Mediator in C++

Mediator is a behavioral design pattern that reduces coupling between components of a program by making them communicate indirectly, through a special mediator object.

The Mediator makes it easy to modify, extend and reuse individual components because they’re no longer dependent on the dozens of other classes.

Learn more about Mediator

Complexity:

Popularity:

Usage examples: The most popular usage of the Mediator pattern in C++ code is facilitating communications between GUI components of an app. The synonym of the Mediator is the Controller part of MVC pattern.

Conceptual Example

This example illustrates the structure of the **Mediator** 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

```
#include <iostream>
#include <string>
/**
 * The Mediator interface declares a method used by components to notify the
 * mediator about various events. The Mediator may react to these events and
 * pass the execution to other components.
 */
```

```
class BaseComponent;
class Mediator {
 public:
  virtual void Notify(BaseComponent *sender, std::string event) const = 0;
};

/**
 * The Base Component provides the basic functionality of storing a mediator's
 * instance inside component objects.
 */
class BaseComponent {
 protected:
  Mediator *mediator_;

 public:
  BaseComponent(Mediator *mediator = nullptr) : mediator_(mediator) {
  }
  void set_mediator(Mediator *mediator) {
    this->mediator_ = mediator;
  }
};

/**
 * Concrete Components implement various functionality. They don't depend on
 * other components. They also don't depend on any concrete mediator classes.
 */
class Component1 : public BaseComponent {
 public:
  void DoA() {
    std::cout << "Component 1 does A.\n";
    this->mediator_->Notify(this, "A");
  }
  void DoB() {
    std::cout << "Component 1 does B.\n";
    this->mediator_->Notify(this, "B");
  }
};

class Component2 : public BaseComponent {
 public:
  void DoC() {
    std::cout << "Component 2 does C.\n";
    this->mediator_->Notify(this, "C");
  }
  void DoD() {
    std::cout << "Component 2 does D.\n";
    this->mediator_->Notify(this, "D");
  }
};

/**
 * Concrete Mediators implement cooperative behavior by coordinating several
 * components.
 */
class ConcreteMediator : public Mediator {
 private:
  Component1 *component1_;
  Component2 *component2_;

 public:
  ConcreteMediator(Component1 *c1, Component2 *c2) : component1_(c1), component2_(c2) {
    this->component1_->set_mediator(this);
    this->component2_->set_mediator(this);
  }
  void Notify(BaseComponent *sender, std::string event) const override {
    if (event == "A") {
      std::cout << "Mediator reacts on A and triggers following operations:\n";
      this->component2_->DoC();
    }
    if (event == "D") {
      std::cout << "Mediator reacts on D and triggers following operations:\n";
      this->component1_->DoB();
      this->component2_->DoC();
    }
  }
};

/**
 * The client code.
 */

void ClientCode() {
  Component1 *c1 = new Component1;
  Component2 *c2 = new Component2;
  ConcreteMediator *mediator = new ConcreteMediator(c1, c2);
  std::cout << "Client triggers operation A.\n";
  c1->DoA();
  std::cout << "\n";
  std::cout << "Client triggers operation D.\n";
  c2->DoD();

  delete c1;
  delete c2;
  delete mediator;
}

int main() {
  ClientCode();
  return 0;
}
```

Output.txt: Execution result

```
Client triggers operation A.
Component 1 does A.
Mediator reacts on A and triggers following operations:
Component 2 does C.

Client triggers operation D.
Component 2 does D.
Mediator reacts on D and triggers following operations:
Component 1 does B.
Component 2 does C.
```