C++ Programming II

Design Patterns
Observer & Factory Pattern

C++ Programming II
November 19, 2018

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

Agenda

- Design Patterns
- Observer Pattern
- Exercise 06
- Code Project

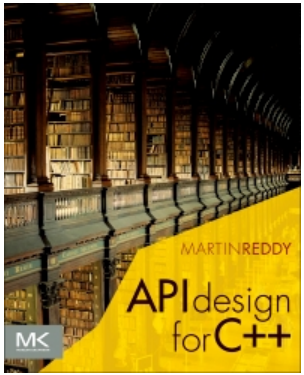
Bern University of Applied Sciences

Design Patterns

Design Patterns

Recommended literature!

- **API Design for C++** (First Edition, 2011), Martin Reddy, ISBN-13: 978-0-12-385003-4



Source code and a lot of examples:
`http://www.apibook.com/blog/source-code`

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

Observer Pattern

Observer Pattern

Simplest approach leads to “tight coupling” of modules

- It’s very common for objects to call methods of other objects.
- Therefore,*e.g.* an object camera.cpp must know about a control’s interface to call it’s methods.
- The simplest approach is for `camera.cpp` to include `control.h` and call the methods directly.

```
#ifndef CAMERA_H
#define CAMERA_H
#include "control.h"

class Camera
{
public:
    Camera(Control* control);
private:
    Control* m_control;
};

#endif // CAMERA_H
```

- However, this introduces a compile time dependency between `Camera` and `Control` and forces the classes to be tightly coupled!

Observer Pattern

Introducing an abstract base class (ABC)

- A more flexible approach is to provide an ABC `ICamera` for `Camera`

```
#ifndef ICAMERA_H
#define ICAMERA_H

class ICamera
{
public:
    ICamera(){}
    virtual ~ICamera(){}

    virtual void newImage() = 0;
};

#endif // ICAMERA_H
```

Observer Pattern

No compile time dependency - “Loose Coupling” of modules

- Pass the interface to the `Camera` at initialisation (subscription) ...

```
#ifndef CAMERA_H
#define CAMERA_H
#include "icamera.h"

class Camera
{
public:
    Camera(ICamera* host);
};

#endif // CAMERA_H
```

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


Camera::Camera(ICamera *host)
{
    std::cout << "Camera acquires new image ..." <<
        std::endl;
    host->newImage();
}
```

- Note: No compile time dependency! `Camera` can be compiled and tested without `Control`

Observer Pattern

Control inherits and implements camera interface

```
#include "camera.h"

class Control : public ICamera
{
public:
    Control();

    // from ICamera
    virtual void newImage() override;
};
```

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

Control::Control()
{
    Camera cam(this); // passing host interface to cam
}

void Control::newImage()
{
    std::cout << "Control got new image ..." <<
        std::endl;

}
```

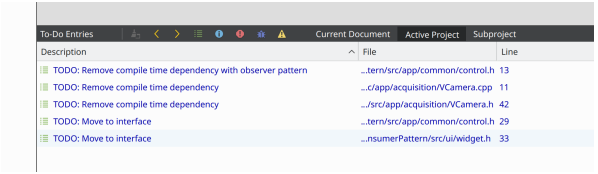
- `newImage()` is called by camera!

Exercise 06

Observer Pattern

Exercise 06

- Register at `https://gitlab.ti.bfh.ch`
- Clone the *Producer Consumer Pattern* project: `https://gitlab.ti.bfh.ch/BMECPPII_2018/designpattern`
- or download .zip file
- Solve the exercises described in README.md
- Activate the Todo-feature in Qt Creator: Help → About plugins. . . → Check Todo (experimental) → restart Qt Creator
- In the To-Do-Entries of the Actice Project you should see 5 hints where to change the code:



Note!

The resulting code will serve as a base for the exercise 07, where you implement the factory pattern.

Code Project

Project

Deadline is 18.12.2018

It's time to start your project!

- **Compiling and Running:**, with GUI, Presentation 5' (10P)
- **Complexity:** external libraries, threads etc. (10P)
- **Workload:** scope, quantity (5P)
- **Style:** Readability, variable naming, maintenance, STL (5P)
- **Bugs:** Memory leaks, initialised variables (5P)
- **Own ideas:** creativity (5P)
- **Total: 40 Points**

Note!

Review Checklist follows

Thank You

Questions

???