Технологии программирования

Паттерны проектирования. Порождающие паттерны. Factory Method, Abstract Factory, Builder

Александр Михалевич

Паттерн проектирования

ИМЯ

ЗАДАЧА

РЕШЕНИЕ

РЕЗУЛЬТАТЫ

Типы паттернов проектирования

ПОРОЖДАЮЩИЕ

СТРУКТУРНЫЕ

ПОВЕДЕНЧЕСКИЕ

Порождающие паттерны

SINGLETON

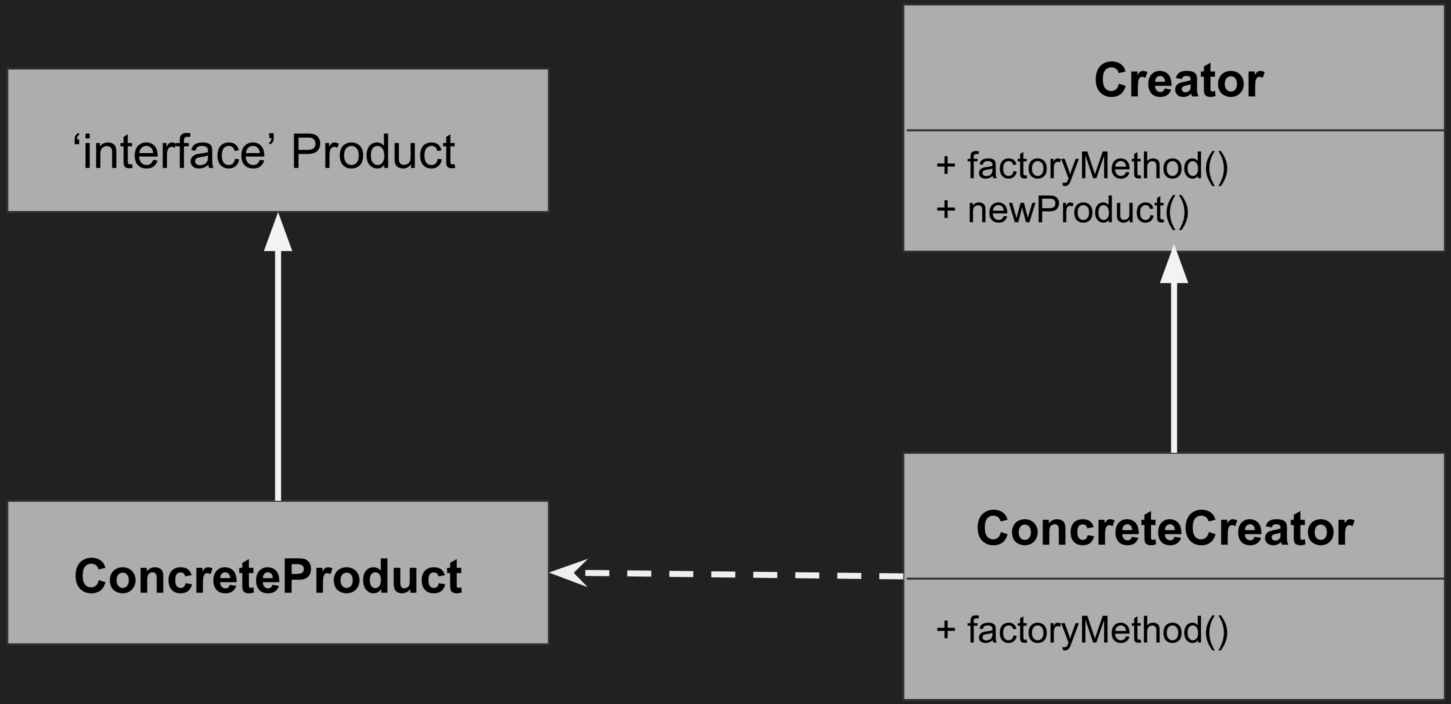
FACTORY METHOD

ABSTRACT FACTORY

BUILDER

PROTOTYPE

Factory method



Factory method

```
class CWarrior {
public:
        virtual void action() = 0;
        virtual ~CWarrior() { }
};
```

```
class CInfantry : CWarrior {
public:
        void action() { … }
};
```

```
class CArcher : CWarrior {
public:
        void action() { … }
};
```

```
class CHorseman : CWarrior {
public:
        void action() { … }
};
```

Factory method

```
class CFactory {
public:
        virtual CWarrior* create() = 0;
        virtual ~CFactory() { }
};
```

```
class CInfantryFactory {
public:
        CWarrior* create() { return new CInfantry; }
        virtual ~CFactory() { }
};
```

```
class CArcherFactory {
public:
        CWarrior* create() { return new CArcher; }
        virtual ~CFactory() { }
};
```

```
class CHorsemanFactory {
public:
        CWarrior* create() { return new CHorseman; }
        virtual ~CFactory() { }
};
```

Factory method

```
…
 CInfantryFactory* infantry_factory = new InfantryFactory;
 CArchersFactory*  archers_factory  = new ArcherFactory;
 CHorsemanFactory*  horseman_factory  = new CHorsemanFactory;

 vector<CWarrior*> v;
 v.push_back( infantry_factory->createWarrior());
 v.push_back( archers_factory->createWarrior());
 v.push_back( horseman_factory->createWarrior());
...
```

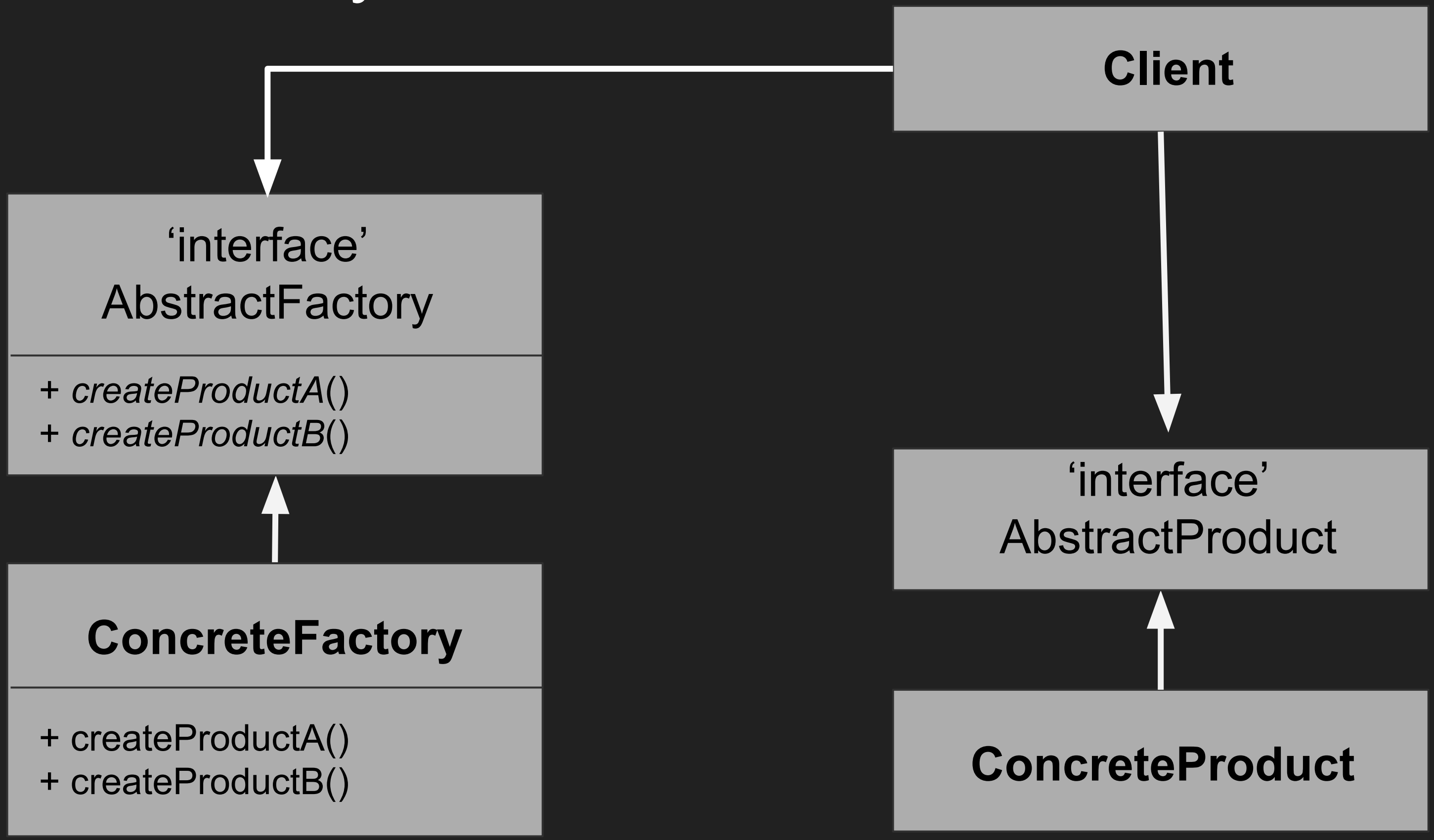
Factory method: параметризованный вариант

```
class CFactory {
public:
        static CWarrior* create(EUnitType type) {
                switch(type) {
                case EUnitType::Infantry:
                        return new CInfantry;
                …
                }
        }
};


…
 vector<CWarrior*> v;
 v.push_back(CFactory::create(EUnitType::Infantry);
 v.push_back(CFactory::create(EUnitType::Archer);

v.push_back(CFactory::create(EUnitType::Horseman);
...
```

Abstract factory



Abstract factory

```
class CRomanInfantry {
public:
        void action() { … };
};

class CCarthaginianInfantry {
public:
        void action() { … };
};
```

```
class CInfantry {
public:
        virtual void action() = 0;
        virtual ~CInfantry() { }
};
```

```
class CArcher {
public:
        virtual void action() = 0;
        virtual ~CArcher() { }
};
```

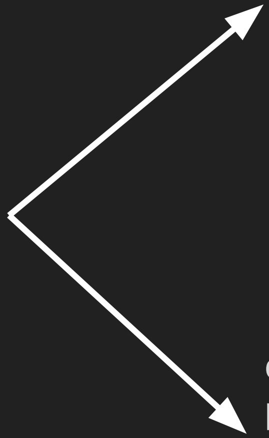
…

```
class CHorseman {
public:
        virtual void action() = 0;
        virtual ~CHorseman() { }
};
```

…

Abstract factory

```
class CArmyFactory {
public:
        virtual CInfantry* create_infantry() = 0;
        virtual CArcher* create_archer() = 0;
        virtual CHorseman* create_horseman() = 0;
        virtual ~CArmyFactory() { }
};
```



```
class CRomanArmyFactory {
public:
        ...
};
```

```
class CCarthaginianArmyFactory {
public:
        ...
};
```

Abstract factory

```
class CArmy {
private:
        vector<CInfantry*> m_infantry;
        vector<CArcher*> m_archer;
        vector<CHorseman*> m_horseman;
public:
        void action() { … }
        void add_infantry() { … }
        void add_archer() { … }
        void add_horseman() { … }
        ~CArmy() { /* delete all units */ }
};
```

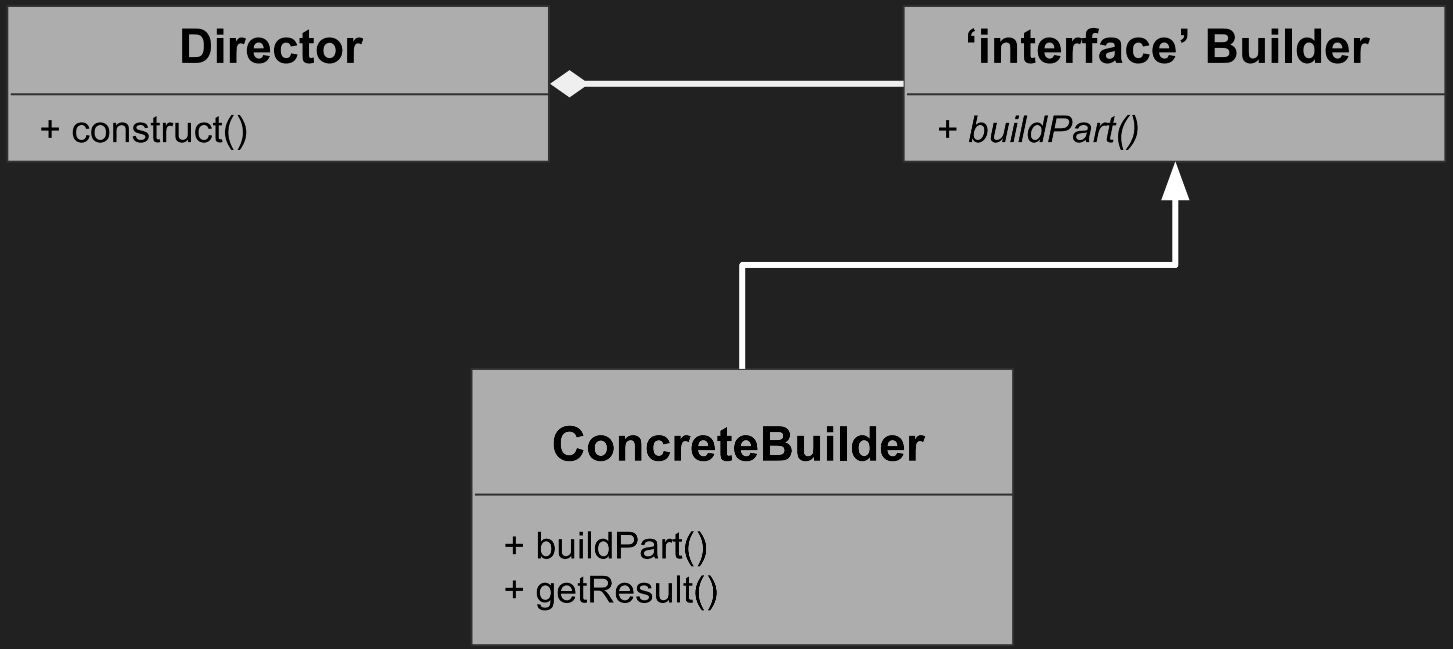
```
class CGame {
public:
        Army* create_army(CArmyFactory& factory) {
                Army* ptr = new Army;
                ptr->add_infantry(factory.create_infantry());
                ptr->add_archer(factory.create_archer());
                ptr->add_horseman(factory.create_horseman());
                return ptr;
        }
};
```

Abstract factory

```
int main() {
	Game game;
	CRomanArmyFactory roman_factory;
	CCarthaginianArmyFactory carthaginian_factory;

	Army* romans = game.create_army(roman_factory);
	Army* carthaginians = game.create_army(carthaginian_factory);
	...
}
```

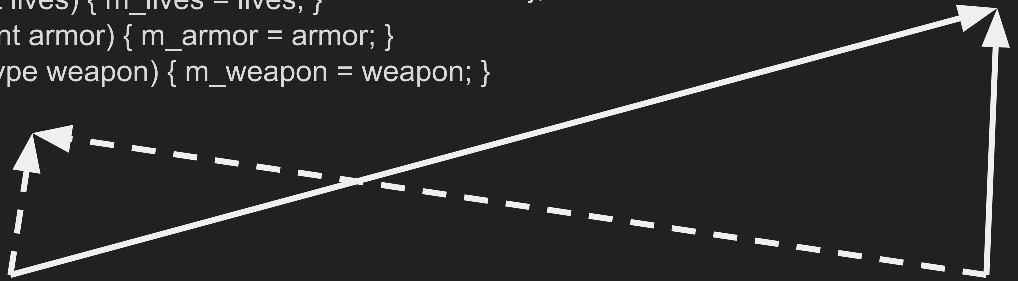
Builder


Director
+ construct()
'interface' Builder
+ *buildPart()*
ConcreteBuilder
+ buildPart()
+ getResult()

Builder

```
class CSpaceship {     // product
private:
        unsigned int m_lives;
        unsigned int m_armor;
        EWeaponType m_weapon;
public:
        void set_lives(unsigned int lives) { m_lives = lives; }
        void set_armor(unsigned int armor) { m_armor = armor; }
        void set_lives(EWeaponType weapon) { m_weapon = weapon; }
};
```

```
class ISpaceshipBuilder {                          // abstract builder
protected:
        std::shared_ptr<CSpaceship> m_spaceship;
public:
        virtual ~ISpaceshipBuilder() { }
        std::shared_ptr<CSpaceship> get_spaceship() { return
m_spaceship; }
        void create_product()  { m_spaceship.reset(new CSpaceship); }
        virtual void build_lives() = 0;
        virtual void build_armor() = 0;
        virtual void build_weapon() = 0;
};
```



```
class CEarthSpaceshipBuilder : public ISpaceshipBuilder {       // concrete builder
public:
        CEarthSpaceshipBuilder() : ISpaceshipBuilder() { }
        ~CEarthSpaceshipBuilder() { }
        void build_lives() { m_spaceship->set_lives(100); }
        void build_armor() { m_spaceship->set_armor(50); }
        void build_weapon() { m_spaceship->set_weapon(EWeaponType::Laser); }
};
```

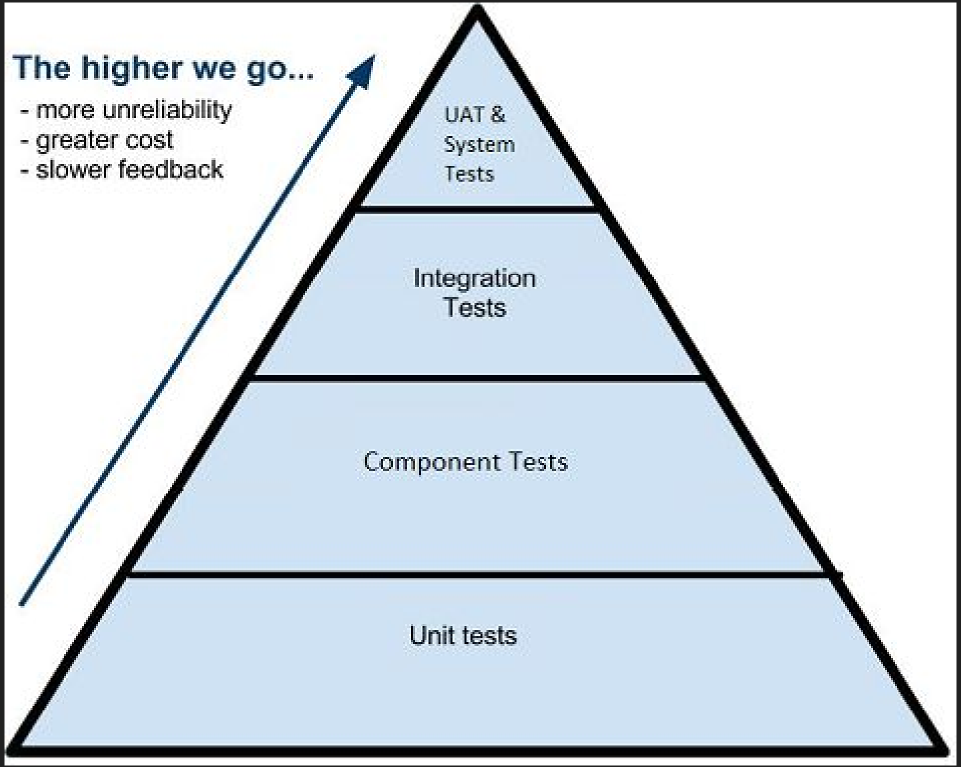
```
class CAlienSpaceshipBuilder : public ISpaceshipBuilder {       // concrete builder
public:
        CAlienSpaceshipBuilder() : ISpaceshipBuilder() { }
        ~CAlienSpaceshipBuilder() { }
        void build_lives() { m_spaceship->set_lives(50); }
        void build_armor() { m_spaceship->set_armor(100); }
        void build_weapon() { m_spaceship->set_weapon(EWeaponType::EM); }
};
```

Builder

```
class ISpaceshipBuilder {                              // abstract builder
protected:
        std::shared_ptr<CSpaceship> m_spaceship;
public:
        ISpaceshipBuilder() { }
        virtual ~ISpaceshipBuilder() { }
        std::shared_ptr<CSpaceship> get_spaceship() { return m_spaceship; }
        void create_product()  { m_spaceship.reset(new CSpaceship); }
        virtual void build_lives() = 0;
        virtual void build_armor() = 0;
        virtual void build_weapon() = 0;
};
```

```
class CShipyard {       // director
private:
        ISpaceshipBuilder* m_spaceship_builder;
public:
        CShipyard() : m_spaceship_builder(NULL) { }
        ~CShipyard() { }
        void set_ship_builder(ISpaceshipBuilder* builder) { m_spaceship_builder = builder; }
        std::shared_ptr<CSpaceship> get_ship() { return m_spaceship_builder->get_spaceship(); }
        void construct_ship() {
                m_spaceship_builder->create_product();
                m_spaceship_builder->build_lives();
                m_spaceship_builder->build_armor();
                m_spaceship_builder->build_weapon();
        }
};
```

Пару слов о unit-тестировании



Пару слов о unit-тестировании

- **Цель:** изолировать отдельные части программы и показать, что по отдельности эти части работоспособны.
- Выполняется программистами

Пример теста с использованием gtest

```
#include <gtest/gtest.h>
...
TEST(TestCase, TestName)
{
    std::string reference_str = “Correct text”;
    str::string result = do_some_stuff();
    EXPECT_EQ(reference_str, result);
}
```