Пример 09.16. Стратегия (Strategy).

# include <iostream>

# include <memory>

# include <vector>

using namespace std;

class Strategy

{

public:

virtual ~Strategy() = default;

virtual void algorithm() = 0;

};

class ConStrategy1 : public Strategy

{

public:

virtual void algorithm() override { cout << "Algorithm 1;" << endl; }

};

class ConStrategy2 : public Strategy

{

public:

virtual void algorithm() override { cout << "Algorithm 2;" << endl; }

};

class Context

{

private:

unique\_ptr<Strategy> strategy;

public:

explicit Context(Strategy\* ptr) : strategy(ptr) {}

void algorithmStrategy() { strategy->algorithm(); }

};

int main()

{

Context obj(new ConStrategy1());

obj.algorithmStrategy();

}

Пример 09.17. Стратегия (Strategy).

# include <iostream>

# include <memory>

# include <vector>

using namespace std;

class Strategy

{

public:

virtual ~Strategy() = default;

virtual void algorithm() = 0;

};

class ConStrategy1 : public Strategy

{

public:

virtual void algorithm() override { cout << "Algorithm 1;" << endl; }

};

class ConStrategy2 : public Strategy

{

public:

virtual void algorithm() override { cout << "Algorithm 2;" << endl; }

};

class Context

{

public:

void algorithmStrategy(shared\_ptr<Strategy> strategy) { strategy->algorithm(); }

};

int main()

{

Context obj;

obj.algorithmStrategy(shared\_ptr<Strategy>(new ConStrategy1()));

obj.algorithmStrategy(shared\_ptr<Strategy>(new ConStrategy2()));

}

Пример 09.18. Стратегия (Strategy). “Статический полиморфизм”.

# include <iostream>

# include <memory>

# include <vector>

using namespace std;

class Strategy

{

public:

virtual ~Strategy() = default;

virtual void algorithm() = 0;

};

class ConStrategy1 : public Strategy

{

public:

virtual void algorithm() override { cout << "Algorithm 1;" << endl; }

};

class ConStrategy2 : public Strategy

{

public:

virtual void algorithm() override { cout << "Algorithm 2;" << endl; }

};

template <typename CStrategy>

class Context

{

private:

unique\_ptr<CStrategy> strategy;

public:

Context() : strategy(new CStrategy()) {}

void algorithmStrategy() { strategy->algorithm(); }

};

int main()

{

Context<ConStrategy1> obj;

obj.algorithmStrategy();

}

Пример 09.29. “Статический полиморфизм”. CRTP (Curiously Recurring Template Pattern).

# include <iostream>

# include <memory>

using namespace std;

template<typename Implementation>

class Product

{

public:

void run()

{

impl()->run();

}

private:

Implementation\* impl()

{

return static\_cast<Implementation\*>(this);

}

};

class ConProd1 : public Product<ConProd1>

{

public:

void run() { cout << "Method run;" << endl; }

};

int main()

{

unique\_ptr<Product<ConProd1>> prod( new ConProd1());

prod->run();

}

Пример 09.19. Команда (Command).

# include <iostream>

# include <memory>

# include <vector>

# include <initializer\_list>

using namespace std;

class Command

{

public:

virtual ~Command() = default;

virtual void execute() = 0;

};

template <typename Reseiver>

class SimpleCommand : public Command

{

private:

using Action = void(Reseiver::\*)();

using Pair = pair<shared\_ptr<Reseiver>, Action>;

Pair call;

public:

SimpleCommand(shared\_ptr<Reseiver> r, Action a) : call(r, a) {}

virtual void execute() override { ((\*call.first).\*call.second)(); }

};

class CompoundCommand : public Command

{

using VectorCommand = vector<shared\_ptr<Command>>;

private:

VectorCommand vec;

public:

CompoundCommand(initializer\_list<shared\_ptr<Command>> lt);

virtual void execute() override;

};

# pragma region Methods

CompoundCommand::CompoundCommand(initializer\_list<shared\_ptr<Command>> lt)

{

for (auto elem : lt)

vec.push\_back(elem);

}

void CompoundCommand::execute()

{

for (auto com : vec)

com->execute();

}

# pragma endregion

class Object

{

public:

void run() { cout << "Run method;" << endl; }

};

int main()

{

shared\_ptr<Object> obj(new Object());

shared\_ptr<Command> command(new SimpleCommand<Object>(obj, &Object::run));

command->execute();

shared\_ptr<Command> complex(new CompoundCommand

{

shared\_ptr<Command>(new SimpleCommand<Object>(obj, &Object::run)),

shared\_ptr<Command>(new SimpleCommand<Object>(obj, &Object::run))

});

complex->execute();

}

Пример 09.20. Цепочка обязанностей (Chain of Responsibility).

# include <iostream>

# include <initializer\_list>

# include <memory>

using namespace std;

class AbstractHandler

{

protected:

shared\_ptr<AbstractHandler> next;

virtual bool run() = 0;

public:

using Default = shared\_ptr<AbstractHandler>;

virtual ~AbstractHandler() = default;

virtual bool handle() = 0;

void add(shared\_ptr<AbstractHandler> node);

void add(shared\_ptr<AbstractHandler> node1, shared\_ptr<AbstractHandler> node2, ...);

void add(initializer\_list<shared\_ptr<AbstractHandler>> list);

};

class ConHandler : public AbstractHandler

{

private:

bool condition{ false };

protected:

virtual bool run() override { cout << "Method run;" << endl; return true; }

public:

ConHandler() : ConHandler(false) {}

ConHandler(bool c) : condition(c) { cout << "Constructor;" << endl; }

virtual ~ConHandler() override { cout << "Destructor;" << endl; }

virtual bool handle() override

{

if (!condition) return next ? next->handle() : false;

return run();

}

};

#pragma region Methods

void AbstractHandler::add(shared\_ptr<AbstractHandler> node)

{

if (next)

next->add(node);

else

next = node;

}

void AbstractHandler::add(shared\_ptr<AbstractHandler> node1, shared\_ptr<AbstractHandler> node2, ...)

{

for (Default\* ptr = &node1; \*ptr; ++ptr)

add(\*ptr);

}

void AbstractHandler::add(initializer\_list<shared\_ptr<AbstractHandler>> list)

{

for (auto elem : list)

add(elem);

}

#pragma endregion

int main()

{

shared\_ptr<AbstractHandler> chain(new ConHandler());

chain->add(

{

shared\_ptr<AbstractHandler>(new ConHandler(false)),

shared\_ptr<AbstractHandler>(new ConHandler(true)),

shared\_ptr<AbstractHandler>(new ConHandler(true)),

AbstractHandler::Default()

}

);

cout << "Result = " << chain->handle() << ";" << endl;;

}

Пример 09.25. Подписчик-издатель (Publish-Subscribe).

# include <iostream>

# include <memory>

# include <vector>

using namespace std;

class Subscriber;

using Reseiver = Subscriber;

class Publisher

{

private:

using Action = void(Reseiver::\*)();

using Pair = pair<shared\_ptr<Reseiver>, Action>;

vector<Pair> callback;

public:

void subscribe(shared\_ptr<Reseiver> r, Action a);

void run();

};

class Subscriber

{

public:

virtual ~Subscriber() = default;

virtual void method() = 0;

};

class ConSubscriber : public Subscriber

{

public:

virtual void method() override { cout << "method;" << endl; }

};

#pragma region Methods Publisher

void Publisher::subscribe(shared\_ptr<Reseiver> r, Action a)

{

Pair pr(r, a);

callback.push\_back(pr);

}

void Publisher::run()

{

cout << "Run:" << endl;

for (auto elem : callback)

((\*elem.first).\*(elem.second))();

}

#pragma endregion

int main()

{

shared\_ptr<Subscriber> subscriber(new ConSubscriber());

shared\_ptr<Publisher> publisher(new Publisher());

publisher->subscribe(subscriber, &Subscriber::method);

publisher->run();

}

Пример 09.24. Посредник (Mediator).

# include <iostream>

# include <memory>

# include <list>

# include <vector>

using namespace std;

class Message {}; // Request

class Mediator;

class Colleague

{

private:

weak\_ptr<Mediator> mediator;

public:

virtual ~Colleague() = default;

void setMediator(shared\_ptr<Mediator> mdr) { mediator = mdr; }

virtual bool send(shared\_ptr<Message> msg);

virtual void receive(shared\_ptr<Message> msg) = 0;

};

class ColleagueLeft : public Colleague

{

public:

virtual void receive(shared\_ptr<Message> msg) override { cout << "Right - > Left;" << endl; }

};

class ColleagueRight : public Colleague

{

public:

virtual void receive(shared\_ptr<Message> msg) override { cout << "Left - > Right;" << endl; }

};

class Mediator

{

protected:

list<shared\_ptr<Colleague>> colleagues;

public:

virtual ~Mediator() = default;

virtual bool send(const Colleague\* coleague, shared\_ptr<Message> msg) = 0;

static bool add(shared\_ptr<Mediator> mediator, shared\_ptr<Colleague> colleague, ...);

};

class ConMediator : public Mediator

{

public:

virtual bool send(const Colleague\* coleague, shared\_ptr<Message> msg) override;

};

#pragma region Methods Colleague

bool Colleague::send(shared\_ptr<Message> msg)

{

shared\_ptr<Mediator> mdr = mediator.lock();

return mdr ? mdr->send(this, msg) : false;

}

#pragma endregion

#pragma region Methods Mediator

bool Mediator::add(shared\_ptr<Mediator> mediator, shared\_ptr<Colleague> colleague, ...)

{

if (!mediator || !colleague) return false;

for (shared\_ptr<Colleague>\* ptr = &colleague; \*ptr; ++ptr)

{

mediator->colleagues.push\_back(\*ptr);

(\*ptr)->setMediator(mediator);

}

return true;

}

bool ConMediator::send(const Colleague\* colleague, shared\_ptr<Message> msg)

{

bool flag = false;

for (auto& elem : colleagues)

{

if (dynamic\_cast<const ColleagueLeft\*>(colleague) && dynamic\_cast<ColleagueRight\*>(elem.get()))

{

elem->receive(msg);

flag = true;

}

else if (dynamic\_cast<const ColleagueRight\*>(colleague) && dynamic\_cast<ColleagueLeft\*>(elem.get()))

{

elem->receive(msg);

flag = true;

}

}

return flag;

}

#pragma endregion

int main()

{

shared\_ptr<Mediator> mediator(new ConMediator());

shared\_ptr<Colleague> col1(new ColleagueLeft());

shared\_ptr<Colleague> col2(new ColleagueRight());

shared\_ptr<Colleague> col3(new ColleagueLeft());

shared\_ptr<Colleague> col4(new ColleagueLeft());

Mediator::add(mediator, col1, col2, col3, col4, shared\_ptr<Colleague>());

shared\_ptr<Message> msg(new Message());

col1->send(msg);

col2->send(msg);

}

Пример 09.22. Посетитель (Visitor).

# include <iostream>

# include <memory>

# include <vector>

using namespace std;

class Circle;

class Rectangle;

class Visitor

{

public:

virtual ~Visitor() = default;

virtual void visit(Circle& ref) = 0;

virtual void visit(Rectangle& ref) = 0;

};

class Shape

{

public:

virtual ~Shape() = default;

virtual void accept(shared\_ptr<Visitor> visitor) = 0;

};

class Circle : public Shape

{

public:

virtual void accept(shared\_ptr<Visitor> visitor) override { visitor->visit(\*this); }

};

class Rectangle : public Shape

{

public:

virtual void accept(shared\_ptr<Visitor> visitor) override { visitor->visit(\*this); }

};

class ConVisitor : public Visitor

{

public:

virtual void visit(Circle& ref) override { cout << "Circle;" << endl; }

virtual void visit(Rectangle& ref) override { cout << "Rectangle;" << endl; }

};

class Formation

{

public:

static vector<shared\_ptr<Shape>> initialization(shared\_ptr<Shape> elem, ...)

{

vector<shared\_ptr<Shape>> vec;

for (shared\_ptr<Shape>\* ptr = &elem; \*ptr; ++ptr)

vec.push\_back(\*ptr);

return vec;

}

};

int main()

{

vector<shared\_ptr<Shape>> figure = Formation::initialization(

shared\_ptr<Shape>(new Circle()),

shared\_ptr<Shape>(new Rectangle()),

shared\_ptr<Shape>(new Circle()),

shared\_ptr<Shape>()

);

shared\_ptr<Visitor> visitor(new ConVisitor());

for (auto& elem : figure)

elem->accept(visitor);

}

Пример 09.26. Опекун (Memento).

# include <iostream>

# include <memory>

# include <list>

using namespace std;

class Memento;

class Caretaker

{

public:

unique\_ptr<Memento> getMemento();

void setMemento(unique\_ptr<Memento> memento);

private:

list<unique\_ptr<Memento>> mementos;

};

class Originator

{

public:

Originator(int s) : state(s) {}

const int getState() const { return state; }

void setState(int s) { state = s; }

std::unique\_ptr<Memento> createMemento() { return make\_unique<Memento>(\*this); }

void restoreMemento(std::unique\_ptr<Memento> memento);

private:

int state;

};

class Memento

{

friend class Originator;

public:

Memento(Originator o) : originator(o) {}

private:

void setOriginator(Originator o) { originator = o; }

Originator getOriginator() { return originator; }

private:

Originator originator;

};

#pragma region Methods Caretaker

void Caretaker::setMemento(unique\_ptr<Memento> memento)

{

mementos.push\_back(move(memento));

}

unique\_ptr<Memento> Caretaker::getMemento() {

unique\_ptr<Memento> last = move(mementos.back());

mementos.pop\_back();

return last;

}

#pragma endregion

#pragma region Method Originator

void Originator::restoreMemento(std::unique\_ptr<Memento> memento)

{

\*this = memento->getOriginator();

}

#pragma endregion

int main()

{

auto originator = make\_unique<Originator>(1);

auto caretaker = make\_unique<Caretaker>();

cout << "State = " << originator->getState() << endl;

caretaker->setMemento(originator->createMemento());

originator->setState(2);

cout << "State = " << originator->getState() << endl;

caretaker->setMemento(originator->createMemento());

originator->setState(3);

cout << "State = " << originator->getState() << endl;

caretaker->setMemento(originator->createMemento());

originator->restoreMemento(caretaker->getMemento());

cout << "State = " << originator->getState() << endl;

originator->restoreMemento(caretaker->getMemento());

cout << "State = " << originator->getState() << std::endl;

originator->restoreMemento(caretaker->getMemento());

cout << "State = " << originator->getState() << std::endl;

}

Пример 09.27. Шаблонный метод (Template Method).

# include <iostream>

using namespace std;

class AbstractClass

{

public:

void templateMethod()

{

primitiveOperation();

concreteOperation();

hook();

}

protected:

virtual void primitiveOperation() = 0;

void concreteOperation() { cout << "concreteOperation;" << endl; }

virtual void hook() { cout << "hook Base;" << endl; }

};

class ConClassA : public AbstractClass

{

protected:

virtual void primitiveOperation() override { cout << "primitiveOperation A;" << endl; }

};

class ConClassB : public AbstractClass

{

protected:

virtual void primitiveOperation() override { cout << "primitiveOperation B;" << endl; }

void hook() { cout << "hook B;" << endl; }

};

int main()

{

ConClassA ca;

ConClassB cb;

ca.templateMethod();

cb.templateMethod();

}

Пример 09.21. Свойство (Property).

# include <iostream>

# include <memory>

using namespace std;

template <typename Owner, typename Type>

class Property

{

private:

using Getter = Type (Owner::\*)() const;

using Setter = void (Owner::\*)(const Type&);

Owner\* owner;

Getter methodGet;

Setter methodSet;

public:

Property() = default;

Property(Owner\* owr, Getter getmethod, Setter setmethod) : owner(owr), methodGet(getmethod), methodSet(setmethod) {}

void init(Owner\* owr, Getter getmethod, Setter setmethod)

{

owner = owr;

methodGet = getmethod;

methodSet = setmethod;

}

operator Type() { return (owner->\*methodGet)(); } // Getter

void operator=(const Type& data) { (owner->\*methodSet)(data); } // Setter

// Property(const Property&) = delete;

// Property& operator=(const Property&) = delete;

};

class Object

{

private:

double value;

public:

Object(double v) : value(v) { Value.init(this, &Object::getValue, &Object::setValue); }

double getValue() const { return value; }

void setValue(const double& v) { value = v; }

Property<Object, double> Value;

};

int main()

{

Object obj(5.);

cout << "value = " << obj.Value << endl;

obj.Value = 10.;

cout << "value = " << obj.Value << endl;

unique\_ptr<Object> ptr(new Object(15.));

cout << "value =" << ptr->Value << endl;

obj = \*ptr;

obj.Value = ptr->Value;

}