Creational Design Pattern

Hung Tran

Fpt software

August 11, 2021

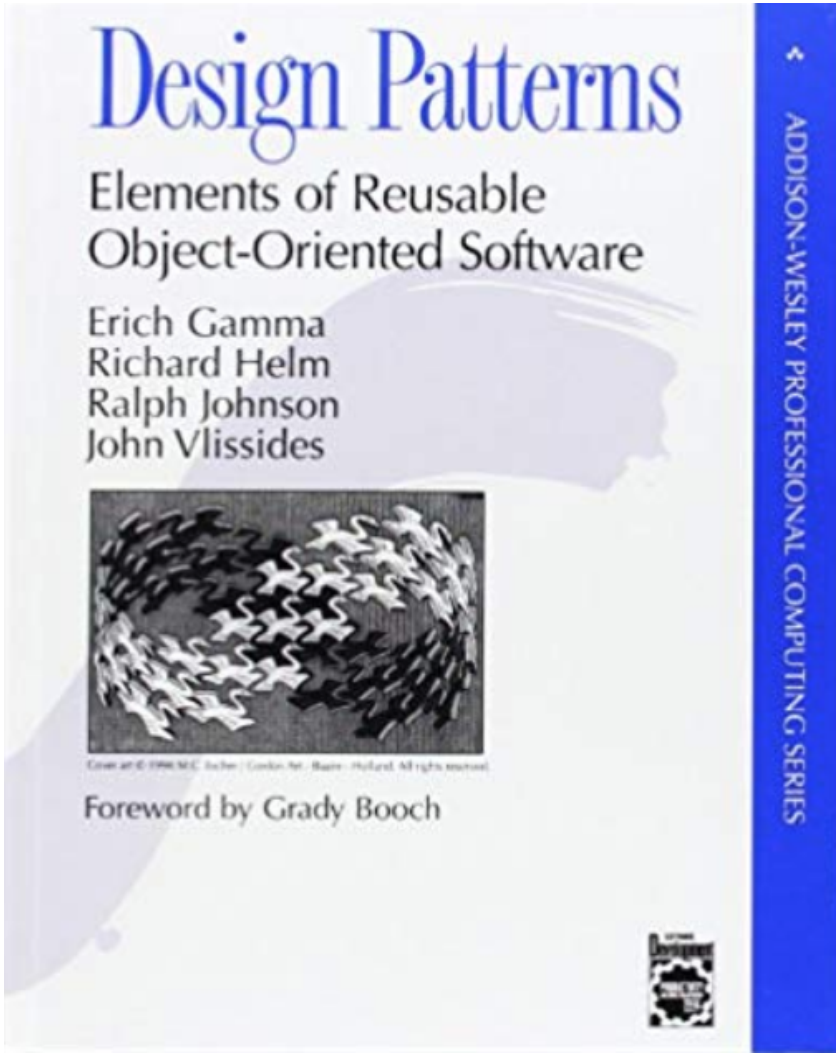
Outline

Today: Introduction

- What is Design Pattern?
- UML basics (to express design pattern)
- SOLID principles

Design Pattern

- Published in 1995.
- Known as gang of four design pattern.
- Describes solutions to common object oriented design problems.
- Examples in small talk and C++.
- Implemented directly in some languages.



What is Design Pattern?

- Language and domain independent strategies for solving common object-oriented design problems.
- These problems are recurring and can appear in all kinds of applications.
- Describes solutions to common object oriented design problems, irrespective of their language or platform.
- Patterns provide suggestions - different ways to solve these common problems.
- **Developers can use these suggestions as guidelines to create solution for their own problems.**

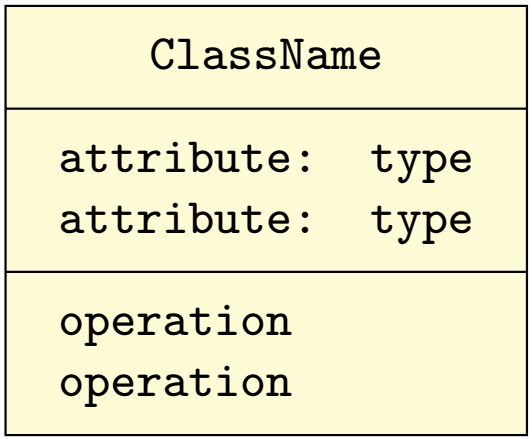
Classification of Design Pattern

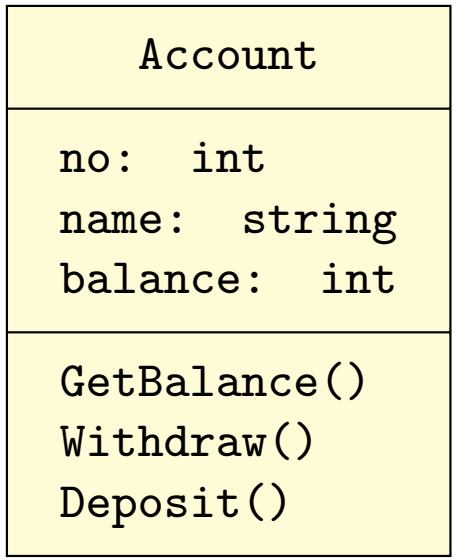
Table 1: Design Pattern Classification

Scope	**Creational**	**Structural**	**Behavioral**
Class	Factory Method	Adapter	Interpreter Template Method
Object	Abstract Factory, Builder, Prototype, Signleton	Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy,	Chain of Responsibility, Command, Iterator, Mediator, Memento, Observer, State, Strategy,

Overview of UML class diagram

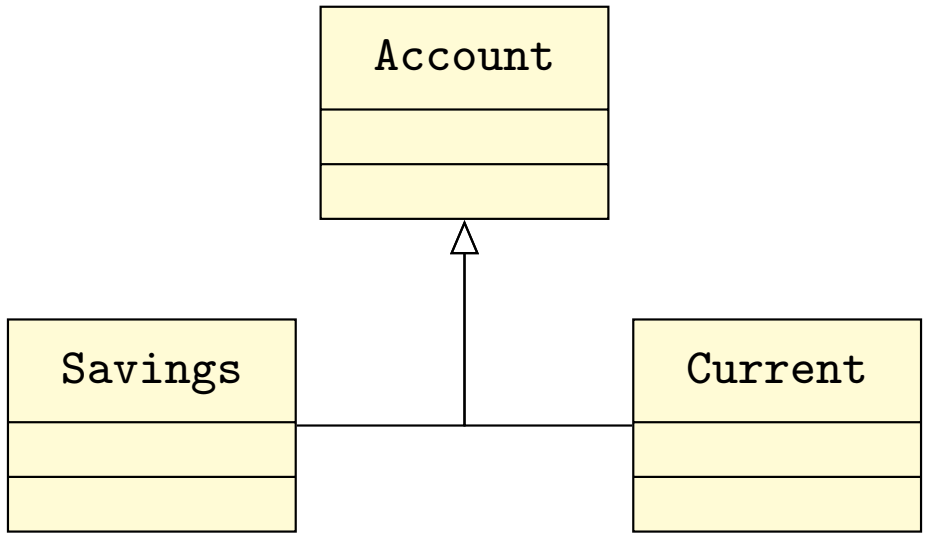
- It depicts relationships between classes that make up the pattern
- It is important to understand class notations to understand the structure of the pattern





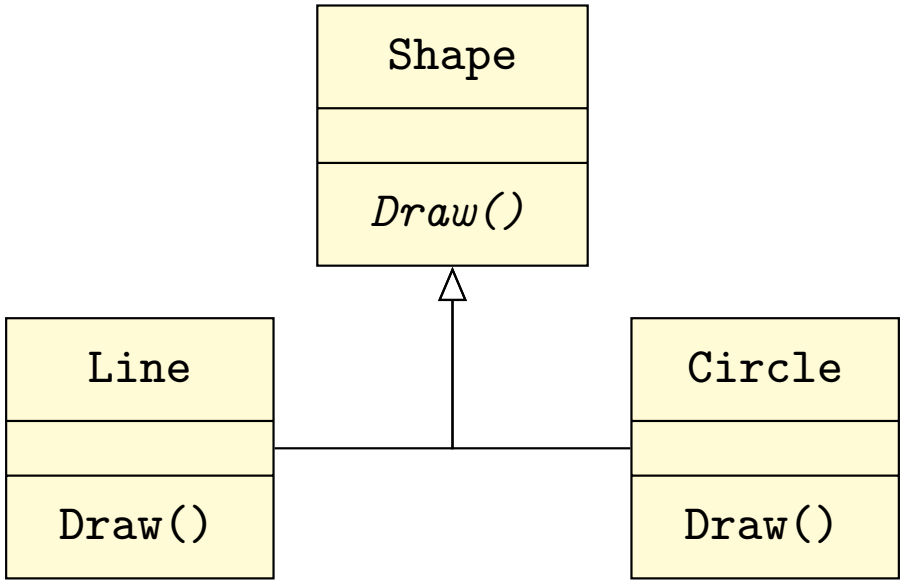
Overview of UML class diagram

- Inheritance (Generalization)



Overview of UML class diagram

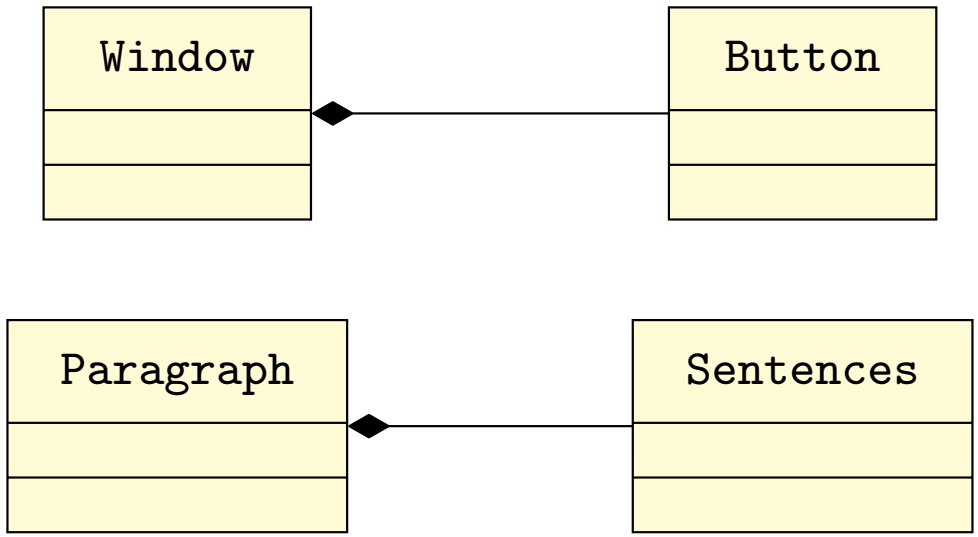
- Abstract class



Overview of UML class diagram

- Composition

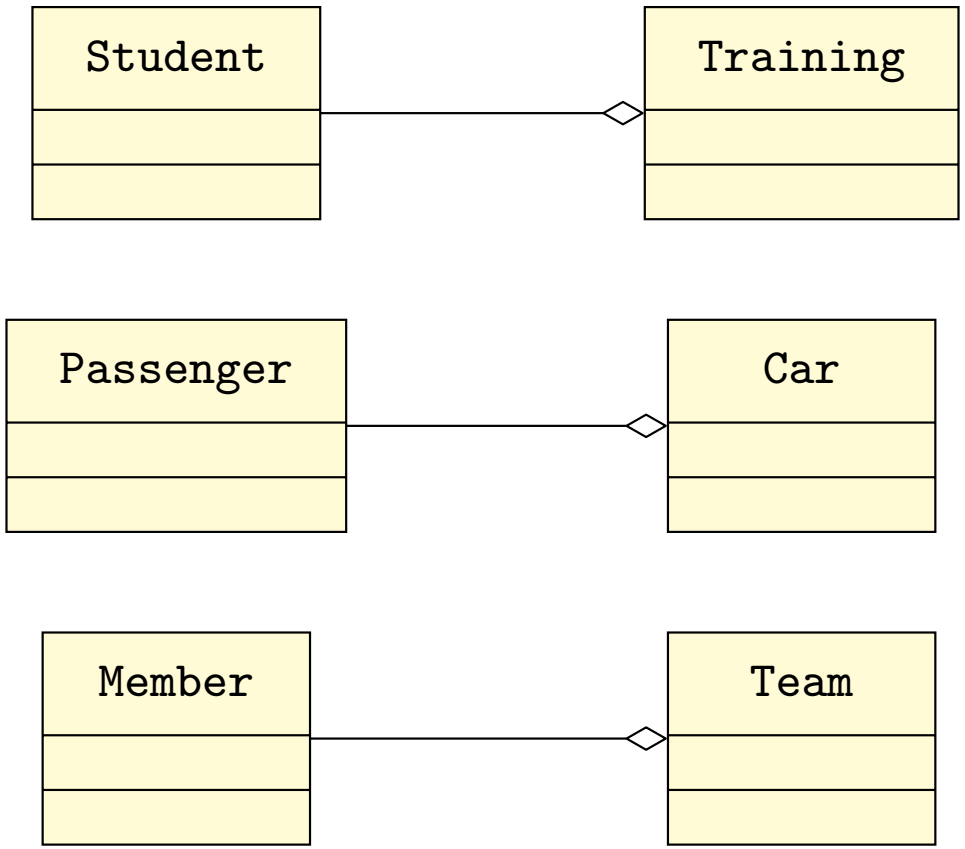
When container destroyed, all its elements destroyed



Overview of UML class diagram

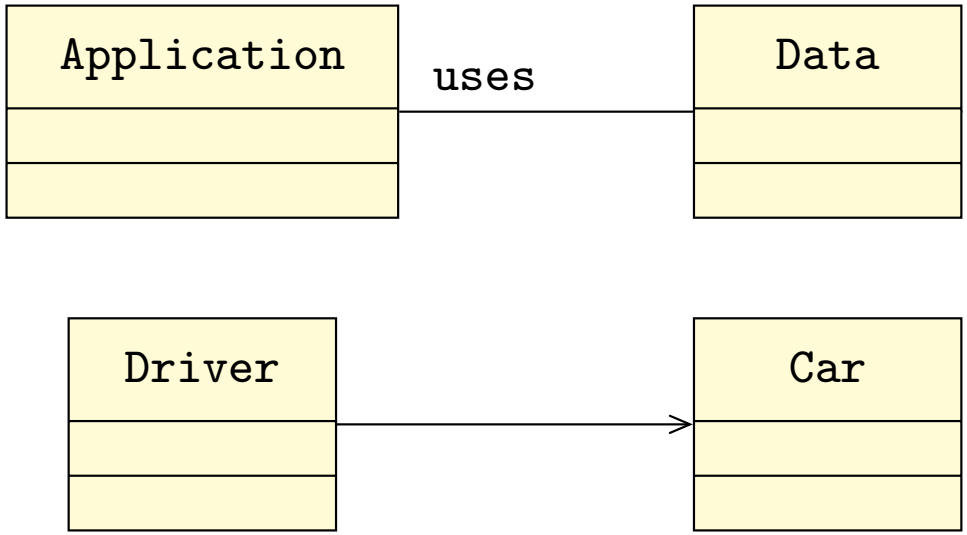
- Aggregation

When container destroyed, its elements may not be destroyed

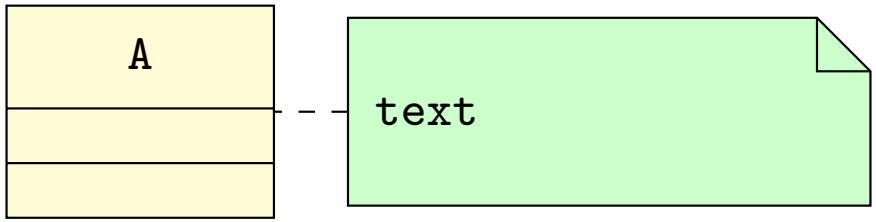


Overview of UML class diagram

- Association

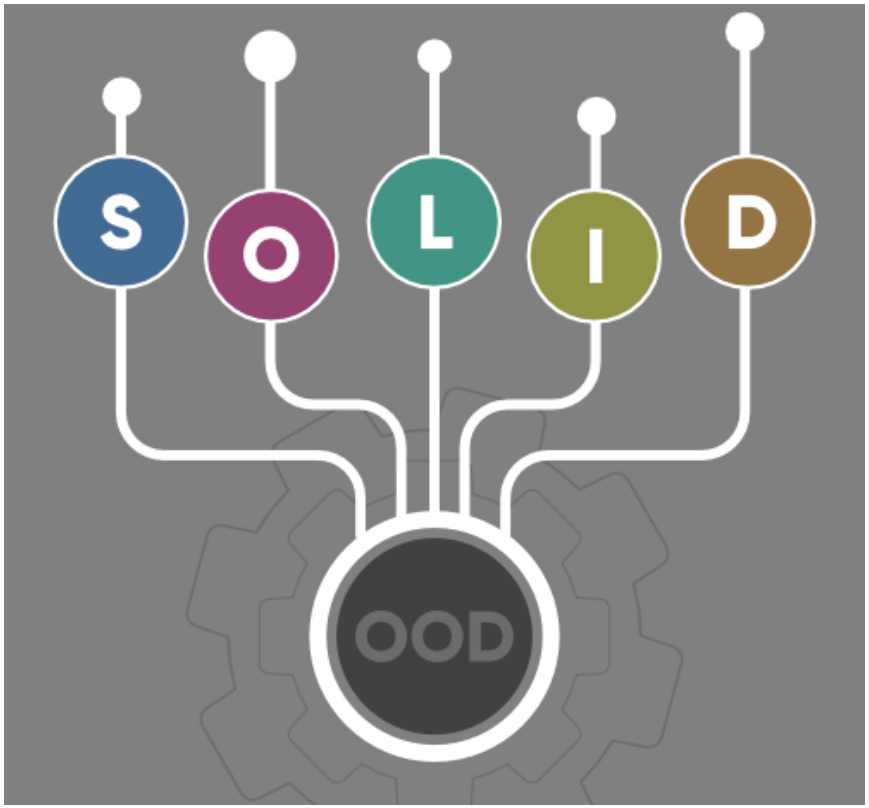


- Note



SOLID principles

- Single Responsibility Principle
- Open Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

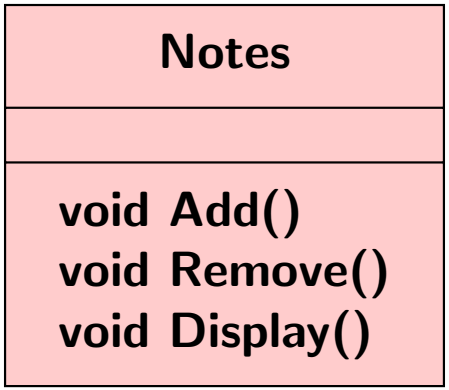


i. Single Responsibility Principle

A class should have only one reason to change

- Should have only one responsibility
- Class with multiple responsibilities break when changed
- Put each responsibility in a separate class

Example:

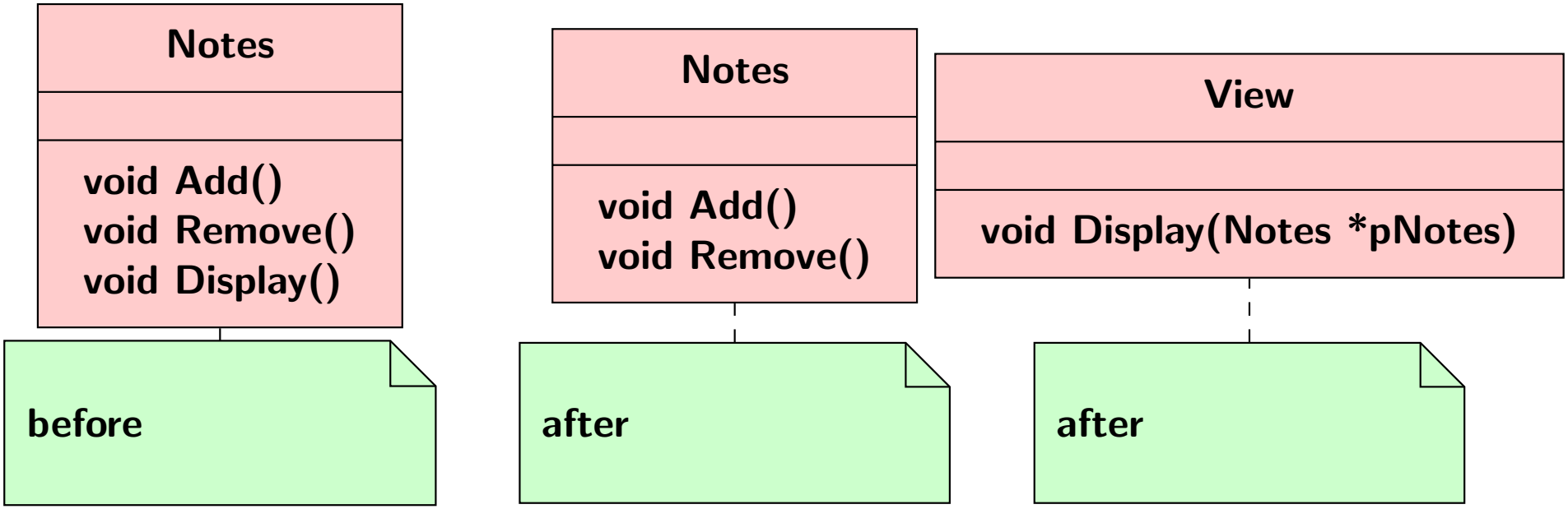


i. Single Responsibility Principle

A class should have only one reason to change

- Should have only one responsibility
- Class with multiple responsibilities break when changed
- Put each responsibility in a separate class

Example:



ii. Open-Closed Principle

Modules should be open for extension but closed for modification

- Modification to existing code leads to bugs and causes the software to break
- It should be possible to change behaviour of existing code without modification
- Instead the behaviour should be changed by adding new code
- Cornerstone of good design

Example: openClosePrin.cpp

ii. Open-Closed Principle

```
class Notes {
public:
  void Add() {}
  void Remove() {}
  void Display() {}
};
```

Before

```
class Notes {
public:
  void Add() {
      // if (contains('!')){
      // ...
      // }
  }

  void Remove() {

  }

  void Display() {

  }
};
```

ii. Open-Closed Principle

```
class Notes {
public:
  virtual void Add() {}

  void Remove() {}

  void Display() {}
};
```

After

```
class TaggedNotes : public
    Notes {
public:
  void Add() override {
    // if (contains('!'))
    // {
    //
    // }
  }
};
```

iii. Liskov-Substitution Principle

Subtypes must be substitutable for their base types

- Applies to inheritance relationship
- The inheritance relationship should be based on behavior
- A subclass must have all the behaviors of its base type and must not remove or change its parent behavior
- This allows a subclass to replace its base type in code
- New subclasses can be added without modifying existing code

Example: liskovsubPrin.cpp

iv. Interface Segregation Principle

Clients should not be forced to depend on methods they do not use

- An interface with too many methods will be complex to use (fat interface).
- Some clients may not use all the methods but will be forced to depend on them.
- Separate the interface and put methods based on the client usage.

Example: interfacesergregationPrin.cpp

iv. Interface Segregation Principle

Before

```
1 struct IFile {
2   virtual void Read() = 0;
3   virtual void Write() = 0;
4   virtual ~IFile() = default
      ;
5 };
```

After

```
1 struct IRead {
2   virtual  void Read() = 0;
3   virtual ~IRead() = default
      ;
4 };
5
6 struct IWrite {
7   virtual void Write() = 0;
8   virtual ~IWrite() =
      default;
9 };
```

v. Dependency Inversion Principle

Abstractions should not depend on details. Details should depend on abstractions

- Abstraction means an interface and details mean concrete classes.
- Using a concrete class directly creates a dependency, software becomes difficult to modify.
- Invert the dependency by using an interface rather a concrete class.

Example: dependencyinversionPrin.cpp

v. Dependency Inversion Principle

Before

```
class ImageReader {
public:
  virtual void Decode() = 0;
  virtual ~ImageReader() =
    default;
};
```

```
class ImageViewer {
  BitmapReader *m_Reader{};
public:
  void Display(){}
};
```

```
class BitmapReader : public
    ImageReader {
public:
  void Decode() {}
};
```

v. Dependency Inversion Principle

After

```
class ImageReader {
public:
  virtual void Decode() = 0;
  virtual ~ImageReader() =
    default;
};
```

```
class ImageViewer {
  ImageReader *m_Reader{};
public:
  void Display(){}
};
```

```
class BitmapReader : public
    ImageReader {
public:
  void Decode() {}
};
```

Further presentations: 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.

Thank You!

Upcoming presentation

Upcoming presentation

Upcoming presentation

Upcoming presentation

Upcoming presentation

Upcoming presentation