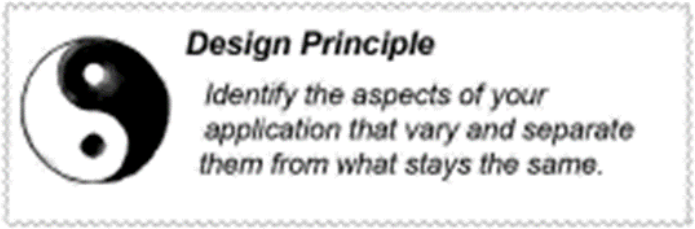
Design Patterns in Modern C++

Contents

- General Design Principles
- Design Patterns
  - Curiously Recurring Template Pattern
  - GoF patterns
  - Architectural Patterns
    - Layers / MVC

General design principles

- “Classes and methods should have a clear and single purpose”
- "A class should be open for extension but closed for modification"
- “Encapsulate what varies”
- “Favor composition over inheritance”
- “Program to an interface, not an implementation”



General design principles (cont'd)

- Loose coupling – high cohesion
  - loose coupling: classes have limited dependence on each other
  - high cohesion: everything that belongs together is in the same class

  →Enhances reusability.
- Tools can help to analyze source code for dependencies, e.g. CPPdepend

SOLID design principles

- Single-responsibility principle
  - Every class/module should be responsible for one portion of the overall system
- Open-closed Principle
  - It should be easy to extend a class's behavior without changing the code of the class itself.
- Liskov Substitution Principle
  - If type "A" is derived from type "B" then you should be able to substitute objects of type "B" for objects of type "A".

SOLID design principles (cont'd)

- Interface segregation principle
  - Clients using your code should not be forced into depending upon methods or other abstractions that they don't need.
  - Benefits:
    - Possible reduction in compile time
    - Maintainability
    - Proper separation of concerns
  - Example:
    - don't force subclasses to implement (virtual) functions they don't need

SOLID design principles (cont'd)

- Dependency inversion principle
  - High-level modules (classes which depend upon other, low-level classes of a program) should not depend on low-level modules directly. They should both depend upon an abstraction.
  - Benefits:
    - Loose coupling of software
    - Code reusability: interface can be reused.
    - Proper separation of concerns
  - Example:

SOLID design principles (cont'd)

NOT OK:

```
class CoffeeMachine {
  vector<int> status;
  …
}
```

◂ Low-level "module"

```
class CoffeeTest {
  void start(CoffeeMachine &machine) {
    for (auto bit: machine.status) {
       // operate on status bits
    }
  }
}
```

◂ High-level "module"

◂ **If the CoffeeMachine class implementation changes than the CoffeeTest class will need to change as well**

OK:

```
struct CoffeeStatusReader {
  virtual vector<int> readStatus();
}
```

◂ Shared abstraction

```
class CoffeeMachine : CoffeeStatusReader {
  vector<int> status;

  void readStatus() {
    for (auto bit: status) {

    }
  }
   …
}
```

◂ Low-level "module"

```
class CoffeeTest {
  void start(CoffeeStatusReader &reader) {
     reader.readStatus();
  }
}
```

◂ **The high-level module no longer depends upon the low-level module. The implementation of the low-level functionality can change without the high-level module needing to change as well.**

OO without inheritance

- Bjarne Stroustrup: OO Programming without inheritance
  - ECOOP Prague 2015
  - https://www.youtube.com/watch?v=xcpSLRpOMJM
  - 1:00:32

Curiously recurring template pattern

- Idiom, originally in C++, in which a class X derives from a class template instantiation using X itself as a template argument (Wikipedia)
- This pattern allows superclass to interact with subclass as if it knows about subclass's methods, without requiring virtual functions or runtime polymorphism.
- Purpose: This pattern allows the superclass to use the subclass type in its implementation, enabling compile-time polymorphism. This can be useful for implementing features like static polymorphism or adding functionality to derived classes without using virtual functions.

```
// class X derives from a class
// template Y, taking a template
// parameter Z, where Y is
// instantiated with Z = X.
// For example,
template<class Z>
class Y {};

class X : public Y<X> {};
```

Curiously Recurring Template Pattern

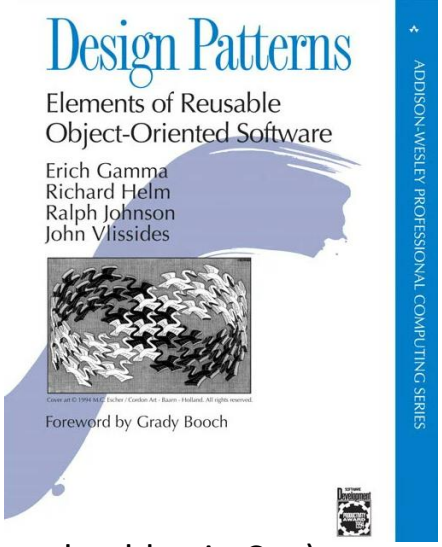
```
#include <cstdio>
template <class Derived>
struct Base
{
    void name() { static_cast<Derived*>(this)->impl(); }
protected:
    Base() = default; // prohibits the creation of Base objects
};
struct D1 : public Base<D1> { void impl() { std::puts("D1::impl()"); } };
struct D2 : public Base<D2> { void impl() { std::puts("D2::impl()"); } };

int main()
{
    D1 d1; d1.name();
    D2 d2; d2.name();
}
```

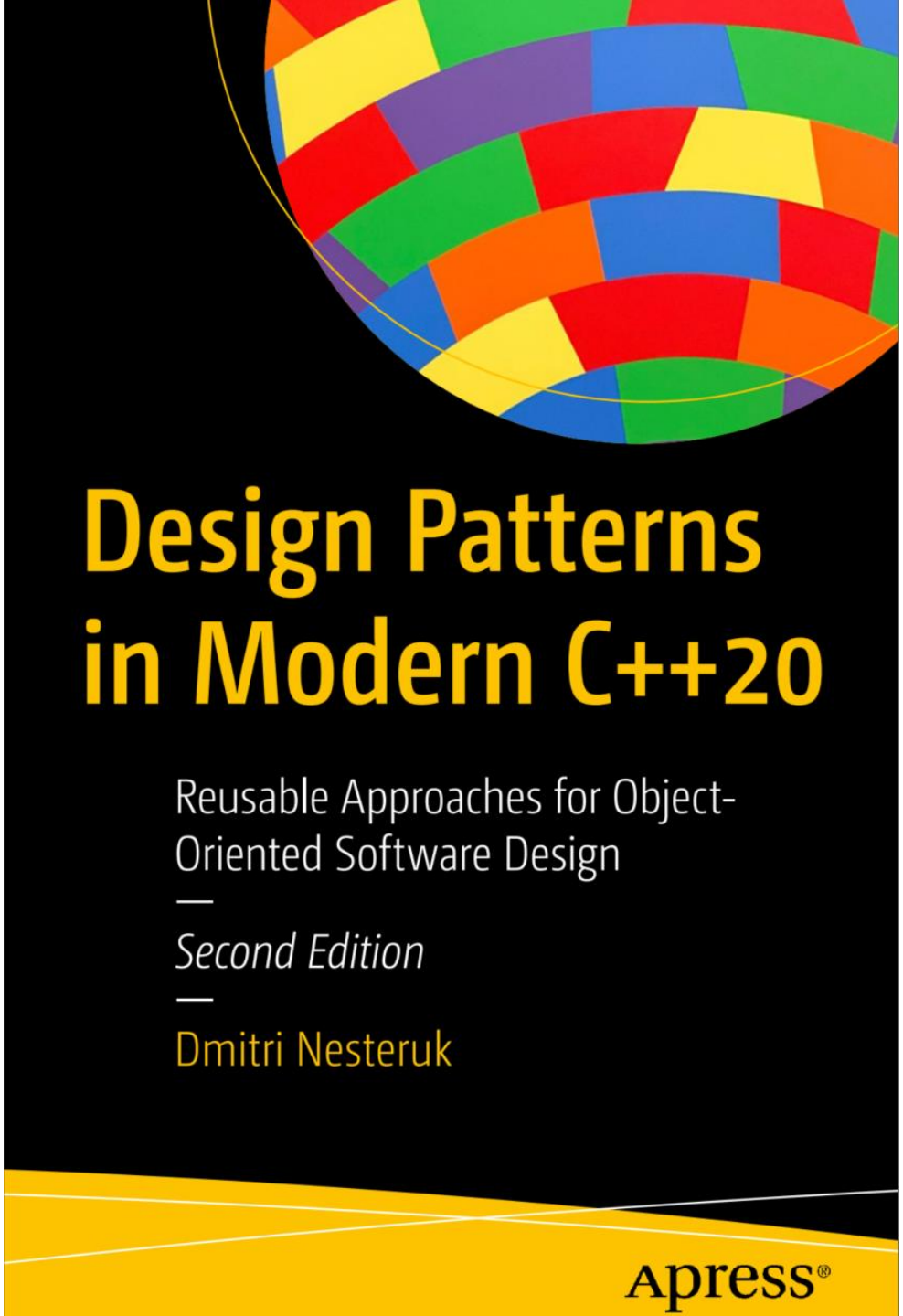
GoF Design Patterns

- Official definition:
  - each pattern describes a problem that returns regularly. You describe the core solution in such a way that this sulution is broadly usable, without each time *reinventing the wheel*.
- Design Patterns are 'smart' object oriented solutions for frequently occurring problems
- For complex problems you combine several Design Patterns
- If programmers know (some) design patterns, they can use the Design Patterns vocabulary to discuss the problems and their solutions
- This promotes clarity and compactness of the communication
- Many libraries and frameworks use design patterns
- If you understand design patternsn, you can easier understand such a library or framework
- The implementation of design patterns differs between programming languages

The gang of four



- The ‘inventors’ of design patterns:
  - Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides†.
- Book:
  - Design Patterns - Elements of Reusable Software (ISBN: 0.201-63361.2) (voorbeelden in C++)
  - In dutch: Design Patterns, de Nederlandse editie (ISBN: 90.430-0217.8)
  - Code in C++
- Examples for all patterns in "Modern C++":
  - http://en.wikibooks.org/wiki/C++_Programming/Code/Design_Patterns
- Design Patterns in Modern C++20, Reusable Approaches for Object-Oriented Software Design, Second Edition, Dmitri Nesteruk, Apress, 2022
- With Java examples: Java™ Design Patterns: A Tutorial (ISBN: 0201485397)
  download: http://www.patterndepot.com/put/8/DesignJava.PDF
- With Visual Basic examples:
  - Visual Basic Design Patterns: VB 6.0 and VB.NET (ISBN: 0201702657)
- Head First - Design Patterns (ISBN: 9780596007126)
- Head First - Design Patterns - Train je hersens in design patterns (ISBN: 9789077442715)


Design Patterns
in Modern C++20
Reusable Approaches for Object-Oriented Software Design
Second Edition
Dmitri Nesteruk
Apress®

Disclaimer

- Learning design patterns happens as follows:
  - “I don't understand it.”
  - "Those things look useful. I think I start understanding".
  - "I already know some patterns and use them if they have an added value".
  - "I know tens of patterns and use them as much as possible to obtain the most flexible design".

Disclaimer

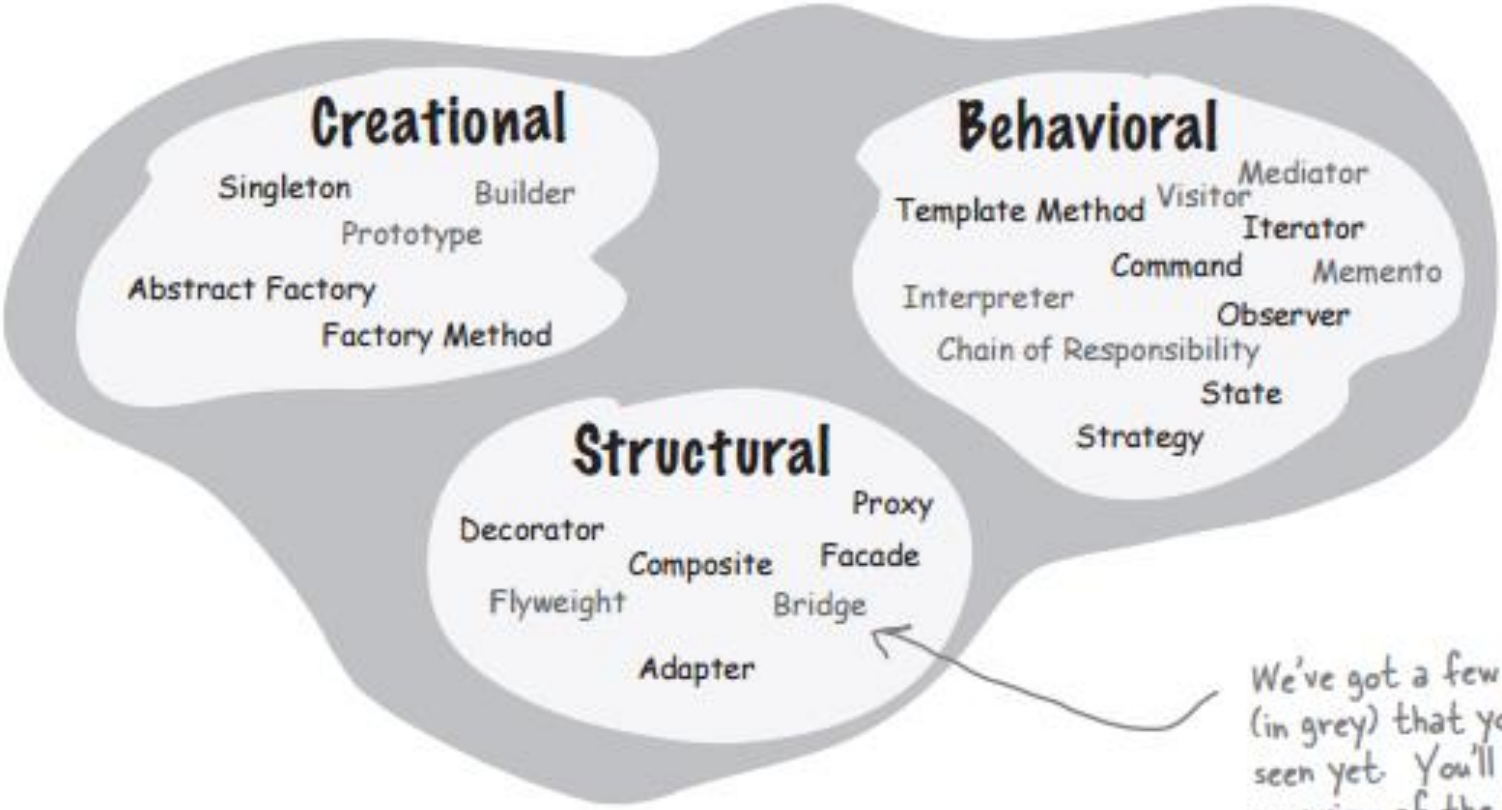
- This last step is called "pattern fever" and we should avoid it.
- Design patterns make your design more adaptable, more extensible, but also more complex.
- It's useless to implement patterns to make pieces of code that will most probably never be changed, more adaptable or extensible.

GoF Patterns: Categories

- Creational Patterns: create objects in a flexible way
  - Singleton
  - Factory
- Structural Patterns: Compositions of objects
  - Façade
  - Composite
  - Adapter
  - Decorator
  - Flyweight
  - Proxy
- Behavioral Patterns: Flexibele communication between objects
  - Iterator
  - Strategy
  - Observer
  - Template method
  - State
  - Command
  - Chain of Responsability
  - Memento
  - Visitor

Creational patterns involve object instantiation and all provide a way to decouple a client from the objects it needs to instantiate.

Any pattern that is a **Behavioral Pattern** is concerned with how classes and objects interact and distribute responsibility.



We've got a few patterns (in grey) that you haven't seen yet. You'll find an overview of the these patterns in the appendix.

Structural patterns let you compose classes or objects into larger structures.

Singleton

- Creational design pattern.
- Ensures that only one instance for a class can be created.
- Ensures easy access to that instance from all code
- Applicable as:
  - analysis shows that a single instance suffices (e.g. PC has only one keyboard)
  - one instance is enough and that instance needs much time or resources to be created
  - Often used for holding configuration parameters, database connections, etc.
- Implementations
  - Public static getter with private (or protected) constructor
  - Singleton wrapper: If the class you want to use as a singleton, is not designed according to the singleton pattern

Singleton

```
#ifndef DATABASE_H
#define DATABASE_H

#include <iostream>
#include <string>

class Database
{
protected:
    Database() { }
public:
    static Database& get()
    {
        // thread-safe since C++11
        static Database database;
        return database;
    }
    void setName(const std::string& n) {name = n;}
    const std::string& getName() {return name;}
    Database(Database const&) = delete;
    Database(Database&&) = delete;
    Database& operator=(Database const&) = delete;
    Database& operator=(Database &&) = delete;
private:
    std::string name="Jef";
};
#endif // DATABASE_H
```

Singleton

```
#include "database.h"

int main()
{
    Database& d1 = Database::get();
    std::cout << d1.getName() << std::endl;  // default "Jef"
    d1.setName("Jan");
    std::cout << d1.getName() << std::endl;  // "Jan"

    Database& d2 = Database::get();
    std::cout << d2.getName() << std::endl;  // still "Jan", not default "Jef"

    d2.setName("Robert");
    std::cout << d2.getName() << std::endl;  // "Robert"
    std::cout << d1.getName() << std::endl;  // also "Robert"

    return 0;
}
```

Alternative: Singleton Wrapper

```
template<typename T>
class Singleton
{
public:
   static T& Instance(void)
   {
      static T singleton;
      return singleton;
   }

private:
   Singleton(void);
};
```

Singleton Wrapper

```
class App
{
public:
    void Exec(int argc, char **argv);
};

int main(int argc, char *argv[])
{
    try
    {
        if (argc > 1)
        {
            Singleton<App>::Instance().Exec(argc,argv);
         }
         else
           …
    }
    else
       …
}
```

Iterator

- Behavioral design pattern.
- aka Cursor
- You iterate through the elements of an arbitrary collection, without knowing the implementation of the collection.
- You can change the implementation of the collection without changing the code of the iterator.
- Built-in in C++ STL, Java, C#

Iterator

```
    // Standard Template Library example

list<int> L;
L.push_back(0); // Insert a new element at the end
L.push_front(0);// Insert a new element at the beginning

L.push_back(5);
L.push_back(6);

list<int>::iterator i;
for(i=L.begin(); i != L.end(); ++i)
       cout << *i << " ";
```

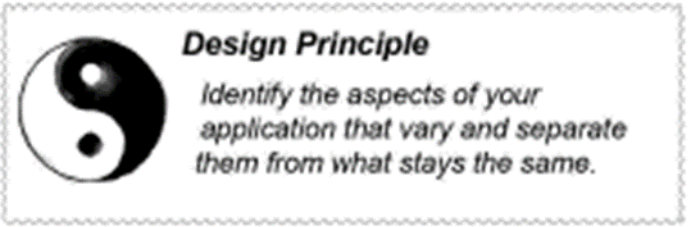
Strategy

- Behavioral design pattern.
- To use if the behavior of a class can be realized through more than one algorithm
- The client of the class chooses which algorithm to execute
- New algorithms can be added later without changing the class itself

'A class should be open for extension but closed for modification'

Strategy: design principles

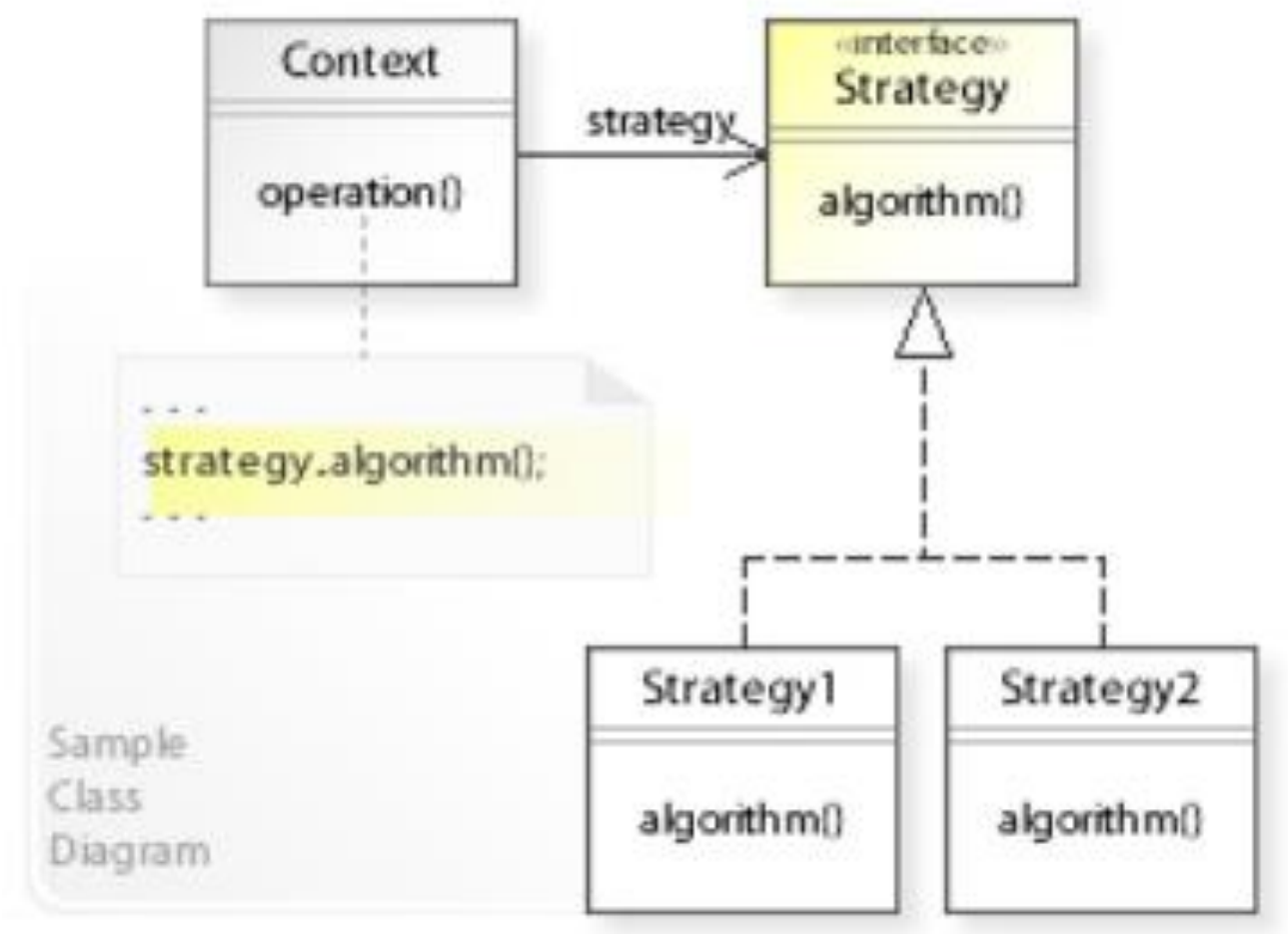
- “Classes and methods should have a clear and single purpose”
- "A class should be open for extension but closed for modification"
- “Encapsulate what varies”



Strategy: example

- See: example Textprocessor

Strategy UML class diagram



Strategy

- The advantages of this pattern are:
  - Since all strategies implement the same interface, they are exchangeable (through the use of polymorphism)
  - The client can change its strategy at runtime, e.g. by providing a method setStrategy(Strategy&)
  - The different strategies are reusable
  - It's very easy to adapt strategies or to add new ones

Strategy

- In this pattern we use a combination of polymorphism and composition for code reuse
- This way of reuse is more flexible and dynamic than inheritance, which is a static form of reuse
- This leads us to the next principle:
  - “Favor composition over inheritance”.

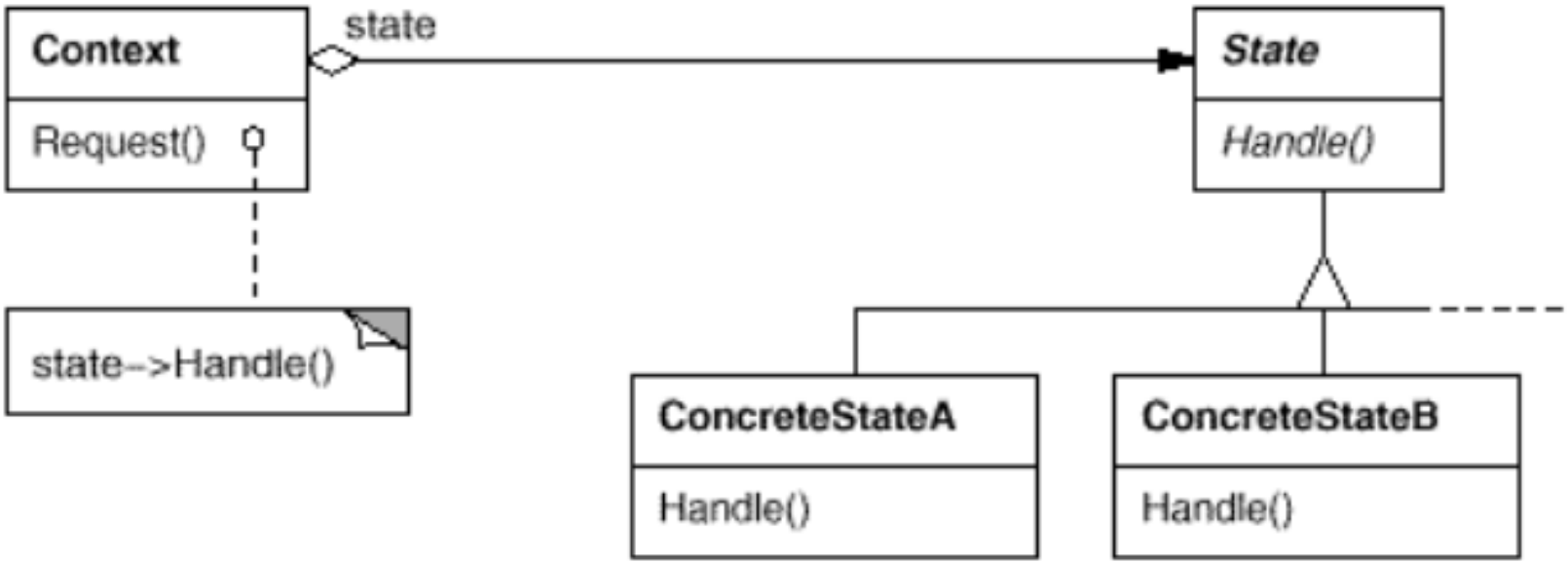
Dynamic vs. Static Strategy

- Dynamic:
  - keeps a pointer/reference to the strategy being used.
  - Change to a different strategy by changing the reference.
- Static:
  - Bake any strategy right into the type using a template class.
  - choose the strategy at compile time and stick with it (there is no scope for changing your mind later on).

State

- Behavioural pattern
- An object changes its behavior when its internal state changes.
- It looks like the object changes class.
- Interesting if the object has states and state transitions.
- Ensures that you don't have a plethora of if statements in your methods.

State



- The state of Context can change from Context (e.g., Request) or from a ConcreteState (e.g., Handle of ConcreteStateB).
- State is similar to Strategy, but choice between ConcreteState A AND B is not done by the Client, but by Context and/or ConcreteState(s).

State

- In the implementation, we used a combination of polymorphism, composition and delegation, just like in the pattern Strategy
- Therefore, the class diagram looks the same as for Strategy.
- Nevertheless, there are important differences.

Strategy vs. State

- Strategy:
  - This pattern is used to dynamically select from different algorithms for the same task.
- State:
  - This pattern is used to implement objects with complex internal states.

Strategy vs. State

- Strategy:
  - The Strategy that a class uses can be chosen by a user of this class.
  - Strategies are thus also externally visible.
- State:
  - The State of an object can only be changed by this object or by another state.
  - States are therefore visible only internally.

State

- See code State\

Exercise

A garage door can be opened or closed using a remote control. This remote control consists of 2 buttons that work as follows:

- the Open/Close button:
  - opens the door when it is closed or closing
  - closes the door when it is open or opening
  - allows the door to continue moving when it was locked
- the Lock button:
  - locks the door in its current position when the door was moving (opening or closing).
  - does nothing when the door is fully open or closed.

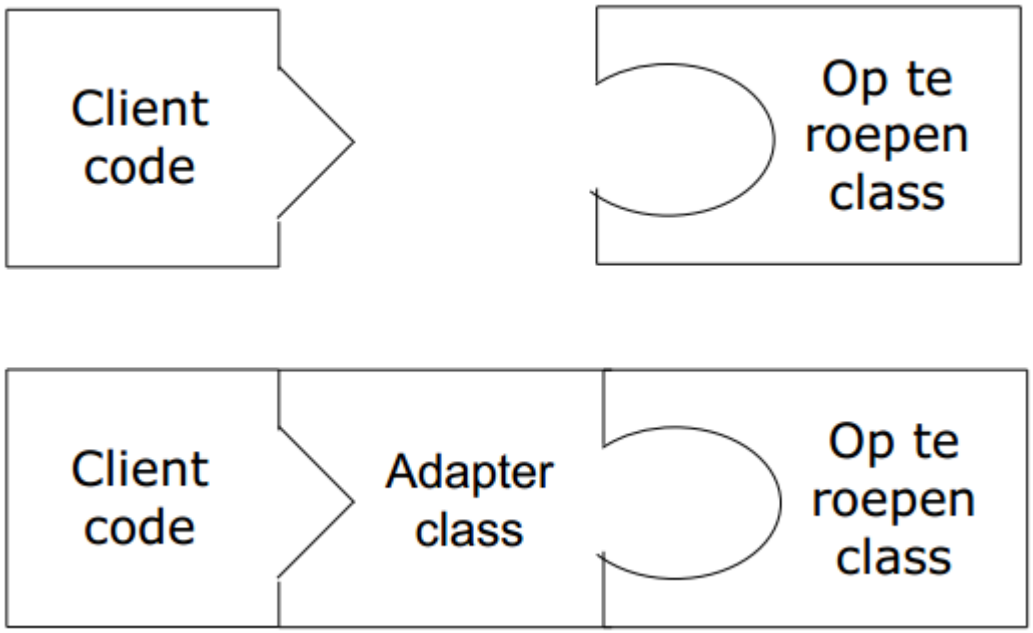
Exercise

- Create the design of the classes in UML
- Implement using the State pattern
- Test in a main program

Adapter

- Structural design pattern
- Convert the programming interface (methods, properties) of a class to the programming interface expected by the client.
- Allow classes to work together that cannot because the interface is different from the expected interface.

Adapter


Client code
Op te roepen class
Client code
Adapter class
Op te roepen class

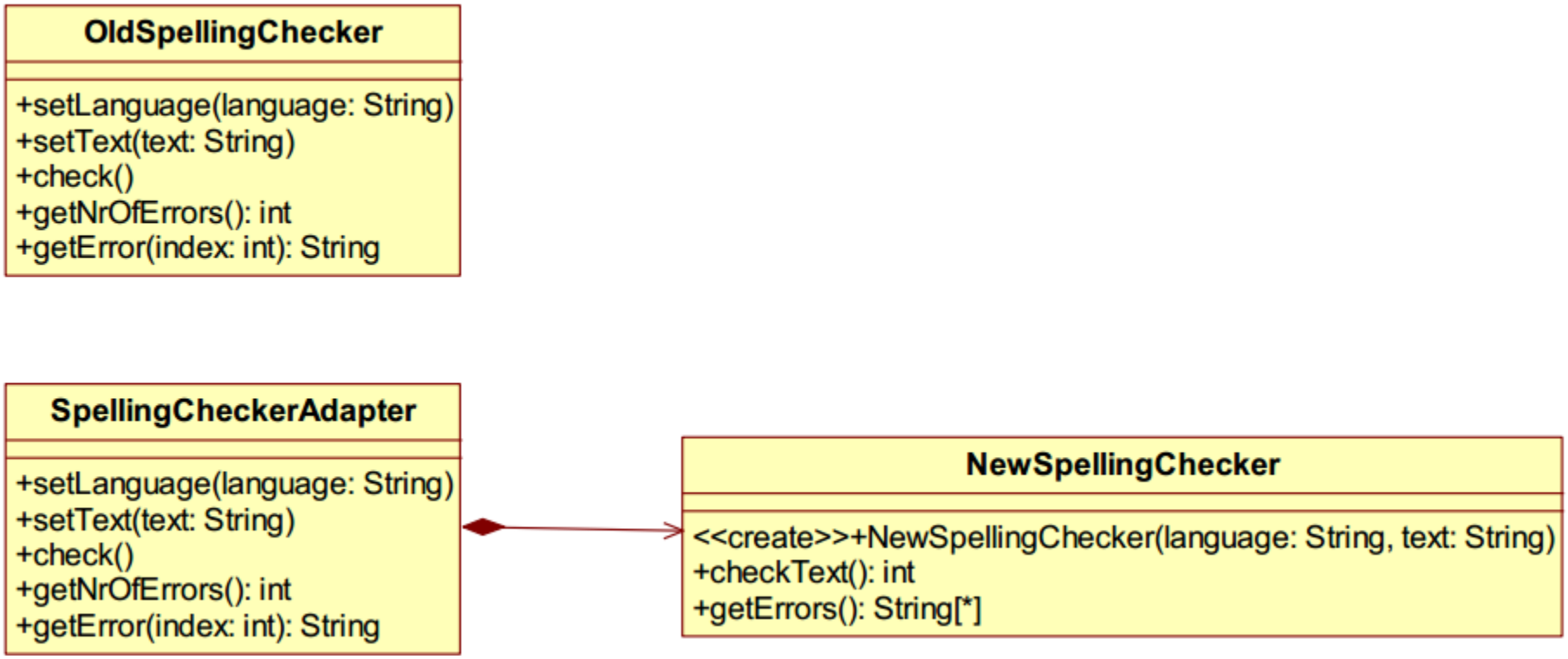
Adaptor: applications

- Your application uses a class from a class library in many places.
- You want to use another class (because of license change, performance, memory usage,...).
- But the other class has a different interface.
- Changing code in many places to use the new class instead of the old one takes a lot of time and risks bugs.
- Solution: Build around the new class an adapter class with the same interface as the old class and call the adapter class.

Adapter

WantsToUseThisClass
+operation1()
+operation2()
+operation3()
<<use>>
Client
<<use>>
Adapter
+operation1()
+operation2()
+operation3()
MustUseThisClass
+operationA()
+operationB()
Adapter heeft dezelfde interface als WantsToUseThisClass
Adapter roept methods op van MustUseThisClass om deze interface te implementeren

Adapter: example



Factory Method: scenario

- Consider a very simple model of building construction
- A building is made of walls.
  - A wall is a structure consisting of
    - Start and end 2D points defining the two bottom points of the wall
    - The wall elevation: the height or the z coordinate of the bottom of the wall relative to some baseline
    - The height of the wall

Factory Method: scenario

```
class Wall
{
  Point2D start, end;
  int elevation, height;
public:
  Wall(Point2D start, Point2D end, int elevation, int height)
    : start{start}, end{end}, elevation{elevation}, height
    {height} { }
};
```

To make things a bit more complicated, we can expand this “thin” wall into a SolidWall that has information about the width of the wall (i.e., how thick it is) and what material it’s made of:

Factory Method: scenario

```
enum class Material
{
  brick,
  aerated_concrete,
  drywall
};

class SolidWall : public Wall
{
  int width;
  Material material;
public:
  SolidWall(Point2D start, Point2D end, int elevation,
    int height, int width, Material material)
    : Wall{start, end, elevation, height},
      width{width}, material{material} {}
};
```

Factory Method: scenario

- Assume
  - Aerated concrete cannot be used for underground construction.
  - Minimum brick wall width is 120mm.

```
SolidWall::SolidWall(const Point2D start, const Point2D end,
                     const int elevation,
                     const int height, const int width,
                     const Material material)
  : Wall{start, end, elevation, height},
    width{width}, material{material}
{
  if (elevation < 0 && material == Material::aerated_concrete)
    throw invalid_argument("elevation");

  if (width < 120 && material == Material::brick)
    throw invalid_argument("width");
}
```

Factory Method: scenario

- It feels wrong to include validation in the c'tor
- We are constrained to exceptions
- We cannot, for example, simply refuse to construct a `SolidWall`, returning some error code or null value.
- We also can only use specific types of walls made by factories.
- We now create 2 factory methods to build 2 specific types of walls.

Factory Method: solution

```
class SolidWall : public Wall
{
    int width;
    Material material;

protected:
    SolidWall(const Point2D start, const Point2D end,const int elevation,
    const int height, const int width, const Material material);

public:
    static SolidWall create_main(Point2D start, Point2D end,
    int elevation, int height)
    {
        return SolidWall{start, end, elevation, height,375,
        Material::aerated_concrete};
    }

    static unique_ptr<SolidWall> create_partition(Point2D start,Point2D end,
    int elevation, int height)
    {
        return make_unique<SolidWall>(start, end, elevation,height, 120,
        Material::brick);
    }
};
```

Factory Method: solution

```
const auto main_wall =
     SolidWall::create_main({0,0},{0,3000},
                            2700, 3000);
cout << main_wall << "\n";

// start: (0,0) end: (0,3000) elevation: 2700 height: 3000
// width: 375 material: aerated concrete
```

Factory: scenario

- We still cannot allow aerated concrete underground.
- We could, for example, redefine the factory method like this:

```
static shared_ptr<SolidWall> create_main(Point2D start,
                    Point2D end, int elevation, int height)
{
   if (elevation < 0) return {};//default ctor of shared_ptr=nullptr
   return make_shared<SolidWall>(start, end, elevation, height,375,
   Material::aerated_concrete);
}
// usage: this will fail
const auto also_main_wall = SolidWall::create_main
                                ({0,0}, {10000,0}, -2000, 3000);
if (!also_main_wall)
       cout << "Main wall not created\n";
```

Factory: scenario

- Imagine that interior walls cannot be created if they intersect other interior walls.
- How would you implement this? You need to track every partition wall created so far, but where would you store this information?
- It doesn’t make sense to store it in SolidWall – particularly if similar mechanisms also require polymorphic interactions.

Factory: solution

```
class WallFactory
{
     static vector<weak_ptr<Wall>> walls;
public:
     static shared_ptr<SolidWall> create_main(Point2D start,Point2D end, int elevation, int height)
     {
     // as before
     }
     static shared_ptr<SolidWall> create_partition(Point2D start,Point2D end,int elevation, int height)
     {
          const auto this_wall = new SolidWall{start, end, elevation, height, 120, Material::brick};
          // ensure we don't intersect other walls
           for (const auto wall: walls)
           {
            if (auto p = wall.lock())
            {
                  if (this_wall->intersects(*p))
                  {
                        delete this_wall;
                        return {};
                   }
             }
            }
            shared_ptr<SolidWall> ptr(this_wall);
            walls.push_back(ptr);
            return ptr;
     }
};
```

Factory (Method) and Polymorphism

- A creation method can return polymorphic types
- Of course, return (ordinary or smart) pointers

```
enum class WallType
{
  basic,
  main,
  partition
};
```

Factory (Method) and Polymorphism

- We can define the following polymorhic factory method:

```
static shared_ptr<Wall> create_wall(WallType type, Point2D start, Point2D end, int elevation, int height)
{
    switch (type)
    {
    case WallType::main:
        return make_shared<SolidWall>(start, end, elevation, height,375, Material::aerated_concrete);
    case WallType::partition:
        return make_shared<SolidWall>(start, end, elevation, height,120, Material::brick);
    case WallType::basic:
        return make_shared<Wall>(start, end, elevation, height);
    }
    return {};
}
```

Factory (Method): summary

- A *factory method* is a class member that is used for creating an object. It typically replaces a constructor.
- A *factory* is typically a separate class that knows how to construct objects

Factory (Method): advantages over constructor call

- A factory can say no, meaning that instead of actually returning an object, it can return, for example, a default-initialized smart pointer or a nullptr.
- Naming is better and unconstrained, unlike the constructor name. You can call the factory methods whatever you want.
- A factory can implement caching and other storage optimizations; it is also a natural choice for approaches such as pooling or the Singleton pattern.

Exercise

- Readings

Observer

- Automatically issues alerts to all objects dependent on an object (‘observers’) when 'subject' changes.
- Subject or Observable: object that sends out alerts when its status changes.
- Observer: object that receives an alert when Subject changes.
- The design pattern allows to:
  - Add observer classes without changing the Subject class.
  - Link multiple observers to one subject

Exercise

- observer.cpp:
  - Implement the notify, subscribe and unsubscribe methods
  - Add the code to test the observer pattern.

Decorator

- Structural design pattern.
- aka Wrapper
- Make an object that allows to add responsabilities in a dynamic way

Inheritance

- Seems to be the obvious technique to add responsabilities, but it has a lot of drawbacks
  - New functionality can be added to only one class, unless you use multiple inheritance, which is complex.
  - It's impossible to add functionality dynamically (i.e. at runtime)
  - There is a strong dependency between superclass and subclass (implementation inheritance)

Decorator

- "Classes should be open for extension, but closed for modification”
- This means: extend a class without changing the class itself

→ Decorator pattern

Step 1

- In its most simple form:

- MyDecorator will contain the new functionality
- MyDecorator has an attribute of type MyClass so it can use the original functionality

Step 1

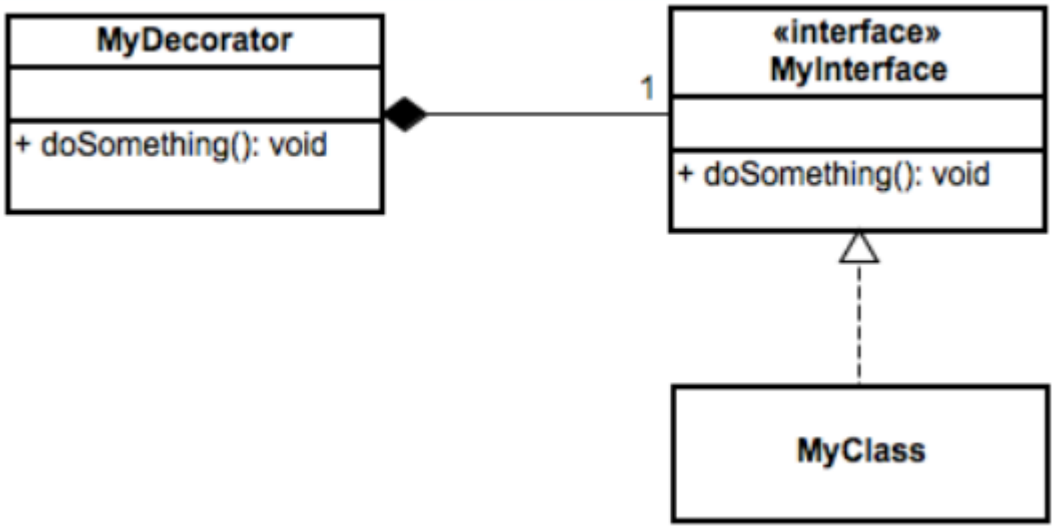
- Inheritance is replaced by composition and delegation, as in strategy.
  - “Favor composition over inheritance”
- This way, functionality can be added dynamically

```
// use original functionality…
MyClass* c = new MyClass();

// … or extend c with new functionality
MyDecorator* d = new MyDecorator(c);
```

Step 2

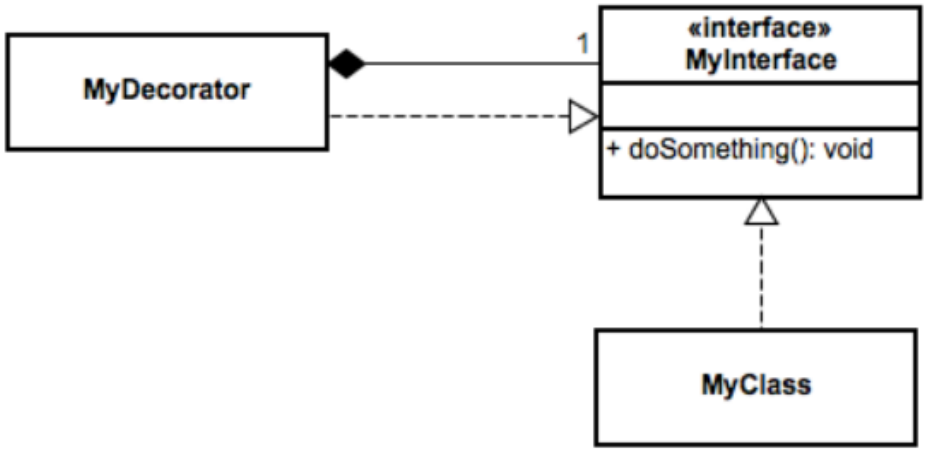
- Use polymorphism to use the functionality in MyDecorator for several classes:



- Now MyDecorator can be used for all classes that inherit from MyClass.

Step 3

- Finally we make another small change to the design



- MyDecorator now also inherits from MyInterface.
- This has two important consequences:

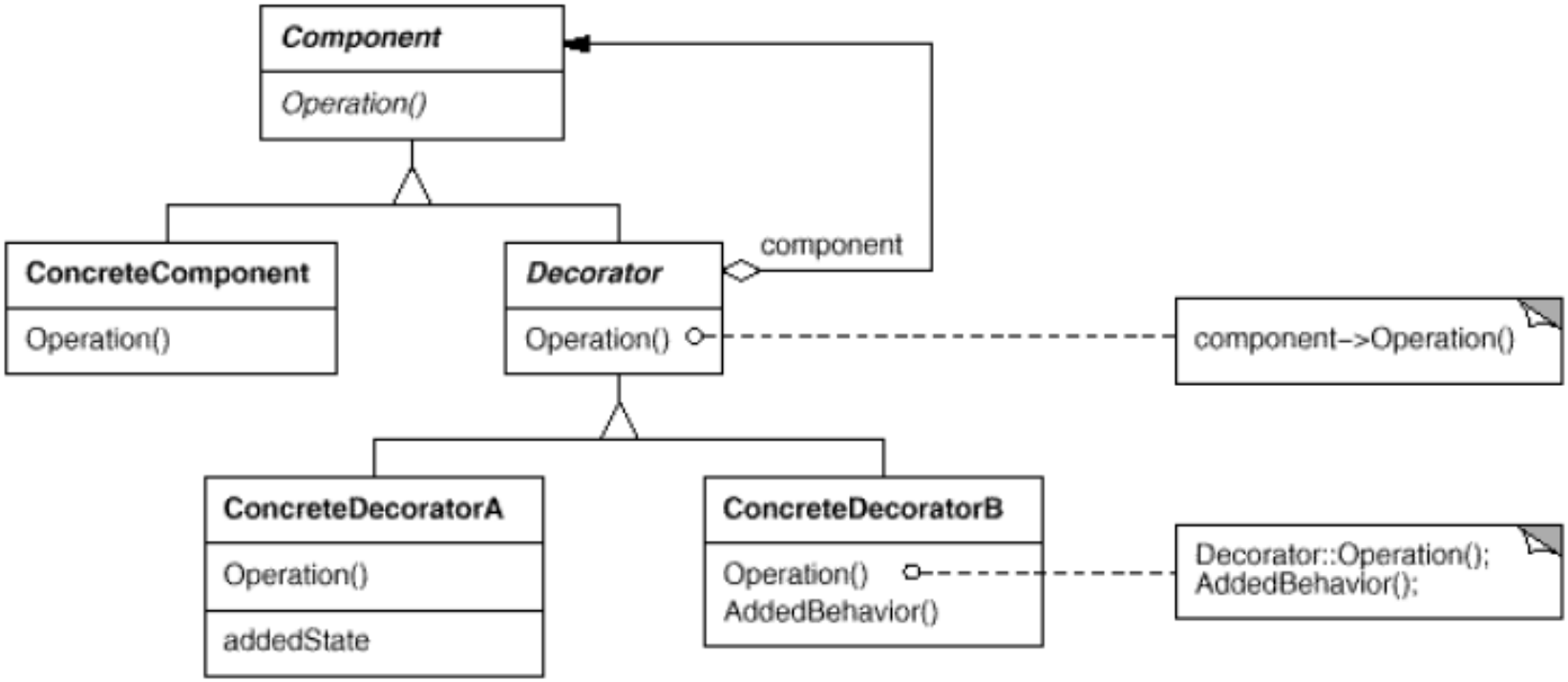
Step 3

- Since MyDecorator and MyClass inherit from the same base class, they can be exchanged
  - Anywhere we used MyClass we can now also use MyDecorator
- Since MyDecorator has an attribute of type MyInterface an MyDecorator is itself inherited from MyInterface, this attribute can again be a decorator.
  - This enables the creation of a chain of decorators, in which each object performs a specific task and delegates to the next object in the chain.
  - The last object in the chain is the original object, which is not a decorator.

Step 3

- The result is the complete decorator pattern
- You can have several decorators and several concrete classes
- If necessary, all decorators can have a common superclass too.

Decorator


Component
Operation()
ConcreteComponent
Operation()
Decorator
Operation()
component
component->Operation()
ConcreteDecoratorA
Operation()
addedState
ConcreteDecoratorB
Operation()
AddedBehavior()
Decorator::Operation();
AddedBehavior();

Conclusion

- This design allows to add functionality in a dynamic way to existing classes
- This functionality can be added to several classes as long as these classes share a common superclass
- The original classes don't need to be changed!

Conclusion

- We applied the known design principles :
  - “Classes and methods should have a clear and single purpose”
  - “Encapsulate what varies”:
  - “Classes should be open for extension, but closed for modification”
  - “Favor composition over inheritance”
  - “Program to an interface, not an implementation”

Drawbacks

- Extreme use of decorators leads to complex structures of nested objects
  - This might be harmful for performance
  - It's also more difficult to determine the equality of objects

- We still make objects, not onions!

Example

- See example decorator_shape.cpp

Decorator: Excercise

- Start from a class for simple coffee (= black) with 1 ingredient: coffee and price = 1€.
- Make a decorator to add milk. Ask for ingredients through decorator: coffee + milk. Costs 0.5 € extra.
- Make a second decorator to add sugar. Costs 0.7 € extra.
- Make sure, after adding milk, you can also add sugar.
- Avoid the use of raw pointers.

Façade

- Structural design pattern.
- Represents a subsystem with one class in which multiple classes work together
- The client does not have to work out the cooperation of the classes of the subsystem itself, but only has to address the facade class.
- You can change the subsystem internally, without having to change the clients of the subsystem, as long as the facade class retains the same interface.
- If applicable, you can allow the client to directly access individual parts of the subsystem.

Façade

- Use case:
  - we have three different subsystems that are expected to work in concert in order to generate random magic squares.
  - If you add up the values in any row, column, or any diagonal, you'll get the same number – in this case, 33.

$$\begin{bmatrix} 1 & 14 & 14 & 4 \\ 11 & 8 & 6 & 9 \\ 8 & 10 & 10 & 5 \\ 13 & 2 & 3 & 15 \end{bmatrix}$$

Façade

- Use case (cont'd) :
  - if we want to generate magic squares, we can imagine it as an interplay of three different subsystems:
  - Generator: A component which simply generates a sequence of random numbers of a particular size
  - Splitter: A component that takes a rectangular matrix and outputs a set of lists representing all rows, columns, and diagonals in the matrix
  - Verifier: A component that checks that the sums of all lists passed into it are the same
- See code generator.cpp