Прототип. Структурные паттерны.

Прототип.

Суть паттерна. *Прототип* — это порождающий паттерн проектирования, который позволяет копировать объекты, не вдаваясь в подробности их реализации.

Проблема. У вас есть объект, который нужно скопировать. Как это сделать? Нужно создать пустой объект такого же класса, а затем поочерёдно скопировать значения всех полей из старого объекта в новый.

Не каждый объект удастся скопировать таким образом, ведь часть его состояния может быть приватной, а значит — недоступной для остального кода программы.

Но есть и другая проблема. Копирующий код станет зависим от классов копируемых объектов. Ведь, чтобы перебрать *все* поля объекта, нужно привязаться к его классу. Из-за этого вы не сможете

Решение. Паттерн Прототип поручает создание копий самим копируемым объектам. Он вводит общий интерфейс для всех объектов, поддерживающих клонирование. Это позволяет копировать объекты, не привязываясь к их конкретным классам. Обычно такой интерфейс имеет всего один метод *clone*.

Реализация этого метода в разных классах очень схожа. Метод создаёт новый объект текущего класса и копирует в него значения всех полей собственного объекта. Так получится скопировать даже приватные поля, так как большинство языков программирования разрешает доступ к приватным полям любого объекта текущего класса.

Объект, который копируют, называется *прототипом* (откуда и название паттерна). Когда объекты программы содержат сотни полей и тысячи возможных конфигураций, прототипы могут служить своеобразной альтернативой созданию подклассов.

1. В вашей игре используется покадровая анимация. Каждая анимация может встречаться несколько раз. Создание анимации связано с доступом к файлу, вырезанию кадров и прочими сложными операциями. Возможно *загрузить*анимации в программу только один раз, а затем *клонировать* их — это должно быть более эффективно, чем создание анимации напрямую;
2. Вам нужна возможность добавлять новые типы существ прямо во время игры. Все существа игры описаны в файлах, в определенном формате. Обращаться к файлу всякий раз, когда требуется добавить особь в игру — не оптимально. Эффективней *загрузить* нужный тип особи с

Реализация паттерна

```
class Meal {
public:
virtual ~Meal();
virtual void eat() = 0;
virtual Meal *clone() const = 0;
//...
};
class Spaghetti : public Meal {
public:
Spaghetti( const Spaghetti &);
void eat();
Spaghetti *clone() const { return new Spaghetti( *this ); }
//... };
```

Применимость

- Когда ваш код не должен зависеть от классов копируемых объектов.

- Когда вы имеете уйму подклассов, которые отличаются начальными значениями полей. Кто-то мог создать все эти классы, чтобы иметь возможность легко порождать объекты с определённой конфигурацией.

Преимущества и недостатки

- «+»:

Позволяет клонировать объекты, не привязываясь к их конкретным классам.

Меньше повторяющегося кода инициализации объектов.

Ускоряет создание объектов.

Альтернатива созданию подклассов для конструирования сложных объектов.

- «-»

Сложно клонировать составные объекты, имеющие ссылки на другие объекты.

Структурные паттерны

Структурные шаблоны — шаблоны проектирования, в которых рассматривается вопрос о том, как из классов и объектов образуются более крупные структуры. При этом могут использоваться следующие механизмы:

- Наследование - базовый класс определяет интерфейс, а подклассы - реализацию. Структуры на основе наследования получаются статичными.
- Композиция - структуры строятся путем объединения объектов некоторых классов. Композиция позволяет получать структуры, которые можно изменять во время выполнения.

Список структурных паттернов

- адаптер (Adapter);
- мост (Bridge);
- компоновщик (Composite);
- декоратор (Decorator);
- фасад (Facade);
- приспособленец (Flyweight);
- заместитель (Proxy).

Адаптер

Суть паттерна. *Адаптер* — это структурный паттерн проектирования, который позволяет объектам с несовместимыми интерфейсами работать вместе.

Проблема. Представим, что у вас на карте памяти есть какие-то изображения и вам надо перенести их на ваш компьютер. Чтобы это сделать, вам нужен какой-то адаптер, который совместим с портами вашего компьютера. В этом случае карт-ридер — это адаптер. Другим примером будет блок питания. Вилку с тремя ножками нельзя вставить в розетку с двумя отверстиями. Для того, чтобы она подошла, надо использовать адаптер. Ещё одним примером будет переводчик,

Решение. Вы можете создать *адаптер*. Это объект-переводчик, который трансформирует интерфейс или данные одного объекта в такой вид, чтобы он стал понятен другому объекту.

При этом адаптер оборачивает один из объектов, так что другой объект даже не знает о наличии первого. Например, вы можете обернуть объект, работающий в метрах, адаптером, который бы конвертировал данные в футы.

Адаптеры могут не только переводить данные из одного формата в другой, но и помогать объектам с разными интерфейсами работать сообща. Это работает так:

1. Адаптер имеет интерфейс, который совместим с одним из объектов.
2. Поэтому этот объект может свободно вызывать методы адаптера.
3. Адаптер получает эти вызовы и перенаправляет их второму объекту, но уже в том формате и последовательности, которые понятны второму объекту.

Иногда возможно создать даже *двухсторонний адаптер*, который работал бы в обе стороны.

Реализация. Задание

Пусть мы разрабатываем систему климат-контроля, предназначенной для автоматического поддержания температуры окружающего пространства в заданных пределах. Важным компонентом такой системы является температурный датчик, с помощью которого измеряют температуру окружающей среды для последующего анализа. Для этого датчика уже имеется готовое программное обеспечение от сторонних разработчиков, представляющее собой некоторый класс с соответствующим интерфейсом. Однако использовать этот класс непосредственно не удастся, так как показания датчика снимаются в градусах Фаренгейта.

Нужен адаптер, преобразующий температуру в шкалу Цельсия.

Классическая реализация

```
#include <iostream>
using namespace std;

// Уже существующий класс температурного датчика окружающей среды
class FahrenheitSensor
{
  public:
    // Получить показания температуры в градусах Фаренгейта
    float getFahrenheitTemp() {
      float t = 32.0;
      // ... какой то код
      return t;
    }
};

class Sensor
{
  public:
    virtual ~Sensor() {}
    virtual float getTemperature() = 0;
};
```

```
class Adapter : public Sensor
{
  public:
    Adapter( FahrenheitSensor* p ) : p_fsensor(p) {}
   ~Adapter() {
      delete p_fsensor;
    }
    float getTemperature() {
      return (p_fsensor->getFahrenheitTemp()-32.0)*5.0/9.0;
    }
  private:
    FahrenheitSensor* p_fsensor;
};

int main()
{
  Sensor* p = new Adapter( new FahrenheitSensor);
  cout << "Celsius temperature = " << p->getTemperature() << endl;
  delete p;
  return 0;
}
```

Закрытое наследование

Пусть наш температурный датчик системы климат-контроля поддерживает функцию юстировки для получения более точных показаний. Эта функция не является обязательной для использования, поэтому соответствующий метод adjust() объявлен разработчиками защищенным в существующем классе FahrenheitSensor.

Разрабатываемая нами система должна поддерживать настройку измерений. Так как доступ к защищенному методу через указатель или ссылку запрещен, то классическая реализация паттерна Adapter здесь уже не подходит. Единственное решение - наследовать от класса FahrenheitSensor. Интерфейс этого класса должен оставаться недоступным пользователю, поэтому

```
class FahrenheitSensor
{
  public:
    float getFahrenheitTemp() {
      float t = 32.0;
      // ...
      return t;
    }
  protected:
    void adjust() {} // Настройка датчика (защищенный метод)
};

class Sensor
{
  public:
    virtual ~Sensor() {}
    virtual float getTemperature() = 0;
    virtual void adjust() = 0;
};
```

```
class Adapter : public Sensor, private FahrenheitSensor
{
  public:
    Adapter() { }
    float getTemperature() {
      return (getFahrenheitTemp()-32.0)*5.0/9.0;
    }
    void adjust() {
      FahrenheitSensor::adjust();
    }
};

int main()
{
  Sensor * p = new Adapter();
  p->adjust();
  cout << "Celsius temperature = " << p->getTemperature() << endl;
  delete p;
  return 0;
}
```

Применимость

- Когда вы хотите использовать сторонний класс, но его интерфейс не соответствует остальному коду приложения.

- Когда вам нужно использовать несколько существующих подклассов, но в них не хватает какой-то общей функциональности, причём расширить суперкласс вы не можете.

Преимущества и недостатки

- «+»

Отделяет и скрывает от клиента подробности преобразования различных интерфейсов.

- «-»

Усложняет код программы из-за введения дополнительных классов.

Задача преобразования интерфейсов может оказаться непростой в случае, если клиентские вызовы и (или) передаваемые параметры не имеют функционального соответствия в адаптируемом объекте.

Мост

Суть паттерна. *Мост* — это структурный паттерн проектирования, который разделяет один или несколько классов на две отдельные иерархии — абстракцию и реализацию, позволяя изменять их независимо друг от друга.

Проблема. У вас есть класс геометрических *Фигур*, который имеет подклассы *Круг* и *Квадрат*. Вы хотите расширить иерархию фигур по цвету, то есть иметь **Красные** и **Синие** фигуры. Но чтобы всё это объединить, вам придётся создать 4 комбинации подклассов,
вроде *СинииеКруги* и *КрасныеКвадраты*. При добавлении новых видов фигур и цветов количество комбинаций будет расти в геометрической прогрессии. Например, чтобы ввести в программу фигуры треугольников, придётся создать сразу два новых подкласса треугольников под каждый цвет. После этого новый цвет потребует создания уже трёх классов для всех видов фигур. Чем дальше,

Решение. Корень проблемы заключается в том, что мы пытаемся расширить классы фигур сразу в двух независимых плоскостях — по виду и по цвету. Именно это приводит к разрастанию дерева классов.

Паттерн Мост предлагает заменить наследование композицией. Для этого нужно выделить одну из таких «плоскостей» в отдельную иерархию и ссылаться на объект этой иерархии, вместо хранения его состояния и поведения внутри одного класса.

Таким образом, мы можем сделать Цвет отдельным классом с подклассами *Красный* и *Синий*. Класс *Фигур* получит ссылку на объект Цвета и сможет делегировать ему работу, если потребуется. Такая связь и станет мостом между Фигурами и Цветом. При добавлении новых классов цветов не потребуется трогать классы фигур и наоборот.

Абстракция и Реализация

Абстракция — это образный слой управления чем-либо. Он не делает работу самостоятельно, а делегирует её слою *реализации* (иногда называемому *платформой*).

Если говорить о реальных программах, то абстракцией может выступать графический интерфейс программы (GUI), а реализацией — низкоуровневый код операционной системы (API), к которому графический интерфейс обращается по реакции на действия пользователя.

Можно развивать программу в двух разных направлениях:

- иметь несколько видов GUI (например, для простых пользователей и администраторов);
- поддерживать много видов API (например, работать под Windows, Linux и MacOS).

Такая программа может выглядеть как один большой клубок кода, в котором намешаны условные операторы слоёв GUI и API.

Вы можете попытаться структурировать этот хаос, создав для каждой вариации интерфейса-платформы свои подклассы. Но такой подход приведёт к росту классов комбинаций, и с каждой новой платформой их будет всё больше.

Мы можем решить эту проблему, применив Мост. Паттерн предлагает распутать этот код, разделив его на две части:

- Абстракцию: слой графического интерфейса приложения.
- Реализацию: слой взаимодействия с операционной системой.

Абстракция будет делегировать работу одному из объектов реализаций. Причём, реализации можно будет взаимозаменять, но только при условии, что все они будут следовать общему интерфейсу.

Таким образом, вы сможете изменять графический интерфейс приложения, не трогая низкоуровневый код работы с операционной системой. И наоборот, вы сможете добавлять поддержку новых операционных систем, создавая подклассы реализации, без необходимости менять классы графического интерфейса.

```
#include <iostream>
using namespace std;

class Drawer {
public:
   virtual void drawCircle(int x, int y, int radius) = 0;
};

class SmallCircleDrawer : public Drawer {
public:
   const double radiusMultiplier = 0.25;

   void drawCircle(int x, int y, int radius) override
   {
      cout << "Small circle center " << x << ", " << y << " radius = " <<
            radius*radiusMultiplier << endl;
   }
};

class LargeCircleDrawer : public Drawer {
public:
   const double radiusMultiplier = 10;

   void drawCircle(int x, int y, int radius) override
   {
      cout << "Large circle center " << x << ", " << y << " radius = " <<
            radius*radiusMultiplier << endl;
   }
};
```

```
class Shape {
protected:
   Drawer* drawer;
public:
   Shape(Drawer* drw) {drawer = drw;}
   Shape() {}
   virtual void draw() = 0;
   virtual void enlargeRadius(int multiplier) = 0;
};

class Circle : public Shape {
   int x, y, radius;
public:
   Circle(int _x, int _y, int _radius, Drawer* drw)
   {
      drawer = drw;
      setX(_x);
      setY(_y);
      setRadius(_radius);
   }
   void draw() override {
      drawer->drawCircle(x, y, radius);
   }
   void enlargeRadius(int multiplier) override {radius *= multiplier;}
   void setX(int _x) {x = _x;}
   void setY(int _y) {y = _y;}
   void setRadius(int _radius) {radius = _radius; }
};
```

```
int main(int argc, char *argv[])
{
    Shape* shapes[2] = {
new Circle(5,10,10, new LargeCircleDrawer()),
new Circle(20,30,100, new SmallCircleDrawer())};
    for (int i =0 ; i < 2; i++)
    {
        shapes[i]->draw();
    }
    return 0;
}
```

Применимость

- Когда вы хотите разделить монолитный класс, который содержит несколько различных реализаций какой-то функциональности (например, если класс может работать с разными системами баз данных).
- Когда класс нужно расширять в двух независимых плоскостях.
- Когда вы хотите, чтобы реализацию можно было бы изменять во время выполнения программы.

Преимущества и недостатки

- «+»

Позволяет строить платформо-независимые программы.

Скрывает лишние или опасные детали реализации от клиентского кода.

Реализует *принцип открытости/закрытости*.

- «-»

Усложняет код программы из-за введения дополнительных классов.

Фасад

Суть паттерна. *Фасад* — это структурный паттерн проектирования, который предоставляет простой интерфейс к сложной системе классов, библиотеке или фреймворку.

Проблема. Вашему коду приходится работать с большим количеством объектов некой сложной библиотеки или фреймворка. Вы должны самостоятельно инициализировать эти объекты, следить за правильным порядком зависимостей и так далее.

В результате бизнес-логика ваших классов тесно переплетается с деталями реализации сторонних классов. Такой код довольно сложно понимать и поддерживать.

Решение. Фасад — это простой интерфейс для работы со сложной подсистемой, содержащей множество классов. Фасад может иметь урезанный интерфейс, не имеющий 100% функциональности, которой можно достичь, используя сложную подсистему напрямую. Но он предоставляет именно те фичи, которые нужны клиенту, и скрывает все остальные.

Фасад полезен, если вы используете какую-то сложную библиотеку со множеством подвижных частей, но вам нужна только часть её возможностей.

К примеру, программа, заливающая видео котиков в социальные сети, может использовать профессиональную библиотеку сжатия видео. Но все, что нужно клиентскому коду этой программы — простой метод encode(filename,

```
#include <iostream>
#include <string>
#include <memory>
#include <string_view>
/** Абстрактный музыкант - не является обязательной составляющей паттерна, введен для упрощения кода */

class Musician {
    const char* name;
public:
    Musician(std::string_view name) {
        this->name = name.data();
    }
    virtual ~Musician() =default;
protected:
    void output(std::string_view text) {
        std::cout << this->name << " " << text << "." << std::endl;
    }
};
```

```
class Vocalist: public Musician {
public:
   Vocalist(std::string_view name): Musician(name) {}
   void singCouplet(int coupletNumber) {
      std::string text = "спел куплет №";
      text += std::to_string(coupletNumber);
      output(text); }
   void singChorus() {
      output("спел припев");}
};

class Guitarist: public Musician {
public:
   Guitarist(std::string_view name): Musician(name) {}
   void playCoolOpening() {
      output("начинает с крутого вступления");}
   void playCoolRiffs() {
      output("играет крутые риффы"); }
   void playAnotherCoolRiffs() {
      output("играет другие крутые риффы"); }
   void playIncrediblyCoolSolo() {
      output("выдает невероятно крутое соло"); }
   void playFinalAccord() {
      output("заканчивает песню мощным аккордом"); }
};
```

```
class Bassist: public Musician {
public:
   Bassist(std::string_view name): Musician(name) {}
   void followTheDrums() {
      output("следует за барабанами");
   }
   void changeRhythm(std::string_view type) {
      std::string text = ("перешел на ритм ");
      text += type;
      text += "a";
      output(text);
   }
   void stopPlaying() {
      output("заканчивает играть");
   }
};

class Drummer: public Musician {
public:
   Drummer(std::string_view name): Musician(name) {}
   void startPlaying() {
      output("начинает играть");
   }
   void stopPlaying() {
      output("заканчивает играть");
   }
};
```

```
/** Фасад, в данном случае - знаменитая рок-группа */
class BlackSabbath {

    std::unique_ptr<Vocalist> vocalist;
    std::unique_ptr<Guitarist> guitarist;
    std::unique_ptr<Bassist> bassist;
    std::unique_ptr<Drummer> drummer;

public:

    BlackSabbath() {
        vocalist = std::make_unique<Vocalist>("Оззи Осборн");
        guitarist = std::make_unique<Guitarist>("Тони
Айомми");
        bassist = std::make_unique<Bassist>("Гизер Батлер");
        drummer = std::make_unique<Drummer>("Билл Уорд");
    }
```

```
void playCoolSong() {
      guitarist->playCoolOpening();
      drummer->startPlaying();
      bassist->followTheDrums();
      guitarist->playCoolRiffs();
      vocalist->singCouplet(1);
      bassist->changeRhythm("припев");
      guitarist->playAnotherCoolRiffs();
      vocalist->singChorus();
      bassist->changeRhythm("куплет");
      guitarist->playCoolRiffs();
      vocalist->singCouplet(2);
      bassist->changeRhythm("припев");
      guitarist->playAnotherCoolRiffs();
      vocalist->singChorus();
      bassist->changeRhythm("куплет");
      guitarist->playIncrediblyCoolSolo();
      guitarist->playCoolRiffs();
      vocalist->singCouplet(3);
      bassist->changeRhythm("припев");
      guitarist->playAnotherCoolRiffs();
      vocalist->singChorus();
      bassist->changeRhythm("куплет");
      guitarist->playCoolRiffs();
      bassist->stopPlaying();
      drummer->stopPlaying();
      guitarist->playFinalAccord();
   }
};
```

```
int main() {
    std::cout << "OUTPUT:" << std::endl;
    BlackSabbath* band = new BlackSabbath();
    band->playCoolSong();
    return 0;
}
```

```
/**
 * OUTPUT:
 * Тони Айомми начинает с крутого вступления.
 * Билл Уорд начинает играть.
 * Гизер Батлер следует за барабанами.
 * Тони Айомми играет крутые риффы.
 * Оззи Осборн спел куплет №1.
 * Гизер Батлер перешел на ритм припева.
 * Тони Айомми играет другие крутые риффы.
 * Оззи Осборн спел припев.
 * Гизер Батлер перешел на ритм куплета.
 * Тони Айомми играет крутые риффы.
 * Оззи Осборн спел куплет №2.
 * Гизер Батлер перешел на ритм припева.
 * Тони Айомми играет другие крутые риффы.
 * Оззи Осборн спел припев.
 * Гизер Батлер перешел на ритм куплета.
 * Тони Айомми выдает невероятно крутое соло.
 * Тони Айомми играет крутые риффы.
 * Оззи Осборн спел куплет №3.
 * Гизер Батлер перешел на ритм припева.
 * Тони Айомми играет другие крутые риффы.
 * Оззи Осборн спел припев.
 * Гизер Батлер перешел на ритм куплета.
 * Тони Айомми играет крутые риффы.
 * Гизер Батлер заканчивает играть.
 * Билл Уорд заканчивает играть.
 * Тони Айомми заканчивает песню мощным аккордом.
 */
```

Применимость

- Когда вам нужно представить простой или урезанный интерфейс к сложной подсистеме.
- Когда вы хотите разложить подсистему на отдельные слои.

Преимущества и недостатки

- «+»

Изолирует клиентов от компонентов сложной подсистемы.

- «-»

Фасад рискует стать «божественным объектом», привязанным ко всем классам программы.