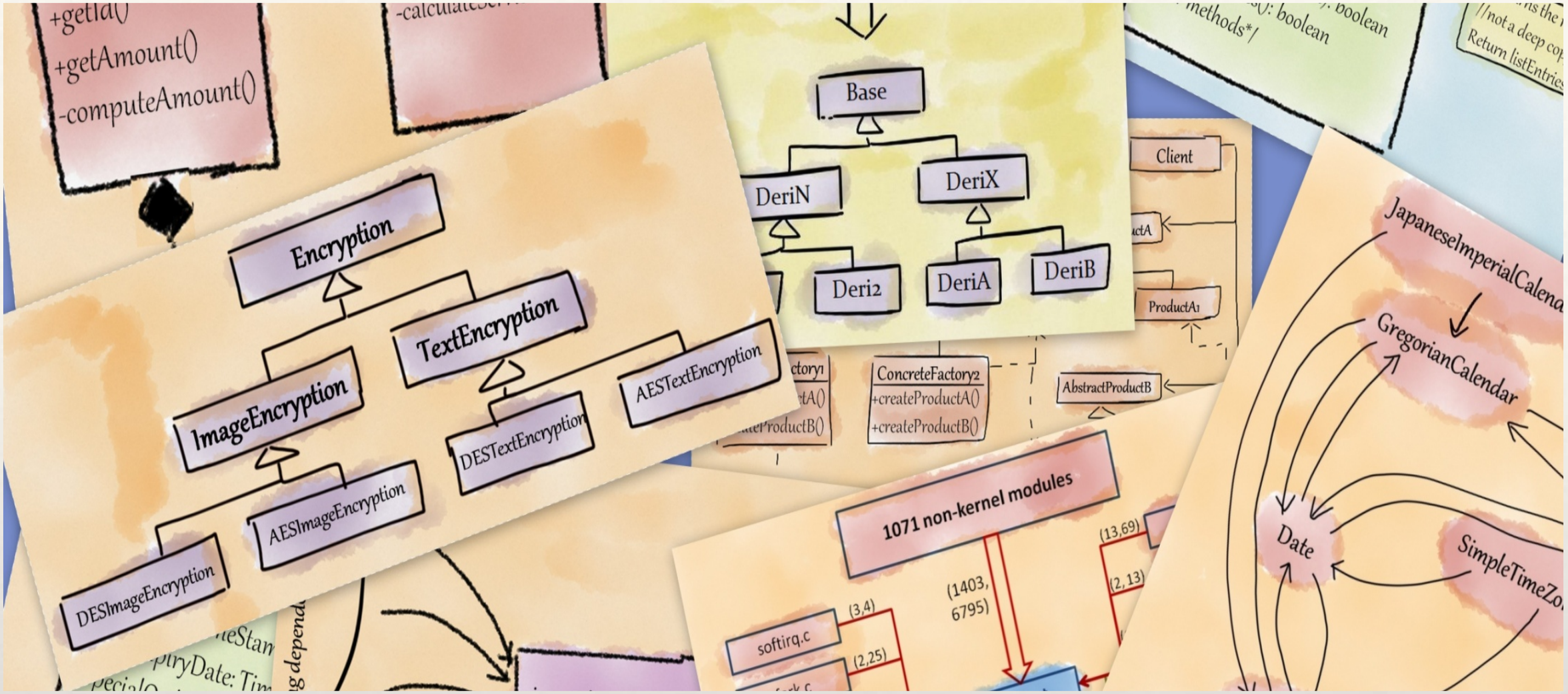


Compiler Case Study - Design Patterns

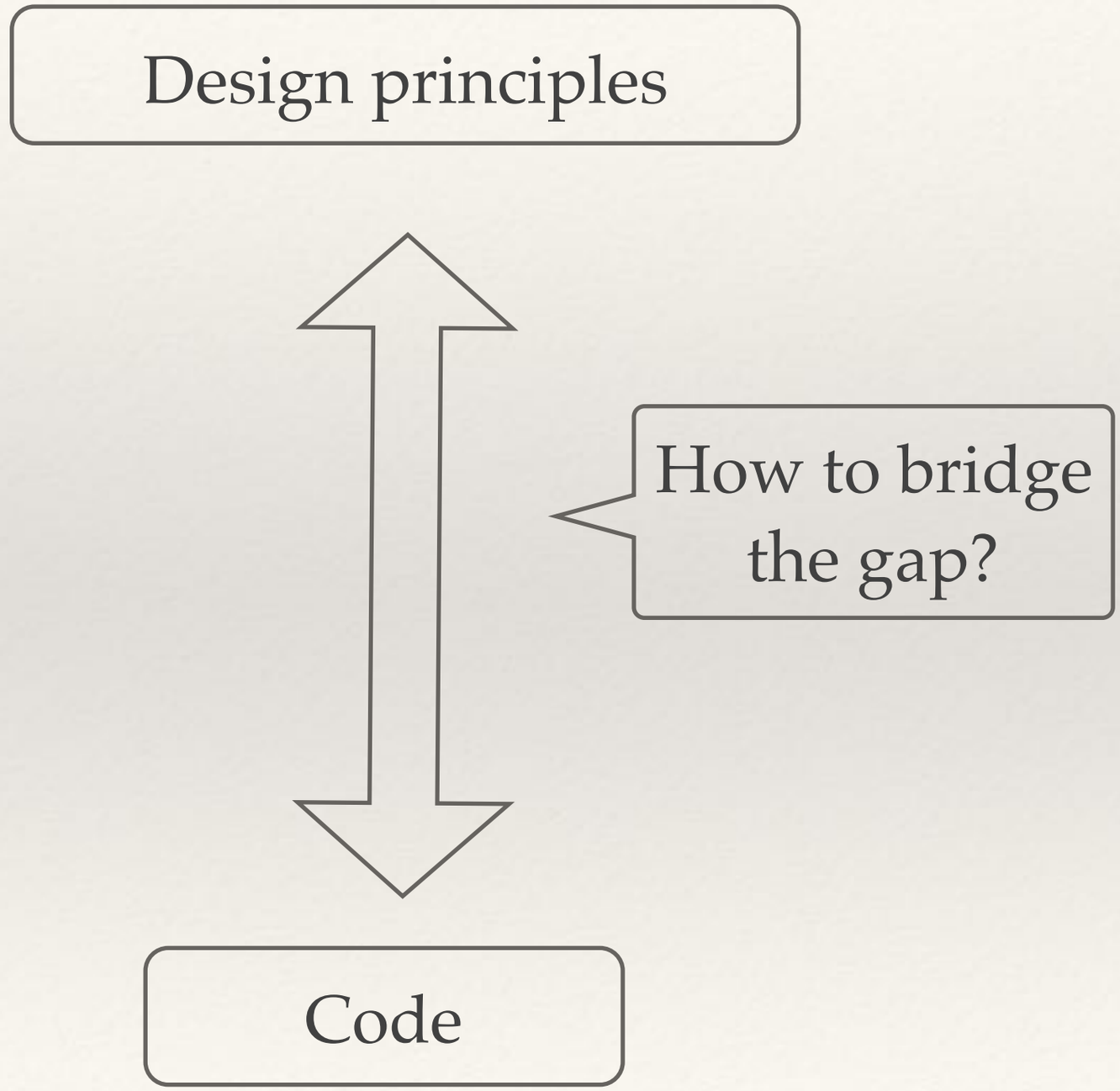
Ganesh Samarthyam
ganesh@codeops.tech

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?



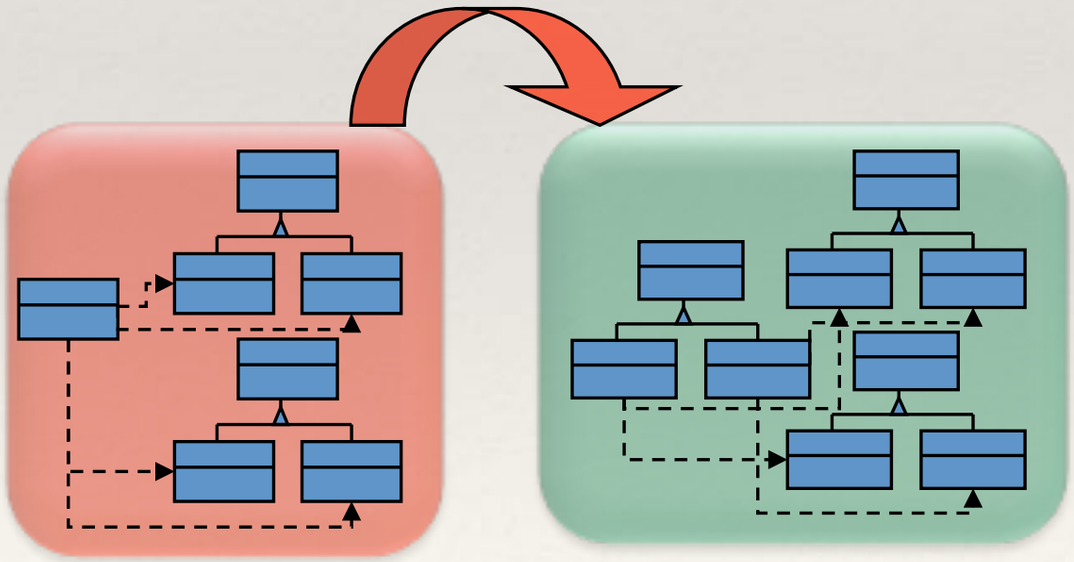
What are Smells?

"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*

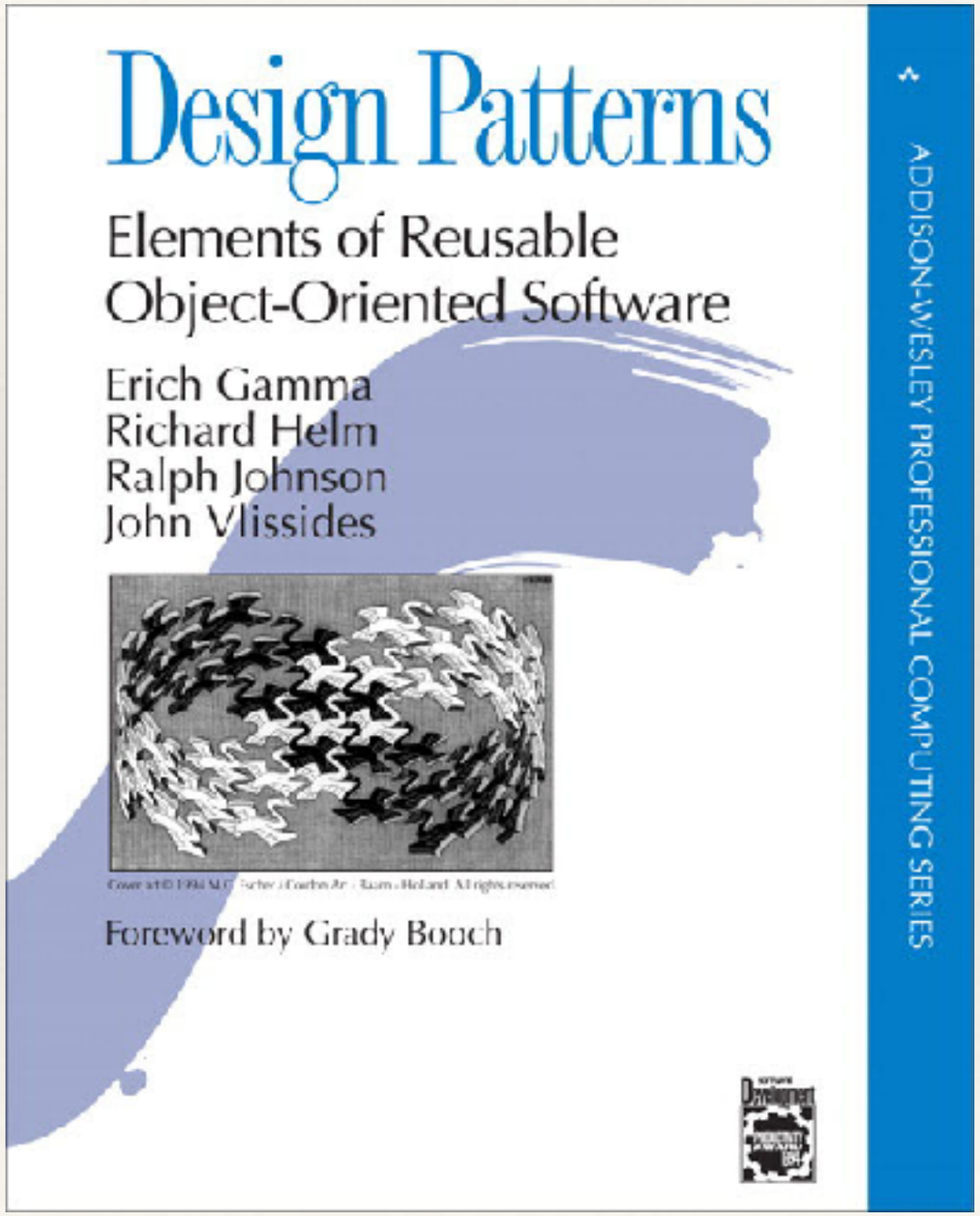
Refactor (verb): to restructure software by applying a series of refactorings without changing its observable behavior



What are Design Patterns?

recurrent solutions
to common
design problems

Pattern Name
Problem
Solution
Consequences



Principles to Patterns

```
Expr expr = Addition.Make(
                    Multiplication.Make(
                        Constant.Make(10),
                        Constant.Make(20)),
                    Constant.Make(10));
```

Encapsulate object creation

```
Expr expr = new ExprBuilder().Const(10).Mult(20).Plus(30).Build();
```

Builder 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

Design Patterns - Catalog

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**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

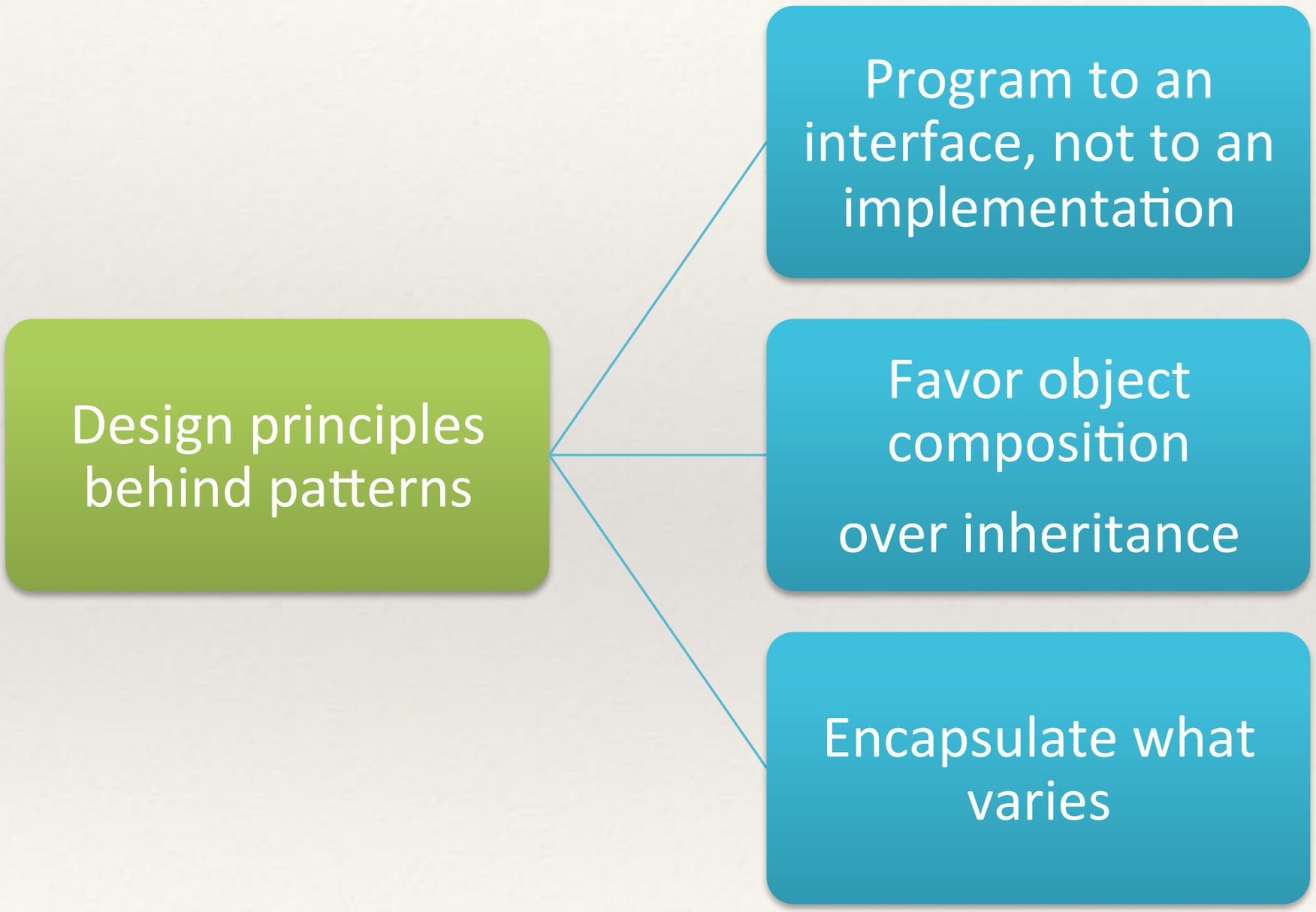
Source: "Design Patterns: Elements of Reusable Object-Oriented Software", Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Addison-Wesley,1994

Design Patterns - Catalog

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

Source: “Design Patterns: Elements of Reusable Object-Oriented Software”, Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Addison-Wesley,1994

3 Principles Behind Patterns



Compiler Case Study

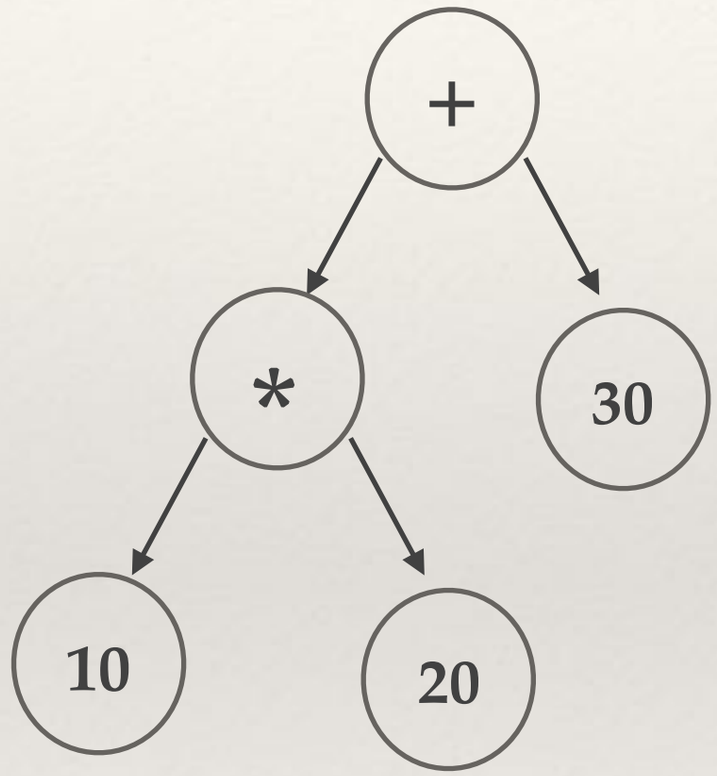
Case Study Overview

- 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
  - https://github.com/CodeOpsTech/CompilerDesignPatternsCSharp
  - https://github.com/CodeOpsTech/DesignPatternsJava

What You Need to Know

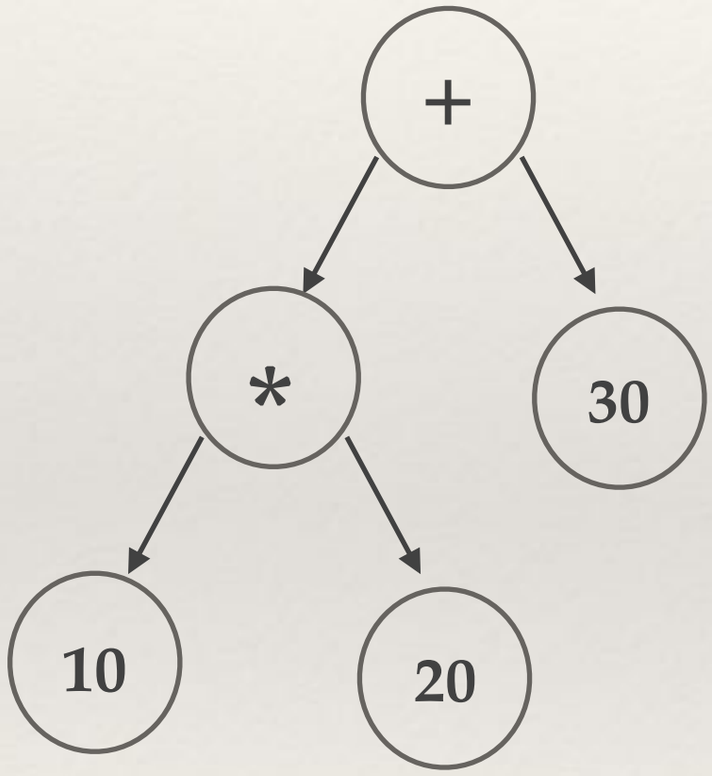
- Understand essential object oriented constructs such as runtime polymorphism (virtual functions), and principles (such as encapsulation)
- Know essential C# or Java language features - classes, inheritance, etc
- Data structures and algorithms (especially binary trees, tree traversals, and stacks)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

Simple Expression Tree Example



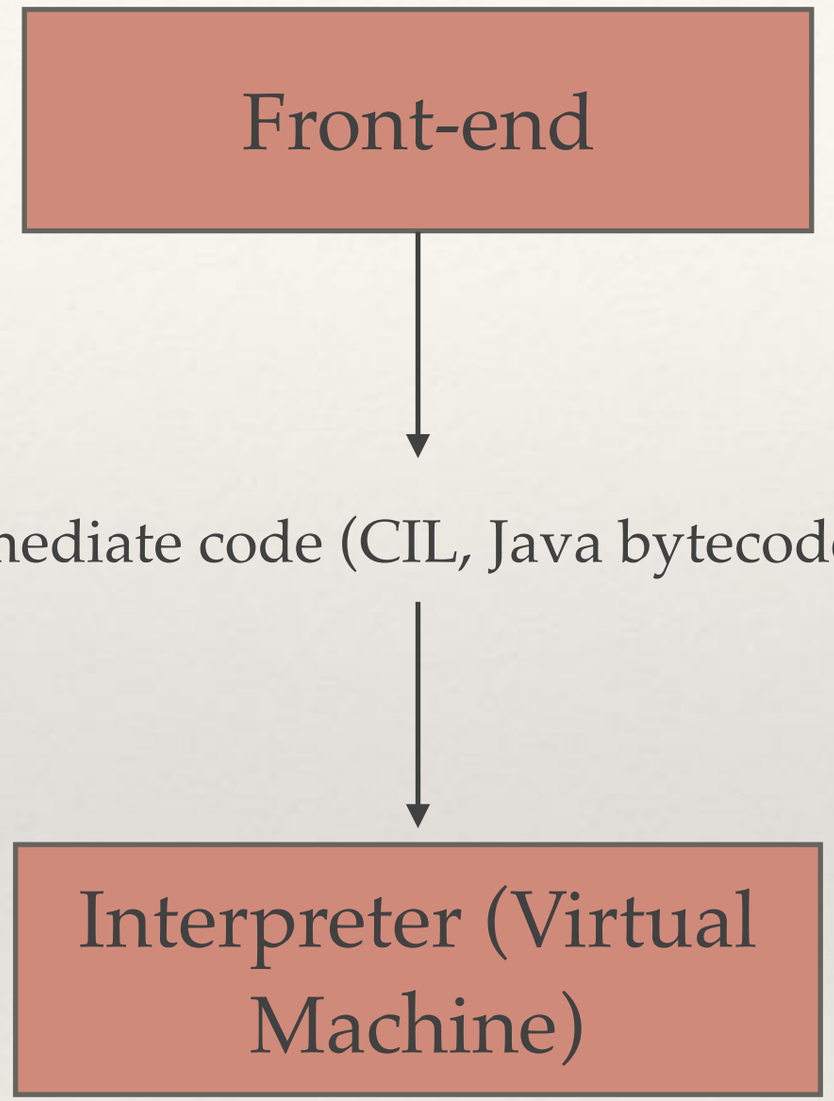
Expression: **((10 * 20) + 30)**

Post-order Traversal: Simulating Code Gen



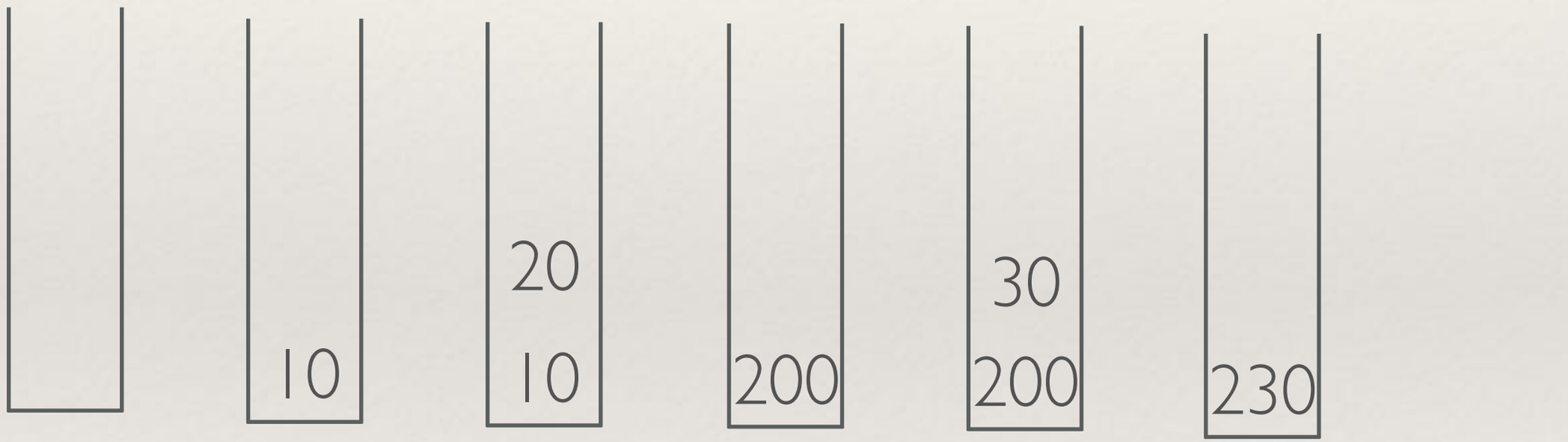
post-order
traversal
result

10 20 * 30 +


Front-end
Intermediate code (CIL, Java bytecodes, …)
Interpreter (Virtual Machine)

Using Stack for Evaluation

10 20 * 30 +



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
```

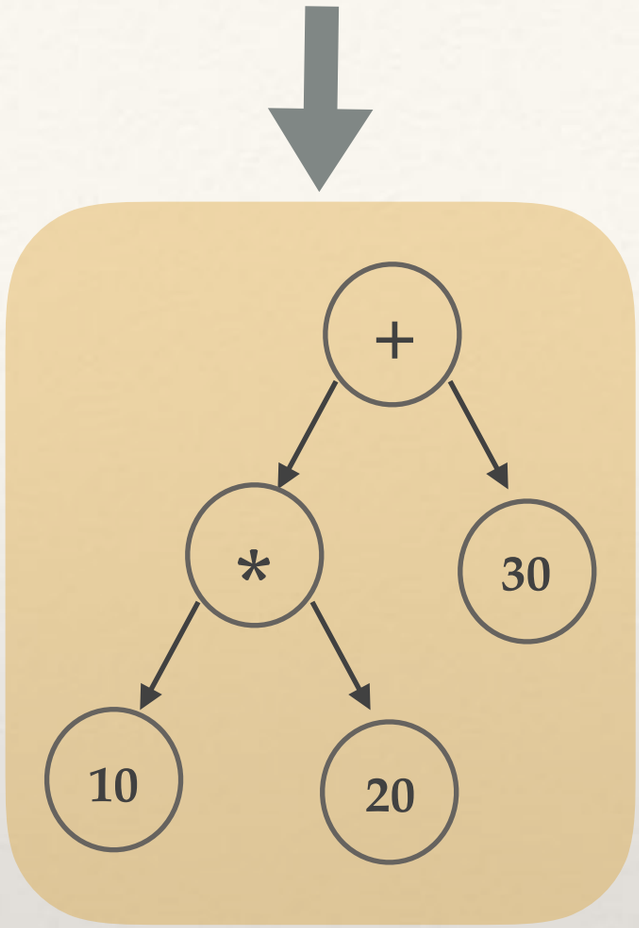
(bipush stands for “byte integer push”)

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)

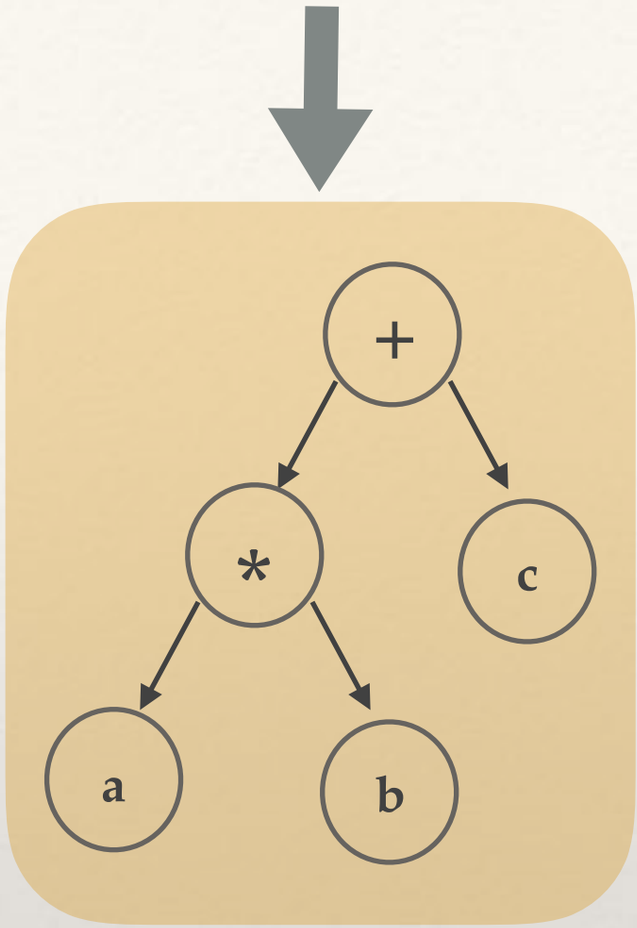


C# Compiler

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

.NET CLR (Common Language Runtime)

Source Code ((a * b) + c)

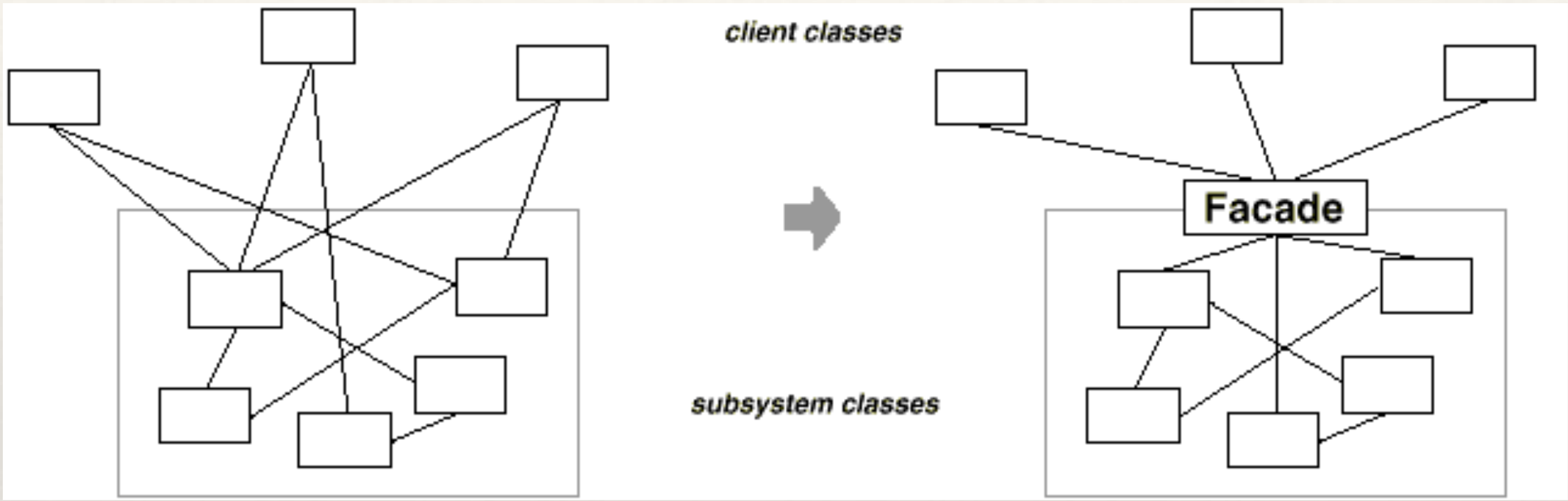


Java Compiler

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

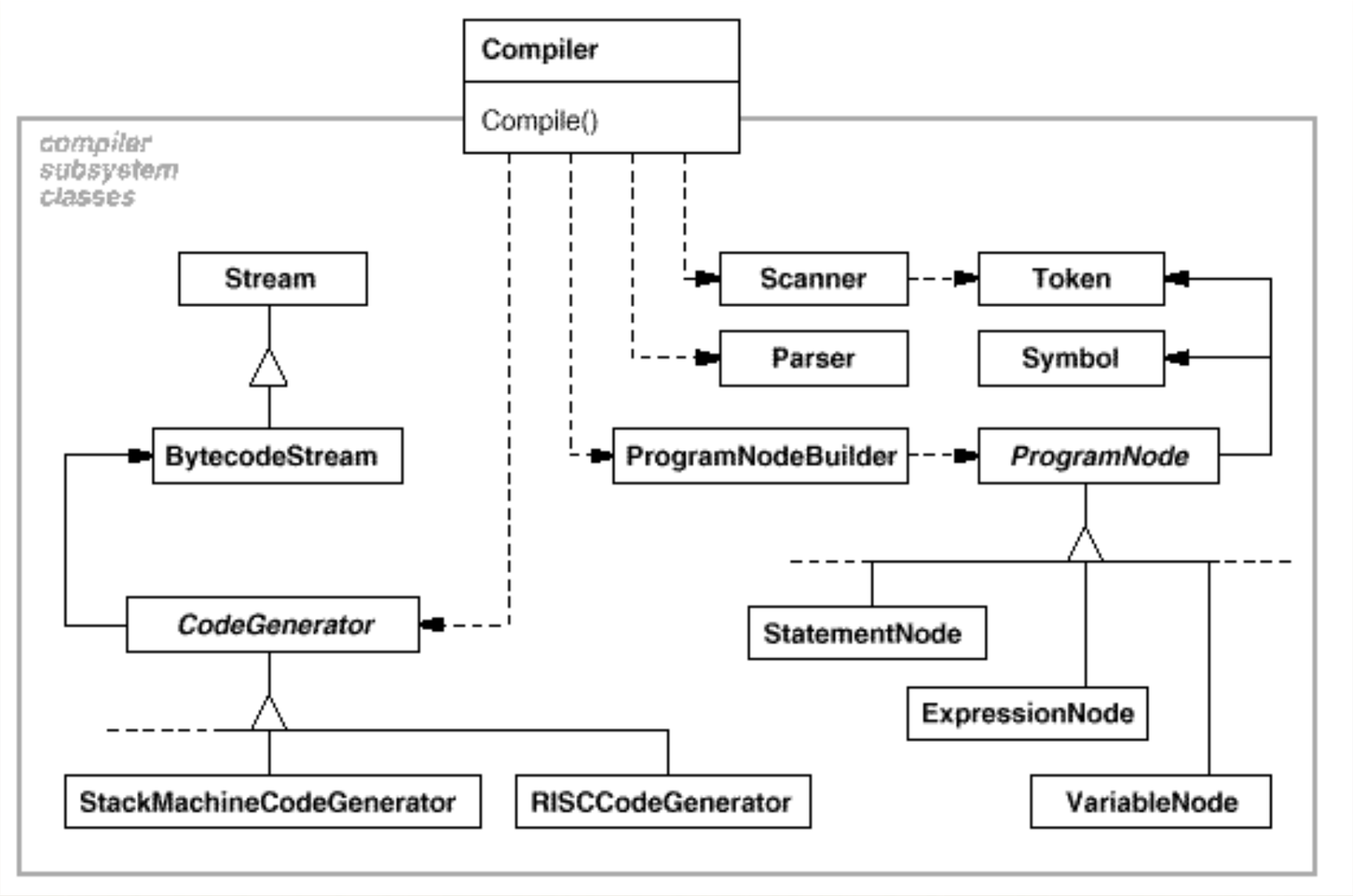
JVM (Java Virtual Machine)

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."

Facade Pattern in Compilers

```
public virtual void Compile(StreamReader input, BytecodeStream output) {
        Scanner scanner = new Scanner(input);
        var builder = new ProgramNodeBuilder();
        Parser parser = new Parser();

        parser.Parse(scanner, builder);

        RISCCodeGenerator generator = new RISCCodeGenerator(output);
        ProgramNode parseTree = builder.GetRootNode();
        parseTree.Traverse(generator);
}
```

Procedural (conditional) Code

```
public void GenCode()
        {
            if (_left == null && _right == null)
            {
                Console.WriteLine("ldc.i4.s " + _value);
            }
            else
            {
                // its an intermediate node
                _left.GenCode();
                _right.GenCode();
                switch (_value)
                {
                    case "+":
                        Console.WriteLine("add");
                        break;
                    case "-":
                        Console.WriteLine("sub");
                        break;
                    case "*":
                        Console.WriteLine("mul");
                        break;
                    case "/":
                        Console.WriteLine("div");
                        break;
                    default:
                        Console.WriteLine("Not implemented yet!");
                        break;
                }
            }
        }
```

How to get rid of conditional statements?

Hands-on Exercise

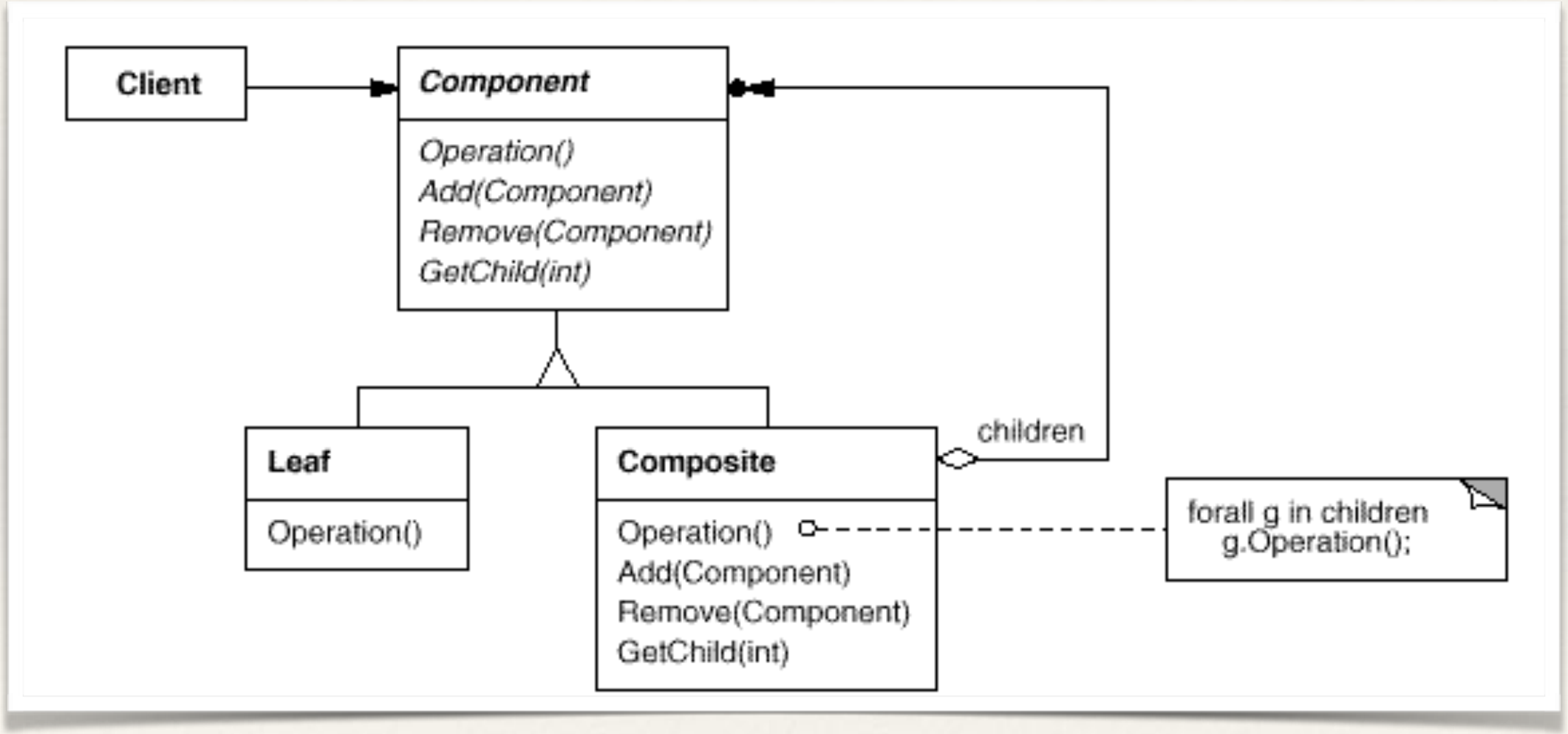
Refactor the "../Composite/Before/Compiler.cs" to get rid of conditional statements and use runtime polymorphism instead

Hands-on Exercise

```
internal abstract class Expr
    {
        public abstract void GenCode();
    }

    internal class Constant : Expr
    {
        private readonly int _val;

        public Constant(int val)
        {
            _val = val;
        }

        public override void GenCode()
        {
            Console.WriteLine("ldc.i4.s " + _val);
        }
    }
```

```
internal class BinaryExpr : Expr
    {
        private readonly Expr _left;
        private readonly Expr _right;

        public BinaryExpr(Expr arg1, Expr arg2)
        {
            _left = arg1;
            _right = arg2;
        }

        public override void GenCode()
        {
            _left.GenCode();
            _right.GenCode();
        }
    }

    internal class Plus : BinaryExpr
    {
        public Plus(Expr arg1, Expr arg2) : base(arg1, arg2)
        {
        }

        public override void GenCode()
        {
            base.GenCode();
            Console.WriteLine("add");
        }
    }
```

Composite Pattern



Source: “Design Patterns: Elements of Reusable Object-Oriented Software”, Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Addison-Wesley,1994

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

BTW, Did You Observe?

```
// ((10 * 20) + 30)
var expr = new Expr(
                new Expr(
                    new Expr(null, "10", null), "*", new Expr(null, "20", null)),
                    "+",
                    new Expr(null, "30", null));
expr.GenCode();
```

```
// ((10 * 20) + 10)
Expr expr = new Plus(
                new Multiply(
                    new Constant(10),
                    new Constant(20)),
                new Constant(30));
expr.GenCode();
```

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)
```

Solution

```
internal class Constant : Expr
    {
        private readonly int _val;

        public Constant(int arg)
        {
            _val = arg;
        }

        public override int Interpret()
        {
            return _val;
        }
    }

    internal class Plus : Expr
    {
        private readonly Expr _left;
        private readonly Expr _right;

        public Plus(Expr arg1, Expr arg2)
        {
            _left = arg1;
            _right = arg2;
        }

        public override int Interpret()
        {
            return _left.Interpret() + _right.Interpret();
        }
    }
```

Register-Based Instructions (x86)

```
int main() {
  int a = 10;
  int b = 20;
  int c = 30;
  return ((a * b) + c);
}
```

cc -S expr.c

```
  movl  $10, -8(%rbp)
  movl  $20, -12(%rbp)
  movl  $30, -16(%rbp)
  movl  -8(%rbp), %eax
  imull -12(%rbp), %eax
  addl  -16(%rbp), %eax
  popq  %rbp
  retq
```

Register-Based Instructions (Android)

```
((a * b) + c);
```

javac -source 1.7 -target 1.7 expr.java
dx --dex --output=expr.dex expr.class
dexdump -d expr.dex

```
const/16 v1, #int 10 // #a
const/16 v2, #int 20 // #14
mul-int v0, v1, v2
const/16 v3, #int 30 // #1e
add-int/2addr v0, v3
return v0
```

Hands-on Exercise

Generate register-based instructions for the given expression ((10 * 20) + 30)

You can generate for an imaginary machine (i.e., a virtual machine like Dalvik/Android) or a real-machine (like x86-64)

Solution

```
internal class Register
    {
        private readonly int _index;

        private Register(int index)
        {
            _index = index;
        }

        public static Register GetRegister(int index)
        {
            return new Register(index);
        }
        // FreeRegister to be implemented
        // not implemented for now for the sake of simplicity

        public int GetIndex()
        {
            return _index;
        }
    }
```

Solution

```
abstract class Expr {
    public abstract Register GenCode();
}

class Constant : Expr {
    int val;
    public Constant(int arg) {
        val = arg;
    }
    public override Register GenCode() {
        Register targetRegister = RegisterAllocator.GetNextRegister();
        Console.WriteLine("const/16 v{0}, #int {1}", targetRegister.GetIndex(), val);
        return targetRegister;
    }
}

class Plus : Expr {
    private Expr left, right;
    public Plus(Expr arg1, Expr arg2) {
        left = arg1;
        right = arg2;
    }
    public override Register GenCode() {
        Register firstRegister = left.GenCode();
        Register secondRegister = right.GenCode();
        Register targetRegister = RegisterAllocator.GetNextRegister();
        Console.WriteLine("add-int v{0}, v{1}, v{2}", targetRegister.GetIndex(), firstRegister.GetIndex(),
secondRegister.GetIndex());
        return targetRegister;
    }
}
```

Solution

```
((10 + 20) + 30);
```

Assuming infinite registers from v0,
with target register as the first argument,
here is the mnemonic code generated for our
imaginary (i.e., virtual) machine

```
const/16 v0, #int 10
const/16 v1, #int 20
add-int v2, v0, v1
const/16 v3, #int 30
add-int v4, v2, v3
```

How to Improve the Tree Creation?

```
Expr expr = new Plus(
                    new Multiply(
                        new Constant(10),
                        new Constant(20)),
                    new Constant(30));
```

```
var expr = Addition.Make(
                   Multiplication.Make(
                       Constant.Make(10),
                       Constant.Make(20)),
                   Constant.Make(10));
```

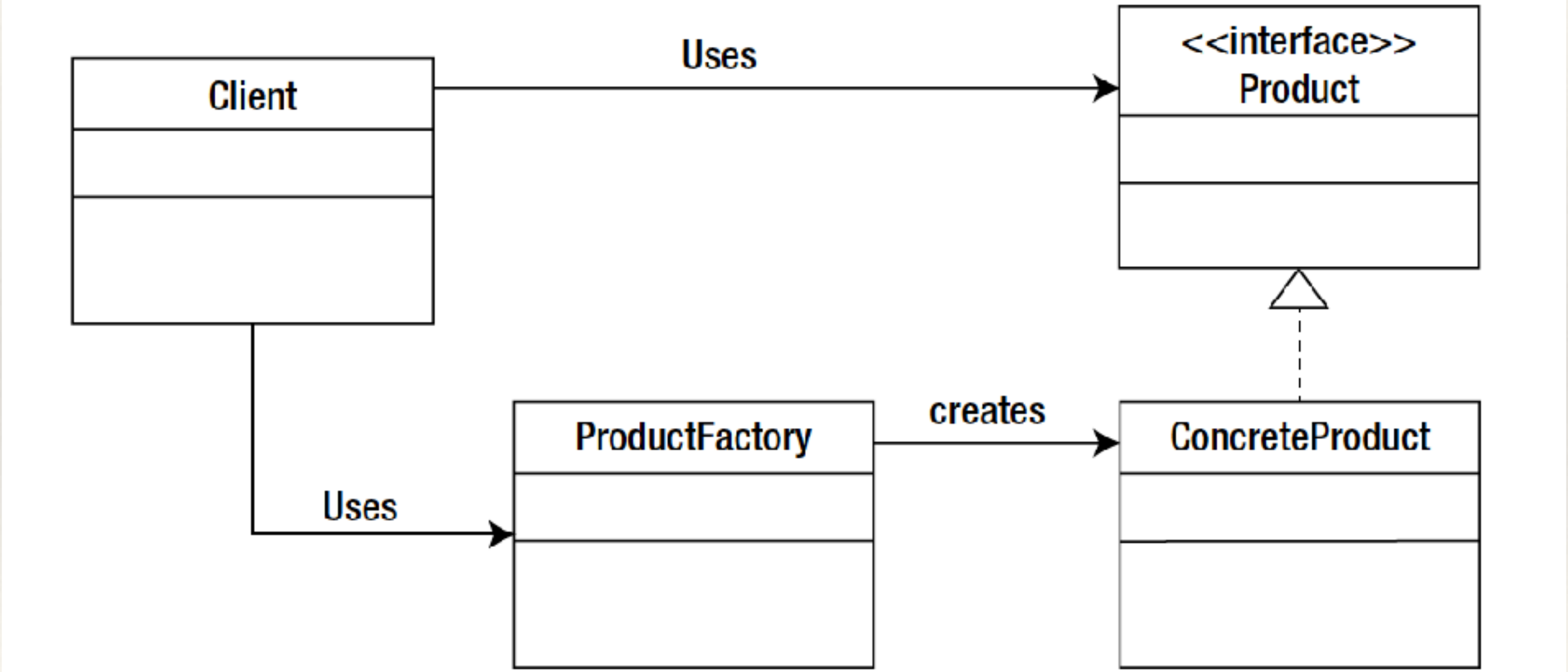
Hands-on Exercise: Solution

```
internal class Constant : Expr
    {
        private readonly int _val;

        public Constant(int val)
        {
            _val = val;
        }

        public override void GenCode()
        {
            Console.WriteLine("ldc.i4.s " + _val);
        }
    }
```



```
internal class Constant : Expr
    {
        private readonly int _val;

        private Constant(int val)
        {
            _val = val;
        }

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

        public override void GenCode()
        {
            Console.WriteLine("ldc.i4.s " + _val);
        }
    }
```

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

Scenario

```
internal class Constant : Expr
    {
        private readonly int _val;

        private Constant(int val)
        {
            _val = val;
        }

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

        public override void GenCode()
        {
            Console.WriteLine("ldc.i4.s " + _val);
        }
    }
```

For expressions like ((10 * 20) + 10), why should we create different Constant objects? Can we reuse them?

Hands-on Exercise

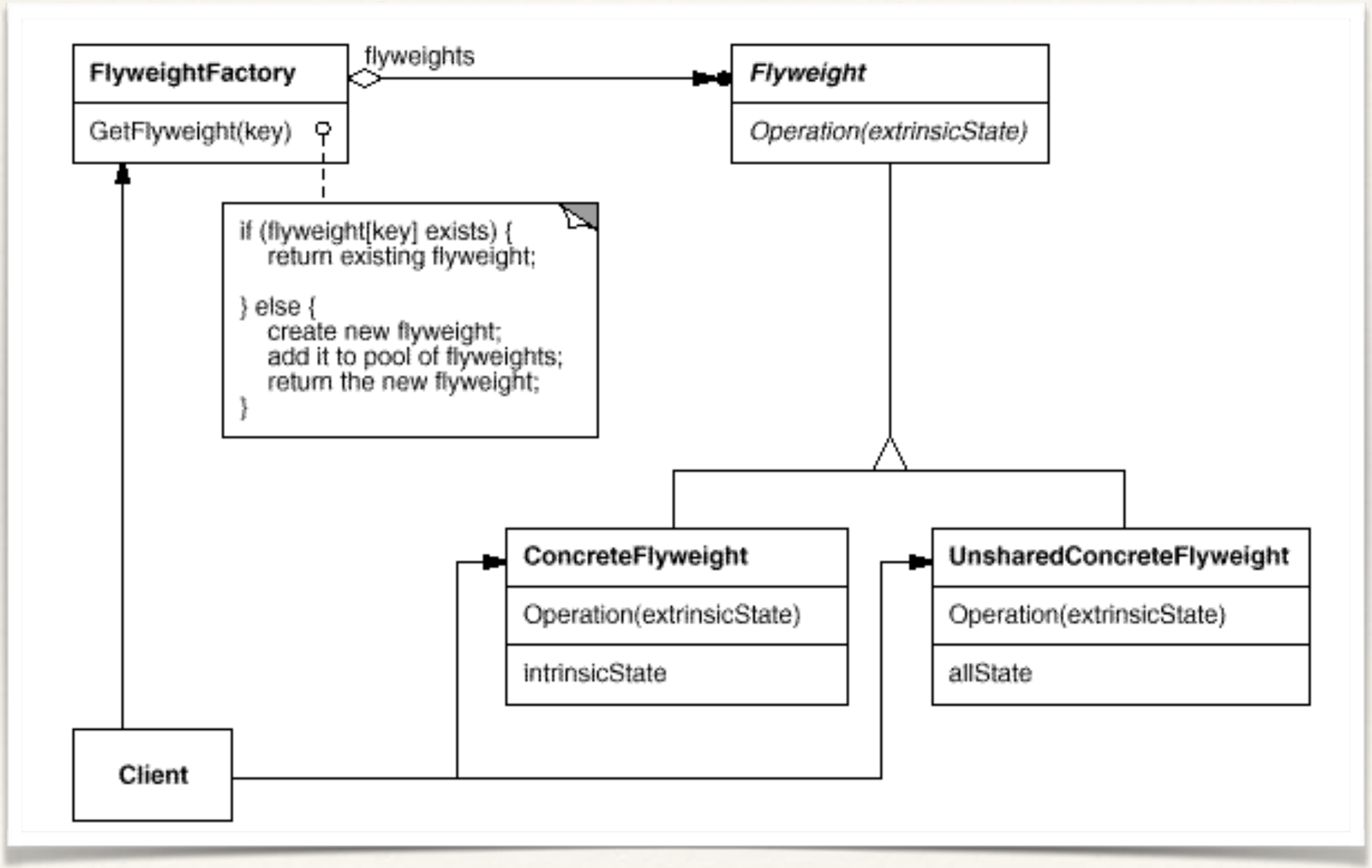
```
public static Constant Make(int arg)
{
    return new Constant(arg);
}
```

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

Hands-on Exercise: Solution

```
internal class Constant : Expr
    {
        private static readonly
          Dictionary<int, Constant> Pool = new Dictionary<int, Constant>();
        private readonly int _val;

        private Constant(int val)
        {
            _val = val;
        }

        public static Constant Make(int arg)
        {
            if (!Pool.ContainsKey(arg)) Pool[arg] = new Constant(arg);
            return Pool[arg];
        }

        public override void GenCode()
        {
            Console.WriteLine("ldc.i4.s " + _val);
        }
    }
```

Flyweight Pattern: Structure



Source: “Design Patterns: Elements of Reusable Object-Oriented Software”, Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Addison-Wesley,1994

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

Scenario

This code still sucks - if the expression becomes more complex, can you really read it? Can we simplify this?

```
// ((10 * 20) + 10)
var expr = Addition.Make(
                Multiplication.Make(
                    Constant.Make(10),
                    Constant.Make(20)),
                Constant.Make(10));
expr.GenCode();
```

Hands-on Exercise

```
// ((10 * 20) + 10)
var expr = Addition.Make(
                Multiplication.Make(
                    Constant.Make(10),
                    Constant.Make(20)),
                Constant.Make(10));
expr.GenCode();
```

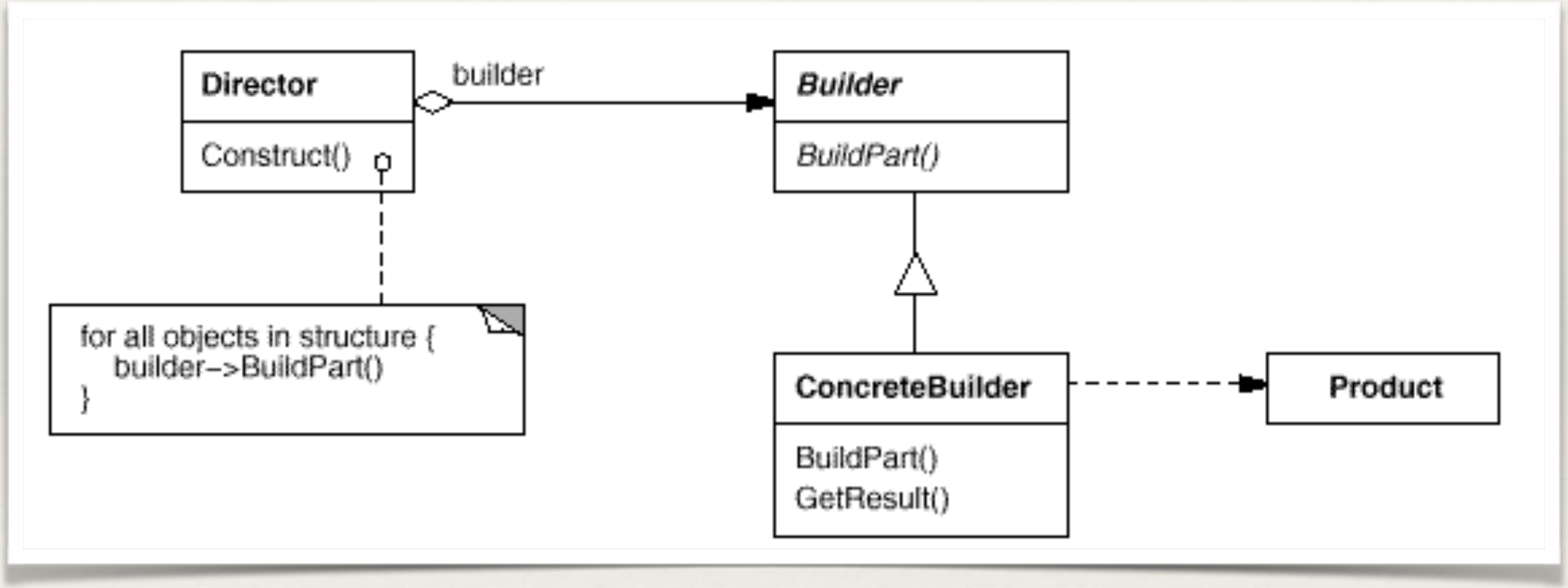
Improve this code to use “builder pattern”.

```
var expr = new ExprBuilder().Const(10).Mult(20).Plus(30).Build();
```

Hands-on Exercise: Solution

```
internal class ExprBuilder
    {
        private Expr _expr;

        public ExprBuilder Const(int arg)
        {
            _expr = Constant.Make(arg);
            return this;
        }

        public ExprBuilder Plus(int arg)
        {
            _expr = new Addition(_expr, Constant.Make(arg));
            return this;
        }

        public ExprBuilder Mult(int arg)
        {
            _expr = new Multiplication(_expr, Constant.Make(arg));
            return this;
        }

        public Expr Build()
        {
            return _expr;
        }
    }
```

Builder Pattern: Structure



Source: “Design Patterns: Elements of Reusable Object-Oriented Software”, Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Addison-Wesley,1994

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

Scenario

```
internal class Constant : Expr
    {
        private readonly int _val;

        public Constant(int val)
        {
            _val = val;
        }

        public override void GenCode()
        {
            Console.WriteLine("ldc.i4.s " + _val);
        }
    }

internal class Plus : BinaryExpr
    {
        public Plus(Expr arg1, Expr arg2) : base(arg1, arg2)
        {
        }

        public override void GenCode()
        {
            base.GenCode();
            Console.WriteLine("add");
        }
    }
```

Provide the ability to traverse the aggregate structure without exposing implementation details (also use iteration instead of recursion)

Hands-on Exercise

```
        public override void GenCode()
        {
            base.GenCode();
            Console.WriteLine("add");
        }
```

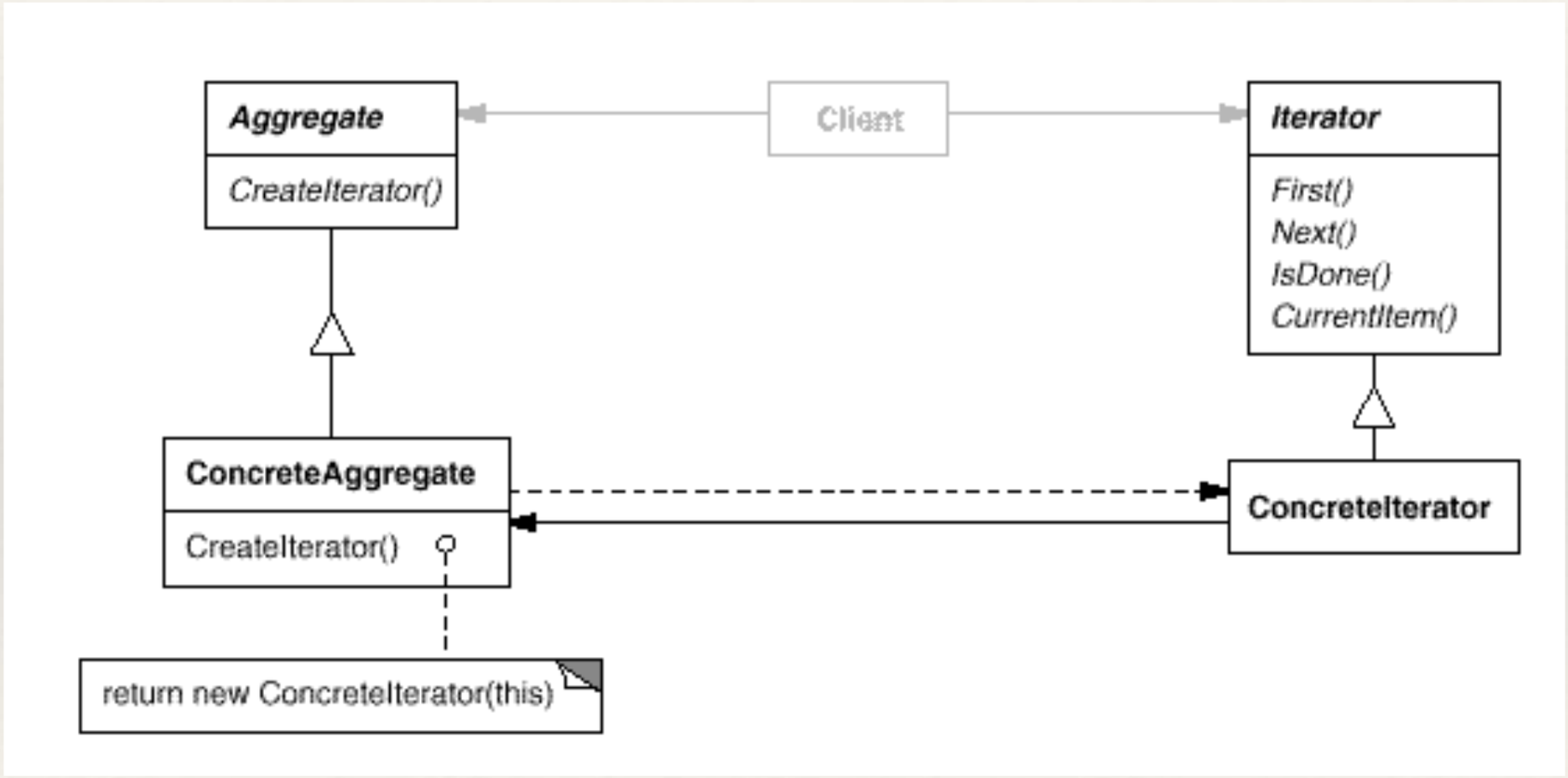
Traverse the expression tree using “external” iteration

Hands-on Exercise: Solution

```
internal class ExprIterator
    {
        public static IEnumerable<Expr> Traverse(Expr node)
        {
            if (node != null)
            {
                foreach (var left in Traverse(node.GetLeft())) yield return left;
                foreach (var right in Traverse(node.GetRight())) yield return right;
                yield return node;
            }
        }
    }
```

```
foreach (var node in ExprIterator.Traverse(expr))
    node.GenCode();
```

Iterator Pattern: Structure



Source: “Design Patterns: Elements of Reusable Object-Oriented Software”, Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Addison-Wesley,1994

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

Scenario

```
class Plus extends Expr {
        private Expr left, right;
        public Plus(Expr arg1, Expr arg2) {
                left = arg1;
                right = arg2;
        }
        public void genCode() {
                left.genCode();
                right.genCode();
                if(t == Target.JVM) {
                        System.out.println("iadd");
                }
                else { / / DOTNET
                        System.out.println("add");
                }
        }
}
```

How to separate:

a) code generation logic from node types?
b) how to support different target types?

Hands-on Exercise

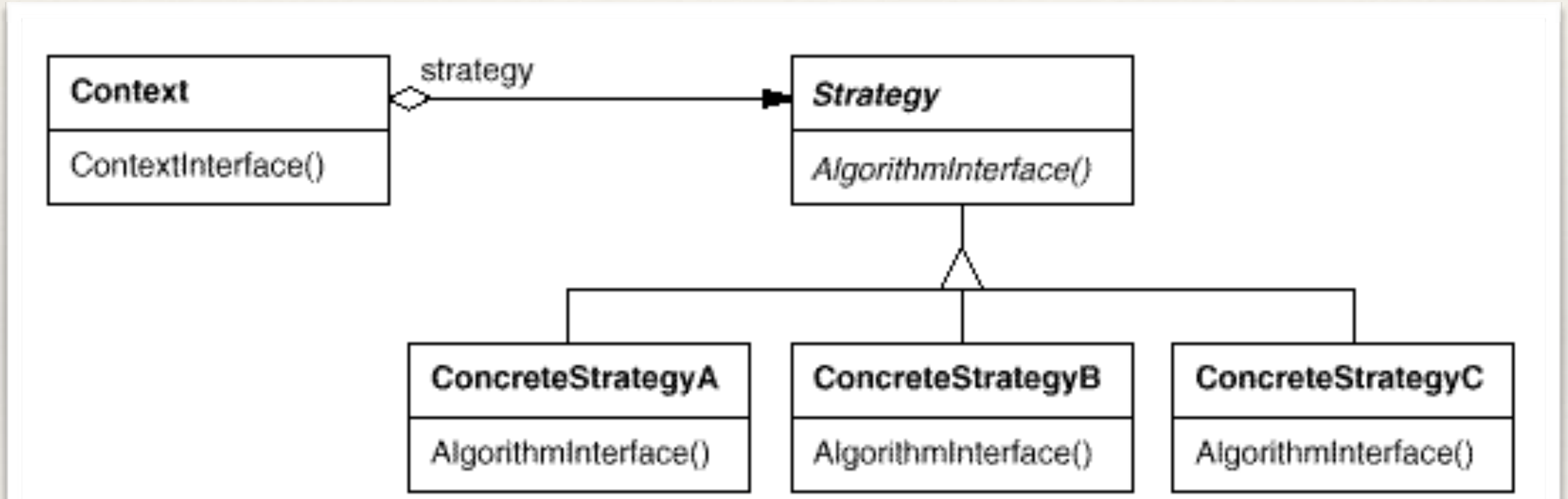
```
if(t == Target.JVM) {
        System.out.println("iadd");
}
else { // DOTNET
        System.out.println("add");
}
```

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

Hands-on Exercise

```
internal abstract class Target
    {
        public abstract void GenCode(Constant constant);
        public abstract void GenCode(Plus plus);
        public abstract void GenCode(Mult mult);
    }

    internal class JvmTarget : Target
    {
        public override void GenCode(Constant constant)
        {
            Console.WriteLine("bipush " + constant.GetValue());
        }

        public override void GenCode(Plus plus)
        {
            Console.WriteLine("iadd");
        }

        public override void GenCode(Mult mult)
        {
            Console.WriteLine("imul");
        }
    }
```

Strategy Pattern: Structure



Source: “Design Patterns: Elements of Reusable Object-Oriented Software”, Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Addison-Wesley,1994

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

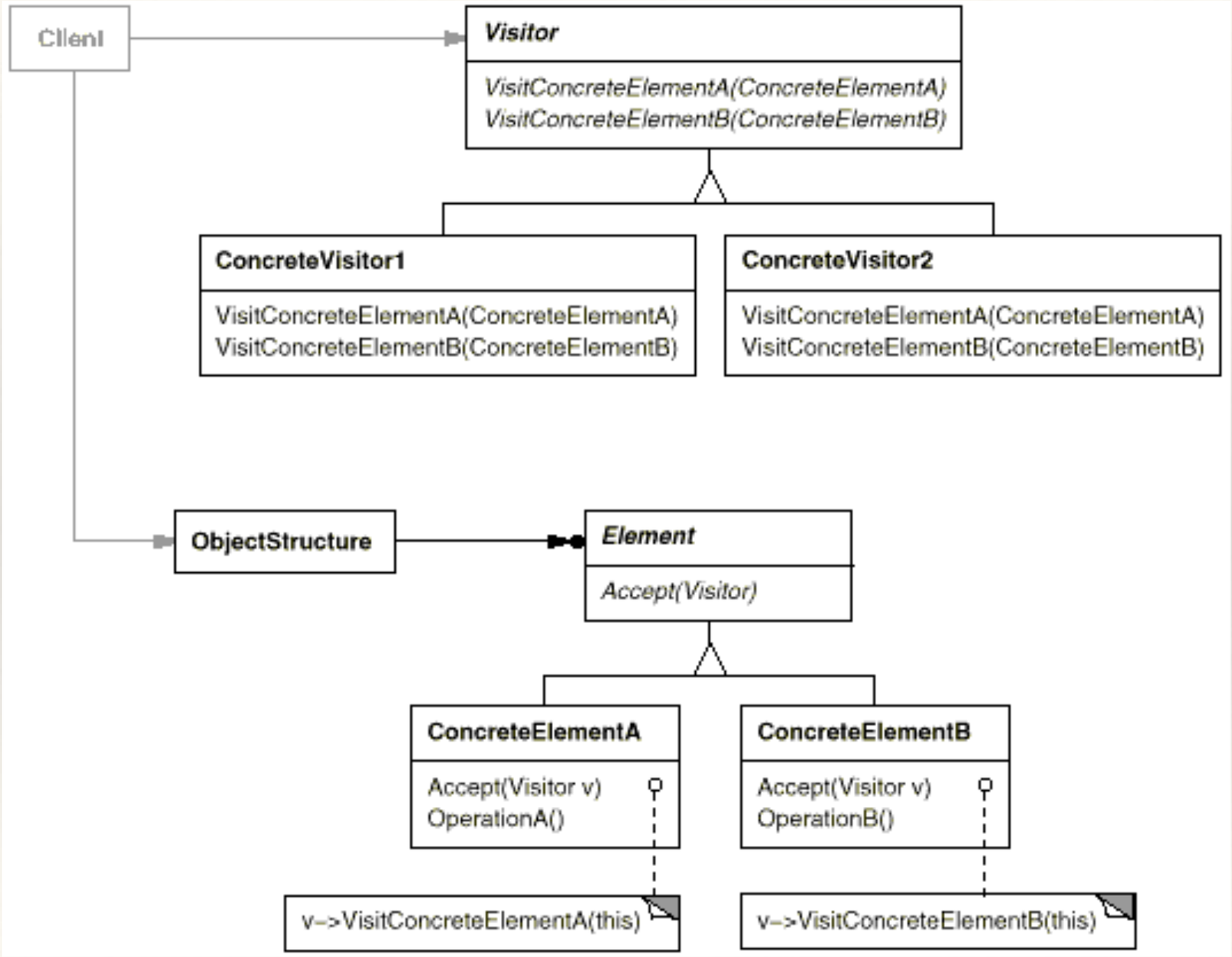
- The implementation of each of the algorithms is kept in a separate class referred to as a strategy.
- An object that uses a Strategy object is referred to as a context object.
- Changing the behavior of a Context object is a matter of changing its Strategy object to the one that implements the required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

Solution Using Visitor pattern

```
class Plus extends Expr {
        private Expr left, right;
        public Plus(Expr arg1, Expr arg2) {
                left = arg1;
                right = arg2;
        }
        public Expr getLeft() {
                return left;
        }
        public Expr getRight() {
                return right;
        }
        public void accept(Visitor v) {
                v.visit(this);
        }
}
```

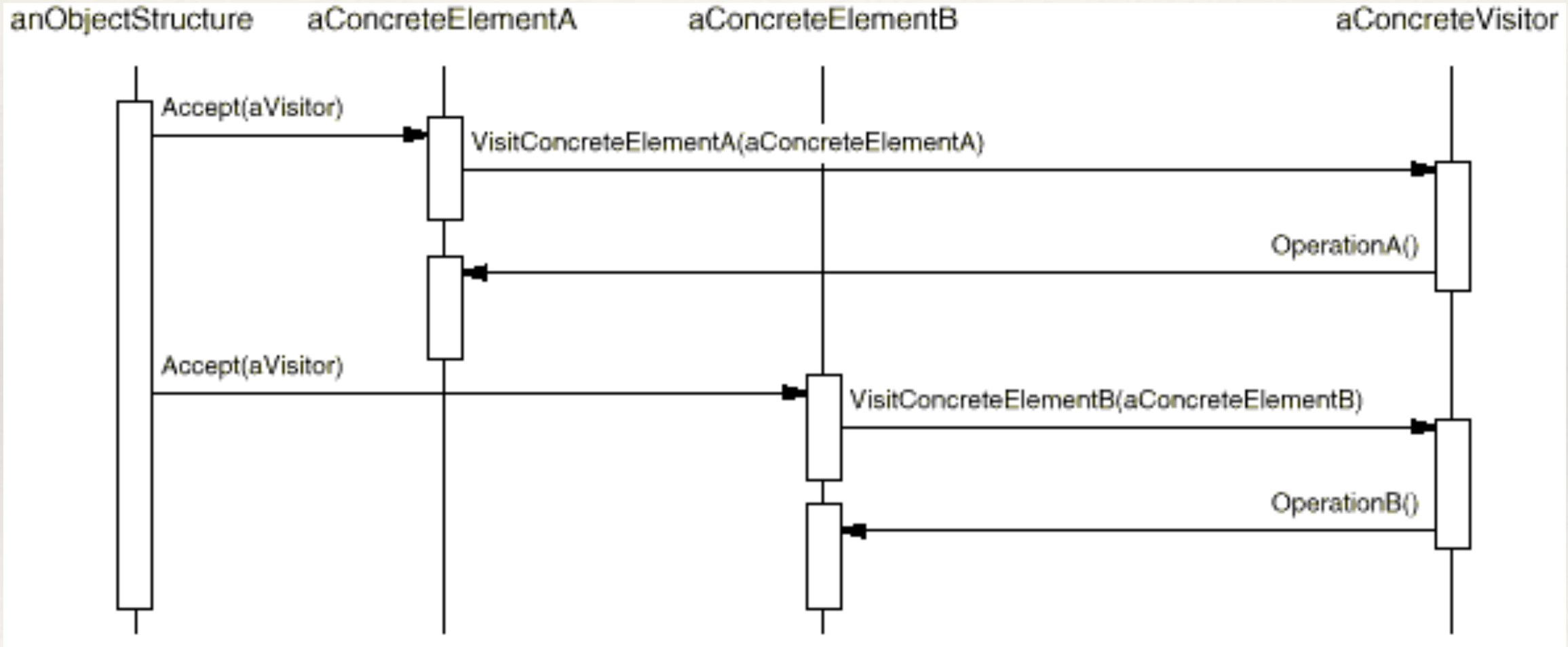
```
class DOTNETVisitor extends Visitor {
        public void visit(Constant arg) {
                System.out.println("ldarg " + arg.getVal());
        }
        public void visit(Plus plus) {
                genCode(plus.getLeft());
                genCode(plus.getRight());
                System.out.println("add");
        }
        public void visit(Sub sub) {
                genCode(sub.getLeft());
                genCode(sub.getRight());
                System.out.println("sub");
        }
        public void genCode(Expr expr) {
                expr.accept(this);
        }
}
```

Visitor Pattern: Structure



Source: "Design Patterns: Elements of Reusable Object-Oriented Software", Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Addison-Wesley,1994

Visitor Pattern: Call Sequence



Source: “Design Patterns: Elements of Reusable Object-Oriented Software”, Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Addison-Wesley,1994

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

How to get rid of conditional statements?

```
internal class Interpreter
    {
        private static readonly Stack<int> ExecutionStack = new Stack<int>();

        private static int Interpret(byte[] byteCodes)
        {
            var pc = 0;
            while (pc < byteCodes.Length)
                switch (byteCodes[pc++])
                {
                    case (byte) BYTECODE.ADD:
                        ExecutionStack.Push(ExecutionStack.Pop() + ExecutionStack.Pop());
                        break;
                    case (byte) BYTECODE.LDCI4S:
                        ExecutionStack.Push(byteCodes[pc++]);
                        break;
                }
            return ExecutionStack.Pop();
        }
  // rest of the code elided ...
}
```

Hands-on Exercise

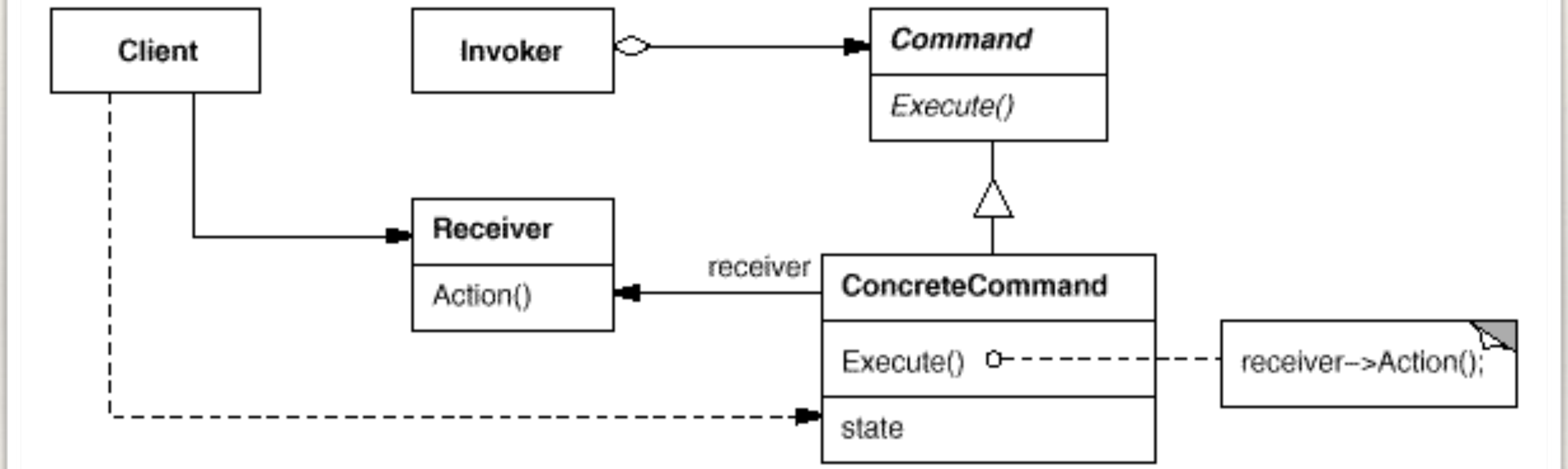
```
switch (byteCodes[pc++])
{
    case (byte) BYTECODE.ADD:
        ExecutionStack.Push(ExecutionStack.Pop() + ExecutionStack.Pop());
        break;
    case (byte) BYTECODE.LDCI4S:
        ExecutionStack.Push(byteCodes[pc++]);
        break;
}
```

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

Command Pattern: Solution

```
internal class Interpreter
    {
        private readonly Stack<int> _evalStack = new Stack<int>();

        public int Interpret(ByteCode[] byteCodes)
        {
            foreach (var byteCode in byteCodes) byteCode.Exec(_evalStack);
            return _evalStack.Pop();
        }
    }

    internal abstract class ByteCode
    {
        public abstract void Exec(Stack<int> evalStack);
    }

   internal class ADD : ByteCode
    {
        public override void Exec(Stack<int> evalStack)
        {
            var lval = evalStack.Pop();
            var rval = evalStack.Pop();
            evalStack.Push(rval + lval);
        }
    }
```

Command Pattern: Structure



Source: “Design Patterns: Elements of Reusable Object-Oriented Software”, Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Addison-Wesley,1994

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.

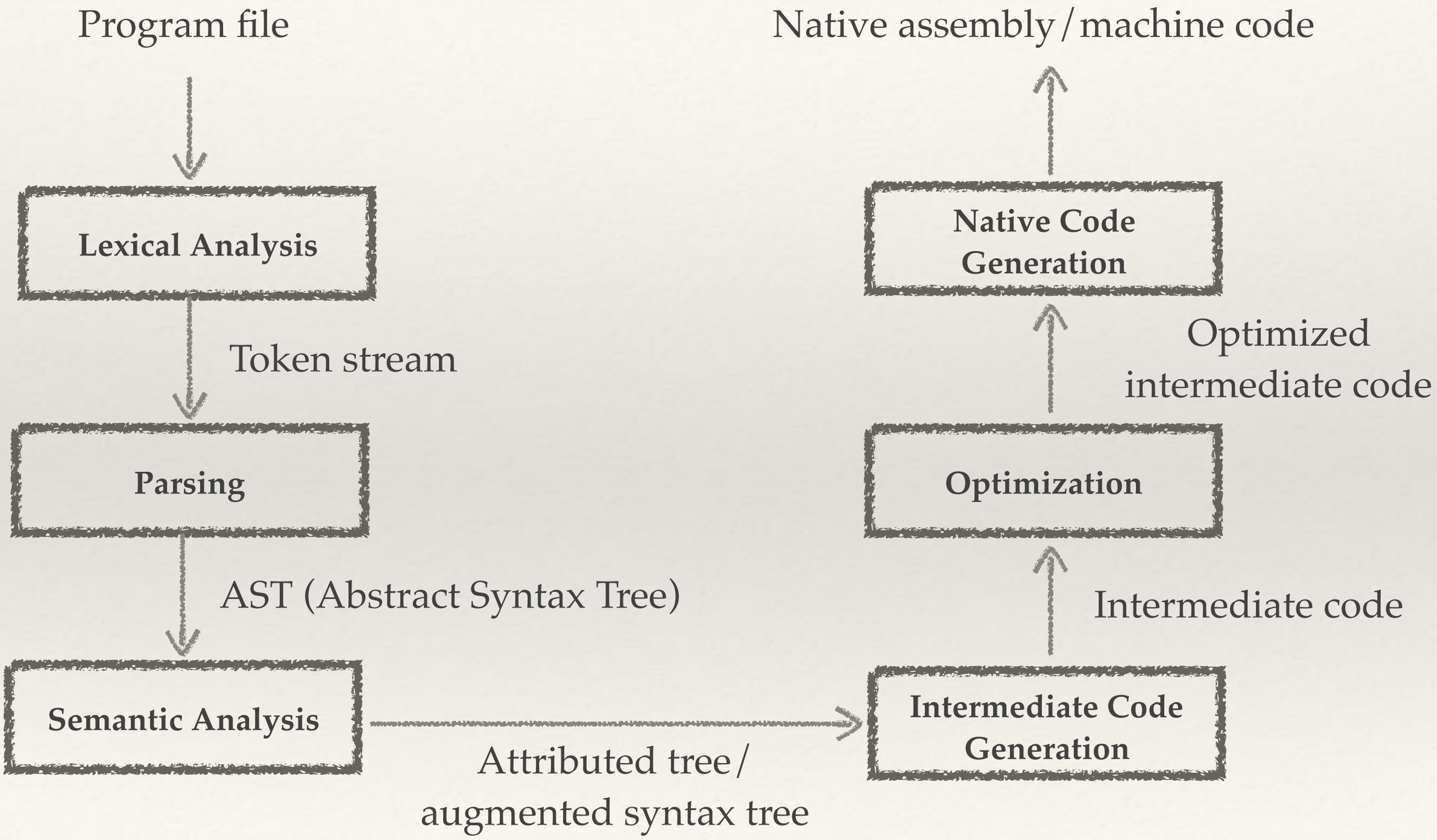
- Want to issue requests or execute commands without knowing anything about the operation being requested or the receiver of the request

- 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

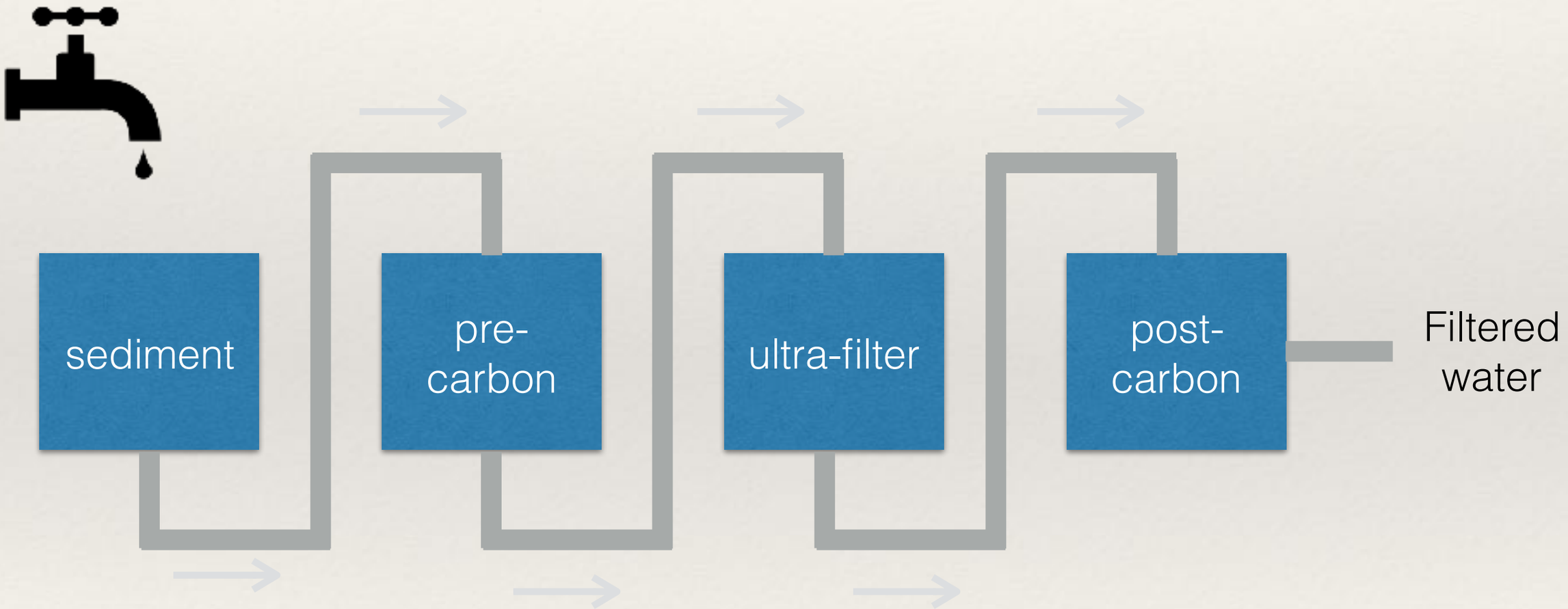
Patterns Discussed in this Case-Study

- ✓ Facade
- ✓ Composite
- ✓ Factory method
- ✓ Flyweight
- ✓ Builder
- ✓ Iterator pattern
- ✓ Strategy
- ✓ Visitor
- ✓ Command

BTW, What Architecture Style is it?

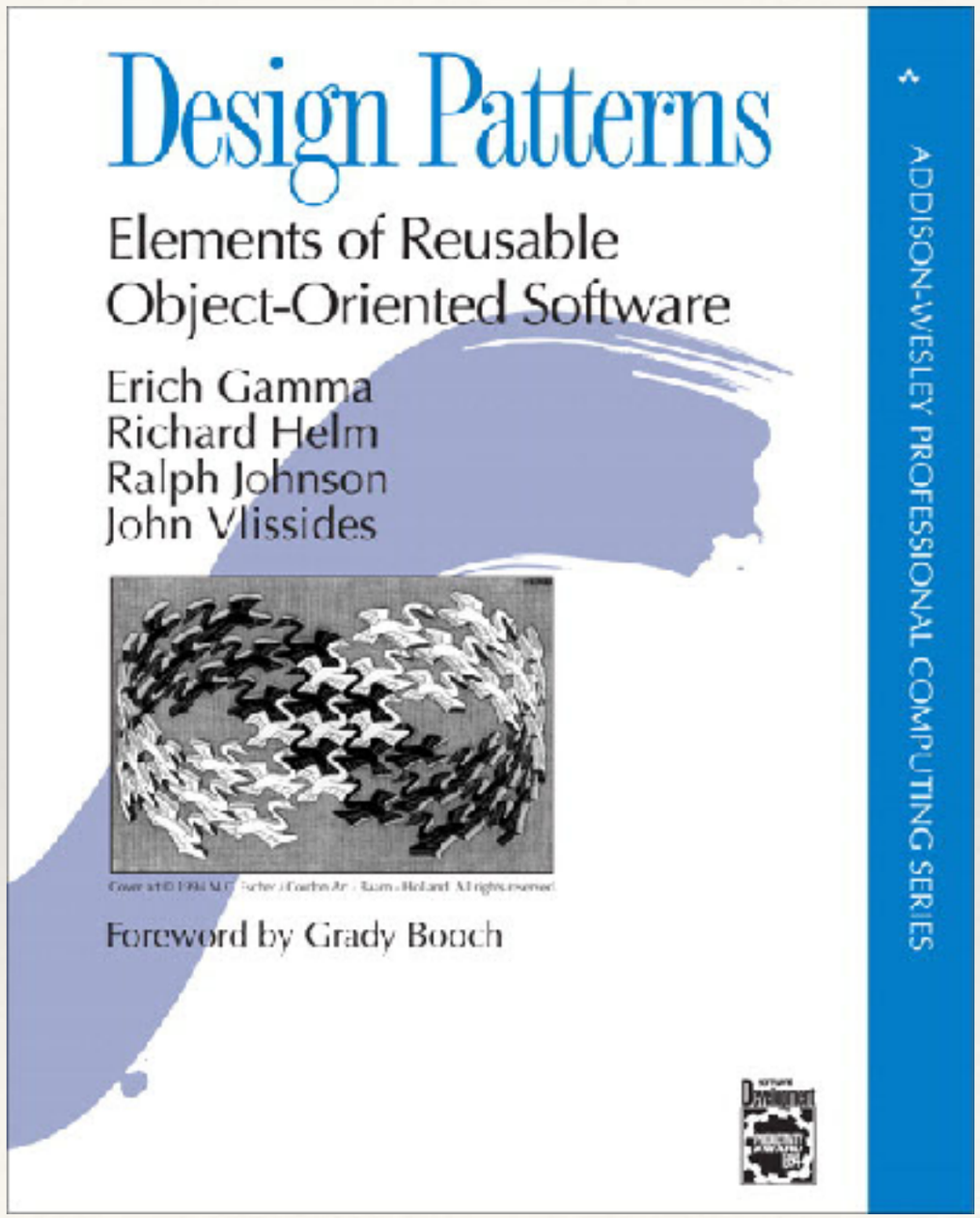


Its "pipe-and-filter" Style!

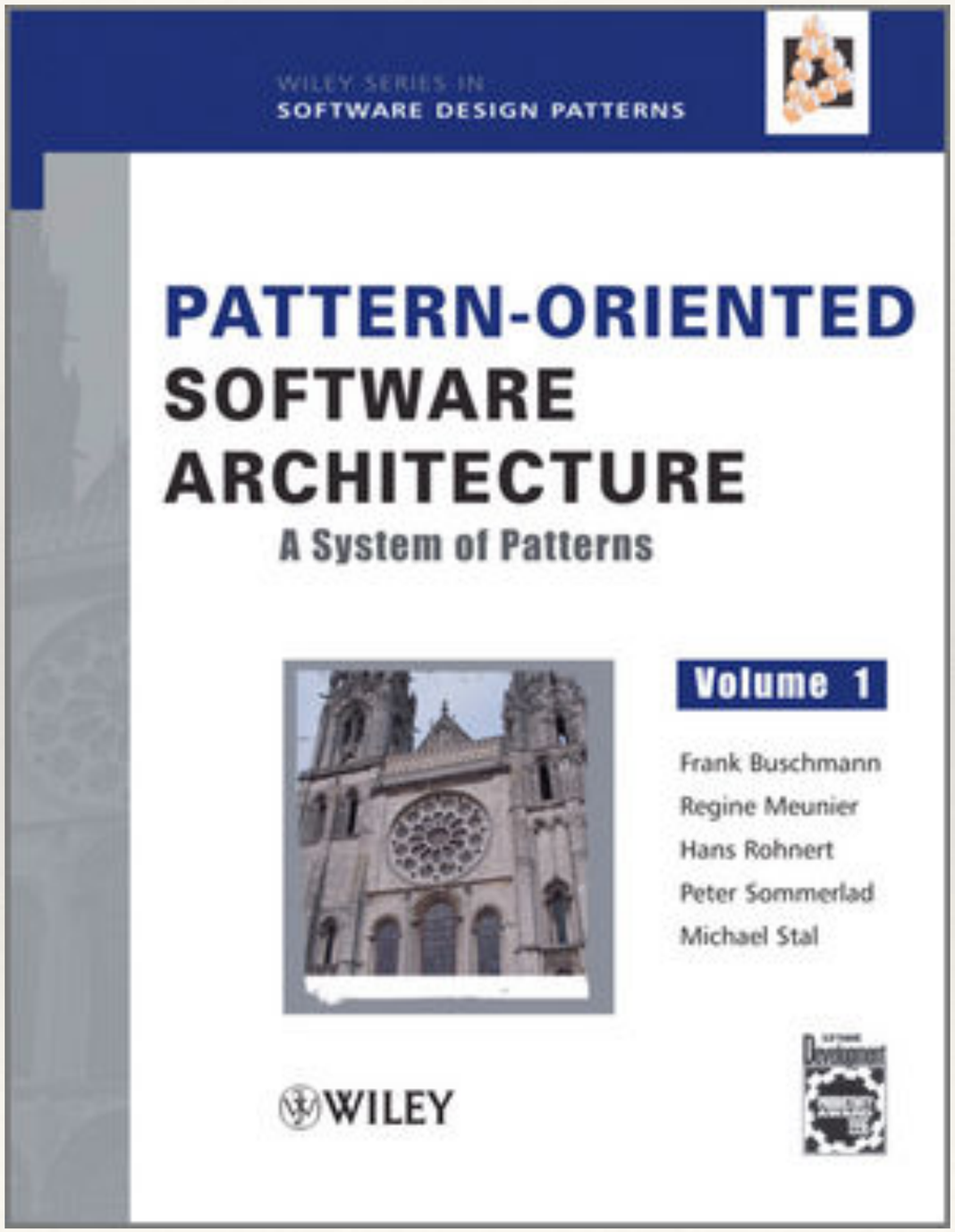


Real-world "pipes and filters"

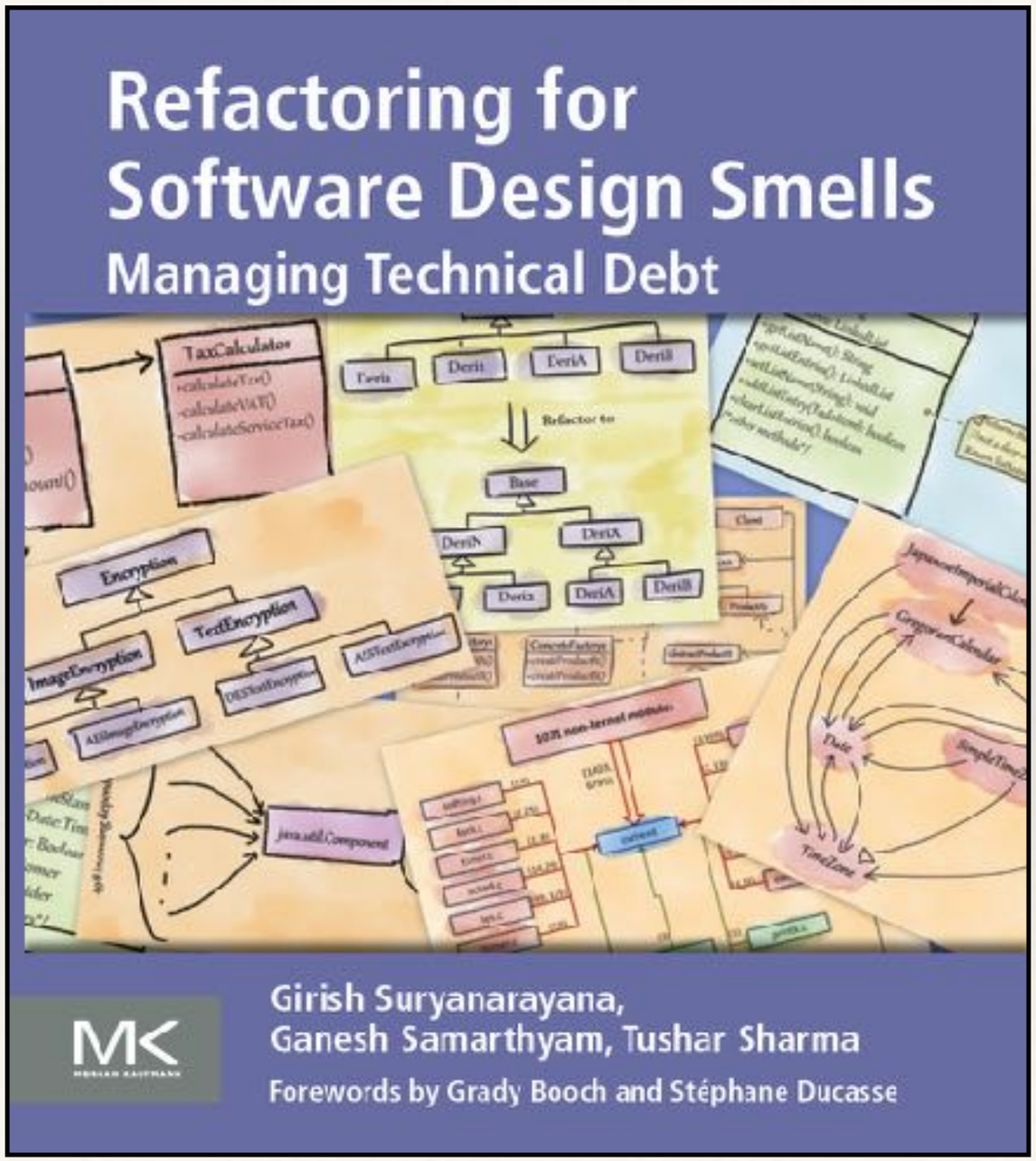
Classic Design Patterns Book

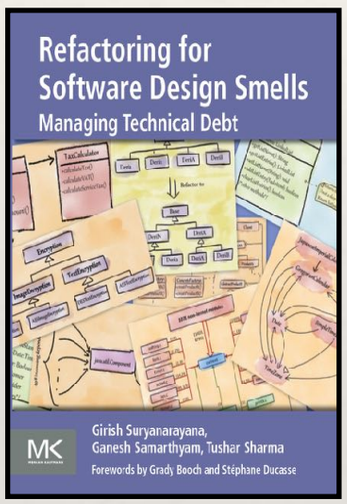


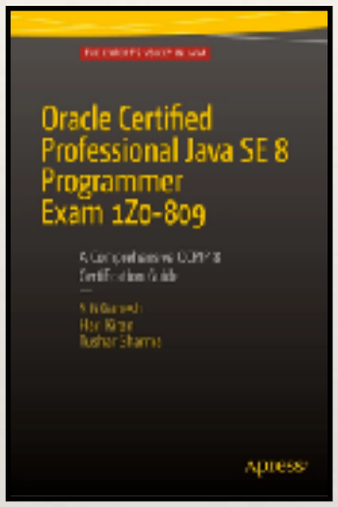
Classic Book on Architectural Patterns

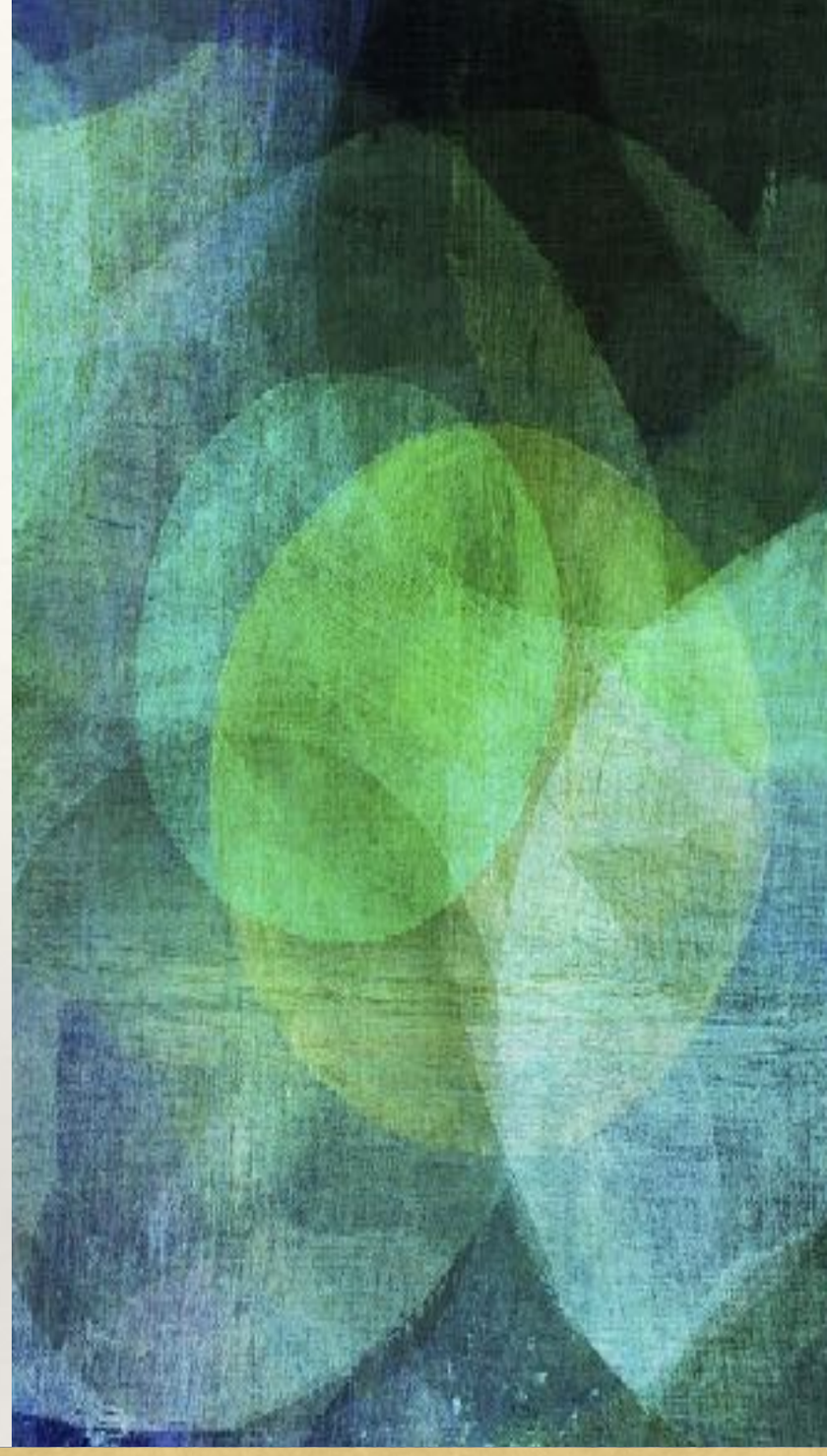


“Principle-based” Refactoring Approach









ganesh@codeops.tech

www.codeops.tech

+91 98801 64463

@GSamarthyam

slideshare.net/sgganesh

bit.ly/sgganesh

CodeOps