The GoF Design Patterns Reference

Version 2.0 / 01.10.2017 / Generated 12.01.2018

Table of Contents

Preface

In software engineering, *design patterns* describe how to solve recurring design problems to design flexible and reusable object-oriented software.

w3sDesign presents the up-to-date version of the well-known GoF[1] design patterns in a compact and memory friendly way so that they can be learned and memorized as fast as possible.

We use a simple and consistent 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.

New design patterns that are widely used today but not included in the original twenty-three GoF design patterns will be added. This release starts with the *Dependency Injection* design pattern, and others will follow in next releases.

By working through individual design patterns, you will learn how to design objects that are easier to *implement*, *change*, *test*, and *reuse*.
Simple, ready-to-run code samples show how to implement design patterns by using object-oriented programming languages such as Java.

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!

[1] **Design Patterns: Elements of Reusable Object-Oriented Software.**
Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides.
Published October 1994. Copyright © 1995 by Addison-Wesley.
(The authors of the book are commonly referred to as "GoF" or "Gang of Four".)

Part I. Introduction

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

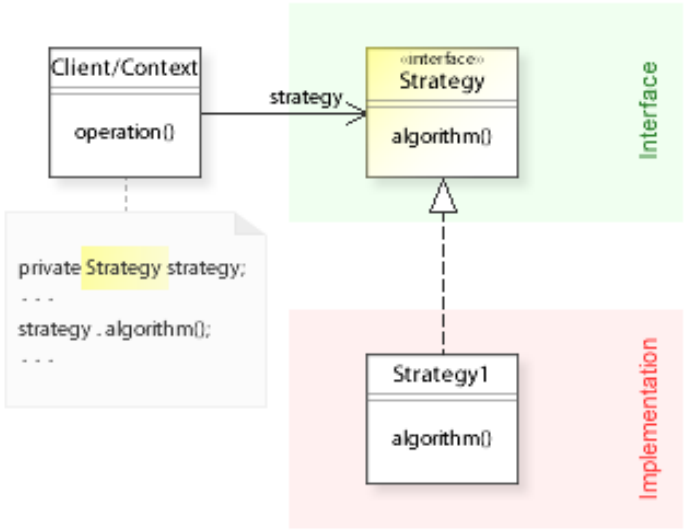


Figure 1 Strategy Design Pattern - Sample Class Diagram

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.

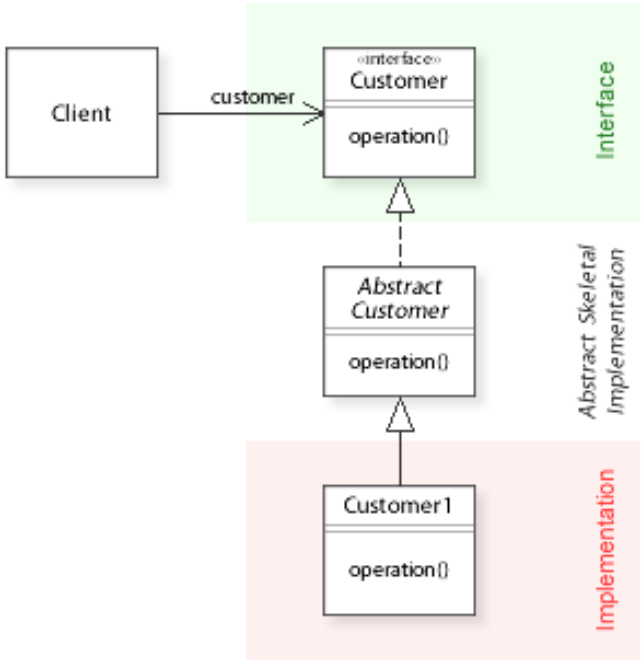


Figure 2 Interface Design - Sample Class Diagram

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]

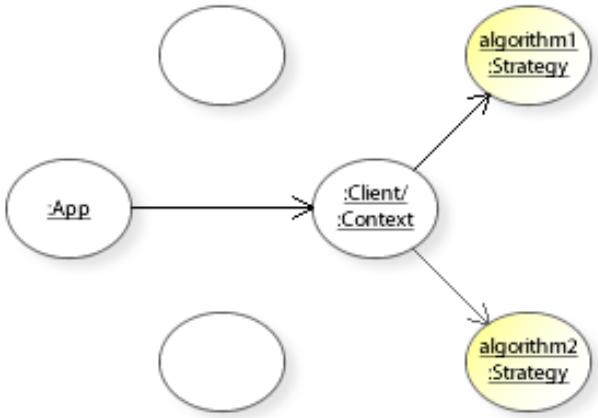


Figure 3 Strategy Design Pattern - Sample Object Collaboration

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 inteface, 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 respresentation 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.

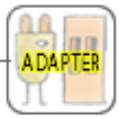
Defining a one-to-many dependency between objects
without making the objects tightly coupled.
Notifying an open-ended number of dependent objects.

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

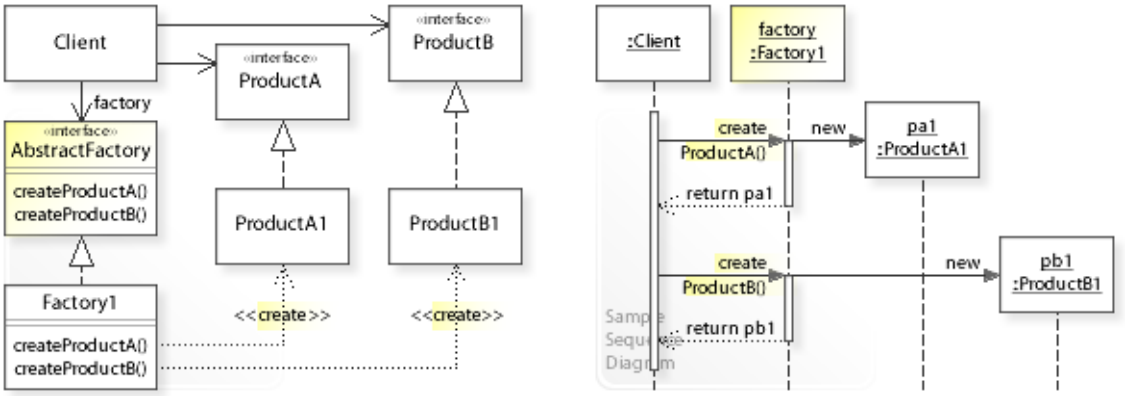
Storing and restoring an object's internal state
without violating encapsulation.

Interpreter / Domain Specific Languages

Interpreting sentences in a simple language.

Part II. Creational Patterns

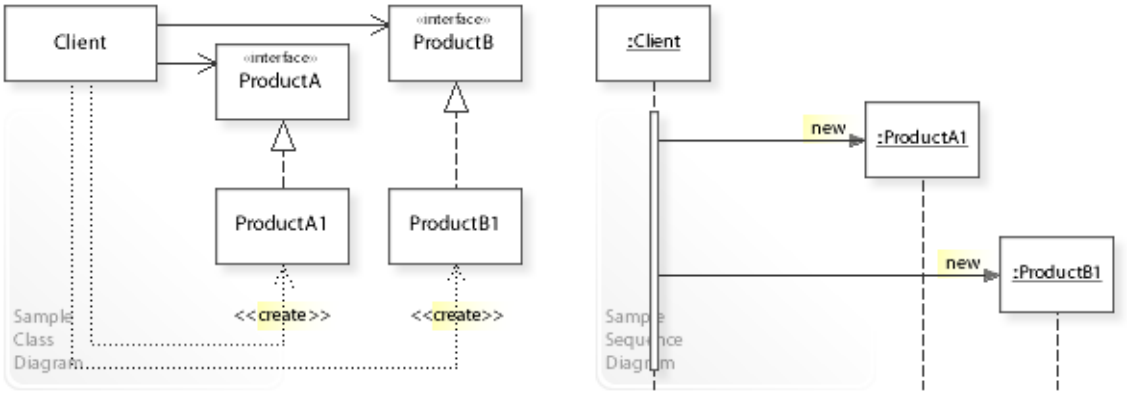
Intent



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 a class be independent of how the objects it requires are created?*
  - *How can different families of related or dependent objects be created?*
- An inflexible way is to create objects directly within the class (`Client`) that requires the objects. This commits the class to particular objects and makes it impossible to change the instantiation later independently from (without changing) the class.
- 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()`, for example) is abstracted by referring to an interface (delegating to a factory object): `factory.createProductA()`.
    There is no longer anything in the client code that instantiates a concrete class.

Problem



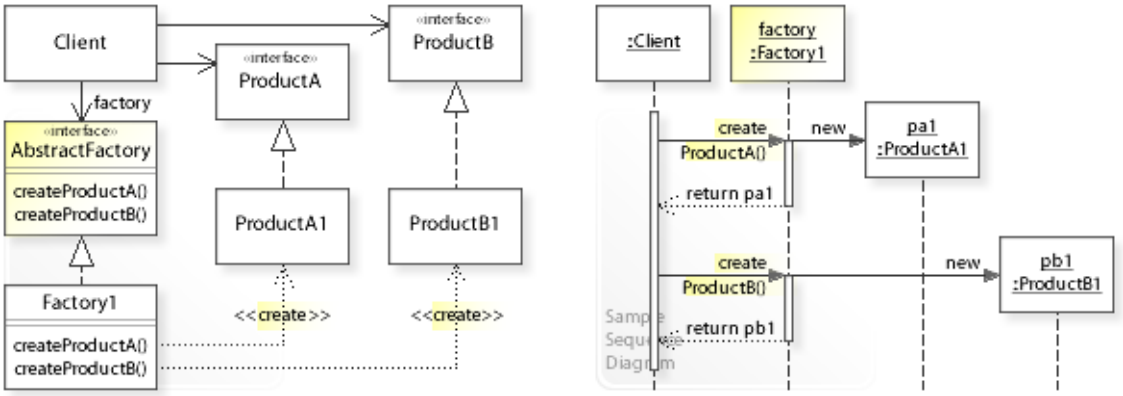
The Abstract Factory design pattern solves problems like:
How can a class be independent of how the objects it requires are created?
How can different families of related or dependent objects be created?
See Applicability section for all problems Abstract Factory can solve. See Solution section for how Abstract Factory solves the problems.

- An inflexible way is to create objects (`new ProductA1(),new ProductB1()`) directly within the class (`Client`) that requires (uses) the objects.
- This commits (couples) 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 if other objects are required, 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 a class is independent of how its objects are created.*
- For example, designing reusable classes that require (depend on) other objects.
  A reusable class should avoid creating the objects it requires directly (and often it doesn't know at compile-time which class to instantiate) so that it can request the objects it requires at run-time (from a factory object).
- For example, supporting different look-and-feels in a Web/GUI application.
  Instantiating look-and-feel-specific classes throughout an application should be avoided so that a look and feel can be selected and exchanged 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



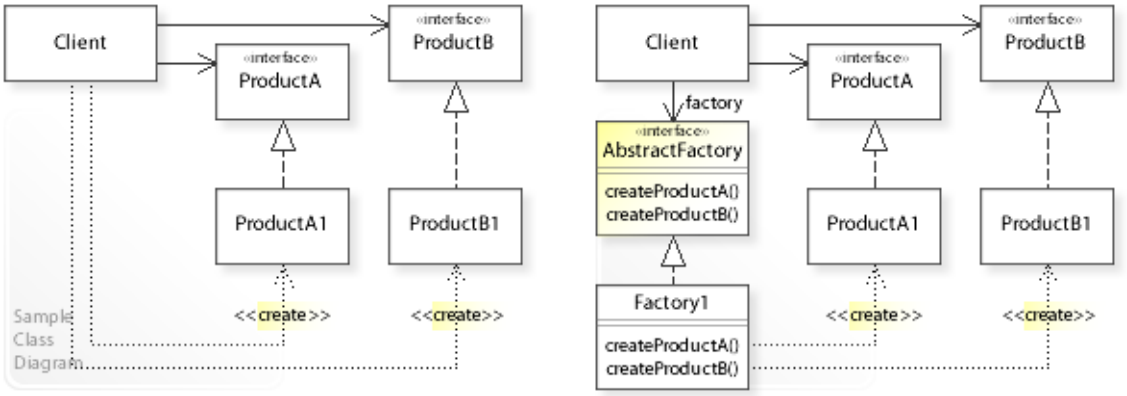
The Abstract Factory design pattern provides a solution:
Encapsulate creating a family of objects in a separate `factory` object.
A class delegates object creation to a `factory` object
instead of instantiating concrete classes directly.
Describing the Abstract Factory design in more detail is the theme 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()`, for example) is abstracted by referring to an interface (delegating to a factory object): `factory.createProductA()`.
  There is no longer anything in the client code that instantiates a concrete 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.
- **A class (`Client`) delegates the responsibility for creating objects to a `factory` object** (`factory.createProductA(),factory.createProductB(),…`).
- This enables *run-time* flexibility (via object composition).
  A class can be configured with a factory object, which it uses to create objects, and even more, the factory object can be exchanged dynamically.

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 referred to as *Factory* or *Factory Object* because all design 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]
- For simple applications that do not need exchangeable 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).
  "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]

Motivation 1



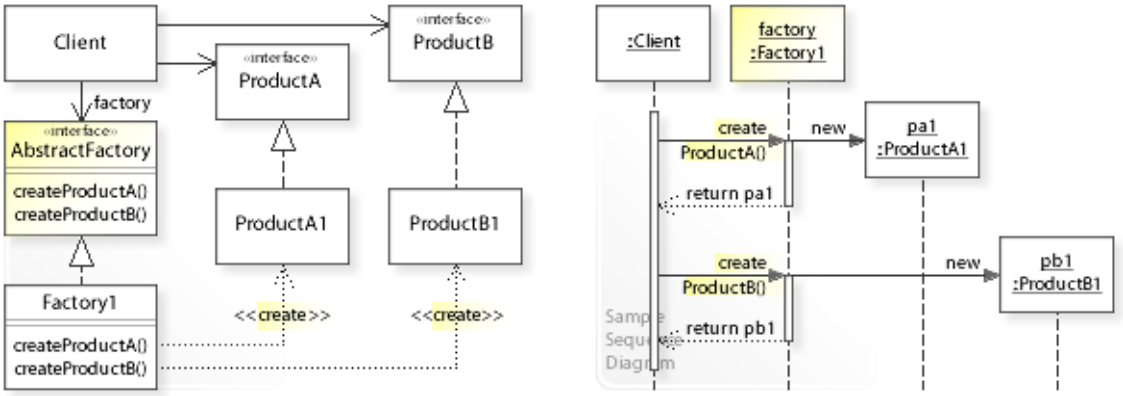
Consider the left design (problem):

- 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 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) clients.
- Centralized object creation.
  - Creating objects is centralized in a single `Factory1` class.

Applicability



Design Problems

- **Creating Objects**
  - How can a class be independent of how the objects it requires are created?
  - How can a class request the objects it requires (from a factory object) instead of creating the objects directly?
  - How can a class delegate object creation to a factory object?
  - How can a class be configured with a factory object?
- **Creating Different Object Families**
  - How can families of related or dependent objects be created?
  - How can be ensured that a family of related or dependent objects is created and used together (consistent object families)?
  - How can an application be configured with a family of objects?
  - How can a family of objects be selected and exchanged at run-time?

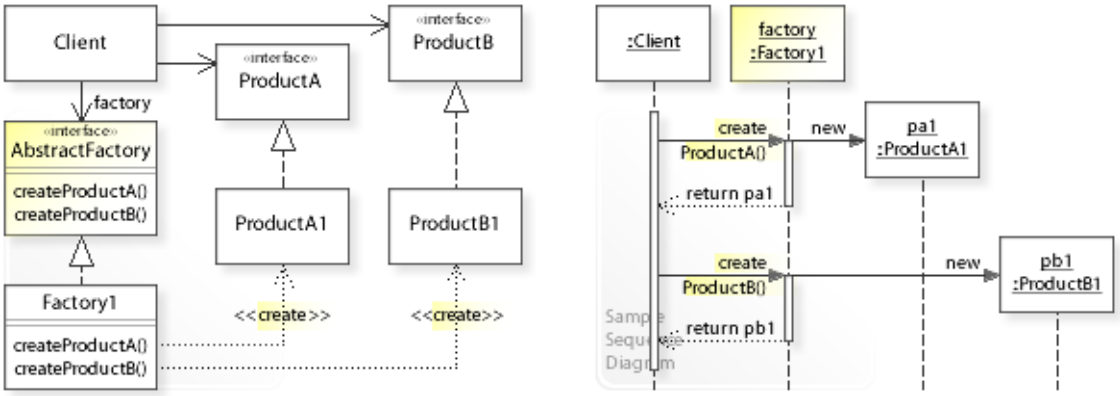
Refactoring Problems

- **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)* [JKerievsky05]

Testing Problems

- **Unit Testing**
  - How can the objects a class requires be replaced with mock objects so that the class can be unit tested in isolation?

Structure, Collaboration



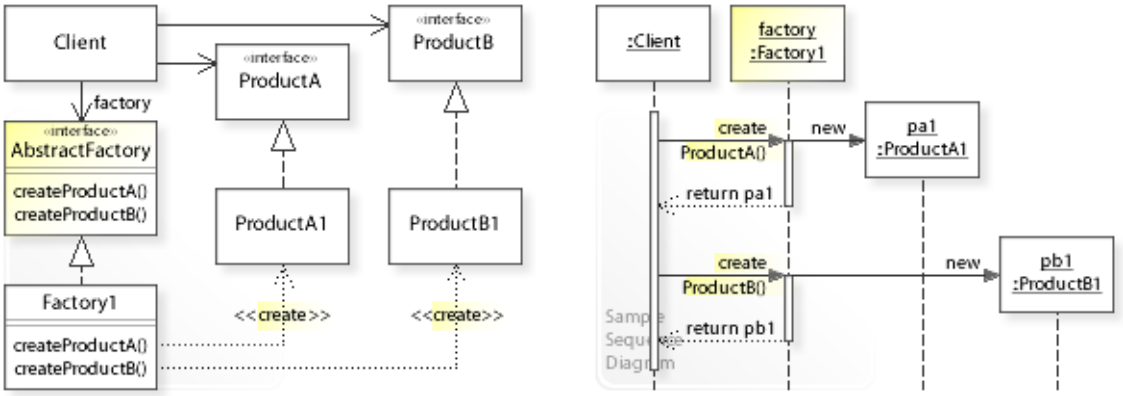
Static Class Structure

- `Client`
  - Requires `ProductA` and `ProductB` objects.
  - 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).
  - Maintains a reference (`factory`) to an `AbstractFactory` object.
- `AbstractFactory`
  - Defines an interface for creating a family of product objects.
- `Factory1`,...
  - Implement the `AbstractFactory` interface by creating and returning the objects.

Dynamic Object Collaboration

- In this sample scenario, a `Client` object delegates creating product 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 the `ProductA1` and `ProductB1` objects as required.
- See also Sample Code / Example 1.

Consequences



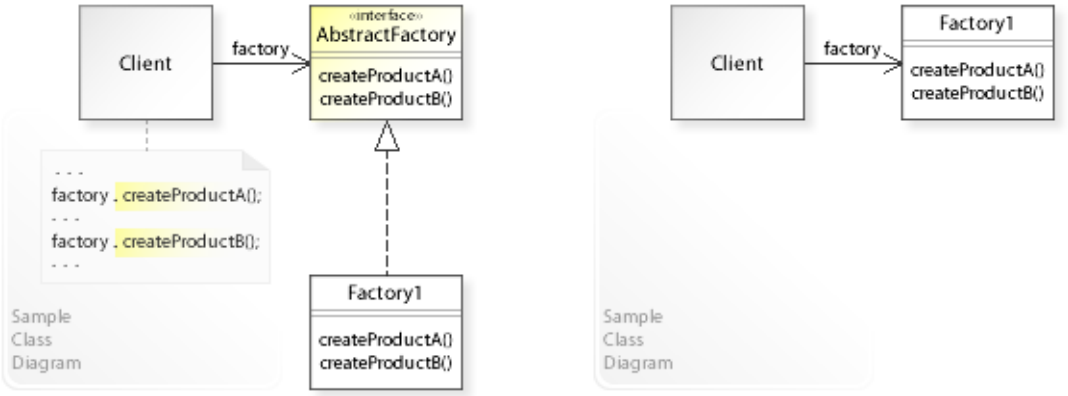
Advantages (+)

- Avoids compile-time implementation dependencies.
  - Instead of instantiating concrete classes directly, clients delegate instantiation to a separate factory object.
- 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 extend a family of objects (to support new kinds of objects).
- Introduces an additional level of indirection.
  - The pattern achieves flexibility by introducing an additional level of indirection (clients delegate instantiation to a separate factory object), which makes clients dependent on a factory object.

Implementation



Implementation Issues

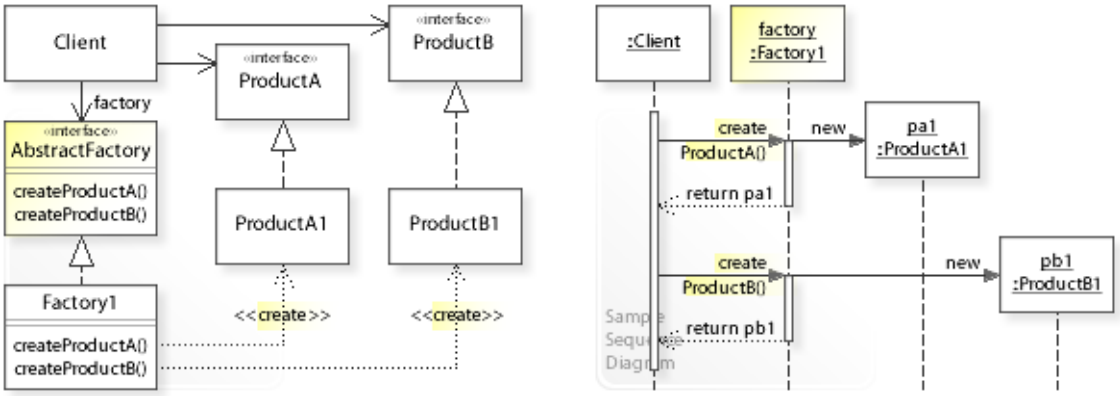
Variant 1: **Abstract Factory**
Creating different families of objects.

- 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 (a family of) objects.

- Interface and implementation are not cleanly separated.
- The concrete `Factory1` class acts as both interface and 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 factory object.
 6          Client client = new Client(new Factory1());
 7          // Calling an operation on the client.
 8          System.out.println(client.operation());
 9      }
10  }
```

```
Client  : Delegating creating objects to a factory object.
Factory1: Creating a ProductA1 object.
Factory1: Creating a ProductB1 object.
Hello World from ProductA1 and ProductB1!
```

```
 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          System.out.println("Client  : Delegating creating objects to a factory object.");
12          productA = factory.createProductA();
13          productB = factory.createProductB();
14          // Doing something appropriate on the created objects.
15          return "Hello World from " + productA.getName() + " and "
16                  + productB.getName() + "!";
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          System.out.println("Factory1: Creating a ProductA1 object.");
 5          return new ProductA1();
 6      }
 7      public ProductB createProductB() {
 8          System.out.println("Factory1: Creating a ProductB1 object.");
 9          return new ProductB1();
10      }
11  }
```

```
 1  package com.sample.abstractfactory.basic1;
 2  public class Factory2 implements AbstractFactory {
 3      public ProductA createProductA() {
 4          System.out.println("Factory2: Creating a ProductA2 object.");
 5          return new ProductA2();
 6      }
 7      public ProductB createProductB() {
 8          System.out.println("Factory2: Creating a ProductB2 object.");
 9          return new ProductB2();
10      }
11  }
```

```
******************************************************
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 ProductAMock and ProductBMock!",
10                  client.operation());
11      }
12      // More tests ...
13  }
```

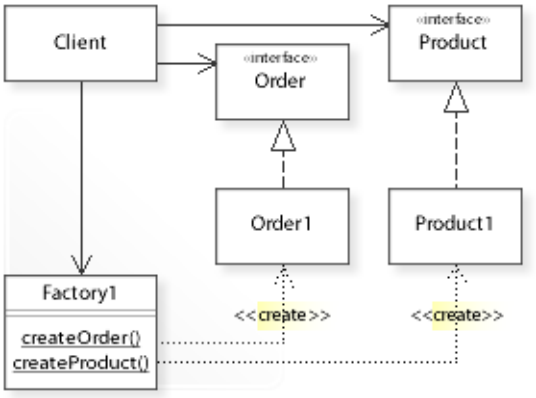
```
 1  package com.sample.abstractfactory.basic1;
 2  public class FactoryMock implements AbstractFactory {
 3      public ProductA createProductA() {
```

```
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 simple applications that do not need to create families of objects but want to separate and centralize 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.
******************************************************
```

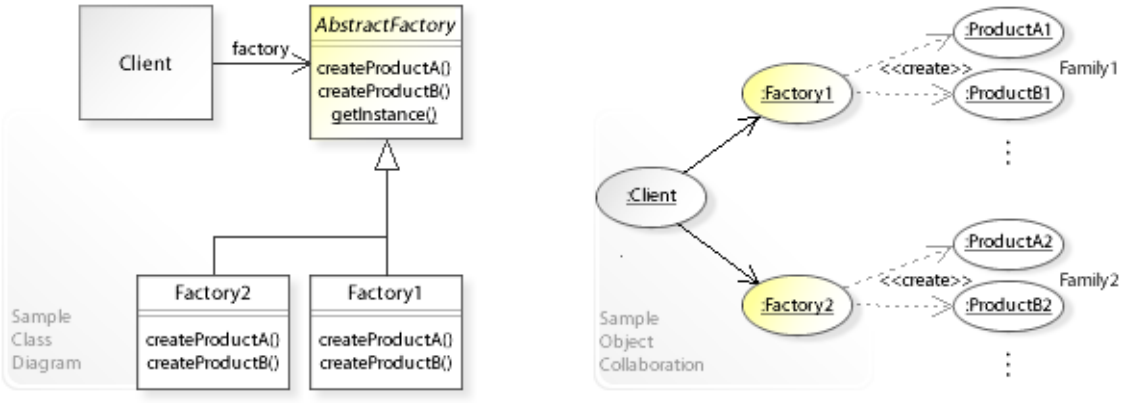
```
 1  package com.sample.abstractfactory.basic2;
 2  public interface Order {
 3      // ...
 4  }
```

```
 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



Creating families of objects. Ensuring that a family of related or dependent objects is created and used together (consistent object families).

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

```
33          public ProductA createProductA() {
34              System.out.println("  creating a ProductA2 object ...");
35              return new ProductA2();
36          }
37          public ProductB createProductB() {
38              System.out.println("  creating a ProductB2 object ...");
39              return new ProductB2();
40          }
41      }
42  }
```

```
*****************************************************
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  }
```

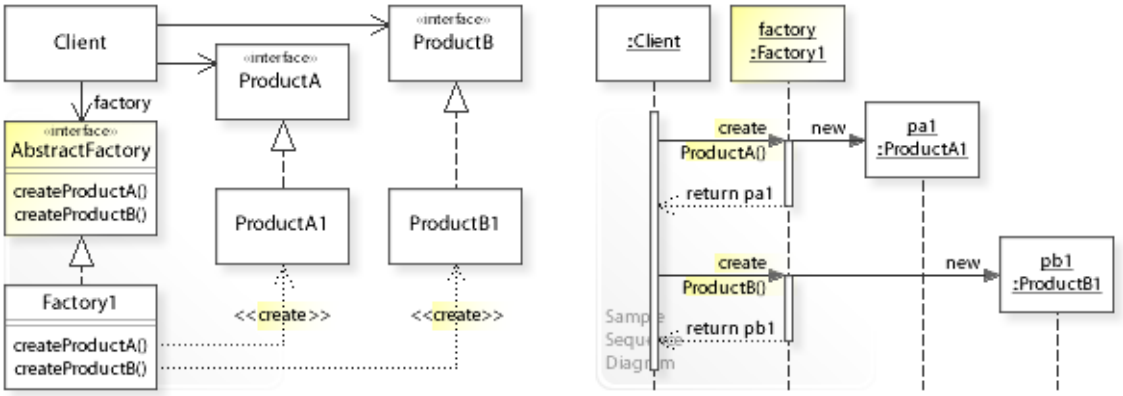
```
1  package com.sample.abstractfactory.basic3;
2  public class ProductA2 implements ProductA {
3      // ...
4  }
```

```
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  }
```

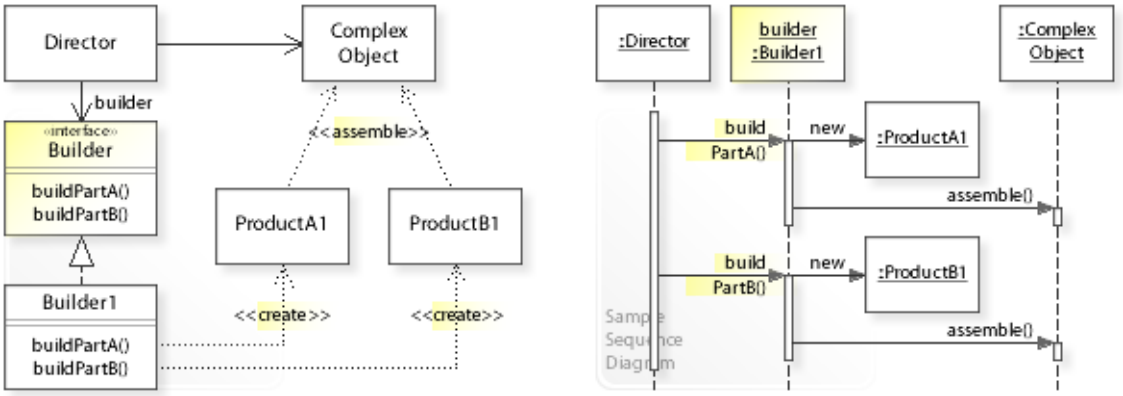
Related Patterns



Key Relationships

- **Abstract Factory - Dependency Injection**
  - Abstract Factory
    A class delegates creating the objects it requires to a factory object, which makes the class dependent on the factory.
  - Dependency Injection
    A class accepts the objects it requires from an injector object without having to know the injector, which greatly simplifies the class.
- **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 performing an algorithm to a strategy object.
  - Abstract Factory
    A class delegates creating an object to a factory object.
- **Strategy - Abstract Factory - Dependency Injection**
  - Strategy
    A class can be configured with a strategy object.
  - Abstract Factory
    A class can be configured with a factory object.
  - Dependency Injection
    Actually performs the configuration by creating and injecting the objects a class requires.

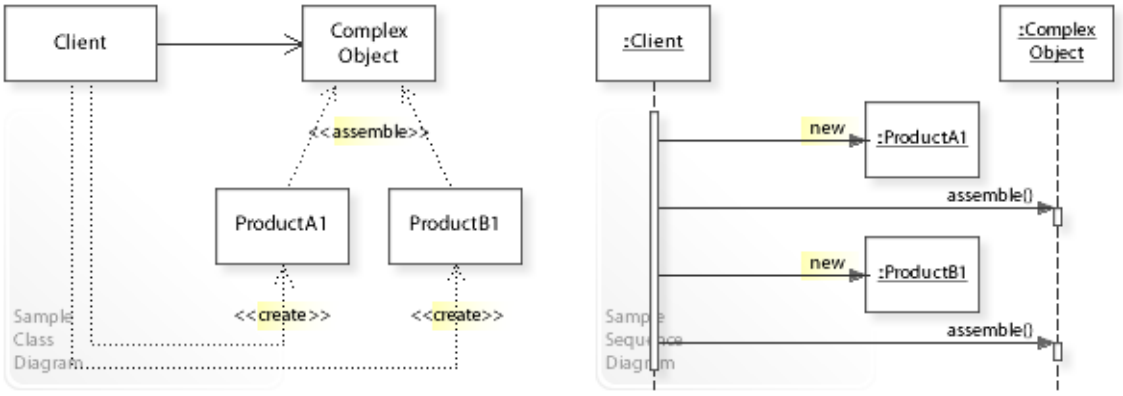
Intent



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 and assembling the parts of a complex object directly within a class makes it impossible to 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 the creation of a complex object in a separate `Builder` object.
  - A class can create different representations of a complex object by delegating to different `Builder` objects.

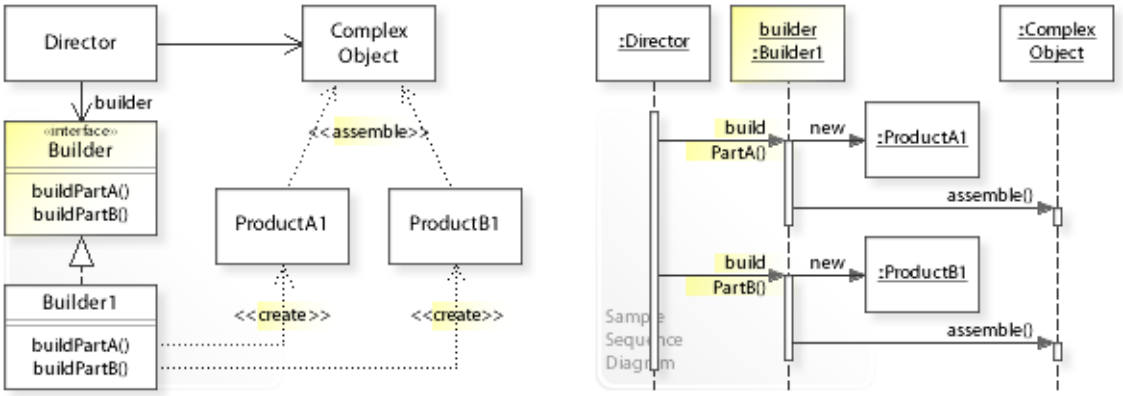
Problem



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. See Solution section for how Builder solves the problems.

- An inflexible way is to create and assemble the parts of a complex object (`new ProductA1(), add to complex object, new ProductB1(), add to complex object, …`) directly within a class (`Client`).
- This commits (couples) 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.
  A class should avoid instantiating concrete product classes directly so that it can create different product structures (representations) of the BOM.

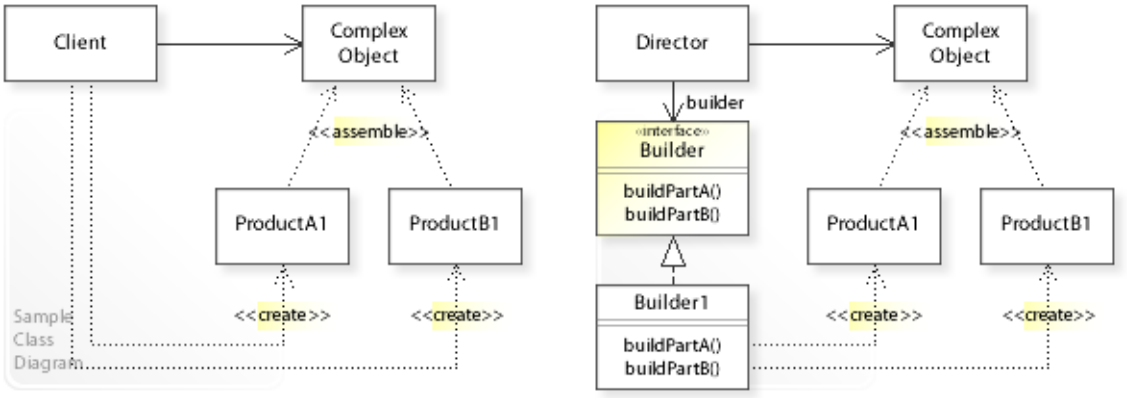
Solution



The Builder design pattern provides a solution:
Encapsulate creating and assembling the parts of a complex object in a separate `Builder` object.
A class delegates object creation to a `Builder` object
instead of instantiating concrete classes directly.
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 separate creating and assembling the parts of a complex object from other (construction) code (`Director`).
- **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 object is created step by step to have finer control over the creation process.
- 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 by defining new (sub)classes.
- **A class (`Director`) delegates the responsibility for creating and assembling the parts of a complex object to a `Builder` object** (`builder.buildPartA(),builder.buildPartB(),…`).
- This enables *run-time* flexibility (via object composition).
  A class (the same construction process) can use different `Builder` objects to create different representations of a complex object.

Motivation 1



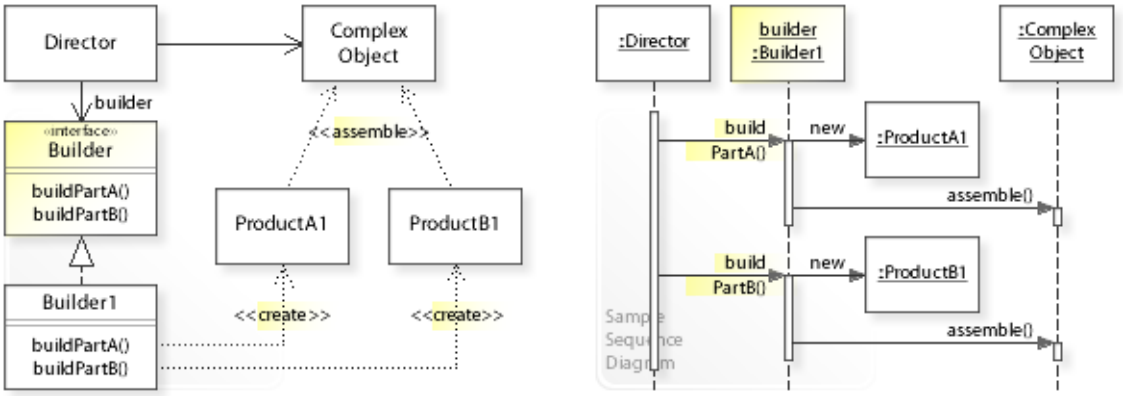
Consider the left design (problem):

- Hard-wired object creation.
  - Creating a representation of a complex object is implemented (hard-wired) directly within a class (`Client`).
  - This makes it hard to create a different representation independently from (without having to change) the class.
- Complicated classes.
  - Classes that include creating a complex object are hard to implement, change, test, 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 create a different representation independently from (without having to change) clients (`Director`).
- Simplified classes.
  - Classes that delegate creating a complex object are easier to implement, change, test, and reuse.

Applicability



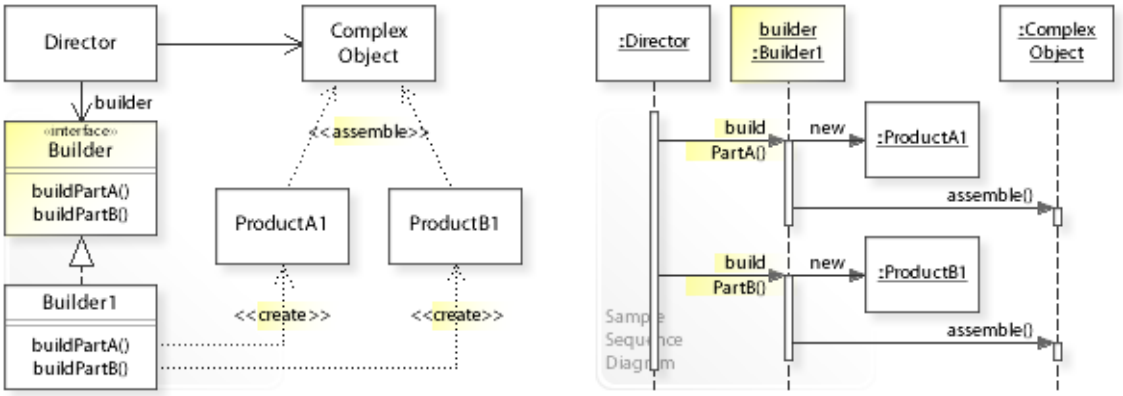
Design Problems

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

Refactoring Problems

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

Structure, Collaboration



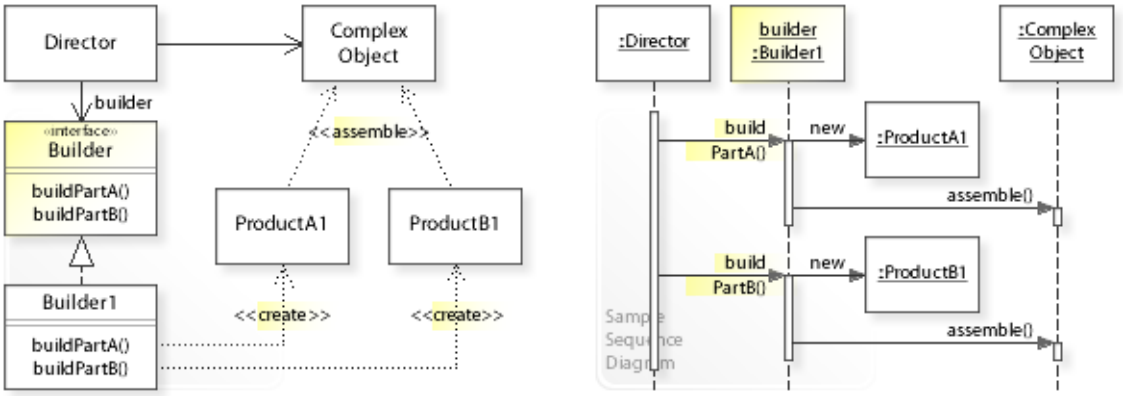
Static Class Structure

- `Director`
  - Refers to the `Builder` interface to create parts of a complex object.
  - Is independent of how the complex object is created (which concrete classes are instantiated, i.e., which representation is created).
  - Maintains a reference (`builder`) to a `Builder` object.
- `Builder`
  - Defines an interface for creating parts of a complex object.
- `Builder1`,...
  - Implement the `Builder` interface by creating and assembling the parts of a complex object.

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 assembled `ComplexObject` from the `Builder1` and use it as required.
- See also Sample Code / Example 1.

Consequences



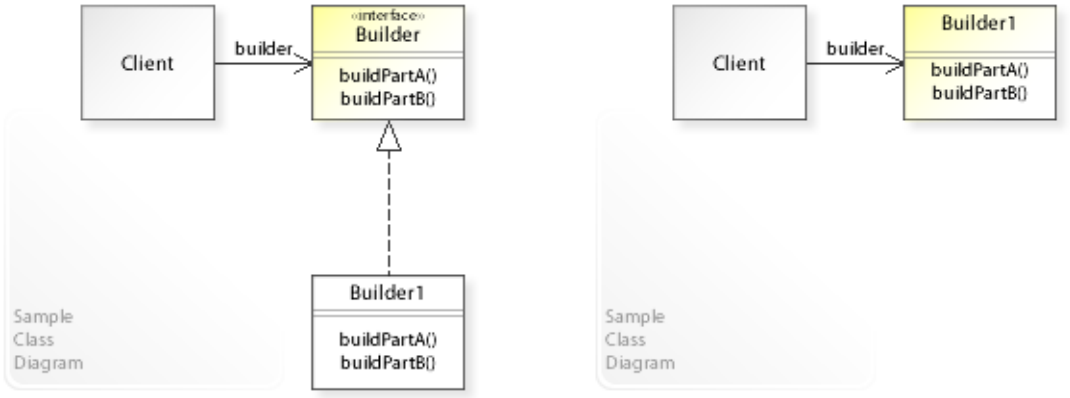
Advantages (+)

- Avoids compile-time implementation dependencies.
  - Instead of instantiating concrete classes directly, clients delegate instantiation to a separate builder object.
- Simplifies clients.
  - Because clients delegate creating a complex object to a builder object, they are easier to implement, change, test, and reuse.

Disadvantages (–)

- Introduces an additional level of indirection.
  - The pattern achieves flexibility by introducing an additional level of indirection (clients delegate object creation to a separate builder object), which makes clients dependent on a builder object.

Implementation



Implementation Issues

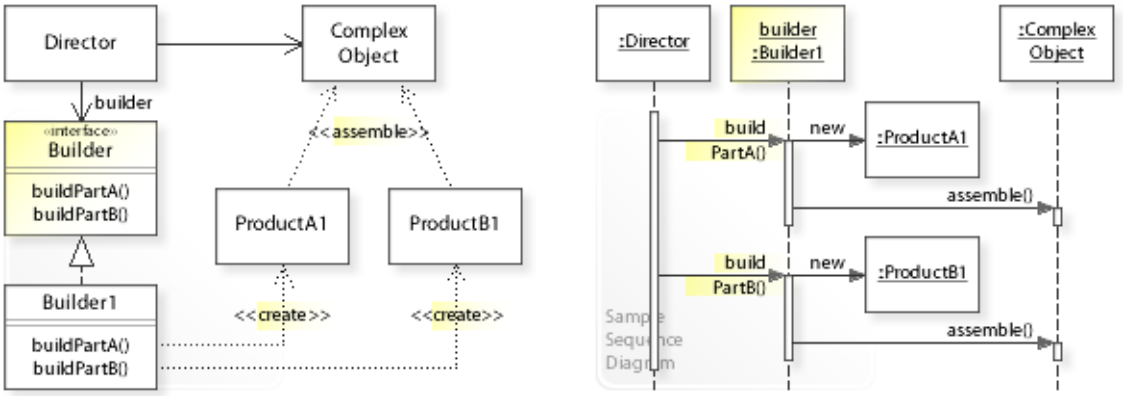
Variant 1: **Abstract Builder**
Creating different representations.

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

Variant 2: **Concrete Builder**
Creating a complex object.

- Interface and implementation are not cleanly separated.
- The concrete builder class acts as both interface and implementation.
- This is a simple way to implement the Builder pattern for clients 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          // Calling construct on the director.
 8          System.out.println(director.construct());
 9      }
10  }
```

```
Director: Delegating constructing a complex object to a builder object.
Builder1: Creating and assembling ProductA1.
Builder1: Creating and assembling ProductB1.
Hello World from Complex Objetct 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          System.out.println("Director: Delegating constructing "
11                  + "a complex object to a builder object.");
12          builder.buildPartA();
13          builder.buildPartB();
14          co = builder.getResult();
15          return "Hello World from " + co.getParts() + " objects!";
16      }
17  }
```

```
 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          System.out.println("Builder1: Creating and assembling ProductA1.");
 7          co.add(new ProductA1());
 8      }
 9      public void buildPartB() {
10          System.out.println("Builder1: Creating and assembling ProductB1.");
11          co.add(new ProductB1());
```

```
12      }
13      public ComplexObject getResult() {
14          return co;
15      }
16  }
```

```
 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 Objetct 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  }
```

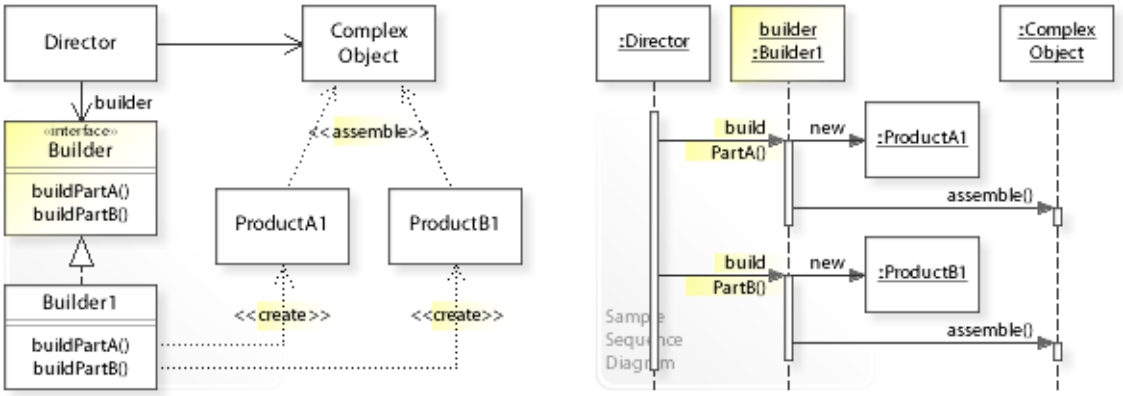
```
 1  package com.sample.builder.basic;
 2  public interface ProductA extends Product {
 3      // ...
 4  }
```

```
 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  }
```

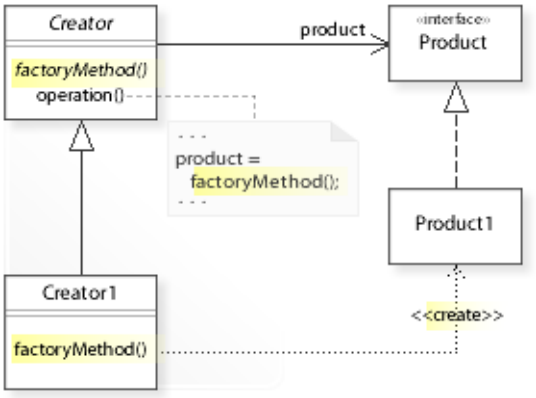
Related Patterns



Key Relationships

- **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).

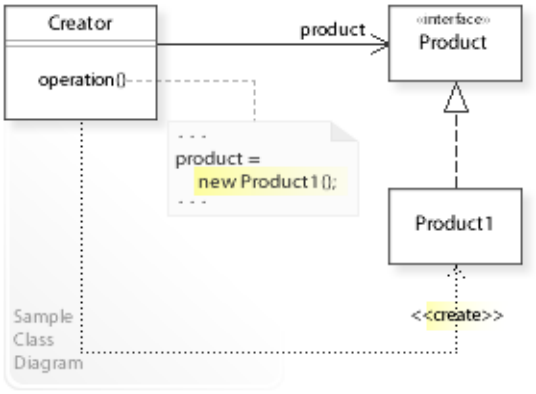
Intent



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?*
- An inflexible way is to create an object directly within the class (`Creator`) that requires (uses) the object. This commits the class to a particular object and makes it impossible to change the instantiation independently from (without having to change) the class.
- 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.

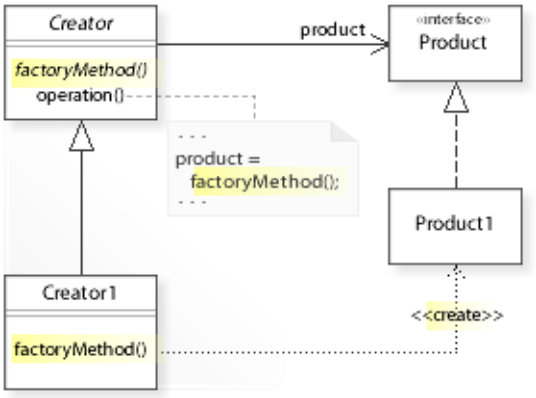
Problem



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. See Solution section for how Factory Method solves the problems.

- An inflexible way is to create an object (`new Product1()`) directly within the class (`Creator|operation()`) that requires (uses) the object.
- This commits (couples) the class to a particular object and makes it impossible to change the instantiation (which class to instantiate) 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 the way the object is created.*
- For example, designing reusable classes that require (depend on) other objects.
  A reusable class should avoid creating the objects it requires directly (and often it doesn't know which class to instantiate) so that users of the class can write subclasses to specify the instantiation they need.

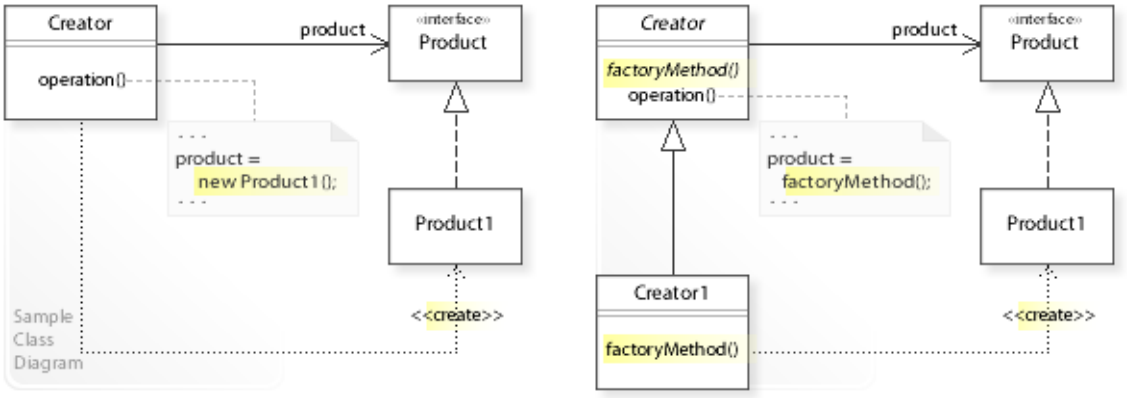
Solution



The Factory Method design pattern provides a solution:
Define a separate operation (`factory method`) for creating an object.
Create an object by calling a `factory method`.
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.
- **Define a separate `factory method`:**
  - The pattern calls a (separate) operation that is (exclusively) responsible for "manufacturing" an object a *factory method*. [GoF, p108]
- **Create an objects by calling a `factory method`**
  (`Product product = factoryMethod()`).
- This enables *compile-time* flexibility (via subclassing).
  Subclasses can be written to redefine the way an object is created.
- "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 the Factory Method pattern can be implemented differently (abstract, concrete, or static factory method). See Implementation.

Motivation 1



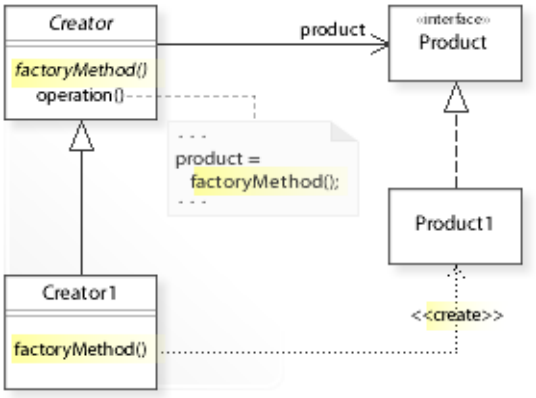
Consider the left design (problem):

- Hard-wired object creation.
  - Object creation is defined (hard-wired) directly within a class (`Creator`).
  - This makes it hard to change the instantiation independently from (without having to change) the class.
- Unknown object creation.
  - A reusable class often doesn't know which class to instantiate.
  - Users of the class should specify which class to instantiate to suit their needs (by writing subclasses).

Consider the right design (solution):

- Encapsulated object creation.
  - Object creation is defined (encapsulated) in a separate operation (factory method).
  - This makes it easy to change the instantiation independently from the class (by adding new subclasses).
- Deferred object creation.
  - Creator defers instantiation to subclasses by calling an abstract factory method.
  - "It gets around the dilemma of having to instantiate unforeseeable classes." [GoF, p110]

Applicability



Design Problems

- **Creating Objects**
  - 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 to Constructors**
  - How can a flexible alternative be provided to direct constructor calls?

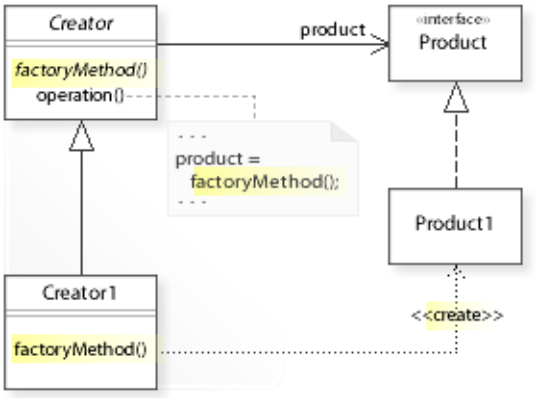
Refactoring Problems

- **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).

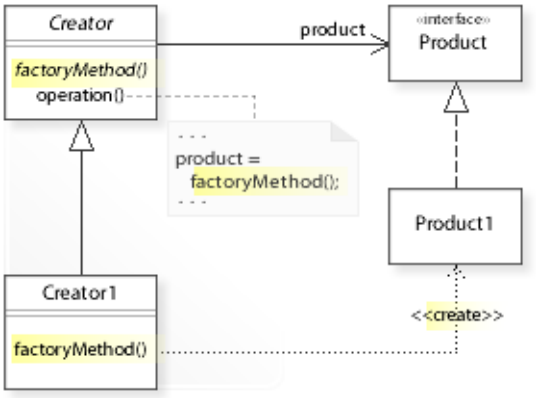
Structure, Collaboration



Static Class Structure

- *`Creator`*
  - Requires a `Product` object.
  - Defines an abstract factory method (`factoryMethod()`) for creating a `Product` object.
  - Is independent of how the `Product` object is created (which concrete class is instantiated).
  - Calls the factory method (`product = factoryMethod()`), but clients from outside the `Creator` may also call the factory method.
- `Creator1,...`
  - Subclasses 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 defer instantiation to subclasses (by calling a factory method) 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 an object is 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]

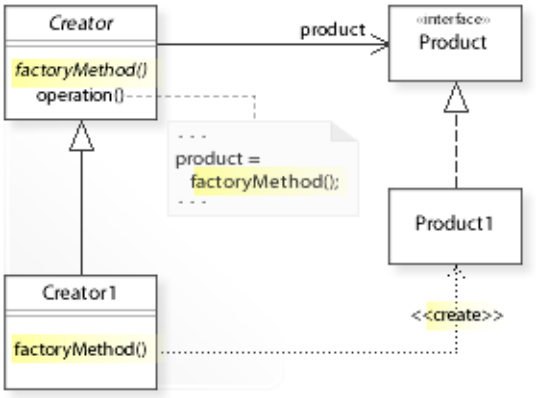
Variant 2: **Concrete Factory Method**

- The factory method is concrete and provides a (default) implementation.
- The concrete factory method acts as both interface and implementation (it abstracts and implements object creation).
- "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]

Background Information

- **Static Factory Method**
  - Static factory methods are widely used when flexibility is needed but overriding via subclassing not.
  - 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]
  - "Consider static factory methods instead of constructors." [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          Creator creator = new Creator1();
 5
 6          System.out.println(creator.operation());
 7      }
 8  }
```

```
Hello World from Creator1!
Product1 created.
```

```
 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() + " created.";
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.
******************************************************
```

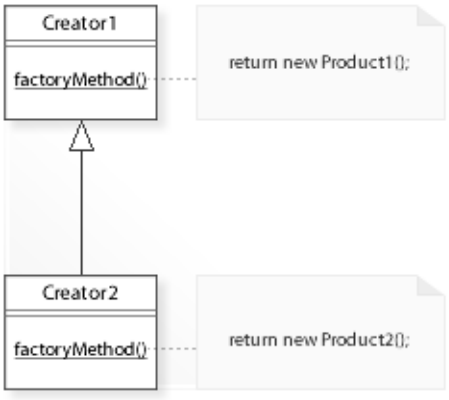
```
 1  package com.sample.factorymethod.basic;
 2  public interface Product {
 3      String getName();
 4  }
```

```
 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          return "Product2";
```

```
5         }
6   }
```

Sample Code 2



Basic Java code for implementing static factory methods.
Static factory methods can't be overridden by subclasses (see Implementation).

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

```
Product1 created.
Product2 created.
```

```
 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  }
```

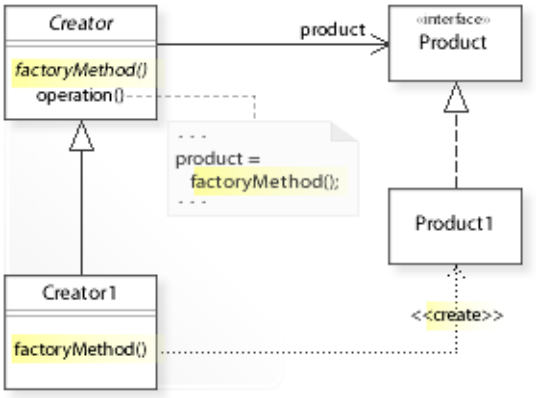
```
*****************************************************
Product inheritance hierarchy.
*****************************************************
```

```
 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  }
```

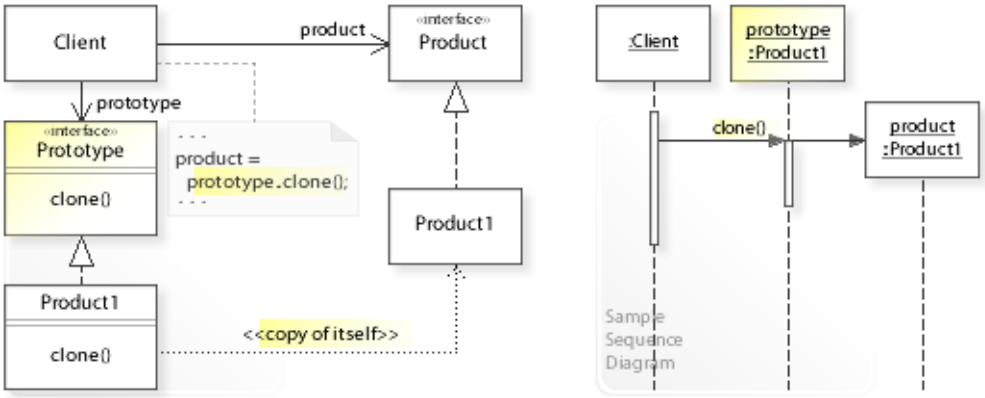
Related Patterns



Key Relationships

- **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 which objects to create dynamically at run-time.
    Prototype will work wherever Factory Method will and with more flexibility.
    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.

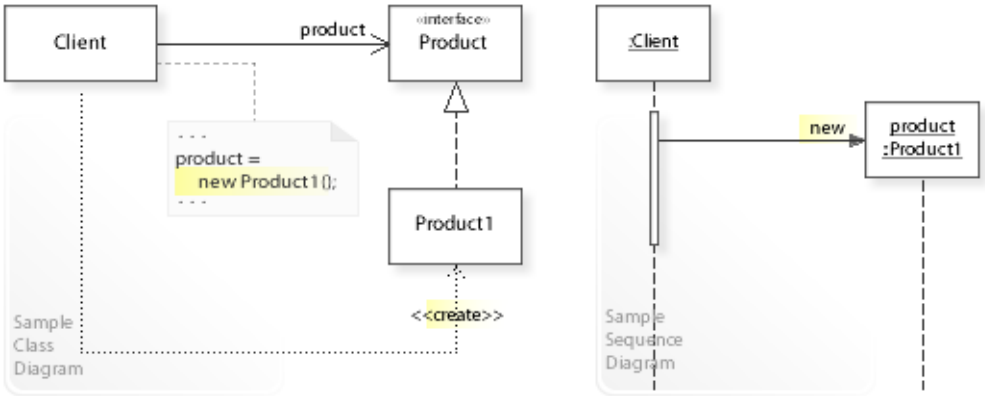
Intent



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 objects be created*
    so that which objects to create can be specified at run-time?
  - *How can dynamically loaded classes be instantiated?*
- 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.

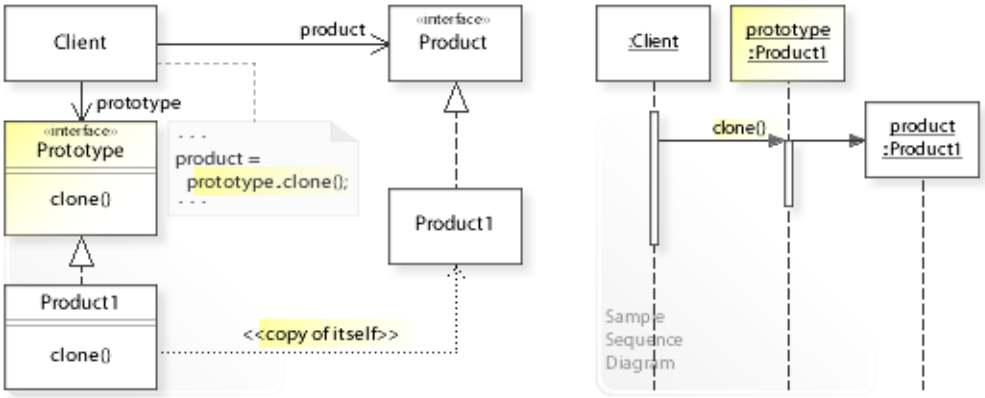
Problem



The Prototype design pattern solves problems like:
How can objects be created
so that which objects to create can be specified at run-time?
How can dynamically loaded classes be instantiated?
See Applicability section for all problems Prototype can solve. See Solution section for how Prototype solves the problems.

- An inflexible way is to create an object (`new Product1()`) directly within the class (`Client`) that requires (uses) the object.
- This commits (couples) the class to a particular object at compile-time and makes it impossible to specify which object to create at run-time.
- *That's the kind of approach to avoid if we want to specify which objects to create at run-time.*
- For example, designing reusable classes that require (depend on) other objects.
  A reusable class should avoid creating the objects it requires directly (and often it doesn't know at compile-time which class to instantiate) so that which objects to create can be specified at run-time.

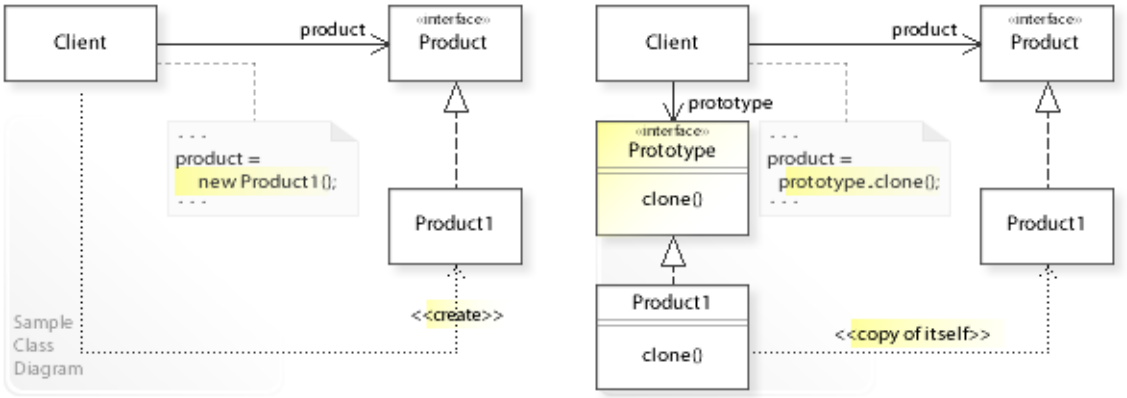
Solution



The Prototype design pattern provides a solution:
Define a `Prototype` object that returns a copy of itself.
Create new objects by copying a `Prototype` object.
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 a `Prototype` object**
  (`Product product = prototype.clone()`).
- This enables *run-time* flexibility (via object composition).
  A class can be configured with different `Prototype` objects, which are copied to create new objects, and even more, `Prototype` objects can be added and removed dynamically at run-time.

Motivation 1



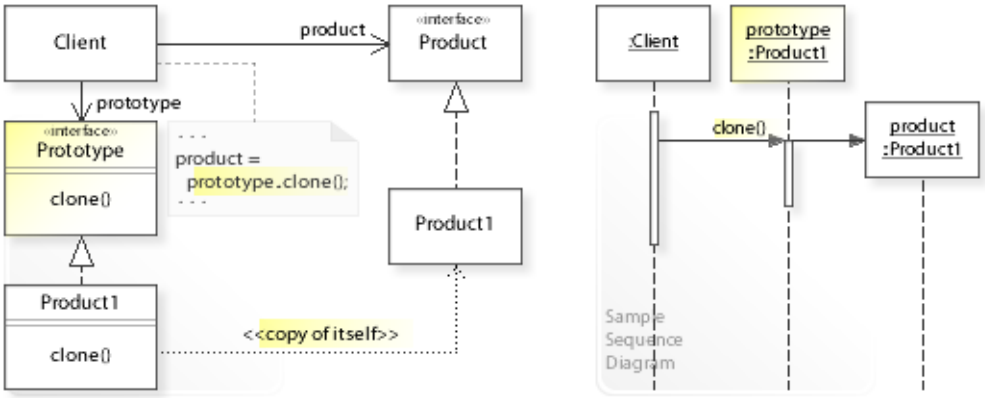
Consider the left design (problem):

- Which object to create is specified at compile-time.
  - Which object to create is specified at compile-time by a direct constructor call (`new Product1()`).

Consider the right design (solution):

- Which object to create is specified at run-time.
  - Which object to create is specified at run-time by copying a prototype object (`prototype.clone()`).
  - Prototypes can be added and removed dynamically at run-time.

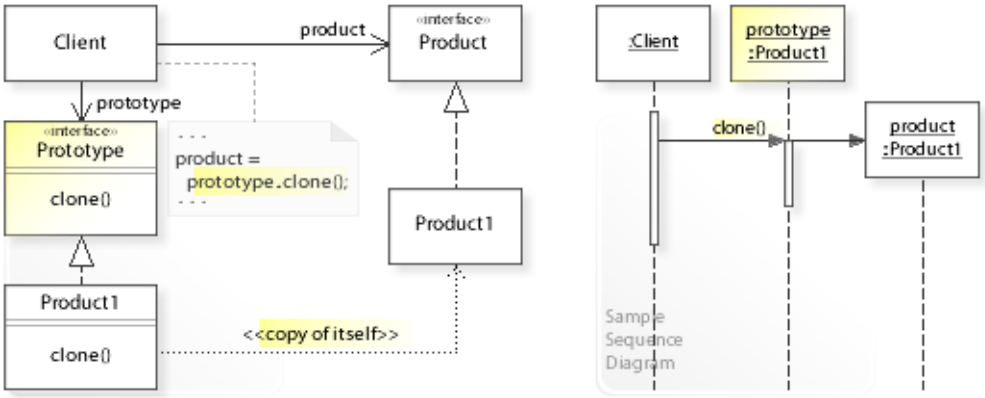
Applicability



Design Problems

- **Creating Objects**
  - How can objects be created
    so that which objects to create can be specified at run-time?
- **Instantiating Dynamically Loaded Classes**
  - How can dynamically loaded classes be instantiated?

Structure, Collaboration



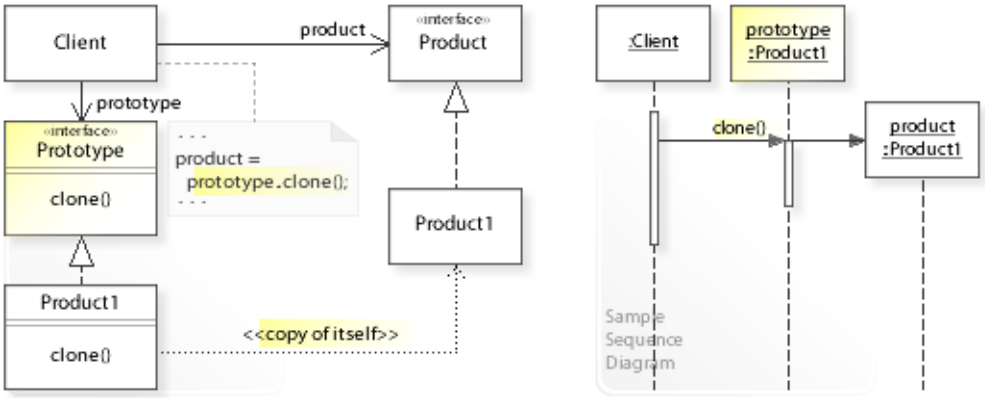
Static Class Structure

- `Client`
  - Requires a `Product` object.
  - Refers to the `Prototype` interface to clone an object (`prototype.clone()`).
- `Prototype`
  - Defines an interface for cloning an object (`clone()`).
- `Product1`,…
  - Implement the `Prototype` interface.
  - Any object that implements the `Prototype` interface can act as prototype for creating a copy of itself.

Dynamic Object Collaboration

- In this sample scenario, a `Client` object calls `clone()` on a `prototype:Product1` object, which creates and returns a copy of itself (`product:Product1`).
- 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 specify which class to instantiate.
  - Prototype will work wherever Factory Method will and with more flexibility.

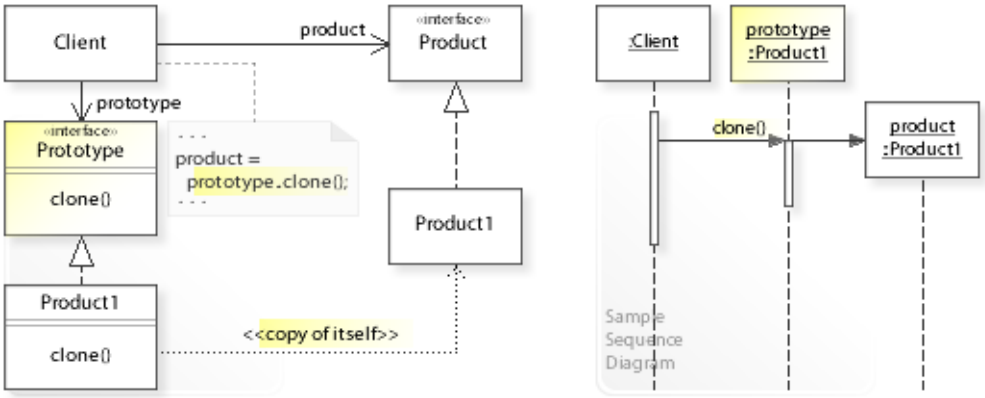
Disadvantages (–)

- 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.

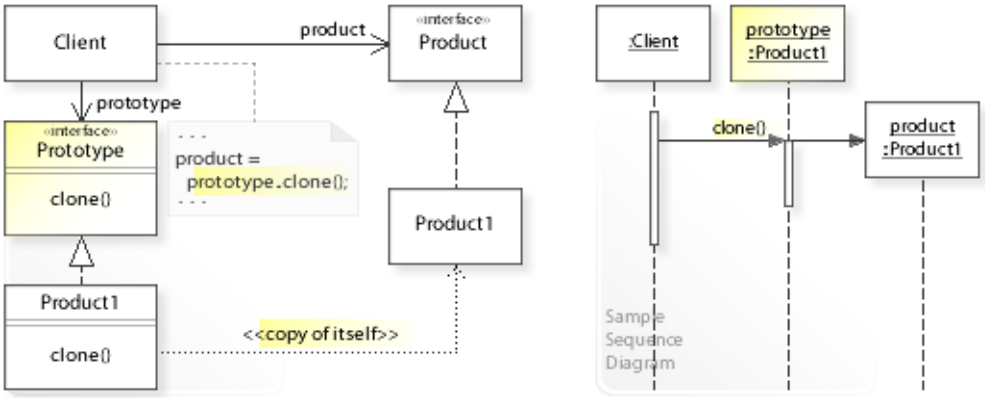
Implementation



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          // Calling an operation on the client.
 8          System.out.println(client.operation());
 9      }
10  }
```

```
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 "Client: Cloning " + prototype.getClass().getSimpleName() + ".\n"
12                  + product.getName() + " object copied.";
13      }
14  }
```

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

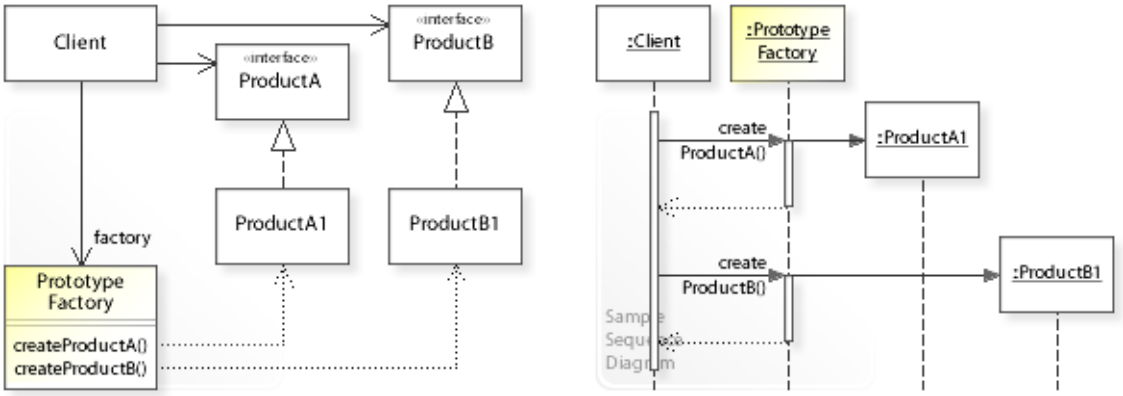
```
 1  package com.sample.prototype.basic;
 2  public interface Product {
 3      String getName();
 4  }
```

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

```
 1  package com.sample.prototype.basic;
 2  // Product1 implements both the Product and Prototype interface.
 3  public class Product1 implements Product, Prototype {
 4      private String name;
 5
 6      public Product1(String name) {
 7          this.name = name;
 8      }
```

```
 9      // Copy constructor needed by clone().
10      public Product1(Product1 p) {
11          this.name = p.getName();
12      }
13      @Override
14      public Product clone() {
15          return new Product1(this);
16      }
17      public String getName() {
18          return name;
19      }
20  }
```

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          System.out.println(client.operation());
 9      }
10  }
```

```
Client: Delegating object creation to a prototype factory.
PrototypeFactory: Cloning a ProductA object.
PrototypeFactory: Cloning a ProductB object.
Hello World from ProductA1 and ProductB1!
```

```
 1  package com.sample.prototype.basicAF;
 2  public class Client {
 3      private ProductA productA;
 4      private ProductB productB;
 5      private PrototypeFactory ptFactory;
 6
 7      public Client(PrototypeFactory ptFactory) {
 8          this.ptFactory = ptFactory;
 9      }
10      public String operation() {
11          System.out.println("Client: Delegating object creation to a prototype factory.");
12          productA = ptFactory.createProductA();
13          productB = ptFactory.createProductB();
14          return "Hello World from " + productA.getName() + " and "
15                  + productB.getName() + "!";
16      }
17  }
```

```
 1  package com.sample.prototype.basicAF;
 2  public class PrototypeFactory {
 3      private ProductA productA;
 4      private ProductB productB;
 5
 6      public PrototypeFactory(ProductA pa, ProductB pb) {
 7          this.productA = pa;
 8          this.productB = pb;
 9      }
10      public ProductA createProductA() {
11          System.out.println("PrototypeFactory: Cloning a ProductA object.");
12          return productA.clone();
13      }
14      public ProductB createProductB() {
15          System.out.println("PrototypeFactory: Cloning a ProductB object.");
16          return productB.clone();
17      }
18  }
```

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

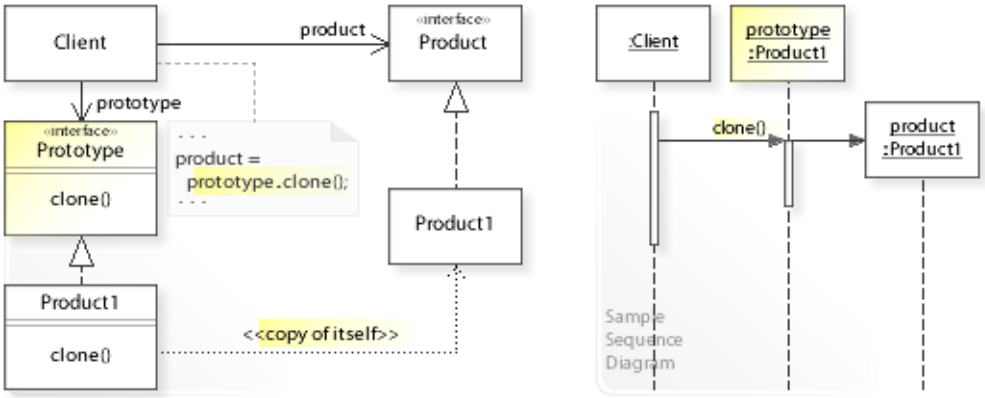
```
 1  package com.sample.prototype.basicAF;
 2  public interface ProductA {
 3      String getName();
 4      ProductA clone();
 5  }
```

```
 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  }
```

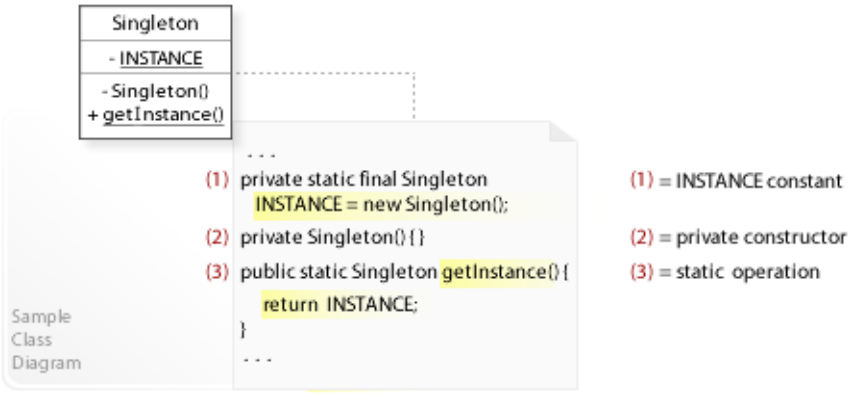
Related Patterns



Key Relationships

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

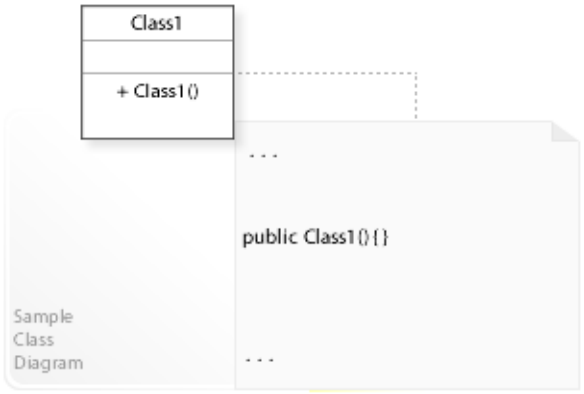
Intent



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, 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 if necessary.

Problem



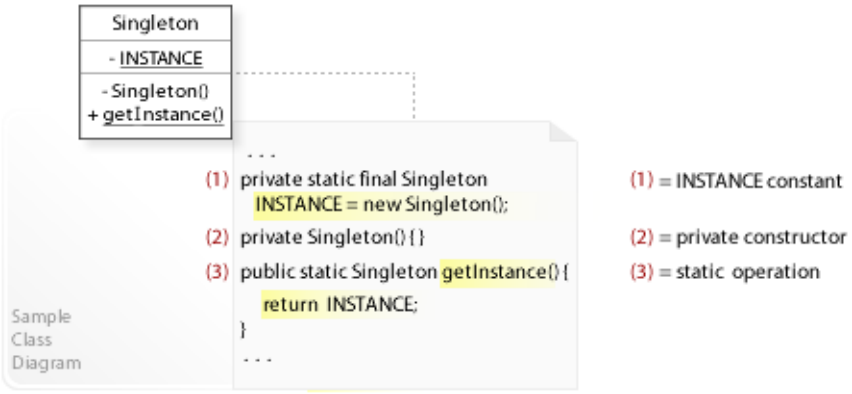
The Singleton design pattern solves problems like:
How can be ensured that a class has only one instance?
How can the sole instance of a class be accessed globally?
See Applicability section for all problems Singleton can solve. See Solution section for how Singleton solves the problems.

- The standard way is to call the public constructor of a class (`new Class1()`) each time a new object is needed.
- *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, 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 if necessary.
- A Registry is
  "A well-known object that other objects can use to find common objects and services."
  Registry (480) [MFowler03]

Solution



The Singleton design pattern provides a solution:
Hide the constructor of a class, and define a static operation (`getInstance()`) **that returns the sole instance of the class.**
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 neither be instantiated (from outside the class) nor subclassed (because subclasses need the constructor of their parent class).
  "Enforce the singleton property with a private constructor or enum type" [JBloch08, Item 3]
- **Define a static operation** (`getInstance()`) **that returns the sole instance of the class.**
  `getInstance()` is declared *public* and *static*.
  A public static operation of a class is easy to access from anywhere within an application by using the class name and operation name (= global point of access): `Singleton.getInstance()`.

Motivation 1

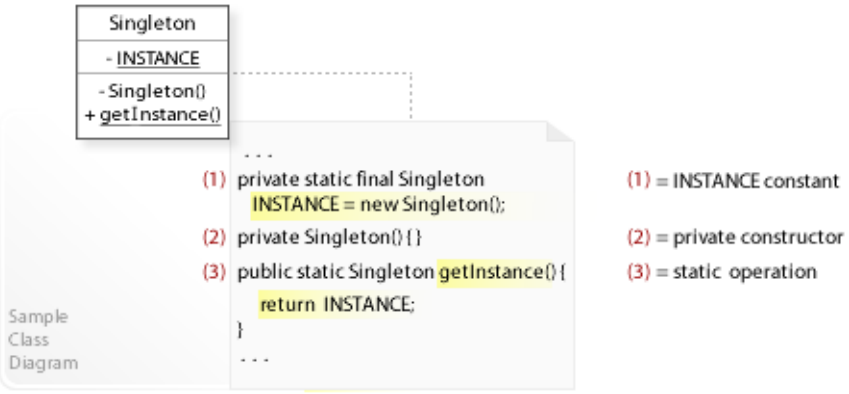
Consider the left design (problem):

- Multiple instances possible.
  - The class provides a public constructor (`public Class() {}`).
  - Clients can call the public constructor of a class each time a new object is needed.

Consider the right design (solution):

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

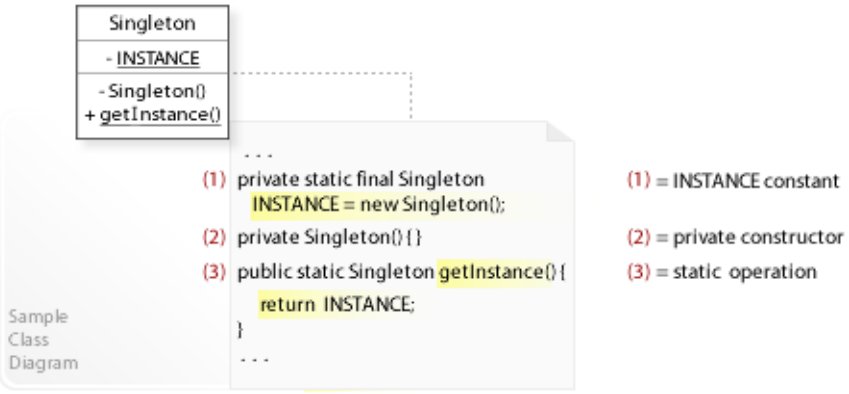
Applicability



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?
- **Controlling Instantiation**
  - How can a class control its instantiation?
  - How can the number of instances of a class be restricted?
  - How can creating large numbers of unnecessary objects be avoided?

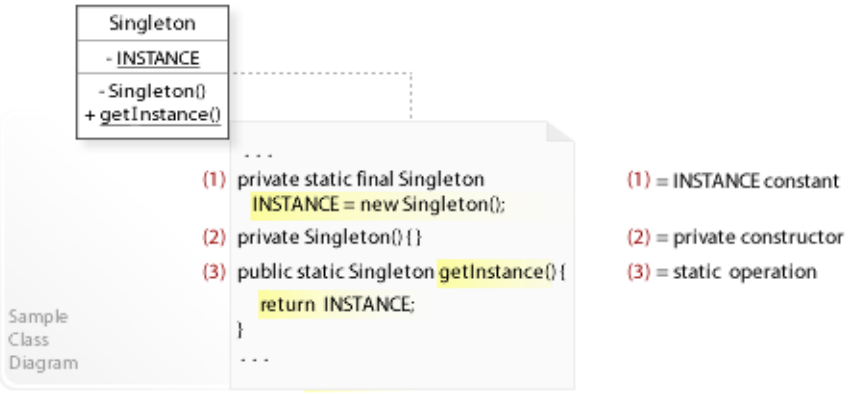
Structure, Collaboration



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 static operation (`getInstance()`) for returning the sole instance of the class.

Consequences



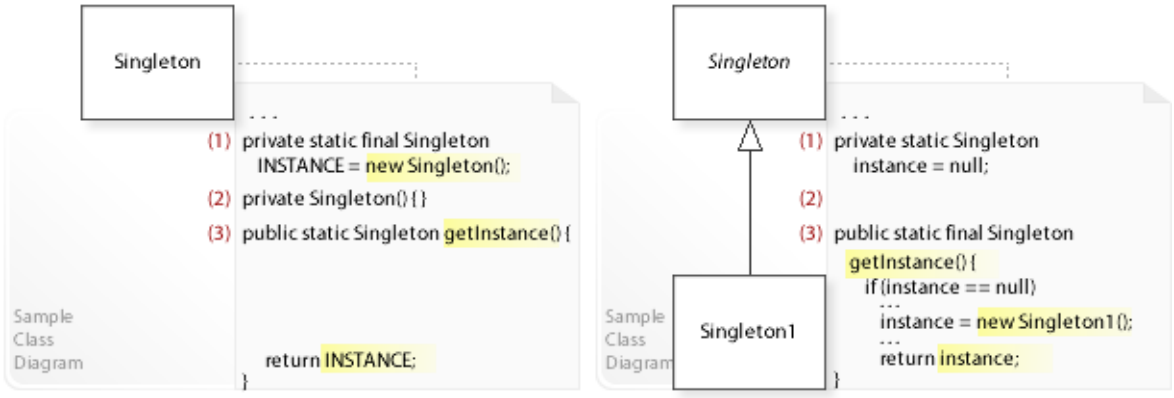
Advantages (+)

- 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 clients dependent on the concrete singleton class.
  - This stops client classes from being reusable and testable.
  - "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]
  - See also Abstract Factory / Sample Code / Example 3 / Creating families of objects.
- Can cause problems in a multi-threaded environment.
  - In multi-threaded applications, a singleton that holds mutable data (i.e., data that can be changed after the object is created) must be implemented carefully (synchronized).

Implementation



Implementation Issues

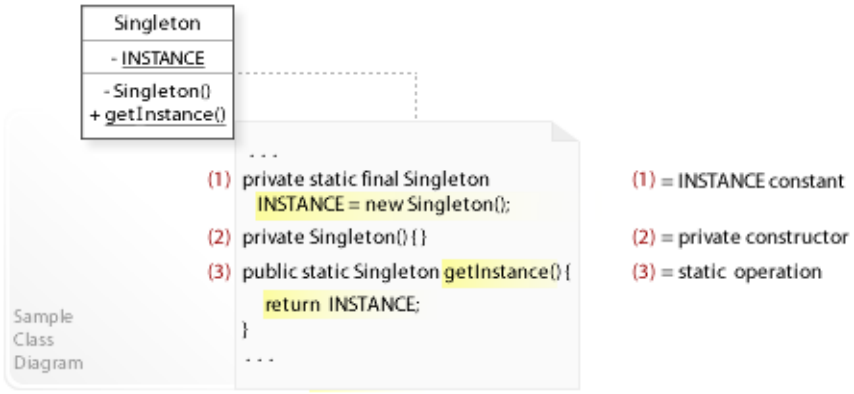
Variant 1: **Subclassing not possible.**

- (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 neither be instantiated (from outside the class) nor subclassed (subclasses need/call the constructor of their parent class).
- (3) The public static `getInstance()` operation returns the INSTANCE constant.
  Clients can access the sole instance easily by calling `Singleton.getInstance()`.
- See also Flyweight and State / Sample Code.

Variant 2: **Subclassing possible.**

- (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.
- (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, etc.
  Note that operations declared `final` can't be redefined by subclasses.
- See also Abstract Factory / Sample Code / Example 3 / Creating families of objects.

Sample Code 1



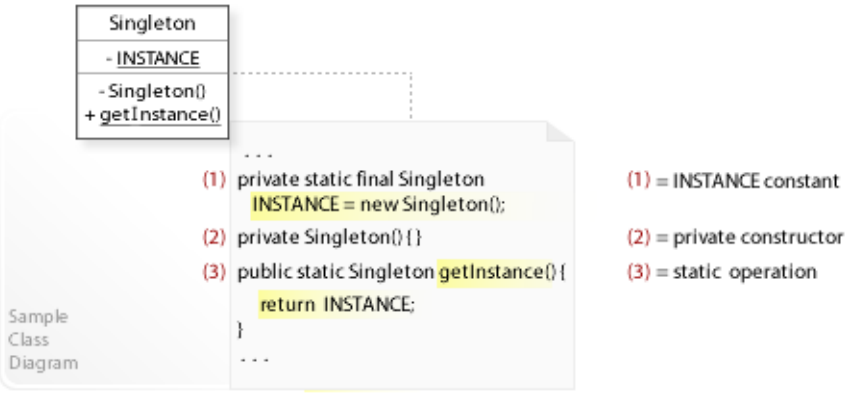
Basic Java code for implementing the sample UML diagram.

```
 1  package com.sample.singleton.basic;
 2  public class MyApp {
 3      public static void main(String[] args) {
 4          Singleton ref1 = null, ref2 = null;
 5          ref1 = Singleton.getInstance();
 6          ref2 = Singleton.getInstance();
 7          if (ref1 == ref2) {
 8              // The two singleton references are identical.
 9              System.out.println("Singleton instantiated only once.");
10          }
11      }
12  }
```

```
Singleton instantiated only once.
```

```
 1  package com.sample.singleton.basic;
 2  public class Singleton {
 3      // (1) INSTANCE constant that holds the sole instance.
 4      private static final Singleton INSTANCE = new Singleton();
 5      // (2) Private (hidden) constructor.
 6      private Singleton() { }
 7      // (3) Static operation that returns the sole instance.
 8      public static Singleton getInstance() {
 9          return INSTANCE;
10      }
11  }
```

Related Patterns

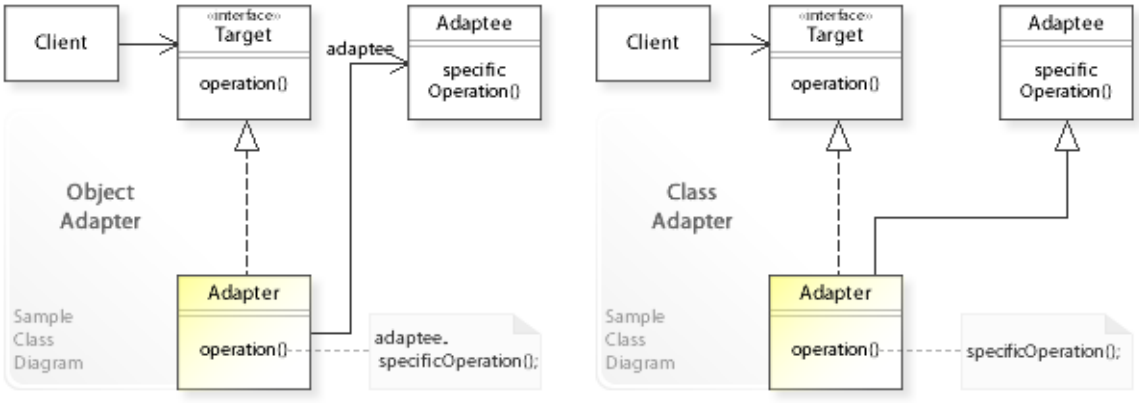


Key Relationships

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

Part III. Structural Patterns

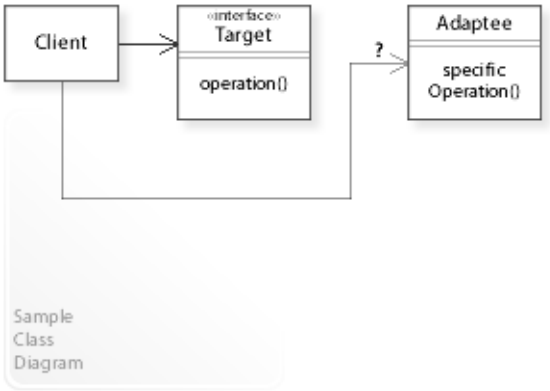
Intent



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 a class be reused that has not the interface clients require?*
  - *How can classes work together that have incompatible interfaces?*
- Often a class (`Adaptee`) can not be reused only because its interface doesn't conform to the interface (`Target`) a client requires.
- The Adapter pattern describes how to solve such problems:
  - *Convert the interface of a class into another interface clients expect.*
    Define a separate `Adapter` class that converts the interface of a class (`Adaptee`) into another interface (`Target`) clients require.
  - Clients that require a `Target` interface can work through an `Adapter` to work with classes that have an incompatible (not the `Target`) interface.

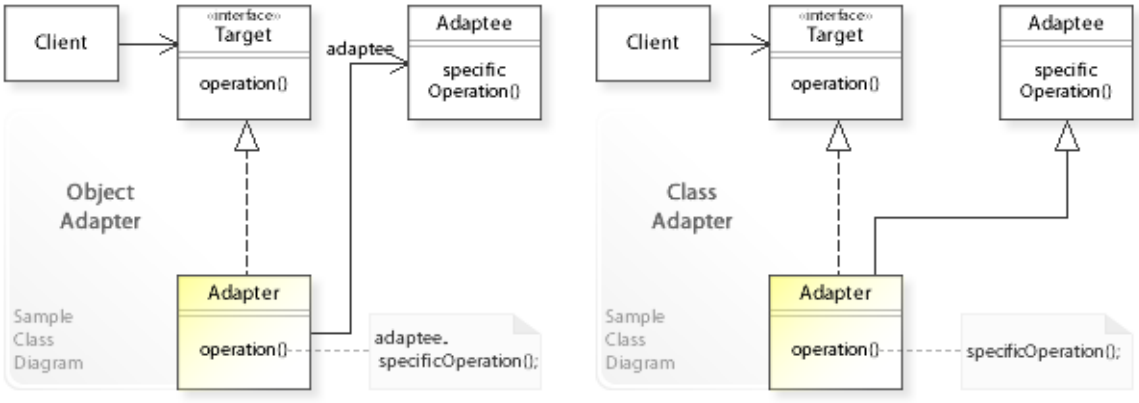
Problem



The Adapter design pattern solves problems like:
How can a class be reused that has not the interface clients require? How can classes work together that have incompatible interfaces?
See Applicability section for all problems Adapter can solve. See Solution section for how Adapter solves the problems.

- Often a class (`Adaptee`) can not be reused only because its interface doesn't conform to the interface (`Target`) a client requires.
- An inflexible way to solve this problem is to change the class so that its interface conforms to the required interface.
  But it's impossible to change a class each time another interface is needed.
- *That's the kind of approach to avoid if we want to reuse a class (`Adaptee`) that has an incompatible interface independently from (without having to change) the class.*
- In the Strategy design pattern, for example, clients require (depend on) a `Strategy` interface for performing an algorithm.
  To perform a specific algorithm, it should be possible that clients can work with a class that has not the `Strategy` interface (see Sample Code / Example 2).
- For example, designing reusable classes.
  To make a class more reusable, its interface should be adaptable so that clients that want to reuse the class can specify the interface they require (built-in interface adaptation).

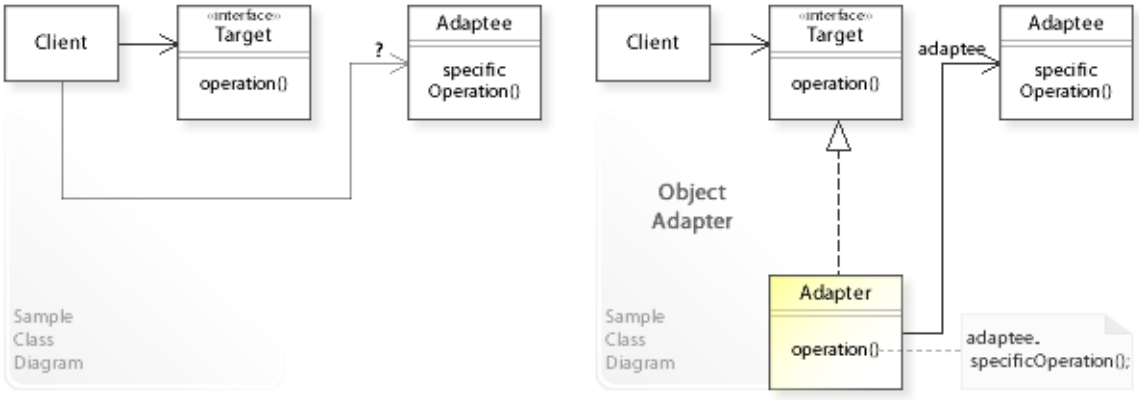
Solution



The Adapter design pattern provides a solution:
Define a separate `Adapter` class that converts the interface of a class (`Adaptee`) into another interface (`Target`) clients require.
Work through an `Adapter` to work with classes that have not the required interface.
Describing the 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 (object version) is to work through a separate `Adapter` object that adapts the interface of an (already existing) object.
  Clients do not know whether they are working with a `Target` object or an `Adapter`.
- **There are two ways to define an `Adapter`:**
  - **Class Adapter**: Uses inheritance to implement a `Target` interface in terms of (*by inheriting from*) an `Adaptee` class.
  - **Object Adapter**: Uses object composition to implement a `Target` interface in terms of (*by delegating to*) an `Adaptee` object (`adaptee.specificOperation()`).
  - A class adapter commits to (inherits from) an `Adaptee` class at compile-time. An object adapter is more flexible because it commits (delegates) to an `Adaptee` object at run-time (see also Sample Code / Example 1).
  - There exists a wide range of possible adaptations, from simply changing the name of an operation to supporting an entirely different set of operations.
- **Built-in interface adaptation.**
  To make a class (`Adaptee`) more reusable, it can be configured with an adapter object that converts the interface of the class into the interface clients (that want to reuse the class) require.

Motivation 1



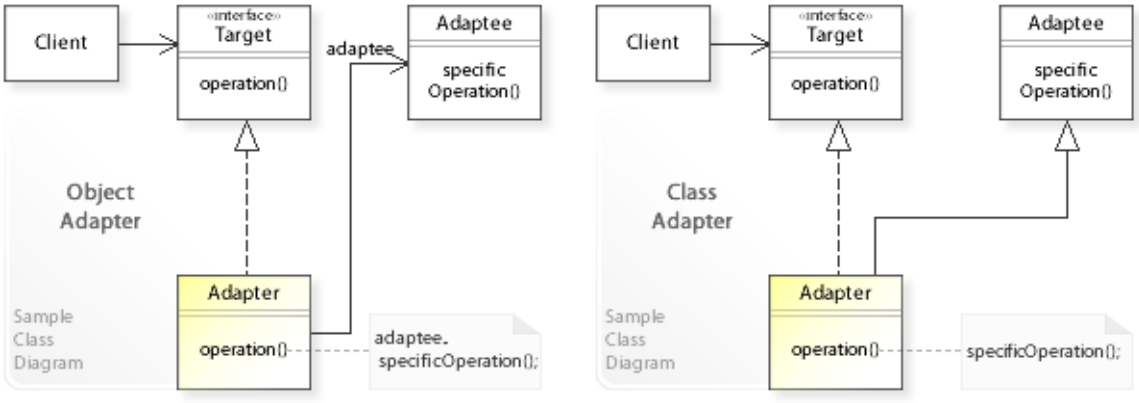
Consider the left design (problem):

- No adapter.
  Clients can not reuse `Adaptee`.
  - Clients that require (depend on) a `Target` interface can not reuse `Adaptee` only because its interface doesn't conform to the `Target` interface.

Consider the right design (solution):

- Working through an adapter.
  Clients can reuse `Adaptee`.
  - Clients that require (depend on) a `Target` interface can reuse `Adaptee` by working through an `Adapter`.

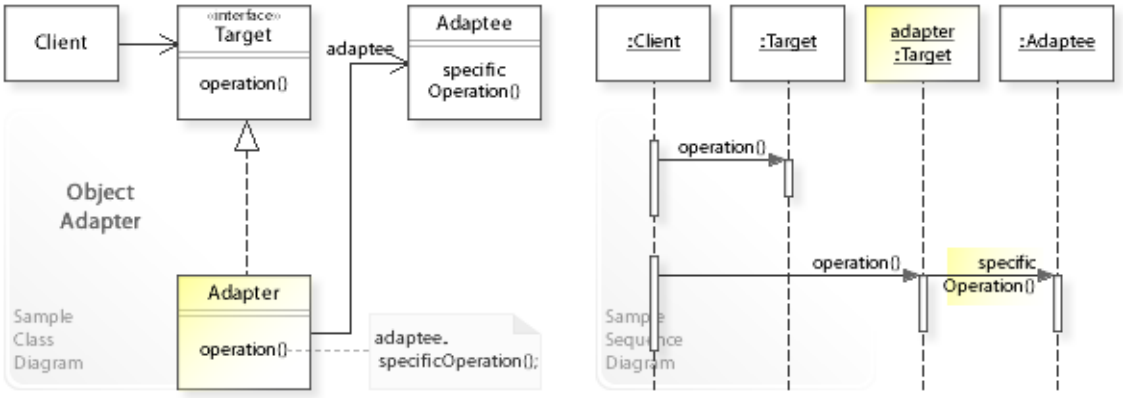
Applicability



Design Problems

- **Class Adapter (Compile-Time Adaptation)**
  - How can a class be reused that has not the interface clients require?
  - How can classes work together that have incompatible interfaces?
  - How can an alternative interface be provided for a class?
- **Object Adapter (Run-Time Adaptation)**
  - How can an object be reused that has not the interface clients require?
  - How can objects work together that have incompatible interfaces?
  - How can an alternative interface be provided for an object?

Structure, Collaboration



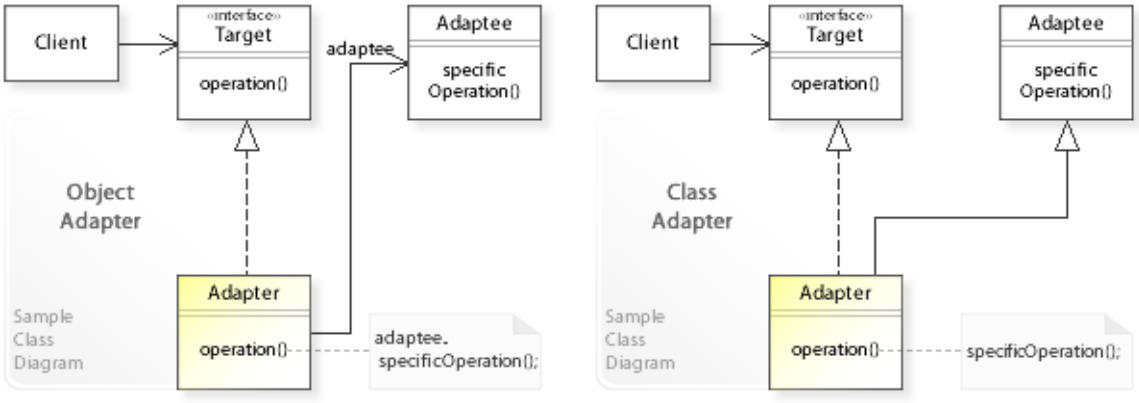
Static Class Structure

- `Client`
  - Refers to (depends on) the `Target` interface.
- `Target`
  - Defines an interface the `Client` class requires.
- `Adapter`
  - Implements the `Target` interface in terms of `Adaptee`.
- `Adaptee`
  - Defines a class that gets adapted.

Dynamic Object Collaboration

- In this sample scenario, a `Client` object works with a `Target` object directly and through an `adapter` (of type `Target`) with an `Adaptee` object.
- The interaction starts with the `Client` object that calls `operation()` on a `Target` object, which performs the request and returns to the `Client`.
- Thereafter, the `Client` calls `operation()` on an `adapter` object (of type `Target`).
- `adapter` calls `specificOperation()` on an `Adaptee` object, which performs the request and returns to the `adapter`, which in turn returns to the `Client`.
- `adapter` can do work of its own before and/or after forwarding a request to `Adaptee`.
- See also Sample Code.

Consequences

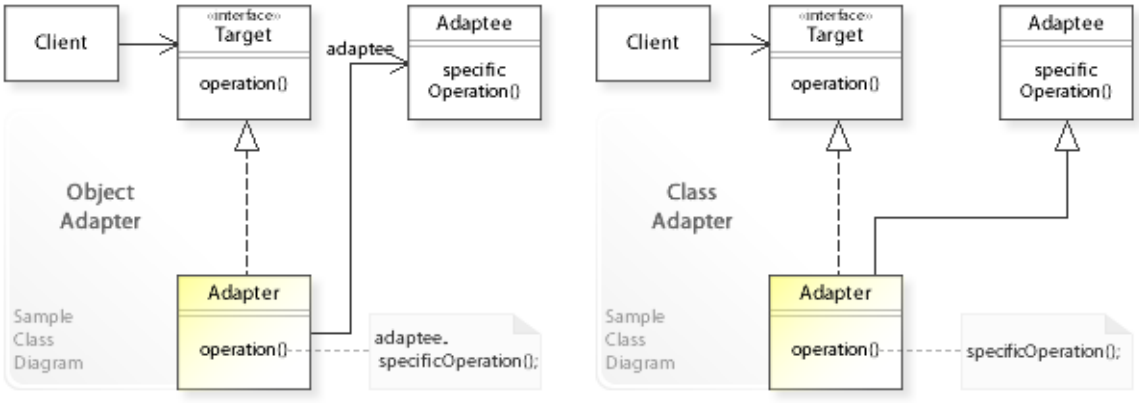


Advantages (+)

- Supports reusing existing functionality.
  - Often an already existing reusable object can not be reused only because its interface does not match the interface a client depends on.
  - By working through an adapter, clients can reuse existing objects that provide the needed functionality but not the needed interface.
- Object adapter is more flexible than class adapter.
  - The class adapter implements a `Target` interface in terms of (by inheriting from) an `Adaptee` class.
  - This commits the class adapter to an `Adaptee` class at compile-time and wouldn't work for other `Adaptee` classes.
  - Furthermore, if the `Adaptee` implementation classes belong to an other application, they are usually hidden and can't be accessed.
  - The object adapter, on the other hand, implements a `Target` interface in terms of (by delegating to) an `Adaptee` object at run-time (independently from `Adaptee` implementation classes).

Disadvantages (–)

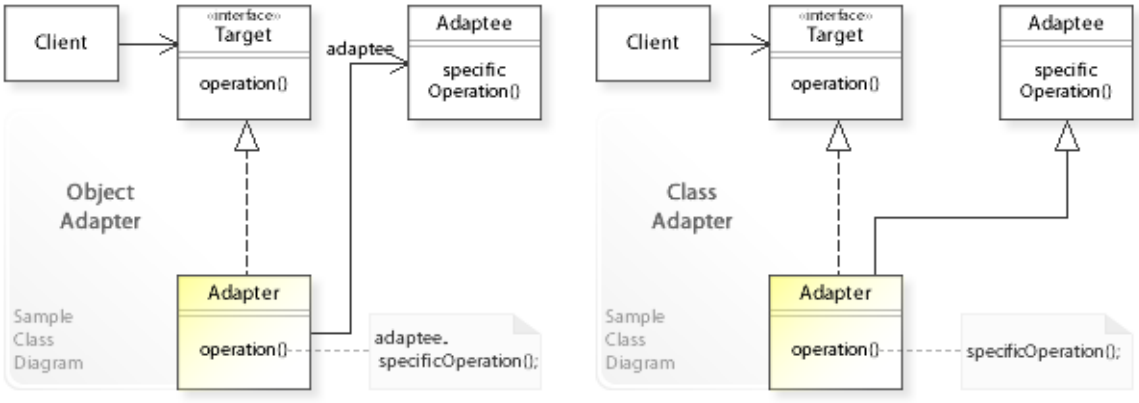
Implementation



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 object adapter
 6          // and configuring it with an Adaptee object.
 7          Target objectAdapter = new ObjectAdapter(new Adaptee());
 8          System.out.println("(1) Object Adapter: " + objectAdapter.operation());
 9
10          // Creating a class adapter
11          // that commits to the Adaptee class at compile-time.
12          Target classAdapter = new ClassAdapterAdaptee();
13          System.out.println("(2) Class Adapter : " + classAdapter.operation());
14      }
15  }
```

```
(1) Object Adapter: Hello World from Adaptee!
(2) Class Adapter : Hello World from Adaptee!
```

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

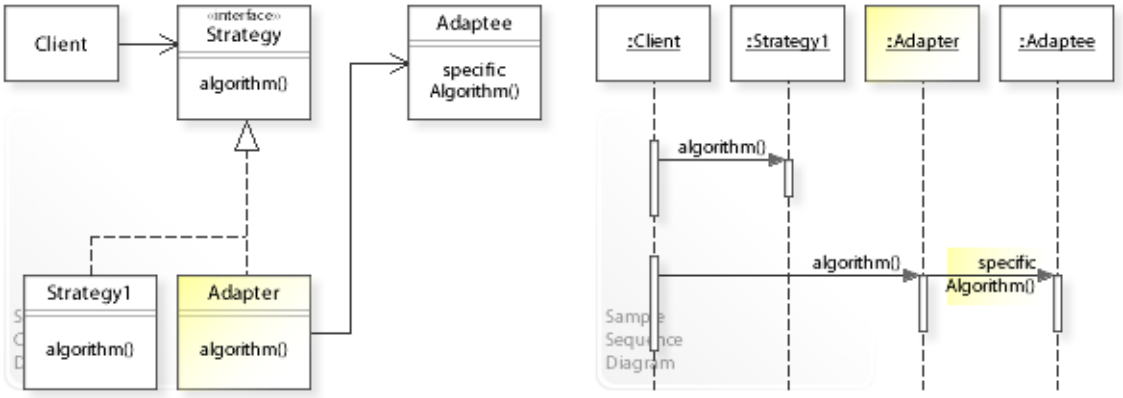
```
 1  package com.sample.adapter.basic;
 2  public class ObjectAdapter implements Target {
 3      private Adaptee adaptee;
 4
 5      public ObjectAdapter(Adaptee adaptee) {
 6          this.adaptee = adaptee;
 7      }
 8      public String operation() {
 9          // Implementing the Target interface in terms of
10          // (by delegating to) an Adaptee object.
11          return adaptee.specificOperation();
12      }
13  }
```

```
1  package com.sample.adapter.basic;
2  public class ClassAdapterAdaptee extends Adaptee implements Target {
3      public String operation() {
4          // Implementing the Target interface in terms of
5          // (by inheriting from) the Adaptee class.
6          return specificOperation();
7      }
8  }
```

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

```
6   }
```

Sample Code 2



Basic Java code for implementing Strategy with Adapter.

```
 1  package com.sample.adapter.strategy;
 2  public class Client {
 3      // Running the Client class as application.
 4      public static void main(String[] args) {
 5          Strategy strategy = new Strategy1();
 6          System.out.println("(1) " + strategy.algorithm());
 7
 8          strategy = new Adapter(new Adaptee());
 9          System.out.println("(2) " + strategy.algorithm());
10      }
11  }
```

```
(1) Working with Strategy1 to perform an algorithm!
(2) Working with Adaptee to perform a specific algorithm!
```

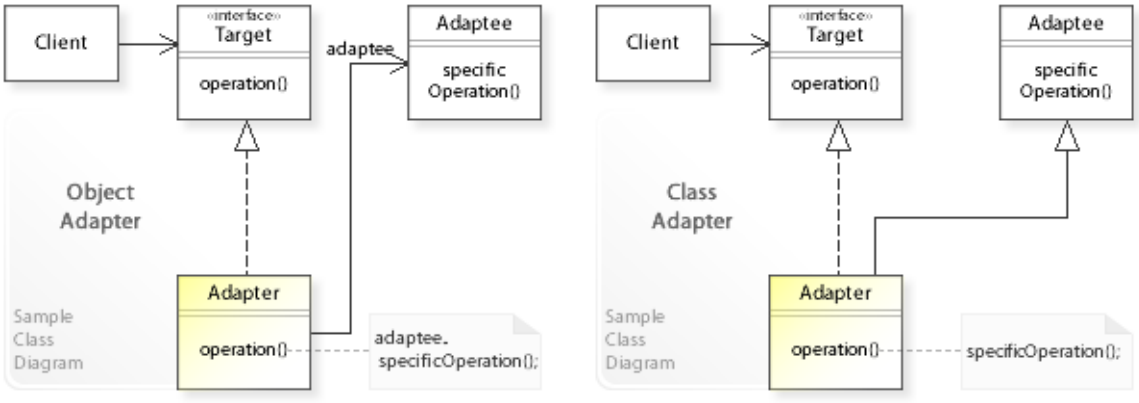
```
 1  package com.sample.adapter.strategy;
 2  public interface Strategy {
 3      String algorithm();
 4  }
```

```
 1  package com.sample.adapter.strategy;
 2  public class Strategy1 implements Strategy {
 3      public String algorithm() {
 4          // Implementing the algorithm.
 5          return "Working with Strategy1 to perform an algorithm!";
 6      }
 7  }
```

```
 1  package com.sample.adapter.strategy;
 2  public class Adapter implements Strategy {
 3      private Adaptee adaptee;
 4
 5      public Adapter(Adaptee adaptee) {
 6          this.adaptee = adaptee;
 7      }
 8      public String algorithm() {
 9          // Implementing the Strategy interface in terms of
10          // (by delegating to) an Adaptee object.
11          return adaptee.specificAlgorithm();
12      }
13  }
```

```
 1  package com.sample.adapter.strategy;
 2  public class Adaptee {
 3      public String specificAlgorithm() {
 4          return "Working with Adaptee to perform a specific algorithm!";
 5      }
 6  }
```

Related Patterns



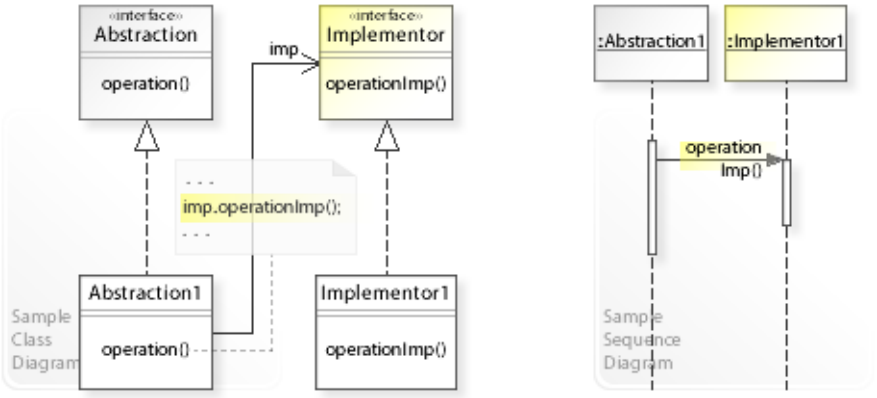
Key Relationships

- **Adapter - Bridge* - Composite - Decorator - Facade - Flyweight* - Proxy**
  These patterns are classified as *structural design patterns*. [GoF, p10]
  - Adapter provides an alternative interface for an (already existing) class or object.
  - Bridge* lets an abstraction and its implementation vary independently.
  - Composite composes (already existing) objects into a tree structure.
  - Decorator provides additional functionality for an (already existing) object.
  - Facade provides an unified interface for (already existing) objects in a subsystem.
  - Flyweight* supports large numbers of fine-grained objects efficiently.
  - Proxy provides additional functionality when accessing an (already existing) object.

Background Information

- *Structural design patterns*
  (shown in the second row of the main menu) are concerned with providing alternative behavior for already existing classes or objects (without touching them).
- Bridge* and Flyweight* should be classified as *behavioral design patterns*
  (shown in the third row of the main menu) that are concerned with designing related classes and interacting objects having a desired behavior.

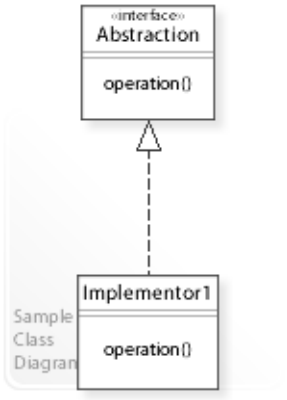
Intent



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 implementation be selected and exchanged at run-time?*
- For example, a reusable application that supports different hardware environments.
  To make an application portable across different hardware environments, it should be possible to select the appropriate hardware-specific implementation 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) an `Implementor` object.

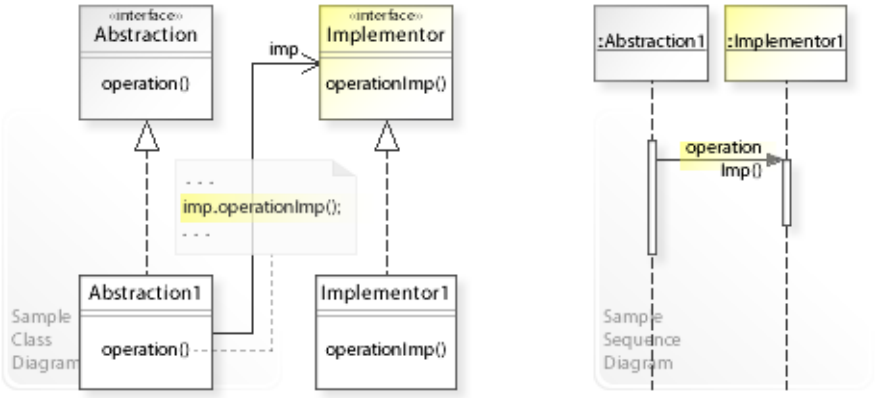
Problem



The Bridge design pattern solves problems like:
How can an abstraction and its implementation vary independently?
How can an implementation be selected and exchanged at run-time?
See Applicability section for all problems Bridge can solve. See Solution section for how Bridge solves the problems.

- The standard way is to implement an abstraction by inheritance, i.e., different (sub)classes (`Implementor1`,…) implement the abstraction (interface) in different ways.
- This commits (binds) the implementation to an abstraction at compile-time and makes it impossible to change the implementation at run-time.
  "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 implementation can be selected and exchanged at run-time instead of committing to an implementation at compile-time.*
- For example, a reusable application that supports different hardware environments.
  To make an application portable across different hardware environments, it should be possible to select the proper hardware-specific implementation at run-time.

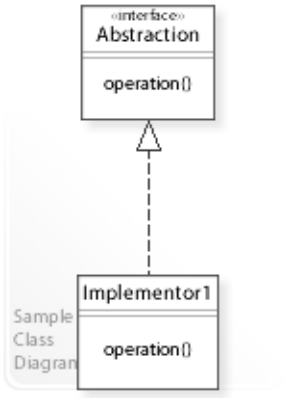
Solution

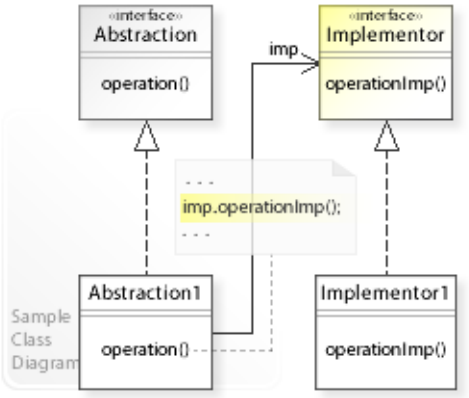


The Bridge design pattern provides a solution:
Define separate inheritance hierarchies for an abstraction and its implementation.
`Abstraction` delegates its implementation to an `Implementor` object
instead of committing to an implementation at compile-time. Describing the Bridge design in more detail is the theme of the following sections.
See Applicability section for all problems Bridge can solve.

- The key idea in this pattern is to separate (decouple) an abstraction from its implementation so that the two can be defined independently from each other.
  The pattern calls the relationship between an abstraction and its implementation 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) an `Implementor` object (`imp.operationImp()`).
  - "Typically the Implementor interface provides only primitive operations, and Abstraction defines higher-level operations based on these primitives." [GoF, p154]
- This enables *compile-time* flexibility (via inheritance).
  Abstraction and implementation can be defined independently from each other.
- **`Abstraction` delegates its implementation to an `Implementor` object**
  (`imp.operationImp()`).
- This enables *run-time* flexibility (via object composition).
  `Abstraction` can be configured with an `Implementor` object, and even more, the `Implementor` object can be exchanged at run-time.

Motivation 1





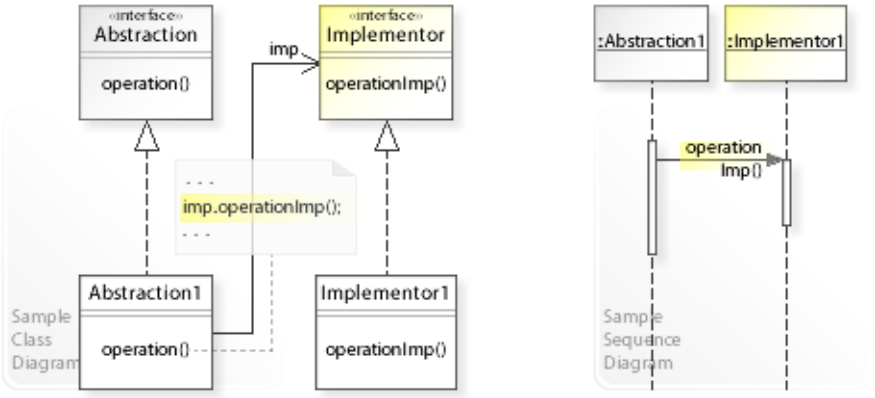
Consider the left design (problem):

- Implementation is coupled to the abstraction.
  - The standard way is to implement an abstraction by inheritance.
  - The `Abstraction` interface is implemented by an `Implementor1` class, which commits (binds) the implementation to the abstraction at compile-time.
  - This makes it impossible to change the implementation at run-time.

Consider the right design (solution):

- Implementation is decoupled from the abstraction.
  - Separate inheritance hierarchies are defined for an abstraction and its implementation.
  - The `Abstraction` interface is implemented in terms of (by delegating to) an `Implementor` object (`imp.operationImp()`).
  - This makes it possible to change the implementation at run-time.

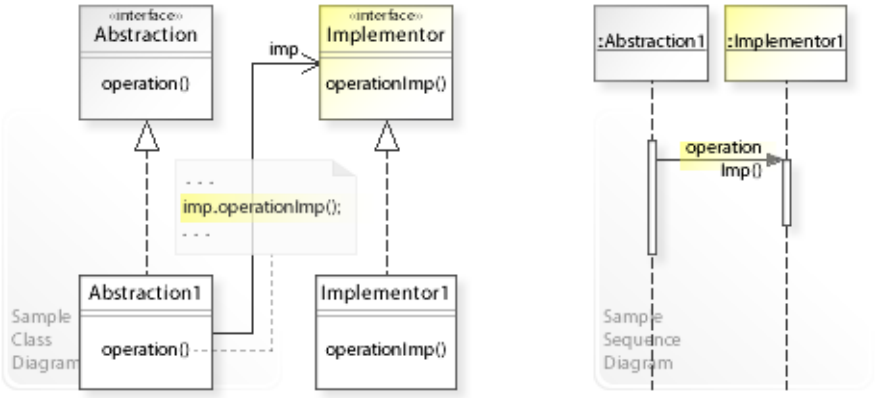
Applicability



Design Problems

- **Defining Abstraction and Implementation Independently**
  - How can an abstraction and its implementation vary independently?
  - How can a compile-time binding between an abstraction and its implementation be avoided?
- **Exchanging Implementations at Run-Time**
  - How can an abstraction be configured with an implementation?
  - How can an implementation be selected and exchanged at run-time?
- **Flexible Alternative to Subclassing**
  - How can a flexible alternative be provided to subclassing for changing an implementation at compile-time?

Structure, Collaboration



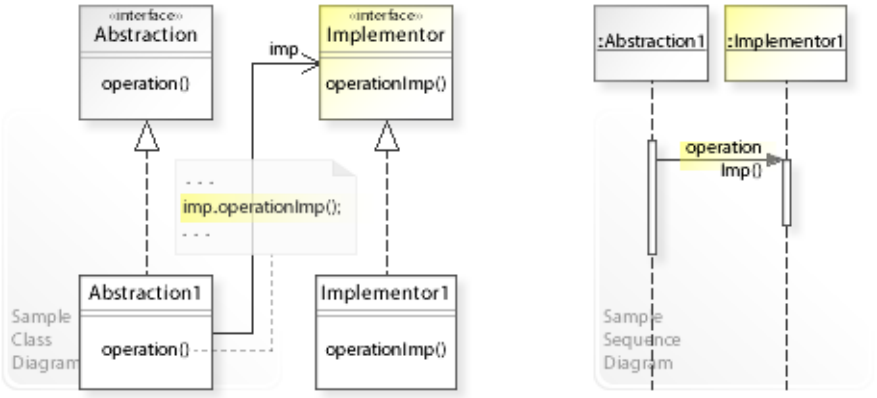
Static Class Structure

- `Abstraction`
  - Defines an interface for an abstraction.
- `Abstraction1`,...
  - Implement the `Abstraction` interface in terms of (by delegating to) the `Implementor` interface (`imp.operationImp()`).
  - Maintains 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 `Abstraction1` object delegates implementation to an `Implementor1` object (by calling `operationImp()` on `Implementor1`).
- See also Sample Code / Example 1.

Consequences



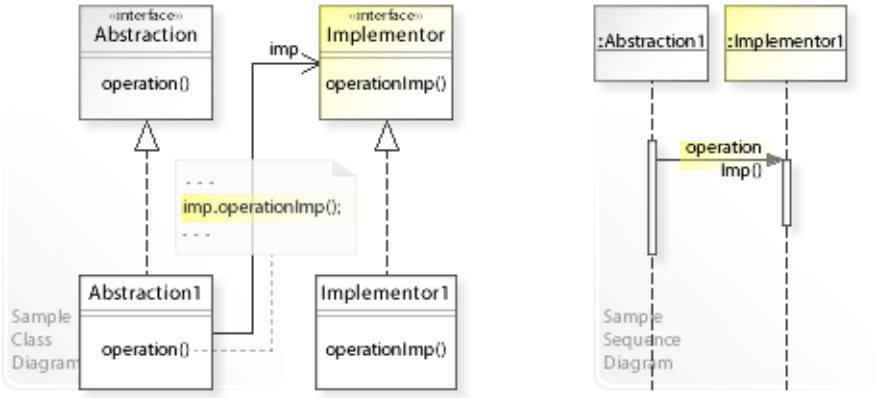
Advantages (+)

- 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 at compile-time and can't be changed at run-time.
  - Furthermore, inheritance would require creating new subclasses for each new abstraction extension class (proliferation of subclasses).
  - Bridge makes it easy to compose abstraction objects and implementation objects dynamically at run-time.

Disadvantages (–)

- Introduces an additional level of indirection.
  - The pattern achieves flexibility by introducing an additional level of indirection (abstraction delegates implementation to a separate `Implementor` object), which makes the abstraction dependent on an `Implementor` object.

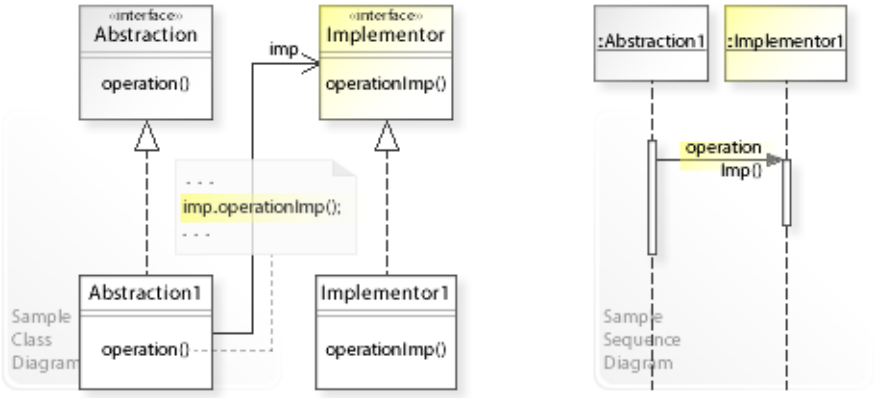
Implementation



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          // Calling an operation on abstraction.
 8          System.out.println(abstraction.operation());
 9      }
10  }
```

```
Abstraction1: Delegating implementation to an implementor.
Implementor1: Hello World1!
```

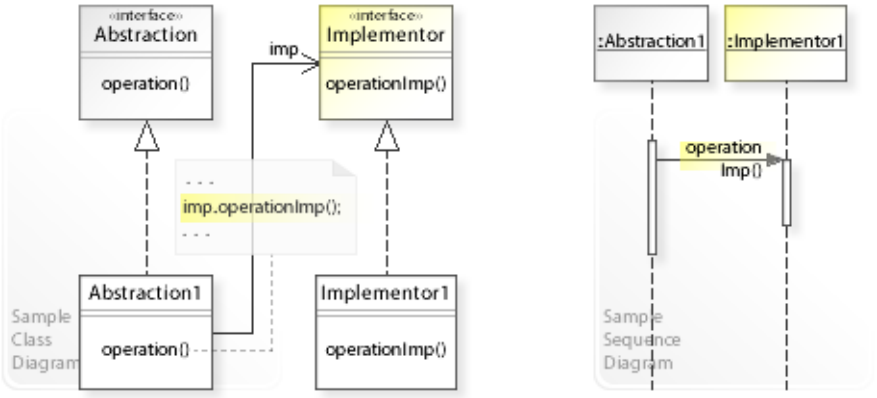
```
 1  package com.sample.bridge.basic;
 2  public interface Abstraction {
 3      String operation();
 4  }
```

```
 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 "Abstraction1: Delegating implementation to an implementor.\n"
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: Hello World1!";
 5      }
 6  }
```

Related Patterns



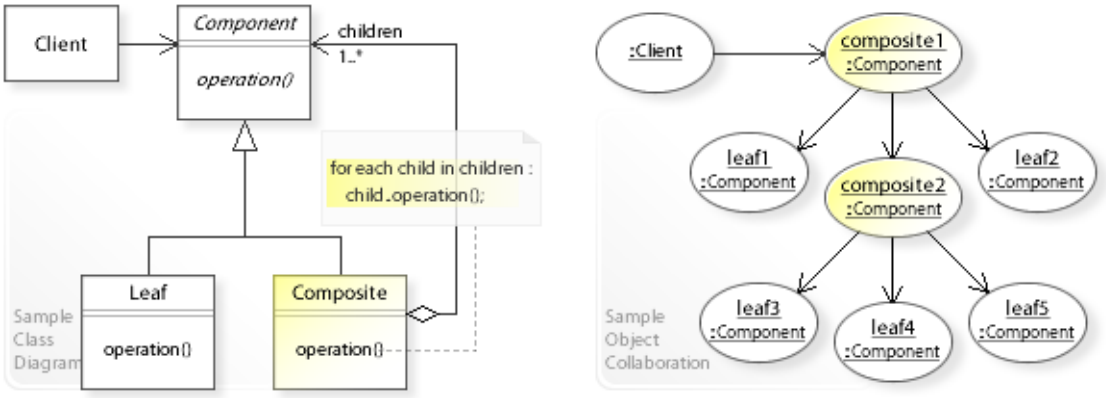
Key Relationships

- **Adapter - Bridge* - Composite - Decorator - Facade - Flyweight* - Proxy**
  These patterns are classified as *structural design patterns*. [GoF, p10]
  - Adapter provides an alternative interface for an (already existing) class or object.
  - Bridge* lets an abstraction and its implementation vary independently.
  - Composite composes (already existing) objects into a tree structure.
  - Decorator provides additional functionality for an (already existing) object.
  - Facade provides an unified interface for (already existing) objects in a subsystem.
  - Flyweight* supports large numbers of fine-grained objects efficiently.
  - Proxy provides additional functionality when accessing an (already existing) object.

Background Information

- *Structural design patterns*
  (shown in the second row of the main menu) are concerned with providing alternative behavior for already existing classes or objects (without touching them).
- Bridge* and Flyweight* should be classified as *behavioral design patterns*
  (shown in the third row of the main menu) that are concerned with designing related classes and interacting objects having a desired behavior.

Intent



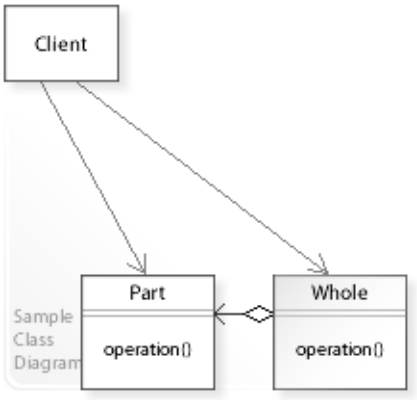
The intent of the Composite design pattern is to:
"Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly." [GoF]
See Problem and Solution sections for a more structured description of the intent.

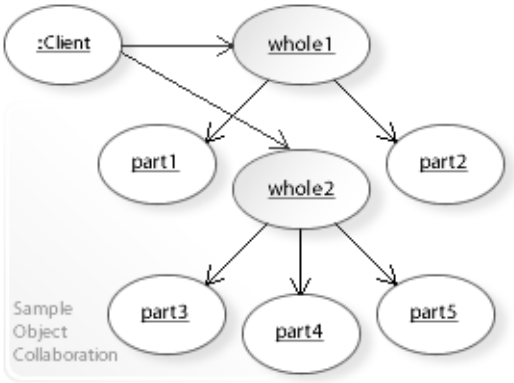
- The Composite design pattern solves problems like:
  - *How can a part-whole hierarchy be represented*
    so that clients can treat individual objects and compositions of objects uniformly?
- *Tree* structures are widely used in object-oriented systems to represent hierarchical object structures like part-whole hierarchies.
- A tree structure consists of individual (`Leaf`) objects and subtree (`Composite`) objects.
  A `Composite` object has children, that is, `Leaf` objects or other (lower-level) `Composite` objects.
- The Composite pattern describes how to solve such problems:
  - *Compose objects into tree structures to represent part-whole hierarchies.*
  - Define separate `Composite` objects that compose the objects in a part-whole hierarchy into tree structures.
  - Clients work through a common `Component` interface to treat `Leaf` and `Composite` objects uniformly, which greatly simplifies clients and makes them easier to implement, change, test, and reuse.

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

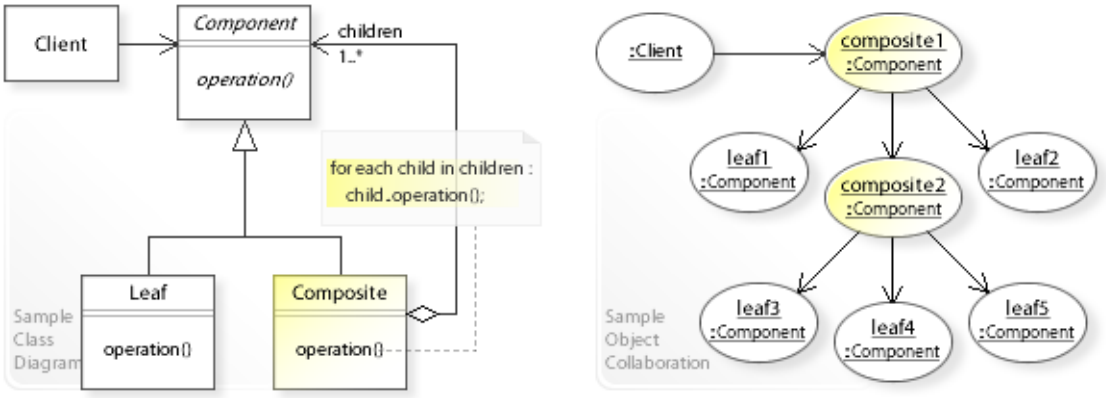




The Composite design pattern solves problems like:
How can a part-whole hierarchy be represented
so that clients can treat individual objects and compositions of objects uniformly?
See Applicability section for all problems Composite can solve. See Solution section for how Composite solves the problems.

- An inflexible way to represent a part-whole hierarchy is to define (1) `Part` objects and (2) `Whole` objects that act as containers for `Part` objects.
  Clients of the hierarchy must treat `Part` and `Whole` objects differently, which makes them more complex especially if the object structure is 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 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).
  It should be possible, for example, to calculate the total price either of an individual part or a complete subcomponent without having to treat part and subcomponent differently (see Sample Code / Example 2 / BOM).
- For example, representing text documents.
  A text document can be organized as part-whole hierarchy consisting of characters, pictures, etc. (parts) and lines, pages, etc. (wholes).
  It should be possible, for example, to treat displaying a particular page or the entire document uniformly.

Solution



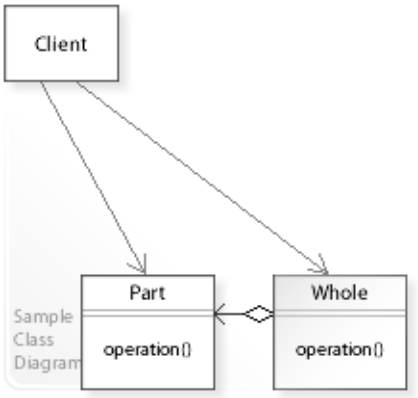
The Composite design pattern provides a solution:
Define separate `Composite` objects that compose the objects of a part-whole hierarchy into a tree structure.
Work through a common `Component` interface to treat `Leaf` and `Composite` objects uniformly.
Describing the Composite design in more detail is the theme of the following sections.
See Applicability section for all problems Composite can solve.

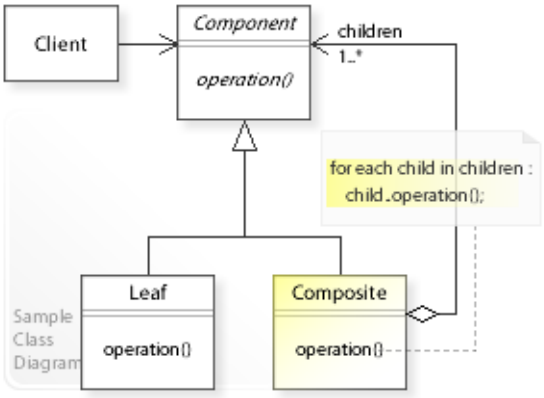
- The key concept in this pattern is to compose `Leaf` objects and `Composite` objects if any into higher-level `Composite` objects recursively *(recursive composition).* The resulting structure is a *tree structure* that represents a part-whole hierarchy.
- **Define separate `Composite` objects:**
  - Define a class (`Composite`) that maintains a container of child `Component` objects (`children`) and forwards requests to these children (`for each child in children: child.operation()`).
  - For implementing child-related operations (like adding or removing child components to or from the container) see Implementation.
- "The key to the Composite pattern is an abstract class [`Component`] that represents *both* primitives and their containers." [GoF, p163]
  - Clients can treat `Leaf` objects and entire `Composite` object structures uniformly (that is, clients do not know whether they are working with `Leaf` or `Composite` objects):
  - If the receiver of a request is a `Leaf`, the request is performed directly.
    If the receiver is a `Composite`, the request is performed on all `Component` objects downwards the hierarchy.
  - This greatly simplifies clients of complex hierarchies and makes them easier to implement, change, test, and reuse.

Background Information

- 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 (see the above Sample Object Collaboration).
  The resulting structure is a *tree structure* that represents a part-whole hierarchy.

Motivation 1





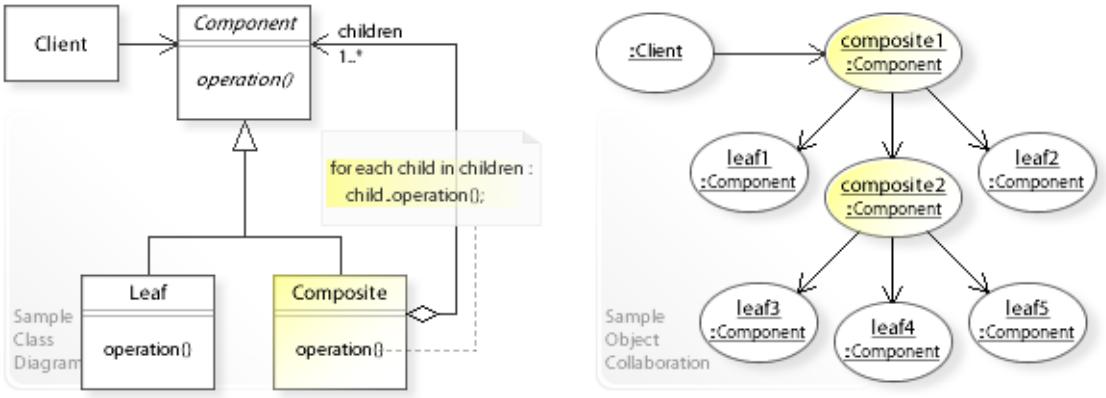
Consider the left design (problem):

- No common interface / direct access.
  Complicated clients.
  - 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):

- Working through a common interface.
  Simplified clients.
  - 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.

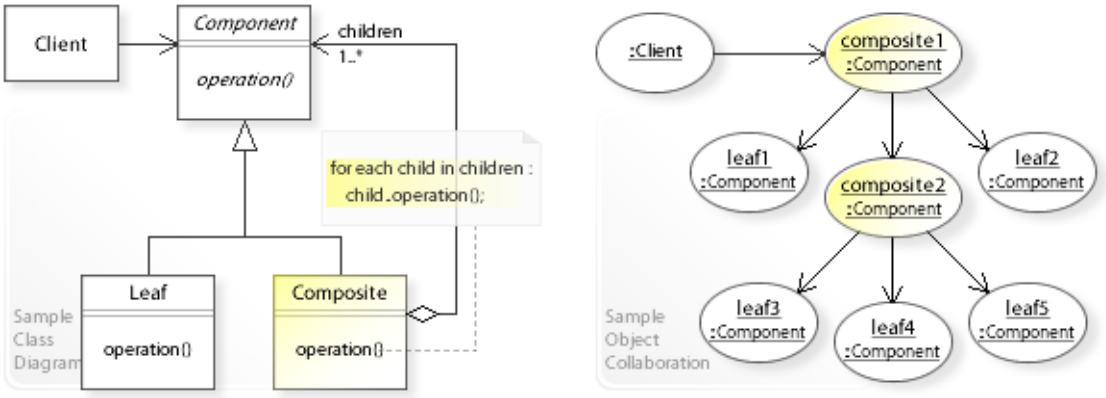
Applicability



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 a part-whole hierarchy be represented
    so that clients can treat the hierarchy as single object?
  - How can a part-whole hierarchy be represented as tree structure?

Structure, Collaboration



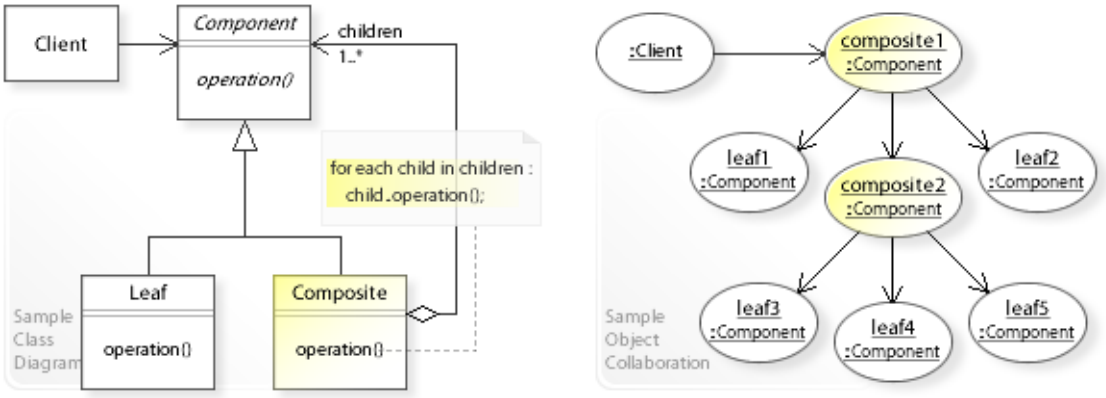
Static Class Structure

- `Client`
  - Refers to the `Component` interface.
- *`Component`*
  - Defines a common interface for `Leaf` and `Composite` objects.
- `Leaf`
  - Defines individual objects that get composed.
- `Composite`
  - Maintains a container of child `Component` objects (`children`).
  - Forwards requests to these children (`for each child in children: child.operation()`).

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 `Component` objects (lower-level `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



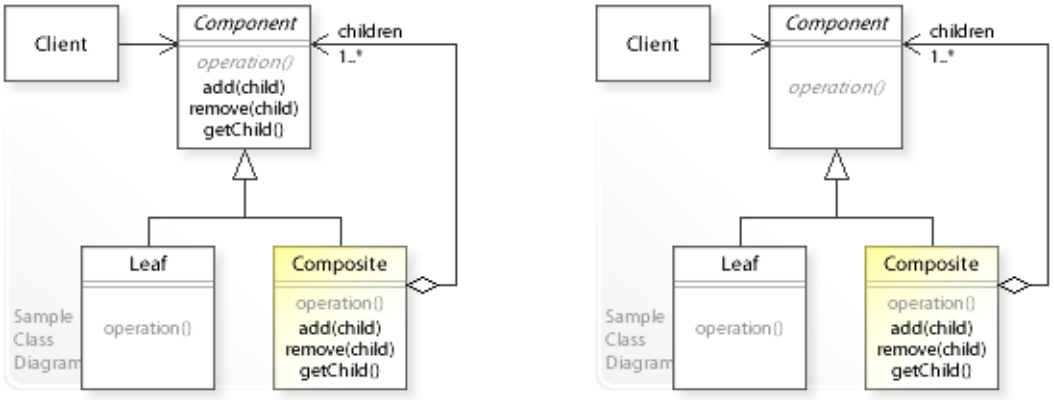
Advantages (+)

- 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 `Composite` or `Leaf` classes are added or existing ones are extended.
- Allows building and changing complex hierarchies dynamically at run-time.
  - The pattern shows how to apply recursive composition to build complex hierarchical object structures dynamically at run-time.

Disadvantages (–)

- Uniformity versus type safety.
  - There are two main design variants to implement child-related operations: design for *uniformity* and design for *type safety* (see Implementation).
  - The Composite pattern emphasizes uniformity over type safety.

Implementation



Implementation Issues
Implementing Child-Related Operations
Adding a child component to the container (`add(child)`), removing a child component from the container (`remove(child)`), and accessing a child component (`getChild()`).

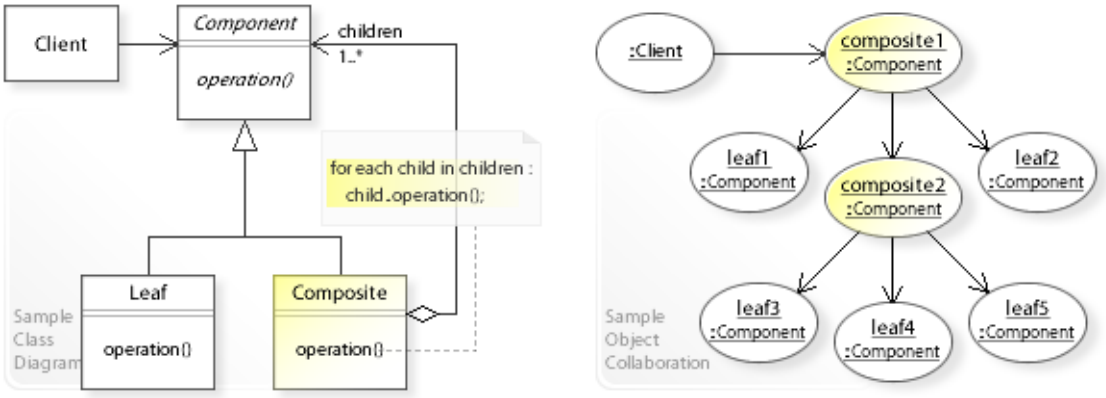
Variant 1: **Design for Uniformity**

- The only way to provide *uniformity* is to define child-related operations 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-related operations like "do nothing" or "throw an exception". The `Leaf` class inherits the default implementations, and `Composite` must redefine them.
- Uniformity is useful for dynamic structures because clients often need to perform child-related operations (in a document editor, for example, where the object structure is dynamically created and changed each time a document is changed).
- The Composite design pattern emphasizes uniformity over type safety.

Variant 2: **Design for Type Safety**

- The only way to provide *type safety* is to define child-related operations solely in the `Composite` class.
- This enables type safety because we can rely on the type system to enforce type constraints (for example, that clients can not perform child-related 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 to perform child-related 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 a 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          // Performing an operation on composite1
16          // (walking down the entire hierarchy).
17          System.out.println("(1) " + composite1.operation());
18
19          // Performing an operation on composite2
20          //(walking down the subtree).
21          System.out.println("(2) " + composite2.operation());
22
23      }
24  }
```

```
(1) composite1 leaf1 composite2 leaf3 leaf4 leaf5 leaf2
(2) composite2 leaf3 leaf4 leaf5
```

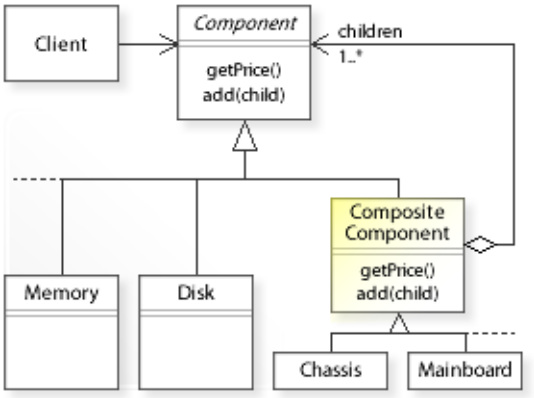
```
 1  package com.sample.composite.basic;
 2  import java.util.Collections;
 3  import java.util.Iterator;
 4  public abstract class Component {
 5      private String name;
 6      public Component(String name) {
 7          this.name = name;
 8      }
 9      public abstract String operation();
10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

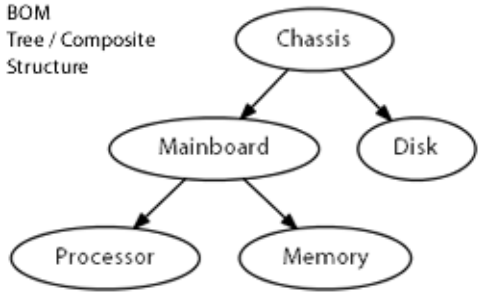
```
 1  package com.sample.composite.basic;
 2  public class Leaf extends Component {
```

```
    public Leaf(String name) {
        super(name);
    }
    public String operation() {
        return getName();
    }
}
```

```
package com.sample.composite.basic;
import java.util.*;
public class Composite extends Component {
    private List<Component> children = new ArrayList<Component>();

    public Composite(String name) {
        super(name);
    }
    public String operation() {
        Iterator<Component> it = children.iterator();
        String str = getName();
        Component child;
        while (it.hasNext()) {
            child = it.next();
            str += child.operation();
        }
        return str;
    }
    // Overriding the default implementation.
    @Override
    public boolean add(Component child) {
        return children.add(child);
    }
    @Override
    public Iterator<Component> iterator() {
        return children.iterator();
    }
}
```

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 child) {
21          return children.add(child);
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  }
```

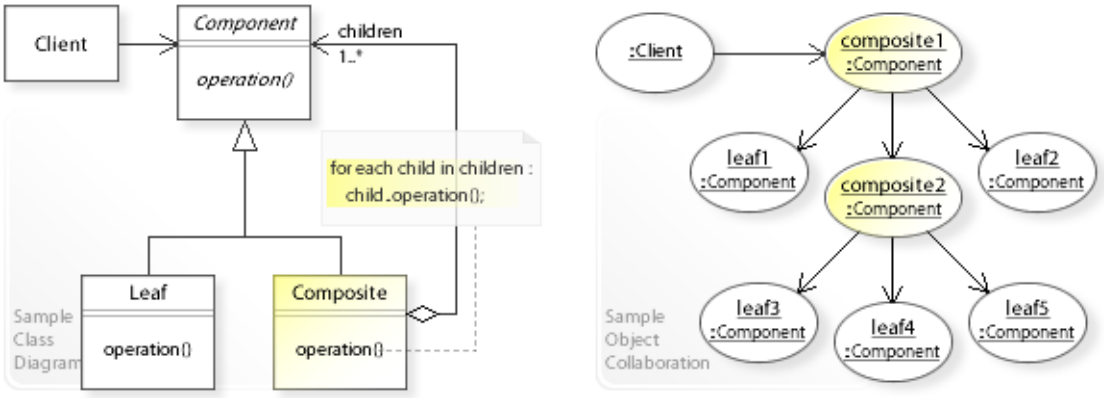
```
1  package com.sample.composite.bom;
2  public class Mainboard extends CompositeComponent {  // Composite
3      public Mainboard(String name, long price) {
4          super(name, price);
5      }
6  }
```

```
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



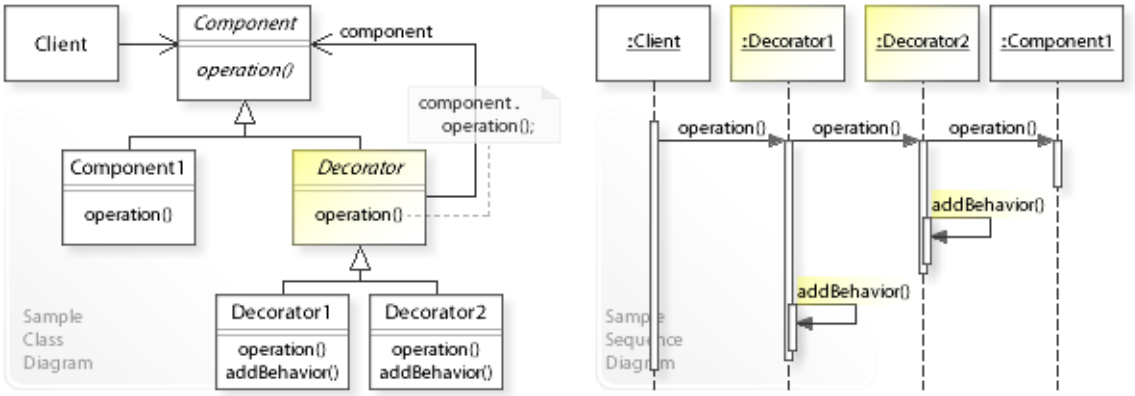
Key Relationships

- **Adapter - Bridge* - Composite - Decorator - Facade - Flyweight* - Proxy**
  These patterns are classified as *structural design patterns*. [GoF, p10]
  - Adapter provides an alternative interface for an (already existing) class or object.
  - Bridge* lets an abstraction and its implementation vary independently.
  - Composite composes (already existing) objects into a tree structure.
  - Decorator provides additional functionality for an (already existing) object.
  - Facade provides an unified interface for (already existing) objects in a subsystem.
  - Flyweight* supports large numbers of fine-grained objects efficiently.
  - Proxy provides additional functionality when accessing an (already existing) object.
- **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.

Background Information

- *Structural design patterns*
  (shown in the second row of the main menu) are concerned with providing alternative behavior for already existing classes or objects (without touching them).
- Bridge* and Flyweight* should be classified as *behavioral design patterns*
  (shown in the third row of the main menu) that are concerned with designing related classes and interacting objects having a desired behavior.

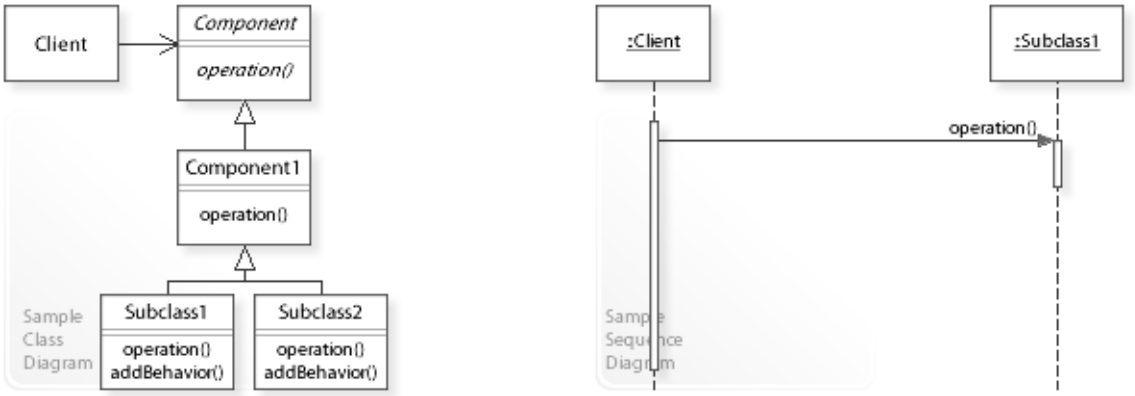
Intent



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 at run-time?*
- "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]

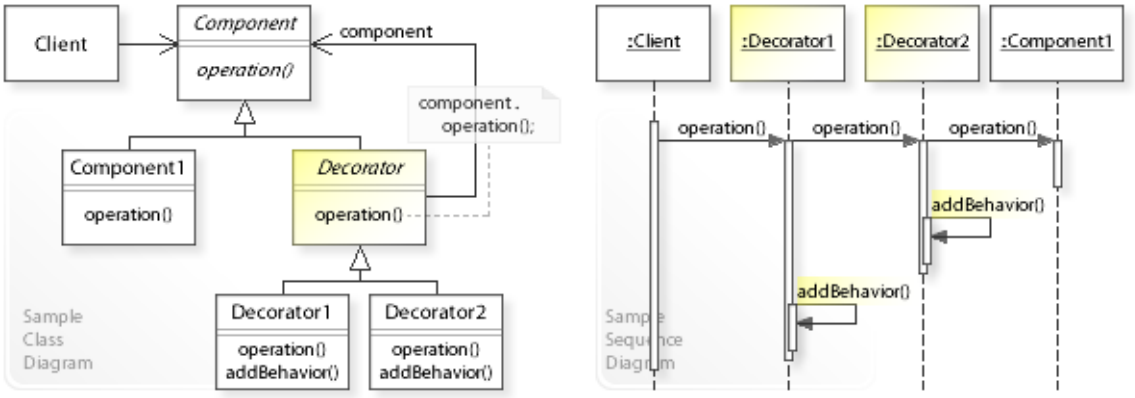
Problem



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 at run-time?
See Applicability section for all problems Decorator can solve. See Solution section for how Decorator solves the problems.

- Subclassing is the standard way to extend the functionality of (add responsibilities to) a class statically at compile-time.
  Once a subclass (`Subclass1`) is instantiated, the functionality is bound to the instance for its life-time and can't be changed at run-time.
- *That's the kind of approach to avoid if we want to extend the functionality of an object at run-time instead of extending the functionality of a class at compile-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.

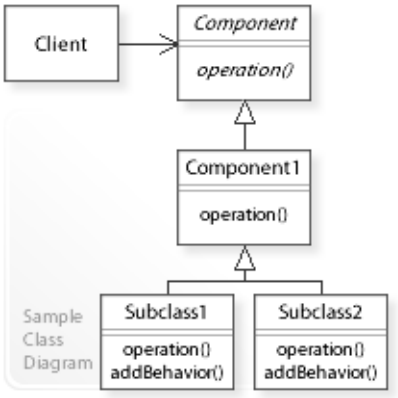
Solution

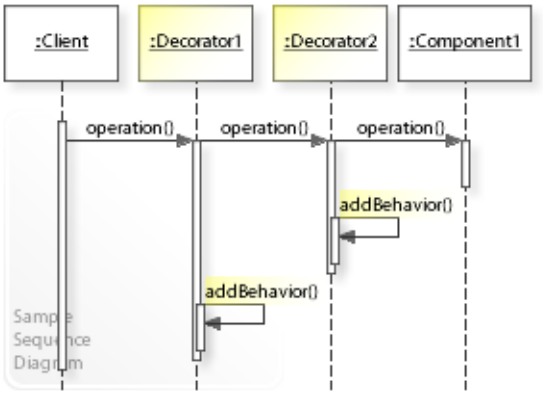


The Decorator design pattern provides a solution:
Define separate `Decorator` objects that add responsibilities to an object.
Work through `Decorator` objects to extend the functionality of an object at run-time.
Describing the Decorator design in more detail is the theme of the following sections.
See Applicability section for all problems Decorator can solve.

- The key idea in this pattern is to work through separate `Decorator` objects that 'decorate' (add responsibilities to) an (already existing) object.
  A decorator implements the `Component` interface transparently so that it can act as *transparent enclosure* of the component that gets decorated.
  "Clients generally can't tell whether they're dealing with the component or its enclosure […]." [GoF, p44]
- **Define separate `Decorator` objects:**
  - Define a class (`Decorator`) that maintains a reference to a `Component` object (`component`) and forwards requests to this component (`component.operation()`).
  - Define subclasses (`Decorator1`,…) that implement additional functionality (`addBehavior()`) to be performed before and/or after forwarding a request.
- Because decorators are transparent enclosures of the decorated component, they can be nested recursively to add an open-ended number of responsibilities.
  Changing the order of decorators allows adding any combinations of responsibilities.
  In the above sequence diagram, for example, a client works through two nested decorator objects that add responsibilities to a `Component1` object (after forwarding the request).

Motivation 1





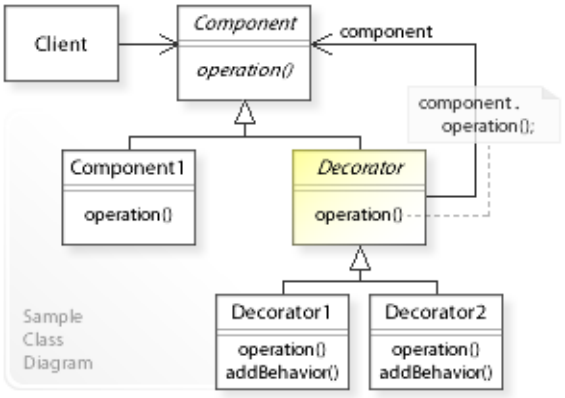
Consider the left design (problem):

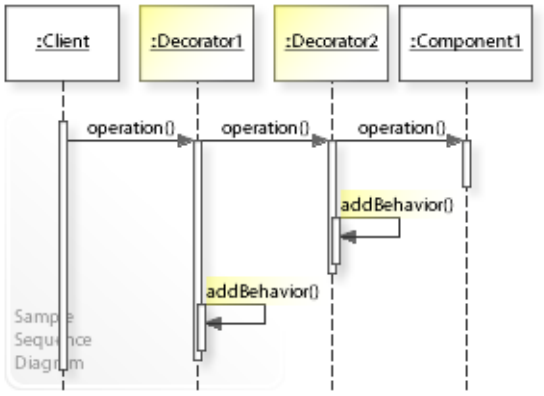
- Extending functionality at compile-time.
  - Subclasses implement additional responsibilities.
  - Once a subclass is instantiated, the responsibility is bound to the instance for its life-time and can't be changed at run-time.
- Explosion of subclasses.
  - Extending functionality by subclassing requires 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):

- Extending functionality at run-time.
  - `Decorator` objects implement additional responsibilities.
  - Clients can work through different `Decorator` objects to add different 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





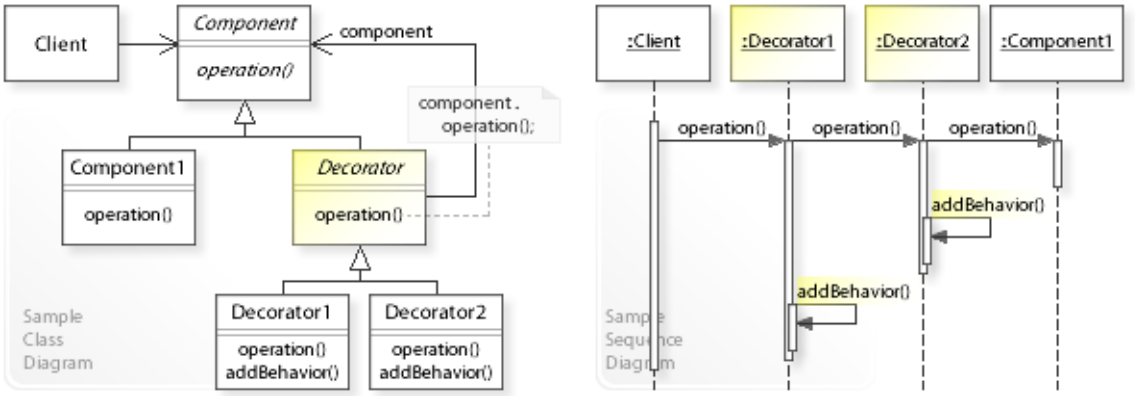
Design Problems

- **Extending Functionality at Run-Time**
  - How can responsibilities be added to (and withdrawn from) an object dynamically?
  - How can the functionality of an object be extended at run-time?
  - How can a simple class be defined that is extended at run-time instead of implementing all foreseeable functionality in a complex class?
- **Flexible Alternative to Subclassing**
  - How can a flexible alternative be provided to subclassing for extending the functionality of a class at compile-time?

Refactoring Problems

- **Inflexible Code**
  - How can classes that include hard-wired extensions (compile-time implementation dependencies) be refactored? *Move Embellishment to Decorator (144)* [JKerievsky05]

Structure, Collaboration



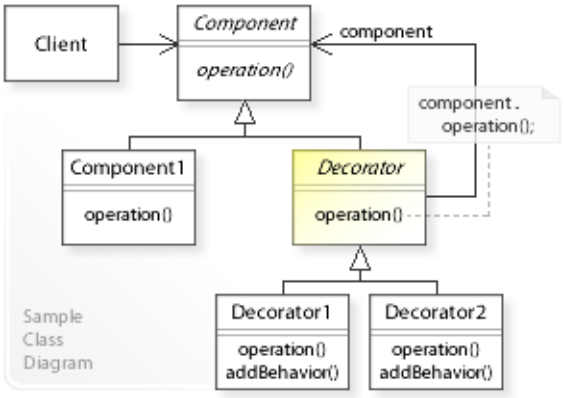
Static Class Structure

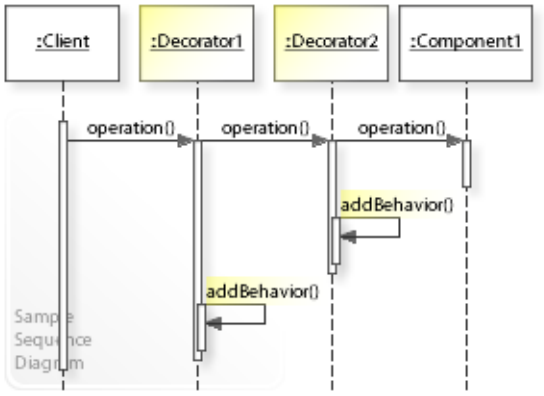
- `Client`
  - Refers to the `Component` interface.
- *`Component`*
  - Defines a common interface for `Component1` and `Decorator` objects.
- `Component1`
  - Defines objects that get decorated.
- *`Decorator`*
  - Maintains a reference to a `Component` object (`component`).
  - Forwards requests to this component (`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 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





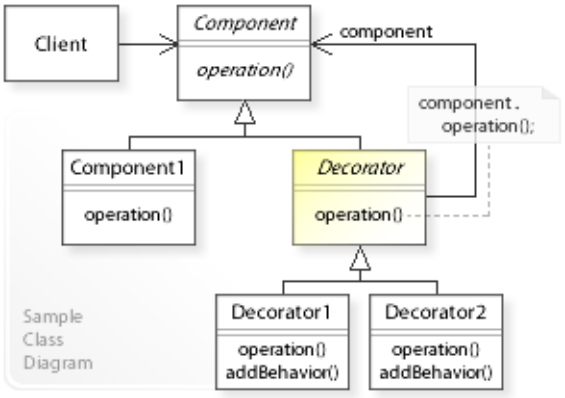
Advantages (+)

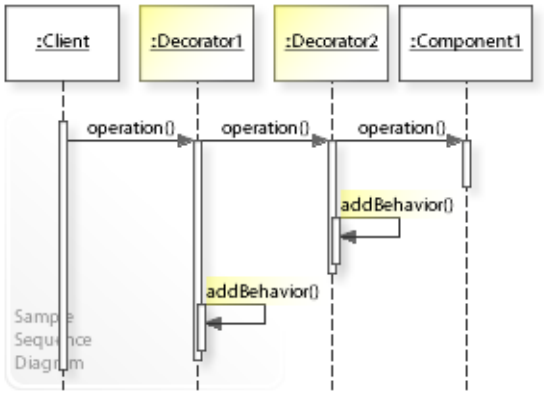
- Provides a flexible alternative to subclassing.
  - Decorator provides a flexible alternative to extending functionality via subclassing.
  - It's easy to combine (mix, sort, duplicate, etc.) functionalities by collaborating with different decorators.
  - Subclassing would require creating a new subclass for each new combination of functionalities.
- Allows an open-ended number of added functionalities.
  - Because decorators are transparent enclosures of the decorated 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 its decorators.
- 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, applications that depend on object identity should not use decorators.

Implementation

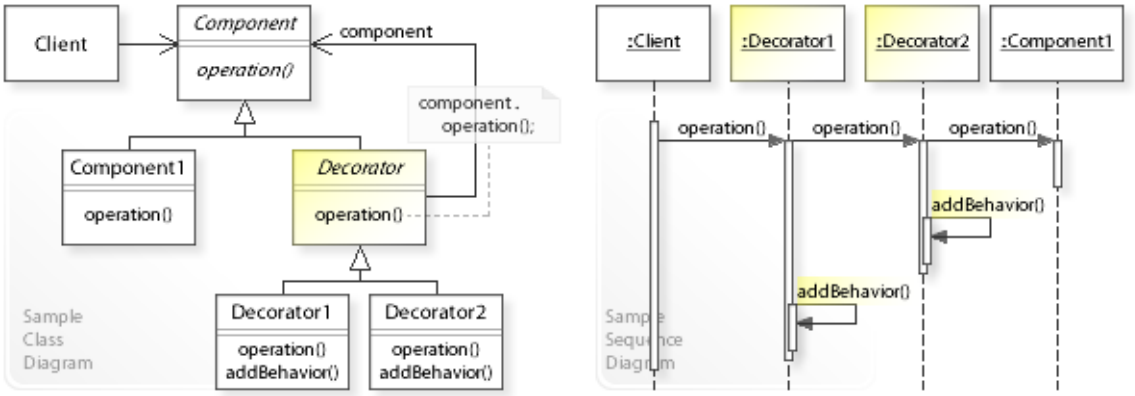




Implementation Issues

- **Interface Conformance**
  - The key to the Decorator is
    (1) to maintain a reference (`component`) to the decorated object and
    (2) to implement the interface of the decorated object transparently by forwarding all requests to it (`component.operation()`).
  - This is called a *transparent enclosure*.
    "Clients generally can't tell whether they're dealing with the component or its enclosure […]" [GoF, p44]

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          // Working with the component directly.
 6          Component component = new Component1();
 7          System.out.println("(1) " + component.operation());
 8          // Working through decorators.
 9          component = new Decorator1(new Decorator2(component));
10          System.out.println("(2) " + component.operation());
11      }
12  }
```

```
(1) Hello World from Component1!
(2) *** === Hello World from Component1! === ***
```

```
 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 from Component1!";
 5      }
 6  }
```

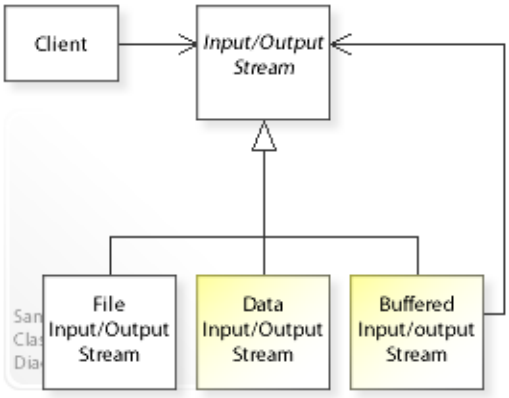
```
 1  package com.sample.decorator.basic;
 2  public abstract class Decorator extends Component {
 3      Component component;
 4      public Decorator(Component component) {
 5          this.component = component;
 6      }
 7      public String operation() {
 8          // Forwarding to component.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

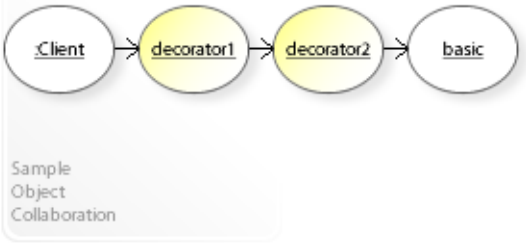
```
 1  package com.sample.decorator.basic;
 2  public class Decorator1 extends Decorator {
 3      public Decorator1(Component component) {
 4          super(component); // calling the super class constructor
 5      }
 6      public String operation() {
 7          // Forwarding to component.
 8          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 + "***";
```

```
    }
}
```

```
package com.sample.decorator.basic;
public class Decorator2 extends Decorator {
    public Decorator2(Component component) {
        super(component); // calling the super class constructor
    }
    public String operation() {
        // Forwarding to component.
        String result = super.operation();
        // Adding functionality to result from component.
        return addBehavior(result);
    }
    private String addBehavior(String result) {
        return " === " + result + " === ";
    }
}
```

Sample Code 2



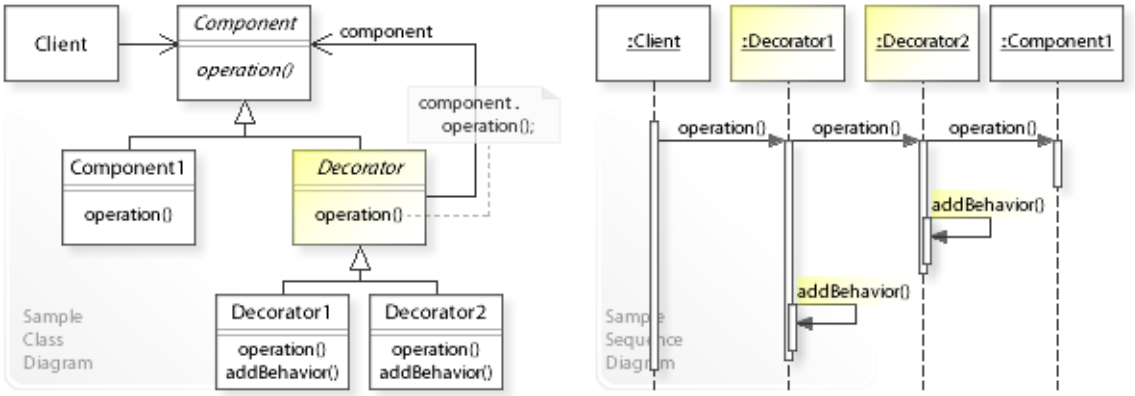


I/O Data Streams (Java Platform) / Adding functionality to basic I/O data streams.

```
 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



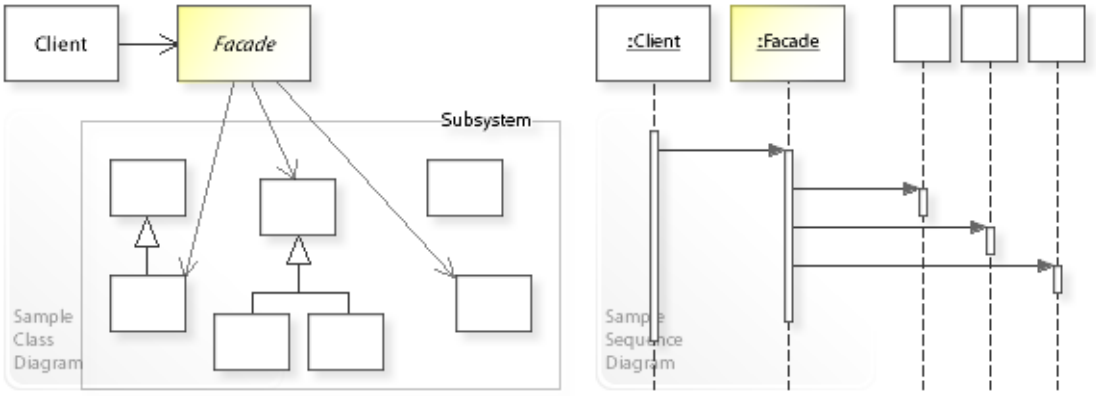
Key Relationships

- **Adapter - Bridge* - Composite - Decorator - Facade - Flyweight* - Proxy**
  These patterns are classified as *structural design patterns*. [GoF, p10]
  - Adapter provides an alternative interface for an (already existing) class or object.
  - Bridge* lets an abstraction and its implementation vary independently.
  - Composite composes (already existing) objects into a tree structure.
  - Decorator provides additional functionality for an (already existing) object.
  - Facade provides an unified interface for (already existing) objects in a subsystem.
  - Flyweight* supports large numbers of fine-grained objects efficiently.
  - Proxy provides additional functionality when accessing an (already existing) object.
- **Strategy - Decorator**
  - Strategy provides a way
    to exchange the algorithm of an object at run-time.
    This is done from *inside* the object.
    The object is designed to delegate an algorithm to a `Strategy` object.
    This is a key characteristic of *object behavioral patterns*.
  - Decorator provides a way
    to extend the functionality of an object at run-time.
    This is done from *outside* the object.
    The object already exists and isn't needed to be touched.
    This is a key characteristic of *object structural patterns*.

Background Information

- *Structural design patterns*
  (shown in the second row of the main menu) are concerned with providing alternative behavior for already existing classes or objects (without touching them).
- Bridge* and Flyweight* should be classified as *behavioral design patterns*
  (shown in the third row of the main menu) that are concerned with designing related classes and interacting objects having a desired behavior.
- "*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]

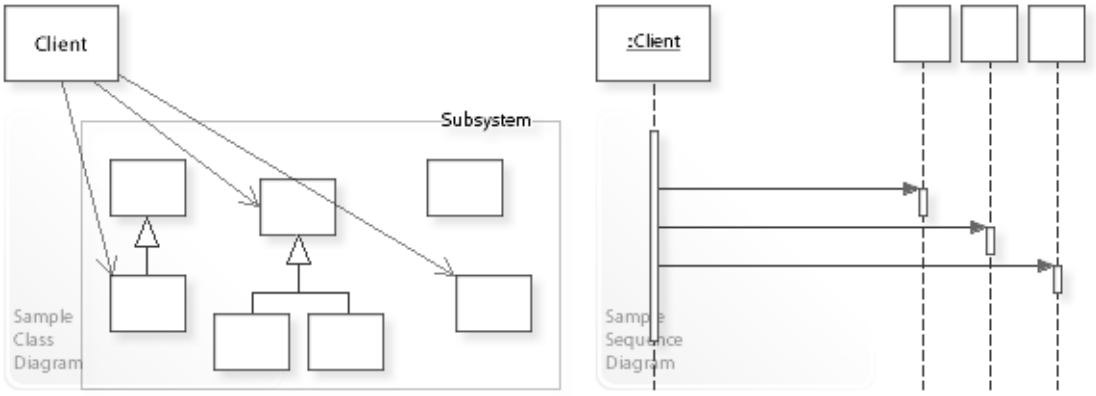
Intent



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 simple interface be provided for a complex subsystem?*
  - *How can tight coupling between clients and the objects in a subsystem be avoided?*
- A complex subsystem should provide a simplified (high-level) view that is good enough for most clients that merely need basic functionality.
- 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 different interfaces in the subsystem, which reduces dependencies and makes clients easier to implement, change, test, and reuse.

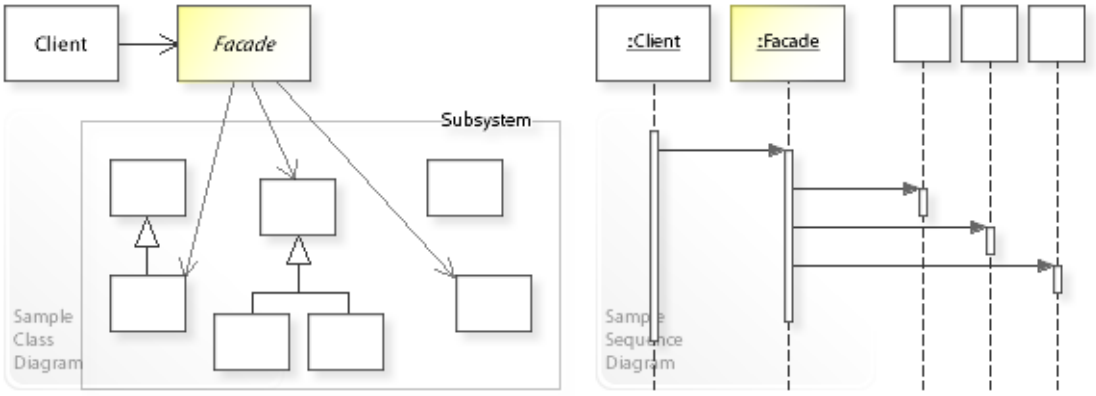
Problem



The Facade design pattern solves problems like:
How can a simple interface be provided for a complex subsystem?
How can tight coupling between clients and the objects in a subsystem be avoided?
See Applicability section for all problems Facade can solve. See Solution section for how Facade solves the problems.

- Complex software systems are often structured (layered) into subsystems.
  Clients of a complex subsystem refer to and know about (depend on) many different objects (having different interfaces), which makes the clients tightly coupled to the subsystem.
  Tightly coupled objects are hard to implement, change, test, and reuse
  because they depend on (refer to and know about) many different objects.
- *That's the kind of approach to avoid if we want to minimize the dependencies on a subsystem.*
  "A common design goal is to minimize the communication and dependencies between subsystems." [GoF, p185]
- 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 objects in the subsystem directly.

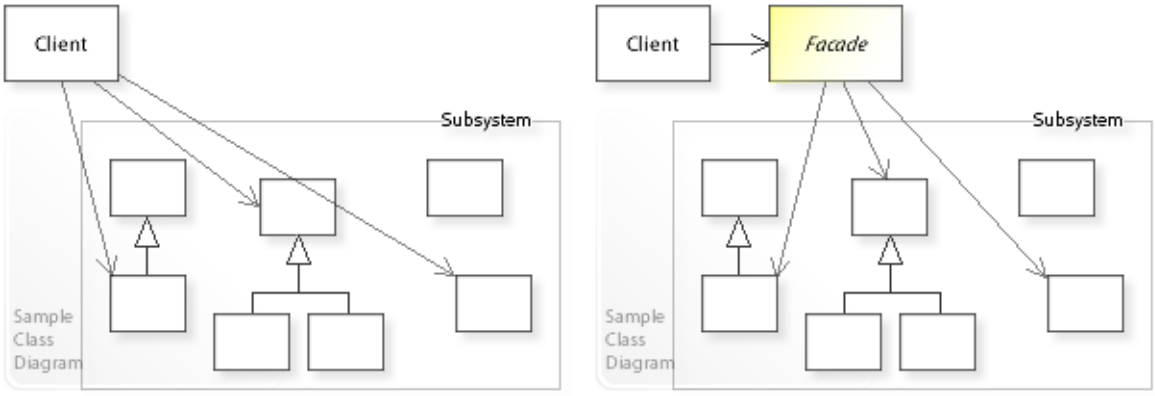
Solution



The Facade design pattern provides a solution:
Define a separate `Facade` object that provides an unified interface for a set of interfaces in a subsystem.
Work through a `Facade` to minimize dependencies on a subsystem.
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 work through a separate `Facade` object that provides a simple interface for (already existing) objects in a subsystem.
  Clients can either work with a subsystem directly or its `Facade`.
- **Define a separate `Facade` object:**
  - Define an unified interface for a set of interfaces in a subsystem (`Facade`).
  - Implement the `Facade` interface
    in terms of (by delegating to) the interfaces in the subsystem.
- Working through a `Facade` object minimizes dependencies on a subsystem (loose coupling), which makes clients easier to implement, change, test, and reuse.
  Clients that need more lower-level functionalities can access the objects in the subsystem directly.

Motivation 1



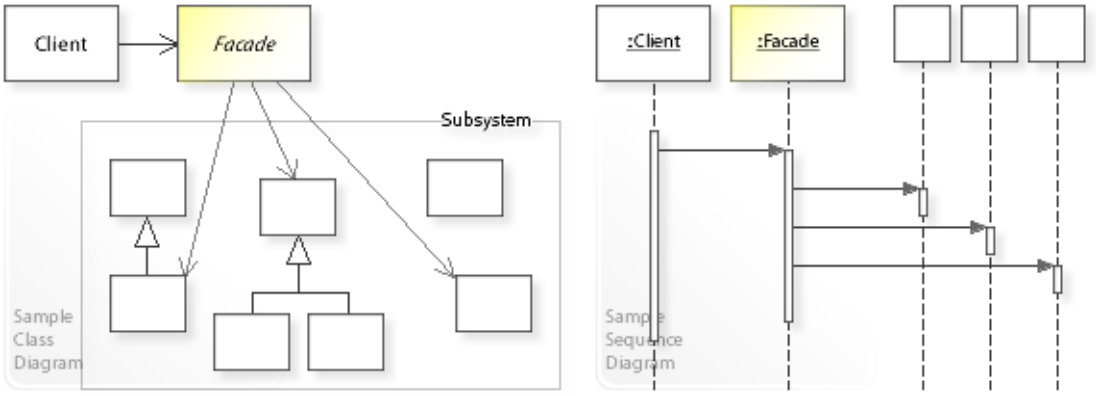
Consider the left design (problem):

- No facade / direct access.
  Tight coupling between client and subsystem.
  - Clients refer to and know about (depend on) many different interfaces in the subsystem, which makes clients harder to implement, change, test, and reuse.
  - Clients must be changed when interfaces in the subsystem are added or extended.

Consider the right design (solution):

- Working through a facade.
  Loose coupling between client and subsystem.
  - 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.

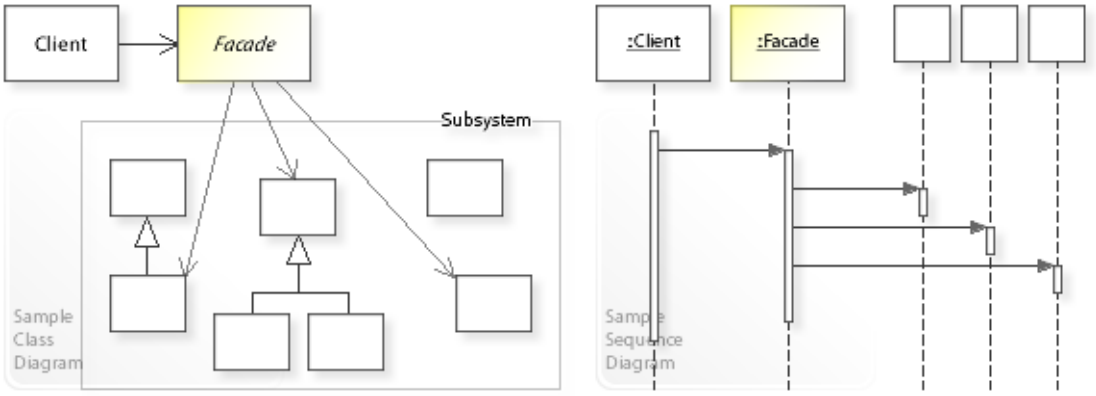
Applicability



Design Problems

- **Making Complex Subsystems Easier to Use**
  - How can a simple interface be provided for a complex subsystem?
  - How can a single entry point be provided for a subsystem?
- **Avoiding Tight Coupling Between Subsystems**
  - How can dependencies on a subsystem be minimized?
  - How can tight coupling between
    clients and the objects in a subsystem be avoided?

Structure, Collaboration



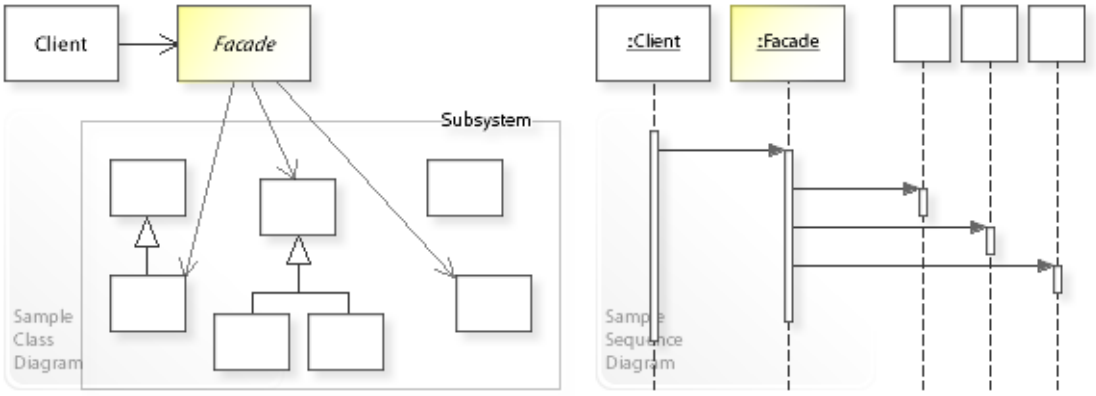
Static Class Structure

- `Client`
  - Refers to the `Facade` interface.
- *`Facade`*
  - Defines a simple interface for a complex subsystem (by referring to many different interfaces in the subsystem).

Dynamic Object Collaboration

- In this sample scenario, a `Client` object works through a `Facade` object to access many different objects in a subsystem.
- The `Client` object calls an operation on the `Facade` object.
- `Facade` delegates the request to the objects in the subsystem that fulfill the request.
- `Facade` may do work of its own before and/or after forwarding a request.
- See also Sample Code / Example 1.

Consequences

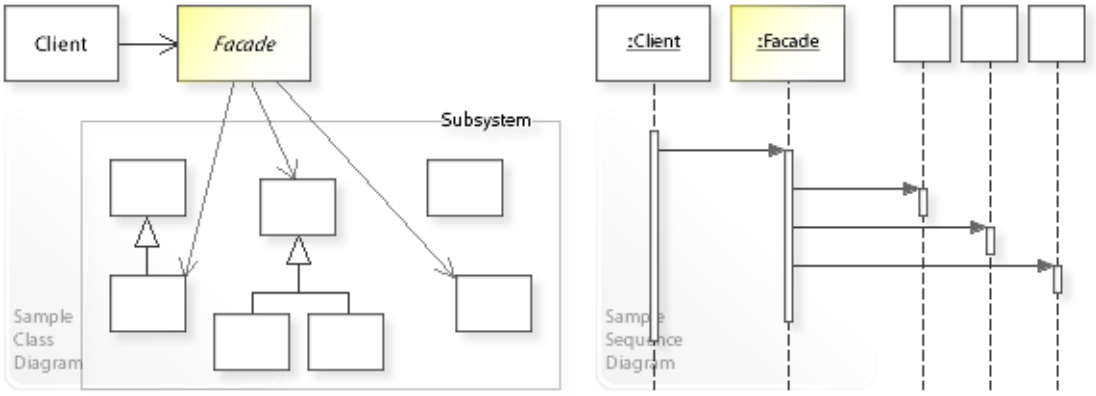


Advantages (+)

- Decouples clients from a subsystem.
  - Clients are decoupled from the subsystem by working through a `Facade` object.
  - Clients only refer to and know about the simple `Facade` interface and are independent of the complex subsystem (loose coupling).
  - This makes clients 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 (–)

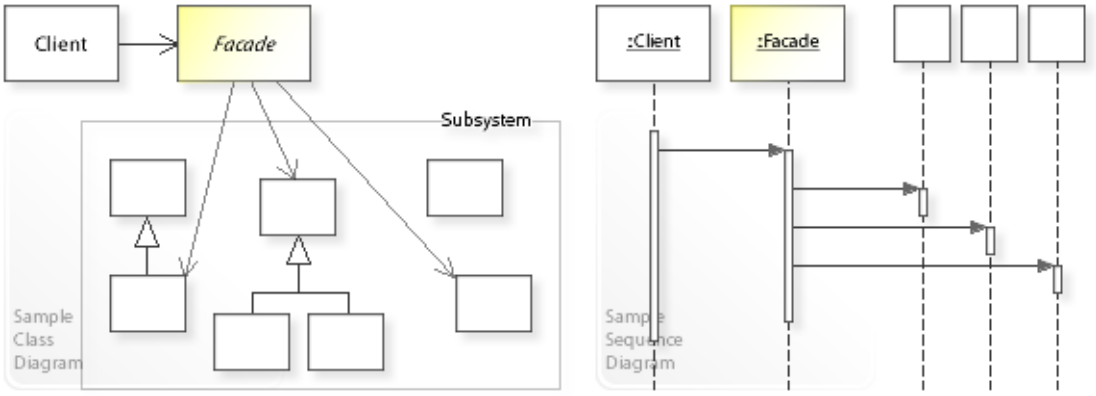
Implementation



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
 7              (new Class1(), new Class2(), new Class3());
 8          // Working through the facade.
 9          System.out.println(facade.operation());
10      }
11  }

Facade forwards to ... Class1 Class2 Class3
```

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

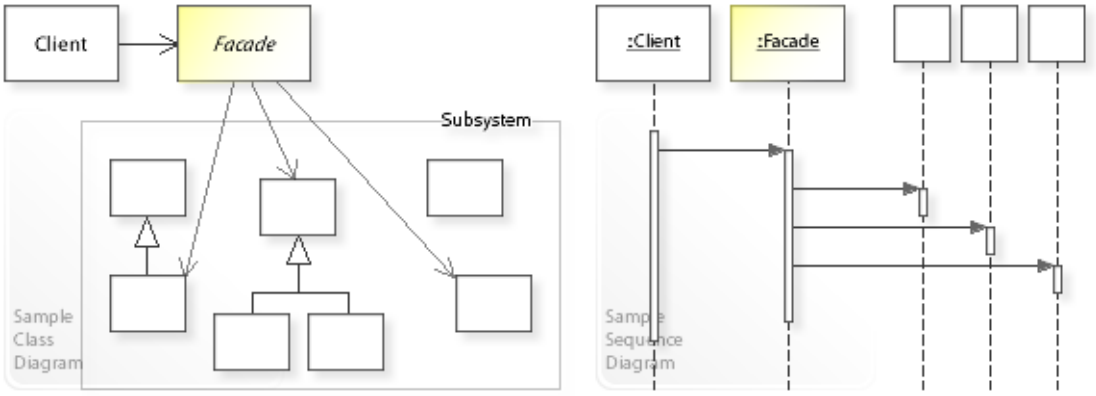
```
 1  package com.sample.facade.basic;
 2  public class Facade1 extends Facade {
 3      private Class1 object1;
 4      private Class2 object2;
 5      private Class3 object3;
 6
 7      public Facade1(Class1 object1, Class2 object2, Class3 object3) {
 8          this.object1 = object1;
 9          this.object2 = object2;
10          this.object3 = object3;
11      }
12      public String operation() {
13          return "Facade forwards to ... "
14                  + object1.operation1()
15                  + object2.operation2()
16                  + object3.operation3();
17      }
18  }
```

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

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

```
1  package com.sample.facade.basic;
2  public class Class3 {
3      public String operation3() {
4          return "Class3 ";
5      }
6  }
```

Related Patterns



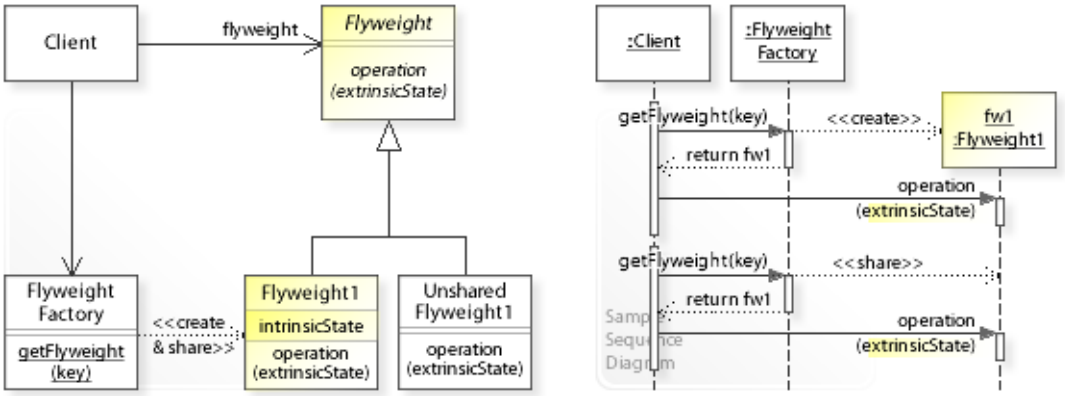
Key Relationships

- **Adapter - Bridge* - Composite - Decorator - Facade - Flyweight* - Proxy**
  These patterns are classified as *structural design patterns*. [GoF, p10]
  - Adapter provides an alternative interface for an (already existing) class or object.
  - Bridge* lets an abstraction and its implementation vary independently.
  - Composite composes (already existing) objects into a tree structure.
  - Decorator provides additional functionality for an (already existing) object.
  - Facade provides an unified interface for (already existing) objects in a subsystem.
  - Flyweight* supports large numbers of fine-grained objects efficiently.
  - Proxy provides additional functionality when accessing an (already existing) object.

Background Information

- *Structural design patterns*
  (shown in the second row of the main menu) are concerned with providing alternative behavior for already existing classes or objects (without touching them).
- Bridge* and Flyweight* should be classified as *behavioral design patterns*
  (shown in the third row of the main menu) that are concerned with designing related classes and interacting objects having a desired behavior.

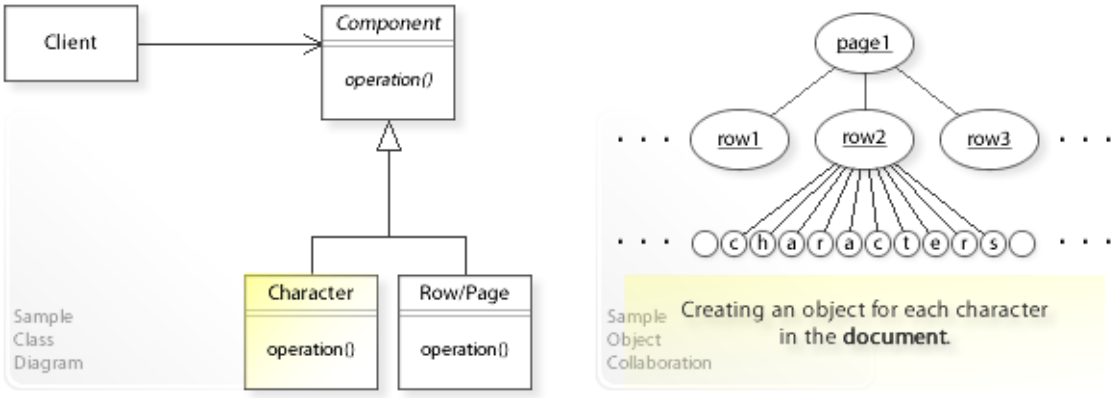
Intent



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 every character in the document, which may result in a huge amount of objects.
- 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 (invariant) state.
    Clients share `Flyweight` objects and pass in extrinsic (variant) state dynamically at run-time when they invoke a flyweight operation (`flyweight.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 and must be passed in.

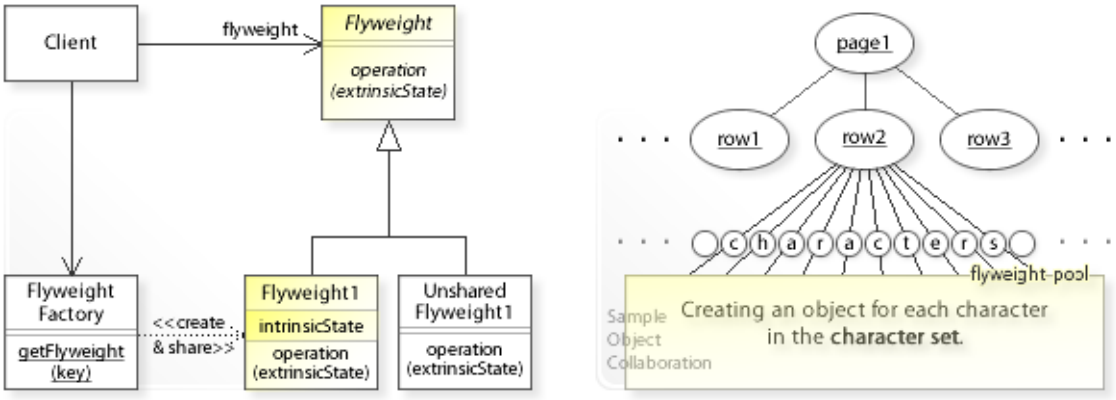
Problem



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. See Solution section for how Flyweight solves the problems.

- 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 needed for every occurrence of a character in the document, which can result in a huge amount of objects. "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]
- *That's the kind of approach to avoid if we want to support large numbers of objects efficiently.*
- 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.

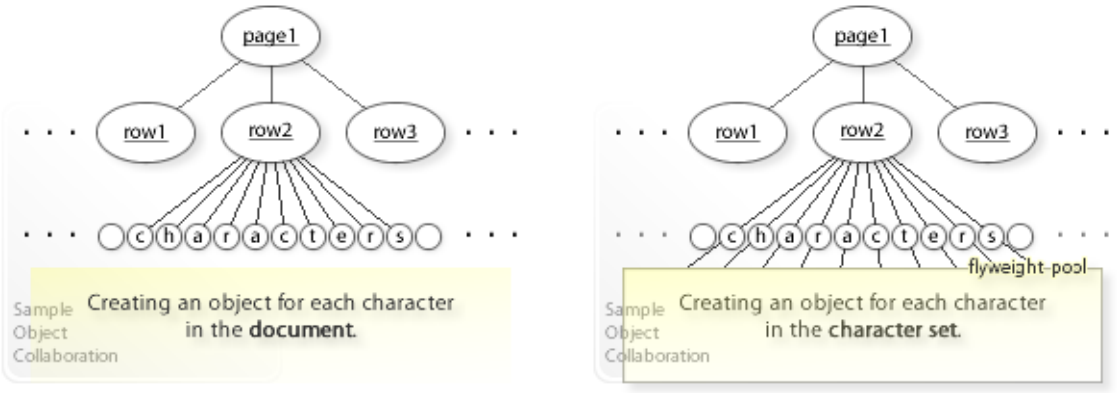
Solution



The Flyweight design pattern provides a solution:
Define separate `Flyweight` objects that store intrinsic (invariant) state.
Clients share `Flyweight` objects and pass in extrinsic (variant) state
instead of creating an object each time it is needed.
Describing the Flyweight design in more detail is the theme of the following sections.
See Applicability section for all problems Flyweight can solve.

- "The key concept here is the distinction between **intrinsic** and **extrinsic** state." [GoF, p196]
  Flyweight objects store intrinsic state, and clients pass in extrinsic state.
  Intrinsic state is invariant (context independent) and therefore can be shared.
  For example, the code of a character in the used character set.
  Extrinsic state is variant (context dependent) and therefore can *not* be shared.
  For example, the position of a character in the document.
- **Define separate `Flyweight` objects:**
  - Define an interface (`Flyweight | operation(extrinsicState)`) through which extrinsic (variant) state can be passed in.
  - Define classes (`Flyweight1`,…) that implement the `Flyweight` interface and store intrinsic (invariant) state that can be shared.
- **Clients share (reuse) `Flyweight` objects and pass in extrinsic state each time they invoke a flyweight operation** (`flyweight.operation(extrinsicState)`).
  - 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.
  - This greatly reduces the number of physically created objects.

Motivation 1



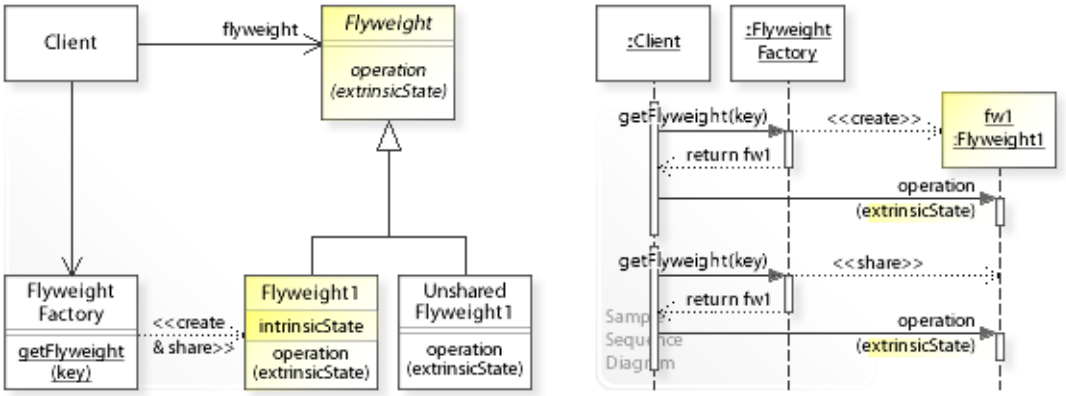
Consider the left design (problem):

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

Consider the right design (solution):

- Small number of physically created objects.
  - To represent a text document, a `Flyweight` object is created for each character in the used character set (`flyweight pool`).
  - The number of physically created character objects is independent of the number of characters in the document.
    It depends on the number of characters in the character set.
  - 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).

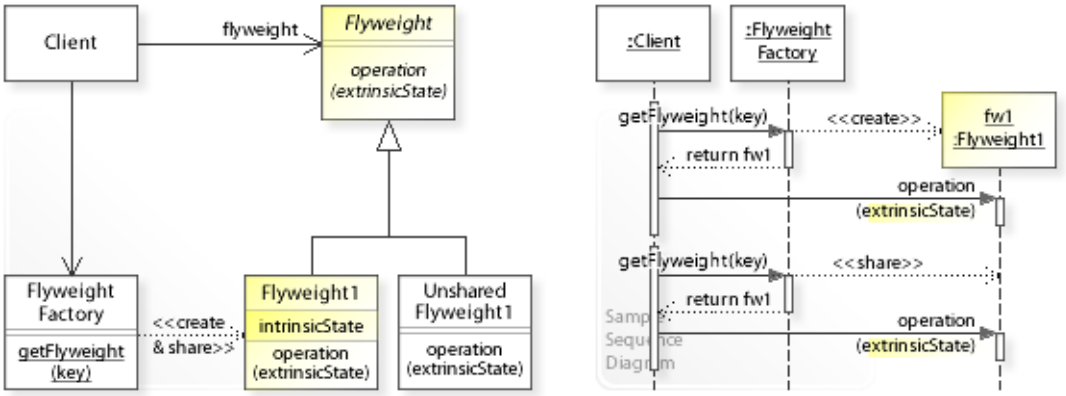
Applicability



Design Problems

- **Supporting Large Numbers of Objects**
  - How can large numbers of fine-grained objects be supported efficiently?
  - How can objects be shared to avoid creating large numbers of objects?
  - How can small numbers of physically created objects represent large numbers of logically different objects?

Structure, Collaboration



Static Class Structure

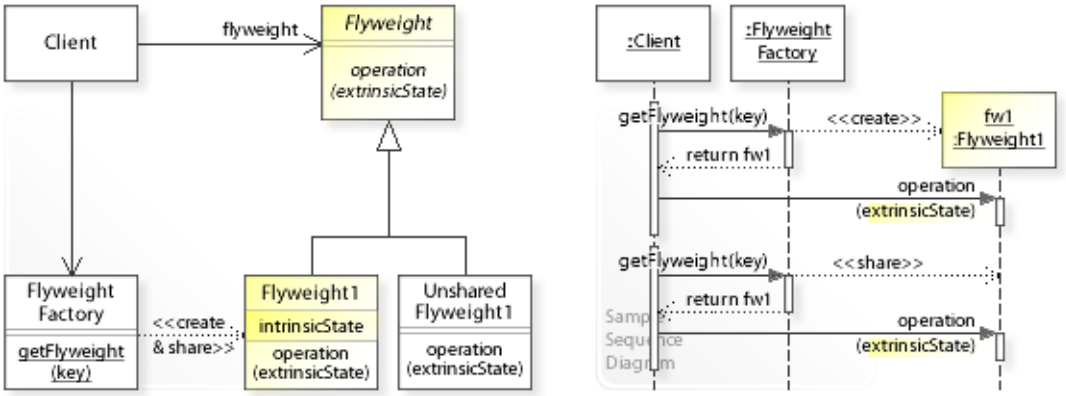
- *`Client`*
  - Refers to the `Flyweight` interface.
  - Maintains a reference (`flyweight`) to a `Flyweight` object.
  - Passes in extrinsic state when invoking a `Flyweight` operation (`operation(extrinsicState)`).
  - Requests a `Flyweight` object from the `FlyweightFactory` (by invoking `getFlyweight(key)`).
- *`Flyweight`*
  - Defines an interface (`operation(extrinsicState)`) through which extrinsic (variant) state can be passed in.
- `Flyweight1,...`
  - Implement the `Flyweight` interface.
  - Store intrinsic (invariant) state that can be shared.
- `UnsharedFlyweight1,...`
  - Implement the `Flyweight` interface but are not shared.
- `FlyweightFactory`
  - Maintains a container of shared `Flyweight` objects (flyweight pool).
  - Creates a `Flyweight` object if it doesn't exist and shares (reuses) an existing one.

Dynamic Object Collaboration

- In this sample scenario, a `Client` object shares a `Flyweight1` object by requesting it from a `FlyweightFactory` object.
- The interaction starts with the `Client` that calls `getFlyweight(key)` on the `FlyweightFactory`.
- Because the flyweight does not already exist, the `FlyweightFactory` **creates** a `Flyweight1` object and returns it to the `Client`.
- The `Client` calls `operation(extrinsicState)` on the returned `Flyweight1` object by passing in the extrinsic state.
- Thereafter, the `Client` again calls `getFlyweight(key)` on the `FlyweightFactory`.
- Because the flyweight already exists, the `FlyweightFactory` **shares** (reuses) the `Flyweight1` object.

- The `Client` calls `operation(extrinsicState)` on the returned `Flyweight1` object.
- See also Sample Code / Example 1.

Consequences



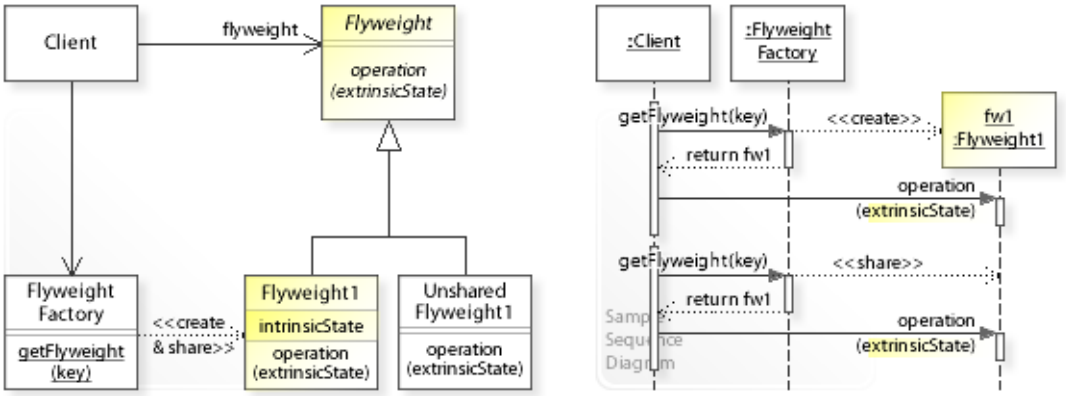
Advantages (+)

- Enables abstractions at the finest levels.
  - A small number of physically created objects can represent an open-ended number of logically different objects.

Disadvantages (–)

- Introduces run-time costs.
  - Clients are responsible for passing in extrinsic state dynamically at run-time.
  - Storing/retrieving/calculating extrinsic state each time a flyweight operation is performed can impact memory usage and system performance.
- Provides no reliability on object identity.
  - The same physically created object represents many logically different objects.
  - Therefore, applications that depend on object identity should not use flyweights.

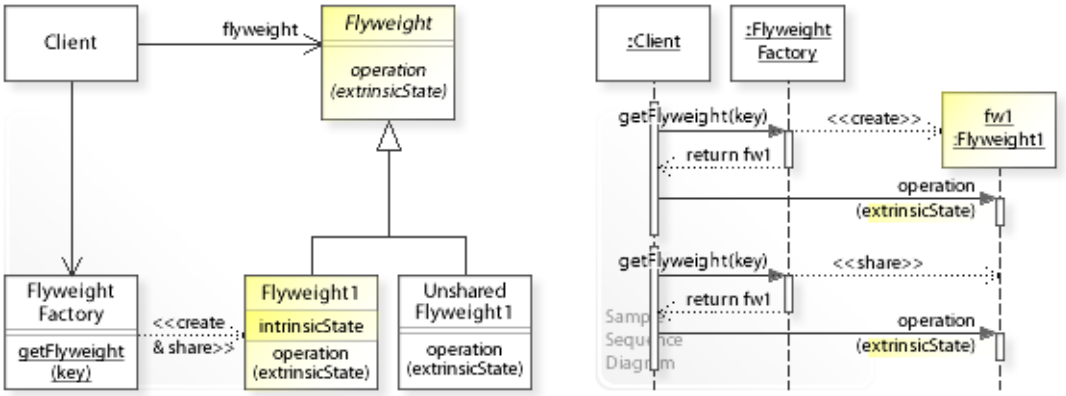
Implementation



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 shared flyweights.
    If the requested flyweight already exists in the pool, it is shared (reused) and returned to the client.
    Otherwise, it is created, added to the pool, and returned.
  - The `key` parameter in the `getFlyweight(key)` operation is needed to look up the right flyweight in the pool (see Sample Code / Example 1).

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          System.out.println("\n*** Number of flyweights created: "
16                  + flyweightFactory.getSize() + " ***");
17      }
18  }
```

```
C r e a t i n g  a flyweight with key = A
  performing an operation on the flyweight
  with intrinsic state = A and passed in extrinsic state = 100.
S h a r i n g    a flyweight with key = A
  performing an operation on the flyweight
  with intrinsic state = A and passed in extrinsic state = 200.

*** Number of flyweights created: 1 ***
```

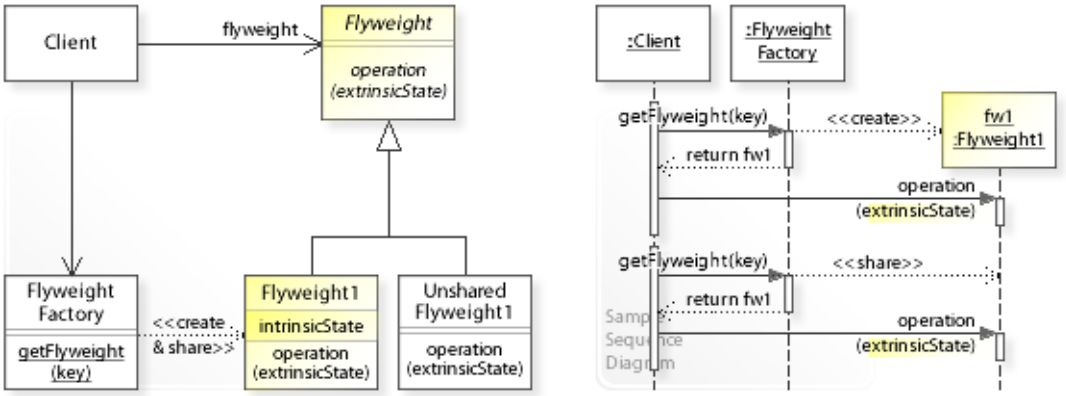
```
 1  package com.sample.flyweight.basic;
 2  public interface Flyweight {
 3      public String operation(int extrinsicState);
 4  }
```

```
 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 on the flyweight\n "
 9                  + " with intrinsic state = " + intrinsicState
10                  + " and passed in extrinsic state = " + extrinsicState + ".";
11      }
12  }
```

```
 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("S h a r i n g    a flyweight with key = " + key);
18              return flyweights.get(key);
19          }  else {
20              System.out.println("C r e a t i n g  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



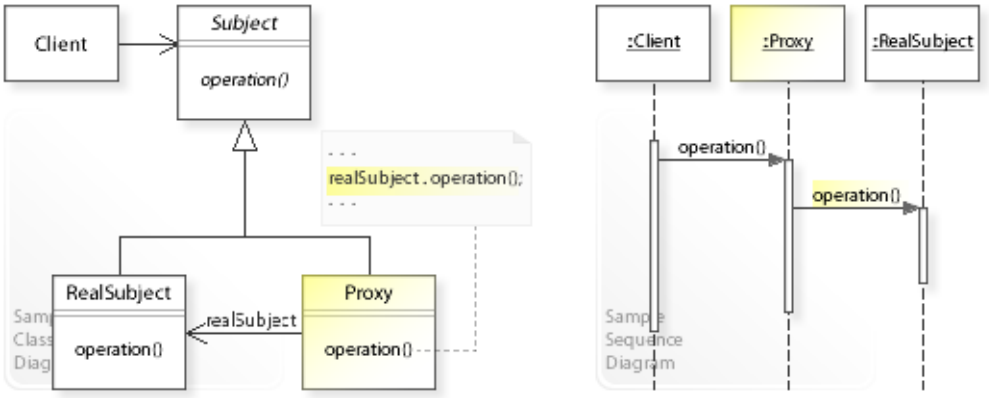
Key Relationships

- **Adapter - Bridge* - Composite - Decorator - Facade - Flyweight* - Proxy**
  These patterns are classified as *structural design patterns*. [GoF, p10]
  - Adapter provides an alternative interface for an (already existing) class or object.
  - Bridge* lets an abstraction and its implementation vary independently.
  - Composite composes (already existing) objects into a tree structure.
  - Decorator provides additional functionality for an (already existing) object.
  - Facade provides an unified interface for (already existing) objects in a subsystem.
  - Flyweight* supports large numbers of fine-grained objects efficiently.
  - Proxy provides additional functionality when accessing an (already existing) object.
- **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.

Background Information

- *Structural design patterns*
  (shown in the second row of the main menu) are concerned with providing alternative behavior for already existing classes or objects (without touching them).
- Bridge* and Flyweight* should be classified as *behavioral design patterns*
  (shown in the third row of the main menu) that are concerned with designing related classes and interacting objects having a desired behavior.

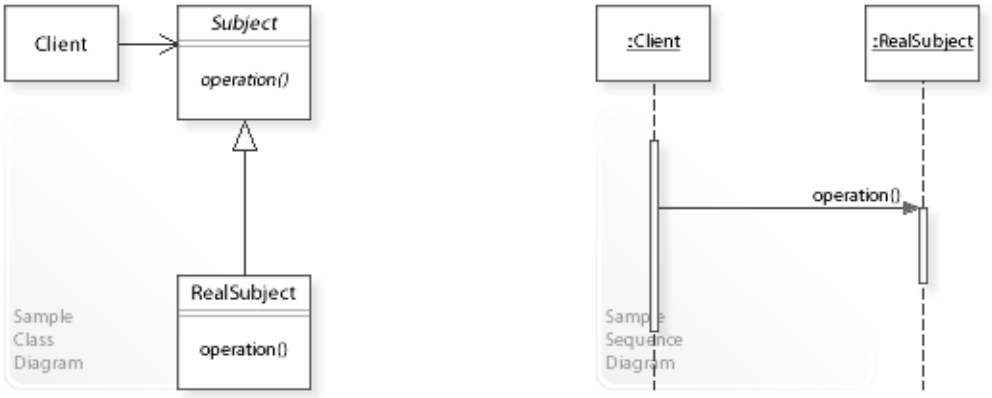
Intent



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 functionality be provided when accessing an object?*
- For example, the access to *sensitive*, *expensive*, or *remote* objects should be controlled.
- 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 acts as placeholder for another object (`Subject`). A proxy implements the `Subject` interface so that it can act as placeholder anywhere a subject is expected.
  - Work through a `Proxy` object to control the access to an (already existing) object.

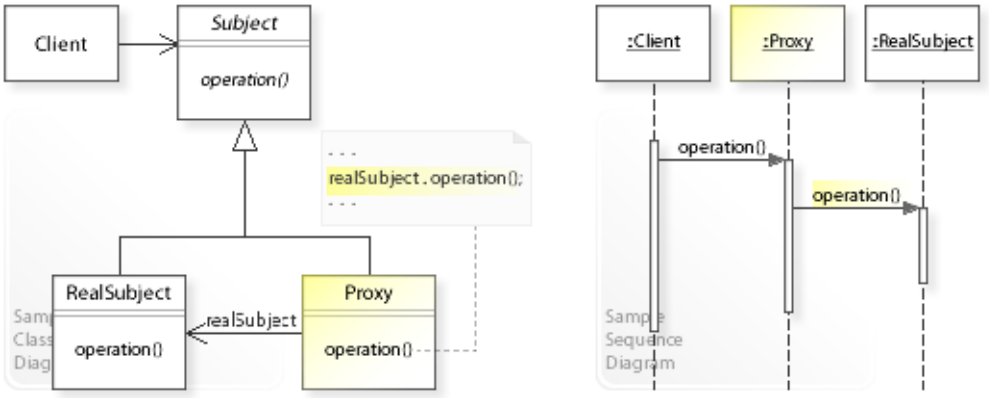
Problem



The Proxy design pattern solves problems like:
How can the access to an object be controlled?
How can additional functionality be provided when accessing an object?
See Applicability section for all problems Proxy can solve. See Solution section for how Proxy solves the problems.

- Often it should be possible to provide additional functionality when accessing an (already existing) object.
  "[…] whenever there is a need for a more versatile or sophisticated reference to an object than a simple pointer." [GoF, p208]
- 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 cache their data.
- For example, when accessing *remote* objects, it should be possible to hide complex network communication details from clients.

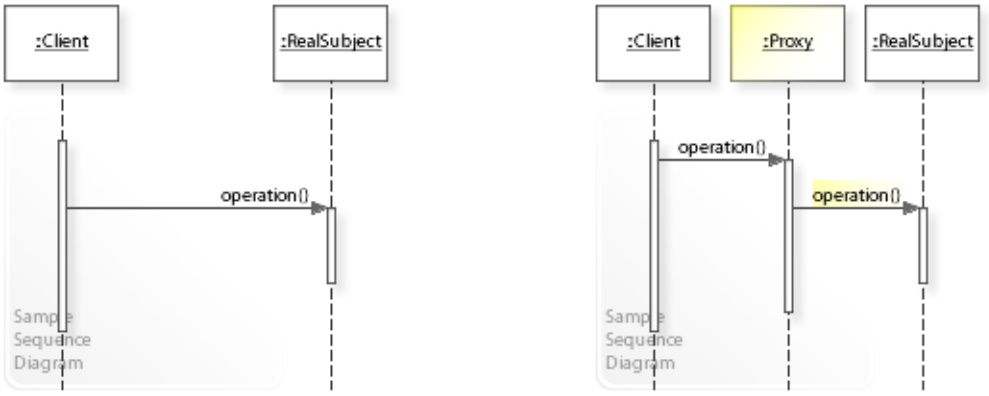
Solution



The Proxy design pattern provides a solution:
Define a separate `Proxy` object that acts as substitute for another object (`Subject`).
Work through a `Proxy` object to control the access to a real subject.
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 work through a separate `Proxy` object that performs additional functionality when accessing an (already existing) object.
  A proxy implements the `Subject` interface so that it can act as substitute wherever a subject is expected. Clients do not know whether they are working with a real subject or its proxy.
- **Define a separate `Proxy` object:**
  - Define a class (`Proxy`) that implements arbitrary functionality to control the access to a `RealSubject` object.
  - "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* acts as placeholder for *sensitive* objects
    to check that clients have the required access rights.
    A *virtual proxy* acts as placeholder for *expensive* objects
    to defer their creation until they are actually needed.
    A *remote proxy* acts as placeholder for *remote* objects
    to hide complex network communication details from clients.

Motivation 1



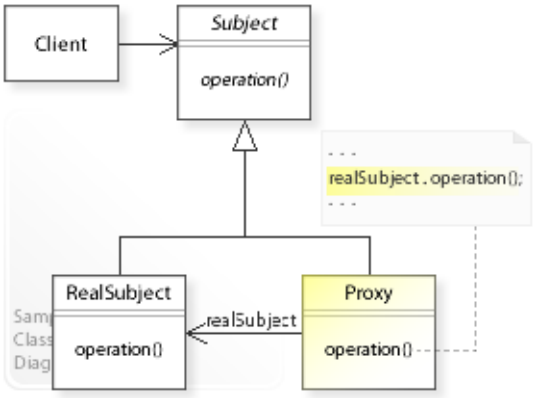
Consider the left design (problem):

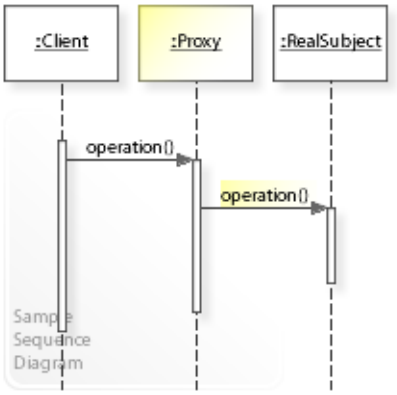
- No proxy / direct access.
  Complicated clients.
  - Clients access `RealSubject` directly.
  - Handling remote or expensive objects, for example, makes clients harder to implement, change, test, and reuse.

Consider the right design (solution):

- Working through a proxy.
  Simplified clients.
  - Clients work through a `Proxy`.
  - `Proxy` hides implementation details from clients, which makes them easier to implement, change, test, and reuse.

Applicability



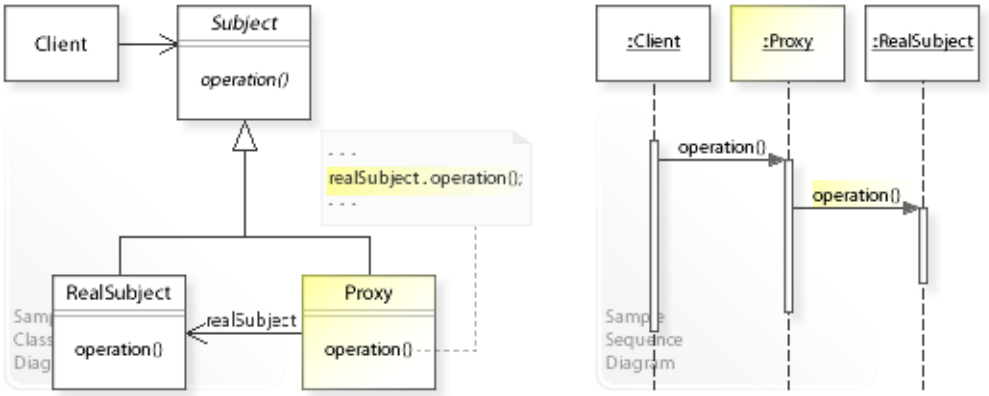


Design Problems

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

- **Common Kinds of Proxies**
  - A *protection proxy* acts as placeholder for *sensitive* objects to check that clients have the required access rights.
  - A *virtual proxy* acts as placeholder for *expensive* objects to defer their creation until they are actually needed.
  - A *remote proxy* acts as placeholder for *remote* objects to hide complex network communication details from clients.

Structure, Collaboration



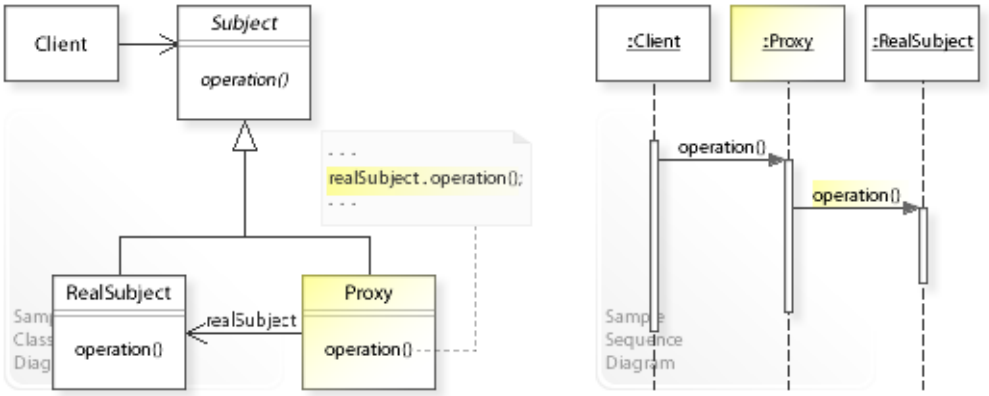
Static Class Structure

- `Client`
  - Refers to the `Subject` interface.
- *`Subject`*
  - Defines a common interface for `RealSubject` and `Proxy` objects.
- `RealSubject`
  - Defines objects that get substituted.
- `Proxy`
  - Maintains a reference to a `RealSubject` object (`realSubject`).
  - Implements additional functionality to control the access to this `RealSubject` object.
  - Implements the `Subject` interface so that it can act as substitute whenever a `Subject` object is expected.

Dynamic Object Collaboration

- In this sample scenario, a `Client` object works through a `Proxy` object that controls the access to a `RealSubject` object.
- The `Client` object calls `operation()` on the `Proxy` object.
- The `Proxy` may perform additional functionality and forwards the request to the `RealSubject` object, which performs the request.
- See also Sample Code / Example 1.

Consequences



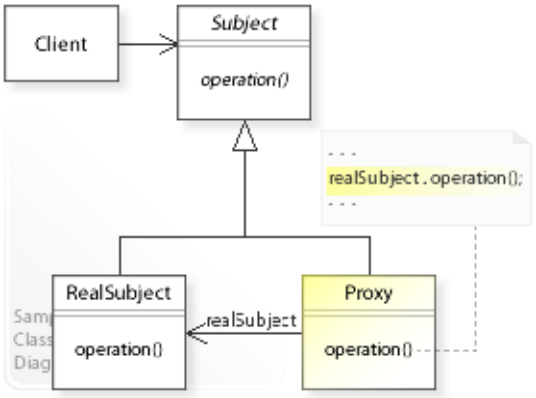
Advantages (+)

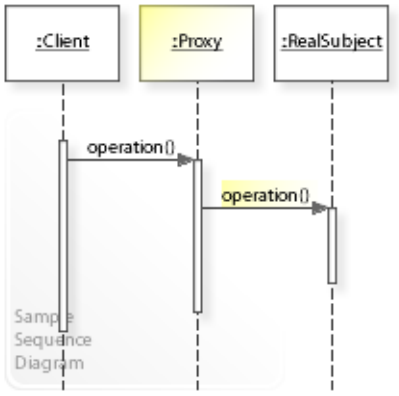
- Simplifies clients.
  - A proxy hides implementation details from clients, which makes them easier to implement, change, test, and reuse.

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 RealSubjects (such as in a virtual proxy), then they have to know the concrete class." [GoF, p213]

Implementation

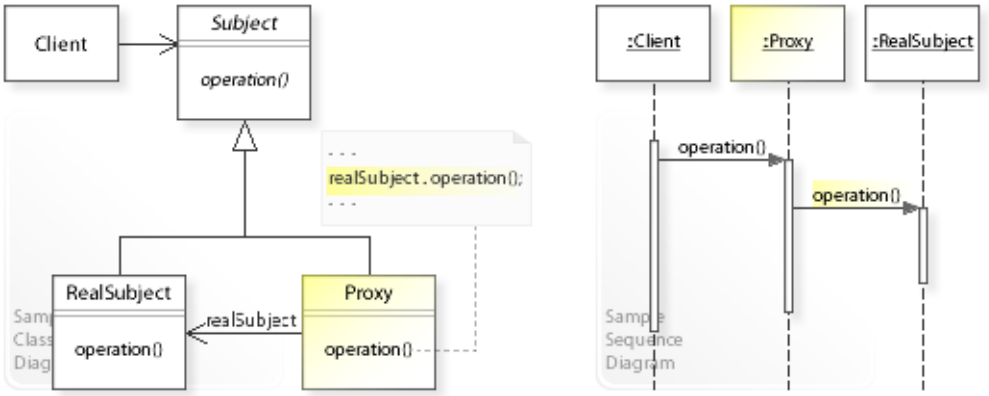




Implementation Issues

- **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 real subject directly or through its 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          // Working through the proxy.
 8          System.out.println(proxy.operation());
 9      }
10  }
```

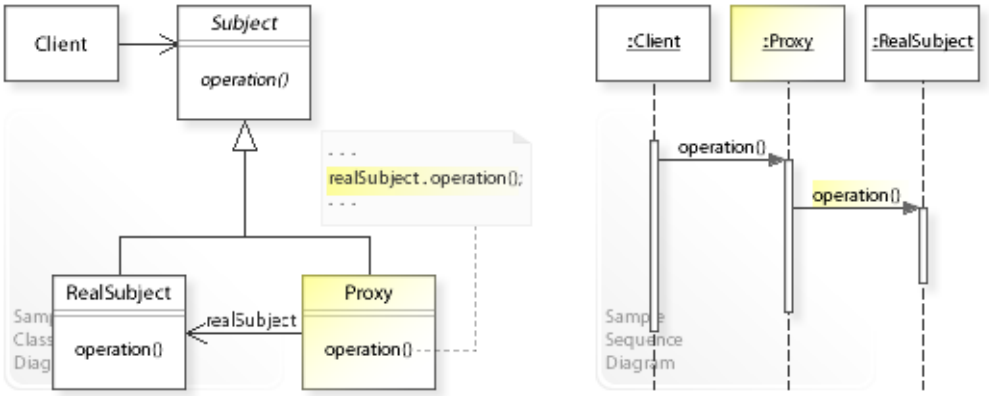
```
Hello world from Proxy and RealSubject!
```

```
 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!";
 5      }
 6  }
```

```
 1  package com.sample.proxy.basic;
 2  public class Proxy extends Subject {
 3      private RealSubject realSubject;
 4
 5      public Proxy(RealSubject subject) {
 6          this.realSubject = subject;
 7      }
 8      public String operation() {
 9          return "Hello world from Proxy and " + realSubject.operation();
10      }
11  }
```

Related Patterns



Key Relationships

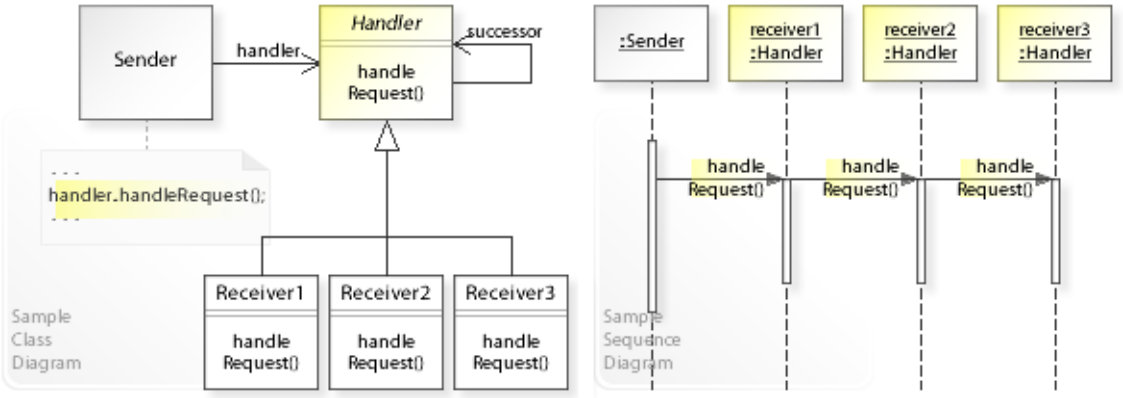
- **Adapter - Bridge* - Composite - Decorator - Facade - Flyweight* - Proxy**
  These patterns are classified as *structural design patterns*. [GoF, p10]
  - Adapter provides an alternative interface for an (already existing) class or object.
  - Bridge* lets an abstraction and its implementation vary independently.
  - Composite composes (already existing) objects into a tree structure.
  - Decorator provides additional functionality for an (already existing) object.
  - Facade provides an unified interface for (already existing) objects in a subsystem.
  - Flyweight* supports large numbers of fine-grained objects efficiently.
  - Proxy provides additional functionality when accessing an (already existing) object.

Background Information

- *Structural design patterns*
  (shown in the second row of the main menu) are concerned with providing alternative behavior for already existing classes or objects (without touching them).
- Bridge* and Flyweight* should be classified as *behavioral design patterns*
  (shown in the third row of the main menu) that are concerned with designing related classes and interacting objects having a desired behavior.

Part IV. Behavioral Patterns

Intent



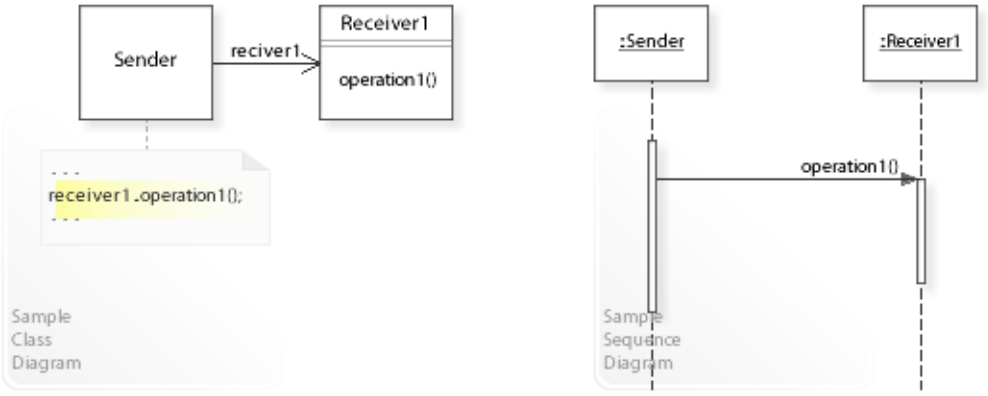
The intent of the Chain of Responsibility design pattern is to:
"Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it." [GoF]
See Problem and Solution 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 more than one object handle a request?*
- A *request* is an operation that one object (sender) performs on another (receiver).
- An inflexible way is to implement a request directly within the class that sends the request. This couples the sender of a request to a particular receiver at compile-time.
- 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 or forward a request. 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]
- Requests in *UML sequence diagrams*:
  A 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.

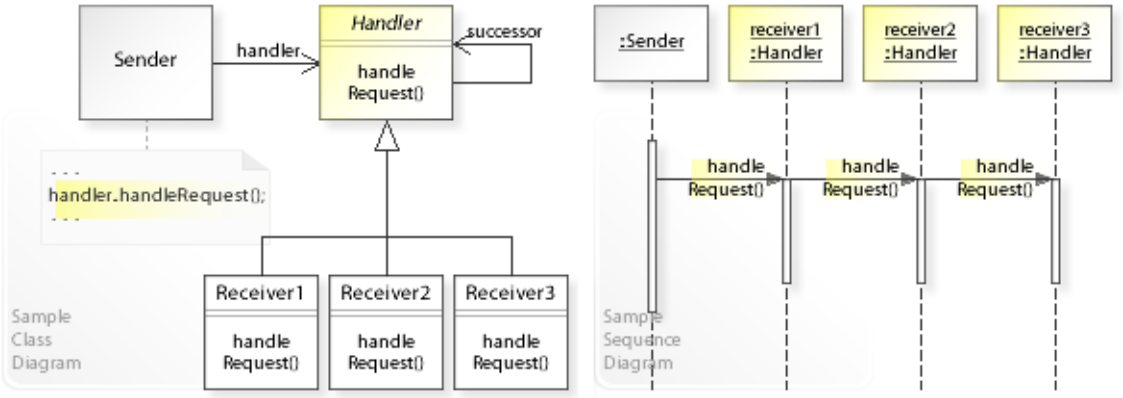
Problem



The Chain of Responsibility design pattern solves problems like:
How can coupling the sender of a request to its receiver be avoided?
How can more than one object handle a request?
See Applicability section for all problems Chain of Responsibility can solve. See Solution section for how Chain of Responsibility solves the problems.

- An inflexible way is to implement (hard-wire) a request (`receiver1.operation1()`) directly within the class (`Sender`) that sends the request.
- This commits (couples) the sender of a request to a particular receiver at compile-time and makes it impossible to specify more than one receiver.
- *That's the kind of approach to avoid if we want to specify multiple objects that can handle a request.*
- For example, providing context-sensitive help in a GUI/Web application.
  In a context-sensitive help system, a user can click anywhere to get help information. That is, multiple objects exist that can handle a help request by providing a specific help information. Which object provides the help isn't known at compile-time and should be determined at run-time (depending on run-time conditions).
  "The problem here is that the object that ultimately *provides* the help isn't known explicitly to the object [sender] (e.g., the button) that *initiates* the help request." [GoF, p223]

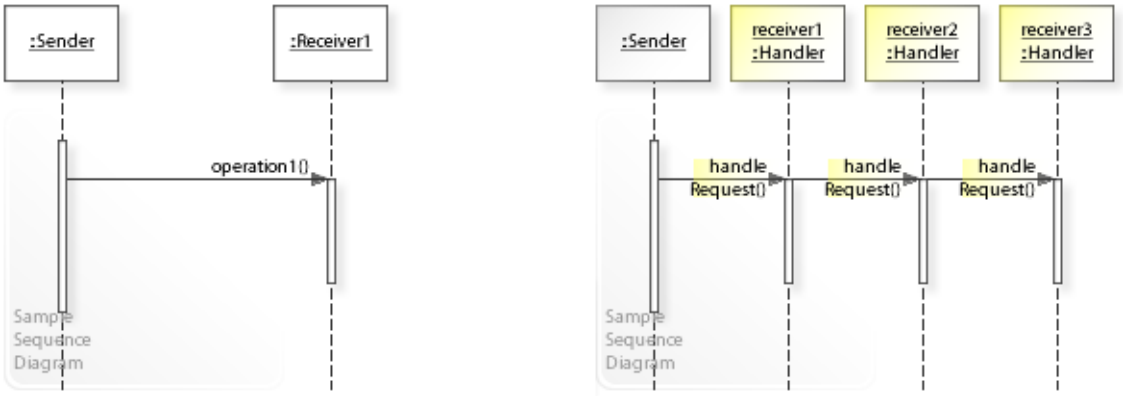
Solution



The Chain of Responsibility design pattern provides a solution:
Define a chain of `Handler` objects having the *responsibility* to either handle a request or forward it to the next handler.
A class sends a request to a chain of handlers
and doesn't know (is independent of) which handler handles the request.
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]
- **A class (`Sender`) sends a request to a chain of handlers** (`handler.handleRequest()`).
  The request gets passed along the chain until a handler handles it.
- This enables loose coupling between the sender of a request and its receiver(s).
  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



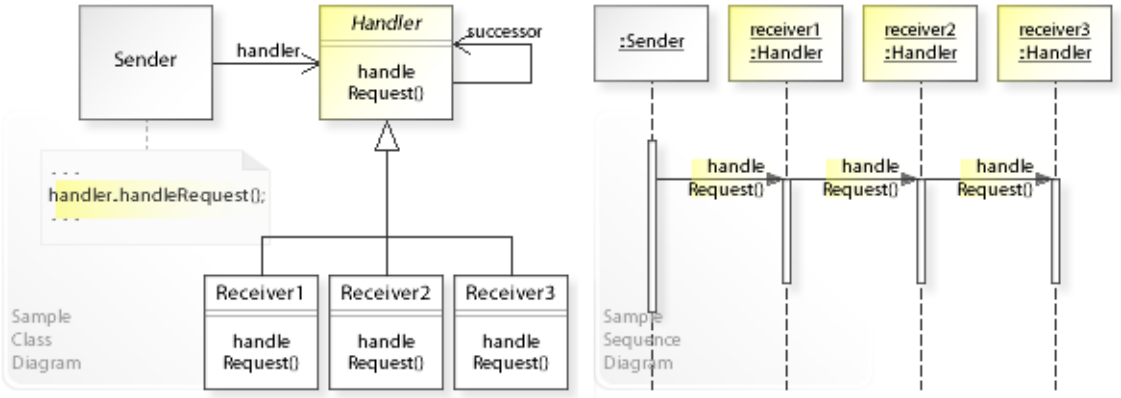
Consider the left design (problem):

- One receiver.
  - The request is sent to a particular receiver (`Receiver1` object).
  - This couples the sender to a particular receiver.

Consider the right design (solution):

- Multiple receivers.
  - The request gets passed along a chain of receivers (`Handler` objects).
  - This decouples the sender from a particular receiver.
  - The sender has no explicit knowledge of the handler (receiver) that ultimately will handle the request.
  - The chain of handlers can be changed at run-time (handlers can be added to and removed from the chain).

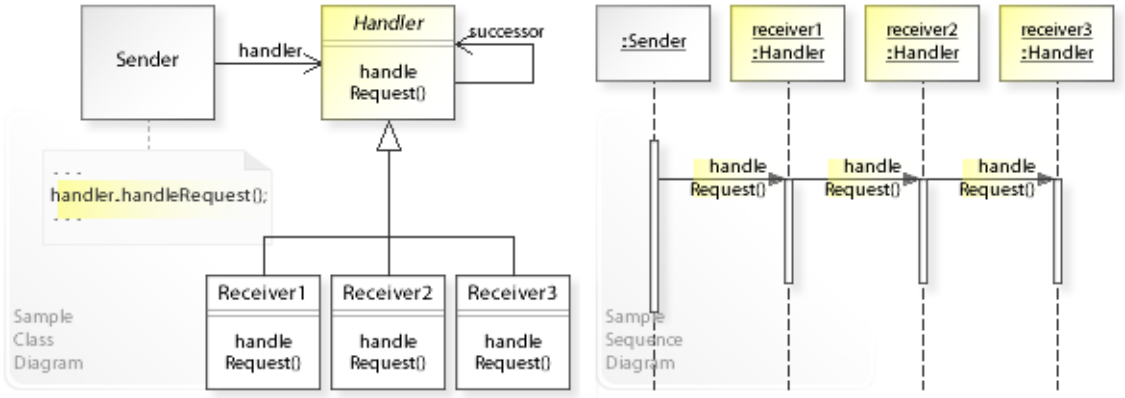
Applicability



Design Problems

- **Avoiding Hard-Wired Requests**
  - How can coupling the sender of a request to its receiver be avoided?
- **Specifying Multiple Receivers**
  - How can more than one object handle a request?
  - How can the set of objects that can handle a request be specified dynamically?

Structure, Collaboration



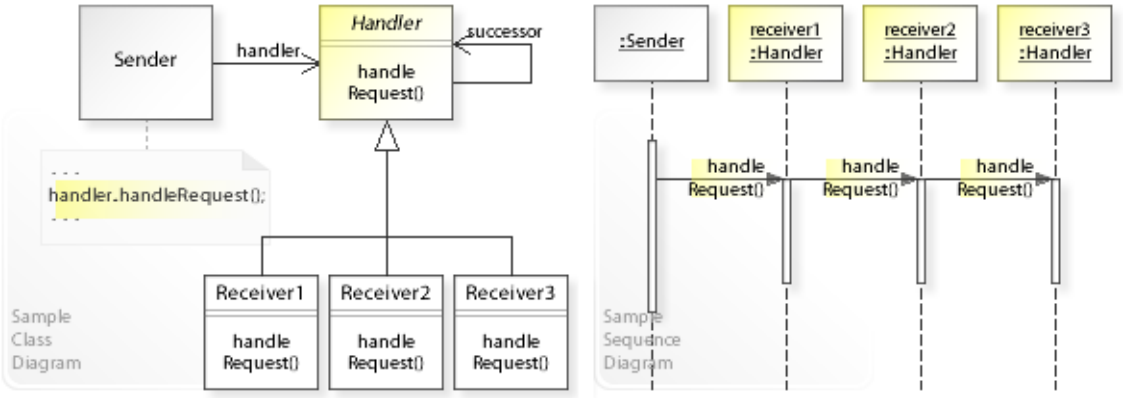
Static Class Structure

- `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 on the chain.
- *`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 `Sender` object sends a request to a `Handler` object on the chain. The request gets forwarded along the chain until a handler (`receiver3`) handles it.
- The interaction starts with the `Sender` 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 `Sender`).
- See also Sample Code / Example 1.

Consequences



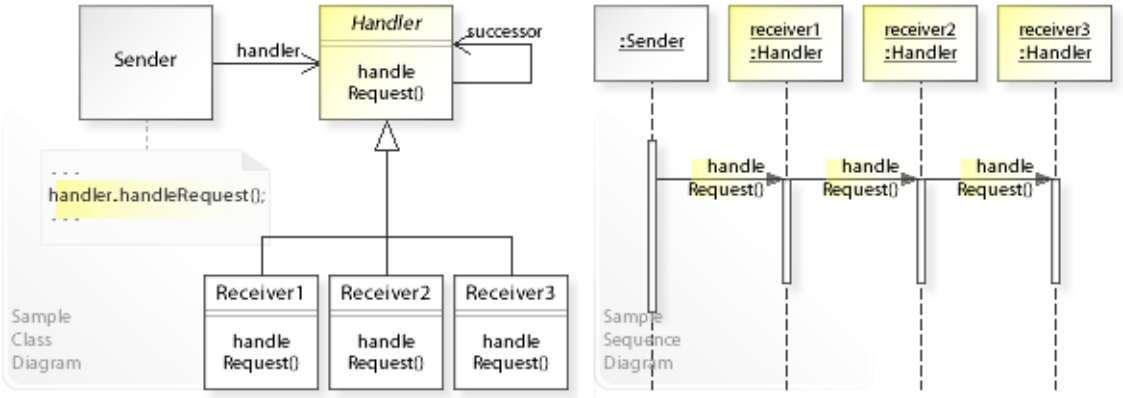
Advantages (+)

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

Disadvantages (–)

- Successor chain can be complex.
  - If there is no existing object structure that can be used to define and maintain a successor chain, it may be hard to implement and maintain the 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]

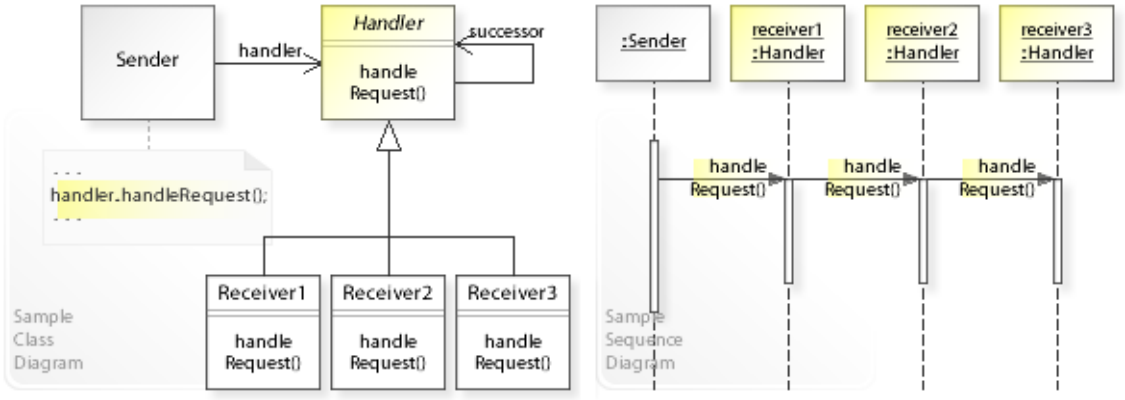
Implementation



Implementation Issues

- **Implementing the successor chain.**
  - There are two main variants to implement the successor chain:
- Variant1: **Using existing links.**
  - Often existing object structures can be used to define the chain (see Composite).
  - "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]
- Variant2: **Defining new links.**
  - If no proper object structures exist, a new chain must be defined.

Sample Code 1



Basic Java code for implementing the sample UML diagrams.

```
 1  package com.sample.cor.basic;
 2  public class Sender {
 3      // Running the Sender 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("Issuing a request to a handler object ... ");
 9                  handler.handleRequest();
10      }
11  }
```

```
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.
```

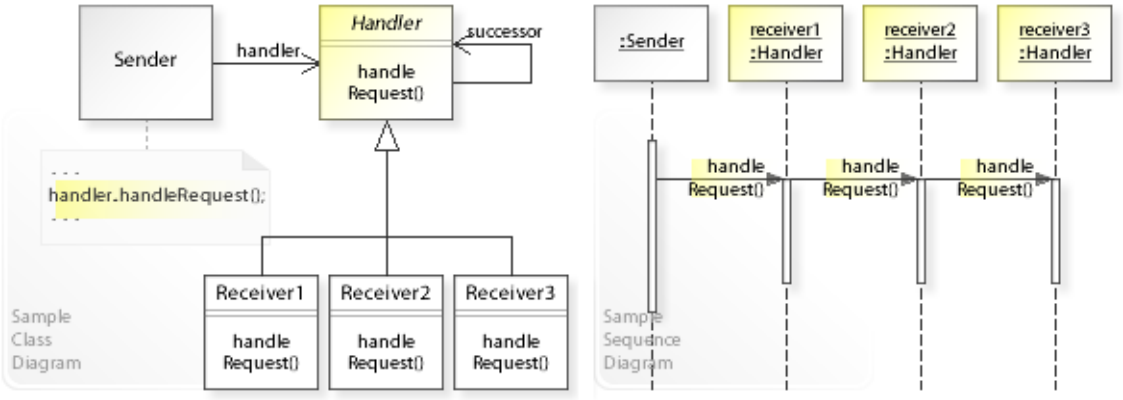
```
 1  package com.sample.cor.basic;
 2  public abstract class Handler {
 3      private Handler successor;
 4      public Handler() { }
 5      public Handler(Handler successor) {
 6          this.successor = successor;
 7      }
 8      public void handleRequest() {
 9          // Forwarding to successor (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  }
```

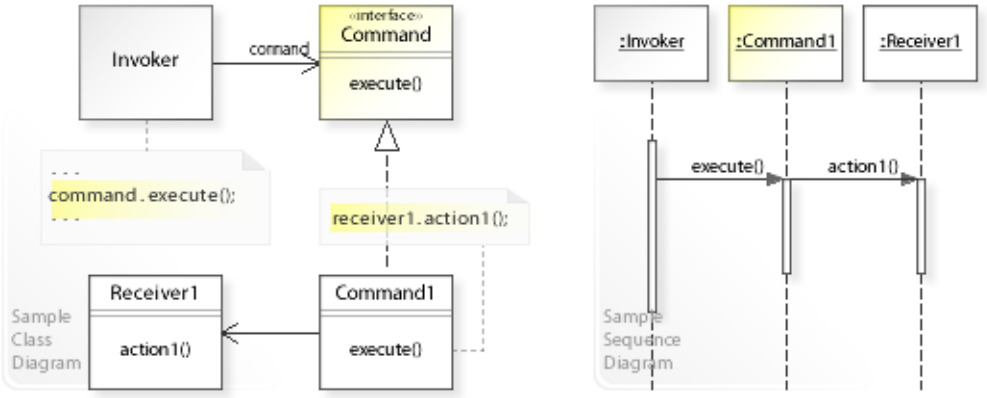
Related Patterns



Key Relationships

- **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



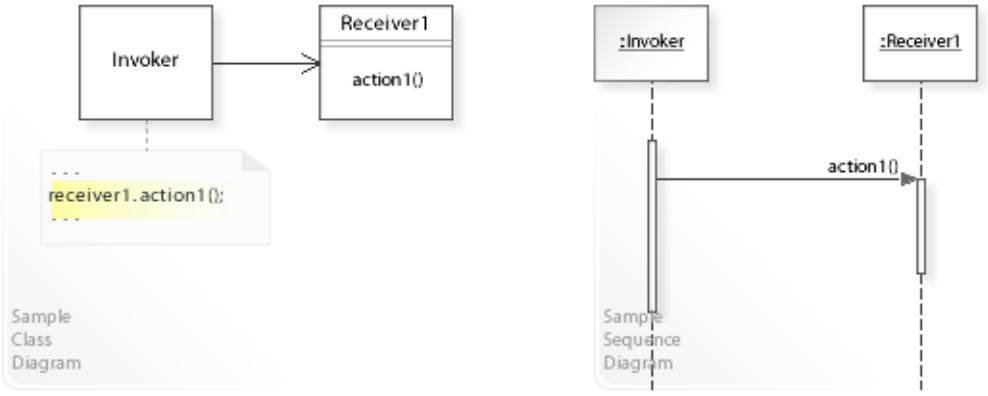
The intent of the Command design pattern is to:
"Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations." [GoF]
See Problem and Solution sections for a more structured description of the intent.

- The Command design pattern solves problems like:
  - *How can coupling the invoker of a request to a request be avoided?*
  - *How can an object be configured with a request?*
- 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 just as *performing, issuing, and sending* a request.
- The Command pattern describes how to solve such problems:
  - *Encapsulate a request as an object* - define separate classes (`Command1,…`) that implement (encapsulate) a request, 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]
- Requests in *UML sequence diagrams*:
  A 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.

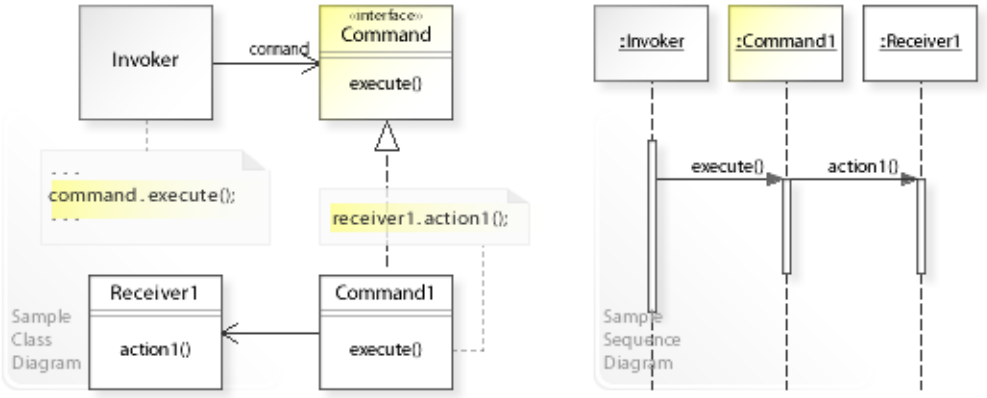
Problem



The Command design pattern solves problems like:
How can coupling the invoker of a request to a request be avoided? How can an object be configured with a request?
See Applicability section for all problems Command can solve. See Solution section for how Command solves the problems.

- An inflexible way is to implement (hard-wire) a request (`receiver1.action1()`) directly within the class (`Invoker`) that invokes the request.
- This commits (couples) the the invoker of a request to a particular request at compile-time and makes it impossible to specify a request at run-time.
  "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 a request at run-time.*
- For example, reusable classes that invoke a request in response to an user input.
  A reusable class should avoid hard-wired requests so that it can be configured with a request at run-time.

Solution



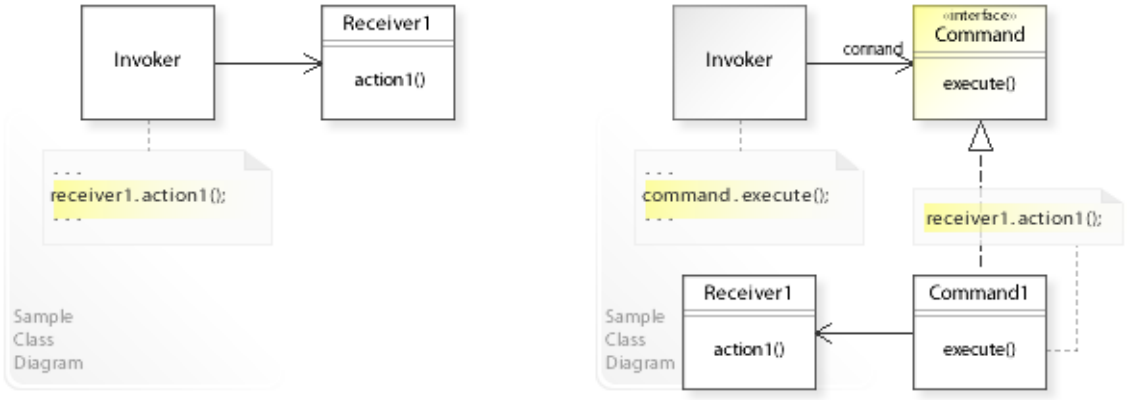
The Command design pattern provides a solution:
Encapsulate a request in a separate `Command` object.
A class delegates a request to a `Command` object
and doesn't know (is independent of) how the request performed.
Describing the Command design in more detail is the theme of the following sections.
See Applicability section for all problems Command can solve.

- The key idea in this pattern is to encapsulate a request in a separate object that (1) can be used and passed around just like other objects and (2) can be queued or logged to be called at a later point.
- **Define separate `Command` objects:**
  - Define a common interface for performing a request (`Command | execute()`).
  - Define classes (`Command1`,…) that implement the `Command` interface.
  - A command can implement arbitrary functionality. In the most simple case, it implements a request by calling an operation on a receiver object (`receiver1.action1()`).
- This enables *compile-time* flexibility (via inheritance).
  New commands can be added and existing ones can be changed independently by defining new (sub)classes.
- **A class (`Invoker`) delegates the responsibility for performing a request to a `Command` object** (`command.execute()`).
- This enables *run-time* flexibility (via object composition).
  A class can be configured with a `Command` object, which it uses to perform a request, and even more, the `Command` object can be exchanged dynamically.
  Commands can be stored (in a history list, for example) to be executed or unexecuted at a later time (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



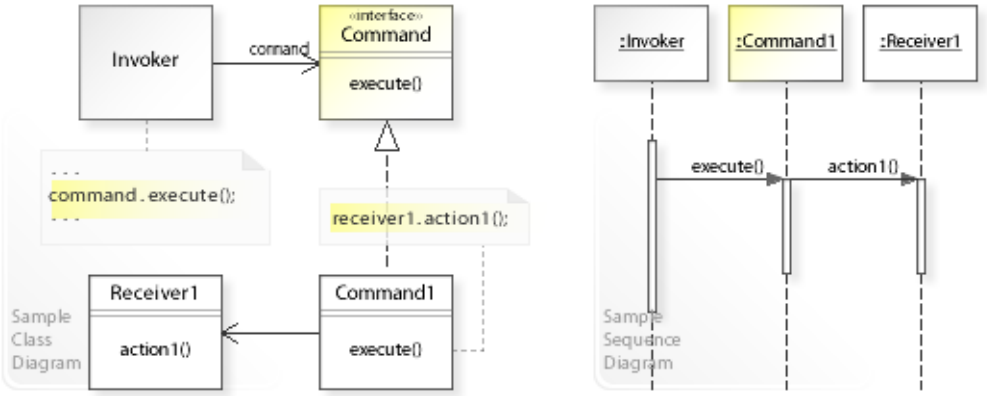
Consider the left design (problem):

- Hard-wired request.
  - The request (`receiver1.action1()`) is implemented (hard-wired) directly within the class (`Invoker`).
  - This makes it impossible to specify a request at run-time.
  - When designing reusable objects, the particular request isn't known at compile-time and should be specified at run-time.

Consider the right design (solution):

- Encapsulated request.
  - The request (`receiver1.action1()`) is implemented (encapsulated) in a separate class (`Command1,…`).
  - The makes it possible to delegate a request to a `Command` object at run-time.
  - Reusable objects can be configured with a `Command` object at run-time.

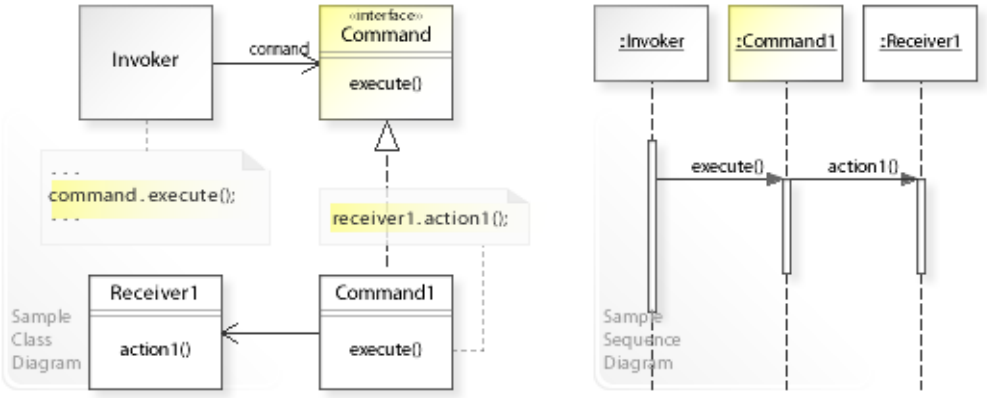
Applicability



Design Problems

- **Avoiding Hard-Wired Requests**
  - How can coupling the invoker of a request to a request be avoided?
- **Exchanging Requests at Run-Time**
  - How can an object be configured with a request?
  - How can a request be selected and exchanged at run-time?
- **Queuing or Logging Requests**
  - How can requests be queued or logged?
  - How can undoable operations be supported?

Structure, Collaboration



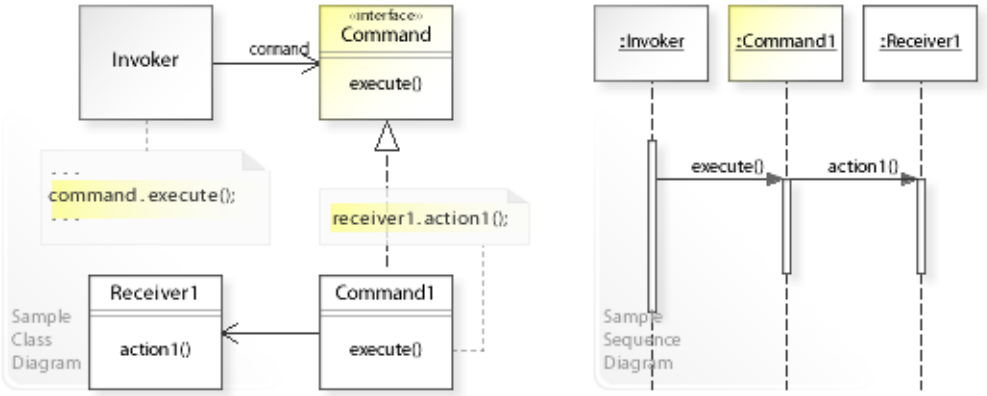
Static Class Structure

- `Invoker`
  - Refers to the `Command` interface to perform a request (`command.execute()`) and is independent of how the request is implemented.
  - Maintains a reference (`command`) to a `Command` object.
- `Command`
  - Defines a common interface for performing a request.
- `Command1`,...
  - Implement the `Command` interface (for example, 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 the `Invoker` object that calls `execute()` on its installed `Command1` object.
- `Command1` calls `action1()` on a `Receiver1` object.
- See also Sample Code / Example 1.

Consequences



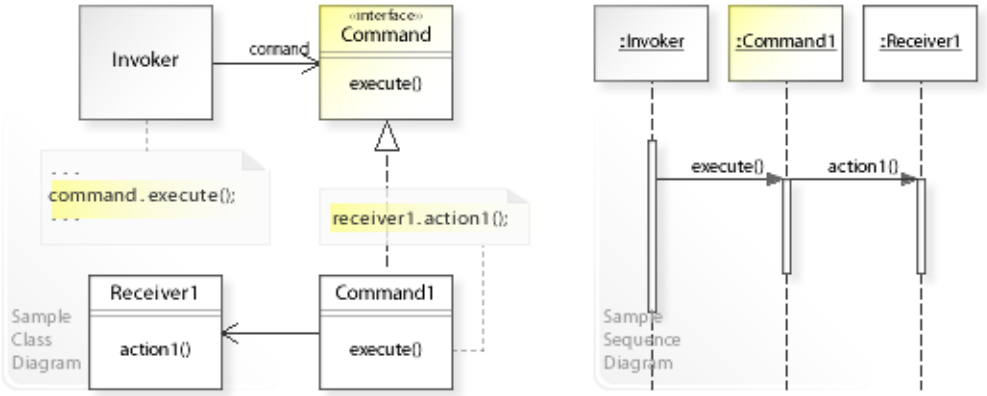
Advantages (+)

- Makes adding new commands easy.
  - "It's easy to add new Commands, because you don' have to change existing classes." [GoF, p237]
- Makes exchanging commands easy.
  - Objects can be configured with the needed `Command` object, and even more, the command can be exchanged dynamically at run-time.

Disadvantages (–)

- Additional level of indirection.
  - Command achieves flexibility by introducing an additional level of indirection (invokers delegate a request to a separate `Command` object), which makes invokers dependent on a `Command` object.

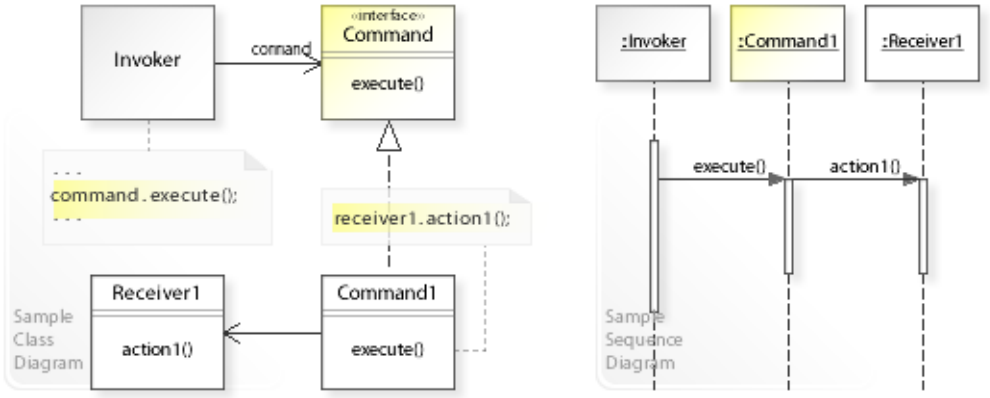
Implementation



Implementation Issues

- **Implementation Variants**
  - A command can implement arbitrary functionality.
    Usually, a command implements a request by calling an operation on a receiver object (`receiver1.action1()`).
  - "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 MyApp {
 3      public static void main(String[] args) {
 4          // Creating an Invoker object
 5          // and configuring it with a Command1 object.
 6          Invoker invoker = new Invoker(new Command1(new Receiver1()));
 7          // Calling an operation on invoker.
 8          invoker.operation();
 9      }
10  }
```

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

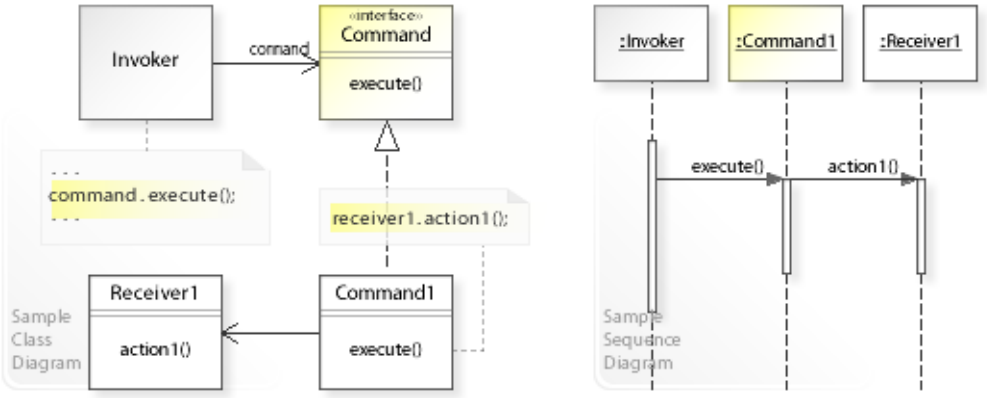
```
 1  package com.sample.command.basic;
 2  public class Invoker {
 3      private Command command;
 4
 5      public Invoker(Command command) {
 6          this.command = command;
 7      }
 8      public void operation() {
 9          System.out.println("Invoker  : Calling execute on the installed 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
 5      public Command1(Receiver1 receiver1) {
 6          this.receiver1 = receiver1;
 7      }
 8      public void execute() {
 9          System.out.println("Command1 : Performing (carrying out) the request ...");
10          receiver1.action1();
11      }
12  }
```

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

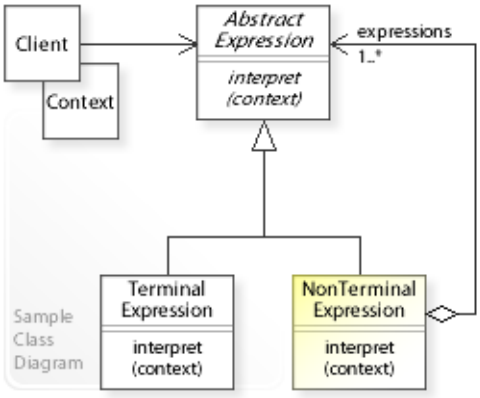
Related Patterns

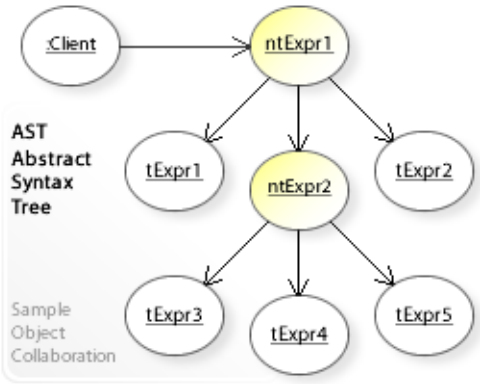


Key Relationships

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

Intent





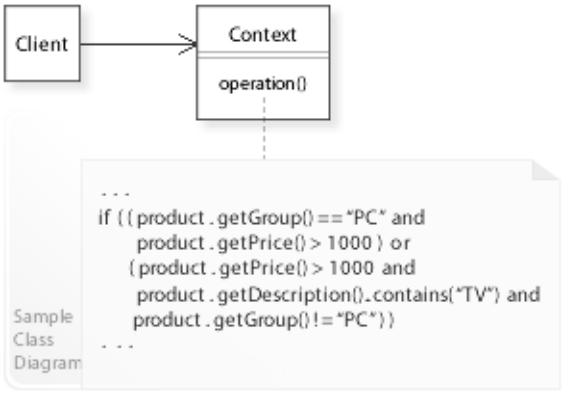
The intent of the Interpreter design pattern is:
"Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language." [GoF]
See Problem and Solution sections for a more structured description of the intent.

- 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.

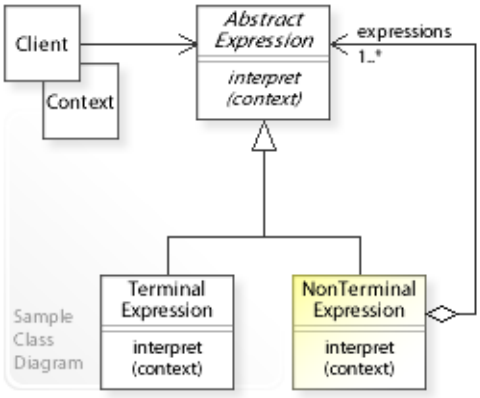
Problem

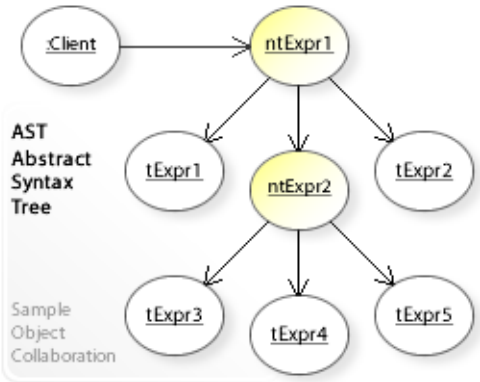


The Interpreter design pattern solves problems like:
How can a grammar for a simple language be defined so that sentences in the language can be interpreted?
See Applicability section for all problems Interpreter can solve. See Solution section for how Interpreter solves the problems.

- "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]
- Specifying a search expression, for example, is a problem that often occurs. Implementing (hard-wiring) it each time it is needed directly within a class (`Context`) is inflexible because it commits (couples) the class to a particular expression and makes it impossible to change or reuse 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 change search expressions dynamically at run-time.*
- For example, an object finder with arbitrary (dynamically changeable) search criteria.
  Instead of hard-wiring 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





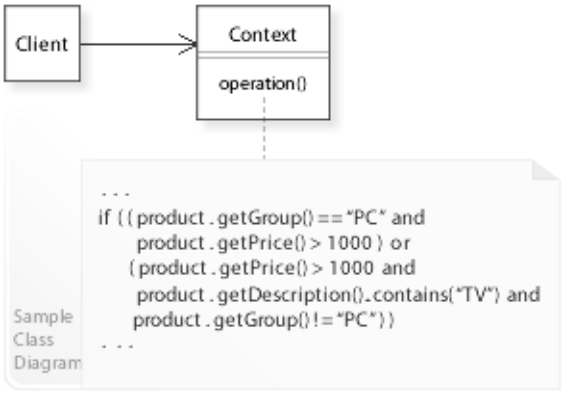
The Interpreter design pattern provides a solution:
(1) **Define a grammar for a simple language by an `Expression` class hierarchy.**
(2) **Represent a sentence in the language by an AST (abstract syntax tree).**
(3) **Interpret a sentence by calling `interpret(context)` on an AST.**
Describing the Interpreter design in more detail is the theme of the following sections. See Applicability section for all problems Interpreter can solve.

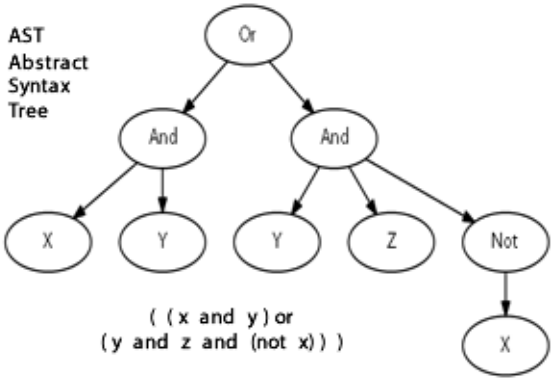
- (1) **Define a grammar for a simple language by an `Expression` class hierarchy:**
  - Define an interface for interpreting an expression (`AbstractExpression | interpret(context)`).
  - Define classes (`TerminalExpression`) that implement the interpretation.
    "[…] many kinds of operations can "interpret" 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





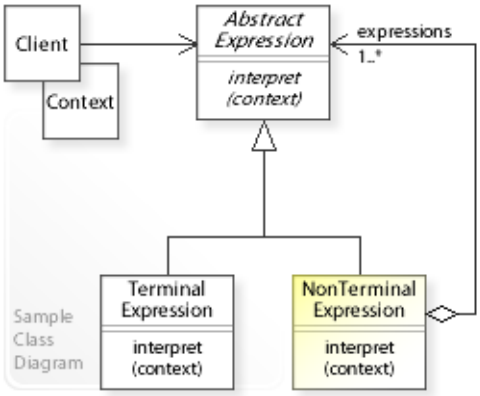
Consider the left design (problem):

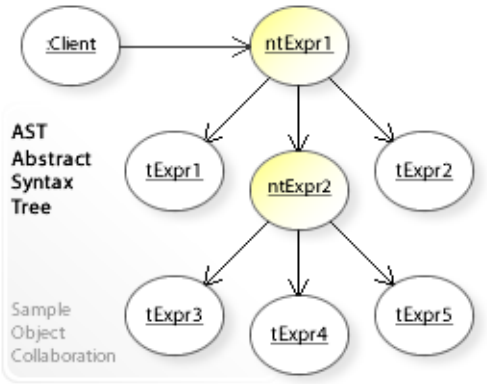
- Hard-wired expression.
  - The search expression is hard-wired directly within a class (`Context`).
  - This makes it hard to specify new expressions or change existing ones both at compile-time and at run-time.

Consider the right design (solution):

- Separated expression.
  - The search expression is represented by a separate 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

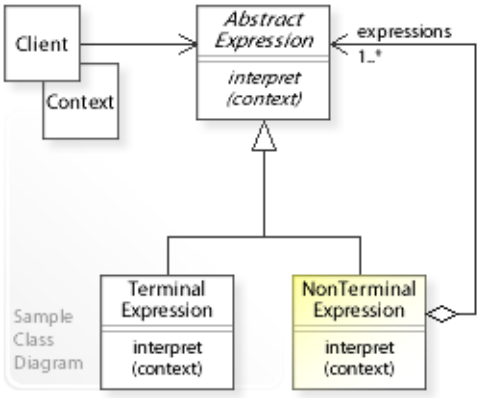


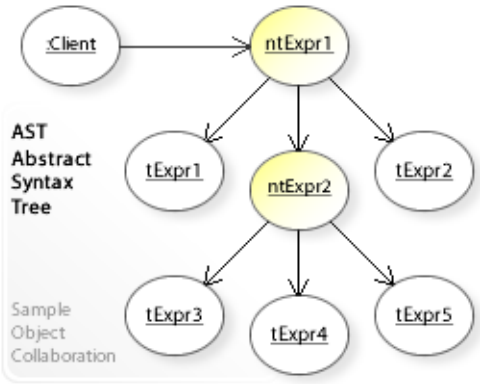


Design Problems

- **Interpreting Sentences in Simple Languages (Domain Specific Languages)**
  - 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





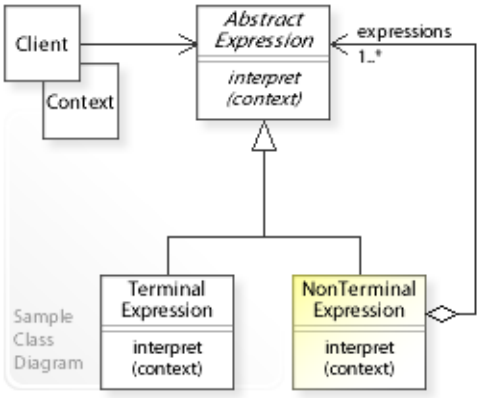
Static Class Structure

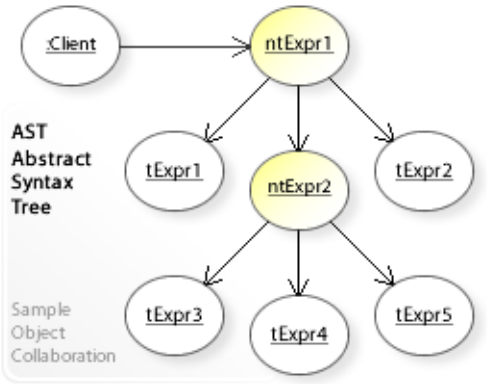
- *`AbstractExpression`*
  - Defines a common interface for interpreting terminal expressions (tExpr) and nonterminal expressions (ntExpr).
- `TerminalExpression`
  - Implements interpretation for terminal expressions.
  - A terminal expression has no child expressions.
- `NonTerminalExpression`
  - 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` sends an interpret request to the AST.
- The nonterminal expression nodes of the AST (`ntExpr1`,`ntExpr2`) forward the request to their child expression nodes.
- The terminal expression nodes (`tExpr1`,`tExpr2`,…) perform the interpretation directly.
- See also Sample Code / Example 1.

Consequences





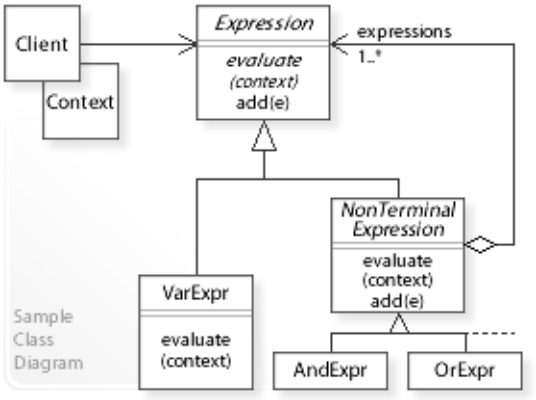
Advantages (+)

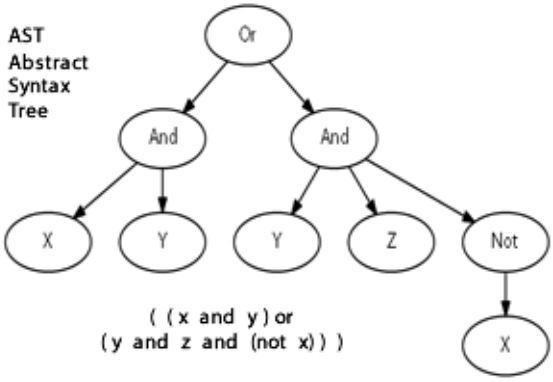
- Makes changing the grammar easy.
  - The Interpreter pattern uses a class hierarchy to represent grammar rules.
  - Clients refer to the `AbstractExpression` interface and are independent of its implementation.
  - Clients do not have to change when new terminal or nonterminal expression classes are 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 representing 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

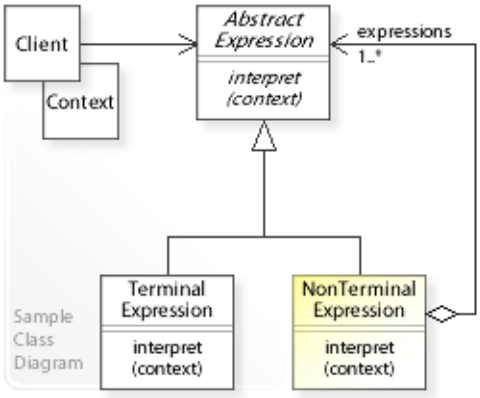
- Let's assume, we want to define a grammar (syntax) for a simple query language so that sentences (search expressions) in the language can be specified and interpreted (evaluated) dynamically. For example, search expressions should look like:
  `( (x and y) or (y and z and (not x)) )`
  where `x,y,z` 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 the above diagrams and Sample Code / Example 2.

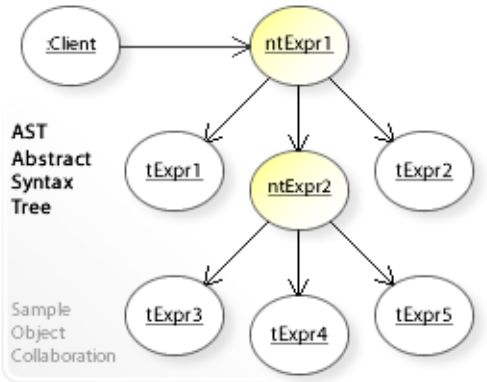
- (1) **Define a grammar for a simple query language:**
  - Grammar rules (in EBNF notation) would look like:
    ```
    expression : andExp | orExp | notExp | varExp | '(' expression ')';
    andExp : expression 'and' expression;
    orExp : expression 'or' expression;
    notExp : 'not' expression;
    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.
```

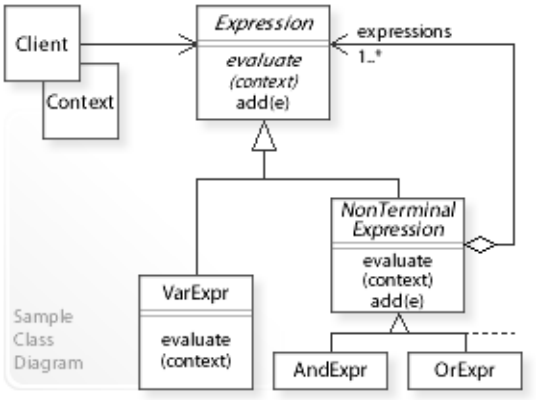
```
 1  package com.sample.interpreter.basic;
 2  public class Context {
 3      // Input data and workspace for interpreting.
 4  }
```

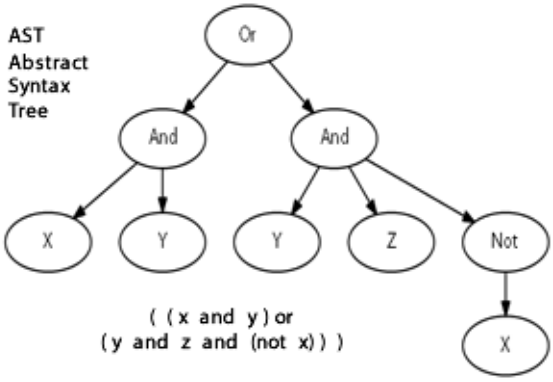
```
 1  package com.sample.interpreter.basic;
 2  public abstract class AbstractExpression {
 3      private String name;
 4      public AbstractExpression(String name) {
 5          this.name = name;
 6      }
 7      public abstract void interpret(Context context);
 8      //
 9      public String 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 and y) or (y and z and (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));
20          //
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          //
```

```
        Expression expression = new OrExpr();
        expression.add(andExpr1);
        expression.add(andExpr2);
        //
        // For each product:
        // - specifying the search criteria dynamically and setting the context
        // - interpreting (evaluating) the AST (search expression).
        //
        Context context = new Context();
        for (Product p : products) {
            // For example, searching products with:
            //  (group == "PC" and prize > 1000) or
            //  (prize > 1000 and description containing "TV" and group is not "PC").
            // Setting VarExpr x,y,z in context to true or false.
            context.setVarExpr(x, p.getGroup() == "PC" ? true : false);
            context.setVarExpr(y, p.getPrice() > 1000 ? true : false);
            context.setVarExpr(z, p.getDescription().contains("TV") ? true : false);
            // Interpreting (evaluating) the AST (search expression).
            if (expression.evaluate(context))
                System.out.println("Product found: " + p.getDescription());
        }
    }
}
```

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

```
package com.sample.interpreter.search;
import java.util.HashMap;
import java.util.Map;
public class Context {
    // Workspace for mapping VarExp name to true or false.
    Map<String, Boolean> varExprMap = new HashMap<String, Boolean>();
    //
    public void setVarExpr(VarExpr v, boolean b) {
        varExprMap.put(v.getName(), b);
    }
    public boolean getVarExpr(String name) {
        return varExprMap.get(name);
    }
}
```

```
package com.sample.interpreter.search;
import java.util.Collections;
import java.util.Iterator;
public abstract class Expression {
    public abstract boolean evaluate(Context context);
    //
    // Defining default implementation for child management operations.
    public boolean add(Expression e) {  // fail by default
        return false;
    }
    public Iterator<Expression> iterator() {
        return Collections.emptyIterator(); // null iterator
    }
}
```

```
package com.sample.interpreter.search;
public class VarExpr extends Expression {  // Terminal Expression
    private String name;
    public VarExpr(String name) {
        this.name = name;
    }
    // Getting true or false from context.
    public boolean evaluate(Context context) {
        return context.getVarExpr(name);
    }
    public String getName() {
        return name;
    }
}
```

```
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  }
```

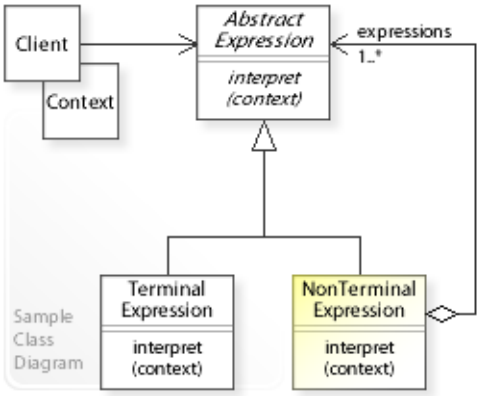
```
*****************************************************************
Other interfaces and classes used in this example.
*****************************************************************
```

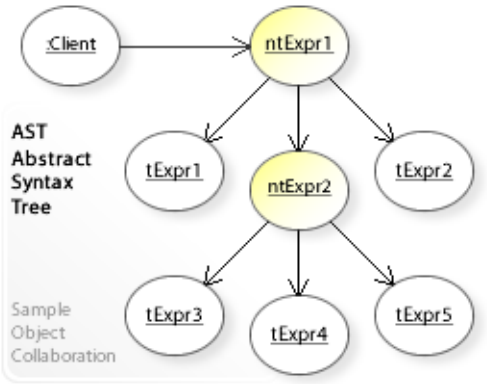
```
 1  package com.sample.data;
 2  public interface Product {
 3      void operation();
 4      String getId();
 5      String getGroup();
 6      String getDescription();
 7      long getPrice();
 8  }
```

```
 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

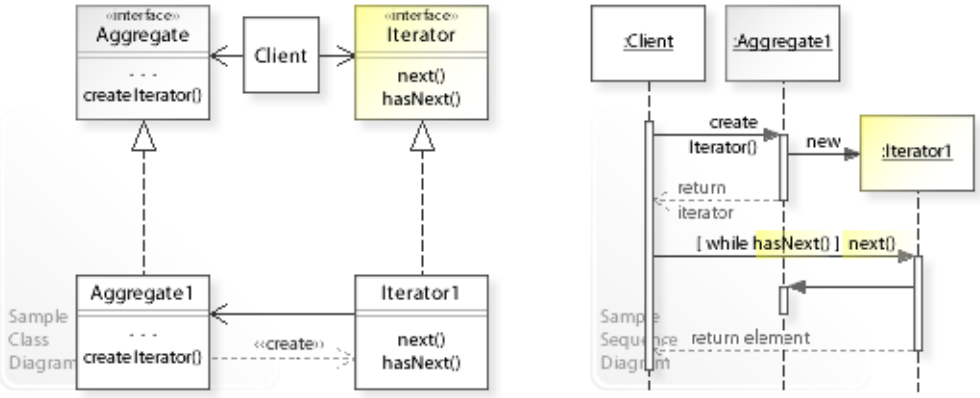




Key Relationships

- **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.

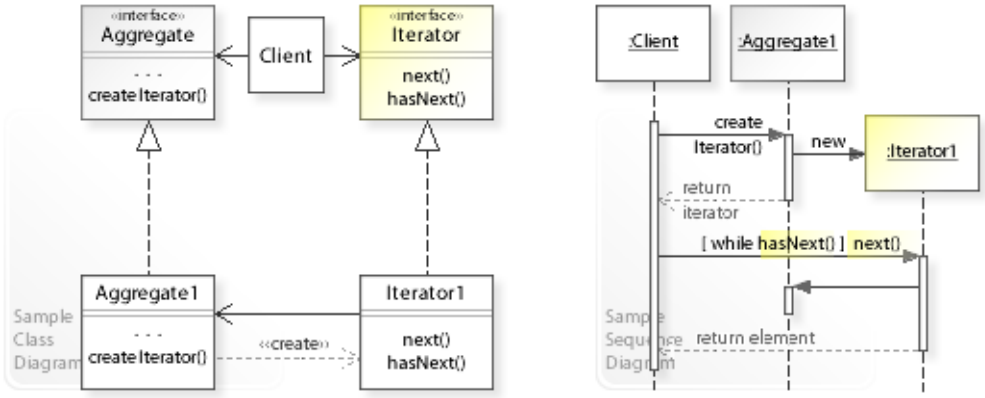
- The Iterator design pattern solves problems like:
  - *How can the elements of an aggregate object be accessed and traversed without exposing its underlying representation?*
- For example, an aggregate object like a list, set, or other kind of collection.
  It should be possible to access and traverse the elements of a collection without having to know its underlying representation (data structures).
- Exposing an aggregate's representation 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?
See Applicability section for all problems Iterator can solve. See Solution section for how Iterator solves the problems.

- One way to solve this problem is to extend the aggregate interface with operations for access and traversal.
  For example, traversing front-to-back: `next(),hasNext()`.
- This commits the aggregate object to particular access and traversal operations and makes it impossible to add new operations later without having to change the aggregate interface.
  For example, traversing back-to-front: `previous(),hasPrivious()`.
  "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 define new access and traversal operations without having to change the aggregate interface.*
- 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 without knowing (depending on) its representation (data structures). See Sample Code / Example 2.

Solution



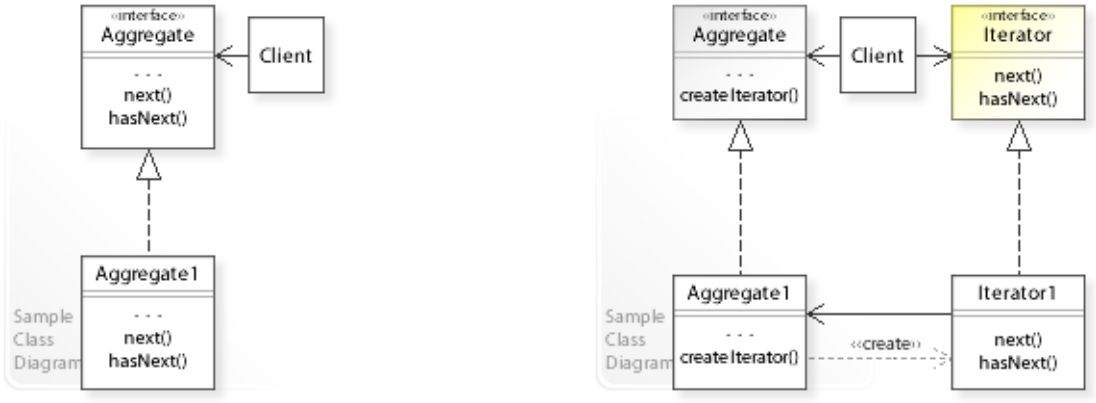
The Iterator design pattern provides a solution:
Encapsulate the access and traversal of an aggregate in a separate `Iterator` object.
Clients request an `Iterator` object from an aggregate (`createIterator()`)
and use it to access and traverse the aggregate.
Describing the Iterator design in more detail 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 an interface for accessing and traversing the elements of an aggregate object (`Iterator | 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 access and traversal operations can be added by defining new iterators.
    For example, traversing back-to-front: `previous(),hasPrevious()`.
- An aggregate provides an interface for creating an iterator (`createIterator()`).
- Clients can use different `Iterator` objects to access and traverse an aggregate object in different ways. Multiple traversals can be in progress on the same aggregate object (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



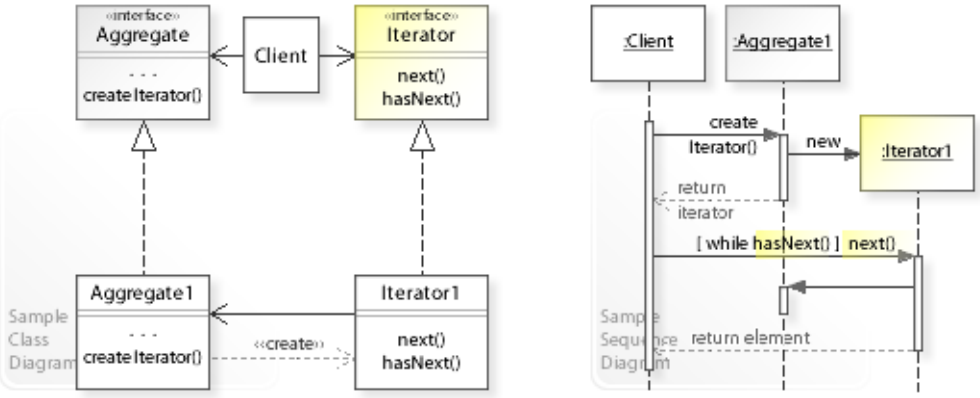
Consider the left design (problem):

- Aggregate responsible for access and traversal.
  - The aggregate is also responsible for accessing and traversing its elements.
  - This makes it impossible to define new traversal operations independently from the aggregate.
- One traversal.
  - Only one traversal can be performed on the same aggregate.

Consider the right design (solution):

- Iterator responsible for 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.
- Multiple traversals.
  - Multiple traversals can be performed on the same aggregate.

Applicability

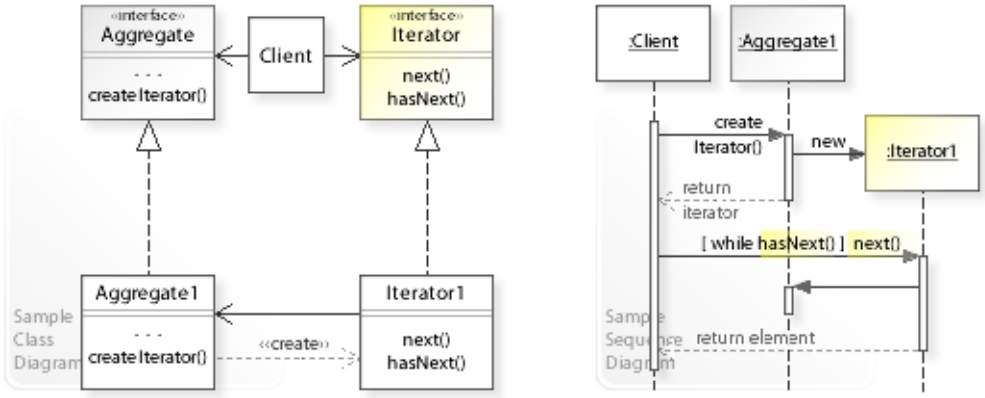


Design Problems

- **Accessing and Traversing Object Structures**
  - How can the elements of an aggregate object be accessed and traversed without exposing its underlying representation?
  - How can new traversal operations be defined for an aggregate object without changing its interface?

- **Performing Different Traversals**
  - How can different traversals be performed on an aggregate object?
  - How can multiple traversals be pending on the same aggregate object (simultaneous traversals)?

Structure, Collaboration



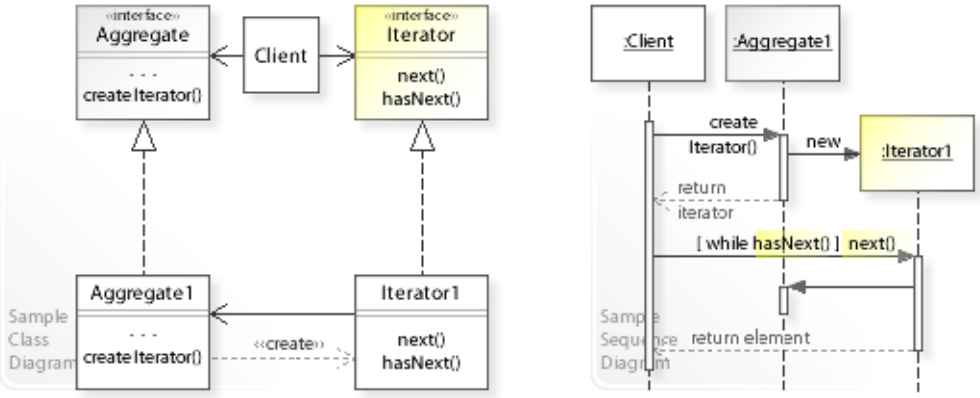
Static Class Structure

- `Client`
  - Refers to the `Aggregate` interface to create an `Iterator` object.
  - Refers to the `Iterator` interface to access and traverse an `Aggregate` object.
- `Aggregate`
  - Defines an interface for creating an `Iterator` object.
- `Aggregate1`,…
  - Implement `createIterator()` by returning an instance of the corresponding iterator class (`Iterator1`).
- `Iterator`
  - Defines an interface for accessing and traversing the elements of an `Aggregate` object.
- `Iterator1`,…
  - Implement the `Iterator` interface.
  - An iterator is usually implemented as inner class of an aggregate class so that it can access the internal (private) data structures of the aggregate.

Dynamic Object Collaboration

- In this sample scenario, a `Client` object uses an `Iterator1` object to traverse an `Aggregate1` object front-to-back.
- The interaction starts with the `Client` object that calls `createIterator()` on the `Aggregate1` object.
- `Aggregate1` creates an `Iterator1` object and returns (a reference to) it to the `Client`.
- Thereafter, the `Client` uses the `Iterator1` to traverse the elements of `Aggregate1` front-to-back (`while iterator.hasNext(): iterator.next()`).
- See also Sample Code / Example 1.

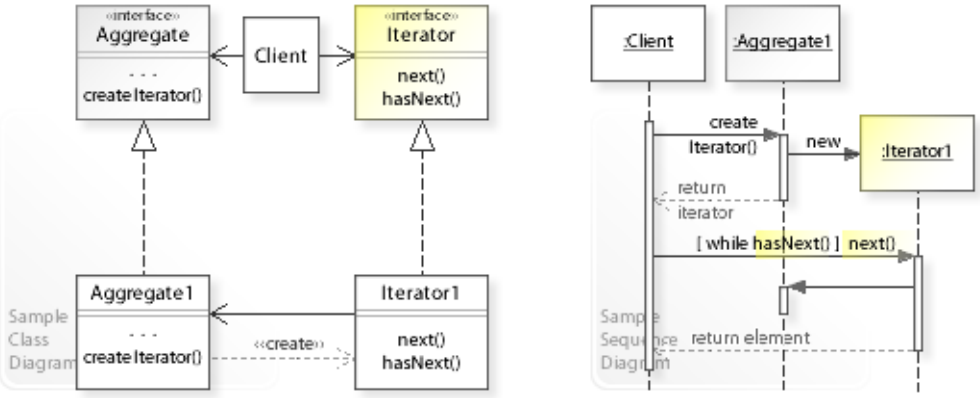
Consequences



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]

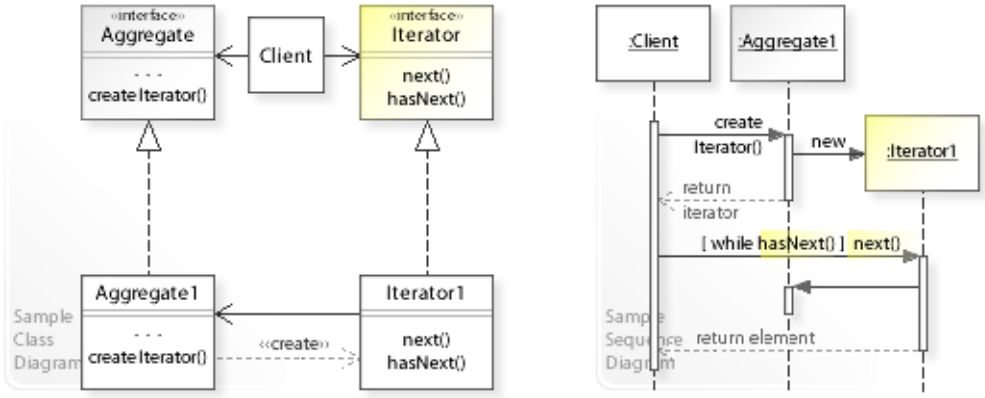
Implementation



Implementation Issues

- **Implementing iterators.**
  - There are two main variants to implement an iterator:
- Variant1: **Iterators have privileged access to an aggregate.**
  - An iterator is implemented as *inner class* of an aggregate implementation class. This enables the iterator to access the private data structures of the aggregate (see Sample Code / Example 1 and 2).
  - The aggregate provides an interface for creating an iterator object (`createIterator()`). Aggregate implementation classes are responsible for instantiating the appropriate iterator class. (This is an example of applying the Factory Method design pattern.)
- Variant2: **Iterators access an aggregate through its interface.**
  - Another way is to design an extended `Aggregate` interface so that iterators can access the aggregate efficiently.

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
```

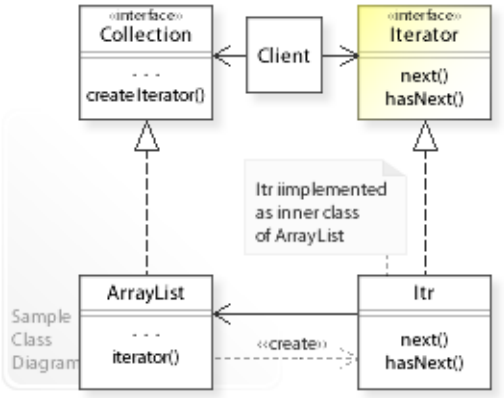
```
 1  package com.sample.iterator.basic;
 2  public interface Aggregate<E> {
 3      // ...
 4      Iterator<E> createIterator();
 5      boolean add(E element);
 6  }
```

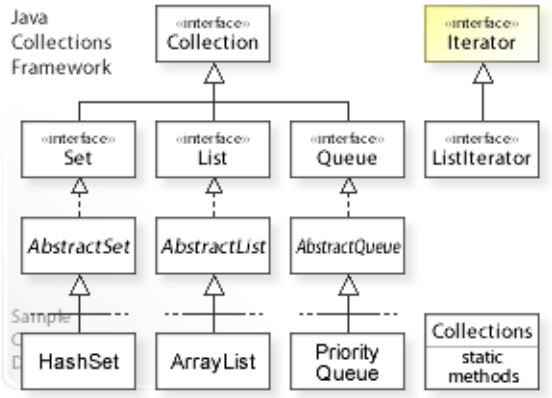
```
 1  package com.sample.iterator.basic;
 2  public interface Iterator<E> {
 3      E next();
 4      boolean hasNext();
 5  }
```

```
 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 }
```

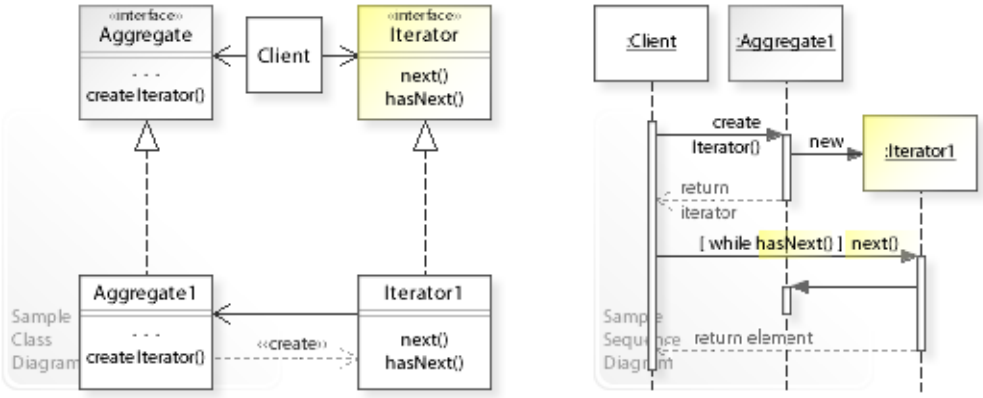
```
1  // List is an ordered collection (also known as a sequence).
```

```
// Unlike sets, lists allow duplicate elements.
// ListIterator allows element insertion/replacement and
// bidirectional access in addition to Iterator.
// ...
package java.util;  // Provided by the Java platform.
public interface List<E> extends Collection<E> {
    // Iterating
    ListIterator<E> listIterator();   // Starting at position 0
    ListIterator<E> listIterator(int index);  // Starting at position index
    // Positional access
    E get(int index);
    E set(int index, E element);
     // Search
    int indexOf(object o);
    // ...
}
```

```
// ArrayList is a resizable array implementation of the List interface.
package java.util;  // Provided by the Java platform.
public class ArrayList<E> extends AbstractList<E>
    implements List<E>, RandomAccess, Cloneable, java.io.Serializable {
    // The array buffer into which the elements of the ArrayList are stored.
    // The capacity of the ArrayList is the length of this array buffer.
    private transient Object[] elementData;
    // The size of the ArrayList (the number of elements it contains).
    private int size;
    // ...
    public ListIterator<E> listIterator() {
        return new ListItr(0);
    }
    public Iterator<E> iterator() {
        return new Itr();
    }
    private class Itr implements Iterator<E> {
        int cursor;       // index of next element to return
        int lastRet = -1; // index of last element returned; -1 if no such
        int expectedModCount = modCount;
        public boolean hasNext() {
            return cursor != size;
        }
        public E next() {
            checkForComodification();
            int i = cursor;
            if (i >= size)
                throw new NoSuchElementException();
            Object[] elementData = ArrayList.this.elementData;
            if (i >= elementData.length)
                throw new ConcurrentModificationException();
            cursor = i + 1;
            return (E) elementData[lastRet = i];
        }  // ...
    }
    private class ListItr extends Itr implements ListIterator<E> {
        ListItr(int index) {
            super();
            cursor = index;
        }
        public boolean hasPrevious() {
            return cursor != 0;
        }
        public int nextIndex() {
            return cursor;
        }
        public int previousIndex() {
            return cursor - 1;
        }
        public E previous() {
            checkForComodification();
            int i = cursor - 1;
            if (i < 0)
                throw new NoSuchElementException();
            Object[] elementData = ArrayList.this.elementData;
            if (i >= elementData.length)
                throw new ConcurrentModificationException();
            cursor = i;
            return (E) elementData[lastRet = i];
```

```
60            }  // ...
61        }  // ...
62    }
```

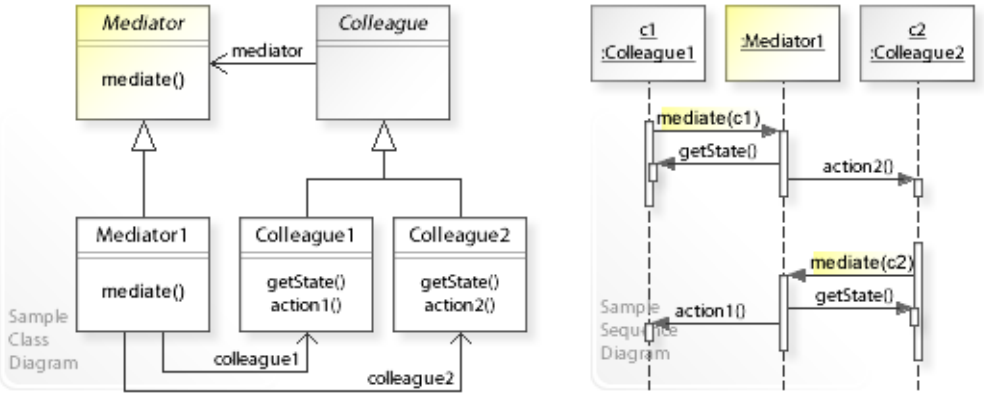
Related Patterns



Key Relationships

- **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.

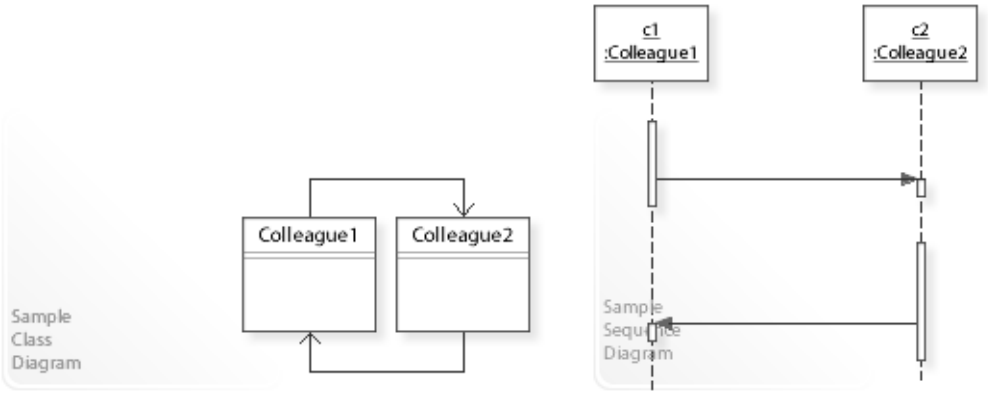
Intent



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 tight coupling between a set of interacting objects be avoided?*
  - *How can the interaction between a set of objects be changed independently?*
- *Coupling* is the degree to which objects depend on each other.
  - *Tightly coupled objects* are hard to implement, change, test, and reuse because they depend on (refer to and know about) many different objects.
  - *Loosely coupled objects* are easier to implement, change, test, and reuse because they have only minimal dependencies on other objects.
- The Mediator pattern describes how to solve such problems:
  - *Define an object (`Mediator`) that encapsulates how a set of objects interact.*
  - The key idea in this pattern is to let objects interact with each other indirectly through a `Mediator` object that controls and coordinates the interaction.
    This makes the objects loosely coupled because they only depend on (refer to and know about) the simple `Mediator` interface.

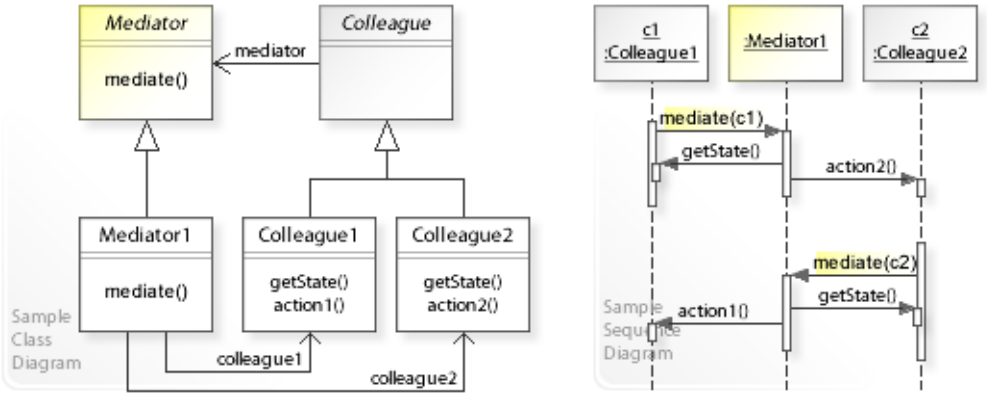
Problem



The Mediator design pattern solves problems like:
How can tight coupling between a set of interacting objects be avoided?
How can the interaction between a set of objects be changed independently?
See Applicability section for all problems Mediator can solve. See Solution section for how Mediator solves the problems.

- An inflexible way is to define a set of interacting objects (`Colleague1`,`Colleague2`,…) by referring to (and update) each other directly, which results in many interconnections between them.
- This tightly couples the objects to each other and makes it impossible to change the interaction independently from (without having to change) the objects, and it stops the objects from being reusable and makes them hard to test.
  Tightly coupled objects are hard to implement, change, test, and reuse
  because they depend on (refer to and know about) many different 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 (1) to change the interaction behavior independently from (without having to change) the objects and (2) to reuse the objects in different applications.

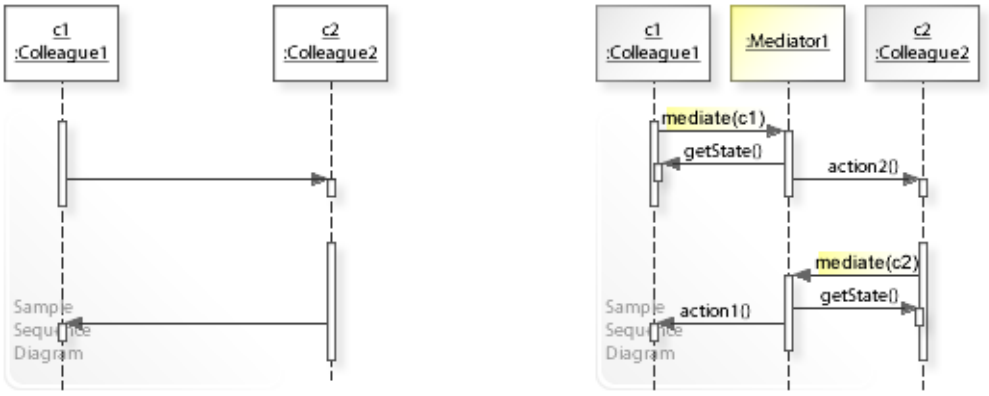
Solution



The Mediator design pattern provides a solution:
Define a separate `Mediator` object that encapsulates how a set of objects interact.
Objects interact with a `Mediator` object
instead of interacting with each other directly.
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 controls and coordinates the interaction.
- **Define a separate `Mediator` object:**
  - Define an interface for interacting with colleagues (`Mediator | mediate()`).
  - Define classes (`Mediator1`,…) that implement the interaction behavior by controlling and coordinating the interaction between colleagues.
- This enables *compile-time* flexibility (via class inheritance).
  New colleagues can be added and the interaction behavior of existing ones can be changed independently by defining new `Mediator` (sub)classes.
- **Colleagues delegate interaction to a `Mediator` object** (`mediator.mediate()`).
- This makes colleagues loosely coupled because they only refer to and know about 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



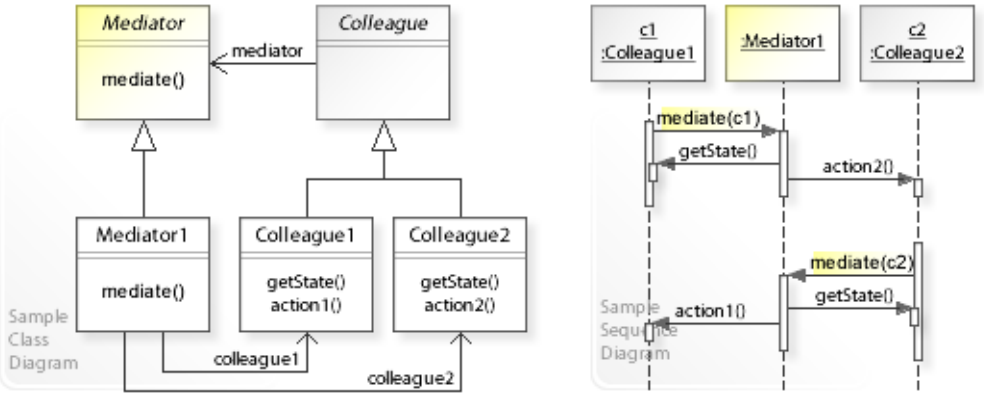
Consider the left design (problem):

- 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 having different interfaces, which makes them hard to implement, change, test, 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, test, 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]

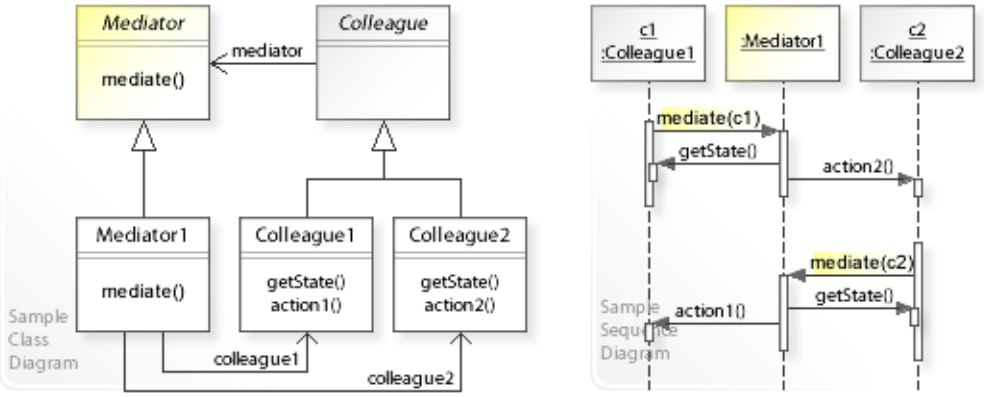
Applicability



Design Problems

- **Avoiding Tight Coupling Between Interacting Objects**
  - How can tight coupling between a set of interacting objects be avoided?
  - How can the interaction between a set of objects be changed independently from the objects?

Structure, Collaboration



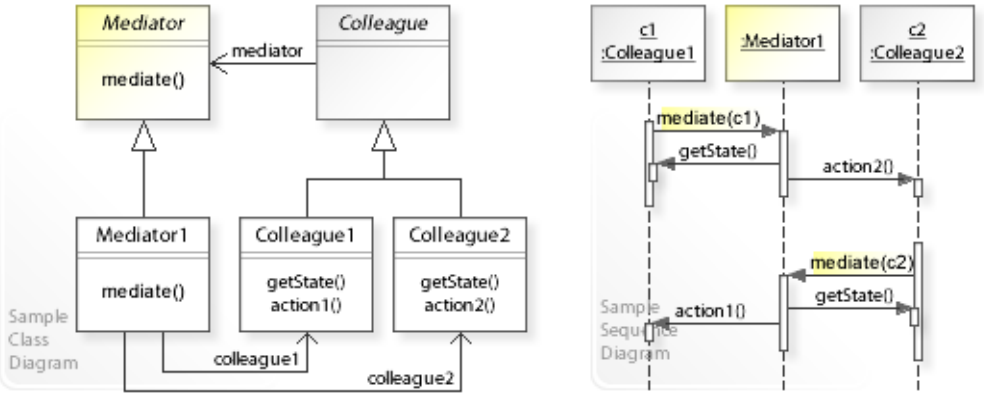
Static Class Structure

- *`Mediator`*
  - Defines an interface for controlling and coordinating the interaction among colleagues.
- `Mediator1,...`
  - Implement the `Mediator` interface.
  - Maintain explicit references (`colleague1,colleague2,...`) to colleagues.
- `Colleague1,Colleague2,...`
  - Refer to the `Mediator` interface instead of referring to each other directly.

Dynamic Object Collaboration

- In this sample scenario, a `Mediator1` object mediates (controls and coordinates) the interaction between `Colleague1` and `Colleague2` objects (to synchronize their state, for example).
  Let's assume that `Colleague1` and `Colleague2` are configured with a `Mediator1` object.
- Let's assume that the state of `Colleague1` changes, which causes `Colleague1` to call `mediate(this)` on its `Mediator1` object.
- `Colleague1` passes itself (`this`) to the `Mediator1` so that `Mediator1` can call back and get the changed data.
- The `Mediator1` gets the changed data from `Colleague1` and performs an `action2()` on `Colleague2`.
- Thereafter, assuming that the state of `Colleague2` changes, `Colleague2` calls `mediate(this)` on its `Mediator1`.
- The `Mediator1` now gets the changed data from `Colleague2` and performs an `action1()` on `Colleague1`.
- See also Sample Code / Example 1.

Consequences



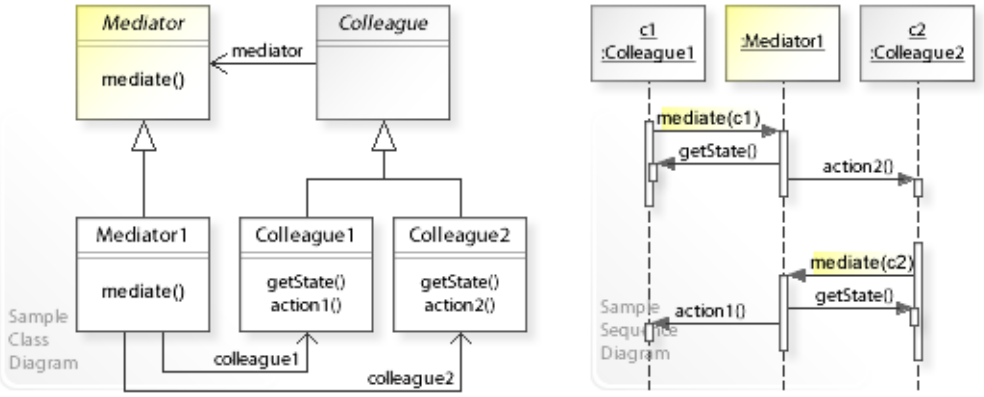
Advantages (+)

- 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` (sub)classes.

Disadvantages (–)

- Can make the mediator complex.
  - Because the mediator encapsulates (centralizes) the interaction behavior of a set of objects, it can get 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]

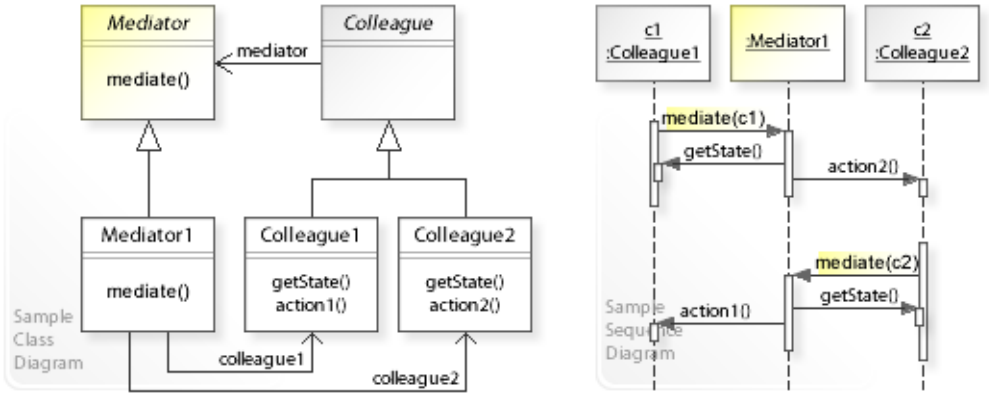
Implementation



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 MyApp {
 3      public static void main(String[] args) {
 4          Mediator1 mediator = new Mediator1();
 5          // Creating colleagues
 6          // and configuring them with a Mediator1 object.
 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) Changing state of Colleague1 ...");
13          c1.setState("Hello World1!");
14
15          System.out.println("\n(2) Changing state of Colleague2 ...");
16          c2.setState("Hello World2!");
17      }
18  }
```

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

(2) Changing state of Colleague2 ...
    Colleague2: My state changed to: Hello World2! Calling my mediator ...
    Mediator  : Mediating the interaction ...
    Colleague1: My 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.
```

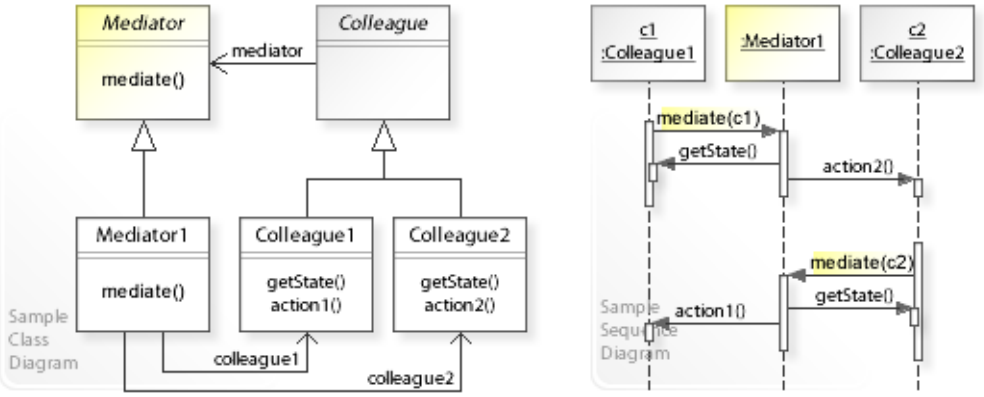
```
18          if (colleague == colleague2) {
19              // Performing an action on colleague1.
20              String state = colleague2.getState();
21              colleague1.action1(state);
22          }
23      }
24  }
```

```
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: My state changed to: "
14                     + this.state + " Calling my 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: My state synchronized to: "
22                     + this.state);
23     }
24 }
```

```
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: My state changed to: "
14                     + this.state + " Calling my 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: My state synchronized to: "
22                     + this.state);
23     }
24 }
```

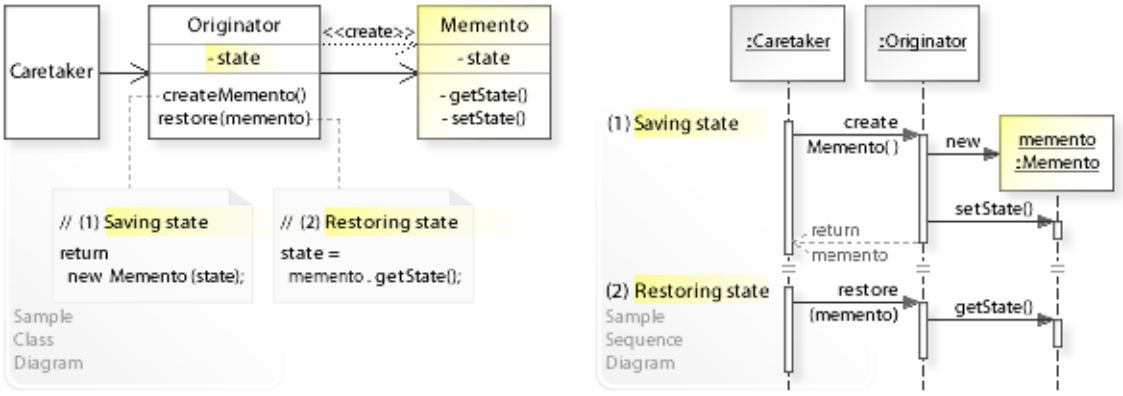
Related Patterns



Key Relationships

- **Mediator - Observer**
  - Mediator provides a way to keep interacting objects loosely coupled by defining a `Mediator` object that centralizes (encapsulates) interaction behavior.
  - Observer provides a way to keep interacting objects loosely coupled by defining `Subject` and `Observer` objects that distribute interaction behavior so that when a subject changes state all registered observers are updated.
  - "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 an object's representation (data structures like fields, arrays, collections, etc.) and implementation inside the object so that they cannot be accessed from outside the object.
- *Internal state* of an object means all internal data structures plus their values.
- The problem here is to save an object's internal state without making the representation (data structures) visible and accessible from outside the object.

Problem

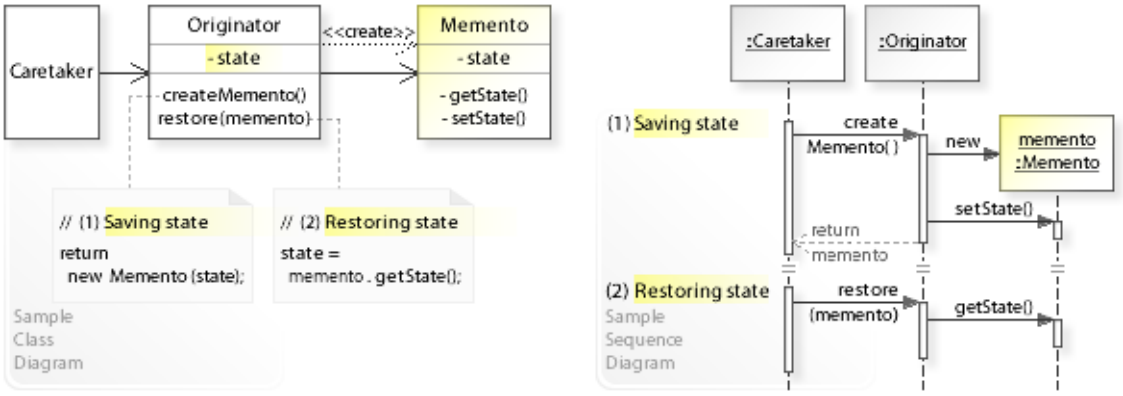
The Memento design pattern solves problems like:
Without violating encapsulation,
how can an object's internal state be captured and externalized
so that the object can be restored to this state later?
See Applicability section for all problems Memento can solve. See Solution section for how Memento solves the problems.

- A well-designed object is *encapsulated*.
  That means, an object's representation (data structures) and implementation are hidden inside the object and are invisible and inaccessible from outside the object.
- In standard object-oriented languages, encapsulation is supported by specifying a *private* access level to protect against access from outside the object (in UML class diagrams this is shown as minus sign).
- *The problem here is to save the internal state of an object externally (to another object) without making the representation (data structures) of the object accessible from outside the object.*
- For example, designing checkpoints and undo mechanisms.
  It should be possible to save a snapshot of an object's internal state externally so that the object can be restored to this state later. A direct access to the object's 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]
- As a reminder, an object has an *outside view* (public interface/operations) and an *inside view* (private/hidden 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 any changes of an object's representation and implementation.
  That's 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]

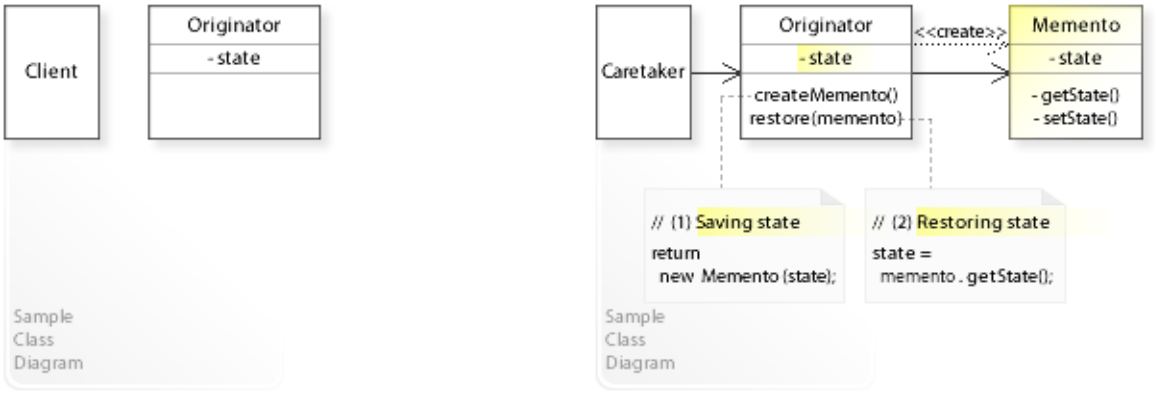
Solution



The Memento design pattern provides a solution:
Define `Originator` and `Memento` objects
so that an originator saves/restores its internal state to/from a memento.
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 to make an object (originator) itself responsible for saving/ restoring its internal state (to/from a memento). Only the originator that created a memento is permitted to access it.
- **Define `Originator` and `Memento` objects:**
  - `Originator` defines an operation for saving its internal state to a memento (`createMemento(): return new Memento(state)`) and for restoring to a previous state from a memento (`restore(memento): state = memento.getState()`).
  - `Memento` defines the required data structures to store an originator's internal state, and it is protected against access by objects other than the originator.
    This is usually achieved by implementing memento as *inner class* of originator and declaring all members of memento *private* (see Implementation and Sample Code).
- Clients (caretaker) that are responsible for saving/restoring an originator's internal state hold a list of mementos so that a memento can be passed back to the originator to restore to a previous state.
  But a caretaker isn't permitted to access a memento. Only the originator that created a memento is permitted to access it. This enables to save and restore originator's internal state without violating its encapsulation.

Motivation 1



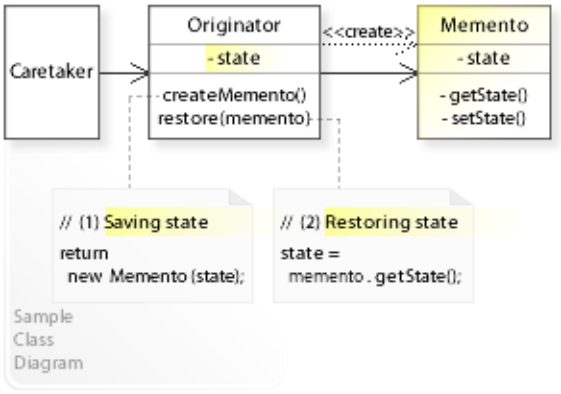
Consider the left design (problem):

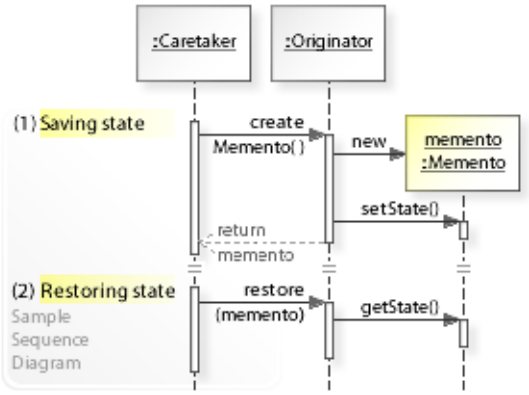
- Originator's internal state can't be saved.
  - Clients can not save the originator's internal state because it is encapsulated (hidden inside the originator) and can not be accessed from outside the originator.

Consider the right design (solution):

- Originator's internal state can be saved.
  - The originator itself is responsible for saving its internal state to a memento (`createMemento()`).
  - Clients (caretaker) can save the originator's internal state by calling `createMemento()` on the originator, which creates and returns a memento.
  - Clients (caretaker) are responsible for requesting and holding mementos, but they aren't permitted to access them.
  - Only the originator that created the mementos is permitted to access them.

Applicability

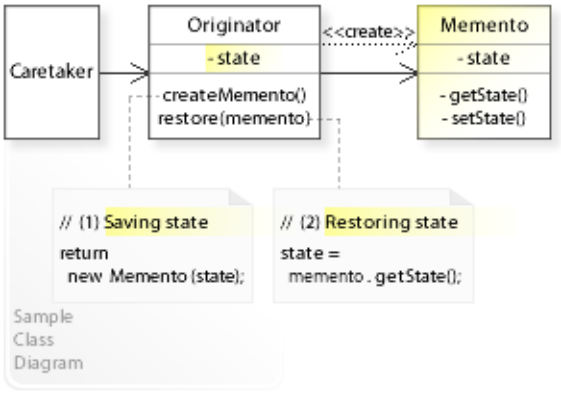


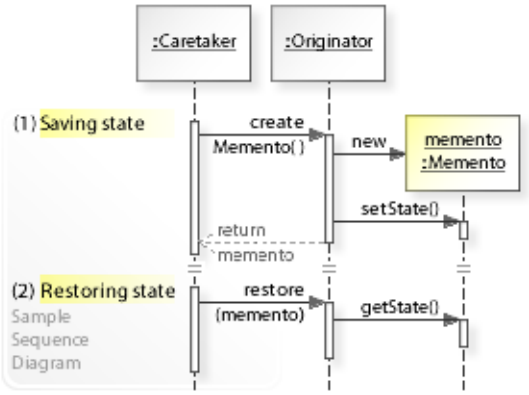


Design Problems

- **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





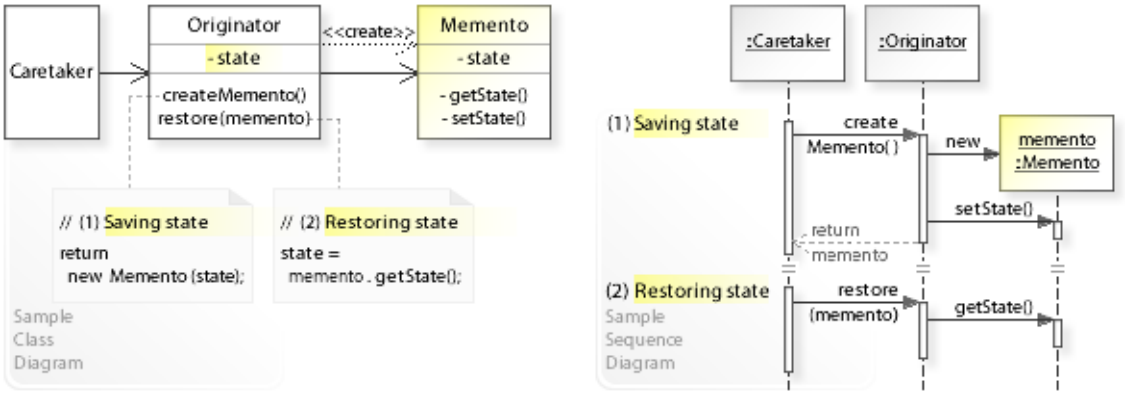
Static Class Structure

- `Caretaker`
  - Refers to the `Originator` class to save and restore originator's internal state.
  - Holds `Memento` objects, created and returned by the originator, and passes back a memento to the originator to restore to a previous state.
  - Isn't permitted to access a `Memento` object.
- `Originator`
  - Defines an operation (`createMemento()`) for saving its current internal state to a `Memento` object.
  - Defines an operation (`restore(memento)`) for restoring to a previous state from a passed in `Memento` object.
- `Memento`
  - Stores an originator's internal state.
  - Only the originator that created a memento can access it.
  - This is usually implemented by making memento an *inner class* of originator and declaring all members of memento *private*.
  - See also Implementation and Sample Code.

Dynamic Object Collaboration

- This sample scenario shows (1) saving and (2) restoring the internal state of an `Originator` object.
- (1) To **save** the `Originator`'s current internal state, the `Caretaker` calls `createMemento()` on the `Originator`.
- The `Originator` creates a new `Memento` object, saves its current internal state (`setState()`), and returns the `Memento` object to the `Caretaker`.
- The `Caretaker` holds (takes care of) the returned `Memento` object(s).
- (2) To **restore** the `Originator` to a previous state, the `Caretaker` calls `restore(memento)` on the `Originator` by providing the `Memento` object to be restored.
- The `Originator` gets the state to be restored from the provided `Memento` object (`getState()`).
- See also Sample Code / Example1.

Consequences



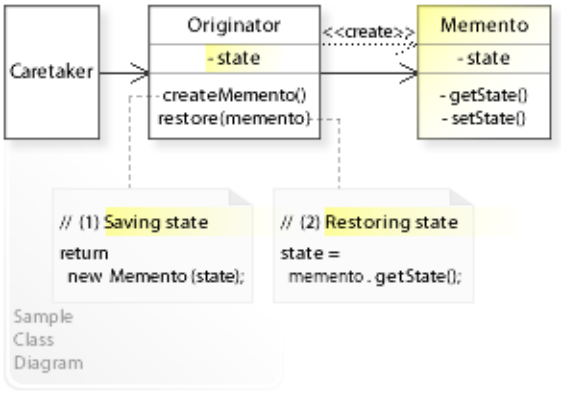
Advantages (+)

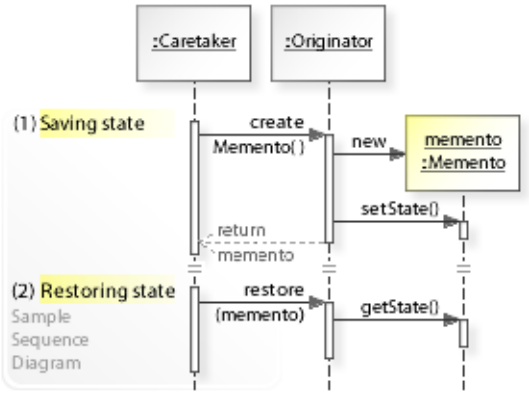
- Preserves encapsulation.
  - An object's internal state can be saved externally (to another object) without violating encapsulation (without making the internal data structures accessible).

Disadvantages (–)

- May introduce run-time costs.
  - Creating 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

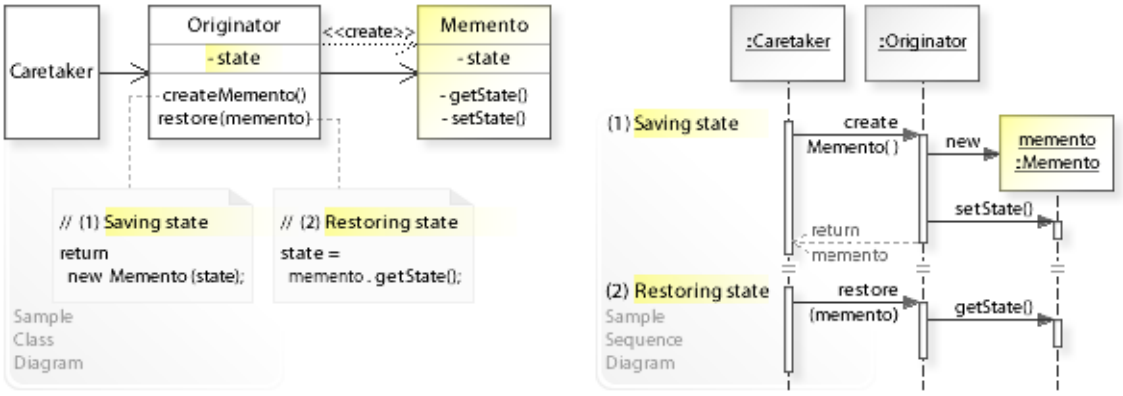




Implementation Issues

- **Originator must have privileged access.**
  - A 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.

Sample Code 1



Basic Java code for implementing the sample UML diagrams.

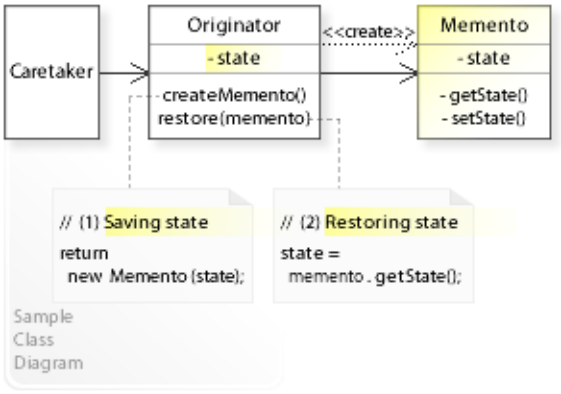
```
 1  package com.sample.memento.basic;
 2  import java.util.ArrayList;
 3  import java.util.List;
 4  public class Caretaker {
 5      // Running the Caretaker class as application.
 6      public static void main(String[] args) {
 7          Originator originator = new Originator();
 8          Originator.Memento memento; // Memento is inner class of Originator
 9          // List of memento objects.
10          List<Originator.Memento> mementos = new ArrayList<Originator.Memento>();
11
12          originator.setState("A");
13          // Saving state.
14          memento = originator.createMemento();
15          mementos.add(memento); // adding to list
16          System.out.println("(1) Saving current state ...... : "
17                             + originator.getState());
18          originator.setState("B");
19          // Saving state.
20          memento = originator.createMemento();
21          mementos.add(memento); // adding to list
22          System.out.println("(2) Saving current state ...... : "
23                             + originator.getState());
24          // Restoring to previous state.
25          memento = mementos.get(0); // getting previous (first) memento from the list
26          originator.restore(memento);
27          System.out.println("(3) Restoring to previous state : "
28                             + originator.getState());
29      }
30  }
```

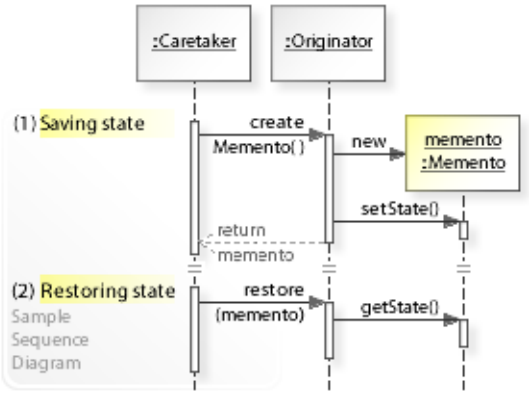
```
(1) Saving current state ...... : A
(2) Saving current state ...... : B
(3) Restoring to previous state : A
```

```
 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 restore(Memento memento) {
14          state = memento.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 by originator.
26      //
27      public class Memento {
28          // Storing Originator's internal state.
29          private String state;
30          // ...
31          private String getState() {
32              return state;
33          }
34          private void setState(String state) {
35              this.state = state;
36          }
37      }
38  }
```

Related Patterns

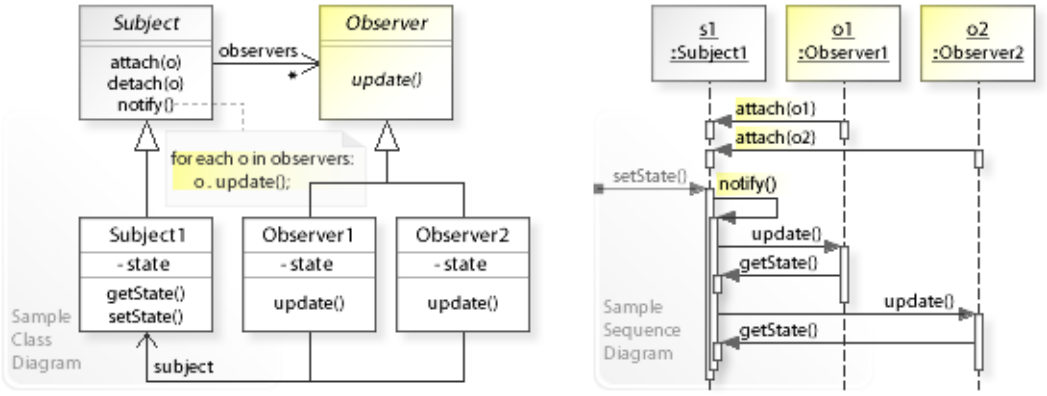




Key Relationships

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

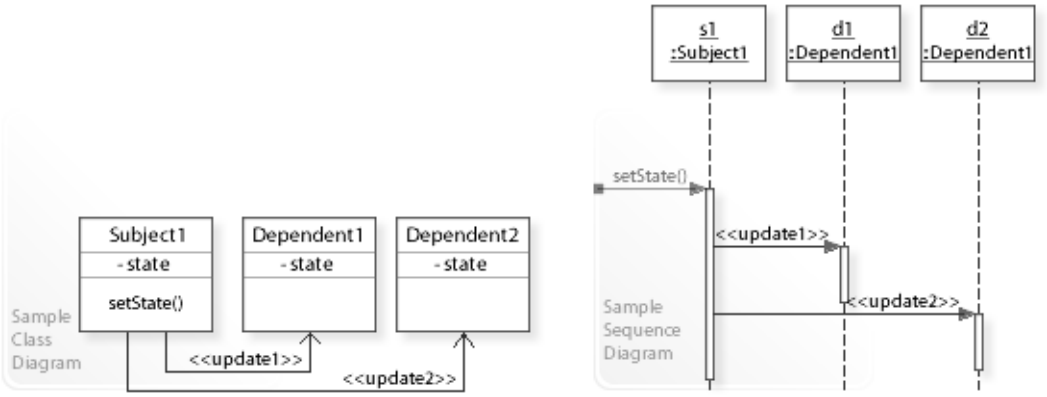
Intent



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 objects depend on each other.
  - *Tightly coupled objects* are hard to implement, change, test, and reuse because they depend on (refer to and know about) many different objects (having different interfaces).
  - *Loosely coupled objects* are easier to implement, change, test, and reuse because they have only minimal dependencies on other objects.
- 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.

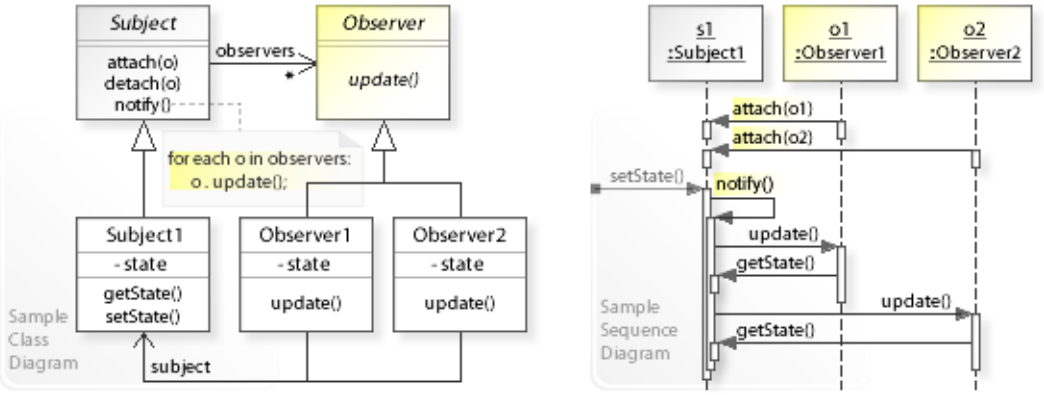
Problem



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. See Solution section for how Observer solves the problems.

- An inflexible way to define a one-to-many dependency between objects is to define one object (`Subject1`) that implements updating the state of dependent objects (`Dependent1`, `Dependent2`, …). That means, subject must know how to update the state of many different objects (having different interfaces).
- This commits (tightly couples) the subject to particular dependent objects and makes it impossible to change the objects (add new ones or remove existing ones) independently from (without having to change) the subject. It stops the subject from being reusable and makes the subject hard to test.
  Tightly coupled objects are hard to implement, change, test, and reuse
  because they depend on (refer to and know about) many different objects.
  "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 in a one-to-many dependency loosely coupled.*
- For example, a data object and multiple presentation objects in a GUI/Web application.
  When the data object changes state, all presentation objects that depend on this data object's state should be updated (synchronized) automatically and immediately to reflect the data change (see Sample Code / Example 2).
- For example, event handing in a GUI/Web application.
  When a user clicks a button, all objects that depend on (listen for) the button's 'click event' should be notified that a 'click event' occurred (see Sample Code / Example 3/4).

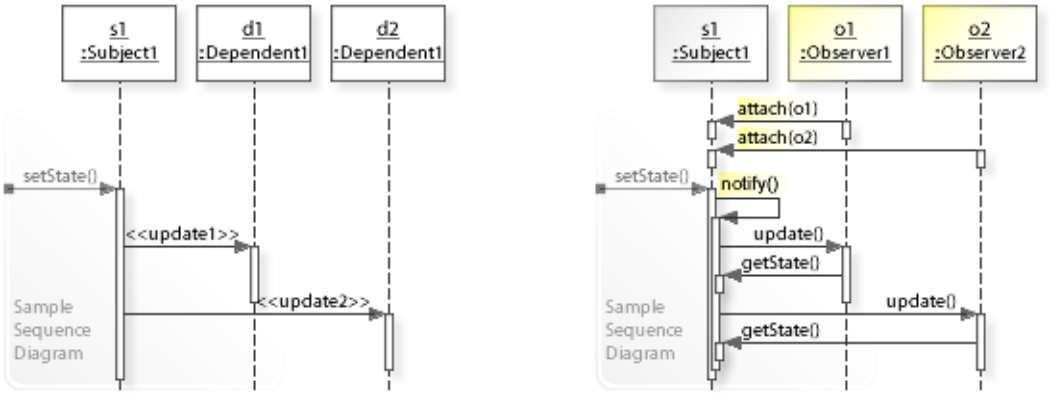
Solution



The Observer design pattern provides a solution:
Define `Subject` and `Observer` objects
so that when a 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* interaction by *notifying* (calling *update* on) all *registered* observers automatically when an event of interest occurs.
- **Define `Subject` and `Observer` objects:**
  - `Subject` defines an interface for registering and unregistering observers (`attach(o)`, `detach(o)`) and for notifying observers (`notify()`), i.e., calling `update()` on all registered observers.
  - `Observer` defines an interface for updating state (`update()`), i.e., synchronizing observer's state with subject's state.
- **When a 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. An open-ended number of observers can observe a subject. New observers can be added to and existing ones can be removed from the subject 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



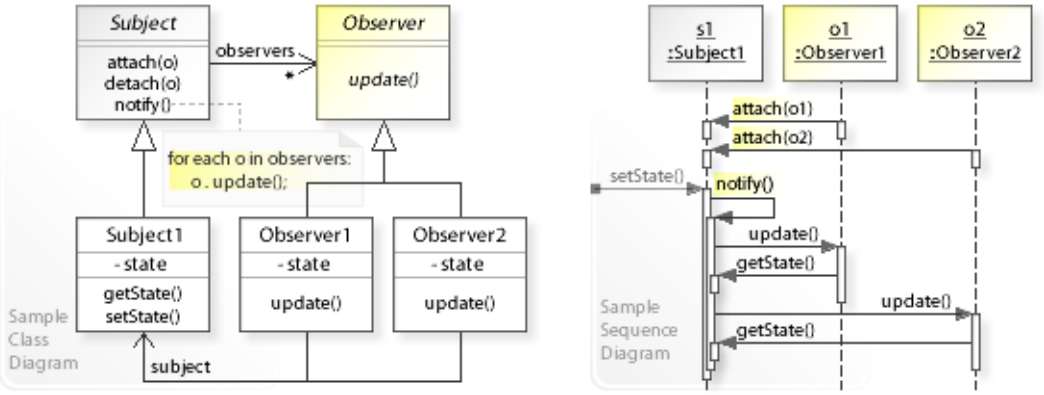
Consider the left design (problem):

- Tight coupling
  between subject and dependents.
  - Subject implements (is responsible for) updating dependent objects.
  - Subject refers to and knows about (how to update) many different objects having different interfaces (tight coupling).
  - Adding new dependent objects or removing existing ones requires changing subject.
  - Tightly coupled objects depend on (refer to and know about) many other objects having different interfaces, which makes the objects hard to implement, change, test, and reuse.

Consider the right design (solution):

- Loose coupling
  between subject and observers.
  - Subject delegates (the responsibility for) updating to dependent objects (observers).
  - Subject only refers to and knows about the common `Observer` interface for updating state (loose coupling).
  - An open-ended number of observers can be added/removed independently and dynamically (`attach(o)/detach(o)`).
  - Loosely coupled objects have only minimal dependencies (by working through a common interface), which makes the objects easier to to implement, change, test, and reuse.

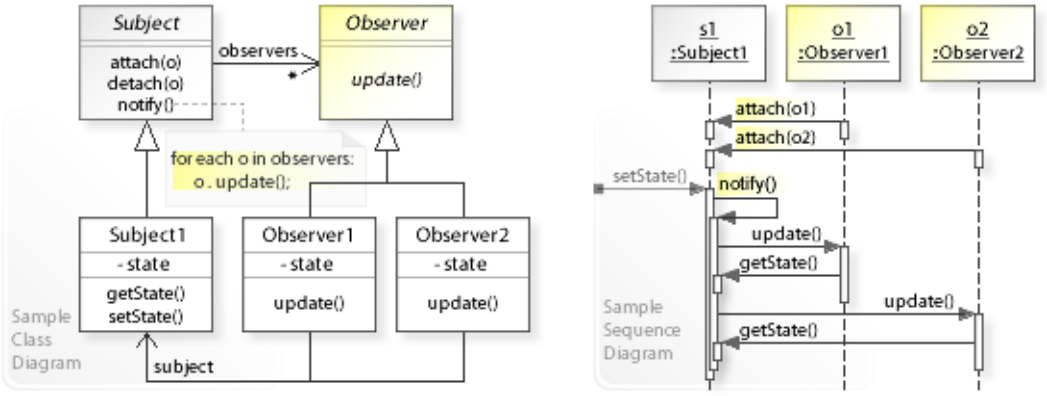
Applicability



Design Problems

- **Defining One-to-many Dependencies Between Objects**
  - 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 dependent objects be maintained?
- **Flexible Notification-Registration (Publish-Subscribe) Interaction**
  - How can an object notify an open-ended number of other objects?
  - How can a publisher notify an open-ended number of subscribers?

Structure, Collaboration



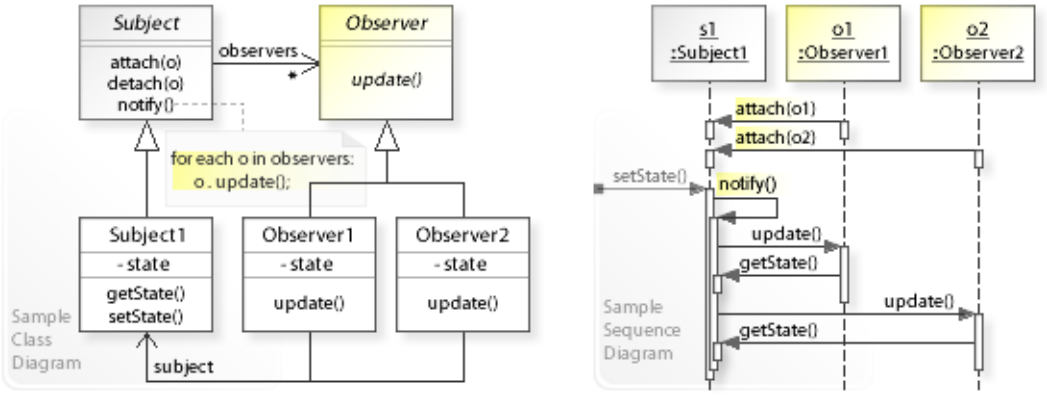
Static Class Structure

- *`Subject`*
  - Refers to the `Observer` interface to update dependent objects (`observers`) and is independent of how the objects are updated.
  - Maintains a list of dependent objects (`observers`).
  - Defines an interface for registering and unregistering observers (`attach(o), detach(o)`).
  - Defines an interface for notifying observers (`notify()`), i.e., calling `update()` on all registered observers: `for each o in observers: o.update()`.
  - Usually, calls `notify()` on itself when its state changes.
- `Subject1`
  - Stores state observers depend on.
- *`Observer`*
  - Defines an interface for updating state (`update()`), i.e., synchronizing observer's state with subject's state.
- `Observer1,Observer2,,...`
  - Dependent objects that implement the `Observer` interface.
  - Store state that should stay consistent with subject's state.
  - Maintain a reference (`subject`) to the subject they observe to get the changed data (`getState`).

Dynamic Object Collaboration

- In this sample scenario, `Observer1` and `Observer2` objects register themselves on a `Subject1` object and are subsequently notified and updated when `Subject1` changes state.
- The interaction starts with the `Observer1` and `Observer2` objects that call `attach(this)` on `Subject1` to register themselves.
- Thereafter, let's assume that the state of `Subject1` changes, `Subject1` calls `notify()` on itself.
- `notify()` calls `update()` on the registered `Observer1` and `Observer2` objects, which in turn get the changed data (`getState()`) from `Subject1` to 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



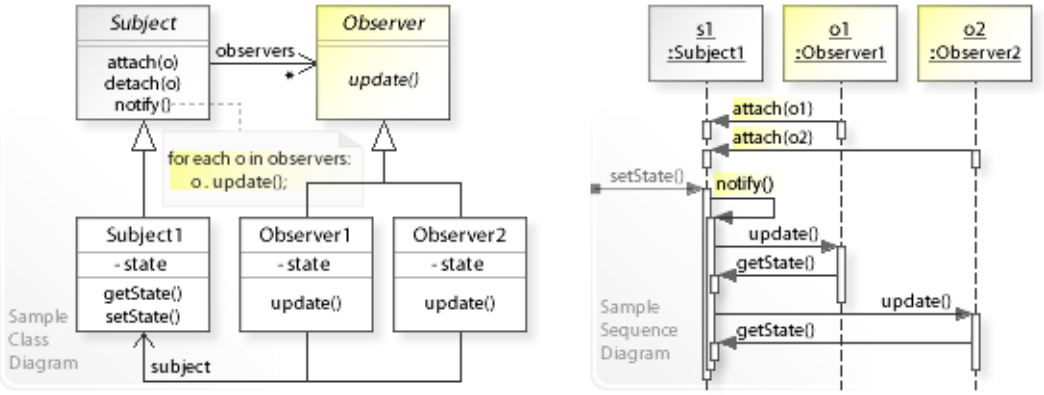
Advantages (+)

- Decouples subject from observers.
  - Subject only refers to and knows about the simple `Observer` interface for updating (synchronizing) 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, test, 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.

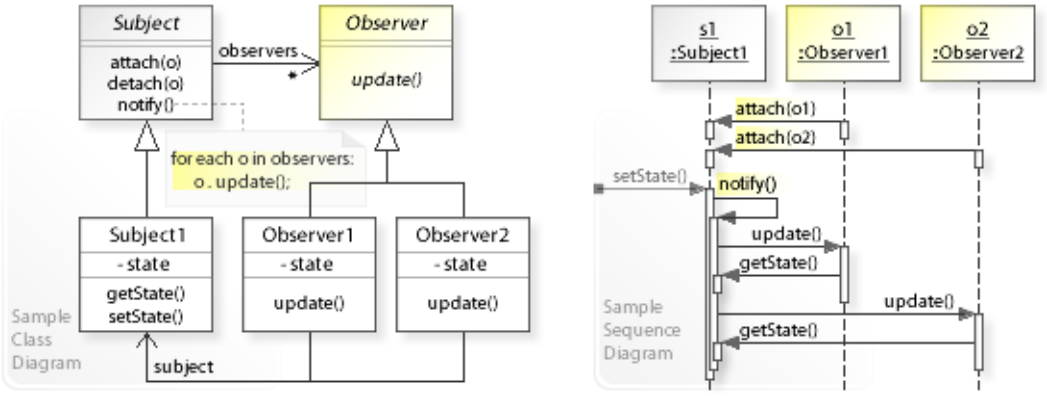
Implementation



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 MyApp {
 3      public static void main(String[] args) {
 4          Subject1 s1 = new Subject1();
 5          // Creating observers and registering them on subject1.
 6          Observer o1 = new Observer1(s1);
 7          Observer o2 = new Observer2(s1);
 8
 9          System.out.println("Changing state of Subject1 ...");
10          s1.setState(100);
11      }
12  }
```

```
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  }
```

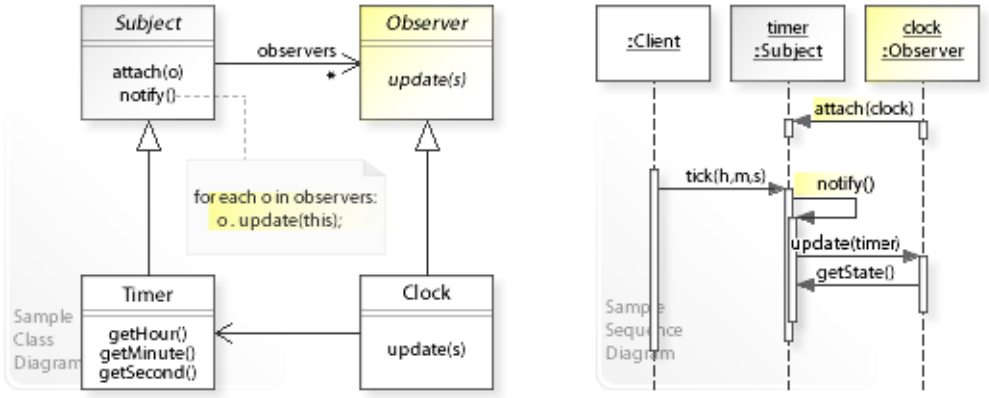
```
 1  package com.sample.observer.basic;
 2  public class Subject1 extends Subject {
 3      private int state = 0;
 4      //
 5      public int getState() {
 6          return state;
 7      }
 8      void setState(int state) {
 9          this.state = state;
10          System.out.println(
11              "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  :  20:20:38
Clock  : Updated/Synchronized to :  20:20:38
Timer  : Time of day changed to  :  20:20:39
Clock  : Updated/Synchronized to :  20:20:39
Timer  : Time of day changed to  :  20:20:40
Clock  : Updated/Synchronized to :  20:20:40
```

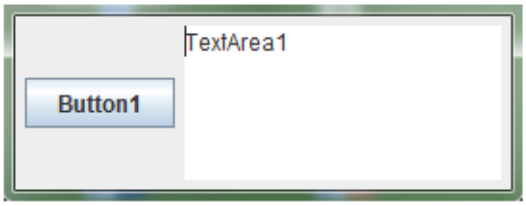
```
 1  package com.sample.observer.timer;
 2  import java.util.ArrayList;
 3  import java.util.List;
 4  public abstract class Subject {
 5      private List<Observer> observers = new ArrayList<Observer>();
 6      // Registration interface.
 7      public void attach(Observer o) {
 8          observers.add(o);
 9      }
10      // Notification interface.
11      public void notifyObservers() {
12          for (Observer o : observers)
13              o.update(this);
14      }
15  }
```

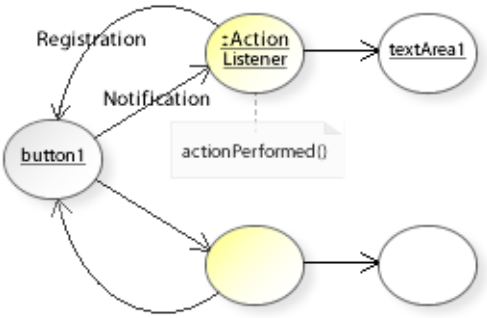
```
package com.sample.observer.timer;
public class Timer extends Subject {
    private int hour = 0;
    private int minute = 0;
    private int second = 0;
    public int getHour() {
        return hour;
    }
    public int getMinute() {
        return minute;
    }
    public int getSecond() {
        return second;
    }
    // Changing time of day and notifying observers.
    public void tick(int hour, int minute, int second) {
        System.out.printf(
            "Timer  : Time of day changed to  :  %02d:%02d:%02d %n",
            hour, minute, second);
        this.hour = hour;
        this.minute = minute;
        this.second = second;
        // Notifying observers that time has changed.
        notifyObservers();
    }
}
```

```
package com.sample.observer.timer;
public abstract class Observer {
    public abstract void update(Subject s);
}
```

```
package com.sample.observer.timer;
public class Clock extends Observer {
    private Timer subject;
    public Clock(Timer subject) {
        this.subject = subject;
        // Registering this clock on subject.
        subject.attach(this);
    }
    public void update(Subject s) {
        if (this.subject == s) {
            System.out.printf(
                "Clock  : Updated/Synchronized to :  %02d:%02d:%02d %n",
                subject.getHour(), subject.getMinute(), subject.getSecond());
        }
    }
}
```

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.
************************************************************************
```

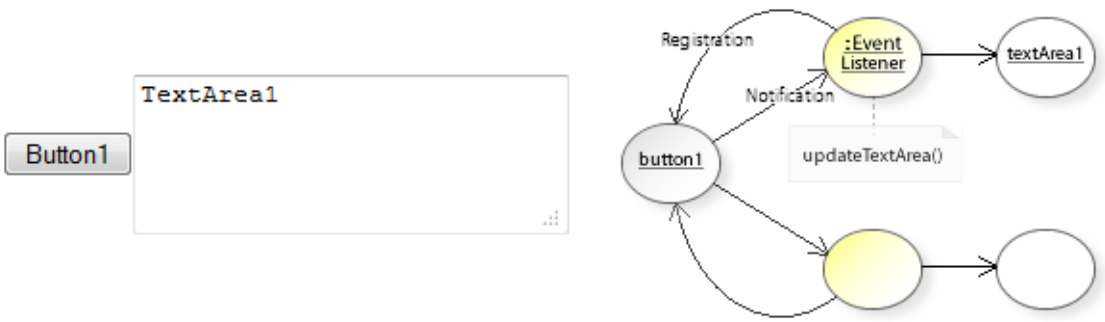
```
 1  package com.sample.observer.gui;
 2  import java.awt.event.*;
 3  import javax.swing.*;
 4  public class GUIDemo1 extends JPanel {
 5      JButton button1;
 6      JTextArea textArea1;
 7      public GUIDemo1() {
 8          button1 = new JButton("Button1");
 9          add(button1);
10          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 / JavaScipt).

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("textArea1");
 7              buttonNode.innerHTML += "\nNotification from Button1: \n  " +
 8                      "User clicked the Button1.";
 9          }
10      </script>
11  </head>
12
13  <body>
14      <button id="button1" onclick="updateTextArea();">Button1</button>
15  <!--                     ========================== -->
16      <textarea id="textArea1" rows="4" cols="25">TextArea1</textarea>
17  </body>
18  </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() {
```

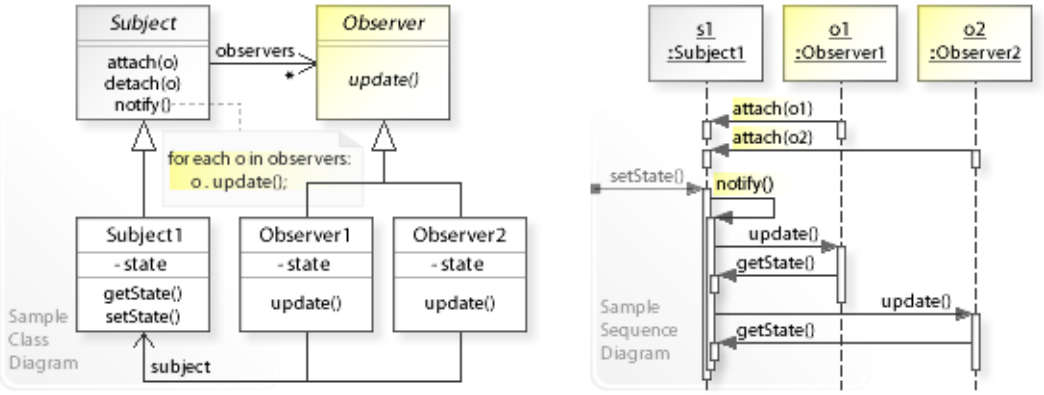
```
            var buttonNode = document.getElementById('button1');
            if (buttonNode.addEventListener) {
                buttonNode.addEventListener('click', updateTextArea, false);
<!--            ========================================================== -->
            }  else {  // Internet Explorer 6-8
                buttonNode.attachEvent('onclick', updateTextArea);
            }
        }
        function updateTextArea() {
            var textNode = document.getElementById("textArea1");
            textNode.innerHTML += "\nNotification from Button1: \n  " +
                    "User clicked the Button1.";
        } ;
    </script>
</head>

<body onload="dynamicRegistration();">
    <button id="button1">Button1</button>
    <textarea id="textArea1" rows="4" cols="25">TextArea1</textarea>
</body>
</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.
****

```
<!DOCTYPE html>
<html>
<head>
    <script type="text/javascript">
        function dynamicRegistration() {
            var buttonNode = document.getElementById('button1');
            buttonNode.onclick = function() {
<!--        ============================== -->
                var textNode = document.getElementById("textArea1");
                textNode.innerHTML += "\nNotification from Button1: \n  " +
                        "User clicked the Button1.";
            }
        } ;
    </script>
</head>

<body onload="dynamicRegistration();">
    <button id="button1">Button1</button>
    <textarea id="textArea1" rows="4" cols="25">TextArea1</textarea>
</body>
</html>
```

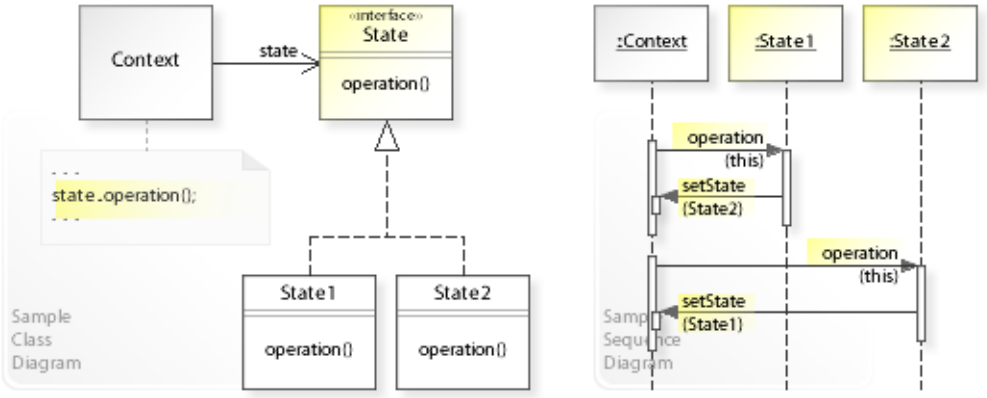
Related Patterns



Key Relationships

- **Mediator - Observer**
  - Mediator provides a way to keep interacting objects loosely coupled by defining a `Mediator` object that centralizes (encapsulates) interaction behavior.
  - Observer provides a way to keep interacting objects loosely coupled by defining `Subject` and `Observer` objects that distribute interaction behavior so that when a subject changes state all registered observers are updated.
  - "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]

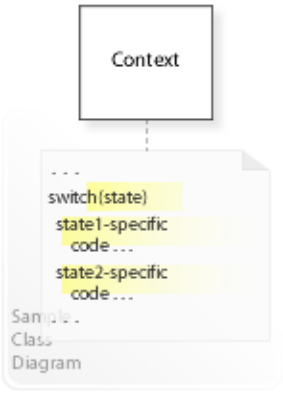
Intent



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 an object alter its behavior when its internal state changes?*
  - *How can state-specific behavior be defined so that new states can be added and the behavior of existing states can be changed independently?*
- 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.

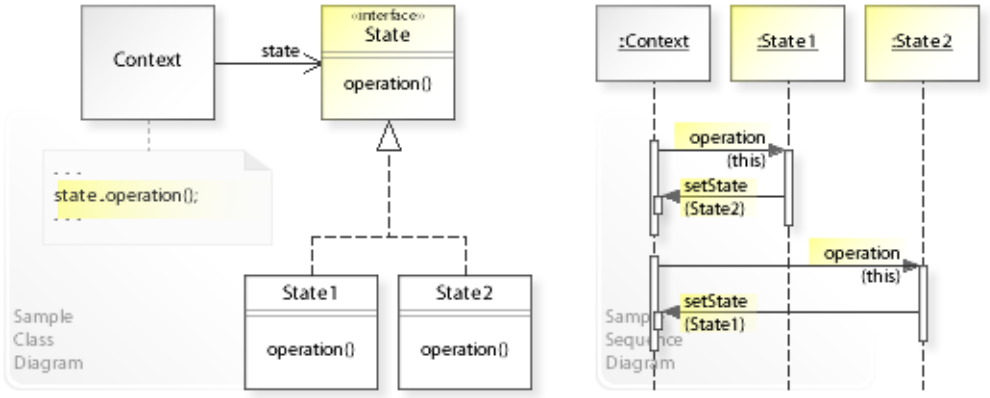
Problem



The State design pattern solves problems like:
How can an object alter its behavior when its internal state changes?
How can state-specific behavior be defined
so that new states can be added and the behavior of existing states can be changed independently?
See Applicability section for all problems State can solve. See Solution section for how State solves the problems.

- An inflexible way is to implement (hard-wire) state-specific behavior directly within a class (`Context`) that depends on its internal state. Conditional statements (`switch(state)`) are required that depend on this state. Each conditional branch implements the corresponding state-specific behavior.
- This commits the class to a particular state-specific behavior and makes it impossible to add new states or change the behavior of existing states later independently from (without having to change) the class.
  Classes that include state-specific behavior are harder to implement, change, test, and reuse. "[…] 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 independently.*
- 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.

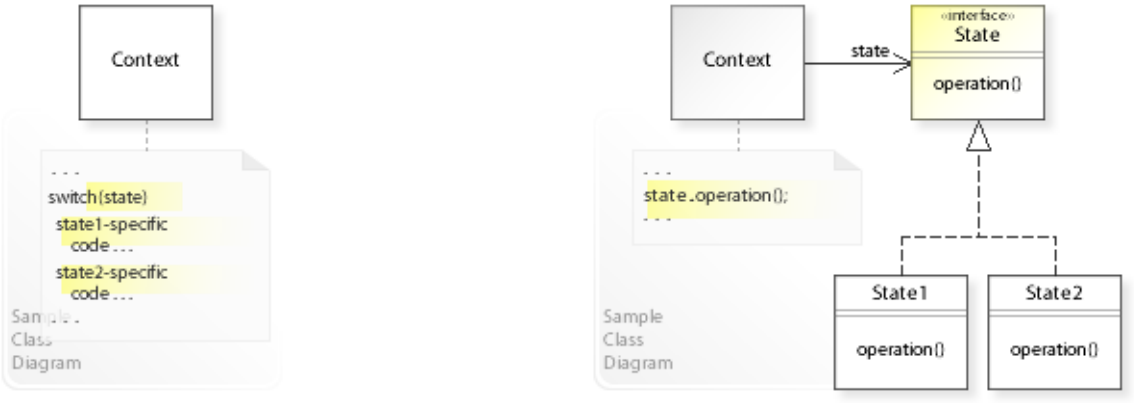
Solution



The State design pattern provides a solution:
Encapsulate state-specific behavior in a separate `State` object.
A class delegates state-specific behavior to its current `State` object
instead of implementing state-specific behavior directly.
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 an object's state-specific behavior in a separate `State` 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 for each state.
- 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]
- **A class (`Context`) delegates the responsibility for performing state-specific behavior to its current `State` object** (`state.operation(…)`).
- This enables *run-time* flexibility (via object composition).
  A class can change its behavior at run-time by changing its current `State` object.
  Usually, the `State` objects are responsible for changing `Context`'s current state at run-time when a state transition occurs (see also Collaboration and Sample Code).

Motivation 1



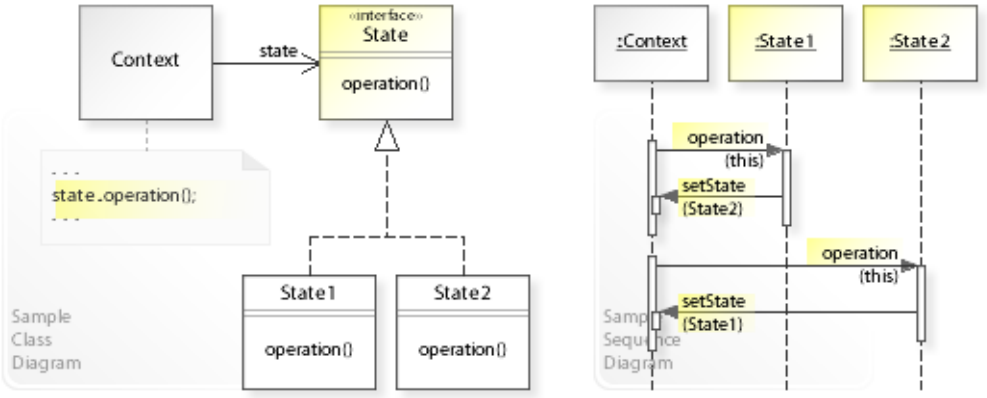
Consider the left design (problem):

- Hard-wired state-specific behavior.
  - The behavior of different states is implemented (hard-wired) directly within a class (`Context`).
  - This makes it impossible 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.
  - Classes that include state-specific behavior get more complex and harder to implement, change, test, and reuse.

Consider the right design (solution):

- Encapsulated state-specific behavior.
  - The behavior for 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.
  - Classes that delegate state-specific behavior get less complex and easier to implement, change, test, and reuse.

Applicability



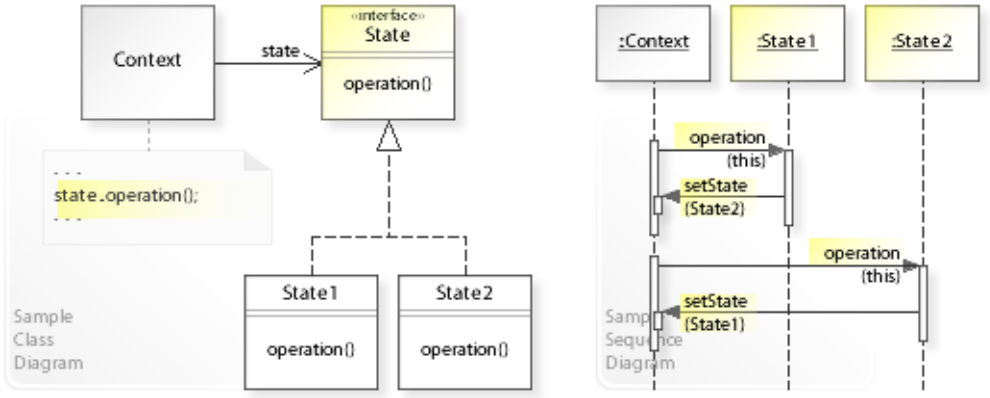
Design Problems

- **Defining State-Specific Behavior**
  - How can an object alter its behavior when its internal state changes?
  - How can state-specific behavior be defined so that new states can be added and the behavior of existing states can be changed independently?
  - How can conditional statements that depend on an object's internal state be avoided?

Refactoring Problems

- **Complicated Code**
  - How can conditional statements that depend on an object's internal state be eliminated? *Replace State-Altering Conditionals with State (166)* [JKerievsky05]

Structure, Collaboration



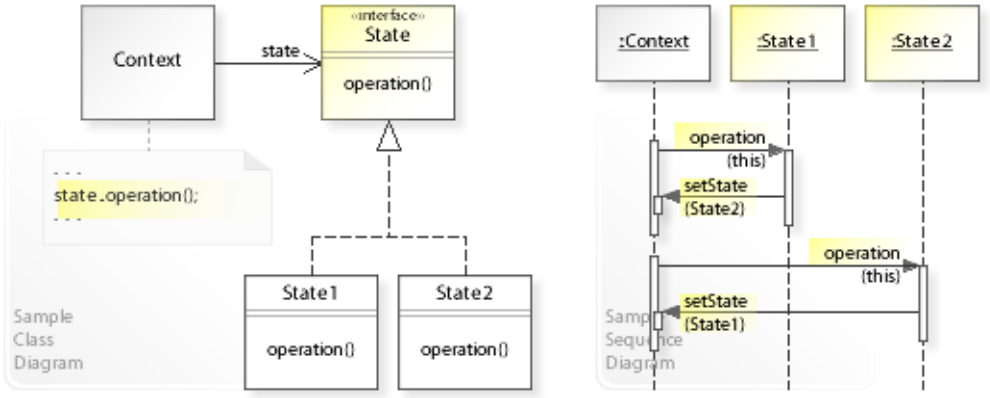
Static Class Structure

- `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 for each state.

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 the `Context` object that calls `operation(this)` on its current state object (`State1`).
- `Context` passes itself (`this`) to `State1` so that `State1` can call back and change context's current state object.
- `State1` performs the state1-specific operation and, assuming that a state transition occurs, changes context's current state object to `State2` by calling `setState(State2)` on `Context`.
- Thereafter, the `Context` object again calls `operation(this)` on its current state object (`State2`).
- `State2` performs the state2-specific operation and calls `setState(State1)` on `Context` to switch to `State1`.
- See also Sample Code / Example 1.

Consequences



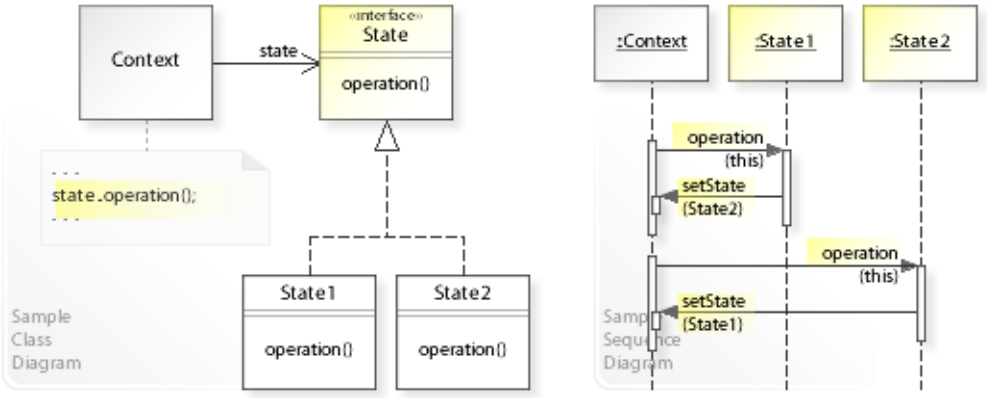
Advantages (+)

- 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 for switching between states.
  - 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 change `Context`'s state.
- Introduces an additional level of indirection.
  - State achieves flexibility by introducing an additional level of indirection (clients delegate to separate `State` objects), which makes clients dependent on a `State` object.

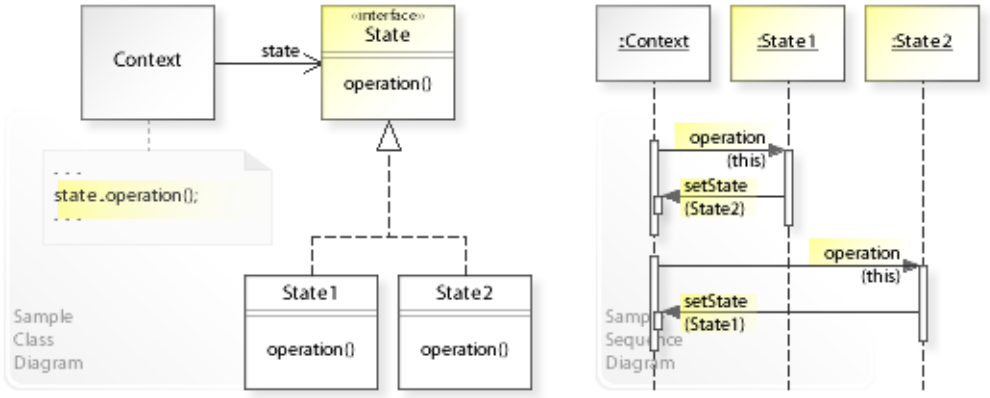
Implementation



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 states.
    The `State` objects define the state transitions and state-specific operations.
    This makes `Context` easier to implement, change, test, and reuse.

Sample Code 1



Basic Java code for implementing the sample UML diagrams.

```
 1  package com.sample.state.basic;
 2  public class MyApp {
 3      public static void main(String[] args) {
 4          // Creating a Context object
 5          // and configuring it with the initial State1 object.
 6          Context context = new Context(State1.getInstance());
 7          // Calling an operation on context.
 8          System.out.println("(1) " + context.operation());
 9          // Calling the operation again.
10          System.out.println("(2) " + context.operation());
11      }
12  }

(1) Context: Delegating state-specific behavior to the current State object.
    State1 : Hello World1! Changing current state of Context to State2.
(2) Context: Delegating state-specific behavior to the current State object.
    State2 : Hello World2! Changing current state of Context to State1.
```

```
 1  package com.sample.state.basic;
 2  public class Context {
 3      private State state; // reference to the current State object
 4
 5      public Context(State state) {
 6          this.state = state;
 7      }
 8      public String operation() {
 9          return "Context: Delegating state-specific behavior to the current State object.\n"
10                  + state.operation(this);
11      }
12      void setState(State state) {  // package private
13          this.state = state;
14      }
15  }
```

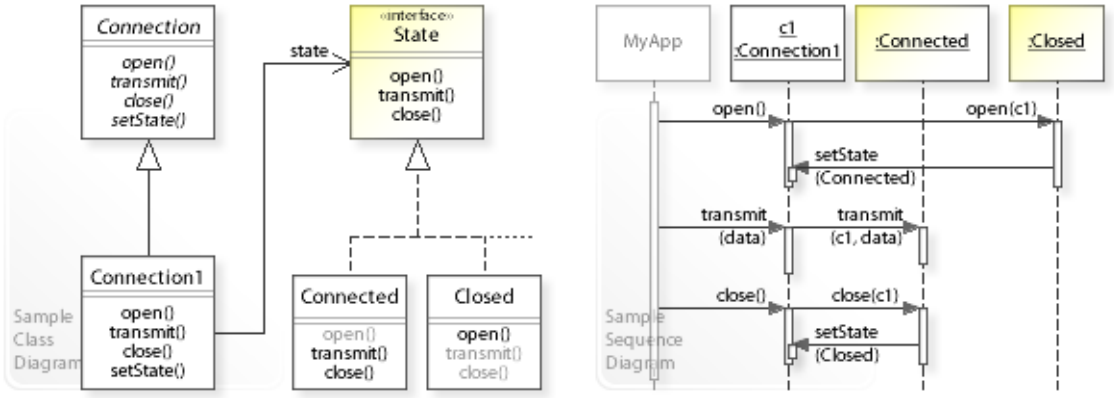
```
 1  package com.sample.state.basic;
 2  public interface State {
 3      String operation(Context context);
 4  }
```

```
 1  package com.sample.state.basic;
 2  public class State1 implements State {
 3      // Implemented as Singleton.
 4      private static final State1 INSTANCE = new State1();
 5      private State1() { }
 6      public static State1 getInstance() {
 7          return INSTANCE;
 8      }
 9      public String operation(Context context) {
10          String result = "    State1 : Hello World1!" +
11              " Changing current state of Context to State2.";
12          context.setState(State2.getInstance()); // state transition
13          return result;
```

```
    }
}
```

```
package com.sample.state.basic;
public class State2 implements State {
    // Implemented as Singleton.
    private static final State2 INSTANCE = new State2();
    private State2() { }
    public static State2 getInstance() {
        return INSTANCE;
    }
    public String operation(Context context) {
        String result = "    State2 : Hello World2!" +
            " Changing current state of Context to State1.";
        context.setState(State1.getInstance()); // state transition
        return result;
    }
}
```

Sample Code 2



Network communication states (Connected / Closed).

```
 1  package com.sample.state.tcp;
 2  import java.io.OutputStream;
 3  public class MyApp {
 4      public static void main(String[] args) {
 5          OutputStream data = null;
 6          Connection connection = new Connection1(Closed.getInstance());
 7          connection.open();
 8          // ...
 9          connection.transmit(data);
10          // ...
11          connection.close();
12      }
13  }
```

```
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  }
```

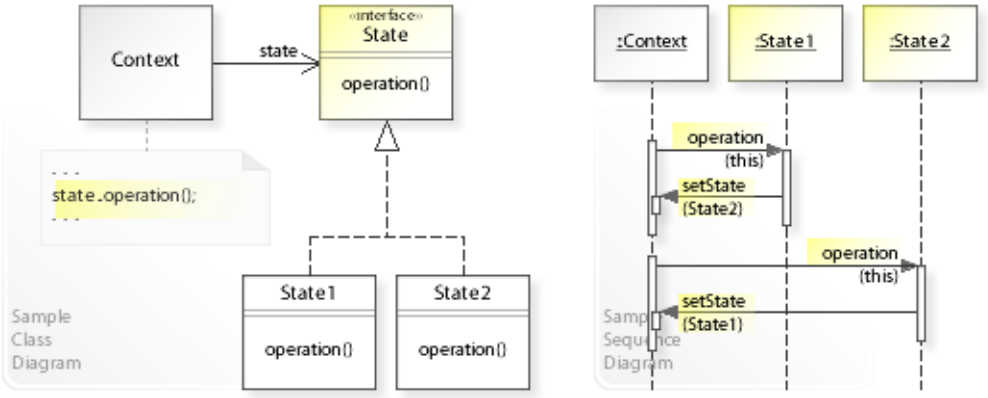
```
 1  package com.sample.state.tcp;
 2  import java.io.OutputStream;
 3  public interface State {
```

```
    void open(Connection c);
    void transmit(Connection c, OutputStream data);
    void close(Connection c);
}
```

```
package com.sample.state.tcp;
import java.io.OutputStream;
public class Connected implements State {
    // Implemented as Singleton.
    private static final Connected INSTANCE = new Connected();
    private Connected() { }
    public static Connected getInstance() {
        return INSTANCE;
    }
    //
    public void open(Connection c) {
        System.out.println(
            "State CONNECTED: *** Can't open connection " +
                "(connection already opened). ***");
        System.exit(-1);
    }
    public void transmit(Connection c, OutputStream data) {
        // ...
        System.out.println(
            "State CONNECTED: Transmitting data ... Finished.");
    }
    public void close(Connection c) {
        // ...
        c.setState(Closed.getInstance());
        System.out.println(
            "State changed from CONNECTED to CLOSED.");
    }
}
```

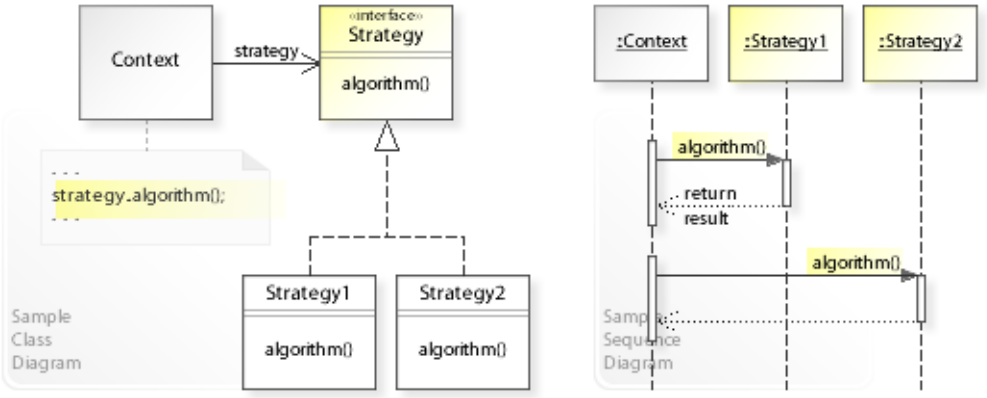
```
package com.sample.state.tcp;
import java.io.OutputStream;

import com.sample.state.basic.State1;
public class Closed implements State {
    // Implemented as Singleton.
    private static final Closed INSTANCE = new Closed();
    private Closed() { }
    public static Closed getInstance() {
        return INSTANCE;
    }
    //
    public void open(Connection c) {
        // ...
        c.setState(Connected.getInstance());
        System.out.println(
            "State changed from CLOSED to CONNECTED.");
    }
    public void transmit(Connection c, OutputStream data) {
        System.out.println(
            "State CLOSED: *** Can't transmit data " +
                "(connection is closed). ***");
        System.exit(-1);
    }
    public void close(Connection c) {
        System.out.println(
            "State CLOSED: *** Can't close connection " +
                "(connection already closed). ***");
        System.exit(-1);
    }
}
```

Related Patterns



Key Relationships

Intent



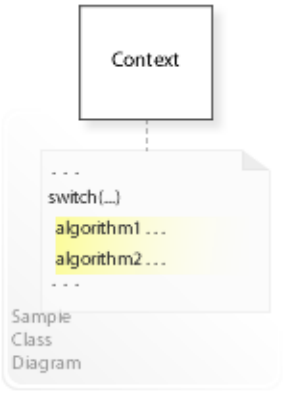
The intent of the Strategy design pattern is to:
"Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it." [GoF]
See Problem and Solution sections for a more structured description of the intent.

- The Strategy design pattern solves problems like:
  - *How can a class be configured with an algorithm at run-time instead of implementing an algorithm directly?*
  - *How can an algorithm be selected and exchanged 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 an *arbitrary piece of code* that does something appropriate.
- 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 (pricing 'strategy')* 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) each algorithm,
  - *and make them interchangeable* - and define a common interface (`Strategy`) through which algorithms can be (inter)changed 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.*

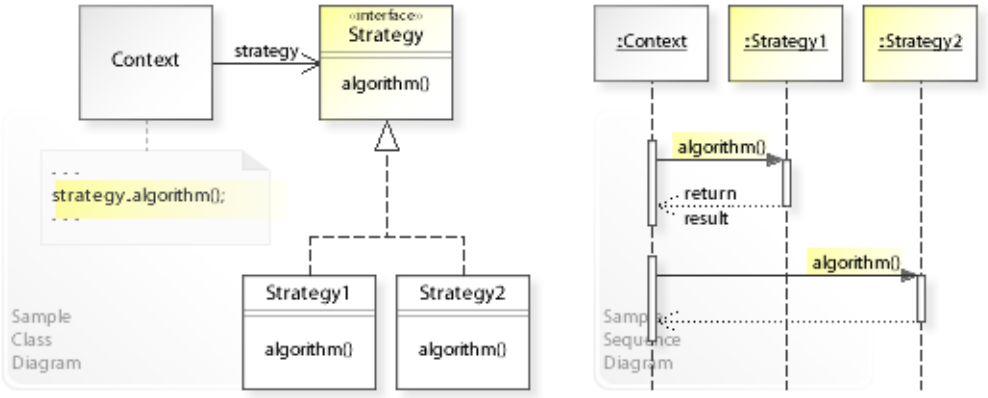
Problem



The Strategy design pattern solves problems like:
How can a class be configured with an algorithm at run-time
instead of implementing an algorithm directly?
How can an algorithm be selected and exchanged at run-time?
See Applicability section for all problems Strategy can solve. See Solution section for how Strategy solves the problems.

- An inflexible way is to implement (hard-wire) an algorithm directly within the class (`Context`) that requires (uses) the algorithm. Conditional statements (`switch(…)`) are needed to switch between different algorithms.
- This commits (couples) the class to particular algorithms at compile-time and makes it impossible to change an algorithm later independently from (without having to change) the class. It makes the class more complex, especially if multiple algorithms are needed, and stops the class from being reusable if other algorithms are required.
  "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 configure a class with an algorithm at run-time.*
- For example, reusable classes that support different algorithms.
  A reusable class should avoid implementing algorithms directly so that it can be configured with an algorithm at run-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 select the right *pricing algorithm* at run-time (see Sample Code / Example 2).
- For example, sorting objects.
  To sort objects in different ways, it should be possible to parameterize a sort operation with a *compare algorithm* (see Sample Code / Example 3).

Solution



The Strategy design pattern provides a solution:
Encapsulate an algorithm in a separate `Strategy` object.
A class delegates an algorithm to a `Strategy` object
instead of implementing an algorithm 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 implement algorithms in a separate inheritance hierarchy so that they can vary independently.
- **Define separate `Strategy` objects:**
  - For all supported algorithms, define a common interface for performing an algorithm (`Strategy | algorithm(…)`).
  - Define classes (`Strategy1`,`Strategy2`,…) that implement the `Strategy` interface (encapsulate an algorithm).
- This enables *compile-time* flexibility (via inheritance).
  New algorithms can be added and existing ones can be changed independently by defining new (sub)classes.
- **A class (`Context`) delegates the responsibility for performing an algorithm to a `Strategy` object** (`strategy.algorithm(…)`).
- This enables *run-time* flexibility (via object composition).
  A class can be configured with a `Strategy` object, which it uses to perform an algorithm, and even more, the `Strategy` object can be exchanged dynamically.

Background Information

- Encapsulation is "The result of hiding a representation and implementation in an object." [GoF, p360]
- As a reminder, an object has an *outside view* (public interface/operations) and an *inside view* (private/hidden 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 any changes of an object's representation and implementation.
  That's the essential benefit of encapsulation.
  See also Design Principles.
- Terms and Definitions
  - 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*.
  - "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

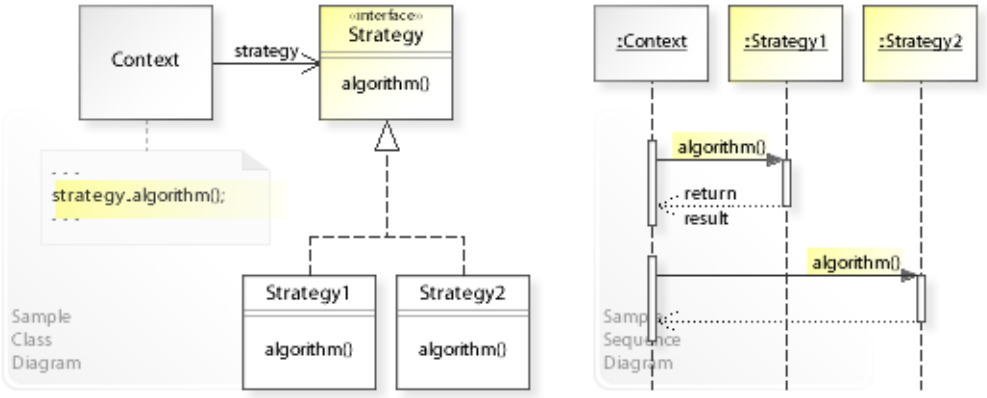
Consider the left design (problem):

- Hard-wired algorithms.
  - Different algorithms are implemented (hard-wired) directly within a class (`Context`).
  - This makes it impossible 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 include multiple algorithms get more complex and harder to implement, change, 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 delegate an algorithm get less complex and easier to implement, change, test, and reuse.

Applicability



Design Problems

- **Exchanging Algorithms at Run-Time**
  - How can a class be configured with an algorithm at run-time instead of implementing an algorithm directly?
  - How can an algorithm be selected and exchanged at run-time?
- **Flexible Alternative to Subclassing**
  - How can a flexible alternative be provided to subclassing for changing an algorithm at compile-time?

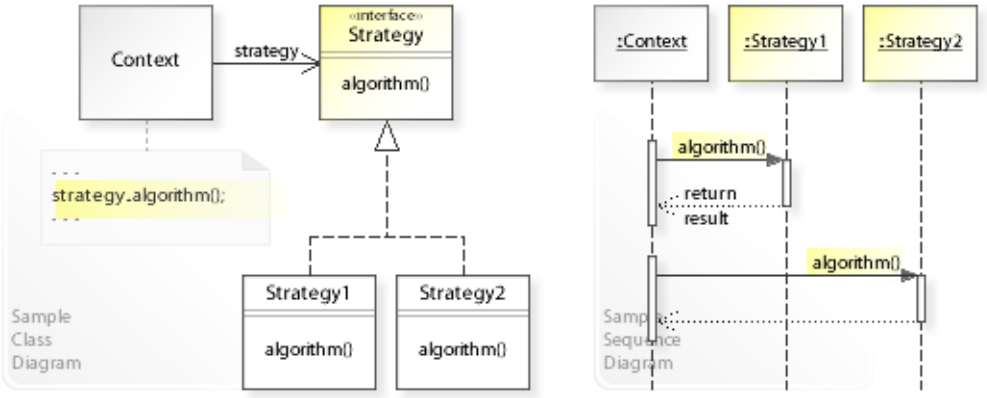
Refactoring Problems

- **Inflexible Code**
  - How can hard-wired algorithms (compile-time implementation dependencies) be refactored?
- **Duplicated Code**
  - How can algorithms that are duplicated in multiple places be refactored?
  - How can many related classes that differ only in their algorithms be replaced by a common class that is configured with one of many algorithms?
- **Complicated Code**
  - How can 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



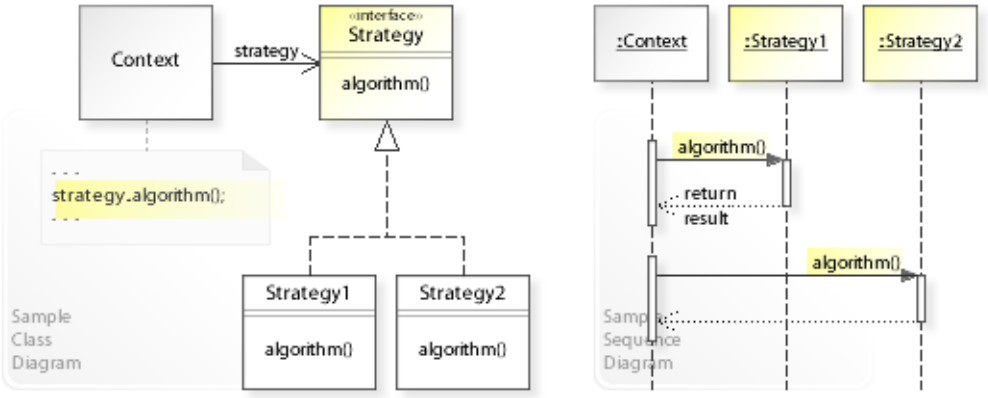
Static Class Structure

- `Context`
  - Refers to the `Strategy` interface to perform an algorithm (`strategy.algorithm()`) and is independent of how the algorithm is implemented.
  - Maintains a reference (`strategy`) to a `Strategy` object.
- `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 performing 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 calls `algorithm()` on its installed `Strategy1` object.
- `Strategy1` performs the algorithm and returns the result to `Context`.
- Let's assume that `Context` changes its strategy to `Strategy2` (because of run-time conditions such as reaching a threshold, for example).
- `Context` now calls `algorithm()` on the `Strategy2` object, which performs the algorithm and returns the result to `Context`.
- There are different ways to select and change a strategy. For example, clients of `Context` might change the strategy or pass the strategy to the `Context`.
- See also Sample Code / Example 1.

Consequences



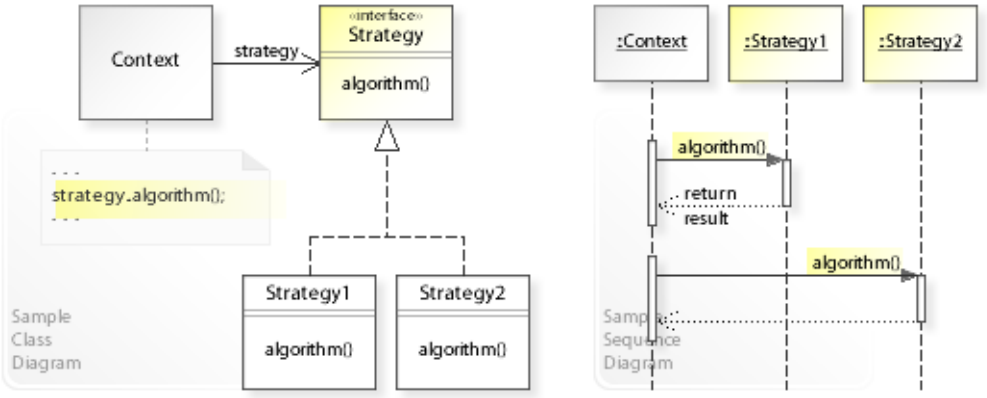
Advantages (+)

- Avoids compile-time 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 (clients delegate an algorithm to a separate `Strategy` object), which makes clients 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]

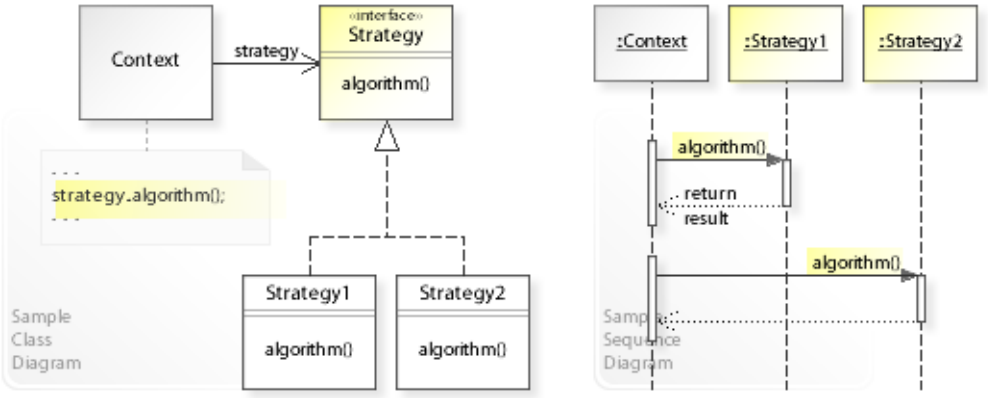
Implementation



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          // Calling an operation on context.
 8          System.out.println("(1) " + context.operation());
 9          // Changing context's strategy.
10          context.setStrategy(new Strategy2());
11          System.out.println("(2) " + context.operation());
12      }
13  }
```

```
(1) Context: Delegating an algorithm to a strategy: Result = 1
(2) Context: Delegating an algorithm to a strategy: Result = 2
```

```
 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          return "Context: Delegating an algorithm to a strategy: Result = "
10                  + strategy.algorithm();
11      }
12      public void setStrategy(Strategy strategy) {
13          this.strategy = strategy;
14      }
15  }
```

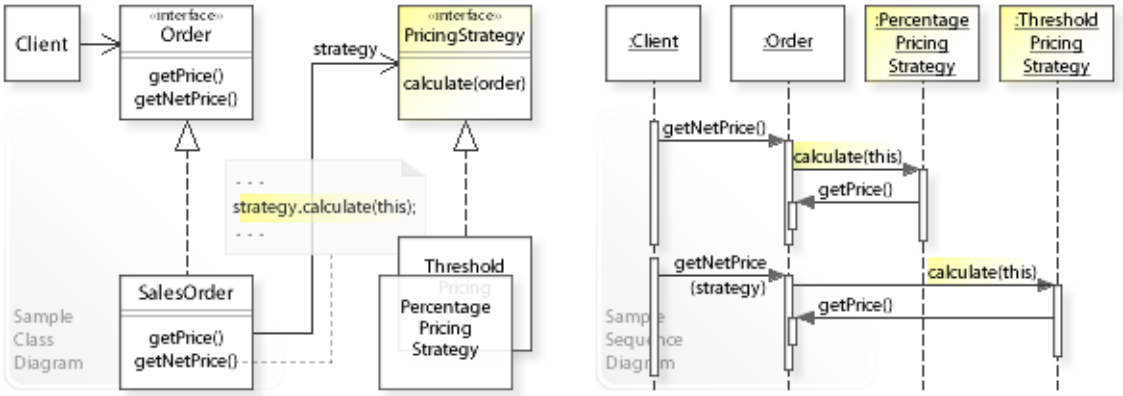
```
 1  package com.sample.strategy.basic;
 2  public interface Strategy {
 3      int algorithm();
 4  }
```

```
 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 object
 7          // and configuring it with a (default) pricing strategy.
 8          Order order = new SalesOrder(new PercentagePricingStrategy());
 9          // Creating products and order lines.
10          Product product1A = new SalesProduct("1A", "01", "Product1A", 100);
11          Product product1B = new SalesProduct("1B", "01", "Product1B", 200);
12          order.createOrderLine(product1A, 1);
13          order.createOrderLine(product1B, 1);
14
15          System.out.println(
16              "(1) Total order brutto price ......................... : " +
17                  order.getPrice());
18          System.out.println(
19              "(2) using the default percentage strategy (10%) ....... : " +
20                  order.getNetPrice());
21          System.out.println(
22              "(3) changing to threshold strategy (10%; above 200: 20%): " +
23                  order.getNetPrice(new ThresholdPricingStrategy()));
24      }
25  }
```

```
(1) Total order brutto price ......................... : 300
(2) using the default percentage strategy (10%) ....... : 270
(3) changing to 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.
**********************************************************************
```

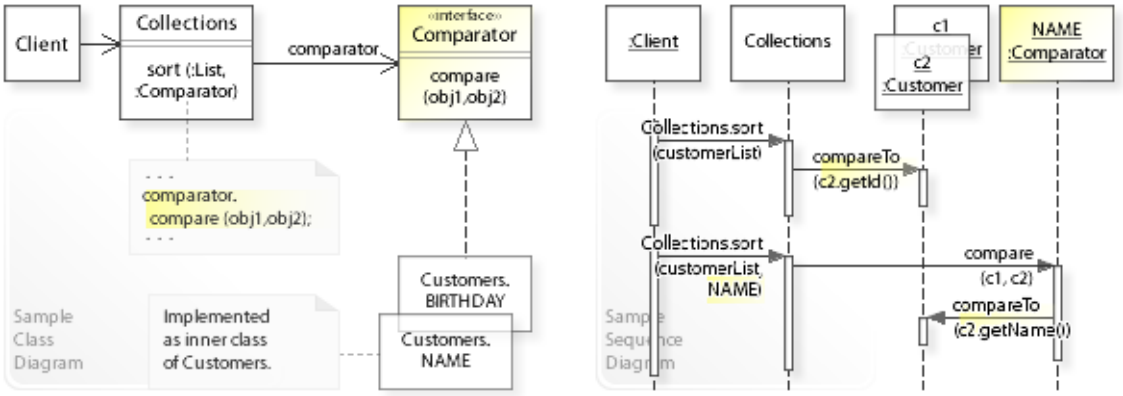
```
 1  package com.sample.data;
 2  public interface OrderLine {
 3      Product getProduct();
 4      int getQuantity();
 5      long getPrice();
 6  }
```

```
 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());
```

```
            }
        } ;
    // PHONENUMBER = reference to a comparator object.
    public static final Comparator<Customer> PHONENUMBER =
        new Comparator<Customer>() {  // inner class
            // Implementing the comparator interface / compare().
            public int compare(Customer c1, Customer c2) {
                // getPhoneNumber() returns an object of type PhoneNumber;
                // compareTo() implemented in the PhoneNumber class.
                return c1.getPhoneNumber().compareTo(c2.getPhoneNumber());
            }
        } ;
    // BIRTHDAY = reference to a comparator object.
    public static final Comparator<Customer> BIRTHDAY =
        new Comparator<Customer>() {  // inner class
            // Implementing the comparator interface / compare().
            public int compare(Customer c1, Customer c2) {
                // compareTo() implemented in the Date class (Java platform).
                return c2.getBirthday().compareTo(c1.getBirthday());
            }
        } ;
    //
    public static void checkData(int id, Name name, PhoneNumber pn, Date birthday) throws Parse
        if (id < 0)
            throw new IllegalArgumentException("Customer ID is negative");
        if (birthday.compareTo(dateFormatter.parse("01.01.1900")) < 0 ||
            birthday.compareTo(dateFormatter.parse("01.01.2000")) > 0)
            throw new IllegalArgumentException("Birthday before 1900 or after 2000");
        // ...
    }
    public static List<Customer> createTestData(int size) throws Exception {
        List<Customer> customerList = new ArrayList<Customer>(size);
        customerList.add(new Customer1(1,
            new Name("FirstName1", " LastName1"),
            new PhoneNumber(1, 2, 2), dateFormatter.parse("03.03.1980")));
        customerList.add(new Customer1(2,
            new Name("FirstName3", " LastName3"),
            new PhoneNumber(1, 3, 3), dateFormatter.parse("01.01.1970")));
        customerList.add(new Customer1(3,
            new Name("FirstName2", " LastName2"),
            new PhoneNumber(2, 1, 1), dateFormatter.parse("02.02.1980")));
        return customerList;
    }
}
```

```
package com.sample.data;
import java.util.Date;
public interface Customer extends Comparable<Customer> {
    int getId();
    Name getName();
    PhoneNumber getPhoneNumber();
    Date getBirthday();
}
```

```
package com.sample.data;
import java.text.SimpleDateFormat;
// Skeletal implementation of the customer interface.
import java.util.Date;
public abstract class AbstractCustomer implements Customer {
    private final int id;
    private final Name name;
    private final PhoneNumber phoneNumber;
    private final Date birthday;
    private static final SimpleDateFormat dateFormatter =
        new SimpleDateFormat("dd.MM.yyyy");
    // TODO Check
    protected AbstractCustomer(int id, Name name, PhoneNumber pn, Date birthday) throws Excepti
        Customers.checkData(id, name, pn, birthday);
        this.id = id;
        this.name = name;
        this.phoneNumber = pn;
        this.birthday = birthday;
    }
    @Override
    public boolean equals(Object o) {
        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.
******************************************************
```

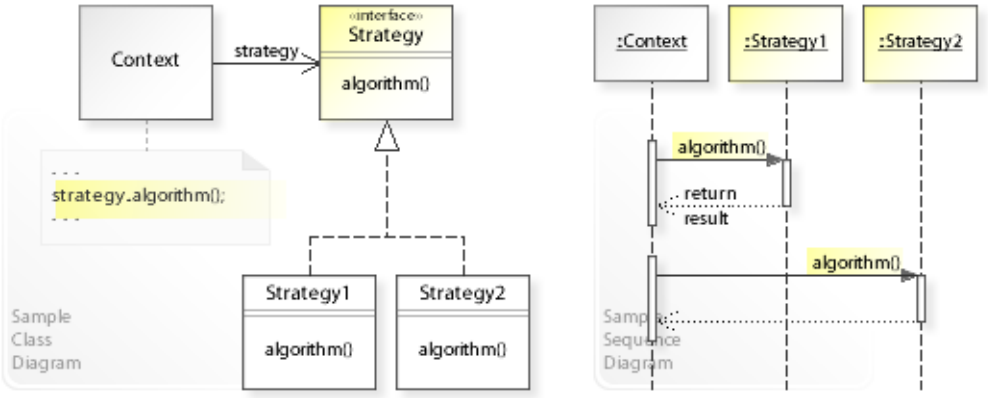
```
1  package com.sample.data;
2  public class Name implements Comparable<Name> {
3      private final String firstName;
4      private final String lastName;
5      // ...
6      public Name(String firstName, String lastName) {
7          if (firstName == null || lastName == null)
8              throw 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

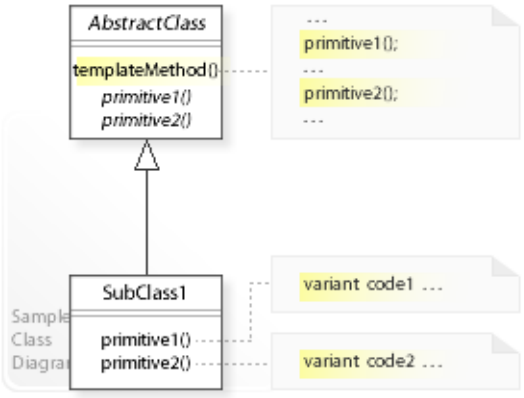


Key Relationships

- **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 change certain parts of 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 performing an algorithm to a strategy object.
  - Abstract Factory
    A class delegates creating an object to a factory object.
- **Strategy - Abstract Factory - Dependency Injection**
  - Strategy
    A class can be configured with a strategy object.
  - Abstract Factory
    A class can be configured with a factory object.
  - Dependency Injection
    Actually performs the configuration by creating and injecting the objects a class requires.
- **Strategy - Decorator**
  - Strategy provides a way
    to exchange the algorithm of an object at run-time.
    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 at run-time.
    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.

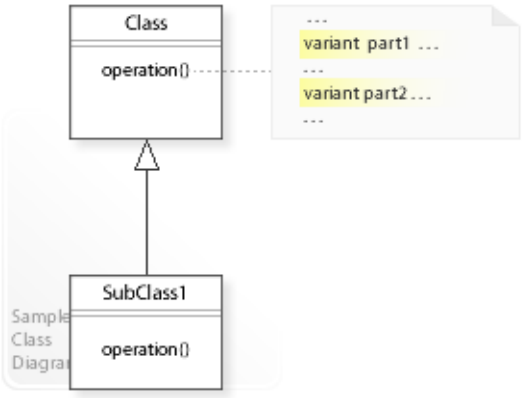
Intent



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 behavior's structure?*
- The standard way is to define subclasses that redefine the behavior of a parent class. This makes it impossible to redefine only certain parts of the behavior independently from (without duplicating) the other 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 a `primitive` operation for each variant part of a behavior (`primitive1()`,`primitive2()`,...).
  - A `templateMethod()` defines the skeleton (structure) of a behavior by
    - implementing the invariant parts and
    - calling `primitives` to defer implementing the variant parts to subclasses.

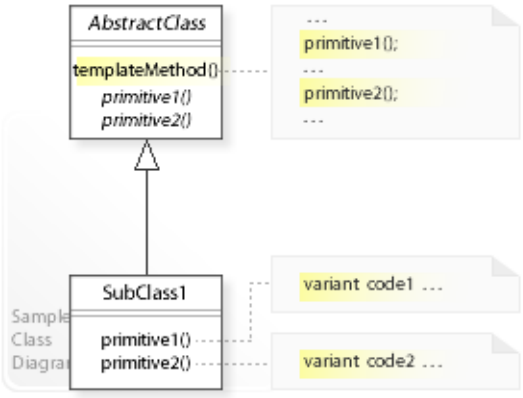
Problem



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 behavior's structure?
See Applicability section for all problems Template Method can solve. See Solution section for how Template Method solves the problems.

- The standard way is to define subclasses (`SubClass1|operation()`) that redefine the behavior of a parent class (`Class|operation()`).
- This makes it impossible to redefine only certain parts of the behavior independently from (without having to duplicate) the other parts.
- *That's the kind of approach to avoid if we want that subclasses can redefine only certain parts of a behavior without changing the other parts or the structure of the behavior.*
- For example, designing reusable classes.
  It should be possible that a class implements the common (invariant) parts of a behavior and let users of the class write subclasses to redefine the variant parts to suit their needs. But subclasses should not be able to change anything else.
- For example, enforcing invariant parts of a behavior.
  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



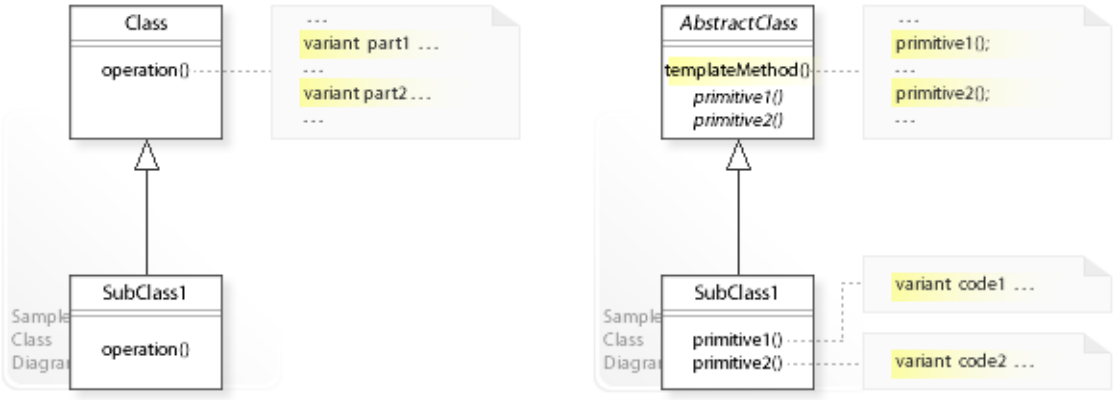
The Template Method design pattern provides a solution:
Define abstract operations (`primitives`) for the variant parts of a behavior.
Define a `template method` that
- implements the invariant parts of a behavior and
- calls `primitives` that subclasses implement.
Describing the Template Method design in more detail is the theme of the following sections.
See Applicability section for all problems Template Method can solve.

- The key idea in this pattern is to control subclassing.
  Subclasses do no longer control how the behavior of a parent class is redefined.
  Instead, a parent class controls how subclasses redefine it.
  This is also referred to as *inversion of control*. "This refers to how a parent class calls the operations of a subclass and not the other way around." [GoF, p327]
- *Inversion of control* is a common feature of *frameworks*.
  When using a *library* (reusable classes), we call the code we want to reuse.
  When using a *framework* (reusable application), we write subclasses and implement the variant code the framework calls.
- Template methods are a fundamental technique for *code reuse*
  (1) 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] (2) and from a refactoring point of view, to eliminate code duplication by factoring out invariant behavior among classes and localizing (generalizing) it in a common class.

Background Information

- The pattern calls abstract operations for variant parts of a behavior *primitives* because the template method composes primitive operations to get a more complex operation.

Motivation 1



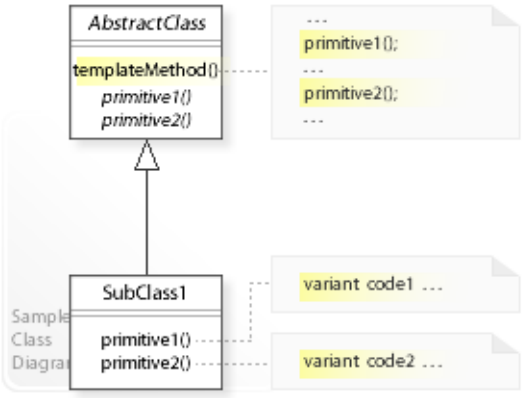
Consider the left design (problem):

- Hard-wired variant parts.
  - The variant parts are implemented (hard-wired) directly within the other code.
  - This makes it impossible 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.
  - For each variant part an abstract operation (`primitive1()`, ...) is defined.
  - This makes it easy for subclasses to redefine the variant parts independently from the other code.
- Controlled subclassing.
  - Subclasses can redefine only the variant parts of a behavior.

Applicability



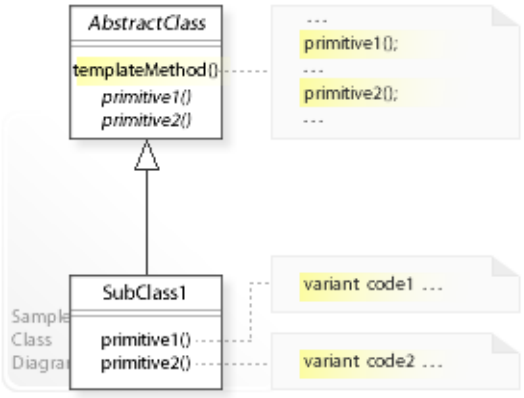
Design Problems

- **Code Reuse / 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 behavior's structure?
- **Extending Behavior at Specific Points**
  - How can subclasses extend a behavior only at specific points (hooks)?
- **Controlling Subclassing**
  - How can a class control how it is subclassed?

Refactoring Problems

- **Duplicated Code**
  - How can common behavior among classes 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]

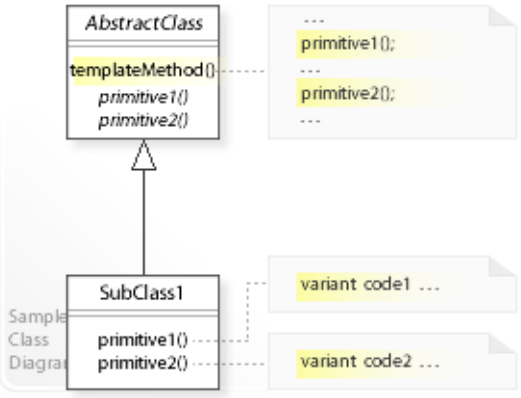
Structure, Collaboration



Static Class Structure

- *`AbstractClass`*
  - Defines a `templateMethod()` operation that defines the skeleton (template) of a behavior by implementing the invariant parts of the behavior and calling abstract `primitive1()` and `primitive2()` operations to defer implementing the variant parts to `SubClass1`.
  - Defines abstract `primitive` operations for the variant parts of a behavior.
- `SubClass1,...`
  - Implement the abstract `primitive` operations.

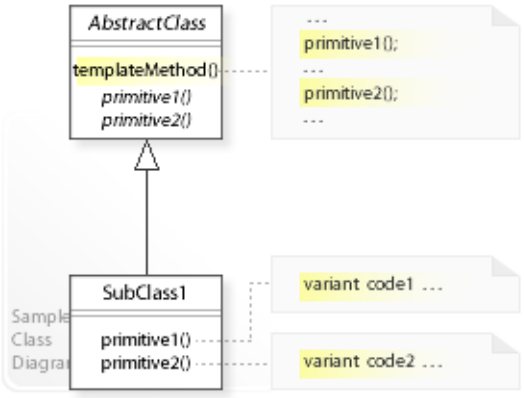
Consequences



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* because subclasses no longer control how the behavior of a parent class is redefined.

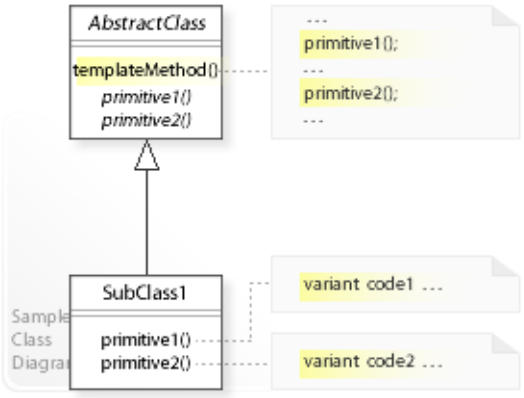
Implementation



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 (see Sample Code).
  - *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

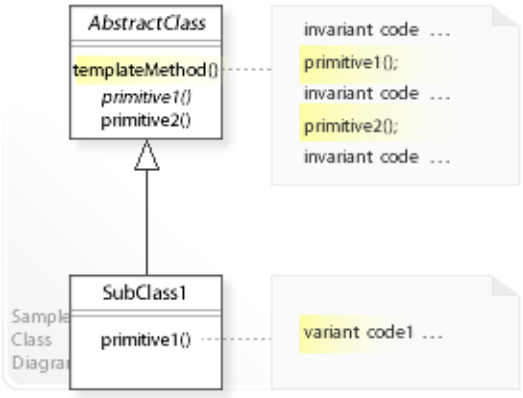


Basic Java code for implementing the sample UML diagram.

```
 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      //
 4      protected void primitive1() {
 5          // ...
 6      }
 7      protected void primitive2() {
 8          // ...
 9      }
10  }
```

Sample Code 2

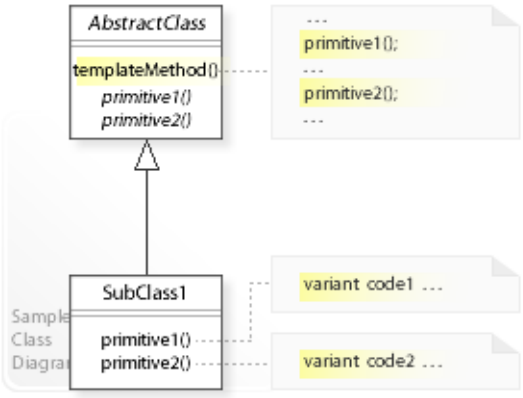


Template method with abstract and concrete primitive operations.

```
package com.sample.templatemethod.steps;
public abstract class AbstractClass {
    // Abstract primitive operation:
    // - provides no default implementation
    // - must be implemented (overridden).
    protected abstract void primitive1();
    //
    // Concrete primitive operation:
    // - provides a default implementation
    // - can be changed (overridden) optionally.
    protected void primitive2() {
        // variant code ...
    }
    public final void templateMethod() {
        // invariant code ...
        primitive1(); // calling primitive1 (variant code)
        // invariant code ...
        primitive2(); // calling primitive2 (variant code)
        // invariant code ...
    }
}
```

```
package com.sample.templatemethod.steps;
public class SubClass1 extends AbstractClass {
    //
    protected void primitive1() {
        // variant code ...
    }
}
```

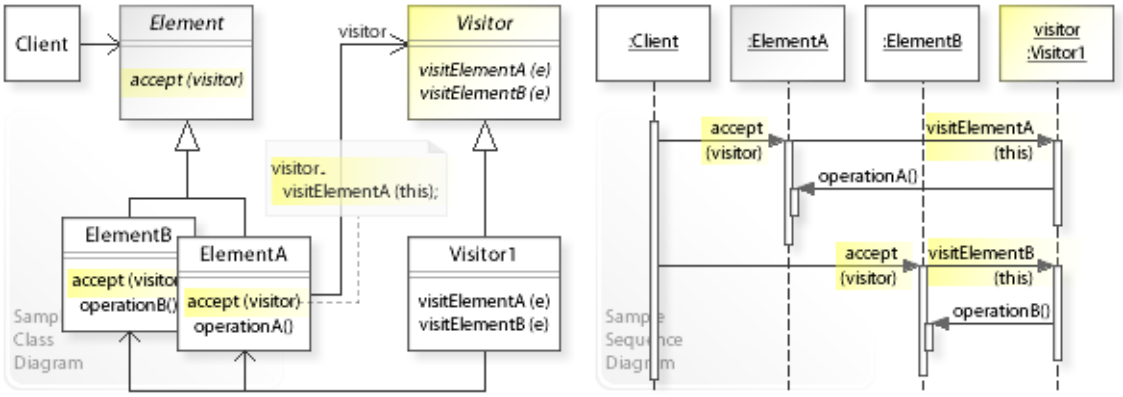
Related Patterns



Key Relationships

- **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 change certain parts of 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.

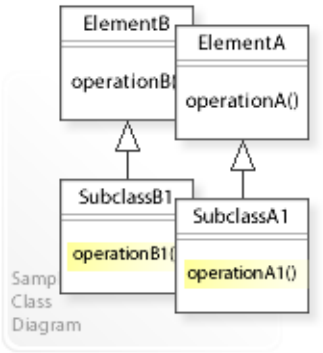
Intent



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 classes of an object structure without changing the classes?*
- For example, an object structure that represents the components of a technical equipment (Bill of Materials).
  Many different applications use the object structure, and it should be possible to define new operations independently from (without having to change) 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 represents operations to be performed on the elements of an object structure.
  - Define a dispatching operation `accept(visitor)` for each element for "dispatching" (delegating) client requests to the "accepted `visitor` object".
  - Clients traverse the object structure and call `accept(visitor)` on each element (by passing in a `visitor` object).

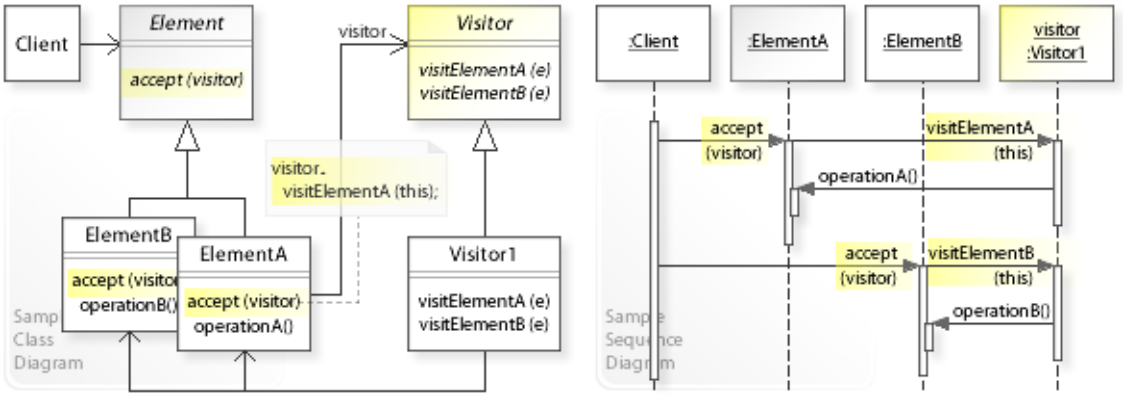
Problem



The Visitor design pattern solves problems like:
How can new operations be defined for the classes of an object structure without changing the classes?
See Applicability section for all problems Visitor can solve. See Solution section for how Visitor solves the problems.

- An inflexible way is to define new subclasses for the classes (`ElementA`,`ElementB`,…) of an object structure each time a new operation is required.
  This makes it hard to define (many) new operations for an object structure that contains (many) different classes (having different interfaces). "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 classes of an object structure independently from (without having to change/extend) the classes.*
- For example, an object structure that represents the components of a technical equipment (Bill of Materials).
  Many different applications use the object structure, and it should be possible (for each application) to define new operations (for example, for calculating total prices, computing inventory, etc.) without having to change/extend the classes of the object structure (see Sample Code / Example 2).

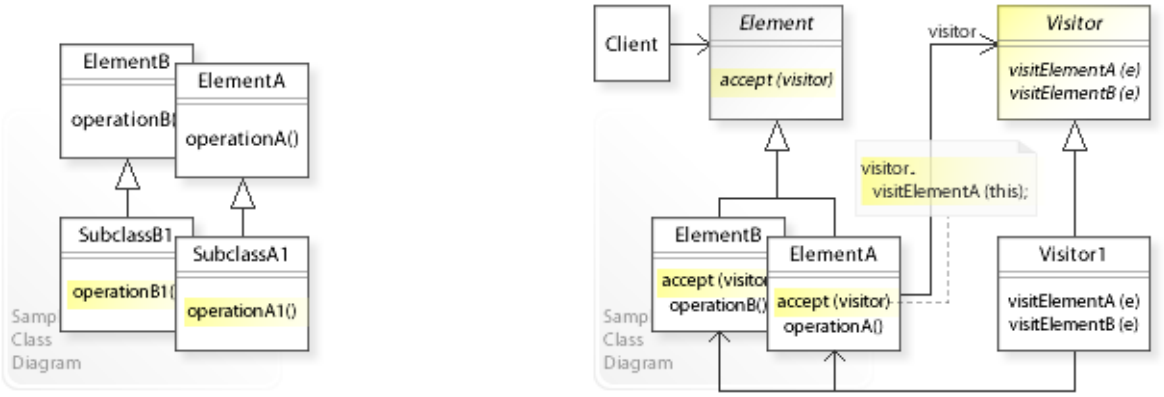
Solution



The Visitor design pattern provides a solution:
Define a separate `Visitor` object that implements operations to be performed on the elements of an object structure.
Clients traverse the object structure and call `accept(visitor)` on each element to delegate the request to the "accepted `visitor` object".
The `visitor` object then performs the request ("visits" the element).
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 define a *double-dispatch operation* **`accept(visitor)`** for each `Element` class (see also Collaboration and Implementation).
  "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:**
  - For all supported `Element` classes, define a common `Visitor` interface by defining a "visit" operation for each `Element` class (`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]

Motivation 1



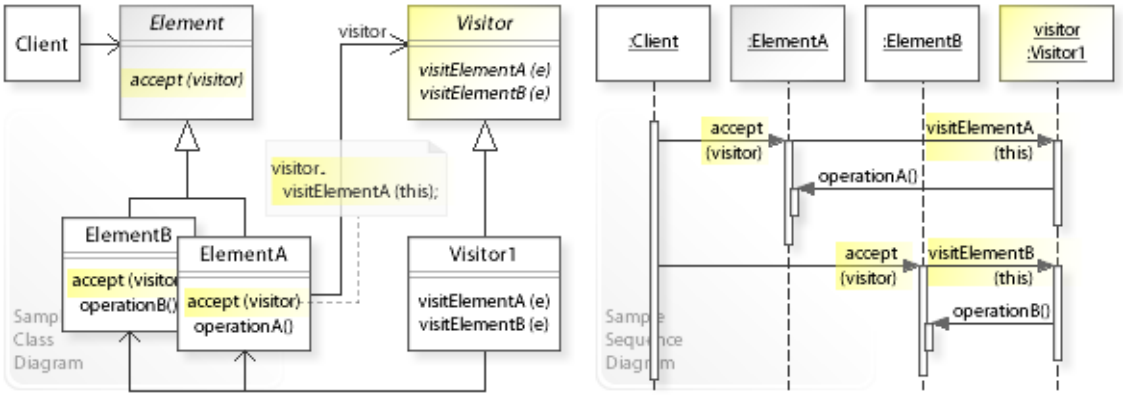
Consider the left design (problem):

- New operations are distributed across the `Element` classes.
  - A new operation for each `Element` class is defined by adding subclasses.
  - "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):

- New operations are encapsulated in a separate `Visitor` class.
  - A new operation for each `Element` class is defined in a separate class (`Visitor1`).
  - "You create a new operation by adding a new subclass to the visitor class hierarchy." [GoF, p333]

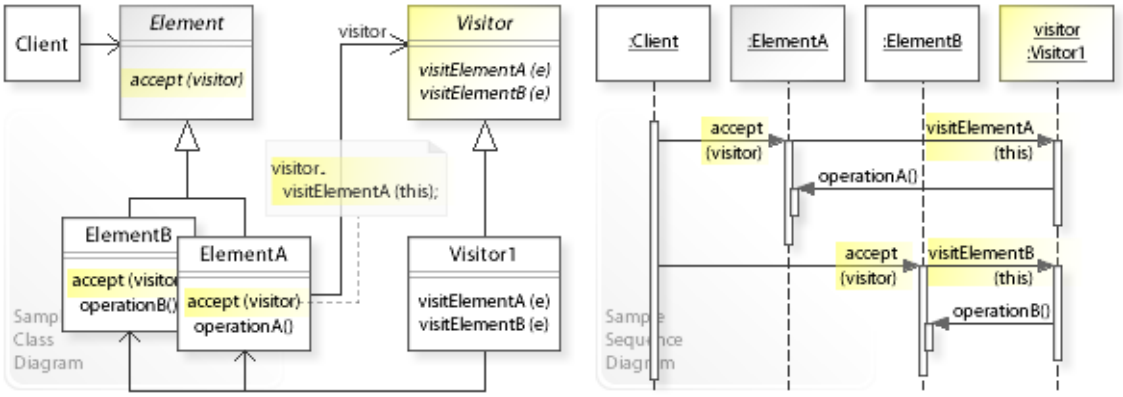
Applicability



Design Problems

- **Defining New Operations for Object Structures**
  - How can new operations be defined for the classes of an object structure without changing the classes?
- **Flexible Alternative to Subclassing**
  - How can a flexible alternative be provided to subclassing for defining new operations for the classes of an object structure?

Structure, Collaboration



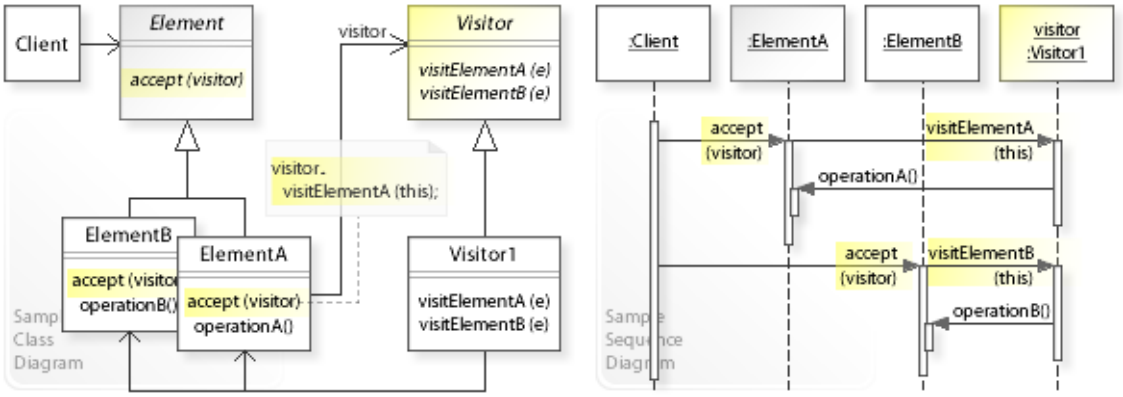
Static Class Structure

- `Client`
  - Traverses the elements of an object structure.
- *`Element`*
  - Defines an interface for dispatching (delegating) client requests to a `Visitor` object (`accept(visitor)`).
- `ElementA,ElementB,...`
  - Implement the dispatching interface (see Implementation).
- *`Visitor`*
  - For all supported `Element` classes, defines a common interface for "visiting" (performing an operation on) each `Element` class.
  - "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 the elements of an object structure (`ElementA,ElementB`) and calls `accept(visitor)` on each element.
  Lets assume that the `Client` provides a `Visitor1` object.
- The interaction starts with the `Client` object that calls `accept(visitor)` on the `ElementA` object.
- The dispatching operation `accept(visitor)` of `ElementA` calls `visitElementA(this)` on the accepted `Visitor1` object.
- `ElementA` passes itself (`this`) to `Visitor1` so that `Visitor1` can visit (call back) `ElementA` and do its work on it (by calling `operationA()`).
- Thereafter, the `Client` calls `accept(visitor)` on `ElementB`, which calls `visitElementB(this)` on the accepted `Visitor1` object.
- `Visitor1` does its work on `ElementB` (by calling `operationB()`).
- See also Sample Code / Example 1.

Consequences



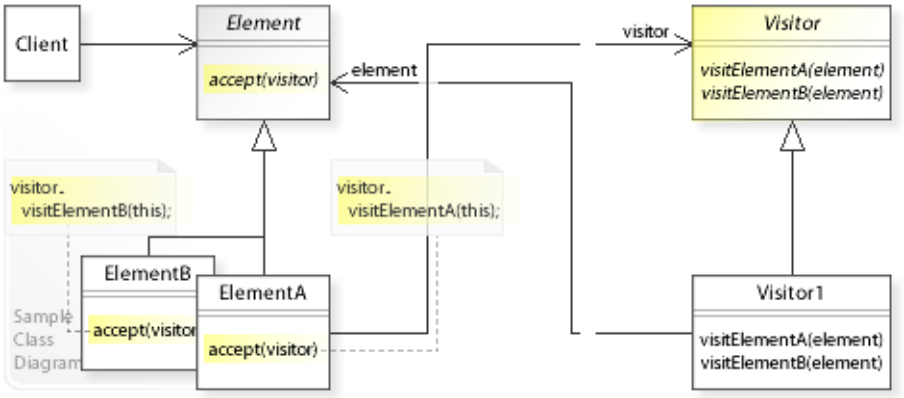
Advantages (+)

- 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 Visitors).

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 may have to be extended to let all visitors do their work and access the needed data and functionality.
- Introduces additional levels of indirection.
  - The pattern achieves flexibility by introducing separate visitor objects and a double-dispatch mechanism, which can complicate a design.

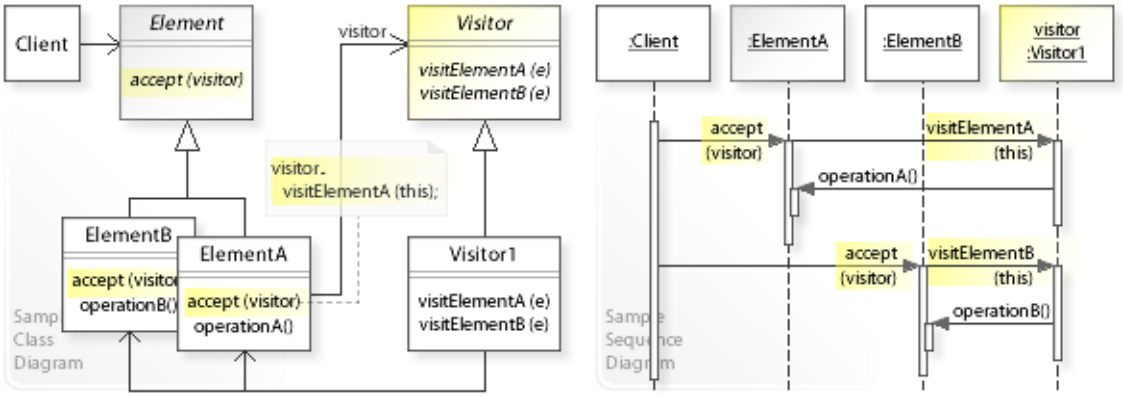
Implementation



Implementation Issues

- **Dispatching Operation `accept(visitor)`**
  - Each `Element` class of the object structure defines an **`accept(visitor)`** operation that delegates (dispatches) client requests to the accepted (passed in) `visitor` object *and* the `visit` operation that corresponds to the `Element` class. For example:
    `ElementA|accept(visitor)` delegates to `visitor.visitElementA(this)`,
    `ElementB|accept(visitor)` 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 so that the visitor can visit (call back) this element and do some work on it.
- **`Visitor` Interface**
  - The `Visitor` interface defines a `visit` operation for each element class that should be visited.
    The interface must be extended 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 Visitor1 object.
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 something on) ElementA.
Hello World from ElementA!
Visitor1: Visiting (doing something on) ElementB.
Hello World from ElementB!
```

```
 1  package com.sample.visitor.basic;
 2  public abstract class Element {
 3      public abstract void accept(Visitor visitor);
 4  }
```

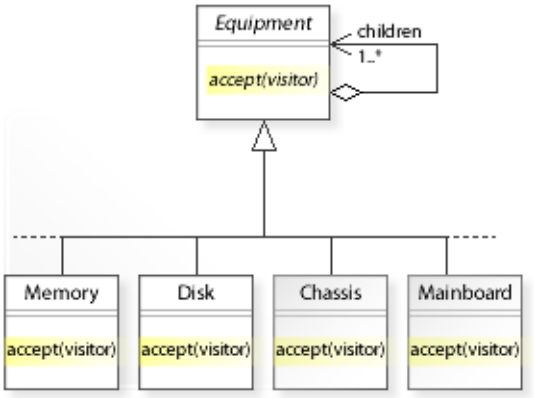
```
 1  package com.sample.visitor.basic;
 2  public class ElementA extends Element {
 3      public void accept(Visitor visitor) {
 4          visitor.visitElementA(this);
 5      }
 6      public String operationA() {
 7          return "Hello World from ElementA!";
 8      }
 9  }
```

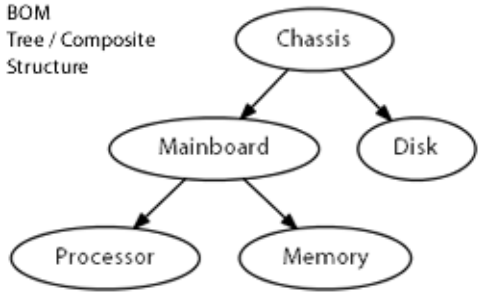
```
 1  package com.sample.visitor.basic;
 2  public class ElementB extends Element {
 3      public void accept(Visitor visitor) {
 4          visitor.visitElementB(this);
 5      }
 6      public String operationB() {
 7          return "Hello World from ElementB!";
 8      }
 9  }
```

```
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("Visitor1: Visiting (doing something on) ElementA.\n"
5                  + element.operationA());
6      }
7      public void visitElementB(ElementB element) {
8          System.out.println("Visitor1: Visiting (doing something on) ElementB.\n"
9                  + element.operationB());
10     }
11 }
```

Sample Code 2





BOM Bill of Materials / Using pricing and inventory visitors.
The pricing visitor calculates the number of components and the total price.
The inventory visitor calculates the inventory of each component.
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  public class MyApp {
 3      public static void main(String[] args) throws Exception {
 4          // Building a BOM tree (composite structure).
 5          Equipment mainboard = new Mainboard("Mainboard", 100);
 6          mainboard.add(new Processor("Processor", 100));
 7          mainboard.add(new Memory("Memory   ", 100));
 8          Equipment chassis = new Chassis("Chassis  ", 100);
 9          chassis.add(mainboard);
10          chassis.add(new Disk("Disk     ", 100));
11
12          System.out.println("(1) Traversing the BOM using a pricing visitor: ");
13          PricingVisitor pricingVisitor = new PricingVisitor();
14          chassis.accept(pricingVisitor);
15          System.out.println(
16              "    Number of components: " + pricingVisitor.getNumberOfElements() +
17            "\n    Total price         : " + pricingVisitor.getTotalPrice());
18
19          System.out.println("(2) Traversing the BOM using an inventory visitor: ");
20          InventoryVisitor inventoryVisitor = new InventoryVisitor(new Inventory());
21          chassis.accept(inventoryVisitor);
22      }
23  }
```

```
(1) Traversing the BOM using a pricing visitor:
    Number of components: 5
    Total price         : 500
(2) 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      //
```

```
    public String getName() {
        return this.name;
    } ;
    public boolean add(Equipment e) {
        return children.add(e);
    }
    public Iterator<Equipment> iterator() {
        return children.iterator();
    }
    public int getChildCount() {
        return children.size();
    }
}
```

```
package com.sample.visitor.bom;
public abstract class EquipmentVisitor {
    public abstract void visitChassis(Chassis e);
    public abstract void visitMainboard(Mainboard e);
    public abstract void visitProcessor(Processor e);
    public abstract void visitMemory(Memory e);
    public abstract void visitDisk(Disk e);
}
```

```
package com.sample.visitor.bom;
public class PricingVisitor extends EquipmentVisitor {
    private int count = 0;
    private long sum = 0;
    public void visitChassis(Chassis e) {
        count++;
        sum += e.getCostPrice();
    }
    public void visitMainboard(Mainboard e) {
        count++;
        sum += e.getBasicPrice();
    }
    public void visitProcessor(Processor e) {
        count++;
        sum += e.getPurchaseCost();
    }
    public void visitMemory(Memory e) {
        count++;
        sum += e.getPrice();
    }
    public void visitDisk(Disk e) {
        count++;
        sum += e.getUnitPrice();
    }
    public int getNumberOfElements() {
        return count;
    }
    public long getTotalPrice() {
        return sum;
    }
}
```

```
package com.sample.visitor.bom;
public class InventoryVisitor extends EquipmentVisitor {
    private Inventory inventory;
    public InventoryVisitor(Inventory inventory) {
        this.inventory = inventory;
    }
    public void visitChassis(Chassis e) {
        inventory.operation(e);
    }
    public void visitMainboard(Mainboard e) {
        inventory.operation(e);
    }
    public void visitProcessor(Processor e) {
        inventory.operation(e);
    }
    public void visitMemory(Memory e) {
        inventory.operation(e);
    }
    public void visitDisk(Disk e) {
        inventory.operation(e);
    }
```

```
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 }
```

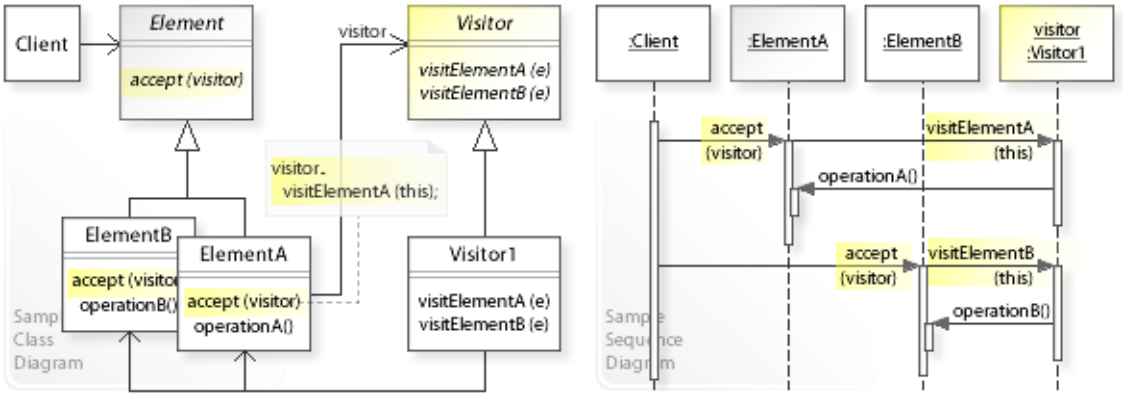
```
1  package com.sample.visitor.bom;
2  public class Mainboard extends Equipment {  // Composite
3      private long price;
4      public Mainboard(String name, long price) {
5          super(name);
6          this.price = price;
7      }
8      public void accept(EquipmentVisitor visitor) {
9          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

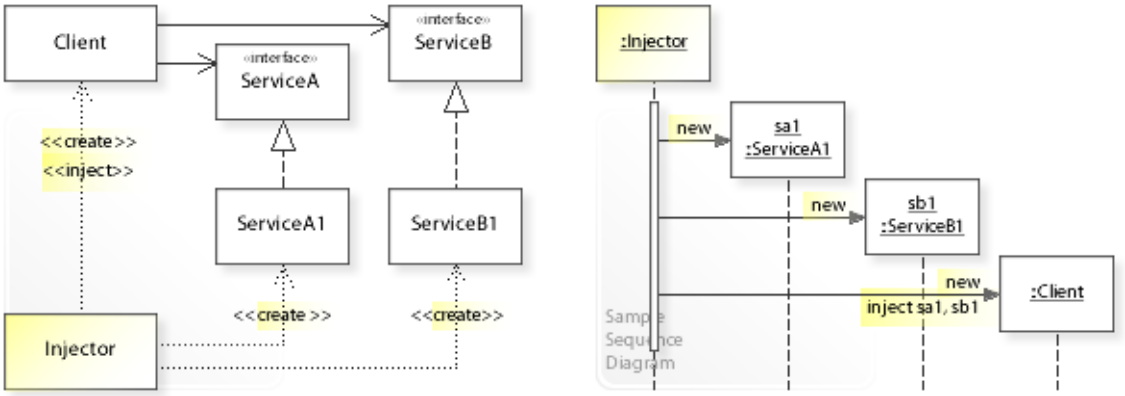


Key Relationships

- **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

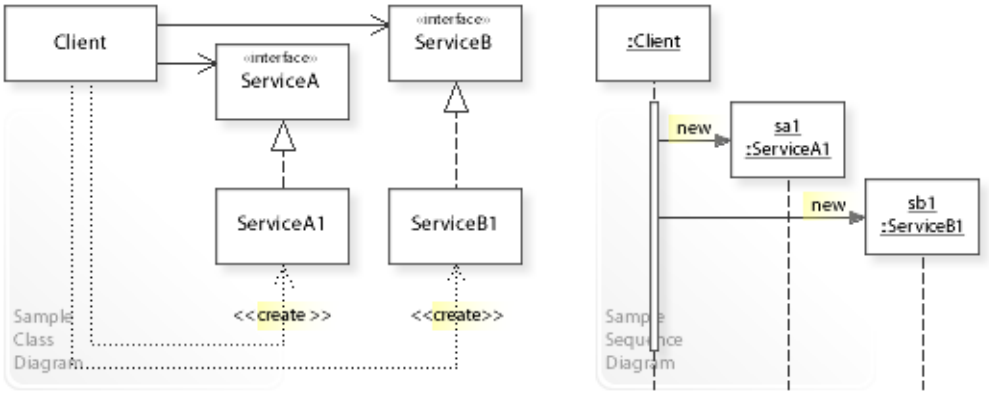
Intent



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 the way objects are created be specified in separate configuration files?*
- An inflexible way is to create objects directly within the class (`Client`) that requires the objects. This commits the class to how the objects are created and makes it impossible to change the instantiation later independently from (without changing) the class.
- The Dependency Injection pattern describes how to solve such problems:
  - *Separate object creation from an application:*
  - Define a separate `Injector` object that creates and injects the objects a class requires.
  - A class accepts the objects it requires from an `Injector` object instead of creating the objects directly.

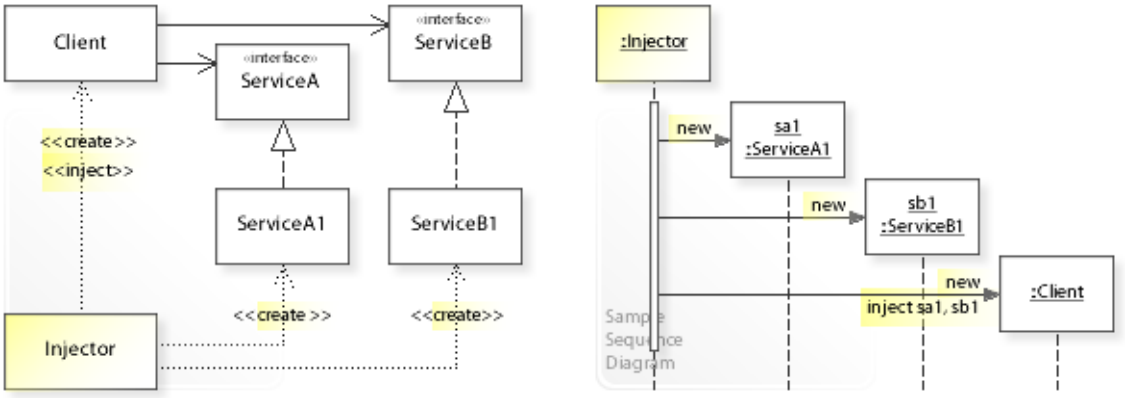
Problem



The Dependency Injection design pattern solves problems like:
How can a class be independent of how the objects it requires are created?
How can the way objects are created be specified in separate configuration files?
See Applicability section for all problems Dependency Injection can solve. See Solution section for how Dependency Injection solves the problems.

- An inflexible way is to create objects (`new ServiceA1(), new ServiceB1()`) directly within the class (`Client`) that requires (uses) the objects.
- This commits (couples) 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 if other objects are required, and it makes the class hard to test because real objects can't be replaced with mock objects.
  Furthermore, a class often requires objects that have further dependencies, which in turn have dependencies, and so on, which results in having to create a complex object structure manually.
- *That's the kind of approach to avoid if we want that a class is independent of how its objects are created.*
- For example, designing reusable classes that require (depend on) other objects.
  A reusable class should avoid creating the objects it requires directly (and often it doesn't know at compile-time which class to instantiate) so that it can accept the objects it requires at run-time (from an injector object).
- For example, supporting different configurations of an application.
  Instantiating concrete classes throughout an application should be avoided so that the way objects are created can be specified in separate (external) configuration files.

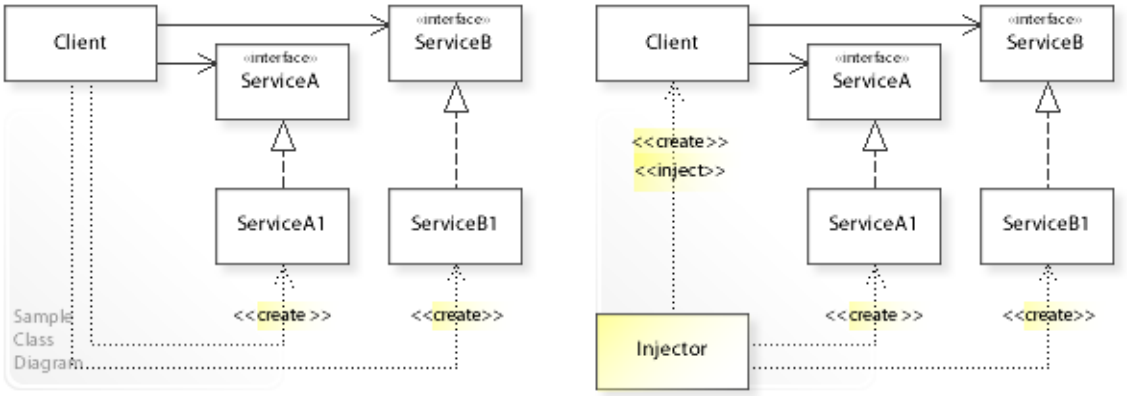
Solution



The Dependency Injection pattern describes a solution:
Define a separate `Injector` object that creates and injects the objects a class requires.
A class accepts the objects it requires from an `Injector` object
instead of creating the objects directly.
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. [See also]

- The key idea in this pattern is to separate *using* objects from *creating* them.
  A class is no longer responsible for creating the objects it requires (uses).
  Instead, a separate injector object creates the objects and injects them into the class.
  This is also referred to as *inversion of control*, which is a common feature of frameworks (compare with Template Method).
- **Define a separate `Injector` object:**
  - The way objects are created (that is, the mapping of interfaces to implementations) is specified in separate *configuration* files or objects (`ServiceA -> ServiceA1`, `ServiceB -> ServiceB1`).
  - To let the injector do its work, classes must provide a *constructor* (and/or *setter methods*) through which the objects can be passed in (injected).
- **A class (`Client`) accepts the objects it requires automatically at run-time.**
  - A class can use objects solely through their interfaces (`ServiceA`, `ServiceB`) and doesn't have to care about how the objects are created.
  - This greatly simplifies classes and makes them easier to implement, change, test, and reuse.

Motivation 1



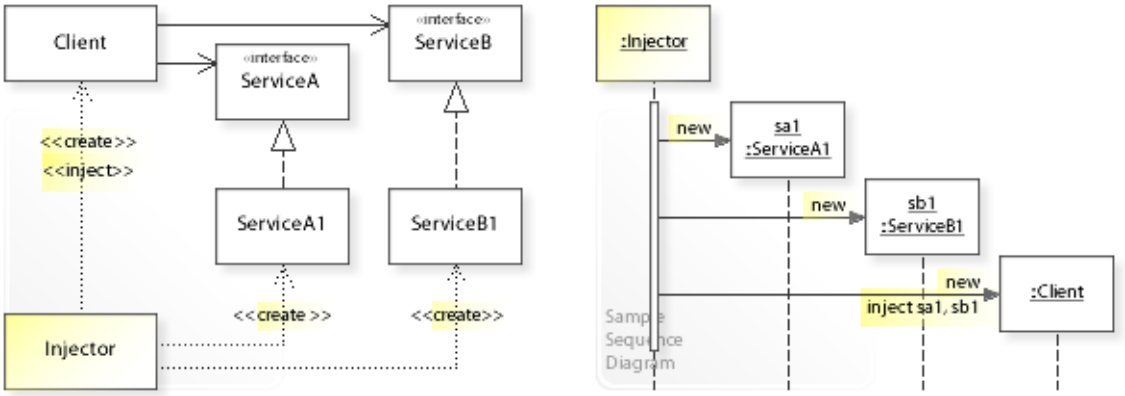
Consider the left design (problem):

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

Consider the right design (solution):

- Separated object creation.
  - A separate `Injector` creates and injects the objects. The way objects are created 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 a class be independent of how the objects it requires are created?
  - How can a class accept the objects it requires (from an injector object) instead of creating the objects directly?
- **Specifying Different Configurations**
  - How can the way objects are created (the mapping of interfaces to implementations) be specified in separate configuration files or objects?
  - How can an application support different configurations?
- **Resolving Dependencies Recursively**
  - How can the objects a class requires be created recursively?

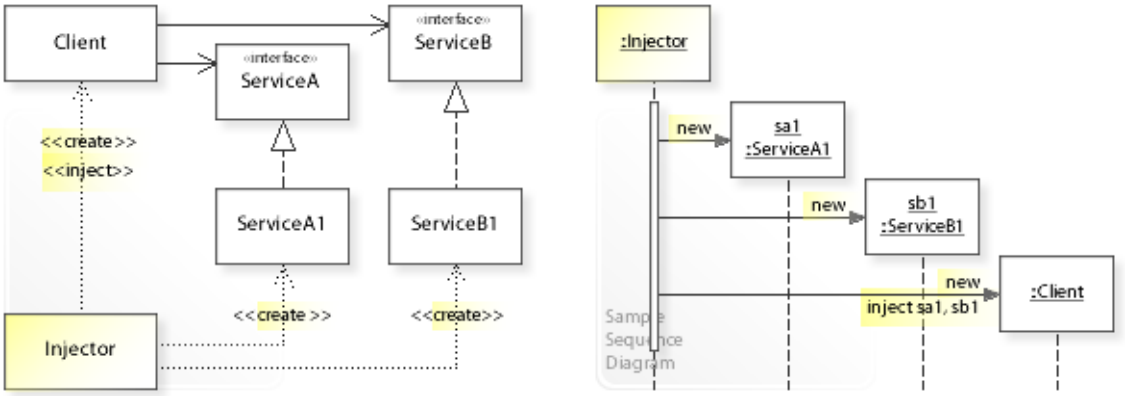
Refactoring Problems

- **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 and externalized?

Testing Problems

- **Unit Testing**
  - How can the objects a class requires be replaced with mock objects so that the class can be unit tested in isolation?

Structure, Collaboration



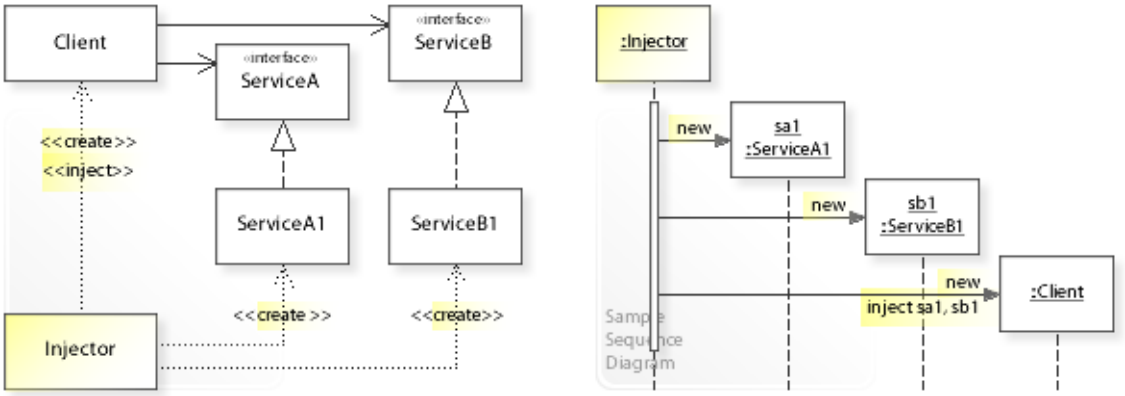
Static Class Structure

- `Client`
  - Requires `ServiceA` and `ServiceB` objects.
  - Accepts the objects from the `Injector`.
  - Is independent of how the objects are created (which concrete classes are instantiated).
  - Doesn't know the `Injector`.
- `Injector`
  - Creates the `ServiceA1` and `ServiceB1` objects; creates the `Client` (if it doesn't already exist) and injects the objects into the `Client`.

Dynamic Object Collaboration

- In this sample scenario, an `Injector` object creates `ServiceA1` and `ServiceB1` objects. Thereafter, it creates a `Client` object and injects the `ServiceA1` and `ServiceB1` objects.
  Let's assume that the `Injector` uses a `Configuration` file that maps `ServiceA` and `ServiceB` interfaces to `ServiceA1` and `ServiceB1` implementations.
- The `Injector` starts with creating the `ServiceA1` and `ServiceB1` objects.
- Thereafter, the `Injector` creates the `Client` object and injects the `ServiceA1` and `ServiceB1` objects.
- The `Client` object can then use the `ServiceA1` and `ServiceB1` objects as required.
- See also Sample Code / Example1.

Consequences

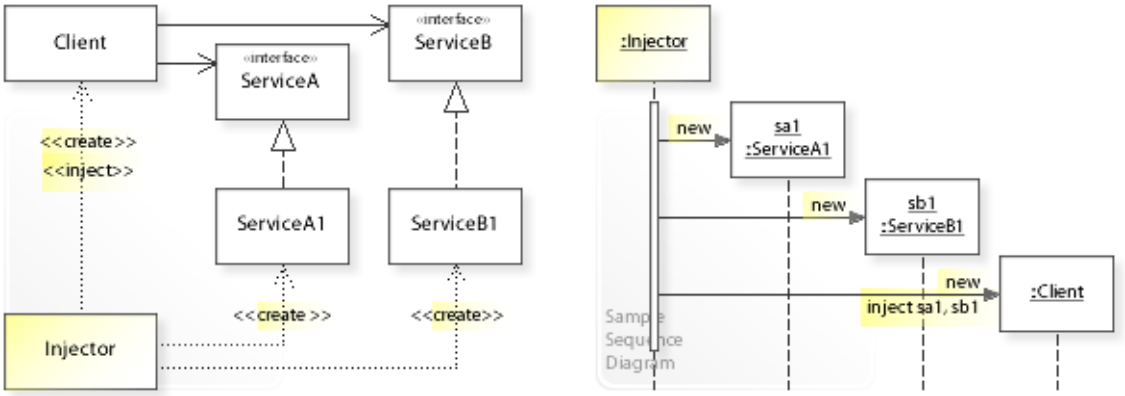


Advantages (+)

- Avoids compile-time 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, which makes the classes easier to implement, change, test, and reuse.
- 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 to pass in the objects they require. There are two main implementation variants:
- Variant1: **Constructor Injection**
  - Classes define constructors that pass in the objects:

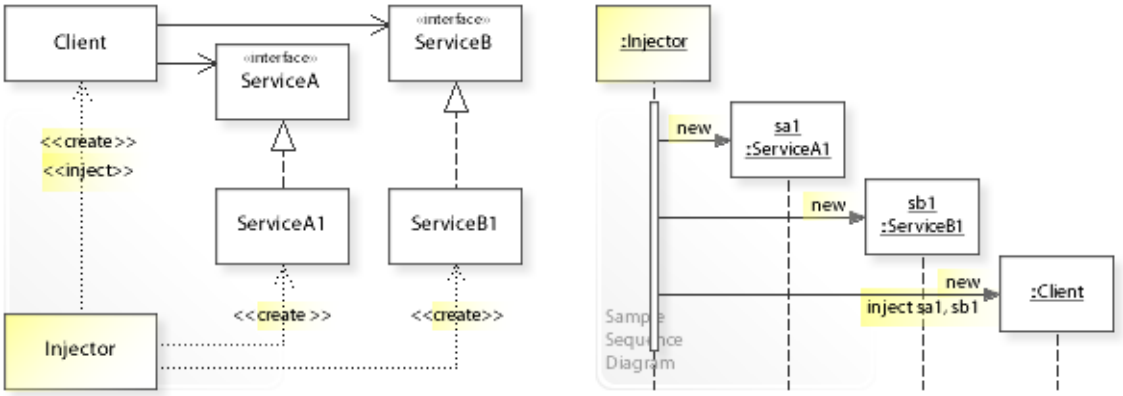
```
class Client ...
   private Service service;
   public Client(Service service) {
      this.service = service;
   }
```

- Variant2: **Setter Injection**
  - Classes define setter methods to pass in the objects:

```
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 a class when it is instantiated. It ensures that an object is configured properly before it is used.
  - 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          // Requesting a Client object from the injector.
 9          Client client = injector.getInstance(Client.class);
10          // Performing an operation on the client.
11          System.out.println(client.operation());
12      }
13  }
```

```
Client : Accepting objects from the injector.
Hello World from ServiceA1 and ServiceB1!
```

```
 1  package com.sample.di.basic;
 2  import com.google.inject.Inject;
 3  public class Client {
 4      private ServiceA serviceA;
 5      private ServiceB serviceB;
 6
 7      @Inject    // Constructor Injection
 8      public Client(ServiceA serviceA, ServiceB serviceB) {
 9          System.out.println("Client : Accepting objects from the injector.");
10          this.serviceA = serviceA;
11          this.serviceB = serviceB;
12      }
13      public String operation() {
14          // Doing something appropriate on the accepted objects.
15          return "Hello World from " + serviceA.getName() + " and "
16                  + serviceB.getName() + "!";
17      }
18  }
```

```
 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.
******************************************************
```

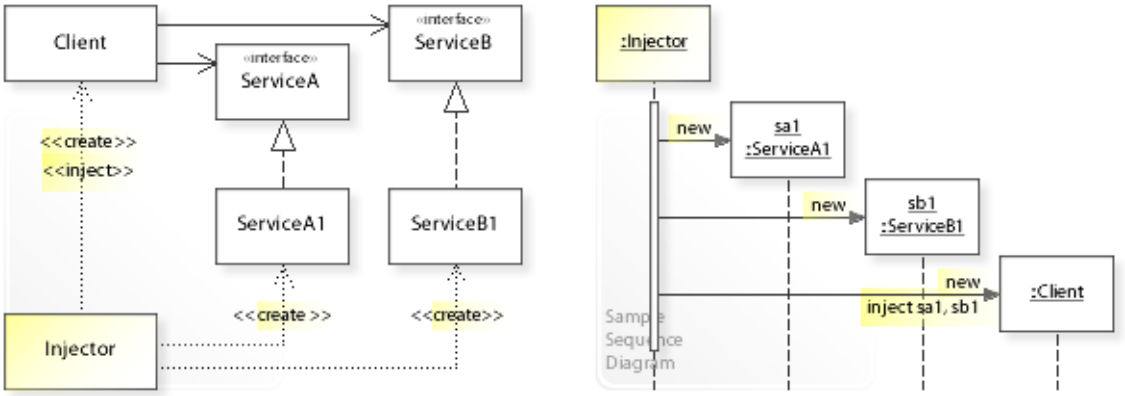
```
 1  package com.sample.di.basic;
 2  import com.google.inject.Guice;
 3  import com.google.inject.Injector;
 4  import junit.framework.TestCase;
 5  public class ClientTest extends TestCase {
 6      // Requesting an Injector object.
 7      Injector injector = Guice.createInjector(new ConfigurationMock());
 8      // Requesting a Client object from the injector.
 9      Client client = injector.getInstance(Client.class);
10
11      public void testOperation() {
12          assertEquals("Hello World from ServiceAMock and ServiceBMock!",
13                  client.operation());
14      }
15      // More tests ...
16  }
```

```
 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

- **Abstract Factory - Dependency Injection**
  - Abstract Factory
    A class delegates creating the objects it requires to a factory object, which makes the class dependent on the factory.
  - Dependency Injection
    A class accepts the objects it requires from an injector object without having to know the injector, which greatly simplifies the class.
- **Strategy - Abstract Factory - Dependency Injection**
  - Strategy
    A class can be configured with a strategy object.
  - Abstract Factory
    A class can be configured with a factory object.
  - Dependency Injection
    Actually performs the configuration by creating and injecting the objects a class 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.