JUPITER

K.Hoshina

July 4, 2002

Abstract

JUPITER (JLC Unified Particle Interaction and Tracking EmulatoR) は、JLC の次世代 Full Simulator として開発中の、Geant4 ベースの高エネルギー加速器実験検出器用 Simulator です。現段階においては、JUPITER の最も重要な役割は検出器パラメータの最適化であるので、開発にあたって、検出器のインストールやアンインストール、構造の変更等が簡単に行えるようにすることにかなりの重点がおかれています。その結果、Geant4 開発グループが例示するコーディング方法とは、かなり異なる部分が出てきてしまいました。初めて御覧になる方は、あるべき場所に必要なコードが存在しないのを見て、きっと驚かれることでしょう。既に Geant4 を使いこなしている方には、かえって使いにくいと感じられるかも知れません。JUPITER では、コーディング作法のかなりの部分を抽象クラスに決められてしまいます。あえてこのような窮屈な形式を選んだのは、のちのち JSF (JLC Study Framework) と連係して JUPITER を動かす可能性をにらんだということと、高エネルギー加速器実験検出器用の Simulator としては、必ずしも Geant4 の持つ豊富な機能の全てを使いこなす必要はないだろうと判断したからです。(少なくとも測定器パラメータの最適化を行っているうちは。) もちろん、JUPITER の使命が検出器パラメータの最適化から物理解析に移行する段階で、JUPITER は構造から見直す必要があると思います。

このマニュアルは、次の 4 つの部分から成っています。

1. Chapter 1 JUPITER の開発方針
2. Chapter 2 JUPITER の実装
3. Chapter 3 JUPITER のコンパイルと実行
4. クラス・リファレンスマニュアル (主なベースクラスについて)

すぐに実行させてみたい方は、Chapter 3 から目を通すのが早いでしょう。ここに、必要な環境変数の定義なども書かれています。一方、JUPITER の実装を始める方は、是非 Chapter 1 から目を通して下さい。複数人数でコードを書くときに決めておくべき事柄が書かれています。

JUPITER はまだα 版のプログラムであり、現在もまだ一番根幹の抽象クラスに手を加えている段階です。なるべく公開メソッドには手を加えないようにしていますが、やむを得ず公開メソッドに変更を加える場合もあります。どうか、いましばらく御容赦いただけますよう、お願い致します。

御意見、御叱責は保科 (hoshina@post.kek.jp) まで。お待ちしております。

Contents

1 JUPITER 開発における方針 **3**

1.1 JUPITER にどこまでシミュレーションさせるか？ . . . 3

1.2 コーディングの約束事 (Coding convention) . . . 3

1.2.1 名前のつけ方 (Naming convention) . . . 4

1.2.2 ファイル名 . . . 4

1.2.3 クラス定義 . . . 5

1.2.4 組み込み型の使用について (raw C type の禁止) . . . 5

1.2.5 Template、Exception handling、RTTI、namespace . . . 5

1.2.6 コメント . . . 6

1.2.7 public, private の順番 . . . 7

1.2.8 推奨コーディングスタイル (Preferred Cording Style) . . . 7

1.3 ディレクトリ構造 . . . 9

2 JUPITER の実装 **11**

2.1 ベースクラスの説明 . . . 11

2.1.1 J4Object と Object の持ち主の管理 . . . 11

2.1.2 J4VComponent . . . 13

2.1.3 J4VSensitive Detector, J4VSD, J4VHit . . . 15

2.1.4 J4VMaterialStore、J4MaterialCatalog . . . 17

2.1.5 new と delete を管理するクラス . . . 17

2.2 具体的な Component の作成 . . . 18

2.2.1 サブグループのベースクラスの作成 . . . 18

2.2.2 Component の作成 . . . 21

2.2.3 コンストラクタに与える引数 . . . 21

2.2.4 Assemble() 関数の実装 . . . 24

2.2.5 InstallIn() 関数の実装 . . . 26

2.2.6 Cabling() 関数の実装 . . . 26

2.3 SensitiveDetector と Hit の作成 . . . 28

2.3.1 SensitiveDetector クラスと Hit クラスの作成 . . . 28

2.3.2 SensitiveDetector の実装 30
2.3.3 Hit クラスの Output() 関数の実装 32
2.4 Jupiter.cc の変更 33

A JUPITER Base Class リファレンス **38**
A.1 J4VComponent 38
A.1.1 データメンバ 38
A.1.2 主なメンバ関数 39

Chapter 1

JUPITER開発における方針

まず始めに、複数人数で JUPITER を開発するにあたり、スムーズな開発を行うために最低限決めておいた方がよいと思われる方針を挙げることにします。もしこの他にあれば、是非保科 (hoshina@post.kek.jp) までご連絡下さい。

1.1 JUPITER にどこまでシミュレーションさせるか？

まず、JUPITER の仕事範囲を明確にしておくことにします。JUPITER は、計算速度、その他の面から、Monte-Carlo Truth をアウトプットするところまでを担当するという取り決めになっています。従って、Hit の Smearing や Digitizing は、JUPITER の外 (Satellites) で行います (図??)。このことから、JUPITER には、最小の Hit data 読み出し単位が組み込まれている必要があります。いくつかの読み出し単位をまとめて大きな単位で読みたい場合には、まず JUPITER に最小の読み出し単位でアウトプットさせ、Satellites でより大きな読み出し単位にまとめて下さい。

1.2 コーディングの約束事 (Coding convention)

大掛かりなプログラムを作成するにあたって、まず初めにファイル名、クラス名、変数名などの付け方を決めておくことは大変有用だと思われます。得に、複数の人がコードを書く場合には、ファイル名やクラス名が重ならないようにしておく必要があります。

JLC では解析プログラムに JSF を使っており、ROOT の扱いに慣れている人が多いだろう、という判断から、Coding convention は大部分を ROOT(Taligent？) の convention に合わせさせていただきました (勝手に付け加えたものもあり)。一部未対応の部分がありますが、早急に対応の予定です。ROOT のホームページにいくと、非常にこと細かな convention が書いてあります。Preferred Coding Style の方は、少々うるさく感じられるかも知れませんが、C の開発者やプログラマの神様のような方々のお言葉はさすがに含蓄があり、納得出来るものですので、これから C++を学ぼうという学生さんなどは、是非このスタイルでコードを書く癖をつけるとよろしいと思います。(くれぐれも、Geant4 の Example の真似はしない方がいいと思います。開発者もあのコードはひどいと認めていますので。)

ROOT の Coding convention のページはこちらにあります (http://root.cern.ch/root/Conventions.html)。以下、該当ページを訳しただけなので、もし訳間違いがあったら教えて下さい。

1.2.1 名前のつけ方 (Naming convention)

Orphan は「孤児」の意味。自分が new で作った Object の所有権を放棄する (つまり delete の責任を Orphan 関数の呼び手に明け渡す) 場合に使います。逆に、他人が New で作った Object の所有権をもらうのが Adopt (養子にする) で、delete の責任も譲り受けます。Adopt は他人の作った Object をウッカリ消してしまう危険の伴う関数なので、なるべく使うべきではない、とのこと。

Identifier	Convention	Example
Base class 名	J4+sub group 名で始める	J4CDCSenseWire
多重継承 class 名	J4M+sub group 名で始める	J4MCDCLockable
Virtual Base class 名	J4V+sub group 名で始める	J4VCDCDetectorComponent
マクロ名	J4+sub group 名で始め 前後を__でかこむ	__J4CDCMATERIAL__
列挙型	E で始める	EInside
Data Member	f で始める	fWire
関数	大文字で始める	Print()
Static 変数	g で始める (関数中の変数と Global 変数)	gDirectory
Static Data Member	fg で始める (Class の Data Member)	fgMode
Local 変数	小文字で始める	track, theCurrentArea
定数	k で始める	kTolerance,
テンプレート Type	A で始める	AType
Getter	Get... または Is...(boolean) で始める	GetWire(), IsDone()
Setter	Set... で始める	SetName()
Object を New で作る	Create..., で始める	CreateName()
Object をコピー	Copy..., で始める	Copy()
Object の所有権を放棄する	Orphan..., で始める	Orphan()
Object の所有権を移行する	Adopt..., で始める	Adopt()

1.2.2 ファイル名

ここは、Geant4 の規則に準じます (Make file を Geant4 のものを使っているため)。したがって、

1. ヘッダーファイルは.hh、実装ファイルは.cc をつける
2. ファイル名はクラス名をそのまま採用 (1 クラス 1 ファイル)

ちなみに、インクルードファイルは次節に述べる理由で作りません。

1.2.3　クラス定義

まず、お約束を先に並べます。

1. データメンバは常に private。継承クラスに Direct Access を許す場合のみ Protected。
2. 1 行でおさまる Getter, Setter は、inline にして.hh ファイルに直接書き込む。
3. 1 行でおさまらない inline は、まとめて.hh ファイルの最後に書く。.icc ファイルに分けない。include 宣言は、宣言部に書く。

まず、1. は Object Oriented の基本だからいいとして、2. の理由は、「プログラムを読む時間の短縮のため」だそうです... 徹底してます。しかし、長いプログラムから Getter や Setter の本体を探して探り当て、「なんだ、ただ field 返してるだけじゃん」という時の脱力感は... 経験者は御存じでしょう。また、Geant4 でお馴染みの.icc ファイルも駄目、とのこと。.icc は、build time を増やし、Makefile を複雑にし、もっと致命的には、あるキーワードを探す File の数を増やすのでいいことはない、ということらしいです。確かに、ヘッダーで関数を探し、それがインラインだった場合には別の icc ファイルを開かなければならない、というのはかなり消耗です。ついでに、(勝手に付け加えましたが) inline 宣言は宣言部についているべきだと考えます。理由は、遥か下の方にある inline 実装部は vi などでは見えないことが多く、実装部を探して.cc ファイルを捜しまわる愚を避けるためです。

Taligent のホームページに曰く、「コードは書くのは 1 度だが、読むのは度々である」。

1.2.4　組み込み型の使用について (raw C type の禁止)

これは Geant4 のお約束に準じます。int ではなく G4int、double ではなく G4double を使う等等、という意味です。

1.2.5　Template、Exception handling、RTTI、namespace

テンプレートについては、近年改善されたものの、まだまだプラットフォームやコンパイラによって対応にばらつきがある、とのこと。教科書を見ると許されているはずなのに、コンパイルが通らない等の問題は頻繁に起こります。

また、大多数のコンパイラは、テンプレートから実クラスをつくるところに大量に時間を費やすらしいです。1 つや 2 つのテンプレートならまだしもですが、それ以上になると指数関数的に link performance が悪くなるらしい... JUPITER はテンプレートを使っていますので、確かにかなりリンクが遅いです。

とりあえずの応急処置として、将来テンプレートに移行出来る形で実クラスを作っておき、コンパイラの改善を待つ、という手はあります。現在の JUPITER のテンプレート使用部分も、そのような応急処置を考案中です。

Exception handling については、Run time に影響が出る、とありますが、Geant4 では本体がガンガン Exception handler を使っていますから、JUPITER でチマチマ削ったところで、あまり効果はないかも知れません。ちょっとでも速くしたい人は、TRY, THROW, CATCH macro を使え、とのこと。

RTTI、Namespace は、まだ対応しているコンパイラが少ないので、当面はおいておきます。

1.2.6 コメント

ソースコードにコメントを書くというのは大事な作業ですが、書く位置を決めておくと、あとで自動的にHTML ファイルを生成するときに役立つだろう、ということで、とりあえず ROOT のお約束に倣います。

データメンバの説明

データメンバの説明は、.hh ファイルの定義の横に書きます。

```
G4int fNcells;              // the number of cells
```

等。

クラスの説明

クラスの説明は、.hh ファイルの source code が始まる前に書きます。その際、 1 行目の区切り線を忘れないようにして下さい。

```
//_____________________________________________________
//
// J4VCDCDetectorComponent
//
// This class is the abstract base class of CDC components.
// Use this class by inheriting from it and overriding the
// members, Assemble() and InstallIn() .etc.
//
```

メンバ関数の説明

関数の説明は、関数の実装部の中に書きます。{ のすぐ次の行に書かれたコメントと、その行に続くコメントは、関数の説明とみなされます。

```
TList::Insert()
{
   // Insert node into linked list.
   // To insert node at end of list use Add().
   ...
   ...
}
```

HTML タグの埋め込み

コメントに直接図を埋め込みたいとき等には、次のようにタグを使います (将来 HTML 書出しができるようになったときに、ブラウザで図が見られるようにするためのもの)。

リンクは、”figs/サブグループ名/” の下に張って下さい。

```
TList::Insert()
{
   // Insert node into linked list.
   // To insert node at end of list use Add().
   //Begin_Html
   /*
   <img src="figs/cdc/list.gif">
   */
   //End_Html
   ...
   ...
}
```

1.2.7 public, private の順番

これは ROOT の推奨というわけではありませんが、Geant4 のコードで統一されていなくて面倒に思った点がありますので、追加します。

ヘッダーファイル (.hh ファイル) は、public 関数、protected 関数、private 関数、public データメンバ、protected データメンバ、private データメンバの順で書きます。private 関数と private データメンバが続いてしまう場合にも、データメンバの前に再度 private 宣言をしなおします。(基本的に、他人が見てもよいものをファイルの上部に持って行く、という思想。インターフェースとデータメンバでは、インターフェースの方が見られる頻度が高い。データメンバは、本来隠しておくべきもの。)

1.2.8 推奨コーディングスタイル (Preferred Cording Style)

以下は、推奨されるコーディングスタイルです。特にこれでなくてはならない、という訳ではありませんが、スタイルを決めかねている人は参考にしたら良いと思います。ちなみに、提唱者は C 言語の開発者 Kernighan と Ritchie のようです。

少なくとも、いろいろスタイルが混じっている、というのは開発者本人以外には大変読みづらいと思いますので、どれかに統一するよう心掛けて下さい。

インデント

インデントにはタブではなくスペースを使え、とのことです。タブにすると、他人が別のエディタで見たときに、タブの設定によっては大変見づらくなるから、ということであるらしい。開発者の人数が増えてきたらいずれ問題になる点ですので、現在出来ている部分もおいおい直していきます。

ちなみに、インデントなしのコーディングというのは問題外です。(大変読みにくい)

かっこの位置

かっこの位置は、如何にして読み易さを損なわずコードの行を減らすか、という観点から、次のように提唱されています。

関数は、条件文の後に開き括弧を書き、とじ括弧は 1 行使う。

```
if (x is true) {
   we do y
}
```

ネストする場合は、とじ括弧の後ろに else 文を書く。

```
if (x == y) {
   ..
} else if (x > y) {
   ...
} else {
   ....
}
```

do while 文の場合は、次の通り。

```
do {
   body of do-loop
} while (condition);
```

例外として、関数の定義の場合のみ、開き括弧にも 1 行使う。

```
int function(int x)
{
   body of function
}
```

関数の場合のみ特別扱いする理由は、まさしく「関数は特別だから」だそうです。(関数はネストすることができない)

条件文の書き方

これも、見やすければ良いわけですが、思想としては条件の文字が目立つような書き方、と言う意味で、Operator と Keyword の間にはスペースを置きます。{ と (、　) と } の間も同様。

推奨例：

```
int aap(int inp)
{
   if (inp > 0) {
      return 0;
      int a = 1;
      if (inp == 0 && a == 1) {
         printf("this is a very long line that is not yet ending", a, inp,
```

```
                a, inp, a, inp);
        a += inp;
        return a;
      }
   } else {
      return 1;
   }

   if (inp == 0)  return -1;
   return 1;
}
```

1.3 ディレクトリ構造

JUPITER のディレクトリ構造は、図 1.1 の通りです。

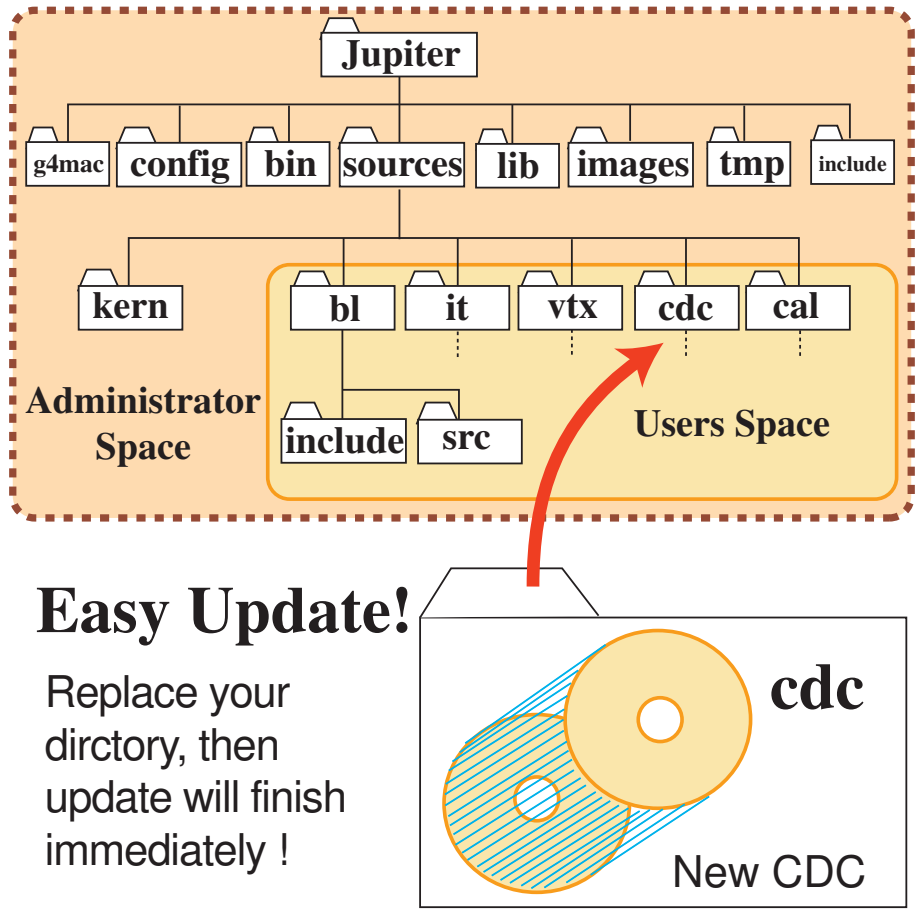

Figure 1.1: JUPITER のディレクトリ構造

基本的に、各サブグループの検出器を記述する部分は、sources 以下のそれぞれのディレクトリに全て置くという方針です。これは、それぞれのサブグループが、検出器部分のバージョンアップをするのに、該当するディレクトリを置き換えるだけで済むようにするためです。

Geant4 の例題では、検出器のインストールは DetectorConstruction クラスの Construct 関数の中で行うことになっていますが、JUPITER では、上記の特性を生かすため、Construct 関数の中で抽象クラス (J4VDetectorComponent クラス) のインストールメソッドを呼ぶのみとなっています。また、Hit データの書き出しも同様で、通例では EventAction クラスの中の EndOfEventAction 関数の中で書き出されるのですが、JUPITER では抽象クラスの OutputAll メソッドが呼ばれるのみです。このことにより、検出器全体で共有するクラスにサブグループ独自のコードが混入するのを防ぎ、これらのファイルを各サブグループのディレクトリから分離します。

このようにして、どこのサブグループにも属さないファイルのみを集めて、kern ディレクトリを作成します。kern は kernel の意味で、まさしく JUPITER の核に相当し、この kern ディレクトリの中のファイルだけで、Experimental Hall (World Volume) がおかれているのみのミニマムセットを作ることが出来ます。そして、この Experimental Hall に実際にインストールする検出器、データの書き出しを行う検出器のリストは、現在は main プログラムである Jupiter.cc の中で登録しています (今後、コマンドラインから install/uninstall するメソッドを追加予定)。したがって、ユーザー領域以外には、唯一 Jupiter.cc が各サブグループで改編可能なファイルになります (原則として)。

これらの抽象クラスの仮想関数が期待通りの働きをするためには、各サブグループが作成するクラスが、対応する抽象クラスを公開継承していなければなりません。次章より、実際の検出器部分の実装手順について説明します。

Chapter 2

JUPITERの実装

2.1 ベースクラスの説明

JUPITER では、kern の下にあるいくつかのベースクラスが非常に重要な役割を果たします。サブグループのコンポーネントは、これらのベースクラスを継承したクラスを定義して作ることになります。はじめに、これらのベースクラスについて述べます。UML 図は図 2.1 を御覧下さい。

2.1.1 J4Object と Object の持ち主の管理

J4Object は、JUPITER でユーザーが定義する全てのクラスの親になるクラスです。JUPITER のベースクラスは全てこの J4Object を継承しています。J4Object の仕事は、new 演算子で作られた Object の持ち主の管理であり、Register() と Deregister() の関数を持ちます。したがって、ユーザーは必ず持ち主が変わることがないと保証できるもの (オブジェクトが 1 つしか作られない RunManager などの singleton が new する Object)、時間的に、どうしてもその管理を行っていると処理速度が遅くなってしまうもの (Hit など 1 イベントで大量にオブジェクトを new するもの) を除いて、必ず new の後に Register() 関数を呼ぶ必要があります。

```
J4Object*  something = new J4Object();
Register(something);
```

また、JUPITER では、new を行った Object が責任を持って new した Object を delete するというきまりになっています[1]。J4VComponent などは、データメンバに必ずしも自分の持ち物でない Object へのポインタも含んでいますので、delete する前に自分の持ち物であるかを確かめる必要があります。デストラクタで、次の作業を必ず行って下さい。

```
if (Deregister(something)) delete something;
```

[1] 唯一の例外が、J4Union, J4Substract を使って Object を作る場合です。これらは、2 つの Solid を合わせて新しい Solid を作りますが、材料となる 2 つの Solid の delete に関しては、これらのクラスが自動的に行ってくれます。この関係で、これらのクラスを使って Solid を作る場合、材料の使い回しはできません。材料となる Solid は、必要な数だけ new して下さい。

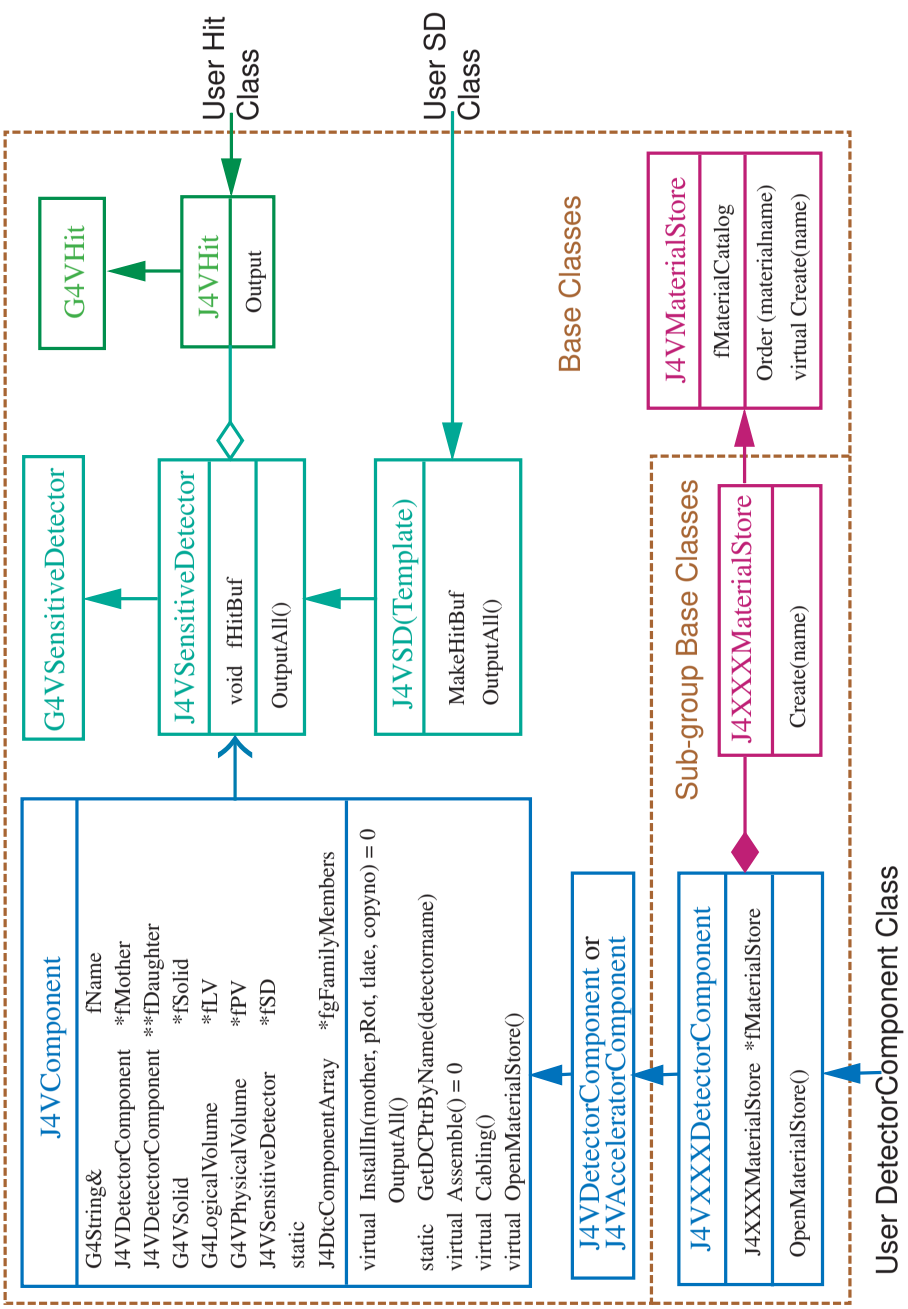

Figure 2.1: JUPITER のベースクラス UML 図

この if 文の中で、something に対する登録された親を探しだし、それが自分自身 (this ポインタ) と一致すれば TRUE を、一致しなければ FALSE を返します。

また、ユーザーが直接 Geant4 のクラスを継承して新しいクラスを作る場合は、かならず J4Object を多重継承する必要があります。

2.1.2 J4VComponent

J4VComponent クラスは、全てのコンポーネント (部品) を代表するクラスです。名前の示す通り、純粋仮想関数 (Pure virtual) を持つ抽象クラスであり、このクラスを継承して仮想関数を実装すると Detector や Accelerator の組立、インストールが行えるようになります。

J4VComponent クラスは、Component に関する全てを知っているクラスです。名前、形、材質、位置はもちろん、自分自身の中に、どんな名前の娘 Component がいくつ入っているか (fDaughters)、親 Component (自分がインストールされる Component) は何か (fMother)、自分自身が World Volume から数えて何番目の階層にいるか、Sensitive Detector へのポインタ (fSD)、同じクラスから作られた形状等の違うオブジェクトがいくつあるか (fNbrothers)、 Copy または Replica で作られるまったく同じオブジェクトがいくつあるか (fNclones)、同じクラスから作られた形状等の違うオブジェクトの何番目か (fMyID)、コピーされた場合のコピーナンバー (fCopyNo) などを知っています。これらの情報の大部分は、このクラスのオブジェクトが作られる際や、親 Component にインストールする際に引数として与えられます。したがって、オブジェクトの生成、インストールの際には、正しい引数を与えるよう注意が必要です。どのような引数を与えればよいかは、第??節を御覧下さい。

Geant4 に慣れている方には、J4VComponent クラスは PhysicalVolume 1 つにつき必ず 1 つ作られるべきものである、という表現が一番分かりやすいかと思います。Geant4 の PhysicalVolume の作り方には、G4PVPlacement、G4PVReplica、G4PVParameterised の 3 種類があります。このうち、G4PVParameterised は、内部構造まで Parameterise してくれる保証が全くありませんので、JUPITER ではサポートしません。G4PVPlacement は、G4LogicalVolume を引数に取りますので、まったく同じ LogicalVolume を並べて置く場合には、同じ LogicalVolume を共有することが出来ます。このような置き方を、JUPITER では「Copy」と呼ぶことにします。

一方、G4PVReplica は、一度呼ぶだけで複数の LogicalVolume を作ってくれるように見えます。しかし実体は、ただ 1 つの PhysicalVolume を作るだけで、この PhysicalVolume は、Tracking の際、Track の通る場所へその都度出張していって役目を果たします。実体を 1 つ作るだけなのでメモリの節約になり、ROGeometory などの実装にも多用されています。JUPITER においても、Replica として配置する場合には、オブジェクトは 1 つしか作らないという思想を踏襲します。したがって、Copy で配置する場合には、Copy する数だけ Component を New して作らねばなりませんが、Replica の場合には 1 度だけ New すればよいということになります。

とはいえ、Copy と Replica の違いは、Geant4 のメモリ上の都合であって、本来あからさまには見えないのが理想であると考えます。従って、Copy または Replica によって、実質的または仮想的にいくつの部品が作られるかは、同じ fNclones 変数に格納されます。fCopyNo は、Copy される場合のみ 0 以上の値が入り、デフォルトは-1 になっています。したがって、fNclones が 2 以上でかつ fCopyNo が-1 の場合には Replica として置かれる Component、fCopyNo が 0 以上ならば Copy で置かれる Component、fNclones が 1 ならば同形のものはただ一つだけの Component となります。

J4VComponent クラスはまた、static メンバーとして、全ての Component のリストを持っています。したがって、プログラムの任意の場所で、static メンバ関数 J4VComponent::GetFamilyMembers() を呼ぶと、JUPITER にインストールされた Component の一覧を得ることが出来ます。ただし、それ以前に全て Component のインストールが終わっていることが条件なので、呼ぶ場所に注意して下さい。

J4VComponent の役割を以下に述べます。

名前の自動生成

Component の名前を自動的につけます。名前は component が置かれた階層をそのまま表しており、階層の区切りは「：」で表されます。名前の規則は、自分がインストールされる親の名前：自分自身の名前 (first name)+ID となります。ID は、fMyID、fCopyNo などから自動生成されます。

したがって、J4VComponent の継承クラスは、コンストラクタの引数に first name を要求します。この first name をデータメンバとして保存しておくか否かはユーザーの自由ですが、安易に GetName() 関数などを実装すると、フルネームを Get したい場面で、間違えて first name をとってくる危険性もあるので、十分に注意して下さい。(GetName() は J4VComponent に含まれる関数です。first name を Get する関数は、GetFirstName() などと名前を変えることをお勧めします。)

Component の組立

Component の組み立ては、Assemble() 関数の中で行います。ここで、Component の形の定義 (Solid の作成)、材質の定義 (Logical Volume の作成)、Visualization の設定をします。

また、LogicalVolume 内に娘 Component を配置する場合も、ここで生成・配置します。まず new で娘 Component を生成し、娘 Component の InstallIn() 関数を呼んで配置します。その後、娘 Component へのポインタを SetDaughter() 関数でセットします。これは、Component の階層を記憶させ、オーナーシップフラッグを立てるために必要です。

Component のインストール

インストールは、InstallIn() 関数によって行われます。インストールするためには、Component の形が出来上がっていなければなりませんから、InstallIn() 関数の最初の行で、必ず Assemble() 関数を呼んで下さい。InstallIn() 関数は、引数に親 Component へのポインタ、Rotation Matrix、Translate Vector をとります。自分自身が、親のどこにインストールされるかを指定する関数であることに注意して下さい。

このようにすると、コーディングの際に自分の親が何になるかを知っていなければならないように見えますが、かならずしもそうではありません。引数の Rotation Matrix、Translate Vector で、親側から明示的に娘 Component の位置を指定することが可能ですし、Replica で置くか PVPlacement で置くかは、娘 Component を new で生成するときに与える fNclones と fCopyNo の値で分岐させることが出来ます。勿論、Copy で置く場合には fCopyNo が-1 以外の値になるので、これも簡単に分岐できます。

検出器の配線 (Sensitive Detector の作成と登録)

Sensitive Detector の作成と登録は、Cabling() 関数の中で行います。インストールは配線まで終えて初めて終了であると考えれば、この関数を別に作る必要はなかったのですが、将来的にインストールと配線を別に行ないたい場合がひょっとしたらあるかもしれない、と考えて作りました。現次点では、InstallIn() 関数の最後で Cabling() 関数を呼べばよいと思います。(Recursive に Cabling() 関数を呼ぶようなメソッドはまだ作っていない)

Hit 情報の出力

OutputAll() 関数は、自分以下の Component について Sensitive Detector の有無を判断し、もし有れば、その Sensitive Detector が作った Hit を全て出力する関数です。これがうまく機能するためには、Sensitive Detector と Hit の実クラスを、対応する仮想クラス (J4VSD, J4VHit) を継承して作る必要があります。

Sensitive Detector のスイッチの切り替え

SwitchOn()/SwitchOff() 関数は、自分以下の Component に対して、もし Sensitive Detector があればその電源を On/Off する関数です (具体的には、SensitiveDetector を Activate/Inactuvate する)。デフォルトの引数は recursive になっていますが、たとえば引数に G4String で"me"などと指定すると、自分の娘 Component 以下に対してはスイッチの On/Off を行いません。

インストールされた Component の全リストを記憶

J4VComponent クラスは、static で J4VComponent の配列を持っています。この配列を用いて、例えば名前から、ダイレクトに各 Component へのポインタを得ることができます。

2.1.3 J4VSensitive Detector, J4VSD, J4VHit

この 3 つのクラスは、SensitiveDetector の登録と Hit の書き出しを自動的に行なう為に必要です。ユーザーは、J4VSD を継承して独自の SensitiveDetector を、J4VHit を継承して SensitiveDetector に対応する Hit クラスを作らねばなりません。以下に、それぞれのクラスの役割を述べます。

J4VSensitiveDetector

SensitiveDetector の機能に附随すべきであるいくつかの Get メソッドと、Hit の書き出しを行なう OutputAll 関数 (pure virtual)、更にその Hit を格納する HitBuffer(HitCollection) を持っています。G4VSensitiveDetector に要求されている ProcessHits 関数を実装していないのと、OutputAll 関数の実装がないことから、純粋仮想関数です。このクラスの役割は、大きく分けて次の 3 つがあります。

1. OutputAll 関数による、Output 形式の統一
2. 基本的な Get メソッドのサポート
3. HitBuffer の所持

まず、1. OutputAll 関数について述べます。J4VSensitiveDetector クラスは、J4VComponent クラスとは has-A の関係にあり、J4VComponent クラスが J4VSensitiveDetector へのポインタを所持しています。J4VComponent クラスの OutputAll 関数では、自分と自分の娘、Component の J4VSensitiveDetector を Recursive に探し、もし存在すれば、その J4VSensitiveDetector の OutputAll 関数を呼びます。この J4VSensitiveDetector の OutputAll 関数は純粋仮想関数ですから、実際に呼ばれるのは、これを継承したユーザー SensitiveDetector クラスの OutputAll 関数になります。

2. 基本的な Get メソッドについては、ユーザーが一般に使用すると考えられるものを用意してありますが、これを使用するためには ProcessHit 関数の先頭で必ず SetNewStep() 関数を呼ばなければなりませ

ん (詳細は??節を参照)。ReadOutGeometry を定義している場合は、同様に SetROHist() 関数を呼ぶ必要があります。ただし、ここで容易されているメソッドは、いずれも SetNewStep() 関数を呼ぶだけで使用することができます。

いずれも、引数に入るのは ProcessHit() 関数のこれは、Get メソッドで要求される情報の源 (G4Step* aStep) が ProcessHit 関数の引数で与えられるためで、この G4Step へのポインタを親クラスである J4VSensitiveDetector クラスに教える必要があるからです。同様に、もし ROGeometory を用いる場合には SetROGeom 関数を呼ぶ必要があります (ROGeometory を定義していない場合は、ProcessHit 関数の rohist 引数には 0 が渡って来るだけなので、Set しても無意味)。ここで用意されているメソッドは、いずれも SetNewStep() 関数を呼ぶだけで使用することができます。

最後に 3. HitBuffer (Geant4 では HitsCollection と呼んでいる。長い上よくわからない名前なのでもう少し短くしたが、JSF に倣うなら SDBuffer とすべきだったかと後悔。) についてですが、J4VComponent 側から操作する可能性を考えて J4VSensitiveDetector の持ち物としました。型は全ての HitsCollection の親である G4VHitsCollection *型になっています。

J4VSD

J4VSensitiveDetector が SensitiveDetector に共通する情報を所持するクラスであるとしたら、J4VSD は個々の SensitiveDetector に対して行なわなければならない作業を自動化するためのクラスであると言えます。したがって、ユーザーは J4VSensitiveDetector を直接継承するのではなく、J4VSD クラスから継承して独自の SensitiveDetector を作ることになります。このようなクラスを中間に置くと、クラスデザインがややこしくなるため、本当はあまりやりたくないのですが、HitBuffer を new で生成する際、G4THitsCollection テンプレートクラスが Buffer におさまるべき Hit クラスを要求するため、J4VSensitiveDetector の中におさめきれなかった、というのが現状です。

J4VSD クラスの役目は、以下の 2 つです。

1. HitBuffer の生成
2. OutputAll 関数の実装

HitBuffer の生成は、MakeHitBuf() 関数の中で行なわれます。したがって、ユーザーはこのクラスを継承して作ったユーザー SensitiveDetector クラスの Initialize() 関数の中で、MakeHitBuf() 関数を呼ぶ必要があります。Initialize() 関数は G4VSensitiveDetector の純粋仮想関数ですから、全てのユーザーが実装しなければなりません。Initialize() 関数は G4HCofThisEvent を引数にとるので、これをそのまま MakeHitBuf() 関数の引数に与えてやると、MakeHitBuf() 関数はその中で新しい HitBuffer を new し、G4HCofThisEvent の配列の末尾に付け足します。

正直な話、未だに G4HCofThisEvent が何をやっているのか、よく理解してません。どんな情報を持っていて、どのタイミングでクリアされるのか、等。一応 OutputAll 関数の引数に与えてはいますが、Hit の書き出しには使っていないし … Hit の書き出しに関しては、もう少し改良が必要だと考えています。

OutputAll 関数の実装は、次のようになっています。まず、ユーザー SensitiveDetector クラスの HitBuffer を呼びます。親クラスである J4VSensitiveDetector の持ちものですから、Get メソッドで取って来ること

ができます。次に、その中につまっている J4VHit の数を数え、for ループでまわしながら、順に J4VHit を取り出し、J4VHit クラスの Output 関数 (pure virtual) を呼びます。従って、OutputAll 関数で書き出したい情報の実装は、J4VHit クラスを継承して作ったユーザー Hit クラスで行なえばよいということになります。

J4VHit

J4VHit クラスは、全てのユーザー Hit クラスの親クラスになります。モンテカルロシミュレータが持つべき基本的な情報 (TrackID、ParticleID など) は、J4VHit クラスがあらかじめ持っていますので、ユーザーはこのクラスを継承して独自のユーザー Hit クラスを作り、各検出器に固有な情報を付け足せばいいということになります[2]。

J4VHit クラスの最も重要な役目は、Hit 情報の Output インターフェースを提供することです。J4VHit クラスは純粋仮想関数である Output() 関数を持っていますので、ユーザーはこれを継承したユーザー Hit クラスの Output() 関数の中で、Hit 情報を書き出さねばなりません。書き出し先は、J4VHit クラスの static データメンバである File Stream になります。この File Stream は、GetOutputFileStream() 関数でとってくることができます[3]。

2.1.4 J4VMaterialStore、J4MaterialCatalog

Component を作る材料を管理するのが、この 2 つのクラスです。MaterialStore は文字通り材料屋で、オーダーに応じて既製品の材料を渡したり、既製品になければ材料を調合して渡してくれます。既製品のカタログが J4MaterialCatalog です。ユーザーは、このカタログにある材料の他に、J4VMaterialStore を継承して作ったユーザー独自の MaterialStore の中で、新しく材料を調合して使うことが出来ます。

注文の方法は、以下の通りです。まず、ユーザー専用の MaterialStore に、Order() 関数を使って材料の注文を行ないます。ユーザーが Order() 関数を実装していなければ、親クラスである J4VMaterialStore の Order() 関数が呼ばれます。J4VMaterialStore の Order 関数は、J4MaterialCatalog を見て、その中に注文の品があればそれを返し、なければ Create() 関数を呼んで新たに材料を作るよう命令を下します。このとき、ユーザー専用の MaterialStore の Create() 関数が実装されていれば、要求通りの材料を生成してくれます。したがって、ユーザーの仕事は、まず J4VMaterialStore を継承して自分専用の MaterialStore を作成することと、catalog にない材料を用いる場合は、Create() 関数の中身を実装することです。

2.1.5 new と delete を管理するクラス

ベースクラスではありませんが、最後に重要なユーティリティクラスを紹介します。J4Object に組み込まれた Register() と Deregister() の関数は、中で TBookKeeper というクラスの関数を呼んでいます。このクラスは、名前の通り、登録を行うクラスです。TBookKeeper は対になったポインタ配列を持ち、ここに Object とその Object を new した Object のそれぞれのポインタを格納すると、GetParent(child) 関数で、娘 Object へのポインタを引数にとって親 Object へのポインタを返してくれます。

[2] このあたりの、J4VHit に付加すべき情報ももう少し吟味する必要あり。
[3] static 関数であるので、プログラム中のどこでも J4VSD::GetOutputFileStream() でとってくることができる。

ユーザーが作成するクラスは全て J4Object の継承クラスである必要があるので、このクラスを生で使うことはありません。

以上で、おおまかなベースクラスの説明は終わりです。次からは、実際に実装する手順について述べます。

2.2 具体的な Component の作成

2.2.1 サブグループのベースクラスの作成

まず初めに、各サブグループが必ず持っていなければならないクラスを作成します。例として、CDC グループの場合を挙げますので、以下、CDC の部分を自分のサブグループ名におきかえて御覧下さい。

1. J4VCDCDetectorComponent ← J4VDetectorComponent から公開継承
2. J4CDCMaterialStore ← J4VMaterialStore から公開継承

先に述べた通り、これらはそれぞれ J4VComponent と J4VMaterialStore から公開継承していなくてはなりません。J4VDetectorComponent は、J4VComponent を公開継承したもので、全ての検出器 Component の親クラスになります[4]。

J4CDCMaterialStore は、CDC サブグループ専用の MaterialStore になりますので、CDC グループの材料の発注は全てこの MaterialStore を通して行なわねばなりません。作り方は簡単で、J4VMaterialStore を継承してサブグループ専用の MaterialStore を作り、サブグループ専用の物質を定義する場合には、Create() 関数の中身を実装するだけです。その他の関数の実装を行なう必要はありません。

J4VCDCDetectorComponent クラスは、CDC グループの全ての Component の親クラスになります。新たに Component を作成するときは、必ずこのクラスを公開継承しなくてはなりません。このクラスの重要な役目は 2 つあります。まず 1 つ目は、CDC の Component の全ての名前に、”CDC”の文字を入れることです。これによって、他のサブグループとの名前の競合を防ぎます。 2 つめは、J4CDCMaterialStore を new することです。

サブグループ名の挿入

全ての Component にサブグループ名をつける方法は、以下の通りです。

1. fSubGroup データメンバを static で作る。
2. fSubGroup を”CDC”で初期化 (それぞれのサブグループ名で初期化)
3. コンストラクタで、サブグループ名を J4VDetectorComponent に渡す

[4]同様に、やはり J4VComponent を公開継承した J4VAcceleratorComponent が存在します。これは加速器 Component の親クラスとなるものです。現時点ではこれら J4VAcceleratorComponent と J4VDetectorComponent の差は皆無ですが、原則として、SensitiveDetector を作るようなものは J4VDetectorComponent、磁場を発生するようなものは J4VAcceleratorComponent に振り分ければよいと考えています。

これにより、J4VDetectorComponent から遡って J4VComponent クラスが、サブグループ名付きの名前を生成してくれます。新たな Component を作る際、この J4VCDCDetectorComponent クラスをスキップしていきなり J4VComponent 等を継承すると、サブグループ名が抜けてしまいますので、かならずグループ専用の DetectorComponent クラスを継承して下さい。

ユーザー専用の MaterialStore の生成

ユーザー専用の MaterialStore を new で生成するには、J4VCDCDetectorComponent クラスの OpenMaterialStore() 関数を実装します。サブグループの DetectorComponent ベースクラスでこの関数を実装しておけば、J4VComponent クラスがコンストラクタで自動的に OpenMaterialStore() 関数を呼んでくれます。詳細は、コードの実装例を参照してください。

実装例

1) J4VCDCDetectorComponent.cc

```
*******************************************************************************

#include "J4VCDCDetectorComponent.hh"

J4CDCMaterialStore* J4VCDCDetectorComponent::fMaterialStore = 0;

G4String J4VCDCDetectorComponent::fSubGroup("CDC");

//=====================================================================
//* constructor -------------------------------------------------

J4VCDCDetectorComponent::J4VCDCDetectorComponent(
                              const G4String       &name,
                              J4VDetectorComponent *parent,
                              G4int                 nclones,
                              G4int                 nbrothers,
                              G4int                 me,
                              G4int                 copyno ) :
                        J4VDetectorComponent(fSubGroup, name,
                                             parent, nclones,
                                             nbrothers, me, copyno )
{
}


//=====================================================================
//* destructor --------------------------------------------------------

J4VCDCDetectorComponent::~J4VCDCDetectorComponent()
{
   if(fMaterialStore) delete fMaterialStore;
   if(fSubGroup)      delete fSubGroup;
}

//=====================================================================
//* OpenMaterialStore -------------------------------------------------

J4VMaterialStore* J4VCDCDetectorComponent::OpenMaterialStore()
{
   if(!fMaterialStore) {
    fMaterialStore = new J4CDCMaterialStore();
   G4cerr << "*** Opend J4CDCMaterialStore ***" << G4endl;
```

```
   }
   return fMaterialStore;
}

*******************************************************************************
```

2) J4CDCMaterialStore.cc

```
*******************************************************************************

G4Material* J4CDCMaterialStore::Create(const G4String& name)
{

  G4Material* material= 0;

  if(name == "CO2Isobutane") {  // CO2Isobutane(90:10)

    //--------------
    // 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);

      A= 15.9994 *g/mole;
      G4Element* elO= new G4Element(name="Oxygen", symbol="O", Z=8., 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;

    // 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);

    // Isobutane gas (C4H10)
      const G4double denIsobutane= 2.67e-3 *g/cm3 * STP_Temperature/expTemp;
      G4Material* Isobutane= new G4Material(name="Isobutane", denIsobutane,
               ncomponents=2,
             kStateGas, expTemp);
      Isobutane-> AddElement(elC, natoms=4);
      Isobutane-> AddElement(elH, natoms=10);


    // CO2(90%) + Isobutane(10%) mixture
      density=  denCO2*0.9 + denIsobutane*0.1;
      G4Material* CO2Isobutane= new G4Material(name="CO2Isobutane", density,
                 nel=2, kStateGas, expTemp);

      CO2Isobutane-> AddMaterial(CO2      , massfraction= 90. * perCent);
      CO2Isobutane-> AddMaterial(Isobutane, massfraction= 10. * perCent);
```

```
      material = CO2Isobutane;

      G4cerr << "CO2Isobutane is installed" << endl;

  }
  return material;
}

*******************************************************************************
```

2.2.2 Component の作成

サブグループのベースクラスの作成が終わったら、いよいよ Component の作成を行ないます。例題として、J4CDCDriftRegion を挙げます。

J4CDCDriftRegion は、z 軸方向に側面を持つ円筒形を、xy 平面内で phi 分割したバウムクーヘンの一切れのような形をしています。１つの J4CDCDriftRegion の中には、1 本の Sense Wire がインストールされています。この原則はどの DriftRegion をとっても変わらないので、このように Sense Wire がインストールされたものを DriftRegion であるとみなすことにします。

この場合、自分自身の内部にインストールされる Sense Wire を new で作成するのは、DriftRegion の仕事になります。このように、親 Component が娘 Component を new 演算子で生成するようにしておけば、例えば DriftRegion の中に Field Wire もインストールしたいと考えた場合、J4CDCDriftRegion だけを変更すればよいことになります。そこで、原則として、親 Component は娘 Component へのポインタをデータメンバに持つ、という約束にしておきます。

もう１点、名前についてですが、JUPITER では Component の名前を自動生成します。Geant4 の例題等では、Solid, LogicalVolume, PhysicalVolume, SensitiveDetector などに全て違う名前をつける習慣があるようですが、JUPITER では同じ Component に属するこれらのオブジェクトには原則として全て同じ名前をつけます (Object の実体によって、これらの違いを判断出来るため)。唯一の例外は、Component をコピーコンストラクタで生成し、LogicalVolume を共有した場合で、この場合は PhysicalVolume の名前 (=Component の名前) と LogicalVolume の名前 (=Solid の名前) が異なります。

2.2.3 コンストラクタに与える引数

まず、Component がどのように生成されるかについて述べます。J4VCDCDetectorComponent (他のサブグループの場合も同じ) を継承するオブジェクトは、コンストラクタで次の引数をとります。

1. 親 Component (自分がインストールされる Component) へのポインタ
2. 同一の親 Component にインストールされる、自分と全く同じオブジェクトの数
3. 同一の親 Component にインストールされる、自分と形等は異なるが同じクラスから作られるオブジェクトの数
4. 3. のうち、自分が何番目かを表す ID
5. 2. がコピーによって生成される場合、自分が何番目かを表す ID

これら 1 ~ 5 の数は、それぞれ fMother, fNclones, fNbrothers, fMyID, fCopyNo に格納されます。

ユーザーは、これらの数を十分に考慮の上決定する必要があります。なぜならば、これらの数は、その Component の形状やインストール方法を左右する情報だからです。以下に、１つの親 Component にインストールされる同じクラスのオブジェクトについて、fNclones, fNbrothers, fMyID, fCopyNo に与えるべき数をいくつか例示します。図 2.2 も参考にして下さい。

	状況	fNclones	fNbrothers	fMyID	fCopyNo	配置法
1	Object １つのみ	1	1	0	-1	Placement
2	大きさ (材質) の違う Object が 5 つ	1	5	0 ~ 4	-1	Placement
3	大きさ (材質) の同じ Object3 つを自由に並べる	3	1	0	0 ~ 2	Placement
4	ある形を同じ大きさ (材質) の Object5 つに分割	5	1	0	-1	Replica
5	2 と 3 のかけ合わせ	3	5	0 ~ 4	0 ~ 4	Placement
6	1 ~ 3,5 と 4 の混合	不可能				

１ ~ ３と 5 は、全て親 Component に対し G4PVPlacement で置かれるオブジェクトです。Geant4 では、G4PVPlacement で PhysicalVolume を生成する限り、同じ親 Volume の中にいくつでも娘 Volume を置くことが出来ます。

PhysicalVolume は引数に LogicalVolume をとります。G4PVParametrised が使えない JUPITER では、形や材質の異なるオブジェクトには、必ずそれぞれの LogicalVolume を生成する必要があります。上の 2 がこれに相当し、fNbrothers の数だけ LogicalVolume を生成するため、コンストラクタを fNbrothers 回呼ぶ必要があります。fCopyNo はデフォルトの-1 のままにしておきます。

一方、全く同じ材質、同じ形のものを複数置きたいのであれば、必ずしも１つの PhysicalVolume につき 1 つの LogicalVolume を用意する必要はなく、同じ 1 つの LogicalVolume を複数の PhysicalVolume で使い回すことが出来ます。これが 3 の場合です。この場合には、LogicalVolume の生成は厳密に最初の１回だけにとどめなければなりません (メモリーリークを起こすので)。作り方は、最初の１つ目 (オリジナルと呼ぶ) を作るときだけ、CopyNo に 0 番を与えて通常のコンストラクタを用い、2 つめからはオリジナルと CopyNo を引数にとってコピーコンストラクタを用いて生成します。3 のコード例を以下に示します。

```
*******************************************************************************
// make first ladder object (MUST define copyNo as 0)
fLadders[0] = new J4VTXLadder(this,numOfLadders, 1, 0, 0);
Register(fLadders[0]);

// copy ladder objects (copyNo must start from 1)
G4int copyNo;
for (copyNo = 1; copyNo < numOfLadders; copyNo++) {
    fLadders[copyNo] = new J4VTXLadder(*fLadders[0], copyNo);
    Register(fLadders[copyNo]);
}
*******************************************************************************
```

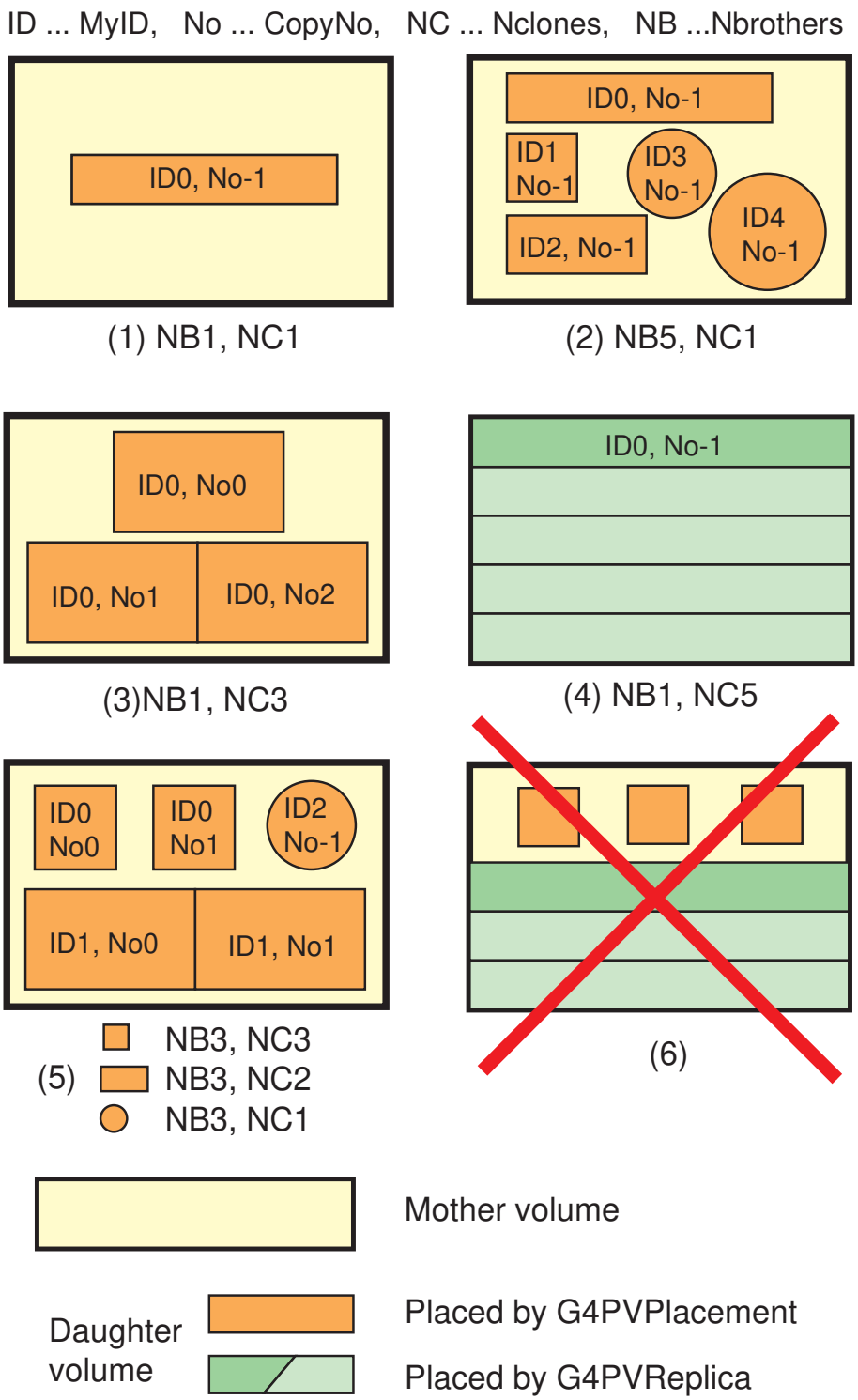

Figure 2.2: J4VComponent と引数の関係

2 と 3 はそれぞれ独立しているので、形の異なるオブジェクトをいくつか置き、そのうちのたとえば 1 つだけを複数個コピーする、などの作業も可能です。その場合は、コピーするオブジェクト (勿論オリジナルも含める) のみ fNclones に 2 以上の値を入れ、重複しないよう順に 0 以上の CopyNo をふっていけばよいことになります。その際、オリジナルの CopyNo は必ず 0 番と約束します。これが 5 の場合です。

全く同じ形のオブジェクトが、直交座標軸の方向、もしくはある回転軸を中心に phi 方向に規則正しく並んでおり、かつそれらの境界が接している場合は、G4PVReplica を用いて配置する方法があります。これが 4 の場合です。G4PVReplica は一度にたくさんオブジェクトを作ってくれるように見えますが、実は等分されたただ 1 つ分の PhysicalVolume しか作りません。Copy が LogicalVolume を使い回す方法ならば、Replica は PhysicalVolume ごと使い回す方法であり、Track の飛んでくるところへ 1 つの PhysicalVolume が出張していって仕事を果たします。従って、配置条件の制限はありますが、メモリの消費量は最も少ない配置法となります。Component オブジェクトの生成も、ただ一度通常のコンストラクタで new すればよいことになります。この場合、fNclones には仮想的にいくつのオブジェクトが作られたか (あるいは何等分されたか) を代入し、fCopyNo は-1 と約束します。

最後に、4 とその他の組み合わせはあり得ません。Geant4 では G4PVReplica と G4PVPlacement を同時に同じ親 Volume に置くことが許されていないため、選択肢は 4 またはその他の組み合わせとなります。

Object を new した後は、必ず Register() を忘れずに呼んで下さい。

2.2.4 Assemble() 関数の実装

次に、Assemble() 関数の中身を見てみます。Assemble() 関数は、Logical Volume の生成までを行ないます。この過程で、少なくとも G4Solid、G4LogicalVolume、G4VisAttributes の 3 つのオブジェクトが new で生成されます。LogicalVolume は、いくつかの Component で共有される場合があり得ますので、誰がオーナーであるのかをはっきりさせると同時に、 2 重にこれらのオブジェクトを作らないためのプロテクションが必要です。

具体的には、Assemble() 関数に入ったらすぐに、

```
if (GetLV())  return;
```

と 1 行入れるか、

```
if (!GetLV()) {
     ⋮
}
```

と書いて、括弧の間に Assemble の実装を行います。

ここまでの作業を行ったら、まず Solid を作ります。幾つかの代表的な形に関しては、簡単に Solid を作る道具が揃っています。

Solid の作成

OrderNewTubs() 関数

OrderNewTubs() 関数は、endcap 付きの筒型 Solid を作る関数です。引数に rmin(円筒の内半径) 、rmax(円

筒の外半径)、halfzlen(z 方向の半長)、totalphi(分割前の各度)、endcaphalfthickness(endcap の半厚)、endcaprmin(endcap の内半径、外半径は rmax と同じ)、sphi(phi の offset) をとります。endcaphalfthickness 引数を 0 に指定すると、endcap なしの筒型を作ることも出来ます。

気をつけなければならないのは、この関数は**J4VComponent** の **fNclones** の値を見て、自動的に円筒を **phi** 方向に分割するという点です。引数に与えられるのは、分割前の全体の phi ですので、その値を fNclones で割った値が OrderNewTubs 関数で作られるバウムクーヘン型の一つの phi 角度になります。したがって、親 Component の中身を fNclones 分割して娘 Component を置きたいという場合、たとえば 1 Layer360 度を 36 等分割して Cell を配置したい (この場合 Cell の fNclones には 36 が入っているはず) 、などという場合には便利ですが、 1 つの DriftRegion の中に完全な円筒形の Field Wire を多数配置したいという場合などでは使えません。DriftRegion の場合、Cell の中に置かれる全く同じ形の DriftRegion は 1 つしかありませんので、fNclones は 1 になります。したがって、totalphi に Cell 1 個分の各度を入れておけば、Cell の角度と同じ各度の DriftRegionSolid を作ってくれます。

OrderNewBox() 関数

OrderNewBox() 関数は、主に加速器の Component を作る為の関数です。箱形の中に箱形の空洞が空いており、更にビームパイプを通す為の円筒が開いています。パラメータの設定によって、箱形の空洞、ビームパイプ用の円筒の空洞をなくすことができます。

これらの Utility 関数を用いずに Solid を new で作成した場合には、必ず SetSolid() 関数を呼んで、作成した Solid のポインタを J4VComponent の fSolid に Set し、Register() 関数を呼びます。

LogicalVolume の作成

Solid を作成したら、次は材料を指定して LogicalVolume を作成します。LogicalVolume の作成は、MakeLVWith() 関数を呼ぶだけで、勝手に J4VComponent が実行してくれます。MakeLVWith() 関数は引数に G4Material をとりますので、引数部分で

```
OpenMaterialStore()->Order("物質名")
```

としてやれば、物質名に相当する G4Material を返してくれます。勿論、その物質がサブグループ独自の物質ならば、サブグループ専用の MaterialStore の Create() 関数の中にその物質の定義をあらかじめしておく必要があります。

この MakeLVWith() 関数を用いずに LogicalVolume を作らなければならないような場面はなかなかないとは思いますが、もしその場合には、Solid の時と同様に、SetLV() 関数で新しく new で作成した LogicalVolume を Set して、Register() してやる必要があります。

Visualization の設定

LogicalVolume まで作成したら、Visualization の設定をします。PaintLV() の 2 つの引数、G4bool (表示するか否か) 、G4Color (表示色) で設定できます。

娘 Component の作成とインストール

自分自身の LogicalVolume の生成が終わったら、自分の直下にインストールされる Component の生成とインストールを行います。

JUPITER では、全ての階層において、それぞれが自分の娘 Component を new 演算子で生成し、そのポインタを所持することになっています。娘 Component の生成にどのようなコンストラクタと引数を用いればよいかは、2.2.3 節を御覧下さい。

娘 Component を生成したら、それぞれの娘 Component に対して、InstallIn() 関数を呼びます。引数には、(娘 Component にとっての親 Component である) 自分自身を表す this ポインタ、G4RotationMatrix、G4ThreeVector を入れます。InstallIn() 関数の実装については、次節を参照して下さい。

娘 Component のインストールが済んだら、かならずこれらの娘 Component を SetDaughter() 関数で自分の娘として登録し、Register() 関数を呼んで下さい。J4VComponent クラスの Recursive call は全てここで Set された情報に頼っていますので、これを忘れると Recursive call がうまく働かなくなります。

2.2.5 InstallIn() 関数の実装

InstallIn() 関数は、先に述べた通り、自分自身が親 Component にどのように配置されるかを指定する関数です。インストールするためには、まず LogicalVolume が出来ていなければなりませんから、必ずこの関数の最初に Assemble() 関数を呼ぶ必要があります。

Install の方法は、大きく分ければ 2 つしかありません。すなわち、SetPVPlacement() 関数を使って G4PVPlacement で置くか、SetPVReplica() 関数を使って G4PVReplica で置くかのどちらかです。これらは、あらかじめ決めておいてもかまいませんし、fNclones と fCopyNo の値によってどちらでも使えるようにコーディングすることも出来ます。J4VComponent クラスには、自分がどのように置かれるべきかを返す GetPlacementType() 関数がありますので[5]、これでプログラムを分岐させることが可能です。GetPlacementType() 関数の戻り値は、kSingle, kReplicated, kCopied の 3 種類です。

Install の位置については、引数に G4RotationMatrix と G4ThreeVector をとりますので、必要ならば親 Component 側から Install の位置を指定することも出来ます。

2.2.6 Cabling() 関数の実装

Cabling() 関数では、SensitiveDetector を new 演算子で生成し、Register() 関数を呼んだあと、SetSD() 関数で登録を行います。SensitiveDetector クラスの書き方については、次節を御覧下さい。

SensitiveDetector は LogicalVolume に対して定義するものなので、SensitiveDetector の登録を行う前に Assemble() 関数を呼ぶか、fLV ポインタが 0 でないことを確かめる必要があります。また、Component をコピーコンストラクタで生成した場合のメモリリークの危険性は Assemble() 関数の場合と同様に存在しますので、fSD ポインタが 0 でない時には SensitiveDetector を new しないようプロテクトをかけておいて下さい。

ここまでのコーディング例を、図 2.3 に示します。これらのコーディングが済んだら、次は SensitiveDetector 領域の作成に取りかかります。

[5] 自分がどのように置かれるかは、自分自身が出来るときの引数で決まる。

```
//*********************************************************************
*
//-------------------------------
7/ constants (detector
parameters)
//-------------------------------
-
G4String
J4CDCDriftRegion::fName("DriftRegion");
//*********************************************************************
*
//--------------------
7/ Class Descrip-
tion
//--------------------
-
```

Full name is made from it (ex. ExpName:CDC:Layer01:Cell:DriftRegion1)

⋮

```
//=====================================================================
7/* Assemble    ------------------------------------------------------
-
void
J4CDCDriftRegion::Assemble()
{
if(!GetLV())
    // define parame-
ters
    G4double len  =
        ((G4Tubs *)GetMother()->GetLV()->GetSolid())->GetZHalfLength();
    G4double motherRmin =
        ((G4Tubs *)GetMother()->GetLV()->GetSolid())->GetInnerRadius();
    G4double motherRmax =
        ((G4Tubs *)GetMother()->GetLV()->GetSolid())->GetOuterRadius();
    G4double phi =
        ((G4Tubs *)GetMother()->GetLV()->GetSolid())-
>GetDeltaPhiAngle();
    G4int    nbrothers  =
GetNBrothers();
    G4int    myid       =
GetMyID();
    G4double thick       = (motherRmax - motherRmin)/(nbrothers +
2);
    G4double rmin        = motherRmin + thick * (myid +
1);
    G4double rmax        = motherRmin + thick * (myid +
2);
    // MakeSolid ----------
//  OrderNewTubs (rmin, rmax, len, phi
);
    // MakeLogicalVolume --//
    MakeLVWith(OpenMaterialStore()-
>Order(_CDCDRIFTREGIONMATERIAL_));
    // SetVisAttribute ----
//  PaintLV(_CDCDriftRegionVisAtt_, G4Color(0.,1.,1.));


    // Install daughter PV
//  // Install Sense Wire
//
    fSenseWire = new
J4CDCSenseWire(this);
    fSenseWire
>InstallIn(this);
    SetDaughter(fSenseWire);


}
}
//=====================================================================
7/* InstallIn   ------------------------------------------------------
-
void J4CDCDriftRegion::InstallIn(J4VDetectorComponent*
mother)
{
  Assemble();                    // You MUST call Assemble(); at
first.                          //


SetPVPlacement(0,0);
  SwitchOn();
}
```

Assemble() is a private method

Making G4Solid

Making G4LogicalVolume

Setting VisAttribute

Making a SenseWire object

Install the SenseWire object into DriftRegion object

Setting the SenseWire object as a daughter of DriftRegion object

InstallIn(this)

InstallIn(mother) is a public method

When a mother component calls this function, the DriftRegion object is installed in (0,0,0) point without rotation

⋮

Figure 2.3: J4CDCDriftRegion のコーディング例

2.3 SensitiveDetector と Hit の作成

2.3.1 SensitiveDetector クラスと Hit クラスの作成

SensitiveDetector の作成も、ベースクラスから継承して行います。まず、SensitiveDetector ですが、2 つのベースクラスのうち J4VSD を継承します。ただし、J4VSD はテンプレートクラスですので、先に Hit クラスを作って実クラスにしておく必要があります。J4VSD が要求する型は、J4VHit を継承して作った Hit クラスの型です。そこで、とりあえず Hit クラスの名前を決めてしまいます。

下のコードは ,J4CDCDriftRegionSD のコード例 (ヘッダー部分) です。Hit クラスの名前は、J4CDCDriftRegionHit としました。

```
*******************************************************************************

class J4CDCDriftRegionSD : public J4VSD<J4CDCDriftRegionHit>{

public:

  J4CDCDriftRegionSD(J4VDetectorComponent* detector);
  ~J4CDCDriftRegionSD();

  virtual G4bool ProcessHits(G4Step* aStep, G4TouchableHistory* ROhist);
  virtual void   Initialize (G4HCofThisEvent* HCTE);
  virtual void   EndOfEvent (G4HCofThisEvent* HCTE);

  virtual void OutputAll(G4HCofThisEvent* HCTE)
  {
     if(GetHitBuf())
     {
       J4VSD<J4CDCDriftRegionHit>::OutputAll(HCTE);
     }
     else
     {
       G4cerr << "J4CDCDriftRegionSD::OutputAll: No Hit! " << G4endl;
     }
  }

  virtual void DrawAll();
  virtual void PrintAll();

  // set/get functions

private:

};

*******************************************************************************
```

注目すべきは OutputAll 関数で、特に特別な書出しを行う必要がなければ、上の例のように書けば勝手に Buffer につまった Hit を吐き出してくれます。Initialize、EndOfEvent, ProcessHits 関数については次に述べるものとして、次に Hit クラスのヘッダー部分を見てみます。

```
*******************************************************************************

//=============================================================
// TypeDef
```

```
class J4CDCDriftRegionHit;
typedef G4Allocator<J4CDCDriftRegionHit> J4CDCDriftRegionHitAllocator;
typedef G4THitsCollection<J4CDCDriftRegionHit> J4CDCDriftRegionHitBuf;

//=============================================================
//---------------------
// class definition
//---------------------


class J4CDCDriftRegionHit : public J4VHit {

public:
  J4CDCDriftRegionHit();
  J4CDCDriftRegionHit(J4VComponent         *detector,
                      G4int                 cloneID,
                      G4int                 trackID        = 0,
                      G4int                 mothertrackID  = 0,
                      G4ParticleDefinition *particle       = 0,
                      G4double              tof            = 0,
                      G4double              edep           = 0,
                      G4double              totalE         = 0,
                      const G4ThreeVector  &momentum       = 0,
                      const G4ThreeVector  &pre            = 0,
                      const G4ThreeVector  &pos            = 0,
                      const G4int           hitnumber      = 0);

  J4CDCDriftRegionHit(const J4CDCDriftRegionHit &right);

  virtual ~J4CDCDriftRegionHit();

  const J4CDCDriftRegionHit&
          operator=(const J4CDCDriftRegionHit &right);
  void* operator new    (size_t    );
  void  operator delete (void *aHit);

  virtual void Output(G4HCofThisEvent *HCTE);
  virtual void Draw();
  virtual void Print();

  virtual G4ThreeVector GetHitPosition() const;

private:

  G4ThreeVector                        fHitPosition;
  static J4CDCDriftRegionHitAllocator  fHitAllocator;

};
*******************************************************************************
```

まず、行わなければならないのは 2 つの typedef です。custom new が本当に必要であるかは議論を要するところなのですが、とりあえずは Geant4 の例題に従って、custom new のための Allocator の定義を行います。HitAllocator は、クラスに 1 つあれば良いものなので、static data member として定義します。HitCollection(HitBuf) の実体は、上で作った SD クラスの Initialize() 関数の中で J4VSD が提供する MakeHitBuf() 関数を呼ぶと、その中で new されます。このとき new される HitCollection オブジェクトの型は、やはり Hit クラスを引数にとる G4THitsCollection テンプレートから作られます。この型名を表すために、 2 つ目の typedef が必要になります。

コード中央にある operator new と operator delete の実装は、inline 関数で行います (速度を要求され

るため）。このあたりのコードは Geant4 の例題の通りです。

```
*******************************************************************************
//---------------------------------------
// inline function for J4CDCDriftRegionHit
//---------------------------------------

inline J4CDCDriftRegionHit::J4CDCDriftRegionHit(const J4CDCDriftRegionHit &right)
{
  fHitPosition  = right.fHitPosition ;
}

inline const J4CDCDriftRegionHit&
  J4CDCDriftRegionHit::operator= (const J4CDCDriftRegionHit &right)
{
  fHitPosition  = right.fHitPosition ;
  return *this;
}

//---------------------------------------
// Allocator

inline void* J4CDCDriftRegionHit::operator new(size_t)
{

  void* aHit;
  aHit= (void*)fHitAllocator.MallocSingle();
  return aHit;
}

inline void J4CDCDriftRegionHit::operator delete(void *aHit)
{
  fHitAllocator.FreeSingle((J4CDCDriftRegionHit*) aHit);
}

*******************************************************************************
```

ヘッダーファイルが作成できたら、関数の実装を行います。

2.3.2 SensitiveDetector の実装

注意点は 2 つあります。まず、Initialize() 関数の中で MakeHitBuf() 関数を呼ぶことです。先に述べたとおり、この関数を呼ばないと Hit を格納するための HitBuffer (HitCollection) を作りません。

2 つめは、ProcessHis() 関数に入ったらまず SetNewStep() 関数を呼ぶことです。J4VSD と J4VSensitiveDetector はいくつかの代表的な Hit 情報についての Getter を提供しますが、これらがまともに動くためには、常に新しい Step の情報を与えてやる必要があります。これを行わずに Getter を用いればヌルポインタ参照で死んでくれると思いますが、もしまかりまちがって古い情報が入っていたりすると、全く意味のない情報を返してくることになります。

なお、ProcessHits() 関数は G4Step だけでなく G4TouchableHistory へのポインタも引数に取りますが、この引数は ReadOutGeometry を定義していないとヌルポインタで渡ってくる模様です。JUPITER では当面 ReadOutGeometry を使用しませんので、当面 SetROHist() 関数を呼ぶ必要はありません。

なお、Hit を生成して HitBuffer に insert するところ、EndOfEvent() 関数の実装の仕方は、Geant4 の例題と同じです。Hit はここで new しますが、HitBuffer の中の Object はどうやら Geant4 が自動的に後始

末してくれるようですので、Register() を呼ぶ必要はありません。J4CDCDriftRegionSD の実装を下に示します。

```
*******************************************************************************
//=============================================================
//* constructor -------------------------------------------

J4CDCDriftRegionSD::J4CDCDriftRegionSD(J4VDetectorComponent* detector)
                   :J4VSD<J4CDCDriftRegionHit>(detector)
{
}

                                      ⋮

//=============================================================
//* Initialize --------------------------------------------------

void J4CDCDriftRegionSD::Initialize(G4HCofThisEvent* HCTE)
{
  //create hit collection(s) and
  //push H.C. to "Hit Collection of This Event"

  MakeHitBuf(HCTE);

// G4cerr <<" J4CDCDriftRegion:Initialize is called " << G4endl;

}

//=============================================================
//* ProcessHits -------------------------------------------

G4bool J4CDCDriftRegionSD::ProcessHits(G4Step* aStep, G4TouchableHistory* ROhist)
{

  //In order to use Get function, you must call SetNewStep() at first.

  SetNewStep(aStep);

  //Only when a charged particle has just come into a sensitive detector,
  //create a new hit

  if(!GetCharge()) return FALSE;

  //Increment HitNumber

  IncrementHitNumber();

  //Get perticle information

  J4VComponent            *location      = GetComponent();
  G4int                    trackID       = GetTrackID();
  G4int                    mothertrackID = GetMotherTrackID();
  G4int                    cloneID       = GetCloneID();
  G4ParticleDefinition    *particle      = GetParticle();
  G4double                 tof           = GetTof();
  G4double                 edep          = GetEnergyDeposit();
  G4double                 etot          = GetTotalEnergy();
  G4ThreeVector            p             = GetMomentum();
  const G4ThreeVector     &pre           = GetPrePosition();
  const G4ThreeVector     &pos           = GetPostPosition();

  // Create a new hit and push them to "Hit Coleltion"

     J4CDCDriftRegionHit* hit =
```

```
      new J4CDCDriftRegionHit( location, cloneID, trackID, mothertrackID, particle,
                               tof, edep, etot, p, pre, pos, GetHitNumber());

    ((J4CDCDriftRegionHitBuf*)GetHitBuf())-> insert(hit);

    return TRUE;
}

*******************************************************************************
```

2.3.3 Hit クラスの Output() 関数の実装

Hit クラスで実装が必要なのは、Output() 関数です。J4VSD の OutputAll() 関数はこの Hit クラスの Output() 関数を呼びますので、最終的に書き出したい情報は全てここで Output する必要があります。

Hit クラスの作り方は Geant4 の例題と同じですので、Output ストリームの呼び出し方についてのみ触れます。J4VHit クラスは、static で Output ストリームを持っています。これを呼び出す関数も Static 関数で、

```
  ofstream& ofs = GetOutputFileStream();
```

とすれば、簡単に Output stream をとってくることが出来ます。J4CDCDriftReigionHit の実装を以下に示します。

```
*******************************************************************************

void J4CDCDriftRegionHit::Output(G4HCofThisEvent* HCTE)
{

  G4int wireNo  = GetComponent()->GetMyID();
  G4int layerNo = GetComponent()->GetMother()->GetMother()->GetMyID();

  G4String pid  = GetParticle()->GetParticleName();
  G4int    pdid = GetParticle()->GetPDGEncoding();

  G4double dphi = ((G4Tubs*)GetComponent()->
                        GetSolid())->GetDeltaPhiAngle();
  G4double sphi = ((G4Tubs*)GetComponent()->
                        GetSolid())->GetStartPhiAngle();

  G4double rmin = ((G4Tubs*)GetComponent()->GetSolid())->
                        GetInnerRadius();
  G4double rmax = ((G4Tubs*)GetComponent()->GetSolid())->
                        GetOuterRadius();

  G4double ghphi    = fHitPosition.phi();
  while (ghphi < 0)      ghphi += 2*M_PI;
  while (ghphi > 2*M_PI) ghphi -= 2*M_PI;
  G4double cellphi = ghphi - sphi;
  while(cellphi < 0)      cellphi += 2*M_PI;
  while(cellphi > 2*M_PI) cellphi -= 2*M_PI;
  G4int cellID     = (G4int)(cellphi/dphi);

  G4double rw   = (rmax + rmin )*0.5;
  G4double gwphi = (cellID+0.5) * dphi + sphi;

  G4double hwphi = cellphi - (cellID + 0.5)*dphi;

  G4double driftlen = rw * hwphi;
```

```
  G4ThreeVector pre  = GetPrePosition();
  G4ThreeVector post = GetPostPosition();

  // output hitdata to output file ....

  ofstream& ofs = GetOutputFileStream();
  if(! ofs.good()) {
    G4String errorMessage= "J4CDCDriftRegionHit::Output(): write error.";
    G4Exception(errorMessage);
  }
  else
  {
     ofs << setw(1) << layerNo << " " << wireNo << " " << setw(3) << cellID << " "
         << setw(5) << GetTrackID() << " " << GetMotherTrackID() << " " << GetHitNumber()
         << setw(6) << pdid << " " << setw(2) << GetCharge()
         << " " << dphi << " " << setw(10) << sphi << " "
         << setw(10) << gwphi << " " << setw(10) << rw << " " << setw(10) << driftlen << " "
         << pre.x() << " " << pre.y() << " " << pre.z() << " "
         << post.x() << " " << post.y() << " " << post.z() << " "
         << fHitPosition.x() << " " << fHitPosition.y() << " " << fHitPosition.z() << " "
         << GetMomentum().x() << " " << GetMomentum().y() << " "
         << GetMomentum().z() << " " << GetTotalEnergy() << " "
         << GetEnergyDeposit() << " " << GetTof() << G4endl;
  }
}
*******************************************************************************
```

2.4 Jupiter.cc の変更

以上のコーディングが済んだら、Jupiter.cc に検出器の登録を行います。Jupiter.cc では、各サブグループの最上位層の Component のインストールを行うことが出来ます。より正確には、Jupiter.cc で WorldVolume である Experimental Hall にインストールする Component を指定する、ということになります。したがって、ここでコントロール出来るのは、CDC や VTX といった検出器をまるごとインストールしたりアンインストールしたり、といった作業です。それより下位の構造については、各サブグループのコードで行って下さい。検出器の登録には、J4DetectorConstruction クラスの AddComponent() 関数を使います。

```
  J4DetectorConstruction* dtcptr = new J4DetectorConstruction;
   J4IR *irptr = new J4IR();
  irptr->SetMother(dtcptr->GetExpHall());
  dtcptr->AddComponent(irptr);
```

これらの設定は、RunManager が生成されたあと、Initialize() 関数が呼ばれるまでの間に行う必要があります (Initialize で Detector の Install が行われるので)。

一方、SensitiveDetector については、インストールした後個々に Switch を ON/OFF することが出来ます。SwitchOn/Off 関数は、デフォルトでは再帰的にその下の全ての階層の SensitiveDetector の Switch を ON/OFF しますが、引数に"me"などと入れてやると、対象の SensitiveDetector のみ Switch の切り替えを行います。この切り替えは、Detector がインストールされた後、すなわち RunManager の Initialize() 関数が呼ばれたあとに行います。

サンプルコードを以下に示します。

```
*******************************************************************************
```

```
// ====================================================================
//    Jupiter.cc
//
//    Exapmle program provided by JLC-CDC group.
//
//    NOTE:
//    Please contact me(hoshina@jlcux1.kek.jp) if any problems
//    or questions. DO NOT bother Geant4 developpers and Geant4 Users
//    Group Japan.
//
//                                K.Hoshina , 2001
// ====================================================================
#include <strstream.h>
#include "G4RunManager.hh"
#include "G4UIterminal.hh"
#include "G4UItcsh.hh"

#include "J4DetectorConstruction.hh"
#include "J4PhysicsList.hh"
#include "J4PrimaryGeneratorAction.hh"
#include "J4RunAction.hh"
#include "J4EventAction.hh"
#include "J4TrackingAction.hh"
#include "TBookKeeper.hh"

#ifdef G4VIS_USE
#include "J4VisManager.hh"
#endif

//#include "CLHEP/Random/Random.h"

#include "J4IR.hh"
#include "J4BD.hh"
#include "J4VTX.hh"
#include "J4IT.hh"
#include "J4CDC.hh"
#include "J4CAL.hh"

#define __IR__
#define __BD__
#define __VTX__
#define __IT__
#define __CDC__
#define __CAL__

TBookKeeper* TBookKeeper::fgBookKeeper = new TBookKeeper();

int main(int argc, char** argv)
{

  //-------------------------
  // Set random engine....
  //-------------------------
  // if you want to change a random engine, for example, ...

#if 0
  HepJamesRandom randomEngine;
  //RanecuEngine randomEngine;
  HepRandom::setTheEngine(&randomEngine);
#endif

  //-------------------------
  // Set run manager
  //-------------------------
```

```
  G4RunManager* runManager = new G4RunManager;  G4cout << endl;
//  runManager->SetVerboseLevel(2);

  //-------------------------------------------
  // set mandatory user initialization classes...

  //*-------------------------------------------
  //* Install detector components...
  //*-------------------------------------------

  J4DetectorConstruction* dtcptr = new J4DetectorConstruction;

  //* beam line

#ifdef __IR__
  J4IR *irptr = new J4IR();
  irptr->SetMother(dtcptr->GetExpHall());
  dtcptr->AddComponent(irptr);
#endif

#ifdef __BD__
  J4BD *bdptr = new J4BD();
  bdptr->SetMother(dtcptr->GetExpHall());
  dtcptr->AddComponent(bdptr);
#endif

  //* vtx
#ifdef __VTX__
  J4VTX *vtxptr = new J4VTX();
  vtxptr->SetMother(dtcptr->GetExpHall());
  dtcptr->AddComponent(vtxptr);
#endif

  //* intermediate tracker

#ifdef __IT__
  J4IT *itptr = new J4IT();
  itptr->SetMother(dtcptr->GetExpHall());
  dtcptr->AddComponent(itptr);
#endif

  //* cdc

#ifdef __CDC__
  J4CDC *cdcptr = new J4CDC();
  cdcptr->SetMother(dtcptr->GetExpHall());
  dtcptr->AddComponent(cdcptr);
#endif

  //* calorimeter

#ifdef __CAL__
  J4CAL *calptr = new J4CAL();
  calptr->SetMother(dtcptr->GetExpHall());
  dtcptr->AddComponent(calptr);
#endif

  //*-------------------------------------------
  //* Installation of detectors end
  //*-------------------------------------------

  runManager-> SetUserInitialization(dtcptr);

  // particles and physics processes
```

```
  runManager-> SetUserInitialization(new J4PhysicsList);

  //------------------------------------------
  // set mandatory user action class...

  //* primary generator

  J4PrimaryGeneratorAction* primarygenerator = new J4PrimaryGeneratorAction;

  runManager-> SetUserAction(primarygenerator);

  //* user action classes... (optional)

  runManager-> SetUserAction(new J4RunAction);
  runManager-> SetUserAction(new J4EventAction);
  runManager-> SetUserAction(new J4TrackingAction);

#ifdef G4VIS_USE
  // initialize visualization package
  G4VisManager* visManager= new J4VisManager;
  visManager-> Initialize();
  G4cout << endl;
#endif

  //-----------------------
  // user session start
  //-----------------------

  runManager-> Initialize();


  //*------------------------------------------
  //* SwitchOn/Off your detector...
  //*------------------------------------------
  //* If you wants call SwitchOn/Off recursively,
  //* call virtual function with resolver, as
  //* "J4VDetectorComponent::SwitchOn()".
  //* SwitchOn/(Off() requires option flag
  //* "recursive" or other strings, however,
  //* default value is set as "recursive".


  //* vtx
#ifdef __VTX__
  vtxptr->J4VDetectorComponent::SwitchOn();
#endif

  //* cdc

  // SwitchOn all SDs.
#ifdef __CDC__
  cdcptr->J4VDetectorComponent::SwitchOn();

  // Then, SwitchOff only SenseWires.
  for (G4int i=0; i<10; i++) {
     for (G4int j=0; j<5; j++) {
        strstream str;
        str << "ExpName:CDC:Layer0" << i << ":Cell:DriftRegion"
            << j << ":SenseWire" ;
        G4String name;
        str >> name;
        J4VDetectorComponent::GetComponentPtrByName(name)->SwitchOff();
     }
  }
#endif
```

```
  //*------------------------------------------
  //* Detector Switch end
  //*------------------------------------------


  if(argc==1) {                 // interactive session, if no arguments given

#ifdef USE_CSH_TERMINAL
    // csh-like
    G4UIterminal* session= new G4UIterminal();
    session-> SetPrompt("G4test(%s)[%/]:");
#else
    // tcsh-like
    G4UItcsh* tcsh= new
      G4UItcsh("ESC[36mJUPITERESC[33m(%s)ESC[32m[%/]ESC[36m[%h]ESC[37m:");
    G4UIterminal* session= new G4UIterminal(tcsh);
    tcsh-> SetLsColor(GREEN, CYAN);
#endif

    session-> SessionStart();
    delete session;

  } else {                      // batch mode

    G4UImanager* UImanager= G4UImanager::GetUIpointer();
    G4String command = "/control/execute ";
    G4String fileName = argv[1];
    UImanager-> ApplyCommand(command+fileName);

  }

  //-----------------------
  // terminating...
  //-----------------------

#ifdef G4VIS_USE
  delete visManager;
#endif

  delete runManager;  G4cout << endl;
  return 0;

}
*****************************************************************************
```

Appendix A

JUPITER Base Class リファレンス

A.1 J4VComponent

J4VComponent のデータメンバと関数は以下の通り。

A.1.1 データメンバ

fSubGroup	サブグループ名。
fName	実験名から始まるフルネーム。継承クラスが持つファーストネームとは異なるので注意。
fMother	自分自身がインストールされる親 Component へのポインタ。
fDaughters	自分の内部にインストールされる娘 Component へのポインタ配列。
fIsOn	Sensitive Detector に電源が入っているか否かを表す Flag。
fSolid	G4VSolid へのポインタ。
fLV	G4LogicalVolume へのポインタ。
fPV	G4PhysicalVolume へのポインタ。
fSD	J4VSensitiveDetector へのポインタ。
fNclones	同一の親ボリュームの中に置かれ、かつ同じクラスから生成されたオブジェクトのうち、同じ LogicalVolume を持つ Component が (実質的、仮想的を問わず) いくつ置かれるかを表す数。Component を Copy して置く場合には実際に Copy する数、Replica で仮想的に置く場合には Replicate される総数 (分割数) を表す。デフォルトは 1。
fNbrothers	同一の親ボリュームの中に置かれ、かつ同じクラスから生成されたオブジェクトのうち、形状の異なった (すなわち LogicalVolume の異なる) Component の種類の数。デフォルトは 1。
fMyID	fNbrothers が 2 以上の場合、そのうちの何番目かを表す数。デフォルトは 0。

fCopyNo　fNclones が 2 以上の場合で、かつ Copy として配置した場合のコピーナンバー。Copy の場合には、オリジナルも含め必ず 0 以上のコピーナンバーを与えなければならない。デフォルトは-1 で、Replica として置く場合及び fNclones が 1 のときには、この値を変えてはならない。

fCanDeleteUserLimits　LogicalVolume に対し定義する UserLimits の Owner Flag.

fMyDepth　World Volume から数えて、自分が何階層目にいるかを表す数。World を 0 として、1 階層下がるごとに-1 される。World 直下にインストールされた Component の階層は-1。

fgFamilyMembers　Static Member であるので、クラスに 1 つしかない。World 以下全ての Component を格納したリストである。

A.1.2　主なメンバ関数

コンストラクタ　親 Component へのポインタ、Copy もしくは Replicate される数の総数、形の違う兄弟オブジェクトの総数、兄弟のうちの何番目か、CopyNo を引数にとる。このとき、CopyNo は、Copy して配置する Component のオリジナルを作る場合に 0 番を指定する場合を除き、デフォルト (-1) のままにしておかなければならない。

コピーコンストラクタ　Copy を配置する場合のみ用いる。まず上のコンストラクタで CopyNo 0 番の Component を作り、これの参照と別の CopyNo をコピーコンストラクタに与えてコピーオブジェクトを作成する。CopyNo が重なってもチェックしてくれないので、重ならないように注意する。

operator=　J4VComponent のデータメンバをそっくりコピーする。オーナーフラッグをまだ作っていないので、うっかり Delete するとコピー元のオブジェクトが new した LogicalVolume なども消してしまうので、注意。

InstallIn　自分自身を親オブジェクトにインストールするための関数。pure viertual 関数。