# SWOP

## GRASP

**GRASP: General Responsibility Assignment Software Principles and Patterns**

9 general principles in assigning responsibility:

1. Information Expert
2. Creator
3. Controller
4. High Cohesion
5. Low Coupling
6. Polymorphism
7. Pure Fabrication
8. Indirection
9. Protected Variations

I Can Cum in Her Legendary Pussy Pie, Indirectly Poggers

-> Mnemonic (ezelsbruggetje)

### Information Expert

Assign a responsibility to the information expert - the class that has the information necessary to fulfill the responsibility.

### Creator

Who creates? (Note that Factory is a common alternate solution)

Assign class B the responsibility  to create an instance of class A if one of these is true:

1. B contains A
2. B aggregates A
3. B has the initializing data for A
4. B records A
5. B closely uses A

### Controller

Who handles a system event?

Assign the responsibility  for handling a system event message to a class representing one of these choices

1. Represents the overall system, device, or a subsystem (facade controller)
2. Represents a use case scenario within which the system event occurs (use-case or session controller)

## High Cohesion

How to keep complexity manageable?

Assign responsibilities so that cohesion remains high

## Low Coupling

How to support low dependency and increased reuse?

Assign responsibilities so that (unnecessary) coupling remains low

## Polymorphism

Who is responsible when behaviour varies by type?

When related alternatives or behaviours vary by type (class), assign responsibility for the behaviour - using polymorphic operations - to the types for which the behaviour varies.

## Pure Fabrication

Who is responsible when you are desperate, and do not want to violate high cohesion and low coupling?

Assign a highly cohesive set of responsibilities to an artificial or convenience "behaviour" class that does not represent a problem domain concept - something made up, in order to support high cohesion, low coupling, and reuse.

## Indirection

How to assign responsibilities to avoid direct coupling?

Assign the responsibility to an intermediate object to mediate between other components or services, so that they are not directly coupled

## Protected Variations

How to assign responsibilities to objects, subsystems and systems so that the variations or instability in these elements do not have an undesirable impact on other elements?

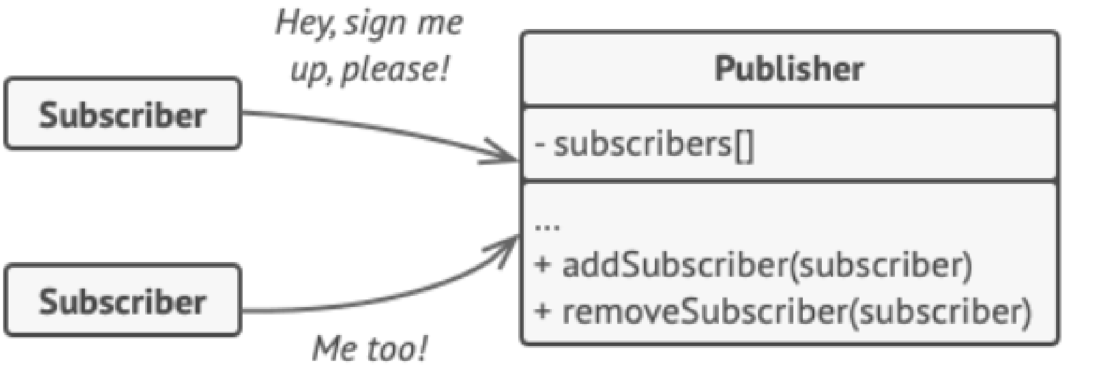
Identify points of predicted variation or instability; assign responsibilities to create a stable "interface" around them.

# Design Patters

## Observer

Say we have one-to-many relationship between objects such as if one object is modified, its dependent objects are to be notified automatically.

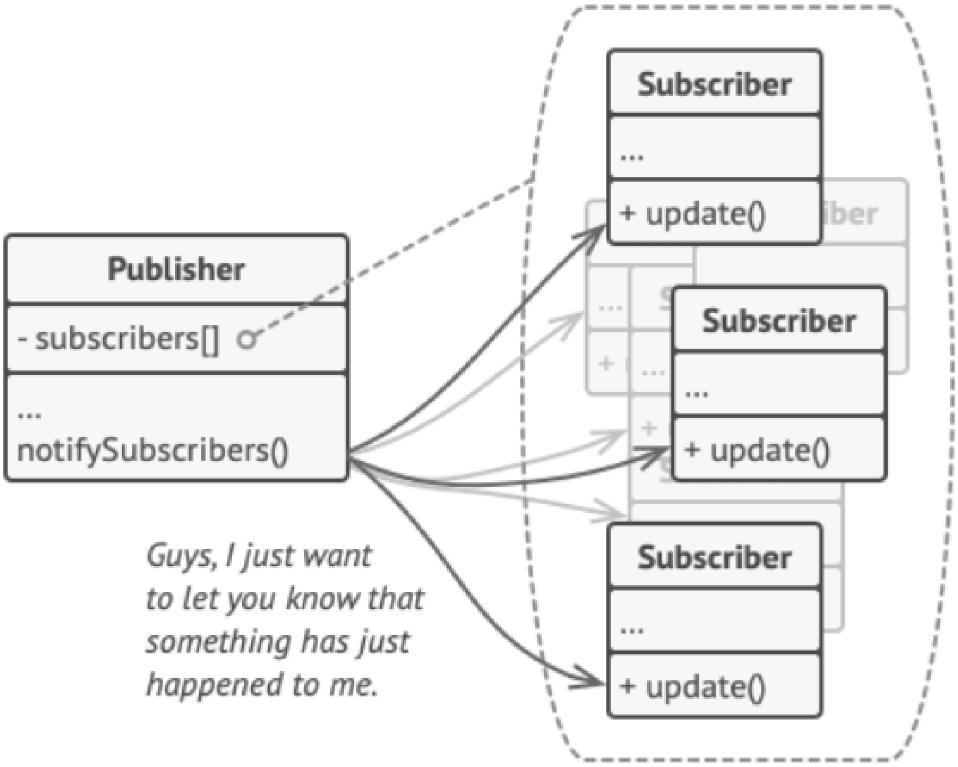
The way the **Observer** pattern tries to solve this, is based on a **subscription** mechanism. So we have **subscribers/observers** and a **subject/publisher**. As can be seen below.



*A subscription mechanism lets individual objects subscribe to event notifications.*

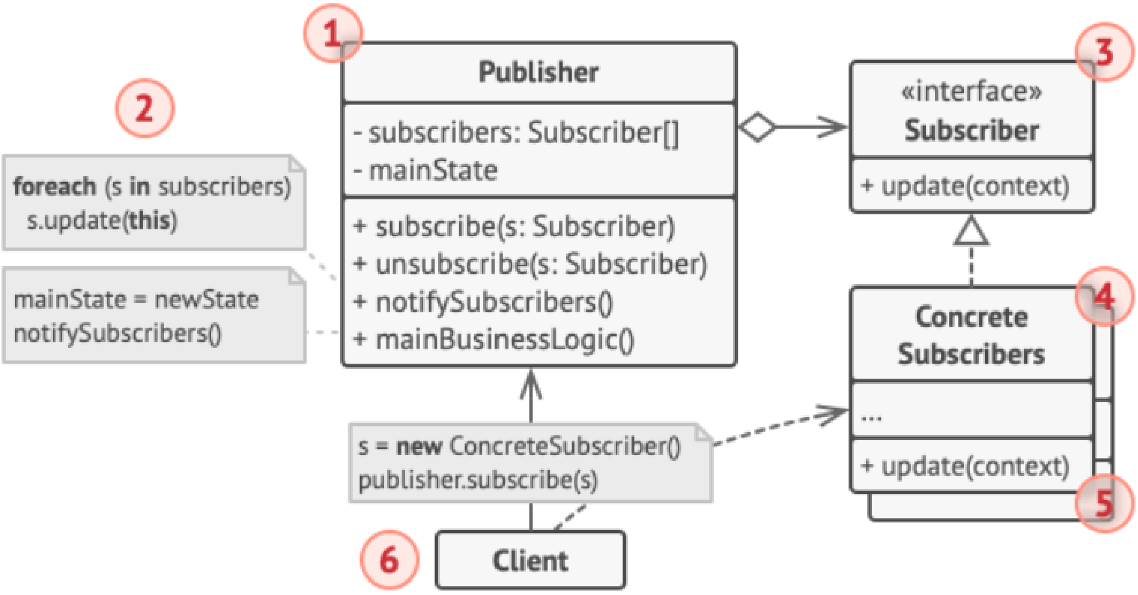
A subscriber/Observer can "subscribe" itself to a publisher and then will be notified every time a change happens. it's crucial that all subscribers implement the **same interface** and that the publisher communicates with them only via that interface. This interface should declare the **notification method** along with a set of parameters that the publisher can use to pass some contextual data along with the notification.

Usually we have a second interface as well, one for the **publishers/subjects**. If they follow the same interface, all of the subscribers will be compatible with all of them.



*Publisher notifies subscribers by calling the specific notification method on their objects.*

Below you can see what the structure would look like in an UML.



## Composite

**Composite** is a structural design pattern that lets you compose objects into tree structures and then work with these structures as if they were individual objects.

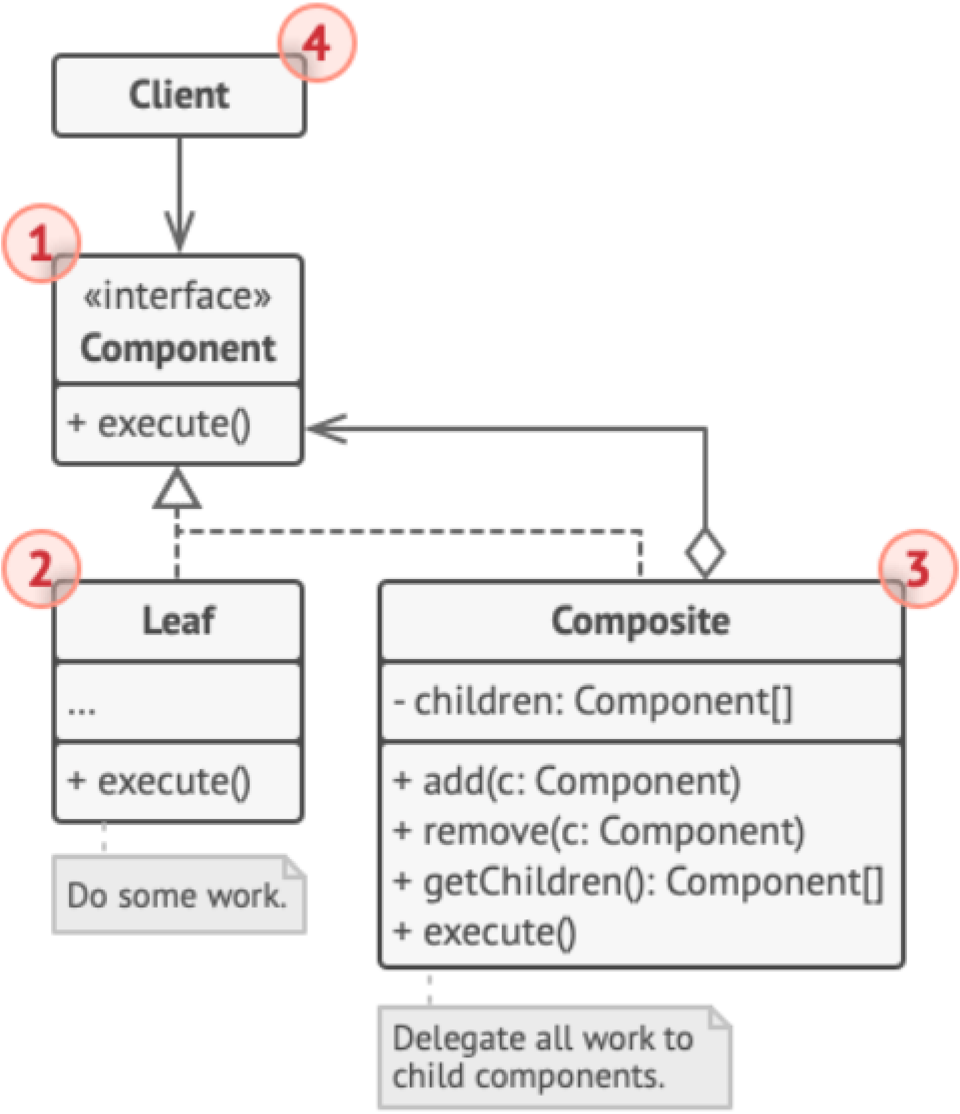
**The problem we are trying to solve:**

For example, imagine that you have two types of objects: `Products` and `Boxes`. A `Box` can contain several `Products` as well as a number of smaller `Boxes`. These little `Boxes` can also hold some `Products` or even smaller `Boxes`, and so on.

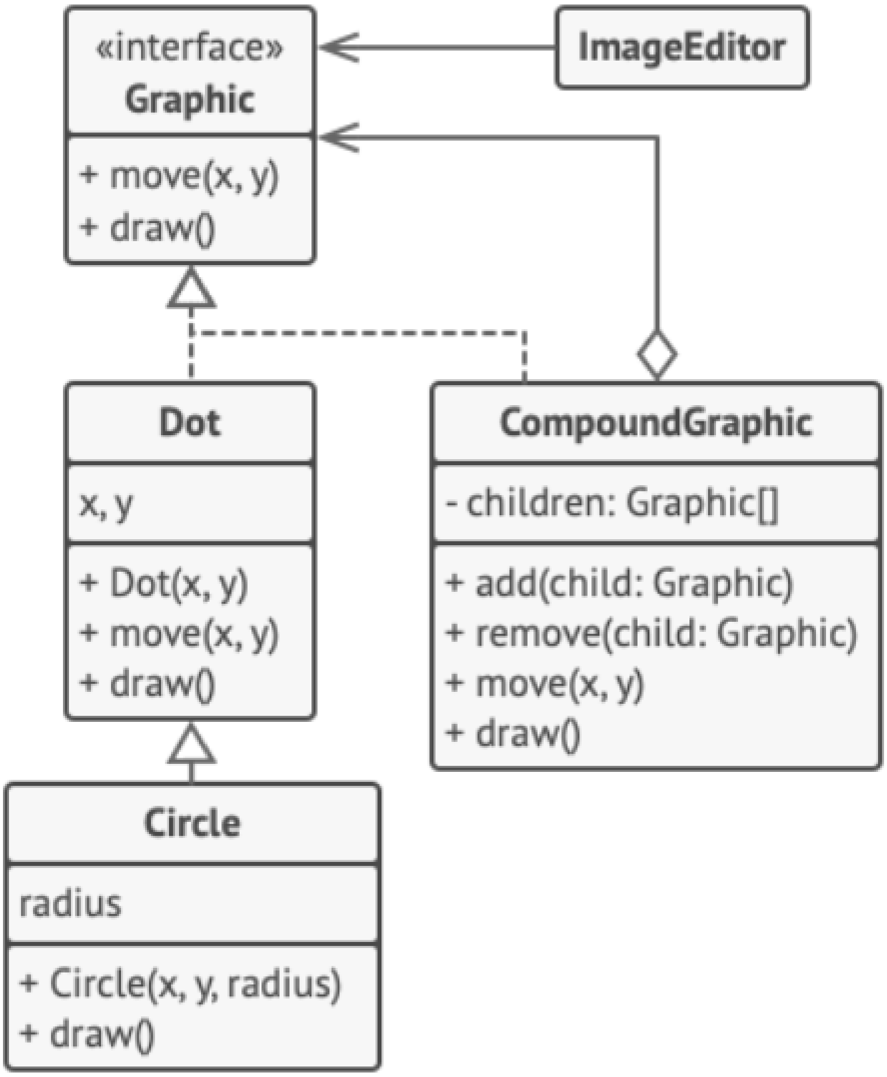
Say you decide to create an ordering system that uses these classes. Orders could contain simple products without any wrapping, as well as boxes stuffed with products...and other boxes. How would you determine the total price of such an order?

**Solution**

The Composite pattern suggests that you work with a common interface which declares a method for calculating the total price. Below you can see what this structure would look like in an UML.



Let's give a concrete example for some clarification. We will work with geometric shapes.



*The geometric shapes editor example.*

The `CompoundGraphic` class is a container that can comprise any number of sub-shapes, including other compound shapes. A compound shape has the same methods as a simple shape. However, instead of doing something on its own, a compound shape passes the request recursively to all its children and "sums up" the result.

Another good explanation:

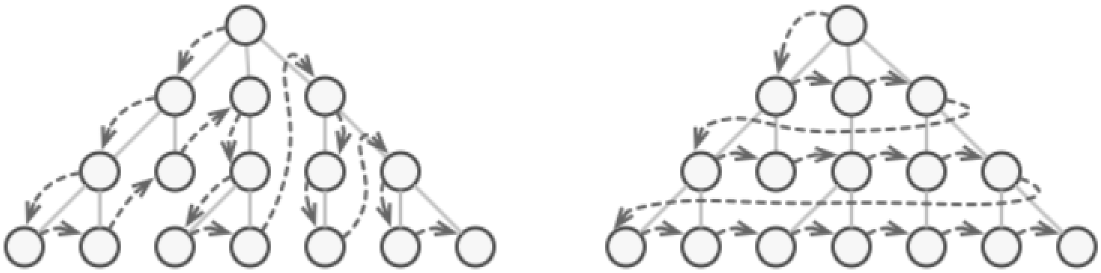
https://dotnettutorials.net/lesson/composite-design-pattern/

## Iterator

Iterator is a behavioral design pattern that lets you traverse elements of a collection without exposing its underlying representation (e.g. list, stack, tree, etc..)

**The problem we are trying to solve:**

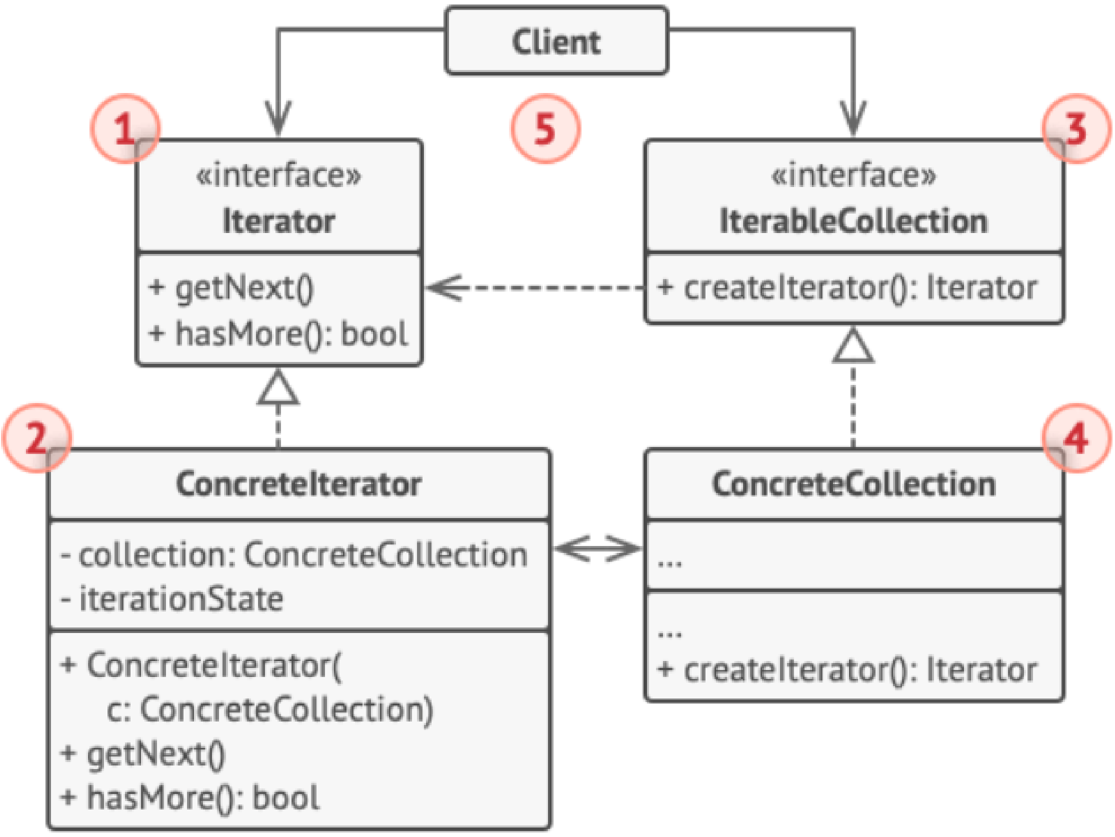
There should be a way to go through each element of the collection without accessing the same elements over and over. This may sound easy if you have a collection like a list. But how do you sequentially traverse elements of complex data structure such as a tree?



*The same collection can be traversed in several different ways.*

**Solution**

The main idea of the Iterator pattern is to extract the traversal logic/behaviour of a collection into a separate object called an iterator. The structure of the UML would look like something below.



1. The **Iterator interface** declares the operations required for traversing a collection.
2. **Concrete Iterators** implement specific algorithms for traversing a collection. The iterator object should track the traversal progress on its own. This allows several iterators to traverse the same collection independently of each other.
3. The **Collection** interface declares one or multiple methods for getting iterators compatible with the collection.
4. **Concrete Collections** return new instances of a particular concrete iterator class each time the client requests one.

## Factory

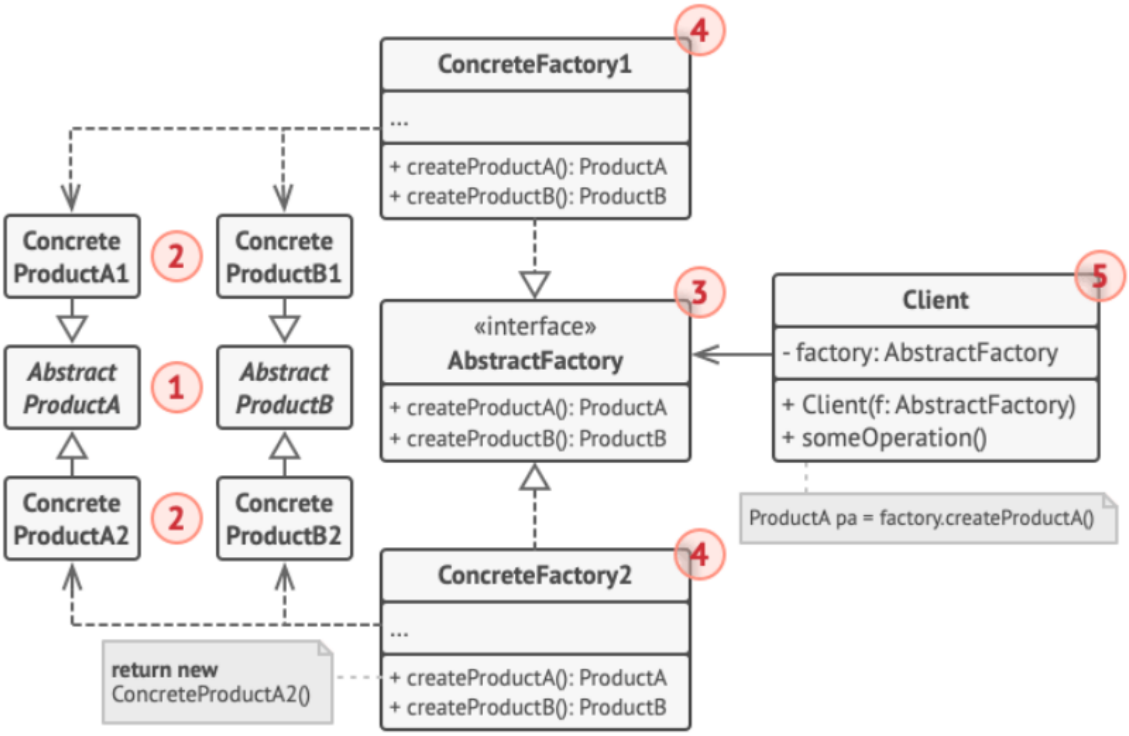
We primarily use the factory to minimize code duplication, by using an interface and being able to add new widgets quickly and easily.

TODO

https://refactoring.guru/design-patterns/factory-method

## AbstractFactory

**The purpose of the Abstract Factory is** to provide an interface for creating families of related objects, without specifying concrete classes**.**


4
ConcreteFactory1
...
+ createProductA(): ProductA
+ createProductB(): ProductB
Concrete ProductA1
2
Concrete ProductB1
Abstract ProductA
1
Abstract ProductB
Concrete ProductA2
2
Concrete ProductB2
3
«interface» AbstractFactory
+ createProductA(): ProductA
+ createProductB(): ProductB
5
Client
- factory: AbstractFactory
+ Client(f: AbstractFactory)
+ someOperation()
ProductA pa = factory.createProductA()
4
ConcreteFactory2
...
+ createProductA(): ProductA
+ createProductB(): ProductB
return new ConcreteProductA2()

https://sourcemaking.com/design_patterns/abstract_factory/java/1