1. Абстрактная фабрика

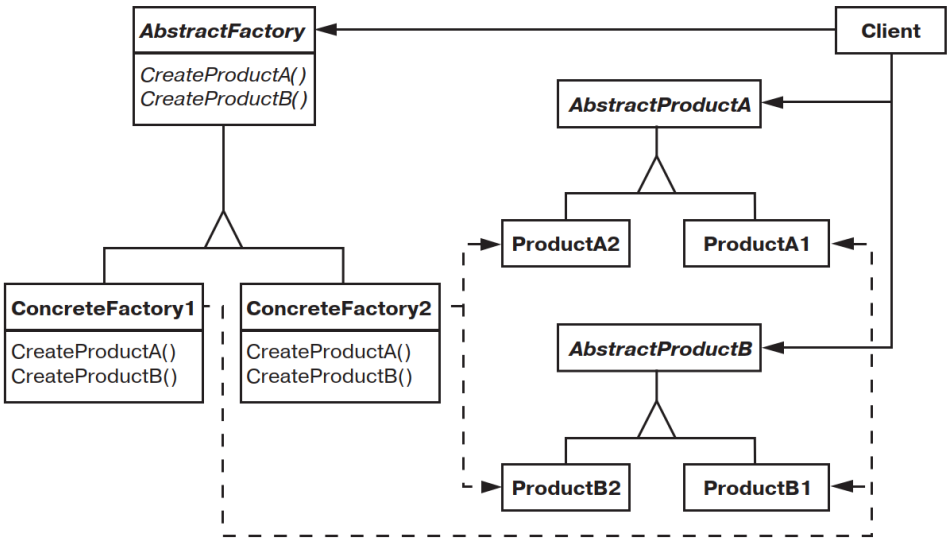
1.1 Название и классификация паттерна

Абстрактная фабрика — паттерн, порождающий объекты.

1.2 Назначение

Предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, не специфицируя их конкретных классов.

1.3 Структура



1.4 Участники

AbstractFactory — абстрактная фабрика:

• объявляет интерфейс для операций, создающих абстрактные объекты-продукты;

ConcreteFactory — конкретная фабрика:

• реализует операции, создающие конкретные объекты-продукты;

AbstractProduct — абстрактный продукт:

• объявляет интерфейс для типа объекта-продукта;

ConcreteProduct — конкретный продукт:

• определяет объект-продукт, создаваемый соответствующей конкретной фабрикой;

• реализует интерфейс AbstractProduct;

Client — клиент:

• пользуется исключительно интерфейсами, которые объявлены в классах AbstractFactory и AbstractProduct.

1.5 EO реализация шаблона

```
+package sandbox
+alias stdout org.eolang.io.stdout
+alias sprintf org.eolang.txt.sprintf

[type] > abstractFactory
  if. > concreteFactory
    eq.
      type
      "1"
    concreteFactory1
    concreteFactory2

  [] > createProductA
    createProductA. > @
      ^.concreteFactory
  [] > createProductB
    createProductB. > @
      ^.concreteFactory

[] > concreteFactory1
  [] > createProductA
    1 > @
  [] > createProductB
    2 > @

[] > concreteFactory2
  [] > createProductA
    "one" > @
  [] > createProductB
    "two" > @

[args...] > appAbstractFactory
  abstractFactory > objFactory
    args.get 0
  stdout > @
    sprintf
      "ProductA: %s\nProductB: %s\n"
      objFactory.createProductA
      objFactory.createProductB
```

Вывод программы

```
$ ./run.sh 1
ProductA: 1
ProductB: 2
$ ./run.sh 2
ProductA: one
ProductB: two
```

Данная программа создает объекты целые числа или строки в зависимости от параметра args[0]. Если args[0] = 1, то создадутся объекты 1 и 2, иначе – “one” и “two”.

Шаблон предполагает использования интерфейсов, который отсутствую в EO. В данном случае предпринята попытка реализации интерфейса через EO объект имеет параметр **`type`** в зависимости от которого выбирается конкретная реализация фабрики объектов. Это делает зависимым

объект-интерфейс, от набора реализаций этого интерфейса(при добавлении новой реализации необходимо внести изменения в объект интерфейс).

2. Singleton (одиночка)

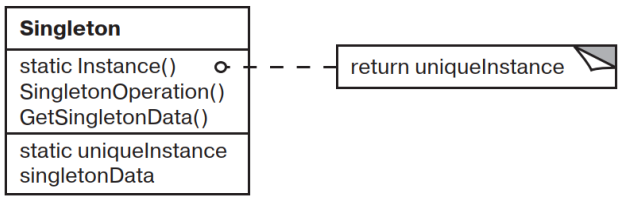
2.1 Название и классификация паттерна

Одиночка — паттерн, порождающий объекты

2.2 Назначение

Гарантирует, что у класса существует только один экземпляр, и предоставляет к нему глобальную точку доступа.

2.3 Структура



2.4 Участники

Singleton — одиночка:

• определяет операцию Instance, которая позволяет клиентам получить доступ к единственному экземпляру. Instance — это операция класса, то есть статический метод класса;

• может нести ответственность за создание собственного уникального экземпляра.

2.5 Отношения

Клиенты получают доступ к экземпляру класса Singleton только через его операцию Instance.

2.6 Java реализация

```
public final class Singleton {
    private static Singleton instance;
    public String value;

    private Singleton(String value){
        this.value = value;
    }

    public static Singleton getInstance(String value) {
        if (instance == null) {
```

```
            instance = new Singleton(value);
        }
        return instance;
    }
}
```

2.7 EO реализация

В EO отсутствуют классы поэтому этот шаблон не реализуем в чистом виде. Если же мы в терминах EO определим Singleton как объект у которого гарантировано существует только одна копия, то реализация этого объекта тоже невозможна по следующим причинам:

- В EO отсутствуют ссылки. Любое использование объекта в месте отличном от места определения есть копирование этого объекта.
- В EO отсутствуют возможности ограничения доступа к объектам и запрета его копирования. Невозможно ограничить создание копий объекта.

3. Прототип

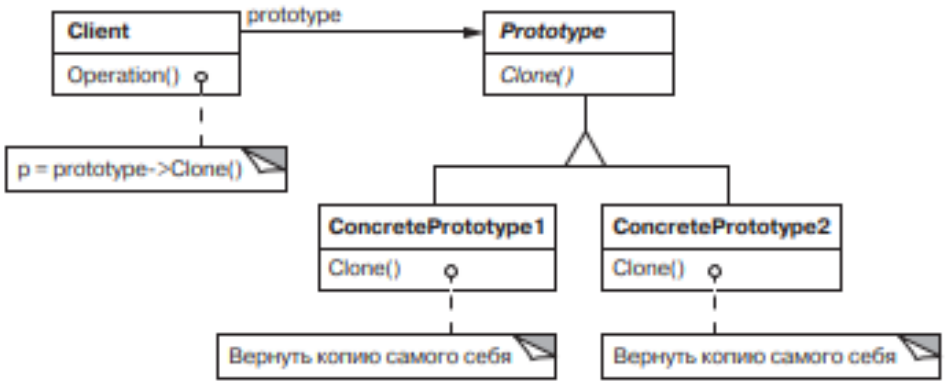
3.1 Название и классификация паттерна

Прототип — паттерн, порождающий объекты.

3.2 Назначение

Задает виды создаваемых объектов с помощью экземпляра-прототипа и создает новые объекты путем копирования этого прототипа.

Структура



3.3 Участники

— Prototype — прототип:

- объявляет интерфейс для клонирования самого себя;

— ConcretePrototype:

- реализует операцию клонирования себя;

— Client — клиент:

- создает новый объект, обращаясь к прототипу с запросом клонировать себя.

3.4 EO реализация

В EO каждый объект может быть скопирован, функции шаблона может выполнять каждый объект.

4. Мост(BRIDGE)

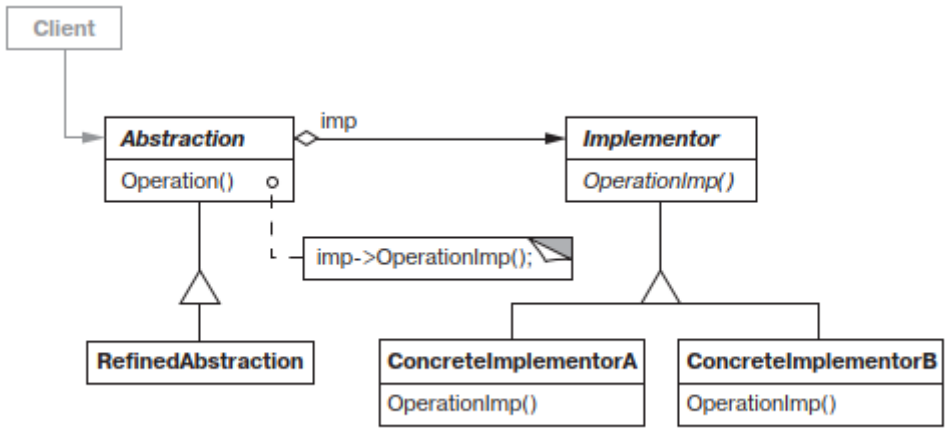
4.1 Название и классификация паттерна

Мост — паттерн, структурирующий объекты.

4.2 Назначение

Отделить абстракцию от ее реализации так, чтобы то и другое можно было изменять независимо.

Структура



4.3 Участники

Abstraction — абстракция:

- определяет интерфейс абстракции;
- хранит ссылку на объект типа Implementor;

RefinedAbstraction — уточненная абстракция:

- расширяет интерфейс, определенный абстракцией Abstraction;

Implementor — реализатор:

- определяет интерфейс для классов реализации. Он не обязан точно соответствовать интерфейсу класса Abstraction. На самом деле оба интерфейса могут быть совершенно различны. Обычно интерфейс класса Implementor предоставляет только примитивные операции, а класс Abstraction определяет операции более высокого уровня, основанные на этих примитивах;

ConcreteImplementor — конкретный реализатор:

- реализует интерфейс класса Implementor и определяет его конкретную реализацию.

4.4 Отношения

Объект Abstraction перенаправляет запросы клиента своему объекту Implementor.

4.5 EO реализация

```
+package sandbox
+alias stdout org.eolang.io.stdout
+alias sprintf org.eolang.txt.sprintf

[] > double
  [a] > execute
    add. > @
      a
      a

[a b] > perimeter
  double > doubleImpl
  [] > calculate
    add. > @
      ^.doubleImpl.execute
        ^.a
      ^.doubleImpl.execute
        ^.b

[args...] > appBridge
  stdout > @
    sprintf
      "%d"
      calculate.
        perimeter
          toInt.
            args.get 0
          toInt.
            args.get 1
```

Программа вычисляет периметр прямоугольника по длинам его сторон заданным в args[0] и args[1]. Объект `perimeter` выступает в качестве Abstraction, объект `double` – Implementor.

5. Цепочка обязанностей (CHAIN OF RESPONSIBILITY)

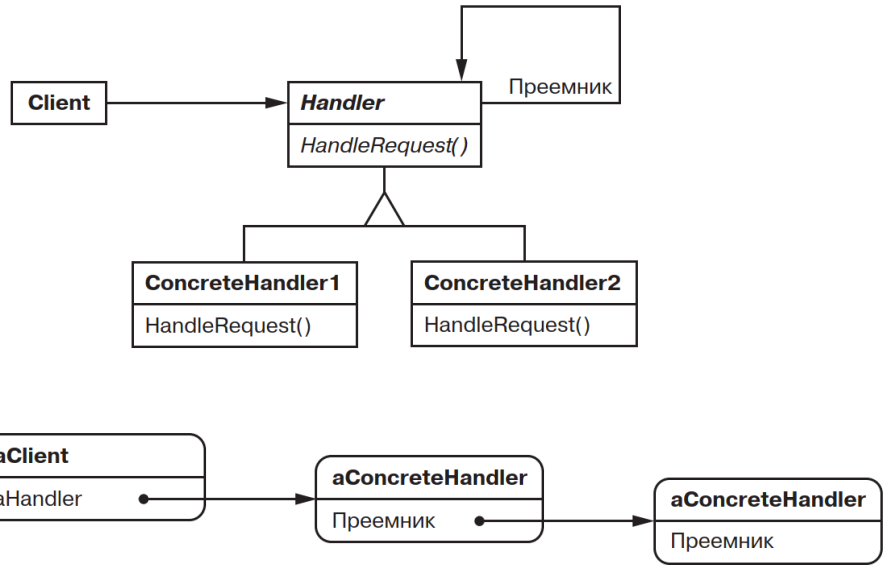
5.1 Название и классификация паттерна

Цепочка обязанностей — паттерн поведения объектов.

5.2 Назначение

Позволяет избежать привязки отправителя запроса к его получателю, предоставляя возможность обработать запрос нескольким объектам. Связывает объекты-получатели в цепочку и передает запрос по этой цепочке, пока он не будет обработан.

Структура



5.3 Участники

Handler — обработчик:

- определяет интерфейс для обработки запросов;
- (необязательно) реализует связь с преемником;

ConcreteHandler — конкретный обработчик:

- обрабатывает запрос, за который отвечает;
- имеет доступ к своему преемнику;
- если ConcreteHandler способен обработать запрос, то так и делает, если не может, то направляет его своему преемнику;

Client — клиент:

- отправляет запрос некоторому объекту ConcreteHandler в цепочке.

5.4 Отношения

Запрос, инициированный клиентом, продвигается по цепочке, пока некоторый объект ConcreteHandler не возьмет на себя ответственность за его обработку.

5.5 EO реализация

```
+package sandbox
+alias stdout org.eolang.io.stdout
+alias sprintf org.eolang.txt.sprintf

[nextHandler] > defaultHandler
  [message] > process
    "" > @

[] > handler1
  [message] > process
    if. > @
      message.eq "1"
      "one"
      ^.nextHandler.process message
  defaultHandler > @
    handler2

[] > handler2
  [message] > process
    if. > @
      message.eq "2"
      "two"
      ^.nextHandler.process message
  defaultHandler > @
    handler3

[] > handler3
  [message] > process
    if. > @
      message.eq "3"
      "three"
      ^.nextHandler.process message
  defaultHandler > @
    handler4

[] > handler4
  [message] > process
    if. > @
      message.eq "4"
      "four"
      ^.nextHandler.process message
  defaultHandler > @
    defaultHandler

[args...] > appChain
  handler1 > hChain
  stdout > @
```

```
sprintf
  "%s\n"
  hChain.process
    args.get 0
```

Входной параметр args[0] передается последовательно 4м обработчикам, каждый из которых обрабатывает свое значение(числа от 1 до 4 преобразуются в слова, если введен другой параметр возвращается пустая строка).

6. Команда(Command)

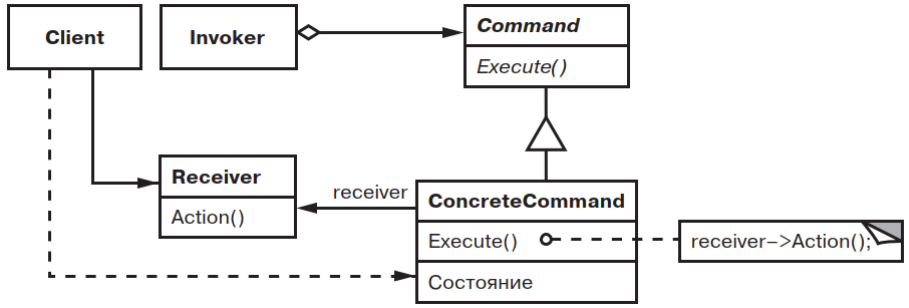
6.1 Название и классификация паттерна

Команда — паттерн поведения объектов.

6.2 Назначение

Инкапсулирует запрос в объекте, позволяя тем самым параметризовать клиенты для разных запросов, ставить запросы в очередь или протоколировать их, а также поддерживать отмену операций.

6.3 Структура



Участники

— Command — команда:

• объявляет интерфейс для выполнения операции;

— ConcreteCommand — конкретная команда:

• определяет связь между объектом-получателем Receiver и действием;

• реализует операцию Execute путем вызова соответствующих операций объекта Receiver;

— Client— клиент:

• создает объект класса ConcreteCommand и устанавливает его получателя;

— Invoker— инициатор:

• обращается к команде для выполнения запроса;

— Receiver — получатель:

• располагает информацией о способах выполнения операций, необходимых для удовлетворения запроса. В роли получателя может выступать любой класс.

6.4 Отношения

— клиент создает объект ConcreteCommand и устанавливает для него получателя;

— инициатор Invoker сохраняет объект ConcreteCommand;

— инициатор отправляет запрос, вызывая операцию команды Execute. Если поддерживается отмена выполненных действий, то ConcreteCommand перед вызовом Execute сохраняет информацию о состоянии, достаточную для выполнения отмены;

— объект ConcreteCommand вызывает операции получателя для выполнения запроса.

6.5 EO реализация

```
+package sandbox
+alias stdout org.eolang.io.stdout
+alias sprintf org.eolang.txt.sprintf

[receiver] > incCommand
  [] > execute
    add. > @
      ^.receiver
      1

[receiver] > decCommand
  [] > execute
    sub. > @
      ^.receiver
      1

[receiver] > doubleCommand
  [] > execute
    mul. > @
      ^.receiver
      2

[cmd1 cmd2 cmd3] > invoker

[args...] > appCommandObjects
  toInt. > receiver
    args.get
      0

  invoker > inv
    incCommand
      receiver
    decCommand
      receiver
    doubleCommand
      receiver

  stdout > @
    sprintf
      "%s - %s - %s"
      inv.cmd1.execute
      inv.cmd2.execute
      inv.cmd3.execute
```

Объект `invoker` последовательно выполняет 3 команды над объектом args[0](целое число):

- инкремент
- декремент
- удвоение

7. Наблюдатель(Observer)

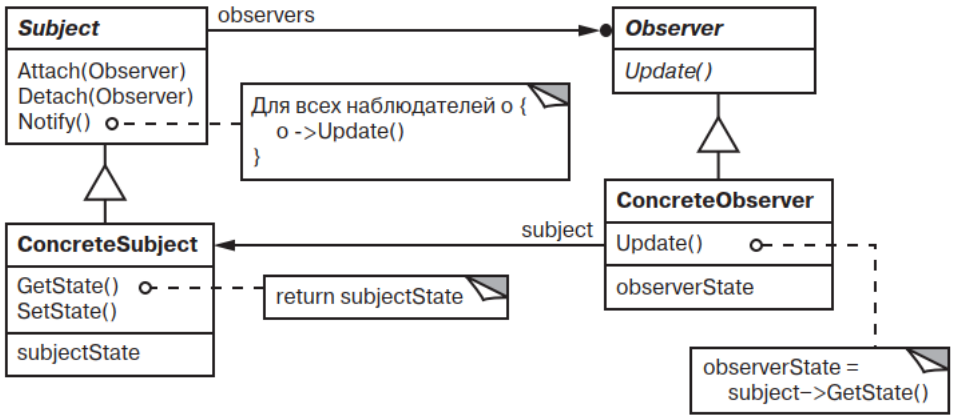
7.1 Название и классификация паттерна

Наблюдатель — паттерн поведения объектов.

7.2 Назначение

Определяет зависимость типа «один ко многим» между объектами таким образом, что при изменении состояния одного объекта все зависящие от него оповещаются об этом и автоматически обновляются.

7.3 Структура



7.4 Участники

Subject — субъект:

• располагает информацией о своих наблюдателях. За субъектом может «следить» любое число наблюдателей;

• предоставляет интерфейс для присоединения и отделения наблюдателей;

Observer — наблюдатель:

• определяет интерфейс обновления для объектов, которые должны уведомляться об изменении субъекта;

ConcreteSubject — конкретный субъект:

• сохраняет состояние, представляющее интерес для конкретного наблюдателя ConcreteObserver;

• посылает информацию своим наблюдателям, когда происходит изменение;

ConcreteObserver — конкретный наблюдатель:

• хранит ссылку на объект класса ConcreteSubject;

• сохраняет данные, которые должны быть согласованы с данными субъекта;

• реализует интерфейс обновления, определенный в классе Observer, чтобы поддерживать согласованность с субъектом.

7.5 ЕО реализация

В EO все объекты имеют неизменяемое состояние. Исходя из назначения шаблона, его применение в EO бессмысленно.

8. Посредник(Mediator)

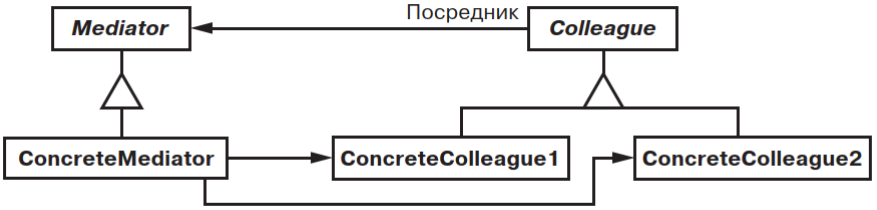
8.1 Название и классификация паттерна

Посредник — паттерн поведения объектов.

8.2 Назначение

Определяет объект, инкапсулирующий способ взаимодействия множества объектов. Посреднr обеспечивает слабую связанность системы, избавляя объекты от необходимости явно ссылаться друг на друга и позволяя тем самым независимо изменять взаимодействия между ними.

8.3 Структура



8.4 Участники

Mediator — посредник:

• определяет интерфейс для обмена информацией с объектами Colleague;

ConcreteMediator — конкретный посредник:

• реализует кооперативное поведение, координируя действия объектов Colleague;

• владеет информацией о коллегах и подсчитывает их;

Классы Colleague — коллеги:

• каждый класс Colleague знает свой объект Mediator;

• все коллеги обмениваются информацией только с посредником во всех случаях, когда ему пришлось бы напрямую взаимодействовать с другими объектами.

8.5 Отношения

Коллеги посылают запросы посреднику и получают запросы от него. Посредник реализует кооперативное поведение путем переадресации каждого запроса подходящему коллеге (или нескольким коллегам).

8.6 EO реализация

Реализацией данного шаблона в EO может быть структура, в которой объект-медиатор либо имеет атрибут массив объектов-коллег, либо его атрибутами напрямую являются объекты-коллеги. Помимо это объект-медиатор содержит атрибуты, которые несут функцию обмена сообщениями между объектами коллегами:

```
[coll1 coll2 coll3] > mediator
  [] > function1
  [] > function2
  [] > function3
```

9. The Builder Pattern

9.1. Name and Classification of the Pattern

The Builder Pattern is a creational object-oriented design pattern.

9.2. Purpose

The pattern is used to create complex objects in a stepwise manner with a varying configuration of the steps.

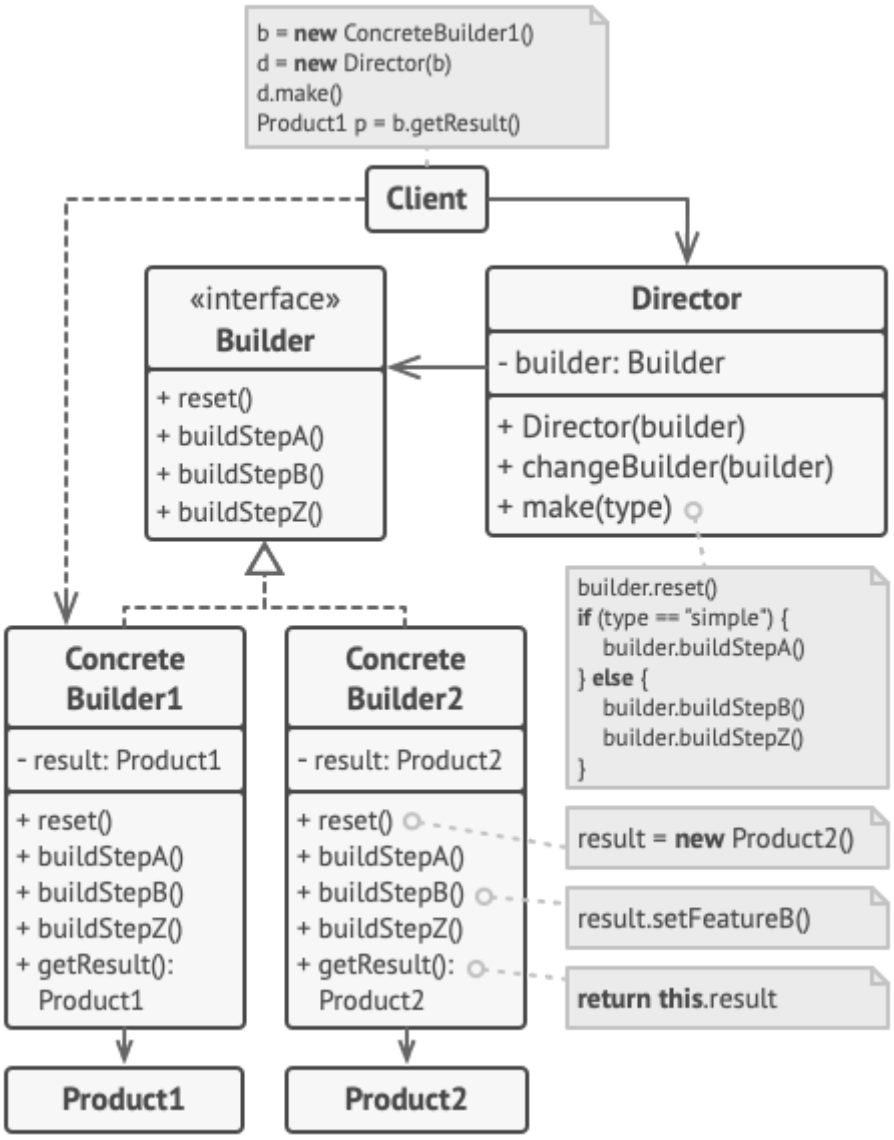
9.3. Problem

Suppose, we have a class with a variety of input parameters. The input parameters are used to configure instances of the class. Some of the parameters may be optional, while some of them are mandatory to be sat up. Hence, the following techniques of configuration of instances of the class may be applied:

1. Configuration of instance variables of an object directly in the user code through Setter Methods calls or by referencing variables straightforwardly. This practice may not be considered appropriate because it makes code instances more cohesive and interdependent while violating encapsulation of the inner state of objects (which may lead to breaking of the integrity of business logic of an application), and, hence, the usage of the practice is not encouraged. In addition, this technique may allow situations in which objects are being in an incomplete state, which also may break the logic of an application.
2. Creation of subclasses of the considered class when each successor has a slightly changed prototype of its constructors. This technique implies that prototypes of constructors of different subclasses have subsets of optional parameters in them while omitting some parameters, which makes it possible to create configurable instances of objects in a controlled manner. This practice is more encouraged to be applied in practice since it implies control over the creational process. However, it is not recommended for use when choosing the sole parent superclass is challenging or when the practice produces a wide or deep hierarchy of inheritance.
3. Overloading of constructors or setting a single constructor with optional parameters. While this practice allows classes to create instances in a controlled way, it is undesirable in cases where the number of parameters or constructor overloads is too large to be manageable and understandable.

The Builder pattern may be considered a universal solution to the problem. The pattern defines the Builder class, which has methods (stages) for building objects. The user code can call the stages in any order, omitting some of them (optional configurations). Also, the Builder class may check that all the required parameters are set up. At the end of construction, user code is required to call a method that finishes the construction process and returns a ready-made object. The pattern encapsulates the creational sensitive logic inside the Builder class and makes the configuration process manageable to the user code.

9.4. Structure



9.5. Code Instances Involved

Builder is an abstract class that defines the contract of the creational steps of ***Products*** for its successors (concrete builders). Also, the **Builder** superclass defines the finizaliotion method.

Product is an interface for products (instances being created and configured through the Builder pattern). The interface defines the contract for all products so that these may be managed by Builders.

(optional) ***Director*** is a class, which defines higher-level (that is, higher than the level of "understanding" of the builder itself, for example, rules for mandatory fields and compliance with business logic) scripts for building objects. The director can be used to reuse some high-level business logic for constructing objects based on various builder implementations.

9.6. Relations

Implementations of the ***Product*** interface are ***products***. Inheritors of the **Builder** class provide concrete implementations for the building steps (or borrow some of those steps from the superclass). The ***Director*** (optional entity) class can manage builders in a general style (based on some configuration) in

accordance with the higher-level logic of business rules. The client code can contact the Director, giving it the configuration, or build an object using the **Builder** directly.

9.7. EO Implementation

First, we should mention that the problem solved by the Builder pattern may be addressed by the partial application mechanism embedded into the language as one of its features. The partial application mechanism allows programmers to partially apply objects (i.e., create objects with some or all of the input attributes omitted and then, optionally, set unbound attributes after throughout the program). This technique may be utilized as a more concise alternative to constructor overloading. Here is an example:

```
[a b c name] > triangle
  add. > perimeter
    add.
      a
      b
    c

  sprintf > toString
    "The triangle is named '%s'."
    name


[args...] > app
  triangle > triangleA
    10:a
  triangleA > triangleABC
    7:b
    8:c
  triangleABC > triangleABCNamed
   "My Triangle":name
  triangle > triangleNamed
    "My Another Triangle":name
```

Here, we have the **triangle** object with input attributes **a**, **b**, **c**, and **name**. The triangle object has two bound attributes: ***perimeter*** (which relies on **a**, **b**, and **c**) and ***toString*** (which relies on **name**). Object ***app*** demonstrates the partial application mechanism. So, ***triangleA*** has only the **a** attribute bound, ***triangleABC*** has all the sides (**a**, **b**, **c**) sat up, ***triangleABCNamed*** has all the ***sides*** and its **name** configured, and ***triangleNamed*** has the **name** only. All three triangles are constructed through partial application (meaning, some of the attributes are left unbound or were bound after). The above example demonstrates an alternative solution to the problem of optional configuration of objects. However, this solution does not encapsulate the creation process of objects. Hence, the **Builder** pattern still may have its place in the EO environment.

Consider the following example:

```
[] > builder
  subbuilder triangle > @
  [triangleEntity] > subbuilder
    # finalizes the construction process
    [] > finalize
```

```
      ^.^.subbuilder > @
    # configures the a free attribute
      [aVal] > setA
        ^.^.subbuilder > @
          ^.^.triangleEntity
            (^.validateSide aVal):a
      # configures the b free attribute
      [bVal] > setb
        ^.^.subbuilder > @
          ^.^.triangleEntity
            (^.validateSide bVal):b
      # configures the c free attribute
      [cVal] > setc
        ^.^.subbuilder > @
        ^.^.triangleEntity
            (^.validateSide cVal):c
      # configures the name free attribute
      [nameVal] > setname
        ^.^.subbuilder > @
          ^.^.triangleEntity
            (^.validateName nameVal):name
      # validates side value
      [val] > validateSide
        if. > @
          val.greater 0
          val
          error
            "The side of a triangle must not be less than 1!"
      # validates name
      [val] > validateName
        if. > @
          val.length.neq 0
          val
          error
            "The name of a triangle must not be empty!"

  [args...] > app
    builder > b
    finalize. > triangleABC
      setC.
        setB.
          setA.
            builder
            10
          12
        0
```

Here we implemented the principles of the **Builder** pattern through measures supplied by the EO language. The **builder** object contains the ***subbuilder*** attribute object that implements the construction steps (***setA***, ***setB***, ***setC***, ***setName***) as well as validation sub-steps (***validateSide***, ***validateName***) and the ***finalize*** attribute that finishes the construction process and returns the resulting object. Initially, the instantiation of the **builder** object is substituted with a copy of the ***subbuilder*** object with an empty

(meaning, all free attributes are unbound) copy of the constructing object. On each construction step, the ***subbuilder*** object returns itself by passing a changed version of the constructing object to its constructor. The construction steps have validation substeps that may implement some complex business logic. Validation steps may return an error or a validated object, which may lead to an interruption of the program execution and prevent inconsistency of the business logic.

In conclusion, we would like to notice that the problem originally stated above (problem of optional configuration of objects with a lot of input parameters) and solved with the **Builder** pattern may not be actual to EO since it has the partial application mechanism that allows programmers to perform such configuration and, in addition, EO does not allow objects to have more than four free attributes (although, this restriction may be mitigated through passing complex object structures as free attributes). Nevertheless, the EO implementation of the Builder pattern may find its utilization in scopes where encapsulation of the creational process of objects is required. For instance, it may be needed when business logic validation of values passed for binding to free attributes is required.

10. Factory Method

10.1. Name and Classification of the Pattern

The Factory Method Pattern is a creational object-oriented design pattern.

10.2. Purpose

Defines the creational method in the **Creator** superclass that defines a rule (that is, an interface or a contract) for creating an object (product) of some supertype **Product**. This method is used by the superclass or its more specific implementations, and the factory method can also be called from outside the class by other entities within the application. Concrete implementations of the **Creator** class with a factory method can return subtypes of the **Product** type, thereby "tailoring" a specific implementation of the product class to the one required by the factory method contract. Hence, the pattern allows programmers to implement seamless configuration of the architecture of the application.

10.3. Problem

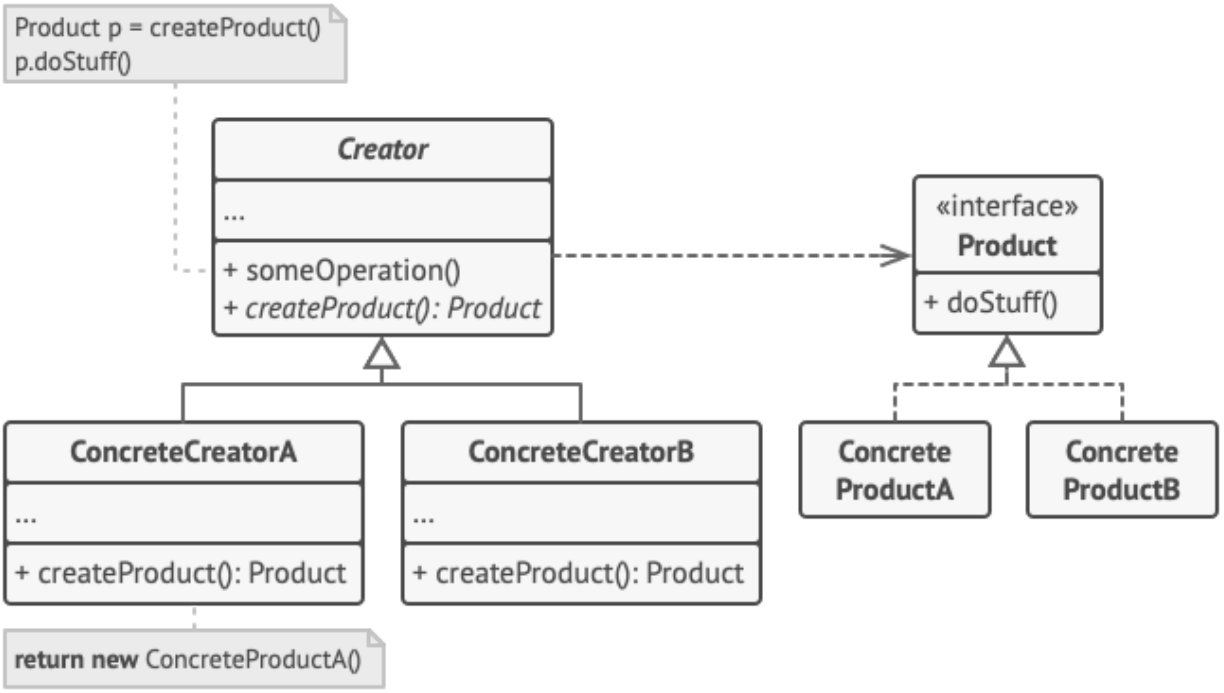
The pattern addresses the problem of extending the architecture of an application. By specifying the product contract (**Product** Interface) and by defining the class contract with the Factory Method Class, the architect separates the responsibility for creating the product itself from other methods of the creator class. This can be useful when:

1. It is not known what types of the **Product** class may be used in the future, but it may be appropriate to leave a headroom for a potential extension of the application architecture. Otherwise, this can be interpreted as an implementation of the "Open / Closed” principle (O in SOLID).

2. Implementation of the principle of "Single Responsibility" (S in SOLID). The code responsible for setting (configuring) a specific version of the product can be placed in a single place, for example, in a class that configures the application based on the environment settings. Here, the Dependency Injection mechanism can also be used to perform such a configuration in an invisible manner.

3. The pattern allows programmers to separate the logic of product creation from other logic of the creator class. This facilitates the reuse of identical code.

10.4. Structure



10.5. Code Instances Involved

Creator is an abstract class that defines the contract of the reutilized steps (here, it is **someOperation**) and the creational step (**createProduct**) of ***Products*** for its successors (concrete creators).

Product is an interface for products (instances being created and configured through the Factory Method pattern). The interface defines the contract for all products so that these may be managed by the pattern.

10.6. Relations

Implementations of the ***Product*** interface are ***products***. Inheritors of the **Creator** class provide concrete implementations for the creational method and inherit the rest methods. The concrete implementation of the creational method may return different implementations of Product, which implies the substitution of logic (or configurability of the application).

10.7. EO Implementation

The EO programming language does not have interfaces, classes, and types. Because of it, we may omit defining the Product interface contract (since it would not impose any requirements). Consider the following implementation of the pattern in EO:

```
[] > creator
  # left to be redefined
  [] > createObject
  # operation over products
  [] > performOperation
    createObject.getWeight.add 1 > @

[] > concreteCreatorA
```

```
  creator > @
  [] > createObject
    productA > @

[] > concreteCreatorB
  creator > @
  [] > createObject
    productB > @

[] > productA
  # let's suppose that this implementation gets value from the
production server
  [] > getWeight
    42 > @

[] > productB
  # let's suppose that this implementation gets value from the
testing server
  [] > getWeight
    24 > @
```

Here, we have the **creator** object with the ***performOperation*** attribute (the logic to be reused) and the ***createObject*** attribute (the logic to be redefined for flexible substitution of objects). The **concreteCreatorA** and **concreteCreatorB** objects have the creator object as their decoratee, so that these might inherit the reusable logic. Both objects define the **createObject** attribute that hides the original attribute of the same name from the decoration hierarchy. Objects **productA** and **productB** implement the attribute of interest (**getWeight**) differently. One of them may get the value from the production server, while another takes it from the testing environment. This example demonstrates the implementation of the classic version of the pattern in EO. However, we may consider a more EO-idiomatic example, free from additional structures (concrete creators) utilized in statically typed class-based object-oriented languages such as Java or C++. Consider the following example:

```
[@] > creator
  # operation over products
  [] > performOperation
    createObject.getWeight.add 1 > @

[] > productA
  # let's suppose that this implementation gets value from the
production server
  [] > getWeight
    42 > @

[] > productB
  # let's suppose that this implementation gets value from the
testing server
  [] > getWeight
    24 > @

[args...] > app
  creator > creatorObject
```

```
[]
  [] > createObject
    if. > @
      (args.get 0).eq "test"
      productB
      productA
```

Here, we used the technique of passing decoratee as a free attribute of the object **creator**. Its decoratee is passed in the **app** object. The decoratee has the only attribute **createObject** that the **creator** object inherits and relies on. The **createObject** attribute decides what version of a product should be chosen based on the environment configuration. This implementation of the pattern may be considered as more idiomatic and flexible from the EO perspective.

11. The Closures Functional Programming Technique

Since the EO programming language may be considered semi-functional, it might be useful to apply one of the widely adopted functional programming techniques, closures, in it. Simply put, the closures mechanism implies capturing outer lexical scope variables inside a function defined inside the scope with a consequent utilization of the function in other scopes. To support this technique, a language must operate over function as if they are first-class citizens (i.e., a language must return function or pass functions as parameters). Here is an example of this technique in JavaScript:

```
function makeAdder(x) {
  return function(y) {
    return x + y;
  };
}
var add5 = makeAdder(5);
var add10 = makeAdder(10);
console.log(add5(2));  // 7
console.log(add10(2)); // 12
```

Here, we have the **makeAdder** function that returns an anonymous function capturing its outer state **x**. The state is then utilized when the returned function is applied with its own parameter **y**. In other words, the inner anonymous function remembers the value of **x** and uses it even when the actual value disappeared from the stack. This technique may be useful to emulate access modificators in functional languages:

```
var counter = (function() {
  var privateCounter = 0;
  function changeBy(val) {
    privateCounter += val;
  }

  return {
    increment: function() {
      changeBy(1);
    },

    decrement: function() {
```

```
      changeBy(-1);
    },

    value: function() {
      return privateCounter;
    }
  };
})();

console.log(counter.value());  // 0.

counter.increment();
counter.increment();
console.log(counter.value());  // 2.

counter.decrement();
console.log(counter.value());  // 1.
```

Here, the outer function **counter** returns a complex object-like structure containing functions that capture the state of the **counter** function. The state of the **counter** function is also complex: it has a mutable local variable **privateCounter**, and the **changeBy** function that mutates the value in the unified manner. The user code may not access the value and the mutating functions directly: both of them disappeared from the stack. However, the closures returned by the outer function still may do it. Hence, the technique allows functional programmers to simulate private state.

We surely may reproduce the similar technique of capturing the lexical scope in EO:

```
[] > counter
  memory 0 > privateCounter
  [val] > changeBy
    privateCounter.write > @
      privateCounter.add val
  [] > @
    [] > increment
      ^.^.changeBy 1 > @
    [] > decrement
      ^.^.changeBy (-1) > @
    [] > value
      ^.^.privateCounter > @
```

However, EO has no local variables or any kind of stack-lifetime storage. Instead, any name refers to an object (stored in heap) that may be accessed through the scope of any other object via the dot-notation mechanism. Even anonymous objects may allow programmers to access its local scope (including parent and decoration hierarchies) freely. In addition, EO has no access modification instruments. This makes closures technique almost similar to the partial application mechanism. Moreover, the publicity of any attribute of any object makes encapsulation impossible in the language. This differentiates EO from functional programming languages and, also, from object-oriented languages. Absence of instruments of access modification (or simulation of it) may be a severe violation of object-oriented principle of encapsulation, which may lead to insecure environments breaking business logic of problem domains.