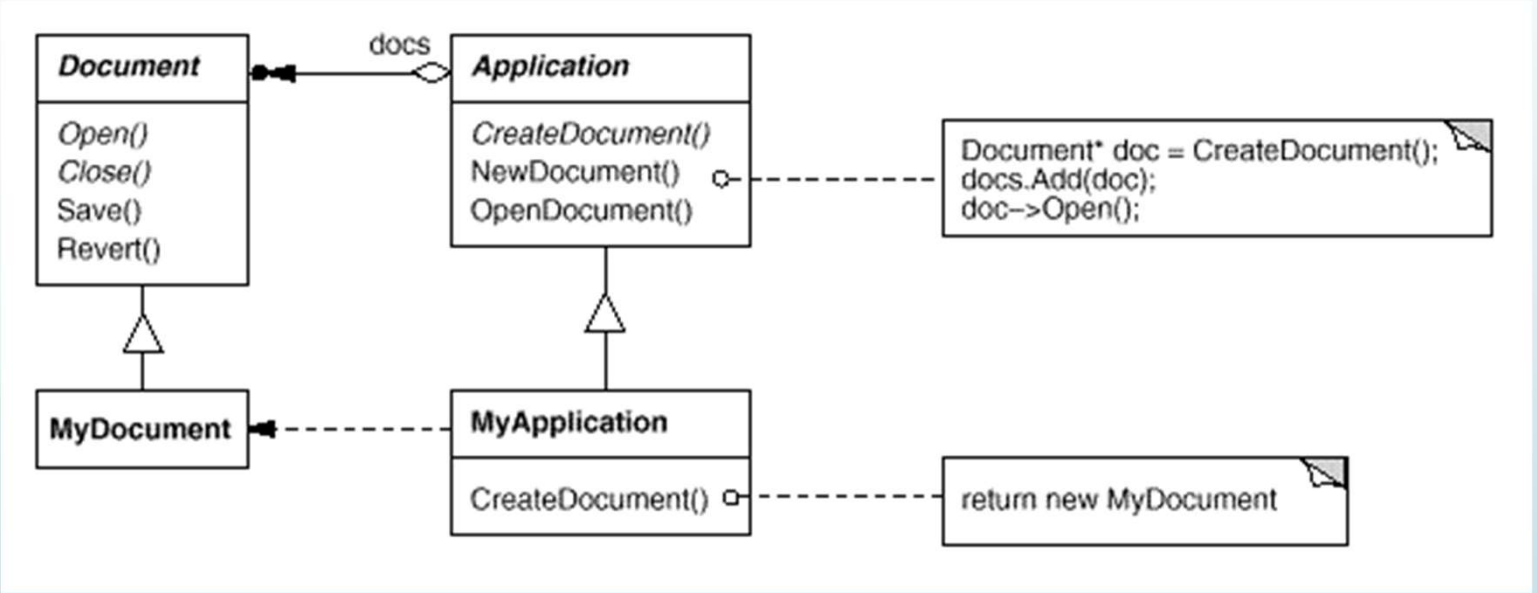
Фабричний метод

- Фабричний метод – це МЕТОД класу, зазвичай віртуальний або абстрактний, який виконує роль «віртуального конструктора» певного продукту.

Мотивація (приклад):

- Майже завжди для створення фреймворків, використовуються абстрактні класи для визначення і підтримки взаємозвязків між об'єктами
- Фреймворк завжди відповідальний за створення цих об'єктів!

Приклад використання



Приклад використання (код фреймворку)

```
class Application {
public:
    virtual Document* CreateDocument() = 0;
    void NewDocument()
    {
        Document* doc = CreateDocument();
        docs_.push_back(doc);
        OpenDocument(doc);
    }
    void OpenDocument(Document* doc) { doc->Open(); }
private:
    std::list<Document*> docs_;
};
```

Ієрархія документів

```
class Document
{
public:
virtual void Open()   = 0;
virtual void Close()  = 0;
virtual void Save()   = 0;
virtual void Revert() = 0;
};
```

```
class MyDocument : public Document
{
public:
void Open()   override {/*...*/}
void Close()  override {/*...*/};
void Save()   override {/*...*/};
void Revert() override {/*...*/};
};
```

Приклад використання (що має зробити користувач)

```
class MyApplication : public Application
{
public:
    Document* CreateDocument() override
    {
        return new MyDocument;
    };
};
```

Чого ми досягли?

- Поліморфізм міг би взяти на себе відповідальність за створення об’єктів.
- Повторне використання вихідного коду
- Гнучкість та розширюваність архітектури
- Відділення логіки створення об’єктів від логіки їх використання
- Менша залежність між типами-користувачами та типами-продуктами або повна ізоляція останніх
- Не передбачено проблему: що робити у випадку, коли користувач захоче, щоб один застосунок міг працювати з РІЗНИМИ типами документів?

(буде вирішено у Шляхах реалізації)

Шляхи реалізації

- Фабричний метод за замовчуванням
- Ліниве створення об’єктів
- Узагальнене програмування для уникнення наслідування
- Параметризовані фабричні методи

Фабричний метод за замовчуванням

- Яку реалізацію обрати?

```
class Creator {
public:
    virtual Product* CreateProduct() = 0;
};
```

```
class Creator {
public:
    virtual Product* CreateProduct() {/*DEFAULT ACTIONS*/ }
};
```

Порівняння реалізацій

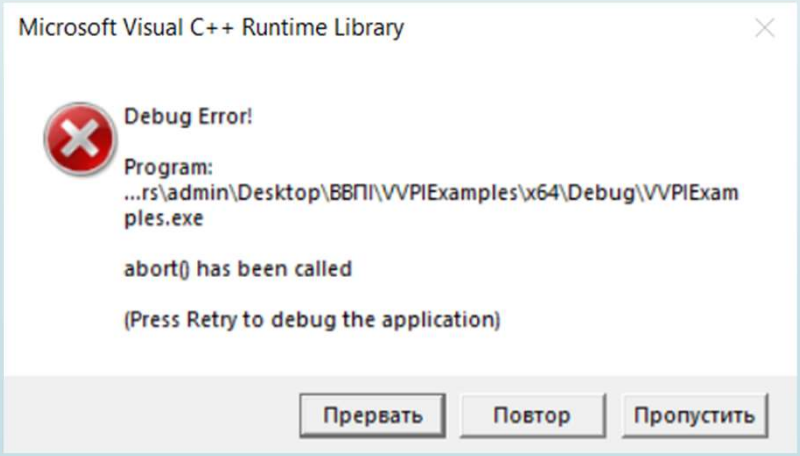
- Обидві реалізації широко використовуються.
- Головна різниця у тому, що варіант з фабричним методом за замовчуванням НЕ ВИМАГАЄ від користувача наслідування від цього класу.
- Через це, об’єкти класу із замовчуванням можна безпосередньо використовувати, погодившись на замовчування.
- Віртуальність фабричного методу із замовчуванням додає гнучкості для користувачів: вони можуть перевизначити цей метод.

Правило (не)використання

- Якщо тип об’єкту, що треба створити, не передбачуваний ні в якому випадку, не слід використовувати фабричний метод за замовчуванням.

Проблема, специфічна для мови C++

- Програмуючи на C++, наражаємося на небезпеку «**Pure Virtual Function Call**»
- Фабричний метод – *взагалі кажучи* абстрактна функція. Може існувати потреба у її виклику в конструкторі базового класу. Це призводить до катастрофи.



Ліниве створення об'єктів

- Попередню проблему можна вирішити за допомогою збереження поля-указника на створюваний об'єкт та лінивого селектора для нього.

```
class Creator {
public:
    Product* GetProduct();
protected:
    virtual Product* CreateProduct();
private:
    Product* _product;
};
```

```
Product* Creator::GetProduct() {
    if (_product == nullptr) {
        _product = CreateProduct();
    }
    return _product;
}
```

Узагальнене програмування для уникнення наслідування

- Припустимо, що у фреймворці міститиметься наступний код:

```
class Creator { //like Application
public:
    virtual Product* CreateProduct() = 0;
};
```

```
template<typename TheProduct>
class StandardCreator : public Creator {
public:
    Product* CreateProduct() override
    {
        return new TheProduct;
    }
};
```

Клієнтський код: «всього лише…»

- Клієнту досить просто визначити свій продукт та спеціалізувати ним шаблонний клас Creator (аналогія з Application), наявний у фреймворці

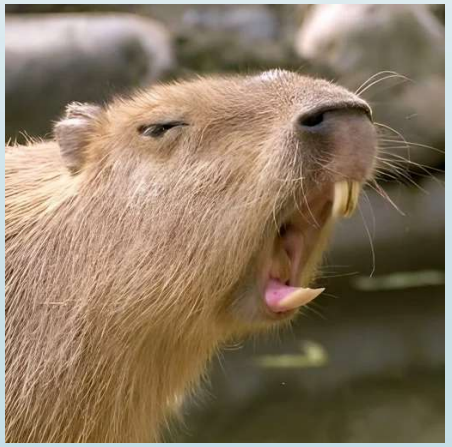
```
class MyProduct : public Product {
public:
    MyProduct();
    // ...
};

using MyProductStandardCreator
          = StandardCreator<MyProduct>;
```

Чи знайшли ми рішення до проблем?

- Менша залежність між типами-користувачами та типами-продуктами або повна ізоляція останніх
- Не передбачено проблему: що робити у випадку, коли користувач захоче, щоб один застосунок міг працювати з РІЗНИМИ типами документів?

Аж ніяк!



Параметризовані фабричні методи

- Додамо до фабричного методу параметр-ідентифікатор, яким будемо вказувати ззовні, продукт якого типу ми хочемо створити.

```
enum class ID
{
    MINE,
    YOURS,
    THEIRS
};
```

```
class Creator {
public:
    virtual Product* Create(ID id) {
        if (id == ID::MINE)
            return new MyProduct;
        if (id == ID::YOURS)
            return new YourProduct;
        // repeat for remaining products...
        return nullptr;
    }
};
```

Конкретний Creator

- Реалізація створеного інтерфейсу дозволяє розширити наявну віртуальну функцію Create:

```
class MyCreator : public Creator {
public:
    Product* Create(ID id) override {
        if (id == ID::YOURS)  return new MyProduct;
        if (id == ID::MINE)   return new YourProduct;
        // N.B.: switched YOURS and MINE
        if (id == ID::THEIRS) return new TheirProduct;
        return Creator::Create(id); // called if all others fail
    }
};
```

Здається, вдалося!

- ~~Не передбачено проблему: що робити у випадку, коли користувач захоче, щоб один застосунок міг працювати з РІЗНИМИ типами документів?~~
- Тепер ми можемо працювати з різними типами продуктів одночасно.
- До того ж, зараз ми визначаємо, який продукт буде створено, НА ЕТАПІ ВИКОНАННЯ, а не на етапі компіляції!

```
auto main() -> int
{
    Creator* creator = new MyCreator;
    int input; cin >> input;
    ID input_id = static_cast<ID>(input);
    Product* product = creator->Create(input_id);
    return 0;
}
```

Як бути із залежністю?

- Менша залежність між типами-користувачами та типами-продуктами або повна ізоляція останніх

Ось хто нам підкаже:

Трюк від Александреску!

- Нехай тепер продукти самі реєструються на фабриці!

```
class Creator
{
public:
    using CreateF = Product* (*)();
private:
    using CallbackMap = std::map<ID, CreateF>;
public:
    bool Register(ID id, CreateF CreateFn);
    bool Unregister(ID id);
    Product* Create(ID id);
private:
    CallbackMap callbacks_;
};
```

```
Product* Creator::Create(ID id)
{
    auto i = callbacks_.find(id);
    if (i != callbacks_.end())
    {
        return (i->second)();
    }
    //error handling...
}
```

Реєстрація продуктів

- У кожному файлі конкретного продукту:

```
namespace
{
    Product* CreateMyProduct()
    {
        return new MyProduct;
    }

    const bool registered =
        Creator::Instance().Register(ID::MINE, CreateMyProduct);
}
```

Не будемо вбивати зайців… – краще попестимо капібар!

Отож Александреску ~~убив одразу двох зайців~~ попестив одразу двох капібар!...

- ~~Менша залежність між типами-користувачами та типами-продуктами або повна ізоляція останніх~~
- ~~Не передбачено проблему: що робити у випадку, коли користувач захоче, щоб один застосунок міг працювати з РІЗНИМИ типами документів?~~

Однак знов кусючі недоліки

- Накладено обмеження на Creator: вона має бути Синглтоном.
- Щоб створювати продукти іншої ієрархії, потрібно продублювати багато коду, скопіювавши щойно написаний клас Creator.
- Указники на функції-створювачі об’єктів не дають можливості використовувати функтори, зокрема лямбда-функції.

Як вирішити проблему з Синглтоном?

Просто!

Реєструємо продукти у фабриках у точці запуску нашої програми!

“main.cpp”:

```
Creator creator;
MyProduct::register_for_serialization(creator);
YourProduct::register_for_serialization(creator);
//…
```

Далі – Dependency Injection (якщо Creator - фабрика)!

“user.cpp”: `User::User(Creator& f) : creator_(f) {}`

“main.cpp”: `User user(creator);`

Generic programming – the great fear of constraints!

```
template<class AbstractProduct, class IdentifierType, class ProductCreator>
class Creator
{
public:
typedef std::map<IdentifierType, AbstractProduct> AssocMap;
bool Register(const IdentifierType& id, ProductCreator creator) {
      return associations_.insert(AssocMap::value_type(id, creator)).second;
}
bool Unregister(const IdentifierType& id){
      return associations_.erase(id) == 1;
}
AbstractProduct* CreateObject(const IdentifierType& id) {
      auto i = associations_.find(id);
      if (i != associations_.end()) return (i->second)();
      //error handling...
}
private:
      AssocMap associations_;
};
```

Що використовувати ідентифікатором?

- Цілочисельні типи
- Рядки
- std::type_info (повернутий тип оператора `typeid`)?...

Але якщо ми використовуємо цей прийом для реалізації серіалізації…

- std::type_info – у жодному разі!
- Він не є стабільним відносно перезапуску програми та його робота залежить від компілятора.

Тож підсумуємо

Фабричний метод дає нам:

- Поліморфізм бере на себе відповідальність за створення об’єктів.
- Повторне використання вихідного коду
- Гнучкість та розширюваність архітектури
- Менша залежність між типами-користувачами та типами-продуктами або повна ізоляція останніх
- Відділення логіки створення об’єктів від логіки їх використання
- Іноді тип продукту, що створюється, обирається легко на етапі виконання.

Підкріпимось перед складним…