

Compiler Case Study - Design Patterns in C++

Ganesh Samarthyam
ganesh@codeops.tech

Design Patterns - Introduction

“Applying design principles is the key to creating high-quality software!”



Architectural principles:
Axis, symmetry, rhythm, datum, hierarchy, transformation

How to Apply Principles in Practice?

Design principles

How to bridge
the gap?

Code

What are Smells?

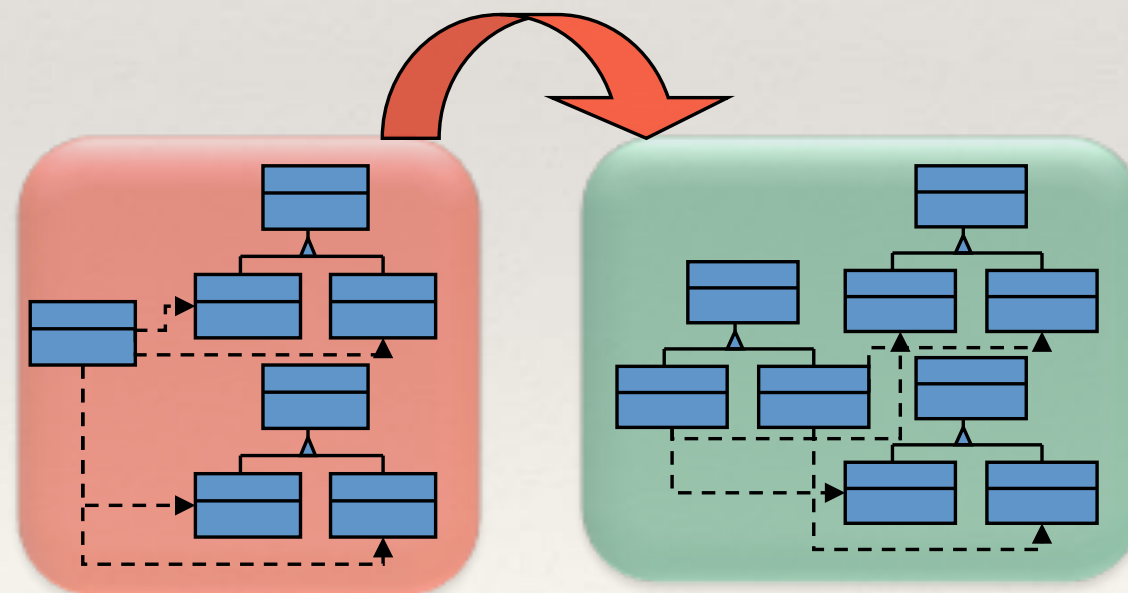
“Smells are certain structures in the code that suggest (sometimes they scream for) the possibility of refactoring.”



What is Refactoring?

Refactoring (noun): a *change* made to the *internal structure* of software to make it *easier to understand and cheaper to modify without changing its observable behavior*

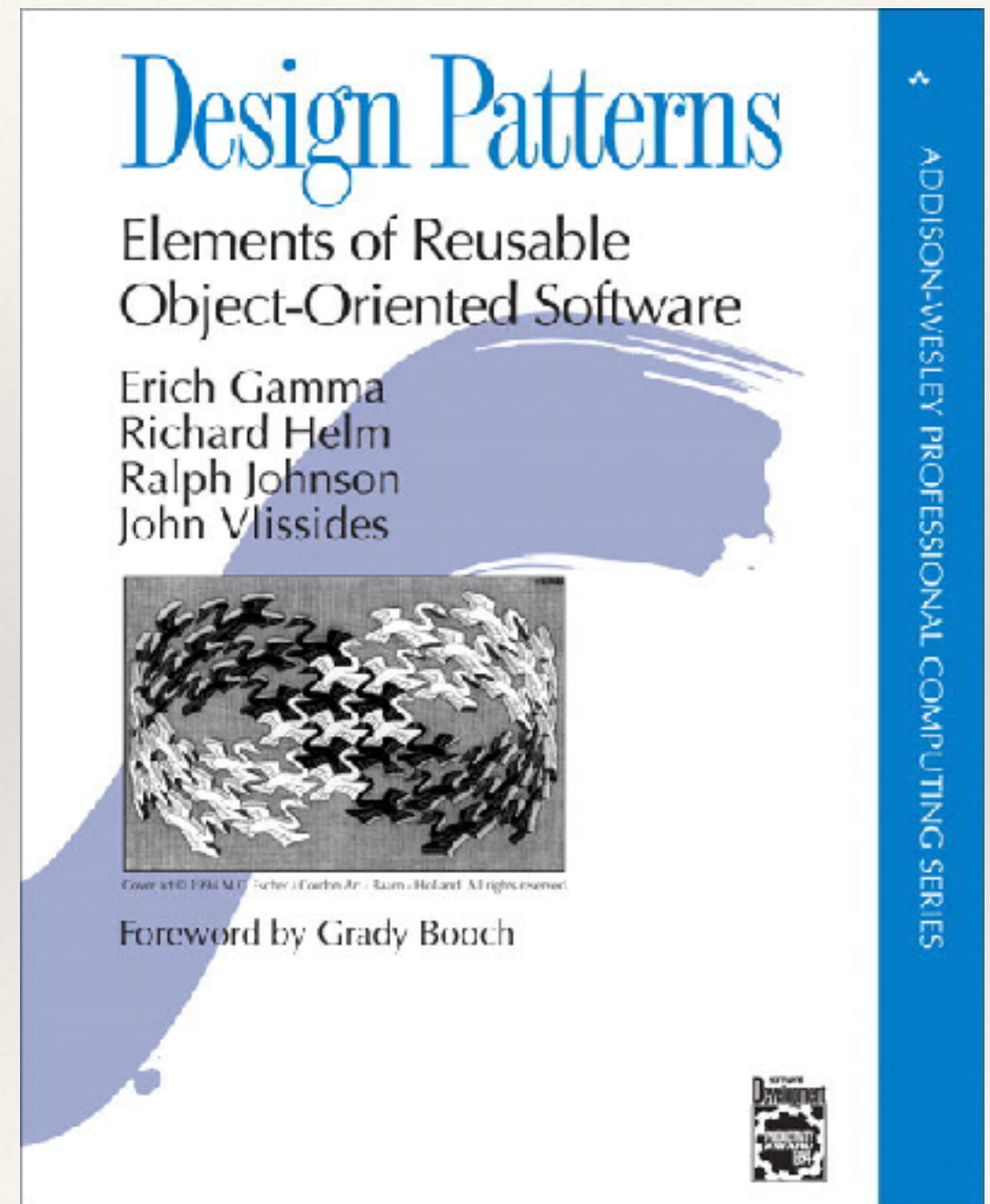
Refactor (verb): to restructure software by applying a series of refactorings *without* changing its observable behavior



What are Design Patterns?

*recurrent solutions
to common
design problems*

Pattern Name
Problem
Solution
Consequences



Principles to Patterns

```
Expr* expr = new Expr(  
    new Expr(  
        new Expr(nullptr, "10", nullptr), "*", new Expr(nullptr, "20",  
nullptr)),  
    "+",  
    new Expr(nullptr, "30", nullptr));  
expr->GenCode();
```



Encapsulate object creation

```
Expr expr = new ExprBuilder()->addConstant(10)  
    ->withMultiplyNode(20)  
    ->withPlusNode(30)  
    ->toExpressionTree();
```

Builder pattern

Fluent interface pattern

Why Care About Patterns?

- ❖ Patterns capture expert knowledge in the form of proven reusable solutions
 - ❖ Better to reuse proven solutions than to “re-invent” the wheel
- ❖ When used correctly, patterns positively influence software quality
 - ❖ Creates maintainable, extensible, and reusable code

**GOOD
DESIGN
IS GOOD
BUSINESS**

-THOMAS J WATSON JR.

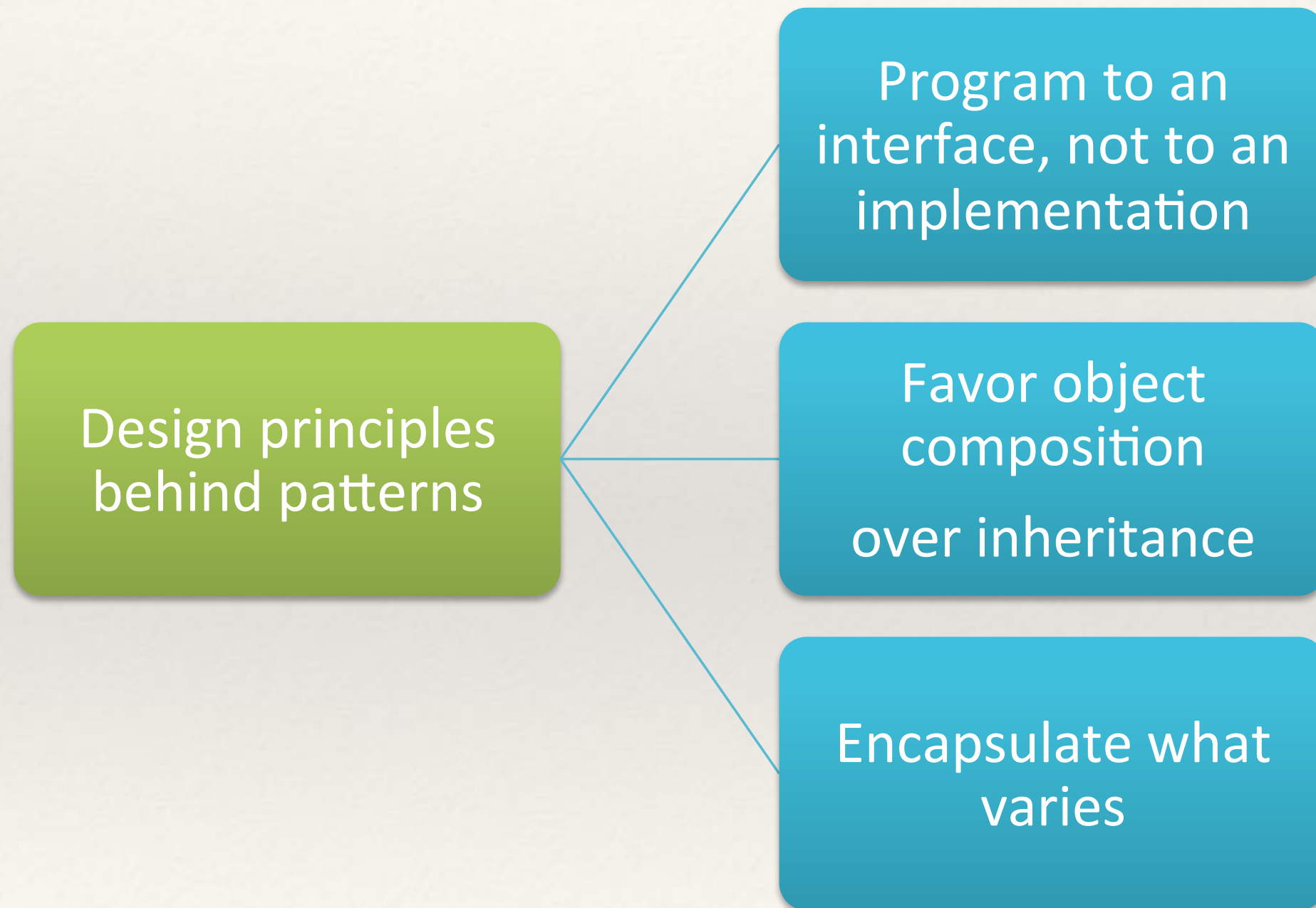
Design Patterns - Catalog

		<i>Purpose</i>		
		Creational	Structural	Behavioral
<i>Scope</i>	Class	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Design Patterns - Catalog

Creational	Deals with controlled object creation	<i>Factory method</i> , for example
Structural	Deals with composition of classes or objects	<i>Composite</i> , for example
Behavioral	Deals with interaction between objects / classes and distribution of responsibility	<i>Strategy</i> , for example

3 Principles Behind Patterns



Compiler Case Study

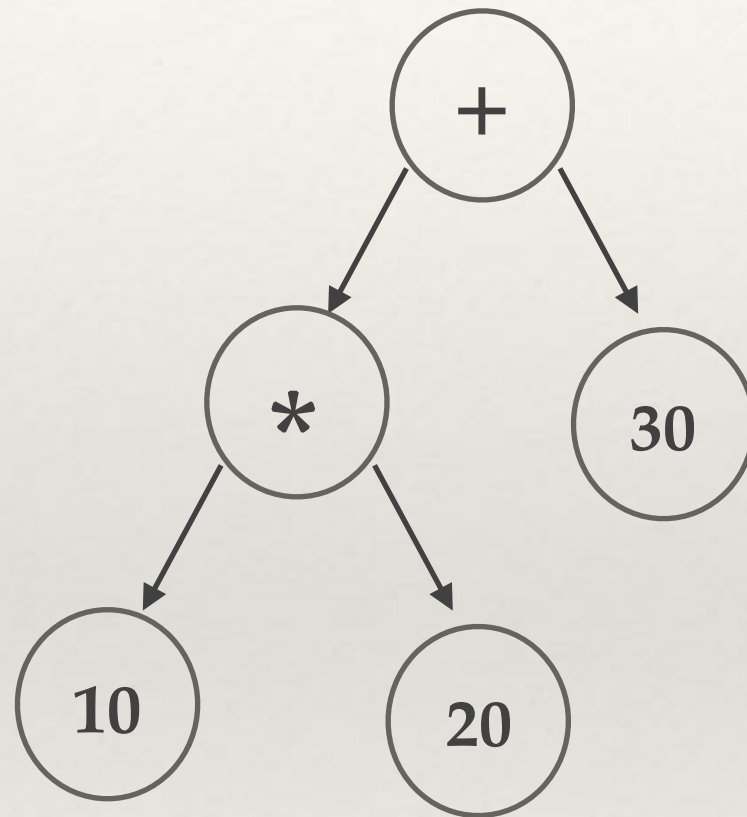
Case Study Overview

- ❖ Smelly code segments (specially if-else based on types and switch statements) used first
 - ❖ They are “refactored” using design patterns
- ❖ The case study shows code-snippets; compilable, self-contained source code files shared with you

What You Need to Know

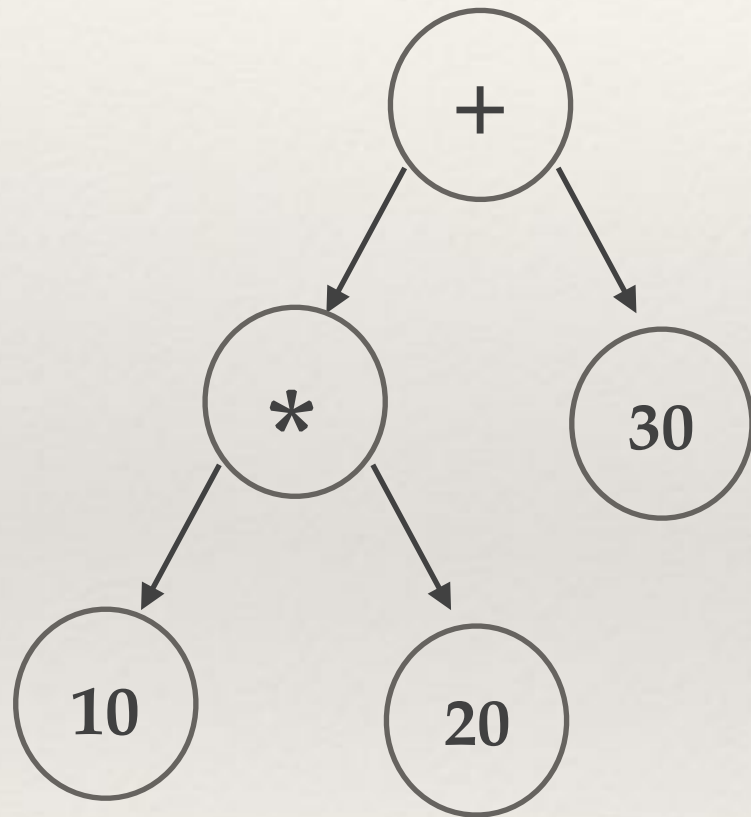
- ❖ Understand essential object oriented constructs such as runtime polymorphism (virtual functions), and principles (such as encapsulation)
- ❖ Know essential C++ language features - classes, inheritance, etc
- ❖ Data structures and algorithms (especially binary trees, tree traversals, and stacks)
- ❖ NO background in compilers needed
 - ❖ Extremely simplified compiler code example; purposefully done to make best use of this case study

Simple Expression Tree Example



Expression: $((10 * 20) + 30)$

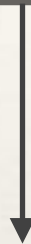
Post-order Traversal: Simulating Code Gen



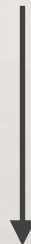
post-order
traversal
result

10 20 * 30 +

Front-end



Intermediate code (CIL, Java bytecodes, ...)



Interpreter (Virtual
Machine)

Using Stack for Evaluation

10 20 * 30 +



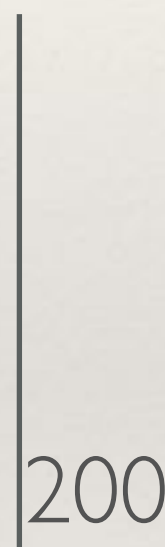
Initial
empty



push 10



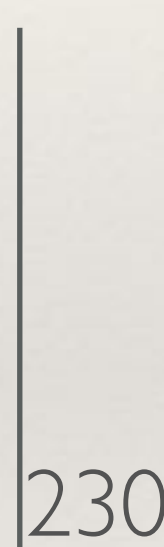
push 20



perform
 $10 * 20$
push
200



push 30



perform
 $200 + 30$
push
230

End
result
 $= 230$

Mnemonic Names in Java and .NET

10 20 * 30 +



push 10
push 20
multiply
push 30
add



.NET CIL Instructions

```
ldc.i4.s 0x0a  
ldc.i4.s 0x14  
mul  
ldc.i4.s 0x1e  
add
```

(ldc => load constant ;
i4 => integer of 32 bits;
s => signed)

Java Bytecode Instructions

```
bipush 10  
bipush 20  
imul  
bipush 30  
iadd
```

(bipush stands for “byte integer push”)

Hands-on Exercise

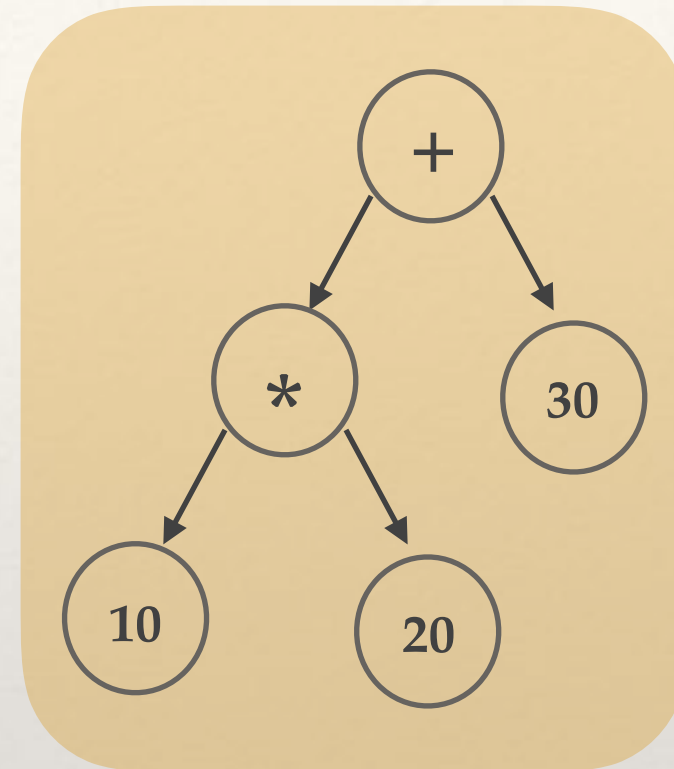
```
// Expr.cs
using System;

class Test {
    public static void Main() {
        int a = 10, b = 20, c = 30;
        int r = ((a * b) + c);
        Console.WriteLine(r);
    }
}
```

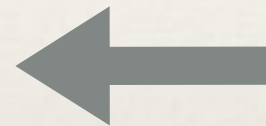
Use the ildasm/javap tool to disassemble the Expr.cs/
Expr.java code

Source Code

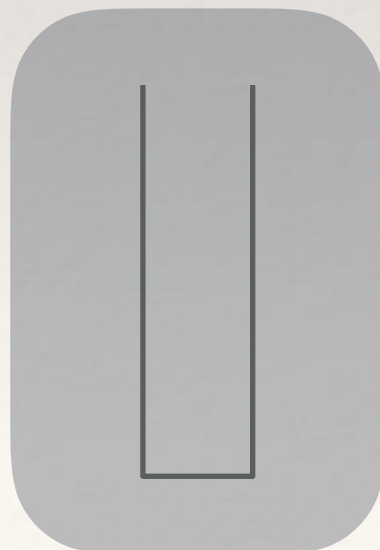
$((10 * 20) + 30)$



**C#
Compiler**



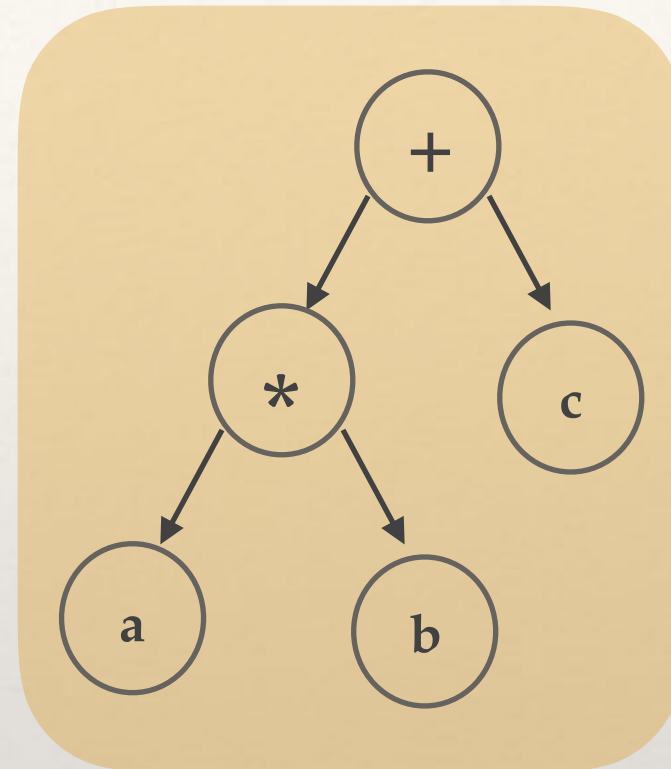
`ldc.i4.s 0x0a
ldc.i4.s 0x14
mul
ldc.i4.s 0x1e
add`



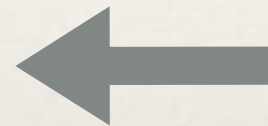
**.NET CLR
(Common
Language
Runtime)**

Source Code

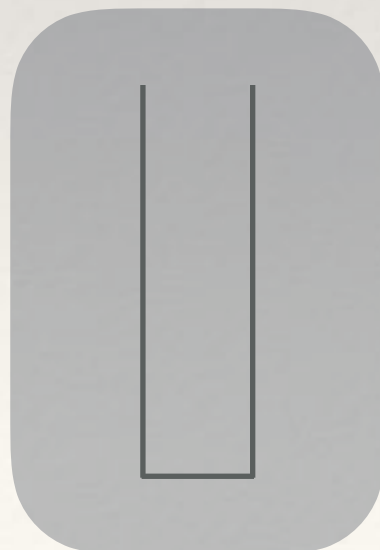
$((a * b) + c)$



**Java
Compiler**

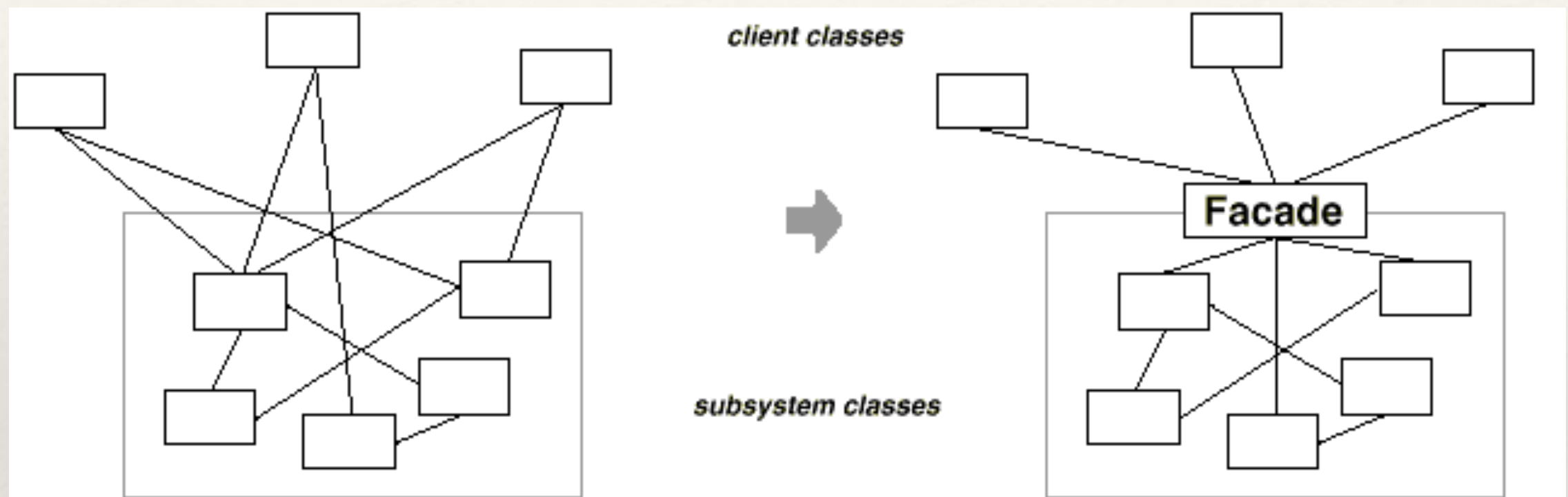


```
9: iload_1  
10: iload_2  
11: imul  
12: iload_3  
13: iadd
```



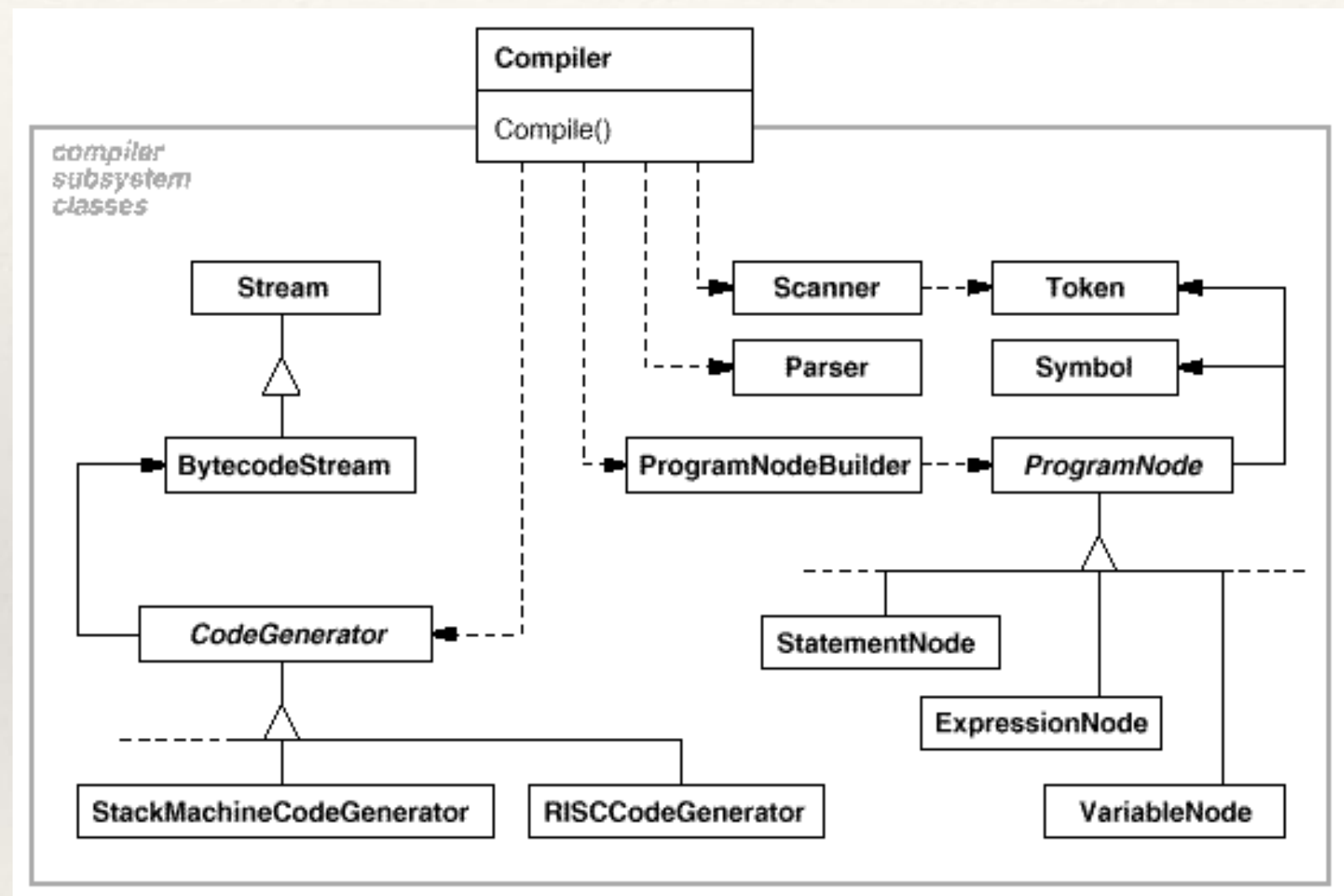
**JVM
(Java
Virtual
Machine)**

Facade Pattern



Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Facade Pattern in Compilers



“The Compiler class acts as a facade: It offers clients a single, simple interface to the compiler subsystem.”

Procedural (conditional) Code

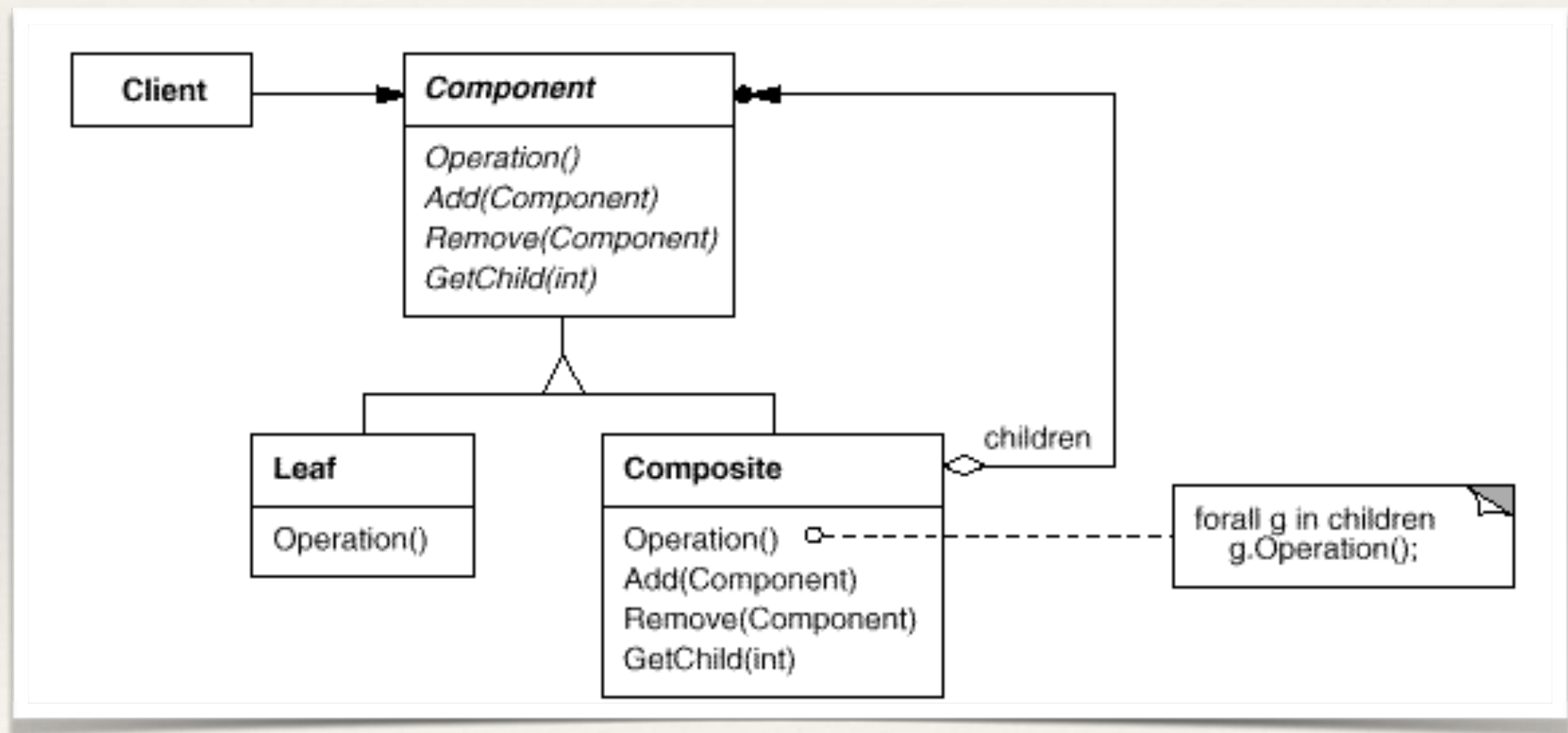
```
void GenCode() {  
    if((_left == nullptr) && (_right == nullptr)) {  
        cout<< "bipush " << _value << endl;  
    }  
    else { // its an intermediate node  
        _left->GenCode();  
        _right->GenCode();  
        if(_value == "+") {  
            cout << "add" << endl;  
        }  
        else if(_value == "-") {  
            cout << "sub" << endl;  
        }  
        else if(_value == "*") {  
            cout << "mul" << endl;  
        }  
        else if(_value == "/") {  
            cout << "div" << endl;  
        }  
        else {  
            cout << "Internal Error: Not implemented yet!";  
        }  
    }  
}
```

How to get rid of
conditional statements?

Hands-on Exercise

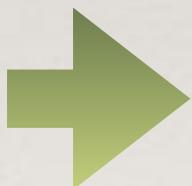
Refactor the code in “compiler.cpp” to get rid of conditional statements and use runtime polymorphism instead

Composite Pattern



Composite Pattern: Discussion

Compose objects into tree structures to represent part-whole hierarchies. Composite lets client treat individual objects and compositions of objects uniformly.

- ❖ There are many situations where a group of components form larger components
 - ❖ Simplistic approach: Make container component contain primitive ones
 - ❖ Problem: Code has to treat container and primitive components differently
- 
- ❖ Perform recursive composition of components
 - ❖ Clients don't have to treat container and primitive components differently

Hands-on Exercise

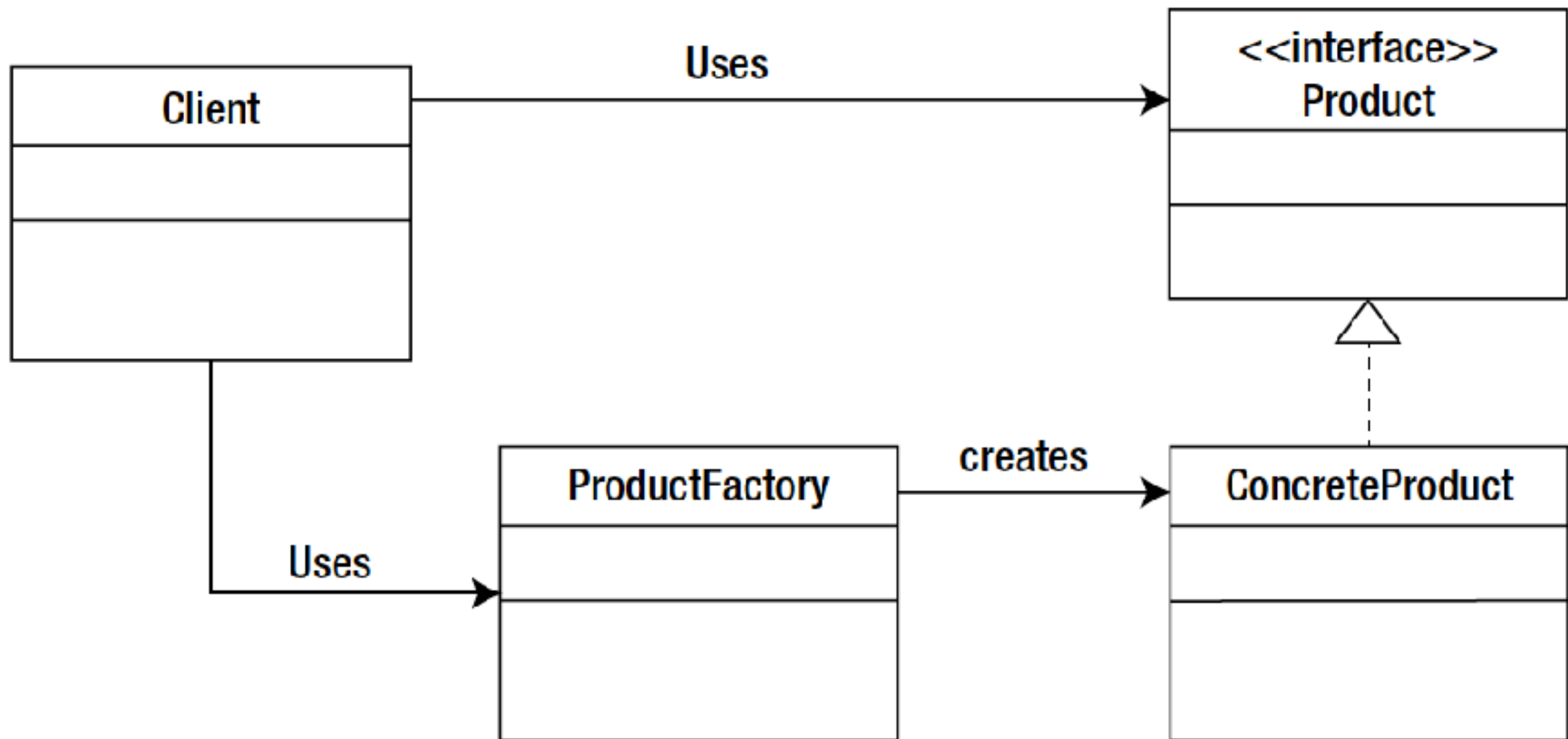
Rewrite the code to “evaluate” the expression instead of generating the code - here is the “pseudo-code”

```
function interpret(node: binary expression tree)
| if node is a leaf then
    return (node.value)
| if node is binary operator op then
    return interpret(node.left) op interpret(node.right)
```

How to improve the tree creation?

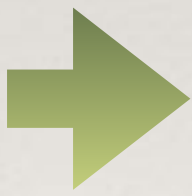
```
Expr* expr = new Expr(  
    new Expr(  
        new Expr(nullptr, "10", nullptr), "*", new Expr(nullptr, "20",  
nullptr)),  
    "+",  
    new Expr(nullptr, "30", nullptr));  
expr->GenCode();
```

Factory Method Pattern: Structure



Factory Method Pattern: Discussion

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory method lets a class defer instantiation to subclasses.

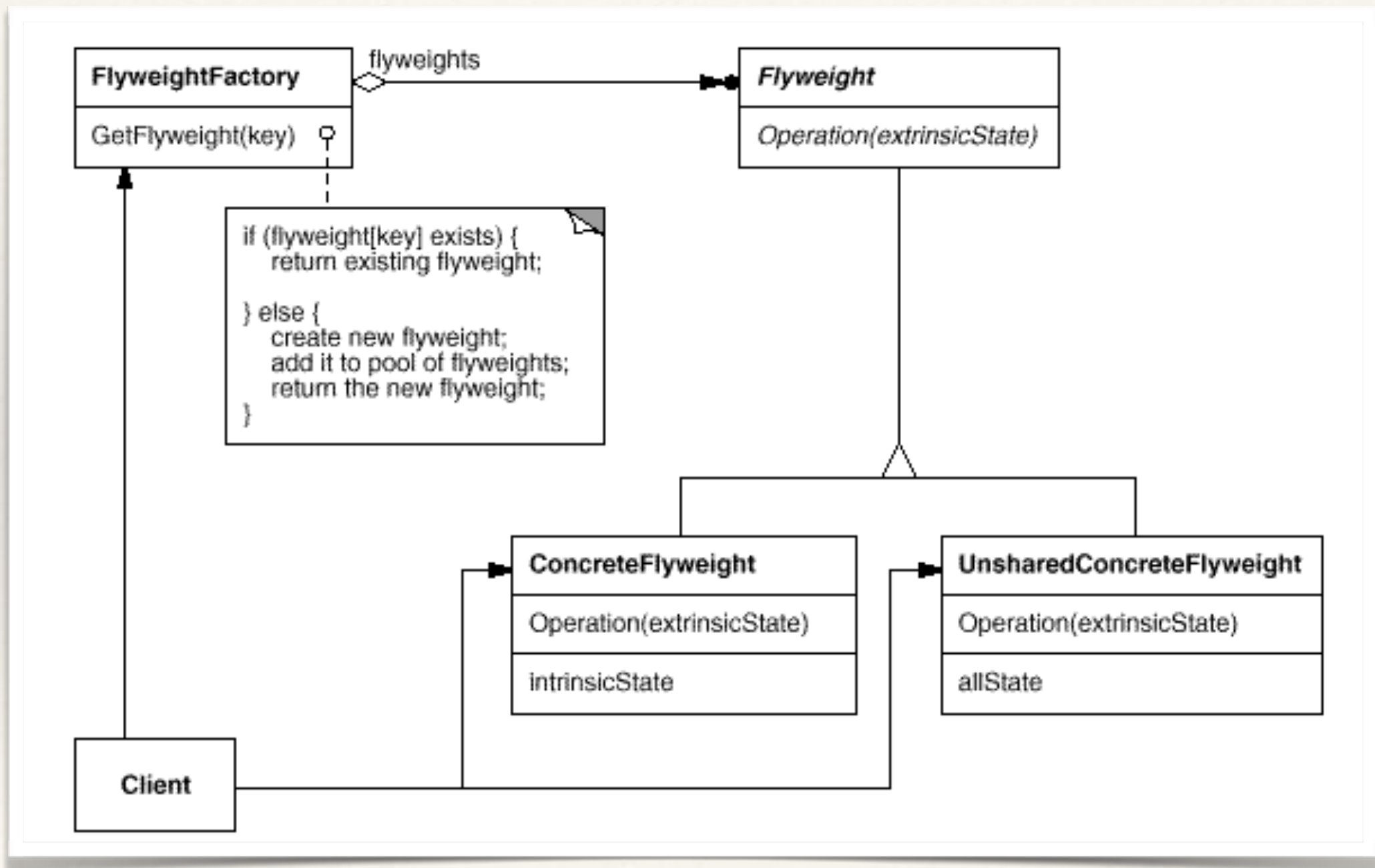
- ❖ A class cannot anticipate the class of objects it must create
 - ❖ A class wants its subclasses to specify the objects it creates
- 
- ❖ Delegate the responsibility to one of the several helper subclasses
 - ❖ Also, localize the knowledge of which subclass is the delegate

Hands-on Exercise

```
public:  
static Constant* Make(int arg)  
{  
    return new Constant(arg);  
}
```

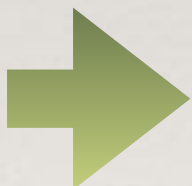
Improve this code to reuse immutable objects (for the ones with the same values) instead of creating duplicate objects.

Flyweight Pattern: Structure



Flyweight Pattern: Discussion

Use sharing to support large numbers of fine-grained objects efficiently

- ❖ When an application uses a large number of small objects, it can be expensive
 - ❖ How to share objects at granular level without prohibitive cost?
- 
- ❖ When it is possible to share objects (i.e., objects don't depend on identity)
 - ❖ When object's value remain the same irrespective of the contexts - they can be shared
 - ❖ Share the commonly used objects in a pool

How to Improve the Tree Creation?

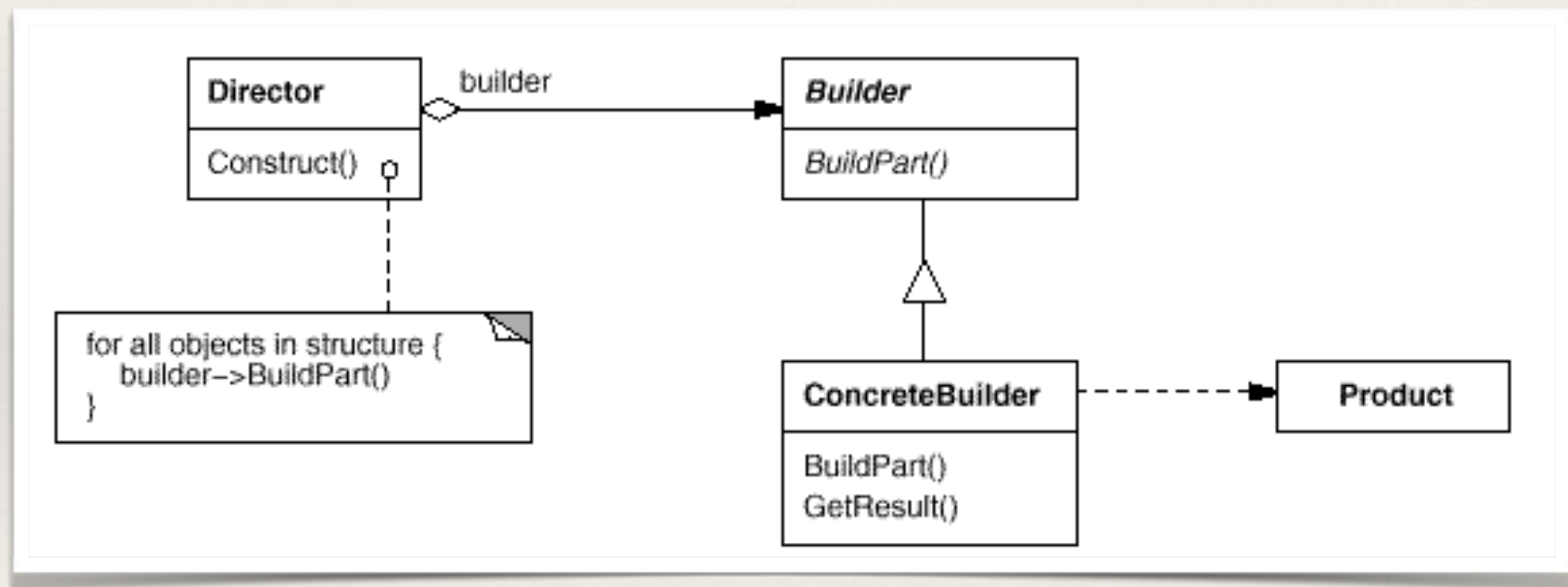
```
Expr* expr = new Expr(  
    new Expr(  
        new Expr(nullptr, "10", nullptr), "*", new Expr(nullptr, "20",  
nullptr)),  
    "+",  
    new Expr(nullptr, "30", nullptr));  
expr->GenCode();
```



$10 * 20 + 30$

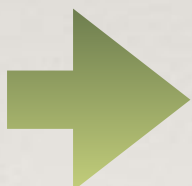
```
Expr expr = new ExprBuilder()->addConstant(10)  
    ->withMultiplyNode(20)  
    ->withPlusNode(30)  
    ->toExpressionTree();
```

Builder Pattern: Structure

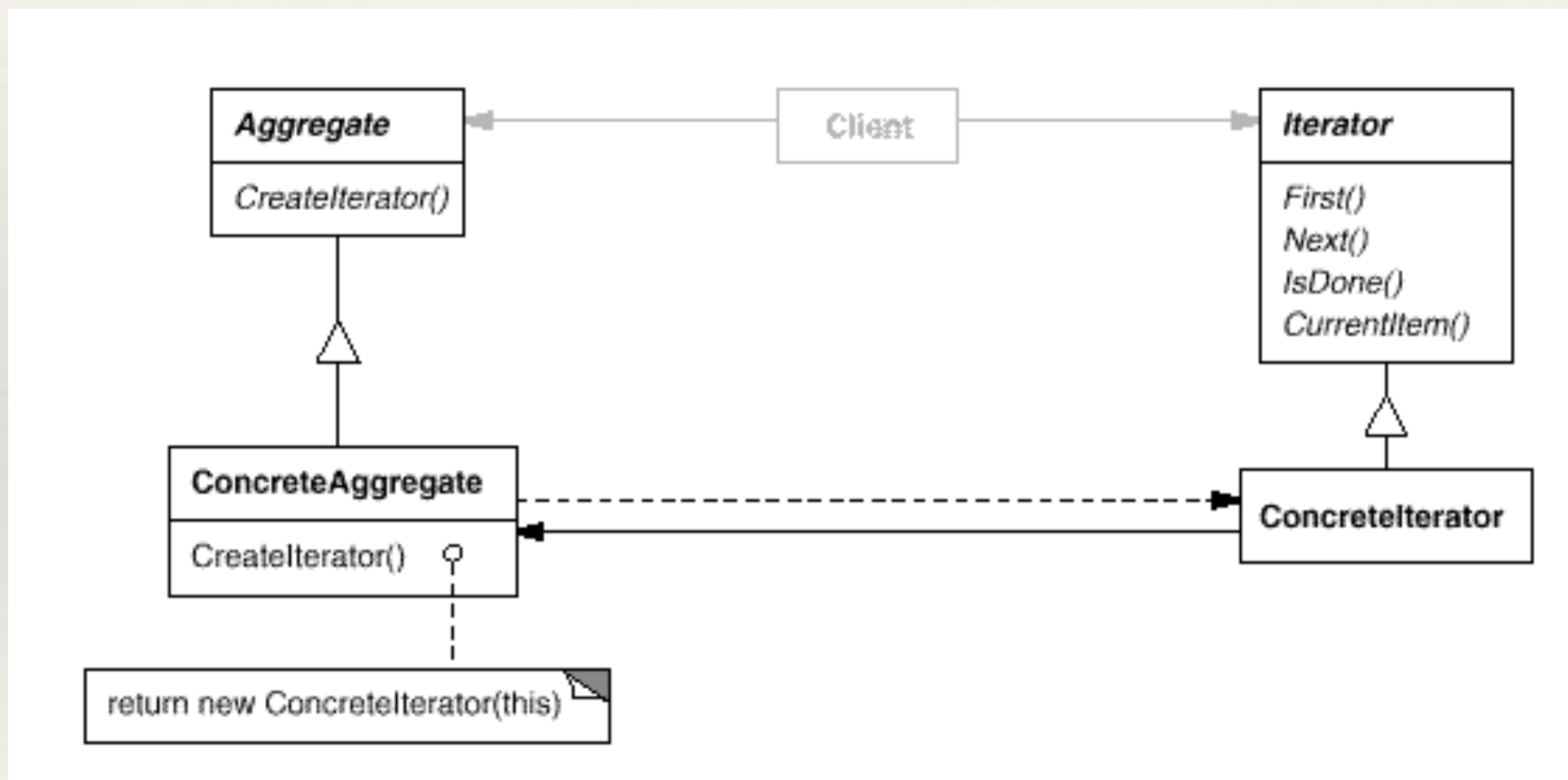


Builder Pattern: Discussion

Separate the construction of a complex object from its representation so that the same construction process can create different representations.

- ❖ Creating or assembling a complex object can be tedious
- 
- ❖ Make the algorithm for creating a complex object independent of parts that make up the object and how they are assembled
 - ❖ The construction process allows different representations for the object that is constructed

Iterator Pattern: Structure



Iterator Pattern: Discussion

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

- ❖ How to provide a way to traverse an aggregate structure in different ways, but without exposing its implementation details?



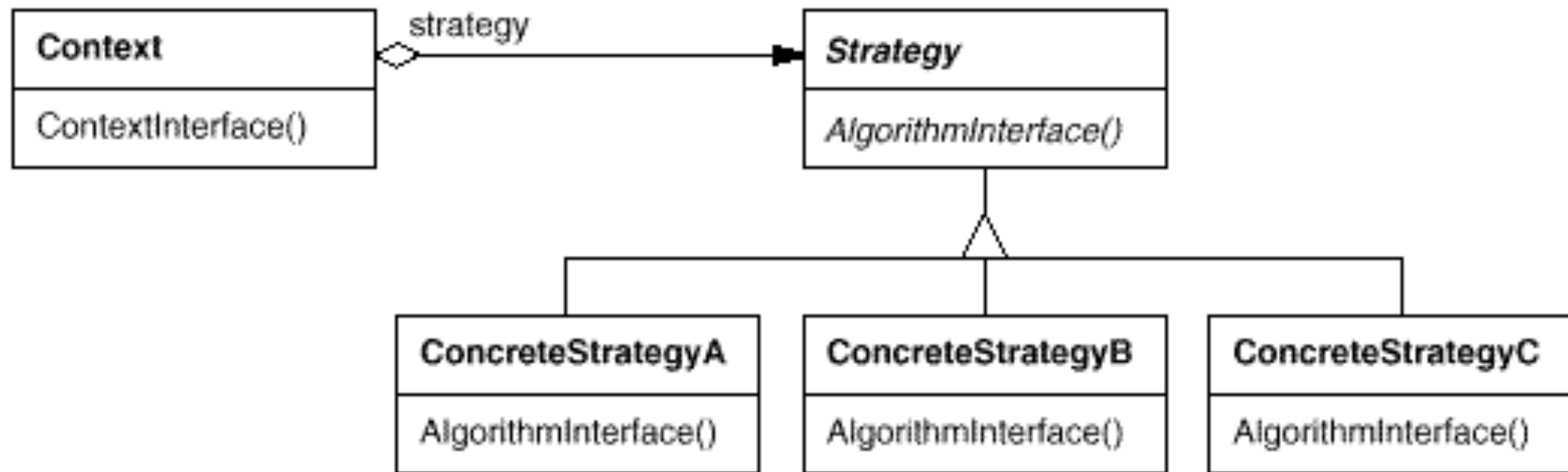
- ❖ Take the responsibility for access and traversal out of the list object and put it into an iterator object
- ❖ Let the iterator keep track of the element visited, traversal order, etc

Hands-on Exercise

```
if(t == Target.JVM) {  
    cout << "iadd" << endl;  
}  
else { // DOTNET  
    cout << "add" << endl;  
}
```

Refactor this code to remove the if-else condition check (explicit type-checking code) and use runtime polymorphism instead

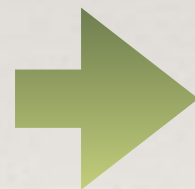
Strategy Pattern: Structure



Strategy Pattern: Discussion

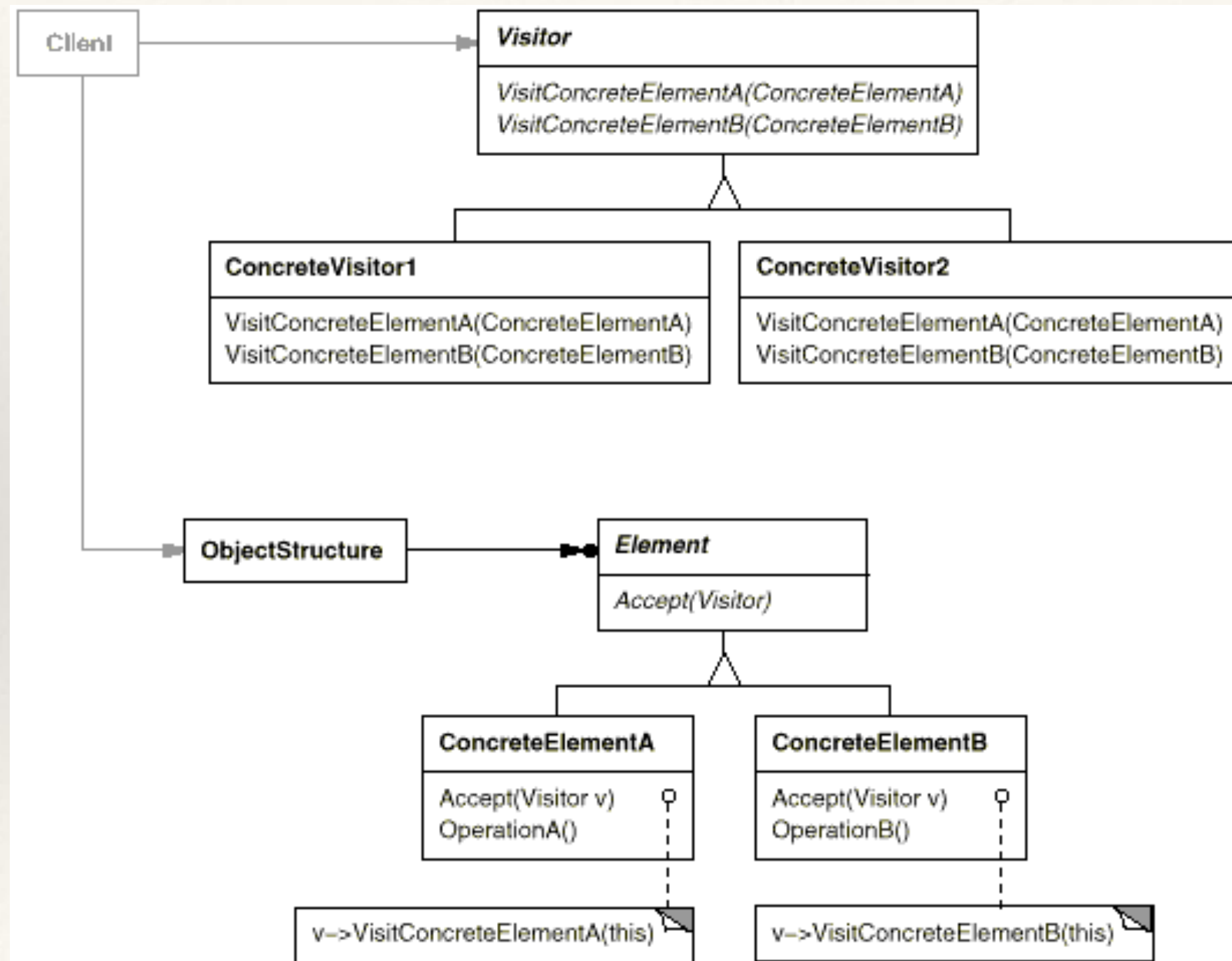
Defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it

- ❖ Useful when there is a set of related algorithms and a client object needs to be able to dynamically pick and choose an algorithm that suits its current need

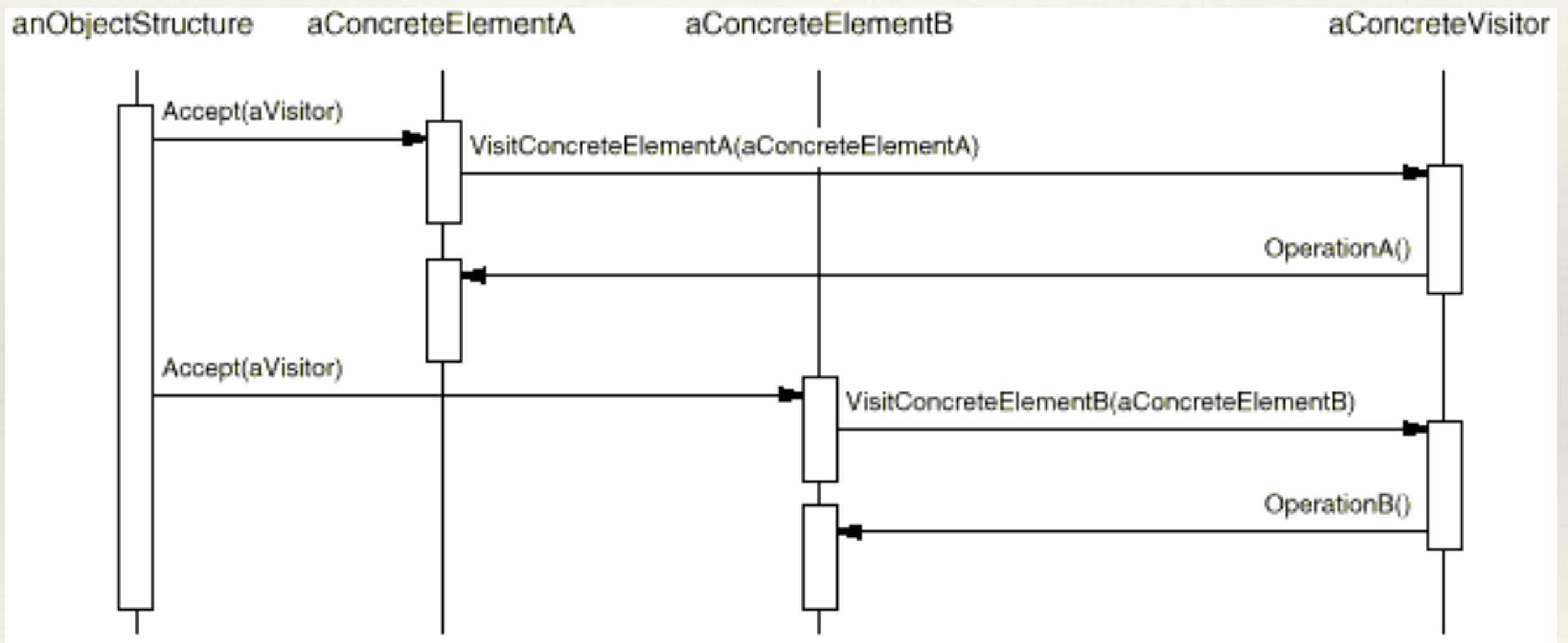


- ❖ The implementation of each of the algorithms is kept in a separate class referred to as a strategy.
- ❖ An object that uses a Strategy object is referred to as a context object.
- ❖ Changing the behavior of a Context object is a matter of changing its Strategy object to the one that implements the required algorithm

Visitor Pattern: Structure

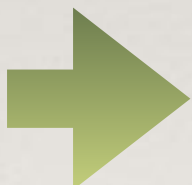


Visitor Pattern: Call Sequence



Visitor Pattern: Discussion

Represent an operation to be performed on the elements of an object structure.
Visitor lets you define a new operation without changing the classes of the elements on which it operations

- ❖ Many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid “polluting” their classes with these operations
 - ❖ Create two class hierarchies:
 - ❖ One for the elements being operated on
 - ❖ One for the visitors that define operations on the elements
- 

Procedural (conditional) Code

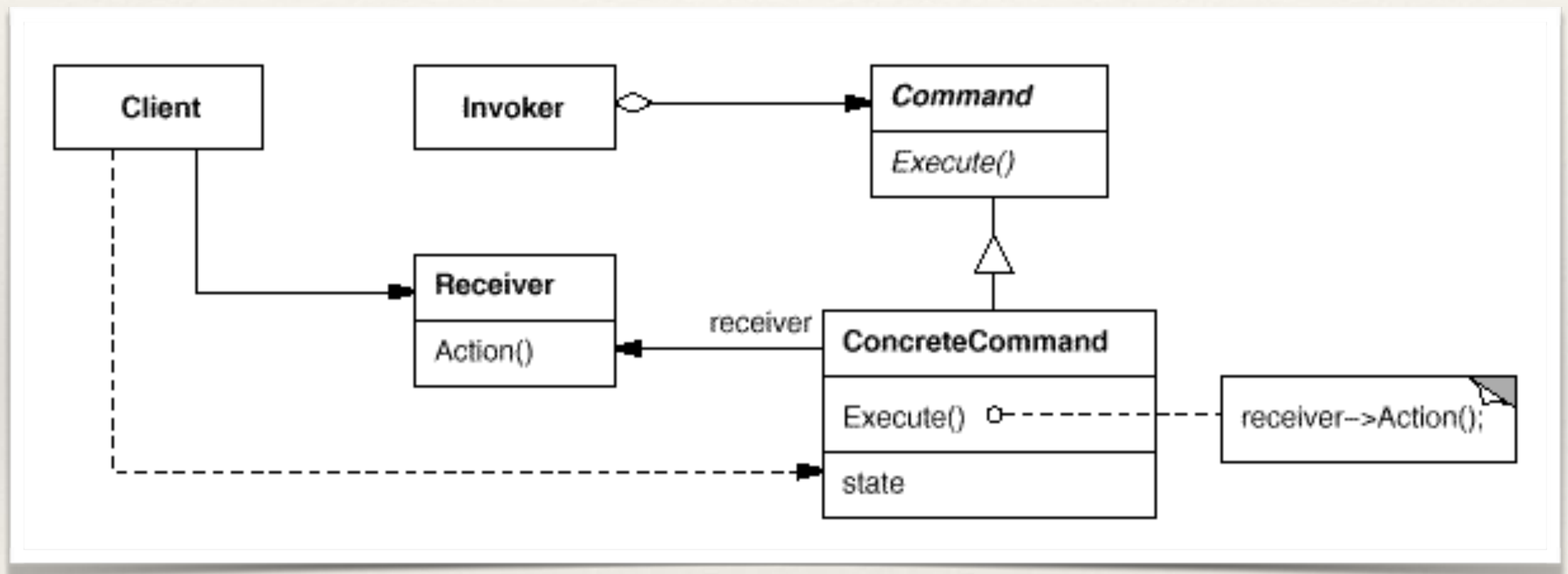
```
void interpret(uchar bytecodes[], int num_bytecodes) {  
    int pc = 0; // program counter  
    while(pc < num_bytecodes) {  
        switch(bytecodes[pc]) {  
            case BIPUSH: {  
                exec_stack.push(bytecodes[++pc]);  
                break;  
            }  
            case IMUL: {  
                imul();  
                break;  
            }  
            case IDIV: {  
                idiv();  
                break;  
            }  
            case IADD: {  
                iadd();  
                break;  
            }  
            case ISUB: {  
                isub();  
                break;  
            }  
        }  
        pc++;  
    }  
}
```

How to get rid of
conditional statements?

Hands-on Exercise

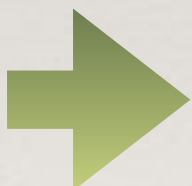
Refactor the code to remove the switch statements and use runtime polymorphism instead

Command Pattern: Structure



Command Pattern: Discussion

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

- ❖ Want to issue requests or execute commands without knowing anything about the operation being requested or the receiver of the request
- 
- ❖ Parameterize objects by an action to perform, with a common method such as Execute
 - ❖ Can be useful to specify, queue, and execute requests at different times; also to support undo or logging

Patterns Discussed in this Case-Study

✓ Facade

✓ Composite

✓ Factory method

✓ Flyweight

✓ Builder

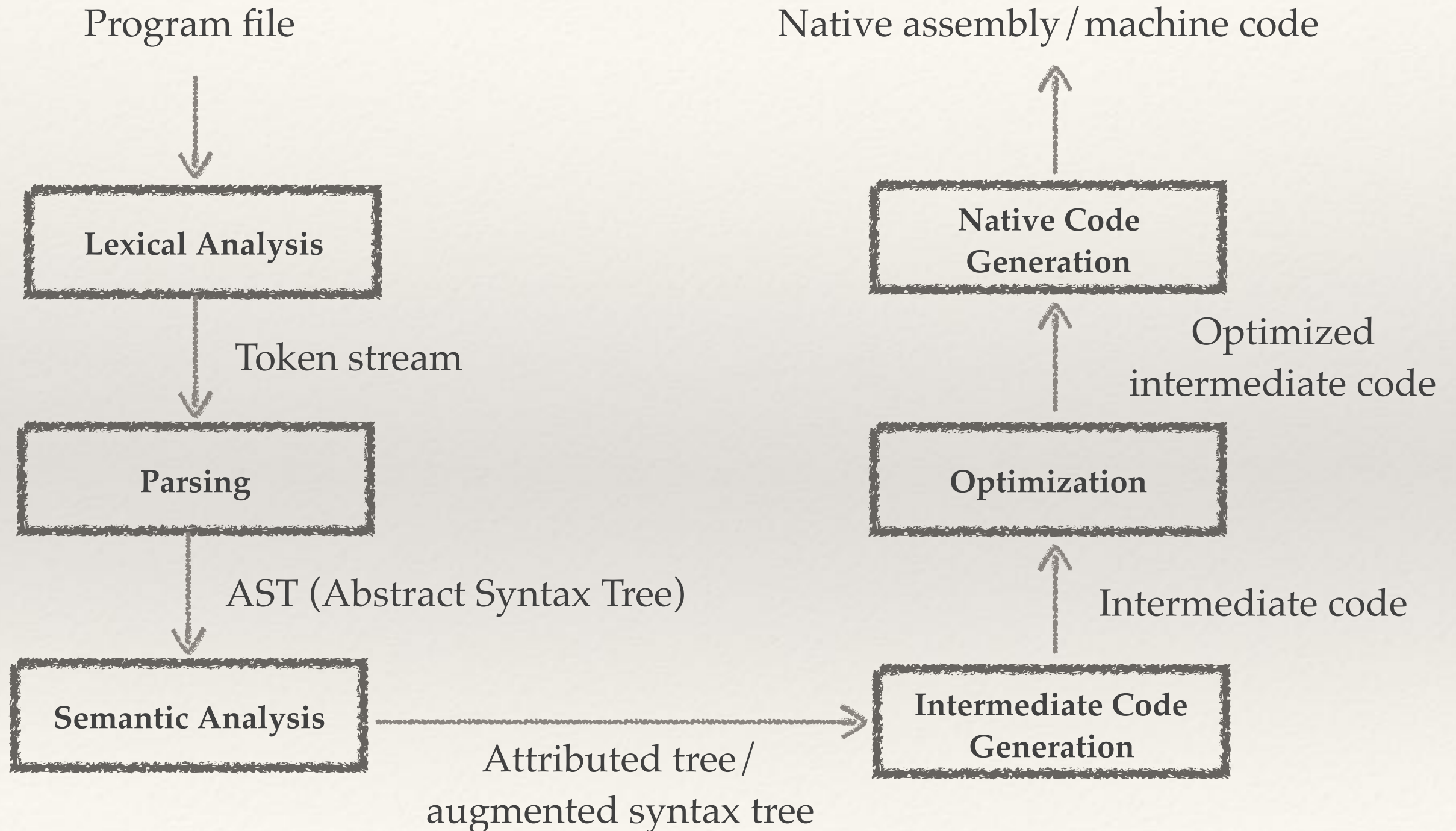
✓ Iterator pattern

✓ Strategy

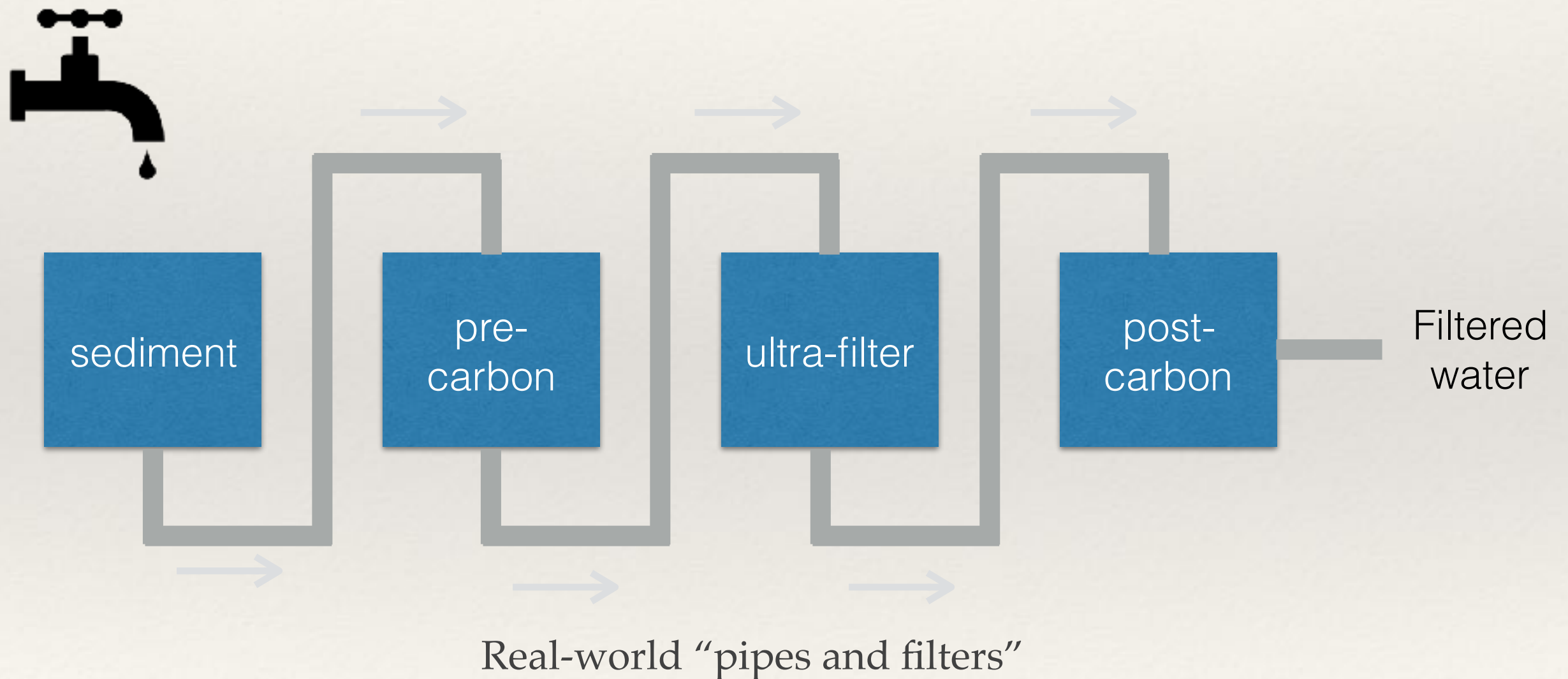
✓ Visitor

✓ Command

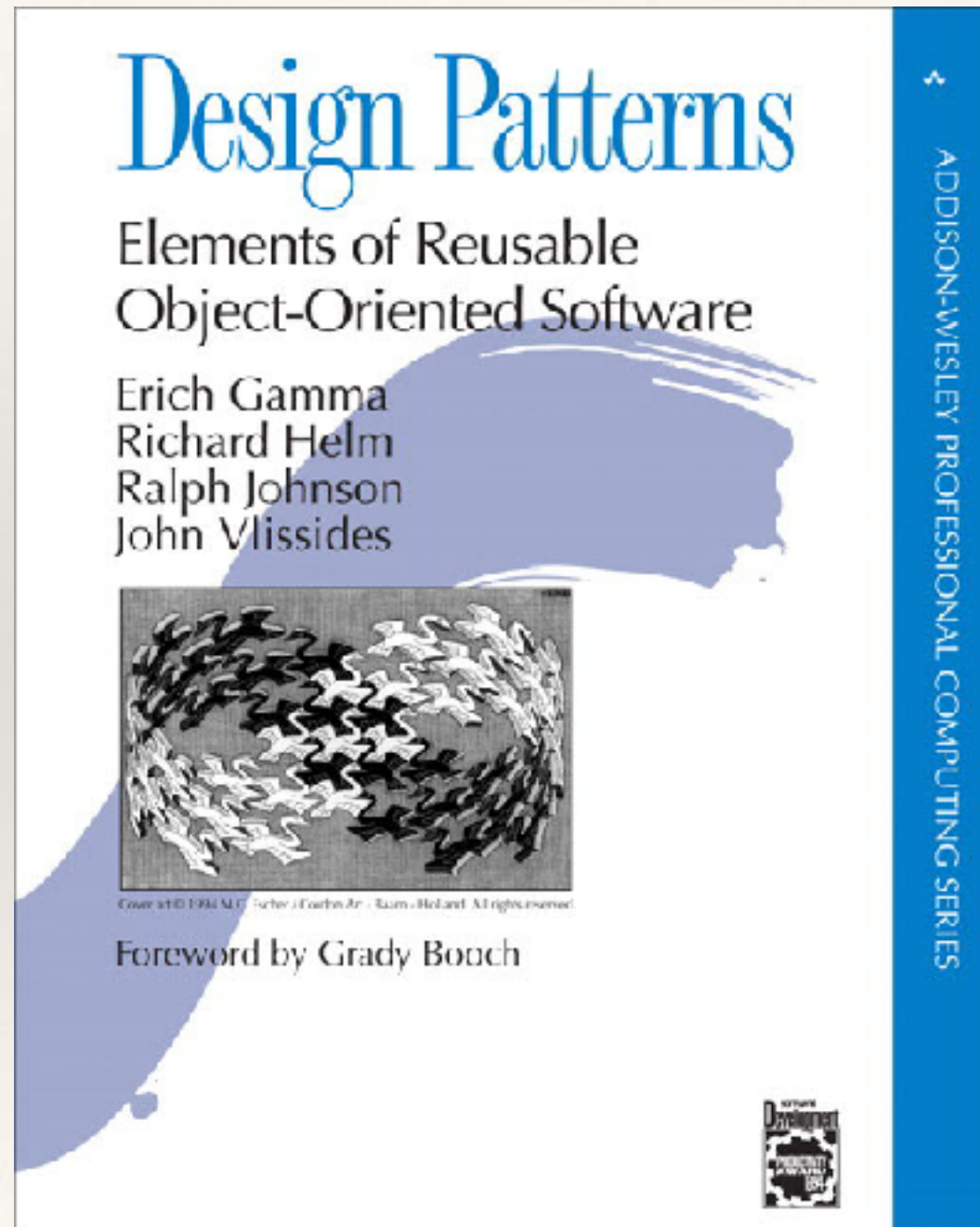
BTW, What Architecture Style is it?



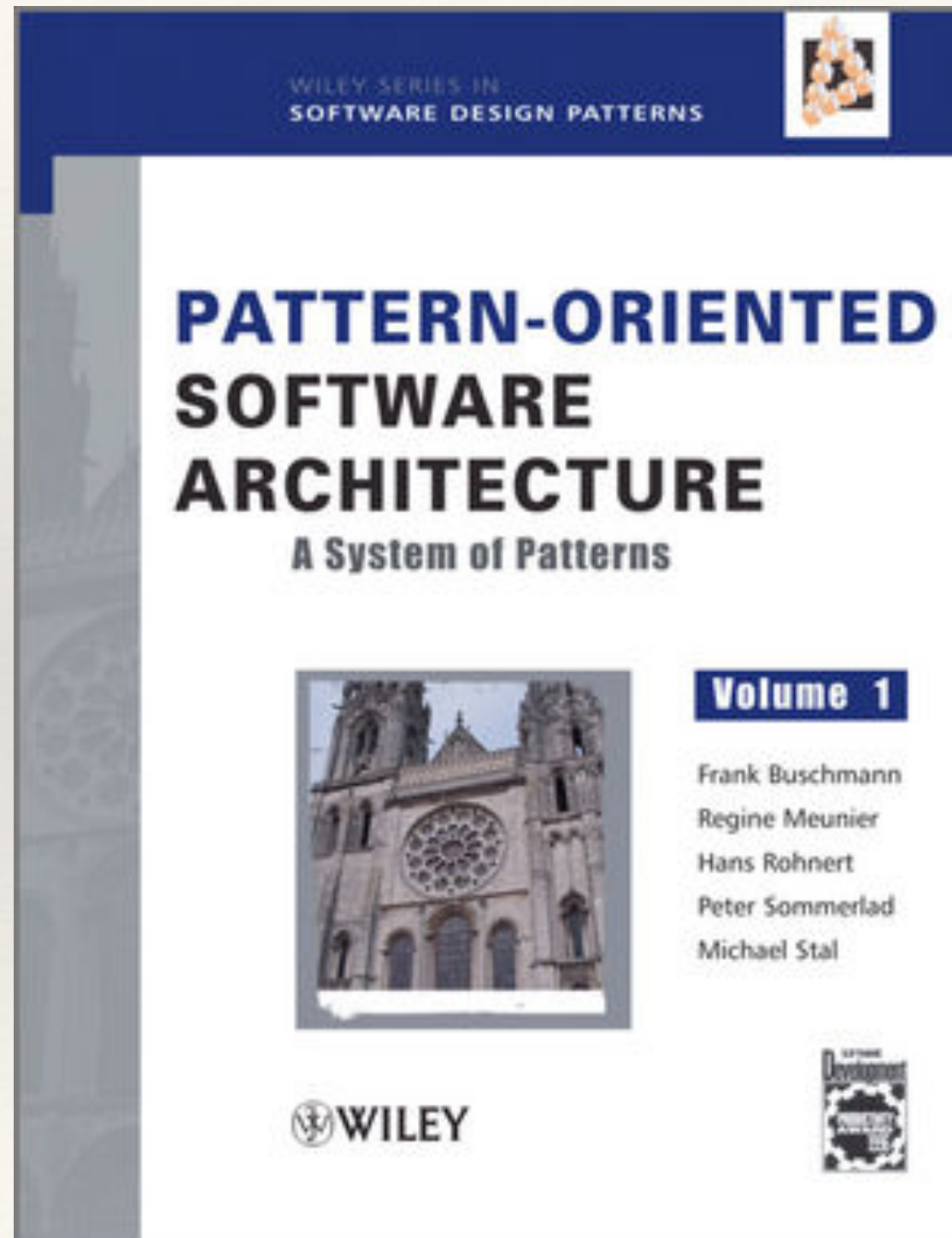
Its “pipe-and-filter” Style!



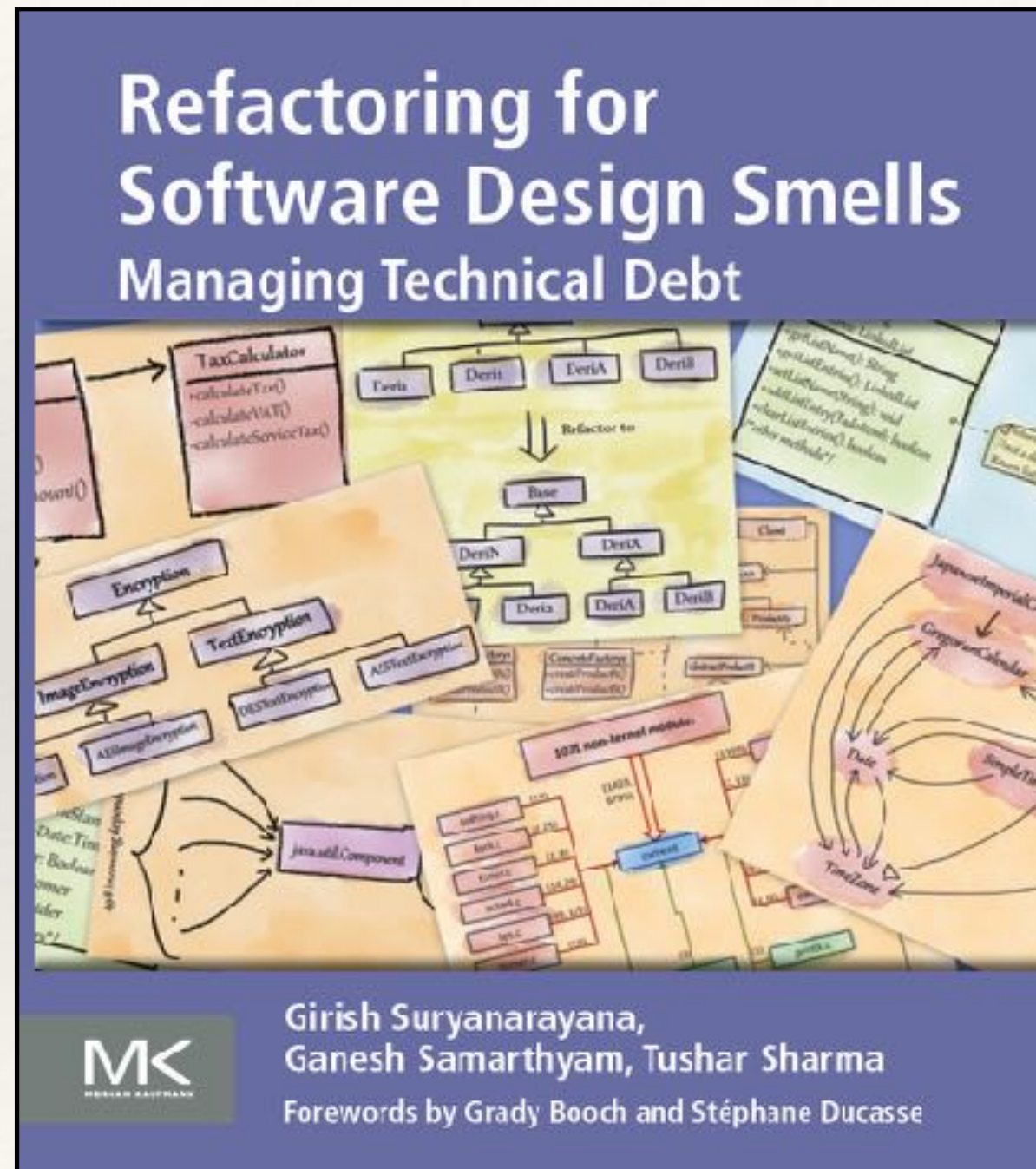
Classic Design Patterns Book

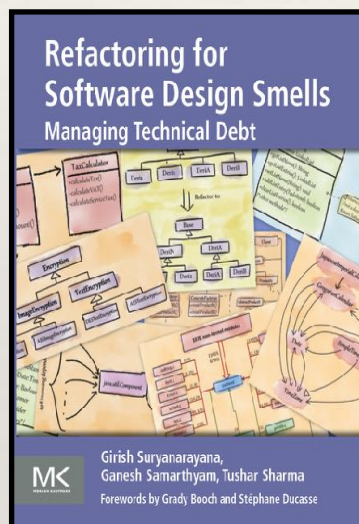


Classic Book on Architectural Patterns



“Principle-based” Refactoring Approach





ganesh@codeops.tech

www.codeops.tech

+91 98801 64463

[@GSamarthyam](https://www.linkedin.com/in/GSamarthyam)

slideshare.net/sgganesh

bit.ly/sgganesh

Code^{ops}