A priori, we see no reason why moving to a language which supports the idea of objects, such as C++, should change the way we think of doing physics analysis.
Why Object-Oriented Programming?

Procedural versus OO Programming

HEP Programming

Programming Paradigms in HEP

Orthogonality

Open-Closed Principle

Liskov Substitution Principle

Dependency-Inversion Principle

Summary

References

Procedural vs. OO Programming, from [?]

The procedural paradigm

The oo paradigm

Top-Down

Bottom-Up
### A History of Code

<table>
<thead>
<tr>
<th></th>
<th>lines of code / 1 loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>JADE</td>
<td>$\sigma(10-100)k$</td>
</tr>
<tr>
<td>OPAL</td>
<td>$\sigma(100)k$</td>
</tr>
<tr>
<td>ATLAS</td>
<td>$\sigma(1)M$</td>
</tr>
</tbody>
</table>

- experiments size and complexity increases
- experiments analysis software size and complexity increases
- **We need tools that deal with this complexity!**

---

### Programming Paradigms in HEP

**physics is about ...**

- modelling nature
- objects interact according to laws of nature
  - fields, particles, atoms, molecules, solid states, liquids

**object-oriented programming is about ...**

- objects and interactions
  - a way of thinking about software well adapted to physics

**object-oriented analysis and design ...**

- is a software engineering practice
- manages large projects professionally
Orthogonality

Definition
A **Responsibility** of a class is defined as *a reason for the class to change*.

Exercise 1
How many responsibilities do classes a) and b) have?

Definition
**Orthogonality** ([2]) of a system of classes can be defined as the degree of how many classes have independent or non-overlapping **responsibilities**.

Single-Responsibility Principle

Theorem (from [5])
A class should only have **one** reason to change, i.e. try to create systems with high orthogonality.

Looking back at Exercise 1 a)

<table>
<thead>
<tr>
<th>before</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Modem</strong></td>
<td><strong>&lt;interface&gt; DataChannel</strong></td>
</tr>
<tr>
<td>+ dial(phoneNumber : String)</td>
<td>+ send(aCharacter : char)</td>
</tr>
<tr>
<td>+ hangup()</td>
<td>+ receive() : char</td>
</tr>
<tr>
<td>+ send(aCharacter : char)</td>
<td>+ &lt;interface&gt; Connection</td>
</tr>
<tr>
<td>+ receive() : char</td>
<td>+ dial(phoneNumber : String)</td>
</tr>
<tr>
<td></td>
<td>+ hangup()</td>
</tr>
</tbody>
</table>

P. Steinbach (IKTP)
Class Design Principles
March 10th, 2011
The Open-Closed Principle

Theorem (from [5])

Software Entities (classes, modules, functions, etc) should be open for extension, but closed for modification.

Open

- the behavior of an entity can be extended
- as requirements of a system change (that’s a fact!), the entities behavior can be extended or modified to satisfy these changes

Closed

- extension of behavior does NOT result in changing the source code
- the binary executable version of a given entity remains untouched

Exercise 2

The above is way too complicated for one slide! Let’s have a look at Exercise 2!

P. Steinbach (IKTP)  
Class Design Principles  
March 10th, 2011 9 / 21

Reviewed: Open-Closed Principle

The Square/Circle Problem

- rigid: adding triangle requires Shape, Square, Circle, DrawAllShapes to be recompiled and redeployed
- fragile: switch/case will be required by all client classes that use Shapes
- immobile: reusing DrawAllShapes is impossible without including Shape, Square, Circle as well

Solution: Using Abstraction

```cpp
struct Shape {
    virtual void Draw() const = 0;
};

struct Square {
    virtual void Draw() const;
};

void DrawAllShapes(
    const std::vector<Shape*>& list) {
    std::vector<Shape*>::const_iterator itr;
    for(itr=list.begin();itr!=list.end(); ++itr) {
        itr->Draw();
    }
}
```

P. Steinbach (IKTP)  
Class Design Principles  
March 10th, 2011 11 / 21
But hold on ...

- did the abstraction from above close DrawAllShapes against all changes?
  - No, there is no model of abstraction that is natural to all contexts!
  - closure can never be complete, only strategic
- how to deal with possible changes?
  1. derive possible changes from software requirements
  2. implement necessary abstractions
  3. wait!

To Summarize

- conforming to the open-closed principle yields greatest benefits of OOP (flexibility, reusability, maintainability)
- apply abstraction to parts of software that exhibit frequent change
- Resisting premature abstraction is as important as abstraction itself.

Liskov Substitution Principle

The Liskov Substitution Principle

Theorem (paraphrased from [4])

Subtypes must be substitutable for their base types.

Exercise 3

Try to answer question 3 a) and b)!
Observations from Exercise 3

- Violations of Liskov Substitution Principle result in Run-Time Type Information to be used
  - violates the Open-Closed Principle
- an (inheritance) model can never be validated in isolation
  - but rather with its use (users) in mind
  - Is-A relationship within inheritance refers to behavior that can be assumed or that clients depend upon.
- how to ensure/enforce Liskov Substitution Principle?
  - Design-by-Contract
  - in C++: only by assertions or Unit Tests

Summary

- this principle ensures: maintainability, reusability, robustness
- Liskov Substitution Principle enables the Open-Closed Principle
- the contract of a base type has to be well understood, if not even enforced by the code

The Dependency-Inversion Principle

Theorem (from [5])

1. High level modules should not depend upon low level modules. Both should depend upon abstractions.
2. Abstractions should not depend upon details. Details should depend upon abstractions.

Exercise 4

Please complete 4 a)!
Observations: The Dependency-Inversion Principle

Exercise 4 continued

1. The vendor of Lamp changes its definition. All methods containing Turn are renamed to Ramp! Face your design with that!
2. Look at Button: Can it be reused for classes of type Signal?

Exercise 4: A Solution

Naive Ansatz

```
Button
+ poll()
```

```
Lamp
+ TurnOn()
+ TurnOff()
```

Inverted Dependency

```
Button
+ poll()
```

```
interface ButtonServer

Lamp
+ TurnOn()
+ TurnOff()
```

Dynamic and Static Polymorphism

in C++, both can help to invert dependencies

Dynamic Polymorphism through Abstract Interfaces

```
Button
+ poll()
```

```
interface ButtonServer

Lamp
+ TurnOn()
+ TurnOff()
```

Static Polymorphism through template classes

```
template <class TurnableObject>
class Button {
    TurnableObject* itsTurnable;

    public:
    Button(TurnableObject* _object = 0):
        itsTurnable(_object) {};

    void poll() {
        if(/*some condition*/) {
            itsTurnable.TurnOn();
            itsturnable.TurnOff();
        }
    }
};
```

- compile-time polymorphism
- design-by-policy, see [1]
Summary

- dependency of policies on details is natural to procedural design
- inversion of dependencies is hallmark of (good) object-oriented design
- Dependency-Inversion Principle is at the heart of reusable frameworks (no matter what size)
- enables the Open-Closed Principle

What is left to say ...

did not cover:
  - module design principles
  - clean code principles
  - useful coding conventions

What I tried to say ...

- although having a slow learning curve, OOP can help do highly-sophisticated physics analysis
- learning OO Class Design prevents sleepless nights of debugging or copy-and-past’ing
- Coding may not be our profession, but we do it everyday anyhow, so we better know our craft!
References

*Modern C++ Design: Generic Programming and Design Patterns Applied.*  

*The Pragmatic Programmer.*  
pragmaticprogrammer.com.

Class design principles.  
Terascale Workshop on Advanced Methods of Software Development 2010.

*Gaudi User Guide.*  
cern.ch/prof-gaudi.

Keynote address - data abstraction and hierarchy.  

*Agile Software Development.*  
Class Design Principles at Author’s Homepage.