

From the Gaudi User Guide, [4]

*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.*

# Class Design Principles in Object-Oriented Programming

Wolfgang F. Mader, **Peter Steinbach**

Institute for Nuclear and Particle Physics, TU Dresden

March 10th, 2011



TECHNISCHE  
UNIVERSITÄT  
DRESDEN



# Outline

## Why Object-Oriented Programming?

- Procedural versus OO Programming

- HEP Programming

- Programming Paradigms in HEP

## Orthogonality

## Open-Closed Principle

## Liskov Substitution Principle

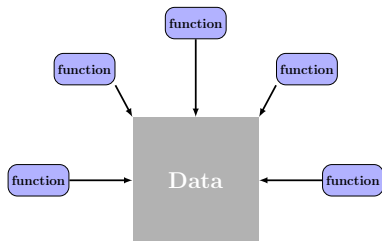
## Dependency-Inversion Principle

## Summary

## References

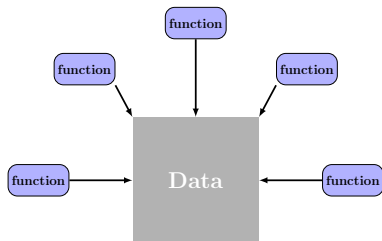
## Procedural vs. OO Programming, from [3]

## the procedural paradigm



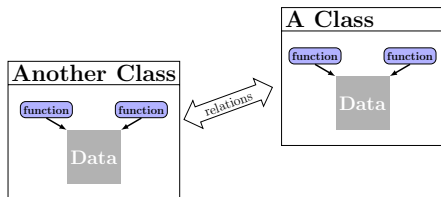
## Procedural vs. OO Programming, from [3]

## the procedural paradigm



Top-Down

## the oo paradigm



Bottom-Up

## Hep Software Sizes

### A History of Code

	lines of code / 1 loc
JADE	$o(10 - 100)k$
OPAL	$o(100)k$
ATLAS	$o(1)M$

## Hep Software Sizes

### A History of Code

	lines of code / 1 loc
JADE	$o(10 - 100)k$
OPAL	$o(100)k$
ATLAS	$o(1)M$

- ▶ experiments size and complexity increases

## Hep Software Sizes

### A History of Code

	lines of code / 1 loc
JADE	$o(10 - 100)k$
OPAL	$o(100)k$
ATLAS	$o(1)M$

- ▶ experiments size and complexity increases
- ▶ experiments analysis software size and complexity increases



## Hep Software Sizes

## A History of Code

	lines of code / 1 loc
JADE	$o(10 - 100)k$
OPAL	$o(100)k$
ATLAS	$o(1)M$

- ▶ 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

## 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

## 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

## 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*.

# 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?

## 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 [6])

*A class should only have **one** reason to change, i.e. try to create systems with high orthogonality.*

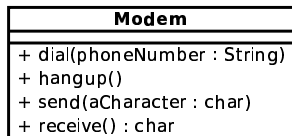
## Single-Responsibility Principle

### Theorem (from [6])

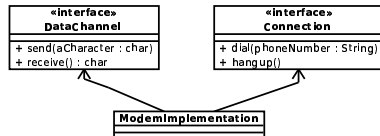
*A class should only have **one** reason to change, i.e. try to create systems with high orthogonality.*

### Looking back at Exercise 1 a)

before



after



## The Open-Closed Principle

### Theorem (from [6])

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

## The Open-Closed Principle

### Theorem (from [6])

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

Open

Closed

## The Open-Closed Principle

### Theorem (from [6])

*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

## The Open-Closed Principle

### Theorem (from [6])

*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**

## The Open-Closed Principle

### Theorem (from [6])

*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!**

## 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



## 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

```
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();  
    }  
}
```

## Summary: The Open-Closed Principle

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

## Summary: The Open-Closed Principle

### 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!

## Summary: The Open-Closed Principle

### 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.**

## The Liskov Substitution Principle

Theorem (paraphrased from [5])

*Subtypes must be substitutable for their base types.*

## The Liskov Substitution Principle

Theorem (paraphrased from [5])

*Subtypes must be substitutable for their base types.*

### Exercise 3

Try to answer question 3 a) and b) !

## Review & Summary: The Liskov Substitution Principle

### Observations from Exercise 3

## Review & Summary: The Liskov Substitution Principle

### Observations from Exercise 3

- ▶ Violations of Liskov Substitution Principle result in Run-Time Type Information to be used
  - ▶ violates the Open-Closed Principle



## Review & Summary: The Liskov Substitution Principle

### 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**.

## Review & Summary: The Liskov Substitution Principle

### 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

## Review & Summary: The Liskov Substitution Principle

### 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 [6])

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.*

## The Dependency-Inversion Principle

### Theorem (from [6])

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 it's definition. All methods containing `Turn` are renamed to `Ramp`! Face your design with that!

## Observations: The Dependency-Inversion Principle

### Exercise 4 continued

1. The vendor of `Lamp` changes it's 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`?

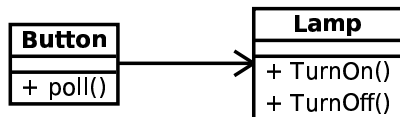
## Observations: The Dependency-Inversion Principle

### Exercise 4 continued

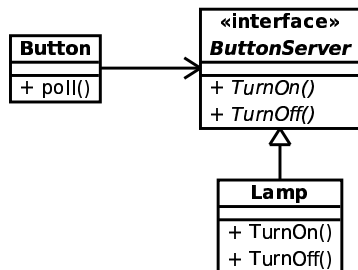
1. The vendor of Lamp changes it's definition. All methods containing Turn are renamed to Ramp! Face your design with that!
2. Look at Button: Can it be reused for classes of type Signal?

### Exercise 4: A Solution

#### Naive Ansatz



#### Inverted Dependency



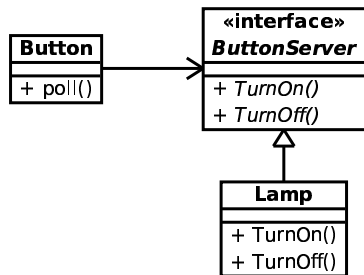


## Review: The Dependency-Inversion Principle

### Dynamic and Static Polymorphism

in C++, both can help to invert dependencies

#### Dynamic Polymorphism through Abstract Interfaces

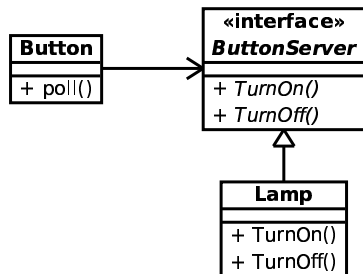


## Review: The Dependency-Inversion Principle

### Dynamic and Static Polymorphism

in C++, both can help to invert dependencies

#### Dynamic Polymorphism through Abstract Interfaces



#### 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();
    }
};
  
```

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

## Summary: The Dependency-Inversion Principle

### 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

## Summary

### What is left to say ...

did not cover:

- ▶ module design principles
- ▶ clean code principles
- ▶ useful coding conventions

## Summary

### 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

- [1] Andrei Alexandrescu.  
*Modern C++ Design: Generic Programming and Design Patterns Applied.*  
Addison-Wesley Professional, 2001.
- [2] Andrew Hunt and David Thomas.  
*The Pragmatic Programmer.*  
Addison Wesley, 2000.  
[pragmaticprogrammer.com](http://pragmaticprogrammer.com).
- [3] Stefan Kluth.  
Class design principles.  
Terascale Workshop on Advanced Methods of Software Development 2010.
- [4] LHCb Collaboration.  
*Gaudi User Guide.*  
[cern.ch/prof-gaudi](http://cern.ch/prof-gaudi).
- [5] Barbara Liskov.  
Keynote address - data abstraction and hierarchy.  
*SIGPLAN Not.*, 23:17–34, January 1987.
- [6] Robert C. Martin, James W. Newkirk, and Robert S. Koss.  
*Agile Software Development.*  
Prentice Hall, 2003.  
Class Design Principles at Author's Homepage.