Behavioral Design Pattern

Hung Tran

Fpt software

December 2, 2021

Outline

Behavioral Pattern Overview

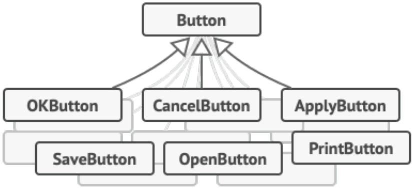
Behavioral design patterns are concerned with algorithms and the assignment of responsibilities between objects.

- **Chain of responsibility**: lets you pass requests along a chain of handlers.
- **Command**: turns a request into a stand-alone object that contains all information about the request.
- **Iterator**: lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).
- **Mediator**: lets you reduce chaotic dependencies between objects.
- **Memento**: lets you save and restore the previous state of an object without revealing the details of its implementation.

Behavioral Pattern Overview

- **Observer**: lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.
- **State**: lets an object alter its behavior when its internal state changes. It appears as if the object changed its class.
- **Strategy**: lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.
- **Template Method**: defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.
- **Visitor**: lets you separate algorithms from the objects on which they operate.

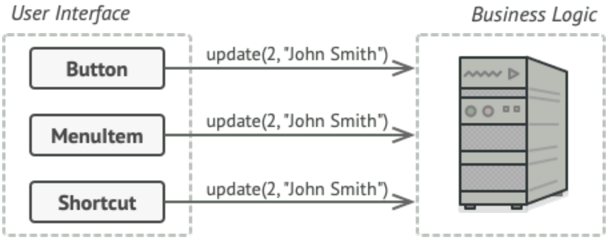
Problem Statement: text-editor app



- Create a toolbar with a bunch of buttons for various operations of the editor.
- Created a very neat Button class that can be used for buttons on the toolbar, as well as for generic buttons in various dialogs.
- While all of these buttons look similar, they're all supposed to do different things.
- Where would you put the code for the various click handlers of these buttons?
- The simplest solution is to create tons of subclasses for each place where the button is used.

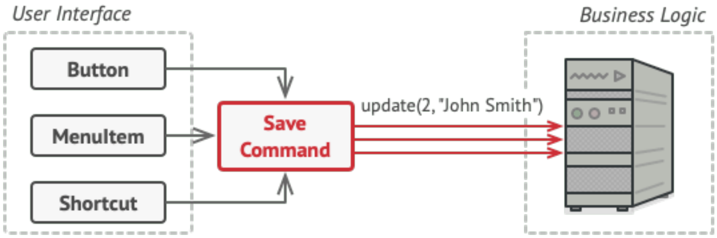
You have an enormous number of subclasses and multiple ways to do the same operation.

The principle of separation of concerns



- A layer for the graphical user interface.
- Another layer for the business logic.
- The GUI layer is responsible for rendering a beautiful picture on the screen, capturing any input and showing results of what the user and the app are doing.

The command pattern

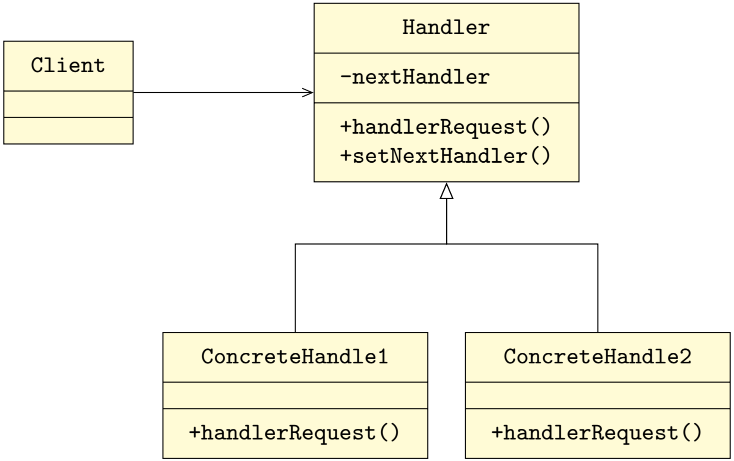


- The Command pattern suggests that GUI objects shouldn't send these requests directly.
- You should extract all of the request details, such as the object being called, the name of the method and the list of arguments.
- Command objects serve as links between various GUI and business logic objects.
- The GUI object doesn't need to know what business logic object will receive the request and how it'll be processed.

The Intent of Command Design Pattern

Command is a behavioral design pattern that turns a request into a stand-alone object that contains all information about the request. This transformation lets you pass requests as a method arguments, delay or queue a request's execution, and support undoable operations.

Structure of Command Pattern



Basic implementation: bulb class

bulb.h

```cpp
#ifndef _BULB_H_
#define _BULB_H_
class Bulb {
public:
  void turnOn();
  void turnOff();
};
#endif // _BULB_H_
```

bulb.cpp

```cpp
#include "bulb.h"
#include <iostream>

void Bulb::turnOn() {
  std::cout << "Bulb has been lit." <<
      std::endl;
}

void Bulb::turnOff() {
  std::cout << "Darkness!" << std::endl;
}
```

command.h

```cpp
#ifndef _COMMAND_H_
#define _COMMAND_H_
class Command {
public:
  virtual void execute(void) = 0;
  virtual void undo(void) = 0;
  virtual void redo(void) = 0;
};
#endif // _COMAND_H_
```

Basic implementation: TunOn class

turnOn.h

```cpp
#ifndef _TURN_ON_H_
#define _TURN_ON_H_
#include <memory>
#include "command.h"
#include "bulb.h"
class TurnOn : public Command {
public:
  TurnOn(std::shared_ptr<Bulb> bulb);
  void execute(void);
  void undo(void);
  void redo(void);
private:
  std::shared_ptr<Bulb> bulb_;
};
#endif // _TURN_ON_H_
```

turnOn.cpp

```cpp
#include "turnOn.h"

TurnOn::TurnOn(std::shared_ptr<Bulb>
      bulb) : bulb_(bulb) {}

void TurnOn::execute(void) {
  bulb_->turnOn();
}

void TurnOn::undo(void) {
  bulb_->turnOff();
}

void TurnOn::redo(void) {
  execute();
}
```

Basic implementation: TurnOff class

turnOff.h

```cpp
#ifndef _TURN_OFF_H_
#define _TURN_OFF_H_
#include "command.h"
#include "bulb.h"
#include <memory>
class TurnOff : public Command {
public:
  TurnOff(std::shared_ptr <Bulb> bulb);
  void execute(void);
  void undo(void);
  void redo(void);
private:
  std::shared_ptr<Bulb> bulb_;
};
#endif // _TURN_OFF_H_
```

turnOff.cpp

```cpp
#include "turnOff.h"

TurnOff::TurnOff(std::shared_ptr<Bulb>
     bulb) : bulb_(bulb) {}

void TurnOff::execute(void) {
  bulb_->turnOff();
}

void TurnOff::undo(void) {
  bulb_->turnOn();
}

void TurnOff::redo(void) {
  execute();
}
```

Basic implementation: remote class

remote.h

```cpp
#ifndef _REMOTE_H_
#define _REMOTE_H_
#include "command.h"
#include <memory>
class Remote {
public:
  void submit(std::shared_ptr<Command> command);
};
#endif // _REMOTE_H_
```

remote.cpp

```cpp
#include "remote.h"

void Remote::submit(std::shared_ptr<Command> command) {
  command->execute();
}
```

Basic implementation: main

main.cpp

```cpp
#include "bulb.h"
#include "turnOn.h"
#include "turnOff.h"
#include "remote.h"
#include <memory>

int main() {
  std::shared_ptr<Bulb> bulb = std::
      make_shared<Bulb>();

  std::shared_ptr<TurnOn> turnOn = std::
      make_shared<TurnOn>(bulb);
  std::shared_ptr<TurnOff> turnOff = std
      ::make_shared<TurnOff>(bulb);

  Remote remote;
  remote.submit(turnOn);
  remote.submit(turnOff);

  return 0;
}
```

Applicability

- Use the Command pattern when you want to parametrize objects with operations.
- Use the Command pattern when you want to queue operations, schedule their execution, or execute them remotely.
- Use the Command pattern when you want to implement reversible operations..

How to Implement

- Declare the command interface with a single execution method.
- Start extracting requests into concrete command classes that implement the command interface. Each class must have a set of fields for storing the request arguments along with a reference to the actual receiver object. All these values must be initialized via the command's constructor.
- Identify classes that will act as senders. Add the fields for storing commands into these classes. Senders should communicate with their commands only via the command interface. Senders usually don't create command objects on their own, but rather get them from the client code.
- Change the senders so they execute the command instead of sending a request to the receiver directly.
- The client should initialize objects in the following order: Create receivers. Create commands, and associate them with receivers if needed. Create senders, and associate them with specific commands.

Pros and Cons

- Single Responsibility Principle. You can decouple classes that invoke operations from classes that perform these operations.
- Open/Closed Principle. You can introduce new commands into the app without breaking existing client code.
- You can implement undo/redo.
- You can implement deferred execution of operations.
- You can assemble a set of simple commands into a complex one.

- The code may become more complicated since you're introducing a whole new layer between senders and receivers.

Relations with Other Patterns

- Chain of Responsibility, Command, Mediator and Observer address various ways of connecting senders and receivers of requests:
- Handlers in Chain of Responsibility can be implemented as Commands. In this case, you can execute a lot of different operations over the same context object, represented by a request.
- You can use Command and Memento together when implementing "undo". In this case, commands are responsible for performing various operations over a target object, while mementos save the state of that object just before a command gets executed.
- Command and Strategy may look similar because you can use both to parameterize an object with some action. However, they have very different intents.
- Prototype can help when you need to save copies of Commands into history.
- You can treat Visitor as a powerful version of the Command pattern. Its objects can execute operations over various objects of different

Thank You!