Абстрактная фабрика

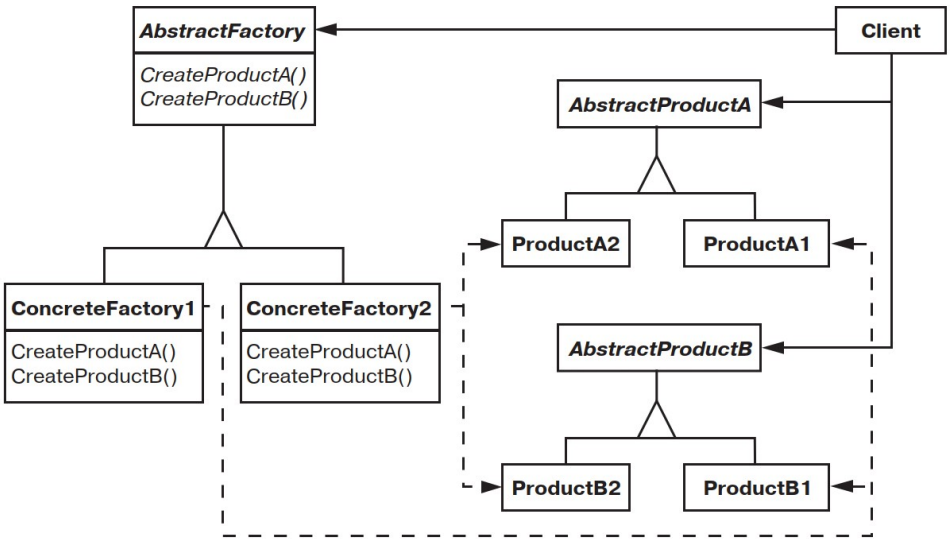
Название и классификация паттерна

Абстрактная фабрика — паттерн, порождающий объекты.

Назначение

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

Структура



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.

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).

Фабричный метод (Factory Method)

Название и классификация паттерна

Фабричный метод (Factory Method) — порождающий (creational) паттерн проектирования.

Назначение

Определяет в суперклассе метод, который задает правило (то есть интерфейс или контракт) создания объекта (продукта) некоторого супертипа Product. Этот метод используется суперклассом или его более конкретными реализациями, а также фабричный метод может вызываться извне класса другими сущностями внутри приложения. Конкретные реализации класса с фабричным методом могут возвращать подтипы типа Product, тем самым «подгоняя» конкретную реализацию класса-продукта под ту, что требует контракт фабричного метода.

Решаемая проблема

Паттерн решает проблему расширения архитектуры приложения. Задав контракт продукта (Product Interface) и определив контракт класса с фабричным методом (Factory Method Class) архитектор разделяет ответственность по созданию самого продукта от прочих методов класса-создателя.

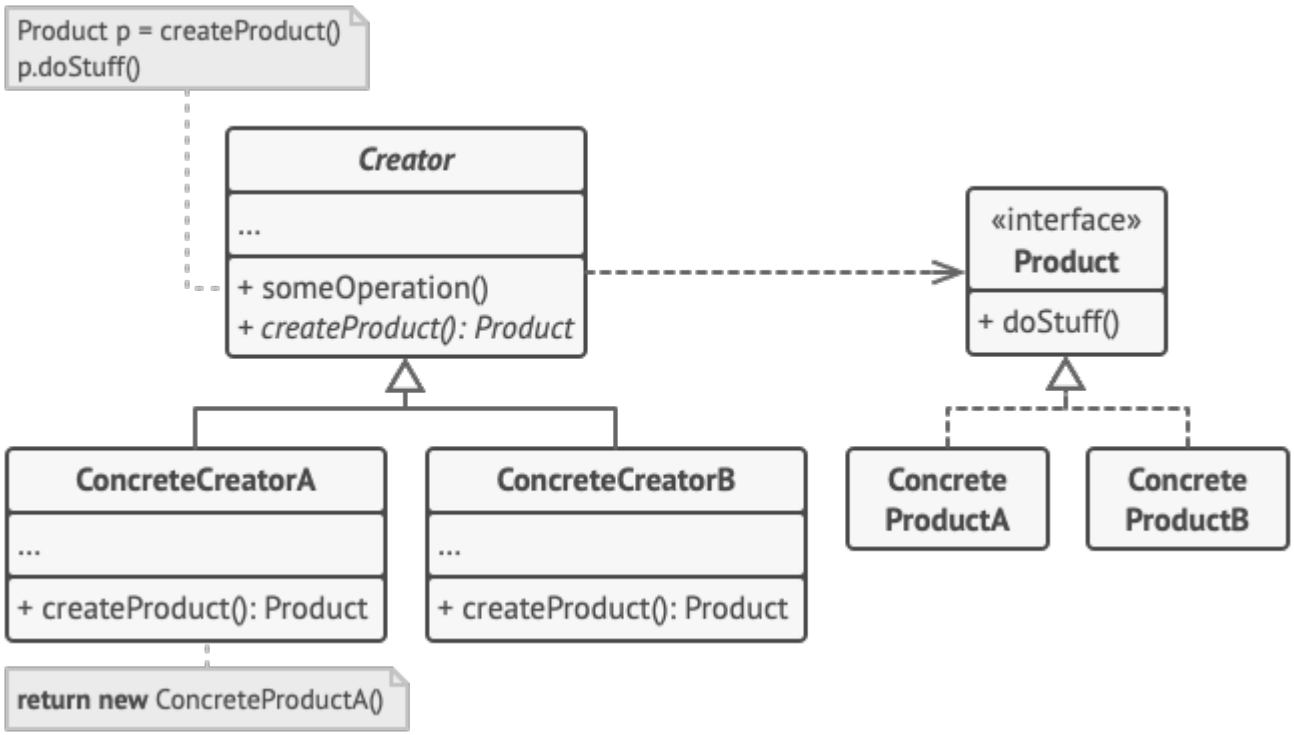
Это может быть полезно в случаях, когда:

1. Не известно, какие типы класса-продукта могут быть использованы в будущем, но при этом может быть уместно оставить задел для потенциального безболезненного расширения архитектуры приложения. Иначе этот пункт можно трактовать как реализацию принципа «Open/Closed Principle».

2. Реализация принципа «Single Responsibily Principle». Код, ответственный за задание (конфигурирования) конкретной версии продукта может быть вынесен в единственное место, например в класс, который осуществляет конфигурацию приложения на основе настроек среды. Здесь же может быть задействован механизм Dependency Injection для более незаметного выполнения такой конфигурации.

3. Паттерн позволяет отделить логику создания продукта от другой логики класса-создателя. Это способствует переиспользованию идентичного кода.

Структура



Участники

Creator — супертип класса создателя. Задает контракт для создания продукта. При этом имеет другие операции, которые полезно переиспользовать в различных реализациях.

Product — интерфейс класса-продукта. Задает контракт для самого продукта, то есть ту форму, которую продукт должен принять, чтобы называться продуктом.

Отношения

Реализации интерфейса Product являются продуктами. Наследники класса-создателя в фабричном методе возвращают соответствующие реализации продукта, тем самым совершая подмену. Остальные методы класса-создателя (по-возможности) наследуются без изменения. Паттерн, таким образом, допускает конфигурируемость вариативного кода равно как и переиспользование единого для всех кода.

Реализация на EO

Мысли:

1. В EO нет типов и интерфейсов. Сразу же отпадает необходимость в большинстве паттернов. Мотивация паттернов в строго типизируемых языках не столь же ясна при рассмотрении проблемы в бестиповом/квазитипизированном языке EO. Строгое соответствие паттерна обнаружить не удастся.

2. С другой стороны, одна (**главная**) из решаемых паттерном проблем не зависит от типов: отделение логики создания продукта от переиспользуемой логики. Гибкая система типов EO, а также отчасти функциональная природа языка позволяют нам переосмыслить паттерн по-новому, что будет показано после подготовки единого для всех трех языков примера.

WIP — добавить на следующей неделе единый пример.

Реализация на Java

WIP — добавить на следующей неделе единый пример.

Реализация на C++

WIP — добавить на следующей неделе единый пример.

Строитель (Builder)

Название и классификация паттерна

Строитель (Builder) — порождающий (creational) паттерн проектирования.

Назначение

Позволяет создавать комплексные объекты пошагово, тем самым конфигурируя вариативную часть объектов.

Решаемая проблема

Паттерн решает проблему создания комплексных конфигурируемых объектов. Проблема имеет и другие решения:

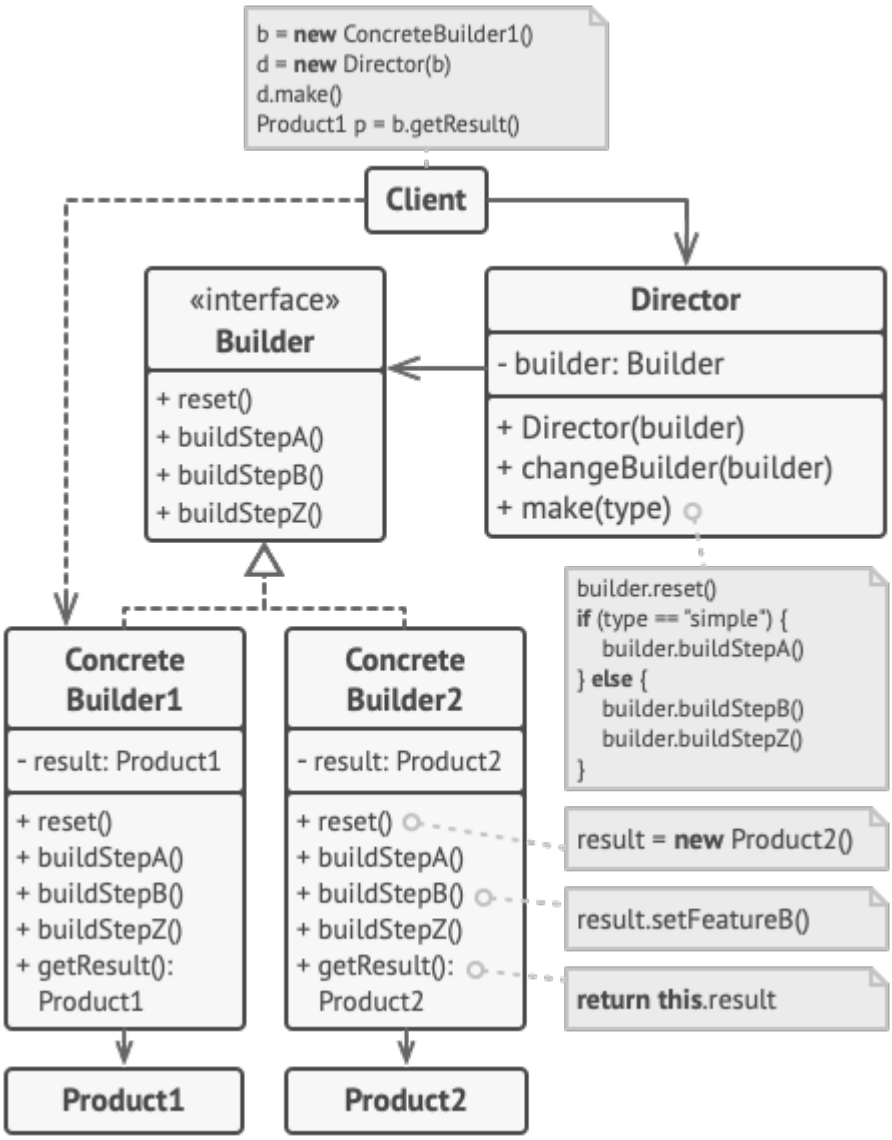
1. Конфигурировать объект непосредственно в клиентском коде, то есть вызывать Setter Methods или напрямую обращаться к полям объкта. Практика считается нежелательной, поскольку увеличивает связность между объектами и нарушает инкапсуляцию внутреннего состояния объекта (и, тем самым, покушается на целостность его бизнес-логики).

2. Создавать подклассы базового класса с частично опущенной/измененной конфигурацией в каждом из наследников. Практика также может быть нежелательной в тех случаях, когда выделение единого родителя может быть затруднительно или когда практика порождает слишком широкую (и/или глубокую) иерархию наследования.

3. Перегрузка конструкторов либо задание единого конструктора с опциональными параметрами. Практика считается нежелательной в случаях, когда число параметров или перегрузок конструктора слишком велико для простого понимания и использования конструктор(а/ов).

Паттерн задает класс Строителя, который имеет методы (этапы) строительства объектов. Пользовательский код может вызвать этапы в любом порядке, опустив часть из них (часть может быть обязательной — строитель может за этим следить). В конце строительства пользовательский код вызывает метод, возвращающий готовый объект.

Структура



Участники

Builder — супертип класса создателя. Задает контракт для создания продукта: то есть методы (шаги), с помощью которых конфигурируется создаваемый объект. Кроме того, имеет метод для завершения строительства с получением результата.

Product — интерфейс класса-продукта. Задает контракт для самого продукта, то есть ту форму, которую продукт должен принять, чтобы называться продуктом.

(опционально) Director — класс директора, который задает более высокоуровневые (то есть выше, чем уровень «понимания» самого строителя, например, обязательность полей и соответствие бизнес-логике) скрипты построения объектов. Директор может использоваться для переиспользования какой-то высокоуровневой бизнес-логики построения объектов на основе различных реализаций строителя.

Отношения

Реализации интерфейса Product являются продуктами. Наследники класса Builder предоставляют конкретные реализации для этапов строительства (или заимствуют часть этих шагов у родителя). Директор (опциональная сущность) может «дерижировать» строителями в общем стиле (на основе какой-либо конфигурации) в соответствии с более высокоуровневой логикой бизнес-правил. Клиентский код может обращаться к Директору, отдав ему конфигурацию, либо строить объект с помощью Строителя самостоятельно.

Реализация на EO

Мысли:

1. С одной стороны, принцип Partial Application уже решает проблему естественным для языка способом. Так, частичная аппликация позволяет задавать лишь часть свободным атрибутов объекта и дозавать остальные после. При этом каждое дозадание формирует копию объекта (что соответствует принципу неизменяемости объектов в EO). Таким образом, можно сказать, что в простых случаях, паттерн Builder может быть опущен и заменен на частичную аппликацию объектов.

2. Однако, с другой стороны, это не решает проблемы создания комплексных объектов с точки зрения разделения сложности создания и внутренней сложности сАмого клиентского кода.

WIP — добавить на следующей неделе единый пример.

Реализация на Java

WIP — добавить на следующей неделе единый пример.

Реализация на C++

WIP — добавить на следующей неделе единый пример.

Singleton (одиночка)

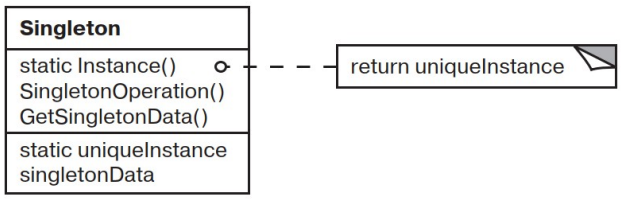
Название и классификация паттерна

Одиночка — паттерн, порождающий объекты

Назначение

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

Структура



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.

Отношения

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

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.

Прототип

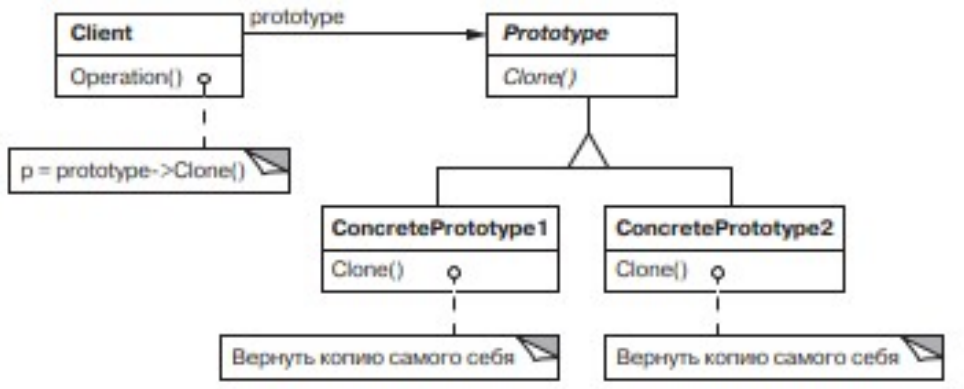
Название и классификация паттерна

Прототип — паттерн, порождающий объекты.

Назначение

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

Структура



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.

ЕО реализация

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

Мост(BRIDGE)

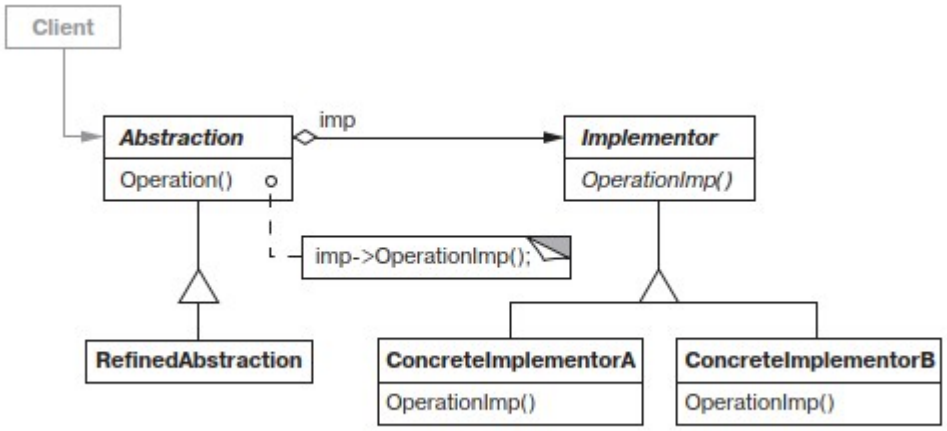
Название и классификация паттерна

Мост — паттерн, структурирующий объекты.

Назначение

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

Структура



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.

Отношения

Объект Abstraction перенаправляет запросы клиента своему объекту Implementor.

Цепочка обязанностей (CHAIN OF RESPONSIBILITY)

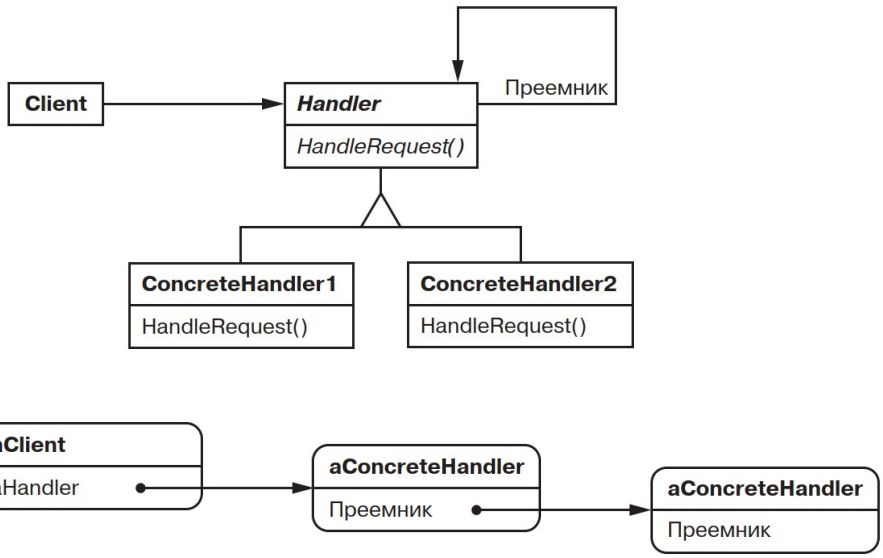
Название и классификация паттерна

Цепочка обязанностей — паттерн поведения объектов.

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



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.

Отношения

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

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).

Команда(Command)

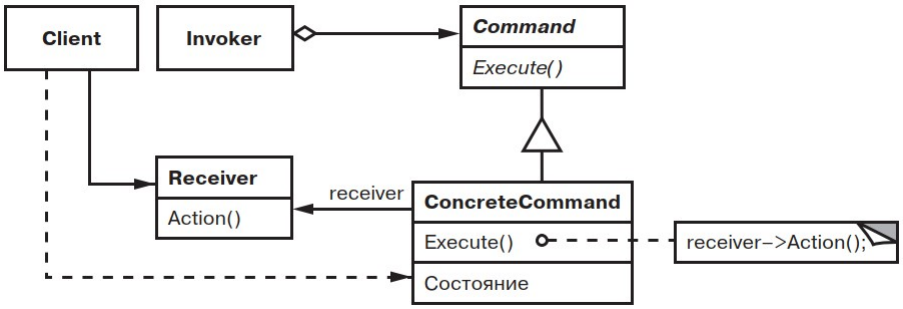
Название и классификация паттерна

Команда — паттерн поведения объектов.

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.

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.

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.

https://docs.google.com/spreadsheets/d/1HQa2ViBNOy1Zz9nTr82aWvk3nwbsQd5RSwF349EZT8k/edit#gid=0