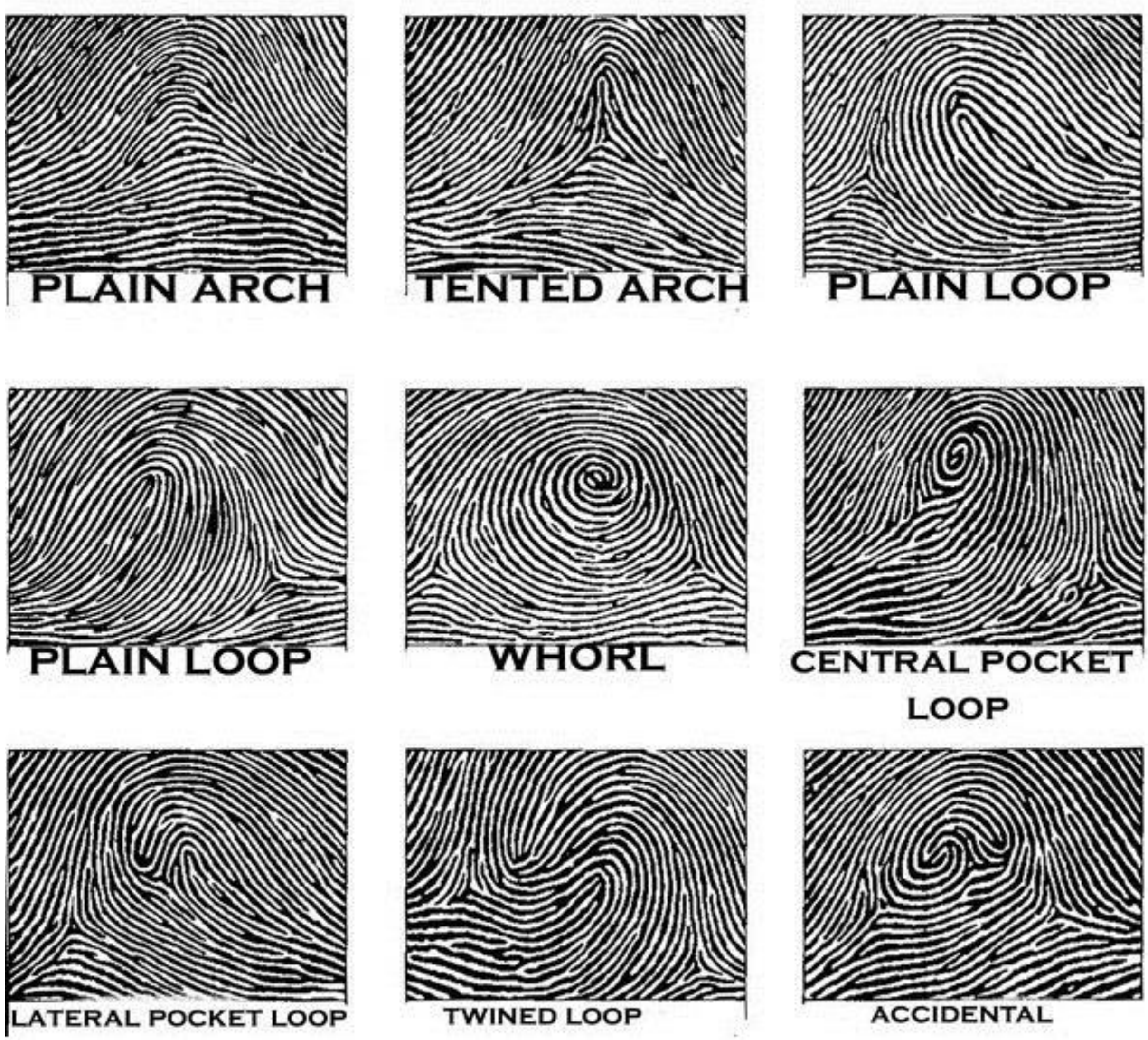
Design Patterns

Rikhard Goldenberg
SberTech JavaSchool, March 2018

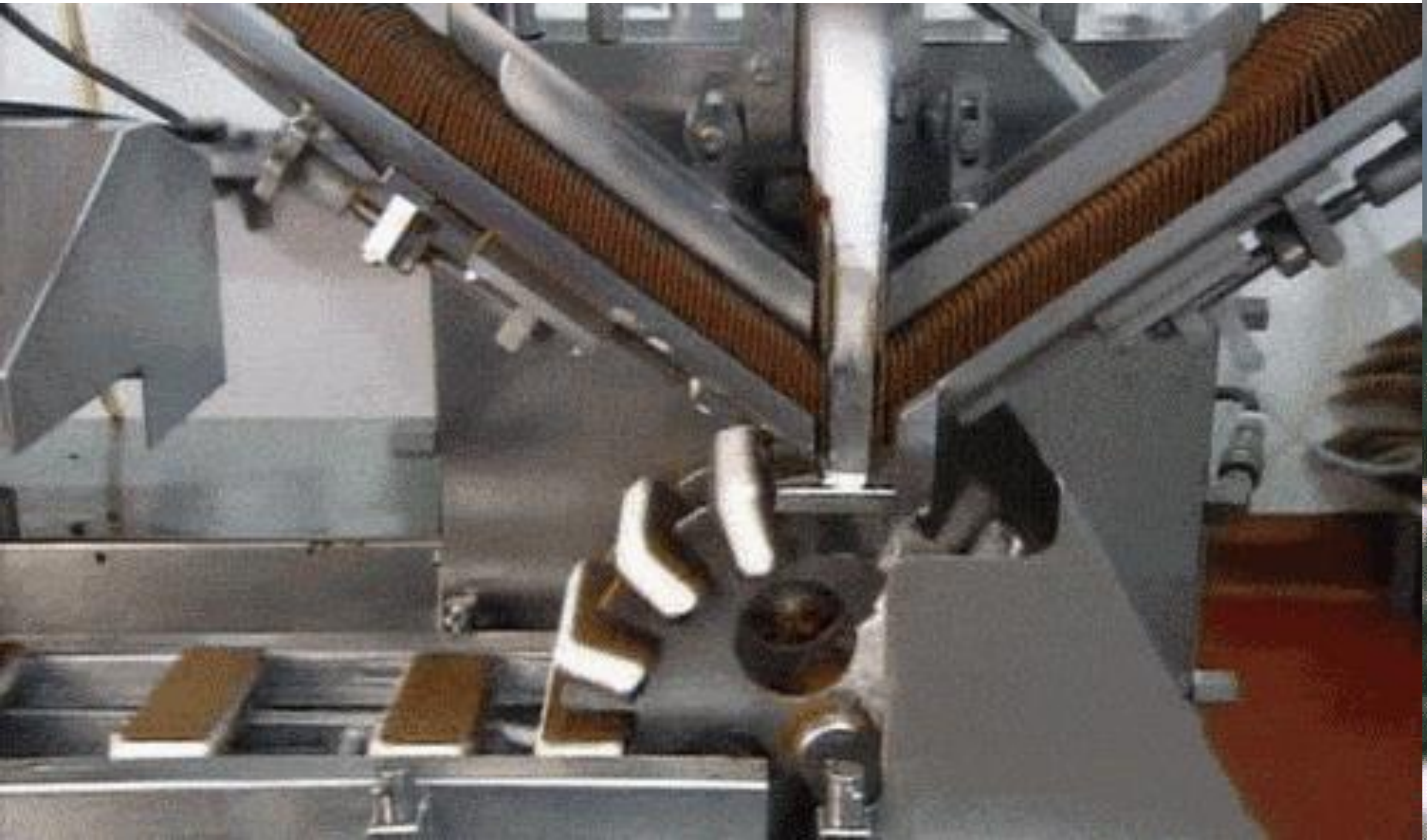
DESIGN PATTERN TYPES

- CREATIONAL
- STRUCTURAL
- BEHAVIORAL



CREATIONAL PATTERNS

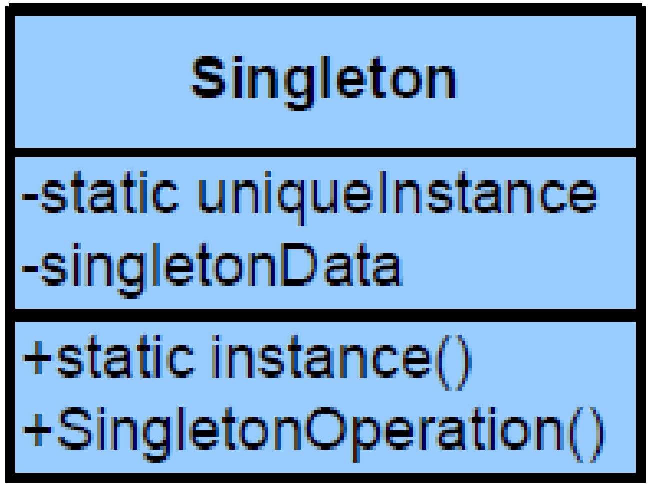
- **Increase flexibility in object creation**
- **Decouple interfaces from implementations**
- ***Facilitate change***



Singleton

Type: Creational

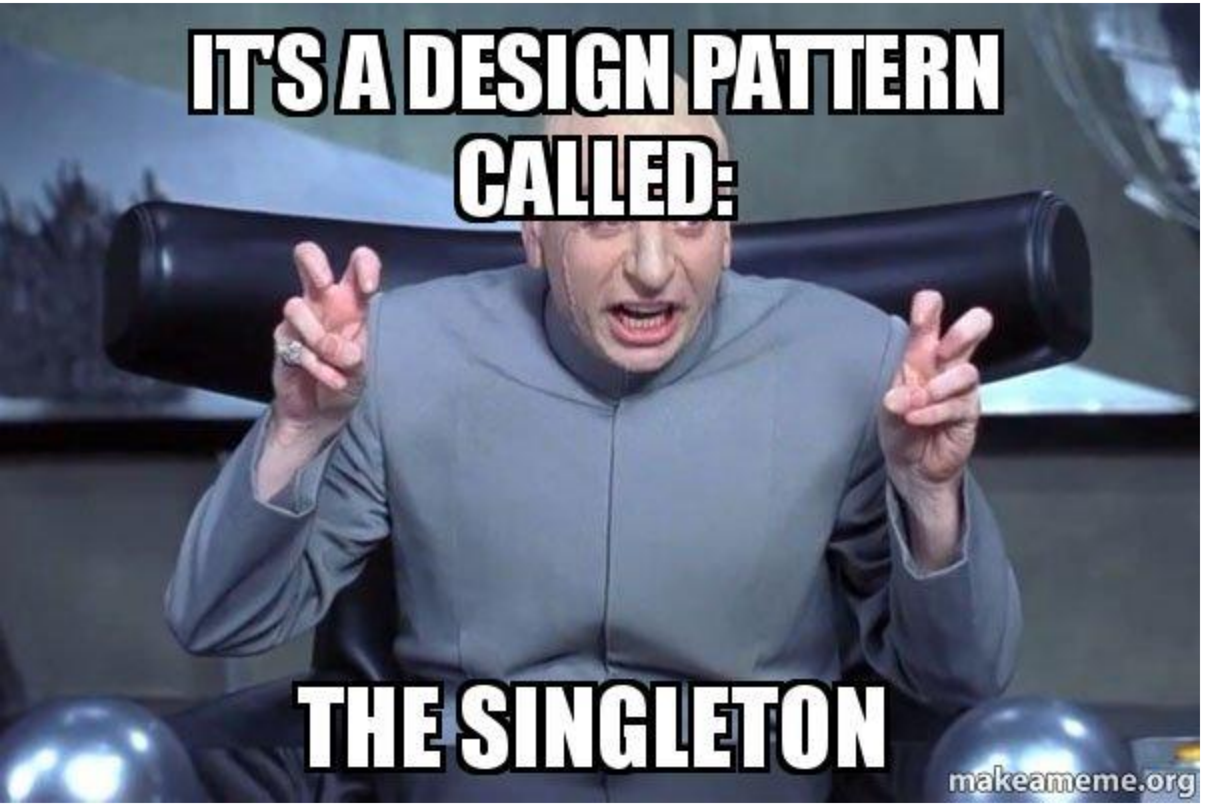
What it is:
Ensure a class only has one instance and provide a global point of access to it.



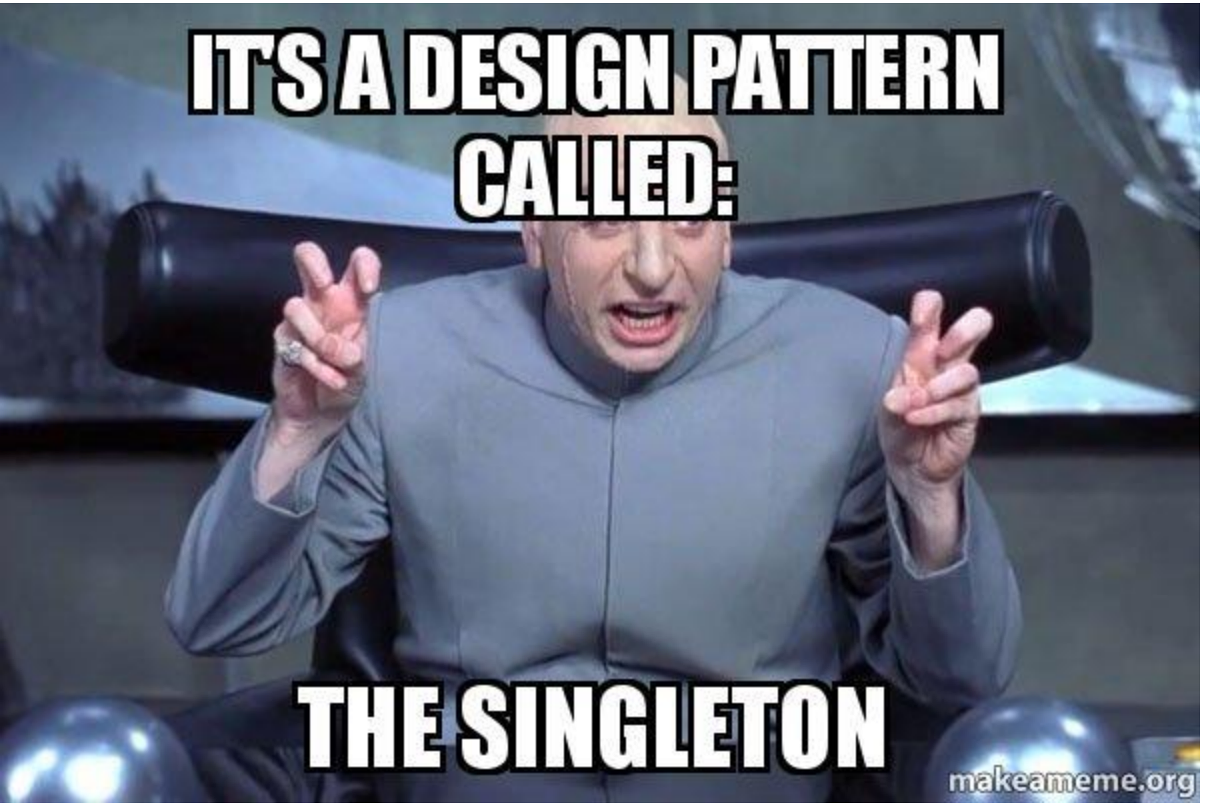
Singleton

-static uniqueInstance
-singletonData

+static instance()
+SingletonOperation()



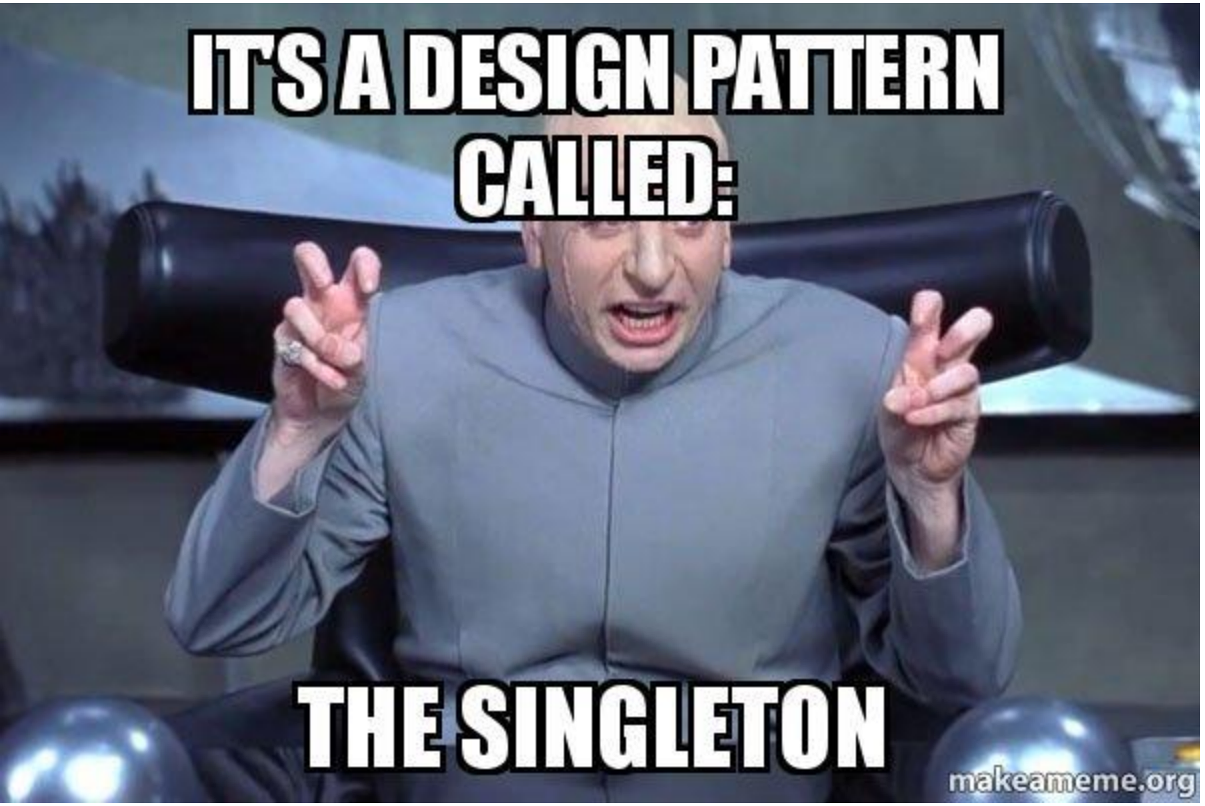
- class restricted to one instance
- provides a global point
  of access to the class
- possible lazy initialization



- class restricted to one instance
- provides a global point of access to the class
- possible lazy initialization

DIY singletons:

- *violate SRP*
- *complicate testing*
- *need maintenance*



- class restricted to one instance
- provides a global point of access to the class
- possible lazy initialization

prefer IoC + DI Singleton approach

DIY singletons:
- *violate SRP*
- *complicate testing*
- *need maintenance*

factory

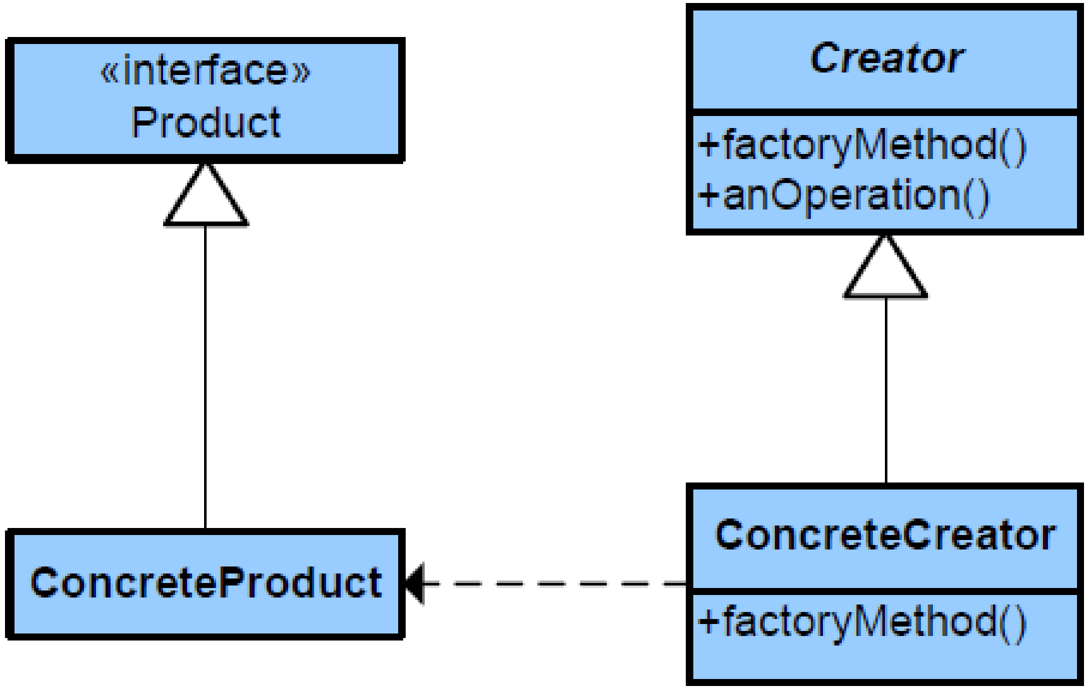
- create objects without exposing the instantiation logic
- refer to the newly created object by an “interface”

FACTORY

Factory Method

Type: Creational

What it is:
Define an interface for creating an object, but let subclasses decide which class to instantiate. Lets a class defer instantiation to subclasses.



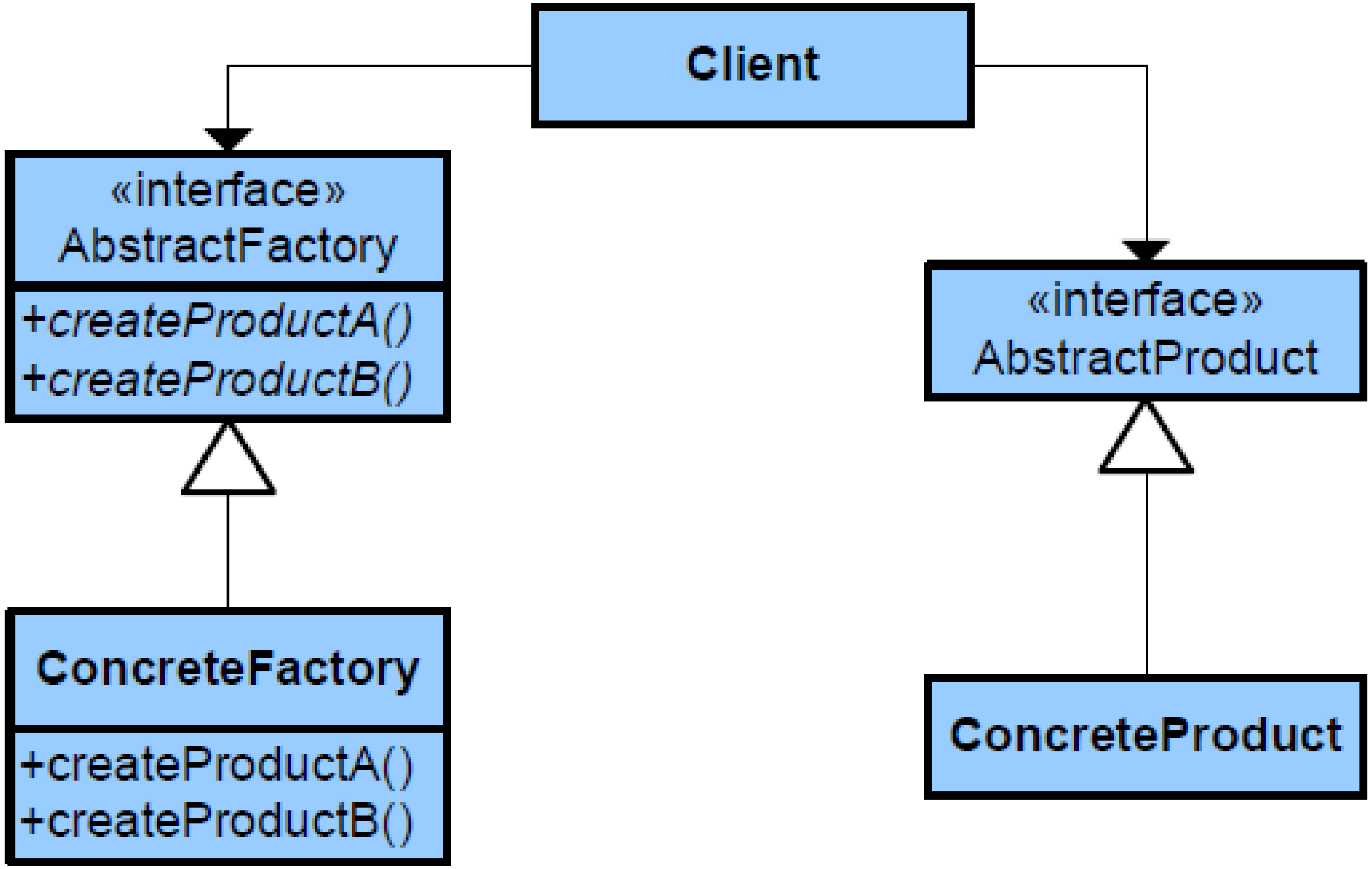
- define an interface for creating objects, but let subclasses decide which class to instantiate
- refer to the newly created object by an “interface”

FACTORY METHOD

Abstract Factory

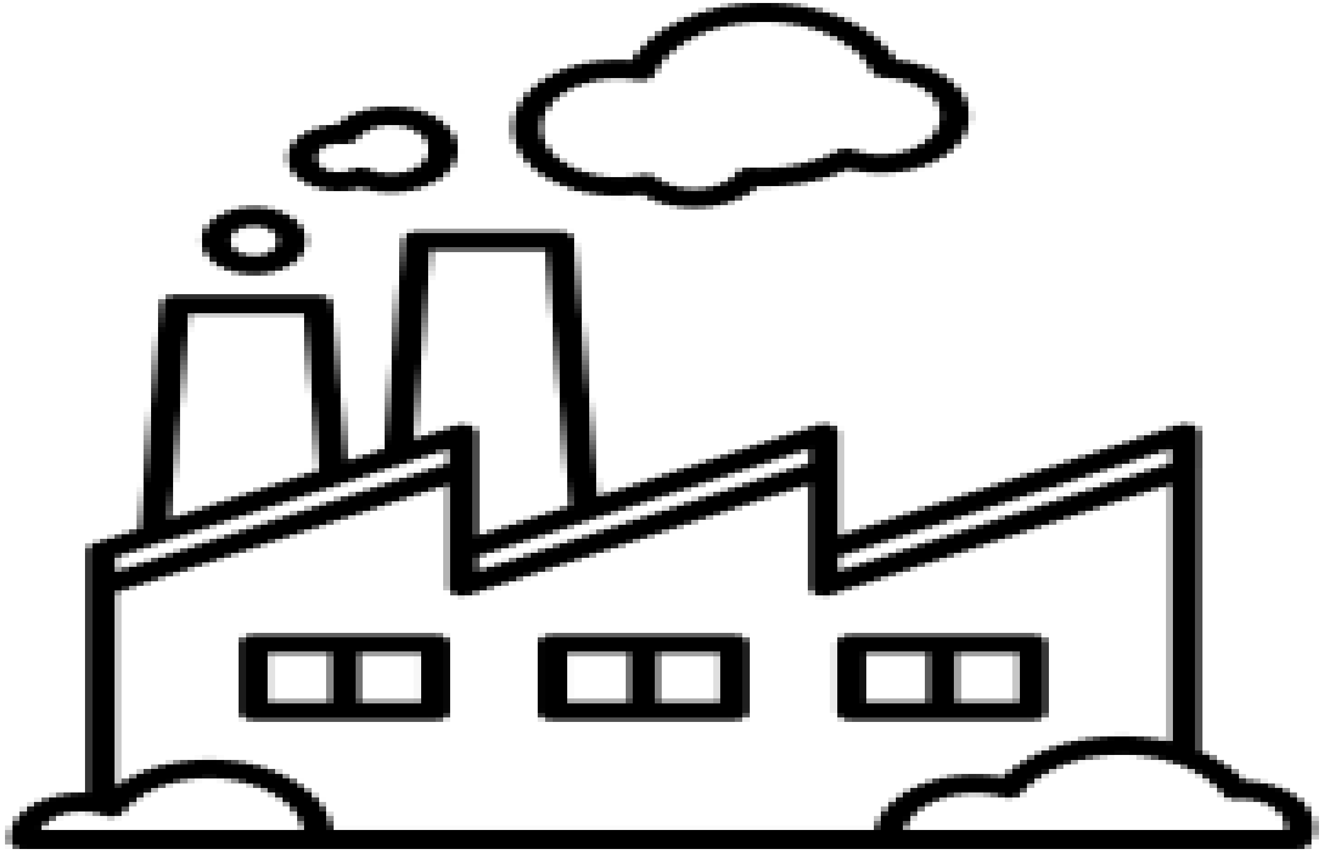
Type: Creational

What it is:
Provides an interface for creating families of related or dependent objects without specifying their concrete class.



- offer the interface for creating a family of related objects, without explicitly specifying their classes.

ABSTRACT FACTORY



PROS & CONS

- Factories facilitate loose coupling, hiding concrete classes from the application.

  They can extend the family of products with minor changes in application code.

- Factories provide customization hooks.

  If a factory is used to create a family of objects,
  the customized objects can easily replace the original objects.

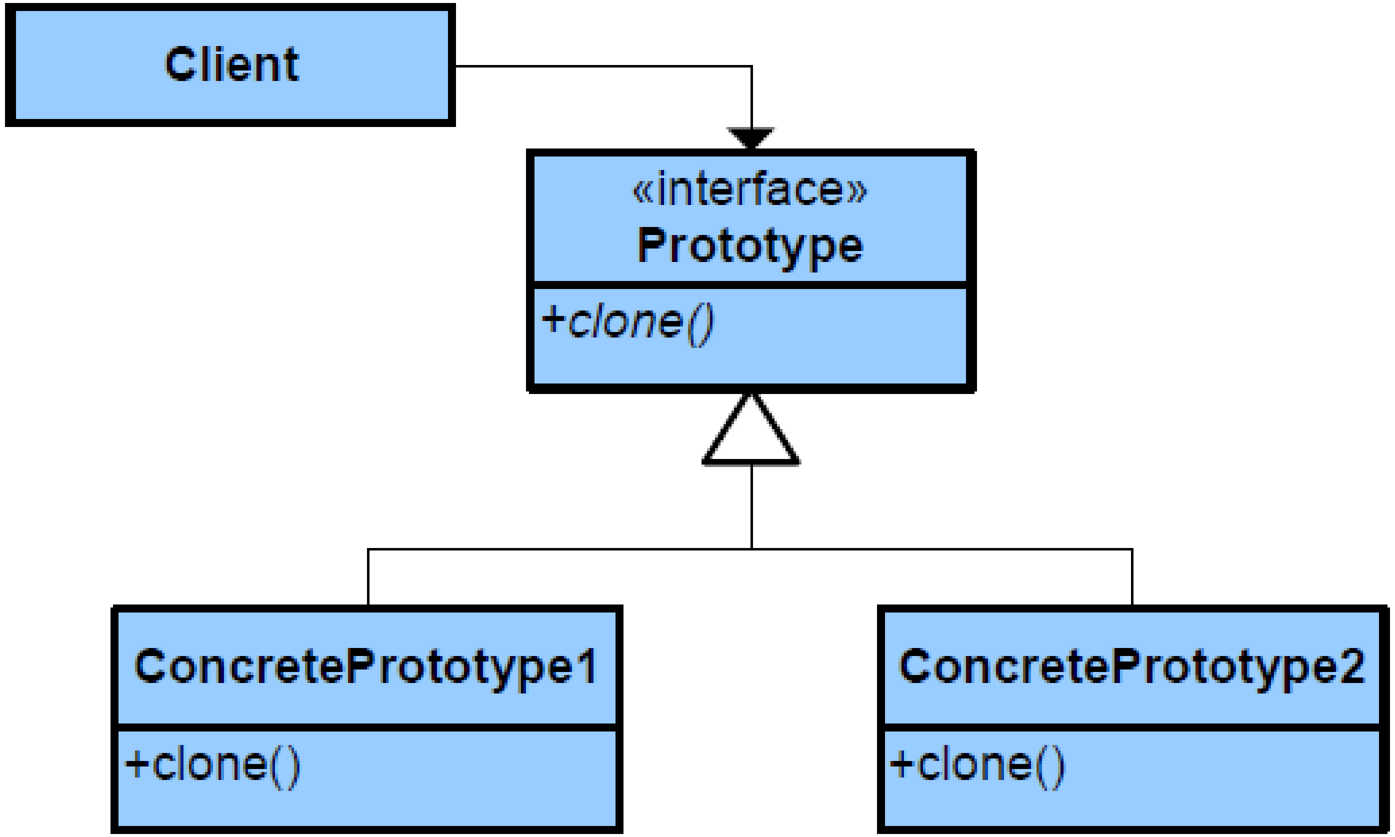
- Factories have to be used for a family of objects

  - common base class or interface needed.

Prototype

Type: Creational

What it is:
Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.



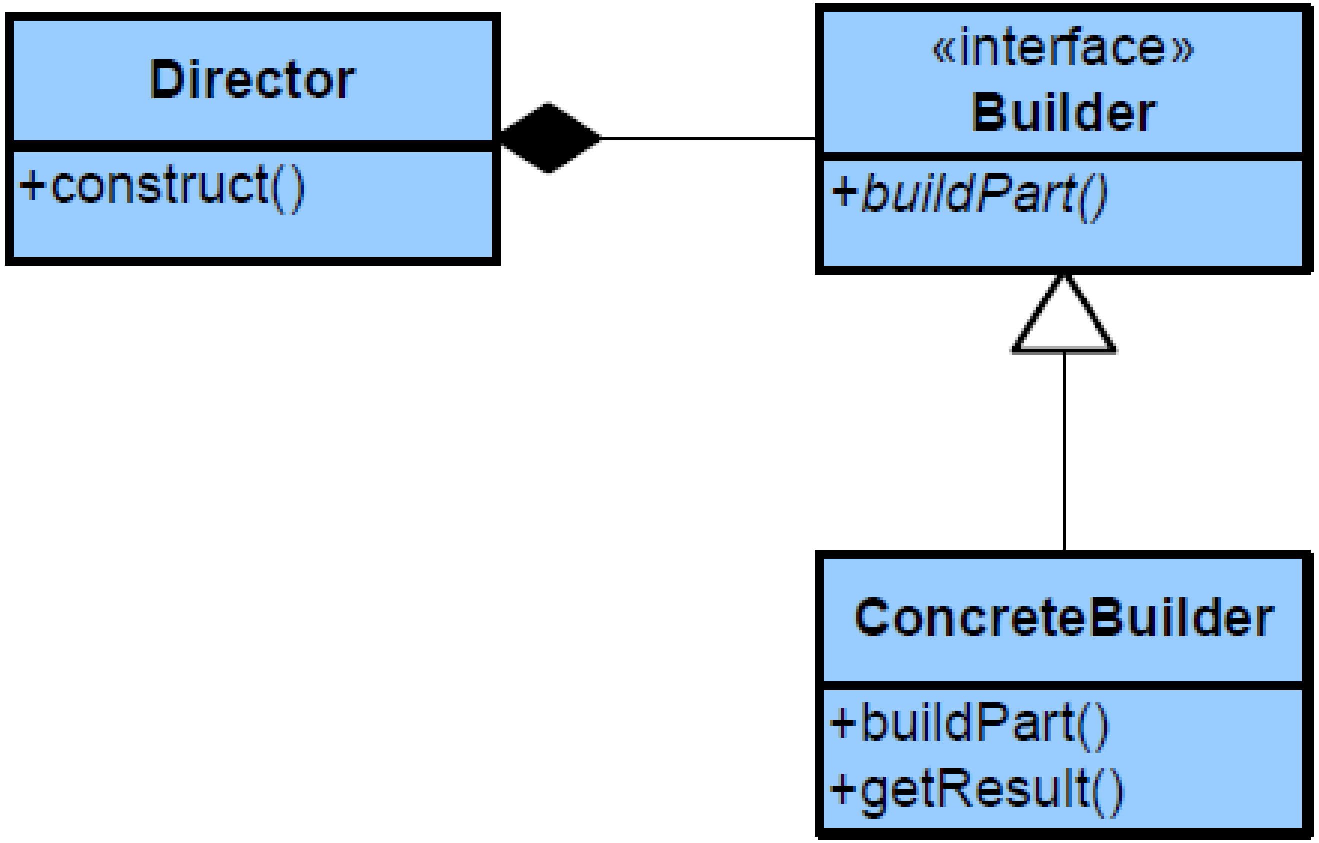
- specify the kind of objects to create using a prototypical instance
- create new objects by copying this prototype

PROTOTYPE

Builder

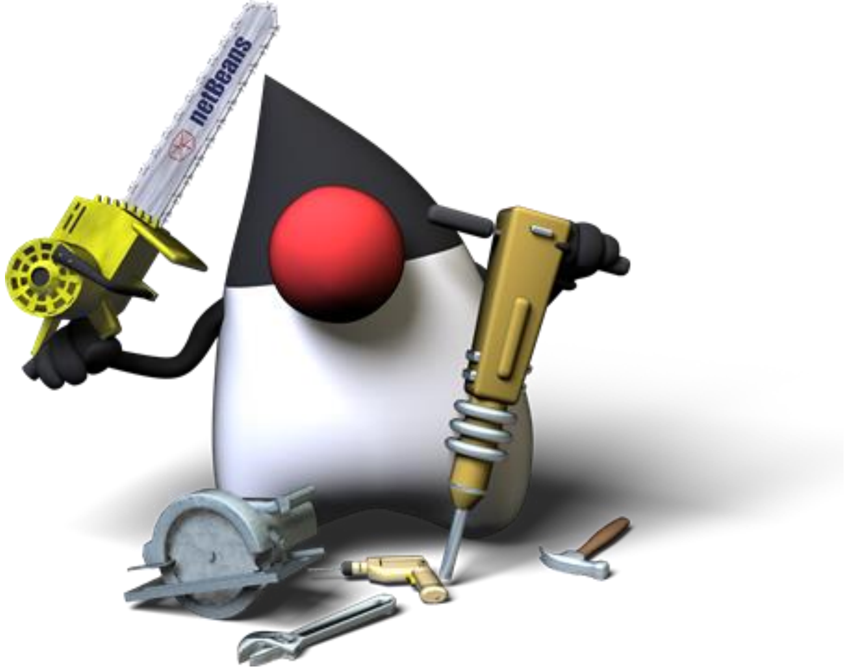
Type: Creational

What it is:
Separate the construction of a complex object from its representing so that the same construction process can create different representations.



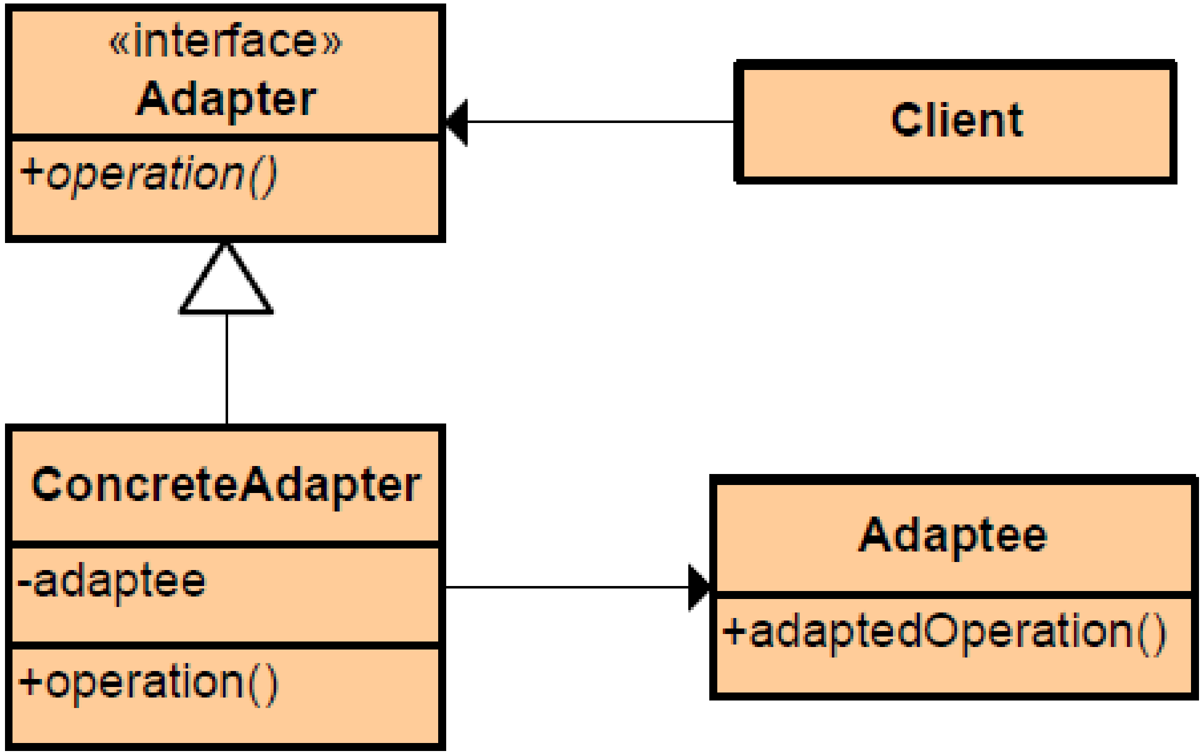
- separate the construction of a complex object from its representation
- allow same construction process to create different representations
- parse a complex representation, create one of several targets.

BUILDER



STRUCTURAL PATTERNS

- **Help establish relationships between objects**
- **Reduce client interface complexity**
- ***Facilitate change***



Adapter

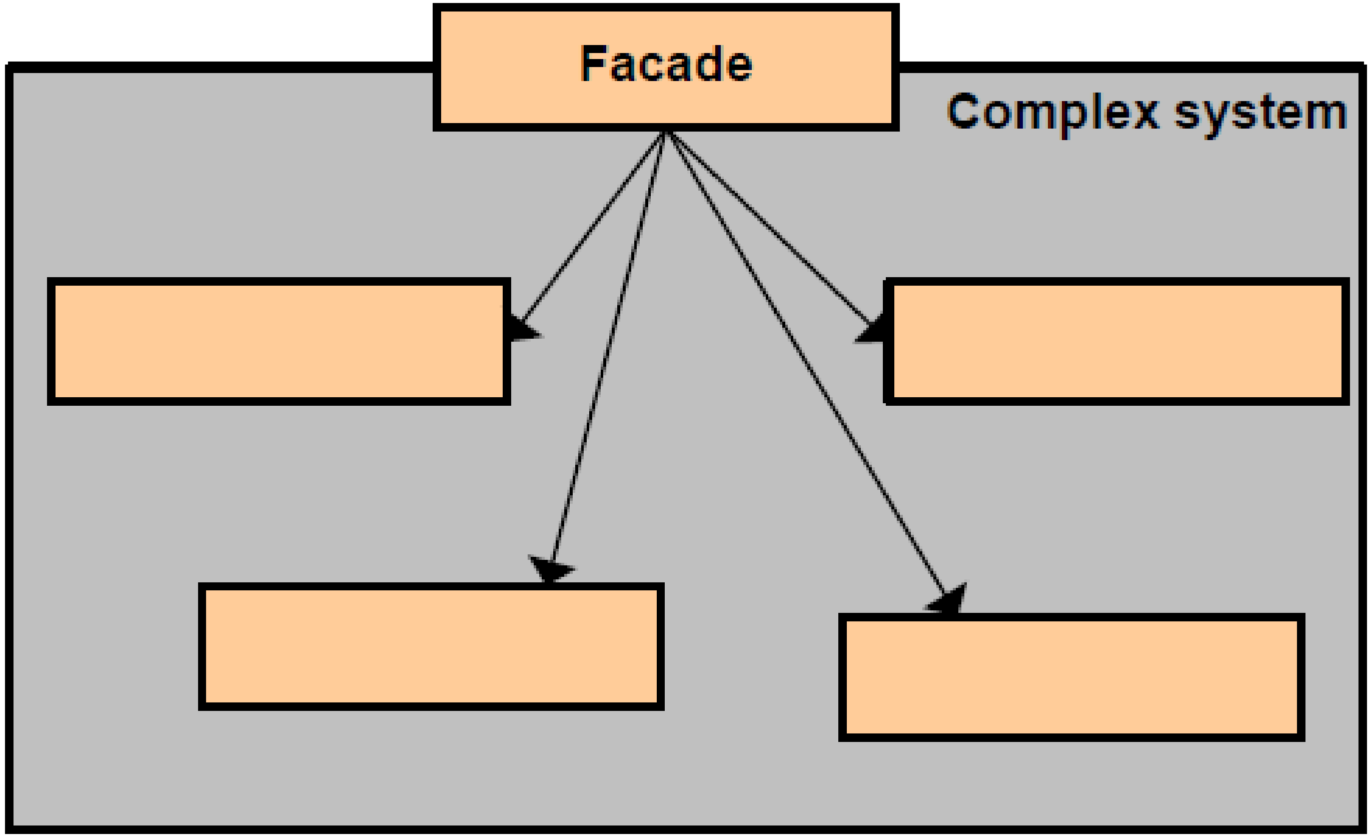
Type: Structural

What it is:
Convert the interface of a class into another interface clients expect. Lets classes work together that couldn't otherwise because of incompatible interfaces.

adapter

- convert some interface into another interface clients expect
- wrap an existing class with a new interface
- match an old component to a new system

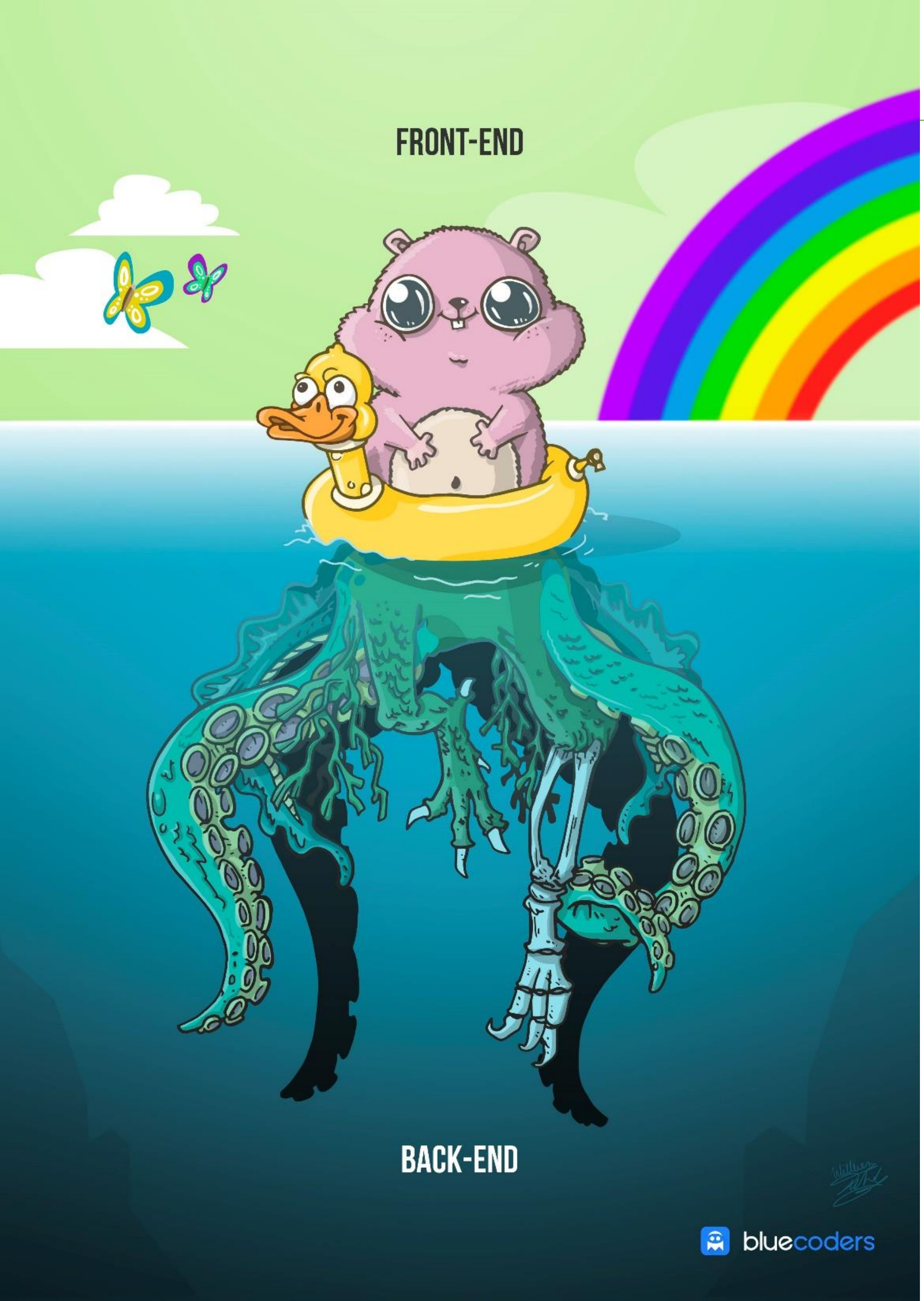
ADAPTER



Facade

Type: Structural

What it is:
Provide a unified interface to a set of interfaces in a subsystem. Defines a high-level interface that makes the subsystem easier to use.



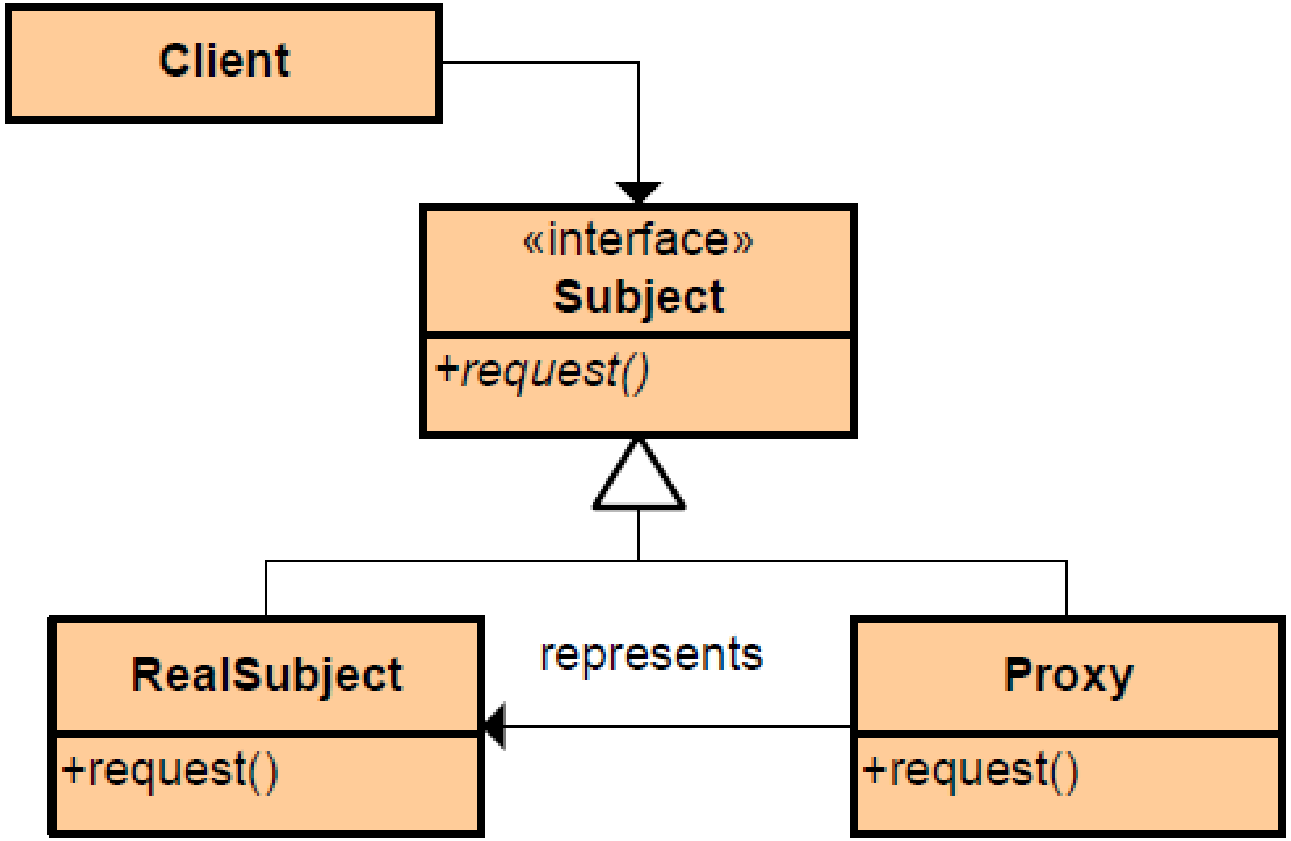
- provide a unified interface
  to a set of interfaces in a subsystem
- define a higher-level interface
  that makes the subsystem easier to use

FACADE

Proxy

Type: Structural

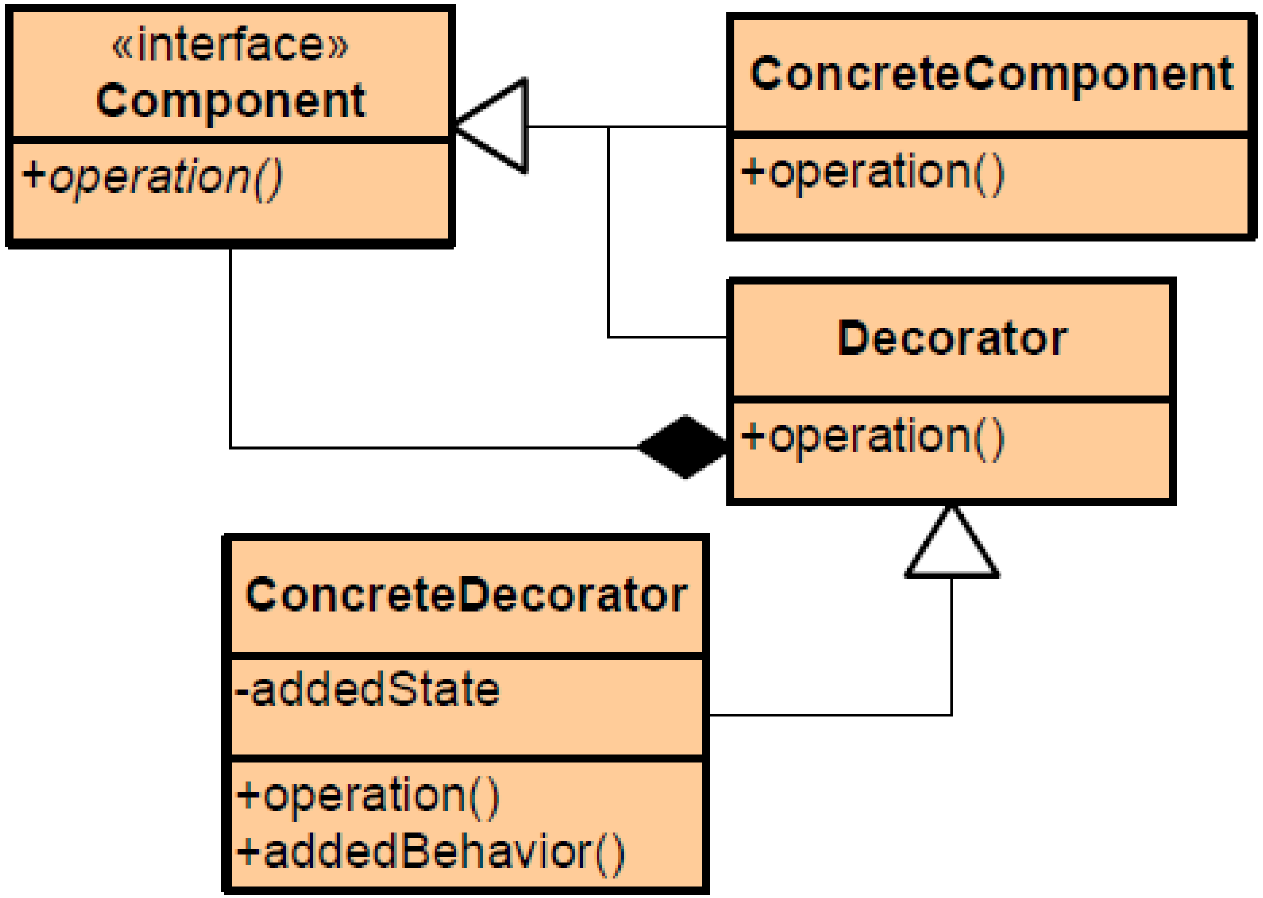
What it is:
Provide a surrogate or placeholder for another object to control access to it.



proxy

- provide a surrogate for another object to control access to it
- use an extra level of indirection to support controlled access
- add delegation to protect the real component from complexity

PROXY



Decorator

Type: Structural

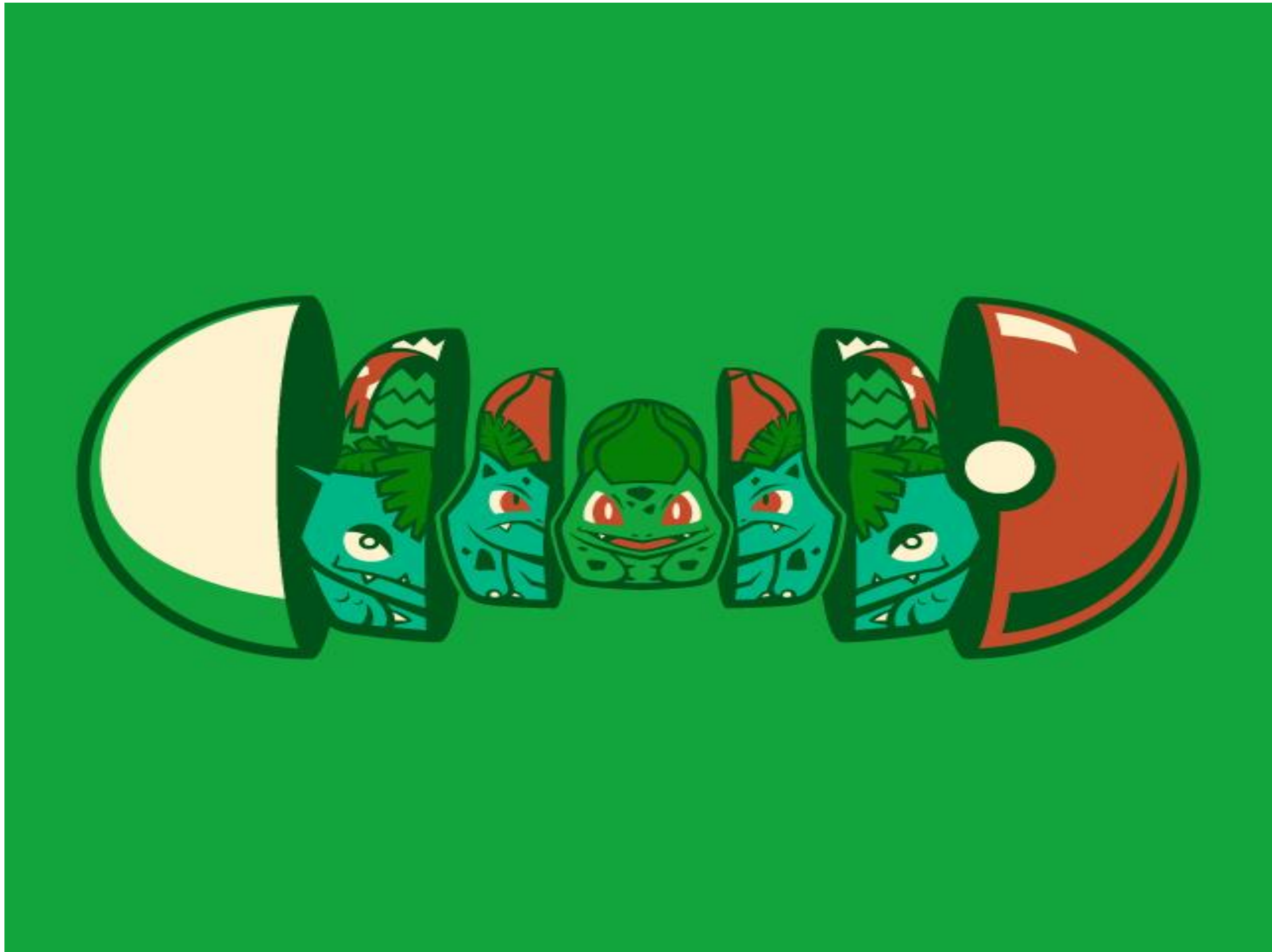
What it is:
Attach additional responsibilities to an object dynamically. Provide a flexible alternative to sub-classing for extending functionality.

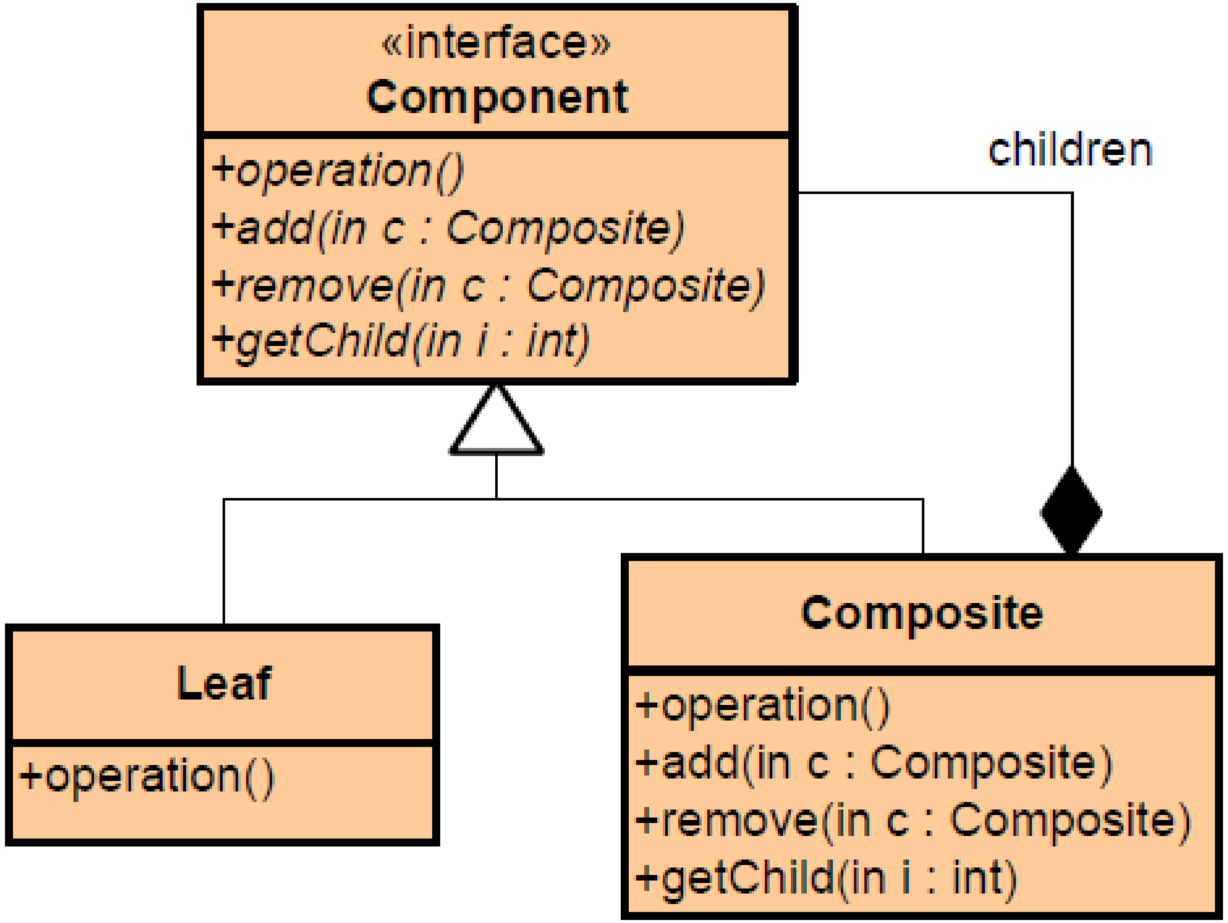
decorator

- **Add responsibilities to an object dynamically - alternative to subclassing**
- **Client-specified enhancement of a core object by recursively wrapping it**

"Wrapping a gift, putting it in a box, and wrapping the box"

DECORATOR





«interface»
Component

+operation()
+add(in c : Composite)
+remove(in c : Composite)
+getChild(in i : int)

children

Leaf

+operation()

Composite

+operation()
+add(in c : Composite)
+remove(in c : Composite)
+getChild(in i : int)

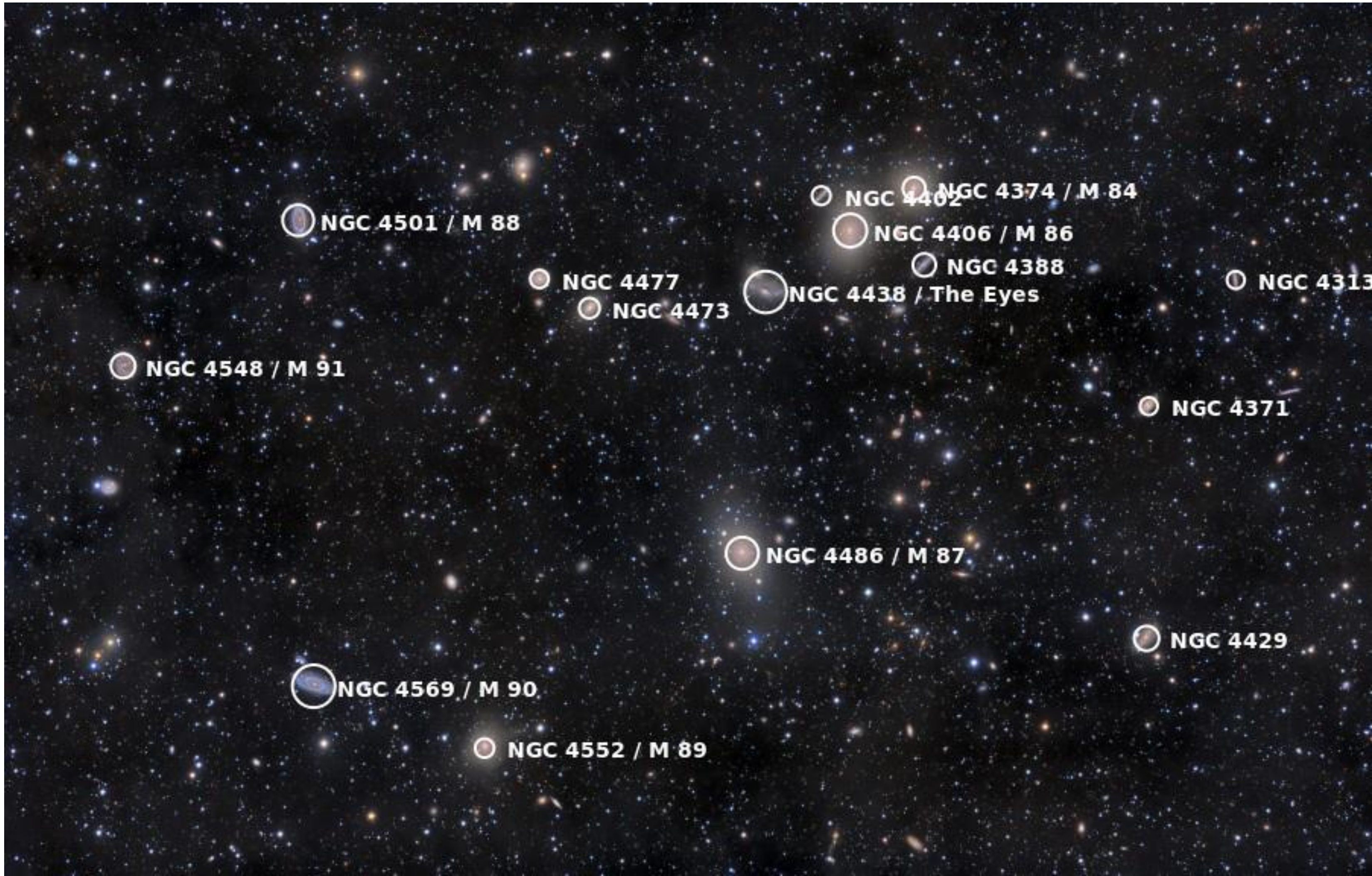
Composite

Type: Structural

What it is:
Compose objects into tree structures to represent part-whole hierarchies. Lets clients treat individual objects and compositions of objects uniformly.

- compose objects into tree structures to represent *part-whole* hierarchies
- treat individual objects and compositions of objects uniformly

COMPOSITE



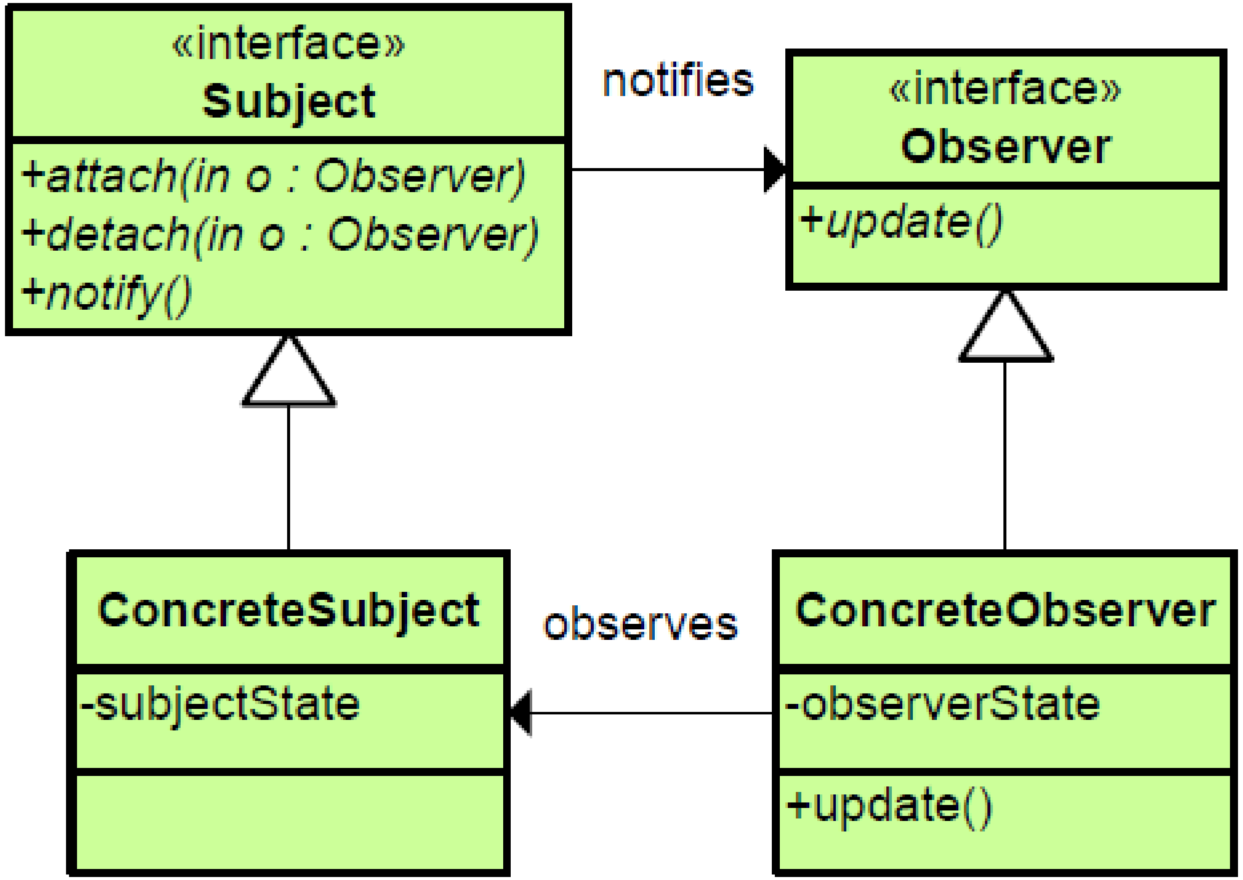
BEHAVIORAL PATTERNS

- identify common communication patterns between objects
- increase flexibility in carrying out this communication.

Observer

Type: Behavioral

What it is:
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.



- defines a one-to-many dependency between objects
- when one object changes state, all its dependents are notified and updated

OBSERVER

```
class TaskScheduler {

    public void schedule() {
        planSchedule();
    }

    private Result planSchedule() {
        //do work and return result
    }
}
```

OBSERVER

```
class TaskScheduler {
    public void schedule() {
        Result result = planSchedule();
        sendEmail(result);
    }

    private Result planSchedule() {
        //do work and return result
    }

    private void sendEmail(Result result) {
        //send email
    }
}
```

OBSERVER

```
class TaskScheduler {
    public void schedule() {
        Result result = planSchedule();
        sendEmail(result);
    }

    private Result planSchedule() {
        //do work and return result
    }

    private void sendEmail(Result result) {
        //send email
    }
}
```

OBSERVER

```
class TaskScheduler {
    private EmailSender emailSender;

    public TaskScheduler(EmailSender sender) {…}

    public void schedule() {
        Result result = planSchedule();
        emailSender.sendEmail(result);
    }
}
```

OBSERVER

observer

```
interface Listener {
     void onEvent(Result result);
}

class EmailSenderListener implements Listener {
    @Override
    void onEvent(Result result) {
        sendEmail(result);
    }
}

interface Observable {
     void attachListener(Listener listener);
     void detachListener(Listener listener);
}
```

OBSERVER

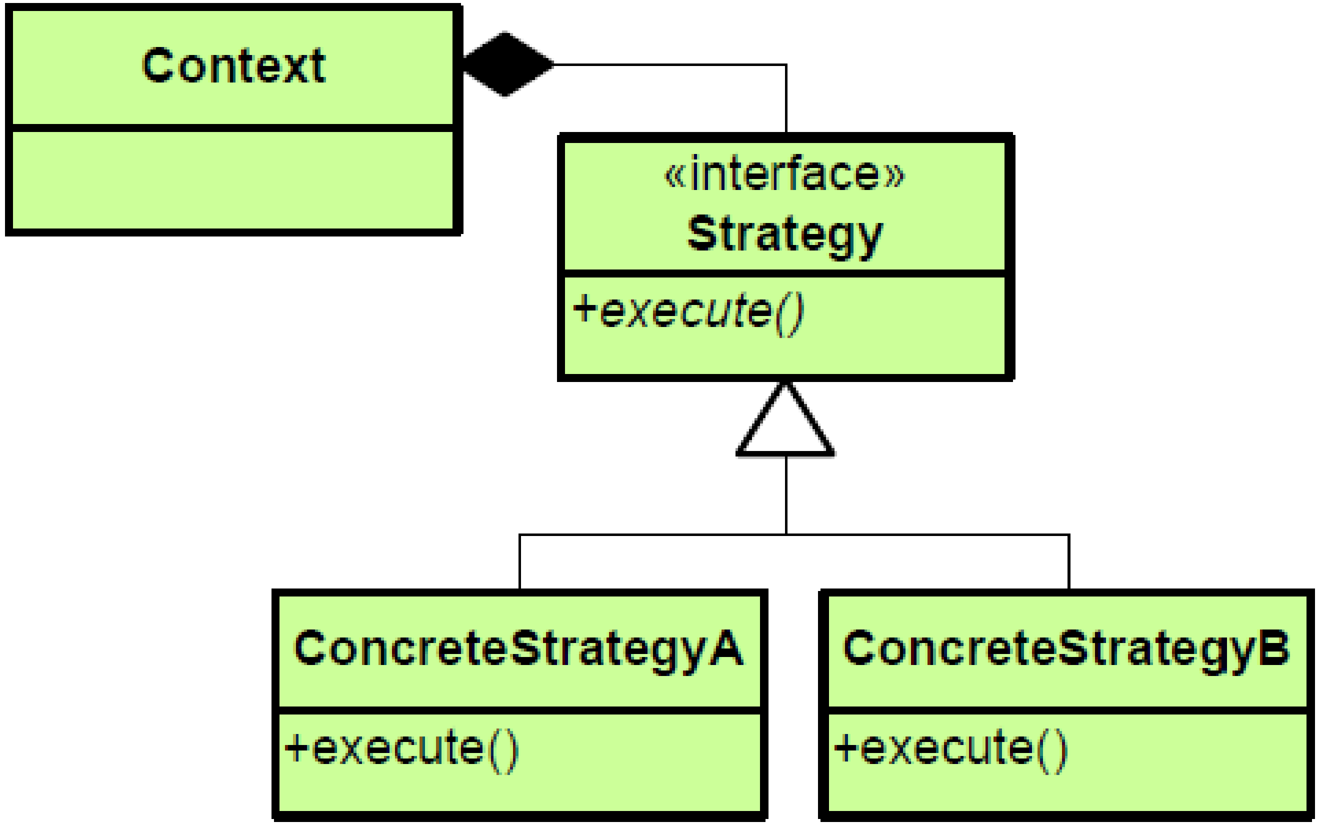
```
class ObservableTaskScheduler implements Observable  {
    private List<Listener> listeners = new ArrayList<>();

    public void schedule() {
        WorkResult result = doWork();
        notifyObservers(result);
    }

    private void notifyObservers(WorkResult result) {
        for (Listener listener : listeners) {
            listener.onEvent(result);
        }
    }
}
```

OBSERVER

Strategy

Type: Behavioral

What it is:
Define a family of algorithms, encapsulate each one, and make them interchangeable. Lets the algorithm vary independently from
clients that use it.



- define a family of algorithms
- encapsulate each one
- make them interchangeable

STRATEGY

strategy

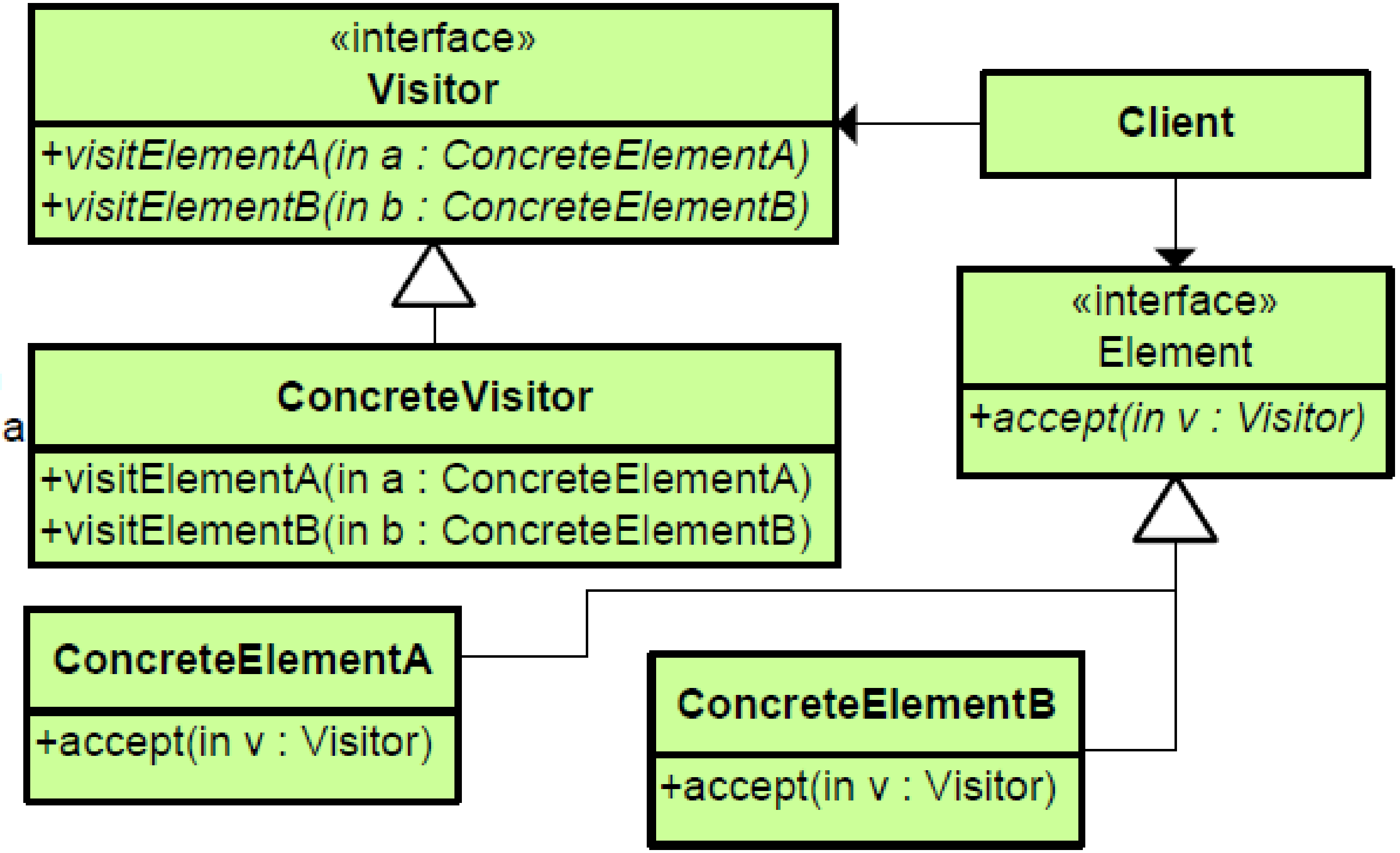
- define a family of algorithms
- encapsulate each one
- make them interchangeable

STRATEGY

Visitor

Type: Behavioral

What it is:
Represent an operation to be performed on the elements of an object structure. Lets you define a new operation without changing the classes of the elements on which it operates.



visitor

- represent an operation to be performed on the elements of an object structure
- define a new operation without changing the classes on which it operates.

VISITOR

```
interface VisitedElement {
     String getContent();
     void accept(Visitor visitor);
}

class ElementOne implements VisitedElement {
     void accept(Visitor visitor) {
          visitor.visit(this);
     }
}

class ElementTwo implements VisitedElement {
     void accept(Visitor visitor) {
          visitor.visit(this);
     }
}
```

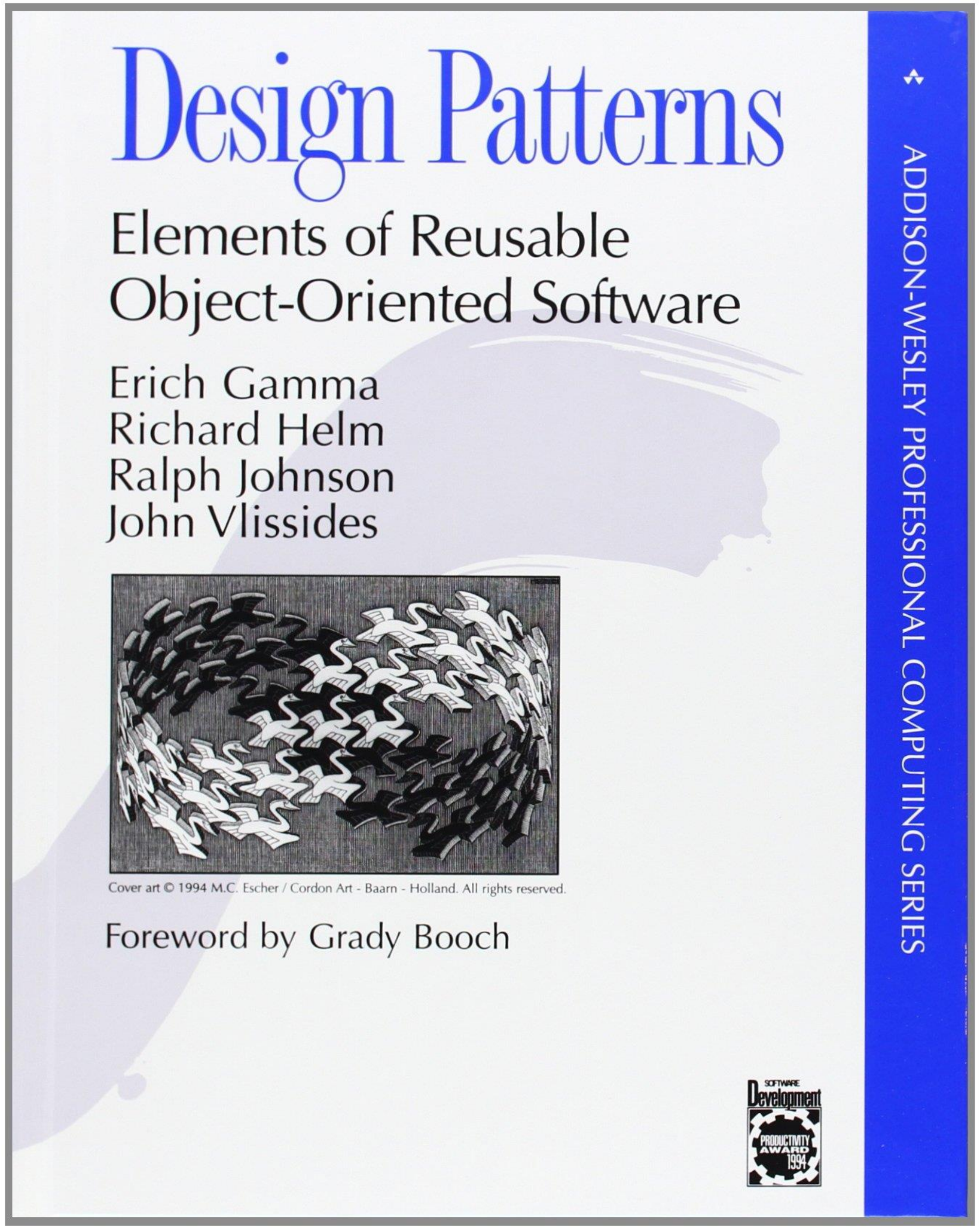
VISITOR

```
interface Visitor {
        void visit(ElementOne element);
        void visit(ElementTwo element);
}

class PrintVisitor implements Visitor {
        @Override
        void visit(ElementOne element) {
                print(element.getContent());
        }
        //…rest goes here
}
```

VISITOR

“Design Patterns”
by
Gang of Four

1994



Design Patterns

Elements of Reusable
Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

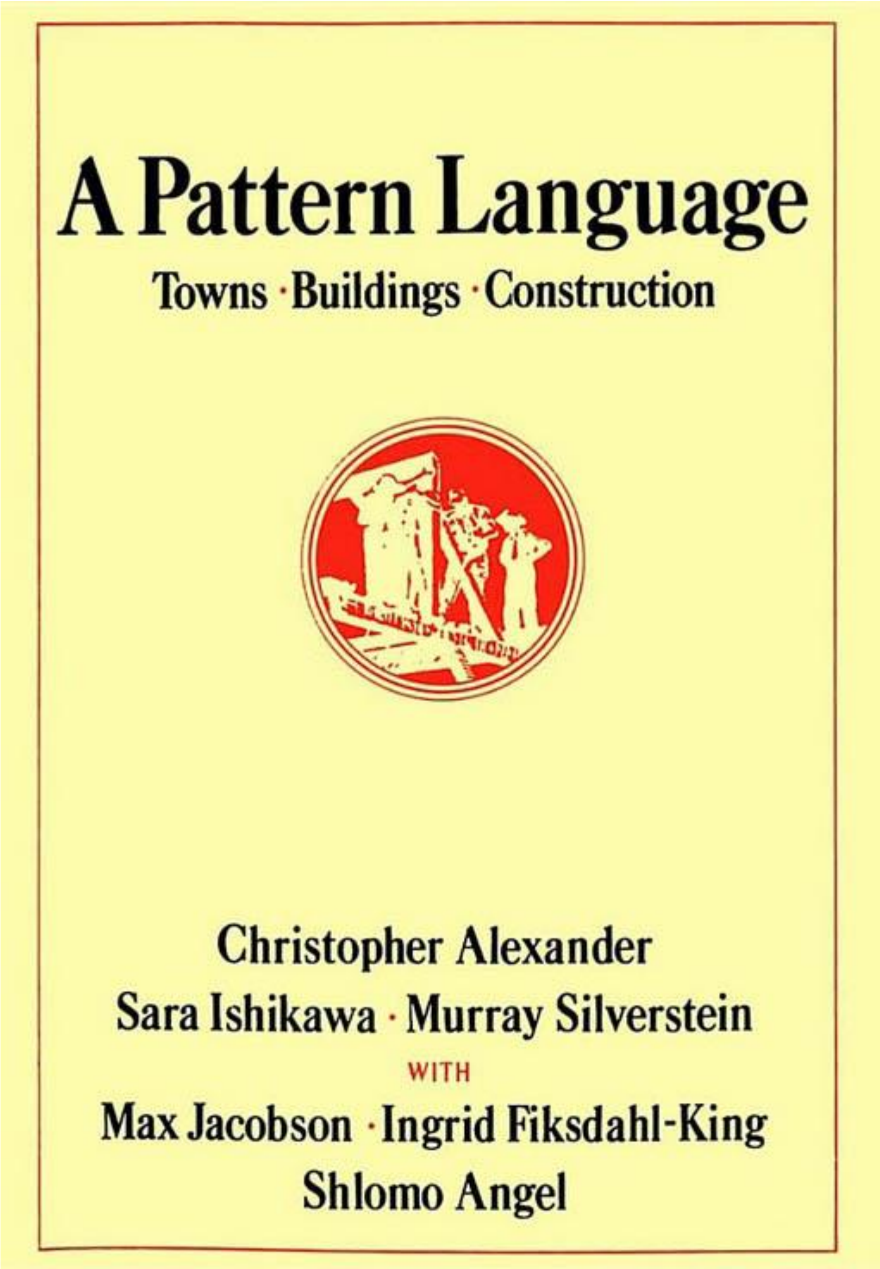
Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

“A Pattern Language”
by
Christopher Alexander et al.

1977



A Pattern Language

Towns · Buildings · Construction

Christopher Alexander
Sara Ishikawa · Murray Silverstein
WITH
Max Jacobson · Ingrid Fiksdahl-King
Shlomo Angel

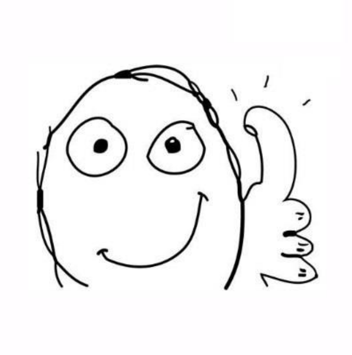
A Pattern Language

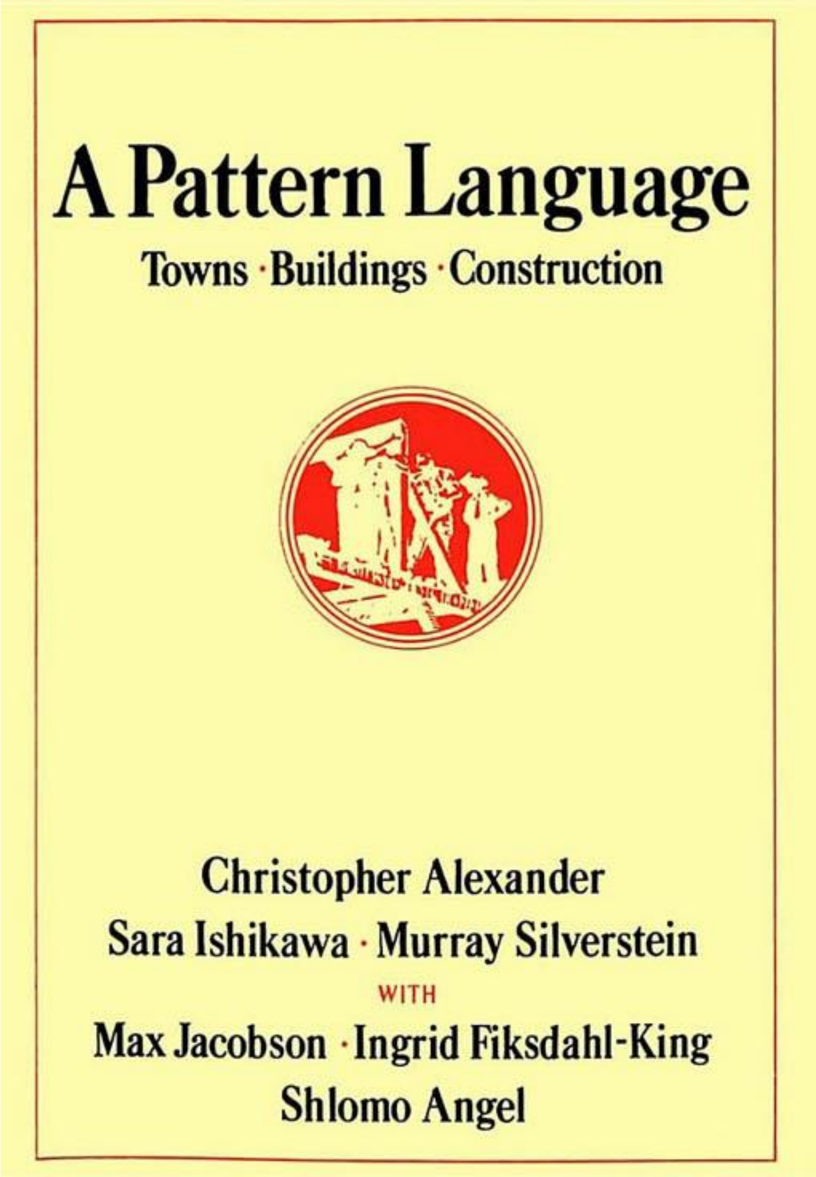
- a dictionary of terms laying out a set of basic design decisions
- design discussions are conducted using this language
- design at all levels springs from this common base
- the common language promotes commonality of design

It does not tell you ***how to*** design anything

It helps you decide ***what*** should be designed

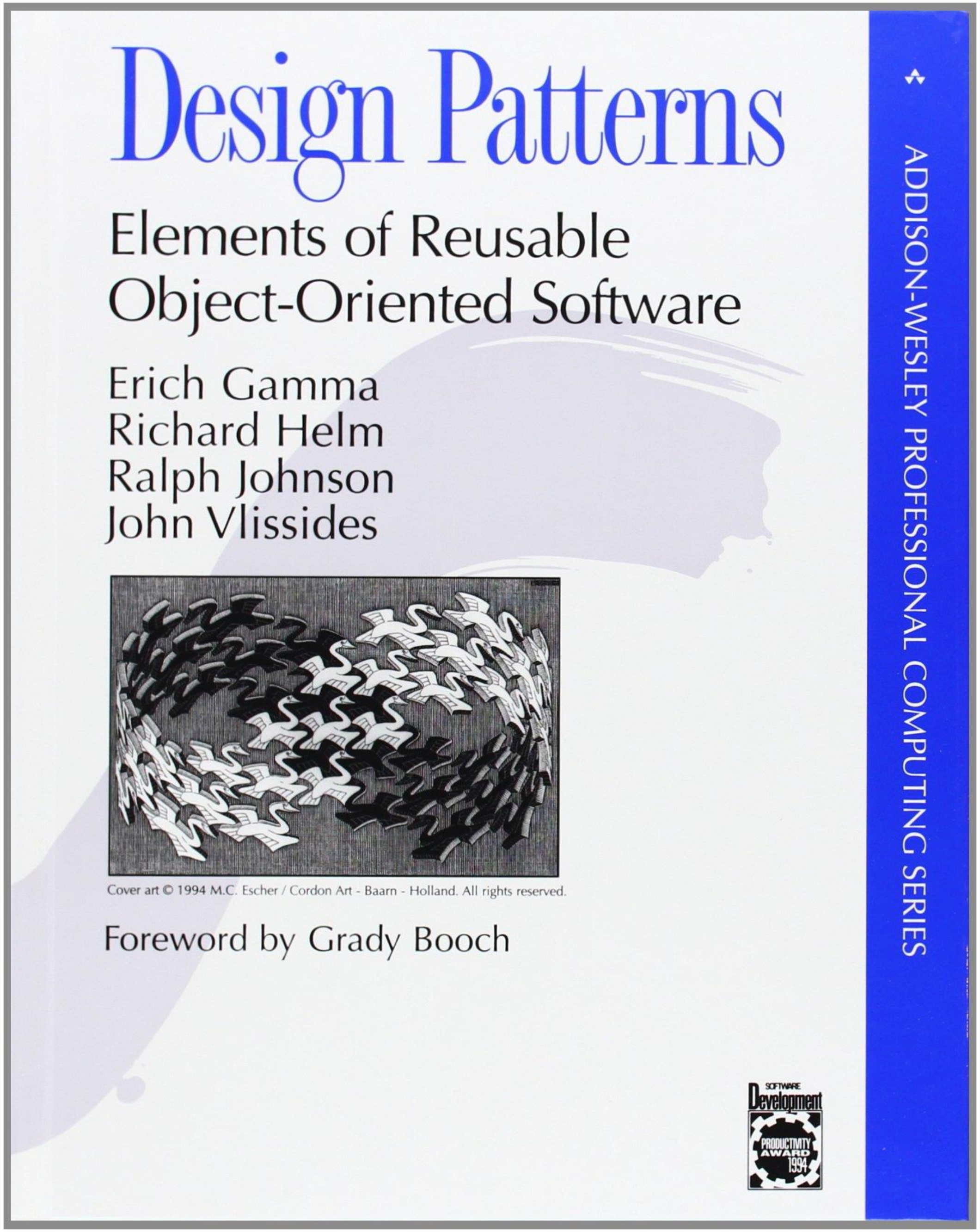
You get to ***make up*** whatever patterns
that lead to good designs




A Pattern Language
Towns · Buildings · Construction
Christopher Alexander
Sara Ishikawa · Murray Silverstein
WITH
Max Jacobson · Ingrid Fiksdahl-King
Shlomo Angel

A Pattern Language

- **a dictionary of term**
- **design discussions a**
- **design at all levels s**
- **the common languag**



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

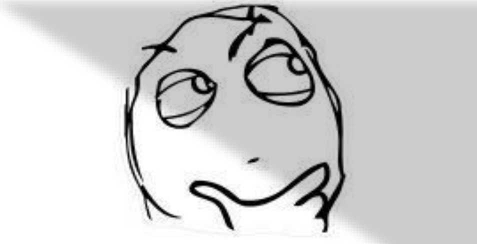
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

SOFTWARE Development PRODUCTIVITY AWARD 1994

It does not tel

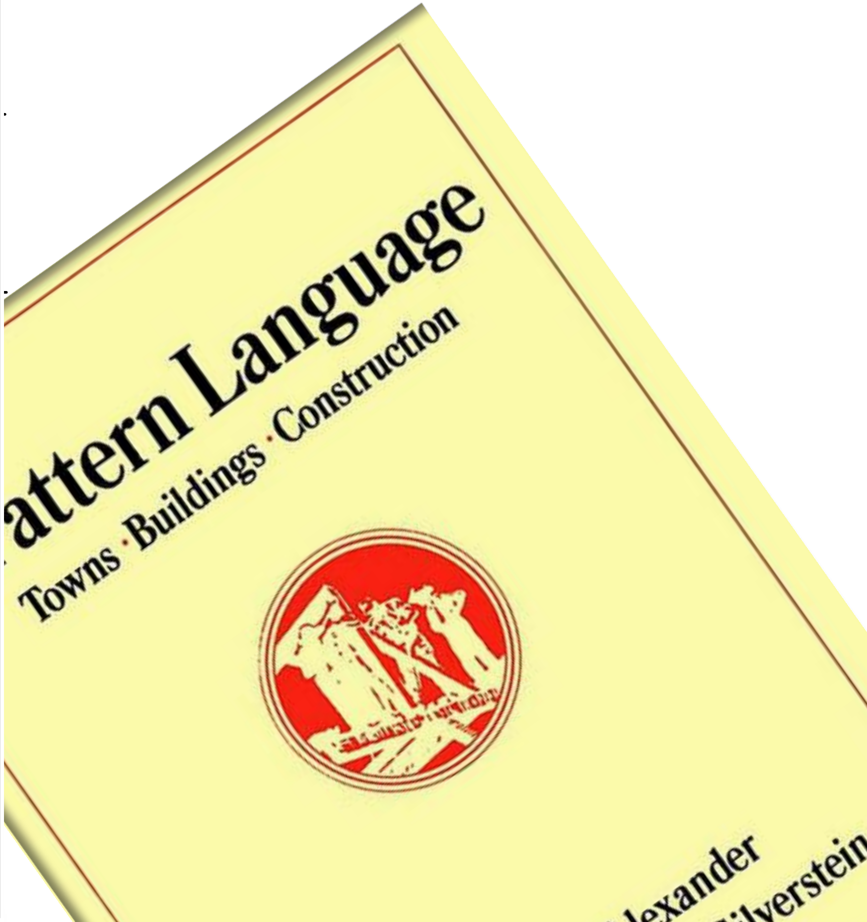
It helps you de

You get to *make*
that lead



“Design Patterns are a library of code templates!”

developers



attern Language
Towns · Buildings · Construction

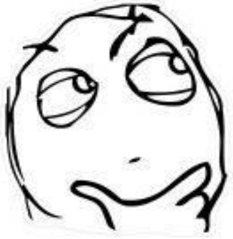
lexander
lverstein

"Design Patterns" Aren't

M. J. Dominus, 2002

Is “Iterator” really "a recurring design problem"?

In C++ it is, because C++ sucks (ditto Java).
But in a better language, it's not a problem at all.



For example, Perl provides a universal solution:

```
foreach $element (@collection) { ... }
```

This fails in C++ because the type system is too weak.
Solutions with higher-order functions fail too.
No anonymous functions or lexical closure (Ditto Java).

Other ~~good~~ solutions to this problem include a good macro system.

But the C++ macro system blows goat dick.

“When I see patterns in my programs, I consider it a sign of trouble.

The shape of a program should reflect only the problem it needs to solve. Any other regularity in the code is a sign, to me at least, that I'm using abstractions that aren't powerful enough-- often that I'm generating by hand ***the expansions of some macro*** that I need to write.”

Paul Graham

“So start small, and think about the details. Don’t think about some big picture and fancy design. If it doesn’t solve some ***fairly immediate*** need, it’s almost certainly over-designed.”

Linus Torvalds

“Patterns, like all forms of compexity, should be avoided until they are ***absolutely necessary***.”

Jeff Atwood

DRY

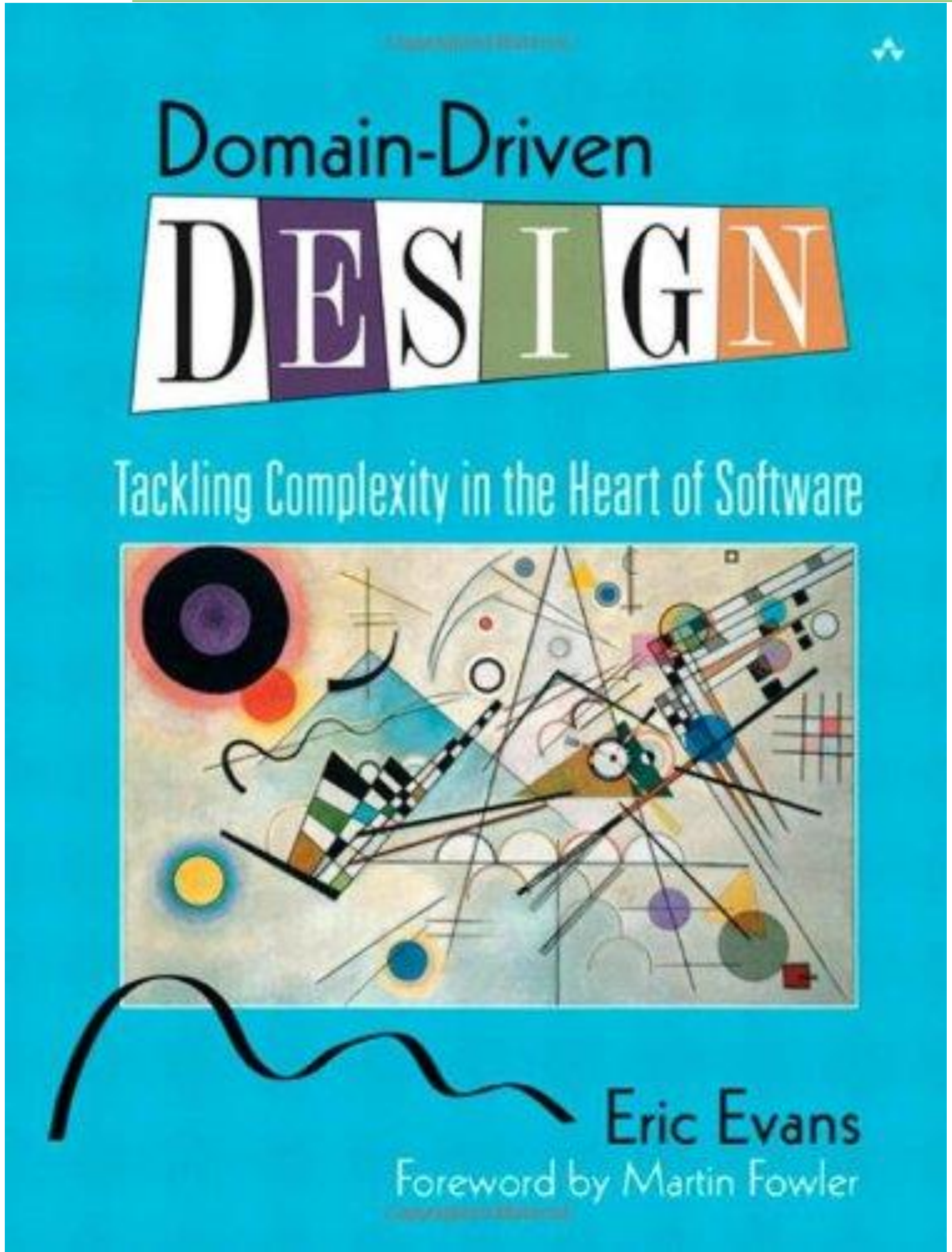
Don’t Repeat Yourself

KISS!

Keep It Simple, Stupid!

YAGNI

You Ain’t Gonna Need It


Domain-Driven
DESIGN
Tackling Complexity in the Heart of Software
Eric Evans
Foreword by Martin Fowler