UNIVERSITÀ DEGLI STUDI DI BARI

FACOLTÀ DI SCIENZE MATEMATICHE FISICHE E NATURALI

CORSO DI LAUREA IN FISICA

TESI DI LAUREA IN FISICA

PROGETTAZIONE DEL SOFTWARE DI VISUALIZZAZIONE GRAFICA DEL TRACCIATORE DI CMS E SVILUPPO DI UN PROTOTIPO PER LA VISUALIZZAZIONE IN 2D

Relatori:
Dott. Giuseppe Zito
Prof. Giorgio Maggi

Laureanda:
Antonietta Regano

ANNO ACCADEMICO 2002-2003

11 luglio 2003

Indice

Introduzione

Questa tesi affronta il problema della progettazione di un prototipo per la visualizzazione in due dimensini (2D) del sistema tracciante (*Tracker*), delle tracce e degli *hit*, da inserire come *plugin*, nel progetto software di analisi grafica interattiva *off-line* dell'esperimento CMS, in costruzione al CERN. Il software sviluppato è stato realizzato in maniera modulare nell'ambito della programmazione orientata ad oggetti (*Object Oriented*) allo scopo di poterlo utilizzare anche al di fuori del software di CMS, ad esempio all'interno di un programma *stand alone*.

Il sistema tracciante dell'esperimento CMS è complesso e articolato e quindi sono necessari opportuni strumenti per visualizzare nel dettaglio l'apparato e gli eventi, in particolare per visualizzare sezioni sempre più piccole, fino ad arrivare al singolo modulo e al singolo canale di lettura, e per visualizzare le tracce, i vertici primari e secondari e gli *hit* ricostruiti e simulati dell'evento. Data la complessità dell'apparato e degli eventi che si vogliono studiare, la sola visualizzazione in tre dimensioni (3D) è risultata talvolta di non facile comprensione, quindi è stata proposta l'integrazione nel software di analisi e grafica interattiva di finestre complementari di visualizzazione in 2D.

In un lavoro precedente [1] è stato realizzato un software dimostrativo (demo) di visualizzazione in 2D che ha dimostrato quanto risulti più facile ed intuitivo l'uso di oggetti grafici in due dimensioni da parte di un utente che può anche non conoscere la geometria del *Tracker*. Quindi la collaborazione CMS ha richiesto lo sviluppo software completo di un prototipo con tutti gli oggetti grafici in 2D.

Il lavoro di questa tesi si è svolto in due fasi: (1) progettazione di tutte le classi che realizzeranno tutti gli oggetti per la visualizzazione in 2D; (2) implementazione di alcune delle suddette classi per realizzare in particolare l'oggetto mappa 2D del *Tracker*.
Nella progettazione si è fatto uso delle tecniche di analisi e disegno orientato agli oggetti che si stanno affermando come tecniche standard nel lavoro di sviluppo di grandi progetti software.

La tesi è articolata in 6 capitoli.

Capitolo 1 - L'esperimento CMS

Fornisce una descrizione alquanto dettagliata del sistema di tracciamento di CMS per mettere in evidenza la complessità dell'apparato e la precisione con cui si effettueranno le misure fisiche.

Capitolo 2 - Il progetto software di CMS

Illustra il software *off-line* di CMS, in particolare il software di visualizzazione e analisi grafica interattiva e le nuove finestre per la visualizzazione del *Tracker* in 2D.

Capitolo 3 - Tecniche di programmazione

Introduzione alla filosofia della programmazione *Object Oriented*, e presentazione delle librerie e di altri strumenti software usati per la progettazione e realizzazione del prototipo. In particolare vengono presentate le nuove tecniche di analisi e disegno orientati agli oggetti (UML e *pattern*) utilizzate.

Capitolo 4 - Progettazione del tool di visualizzazione del Tracker

In questo capitolo si descrivono le funzionalità richieste al prototipo, la fase di progettazione dello stesso attraverso diagrammi UML e la tecnica per la realizzazione del *plugin* che permette di inserire il *tool* di visualizzazione del *Tracker* all'interno del software di analisi e grafica interattiva di CMS.

Capitolo 5 - Mappe 2D del Tracker

Illustra e descrive le mappe 2D del *Tracker* che sono state ideate per visualizzare la geometria e le caratteristiche del tracciatore e gli eventi (*hit* simulati e ricostruiti).

Capitolo 6 - Esempio di uso delle mappe 2D

Illustra un esempio di uso delle mappe 2D del *Tracker*.

Capitolo 1

L'esperimento CMS

1.1 Il Large Hadron Collider

La fisica delle alte energie (*High Energy Physics*) si avvale per le sue ricerche di acceleratori di particelle sempre più potenti. Nell'ultimo decennio, in Europa, un gran contributo è stato fornito dal LEP (*Large Electron Positron collider*), costruito al CERN, e che ha permesso di raggiungere l'energia limite nel centro di massa di ~ 208 GeV per le collisioni elettrone-positrone.

Alla fine dell'anno 2000 è cominciata la costruzione di un nuovo acceleratore per adroni che sarà ospitato nel tunnel di 27 Km di circonferenza del LEP. Questo progetto, noto come *Large Hadron Collider* (LHC)[2], sarà capace di produrre collisioni protone-protone con un'energia nel centro di massa di 14 TeV, un valore di energia mai raggiunta dai precedenti acceleratori.

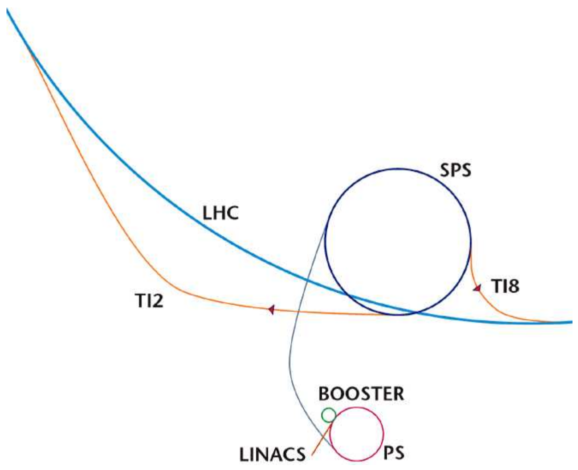

Figura 1.1: Sistema di preaccelerazione e iniezione nell'anello di LHC.

I protoni saranno iniettati nell'anello dell'LHC dopo tre stadi di pre-accelerazione (fig.1.1) che useranno prima un acceleratore lineare (LINACS), capace di accelerare i protoni fino a 50 MeV, poi un ProtoSincrotone (PS) fino a 26 GeV, e infine un SuperProtoSincrotone (SPS) fino a 450 GeV, quindi i due fasci saranno forzati a percorrere una traiettoria circolare per mezzo di magneti superconduttori e verranno ancora accelerati per mezzo di cavità risonanti. Una volta raggiunta l'energia finale i fasci verranno fatti interagire nei quattro punti in cui saranno installati i quattro rivelatori previsti dal progetto di LHC (fig.1.2): CMS (*Compact Muon Solenoid*), ATLAS (*A Toroidal LHC ApparatuS*), ALICE (*A Large Ion Collider Experiment*) e LHCb. I primi due rivelatori, *general purpose*, una volta terminati, saranno utilizzati per la ricerca dei bosoni di Higgs e di indizi di supersimmetria, gli altri due, studieranno fenomeni più specifici, in particolare ALICE si occuperà dell'interazione tra ioni pesanti e LHCb della fisica del quark b (*beauty*) con particolare rilievo alla violazione di CP.

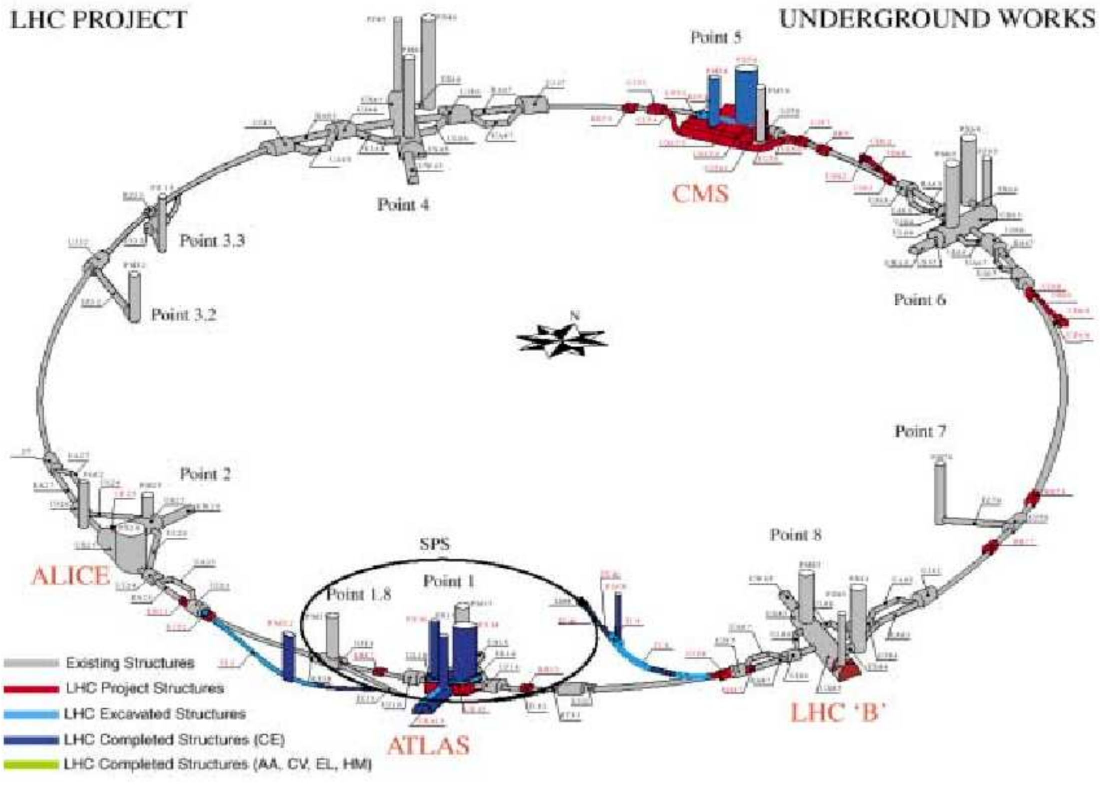

Figura 1.2: L'anello di LHC ed i punti in cui saranno costituiti i 4 rivelatori

Una delle richieste rilevanti, per simili esperimenti, è la necessità di lavorare ad alta luminosità[1], questo perchè la sezione d'urto decresce con l'inverso del quadrato della massa della particella che si vuole rivelare. Quindi ad LHC dopo un periodo iniziale a bassa lunimosità (2×10^{33} cm^{-2}s^{-1}), si passerà a una fase di alta luminosità (10^{34} cm^{-2}s^{-1}) che renderà possibile la ricerca di nuove particelle con masse fino a $\sim$ 5 TeV. Per raggiungere tali luminosità, ogni fascio sarà costituito da 2835 pacchetti (*bunches*) composti a loro volta da circa 1.1×10^{11} protoni che corrisponde a una frequenza di incidenza dei pacchetti di 40 MHz nella fase di alta luminosità con 25 interazioni per ogni *bunch crossing*. La maggior parte di queste interazioni sono di tipo *minimum bias*[2] e per selezionare i rari eventi interessanti dopo un *trigger* di primo livello, seguirà la discriminazione dell'informazione con complessi algoritmi informatici, che, in tempo reale, selezioneranno quali eventi scegliere per l'analisi successiva *off-line* più complessa.

1.2 Il rivelatore CMS

L'esperimento CMS a LHC si propone come obiettivo quello di rispondere ad alcune questioni fondamentali della fisica delle particelle ancora irrisolte:

- Qual'è l'origine della massa? Il Modello Standard predice l'esistenza del bosone di Higgs, il cui valore della massa deve essere compreso tra un limite inferiore di 114 GeV/c^2 (individuato dalle misure sperimentali fatte a LEPII) e un limite superiore di circa 1 TeV/c^2 .

- Esistono evidenze sperimentali che supportano l'ipotesi della Grande Unificazione delle tre interazioni fondamentali - elettromagnetica, debole, e forte?

- Qual'è l'origine della "materia oscura" nell'universo? Esiste una nuova fisica al di là del Modello Standard, caratterizzata da nuove particelle che potrebbero spiegare l'esistenza della materia oscura (neutralinos)?

- Perchè ci sono asimmetrie nella quantità di materia confrontata con l'antimateria?

- Le particelle elementari hanno una sottostruttura?

[1] La luminosità L è quel fattore di proporzionalità che lega la frequenza degli eventi alla sezione d'urto di interazione in un collisore. La luminosità è normalmente espressa in cm^{-2}s^{-1}.

[2] *Minimum bias*: eventi a basso impulso trasverso, dovuti a collisioni periferiche tra protoni.

- Esistono nuove forme di materia, come il cosiddetto plasma quark-gluon, e come sarebbe dovuto esistere nell'universo iniziale ?

Per rispondere a queste domande la collaborazione CMS ha progettato un rivelatore composto da:

a) un solenoide in grado di generare un intenso campo magnetico del valore di 4 Tesla, per permettere la più accurata misura possibile del momento dei muoni dell'ordine del TeV nella regione di pseudorapidità[3] $|\eta| < 5$.

b) un calorimetro elettromagnetico per la misura dell'energia di fotoni (γ) ed elettroni ($e^{\pm}$) con la migliore precisione possibile e consistente con il punto a);

c) un rivelatore centrale per misurare il parametro d'impatto e il momento della particella carica per ottenere una traccia di alta qualità per verificare i punti a) e b);

L'apparato in via di realizzazione sarà in grado di lavorare ad alta e bassa luminosità e di identificare, in un'ampia regione cinematica, attraverso precise misure di energia e di impulso, muoni, elettroni, fotoni e *jet* che, con la loro presenza, potrebbero segnalare la produzione di nuove particelle come il bosone di Higgs.

Proprio per ottenere gli obiettivi prefissi il rivelatore CMS deve soddisfare precisi requisiti:

- alta granularità (cioè elementi sensibili di piccole dimensioni), e un gran numero di canali di rivelazione per ottenere un'efficiente ricostruzione delle tracce e individuare con elevata precisione le coordinate del punto di passaggio delle particelle;

- resistenza alle radiazioni in quanto, l'alta energia del fascio combinata con l'elevata luminosità creerà un ambiente di radiazione particolarmente elevato, soprattutto per i rivelatori più vicini al punto di collisione;

- un'elettronica di lettura veloce per evitare la sovrapposizione (*pile-up*) dei segnali relativi a eventi provenienti da collisioni di diversi pacchetti.

[3] La pseudorapidità è una grandezza cinematica definita come η=-ln(tan(θ/2)) essendo θ=arcos(p_z/p) in cui p e p_z sono l'impulso della particella e la sua proiezione lungo la direzione z dei fasci.

1.2.1 La geometria di CMS

Seguendo il classico schema dei rivelatori impiegati nella fisica delle alte energie, l'apparato di CMS (fig.1.3) si sviluppa secondo una simmetria cilindrica intorno al punto di interazione dei fasci ed è costituito da diversi rivelatori, alloggiati l'uno all'interno dell'altro, che hanno il compito di eseguire specifiche osservazioni, prima che le particelle entrino in quello successivo.

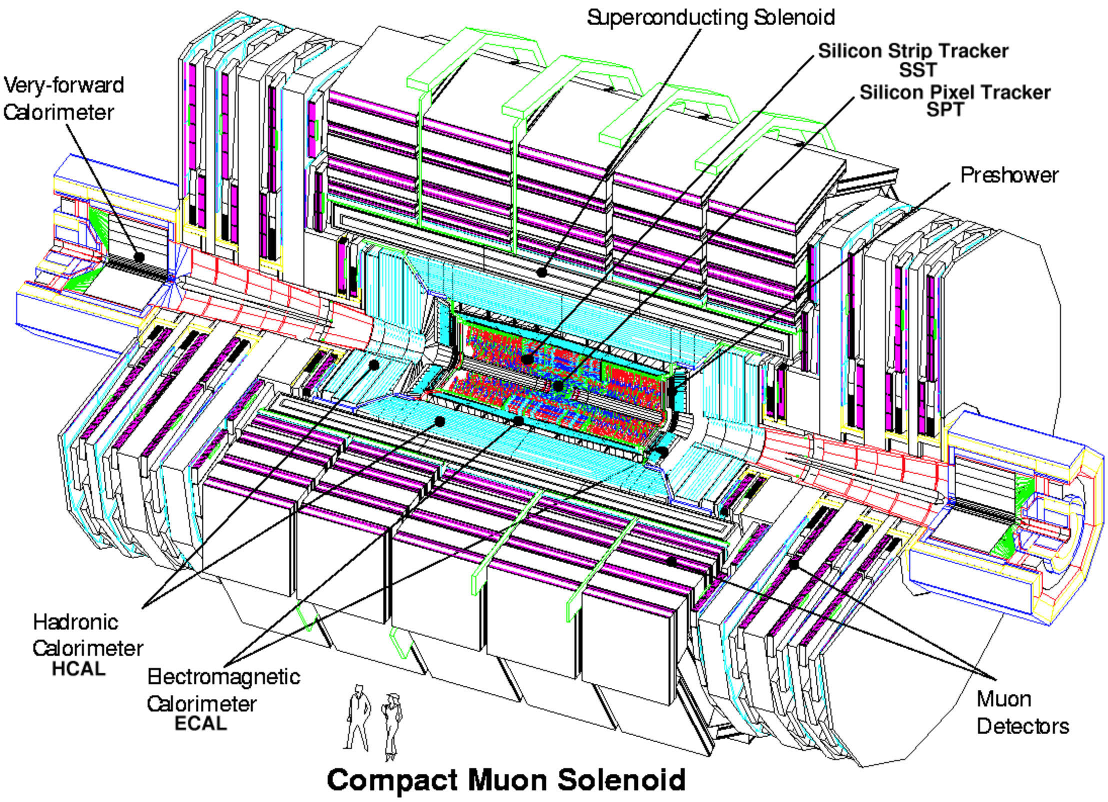

Figura 1.3: Struttura interna dell'apparato CMS.

Ogni sistema di rivelazione è formato da una zona centrale, *barrel*, e due terminali, *endcap*. Nella prima gli elementi sensibili dei rivelatori, collocati su superfici cilindriche, sono posti parallelamente all'asse del fascio e nelle altre due, su dischi coassiali e perpendicolari allo stesso asse. Le dimensioni complessive dell'apparato sono: lunghezza 22 m e diametro 14.6 m, per un peso totale di 12500 tonnellate.

Il sistema di coordinate cartesiane utilizzato ha l'asse Z diretto secondo la direzione dei fasci, l'asse X punta verso il centro dell'anello di accumulazione e l'asse Y verso l'alto (fig.1.6).

L'apparato sperimentale può essere schematicamente suddiviso in quattro sistemi principali: il rivelatore interno (rivelatore di vertice e tracciatore), i calorimetri (elettromagnetico e adronico), il solenoide superconduttore e il rivelatore di muoni.

Il primo apparato che le particelle secondarie prodotte incontreranno è il rivelatore di vertice (*Silicon Pixel Tracker*) e il sistema tracciante (*Silicon Strip Tracker*), atti a ricostruire le tracce. In seguito le particelle giungeranno prima al calorimetro elettromagnetico (ECAL), cui è demandata la misura della posizione e dell'energia degli elettroni e dei fotoni, e poi a quello adronico (HCAL), che identificherà gli adroni e ne misurerà energia e posizione. Le particelle che riusciranno a superare indisturbate il sistema calorimetrico saranno muoni e neutrini. Queste, dopo aver attraversato il solenoide superconduttore, che produce un intenso campo magnetico di 4 Tesla, diretto lungo l'asse Z, entreranno nel rivelatore muonico, la regione più esterna dell'apparato. Se si tratta di muoni sarà possibile determinare l'impulso attraverso le tracce lasciate nel rivelatore, mentre i neutrini saranno rilevati da una mancanza di energia tra lo stato iniziale e quello finale.

1.3 Il Tracker: sistema di tracciamento interno

Ora segue una descrizione più dettagliata del sistema di tracciamento interno [3], perchè il lavoro svolto in questa tesi si propone proprio di progettare nuovi oggetti grafici 2D per il software di visualizzazione del *Tracker* e degli eventi.

Il compito fondamentale per un tracciatore a LHC è la "*pattern recognition*", ovvero il riconoscimento delle singole tracce in eventi con una elevata densità di particelle cariche. Ad una luminosità di 10^{34} cm^{-2}s^{-1} le tracce interessanti dal punto di vista fisico saranno mescolati a circa 1000 tracce cariche derivanti da eventi di *minimum bias* che si hanno nella stessa collisione dei pacchetti (*bunch crossing*). Per discriminare le tracce interessanti dalle altre, ad alta luminosità, si richiede un'elevata granularità e il maggior numero possibile di punti per traccia (*hit*). Le simulazioni delle collisioni hanno dimostrato che in assenza del campo magnetico la densità delle tracce diminuisce come $1/\mathrm{r}^2$; sotto l'effetto del campo di 4 T, la diminuzione di queste è inizialmente più graduale e poi significativamente più pronunciata di $1/\mathrm{r}^2$. Tutto ciò ha una importante implicazione per l'architettura del tracciatore di CMS; per cui sono state scelte due diverse tipologie di rivelatori: a *pixel* di silicio e a *microstrip* di silicio.

Il problema forse più critico, poi, riguarda la funzionalità a lungo termine sotto l'influenza di un pesante irraggiamento. Per garantire una buona funzionalità, i rivelatori al silicio avranno bisogno di essere raffreddati; quindi, l'intero volume del sistema tracciante sarà mantenuto permanentemente ad una temperatura di circa -10°C durante la presa dati e solo per limitati periodi di tempo raggiungerà la temperatura di 0°C per la manutenzione.

1.3.1 La geometria del Tracker

Il tracciatore sarà composto da rivelatori a *pixel* di silicio (*Silicon Pixel Tracker*) nella zona più interna e a *microstrip* di silicio (*Silicon Strip Tracker*) nella zona più esterna; questi ultimi a loro volta sono divisi in una parte interna ed in una esterna. Tutto l'insieme del tracciatore occupa un cilindro lungo 5,40 m con un diametro esterno di 2.2 m.

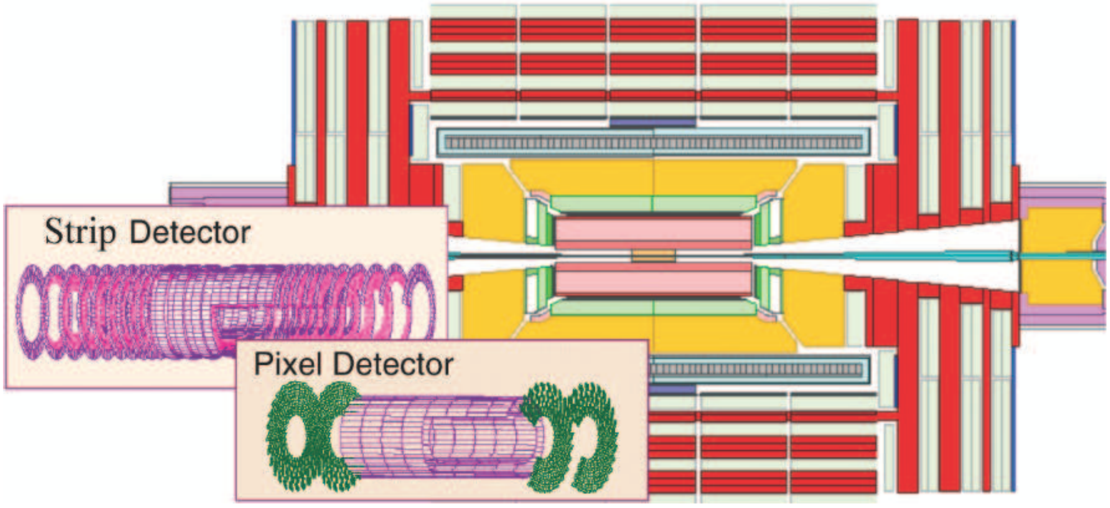

Figura 1.4: Sezione del CMS in cui sono mostrati il *Silicon Strip Detector* (SST) e il *Silicon Pixel Detector* (SPT).

Anche per il tracciatore i dispositivi seguono la divisione in *barrel* e *endcap* già vista per le altre componenti di CMS. La struttura è ottimizzata per ottenere in media 12-14 *hit* per traccia carica, in modo da assicurare sia un'elevata efficienza sia una bassa frequenza di *ghost*[4].

Il Silicon Pixel Tracker (SPT)

Il sistema a *pixel* di CMS consiste di tre o due cilindri nella zona del *barrel* e due dischi per ogni *endcap*. In figura 1.5 è riportato lo schema tridimensionale

[4]Si definisce *ghost* una traccia ricostruita per errore e che in realtà non corrisponde a nessuna delle tracce delle particelle che hanno interagito all'interno del rivelatore.

del rivelatore a *pixel*.

Le superfici cilindriche sono costituite da unità di rivelatori modulari, ognuno consistente in un sottile sensore di silicio segmentato con *pixel* n^+ su un substrato di tipo n. Il compito principale del sistema di *pixel* è la misura

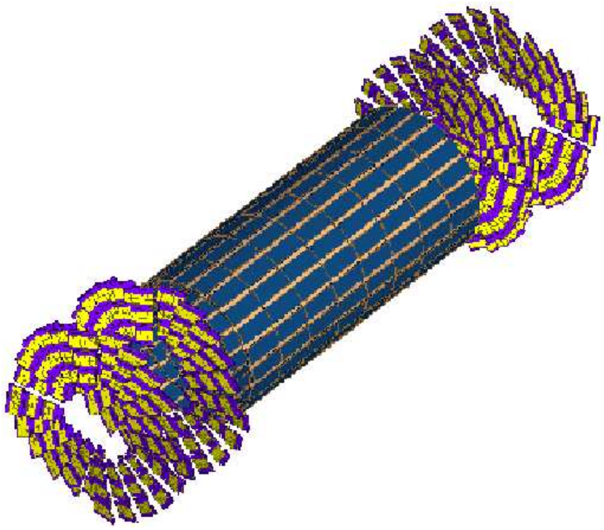

Figura 1.5: Vista tridimensionale del rivelatore a *pixel* di silicio.

del parametro d'impatto della traccia, ovvero la distanza minima dal vertice di interazione, ed è per questa ragione che è stata scelta una forma quadrata per i *pixel* in modo da ottimizzare la risoluzione spaziale in entrambe le coordinate simultaneamente.
Nella regione del *barrel*, la dimensione dei *pixel* sarà di circa 150 μm x 150 μm con uno spessore del rivelatore compreso fra 200 e 250 μm. Nei dischi degli *endcap*, il campo magnetico è parallelo al campo elettrico e la maggior parte delle tracce avranno un angolo di incidenza vicino alla normale del rivelatore. Per questa ragione, i rivelatori saranno ruotati di circa 20° intorno al loro asse di simmetria radiale così da aumentare la divisione di carica fra i *pixel* vicini e migliorare la risoluzione in ϕ.

La configurazione dei rivelatori a *pixel* nel *barrel* sarà differente nel periodo a bassa luminosità e in quello ad alta luminosità. Nella prima fase il primo strato di *pixel* avrà un raggio di 4 cm e il secondo avrà un raggio di 7 cm, con una superficie aggiuntiva posta a 11.5 cm per migliorare il riconoscimento delle tracce ed il *trigger*; i tre strati saranno identici in lunghezza (53 cm lungo z) ed ogni strato consisterà di "*ladder*" di 8 rivelatori ciascuno. Ad alta luminosità, invece, si prevede di eliminare lo strato a 4 cm lasciando gli altri due. Tuttavia, risulterà necessaria una sostituzione dello strato a 7 cm dopo sei o sette anni di presa dati causa il danneggiamento delle radiazioni.

I due dischi negli *endcap* sono posti a $|z|$=34.5 cm e 46.5 cm e coprono la regione compresa tra 6 cm $< r <$ 15 cm. Ogni disco è formato da 24 lame (*blade*) disposte secondo una geometria a turbina, cioè ogni lama, formata da 7 rivelatori (4 nella parte anteriore e 3 in quella posteriore), risulta ruotata di 20° intorno al proprio asse di simmetria radiale. Anche in questo caso è prevista la sostituzione della parte più interna durante l'esperimento.

Il Silicon Strip Tracker (SST)

I rivelatori a *microstrip* grazie a un'elevata precisione spaziale e risoluzione temporale, combinata con un'adeguata resistenza alla radiazione ostile, sono l'ideale per la regione esterna del tracciatore di CMS.

Il SST occupa una regione di circa 5.6 m lungo l'asse z, coprendo una zona che va dai 20 cm ai 110 cm in r, ed un'area attiva di 230 m^2. La regione del *barrel* è costituita da 10 cilindri concentrici che ci forniscono informazioni sulle coordinate r e ϕ; le quattro superfici più interne formano il *Tracker Inner Barrel* (TIB) e si estendono da -65 cm a +65 cm lungo l'asse z, e sono costituiti da sottili rivelatori rettangolari. I sei cilindri più esterni formano il *Tracker Outer Barrel* (TOB) e si estendono da -110 cm a +110 cm lungo l'asse z, e sono costituiti da rivelatori più spessi (500 mm).

Per quanto riguarda gli *endcap*, sono costituiti da 9 dischi, *Tracker End Caps* (TEC), la cui posizione lungo l'asse z (120 cm $<= |z| <=$ 280 cm) è stata ottimizzata per la piena copertura e da tre dischi più piccoli, *Tracker Inner Disks* (TID), che chiudono l'*inner barrel*; tutti gli anelli ci danno le misure in zϕ. Per evitare zone morte nel volume del tracciatore e permettere un allineamento più facile, i rivelatori saranno assemblati con una sovrapposizione di pochi millimetri sia nella coordinata rϕ che in z. Nella parte degli *endcap* le *strip* sono disposte radialmente, perpendicolarmente alla direzione del fascio, per ottenere una ottima misura di ϕ.

Questa complessa struttura spaziale del tracciatore consiste di circa 17000 moduli. Generalmente i moduli sono fatti a singola faccia (*single side*), ma alcuni sono costituiti da moduli a doppia faccia (*double side*). I moduli a doppia faccia sono moduli a singola faccia in configurazione "*back to back*", dove uno dei due strati (quello stereo) è ruotato nel piano del modulo. In particolare sono moduli *double side* quelli dei primi due strati del TIB e TOB, i primi due anelli del TID e gli anelli 1, 2, 5 del TEC.

Nelle figure 1.7 e 1.8 sono mostrate le strutture di supporto dei moduli del TIB e del TEC.

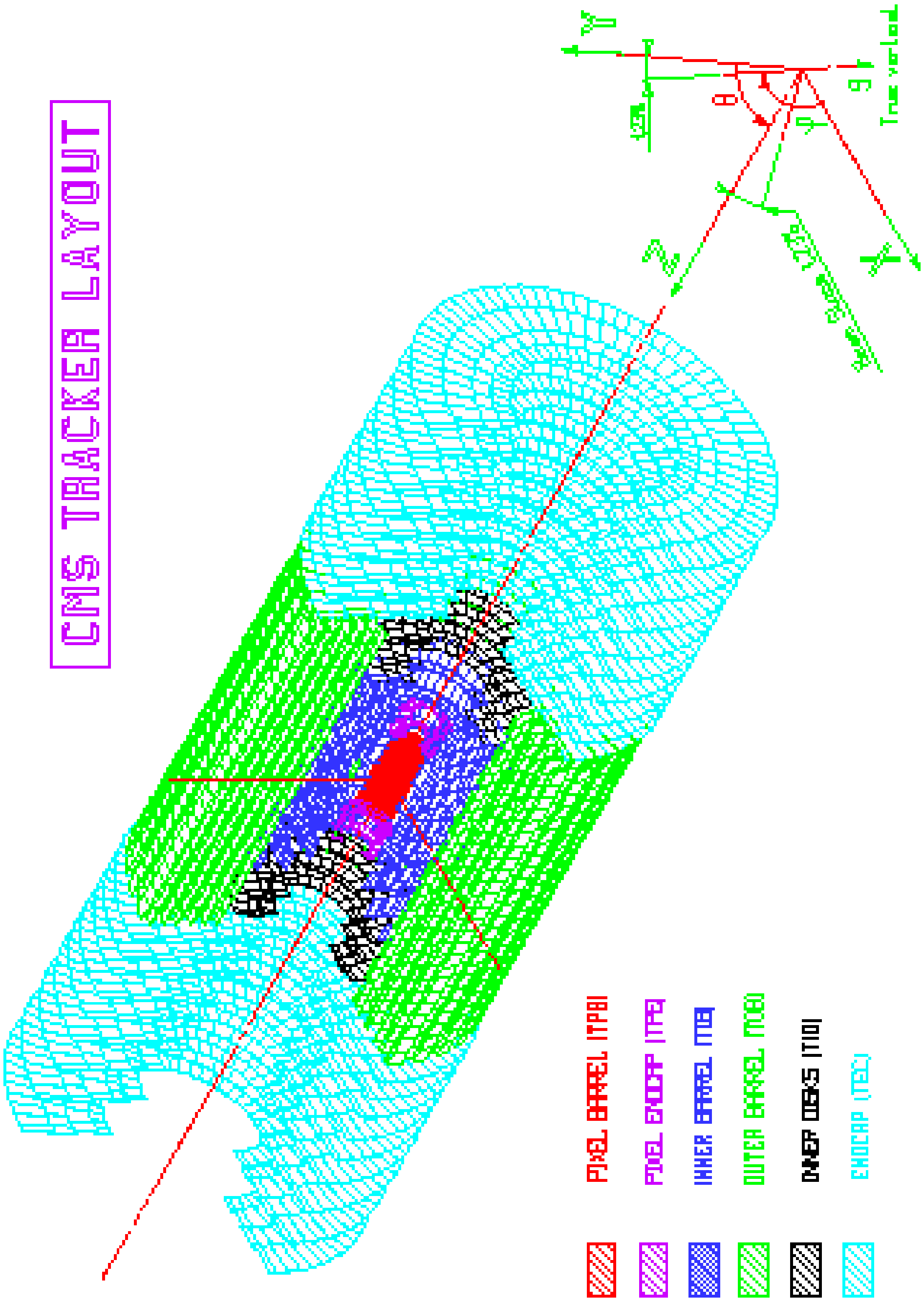

Figura 1.6: Spaccato tridimensionale del Tracker di CMS.

Figura 1.7: Struttura di supporto dei moduli del TIB.

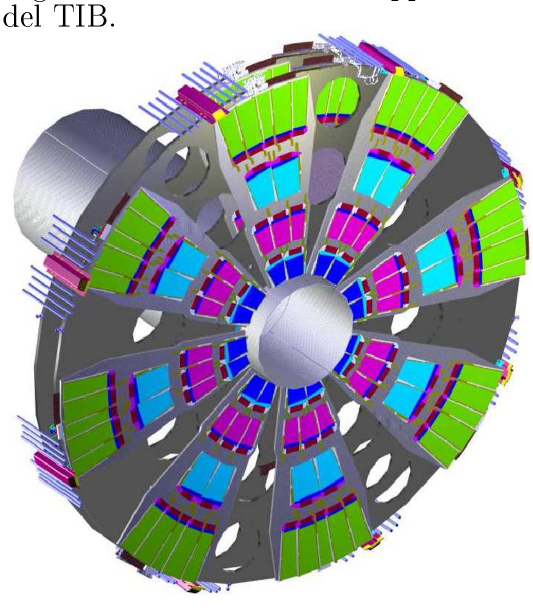

Figura 1.8: Struttura di supporto dei moduli del TEC.

Capitolo 2

Il progetto software di CMS

2.1 Struttura del software

Le risorse necessarie per il calcolo di CMS saranno maggiori e la struttura software sarà più complessa di quelli degli attuali esperimenti di fisica delle alte energie, non solo per la complessità del rivelatore e della fisica che si vuole studiare, ma anche per la dimensione della collaborazione e il lungo tempo di vita previsto per questo apparato sperimentale. Quindi il software deve essere sviluppato tenendo presente non solo le prestazioni, ma anche che sia modulare, flessibile e facile da mantenere.

La programmazione *Object Oriented* (OO), di cui si parlerà più approfonditamente nel prossimo capitolo, è stata identificata come la metodologia adatta per affrontare questo tipo di problemi.

La struttura dell'architettura software di CMS è stata scelta per rispondere alle seguenti esigenze [5]:

- ambienti multipli: vari moduli software devono essere in grado di girare in una varietà di ambienti;

- migrazione tra gli ambienti: un modulo software può essere sviluppato in un certo ambiente, ma in seguito venire usato in altri ambienti non previsti precedentemente;

- sviluppo distribuito del codice: il software viene sviluppato da gruppi di persone sparse geograficamente, spesso da programmatori non professionisti;

- flessibilità: le richieste a cui deve soddisfare il sistema software in sviluppo non si conoscono a priori, pertanto quest'ultimo deve essere facilmente adattabile;

- semplice uso: il sistema software deve essere facile da usare da parte della comunità dei fisici che spesso non è esperta di software.

Per soddisfare queste esigenze il software di CMS è stato strutturato nel seguente modo:

- un *framework*[1] personalizzabile per ogni ambiente di calcolo;
- moduli software riguardanti la fisica con interfacce ben definite;
- un *toolkit*[2] di programmi di utilità e servizio che possono essere usati da ognuno dei moduli di fisica.

Le componenti fondamentali del software di CMS e le loro relazioni sono mostrate in figura 2.1.

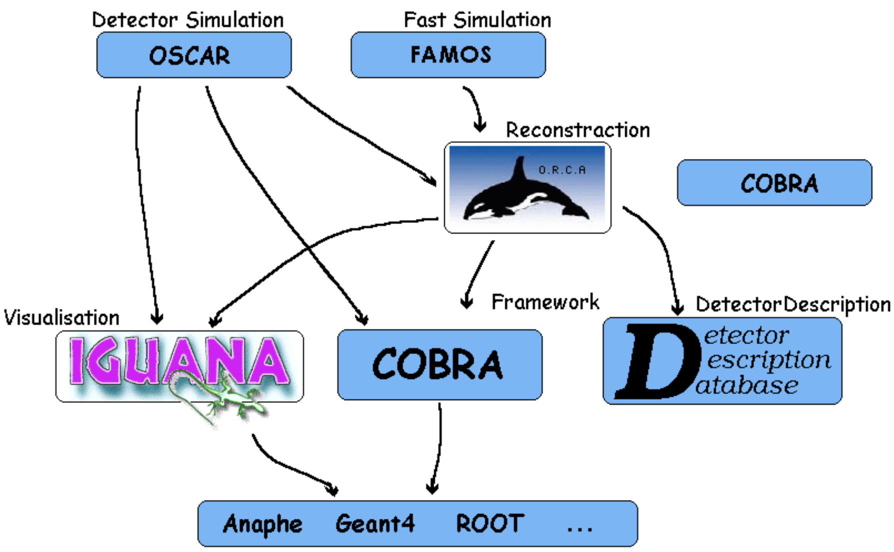

Figura 2.1: Le componenti fondamentali del software di CMS e le loro relazioni.

COBRA (*Coherent Object-oriented Base for simulation Reconstraction and Analysis*) è il *framework* per la simulazione, ricostruzione, visualizzazione ed analisi interamente sviluppato in OO/C^{++} ed ha i seguenti compiti:

[1]Un *framework* è un sistema che realizza un'architettura; è essenzialmente un codice che invoca delle procedure (*classi*) che lavorano secondo un preciso ordine. Il suo compito è quello di un "manager" che delega le responsabilità e mantiene il controllo del flusso dei dati.

[2]Un *toolkit* è un insieme di generiche procedure (*classi*) che possono essere invocate per sviluppare nuove applicazioni.

- rendere disponibili in maniera efficiente i moduli richiesti per la simulazione, ricostruzione e analisi;
- nascondere la complessità del sistema (accesso al *database*, etc);
- mettere a disposizione gli strumenti per la gestione dei dati;
- permettere di condividere delle parti di codice (*re-use*).

ORCA (*Object-oriented Reconstraction for CMS Analysis*)[6] è un'applicazione di COBRA che analizza gli eventi acquisiti dal rivelatore ricostruendo gli *hit* e alle tracce. I dati degli eventi simulati o acquisiti sono richiesti a COBRA che provvede ad accedere al *database*.

OSCAR (*Object-oriented Simulation for CMS Analysis and Reconstraction* è un'applicazione di COBRA che genera eventi simulati ("di Montecarlo") usando una dettagliata descrizione del rivelatore. Gli eventi generati devono essere il più possibile identici a quelli reali proprio perché sono usati per testare ORCA e altro software e per studi dettagliati sul comportamento del detector, sulla fisica, etc.

FAMOS (*FAst MOnte-Carlo Simulation*) genera eventi simulati con un algoritmo veloce che ignora la struttura dettagliata del rivelatore. Gli eventi generati vengono poi usati per studi approssimati.

Il software *off-line* di CMS, cioé quello per la ricostruzione, simulazione, analisi e visualizzazione, ha bisogno dei dati relativi al rivelatore. Da questo punto di vista ogni programma ha le sue esigenze specifiche: ad esempio il software per la simulazione (OSCAR) ha bisogno di una descrizione altamente granulare di tutte le parti sensibili e non del rivelatore, mentre il software di ricostruzione (ORCA) per la ricostruzione delle tracce necessita solo della posizione e delle dimensioni degli elementi sensibili del tracciatore. Per raggiungere questo in maniera coerente e consistente ogni applicazione deve consultare una sorgente comune di informazioni ovvero un database contenente la descrizione del rivelatore, questa sorgente è il DDD (*Detector Description Database*).

IGUANA (*Interactive Graphical User ANAlysis*) è il software di analisi grafica interattiva di CMS che fornisce le librerie e le *classi* per lo sviluppo del pacchetto di visualizzazione all'interno di ORCA. Di questo si parlerà più approfonditamente nel prossimo paragrafo.

2.2 La visualizzazione con IGUANA

Il progetto IGUANA [7] [8], iniziato nel maggio 1999, come parte integrante del software *off-line* di CMS, ha l'obiettivo di fornire un *framework* di visualizzazione basato su *plugin*[3] e una serie di strumenti per la grafica interattiva in 2D e 3D, con particolare riguardo verso l'*event display* e la visualizzazione interattiva del rivelatore simulato.

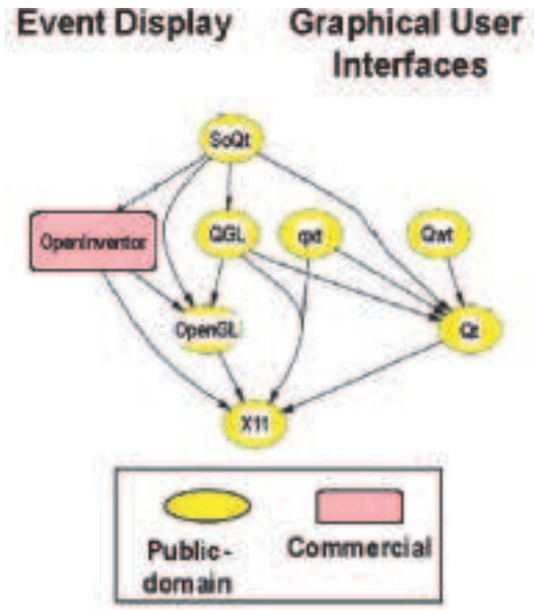

Figura 2.2: I pacchetti software per la realizzazione dell'*event display* in IGUANA e le loro interdipendenze. La freccia che collega un pacchetto a un altro denota una dipendenza del primo dal secondo, cioè il primo pacchetto non può essere usato in modo isolato dal secondo.

IGUANA è principalmente sviluppato tenendo conto delle necessità dell'esperimento CMS, ma risulta indipendente da esso, nel senso che può essere usato anche da altri esperimenti visto che sviluppa i seguenti tre campi principali:

1. un'interfaccia grafica per l'utente (GUI);
2. una visualizzazione interattiva del rivelatore e degli eventi;

[3]Molte applicazioni scritte in C o C^{++} utilizzano l'architettura a *plugin* permettendo agli sviluppatori di aggiungere, in maniera semplice e dinamica, nuove funzionalità. A differenza delle applicazioni monolitiche, dove tutte le funzionalità dell'applicazione sono racchiuse all'interno del singolo eseguibile, le applicazioni modulari hanno un "motore" centrale e un insieme di librerie complementari. Ogni libreria (chiamata di solito *plugin* o modulo) implementa un'unica funzionalità, e quando la funzionalità è necessaria, il "motore" carica su richiesta il *plugin* e lo esegue. In una applicazione con architettura a *plugin*, ogni *plugin* in genere aderisce all'interfaccia adottata, e tutti i *plugin* addizionali possono essere aggiunti in qualsiasi momento fornendo così all'applicazione un'estensibilità ad *hoc*.

3. una presentazione e analisi interattiva dei dati.

Il software di IGUANA non si presenta come un programma monolitico, piuttosto come un "*toolkit* modulare" in C^{++}. Gli sviluppatori scelgono dal *toolkit* solo quelle parti che sono rilevanti per il loro caso specifico. A tale fine IGUANA prevede l'uso di un numero consistente di pacchetti software già esistenti. In figura 2.2 sono mostrati i pacchetti software che servono per la realizzazione dell'*event display*, in particolare pacchetti e librerie di grafica in 2D e 3D commerciali e non (Qt [9], SoQt [10], OpenInventor [11], OpenGL, ...). Questi pacchetti sono stati utilizzati per sviluppare buona parte del codice del prototipo di cui si parla in questa tesi.

Il software IGUANA viene utilizzato per la visualizzazione dei rivelatori di CMS, ed in particolare:

- per una verifica della geometria di simulazione e ricostruzione, cioé la visualizzazione del database di descrizione del rivelatore (DDD);
- per il *display* degli eventi (anche in ambiente di *test beam*);
- per il monitoraggio durante la fase di costruzione;
- per la verifica del corretto funzionamento del rivelatore durante la fase di installazione e di *commissioning*[4].

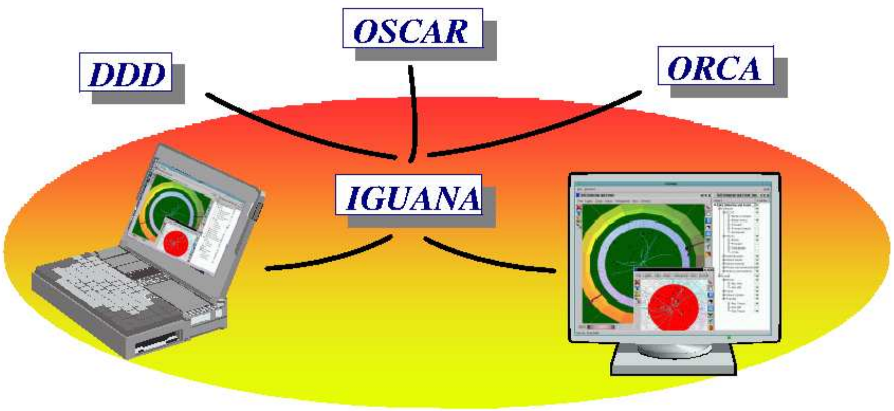

Figura 2.3: La visualizzazione con IGUANA.

Nell'ottobre 2002 è uscita la release 4.0.0 di IGUANA [12] che include: i programmi di visualizzazione in ORCA e in OSCAR inseriti come *plugin*;

[4]É la fase iniziale di costruzione del rivelatore.

un sistema di visualizzazione interattiva (DDD); e l'integrazione dei *browser* per la visualizzazione in 2D e 3D. In pratica ORCA e OSCAR hanno al loro interno un sottosistema di visualizzazione realizzato con IGUANA. Il pacchetto di visualizzazione in OSCAR è il programma che serve per il controllo della descrizione del rivelatore in termini di geometria e di materiali, mentre quello in ORCA è il programma per il *display* dei dati di simulazione e ricostruzione di CMS. In entrambi i casi il rivelatore e gli eventi sono visualizzati in 3D come *scena grafica* [1] di OpenInventor.

Questo lavoro di tesi si propone proprio lo sviluppo della visualizzazione in 2D all'interno del pacchetto di visualizzazione in ORCA.

2.3 Oggetti grafici 2D per la visualizzazione del Tracker e degli eventi

Per ampliare le funzionalità del software di visualizzazione sotto ORCA furono proposte [1] in un lavoro di tesi precedente a quasto, delle nuove rappresentazioni grafiche 2D di selezione e visualizzazione del *Tracker* che richiamo brevemente:

1. Una *finestra* di selezione/deselezione (fig.2.4) che permette una rapida e semplice selezione delle parti (anello, disco, cilindro, $\cdots$) del tracciatore di CMS da far vedere nella finestra della *scena grafica* di OpenInventor (finestra 3D). Sostanzialmente è una mappa 2D della metà superiore

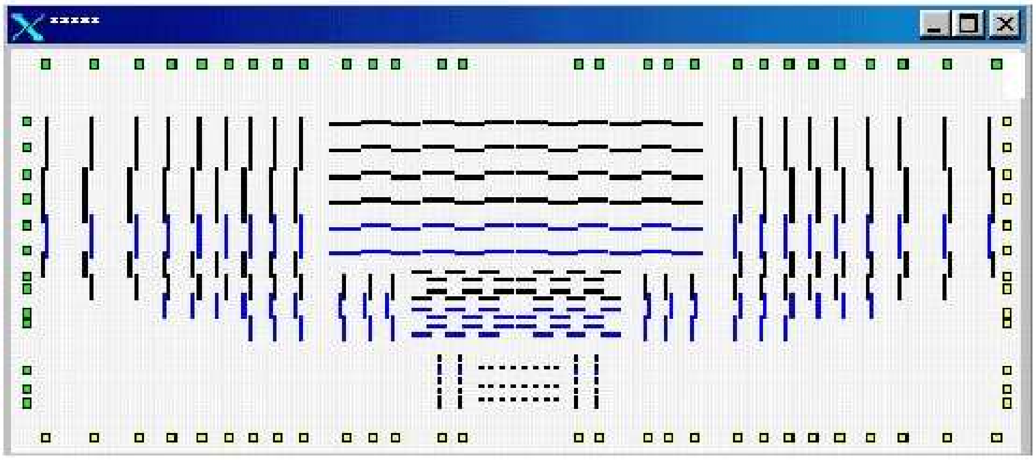

Figura 2.4: Finestra di selezione/deselezione di anelli, *rod*, petali e *layer* del sistema tracciante di CMS.

del rivelatore. Ogni barretta permette di selezionare l'intero anello di un disco dell'*endcap* o di un *layer* del *barrel*. Le barrette colorate in blu rappresentano gli anelli costituiti da moduli doppi. L'*oggetto* permette anche la selezione di un intero disco dell'*endcap* cliccando sui quadratini di colore giallo in basso e la selezione di un intero strato del *barrel* cliccando sui quadratini in giallo a destra. Infatti i quadratini gialli posizionati sotto e a destra della figura sono disposti lungo la direzione dei dischi e degli strati cilindrici. Invece i quadratini verdi posti in alto e a sinistra, sempre corrispondenti ai dischi e agli strati cilindrici, servono a far apparire una nuova finestra che conterrà l'*oggetto* che permette di selezionare il singolo modulo (vedere punto 2).

2. Una *finestra* di selezione/deselezione (fig.2.5) che permette la selezione di ogni singolo modulo di ognuno dei 41 cilindri o dischi del rivelatore da far vedere nella finestra 3D; la colorazione in blu rappresenta i moduli *stereo*. L'*oggetto* rappresentativo del *barrel*, figura 2.5 *b*, è ottenuto tagliando il cilindro lungo un asse parallelo all'asse di simmetria (ϕ=0) e aprendolo su di un piano. Ogni colonna corrisponde perciò ad un anello e i moduli in blu sono *stereo*.

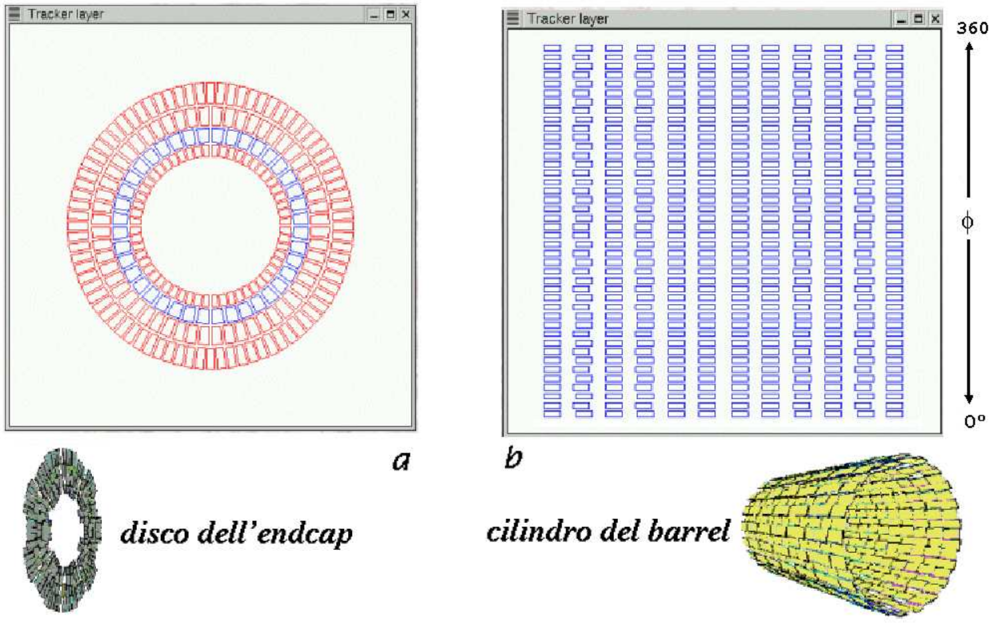

Figura 2.5: Finestra di selezione/deselezione di un singolo modulo di un layer del Tracker. La figura *a* mostra un disco dell'*endcap*, la *b* un cilindro del *barrel*.

3. Una *finestra* che permette di selezionare una regione θ/ϕ per poter risalire alla traccia di interesse con i suoi *hit* e i moduli da essa attraversati. Questo è realizzato, come mostra la figura 2.6, dalla distribuzione di *SimHit* utilizzando il metodo di rappresentazione del V-Plot[5] di H.Drevermann.

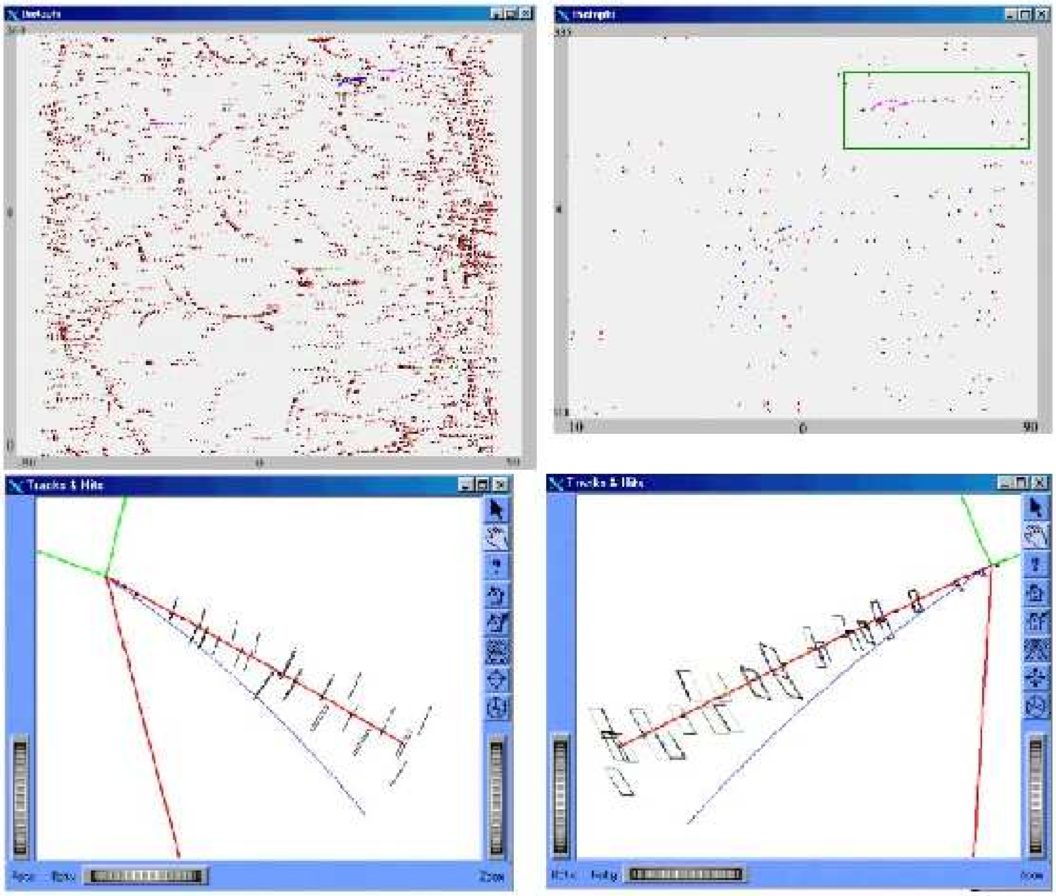

Figura 2.6: Finestra di selezione di una zona θ/ϕ.

4. Un *oggetto grafico* rappresentativo dell'intero sistema tracciante di CMS. Si tratta proprio di una mappa 2D del *Tracker* (fig. 2.7) in cui si possono rappresentare le informazioni di ogni sensore in maniera codificata

[5]Secondo il metodo di rappresentazione *V-Plot* per ogni *hit* se ne disegnano due, uno dei due *hit* viene visualizzato nella posizione nominale, il secondo *hit* in una posizione la cui coordinata θ differisce dalla posizione nominale di una quantità proporzionale alla distanza dell'*hit* dal bordo del *Tracker*. Pertanto ogni traccia sarà visibile come una specie di V più o meno deformata, con la punta diretta verso il punto dove la traccia esce dal *Tracker*.

ed anche in maggior dettaglio utilizzando per esempio uno zoom. La mappa non è fedele alla realtà dato che la parte del rivelatore a *pixel* è ingrandita e meno distante rispetto al rivelatore a microstrisce al silicio.

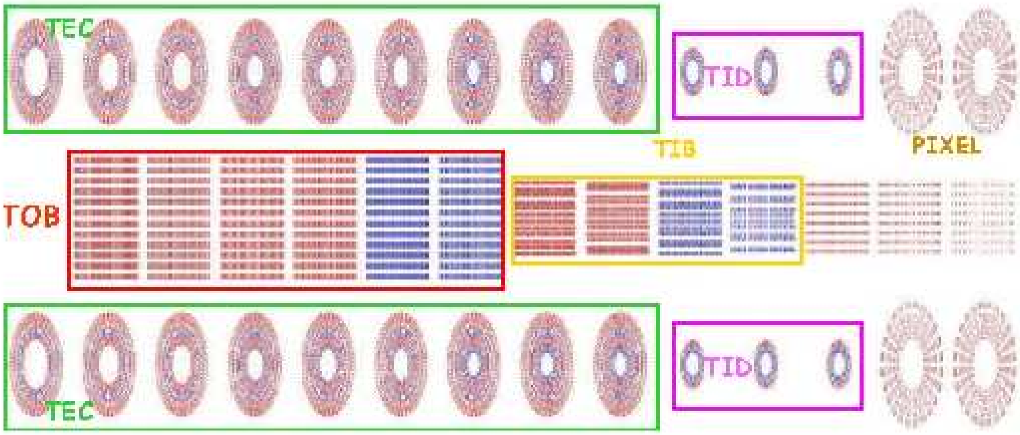

Figura 2.7: *Oggetto grafico* che rappresenta l'intera mappa del *Tracker*. Sono state contornate le diverse sottosezioni di cui è costituito il rivelatore per semplificarne l'identificazione delle varie parti.

Fu anche realizzato un software dimostrativo (demo)[1] per la versione 6.2.3 di ORCA, per testare la funzionalità degli *oggetti grafici* 2D di visualizzazione propisti. In particolare furono realizzati i primi due elementi di selezione/deselezione delle componenti del *Tracker*. L'accesso a questi due *oggetti grafici* è stato fatto aggiungendo un nuovo ramo denominato *CustomTracker* sotto il ramo *Detector* dell'albero di controllo della *scena* di IGUANA (fig.2.8). Selezionando sotto *CustomTracker* la voce *TrackerSelection* appare una nuova finestra 2D con il primo *oggetto grafico*. Poi se l'utente clicca su uno dei quadratino verdi presenti in questa finestra, compare una nuova finestra 2D (fig.2.8), contenente il secondo *oggetto grafico* descritto.

L'implementazione della demo ha dimostrato che l'approccio proposto di usare delle rappresentazioni grafiche 2D per rendere più accessibile la grafica 3D del tracciatore è corretta. Inoltre l'uso di tali finestre di selezione si è dimostrato facile ed intuitivo per l'utente che non ha bisogno di conoscere la geometria del tracciatore per servirsene.

Quindi in questo lavoro di tesi si è proceduto alla reingegnerizzazione del software di visualizzazione, cosa che ha richiesto di ripartire dalla progettazione degli *oggetti grafici* proposti per la versione 7_2_0 di ORCA iniziandone

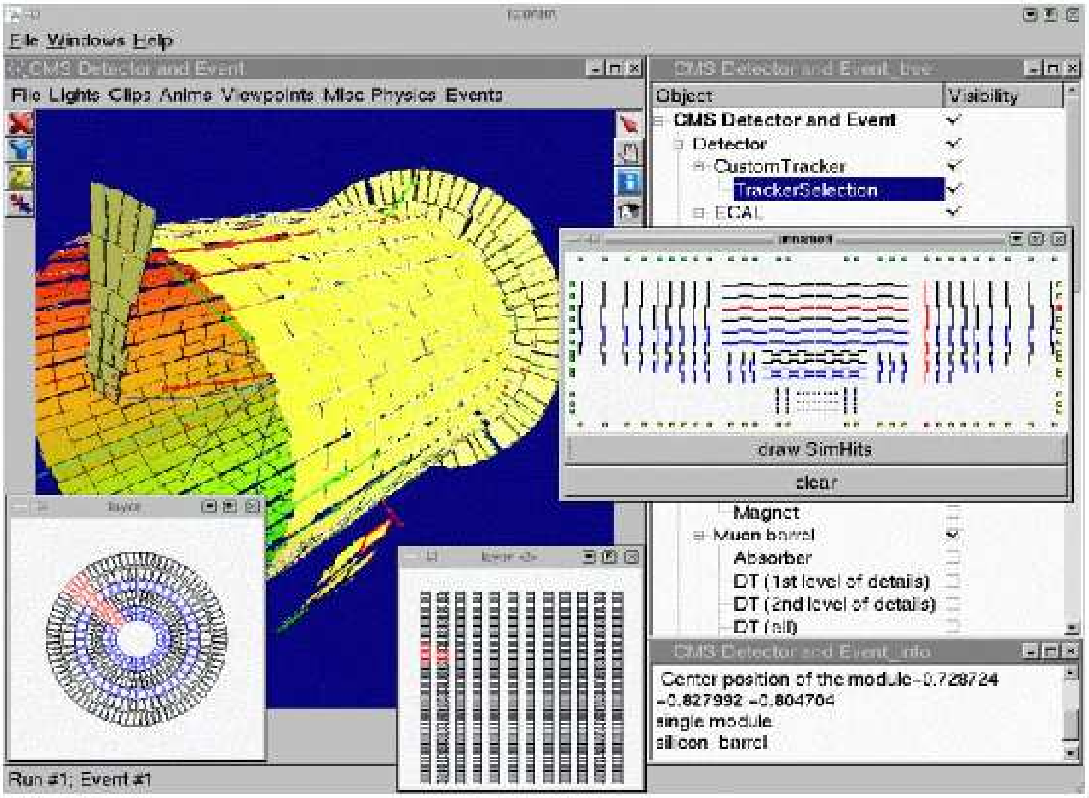

Figura 2.8: *Screenshot* della demo che mostra il primo e il secondo *oggetto grafico*. Cliccando su un quadratino verde della prima finestra di selezione/deselezione (a destra), si apre una nuova finestra (a sinistra) per la selezione dei singoli moduli che devono essere mostrati nella finestra 3D. Si noti che se si seleziona un modulo nella finestra 3D, allora nell'apposita finestra delle informazioni di IGUANA (in basso a destra) vengono stampate le informazioni relative a quel modulo.

l'implementazione e poi in particolare sono state studiate le possibile rappresentazioni dell'intero sistema tracciante di CMS per il quarto *oggetto grafico* descritto.

Capitolo 3

Tecniche di programmazione

Per la complessità del *Tracker* di CMS, con migliaia di rivelatori, milioni di canali di lettura e decine di centinaia di *hit* per *bunch crossing*, è necessario uno strumento software altrettanto complesso per ottenere la visualizzazione in 3D e 2D del *Tracker* e degli *hit*. Come detto nel paragrafo 2.1, il software oltre ad essere complesso deve essere modulare, flessibile e facile da modificare, questi obiettivi possono essere raggiunti con una programmazione *Object Oriented* (OO) ed in particolare usando le tecniche di: *incapsulazione*, *ereditarietà* e *polimorfismo* [13].

3.1 Programmazione orientata agli oggetti

Un programma OO è costituito da *oggetti* che sono instanziati da *classi*. Una *classe* è definita dal suo nome, dal suo stato e dal suo comportamento. Una rappresentazione grafica di una *classe* è mostrata in figura 3.1 (questo tipo di rappresentazione di *classe* è usata nella modellizzazione UML par.3.2.1).

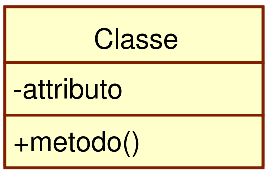

Figura 3.1: Rappresentazione grafica della *classe* in cui si specifica il nome della *classe* con la lista degli *attributi* e dei *metodi*.

Gli *attributi* di una *classe* sono i suoi "*dati membro*" che definiscono il suo stato, i *metodi* sono le operazioni che la *classe* può eseguire sugli *attributi*, cioé

definiscono il comportamento della *classe*. L'insieme del nome, dello stato e del comportamento di un *oggetto* è detto *incapsulazione*, cioé ogni *oggetto* contiene tutte le informazioni necessarie ed è pienamente descritto da queste tre caratteristiche. Questa è la distinzione principale tra la programmazione OO e lo stile di programmazione *procedurale*, dove dati e funzioni sono separati. L'*incapsulazione* è un'importante principio che permette a un software OO di avere una struttura modulare.
Un esempio concreto di *classe* potrebbe essere quello della *classe* `Modulo` (fig.3.2) che descrive l'*oggetto* modulo rivelatore con i sui *attributi* `forma` e `posizione` e con il *metodo* `rotazione()`.

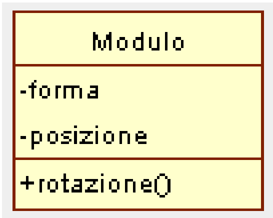

Figura 3.2: Esempio della *classe* `Modulo` che ha come *attributi* `forma` e `posizione` e come *metodo* `rotazione()`.

Un'altro importante concetto della programmazione OO è l'*ereditarietà*. Le *classi* già esistenti possono essere estese con particolari caratteristiche, *attributi* o *metodi*, che specializzano la *classe* esistente. In pratica l'*ereditarietà* è una relazione tra *classi* che quindi può dare origine a una *generalizzazione* o *specializzazione*.
Un esempio di *specializzazione* (fig.3.3) è quello delle *classi* `ModuloPixel` e `ModuloSStrip` che ereditano dalla *classe* generale `Modulo` le caratteristiche base specializzandosi poi in modulo rivelatore di tipo *pixel* e di tipo *silicon strip*.

Infine il *polimorfismo* è il concetto che permette agli *oggetti* di reagire in modo differente ad uno stesso messaggio (chiamata di un *metodo*). Questo significa che il *metodo* che sarà chiamato da un certo messaggio sarà deciso solo al tempo di esecuzione del programma OO.
In pratica, nel caso delle *classi* `ModuloPixel` e `ModuloSStrip`, quando il programma richiama il *metodo* `rotazione()`, al tempo di esecuzione il compilatore richiamerà proprio il *metodo* `rotazione()` dell'*oggetto* per il quale era stato chiamato.

Un altro strumento fondamentale è quello di *classe astratta* che corrisponde ad un concetto astratto. Ad esempio, la stessa *classe* `Modulo` potrebbe essere definita come *classe astratta* che definisce i *metodi* e gli *attributi* che le

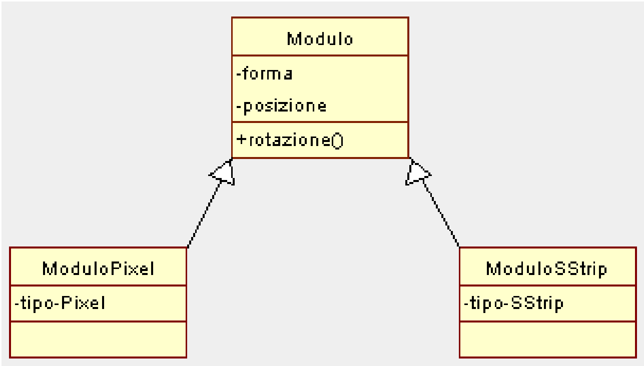

Figura 3.3: Esempio di *ereditarietà*. Le *classi* `ModuloPixel` e `ModuloSStrip` ereditano le caratteristiche (*attributi* e *metodi*) della *classe* base `Modulo` ed in più hanno l'*attributo* del tipo.

classi che la ereditano devono avere. Come conseguenza di ciò nel programma non si potrà creare l'*oggetto* `Modulo`, ma solo gli *oggetti* `ModuloPixel` e `ModuloSStrip`. A livello di programmazione, una *classe astratta* rappresenta una *classe* che contiene al suo interno una o più *funzioni membro virtuali pure*, cioè per esempio la funzione `rotazione()` di `Modulo` che le due *classi figlie* `ModuloPixel` e `ModuloSStrip` devono ridefinire.

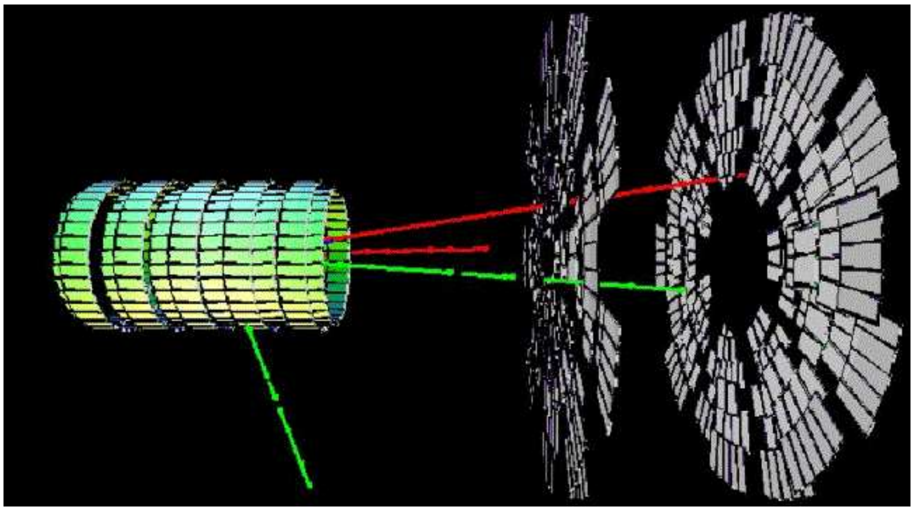

Figura 3.4: Schermata del programma di visualizzazione in tre dimensionei del tracciatore.

Nella figura 3.4 tratta dal software di visualizzazione di CMS sono mostrati un certo numero di *moduli* del sistema tracciante attraversati da tracce delle quali si possono vedere gli *hit*, il tutto nello spazio tridimensionale. Nella figura ci sono diversi *moduli*: ognuno di questi è un *oggetto* della *classe* `ModuloPixel` oppure `ModuloSStrip`. La *classe* definisce il *comportamento*, cioè che cosa gli *oggetti* di questa *classe* possono fare (l'*interfaccia*) e come (l'*implementazione*). I vari *oggetti* differiscono per il loro stato: i vari *moduli* sono differenti e ognuno ha la sua posizione, appartiene a un determinato *layer* etc. Queste informazioni sono *incapsulate* nell'*oggetto* come è *incapsulata* l'implementazione di ogni servizio. L'utente parla con l'*oggetto* attraverso un'*interfaccia*: come è rappresentato internamente lo stato dell'*oggetto* e come è implementato ogni servizio è nascosto e può essere quindi modificato senza dover modificare anche il codice dell'utente. Infine i vari *moduli* e le varie *tracce* hanno un'*interfaccia* comune, possono errese ruotati, zoommati, selezionati etc, ma ognuno svolge quest'attività a modo suo. Questo è il *polimorfismo* che è implementato attraverso la relazione di *ereditarietà*.

Questa è l'essenza della programmazione ad oggetti.

3.1.1 Programmazione generica e STL

Nella programmazione OO l'obiettivo primario è la riusabilità, quindi servono strumenti che permettano una generalizzazione. Parte integrante della programmazione a oggetti è la tecnica chiamata della *Programmazione Generica*, realizzata in C^{++} mediante l'uso di *Template* che permettono di scrivere programmi che possono lavorare con diversi tipi di dati. Un *modello* (*template*) è una formula astratta per produrre codice concreto, sia per funzioni che per *classi*. Il medesimo *modello* può poi essere utilizzato per generare occorrenze diverse; questo si ottiene mediante parametri del *modello*.

Il C^{++} oltre a fornire i costrutti sintattici per realizzare questi *template* fornisce con la STL[1] (*Standard Template Library*) [14] [13] *classi template* di utilizzo generale. La libreria STL è un progetto nato nel 1992 dal lavoro di Alex Stepanov e Meng Lee dei laboratori Hewlett-Packard, che si propone di standardizzare l'utilizzo di alcuni tipi di dati e funzioni comunemente usati. La libreria in questione fa un largo utilizzo del meccanismo dei *template* (da cui nasce il nome), in pratica fornisce *classi template* di utilizzo generale, funzioni che implementano algoritmi e strutture dati di uso comune comprendendo, ad esempio, il supporto di *vettori*, *liste*, *code*, *stack* etc, inoltre definisce varie *routine* di accesso a tali elementi. Poiché la libreria STL si basa su *classi template*, gli algoritmi e le strutture dati possono essere applicate praticamente ad ogni tipo di dati. Cioé la STL fornisce vari tipi

[1] La STL è stata adottata nel luglio 1994 come parte della libreria standard dall'ANSI/ISO C^{++} Standards Committee.

di *contenitori*[2] realizzati con *template*. Per navigare nei *contenitori* vengono usati degli speciali *oggetti* detti *iteratori* che permettono di scrivere algoritmi indipendenti dal particolare *contenitore* usato. La STL stessa contiene parecchi di questi algoritmi generici che funzionano con vari tipi di dati.

Il *contenitore* della STL più usato ed efficiente è il *vector*. Un *vector* è una sequenza (struttura dati con elementi memorizzati in ordine sequenziale) che consente di accedere in modo casuale alle componenti che possono essere sia *oggetti* che *dati primitivi* di qualsiasi tipo. Il tempo di inserimento e rimozione di elementi è costante alla fine del vettore, mentre è lineare se l'elemento deve essere inserito all'inizio o nel mezzo. Nella progettazione del prototipo per la visualizzazione del *Tracker* delle tracce e degli *hit* il *contenitore vector* è stato largamente usato.

Un altro elemento che è stato usato è *set* che è un contenitore associativo ordinato, infatti fornisce la possibilità di accedere velocemente ai dati in base al valore di chiavi (un valore unico associato ad ogni elemento) e in un contenitore di tipo *set* non possono coesistere due elementi uguali.

3.2 Analisi e disegno orientati agli oggetti

Il paradigma *Object Oriented* richiede tecniche di *analisys* e *design* adeguate, in particolare per grandi progetti software sono necessari appositi strumenti di modellizzazione per documentare lo sviluppo del progetto e per facilitare anche la comunicazione all'interno del *team* di lavoro. L'UML è una modellizzazione sviluppata recentemente nello sforzo di produrre un'unica metodologia di riferimento per l'analisi e il progetto OO. Lo scopo è quello di avere un'unica metodologia che descriva in modo completo tutti gli aspetti della modellizzazione agli *oggetti*, permettendo agli sviluppatori di avere un unico strumento attraverso il quale possano seguire l'intero ciclo dello sviluppo OO.

Nella programmazione ad *oggetti* frequentemente si presentano gli stessi problemi, questo ha portato gli sviluppatori di software a definire delle soluzioni standard precedentemente usate e ottimizzate per quello specifico problema. Questi particolari tipi di problemi con la rispettiva risoluzione sono detti *pattern*.

Alcuni diagrammi UML e *pattern* sono stati utilizzati per lo sviluppo del prototipo di visualizzazione in 2D del *Tracker*, per questo ora vediamo in dettaglio cosa sono. In particolare noi useremo il *pattern Model View Control* (MVC) che è alla base di tutte le moderne interfacce grafiche e alcuni dei cosiddetti *design pattern* di uso comune nella programmazione ad oggetti.

[2]Un *contenitore* è un *oggetto* formato da un insieme di altri *oggetti* come un *array* o un *vettore* o una *lista*. Il costrutto *template* permette di realizzare *contenitori* non solo di *oggetti* ma anche di *dati primitivi*.

3.2.1 La modellizzazione UML

L'*Unified Modelling Language* (UML)[15][16] è un linguaggio di modellizzazione visuale per analizzare, specificare, visualizzare, costruire e documentare lo sviluppo di prodotti di sistemi software e tutte le altre realtà non necessariamente legate al mondo dell'informatica. L'UML è una collezione dei migliori aspetti ereditati da diversi linguaggi di modellizzazione fino ad ora implementati (Booch, OMT, OOSE/Objectory)[3].

Una buona modellizzazione è la base per lo sviluppo di buoni e robusti progetti *Object Oriented*. Buoni modelli sono essenziali per la comunicazione fra i diversi gruppi di sviluppo e per controllare la correttezza architetturale del sistema. La costruzione dei modelli di sistemi complessi ha lo scopo di facilitare la comprensione di molti aspetti dei sistemi stessi: maggiore è la complessità del sistema e più alta è la necessità di una buona modellizzazione.

L'UML il linguaggio di progettazione più diffuso perché:

- è indipendente da qualsiasi linguaggio di programmazione e da qualsiasi processo di sviluppo,
- fornisce un formalismo base per comprendere i modelli del linguaggio;
- incoraggia la crescita di strumenti *Object Oriented*;
- supporta diversi livelli di sviluppo;
- integra le migliori tecniche dei linguaggi di modellizzazione;

Uno degli aspetti vantaggiosi di UML è il formalismo semplice che viene utilizzato e che comprende 9 tipi di diagrammi divisi in 5 categorie. La tabella seguente mostra le categorie e i diagrammi corrispondenti:

[3] Nota storica: l'UML è nato dall'unificazione di diversi metodi di modellizzazione come:

- OOSE:Object Oriented Software Engineering,
- OMT: Object Modelling Technique,
- Booch,
- ed altri ancora.

La prima versione ufficiale di UML risale al gennaio 1997 (versione 1.0), attualmente si è arrivati alla versione 2.0.

Categoria	Diagrammi
Diagrammi introduttivi	Diagrammi dei casi d'uso (use case)
Diagrammi di struttura statica	Diagrammi delle classi (class)
	Diagrammi degli oggetti (object)
Diagrammi d'interazione	Diagrammi di sequenza (sequence)
	Diagrammi di collaborazione (collaboration)
Diagrammi di stato	Diagrammi di stato (statechart)
	Diagrammi di attività (activity)
Diagrammi di implementazione	Diagrammi dei package
	Diagrammi dei componenti (component)
	Diagrammi di distribuzione (deployement)

Nell'ambito del lavoro di questa tesi sono stati usati il "*diagramma dei casi d'uso*", il "*diagramma delle classi*" e il "*diagramma dei pacchetti*" questo perché, dato il numero limitato di *classi* da progettare e il loro tipo, non è stato ritenuto necessario procedere ad altre più dettagliate descrizioni con altri diagrammi. Di seguito vengono descritti i tre tipi di diagrammi usati.

Il diagramma dei casi d'uso

Un *diagramma dei casi d'uso* [17] è un grafo che mostra le interazioni tra un utente e un "*caso d'uso*" di un sistema. Risulta particolarmente utile durante la fase di specifica dei requisiti del progetto. Gli *attori* sono i soggetti, esterni al sistema, che interagiscono con il sistema tramite messaggi (richieste, comunicazioni, risposte).
Il sistema è l'entità i cui utilizzi vengono descritti dall'insieme dei *casi d'uso*. Più precisamente, un insieme completo di *casi d'uso* descrive in modo completo gli utilizzi del sistema dall'esterno, ossia dal punto di vista degli *attori* che interagiscono con esso, senza rivelare la struttura interna del sistema. L'associazione tra *attori* e *casi d'uso* rappresenta la "comunicazione" che può essere bi-direzionale o uni-direzionale tra *attori* e *casi d'uso*. Ogni *attore* può comunicare con più *casi d'uso*.

Le associazioni ammesse sono tre:

1. *Generalizzazione/Specializzazione*
   Associa un *caso d'uso* di tipo generale ad uno o più *casi d'uso* specializzati. Ogni *caso d'uso* specializzato eredita le caratteristiche, i passi, gli eventuali punti di intersezione e le associazioni del *caso d'uso* generale. Il *caso d'uso* specializzato può aggiungere nuovi passi, oppure ridefinire

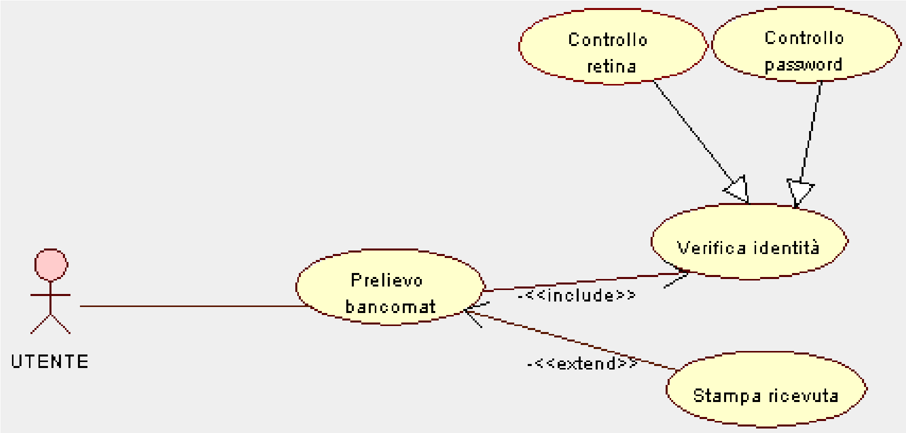

Figura 3.5: Struttura tipica di un *diagramma dei casi d'uso*. L'utente va al bancomat per un prelievo. Il *caso d'uso* "prelievo bancomat" «include» il *casi d'uso* "verifica identità" ed «estende» il *casi d'uso* "stampa ricevuta". Il *caso d'uso* "verifica identità" viene specializzato dai *casi d'uso* "controllo password" e "controllo retina".

i passi ereditati da quello generale (*override*). É rappresentata graficamente con una linea continua, e con una punta di freccia triangolare bianca. Nell'esempio di figura 4.1 il *caso d'uso* "verifica identità" viene specializzato dai *casi d'uso* "controllo password" e "controllo retina".

2. *Include*
   Casi d'uso diversi possono avere in comune una sequenza di passi da svolgere. In questo caso è possibile enucleare la sequenza comune, e definirla come un *caso d'uso* a sè stante, da "includere" nei *casi d'uso* originari. Così facendo si evidenziano le parti comuni, e si evitano le ripetizioni nelle descrizioni dei *casi d'uso*. Graficamente è rappresentato con una linea con la punta della freccia aperta e lo stereotipo *«include»*, la cui direzione va dal *caso d'uso* "includente" al *caso d'uso* "incluso". Nell'esempio di figura 4.1 il *caso d'uso* "prelievo bancomat" «include» il *caso d'uso* "verifica identità".

3. *Extend*
   L'associazione "*extend*" permette di definire che un *caso d'uso* base può venire esteso con il comportamento definito in un altro *caso d'uso*, detto di estensione. L'estensione riguarda un comportamento opzionale del *caso d'uso* base, ed è soggetta ad una condizione di attivazione.Il *caso d'uso* di estensione, inoltre, definisce la condizione per la propria attivazione. Se la condizione viene verificata, il comportamento del *caso*

d'uso di estensione verrà attivato nei punti di estensione opportuni; in caso contrario no. Graficamente è rappresentato con una linea con la punta della freccia aperta e lo stereotipo *«extend»*, la cui direzione va dal *caso d'uso* di "estensione" al *caso d'uso* "base". Nell'esempio di figura 4.1 il *caso d'uso* "prelievo bancomat" «estende» il *caso d'uso* "stampa ricevuta".

Il diagramma delle classi

Un diagramma delle *classi* [18] è un grafo che illustra le *classi* e gli *oggetti* di un sistema e le diverse relazioni tra di essi. Descrive:

- il nome della *classe*;
- gli *attributi* della *classe* specificandone il tipo, la visibilità e l'eventuale inizializzazione. La visibilità è indicata mediante i simboli + (*public*), # (*protected*), e - (*private*).
- le operazioni (*metodi*) contenute nella *classe* e le loro signature. La visibilità è indicata mediante i simboli + (*public*), # (*protected*), e - (*private*).
- in corsivo i nomi delle *classi astratte* e dei rispettivi *metodi astratti*.

Le relazioni tra le *classi* sono di tre tipi fondamentali:

1. *Associazione*
   Indica che le *classi* sono connesse l'una all'altra da un punto di vista concettuale. Graficamente, un'*associazione* viene rappresentata con una linea che connette le due *classi*. Per questo tipo di relazione si può indicare anche la *molteplicità*, cioé il numero di *oggetti* appartenenti ad una *classe* che interagisce con il numero di *oggetti* della *classe associata*. Come casi particolari di *associazione* si hanno:

   - *Aggregazione*
     Specifica un'*associazione* di tipo "parte/intero" in cui una *classe* che rappresenta l'"intero" è costituita da più *classi* che la compongono ("parte"). Graficamente viene rappresentata con una linea che unisce l'"intero" e il componente singolo tramite un rombo bianco disegnato dalla parte dell'intero.
   - *Composizione*
     É un tipo di *aggregazione* più forte, in cui ogni componente in una *composizione* può appartenere soltanto ad un "intero". Il simbolo

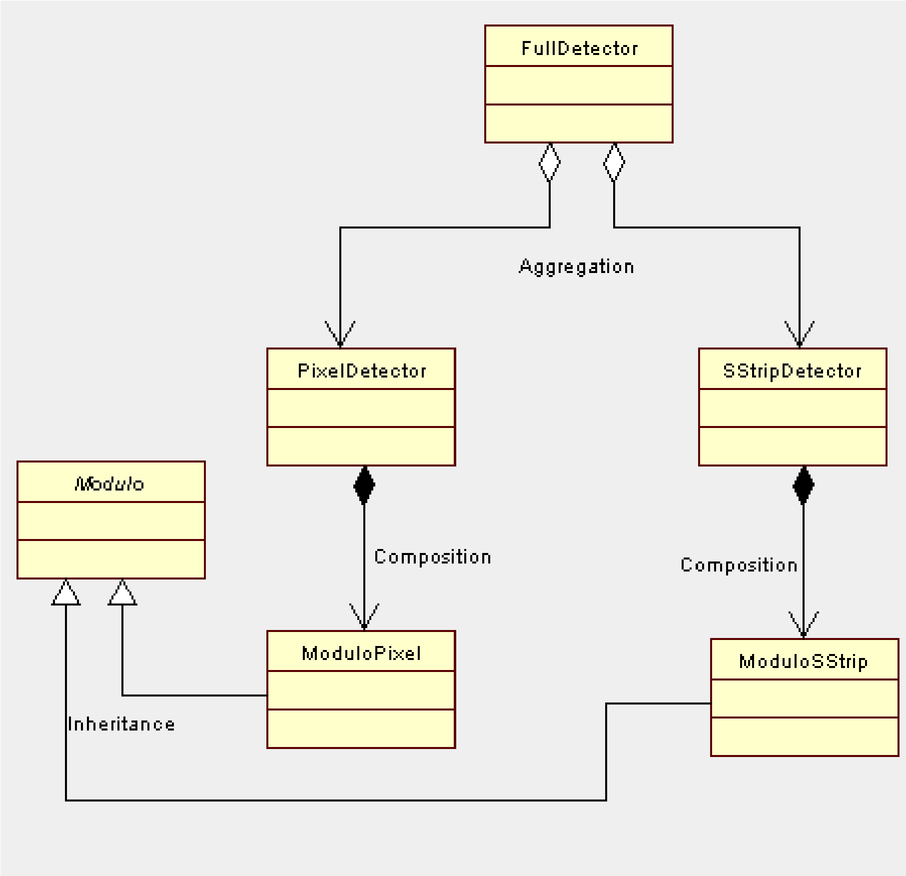

Figura 3.6: Esempio di una struttura tipica di un *diagramma delle classi*.

utilizzato per una *composizione* è lo stesso usato per un'aggregazione eccetto il fatto che il rombo è colorato di nero.

2. *Dipendenza*
   Tale relazione indica che il funzionamento di un elemento (componente) richiede la presenza di uno o più elementi, questo implica che se un elemento viene modificato, potrebbe essere necessario modificare anche ogni elemento che da esso dipende. É visualizzata nel modello da una linea tratteggiata con un triangolo aperto dalla parte del componente da cui l'altro dipende.

3. *Generalizzazione/ereditarietà*
   É una relazione che collega un elemento più generico ad un elemento più specifico, cioé descrive la relazione di *ereditarietà* tra le varie *classi* di

un progetto. Graficamente viene rappresentato con una linea continua, e con una punta di una freccia triangolare bianca.

I pacchetti (package diagram)

Il *package*, in UML, è un contenitore generico di altri elementi (come una cartella in un qualsiasi sistema operativo). Nell'ambito *Object Oriented* il *package* è una struttura utilizzata per organizzare logicamente gli elementi relazionati.

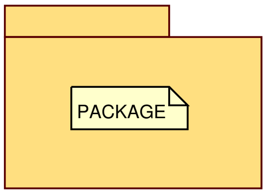

Figura 3.7: Struttura tipica di *package*.

3.2.2 L'architettura MVC

L'architettura software *Model View Control* (MVC) è alla base di IGUANA come di ogni moderna *graphical user interface*. L'obiettivo dell'MVC [19] è di disaccoppiare il più possibile tra loro le parti dell'applicazione adibite al controllo, all'accesso ai dati e alla presentazione.
Con il termine *Model* si individua la rappresentazione dei dati dell'applicazione e le regole con cui si accede e si modificano tali dati.
La *View* è la vista del modello. Uno stesso modello può quindi essere presentato secondo diverse viste.
Il *Controller* è colui che interpreta le richieste della *View* in azioni che vanno ad interagire con il *Model* aggiornando conseguentemente la *View* stessa.
Quest'approccio porta ad innegabili vantaggi come:

- indipendenza tra i dati (*Model*), la logica di presentazione (*View*) e quella di controllo (*Controller*);
- separazione dei ruoli e delle relative interfacce;
- viste diverse per il medesimo *Model*;

Nel caso del progetto software IGUANA, il *modello* è formato dai dati sottostanti da visualizzare, questo comporta la necessità di leggere e visualizzare:

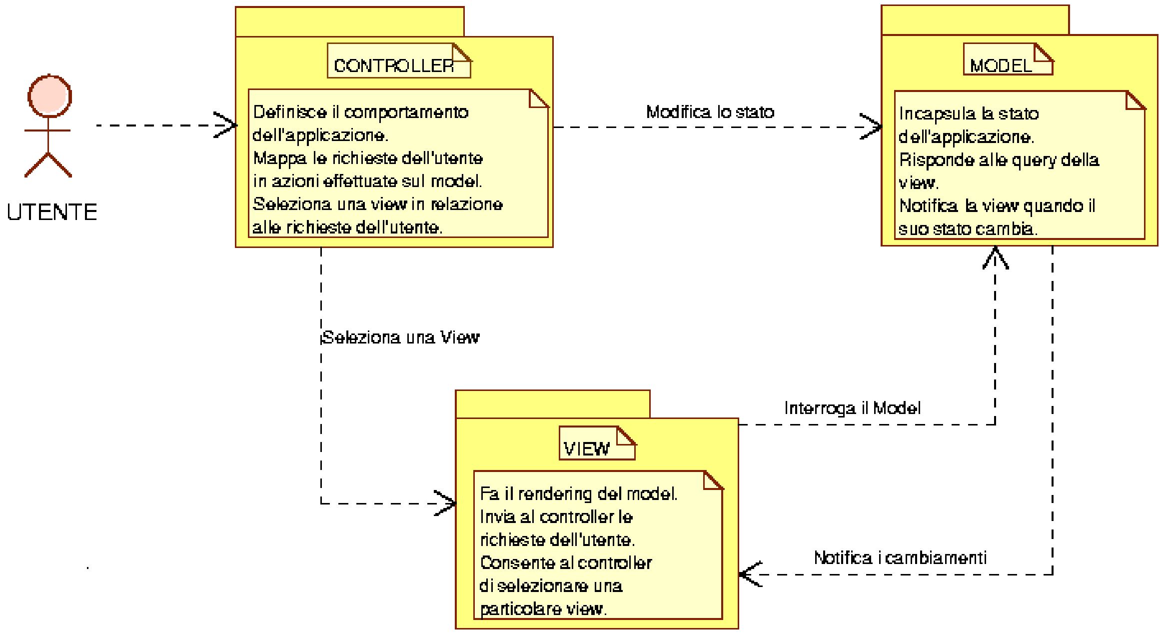

Figura 3.8: *Architettura Model View Control*: schema funzionale.

- la geometria del rivelatore di CMS;
- i dati transienti e persistenti attraverso COBRA, quindi i dati simulati e quelli ricostruiti da ORCA.

Per la *vista* IGUANA offre la possibilità di rappresentare i dati in 2D e 3D. In particolare usa le librerie X11 e OpenGL per un basso livello di grafica rispettivamente in 2D e 3D; mentre ad alto livello usa Qt [20] per la grafica 2D e OpenInventor [21] per creare una *scena grafica* in 3D.
Il *controllo* infine si riferisce alle interazioni tra l'utente e le prime due parti (*modello* e *vista*). La gestione di questa interazione nel caso specifico della grafica in 2D con Qt è fatta col sistema *slot/segnale*. Ogni *oggetto* può rispondere ad azioni dell'utente attraverso *slot* pubbliche, cioè particolari *metodi* che possono essere eseguiti ogni qualvolta un altro *oggetto* emette un *segnale* connesso a quella *slot*.

Sebbene l'idea dell'architettura MVC è nata prima dei *pattern*, in seguito lo stesso MVC è stato identificato e classificato come *pattern*, perchè effettivamente l'MVC propone un modo (già testato da altri programmatori) per organizzare l'architettura di un'interfaccia grafica. Dei *pattern* parlerò più approfonditamente nel successivo paragrafo.

3.2.3 I Pattern e i Design Pattern

Progettare sistemi secondo il paradigma OO non è facile e creare software OO che sia riutilizzabile è ancora più difficile: una soluzione dovrebbe essere specifica al problema ma abbastanza generale da poter essere riutilizzata. Di solito è molto difficile creare del software che vada bene la prima volta e si impiega molto tempo ad imparare a creare del buon codice OO. Un buon progettista OO non cerca di risolvere più volte lo stesso problema, bensì cerca di utilizzare soluzioni già esistenti.

Il riutilizzo di elementi nella fase di progettazione è l'obiettivo dei *pattern*[4] [22] che rendono il progetto maggiormente flessibile, elegante e soprattutto riusabile. Proprio negli ultimi anni i *pattern* per lo sviluppo del software sono diventati uno dei *tool* concettuali largamente adottato dai programmatori. I *pattern* non sono semplicemente suggerimenti per risolvere particolari problemi, ma una vera e propria disciplina scientifica che ha lo scopo di aiutare gli sviluppatori del software a risolvere problemi ricorrenti che si incontrano nella fase di analisi e codifica del software. In fase di sviluppo del software queste soluzioni di progettazione standard permettono di velocizzare la produzione di codice flessibile. Al fine di raggiungere questo scopo, è necessario definire formalmente un linguaggio comune per comunicare esperienza, per definire problemi e la loro soluzione. Questo approccio formale permette, quindi, di acquisire quella conoscenza profonda del problema e giustificare una soluzione piuttosto che un'altra.
Una definizione formale del *pattern* è la seguente [22]:

"Un Pattern è la descrizione chiara e formale di un problema (o famiglia di problemi) ricorrente, e della sua risoluzione. Questa descrizione ha lo scopo di catturare l'essenza stessa del problema all'interno di certi contesti. La comprensione completa di tali problemi è necessaria per apprezzare la soluzione proposta. Tale soluzione può essere non-ottimale se il problema viene estrapolato dal contesto considerato".

I problemi descritti nei *pattern* sono riconosciuti, nel senso che ogni sviluppatore ha incontrato lo stesso problema nel passato e lo incontrerà nuovamente nel futuro. Quindi creare un *pattern* significa riutilizzare e condividere l'esperienza di qualche altro sviluppatore che ha dovuto affrontare gli stessi

[4] Il termine *Pattern* fu coniato da alcuni lavori dell'architetto Christopher Alexander ed applicato allo sviluppo di piani urbani ed alle architetture di costruzioni urbane. Alexander li definisce nel seguente modo: *"Ciascun Pattern descrive un problema ricorrente e la sua soluzione, in modo tale che si potrà riusare questa soluzione un milione di volte, senza mai farlo allo stesso modo due volte".* Questo paradigma per l'architettura venne poi ripreso ed adattato da Ward Cunningham e Kent Beck, mentre lavoravano con Smalltalk alla creazione di interfacce utenti.

ostacoli logici e architetturali.

Un *pattern* è formato da quattro elementi essenziali:

- Nome: identificatore utilizzato per descrivere un problema di *design* e la relativa soluzione.
- Problema: quando applicare il *pattern*. Illustra il problema e il relativo contesto.
- Soluzione: gli elementi che costituiscono il *design* e le relazioni tra essi.
- Conseguenze: risultati derivanti dall'applicazione dei *pattern*.

Tra i vari *pattern* segnaliamo oltre al *Pattern Model View Control* di cui ho già parlato nel paragrafo precedente, i *Design Pattern* che sono stati usati nell'implementazione del prototipo di *event display* per il tracciatore di CMS, progettato in questo lavoro di tesi.

I *design pattern*,[5][23] vengono definiti come: "*la descrizione di oggetti e classi comunicanti che hanno lo scopo di risolvere problemi generali di programmazione in un particolare contesto*". In questo tipo di *pattern* vengono identificate tutte le classi partecipanti, le loro istanze, i loro ruoli e la distribuzione delle loro responsabilità. Ognuno dei *pattern* si concentrerà su particolari problemi di *design Object Oriented*.

Tra tutti i tipi di *design pattern* [23] che finora sono stati implementati, qui se ne descriveranno solo quelli che sono stati usati nella progettazione del prototipo per la visualizzazione del *Tracker*, delle tracce e degli *hit*. Per capire meglio il funzionamento dei *pattern* saranno riportati per ognuno di essi i partecipanti, cioé la lista delle *classi* e gli *oggetti* che partecipano nel *pattern* con le loro responsabilità e uno schema grafico usando il *diagramma delle classi* dell'UML.

Il Design Pattern Builder

Separa la costruzione di un oggetto complesso dalla sua rappresentazione, in modo da poter usare lo stesso processo di costruzione per altre rappresentazioni. [23]
Questo *pattern* (fig.3.9) propone di separare la "logica del processo di costruzione" dalla "costruzione stessa". Per fare ciò si utilizza un *oggetto "Director"*, che determina la logica di costruzione del prodotto, e che invia le istruzioni

[5]Letteralmente "*Modello di Progettazione*". Il *design pattern* non è un programma o codice, ma è un disegno che deve essere adattato per soddisfare i requisiti specifici di un particolare problema e quindi implementato.

necessarie ad un *oggetto Builder*, incaricato della sua realizzazione. Siccome i prodotti da realizzare sono di diversa natura, ci saranno *Builder* particolari per ogni tipo di prodotto, ma soltanto un unico *Director*, che nel processo di costruzione invocherà i metodi del *Builder* scelto secondo il tipo di prodotto desiderato (i *Builder* dovranno implementare un'interfaccia comune per consentire al *Director* di interagire con tutti questi).

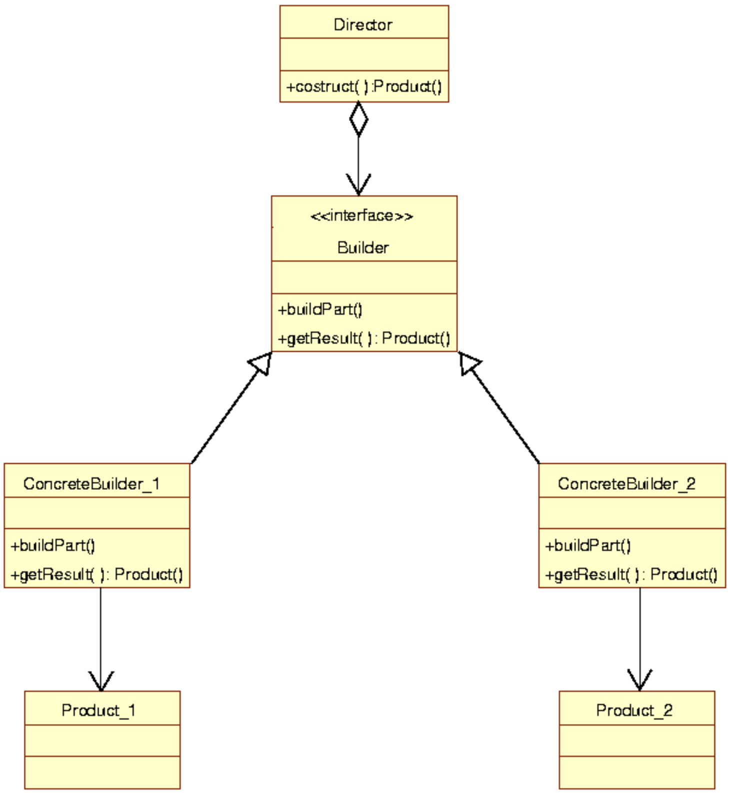

Figura 3.9: *Pattern Builder*: schema del modello.

Partecipanti:

Builder: *classe astratta Builder*

- Dichiara un'interfaccia per le operazioni che creano le parti dell'oggetto *Product*.
- Implementa il comportamento di *default* per ogni operazione.

ConcreteBuilder: *classi ConcreteBuilder_1* e *ConcreteBuilder_2*

- Forniscono le operazioni concrete dell'interfaccia corrispondente al *Builder*.
- Costruiscono e assemblano le parti del *Product*.
- forniscono un metodo per restituire il *Product* creato.

Director: *classe Director*

- Costruisce il *Product* invocando i metodi dell'interfaccia del *Builder*

Product: *classi Product_1 e Product_2*

- Rappresenta l'oggetto complesso in costruzione. I *ConcreteBuilder* costruiscono la rappresentazione interna del *Product*.
- Include classi che definiscono le parti costituenti del *Product*.

Questo *design pattern* è stato usato nella progettazione delle *classi* che devono accedere al *database* dei dati del rivelatore e degli eventi. Uno scopo che ci si è prefissi è quello di rendere il programma indipendente da IGUANA in modo da usarlo come programma *stand alone*, quindi in tal caso l'accesso ad *database* sarà diverso rispetto a quando si è in IGUANA.

Il Design Pattern Singleton

Assicura che la classe abbia una sola istanza e provvede un modo di accesso. [23]
Il "*Singleton*" *pattern* (fig.3.10) definisce una *classe* della quale è possibile la istanziazione di un unico *oggetto*, tramite l'invocazione a un metodo della *classe*. Le diverse richieste di istanziazione, comportano la restituzione di un riferimento allo stesso *oggetto*.

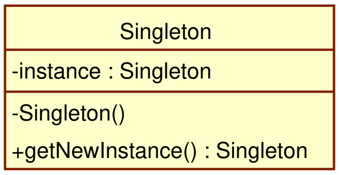

Figura 3.10: *Pattern Singleton*: schema del modello.

Partecipanti:

Singleton: *classe Singleton*

- Definisce un metodo *getNewInstance()* che restituisce un riferimento all'unica istanza di se stessa.
- É responsabile della creazione della propria unica istanza.

Questo *design pattern* è stato usato ogni volta che un *oggetto* poteva essere instanziato da più *oggetti*, ma una volta creato doveva restare unico, cioé gli altri possibili instanziatori non ne devono creare uno nuovo, ma usare quello che già esiste una volta che è stato creato.

Il Design Pattern Observer

Definisce una dipendenza 1:N tra oggetti in modo che se uno cambia stato gli altri siano aggiornati automaticamente. [23]
Il *pattern* "*Observer*" (fig.3.11) assegna all'*oggetto* monitorato (*Subject*) il ruolo di registrare al suo interno un riferimento agli altri oggetti che devono essere avvisati (*ConcreteObservers*) degli eventi (cambiamento di stato) del *Subject*, e notificarli tramite l'invocazione a un loro metodo, presente nell'interfaccia che devono implementare (*Observer*). É prevista la registrazione dell'*Observer* presso il *Subject*, indicando addizionalmente il tipo di evento davanti al quale l'*Observer* deve essere notificato, e la funzione dell'*Observer* da invocare.
Nella progettazione useremo una semplice implementazione descritta di seguito che è possibile solo nel caso in cui gli oggetti generano un solo tipo di evento.

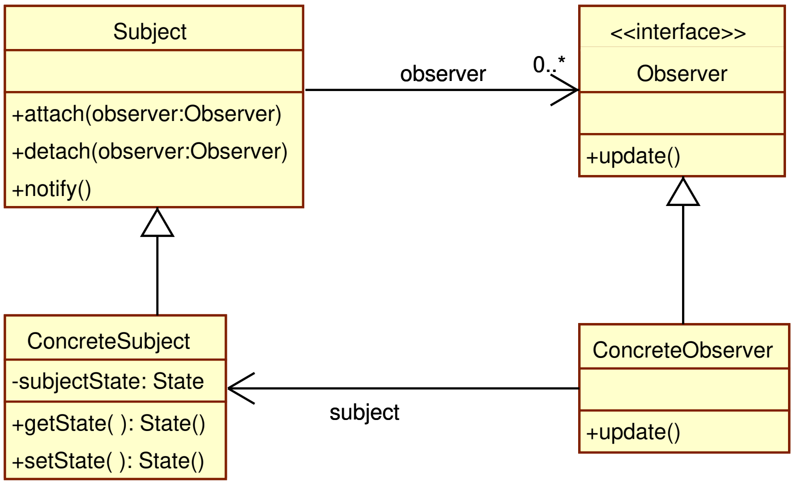

Figura 3.11: *Pattern Observer*: schema del modello.

Partecipanti:

Subject: *classe Subject*

- Ha conoscenza dei propri *Observer*, i quali possono esserci in numero illimitato.
- Fornisce operazioni per l'addizione e la cancellazione di oggetti *Observer*.
- Fornisce operazioni per la notifica agli *Observer*.

Observer: *interfaccia (classe base astratta) Observer*

- Specifica un'interfaccia per la notifica di eventi agli oggetti interessati in un *Subject*.

ConcreteSubject: *classe ConcreteSubject*

- Possiede uno stato dell'interesse dei *ConcreteSubject*.
- Invoca le operazioni di notifica ereditate dal *Subject*, quando devono essere informati i *ConcreteObserver*.

ConcreteObserver: *classi ConcreteObserver*

- Implementa l'operazione di aggiornamento dell'observer.

Questo *design pattern* è stato usato per controllare le parti del tracciatore che vengono selezionate, perché si è prevista la possibilità di selezionare le componenti del *Tracker* in più modi, quindi ogni oggetto di selezione deve comunicare con gli altri in modo da sapere sempre cosa è stato già selezionato. Il tipo di evento gestito è quindi la modifica delle parti selezionate.

Gli *oggetti* che devono essere informati dei cambiamenti, al momento della creazione devono registrarsi come *Observer* presso il *Subject*. Ogni volta che uno degli *Observer* subisce un cambiamento, invia un segnale al *Subject* che provvede a notificare la modifica evvenuta agli altri *Observer* registrati, che così possono aggiornarsi (fig.3.12).

3.3 Come collaborare per un grande progetto software

Uno degli obiettivi del software di CMS è la realizzazione di un'interfaccia grafica che permetta all'utente la visualizzazione e l'analisi della geometria del rivelatore e degli eventi simulati e ricostruiti. Per le grandi dimensioni del rivelatore, costituito da diversi tipi di rivelatori, e per la grande quantità di dati che saranno prodotti, il software di CMS sarà complesso e di grandi

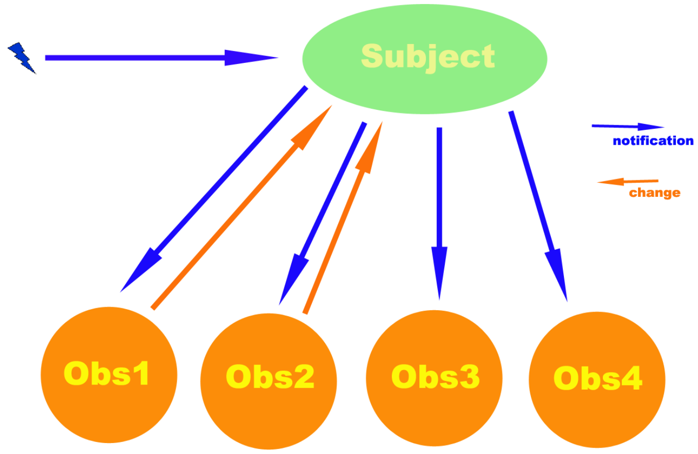

Figura 3.12: Schema grafico che mostra il funzionamento della comunicazione tra il *Subject* e gli *Observer*. Il *Subject* viene creato all'avvio del programma, mentre gli *Observer* (Obs1, Obs2, ...) possono essere creati in seguito. Quando Obs1 o Obs2 subiscono dei cambiamenti ne informano (freccia arancio) il *Subject* che provvede a notificare (freccia blu) questo cambiamento a tutti gli Obs.

dimensioni. Di conseguenza, per un progetto di tali dimensioni è richiesta la collaborazione di molti programmatori che si occupino dell'implementazione dei diversi moduli del software. Questi moduli dovranno interagire per ottenere il prodotto finale, quindi gli sviluppatori del software dovranno seguire delle regole comuni generali di programmazione e dovranno comunicare tra loro in modo da non sviluppare moduli software che vanno in conflitto o su cui sta già lavorando altra gente. Essendo questo poi, un progetto internazionale si deve permettere a chi è interessato di accedere ai sorgenti del software che è già stato implementato per la realizzazione del primo prototipo funzionante o a quelli della versione in fase di sviluppo.

3.3.1 Il sistema CVS

Il *Concurrent Versions System* (CVS)[24] [25] è un sistema per l'archiviazione e la condivisione di materiale digitale, che consente la modifica concorrente dello stesso tra utenti autorizzati, che generalmente comunicano tra loro e con il server tramite internet. In pratica, permette a un gruppo di persone di lavorare simultaneamente sullo stesso gruppo di file (generalmente si tratta di sorgenti di un programma), mantenendo il controllo dell'evoluzione delle modifiche che vengono apportate. Per attuare questo obiettivo, il sistema CVS mantiene un deposito centrale (*repository*) dal quale i collaboratori di un progetto possono ottenere una copia di lavoro. I collaboratori modificano i file della loro copia di lavoro e sottopongono le loro modifiche al sistema CVS che le integra nel deposito. Il compito di un sistema CVS non si limita

```
/var/
 |
 |-- radice-cvs/
 |     |
 :     |-- CVSROOT/
       |        (directory amministrativa)
       |
       |-- esercizi/
       |    |    (modulo «esercizi»)
       :    |
            |-- basic/
            |    (modulo «esercizi/basic»)
            |
            |-- c/
            |    (modulo «esercizi/c»)
            |
            |-- pascal/
            |    (modulo «esercizi/pascal»)
            :
```

Figura 3.13: Struttura di un deposito CVS.

a questo; per esempio è sempre possibile ricostruire la storia delle modifiche apportate a un gruppo di file, oltre a essere anche possibile ottenere una copia che faccia riferimento a una versione passata di quel lavoro.

Il *repository* contiene le informazioni sullo svolgimento di uno o più progetti gestiti da uno o più gruppi di persone. Questo deposito è costituito da una *directory* che si sviluppa in una gerarchia più o meno complessa, in base alla struttura di ciò che si vuole amministrare. All'interno di un deposito si possono articolare diverse *directory*, con la loro struttura, riferite a uno stesso progetto o a progetti differenti, che possono articolarsi in sottoprogetti a seconda delle necessità. Nella terminologia di CVS, questi progetti sono dei moduli. La figura 3.13 mostra un esempio di come potrebbe essere collocata e strutturata la gerarchia di un deposito; in particolare, tutti i nomi che si vedono sono delle *directory*.

Il collaboratore che intende partecipare allo sviluppo di un modulo (o di un sottomodulo), riproduce una copia di questo a partire da una sua *directory* di lavoro. La figura 3.14 mostra il caso dell'utente tizio che possiede una copia del modulo esercizi/pascal (e non degli altri) a partire dalla propria *directory* personale.

```
~tizio/
  |
  |-- esercizi/
  |     |
  :     |-- CVS/
        |     (directory amministrativa)
        |
        |-- pascal/
        |     |
        :     |-- CVS/
              |     (directory amministrativa)
              |
              |-- BSort.pas
              |-- BSort2.pas
              :     (sorgenti pascal da sviluppare)
              :
```

Figura 3.14: Esempio di struttura della copia di un modulo appartenente a un collaboratore.

Quando un collaboratore ha riprodotto la propria copia di lavoro del modulo di suo interesse, può apportare le modifiche che ritiene opportune nei file e quindi dovrebbe fare in modo di aggiornare anche il deposito generale. A questo proposito conviene distinguere le fasi seguenti:

- modifica dei file;
- aggiornamento della copia locale;
- sistemazione dei conflitti;
- invio delle modifiche al deposito.

3.3.2 Sviluppare moduli software per ORCA

Per iniziare a programmare in ORCA, bisogna scaricare nella propria *home* un ambiente con tutte le dipendenze settate e funzionanti.
Con il comando:
`scram list ORCA`
si ottiene la lista delle versioni di ORCA disponibili. Scelta la versione che si vuole scaricare, per esempio la `ORCA_X_X_X`, con il comando:
`scram project ORCA ORCA_X_X_X`
si crea nella propria *home* una *directory* chiamata `ORCA_X_X_X` con le *sotto-directory*:

src contiene vari *Sub-System* con diversi *package* che contengono il codice sorgente;

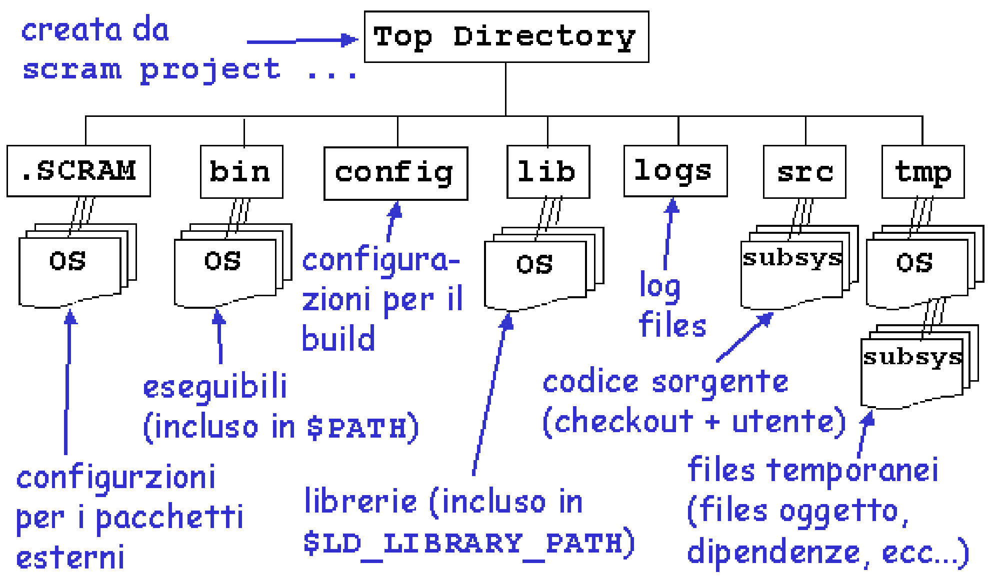

Figura 3.15: Struttura delle *directory* di ORCA.

bin contiene il codice compilato e linkato;

tmp contiene i file temporanei;

.SCRAM contiene la configurazione e le dipendenze nel codice;

lib contiene i *link* alle librerie.

come rappresentato in figura 3.15.
Ora la cartella `src` che contiene il codice, è vuota perché abbiamo scaricato solo la struttura di ORCA e non il codice stesso. Con il comando:

```
cvs co -r ORCA_X_X_X Visualisation
```

si scarica il sottosistema `Visualisation` che contiene il codice per la visualizzazione in ORCA. Se non si specifica la versione vengono scaricati i sorgenti della versione correntemente in sviluppo, detta *head*, che probabilmente non funziona.

I vari sottosistemi di `src` contengono diversi *package* costituiti con la seguente struttura interna (fig. 3.16):

interface contiene l'interfaccia delle *classi*, cioé gli *header file* (con estensione *.h*);

src contiene il codice C^{++} delle *classi*;

test contiene il codice utile per testare il pacchetto;

doc contiene la documentazione del pacchetto.

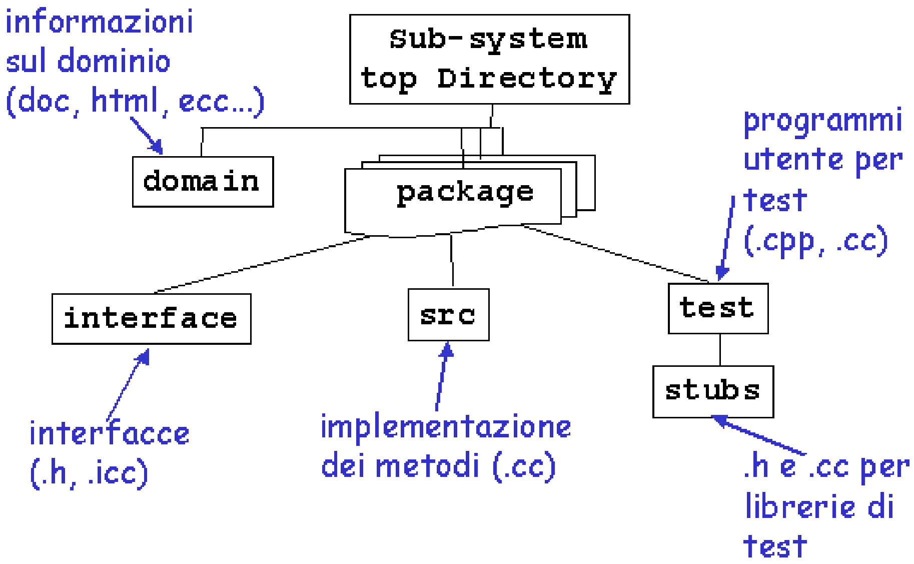

Figura 3.16: Struttura di un *Sub-System* di ORCA.

Una volta scaricato il sottosistema `Visualisation` si compila il codice col comando:
`scram build`
nella corrispondente *directory*.

Quindi ORCA ha una struttura gerarchica a due livelli con diversi *Sub-Systems* ognuno dei quali ha diversi *package*. La figura 3.17 dà un'idea dei diversi *Sub-System* esistenti sotto `src`.

Tra i diversi pacchetti che il sottosistema `Visualisation` contiene, è stato creato [1] **CustomTracker** (fig.3.18) che contiene il codice per la visualizzazione in 2D del *Tracker*, all'interno di questo sistema è stato inserito il software progettato e sviluppato in questo lavoro di tesi.

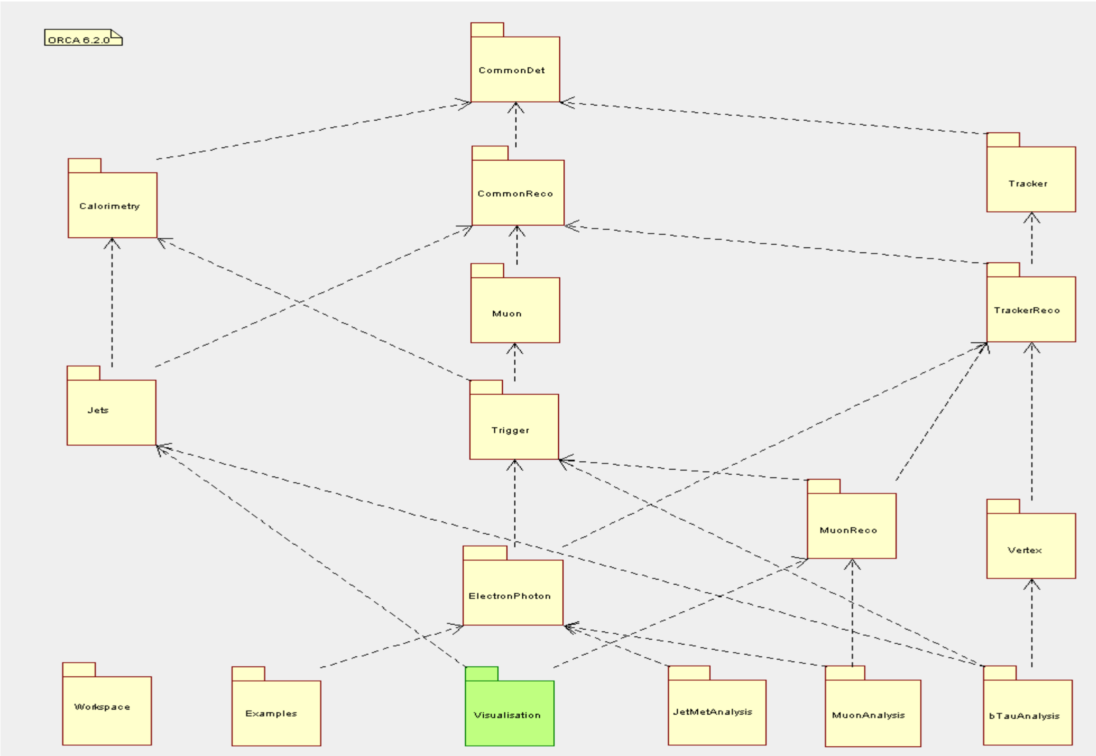

Figura 3.17: ORCA project structure. Struttura dei Sub-Systems di ORCA con le dipendenze.

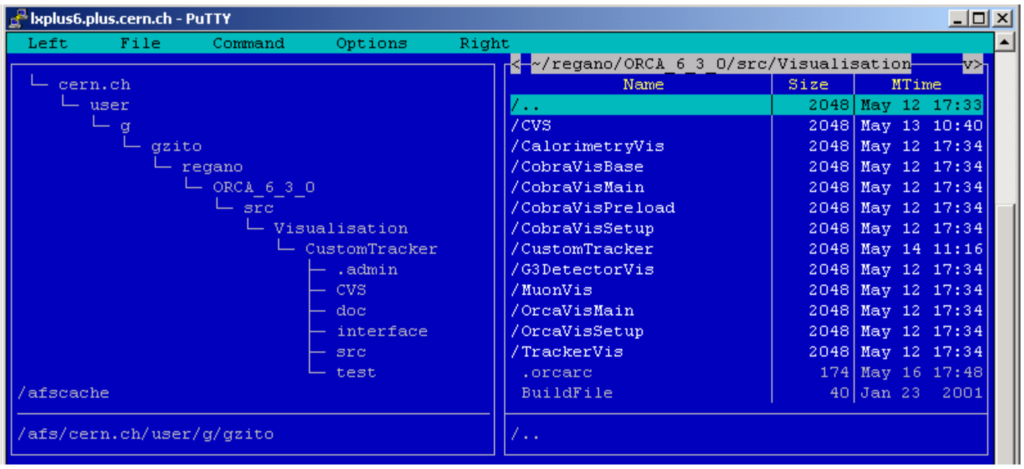

Figura 3.18: A sinistra è mostrato il percorso delle cartelle sotto ORCA_6_3_0 a partire dalla home e a destra è riportata la lista dei package contenuti nel Sub-System *Visualisation*, tra i package c'è *CustomTracker*.

Capitolo 4

Progettazione del tool di visualizzazione del Tracker

Il progetto del *tool* di visualizzazione del *Tracker* è partito dall'esperienza acquisita con la demo [1] realizzata in precedenza; il progetto ha richiesto una ulteriore e più approfondita analisi delle specifiche desiderate dagli utenti (*casi d'uso*) e si è avvalsa di *tool* di modellizzazione come l'UML che consentono di specificare le funzionalità richieste al software e ne rendono più semplice la sua implementazione.

Come strumento di supporto all'UML è stato utilizzato il software *Poseidon* [26] con il quale sono state presentate le viste, i diagrammi e gli elementi necessari per la comprensione del progetto.

Nella fase di progettazione sono stati usati alcuni *design pattern* per risolvere dei problemi particolari. Il progetto è stato sviluppato in modo tale da permettere poi la realizzazione sia del prototipo per la versione 7_2_0 di ORCA, sia per il prototipo *stand alone*.

4.1 Analisi delle funzionalità del software di visualizzazione del Tracker

Nella prima fase della progettazione si è cercato di stabilire, nel modo più dettagliato possibile, le principali specifiche richieste al software di visualizzazione del *Tracker*. Il *diagramma dei casi d'uso* di figura 4.1, mostra in modo schematico le funzionalità che il software di visualizzazione metterà a disposizione dell'utente di ORCA.

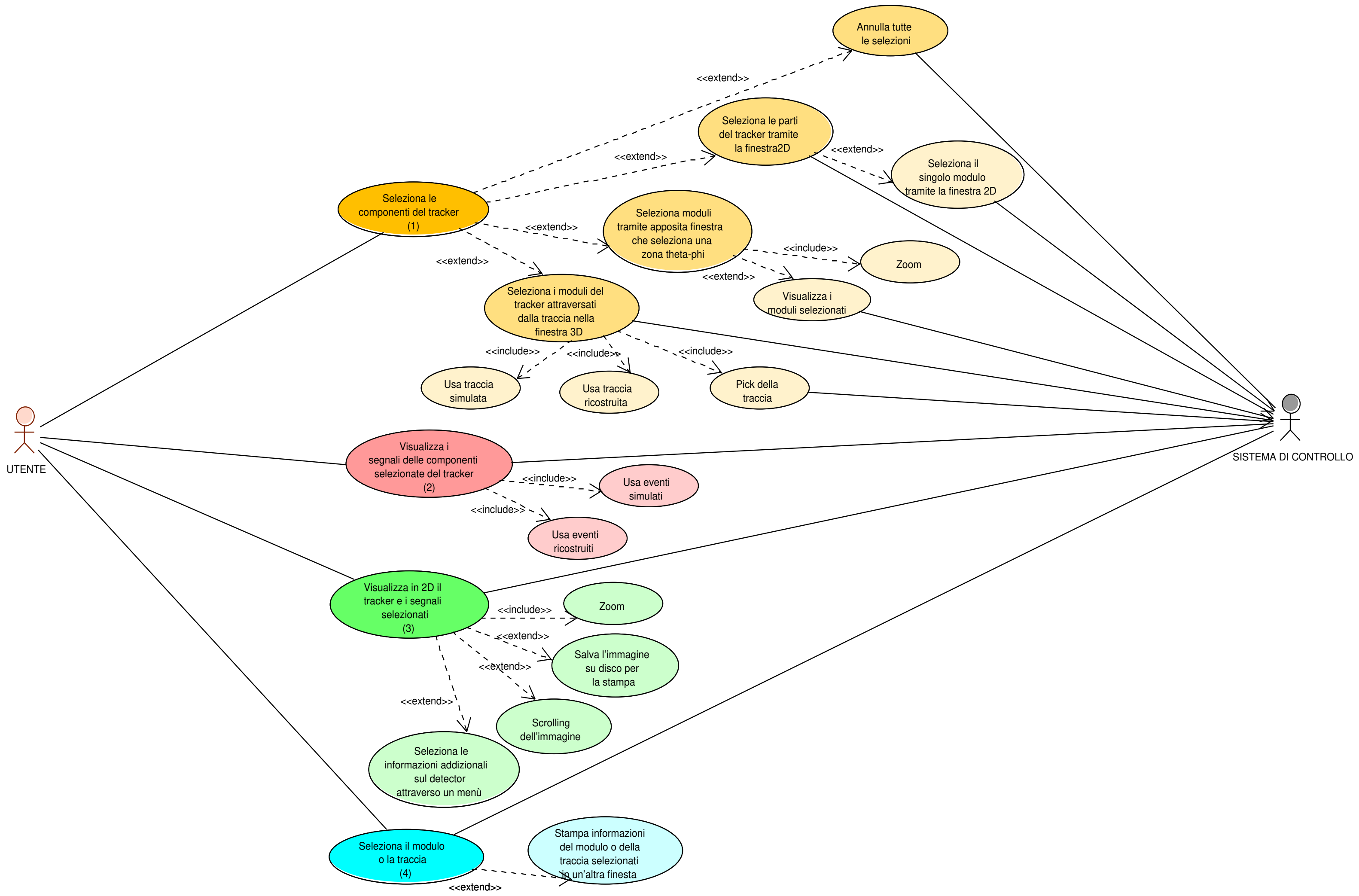

Figura 4.1: *Diagramma dei casi d'uso.* Questo *diagramma dei casi d'uso* mostra tutte le funzioni che il software metterà a disposizione dell'utente. Il "sistema di controllo" è il sistema, che si occupa di gestire la comunicazione tra i diversi *casi d'uso*.

I principali *casi d'uso* sono tre:

(1) la selezione delle componenti del *Tracker*, rappresentata in giallo;

(2) la visualizzazione in 3D dei segnali e delle componenti del *Tracker*, indicato col colore rosa;

(3) la visualizzazione in 2D del *Tracker* e dei segnali, indicato in verde.

Questi tre *casi d'uso* descrivono la modalità con cui l'utente di ORCA potrà selezionare gli elementi cui è interessato e potrà quindi scegliere se visualizzarli nella finestra 3D di *OpenInventor* (OI), oppure nella finestra di visualizzazione in 2D o in entrambe.

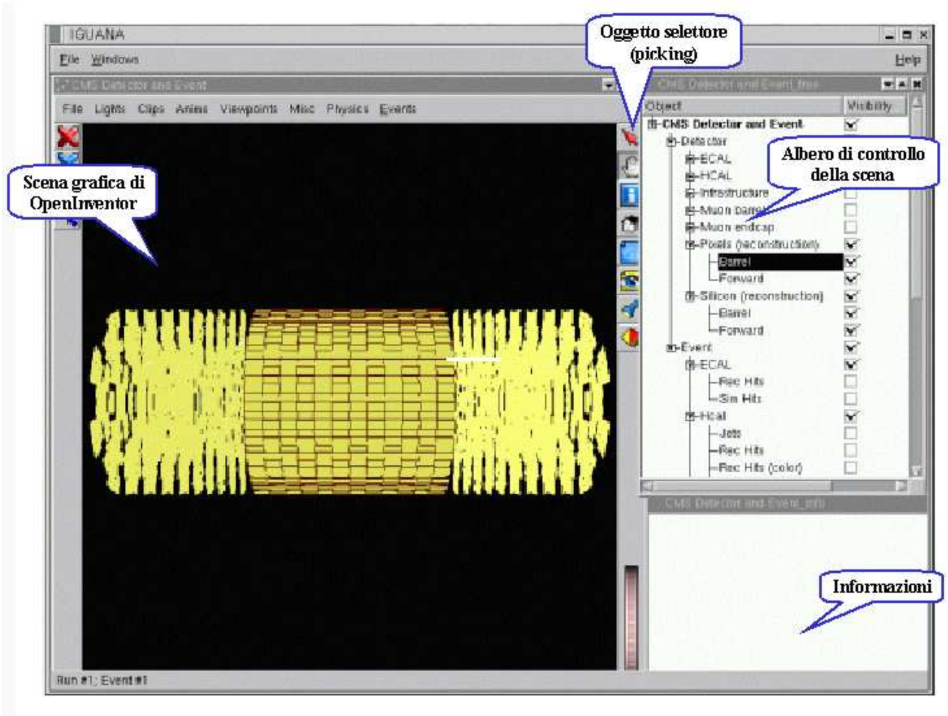

Figura 4.2: La visualizzazione in ORCA usando IGUANA.

Infine è presente un quarto *caso d'uso*:

(4) "Selezione del modulo o della traccia", indicato in blu.

Questo *caso d'uso* permetterà all'utente di fare una selezione (*pick*) nella finestra 3D di OI, e di selezionare così uno dei moduli o una delle tracce già visualizzate nella finestra; dell'elemento selezionato verranno stampate le relative informazioni nella finestra di "testo" (Informazioni) di IGUANA (fig.4.2). Il*caso d'uso* "Selezione del modulo o della traccia" è già sviluppato all'interno del software IGUANA, infatti come si vede in figura 4.2 tra i pulsanti vi è l'"oggetto selettore" che è quello che permette di fare il *pick* nella finestra di OI.

Invece per accedere agli altri tre *casi d'uso* principali bisogna farlo attraverso l'"albero di controllo" (fig.4.3), dove nel sottoalbero **Event** è prevista la voce **CustomTracker**. Sotto il "ramo" **CustomTracker** è stato costruito

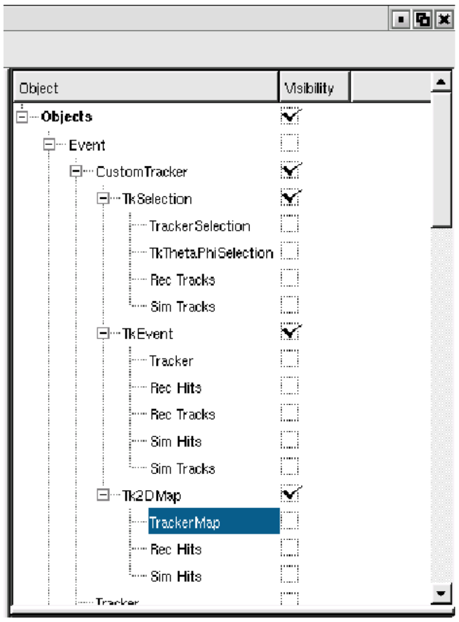

Figura 4.3: Parte dell'"albero di controllo" in cui sono mostrati tutti i "rami" e le "foglie" presenti sotto *CustomTracker* che permettono l'accesso agli strumenti software forniti dal prototipo.

un'ulteriore sottoalbero con i tre "rami" principali:

TkSelection
TkEvent
Tk2DMap

per i tre *casi d'uso* principali, poi ognuno di questi "rami" ha diverse "foglie". Inizialmente il sottoalbero di **CustomTracker** non è visualizzato, quando l'utente seleziona **CustomTracker**, gli vengono mostrati i tre "rami" principali, e poi selezionando uno di questi vengono visualizzate le relative "foglie" la cui selezione consentirà di usare gli strumenti software che il *tool* di visualizzazione grafica del *Tracker* fornirà.

I diversi *casi d'uso* non sono isolati, ma comunicano con il "sistema di controllo", che si occupa dell'aggiornamento delle viste e della comunicazione tra i *casi d'uso* di selezione e quelli di visualizzazione.

Nello sviluppare questi casi d'uso, nel prossimo paragrafo saranno descritti in dettaglio l'interfaccia utente utilizzando le immagini della precedente proposta e della demo.

4.1.1 Primo caso d'uso principale: la selezione delle componenti del Tracker

Il *caso d'uso* di selezione delle componenti del tracciatore non è unico, ma comporta quattro ulteriori *casi d'uso* di estensione con altri sottocasi indicati in figura 4.1 con diverse sfumature di giallo. I tre *casi d'uso* di estensione principali sono:

1. selezione delle parti del *Tracker* tramite una finestra contenente una mappa stilizzata del *Tracker* stesso;

2. selezione dei moduli del *Tracker* tramite apposita finestra in cui l'utente seleziona una zona θ/ϕ;

3. selezione dei moduli del *Tracker* attraversati da una traccia visualizzata nella finestra 3D;

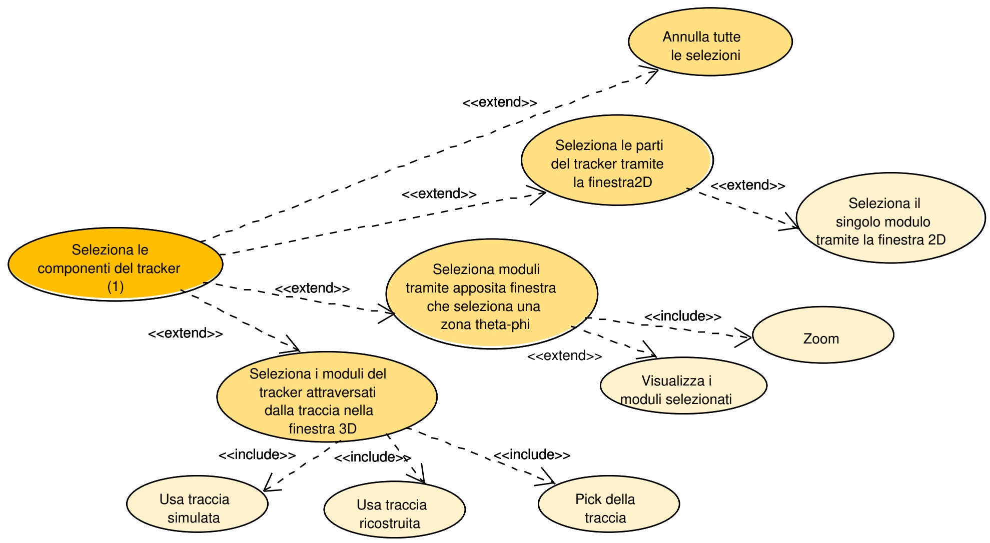

Figura 4.4: Ingrandimento dei *casi d'uso* di selezione.

infine, vi è il quarto *caso d'uso* di estensione: "annulla tutte le selezioni" che prevede il *reset* di tutte le selezioni effettuate dall'utente permettendo al sistema di tornare nella situazione iniziale.

L'utente, per selezionare le parti del tracciatore che vuole visualizzare nella finestra 3D di OI o nella finestra 2D, può usare indipendentemente uno dei tre modi di selezione o anche tutti e tre. Questo comporta che gli *oggetti* che implementano questi tre modi devono comunicare tra loro in modo che ognuno di essi sappia sempre cosa è stato caricato (visualizzato) nella finestra 3D o 2D, infatti questi *casi d'uso* sono collegati al "sistema di controllo" che gestisce questa comunicazione.

Selezione delle parti del Tracker tramite una sua mappa stilizzata

Selezionando nell'"albero di controllo" sotto **CustomTracker/TkSelection** la "foglia" **TrackerSelection**, si apre una nuova finestra che permetterà la selezione delle componenti del *Tracker*. Rispetto alla prima demo realizzata [1] (fig.2.4), questa finestra (fig.4.5 e 4.6) avrà dei nuovi elementi di selezione.

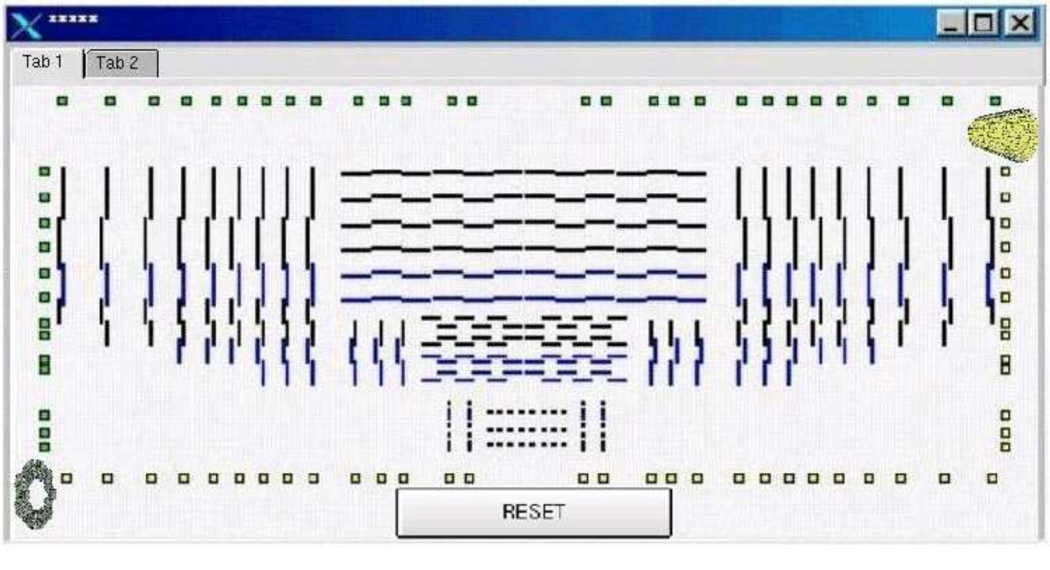

Figura 4.5: Finestra di selezione delle parti del *Tracker* Tab1.

La finestra avrà un TAB con due "linguette" e un tasto per il *reset* nella parte inferiore. Selezionando "Tab 1" (fig.4.5), si otterrà la stessa mappa stilizzata del tracciatore già usata per la demo (fig.2.8) con in più due bottoni che permetteranno di selezionare rispettivamente tutti i *layer* dell'*endcap*, quello

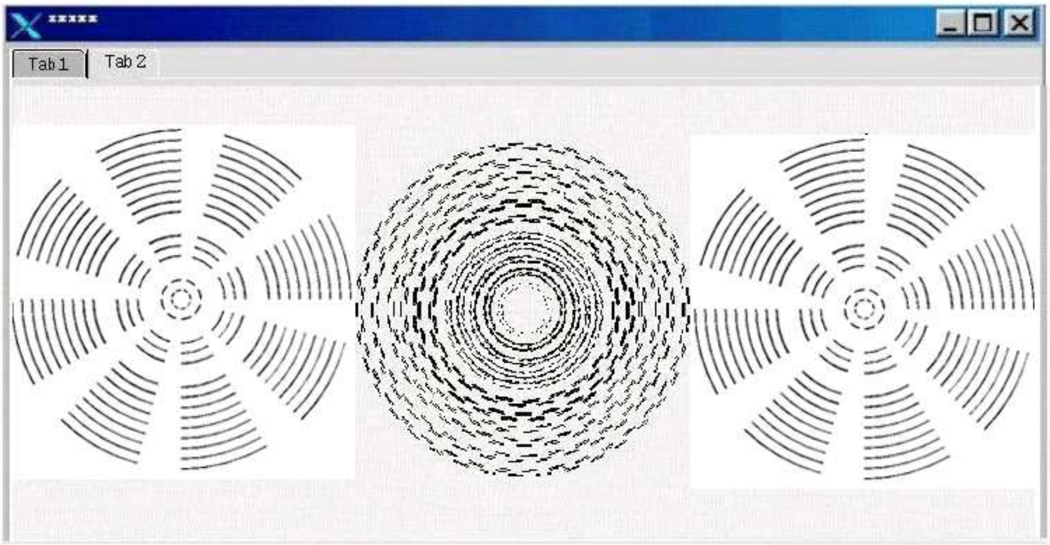

Figura 4.6: Finestra di selezione delle parti del *Tracker* Tab2.

in basso a sinistra, e tutti i *layer* del *barrel*, quello in alto a destra, così cliccando su entrambi si selezionerà l'intero *Tracker*. Invece selezionando "Tab 2" si avrà un'altra mappa stilizzata del *Tracker* che permetterà la selezione di petali e *rod* dei *layer*. Il disegno centrale (fig.4.6) si riferisce ai *layer* del *barrel*, in esso ogni anello rappresenta un *layer* del *barrel* e ogni arco una *rod* di quel *layer*, così si potranno selezionare le *rod* volute cliccando con il mouse sull'arco che gli corrisponde. In realtà la *rod* nel tracciatore indica solo un elemento costruttivo del TIB che contiene i moduli di metà di ognuna delle file che formano un cilindro; qui noi usiamo il termine *rod* per indicare una riga completa di moduli sia del TIB che del TOB e della parte *pixel*. Le due figure laterali (fig.4.6), invece, si riferiscono ai *layer* dell'*end-cap*, in essi ogni anello rappresenta un *layer* dell'*endcap* e ogni arco un petalo di quel *layer*, così si potranno selezionare i petali voluti cliccando con il mouse sull'arco corrispondente. Come estensione di questo *caso d'uso* vi è quello di "selezione del singolo modulo tramite una finestra 2D". Questo elemento è rimasto sostanzialmente identico a quello realizzato per la demo (fig.4.7) e vi si accede sempre cliccando sui quadratini verdi della finestra di figura 4.5.

La finestra di selezione di una zona θ/ϕ

Selezionando nell'"albero di controllo" sotto **CustomTracker/TkSelection** la foglia **TkThetaPhiSelection** si apre una nuova finestra che mostra una distribuzione di *hit* simulati (*SimHit*). Il funzionamento di questo strumento

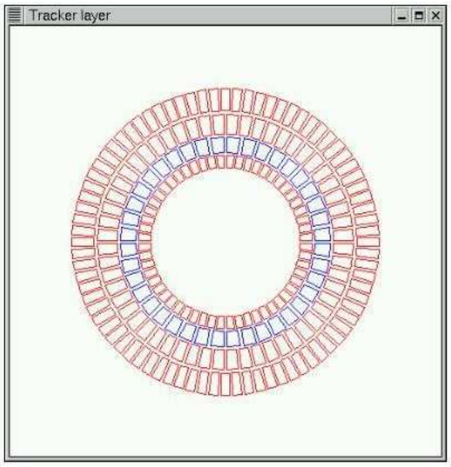

Figura 4.7: Finestra di selezione del singolo modulo di un determinato *layer*.

di selezione sarà lo stesso di quello descritto nel paragrafo 2.3 (fig.2.6) in cui sono stati presentati i nuovi oggetti grafici 2D per la visualizzazione del *Tracker* e degli eventi. L'utente potrà zoomare in questa finestra (fig.4.8) fino a trovare una zona interessante ed allora richiede la "visualizzazione dei moduli selezionati".

La selezione dei moduli del Tracker attraversati dalla traccia

L'utente potrà anche, selezionare i moduli attraversati da una traccia particolare selezionando con l' "oggetto selettore " di IGUANA la traccia di suo interesse nella finestra 3D di OI (fig.4.9). Questo è possibile farlo sia con tracce simulate che con tracce ricostruite. Per usare questo modo di selezione dei moduli del *Tracker* si deve cliccare nell'"albero di controllo" sotto **CustomTracker/TkSelection** la "foglia":

RecTracks - per selezionare i moduli attraversati dalla traccia ricostruita.

SimTracks - per selezionare i moduli attraversati dalla traccia simulata.

4.1.2 Secondo caso d'uso principale: la visualizzazione dei segnali delle parti selezionate del Tracker

Il secondo *caso d'uso* riguarda la visualizzazione dei segnali legati alle componenti del *Tracker* selezionate indicato in rosa in figura 4.1. Nell'"albero di controllo" sotto **CustomTracker/TkEvent** sono previste le "foglie":

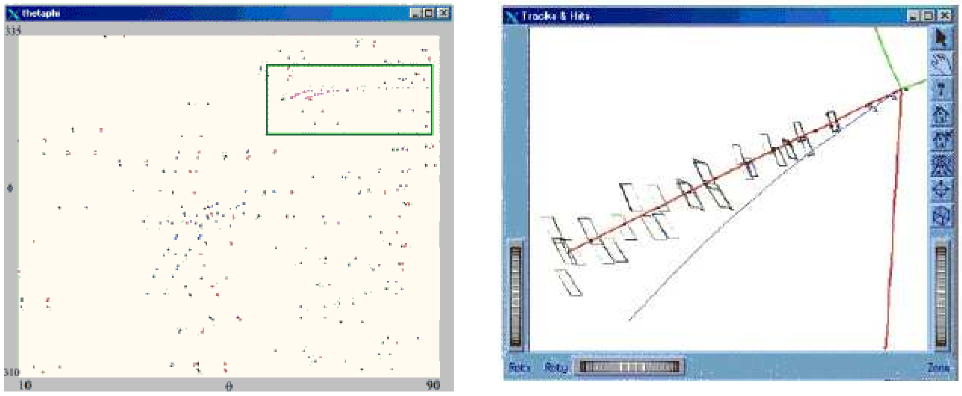

Figura 4.8: Finestra di selezione di una zona θ/ϕ. Nella prima finestra l'utente fa uno zoom su una particolare area e seleziona la regione di suo interesse, nella seconda finestra (quella di OI) vengono mostrati le componenti del *Tracker* presenti nella regione selezionata.

Rec Hits
Rec Tracks
Sim Hits
Sim Tracks

Selezionandole si attiva l'*oggetto* in questione che funziona nel seguente modo:

- Se la finestra 3D di OI (o quella 2D della mappa del *Tracker*) è vuota, la selezione di una delle quattro voci sembrerà non produrre effetti, ma nel momento in cui l'utente selezionerà delle componenti del *Tracker* facendole visualizzare nella finestra 3D (o 2D), allora verranno visualizzati anche le tracce o gli *hit* relativi ai moduli selezionati.

- Se l'utente ha già selezionato alcune componenti del *Tracker* che sono già visibili nella finestra 3D di OI (o quella 2D della mappa del *Tracker*), allora selezionando per esempio la foglia **Sim Hits** verrannno visualizzati, sempre nella finestra 3D (o 2D), gli *hit* simulati relativi alle parti del *Tracker* che sono già nella finestra. Analogamente per **Sim Tracks** etc.

Si noti che gli *oggetti* che implementano questo *caso d'uso* devono in qualche modo comunicare con gli *oggetti* che creano la finestra di selezione delle componenti del *Tracker*, per sapere quali sono gli elementi del *Tracker* selezionati, per poter poi caricare e mostrare gli *hit* e le tracce che riguardano proprio questi moduli. Di questo aggiornamento si occupa il sistema di controllo con cui questo *caso d'uso* è collegato (comunica).

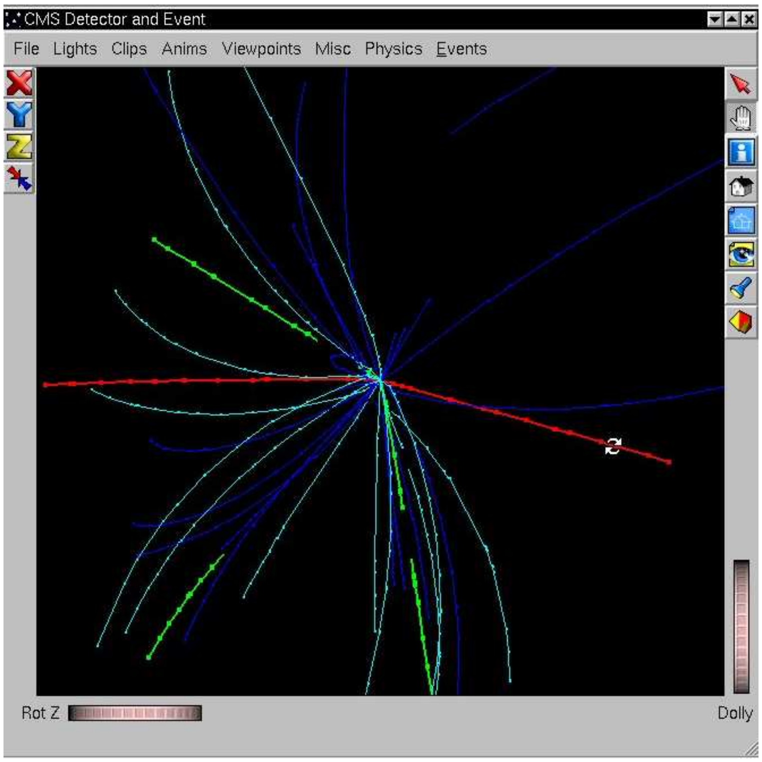

Figura 4.9: Finestra di selezione dei moduli del *Tracker* attraversati da una specifica traccia.

Nell'"albero di controllo" sotto **CustomTracker/TkEvent** è presente anche la "foglia" **Tracker**, la selezione di questa fa si che gli elementi selezionati con gli strumenti software forniti con il primo *caso d'uso* principale, quello di selezione, vengano caricati e visualizzati nella finestra 3D di OI. Questo è stato previsto per evitare di caricare gli elementi selezionati nella finestra 3D quando li si vuole visualizzare solo nella finestra 2D del *caso d'uso* descritto di seguito.

4.1.3 Terzo caso d'uso principale: la visualizzazione in 2D del Tracker e dei segnali

Nell'"albero di controllo" sotto **CustomTracker/Tk2DMap** è prevista la "foglia" **TrackerMap**, selezionandola si apre una nuova finestra 2D, costruita in Qt. Questo *caso d'uso* (indicato in verde in fig.4.1) è legato alla selezione dei moduli del rivelatore fatta con gli strumenti software descritti nel primo *caso d'uso* principale (paragrafo 4.1.1). In pratica quando l'utente selezionerà dei moduli del tracciatore usando i metodi visti prima, questi verranno visualizzati nella mappa 2D.

Nella finestra di visualizzazione in 2D l'utente potrà selezionare una regione e visualizzarla ingrandita (zoom) e la stessa finestra sarà dotata di *scrollbar* orizzontale e verticale che permetteranno lo *scrolling* dell'intera immagine che deve essere molto grande se si vuole visualizzare l'intero *Tracker* in dettaglio. Con uno schermo grande (almeno 21 pollici) l'utente potrà visionare l'intera mappa del *Tracker* con una buona risoluzione senza dover usare lo *scrolling*. L'utente potrà anche salvare in un file l'immagine in formato grafico per farne una stampa su carta, per archiviarla, etc. La dimensione dell'immagine sullo schermo è stata scelta in modo da poter essere stampata in ogni dettaglio su un foglio A4. Quindi l'utente che dispone di un piccolo schermo potrà sempre ottenere una rappresentazione dell'intero *Tracker* su carta. La finestra sarà dotata anche di un menù in cui l'utente potrà scegliere il tipo di informazioni relative al tracciatore o agli eventi che vuole visualizzare.

In questa rappresentazione saranno mostrate varie caratteristiche geometriche dei moduli del *Tracker* che saranno rappresentati conservando la loro forma e la loro posizione rispetto ad altri moduli dello stesso. Su questa forma base sarà possibile rappresentare altre informazioni relative allo stesso modulo, codificandole in maniera opportuna. La maniera più semplice è il colore. Ad esempio una rappresentazione a due colori potrà indicare i moduli singoli e doppi. Una rappresentazione usando i colori di una tavolozza potrà rappresentare informazioni come: tipo di sensore usato, numero di *strip* morte, temperatura, etc. La stessa mappa potrà anche rappresentare il segnale nel *Tracker* in maniera dettagliata sotto forma di singoli punti per gli *hit* simulati e di *strip* e *pixel* accesi per i gli *hit* ricostruiti, (*RecHit*).

Data la grande quantità di informazioni da rappresentare è previsto che a seconda del livello di zoom si passi da una rappresentazione poco dettagliata a una più dettagliata in maniera automatica. Ad esempio i *RecHit* saranno rappresentati a diversi livello di dettaglio come:

- numero totale di *strip* accese rappresentato come colore del modulo;
- ogni *cluster* (raggruppamento di *strip* accese) rappresentato come un punto al centro della *strip* centrale;
- rappresentazione delle singole *strip* o *pixel* accesi.

Della parte descritta da questo *caso d'uso* è stata realizzata una prima implementazione con un programma in linguaggio java. Questo programma ha la stessa interfaccia che avrà il programma finale in C^{++} e i risultati con esso ottenuti saranno mostrati nel capitolo successivo. Il ritardo con cui è stata rilasciata la versione 7_2_0 di ORCA (28/maggio/2003) e la mancanza di tempo non mi hanno permesso di procedere alla scrittura del codice definitivo in C^{++}.

4.2 La progettazione

Stabilite le funzionalità del software di visualizzazione del *Tracker*, il passo

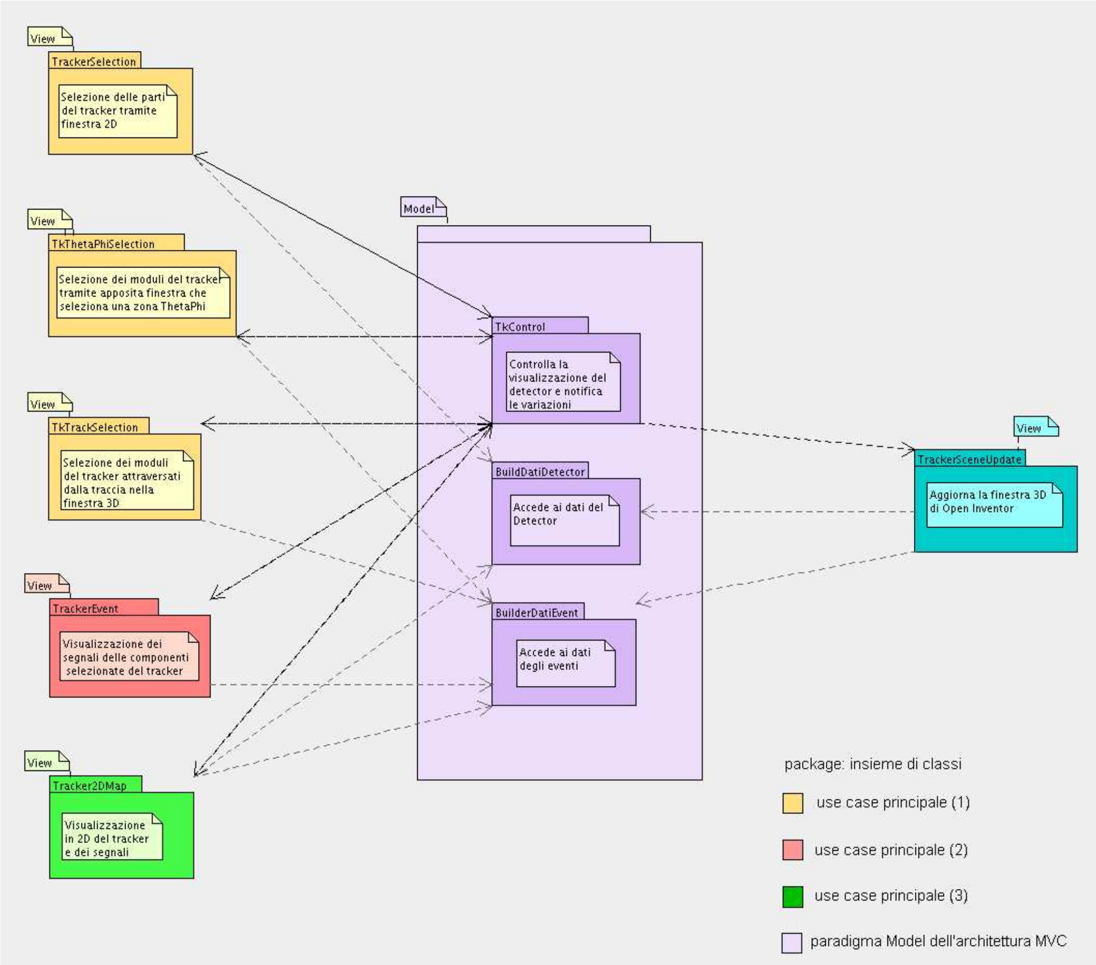

Figura 4.10: *Diagramma dei pacchetti.* Ogni *pacchetto* rappresenta un insieme di *classi* che realizzano uno specifico elemento del prototipo. Tutte le *classi* di questi *pacchetti* saranno inseriti nella cartella `CustomTracker` nel *Sub-System* `Visualisation` di ORCA.

successivo è stato quello di individuare le *classi* che realizzeranno il software.

Dato il numero consistente di *casi d'uso* presenti nel *diagramma dei casi d'uso* (fig.4.1) ci si è resi conto che realizzare un unico *diagramma delle classi* significava realizzare uno schema di grandi dimensioni la cui lettura non sarebbe stata semplice, quindi é stato ideato un *diagramma dei pacchetti* (fig.4.10) che rappresentasse in modo sintetico tutto il progetto software. Questo *diagramma di pacchetti* rappresenta ogni *caso d'uso* principale del *diagramma dei casi d'uso* mediante un *pacchetto*, nonché la comunicazione tra i *pacchetti*. Per ogni *pacchetto*, è stato disegnato poi il corrispondente *diagrama delle classi* con i relativi dettagli. Si è usata la convenzione di colorare ogni *pacchetto* con lo stesso colore dei *casi d'uso* le cui funzionalità saranno realizzate dalle *classi* di quel *pacchetto*, mentre sono stati colorati in viola i *pacchetti* che realizzeranno il sistema di controllo e la struttura dei dati del *Tracker* e degli eventi.

I cinque *pacchetti* sulla sinistra più quello a destra realizzano la parte *View* dell'*architettura* MVC cioé la vista del modello, in particolare si occupano:

TrackerSelection: selezione delle parti del *Tracker* tramite finestre 2D, cioè realizza il primo modo di selezione delle componenti del *Tracker*.

TkThetaPhiSelection: selezione dei moduli del *Tracker* tramite apposita finestra che seleziona una zona θ/ϕ, cioè realizza il secondo modo di selezione delle componenti del *Tracker*.

TkTrackSelection: selezione dei moduli del *Tracker* attraversati dalla traccia nella finestra 3D, cioè realizza il terzo modo di selezione delle componenti del *Tracker*.

TrackerEvent: visualizzazione dei segnali delle componenti selezionate del *Tracker*, cioè realizza il secondo *caso d'uso* principale.

Tracker2DMap: visualizzazione in 2D del *Tracker* e dei segnali, cioè realizza il terzo *caso d'uso* principale.

TrackerSceneUpdate: aggiorna la finestra 3D di OpenInventor e si occupa anche di stampare le informazioni relative agli elementi selezionati.

Il *pacchetto* centrale (in viola), diviso poi in tre sottopacchetti, realizza la parte *Model* dell'*architettura* MVC, che individua la rappresentazione dei dati dell'applicazione e le regole con cui si accede e si modificano tali dati. In particolare questi tre *pacchetti* si occupano di:

TkControl: realizza il "sistema di controllo" che riceve le informazioni dagli altri *pacchetti*, controlla la visualizzazione del rivelatore e notifica le eventuali variazioni avvenute nella selezione delle componenti del *Tracker*.

BuilderDatiDetector: accede ai dati del tracciatore.

BuilderDatiEvent: accede ai dati degli eventi.

Ad ogni *pacchetto* mostrato in figura 4.10 corrisponderà un *diagramma delle classi* che mostrerà in dettaglio tutte le relative *classi*. Questi diagrammi saranno mostrati e descritti qui di seguito.

Per convenzione i nomi delle *classi* iniziano con `Cu` e contengono al loro interno o la sigla `Tk` o la parola `Tracker` per indicare che sono *classi* del *package* `CustomTracker` del *Sub-System* `Visualisation`. Per ogni *classe* sono riportati gli *attributi* e i *metodi* principali che sono stati individuati. Nella fase di sviluppo del software potranno essere aggiunti altri *attributi* o *metodi*.

4.2.1 Il sistema di controllo

La scelta di come realizzare il sistema di controllo è stata molto ponderata, e ha richiesto lo studio di diverse possibilità. L'obiettivo era quello di ottenere un *oggetto* che fosse "unico" e che si occupasse di notificare gli aggiornamenti agli *oggetti* opportuni quando questo fosse necessario. La scelta finale è stata quella di utilizzare due particolari *design pattern*: *Observer*[1] e *Singleton*.

Le due *classi* fondamentali di questo *pacchetto* sono `CuTkObserver` e `CuTkControl` (sono le due *classi* in viola in fig.4.11). La prima è un'interfaccia (*classe base astratta*) che definisce il *metodo* `update(arg:CuTkComposite-Detector`, select:bool) che deve essere ridefinito dalle *classi* che la ereditano (`CuTkLayerSelection, CuTkSelectionBar, CuTkMap2D, ...`).
Le *classi* che ereditano `CuTkObserver` sono le *classi* degli altri *pacchetti* che necessitano di essere avvisate quando c'è stato un cambiamento (per es. devono sapere quando c'è stata una nuova selezione) tra queste, poi ci sono quelle che necessitano anche di comunicare al sistema di controllo eventuali modifiche (per es. nuove selezioni delle componenti del *Tracker* da parte dell'utente).

Il cuore del sistema di controllo è la *classe* `CuTkControl` che gestisce l'intero sistema. Questa *classe* è stata realizzata come *Singleton* in modo che un unico *oggetto* `CuTkControl` viene creato una sola volta all'avvio del software. In questo modo tutti gli *Observer* (*classi* che ereditano `CuTkObserver`) faranno riferimento a quest'unico elemento `CuTkControl` e si evita la creazione di copie di quest'*oggetto* che creerebbero un comportamento anomalo del software.

[1]ORCA ha una sua implementazione del pattern Observer che si è preferito non usare perché sarebbe stato difficile da trasferire nel programma *stand alone*.

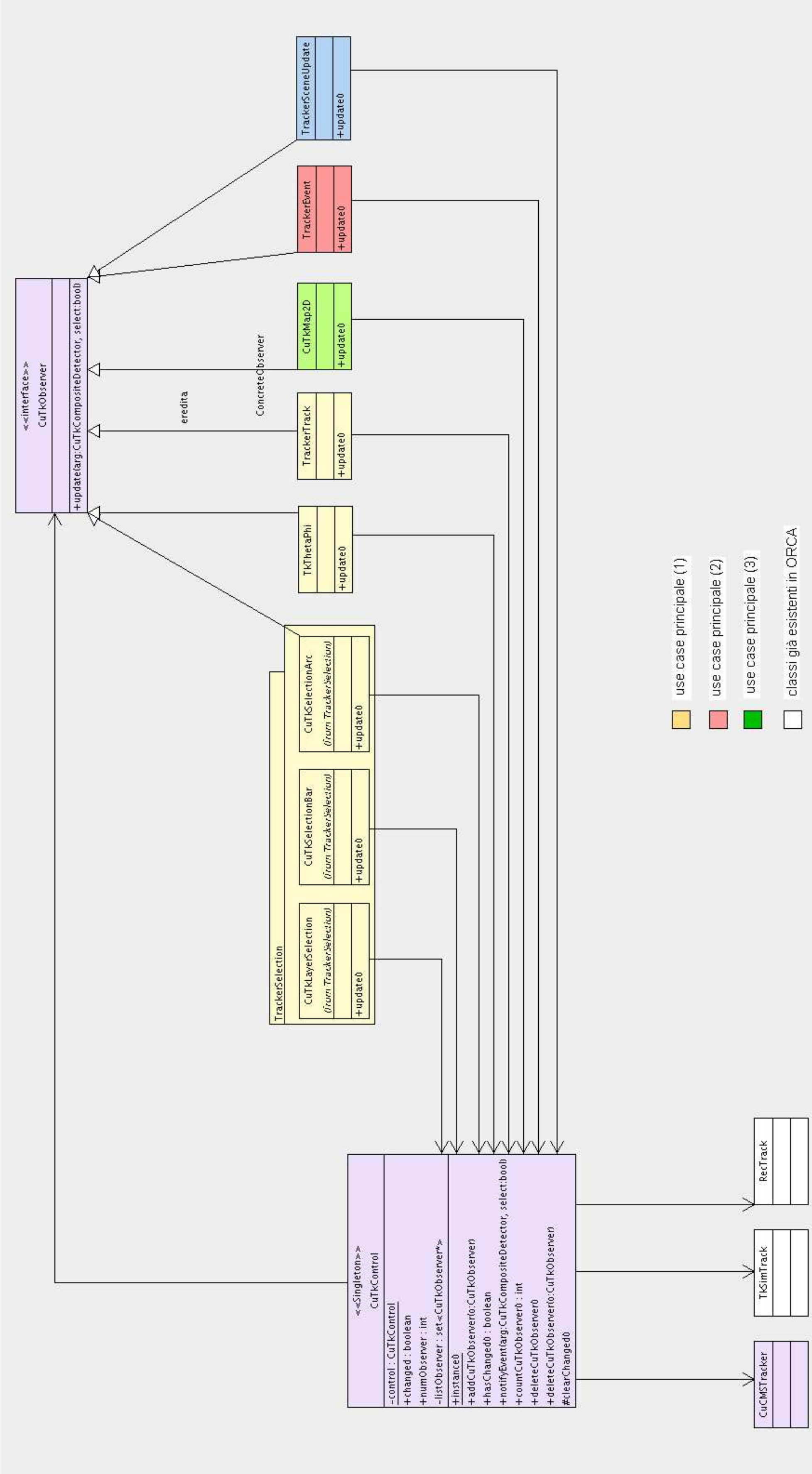

Figura 4.11: *Diagramma delle classi* del "sistema di controllo" che gestisce la comunicazione tra *classi*.

Quando un *Observer* viene creato, questo si registra presso il `CuTkControl` richiamando il *metodo* `addCuTkObserver(o:CuTkObserver)` che, dopo aver controllato che l'*oggetto* non è stato creato precedentemente, lo memorizza nel *set* `listObserver`. Si è preferito usare un *set* (par.3.1.1) al posto di un semplice *vector* perché quest'elemento permette di accedere velocemente ai dati in base al valore di una chiave associata ad ogni elemento in modo univoco, per cui non possono essere inseriti due elementi uguali.
Gli *Observer* che devono comunicare delle modifiche al "sistema di controllo" lo fanno richiamando il *metodo* `notifyEvent(arg:CuTkCompositeDetector, select:bool)` del `CuTkControl`, il quale registra la nuova selezione/deselezione in un'apposita *variabile* della struttura dei dati che rappresenta il *Tracker* e provvede a richiamare il metodo `update(arg:CuTkCompositeDetector, select:bool)` di ciascun *Observer* registrato.

Si noti che nelle chiamate di questi *metodi* `notifyEvent()` ed `update()` viene passato come argomento un *oggetto* che sarà un *puntatore* all'*indirizzo* dell'elemento che è stato selezionato/deselezionato. Quando un *Observer* viene distrutto deve richiamare il *metodo* `deleteCuTkObserver(o:CuTkObserver)`, se invece devono essere distrutti tutti gli *Observer*, conviene usare il *metodo* `deleteCuTkObserver()` che provvederà a cancellarli tutti in modo sequenziale automaticamente.

È stato realizzato in C^{++} prima in un software indipendente (*stand alone*) e poi in ORCA un prototipo delle *classi* `CuTkControl` e `CuTkObserver` per testare la funzionalità del sistema adottato. Il prototipo realizzato funziona proprio come ci si aspettava.

4.2.2 Il modello ad oggetti del Tracker

Le informazioni della geometria del *Tracker* sono contenute nel DDD a cui si accede tramite *classi* che realizzano un modello del *Tracker*. Nella programmazione del modello a oggetti del *Tracker* ci si è trovati di fronte a due problemi:

1. ridescrivere il modello a oggetti perché quello in ORCA non era sufficientemente dettagliato per i seguenti motivi:

   - non presentava un modello dettagliato del *Tracker*, infatti questo viene realizzato attraverso un contenitore (*CMSTracker*) di *layer* che a loro volta sono contenitori di singoli moduli.

   - non metteva a disposizione variabili che potessero essere usate per memorizzare lo stato della selezione delle componenti del *Tracker*, cioé variabili che registrassero se l'elemento è stato selezionato o no dall'utente.

2. permettere l'accesso a questo modello sia dentro che fuori IGUANA.

Va tenuto inoltre presente che questo modello del tracciatore è stato realizzato da altri programmatori per i loro scopi, quindi non ci è consentito modificarlo per adattarlo alle nostre esigenze, anche perchè queste modifiche sarebbero certamente di intralcio al lavoro di quelle persone che hanno realizzato altri moduli del software di ORCA basandosi proprio su questo modello del *Tracker*. Quindi oltre a costruire le *classi* che permettessero di accedere alla struttura dei dati del tracciatore è stato necessario costruire un modello completo del *Tracker* stesso.

La soluzione adottata per risolvere il secondo problema è stata quella di usare il tipo *design pattern Builder* e cioé quella di separare la logica del processo di costruzione del sistema di accesso ai dati dalla costruzione stessa di tale sistema. Come mostrato in figura 4.12 ci sarà un unico *Director* (`CuFullTracker`) che nel processo di accesso ai dati invocherà i *metodi* del *Builder* scelto (`CuTkBuilderINIGUANA` o `CuTkBuilderOUTIGUANA`) secondo la situazione. I due *Builder*, `CuTkBuilderINIGUANA` e `CuTkBuilderOUTIGUANA` dovranno implementare un'interfaccia comune, `CuTrackerBuilder` per consentire al *Director* di interagire con essi. I due *Builder* così consentono di accedere al modello del *Tracker* che ne deve descrivere in dettaglio la struttura.

Il modello del *Tracker* è costituito dai seguenti contenitori (fare riferimento alla figura 4.12:

`CuCMSTracker` composto da elementi del `SubDetector`, qui si può scegliere tra la parte *silicon pixel* e quella *silicon strip* del *Tracker*, la parte *silicon strip* è poi divisa in *sstripInner* e *sstripOuter* formate rispettivamente da TIB e TID la prima e TOB e TEC la seconda.

`CuTkSubDetector` composta da elementi del `PartDetector`, permette di scegliere tra *barrel*, *endCap-Z* ed *endCap+Z*.

`CuTkPartDetector` composto da `Layer`.

`CutkLayer` composto da elementi del `SubLayer`, cioè *ring* e *panel* (petali per i dischi dell'*endcap* e *rod* per i cilindri del *barrel*).

`CuTkSubLayer` composto dai moduli del *Tracker*.

`CuTkModule` che eredita la *classe* `DetUnit` di ORCA.

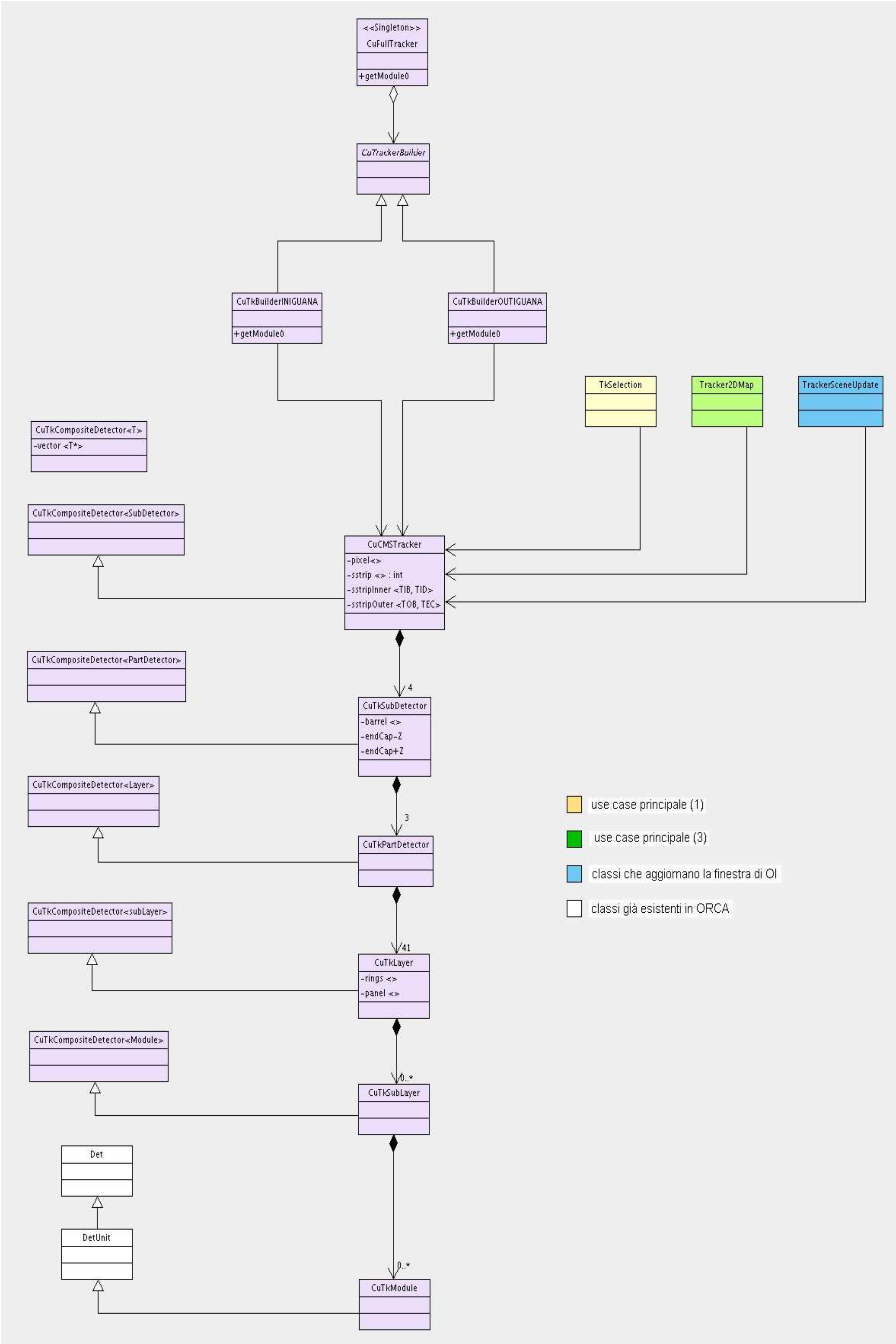

Figura 4.12: *Diagramma delle classi* del sistema di accesso ai dati del *Tracker*. La struttura a oggetti del tracciatore ha come nodo principale la *classe* `CuFullTracker` usata per realizzare il *pattern Builder*

Il tipo di contenitore usato è il `<vector>` che consente di dare un numero d'ordine a ogni *oggetto* contenuto (esattamente come in un *array*). Questa numerazione rispecchia l'ordine degli elementi costitutivi delle varie parti del del rivelatore. Quindi se vogliamo accedere al modulo *i* dell'anello *j* del *layer* *k* del *silicon* lo faremo seguendo nella gerarchia dei contenitori: il *SubDetector* 1 (*silicon*), *PartDetector* 1 (*barrel*), *Layer k*, *SubLayer j* (anello), modulo *i*.

Le *classi* `Det` e `DetUnit`, mostrate di colore bianco in figura 4.12, sono delle *classi* già esistenti in ORCA. La *classe* `Det` è una *classe base astratta* per qualsiasi tipo di rivelatore (*silicon strip detector* e *pixel detector*) essa è estesa dalla *classe base astratta* derivata `DetUnit`. La `DetUnit` fornisce un'interfaccia comune per ottenere le informazioni specifiche del rivelatore e quindi rappresenta il rivelatore stesso.

Questo stesso modello del *Tracker* sarà usato anche nel programma *stand alone* ovviamente qui saranno ricostruite anche le *classi* `Det` e `DetUnit` di ORCA adattandole.

4.2.3 L'accesso ai dati degli eventi

Anche per la visualizzazione degli eventi, come nel caso precedente, occorre accedere ai dati contenuti nel *database* tramite *classi* che realizzano un modello degli eventi stessi.
In questo caso non avevamo bisogno di particolari esigenze, per cui il modello a oggetti degli eventi già presente in ORCA era adatto ai nostri scopi. Infatti, nel prototipo per ORCA si è usato questo modello, invece per il prototipo *stand alone* si costruirà un modello ad oggetti degli eventi che riprodurrà il modello presente in ORCA adattandolo.
Le *classi* (fig.4.13) che permettono di accedere a questo modello ad oggetti degli eventi sono state realizzate seguendo di nuovo la filosofia del *design pattern Builder*, al fine sempre, di separare la logica del processo di costruzione del sistema di accesso ai dati dalla costruzione stessa di tale sistema, così da avere due sistemi di accesso: uno per il prototipo per la versione 7_2_0 di ORCA ed uno per il prototipo *stand alone*.

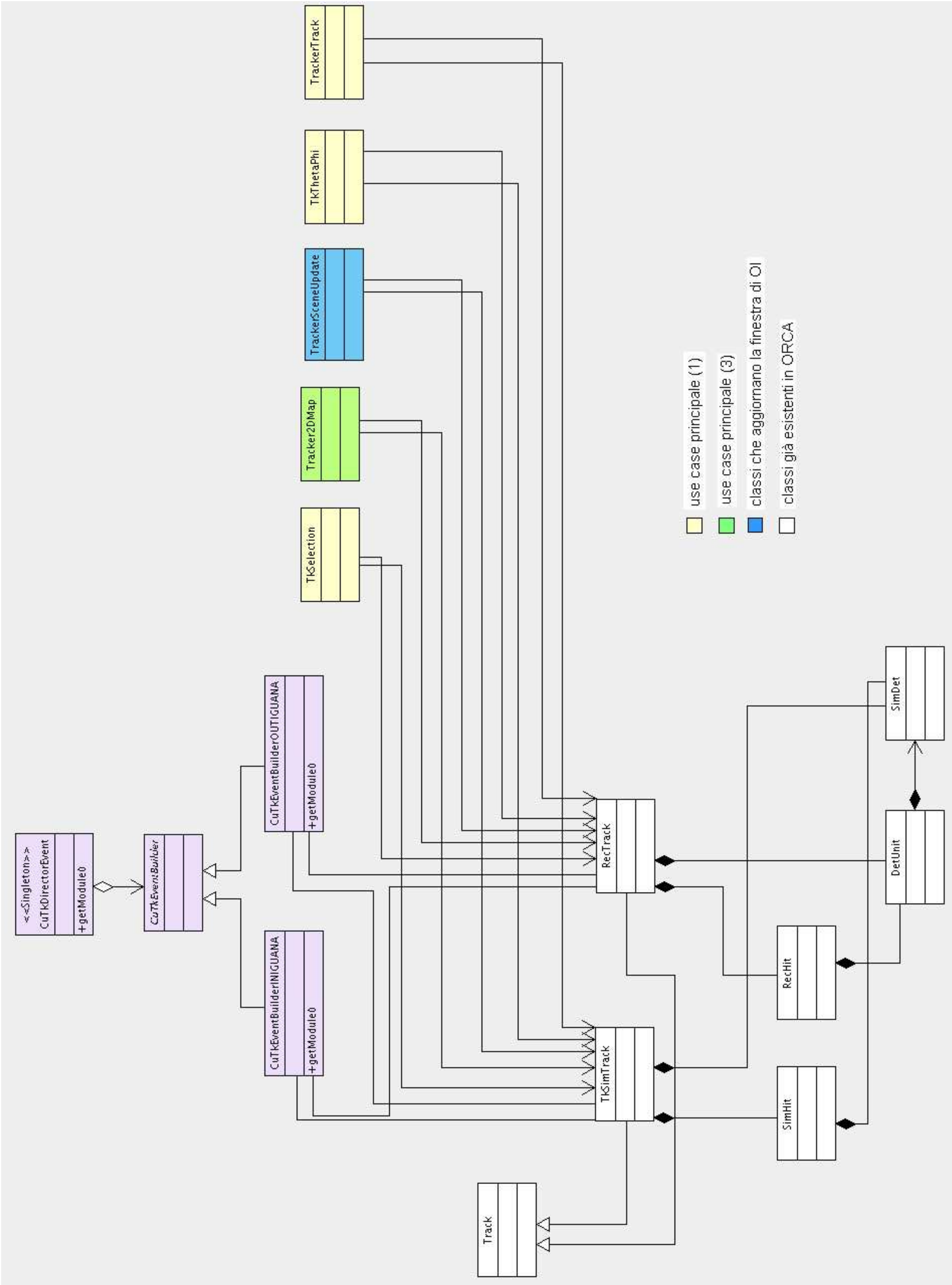

Figura 4.13: *Diagramma delle classi* del sistema di accesso ai dati degli eventi.

Il modello a oggetti degli eventi di ORCA

Ogni elemento sensibile di CMS è rappresentato da un oggetto `DetUnit`, nel caso del *Tracker* questo è il singolo modulo. Per accedere agli *hit* si passa attraverso il `DetUnit` che li ha generati: direttamente per *Digits* e `RecHits`(*hit* ricostruiti) o indirettamente attraverso `SimDet` per i `SimHits` (*hit* simulati sono i dati del Montecarlo). ORCA crea tanti `DetUnit` quanti sono i gli elementi sensibili del rivelatore. Tutte le informazioni associate a

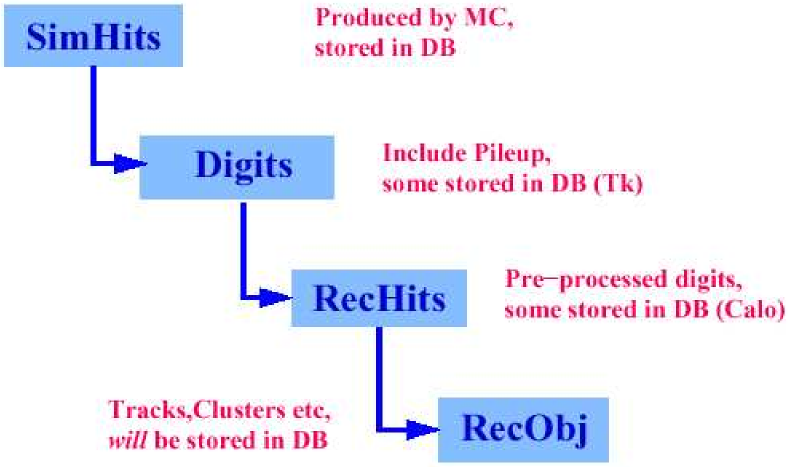

Figura 4.14: Schema del processo di ricostruzione. I `SimHits` vengono trasformati in *Digits*. I *Digits* sono a loro volta trasformati in `RecHits`. Infine dai `RecHits` si ottengono i `RecObj` come tracce, clusters (depositi di energia nel calorimetro), etc. I *Digits* per il *Tracker* rappresentano il segnale di una singola *strip* o *pixel* e sono i dati che saranno letti evento per evento quando comincerà l'acquisizione. I `RecHits` sono invece prodotti da uno speciale software che raggruppa *strip* o *pixel* in *cluster* (raggruppamenti) di *strip*/*pixel*.

un modulo del rivelatore come:

- *hit* simulati (`SimHits`),
- *Digits* (segnale delle singole *strip*/*pixel*),
- *hit* ricostruiti (`RecHits` = *cluster* di *digits*),
- etc.

sono accessibili per mezzo del corrispondente *oggetto* `DetUnit`.

La *classe* (fig.4.13) `RecHit` rappresenta le misure fondamentali del sistema tracciante. Esso contiene tutte le informazioni necessarie per conoscere un *hit* ricostruito nel rivelatore come:

- la posizione globale e locale degli *hit*,
- la direzione globale e locale,
- i corrispondenti errori,
- il modulo del rivelatore in cui è avvenuto l'*hit*.

Tutte le informazioni rilevanti associate con un particolare evento come:

- tracce simulate (*SimTrack*)
- tracce ricostruite (*RecTrack*)

sono accessibili mediante le *classi* `RecTrack` e `TkSimTrack` che ereditano la *classe generale astratta* `Track` (fig.4.13).

Le `RecTrack` e `TkSimTrack` hanno metodi che ritornano contenitori di `RecHits` e `SimHits` rispettivamente. A loro volta ogni *hit* simulato o ricostruito ha un *metodo* `det()` che ritorna la `DetUnit` ad esso collegata.

4.2.4 Le finestre 2D di selezione

Le *classi* di figura 4.15 permetteranno di realizzare la finestra di selezione/deselezione di anelli, *layer*, *rod*, petali o singoli moduli del tracciatore.

Per il software all'interno di ORCA bisognava costruire il ramo "**CustomTracker**" dell'"albero di controllo" con le relative foglie al fine di costruire le voci (fig.4.3) che permetteranno all'utente di accedere alle diverse funzioni del software. In tal caso tramite la selezione della voce "**CustomTracker/TkSelection/TrackerSelection**" l'utente può far aprire proprio le finestre 2D di selezione (fig.4.5, 4.6 e 4.7).

Per costruire il ramo "**CustomTracker**" con i relativi rami principali e foglie è stata creata la *classe* `VisTkTwig` che crea il ramo principale "**CustomTracker**" poi con il file `VisCuTkEventDataProxy.cc` si costruiscono tutti i sottorami di "**CustomTracker**" associando alle varie foglie una *classe* con un nome del tipo `''Vis...Twig''` che si occupa della creazione degli *oggetti* legati a questo quando l'utente clicca su di esso.

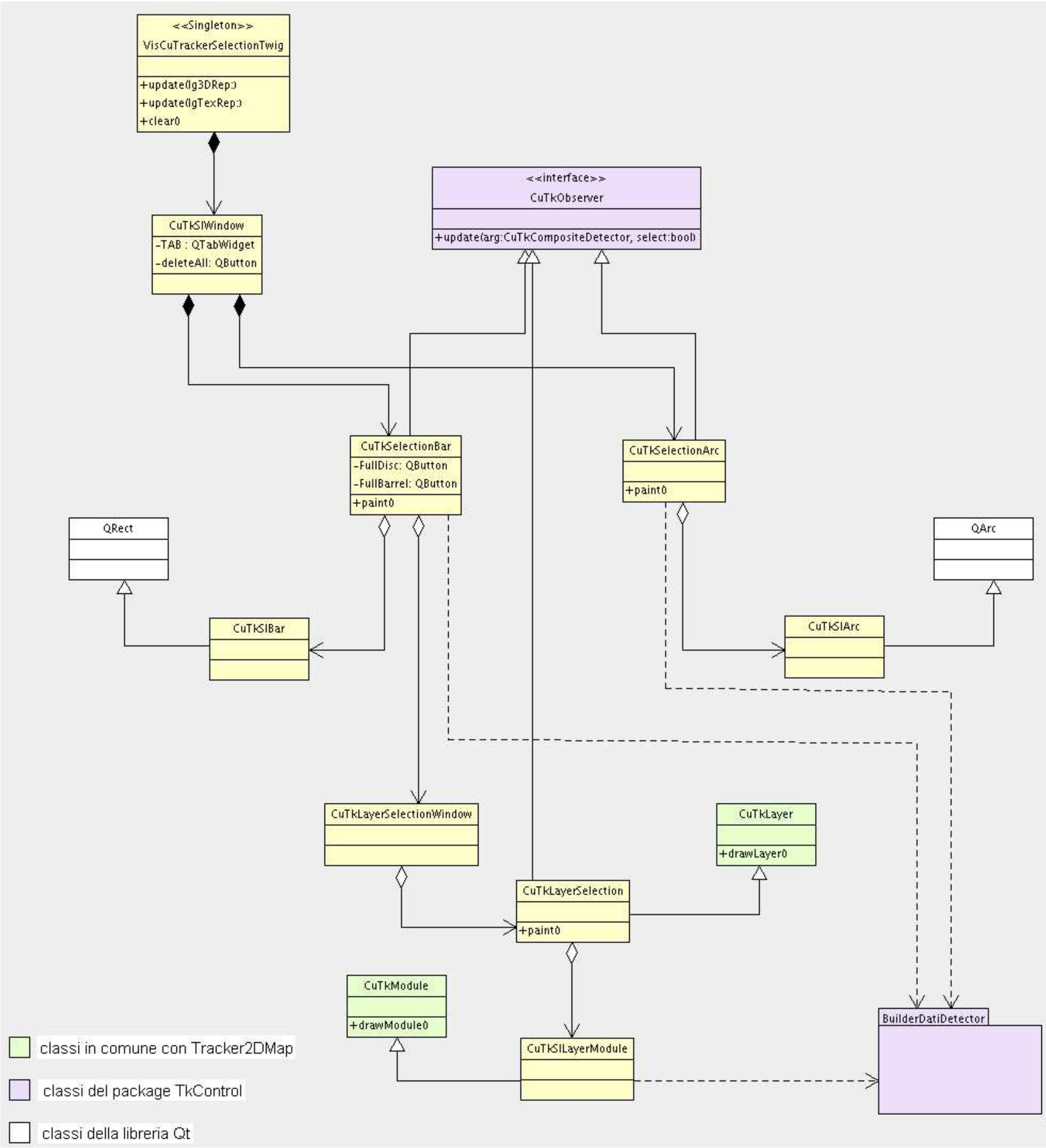

Figura 4.15: *Diagramma delle classi* che realizza le finestre 2D di selezione delle componenti del *Tracker*.

Nel caso delle finestre 2D di selezione, quando l'utente clicca nell'"albero di controllo" sulla voce "**"CustomTracker/TkSelection/TrackerSelection**", la *classe* `VisCuTrackerSelectionTwig` (fig.4.15) ad essa associata provvede a creare l'*oggetto* `CuTkSlWindow` che crea la finestra con il Tab e il tasto "delete". Il contenuto dei 2 Tab viene realizzato dalle *classi* `CuTkSelectionBar` (fig.4.5) e `CuTkSelectionArc` (fig.4.6). Nella finestra del primo Tab se si clicca su uno dei quadratini verdi viene creata la finestra di selezione del singolo modulo (fig.4.7) di un dato *layer* grazie alla *classe* `CuTkLayerSelectionWindow`. Il contenuto di tale finestra viene costruito dalla *classe* `CuTkLayerSelection` che ricostruisce l'intero *layer* implementando la *classe* `CuTkSlLayerModule`. Si noti che `CuTkSelectionBar`, `CuTkSelectionArc` e `CuTkLayerSelection` ereditano la *classe base astratta* `CuTkObserver`, cioè gli oggetti creati da queste *classi* sono degli *Observer* che al momento della creazione si registrano presso il `CuTkControl` (fig.4.11) al fine di ricevere da questo le informazioni di selezioni che l'utente ha fatto in un'altra finestra. Le eventuali selezioni che l'utente effettua da questa finestra sono comunicate usando direttamente il metodo `notifyEvent()` di `CuTkControl`
Le tre *classi* `CuTkSelectionBar`, `CuTkSelectionBar` e `CuTkSlLayerModule` acceddono al *pacchetto* `BuilderDatiDetector`, cioè al sistema di accesso ai dati della geometria del *Tracker*.

4.2.5 La mappa 2D del Tracker

L'*oggetto* che si occupa della creazione della finestra che contiene la mappa 2D del *Tracker*, quando l'utente clicca sul ramo "**CustomTracker/Tk2D-Map/TrackerMap**", è `VisCuTkMapTrackerTwig` (fig.4.16) che implementa l'*oggetto* `CuTkMapWindow`. Il `CuTkMapWindow` tramite *"QSizePolicy"* definisce un'area di disegno molto grande su cui verrà caricata l'intera mappa del *Tracker* grazie a `CuTkMap2D`, inoltre vi sono: la *"scrollbar"* orizzontare e verticale per poter scrollare tutta la mappa, il "tasto save" che permette di azionare la funzione che consente di salvare (`CuTkMapSave`) l'immagine in un file e il "tasto zoom" (`CuTkMapZoom`) per zoomare su una particolare area della mappa che interessa. Le *classi* `VisTkMapRecHitsTwig` e `VisTkMapSimHitsTwig` sono attivate dall'utente quando questi clicca nell'"albero di controllo" (fig.4.3) su "**CustomTracker/Tk2DMap/Rec Hits**" o "**CustomTracker/Tk2DMap/Sim Hits**", queste *classi* comunicano all'*oggetto* `CuTkMap2D` che l'utente ha richiesto la visualizzazione anche degli eventi rispettivamente "Rec Hits" o "Sim Hits".

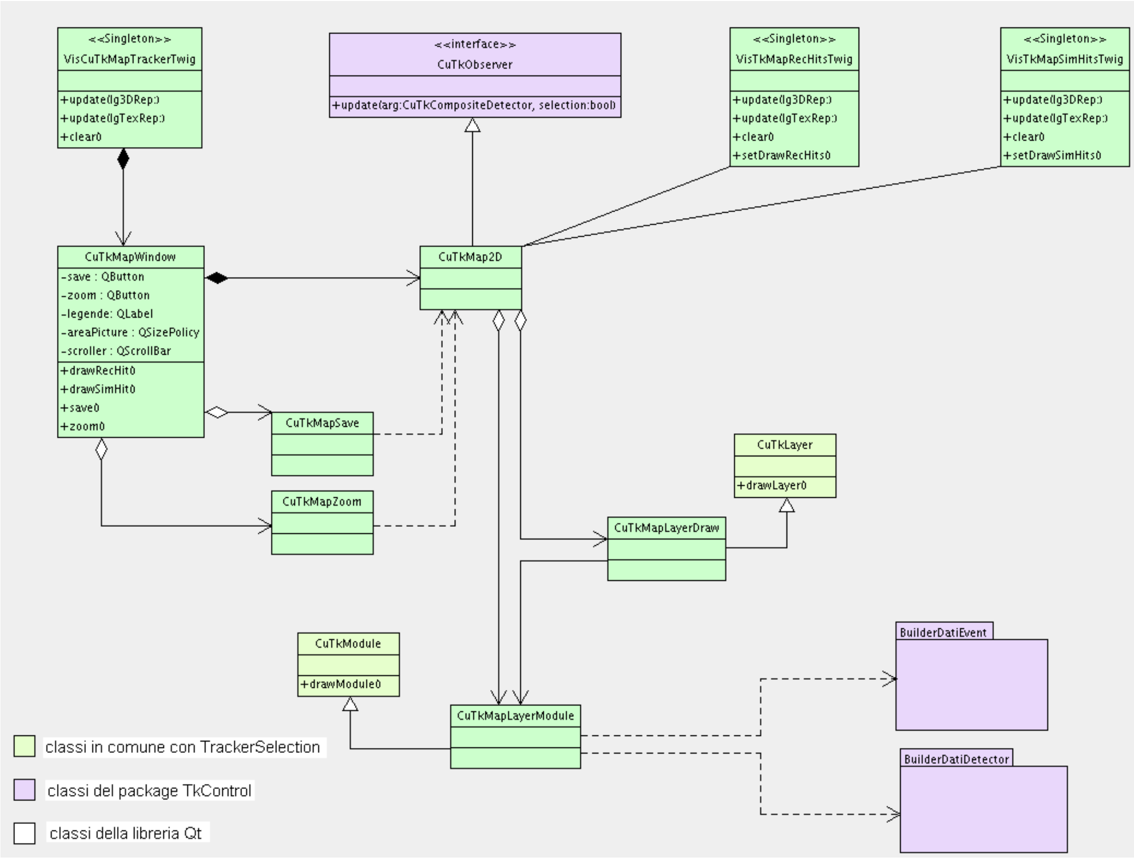

Figura 4.16: *Diagramma delle classi* che realizza la finestra con la mappa 2D del tracciatore.

`CuTkMap2D` fa uso delle *classi* `CuTkMapLayerDraw` e `CuTkMapLayerModule` per la creazione della mappa del *Tracker*, in particolare `CuTkMapLayerModule` è la *classe* che accede ai pacchetti `BuilderDatiEvent` e `BuilderDatiDetector` per accedere ai dati della geometria del *Tracker* e a quelli degli eventi.

Si noti che `CuTkMap2D` eredita dall'interfaccia `CuTkObserver` (fig.4.11), cioè questa *classe* alla creazione si registra presso il `CuTkControl` (Subject) come *Observer* per essere informata delle selezioni che l'utente fa nelle finestre di selezione, infatti tale *classe* ridefinisce il *metodo* `update(arg:CuTkCompositeDetector,` select:bool).

4.3 Realizzazione di un plugin in IGUANA

Il software di visualizzazione in 2D del tracciatore dovendo girare all'interno di IGUANA deve essere realizzato rispettando le regole di costruzione del *plugin*.

A seguito si dà una breve descrizione di come è stato realizzato il *plugin* che ha come funzionalità la visualizzazione 2D del *Tracker*.

Il linguaggio C^{++} non ha istruzioni particolari per creare i *plugin*, per cui gli sviluppatori di IGUANA hanno dovuto individuare un modo per semplificare al massimo la creazione degli stessi. La maniera trovata è quella di sfruttare le possibilità offerte dal preprocessore di C^{++} di estendere il linguaggio definendo dei nuovi comandi sotto forma di *macro*[2]. A questo punto la creazione del *plugin* viene realizzata da file particolari contenenti le informazioni che descrivono lo stesso: questi file è inteso che saranno processati prima che inizi la compilazione e permetteranno di creare automaticamente tutta l'interfaccia dello stesso *plugin*. L'implementazione dei *plugin* prevede anche una fase finale di registrazione dello stesso in un database dei *plugin*. Anche questa operazione è automatica se si rispettano alcune semplici regole descritte nel seguito. Innanzituto è bene sottolineare che ogni *pacchetto* del sottosistema `Visualisation` di ORCA è scritto in modo da creare una singola libreria che potrebbe fornire un singolo *plugin*. In effetti i singoli *pacchetti/librerie* formano un *"superplugin"* disposti in una struttura ad albero che rispecchia quella dell'"albero di controllo". Questi *"superplugin"* possono essere visti e attivati all'inizio quando si fa partire IGUANA che interroga il *database* del *plugin* e ne dà una lista in cui possiamo attivare il *plugin* che ci interessa (nel nostro caso quello chiamato ORCA fig.4.17).

I passi che sono stati seguiti nella realizzazione di questo *plugin*, sono i seguenti [27]:

1. creare un *pacchetto*, nel nostro caso `CustomTracker` con una struttura interna costituita dalle cartelle `interface`, `src`, ... come descritto nel paragrafo 3.3.2.

2. creare nella cartella `interface` un file in cui si dichiararano le *classi* che devono essere descritte e si danno le informazioni necessarie alla *macro* `XTYPEINFO_DECLARE` (file `xtypeinfo.h`).

3. creare nella cartella `src` un file in cui da un lato si definiscono alcune informazioni per le *classi* menzionate nella fase precedente e dall'altro l'*entry point* di definizione del *plugin* (file `plugin.cc`).

[2]Una *macro* associa un nome ad un segmento di codice, il nome così definito può essere usato (più volte) nel corso del programma, la macro costituisce una sorta di convenzione stenografica (accettata dal programmatore) e serve a rendere il programma più compatto (e più leggibile).

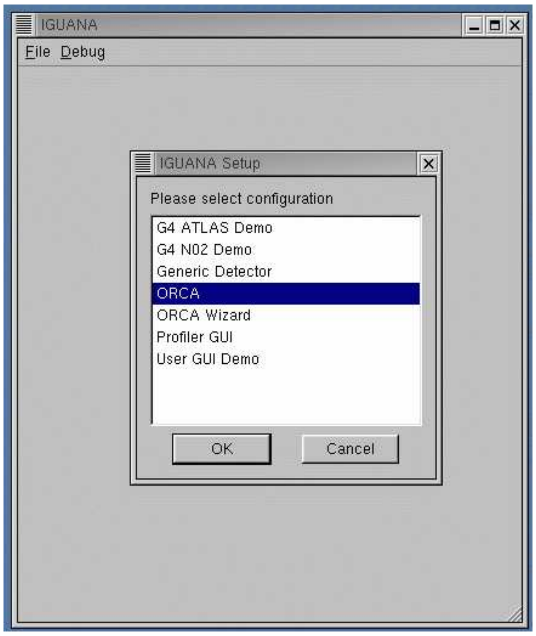

Figura 4.17: Finestra che compare all'avvio di IGUANA per permettere all'utente di caricare il *plugin* che gli interssa. Nel nostro caso interressa il *plugin* ORCA che carica le librerie legate all'"albero di controllo", tra queste c'è quella di `CustomTracker`.

4. modificare il `Buildfile`[3] per poter costruire e registrare il pacchetto come *plugin* all'interno della cartella `lib/arch/iguana-plugins` (cartella del database dei *plugin*) di ORCA.

Il *plugin* realizzato costituisce lo scheletro essenziale del programma dove saranno inserite le *classi* man mano che vengono realizzate. Attualmente, sono presenti le *classi* che costituiscono l'"albero di controllo" (fig.4.3) e le *classi* `CuTkObserver` e `CuTkControl` che hanno permesso di testare il funzionamento del sistema di controllo che si occupa della comunicazione tra le classi di selezione e visualizzazione.

[3]Il Buildfile serve ad "aiutare" il comando `scram build` specificando quali librerie devono essere linkate.

Capitolo 5

Mappe 2D del Tracker

Prima di implementare la finestra della mappa 2D del *Tracker* in ORCA (anche per la mancanza di tempo, dato il ritardo con cui è stata rilasciata la versione 7_2_0 di ORCA), si è deciso di realizzare un prototipo, scritto con il linguaggio di programmazione *java*, al fine di ideare e testare le possibili mappe che poteva aver senso realizzare.

La figura 5.1 mostra uno *screenshot* dell'applicazione in java che è stata realizzata, questa realizza all'incirca la stessa interfaccia del progetto della mappa 2D. Si può notare l'area di disegno molto grande (1400×2800 *pixel*) per contenere la mappa completa del *Tracker* che può essere visionata per mezzo di *scrollbar*, il tasto per ingrandire (zoom) sulla mappa e quello per salvare su file l'immagine della mappa mostrata nell'area grafica. Nel caso di questa applicazione ho dovuto inserire in alto anche un *menù a tendina* per permettere di scegliere se visualizzare un dato *layer* o la mappa completa. Invece in ORCA la scelta delle componenti da visualizzare avverrà, come già detto, tramite le finestre di selezione appositamente progettate.

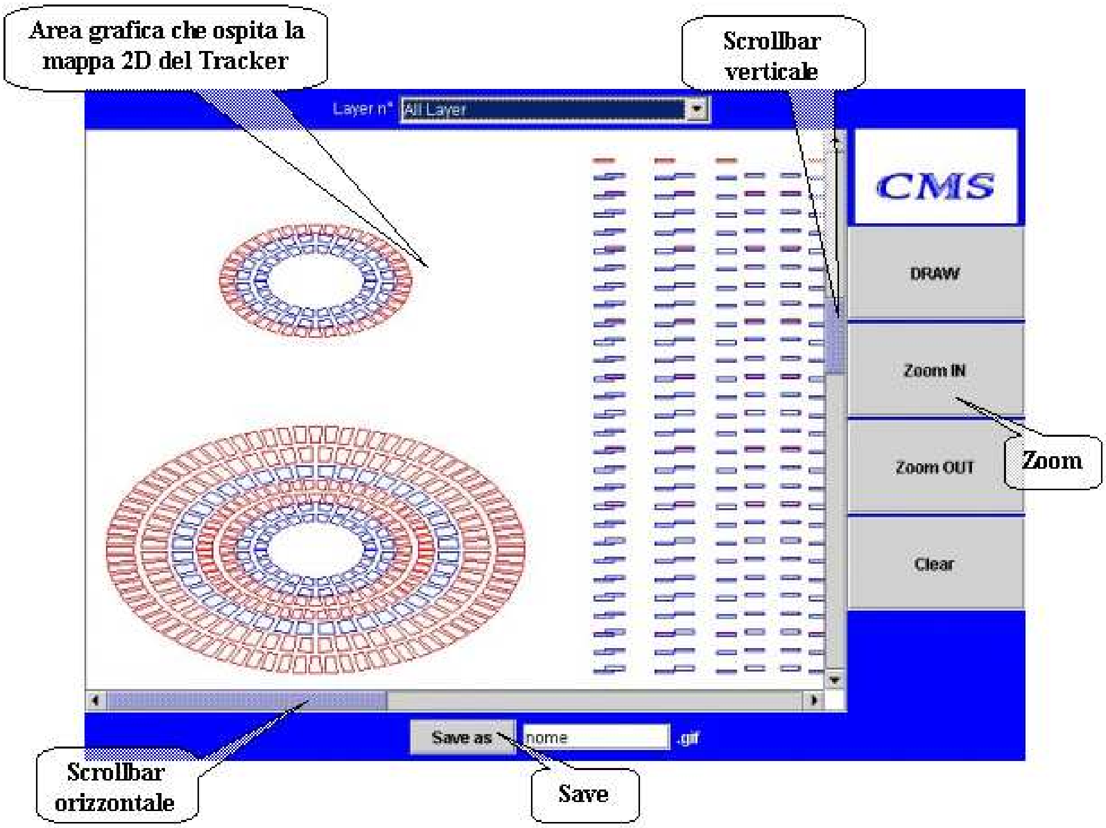

Figura 5.1: *Screenshot* dell'applicazione in java realizzata per creare dei prototipi delle mappe del *Tracker*.

5.1 Visualizzazione del Tracker in 2D

Le figure che seguono, realizzate col programma java, fanno parte di uno studio per idividuare la maniera migliore di rappresentare le mappa al fine di avere un quadro globale della geometria e dei difetti dell'apparato ("*strip* difettose"), da usare nel monitoraggio dello stesso, ed anche per poter visualizzare e controllare gli eventi.

La prima mappa (fig.5.2) mostra la geometria del *Tracker*, in essa sono indicati in rosso i moduli singoli e in blu quelli doppi. La prima colonna è costituita dai *layer* dell'*endcap* -z, questi sono disposti in modo tale che il primo *layer*, partendo dal basso, corrisponde a quello più esterno del tracciatore, di seguito sono posti man mano i *layer* più interni fino ad arrivare a quelli del *Silicon Pixel Tracker* (SPT) (i primi due in alto).

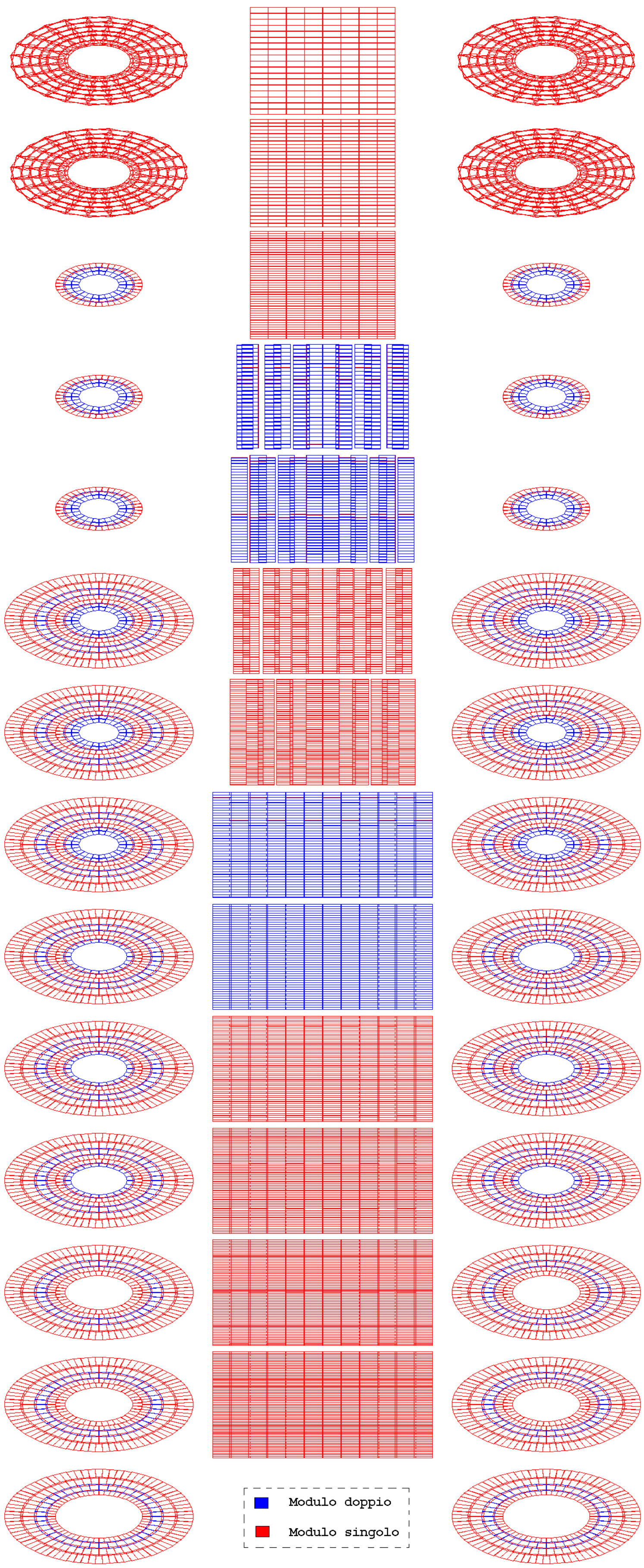

Figura 5.2: Mappa del *Tracker* in cui nella prima colonna sono disposti i dischi dell'*endcap* -z, nella seconda i cilindri del *barrel* e nell'utima i dischi dell'*endcap* +z. In rosso sono indicati i moduli singoli e in blu quelli doppi.

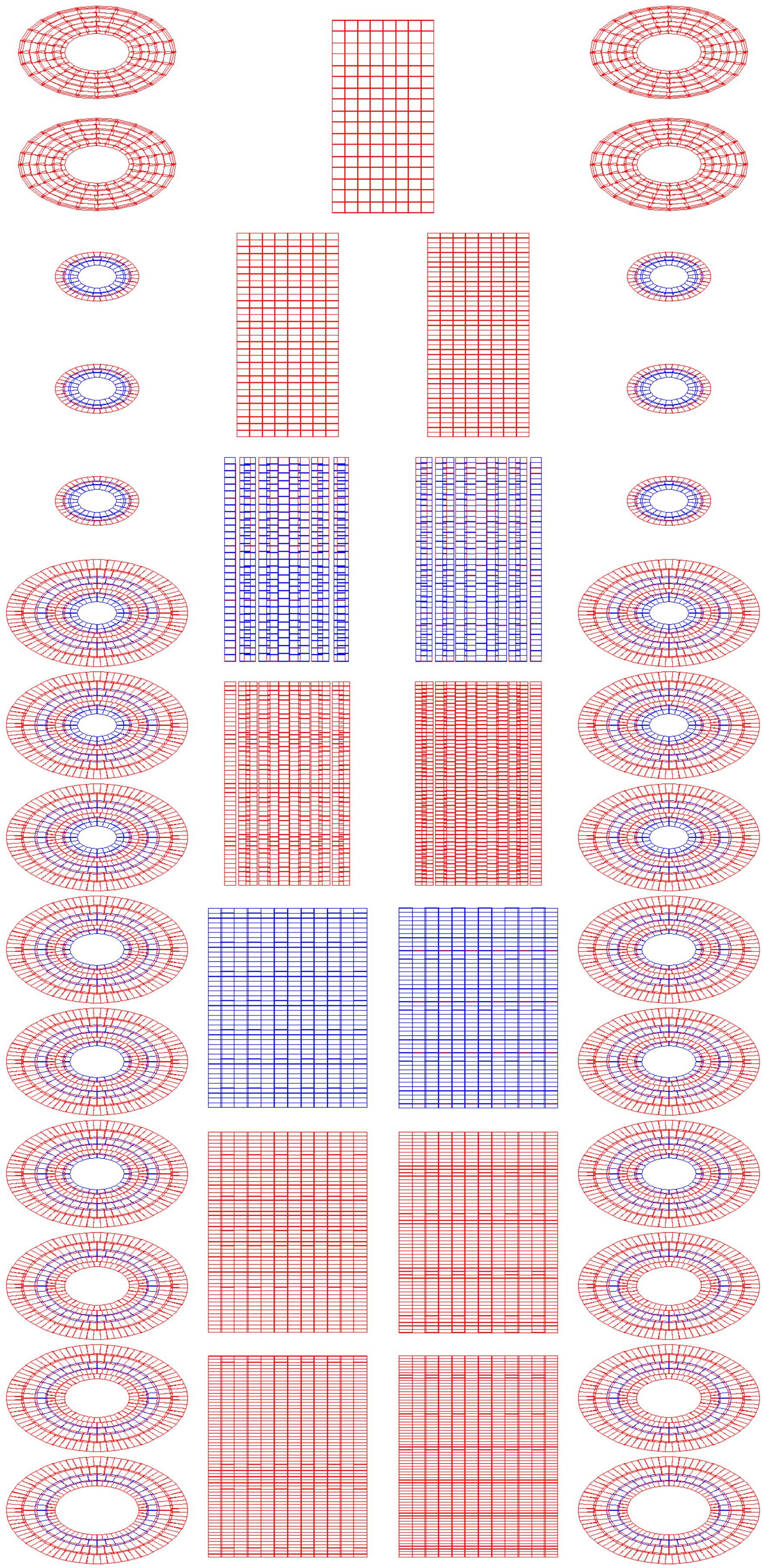

Figura 5.3: Mappa del *Tracker* in cui nella prima colonna sono disposti i dischi dell'*endcap* -z, nelle due colonne centrali i cilindri del *barrel* e nell'utima i dischi dell'*endcap* +z. In rosso sono indicati i moduli singoli e in blu quelli doppi.

In modo analogo nella colonna all'estrema destra sono disegnati i *layer* dell'*endcap* +z. Nella colonna centrale, invece, sono mostrati i *layer* del *barrel*, il primo in alto è quello più interno del SPT, seguono poi gli altri fino all'ultimo che corrisponde al cilindro del *barrel* più esterno.

La disposizione dei *layer* usata in questa mappa (fig.5.2), seppur permette di mantenere una corrispondenza con la reale disposizione geometrica rispettivamente dei dischi e dei cilindri, non permette di distinguere molto bene i singoli moduli dei cilindri del *barrel* (nella mappa completa stampata su foglio A4), come invece avviene per i dischi dell'*endcap*. Quindi si è preferito usare la disposizione mostrata in figura 5.3 nella quale i *layer* dell'*endcap* sono rimasti nella stessa posizione, mentre i cilindri del *barrel* sono stati disegnati su due colonne occupando così uno spazio doppio rispetto al caso precedente.

Si tenga presente che la mappa non è fedele alla realtà dato che la parte del rivelatore a *pixel* è ingrandita e meno distante rispetto al rivelatore a microstrisce di silicio.

Nella figura seguente è riportata la stessa mappa di figura 5.4 ruotata di 90° in senso orario. In essa sono state contornate le diverse sottosezioni di

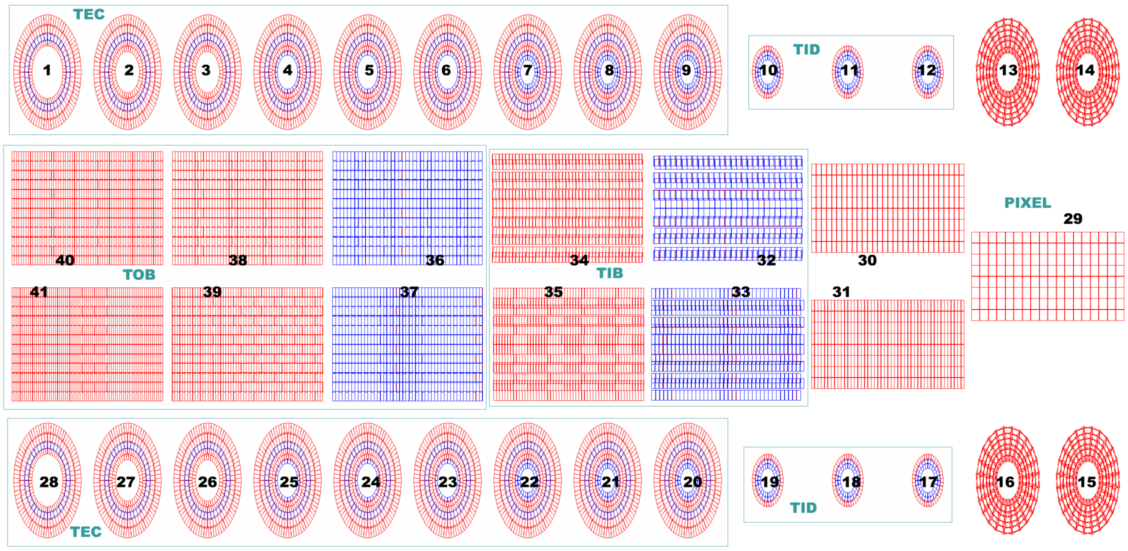

Figura 5.4: Mappa ruotata in cui sono indicate le diverse sottosezioni di cui è costituito il rivelatore a microstrisce al silicio e i numeri attribuiti ad ogni *layer*.

cui è costituito il rivelatore a microstrisce al silicio per semplificare l'identificazione delle varie parti del rivelatore, sono inoltre riportati i numeri che per convenzione sono attribuiti ai 41 *layer* per poterli identificare rapidamente. I *layer* dell'*endcap* sono disegnati proiettando i moduli nel piano xy, mentre quelli del *barrel* sono disegnati tagliando il cilindro lungo un asse parallelo all'asse di simmetria (ϕ=0) e aprendolo su di un piano. Ogni colonna corrisponde perciò ad un anello e i moduli in blu sono stereo. Poichè il disegno

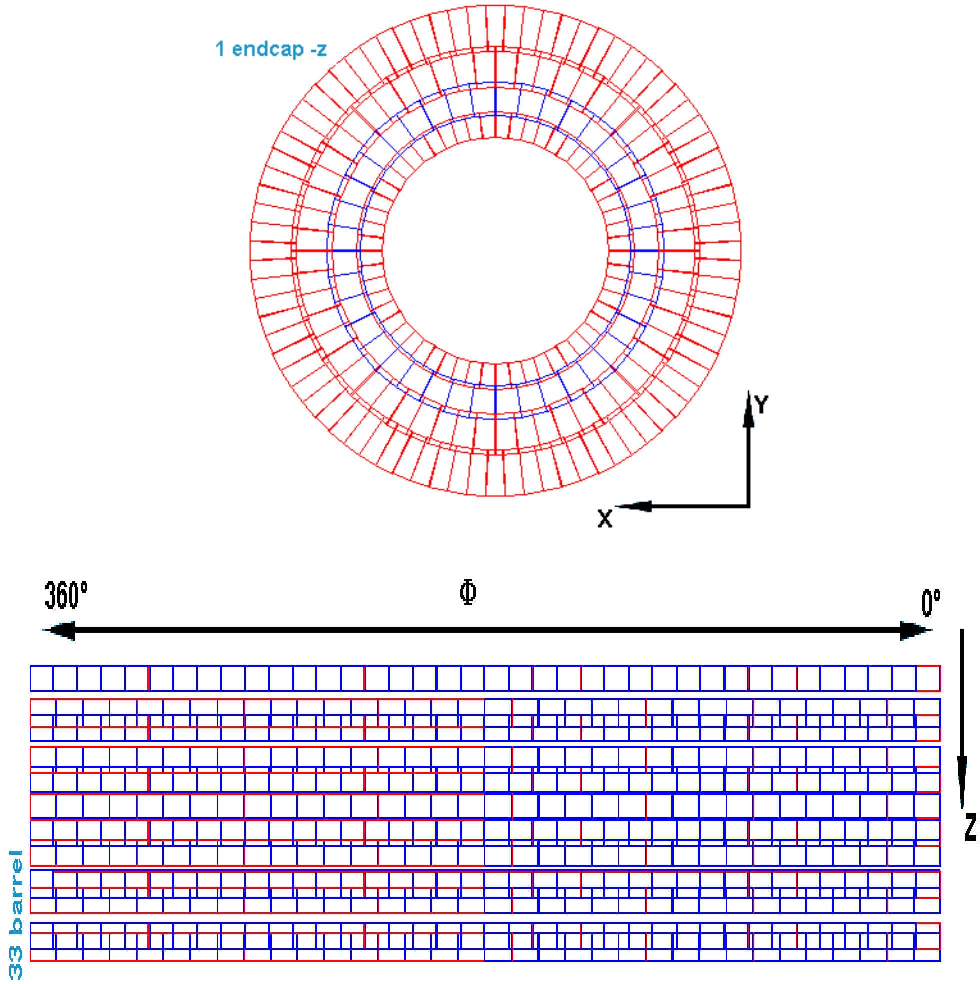

Figura 5.5: Esempio di ingrandimento di un disco dell'*endcap* e un cilindro del *barrel* in cui viene riportato anche il sistema di riferimento.

è stato ottenuto con una proiezione dei *layer* in un piano, la mappa mostra i moduli sovrapposti proprio come lo saranno nella realtà.

Una variante di questa mappa è quella mostrata negli ingrandimenti in figura 5.6, nella quale i moduli sono stati traslati nel piano in cui sono stati proiettati al fine di separarli (eliminando la sovrapposizione) e poterli così distinguere nettamente l'uno dall'altro. I moduli dei *layer* degli *endcap* sono stati traslati radialmente, mentre quelli del *barrel* lateralmente. Poiché la

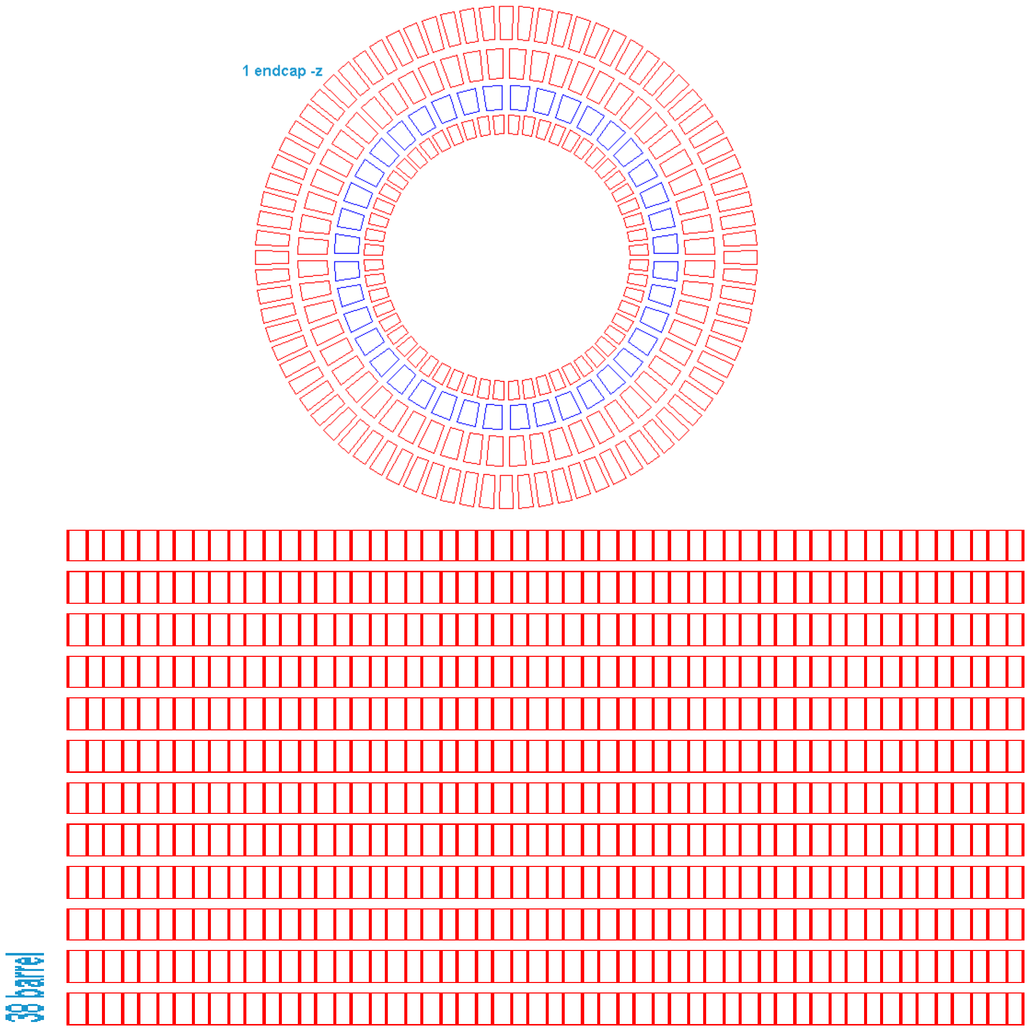

Figura 5.6: Esempio di ingrandimento di un disco dell'*endcap* e un cilindro del *barrel* in cui i moduli sono disegnati in modo non sovrapposto, eccetto quelli doppi che sono colorati di blu.

traslazione avviene moltiplicando le coordinate che individuano il centro del modulo per un'opportuna costante, questo fa si che i moduli doppi, quelli in configurazione *"back to back"*, restano sovrapposti, per cui servono altri espedienti per poterli distinguere, in questo caso sono stati colorati di blu.

Un primo obiettivo che ci si è posti è quello di fornire con queste mappe uno strumento di ausilio per il monitoraggio dell'apparato, di cui ora vediamo un esempio.

La figura 5.7 mostra il modo proposto per mappare le “*strip* difettose”

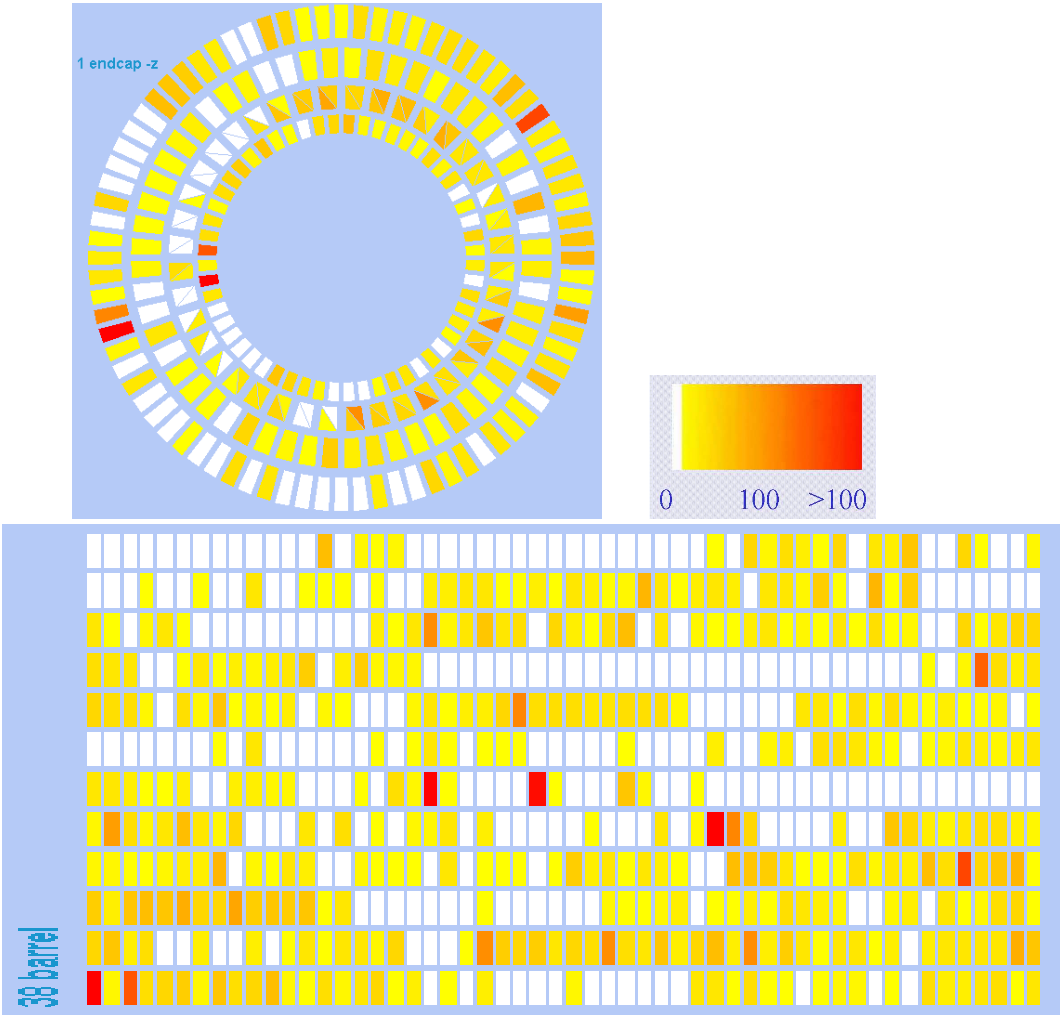

Figura 5.7: Esempio di ingrandimento di un disco dell’*endcap* e un cilindro del *barrel* in cui i moduli sono colorati in base al numero di “*strip* difettose”.

dei moduli del *Silicon Strip Tracker* (SST), che consiste nel colorare i moduli in base al numero di “*strip* morte” (in bianco i moduli con nessuna “*strip* difettosa”, e con una gradazione di colore che va dal giallo al rosso quelle con “*strip* difettose” da 0 a 100, in rosso un numero di “*strip* difettose” maggiore o uguale a 100). In questo caso colorando i moduli anche all’interno è convenuto usare il disegno in cui i moduli sono traslati (e quindi separati) nel piano di proiezione. Restava poi il problema di come rappresentare i due valori del numero di "*strip* difettose" dei moduli in configurazione "*back to back*". Per

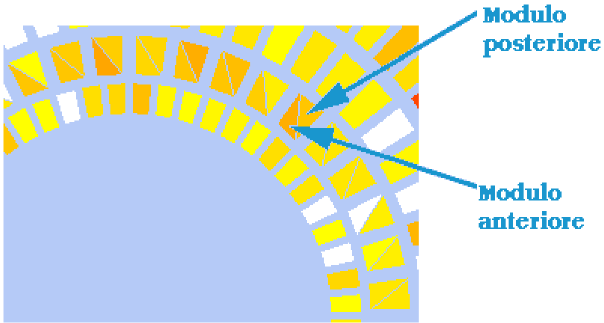

Figura 5.8: Esempio di ingrandimento di alcuni moduli doppi.

risolvere questo problema è stato proposto di dividere i moduli doppi lungo la diagonale e colorare i due spicchi così ottenuti con il numero di "*strip* difettose" rispettivamente del primo e del secondo modulo (fig.5.1).

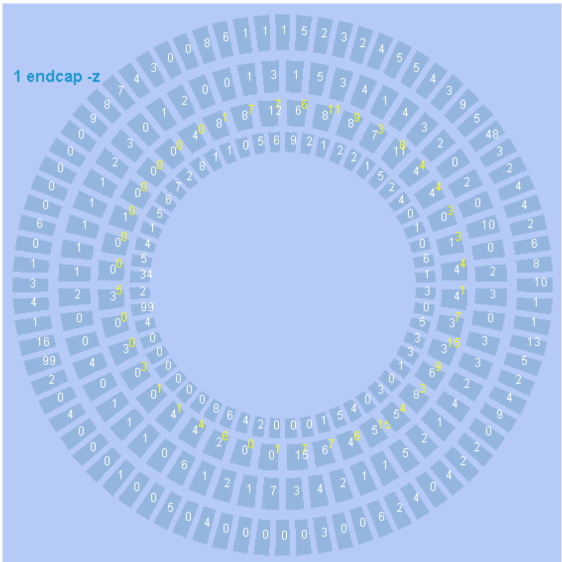

Figura 5.9: Esempio di ingrandimento di un disco dell'*endcap* in cui sui moduli sono indicati il rispettivo numero di *strip* difettose. Per quelli doppi sono indicati due valori, quello in giallo è per il modulo posteriore.

Con la stessa tecnica utilizzata per rappresentare le "*strip* difettose" dei moduli potrebbero essere rappresentate altre caratteristiche di questi come per esempio la loro temperatura. In alternativa l'utente potrà passare a una mappa come quella di figura 5.1, nella quale sui moduli sono indicati il rispettivo numero di "*strip* difettose". Nel caso dei moduli doppi sono riportati due valori dei quali quello in giallo è per il modulo posteriore nella configurazione "*back to back*".

Un altro modo per rappresentare le "*strip* difettose" potrebbe essere quello di disegnarle con delle linee all'interno del modulo nel punto in cui sono allocate. Di questo tipo alternativo di rappresentazione delle "*strip* difettose" parleremo in seguito quando vedremo la visualizzazione dei *RecHit* (fig.5.13).

5.2 Visualizzazione degli eventi con una mappa del Tracker in 2D

Oltre alle mappe appena viste che permettono di mostrare alcune caratteristiche dei moduli rivelatori che costituiscono il *Tracker*, sono state costruite altre mappe che permettono di visualizzare anche gli eventi (*SimHit* e *RecHit*). Queste mappe saranno ora descritte mentre, nel prossimo capitolo si mostrerà come queste permettono di avere una visione d'insieme di un dato evento che si vuole analizzare permettendo anche di controllare la ricostruzione delle tracce.

Un primo modo che può essere usato per rappresentare sia gli *hit* simulati che quelli ricostruiti di un dato evento è quello di colorare i moduli in base al numero di *hit* (simulati o ricostruiti) che sono avvenuti in quel modulo. Un esempio è quello di figura 5.10 nella quale sono colorati di bianco i moduli con nessun *SimHit* (analogamente per i *RecHit*) e gli altri con una gradazione di colore che va dal giallo al rosso in base al numero di *SimHit*.

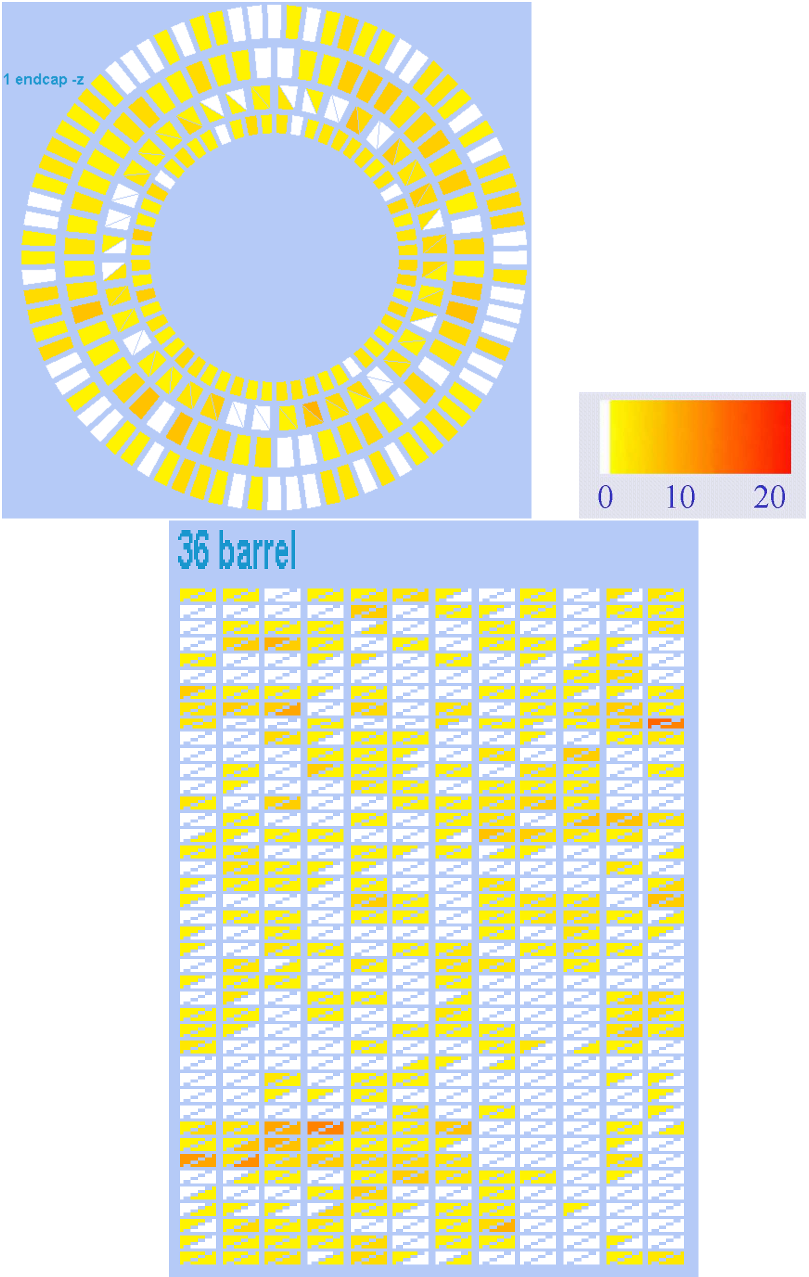

Figura 5.10: Esempio di ingrandimento di un disco dell'*endcap* e un cilindro del *barrel* colorati in base al numero di *hit* simulati avvenuti in quel modulo.

Nel caso dei *SimHit* poi, un modo per rappresentarli in dettaglio potrebbe essere quello proposto in figura 5.11 in cui con dei puntini sui moduli sono indicati gli *hit* simulati. Il disegno degli *hit* simulati è stato ottenuto proiet-

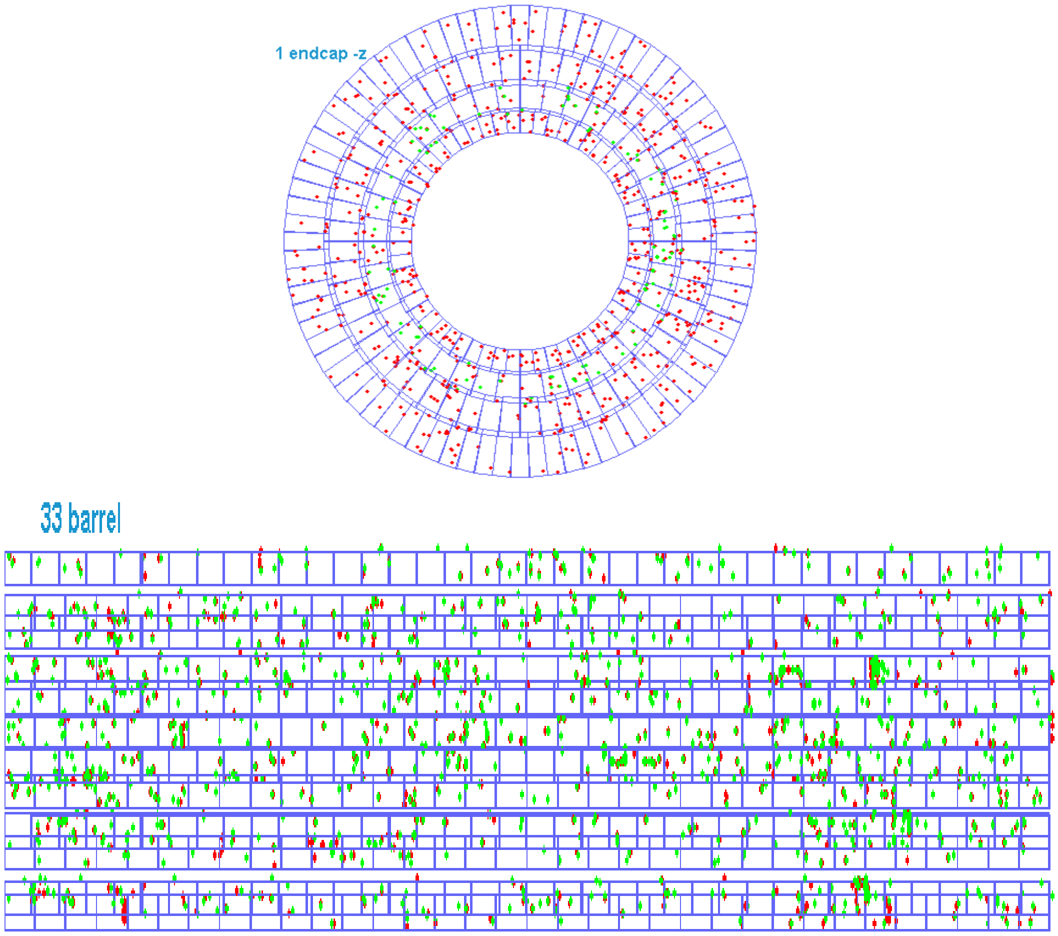

Figura 5.11: Esempio di ingrandimento di un disco dell'*endcap* e un cilindro del *barrel* in cui all'interno dei moduli sono disegnati con dei puntini gli *hit* simulati. In verde sono disegnati gli *hit* simulati che cadono nel modulo posteriore dei moduli doppi.

tando le coordinate nello spazio dei *SimHit* nel piano xy per quelli relativi agli *endcap*, nel piano zϕ per quelli relativi al *barrel* usando lo stesso algoritmo usato per riprodurre nel piano i moduli del *Tracker*.
Per distinguere gli *hit* simulati che cadono nei moduli doppi si usano due colori: in rosso quelli che appartengono al modulo anteriore e in verde quelli doppi.

I *RecHit*, in modo simile ai *SimHit*, possono essere rappresentati come in figura 5.12 in cui con dei puntini al centro dei moduli sono indicati gli *hit* ricostruiti. Il disegno degli *hit* ricostruiti è stato ottenuto proiettando le

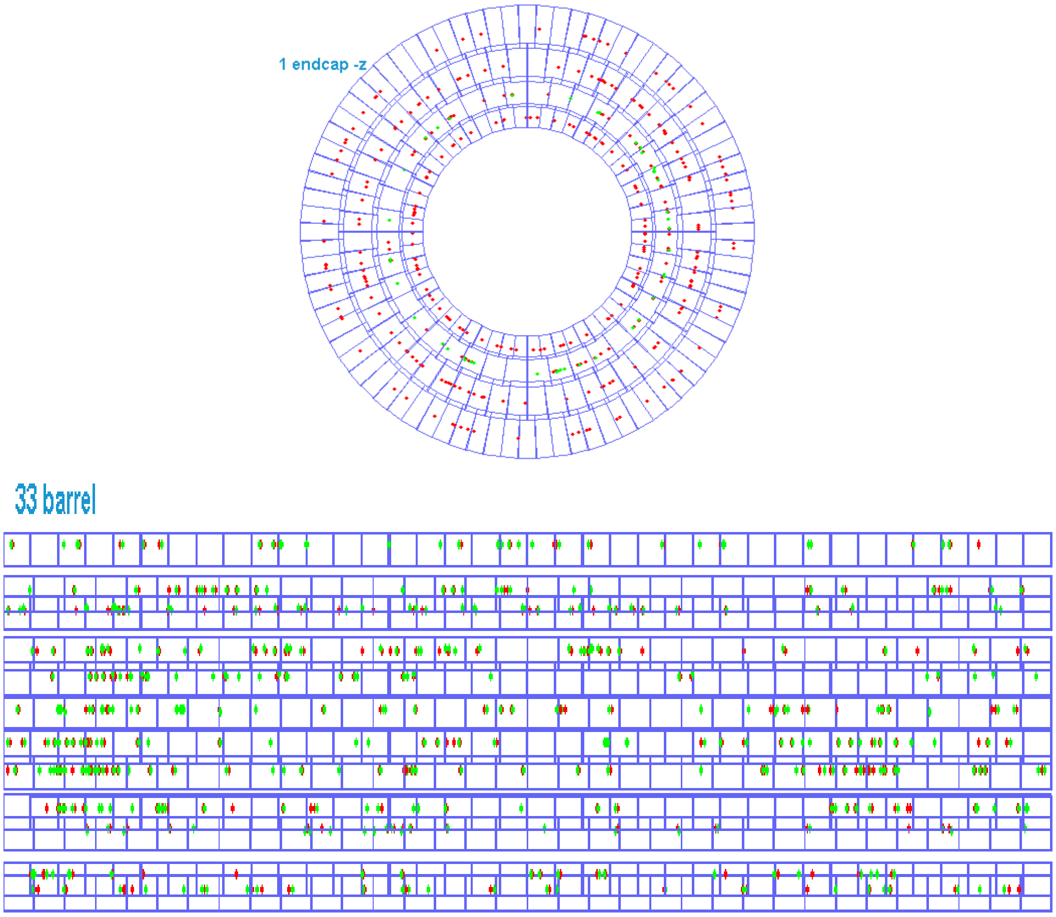

Figura 5.12: Esempio di ingrandimento di un disco dell'*endcap* e un cilindro del *barrel* in cui al centro dei moduli sono disegnati con dei puntini gli *hit* ricostruiti.

coordinate nello spazio del centro della *strip* o del *pixel* acceso dal *RecHit* nel piano xy per quelli relativi agli *endcap*, nel piano $z\phi$ per quelli relativi al *barrel* usando lo stesso algoritmo usato per riprodurre nel piano i moduli del *Tracker*. Per distinguere gli *hit* ricostruiti che cadono nei moduli doppi si usano sempre due colori: il rosso e il verde.

Per i *RecHit* si possono poi rappresentare anche le *strip* o i *pixel* accesi. le *strip* accese si possono disegnare come linee mentri i *pixel* come punti. Il disegno degli *hit* ricostruiti è stato ottenuto questa volta usando, anziché le coordinate globali, quelle locali riferite al singolo modulo.

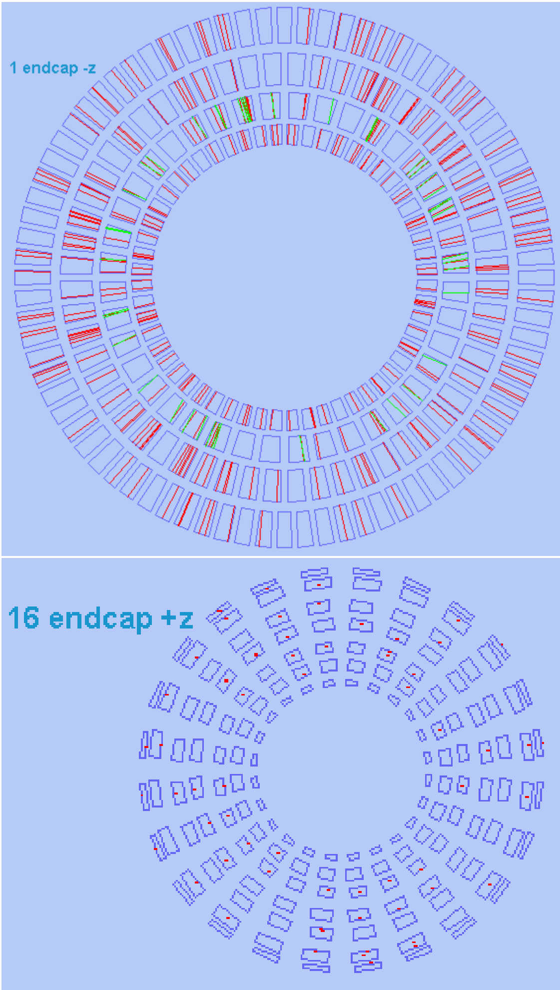

Figura 5.13: Esempio di ingrandimento di due dischi dell'*endcap* il primo di tipo *silicon strip* e il secondo di tipo *pixel* in cui sono disegnati le *strip* accese del SST con delle linee e i *pixel* accesi con dei puntini. In verde, nella prima figura, sono disegnate le *strip* accese relative al modulo posteriore dei moduli doppi.

Per distinguere gli *hit* ricostruiti che cadono nei moduli doppi si usano due colori: in rosso quelli che appartengono al modulo anteriore e in verde quelli che appartengono al modulo posteriore. In questo caso vedendo le intersezioni tra le *strip* è possibile risalire al punto di passaggio della traccia nel sensore.

Si noti che nelle varie mappe che sono state ideate, di volta in volta è stato fatto uno studio sui colori da usare in modo da visualizzare meglio gli eventi.

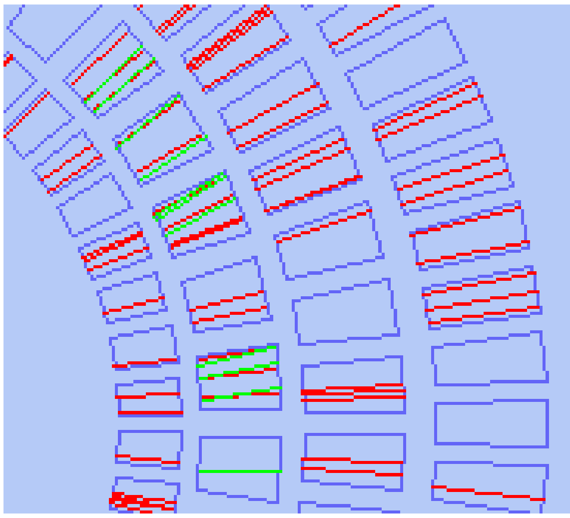

Figura 5.14: Ingrandimento di alcuni moduli di un disco dell'*endcap* di tipo *silicon strip*.

Nelle mappe che visualizzano gli *hit* simulati o ricostruiti si può pensare di disegnarle imponendo delle condizioni: per esempio richiedendo di disegnare gli *hit* simulati o ricostruiti relativi a determinate tracce cui l'utente è interessato.

Capitolo 6

Esempio di uso delle mappe 2D

In questo capitolo vedremo un utilizzo delle mappe del *Tracker* ideate per controllare la ricostruzione delle tracce e per mappare le caratteristiche dell'apparato.

Si è utilizzato il *database* della produzione disponibile su un *cluster* di macchine *lxplus* del CERN. In particolare sono stati generati e ricostruiti eventi di interazione p-p ad LHC con produzione del bosone di Higgs nel canale:

$$H \to ZZ \to ee\mu\mu.$$

Tale canale è uno dei più promettenti ad LHC per la scoperta del bosone di Higgs per masse comprese tra 114 e 500 GeV/c^2.

6.1 Visualizzazione di un evento

Ora saranno presentate una serie di immagini che sono state fatte con la versione 4.2.2 di IGUANA all'interno del pacchetto `Visualisation` di ORCA versione 7_2_0.
La figura 6.1 mostra in 3D un evento simulato di interazione p-p del suddetto database con *pile-up* a bassa luminosità con m_H=300 GeV/c^2. La figura 6.2 mostra invece lo stesso evento nel piano xy. I punti bianchi rappresentano gli *hit* simulati. Le tracce simulate sono state raffigurate con un colore diverso per rappresentare i diversi tipi di particelle prodotte nell'interazione. In particolare in rosso sono rappresentati i muoni, in verde gli elettroni, in blu i pioni, le altre particelle di colore celeste.
La figura 6.3 mostra l'evento sempre in 3D da un'altra angolazione, si noti come i puntini si concentrano proprio in corrispondenza dei *layer*.

Invece, la figura 6.4 mostra la proiezione di questo evento nel piano yz, an-

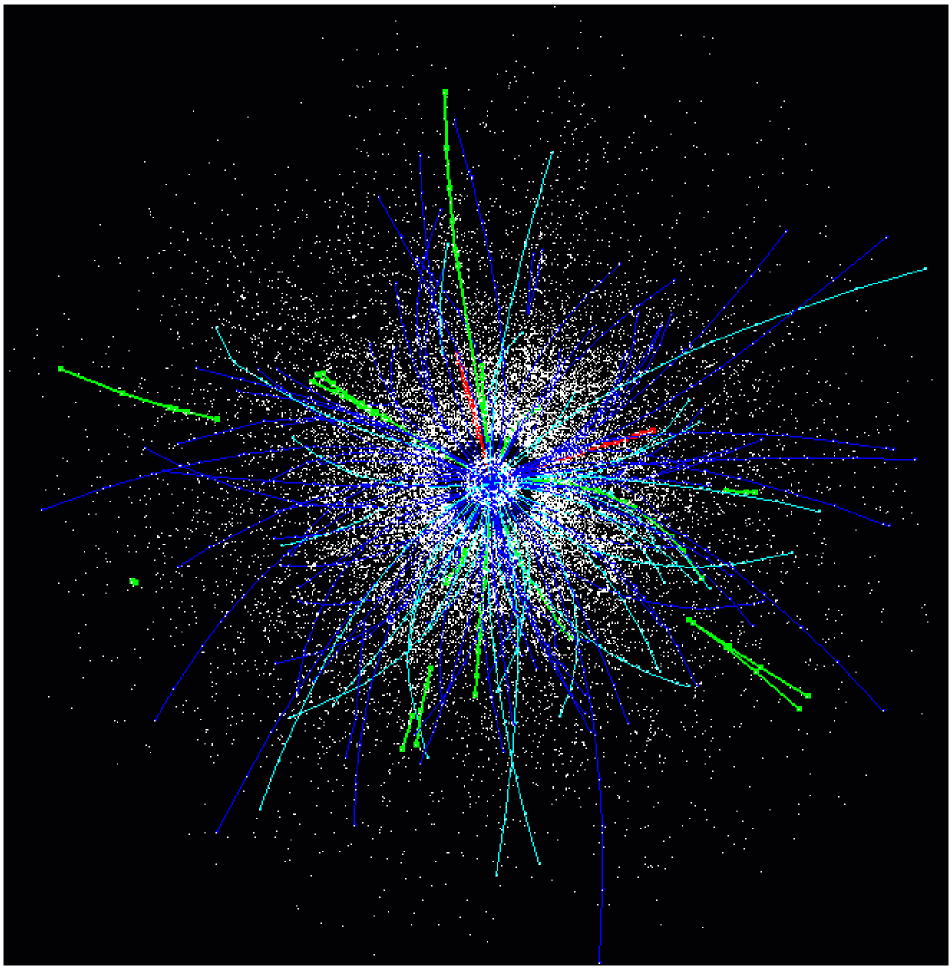

Figura 6.1: Visualizzazione in 3D dell'evento simulato di interazione p-p.

che in questo caso si può notare che la concentrazione dei puntini corrisponde proprio ai punti in cui si trovano i *layer*.

Questo evento contiene circa 1000 tracce e 30000 *RecHit*. In queste immagini viene mostrato solo un numero limitato di tracce per rendere più chiaro il *display* dell'evento.

In particolare vengono mostrate solo le tracce con p_t>50 Gev/c^2, questo spiega perché per la maggioranza dei *SimHit* non passa nessuna traccia. Col taglio ad alto p_t vengono di fatto escluse la maggioranza delle tracce create dalle interazioni periferiche p-p (*pileup*) che si sovrappongono alle tracce dell'evento interessante e cioè i due muoni e i due elettroni. Il programma di simulazione usa degli algoritmi di digitizzazione per passare dai *SimHit* a gruppi di *strip* (o *pixel*) vicini (*cluster*) da cui poi si ottengono i *RecHit*. Nei

RecHit, assieme alla posizione del *cluster*, viene memorizzato, anche l'errore proporzionale al numero di *strip* accese.

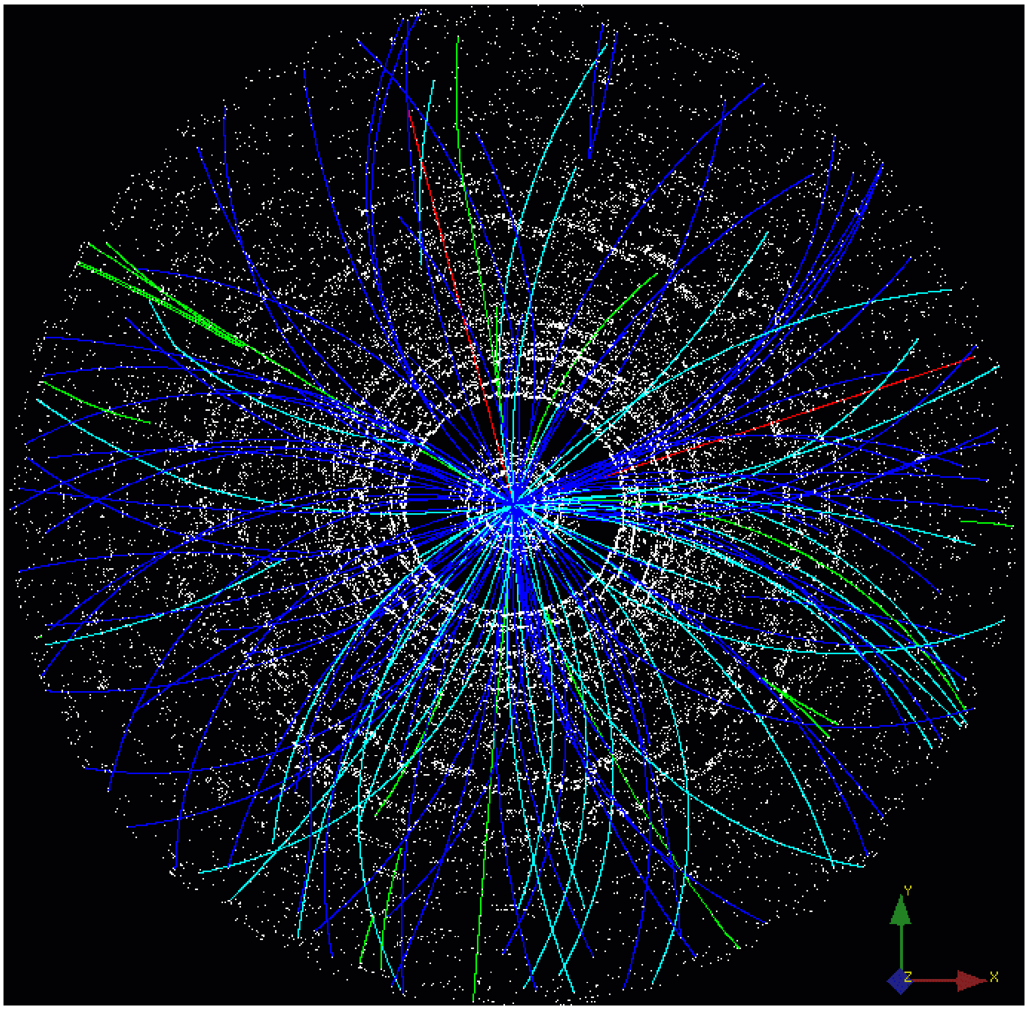

Figura 6.2: Proiezione nel piano xy dell'evento simulato.

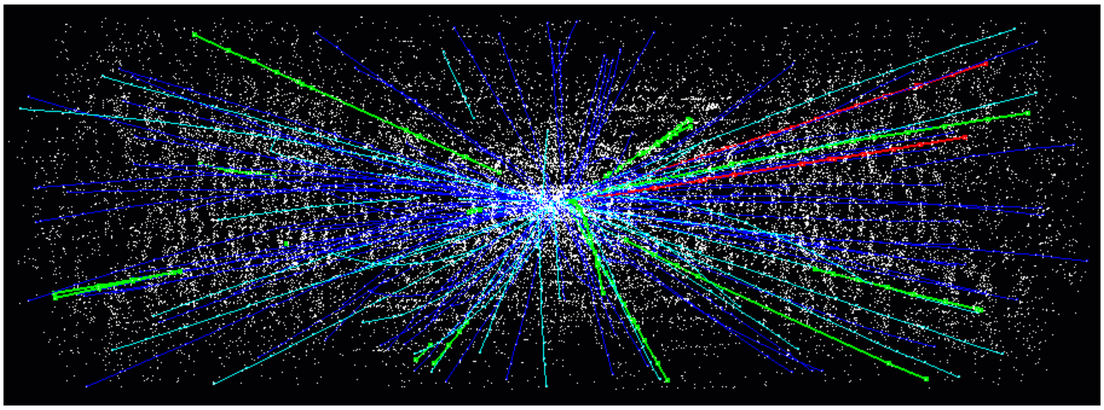

Figura 6.3: Visualizzazione in 3D dell'evento simulato di interazione p-p.

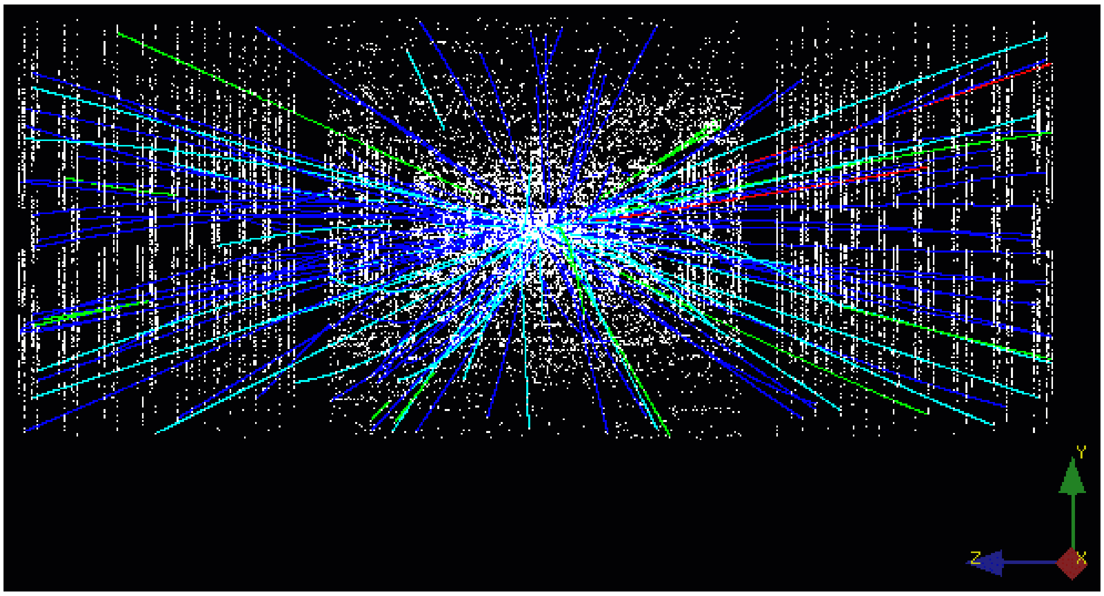

Figura 6.4: Proiezione nel piano yz dell'evento simulato.

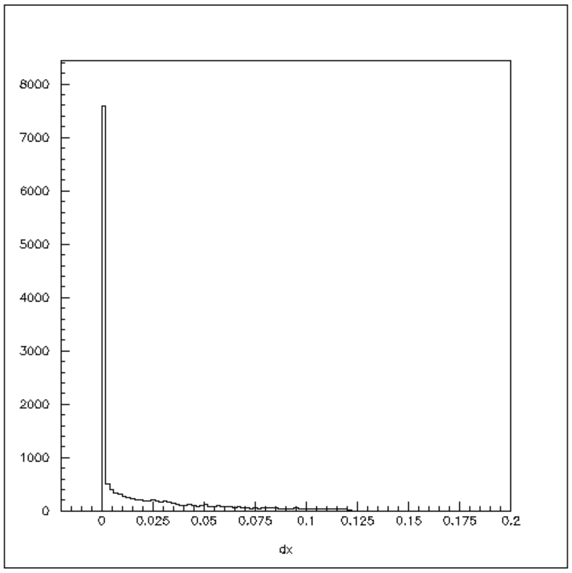

Figura 6.5: Istogramma che mostra le distribuzioni dell'errore per tutti i *RecHit* di questo evento.

6.2 Visualizzazione del Tracker con gli eventi

Lo scopo della mappa 2D progettata per il software di visualizzazione del *Tracker* è quello di permettere all'utente di avere sia una immediata visione d'insieme di informazioni riguardanti l'evento o le caratteristiche del rivelatore o, all'occorenza, di vedere queste informazioni in dettaglio per una particolare regione del *Tracker*.

La mappa di figura 6.6 mostra gli *hit* simulati relativi ai vari moduli del *Tracker* dell'evento che si è scelto di visualizzare. Invece la mappa di figura 6.10 mostra il numero di *hit* ricostruiti relativi ai vari moduli del *Tracker* dell'evento. Sono in bianco i moduli nei quali non si è verificato nessun *hit* ricostruito e con una gradazione di colore che va dal giallo al rosso quelli con un numero di *RecHit* che va da 1 a 20. Questa mappa permette di avere subito una visione d'insieme dell'evento, mentre la mappa di figura 6.11 permette di vedere le cose con più dettaglio, mostrando per i *RecHit* proprio le *strip* e i *pixel* accesi.

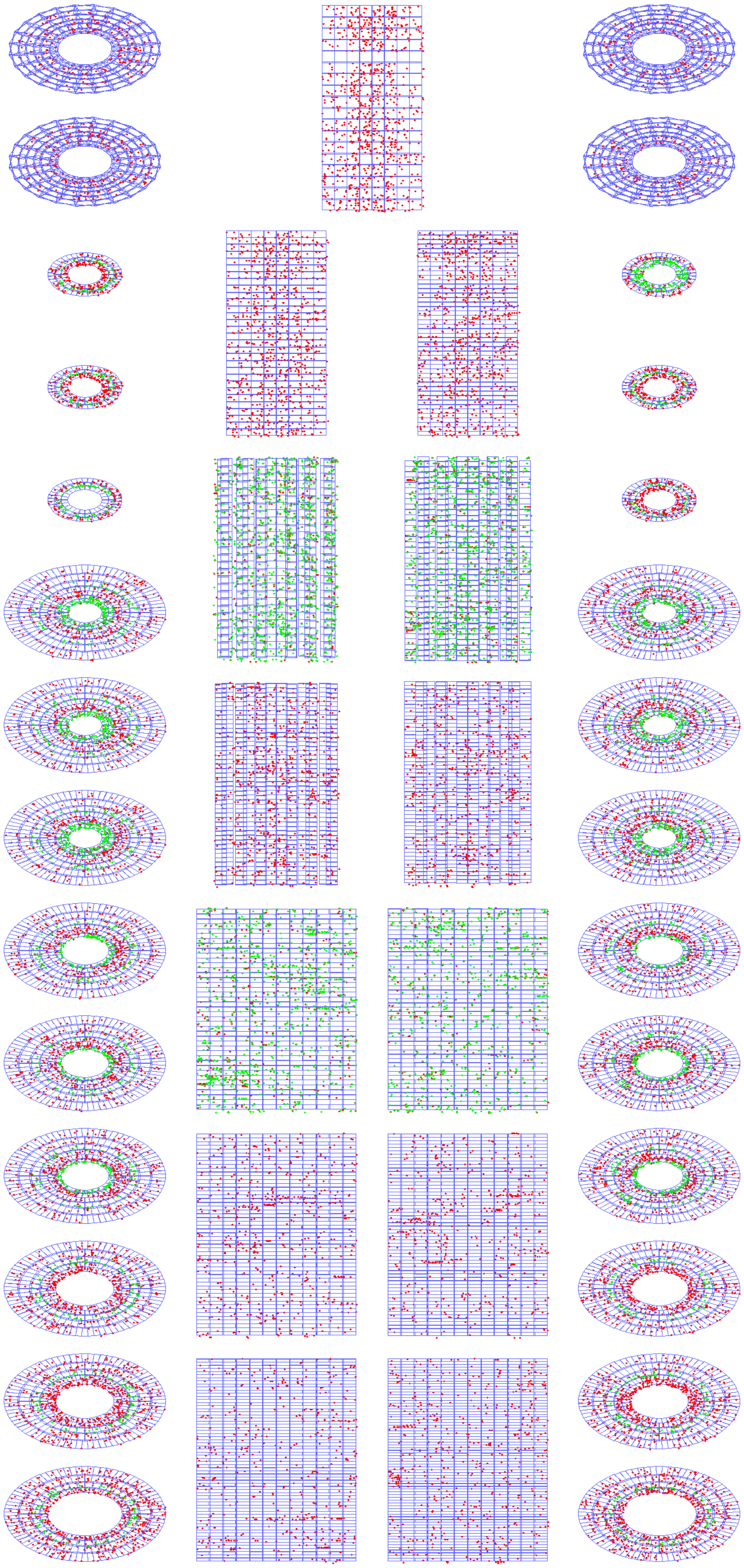

Figura 6.6: Mappa del *Tracker* che mostra i *SimHit* come punti nel modulo.

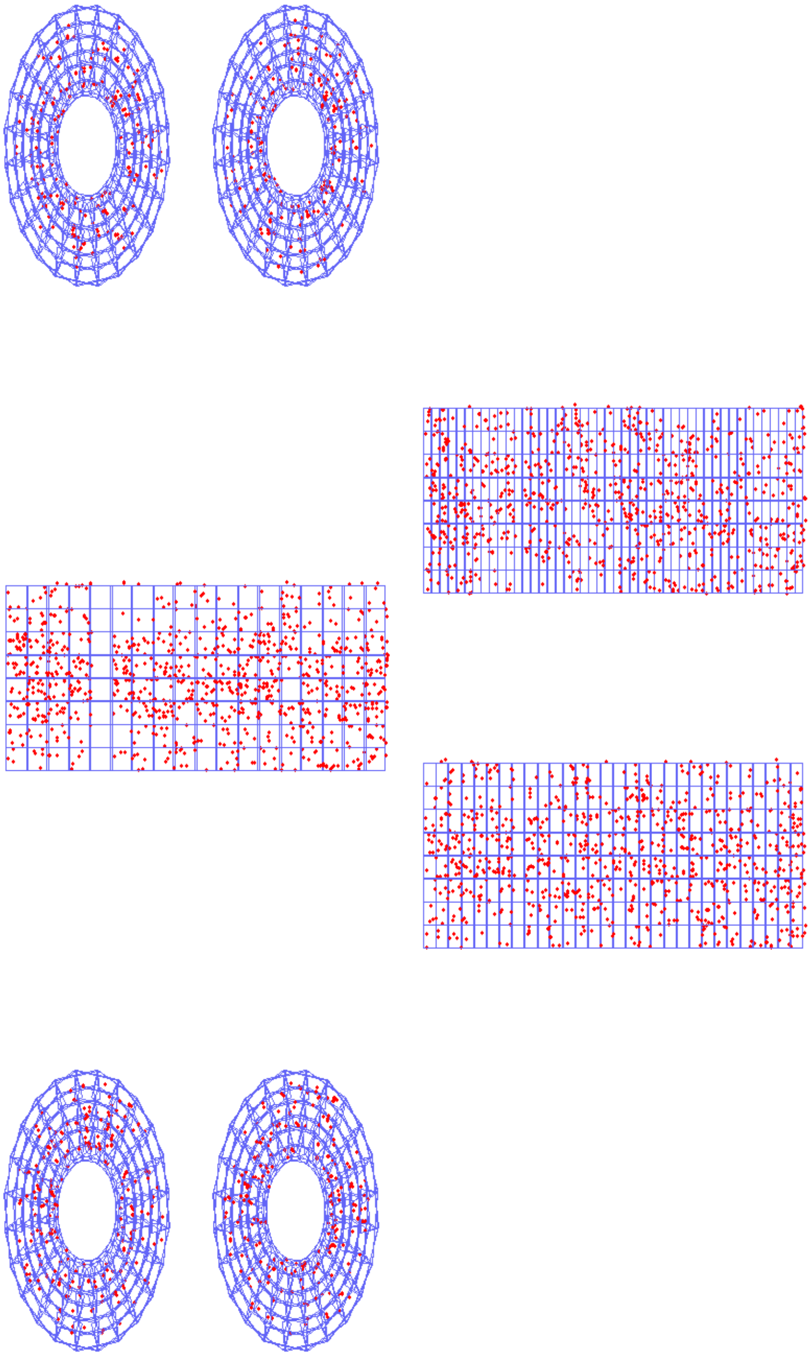

Figura 6.7: Ingrandimento della rappresentazione dei *SimHit* della parte *pixel* del *Tracker*.

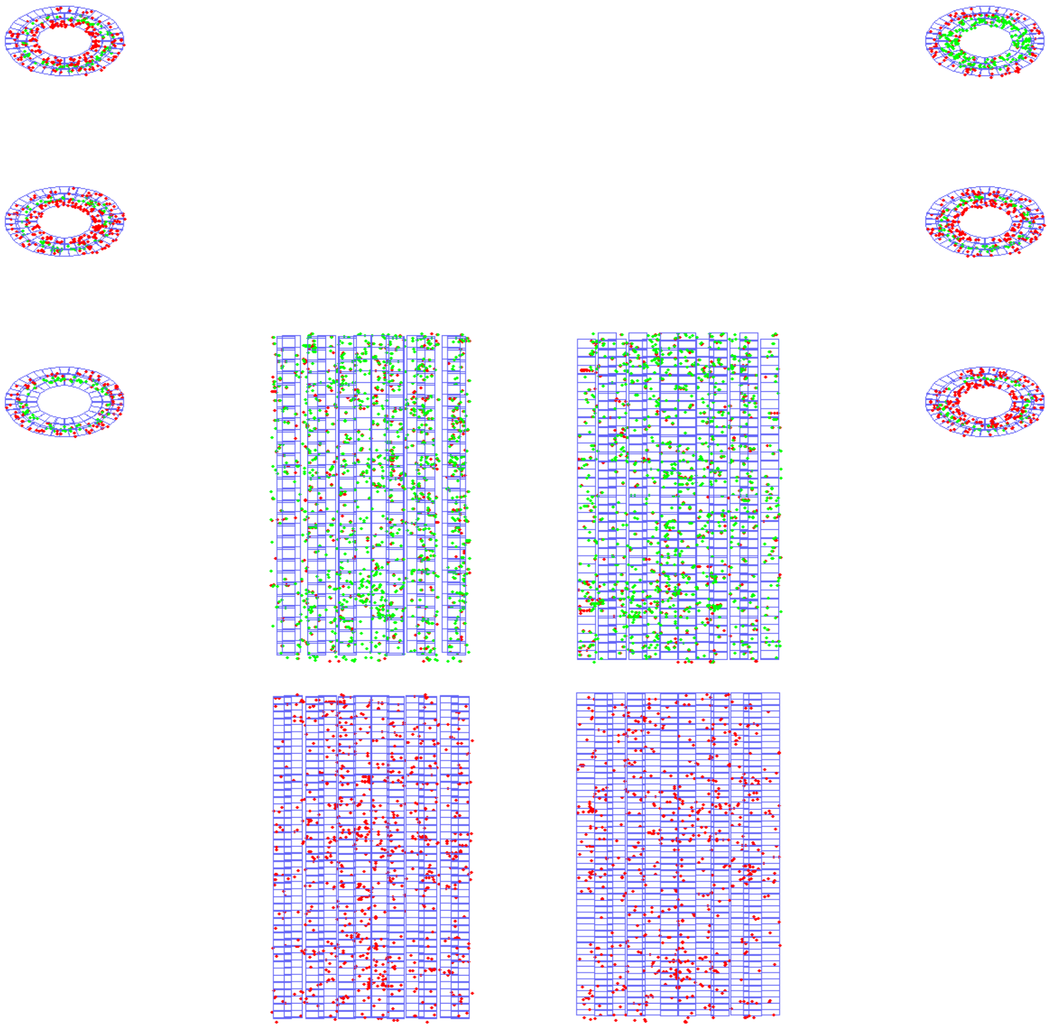

Figura 6.8: Ingrandimento della rappresentazione dei *SimHit* della parte *inner* del *Tracker*.

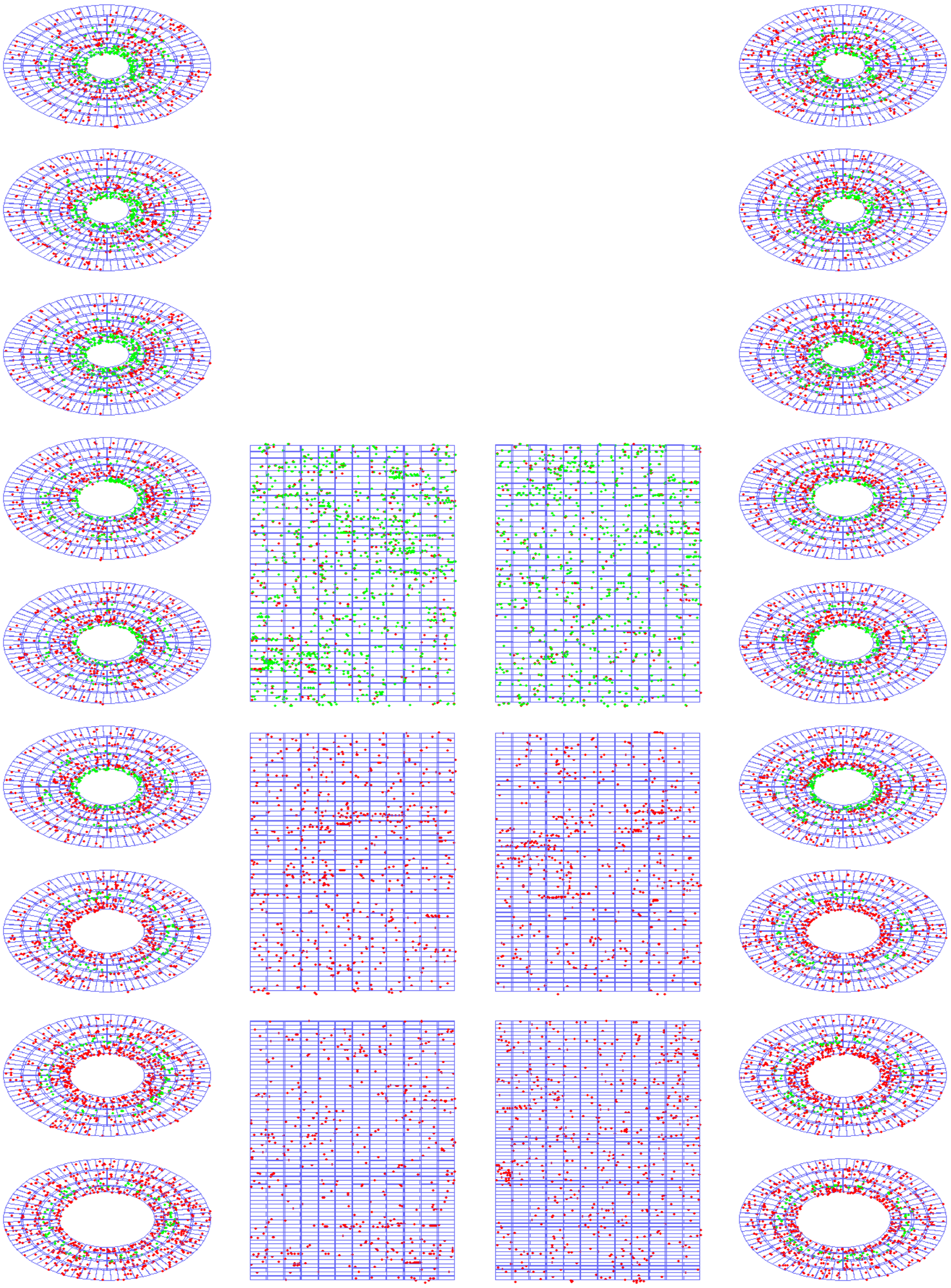

Figura 6.9: Ingrandimento della rappresentazione dei *SimHit* della parte *outer* del *Tracker*.

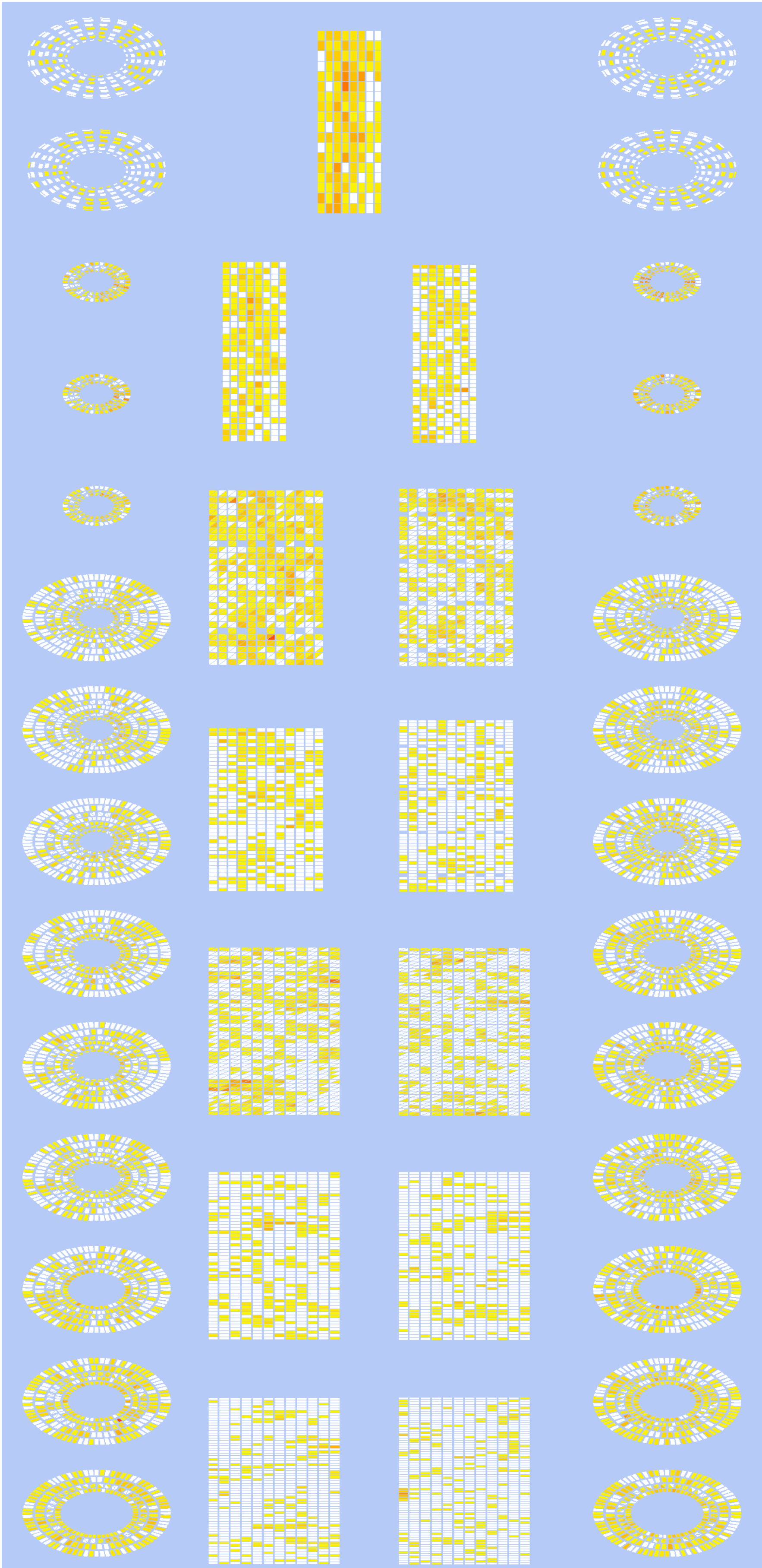

Figura 6.10: Mappa del *Tracker* che mostra il numero di *hit* ricostruiti relativa ai vari moduli (bianco nessun *RecHit*, giallo-rosso per *RecHit* 1-20).

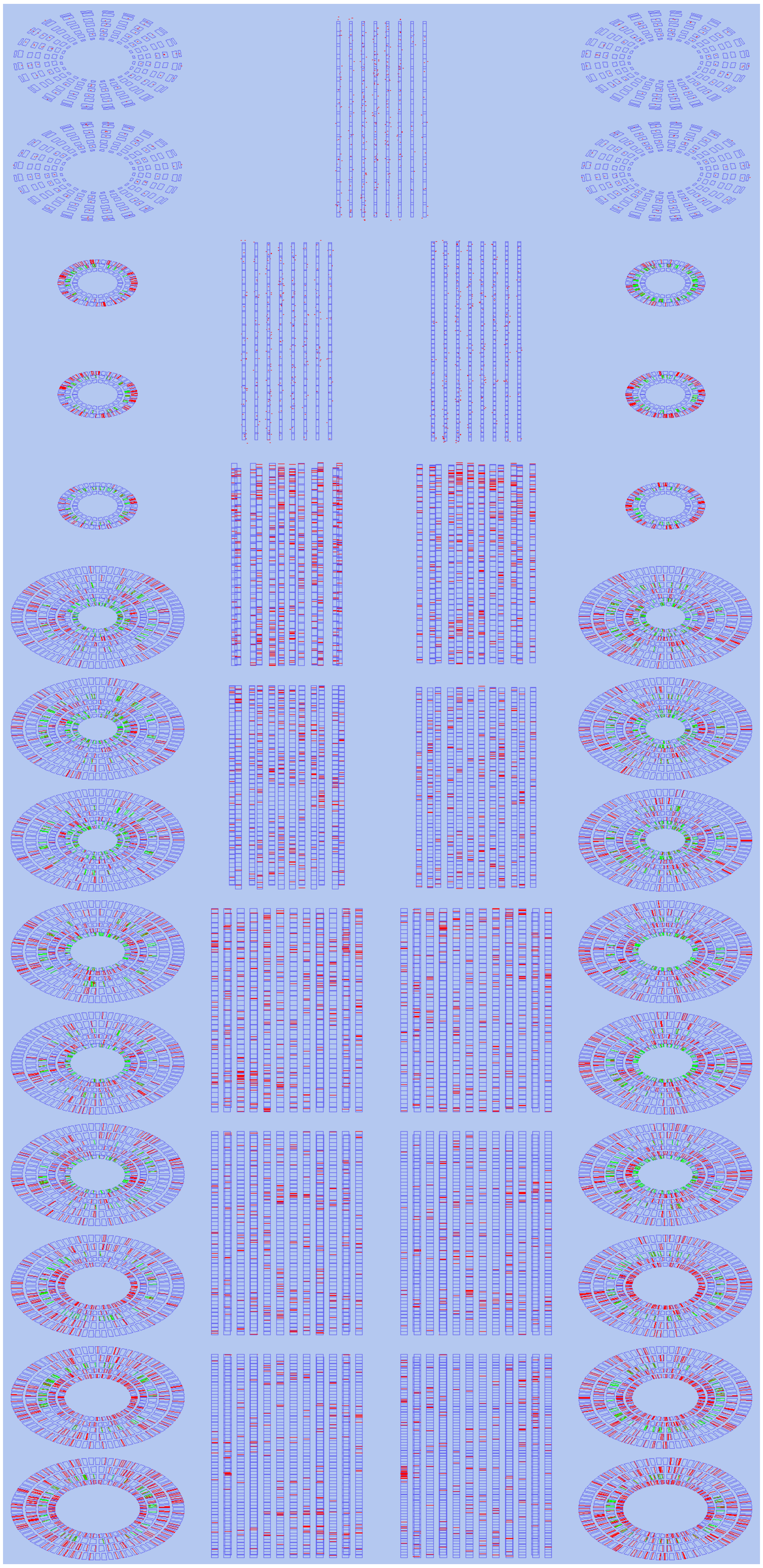

Figura 6.11: Mappa del *Tracker* che mostra le *strip* e i *pixel* accesi.

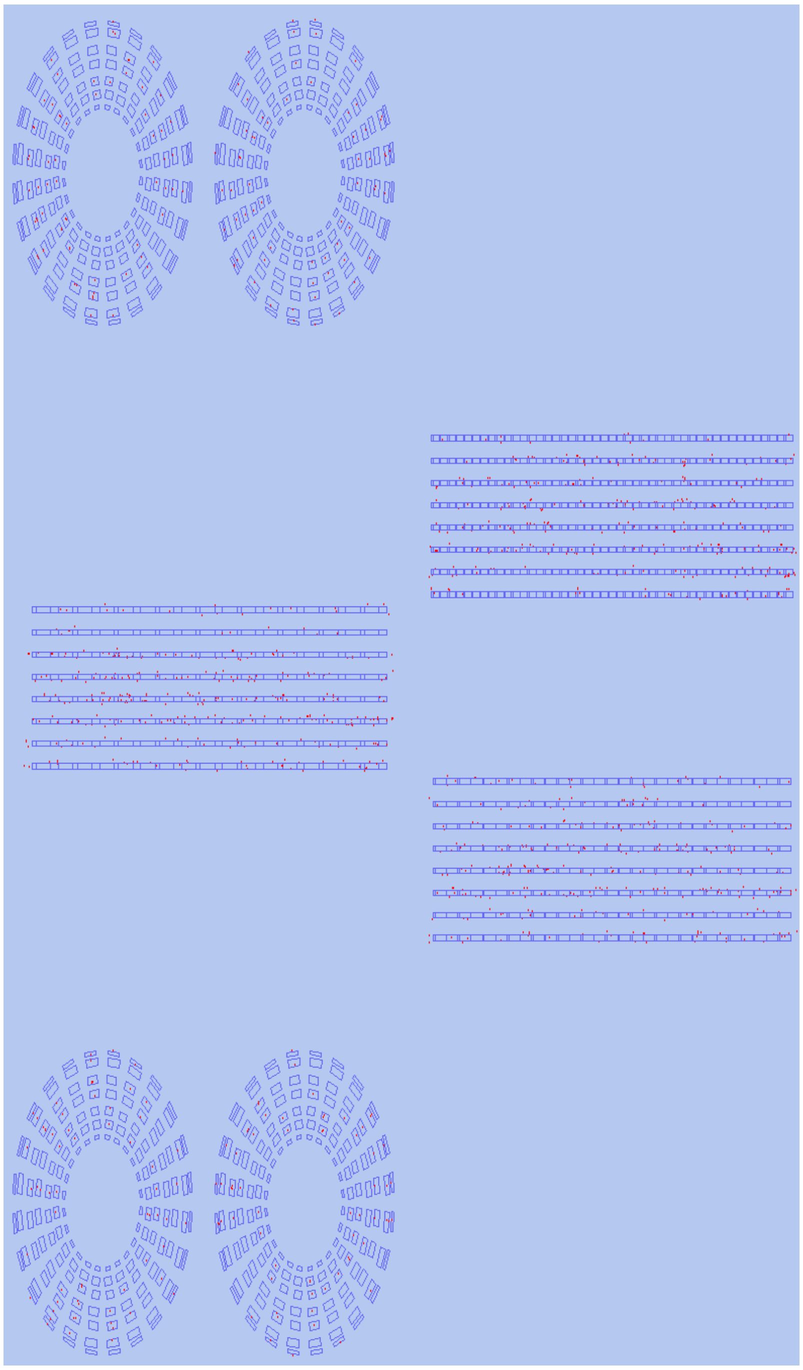

Figura 6.12: Ingrandimento della rappresentazione dei *pixel* accesi della parte *pixel* del *Tracker*.

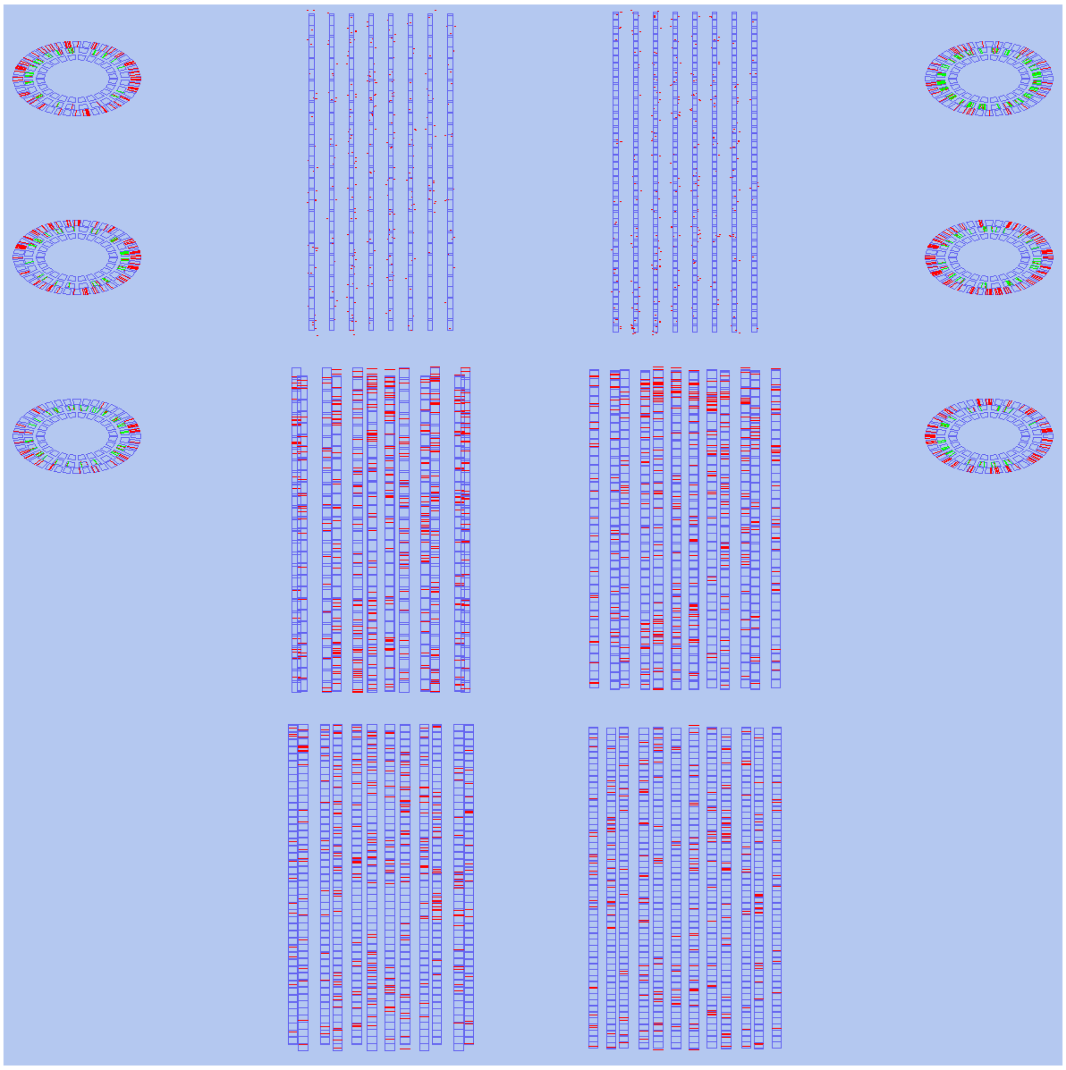

Figura 6.13: Ingrandimento della rappresentazione delle *strip* accese della parte *inner* del *Tracker*.

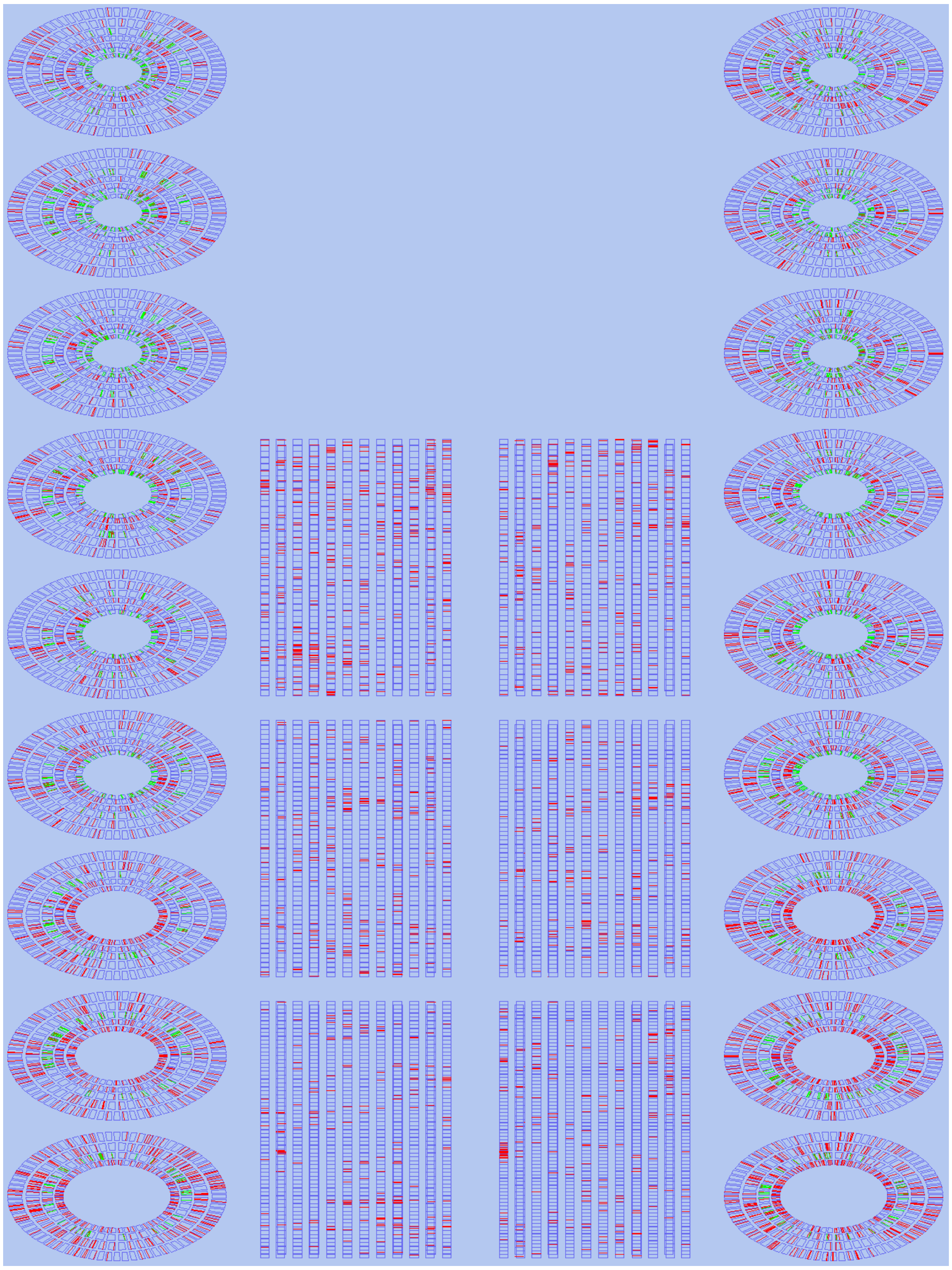

Figura 6.14: Ingrandimento della rappresentazione delle *strip* accese della parte *outer* del *Tracker*.

6.3 Monitorare le caratteristiche del Tracker e la correttezza dei dati e degli algoritmi

Diamo in questo paragrafo alcuni esempi dell'utilità delle mappe del *Tracker*.

1. Controllo della validità dei dati del *database* (DDD). La figura 6.15 mostra la mappa dei *layer* del *barrel* 32 33 e 35 dove è evidente l'assenza di alcuni moduli: questi in realtà erano presenti nella precedente versione del software e del *database*. La mappa evidenzia quindi un errore nel *database* o dell'interfaccia ai dati del *database*.

2. Confronto tra *SimHit* e *RecHit*. Quando si usa la mappa per rappresentare l'evento o meglio ancora più eventi l'uno sovrapposto all'altro (fig.6.16 o fig.6.17) possiamo usare queste rappresentazioni per controllare la consistenza tra l'evento e la geometria del *Tracker*. In particolare gli errori presenti nei dati o nel software possono produrre delle incongruenze nelle figure, per esempio:

   - *SimHit* o *RecHit* disegnati al di fuori del modulo (come per esempio in fig.6.16 dove per il disegno dei moduli sono stati usati i dati della precedente geometria del *Tracker*),
   - presenza di zone senza segnale,
   - *SimHit* che non corrispondono ai *RecHit* (come per esempio in fig.6.17 nel quale sono stati disegnati insieme i *SimHit* come punti e i *RecHit* come *strip* accese dello stesso evento),
   - se una traccia attraversa una regione in cui due moduli si sovrappongono dovrebbe generare due *RecHit* (o *SimHit*), uno per ogni modulo.

3. Monitoraggio *online* dell'apparato: rappresentazione delle "*strip* difettose". Nel nostro caso si aveva a disposizione, dal *database* della costruzione, le informazioni sul numero di "*strip* morte" di 2173 moduli del *Silicon Strip Tracker* (SST), dei quali però non si conosce ancora la posizione in cui saranno montati. Ripetendo i valori di questi 2173 moduli fino a raggiungere il numero totale dei moduli che costituiranno il SST sono stati così ottenuti i valori necessari per creare la mappa mostrata nella figura 6.18, in cui i moduli colorati di bianco sono quelli che non hanno nessuna "*strip* difettosa" e con una gradazione di colore che va dal giallo al rosso per un numero di "*strip* difettose" che va da 1 a 100 ed in rosso quelle con un numero di "*strip* difettose" >100.

Questa mappa è un esempio di come sia possibile con questo strumento un monitoraggio di qualsiasi quantità riguardante i circa 20000 moduli del tracciatore, ad esempio: temperatura, voltaggi, correnti, etc. Una sola occhiata permette di accertarsi se tutto è nella norma.

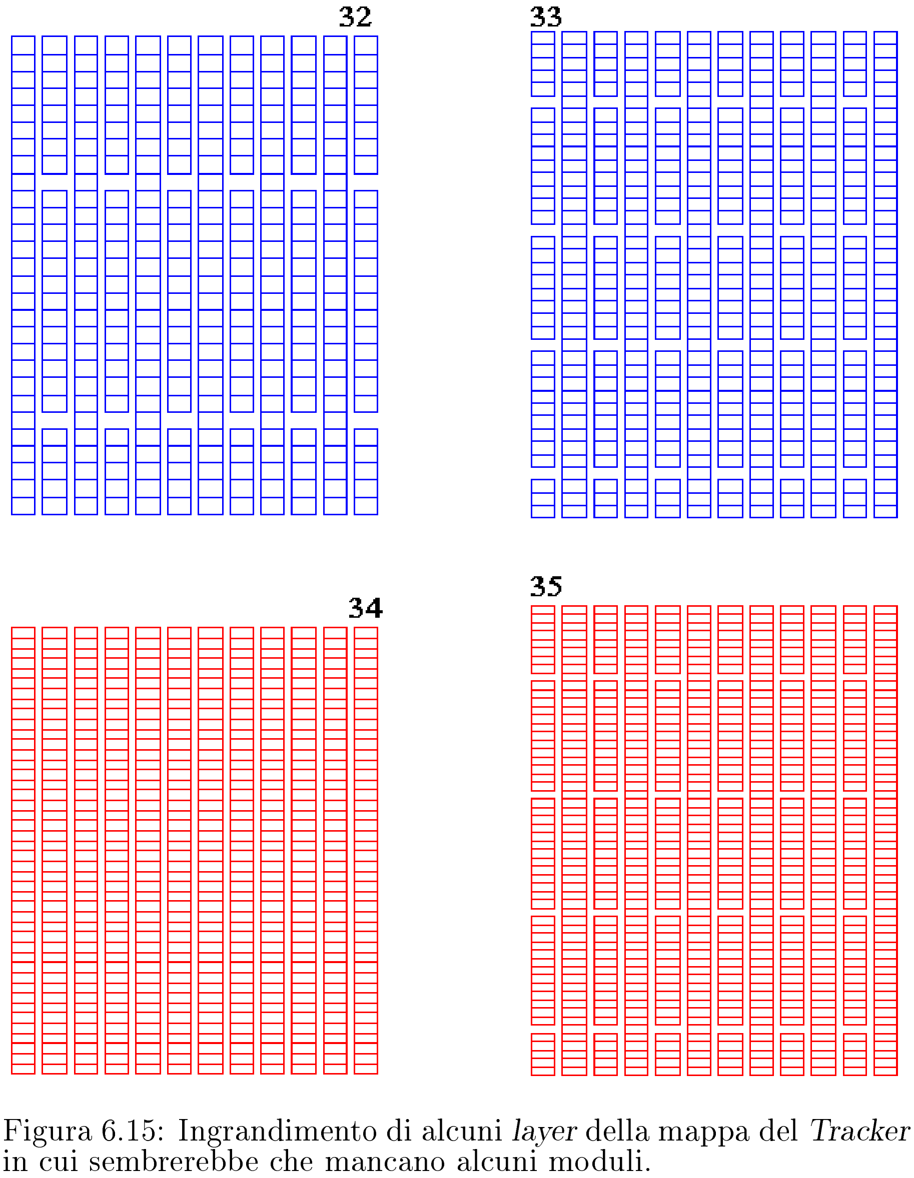

Figura 6.15: Ingrandimento di alcuni *layer* della mappa del *Tracker* in cui sembrerebbe che mancano alcuni moduli.

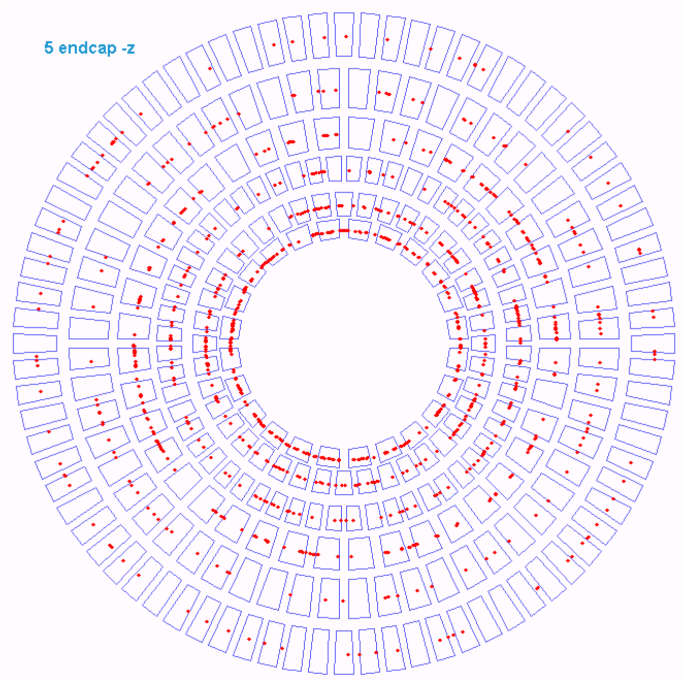

Figura 6.16: Ingrandimento di un disco della mappa del *Tracker* che dovrebbe mostrare i *RecHit* come punti al centro del modulo, invece ce ne sono alcuni disegnati al di fuori. Questa figura è stata ottenuta usando un file con i dati della precedente geometria del *Tracker*.

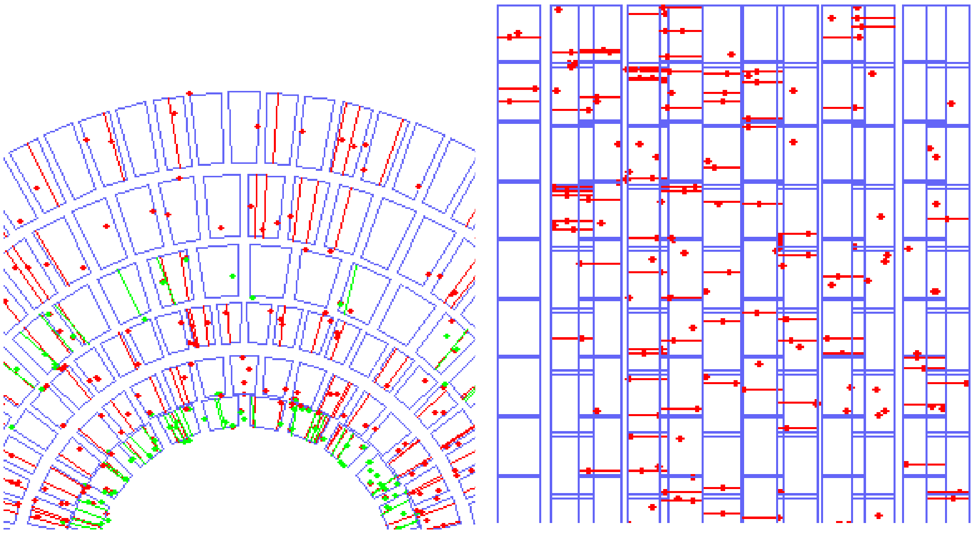

Figura 6.17: Ingrandimento di alcuni 1slshape layer nei quali sono stati disegnati insieme i *SimHit* come punti e i *RecHit* come *strip* accese dello stesso evento.

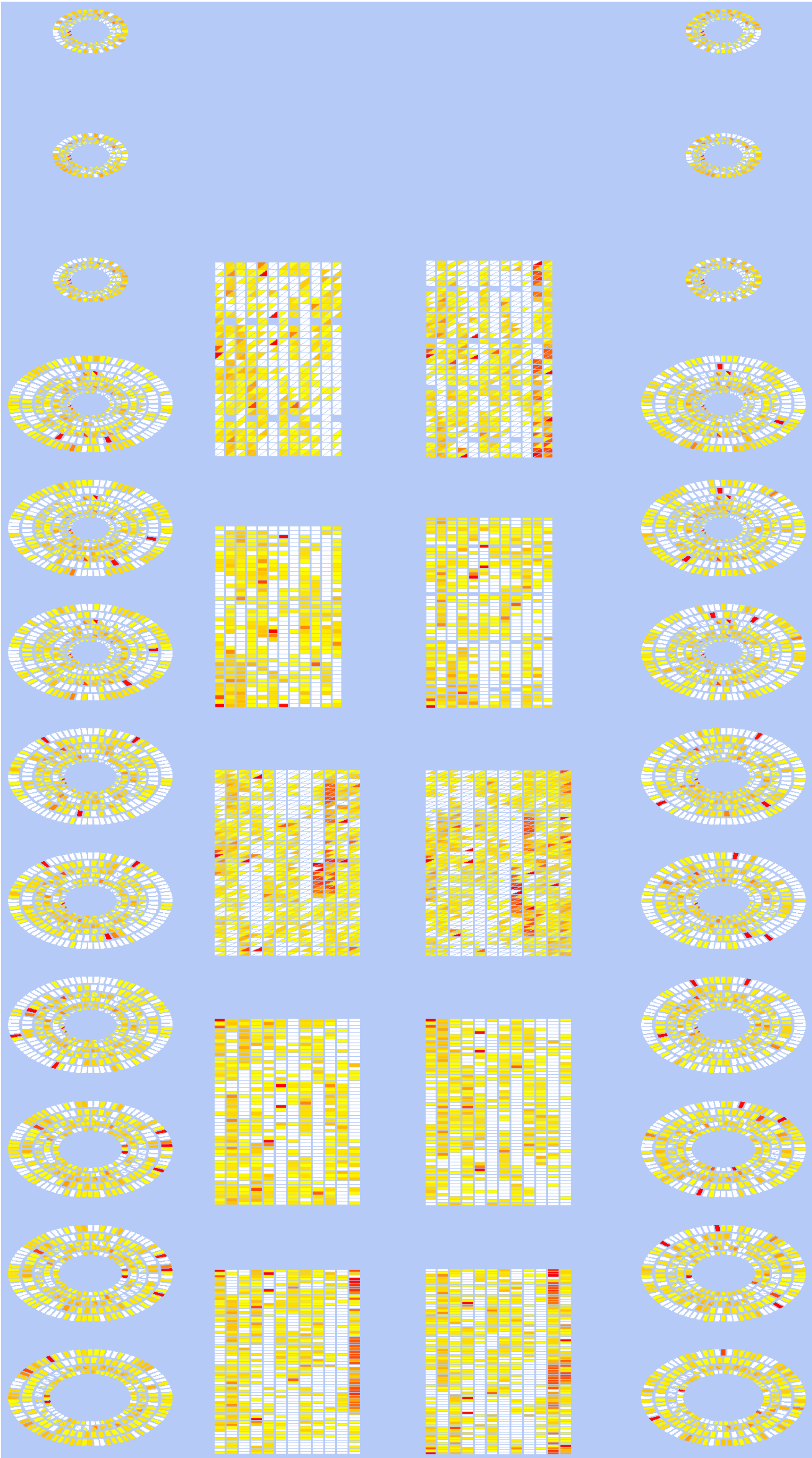

Figura 6.18: Mappa del *Tracker* che mostra il numero di "*strip* difettose" relativa ai vari moduli (bianco nessuna "*strip* difettosa", giallo-rosso per "*strip* difettose" 1-100, rosso >100).

Conclusioni

Il mio lavoro per questa tesi ha comportato due fasi:

(1) partecipazione alla progettazione *Object Oriented* con la creazione di diagrammi UML e loro discussione;

(2) implementazione di alcune delle classi progettate per il prototipo funzionante in ORCA e implementazione della mappa 2D con un prototipo java.

Nella prima fase ho constatato che l'uso dell'UML è complesso, quindi va adattato in base alle specifiche esigenze dei progettisti e dei progetti, utilizzando ciò che serve nello specifico contesto. Infatti nel nostro caso è stato ritenuto utile usare solo tre dei nove diagrammi dell'UML e in particolare poi il *diagramma dei pacchetti* è stato realizzato in modo "libero" al fine di rappresentare solo gli aspetti che si volevano mettere in evidenza. Nella fase di sviluppo del prototipo per ORCA ho constatato che c'è una grande differenza tra disegnare una collezione di buoni diagrammi e implementarli in un linguaggio come il C^{++}.

Comunque l'*Object Oriented Designer* (OOD) si è rivelata attualmente la tecnica migliore e vantaggiosa perchè permette di facilitare sia la comunicazione all'interno del gruppo di lavoro, sia le fasi successive di perfezionamento e modifica del software.

L'idea di implementare subito le *classi* senza un progetto di base all'inizio può sembrare più semplice e veloce, ma poi si rivela essere un consumo di tempo nel momento in cui bisogna perfezionarlo e modificarlo.
Invece usando l'OOD il processo di sviluppo del software è progredito nel seguente modo: ho analizzato di volta in volta un dato problema e ne ho realizzato il rispettivo diagramma UML che ho presentato al gruppo di lavoro. Nella riunione del gruppo sono stati analizzati e discussi i vari dettagli del disegno presentato, così tramite critiche costruttive il progetto è stato di volta in volta migliorato. Ogni volta che si è deciso di apportare un'altra modifica è stato facile individuare la classi coinvolte.

Un altro strumento che si è rivelato molto utile è il *design pattern*, perché propone delle soluzioni già testate da altri per particolari problemi che hanno accelerato il processo di sviluppo.

Le mappe sviluppate per la visualizzazione in 2D del *Tracker* hanno mostrato come sia semplice con queste avere un quadro globale della sua geometria con la possibilità di controllare che i dati contenuti nel *database* siano corretti. Esse permettono inoltre di visualizzare i difetti ("*strip* difettose") dell'apparato e altre informazioni. Per gli eventi le mappe permettono sia di visualizzare gli eventi stessi sia di controllare che gli algoritmi (per esempio quelli per la generazione degli *hit* simulati) siano corretti.

Il progetto e le mappe 2D del *Tracker* sono state presentate alla riunione del gruppo italiano della collaborazione CMS in occasione del consorzio tenutosi a Bari il 20/giugno/2003, riscuotendo un grande interesse.

Bibliografia

[1] M.S. Mennea. *Sviluppo di sistemi di visualizzazione dati per la fase di costruzione e presa dati del tracker dell'esperimento CMS*. PhD thesis, Dipartimento Interateneo di Fisica dell'Universitá degli Studi di Bari, 2002-2003.

[2] CERN/AC/95-05(LHC), editor. *The Large Hadron Collider, Conceptual Design*. CERN, 1995.

[3] CERN/LHCC 98-6 TDR 5, editor. *CMS Collaboration, "Addendum to CMS Tracker TDR"*. CERN, April 1998.

[4] CERN/LHCC 96-45, editor. *The Compact Muon Solenoid, Tecnical Proposal*. CERN, 1996.

[5] V.Innovente, L.Silvestris, and D.Stickland. CMS Software Architecture Software framework, services and persistency in high level trigger, reconstraction and analysis, Computer Physics Communications. http://cobra.web.cern.ch/cobra/, 2001.

[6] CMS OO Reconstruction. http://cmsdoc.cern.ch/orca/. Home page for information related to the ORCA program for CMS Reconstruction.

[7] IGUANA. http://iguana.web.cern.ch/iguana. Home page IGUANA.

[8] G.Alverson, I.Gaponenko, L.Taylor, and L.Tuura. CMS INTERNAL NOTE: The CMS Interactive Graphical User Analysis (IGUANA) "Functional Prototype" Software. http://iguana.cern.ch, 31 October 2000.

[9] Qt Reference Documentation. http://doc.trolltech.com/3.0/index.html. On-line documentation.

[10] SoQt Documentation. http://doc.coin3d.org/SoQt/index.html. On-line documentation.

[11] TGS Open Inventor. http://anaphe.web.cern.ch/anaphe/documentation /OpenInventor/pro/overview.html. On-line documentation.

[12] G.Alverson, S.Muzaffar, I.Osborne, L.Taylor, and L.Tuura. IGUANA4 status. http://iguana.cern.ch/docs/presentations/2002-12-03-status/iguana4.pdf.

[13] MindView Inc. Bruce Eckel, President. *Thinking in C++ 2nd Edition*. http://www.mindview.net/Books/TICPP/ThinkingInCPP2e.html, 13 gennaio 2000.

[14] Herbert Schildt. *C++ La guida completa seconda edizione*. McGrawHill, 1999.

[15] Unified Modeling Language. www.uml.org. Home page UML.

[16] Adriano Comai. INTRODUZIONE a UML. www.analisi-disegno.com, 1998-2001.

[17] Adriano Comai. Linee Guida -UML - casi d'uso. www.analisi-disegno.com, 2001.

[18] Robert C. Martin. Class Diagrams - Part 1. http://www.objectmentor.com/publications/umlClassDiagrams.pdf.

[19] S.Rossini and L. Dozio. Il Pattern MVC. *Mokabyte http://mokabyte.infomedia.it/*, Gennaio 2003.

[20] Matthias Kalle Dalheimer. *Programming with Qt: Writing Potable GUI applications on Unix and Win32*. O'Reilly, http://www.oreilly.de/catalog/prowqt2/, 2nd edition, Februar 2002.

[21] J.Wernecke. *The Inventor Mentor: Programming Object-Oriented 3D Graphics with OpenInventor*. Addison Wesley, http://oss.sgi.com/, 1994.

[22] Giancarlo Crocetti. Design Patterns. *Mokabyte http://mokabyte.infomedia.it/*, Giugno 2002.

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

[24] Concurrent Versions System: the open standard for version control. http://www.cvshome.org/. Home page.

[25] Daniele Giacomini. Appunti di informatica libera. http://linux.oasi.asti.it/a2205.html. Sito internet.

[26] Poseidon for UML. http://www.gentleware.com/. Gentleware.

[27] IGUANA Developer Guide: How To Create a Plug-in. http://iguana.web.cern.ch/iguana/snapshot/devguide/howto/new-plugin.html.