Creational Design Pattern

Hung Tran

Fpt software

August 19, 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?

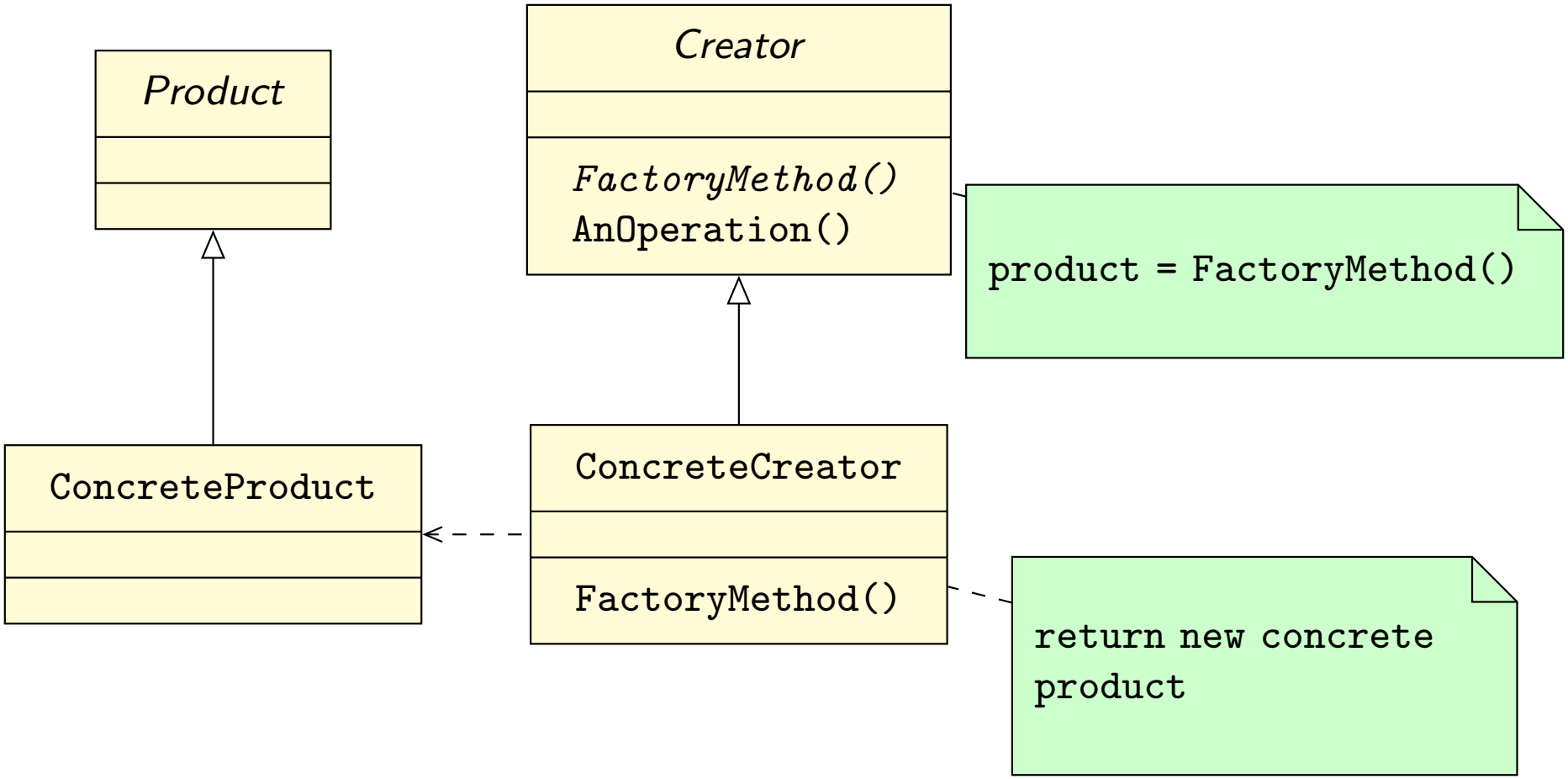
The Intent of Factory Method Design Pattern

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
```

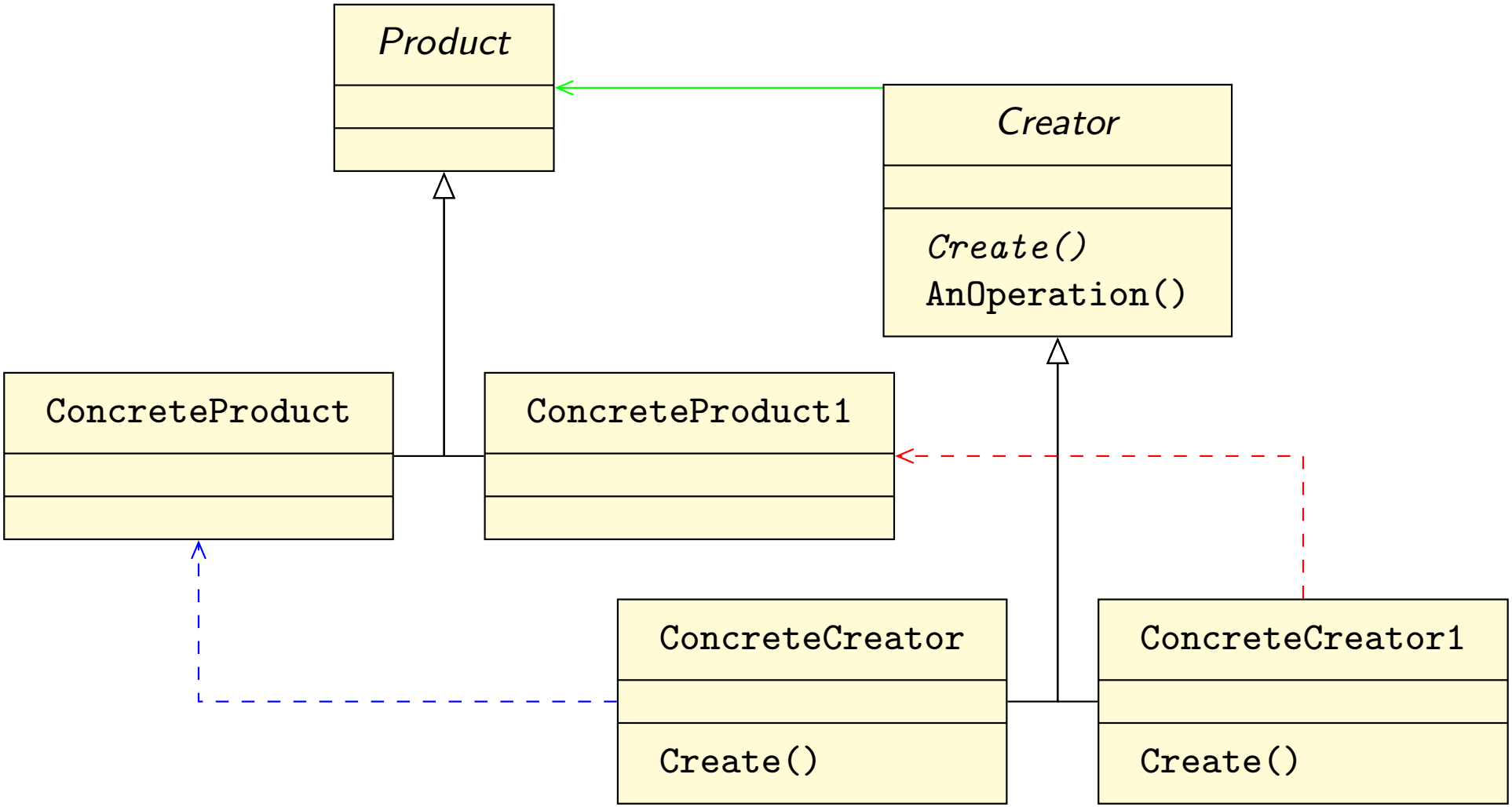
ConcreteCreator1.cpp

```
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

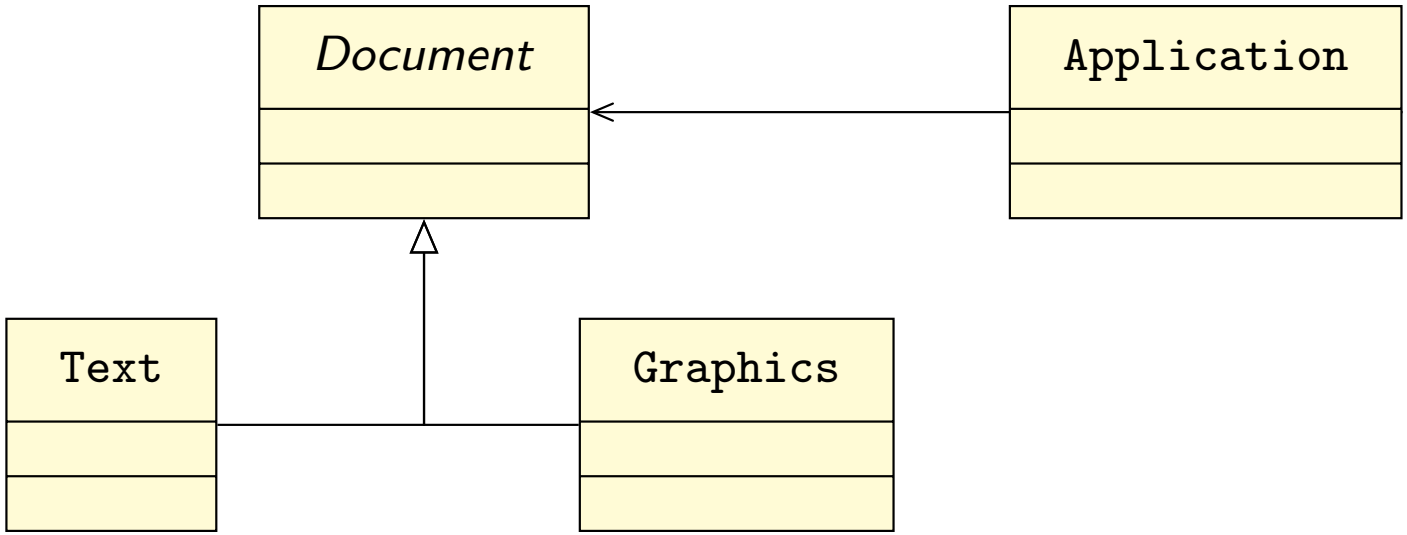


Real World Example: Application Framework?

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

```
1  #ifndef DOCUMENT_H
2  #define DOCUMENT_H
3  class Document {
4  public:
5    virtual void Write() = 0;
6    virtual void Read() = 0;
7    virtual ~Document() = default;
8  };
9  #endif
```

TextDocument.h

```
1  #ifndef TEXT_DOCUMENT_H
2  #define TEXT_DOCUMENT_H
3  #include "Document.h"
4  class TextDocument : public Document {
5  public:
6    void Write() override;
7    void Read() override;
8  };
9  #endif
```

TextDocument.cpp

```
1  #include "TextDocument.h"
2  #include <iostream>
3  void TextDocument::Write() {
4    std::cout << "TextDocument::Write()"
       << std::endl;
5  }
6  void TextDocument::Read() {
7    std::cout << "TextDocument::Read()" <<
       std::endl;
8  }
```

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 docs?

- Make change to Application class.
- But it is Framework (not support for modification)
- Application class is tightly coupled with TextDocument class
- Remove the dependency on TextDocument class
- Application should be worked with any kind of data.

Implement Factory Method

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

SpreadSheetDocument.h

```
1  #ifndef SPREAD_SHEET_DOCUMENT_H
2  #define SPREAD_SHEET_DOCUMENT_H
3  #include "Document.h"
4  class SpreadSheetDocument : public
       Document {
5  public:
6    void Write() override;
7    void Read() override;
8  };
9  #endif
```

SpreadSheetDocument.cpp

```
1  #include "SpreadSheetDocument.h"
2  #include <iostream>
3  void SpreadSheetDocument::Write() {
4    std::cout << "SpreadSheetDocument::
       Write()" << std::endl;
5  }
6  void SpreadSheetDocument::Read() {
7    std::cout << "SPreadSheetDocument::
       Read()" << std::endl;
8  }
```

SpreadSheetApplication.h

```
1  #ifndef SPREAD_SHEET_APPPLICATION_H
2  #define SPREAD_SHEET_APPLICATION_H
3  #include "Application.h"
4  class SpreadSheetApplication : public
       Application {
5  public:
6    Document* Create() override;
7  };
8  #endif
```

SpreadSheetApplication.cpp

```
1  #include "SpreadSheetApplication.h"
2  #include "SpreadSheetDocument.h"
3
4  Document* SpreadSheetApplication::Create
       () {
5    return new SpreadSheetDocument{};
6  }
```

App framework

Application.h

```
1  #ifndef APPLICATION_H
2  #define APPLICATION_H
3  class Document;
4  class Application {
5    Document* m_pDocument;
6  public:
7    void New();
8    void Open();
9    void Save();
10   virtual Document* Create(){return
         nullptr;}
11 };
12 #endif
```

TextApplication.h

```
1  #ifndef TEXT_APPLICATION_H
2  #define TEXT_APPLICATION_H
3  #include "Application.h"
4  class TextApplication : public
       Application {
5  public:
6    Document* Create() override;
7  };
8  #endif
```

Application.cpp

```
1  #include "Application.h"
2  #include "Document.h"
3
4  void Application::New() {
5    m_pDocument = Create();
6  }
7
8  void Application::Open() {
9    m_pDocument = Create();
10   m_pDocument->Read();
11 }
12
13 void Application::Save() {
14   m_pDocument->Write();
15 }
```

TextApplication.cpp

```
1  #include "TextApplication.h"
2  #include "TextDocument.h"
3  Document* TextApplication::Create() {
4    return new TextDocument{};
5  }
```

Memory management problem?

If we want to manage with different docs?

- Make change to Application class.
- But it is Framework (not support for modification)
- Application class is tightly coupled with TextDocument class
- Remove the dependency on TextDocument class
- Application should be worked with any kind of data.

Implement Factory Method

App framework: using smart pointer

Document.h

```
1  #ifndef DOCUMENT_H
2  #define DOCUMENT_H
3  class Document {
4  public:
5    virtual void Write() = 0 ;
6    virtual void Read() = 0 ;
7    virtual ~Document() = default ;
8  };
9  #endif
```

TextDocument.h

```
1  #ifndef TEXT_DOCUMENT_H
2  #define TEXT_DOCUMENT_H
3  #include "Document.h"
4  class TextDocument : public Document {
5  public:
6    void Write() override;
7    void Read() override;
8  };
9  #endif
```

TextDocument.cpp

```
1  #include "TextDocument.h"
2  #include <iostream>
3  void TextDocument::Write() {
4    std::cout << "TextDocument::Write()"
       << std::endl;
5  }
6  void TextDocument::Read() {
7    std::cout << "TextDocument::Read()" <<
       std::endl;
8  }
```

App framework: using smart pointer

SpreadSheetDocument.h

```
1  #ifndef SPREAD_SHEET_DOCUMENT_H
2  #define SPREAD_SHEET_DOCUMENT_H
3  #include "Document.h"
4  class SpreadSheetDocument : public
       Document {
5  public:
6    void Write() override;
7    void Read() override;
8  };
9  #endif
```

SpreadSheetDocument.cpp

```
1  #include "SpreadSheetDocument.h"
2  #include <iostream>
3  void SpreadSheetDocument::Write() {
4    std::cout << "SpreadSheetDocument::
       Write()" << std::endl;
5  }
6  void SpreadSheetDocument::Read() {
7    std::cout << "SPreadSheetDocument::
       Read()" << std::endl;
8  }
```

SpreadSheetApplication.h

```
1  #ifndef SPREAD_SHEET_APPPLICATION_H
2  #define SPREAD_SHEET_APPLICATION_H
3  #include "Application.h"
4  class SpreadSheetApplication : public
       Application {
5  public:
6    DocumentPtr Create() override;
7  };
8  #endif
```

SpreadSheetApplication.cpp

```
1  #include "SpreadSheetApplication.h"
2  #include "SpreadSheetDocument.h"
3
4  DocumentPtr SpreadSheetApplication::
       Create() {
5    return std::make_unique<
       SpreadSheetDocument>();
6  }
```

App framework: using smart pointer

Application.h

```
1  #ifndef APPLICATION_H
2  #define APPLICATION_H
3  #include <memory>
4  #include "Document.h"
5  //class Document;
6  using DocumentPtr = std::unique_ptr<
       Document>;
7  class Application {
8    DocumentPtr m_pDocument;
9  public:
10   void New();
11   void Open();
12   void Save();
13   virtual DocumentPtr Create(){return
       nullptr;}
14 };
15 #endif
```

TextApplication.h

```
1  #ifndef TEXT_APPLICATION_H
2  #define TEXT_APPLICATION_H
3  #include "Application.h"
4  class TextApplication : public
       Application {
5  public:
6    DocumentPtr Create() override;
```

Application.cpp

```
1  #include "Application.h"
2  #include "Document.h"
3
4  void Application::New() {
5    m_pDocument = Create();
6  }
7
8  void Application::Open() {
9    m_pDocument = Create();
10   m_pDocument->Read();
11 }
12
13 void Application::Save() {
14   m_pDocument->Write();
15 }
```

TextApplication.cpp

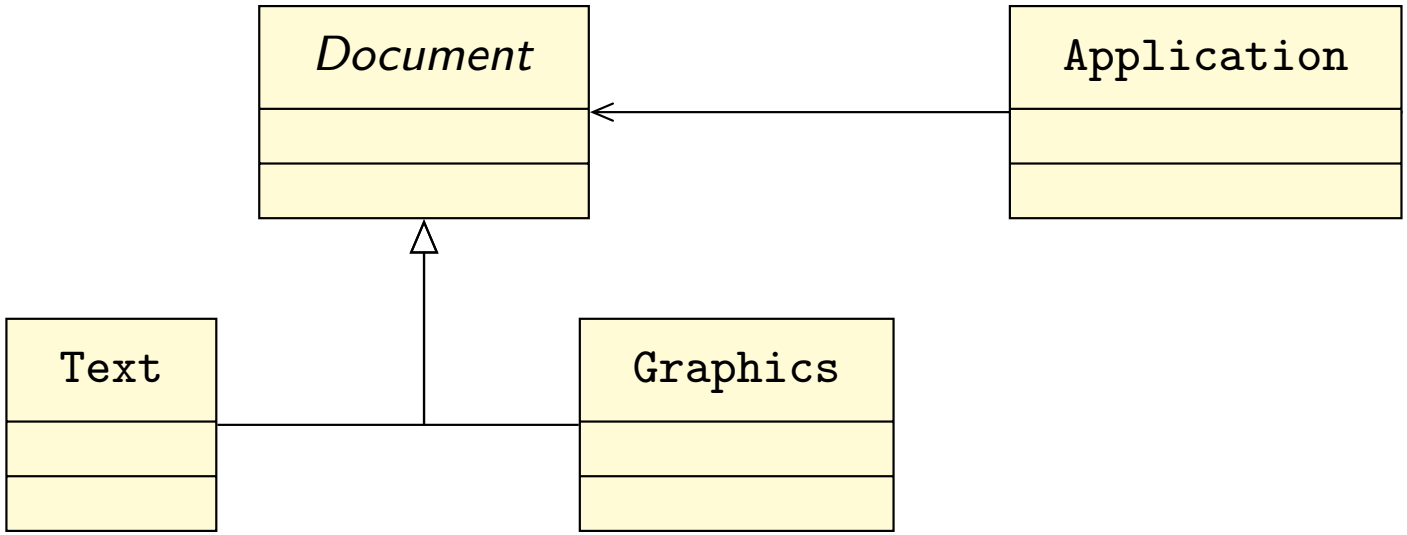
```
1  #include "TextApplication.h"
2  #include "TextDocument.h"
3  DocumentPtr TextApplication::Create() {
4    return std::make_unique<TextDocument
       >();
5  }
```

Add new product, add more creator?

How to create multiple instances without creating corresponding application class?

Using Parameterized Factory

Classes structure: Parameterized Factory



What if you want to read different kinds of document?

- Specify the type of the string in Application.cpp
- The string can take from user interface
- DocumentFactory class has no state because of no attribute.
- No attribute so not necessary to have multiple instances.
- Make Create() function in DocumentFactory static
- Only one negative point is that if you want to add more documents in the future, you will modify the Create() method.
- It is trivial change because of only add some if condition
- Or you can make Create() virtual, so add more document, just add more document factory.

App framework: parameterized Factory

Document.h

```
#ifndef DOCUMENT_H
#define DOCUMENT_H
#include <memory>

class Document {
public:
  virtual void Write() = 0;
  virtual void Read() = 0;
  virtual ~Document() = default;
};
using DocumentPtr = std::unique_ptr<
    Document>;
#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: parameterized Factory

SpreadSheetDocument.h

```
1  #ifndef SPREAD_SHEET_DOCUMENT_H
2  #define SPREAD_SHEET_DOCUMENT_H
3  #include "Document.h"
4  class SpreadSheetDocument : public
       Document {
5  public:
6    void Write() override;
7    void Read() override;
8  };
9  #endif
```

SpreadSheetDocument.cpp

```
1  #include "SpreadSheetDocument.h"
2  #include <iostream>
3  void SpreadSheetDocument::Write() {
4    std::cout << "SpreadSheetDocument::
       Write()" << std::endl;
5  }
6  void SpreadSheetDocument::Read() {
7    std::cout << "SpreadSheetDocument::
       Read()" << std::endl;
8  }
```

App framework: parameterized Factory

Application.h

```
1  #ifndef APPLICATION_H
2  #define APPLICATION_H
3  #include <memory>
4  #include "Document.h"
5  class Application {
6    DocumentPtr m_pDocument;
7  public:
8    void New();
9    void Open();
10   void Save();
11 };
12 #endif
```

Application.cpp

```
1  #include "Application.h"
2  #include "DocumentFactory.h"
3
4  void Application::New() {
5    m_pDocument = DocumentFactory::Create(
       "text");
6  }
7
8  void Application::Open() {
9    DocumentFactory factory;
10   m_pDocument = DocumentFactory::Create(
       "text");
11   m_pDocument->Read();
12 }
13
14 void Application::Save() {
15   m_pDocument->Write();
16 }
```

Pros and Cons

Pros

- Instances can be created at runtime
- Promote loose coupling
- Construction becomes simple due to abstraction
- Construction becomes encapsulated
- May not return new instance every time (return a cache instance), useful for object pool

Cons

- Every new product class may require a corresponding factory class.

Where to use?

- A class does not know which instance it needs at runtime.
- A class does not want to depend on concrete classes that it uses.
- You want to encapsulate the creation process.