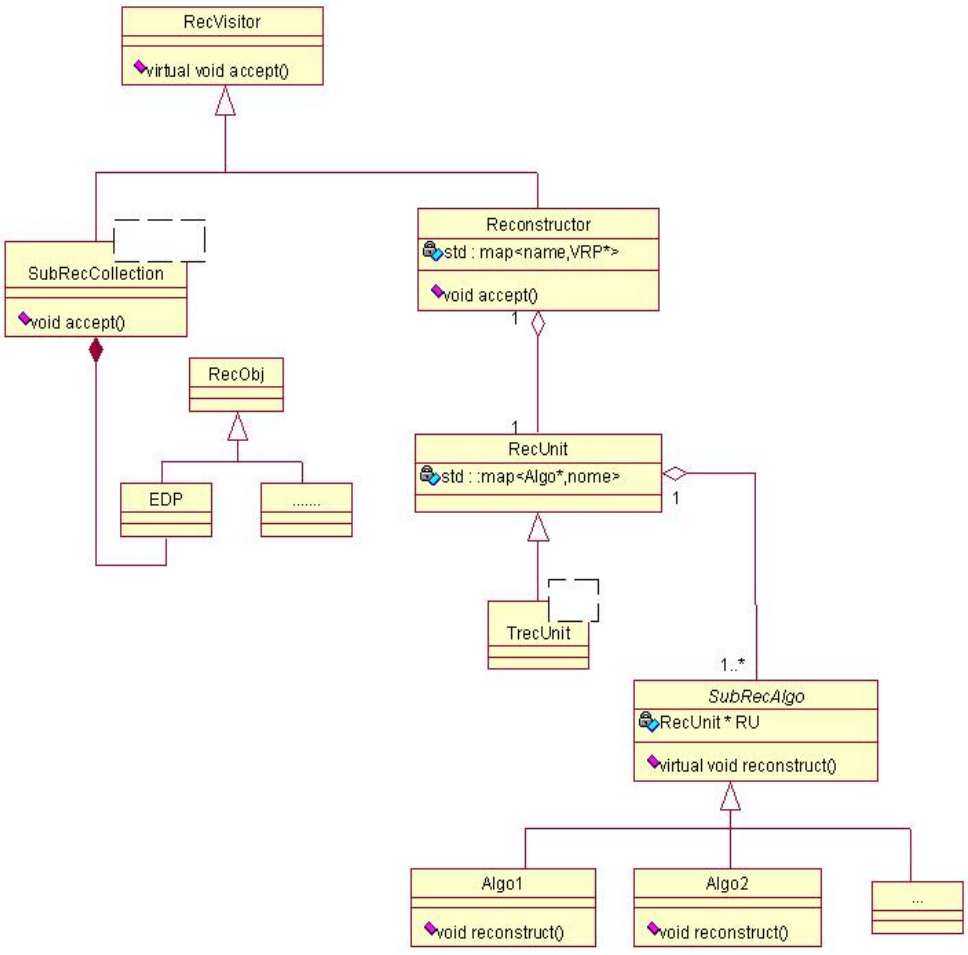

Figura 4.10: L'insieme delle classi principali della nuova libreria Reco

Use-Case applicati al Prototipo degli AOD

Dopo averne descritto il design del nuovo framework, passiamo a considerarne l'utilizzo applicandolo ad un un prototipo per l'analisi dei dati, gli ***Analysis Object Data Prototype***(AOD)[15], sviluppato presso la Sezione di Napoli dell'I.N.F.N.
Di seguito descriveremo brevemente la loro struttura e il loro funzionamento, tramite possibili Use-Case, con il nuovo modello di framework.

Le classi del prototipo degli AOD sono raggruppati in due pacchetti: **AODData**; **AODAlgos**.

Il pacchetto AODData contiene le classi che rappresentano gli elementi ricostruiti o le particelle e sono caratterizzati da alcune proprietà fisiche come il *momento*, la *carica*, il *vertice*, ecc. Questi, sono formati dalla composizione di altri oggetti detti *costituenti*[8].

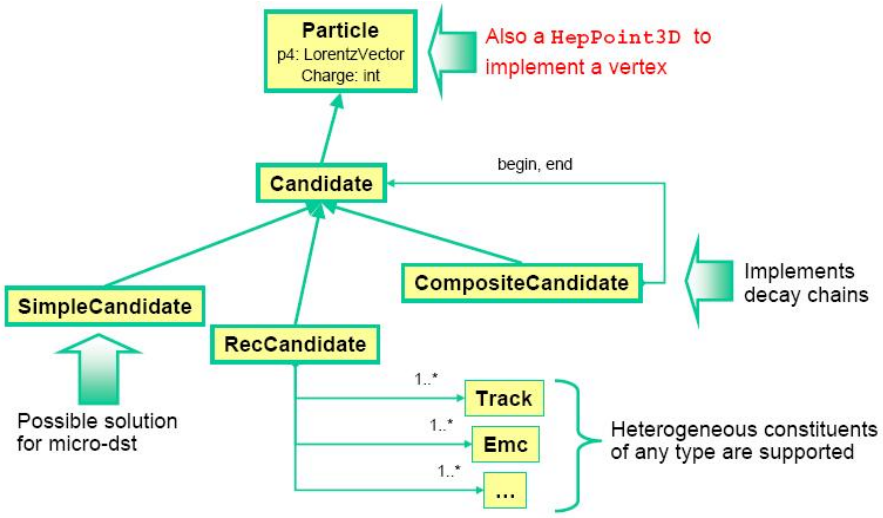

Figura 4.11: struttura schematica degli AODData

Le classi pricipali del pacchetto AODData sono rappresentate in Figura 4.11. La classe principale è ***Candidate*** che definisce l'interfaccia di base di un oggetto ricostruito.

[8] i costituenti possono essere ad esempio le particelle composite del decadimento $J/\psi \longrightarrow \mu\mu$ visto nella sez. 4.2.2

Il pacchetto AODAlgos contiene, invece, le classi di base per gli algoritmi di ricostruzione di catene di decadimento secondarie. Ad esempio le due classi **AtoXXBuilder** e **AtoXYBuilder** realizzano algoritmi che combinano rispettivamente costituenti simili (per esempio $J/\phi \longrightarrow \mu\mu$, sez 4.2.2) e costituenti eterogenei (per esempio $B_s \longrightarrow J/\psi\phi$, sez 4.2.2).

possibili Use-Case

Un possibile Use-Case può essere la ricostruzione del decadimento $B_s \longrightarrow J/\psi\phi$ (vedi sez. 4.2.2), implementato negli *AOD Prototype*.
La ricostruzione del B_s(vedi Figura 4.12) avviene partendo da due collezioni di input, μ e K che a loro volta sono generate dalle *Tracce* ricostruite. Le altre collezioni, invece, apparterranno ad un unico Reconstructor e gli algoritmi ad un'unica RecUnit, secondo lo schema mostrato in Figura 4.13.

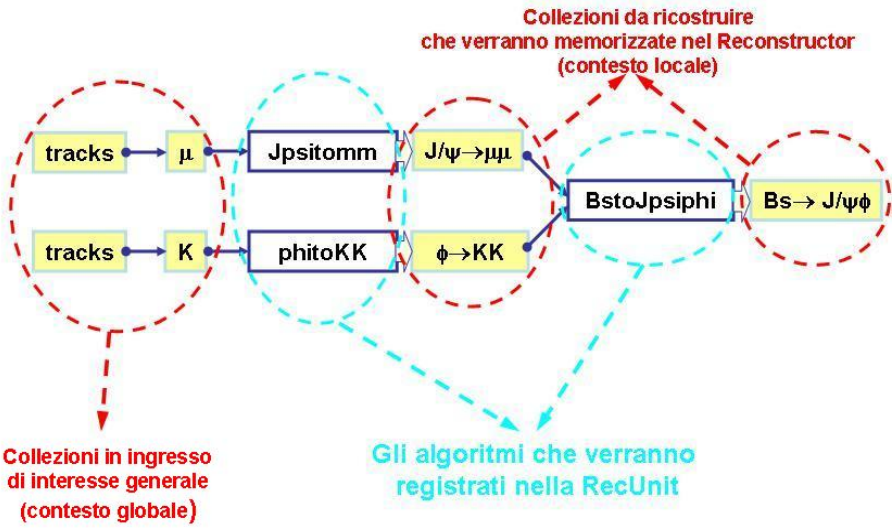

Figura 4.12: struttura della ricostruzione del decadimento B_s

Un possibile Use-Case si ha quando un programma che utilizza gli AOD Prototype richiede una collezione non ancora ricostruita. Ne riportiamo il diagramma di sequenza in Figura 4.14

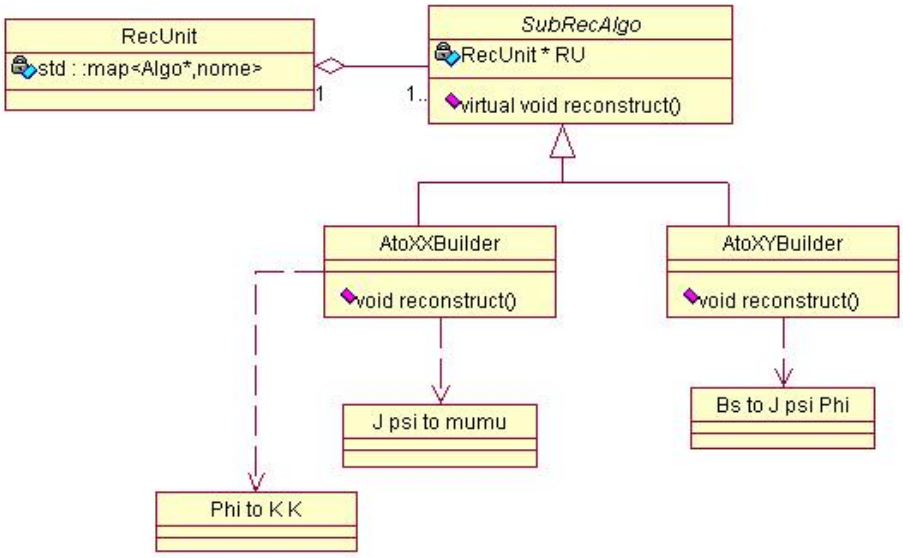

Figura 4.13: incapsulamento degli algoritmi in un'unica RecUnit

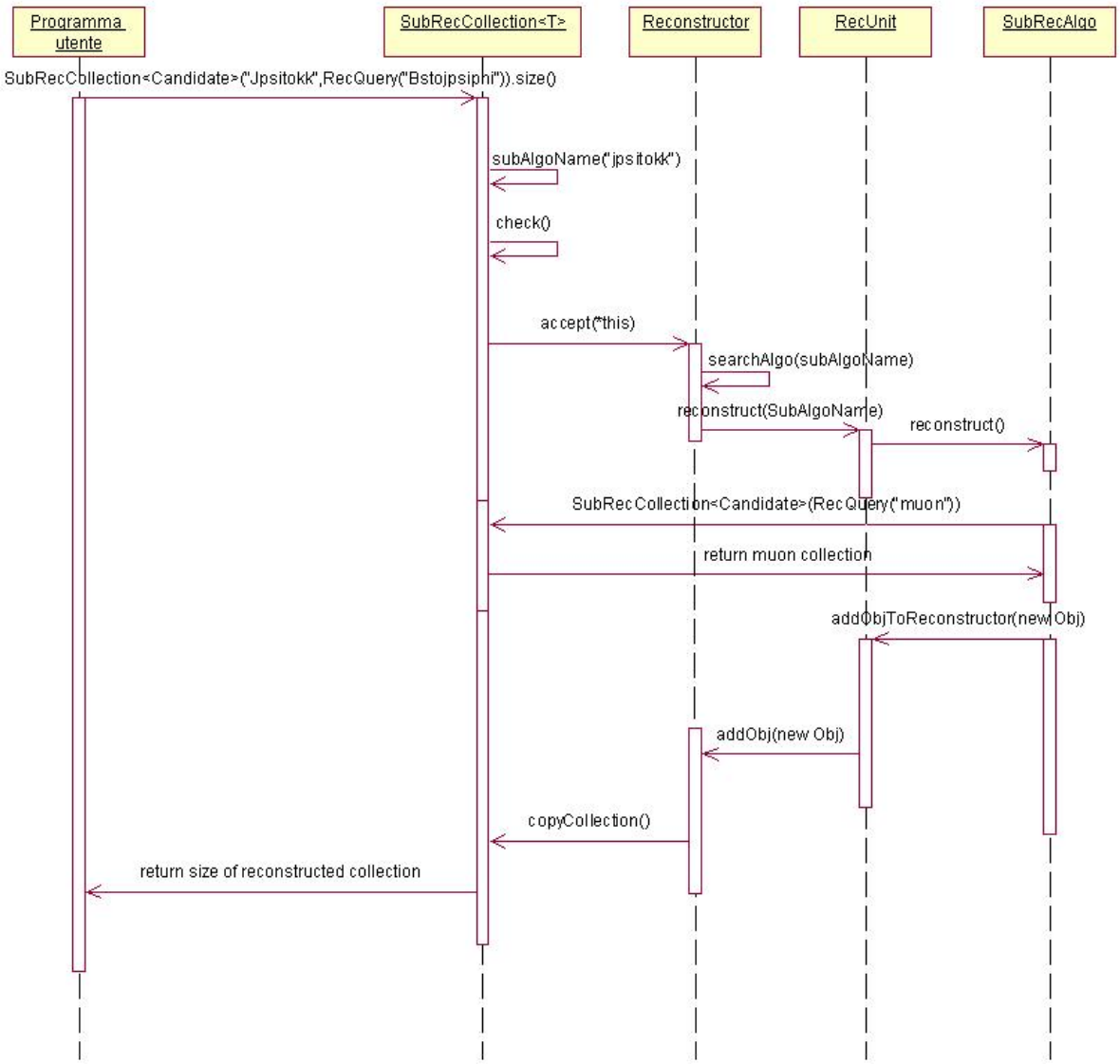

Figura 4.14: caso in cui una collezione richiesta non sia stata ancora ricostruita

Un altro possibile Use-Case si ha quando la collezione è già stata ricostruita. Ne riportiamo il diagramma di sequenza in Figura 4.15

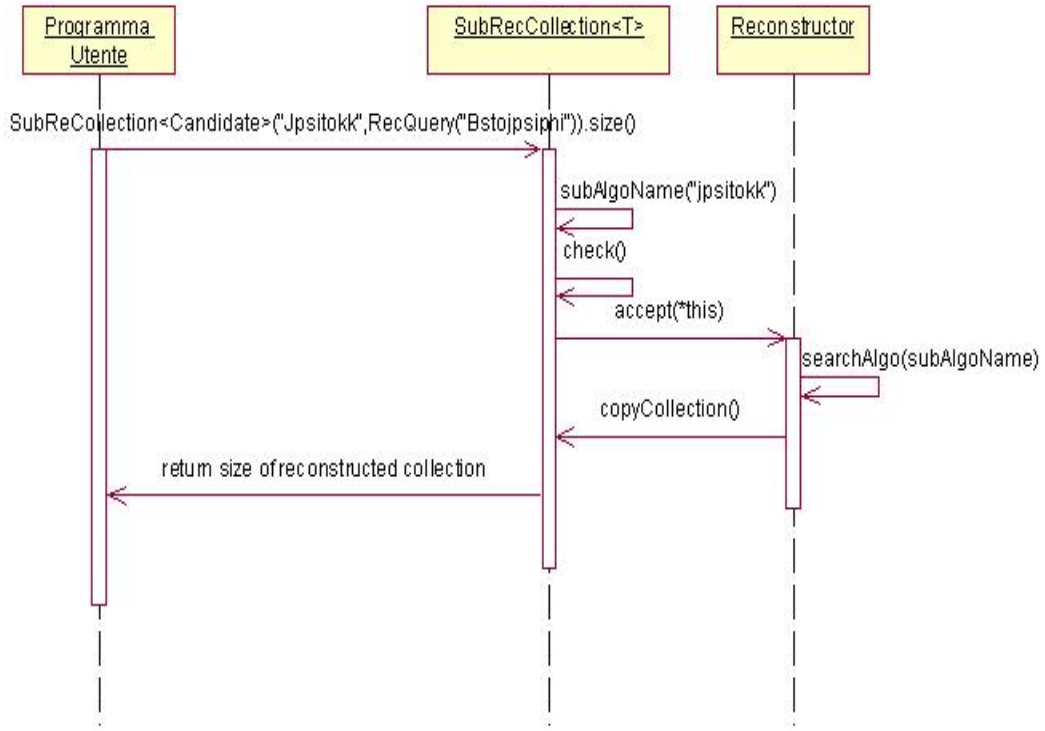

Figura 4.15: caso in cui una collezione richiesta sia già stata ricostruita

Capitolo 5

Conclusioni

Questo lavoro di tesi è stato svolto in due momenti, uno presso il CERN e l'altro nell'ambito del gruppo della sezione di Napoli dell'Istituto Nazionale di Fisica Nucleare (I.N.F.N.) con l'obiettivo di estendere il framework di ricostruzione utilizzato nel progetto CMS e applicarlo ad un'applicazione esistente. Questo framework ha lo scopo di ricostruire le collisioni protone-protone avvenute lungo l'accelleratore LHC e di fornire ai fisici un modello per effettuare misure e analisi sui dati ottenuti. La prima fase, iniziata questa estate presso il CERN di Ginevra (Svizzera) in qualità di Summer Student, ha avuto lo scopo di fornire un primo approccio con il framework e, prima ancora, di comprendere il tipo di ricerche e di esperimenti che sono in fase di sviluppo al CERN. Il framework, sviluppato con un linguaggio orientato agli oggetti (C++), presentava una struttura molto complessa da analizzare e, grazie al *supervisor* dott. Vincenzo Innocente, a cui ero stato affidato, sono riuscito a capirne il funzionamento e ad avere le basi per poter analizzare tale struttura. L'obbiettivo primario era quello di riuscire ad implementare il meccanismo di ricostruzione parziale, ossia la possibilità di ricostruire a richiesta una parte della collisione da analizzare. Per permettere questo tipo di ricostruzione è stato necessario apportare alcuni cambiamenti alla struttura esistente, mantenendone però lo standard in modo tale da garantire il funzionamento dei moduli sviluppati sulla base del vecchio modello.
Prima di ritornare in Italia ho presentato il lavoro svolto ad un meeting del gruppo fisico (PH Group) del CERN con cui ho collaborato ed ho rilasciato una versione funzionante insieme ad un report che ne descriveva le funzionalità.

La seconda fase è stata quella di utilizzare il nuovo framework con un'ap-

plicazione esistente (i Prototipi AOD) in modo tale da testarne le funzionalità. Questi prototipi sono stati sviluppati all'I.N.F.N. di Napoli dal dott. Luca Lista e dal dott. Francesco Fabozzi che hanno saputo illustrarmi le caratteristiche e sciolto ogni dubbio durante il processo di migrazione dal vecchio a nuovo framework.

Al termine di questa esperienza ho potuto capire cosa significa lavorare in un gruppo di progettazione e cosa vuol dire doversi attenere a degli standard. L'insegnamento più importante però è stato capire quei concetti della programmazione ad oggetti ed in particolare dell'analisi e design che determinano la robustezza e la durata del software prodotto. Ho dimostrato, in conclusione, le funzionalità del nuovo design del Framework realizzato durante il periodo di Summer Student al CERN applicandolo al prototipo sviluppato alla Sezione di Napoli dell'I.N.F.N. .

Bibliografia

[1] CMS, The Compact Muon Solenoid:thecnical proposal CERN-LHCC-54-38, 1994.

[2] LHC, Technical Design Report, 1994.

[3] CERN http://www.cern.ch.

[4] ALICE, Technical proposal, CERN-LHCC-95-71, 1995.

[5] LHCB, Technical proposal, CERN-LHCC-58-4.

[6] ATLAS, Technical proposal, CERN-LHCC-94-43, Dic. 1994.

[7] http://humanresources.web.cern.ch/HumanResources/external/students/students.asp

[8] E.Gamma, Design Patterns, Addison-Wesley, Reading, MA, 1994.

[9] Stoustrup, Bjarne The C++ Programming Language, 3rd Edition, Addison-Wesley, 1997. ISBN 0-201-88954-4.

[10] UML http://www.omg.orh/uml

[11] http://pool.cern.ch

[12] High Performance Storage System, http://www.sdsc.edu/hpps/hppsl.html

[13] http://www.na.infn.it

[14] M. Shaw, Some patterns for software architectures, in: Proc 2nd Workshop on Pattern Languages For Programming, Addison-Wesley, Reading, MA, 1996

[15] http://cms.na.infn.it/aod

[16] D.D. Corkill -Blackboard and Multi-Agent Systems & the Future

[17] decadimento $B_s \longrightarrow J/\psi\phi$

[18] P.W. Higgs, Phys Lett. 12 (1964) 132;
P.W. Higgs, Phys Rev. Lett. 13 (1964) 508;

[19] UML http://www.omg.org/uml/

[20] D.Vandevoorde, N. M. Josuttis C++ Templates:The Complete Guide, Hardcover

Appendice A

Appendice

A causa delle numerose classi presenti nel framework si è deciso di riportare solo i file di intestazione (*Header*) delle classi principali, omettendo le classi che sono state derivate per permettere l'effettivo funzionamento del framework. Queste ultime verranno solo menzionate nel diagramma di ereditarietà che accompagnerà la descrizione.

RecObj.h

```
# ifndef RECOBJ_H
# define RECOBJ_H
//
//  an Abstract Class for a generic Reconstructed Object

#include<string>
#include<iosfwd>
class Reconstructor;

/** A Reconstructed Object Class
Generic CMS Reconstructed Object
 */
class RecObj {

public:
    static std::string const & name() {
    static std::string const local("GenericRecObj");
    return local;
  };

  virtual std::string const & getName() const { return name();};
```

```
public:

  /// Constructors
  RecObj() : id_(-1), status_(0) , reconstructor_(0) {}

  explicit RecObj(int iid) :  id_(iid), status_(0),
reconstructor_(0) {}

  /// copy constructor (reset treconstructor_)
  RecObj(const RecObj& r) : id_(r.id_), status_(r.status_),
reconstructor_(0) {}

  /// destructor
  virtual ~RecObj();

  /// cloning for persistency
  virtual RecObj * replica() const;

  virtual void activate(){}

  virtual void deactivate(){}

  ///  reco methods
  virtual void update() { if (!amValid()) myUpdate();}

  /// tell methods
  inline int id()  const { return id_;}

  inline Reconstructor * reconstructor() const { return reconstructor_;}


  virtual void dump(std::ostream& co) const ;

protected:

  friend class TReconstructor;
  friend class TReconstructorR;
  template<class T> friend class TReconstructorV;

  /// force id
  inline void forceId(int iid) { id_ = iid;}

  friend class Reconstructor;
```

```
  // set reconstructor
  inline void setReconstructor(Reconstructor *  rec) {reconstructor_=rec;}
  friend class PReconstructor;

  virtual void myUpdate();

  virtual bool  amValid();

protected:

  int id_;
  int status_;

  Reconstructor * reconstructor_;

};
std::ostream & operator << (std::ostream & o, const RecObj & ro);
#endif // RECOBJ_H
```

RecUnit.h

```
#ifndef RecUnit_H
#define RecUnit_H
//
//  a Generic RecUnit
//

#include <string>

class Reconstructor;
class RecoRegistry;
class RecEvent;

class  PRecObjBuilder;

#include "CARF/Reco/interface/ConfigUnit.h"
#include "CARF/Reco/interface/RecUserRequest.h"

#include "CARF/Reco/interface/RecObj.h"

#include "CARF/Application/interface/CARFSkipEventexception.h"
#include<utility>

#include "CARF/Reco/interface/EDProducer.h"
#include "CARF/Reco/interface/BaseRecQuery.h"
```

```
#include "CARF/Reco/interface/TRecEvent.h"

class SubRecAlgo;

/** The generic Reconstruction Unit.
  a virtual class
 */
class RecUnit : public ConfigUnit {
public:
  typedef std::map< std::string,SubRecAlgo* > Maptype;
  typedef Maptype::iterator iteratorM;
  typedef Maptype::const_iterator const_iteratorM;
  typedef Maptype::value_type value_typeM;


  static RecoRegistry & registry();

public:

  /// exception to be thrown in case of Bad reconstruction
  class ReconstructionError : public CARFSkipEventexception {
  public:
    ReconstructionError() throw() {}
    virtual ~ReconstructionError() throw() {}

    explicit ReconstructionError(const char * s) : CARFSkipEventexception(s){}
    explicit ReconstructionError(const std::string& s) : CARFSkipEventexception(s){}
  };

public:

  /// the name of the deliverable RecObj
  static std::string const & deliverableName() { return RecObj::name(); }

  /// the name of the deliverable RecObj (virtual method...)
  virtual std::string const & nameOfRecObj() const {
return deliverableName(); }


public:
  typedef CARF::Reco::UserRequest UserRequest;


public:
```

```
  /// old constructor
  explicit RecUnit(const std::string & aname, bool regist=false);

  /// constructor
  explicit RecUnit(const BaseRecQuery & rq, bool regist=false);

  //private:
  RecUnit(const RecUnit &);

public:
  /// destructor
  virtual ~RecUnit();

  /// perform reconstruction
  virtual void reconstruct()=0;

  virtual void reconstruct(std::string const & subAlgoName);

  /// set PRecObj Builder
  void setPBuilder(PRecObjBuilder const * p) {
pBuilder_= const_cast<PRecObjBuilder *>(p);}

  /// build persistent version (return builder...)
  virtual PRecObjBuilder * goPersistent() { return pBuilder_;}

  void addAlgoToUnit(SubRecAlgo * subAlgo)

  /// assign current event reconstructor
  void setReconstructor(Reconstructor * reconstructor);

  ///build a new reconstructor
  virtual TReconstructor * buildReconstructor(
                   const TRecEvent * ev, RecUnit * ru )const=0;


  /// set User Request
  inline void setUserRequest(UserRequest request) {userRequest_ = request;}

  /// return User Request
  inline UserRequest getUserRequest() const { return userRequest_;}

  /// return User Request (alternate syntax)
  inline UserRequest userRequest() const { return userRequest_;}
```

```
  /// return its "name"
  std::string name() const;


  const RecEvent * event() const { return  event_;}

  /// return its "id"
  inline const RecoId & id() const { return id_;}

  /// return its "query"
  inline const BaseRecQuery & query() const { return id().me();}

  /// index in RecEvent
  int index() const { return id_.index();}

protected:
  friend class SubRecAlgo;
  template<typename R>
  R const * region() const;

  /// to be used by the children ...
  void addObjToReconstructor(RecObj * objR) const ;

  /// finalize intialization
  void bootstrap();

protected:

  RecoId id_;

  Reconstructor * reconstructor_;
  const RecEvent * event_;

  PRecObjBuilder * pBuilder_;

private:
  UserRequest userRequest_;

  bool registered;
  std::auto_ptr<EDProducer> edp;
  std::auto_ptr<EDProducer::Producing> pr;

private:
  friend class RecoRegistry;
```

```
};


#include "CARF/Reco/interface/SubRecEvent.h"

template<typename R>
inline R const * RecUnit::region() const {
  SubRecEvent<R> const * s = dynamic_cast<SubRecEvent<R> const *>(event_);
  if (s) return &(s->region());
  else return 0;
}

#endif  // RecUnit_H
```

Reconstructor.h

```
#ifndef Reconstructor_H
#define Reconstructor_H

//  a manager of reconstructed objects

#include <string>
#include<iosfwd>

#include "CARF/Reco/interface/RecUserRequest.h"
#include "Utilities/GenUtil/interface/own_ptr.h"
#include "CARF/Reco/interface/BaseRecQuery.h"
#include "CARF/Reco/interface/EDProducer.h"
#include <map>

class RecEvent;


class RecObj;
class RecData;

class RecVisitor;
class RecUnit;
class RecoRegistry;
class RecEvent;
class Reconstructor;

namespace TRecPrivate {
```

```
  /** helper class
   */
  class PWrap {
  public:
    struct Go {
      Go(RecEvent * him)  {
PWrap::count++;
PWrap::building = him;
      }
      ~Go() {
PWrap::count--;
if (PWrap::count==0) PWrap::building =0;
      }
    };
  public:
    static RecEvent * building;
    static bool going(); //true if going persistent...
    static int count;
    virtual ~PWrap(){}
    virtual  PWrap* clone() const=0;
    virtual Reconstructor & me()=0;
  };

}



/**  a manager of reconstructed objects.
     it's a part of Event
     it is responsible to check what actions have been required
     by the user
     and which actions are actually required to satisfy the request.
*/
class Reconstructor {

public:
  typedef TRecPrivate::PWrap PWrap;
  typedef std::map<std::string,RecVisitor*> Maptype;
  typedef Maptype::iterator iteratorM;
  typedef Maptype::const_iterator const_iteratorM;
  typedef Maptype::value_type value_typeM;


public:
```

```
  enum Action {Nothing, Reconstruct, Update};

public:

  static const std::string & localContext();

protected:

  static RecoRegistry & registry();

protected:

  /// default constructor
  Reconstructor();

  /// reconstructor for "aname"
  explicit Reconstructor(const RecoId & anId);

  explicit Reconstructor(RecUnit * ru);

  /// copy constructor (reset persistent...)
  Reconstructor(const Reconstructor& rh);

public:
  /// destructor
  virtual ~Reconstructor();


  /// clone in persistent
  virtual const PWrap & pclone() const=0;


  virtual void activate()  {}

  virtual void deactivate()  {}

  void setPersistent(PWrap * wrap) const {  persistent_.reset(wrap);}

  // if not yet persistent but ready to go
  virtual bool goPersistent() const =0;

  /// return present persistent copy
  inline PWrap * persistentCopy() const { return persistent_.get();}

  virtual void actuator(const RecEvent * ev);
```

```
  void setRecUnit(RecUnit * recUnit) { unit_=recUnit;}

  virtual void addObj(RecObj * objR) {}

  /// the check of the status and the corresponding action
  /// is still in a messy state
  bool check();

  ///set a new RecData
  virtual void set(RecData *)=0;

  ///get a RecData
  virtual const RecData * get() const=0;

  Action action() const;

  inline int obsolete() const { return action()!=Nothing;}


  inline const std::string & name() const { return myname();}

  inline RecoId ruId() const { return myId();}

  virtual EDProducer::DepSet const & dependencies() const=0;
  virtual EDProducer const & edpr() const=0;

  /// to be used for type check
  inline const RecObj * first() const {me().check();
return myfirst();}

  inline int size() const {me().check(); return mysize();}

  /// double dispatch entry point
  virtual void accept(RecVisitor& v) {}
  virtual void accept(RecVisitor& v) const {}

  /// return object i
  virtual const RecObj * get(int i) const { return 0;}

  virtual const RecObj * get(int i, std::string const & aname) const{
return 0;}

  CARF::Reco::UserRequest getUserRequest() const;
```

```
  virtual const RecEvent * eventR() const { return 0;}


  virtual void dump(std::ostream& co) const;

  virtual void dumpAll(std::ostream& co) const=0;

  virtual std::string const  & algoName()const =0;

protected:

  virtual const RecObj * myfirst() const { return 0; }

  virtual bool postcheck(){ return true;}

  virtual int mysize() const  { return 0;}

  virtual void reconstruct();

  virtual void update(){}

  virtual void addtoDep()=0;

  virtual void setAlgo(std::string const & aname)=0;

private:
  virtual const std::string & myname() const = 0;
  virtual RecoId myId()  const = 0;

protected:
  enum Status {UnReconstructed, Reconstructed, Updated,
Cloning, Writing, Doing};

  Status status() const {return status_;}
  void setStatus(Status newstatus) {status_=newstatus;}
  Status status_;

  RecUnit * unit() { return unit_;}

  RecUnit * unit() const { return me().unit_;}

  Reconstructor & me() const { return const_cast<Reconstructor&>(*this);}

  RecUnit * unit_; //! Transient
```

```
  mutable own_ptr<PWrap> persistent_;//! Transient

  mutable bool activated_; //! Transient


};


#endif //  Reconstructor_H
```

TReconstructor.h

```
TRECONSTRUCTOR
#ifndef TReconstructor_H
#define TReconstructor_H

//
//  a manager of reconstructed objects
//



#include "CARF/Reco/interface/EDProducer.h"

#include "CARF/Reco/interface/Reconstructor.h"

#include "CARF/Reco/interface/RecObj.h"
#include "CARF/Reco/interface/RecData.h"

#include "Utilities/GenUtil/interface/refc_ptr.h"
#include "Utilities/GenUtil/interface/own_ptr.h"

#include <map>

#include <vector>
#include <set>
#include <string>

class RecEvent;
class TRecEvent;

class RecVisitor;
```

```
class RecUnit;


/**  a manager of reconstructed objects.
     it's a part of Event
     it is responsible to check what actions have been required
     by the user
     and which actions are actually required to satisfy the request.
*/
class TReconstructor : public Reconstructor, public EDProducer {

public:
  typedef TRecPrivate::PWrap PWrap;

  //pointer to RecData
  typedef own_ptr<RecData, OwnerPolicy::Replica> APD;
public:

  TReconstructor() :  event(0){}

  /// constructor from the RecUnit identifier
  TReconstructor(const TRecEvent * ev, RecUnit * ru);

  /// default copy constructor goes deep....

   /// destructor
  virtual ~TReconstructor();

  ///status check and  corresponding action
  //virtual bool mycheck(RecVisitor&v)=0;

  /// cloning for persistency
  virtual   TReconstructor * replica() const=0;

  ///reset a RacData pointer
  inline void set(RecData * rData){
    recD.reset(rData);
  }

  ///get a RecData pointer
  inline const RecData* get() const {return recD.get();}

  /// clone in persistent
  virtual const PWrap & pclone() const { return pwrap();}
```

```
  // if not yet persistent but ready to go
  virtual bool goPersistent() const;


  virtual EDProducer::DepSet const & dependencies() const {
    return EDProducer::dependencies();
  }

  virtual EDProducer const & edpr() const { return *this;}


  virtual void addObj(RecObj * objR)=0;


  /// double dispatch entry point
  virtual void accept(RecVisitor& v)=0;
  virtual void accept(RecVisitor& v) const=0;


  virtual bool           empty() =0;
  virtual int            size() =0;
  virtual const RecObj & operator[](int i) =0;
  virtual const RecObj * get(int i)const =0;
  virtual const RecObj * get(int i,std::string const & aname)const =0;

  inline const RecEvent * eventR() const { return event;}

  virtual void dump(std::ostream& co) const;

  virtual void dumpAll(std::ostream& co) const=0;

  // make it persistent if not yet....
  virtual const PWrap & pwrap() const;//=0;


protected:

  virtual const RecObj * myfirst() {
    if (empty()) return 0;
    return get(0);
  }


  virtual int mysize() const=0;
```

```
  virtual void addtoDep() { addDep();}

  virtual void reconstruct()=0;

  virtual void update()=0;


private:


  friend class PReconstructor;
  RecObj * getO(int i) {
    return const_cast<RecObj *>(get(i));
  }
  RecData* get() {return recD.get();}


private:

  inline std::string const & locname() const { return name_;}

  inline const std::string &  myname() const { return locname();}

  RecoId myId() const;

  APD recD;

protected:

  std::string name_;

  const RecEvent * event;



};


#endif //  TReconstructor_H
```

TReconstructorR.h

```
#ifndef TReconstructorR_H
#define TReconstructorR_H

#include "CARF/Reco/interface/TReconstructor.h"
#include "CARF/Reco/interface/RecVisitor.h"
//  a manager of reconstructed objects by reference

#include<iostream>

class RecEvent;
class TRecEvent;
class RecVisitor;
class RecUnit;


/**  a manager of reconstructed objects.
     it's a part of Event
     it is responsible to check what actions have been required by the user
     and which actions are actually required to satisfy the request.
*/
class TReconstructorR : public TReconstructor{

  //public:
  // typedef TRecPrivate::PWrap PWrap;
public:

  ///algorithm scheduler
  typedef std::vector<std::string> Scheduler;

  // the container of RecObj
  typedef own_ptr<RecObj, OwnerPolicy::Transfer> AP;
  typedef std::vector<AP > VRP;
  typedef std::vector<AP>  Container;
  typedef VRP::iterator iterator;
  typedef VRP::const_iterator const_iterator;
  typedef VRP::value_type value_type;

  typedef std::map<std::string,VRP*> Maptype;
  typedef Maptype::iterator iteratorM;
  typedef Maptype::const_iterator const_iteratorM;
  typedef Maptype::value_type value_typeM;


public:
```

```
 //friend class TRecEvent;

 TReconstructorR(){}

 /// constructor from the RecUnit identifier
 TReconstructorR(const TRecEvent * ev, RecUnit * ru):TReconstructor(ev,ru){}

 /// deep copy
 TReconstructorR(const  TReconstructorR& tr);

 /// deep copy of a specific sub-event
 TReconstructorR(const TReconstructorR& tr, std::string const & aname);

 /// destructor
 ~TReconstructorR(){}

 virtual  TReconstructor * replica() const {
   return new TReconstructorR(*this);
 }

 ///status check and corresponding action
 bool mycheck();

 void addObj(RecObj * objR);


 /// double dispatch entry point
 void accept(RecVisitor& v);
 void accept(RecVisitor& v) const;

 ///update the algorithm scheduler
 void postSchedule(){
   if(scheduler_.size()==1){
       setAlgo(*(scheduler_.end()-1));
       scheduler_.erase(scheduler_.end()-1);
   }
   else {
       scheduler_.erase(scheduler_.end()-1);
       setAlgo(*(scheduler_.end()-1));
   }
}

 ///insert the last algorithm requested in the scheduler
 void schedule(std::string const & aname){
```

```
scheduler_.push_back(aname); mycheck();}


  inline std::string const & algoName()const{return aname_;}

  /// Return a pointer to a RecObj's vector of subalgorithm
  inline  VRP * getVec(std::string const  & aname) const  {
return ((*(mapsubalg.find(aname))).second);}


   inline const_iterator begin()  {return mapsubalg[algoName()]->begin();}
   inline const_iterator   end()  {return mapsubalg[algoName()]->end();}
   inline int             size()  {return msize();}
   inline bool           empty()  {return mapsubalg[algoName()]->empty();}

   inline const_iteratorM mbegin() const {return mapsubalg.begin();}
   inline const_iteratorM mend()   const {return mapsubalg.end();}
   inline int            msize()  const { return mapsubalg.size();}
   inline bool          mempty()  const { return mapsubalg.empty();}


   inline const RecObj & operator[](int i) {
                        VRP  recobjs = *(getVec(algoName()));
                        return *(recobjs[i]);
   }

   inline const RecObj * get(int i) const  {
                        VRP *recobjs=getVec(algoName());
                        return ((*recobjs)[i]).get();
   }


   inline const RecObj * get(int i, std::string const & aname) const {
     if(algoName()=="default")
 return ((*(getVec(algoName())))[i]).get();
     else
         return ((*(getVec(aname)))[i]).get();
   }

   /// dump all information about the reconstructor
   void dumpAll(std::ostream& co)const;

protected:

  int mysize() const { return  mapsubalg.size();}
```

```
  void reconstruct();

  void update();



protected:
  friend class PReconstructor;

  Scheduler scheduler_;

  Maptype  mapsubalg;

public:

  inline void setAlgo(std::string const & aname){aname_=aname;}

private:

  std::string aname_;

  void setDefAlgo(){setAlgo("default");}
};


#endif //  TReconstructorR_H
```

SubRecAlgo.h

```
#ifndef SUB_REC_ALGO_H
#define SUB_REC_ALGO_H

#include "CARF/Reco/interface/RecConfig.h"
#include "CARF/Reco/interface/RecUnit.h"

#include <string>

/*
   A class for a sub-reconstruction algorithm
*/

//class RecUnit;
#include <iostream>
```

```
class SubRecAlgo {

public:

  typedef  std::string     Name;
  typedef  std::string     Version;

  /* Constructor from RecConfig.
   */
  explicit SubRecAlgo(const RecConfig& config) : theConfig(config),theUnit(0){}

  virtual ~SubRecAlgo() {}

  virtual void setRecUnit(RecUnit * ru) { if(!(theUnit)) theUnit=ru;}

   /* Returns the current configuration of the SubRecAlgo.
   *  This is the default configuration, updated with .orcarc and
   *  the current RecQuery.
   */
  const RecConfig& config() const {return theConfig;}

  virtual Name name() const { return config().name();}

  virtual Version version() const { return config().version();}

  virtual void reconstruct()=0;
  template <class T>
  T parameter(const Name& name) const {
    return config().parameters().MultiTypeSet::value<T>( name);
  }

  const RecConfig& component(const Name& name) const {
    return config().component(name);
  }

  virtual const ParameterSet& parameters() const {
return config().parameters(); }

  virtual const ComponentSet& components() const {
return config().components(); }

protected:
  void addObjToReconstructor(RecObj * objR) const {
theUnit->addObjToReconstructor(objR);}
```

```
private:

  RecConfig theConfig;
  RecUnit *  theUnit;

};

#endif //SUB_REC_ALGO_H
```

SubRecCollection.h

```
#ifndef SubRecCollection_H
#define SubRecCollection_H

//
// a sub rec collection which "reconstruct"
// objects of type T when dispatched an event of type EV
//

#include "Utilities/Notification/interface/LazyObserver.h"
#include<utility>
#include "CARF/Reco/interface/BaseRecCollection.h"
#include "Utilities/Configuration/interface/Architecture.h"
/**
    rec collection which "reconstruct"
    objects of type T when dispatched an event of type Event (aka RecEvent)
 */
template<class T>
class SubRecCollection :
      private LazyObserver<typename BaseRecCollection<T>::Event const *> {
public:

  typedef LazyObserver<typename BaseRecCollection<T>::Event const *> super;
  typedef BaseRecCollection<T> Collection;

  typedef typename Collection::Event const *  Event;

  typedef typename Collection::iterator iterator;
  typedef typename Collection::const_iterator const_iterator;
  typedef std::pair<const_iterator,const_iterator> Range;
  typedef typename Collection::const_reference const_reference;

  typedef size_t size_type;
```

```
  /// constructor
  SubRecCollection() {}
  explicit SubRecCollection(const BaseRecQuery & rq) : collection(rq){}
  explicit SubRecCollection(const BaseRecQuery & rq,const
                            std::string & aname) : collection(rq,aname){}
  explicit SubRecCollection(const std::string & aname) : collection(aname) {}
  explicit SubRecCollection(const RecEvent * eventR) : collection(eventR){}
  explicit SubRecCollection(const RecEvent * eventR,
                      const std::string & aname) : collection(eventR,aname){}

  /// destructor
  ~SubRecCollection(){}

  ///
  const_iterator begin() const { check(); return collection.begin();}
  ///
  const_iterator end() const { check(); return collection.end();}
  ///
  int size() const { check(); return collection.size();}

  // empty?
  inline bool empty() const {check(); return collection.empty();}

  ///
  Range cache() const { check();
                   return Range(collection.begin(), collection.end());}
  Range range() const { check();
   return Range(collection.begin(), collection.end());}
  Range operator()() const {
    check(); return Range(collection.begin(), collection.end());
  }

  /// return first element
  inline const_reference front() const { return *begin(); }

  /// return last element
  inline const_reference back() const { return *(end() - 1); }

  /// return element n
  inline const_reference operator[](size_type n) const {
return *(begin() + n); }

  ///
  const Reconstructor * reconstructor() const {
```

```
     check(); return collection.reconstructor();}

  std::string name() const {
    return collection.name();
  }

  /// return its "query"
  inline const BaseRecQuery & query() const {
return collection.query();}

  /// return its "config"
  inline const BaseRecQuery & config() const { return collection.query();}

private:
  void check() const {
   super::check();
  }

  virtual void lazyUpDate(Event ev) {
    if (ev!=0) {
      collection.reset( (*ev)());
    }
  }

private:
  mutable Collection collection;
};

#endif // SubRecCollection_H
```