Паттерны проектирования

Порождающие паттерны

Если мы создали объект с помощью оператора new, то необходимо вызвать и delete. Проблему с необходимостью вызывать delete призваны решить порождающие паттерны. Также порождающие паттерны полезны, когда необходимо увеличить независимость.

- Абстрактная фабрика
- Фабричный метод
- Строитель
- Одиночка
- Прототип

Абстрактная фабрика

- Назначение: представляет собой интерфейс для создания семей взаимосвязанных или взаимозависимых объектов, не специфицируя их конкретных классов.
- Плюсы: изолирует конкретные классы ("продукты"), упрощает замену семейств "продуктов", гарантирует сочетаемость "продуктов".
- Минусы: сложно добавить поддержку новых "продуктов".

Можно использовать этот шаблон

- когда система должна оставаться независимой как от процесса создания новых объектов, так и от типов порождаемых объектов. (Оператор new внутри клиента нежелателен)
- когда необходимо создавать группы или семейства взаимозависимых объектов, исключая возможность одновременного использования объектов разных семейств в одном контексте.

```
class Car{
    ...;
public:
    virtual void info() = 0;
};

class Lada : public Car{
public:
    void info(){
        cout << "Лада" << endl;
    }
};

class Toyota : public Car{
public:
    void info(){
        cout << "DEJA VU I HAVE BEEN IN THIS PLACE BEFORE" << endl;
    }
};

class Engine{
public:
```

```
    virtual void getEngine() = 0;
};

class EngineLada : public Engine{
public:
    virtual void getEngine(){
        cout << "Пыхчпых" << endl;
    }
};

class EngineToyota : public Engine{
public:
    virtual void getEngine(){
        cout << "Врумврум" << endl;
    }
};

class FactoryCar{
    public:
    Car *createCar(){
        return getCar();
    }
    Engine *createEngine(){
        return getEngine();
    }
    protected:
    virtual Car *getCar() = 0;
    virtual Engine *getEngine() = 0;
};

class FactoryLada : public FactoryCar{
    protected:
    Car *getCar(){
        return new Lada();
    }
    Engine *getEngine(){
        return new EngineLada();
    }
};

class FactoryToyota : public FactoryCar{
    protected:
    Car *getCar(){
        return new Toyota();
    }
    Engine *getEngine(){
        return new EngineTotota();
    }
};

int main(){
    FactoryCar *currentfactory = NULL;
    Car *mashina = NULL;
```

```
    Engine *dvigatel = NULL;
    FactoryLada f_lada;
    FactoryToyota f_toyota;
    currentfactory = &f_lada;
    mashina = currentfactory->createCar();
    mashina->info(); // Лада
    dvigatel = currentfactory->createEngine();
    dvigatel->info(); // Пыхчпых
    delete mashina;
    delete dvigatel;
    delete currentfactory;
    currentfactory = &f_toyota;
    mashina = currentfactory->createCar();
    mashina->info(); // DEJA VU
    dvigatel = currentfactory->createEngine();
    dvigatel->info(); // Врумврум
    delete mashina;
    delete dvigatel;
    delete currentfactory;
}

```

Фабричный метод