

# The GoF Design Patterns Memory

Version 01.00 / 01.10.2016 / Printed 29.03.2017



## Table of Contents

Preface .....	viii
I. Introduction .....	1
1. DESIGN PRINCIPLES .....	2
2. OVERVIEW .....	7
II. Creational Patterns .....	11
1. ABSTRACT FACTORY .....	12
Intent .....	12
Problem .....	13
Solution .....	14
Motivation 1 .....	16
Applicability .....	17
Structure, Collaboration .....	18
Consequences .....	19
Implementation .....	20
Sample Code 1 .....	21
Sample Code 2 .....	24
Sample Code 3 .....	26
Related Patterns .....	28
2. BUILDER .....	29
Intent .....	29
Problem .....	30
Solution .....	31
Motivation 1 .....	32
Applicability .....	33
Structure, Collaboration .....	34
Consequences .....	35
Implementation .....	36
Sample Code 1 .....	37
Related Patterns .....	39
3. FACTORY METHOD .....	40
Intent .....	40
Problem .....	41
Solution .....	42
Motivation 1 .....	43
Applicability .....	44
Structure, Collaboration .....	46
Consequences .....	47
Implementation .....	48
Sample Code 1 .....	50
Sample Code 2 .....	52
Related Patterns .....	53
4. PROTOTYPE .....	54
Intent .....	54
Problem .....	55
Solution .....	56
Motivation 1 .....	57
Applicability .....	58
Structure, Collaboration .....	59
Consequences .....	60
Implementation .....	61
Sample Code 1 .....	62
Sample Code 2 .....	64
Related Patterns .....	66

5. SINGLETON .....	67
Intent .....	67
Problem .....	68
Solution .....	69
Motivation 1 .....	70
Applicability .....	71
Structure, Collaboration .....	72
Consequences .....	73
Implementation .....	74
Sample Code 1 .....	75
Related Patterns .....	76
III. Structural Patterns .....	77
1. ADAPTER .....	78
Intent .....	78
Problem .....	79
Solution .....	80
Motivation 1 .....	81
Applicability .....	82
Structure, Collaboration .....	83
Consequences .....	84
Implementation .....	85
Sample Code 1 .....	86
Related Patterns .....	87
2. BRIDGE .....	88
Intent .....	88
Problem .....	89
Solution .....	90
Motivation 1 .....	91
Applicability .....	92
Structure, Collaboration .....	93
Consequences .....	94
Implementation .....	95
Sample Code 1 .....	96
Related Patterns .....	97
3. COMPOSITE .....	98
Intent .....	98
Problem .....	99
Solution .....	100
Motivation 1 .....	101
Applicability .....	102
Structure, Collaboration .....	103
Consequences .....	104
Implementation .....	105
Sample Code 1 .....	106
Sample Code 2 .....	108
Related Patterns .....	110
4. DECORATOR .....	111
Intent .....	111
Problem .....	112
Solution .....	113
Motivation 1 .....	114
Applicability .....	115
Structure, Collaboration .....	116
Consequences .....	117
Implementation .....	118

Sample Code 1 .....	119
Sample Code 2 .....	121
Related Patterns .....	122
5. FACADE .....	123
Intent .....	123
Problem .....	124
Solution .....	125
Motivation 1 .....	126
Applicability .....	127
Structure, Collaboration .....	128
Consequences .....	129
Implementation .....	130
Sample Code 1 .....	131
Related Patterns .....	133
6. FLYWEIGHT .....	134
Intent .....	134
Problem .....	135
Solution .....	136
Motivation 1 .....	137
Applicability .....	138
Structure, Collaboration .....	139
Consequences .....	140
Implementation .....	141
Sample Code 1 .....	142
Related Patterns .....	144
7. PROXY .....	145
Intent .....	145
Problem .....	146
Solution .....	147
Motivation 1 .....	148
Applicability .....	149
Structure, Collaboration .....	150
Consequences .....	151
Implementation .....	152
Sample Code 1 .....	153
Related Patterns .....	154
IV. Behavioral Patterns .....	155
1. CHAIN OF RESPONSIBILITY .....	156
Intent .....	156
Problem .....	157
Solution .....	158
Motivation 1 .....	159
Applicability .....	160
Structure, Collaboration .....	161
Consequences .....	162
Implementation .....	163
Sample Code 1 .....	164
Related Patterns .....	166
2. COMMAND .....	167
Intent .....	167
Problem .....	168
Solution .....	169
Motivation 1 .....	170
Applicability .....	171
Structure, Collaboration .....	172

Consequences .....	173
Implementation .....	174
Sample Code 1 .....	175
Related Patterns .....	176
3. INTERPRETER .....	177
Intent .....	177
Problem .....	178
Solution .....	179
Motivation 1 .....	181
Applicability .....	182
Structure, Collaboration .....	183
Consequences .....	184
Implementation .....	185
Sample Code 1 .....	186
Sample Code 2 .....	188
Related Patterns .....	192
4. ITERATOR .....	193
Intent .....	193
Problem .....	194
Solution .....	195
Motivation 1 .....	196
Applicability .....	197
Structure, Collaboration .....	198
Consequences .....	199
Implementation .....	200
Sample Code 1 .....	201
Sample Code 2 .....	203
Related Patterns .....	207
5. MEDIATOR .....	208
Intent .....	208
Problem .....	209
Solution .....	210
Motivation 1 .....	211
Applicability .....	212
Structure, Collaboration .....	213
Consequences .....	214
Implementation .....	215
Sample Code 1 .....	216
Related Patterns .....	219
6. MEMENTO .....	220
Intent .....	220
Problem .....	221
Solution .....	222
Motivation 1 .....	223
Applicability .....	224
Structure, Collaboration .....	225
Consequences .....	226
Implementation .....	227
Sample Code 1 .....	228
Related Patterns .....	230
7. OBSERVER .....	231
Intent .....	231
Problem .....	232
Solution .....	233
Motivation 1 .....	234

Applicability .....	235
Structure, Collaboration .....	236
Consequences .....	237
Implementation .....	238
Sample Code 1 .....	239
Sample Code 2 .....	241
Sample Code 3 .....	243
Sample Code 4 .....	245
Related Patterns .....	247
8. STATE .....	248
Intent .....	248
Problem .....	249
Solution .....	250
Motivation 1 .....	251
Applicability .....	252
Structure, Collaboration .....	253
Consequences .....	254
Implementation .....	255
Sample Code 1 .....	256
Sample Code 2 .....	258
Related Patterns .....	260
9. STRATEGY .....	261
Intent .....	261
Problem .....	262
Solution .....	263
Motivation 1 .....	265
Applicability .....	266
Structure, Collaboration .....	268
Consequences .....	269
Implementation .....	270
Sample Code 1 .....	271
Sample Code 2 .....	273
Sample Code 3 .....	276
Related Patterns .....	282
10. TEMPLATE METHOD .....	283
Intent .....	283
Problem .....	284
Solution .....	285
Motivation 1 .....	286
Applicability .....	287
Structure, Collaboration .....	288
Consequences .....	289
Implementation .....	290
Sample Code 1 .....	291
Sample Code 2 .....	292
Related Patterns .....	293
11. VISITOR .....	294
Intent .....	294
Problem .....	295
Solution .....	296
Motivation 1 .....	297
Applicability .....	298
Structure, Collaboration .....	299
Consequences .....	300
Implementation .....	301

Sample Code 1 .....	302
Sample Code 2 .....	304
Related Patterns .....	309
V. GoF Design Patterns Update .....	310
1. DEPENDENCY INJECTION .....	311
Intent .....	311
Problem .....	312
Solution .....	313
Motivation 1 .....	314
Applicability .....	315
Structure, Collaboration .....	316
Consequences .....	317
Implementation .....	318
Sample Code 1 .....	319
Related Patterns .....	321
A. Bibliography .....	322

## Preface

The GoF Design Patterns Memory presents the standard GoF<sup>1</sup> design patterns in a compact and memory friendly form so that they can be searched, compared, studied, and ultimately *memorized* as fast as possible.

GoF design patterns describe how to solve recurring problems when designing flexible and reusable object-oriented software.

By learning the 23 GoF design patterns, you will learn how to design objects that are easier to *implement, change (maintain), and reuse*.

By working through the ready-to-run code samples, you will learn how to implement design patterns by using object-oriented programming languages such as Java.

We use a consistent and simple language and repeat important phrases whenever appropriate. Because a picture is worth a thousand words, each section of each design pattern starts with UML diagrams to quickly communicate the key aspects of the design under discussion.

At w3sDesign you will find all you need to know, and you will get the skills that software developers need today.

*It's all for free, and it's pretty fast. Enjoy it!*

<sup>1</sup> **Design Patterns: Elements of Reusable Object-Oriented Software.**

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides.

Copyright © 1995 by Addison-Wesley.

## **Part I. Introduction**

Program to an interface, not an implementation. First Design Principle [GoF, p18]

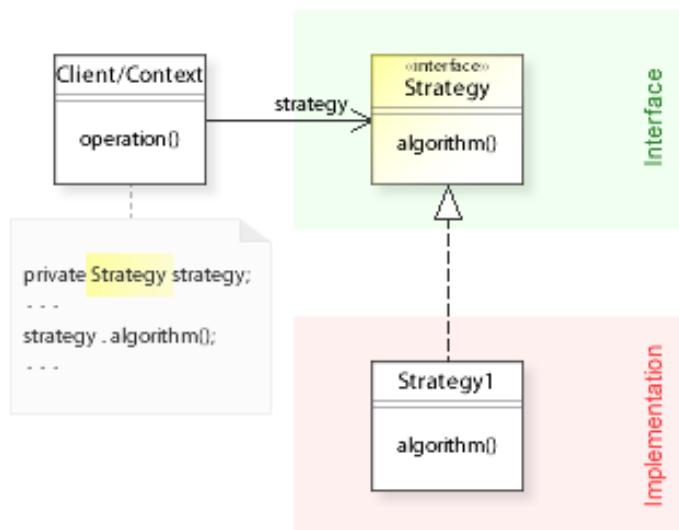


Figure 1 Strategy Design Pattern - Sample Class Diagram

This design principle greatly reduces implementation dependencies:

**Clients refer to an interface and are independent of an implementation.**

That means, an implementation can vary independently from (without having to change) existing clients. This is a common theme of the design patterns described here, and it ensures that a system is written in terms of interfaces, not implementations.

As a consequence, **clients depend on an interface**.

That means, varying an interface will break existing clients.

Therefore, interfaces must be designed carefully.

*Figure 1 shows an example of the Strategy design pattern:*

Context refers to the strategy interface (`Strategy`) and is independent of an implementation (`Strategy1`, ...). The implementation can vary (new algorithms can be added and existing ones can be changed) independently from (without having to change) the context.

Cleanly separate interface and implementation.

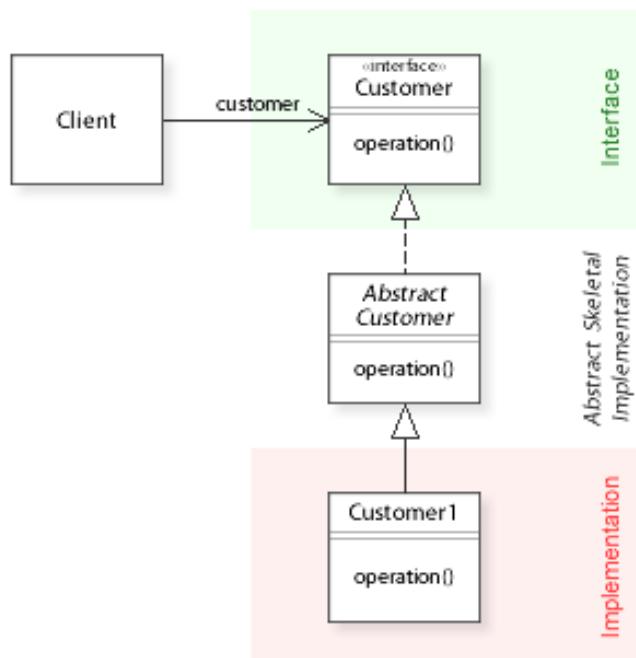


Figure 2 Interface Design - Sample Class Diagram

When designing interfaces you are in one of two situations:

### (1) Internal Interface

Designing an interface that you do not publish to the public world.

It is used only internally, and its clients are known and under your control.

Varying an internal interface is possible because you can change existing clients.

### (2) Published Interface

Designing an interface that you publish to the public world.

It is widely used, and its clients are not known and not under your control.

Once you have published an interface, you must support it forever.

Varying a published interface is almost impossible because you would break existing clients.

Publishing an interface is a big investment.

*Figure 2 shows an example of an interface design:*

Interface (`Customer`) and implementation (`Customer1`,...) are cleanly separated.

Additionally, "Provide an abstract skeletal implementation class to go with each nontrivial interface that you export." [JB08, p94]

This design combines the power of interface and abstract skeletal implementation:

The *interface* lets you define types independently from the existing class hierarchy.

The *abstract class* lets you (1) add new behavior without breaking clients, (2) implement common/default/invariant behavior, and (3) control subclassing (enforce invariants and enable variants; see also Template Method).

For example, the *Java Collections API* is a great place for studying interface design.

Favor object composition over class inheritance. Second Design Principle [GoF, p18]

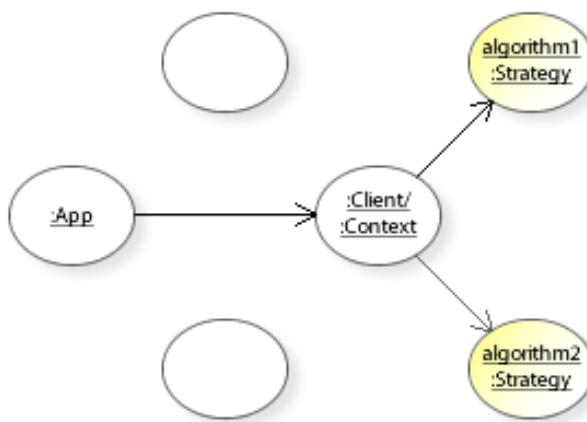


Figure 3 Strategy Design Pattern - Sample Object Collaboration

#### Use *class inheritance* to change behavior statically at compile-time.

Use class inheritance only if there is really an “is a” relationship between parent class and subclasses. Whenever you want to represent some kind of *classification*, where something more *specialized* “is a” kind of something more *generalized*.

For example, a rose “is a” flower, a dog “is an” animal, a rectangle “is a” shape, and the like. This results in well-designed “is a” inheritance hierarchies among classes.

The parent class implements the generalized (common/invariant) behavior, and subclasses inherit from their parent class and implement the specialized (custom/variant) behavior.

#### Use *object composition* to change behavior dynamically at run-time.

With object composition, a set of objects collaborate to perform and change behavior dynamically at run-time.

Object composition requires well-designed objects that can collaborate solely through their interfaces.

*Figure 3 shows an example of the Strategy design pattern:*

Instead of changing an algorithm via class inheritance (statically at compile-time), context delegates an algorithm to different strategy objects (dynamically at run-time).

Hint: Start studying design patterns with *Strategy*, *Template Method*, and *Decorator*.

## Background Information

### Interface

The (public) interface of an object consists of the object's (public) operations.

Interface is "The set of all signatures defined by an object's operations. The interface describes the set of requests to which an object can respond." [GoF, p361]

"An operation's signature defines its name, parameters, and return value." [GoF, p361]

Note that in languages such as Java, an operation's signature only comprises the method's name and the parameter types.

Interface is "The outside view of, for example, a class, object, component, or composite structure, that emphasizes its abstraction while hiding its structure and the secrets of its behavior." [GB07, p596]

Structure is "The concrete representation of the state of an object." [GB07, p601]

"An interface is a contract between a class and the outside world. When a class implements an interface, it promises to provide the behavior published by that interface."

[Oracle Java Tutorial]

In languages that do not support a separate language construct, interfaces are declared as pure (100%) abstract classes.

### Implementation / Encapsulation

To cleanly separate interface and implementation, an implementation must be encapsulated.

Encapsulation is a fundamental object-oriented concept for hiding all implementation details (secrets) of an object. Objects then collaborate only through their interfaces and are independent of how they are implemented.

Hiding implementation details means, hiding the implementation of an object's operations, and hiding the representation of an object's state (i.e., information hiding, not just data hiding).

"Encapsulation hides the details of the implementation of an object." [GB07, p51]

"After carefully designing your class's public API, your reflex should be to make all other members private." [JB08, p69]

"A well-designed module hides all of its implementation details, cleanly separating its API from its implementation." [JB08, p67]

An object's interface = outside view = client view.

An object's implementation = inside view = hidden from clients.

## Design for Class Inheritance

The conceptual dependency between a parent class and its subclasses has a consequence:  
**Class inheritance breaks encapsulation.**

That means, the implementation details of a parent class are not hidden from its subclasses.  
Changing a parent's implementation detail will change (and may break) existing subclasses.  
Therefore, "Design and document for [class] inheritance or else prohibit it." [JB08, Item 17]  
See also *Template Method design pattern*.

## Design for Object Composition

With object composition, a set of objects collaborate solely through their interfaces.

### Finding the Right Objects

"The hard part about object-oriented design is decomposing a system into objects." [GoF, p11]  
Many objects in a design come from the analysis model and have counterparts in the real world. Such key abstractions are part of the vocabulary of the problem domain;  
"if the domain expert talks about it, the abstraction is usually important." [GB07, p139]  
For example, objects like customer, product, order, etc.

"But object-oriented designs often end up with classes that have no counterparts in the real world." [GoF, p11]

To make a system more flexible and reusable, separate objects must be designed that have no counterparts in the real world.

"Design patterns help you identify less-obvious abstractions and the objects that can capture them. For example, objects that represent a process or algorithm don't occur in nature, yet they are a crucial part of flexible designs. The Strategy (315) pattern describes how to implement interchangeable families of algorithms." [GoF, p13]

## Changing Behavior – Making a system independent of changing requirements.

### Run-Time Flexibility via Object Composition



Using different algorithms.  
Selecting and changing which algorithm to use dynamically.  
Adding new algorithms and changing existing ones independently.



Changing the behavior of an object when its internal state changes.  
Adding new states and changing the behavior of existing ones independently.



Adding responsibilities to an object dynamically.  
Extending the functionality of an object dynamically.



Controlling the way an object is accessed.  
Providing additional functionality when accessing an object.

### Compile-Time Flexibility via Class Inheritance



Defining a behavior  
so that subclasses can change certain parts of the behavior  
without changing the behavior's structure.

Changing Object Creation – Making a system independent of how its objects are created.

## Run-Time Flexibility via Object Composition



Creating different families of objects.  
Selecting and changing which family of objects to create dynamically.



Creating different representations of a complex object.  
Selecting and changing which representation to create dynamically.



Creating new objects by cloning prototype objects.  
Selecting and changing prototype objects dynamically.

## Compile-Time Flexibility via Class Inheritance



Creating an object  
so that subclasses can change the way the object is created.



Ensuring that a class has only one instance.  
Providing global access to the sole instance of a class.

## Object Structures – Working with complex object structures efficiently.



Representing part-whole hierarchies as tree structures.  
Treating all objects in a part-whole hierarchy uniformly.



Accessing the elements of an object structure  
without exposing its underlying representation.



Defining new operations for the classes of an object structure  
independently from (without changing) the classes.



Supporting large numbers of fine-grained objects efficiently.

## Object Collaboration – Avoiding tight coupling between interacting objects.



Defining a one-to-many dependency between objects  
without making the objects tightly coupled.  
Notifying an open-ended number of dependent objects.



Avoiding tight coupling between a set of interacting objects.  
Changing the interaction behavior independently.

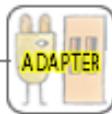


Avoiding tight coupling the sender of a request to its receiver.  
Determining the receiver (handler) of a request dynamically.

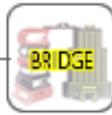


Avoiding tight coupling the sender of a request to its receiver.  
Configuring the sender of a request with a request.  
Queuing and logging requests.

## Changing Interfaces Independently



Providing a different interface to an object.  
Letting objects work together that have incompatible interfaces.



Decoupling an abstraction from its implementation.  
Letting an abstraction and its implementation vary independently.



Providing a simple interface to a complex subsystem.  
Making a complex subsystem easier to use.

## Storing and Restoring Object State



Storing and restoring an object's internal state  
without violating encapsulation.

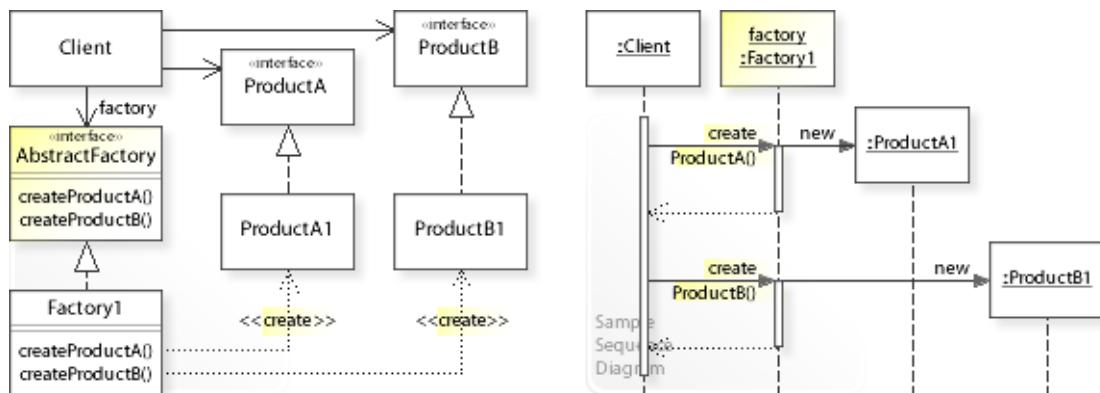
## Interpreter / Domain Specific Languages



Interpreting sentences in a simple language.

## **Part II. Creational Patterns**

## Intent



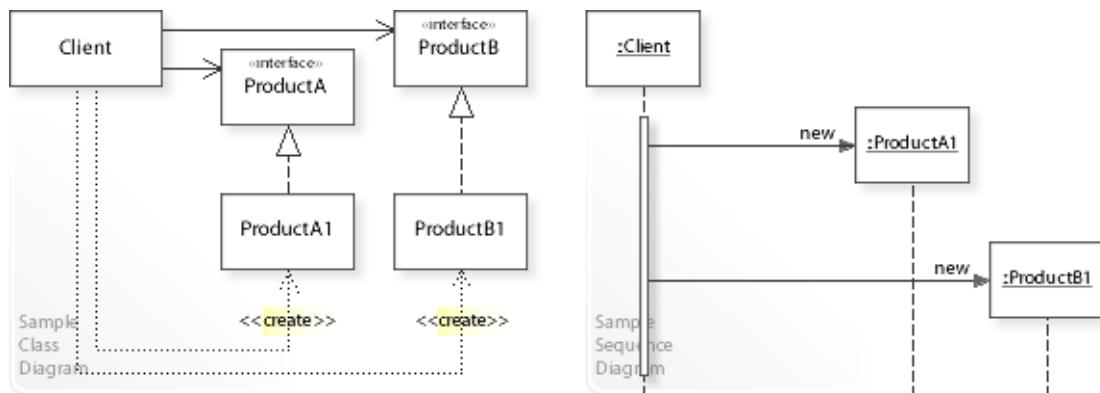
The intent of the Abstract Factory design pattern is to:

**"Provide an interface for creating families of related or dependent objects without specifying their concrete classes."** [GoF]

See Problem and Solution sections for a more structured description of the intent.

- The Abstract Factory design pattern solves problems like:
  - *How can an application be independent of how its objects are created?*
  - *How can families of related or dependent objects be created?*
- Instantiating concrete classes directly within a class makes it impossible to change the instantiation later independently from (without having to change) the class.
- For example, supporting different look-and-feels in a Web/GUI application.  
Instantiating look-and-feel-specific classes throughout the application should be avoided so that a look and feel can be selected and changed at run-time.
- The Abstract Factory pattern describes *how* to solve such problems:
  - *Provide an interface for creating families of related or dependent objects without specifying their concrete classes:*  
`AbstractFactory | createProductA(), createProductB(), ...`
  - The process of object creation: `new ProductA1()`, ...  
is abstracted by delegating to a factory object: `factory.createProductA()`, ...  
There is no longer anything in the client code that refers to a concrete product class.

## Problem



The Abstract Factory design pattern solves problems like:

**How can an application be independent of how its objects are created?**

**How can families of related or dependent objects be created?**

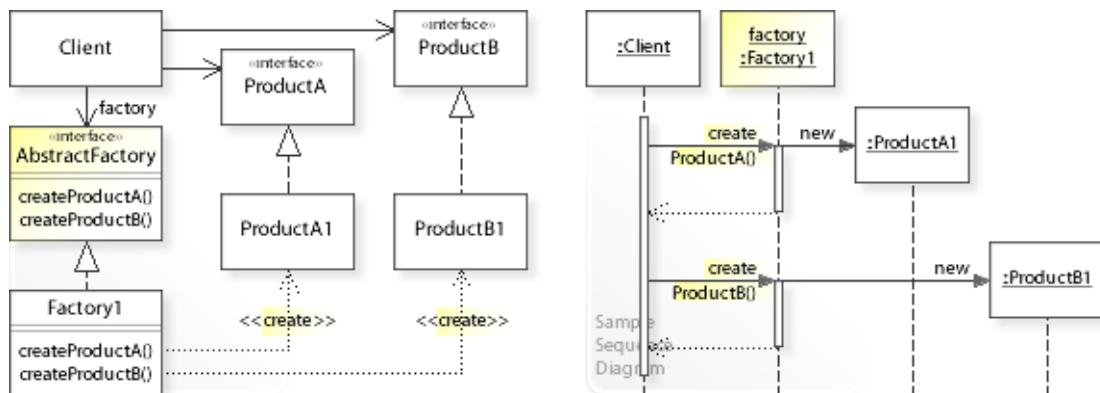
See Applicability section for all problems Abstract Factory can solve.

- An inflexible way to create objects is to instantiate concrete classes (`new ProductA1()`, `new ProductB1()`) directly within a class (`Client`).
- This commits the class to creating particular objects and makes it impossible to change the instantiation later independently from (without having to change) the class.  
It stops the class from being reusable with different objects, and it makes the class hard to test because real objects can't be replaced with mock objects.  
"Instantiating look-and-feel-specific classes of widgets throughout the application makes it hard to change the look and feel later." [GoF, p87]  
Furthermore, "Specifying a class name when you create an object commits you to a particular implementation instead of a particular interface." [GoF, p24]
- *That's the kind of approach to avoid if we want that the way objects are created can be changed at run-time.*
- For example, classes in an order processing application require many other objects like customers, products, and inventory.  
Classes should avoid instantiating concrete classes directly so that the way objects are created can be changed at run-time.
- For example, supporting different look-and-feels in a Web/GUI application.  
Instantiating look-and-feel-specific classes throughout the application should be avoided so that a look and feel can be selected and changed at run-time.

## Background Information

- The Swing GUI toolkit of the Java platform, for example, lets you create (and switch between) different families of objects to support different look-and-feels (like Java, Windows, and custom look-and-feels).

## Solution



The Abstract Factory pattern describes a solution:

**Encapsulate object creation in a separate factory object.**

Describing the Abstract Factory design in more detail is the theme of the following sections.  
See Applicability section for all problems Abstract Factory can solve.

- The key idea in this pattern is to abstract the process of object creation.  
The process of object creation: `new ProductA1()`, ...  
is abstracted by delegating to a factory object: `factory.createProductA()`, ...  
There is no longer anything in the client code that refers to a concrete product class.  
"Creational patterns ensure that your application is written in terms of interfaces, not implementations." [GoF, p18]
- **Define separate factory objects:**
  - For all supported families of objects, define a common interface for creating a family of objects (`AbstractFactory` | `createProductA()`, `createProductB()`, ...).
  - Define classes (`Factory1`, ...) that implement the interface.
- This enables *compile-time* flexibility (via inheritance).  
The way objects are created can be implemented and changed independently from clients by defining new (sub)classes.
- **Clients delegate the responsibility to create objects to a factory object** (`factory.createProductA()`, `factory.createProductB()`, ...).
- This enables *run-time* flexibility (via object composition).  
Clients can use different factory objects to change the way objects are created at run-time.  
Clients can be configured with a factory object, which they use to create objects, and even more, the factory object can be exchanged at run-time.

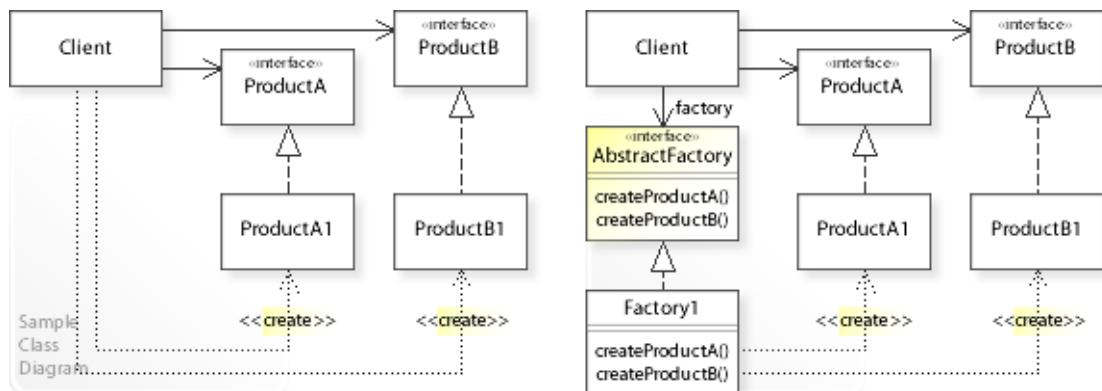
## Background Information

- "Not only must we avoid making explicit constructor calls; we must also be able to replace an entire widget set easily. We can achieve both by *abstracting the process of object creation.*" [GoF, p48]
- Abstract Factory is often simply called Factory because most patterns do some kind of abstraction. The Strategy pattern, for example, abstracts and encapsulates an algorithm. "Abstraction and encapsulation are complementary concepts [...] For abstraction to work, implementations must be encapsulated." [GBooch07, p51]
- Abstract Factory is also known as Factory Object because object creation must be encapsulated in a separate factory object before a client's creation code can be abstracted (by delegating instantiation to that factory object).

The name Factory Object also clearly communicates the relationship to Factory Method that defines a separate factory method for creating an object.

- Note that for simple applications that do not need interchangeable families of objects, a common implementation of the Abstract Factory pattern is just a concrete factory class that acts as both the interface and implementation (see Implementation).

## Motivation 1



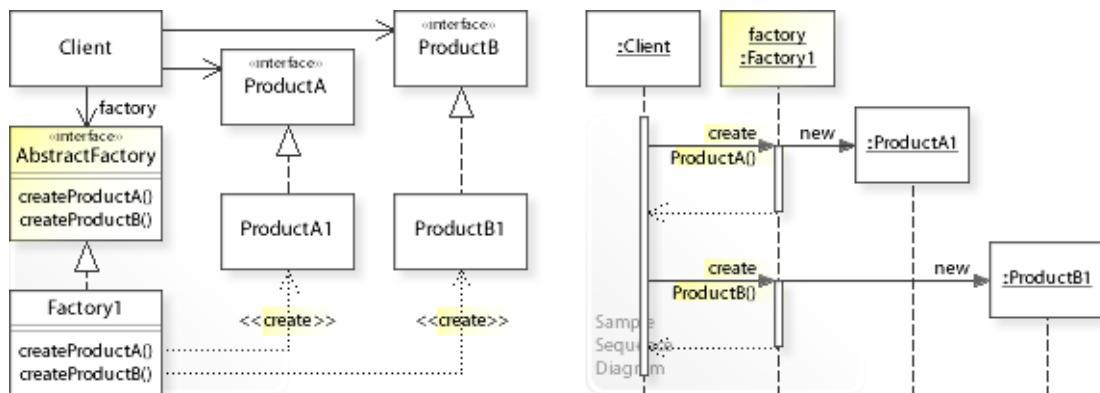
**Consider the left design (problem):**

- Hard-wired object creation.
  - Creating objects is implemented (hard-wired) directly within a class (`Client`).
  - This makes it hard to change the way objects are created (which concrete classes get instantiated) independently from (without having to change) the `Client` class.
- Distributed object creation.
  - Creating objects is distributed across the classes of an application.

**Consider the right design (solution):**

- Encapsulated object creation.
  - Creating objects is implemented (encapsulated) in a separate class (`Factory1`).
  - This makes it easy to change the way objects are created (which concrete classes get instantiated) independently from (without having to change) the `Client` class.
- Centralized object creation.
  - Creating objects is centralized in a single `Factory1` class.

## Applicability



## Design Problems

- **Creating Objects**
  - How can an application be independent of how its objects are created?
  - How can object creation be separated from an application?
  - How can the way objects are created be changed at run-time?
  - How can a class delegate instantiation to another object (factory object)?
  - How can a class be configured with a factory object?
- **Supporting Interchangeable Families of Objects**
  - How can families of related or dependent objects be created?
  - How can the way a family of objects is created be changed at run-time?
  - How can be ensured that a family of objects is created and used together (consistent object families)?

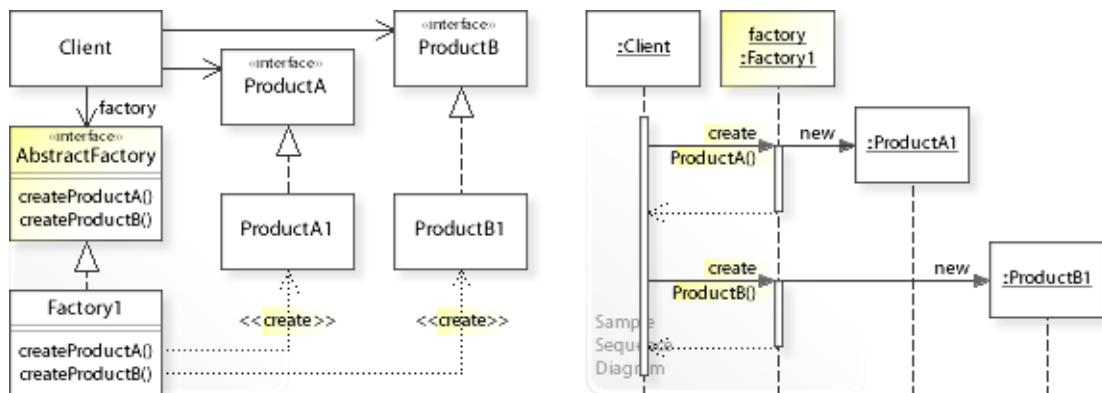
## Refactoring Problems

- **Inflexible Code**
  - How can instantiating concrete classes throughout an application (compile-time implementation dependencies) be refactored?
  - How can object creation that is distributed across an application be centralized? *Move Creation Knowledge to Factory* (68) [JKerivsky05]

## Testing Problems

- **Unit Testing**
  - How can a class be configured with a mock factory instead of a real factory so that the class can be unit tested in isolation?

## Structure, Collaboration



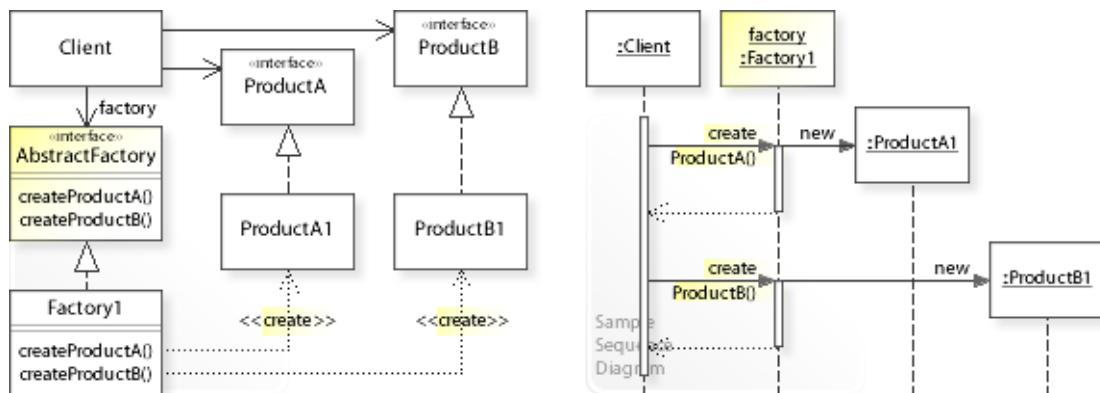
## Static Class Structure

- Client
  - Refers to the **AbstractFactory** interface to create **ProductA** and **ProductB** objects and is independent of how the objects are created (which concrete classes are instantiated).
- **AbstractFactory**
  - Defines an interface for creating a family of objects.
- **Factory1,...**
  - Implement the **AbstractFactory** interface by creating and returning the objects.

## Dynamic Object Collaboration

- In this sample scenario, a **Client** object delegates creating objects to a **Factory1** object. Let's assume that the **Client** is configured with a **Factory1** object.
- The interaction starts with the **Client** that calls `createProductA()` on the installed **Factory1** object.
- **Factory1** creates a **ProductA1** object and returns (a reference to) it to the **Client**.
- Thereafter, the **Client** calls `createProductB()` on **Factory1**.
- **Factory1** creates a **ProductB1** object and returns it to the **Client**.
- The **Client** can then use (perform an operation on) **ProductA1** and **ProductB1**.
- See also Sample Code / Example 1.

## Consequences



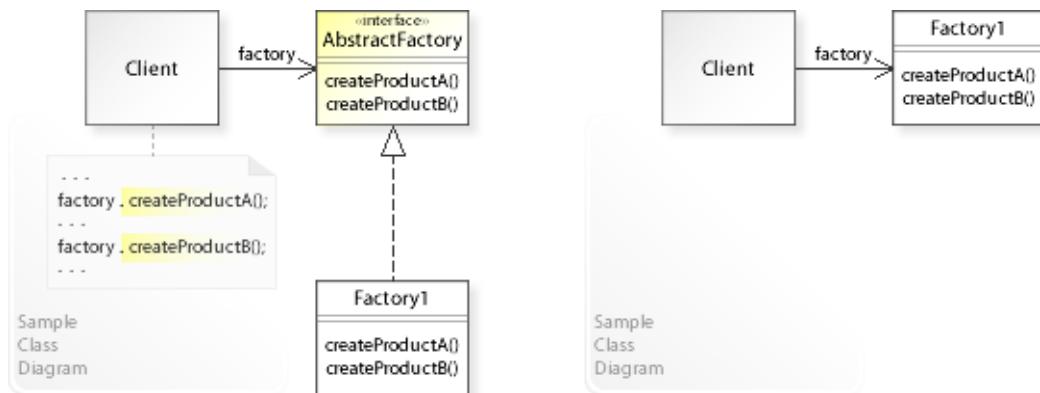
### Advantages (+)

- Avoids implementation dependencies.
  - Classes delegate object creation to a separate factory object and are independent of (do not know) which concrete classes are instantiated.
- Ensures creating consistent object families.
  - When an application supports creating multiple families of related objects, it must be ensured that a family of related objects is created and used together (see Sample Code / Example 3).
- Makes exchanging whole object families easy.
  - Because a factory object encapsulates creating a complete family of objects, the whole family can be exchanged by exchanging the factory object.

### Disadvantages (-)

- Requires extending the `Factory` interface to extend an object family.
  - The `Factory` interface must be extended to support new kinds of objects.
- Introduces an additional level of indirection.
  - The pattern achieves flexibility by introducing an additional level of indirection (classes delegate instantiation to a separate factory object), which makes the classes dependent on a factory object.

## Implementation



### Implementation Issues

#### Variant 1: Abstract Factory

Creating families of objects.

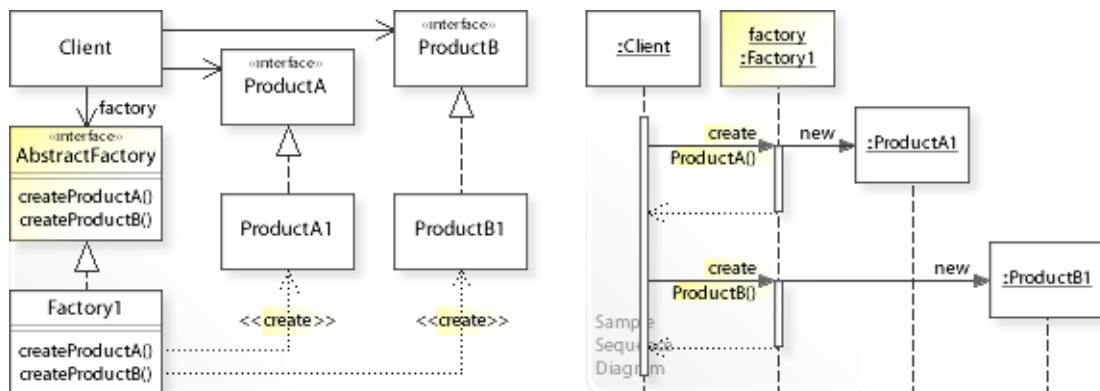
- Interface and implementation are cleanly separated.
- This is the way to implement the Abstract Factory pattern for applications that support creating families of related or dependent objects (see Sample Code / Example 1).
- "An application typically needs only one instance of a ConcreteFactory per product family. So it's usually best implemented as a Singleton(127)." [GoF, p90] See Sample Code / Example 3.

#### Variant 2: Concrete Factory

Creating objects.

- Interface and implementation are not cleanly separated.
- The concrete `Factory1` class acts as both the interface and the implementation (it abstracts and implements object creation).
- This is a common way to implement the Abstract Factory pattern for applications that do not need to create families of objects but want to be independent of how their objects are created (see Sample Code / Example 2).
- "Also note that `MazeFactory` is not an abstract class; thus it acts as both the `AbstractFactory` and the `ConcreteFactory`. This is another common implementation for simple applications of the Abstract Factory pattern." [GoF, p94]
- "Notice that the [concrete] `MazeFactory` is just a collection of factory methods. This is the most common way to implement the Abstract Factory pattern." [GoF, p94]

## Sample Code 1



### Basic Java code for implementing the sample UML diagrams.

```

1 package com.sample.abstractfactory.basic1;
2 public class MyApp {
3     public static void main(String[] args) {
4         // Creating a Client object
5         // and configuring it with a Factory1 object.
6         Client client = new Client(new Factory1());
7
8         System.out.println(client.operation());
9     }
10 }

Hello World from Client!
Using Factory1 to create (a family of) objects.
ProductA1 ProductB1 objects created.

1 package com.sample.abstractfactory.basic1;
2 public class Client {
3     private ProductA productA;
4     private ProductB productB;
5     private AbstractFactory factory;
6
7     public Client(AbstractFactory factory) {
8         this.factory = factory;
9     }
10    public String operation() {
11        productA = factory.createProductA();
12        productB = factory.createProductB();
13        return "Hello World from Client!\n"
14            + "Using " + factory.getClass().getSimpleName()
15            + " to create (a family of) objects.\n"
16            + productA.getName() + productB.getName() + "objects created.";
17    }
18 }

1 package com.sample.abstractfactory.basic1;
2 public interface AbstractFactory {
3     ProductA createProductA();
4     ProductB createProductB();
5 }

1 package com.sample.abstractfactory.basic1;
2 public class Factory1 implements AbstractFactory {
3     public ProductA createProductA() {
4         return new ProductA1();
5     }
6     public ProductB createProductB() {
7         return new ProductB1();
8     }
9 }

1 package com.sample.abstractfactory.basic1;
2 public class Factory2 implements AbstractFactory {
3     public ProductA createProductA() {

```

```
4         return new ProductA2();
5     }
6     public ProductB createProductB() {
7         return new ProductB2();
8     }
9 }

*****
Product inheritance hierarchy.
*****
```

```
1 package com.sample.abstractfactory.basic1;
2 public interface ProductA {
3     String getName();
4 }

1 package com.sample.abstractfactory.basic1;
2 public class ProductA1 implements ProductA {
3     public String getName() {
4         return "ProductA1 ";
5     }
6 }

1 package com.sample.abstractfactory.basic1;
2 public class ProductA2 implements ProductA {
3     public String getName() {
4         return "ProductA2";
5     }
6 }

1 package com.sample.abstractfactory.basic1;
2 public interface ProductB {
3     String getName();
4 }

1 package com.sample.abstractfactory.basic1;
2 public class ProductB1 implements ProductB {
3     public String getName() {
4         return "ProductB1 ";
5     }
6 }

1 package com.sample.abstractfactory.basic1;
2 public class ProductB2 implements ProductB {
3     public String getName() {
4         return "ProductB2";
5     }
6 }

*****
Unit test classes.
*****
```

```
1 package com.sample.abstractfactory.basic1;
2 import junit.framework.TestCase;
3 public class ClientTest extends TestCase {
4     // Creating a Client object
5     // and configuring it with a mock factory.
6     Client client = new Client (new FactoryMock());
7
8     public void testOperation() {
9         assertEquals("Hello World from Client!\n"
10             + "Using FactoryMock to create (a family of) objects.\n"
11             + "ProductAMock ProductBMock objects created.",
12             client.operation());
13     }
14     // More tests ...
15 }

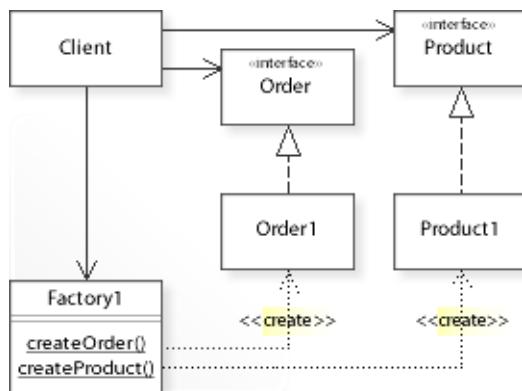
1 package com.sample.abstractfactory.basic1;
2 public class FactoryMock implements AbstractFactory {
3     public ProductA createProductA() {
4         return new ProductAMock();
5     }
6     public ProductB createProductB() {
7         return new ProductBMock();
```

```
8      }
9 }

1 package com.sample.abstractfactory.basic1;
2 public class ProductAMock implements ProductA {
3     public String getName() {
4         return "ProductAMock ";
5     }
6 }

1 package com.sample.abstractfactory.basic1;
2 public class ProductBMock implements ProductB {
3     public String getName() {
4         return "ProductBMock ";
5     }
6 }
```

## Sample Code 2



### Concrete Factory with static factory methods.

For applications that do not need to create families of objects but want to separate and centralize their object creation.

```

1 package com.sample.abstractfactory.basic2;
2 public class Client {
3     // Running the Client class as application.
4     public static void main(String[] args) {
5
6         System.out.println("Creating an order object:");
7         Factory1.createOrder();
8
9         System.out.println("Creating a product object:");
10        Factory1.createProduct();
11    }
12 }

Creating an order object:
Order1 object created.
Creating a product object:
Product1 object created.
  
```

```

1 package com.sample.abstractfactory.basic2;
2 public class Factory1 {
3     public static Order createOrder() {
4         System.out.println(" Order1 object created.");
5         return new Order1();
6     }
7     public static Product createProduct() {
8         System.out.println(" Product1 object created.");
9         return new Product1();
10    }
11 }

*****
Order and Product hierarchies.
*****
  
```

```

1 package com.sample.abstractfactory.basic2;
2 public interface Order {
3     // ...
4 }

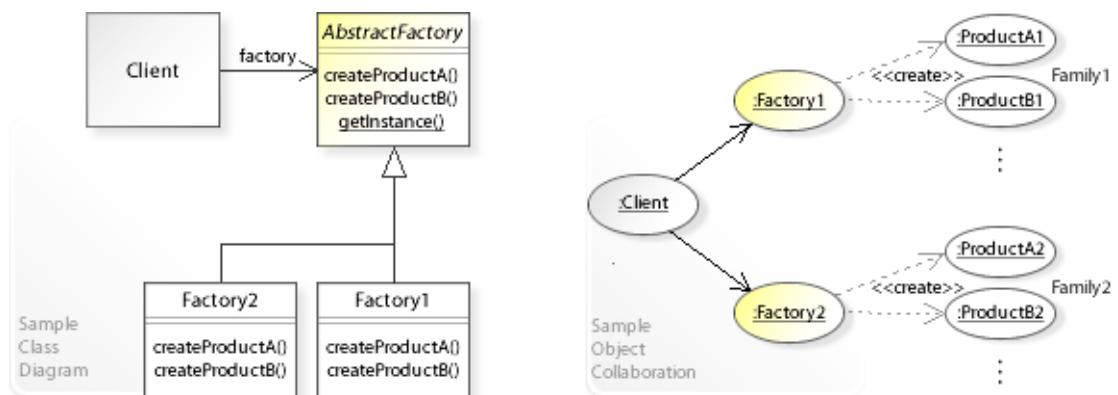
1 package com.sample.abstractfactory.basic2;
2 public class Order1 implements Order {
3     // ...
4 }

1 package com.sample.abstractfactory.basic2;
2 public interface Product {
3     // ...
4 }

1 package com.sample.abstractfactory.basic2;
  
```

```
2  public class Product1 implements Product {  
3      // ...  
4 }
```

### Sample Code 3



**Ensuring that a family of related or dependent objects is created and used together.**

```

1 package com.sample.abstractfactory.basic3;
2 public class Client {
3     // Running the Client class as application.
4     public static void main(String[] args) {
5         // Getting a factory object.
6         AbstractFactory factory = AbstractFactory.getInstance();
7
8         System.out.println("Creating a family of objects:");
9         factory.createProductA();
10        factory.createProductB();
11        System.out.println("Family of objects created.");
12    }
13 }

Creating a family of objects:
creating a ProductA1 object ...
creating a ProductB1 object ...
Family of objects created.

```

```

1 package com.sample.abstractfactory.basic3;
2 public abstract class AbstractFactory {
3     // See also Singleton / Implementation / Variant 2.
4     private static AbstractFactory factory;
5     public static final AbstractFactory getInstance() {
6         // Deciding which factory to use.
7         if (factory == null) {
8             factory = new Factory1();
9         }
10        return factory;
11    }
12    //
13    public abstract ProductA createProductA();
14    public abstract ProductB createProductB();
15    //
16    // Factory subclasses are implemented as private static nested classes
17    // to ensure that clients can't instantiate them directly.
18    //
19    private static class Factory1 extends AbstractFactory { // Family1
20        public ProductA createProductA() {
21            System.out.println(" creating a ProductA1 object ...");
22            return new ProductA1();
23        }
24        public ProductB createProductB() {
25            System.out.println(" creating a ProductB1 object ...");
26            return new ProductB1();
27        }
28    }
29
30    private static class Factory2 extends AbstractFactory { // Family2
31        public ProductA createProductA() {
32            System.out.println(" creating a ProductA2 object ...");
33            return new ProductA2();
34        }

```

```
35         public ProductB createProductB() {
36             System.out.println(" creating a ProductB2 object ...");
37             return new ProductB2();
38         }
39     }
40 }
```

\*\*\*\*\*  
Product inheritance hierarchy.  
\*\*\*\*\*

```
1 package com.sample.abstractfactory.basic3;
2 public interface ProductA {
3     // ...
4 }

1 package com.sample.abstractfactory.basic3;
2 public class ProductA1 implements ProductA {
3     // ...
4 }

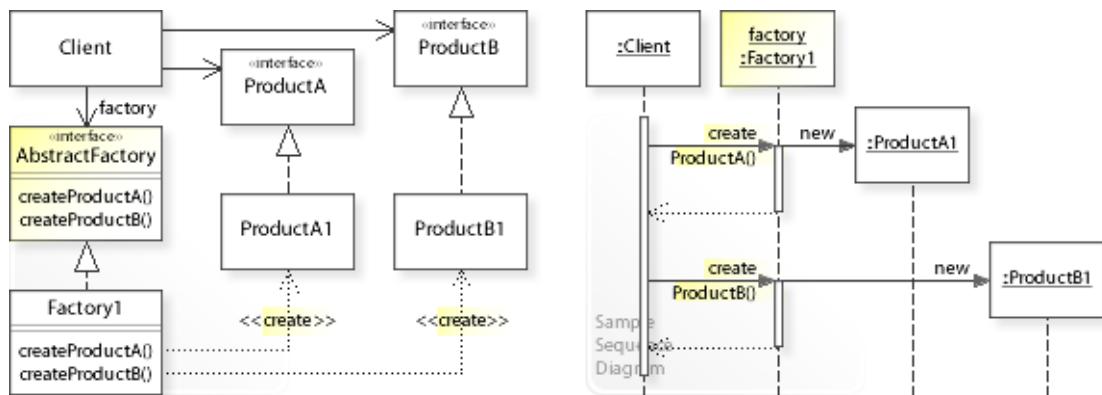
1 package com.sample.abstractfactory.basic3;
2 public class ProductA2 implements ProductA {
3     // ...
4 }

1 package com.sample.abstractfactory.basic3;
2 public interface ProductB {
3     // ...
4 }

1 package com.sample.abstractfactory.basic3;
2 public class ProductB1 implements ProductB {
3     // ...
4 }

1 package com.sample.abstractfactory.basic3;
2 public class ProductB2 implements ProductB {
3     // ...
4 }
```

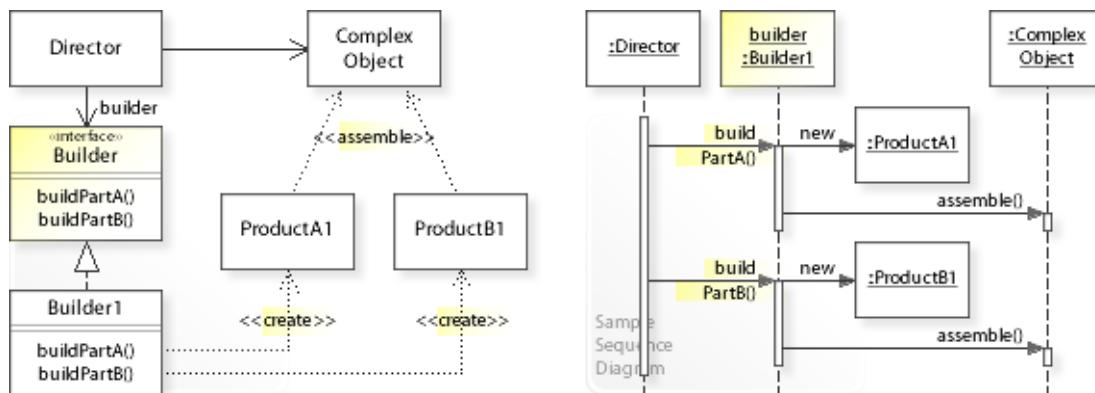
## Related Patterns



### Key Relationships (see also Overview)

- **Abstract Factory - Dependency Injection**
  - Abstract Factory
    - A class requests the objects it requires from a factory object.
    - A class must know its factory.
  - Dependency Injection
    - A class accepts the objects it requires from an injector object.
    - A class has no knowledge of its injector.
- **Abstract Factory - Factory Method**
  - Abstract Factory
    - Defines a separate factory object for creating objects.
  - Factory Method
    - Defines a separate factory method for creating an object.
- **Strategy - Abstract Factory**
  - Strategy
    - A class delegates an algorithm to a strategy object.
  - Abstract Factory
    - A class delegates instantiation to a factory object.

## Intent



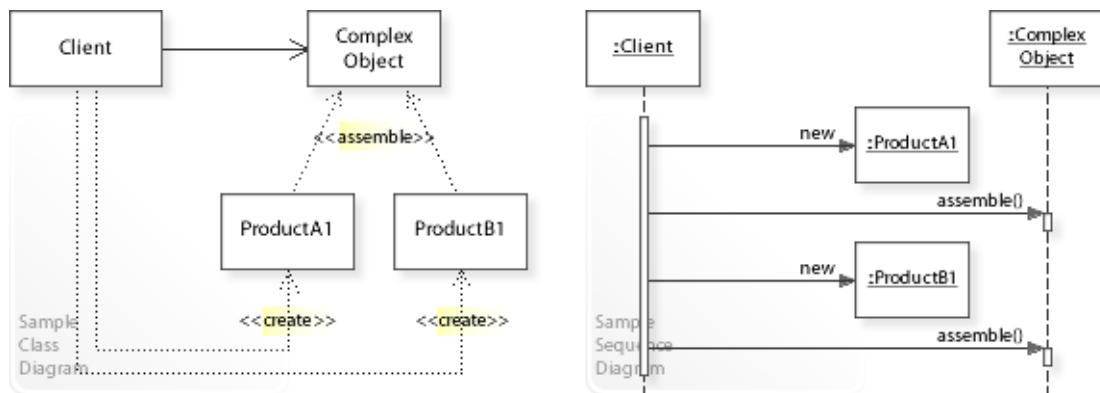
The intent of the Builder design pattern is to:

**"Separate the construction of a complex object from its representation so that the same construction process can create different representations."** [GoF]

See Problem and Solution sections for a more structured description of the intent.

- The Builder design pattern solves problems like:
  - *How can a class (the same construction process) create different representations of a complex object?*
- Creating a complex object directly within a class makes it impossible to change the instantiation (create a different representation) independently from (without having to change) the class.
- For example, creating a bill of materials object (BOM).  
It should be possible that a class (the same construction process) can create different product structures (representations) of the BOM.
- The Builder pattern describes how to solve such problems:
  - *Separate the construction of a complex object from its representation - encapsulate creating a complex object in a separate **Builder** object.*
  - A class can create different representations of a complex object by delegating to different **Builder** objects.

## Problem



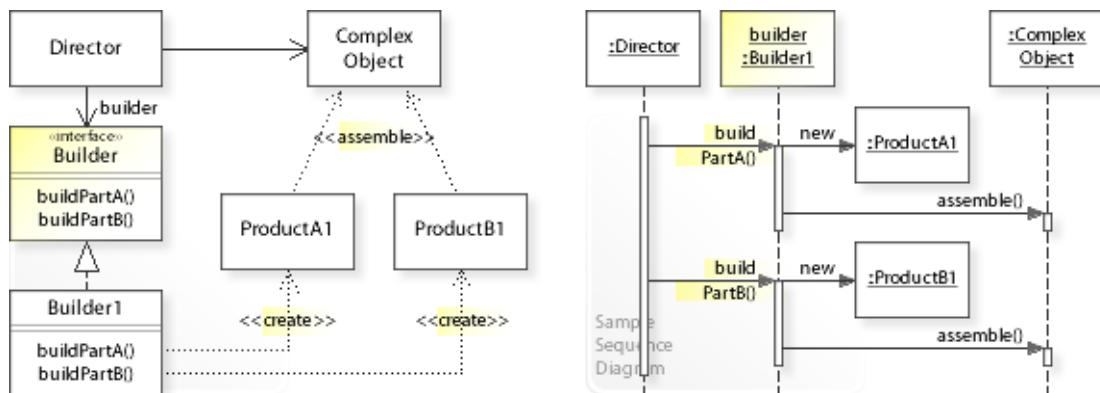
The Builder design pattern solves problems like:

***How can a class (the same construction process)  
create different representations of a complex object?***

See Applicability section for all problems Builder can solve.

- An inflexible way to create a complex object is to instantiate concrete classes (`new ProductA1()`, add to complex object, `new ProductB1()`, add to complex object, ...) directly within a class (`Client`).
- This commits the class to creating a particular representation of the complex object (`ProductA1`, `ProductB1`), which makes it impossible to create a different representation (`ProductA2`, `ProductB2`, for example) independently from (without having to change) the class.
- *That's the kind of approach to avoid if we want that a class (the same construction process) can create different representation of a complex object.*
- For example, creating a bill of materials object (BOM).  
A bill of materials is organized into a part-whole hierarchy (see also Composite for representing a part-whole hierarchy). It describes the parts that make up a manufactured product and how they are assembled.  
It should be possible that a class can create different product structures (representations) of the BOM.

## Solution



The Builder pattern describes a solution:

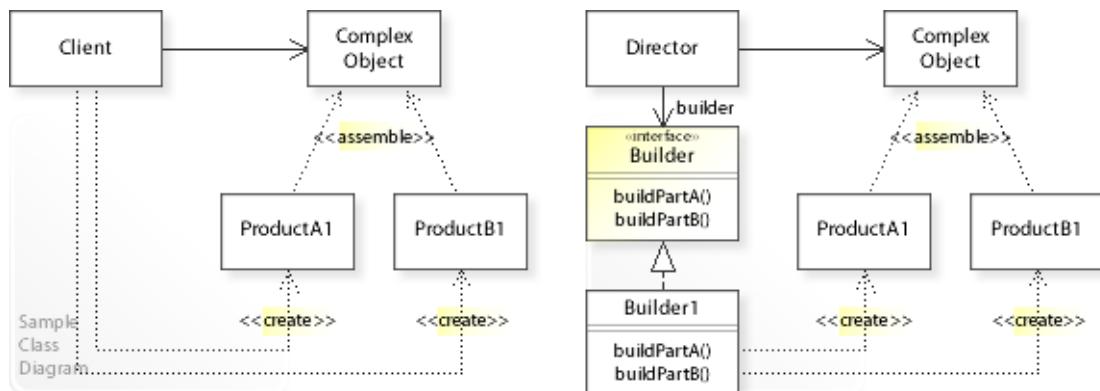
**Encapsulate creating a complex object in a separate `Builder` object.**

Describing the Builder design in more detail is the theme of the following sections.

See Applicability section for all problems Builder can solve.

- The key idea in this pattern is to encapsulate the creation of a complex object in a separate **Builder** object so that a class (the same construction process) can use different builders to create different representations of a complex object.
- **Define separate `Builder` objects:**
  - Define an interface for creating parts of a complex object (`Builder` | `buildPartA()`, `buildPartB()`, ...).
  - Define classes (`Builder1`, ...) that implement the interface.  
The complex object is created step by step so that clients (`Director`) have finer control over the creation process (which concrete classes are instantiated, i.e., how the complex object is represented).
- This enables *compile-time* flexibility (via inheritance).  
The way the parts of a complex object are created and assembled can be implemented and changed independently from clients by defining new (sub)classes.
- **Clients delegate the responsibility to create a complex object to a `Builder` object** (`builder.buildPartA()`, `builder.buildPartB()`, ...).
- This enables *run-time* flexibility (via object composition).  
Clients (`Director`) can use different `Builder` objects to create different representations of a complex object.

## Motivation 1



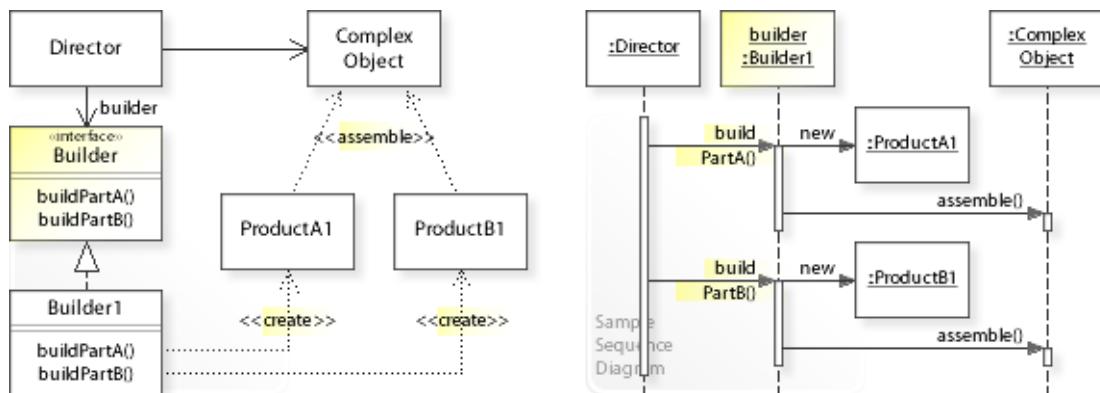
### Consider the left design (problem):

- Hard-wired object creation.
  - Creating a representation of a complex object is implemented (hard-wired) directly within a client class (**Client**).
  - This makes it hard to change the representation both at compile-time and at run-time.
- Complicated classes.
  - Classes that need creating a complex object get more complicated if they include the object creation.
  - This makes the classes harder to implement, change (maintain), and reuse.

### Consider the right design (solution):

- Encapsulated object creation.
  - Creating a representation of a complex object is implemented (encapsulated) in a separate class (**Builder1**).
  - This makes it easy to change the representation both at compile-time and at run-time.
- Simplified classes.
  - Classes that need creating a complex object get less complicated if they delegate the object creation.
  - This makes the classes easier to implement, change (maintain), and reuse.

## Applicability



## Design Problems

- **Creating Complex Objects**
  - How can a class be independent of how its objects are created?
  - How can a class (the same construction process) create different representations of a complex object?
  - How can a class delegate creating a complex object to another object (builder object)?
  - How can a class be configured with a builder object?

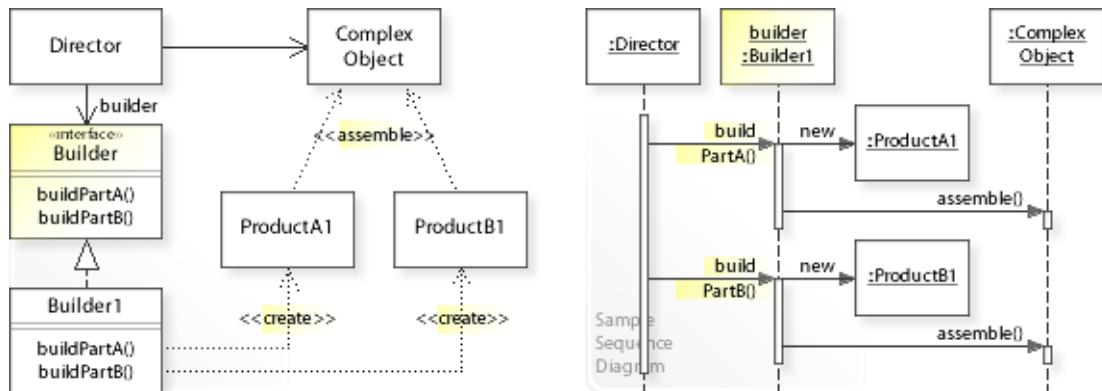
## Refactoring Problems

- **Complicated Code**
  - How can a class that includes creating a complex object be simplified?  
*Encapsulate Composite with Builder (96)* [JKerievsky05]

## Testing Problems

- **Unit Testing**
  - How can a class be configured with a mock builder instead of a real builder so that the class can be unit tested in isolation?

## Structure, Collaboration



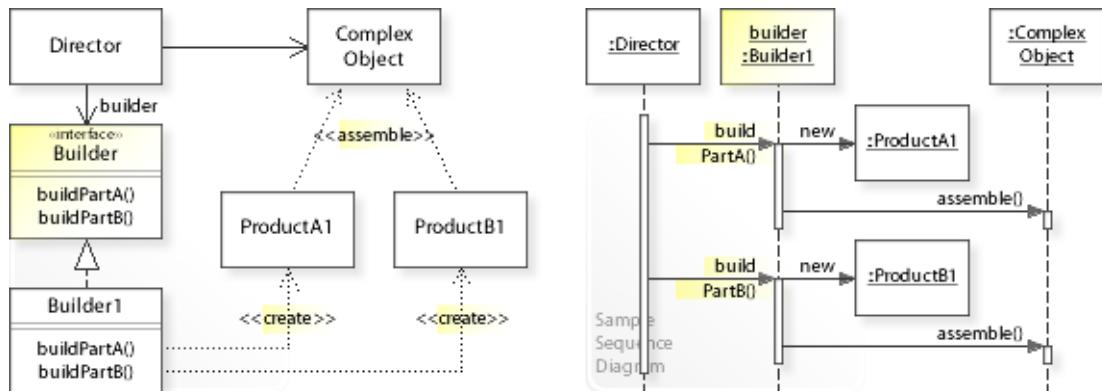
### Static Class Structure

- **Director**
  - Refers to the **Builder** interface to create and assemble the parts of a complex object and is independent of how the object is created (which representation is created).
- **Builder**
  - Defines an interface for creating parts of a complex object.
- **Builder1,...**
  - Implement the **Builder** interface.

### Dynamic Object Collaboration

- In this sample scenario, a **Director** object delegates creating and assembling the parts of a complex object to a **Builder1** object.  
Let's assume that the **Director** is configured with a **Builder1** object.
- The interaction starts with the **Director** that calls `buildPartA()` on the installed **Builder1** object.
- **Builder1** creates a **ProductA1** object and adds it to the **ComplexObject**.
- Thereafter, the **Director** calls `buildPartB()` on **Builder1**.
- **Builder1** creates a **ProductB1** object and adds it to the **ComplexObject**.
- The **Director** can then get the resulted **ComplexObject** from the **Builder1** and perform some work on it.
- See also Sample Code / Example 1.

## Consequences



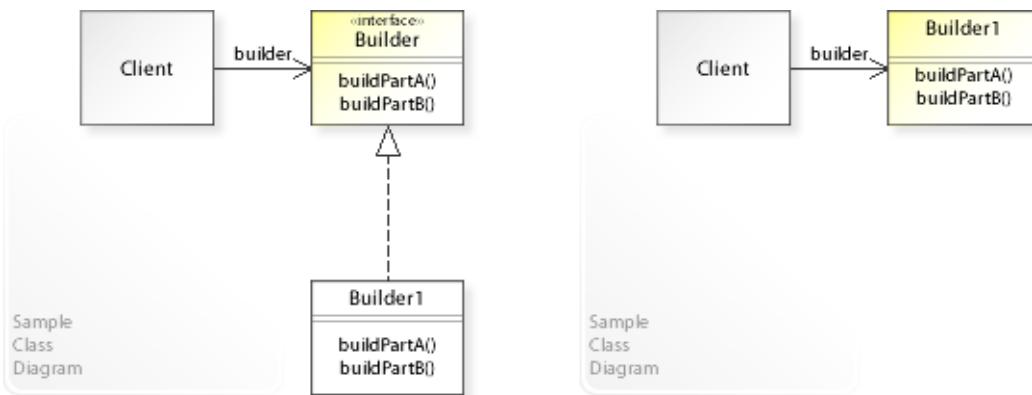
### Advantages (+)

- Avoids implementation dependencies.
  - Client classes do not instantiate concrete classes directly.
  - They delegate object creation to a separate builder object and are independent of which concrete classes are instantiated.
- Simplifies clients.
  - Because clients delegate creating a complex object to a builder object, they are easier to implement, change, and reuse.
- Makes creating different representations easy.
  - Because a builder object encapsulates creating and representing a complex object, different representations can be created by delegating to different builder objects.

### Disadvantages (-)

- Introduces an additional level of indirection.
  - The pattern achieves flexibility by introducing an additional level of indirection (delegating to separate builder objects).

## Implementation



### Implementation Issues

#### Variant 1: Abstract Builder

Creating different representations.

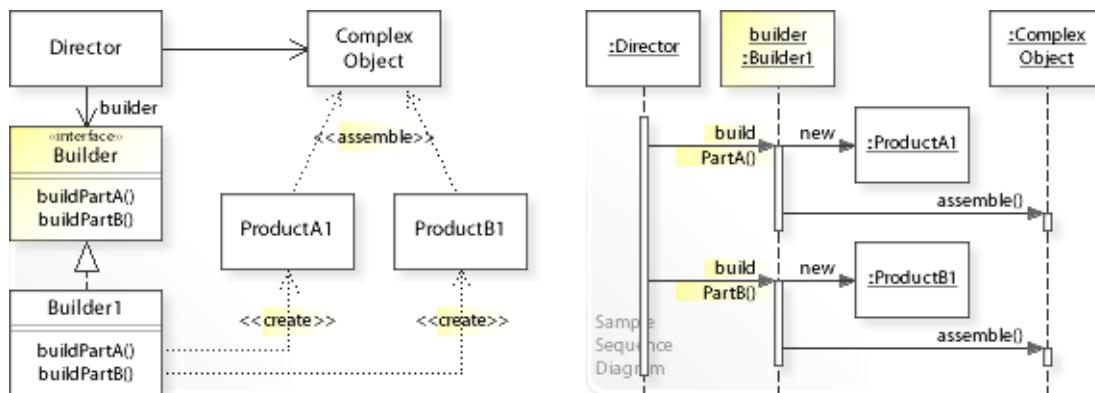
- Interface and implementation are cleanly separated.
- This is the way to implement the Builder pattern for applications that support creating different representations of a complex object.

#### Variant 2: Concrete Builder

Creating a single representation.

- Interface and implementation are not cleanly separated.
- The concrete builder class acts as both the interface and the implementation (it abstracts *and* implements creating a complex object).
- This is a common way to implement the Builder pattern for applications that do not need to create different representations but want to be independent of how a complex object is created.

## Sample Code 1



### Basic Java code for implementing the sample UML diagrams.

```

1 package com.sample.builder.basic;
2 public class MyApp {
3     public static void main(String[] args) {
4         // Creating a Director object
5         // and configuring it with a Builder1 object.
6         Director director = new Director(new Builder1());
7
8         System.out.println(director.construct());
9     }
10 }

Hello World from Director!
Using Builder1 to create a complex object step by step.
Complex Objetc made up of ProductA1 ProductB1 objects.

1 package com.sample.builder.basic;
2 public class Director {
3     private ComplexObject co;
4     private Builder builder;
5
6     public Director(Builder builder) {
7         this.builder = builder;
8     }
9     public String construct() {
10         builder.buildPartA();
11         builder.buildPartB();
12         co = builder.getResult();
13         return "Hello World from Director!\n"
14             + "Using " + builder.getClass().getSimpleName()
15             + " to create a complex object step by step.\n"
16             + co.getParts() + " objects.";
17     }
18 }

1 package com.sample.builder.basic;
2 public interface Builder {
3     void buildPartA();
4     void buildPartB();
5     ComplexObject getResult();
6 }

1 package com.sample.builder.basic;
2 public class Builder1 implements Builder {
3     private ComplexObject co = new ComplexObject();
4
5     public void buildPartA() {
6         co.add(new ProductA1());
7     }
8     public void buildPartB() {
9         co.add(new ProductB1());
10    }
11    public ComplexObject getResult() {
12        return co;
13    }
}

```

```
13      }
14 }

1 package com.sample.builder.basic;
2 import java.util.*;
3 public class ComplexObject {
4     private List<Product> children = new ArrayList<Product>();
5
6     public String getParts() {
7         Iterator<Product> i = children.iterator();
8         String str ="Complex Objetc made up of";
9         while (i.hasNext()) {
10             str += i.next().getName();
11         }
12         return str;
13     }
14     public boolean add(Product child) {
15         return children.add(child);
16     }
17     public Iterator<Product> iterator() {
18         return children.iterator();
19     }
20 }

*****
Product inheritance hierarchy.
*****

1 package com.sample.builder.basic;
2 public interface Product {
3     String getName();
4 }

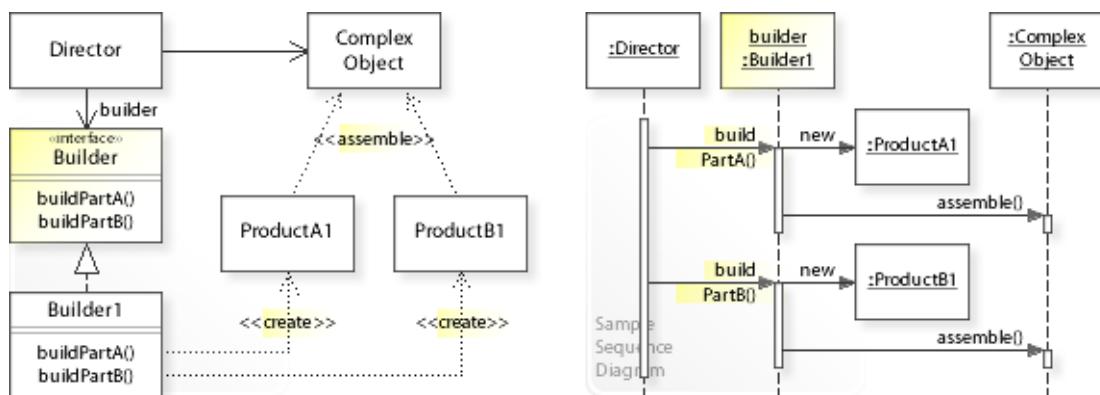
1 package com.sample.builder.basic;
2 public interface ProductA extends Product {
3     // ...
4 }

1 package com.sample.builder.basic;
2 public class ProductA1 implements ProductA {
3     public String getName() {
4         return " ProductA1";
5     }
6 }

1 package com.sample.builder.basic;
2 public interface ProductB extends Product {
3     // ...
4 }

1 package com.sample.builder.basic;
2 public class ProductB1 implements ProductB {
3     public String getName() {
4         return " ProductB1";
5     }
6 }
```

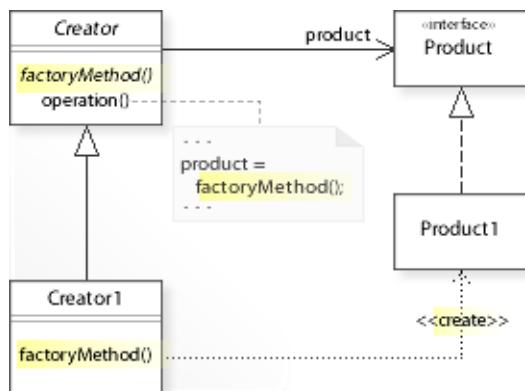
## Related Patterns



### Key Relationships (see also Overview)

- **Composite - Builder - Iterator - Visitor - Interpreter**
  - Composite provides a way to represent a part-whole hierarchy as a tree (composite) object structure.
  - Builder provides a way to create the elements of an object structure.
  - Iterator provides a way to traverse the elements of an object structure.
  - Visitor provides a way to define new operations for the elements of an object structure.
  - Interpreter represents a sentence in a simple language as a tree (composite) object structure (abstract syntax tree).

## Intent

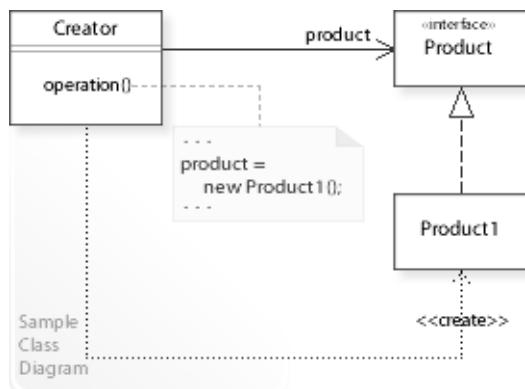


The intent of the Factory Method design pattern is to:

**"Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses."** [GoF]  
 See Problem and Solution sections for a more structured description of the intent.

- The Factory Method design pattern solves problems like:
  - *How can an object be created  
so that subclasses can redefine which class to instantiate?*
  - *How can a class defer instantiation to subclasses?*
- For example, instantiating unforeseeable classes in a framework.  
 It should be possible that a class can create a default object (or completely defer instantiation to subclasses) so that users of the framework can write subclasses to instantiate the class they need.
- The Factory Method pattern describes how to solve such problems:
  - *Define an interface for creating an object,  
i.e., define a separate operation (factory method) for creating an object*
  - *but let subclasses decide which class to instantiate.*  
 so that subclasses can redefine which class to instantiate.

## Problem



The Factory Method design pattern solves problems like:

**How can an object be created**

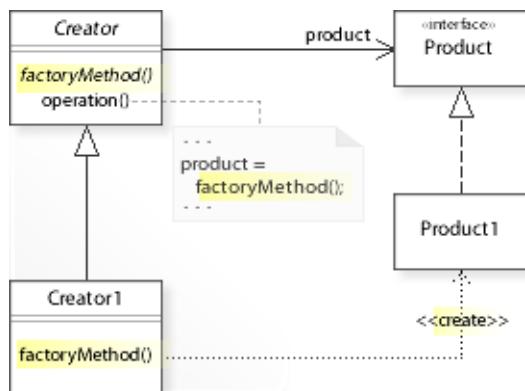
**so that subclasses can redefine which class to instantiate?**

**How can a class defer instantiation to subclasses?**

See Applicability section for all problems Factory Method can solve.

- An inflexible way to create an object is to instantiate a concrete class (`new Product1()`) directly within a class (**Creator**).
- This commits the class to a particular object and makes it impossible to change the instantiation independently from (without having to change) the class.
- *That's the kind of approach to avoid if we want to create an object so that subclasses can redefine which class to instantiate.*
- For example, simply to ensure that subclasses can change the way an object is created if necessary.
- For example, instantiating unforeseeable classes in a reusable application.  
It should be possible that a class can defer instantiation to subclasses so that users of the framework can define subclasses and specify the instantiation to suit their needs.

## Solution



The Factory Method pattern describes a solution:

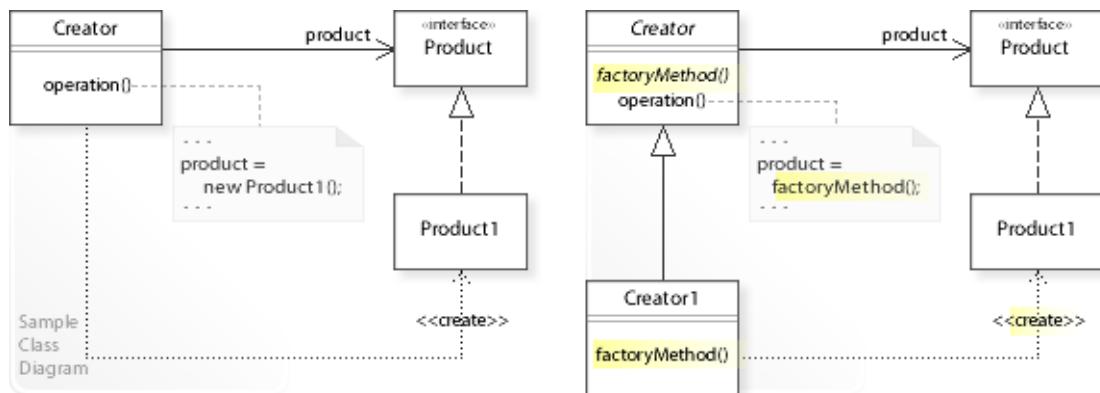
**Define a separate `factory method` for creating an object.**

Describing the Factory Method design in more detail is the theme of the following sections.

See Applicability section for all problems Factory Method can solve.

- The key idea in this pattern is to create an object in a separate operation so that subclasses can redefine which class to instantiate if necessary.
- The pattern calls an operation that is solely responsible for "manufacturing" an object a *factory method*. [GoF, p108]
- "People often use Factory Method as the standard way to create objects, but it isn't necessary when the class that's instantiated never changes [...]" [GoF, p136]
- Note that a common implementation of the Factory Method pattern is simply a concrete factory method that acts as both interface and implementation (it abstracts and implements the process of object creation; see also Implementation).
- Note the relationship between Template Method and Factory Method.  
A template method's primitive operation that is responsible for creating an object is a factory method.

## Motivation 1



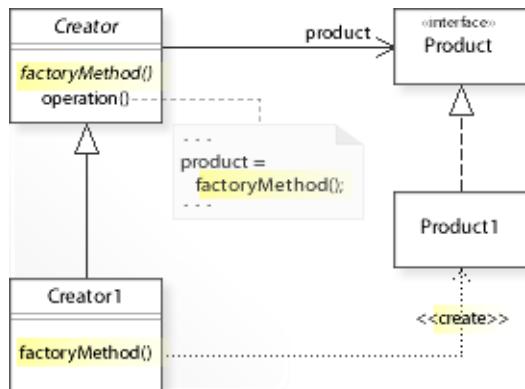
**Consider the left design (problem):**

- Hard-wired object creation.
  - Creating an object (`new Product1()`) is hard-wired directly within a class.
  - This makes it hard to change the instantiation independently from (without having to change) the class.
- Creator must specify which class to instantiate.
  - When designing a framework, we often need to create an object but do not know which class to instantiate.
  - Users of the framework should specify which class to instantiate to suit their needs (by writing subclasses).

**Consider the right design (solution):**

- Encapsulated object creation.
  - A separate factory method is defined for creating an object.
  - This makes it easy to change the instantiation independently from the class (by adding new subclasses).
- Subclasses can specify which class to instantiate.
  - Creator creates an object by calling a factory method that subclasses can redefine.
  - "It gets around the dilemma of having to instantiate unforeseeable classes." [GoF, p110]

## Applicability



## Design Problems

- **Creating Objects**
  - How can a class be independent of how its objects are created?
  - How can an object be created so that subclasses can redefine which class to instantiate?
  - How can a class defer instantiation to subclasses?
- **Flexible Alternative for Constructors**
  - How can a flexible alternative be provided for direct constructor calls?

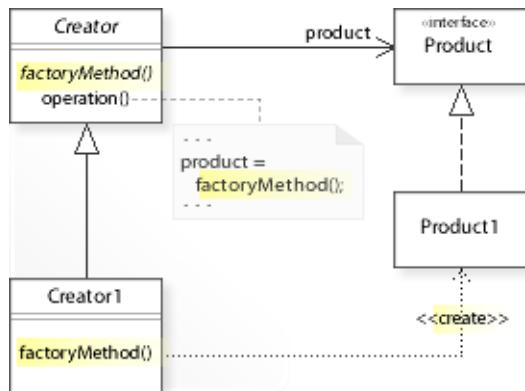
## Refactoring Problems

- **Unclear Code**
  - How can multiple constructors of a class that differ only in their arguments be named differently to avoid unclear code?  
*Replace Constructors with Creation Methods (57) [JKerievsky05]*

## Background Information

- Inflexible constructor names can cause unclear code.
  - In most languages, the constructor of a class must be named after the class.  
If a class has multiple constructors, they all must have the same name, which makes it hard to distinguish them and call the right one.
  - Factory methods can be named freely to clearly communicate their intent.
- "Consider static factory methods instead of constructors." [JBloch08, Item 1]  
See also Implementation.
- Refactoring and "Bad Smells in Code" [MFowler99] [JKerievsky05]
  - *Code smells* are certain structures in the code that "smell bad" and indicate problems that can be solved by a refactoring.
  - The most common code smells are:  
*complicated code* (including complicated/growing conditional code),  
*duplicated code*,  
*inflexible code* (that must be changed whenever requirements change), and  
*unclear code* (that doesn't clearly communicate its intent).

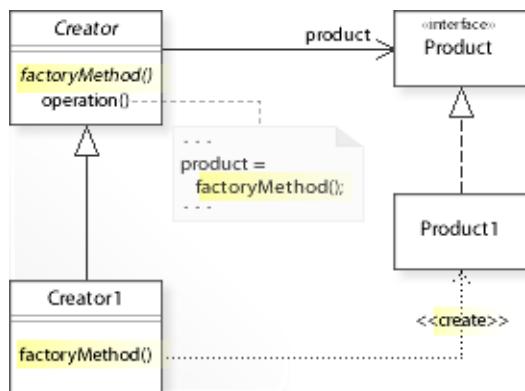
## Structure, Collaboration



## Static Class Structure

- *Creator*
  - Defines an abstract factory method (`factoryMethod()`) for creating an object.
  - May define a default implementation of the factory method (see Implementation).
  - May call the factory method (`Product product = factoryMethod()`); clients from outside the *Creator* may also call the factory method.
- *Creator1, ...*
  - Implement the factory method.
  - See Implementation for the two main implementation variants of the Factory Method pattern.

## Consequences



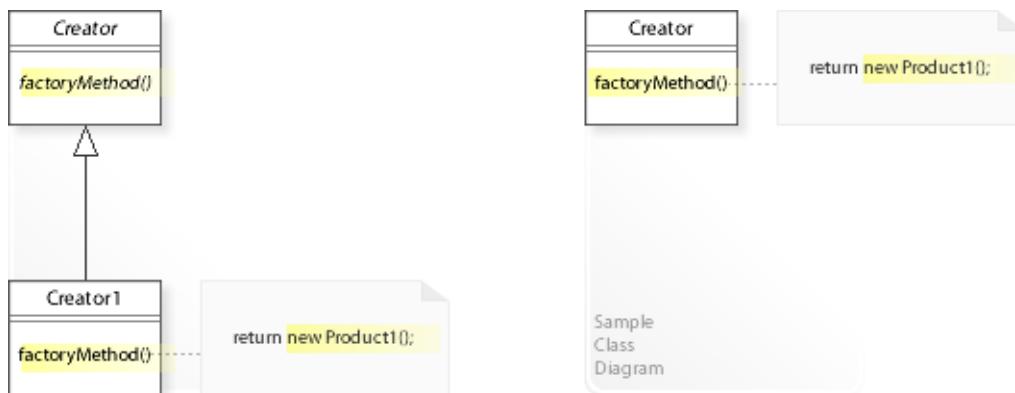
### Advantages (+)

- Avoids implementation dependencies.
  - Creator classes do not instantiate concrete classes directly.
  - They create objects by calling a factory method (that subclasses can redefine) and are independent of which concrete classes are instantiated.

### Disadvantages (-)

- May require adding many subclasses.
  - New subclasses may have to be added to change the way objects are created.
  - Subclassing is fine when a class hierarchy already exists.

## Implementation



## Implementation Issues

### Variant 1: Abstract Factory Method

- The factory method is abstract and subclasses must provide an implementation.
- "The first case [abstract factory method] *requires* subclasses to define an implementation, because there's no reasonable default. It gets around the dilemma of having to instantiate unforeseeable classes." [GoF, p110]

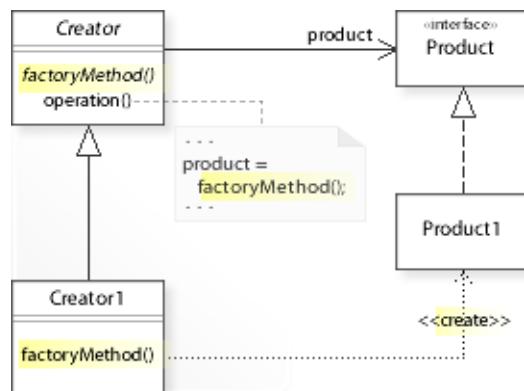
### Variant 2: Concrete Factory Method

- The factory method is concrete and provides a (default) implementation.
- The concrete factory method acts as both interface and implementation (it abstracts *and* implements the process of creating an object).
- This is a common way to implement the Factory Method pattern for classes that do *not require* to defer instantiation to subclasses but want to be independent of how their objects are created (so that subclasses can be added to redefine the instantiation if necessary).
- "In the second case [concrete factory method], the concrete Creator uses the factory method primarily for flexibility. It's following a rule that says, "Create objects in a separate operation so that subclasses can override the way they're created." This rule ensures that designers of subclasses can change the class of objects their parent class instantiates if necessary." [GoF, p110]
- "Consider static factory methods instead of constructors." [JBloch08, Item 1] [see BGI]

## Background Information

- Static Factory Methods
  - Static factory methods are widely used when flexibility is needed but overriding via subclassing is not needed.
  - In most languages, the constructor of a class must be named after the class.  
If a class has multiple constructors, they all must have the same name, which causes unclear code. Factory methods can be named freely to clearly communicate their intent.
  - Static factory methods can be accessed easily (via class name and operation name), and their classes can control what instances exist at any time (for example, to avoid creating unnecessary or duplicate objects, to cache objects as they are created [see also Flyweight], or to guarantee to create only a single object [see also Singleton]).
  - "A second advantage of static factory methods is that, unlike constructors, they are not required to create a new object each time they're invoked." [JBloch08, Item 1]

## Sample Code 1



**Basic Java code for implementing the sample UML diagrams.**

```
1 package com.sample.factorymethod.basic;
2 public class MyApp {
3     public static void main(String[] args) {
4
5         // Creating a Creator1 object.
6         Creator creator = new Creator1();
7
8         System.out.println(creator.operation());
9     }
10 }
```

Hello World from Creator1!

Product1 instantiated.

```
1 package com.sample.factorymethod.basic;
2 public abstract class Creator {
3     private Product product;
4
5     public abstract Product factoryMethod();
6
7     public String operation() {
8         product = factoryMethod();
9         return "Hello World from "
10            + this.getClass().getSimpleName() + "!\n";
11            + product.getName() + " instantiated.";
12    }
13 }
```

```
1 package com.sample.factorymethod.basic;
2 public class Creator1 extends Creator {
3     public Product factoryMethod() {
4         return new Product1();
5     }
6 }
```

\*\*\*\*\*

Product inheritance hierarchy.

\*\*\*\*\*

```
1 package com.sample.factorymethod.basic;
2 public interface Product {
3     String getName();
4 }
```

```
1 package com.sample.factorymethod.basic;
2 public class Product1 implements Product {
3     public String getName() {
4         return "Product1";
5     }
6 }
```

```
1 package com.sample.factorymethod.basic;
2 public class Product2 implements Product {
3     public String getName() {
4 }
```

```
4           return "Product2";
5       }
6 }
```

## Sample Code 2



### Basic Java code for implementing static factory methods.

```

1 package com.sample.factorymethod.staticFM;
2 public class MyApp {
3     public static void main(String[] args) {
4
5         // Calling static factory methods.
6         System.out.println(Creator1.factoryMethod().getName() + " instantiated.");
7
8         System.out.println(Creator2.factoryMethod().getName() + " instantiated.");
9     }
10 }

Product1 instantiated.
Product2 instantiated.

1 package com.sample.factorymethod.staticFM;
2 public class Creator1 {
3     // Static factory method.
4     public static Product factoryMethod() {
5         return new Product1();
6     }
7 }

1 package com.sample.factorymethod.staticFM;
2 public class Creator2 extends Creator1 {
3     // Static methods can't be overridden by subclasses.
4     public static Product factoryMethod() {
5         return new Product2();
6     }
7 }

*****
Product inheritance hierarchy.
*****

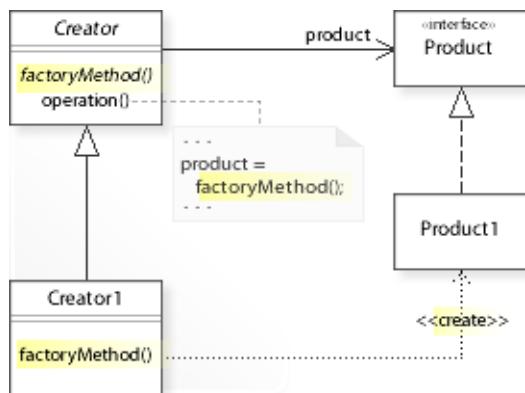
1 package com.sample.factorymethod.staticFM;
2 public interface Product {
3     String getName();
4 }

1 package com.sample.factorymethod.staticFM;
2 public class Product1 implements Product {
3     public String getName() {
4         return "Product1";
5     }
6 }

1 package com.sample.factorymethod.staticFM;
2 public class Product2 implements Product {
3     public String getName() {
4         return "Product2";
5     }
6 }

```

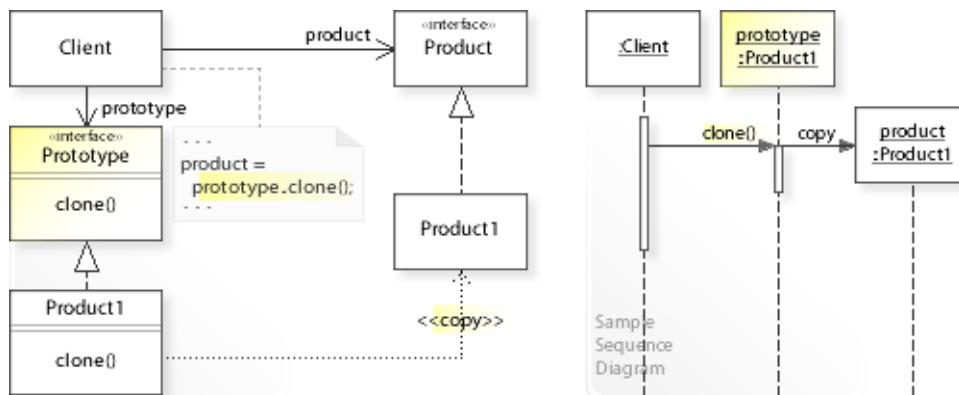
## Related Patterns



### Key Relationships (see also Overview)

- **Abstract Factory - Factory Method**
  - Abstract Factory  
Defines a separate factory object for creating objects.
  - Factory Method  
Defines a separate factory method for creating an object.
- **Factory Method - Prototype**
  - Factory Method uses subclasses to specify which class to instantiate statically at compile-time.
  - Prototype uses prototypes to specify the kinds of objects to create dynamically at run-time.  
For example, Abstract Factory can be implemented by using prototypes instead of factory methods (see Prototype / Sample Code / Example 2).
- **Iterator - Factory Method**
  - The operation for creating an iterator object is a factory method.
- **Template Method - Factory Method**
  - A template method's primitive operation that is responsible for creating an object is a factory method.

## Intent



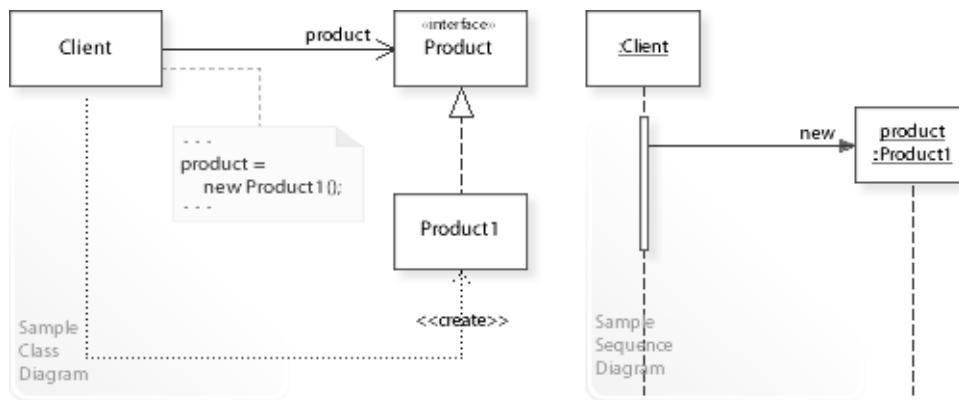
The intent of the Prototype design pattern is to:

**"Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype."** [GoF]

See Problem and Solution sections for a more structured description of the intent.

- The Prototype design pattern solves problems like:
  - *How can the kinds of objects to create be specified at run-time?*
- Often the kinds of objects to create aren't known at compile-time.  
It should be possible to specify them dynamically at run-time.
- The Prototype pattern describes how to solve such problems:
  - *Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.*
  - To act as a *prototype*, an object must implement the `Prototype` interface (`clone()`) for copying itself.
  - For example, a `Product1` object that implements the `clone()` operation can act as a prototype for creating `Product1` objects.

## Problem



The Prototype design pattern solves problems like:

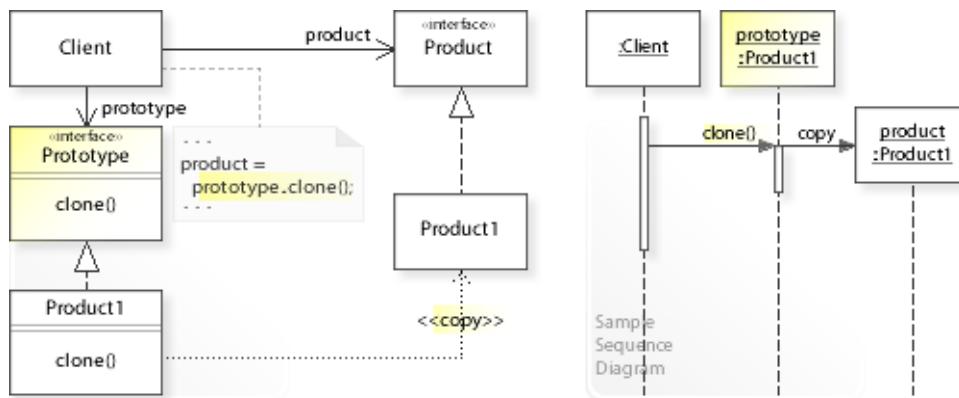
***How can the kinds of objects to create be specified at run-time?***

***How can classes be instantiated that are specified at run-time?***

See Applicability section for all problems Prototype can solve.

- An inflexible way to create an object is to instantiate a concrete class (`new Product1()` in the above example) directly within the class that requires the object (`Client`).
- This specifies which class to instantiate (`Product1`) statically at compile-time.
- *That's the kind of approach to avoid if we want to specify the kinds of objects to create (which classes to instantiate) dynamically at run-time.*
- For example, instantiating unforeseeable classes in a reusable application.  
It should be possible to create objects so that users of the framework can specify the kind of objects to create at run-time to suit their needs.
- For example, instantiating dynamically loaded classes  
(that are not known at compile-time).

## Solution



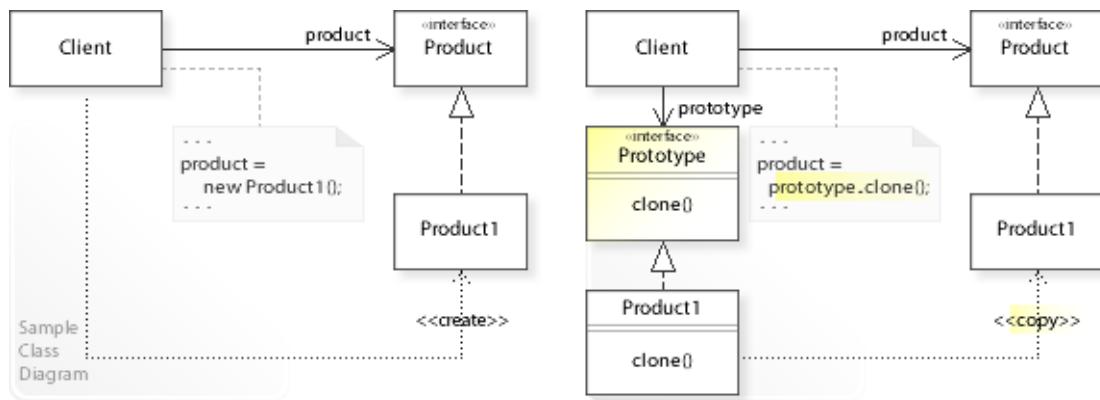
The Prototype pattern describes a solution:

**Specify the kinds of objects to create by using prototypical instances, and create new objects by copying these prototypes.**

Describing the Prototype design in more detail is the theme of the following sections.  
See Applicability section for all problems Prototype can solve.

- The key idea in this pattern is to create new objects by copying existing objects.
- **Define Prototype objects:**
  - To act as a *prototype*, an object must implement the `Prototype` interface (`clone()`) for copying itself.
  - For example, a `Product1` object that implements the `clone()` operation can act as a prototype for creating `Product1` objects.
- **Create new objects by copying Prototype objects**  
(`product = prototype.clone()`).
- This enables *run-time* flexibility (via object composition).  
Clients can be configured with different `Prototype` objects, which are then copied to create new objects, and even more, `Prototype` objects can be added, removed, and exchanged at run-time.

## Motivation 1



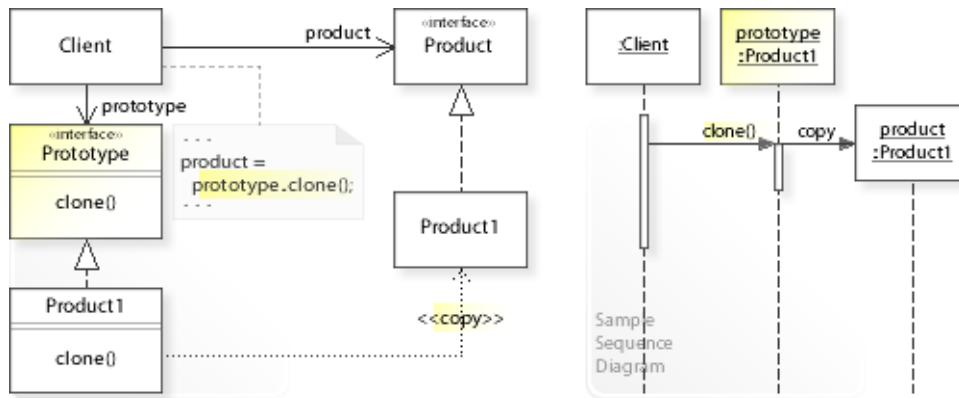
**Consider the left design (problem):**

- Statically specified object creation.
  - An object is created by specifying which class to instantiate (`new Product1()`) statically at compile-time.

**Consider the right design (solution):**

- Dynamically specified object creation.
  - An object is created by copying a prototype object (`prototype.clone()`) that can be specified (selected and changed) dynamically at run-time.

## Applicability



## Design Problems

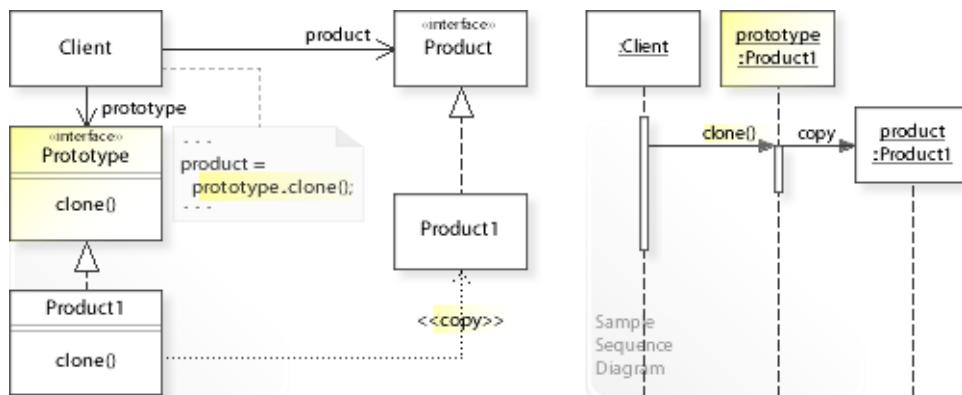
- **Creating Objects**

- How can a class be independent of how its objects are created?
- How can the kinds of objects to create can be specified at run-time?
- How can a class be configured with the kinds of objects to create?

- **Instantiating Dynamically Loaded Classes**

- How can classes be instantiated that are specified at run-time, for example, by dynamic loading?

## Structure, Collaboration



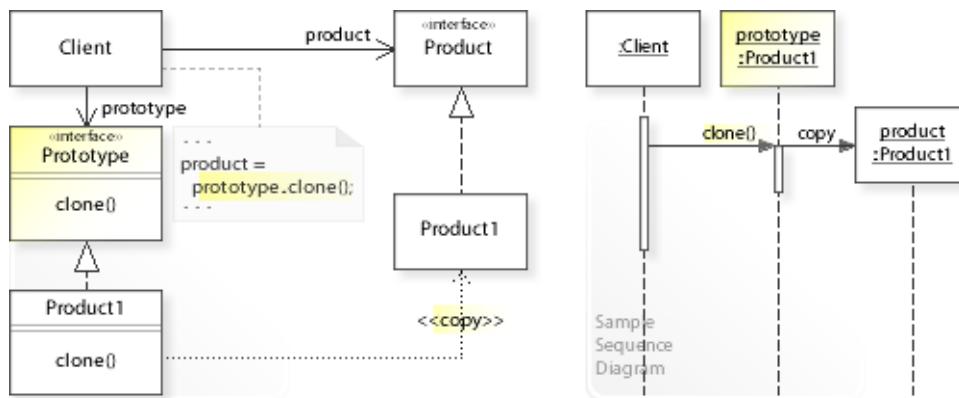
### Static Class Structure

- Client
  - Creates a new object by copying a **Prototype** object (`prototype.clone()`).
- Prototype
  - Defines an interface for copying itself (`clone()`).
- Product1, ...
  - Implement the **Prototype** and **Product** interfaces.
  - Any object that implements the **Prototype** interface (by returning a copy of itself) can act as a prototype.

### Dynamic Object Collaboration

- In this sample scenario, a **Client** object creates (clones) an object by calling `clone()` on a **Product1** object, which returns a copy of itself.
- The **Client** can then use (perform an operation on) the copied **Product1** object.
- See also Sample Code / Example 1.

## Consequences



### Advantages (+)

- Allows adding and removing prototypes dynamically at run-time.
  - "That's a bit more flexible than other creational patterns, because a client can install and remove prototypes at run-time." [GoF, p119]
- Allows instantiating dynamically loaded classes.
  - An instance of each dynamically loaded class is created automatically and can be stored in a registry of available prototypes.
  - "A client will ask the registry for a prototype before cloning it. We call this registry a **prototype manager**." [GoF, p121]
- Provides a flexible alternative to Factory Method.
  - Prototype doesn't need subclasses to redefine which class to instantiate.
  - Prototype will work wherever Factory Method will and with more flexibility.

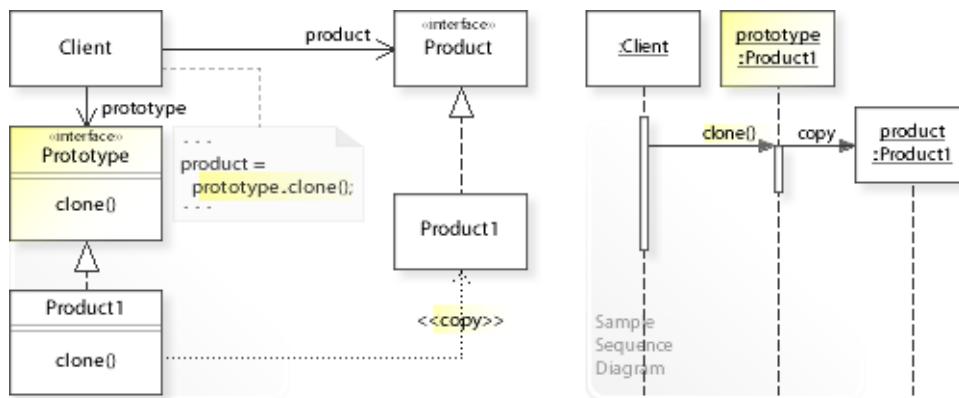
### Disadvantages (-)

- Can make the implementation of the clone operation difficult.
  - "The hardest part of the Prototype pattern is implementing the Clone operation correctly. It's particularly tricky when object structures contain circular references." [GoF, p121]
  - "Override clone judiciously" [JBloch08, Item 11]

### Background Information

- A Registry is
  - "A well-known object that other objects can use to find common objects and services." [MFowler03, Registry (480)]
- A Registry is often implemented as Singleton.

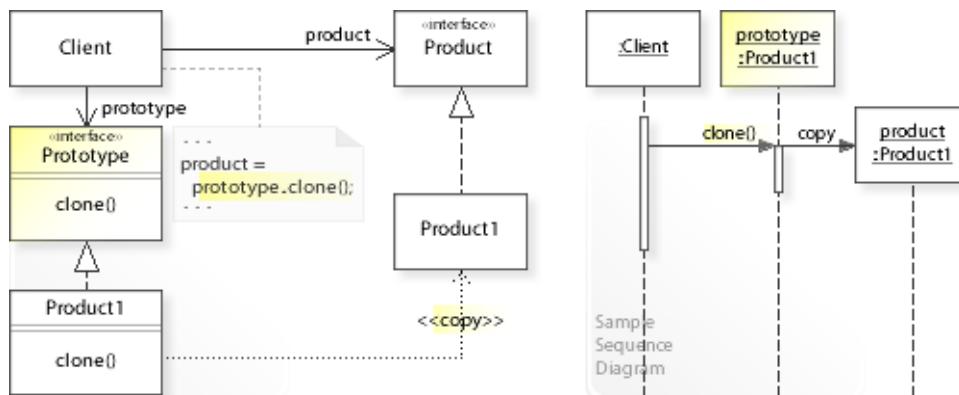
## Implementation



### Implementation Issues

- **Implementing the clone operation.**
  - "The hardest part of the Prototype pattern is implementing the Clone operation correctly. It's particularly tricky when object structures contain circular references." [GoF, p121]
  - "Override clone judiciously"  
[JBloch08, Item 11]

## Sample Code 1



**Basic Java code for implementing the sample UML diagrams.**

```

1 package com.sample.prototype.basic;
2 public class MyApp {
3     public static void main(String[] args) {
4         // Creating a Client object
5         // and configuring it with a Prototype object.
6         Client client = new Client(new Product1("Product1"));
7
8         System.out.println(client.operation());
9     }
10 }

Hello World from Client!
Cloning Product1.
Product1 object copied.

1 package com.sample.prototype.basic;
2 public class Client {
3     private Product product;
4     private Prototype prototype;
5
6     public Client(Prototype prototype) {
7         this.prototype = prototype;
8     }
9     public String operation() {
10         product = prototype.clone();
11         return "Hello World from Client!\n"
12             + "Cloning " + prototype.getClassName().getSimpleName() + ".\n"
13             + product.getName() + " object copied.";
14     }
15 }

*****
Product inheritance hierarchy.
*****

1 package com.sample.prototype.basic;
2 public interface Product {
3     String getName();
4 }

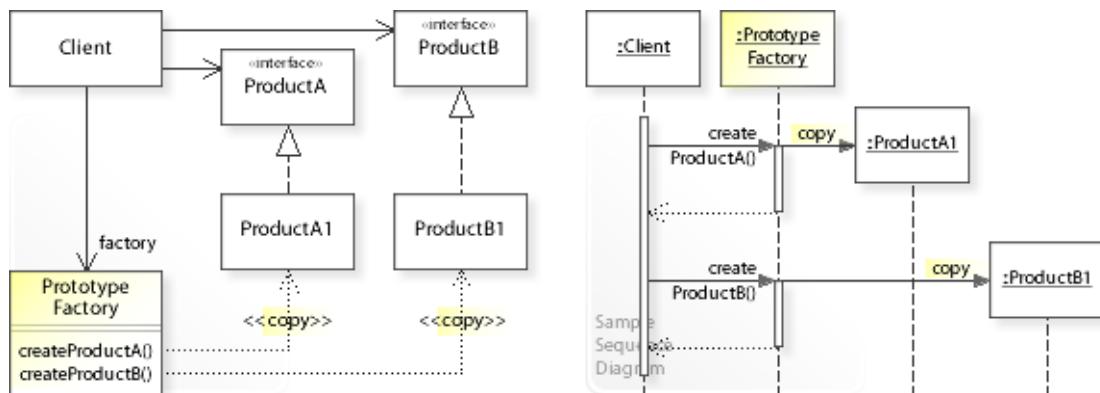
1 package com.sample.prototype.basic;
2 public interface Prototype {
3     Product clone();
4 }

1 package com.sample.prototype.basic;
2 public class Product1 implements Product, Prototype {
3     private String name;
4
5     public Product1(String name) {
6         this.name = name;
7     }

```

```
8     // Copy constructor needed by clone().
9     public Product1(Product1 p) {
10         this.name = p.getName();
11     }
12     @Override
13     public Product clone() {
14         return new Product1(this);
15     }
16     public String getName() {
17         return name;
18     }
19 }
```

## Sample Code 2



### Implementing Abstract Factory with prototypes instead of factory methods.

```

1 package com.sample.prototype.basicAF;
2 public class MyApp {
3     public static void main(String[] args) {
4         // Creating a Client object
5         // and configuring it with a PrototypeFactory object.
6         Client client = new Client(new PrototypeFactory(
7             new ProductA1("ProductA1"), new ProductB1("ProductB1") ));
8
9         System.out.println(client.operation());
10    }
11 }

Hello World from Client!
Using PrototypeFactory to create (a family of) objects.
ProductA1 object copied.
ProductB1 object copied.

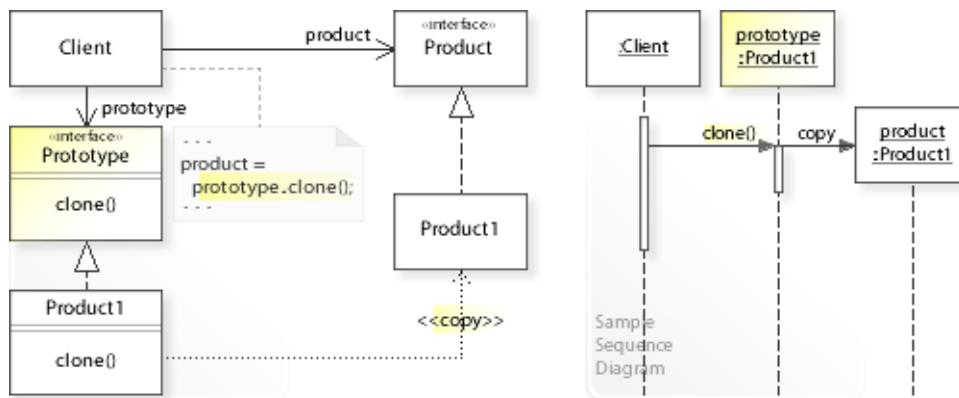
1 package com.sample.prototype.basicAF;
2 public class Client {
3     private ProductA productA;
4     private ProductB productB;
5     private PrototypeFactory factory;
6
7     public Client(PrototypeFactory factory) {
8         this.factory = factory;
9     }
10    public String operation() {
11        productA = factory.createProductA();
12        productB = factory.createProductB();
13        return "Hello World from Client!\n"
14            + "Using " + factory.getClass().getSimpleName()
15            + " to create (a family of) objects.\n"
16            + productA.getName() + " object copied.\n"
17            + productB.getName() + " object copied.";
18    }
19 }

1 package com.sample.prototype.basicAF;
2 public class PrototypeFactory {
3     private ProductA productA;
4     private ProductB productB;
5
6     public PrototypeFactory(ProductA1 pa, ProductB pb) {
7         this.productA = pa;
8         this.productB = pb;
9     }
10    public ProductA createProductA() {
11        return productA.clone(); //new ProductA1();
12    }
13    public ProductB createProductB() {
14        return productB.clone(); //new ProductB1();
15    }
16 }

```

```
*****
Product inheritance hierarchy.
*****  
  
1 package com.sample.prototype.basicAF;  
2 public interface ProductA {  
3     String getName();  
4     ProductA clone();  
5 }  
  
1 package com.sample.prototype.basicAF;  
2 public class ProductA1 implements ProductA {  
3     private String name;  
4  
5     public ProductA1(String name) {  
6         this.name = name;  
7     }  
8     // Copy constructor needed by clone().  
9     public ProductA1(ProductA1 pa) {  
10         this.name = pa.getName();  
11     }  
12     @Override  
13     public ProductA1 clone() {  
14         return new ProductA1(this);  
15     }  
16     public String getName() {  
17         return name;  
18     }  
19 }  
  
1 package com.sample.prototype.basicAF;  
2 public interface ProductB {  
3     String getName();  
4     ProductB clone();  
5 }  
  
1 package com.sample.prototype.basicAF;  
2 public class ProductB1 implements ProductB {  
3     private String name;  
4  
5     public ProductB1(String name) {  
6         this.name = name;  
7     }  
8     // Copy constructor needed by clone().  
9     public ProductB1(ProductB1 pa) {  
10         this.name = pa.getName();  
11     }  
12     @Override  
13     public ProductB1 clone() {  
14         return new ProductB1(this);  
15     }  
16     public String getName() {  
17         return name;  
18     }  
19 }
```

## Related Patterns



### Key Relationships (see also Overview)

- **Factory Method - Prototype**

- Factory Method uses subclasses to specify which class to instantiate statically at compile-time.
- Prototype uses prototypes to specify the kinds of objects to create dynamically at run-time. For example, Abstract Factory can be implemented by using prototypes instead of factory methods (see Prototype / Sample Code / Example 2).

## Intent



The intent of the Singleton design pattern is to:

**"Ensure a class only has one instance, and provide a global point of access to it."** [GoF]

See Problem and Solution sections for a more structured description of the intent.

- The Singleton design pattern solves problems like:
  - *How can be ensured that a class has only one instance?*
  - *How can the sole instance of a class be accessed globally?*
- For example, system objects that hold global data (like database access, file system, printer spooler, or registry).  
It must be ensured that such objects are instantiated only once within a system and that their sole instance can be accessed easily from all parts of the system.
- As a reminder:  
Global data should be kept to a minimum (needed primarily by system objects).  
In the object-oriented approach, "there is little or no global data." [GBooch07, p36]  
Instead, data should be stored (encapsulated) in those objects that primarily work on it and passed (as parameter) to other objects that need it.

## Problem



The Singleton design pattern solves problems like:

***How can be ensured that a class has only one instance?***

***How can the sole instance of a class be accessed globally?***

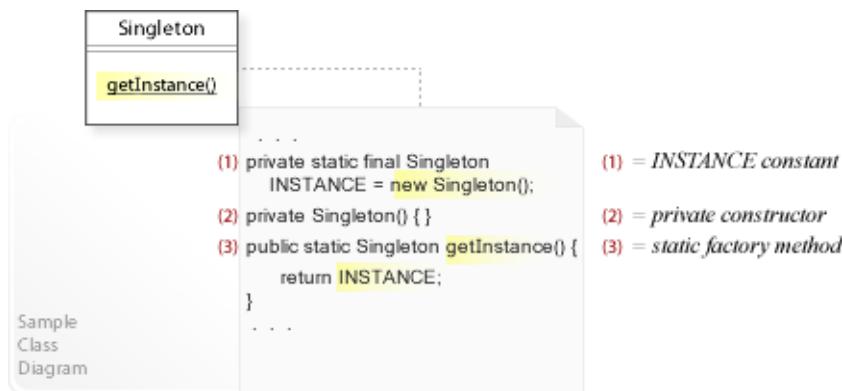
See Applicability section for all problems Singleton can solve.

- The standard way to create an object is to call the public constructor of a class (`NoSingleton()`).
- *That's the kind of approach to avoid if we want to ensure that a class can be instantiated only once (has only one instance).*
- For example, system objects that hold global data (like database access, file system, printer spooler, or registry).  
It must be ensured that such objects are instantiated only once within a system and that their sole instance can be accessed easily from all parts of the system.
- For example, avoiding creating large numbers of unnecessary objects.  
It should be possible to avoid creating unnecessary (duplicate, functionally equivalent) objects over and over again to avoid excessive memory usage and system performance problems.  
"It is often appropriate to reuse a single object instead of creating a new functionally equivalent object each time it is needed." [JBloch08, p20]

## Background Information

- Global data should be kept to a minimum (needed primarily by system objects).  
In the object-oriented approach, "there is little or no global data." [GBooch07, p36]  
Instead, data should be stored (encapsulated) in those objects that primarily work on it and passed to other objects that need it.
- A Registry is  
"A well-known object that other objects can use to find common objects and services."  
*Registry (480)* [MFowler03]

## Solution



The Singleton pattern describes a solution:

**Hide the constructor of a class.**

**Define a static factory method (`getInstance()`) that provides the sole instance.**

Describing the Singleton design in more detail is the theme of the following sections. See Applicability section for all problems Singleton can solve.

- The key idea in this pattern is to make a class itself responsible that it can be instantiated only once.
- **Hide the constructor of a class.**  
Declaring the constructor of a class *private* ensures that the class can never be instantiated from outside the class.  
"Enforce the singleton property with a private constructor or enum type" [JBloch08, Item 3]
- **Define a static factory method (`getInstance()`) that provides the sole instance of a class.**
- Clients can access the sole instance of a class easily by specifying the class name and factory method name (global point of access):  
`Singleton.getInstance()`.

### Background Information

- The `getInstance()` operation is defined *public* and *static*.  
A public static operation of a class is easy to find anywhere within an application (by using the class name and operation name):  
`Singleton.getInstance()`.
- For a discussion of static factory methods see Factory Method / Implementation.

## Motivation 1



### Consider the left design (problem):

- Multiple instances possible.
  - The class provides a public constructor (`public NoSingleton() {}`).
  - Clients can call a public constructor as often as needed.

### Consider the right design (solution):

- Only one instance possible.
  - The class hides its constructor (`(private Singleton() {})`).
  - Instead of a public constructor, a public static factory method is provided to create and return the sole instance of the class (`(Singleton.getInstance())`).
  - A public static operation is easy to access from anywhere by using the class name and operation name.

## Applicability



## Design Problems

- **Creating Single Objects**

- How can be ensured that a class has only one instance?
- How can the sole instance of a class be accessed globally?

- **Limiting the Number of Objects**

- How can the number of instances of a class be limited?
- How can creating large numbers of unnecessary objects be avoided?

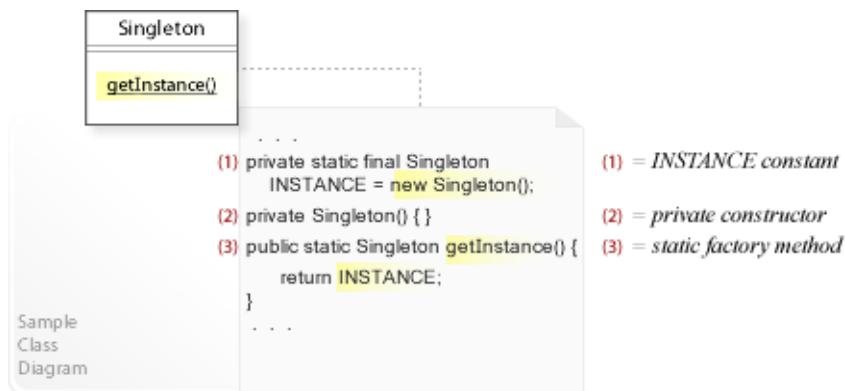
## Structure, Collaboration



### Static Class Structure

- Singleton
  - (1) Defines an `INSTANCE` constant of type `Singleton` that holds the sole instance of the class.
  - Fields declared `final` are initialized once and can never be changed.
  - (2) Hides its constructor (`private Singleton() {}`).
  - This ensures that the class can never be instantiated from outside the class.
  - (3) Defines a public static factory method (`getInstance()`) for returning the sole instance of the class.

## Consequences



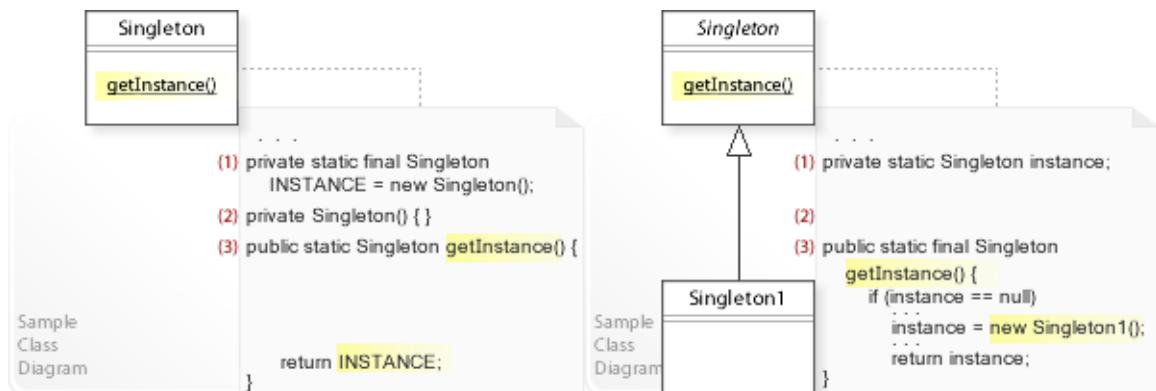
### Advantages (+)

- Can control object creation.
  - The getInstance operation can control the creation process, for example, to allow more than one instance of a class.

### Disadvantages (-)

- Makes testing its clients difficult.
  - "Making a class a singleton can make it difficult to test its clients, as it's impossible to substitute a mock implementation for a singleton unless it implements an interface that serves as its type." [JBloch08 , p17]
- May cause problems in a multi-threaded environment.
  - In a multi-threaded application, a singleton that holds mutable data (i.e., data that can be changed after an object is created) must be implemented carefully (synchronized).

## Implementation



### Implementation Issues

Variant 1:

**Ensuring that a class can be instantiated only once.**

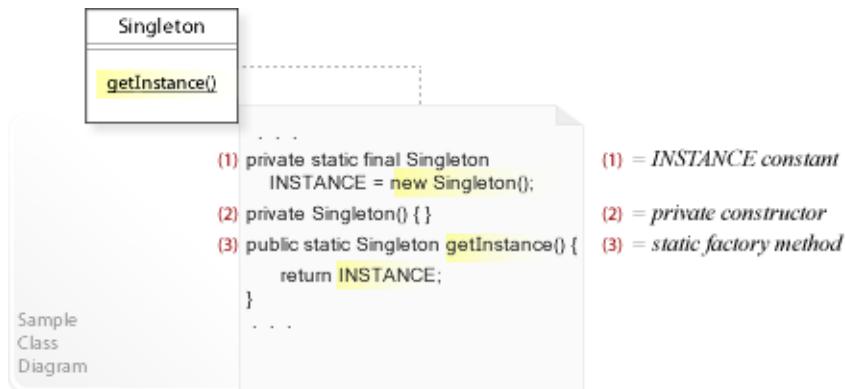
- (1) The INSTANCE constant is set to the instance of the class.  
Fields declared `final` are initialized once and can never be changed.  
In Java, "Final fields also allow programmers to implement thread-safe immutable objects without synchronization." [JLS12, 17.5 Final Field Semantics]
- (2) The constructor of the class is hidden (declared `private`).  
This ensures that the class can't be instantiated (from outside the class) or subclassed.
- (3) The public static `getInstance()` operation simply returns the INSTANCE constant.  
Clients can access the sole instance easily by calling `Singleton.getInstance()`.

Variant 2:

**Ensuring that only one of multiple subclasses can be instantiated only once.**

- (1) The `instance` field holds the instance of a subclass.
- (2) Singleton's default constructor is used.  
Because the class is abstract, it can't be instantiated.  
Note that a class having a private constructor can't be subclassed (because a subclass needs to call the parent class's constructor).
- (3) The public static final `getInstance()` operation must decide which subclass to instantiate.  
This can be done in different ways: according to user input, system environment, configuration file, dependency injection, etc.  
Note that operations declared `final` can't be redefined by subclasses.
- See also Abstract Factory / Sample Code / Example 3 / Creating different (consistent) families of objects.

## Sample Code 1



**Basic Java code for implementing the sample UML diagram.**

```
1 package com.sample.singleton.basic;
2 public class MyApp {
3     public static void main(String[] args) {
4
5         System.out.println(Singleton.getInstance().getName()
6             + " instantiated only once.");
7     }
8 }
Singleton instantiated only once.

1 package com.sample.singleton.basic;
2 public class Singleton {
3     private static final Singleton INSTANCE = new Singleton();
4
5     private Singleton() { }
6
7     public static Singleton getInstance() {
8         return INSTANCE;
9     }
10    public String getName() {
11        return "Singleton";
12    }
13 }
```

## Related Patterns

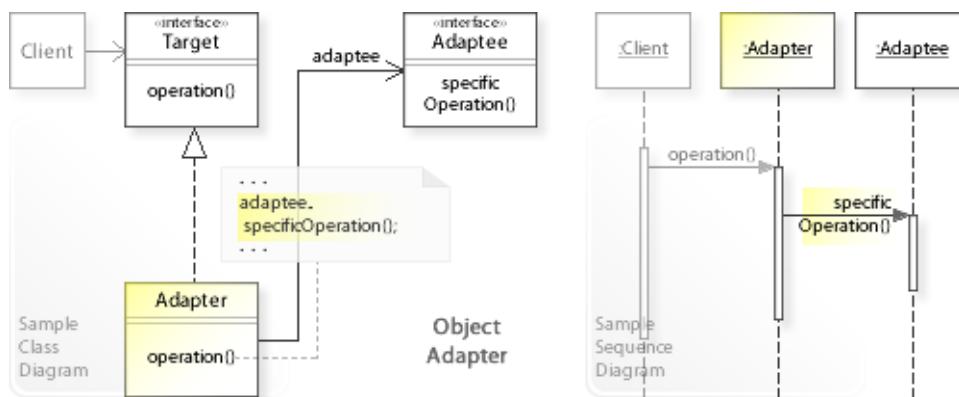


### Key Relationships (see also Overview)

- **Flyweight - Singleton**
  - The Flyweight factory is usually implemented as Singleton.

## Part III. Structural Patterns

## Intent



The intent of the Adapter design pattern is to:

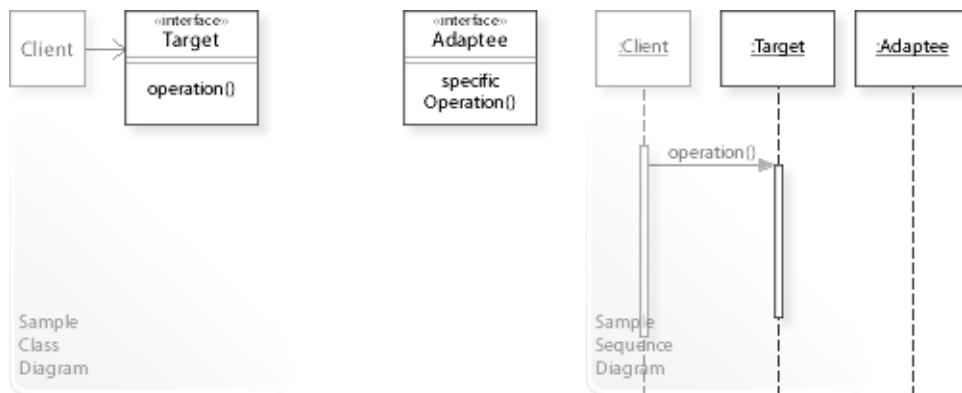
**"Convert the interface of a class into another interface clients expect."**

**Adapter lets classes work together that couldn't otherwise because of incompatible interfaces."** [GoF]

See Problem and Solution sections for a more structured description of the intent.

- The Adapter design pattern solves problems like:
  - *How can an object be accessed that has an incompatible interface?*
  - *How can objects work together that have incompatible interfaces?*
- Often we want to access an already existing reusable object (**Adaptee** | **specificOperation()**) that provides the needed functionality but has an incompatible interface.
- The Adapter pattern describes how to solve such problems:
  - *Convert the interface of a class into another interface clients expect.*  
Define a separate **Adapter** object that converts an incompatible interface (**Adaptee**) into another interface clients expect (**Target**).
  - Work through an **Adapter** object to access an object that has an incompatible interface.

## Problem



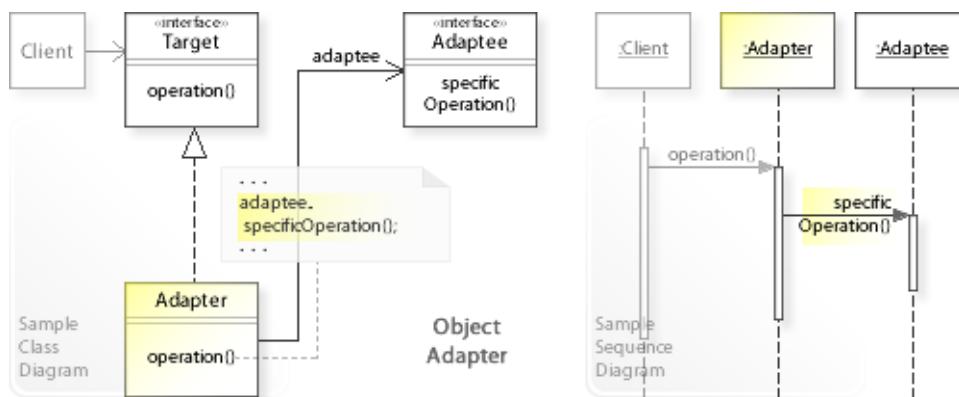
The intent of the Adapter design pattern is to solve problems like:

***How can an object be accessed that has an incompatible interface? How can objects work together that have incompatible interfaces?***

See Applicability section for all problems Adapter can solve.

- Often we want to access an already existing reusable object (Adaptee | specificOperation()) that provides the needed functionality but has an incompatible interface.
- An inflexible way to solve this problem is to change the object so that its interface conforms to the needed interface (Target | operation()).  
But it's impossible to change an (already existing) object each time an application needs another interface.
- *That's the kind of approach to avoid if we want to access an object that has an incompatible interface without having to change the object.*

## Solution



The Adapter pattern describes a solution:

**Define a separate Adapter object that converts an incompatible interface (**Adaptee**) into another interface clients expect (**Target**).**

**so that clients can work through an Adapter object to access an incompatible object.**

**Work through an Adapter object to access an object that has an incompatible interface.**

Describing the Object Adapter in more detail is the theme of the following sections.

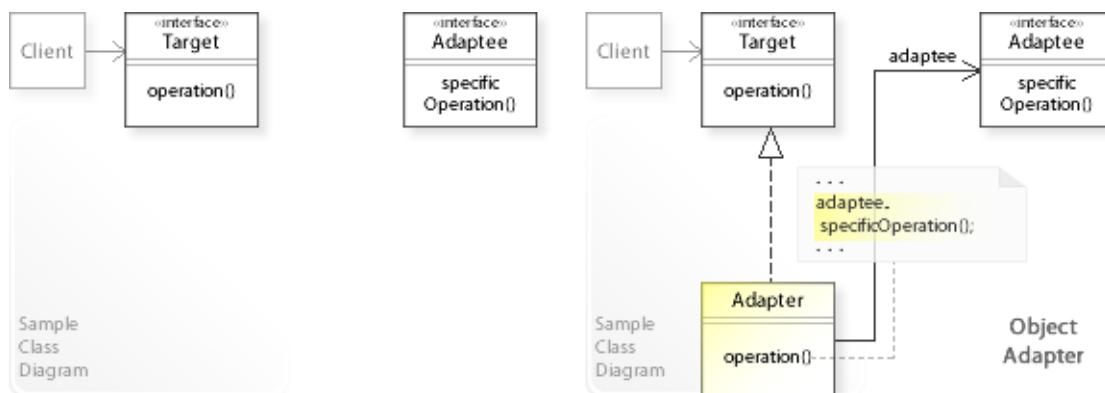
See Applicability section for all problems Adapter can solve.

- The key idea in this pattern is to adapt the interface of an (already existing) object independently from (without changing) the object itself.
- **Define a separate Adapter object:**
  - **Class Adapter:** Define a class (**Adapter**) that implements the needed interface (**Target** | `operation()`) in terms of (*by inheriting from*) the incompatible **Adaptee** implementation class.
  - **Object Adapter:** Define a class (**Adapter**) that implements the needed interface (**Target** | `operation()`) in terms of (*by delegating to*) the incompatible **Adaptee** interface (`adaptee.specificOperation()`).

The object adapter is more flexible because it can adapt an incompatible interface independently from (without having to know/subclass) any implementation classes.
- There exists a wide range of possible adaptations, from simply changing the name of an operation to supporting an entirely different set of operations.
- **Work through an Adapter object to work with an object (**Adaptee**) that has an incompatible interface (`specificOperation()`).**
- **Built-in interface adaptation:**  
To make an object more reusable, it can be designed with built-in interface adaptation by delegating the adaptation to an **Adapter** object.  
This enables run-time flexibility (via object composition).  
An object can be configured with an **Adapter** object, which it then uses to work with an unforeseeable interface, and even more, the **Adapter** object can be (ex)changed dynamically at run-time (pluggable adapters).

Unless otherwise stated, the following sections describe the **Object Adapter**.

## Motivation 1



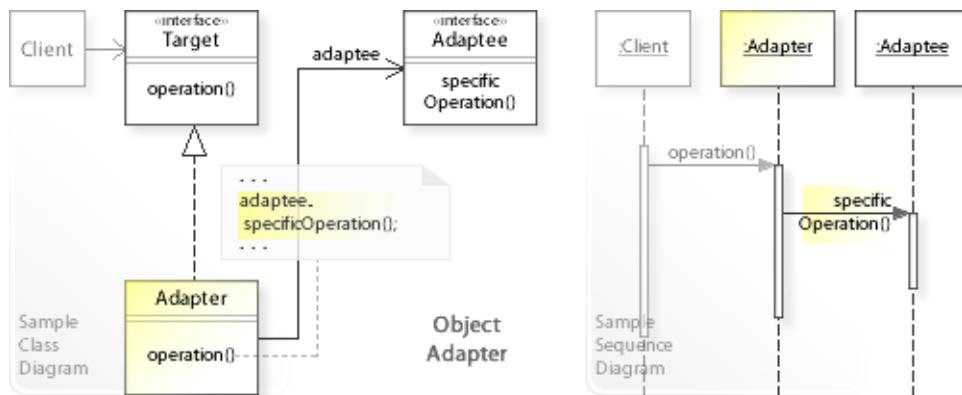
**Consider the left design (problem):**

- Clients can not work with `Adaptee`.
  - A client can not reuse an `Adaptee` object that provides the needed functionality but not the needed interface.
  - Clients of an application expect the `Target` interface (`operation()`), which can not be changed without breaking the clients.

**Consider the right design (solution):**

- Clients can work with `Adaptee`.
  - A client can reuse an `Adaptee` object that provides the needed functionality but not the needed interface.
  - Clients work through an `Adapter` that delegates requests to the `Adaptee` interface (`adaptee.specificOperation()`).

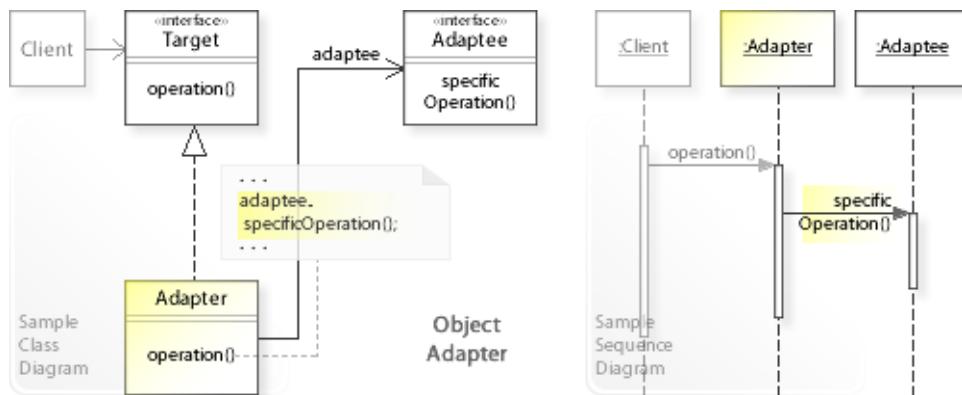
## Applicability



## Design Problems

- **Adapting Incompatible Interfaces**
  - How can an object be accessed that has an incompatible interface?
  - How can objects work together that have incompatible interfaces?
- **Built-In Interface Adaptation**
  - How can a reusable object be designed so that it can work with unforeseeable types of objects?

## Structure, Collaboration



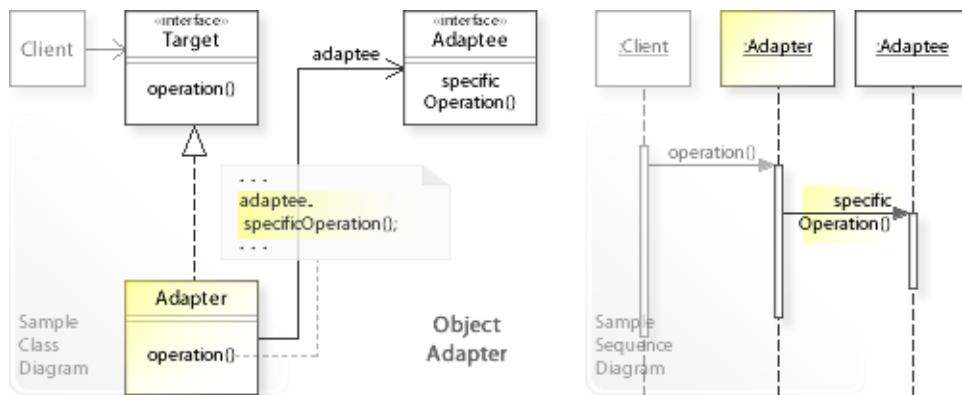
### Static Class Structure

- `Target`
  - Defines an interface clients expect.
- `Adaptee`
  - Defines an incompatible interface (that needs adaptation).
- `Adapter`
  - Implements the `Target` interface in terms of (by delegating to) the `Adaptee` interface.

### Dynamic Object Collaboration

- In this sample scenario, a `Client` object works through an `Adapter` object to access an `Adaptee` object.
- The `Client` calls `operation()` on the `Adapter` object (of type `Target`).
- `Adapter` calls (delegates the request to) `specificOperation()` on the `Adaptee` object that fulfills the request.
- `Adapter` may do work of its own before and/or after delegating a request.
- See also Sample Code / Example 1.

## Consequences

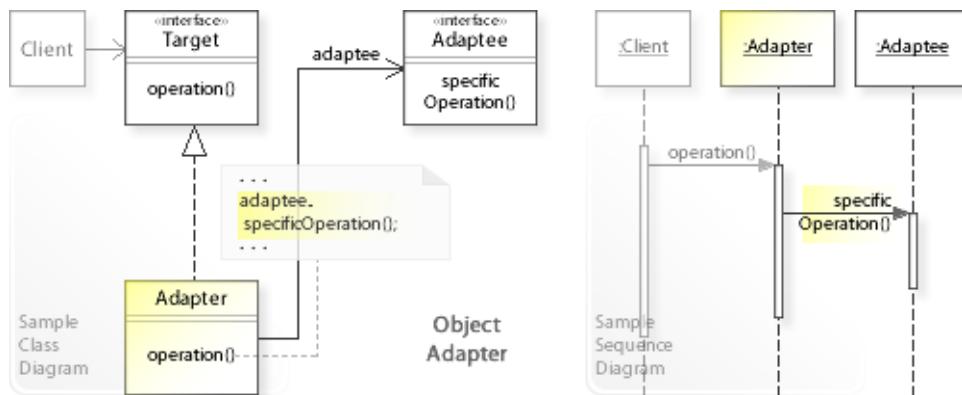


### Advantages (+)

- Supports reusing existing functionality.
  - Often an existing reusable object isn't reused only because its interface doesn't match the needed interface defined in an application.
  - By working through an adapter, such objects can be reused without having to change existing interfaces or implementations.
- Object adapter provides a flexible alternative to subclassing.
  - Inheritance offers another way to adapt an incompatible interface (class adapter).
  - The adapter implements the target interface in terms of (by inheriting from) an adaptee implementation class.
  - But this commits to a concrete adaptee class and wouldn't work for other adaptee (sub)classes.
  - Furthermore, if the adaptee implementation classes belong to an other application, they are usually hidden and can't be accessed.

### Disadvantages (-)

## Implementation

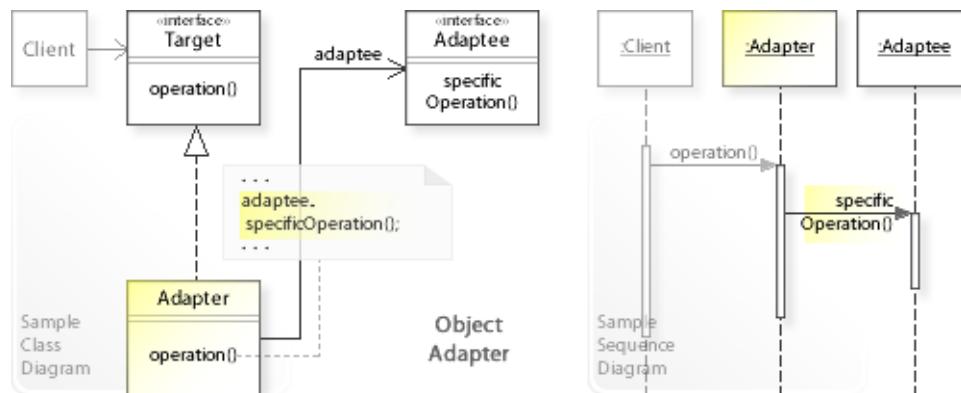


### Implementation Issues

- **Implementation Variants**

- The implementation of an adapter depends on how different **Adaptee** and **Target** interfaces are.  
"There is a spectrum of possible work, from simple interface conversion - for example, changing the names of operations - to supporting an entirely different set of operations." [GoF, p142]
- An adapter can implement additional functionality that the adapted class doesn't provide but the clients need.

## Sample Code 1



### Basic Java code for implementing the sample UML diagrams.

```

1 package com.sample.adapter.basic;
2 public class Client {
3     // Running the Client class as application.
4     public static void main(String[] args) {
5         // Creating an adapter for an incompatible adaptee.
6         Adapter adapter = new Adapter(new Adaptee1());
7         System.out.println(
8             // Working through the adapter.
9             adapter.operation());
10    }
11 }

Adapter: Delegating client request to the incompatible adaptee interface ...
Adaptee: Hello World!

1 package com.sample.adapter.basic;
2 public interface Target {
3     String operation();
4 }

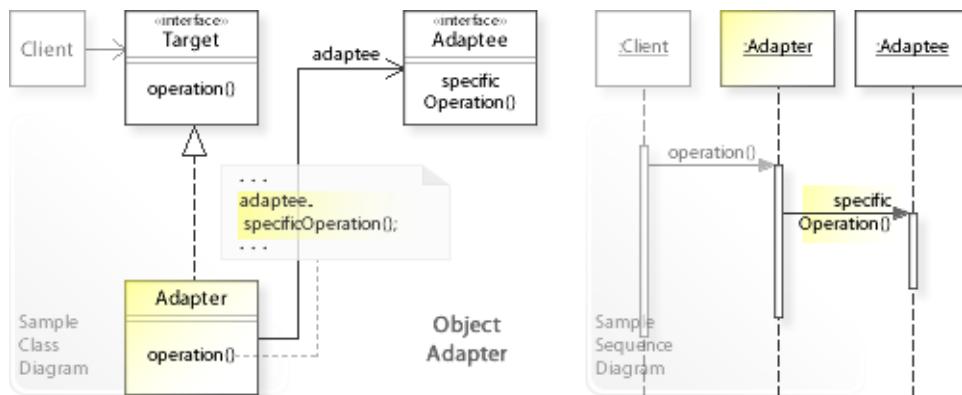
1 package com.sample.adapter.basic;
2 public class Adapter implements Target {
3     private Adaptee adaptee;
4     public Adapter(Adaptee adaptee) {
5         this.adaptee = adaptee;
6     }
7     public String operation() {
8         // Implementing the target operation in terms of
9         // (by delegating to) the adaptee specificOperation.
10        System.out.println(
11            "Adapter: Delegating client request to the incompatible adaptee interface ...");
12        return adaptee.specificOperation();
13    }
14 }

1 package com.sample.adapter.basic;
2 public interface Adaptee {
3     String specificOperation();
4 }

1 package com.sample.adapter.basic;
2 public class Adaptee1 implements Adaptee {
3     public String specificOperation() {
4         return "Adaptee: Hello World!";
5     }
6 }

```

## Related Patterns

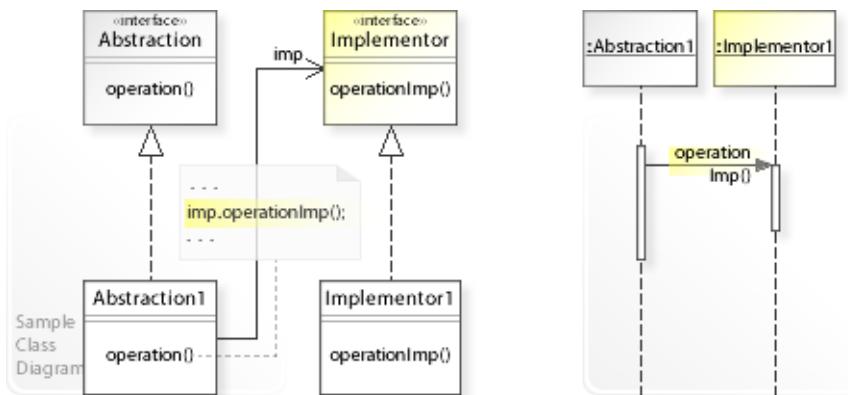


### Key Relationships (see also Overview)

- **Adapter - Facade**

- Adapter does not create or change an interface,  
it merely forwards client requests to an incompatible interface  
(Object Adapter).
- Facade creates a new (simple) interface  
for a set of (complex) interfaces.

## Intent



The intent of the Bridge design pattern is to:

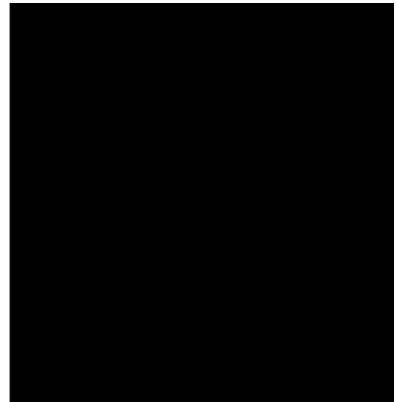
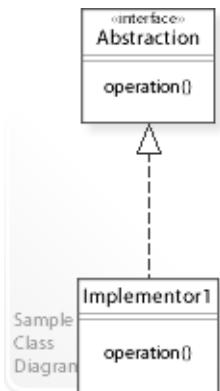
**"Decouple an abstraction from its implementation so that the two can vary independently."** [GoF]

See Problem and Solution sections for a more structured description of the intent.

- The Bridge design pattern solves problems like:
  - *How can an abstraction and its implementation vary independently?*
  - *How can an abstraction support different implementations?*
  - *How can an implementation be selected and switched dynamically?*
- For example, supporting different hardware environments.  
To make an application portable across different hardware environments, it should be possible that an abstraction can support different hardware specific implementations so that an implementation can be selected and switched dynamically at run-time.
- The Bridge pattern describes how to solve such problems:
  - *Decouple an abstraction from its implementation:*
  - Define separate inheritance hierarchies for an abstraction (`Abstraction`) and its implementation (`Implementor`).
  - The `Abstraction` interface is implemented in terms of (by delegating to) the `Implementor` interface.

The pattern calls the relationship between abstraction and implementation a *bridge* "because it bridges the abstraction and its implementation, letting them vary independently." [GoF, p152]

## Problem



The intent of the Bridge design pattern is to solve problems like:

***How can an abstraction and its implementation vary independently?***

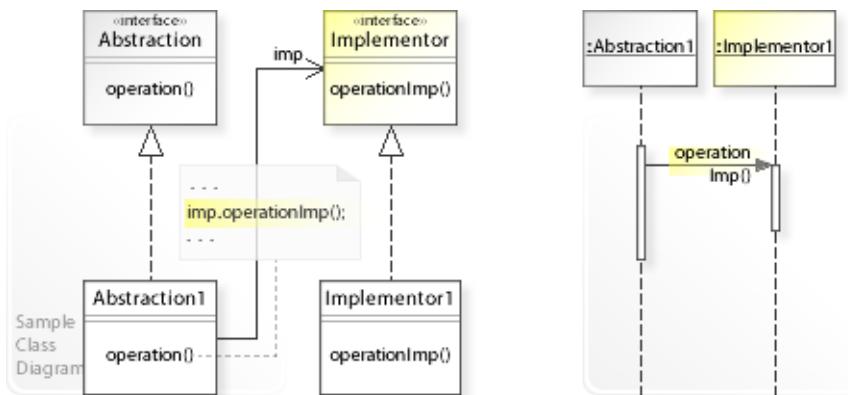
***How can an abstraction support different implementations?***

***How can an implementation be selected and switched dynamically?***

See Applicability section for all problems Bridge can solve.

- The standard way to support different implementations is by inheritance, i.e., different (sub)classes (`Implementor1`,...) implement the abstraction (interface) in different ways.  
"Inheritance binds an implementation to the abstraction permanently, which makes it difficult to modify, extend, and reuse abstractions and implementations independently. [GoF, p151]"
- *That's the kind of approach to avoid if we want that an abstraction and its implementation can vary (be implemented and changed) independently from each other.*
- For example, supporting different hardware environments.  
To make an application portable across different hardware environments, it should be possible that an abstraction can support different hardware specific implementations so that an implementation can be selected and switched dynamically at run-time.

## Solution



The Bridge pattern describes a solution:

**Define separate inheritance hierarchies for an abstraction and its implementation.**

**Abstraction delegates its implementation to an Implementor object.**

Describing the Bridge design in more detail is the theme of the following sections.

See Applicability section for all problems Bridge can solve.

- The key idea in this pattern is to decouple an abstraction from its implementation. The relationship between abstraction and implementation is called a *bridge* "because it bridges the abstraction and its implementation, letting them vary independently." [GoF, p152]
- **Define separate inheritance hierarchies for an abstraction (Abstraction) and its implementation (Implementor).**
  - The `Abstraction` interface is implemented in terms of (by delegating to) the `Implementor` interface (`imp.operationImp()`).
- This enables abstraction and implementation to be implemented and changed independently from each other.
- **Abstraction delegates its implementation to an Implementor object (`imp.operationImp()`).**
  - This enables *run-time* flexibility (via object composition). Usually, an `Abstraction` object is responsible for getting the right `Implementor` object at run-time. For example, from a factory object (`Factory1 | getImp()`); see Abstract Factory design pattern.

## Motivation 1



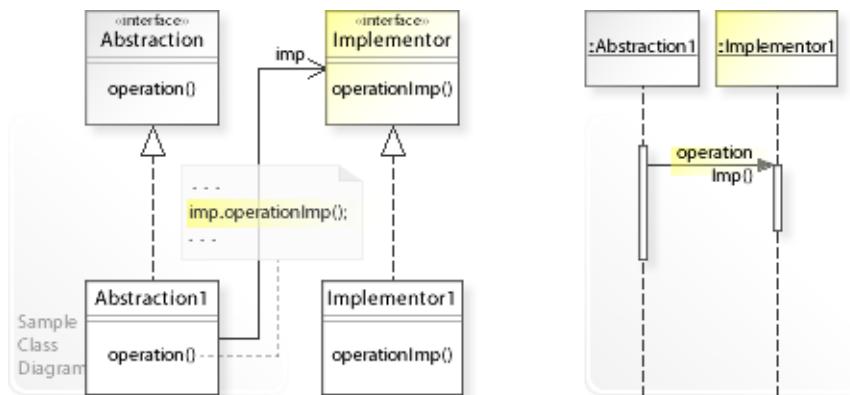
**Consider the left design (problem):**

- Implementation is coupled to the abstraction.
  - An implementation is bound to its abstraction (by defining an inheritance hierarchy).
  - This makes it hard to change (maintain) abstraction and implementation independently from each other.
  - "Inheritance binds an implementation to the abstraction permanently, which makes it difficult to modify, extend, and reuse abstractions and implementations independently. [GoF, p151]

**Consider the right design (solution):**

- Implementation and abstraction are decoupled.
  - An implementation is decoupled from its abstraction (by defining separate inheritance hierarchies).
  - This makes it easy to change (maintain) abstraction and implementation independently from each other.

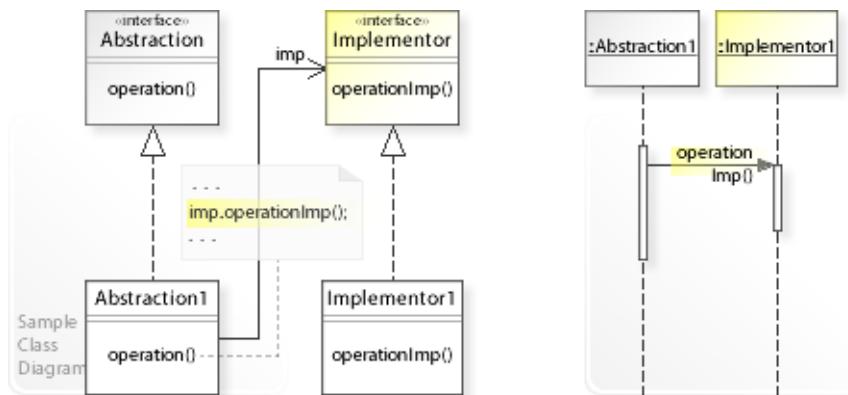
## Applicability



### Design Problems

- **Varying Abstraction and Implementation Independently**
  - How can an abstraction and its implementation vary independently?
  - How can a permanent binding between an abstraction and its implementation be avoided?
- **Supporting Different Implementations**
  - How can an abstraction support different implementations?
  - How can an implementation be selected and switched dynamically?
- **Flexible Alternative for Subclassing**
  - How can a flexible alternative be provided for supporting different implementations by subclassing?

## Structure, Collaboration



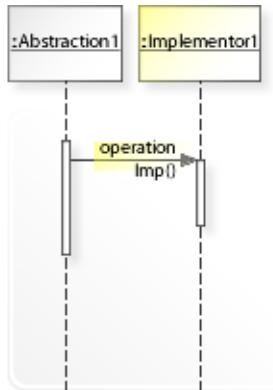
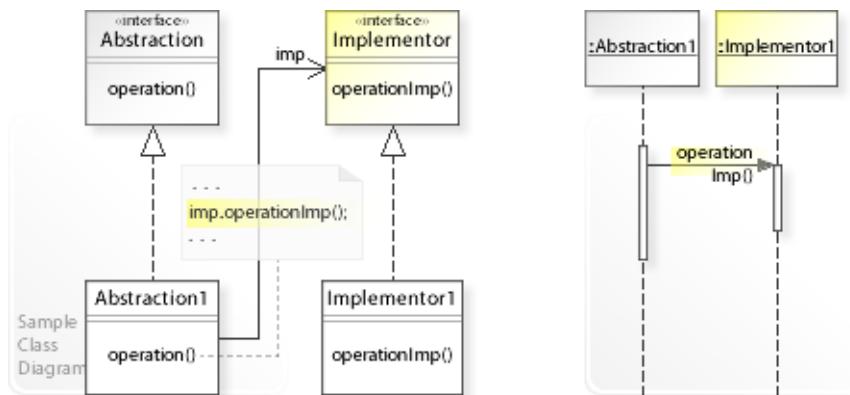
### Static Class Structure

- **Abstraction**
  - Defines an interface for an abstraction.
- **Abstraction1,...**
  - Implement the `Abstraction` interface in terms of (by delegating to) the `Implementor` interface (`imp.operationImp()`).
  - Maintain a reference (`imp`) to an `Implementor` object.
- **Implementor**
  - For all supported implementations, defines a common interface for implementing an abstraction.
  - "Typically the `Implementor` interface provides only primitive operations, and `Abstraction` defines higher-level operations based on these primitives." [GoF, p154]
- **Implementor1,...**
  - Implement the `Implementor` interface.

### Dynamic Object Collaboration

- In this sample scenario, an `Abstraction` object delegates its implementation to an `Implementor` object.  
Let's assume that `Abstraction1` uses a `Factory1` object to create an `Implementor` object.
- The interaction starts with a `Client` object that calls `operation()` on the `Abstraction1` object.
- `Abstraction1` calls `getImp()` on the `Factory1` object, which returns an `imp1` object (of type `Implementor1`).
- Thereafter, `Abstraction1` delegates its implementation by calling `operationImp()` on the `imp1` object.
- See also Sample Code / Example 1.

## Consequences



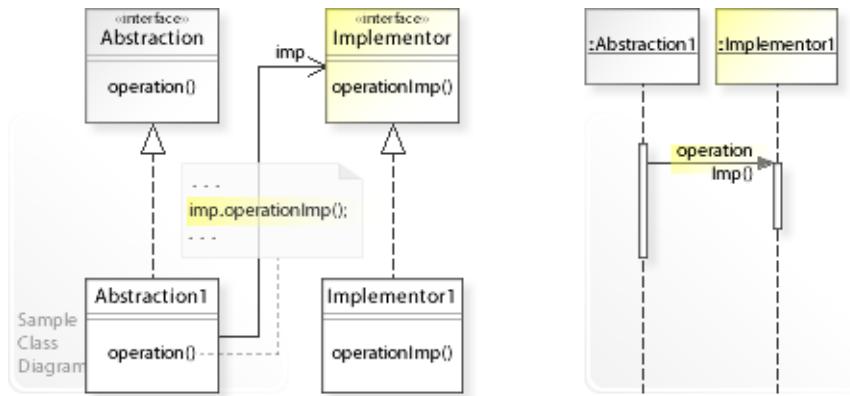
### Advantages (+)

- Provides a flexible alternative to subclassing.
  - Inheritance is the standard way to support different implementations of an abstraction.
  - But with inheritance, an implementation is bound to its abstraction and can't be changed dynamically at run-time.
  - Furthermore, inheritance would require creating a new subclass for each new combination of abstraction and implementation (explosion of subclasses).
  - Bridge makes it easy to compose abstractions and implementations dynamically at run-time (by object composition).

### Disadvantages (-)

- Introduces an additional level of indirection.
  - Bridge achieves flexibility by introducing an additional level of indirection (delegating to separate `Implementor` objects).

## Implementation

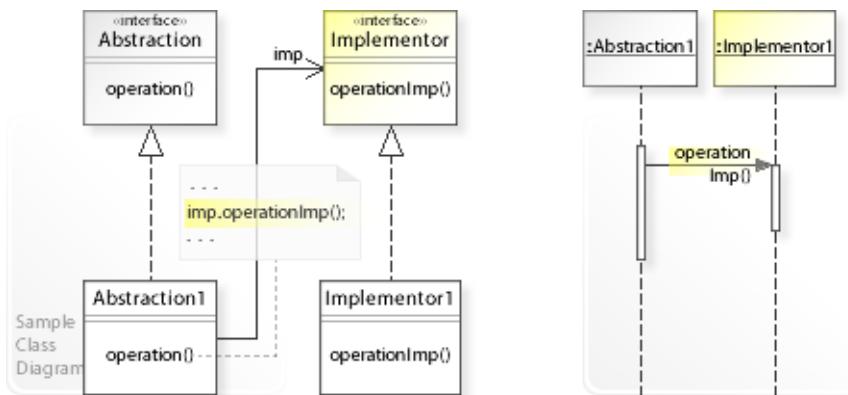


### Implementation Issues

- **Interface Design**

- The **Abstraction** and **Implementor** interfaces must be designed carefully so that the **Abstraction** interface can be implemented in terms of (by delegating to) the **Implementor** interface.
- "Typically the **Implementor** interface provides only primitive operations, and **Abstraction** defines higher-level operations based on these primitives." [GoF, p154]

## Sample Code 1



**Basic Java code for implementing the sample UML diagrams.**

```

1 package com.sample.bridge.basic;
2 public class MyApp {
3     public static void main(String[] args) {
4         // Creating an Abstraction1 object
5         // and configuring it with an Implementor1 object.
6         Abstraction abstraction = new Abstraction1(new Implementor1());
7         //
8         System.out.println(abstraction.operation());
9     }
10 }
Hello World from Abstraction1
using Implementor1!

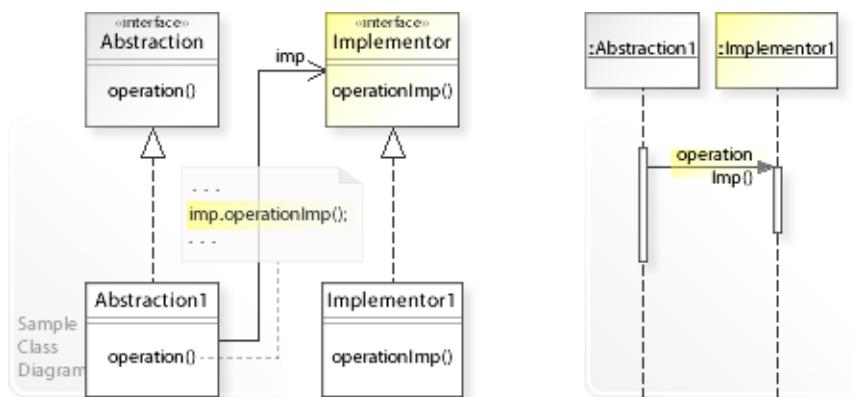
1 package com.sample.bridge.basic;
2 public interface Abstraction {
3     String operation();
4 }

1 package com.sample.bridge.basic;
2 public class Abstraction1 implements Abstraction {
3     private Implementor imp;
4     //
5     public Abstraction1(Implementor imp) {
6         this.imp = imp;
7     }
8     public String operation() {
9         return "Hello World from Abstraction1 \nusing "
10            + imp.operationImp() + "!";
11     }
12 }

1 package com.sample.bridge.basic;
2 public interface Implementor {
3     String operationImp();
4 }

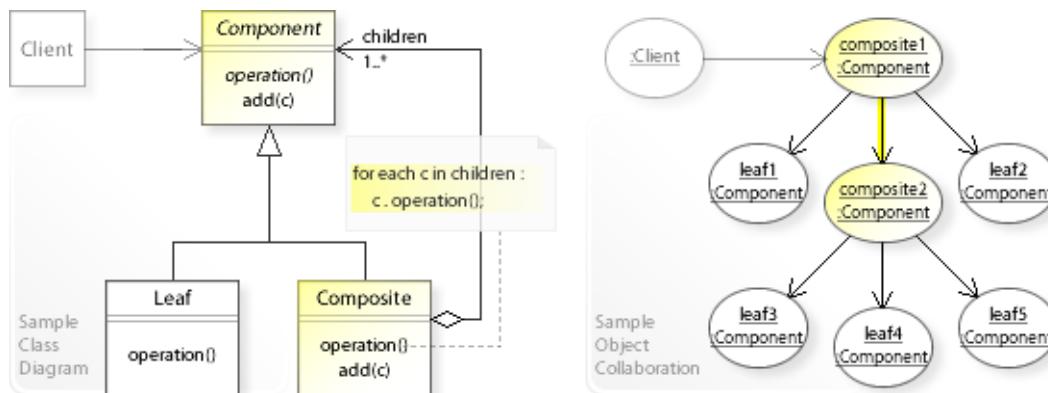
1 package com.sample.bridge.basic;
2 public class Implementor1 implements Implementor {
3     public String operationImp() {
4         return "Implementor1";
5     }
6 }
  
```

## Related Patterns



**Key Relationships** (see also Overview)

## Intent



The intent of the Composite design pattern is to:

**"Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly."** [GoF]

See Problem and Solution sections for a more structured description of the intent.

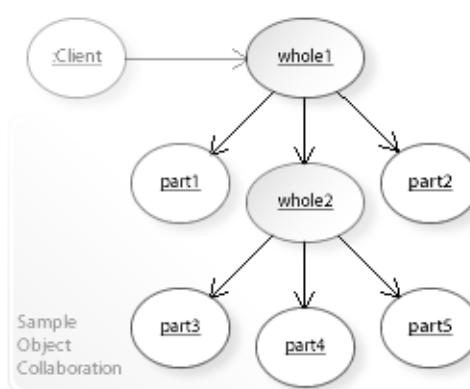
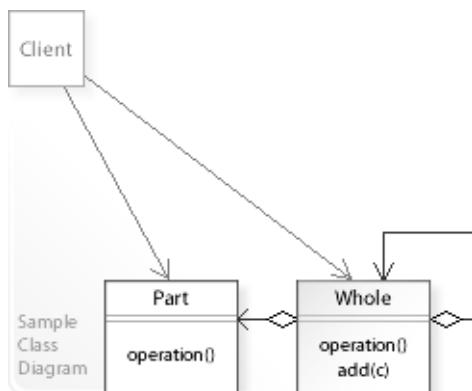
- The Composite design pattern solves problems like:
  - *How can a part-whole hierarchy be represented so that clients can treat individual objects and compositions of objects uniformly?*
- A *tree* structure consists of *Leaf* (individual/primitive) objects and *Composite* (subtree/container) objects organized into a hierarchy.  
*Leaf* objects have no children, *Composite* objects have children.  
 Children can be *Leaf* or *Composite* objects.
- Trees are widely used in object-oriented systems to represent ordered and nested data structures (like part-whole hierarchies).
- The Composite pattern describes how to solve such problems:
  - *Compose objects into tree structures to represent part-whole hierarchies.*
  - The key concept here is to compose objects recursively (*recursive composition*).  
 The resulting structure is a *tree structure*.
  - Clients work through the common *Component* interface and can treat all objects in the hierarchy uniformly. This greatly simplifies client code for working with arbitrary complex hierarchies.

## Background Information

- "A tree is a data structure composed of a set of nodes organized into a hierarchy. Each node has a parent and an ordered list of zero, one, or multiple children. The children can be simple nodes or complete subtrees.

In computer science, we draw trees with the root node at the top and the branches descending below. Root nodes are analogous to the root directory on a disk. Children are analogous to files and subdirectories." [TParr07, (2) p75]

## Problem



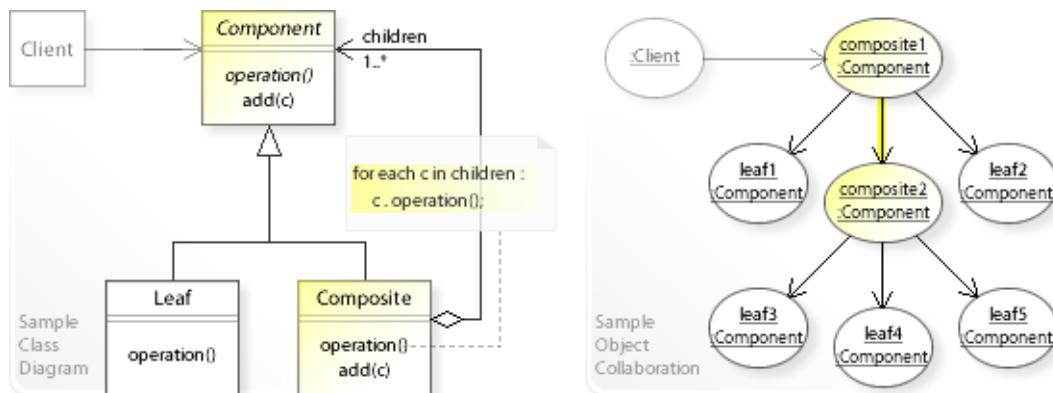
The intent of the Composite design pattern is to solve problems like:

**How can a part-whole hierarchy be represented  
so that clients can treat individual objects and compositions of objects uniformly?**

See Applicability section for all problems Composite can solve.

- One way to represent a part-whole hierarchy is to define (1) **Part** objects and (2) **Whole** objects that act as containers for **Part** and other **Whole** objects.  
Clients that create and traverse a hierarchy must treat **Part** and **Whole** objects differently, which makes client code more complex especially for multi-level object structures that are constructed and traversed dynamically.
- *That's the kind of approach to avoid if we want to simplify client code so that all objects in the hierarchy can be treated uniformly.*
- For example, representing a Bill of Materials.  
A Bill of Materials (BOM) is a part-whole structure that describes the parts and subcomponents (wholes) that make up a manufactured product (see also Builder for creating a BOM).  
For example, calculating total prices: If a client refers to a part, the total price of this part is calculated, and if it refers to a whole, the total price of this subcomponent is calculated (see Sample Code / Example 2 / BOM).
- For example, representing a text document.  
A text document can be organized as a part-whole hierarchy that consists of characters, pictures, etc. (parts) and lines, pages, etc. (wholes).  
For example, to display the entire document, a client calls display on the top most whole object.

## Solution



The Composite pattern describes a solution:

**Define separate Composite objects that compose Leaf and Composite objects recursively.**

**Work through the Component interface to treat Leaf and Composite objects uniformly.**

Describing the Composite design in more detail is the theme of the following sections.

See Applicability section for all problems Composite can solve.

- "The key to the Composite pattern is an abstract class [Component] that represents *both* primitives and their containers." [GOF, p163]

There are two ways to design the Component interface:

*Design for uniformity* versus *design for type safety* (see Implementation).

- The key concept here is to compose Leaf and Composite objects recursively (*recursive composition*).

For example, compose bottom-level leaf objects (leaf3, leaf4, leaf5) into a composite object (composite2), compose this composite object and same-level leaf objects (leaf1, leaf2) into a higher-level composite object (composite1), and so on recursively.

The resulting structure is a **tree structure** that represents a part-whole hierarchy.

- **Define separate Composite objects:**

- Define a class (Composite) that defines a container for child components (i.e., Leaf and Composite objects) and forwards operations to these children (for each c in children: c.operation()).
- Additionally, Composite implements (redefines) the child management operations (add(c)).

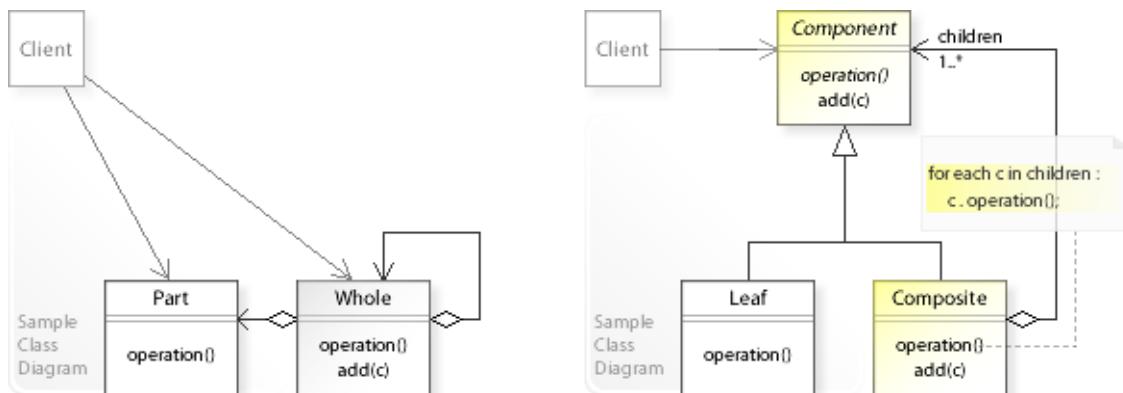
- **Work through the common Component interface to treat Leaf and Composite objects uniformly.**

If the receiver of a request is a leaf, the request is performed directly.

If the receiver is a composite, the request is forwarded to its child components recursively.

This greatly simplifies client code for interacting with arbitrary complex hierarchies.

## Motivation 1



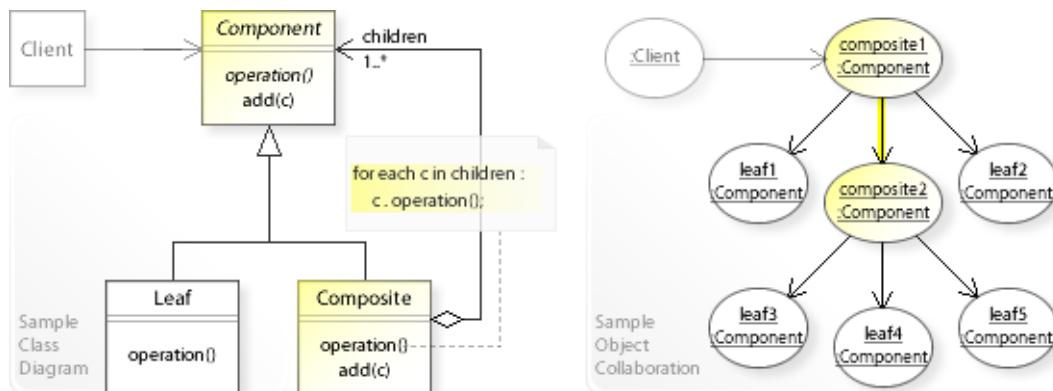
**Consider the left design (problem):**

- No common interface.
  - No common interface is defined for individual objects (**Part**) and their containers (**Whole**).
  - This forces clients to treat **Part** and **Whole** objects differently, which greatly complicates client code for constructing and traversing complex hierarchies.

**Consider the right design (solution):**

- Common interface.
  - A common interface (**Component**) is defined for individual objects (**Leaf**) and their containers (**Composite**).
  - This lets clients treat **Leaf** and **Composite** objects uniformly, which greatly simplifies client code for constructing and traversing complex hierarchies.

## Applicability

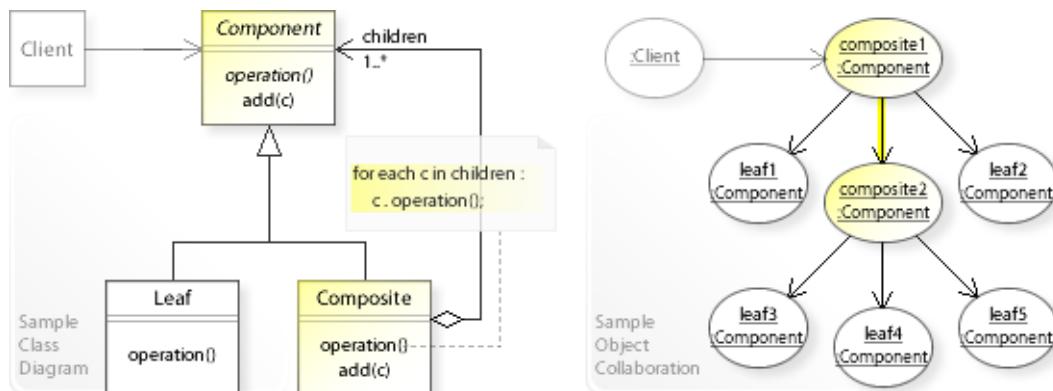


## Design Problems

- **Representing Part-Whole Hierarchies**

- How can a part-whole hierarchy be represented so that clients can treat individual objects and compositions of objects uniformly?
- How can clients treat a set of objects in a hierarchy as one object?
- How can clients treat an entire hierarchy as a single object?

## Structure, Collaboration



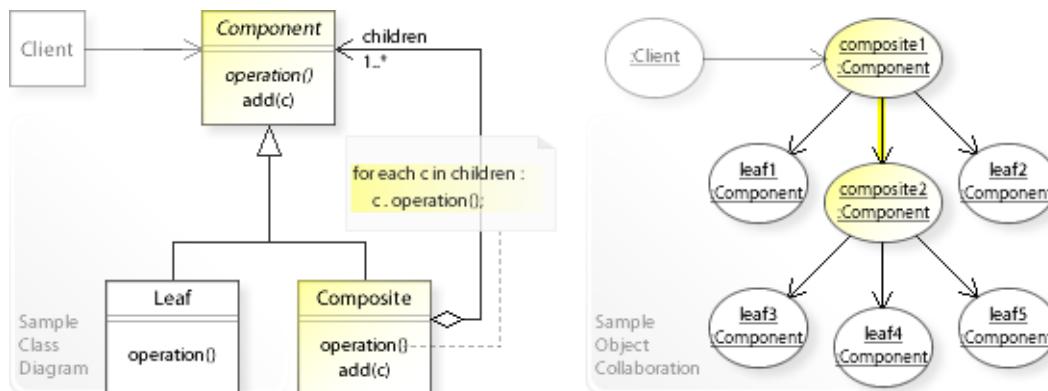
### Static Class Structure

- **Component**
  - Defines a common interface for **Leaf** and **Composite** objects.
  - `operation()` stands for non-child management operations.
  - `add(c)` stands for child management operations like adding child components to a **Composite** container.
  - Usually implements default ("do nothing") behavior for child management operations.
- **Leaf**
  - Has no children.
  - Implements the non-child management operations.
- **Composite**
  - Maintains a container of child components (`children`).
  - Implements the non-child management operations by forwarding requests to its children (`for each c in children: c.operation()`).
  - Implements (redefines) the child management operations.

### Dynamic Object Collaboration

- In this sample scenario, a **Client** object sends a request to the top-level **Composite** object in the hierarchy.
- The request is forwarded to the child components (i.e., **Leaf** and **Composite** objects) recursively, that is, the request is performed on all objects downwards the hierarchy.
- A **Composite** object may do work of its own before and/or after forwarding a request, for example, to compute total prices (see Sample Code / Example 2).
- See also Sample Code / Example 1.

## Consequences



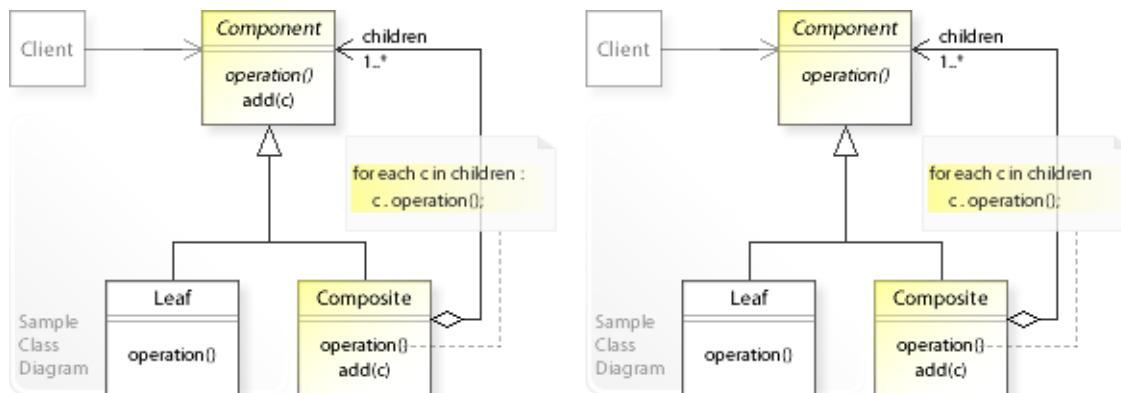
### Advantages (+)

- Simplifies clients.
  - Clients can treat all objects in the hierarchy uniformly, which greatly simplifies client code.
- Makes adding new components easy.
  - Clients refer to the common `Component` interface and are independent of its implementation.
  - That means, clients do not have to be changed when new `Component` classes are added or existing ones are changed.
- Allows building and changing complex hierarchies dynamically at run-time.
  - The pattern shows how to apply recursive composition to build any potentially complex hierarchical object structure dynamically at run-time.

### Disadvantages (-)

- Uniformity versus type safety.
  - There are two main design variants (where to define the child management operations): design for *uniformity* and design for *type safety*.
  - The Composite pattern emphasizes uniformity over type safety.
  - See Implementation.

## Implementation



### Implementation Issues

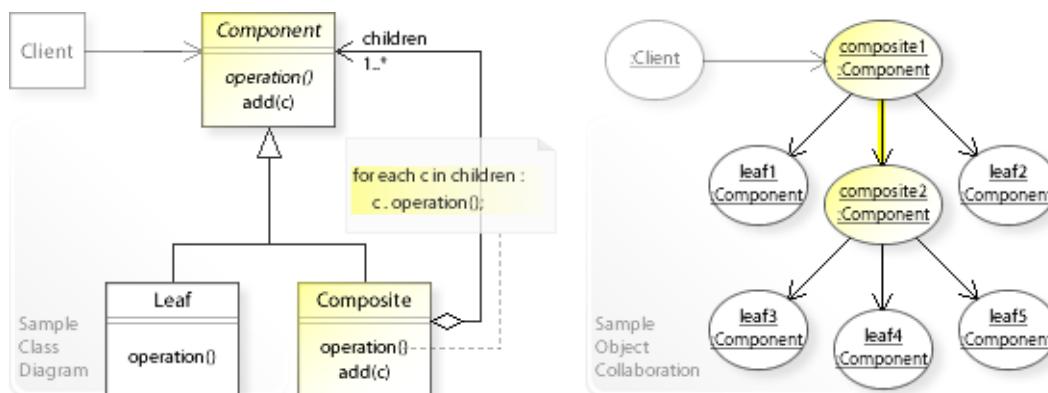
#### Variant 1: Design for Uniformity

- The only way to provide *uniformity* is to define the child management operations (`add(c)`) in the `Component` interface.
- This enables uniformity because clients can treat `Leaf` and `Composite` objects uniformly.
- But we loose type safety because `Leaf` and `Composite` interfaces (types) are not cleanly separated.
- The abstract `Component` class implements default behavior for child management operations like "do nothing" or "throw an exception". `Leaf` classes can use the default implementation, and `Composite` classes have to redefine them.
- Uniformity is useful for dynamic structures because clients often need/perform child management operations (in a document editor, for example, where the object structure is dynamically created and changed each time the document is formatted or changed).

#### Variant 2: Design for Type Safety

- The only way to provide *type safety* is to define the child management operations (`add(c)`) in the `Composite` interface.
- This enables type safety because we can rely on the type system to enforce type constraints (for example, that clients can not perform child management operations on `Leaf` components).
- But we loose uniformity because clients must treat `Leaf` and `Composite` objects differently.
- Type safety is useful for static structures (that do not change very often) because most clients do not need/perform child management operations.

## Sample Code 1



### Basic Java code for implementing the sample UML diagrams.

```

1 package com.sample.composite.basic;
2 public class Client {
3     // Running the Client class as application.
4     public static void main(String[] args) {
5         // Building the tree structure.
6         Component composite2 = new Composite("composite2");
7         composite2.add(new Leaf("leaf3"));
8         composite2.add(new Leaf("leaf4"));
9         composite2.add(new Leaf("leaf5"));
10        Component composite1 = new Composite("composite1");
11        composite1.add(new Leaf("leaf1"));
12        composite1.add(composite2);
13        composite1.add(new Leaf("leaf2"));
14        //
15        // Clients can treat the hierarchy as a single object.
16        // If the receiver is a leaf,
17        //   the request is performed directly.
18        // If the receiver is a composite,
19        //   the request is forwarded to its child components recursively.
20        //
21        composite1.operation();
22    }
23 }
```

```

composite1:
    forwarding to ... leaf1
    forwarding to ... composite2
composite2:
    forwarding to ... leaf3
    forwarding to ... leaf4
    forwarding to ... leaf5
composite2 finished.
    forwarding to ... leaf2
composite1 finished.
```

```

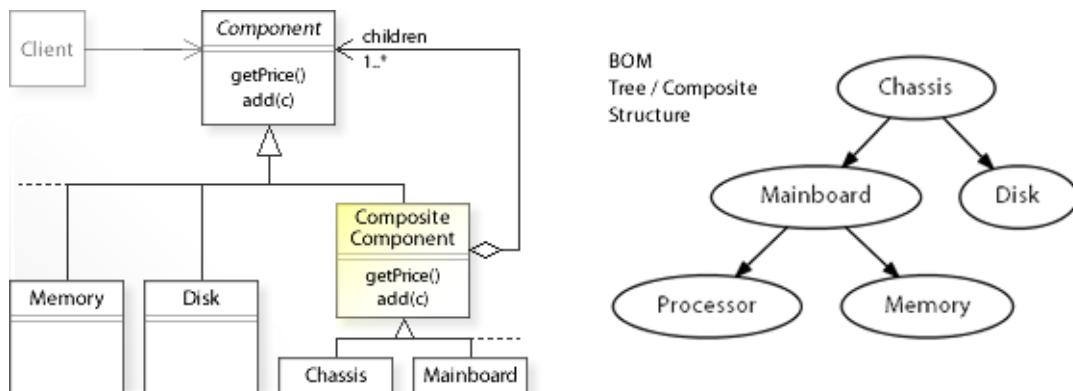
1 package com.sample.composite.basic;
2 import java.util.Collections;
3 import java.util.Iterator;
4 public abstract class Component {
5     private String name;
6     public Component(String name) {
7         this.name = name;
8     }
9     public abstract void operation();
10    public String getName() {
11        return name;
12    }
13    // Defining default implementation for child management operations.
14    public boolean add(Component c) { // fail by default
15        return false;
16    }
17    public Iterator<Component> iterator() { // null iterator
18        return Collections.<Component>emptyIterator();
```

```
19      }
20  }

1 package com.sample.composite.basic;
2 public class Leaf extends Component {
3     public Leaf(String name) {
4         super(name);
5     }
6     public void operation() {
7         // ...
8     }
9 }

1 package com.sample.composite.basic;
2 import java.util.*;
3 public class Composite extends Component {
4     private List<Component> children = new ArrayList<Component>();
5     //
6     public Composite(String name) {
7         super(name);
8     }
9     public void operation() {
10        System.out.println(getName() + ": ");
11        // Forwarding to children.
12        Iterator<Component> i = children.iterator();
13        Component c;
14        while (i.hasNext()) {
15            c = i.next();
16            System.out.println("  forwarding to ... " + c.getName());
17            c.operation();
18        }
19        System.out.println(getName() + " finished.");
20    }
21    // Overriding the default implementation.
22    @Override
23    public boolean add(Component child) {
24        return children.add(child);
25    }
26    @Override
27    public Iterator<Component> iterator() {
28        return children.iterator();
29    }
30 }
```

## Sample Code 2



### BOM Bill of Materials / Representing the BOM as tree/composite structure.

Calculating total prices (`getPrice()`) for composite components (Chassis and Mainboard).  
See also Visitor design pattern, Sample Code / Example 2 (pricing visitor).

```

1 package com.sample.composite.bom;
2 public class Client {
3     // Running the Client class as application.
4     public static void main(String[] args) throws Exception {
5         // Building the tree/composite structure.
6         Component mainboard = new Mainboard("Mainboard", 100);
7         mainboard.add(new Processor("Processor", 100));
8         mainboard.add(new Memory("Memory ", 100));
9         Component chassis = new Chassis("Chassis ", 100);
10        chassis.add(mainboard);
11        chassis.add(new Disk("Disk      ", 100));
12        //
13        // Clients can treat the hierarchy as a single object.
14        // If the receiver is a leaf,
15        //   the request is performed directly.
16        // If the receiver is a composite,
17        //   the request is forwarded to its child components recursively.
18        //
19        System.out.println(chassis.getName() + " total price: " +
20                           chassis.getPrice());
21        //
22        System.out.println(mainboard.getName() + " total price: " +
23                           mainboard.getPrice());
24    }
25 }

Chassis  total price: 500
Mainboard total price: 300
  
```

```

1 package com.sample.composite.bom;
2 import java.util.Collections;
3 import java.util.Iterator;
4 public abstract class Component {
5     private String name;
6     private long price;
7     public Component(String name, long price) {
8         this.name = name;
9         this.price = price;
10    }
11    public String getName() {
12        return name;
13    }
14    public long getPrice() { // in cents
15        return price;
16    }
17    // Defining default implementation for child management operations.
18    public boolean add(Component c) { // fail by default
19        return false;
20    }
21    public Iterator<Component> iterator() {
  
```

```
22         return Collections.emptyIterator(); // null iterator
23     }
24     public int getChildCount() {
25         return 0;
26     }
27 }

1 package com.sample.composite.bom;
2 import java.util.ArrayList;
3 import java.util.Iterator;
4 import java.util.List;
5 public class CompositeComponent extends Component {
6     private List<Component> children = new ArrayList<Component>();
7     public CompositeComponent(String name, long price) {
8         super(name, price);
9     }
10    // Overriding the default implementation.
11    @Override
12    public long getPrice() {
13        long sum = super.getPrice();
14        for (Component child : children) {
15            sum += child.getPrice();
16        }
17        return sum;
18    }
19    @Override
20    public boolean add(Component c) {
21        return children.add(c);
22    }
23    @Override
24    public Iterator<Component> iterator() {
25        return children.iterator();
26    }
27    @Override
28    public int getChildCount() {
29        return children.size();
30    }
31 }

1 package com.sample.composite.bom;
2 public class Chassis extends CompositeComponent { // Composite
3     public Chassis(String name, long price) {
4         super(name, price);
5     }
6 }

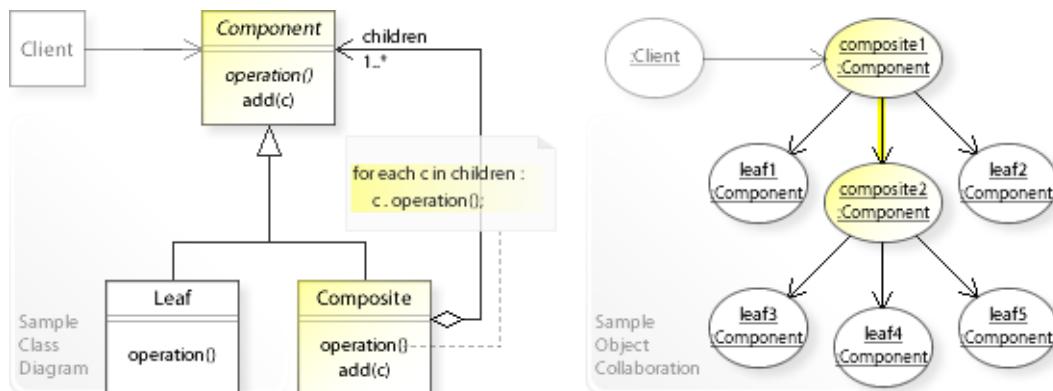
1 package com.sample.composite.bom;
2 public class Mainboard extends CompositeComponent { // Composite
3     public Mainboard(String name, long price) {
4         super(name, price);
5     }
6 }

1 package com.sample.composite.bom;
2 public class Processor extends Component { // Leaf
3     public Processor(String name, long price) {
4         super(name, price);
5     }
6 }

1 package com.sample.composite.bom;
2 public class Memory extends Component { // Leaf
3     public Memory(String name, long price) {
4         super(name, price);
5     }
6 }

1 package com.sample.composite.bom;
2 public class Disk extends Component { // Leaf
3     public Disk(String name, long price) {
4         super(name, price);
5     }
6 }
```

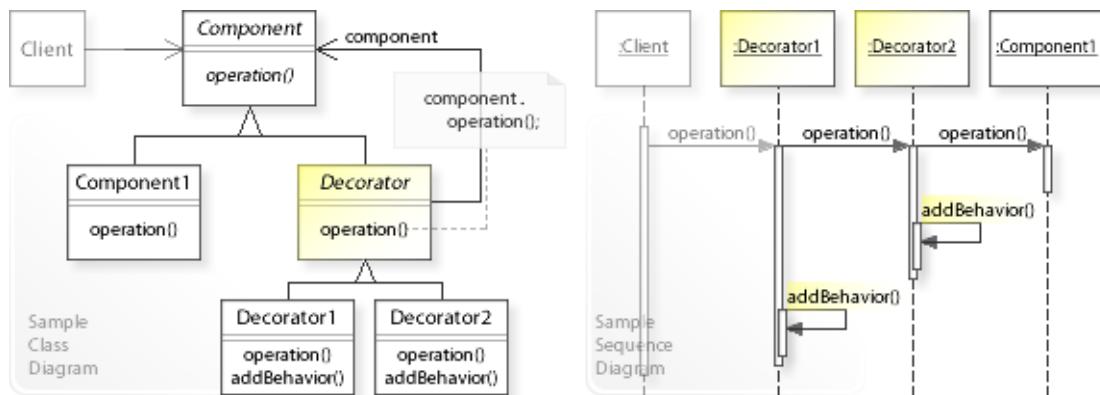
## Related Patterns



### Key Relationships (see also Overview)

- **Composite - Builder - Iterator - Visitor - Interpreter**
  - Composite provides a way to represent a part-whole hierarchy as a tree (composite) object structure.
  - Builder provides a way to create the elements of an object structure.
  - Iterator provides a way to traverse the elements of an object structure.
  - Visitor provides a way to define new operations for the elements of an object structure.
  - Interpreter represents a sentence in a simple language as a tree (composite) object structure (abstract syntax tree).
- **Composite - Flyweight**
  - Composite and Flyweight often work together.  
Leaf objects can be implemented as shared flyweight objects.
- **Composite - Chain of Responsibility**
  - Composite and Chain of Responsibility often work together.  
Existing composite object structures can be used to define the successor chain.

## Intent



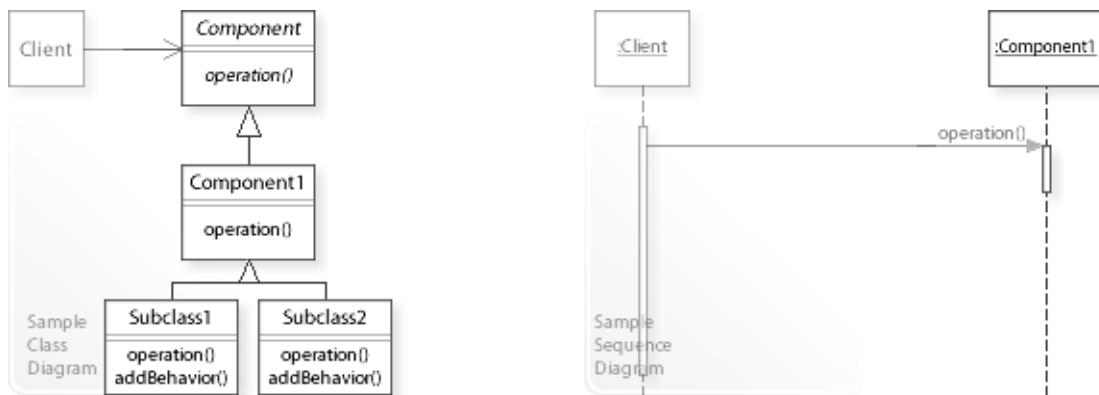
The intent of the Decorator design pattern is to:

**"Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality."** [GoF]

See Problem and Solution sections for a more structured description of the intent.

- The Decorator design pattern solves problems like:
  - *How can responsibilities be added to an object dynamically?*
  - *How can the functionality of an object be extended dynamically?*
- "A responsibility denotes the obligation of an object to provide a certain behavior." [GBooch07, p600]  
The terms *responsibility*, *behavior*, and *functionality* are usually interchangeable.
- "Sometimes we want to add responsibilities to individual objects, not to an entire class. A graphical user interface toolkit, for example, should let you add properties like borders or behaviors like scrolling to any user interface component." [GoF, p175]
- For example, reusable GUI/Web objects (like buttons, menus, or tree widgets).  
It should be possible to add embellishments (i.e., borders, scroll bars, etc.) to basic GUI/Web objects dynamically at run-time.  
"In the Decorator pattern, embellishment refers to anything that adds responsibilities to an object." [GoF, p47]

## Problem



The Decorator design pattern solves problems like:

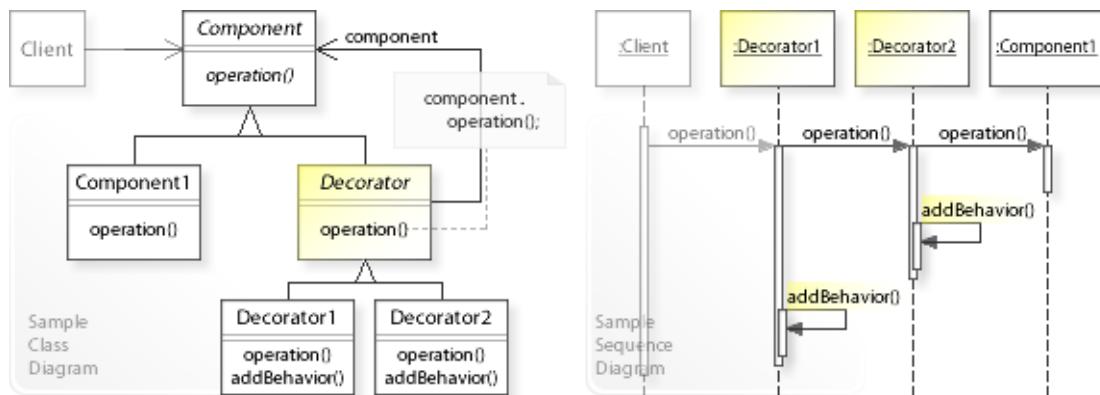
***How can responsibilities be added to an object dynamically?***

***How can the functionality of an object be extended dynamically?***

See Applicability section for all problems Decorator can solve.

- Subclassing is the standard way to extend the functionality of (add responsibilities to) a class statically at compile-time.  
Once a subclass is instantiated, the functionality is bound to the object for the life-time of the object and can not be changed at run-time.
- *That's the kind of approach to avoid if we want to extend the functionality of (add responsibilities to) an object dynamically at run-time.*
- For example, reusable GUI/Web objects (like buttons, menus, or tree widgets).  
It should be possible to add embellishments (i.e., borders, scroll bars, etc.) to basic GUI/Web objects dynamically at run-time.
- For example, I/O data stream objects [Java Platform].  
It should be possible to add responsibilities like handling data types and buffered data to basic I/O objects that only handle raw binary data (see Sample Code / Example 2).
- For example, collections [Java Collections Framework].  
It should be possible to add automatic synchronization (thread-safety) to a collection or taking away the ability to modify a collection.

## Solution



The Decorator pattern describes a solution:

**Define separate `Decorator` objects that act as transparent enclosures of an object and add responsibilities to it.**

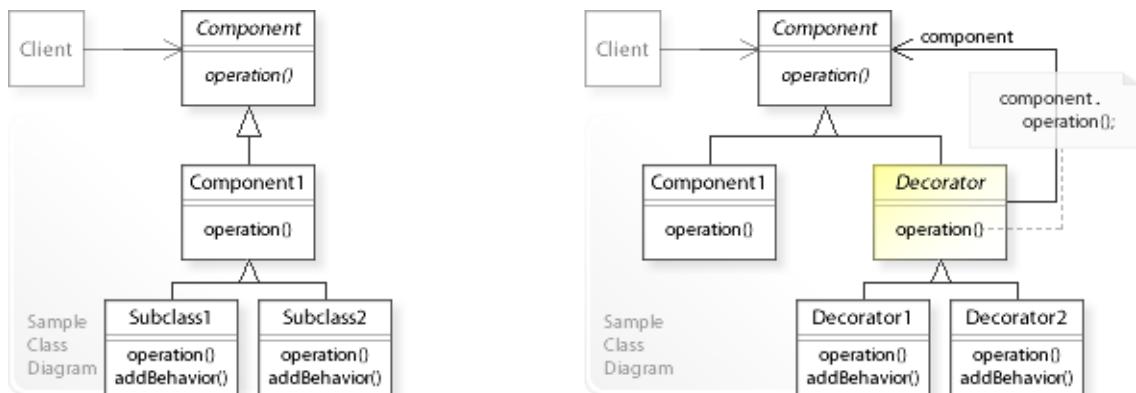
**Work through `Decorator` objects to add responsibilities to an object dynamically.**

Describing the Decorator design in more detail is the theme of the following sections.

See Applicability section for all problems Decorator can solve.

- The key idea in this pattern is to add responsibilities to an (already existing) object independently from (without changing) the object itself.
  - The key concept here is to act as a *transparent enclosure* of an object (`Component`). "Clients generally can't tell whether they're dealing with the component or its enclosure [...]" [GoF, p44] That's why `Decorator` is also known as `Wrapper`.
  - **Define separate `Decorator` objects:**
    - Define a class (`Decorator`) that defines a *transparent enclosure* of the decorated object (`Component`) by forwarding all requests to the next `Component` object (`component.operation()`).
    - Define subclasses (`Decorator1,...`) that implement additional functionality (`addBehavior()`) to be performed before and/or after forwarding a request.
  - **Work through `Decorator` objects to add (an open-ended number of) responsibilities to an object dynamically and transparently.**
- Because `Decorators` are transparent enclosures of the decorated object (`Component`), they can be nested recursively to add an open-ended number of responsibilities.
- Changing the order of the `decorators` allows to add any combinations of responsibilities. In the above sequence diagram, for example, a client works through two nested `decorators` that add two responsibilities after forwarding to a `Component` object.

## Motivation 1



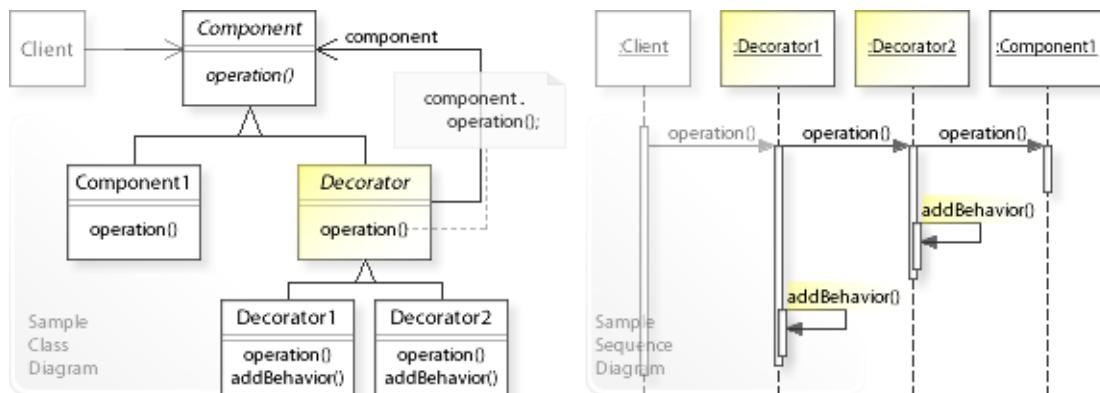
**Consider the left design (problem):**

- Static class inheritance.
  - Subclasses add responsibilities to a `Component1` class.
  - Once a subclass is instantiated, the responsibility is bound to the object for the life-time of the object.
- Explosion of subclasses.
  - Extending functionality by subclassing requires creating a new subclass for each new functionality *and* for each new combination of functionalities.
  - Supporting a large number of functionalities and their combinations would produce an explosion of subclasses.

**Consider the right design (solution):**

- Dynamic object composition.
  - Separate `Decorator` subclasses add responsibilities to a `Component` class.
  - Clients work through different `Decorator` objects to add and remove responsibilities at run-time.
- Recursively nested decorators
  - Extending functionality by decorators requires creating a new decorator for each new functionality *but not* for each new combination of functionalities.
  - Decorators can be nested recursively for supporting an open-ended number of functionalities and their combinations.

## Applicability



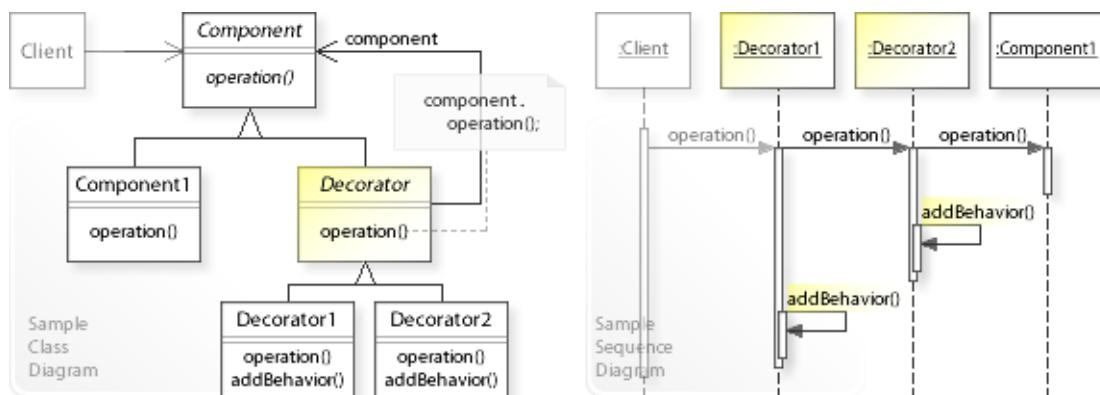
## Design Problems

- **Extending Functionality Dynamically at Run-Time**
  - How can responsibilities be added to (and withdrawn from) an object dynamically?
  - How can the functionality of an object be extended dynamically?
- **Flexible Alternative for Subclassing**
  - How can a flexible alternative be provided for extending the functionality of a class by subclassing?
  - How can defining a subclass for every new extension and combination of extensions (explosion of subclasses) be avoided?

## Refactoring Problems

- **Complicated Code**
  - How can a class that includes many different extensions of its functionality be simplified?  
*Move Embellishment to Decorator (144) [JKerievsky05]*

## Structure, Collaboration



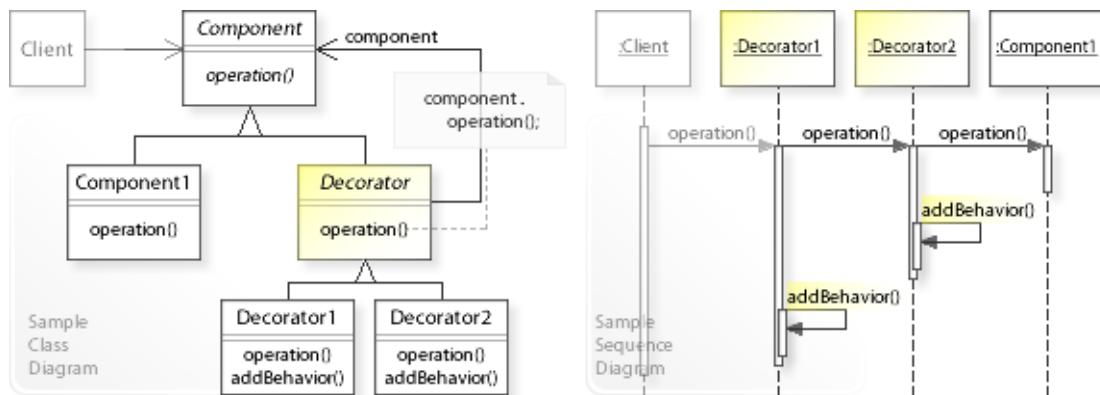
### Static Class Structure

- **Component**
  - Defines an interface for objects that get decorated.
- **Component1,...**
  - Implement the **Component** interface.
- **Decorator**
  - Defines a *transparent enclosure* of the decorated object (**Component**) by forwarding all requests to the next **Component** object (**component.operation()**).
- **Decorator1,Decorator2,...**
  - Implement additional functionality (**addBehavior()**) to be performed before and/or after forwarding a request.

### Dynamic Object Collaboration

- In this sample scenario, a **Client** object works through two decorators that add two responsibilities to a **Component1** object.
- The **Client** calls **operation()** on the **Decorator1** object.
- **Decorator1** forwards the request to the **Decorator2** object.
- **Decorator2** forwards the request to the **Component1** object.
- **Component1** performs the request and returns to **Decorator2**.
- **Decorator2** performs additional functionality (by calling **addBehavior()** on itself) and returns to **Decorator1**.
- **Decorator1** performs additional functionality and returns to the **Client**.
- See also Sample Code / Example 1.

## Consequences



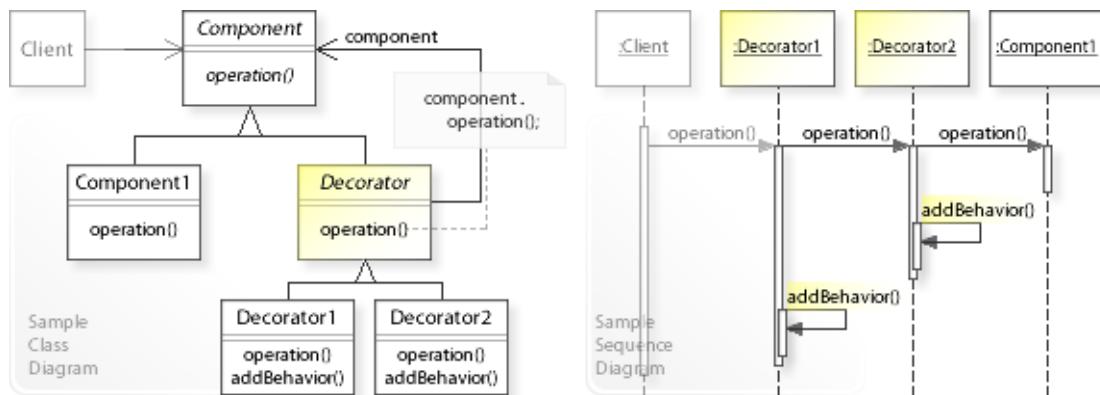
### Advantages (+)

- Provides a flexible alternative to subclassing.
  - Decorator provides a flexible alternative to extending functionality via subclassing.
  - It's easy to combine (mix, sort, duplicate, etc.) functionalities by collaborating with different decorators.
  - Subclassing would require creating a new class for each new combination of functionalities (explosion of subclasses).
- Allows an open-ended number of added functionalities.
  - Because decorators are transparent enclosures of an object, they can be nested recursively, which allows an open-ended number of added functionalities.
  - Clients do not know whether they work with an object directly or through one or multiple decorators. That's very powerful.
- Simplifies classes.
  - "Instead of trying to support all foreseeable features in a complex, customizable class, you can define a simple class and add functionality incrementally with Decorator objects." [GoF, p178]

### Disadvantages (-)

- Provides no reliability on object identity.
  - A decorator object is transparent but not identical to the decorated object.
  - Therefore, software that relies on object identity should not use decorators.

## Implementation



## Implementation Issues

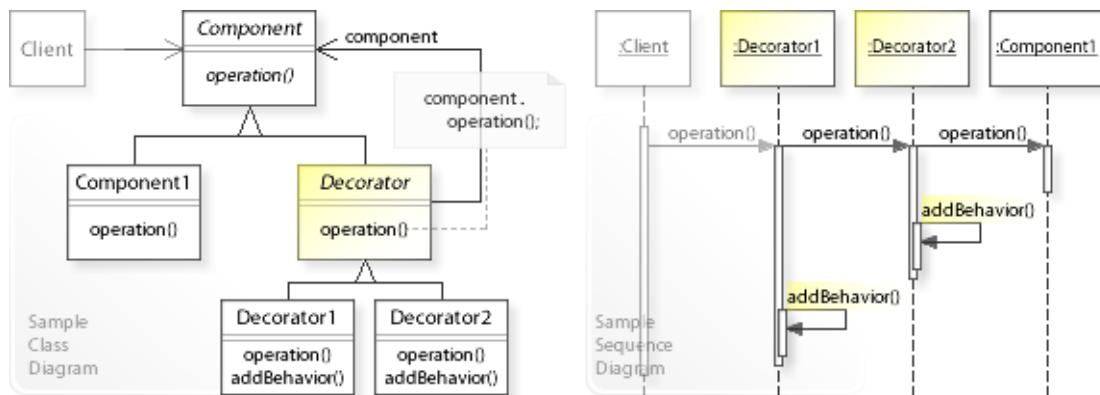
- **Interface Conformance**

- The key to the Decorator pattern is to implement the interface of the decorated object (**Component**) by forwarding all requests to it (**component.operation()**) so that its presence is transparent to the component's clients.
- This is called a *transparent enclosure*.  
"Clients generally can't tell whether they're dealing with the component or its enclosure [...]" [GoF, p44]

- **Strategy versus Decorator**

- Strategy provides a way to change an algorithm of an object dynamically at run-time.  
This is done from *inside* the object  
The object must be designed to delegate an algorithm to a strategy.  
This is a key characteristic of *object behavioral patterns*.
- Decorator provides a way to extend the functionality of an object dynamically at run-time.  
This is done from *outside* the object.  
The object already exists (already has been designed) and isn't needed to be touched.  
This is a key characteristic of *object structural patterns*.  
"*Changing the skin of an object versus changing its guts*. We can think of a decorator as a skin over an object that changes its behavior. An alternative is to change the object's guts. The Strategy(315) pattern is a good example of a pattern for changing the guts." [GoF, p179]

## Sample Code 1



### Basic Java code for implementing the sample UML diagrams.

```

1 package com.sample.decorator.basic;
2 public class Client {
3     // Running the Client class as application.
4     public static void main(String[] args) {
5         //
6         System.out.print("Working with the component directly :  ");
7         Component component = new Component1();
8         System.out.println(component.operation());
9         //
10        System.out.print("Working through decorator 1 and 2   :  ");
11        component = new Decorator1(new Decorator2(component));
12        System.out.println(component.operation());
13    }
14 }

Working with the component directly :  Hello World
Working through decorator 1 and 2   :  *** Hello World! ***

1 package com.sample.decorator.basic;
2 public abstract class Component {
3     public abstract String operation();
4 }

1 package com.sample.decorator.basic;
2 public class Component1 extends Component {
3     public String operation() {
4         return "Hello World";
5     }
6 }

1 package com.sample.decorator.basic;
2 public abstract class Decorator extends Component {
3     Component component;
4     public Decorator(Component component) {
5         this.component = component;
6     }
7     public String operation() {
8         // Forwarding to component.
9         return component.operation();
10    }
11 }

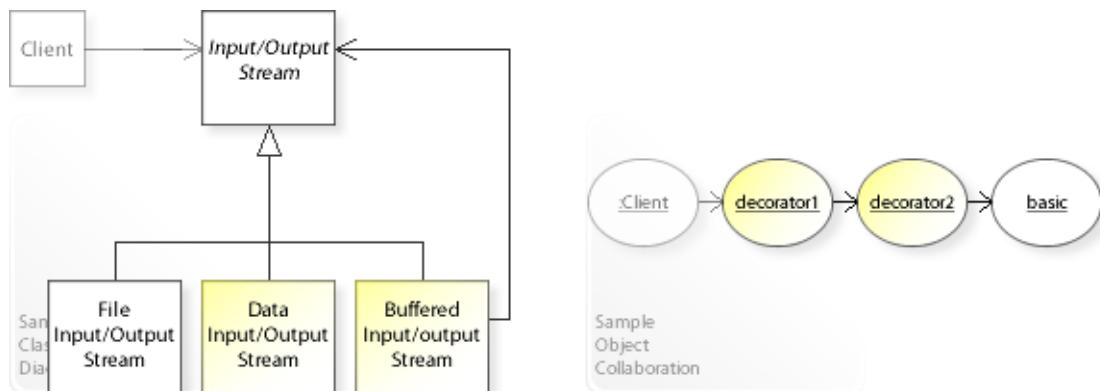
1 package com.sample.decorator.basic;
2 public class Decorator1 extends Decorator {
3     public Decorator1(Component component) {
4         super(component); // calling the super class constructor.
5     }
6     public String operation() {
7         // Forwarding to component.
8         String result = super.operation();
9         // Adding functionality to result from component.
10        return addBehavior(result);
11    }
12    private String addBehavior(String result) {

```

```
13         return "*** " + result + " ***";
14     }
15 }

1 package com.sample.decorator.basic;
2 public class Decorator2 extends Decorator {
3     public Decorator2(Component component) {
4         super(component); // calling the super class constructor.
5     }
6     public String operation() {
7         // Forwarding to component.
8         String result = super.operation();
9         // Adding functionality to result from component.
10        return addBehavior(result);
11    }
12    private String addBehavior(String result) {
13        return result + "!";
14    }
15 }
```

## Sample Code 2



### I/O Data Streams (Java Platform) / Adding functionality to basic I/O data streams.

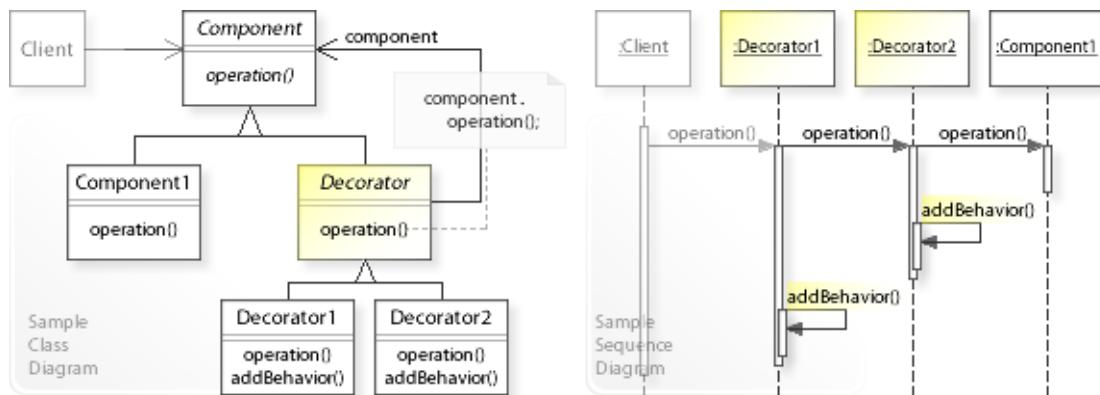
```

1 package com.sample.decorator.DataStreams;
2 import java.io.*;
3 public class Client {
4     // Running the Client class as application.
5     public static void main(String[] args) throws IOException {
6         final String FILE = "testdata";
7         //
8         // Creating decorators for FileOutputStream (out).
9         //
10        DataOutputStream out =
11            // Decorator1 adds support for writing data types
12            // (UTF-8, integer, etc.).
13            new DataOutputStream(
14                // Decorator2 adds support for buffered output.
15                new BufferedOutputStream(
16                    // Basic binary output stream.
17                    new FileOutputStream(FILE)));
18        //
19        // Working through the decorators (out).
20        //
21        out.writeUTF("ABC "); // writes string in UTF-8 format
22        out.writeInt(123); // writes integer data type
23        out.close();
24        //
25        // Creating decorators for FileInputStream (in).
26        //
27        DataInputStream in =
28            // Decorator1 adds support for reading data types
29            // (UTF-8, integer, etc.).
30            new DataInputStream(
31                // Decorator2 adds support for buffered input.
32                new BufferedInputStream(
33                    // Basic binary input stream.
34                    new FileInputStream(FILE)));
35        //
36        // Working through the decorators (in).
37        //
38        // in.readUTF() reads string in UTF-8 format.
39        // in.readInt() reads integer data type.
40        System.out.println(in.readUTF() + in.readInt());
41        in.close();
42    }
43 }

```

ABC 123

## Related Patterns



### Key Relationships (see also Overview)

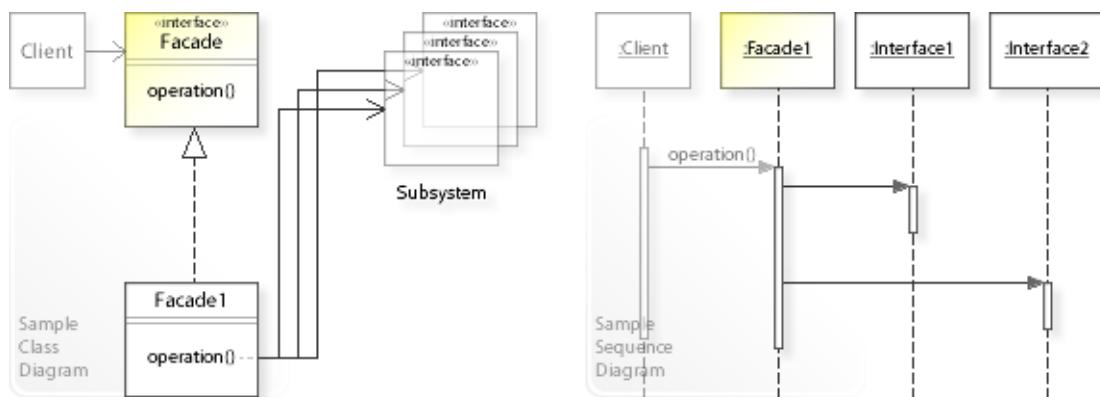
- **Strategy - Decorator**

- Strategy provides a way to exchange the algorithm of an object dynamically. This is done from *inside* the object. The object is designed to delegate an algorithm to a `strategy` object.
- Decorator provides a way to extend the functionality of an object dynamically. This is done from *outside* the object. The object already exists and isn't needed to be touched. That's very powerful.

- **Decorator - Proxy**

- Decorator provides a way to extend the functionality of an object dynamically.
- Proxy provides a way to restrict, extend, and change the functionality of an object.

## Intent



The intent of the Facade design pattern is to:

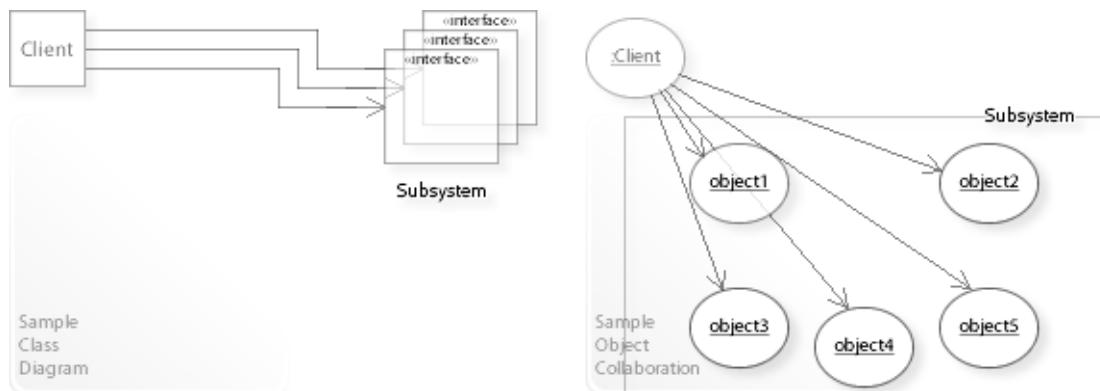
**"Provide an unified interface to a set of interfaces in a subsystem."**

**Facade defines a higher-level interface that makes the subsystem easier to use."** [GoF]

See Problem and Solution sections for a more structured description of the intent.

- The Facade design pattern solves problems like:
  - *How can a complex subsystem be made easier to use?*
  - *How can dependencies between a subsystem and its clients be minimized?*
- For example, a complex subsystem should provide a simplified (high-level) view that is good enough for most clients that merely need some basic functionalities.  
Clients that need more lower-level functionalities should be able to access the interfaces in the subsystem directly.
- The Facade pattern describes how to solve such problems:
  - *Provide an unified interface to a set of interfaces in a subsystem:*  
Facade | operation().
  - Clients of the subsystem only refer to and know about the (simple) facade interface and are independent of the (many/complex) interfaces in the subsystem, which makes clients easier to implement, change, test, and reuse.

## Problem



The intent of the Facade design pattern is to solve problems like:

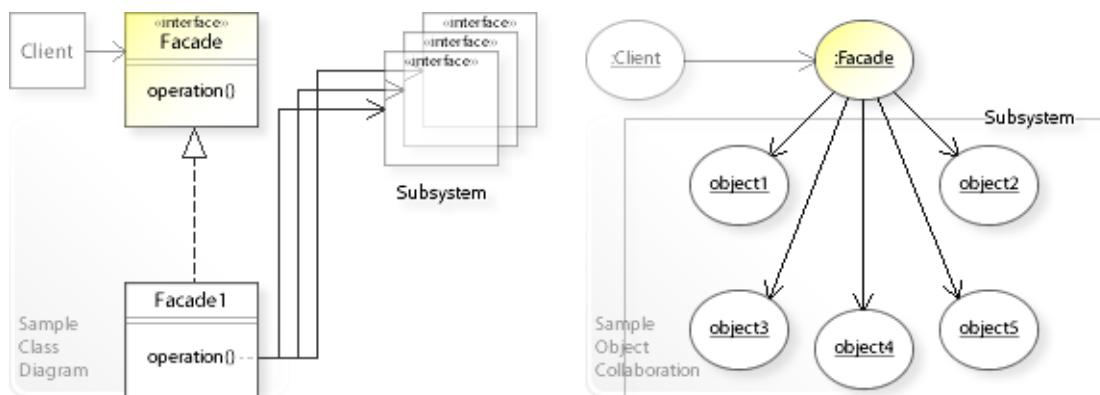
***How can a complex subsystem be made easier to use?***

***How can dependencies between a subsystem and its clients be minimized?***

See Applicability section for all problems Facade can solve.

- Complex software systems are often structured (layered) into subsystems. Clients of a subsystem must refer to and know about many and/or complex interfaces in the subsystem, which makes clients hard to implement, change, test, and reuse.
- "A common design goal is to minimize the communication and dependencies between subsystems." [GoF, p185]
- For example, structuring (layering) a software system into multiple subsystems. It's a vital requirement to minimize the dependencies between subsystems or between a subsystem and its clients. It should be possible to minimize the number of interfaces clients must refer to and know about.
- For example, a complex subsystem should provide a simplified (high-level) view that is good enough for most clients that merely need some basic functionalities. Clients that need more lower-level functionalities should be able to access the interfaces in the subsystem directly.

## Solution



The Facade pattern describes a solution:

**Define a separate `Facade` object that provides an unified interface for a set of interfaces in a subsystem.**

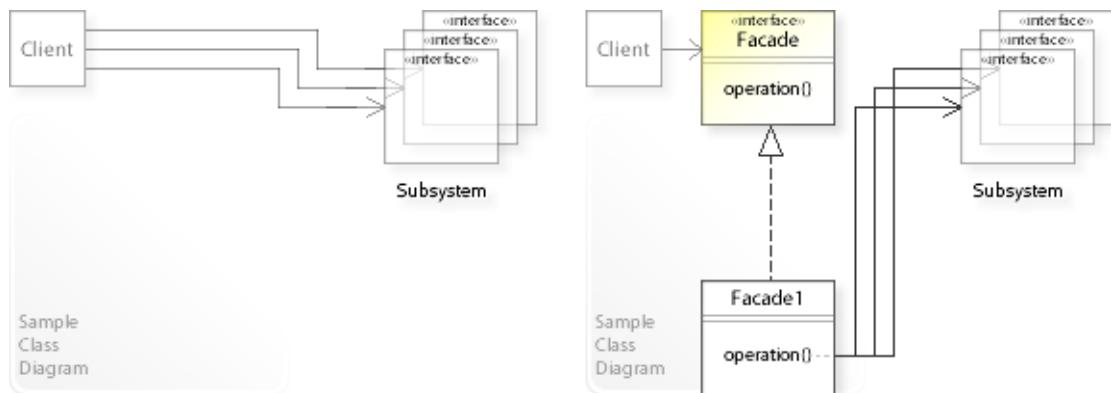
**Work through a `Facade` object to make a complex subsystem easier to use.**

Describing the Facade design in more detail is the theme of the following sections.

See Applicability section for all problems Facade can solve.

- The key idea in this pattern is to make a complex subsystem easier to use independently from (without changing) the subsystem itself.
- **Define a separate `Facade` object:**
  - Define an unified interface for a set of interfaces in a subsystem (`Facade` | `operation()`).
  - Define classes (`Facade1`,...) that implement the facade interface in terms of (by delegating to) the interfaces in the subsystem.
- **Work through a `Facade` object to make a complex subsystem easier to use.**  
This greatly minimizes and simplifies dependencies between subsystems or between a subsystem and its clients (loose coupling).  
Clients that need more lower-level functionalities are able to access the interfaces in the subsystem directly.

## Motivation 1



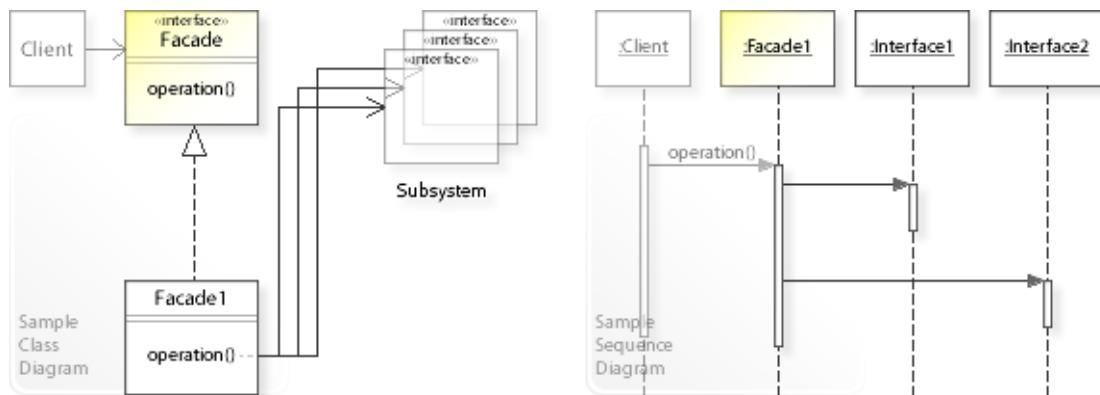
### **Consider the left design (problem):**

- Complicated clients (tight coupling).
    - Clients refer to and know about (depend on) many/complex interfaces in the subsystem, which makes clients hard to implement, change, test, and reuse.
    - Clients must be changed when interfaces in the subsystem are added or extended.

### **Consider the right design (solution):**

- Simplified clients (loose coupling).
    - Clients only refer to and know about (depend on) the simple facade interface which makes clients easier to implement, change, test, and reuse.
    - Clients do not have to be changed when interfaces in the subsystem are added or extended.

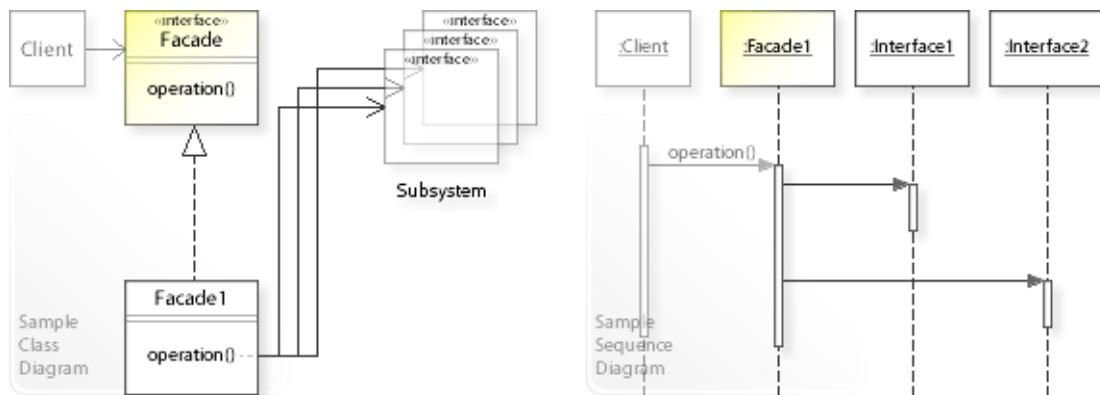
## Applicability



## Design Problems

- **Making Complex Subsystems Easier to Use**
  - How can a complex subsystem be made easier to use?
  - How can a single entry point be provided for a subsystem?
- **Avoiding Tight Coupling**
  - How can dependencies between subsystems be minimized?
  - How can dependencies between a subsystem and its clients be minimized?

## Structure, Collaboration



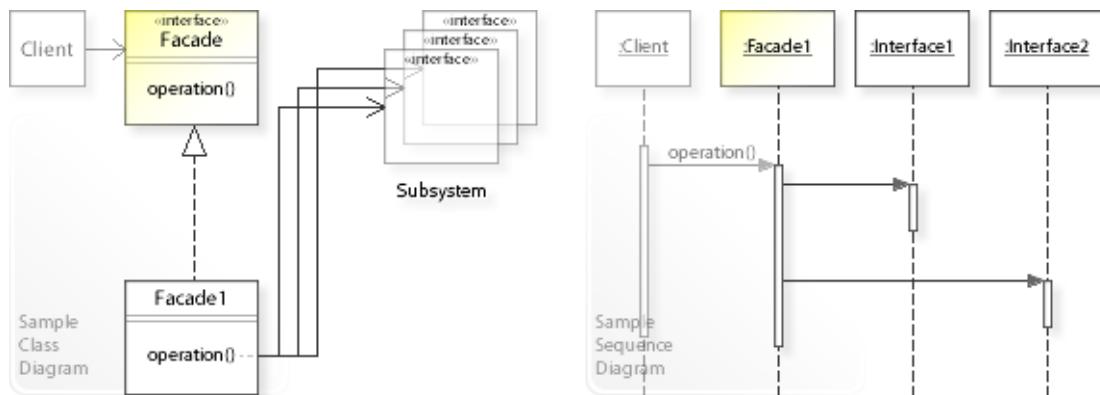
### Static Class Structure

- *Facade*
  - Defines an unified (higher-level) interface for a set of interfaces in a subsystem.
- *Facade1,...*
  - Implement the *Facade* interface in terms of (by delegating to) the interfaces in the subsystem.

### Dynamic Object Collaboration

- In this sample scenario, a *Client* object works through a *Facade* object to access a set of interfaces in a subsystem.
- The *Client* object calls *operation()* on the *Facade1* object.
- *Facade1* delegates the request to objects of type *Interface1* and *Interface2* that fulfill the request.
- *Facade1* may do work of its own before and/or after forwarding a request.
- See also Sample Code / Example 1.

## Consequences

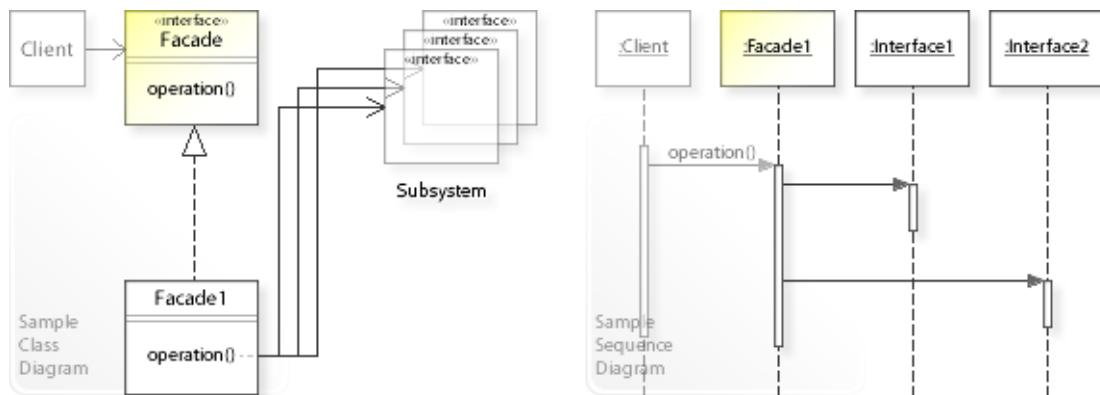


### Advantages (+)

- Simplifies clients.
  - Clients only refer to and know about the simple **Facade** interface and are independent of the complex subsystem.
  - This makes clients easier to implement, change, test, and reuse.
- Decouples clients from a subsystem.
  - Clients are decoupled from the subsystem by working through a **Facade** object.
  - Loosely coupled objects are easier to implement, change, test, and reuse.
- Decouples subsystems.
  - When layering a complex system, Facade can define a single entry point for each subsystem.
  - Subsystems collaborate with each other solely through their facades, which reduces and simplifies dependencies between subsystems.

### Disadvantages (-)

## Implementation

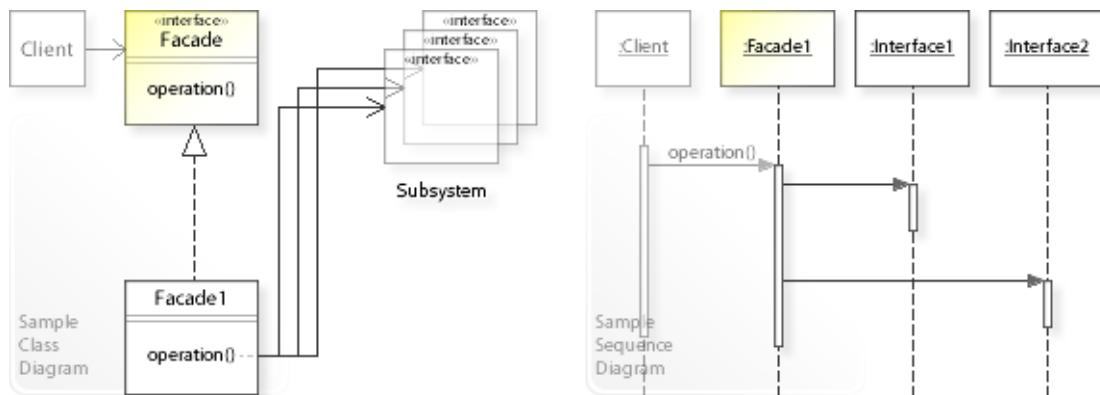


### Implementation Issues

- **Implementation Variants**

- The Facade interface is implemented in terms of (by delegating to) the appropriate interfaces in the subsystem.
- "Although the subsystem objects perform the actual work, the facade may have to do work of its own to translate its interface to subsystem interfaces." [GoF, p187]
- "A facade can provide a simple default view of the subsystem that is good enough for most clients. Only clients needing more customizability will need to look behind the facade." [GoF, p186]

## Sample Code 1



### Basic Java code for implementing the sample UML diagrams.

```

1 package com.sample.facade.basic;
2 public class Client {
3     // Running the Client class as application.
4     public static void main(String[] args) {
5         // Creating a facade for a subsystem.
6         Facade facade = new Facade1(new Class1(), new Class2());
7         System.out.println(
8             // Working through the facade.
9             facade.operation());
10    }
11 }

Facade delegates to the subsystem ...
Subsystem Interface1 returns: Hello
Subsystem Interface2 returns: World!
Facade returns: Hello World!

1 package com.sample.facade.basic;
2 public interface Facade {
3     String operation();
4 }

1 package com.sample.facade.basic;
2 public class Facade1 implements Facade {
3     private Interface1 object1;
4     private Interface2 object2;
5     // ...
6     public Facade1(Interface1 object1, Interface2 object2) {
7         this.object1 = object1;
8         this.object2 = object2;
9     }
10    public String operation() {
11        System.out.println("Facade delegates to the subsystem ...");
12        //
13        String str1 = " Subsystem Interface1 returns: " + object1.operation1();
14        String str2 = "\n Subsystem Interface2 returns: " + object2.operation2();
15        //
16        return str1 + str2 + "\nFacade returns: " +
17            object1.operation1() + object2.operation2();
18    }
19 }

```

Subsystem interfaces:

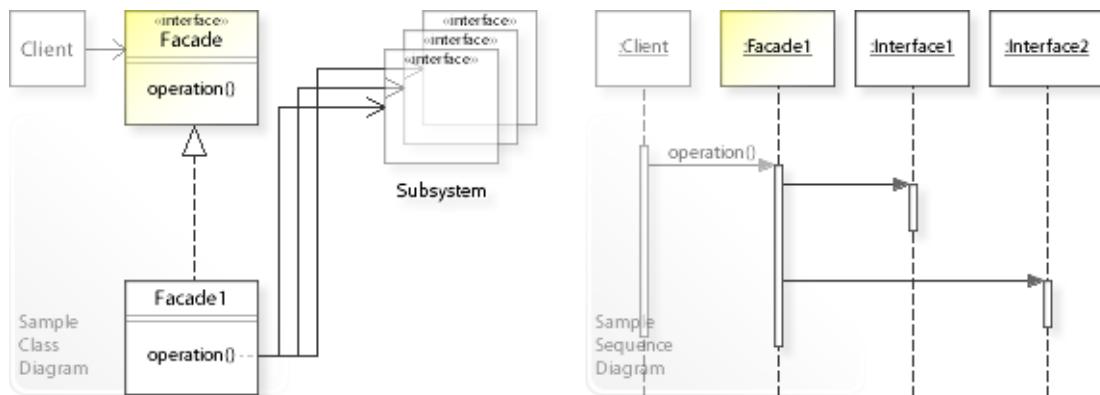
```
1 package com.sample.facade.basic;
2 public interface Interface1 {
3     String operation1();
4 }

1 package com.sample.facade.basic;
2 public class Class1 implements Interface1 {
3     public String operation1() {
4         return "Hello ";
5     }
6 }

1 package com.sample.facade.basic;
2 public interface Interface2 {
3     String operation2();
4 }

1 package com.sample.facade.basic;
2 public class Class2 implements Interface2 {
3     public String operation2() {
4         return "World!";
5     }
6 }
```

## Related Patterns

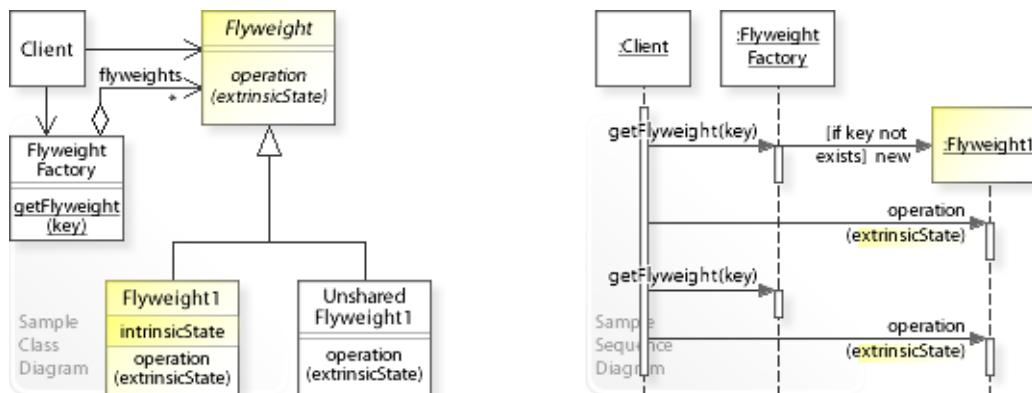


### Key Relationships (see also Overview)

- **Adapter - Facade**

- Adapter does not create or change an interface,  
it merely forwards client requests to an incompatible interface  
(Object Adapter).
- Facade creates a new (simple) interface  
for a set of (complex) interfaces.

## Intent

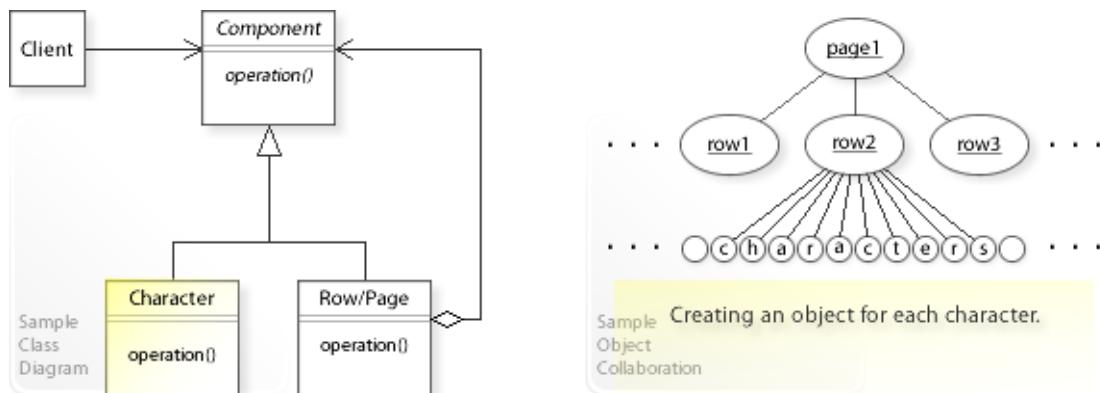


The intent of the Flyweight design pattern is to:

**"Use sharing to support large numbers of fine-grained objects efficiently."** [GoF]  
See Problem and Solution sections for a more structured description of the intent.

- The Flyweight design pattern solves problems like:
  - *How can large numbers of fine-grained objects be supported efficiently?*
- For example, to represent a text document at the finest levels, an object is needed for each character in the document.
- The Flyweight pattern describes how to solve such problems:
  - *Use sharing to support large numbers of fine-grained objects efficiently.*
  - Define Flyweight objects that store intrinsic state.  
Share Flyweight objects (intrinsic state) and pass in extrinsic state dynamically at run-time when invoking a  
**<object>Flyweight</object>**  
**operation(operation(extrinsicState))**.
  - Intrinsic state is invariant (context independent) and therefore can be shared.  
Extrinsic state is variant (context dependent) and therefore can *not* be shared.

## Problem



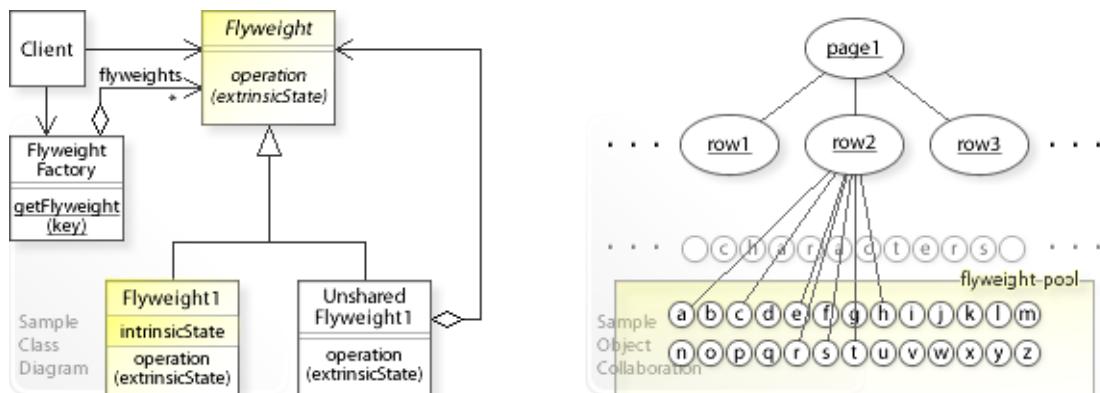
The Flyweight design pattern solves problems like:

***How can large numbers of fine-grained objects be supported efficiently?***

See Applicability section for all problems Flyweight can solve.

- A naive way to support large numbers of objects in an application is to create an object each time it is needed.  
For example, text editing applications.  
To represent a text document at the finest levels, an object is created for every occurrence of a character in a document. A `Character` object stores information about its character code, font, location in the document, etc.
- *That's the kind of approach to avoid.* "Even moderate-sized documents may require hundreds of thousands of character objects, which will consume lots of memory and may incur unacceptable run-time overhead." [GoF, p195]
- It should be possible to reduce the number of *physically* created objects. *Logically*, there should be an object for every occurrence of a character in the document.
- For example, language processing and translation applications.  
It should be possible to process any size of documents efficiently.

## Solution



The Flyweight pattern describes a solution:

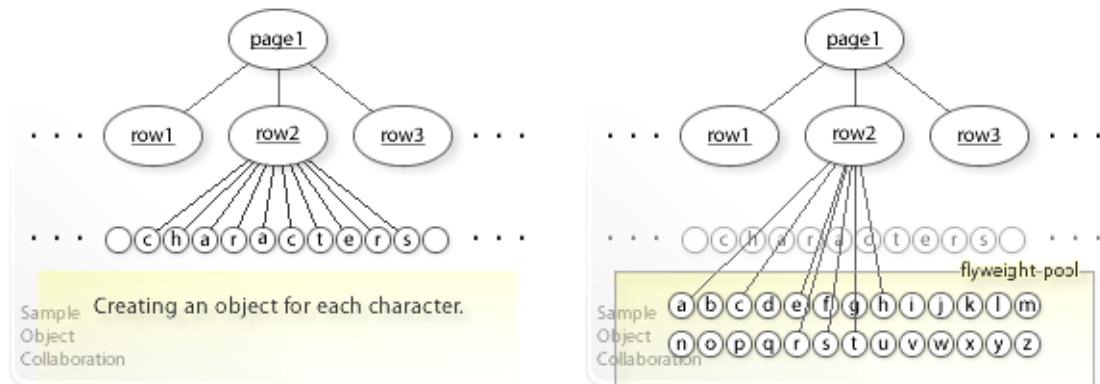
**Define separate Flyweight objects that store intrinsic state.**

**Share Flyweight objects (intrinsic state) and pass in extrinsic state.**

**See Applicability section for all problems Flyweight can solve.** Describing the Flyweight design in more detail is the theme of the following sections.

- "The key concept here is the distinction between **intrinsic** and **extrinsic** state." [GoF]  
Flyweight objects store intrinsic state, and clients must pass in extrinsic state.  
**Intrinsic** state is invariant (context independent) and therefore can be shared.  
For example, a `Character` object's character code.  
**Extrinsic** state is variant (context dependent) and therefore can *not* be shared.  
For example, a `Character` object's font and location.
- **Define separate Flyweight objects:**
  - Define an interface through which extrinsic state can be passed in `(Flyweight | operation(extrinsicState))`.
  - Define classes (`Flyweight1, ...`) that implement the `Flyweight` interface and store intrinsic state (that can be shared).
- **Share (reuse) Flyweight objects (intrinsic state) and pass in extrinsic state** (`flyweight.operation(extrinsicState)`).
  - Clients are responsible for passing in extrinsic state dynamically at run-time when they invoke a `Flyweight` operation.
  - To ensure that `Flyweight` objects are shared properly, clients must obtain flyweights solely from the `Flyweight` factory (`getFlyweight(key)`) that maintains a pool of shared `Flyweight` objects.

## Motivation 1



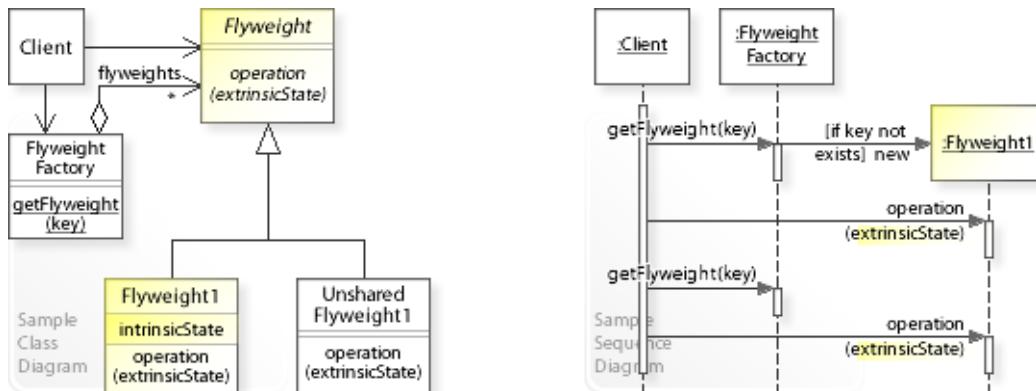
**Consider the left design (problem):**

- Large number of objects.
  - To represent a text document, an object is created for each character in the document.
  - The number of character objects depends on the number of characters in the document.

**Consider the right design (solution):**

- Small number of objects.
  - To represent a text document, a `Flyweight` object is created for each character in the used character set (`flyweight pool`).
  - The number of character objects is independent of the number of characters in the document.
  - A flyweight stores only the intrinsic state (for example, the character code). Clients provide the extrinsic state dynamically at run-time (for example, the current position, font, and color of the character in the document).

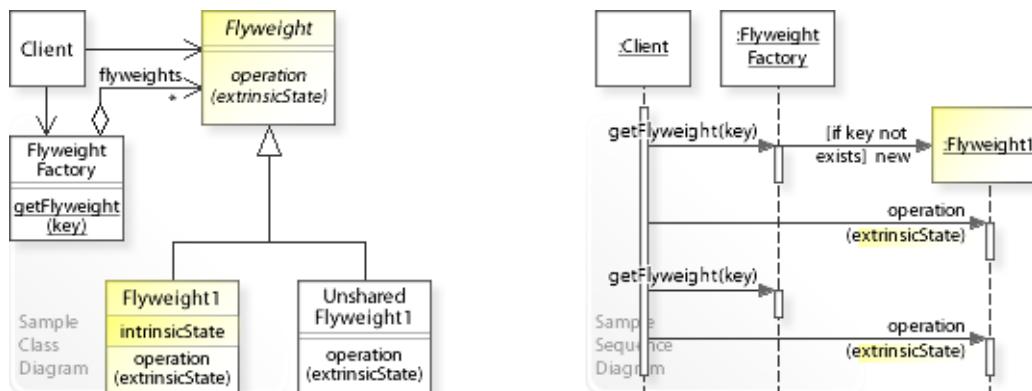
## Applicability



## Design Problems

- **Supporting Large Numbers of Objects**
  - How can large numbers of fine-grained objects be supported efficiently?

## Structure, Collaboration



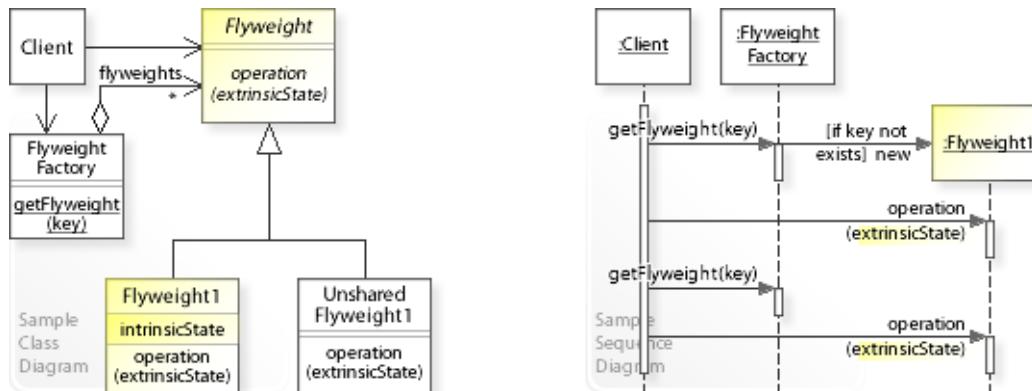
### Static Class Structure

- **Flyweight**
  - Defines a common interface for shared and unshared objects through which extrinsic state can be passed in.
- **Flyweight1,...**
  - Implement the **Flyweight** interface.
  - Store intrinsic (invariant) state that can be shared.
- **UnsharedFlyweight1,...**
  - Implement the **Flyweight** interface.
  - Have no intrinsic (invariant) state and can not be shared.
- **FlyweightFactory**
  - Maintains a container (pool) of shared **Flyweight** objects (**flyweights**).
  - To ensure that **Flyweight** objects are shared properly, clients must obtain flyweights solely from the **FlyweightFactory** (`getFlyweight(key)`).

### Dynamic Object Collaboration

- In this sample scenario, a **Client** shares **Flyweight** objects by getting them from a **FlyweightFactory** that ensures that they are shared properly.
- The interaction starts with the **Client** that calls `getFlyweight(key)` on the **FlyweightFactory**.
- Assuming that the flyweight does not already exist, the **FlyweightFactory** **creates** a new **Flyweight1** object and returns it to the **Client**.
- The **Client** calls `operation(extrinsicState)` on the returned (new) **Flyweight1** and passes in the extrinsic state.
- Thereafter, the **Client** again calls `getFlyweight(key)` on the **FlyweightFactory**.
- Assuming that the flyweight already exists, the **FlyweightFactory** **shares** (reuses) the existing **Flyweight1** object and returns it to the **Client**.
- The **Client** calls `operation(extrinsicState)` on the returned (shared) **Flyweight1** and passes in the extrinsic state.
- See also Sample Code / Example 1.

## Consequences



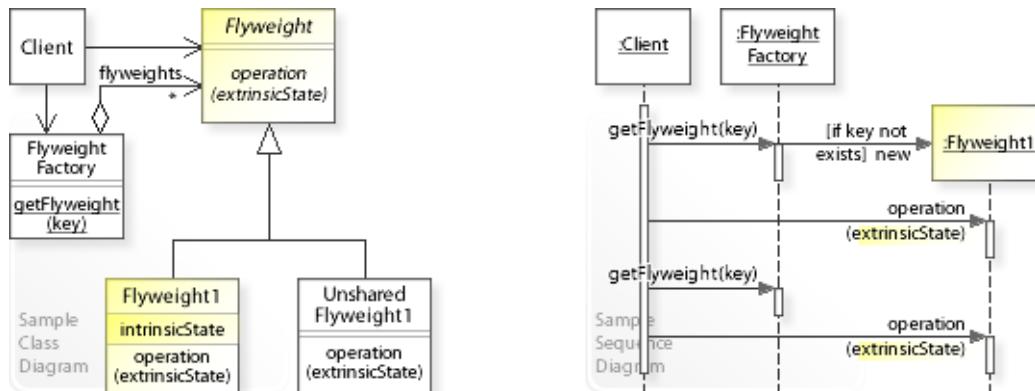
### Advantages (+)

- Enables abstraction at the finest levels.
  - An open-ended number of logically different objects can be represented by a far less constant number of physically created Flyweight objects.

### Disadvantages (-)

- May introduce run-time costs.
  - Clients are responsible for passing in extrinsic state dynamically at run-time.
  - Retrieving and computing extrinsic state each time a flyweight is used may impact memory usage and system performance.
- Provides no reliability on object identity.
  - Because Flyweight objects are shared, logically different objects are represented by the same physically created object.
  - Therefore, testing the identifier (key) of an object will return true for logically different objects.

## Implementation

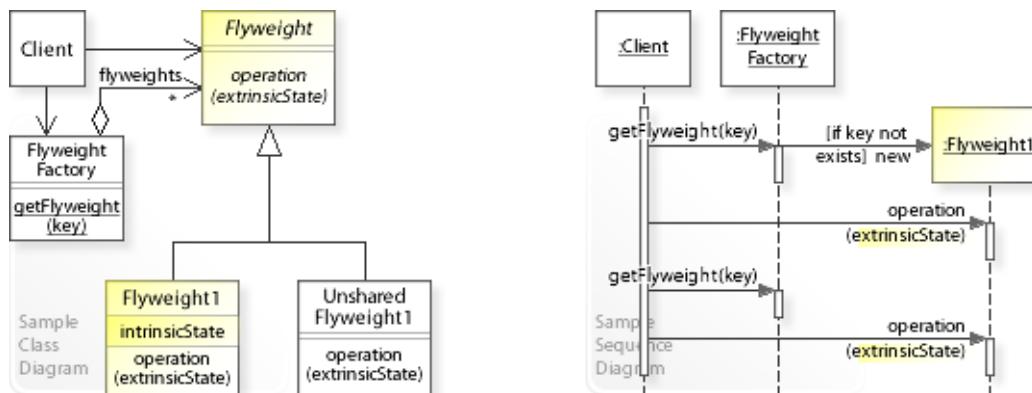


### Implementation Issues

- **Flyweight Factory**

- To ensure that **Flyweight** objects are shared properly, clients must obtain flyweights solely from the flyweight factory (**getFlyweight(key)**).
- A flyweight factory maintains a pool of flyweights.  
If the requested flyweight already exists, it is returned.  
Otherwise, it is created, added to the pool, and returned.
- The **key** is needed to look up the flyweight in the pool (see Sample Code / Basic).

## Sample Code 1



### Basic Java code for implementing the sample UML diagrams.

```

1 package com.sample.flyweight.basic;
2 public class Client {
3     // Running the Client class as application.
4     public static void main(String[] args) {
5         Flyweight flyweight;
6         // Getting a FlyweightFactory object.
7         FlyweightFactory flyweightFactory = FlyweightFactory.getInstance();
8         //
9         flyweight = flyweightFactory.getFlyweight("A");
10        System.out.println(flyweight.operation(100));
11        //
12        flyweight = flyweightFactory.getFlyweight("A");
13        System.out.println(flyweight.operation(200));
14        //
15        flyweight = flyweightFactory.getFlyweight("A");
16        System.out.println(flyweight.operation(300));
17        //
18        System.out.println("*** Number of flyweights created: " +
19                      flyweightFactory.getSize());
20    }
21 }

Creating a flyweight with key A ...
    performing an operation with passed in extrinsic state 100.
Sharing a flyweight with key A ...
    performing an operation with passed in extrinsic state 200.
Sharing a flyweight with key A ...
    performing an operation with passed in extrinsic state 300.
*** Number of flyweights created: 1

1 package com.sample.flyweight.basic;
2 public interface Flyweight {
3     public String operation(int extrinsicState);
4 }

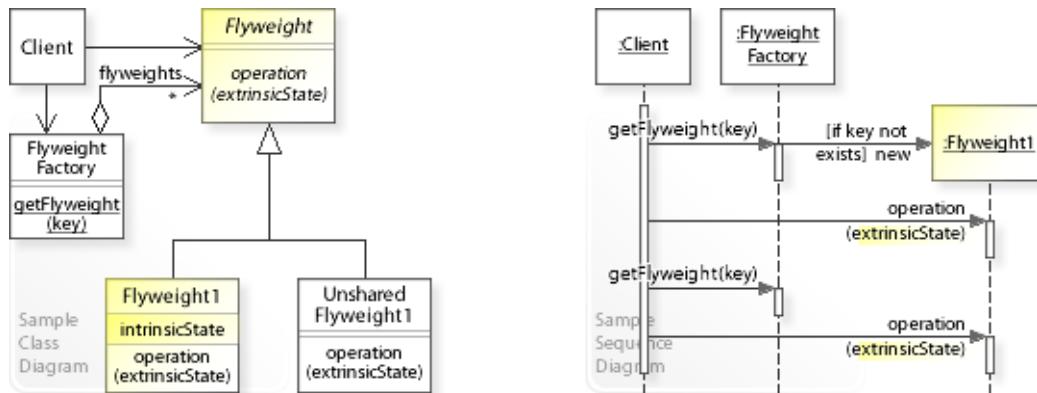
1 package com.sample.flyweight.basic;
2 public class Flyweight1 implements Flyweight {
3     private String intrinsicState;
4     public Flyweight1(String intrinsicState) {
5         this.intrinsicState = intrinsicState;
6     }
7     public String operation(int extrinsicState) {
8         return " performing an operation with passed in extrinsic state " +
9                 extrinsicState + ".";
10    }
11 }

1 package com.sample.flyweight.basic;
2 import java.util.HashMap;
3 import java.util.Map;
4 public class FlyweightFactory {
5     // Implemented as Singleton.
6     // See also Singleton / Implementation / Variant 1.

```

```
7     private static final FlyweightFactory INSTANCE = new FlyweightFactory();
8     private FlyweightFactory() { }
9     public static FlyweightFactory getInstance() {
10         return INSTANCE;
11     }
12     // Shared flyweight pool.
13     private Map<String, Flyweight> flyweights = new HashMap<String, Flyweight>();
14     // Creating and maintaining shared flyweights.
15     public Flyweight getFlyweight(String key) {
16         if (flyweights.containsKey(key)) {
17             System.out.println("Sharing a flyweight with key " + key + " ... ");
18             return flyweights.get(key);
19         } else {
20             System.out.println("Creating a flyweight with key " + key + " ... ");
21             Flyweight flyweight = new Flyweight1(key); // assuming key = intrinsic state
22             flyweights.put(key, flyweight);
23             return flyweight;
24         }
25     }
26     public int getSize() {
27         return flyweights.size();
28     }
29 }
```

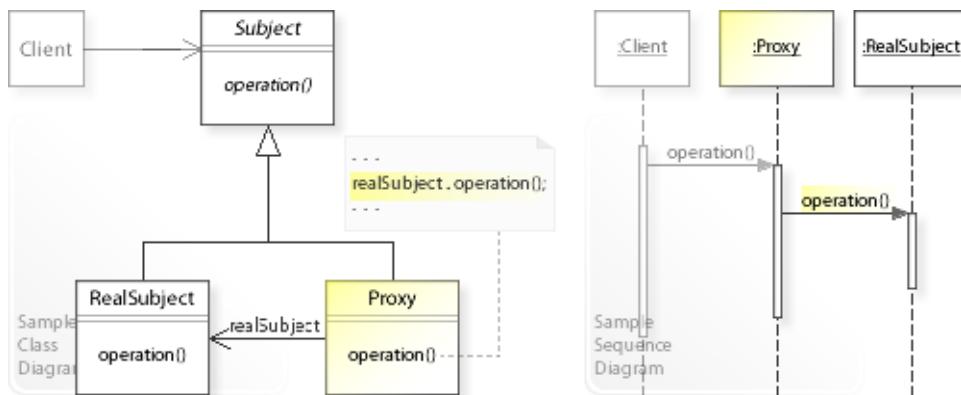
## Related Patterns



### Key Relationships (see also Overview)

- **Composite - Flyweight**
  - Composite and Flyweight often work together.  
Leaf objects can be implemented as shared flyweight objects.
- **Flyweight - Singleton**
  - The flyweight factory is usually implemented as Singleton.

## Intent



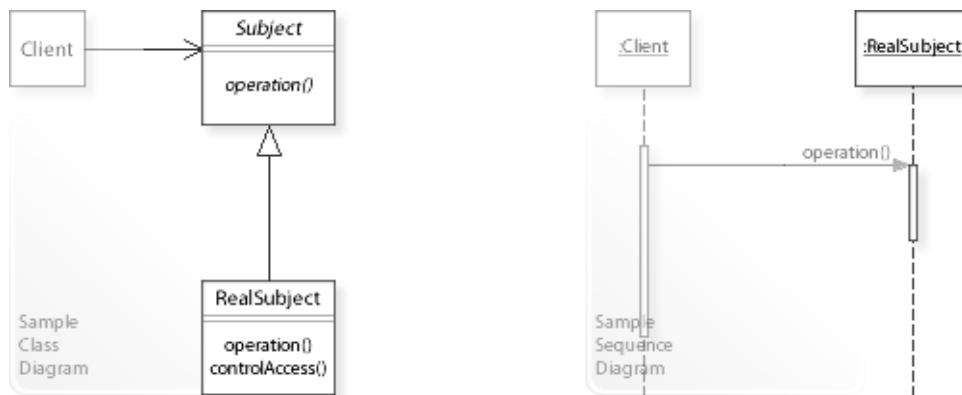
The intent of the Proxy design pattern is to:

**"Provide a surrogate or placeholder for another object to control access to it."** [GoF]

See Problem and Solution sections for a more structured description of the intent.

- The Proxy design pattern solves problems like:
  - *How can the access to an object be controlled?*
  - *How can additional or stand-in functionality be provided when accessing an object?*
- Often we want to control the access to an object.
- For example, when accessing *sensitive*, *expensive*, or *remote* objects.
- The Proxy pattern describes how to solve such problems:
  - *Provide a surrogate or placeholder for another object to control access to it.*  
Define a separate **Proxy** object that provides a surrogate or placeholder for an object (**RealSubject**). A proxy implements the **Subject** interface so that it can act anywhere a subject is expected.
  - Work through a **Proxy** object to control the access to an object (**RealSubject**).

## Problem



The intent of the Proxy design pattern is to solve problems like:

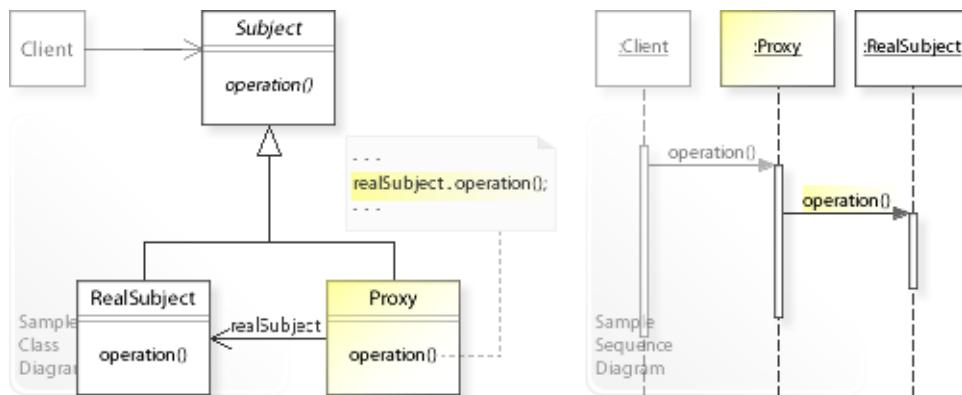
***How can the access to an object be controlled?***

***How can additional or stand-in functionality be provided when accessing an object?***

See Applicability section for all problems Proxy can solve.

- An inflexible way to control the access to an object (`RealSubject`) is to implement the functionality directly into the `RealSubject` class (`controlAccess()`). But it's impossible to change an (already existing) object each time we want to control the access to it.
- *That's the kind of approach to avoid if we want to control the access to an object without changing the object.*
- For example, when accessing *sensitive* objects, it should be possible to check that clients have the required access rights.
- For example, when accessing *expensive* objects, it should be possible to create them on demand (i.e., to defer their instantiation until they are actually needed) and to cache their data.
- For example, when accessing *remote* objects, it should be possible to provide stand-in functionality to hide network communication from clients.
- *Generally, when accessing such objects, it should be possible to provide additional or stand-in functionality.*  
"[...] whenever there is a need for a more versatile or sophisticated reference to an object than a simple pointer." [GoF, p208]

## Solution



The Proxy pattern describes a solution:

**Define a separate `Proxy` object that provides a surrogate or placeholder for an object (`RealSubject`) to control the access to it.**

**Work through a `Proxy` object to control the access to an object.**

Describing the Proxy design in more detail is the theme of the following sections.

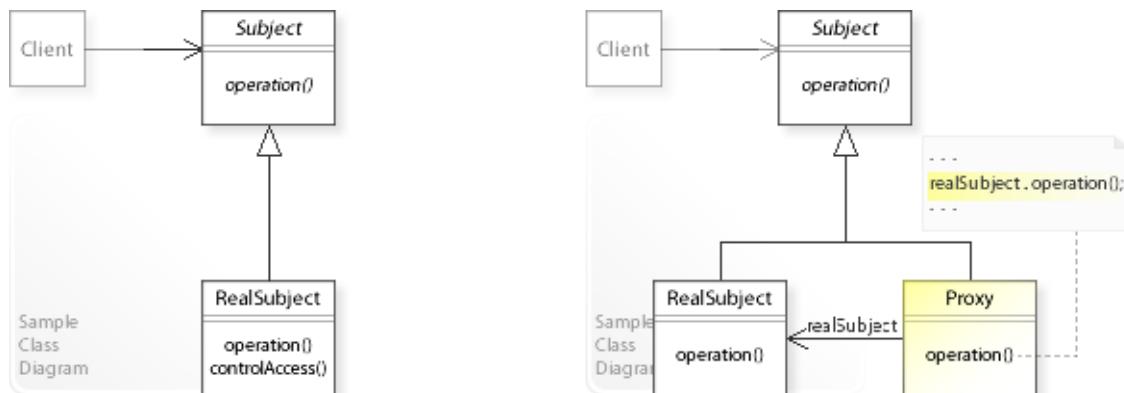
See Applicability section for all problems Proxy can solve.

- The key idea in this pattern is to control the access to an (already existing) object independently from (without changing) the object.
- **Define a separate `Proxy` object:**
  - A proxy implements the `Subject` interface so that it can act as a stand-in anywhere a subject is expected. Clients do not know whether they work with a subject directly or through its proxy.
  - Usually, a proxy has direct access to the concrete `RealSubject` class.
  - A proxy can implement arbitrary functionality.

"The Proxy pattern introduces a level of indirection when accessing an object. The additional indirection has many uses, depending on the kind of proxy:" [GoF, p210] For example:

  - A *protection proxy* checks access rights of clients.
  - A *virtual proxy* defers creating an *expensive* object until it is actually needed.
  - A *remote proxy* hides network details from clients.
- **Work through a `Proxy` object to control the access to an object.**
- A proxy can restrict, extend, or change the functionality of an object.

## Motivation 1



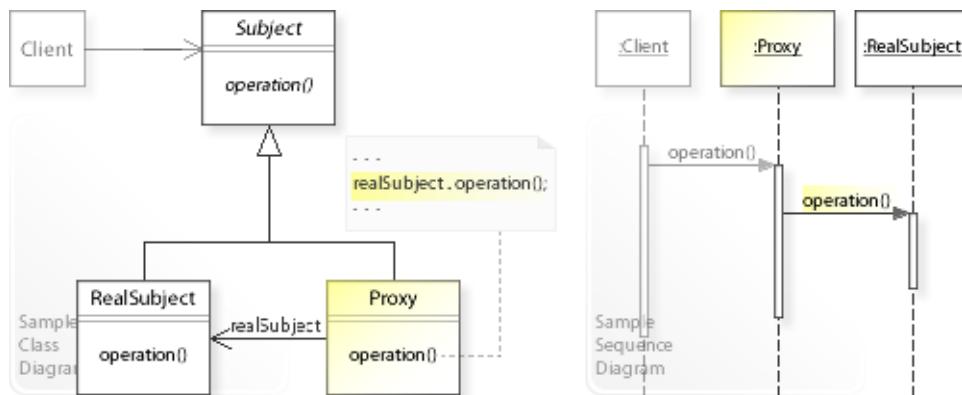
**Consider the left design (problem):**

- Complicated `RealSubject`.
  - The functionality to control the access is implemented directly into the `RealSubject` class.
  - `RealSubject` gets more complex and harder to implement, change, test, and reuse.
  - But it's impossible to change an (already existing) object each time we want to control the access to it.

**Consider the right design (solution):**

- Simplified `RealSubject`.
  - The functionality to control the access is implemented (encapsulated) in a separate `Proxy` class.
  - The `RealSubject` class does not have to be changed.
  - It's possible to control the access to an (already existing) object without having to change it.

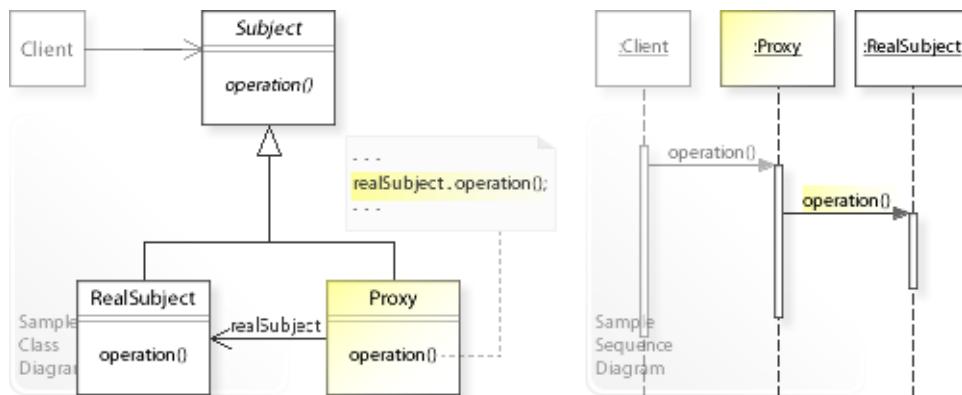
## Applicability



## Design Problems

- **Controlling Access to Objects**
  - How can the access to an object be controlled?
  - How can additional or stand-in functionality be provided when accessing an object?

## Structure, Collaboration



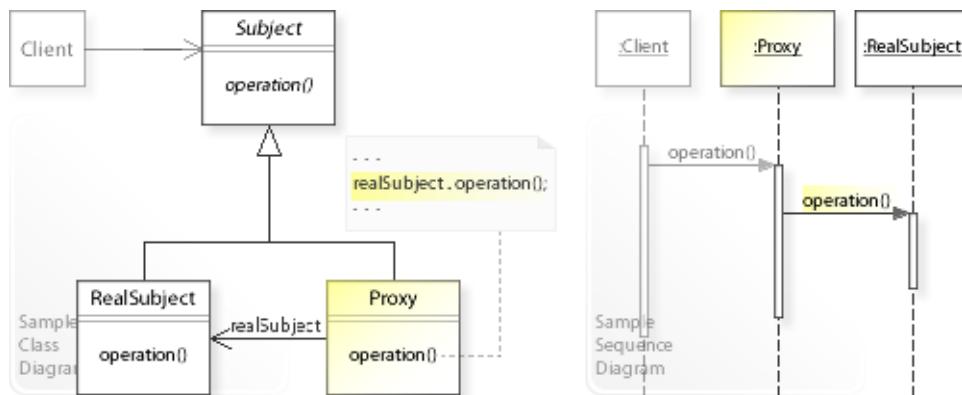
### Static Class Structure

- **Subject**
  - Defines an interface for objects that need access control.
- **RealSubject**
  - Implements the **Subject** interface.
- **Proxy**
  - Implements the **Subject** interface.
  - Implements arbitrary functionality to control the access to **RealSubject**.
  - Maintains a reference (**realSubject**) to a **RealSubject** object.

### Dynamic Object Collaboration

- In this sample scenario, a **Client** object works through an **Proxy** object that controls access to a **RealSubject** object.
- The **Client** object calls **operation()** on the **Proxy** object.
- **Proxy** controls the access to **RealSubject** and may or may not forward the request to **RealSubject**.
- See also Sample Code / Example 1.

## Consequences



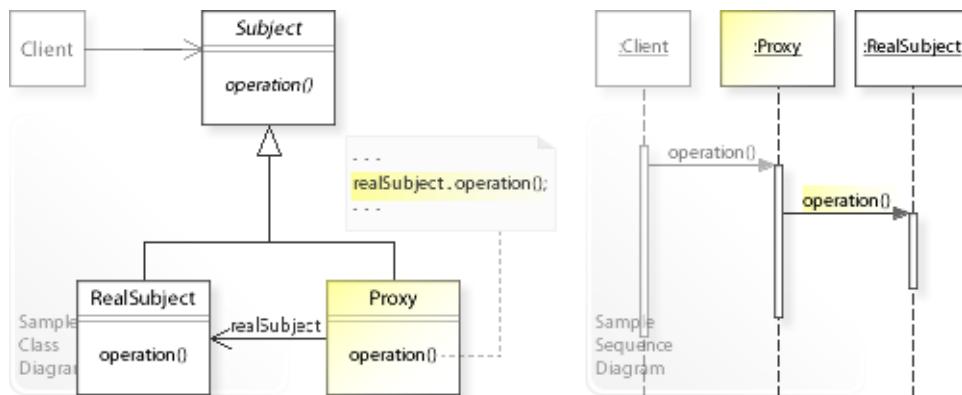
### Advantages (+)

- Simplifies real subject.
  - The functionality to control the access isn't hard-wired directly into the real subject, which simplifies it.
  - We can control the access to the real subject without changing it.

### Disadvantages (-)

- Proxy is coupled to real subject.
  - A proxy implements the **Subject** interface and (usually) has direct access to the concrete **RealSubject** class.
  - "But if Proxies are going to instantiate RealSubject (such as in virtual proxy), then they have to know the concrete class." [GoF, p213]

## Implementation



### Implementation Issues

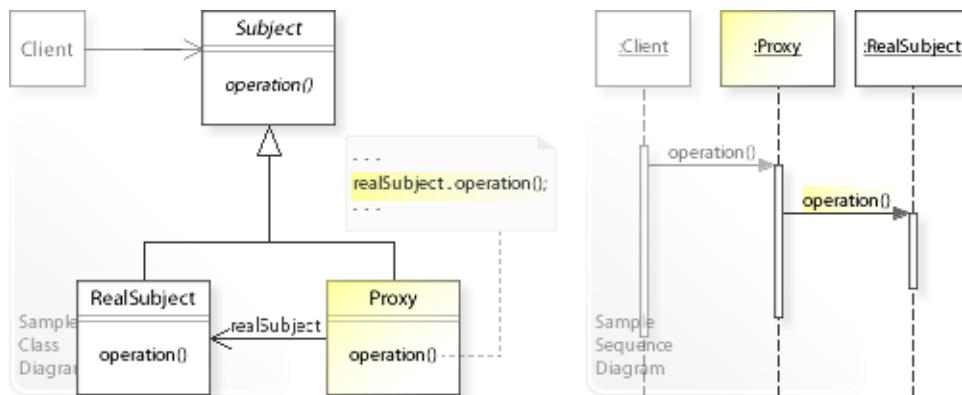
- **Interface Conformance**

- Proxy implements the **Subject** interface so that it can act as a surrogate or placeholder anywhere a subject is expected.
- Clients generally can't tell whether they're dealing with the subject directly or through a proxy.
- Proxy (usually) has direct access to the concrete **RealSubject** class.

- **Implementation Variants**

- A proxy can implement arbitrary functionality.
- "The Proxy pattern introduces a level of indirection when accessing an object. The additional indirection has many uses, depending on the kind of proxy:" [GoF, p210]

## Sample Code 1



### Basic Java code for implementing the sample UML diagrams.

```

1 package com.sample.proxy.basic;
2 public class Client {
3     // Running the Client class as application.
4     public static void main(String[] args) {
5         // Creating a proxy for a real subject.
6         Proxy proxy = new Proxy(new RealSubject());
7         System.out.println(
8             // Working through the proxy.
9             proxy.operation());
10    }
11 }

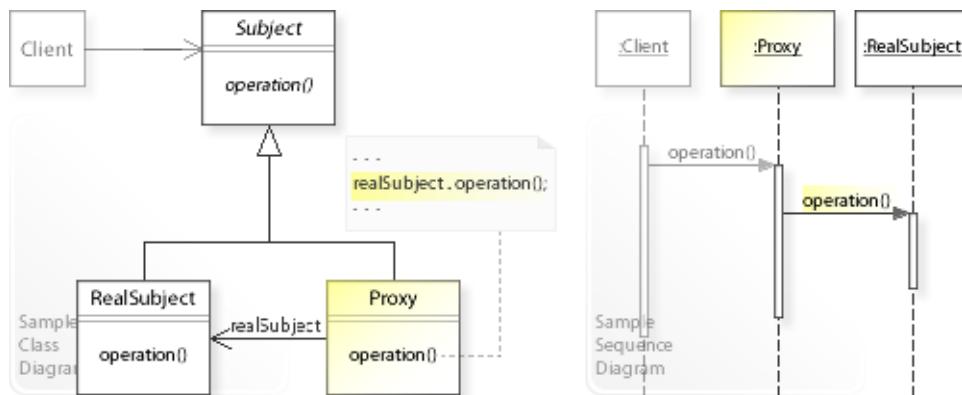
Proxy      : Controlling (permitting) access to real subject ...
RealSubject: Hello World!


1 package com.sample.proxy.basic;
2 public abstract class Subject {
3     public abstract String operation();
4 }

1 package com.sample.proxy.basic;
2 public class RealSubject extends Subject {
3     public String operation() {
4         return "RealSubject: Hello World!";
5     }
6 }

1 package com.sample.proxy.basic;
2 public class Proxy extends Subject {
3     private RealSubject realSubject;
4     public Proxy(RealSubject subject) {
5         this.realSubject = subject;
6     }
7     public String operation() {
8         System.out.println(
9             "Proxy      : Controlling (permitting) access to real subject ...");
10    return realSubject.operation();
11 }
12 }
```

## Related Patterns

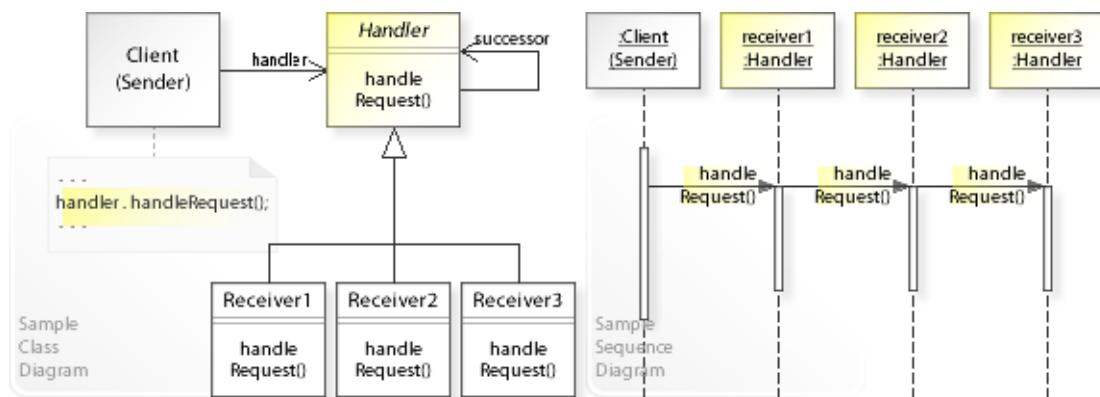


### Key Relationships (see also Overview)

- **Decorator - Proxy**
  - Decorator provides a way to extend the functionality of an object dynamically.
  - Proxy provides a way to restrict, extend, and change the functionality of an object.

### **Part IV. Behavioral Patterns**

## Intent



The intent of the Chain of Responsibility design pattern is to:

**"Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it."** [GoF]

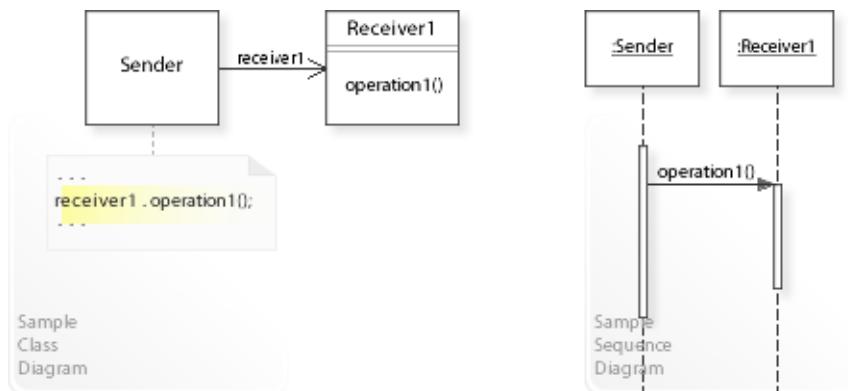
See Problem and Solution sections for a more structured description of the intent.

- The Chain of Responsibility design pattern solves problems like:
  - *How can coupling the sender of a request to its receiver be avoided?*
  - *How can different objects handle a request?*
  - *How can the object that handles a request be determined dynamically?*
- A *request* is an operation that one object (sender) performs on another (receiver). The receiver object performs (handles, carries out) the request (or forwards it to another object).
- The Chain of Responsibility pattern describes how to solve such problems:
  - *Chain the receiving objects and pass the request along the chain until an object handles it.*
  - Define and chain `Handler` objects that either handle a request or forward it to the next handler. This results in a *chain of objects* having the *responsibility* to handle a request.

## Background Information

- Terms and definitions:
  - "An object performs an operation when it receives a corresponding request from an other object. A common synonym for request is **message**." [GoF, p361]
  - A receiver is the target object of a request.
  - A message is "An operation that one object performs on another. The terms *message*, *method*, and *operation* are usually interchangeable." [GBooch07, p597]
  - *Coupling* is "The degree to which software components depend on each other." [GoF, p360]
- An *UML sequence diagram* shows the objects of interest and the requests (messages) between them. Requests are drawn horizontally from sender to receiver and their ordering is indicated by their vertical position. That means, the first request is shown at the top and the last at the bottom.

## Problem



The Chain of Responsibility design pattern solves problems like:

**How can coupling the sender of a request to its receiver be avoided?**

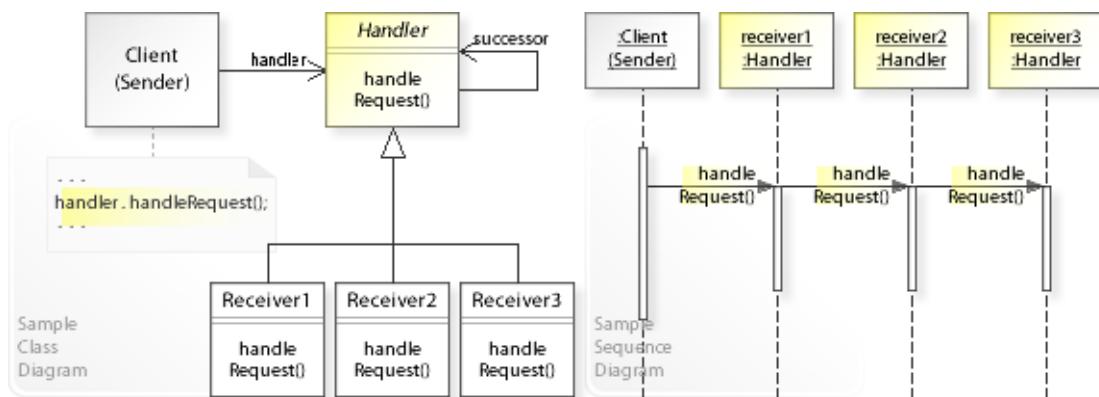
**How can different objects handle a request?**

**How can the object that handles a request be determined dynamically?**

See Applicability section for all problems Chain of Responsibility can solve.

- The standard way to specify a request is to hard-wire a particular receiver and operation (`receiver1.operation1()`) directly within a class (`Sender`).
- This commits (couples) the class (the sender of the request) to a particular receiver (that handles the request), which makes it hard to change the receiver later independently from (without having to change) the class.
- *That's the kind of approach to avoid if we want that the sender of a request isn't coupled to a particular receiver.*  
In a context-sensitive help system, a user can click anywhere to get help information. The provided help depends on the corresponding object and context. If no specific help information is available, the help system should provide the next more general help information.  
"The problem here is that the object [receiver] that ultimately *provides* the help isn't known explicitly to the object [sender] (e.g., the button) that *initiates* the help request." [GoF, p223]

## Solution



The Chain of Responsibility pattern describes a solution:

**Define and chain Handler objects that either handle or forward a request.**

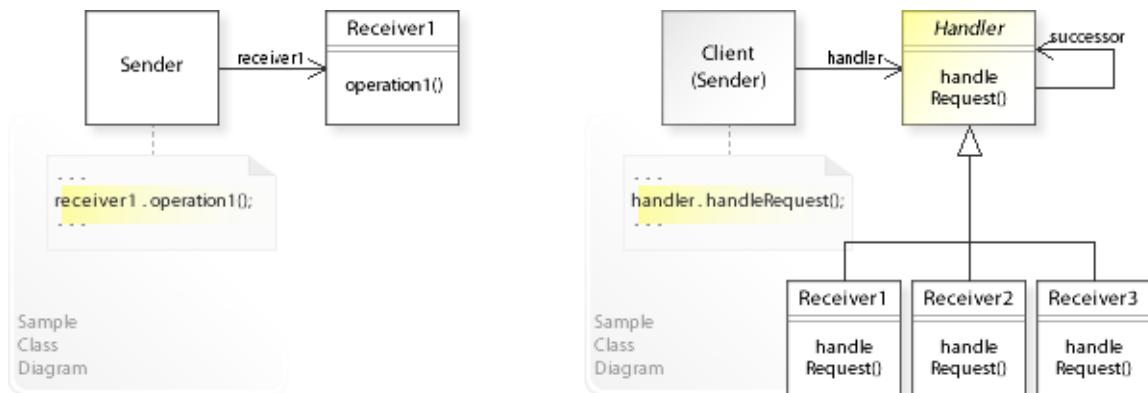
**Send a request to a Handler object on the chain.**

Describing Chain of Responsibility in more detail is the theme of the following sections.

See Applicability section for all problems Chain of Responsibility can solve.

- "The idea of this pattern is to decouple senders and receivers by giving multiple objects a chance to handle a request. The request gets passed along a chain of objects until one of them handles it." [GoF, p223]
- **Define and chain Handler objects:**
  - Define an interface for handling a request (`Handler | handleRequest()`).
  - Objects that can handle a request implement the `Handler` interface by either handling the request directly or forwarding it to the next handler (if any) on the chain.
  - "Chain of Responsibility is a good way to decouple the sender and the receiver if the chain is already part of the system's structure, and one of several objects may be in a position to handle the request." [GoF, p348]
- **Send a request to a Handler object on the chain** (`handler.handleRequest()`).  
The request is forwarded along the chain until a handler handles it.
- This enables loose coupling between the sender of a request and its receiver.  
The object that sends a request has no explicit knowledge of the object (receiver) that ultimately will handle the request.  
The chain of `Handler` objects can be specified dynamically at run-time (`Handler` objects can be added to and removed from the chain).

## Motivation 1



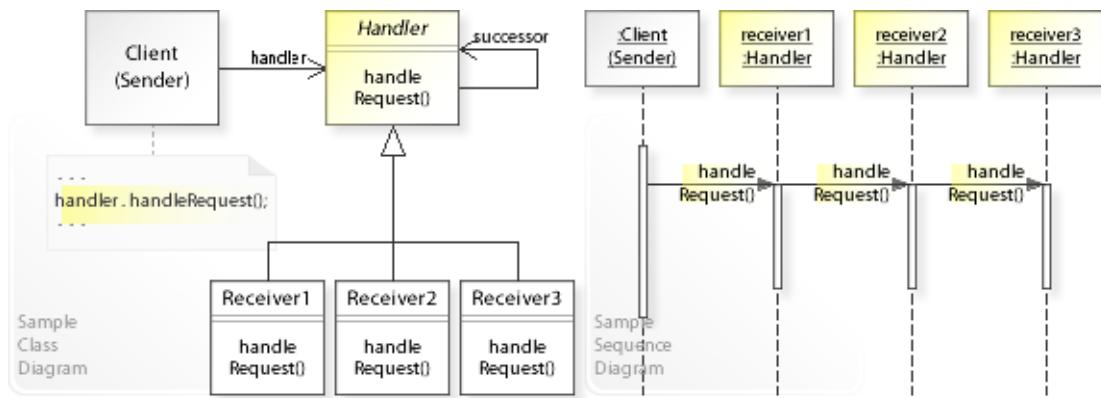
**Consider the left design (problem):**

- Tight coupling between sender and receiver.
  - The request is sent to a particular `Receiver1` object.
  - This couples the sender of a request to a particular receiver.

**Consider the right design (solution):**

- Loose coupling between sender and receiver.
  - The request is sent to a `Handler` object on the chain of handlers.
  - This decouples the sender of a request from a particular receiver.
  - The sender has no explicit knowledge of the receiver (handler) that ultimately will handle the request.
  - The chain of handlers can be specified dynamically at run-time (handlers can be added to and removed from the chain).

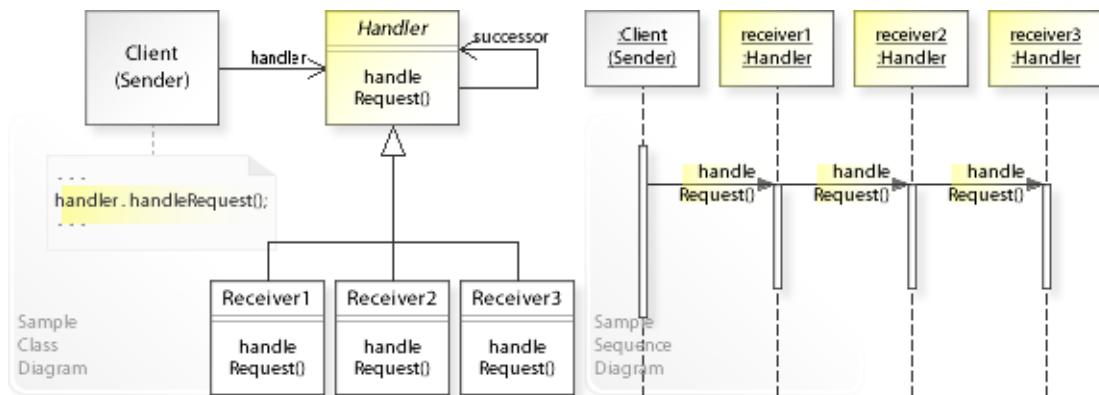
## Applicability



## Design Problems

- **Avoiding Sender-Receiver Coupling**
  - How can coupling the sender of a request to its receiver be avoided?
- **Supporting Different Handlers**
  - How can different objects handle a request?
  - How can the object that handles a request be determined dynamically?

## Structure, Collaboration



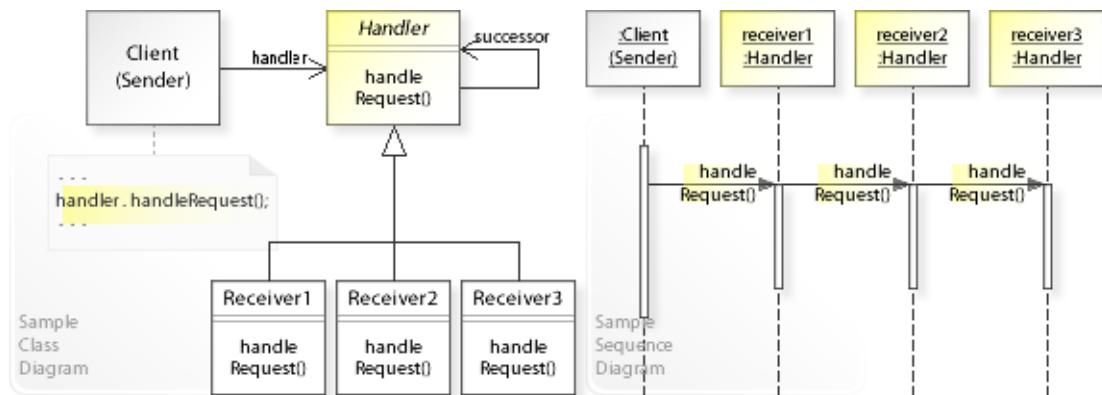
### Static Class Structure

- **Client (Sender)**
  - Refers to the **Handler** interface to handle a request (`handler.handleRequest()`) and is independent of how the request is handled (which handler handles the request).
  - Maintains a reference (`handler`) to a **Handler** object.
- **Handler**
  - Defines an interface for handling a request.
  - Maintains a reference (`successor`) to the next **Handler** object on the chain.
- **Receiver1, Receiver2, Receiver3, ...**
  - Implement the **Handler** interface by either handling a request directly or forwarding it to the next handler (if any) on the chain.

### Dynamic Object Collaboration

- In this sample scenario, a **Client (Sender)** object issues a request to a **Handler** object on the chain. The request is forwarded along the chain until a handler (**receiver3**) handles it.
- The interaction starts with the **Client** that calls `handleRequest()` on the **receiver1** object (of type **Handler**).
- **receiver1** forwards the request by calling `handleRequest()` on the **receiver2** object.
- **receiver2** forwards the request by calling `handleRequest()` on the **receiver3** object.
- **receiver3** handles the request and returns to **receiver2** (which returns to **receiver1**, which in turn returns to the **Client**).
- See also Sample Code / Example 1.

## Consequences



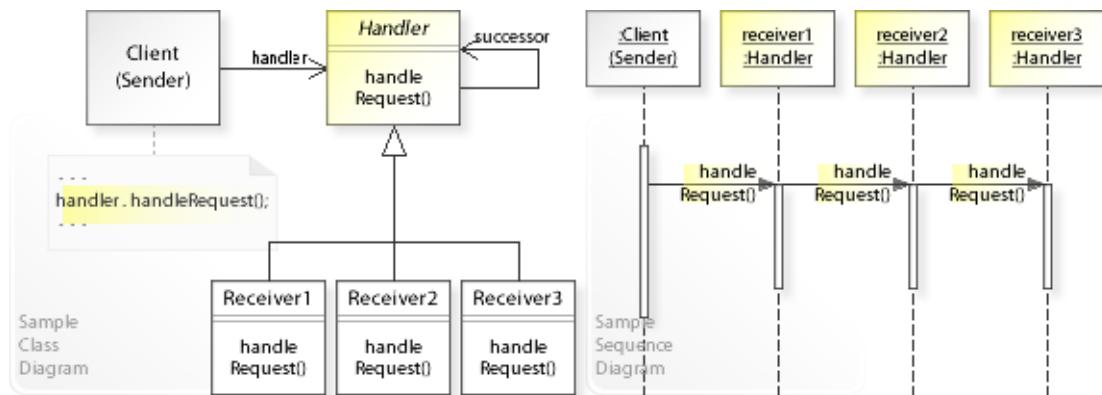
### Advantages (+)

- Decouples sender from receiver.
  - The pattern decouples the sender of a request from a particular receiver (handler) by passing the request along a chain of receivers.
- Makes changing handlers easy.
  - Handlers can be added to and removed from the chain dynamically at run-time.

### Disadvantages (-)

- Successor chain may be hard to implement.
  - If there is no existing object structure that can be used to define and maintain the successor chain, it may be hard to implement and maintain the successor chain (see Implementation).
  - "Chain of Responsibility is a good way to decouple the sender and the receiver if the chain is already part of the system's structure, and one of several objects may be in a position to handle the request." [GoF, p348]

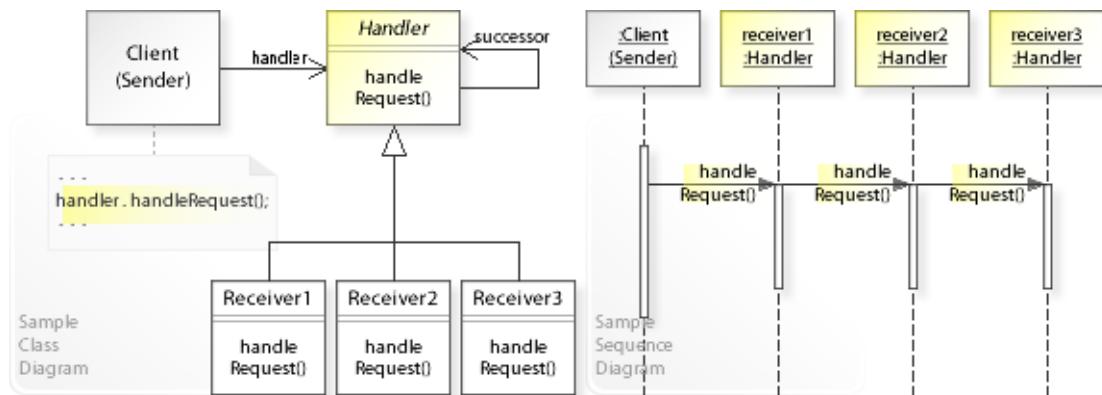
## Implementation



### Implementation Issues

- **Implementing the successor chain.**
  - There are two main variants to implement the successor chain:
- **Variant1: Defining new links.**
  - If there are no existing object structures that can be used to define the successor chain, a new chain must be defined.
- **Variant2: Using existing links.**
  - "Chain of Responsibility is a good way to decouple the sender and the receiver if the chain is already part of the system's structure, and one of several objects may be in a position to handle the request." [GoF, p348]
  - Often existing composite object structures can be used to define the successor chain (see Composite).

## Sample Code 1



### Basic Java code for implementing the sample UML diagrams.

```

1 package com.sample.cor.basic;
2 public class Client {
3     // Running the Client class as application.
4     public static void main(String[] args) {
5         // Creating the chain of handler objects.
6         Handler handler = new Receiver1(new Receiver2(new Receiver3()));
7         //
8         System.out.println("Client : Issuing a request to a handler object ... ");
9         handler.handleRequest();
10    }
11 }

Client : Issuing a request to a handler object ...
Receiver1: Passing the request along the chain ...
Receiver2: Passing the request along the chain ...
Receiver3: Handling the request.

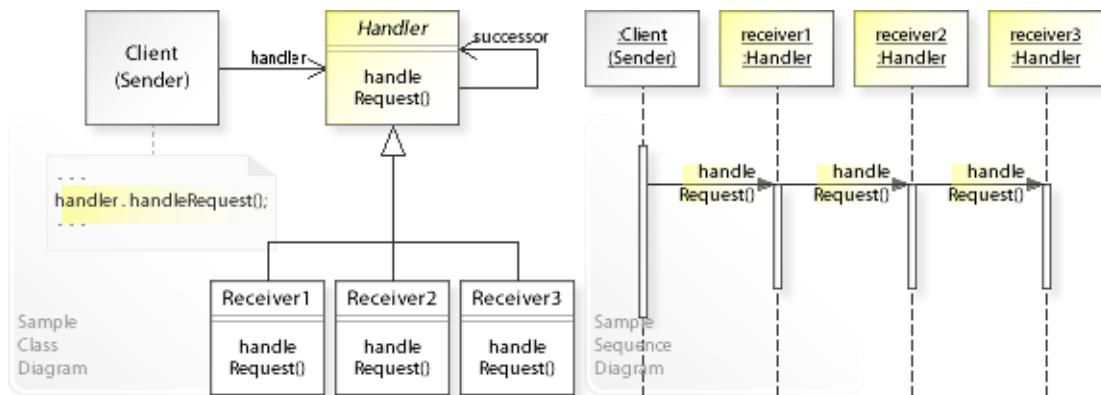
1 package com.sample.cor.basic;
2 public abstract class Handler {
3     private Handler successor;
4     public Handler() { }
5     public Handler(Handler successor) {
6         this.successor = successor;
7     }
8     public void handleRequest() {
9         // Forwarding to successor (if any).
10        if (successor != null) {
11            successor.handleRequest();
12        }
13    }
14    public boolean canHandleRequest() {
15        // Checking run-time conditions ...
16        return false;
17    }
18 }

1 package com.sample.cor.basic;
2 public class Receiver1 extends Handler {
3     public Receiver1(Handler successor) {
4         super(successor);
5     }
6     @Override
7     public void handleRequest() {
8         if (canHandleRequest()) {
9             System.out.println("Receiver1: Handling the request ...");
10        } else {
11            System.out.println("Receiver1: Passing the request along the chain ...");
12            super.handleRequest();
13        }
14    }
15 }
```

```
1 package com.sample.cor.basic;
2 public class Receiver2 extends Handler {
3     public Receiver2(Handler successor) {
4         super(successor);
5     }
6     @Override
7     public void handleRequest() {
8         if (canHandleRequest()) {
9             System.out.println("Receiver2: Handling the request ...");
10        } else {
11            System.out.println("Receiver2: Passing the request along the chain ...");
12            super.handleRequest();
13        }
14    }
15 }

1 package com.sample.cor.basic;
2 // End of chain.
3 public class Receiver3 extends Handler {
4     @Override
5     public void handleRequest() {
6         // Must handle the request unconditionally.
7         System.out.println("Receiver3: Handling the request.");
8     }
9 }
```

## Related Patterns

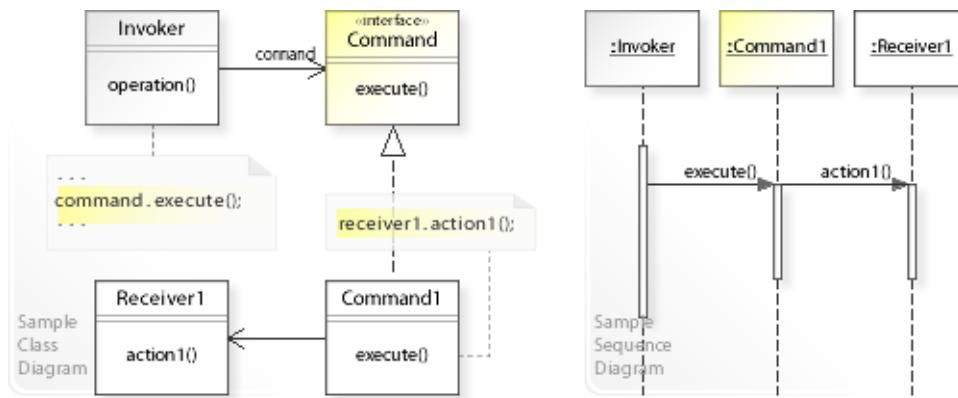


### Key Relationships (see also Overview)

- **Composite - Chain of Responsibility**

- Composite and Chain of Responsibility often work together.  
Existing composite object structures can be used to define the successor chain.  
"Chain of Responsibility is a good way to decouple the sender and the receiver if the chain is already part of the system's structure, and one of several objects may be in a position to handle the request." [GoF, p348]

## Intent



The intent of the Command design pattern is to:

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

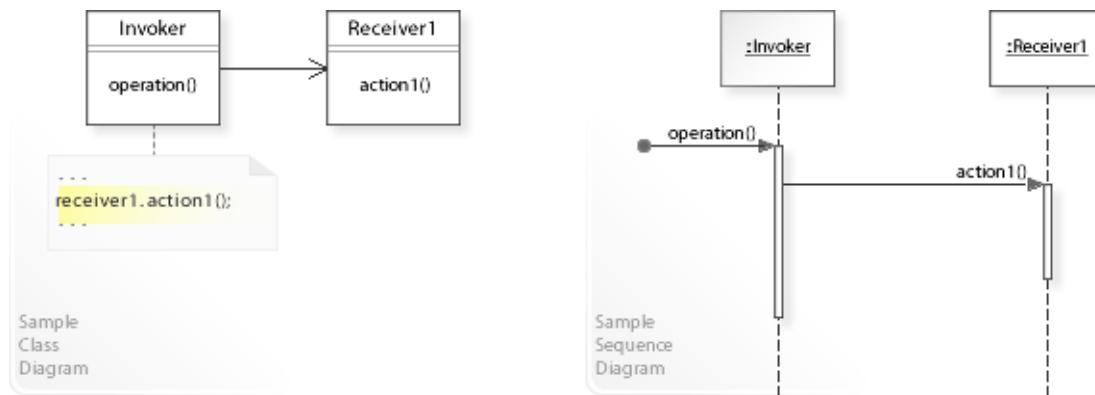
See Problem and Solution sections for a more structured description of the intent.

- The Command design pattern solves problems like:
  - *How can an object be configured with a request?*
  - *How can requests be queued or logged?*
  - *How can undoable operations be supported?*
- A *request* is an operation that one object performs on another.  
From a more general point of view, a request is an arbitrary *action to perform*.  
The terms *request*, *message*, *operation*, and *method* are usually interchangeable.  
The terms *performing*, *issuing*, and *sending* a request are usually interchangeable.
- The Command pattern describes how to solve such problems:
  - *Encapsulate a request as an object* -  
define separate classes (`Command1, ...`)  
that implement (encapsulate) different requests, and  
define a common interface (`Command | execute()`)  
through which a request can be executed.

## Background Information

- Terms and definitions:
  - "An object performs an operation when it receives a corresponding request from an other object. A common synonym for request is **message**." [GoF, p361]
  - A receiver is the target object of a request.
  - A message is "An operation that one object performs on another. The terms *message*, *method*, and *operation* are usually interchangeable." [GBooch07, p597]
- An *UML sequence diagram*  
shows the objects of interest and the requests (messages) between them.  
Requests are drawn horizontally from sender to receiver and their ordering is indicated by their vertical position. That means, the first request is shown at the top and the last at the bottom.

## Problem



The Command design pattern solves problems like:

***How can an object be configured with a request?***

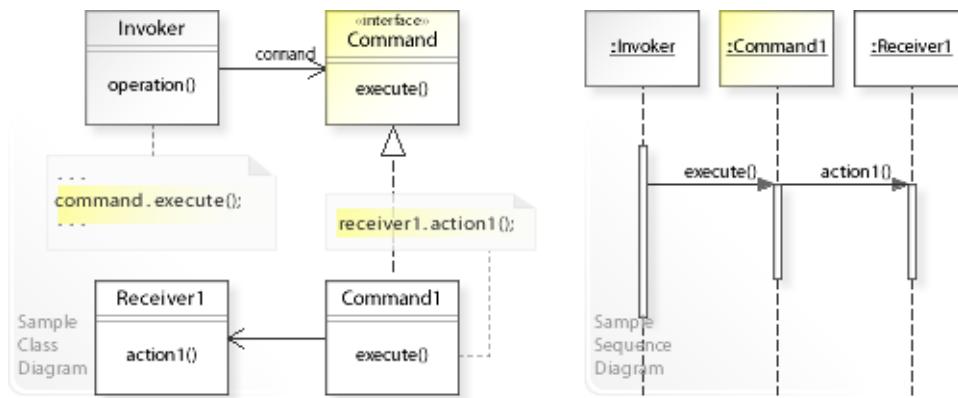
***How can requests be queued or logged?***

***How can undoable operations be supported?***

See Applicability section for all problems Command can solve.

- The standard way to specify a request is to hard-wire a particular receiver and operation (`receiver1.action1()`) directly within a class (`Invoker`).
- This commits the class to a particular request, which makes it hard to change the request later independently from (without having to change) the class.  
"When you specify a particular operation, you commit to one way of satisfying a request. By avoiding hard-coded requests, you make it easier to change the way a request gets satisfied both at compile-time and run-time." [GoF, p24]
- *That's the kind of approach to avoid if we want to configure an object with the actually needed request at run-time.*
- For example, reusable GUI/Web objects (like buttons or menus) that perform a request in response to an user input.  
When implementing objects that are reused by many different applications, we do not know the receiver of the request or the operation to be performed. The applications that reuse the object should configure it with the needed request at run-time.

## Solution



The Command pattern describes a solution:

**Define separate Command objects that encapsulate different requests.**

**Delegate a request to a Command object.**

Describing the Command design in more detail is the theme of the following sections.

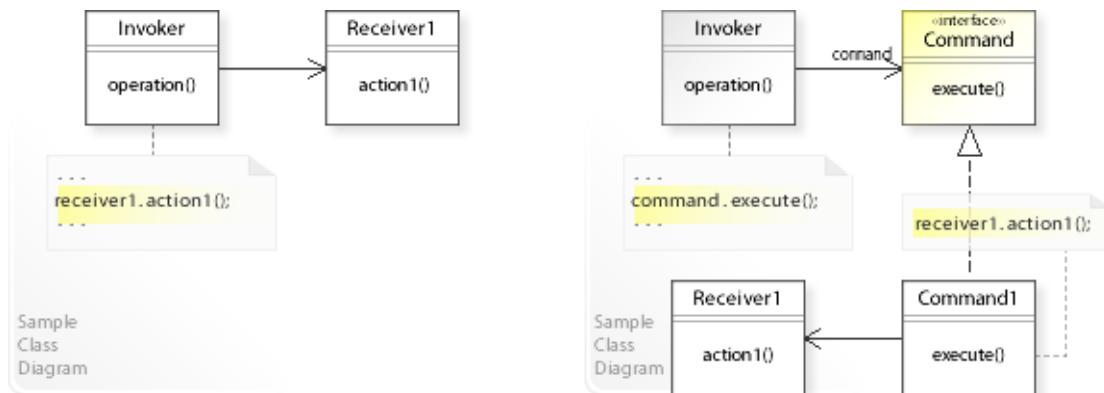
See Applicability section for all problems Command can solve.

- The key idea in this pattern is to encapsulate a request in a separate object. The request can be stored somewhere (in a history list, for example) to be called at a later point, and it can be passed around like other objects.
- **Define separate Command objects:**
  - Define a common interface for performing a request (`Command | execute()`).
  - Define classes (`Command1, ...`) that implement the `Command` interface.
  - Implementing the `Command` interface can involve calling an operation on a receiver, calling many operations on many receivers, performing arbitrary actions, or something in between.
- This enables *compile-time* flexibility (via inheritance). New requests can be added and existing ones can be changed independently by defining new (sub)classes.
- **Delegate (the responsibility for performing) a request to a Command object** (`command.execute()`).
  - This enables *run-time* flexibility (via object composition). An object can be configured (customized) with the needed `Command` object, and even more, the `Command` object can be exchanged dynamically at run-time. Commands can be stored (in a history list, for example) so that they can be executed and unexecuted at different times (to queue or log requests and to support undoable operations).

## Background Information

- In a procedural language, a *callback* function is "a function that's registered somewhere to be called at a later point. Commands are an object-oriented replacement for callbacks." [GoF, p235]

## Motivation 1



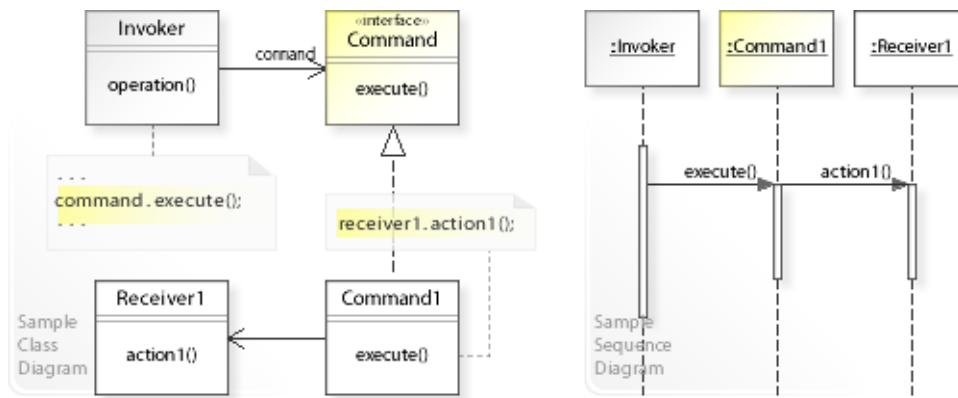
### Consider the left design (problem):

- Hard-wired request.
  - The request (`receiver.action()`) is implemented (hard-wired) directly within a class (Client/Sender).
  - This makes it hard to change the request later independently from (without having to change) the class.
- The request must be specified at compile-time.
  - When designing reusable objects, often the particular request isn't known (and can't be specified) at compile-time and should be specified at run-time.

### Consider the right design (solution):

- Encapsulated request.
  - Each request is implemented (encapsulated) in a separate class (`Command1, Command2, ...`).
  - This makes it easy to change the request later independently from (without having to change) the client class.
- The request can be specified at run-time.
  - The client delegates a request to a Command object, which can be specified at run-time.

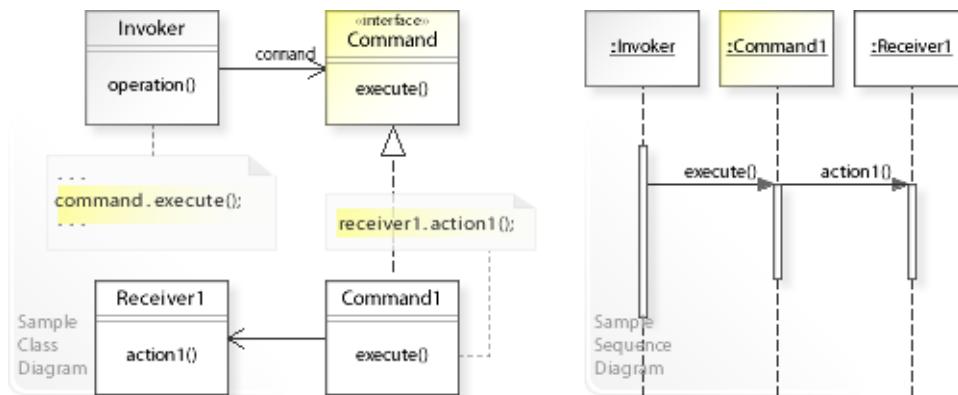
## Applicability



## Design Problems

- **Configuring Objects With Different Requests**
  - How can an object be configured with a request (action to perform)?
- **Storing and Passing Around Requests**
  - How can requests be queued or logged?
  - How can undoable operations be supported?

## Structure, Collaboration



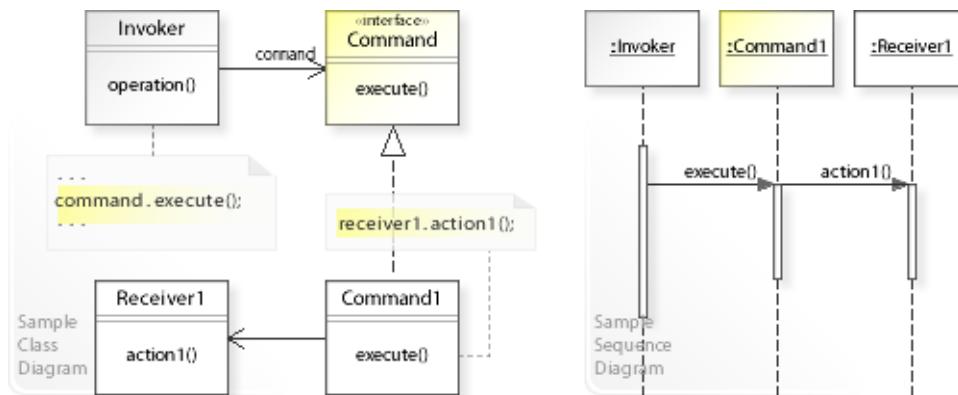
### Static Class Structure

- **Invoker**
  - Refers to the **Command** interface to perform a request (`command.execute()`) and is independent of how the request is performed.
  - Maintains a reference (`command`) to a **Command** object.
- **Command**
  - Defines a common interface for performing a request.
- **Command1**
  - Implements the **Command** interface (by calling `action1()` on a **Receiver1** object).
  - See also Implementation.

### Dynamic Object Collaboration

- In this sample scenario, an **Invoker** object delegates performing a request to a **Command** object. Let's assume that **Invoker** is configured with a **Command1** object.
- The interaction starts with a **Client** object that calls `operation` on the **Invoker** object.
- While performing the operation, **Invoker** calls `execute()` on the installed **Command1** object.
- **Command1** calls `action1()` on a **Receiver1** object.
- See also Sample Code / Example 1.

## Consequences



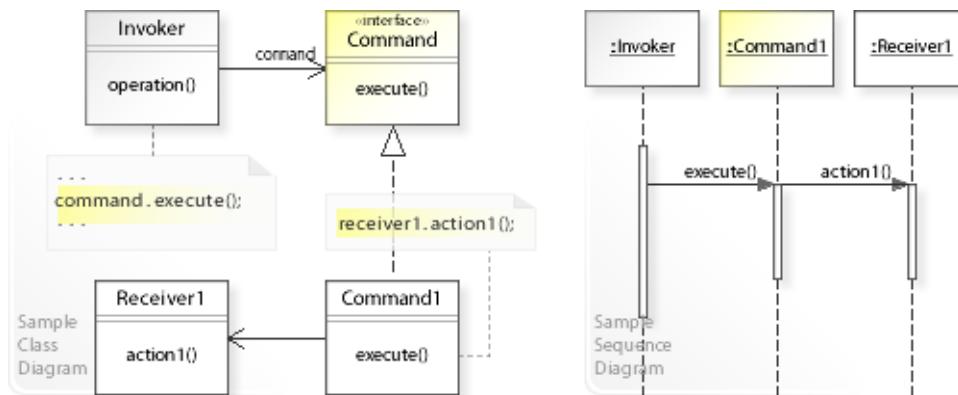
### Advantages (+)

- Makes adding new commands easy.
  - "It's easy to add new Commands, because you don't have to change existing classes." [GoF, p237]
- Makes exchanging commands easy.
  - Clients can be configured with the needed Command object, and even more, the command can be exchanged dynamically at run-time (pluggable commands).

### Disadvantages (-)

- Additional level of indirection.
  - Command achieves flexibility by introducing an additional level of indirection (delegating to separate Command objects).

## Implementation

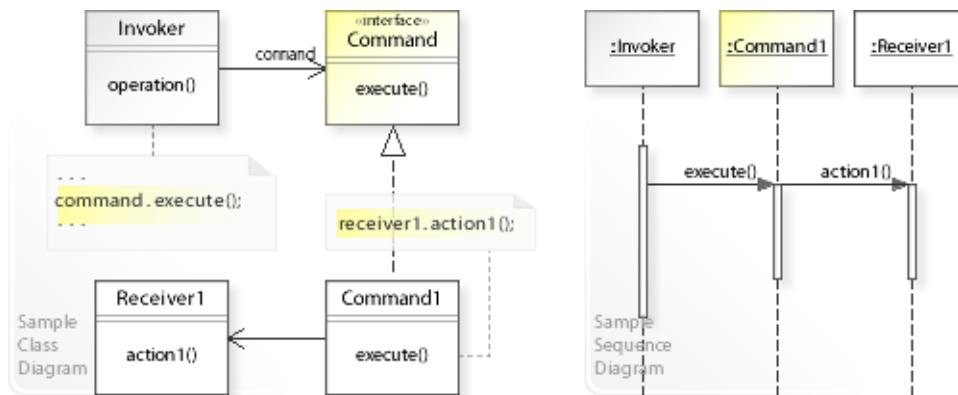


### Implementation Issues

- **Implementation Variants**

- A command can implement arbitrary functionality.  
Usually, a command implements a request by calling an operation on a receiver object (`receiver.action()`).
- "At one extreme it merely defines a binding between a receiver and the actions that carry out the request. At the other extreme it implements everything without delegating to a receiver at all. [...] Somewhere in between these extremes are commands[...]" [GoF, p238]

## Sample Code 1



### Basic Java code for implementing the sample UML diagrams.

```

1 package com.sample.command.basic;
2 public class Client {
3     // Running the Client class as application.
4     public static void main(String[] args) {
5         // Creating an invoker object and configuring it with
6         // the needed command.
7         Invoker invoker = new Invoker(new Command1(new Receiver1()));
8         //
9         invoker.operation();
10    }
11 }

Invoker : Calling execute on the installed command ...
Command1 : Executing (carrying out) the request ...
Receiver1: Hello World1!

1 package com.sample.command.basic;
2 public class Invoker {
3     private Command command;
4     public Invoker(Command command) {
5         // Configuring with a command.
6         this.command = command;
7     }
8     public void operation() {
9         System.out.println("Invoker : Calling execute on the installed command ... ");
10        command.execute();
11    }
12 }

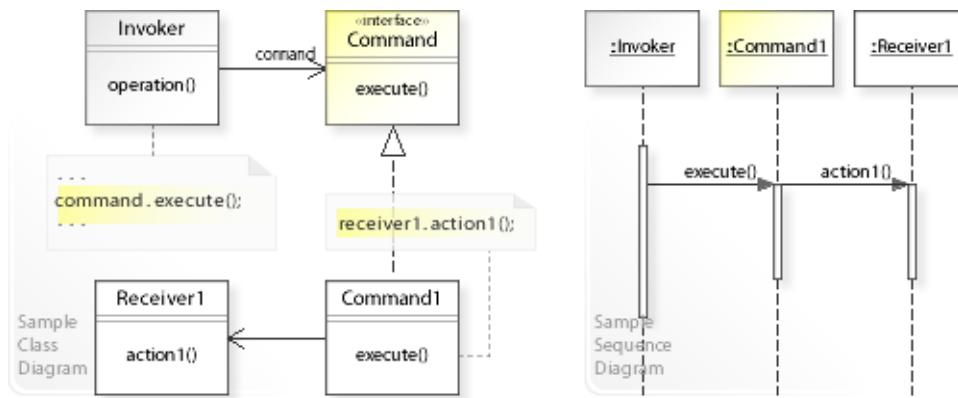
1 package com.sample.command.basic;
2 public interface Command {
3     void execute();
4 }

1 package com.sample.command.basic;
2 public class Command1 implements Command {
3     private Receiver1 receiver1;
4     public Command1(Receiver1 receiver1) {
5         this.receiver1 = receiver1;
6     }
7     public void execute() {
8         System.out.println("Command1 : Executing (carrying out) the request ...");
9         receiver1.action1();
10    }
11 }

1 package com.sample.command.basic;
2 public class Receiver1 {
3     public void action1() {
4         System.out.println("Receiver1: Hello World1!");
5     }
6 }

```

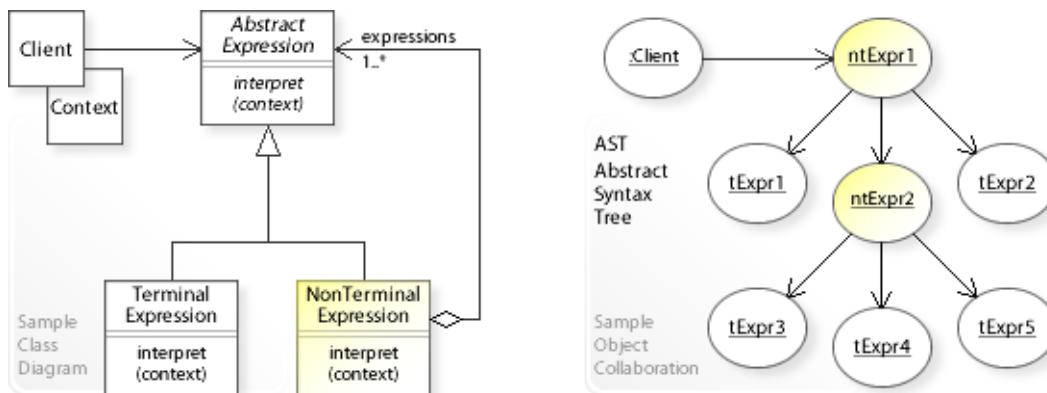
## Related Patterns



### Key Relationships (see also Overview)

- **Strategy - Command**
  - Strategy provides a way to configure an object with an algorithm to perform.
  - Command provides a way to configure an object with an action to perform.
- **Command - Memento**
  - Command and Memento often work together to support undoable operations.  
Memento stores state that command requires to undo its effects.

## Intent



The intent of the Interpreter design pattern is:

**"Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language."** [GoF]

See Problem and Solution sections for a more structured description of the intent.

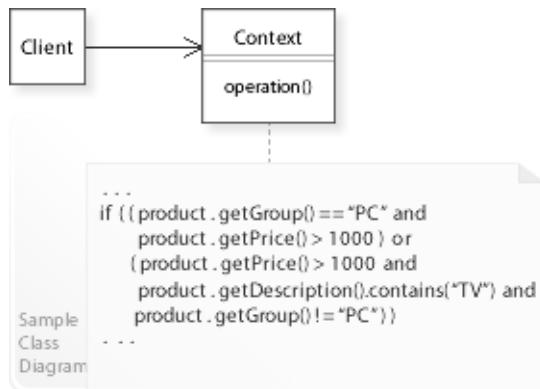
- The Interpreter design pattern solves problems like:
  - *How can a grammar for a simple language be defined so that sentences in the language can be interpreted?*
- Terms and definitions:
  - A *language* is a set of valid sentences.
  - A *sentence* = *statement* = *expression*.  
[...] expressions are the core components of statements [...] An *expression* is a construct made up of variables, operators, and method invocations, which are constructed according to the syntax of the language, that evaluates to a single value." [The Java Language]
  - A *grammar* is a way of formally describing the structure (syntax) of a language.  
It's a list of rules, which Interpreter uses to interpret sentences in the language.  
The most common grammar notation is Extended Backus-Naur Form (EBNF).
- The Interpreter pattern describes how to solve such problems:
  - *Given a language, define a representation for its grammar - by defining an Expression class hierarchy*
  - *along with an interpreter that uses the representation to interpret sentences in the language - every sentence in the language is represented by an abstract syntax tree (AST) made up of instances of the Expression classes.*

A sentence is interpreted by calling `interpret(context)` on its AST.

## Background Information

- Domain-specific languages (DSL) [MFowler11]  
are designed to solve problems in a particular domain. In contrast, general-purpose languages are designed to solve problems in many domains.  
DSLs are widely used, for example, ANT, CSS, regular expressions, SQL, HQL (Hibernate Query Language), XML, framework configuration files, XSLT, etc.

## Problem



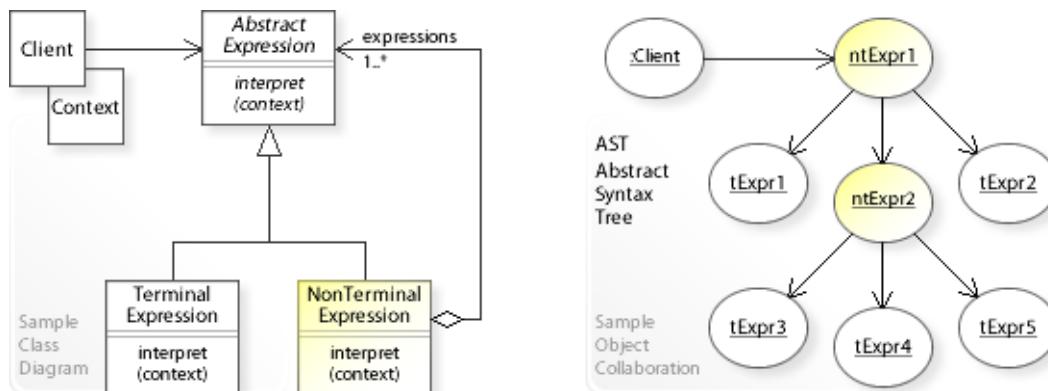
The Interpreter design pattern solves problems like:

***How can a grammar for a simple language be defined so that sentences in the language can be interpreted?***

See Applicability section for all problems Interpreter can solve.

- "If a particular kind of problem occurs often enough, then it might be worthwhile to express instances of the problem as sentences in a simple language. Then you can build an interpreter that solves the problem by interpreting these sentences." [GoF, p243]
- For example, specifying a search expression is a problem that often occurs. The standard way to specify a search expression is to implement (hard-code) the expression each time it is needed directly within a client class (`Context`). This commits to a particular expression, which makes it hard to change the expression later independently from (without having to change) the class.
- *That's the kind of approach to avoid if we want to specify and interpret (evaluate) search expressions dynamically.*
- For example, an object finder with arbitrary (dynamically changeable) search criteria. Instead of hard-coding a search expression each time it is needed, it should be possible to define a simple query language so that search expressions can be specified and interpreted (evaluated) dynamically (see Implementation and Sample Code / Example 2).

## Solution



The Interpreter pattern describes a solution:

- (1) **Define a grammar for a simple language by an expression class hierarchy.**
- (2) **Represent a sentence in the language by an AST (abstract syntax tree).**
- (3) **Interpret a sentence by calling `interpret(context)` on an AST.**

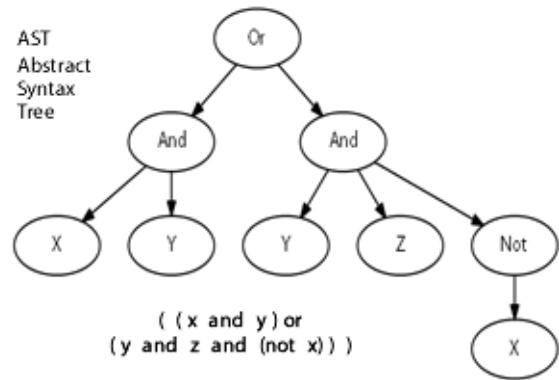
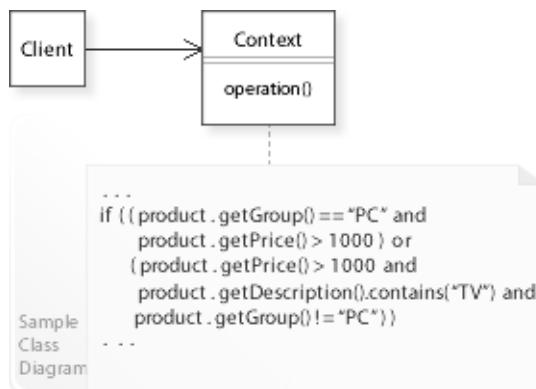
Describing the Interpreter design in more detail is the theme of the following sections. See Applicability section for all problems Interpreter can solve.

- (1) **Define a grammar for a simple language by an expression class hierarchy:**
  - Define an interface for interpreting an expression (`AbstractExpression | interpret(context)`).
  - Define classes (`TerminalExpression`) that implement the interpretation.  
" [...] many kinds of operations can "interpret" a sentence." [GoF, p254]  
Usually, an interpreter is considered to interpret (evaluate) an expression and return a simple result, but any kind of operation can be implemented.
  - Define classes (`NonTerminalExpression`) that forward interpretation to their child expressions.
- (2) **Represent a sentence in the language by an AST (abstract syntax tree):**
  - Every sentence in the language is represented by an abstract syntax tree made up of `TerminalExpression` instances (`tExpr1, tExpr2, ...`) and `NonTerminalExpression` instances (`ntExpr1, ntExpr2, ...`).
  - The expression objects are composed recursively into a composite/tree structure that is called *abstract syntax tree* (see Composite pattern).  
Terminal expressions have no child expressions and perform interpretation directly.  
Nonterminal expressions forward interpretation to their child expressions.  
"The Interpreter pattern doesn't explain how to *create* an abstract syntax tree. In other words, it doesn't address parsing. The abstract syntax tree can be created by a [...] parser, or directly by the client." [GoF, p247]
- (3) **Interpret a sentence by calling `interpret(context)` on an AST.**
  - See also Implementation and Sample Code.

## Background Information

- A Parser Generator
  - uses a grammar file to generate a parser. The parser can be updated simply by updating the grammar and regenerating. The generated parser can use efficient techniques to create ASTs that would be hard to build and maintain by hand.
- ANTLR (ANother Tool for Language Recognition) [TParr07]
  - is a open source parser generator for reading, processing, executing, or translating structured text or binary files. It's widely used to build languages, tools, and frameworks. From a grammar, ANTLR generates a parser that can build and walk parse trees.

## Motivation 1



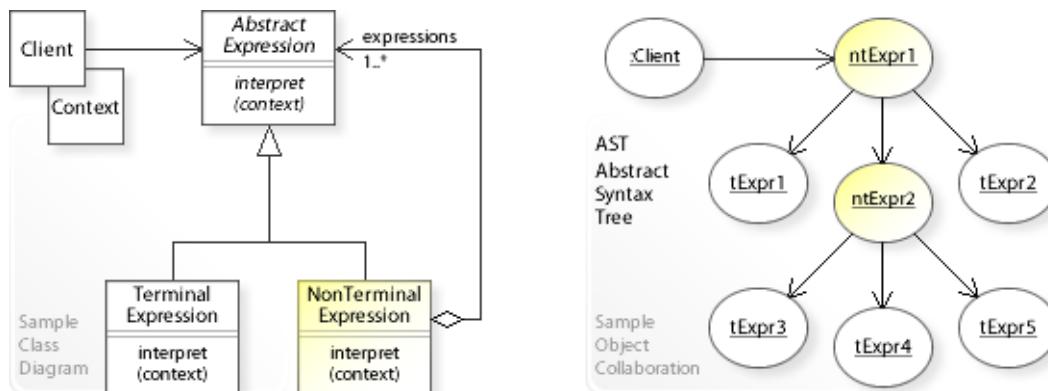
**Consider the left design (problem):**

- Hard-wired expression.
  - The search expression is hard-wired directly within a class (`Context`).
  - This makes it hard to specify new expressions or change existing ones both at compile-time and at run-time.

**Consider the right design (solution):**

- Expression represented by an AST.
  - The search expression is represented by an AST (abstract syntax tree).
  - This makes it easy to create new expressions (ASTs) or change existing ones dynamically at run-time (by a parser).
  - A *Parser Generator* uses a grammar file to generate a parser. The generated parser can then create new ASTs efficiently.

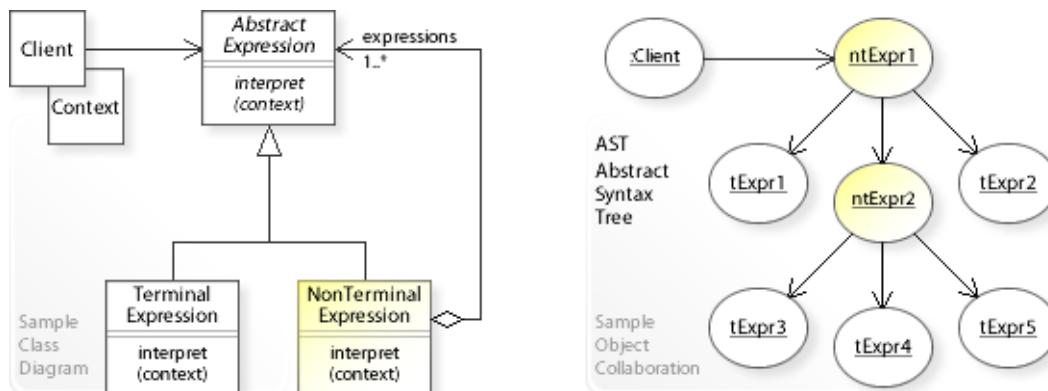
## Applicability



## Design Problems

- **Domain Specific Languages / Interpreting Statements**
  - How can a grammar for a simple language be defined so that sentences in the language can be interpreted?
  - How can instances of a problem be represented as sentences in a simple language so that these sentences can be interpreted to solve the problem?

## Structure, Collaboration



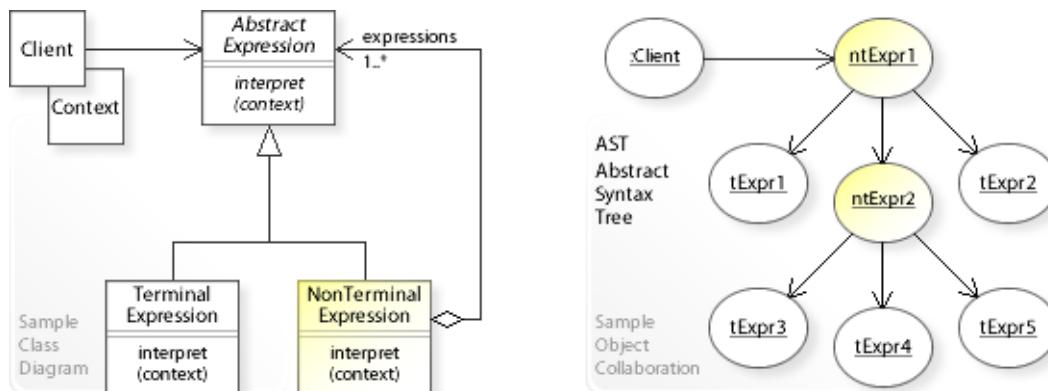
## Static Class Structure

- `AbstractExpression`
  - Defines a common interface for interpreting terminal expressions (`tExpr`) and nonterminal expressions (`ntExpr`).
- `TerminalExpression`
  - Implements interpretation for terminal expressions.
  - A terminal expression has no child expressions.
- `NonTerminalExpression`
  - Maintains a container of child expressions (`expressions`).
  - Forwards interpretation to its child expressions.
  - A nonterminal expression is a composite expression and has child expressions (terminal and nonterminal expressions). See also Composite pattern.

## Dynamic Object Collaboration

- Let's assume that a `Client` object builds an abstract syntax tree (AST) to represent a sentence in the language.
- The `Client` calls `interpret(context)` on the AST.
- The nonterminal expression nodes of the AST (`ntExpr1, ntExpr2`) call `interpret(context)` on their child expression nodes.
- The terminal expression nodes (`tExpr1, tExpr2, ...`) perform interpretation directly.
- See also Sample Code / Example 1.

## Consequences



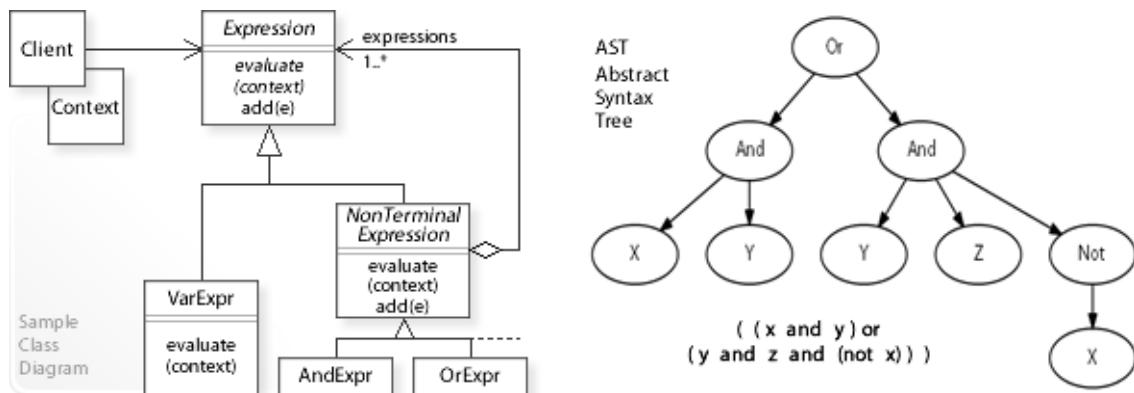
### Advantages (+)

- Makes changing the grammar easy.
  - The Interpreter pattern uses a class hierarchy to represent grammar rules.
  - Clients refer to the `AbstractExpression` interface and are independent of its implementation.
  - Clients do not have to change when new terminal or nonterminal expression classes are added.
- Makes adding new kinds of interpret operations easier.
  - "[...] many kinds of operations can "interpret" a sentence." [GoF, p254]
  - Usually, an interpreter is considered to interpret an expression and return a simple result, but any kind of operation can be performed.
  - The `Visitor` pattern can be used to define new kinds of interpret operations without having to change the existing expression class hierarchy.

### Disadvantages (-)

- Makes maintaining complex grammars hard.
  - The Interpreter pattern uses (at least) one class to represent each grammar rule.
  - Therefore, for complex grammars, the class hierarchy becomes large and hard to maintain.
  - Parser generators are an alternative in such cases. They can represent complex grammars without building complex class hierarchies.

## Implementation



### Implementation Issues

- Let's assume, we want to define a grammar (syntax) for a simple query language so that sentences (search expressions) in the language can be specified and interpreted (evaluated) dynamically. For example, search expressions should look like:

$(x \text{ and } y) \text{ or } (y \text{ and } z \text{ and } (\text{not } x))$

where  $x, y, z$  are terminal expressions (**VarExpr**) for arbitrary search criteria.

For example, searching product objects:

$(\text{group} == "PC" \text{ and } \text{prize} > 1000) \text{ or } (\text{prize} > 1000 \text{ and } \text{description containing "TV"} \text{ and } (\text{group is not } "PC"))$ .

See the above diagrams and Sample Code / Example 2.

- (1) Define a grammar for a simple query language:**

- Grammar rules (in EBNF notation) would look like:

```

expression : andExp | orExp | notExp | varExp | '(' expression ')';
andExp : expression 'and' expression;
orExp : expression 'or' expression;
notExp : 'not' expression;
varExp : 'x' | 'y' | 'z';
    
```

- andExp, orExp, notExp are nonterminal expression rules.

- varExp x,y,z are terminal expression rules for arbitrary search criteria that evaluate to true or false.

For example, searching product objects:

```

setVarExp(x, product.getGroup() == "PC" ? true : false);
setVarExp(y, product.getPrice() > 1000 ? true : false);
setVarExp(z, product.getDescription().contains("TV") ? true : false);
    
```

- (2) Represent a sentence (search expression) in the language by an AST:**

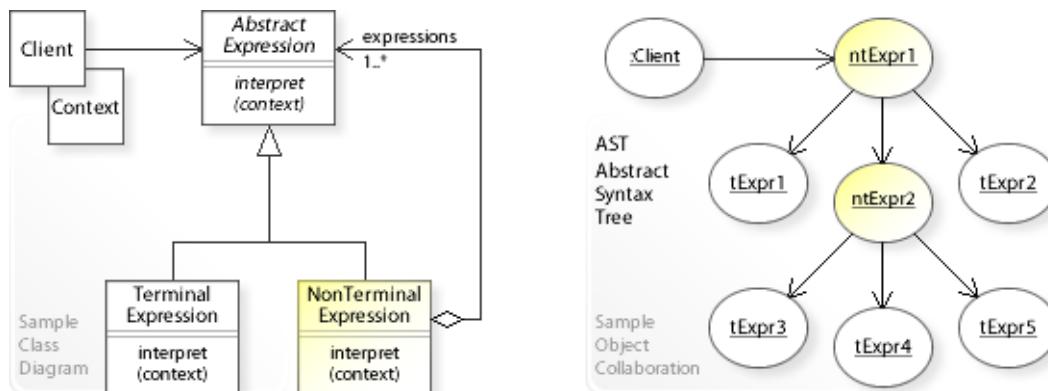
- Every sentence in the language is represented by an abstract syntax tree (AST) made up of instances of the **Expression** classes.

"The Interpreter pattern doesn't explain how to *create* an abstract syntax tree. In other words, it doesn't address parsing. The abstract syntax tree can be created by a [...] parser, or directly by the client." [GoF, p247]

- (3) Interpret a sentence (evaluate a search expression):**

- Clients call **evaluate(context)** on an AST.
- The interpret (evaluate) operation on each terminal expression node uses the context to store and access the state of the interpretation.
- The interpret (evaluate) operation on each nonterminal expression node forwards interpretation to its child expression nodes. See also Sample Code / Example 2.

## Sample Code 1



Basic Java code for implementing the sample UML diagrams.

```

1 package com.sample.interpreter.basic;
2 public class Client {
3     // Running the Client class as application.
4     public static void main(String[] args) throws Exception {
5         // Building an abstract syntax tree (AST).
6         AbstractExpression ntExpr2 = new NonTerminalExpression("ntExpr2");
7         ntExpr2.add(new TerminalExpression(" tExpr3"));
8         ntExpr2.add(new TerminalExpression(" tExpr4"));
9         ntExpr2.add(new TerminalExpression(" tExpr5"));
10        AbstractExpression ntExpr1 = new NonTerminalExpression("ntExpr1");
11        ntExpr1.add(new TerminalExpression(" tExpr1"));
12        ntExpr1.add(ntExpr2);
13        ntExpr1.add(new TerminalExpression(" tExpr2"));
14        Context context = new Context();
15        // Interpreting the AST (walking the tree).
16        ntExpr1.interpret(context);
17    }
18 }

ntExpr1:
    interpreting ... tExpr1
    interpreting ... ntExpr2
ntExpr2:
    interpreting ... tExpr3
    interpreting ... tExpr4
    interpreting ... tExpr5
ntExpr2 finished.
    interpreting ... tExpr2
ntExpr1 finished.

1 package com.sample.interpreter.basic;
2 public class Context {
3     // Input data and workspace for interpreting.
4 }

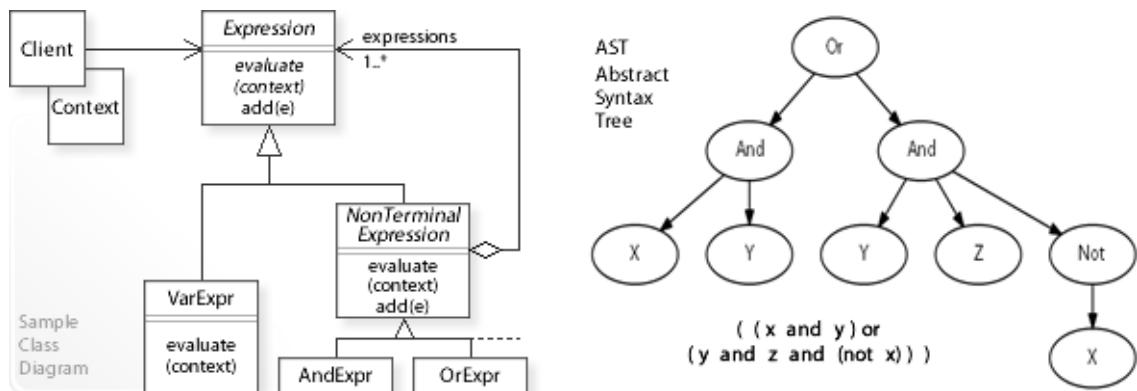
1 package com.sample.interpreter.basic;
2 public abstract class AbstractExpression {
3     private String name;
4     public AbstractExpression(String name) {
5         this.name = name;
6     }
7     public abstract void interpret(Context context);
8     //
9     public String getName() {
10         return name;
11     }
12     // Defining default implementation for child management operations.
13     public boolean add(AbstractExpression e) { // fail by default
14         return false;
15     }
16 }

```

```
1 package com.sample.interpreter.basic;
2 import java.util.ArrayList;
3 import java.util.List;
4 public class NonTerminalExpression extends AbstractExpression {
5     private List<AbstractExpression> expressions = new ArrayList<AbstractExpression>();
6     //
7     public NonTerminalExpression(String name) {
8         super(name);
9     }
10    public void interpret(Context context) {
11        System.out.println(getName() + ": ");
12        for (AbstractExpression expression : expressions) {
13            System.out.println(
14                "    interpreting ... " + expression.getName());
15            expression.interpret(context);
16        }
17        System.out.println(getName() + " finished.");
18    }
19    // Overriding the default implementation.
20    @Override
21    public boolean add(AbstractExpression e) {
22        return expressions.add(e);
23    }
24 }

1 package com.sample.interpreter.basic;
2 public class TerminalExpression extends AbstractExpression {
3     public TerminalExpression(String name) {
4         super(name);
5     }
6     public void interpret(Context context) {
7         // ...
8     }
9 }
```

## Sample Code 2



### Object finder with arbitrary (dynamically changeable) search criteria.

Defining a simple query language so that search expressions can be specified and interpreted (evaluated) dynamically.

For example, search expressions should look like:

$( (x \text{ and } y) \text{ or } (y \text{ and } z \text{ and } (\text{not } x)) )$

where  $x, y, z$  are terminal expressions (**VarExpr**) for arbitrary search criteria.

For example, searching product objects:

```
( (group == "PC" and prize > 1000) or
  (prize > 1000 and description containing "TV" and (group is not "PC")) ).
```

See also Implementation for a detailed description of this example.

```

1 package com.sample.interpreter.search;
2 import java.util.ArrayList;
3 import java.util.List;
4 import com.sample.data.Product;
5 import com.sample.data.SalesProduct;
6 public class Client {
7     // Running the Client class as application.
8     public static void main(String[] args) throws Exception {
9         //
10        // Creating a collection of product objects.
11        //
12        List<Product> products = new ArrayList<Product>();
13        products.add(new SalesProduct("PC1", "PC", "Product PC 1000", 1000));
14        products.add(new SalesProduct("PC2", "PC", "Product PC 2000", 2000));
15        products.add(new SalesProduct("PC3", "PC", "Product PC 3000", 3000));
16        //
17        products.add(new SalesProduct("TV1", "TV", "Product TV 1000", 1000));
18        products.add(new SalesProduct("TV2", "TV", "Product TV 2000", 2000));
19        products.add(new SalesProduct("TV3", "TV", "Product TV 3000", 3000));
19        //
21        // Representing the search expression:
22        // ( (x and y) or (y and z and (not x)) )
23        // by an AST (usually generated by a parser).
24        //
25        VarExpr x = new VarExpr("X");
26        VarExpr y = new VarExpr("Y");
27        VarExpr z = new VarExpr("Z");
28        //
29        Expression andExpr1 = new AndExpr();
30        andExpr1.add(x);
31        andExpr1.add(y);
32        //
33        Expression andExpr2 = new AndExpr();
34        andExpr2.add(y);
35        andExpr2.add(z);
36        Expression notExpr = new NotExpr();
37        notExpr.add(x);
38        andExpr2.add(notExpr);
39        //
40        Expression expression = new OrExpr();

```

```

41         expression.add(andExpr1);
42         expression.add(andExpr2);
43         //
44         // For each product:
45         // - specifying the search criteria dynamically and setting the context
46         // - interpreting (evaluating) the AST (search expression).
47         //
48         Context context = new Context();
49         for (Product p : products) {
50             // For example, searching products with:
51             // (group == "PC" and prize > 1000) or
52             // (prize > 1000 and description containing "TV" and group is not "PC").
53             // Setting VarExpr x,y,z in context to true or false.
54             context.setVarExpr(x, p.getGroup() == "PC" ? true : false);
55             context.setVarExpr(y, p.getPrice() > 1000 ? true : false);
56             context.setVarExpr(z, p.getDescription().contains("TV") ? true : false);
57             // Interpreting (evaluating) the AST (search expression).
58             if (expression.evaluate(context))
59                 System.out.println("Product found: " + p.getDescription());
60         }
61     }
62 }

Product found: Product PC 2000
Product found: Product PC 3000
Product found: Product TV 2000
Product found: Product TV 3000

```

```

1 package com.sample.interpreter.search;
2 import java.util.HashMap;
3 import java.util.Map;
4 public class Context {
5     // Workspace for mapping VarExp name to true or false.
6     Map<String, Boolean> varExprMap = new HashMap<String, Boolean>();
7     //
8     public void setVarExpr(VarExpr v, boolean b) {
9         varExprMap.put(v.getName(), b);
10    }
11    public boolean getVarExpr(String name) {
12        return varExprMap.get(name);
13    }
14 }

1 package com.sample.interpreter.search;
2 import java.util.Collections;
3 import java.util.Iterator;
4 public abstract class Expression {
5     public abstract boolean evaluate(Context context);
6     //
7     // Defining default implementation for child management operations.
8     public boolean add(Expression e) { // fail by default
9         return false;
10    }
11    public Iterator<Expression> iterator() {
12        return Collections.emptyIterator(); // null iterator
13    }
14 }

1 package com.sample.interpreter.search;
2 public class VarExpr extends Expression { // Terminal Expression
3     private String name;
4     public VarExpr(String name) {
5         this.name = name;
6     }
7     // Getting true or false from context.
8     public boolean evaluate(Context context) {
9         return context.getVarExpr(name);
10    }
11    public String getName() {
12        return name;
13    }
14 }

1 package com.sample.interpreter.search;
2 import java.util.ArrayList;

```

```
3 import java.util.Iterator;
4 import java.util.List;
5 public abstract class NonTerminalExpression extends Expression {
6     private List<Expression> expressions = new ArrayList<Expression>();
7     //
8     public abstract boolean evaluate(Context context);
9     // Overriding the default implementation.
10    @Override
11    public boolean add(Expression e) {
12        return expressions.add(e);
13    }
14    @Override
15    public Iterator<Expression> iterator() {
16        return expressions.iterator();
17    }
18 }

1 package com.sample.interpreter.search;
2 import java.util.Iterator;
3 public class AndExpr extends NonTerminalExpression { // NonTerminal Expression
4     public boolean evaluate(Context context) {
5         Iterator<Expression> it = iterator();
6         while (it.hasNext()) {
7             if (!it.next().evaluate(context))
8                 return false;
9         }
10        return true;
11    }
12 }

1 package com.sample.interpreter.search;
2 import java.util.Iterator;
3 public class OrExpr extends NonTerminalExpression { // NonTerminal Expression
4     public boolean evaluate(Context context) {
5         Iterator<Expression> it = iterator();
6         while (it.hasNext()) {
7             if (it.next().evaluate(context))
8                 return true;
9         }
10        return false;
11    }
12 }

1 package com.sample.interpreter.search;
2 import java.util.Iterator;
3 public class NotExpr extends NonTerminalExpression { // NonTerminal Expression
4     public boolean evaluate(Context context) {
5         Iterator<Expression> it = iterator();
6         while (it.hasNext()) {
7             if (it.next().evaluate(context))
8                 return false;
9         }
10        return true;
11    }
12 }

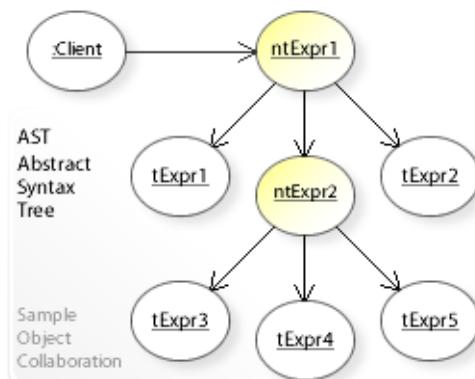
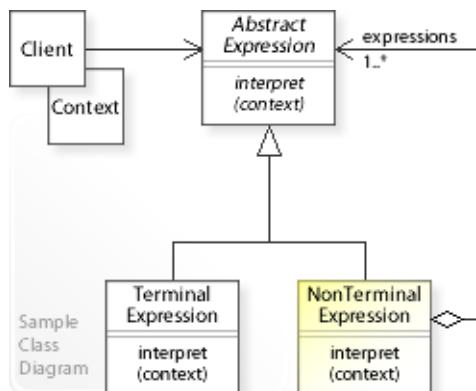
*****  
Other interfaces and classes used in this example.  
*****
```

```
1 package com.sample.data;
2 public interface Product {
3     void operation();
4     String getId();
5     String getGroup();
6     String getDescription();
7     long getPrice();
8 }

1 package com.sample.data;
2 public class SalesProduct implements Product {
3     private String id;
4     private String group;
5     private String description;
6     private long price;
7     //
```

```
8     public SalesProduct(String id, String group, String description, long price) {
9         this.id = id;
10        this.group = group;
11        this.description = description;
12        this.price = price;
13    }
14    public void operation() {
15        System.out.println("SalesProduct: Performing an operation ...");
16    }
17    public String getId() {
18        return id;
19    }
20    public String getGroup() {
21        return group;
22    }
23    public String getDescription() {
24        return description;
25    }
26    public long getPrice() {
27        return price;
28    }
29 }
```

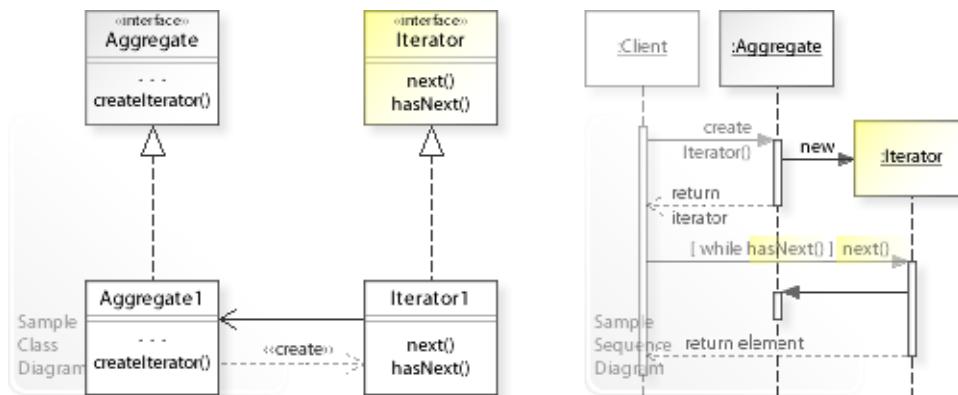
## Related Patterns



### Key Relationships (see also Overview)

- **Composite - Builder - Iterator - Visitor - Interpreter**
  - Composite provides a way to represent a part-whole hierarchy as a tree (composite) object structure.
  - Builder provides a way to create the elements of an object structure.
  - Iterator provides a way to traverse the elements of an object structure.
  - Visitor provides a way to define new operations for the elements of an object structure.
  - Interpreter represents a sentence in a simple language as a tree (composite) object structure (abstract syntax tree).

## Intent



The intent of the Iterator design pattern is to:

**"Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation."** [GoF]

See Problem and Solution sections for a more structured description of the intent.

- The Iterator design pattern solves problems like:
  - *How can the elements of an aggregate object be accessed and traversed without exposing its underlying representation?*
  - *How can an aggregate object be traversed in different ways without extending its interface with different traversal operations?*
- For example, an aggregate object like a list, set, or other kind of collection. It should be possible to access and traverse the elements of a collection in different ways independently from (without exposing) its internal representation.
- A direct access to an aggregate's representation (internal data structures) isn't possible because this would break its encapsulation.

## Problem



The Iterator design pattern solves problems like:

***How can the elements of an aggregate object be accessed and traversed without exposing its underlying representation?***

***How can multiple traversals be performed on the same aggregate object?***

See Applicability section for all problems Iterator can solve.

- One way to access and traverse an aggregate object without exposing its underlying representation is to put the access and traversal operations into the interface of the aggregate object (`Aggregate`).

For example, the minimal interface for traversing front-to-back: `next()`, `hasNext()`.

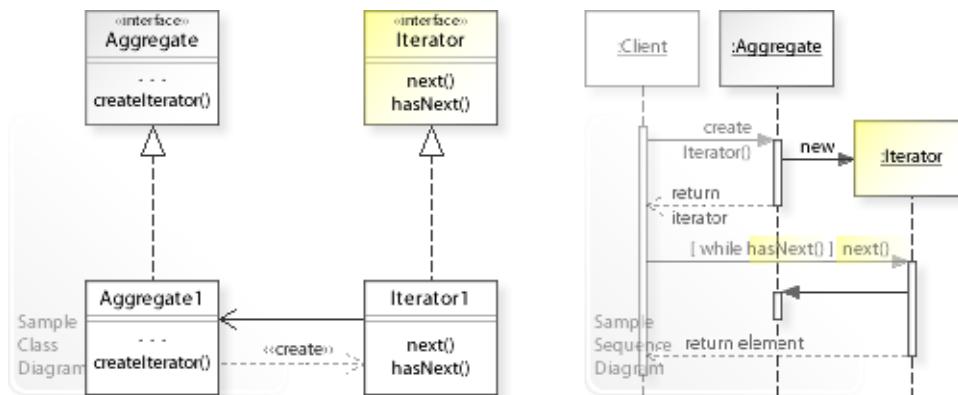
This commits the `Aggregate` to particular traversal operations, which makes it hard to add new operations later without having to change the `Aggregate` interface and implementation classes.

For example, traversing back-to-front: `previous()`, `hasPrevious()`.

"But you probably don't want to bloat the `List` [`Aggregate`] interface with operations for different traversals, even if you could anticipate the ones you will need." [GoF, p257]

- *That's the kind of approach to avoid if we want to traverse an aggregate object independently from its representation, and if we want to perform multiple traversals on the same aggregate.*
- For example, an aggregate object like a list, set, or other kind of collection.  
It should be possible to access and traverse collections in different ways independently from (without having to know) how they are represented (see also Sample Code / Example 2).

## Solution



The Iterator pattern describes a solution:

**Define separate `Iterator` objects that encapsulate different ways to traverse an aggregate object.**

**Work through an `Iterator` object to traverse an aggregate object.**

Describing the Iterator design in more detail is the theme of the following sections.

See Applicability section for all problems Iterator can solve.

- "The key idea in this pattern is to take the responsibility for access and traversal out of the list [aggregate] object and put it into an `iterator` object." [GoF, p257]

- Define separate `Iterator` objects:**

- Define a common interface for accessing and traversing an aggregate object.  
For example, the minimal interface for traversing front-to-back: `next()`, `hasNext()`

- Define classes (`Iterator1`, ...) that implement the `Iterator` interface.

An iterator is usually implemented as inner class of an aggregate class. This enables the iterator to access the internal data structures of the aggregate (see Implementation).

- New traversal operations can be implemented and existing ones can be changed independently from the aggregate object. Clients refer to the `Iterator` interface to access and traverse an aggregate object without knowing how the interface is implemented and the aggregate is represented. For example, traversing front-to-back:

```
Iterator iterator = aggregate.createIterator();
while (iterator.hasNext()) { ... iterator.next(); ... }
```

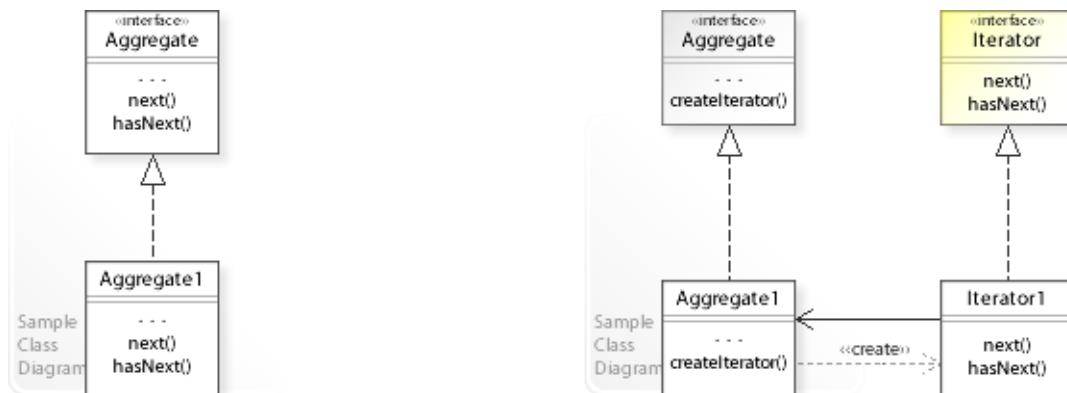
- Work through an `Iterator` object to traverse an aggregate object.**

- Clients can create and work through different iterators to traverse an aggregate in different ways, and even more, multiple traversals can be in progress on the same aggregate (simultaneous traversals).

## Background Information

- For example, the Java Collections Framework provides
  - a general purpose `iterator`  
(`next()`, `hasNext()`, `remove()`)
  - and an extended `listIterator`  
(`next()`, `hasNext()`, `previous()`, `hasPrevious()`, `remove()`, ... ).
- Consequently, there are two factory methods for creating an iterator (`iterator()` and `listIterator()`).

## Motivation 1



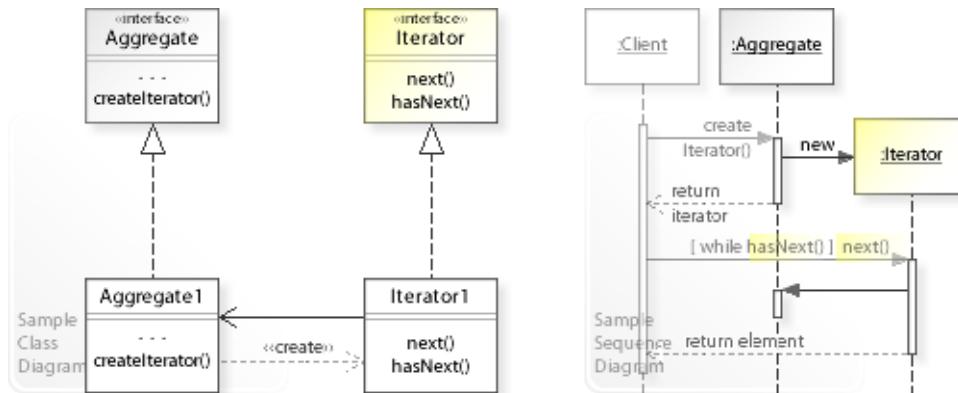
**Consider the left design (problem):**

- Included access and traversal.
  - The responsibility for access and traversal is included in the aggregate.
  - This makes it hard to define new traversal operations independently from the aggregate.
- No simultaneous traversals.
  - Only one traversal can be in progress on the same aggregate.

**Consider the right design (solution):**

- Separated access and traversal.
  - The responsibility for access and traversal is separated from the aggregate.
  - This makes it easy to define new traversal operations independently from the aggregate.
- Simultaneous traversals.
  - Multiple traversals can be in progress on the same aggregate.

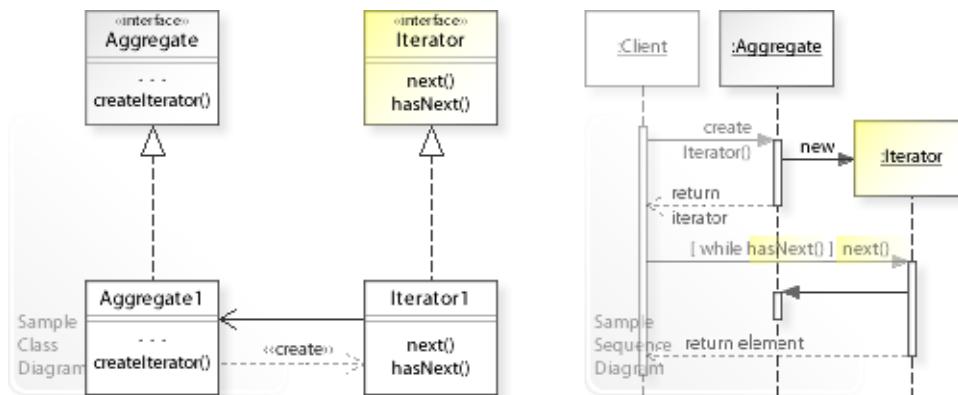
## Applicability



## Design Problems

- **Accessing and Traversing Object Structures**
  - How can the elements of an aggregate object be accessed and traversed without exposing its underlying representation (independently from its representation)?
  - How can new traversal operations be defined for an aggregate object without changing its classes?
- **Supporting Multiple Traversals**
  - How can multiple traversals be performed on the same aggregate object?

## Structure, Collaboration



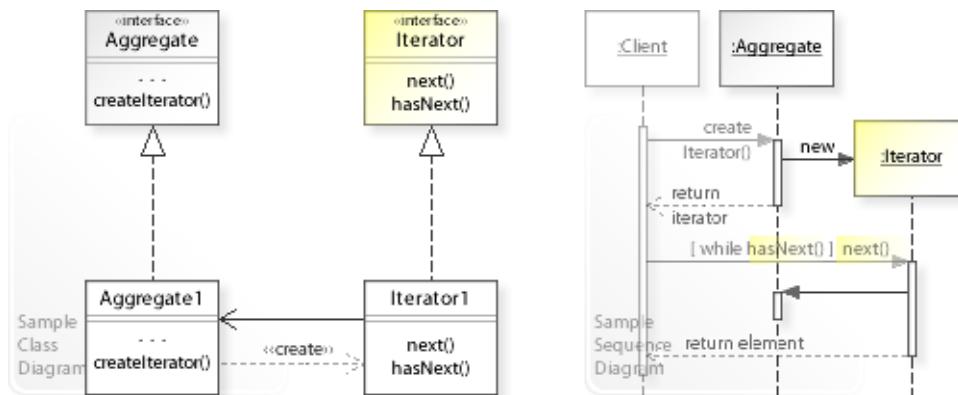
### Static Class Structure

- `Aggregate`
  - Defines an interface for creating an `Iterator` object.
- `Aggregate1,...`
  - Implement the `Aggregate` interface.
  - The factory method `createIterator()` returns an instance of the iterator class (`Iterator1`) that iterates over the aggregate class (`Aggregate1`).
- `Iterator`
  - Defines an interface for accessing and traversing the elements of an `Aggregate`.
- `Iterator1,...`
  - Implement the `Iterator` interface.
  - An iterator is usually implemented as inner class of an aggregate class. This enables the iterator to access the internal (private) data structures of the aggregate.

### Dynamic Object Collaboration

- In this sample scenario, a `Client` object creates and uses an `Iterator` object to traverse an `Aggregate` object front-to-back.
- The interaction starts with the `Client` object that calls `createIterator()` on the `Aggregate` object.
- `Aggregate` creates an `Iterator` object and returns (a reference to) it to the `Client`.
- Thereafter, the `Client` uses the `Iterator` to traverse the `Aggregate` front-to-back:  
`while (iterator.hasNext()) { ... iterator.next(); ... }`
- See also Sample Code / Example 1.

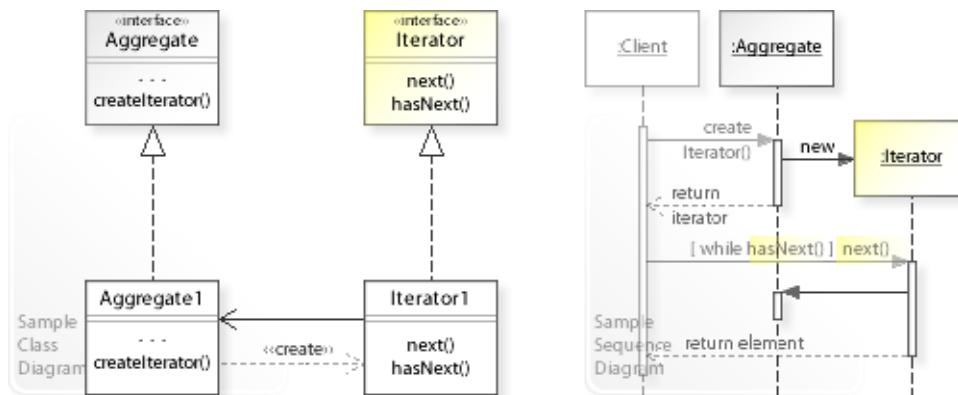
## Consequences



### Advantages (+)

- Enables simultaneous traversals.
  - Multiple traversals can be in progress on the same aggregate.
- Simplifies the aggregate interface.
  - The iterator interface is separated, and this simplifies the aggregate interface.
- Allows changing the traversal dynamically at run-time.
  - "Iterators make it easy to change the traversal algorithm: just replace the iterator instance with a different one." [GoF, p260]

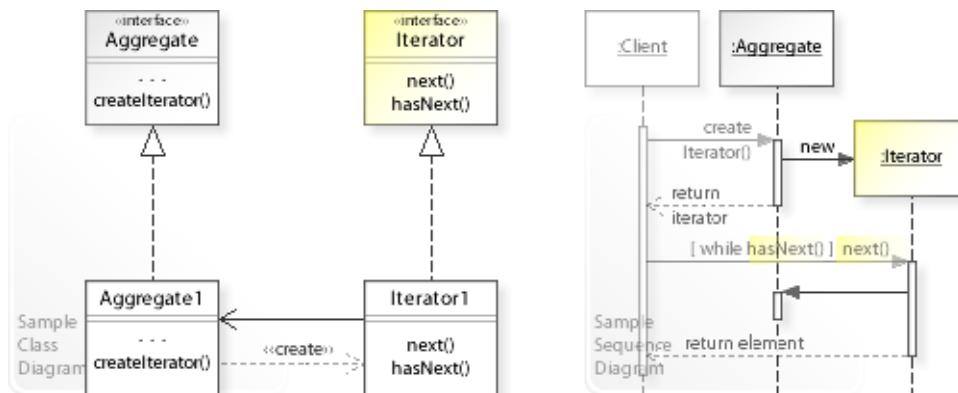
## Implementation



### Implementation Issues

- **Iterators may have privileged access.**
  - In standard object-oriented languages, an iterator is usually implemented as **inner class** of an aggregate class.
  - This enables the iterator to access the private data structures of the aggregate (see Sample Code / Example 1 and 2).
  - Another way is to extend the `Aggregate` interface to let iterators access the aggregate efficiently.
- **Creating Iterators.**
  - The aggregate provides an interface for creating an iterator object (`createIterator()`).
  - The aggregate class is responsible for instantiating the appropriate iterator class.
  - This is an example of a factory method (see Factory Method).

## Sample Code 1



Basic Java code for implementing the sample UML diagrams.

```

1 package com.sample.iterator.basic;
2 public class Client {
3     // Running the Client class as application.
4     public static void main(String[] args) {
5         // Setting up an aggregate.
6         Aggregate<String> aggregate = new Aggregate1<String>(3);
7         aggregate.add(" ElementA ");
8         aggregate.add(" ElementB ");
9         aggregate.add(" ElementC ");
10        //
11        // Creating an iterator.
12        Iterator<String> iterator = aggregate.createIterator();
13        //
14        System.out.println("Traversing the aggregate front-to-back:");
15        while (iterator.hasNext()) {
16            System.out.println(iterator.next());
17        }
18    }
19 }

Traversing the aggregate front-to-back:
ElementA
ElementB
ElementC

1 package com.sample.iterator.basic;
2 public interface Aggregate<E> {
3     // ...
4     Iterator<E> createIterator();
5     boolean add(E element);
6 }

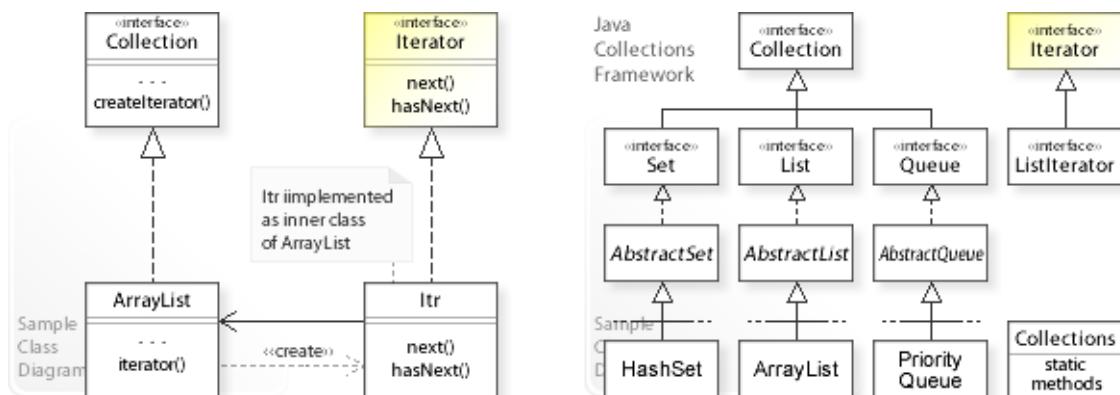
1 package com.sample.iterator.basic;
2 public interface Iterator<E> {
3     E next();
4     boolean hasNext();
5 }

1 package com.sample.iterator.basic;
2 import java.util.NoSuchElementException;
3 public class Aggregate1<E> implements Aggregate<E> { // E = Type parameter
4     // Hiding the representation.
5     private Object[] elementData; // represented as object array
6     private int idx = 0;
7     private int size;
8     //
9     public Aggregate1(int size) {
10         if (size < 0)
11             throw new IllegalArgumentException("size: " + size);
12         this.size = size;
13         elementData = new Object[size];
14     }
15     public boolean add(E element) {

```

```
16         if (idx < size) {
17             elementData[idx++] = element;
18             return true;
19         } else
20             return false;
21     }
22     public int getSize() {
23         return size;
24     }
25     // Factory method for instantiating Iterator1.
26     public Iterator<E> createIterator() {
27         return new Iterator1<E>();
28     }
29     //
30     // Implementing Iterator1 as inner class.
31     //
32     private class Iterator1<E> implements Iterator<E> {
33         // Holds the current position in the traversal.
34         private int cursor = 0; // index of next element to return
35         //
36         public boolean hasNext() {
37             return cursor < size;
38         }
39         public E next() { // E = Type of element returned by this method
40             if (cursor >= size)
41                 throw new NoSuchElementException();
42             return (E) elementData[cursor++]; // cast from Object to E
43         }
44     }
45 }
```

## Sample Code 2



### Using the iterator provided by the Java Collections Framework.

```

1 package com.sample.iterator.collection;
2 import java.util.List;
3 import java.util.ArrayList;
4 import java.util.Iterator;
5 import java.util.ListIterator;
6 public class Client {
7     // Running the Client class as application.
8     public static void main(String[] args) throws Exception {
9         // Setting up a collection (list) of customers.
10        int size = 50;
11        List<Customer> list = new ArrayList<Customer>();
12        for (int i = 0; i < size; i++)
13            list.add(new Customer1("Customer" + i, 100));
14        //
15        //
16        //=====
17        System.out.println("(1) Front-to-end traversal (via basic iterator):  ");
18        //=====
19        int count = 0;
20        long sum = 0;
21        Iterator<Customer> iterator = list.iterator();
22        while (iterator.hasNext()) {
23            sum += iterator.next().getSales();
24            count++;
25        }
26        System.out.println("    Total sales of " + count + " customers is: " + sum);
27        //
28        //
29        //=====
30        System.out.println("\n(2) Backward traversal from position-to-front " +
31                           (size / 2) + " (via list Iterator):  ");
32        //=====
33        count = 0;
34        sum = 0;
35        ListIterator<Customer> listIterator = list.listIterator(size / 2);
36        while (listIterator.hasPrevious()) {
37            sum += listIterator.previous().getSales();
38            count++;
39        }
40        System.out.println("    Total sales of " + count + " customers is: " + sum);
41        //
42        //
43        //=====
44        System.out.println("\n(3) Direct access customer (via list interface):  ");
45        //=====
46        int position = size / 10;
47        Customer customer = list.get(position);
48        System.out.println("    Customer at position " +
49                           position + " is: " + customer.getName());
50        //
51        //
52        //=====
53        System.out.println("\n(4) Search customer (via list interface):  ");
54        //=====
```

```
55         int index = list.indexOf(customer);
56         System.out.println("    Index of first occurrence of " +
57                             list.get(index).getName() + " is: " + index);
58     }
59 }

(1) Front-to-end traversal (via basic iterator):
Total sales of 50 customers is: 5000

(2) Backward traversal from position-to-front 25 (via list Iterator):
Total sales of 25 customers is: 2500

(3) Direct access customer (via list interface):
Customer at position 5 is: Customer5

(4) Search customer (via list interface):
Index of first occurrence of Customer5 is: 5

1 package java.util; // Provided by the Java platform.
2 public interface Iterator<E> {
3     boolean hasNext();
4     E next();
5     ...
6 }

1 package java.util; // Provided by the Java platform.
2 public interface ListIterator<E> extends Iterator<E> {
3     boolean hasPrevious();
4     E previous();
5     ...
6 }

1 package com.sample.iterator.collection;
2 public interface Customer {
3     long getSales();
4     String getName();
5 }

1 package com.sample.iterator.collection;
2 public class Customer1 implements Customer {
3     private String name;
4     private long sales;
5     public Customer1(String name, long sales) {
6         this.name = name;
7         this.sales = sales;
8     }
9     public long getSales() {
10        return sales;
11    }
12    public String getName() {
13        return name;
14    }
15 }

*****
Background Information:
Copyright (c) 1997, 2010, Oracle and/or its affiliates.
All rights reserved.
*****

1 package java.util; // Provided by the Java platform.
2 public interface Collection<E> extends Iterable<E> {
3     int size();
4     boolean isEmpty();
5     boolean contains(Object o);
6     ...
7     Iterator<E> iterator();
8     ...
9     boolean add(E e);
10    boolean remove(Object o);
11    ...
12 }

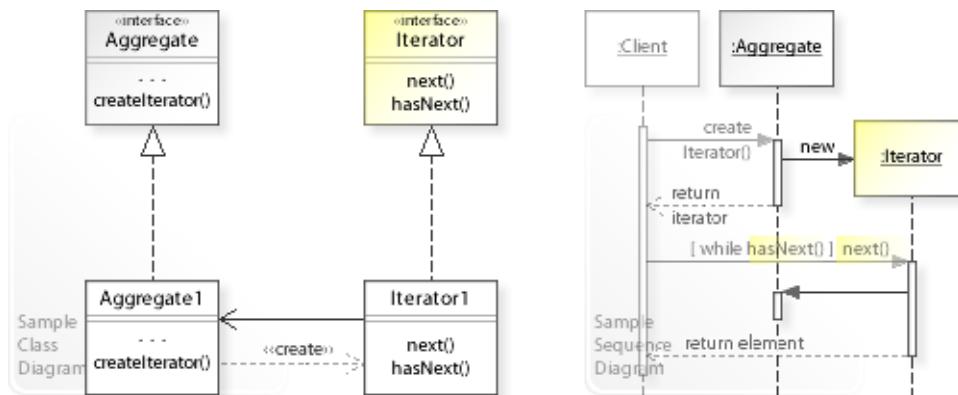
1 // List is an ordered collection (also known as a sequence).
2 // Unlike sets, lists allow duplicate elements.
```

```
3 // ListIterator allows element insertion/replacement and
4 // bidirectional access in addition to Iterator.
5 //
6 package java.util; // Provided by the Java platform.
7 public interface List<E> extends Collection<E> {
8     // Iterating
9     ListIterator<E> listIterator(); // Starting at position 0
10    ListIterator<E> listIterator(int index); // Starting at position index
11    // Positional access
12    E get(int index);
13    E set(int index, E element);
14    // Search
15    int indexOf(object o);
16    // ...
17 }

1 // ArrayList is a resizable array implementation of the List interface.
2 package java.util; // Provided by the Java platform.
3 public class ArrayList<E> extends AbstractList<E>
4     implements List<E>, RandomAccess, Cloneable, java.io.Serializable {
5     // The array buffer into which the elements of the ArrayList are stored.
6     // The capacity of the ArrayList is the length of this array buffer.
7     private transient Object[] elementData;
8     // The size of the ArrayList (the number of elements it contains).
9     private int size;
10    // ...
11    public ListIterator<E> listIterator() {
12        return new ListItr(0);
13    }
14    public Iterator<E> iterator() {
15        return new Itr();
16    }
17    private class Itr implements Iterator<E> {
18        int cursor; // index of next element to return
19        int lastRet = -1; // index of last element returned; -1 if no such
20        int expectedModCount = modCount;
21        public boolean hasNext() {
22            return cursor != size;
23        }
24        public E next() {
25            checkForComodification();
26            int i = cursor;
27            if (i >= size)
28                throw new NoSuchElementException();
29            Object[] elementData = ArrayList.this.elementData;
30            if (i >= elementData.length)
31                throw new ConcurrentModificationException();
32            cursor = i + 1;
33            return (E) elementData[lastRet = i];
34        } // ...
35    }
36    private class ListItr extends Itr implements ListIterator<E> {
37        ListItr(int index) {
38            super();
39            cursor = index;
40        }
41        public boolean hasPrevious() {
42            return cursor != 0;
43        }
44        public int nextIndex() {
45            return cursor;
46        }
47        public int previousIndex() {
48            return cursor - 1;
49        }
50        public E previous() {
51            checkForComodification();
52            int i = cursor - 1;
53            if (i < 0)
54                throw new NoSuchElementException();
55            Object[] elementData = ArrayList.this.elementData;
56            if (i >= elementData.length)
57                throw new ConcurrentModificationException();
58            cursor = i;
59            return (E) elementData[lastRet = i];
60        } // ...
```

```
61      } // ...
62 }
```

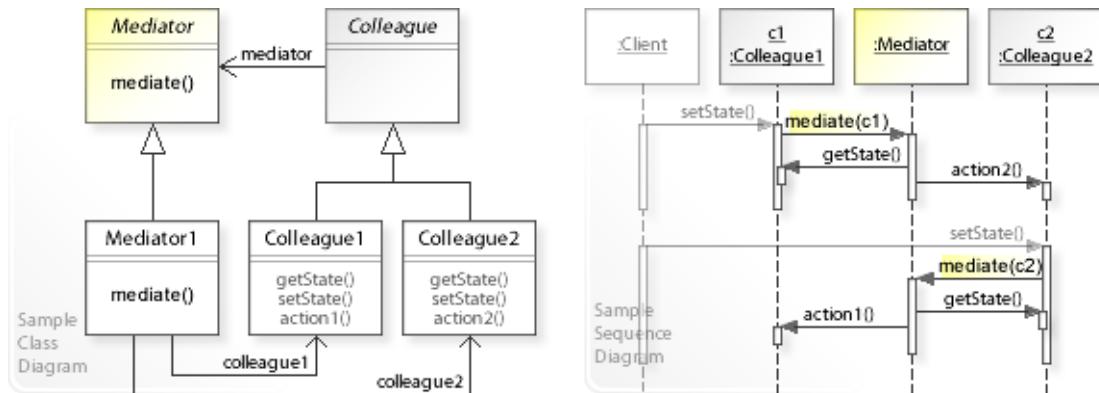
## Related Patterns



### Key Relationships (see also Overview)

- **Composite - Builder - Iterator - Visitor - Interpreter**
  - Composite provides a way to represent a part-whole hierarchy as a tree (composite) object structure.
  - Builder provides a way to create the elements of an object structure.
  - Iterator provides a way to traverse the elements of an object structure.
  - Visitor provides a way to define new operations for the elements of an object structure.
  - Interpreter represents a sentence in a simple language as a tree (composite) object structure (abstract syntax tree).
- **Iterator - Factory Method**
  - The operation for creating an iterator object is a factory method.

## Intent



The intent of the Mediator design pattern is to:

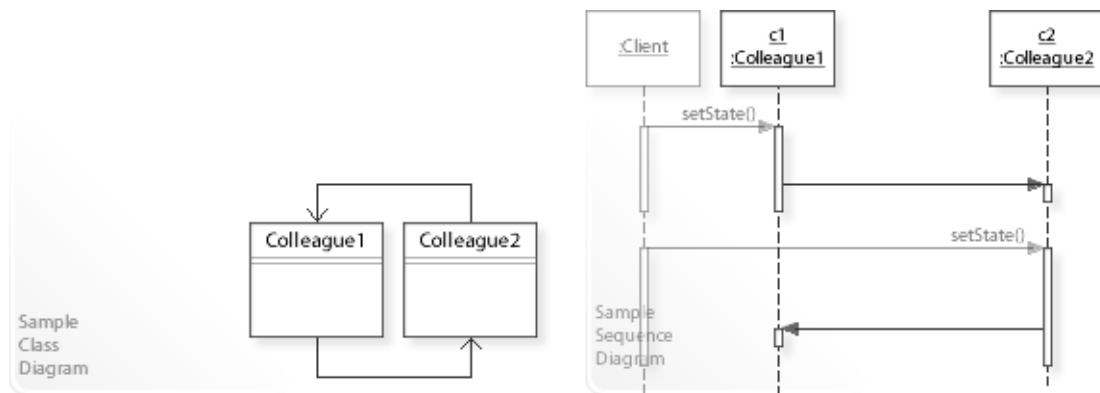
**"Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently."** [GoF]

See Problem and Solution sections for a more structured description of the intent.

- The Mediator design pattern solves problems like:
  - *How can a set of interacting objects be defined without making the objects tightly coupled?*
  - *How can the interaction between a set of objects be changed independently?*
- *Coupling* is the degree to which things depend on each other.
  - *Tightly coupled objects* depend on (refer to and know about) many other objects and interfaces, which makes them hard to implement, change, and reuse.
  - *Loosely coupled objects* have only minimal dependencies on other objects and interfaces (most often achieved by working through a common interface), which makes them easier to implement, change, and reuse.
- The Mediator pattern describes how to solve such problems:
  - *Define an object that encapsulates how a set of objects interact.*
  - The key idea in this pattern is to let the objects interact with each other indirectly through a Mediator object that encapsulates the interaction behavior.

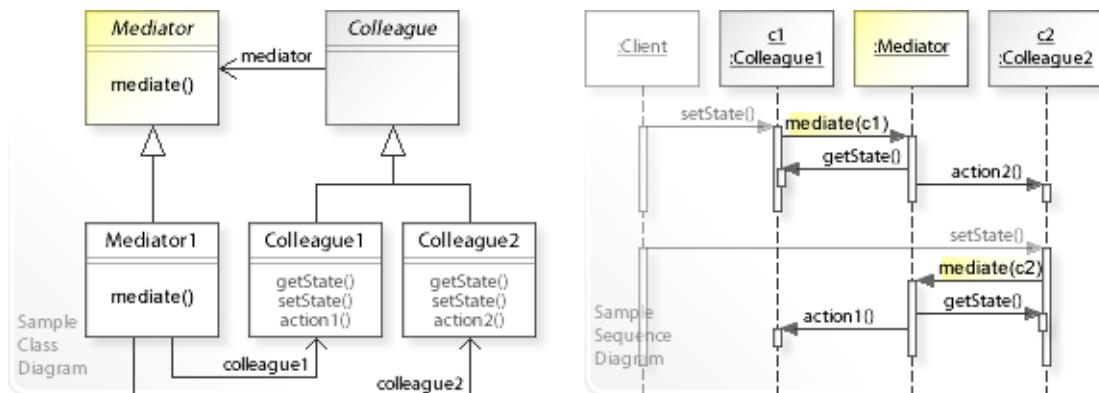
This makes the objects loosely coupled because they only depend on (refer to and know about) the simple Mediator interface.

## Problem



- An inflexible way to define a set of interacting objects (`Colleague1`, `Colleague2`, ...) is that each object refers to and knows about the other objects, which results in many interconnections between the objects.
- This makes the objects tightly coupled.  
Tightly coupled objects depend on (refer to and know about) many other objects and interfaces, which makes them hard to implement, change, test, and reuse.  
Furthermore, it's hard to change the way the objects interact because the interaction is distributed among the objects.
- That's the kind of approach to avoid if we want to keep a set of interacting objects loosely coupled.*
- For example, defining a set of interacting objects (like buttons, menu items, and input/display fields) in a GUI/Web application.  
It should be possible to change the interaction behavior independently from (without having to change) the objects. The objects should be reusable in different applications.
- For example, maintaining consistency (synchronizing state) between a set of related or dependent objects.

## Solution



The Mediator pattern describes a solution:

**Define a separate `Mediator` object that encapsulates how a set of objects interact.**

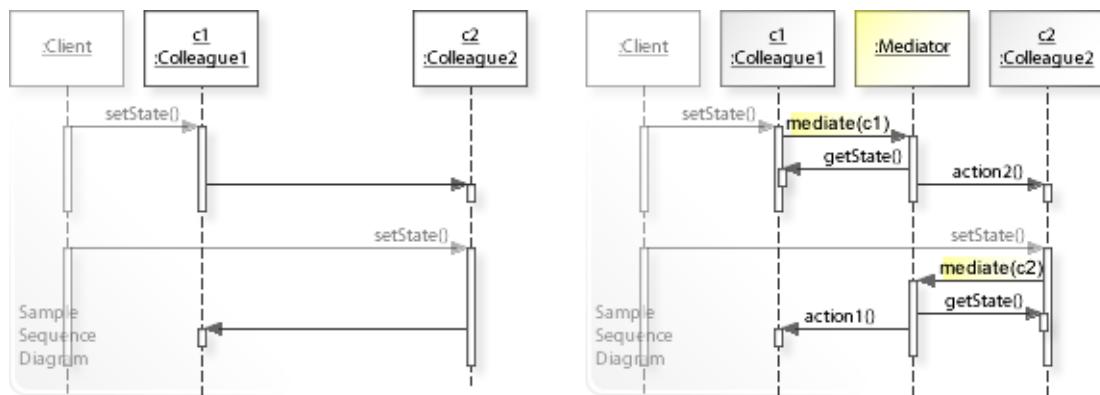
**Delegate interaction to a `Mediator` object.**

Describing the Mediator design in more detail is the theme of the following sections.

See Applicability section for all problems Mediator can solve.

- The key idea in this pattern is to let objects interact with each other indirectly through a common `Mediator` object that encapsulates (centralizes) the interaction behavior.
- **Define a separate `Mediator` object:**
  - Define an interface for interacting with colleagues (`Mediator | mediate()`).
  - Define a class (`Mediator1`) that implements the interaction behavior to control and coordinate the interaction between colleagues.
- This enables *compile-time* flexibility (via class inheritance).  
The interaction behavior can be changed independently from colleagues (which makes it possible to add and/or remove colleagues).
- **Colleagues delegate (the responsibility for) performing their interaction to a `Mediator` object** (`mediator.mediate()`).  
Instead of interacting with each other directly, colleagues interact with each other indirectly through a `Mediator` object.
- This makes colleagues loosely coupled.  
Colleagues only know their mediator and have no explicit knowledge of each other.  
"[...] each colleague communicates with its mediator whenever it would have otherwise communicated with another colleague." GoF [p277]

## Motivation 1



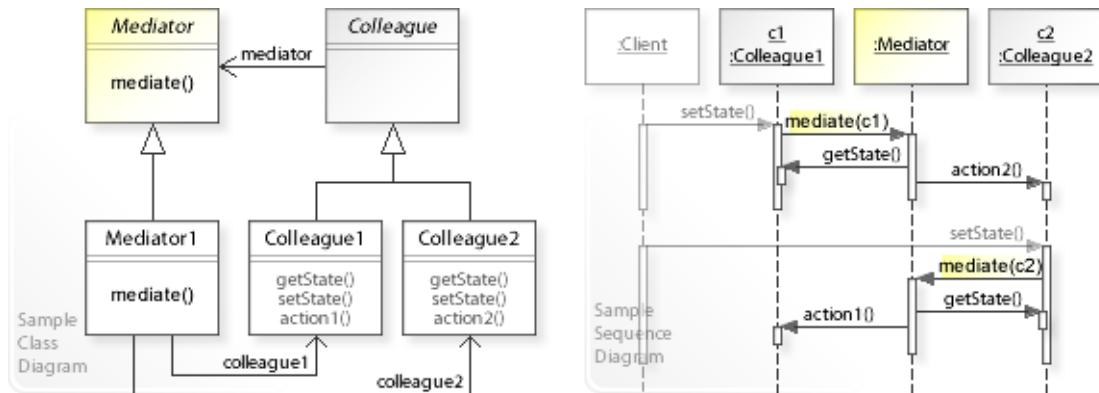
**Consider the left design (problem):**

- Tightly coupled colleagues.
  - A set of colleagues interact with each other directly by referring to and knowing about each other (tight coupling).
  - Tightly coupled objects depend on (refer to and know about) many other objects with different interfaces, which makes them hard to implement, change, and reuse.
- Distributed interaction behavior.
  - It's hard to change the way the objects interact with each other because the interaction is distributed among the objects.

**Consider the right design (solution):**

- Loosely coupled colleagues.
  - A set of colleagues interact with each other indirectly by referring to and knowing about the `Mediator` interface (loose coupling).
  - Loosely coupled objects have only minimal dependencies (by working through a common interface), which makes them easier to implement, change, and reuse.
- Encapsulated interaction behavior.
  - It's easy to change the way the objects interact with each other because it is encapsulated in a separate `Mediator` object.
  - Note that the mediator itself isn't designed for being reusable, but it is designed for making colleagues reusable.
  - "This can make the mediator itself a monolith that's hard to maintain." [GoF, p277]

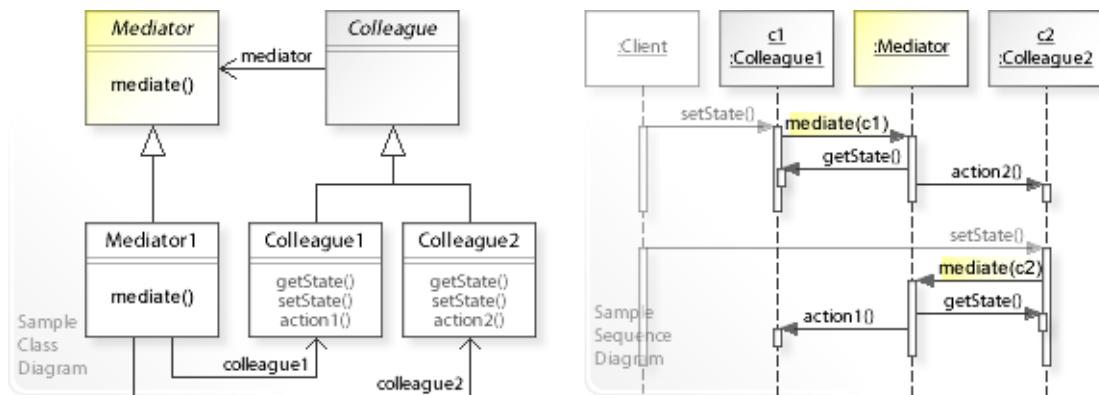
## Applicability



## Design Problems

- **Defining Sets of Interacting Objects**
  - How can a set of interacting objects be defined without making the objects tightly coupled?
  - How can the interaction between a set of objects be changed independently?

## Structure, Collaboration



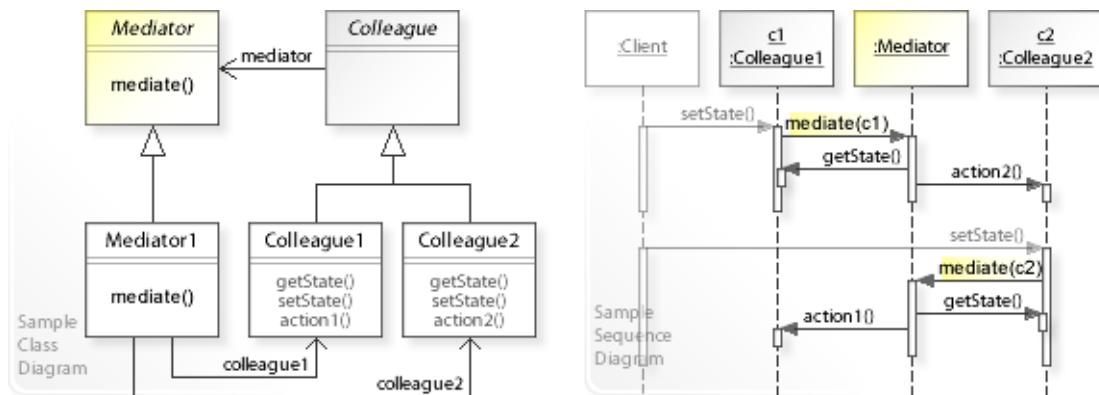
### Static Class Structure

- **Mediator**
  - Defines an interface for interacting with colleagues.
- **Mediator1**
  - Implements the interaction behavior to control and coordinate the interaction among colleagues.
  - Maintains explicit references (`colleague1`, `colleague2`, ...) to its colleagues.
- **Colleague**
  - Maintains a reference (`mediator`) to a **Mediator** object.
- **Colleague1**, **Colleague2**, ...
  - Refer to their **Mediator** object instead of referring to each other directly.

### Dynamic Object Collaboration

- In this sample scenario, a **Mediator** object mediates the interaction between colleagues (**Colleague1** and **Colleague2** objects).
- The interaction starts with a **Client** object that changes the state of (calls `setState()` on) the **Colleague1** object, which causes **Colleague1** to call `mediate(this)` on its **Mediator** object.
- **Colleague1** passes itself (`this`) to the **Mediator** so that the **Mediator** can call back and get the changed data.
- The **Mediator** gets the changed data from **Colleague1** and performs an `action2()` on **Colleague2**.
- Thereafter, the **Client** changes the state of **Colleague2**, which causes **Colleague2** to call `mediate(this)` on its **Mediator**.
- The **Mediator** now gets the changed data from **Colleague2** and performs an `action1()` on **Colleague1**.
- See also Sample Code / Example 1.

## Consequences



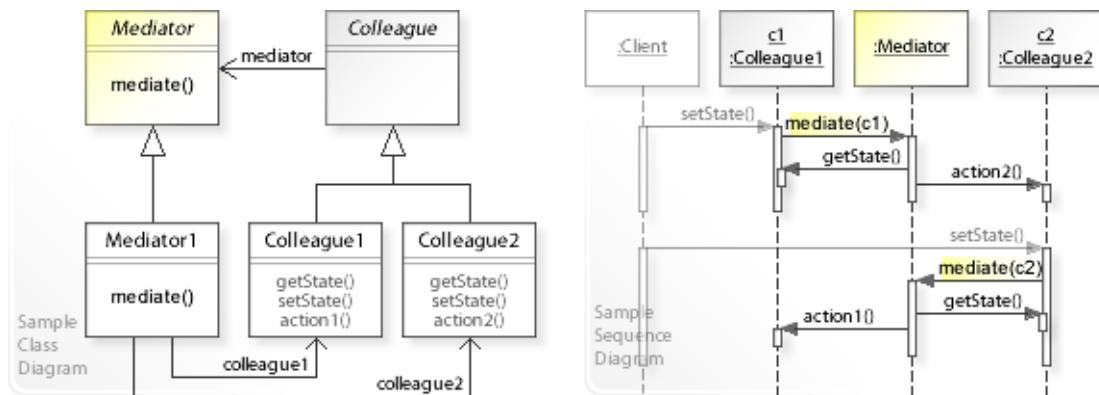
### Advantages (+)

- Decouples colleagues.
  - The colleagues interact with each other indirectly through the **Mediator** object/interface and have no explicit knowledge of each other.
  - Loosely coupled objects are easier to implement, change, and reuse.
- Centralizes interaction behavior.
  - The mediator encapsulates (centralizes) the interaction behavior that otherwise would be distributed among the interacting colleagues.
- Makes changing the interaction behavior easy.
  - The interaction behavior can be changed independently from colleagues by adding new **Mediator** subclasses.

### Disadvantages (-)

- Can make the mediator complex.
  - Because the mediator encapsulates (centralizes) the interaction behavior of a set of objects, it can become complex.
  - The complexity increases with the complexity and number of colleagues.
  - "This can make the mediator itself a monolith that's hard to maintain." [GoF, p277]

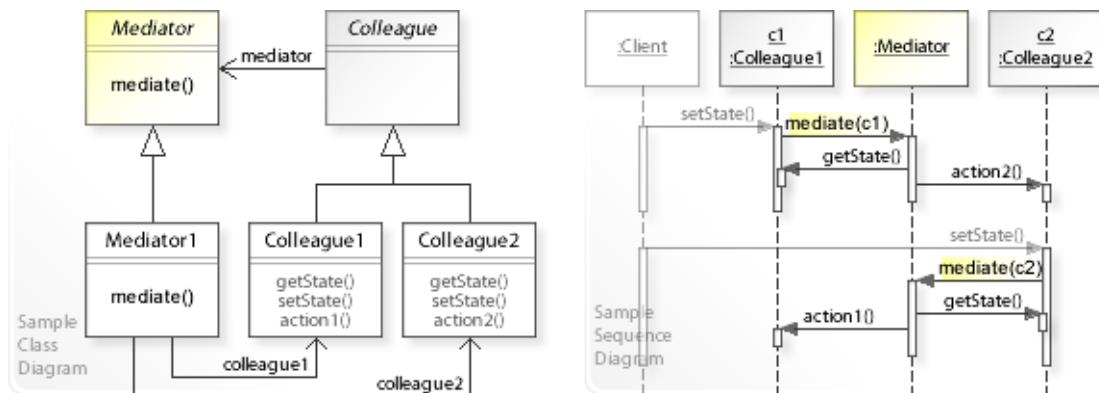
## Implementation



### Implementation Issues

- **Implementing the interaction behavior.**
    - The mediator is responsible for controlling and coordinating the interactions (updates) of the colleagues.
    - The complexity of mediator increases with the complexity and number of colleagues.
    - Colleagues interact with each other indirectly by calling `mediate(this)` on their mediator.
    - A colleague passes itself (`this`) to the mediator so that the mediator can call back to know what changed (to get the required data from the colleague).
- "When communicating with the mediator, a colleague passes itself as an argument, allowing the mediator to identify the sender." [GoF, p278]

## Sample Code 1



### Basic Java code for implementing the sample UML diagrams.

```

1 package com.sample.mediator.basic;
2 public class Client {
3     // Running the Client class as application.
4     public static void main(String[] args) {
5         Mediator1 mediator = new Mediator1();
6         // Configuring colleagues with a mediator.
7         Colleague1 c1 = new Colleague1(mediator);
8         Colleague2 c2 = new Colleague2(mediator);
9         // Setting mediator's colleagues.
10        mediator.setColleagues(c1, c2);
11        //
12        System.out.println("(1) Client      : Changing state of Colleague1 ...");
13        c1.setState("Hello World1!");
14        //
15        System.out.println("\n(2) Client      : Changing state of Colleague2 ...");
16        c2.setState("Hello World2!");
17    }
18 }

(1) Client      : Changing state of Colleague1 ...
Colleague1: State changed to: Hello World1! Calling the mediator ...
Mediator   : Mediating the interaction ...
Colleague2: State synchronized to: Hello World1!

(2) Client      : Changing state of Colleague2 ...
Colleague2: State changed to: Hello World2! Calling the mediator ...
Mediator   : Mediating the interaction ...
Colleague1: State synchronized to: Hello World2!


1 package com.sample.mediator.basic;
2 public abstract class Mediator {
3     // Mediating the interaction between colleagues.
4     public abstract void mediate(Colleague colleague);
5 }

1 package com.sample.mediator.basic;
2 public class Mediator1 extends Mediator {
3     private Colleague1 colleague1;
4     private Colleague2 colleague2;
5     void setColleagues(Colleague1 colleague1, Colleague2 colleague2) {
6         this.colleague1 = colleague1;
7         this.colleague2 = colleague2;
8     }
9     public void mediate(Colleague colleague) {
10        System.out.println("    Mediator : Mediating the interaction ...");
11        // Message from colleague1 that its state has changed.
12        if (colleague == colleague1) {
13            // Performing an action on colleague2.
14            String state = colleague1.getState();
15            colleague2.action2(state);
16        }
17        // Message from colleague2 that its state has changed.
18        if (colleague == colleague2) {
  
```

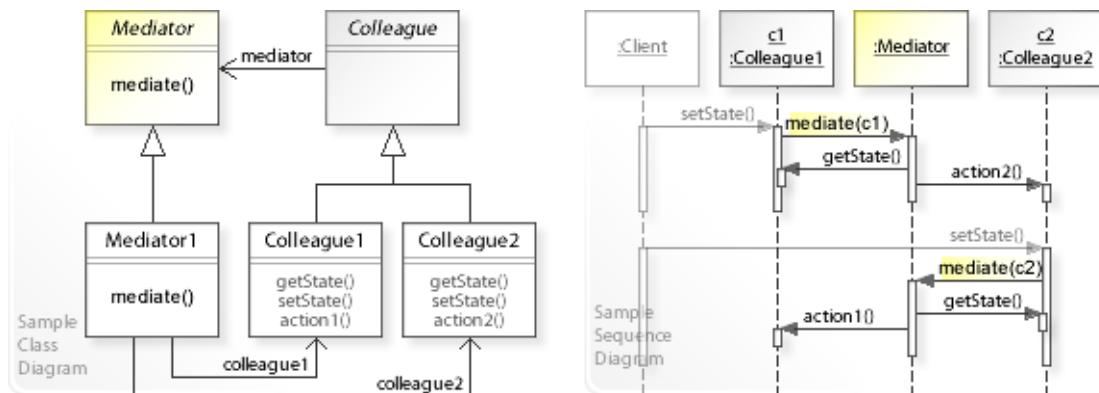
```
19          // Performing an action on colleague1.
20          String state = colleague2.getState();
21          colleague1.action1(state);
22      }
23  }
24 }
```

```
1 package com.sample.mediator.basic;
2 public abstract class Colleague {
3     Mediator mediator;
4     public Colleague(Mediator mediator) {
5         this.mediator = mediator;
6     }
7 }

1 package com.sample.mediator.basic;
2 public class Colleague1 extends Colleague {
3     private String state;
4     public Colleague1(Mediator mediator) {
5         super(mediator); // Calling the super class constructor.
6     }
7     public String getState() {
8         return state;
9     }
10    void setState(String state) {
11        if (state != this.state) {
12            this.state = state;
13            System.out.println("    Colleague1: State changed to: " + this.state +
14                " Calling the mediator ...");
15            mediator.mediate(this);
16        }
17    }
18    void action1 (String state) {
19        // For example, synchronizing and displaying state.
20        this.state = state;
21        System.out.println("    Colleague1: State synchronized to: " + this.state);
22    }
23 }

1 package com.sample.mediator.basic;
2 public class Colleague2 extends Colleague {
3     private String state;
4     public Colleague2(Mediator mediator) {
5         super(mediator);
6     }
7     public String getState() {
8         return state;
9     }
10    void setState(String state) {
11        if (state != this.state) {
12            this.state = state;
13            System.out.println("    Colleague2: State changed to: " + this.state +
14                " Calling the mediator ...");
15            mediator.mediate(this);
16        }
17    }
18    void action2 (String state) {
19        // For example, synchronizing and displaying state.
20        this.state = state;
21        System.out.println("    Colleague2: State synchronized to: " + this.state);
22    }
23 }
```

## Related Patterns

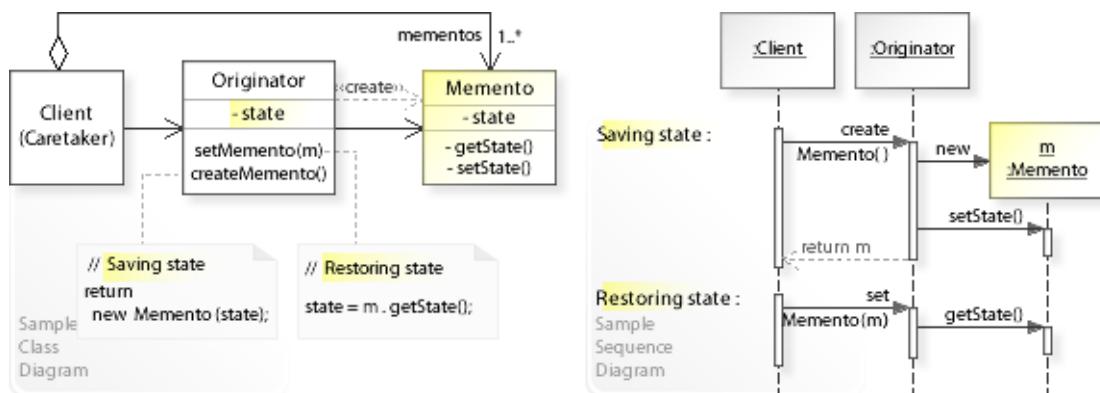


### Key Relationships (see also Overview)

- **Mediator - Observer**

- Mediator provides a way to keep a set of interacting objects loosely coupled by defining a **Mediator** object that controls and coordinates the interaction.  
Colleagues interact with each other indirectly through the **Mediator** object.
- Observer provides a way to keep objects in a one-to-many dependency loosely coupled by defining **Subject** and **Observer** objects that establish a flexible notification-registration mechanism to notify and update all registered observers.
- Observer and Mediator are competing patterns.  
"The difference between them is that Observer distributes communication by introducing **Observer** and **Subject** objects, whereas a Mediator object encapsulates the communication between other objects." [GoF, p346]

## Intent



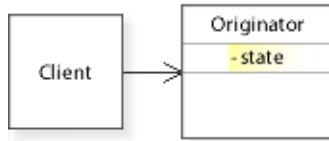
The intent of the Memento design pattern is:

**"Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later."** [GoF]

See Problem and Solution sections for a more structured description of the intent.

- The Memento design pattern solves problems like:
  - *Without violating encapsulation, how can an object's internal state be captured and externalized so that the object can be restored to this state later?*
- *Encapsulation* means hiding the representation of the state (the internal data structures like fields, variables, arrays, collections, etc. that store the state) and the implementation of the behavior in an object.
- *Internal state* of an object means all its internal data structures plus their values.
- The problem here is to save an object's internal state without making the representation of the state (the internal data structures) visible and accessible from outside the object.

## Problem



Sample  
Class  
Diagram

The Memento design pattern solves problems like:

**Without violating encapsulation,  
how can an object's internal state be captured and externalized  
so that the object can be restored to this state later?**

See Applicability section for all problems Memento can solve.

- A well-designed object is *encapsulated*.

That means, the representation of its state (the internal data structures that store the state) and the implementation of its behavior are hidden inside the object and cannot be accessed from outside the object (*Client*).

- In standard object-oriented languages, encapsulation is supported by specifying a *private* access level or visibility to protect against access from outside the object (in UML class diagrams shown as minus sign).
- *The problem here is to save the internal state of an object externally (to another object) without making the representation (the internal data structures) visible and accessible to other objects.*
- For example, designing checkpoints and undo mechanisms.  
It should be possible to save a snapshot of (some portion of) an object's internal state externally so that the object can be restored to this state later.  
A direct access to an object's internal data structures isn't possible because this would break its encapsulation.

## Background Information

- Encapsulation is

"The result of hiding a representation and implementation in an object." [GoF, p360]

Clients (from outside the object) can only see and access the (public) operations defined by an object's interface.

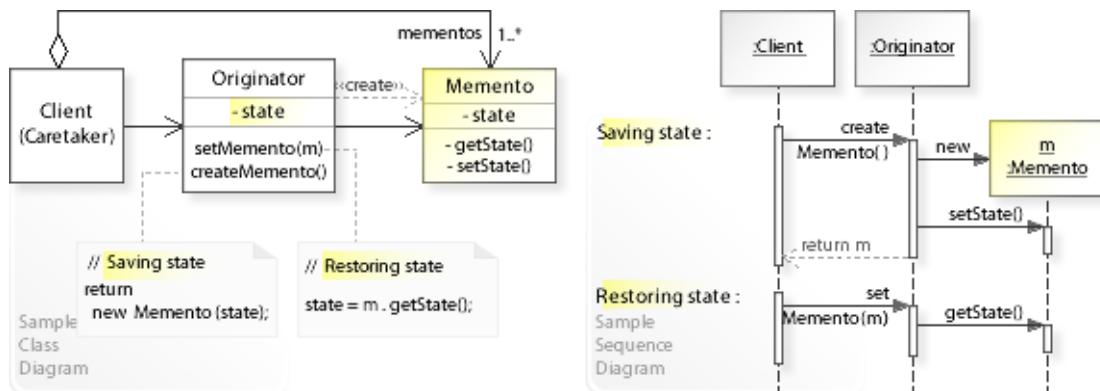
Clients do not see (are independent of) any changes of an object's (private/hidden) representation and implementation.

*This is the essential benefit of encapsulation.*

See also Design Principles.

- "At any given point in time, the state of an object encompasses all of the (usually static) properties of the object plus the current (usually dynamic) values of each of these properties." [GBooch07, p600]

## Solution



The Memento pattern describes a solution:

**Define separate `Memento` objects that store different snapshots of an object's (`Originator`) internal state.**

An `Originator` object creates a `Memento` object to save a snapshot of its current internal state, and it uses a `Memento` object to restore to a previous internal state.

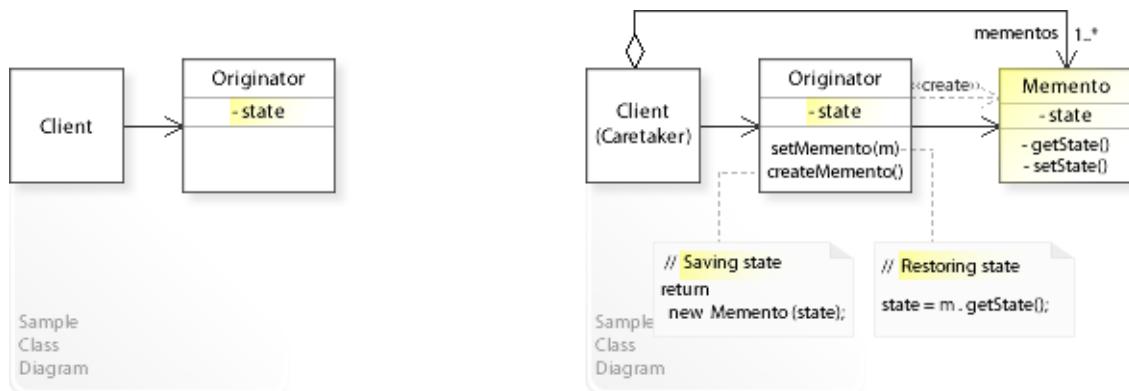
Describing the Memento design in more detail is the theme of the following sections.

See Applicability section for all problems Memento can solve.

- The key idea in this pattern is that only the originator that created a memento is permitted to access it.
  - Define separate `Memento` objects:**
    - Define the data structure to store a snapshot of an originator's internal state.
    - Protect the memento against access by objects other than the originator.

This is usually implemented by making the `Memento` class an *inner class* of originator and declaring all members of memento *private* (see Implementation and Sample Code).
  - Define an `Originator` object:**
    - `Originator` defines an interface for creating a `Memento` object to **save** a snapshot of its current internal state: `createMemento(): return new Memento(state)`
    - and for **restoring** to a previous internal state (by using a `Memento` object): `setMemento(m): state = m.getState();`
  - This enables to save and restore an `Originator` object's internal state without violating its encapsulation.
- Clients of originator (caretaker) are responsible for maintaining a container of mementos and for passing a memento back to the originator when restoring to a previous state, but clients aren't permitted to access mementos. Only the originator that created a memento is permitted to access it.

## Motivation 1



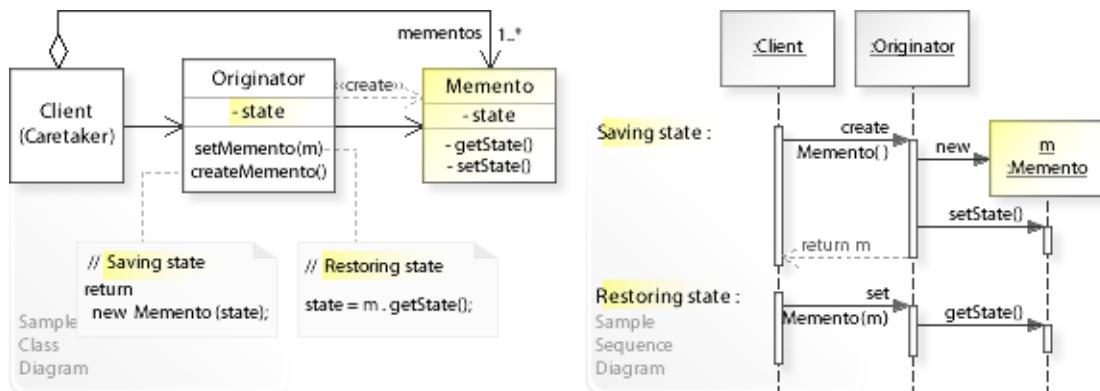
**Consider the left design (problem):**

- Originator's internal state can't be saved.
    - Clients can not save the originator's internal state because it is encapsulated (hidden inside the originator) and can not be accessed from outside the originator.

### **Consider the right design (solution):**

- Originator's internal state can be saved.
    - The originator creates a memento to save its internal state (`createMemento()`).
    - The originator uses a memento (`m`) to restore to a previous state (`setMemento(m)`).
    - Only the originator that created a memento is permitted to access the memento's internal state.
    - Clients (caretaker) are responsible for caretaking mementos, but they aren't permitted to access them.

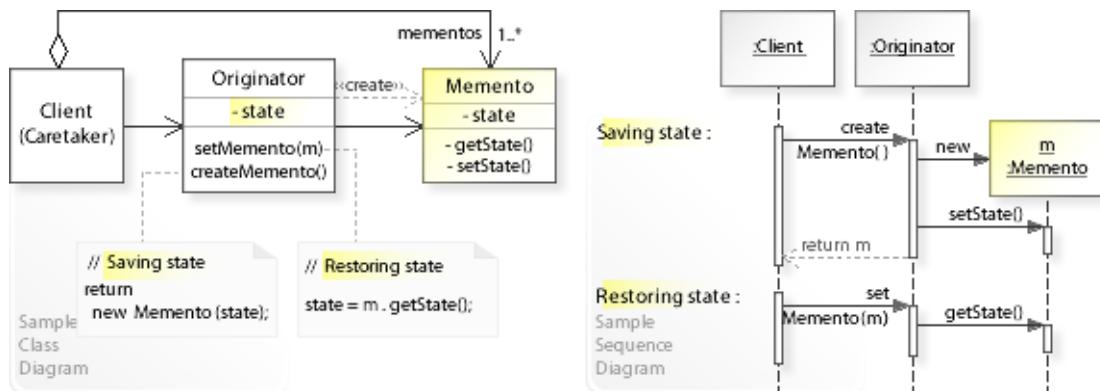
## Applicability



## Design Problems

- **Saving and Restoring an Object's Internal State**
  - Without violating encapsulation,  
how can an object's internal state be captured and externalized  
so that the object can be restored to this state later?

## Structure, Collaboration



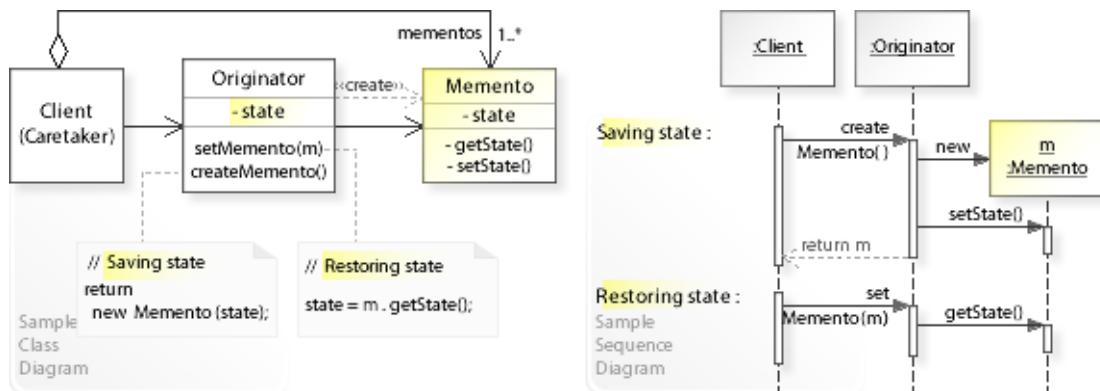
### Static Class Structure

- Client (Caretaker)
  - Maintains a container (list) of **Memento** objects (**mementos**).
  - Passes a **Memento** back to the **Originator** when restoring to a previous state.
  - Isn't permitted to access **Memento**.
- Originator
  - Creates a **Memento** object to save a snapshot of its current internal state (**createMemento()**).
  - Uses a **Memento** object to restore to a previous internal state (**setMemento(m)**).
- Memento
  - Stores a snapshot of the **Originator**'s internal state.
  - Defines a private interface that protects **Memento** against access by objects other than the **Originator**.
  - Only the **Originator** that created a **Memento** is permitted to access the **Memento**'s internal state.
  - See also Implementation and Sample Code.

### Dynamic Object Collaboration

- In this sample scenario, the internal state of an **Originator** object is saved and restored.
- To **save** the **originator**'s current internal state, the **Client (Caretaker)** calls **createMemento()** on the **Originator**.
- The **Originator** creates a new **Memento** object, saves a snapshot of its current internal state (**setState()**), and returns the **Memento** to the **Client**.
- The **Client** holds (takes care of) the returned **Memento** object(s).
- Thereafter, to **restore** the **Originator** to a previous state, the **Client** calls **setMemento(m)** on the **Originator** and provides the **Memento** object (**m**) to be used.
- The **Originator** restores its internal state from the provided **Memento** object (**getState()**) and returns to the **Client**.
- See also Sample Code / Example1.

## Consequences



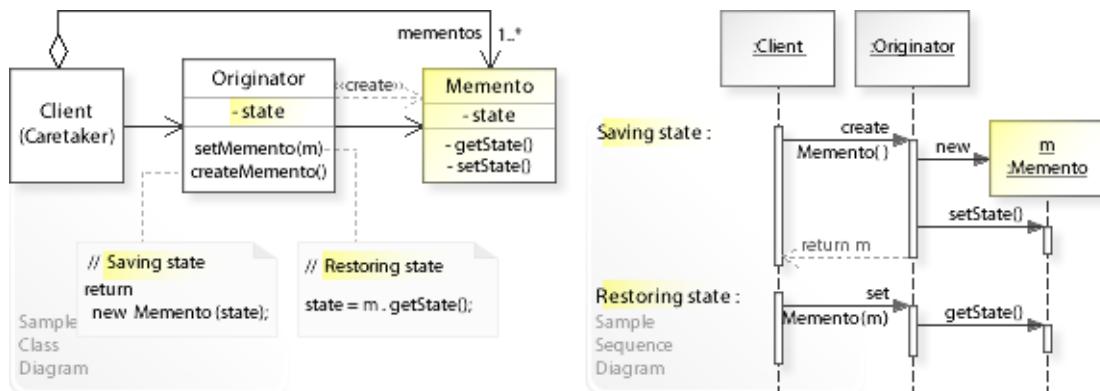
### Advantages (+)

- Preserves encapsulation.
  - An object's internal state can be saved externally (to another object) without violating encapsulation (without making the internal data structures accessible to other objects).
- Simplifies the originator.
  - Clients of originator (caretaker) are responsible to hold and maintain the different versions of the originator's internal state (mementos), and this simplifies the originator.

### Disadvantages (-)

- May introduce run-time costs.
  - Creating and restoring large numbers of mementos with large amounts of data may impact memory usage and system performance.
  - "Unless encapsulating and restoring Originator state is cheap, the pattern might not be appropriate." [GoF, p286]

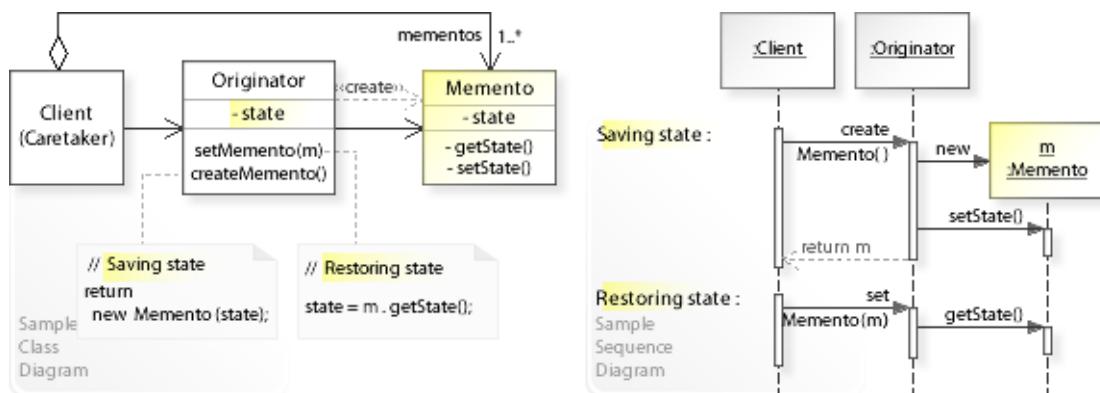
## Implementation



### Implementation Issues

- **Originator must have privileged access.**
  - Memento must be protected against access by objects other than originator.
  - This is usually implemented by making the **Memento** class an *inner class* of the **Originator** class and declaring all members of the **Memento** class *private*.
  - This enables originator to access the private data structures of memento.
  - Only the originator that created a memento can access memento's internal state.

## Sample Code 1



### Basic Java code for implementing the sample UML diagrams.

```

1 package com.sample.memento.basic;
2 import java.util.ArrayList;
3 import java.util.List;
4 public class Client {
5     // Running the Client class as application.
6     public static void main(String[] args) {
7         Originator originator = new Originator();
8         Originator.Memento m; // Memento inner class of Originator
9         // Container of memento objects.
10        List<Originator.Memento> mementos = new ArrayList<Originator.Memento>();
11        //
12        originator.setState("A");
13        System.out.println("Originator's first state : " + originator.getState());
14        //
15        // Saving first state.
16        m = originator.createMemento();
17        mementos.add(m); // add to container
18        //
19        originator.setState("B");
20        System.out.println("Originator's second state : " + originator.getState());
21        //
22        // Restoring first state.
23        m = mementos.get(0); // get first memento from container
24        originator.setMemento(m);
25        System.out.println("Originator restored to : " + originator.getState());
26    }
27 }

Originator's first state : A
Originator's second state : B
Originator restored to : A

```

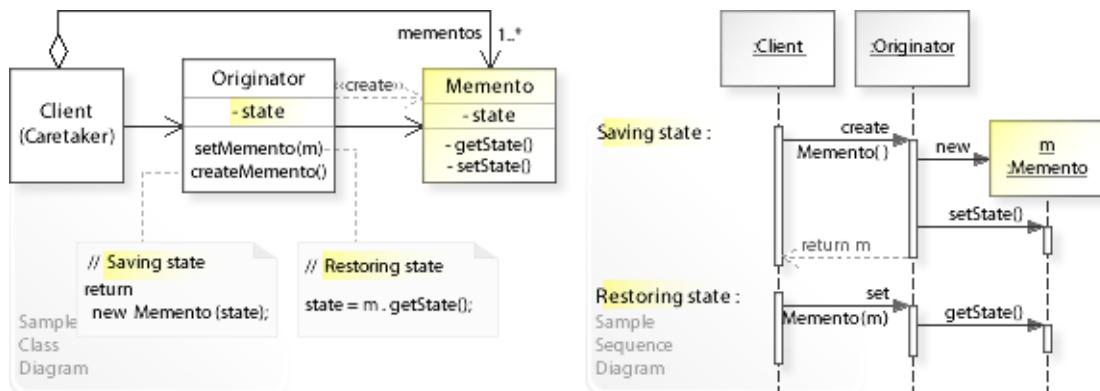
```

1 package com.sample.memento.basic;
2 public class Originator {
3     // Hiding internal state.
4     private String state;
5     // ...
6     // Saving internal state.
7     public Memento createMemento() {
8         Memento memento = new Memento();
9         memento.setState(state);
10        return memento;
11    }
12    // Restoring internal state.
13    void setMemento(Memento m) {
14        state = m.getState();
15    }
16    //
17    public String getState() {
18        return state;
19    }
20    void setState(String state) {
21        this.state = state;
22    }
}

```

```
23      //  
24      // Implementing Memento as inner class.  
25      // All members are private and accessible only to originator.  
26      //  
27      public class Memento {  
28          private String state;  
29          // ...  
30          private String getState() {  
31              return state;  
32          }  
33          private void setState(String state) {  
34              this.state = state;  
35          }  
36      }  
37  }
```

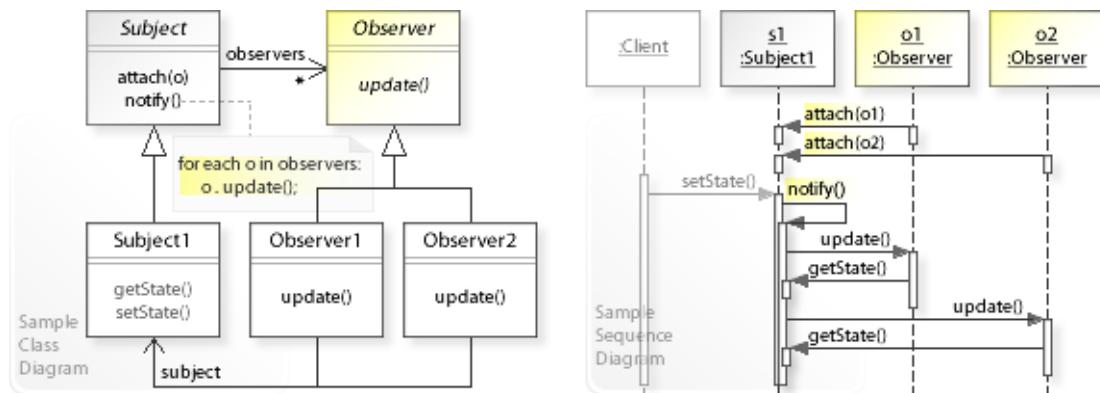
## Related Patterns



## Key Relationships (see also Overview)

- **Command - Memento**
  - Command and Memento often work together to support undoable operations.  
Memento stores state that command requires to undo its effects.

## Intent



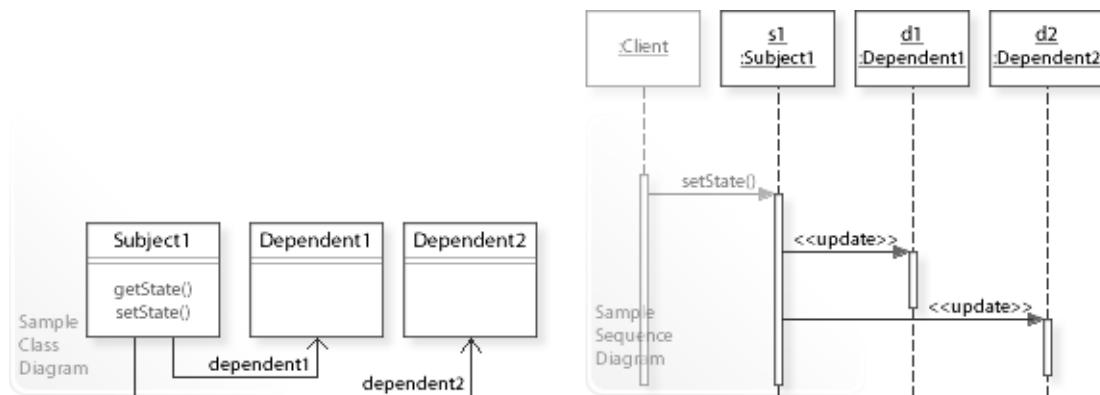
The intent of the Observer design pattern is to:

**"Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically."** [GoF]

See Problem and Solution sections for a more structured description of the intent.

- The Observer design pattern solves problems like:
  - *How can a one-to-many dependency between objects be defined without making the objects tightly coupled?*
  - *How can an object notify an open-ended-number of other objects?*
- *Coupling* is the degree to which things depend on each other.
  - *Tightly coupled objects* depend on (refer to and know about) many other objects and interfaces, which makes them hard to implement, change, and reuse.
  - *Loosely coupled objects* have only minimal dependencies on other objects and interfaces (most often achieved by working through a common interface), which makes them easier to implement, change, and reuse.
- The Observer pattern describes how to solve such problems:
  - *Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.*
  - The key idea in this pattern is to establish a flexible *notification-registration* mechanism that *notifies* all *registered* objects automatically when an event of interest occurs.

## Problem



The Observer design pattern solves problems like:

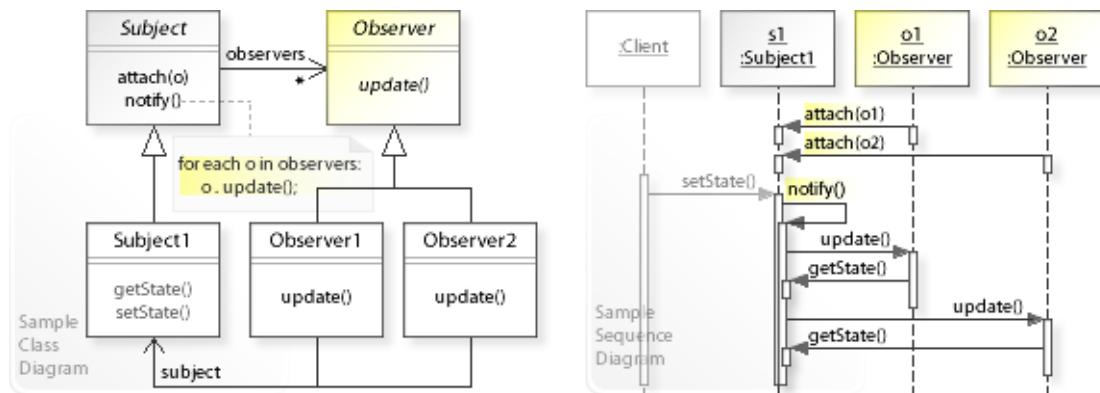
***How can a one-to-many dependency between objects be defined without making the objects tightly coupled?***

***How can an object notify an open-ended number of other objects?***

See Applicability section for all problems Observer can solve.

- An inflexible way to define a one-to-many dependency between objects is to define one object (Subject1) that refers to and knows about (how to update) all its dependent objects (Dependent1, Dependent2, ...). This couples the subject to many other objects having different interfaces.
- This makes the objects tightly coupled.  
Tightly coupled objects depend on (refer to and know about) many other objects and interfaces, which makes them hard to implement, change, test, and reuse.  
"You don't want to achieve consistency by making the classes tightly coupled, because that reduces their reusability." [GoF, p293]
- *That's the kind of approach to avoid if we want to keep the objects loosely coupled.*
- For example, defining a data object and multiple presentation objects in a GUI/Web application.  
When the data object changes state, all presentation objects that depend on the data object's state should be updated (synchronized) automatically and immediately to reflect the data change (see Sample Code / Example 2).
- For example, event handling in a GUI/Web application.  
When a user clicks a button, all objects that depend on (listen for) the button's events should be notified that a 'mouse click' event occurred (see Sample Code / Example 3/4).

## Solution



The Observer pattern describes a solution:

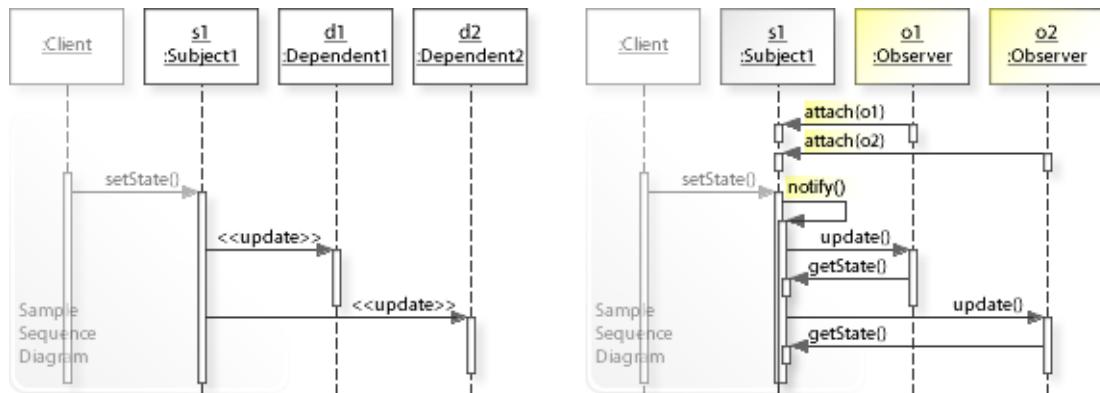
**Define Subject and Observer objects so that when Subject changes state, all registered observers are notified and updated automatically.**

Describing the Observer design in more detail is the theme of the following sections.

See Applicability section for all problems Observer can solve.

- The key idea in this pattern is to establish a flexible *notification-registration* mechanism that *notifies* all *registered* observers automatically when subject changes state.
- Define Subject and Observer objects:**
  - Subject defines an interface for registering/unregistering Observer objects (`attach(o)` / `detach(o)`). (To simplify UML diagrams, `detach(o)` isn't shown.)
  - Observer defines an interface for updating state (`update()`), i.e., synchronizing Observer's state with Subject's state.
- When Subject changes state, all registered observers are notified and updated automatically** (for each `o` in `observers`: `o.update()`).
- This enables loose coupling between subject and observers. Subject and observers have no explicit knowledge of each other. Any number of observers can be added and removed independently and dynamically. "This kind of interaction is also known as **publish-subscribe**. The subject is the publisher of notifications. It sends out these notifications without having to know who its observers are. Any number of observers can subscribe to receive notifications." [GoF, p294]

## Motivation 1



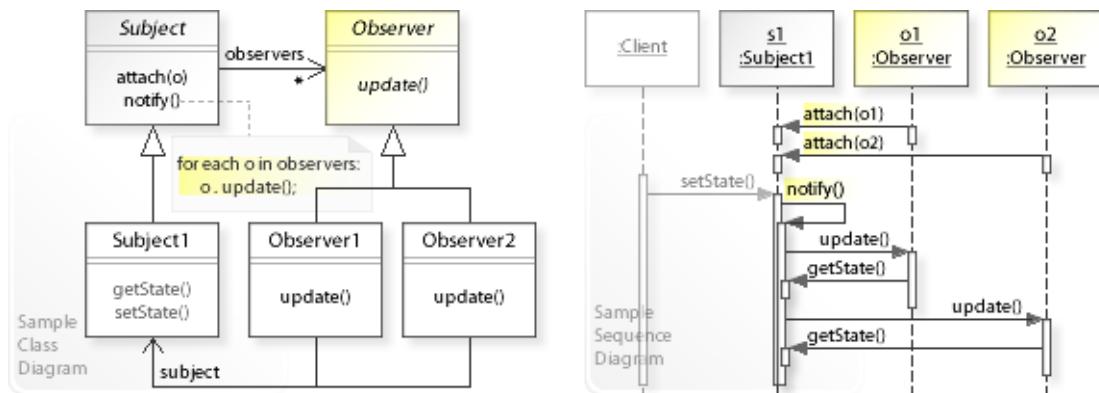
Consider the left design (problem):

- Tight coupling between subject and dependents.
  - Subject refers to and knows about (how to update) all its dependent objects with their specific interfaces (tight coupling).
  - Adding new dependent objects or remove existing ones requires changing subject.
  - Tightly coupled objects depend on (refer to and know about) many other objects with different interfaces, which makes them hard to implement, change, and reuse.

Consider the right design (solution):

- Loose coupling between subject and observers.
  - Subject only refers to and knows about the common Observer interface for updating state (loose coupling).
  - Subject provides an interface for registering or unregistering any number of observers (**attach(o)** / **detach(o)**).
  - Loosely coupled objects have only minimal dependencies (by working through a common interface), which makes them easier to implement, change, and reuse.

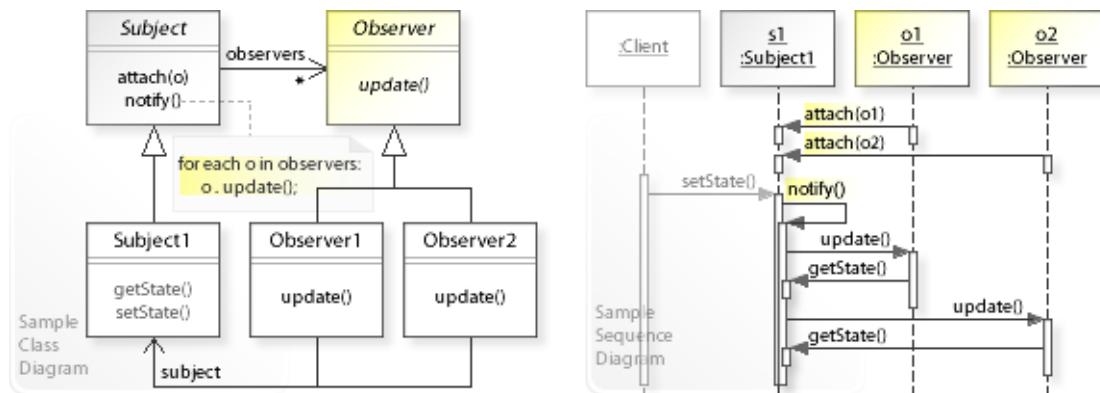
## Applicability



## Design Problems

- **Defining One-To-Many Dependencies**
  - How can a one-to-many dependency between objects be defined without making the objects tightly coupled?
  - How can be ensured that when one object changes state an open-ended number of dependent objects are updated automatically?
  - How can consistency between related objects be maintained efficiently without making the objects tightly coupled?
- **Flexible Notification-Registration Mechanism**
  - How can an object notify an open-ended number of other objects without making the objects tightly coupled?

## Structure, Collaboration



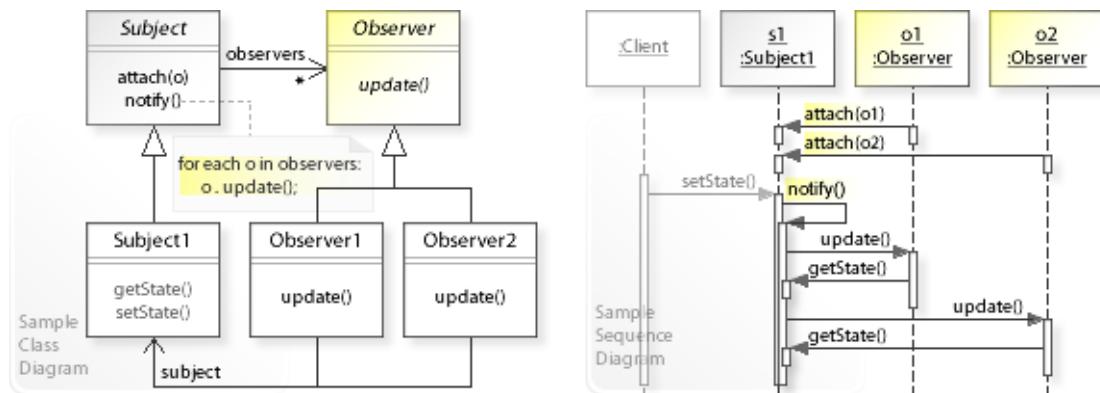
## Static Class Structure

- **Subject**
  - Refers to the `Observer` interface and is independent of the concrete class of any observer.
  - Stores a list of `Observer` objects (`observers`).
  - Defines an interface for registering `Observer` objects (`attach(o)`).
  - To simplify UML diagrams, unregistering `Observer` objects (`detach(o)`) is omitted.
  - Defines an interface for notifying (calling the `Observer` interface for) registered observers (`notify()`).
  - Usually, a subject calls `notify()` on itself when its state changes.
- **Subject1**
  - Stores state observers depend on (`getState()`, `setState()`).
- **Observer**
  - Defines an interface for updating its state, i.e., synchronizing its state with `Subject`'s state (`update()`).
- **Observer1**, **Observer2**, ...
  - Implement the `Observer` interface.
  - Maintain a reference (`subject`) to a `Subject` object.

## Dynamic Object Collaboration

- In this sample scenario, the `Observer` objects `o1` and `o2` register themselves on the `Subject1` object and are subsequently notified and updated (synchronized) when `Subject1` changes state.
- The interaction starts with the `Observer` objects `o1` and `o2` that call `attach(this)` on `Subject1` to register themselves.
- Thereafter, a `Client` object changes the state of `Subject1`, which causes `Subject1` to call `notify()` on itself.
- `notify()` calls `update()` on the registered `Observer` objects `o1` and `o2`, which in turn get the changed data (`getState()`) from `Subject1` and `update` (synchronize) their state.
- There are different ways to exchange (push/pull) data between `Subject` and `Observer` objects (see Implementation).
- See also Sample Code / Example 1.

## Consequences



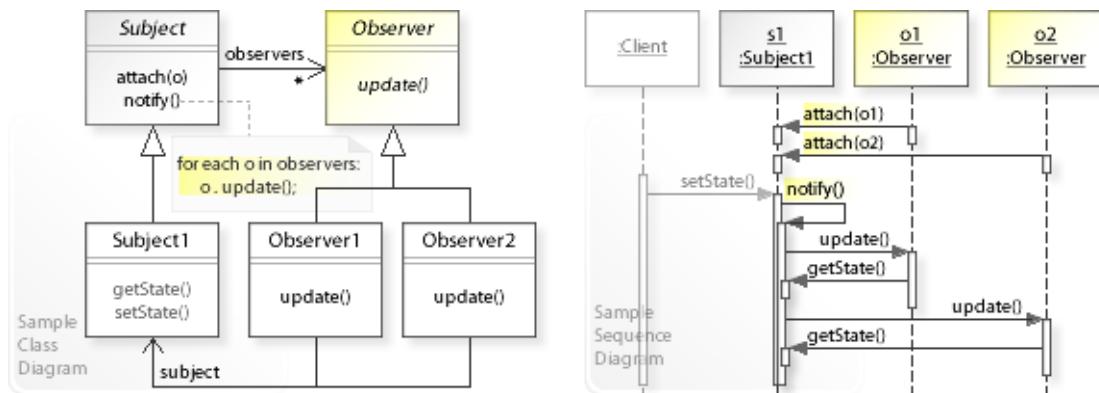
### Advantages (+)

- Decouples subject from observers.
  - Subject only refers to and knows about the common **Observer** interface for updating state (`update()`).
  - "Because Subject and Observer aren't tightly coupled, they can belong to different layers of abstraction in a system. [GoF, p296]
  - Loosely coupled objects are easier to implement, change, and reuse.
- Makes adding/withdrawing observers easy.
  - Observers can be added to (`attach(o)`) and withdrawn from a subject independently and dynamically.
  - Usually, observers are responsible for registering and unregistering themselves on a subject.
  - Subject's sole responsibility is to hold a list of observers and notify (call `update()` on) them when its state changes.

### Disadvantages (-)

- Can make the update behavior complex.
  - A change on the subject may cause a cascade of updates to observers and their dependent objects.
  - The Mediator design pattern can be applied to implement a complex dependency relationship between subject(s) and observers.

## Implementation



## Implementation Issues

- **Implementation Variants**

- The **Subject** and **Observer** interfaces must be designed carefully so that the data can be passed/accessed efficiently to let observers know what changed in subject. There are two main variants:

- **Variant1: Push Data**

- Subject passes the changed data to its observers:
- ```
update(data1, data2, ...)
```
- The **Observer** interface may get complex because it must enable to pass in the changed data for all supported observers (whether the data is simple or complex).

- **Variant2: Pull Data**

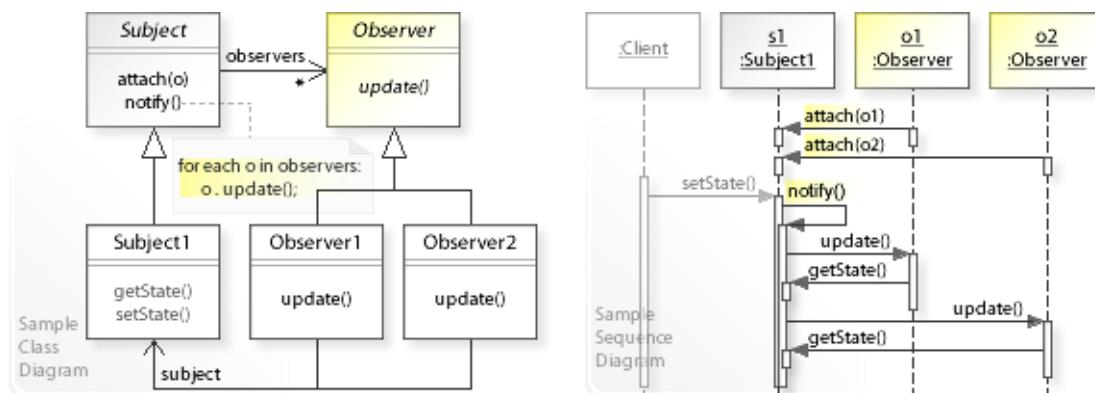
- Subject passes nothing but itself to its observers so that they can call back to get (pull) the required data from subject:

```
update(this)
```

- The **Subject** interface may get complex because it must enable all supported observers to access the needed data.

- "The pull model emphasizes the subject's ignorance of its observers, whereas the push model assumes subjects know something about their observers' needs." [GoF, p298]

## Sample Code 1



## Basic Java code for implementing the sample UML diagrams.

```

1 package com.sample.observer.basic;
2 public class Client {
3     // Running the Client class as application.
4     public static void main(String[] args) {
5         Subject1 s1 = new Subject1();
6         // Creating observers and registering them on subject1.
7         Observer o1 = new Observer1(s1);
8         Observer o2 = new Observer2(s1);
9         //
10        System.out.println("Client : Changing state of Subject1 ...");
11        s1.setState(100);
12    }
13 }

Client : Changing state of Subject1 ...
Subject1 : State changed to : 100
          Notifying observers ...
Observer1: State updated to : 100
Observer2: State updated to : 100

1 package com.sample.observer.basic;
2 import java.util.ArrayList;
3 import java.util.List;
4 public abstract class Subject {
5     private List<Observer> observers = new ArrayList<Observer>();
6     // Registration interface.
7     public void attach(Observer o) {
8         observers.add(o);
9     }
10    // Notification interface.
11    // notify() is already used by the Java Language (to wake up threads).
12    public void notifyObservers() {
13        for (Observer o : observers)
14            o.update();
15    }
16 }

1 package com.sample.observer.basic;
2 public class Subject1 extends Subject {
3     private int state = 0;
4     //
5     public int getState() {
6         return state;
7     }
8     void setState(int state) {
9         this.state = state;
10        System.out.println(
11            "Subject1 : State changed to : " + state +
12            "\n          Notifying observers ...");
13        // Notifying observers that state has changed.
14        notifyObservers();
15    }
16 }

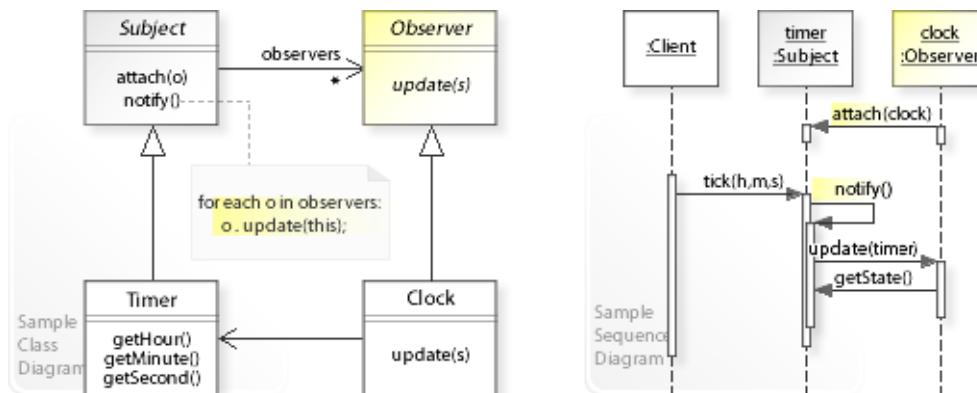
```

```
1 package com.sample.observer.basic;
2 public abstract class Observer {
3     // Synchronizing observer's state with subject's state.
4     public abstract void update();
5 }

1 package com.sample.observer.basic;
2 public class Observer1 extends Observer {
3     private int state;
4     private Subject1 subject;
5     public Observer1(Subject1 subject) {
6         this.subject = subject;
7         // Registering this observer on subject.
8         subject.attach(this);
9     }
10    public void update() {
11        this.state = subject.getState();
12        System.out.println(
13            "Observer1: State updated to : " + this.state);
14    }
15 }

1 package com.sample.observer.basic;
2 public class Observer2 extends Observer {
3     private int state;
4     private Subject1 subject;
5     public Observer2(Subject1 subject) {
6         this.subject = subject;
7         // Registering this observer on subject.
8         subject.attach(this);
9     }
10    public void update() {
11        this.state = subject.getState();
12        System.out.println(
13            "Observer2: State updated to : " + this.state);
14    }
15 }
```

## Sample Code 2



Synchronizing state between a timer object (time of day) and a clock object.

```

1 package com.sample.observer.timer;
2 import java.util.Calendar;
3 public class Client {
4     public static void main(String[] args) throws InterruptedException {
5         Timer timer = new Timer(); // subject
6         // Creating a clock (observer) and registering it on timer (subject).
7         Clock clock = new Clock(timer);
8         final Calendar calendar = Calendar.getInstance();
9         for (int i = 0; i < 3; i++) {
10             Thread.sleep(1000); // one second
11             calendar.setTimeInMillis(System.currentTimeMillis());
12             int h = calendar.get(Calendar.HOUR_OF_DAY);
13             int m = calendar.get(Calendar.MINUTE);
14             int s = calendar.get(Calendar.SECOND);
15             // Changing timer's state every second.
16             timer.tick(h, m, s);
17         }
18     }
19 }

Timer : Time of day changed to : 15:38:02
Clock : Updated/Synchronized to : 15:38:02
Timer : Time of day changed to : 15:38:03
Clock : Updated/Synchronized to : 15:38:03
Timer : Time of day changed to : 15:38:04
Clock : Updated/Synchronized to : 15:38:04
  
```

```

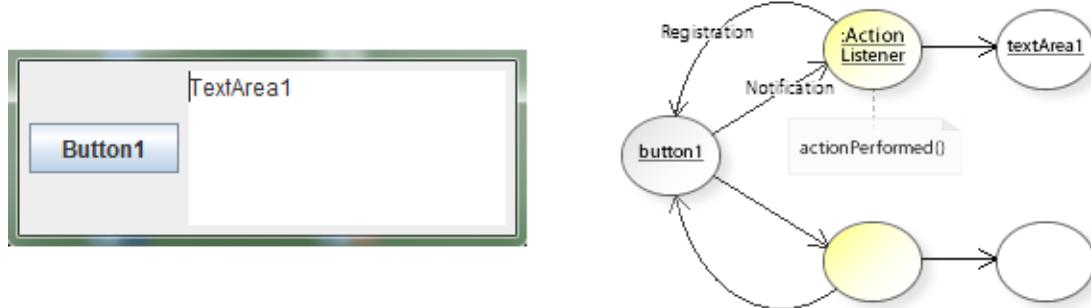
1 package com.sample.observer.timer;
2 import java.util.ArrayList;
3 import java.util.List;
4 public abstract class Subject {
5     private List<Observer> observers = new ArrayList<Observer>();
6     // Registration interface.
7     public void attach(Observer o) {
8         observers.add(o);
9     }
10    // Notification interface.
11    public void notifyObservers() {
12        for (Observer o : observers)
13            o.update(this);
14    }
15 }
  
```

```
1 package com.sample.observer.timer;
2 public class Timer extends Subject {
3     private int hour = 0;
4     private int minute = 0;
5     private int second = 0;
6     public int getHour() {
7         return hour;
8     }
9     public int getMinute() {
10        return minute;
11    }
12    public int getSecond() {
13        return second;
14    }
15    // Changing time of day and notifying observers.
16    public void tick(int hour, int minute, int second) {
17        System.out.printf(
18            "Timer : Time of day changed to : %02d:%02d:%02d %n",
19            hour, minute, second);
20        this.hour = hour;
21        this.minute = minute;
22        this.second = second;
23        // Notifying observers that time has changed.
24        notifyObservers();
25    }
26 }

1 package com.sample.observer.timer;
2 public abstract class Observer {
3     public abstract void update(Subject s);
4 }

1 package com.sample.observer.timer;
2 public class Clock extends Observer {
3     private Timer subject;
4     public Clock(Timer subject) {
5         this.subject = subject;
6         // Registering this clock on subject.
7         subject.attach(this);
8     }
9     public void update(Subject s) {
10        if (this.subject == s) {
11            System.out.printf(
12                "Clock : Updated/Synchronized to : %02d:%02d:%02d %n",
13                subject.getHour(), subject.getMinute(), subject.getSecond());
14        }
15    }
16 }
```

### Sample Code 3



#### Event handling in a GUI application (Java Swing).

The **ActionListener** interface has a single operation: **actionPerformed(event)**.

This is the action to be performed when an action event occurs.

This example shows registering an event (=action) listener for a mouse click on a button:

When clicking the `Button1`, a message is shown in the `TextArea1`.

There are different variants to implement the interface:

```
*****
Variant1: Implementing the ActionListener interface with inner classes.
*****
```

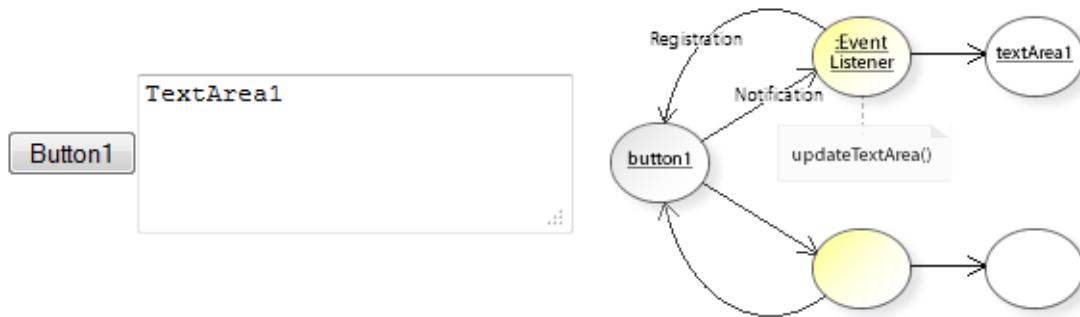
```

1 package com.sample.observer.gui;
2 import java.awt.event.*;
3 import javax.swing.*;
4 public class GUIDemo1 extends JPanel {
5     JButton button1;
6     JTextArea textArea1;
7     public GUIDemo1() {
8         button1 = new JButton("Button1");
9         add(button1);
10        textArea1 = new JTextArea("TextArea1", 5, 15);
11        add(textArea1);
12        // Creating an ActionListener object and registering it on button1
13        // (for being notified when an action event occurs).
14        button1.addActionListener(new ActionListener() {
15            // Anonymous inner class.
16            // Implementing the ActionListener interface.
17            public void actionPerformed(ActionEvent e) {
18                textArea1.append("\nNotification from Button1:\n " +
19                    "User clicked the Button1.");
20            }
21        });
22    }
23    private static void createAndShowGUI() {
24        // Creating the GUI.
25        JFrame frame = new JFrame("GUIDemo1");
26        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
27        JComponent contentPane = new GUIDemo1();
28        frame.setContentPane(contentPane);
29        // Showing the GUI.
30        frame.pack();
31        frame.setVisible(true);
32    }
33    public static void main(String[] args) {
34        // For thread safety, invoked from event-dispatching thread.
35        javax.swing.SwingUtilities.invokeLater(new Runnable() {
36            public void run() {
37                createAndShowGUI();
38            }
39        });
40    }
41 }
```

```
*****
Variant2: Implementing without inner classes.
*****
```

```
1 package com.sample.observer.gui;
2 import java.awt.event.*;
3 import javax.swing.*;
4 public class GUIDemo2 extends JPanel
5     implements ActionListener {
6     JButton button1;
7     JTextArea textArea1;
8     public GUIDemo2() {
9         button1 = new JButton("Button1");
10        add(button1);
11        textArea1 = new JTextArea("TextArea1", 10, 20);
12        add(textArea1);
13        // Registering this object/ActionListener on button1
14        // (for being notified when an action event occurs).
15        button1.addActionListener(this);
16    }
17    // Implementing the ActionListener interface.
18    public void actionPerformed(ActionEvent e) {
19        if (e.getSource() == button1) {
20            textArea1.append("\nNotification from Button1: \n" +
21                "User clicked the Button1.");
22        }
23    }
24    private static void createAndShowGUI() {
25        // Creating the GUI.
26        JFrame frame = new JFrame("GUIDemo2");
27        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
28        JComponent contentPane = new GUIDemo2();
29        frame.setContentPane(contentPane);
30        // Showing the GUI.
31        frame.pack();
32        frame.setVisible(true);
33    }
34    public static void main(String[] args) {
35        // For thread safety, invoked from event-dispatching thread.
36        javax.swing.SwingUtilities.invokeLater(new Runnable() {
37            public void run() {
38                createAndShowGUI();
39            }
40        } );
41    }
42 }
```

## Sample Code 4



### Event handling in a HTML document (DOM / JavaScript).

The W3C Document Object Model (DOM) is a standard interface for accessing and updating HTML and XML documents. It is separated into 3 different parts: Core DOM, XML DOM, and HTML DOM.

The HTML DOM represents a HTML document as a tree (=composite) structure of objects (nodes).

Everything found in a HTML document can be accessed and changed dynamically.

The DOM Event Model allows registering event listeners (=observers) on DOM element nodes.

This example shows registering an event listener for a mouse click on a button:

When clicking the `Button1`, a message is shown in the `TextArea1`.

There are three variants for registering event listeners:

```
*****
Variant1: Hard-wiring event handling directly into the HTML code:
<button onclick="function() {...};"></button>
*****
```

```

1  <!DOCTYPE html>
2  <html>
3  <head>
4      <script>
5          function updateTextArea() {
6              var buttonNode = document.getElementById("button1");
7              buttonNode.innerHTML += "\nNotification from Button1: \n" +
8                  "User clicked the Button1.";
9          }
10     </script>
11 </head>
12 <body>
13     <button id="button1" onclick="updateTextArea();">Button1</button>
14 <!-- ===== -->
15     <textarea id="textArea1" rows="4" cols="25">TextArea1</textarea>
16 </body>
17 </html>
```

```
*****
Variant2: Separating event handling (JavaScript code) from HTML code:
buttonNode.addEventListener('click', function() {...}, false);
This is the most flexible way.
Note that Internet Explorer 6-8 didn't support the DOM standard.
*****
```

```

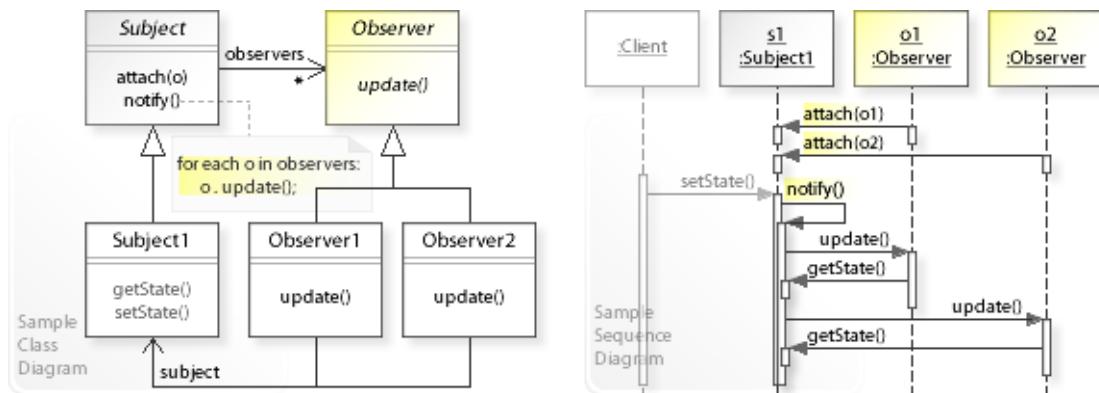
1  <!DOCTYPE html>
2  <html>
3  <head>
4      <script type="text/javascript">
5          function dynamicRegistration() {
6              var buttonNode = document.getElementById('button1');
```

```
7         if (buttonNode.addEventListener) {
8             buttonNode.addEventListener('click', updateTextArea, false);
9         <!-- ===== -->
10        } else { // Internet Explorer 6-8
11            buttonNode.attachEvent('onclick', updateTextArea);
12        }
13    }
14    function updateTextArea() {
15        var textNode = document.getElementById("textArea1");
16        textNode.innerHTML += "\nNotification from Button1: \n " +
17                            "User clicked the Button1.";
18    }
19    </script>
20 </head>
21
22 <body onload="dynamicRegistration();">
23     <button id="button1">Button1</button>
24     <textarea id="textArea1" rows="4" cols="25">TextArea1</textarea>
25 </body>
26 </html>
```

\*\*\*\*\*  
Variant3: Separating event handling (JavaScript code) from HTML code:  
**buttonNode.onclick = function(){};**  
This way, only one handler can be set per event and element.  
\*\*\*\*\*

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4      <script type="text/javascript">
5          function dynamicRegistration() {
6              var buttonNode = document.getElementById('button1');
7              buttonNode.onclick = function() {
8                  <!-- ===== -->
9                  var textNode = document.getElementById("textArea1");
10                 textNode.innerHTML += "\nNotification from Button1: \n " +
11                               "User clicked the Button1.";
12             }
13         }
14     </script>
15 </head>
16
17 <body onload="dynamicRegistration();">
18     <button id="button1">Button1</button>
19     <textarea id="textArea1" rows="4" cols="25">TextArea1</textarea>
20 </body>
21 </html>
```

## Related Patterns

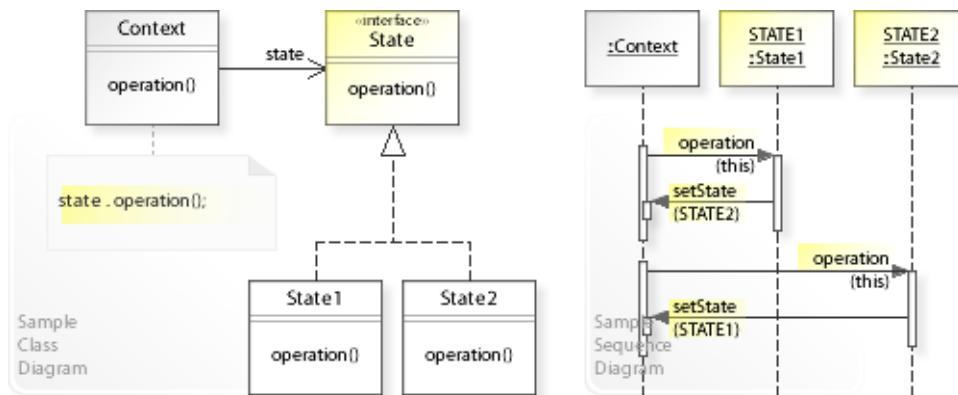


### Key Relationships (see also Overview)

- **Mediator - Observer**

- Mediator provides a way to keep a set of interacting objects loosely coupled by defining a `Mediator` object that controls and coordinates the interaction.  
Colleagues interact with each other indirectly through the `Mediator` object.
- Observer provides a way to keep objects in a one-to-many dependency loosely coupled by defining `Subject` and `Observer` objects that establish a flexible notification-registration mechanism to notify and update all registered observers.
- Observer and Mediator are competing patterns.  
"The difference between them is that Observer distributes communication by introducing `Observer` and `Subject` objects, whereas a Mediator object encapsulates the communication between other objects." [GoF, p346]

## Intent



The intent of the State design pattern is to:

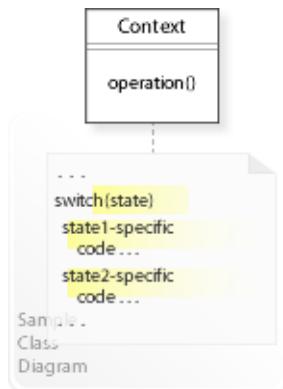
**"Allow an object to alter its behavior when its internal state changes."**

**The object will appear to change its class."** [GoF]

See Problem and Solution sections for a more structured description of the intent.

- The State design pattern solves problems like:
  - *How can a state-specific behavior be defined so that new states can be added and the behavior of existing states can be changed independently?*
  - <listitem>
 *How can an object alter its behavior when its internal state changes?*
</listitem>
- For example, a sales order object in an order processing system.  
A sales order object can be in one of different states. When it receives a request, it behaves differently depending on its current internal state.  
It should be possible to add new states and change the behavior of existing states independently from (without having to change) the sales order classes.

## Problem



The State design pattern solves problems like:

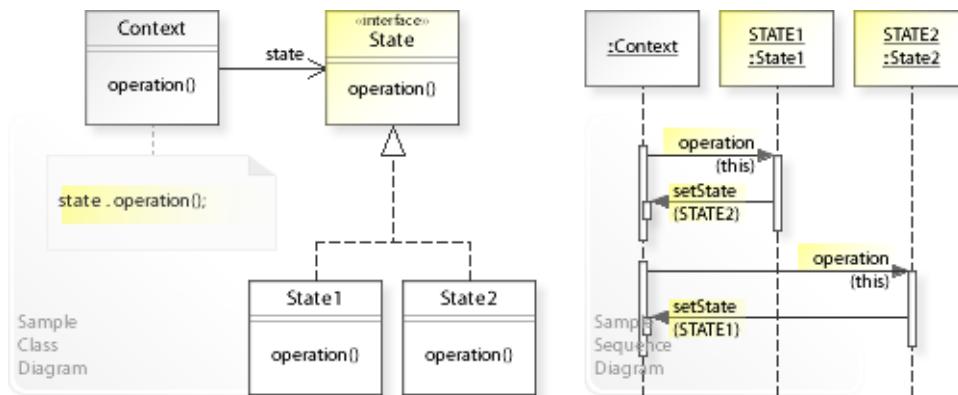
***How can a state-specific behavior be defined so that new states can be added and the behavior of existing states can be changed independently?***

***How can an object alter its behavior when its internal state changes?***

See Applicability section for all problems State can solve.

- An inflexible way to define a state-specific behavior is to implement it directly within a class (Context). Conditional statements (`switch(state)`) are required to switch between different states. Each conditional branch implements the corresponding state-specific behavior.  
" [...] we'd have look-alike conditional or case statements scattered throughout Context's implementation. Adding a new state could require changing several operations, which complicates maintenance." [GoF, p307]
- That's the kind of approach to avoid if we want that new states can be added and the behavior of existing states can be changed easily without touching existing classes.
- For example, a sales order object in an order processing system.  
A sales order object can be in one of different states. When it receives a request, it behaves differently depending on its current internal state.  
It should be possible to add new states and change the behavior of existing states independently from (without having to change) the sales order classes.

## Solution



The State pattern describes a solution:

**Define separate state objects that encapsulate the state-specific behavior of different states.**

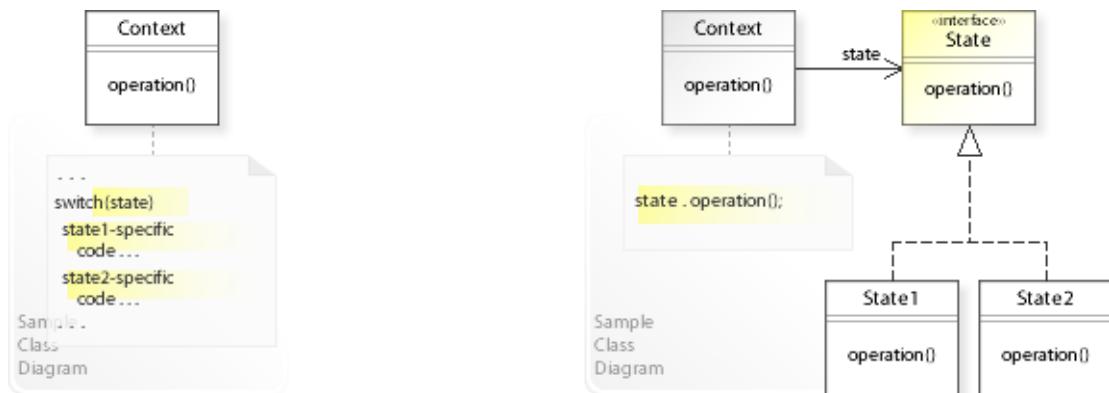
**Delegate state-specific behavior to the current state object.**

Describing the State design in more detail is the theme of the following sections.

See Applicability section for all problems State can solve.

- The key idea in this pattern is to encapsulate the state-specific behavior of a state in a separate object. "This lets you treat the object's state as an object in its own right that can vary independently from other objects." [GoF, p306]
- **Define separate state objects:**
  - For all possible states, define a common interface for performing state-specific behavior (**State** | **operation(...)**).
  - Define classes (**State1**, **State2**,...) that implement the **State** interface.
- This enables *compile-time* flexibility (via inheritance). "Because all state-specific code lives in a State subclass, new states and transitions can be added easily by defining new subclasses." [GoF, p307]
- **Delegate (the responsibility for performing) state-specific behavior to the current state object** (**state.operation(...)**).
  - This enables *run-time* flexibility (via object composition). An object (**Context**) can change its behavior by changing its current **State** object dynamically. Usually, the **State** objects are responsible to change **Context**'s current state at run-time when a state transition occurs (see also Collaboration and Sample Code).

## Motivation 1



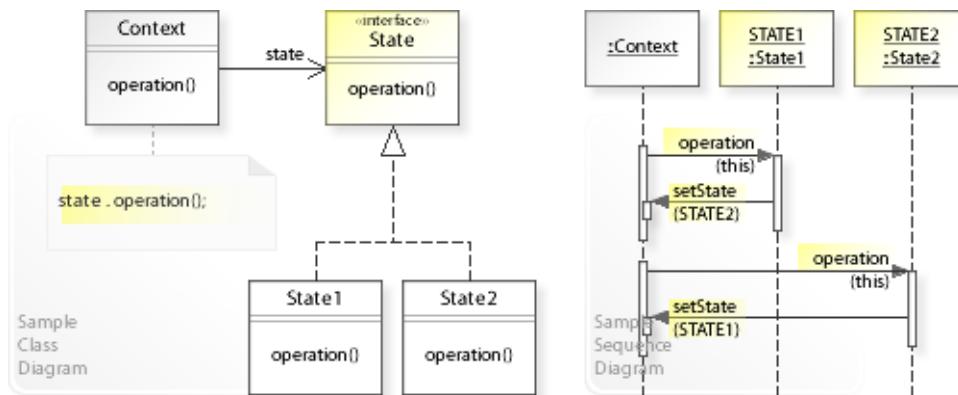
**Consider the left design (problem):**

- Hard-wired state-specific behavior.
  - The object's state-specific behavior is implemented (hard-wired) directly within a class (`Context`).
  - This makes it hard to add new states or change the behavior of existing states independently from (without having to change) the `Context` class.
- Conditional statements required.
  - Conditional statements are needed to switch between different states.
- Complicated class.
  - A class that includes state-specific behavior gets more complex and harder to implement, change, test, and reuse.

**Consider the right design (solution):**

- Encapsulated state-specific behavior.
  - The state-specific behavior of each state is implemented (encapsulated) in a separate class (`State1`, `State2`, ...).
  - This makes it easy to add new states or change the behavior of existing states independently from (without having to change) the `Context` class.
- No conditional statements required.
  - Conditional statements are replaced by delegating to different `State` objects.
- Simplified class.
  - A class that delegates state-specific behavior gets less complex and easier to implement, change, test, and reuse.

## Applicability



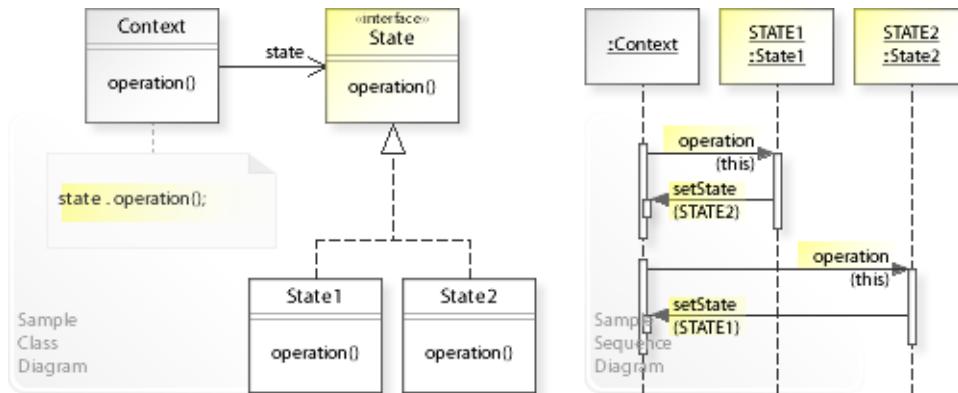
## Design Problems

- **Defining State-Specific Behavior**
  - How can a state-specific behavior be defined so that new states can be added and the behavior of existing states can be changed independently?
- **Dynamic State Transition**
  - How can an object alter its behavior when its internal state changes?

## Refactoring Problems

- **Complicated Code**
  - How can multiple conditional statements that switch between different states be eliminated?  
*Replace State-Altering Conditionals with State (166) [JKerievsky05]*

## Structure, Collaboration



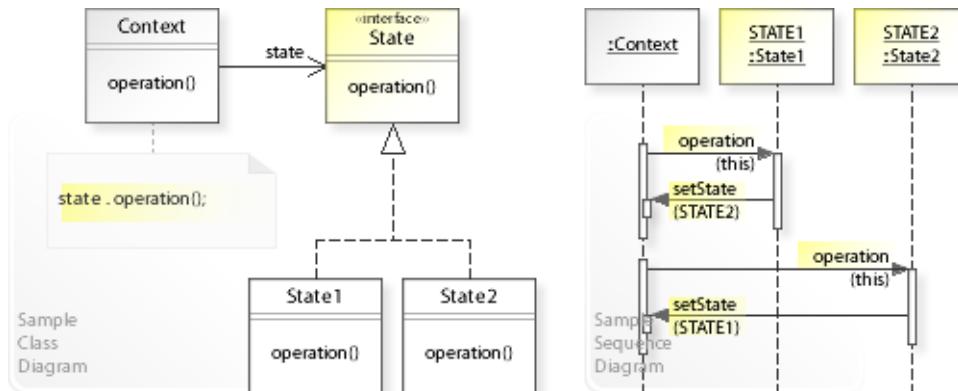
### Static Class Structure

- **Context**
  - Refers to the **State** interface to perform state-specific behavior (`state.operation()`) and is independent of how the behavior is implemented.
  - Maintains a reference (`state`) to its current **State** object.
- **State**
  - For all possible states, defines a common interface for performing state-specific behavior.
- **State1, State2, ...**
  - Implement the **State** interface.

### Dynamic Object Collaboration

- In this sample scenario, a **Context** object delegates state-specific behavior to its current **State** object.  
Let's assume that **Context** is configured with an (initial) **STATE1** object.
- The interaction starts with a **Client** object that calls **operation()** on **Context**.
- **Context** delegates state-specific behavior to its current **STATE1** object by calling **operation(this)** on it.
- **Context** passes itself (**this**) to **STATE1** so that **STATE1** can call back and change **Context**'s current state.
- **STATE1** performs the **state1**-specific operation and, assuming a state transition occurs, changes **Context**'s current state by calling **setState(STATE2)** on it.
- Thereafter, the **Client** again calls **operation()** on **Context**, which now delegates state-specific behavior to the current **STATE2** object.
- **STATE2** performs the **state2**-specific operation and calls **setState(STATE1)** on **Context**.
- See also Sample Code / Example 1.

## Consequences



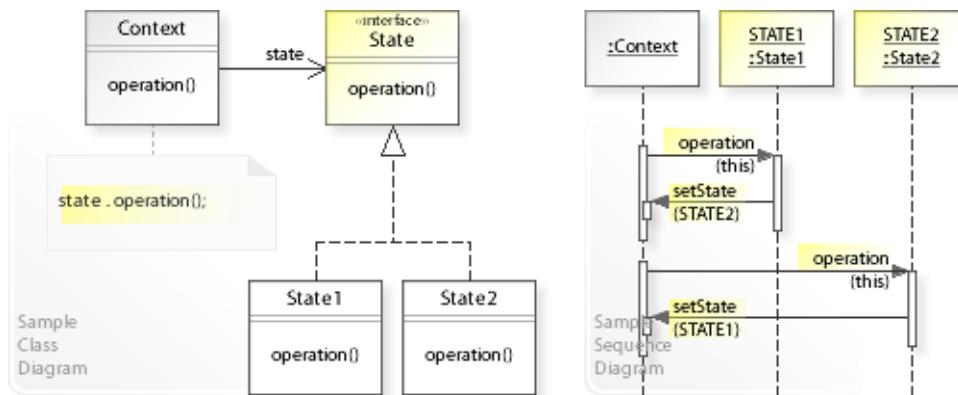
### Advantages (+)

- Makes adding new states easy.
  - "Because all state-specific code lives in a `State` subclass, new states and transitions can be added easily by defining new subclasses." [GoF, p307]
- Avoids conditional statements.
  - Instead of hard-coding multiple/large conditional statements that switch between the different states, `Context` delegates state-specific behavior to its current `State` object.
  - "That imposes structure on the code and makes its intent clearer." [GoF, p307]
- Ensures consistent states.
  - `Context`'s state is changed by replacing its current `State` object. This can avoid inconsistent internal states.
- Makes state transitions explicit.
  - "Introducing separate objects for different states makes the transitions more explicit." [GoF, p307]

### Disadvantages (-)

- May require extending the `Context` interface.
  - The `Context` interface may have to be extended to let `State` objects access `Context`'s state.
- Introduces an additional level of indirection.
  - `State` achieves flexibility by introducing an additional level of indirection (delegating to separate `State` objects).

## Implementation

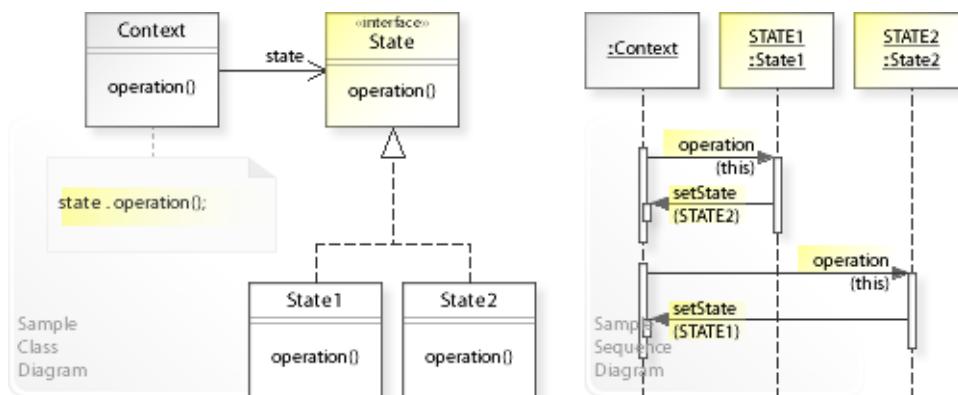


### Implementation Issues

- **State Transitions**

- Usually, the **State** objects are responsible to change **Context**'s current state dynamically when a state transition occurs.
- **Context** doesn't know anything about its state.  
The **state** objects define the state transitions and state-specific operations.

## Sample Code 1



### Basic Java code for implementing the sample UML diagrams.

```

1 package com.sample.state.basic;
2 public class Client {
3     // Running the Client class as application.
4     public static void main(String[] args) {
5         // Creating a context object and configuring it with
6         // the initial state.
7         Context context = new Context(State1.STATE1);
8         //
9         System.out.println("(1) Calling an operation on Context ..... : " +
10             context.operation());
11        //
12        System.out.println("(2) Calling an operation on Context ..... : " +
13            context.operation());
14    }
15 }

(1) Calling an operation on Context ..... : Hello World1!
    Changing Context's current state to STATE2.
(2) Calling an operation on Context ..... : Hello World2!
    Changing Context's current state to STATE1.

1 package com.sample.state.basic;
2 public class Context {
3     private State state; // current state
4     //
5     public Context(State s) { // constructor
6         // Configuring with the initial state.
7         this.state = s;
8     }
9     //
10    public String operation() {
11        // Delegating state-specific behavior to the current state.
12        return state.operation(this);
13    }
14    // Setting current state.
15    void setState(State s) { // package private
16        this.state = s;
17    }
18 }

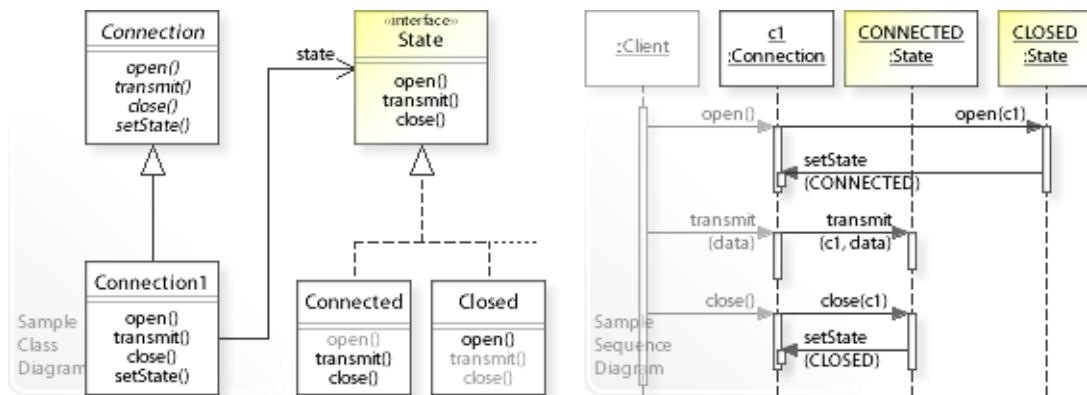
1 package com.sample.state.basic;
2 public interface State {
3     String operation(Context context);
4 }

1 package com.sample.state.basic;
2 class State1 implements State {
3     // Implemented as Singleton.
4     public static final State STATE1 = new State1();
5     private State1() { }
6     //
7     public String operation(Context context) {
8         String result = "Hello World1!" +

```

```
9          "\n      Changing Context's current state to STATE2.";  
10         // Changing state (state transition).  
11         context.setState(State2.STATE2);  
12         return result;  
13     }  
14 }  
  
1 package com.sample.state.basic;  
2 public class State2 implements State {  
3     // Implemented as Singleton.  
4     public static final State STATE2 = new State2();  
5     private State2() { }  
6     //  
7     public String operation(Context context) {  
8         String result = "Hello World2!" +  
9             "\n      Changing Context's current state to STATE1."  
10            // Changing state (state transition).  
11            context.setState(State1.STATE1);  
12            return result;  
13        }  
14 }
```

## Sample Code 2



### Network communication states (Connected / Closed).

```

1 package com.sample.state.tcp;
2 import java.io.OutputStream;
3 public class Client {
4     // Running the Client class as application.
5     public static void main(String[] args) {
6         OutputStream data = null;
7         Connection connection = new Connection1(Closed.CLOSED);
8         connection.open();
9         // ...
10        connection.transmit(data);
11        // ...
12        connection.close();
13    }
14}

State changed from CLOSED to CONNECTED.
State CONNECTED: Transmitting data ... Finished.
State changed from CONNECTED to CLOSED.

```

```

1 package com.sample.state.tcp;
2 import java.io.OutputStream;
3 public abstract class Connection {
4     public abstract void open();
5     public abstract void transmit(OutputStream data);
6     public abstract void close();
7     abstract void setState(State state); // package private
8 }

1 package com.sample.state.tcp;
2 import java.io.OutputStream;
3 public class Connection1 extends Connection {
4     private State state;
5     // Configuring Context with a State.
6     public Connection1(State state) {
7         this.state = state;
8     }
9     public void open() {
10         state.open(this);
11     }
12     public void transmit(OutputStream data) {
13         state.transmit(this, data);
14     }
15     public void close() {
16         state.close(this);
17     }
18     void setState(State state) {
19         this.state = state;
20     }
21 }

1 package com.sample.state.tcp;
2 import java.io.OutputStream;
3 public interface State {

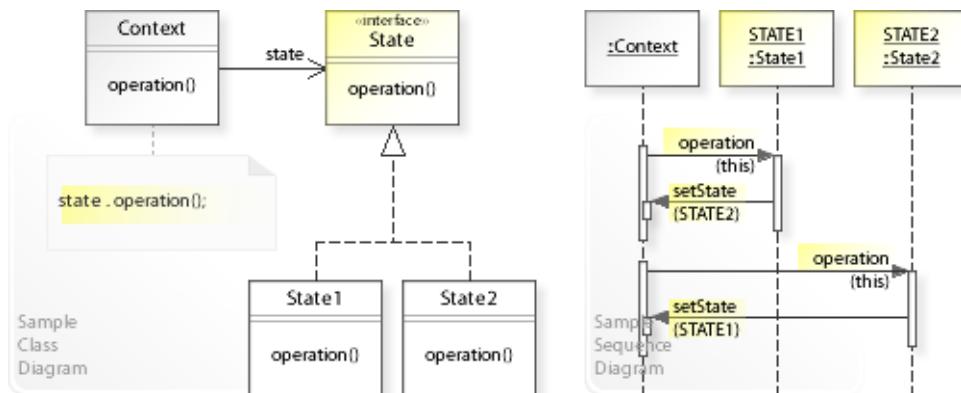
```

```
4     void open(Connection c);
5     void transmit(Connection c, OutputStream data);
6     void close(Connection c);
7 }

1 package com.sample.state.tcp;
2 import java.io.OutputStream;
3 public class Connected implements State {
4     // Implemented as Singleton.
5     public static final State CONNECTED = new Connected();
6     private Connected() { }
7     //
8     public void open(Connection c) {
9         System.out.println(
10            "State CONNECTED: *** Can't open connection " +
11            "(connection already opened). ***");
12         System.exit(-1);
13     }
14     public void transmit(Connection c, OutputStream data) {
15         // ...
16         System.out.println(
17             "State CONNECTED: Transmitting data ... Finished.");
18     }
19     public void close(Connection c) {
20         // ...
21         c.setState(Closed.CLOSED);
22         System.out.println(
23             "State changed from CONNECTED to CLOSED.");
24     }
25 }

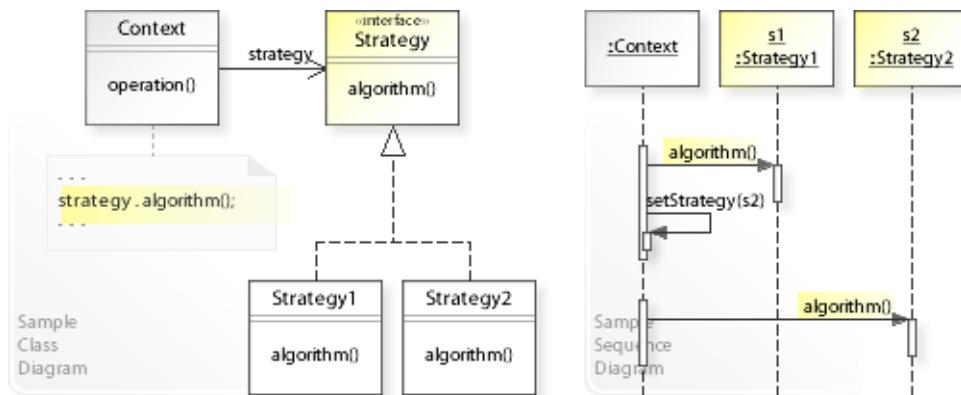
1 package com.sample.state.tcp;
2 import java.io.OutputStream;
3 public class Closed implements State {
4     // Implemented as Singleton.
5     public static final State CLOSED = new Closed();
6     private Closed() { }
7     //
8     public void open(Connection c) {
9         // ...
10        c.setState(Connected.CONNECTED);
11        System.out.println(
12            "State changed from CLOSED to CONNECTED.");
13    }
14    public void transmit(Connection c, OutputStream data) {
15        System.out.println(
16            "State CLOSED: *** Can't transmit data " +
17            "(connection is closed). ***");
18        System.exit(-1);
19    }
20    public void close(Connection c) {
21        System.out.println(
22            "State CLOSED: *** Can't close connection " +
23            "(connection already closed). ***");
24        System.exit(-1);
25    }
26 }
```

## Related Patterns



**Key Relationships** (see also Overview)

## Intent



The intent of the Strategy design pattern is to:

**"Define a family of algorithms, encapsulate each one, and make them interchangeable."**

**Strategy lets the algorithm vary independently from clients that use it."** [GoF]

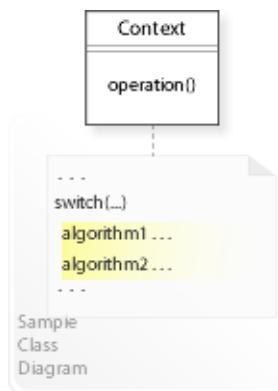
See Problem and Solution sections for a more structured description of the intent.

- The Strategy design pattern solves problems like:
  - *How can a family of interchangeable algorithms be supported?*
  - *How can an algorithm be selected and changed at run-time?*
- The term *algorithm* is usually defined as a procedure that takes some value as input, performs a finite number of steps, and produces some value as output.  
From a more general point of view, an algorithm is a *piece of code*.
- For example, calculating prices in an order processing system.  
To calculate prices in different ways (depending on run-time conditions like type of customer, volume of sales, product quantity, etc.), it should be possible to select the right *pricing algorithm* at run-time.
- The Strategy pattern describes how to solve such problems:
  - *Define a family of algorithms, encapsulate each one,* - define separate classes (**Strategy1**, **Strategy2**, ...) that implement (encapsulate) different algorithms,
  - *and make them interchangeable* - and define a common interface (**Strategy**) through which clients can select and (inter)change an algorithm at run-time.

## Background Information

- The **Intent** section is "A short statement that answers the following questions: What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address?" [GoF, p6]
- For providing a more structured and better comparable description of the intent, w3sDesign has introduced separate **Problem** and **Solution** sections.
  - The Problem section describes the key problems the design pattern can solve.
  - The Solution section describes how the design pattern solves the problems.
- *Hint: View how UML diagrams change when switching between sections or patterns.*

## Problem



The Strategy design pattern solves problems like:

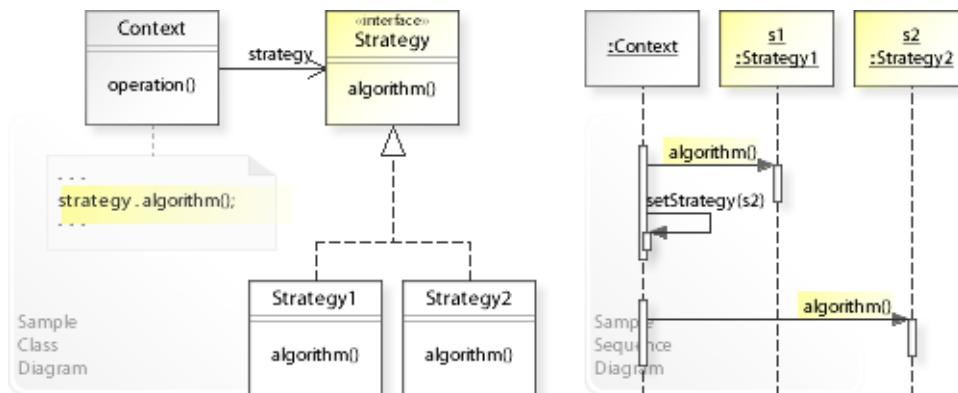
***How can a family of interchangeable algorithms be supported?***

***How can an algorithm be selected and changed at run-time?***

See Applicability section for all problems Strategy can solve.

- An inflexible way is to implement algorithms directly within the classes that require them (Context). Conditional statements (`switch(...)`) are needed to switch between different algorithms, and each conditional branch implements one of different algorithms.
- This commits the classes to particular algorithms and makes it hard to change them later both at compile-time and run-time, and it stops the classes from being reusable with different algorithms.  
Classes that include multiple algorithms get more complex and harder to implement, change, test, and reuse.  
"Hard-wiring all such algorithms into the classes that require them isn't desirable for several reasons:" [GoF, p315]    "Algorithms are often extended, optimized, and replaced during development and reuse." [GoF, p24]
- *That's the kind of approach to avoid if we want to select and change an algorithm at run-time instead of committing to an algorithm at compile-time.*
- For example, calculating prices in an order processing system.  
To calculate prices in different ways (depending on run-time conditions), it should be possible to support (many) different *pricing algorithms* so that the right one can be selected at run-time (see Sample Code / Example 2).
- For example, an operation for sorting objects.  
To sort objects in different ways, it should be possible to parameterize a sort operation with one of different *compare algorithms* (see Sample Code / Example 3).

## Solution



The Strategy pattern describes a solution:

**Encapsulate an algorithm in a separate `Strategy` object.**

**Classes delegate an algorithm to a `Strategy` object at run-time instead of implementing algorithms directly.**

Describing the Strategy design in more detail is the theme of the following sections. See Applicability section for all problems Strategy can solve.

- The key idea in this pattern is to encapsulate an algorithm in a separate object so that other objects can use it via object composition to gain run-time flexibility.
- **Define separate `Strategy` objects:**  
As a reminder, an object has an *outside view* (public interface/operations) and an *inside view* (private representation and implementation).  
*Encapsulation* means hiding a representation and implementation in an object.  
Clients can only see the outside view of an object and are independent of the inside view. This is the essential benefit of encapsulation.
  - For all supported algorithms, define a common interface for performing an algorithm (`Strategy` | `algorithm(...)`).
  - Define classes (`Strategy1`, `Strategy2`, ...) that implement the `Strategy` interface.
- This enables *compile-time* flexibility (via inheritance).  
New algorithms can be added and existing ones can be changed independently by defining new (sub)classes.
- **Clients delegate the responsibility to perform the algorithm to a `Strategy` object** (`strategy.algorithm(...)`).  
This enables *run-time* flexibility (via object composition).  
Clients can use different `Strategy` objects to change an algorithm at run-time.  
Clients can be configured with a `Strategy` object, which they use to perform an algorithm, and even more, the `Strategy` object can be exchanged at run-time.

## Background Information

- Encapsulation is
  - "The result of hiding a representation and implementation in an object." [GoF, p360]
  - Clients (from outside the object) can only see and access the (public) operations defined by an object's interface.
  - Clients do not see (are independent of) any changes of an object's (private/hidden) representation and implementation.
  - *This is the essential benefit of encapsulation.*
  - See also Design Principles.
- Terms and Definitions
  - "A responsibility denotes the obligation of an object to provide a certain behavior." [GBooch07, p600]
  - The terms *responsibility*, *behavior*, and *functionality* are usually interchangeable.

## Motivation 1



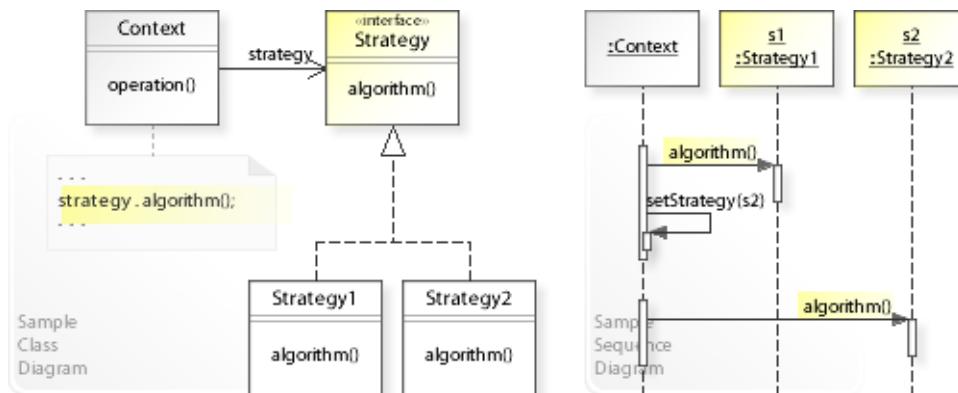
**Consider the left design (problem):**

- Hard-wired algorithms.
  - Different algorithms are implemented (hard-wired) directly within a class (`Context`).
  - This makes it hard to add new algorithms or change existing ones independently from (without having to change) the `Context` class.
- Conditional statements required.
  - Conditional statements are needed to switch between different algorithms.
- Complicated classes.
  - Classes that require multiple algorithms get more complex if they include them.
  - This makes classes harder to implement, change (maintain), test, and reuse.

**Consider the right design (solution):**

- Encapsulated algorithms.
  - Each algorithm is implemented (encapsulated) in a separate class (`Strategy1`, `Strategy2`, ...).
  - This makes it easy to add new algorithms or change existing ones independently from (without having to change) the `Context` class.
- No conditional statements required.
  - Conditional statements are replaced by delegating to different `Strategy` objects.
- Simplified classes.
  - Classes that require multiple algorithms get less complex if they delegate them.
  - This makes classes easier to implement, change (maintain), test, and reuse.

## Applicability



## Design Problems

- **Supporting Interchangeable Algorithms**
  - How can a class use different algorithms?
  - How can a class be configured with an algorithm?
  - How can an algorithm be selected and changed at run-time?
- **Flexible Alternative to Subclassing**
  - How can a flexible alternative be provided to changing the algorithm of a class by subclassing?

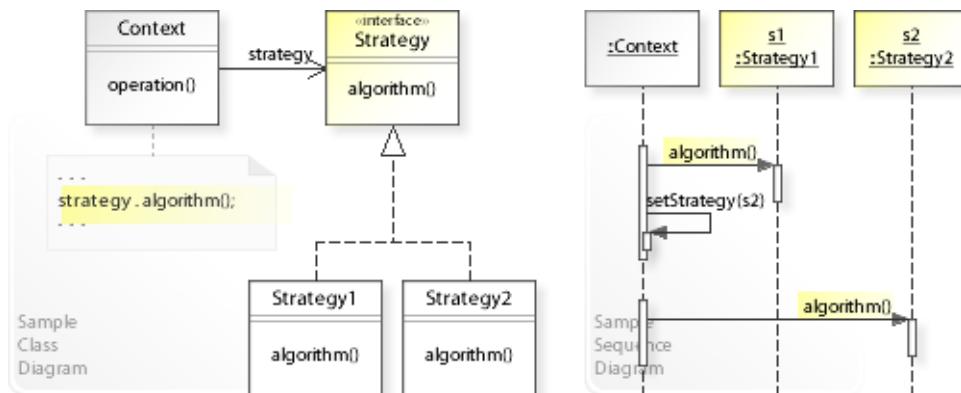
## Refactoring Problems

- **Duplicated Code**
  - How can algorithms that are duplicated in multiple places be refactored?
  - How can related classes that differ only in their algorithms be replaced by a common class that is configured with an algorithm?
- **Complicated Code**
  - How can multiple conditional statements that switch between different algorithms be eliminated? *Replace Conditional Logic with Strategy (129) [JKerievsky05]*

## Background Information

- Refactoring and "Bad Smells in Code" [MFowler99] [JKerievsky05]
  - *Code smells* are certain structures in the code that "smell bad" and indicate problems that can be solved by a refactoring.
  - The most common code smells are:  
*complicated code* (including complicated/growing conditional code),  
*duplicated code*,  
*inflexible code* (that must be changed whenever requirements change), and  
*unclear code* (that doesn't clearly communicate its intent).

## Structure, Collaboration



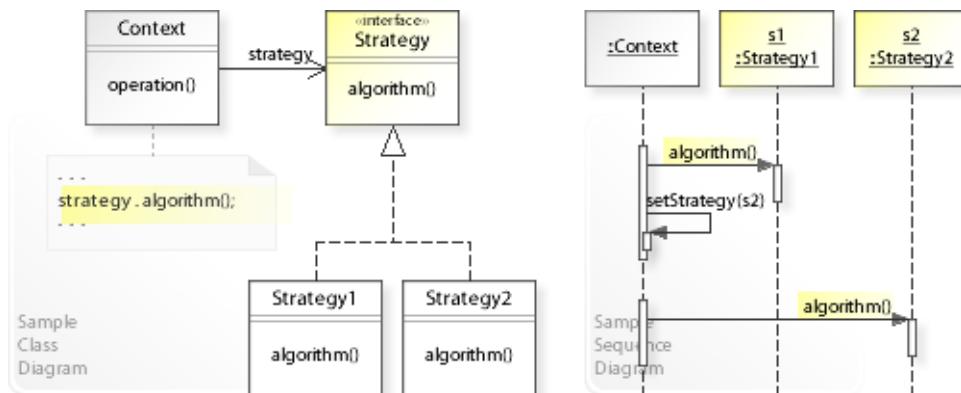
### Static Class Structure

- **Context**
  - Refers to the **Strategy** interface to perform an algorithm (`strategy.algorithm()`) and is independent of how the algorithm is implemented.
- **Strategy**
  - For all supported algorithms, defines a common interface for performing an algorithm.
- **Strategy1, Strategy2,...**
  - Implement the **Strategy** interface.

### Dynamic Object Collaboration

- In this sample scenario, a **Context** object delegates an algorithm to different **Strategy** objects. Let's assume that **Context** is configured with a **Strategy1** object.
- The interaction starts with the **Context** object that performs a client request by calling **algorithm()** on the installed **Strategy1** object.
- **Strategy1** performs the algorithm and returns the result to **Context**.
- Let's assume that **Context** changes its strategy by calling **setStrategy(s2)** on itself (because of run-time conditions such as reaching a threshold, for example).
- There are different ways to select and change the strategy. For example, a client of **Context** could change the strategy by calling **setStrategy(s2)** on **Context**, or it could pass in the strategy by calling **operation(s2)** on **Context**.
- Thereafter, **Context** again performs a client request but now by calling **algorithm()** on the **Strategy2** object.
- See also Sample Code / Example 1.

## Consequences



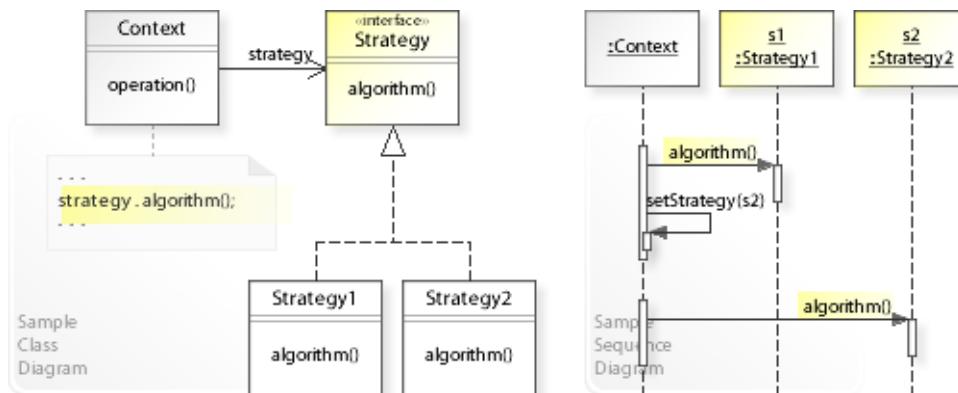
### Advantages (+)

- Avoids implementation dependencies.
  - Clients refer to an interface (`Strategy`) and are independent of an implementation.
- Provides a flexible alternative to subclassing.
  - Subclassing provides a way to change the algorithm of a class (at compile-time). When a subclass is instantiated, its algorithm is fixed and can't be changed for the life-time of the object.
  - Strategy provides a way to change the algorithm of an object (at run-time) by delegating to different `Strategy` objects.
- Avoids conditional statements for switching between algorithms.
  - Conditional statements that switch between different algorithms are replaced by delegating to different `Strategy` objects.
  - "Code containing many conditional statements often indicates the need to apply the Strategy pattern." [GoF, p318]

### Disadvantages (-)

- Can make the common `Strategy` interface complex.
  - The `Strategy` interface may get complex because it must pass in the needed data for all supported algorithms (whether they are simple or complex; see Implementation).
- Requires that clients understand how strategies differ.
  - "The pattern has a potential drawback in that a client must understand how Strategies differ before it can select the appropriate one. Clients might be exposed to implementation issues." [GoF, p318]
- Introduces an additional level of indirection.
  - Strategy achieves flexibility by introducing an additional level of indirection (classes delegate an algorithm to a separate `Strategy` object), which makes classes dependent on a `Strategy` object.
  - This "can complicate a design and/or cost you some performance. A design pattern should only be applied when the flexibility it affords is actually needed." [GoF, p31]

## Implementation



### Implementation Issues

- **Implementation Variants**

- The **Context** and **Strategy** interfaces must be designed carefully so that the needed data can be passed/accessed efficiently and new algorithms can be added without having to extend an interface. There are two main variants:

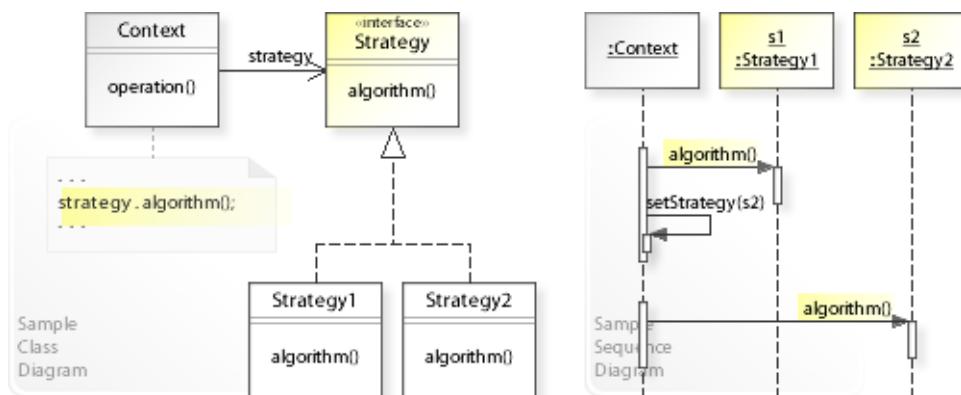
- **Variant1: Push Data**

- Context passes the data to the strategy:  
`strategy.algorithm(data1, data2, ...)`
- The **Strategy** interface may get complex because it must pass in the needed data for all supported algorithms (whether they are simple or complex).

- **Variant2: Pull Data**

- Context passes nothing but itself to the strategy, letting strategy call back to get (pull) the required data from context:  
`strategy.algorithm(this)`
- The **Context** interface may have to be extended to let strategies do their work and access the needed data.

## Sample Code 1



### Basic Java code for implementing the sample UML diagrams.

```

1 package com.sample.strategy.basic;
2 public class MyApp {
3     public static void main(String[] args) {
4         // Creating a Context object
5         // and configuring it with a Strategy1 object.
6         Context context = new Context(new Strategy1());
7         //
8         System.out.println("(1) " + context.operation());
9         //
10        System.out.println("(2) " + context.operation());
11    }
12 }

(1) Hello World from Context using Strategy1!
(2) Hello World from Context using Strategy2!

1 package com.sample.strategy.basic;
2 public class Context {
3     private Strategy strategy;
4     //
5     public Context(Strategy strategy) {
6         this.strategy = strategy;
7     }
8     public String operation() {
9         // Delegating an algorithm to the installed strategy.
10        int result = strategy.algorithm();
11        //
12        // Changing the strategy.
13        setStrategy(new Strategy2());
14        //
15        return "Hello World from Context using Strategy"
16                + result + "!";
17    }
18    // Setting/changing the strategy.
19    public void setStrategy(Strategy strategy) {
20        this.strategy = strategy;
21    }
22 }

1 package com.sample.strategy.basic;
2 public interface Strategy {
3     int algorithm();
4 }

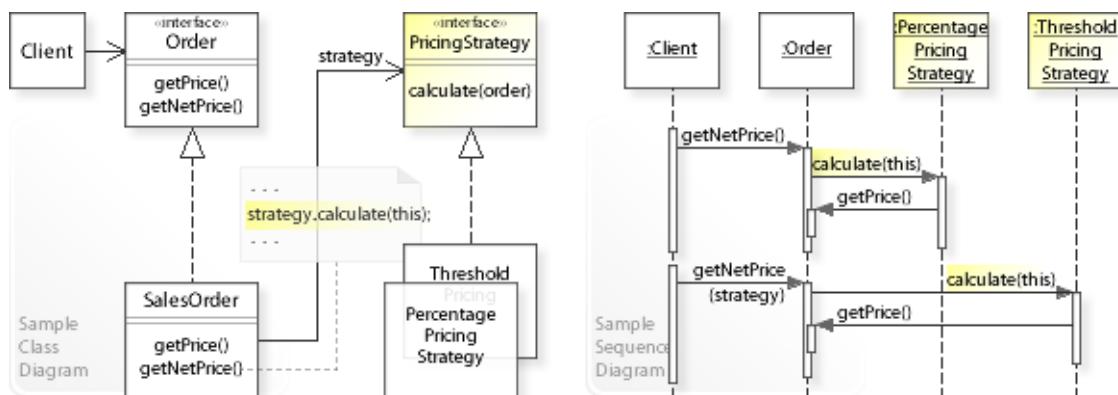
1 package com.sample.strategy.basic;
2 public class Strategy1 implements Strategy {
3     public int algorithm() {
4         // Implementing the algorithm.
5         return 1; // return result
6     }
7 }

1 package com.sample.strategy.basic;

```

```
2  public class Strategy2 implements Strategy {  
3      public int algorithm() {  
4          // Implementing the algorithm.  
5          return 2; // return result  
6      }  
7  }
```

## Sample Code 2



## Order Processing / Calculating order netto prices using different pricing strategies.

```

1 package com.sample.strategy.order;
2 import com.sample.data.*;
3 public class Client {
4     // Running the Client class as application.
5     public static void main(String[] args) {
6         // Creating a sales order and configuring it with a (default) strategy.
7         Order order = new SalesOrder(new PercentagePricingStrategy());
8         // Creating products and order lines.
9         Product product1A = new SalesProduct("1A", "01", "Product1A", 100);
10        Product product1B = new SalesProduct("1B", "01", "Product1B", 200);
11        order.createOrderLine(product1A, 1);
12        order.createOrderLine(product1B, 1);
13        //
14        System.out.println(
15            "Total order brutto price ..... : " +
16            order.getPrice());
17        //
18        System.out.println(
19            "Calculating the total order netto price by\n" +
20            "(1) using the default percentage strategy (10%) ..... : " +
21            order.getNetPrice());
22        //
23        System.out.println(
24            "(2) specifying the threshold strategy (10%; above 200: 20%): " +
25            order.getNetPrice(new ThresholdPricingStrategy()));
26    }
27 }
```

```

Total order brutto price ..... : 300
Calculating the total order netto price by
(1) using the default percentage strategy (10%) ..... : 270
(2) specifying the threshold strategy (10%; above 200: 20%): 240
```

```

1 package com.sample.data;
2 public interface Order { // prices are in cents
3     long getPrice();
4     long getNetPrice();
5     long getNetPrice(PricingStrategy strategy);
6     void createOrderLine(Product product, int quantity);
7 }
```

```
1 package com.sample.data;
2 import java.util.ArrayList;
3 import java.util.List;
4 public class SalesOrder implements Order {
5     private List<OrderLine> orderLines = new ArrayList<OrderLine>();
6     private PricingStrategy strategy;
7     // Configuring sales order with a (default) strategy.
8     public SalesOrder(PricingStrategy strategy) {
9         this.strategy = strategy;
10    }
11    public long getPrice() {
12        long total = 0;
13        for (OrderLine orderLine : orderLines) {
14            total += orderLine.getPrice();
15        }
16        return total;
17    }
18    public long getNetPrice() {
19        // Delegating the calculation to the default strategy.
20        // Passing a reference to itself (this) so that strategy
21        // can act (call back) through the order interface.
22        return strategy.calculate(this);
23    }
24    public long getNetPrice(PricingStrategy strategy) {
25        // Delegating the calculation to the passed in strategy.
26        return strategy.calculate(this);
27    }
28    public void createOrderLine(Product product, int quantity) {
29        orderLines.add(new SalesOrderLine(product, quantity));
30    }
31 }

1 package com.sample.data;
2 public interface PricingStrategy {
3     long calculate(Order order);
4 }

1 package com.sample.data;
2 public class PercentagePricingStrategy implements PricingStrategy {
3     public long calculate(Order order) {
4         // Calculating percentage ...
5         int percentage = 10;
6         //
7         long amount = order.getPrice();
8         long rabat = amount / 100 * percentage;
9         return amount - rabat;
10    }
11 }

1 package com.sample.data;
2 public class ThresholdPricingStrategy implements PricingStrategy {
3     public long calculate(Order order) {
4         // Calculating threshold, percentage low/high ...
5         long threshold = 200;
6         short percentageLow = 10;
7         short percentageHigh = 20;
8         //
9         long amount = order.getPrice();
10        if (amount < threshold)
11            return amount - amount / 100 * percentageLow;
12        else
13            return amount - amount / 100 * percentageHigh;
14    }
15 }
```

```
*****
Other interfaces and classes used in this example.
*****
```

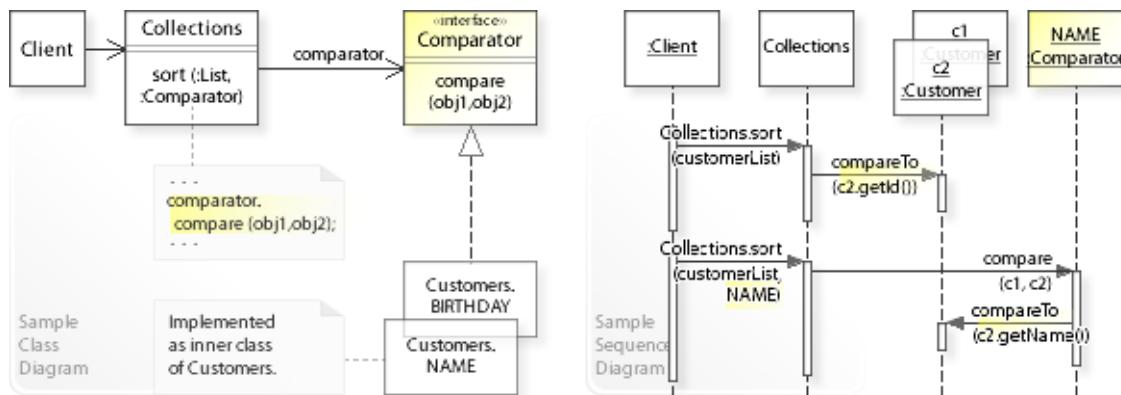
```
1 package com.sample.data;
2 public interface OrderLine {
3     Product getProduct();
4     int getQuantity();
5     long getPrice();
6 }

1 package com.sample.data;
2 public class SalesOrderLine implements OrderLine {
3     private Product product;
4     private int quantity;
5     //
6     public SalesOrderLine(Product product, int quantity) {
7         this.product = product;
8         this.quantity = quantity;
9     }
10    public Product getProduct() {
11        return product;
12    }
13    public int getQuantity() {
14        return quantity;
15    }
16    public long getPrice() {
17        return product.getPrice() * quantity;
18    }
19 }

1 package com.sample.data;
2 public interface Product {
3     void operation();
4     String getId();
5     String getGroup();
6     String getDescription();
7     long getPrice();
8 }

1 package com.sample.data;
2 public class SalesProduct implements Product {
3     private String id;
4     private String group;
5     private String description;
6     private long price;
7     //
8     public SalesProduct(String id, String group, String description, long price) {
9         this.id = id;
10        this.group = group;
11        this.description = description;
12        this.price = price;
13    }
14    public void operation() {
15        System.out.println("SalesProduct: Performing an operation ...");
16    }
17    public String getId() {
18        return id;
19    }
20    public String getGroup() {
21        return group;
22    }
23    public String getDescription() {
24        return description;
25    }
26    public long getPrice() {
27        return price;
28    }
29 }
```

### Sample Code 3



#### Sorting customers using different compare strategies.

Customer Test Data:

```
ID Name-----PhoneNumber---Birthday--
01 FirstName1 LastName1 (001) 002-0002 03.03.1980
02 FirstName3 LastName3 (001) 003-0003 01.01.1970
03 FirstName2 LastName2 (002) 001-0001 02.02.1980
PhoneNumber = (areaCode) + prefix + lineNumber.
```

```

1 package com.sample.strategy.sort;
2 import java.util.Comparator; // = Strategy (compare algorithm)
3 import com.sample.data.Customer;
4 import com.sample.data.Customers;
5 import java.util.Collections;
6 import java.util.List;
7 public class Client {
8     // Running the Client class as application.
9     public static void main(String[] args) throws Exception {
10         // Creating the customers.
11         List<Customer> customerList = Customers.createTestData(3);
12         //
13         System.out.println(
14             "SORTING CUSTOMERS:\n\n" +
15             "(1) by using the default comparator \n" +
16             "    = according to the customer ID: ");
17         Collections.sort(customerList);
18         System.out.println(customerList);
19         //
20         System.out.println("\n" +
21             "(2) by specifying the NAME comparator \n" +
22             "    = according to the customer name: ");
23         Collections.sort(customerList, Customers.NAME);
24         System.out.println(customerList);
25         //
26         // Implementing individual requirements directly, for example:
27         System.out.println("\n" +
28             "(3) by implementing the comparator directly \n" +
29             "    = according to the (area code) of the customer phone number \n" +
30             "    and the customer name: ");
31         Collections.sort(customerList, new Comparator<Customer>() { // inner class
32             public int compare(Customer c1, Customer c2) {
33                 // Implementing the comparator interface / compare().
34                 if (c1.getPhoneNumber().getAreaCode() <
35                     c2.getPhoneNumber().getAreaCode()) return -1;
36                 if (c1.getPhoneNumber().getAreaCode() >
37                     c2.getPhoneNumber().getAreaCode()) return 1;
38                 // Area codes are equal, compare names:
39                 // getName() returns an object of type Name;
40                 // compareTo() implemented in the Name class.
41                 return (c1.getName().compareTo(c2.getName()));
42             }
43         });
44         System.out.println(customerList);
45     }

```

```

46     System.out.println("\n" +
47         "(4) by specifying the BIRTHDAY comparator \n" +
48         "      = according to the customer birthday descending: ");
49     Collections.sort(customerList, Customers.BIRTHDAY);
50     System.out.println(customerList);
51 }
52 }

SORTING CUSTOMERS:

(1) by using the default comparator
    = according to the customer ID:
[
Customer: 1 FirstName1 LastName1 (001) 002-0002 03.03.1980,
Customer: 2 FirstName3 LastName3 (001) 003-0003 01.01.1970,
Customer: 3 FirstName2 LastName2 (002) 001-0001 02.02.1980]

(2) by specifying the NAME comparator
    = according to the customer name:
[
Customer: 1 FirstName1 LastName1 (001) 002-0002 03.03.1980,
Customer: 3 FirstName2 LastName2 (002) 001-0001 02.02.1980,
Customer: 2 FirstName3 LastName3 (001) 003-0003 01.01.1970]

(3) by implementing the comparator directly
    = according to the (area code) of the customer phone number
        and the customer name:
[
Customer: 1 FirstName1 LastName1 (001) 002-0002 03.03.1980,
Customer: 2 FirstName3 LastName3 (001) 003-0003 01.01.1970,
Customer: 3 FirstName2 LastName2 (002) 001-0001 02.02.1980]

(4) by specifying the BIRTHDAY comparator
    = according to the customer birthday descending:
[
Customer: 1 FirstName1 LastName1 (001) 002-0002 03.03.1980,
Customer: 3 FirstName2 LastName2 (002) 001-0001 02.02.1980,
Customer: 2 FirstName3 LastName3 (001) 003-0003 01.01.1970]

1 package java.util;
2 // From the Java Language = Strategy (compare algorithm).
3 public interface Comparator<T> {
4     /**
5      * The compare method must be implemented to compare
6      * two objects for order:
7      * returns a negative integer, zero, or a positive integer
8      * as the first argument is less than, equal to,
9      * or greater than the second.
10     */
11     int compare(T object1, T object2);
12     ...
13 }

1 package com.sample.data;
2 import java.util.Comparator;
3 import java.util.List;
4 import java.util.ArrayList;
5 import java.util.Date;
6 import java.text.SimpleDateFormat;
7 import java.text.ParseException;
8 //
9 // This is a non-instantiable class that holds (public)
10 // static utility methods needed for handling customers.
11 //
12 public class Customers {
13     private static final SimpleDateFormat dateFormatter =
14         new SimpleDateFormat("dd.MM.yyyy");
15     private Customers() { }
16     // NAME = reference to a comparator object.
17     public static final Comparator<Customer> NAME =
18         new Comparator<Customer>() { // inner class
19             // Implementing the comparator interface / compare().
20             public int compare(Customer c1, Customer c2) {
21                 // compareTo() implemented in the Name class.
22                 return c1.getName().compareTo(c2.getName());
23             }
24         };
25 }

```

```

23         }
24     } ;
25     // PHONENUMBER = reference to a comparator object.
26     public static final Comparator<Customer> PHONENUMBER =
27         new Comparator<Customer>() { // inner class
28             // Implementing the comparator interface / compare().
29             public int compare(Customer c1, Customer c2) {
30                 // getPhoneNumber() returns an object of type PhoneNumber.
31                 // compareTo() implemented in the PhoneNumber class.
32                 return c1.getPhoneNumber().compareTo(c2.getPhoneNumber());
33             }
34         } ;
35     // BIRTHDAY = reference to a comparator object.
36     public static final Comparator<Customer> BIRTHDAY =
37         new Comparator<Customer>() { // inner class
38             // Implementing the comparator interface / compare().
39             public int compare(Customer c1, Customer c2) {
40                 // compareTo() implemented in the Date class (Java platform).
41                 return c2.getBirthday().compareTo(c1.getBirthday());
42             }
43         } ;
44     //
45     public static void checkData(int id, Name name, PhoneNumber pn, Date birthday) throws ParseException {
46         if (id < 0)
47             throw new IllegalArgumentException("Customer ID is negative");
48         if (birthday.compareTo(dateFormatter.parse("01.01.1900")) < 0 || 
49             birthday.compareTo(dateFormatter.parse("01.01.2000")) > 0)
50             throw new IllegalArgumentException("Birthday before 1900 or after 2000");
51         // ...
52     }
53     public static List<Customer> createTestData(int size) throws Exception {
54         List<Customer> customerList = new ArrayList<Customer>(size);
55         customerList.add(new Customer1(1,
56             new Name("FirstName1", " LastName1"),
57             new PhoneNumber(1, 2, 2), dateFormatter.parse("03.03.1980")));
58         customerList.add(new Customer1(2,
59             new Name("FirstName3", " LastName3"),
60             new PhoneNumber(1, 3, 3), dateFormatter.parse("01.01.1970")));
61         customerList.add(new Customer1(3,
62             new Name("FirstName2", " LastName2"),
63             new PhoneNumber(2, 1, 1), dateFormatter.parse("02.02.1980")));
64         return customerList;
65     }
66 }

1 package com.sample.data;
2 import java.util.Date;
3 public interface Customer extends Comparable<Customer> {
4     int getId();
5     Name getName();
6     PhoneNumber getPhoneNumber();
7     Date getBirthday();
8 }

1 package com.sample.data;
2 import java.text.SimpleDateFormat;
3 // Skeletal implementation of the customer interface.
4 import java.util.Date;
5 public abstract class AbstractCustomer implements Customer {
6     private final int id;
7     private final Name name;
8     private final PhoneNumber phoneNumber;
9     private final Date birthday;
10    private static final SimpleDateFormat dateFormatter =
11        new SimpleDateFormat("dd.MM.yyyy");
12    // TODO Check
13    protected AbstractCustomer(int id, Name name, PhoneNumber pn, Date birthday) throws Exception {
14        Customers.checkData(id, name, pn, birthday);
15        this.id = id;
16        this.name = name;
17        this.phoneNumber = pn;
18        this.birthday = birthday;
19    }
20    @Override
21    public boolean equals(Object o) {
22        if (o == this) return true;

```

```

23         if (!(o instanceof Customer)) return false;
24         Customer c = (Customer) o;
25         return c.getId() == id;
26     }
27     @Override
28     public int hashCode() {
29         int result = 17;
30         result = 31 * result + id;
31         return result;
32     }
33     @Override
34     public String toString() {
35         return "\nCustomer: " + id + " " + name + " " + phoneNumber + " " +
36             dateFormatter.format(birthday);
37     }
38     // The compareTo method implements the Comparable interface.
39     // It defines the "natural ordering" (default ordering).
40     public int compareTo(Customer c) {
41         if (id < c.getId()) return -1;
42         if (id > c.getId()) return 1;
43         // All fields are equal.
44         return 0;
45     }
46     //
47     public int getId() {
48         return id;
49     }
50     public Name getName() {
51         return name;
52     }
53     public PhoneNumber getPhoneNumber() {
54         return phoneNumber;
55     }
56     public Date getBirthday() {
57         return birthday;
58     }
59 }

1 package com.sample.data;
2 import java.util.Date;
3 public class Customer1 extends AbstractCustomer {
4     public Customer1(int id, Name name, PhoneNumber pn, Date birthday) throws Exception {
5         super(id, name, pn, birthday);
6         // ...
7     }
8     // ...
9 }

*****
Other classes used in this example.
*****


1 package com.sample.data;
2 public class Name implements Comparable<Name> {
3     private final String firstName;
4     private final String lastName;
5     // ...
6     public Name(String firstName, String lastName) {
7         if (firstName == null || lastName == null)
8             throw new NullPointerException();
9         this.firstName = firstName;
10        this.lastName = lastName;
11    }
12    @Override
13    public boolean equals(Object o) {
14        if (!(o instanceof Name)) return false;
15        Name name = (Name) o;
16        return name.firstName.equals(firstName)
17            && name.lastName.equals(lastName);
18    }
19    @Override
20    public int hashCode() {
21        return 31 * firstName.hashCode() + lastName.hashCode();
22    }
23    @Override
24    public String toString() {

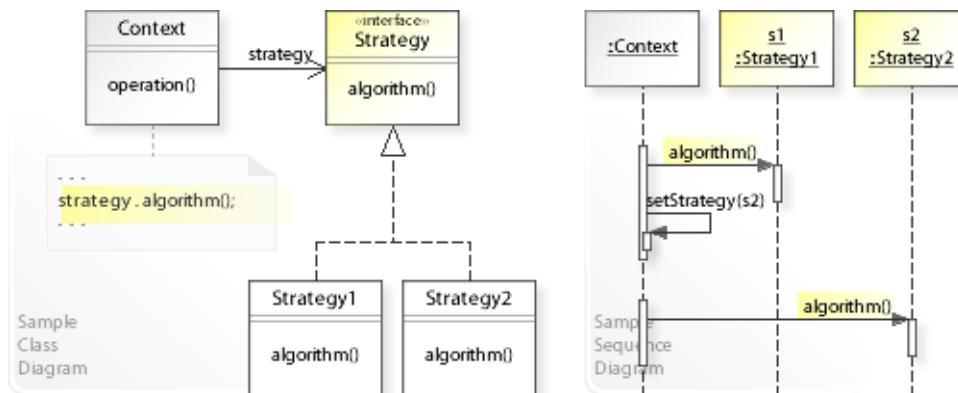
```

```
25         return firstName + " " + lastName;
26     }
27     // The compareTo method implements the Comparable interface.
28     // It defines the "natural ordering" (default ordering).
29     public int compareTo(Name name) {
30         int cmp = lastName.compareTo(name.lastName);
31         if (cmp != 0) return cmp;
32         // Last names are equal, compare first names.
33         return firstName.compareTo(name.firstName);
34     }
35     //
36     public String getFirstName() {
37         return firstName;
38     }
39     public String getLastName() {
40         return lastName;
41     }
42 }

1 // Based on Joshua Bloch / Effective Java / Item 9,10, and 12.
2 package com.sample.data;
3 public class PhoneNumber implements Comparable<PhoneNumber> {
4     private final short areaCode;
5     private final short prefix;
6     private final short lineNumber;
7     public PhoneNumber(int areaCode, int prefix, int lineNumber) {
8         rangeCheck(areaCode, 999, "area code");
9         rangeCheck(prefix, 999, "prefix");
10        rangeCheck(lineNumber, 9999, "line number");
11        this.areaCode = (short) areaCode;
12        this.prefix = (short) prefix;
13        this.lineNumber = (short) lineNumber;
14    }
15    private static void rangeCheck(int arg, int max, String name) {
16        if (arg < 0 || arg > max)
17            throw new IllegalArgumentException(name + ": " + arg);
18    }
19    @Override
20    public boolean equals(Object o) {
21        if (o == this) return true;
22        if (!(o instanceof PhoneNumber)) return false;
23        PhoneNumber pn = (PhoneNumber) o;
24        return pn.lineNumber == lineNumber
25            && pn.prefix == prefix
26            && pn.areaCode == areaCode;
27    }
28    @Override
29    public int hashCode() {
30        int result = 17;
31        result = 31 * result + areaCode;
32        result = 31 * result + prefix;
33        result = 31 * result + lineNumber;
34        return result;
35    }
36    /**
37     * Returns the string representation of this phone number. The string
38     * consists of 14 characters whose format is "(XXX) YYY-ZZZZ", where XXX is
39     * the area code, YYY is the prefix, and ZZZZ is the line number.
40     */
41    @Override
42    public String toString() {
43        return String.format("(%03d) %03d-%04d",
44            areaCode, prefix, lineNumber);
45    }
46    // The compareTo method implements the Comparable interface.
47    // It defines the "natural ordering" (default ordering).
48    public int compareTo(PhoneNumber pn) {
49        if (areaCode < pn.areaCode) return -1;
50        if (areaCode > pn.areaCode) return 1;
51        // Area codes are equal, compare prefixes.
52        if (prefix < pn.prefix) return -1;
53        if (prefix > pn.prefix) return 1;
54        // Area codes and prefixes are equal, compare line numbers.
55        if (lineNumber < pn.lineNumber) return -1;
56        if (lineNumber > pn.lineNumber) return 1;
57        // All fields are equal.
```

```
58         return 0;
59     }
60     //
61     public int getAreaCode() {
62         return areaCode;
63     }
64     public int getPrefix() {
65         return prefix;
66     }
67     public int getLineNumber() {
68         return lineNumber;
69     }
70 }
```

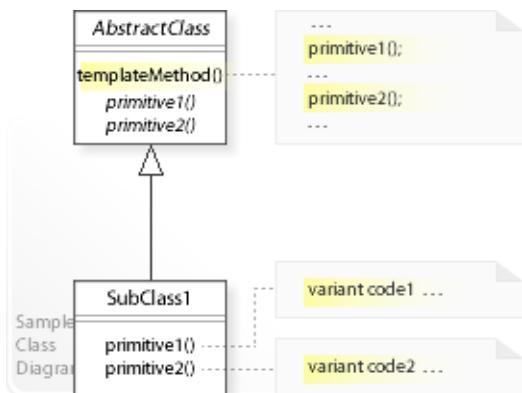
## Related Patterns



### Key Relationships (see also Overview)

- **Strategy - Template Method - Subclassing**
  - Strategy provides a way to change the algorithm/behavior of an object at run-time.
  - Template Method provides a way to control how subclasses change the algorithm/behavior of a class at compile-time.
  - Subclassing is the standard way to change the algorithm/behavior of a class at compile-time.
- **Strategy - Abstract Factory**
  - Strategy A class delegates an algorithm to a strategy object.
  - Abstract Factory A class delegates instantiation to a factory object.
- **Strategy - Dependency Injection**
  - Strategy A class can be configured with a strategy object.
  - Dependency Injection A class can be configured with the objects it requires.
- **Strategy - Decorator**
  - Strategy provides a way to exchange the algorithm of an object dynamically. This is done from *inside* the object. The object is designed to delegate an algorithm to a `Strategy` object.
  - Decorator provides a way to extend the functionality of an object dynamically. This is done from *outside* the object. The object already exists and isn't needed to be touched. That's very powerful.
- **Strategy - Command**
  - Strategy provides a way to configure an object with an algorithm at run-time instead of committing to an algorithm at compile-time.
  - Command provides a way to configure an object with a request at run-time instead of committing to a request at compile-time.

## Intent



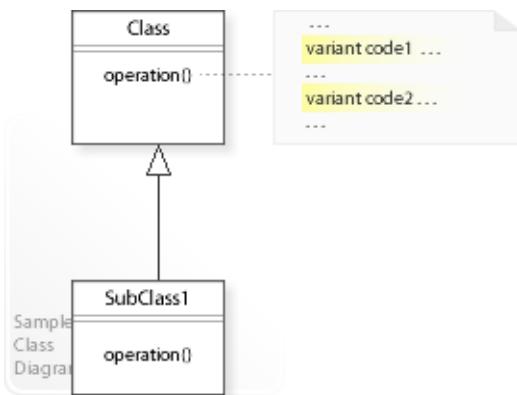
The intent of the Template Method design pattern is to:

**"Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure."** [GoF]

See Problem and Solution sections for a more structured description of the intent.

- The Template Method design pattern solves problems like:
  - *How can the invariant parts of a behavior be implemented once so that subclasses can implement the variant parts?*
  - *How can subclasses redefine certain parts of a behavior (steps of an algorithm) without changing the other parts?*
- For example, designing reusable applications (frameworks).  
It should be possible to implement the common (invariant) parts of a behavior and let users of the application write subclasses to redefine the variant parts to suit their needs.  
But subclass writers should not be able to change the behavior's invariant parts.
- The Template Method pattern describes how to solve such problems:
  - *Define the skeleton of an algorithm in an operation, deferring some steps to subclasses.*
  - Define abstract primitive operations for certain (variant) parts of an algorithm/behavior (`primitive1()`, `primitive2()`, ...).
  - Define a `templateMethod()` that defines the skeleton of an algorithm/behavior by
    - implementing the invariant parts and
    - calling primitives to defer implementing the variant parts to subclasses.

## Problem



The Template Method design pattern solves problems like:

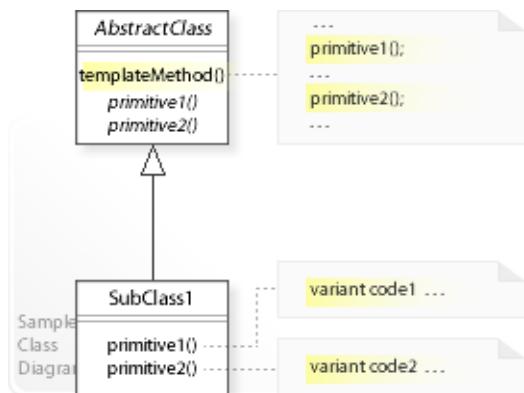
***How can the invariant parts of a behavior be implemented once so that subclasses can implement the variant parts?***

***How can subclasses redefine certain parts of a behavior without changing the other parts?***

See Applicability section for all problems Template Method can solve.

- The standard way to redefine a behavior (`Class | operation()`) is to define a new subclass and implement (redefine) the behavior (`SubClass1 | operation()`). This makes it hard for subclasses to redefine only certain parts of a behavior independently from (without changing/duplicating) the other parts.
- *That's the kind of approach to avoid if we want that subclasses can redefine only certain parts of a behavior (variant parts) without changing the other parts (invariant parts).*
- For example, designing reusable applications (frameworks). It should be possible to implement the common (invariant) parts of a behavior and let users of the application write subclasses to redefine the variant parts to suit their needs. But subclass writers should not be able to change the behavior's invariant parts.
- For example, enforcing invariants. A behavior often requires invariant functionality to be performed before and/or after its core functionality (for example, for setting up and resetting state). It should be possible to redefine only the core functionality without changing the invariant functionality.

## Solution



The Template Method pattern describes a solution:

**Define separate `primitive` operations for the variant parts of a behavior.**

**Define a `template` method that defines the skeleton of a behavior by**

**- implementing the invariant parts of the behavior and**

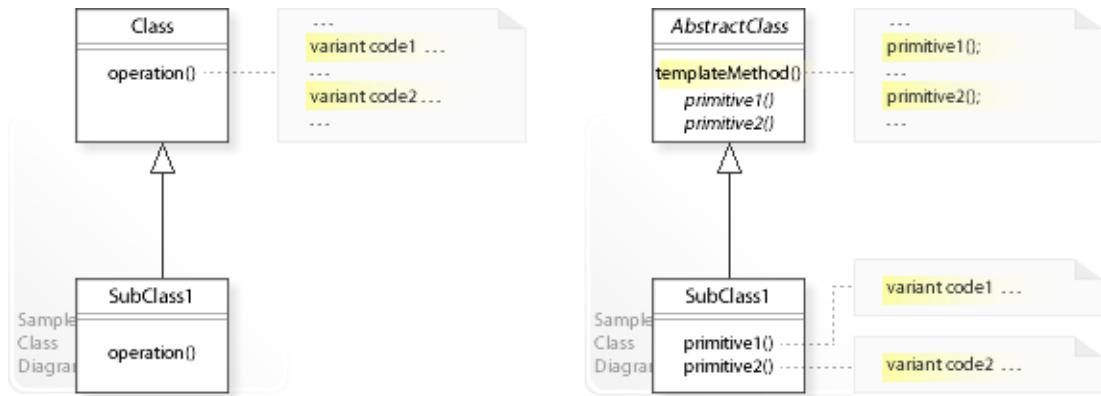
**- deferring the variant parts to `primitives` that subclasses can redefine.**

Describing the Template Method design in more detail is the theme of the following sections.

See Applicability section for all problems Template Method can solve.

- The pattern calls them *primitives* because a template method composes primitive operations to get a more complex operation.
- Subclasses can redefine only the variant parts without being able to change the invariant parts or the skeleton of the behavior (the behavior's structure).
- Template methods are often used to implement the common (invariant) parts of a behavior once "and leave it up to subclasses to implement the behavior that can vary." [GoF, p326] And from the refactoring point of view, common (invariant) behavior among classes can be factored out and localized (generalized) in a common class to eliminate code duplication.
- Template methods lead to an *inversion of control*.  
A parent class controls the way it is subclassed. "This refers to how a parent class calls the operations of a subclass and not the other way around." [GoF, p327]
- That's what distinguishes a *framework* from a *toolkit*.  
When using a toolkit (reusable classes), we call the code we want to reuse.  
When using a framework (reusable application), we write subclasses and implement the code the framework (parent class) calls.

## Motivation 1



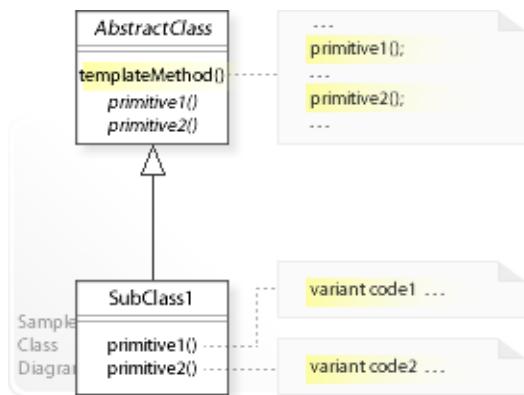
**Consider the left design (problem):**

- Hard-wired variant parts.
  - The variant parts are implemented (hard-wired) directly within the other code.
  - This makes it hard for subclasses to redefine the variant parts independently from the other code.
- Uncontrolled subclassing.
  - Subclasses can redefine all parts of a behavior, even the invariant parts.

**Consider the right design (solution):**

- Separated variant parts.
  - Each variant part is defined in a separate operation (`primitive1()`, ...).
  - This makes it easy for subclasses to redefine the variant parts independently from the other code.
- Controlled subclassing.
  - Subclasses can redefine the variant parts of a behavior without changing the invariant parts.

## Applicability



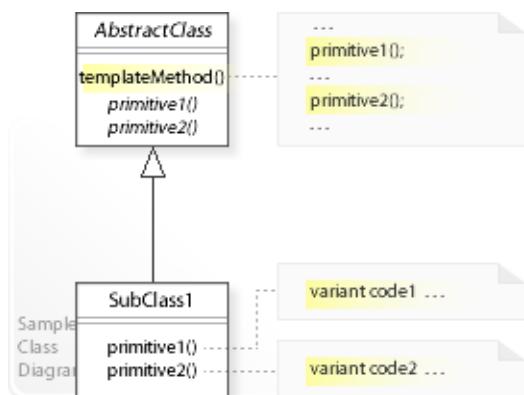
## Design Problems

- **Redefining Parts of a Behavior**
  - How can the invariant parts of a behavior be implemented once so that subclasses can implement the variant parts?
  - How can subclasses redefine certain parts of a behavior without changing the other parts?
- **Controlling Subclassing**
  - How can a class control the way it is subclassed (inversion of control)?
  - How can subclasses extend a behavior only at specific points (hooks)?

## Refactoring Problems

- **Duplicated Code**
  - How can common behavior among subclasses be factored out and localized (generalized) in a common class?  
*Form Template Method (205) [JKerievsky05]*  
"[...] common behavior among subclasses should be factored and localized in a common class to avoid code duplication." [GoF, p326]

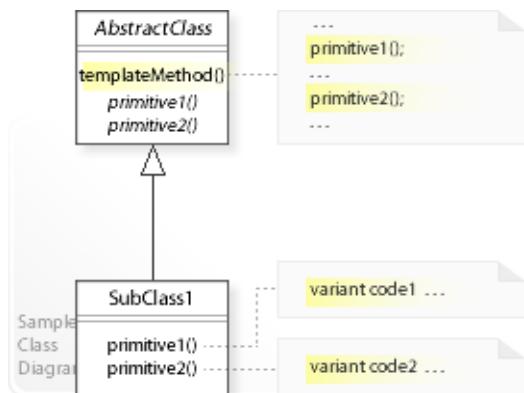
## Structure, Collaboration



### Static Class Structure

- *AbstractClass*
  - Defines a `templateMethod()` operation that defines the skeleton of a behavior by implementing the common behavior and calling abstract `primitive1()` and `primitive2()` operations that `SubClass1` implement.
  - Defines abstract `primitive` operations for certain (variant) parts of a behavior.
- *SubClass1, ...*
  - Implement the abstract `primitive` operations.

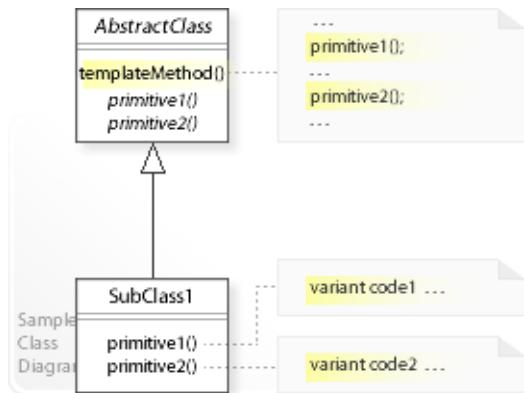
## Consequences



### Advantages (+)

- Code Reuse
  - "Template methods are a fundamental technique for code reuse. They are particularly important in class libraries because they are the means for factoring out common behavior in library classes." [GoF, p327]
- Inversion of Control
  - Template methods lead to an *inversion of control*. "This refers to how a parent class calls the operations of a subclass and not the other way around." [GoF, p327]
  - This is what distinguishes a *framework* from a *toolkit*:
  - When using a toolkit (reusable classes),  
we call the code we want to reuse.
  - When using a framework (reusable apps),  
we write the code the framework calls.

## Implementation

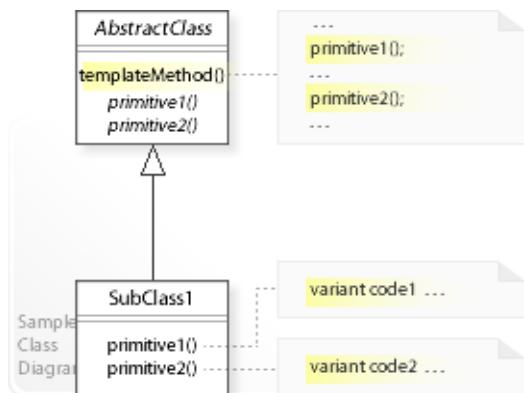


### Implementation Issues

- **Different Kinds of Operations**

- "To reuse an abstract class effectively, subclass writers must understand which operations are designed for overriding." [GoF, p328]
- *Primitive operations* - abstract operations that *must* be implemented by subclasses; or concrete operations that provide a default implementation and *can* be redefined by subclasses if necessary.  
Primitive operations can be declared *protected* to enable subclassing over package boundaries but keeping clients from calling them directly.
- *Final operations* - concrete operations that *can not* be overridden by subclasses.
- *Hook operations* - concrete operations that do nothing by default and *can* be redefined by subclasses if necessary.
- *Template methods* themselves can be declared *final* so that they can not be overridden.

## Sample Code 1

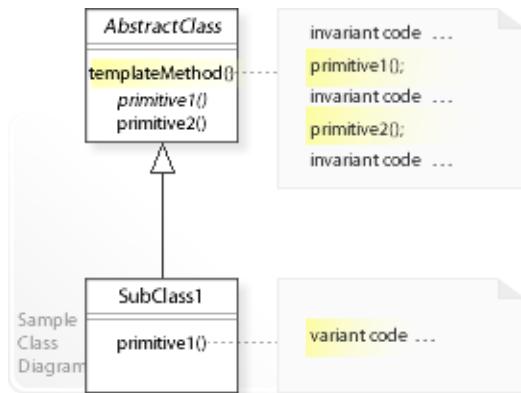


**Basic Java code for implementing the sample UML diagram.**

```
1 package com.sample.templatemethod.basic;
2 public abstract class AbstractClass {
3     //
4     protected abstract void primitive1();
5     protected abstract void primitive2();
6     //
7     public final void templateMethod() {
8         //
9         primitive1();
10        //
11        primitive2();
12        //
13    }
14 }

1 package com.sample.templatemethod.basic;
2 public class SubClass1 extends AbstractClass {
3     protected void primitive1() {
4         //
5     }
6     protected void primitive2() {
7         //
8     }
9 }
```

## Sample Code 2

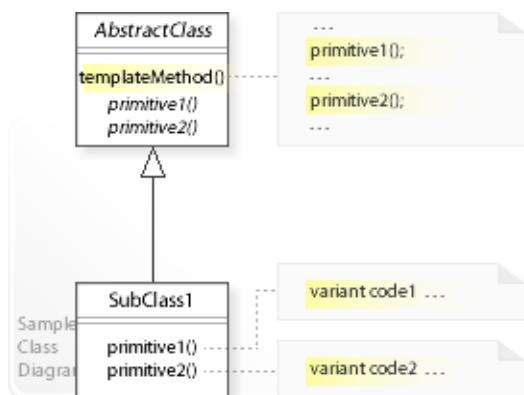


### Template method with abstract and concrete primitive operations.

```

1 package com.sample.templatemethod.steps;
2 public abstract class AbstractClass {
3     // Abstract primitive operation:
4     // - provides no default implementation
5     // - must be implemented (overridden).
6     protected abstract void primitive1();
7     //
8     // Concrete primitive operation:
9     // - provides a default implementation
10    // - can be changed (overridden) optionally.
11    protected void primitive2() {
12        // variant code ...
13    }
14    public final void templateMethod() {
15        // invariant code ...
16        primitive1(); // calling primitive1 (variant code)
17        // invariant code ...
18        primitive2(); // calling primitive2 (variant code)
19        // invariant code ...
20    }
21 }
2
1 package com.sample.templatemethod.steps;
2 public class SubClass1 extends AbstractClass {
3     protected void primitive1() {
4         // variant code ...
5     }
6 }
  
```

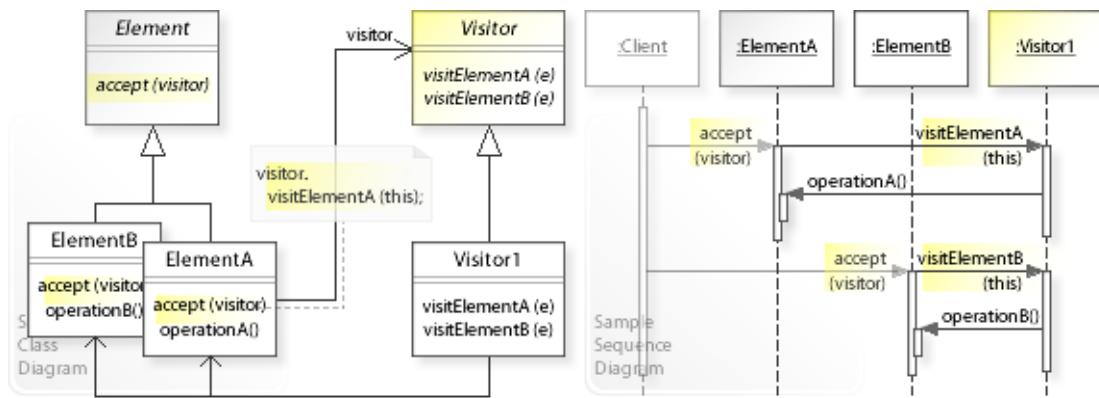
## Related Patterns



### Key Relationships (see also Overview)

- **Strategy - Template Method - Subclassing**
  - Strategy provides a way to change the algorithm/behavior of an object at run-time.
  - Template Method provides a way to control how subclasses change the algorithm/behavior of a class at compile-time.
  - Subclassing is the standard way to change the algorithm/behavior of a class at compile-time.
- **Template Method - Factory Method**
  - A template method's primitive operation that is responsible for creating an object is a factory method.

## Intent



The intent of the Visitor design pattern is to:

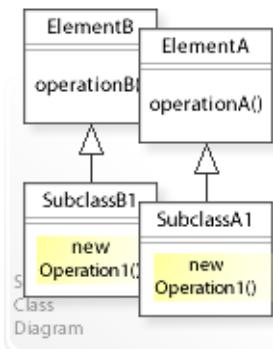
**"Represent an operation to be performed on the elements of an object structure."**

**Visitor lets you define a new operation without changing the classes of the elements on which it operates."** [GoF]

See Problem and Solution sections for a more structured description of the intent.

- The Visitor design pattern solves problems like:
  - *How can new operations be defined for the elements of an object structure without changing the classes of the elements?*
- For example, an object structure that represents the components of a technical equipment (Bill of Materials).  
Many different applications use (share) the object structure, and it should be possible to define arbitrary new operations (for example, calculating total prices, computing inventory, etc.) independently from (without having to extend) the classes of the object structure.
- The Visitor pattern describes how to solve such problems:
  - *Represent an operation to be performed on the elements of an object structure.*  
Define a separate **Visitor** object that encapsulates an operation to be performed on the elements of an object structure.
  - Define a dispatching operation `accept(visitor)` on each **Element** class that delegates the operation to be performed on an element to a **Visitor** object.
  - Clients traverse the object structure and call `accept(visitor)` on each element by passing in the needed **visitor** object.

## Problem



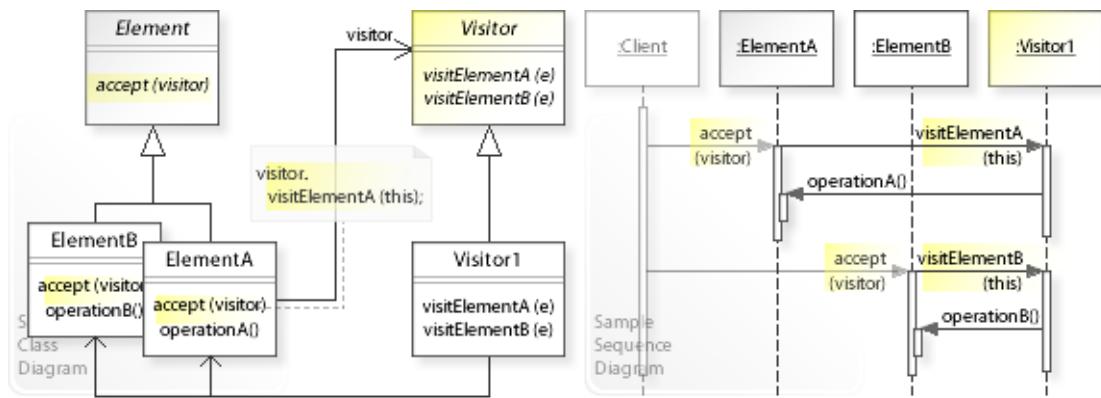
The Visitor design pattern solves problems like:

***How can new operations be defined for the elements of an object structure without changing the classes of the elements?***

See Applicability section for all problems Visitor can solve.

- One way to define a new operation (`newOperation1()`) for the elements of an object structure (`ElementA, ElementB, ...`) is to define a new subclass for each element class (`SubclassA1, SubclassB1, ...`) and implement the new operation.  
This makes it hard to define (many) new operations for an object structure that contains (many) different classes. "The problem here is that distributing all these operations across the various node [element] classes leads to a system that's hard to understand, maintain, and change." [GoF, p331]
- *That's the kind of approach to avoid if we want to define new operations for the elements of an object structure without having to change/extend the element classes.*
- For example, an object structure that represents the components of a technical equipment (Bill of Materials).  
Many different applications use (share) the object structure, and it should be possible for each application to define individual operations (for example, for calculating total prices, computing inventory, etc.) independently from (without having to extend) the classes of the object structure (see Sample Code / Example 2).

## Solution



The Visitor pattern describes a solution:

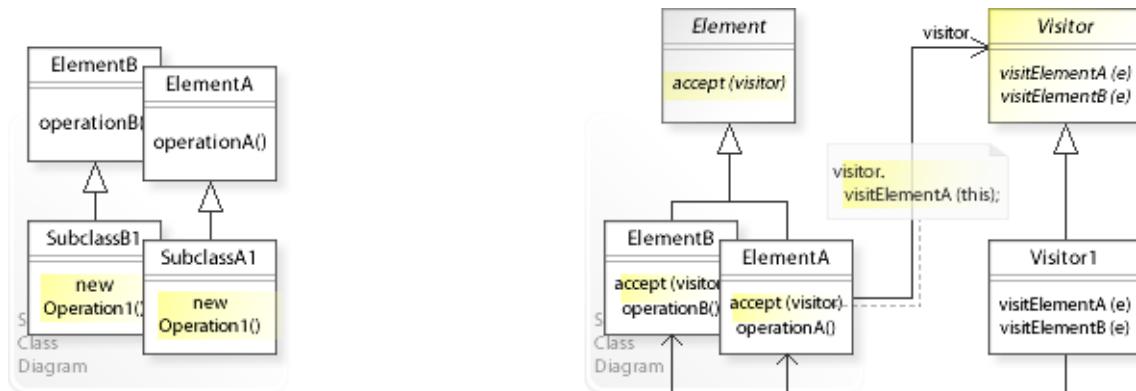
**Define separate visitor objects that encapsulate different operations to be performed on the elements of an object structure.**

**Define a dispatching operation `accept(visitor)` for each Element class that delegates the operation to the accepted visitor object.**

Describing the Visitor design in more detail is the theme of the following sections. See Applicability section for all problems Visitor can solve.

- The key idea in this pattern is to delegate the operation to be performed on an element to a **Visitor** object by means of the *double-dispatch* operation `accept(visitor)`.  
"This is the key to the Visitor pattern. The operation that gets executed depends on both the type of Visitor and the type of Element it visits." [GoF, p339]
- **Define separate visitor objects:**
  - Define an interface for *visiting* (performing an operation on) **Element** classes:  
`Visitor | visitElementA(e), visitElementB(e), ...`  
"We'll use the term **visitor** to refer generally to classes of objects that "visit" other objects during a traversal and do something appropriate." [GoF, p74]
  - Define classes (**Visitor1**,...) that implement the **Visitor** interface.
- This enables *compile-time* flexibility (via class inheritance).  
"You create a new operation by adding a new subclass to the visitor class hierarchy." [GoF, p333]
- **Define a dispatching operation `accept(visitor)` for each Element class** that delegates the operation (to be performed on an element) to the "accepted" **Visitor** object. For example, the dispatching operation of the **ElementA** class delegates to `visitor.visitElementA(this)`. See also Collaboration and Implementation.
- This enables *run-time* flexibility (via object composition). Clients traverse the object structure and call `accept(visitor)` on each element by passing in one of (many) different **Visitor** objects.

## Motivation 1



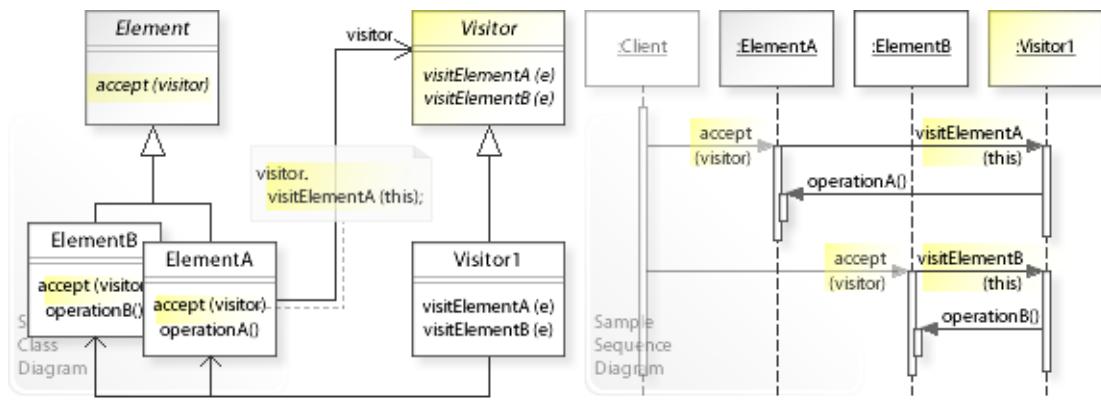
Consider the left design (problem):

- A new operation is distributed across the `Element` classes.
  - A new operation (`newOperation1()`) is defined by extending each `Element` class (`ElementA()`, `ElementB()`, ...).
  - This makes it hard to add new operations independently from (without having to extend) the `Element` classes.
  - "The problem here is that distributing all these operations across the various node [element] classes leads to a system that's hard to understand, maintain, and change." [GoF, p331]

Consider the right design (solution):

- A new operation is encapsulated (centralized) in a separate `Visitor` class.
  - A new operation for each `Element` class (`visitElementA()`, `visitElementB()`, ...) is defined in one class (`Visitor1`).
  - This makes it easy to add new operations independently from (without having to extend) the `Element` classes.
  - "You create a new operation by adding a new subclass to the visitor class hierarchy." [GoF, p333]

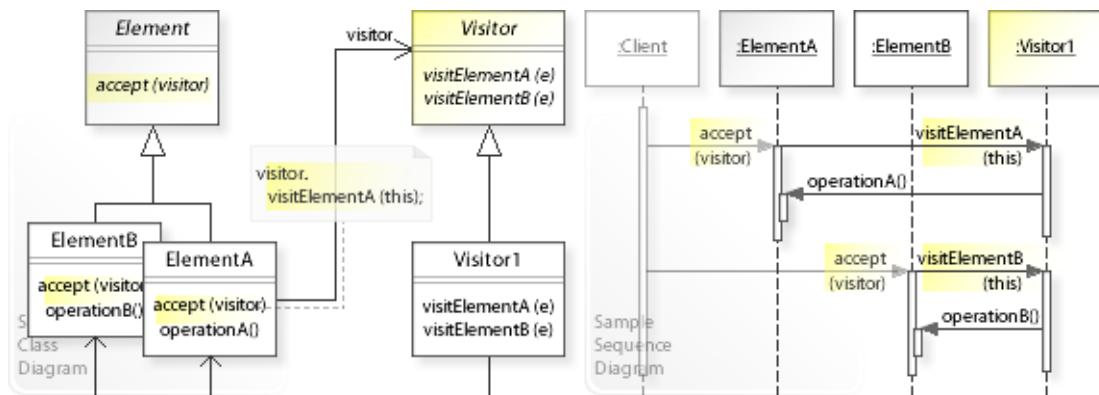
## Applicability



## Design Problems

- **Defining New Operations for Object Structures**
  - How can new operations be defined for elements of an object structure without changing the classes of the elements?
- **Flexible Alternative for Subclassing**
  - How can a flexible alternative be provided for defining new operations for an object structure by subclassing?

## Structure, Collaboration



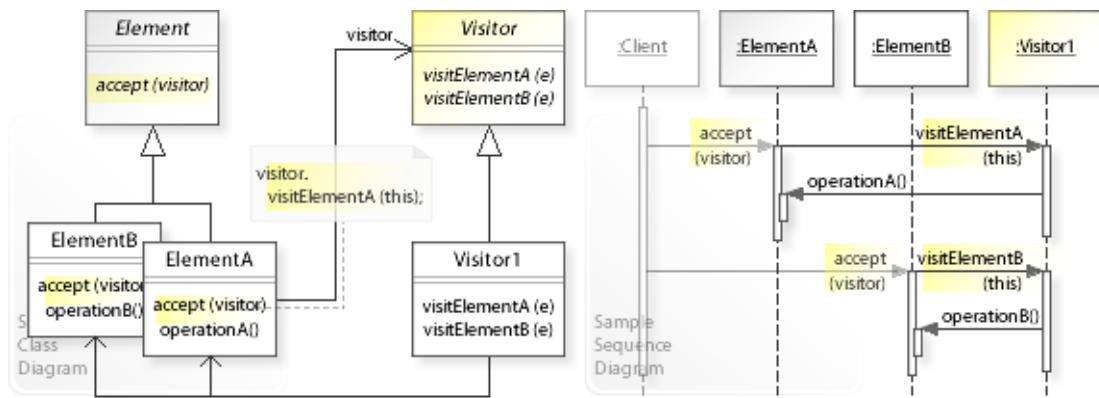
### Static Class Structure

- **Element**
  - Defines a dispatching operation (`accept(visitor)`).
- **ElementA, ElementB, ...**
  - Implement the dispatching operation.
  - For example, **ElementA** implements: `visitor.visitElementA(this)`.
- **Visitor**
  - Defines an interface for performing an operation on each element class, that is, for *visiting* each element class and do some work on it.
  - "We'll use the term **visitor** to refer generally to classes of objects that "visit" other objects during a traversal and do something appropriate." [GoF, p74]
- **Visitor1, ...**
  - Implement the **Visitor** interface.

### Dynamic Object Collaboration

- In this sample scenario, a **Client** object traverses an object structure (**ElementA**, **ElementB**, ...) and calls `accept(visitor)` on each element.
- The interaction starts with the **Client** that calls `accept(visitor)` on the **ElementA** object.
- **ElementA** calls `visitElementA(this)` on the passed in (accepted) **Visitor** object (of type **Visitor1**).
- **ElementA** passes `itself(this)` to the **Visitor1** so that **Visitor1** can visit (call back) **ElementA** and do some work on it.
- **Visitor1** does its work on **ElementA** (by calling `operationA()`, for example) and returns to the **ElementA**, which in turn returns to the **Client**.
- Thereafter, the **Client** calls `accept(visitor)` on the **ElementB** object.
- **ElementB** calls `visitElementB(this)` on the passed in (accepted) **Visitor** object (of type **Visitor1**), which does its work on **ElementB**.
- See also Sample Code / Example 1.

## Consequences



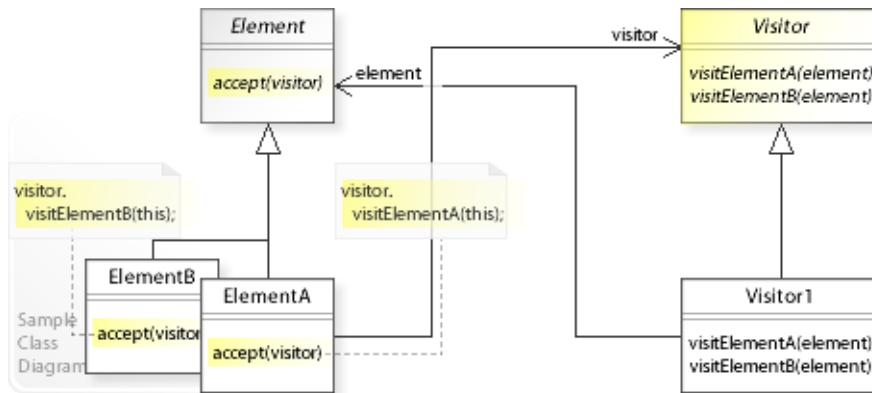
### Advantages (+)

- Makes adding new operations easy.
  - "You create a new operation by adding a new subclass to the visitor class hierarchy." [GoF, p333]
- Enables visiting elements of different types across inheritance hierarchies.
  - Visitor can visit elements that do not have a common interface, i.e., it can visit different types of elements (**ElementA**, **ElementB**, ...), that do not have to be related through inheritance.
- Makes accumulating state easy.
  - Visitor makes it easy to accumulate state while traversing an object structure.
  - It eliminates the need to pass the state to operations that perform the accumulation. The state is accumulated and stored in the visitor object (see Sample Code / Example 2 / Pricing and Inventory Visitor).

### Disadvantages (-)

- Requires extending the visitor interface to support new element classes.
  - The visitor interface must be extended to support new element classes in the object structure.
  - Therefore, the Visitor pattern should be used only when the object structure is stable and new element classes aren't added frequently.
- May require extending the element interfaces.
  - The element interfaces (**ElementA|operationA()**, **ElementB|operationB()**, ...) may have to be extended and may get bloated to let all visitors do their work and access the needed functionality and data.
- Introduces additional level of indirection.
  - The pattern achieves flexibility by introducing separate visitor objects and a double-dispatch mechanism, which complicates the design.

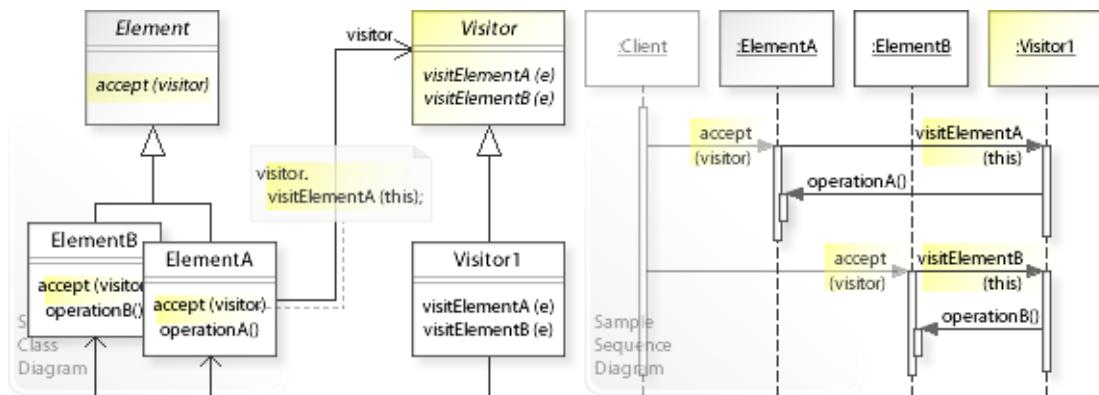
## Implementation



### Implementation Issues

- **Dispatching Operation `accept(visitor)`**
  - Each **Element** class of the object structure defines an `accept(visitor)` operation that delegates performing the operation to the passed-in ("accepted") **Visitor** object *and* the `visit` operation that corresponds to the **Element** class:  
`ElementA` delegates to `visitor.visitElementA(this)`,  
`ElementB` delegates to `visitor.visitElementB(this)`, ...
  - The `accept(visitor)` operation is a *double-dispatch* operation:  
 "This is the key to the Visitor pattern. The operation that gets executed depends on both the type of Visitor and the type of Element it visits." [GoF, p339]
  - The element itself (`this`) is passed to the visitor to let the visitor visit (call back and do its work on) this element.
- **Visitor Interface**
  - The **Visitor** interface defines a `visit` operation for each element class that should be visited.  
 The interface may have to change when new element classes are added to the object structure that should be visited.
- **Element Interfaces**
  - The **Element** interfaces may have to be extended to let all visitors do their work and access the needed data and functionality.

## Sample Code 1



Basic Java code for implementing the sample UML diagrams.

```

1 package com.sample.visitor.basic;
2 import java.util.ArrayList;
3 import java.util.List;
4 public class Client {
5     // Running the Client class as application.
6     public static void main(String[] args) {
7         // Setting up an object structure.
8         List<Element> elements = new ArrayList<Element>();
9         elements.add(new ElementA());
10        elements.add(new ElementB());
11        // Creating a visitor.
12        Visitor visitor = new Visitor1();
13        // Traversing the object structure and
14        // calling accept(visitor) on each element.
15        for (Element element : elements) {
16            element.accept(visitor);
17        }
18    }
19 }
```

Visitor1: visiting (doing work on) ElementA: Hello WorldA!  
 Visitor1: visiting (doing work on) ElementB: Hello WorldB!

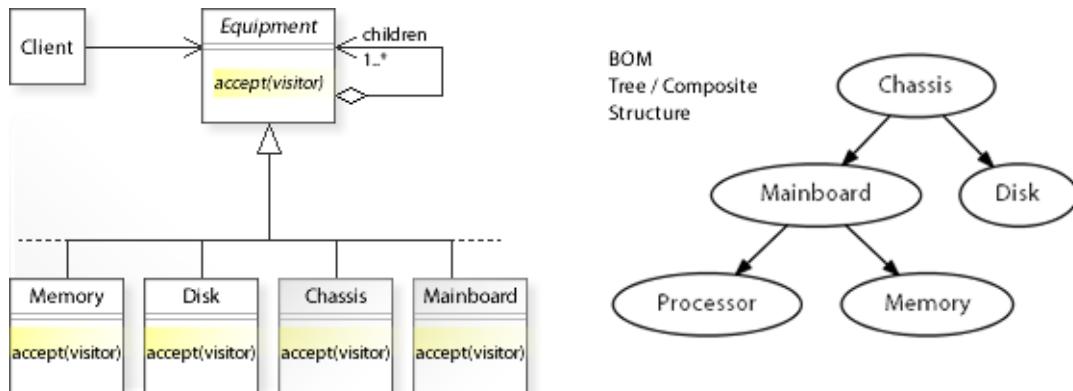
```

1 package com.sample.visitor.basic;
2 public abstract class Element {
3     public abstract void accept(Visitor visitor);
4 }
5
6 package com.sample.visitor.basic;
7 public class ElementA extends Element {
8     public void accept(Visitor visitor) {
9         visitor.visitElementA(this);
10    }
11    public String operationA() {
12        return "A";
13    }
14 }
15
16 package com.sample.visitor.basic;
17 public class ElementB extends Element {
18     public void accept(Visitor visitor) {
19         visitor.visitElementB(this);
20     }
21     public String operationB() {
22         return "B";
23     }
24 }
```

```
1 package com.sample.visitor.basic;
2 public abstract class Visitor {
3     public abstract void visitElementA(ElementA e);
4     public abstract void visitElementB(ElementB e);
5 }

1 package com.sample.visitor.basic;
2 public class Visitor1 extends Visitor {
3     public void visitElementA(ElementA element) {
4         System.out.println(
5             "Visitor1: visiting (doing work on) ElementA: "
6             + "Hello World" + element.operationA() + "!");
7     }
8     public void visitElementB(ElementB element) {
9         System.out.println(
10            "Visitor1: visiting (doing work on) ElementB: "
11            + "Hello World" + element.operationB() + "!");
12    }
13 }
```

## Sample Code 2



### BOM Bill of Materials / Using a pricing and inventory visitor.

The pricing visitor calculates number of components and total prices.

The inventory visitor calculates the inventory of components.

The BOM is implemented as tree (composite) structure.

See also Composite design pattern, Sample Code / Example 2 (calculating total prices).

```

1 package com.sample.visitor.bom;
2 import java.io.IOException;
3 public class Client {
4     // Running the Client class as application.
5     public static void main(String[] args) throws Exception {
6         // Building a BOM tree (composite structure).
7         Equipment mainboard = new Mainboard("Mainboard", 100);
8         mainboard.add(new Processor("Processor", 100));
9         mainboard.add(new Memory("Memory ", 100));
10        Equipment chassis = new Chassis("Chassis ", 100);
11        chassis.add(mainboard);
12        chassis.add(new Disk("Disk      ", 100));
13        //
14        System.out.println("Traversing the BOM using a pricing visitor: ");
15        //
16        PricingVisitor pricingVisitor = new PricingVisitor();
17        chassis.accept(pricingVisitor);
18        System.out.println(
19            " Number of components: " + pricingVisitor.getNumberOfElements() +
20            "\n Total price       : " + pricingVisitor.getTotalPrice());
21        //
22        System.out.println("Traversing the BOM using an inventory visitor: ");
23        //
24        InventoryVisitor inventoryVisitor = new InventoryVisitor(new Inventory());
25        chassis.accept(inventoryVisitor);
26    }
27 }
```

Traversing the BOM using a pricing visitor:

```

Number of components: 5
Total price       : 500

```

Traversing the BOM using an inventory visitor:

```

Inventory for Processor: 10
Inventory for Memory   : 10
Inventory for Mainboard: 10
Inventory for Disk     : 10
Inventory for Chassis  : 10

```

```

1 package com.sample.visitor.bom;
2 import java.util.ArrayList;
3 import java.util.Iterator;
4 import java.util.List;
5 public abstract class Equipment {
6     private String name;
7     List<Equipment> children = new ArrayList<Equipment>();
8     public Equipment(String name) {
9         this.name = name;

```

```
10      }
11      //
12      public abstract void accept(EquipmentVisitor visitor);
13      //
14      public String getName() {
15          return this.name;
16      ;
17      public boolean add(Equipment e) {
18          return children.add(e);
19      }
20      public Iterator<Equipment> iterator() {
21          return children.iterator();
22      }
23      public int getChildCount() {
24          return children.size();
25      }
26  }

1 package com.sample.visitor.bom;
2 public abstract class EquipmentVisitor {
3     public abstract void visitChassis(Chassis e);
4     public abstract void visitMainboard(Mainboard e);
5     public abstract void visitProcessor(Processor e);
6     public abstract void visitMemory(Memory e);
7     public abstract void visitDisk(Disk e);
8 }

1 package com.sample.visitor.bom;
2 public class PricingVisitor extends EquipmentVisitor {
3     private int count = 0;
4     private long sum = 0;
5     public void visitChassis(Chassis e) {
6         count++;
7         sum += e.getCostPrice();
8     }
9     public void visitMainboard(Mainboard e) {
10        count++;
11        sum += e.getBasicPrice();
12    }
13    public void visitProcessor(Processor e) {
14        count++;
15        sum += e.getPurchaseCost();
16    }
17    public void visitMemory(Memory e) {
18        count++;
19        sum += e.getPrice();
20    }
21    public void visitDisk(Disk e) {
22        count++;
23        sum += e.getUnitPrice();
24    }
25    public int getNumberOfElements() {
26        return count;
27    }
28    public long getTotalPrice() {
29        return sum;
30    }
31 }

1 package com.sample.visitor.bom;
2 public class InventoryVisitor extends EquipmentVisitor {
3     private Inventory inventory;
4     public InventoryVisitor(Inventory inventory) {
5         this.inventory = inventory;
6     }
7     public void visitChassis(Chassis e) {
8         inventory.operation(e);
9     }
10    public void visitMainboard(Mainboard e) {
11        inventory.operation(e);
12    }
13    public void visitProcessor(Processor e) {
14        inventory.operation(e);
15    }
16    public void visitMemory(Memory e) {
17        inventory.operation(e);
18    }
19 }
```

```
18      }
19      public void visitDisk(Disk e) {
20          inventory.operation(e);
21      }
22  }
```

```
1 package com.sample.visitor.bom;
2 public class Inventory {
3     private int quantity = 10;
4     public void operation(Equipment e) {
5         // Calculating inventory (quantity in stock).
6         // ...
7         System.out.println(" Inventory for " + e.getName() + ":" + quantity);
8     }
9 }

1 package com.sample.visitor.bom;
2 public class Chassis extends Equipment { // Composite
3     private long price;
4     public Chassis(String name, long price) {
5         super(name);
6         this.price = price;
7     }
8     public void accept(EquipmentVisitor visitor) {
9         for (Equipment child : children) {
10            child.accept(visitor);
11        }
12        visitor.visitChassis(this);
13    }
14    public long getCostPrice() { // Net cost price in cents
15        return price;
16    }
17 }

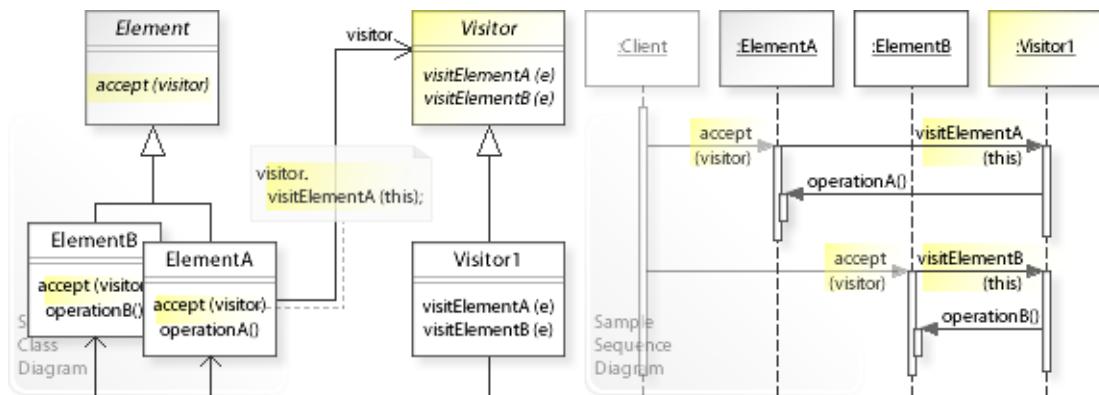
1 package com.sample.visitor.bom;
2 public class Mainboard extends Equipment { // Composite
3     private long price;
4     public Mainboard(String name, long price) {
5         super(name);
6         this.price = price;
7     }
8     public void accept(EquipmentVisitor visitor) {
9         for (Equipment child : children) {
10            child.accept(visitor);
11        }
12        visitor.visitMainboard(this);
13    }
14    public long getBasicPrice() { // Basic price in cents
15        return price;
16    }
17 }

1 package com.sample.visitor.bom;
2 public class Processor extends Equipment { // Leaf
3     private long price;
4     public Processor(String name, long price) {
5         super(name);
6         this.price = price;
7     }
8     public void accept(EquipmentVisitor visitor) {
9         visitor.visitProcessor(this);
10    }
11    public long getPurchaseCost() { // Cost of purchase in cents
12        return price;
13    }
14 }
```

```
1 package com.sample.visitor.bom;
2 public class Memory extends Equipment { // Leaf
3     private long price;
4     public Memory(String name, long price) {
5         super(name);
6         this.price = price;
7     }
8     public void accept(EquipmentVisitor visitor) {
9         visitor.visitMemory(this);
10    }
11    public long getPrice() { // Unit price in cents
12        return price;
13    }
14 }
```

```
1 package com.sample.visitor.bom;
2 public class Disk extends Equipment { // Leaf
3     private long price;
4     public Disk(String name, long price) {
5         super(name);
6         this.price = price;
7     }
8     public void accept(EquipmentVisitor visitor) {
9         visitor.visitDisk(this);
10    }
11    public long getUnitPrice() { // Unit price in cents
12        return price;
13    }
14 }
```

## Related Patterns

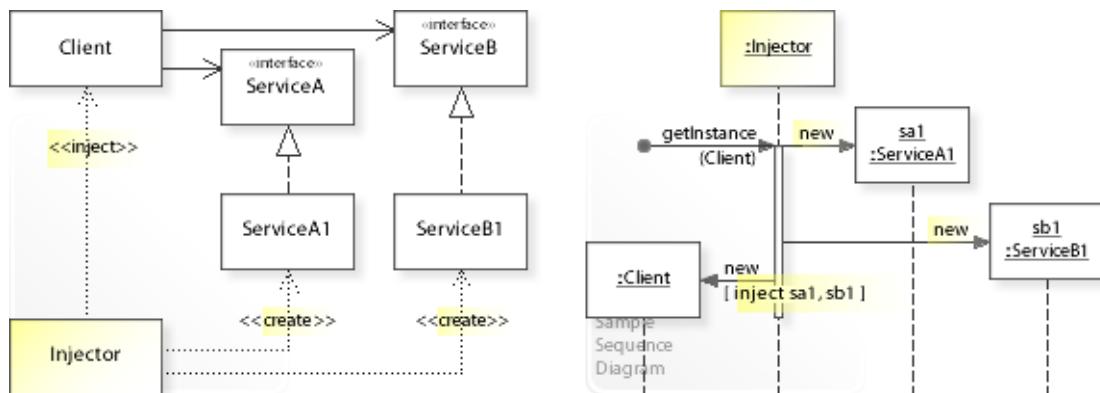


### Key Relationships (see also Overview)

- **Composite - Builder - Iterator - Visitor - Interpreter**
  - Composite provides a way to represent a part-whole hierarchy as a tree (composite) object structure.
  - Builder provides a way to create the elements of an object structure.
  - Iterator provides a way to traverse the elements of an object structure.
  - Visitor provides a way to define new operations for the elements of an object structure.
  - Interpreter represents a sentence in a simple language as a tree (composite) object structure (abstract syntax tree).

## **Part V. GoF Design Patterns Update**

## Intent



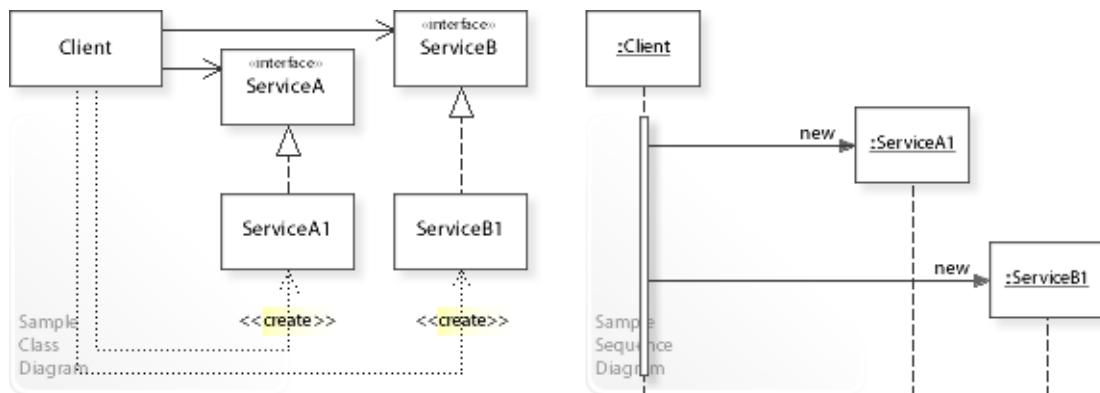
The intent of the Dependency Injection design pattern is to:

**Separate object creation from an application. Dependency Injection makes an application independent of how its objects are created.**

See Problem and Solution sections for a more structured description of the intent.

- The Dependency Injection design pattern solves problems like:
  - *How can a class be independent of how the objects it requires are created?*
  - *How can a class be configured with the objects it requires?*
- For example, classes in an order processing application require many other objects like customers, products, and inventory.  
A class should avoid instantiating concrete classes directly so that it can be configured with the objects it requires at run-time.
- The Dependency Injection pattern describes how to solve such problems:
  - *Define a separate Injector object that creates the objects a class requires and injects them into the class.*
  - A class gets the objects it requires injected at run-time and is independent of how the objects are created.

## Problem



The Dependency Injection design pattern solves problems like:

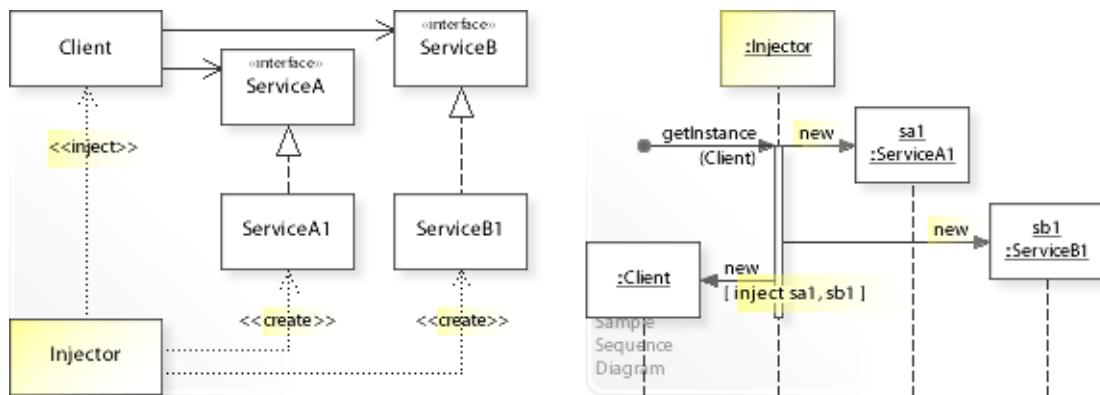
**How can a class be independent of how its objects are created?**

**How can a class be configured with the objects it requires?**

See Applicability section for all problems Dependency Injection can solve.

- An inflexible way to create objects is to instantiate concrete classes (`new ServiceA1()`, `new ServiceB1()`) directly within a class (`Client`).  
This commits the class to particular objects and makes it impossible to change the instantiation later independently from (without having to change) the class.  
It stops the class from being reusable with different objects, and it makes the class hard to test because real objects can't be replaced with mock objects.  
Ultimately, classes that include object creation are harder to implement (especially if objects have further dependencies, which in turn have dependencies, and so on), change, test, and reuse.
- *That's the kind of approach to avoid if we want that a class is independent of how the objects it requires (depends on) are created.*
- For example, classes in an order processing application require many other objects like customers, products, and inventory.  
A class should avoid instantiating concrete classes directly so that it can be configured with the objects it requires at run-time.
- For example, supporting different configurations of a reusable application.  
Instantiating concrete classes throughout the application should be avoided so that the application can be configured with the required objects at run-time.

## Solution



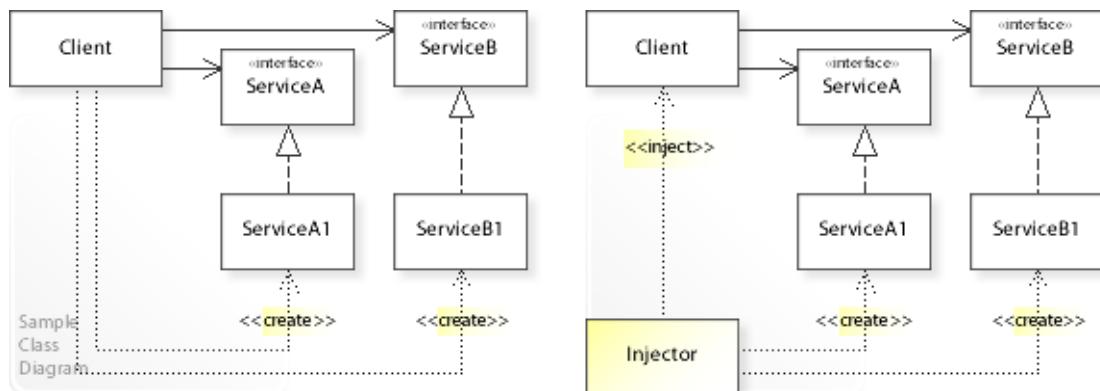
The Dependency Injection pattern describes a solution:

**Define a separate `Injector` object that creates the objects a class requires and injects them into the class.**

Describing the Dependency Injection design in more detail is the theme of the following sections. See Applicability section for all problems Dependency Injection can solve.

- The key idea in this pattern is to separate creating objects from classes that use them. A class is not responsible for creating the objects it uses. Instead, the objects are injected into the class at run-time. Hence it's often called *inversion of control*, which is a common feature of frameworks (see Template Method). []
- **Define a separate `Injector` object:**
  - The way objects are created (that is, the mapping of interfaces to implementations) is specified in separate (external) *configuration* files or objects: (`ServiceA -> ServiceA1`, `ServiceB -> ServiceB1`).
  - To let the injector do its work, a class must provide a *constructor* (and/or *setter methods*) through which the objects can be passed in (injected).
- A class can be implemented solely in terms of interfaces (`ServiceA`, `ServiceB`) while their implementations (`ServiceA1`, `ServiceB1`) are injected automatically at run-time (by an injector object). This greatly simplifies classes and makes them easier to implement, change, test, and reuse.

## Motivation 1



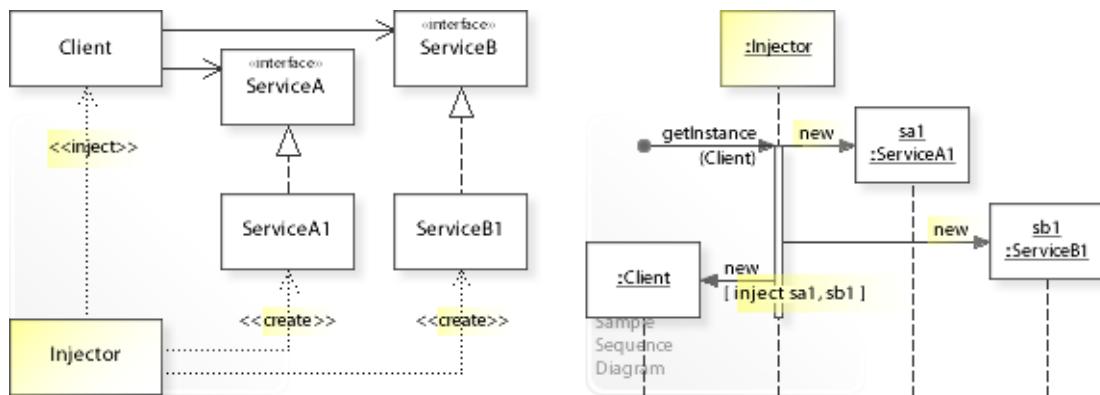
**Consider the left design (problem):**

- Hard-wired object creation.
  - Creating objects is implemented (hard-wired) directly within the classes (**Client**) that use the objects.
  - This makes it hard to change the way objects are created independently from (without having to change) the classes.
- Distributed object creation.
  - Creating objects is distributed across the classes of an application.

**Consider the right design (solution):**

- Separated object creation.
  - The way objects are created (the mapping of interfaces to implementations) is defined in separate configuration files.
  - This makes it easy to change the way objects are created independently from (without having to change) existing classes.
- Centralized object creation.
  - Creating objects is centralized in a single **Injector** class.

## Applicability



## Design Problems

- **Creating Objects**
  - How can an application be independent of how its objects are created?
  - How can object creation be separated from an application?
  - How can a class be configured with the objects it requires?
  - How can the objects a class requires be injected automatically at run-time?
- **Supporting Different Configurations**
  - How can the way objects are created (the mapping of interfaces to implementations) be specified in external configuration files?
- **Resolving Dependencies Recursively**
  - How can object dependencies be resolved recursively?

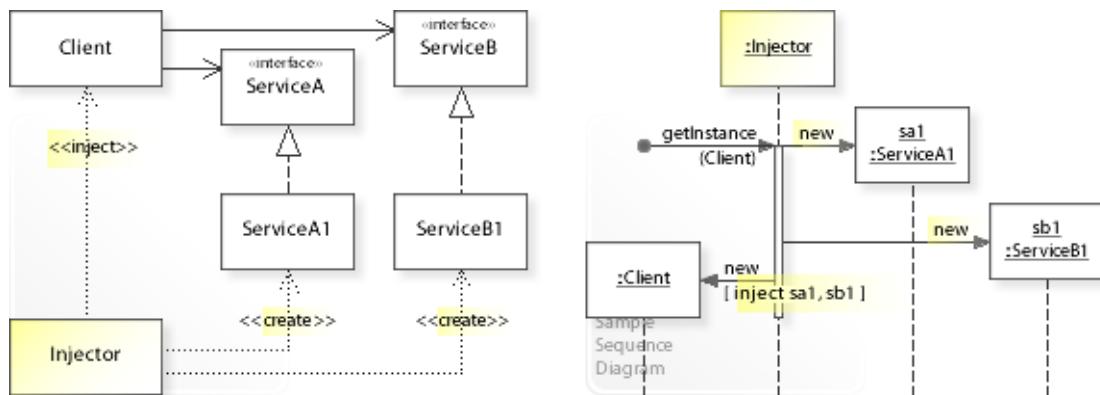
## Refactoring Problems

- **Inflexible Code**
  - How can instantiating concrete classes throughout an application (compile-time implementation dependencies) be refactored?
  - How can object creation that is distributed across an application be centralized?

## Testing Problems

- **Unit Testing**
  - How can a class be configured with mock objects instead of real objects so that the class can be unit tested in isolation?

## Structure, Collaboration



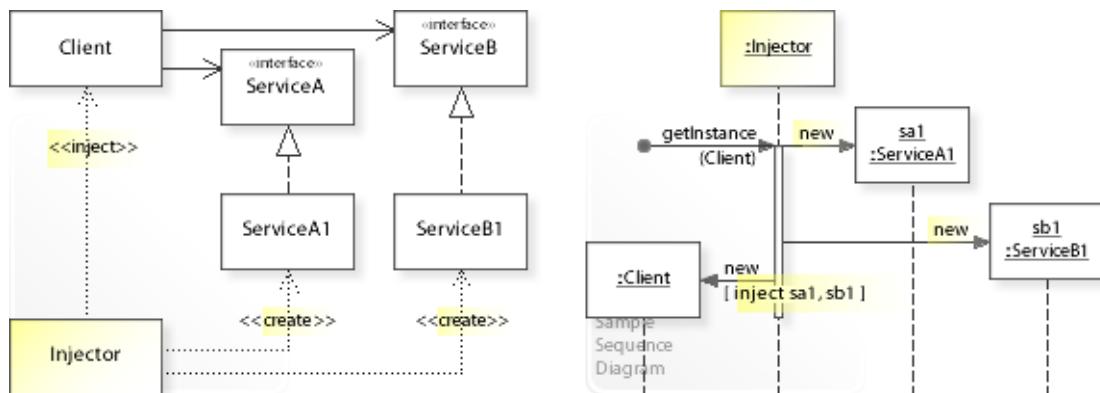
## Static Class Structure

- `Client`
  - Refers to (depends on) a `Service` interface.
  - Gets a `Service` object injected from the `Injector` and is independent of how the object is created (which concrete class is instantiated).
- `Service`
  - Defines an interface.
- `Service1`
  - Implements the `Service` interface.
- `Injector`
  - Creates and injects a `Service1` object by using a `Configuration` class.
- `Configuration`
  - Defines the mapping between interfaces and implementations (`Service -> Service1`).

## Dynamic Object Collaboration

- In this sample scenario, a `Client` object gets a `s1:Service1` object injected from an `Injector` object.  
Let's assume that the `Injector` uses a `Configuration` file that maps the `Service` interface to the `Service1` implementation.
- The interaction starts with some object that calls `getInstance(Client)` on the `Injector`. Usually, this call comes from a main class (the class with the `main()` method that starts an application) that needs an instance of the specified type (`Client`).
- The `Injector` creates a `s1:Service1` object.
- Thereafter, the `Injector` creates a `Client` object and injects `s1`.
- The `Client` object can then use (perform an operation on) the `s1` object.
- See also Sample Code.

## Consequences

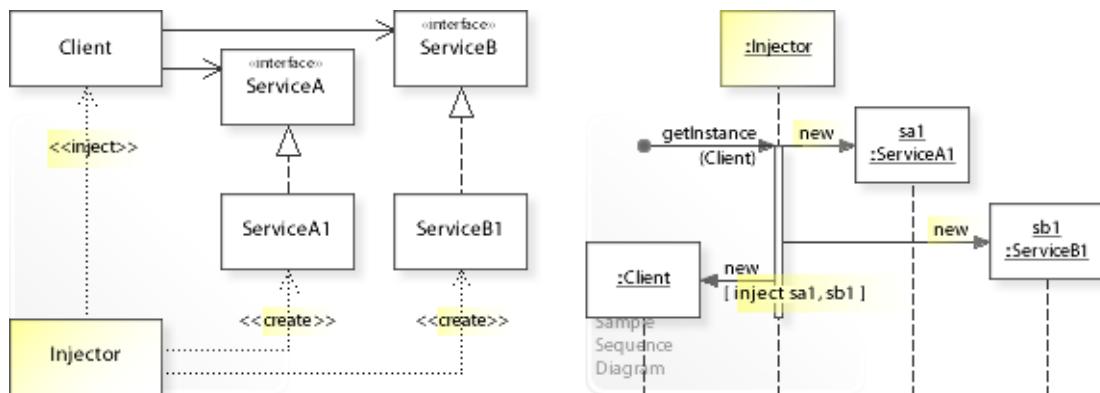


### Advantages (+)

- Avoids implementation dependencies.
  - Classes get their objects injected at run-time and are independent of (do not know) which concrete classes are instantiated.
- Greatly simplifies classes.
  - Classes get their objects injected automatically at run-time instead of having to create them directly, which makes the classes easier to implement, change, test, and reuse.
  - Classes can refer to other objects through their interfaces and get instances of concrete classes injected at run-time.
- Makes changing the configuration of an application easy.
  - Because the way objects are created is defined in separate configuration files, the configuration of an application can be changed easily by using different configuration files.
- Ensures objects are configured properly.
  - When using constructor injection, the dependencies of an object are created and injected before it can be used.

### Disadvantages (-)

## Implementation



## Implementation Issues

- **Implementation Variants**

- To let the injector do its work, classes must provide a way through which their dependencies can be passed in. There are two main implementation variants:

- **Variant1: Constructor Injection**

- Classes define a constructor through which dependencies can be passed in.

```
class Client ...
    private Service service;
    public Client(Service service) {
        this.service = service;
    }
```

- **Variant2: Setter Injection**

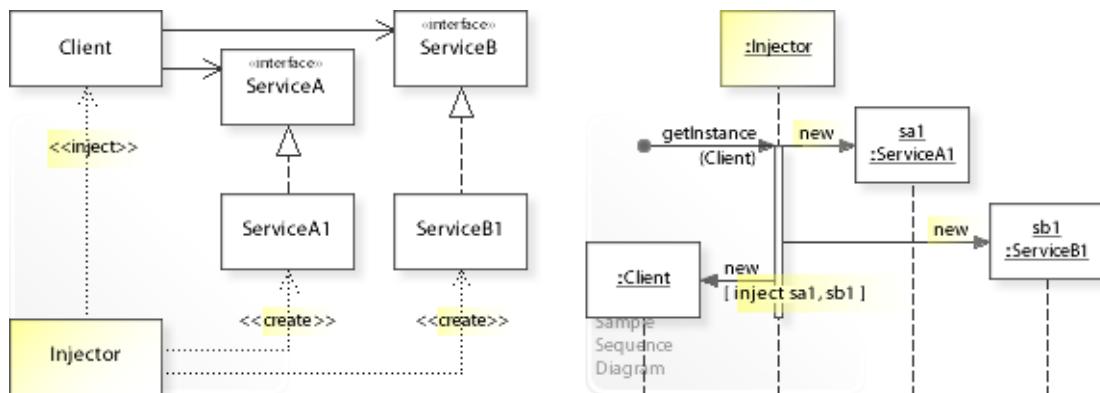
- Classes define a setter method for each dependency.

```
class Client ...
    private Service service;
    public void setService(Service service) {
        this.service = service;
    }
```

- **Constructor Injection versus Setter Injection**

- Constructor injection is a clear way to inject the dependencies of an object when it is created. It ensures that an object is configured properly before it can be used and that the dependencies can't be changed after the object is created.
- With setter injection, it can't be ensured that an object is configured before it is used because dependencies can be injected at any time after the object is created.

## Sample Code 1



Basic Java code by using the open source Google Guice Injector.

```

1 package com.sample.di.basic;
2 import com.google.inject.Guice;
3 import com.google.inject.Injector;
4 public class MyApp {
5     public static void main(String[] args) {
6         // Requesting an Injector object.
7         Injector injector = Guice.createInjector(new Configuration1());
8
9         // Requesting a Client object.
10        Client client = injector.getInstance(Client.class);
11
12        System.out.println(client.operation());
13    }
14 }

Hello World from Client!
Getting ServiceA1 ServiceB1 objects injected.

1 package com.sample.di.basic;
2 import com.google.inject.Inject;
3 public class Client {
4     private ServiceA sa;
5     private ServiceB sb;
6
7     @Inject
8     public Client(ServiceA sa, ServiceB sb) {
9         this.sa = sa;
10        this.sb = sb;
11    }
12    public String operation() {
13        return "Hello World from Client!\n"
14            + "Getting " + sa.getName() + sb.getName() + "objects injected.";
15    }
16 }

1 package com.sample.di.basic;
2 public interface ServiceA {
3     String getName();
4 }

1 package com.sample.di.basic;
2 public class ServiceA1 implements ServiceA {
3     public String getName() {
4         return "ServiceA1 ";
5     }
6 }

1 package com.sample.di.basic;
2 public interface ServiceB {
3     String getName();
4 }

1 package com.sample.di.basic;

```

```
2 public class ServiceB1 implements ServiceB {
3     public String getName() {
4         return "ServiceB1 ";
5     }
6 }

1 package com.sample.di.basic;
2 import com.google.inject.*;
3 public class Configuration1 extends AbstractModule {
4     @Override
5     protected void configure() {
6         // Mapping (binding) interfaces to implementations.
7         bind(ServiceA.class).to(ServiceA1.class);
8         bind(ServiceB.class).to(ServiceB1.class);
9     }
10 }

*****
Unit test classes.
*****

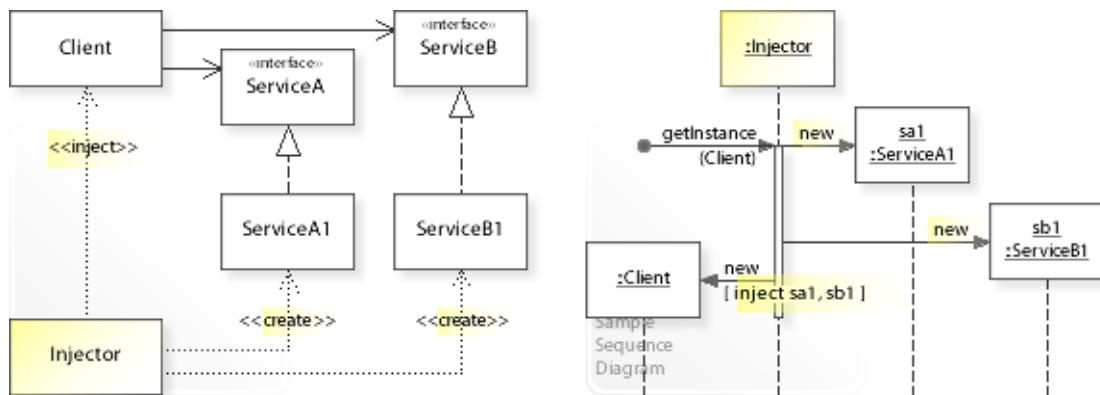
1 package com.sample.di.basic;
2 import com.google.inject.Guice;
3 import com.google.inject.Injector;
4 import junit.framework.TestCase;
5 public class ClientTest extends TestCase {
6     // Requesting an Injector object.
7     Injector injector = Guice.createInjector(new ConfigurationMock());
8
9     // Requesting a Client object.
10    Client client = injector.getInstance(Client.class);
11
12    public void testOperation() {
13        assertEquals("Hello World from Client!\n"
14                    + "Getting *ServiceAMock* *ServiceBMock* objects injected.",
15                    client.operation());
16    }
17    // More tests ...
18 }

1 package com.sample.di.basic;
2 public class ServiceAMock implements ServiceA {
3     public String getName() {
4         return "*ServiceAMock* ";
5     }
6 }

1 package com.sample.di.basic;
2 public class ServiceBMock implements ServiceB {
3     public String getName() {
4         return "*ServiceBMock* ";
5     }
6 }

1 package com.sample.di.basic;
2 import com.google.inject.*;
3 public class ConfigurationMock extends AbstractModule {
4     @Override
5     protected void configure() {
6         // Mapping (binding) interfaces to implementations.
7         bind(ServiceA.class).to(ServiceAMock.class);
8         bind(ServiceB.class).to(ServiceBMock.class);
9     }
10 }
```

## Related Patterns



### Key Relationships (see also Overview)

- **Abstract Factory - Dependency Injection**
  - Abstract Factory
    - A class requests the objects it requires from a factory object.
    - A class must know its factory.
  - Dependency Injection
    - A class accepts the objects it requires from an injector object.
    - A class has no knowledge of its injector.
- **Strategy - Dependency Injection**
  - Strategy
    - A class can be configured with a strategy object.
  - Dependency Injection
    - A class can be configured with the objects it requires.

## Appendix A. Bibliography

[JBloch08] [JB08]

Joshua Bloch.

**Effective Java.** Second Edition.

Sun Microsystems / Addison-Wesley, 2008.

[GBooch07] [GB07]

Grady Booch.

**Object-Oriented Analysis and Design with Applications.** Third Edition.

Addison-Wesley, 2007.

[MFowler99] [MF99]

Martin Fowler.

**Refactoring - Improving the Design of Existing Code.**

Addison-Wesley, 1999.

[MFowler03] [MF03]

Martin Fowler.

**Patterns of Enterprise Application Architecture.**

Addison-Wesley, 2003.

[MFowler11] [MF11]

Martin Fowler.

**Domain-Specific Languages.**

Addison-Wesley, 2011.

[MFowlerInjection] [MFI]

Martin Fowler.

**Inversion of Control Containers and the Dependency Injection pattern.**

<https://martinfowler.com/articles/injection.html>

[GoF]

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides.

**Design Patterns: Elements of Reusable Object-Oriented Software.**

Addison-Wesley, 1995.

[Java Collections]

Joshua Bloch.

**Java Collections Framework.**

Oracle, 2015: <http://docs.oracle.com/javase/tutorial/collections/index.html>

[Java Language Specification] [JLS12]

James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley.

**The Java Language Specification. Java SE 8 Edition.**

Oracle, 2015: <http://docs.oracle.com/javase/specs/jls/se8/html/index.html>

[JKerievsky05] [JK05]

Joshua Kerievsky.

**Refactoring to Patterns.**

Addison-Wesley, 2005.

[TParr07]

Terence Parr.

**The Definitive ANTLR Reference. Building Domain-Specific Languages.**

The Pragmatic Bookshelf, 2007.

*Language Implementation Patterns.*  
*Create Your Own Domain-Specific and General Programming Languages.*  
The Pragmatic Bookshelf, 2009.