次世代線型加速器実験用
測定器シミュレータの開発

神戸大学大学院自然科学研究科 博士前期課程
物理学専攻
039S108N
岸本 晋
指導教官 川越 清以

2005年2月10日

概 要

高エネルギー物理学とは、物質の究極の構成要素を探求し、その間に働く相互作用を解明することを目的とする学問である。高エネルギー物理学発展の歴史は、その主要な研究手段である高エネルギー加速器の進展とともにあった。新しいより高いエネルギーの加速器での衝突実験は、より小さな領域の探索を可能とし、我々の極微の世界に対する理解を深めてきた。現在我々は、標準理論に集約される世界像、すなわち、「自然がクォークとレプトンと呼ばれる少数の物質粒子と、その間の相互作用を媒介するゲージ粒子からなり、それらがゲージ対称性で関係づけられている」とする世界像（ゲージ原理）を手にするに至っている。この標準理論は、現在までの全ての実験事実を見事に説明し、その破れを示す確かな証拠はこれまでの所見つかっていない。しかしながら、標準理論を形作るもう一つの重要な要素である自発的対称性の破れを引き起こす実体、つまり、この標準理論にあって、ゲージ粒子や物質粒子の質量を生み出すなくてはならない粒子、すなわちヒッグス粒子は未だ未発見のままである。現在の高エネルギー物理学の最重要課題は、このヒッグス粒子の発見とその性質の詳細な研究、さらには標準理論の枠組みを本質的に越える理論（超対称性、余次元、大統一など）の実験的な手がかりを得ることである。

このような情勢をうけて、現在測定器全体のシステム開発に不可欠な測定器のフルシミュレータの開発が急ピッチで進められている。この開発中のシミュレータは JUPITER(JLC Unified Particle Interaction and Tracking EmuratoR) と呼ばれる、粒子と物質の相互作用のためのツールキットである Geant4 をベースにするモンテカルロシミュレータと解析のためのフレームワークを提供する ROOT ライブラリベースの Sattelites からなる。

JUPITER の開発は、現在は GLD（GLC based Detector）のデザインをインストールしデバッグを行いながら、解析に移行中というところである。また、現行の JUPITER では測定器のインストール、アンインストールに関してもっとシェイプアップすべき部分が存在する。そのため開発の一環として XML インターフェース追加を行った。

以上を本研究では報告する。

目 次

第1章　序論

ILC 計画とは International Linear Collider の略で巨大な電子陽電子衝突型線形加速器を建設し、高エネルギー領域の物理現象を測定する世界的計画である。以下ではまず我が国における計画の経緯を見、電子陽電子衝突型線形加速器の建設が望まれている理由について加速器と観測したい物理現象の面からそれぞれ述べる。

1.1　我が国における計画の経緯

まず我が国におけるリニアコライダー計画の歴史を概観する。

1978 年 リニアコライダー計画の始まり

リニアコライダー計画の開発研究は1978 年までさかのぼる。その年のICFA(International Collider for Future Accelerators) セミナーの参加者は日本の高エネルギー物理学者にリニアコライダーという新しい概念を報告した。そしてトリスタン計画がスタートしたちょうど 1982 年に、リニアコライダーのための開発研究グループが小規模ながら発足した。このグループでは大電力高周波電源や非常に高い加速勾配を持つ加速管といった加速器に関する先進技術に関して研究を行った。

1986 年 電子陽電子リニアコライダー計画の勧告

1986 年に高エネルギー委員会での総意により、トリスタン計画後のエネルギーフロンティアを担うものとしての電子陽電子リニアコライダー計画が勧告された。この勧告を受け鍵となり基礎となる各加速器技術の項目の検討および全体にわたる詳細な計画を練るといった JLC(Japan Linear Collider) 計画のための広範囲にわたる開発研究プログラムが始められた。これと並行して物理や実現可能性についての検討も始められた。

1992 年代始め JLC-I レポート

90 年代始めには LEP 実験をはじめとする様々な実験による電弱相互作用の精密測定の結果、軽いヒッグス粒子の存在と超対称性理論（SUSY）が標準理論を超える物理のシナリオとして注目され始めた。そのような状況のもとで 1992 年にまとめられたのが、いわゆる「Green Book」と称される「JLC-I」レポートである [1]。このレポートでは重心系 500GeV 以下での実験が急務であることが説かれ、さらに TeV 領域での物理へつながる道筋が示された。物理探索に関するシナリオ、必要となる測定器の設計、また加速器の概要

などについても記述が及び、その中にはX線自由電子レーザーへの応用に関してのアイデアも示されていた。これはリニアコライダー計画の全体像を描いた世界で最初のレポートであった。

1997年 次期主要計画に

1997年には高エネルギー物理学の将来計画に関する高エネルギー委員会の小委員会で、電子陽電子リニアコライダー計画が高エネルギーの研究分野における次期主要計画であることが再確認された。小委員会は研究施設は世界に向けて開かれたものであり、日本は受入国として指導的役割を果たすべきであることを勧告した。これを受けて日本でのJLC計画を推進するために1997年KEKにリニアコライダー推進室が設置された。

2001年 リニアコライダー推進委員会の設置

2001年、リニアコライダー推進室は改組され、KEKの機構長の元にリニアコライダー推進委員会が設置された。この委員会の委員はKEKの管理局の事務官やKEKおよび日本の大学、さらにはアジアの研究所からの科学者によって構成された。またこの委員会の元で、加速器および物理と測定器の作業グループが組織された。サイトに関する調査や必要とされる施設、またかかるであろうコストに関してもそれぞれの作業グループが形成された。

2003年4月 JLCからGLCに

計画の国際性に鑑み、2002年末、ACFA（**A**sian **C**ommitte for **F**uture **A**ccelerators）はその国際性をより反映する新しい名称を募集。結果60を超える名前の候補が寄せられ議論の末2003年4月に、ACFAは「Global Linear Collider」を意味する「GLC」を新たなプロジェクト名とした。

2004年9月 GLCからILCに

2004年9月、世界の高エネルギー物理学研究者・加速器研究者は、超伝導技術に基づいたリニアーコライダーを国際協力で建設することについて合意した[1]。これがILCである。この国際的合意を受け、日本の高エネルギー物理学研究者・加速器研究者はこれまで推進してきた、GLC計画に基づく活動を発展的に解消し、KEKを基地としてILC建設に向けて大きな役割を果たしてゆく活動に着手することになった。

[1] これまでは日米がwarmと呼ばれる四極マグネット、ヨーロッパがcoldと呼ばれる超伝導技術で別々にR&Dを進めてきていた。

1.2 加速器について

加速器には加速する粒子の種類によってハドロン加速器、レプトン加速器の 2 種類に分けられる。一般的にハドロン加速器では加速粒子として陽子、反陽子が用いられ、レプトン加速器では電子、陽電子が用いられる。

1.2.1 固定標的型と正面衝突型実験

まず衝突実験の形態について述べる。
粒子の衝突のさせ方には、固定標的型 (ターゲット)、正面衝突型 (コライダー) の 2 種類がある。実験において新しい素粒子を探索する場合には高い重心系のエネルギー $E_{CM}(=\sqrt{s})$ を実現する必要がある。
以下にそれぞれの衝突実験において得られる重心系のエネルギーを挙げておく。

- 固定標的衝突型実験 (ターゲット実験) の場合
  重心系のエネルギーを $E_{CM}(=\sqrt{s})$ とし、
  E : 入射粒子のエネルギー、m : 入射粒子の質量、M : ターゲット粒子の質量とすると

$$\begin{aligned} E_{CM} &= \sqrt{m^2+M^2+2EM} \\ &\sim \sqrt{2EM} \\ &\propto \underline{\sqrt{E}} \end{aligned}$$

- 正面衝突型 (コライダー実験) の場合
  重心系のエネルギーを $E_{CM}(=\sqrt{s})$ とし、
  質量 m_1, m_2 の粒子をエネルギー E_1, E_2 で衝突させたとすると

$$\begin{aligned} E_{CM} &= \sqrt{m_1^2+m_2^2+2E_1E_2+2p_1p_2} \\ &\sim \sqrt{4E_1E_2} \\ &\propto \underline{E} \end{aligned}$$

入射粒子のエネルギー E に対する重心系のエネルギー $E_{CM}(\sqrt{S})$ との対応は上記のように、正面衝突型の場合には入射エネルギーに対して重心系のエネルギーが線形 $(\propto E)$ に比例して増えて行くのに対し、ターゲット型の場合には入射エネルギーの平方根 $(\propto \sqrt{E})$ でしか増加して行かない。また固定標的型実験の場合にはターゲット粒子の質量も重心系のエネルギーの項に含まれているので固定標的実験において高い重心系のエネルギーを得るには入射エネルギー E を大きくし、ターゲットの粒子に質量 M の大きいものを用いる必要がある。そのため一般的にターゲット粒子としては核子が用いられる。
加速器で同じエネルギー E まで加速させた粒子を衝突させる場合に、正面衝突実験の場

合には $\sqrt{S} = E + E = 2E$ の重心系のエネルギーを得られるのに対し、標的型実験においては 2 倍の入射エネルギーの粒子をターゲットに衝突させた場合には $\sqrt{2}$ 倍の重心系のエネルギーしか得られないことになる。
従って、高いエネルギー領域を目指す場合には正面衝突型加速器実験の方が加速エネルギーを固定標的実験にくらべて十分有効に用いることが出来る。最近のエネルギーフロンティアを目指す加速器実験においては常に正面衝突型のコライダーが用いられるのはそのためである。

1.2.2 円形加速器と線形加速器

次に加速器の加速の仕方について述べる。加速の仕方についても直線上を加速する線形加速器、マグネットを使って円軌道を描くようにして加速する円形加速器が存在する。加速させるビームパイプは地下トンネル内に置かれるので円形に走らせる方が何度も加速させることが出来るので、直線に走らせるよりもトンネルを掘る距離が少なくて済む。従って、これまでは衝突型円形加速器が加速器実験の主流となってきた。

線型加速と円形加速の違いは電子を加速する場合には非常に問題になってくる。円形に電子を加速した場合には電子の質量が非常に小さいため円形に軌道を曲げる際にシンクロトロン放射を起こしてエネルギーをロスしてしまう。シンクロトロン放射によるエネルギー損失は以下の式で表され、エネルギーの 4 乗に比例してその損失は大きくなる。

$$\begin{aligned} \Delta E &= \frac{4\pi e^2}{3R}\beta^3\gamma^4 \\ &\propto \frac{E^4}{R} \end{aligned}$$

ここで E はビームエネルギー [GeV]、R は曲率半径 [m] である。例えば電子が光速に近づいた $\beta \sim 1$ の場合エネルギー損失は $\Delta E = 88.5 \times E^4/R$ [keV] と表せる。

そのためより高いエネルギーまで電子を加速する必要がある場合、円形加速器を用いるとシンクロトロン放射によるロスが大きく、加えた電力に対して加速に用いられるエネルギーが少なくなり非効率である。線形加速器を行なう場合にはシンクロトロン放射が発生しないので加えた電力が効率良く加速に利用される。

1.3 粒子衝突（コライダー実験について）

次に衝突型実験 (コライダー実験) についてより詳しく見ていく。
一般的に加速する粒子がハドロンの場合ハドロンコライダーと呼ばれ主に陽子 · 反陽子 ($p\bar{p}$) が用いられる。加速する粒子がレプトンの場合にはレプトンコライダーと呼ばれ電子 · 陽電子 (e^+e^-) が用いられる。
従って、ハドロンコライダー実験では陽子 · 陽子衝突実験が最も一般的であり、レプトンコライダー実験は電子 · 陽電子衝突実験が一般的である。表.1.3 は陽子 · 反陽子衝突実験と電子 · 陽電子衝突実験を比較したものである。

衝突実験の種類	主な衝突粒子	バックグラウンド	エネルギーフロンティア
ハドロンコライダー (陽子 · 陽子衝突)	pp 陽子 · 陽子	多い 複雑な解析手法	円形加速で可能 → エネルギーフロンティアを開拓
レプトンコライダー (電子 · 陽電子衝突)	e^+e^- 電子 · 陽電子	少ない 精密測定が可能	円形加速では難しい → シンクロトロン放射によるロス ⇒ 線型加速ならば可能

表 1.1: 衝突型実験衝突粒子による違い

表.1.1 からも分るようにハドロンコライダー (pp) とレプトンコライダー (e^+e^-) の違いはまずバックグラウンドイベントの量が挙げられる。例えば pp コライダー一場合、陽子を構成しているパートン（クォークとグルーオン）同士の強い相互作用によって必要なイベントが起こる。陽子、反陽子のクォーク構成はそれぞれ $p(uud)$, $\bar{p}(\bar{u}\bar{u}\bar{d})$ であり、実際に衝突を起こして対消滅するのが例えば $u\bar{u}$ であったとするとその他のクォーク同士の相互作用は全てバックグラウンドイベントして検出されることになる。

ハドロンコライダーの利点として挙げられるのは円形加速においても高エネルギー領域まで加速することが可能になる部分である。レプトン、特に電子、陽電子を円形に加速する場合にはシンクロトロン放射によって $\Delta E \propto E^4/R$ の大きさで加速エネルギーをロスしていくのでエネルギーフロンティアを狙うには非効率である。よって、非常に高いエネルギー、即ち大きな質量の新粒子を探求するような先進的な物理現象を観測する実験ではハドロンコライダーが活躍している。
一方、電子 · 陽電子衝突実験においは、電子は陽子のように内部構造を持っていないので $e^+e^- \to \gamma \to X$ という対消滅の過程を経て全エネルギーが粒子を生成するのに使われるため、非常にクリーンなイベントとして観測することが出来る。従って電子 · 陽電子衝突型の加速器は粒子の寿命や質量、崩壊幅など各パラメータの精密測定を行なう実験に非常に適している。

現在 warm から cold への技術の見直し中であるが、warm 時の ILC 加速器の完成予想図を図 1.1 に示す。

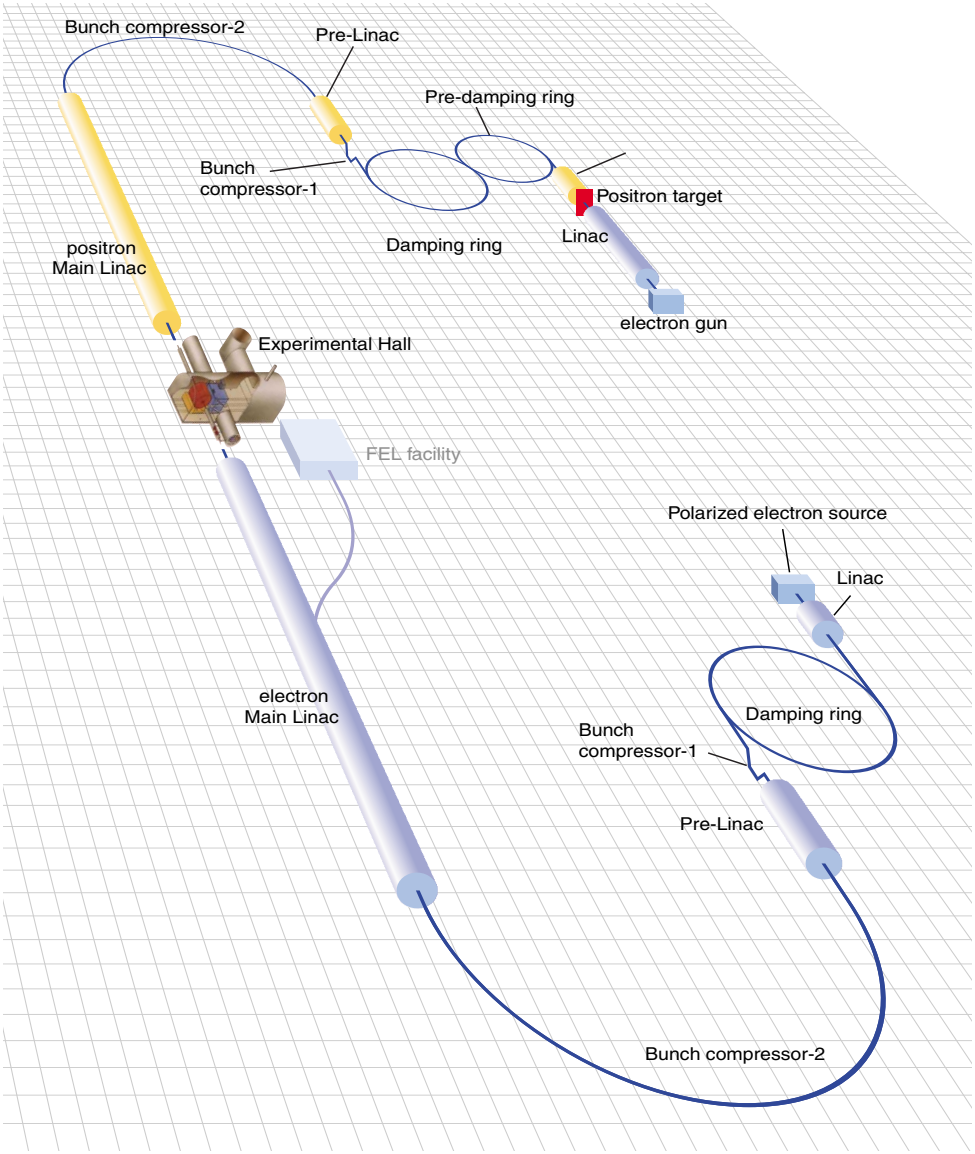

図 1.1: ILC 加速器の完成予想図

1.4 ILC 計画で期待される物理

ILC 計画ではエネルギーフロンティアと呼ばれる TeV エネルギーの領域に到達することによって、今までにない新しい物理現象を探索することを目的としている。
表.1.2 は ILC 実験において目標、目的とされている物理現象である。

表 1.2: ILC 実験において発見、測定が期待される物理現象

物理探索、測定	観測粒子、崩壊過程等
観測精度の良い測定	トップクォークの各種パラメータの精密測定 $\Rightarrow e^+e^- \rightarrow t\bar{t}$
標準模型で予言される現象	質量の起原 ヒッグス粒子の探索 $\Rightarrow e^+e^- \rightarrow Z^0H^0$
標準模型を超えた現象	超対称性粒子 SUSY 粒子の探索 (スクォーク、スレプトン等) $\Rightarrow$ ヒッグス粒子の生成断面積、 崩壊分岐比の標準模型からのずれの測定 大統一理論 (GUT) の検証

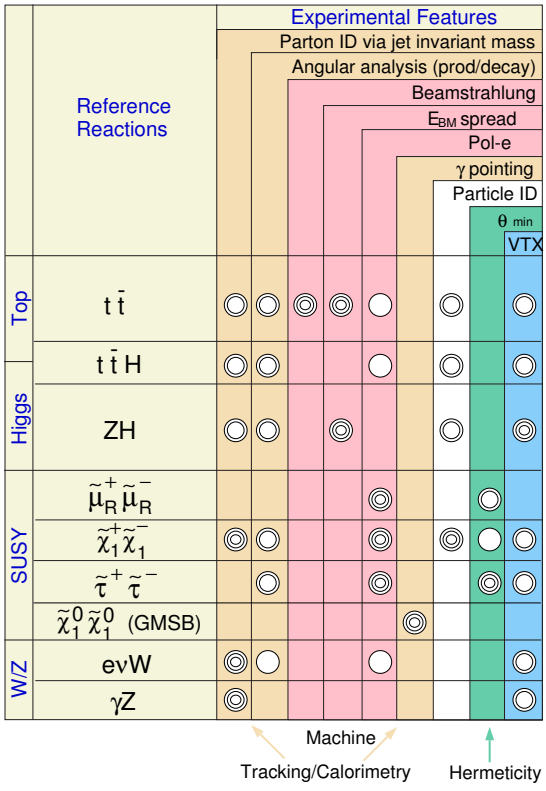

図 1.2: ILC 実験での検出が予想させる粒子

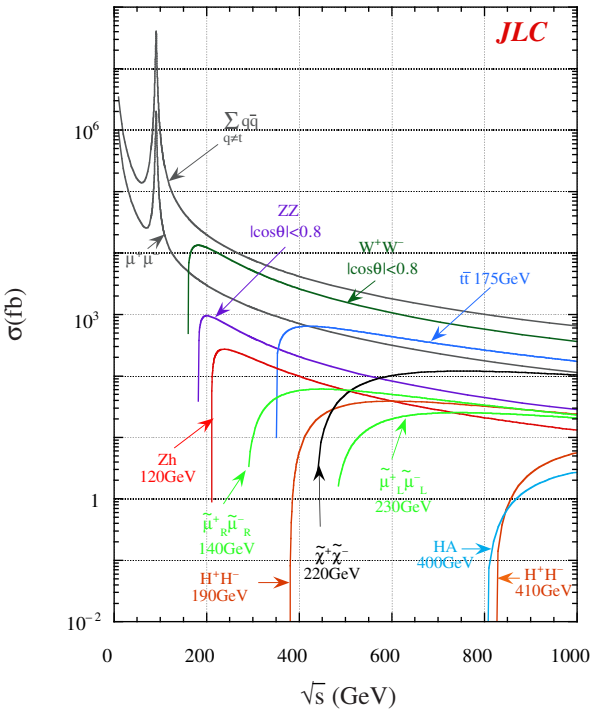

図 1.3: 粒子の反応断面積とエネルギーの関係

図.1.3 からも分かるように、ILC 加速器の重心系のエネルギー $\sqrt{S}$ は 120GeV 程度の位置でもっとも生成断面積が大きい。また LEP EW グループの 2003 年春現在で標準模型

ヒッグス粒子の質量には上限、下限が与えられており

$$116GeV < m_H < 211GeV\ (95\%C.L.)$$

の間に観測されることが期待されている。

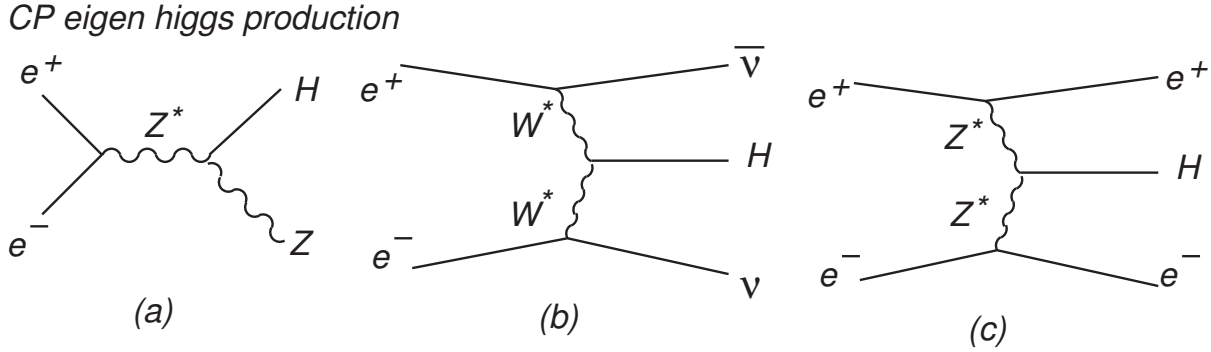

図 1.4: ヒッグス粒子の生成モード

表 1.3: 超対称性粒子 (SUSY 粒子)

属性	スピン	粒子	
クォーク族	0	squark	$\begin{pmatrix}\tilde{u}\\ \tilde{d}\end{pmatrix}\begin{pmatrix}\tilde{c}\\ \tilde{s}\end{pmatrix}\begin{pmatrix}\tilde{t}\\ \tilde{b}\end{pmatrix}$
	1/2	quark	$\begin{pmatrix}u\\ d\end{pmatrix}\begin{pmatrix}c\\ s\end{pmatrix}\begin{pmatrix}t\\ b\end{pmatrix}$
レプトン族	0	slepton	$\begin{pmatrix}\tilde{\nu_e}\\ \tilde{e}\end{pmatrix}\begin{pmatrix}\tilde{\nu_\mu}\\ \tilde{\mu}\end{pmatrix}\begin{pmatrix}\tilde{\nu_\tau}\\ \tilde{\tau}\end{pmatrix}$
	1/2	lepton	$\begin{pmatrix}\nu_e\\ e\end{pmatrix}\begin{pmatrix}\nu_\mu\\ \mu\end{pmatrix}\begin{pmatrix}\nu_\tau\\ \tau\end{pmatrix}$
ヒッグス粒子	0	Higgs boson	$\begin{pmatrix}\phi_1^0\\ \phi_1^0\end{pmatrix}\begin{pmatrix}\phi_2^+\\ \phi_2^0\end{pmatrix}$
	1/2	Higgsino	$\begin{pmatrix}\tilde{\phi_1^0}\\ \tilde{\phi_1^-}\end{pmatrix}\begin{pmatrix}\tilde{\phi_2^+}\\ \tilde{\phi_2^0}\end{pmatrix}$
ゲージ粒子	1/2	Gagino	$\tilde{\gamma}, \tilde{Z^0}, \tilde{W^\pm}, \tilde{g}$
	1	Gauge boson	$\gamma, Z^0, W^\pm, g$

またヒッグス粒子の超対称性におけるモデルとしては最小超対称標準模型 (MSSM) があり、このモデルにおいては表.1.3 のように H_1, H_2 のヒッグス 2 重項を二つ仮定している。

$$H_1 = \begin{pmatrix}\phi_1^0\\ \phi_1^-\end{pmatrix}, H_2 = \begin{pmatrix}\phi_2^+\\ \phi_2^0\end{pmatrix}$$

これらの ϕ_1, ϕ_2 を用いると MSSM モデルにおいてはヒッグス粒子は 5 種類存在することとなる。表 1.4 にそれらを示す。

表 1.4: ヒッグスボソンが形成する 5 種類のヒッグス粒子

ϕ_1^-, ϕ_2^+	$H^\pm$：荷電ヒッグス (Charged Higgs)
ϕ_1^0, ϕ_2^0 の虚部 CP-奇数 ($Im(\phi^0)$)	A^0：擬スカラーヒッグス (Pscudo scalar Higgs)
ϕ_1^0, ϕ_2^0 の実部 CP-隅数 ($Re(\phi^0)$)	η_1, η_2

表 1.4 のように 5 種類のヒッグス粒子の状態が存在し、更に η_1, η_2 の混合によって h^0, H^0 という二つの質量の固有状態を形成すると (但し 質量 $m_{h^0} < m_{H^0}$ である) h^0, H^0 は二つの質量の異なる中性ヒッグスとなる。

$$\begin{pmatrix} H^0 \\ h^0 \end{pmatrix} = \begin{pmatrix} \cos\alpha & \sin\alpha \\ -\sin\alpha & \cos\alpha \end{pmatrix} \begin{pmatrix} \eta_1 \\ \eta_2 \end{pmatrix}$$

従って最小超対称模型 MSSM で予言されているヒッグスは荷電ヒッグス ($H^\pm$)、CP-隅の質量固有状態の h、H ($m_h < m_H$)、CP-奇のスカラー (A) の 5 通りであり h 又は H のどちらかが標準模型で予言されている軽いヒッグス粒子 ($m_h \sim 140$ GeV) である。もう一方の粒子は質量が 500GeV 程度の重いヒッグス粒子となる。

標準模型を超える物理現象を調べる、或は超対称性粒子を探索する上では、これらのヒッグス粒子を探索するとともに標準模型からのずれを観測する上でも予言のハッキリしている最小対称性標準模型 (MSSM) との比較、則ちヒッグス粒子が発見された後には、ヒッグス粒子の性質の精密測定を行なうことが重要となる。
注目すべき測定量としては、

- ヒッグス粒子の生成過程
  $e^+e^- \to h^0Z^0$ の全断面積 σ_{total} の測定
- ヒッグス粒子の全崩壊幅
  Γ_{total} の測定
- ヒッグス粒子の $b\bar{b}$ への崩壊分岐比
  $Br(h \to b\bar{b})$ と σ_{total} との積
- ヒッグス粒子の 2 光子過程への崩壊分岐比
  $Br(h \to \gamma\gamma)$ の測定
- ヒッグス粒子の $c\bar{c}$ への崩壊分岐比
  $Br(h \to c\bar{c})$ の測定

等が考えられており、それらの値についてのシミュレーションなどが ILC グループにおいて精力的に行なわれている。これらの観測は容易ではないが、リニアコライダーを用いる場合においてはバックグラウンドの少ない精密測定が可能になる。

このように「ヒッグスファクトリー」として ILC 加速器を建設することは今後の物理の発展に大きく寄与すると考えられ建設が熱望されている。

1.5 ILC 検出器

ILC 実験では · 電子陽電子衝突実験である特性上、ハドロンコライダーに比べてバックグラウンドが非常に少ない、クリーンなイベントを観測することが出来る。そのため、ILC 実験において用いられる測定器では高精度、高分解能、高感度という高い性能を実現し、精密測定を行なう必要がある。

またトップクォークの研究やヒッグス粒子に対するバックグラウンドを抑えるため、b ジェット識別が効率良く行える必要がある。さらには、超対称粒子探索のため十分広い立体角を隙間なく覆う測定器でなければならない。これらの条件をシミュレーションを行って検討した結果、「標準測定器」として図 1.5 に示すような構成の測定器を考え、表.1.5 に示すような性能を達成することを目標としている。

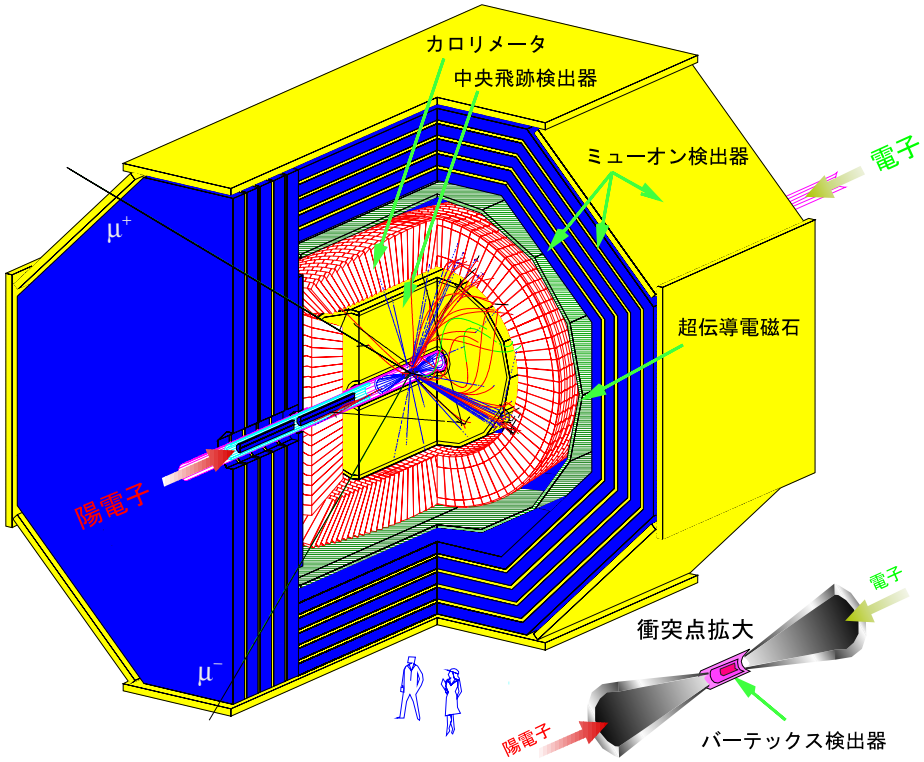

図 1.5: ILC 検出器の予想図

表 1.5: 各検出器に求められる性能

検出器	設定	性能
崩壊点検出器	$\cos\theta < 0.90$	位置分解能 $\sigma = 4.0\mu m$
(CCD)	ピクセル $25\mu m$、厚さ $300\mu m$	$\delta^2 = 7^2 + (20/p)^2/\sin^3\theta[\mu m]$
	4 層 $r = 2.4, 3.6, 4.8, 6.0cm$	$t_b = 50\%@93\%purity$
中間飛跡検出器	$\cos\theta < 0.90$	$\sigma = 4.0\mu m$
(Si-strip)	ストリップ幅 $100\mu m$、厚さ $300\mu m$	
	5 層 $r = 9, 16, 23, 30, 37cm$	
中央飛跡検出器	$\cos\theta < 0.70(fullsample)$	$\sigma_s = 1.0mm$
一般	$\cos\theta = 0.95(1/5samples)$	2 トラック分離 $2mm$
2Tesla	$r = 45 \sim 230cm, L = 460cm$	$\sigma = 100\mu m$
	サンプリング数 80	$\sigma_{Pt}/P_t = 3 \times 10^{-4}P_t + 0.1\%$
3 Tesla	$r = 45 \sim 155cm, L = 310cm$	$\sigma_s = 85\mu m$
	サンプリング数 50	$\sigma_P t/P_t = 3 \times 10^{-4}P_t + 0.1\%$
カロリメータ	EM=$27X_0$ (3 部分)	$\sigma/E = 15\%/\sqrt{E} + 1\%(EM)$
(Pb/Sci) 一般	HAD=$6.5\lambda_0$ (4 部分)	$\sigma/E = 40\%/\sqrt{E} + 2\%(HAD)$
	$\sigma_{\theta,\phi} = 24mrad(EM), 72mrad(Had)$	e/π ID = 1/1000
2Tesla	$\cos\theta < 0.985$(全厚)	
	$r = 250 \sim 400cm, z = \pm 290cm$	
3Tesla	$\cos\theta < 0.966$(全厚)	
	$r = 160 \sim 340cm, z = \pm 190cm$	
シャワーマックス	シンチストリップ (1cm 幅)	
μ 粒子検出器	$\cos\theta < 0.998$	$\sigma = 0.5mm$
(SWDC/RPC/TGC)	6 Super Layer	μ 粒子 ID

1.5.1 測定器構成

中央を走る電子、陽電子のビームパイプの衝突点を覆うようにして測定器は配置される。一番中央部分に見えるのが粒子の崩壊点を検出する崩壊点検出器 (バーテックス検出器)、崩壊点検出器を補うストリップタイプのシリコン中間飛跡検出器、その周りを粒子の運動量、飛跡を測定する中央飛跡検出器、更にその外側には粒子のエネルギー、不変質量を測定し、荷電中性粒子の同定を行なうカロリメータが備え付けられ、最外層には μ 粒子の同定を行なう μ 粒子検出器が配置される予定である。
検出器は以下の数字の小さい方から順にビームパイプ近傍の最内層から外層方向へ向かって配置される。

1. バーテックス検出器
   衝突点の極近傍で荷電粒子の飛跡を精密に測定する為に用いられる。特に B 中間子、D 中間子の崩壊点を測定することで b クォーク、c クォークの同定 (タギング) を行なうことが出来る。ILC 実験においてヒッグス粒子が生成されるチャンネルとして、例えば $e^+e^- \to Z^0H \to l^+l^-q\bar{q}$ のようなクォーク対が出るようなチャンネルがある。クォーク対はすぐにグルーオンを飛ばして結合しながら高密度のジェット状になって飛びでる。そのため高密度ジェット粒子の同定を行なうために CCD(電荷結合素子) のような 2 次元分解能の高いモジュールが想定されている。全体を覆う立体角、位置分解能としては

   $$\begin{aligned} \cos\theta &= 0.90 \\ \sigma &= 4.0\mu m \end{aligned}$$

   が要求されている。

2. 中央飛跡検出器
   中央飛跡検出器は荷電粒子が磁場中を通過する際の飛跡を検出することで、通過粒子の描く飛跡の曲率から荷電粒子の運動量を測定する検出器である。位置分解能、2 つの飛跡の分離能力として、

   $$\begin{aligned} \sigma &= 1.0mm, \sim 85\mu m\ (@\ 3\,Tesla) \\ 2\,track\ separation &= 2.0mm \end{aligned}$$

   が要求される。

3. カロリメータ
   粒子のエネルギー、位置、通過した粒子の同定等を行なうのがカロリメータである。ILC 実験においては、ヒッグス粒子が生成されるチャンネルにおいて Z_0, W 等の粒子が高密度ジェットを生成する。これらのジェットのエネルギーを正確に求め、再構成を行なうことでジェットになった元の粒子を同定することが出来る。特に不変質量 M を再構成して Z_0, W を分離する能力を要求される。またヒッグス粒子の崩壊分岐にはレプトンが出るモードもあり、特に $e^+e^- \to h^0Z^0 \to b\bar{b}\nu_e\bar{\nu_e}$ のように 2 つニュートリノが出る場合にはカロリメータにおいて正確に測る必要がある。この

ような中性粒子によって検出されない質量欠損分はカロリメータにおいてその他の荷電粒子を正確に測定することで検出することが出来る。中性粒子を検出する為にもカロリメータは出来る限り衝突点から 4π 方向をカバーできることが理想である。カロリメータにはデッドスペースが出来ないように作製されることが要求される。また高密度ジェットが発生した場合、このジェットの大元の粒子を同定する為にはジェットを細かく細分化して検出できる必要がある。そのため ILC 実験においては検出部分が小セグメント化された構造である必要がある。このような構成において

$$EM \ : \ \frac{\sigma}{< E >} = \frac{15\%}{\sqrt{E}} \oplus 1\%$$
$$HAD \ : \ \frac{\sigma}{< E >} = \frac{40\%}{\sqrt{E}} \oplus 2\%$$

のエネルギー分解能が要求されている。$(A \oplus B = \sqrt{A^2 + B^2})$

4. μ 粒子検出器
   カロリメータを通過して来た粒子を μ 粒子として同定するモジュールが μ 粒子検出器である。μ 粒子はレプトン対が出るようなモードにおいて明快な信号として検出されるので広い立体角を覆うと共にデッドスペースとなる領域を極力少くするように配置される。

1.6 シミュレータの必要性

以上のように基礎デザインから要求される各測定量への測定精度が決まったところで、これらの測定精度が実際に達成できるか否かを見極めるためのR&Dが行われている。我々のR&Dはシミュレータが必要な段階まできており、現在その開発が進行中である。我々の進めているシミュレータの構成を以下に示す。

1. イベント生成部分（粒子以前のパートンを粒子に変換）
2. Monte-Carlo Truth 生成部分（検出器シミュレーション部）
3. イベント解析プログラム（イベント再構成部）
4. 物理解析プログラム（物理解析部）

このうち、1. と 4. は既存のプログラムが存在するので、それを用いる。

3. の部分に関しては、JLC の解析フレームワークとして開発されてきた JSF[2] のフレームワークを用いて、まず JSF の下位にシミュレータ用のフレームワークを作成する。実際の解析部分は、ローカルパラメータの測定値の解析に際して開発された解析プログラムを組み込みながら、全体のトラッキングメソッドや、シミュレータ特有の解析ルーチン（Monte-Carlo Truth の情報を利用したカンニングルーチン）を開発する。

2. の部分に関しては、検出器と粒子の反応に関するモンテカルロ・シミュレータであるGeant4をベースに開発が行われてきており、JUPITER（**J**LC **U**nified **P**article **I**nteraction and **T**racking **E**mulato**R**）と名付けられ、さらに 3. のイベント再構成部分は JUPITER の衛星プログラムとして Satellites と名付けられた。それぞれ、Geant4[5] と ROOT[6] をベースにC++言語を用いたオブジェクト指向技術を最大限に利用した設計になっている。これまで、JUPITER はフレームワークの第一段階の開発を終え、各検出器の実装が進行中である。

1.7 本研究の主旨

JUPITER は各検出器の実装が現在も進行中であるが、検出器のインストールやメンテナンスのしにくさなどを問題としこれらの点を改善する案が求められていた。

この改善案として JUPITER のフレームワークに汎用的なデータフォーマットである XML[11] のインターフェースを追加する方法が挙げられる。我々は検出器インストールの一部を XML データとして置き換え自動生成させる部分を追加した。

以下の章ではまずシミュレータについて概観し、現在の JUPITER フレームワークとその問題点を報告する。そして XML データをどのように用いれば、JUPITER 上でインストール部分を自動生成させることができメンテナンスをし易くできるのかについて報告する。そして実際にそれを用い中央飛跡検出器の部分のインストールを XML データとして置き換えた。以下それを報告する。

第2章　高エネルギー実験で用いられる測定器フルシミュレータについて

この章の概説

この章では、まず粒子の測定器、つまり粒子と物質の相互作用をシミュレーションするとは何かを説明する。さらにその際に用いるツールキット[1]である Geant4 について述べ、次に Geant4 を用いて作成される高エネルギー実験で用いられる測定器フルシミュレータについて説明する。

2.1　粒子と物質の相互作用のシミュレーション

高エネルギー実験に限らず粒子と物質の相互作用を検出する測定器シミュレータには一般的に次が求められる。

- 測定器の構造をプログラム内で組み立てることができる。
- モンテカルロ法に基づいた物質内での粒子の相互作用をシミュレートすることにより入射した粒子をプログラムが自動的に輸送できなければならない。

従って、物質内で粒子の相互作用をシミュレートする方法であるモンテカルロ法が測定器シミュレーションの核となる。

モンテカルロ法とは乱数を用いた統計サンプリングを何度も行なうことにより近似解を求める数学的手法である。

以下で粒子と物質の相互作用に関する、モンテカルロ法について基本的概念である「粒子が今どこの物質内にいるかに関わらず乱数を振ることができる」ということを説明する。

基本的概念

次を定義する。

[1]ツールキットとは有用でかつ汎用的な機能を提供するために設計された、関連し合う再利用可能なクラスの集合、すなわちクラスライブラリである [13]。Geant4 は、放射線と物質の相互作用を扱うためのツールキットである。また、C++の入力ストリームライブラリや STL もツールキットである。ツールキットはアプリケーションの特定の設計には立ち入らない。アプリケーションの助けとなるような機能を提供するだけである。これはまた、共通機能の再コーディングを避けることにもつながる。ツールキットはコードの再利用である。いわば、サブルーチンライブラリのオブジェクト指向版と言える。
ツールキットの設計はアプリケーションの設計よりもはるかに難しい。なぜならば、ツールキットは多くのアプリケーションで有用でなくてはならないからである。さらにツールキットの提供者は、どんなアプリケーションがあるのかや、それらの特別のニーズを知る立場にいないからである。したがって、ツールキットの柔軟性、さらには適用可能性や有効性を制限してしまうことに [13]

- $P(x)$ 粒子が距離 x 進んだ後、物質と相互作用しない確率。
- wdx 粒子が位置 x と $x+dx$ の間で物質と相互作用する確率。

ここで w は

$$w = N{\cdot}\sigma$$

である。N は単位ボリューム内のターゲット粒子の数、σ は相互作用の cross section である。

したがって、この定義から $P(x+dx)$ は粒子が位置 $x+dx$ で物質と相互作用しない確率であるから $P(x)$ と wdx を用いて次のように表せる。

$$P(x+dx) = P(x)(1-wdx)$$

ここで $1-wdx$ は dx の間で粒子が物質と相互作用しない確率である。この微分方程式を解くと

$$P(x) = exp(-wx)$$

となる。ここで $P(0)=1$ とした。

相互作用の生成

位置 x から $x+dx$ 内で粒子が物質と相互作用する確率を $P_{int}(x)$ とすると上記により

$$P_{int}(x)dx = P(x)wdx$$

である。ここで $P_{int}(x)$ を PDF(**P**robability **D**ensity **F**unction) と呼ぶ。一方、これの積分

$$\int P_{int}(x)dx = \int P(x)wdx = \int wexp(-wx)dx = 1-exp(-wx)$$

を CDF(**C**umulative **D**istribution **F**unction) と呼ぶ。

ここで、

$$\eta = 1-exp(-wx)$$

はとなる。この η は $[0,1]$ の一様乱数である。ここから

$$x = -ln(1-\eta)/w$$

となる。

異種混成の物質の中での粒子と物質の相互作用の生成

上記で

$$x = -ln(1-\eta)/w$$

となったがこの x は長さの次元を持ち物質に依存する。[2]したがって粒子を物質内で輸送する際に物質に依存しない無作為抽出ができない。しかしながら、w を左辺に移した

$$xw = -ln(1-\eta)$$

[2]w は定義により物質に依存している。

の右辺は物質には依存しない。したがって次のように平均自由行程 λ を定義する。

$$\lambda = \int xP(x)dx / \int P(x)dx = 1/w$$

したがって、この λ を用いて xw は

$$x/\lambda = -ln(1-\eta)$$

と書き直すことができる。よって平均自由行程 λ を単位とすることによって粒子がどの物質と相互作用しているのかに依存しない形で一様乱数を与えることができる。この $[x/\lambda]$ を Number of Mean Free Path(NMFP) と呼ぶ。

粒子輸送

粒子は段階的にステップ踏んで行くやりかた (stepwise manner) で輸送されていく。Step を図 2.1 に示す。

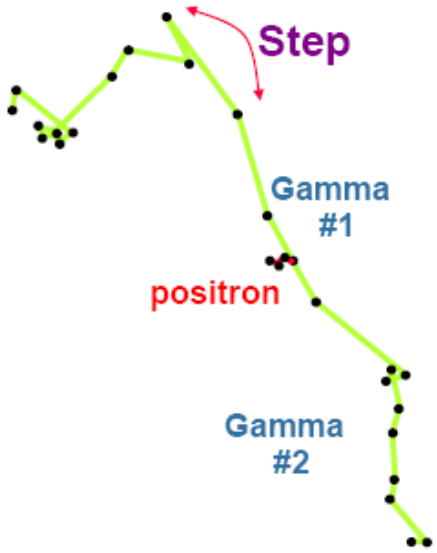

図 2.1: 水中の 8MeV の positron の対消滅

以下でこの粒子の輸送のされ方を説明する。[3]

1. step の最初で、粒子に関連付けられたそれぞれの物理プロセスに対する NMFP は物質に独立なやり方で無作為抽出される。
2. 現在粒子がいる位置の物質における cross-section を用いて NMFP を PL(Physical Length) に変換する
3. 最も短い PL を持つプロセスを step 長と決める。
4. 粒子を上で決められた step 長だけ粒子を輸送する。

[3] もう一度、詳しく Geant4 の仕組みを説明する箇所で述べる。

5. 粒子が相互作用の後、生きていたら、もう一度 NMFP に対して乱数を振り 1. から繰り返す

6. もし粒子が相互作用で消滅したら、輸送は終了。

以上、粒子と物質の相互作用のシミュレーションにおける基礎概念を説明した。

これらの step-by-step な粒子の輸送法を用いて測定器のシミュレーションを行うツールキットとして Geant4 がある。我々ILC 日本グループの測定器モンテカルロシミュレータもこの Geant4 をベースとして開発中である。以下の節では Geant4 について概略を説明する。

2.2 Geant4

2.2.1 Geant4 とは何か

Geant4(**GE**ometry **AN**d **T**racking **4**）とは粒子と物質の相互作用をシミュレーションする汎用ソフトウェア・ツールキットである。高エネルギー物理学実験 (HEP, High Energy Physics) で使われる測定器の振る舞いをシミュレーションするのを第一義として開発されたが設計段階から、HEP 以外の分野での応用も考慮されていた。現在では宇宙、医療をはじめとする広い分野でも使用されている。

2.2.2 開発の歴史

Geant4 は国際共同研究として開発され、国際協力グループにより維持されている。以下に開発の歴史を示す。

1994 年 12 月

日本と CERN が中心となり，世界 9 カ国から約 30 名の研究者が参加して R&D 開始した。

1998 年 12 月

Geant4 version1.0 を世界公開．R&D 終了.

1999 年 4 月

維持開発の世界共同グループ結成（>100 名)。日本は中心メンバーである。

2005 年 2 月

現在、version 4.7.0 である。年に 2 回の Public Releases（および必要に応じた minor patch release) がある。

2.2.3 Geant4 のしてくれること

Geant4 はシミュレーションしたい事象に含まれている粒子を、物質および外部電磁場との相互作用を考慮しつつ、次のいずれかの条件が成り立つまでトランスポート（輸送）する。

1. 運動エネルギーがゼロになるまで。
2. 相互作用により消滅するまで。
3. ユーザか指定するシミュレーション空間の境界（世界の果て）に到達するまで。

さらに、粒子のトランスポートのさまざまな段階で、ユーザがシミュレーションに介在できる手段を提供してくれる。例えば、

- 測定器の有感部分に粒子が入ると、その時点での粒子の運動学情報を用いて、ユーザが任意のデータ処理を行える。
- 粒子をトランスポートする途中のステップで、ユーザが任意のデータ処理を行える。
- 一つの粒子のトランスポートの最初と最後に、ユーザが任意のデータ処理を行える。

等である。

また Geant4 は以下を提供している。

1. シミュレーションを対話的に或いはバッチ処理として実行する手法。
2. シミュレーション過程を各種グラフィック・ツールで可視化できる手段を提供する。
3. シミュレーションのチェック、デバッグ・ツールを提供する。

2.2.4 Geant4 のしてくれないこと

Geant4 してくれないことを以下に説明する。

必要最低限の情報はユーザが Geant4 に知らせなければならない。

ユーザーは粒子をトランスポートするにあたり、最低限必要となる以下の 3 つの情報を与えなければならない。

1. 測定器の構造情報
2. シミュレーションしたい事象に含まれる粒子の種類、始点と運動量ベクトル。
3. 粒子をトランスポートするにあたり、Geant4 が考慮すべき粒子および相互作用の種類。

さらに、シミュレーションを意味あるものにするには、以下の情報のいずれか或いは全てをユーザから与えられなければならない。

1. 外部電磁場がある場合、その分布情報。
2. 粒子トランスポートのさまざまな段階でユーザが行いたいデータ処理。

Geant4 はツールキットである。

一般的に使える実行可能なシミュレーションプログラムの提供はしてくれない。[4]Geant4 はツールキット[5]であり、ユーザは用意されているコンポーネントを使って自分で必要なシミュレーションプログラムを組み立てなければならない。具体的にはメインプログラムをユーザが書かなければならないということである。

正しいシミュレーション結果は自動的には与えられるわけではない。

ユーザは必ず、自分がどのような条件でシミュレーションをしているかを理解する必要がある。例えば

- いくつかのパラメータ（例：Production Threshold）の調整が必要。
- Geant4 の用意している粒子相互作用のうち、適用範囲の正しいものを選択しているか。

等が挙げられる。

2.2.5 Geant4 は C++言語で実装されている。

Geant4 を使うのに最低、知らなければならないことは C++言語である。

なぜ C++が使われているのか？

- FORTRAN で大規模ソフトウエアを開発することの反省から。
- オブジェクト指向言語はシミュレーションに向いている（後述）。
- C++は他のオブジェクト指向言語と比較すると処理速度が優れている。

[4]例題は多数ある。
[5]ツールキット、アプリケーション、フレームワークなどの用語については後述

- 使用 community が大きく、多くの computing platform で使うことが可能。
- ANSI/ISO 標準言語である。

なぜオブジェクト指向言語を使うのか？

モンテカルロ・シミュレーションはシステムをモデル化し、その振る舞いを調べる。システムは実際には実体 (object) から構成されているので、その実体をオブジェクト指向言語で直接プログラムの object にマップするのは自然である。例えば、素粒子を「素粒子オブジェクトとして表現」、測定器を「測定器オブジェクトとして表現」などである。オブジェクト指向プログラム言語はもともとシミュレーション言語として出発した。したがって、測定器シミュレーションのプログラム言語として使うのは自然である。

以上で Geant4 の概要を説明した。次節で Geant4 を用いたアプリケーションプログラムを作成する方法を見ていく。

2.2.6 Geant4 ライブラリの使い方概説

Geant4 でアプリケーションプログラムを実装するユーザは以下、最低 3 種類の情報を用意しなければならない。

1. 測定器の構造情報（Geometry）。通常、一番時間がかかる作業である。
2. シミュレーションしたい事象（Primary Event）に含まれる粒子の種類、始点と運動量ベクトル。
3. 粒子をトランスポートするにあたり、Geant4 が考慮すべき粒子（Particles）および相互作用の種類（Physics Processes）。

これらをを Geant4 へ与る際の考え方を以下の節で説明する。

2.2.7 測定器の構造（Geometry）の作成

測定器を構成する要素（Volume）という考え方を説明する。

測定器の構造情報（Geometry）

測定器はいろいろな構成要素からできており、Geant4 ではこの要素を Volume とよぶ。測定器のどの部分を一つの Volume として表現するかは、ユーザがシミュレーションで得たい結果を考慮して決めなければならない。測定器の全ての詳細構造を Geometry 定義に入れることは通常は避ける。シミュレーションに必要最低限な Volume を見極め、それのみを定義する。これは通常一番時間がかかり、難しい作業である。

測定器の構造情報（Geometry）の作成 (ステップ 1)

測定器を構成する要素、すなわち Volume を表現する第１ステップはその幾何学形状を定めることである。Geant4 は Volume の幾何学形状を指定するために以下の３種類の立体表現（Solid）手法を用意している。

1. 円球、円筒、直方体、台形体など様々な基本立体要素を組み合わせることによる表現。これを Constructed Solid Geometry (CSG) とよぶ。

2. 複数の CSG を用い引き算、足し算、ユニオンなどの論理操作を行うことで表現。これを、Boolean Operations とよぶ。

3. Volume を形作る全ての面を関数を用いて表現。これを Boundary Representation(BREPS) とよぶ

これらの solids 表現に ISO 世界標準（ISO10303）の STEP 形式（Standard for the Exchange of Product ）を採用している．[6] STEP を採用することで，CAD とのデータ交換が可能となっている。

それぞれの Solid 表現の特徴

それぞれの Solid 表現 (CSG, Boolean Operation, BREPS) の特徴を以下に挙げる。

1. CSG はもっとも基本的な solid 表現法．粒子の輸送計算が高速であるが，非常に変則的な形状の volume 表現には向かない。

2. Boolean Operation は粒子の輸送計算に時間がかかる。

3. BREPS はもっとも自由度の高い表現方法。ただし，粒子の輸送計算に時間がかかる。

測定器の構造情報（Geometry）の作成 (ステップ 2)

ここまでで測定器の形状は作成できたので次に、測定器に物質情報を追加する。ここに Logical Volume という考え方がでてくる。ユーザーはそれぞれの Solid の情報とそれがどのような物質からできているかの情報を追加して Logical Volume とよばれているものを作る。物質の定義の仕方は Geant4 で用意された手法でユーザが行うが、よく使われる物質については定義例が豊富に用意されている。Logical Volume の特徴はそれがまだ空間

[6]STEP(Standard for the Exchange Product data model) は、正式名称 ISO10303(Product Data Representation and Exchange) と呼ばれる。STEP では製品データ (Product Data) の表現及びデータ交換のための標準規格を取り決めようとしており、製品のライフサイクル (設計、解析、製造、検査、利用、保守) 全般にわたり利用できるデータが含まれている。このデータの中には CAD の形状データなども含まれている。これらのデータは、STEP で規定している独自の言語 (Express) を使って表現し、CAD やその他のいろいろなアプリケーションから共通のデータを引き出せるようになる。CAD などのアプリケーションでは、STEP から提供される Express で記述されるツールを使って、機械的にデータを変換できるようになるため、異なる CAD システム間でデータのやりとりが行える。ただし、現実的にはいろいろな業種や業界により必要なデータの種類や構造が異なっているため、数多くのアプリケーションプロトコル (AP) という枠組みをつくり、業種別や作業別に規格を決めている。

的にはどこにも置かれていないということ測定器が同一の形／サイズ／物質からなる要素を複数もっている場合、それらを個別に定義する必要はなく、一つの Logical Volume を共用することで表現できる。複雑な測定器を表現するにはプログラムサイズの観点からこの考えが非常に重要となる。

測定器の構造情報（Geometry）の作成（ステップ 3）

測定器を組上げる Physical Volume という考え方を説明する。Logical Volume をお互いに相対的に配置していくことで測定器を組み上げる。この時の出発点となるのが World Volume である。すべての Volume は World Volume 内に置かれる。World Volume に対して Logical Volume を配置すると、Physical Volume が作られる。Physical Volume とは基本的には Logical Volume の情報に、相対位置情報が追加されたものである。

Logical Volume を配置する基準として使えるものを以下に挙げる。[7]

1. World Volume（Physical Volume として必ず定義する）
2. 既に定義されている他の Physical Volume
3. 既に定義されている他の Logical Volume

配置の基準となる Volume を Mother Volume、配置された Volume を Daughter Volume とよぶ。Daughter Volume が置かれると、もとあった Mother Volume の物質は Daughter のものに 置き換えられる。

測定器の構造情報（Geometry）の作成（その他の情報）

1. Logical Volume を配置するときの注意
   Volume 同士がお互いに部分的にオーバーラップさせることは許されない。例外は Boolean Operation を使うときのみである。
2. 電磁場について
   ユーザは測定器が電場／磁場を持つ場合、その分布情報を World Volume を基準に与える。必要に応じて、電場／磁場を任意の logical volume に対して与えることもできる。

2.2.8 Primary Event Generator の作成

Primary Event Generator とはユーザがシミュレーションしたい事象を発生させるものである。以下の情報を作る。

- 事象の発生点と発生時間

[7] 測定器の配置を表す PhysicalVolume の抽象クラスである G4VPhysicalVolume の継承クラスのコンストラクタに与える仮引数のこと。

- 事象中に含まれる粒子（通常複数個）の種類
- それぞれの粒子の momentum vector

標準で用意されている Primary Event Generator は

- ParticleGun とよばれる単一粒子の generator
- GeneralParticleSouce とよばれる、2 または 3 次元に広がった粒子発生源を持つ generator
- 既存の FORTRAN で書かれた event generator との inteface クラス．/HEPEVT/ common を介して PYTHIA, IsaJet 等とインターフェイス
- C++で書かれた event generator との HepMC クラスを通したインターフェイス（例題としてして提供）

である。標準で用意されている Generator が目的に合わない場合、標準例題で用意されているコード例を用いてユーザが自作する。

取り扱うべき Particles の指定

1. Geant4 が用意している particle クラス
   PDG[8]に掲載されている主な粒子（数百種類）、およびシミュレーションにのみ用いる特殊な粒子（例、Geantino[9]）が以下の 6 種類にグループ分けされて、前もって定義されている。

   (a) Lepton
   (b) Boson
   (c) Meson
   (d) ShortlivedParticle
   (e) Baryon
   (f) Ion

   ユーザは必ず使用する粒子の指定を行なわなければならない。

2. Geant4 グループが標準例題で用意しているコード例にもとづいて、ユーザが必要に応じてそれを編集することで、使用する particles を指定すればよい。

[8] **P**article **D**ata **G**roup
[9] 何も相互作用しない仮想的な粒子

2.2.9 使用したいPhysics Processesの指定

Physics Processes とは particles が物質を通過する際に起こる相互作用を総称的に表したものである。Geant4 には Physics Processes として sub-keV 領域から PeV 領域までの粒子相互作用が以下の分類で用意されている。

- Electromagnetic processes
- Hadronic processes
- Decay processes
- その他
  - Transportation process など

取り扱うべき Processes の指定

Geant4 が用意している Physics Processes の種類は膨大である。ユーザは必ず使用する processes の指定を行なわなければならない。Geant4 グループが標準例題で用意しているコード例にもとづいて、ユーザが必要に応じてそれを編集することで、使用する Physics Processes を指定する。

Particles と Processes の関係付け

ユーザが使用したい Particles と Physics Processes の指定を終了し、それを Geant4 に伝えると、Geant4 はそれぞれの Particle に対して、物理的に妥当な Physics Processes を関連付けてくれる (Applicability 情報)

Production Threshold（Cut Value）とは？

Physics Process により二次粒子 (Secondaries) が生成された場合、あるエネルギー以下の secondary particle はトランスポートされない。このエネルギー値を Production Threshold とよぶ。Geant4 はゼロ・エネルギーになるまで粒子をトランスポートするが、これは無限にゼロ・エネルギーに近い粒子を全てトランスポートすることではない。従って無限にゼロ・エネルギーに近い粒子までトランスポートしていると Geant4 の処理が遅くなる。[10]

Production Threshold （Cut Value）の設定

cut value は stopping range（空間的長さ）で与える（例： 1mm）。photon の場合は absorption length で与える。Cut value を range cut よぶ場合もある。標準的には cut value

[10]無限にゼロエネルギーに近い粒子を作ると Electromagnetic processes では赤外発散がおこる。赤外発散については本論文の範囲を超えるので述べない。

は粒子に対して一つの値が設定される。全ての Physics Processes は secondaries を作る際にこの値を参照する。作られた二次粒子のエネルギーが指定された range を走るだけの energy をもっていないと、その粒子はトランスポートされない。トランスポートされない粒子の energy は生成された vertex 位置での energy loss として扱う。

これを図 2.2 に示す。

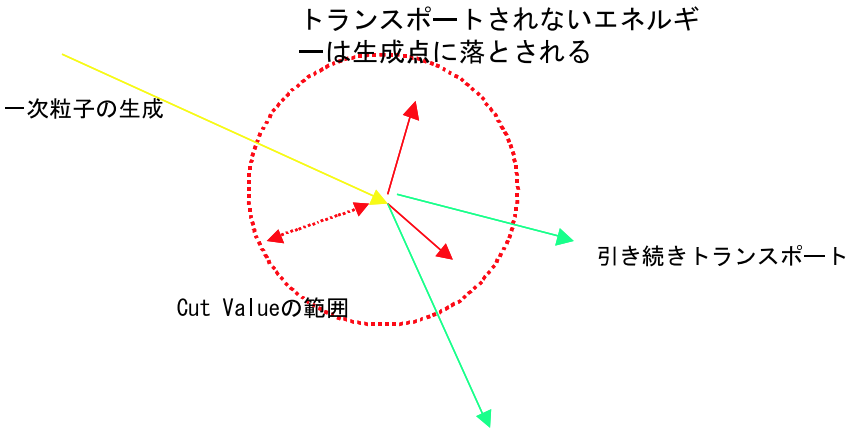

図 2.2: 作られた二次粒子のエネルギーが指定された range を走るだけの energy を持っていないとその粒子はトランスポートされない

ただし Production Threshold は無視される場合もある。physics process は必ず cut value を守らなければならないわけでない。以下例を挙げる。

1. pair creation
   この場合、positron は zero energy でも必ず作られる。これは positron annihilation process を無視しないためである。
2. 境界領域の近くを走る粒子

単一粒子について一つの cut value が原則であるが、より自由度の高い設定もできる。

- Material ごと
- Logical volume ごと

ここまでで Geant4 に粒子をトランスポートさせるためにユーザが知らなければならない最低の概念の説明を終えた。

2.2.10 測定器応答のシミュレーション

ユーザがシミュレーションで実際にやりたいことは、Geant4 が粒子をトランスポートさせる過程に介在して，その粒子に対する測定器の応答をシミュレーションしたいということである。これを Geant4 で行うには、次の考え方を理解する必要がある。

1. Sensitive Detector
2. Hit
3. Digitizer

2.2.11 Sensitive Detector

一般的に測定器は粒子応答を観測する部分と (sensitive part) とそうでない部分 (insensitive part) に別れる。sensitive part を Geant4 は Sensitive Detector とよぶ。Sensitive Detector を指定するには Logical volume を単位として、それが sensitive であるか insensitive であるかを指定することができる。具体的には、ユーザは Sensitive Detector オブジェクトをつくり、そのなかで粒子の運動学情報を使って Hit（次を参照）オブジェクトを作る手法を定義する。この Sensitive Detector オブジェクトで表現される機能を対応する Logical volume に付加する。

2.2.12 Hit

Hits とは

Hits とは Sensitive Detector 中の粒子がトランスポートされる際の各ステップでの粒子の状態のスナップショット情報である。スナップショット情報として記録したい内容はユーザーが独自に定義する。典型的な情報例としてはステップの前後での粒子の位置・時間情報、ステップの前後での粒子の運動量、ステップでのエネルギー損失などである。

HitsCollection とは

一つの事象内で、ある Sensitive Detector 中で作られた一連の Hits を一まとめにするオブジェクトである。例えば測定器 A と測定器 B が Sensitive であるとき、測定器 A と測定器 B で作られた Hits は別々の HitsCollection にまとめられる。

Hits をどの様に使うか（非常に単純な使用例）

- 一つの事象ごとに、作られた Hits Collections の内容を外部ファイルに書き出す。必ずしも、Hits ないし Hits Collections オブジェクトとして書き出す必要はない。（例、ROOT オブジェクトや Ntuple）
- Geant4 とは独立なユーザ・プログラムで外部ファイルから Hits の内容を読み込み、任意の処理をする。（例、ヒストグラムを作る、等）

2.2.13 Digitizer とは

Digitizer とは Hits を用いて、測定器の応答（例えば、生データ形式）を Geant4 シミュレーションの枠組みの中で作り出すために用いる。しばしば、ユーザはこの枠組みを用いることはせずに、Hits の単純な使用例で示した手段で測定器の生データ形式を作り出している。Digitizer は絶対につかわなければならないというものではない。

2.2.14 Geant4 の仕組み

ここまでで Geant4 をツールキットとして用いて測定器シミュレータを組み上げるための概念の概略説明を終えた。以下では Geant4 の中の仕組みを見て行く。

2.2.15 Geant4 はどのように粒子をトランスポートするか

Geant4 は与えられた事象に含まれる一つ一つの粒子に対して、Step を切ってトランスポートしていく。これを Step Transportation Algorithm と呼ぶ。

Geant4 は Step をどのように決めるか

1. 一つの粒子をトランスポートするにあたり、まず最初に、Geant4 はその粒子が起こすことの可能な Physics Processes の一つ一つに対して、乱数を用いて相互作用が起こるまでの距離を Interaction Length 単位で決定する。粒子の運命はトランスポートを開始する段階で原則的に決まってしまう。ただし、粒子が相互作用をおこしても、その粒子が消滅しない場合（例えば、gamma が compton scattering した場合）には、改めて、その相互作用の運命付けを行う。例として Positron を考える。Positron は以下の Physics Processes を持つが、それぞれについて相互作用が起こるまでの距離（Number Of Interaction Length Left = NILL）を乱数で決める。

   - Bremsstrahlung NILL = NILLb
   - Ionisation NILL = NILLi
   - Positron annihilation NILL = NILLp
   - Transportation 粒子の現在位置から最も近い volume 境界までの距離（例外）

2. 粒子が現在いる場所の物質を考慮して、NILL を実際の距離（Physical Interaction Length = PIL）に変換する。

   これを図 2.3 に示す。

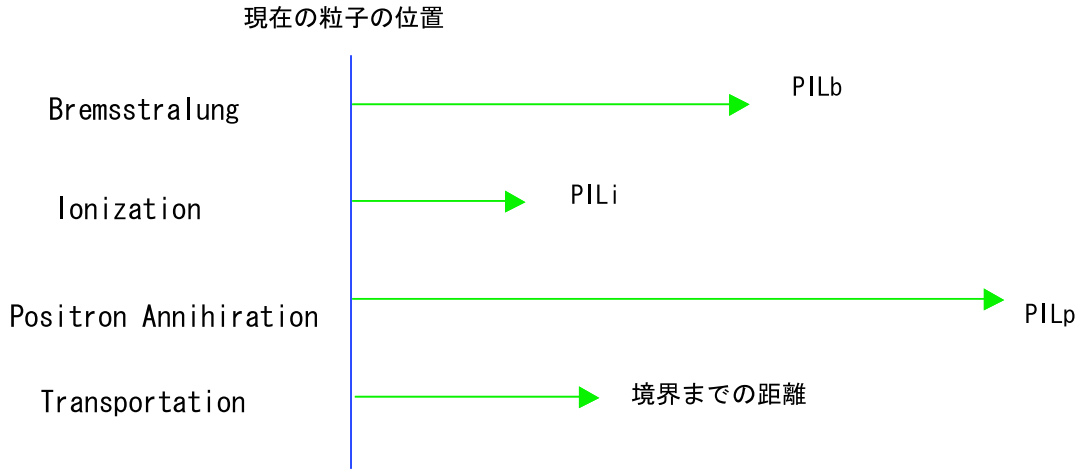

図 2.3: 粒子が現在いる場所の物質を考慮して、NILL を実際の距離（Physical Interaction Length = PIL）に変換する

3. PIL の最小値をもつ PhysicsProcess が Step を決定する。上の例では Ionisation process
4. 決定された Step サイズだけ粒子をトランスポートさせる
5. この Step で二次粒子が発生した場合、それは Geant4 が所持しているスタックに一旦、保存され、後々、あらためてトランスポートされる
6. 粒子の持っている Physics Processes の全ての NILL をトランスポートした距離分だけ減らす
7. もし Step の終了後、粒子が Sensitive Volume 領域に入った場合、対応する Sensitive Detector をよぶ
8. 1. に戻る

2.2.16 Geant4 シミュレーションの全体進行と階層構造

Geant4 はシミュレーションの全体の進行を階層単位で進めていく。それぞれの階層にはマネージャとよばれるオブジェクトが存在し、その階層でのシミュレーションは階層マネージャが制御する。それぞれの階層で作られる情報は階層を表現する階層オブジェクトに保存される。それぞれの階層でユーザがシミュレーションの進行に介在できる手段が用意されている。[11]全体進行と階層マネジャー、そしてユーザ介在手段を表 2.2.16 にまとめる。以下これらを説明する。

階層オブジェクト	階層マネジャー	ユーザ介在手段
Run	RunManager	UserRunAction
↓	↓	↓
Event	EventManager	UserEventAction
↓	↓	↓
Track	TrackingManager	UserTrackingAction
↓	↓	↓
Step	SteppingManager	UserSteppingAcrion

2.2.17 Run と RunManager

Run とは

同一の測定器 Geometry，Physics Processes の組など、同一の条件の下で，決まった数だけの事象 (Events) をシミュレーションすることを一つの Run を実行するという。実験で一定の条件のもとに複数のデータを取得するのを‘ run ’とよぶのと同じ概念である。Run が持つ情報は

[11] これらのマネジャーは Singleton パターンで実装されている。デザインパターンについては後の節で簡単に述べている。

- Run number
- 処理すべき事象の総数

等である。

RunManager とは

一つの Run のシミュレーション全体を制御するシミュレーション進行の親分のようなものが RunManager である。例えば、Geant4 を用いた測定器シミュレーションはユーザが RunManager に対して beamOn ということで開始される。

RunManager の代表的な役割は

- ユーザから測定器 geometry 情報を受け取り、Geometry をプログラム内で使えるように初期化する。
- ユーザから使用する粒子と相互作用の情報を受け取り、それらを使えるように初期化。
- ユーザが指定する Primary Event Generator からシミュレーションすべき事象を一つずつ受け取り、それを次の階層の EventManager にわたす
- Run に関する情報（Run オブジェクト）の保持
- UserRunAction の受け付け

等である。

2.2.18 Event と EventManager

Event とは

例えば，衝突型加速器実験において衝突によって生成される全粒子 (Primary Particles) が一事象を構成する。Primary Particles が作る二次粒子も Event に含まれる。Event が持つ情報は

- Primary Particles の vertexes 運動量等の情報
- Hits 情報

等である。

EventManager とは

EventManager は一つの Event に含まれる全粒子のトランスポートを制御する。代表的な役割は

- Run Manager から一つの事象を受け取り、それに含まれる粒子を一つずつ、次の階層の TrackingManager にわたす。
- 相互作用でできた 2 次粒子を保持し、それも順次、 TrackingManager にわたす。
- 粒子を TrackingManager にわたす順序の制御を担う。
- Event に関する情報（Event オブジェクト）を保持する。
- UserEventAction の受け付け

等である。

2.2.19 Track と TrackingManager

Track とは

実験で実際に飛んでいる粒子を表す。したがって、その粒子の種類、ある瞬間の位置、運動量などの情報をもつ。Particle という名前を使わなかっのは、static な粒子情報（PDG の情報）を保持するオブジェクトを Particle としている為である。[12]

Track が持つ情報は

- PDG 情報 (ParticleDefinition オブジェクトをとおして)
- Step 後の Particle 位置情報
- Track の最初の Vertex 位置、 運動量情報
- 全トラック長さ
- トレースし始めてから経過時間

等である。

TrackingManager とは

一つの Track を次のいずれかの条件を満たすまでトランスポートさせる。

- ゼロ・エネルギーになる
- 相互作用で消滅する
- world volume の端に到達するまでトランスポートする

代表的な役割は

- Event Manager から一つの Track を受け取る

[12]後に述べる Trajectory と混乱しない様注意する。

- Track を一つの Step だけ進めることを次の階層の SteppingManager に命令する
- Track に関する情報（Track オブジェクト）の保持
- UserTrackingAction の受け付け

等である。

2.2.20 Step と SteppingManager

Step とは

Step とは粒子トランスポートの最小単位．一つの Step の前後での粒子の物理情報を保持する。一つの Step の開始時点の情報は Pre-Step Point，終了後の情報は Post-Step Point で保持する。この概念図をモンテカルロシミュレータの構造を図 2.4 に示す。に示す。

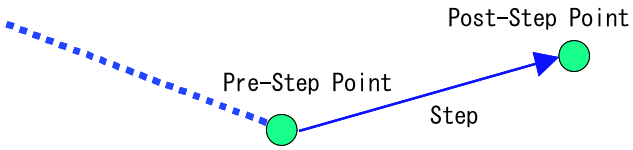

図 2.4: Pre-Step Point と Post-Step Point

[13]

SteppingManager とは

Track を一つの Step だけ進ませるのに必要な処理全体を制御する。代表的な役割として

- Tracking Manager からの命令を受け、その Track が起こす可能性のある相互作用と現在いる Volume の境界までの距離を考慮して Step の長さを決定する。
- 決定した Step 長だけ Track をトランスポートする。
- Step の終点が Sensitive Volume であれば対応する SensitiveDetector をよぶ。
- UserSteppingAction を受け付ける。

等がある。

[13]前述の通り、Geant4 は Step が終了した際、そこが Sensitive Volume 領域であるなら対応する Sensitive Detector を呼ぶ構造になっている。Step が測定器間の境界を超えて新たな SensitiveDetector 領域に入った際には、それまで飛んでいた領域の情報は Post-Step Point が持っているのではなく Pre-StepPoint が保持している。したがって Volume の情報を得るには必ず Pre-Step Point に聞かなければならない。ここは間違い易いので注意すること。

2.2.21 Trajectory について

Trajectory とは

Track 及び Step は粒子トランスポートでの粒子の最新物理状態の情報のみを保持している。これに対して，Trajectory は粒子トランスポートの全履歴情報を Step 単位で保持している。Trajectory は結果的に膨大な情報を保持することになるので，ユーザーが明示的に要求しない限り Geant4 はそれを作らない。

Trajectory Point とは

一つの Trajectory は複数の Trajectory Points から成る．一つの Step のトランスポートが完了するたびに Step が持つ情報を Trajectory Point へ移す。Trajectory Point に何を保持させるかはユーザーが決める。

2.2.22 User Interface と Graphics

User Interface とは

Geant4 には様々なユーザ・コマンドが用意されており，それらを実行することでシミュレーションの条件を設定することができる。ユーザーコマンドはユーザが作るプログラム内から実行することもできるし，interactive mode でターミナルあるいはスクリプトからも実行できる。

Graphics

組み込んだ geometry が正しいかを確かめる，あるいは，発生させた粒子の測定器内での振る舞いを観察するには，graphics は不可欠である．Geant4 では必要に応じて様々な graphics を使えるように作られている．仕様可能な主な graphics は以下のとおり．

- X11
- PostScript
- OpenGL/OpenInventor
- VRML
- DAWN

VisualizationManager が Graphics に関連する動作を制御する。

2.3 測定器フルシミュレータ

2.3.1 測定器モンテカルロシミュレータ

ここでは、一般的に高エネルギー物理実験で用いられるシミュレータの役割について説明する。ILC を例に取ると、まず始めに目標とする重要な物理を明らかにしなければならない。このターゲットが決まると、次はビームエネルギー、ルミノシティ、バックグラウンドなどのマシンパラメータの設定、さらにそれを受けて運動量分解能、飛跡再構成の精度、エネルギー分解能、粒子同定の精度など、検出器のパラメータの設定が行われる。これらすべてがモンテカルロシミュレータにより検証され、目標とする物理が達成できるかどうかの判断を行う。こうした一連の過程を繰り返し、各パラメータの調整を行い、数多くの R&D を経て最終的なパラメータが求められる。モンテカルロシミュレータは、プロジェクトの重要な３つの根幹である物理、加速器、検出器のパラメータを相関づける唯一の手段であり、それゆえに十分な計算速度と精度をもって、実験の特性をシミュレートできるものでなくてはならない。

モンテカルロシミュレータの構造を図 2.5 に示す。

2.3.2 イベント再構成を入れた測定器フルシミュレーション

測定器フルシミュレーションはまず、ビーム相互作用と Initial State Radiation を経て生成されるパートン（４元運動量で表される）からスタートする。パートンはすぐにシャワー、ハドロン化、崩壊などを起こし、最終的には安定粒子になる。これを２つの４元ベクトル (x^μ, p^μ) で表す。ここで出発点の粒子の x^μ は位置であり p^μ は運動量である。これらの粒子は、検出器の中を、検出器中の様々な物質と相互作用しながら走って行く。このとき粒子は、飛跡検出器中では（荷電粒子であれば）通常 Helix のパラメータで表される螺旋を描き、ヒットの形で飛跡を残す。カロリーメータの中では、それぞれ粒子の特性に従ってエネルギークラスターの形でシャワーの痕跡を残す。更に、必要であれば、これらのヒットやクラスターを ADC や TDC の信号の形に変換することもできる。

一方実際の解析と同様に、この逆の過程は、ADC や TDC の信号から始まって、最終的にはビーム衝突点で反応が起こった直後のパートンにまで戻っていくことでビーム相互作用で起こった情報を得ることができターゲットの物理が検地できるかどうかの判断を行う。これが、イベント再構成と解析の作業である。飛跡検出器の場合には、まず ADC と TDC のデータから、検出器のどこを粒子が通ったかについての情報（ヒット点）を作成する。次に、どのヒット点が同一の粒子から作られたものかについてのグループ分けを行い（Track Finding)、更に、グループ分けした点の集合を Fitting して Helix パラメータに変換する（Track Finding)。同様にして、カロリーメータでも個々のタワーの信号からもクラスターを再構成する（Clustering)。このようにして再構成された飛跡とクラスターを対応させ、Energy Flow の分解能を向上してから、再び２組の４元ベクトルで粒子の初期状態を表す。このような粒子の情報を多数集めて、ジェットの再構成を行い、最終的にはもとの 4 元運動量で表されるパートン再構成を目指す。

以上が、実験を、可能な限り忠実にモンテカルロシミュレーションに焼き直したとき

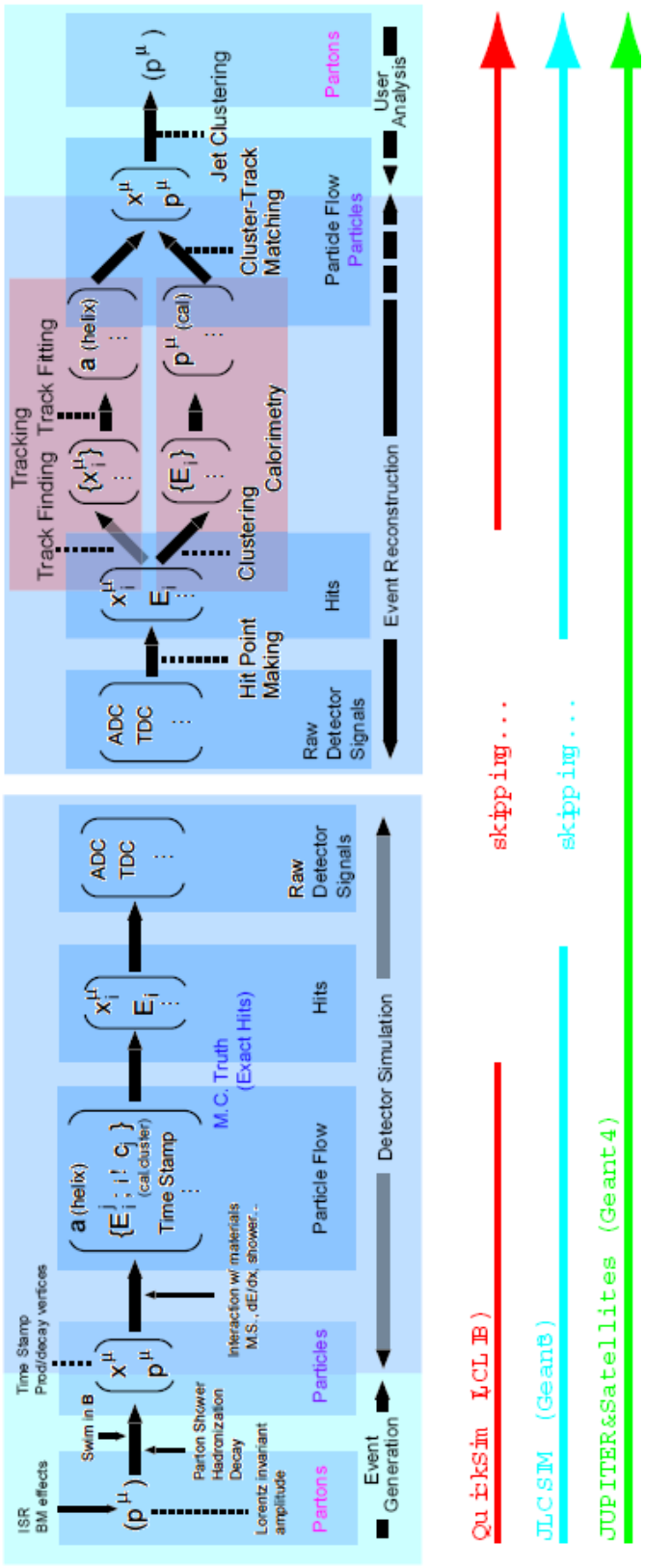

図 2.5: モンテカルロシミュレーションの流れ

の手順である。このようなシミュレーションをフルシミュレーションと呼ぶ。しかし、シミュレータの使命はただ実験を忠実に再現することだけではない。このように計算機技術が発展した現在でさえ、コンピュータが１つのイベントを生成する速度は自然のそれに敵うべくもないが、それでもいくつかの過程を省略することによって、若干シミュレーション時間を早くすることが出来る。このとき、どの部分を省略するかによって、シミュレーションのレベルを変えることが出来る。ILC では、図 2.5 の下部に示した通り、シミュレーションレベルによって、QuickSim、JIM[3]+Analysis Program、JUPITER&Satellites（開発中）などのシミュレータが存在する。

2.3.3 フルシミュレータ

クイック シミュレータが検出器の基本仕様を設定したり、物理計算を行う際に用いられるのに対し、フルシミュレータの使命はほとんど検出器の具体的な開発と、実際の実験における誤差の見積もりに集中している。まず、フルシミュレータは、検出器の全ての物質（構造）を、可能な限り実物に近い形でインストールしたものでなければならない。これによって、次の３つの特性が顕著になる。

1. 粒子と物質との多重散乱による誤差を正確に見積もることができる。

   これはクイック シミュレータの不足を補う意味で重要である。クイック シミュレータにも多重散乱の効果は入っているが前述の通り Helix パラメータに対しぼかしを入れる、といった手法なので、物質の違い（例えば中央飛跡検出器の内筒と内側のガスなど）によって局部的に多重散乱が大きくなったりする効果は見積もることができない。したがって、検出に関係のない部分も、物質のあるところは全てインストールすることが重要である。

2. 検出器の形状によって検出効率が落ちる部分が、検出器全体に及ぼす影響を見積もることができ、具体的な検出器の構造やパラメータについての検討が可能になる。

   例えば中央飛跡検出器の Cell における位置分解能の位置依存性などをあらかじめテスト実験で求めておき、この値をフルシミュレータに代入することによって、中央飛跡検出器全体でどのくらいの分解能が得られるかについての情報が得られるという意味である。重要なことは、この解は一通りではなく、Cell の配置によって変わり得るということである。巨大な検出器全体について、もっともよい運動量分解能が得られる Cell の配置を試行錯誤できる場は、フルシミュレータをおいて他にはない。

3. 実際の信号検出の手続きを踏むことができ、実際の信号とほとんど同じ形のシミュレーションアウトプットが得られるため、イベント再構成の解析アルゴリズム検証が可能になる。

イベントを再構成する際に混入する誤差を見積もり、イベント再構成のアルゴリズムを発展させる意味で非常に重要である。多重散乱による誤差が、物質と粒子の相互作用に起因する検出アルゴリズムに無関係の誤差であるとすれば、イベント再構成に於ける誤差は、検出アルゴリズムに直結した誤差である。したがって、これを完全になくすことはできないが、可能な限り減らす努力は行わなければならない。このシミュレータで培ったアルゴリズムは、そのまま実際の実験の解析プログラムに移植されることが可能となる。

第3章　次世代線型加速器用測定器シミュレータ

この節の概要

この章では測定器フルシミュレータのフレームワークである JUPITER について述べる。まず JUPITER について述べた後、JUPITER の主なベースクラスについて概説する。そして現在の JUPITER における問題点とその改善案を見ていく。

3.1　JUPITER とは

日本の ILC グループにおける測定器フルシミュレータはまず以下を目標として中央飛跡検出器グループにより開発されてきた。

1. 中央飛跡検出器の 3Tesla デザインにおける飛跡検出器のフルシミュレーションを行う。
2. 他の検出器グループが、自由にそれぞれの検出器部分をインストールできる仕様にする。
3. オブジェクト指向言語の特徴を生かし、部品を入れ替えたり、検出器のデザインパラメータを変更したりといった、検出器のデザインの変更が簡単に行えるようにする。(デザインパラメータチューニングのため)。
4. イベント再構成段階で混入する誤差を正確に見積もるため、最終的にはほぼ実験データ解析用のプログラムと同じ行程をシミュレートできる仕様にする。
5. 3T デザインが提案された原因ともなった、ビームラインからのバックグラウンドが検出器に与える影響を見積もるため、加速器の衝突点周辺の構造をもシミュレータに組み込み、バックグラウンド Study を加速器と統合して行えるようにする。

以上を踏まえて設計したシミュレータの開発当初のおおまかな構造と役割分担を
図 3.1 に簡単に示す。

主に検出器物質と粒子の反応部分のプログラムを軽くするため、Monte-Carlo Exact Hit(より一般的には Monte-Carlo Truth と呼ぶ) を出力するところで一度区切る仕様になっている。

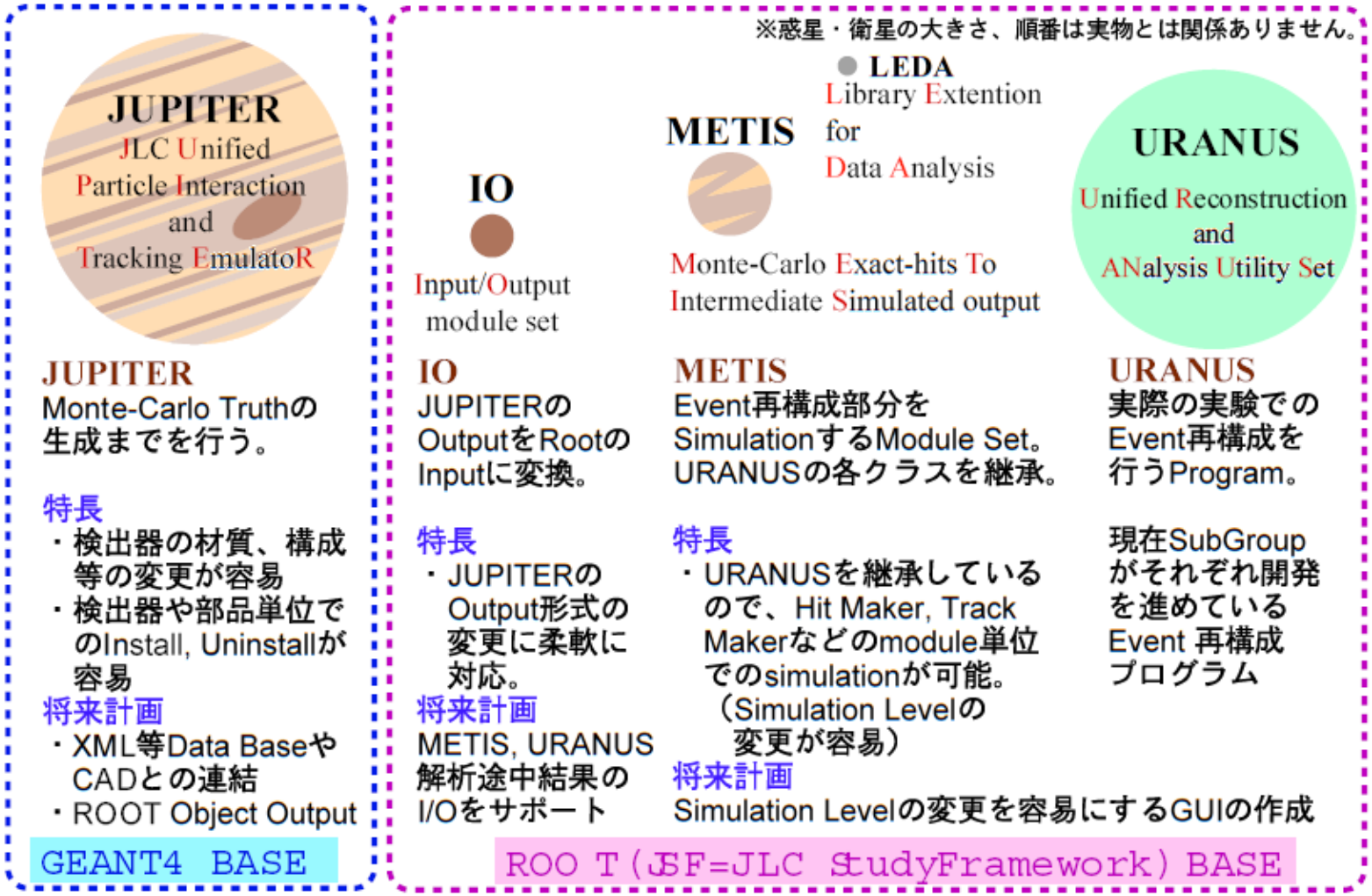

図 3.1: JUPITER、Sattelites、URANUS の役割分担

Monte-Carlo Truth を生成する部分を JUPITER (**J**LC **U**nified **P**article **I**nteraction and **T**racking Emulato**R**) と呼び、Geant4 ベースに開発されてきた。[1]

Geant4 とは前述の通り、近年、宇宙線分野、医学から地雷発見などの平和利用まで、広く使用されるプログラムに発展しており、Geant3 に比べ、より高エネルギー実験色の薄い汎用プログラムになった。従って、個々の事例に応用する際、特に ILC のような大型プロジェクトで使用する際には、よりユーザーが使用しやすいように、アプリケーションフレームワークを開発する必要がある。

主な高エネルギー実験プロジェクトのうち、Geant4 をもとに独自のアプリケーションを開発した例である (もともとあるフレームワークの一部として開発された例を含む)。

測定器（プロジェクト名)	アプリケーション名
ATLAS	Athena
BaBar	Bogus
Tesla	Mokka
NLC	JLDG4
JLC	JUPITER

JUPITER では、測定器パラメータがまだ完全に決定していないこと、今後のバックグラウンドの研究によっては、大幅なデザイン変更があり得ることなどから以下の点を考慮されて開発されてきた。

1. 複数の研究グループが同時に開発可能であること。
2. 可能な限り、実器のデザインに近付けるため、検出器のコーディングそのものが簡単であること。
3. 検出器デザインの変更が簡単であること。特定の測定器や、測定器の一部のインストール／アンインストール、置き換えを含む。

これらがどのように JUPITER 内で実現されているかを見ていく。

3.1.1 複数のサブグループによる同時開発

ILC には、バーテックス、Intermediate Tracker、中央飛跡検出器、カロリメータ、ミューオン粒子検出器の測定器開発グループがあり、さらにビーム衝突点グループ、ソレノイド磁石などの測定器以外の研究グループがある。

これらのサブグループが同時にシミュレータの開発を行えるようにするためには、これらのグループが担当する部分を完全に独立になるように分けてしまうのが良い（各検出器のモジュール化)。

各サブグループには、それぞれのディレクトリ（ワークスペース）を割り当て、自分のディレクトリの中のファイル以外の変更は、原則として行えないようにした。これにより、

[1]開発当初の線型加速器計画は JLC であった

各サブグループは、いつでも好きなときに自分の担当する検出器のアップグレードを行うことが出来る。アップグレードは、ディレクトリごと新しいものに置き換える。シミュレータの管理者は、kern（kernel の略。意味するところは木星の核である）ディレクトリの管理を行い、各サブグループに物質を置いてよいスペースを配分する。kern ディレクトリには、Geant4 を走らせる為のクラス（Event , Run, Physics に関係するクラス）、視覚化のためのクラス、メインプログラムなどが置かれており、この kern クラスだけで、実験室がひとつ置かれているだけのミニマムセットを作成し、イベントを走らせることができる。

3.1.2 プログラムコーディングの簡易さ

一般に、このようなシミュレータを開発するのは、実験家であってプログラマーではない（既に実験開始が決まっている大型プロジェクトを除く）。実験家は多忙であり、時間もコストもかかるビーム実験をいかにして効率よく行うかを知る為にシミュレータを作成し、シミュレーションを行うのであるから、シミュレーションを作成する場面で時間がかかってしまっては、その魅力は半減してしまう。まず第１に要求されることは、検出器の構造を細部まで作り込む作業の回数を可能な限り減らすことである。同じ構造を持つものを組み立てるのはただ１度でよく、あとはその形をもつオブジェクトを大量に量産すればよい。これはまさしく、ユーザー定義型を提供する C++のもっとも得意とするところであり、この特性をいかに生かすかが、アプリケーション構築のキーポイントになる。第２には、マニュアルを熟読しなくても、そこそこのコーディングが行えるだけのプログラミングコンセプトを提供することである。勿論、細かいシミュレーションを行うにはマニュアルは不可欠であるが、時間がない実験家が、単純な検出器部品の構造をコーディングするのに、分厚いマニュアルを読破しなければならないようでは使いにくいことこの上ない。Geant4 は大変優れたプログラムであるが、汎用に開発されているが故に、非常に細かい機能に分けてクラスが定義されており、ひとつひとつを眺めてみてもそれが構成する具体的な物体、概念等（オブジェクトと呼ぶ）が見えにくい。そこで、JUPITER では、プログラミングそのものが、なるべく実際の実験における検出器の組立、インストールなどと同じ感覚で行えることを目標とした。実際の検出器部品をそのままオブジェクトとみなしてコーディングすれば、あとは実際の実験で行う作業を表す関数を呼べばよいように設計された。

3.1.3 検出器デザインの柔軟性

Geant4 には、検出器のジオメトリを記述するための様々な道具が揃っている。前章で述べたように、フルシミュレータには、検出に関係ない大変複雑な形をした部分も詳細に入れることが必要になるため、このような部分は、一般に、CAD 等のアプリケーションを用いて記述したものを Geant4 のオブジェクトに変換する形がとられる。しかし、検出器の構造は、パラメータチューニング（グローバル、ローカルの両方を含む）のために頻繁に変更される。１レイヤーに含まれるセルの数を変更する度、内部のセンスワイヤーの配置情報も含めて、いちいち CAD で書き直していたのでは、大変な時間の無駄である。

また、検出器を構成する部品は、一般に円柱や箱形で表現できるような単純な形をしたものが多く、必ずしも CAD の助けを借りる必要はない。更に、ILC では、Intermediate Tracker のようにまだ具体的な内部構造が決まっていない検出器も存在する。このような場合に、まず Intermediate Tracker の部分に物質の層をいれておき、具体的なデザインが設定された時点で、層の内部に詳細な構造を作り込めることが重要である。これらのことが簡単に出来るためには、検出器の部品が適切な階層構造を持ち、様々な命令（関数）が階層の終端まで再帰的に呼ばれて、検出器全体に伝わるような形を取るのが好ましい。

3.1.4 ベースクラス構造

JUPITER のベースクラスは、大きく分けて、次の 3 つの区分に分けられる。

1. 検出器のジオメトリカルな部品（Detector Component）が共通して持つべき特性を集約したベースクラス（J4VComponent、J4VXXXDetectorComponent）
2. 検出器の反応部分（Sensitive Detector）に関する共通特性を集めたクラスと、Monte-Carlo Exact Hit を扱うベースクラス（J4VSensitiveDetector、J4VSD、J4VHit）
3. 物質の定義と管理を行うクラス (J4VMaterialStore、J4XXXMaterialStore）

これらのクラス同士の UML 図[2] を、図 3.2 に簡単に示す。に示す。なお、頭に J4 がついているのが JUPITER のクラス、G4 がついているものは Geant4 のデフォルトのクラスである。

[2]UML とは，Unified Modeling Language の略で，統一モデリング言語を意味している。オブジェクト指向で業務を分析したり，システムを開発したり，ソフトウェアモジュールを設計したりする際のダイアグラムの描き方・記法を定めた米国のオブジェクト技術標準化団体 OMG(Object Management Group) の標準であるオブジェクト指向分析・設計の標準表記法と言える．Rational Software 社の Grady Booch 氏、James Rumbaugh 氏、Ivar Jacobson 氏の 3 人によって開発された。従来、オブジェクト指向設計の表記法は 50 以上の規格が乱立していたが、1997 年 11 月に OMG によって UML が標準として認定された。Microsoft 社や IBM 社、Oracle 社、Unisys 社などの大手企業が支持を表明している。

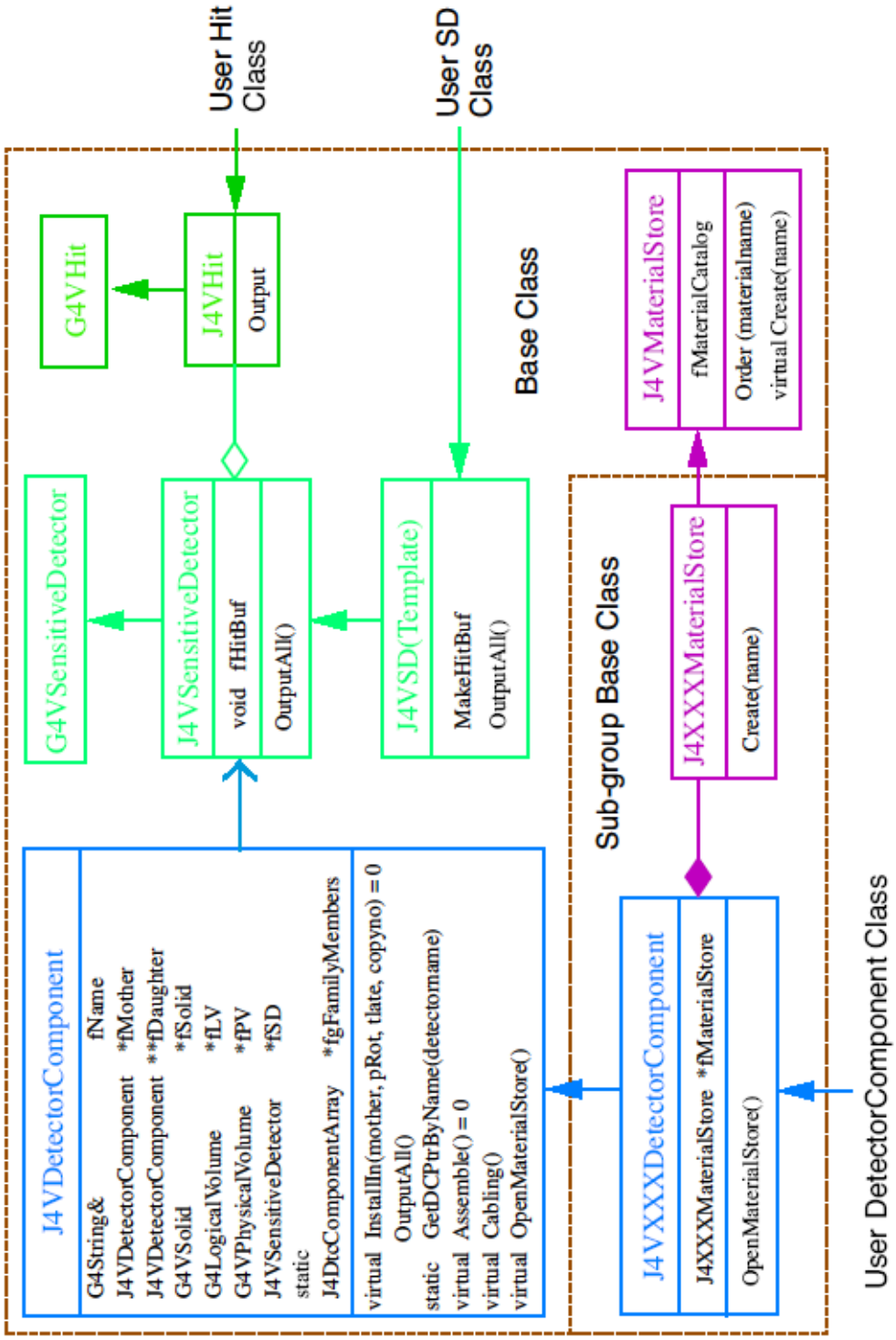

図 3.2: JUPITER のベースクラス

J4VComponent

J4VComponent は、全ての検出器部品（広くは加速器部分の部品も含まれる）の親となるベースクラスである。J4VComponent クラスは、検出器を構成するそれぞれの部品（コンポーネント）のひな型の原形として働くが、そのひな型は固定された形、材質のコンポーネントを量産できるだけではなく、少しずつ大きさの違うコンポーネントを生成したり、材質の違うコンポーネントを生成したりすることもできる。J4VComponent クラスは、Geant4 の用語でいう PhysicalVolume[3]1 つに対し、１つのオブジェクト[4] を作るよう設計されている。これは、Geant4 で測定器のジオメトリを定義する際に必ず必要な 3 つのオブジェクト（Solid、LogicalVolume、PhysicalVolume）のうち、もっとも実際の測定器部品に近い特徴を備えているからである。[5] 以下に、J4VComponent クラスのデータメンバと、代表的な関数（機能）について述べる。

データメンバと自動ネーミング機能

名前は、個々のコンポーネントを特定するために大変重要であるが、それゆえに重複した名前をつけると大変なことになる。したがって、JUPITER では全てのコンポーネントの名前は J4VComponent クラスが重複しないようにつける仕様になっている。名前を正しくつけるためには、コンポーネントを生成するひな形である DetectorComponent クラスが、自分自身からいくつのオブジェクトが作られるかを知っていなければならない。また、これらの情報をひな形が所持することによって、円筒の分割によって部品を作ったりする場合に作業を簡略化出来る（例：１レイヤー中に N 個詰められたセルの幅を自動計算させるなど)。そこで、これらの値をデータメンバに加える。更に、プログラムの操作上検出器部品に付加しておいた方がよいと考えられるものを挙げる。ただし、ここでの数とは、同じ階層に属するコンポーネントの数（同じ親コンポーネント中にインストールされる兄弟コンポーネントの数）を示す。

- 自分自身がインストールされる親コンポーネント fMother
- 同じクラスで作られる形等の違うコンポーネントの数 fNbrother
- 形等の違ったコンポーネントのうちの何番目か fMyID
- 同じクラスで作られる同じ形のコンポーネントの数 fNClones
- 同形のコンポーネントのうち何番目か fCopyNo
- 測定器全体に含まれるコンポーネントのリスト fgFamiryMembers

[3]PV の説明

[4]オブジェクト指向のオブジェクトだが、ここでは測定器の個々の部品を表すコンポーネントだとしてよい。

[5]Solid は検出器部品の形のみを定義する。LogicalVolume は、Solid に材質、感応領域であるか否か、絵を描かせたときに表示するか否か、などの情報を合わせたものである。PhysicalVolume は、LogicalVolume にそれが置かれる位置、回転角度などの配置情報を合わせたものである。

まず、形や材質の違うコンポーネントを考える。例としては、たとえば1 セルの中に含まれる５本のワイヤーに割り当てられたDriftRegion などが挙げられる。これらは、Sense ワイヤーを内包し、同じ原理でトラックの通過位置を検出するものであるから、同じひな形から作られるべきであるが、セルは扇形をしているので、５つのDriftRegion は微妙にドリフト領域の大きさが違う（仮に兄弟コンポーネントと呼ぶ)。これらの大きさの変化をパラメトライズするためには、全体でいくつの兄弟コンポーネントがあるかを表す変数と、自分自身がそのうちの何番目かを記憶させる変数とを、コンポーネントのひな形が持っていることが望ましい。このことによって、大きさなどのパラメトライズの部分を完全にひな形クラスの中に閉じ込めることができる。

次に、まったく同じ形、同じ材質のクローンコンポーネントを、配置する場合を考える。単純にコピーを並べる場合には、上と同じ議論でクローンの数と自分が何番目かを表す数があればよい。クローンの数を指定するのは、測定器の部品には全体の何分割かで一つの形が決まるものが多いからである。更に、特別な場合として、ある単純な形を平面で等分割し、すきまなく並べる場合を考える。この場合には、Geant4 側でレプリカ (G4Replica) というクラスが用意されており、これを用いることで若干プログラム実行時のメモリを減らすことができる。レプリカは、円筒、箱形などを与えられた分割数で等分割して配置するコンポーネントの作り方であるが、実体は分割された１つ分の PhysicalVolume が存在するだけであり、この一つの PhysicalVolume がトラックのやってくる場所に移動して、あたかもたくさんのクローンコンポーネントが存在するかのように働いている。したがって、コンポーネントのひな形が持つべき数は、レプリカに対してはクローンの総数だけでよい。更に、fCopyNo に-1 を入れることによって、生成されたコンポーネントがレプリカによって配置されるものであることを示す。[6]

最後の fgFamilyMembers は、この J4VComponent クラスから作られたオブジェクト全てに対し１つしかない配列であり、J4VComponent から継承して作られた全ての検出器コンポーネント（オブジェクト）へのポインタを記憶している。つまり、J4VComponent クラスは、実験室にどんな名前の検出器コンポーネントが存在するかを知っているのである。そこで、名前から逆に個々の検出器コンポーネントへのポインタを辿る GetDtcComponentByName 関数を用意し、検出器のヒットを書き出すか否かをコンポーネント単位で設定したいとき等に使えるようにした。類似の機能は Geant4 にも存在するが、この関数の利点は戻り値のポインタが J4VComponent 型のポインタであり、J4VComponent にインプリメントされている便利な関数が自在に使える点にある。

Assemble 関数

Assemble 関数は、名前の通り、検出器コンポーネントの組み立てを行う関数である。Geant4 の用語でいえば、LogicalVolume をセットするところまでを行う。検出器コンポーネントは、その内部にまだ内部構造があれば、その内部構造を作り込んだところで形が完

[6]Geant4 では、(コピーを含め) 単純に親オブジェクトの中に娘オブジェクトを配置する場合 G4PVPlacement を使うが、Replica と PVPlacement を同じ親オブジェクトの中に同時に配置することを禁じている。したがって、fNclones に入る数は、コピーによるクローンの総数か、レプリカによるクローンの総数かのいずれかである。

成するのであるから、内部にインストールされるコンポーネントを作成[7] するのは、基本的にはこの Assemble 関数の中である。

検出器コンポーネントの組立は、実験ではインストールよりも随分前に行うことが多いが、かならずしもこの点においてまで実験を模倣する必要はなく、インストールの直前で組み立てが仕上がっていればよい。

したがって、基本的に Assemble 関数は後に述べる InstallIn 関数の中で呼ばれるべきものであり、外部に公開する必要のないものである。

そこで、Assemble 関数はプライベート関数に設定した。これにより、検出器コンポーネントのジオメトリ・材質等についての完全な情報隠蔽が行われ、あるコンポーネントの内部の構造変更がその他のプログラムに影響を及ぼさない仕様になっている。

InstallIn 関数

InstallIn 関数は、自分自身を親コンポーネントにインストールするための関数である。引数に親コンポーネントへのポインタをもらい、主に、その中にどのような形態でインストールされるのかを記述する。[8] 引数に位置や回転をとることも出来るので、親コンポーネントの Assemble 関数の中で、位置を指定して娘コンポーネントの InstallIn 関数を読んでやれば、親コンポーネントの望みの位置にインストールすることが可能である。この関数は親コンポーネントから呼ばれるため、当然パブリック関数である。実際のコーディングの一例を、図に示す。

J4VXXXDetectorComponent クラス

表題の XXX には、サブグループ名が入る。ユーザーは、各サブグループに一つ、J4VComponent クラスを公開継承してサブグループ独自のベースクラスを作り、全ての検出器コンポーネントは、このサブグループ専用のベースクラスを公開継承しなくてはならない。J4VXXXDetectorComponent クラスの役目は３つある。一つは、名前の中にサブグループ名を入れることである。これにより、もしコンポーネントに他の検出器グループと重複する名前（例えばレイヤーなど）を指定しても、全体では違った名前がつけられることになる。あとの二つは、各サブグループに一つあればよいクラスを開くという役目である。クラスの中には、複数存在すると混乱を来すものがある。代表的なものが定義を行うクラスであり、検出器パラメータや検出器の材質を定義するクラスがこれに相当する。前者を ParameterList クラスとし、後者を別に MaterialStore クラスとする。これは、後者がコンポーネントの物質を合成するためのマテリアル工場を持っており、単なるパラメータリスト以上の仕事を行うためである。これらのクラスについては、後に言及する。勿論、各サブグループ独自に付け加えたい機能があれば、このクラスの中で定義することによって、同じサブグループに属する全ての検出器コンポーネントでその機能が使えるようになる。

---

[7] C++に関して言えば、new 演算子で娘オブジェクトを生成すること。

[8] 現材の JUPITER においては、PVPlacement を呼ぶか、Replica を呼ぶかの二者択一である。Geant4 にはもう一つ PVParametrized というオブジェクトの置き方があるが、これは内部構造をもつオブジェクトではうまく作動しない上、実行メモリを気にしなければ PVPlacement で代用出来るので、JUPITER ではサポートしていない。

J4VSensitiveDetector、J4VSD、J4VHit

これらの３つのクラスは互いに連係して働くので、3 つで一つの機能を提供すると考えてさし支えない。Geant4 では、測定器の感応部分（Sensitive Region）に対し、G4VSensitiveDetector の指定を行う。SensitiveDetector は、LogicalVolume に対して定義され、LogicalVolume の範囲内に粒子が飛び込んで来たときに、何らかのヒット情報を生成し、このヒット情報を G4VHit クラスのオブジェクトに詰めた上で、バッファに記録する役目を負う。バッファに記録されたヒット情報は、イベントの終わりに出力される。このとき、書出し命令を行うのは G4EventAction クラスであり、このクラスはシミュレータ全体の動作に関わるクラスなので、当然サブグループユーザの触れない kern ディレクトリの中におさめられている。ここで、G4EventAction にユーザがアクセスしなくてもヒットの書出しを行えるようにしたのが、この３つのクラスからなる仕組みである。まず、G4EventAction の中からは、J4VComponent クラスの OutputAll 関数が呼ばれる。この関数は、全ての検出器コンポーネントを再帰的にスキャンし、SensitiveDetector に指定されたコンポーネントがあれば、そのコンポーネントの SensitiveDetector の OutputAll 関数を呼び出す。SensitiveDetector の OutputAll 関数は、自分が持っているバッファからヒットオブジェクトを一つずつ取り出し、そのヒットオブジェクトの Output 関数を呼ぶ。こうして、ヒットオブジェクトに詰め込まれたヒット情報がアウトプットされる。この仕組みがうまく働くためには、全てのサブグループの SensitiveDetector は J4VSD クラスを公開継承して作られねばならない。また、Hit クラスも同様に、J4VHit を公開継承して作られる必要がある。

J4VMaterialStore、J4XXXMaterialStore

これらは、検出器の物質を定義し、提供するためのクラスである。J4XXXMaterialStore の XXX の部分には、サブグループ名が入る。Geant4 では、検出器の物質を自分で定義できる仕様になっている。ガスの混合比を変えたり、特殊合金を作ったりといったことも可能である。検出器のデザインを試行錯誤する段階で、物質を構成する材料の混合比を少しずつ変えてみたり、あるいは温度や圧力を変えてみたりすることは十分あり得る話であるが、その度に管理者に頼んで望みの物質を作ってもらうのは面倒である。そこで、管理者は基本的な材料カタログを提供するのみとし、各サブグループではカタログにない材料を自分で合成出来る仕様にした。J4VMaterialStore クラスは、仮想関数で Order 関数と Create 関数を持つ。これを継承して J4XXXMaterialStore クラスを作り、Create 関数のみ物質を定義して実装すると、材料の Order を受けたときにまず J4VMaterialStore に行ってカタログの中を探し、そこになければ自分の Create 関数の中を見て物質を合成する、といった作業を行わせることができる。このしくみは、全体で統一して使用されるべき物質は管理者の管理とし、サブグループに対し、他の検出器部分に影響を与えるような勝手な物質定義を許さない、という意味でも重要である。

3.1.5 メインプログラム

さて、各サブグループのユーザーが手を加えることのできるファイルは基本的に各サブグループのディレクトリのみだが、この他に、唯一若干の変更が許されているファイルがある。それがメインプログラム Jupiter.cc である。メインプログラムでは、どの検出器をインストールするか、どの検出器のスイッチをオンにするか（ヒットの書出しを行うか)、どのようなイベントを発生させるか等を操作することができる。特に Hit の書出しについては、前述の GetDtcComponentByName 関数を用いて、名前から特定のコンポーネントへのポインタを得、ダイレクトにヒットの書出しをオンにしたりオフにしたりすることができる。これは、検出器の一部分が故障して信号を出力できなくなった場合のシミュレーション等を可能にする 18。ちなみに Geant4 では、プログラムを走らせている間ならば、同様の作業が可能な仕様になっている。このメインプログラムは、その機能をソースコードの段階で簡単に行えるようにしたものである。

第4章　JUPITERにXMLインターフェースを追加する。

この章の概説

まず最初に現行の JUPITER の問題点を見て行く。それからなぜ XML インターフェースを付け加えるに至ったのか、そして、その実装を見て行く。

4.1　現在の JUPITER の問題点

JUPITER のフレームワークでサブグループにおいて測定器を作る際には自分のディレクトリ内で JUPITER の提供するベースクラス (J4VComponent) を継承し Assemble 関数と InstallIn 関数を呼ばねばならない構造になっている。また、一度測定器構造を作ってしまうと測定器コンポーネントのクラスが強い結びつきを持ち変更がし難い。つまり自分の階層構造を知っていなければ何もできなくなるというような構造になってしまっている。現段階の ILC 測定器シミュレーショングループにおいての最も重要な命題は検出器パラメータの最適化であり、サブグループディレクトリのアップデートを頻繁にする必要が生じる。クラス間が強い結びつきをもっているため少量のアップデートの度に大量のコードを再コンパイルと再リンクを繰り返すことになる。JUPITER は J4VComponent を提供することでユーザにユーザに測定器の持つべき情報のセットを与えることができた。しかしユーザのクラス群は大きく違うことをやっているわけではない。つまり

1. 各測定器のパラメータを集めてハードコーディングされたクラスから各測定器のパラメータをとってきて Geant4 の形状をつくり物質を設定し視覚化条件を決める。(Assemble 関数)

2. 各測定器を自分の親ボリュームとなる測定器の中に配置する。(InstallIn 関数)

3. 各測定器がセンシティブ (hit 情報を残すか否か) であるかを決める。(Cabling 関数)

をそれぞれのクラスで繰り返しているだけである。それならテンプレートにすればよいかというと、そういうわけでもなく、それぞれの測定器のパラメータは測定器ごとに異なるのである。このパラメータをとってくる構造は JUPITER では各サブグループディレクトリで Singleton のパラメータ管理クラスを用意してとってくる構造になっている。このパラメータはソースコードの値で管理してあり、データをプログラミングしている現状である。この点はやはりメインテナンスの面から考えても改善すべきである。データをとってくる構造を各測定器クラスで抽象化できるのであれば各測定器の生成は自動化できるよ

うになるであろう。これをするには測定器の構造をまとめたデータモデルをつくるのがよい。さらに測定器のインストール・アンインストールに関する問題であるがこれは構造に関してもデータにしてしまえばそのデータをとってくるだけで異なるシミュレーションが行えるようになるので便利である。

したがって JUPITER の改善案としては

- 測定器の情報をハードコードされたデータでないものにし、そのデータによって J4VComponent 継承クラスをつくる。
- さらに、データにより簡単な構造の作成を自動化させる

ということになる。

ところで JUPITER は現在既に動いているため改善の結果は徐々に置き換えていくようにせねばならない。データとして改善できるような場所は積極的に改善し、とても複雑に絡み合いデータとして表現する意図を見出せないところはプログラムとして開発可能な自由度を残さなければならない。

4.2 データフォーマットとして何を選択すればよいか

データフォーマットの候補は多々あるが以下を考慮せねばならない。

現段階の ILC 測定器シミュレーショングループにおいての最も重要な命題として検出器パラメータの最適化を挙げたが、これはシミュレーションに使うデータをアップデートしやすいということである。つまりデータとして、アップデートのしやすさを表現せねばならない。アップデートのしやすさとは

- 単純な形式である。つまり厳格な規則があるということである。
- 編集のしやすいプレインテキスト形式である。
- 移植性がある。ユーザのプラットフォームに依存してはならない。
- カスタマイズ可能。ユーザが新たなデータ形式を加えれる拡張性がある。
- 可読性に優れる。

等が挙げられる。

また、測定器の構造を構築するのであるから、測定器間の構造化も表現できるデータフォーマットであるのが好ましい。

さらに、世界の趨勢に従い便利なライブラリが整っているものが望ましい。

これらの条件満たすフォーマットとしては XML が適切であろう。XML[11](e**X**tensible **M**arkup **L**anguage) は現在ポータビリティに優れたデータフォーマットとして web 上を中心とし、世界中で活躍しているフォーマットである。上に挙げた条件を以下のように満たしていることがわかる。

- マークアップ形式なのでデータは < と > で囲まれたタグで表現でき単純な形式である。

- データが妥当かどうかをチェックするスキーマが存在し、規則性にも優れている。
- 単なるプレインテキストであるので移植性がよい。
- ユーザ型を定義できる W3C XML Schema という W3C 勧告の技術によりカスタマイズ性かつ再利用性に優れている。
- ブラウザで表示させたり、様々なツールもそろっていることから可読性にも優れている。
- 構造化を表すのは XML の最も得意とするところである。

また、Apache のプロジェクトにより、xerces-c/xalan-c というオープンソースでフリーの XML の構文解析ライブラリが存在する。
以上により、XML をデータフォーマットとして選択した。

4.3 JUPITER の測定器データモデリング

では測定器に関する属性を XML データに置き換えていこう。測定器に関して持つべき属性は J4VComponent クラスによりまとめられているのでそこから考えていくことにする。測定器データのもつべき情報は

- (solid, material, colour, visibility, sd)

としてよいだろう。配置は後で考えるほうがデータとしての再利用性は高くなると考えられるのでここには含めないことにする。これらはすべて論理ボリュームがもつべき情報である。

論理ボリューム

この中で測定器シミュレーションの要でもある測定器の形状を決める部分はユーザに対して拡張性をもたせたい。そのためには solid というデータを抽象的にする必要がある。幸いなことに、W3C XML Schema ではそれを行うための方法として「代替グループ」というオブジェクト指向の抽象クラスに似た概念が存在するのでそれを用いる。ユーザはこの代替グループに属することによって自分の定義型をつけ加えることができる。これを実現するための W3C XML Schema の例を以下に示す。

```
<!--
空のデータタイプ。ユーザはこれを継承する。
-->
<xs:complexType name="solidmodelType"/>

<!--
抽象的なデータの定義
-->
<xs:element abstract="true" name="solidSubstitutionGroup" type="solidmodelType"/>

<!--
ユーザ定義型。
G4 ではサポートされていないエンドキャップつきの円筒を作るためのものである。
これは J4VComponent でサポートされている。
このようにすれば形状データ型として置き換えることができる。
-->
<xs:element name="tubswithendcaps" substitutionGroup="solidSubstitutionGroup">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="solidmodelType">
        <xs:attribute name="rmin"                type="idrefstringdoubleType"/>
        <xs:attribute name="rmax"                type="idrefstringdoubleType"/>
        <xs:attribute name="halfzlength"         type="idrefstringdoubleType"/>
        <xs:attribute name="totalphi"            type="idrefstringdoubleType"/>
        <xs:attribute name="endcaphalfthickness" type="idrefstringdoubleType"/>
        <xs:attribute name="endcaprmin"          type="idrefstringdoubleType"/>
        <xs:attribute name="sphi"                type="idrefstringdoubleType"/>
        <xs:attribute name="leftrmin"            type="idrefstringdoubleType"/>
        <xs:attribute name="rightrmin"           type="idrefstringdoubleType"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<!--
solid に関するデータ型の定義
代替グループを持つ。
したがってここには代替グループに設定されている element が入ることができる。
-->
<xs:complexType name="solid">
```

```
  <xs:sequence>
    <xs:element ref="solidSubstitutionGroup"/>
  </xs:sequence>
</xs:complexType>
```

JUPITERの特性としては、Brotherという概念で各階層の測定器を管理しており同じ階層に属する測定器は同じ名前を元に管理されている。前章で説明したとおり、ユーザは自分でsolidの数などを指定して分けなければならない。ユーザにとって一々コンストラクタを変えるのは面倒だし一貫性がない。ユーザは意識しなくて使えるとうれしいだろう。しかもこれらはLogical Volumeのメモリサイズを削減する概念であるからLogical Volumeを表すデータのスキーマは次のようにした。

これを以下に示す。

```
  <!-- lvGroupは論理ボリュームのデータグループをまとめる -->
  <xs:group name="lvGroup">
    <xs:choice>
      <!-- 各xs:groupは(solid, material, colour, visibility, sd)の情報をもつものである -->
      <xs:group ref="onlyoneoriginGroup"/>       <!-- 一個だけのLVに対応 -->
      <xs:group ref="cloneGroup"/>               <!-- LVをクローンでつくる場合に対応 -->
      <xs:group ref="brotherGroup"/>             <!-- 名前だけ同じもので管理 -->
      <xs:group ref="samematerialbrotherGroup"/> <!-- materialは同じものを使う -->
      <xs:group ref="samesolidbrotherGroup"/>    <!-- solidは同じものを使う -->
    </xs:choice>
  </xs:group>
```

これらをもとに考えるとJ4VComponentの継承クラスがもつべき論理的なXMLデータのスキーマは次のような形式をもつ。

```
<xs:group name="componentGroup">
  <xs:sequence>
    <xs:group ref="lvGroup"/>
  </xs:sequence>
</xs:group>
```

配置

XMLによりジオメトリを構築することについてpvplacemntで中心位置に置く[1]ことを例に考える。JUPITERではAssembleを再帰的に呼ぶことにより自分の娘ボリュームをインストールしている。

[1]JUPITERでは測定器のほとんどはworldの中心と同じ中心位置に置かれる。

例えばAという測定器がB_0, B_1という測定器を自分の娘ボリュームとして持っていたとする。される場合があるとする。このAをworldに3個インストールしたいならworld直下でAを3つつくる。このときA_0に対しB_0, B_1にがインストールされ残り2つのA_1, A_2に対してもB_0, B_1をもつことになる。
これを表すと以下のように考えられる。

```
  <world>
    <pvplacement ref="A">
<!-- デフォルトで中心におかれるとした -->
      <pvplacement ref="B"/>
      <pvplacement ref="B"/>
    </pvplacement>
    <pvplacement ref="A"/>
      <pvplacement ref="B"/>
      <pvplacement ref="B"/>
    </pvplacement>
    <pvplacement ref="A"/>
      <pvplacement ref="B"/>
      <pvplacement ref="B"/>
    </pvplacement>
  </world>
```

3つくらいのデータならなんとかなるかもしれないが大量に置くときは不便であるし、クラスの構造をインストールアンインストールしやすいように分離させる目的が達せていない。簡単にこれを解消する方法としてmotherをランダムアクセス可能にする。

```
  <world>
    <pvplacement ref="A"/>
    <pvplacement ref="A"/>
    <pvplacement ref="A"/>
  </world>

  <detectormother ref="A">
    <pvplacement ref="B"/>
    <pvplacement ref="B"/>
  </detectormother>
```

これで分離することができたが、Aのbrotherたちにたいしてすべて B_0, B_1をインストールする必要がある。またJUPITERではひとつ置くのとその他を置くのを明示的にわけているのだからそれらを”origin”と”link”という概念にし、分けることにする。よって以下のようになる。

```
  <world>
```

```
    <placeorigin>
      <pvplacement ref="A"/>
    </placeorigin>
    <placelink>
      <pvplacement ref="A"/>
    </placelink>
    <placelink>
      <pvplacement ref="A"/>
    </placelink>
  </world>

  <detectormother ref="A">
    <placeorigin>
      <pvplacement ref="B"/>
    </placeorigin>
    <placelink>
      <pvplacement ref="B"/>
    </placelink>
  </detectormother>
```

この link は 2 個目以上は lvGroup を決定する際にもうすでにきめられているものである。従って、同じ位置に置くなどの際には二個目以上はユーザはきにしなくていいようにしたい。よって以下のようにした。

```
  <world>
    <placeorigin>
      <pvplacement ref="A"/>
    </placeorigin>
    <placelinks>
      <pvplacement ref="A"/>
    </placelinks>
  </world>

  <detectormother ref="A">
    <placeorigin>
      <pvplacement ref="B"/>
    </placeorigin>
    <placelink>
      <pvplacement ref="B"/>
    </placelink>
  </detectormother>
```

placelink は明示的に名前を指定して置く場合、placelinks は自動的に判断しておく場合と決めた。

以下これらを処理するクラスを説明する。

4.4 XML インターフェースを実装するためのクラス群

4.5 ジオメトリをつくるソフトウェアの構成

geant4 ジオメトリの構築を以下の 3 ステップに分けた。

1. XML のイベントをメモリ内に読み込み木構造を組み立てる。
2. XML のイベントの木構造を渡り歩き J4VComponent クラスの木構造をつくる。
3. J4VComponent の木構造から Geant4 ジオメトリをつくる。

各ステップには Facade を Singleton で用意しアクセスできるようにした。以下各コンポーネントについて詳述する。

4.6 XML のイベントの木構造をメモリ内に読み込む部分

4.6.1 SAX2EventTreeDirector

まず XML を解釈するパーサを作らねばならない。パーサのビルドをカプセル化するために Builder パターンを用いた。これを以下のクラス図 4.1 に示す。

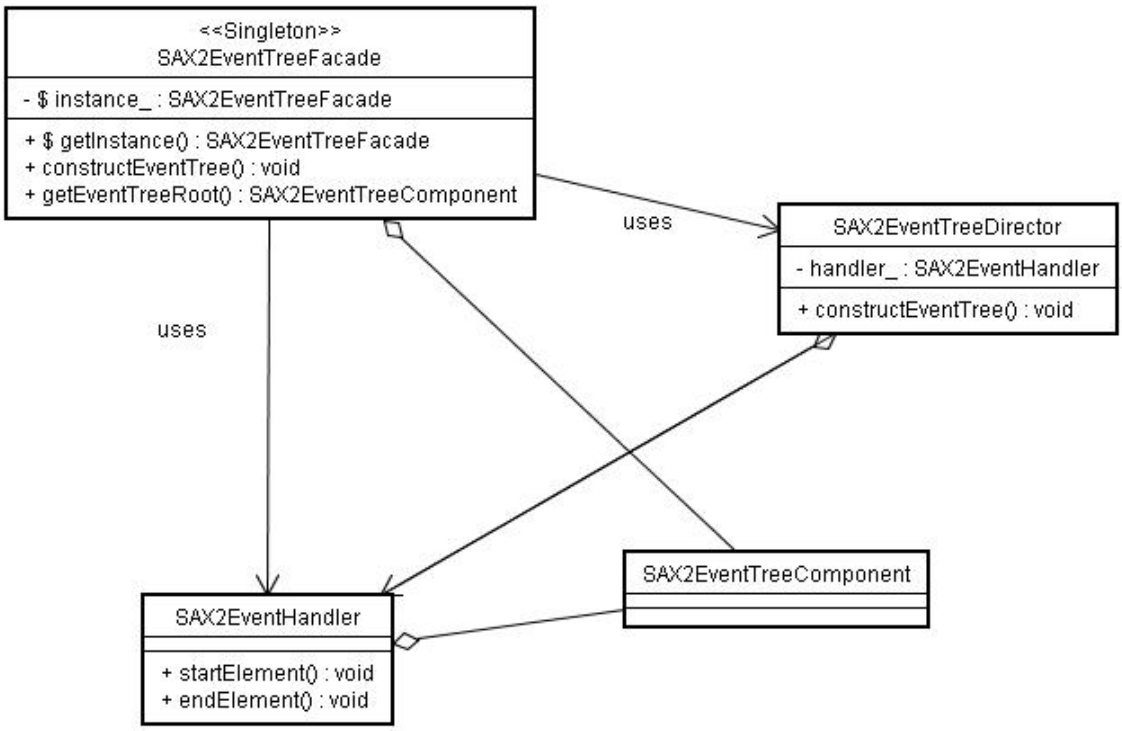

図 4.1: XML の木構造の解釈をスタートする

SAX2EventTreeHandler

ここで用いるパーサは上記の選択により SAX2 である。SAX2 は XML イベント駆動型なのでイベントに対するハンドラが必要である。これは SAX2 のインターフェースである DefaultHandler を継承して実装する。これを以下のクラス図 4.2 に示す。

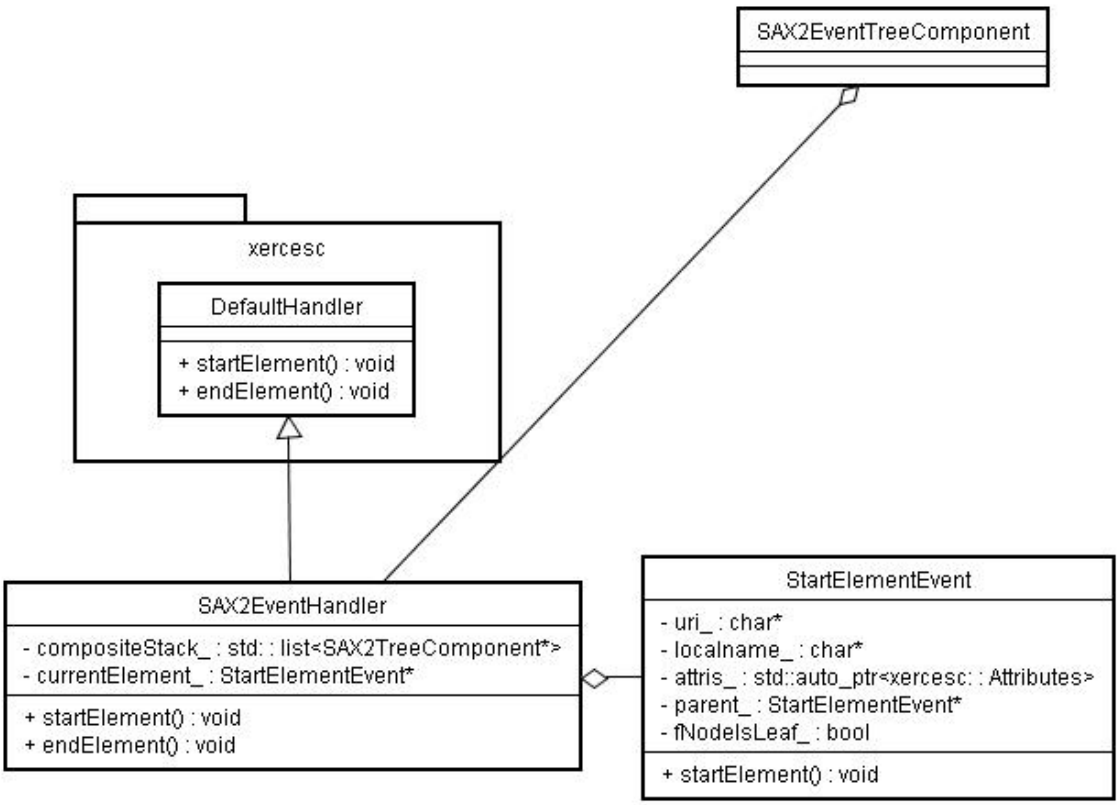

図 4.2: SAX2 により XML を解釈する

4.6.2 SAX2EventTreeComponent

SAX2 の各ハンドラのコールバックはイベントの解釈の結果をスタックに取るのでそれらはイベントコールバック関数内でしか生存することができない。従ってこれらを後に解釈する場合にはメモリ内に保存しておく必要がある。このジオメトリ構築においてはイベントのそれぞれが意味を持ちそのイベントから Geant4 へと結合するために、イベントの解釈を遅延する方法をとっている。したがってこの XML 木構造を表すクラスが必要となる。ここで解釈されたイベントは SAX2 の XMLCh*よりも std::string に読み直しておいたほうが便利であろう。

さらに、これらの木構造は解釈を遅延されるのでクラスの外から訪問を受けるよう設計が便利であるので木構造を表すには Composite パターン、外からの階層構造の訪問を表すのに Visitor パターンを用いた。Visitor パターンを用いる際にはコンパイルとリンクの複雑さから階層構造の安定性が求められる。この XML の木構造を表す Composite の Component 継承クラスは XML が子要素を持てば Composite パターンの Composite、子要素を持たないのであれば Composite パターンの Leaf であり、この二つしか存在しないといえる。したがって継承のクラス階層は安定しており、Visitor パターンを用いる条件を満たしている。[2]

これを以下のクラス図 4.3 に示す。

4.6.3 SAX2EventTreeFacade

これで XML の木構造を文字列としてメモリ内に読み込めたことになるがそれらのアクセスに便利なクラスも欲しい。この木構造への入り口、つまり Facade を Singleton として用意した。これを以下のクラス図 4.4 に示す。

[2]C++で Visitor パターンの実装にはオープンソースの Loki テンプレートライブラリが便利であるのでこれを用いた。

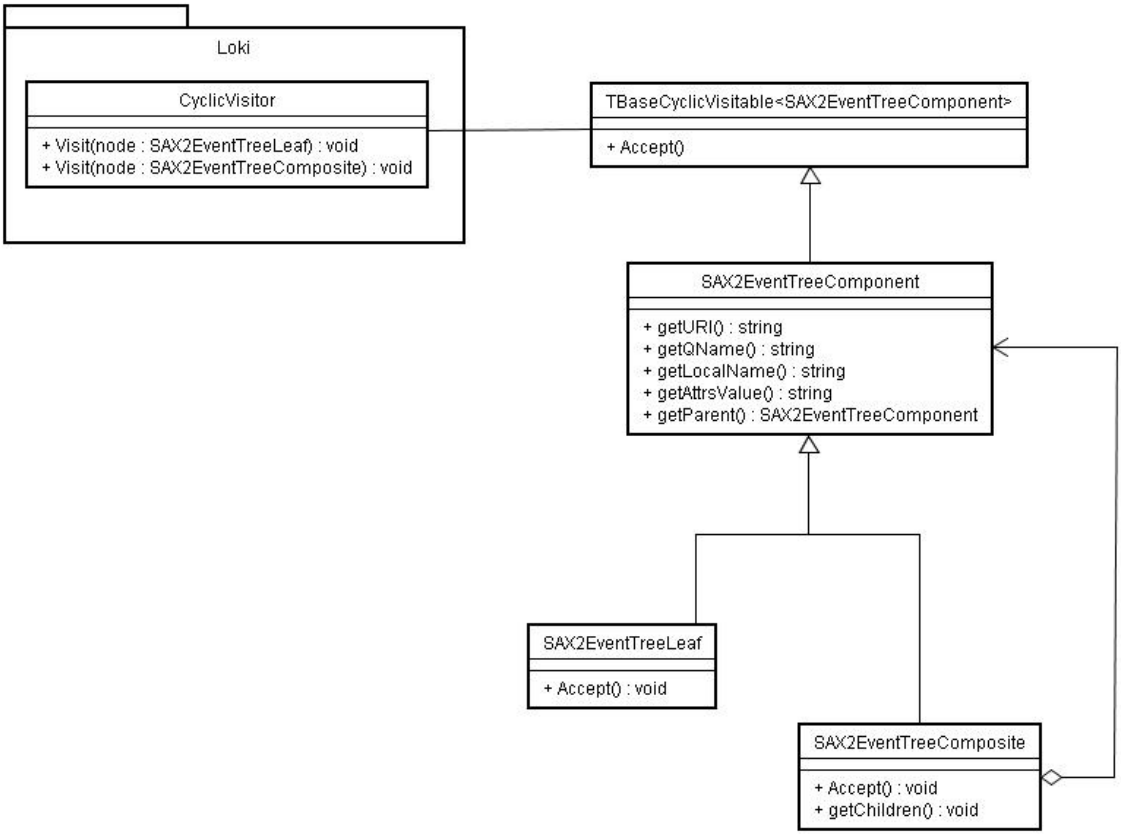

図 4.3: XML の木構造を表し外からの訪問を受け付ける。

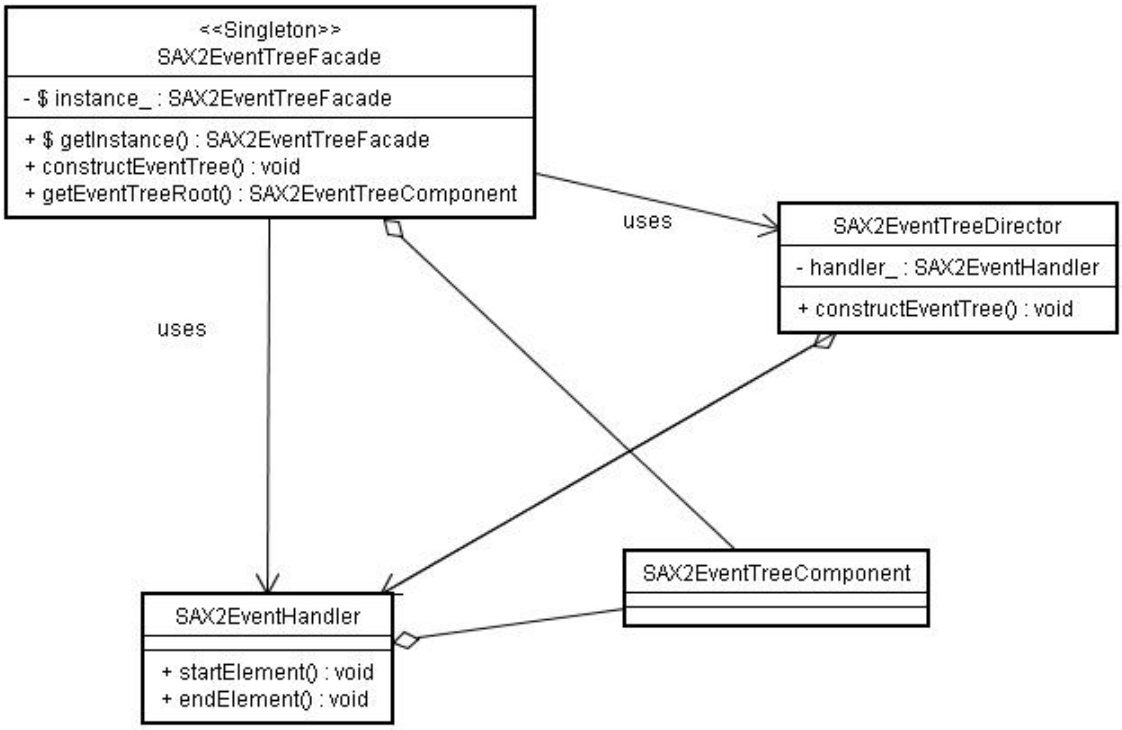

図 4.4: 解釈された XML の木構造への入り口

4.7 XMLのイベントの木構造訪問してJ4VComponent等価クラス木構造を生成する

ここから文字列に焼きなおされたXMLの木構造を訪問する部分が始まる。

Geant4に情報を渡すために、上記のデータモデリングで述べた通り物質テーブルの初期化とインストレーションにステップを分けている。ここではそれが物質テーブル初期化に関する部分と、インストレーションに関する部分のXMLの木構造の訪問として表現される。

4.7.1 BuildingPublisherとBuildingMediator

まず各XMLイベント独自のJUPITERに対する適切な処理をするクラスが必要となる。さらにこれらはすでに読み込まれてある木構造のXMLイベントとマッピングされなければならない。

従って、木構造の訪問の通知に対し適切な処理をマッピングする構造を考えなければならない。これには通知と購読を表すObserverパターンを用いる。この通知には間に一段階おいてマッピングされたXMLイベントを表すクラスを呼び出すようにしなければならない。間に介在して適切なクラスを呼び出す処理をするためにMediatorパターンを用いた。

この構造を以下のクラス図4.5に示す。

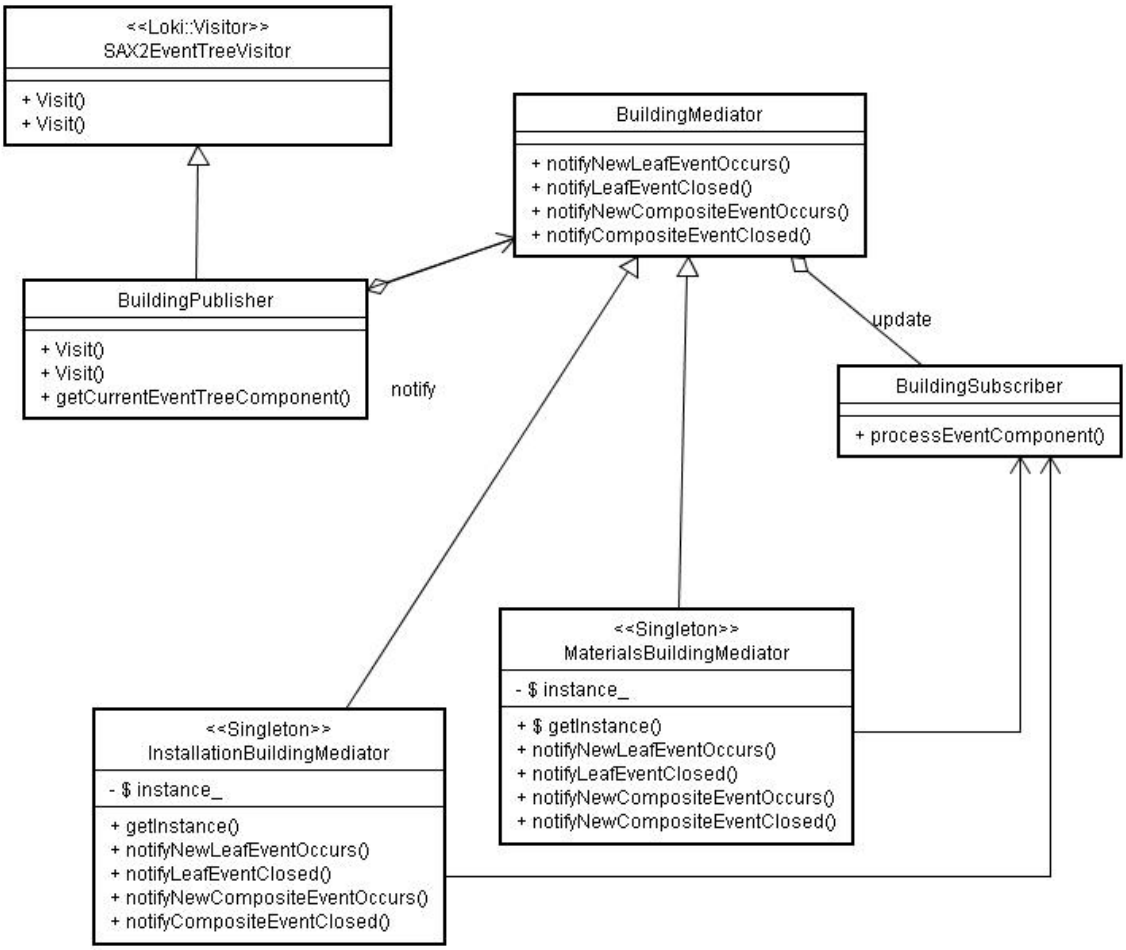

図 4.5: XML イベントに対して適切な処理をするクラスへと通知する

4.7.2 MaterialsBuildingMediator

物質テーブルに関する処理をするクラス群のマッピング。BuildingMediator のインターフェースを継承し適切な Subscriber を呼び出す。

この構造を以下のクラス図 4.6 に示す。

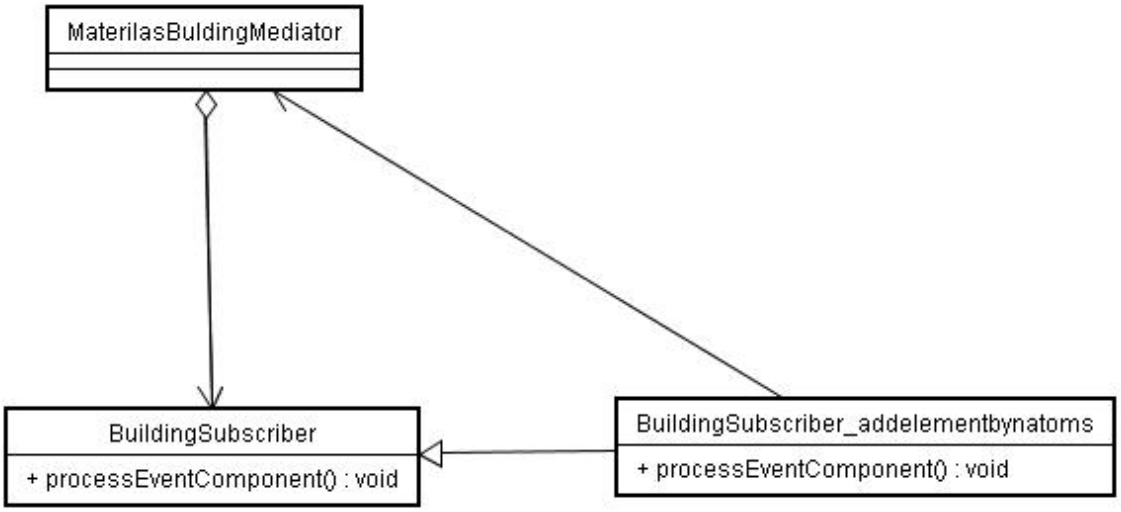

図 4.6: 物質テーブルに関する XML イベントに対して適切な処理をするクラスへと通知する

4.7.3 InstallationBuildingMediator

測定器のインストレーションに関する処理をするクラス群のマッピング。BuildingMediator のインターフェースを継承し適切な Subscriber を呼び出す。

この構造を以下のクラス図 4.7 に示す。

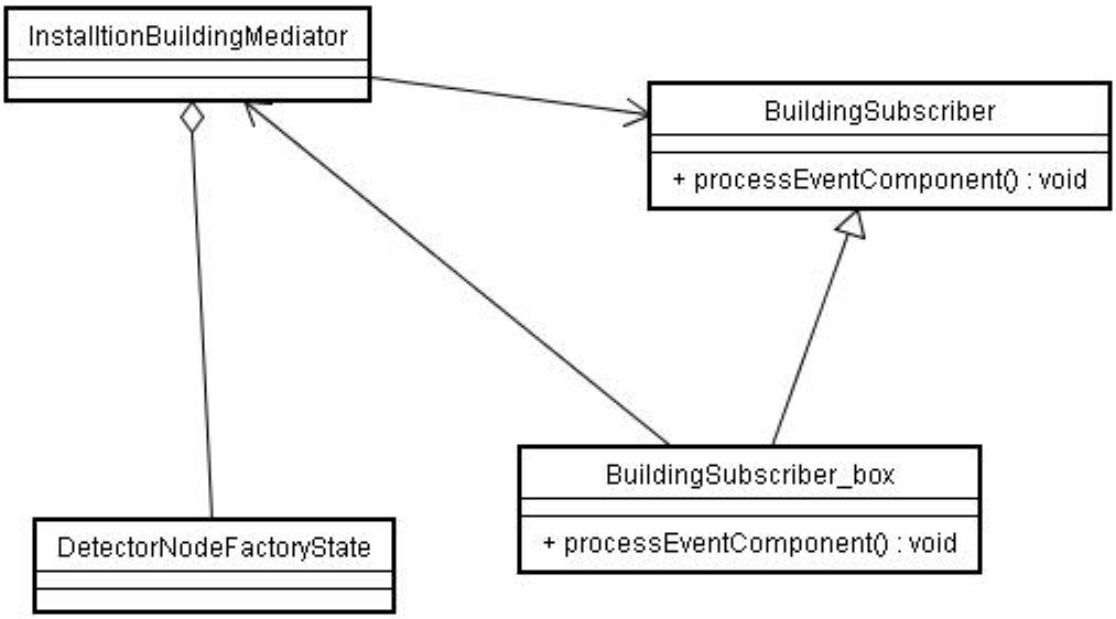

図 4.7: 測定器のインストールに関する XML イベントに対して適切な処理をするクラスへと通知する

4.7.4 LVComponentState

インストールのまず最初。Geant4 の LogicalVolume（LV）をつくる部分を始める。上記のデータモデリングで述べたとおり、solid の数や material の数に対してそれぞれ異なるやり方で LogicalVolume をつくらなければならない。これらは <subgroupcomponent> 内にどれだけ子要素があるかで処理が異なる。つまり </subgroupcomponent> の通知をもって初めて何をすべきかを知る。よって </subgroupcomponent> の通知まで処理を遅延させねばならずその間、子要素の数により状態を変える。

その状態を表すクラス群として State パターンを用い、Singleton で実装する。この構造を以下のクラス図 4.8 に示す。

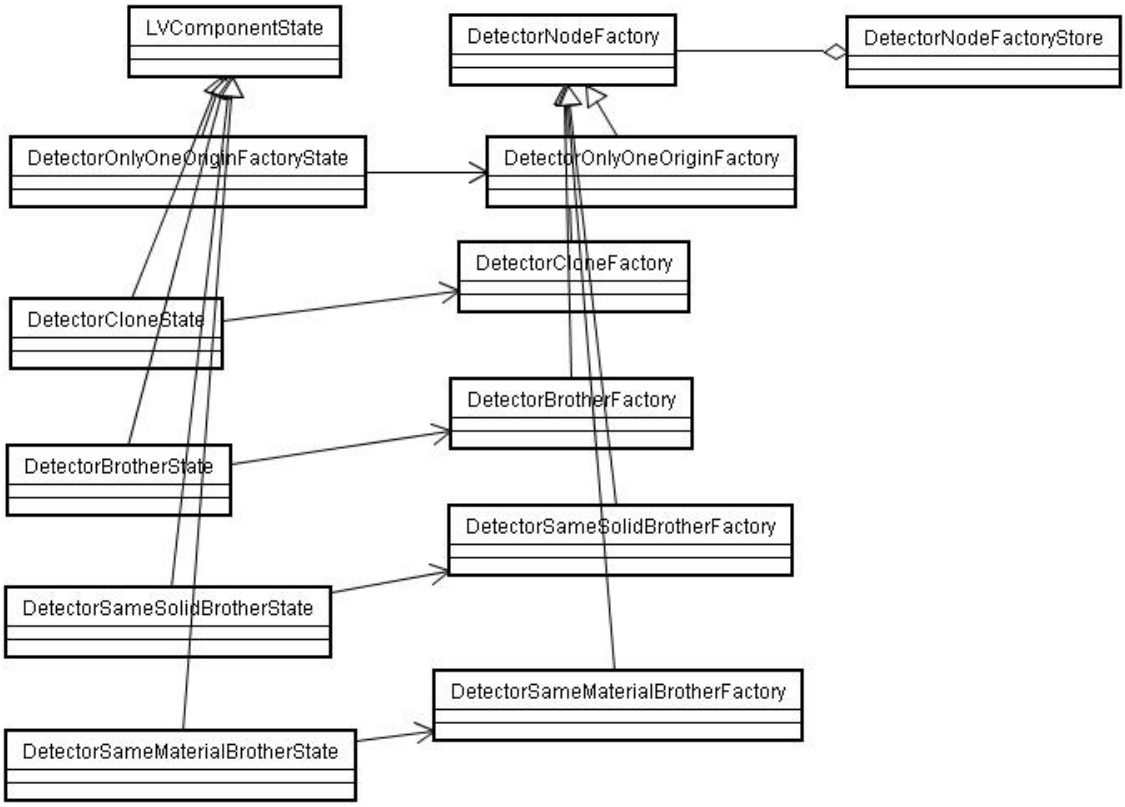

図 4.8: <subgroupcomponent> から </subgroupcomponent> までの状態遷移を表すクラス群

4.7.5 DetectorNodeFactory

上記の State たちは </subgroupcomponent> に到達するとその状態に対クラス群を作り出す Factory をビルドし、後でこれらの LogicalVolume は PhysicalVolume の構築に必要とされるのでメモリ内に保存する。

この構造を以下のクラス図 4.9 に示す。

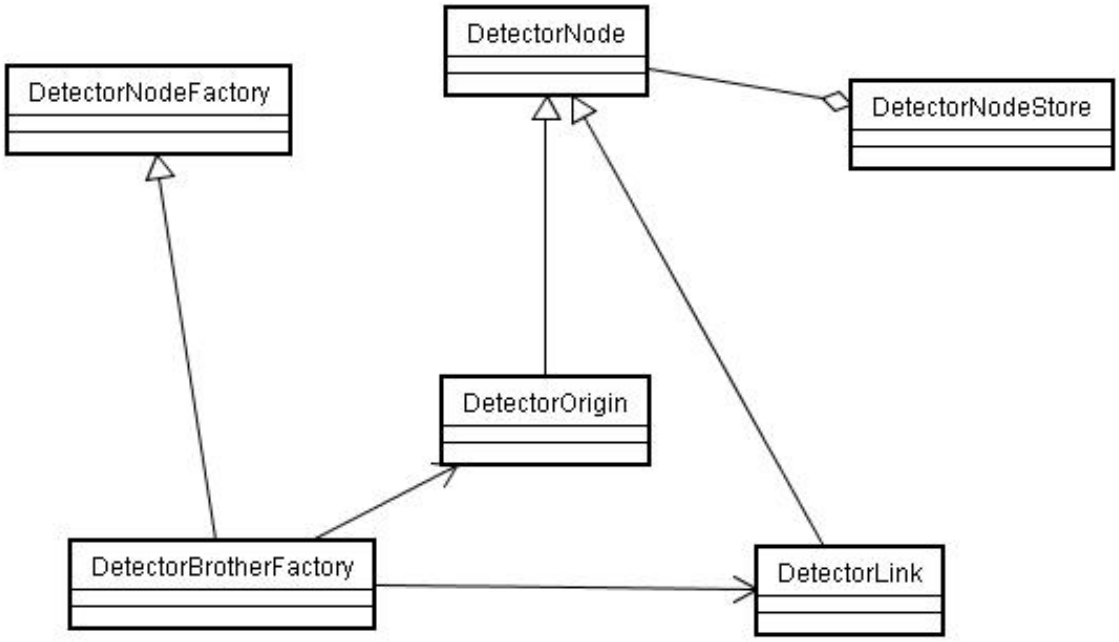

図 4.9: DetectorNode の Factory たち

4.7.6 DetectorNode

測定器を実際に配置する段階で論理ボリュームから実体が必要になる。上記のデータモデリングで述べたとおり、JUPITER で置きかたの違いを各種フラグで分けていたものをクラスとして実装した。それぞれに違う処理をするのでクラスとして分けるべきである。この DetectorNode が JUPITER で XML によるジオメトリ生成させる際の中心的クラスとなる。

この構造を以下のクラス図 4.10 に示す。

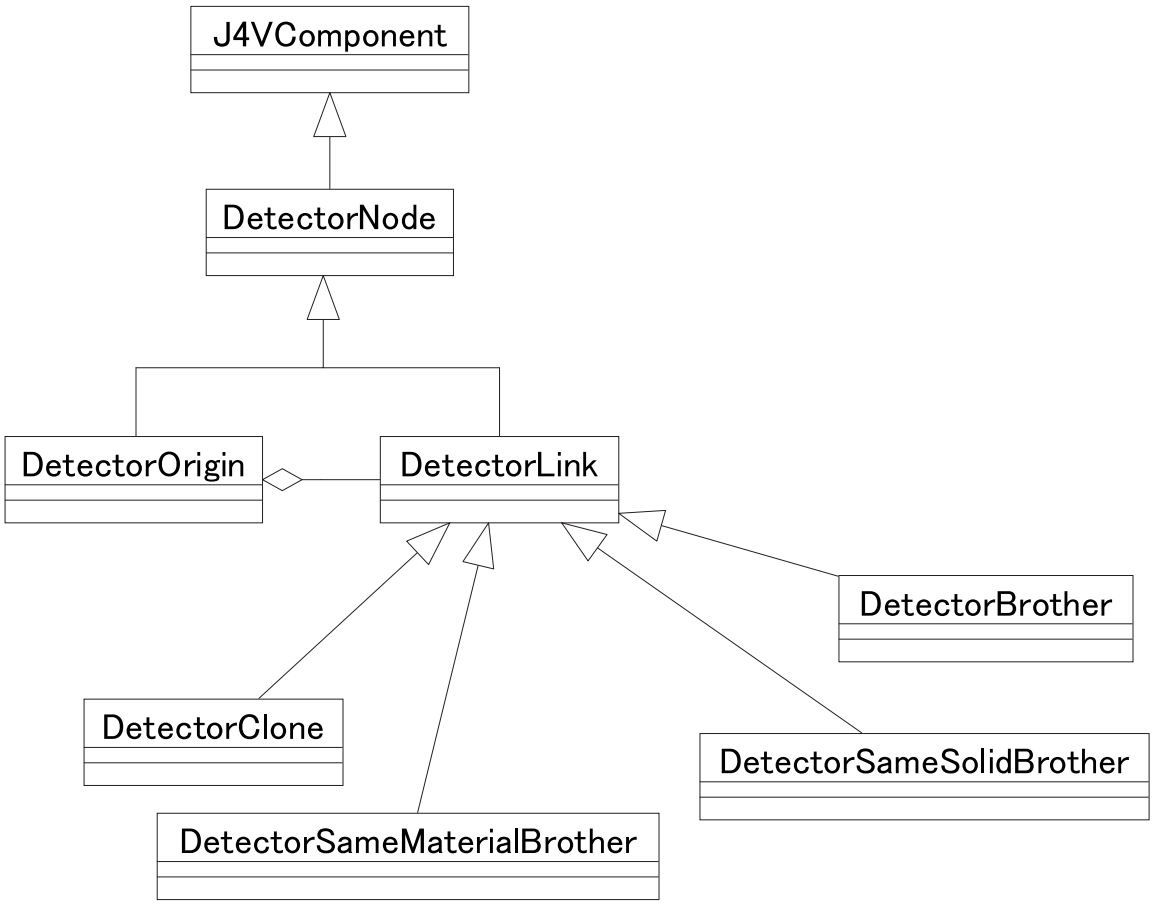

図 4.10: DetectorNode

以上で J4VComponent のクラス木構造をつくることができた。

4.8 J4VComponent 等価クラス木構造を訪問して Geant4 ジオメトリを構築する

前のステップまでで Geant4 のジオメトリをつくる準備ができたわけであるが、ここまでは G4Material や G4Element を除いて一切 Geant4 のコンポーネントを用いていないことに注意されたい。Geant4 のジオメトリの生成に関して、例えば ILC 測定器を作るような場合は巨大なリソースが必要となるので Geant4 のコンポーネントを全く使わず軽いプロキシークラス群を用いてジオメトリ構築の一歩手前で状態を一端静止させるのは妥当な判断であろう。

ここまでであれば XML に戻して書き出すこともできるし編集することもできる。また ROOT の TGeometry などを用いて Geometry を確かめることも可能である。[3]

そして本当に Geant4 のジオメトリを作るすべての準備が整ったと判断できたらいよいよ J4VComponent 等価クラス木構造を訪問して Geant4 ジオメトリを構築するフェーズへと突入する。

これは JUPITER のフレームワークで行うわけだから Assemble と InstallIn で行う必要がある。

つまり DetectorNode の継承クラスが各々に設定された娘ボリュームをみてその数だけ DetectorNodeFactory を呼べばよいということにした。DetectorNodeFactory は XML の情報から自分がどの Brother かを知っている。したがって doMakeOrigin とイテレータにより doMakeLink を行うと適切な数だけ娘ボリュームをつくれるようにした。

[3] これらのインターフェースを設計するのは今後の課題である。

第5章　例題としてILCのジオメトリを構築する

前章までで説明したXMLデータを用いる方法でILCのジオメトリの一部を構築する。例題として中央飛跡検出器を選び説明する。

5.1　XMLデータに置き換える部分

前章までで説明した通り、XMLデータに置き換えるのは物質定義、定数、単純な検出器インストールである。以下の節でそれらを順に説明する。

5.2　物質定義

現在のJUPITERではJ4VMaterialStoreを継承したクラスのCreateメソッド内で全体で提供されていない各グループ独自の物質を「プログラミング」していた。そして名前により検索する構造になっている。以下その例を示す。

```
//*****************************************************************

#include "J4TPCMaterialStore.hh"
#include "G4Element.hh"
#include "G4Material.hh"

G4Material* J4TPCMaterialStore::Create(const G4String &name,
                                       G4MaterialPropertiesTable *)
{

  G4Material* material= 0;

  if (name == "InShellC") {
    G4double A, Z, density;
    G4String name;

    A= 12.011 *g/mole;
    density = 0.1317 * (4.215 - 0.056) / 4.215 *g/cm3;
```

```
  G4Material *InShellC = new G4Material(name="InShellC", Z=6., A, density);

  material = InShellC;
}

if (name == "OutShellC") {
  G4double A, Z, density;
  G4String name;

  A= 12.011 *g/mole;
  density = 0.08815 *g/cm3;
  G4Material *OutShellC = new G4Material(name="OutShellC", Z=6., A, density);

  material = OutShellC;
}

if (name == "Copper") {
  G4double A, Z, density;
  G4String name;

  A = 63.546 *g/mole;
  density = 8.92 *g/cm3;
  G4Material *Copper = new G4Material(name="Copper", Z=29., A, density);

  material = Copper;
}


if (name == "EndCu") {
  G4double A, Z, density;
  G4String name;

  A = 63.546 *g/mole;
  density = 7.716 *g/cm3;
  G4Material *EndCu = new G4Material(name="EndCu", Z=29., A, density);

  material = EndCu;
}


if(name == "P10" || name == "TDR") {
```

```
//--------------
// elements...
//--------------
G4double A, Z;
G4String name, symbol;

A= 1.00794 *g/mole;
G4Element* elH= new G4Element(name="Hydrogen", symbol="H", Z=1., A);

A= 12.011 *g/mole;
G4Element* elC= new G4Element(name="Carbon", symbol="C", Z=6., A);


//--------------
// materials...
//--------------
G4double density, massfraction;
G4int natoms, nel, ncomponents;

//temperature of experimental hall is controlled at 20 degree.
const G4double expTemp= STP_Temperature+20.*kelvin;

// Methane gas (CH4)
const G4double denMethane= 0.717e-3 *g/cm3 * STP_Temperature/expTemp;
G4Material* Methane= new G4Material(name="Methane", denMethane,
                                    ncomponents=2,
                                    kStateGas, expTemp);
Methane-> AddElement(elC, natoms=1);
Methane-> AddElement(elH, natoms=4);

if (name == "TDR") {
  //element O for TDR
  A= 15.9994 *g/mole;
  G4Element* elO= new G4Element(name="Oxygen", symbol="O", Z=8., A);

  // CO2 gas
  const G4double denCO2= 1.977e-3 *g/cm3 * STP_Temperature/expTemp;
  G4Material* CO2= new G4Material(name="CO2Gas", denCO2, ncomponents=2,
                                  kStateGas, expTemp);
  CO2-> AddElement(elC, natoms=1);
```

```
      CO2-> AddElement(elO, natoms=2);

      // Ar(93%) + Methane(5%) + CO2(2%) mixture
      density = denMethane*0.93 + denAr*0.05 + denCO2*0.02;
      G4Material* TDR = new G4Material(name="TDR", density, nel=2,
                                       kStateGas, expTemp);
      TDR-> AddMaterial(Ar     , massfraction= 90. * perCent);
      TDR-> AddMaterial(Methane, massfraction= 10. * perCent);

      material = TDR;

    } else {
       // Ar(90%) + Methane(10%) mixture
       density = denMethane*0.9 + denAr*0.1;
       G4Material* P10= new G4Material(name="P10", density, nel=2,
                                       kStateGas, expTemp);
       P10-> AddMaterial(Ar     , massfraction= 90. * perCent);
       P10-> AddMaterial(Methane, massfraction= 10. * perCent);

       material = P10;
    }
  }

  return material;

}
```

このソースコードは前章までで説明したやり方を用いて以下のようにXMLデータに置き換えることができる。これらはプレインテキストのデータであってC++ソースコードではないことに再度注意されたい。XML属性の文字列はJ4kern::Evaluatorクラス（CLHEPのHepTool::Evaluatorのラッパークラス）を用いて解釈するのでデータ内で数式や文字列の定数が使えることにも注意して欲しい。単位などはCLHEPで使えるもの、すなわちGeant4内で使えるものを使っている。また各データに対してスキーマ内で定義されているデフォルトの値もあるのでデータの一部を省くことができる。

またこれはサブグループ内独自の設定である。J4kern::Evaluatorクラスや各コンスタントテーブル管理クラスに渡す定数はすべて接頭辞に”TPC::”という文字列で管理されることになる。つまり以下のデータで参照している文字列はまず”TPC::”という接頭辞を前に付け加えられてから検索されることになる。もしも”TPC::”という名前で見つからない場合に”TPC::”をはずした名前で探索される。

```
<?xml version="1.0" encoding="utf-8"?>

<!-- TPC Materials -->

<subgroupmaterials xmlns:jupiter="http://lcdev.kek.jp/jupiter"
                   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                   xsi:noNamespaceSchemaLocation="../../schemas/jupiter.xsd"
                   name="TPC"/>

  <!-- define constants -->
  <constant name="densityAr" value=""/>
  <constant name="densityMethane" value=""/>
  <constant name="densityCO2" value=""/>

  <!-- elements -->
  <!-- element O for TDR -->
  <buildelementdirectly name="Oxygen" symbol="O" zeff="8" aeff="15.9994 *g/mole"/>

  <!-- materials -->
  <buildmaterialfromscratch name="InShellC"
                            z="6"
                            a="12.011*g/mole"
                            density="0.1317 * (4.215 - 0.056) / 4.215 *g/cm3"/>

  <buildmaterialfromscratch name="OutShellC"
                            z="6"
                            a="12.011*g/mole"
                            density="0.08815*g/cm3"/>

  <buildmaterialfromscratch name="Copper"
                            z="29"
                            a="63.546*g/mole"
                            density="8.92*g/cm3"/>

  <buildmaterialfromscratch name="EndCu"
                            z="29"
                            a="63.546*g/mole"
                            density="7.716*g/cm3"/>

  <buildmaterialfromcombination name="Methane"
                                density="1.2635e-03*g/cm3"
```

```
                                ncomponents="2"
                                state="kStateGas"/>
    <addelementbynatoms ref="C" natoms="1"/>
    <addelementbynatoms ref="H" natoms="4"/>
  </buildmaterialfromcombination>

  <buildmaterialfromcombination name="CO2"
                                density="1.977e-3 *g/cm3 * STP_Temperature/expTemp"
                                ncomponents="2"
                                state="kStateGas"
                                temp="expTemp"/>
    <addelementbynatoms ref="C" natoms="1"/>
    <addelementbynatoms ref="TPC_O" natoms="2"/>
  </buildmaterialfromcombination>

  <!-- Ar(93%) + Methane(5%) + CO2(2%) mixture -->
  <buildmaterialfromcombination name="TDR"
                                density="densityMethane*0.93 + densityAr*0.05 + densi
                                ncomponents="2"
                                state="kStateGas"
                                temp="expTemp"/>
    <addmaterial ref="Ar"      massfraction="90." massfractionunit="perCent"/>
    <addmaterial ref="Methane" massfraction="10." massfractionunit="perCent"/>
  </buildmaterialfromcombination>
</subgroupmaterials>
```

これらはXMLデータなのでブラウザなどで表示することができる。
図5.1にブラウザで表示させた図を示す。

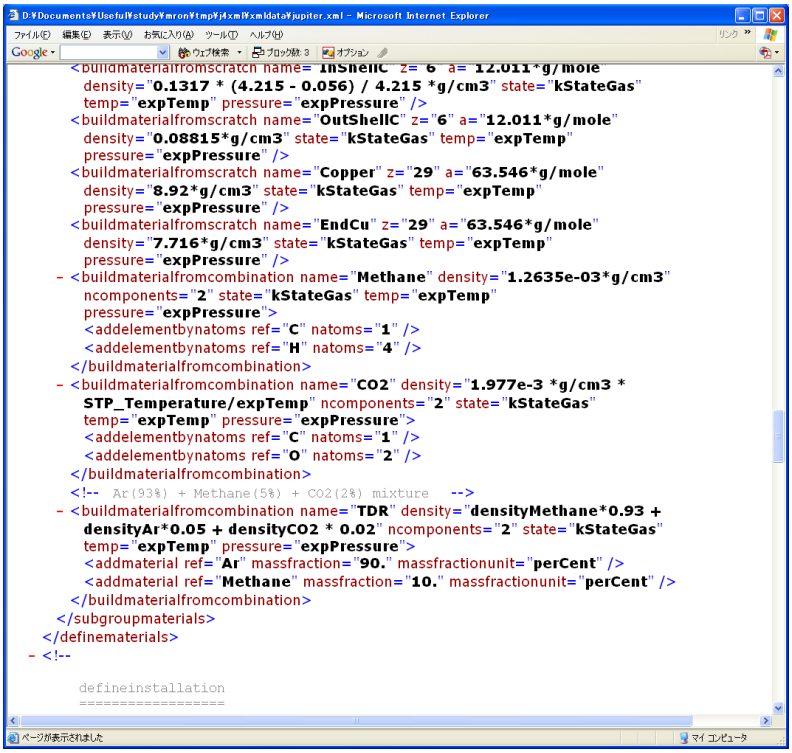

図 5.1: tpc_materials.xml をブラウザで表示

このようにして XML データとなった TPC の物質定義 JUPITER の XML 構成ファイルのエントリポイントである jupiter.xml 内で ENTITY 宣言し definematerials 内で&DEFINE_TPCMATERIALS; とし参照し JUPITER に組み込む。[1]

```
<!-- jupiter entity definition -->

<!DOCTYPE jupiter [
<!ENTITY DEFINE_LCEXP_MATERIALS SYSTEM "lcexp_materials.xml">
<!ENTITY DEFINE_TPC_MATERIALS   SYSTEM "tpc_materials.xml">
]>

<jupiter xmlns:jupiter="http://lcdev.kek.jp/jupiter"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="../schemas/jupiter.xsd">


<!--
  definematerials
  ===============
-->
```

[1] Java 版の xerces-J では実装されている XInclude[?] という技術が現在の C++版の xerces-c では実装されていない。従って xerces-c が XInclude に対応した際にはここを変える必要がある。

```
  <definematerials>
    &DEFINE_LCEXP_MATERIALS;
    &DEFINE_TPC_MATERIALS;
  </definematerials>

<!--
   defineinstallation
   ==================
-->
  <!-- installation data -->

</jupiter>
```

5.3 ジオメトリの構築

まず現在の JUPITER に入っている TPC ジオメトリコンポーネントのクラス図を図 5.2 に示す。

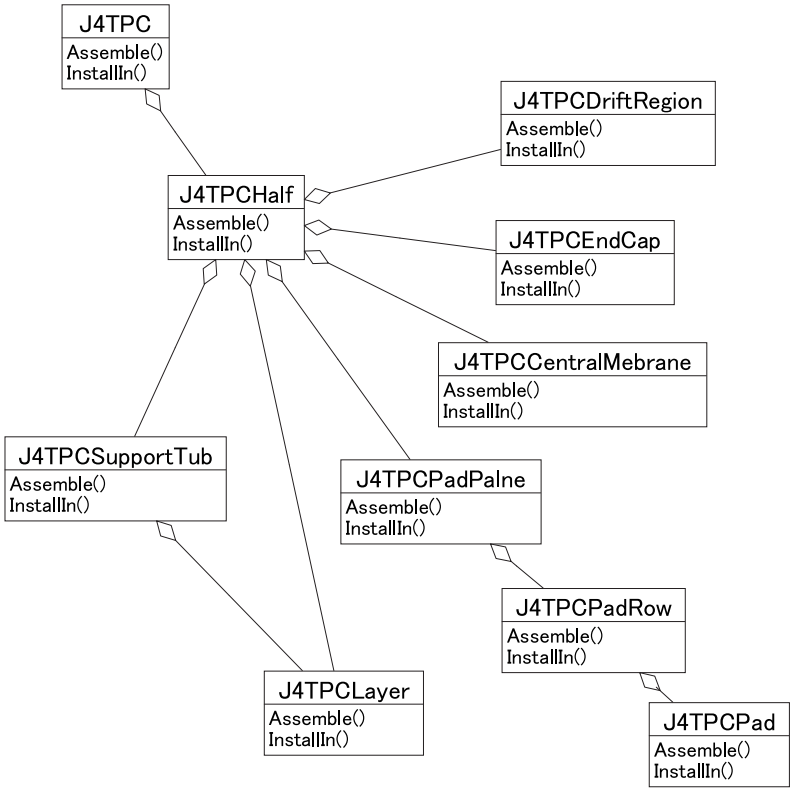

図 5.2: 現在の JUPITER に入っている TPC ジオメトリのクラス図

これらはすべてひとつずつクラスとして実装されている。

- J4TPC クラスは娘ボリュームとして J4TPCHalf をもつ。ふたつめは一つ目のクローンとしておく。
- J4TPCHalf は娘ボリュームとして J4TPCDriftRegion, J4TPCSuppotTub, J4TPCEndCap, J4TPCCentralMembrane, J4TPCPadPlane, J4TPCLayer を持つ。
- J4TPCDriftRegion は J4TPCLayer を娘ボリュームとして持つ。
- J4TPCPadPlane は J4TPCPadRow を娘ボリュームとして持つ。

以下これを XML データに置き換えていく。

5.3.1 論理ボリュームの作成

まず最初に必要となる J4VDetectorComponent 継承クラスのリストアップにあたる subgroupcomponent の定義をする。

```
<subgroupinstallation xmlns:jupiter="http://lcdev.kek.jp/jupiter"
                      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                      xsi:noNamespaceSchemaLocation="../../schemas/jupiter.xsd"
                      name="TPC"/>
   <!-- 定数の定義 -->
   <subgroupconstants>
   </subgroupconstants>

   <!-- コンポーネントの定義 -->
   <subgroupcomponent name="J4TPC">
      <solid>
         <tubswithendcaps/>
      </solid>
      <material ref="vacuum"/>
      <colour   ref="red"/>
      <visibility value="false"/>
   </subgroupcomponent>

   <subgroupcomponent name="J4TPCHalf">
      <solid>
         <tubswithendcaps/>
      </solid>
      <material ref="vacuum"/>
      <colour   ref="magenda"/>
      <visibility value="false"/>
   </subgroupcomponent>

   <subgroupcomponent name="J4TPCDriftRegion">
      <solid>
         <tubswithendcaps/>
      </solid>
      <material  ref="vacuum"/>
      <colourref ref="red"/>
      <visibility value="false"/>
      <sensitive class="J4TPCDriftRegionSD"/>
   </subgroupcomponent>

   <subgroupcomponent name="J4TPCSupportTub">
      <solid>
         <tubswithendcaps/>
      </solid>
```

```
   <material  ref="vacuum"/>
   <colourref ref="green"/>
   <visibility value="true"/>
</subgroupcomponent>

<subgroupcomponent name="J4TPCEndCap">
   <solid>
      <tubswithendcaps/>
   </solid>
   <material  ref="vacuum"/>
   <colourref ref="blue"/>
   <visibility value="true"/>
</subgroupcomponent>

<subgroupcomponent name="J4TPCCentralMembrane">
   <solid>
      <tubswithendcaps/>
   </solid>
   <material  ref="vacuum"/>
   <colourref ref="yellow"/>
   <visibility value="true"/>
</subgroupcomponent>

<subgroupcomponent name="J4TPCPadPlane">
   <solid>
      <tubswithendcaps/>
   </solid>
   <material  ref="vacuum"/>
   <colourref ref="white"/>
   <visibility value="false"/>
</subgroupcomponent>

<subgroupcomponent name="J4TPCLayer">
   <solid>
      <tubswithendcaps/>
   </solid>
   <material  ref="vacuum"/>
   <colourref ref="magenda"/>
   <visibility value="false"/>
   <sensitive class="J4TPCLayerSD"/>
</subgroupcomponent>
```

```
   <subgroupcomponent name="J4TPCPadRow">
      <solid>
         <tubswithendcaps/>
      </solid>
      <material  ref="vacuum"/>
      <colourref ref="white"/>
      <visibility value="false"/>
   </subgroupcomponent>

   <subgroupcomponent name="J4TPCPad">
      <solid>
         <tubswithendcaps/>
      </solid>
      <material   ref="vacuum"/>
      <colourref  ref="red"/>
      <visibility value="true"/>
   </subgroupcomponent>

   <!-- 以下にインストールの定義をする -->
</subgroupinstallation>
```

ここまでで、ジオメトリの為のクラスのリストアップが終わった。10 個のクラスで 4000 行ほどのソースコードを書いていたものを 100 行程度の XML データにすることができた。JUPITER で新たにジオメトリクラスを作る操作はこの XML データの <subgroupcomponent> を加える操作に置き換えることができる。これはデータであってソースコードではない。リンクのし直しや同等コードの連なり、ハードコーディングされたデータをこれによって大いに改善することができた。

5.3.2 実体の配置

次に実体を配置していく。実体配置の際にはどのように置くのか、何個置くのかという情報が必要である。現在の JUPITER に入っている TPC では次のようになっている。

- world ボリューム内に J4TPC を置く。これは center に置かれる。
- J4TPC クラスは娘ボリュームとして J4TPCHalf をもつ。ふたつめは一つ目のクローンとしておく。J4TPC の上のボリュームは world ボリュームで
- J4TPCHalf は娘ボリュームとして J4TPCDriftRegion, J4TPCSuppotTub, J4TPCEndCap, J4TPCCentralMembrane, J4TPCPadPlane, J4TPCLayer を持つ。
- J4TPCDriftRegion は J4TPCLayer を娘ボリュームとして持つ。

- J4TPCPadPlane は J4TPCPadRow を娘ボリュームとして持つ。

このように現在の JUPITER ではクラス内にすべて構造がハードコーディングされている。これは変更しづらくメンテナンスが面倒なコードである。どのクラスがどのようにインストールされているのかを見るのにいろんなファイルを見にいかなければならない。前章までで説明した方法を用いると以下のようにひとつのプレインテキストファイルにまとめることができる。

```
<?xml version="1.0" encoding="utf-8"?>
<subgroupinstallation xmlns:jupiter="http://lcdev.kek.jp/jupiter"
                      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                      xsi:noNamespaceSchemaLocation="../../schemas/jupiter.xsd"
                      name="TPC"/>

  <!-- 上記で定数とインストールは終わった。-->

  <!-- インストールの定義  -->
  <world>
    <pvplacement ref="J4TPC">
      <placeorigin>
         <hep3vector/>
         <heprotation/>
      </placeorigin>
    </pvplacement>
  </world>

  <detectormother ref="J4TPC">
    <pvplacement ref="J4TPCHalf">
      <placeorigin>
      </placeorigin>
      <placelink>
      </placelink>
    </pvplacement>
  </detectormother>

  <detectormother ref="J4TPCHalf">
    <pvplacement ref="J4TPCDriftRegion">
      <placeorigin>
         <hep3vector/>
         <heprotation/>
      </placeorigin>
    </pvplacement>
```

```
  <pvplacement ref="J4TPCSupportTub">
    <placeorigin>
       <hep3vector/>
       <heprotation/>
    </placeorigin>
  </pvplacement>

  <pvplacement ref="J4TPCEndCap">
    <placeorigin>
       <hep3vector/>
       <heprotation/>
    </placeorigin>
  </pvplacement>

  <pvplacement ref="J4TPCCentralMembrane">
    <placeorigin>
       <hep3vector/>
       <heprotation/>
    </placeorigin>
  </pvplacement>

  <pvplacement ref="J4TPCPadPlane">
    <placeorigin>
       <hep3vector/>
       <heprotation/>
    </placeorigin>
  </pvplacement>

  <pvplacement ref="J4TPCLayer">
  </pvplacement>
</detectormother>

<detectormother ref="J4TPCDriftRegion">
  <pvplacement ref="J4TPCLayer">
    <placeorigin>
       <hep3vector/>
       <heprotation/>
    </placeorigin>
  </pvplacement>
</detectormother>
```

```
   <detectormother ref="J4TPCPadPlane">
     <pvplacement ref="J4TPCPadRow">
       <placeorigin>
          <hep3vector/>
          <heprotation/>
       </placeorigin>
     </pvplacement>
   </detectormother>

   <detectormother ref="J4TPCPadRow">
     <pvplacement ref="J4TPCPad">
       <placeorigin>
          <hep3vector/>
          <heprotation/>
       </placeorigin>
     </pvplacement>
   </detectormother>

</subgroupinstalltion>
```

これによりハードコーディングされていた構造が、ひとつのデータファイルとしてまとめることができるようになった。

5.4 XML データジオメトリ自動生成構造を JUPITER に組み込む

JUPITER に組み込むには JUPITER のエントリポイントである main.cc 内に以下を追加する。

まず XML ファイルを読み込むために次をインクルードする。

```
//*-----------------------------------------------------------
// XML 操作とそれにより作られたインスタンスの管理。Facade の役割ラス
#include "J4kern/DetectorNodeManager.hh"

// XML イベント処理コールバッククラスをリンクさせる
#include "J4kern/BuildingSubscribersRegistry.hh"
//*-----------------------------------------------------------
```

そして main 内で次を行う。

```
//*-----------------------------------------------------------
std::string xmlfilename(getenv("JUPITERROOT");
xmlfilename += "/xmldata/jupiter.xml";

DetectorNodeManager\& detectorNodeManager = DetectorNodeManager::getInstance();

// jupiter.xml のありかを教える。
// xml から DetectorNodeTree を生成する。
detectorNodeManager.setXMLGeometryFile(xmlfilename);
detectorNodeManager.constructDetectorNodeTree();

J4VComponent* tpc = NULL;

try {
   // 作った DetectorNodeTree から"J4TPC"という測定器名のキーを検索。
   tpc = &J4kern::DetectorNodeManager::getInstance().find("J4TPC");

} catch(std::runtime\_error\& e) { // 検索できなかった場合の例外処理
   std::cerr << e.what() << std::endl;
   DetectorNodeManager::getInstance().terminate();
   exit(EXIT\_FAILURE);
}
```

```
// JUPITER の DetectorConstruction
// G4VUserDetectorConstruction 継承クラス
// JUPITER のジオメトリを生成
J4DetectorConstruction* dtcptr = new J4DetectorConstruction;


// JUPITER の他の測定器ジオメトリを作る

// TPC ジオメトリを作る
dtcptr-> AddComponent(\&tpc);

// Geant4 の RunManager に知らせる。
runManager-> SetUserInitialization(dtcptr);
//*------------------------------------------------------------
```

5.5 XMLデータにより作成されたTPCの図

以上、ソースコードをXMLデータに置き換える流れを見てきた。これによりJUPITERフレームワーク上でXMLデータによりジオメトリを作成することができるようになった。以下にXMLデータを用いて作成したTPCジオメトリを図5.3に示す。

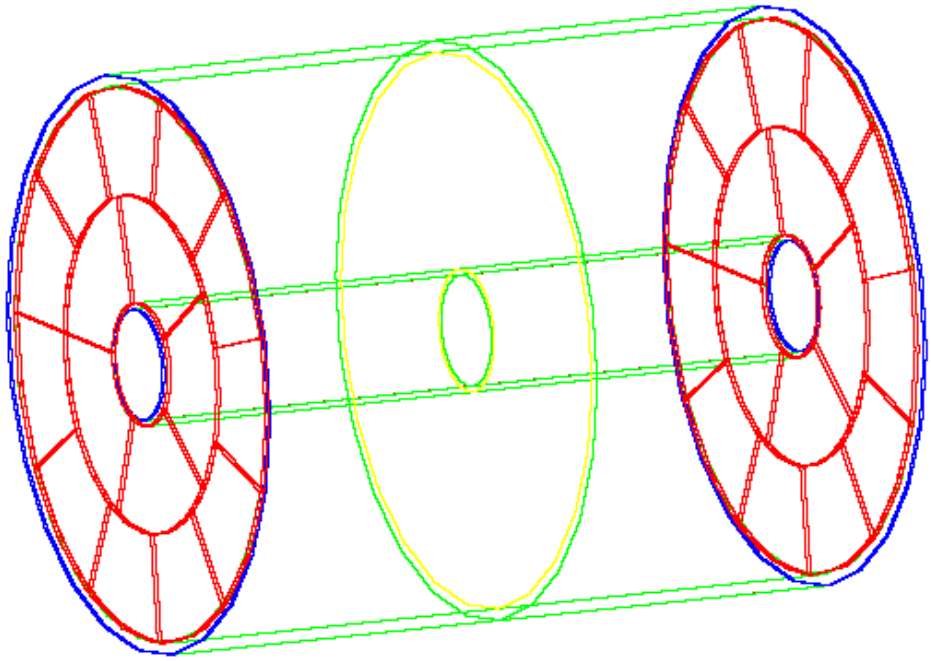

図 5.3: TPCジオメトリ図

第6章　結論

開発の結果

ILC 日本グループ、オフラインソフトウェア解析チームは開発の結果、JUPITER をフレームワークとし、線型加速器用測定器シミュレータを開発してきた。現在、図 6.1 に示すような GLD のジオメトリが入っており、デバッグとともに解析の段階に移行しようとしている。

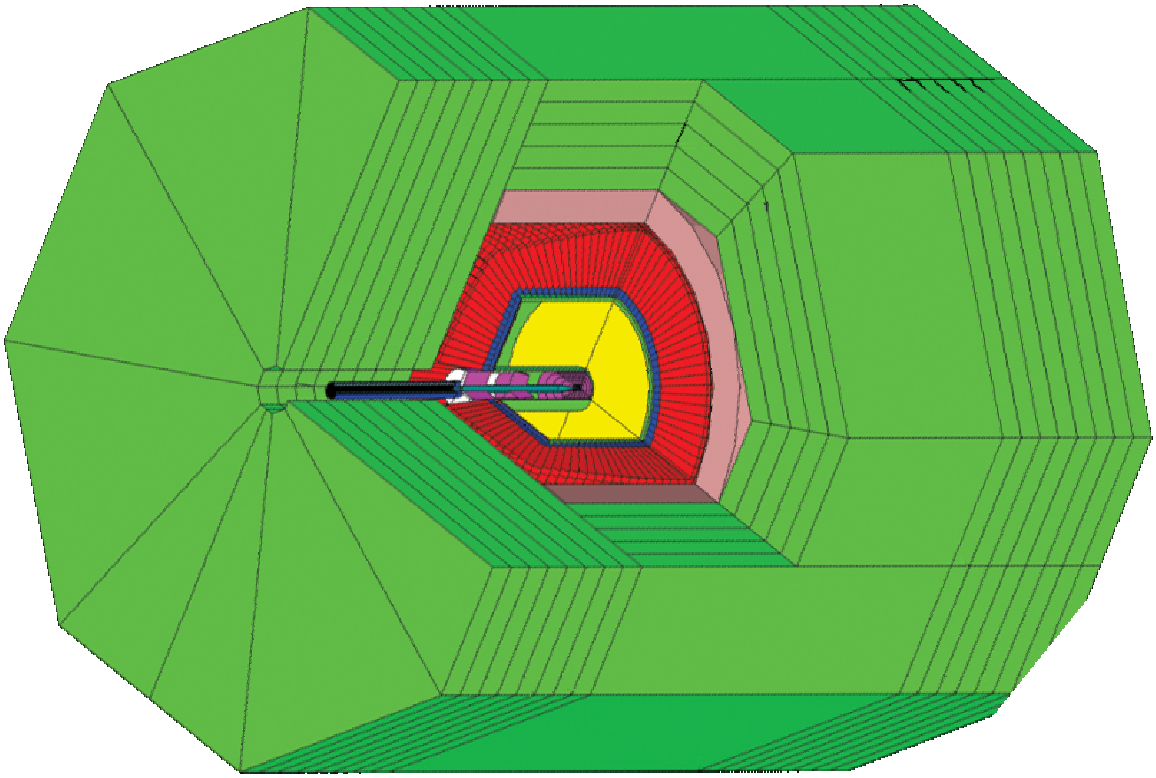

図 6.1: JUPITER に入った GLD のジオメトリ

XML インターフェースの開発

しかし JUPITER 内のジオメトリは急ピッチで作成されてきたため一度ジオメトリを入れるとその構造は開発者以外は動かし難い状況になってしまっており、プログラマーでない実験家に測定器シミュレーションにおける検出器のインストール、アンインストールの簡単な方法を提供するという目的を達せれていない。

これを解消すべく汎用的なデータフォーマットであり、可読性に優れる XML をデータ形式として選択し JUPITER にそのインターフェースを追加した。JUPITER の開発と同時にこれを進め、XML インターフェースクラス群、JUPITER がもつべきデータモデルである W3C XML Schema を用いたスキーマを開発した。またこれが実際に動くかどうか、中央飛跡 TPC を例題に作成し、テストした。その結果、このインターフェースを用いれば XML データを用いて JUPITER フレームワークの中でジオメトリを作成することができ、単純な数千行のソースコードを数百行のプレインテキストの XML 形式のデータに置き換えることができるようになった。またこれは現在の手作業でのクラスのインストールという手段を提供する部分を壊してはいないため現在のシステムを壊すものではなく、完全に新たな方法を追加することができたといえる。

測定器応答 (SD) やヒット (Hit) に関しては、クラスをデータに置き換える意義は考えられないためそれぞれをユーザが作成するようにし、それらのクラスの名前をデータとしてつなげるようにした。しかしこれは、現在の JUPITER に入っている SD や Hit をそのまま使えばいいというわけではない。SD や Hit は多くの場合、JUPITER 内でプログラミングされているデータを使っているからである。つまり現在 SD 内でも Geometry クラス群からの情報を多数利用している。ここを書き直せば、これから変化が激しいと予想できるジオメトリをクラスとしてではなく変更しやすい XML フォーマットに完全に移行させることができたといえる。現在、その部分のデバッグ中であり、あと一歩という状況である。

今後の展望

まず SD や Hit の部分のデバッグを完了させ TPC の SD や Hit を XML のジオメトリで完全に置き換えることができたという例題を示す。それからその他のジオメトリの簡単な部分を徐々にデータとして置き換えていく予定である。

謝辞

本研究を進めるにあたり、適切なご指導、ご助言をあたえて下さった指導教官である川越清以先生に深く感謝致します。またビームテストでお世話になったカロリメータグループ、いっしょに開発をすすめてきた acfa-sim-j のメンバーの皆様に深く感謝いたします。さらに、武田廣教授、野崎光昭教授、原俊雄助教授、蔵重久弥助教授、石井恒次助手、鈴木州助手、越智敦彦助手をはじめとする神戸大学粒子物理学研究室の皆様にも深くお礼を申し上げます。最後に、両親となぎちゃん、いっつも支援本当にありがとう。

参考文献

[1] ”JLC-I”, KEK Report 92-16, December, 1992

[2] http://www-jlc.kek.jp/subg/offl/jsf

[3] http://www-jlc.kek.jp/subg/offl/www-jlcsim/index.html

[4] KEK Report 2003-7 “GLC Project” Linear Collider for TeV Physics (2003)

[5] http://wwwasd.web.cern.ch/wwwasd/geant4/geant4.html

[6] http://root.cern.ch/

[7] http://xml.apache.org/xerces-c/

[8] http://xml.apache.org/xalan-c/

[9] http://apache.xml.org/

[10] http://gdml.web.cern.ch/GDML/

[11] http://www.w3.org/XML/

[12] http://www.w3.org/XML/Schema

[13] オブジェクト指向における再利用のためのデザインパターン改訂版

表 目 次

図 目 次