Creational Design Pattern

Hung Tran

Fpt software

August 12, 2021

Outline

Creational Pattern Overview

Construction process of an object.

- **Singleton**: Ensure only one instance.
- **Factory Method**: Create instance without depending on its concrete type.
- **Object pool**: Reuse existing instances.
- **Abstract factory**: Create instances from a specific family.
- **Prototype**: Clone existing objects from a prototype.
- **Builder**: Construct a complex object step by step.

"new" operator problem

```
1  #include <iostream>
2
3  using namespace std;
4
5  class Box {
6  private:
7    double length;
8    double breadth;
9    double height;
10 };
11
12 int main(void) {
13   Box *pBox = new Box();
14   delete pBox;
15   return 0;
16 }
```

- Need name of class
- Tightly coupled with the name
- Add new class, modify the existing code
- Compiler does not know which instance created at compile time or an instance has to be created at runtime?

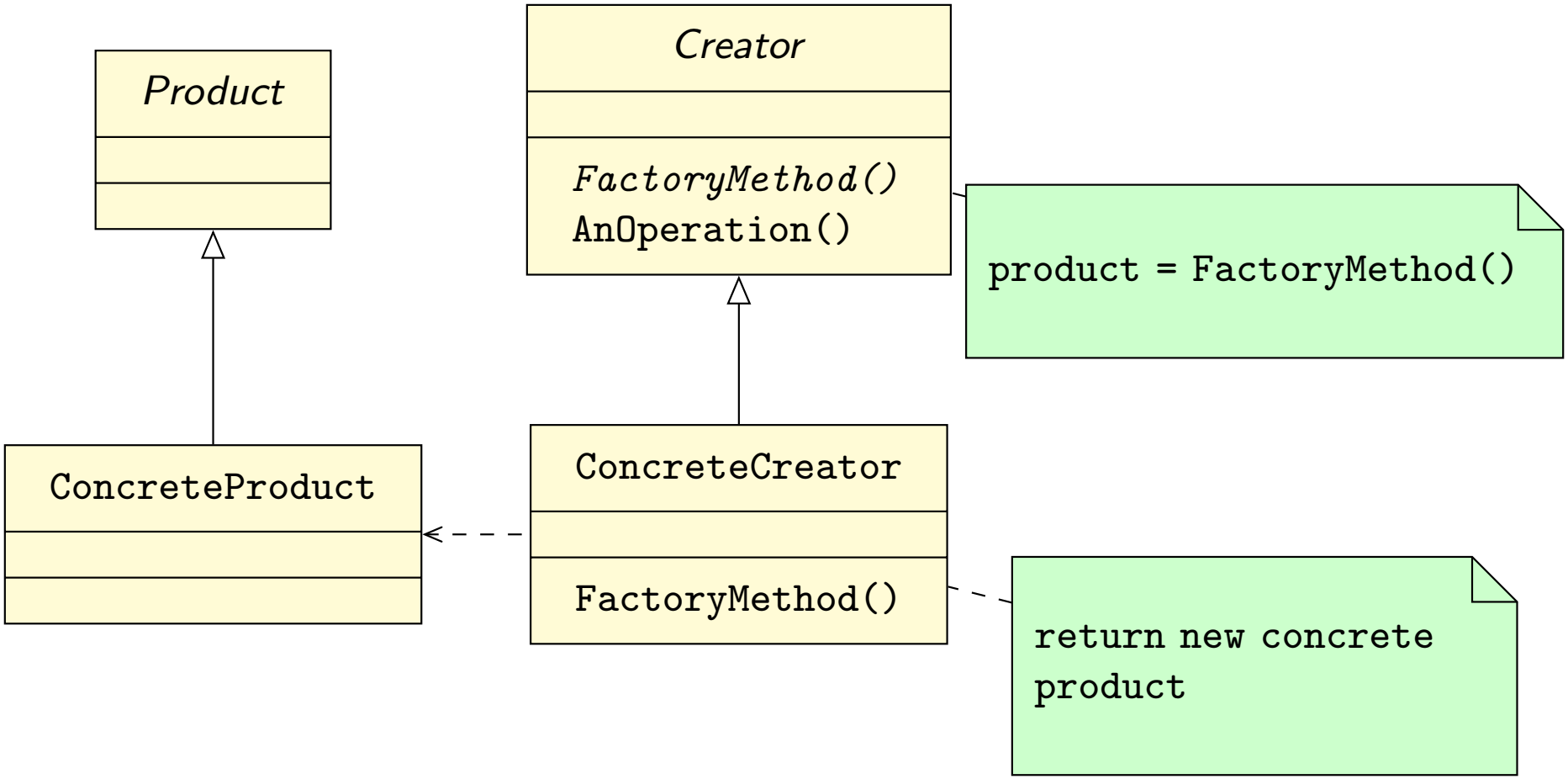
The Intent of Factory Method Design Pattern

Define an interface for creating an object, but let subclasses which class to instantitate. Factory method lets class defer instantiation to subclasses.

How to Implement of Factory Method Design Pattern?

- Different ways to implement
- An overridable method is provide that returns an instance of a class
- This method can be overridden to return instance of a subclass
- Behave likes **constructor**
- However, the constructor always returns the same instance
- The factory method can returns any sub-type
- The factory method also called **virtual constructor**
- C++ language does not allow virtual constructor

Structure of Factory Method Design Pattern



Modify existing code problem

Product.h

```
#ifndef PRODUCT_H
#define PRODUCT_H
class Product{
public:
  virtual void Operation() = 0;
  virtual ~Product() = default;
};
#endif
```

ConcreteProduct.h

```
#ifndef CONCRETE_PRODUCT_H
#define CONCRETE_PRODUCT_H
#include "Product.h"
class ConcreteProduct : public Product{
public:
  void Operation() override;
};
#endif
```

ConcreteProduct.cpp

```
#include "ConcreteProduct.h"
#include <iostream>
void ConcreteProduct::Operation() {
  std::cout << "ConcreteProduct::
      Operation()" << std::endl;
```

Creator.h

```
#ifndef CREATOR_H
#define CREATOR_H
class Product;
class Creator{
  Product *m_pProduct;
public:
  void AnOperation();
};
#endif
```

Creator.cpp

```
#include "Creator.h"
#include "Product.h"
#include "ConcreteProduct.h"
void Creator::AnOperation() {
  m_pProduct = new ConcreteProduct{};
  m_pProduct->Operation();
}
```

main.cpp

```
#include "Creator.h"
int main() {
  Creator ct;
  ct.AnOperation();
  return 0;
```

What if we add one more ConcreteProduct class?

ConcreteProduct1.h

```
1  #ifndef CONCRETE_PRODUCT_H
2  #define CONCRETE_PRODUCT_H
3  #include "Product.h"
4  class ConcreteProduct1 : public Product{
5  public:
6    void Operation() override;
7  };
8  #endif
```

ConcreteProduct1.cpp

```
1  #include "ConcreteProduct.h"
2  #include <iostream>
3  void ConcreteProduct1::Operation() {
4    std::cout << "ConcreteProduct1::
       Operation()" << std::endl;
5  }
```

Creator.cpp

```
1  #include "Creator.h"
2  #include "Product.h"
3  #include "ConcreteProduct.h"
4  void Creator::AnOperation() {
5    m_pProduct = new ConcreteProduct{};
6    m_pProduct->Operation();
7  }
```

Factory Method Design Pattern comes in handy

Basic Implementation

Product.h

```
#ifndef PRODUCT_H
#define PRODUCT_H
class Product {
public:
  virtual void Operation() = 0;
  virtual ~Product() = default;
};
#endif
```

ConcreteProduct.h

```
#ifndef CONCRETE_PRODUCT_H
#define CONCRETE_PRODUCT_H
#include "Product.h"
class ConcreteProduct : public Product {
public:
  void Operation() override;
};
#endif
```

ConcreteProduct.cpp

```
#include "ConcreteProduct.h"
#include <iostream>
void ConcreteProduct::Operation() {
  std::cout << "ConcreteProduct::
      Operation()" << std::endl;
```

ConcreteProduct1.h

```
#ifndef CONCRETE_PRODUCT1_H
#define CONCRETE_PRODUCT1_H
#include "Product.h"
class ConcreteProduct1 : public Product
    {
public:
  void Operation() override;
};
#endif
```

ConcreteProduct1.cpp

```
#include "ConcreteProduct1.h"
#include <iostream>
void ConcreteProduct1::Operation() {
  std::cout << "ConcreteProduct1::
      Operation()" << std::endl;
}
```

Basic Implementation

Creator.h

```
#ifndef CREATOR_H
#define CREATOR_H
class Product;
class Creator {
  Product *m_pProduct;
public:
  void AnOperation();
  virtual Product * Create() {return
      nullptr;};
};
#endif
```

Creator.cpp

```
#include "Creator.h"
#include "Product.h"
void Creator::AnOperation() {
  m_pProduct = Create();
  m_pProduct->Operation();
}
```

ConcreteCreator.h

```
#ifndef CONCRETE_CREATOR_H
#define CONCRETE_CREATOR_H
#include "Creator.h"
class ConcreteCreator : public Creator {
public:
  Product* Create() override;
};
#endif
```

ConcreteCreator.cpp

```
#include "ConcreteCreator.h"
#include "ConcreteProduct.h"
Product* ConcreteCreator::Create() {
  return new ConcreteProduct{};
}
```

Basic Implementation of Factory Method Pattern

ConcreteCreator1.h

```
1  #ifndef CONCRETE_CREATOR1_H
2  #define CONCRETE_CREATOR1_H
3  #include "Creator.h"
4  class ConcreteCreator1 : public Creator
       {
5  public:
6    Product* Create() override;
7  };
8  #endif
```

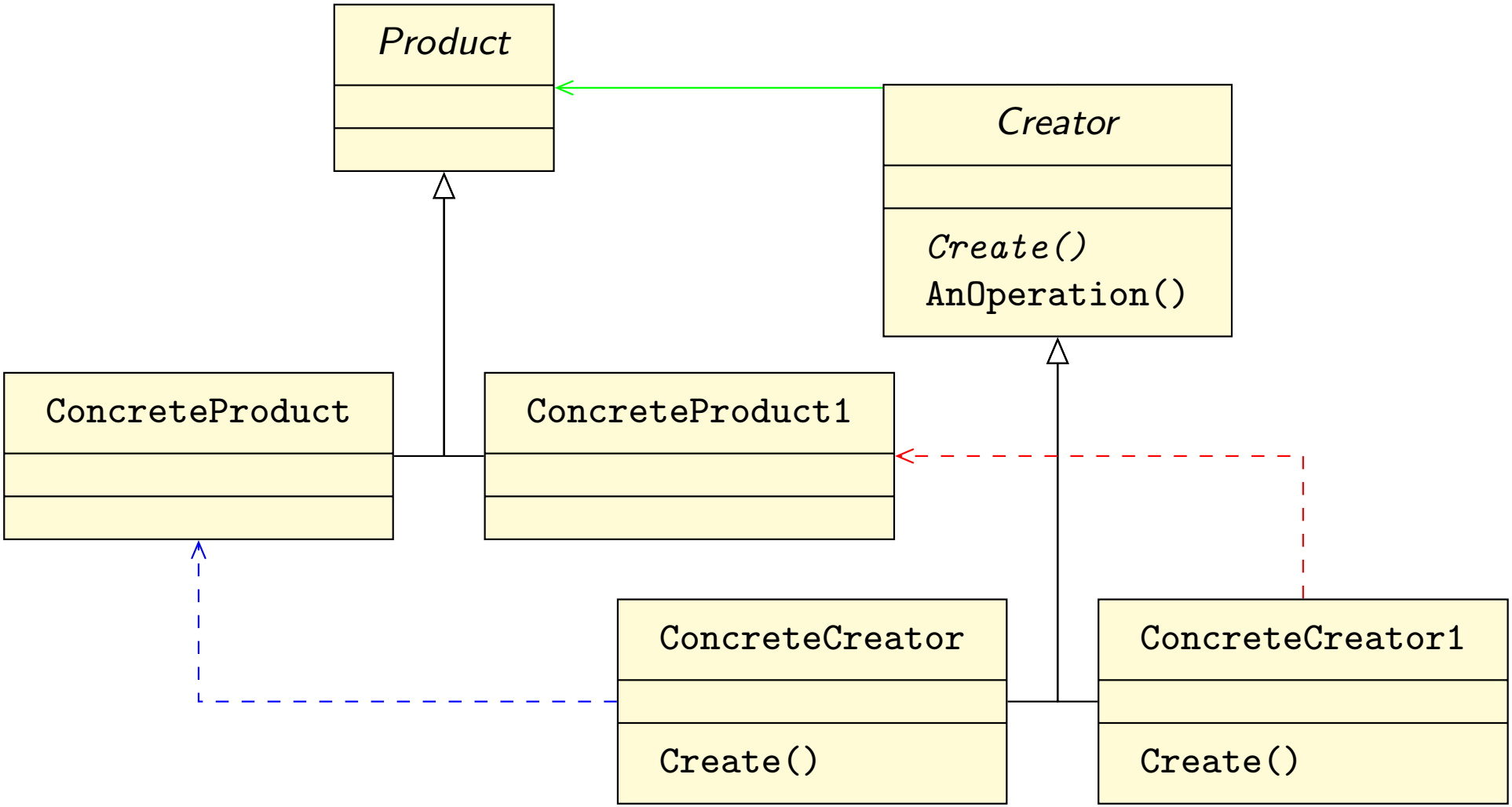
ConcreteCreator1.cpp

```
1  #include "ConcreteCreator1.h"
2  #include "ConcreteProduct1.h"
3  Product* ConcreteCreator1::Create() {
4    return new ConcreteProduct1{};
5  }
```

main.cpp

```
1  #include "Creator.h"
2  #include "ConcreteCreator.h"
3  #include "ConcreteCreator1.h"
4  int main() {
5    ConcreteCreator1 ct;
6    ct.AnOperation();
7  }
```

Class Diagram Explaining

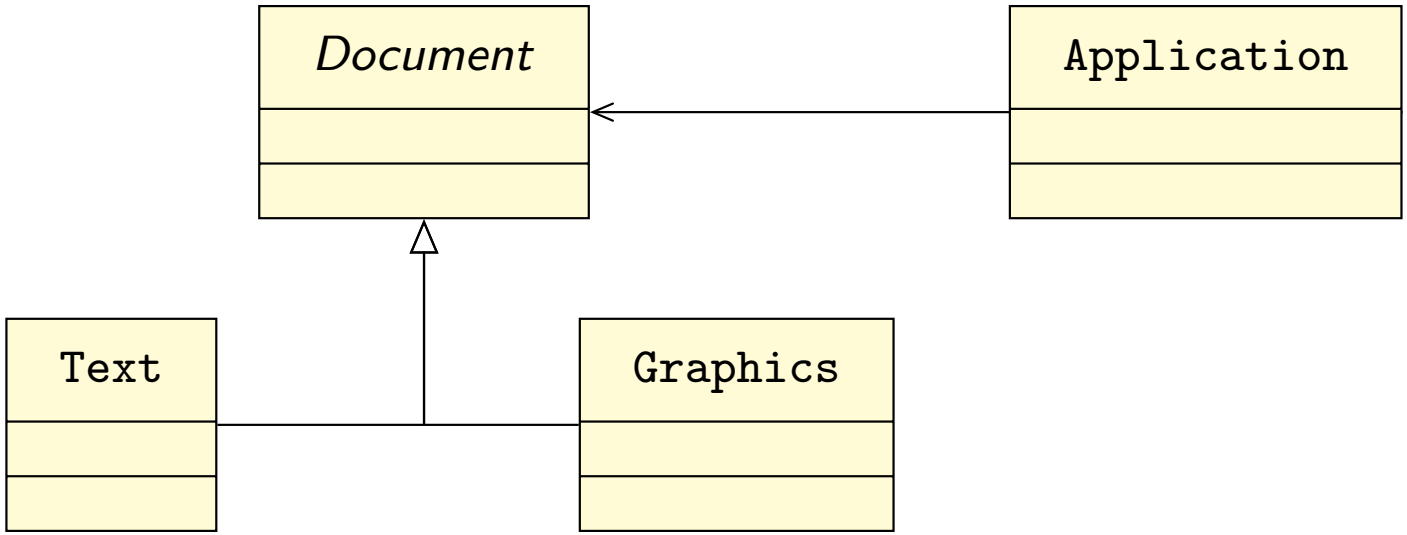


Real World Example: Application Framework?

We want to create an framework

- Managing different kinds of document.
-
- multiple instances are not required.

Real World Example: Application Framework



App framework

Document.h

```
#ifndef DOCUMENT_H
#define DOCUMENT_H
class Document {
public:
  virtual void Write() = 0;
  virtual void Read() = 0;
  virtual ~Document() = default;
};
#endif
```

TextDocument.h

```
#ifndef TEXT_DOCUMENT_H
#define TEXT_DOCUMENT_H
#include "Document.h"
class TextDocument : public Document {
public:
  void Write() override;
  void Read() override;
};
#endif
```

TextDocument.cpp

```
#include "TextDocument.h"
#include <iostream>
void TextDocument::Write() {
  std::cout << "TextDocument::Write()"
     << std::endl;
}
void TextDocument::Read() {
  std::cout << "TextDocument::Read()" <<
       std::endl;
}
```

App framework

Application.h

```
#ifndef APPLICATION_H
#define APPLICATION_H
class Document;
class Application
{
  Document *m_pDocument;
public:
  void New();
  void Open();
  void Save();
};
#endif
```

main.cpp

```
#include "Application.h"

int main() {
  Application app;
  app.New();
  app.Open();
  app.Save();
  return 0;
}
```

Application.cpp

```
#include "Application.h"
#include "TextDocument.h"
void Application::New() {
  m_pDocument = new TextDocument{};
}
void Application::Open() {
  m_pDocument = new TextDocument{};
  m_pDocument->Read();
}
void Application::Save() {
  m_pDocument->Write();
}
```

The above implementation Problem

If we want to manage with different doc?

- Managing different kinds of document.
-
- multiple instances are not required.

Pros and Cons

Pros

- Class itself control the instantiation process.
- Can allow multiple instances.
- Better than global variable.
- Can be subclassed.

Cons

- Testing is difficult
- DCLP is defective
- Lazy destruction is complex

Where to use?

When only one instance should be use because:

- multiple instances cause data corruption.
- managing global state or shared state.
- multiple instances are not required.