# Command in C++

Back to **Command** description

## Command design pattern

1. Create a class that encapsulates some number of the following:
   - a "receiver" object
   - the method to invoke
   - the arguments to pass
2. Instantiate an object for each "callback"
3. Pass each object to its future "sender"
4. When the sender is ready to callback to the receiver, it calls `execute()`

```
#include <iostream>  #include <string>  using namespace std;
class Person;

class Command
{
    // 1. Create a class that encapsulates an object and a member function
    // a pointer to a member function (the attribute's name is "method")
    Person *object; //
    void(Person:: *method)();
  public:
    Command(Person *obj = 0, void(Person:: *meth)() = 0)
    {
        object = obj; // the argument's name is "meth"
        method = meth;
    }
    void execute()
    {
        (object-> *method)(); // invoke the method on the object
    }
};

class Person
{
    string name;

    // cmd is a "black box", it is a method invocation
    // promoted to "full object status"
    Command cmd;
  public:
    Person(string n, Command c): cmd(c)
    {
        name = n;
    }
    void talk()
    {
        // "this" is the sender, cmd has the receiver
        cout << name << " is talking" << endl;
        cmd.execute(); // ask the "black box" to callback the receiver
    }
    void passOn()
    {
        cout << name << " is passing on" << endl;

        // 4. When the sender is ready to callback to the receiver,
        // it calls execute()
        cmd.execute();
    }
    void gossip()
```

```
    {
        cout << name << " is gossiping" << endl;
        cmd.execute();
    }
    void listen()
    {
        cout << name << " is listening" << endl;
    }
};

int main()
{
  // Fred will "execute" Barney which will result in a call to passOn()
  // Barney will "execute" Betty which will result in a call to gossip()
  // Betty will "execute" Wilma which will result in a call to listen()
  Person wilma("Wilma", Command());
  // 2. Instantiate an object for each "callback"
  // 3. Pass each object to its future "sender"
  Person betty("Betty", Command(&wilma, &Person::listen));
  Person barney("Barney", Command(&betty, &Person::gossip));
  Person fred("Fred", Command(&barney, &Person::passOn));
  fred.talk();
}
```

### Output

```
Fred is talking
Barney is passing on
Betty is gossiping
Wilma is listening
```

## Read next

This article is taken from our book **Design Patterns Explained Simply**.

All of the design patterns are compiled there. The book is written in clear, simple language that makes it easy to read and understand (just like this article).

We distribute it in PDF & EPUB formats so you can get it onto your iPad, Kindle, or other portable device immediately after your purchase.


## Code examples

| | | | |
|---|---|---|---|
| Java | Command in Java: Decoupling producer from consumer | Command in Java | |
| C++ | Command in C++: Before and after | Command in C++ | Command in C++: Simple and 'macro' commands |
| PHP | Command in PHP | | |
| Delphi | Command in Delphi | | |

Design Patterns
AntiPatterns
Refactoring
UML

My account
Forum
Contact us
About us