# C++ Programming II

## Design Patterns II
## Factory Method

*C++ Programming II*
*November 26, 2018*

Prof. Dr. P. Arnold
Bern University of Applied Sciences

# Agenda

- **Design Patterns II**



# Agenda

- **Design Patterns II**
- **Factory Method**

# Agenda

- Design Patterns II
- Factory Method
- Code Project - Review

# Design Patterns II



# Design Patterns

## What is a design pattern and why should we use it?

- A design pattern is a general solution to a common software design problem.
- The term was made popular by the book *Design Patterns: Elements of Reusable Object-Oriented Software*, also known as the Gang of Four [1].

[1] Gamma et al., 1994

# Design Patterns

## What is a design pattern and why should we use it?

- A design pattern is a general solution to a common software design problem.
- The term was made popular by the book *Design Patterns: Elements of Reusable Object-Oriented Software*, also known as the Gang of Four [1].

| | |
|---|---|
| **Creational Patterns** | |
| Abstract Factory | Encapsulates a group of related factories. |
| Builder | Separates a complex object's construction from its representation. |
| Factory Method | Lets a class defer instantiation to subclasses. |
| Prototype | Specifies a prototypical instance of a class that can be cloned to produce new objects. |
| Singleton | Ensures a class has only one instance. |
| **Structural Patterns** | |
| Adapter | Converts the interface of one class into another interface. |
| Bridge | Decouples an abstraction from its implementation so that both can be changed independently. |
| Composite | Composes objects into tree structures to represent part–whole hierarchies. |
| Decorator | Adds additional behavior to an existing object in a dynamic fashion. |
| Façade | Provides a unified higher-level interface to a set of interfaces in a subsystem. |
| Flyweight | Uses sharing to support large numbers of fine-grained objects efficiently. |
| Proxy | Provides a surrogate or placeholder for another object to control access to it. |
| **Behavioral Patterns** | |
| Chain of Responsibility | Gives more than one receiver object a chance to handle a request from a sender object. |
| Command | Encapsulates a request or operation as an object, with support for undoable operations. |
| Interpreter | Specifies how to represent and evaluate sentences in a language. |
| Iterator | Provides a way to access the elements of an aggregate object sequentially. |
| Mediator | Defines an object that encapsulates how a set of objects interact. |
| Memento | Captures an object's internal state so that it can be restored to the same state later. |
| Observer | Allows a one-to-many notification of state changes between objects. |
| State | Allows an object to appear to change its type when its internal state changes. |
| Strategy | Defines a family of algorithms, encapsulates each one, and makes them interchangeable at run time. |
| Template Method | Defines the skeleton of an algorithm in an operation, deferring some steps to subclasses. |
| Visitor | Represents an operation to be performed on the elements of an object structure. |

[1] Gamma et al., 1994

# Factory Method

# Factory Method

## Create different classes at runtime

- A *Factory Method* is a creational design pattern.
- It allows to create objects without having to specify the C++ type of the object.

# Factory Method

## Create different classes at runtime

- A *Factory Method* is a creational design pattern.
- It allows to create objects without having to specify the C++ type of the object.
- A *Factory Method* is essentially a generalised form of a constructor, but without the following limitations:
  1. **No return value:** Not possible to return a result from a constructor, *e.g.* to signal an error during initialisation and returning `nullptr`.
  2. **Statically bound creation:** The name of a concrete class of the constructed object must be known at compile time. There is no concept of dynamic binding at run time for constructors in C++

# Factory Method

## Create different classes at runtime

- A *Factory Method* is a creational design pattern.
- It allows to create objects without having to specify the C++ type of the object.
- A *Factory Method* is essentially a generalised form of a constructor, but without the following limitations:
  1. **No return value:** Not possible to return a result from a constructor, *e.g.* to signal an error during initialisation and returning `nullptr`.
  2. **Statically bound creation:** The name of a concrete class of the constructed object must be known at compile time. There is no concept of dynamic binding at run time for constructors in C++
- The *Factory Method* circumvent these limitations.
- In general, the *Factory Method* is simply a normal method call that can return an instance of a class.

# Factory Method

## By Example from "API Design for C++"

- Let's start by an *ABC* for an extremely simple 3D graphics renderer

```
#ifndef RENDERER_H
#define RENDERER_H

#include <string>

///
/// An abstract interface for a 3D renderer.
///
class IRenderer
{
public:
    virtual ~IRenderer() {}
    virtual bool LoadScene(const std::string &filename) = 0;
    virtual void SetViewportSize(int w, int h) = 0;
    virtual void SetCameraPos(double x, double y, double z) = 0;
    virtual void SetLookAt(double x, double y, double z) = 0;
    virtual void Render() = 0;
};

#endif
```

**Note!**

Always declare the destructor of an abstract base class to be virtual!

# Factory Method

## By Example from “API Design for C++ ”

- Now we declare a *Factory Method* which generates *renderers* for us:

```
#ifndef RENDERERFACTORY_H
#define RENDERERFACTORY_H

#include "renderer.h"
#include <string>

///
/// A factory object that creates instances of different
/// 3D renderers.
///
class RendererFactory
{
public:
    /// Create a new instance of a named 3D renderer.
    /// type can be one of "opengl", "directx", or "mesa"
    IRenderer *CreateRenderer(const std::string &type);
};

#endif
```

# Factory Method

## By Example from "API Design for C++"

- Now we declare a *Factory Method* which generates *renderers* for us:

```
#ifndef RENDERERFACTORY_H
#define RENDERERFACTORY_H

#include "renderer.h"
#include <string>

///
/// A factory object that creates instances of different
/// 3D renderers.
///
class RendererFactory
{
public:
    /// Create a new instance of a named 3D renderer.
    /// type can be one of "opengl", "directx", or "mesa"
    IRenderer *CreateRenderer(const std::string &type);
};

#endif
```

- That's it!
- You can use the string argument to `CreateRenderer()` to specify which derived type you want to create.

# Factory Method

## By Example from “API Design for C++ ”

- Let’s assume we have implemented 3 concrete classes `OpenGLRenderer` , `DirectXRenderer` , and `MesaRenderer`
- Further we specify that these renderer types are completely hidden behind the API



# Factory Method

## By Example from “API Design for C++ ”

- Let’s assume we have implemented 3 concrete classes `OpenGLRenderer` , `DirectXRenderer` , and `MesaRenderer`
- Further we specify that these renderer types are completely hidden behind the API
- Based on these conditions the implementation of the factory method will look as follows

```
#include "rendererfactory.h"
#include <iostream>

using std::cout;
using std::endl;

class OpenGLRenderer : public IRenderer
{
public:
    ~OpenGLRenderer() {}
    bool LoadScene(const std::string &filename) { return true; }
    void SetViewportSize(int w, int h) {}
    void SetCameraPos(double x, double y, double z) {}
    void SetLookAt(double x, double y, double z) {}
    void Render() { cout << "OpenGL Render" << endl; }
};
```

# Factory Method

## By Example from “API Design for C++ ”

- Let’s assume we have implemented 3 concrete classes `OpenGLRenderer` , `DirectXRenderer` , and `MesaRenderer`
- Further we specify that these renderer types are completely hidden behind the API
- Based on these conditions the implementation of the factory method will look as follows

```
class DirectXRenderer : public IRenderer
{
public:
    bool LoadScene(const std::string &filename) { return true; }
    void SetViewportSize(int w, int h) {}
    void SetCameraPos(double x, double y, double z) {}
    void SetLookAt(double x, double y, double z) {}
    void Render() { cout << "DirectX Render" << endl; }
};
```

# Factory Method

## By Example from “API Design for C++ ”

- Let’s assume we have implemented 3 concrete classes `OpenGLRenderer` , `DirectXRenderer` , and `MesaRenderer`
- Further we specify that these renderer types are completely hidden behind the API
- Based on these conditions the implementation of the factory method will look as follows

```
class MesaRenderer : public IRenderer
{
public:
    bool LoadScene(const std::string &filename) { return true; }
    void SetViewportSize(int w, int h) {}
    void SetCameraPos(double x, double y, double z) {}
    void SetLookAt(double x, double y, double z) {}
    void Render() { cout << "Mesa Render" << endl; }
};
```

# Factory Method

## By Example from “API Design for C++ ”

- Let’s assume we have implemented 3 concrete classes `OpenGLRenderer` , `DirectXRenderer` , and `MesaRenderer`
- Further we specify that these renderer types are completely hidden behind the API
- Based on these conditions the implementation of the factory method will look as follows

```
IRenderer *RendererFactory::CreateRenderer(const string &type)
{
    if (type == "opengl")
        return new OpenGLRenderer;

    if (type == "directx")
        return new DirectXRenderer;

    if (type == "mesa")
        return new MesaRenderer;

    return NULL;
```

# Factory Method

## By Example from “API Design for C++ ”

- Using the factory look like this:

```
#include "rendererfactory.h"

int main( )
{
    // create the factory object
    RendererFactory *f = new RendererFactory;

    // create an OpenGL renderer
    IRenderer *ogl = f->CreateRenderer("opengl");
    ogl->Render();
    delete ogl;

    // create a Mesa software renderer
    IRenderer *mesa = f->CreateRenderer("mesa");
    mesa->Render();
    delete mesa;

    delete f;
    return 0;
}
```

# Factory Method

## By Example from “API Design for C++ ”

- Using the factory look like this:

```
#include "rendererfactory.h"

int main( )
{
    // create the factory object
    RendererFactory *f = new RendererFactory;

    // create an OpenGL renderer
    IRenderer *ogl = f->CreateRenderer("opengl");
    ogl->Render();
    delete ogl;

    // create a Mesa software renderer
    IRenderer *mesa = f->CreateRenderer("mesa");
    mesa->Render();
    delete mesa;

    delete f;
    return 0;
}
```

- Tip: Use smart pointers instead of raw pointers
- Demo

# Code Project - Review

# Project

## First in Class Code Review: 03.12.2018 - Time: 25' each

Make sure to have some code in place, most important:

- `CMakeLists.txt`, which *e.g.* includes external libraries *etc.*
- *Makro design* of application, no detailed implementation
- For the review create a git project on gitlab and give access to your team partner and me.
- Without git, use .zip files and some version numbers in the file name for the exchange.

# Project

## First in Class Code Review: 03.12.2018 - Time: 25' each

Make sure to have some code in place, most important:

- `CMakeLists.txt`, which *e.g.* includes external libraries *etc.*
- *Makro design* of application, no detailed implementation
- For the review create a git project on gitlab and give access to your team partner and me.
- Without git, use .zip files and some version numbers in the file name for the exchange.
- Write your feedback directly into the code in form of comments using the Todo-feature of Qt-Creator:
  - `// TODO`
  - `// NOTE`
  - `// WARNING`
  - `// FIXME`
  - `// BUG`

# Project

## First in Class Code Review: 03.12.2018 - Time: 25' each

Make sure to have some code in place, most important:

- `CMakeLists.txt`, which *e.g.* includes external libraries *etc.*
- *Makro design* of application, no detailed implementation
- For the review create a git project on gitlab and give access to your team partner and me.
- Without git, use .zip files and some version numbers in the file name for the exchange.
- Write your feedback directly into the code in form of comments using the Todo-feature of Qt-Creator:
  - `// TODO`
  - `// NOTE`
  - `// WARNING`
  - `// FIXME`
  - `// BUG`
- Discuss your ideas and issues with your team partner
- Forward the unsolved issues and question to me.

# Project

## First in Class Code Review: 03.12.2018 - Time: 25' each

Make sure to have some code in place, most important:

- `CMakeLists.txt`, which *e.g.* includes external libraries *etc.*
- *Makro design* of application, no detailed implementation
- For the review create a git project on gitlab and give access to your team partner and me.
- Without git, use .zip files and some version numbers in the file name for the exchange.
- Write your feedback directly into the code in form of comments using the Todo-feature of Qt-Creator:
  - `// TODO`
  - `// NOTE`
  - `// WARNING`
  - `// FIXME`
  - `// BUG`
- Discuss your ideas and issues with your team partner
- Forward the unsolved issues and question to me.

**Note!**

The more you have, the better the review!

# Thank You

## Questions

???