Design Patterns in C++

A mini-lecture series

CSE498 Collaborative Design (W) - Secure and Efficient C++ Software Development

03/26/2025

Kira Chan

https://cse.msu.edu/~chanken1/

Definition

- General and reusable solution to a commonly occurring problem in software design
- It is a design pattern, not a snippet of code
- Pair<Problem, Solution>

- Introduced by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (a.k.a., the group of four) in 1994
- Introduced the original 22 class design patterns in their book
- The book is written by analysing code and extracting patterns from them, not designing the patterns to be implemented

Analogy

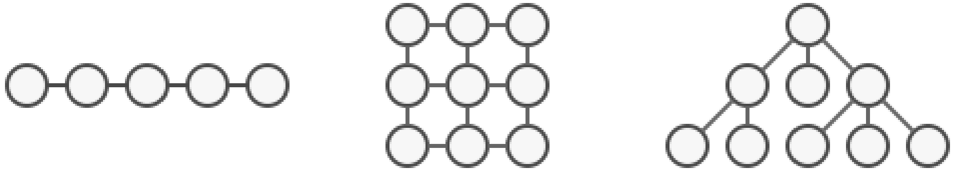
- They are similar to a blueprint that you can customize to solve a particular problem
- Can be categorized based on complexity, levels of abstraction, etc.

Three classification of patterns

- Creational patterns
    - Designed for class instantiation
    - Examples: Abstract factory, builder, singleton, etc.
- Structural patterns
    - Designed for class composition and package structure
    - Examples: Adapter, Bridge, Façade, Flyweight, etc.
- Behavioral patterns
    - Design for communications between classes
    - Examples: Interpreter, mediator, observer, etc.

Example: Iterator pattern

- Turns out traversing a data is a commonly reoccurring theme
- Data can be organised in many ways, and even traversed in many ways in the same container



Various types of collections.

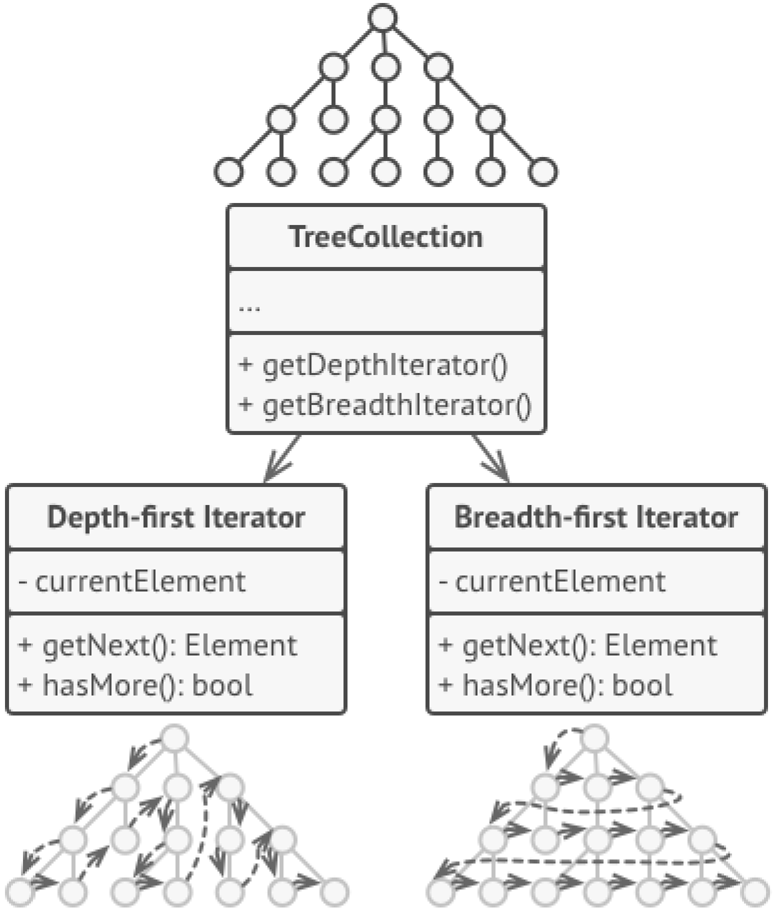
https://refactoring.guru/design-patterns/iterator

Iterator Pattern

- Generate an iterator that describes how an object shall be traversed

- Implement a getNext() function that describes how to step to the next item
- Implement a hasMore() function to check whether there are items still to be traversed

Visualisation



Iterators implement various traversal algorithms. Several iterator objects can traverse the same collection at the same time.

Discussions

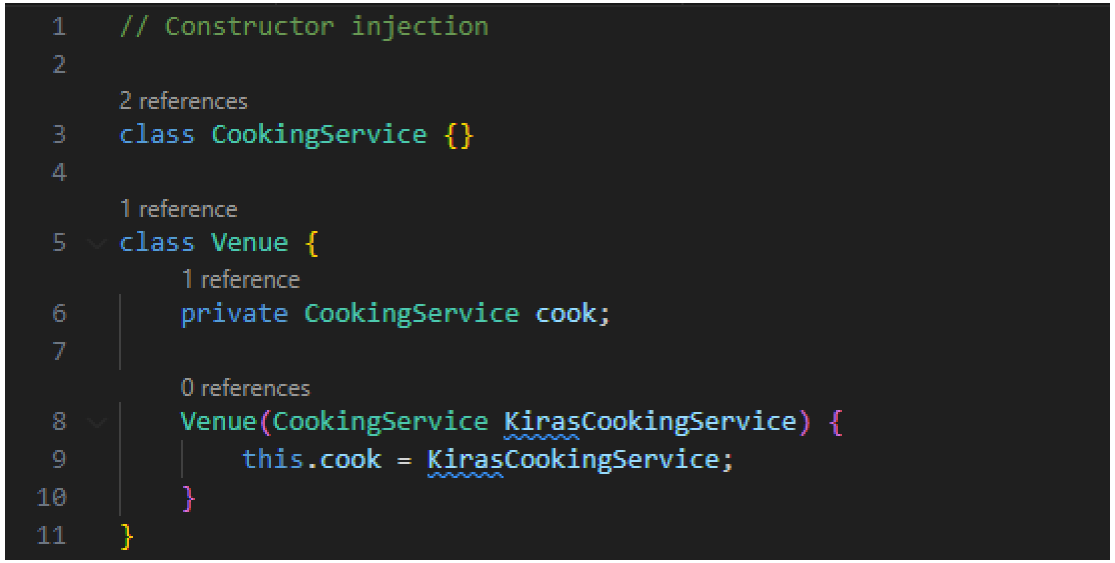
- Design patterns are not intended to force a way to implement something
- They are merely commonly used patterns to address some problem
- It will start ` ` clicking’’ when you look at a snippet of code and get that feeling of “oh, I have seen this before”

Example: Dependency Injection

- Dependency injection: pass the classes that your class depends on as interfaces rather than creating a separate instance of them
- Removes dependencies on other classes

- Example: Venue hosting for food
- Commonly used in application

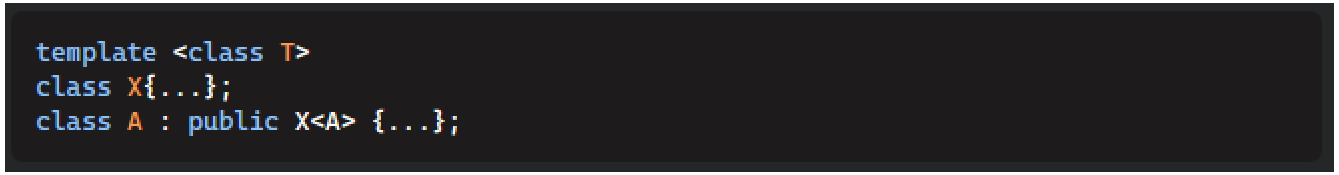
  or web development



```
// Constructor injection

2 references
class CookingService {}

1 reference
class Venue {
    1 reference
    private CookingService cook;

    0 references
    Venue(CookingService KirasCookingService) {
        this.cook = KirasCookingService;
    }
}
```

Example: Curiously reoccurring template pattern (CRTP)

- Where a class has a base class which is a template specialization for the class itself
- More generally known as “F-bound polymorphism”
- Allows for static polymorphism (decides which method to execute during compile time)
- Also gives the template class the ability to be a base class for its specialisations

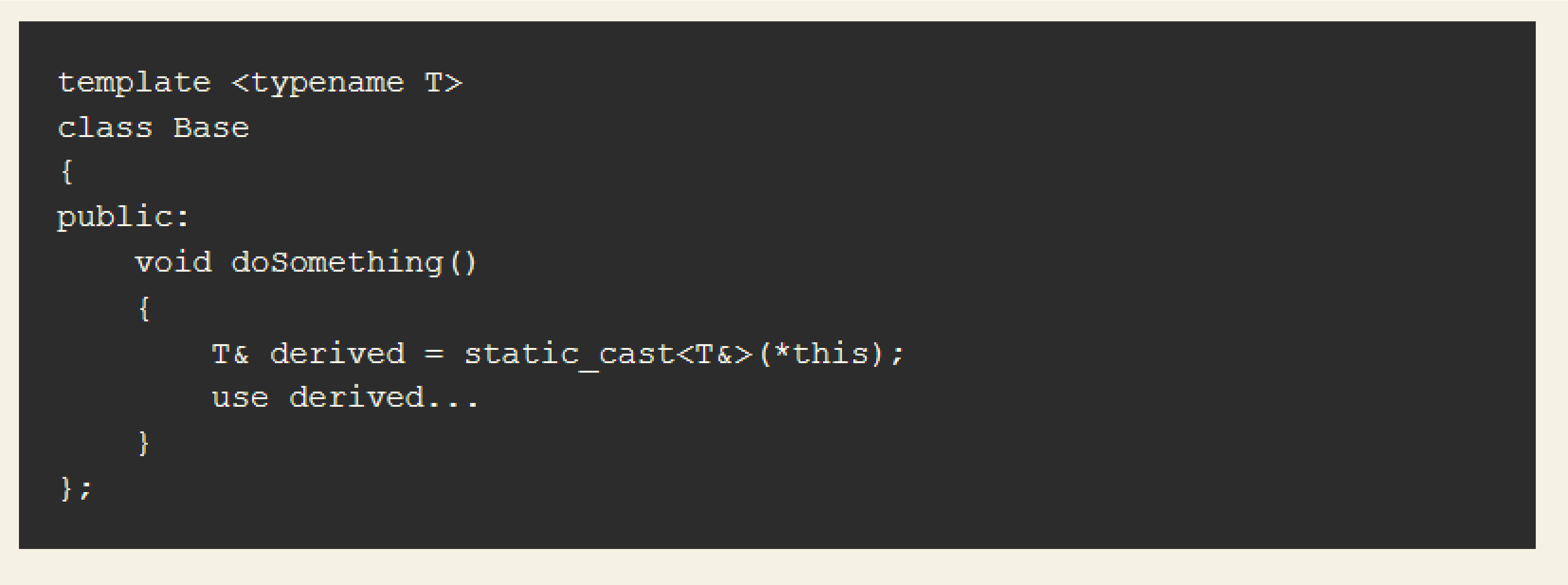


```
template <class T>
class X{...};
class A : public X<A> {...};
```

- https://stackoverflow.com/questions/4173254/what-is-the-curiously-recurring-template-pattern-crtp
- https://www.fluentcpp.com/2017/05/12/curiously-recurring-template-pattern/

CRTP

- From the perspective of the base object, the derived object is itself, but downcasted
- Therefore, the base class can access the derived class by static_casting itself into the derived class

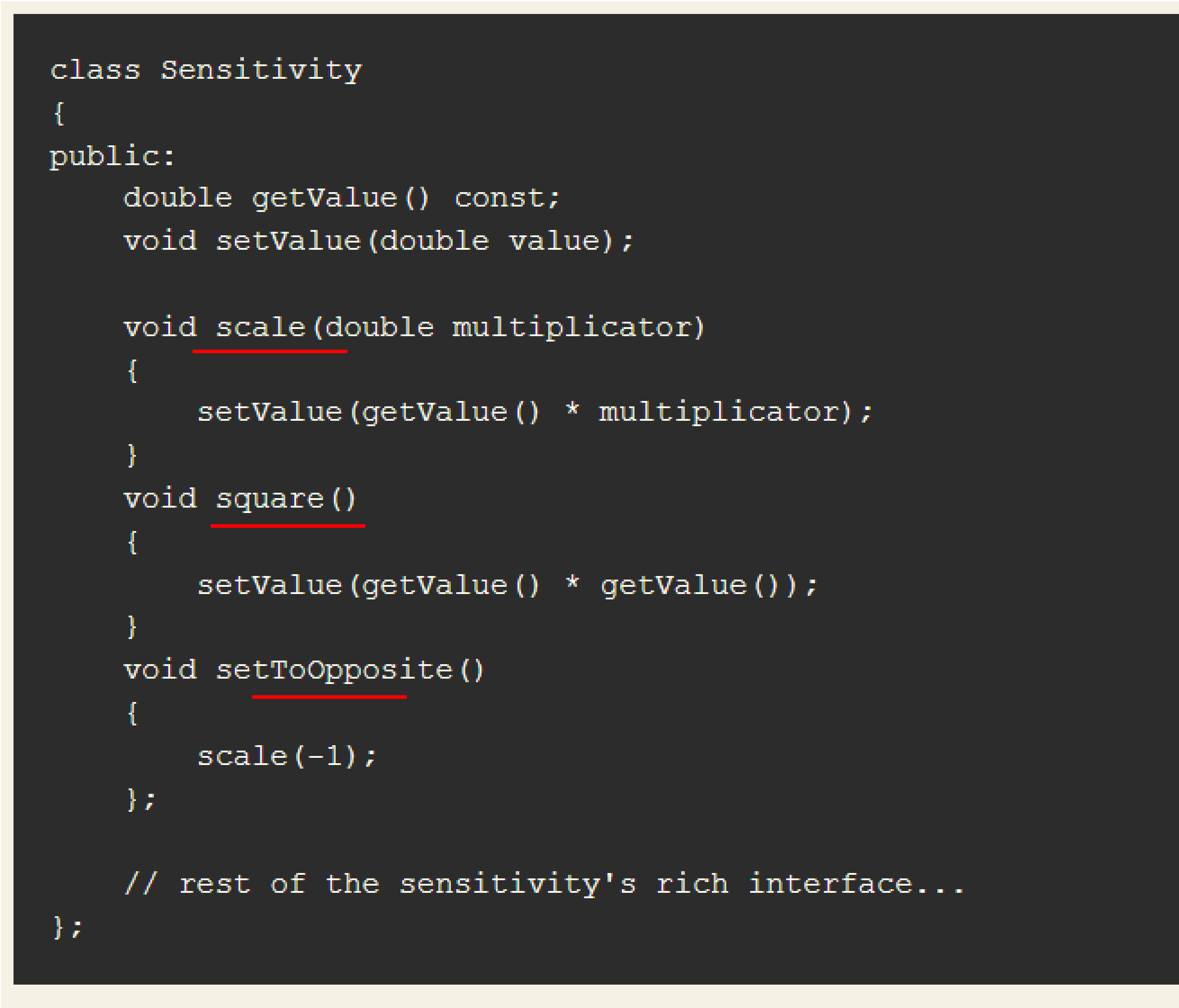


```
template <typename T>
class Base
{
public:
    void doSomething()
    {
        T& derived = static_cast<T&>(*this);
        use derived...
    }
};
```

- Example on these slides are from: https://www.fluentcpp.com/2017/05/12/curiously-recurring-template-pattern/

Usefulness

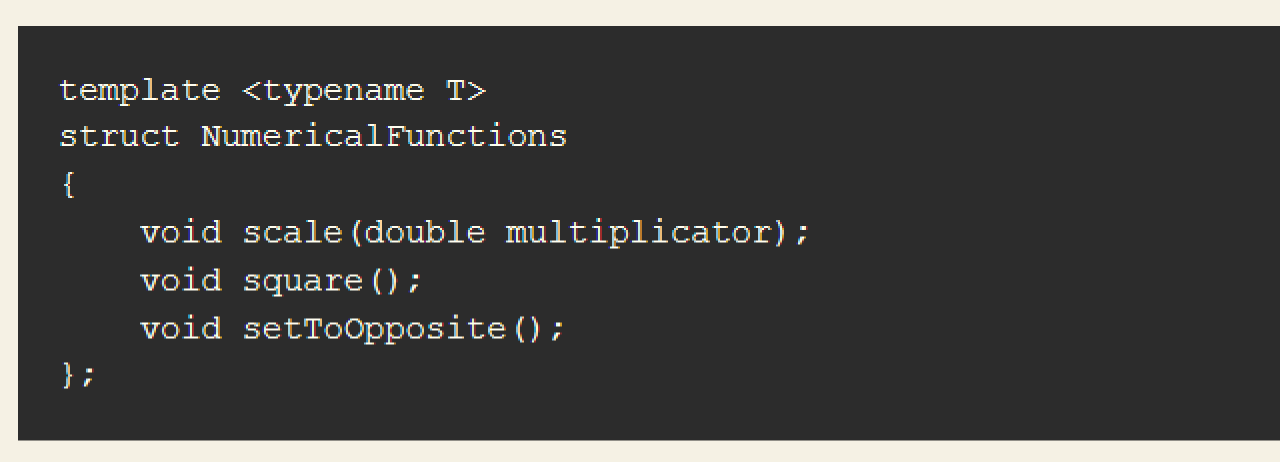
- Here’s a class that has an attribute *value* and 3 different functions
  - Scale
  - Square
  - SetToOpposite
- Supposed I have another class with a *value* that want the same functions
  - Should we just copy over the functions?



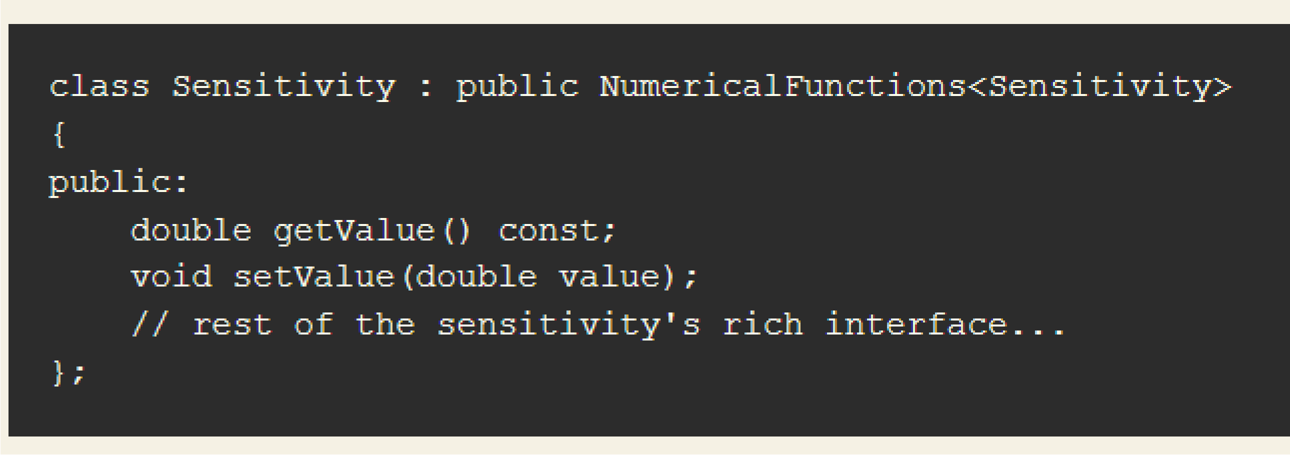
```
class Sensitivity
{
public:
    double getValue() const;
    void setValue(double value);

    void scale(double multiplicator)
    {
        setValue(getValue() * multiplicator);
    }
    void square()
    {
        setValue(getValue() * getValue());
    }
    void setToOpposite()
    {
        scale(-1);
    };

    // rest of the sensitivity's rich interface...
};
```

CRTP approach

- Pull out the functions into a separate Base class
- Have the Derived class inherent from it
- Now other classes can take the same approach!
- And we can add more functionality generically

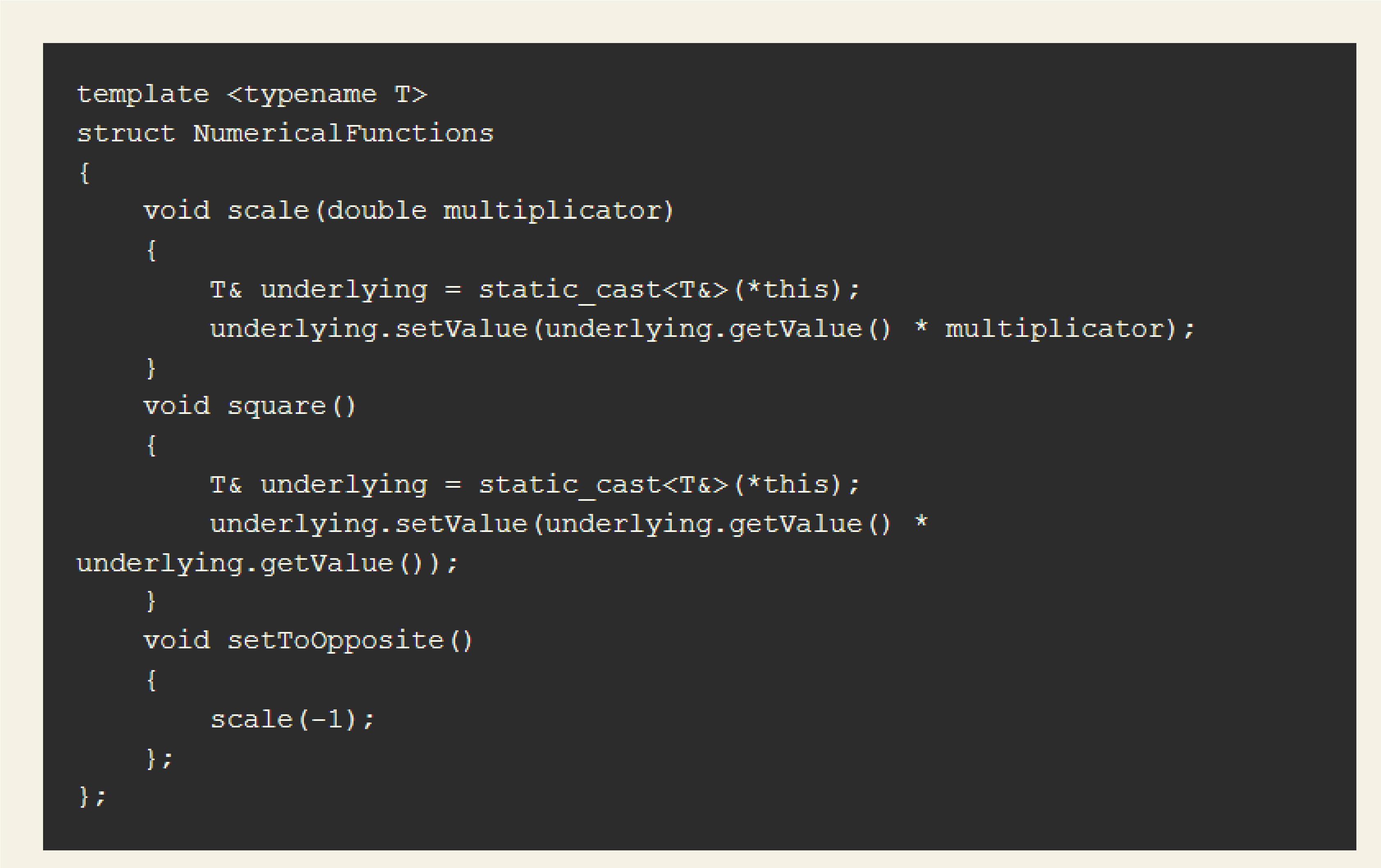


```cpp
template <typename T>
struct NumericalFunctions
{
    void scale(double multiplicator);
    void square();
    void setToOpposite();
};
```



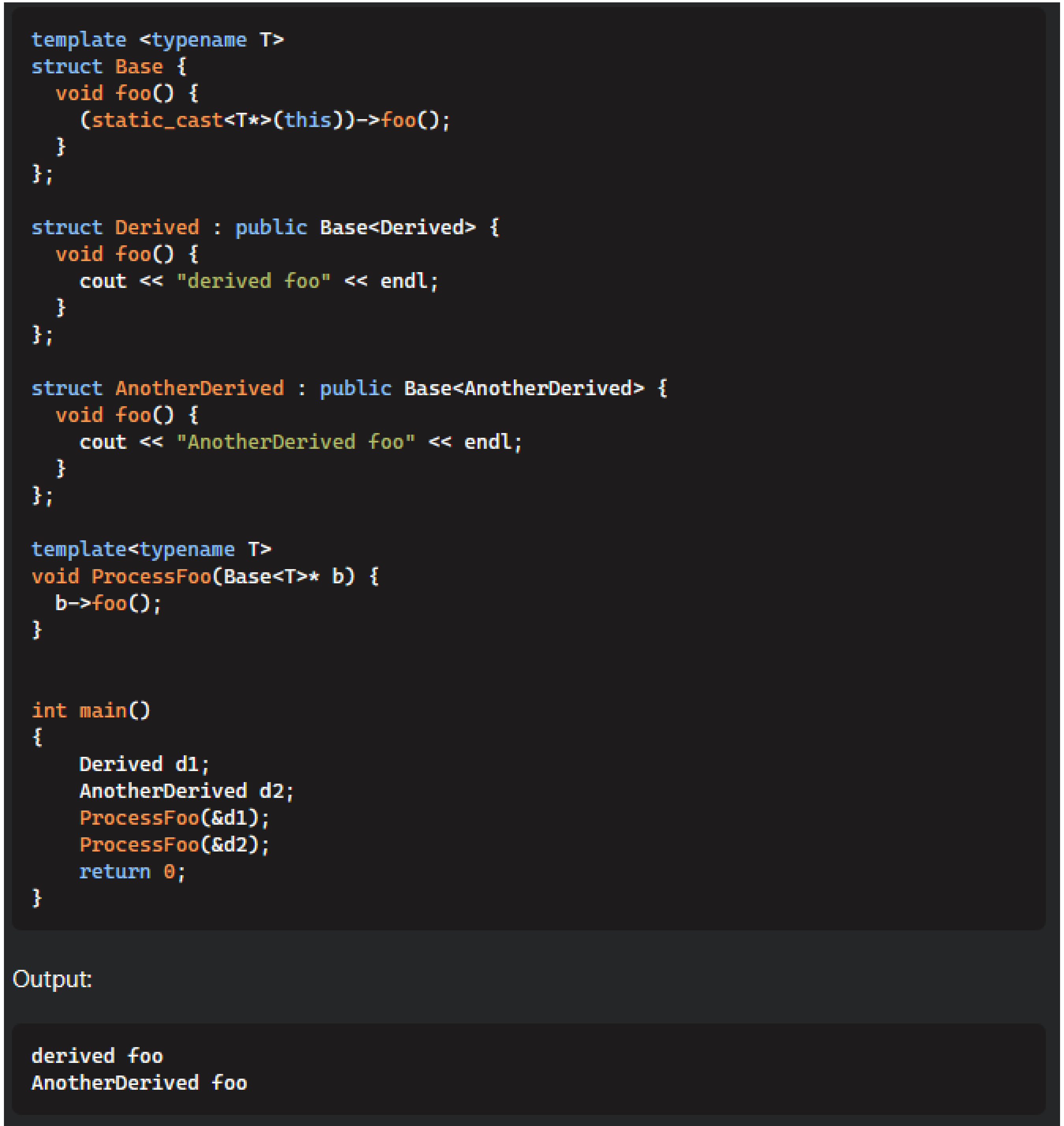
```cpp
class Sensitivity : public NumericalFunctions<Sensitivity>
{
public:
    double getValue() const;
    void setValue(double value);
    // rest of the sensitivity's rich interface...
};
```

Implementation of the Base class



```
template <typename T>
struct NumericalFunctions
{
    void scale(double multiplicator)
    {
        T& underlying = static_cast<T&>(*this);
        underlying.setValue(underlying.getValue() * multiplicator);
    }
    void square()
    {
        T& underlying = static_cast<T&>(*this);
        underlying.setValue(underlying.getValue() *
underlying.getValue());
    }
    void setToOpposite()
    {
        scale(-1);
    };
};
```

Example use case



```
template <typename T>
struct Base {
  void foo() {
    (static_cast<T*>(this))->foo();
  }
};

struct Derived : public Base<Derived> {
  void foo() {
    cout << "derived foo" << endl;
  }
};

struct AnotherDerived : public Base<AnotherDerived> {
  void foo() {
    cout << "AnotherDerived foo" << endl;
  }
};

template<typename T>
void ProcessFoo(Base<T>* b) {
  b->foo();
}


int main()
{
    Derived d1;
    AnotherDerived d2;
    ProcessFoo(&d1);
    ProcessFoo(&d2);
    return 0;
}
```

Output:

```
derived foo
AnotherDerived foo
```

Patterns become standardised eventually if used enough

- C++23 introduces “deducing this” that allows you to access the derived class from the base class
- Iterators are basically the standard way to access items now

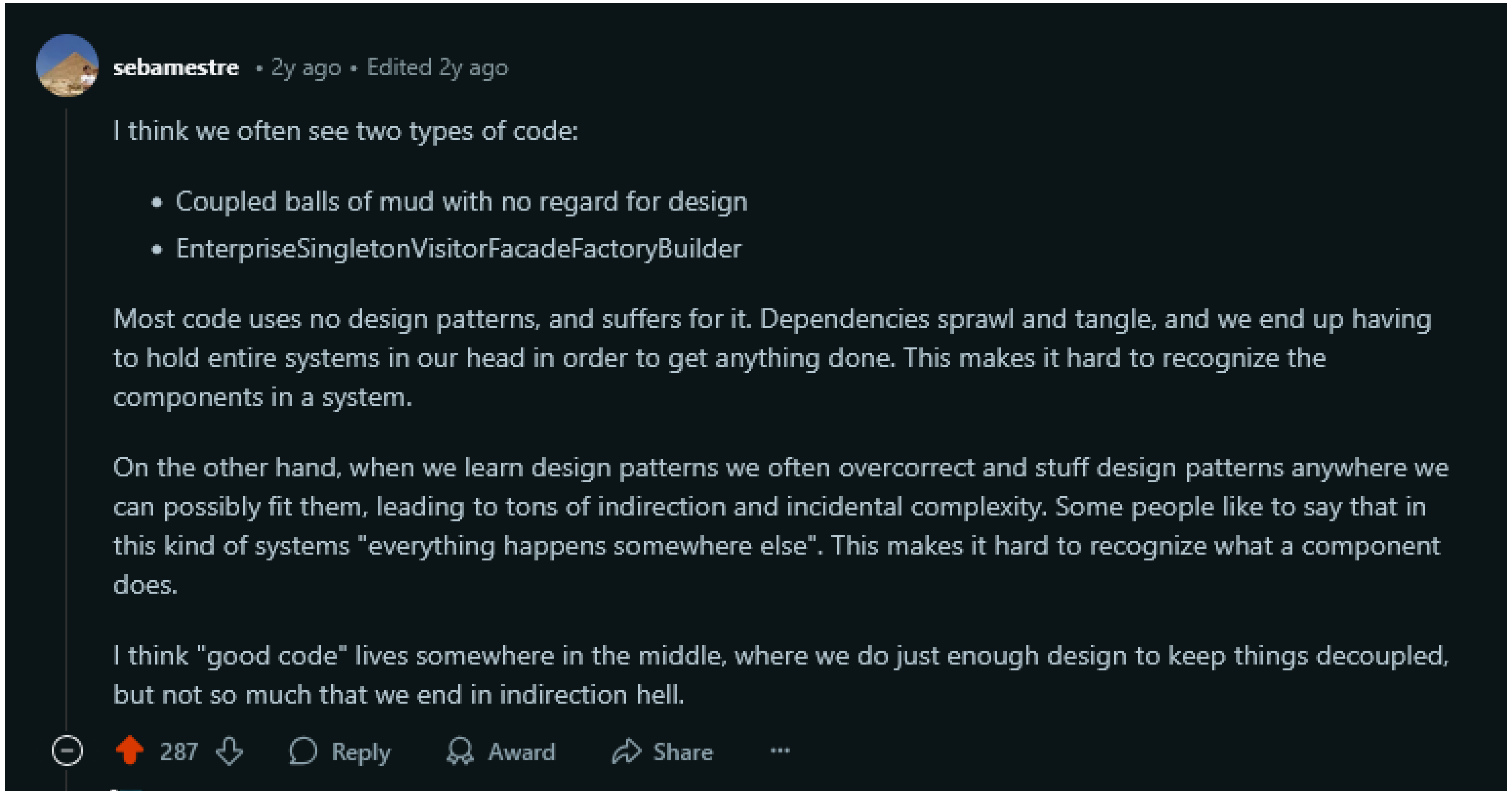
Criticism

- Can lead to inefficient solutions
- Introduces complexity
- Lead to anti-patterns
    - Commonly-used process or pattern that has more consequences than good effects

Summary

- They really allow for developers to talk about a problem and prevent re-inventing a (poorly designed) wheel
- Typically, design patterns can emerge from good coding without explicitly trying to incorporate them
- If you are forcing code to fit based on a design pattern, you are probably doing it wrong
  - They should occur naturally
- The C++ standard library actually uses and incorporates them profusely

Perfect discussion post found on reddit



sebamestre • 2y ago • Edited 2y ago

I think we often see two types of code:

- Coupled balls of mud with no regard for design
- EnterpriseSingletonVisitorFacadeFactoryBuilder

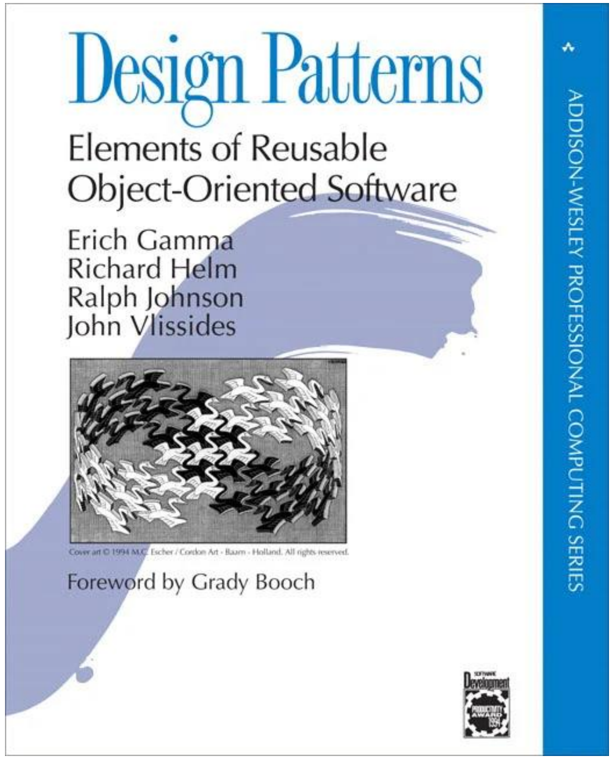
Most code uses no design patterns, and suffers for it. Dependencies sprawl and tangle, and we end up having to hold entire systems in our head in order to get anything done. This makes it hard to recognize the components in a system.

On the other hand, when we learn design patterns we often overcorrect and stuff design patterns anywhere we can possibly fit them, leading to tons of indirection and incidental complexity. Some people like to say that in this kind of systems "everything happens somewhere else". This makes it hard to recognize what a component does.

I think "good code" lives somewhere in the middle, where we do just enough design to keep things decoupled, but not so much that we end in indirection hell.

Persons of the day
Group of four

- Commonly referred to as the Gang of Four (GOF)
  - The authors do not like that name
- Gamma, Helm, Johnson, Vlissi
- Known for their software engineering book on
  - Includes examples in C++



Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES