Poznan University of Technology
Object Oriented Programming
Design Patterns - C++ code examples

Jakub Piotr Hamerliński, M.Eng.

https://www.linkedin.com/in/hamerlinski

https://github.com/hamerlinski

Object Oriented Programming

Agenda

01. Design Patterns - C++ code examples

> "I do not apologize unless I think I'm wrong. And if you don't like it you can leave."

Stanley Hudson

Design Patterns

Object Oriented Programming

Design Patterns

List of design patterns we will learn more about:

01. Resource acquisition is initialization aka initialism,
02. Chain of Responsibility,
03. Lazy Initialization,
04. Abstract Factory,
05. Specification,
06. Null Object,
07. Object Pool…

Object Oriented Programming

Design Patterns

And:

01. Composite,
02. Decorator (we skip it since it has been brought already),
03. Strategy,
04. Adapter,
05. Command,
06. Bridge,
07. Proxy.

Object Oriented Programming

RAII

Resource acquisition is initialization aka initialism: Resource allocation (or acquisition) is done during object creation (specifically initialization), by the constructor, while resource deallocation (release) is done during object destruction (specifically finalization), by the destructor.

Object Oriented Programming

RAII

```cpp
#include <mutex>
#include <stdexcept>
#include <iostream>

class Permit {
 public:
  explicit Permit(std::mutex& m) : m_(m) {
    m_.lock();
  }

  ~Permit() {
    m_.unlock();
  }

  Permit(const Permit&) = delete;
  Permit& operator=(const Permit&) = delete;

 private:
  std::mutex& m_;
};
```

Object Oriented Programming

RAII

```cpp
class Foo {
 public:
  void Print(int x) {
    std::unique_lock<std::mutex> lock(mutex_);
    try {
      Permit p(mutex_);
      if (x > 1000) {
        throw std::runtime_error("Too large!");
      }
      std::cout << "x = " << x << std::endl;
    } catch (const std::exception& e) {
      std::cerr << e.what() << std::endl;
    }
  }

 private:
  std::mutex mutex_;
};
```

Object Oriented Programming

RAII

```cpp
int main() {
  Foo foo;
  foo.Print(500);
  foo.Print(2000);
  foo.Print(800);
  return 0;
}
```

Object Oriented Programming

Chain of Responsibility

You may transmit requests along a chain of handlers using the behavioral design pattern known as Chain of Responsibility. Each handler chooses whether to execute a request or pass it on to the handler behind it in the chain.

Benefits:

01. The sequence in which a request is handled can be changed.
02. Principle of a single responsibility. Classes that conduct operations and classes that invoke operations can be separated.
03. Principle of Open/Closed. The app can add additional handlers without causing the client code to malfunction.

Object Oriented Programming

Chain of Responsibility

```cpp
#include <iostream>
class Approver {
 public:
  virtual ~Approver() {}
  virtual void Approve(int amount) = 0;

 protected:
  Approver* next_approve_ = nullptr;
};
class Manager : public Approver {
 public:
  Manager(Approver* next_approve) : next_approve_(next_approve) {}
  void Approve(int amount) override {
    if (amount <= 1000) {
      std::cout << "Manager approved $" << amount << std::endl;
    } else if (next_approve_) {
      next_approve_->Approve(amount);
    }
  }
 private:
  Approver* next_approve_;
};
```

Object Oriented Programming

Chain of Responsibility

```cpp
class Director : public Approver {
 public:
  Director(Approver* next_approve) : next_approve_(next_approve) {}
  void Approve(int amount) override {
    if (amount <= 10000) {
      std::cout << "Director approved $" << amount << std::endl;
    } else if (next_approve_) {
      next_approve_->Approve(amount);
    }
  }
 private:
  Approver* next_approve_;
};
class CEO : public Approver {
 public:
  void Approve(int amount) override {
    if (amount <= 100000) { std::cout << "CEO approved $" << amount << std::endl; }
    else { std::cout << "Amount too high, approval denied." << std::endl; }
  }
};
```

Object Oriented Programming

Chain of Responsibility

```cpp
int main() {
  CEO ceo;
  Director director(&ceo);
  Manager manager(&director);
  manager.Approve(500);
  manager.Approve(5000);
  manager.Approve(50000);
  return 0;
}
```

Object Oriented Programming

Lazy Initialization

Delaying the construction of an object, computing a value, or engaging in any other computationally expensive activity until it is to be utilized for the first time is a strategy called lazy initialization. Benefits:

01. reduces startup time by spreading out complicated computations over time.
02. avoids conducting infrequent operations unless absolutely essential.

Object Oriented Programming

Lazy Initialization

```cpp
#include <cassert>
class Response {
 public:
  static Response& Instance() {
    static Response response;
    return response;
  }
 private:
  Response() = default;
  Response(const Response&) = delete;
  Response& operator=(const Response&) = delete;
};
int main() {
  Response& response1 = Response::Instance();
  Response& response2 = Response::Instance();
  assert(&response1 == &response2);
  return 0;
}
```

Object Oriented Programming

Lazy Initialization

In this example, the Response class uses the Lazy Initialization design pattern to create only one instance of the class when it is needed. The instance variable is initially set to null and is only created using the new operator when the instance() method is called for the first time. This ensures that the instance is only created when it is needed, and not before.

Object Oriented Programming

Abstract Factory

A creational design pattern called Abstract Factory enables you to create families of linked things without identifying their particular classes.

```cpp
#include <iostream>
#include <memory>
class Button {
 public:
  virtual ~Button() {}
  virtual void Paint() = 0;
};
class MacOSButton : public Button {
 public:
  void Paint() override { std::cout << "You have created MacOSButton." << std::endl; }
};
class WindowsButton : public Button {
 public:
  void Paint() override { std::cout << "You have created WindowsButton." << std::endl; }
};
```

Object Oriented Programming

Abstract Factory

```cpp
class GUIFactory {
 public:
  virtual ~GUIFactory() {}
  virtual std::unique_ptr<Button> CreateButton() = 0;
};
class MacOSFactory : public GUIFactory {
 public:
  std::unique_ptr<Button> CreateButton() override {
    return std::make_unique<MacOSButton>();
  }
};
class WindowsFactory : public GUIFactory {
 public:
  std::unique_ptr<Button> CreateButton() override {
    return std::make_unique<WindowsButton>();
  }
};
```

Object Oriented Programming

Abstract Factory

```cpp
class Application {
 public:
  explicit Application(std::unique_ptr<GUIFactory> factory)
      : button_(std::move(factory->CreateButton())) {}
  void Paint() {
    button_->Paint();
  }

 private:
  std::unique_ptr<Button> button_;
};
```

Object Oriented Programming

Abstract Factory

```cpp
int main() {
  std::unique_ptr<GUIFactory> factory;
  std::string os_name = std::getenv("OS");
  if (os_name.find("Mac") != std::string::npos) {
    factory = std::make_unique<MacOSFactory>();
  } else {
    factory = std::make_unique<WindowsFactory>();
  }
  Application app(std::move(factory));
  app.Paint();
  return 0;
}
```

Object Oriented Programming

Abstract Factory

In this implementation, we define two concrete button classes, MacOSButton and WindowsButton, which implement the Button interface. We also define two concrete factory classes, MacOSFactory and WindowsFactory, which implement the GUIFactory interface and return instances of the appropriate button class.

Finally, we define the Application class, which takes a GUIFactory object as a parameter to its constructor and uses it to create a Button object. The Paint function of the Application class then calls the Paint function of the Button object to display the appropriate message to the console. In the main function, we create an instance of the Application class by passing a concrete factory object to its constructor based on the current operating system.

Object Oriented Programming

Specification

The Specification design pattern is a pattern that allows for the creation of complex logical expressions, made up of simpler expressions, that can be used to filter a collection of objects. It separates the logic of filtering objects from the objects themselves, making the code more reusable and readable.

The Specification pattern defines two main components: the Specification interface and the Specification implementations. The Specification interface defines the methods that need to be implemented by the concrete classes, such as isSatisfiedBy() and and(). The concrete classes implement these methods and define the logic for filtering the objects.

Object Oriented Programming

Specification

```cpp
#include <iostream>
#include <memory>
#include <vector>

class Product {
 public:
  Product(const std::string& name, double price) : name_(name), price_(price) {}

  const std::string& Name() const { return name_; }
  double Price() const { return price_; }

 private:
  std::string name_;
  double price_;
};

class Specification {
 public:
  virtual ~Specification() = default;
  virtual bool IsSatisfied(const Product& product) const = 0;
};
```

Object Oriented Programming

Specification

```cpp
class PriceSpecification : public Specification {
 public:
  PriceSpecification(double min_price, double max_price)
      : min_price_(min_price), max_price_(max_price) {}
  bool IsSatisfied(const Product& product) const override {
    return product.Price() >= min_price_ && product.Price() <= max_price_;
  }
 private:
  double min_price_;
  double max_price_;
};
class NameSpecification : public Specification {
 public:
  explicit NameSpecification(const std::string& name) : name_(name) {}
  bool IsSatisfied(const Product& product) const override {
    return product.Name() == name_;
  }
 private:
  std::string name_;
};
```

Object Oriented Programming

Specification

```cpp
class AndSpecification : public Specification {
 public:
  AndSpecification(const Specification& spec1, const Specification& spec2)
      : spec1_(spec1), spec2_(spec2) {}
  bool IsSatisfied(const Product& product) const override {
    return spec1_.IsSatisfied(product) && spec2_.IsSatisfied(product);
  }
 private:
  const Specification& spec1_;
  const Specification& spec2_;
};
```

Object Oriented Programming

Specification

```cpp
class Filter {
 public:
  static std::vector<Product> ApplyFilter(
      const std::vector<Product>& products,
      const Specification& specification) {
    std::vector<Product> filtered_products;
    for (const auto& product : products) {
      if (specification.IsSatisfied(product)) {
        filtered_products.push_back(product);
      }
    }
    return filtered_products;
  }
};
```

Object Oriented Programming

Specification

```cpp
int main() {
  std::vector<Product> products = {
      Product("Apple", 1.5),
      Product("Orange", 1.0),
      Product("Apple", 1.9),
      Product("Banana", 0.8),
      Product("Mango", 2.0),
      Product("Apple", 2.1),
  };
  PriceSpecification price_spec(1.0, 2.0);
  NameSpecification name_spec("Apple");
  AndSpecification and_spec(price_spec, name_spec);
  auto filtered_products = Filter::ApplyFilter(products, and_spec);
  for (const auto& product : filtered_products) {
    std::cout << product.Name() << " - " << product.Price() << std::endl;
  }
  return 0;
}
```

Object Oriented Programming

Null Object

The Null Object pattern is a pattern that allows for the handling of null values in an object-oriented way. Instead of using null values, the pattern uses a special Null object that implements the same interface as the actual objects, but has no implementation for its methods. This allows for a consistent way of handling null values without having to check for null values everywhere in the code.

Object Oriented Programming

Null Object

```cpp
#include <iostream>
#include <memory>
#include <string>
class Shape {
 public:
  virtual ~Shape() {}
  virtual void Draw() = 0;
};
class Rectangle : public Shape {
 public:
  void Draw() override {
    std::cout << "Drawing a rectangle." << std::endl;
  }
};
```

Object Oriented Programming

Null Object

```cpp
class NullShape : public Shape {
 public:
  void Draw() override { /* Do nothing */ }
};
class ShapeFactory {
 public:
  std::unique_ptr<Shape> CreateShape(const std::string& shape_type) {
    if (shape_type.empty()) {
      return std::make_unique<NullShape>();
    }
    if (shape_type == "rectangle") {
      return std::make_unique<Rectangle>();
    }
    return std::make_unique<NullShape>();
  }
};
```

Object Oriented Programming

Null Object

```cpp
int main() {
  ShapeFactory shape_factory;
  std::unique_ptr<Shape> shape1 = shape_factory.CreateShape("rectangle");
  shape1->Draw();  // Outputs "Drawing a rectangle."
  std::unique_ptr<Shape> shape2 = shape_factory.CreateShape("");
  shape2->Draw();  // Does nothing
  return 0;
}
```

Object Oriented Programming

Object Pool

The Object Pool pattern is a software creational pattern that allows for the reuse of objects that are expensive to create. Instead of creating a new object every time one is needed, the object pool maintains a pool of objects that can be reused. When a client requests an object, the pool checks if it has an available object to give, and if so, it returns it. If not, it creates a new object and adds it to the pool.

Object Oriented Programming

Object Pool

```
#include <memory>
#include <iostream>
#include <list>
class PooledObject {
 public:
  virtual ~PooledObject() {}
  virtual void Reset() = 0;
};
class Connection : public PooledObject {
 public:
  bool IsInUse() const { return in_use_; }
  void ChangeUsage() { in_use_ = !in_use_; }
  void Reset() override {
    std::cout << "Connection reset\nUsage false\n";
    // code to reset the connection
  }
  Connection () {
    this->in_use_ = false;
  }
 private:
  bool in_use_ = false;
};
```

Object Oriented Programming

Object Pool

```cpp
class ConnectionPool {
 public:
  std::shared_ptr<Connection> CreateConnection() {
    std::shared_ptr<Connection> connection;
    if (available_connections_.empty()) {
      connection = std::make_shared<Connection>();  // create a new connection
      std::cout << "New connection created\n";
    } else {
      connection = available_connections_.front();  // return an available connection
      available_connections_.pop_front();
      std::cout << "Existing connection reused\n";
    }
    connection->ChangeUsage();
    in_use_connections_.push_back(connection);
    return connection;
  }
  void ReleaseConnection(std::shared_ptr<Connection> connection);
 private:
  std::list<std::shared_ptr<Connection>> available_connections_;
  std::list<std::shared_ptr<Connection>> in_use_connections_;
};
```

Object Oriented Programming

Object Pool

```cpp
void ConnectionPool::ReleaseConnection(std::shared_ptr<Connection> connection) {
    connection->Reset();
    in_use_connections_.remove(connection);
    available_connections_.push_back(connection);
    std::cout << "Connection released\n";
  }
int main() {
  ConnectionPool pool;
  std::shared_ptr<Connection> connection1 = pool.CreateConnection();
  std::shared_ptr<Connection> connection2 = pool.CreateConnection();
  pool.ReleaseConnection(connection1);
  std::shared_ptr<Connection> connection3 = pool.CreateConnection();
  pool.ReleaseConnection(connection2);
  pool.ReleaseConnection(connection3);
  return 0;
}
```

Object Oriented Programming

Object Pool

In this example, the Connection class implements the PooledObject interface, and it defines the reset method that will be used when the connection is released back to the pool. The ConnectionPool class is responsible for maintaining a pool of connections and providing them to the client when requested. When a client requests a connection, the pool checks if there are available connections, and if not it creates a new one. When a client releases a connection, the pool resets it and adds it back to the available connections list.

Object Oriented Programming

Composite

With the use of the structural design pattern known as Composite, you can group things into tree structures and then treat those structures like individual objects. Use polymorphism and recursion to your advantage to work with complex tree structures more easily. Principle of Open/Closed. The existing code can continue to function with the object tree without being affected by the addition of additional element types.

Object Oriented Programming

Composite

```cpp
#include <iostream>
#include <memory>
#include <vector>
class Component {
 public:
  virtual ~Component() {}
  virtual void Operation() = 0;
};
class Leaf : public Component {
 public:
  void Operation() override {
    std::cout << "Leaf operation\n";
  }
};
```

Object Oriented Programming

Composite

```cpp
class Composite : public Component {
 public:
  void Operation() override {
    std::cout << "Composite operation\n";
    for (const auto& child : children_) {
      child->Operation();
    }
  }
  void Add(std::unique_ptr<Component> child) { children_.emplace_back(std::move(child)); }
  void Remove(Component* child) {
    for (auto it = children_.begin(); it != children_.end(); ++it) {
      if (it->get() == child) {
        children_.erase(it);
        break;
      }
    }
  }
  Component* Child(int index) const { return children_[index].get(); }
 private:
  std::vector<std::unique_ptr<Component>> children_;
};
```

Object Oriented Programming

Composite

```cpp
int main() {
  auto composite = std::make_unique<Composite>();
  composite->Add(std::make_unique<Leaf>());
  composite->Add(std::make_unique<Leaf>());
  composite->Operation();
}
```

Object Oriented Programming

Composite

In this example, the Component interface defines the Operation() method that must be implemented by both the Leaf and Composite classes. The Leaf class represents the leaf objects in the tree structure, and it implements the Operation() method with its specific behavior. The Composite class represents the composite objects in the tree structure, and it implements the Operation() method by iterating over its children and calling the Operation() method on each of them. The main uses the composite and leaf objects in the same way, by calling the Operation() method on them.

Object Oriented Programming

Strategy

The Strategy design pattern is a behavioral pattern that allows for the selection of an algorithm at runtime. It defines a family of algorithms, encapsulates each one, and makes them interchangeable. It also allows the client to choose the appropriate algorithm without having to know its implementation details.

Object Oriented Programming

Strategy

```cpp
#include <iostream>
#include <memory>
#include <string>
#include <vector>

class PaymentStrategy {
 public:
  virtual ~PaymentStrategy() {}
  virtual void Pay(double amount) = 0;
};
```

Object Oriented Programming

Strategy

```cpp
class CreditCardStrategy : public PaymentStrategy {
 public:
  CreditCardStrategy(const std::string& name, const std::string& cardNumber,
                     const std::string& cvv, const std::string& dateOfExpiry)
      : name_(name), cardNumber_(cardNumber), cvv_(cvv), dateOfExpiry_(dateOfExpiry) {}

  void Pay(double amount) override {
    std::cout << amount << " paid with credit/debit card\n";
  }

 private:
  std::string name_;
  std::string cardNumber_;
  std::string cvv_;
  std::string dateOfExpiry_;
};
```

Object Oriented Programming

Strategy

```cpp
class PayPalStrategy : public PaymentStrategy {
 public:
  PayPalStrategy(const std::string& emailId, const std::string& password)
      : emailId_(emailId), password_(password) {}

  void Pay(double amount) override {
    std::cout << amount << " paid using PayPal.\n";
  }

 private:
  std::string emailId_;
  std::string password_;
};

class Item {
 public:
  virtual ~Item() {}
  virtual double Price() const = 0;
};
```

Object Oriented Programming

Strategy

```cpp
class ShoppingCart {
 public:
  ShoppingCart(const std::vector<std::unique_ptr<Item>>& items, PaymentStrategy* paymentMethod)
      : items_(items), paymentMethod_(paymentMethod) {}

  void Pay() {
    double totalAmount = CalculateTotal();
    paymentMethod_->Pay(totalAmount);
  }

 private:
  double CalculateTotal() const {
    double total = 0;
    for (const auto& item : items_) {
      total += item->Price();
    }
    return total;
  }

  std::vector<std::unique_ptr<Item>> items_;
  PaymentStrategy* paymentMethod_;
};
```

Object Oriented Programming

Strategy

```cpp
class Book : public Item {
 public:
  Book(double price) : price_(price) {}
  double Price() const override {
    return price_;
  }
 private:
  double price_;
};

class Movie : public Item {
 public:
  Movie(double price) : price_(price) {}
  double Price() const override {
    return price_;
  }
 private:
  double price_;
};
```

Object Oriented Programming

Strategy

```cpp
int main() {
  std::vector<std::unique_ptr<Item>> items;
  items.emplace_back(std::make_unique<Book>(9.99));
  items.emplace_back(std::make_unique<Movie>(14.99));
  items.emplace_back(std::make_unique<Book>(12.99));
  items.emplace_back(std::make_unique<Movie>(19.99));
  PaymentStrategy* paymentMethod = new CreditCardStrategy("John Doe", "1234 5678 9012 3456", "123", "06/24");
  ShoppingCart cart(items, paymentMethod);
  cart.Pay();
  delete paymentMethod;
  paymentMethod = new PayPalStrategy("john.doe@example.com", "password123");
  cart = ShoppingCart(items, paymentMethod);
  cart.Pay();
  delete paymentMethod;
  return 0;
}
```

Object Oriented Programming

Strategy

In this example, the PaymentStrategy interface defines the pay method that must be implemented by the concrete classes (CreditCardStrategy and PayPalStrategy). The ShoppingCart class uses the PaymentStrategy interface to allow the client to choose the appropriate payment method at runtime, and it doesn't need to know the implementation details. It allows the client to use different Payment strategies that implements the PaymentStrategy interface and use the pay method accordingly. You can also add more Payment strategies like BankTransfer, Wallet etc. and use them in the ShoppingCart class.

Object Oriented Programming

Adapter

The Adapter design pattern is a structural pattern that allows the adaptation of an existing interface to a different one that the client expects. It helps to integrate new systems and legacy systems by allowing them to work together seamlessly.

Object Oriented Programming

Adapter

```cpp
#include <iostream>
#include <memory>
#include <string>
class LegacySystem {
 public:
  virtual void processData(std::string data) = 0;
  virtual ~LegacySystem() = default;
};
class LegacySystemImpl : public LegacySystem {
 public:
  void processData(std::string data) override {
    // code to process data using the legacy system
    std::cout << "Processing data using legacy system: " << data << std::endl;
  }
};
```

Object Oriented Programming

Adapter

```
class NewSystem {
 public:
  virtual void process(std::string data) = 0;
  virtual ~NewSystem() = default;
};
class NewSystemAdapter : public NewSystem {
 public:
  NewSystemAdapter(std::shared_ptr<LegacySystem> legacySystem)
      : legacySystem_(legacySystem) {}
  void process(std::string data) override {
    legacySystem_->processData(data);
  }
 private:
  std::shared_ptr<LegacySystem> legacySystem_;
};
```

Object Oriented Programming

Adapter

```cpp
int main() {
  auto legacySystem = std::make_shared<LegacySystemImpl>();
  auto adapter = std::make_unique<NewSystemAdapter>(legacySystem);
  adapter->process("data to process");
  return 0;
}
```

Object Oriented Programming

Command

The Command design pattern is a behavioral pattern that allows for the encapsulation of a request as an object, separate from the object that actually carries out the request. This allows for decoupling the objects that initiate the request from the objects that execute it.

Object Oriented Programming

Command

```cpp
#include <iostream>
#include <memory>
class Command {
 public:
  virtual ~Command() {}
  virtual void Execute() = 0;
};
class Light {
 public:
  void TurnOn() { std::cout << "The dark is on\n"; }
  void TurnOff() { std::cout << "The dark is off\n"; }
};
```

Object Oriented Programming

Command

```cpp
class TurnOnLightCommand : public Command {
 public:
  TurnOnLightCommand(std::shared_ptr<Light> dark) : dark_(dark) {}
  void Execute() override {
    dark_->TurnOn();
  }
 private:
  std::shared_ptr<Light> dark_;
};
class TurnOffLightCommand : public Command {
 public:
  TurnOffLightCommand(std::shared_ptr<Light> dark) : dark_(dark) {}
  void Execute() override {
    dark_->TurnOff();
  }
 private:
  std::shared_ptr<Light> dark_;
};
```

Object Oriented Programming

Command

```cpp
class RemoteControl {
 public:
  void ChangeCommand(std::shared_ptr<Command> command) {
    command_ = command;
  }
  void PressButton() {
    command_->Execute();
  }
 private:
  std::shared_ptr<Command> command_;
};
```

Object Oriented Programming

Command

```cpp
int main() {
  std::shared_ptr<Light> dark = std::make_shared<Light>();
  std::shared_ptr<Command> turnOnCommand = std::make_shared<TurnOnLightCommand>(dark);
  std::shared_ptr<Command> turnOffCommand = std::make_shared<TurnOffLightCommand>(dark);
  RemoteControl remote;
  remote.ChangeCommand(turnOnCommand);
  remote.PressButton(); // prints "The dark is on"
  remote.ChangeCommand(turnOffCommand);
  remote.PressButton(); // prints "The dark is off"
  return 0;
}
```

Object Oriented Programming

Command

In this example, the Light class is the object that actually carries out the request, while the TurnOnLightCommand and TurnOffLightCommand classes are the objects that encapsulate the request and implement the Command interface. The RemoteControl class is the object that initiates the request, it holds a command and when the pressButton method is called, it calls the execute method of the command. This example shows how the command pattern allows for decoupling the objects that initiate the request from the objects that execute it.

Object Oriented Programming

Bridge

The Bridge design pattern is a structural pattern that allows for the separation of an abstraction from its implementation. It allows for the decoupling of the interface and the implementation, so that they can evolve independently.

Object Oriented Programming

Bridge

```cpp
#include <iostream>
class TV {
 public:
  virtual void On() = 0;
  virtual void Off() = 0;
  virtual void TuneChannel(int channel) = 0;
  virtual void ChangeVolume(int volume) = 0;
};
```

Object Oriented Programming

Bridge

```cpp
class RemoteControl {
 protected:
  TV* tv;
 public:
  RemoteControl(TV* tv) : tv(tv) {}
  void TurnOn() { tv->On(); }
  void TurnOff() { tv->Off(); }
  void ChangeChannel(int channel) { tv->TuneChannel(channel); }
};

class AdvancedRemoteControl : public RemoteControl {
 public:
  AdvancedRemoteControl(TV* tv) : RemoteControl(tv) {}
  void ChangeVolume(int volume) { tv->ChangeVolume(volume); }
};
```

Object Oriented Programming

Bridge

```cpp
class LGTV : public TV {
 public:
  void On() override { std::cout << "LG TV is turned on." << std::endl; }
  void Off() override { std::cout << "LG TV is turned off." << std::endl; }
  void TuneChannel(int channel) override { std::cout << "LG TV tuned to channel: " << channel << std::endl; }
  void ChangeVolume(int volume) override { std::cout << "LG TV volume set to: " << volume << std::endl; }
};
class SonyTV : public TV {
 public:
  void On() override { std::cout << "Sony TV is turned on." << std::endl; }
  void Off() override { std::cout << "Sony TV is turned off." << std::endl; }
  void TuneChannel(int channel) override { std::cout << "Sony TV tuned to channel: " << channel << std::endl; }
  void ChangeVolume(int volume) override { std::cout << "Sony TV volume set to: " << volume << std::endl; }
};
```

Object Oriented Programming

Bridge

```cpp
int main() {
  TV* lgTV = new LGTV();
  auto* advancedRemote = new AdvancedRemoteControl(lgTV);
  advancedRemote->TurnOn();             // prints "LG TV is turned on."
  advancedRemote->ChangeVolume(10);     // prints "LG TV volume set to: 10"
  advancedRemote->ChangeChannel(3);     // prints "LG TV tuned to channel: 3"
  advancedRemote->TurnOff();            // prints "LG TV is turned off."

  TV* sonyTV = new SonyTV();
  auto* remote = new RemoteControl(sonyTV);
  remote->TurnOn();             // prints "Sony TV is turned on."
  remote->ChangeChannel(7);     // prints "Sony TV tuned to channel: 7"
  remote->TurnOff();            // prints "Sony TV is turned off."
  delete lgTV, sonyTV, advancedRemote, remote;
  return 0;
}
```

Object Oriented Programming

Bridge

In this example, the RemoteControl interface defines the basic operations that can be performed on a TV. The RemoteControl class provides a basic implementation of these operations, delegating the actual work to the TV object. The AdvancedRemoteControl class extends the basic functionality by adding the ability to set the volume. The TV interface defines the basic operations that a TV must implement, and the LGTV and SonyTV classes are concrete implementations of this interface. The client can use the RemoteControl interface to interact with different TV implementations without having to know the details of their specific implementations.

Object Oriented Programming

Proxy

The Proxy design pattern is a structural pattern that provides a surrogate or placeholder object that controls access to the original object. The proxy object controls access to the original object, and it can add additional functionality, such as caching, logging, or security checks.

Object Oriented Programming

Proxy

```cpp
#include <iostream>
#include <string>
class Image {
public:
  virtual void display() = 0;
  virtual ~Image() = default;
};
class RealImage : public Image {
public:
  explicit RealImage(const std::string& fileName) : fileName_{fileName} {
    loadFromDisk(fileName);
  }
  void display() override { std::cout << "Displaying " << fileName_ << "\n"; }
private:
  void loadFromDisk(const std::string& fileName) { std::cout << "Loading " << fileName << "\n"; }
  std::string fileName_;
};
```

Object Oriented Programming

Proxy

```cpp
class ImageProxy : public Image {
public:
  explicit ImageProxy(const std::string& fileName) : fileName_{fileName} {}
  void display() override {
    if (!realImage_) {
      realImage_ = std::make_unique<RealImage>(fileName_);
    }
    realImage_->display();
  }
private:
  std::unique_ptr<RealImage> realImage_;
  std::string fileName_;
};
int main() {
  ImageProxy image{"test_10mb.jpg"};
  image.display();
  std::cout << "\n";
  image.display();
}
```

Object Oriented Programming

Proxy

In this example, the RealImage class is the actual object that will be displayed, and the ImageProxy class is the proxy that controls access to the RealImage object. The main requests the image to be displayed, but the actual image is only loaded from disk when the display() method is called on the ImageProxy object. This allows for lazy loading, where the image is only loaded when it's needed, which can be useful when working with large files or slow networks.

Additionally, the proxy can be used to add additional functionality, such as caching, logging or security checks before accessing the real image, this is the main purpose for having a proxy object.

Sources

Object Oriented Programming

Based on:

01. https://www.yegor256.com/2016/02/03/design-patterns-and-anti-patterns.html
02. https://www.yegor256.com/2017/08/08/raii-in-cpp.html
03. https://refactoring.guru/design-patterns

Thank you

Feel free to reach me via LinkedIn

Fin