Пример 09.10. Адаптер (Adapter).

# include <iostream>

# include <memory>

using namespace std;

class Adapter

{

public:

virtual ~Adapter() = default;

virtual void request() = 0;

};

class BaseAdaptee

{

public:

virtual ~BaseAdaptee() = default;

virtual void specificRequest() = 0;

};

class ConAdapter : public Adapter

{

private:

shared\_ptr<BaseAdaptee> adaptee;

public:

ConAdapter(shared\_ptr<BaseAdaptee> ad) : adaptee(ad) {}

virtual void request() override;

};

class ConAdaptee : public BaseAdaptee

{

public:

virtual void specificRequest() override { cout << "Method ConAdaptee;" << endl; }

};

#pragma region Methods

void ConAdapter::request()

{

cout << "Adapter: ";

if (adaptee)

{

adaptee->specificRequest();

}

else

{

cout << "Empty!" << endl;

}

}

#pragma endregion

int main()

{

shared\_ptr<BaseAdaptee> adaptee(new ConAdaptee());

shared\_ptr<Adapter> adapter(new ConAdapter(adaptee));

adapter->request();

}

Пример 09.32. Объект расширения (Adapter - Extension Object). Расширение интерфейса.

# include <iostream>

# include <memory>

using namespace std;

class Extension;

class Subject

{

public:

virtual ~Subject() = default;

virtual void operation() = 0;

virtual shared\_ptr<Extension> getExtension() = 0;

};

class Extension

{

protected:

Subject\* owner;

public:

Extension(Subject\* ow) : owner(ow) {}

virtual ~Extension() = default;

virtual void doSomething() = 0;

};

class ConExtension : public Extension

{

public:

using Extension::Extension;

virtual void doSomething() override

{

cout << "Method ConExtension;" << endl;

owner->operation();

}

};

class ConSubject : public Subject

{

private:

shared\_ptr<Extension> extension;

public:

ConSubject() { extension = shared\_ptr<Extension>(new ConExtension(this)); }

virtual void operation() override { cout << "Method ConSubject;" << endl; }

virtual shared\_ptr<Extension> getExtension() override { return extension; }

};

int main()

{

shared\_ptr<Subject> subject( new ConSubject());

shared\_ptr<Extension> extension = subject->getExtension();

extension->doSomething();

}

Пример 09.11. Декоратор (Decorator).

# include <iostream>

# include <memory>

using namespace std;

class Component

{

public:

virtual ~Component() = default;

virtual void operation() = 0;

};

class ConComponent : public Component

{

public:

virtual void operation() override { cout << "ConComponent; "; }

};

class Decorator : public Component

{

protected:

shared\_ptr<Component> component;

public:

Decorator(shared\_ptr<Component> comp) : component(comp) {}

};

class ConDecorator : public Decorator

{

public:

using Decorator::Decorator;

virtual void operation() override;

};

#pragma region Method

void ConDecorator::operation()

{

if (component)

{

component->operation();

cout << "ConDecorator ";

}

}

#pragma endregion

int main()

{

shared\_ptr<Component> component(new ConComponent());

shared\_ptr<Component> decorator1(new ConDecorator(component));

decorator1->operation();

cout << ";" << endl;

shared\_ptr<Component> decorator2(new ConDecorator(decorator1));

decorator2->operation();

cout << ";" << endl;

}

Пример 09.12. Компоновщик (Composite).

# include <iostream>

# include <memory>

# include <vector>

# include <iterator>

using namespace std;

class Component;

using VectorComponent = vector<shared\_ptr<Component>>;

using IteratorComponent = VectorComponent::const\_iterator;

class Component

{

public:

virtual ~Component() = default;

virtual void operation() = 0;

virtual bool isComposite() const { return false; }

virtual bool add(shared\_ptr<Component> comp) { return false; }

virtual bool remove(const IteratorComponent& it) { return false; }

virtual IteratorComponent begin() const { return IteratorComponent(); }

virtual IteratorComponent end() const { return IteratorComponent(); }

};

class Figure : public Component

{

public:

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

};

class Camera : public Component

{

public:

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

};

class Composite : public Component

{

private:

VectorComponent vec;

public:

Composite() = default;

Composite(shared\_ptr<Component> first, ...);

virtual void operation() override;

virtual bool isComposite() const override { return true; }

virtual bool add(shared\_ptr<Component> comp) { vec.push\_back(comp); return true; }

virtual bool remove(const IteratorComponent& it) { vec.erase(it); return true; }

virtual IteratorComponent begin() const override { return vec.begin(); }

virtual IteratorComponent end() const override { return vec.end(); }

};

#pragma region Methods

Composite::Composite(shared\_ptr<Component> first, ...)

{

for (shared\_ptr<Component>\* ptr = &first; \*ptr; ++ptr)

vec.push\_back(\*ptr);

}

void Composite::operation()

{

cout << "Composite method:" << endl;

for (auto elem : vec)

elem->operation();

}

#pragma endregion

int main()

{

using Default = shared\_ptr<Component>;

shared\_ptr<Component> fig1(new Figure()), fig2(new Figure), cam1(new Camera()), cam2(new Camera());

shared\_ptr<Component> composite1(new Composite(fig1, cam1, fig2, cam2, Default()));

composite1->operation();

cout << endl;

IteratorComponent it = composite1->begin();

composite1->remove(++it);

composite1->operation();

cout << endl;

shared\_ptr<Component> composite2(new Composite(shared\_ptr<Component>(new Figure()), composite1, Default()));

composite2->operation();

}

Пример 09.13. Заместитель (Proxy).

# include <iostream>

# include <memory>

# include <map>

# include <random>

using namespace std;

class Subject

{

public:

virtual ~Subject() = default;

virtual pair<bool, double> request(size\_t index) = 0;

virtual bool changed() { return true; }

};

class RealSubject : public Subject

{

private:

bool flag{ false };

size\_t counter{ 0 };

public:

virtual pair<bool, double> request(size\_t index) override;

virtual bool changed() override;

};

class Proxy : public Subject

{

protected:

shared\_ptr<RealSubject> realsubject;

public:

Proxy(shared\_ptr<RealSubject> real) : realsubject(real) {}

};

class ConProxy : public Proxy

{

private:

map<size\_t, double> cache;

public:

using Proxy::Proxy;

virtual pair<bool, double> request(size\_t index) override;

};

#pragma region Methods

bool RealSubject::changed()

{

if (counter == 0)

{

flag = true;

}

if (++counter == 7)

{

counter = 0;

flag = false;

}

return flag;

}

pair<bool, double> RealSubject::request(size\_t index)

{

random\_device rd;

mt19937 gen(rd());

return pair<bool, double>(true, generate\_canonical<double, 10>(gen));

}

pair<bool, double> ConProxy::request(size\_t index)

{

pair<bool, double> result;

if (!realsubject)

{

cache.clear();

result = pair<bool, double>(false, 0.);

}

if (!realsubject->changed())

{

cache.clear();

result = realsubject->request(index);

cache.insert(map<size\_t, double>::value\_type(index, result.second));

}

else

{

map<size\_t, double>::const\_iterator it = cache.find(index);

if (it != cache.end())

{

result = pair<bool, double>(true, it->second);

}

else

{

result = realsubject->request(index);

cache.insert(map<size\_t, double>::value\_type(index, result.second));

}

}

return result;

}

#pragma endregion

int main()

{

shared\_ptr<RealSubject> subject(new RealSubject());

shared\_ptr<Subject> proxy(new ConProxy(subject));

for (size\_t i = 0; i < 21; ++i)

{

cout << "( " << i + 1 << ", " << proxy->request(i % 3).second << " )" << endl;

if ((i + 1) % 3 == 0)

cout << endl;

}

}

Пример 09.14. Мост (Bridge).

# include <iostream>

# include <memory>

using namespace std;

class Implementor

{

public:

virtual ~Implementor() = default;

virtual void operationImp() = 0;

};

class Abstraction

{

protected:

shared\_ptr<Implementor> implementor;

public:

Abstraction(shared\_ptr<Implementor> imp) : implementor(imp) {}

virtual ~Abstraction() = default;

virtual void operation() = 0;

};

class ConImplementor : public Implementor

{

public:

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

};

class Entity : public Abstraction

{

public:

using Abstraction::Abstraction;

virtual void operation() override { cout << "Entity: "; implementor->operationImp(); }

};

int main()

{

shared\_ptr<Implementor> implementor(new ConImplementor());

shared\_ptr<Abstraction> abstraction(new Entity(implementor));

abstraction->operation();

}