Object-Oriented Design and Implementation

CSC2000 - Marathon

Makoto ASAI

Hiroshima Institute of Technology

(Geant4 / ATLAS)

Makoto.Asai@cern.ch

Contents

Chapter 1

Introduction

Motivation of this lecture

- If you are writing your code which is exclusively used by yourself and it will be used within a temporary short duration, you can ignore this lecture.
- But you are developing your code with your colleagues and/or your code will be used by your collaborators for years, you should be aware of “good software”.

Motivation of this lecture

- Good software is
  - Easy to understand the structure
  - Easy to find/localize/fix a bug
  - Easy to change one part without affecting to other parts
  - Well modularized and reusable
  - Easy to maintain and upgrade
  - etc. etc.
- Object-Orientation is a paradigm which helps you to make a good software.

Motivation of this lecture

- Use of so-called “Object-Oriented language” such as C++ or Java does not guarantee the Object-Oriented Programming.
  - Badly written C++/Java code is worse than badly written Fortran code.
- Well designed, Object-Oriented good software can be relatively easily implemented by using Object-Oriented language.
  - Language is a tool to realize Object-Orientation.

Motivation of this lecture

- In this lecture I will show you some basic concepts of Object-Oriented Programming.
- These concepts are more important than the detailed syntaxes of a language and they will guide you to learn C++/Java as a language which stands on Object-Orientation.

Object-Oriented Programming

- Object-Oriented Programming (OOP) is the programming methodology of choice in the 1990s.
- OOP is the product of 30 years of programming practice and experience.
  - Simula67
  - Smalltalk, Lisp, Clu, Actor, Eiffel, Objective C
  - and C++, Java
- OOP is a programming style that captures the behavior of the real world in a way that hides detailed implementation.

Fundamental Ideas

- When successful, OOP allows the problem solver to think in terms of the problem domain.
  - Requirements document
  - Object-Oriented Analysis and Design (OOA&D)
  - CASE tools
- Three fundamental ideas characterize Object-Oriented Programming.
  - Class/Object, Encapsulation
  - Class hierarchies, Inheritance
  - Abstraction, Polymorphism

Chapter 2

Class/Object and Encapsulation

Class and Object

- Object-Oriented Programming (OOP) is a data-centered view of programming in which data and behavior are strongly linked.
- Data and behavior are conceived of as classes whose instances are objects.
- OOP also views computation as simulating behavior. What is simulated are objects represented by a computational abstraction.

Abstract Data Type

- The term abstract data type (ADT) means a user-defined extension to the native types available in the language.
- ADT consists of
  - a set of values
  - a collection of operators and methods that can act on those values

Abstract Data Type

- Class objects are class variables. OOP allows ADT to be easily created and used.
  - For example, integer objects, floating point number objects, complex number objects, four momentum objects, etc., all understand addition and each type has its own code of executing addition.

```
FourMomentum a, b, c;

c = a + b;
```

- An ADT object can be used in exactly same manner as a variable of native type. This feature increases the readability of the code.

Abstract Data Type

- In OOP, classes are responsible for their behavior.

```
class FourMomentum
{
  public:
    FourMomentum(double px, double py, double pz, double e);
    ~FourMomentum();
  public:
    FourMomentum& operator = (const FourMomentum & right);
    FourMomentum operator + (const ThreeMomentum & right);
                ....
```

Encapsulation

- Encapsulation consists of
  - the internal implementation details of a specific type
  - the externally available operators and functions that can act on objects of that type
- The implementation details should be inaccessible to client code that uses the type.
- Make data members private and provide public Set/Get methods accessible to them.
- Make all Get and other methods which do not modify any data member “cosnt”.
  - “const” methods can be accessed even for constant ADT objects.
  - Strict use of constant ADT objects allows you the safe programming.

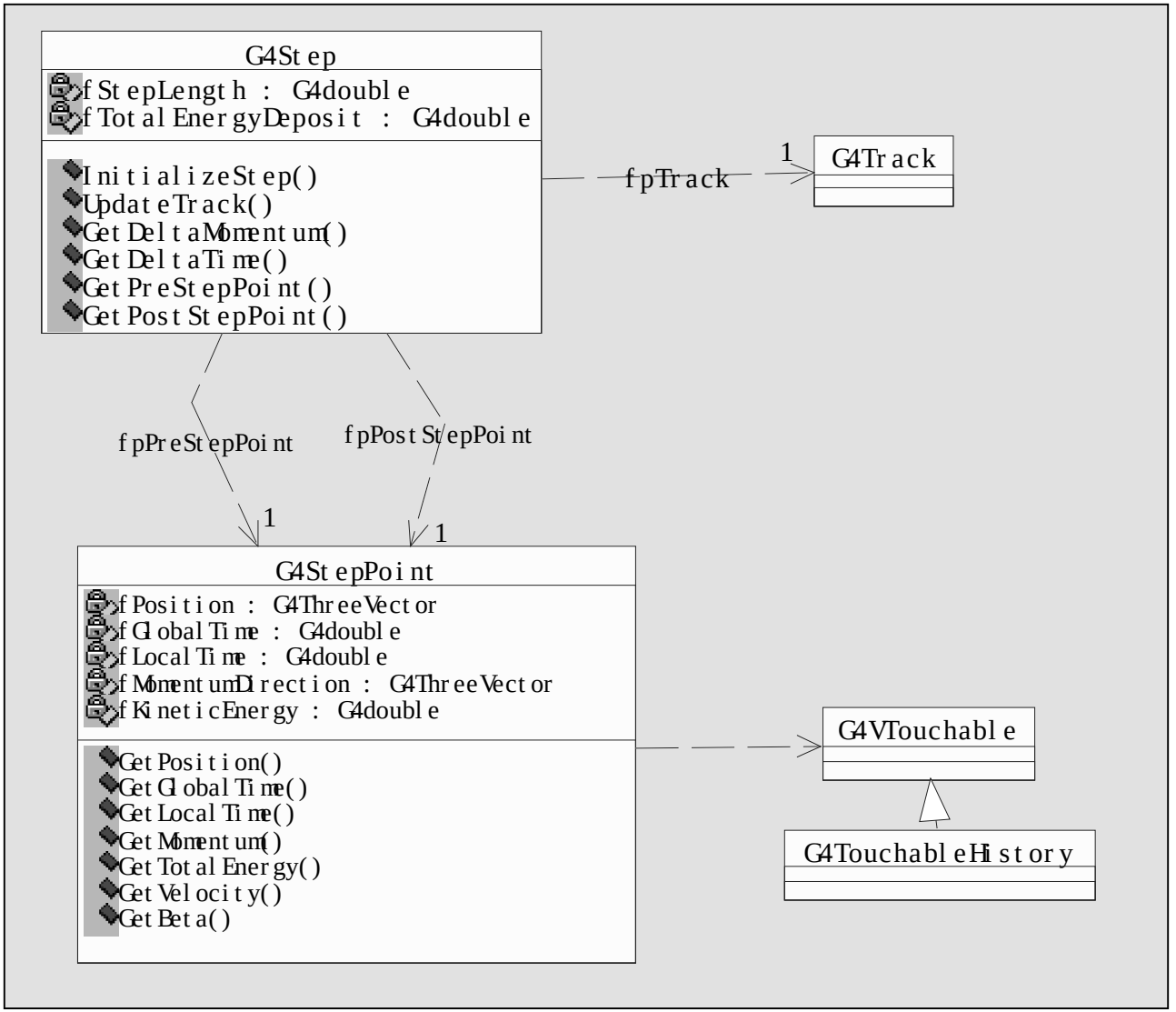
Encapsulation

- Changes of the internal implementation should not affect on how to use that type externally.

```
class FourMomentum
{
   …
 private:
   double m_Px;
   double m_Py;
   double m_Pz;
   double m_E;
 public:
   void SetP(double p);
   double GetP() const;
....
```

```
class FourMomentum
{
   …
 private:
   double m_P;
   double m_Theta;
   double m_Phi;
   double m_E;
 public:
   void SetP(double p);
   double GetP() const;
....
```

G4Step and G4StepPoint



Chapter 3

Class hierarchies and Inheritance

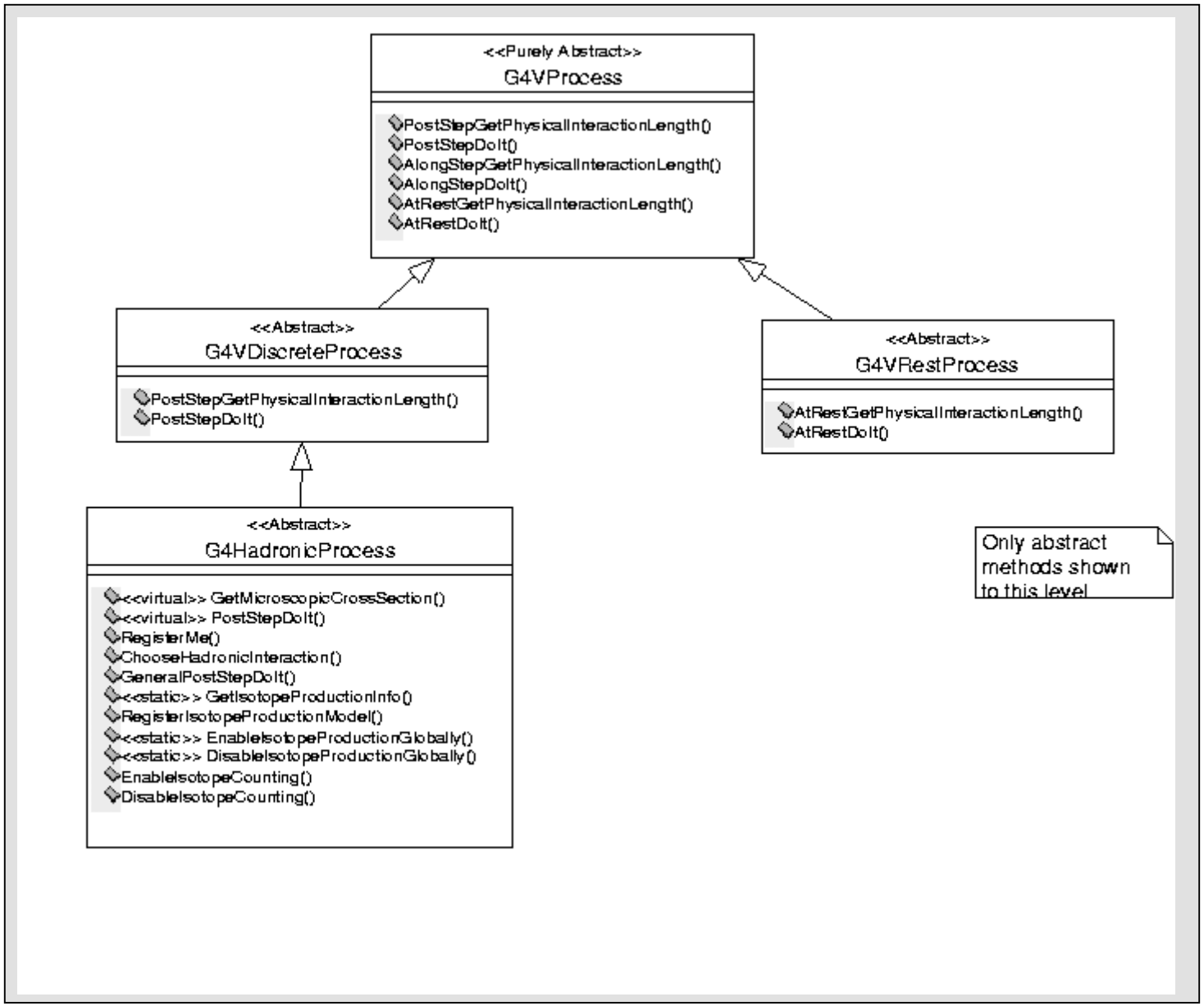
Class hierarchies and Inheritance

- Inheritance is a mean of deriving a new class from existing classes, called base classes. The newly derived class uses existing codes of its base classes.
- Through inheritance, a hierarchy of related types can be created that share codes and interfaces.
- A derived class inherits the description of its base class. Inheritance is a method for copying with complexity.

Class hierarchies and Inheritance

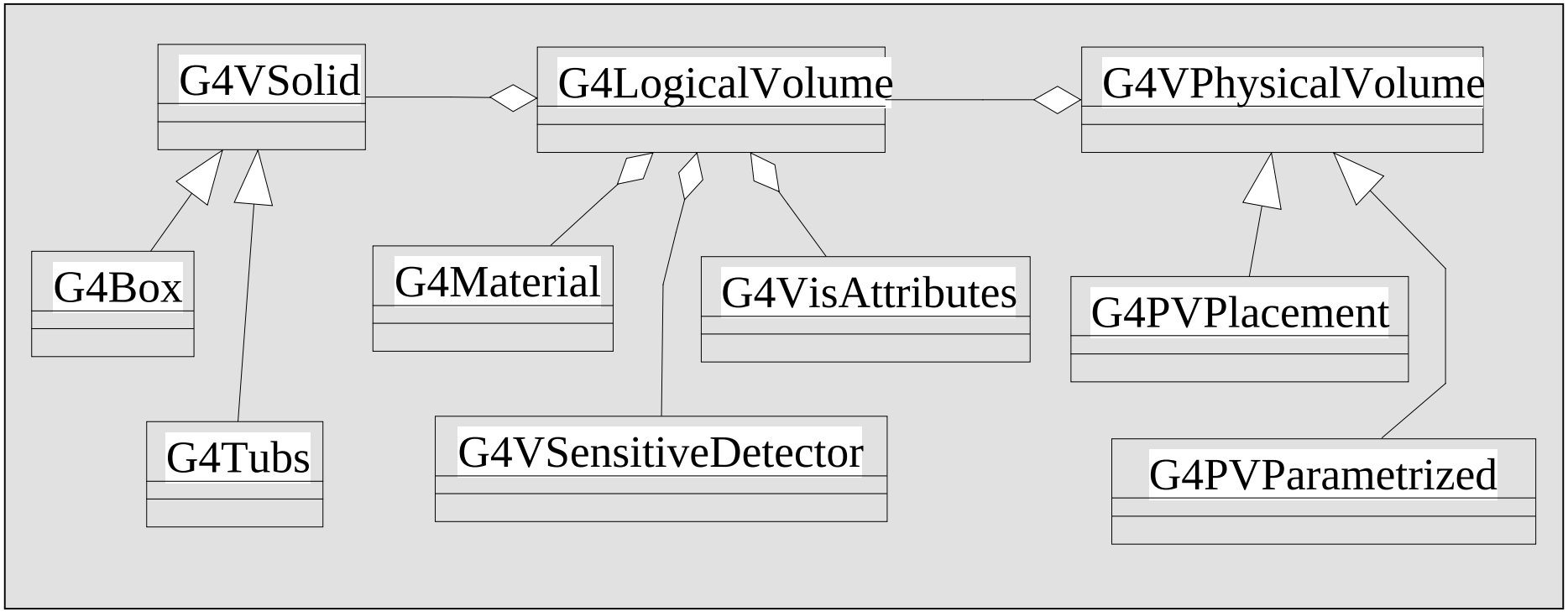
- It is better to avoid protected data members.
  - Make data members in a base class private and provide protected non-virtual access methods to them.
- Avoid unnecessary deep hierarchies.
  - Should a trajectory class and a detector volume class be derived from a single base class, even though both of them have a “Draw()” method?
  - Follow the naïve concepts everyone can easily understand.

Class hierarchies and Inheritance



Class hierarchies and Inheritance

- Avoid unnecessary multiple inheritance.
  - In many cases, delegation can solve the problem.



Comments on Collection

- Type-unsafe collection is quite dangerous.
  - C++ case, pointer collection of void or very bogus base class
  - Java case, default vector collection of “Object” base class
- Type-unsafe collection easily reproduces the terrible difficulties we experienced with the Fortran common block.

Chapter 4

Abstraction and Polymorphism

Rapid Prototyping

- Abstraction and Polymorphism enable "Rapid Prototyping".
  - High level class diagrams and scenario diagrams should be made first before going to the detailed design/implementation of actual concrete classes.
  - "Proof of concepts" demonstration must be done with just a couple of concrete classes (or just one dummy concrete class) for each abstract base class.

Abstraction and Polymorphism

- Abstraction and polymorphism localizes responsibility for an abstracted behavior.
- They also help the modularity and portability of the code.
  - For example, Geant4 is free from the choice of histogramming and persistency techniques. Also, GUI and visualization are completely isolated from Geant4 kernel via the abstract interfaces.

Polymorphism

- Polymorphism has lots of forms.
  - Function and operator overloading
  - Function overriding
  - Parametric polymorphism

Refer A.Johnson's lecture for dynamic class loading featured in Java.

Operator Overloading

- In C++, an operator is overloadable . A function or an operator is called according to its signature, which is the list of argument types.
  - If the arguments to the addition operator are integral, then integer addition is used. However, if one or both arguments are floating point, then floating point addition is used.
- Operator overloading helps the readability.

```
double p, q, r;

r = p + q;

FourMomentum a, b, c;

c = a + b;
```

Function Overriding

- Using virtual member functions in an inheritance hierarchy allows run-time selection of the appropriate member function. Such functions can have different implementations that are invoked by a run-time determination of the subtype (virtual method invocation, dynamic binding).

```
G4VHit* aHit;
for(int i = 0; i < hitCol->entries(); i++)
{
  aHit = (*hitCol)[i];
  aHit->Draw();
}
```

Function Overloading

- Functions of same name are distinguished by signatures.
- For the case of function overloading of “non-pure virtual” virtual functions, all (or none) of them should be overridden.
  - Overriding “overrides” overloading!!!
  - Intrinsic source of a bug even though compiler warns.
  - You will see this warning for G4VParameterisedVolume…

```
class Base {
  public:
    Base() {;}
    virtual void Show(int i) { cout << "Int" << endl; }
    virtual void Show(double x) { cout << "Double" << endl; }
}
Class Derived : public Base {
  public:
    Derived() {;}
    virtual void Show(int i) { cout << "Int" << endl; }
}
main() {
  Base* a = new Derived();
  a->Show(1.0);
}
```

➡ Gives "Int"!!!

Template

- C++ also has parametric polymorphism, where type is left unspecified and is later instantiated.
- STL (Standard Template Library) helps a lot for easy code development.

CSCG4ExEmCalorimeter

```
class CSCG4ExEmCalorimeterHit : public G4VHit
{
  public:
     CSCG4ExEmCalorimeterHit();
     CSCG4ExEmCalorimeterHit(G4int z);
     virtual ~CSCG4ExEmCalorimeterHit();
     const CSCG4ExEmCalorimeterHit& operator=(const
     CSCG4ExEmCalorimeterHit &right);
     virtual void Draw();
     virtual void Print();
       ……
};

typedef G4THitsCollection<CSCG4ExEmCalorimeterHit>
     CSCG4ExEmCalorimeterHitsCollection;
```

Chapter 5

Unified Software Development Process

Software Development Process

- A software development process is the set of activities needed to transform a user's requirements to a software system.
- The Unified Software Development Process is a software development process which is characterized by
  - Use-case driven
  - Architecture centered
  - Iterative and incremental

Requirements

- There are many different types of requirements.
  - Functional requirements
  - Data requirements
  - Performance requirements
  - Capacity requirements
  - Accuracy requirements
  - Test/Robustness requirements
  - Maintainability, extensibility, portability, etc., “ability” requirements
- Requirements drives use-cases and architectures.

Use-case

- A software system should be used by the users. Thus the developers of the system must know the users' needs.
- The term user refers not only to human users but also to other system which interacts with the system being developed.
- An interaction from/to the user is a use-case. A use-case is a piece of functionality in the system which captures a requirement.

Architecture

- The role of software architecture is similar in nature to the role of architecture plays in building construction.
  - A plan of building is looked at from various viewpoints, such as structure, services, heat conduction, electricity. This allows the builder to see a complete picture before actual construction.
  - The software architecture must be influenced by the requirements of both use-case dependent and use-case independent.
- Architecture is not a framework.

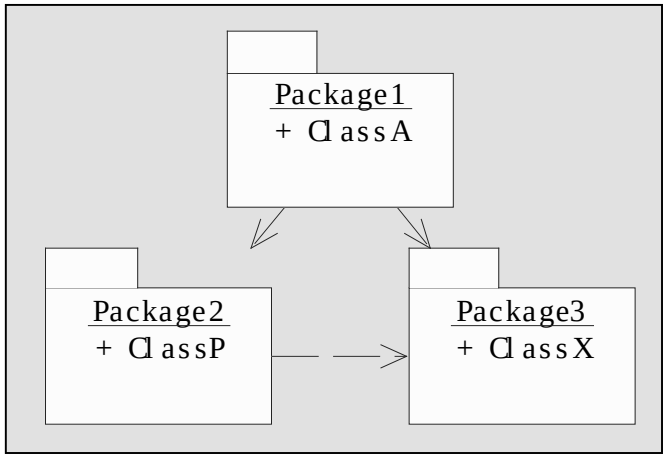
Major UML diagrams

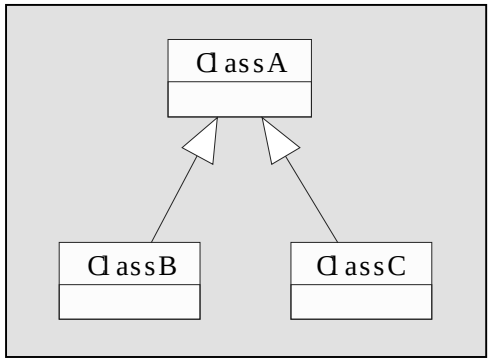
- Dynamic diagrams
  - Use-case diagram
  - Scenario (sequence) diagram
  - State diagram
- Static diagrams
  - Domain Model diagram
  - Class diagram

Refer R.Jones' lecture for details of UML.

Domain Model diagram

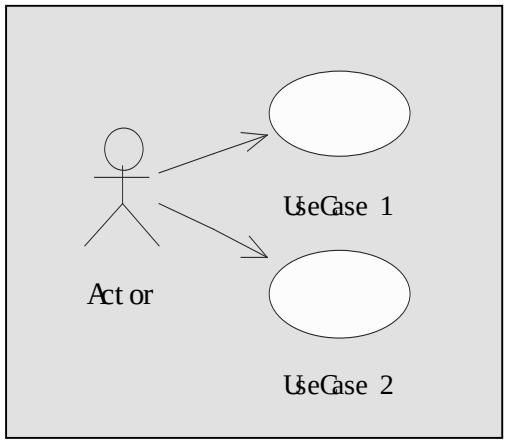
- The term “problem domain” refers to the area which encompasses real-world things and concepts related to the problem that the system is being designed to solve.
- Domain modeling is a task of discovering objects (classes) that represent those things and concepts.





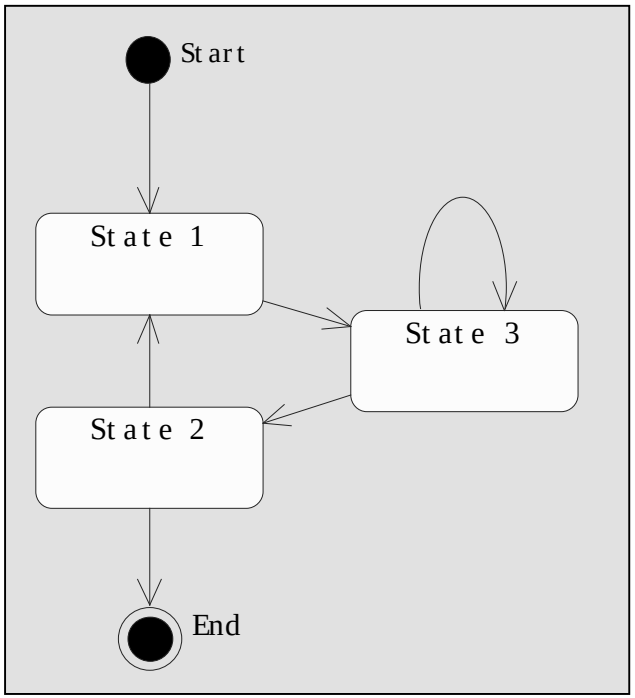
Use-case diagram

- Major use-cases can be found in the user's functional requirements.
- Two courses of use-cases must be designed simultaneously.
  - Basic courses
  - Alternative courses



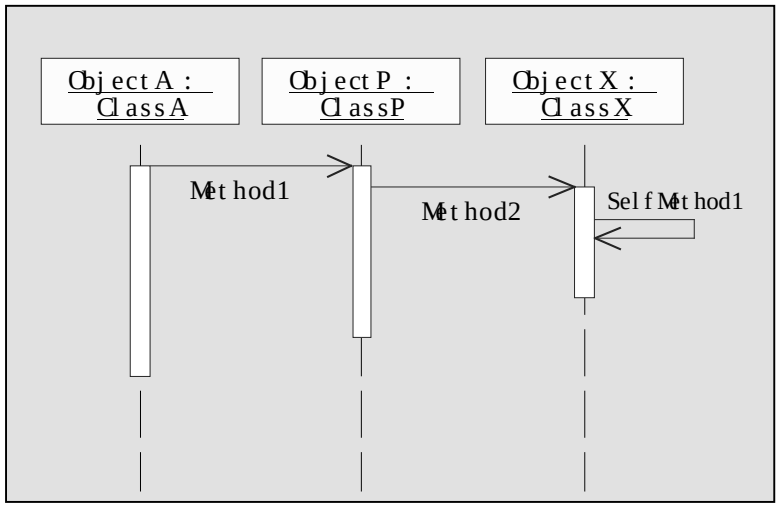
State diagram

- State diagram captures the lifecycle of objects.
- This cycle is expressed in terms of the different states that the objects can assume, and the events that cause state changes.

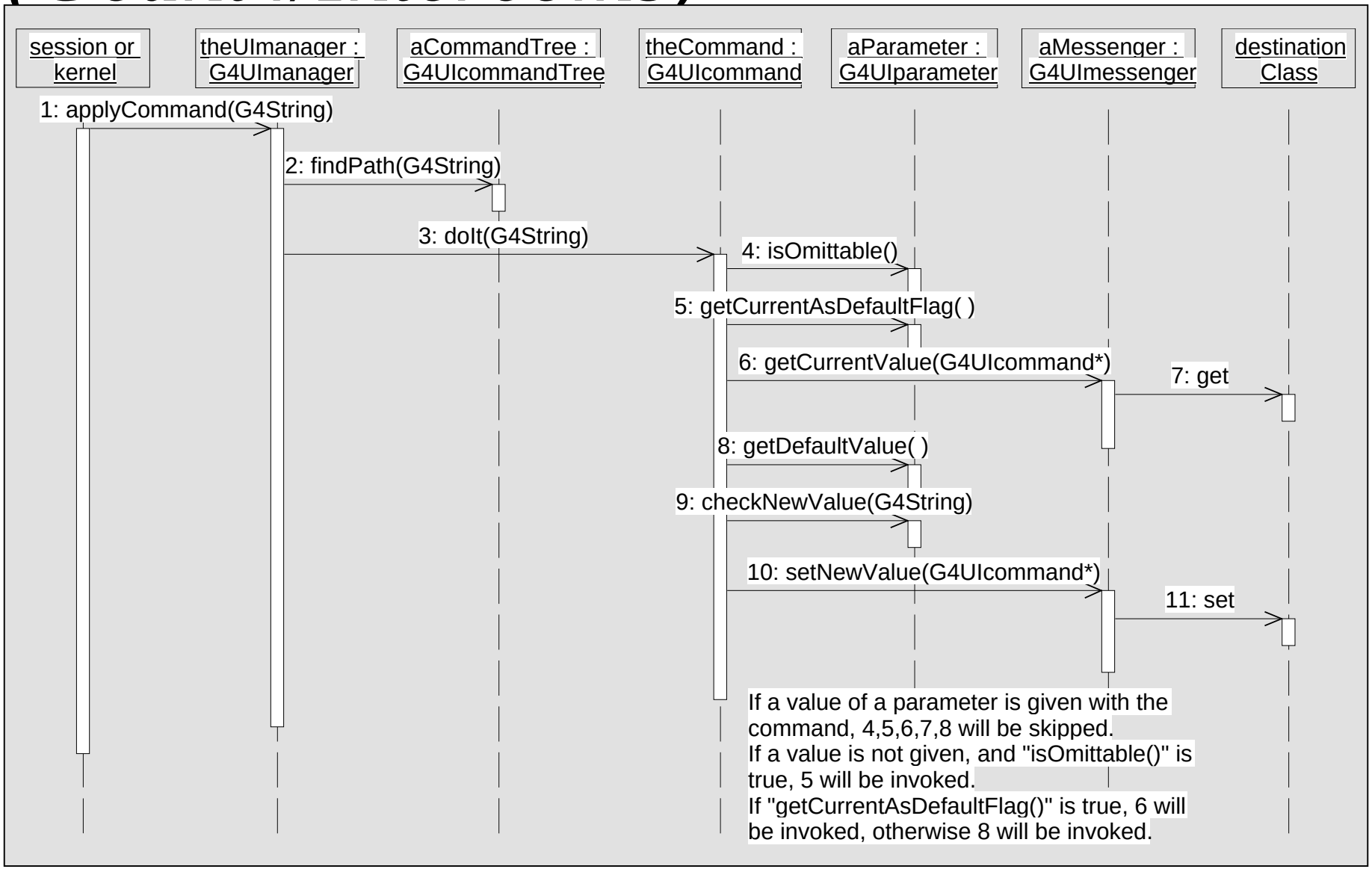


Scenario (sequence) diagram

- A scenario diagram should be prepared for each use-case.
- Methods necessary for a class are found with writing this diagram.

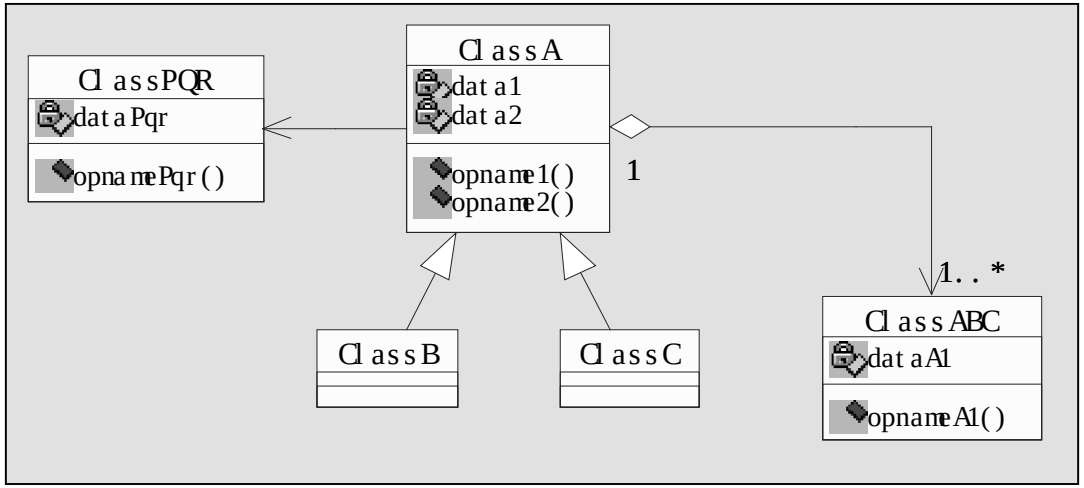


Scenario diagram (Geant4/Intercoms)

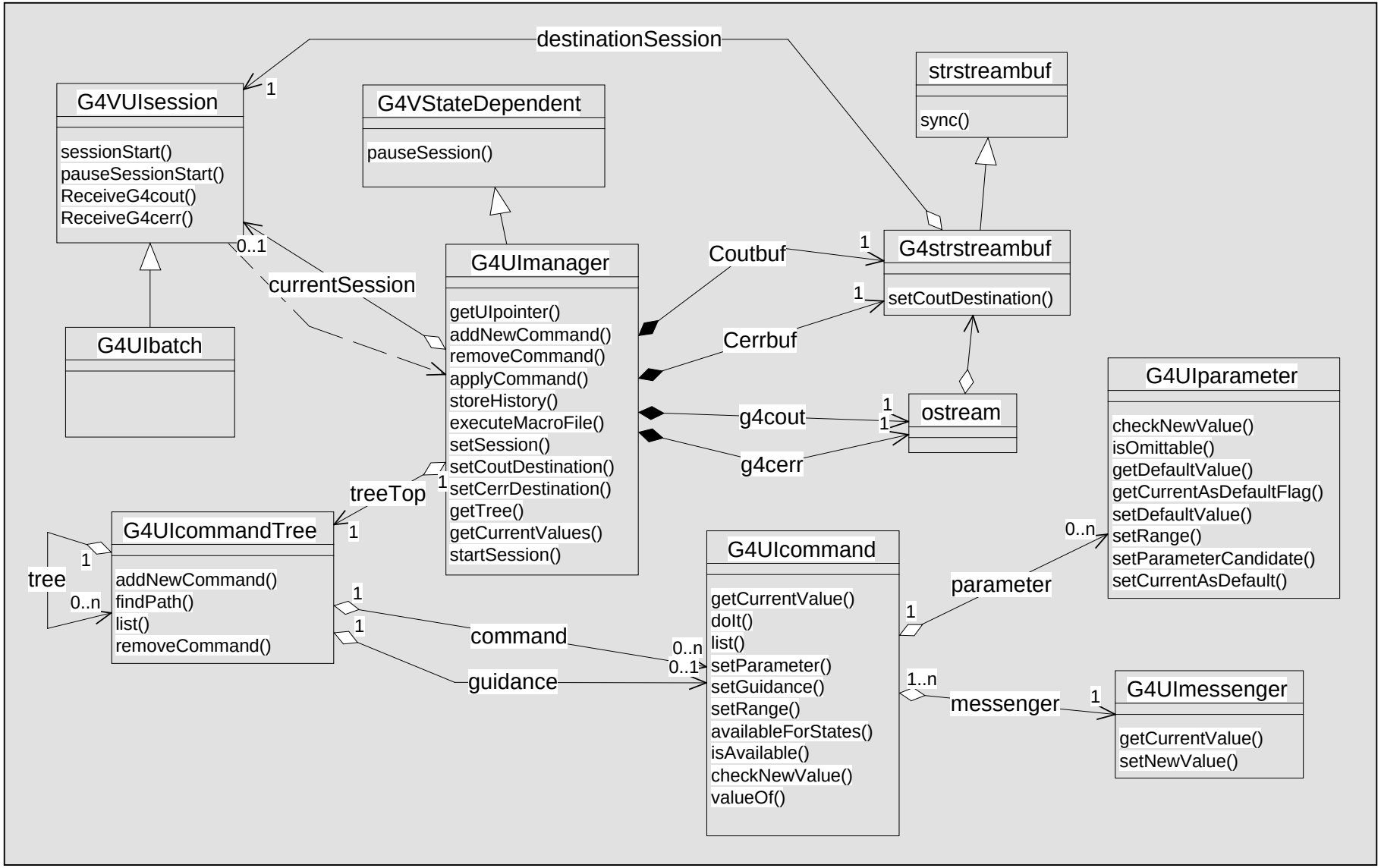


Class diagram

- Domain Model diagram is upgraded to class diagram by adding data members, methods, multiplicities, etc.



Class diagram (Geant4/Intercoms)



Spiral approach

- Four steps
  - Object-Oriented Analysis
  - Object-Oriented Design
  - Implementation
  - Test
- Repeat these steps several turns to make a software products.
- Don’t hesitate to update diagrams in earlier cycles.