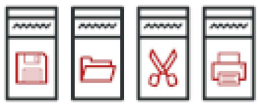

Command in C++

Command is behavioral design pattern that converts requests or simple operations into objects.

The conversion allows deferred or remote execution of commands, storing command history, etc.

Learn more about Command

Complexity:

Popularity:

Usage examples: The Command pattern is pretty common in C++ code. Most often it's used as an alternative for callbacks to parameterizing UI elements with actions. It's also used for queueing tasks, tracking operations history, etc.

Identification: The Command pattern is recognizable by behavioral methods in an abstract/interface type (sender) which invokes a method in an implementation of a different abstract/interface type (receiver) which has been encapsulated by the command implementation during its creation. Command classes are usually limited to specific actions.

Conceptual Example

This example illustrates the structure of the **Command** design pattern. It focuses on answering these questions:

- What classes does it consist of?
- What roles do these classes play?
- In what way the elements of the pattern are related?

main.cc: Conceptual example

```
/**
 * The Command interface declares a method for executing a command.
 */
class Command {
 public:
  virtual ~Command() {
  }
  virtual void Execute() const = 0;
};
/**
 * Some commands can implement simple operations on their own.
 */
class SimpleCommand : public Command {
 private:
  std::string pay_load_;

 public:
  explicit SimpleCommand(std::string pay_load) : pay_load_(pay_load) {
  }
  void Execute() const override {
    std::cout << "SimpleCommand: See, I can do simple things like printing (" << this->pay_lo
  }
};

/**
 * The Receiver classes contain some important business logic. They know how to
 * perform all kinds of operations, associated with carrying out a request. In
 * fact, any class may serve as a Receiver.
 */
class Receiver {
 public:
  void DoSomething(const std::string &a) {
    std::cout << "Receiver: Working on (" << a << ".)\n";
  }
  void DoSomethingElse(const std::string &b) {
    std::cout << "Receiver: Also working on (" << b << ".)\n";
  }
};

/**
 * However, some commands can delegate more complex operations to other objects,
 * called "receivers."
 */
class ComplexCommand : public Command {
  /**
   * @var Receiver
   */
 private:
  Receiver *receiver_;
  /**
   * Context data, required for launching the receiver's methods.
```

```
    */
   std::string a_;
   std::string b_;
   /**
    * Complex commands can accept one or several receiver objects along with any
    * context data via the constructor.
    */
  public:
   ComplexCommand(Receiver *receiver, std::string a, std::string b) : receiver_(receiver), a_(
   }
   /**
    * Commands can delegate to any methods of a receiver.
    */
   void Execute() const override {
     std::cout << "ComplexCommand: Complex stuff should be done by a receiver object.\n";
     this->receiver_->DoSomething(this->a_);
     this->receiver_->DoSomethingElse(this->b_);
   }
};

/**
 * The Invoker is associated with one or several commands. It sends a request to
 * the command.
 */
class Invoker {
   /**
    * @var Command
    */
  private:
   Command *on_start_;
   /**
    * @var Command
    */
   Command *on_finish_;
   /**
    * Initialize commands.
    */
  public:
   ~Invoker() {
     delete on_start_;
     delete on_finish_;
   }

   void SetOnStart(Command *command) {
     this->on_start_ = command;
   }
   void SetOnFinish(Command *command) {
     this->on_finish_ = command;
   }
   /**
    * The Invoker does not depend on concrete command or receiver classes. The
    * Invoker passes a request to a receiver indirectly, by executing a command.
```

```
   */
  void DoSomethingImportant() {
    std::cout << "Invoker: Does anybody want something done before I begin?\n";
    if (this->on_start_) {
      this->on_start_->Execute();
    }
    std::cout << "Invoker: ...doing something really important...\n";
    std::cout << "Invoker: Does anybody want something done after I finish?\n";
    if (this->on_finish_) {
      this->on_finish_->Execute();
    }
  }
};
/**
 * The client code can parameterize an invoker with any commands.
 */

int main() {
  Invoker *invoker = new Invoker;
  invoker->SetOnStart(new SimpleCommand("Say Hi!"));
  Receiver *receiver = new Receiver;
  invoker->SetOnFinish(new ComplexCommand(receiver, "Send email", "Save report"));
  invoker->DoSomethingImportant();

  delete invoker;
  delete receiver;

  return 0;
}
```

Output.txt: Execution result

```
Invoker: Does anybody want something done before I begin?
SimpleCommand: See, I can do simple things like printing (Say Hi!)
Invoker: ...doing something really important...
Invoker: Does anybody want something done after I finish?
ComplexCommand: Complex stuff should be done by a receiver object.
Receiver: Working on (Send email.)
Receiver: Also working on (Save report.)
```