ОБРАЗЦЫ ПРОЕКТИРОВАНИЯ

Д. Мигинский

Образец проектирования

Образец (шаблон) проектирования – повторно используемое решение типичной проблемы проектирования.

Состоит из*:

- Имени
- (Абстрактной) формулировки задачи, для решения которой применим шаблон
- (Абстрактного) решения – описание элементов дизайна, их поведения и отношений между ними
- Результаты – последствия применения образца, побочные эффекты, в т.ч. нежелательные

*Э. Гамма и др., Приемы объектно-ориентированного проектирования: шаблоны проектирования

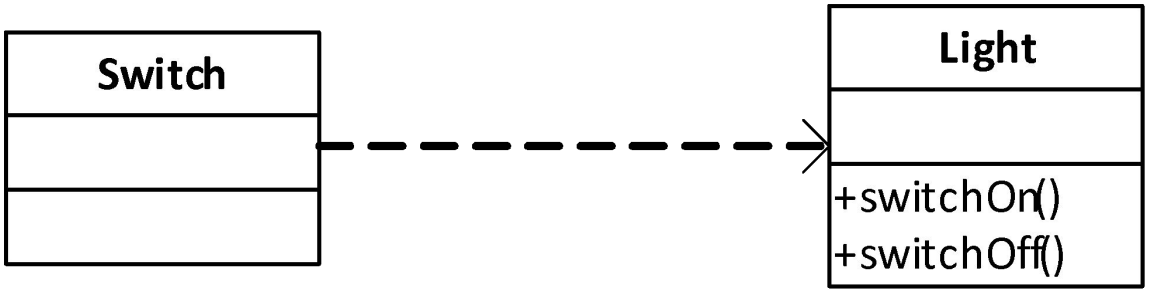
Классификация образцов

- Структурные – решают задачи, связанные с отношением между классами (или другими сущностями)
- Поведенческие – решают задачи поведения классов
- Порождающие – решают задачи создания экземпляров классов и их инициализации

Базовые структурные образцы

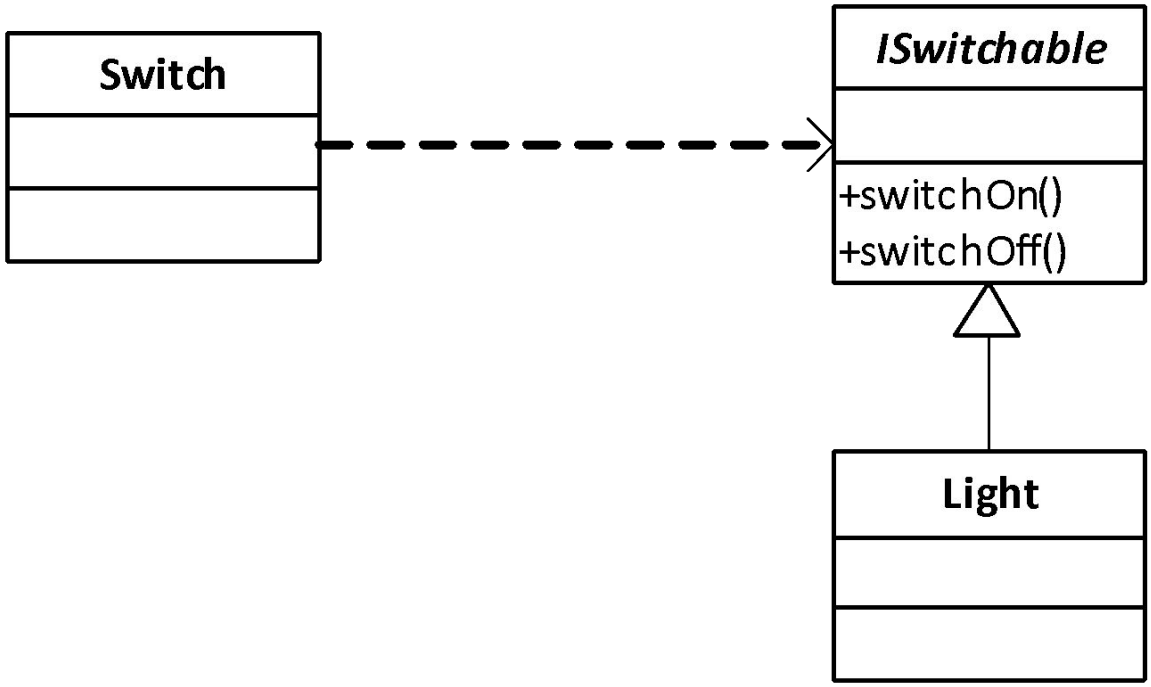
- Abstract Server
- Adapter
- Proxy

Abstract Server: пример задачи



Проблема: выключатель нельзя использовать для утюга и других приборов.

Abstract Server: пример решения

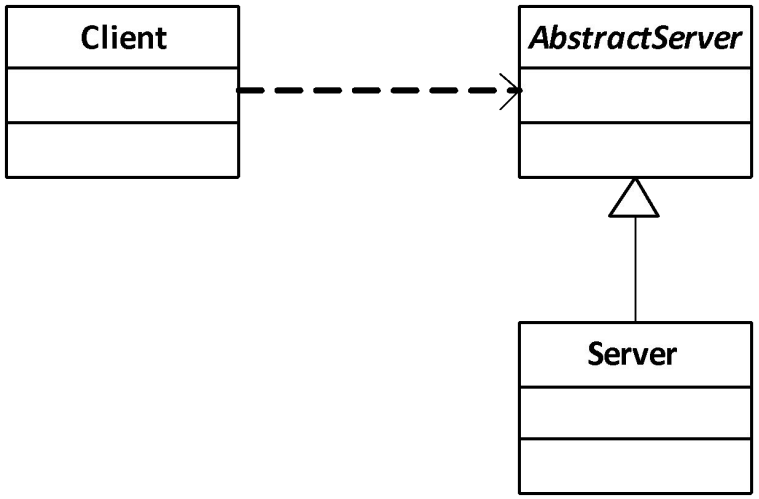


Решение: поставить выключатель в зависимость от абстракции (применить DIP)

Abstract Server: образец

Задача: исключить зависимость класса-клиента от реализации класса-сервера

Решение: использовать абстракцию (абстрактный класс или интерфейс) вместо сервера



Результаты:

- повторно используемый клиент
- соблюдение DIP

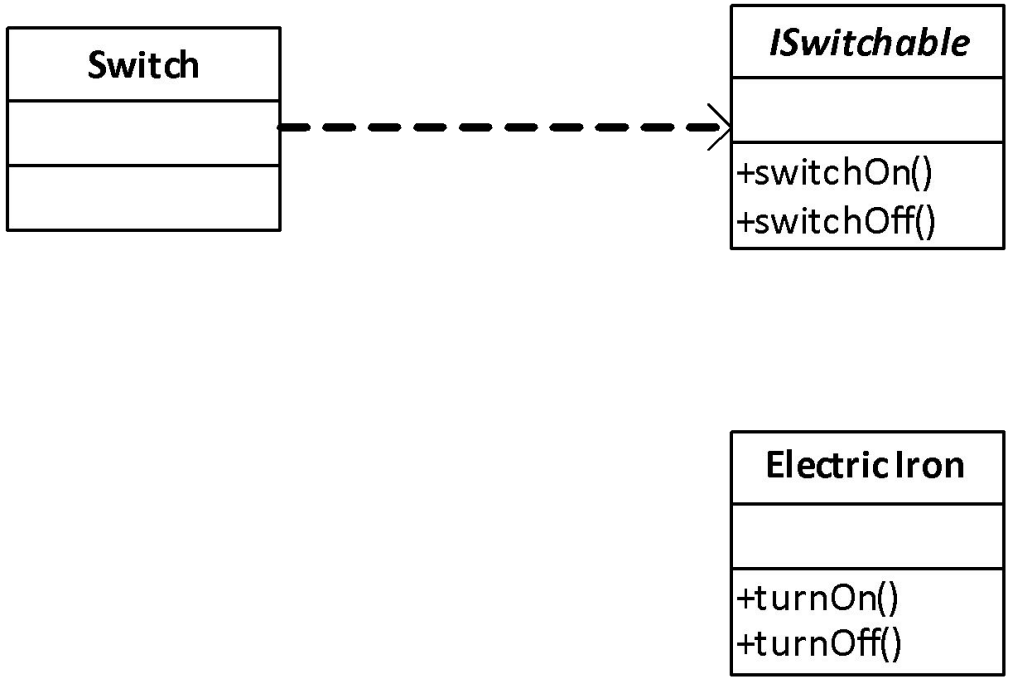
Abstract Server: замечания

Практически любой класс, предоставляющий функциональность наружу пакета рассматривается как сервер

=> использование такого класса напрямую нарушает DIP

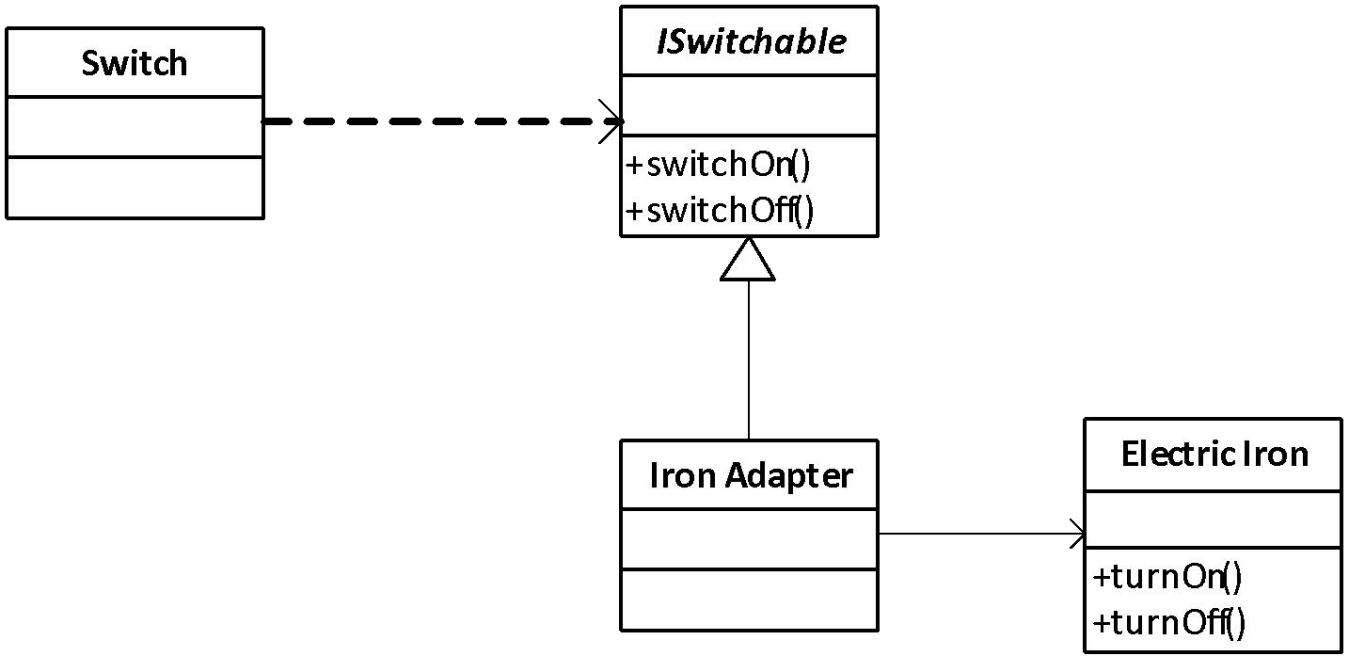
=> вся функциональность пакета должна быть абстрагирована с помощью применения Abstract Server, либо других паттернов с аналогичных эффектов (напр. Facade)

Adapter: пример задачи



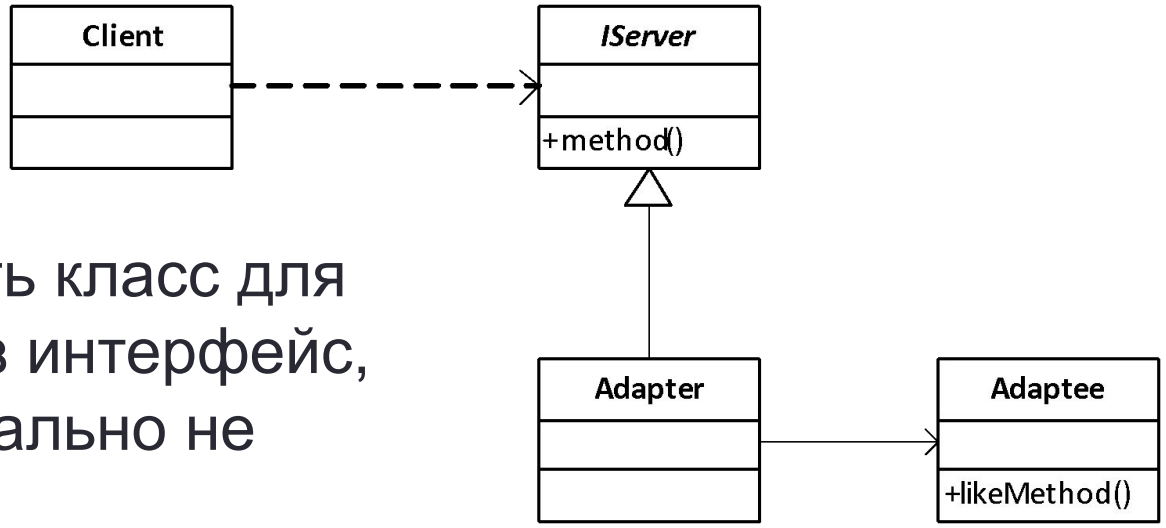
Проблема: использовать выключатель для прибора с несовместимой по форме (но совместимой по напряжению) розеткой

Adapter: пример решения



Решение: использовать переходник из одной розетки в другую

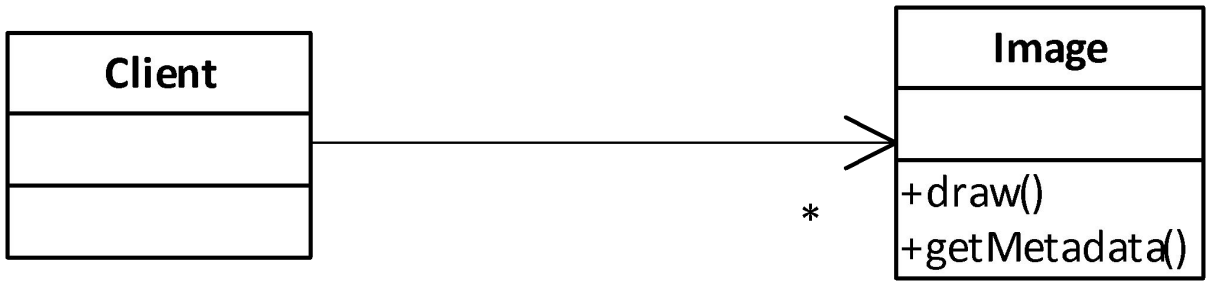
Adapter: образец



Задача: адаптировать класс для использования через интерфейс, который класс изначально не реализует

Решение: использовать класс-адаптер, реализующий требуемый интерфейс и делегирующий их реализацию адаптируемому классу

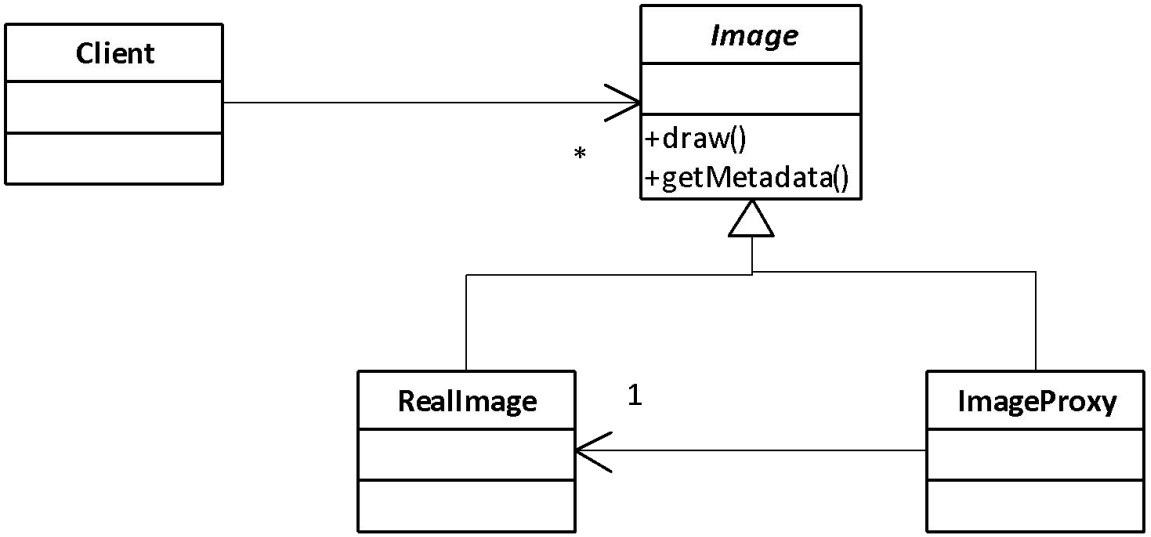
Proxy: пример задачи



Клиент оперирует большим количеством изображений, при этом только некоторые будут отображены.

Проблема: большой расход ресурсов на загрузку всех изображений.

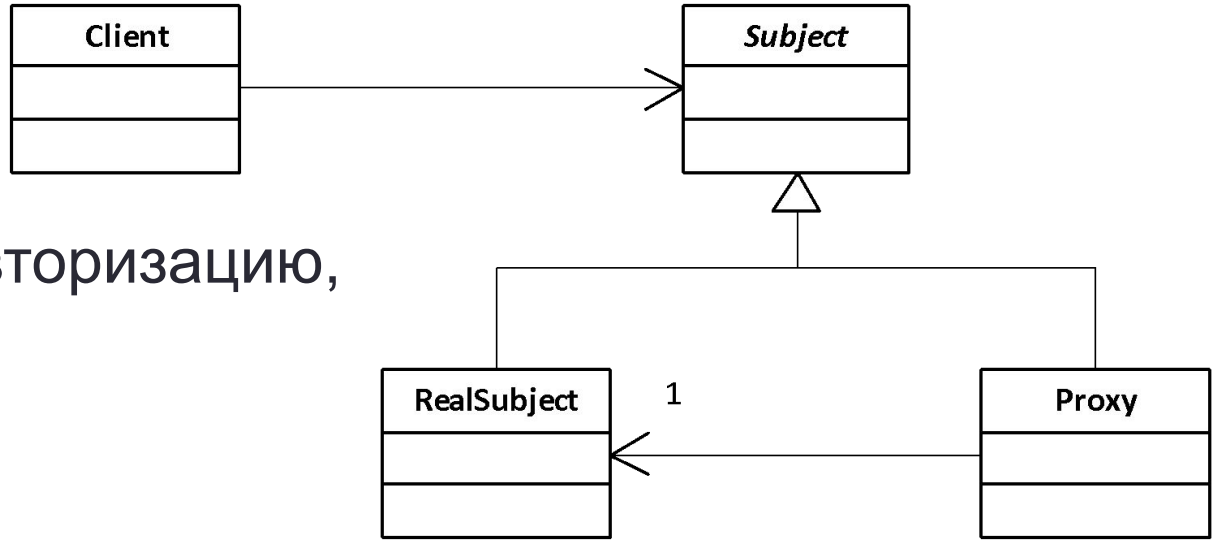
Proxy: пример решения



Решение: вместо настоящего изображения, используется заместитель, который загружает изображение только в случае его отображения.

Proxy: образец

Задача: обеспечить контроль доступа к объектам класса: авторизацию, загрузку, удаленное обращение и т.д.



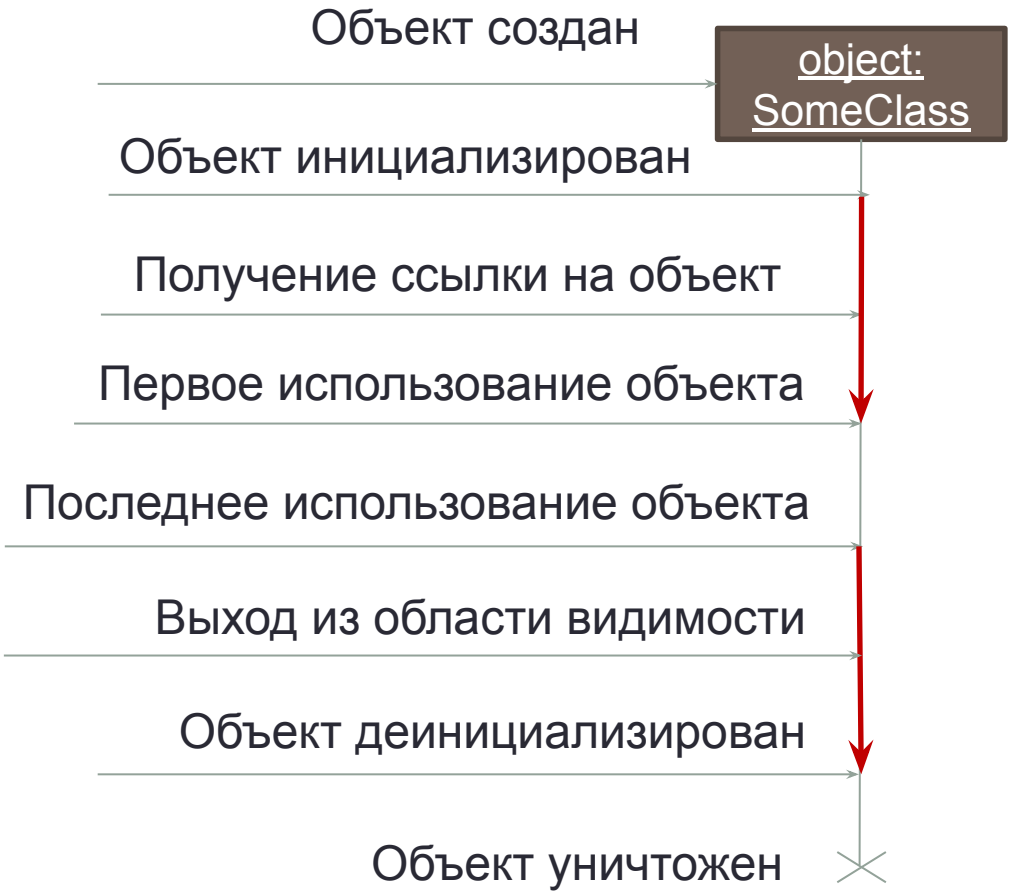
Решение: подменить объект суррогатным, который обеспечивает контроль доступа и делегирует функциональность основному объекту.

Последствия: потенциально ухудшает контроль за производительностью

Области применения Proxy

- Доступ к удаленному объекту
- Доступ к «тяжелому» объекту: отложенная загрузка, выгрузка из памяти
- Мемоизация вычислений
- Авторизация доступа
- «Умные» указатели

Жизненный цикл объекта



Проблемы контроля жизненного цикла объектов

Контроль жизненного цикла – cross-cutting concern

Проблемы:

- Вызов конструктора невозможно сделать виртуальным в классической объектной модели, т.е. Abstract Server не применим при создании.
- Для создания объекта могут требоваться данные, которыми создающий объект обладать не должен (чтобы не нарушать ORR)
- Явное удаление объекта может нарушать ORR
- Неявное удаление объекта (через сборщик мусора) не гарантирует освобождения ресурсов.
- Потенциальное нарушение LoD при использовании одного и того же объекта в разных частях объектной модели
- Использование объекта до инициализации
- Использование объекта после деинициализации

Порождающие образцы

- RAII
- Lazy Initialization
- Singleton
- Abstract Factory
- Prototype
- Builder
- Dependency Injection

Resource Acquisition Is Initialization (RAII)

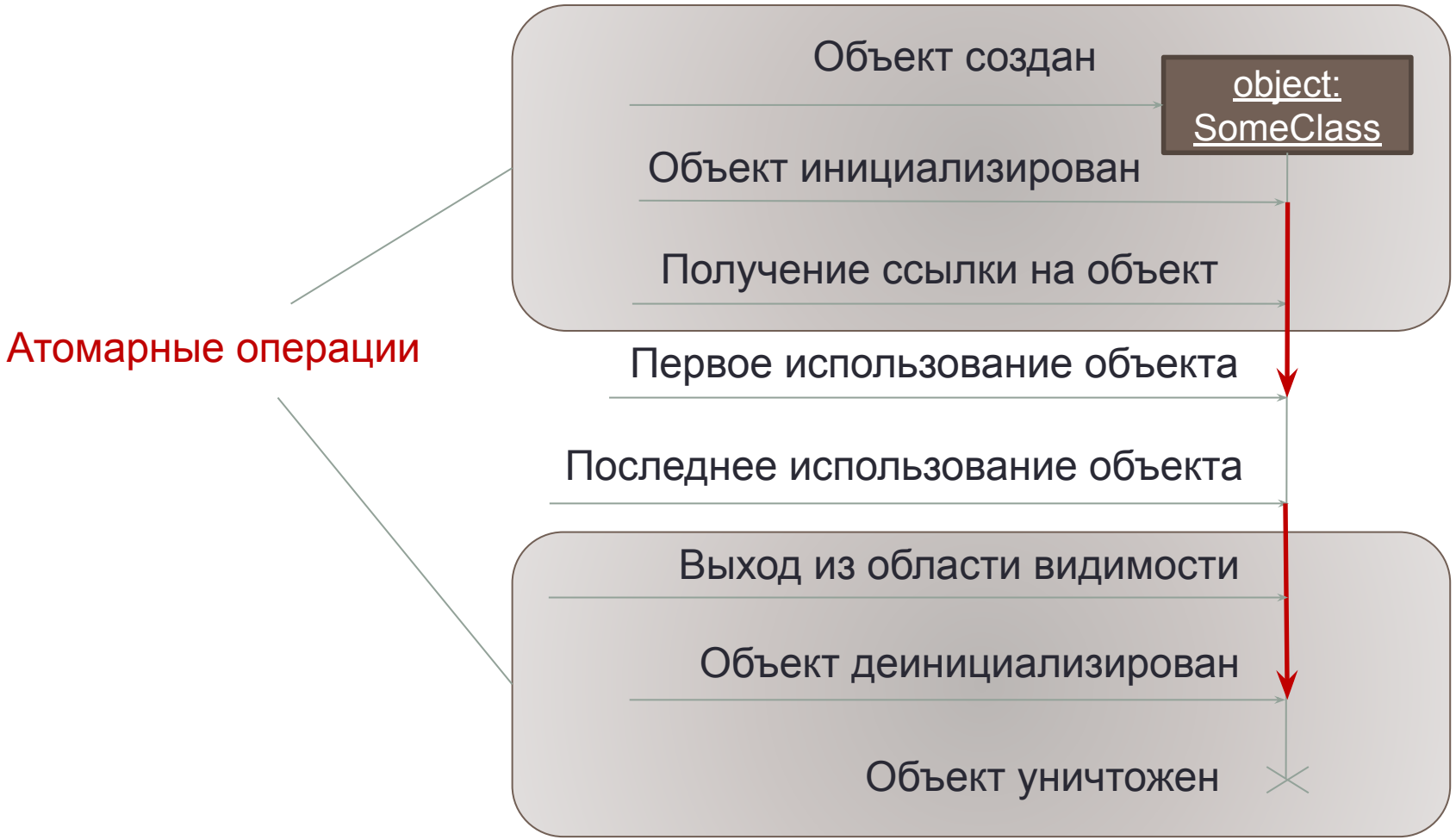
Пример задачи: необходимо обеспечить простую и эффективную с точки зрения блокирования ресурсов работу с файловыми потоками.

Пример решения (C++):

1. объявляем класс файлового потока;
2. объявляем конструктор, открывающий поток;
3. объявляем деструктор, закрывающий поток;
4. объявляем методы для чтения/записи;
5. используем, как стековый объект;

Для захвата ресурса достаточно создать объект-поток. Объект будет гарантированно закрыт в момент выхода объекта из области видимости (в т.ч. в связи с исключительной ситуации)

Resource Acquisition Is Initialization (RAII)



RAII: конструирование

Конструирование (захват ресурсов) должно производиться атомарным вызовом (конструктора или порождающей функции).

RAII: уничтожение

C++: для стековых объектов в момент выхода переменной из области видимости

Языки с динамической сборкой мусора: момент уничтожения объекта не определен, требуется явная деинициализация

RAII: Java

```
FileOutputStream out = null;
try{
  out = new FileOutputStream (outFile);
  //…
}
finally{
  out.close();
}
```

RAII: анализ

Преимущества:

- Повышает логическую безопасность системы: невозможно использовать неинициализированные объекты.
- Обеспечивает четкий контроль за использованием ресурсов

Недостатки:

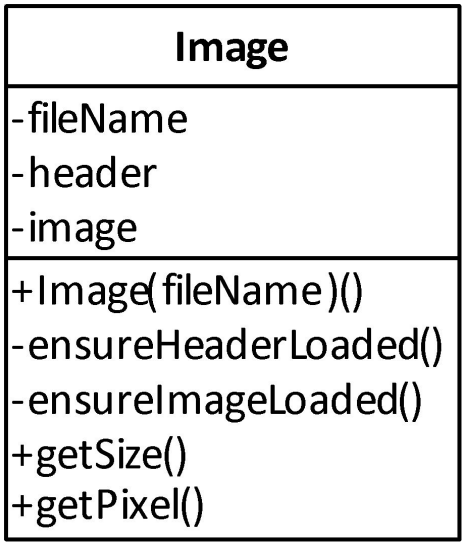
- В большинстве языков в явном виде не встречается, требует дополнительных усилий по реализации

Отложенная инициализация (Lazy Initialization)

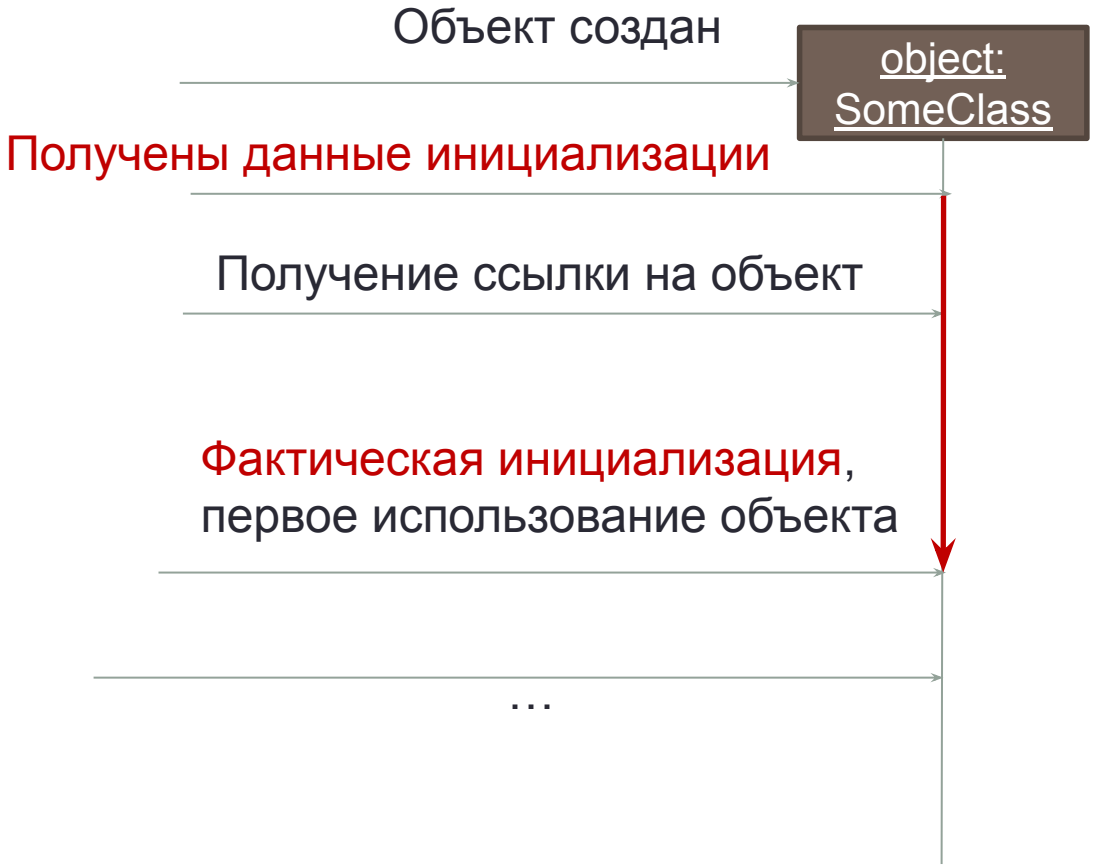
Пример задачи: необходимо реализовать класс изображений, загружаемых с файловой системы. Класс должен быть эффективным с точки зрения использования ресурсов.

Пример решения:

1. Конструктор класса принимает имя файла, и сохраняет его без загрузки изображения.
2. Вспомогательные внутренние методы проверяют, загружено ли изображение и загружают его при необходимости.
3. Публичные методы вызывают соответствующие вспомогательные методы



Отложенная инициализация: образец



Отложенная инициализация: анализ

Преимущества:

- Оптимизирует вычисления, избавляясь от инициализации ненужных объектов.
- Оптимизирует доступ к ресурсам, захватывая их непосредственно перед фактическим использованием.

Недостатки:

- Теряется контроль над использованием вычислительных ресурсов (плохо для систем реального времени).
- Не контролируется порядок захвата ресурсов => мертвые и живые блокировки.

Лирическое отступление: отложенное исполнение

Отложенные вычисления: вычисления происходят не в момент вызова, а, либо в момент затребования результата, либо асинхронно в произвольный момент времени.

Необходимое условие: чистота отложенного вызова

Распространенные примеры:

- отложенная инициализация
- copy-on-write
- мелкозернистые вычисления
- бесконечные структуры данных

Понятие чистой функции

Чистая функция (referential transparent) – функция, значение которой полностью определяется ее параметрами, и функция не имеет побочных эффектов, (т. е. никаких дополнительных эффектов кроме возврата значения)

Использование чистых функций повышает логическую безопасность программы, следует использовать

Singleton

Задача: обеспечить создание ровно одного экземпляра класса.

Решение:

```
public class Singleton(){
  private static Singleton instance = null;
  private Singleton(){…}
  public static Singleton getInstance(){
    if (instance == null) instance = new Singleton();
    return instance;
  }
}
```

Singleton: анализ

Применение:

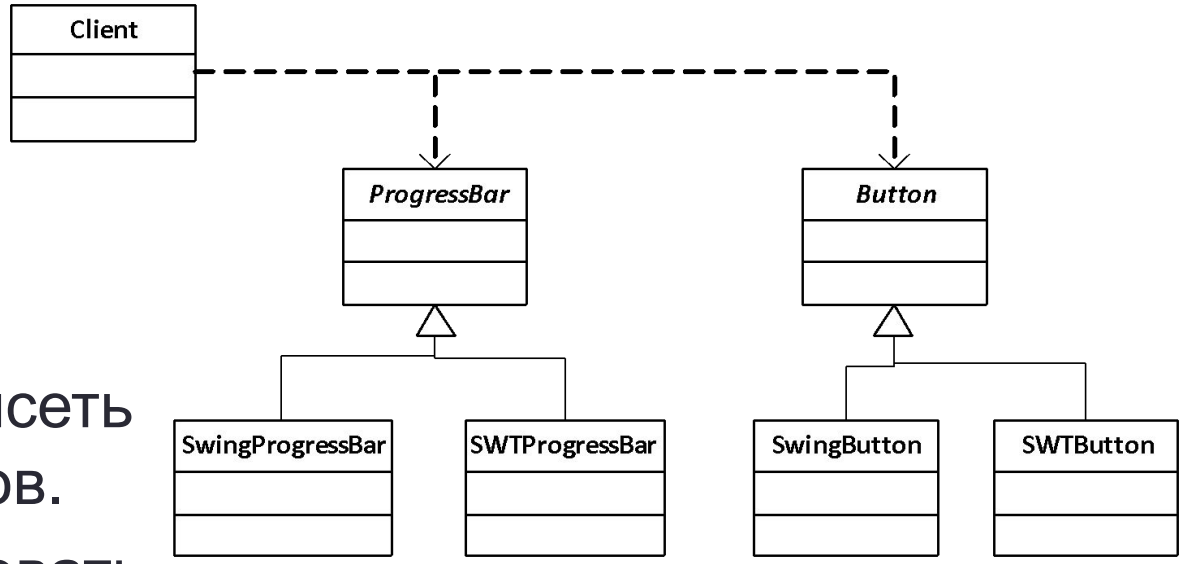
глобально-доступный инструментарий, который не может быть представлен обычными функциями (например, использует дополнительные конфигурационные параметры)

Как правило, с точки зрения бизнес-логики синглтон не хранит какого-либо состояния.

Отличия от глобальной переменной:

- Может участвовать в полиморфизме
- Инициализируется только при реальном использовании
- Интерфейс не содержит предположений относительно количества экземпляров

Abstract Factory: пример задачи

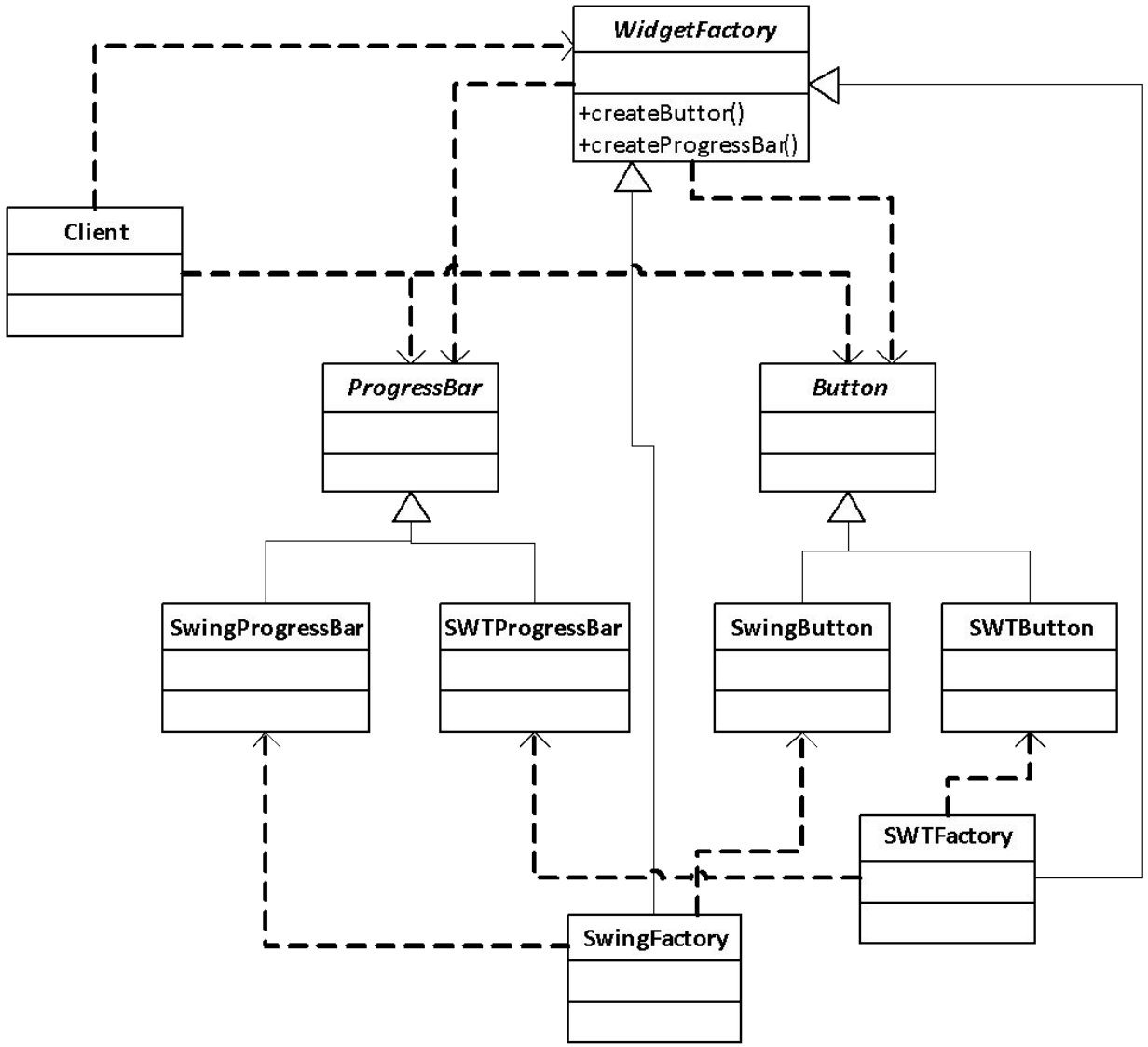


Клиент не должен зависеть от реализации виджетов.

Клиент может использовать их через базовые платформенно-независимые классы.

Как дать клиенту возможность инстанцировать эти классы?

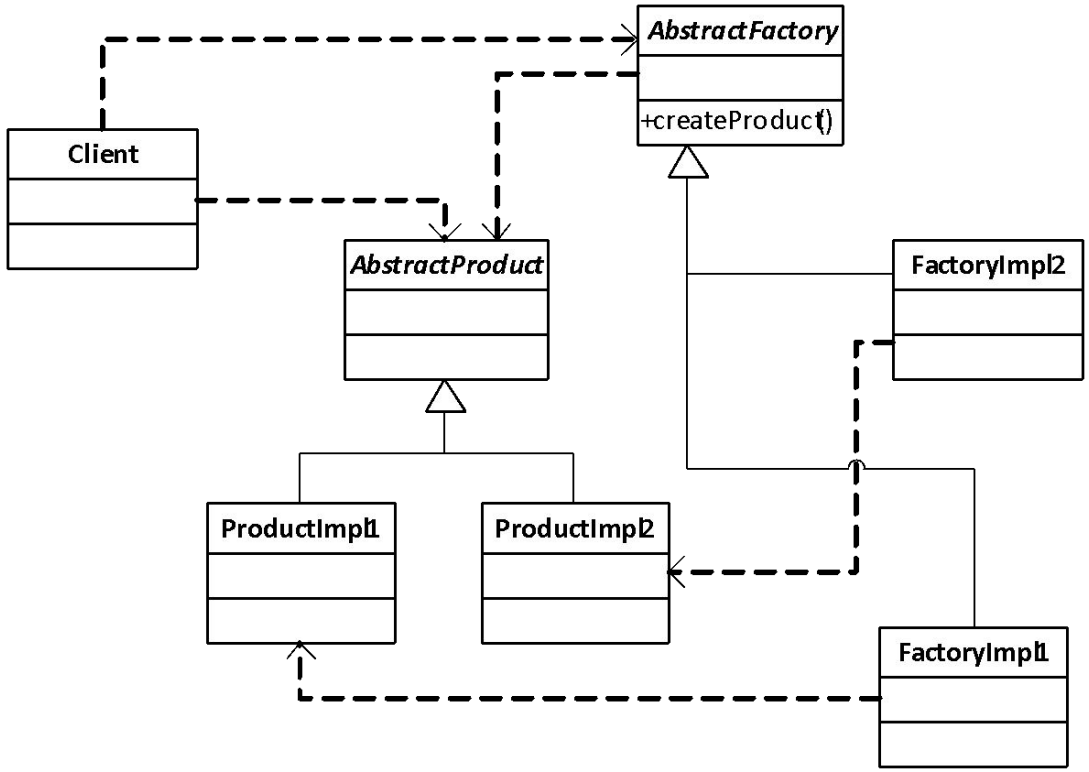
Abstract Factory: пример решения



Abstract Factory: образец

Задача: обеспечить «полиморфный» конструктор

Решение: конструирование выполняется с помощью объекта дополнительного полиморфного класса



Последствия:

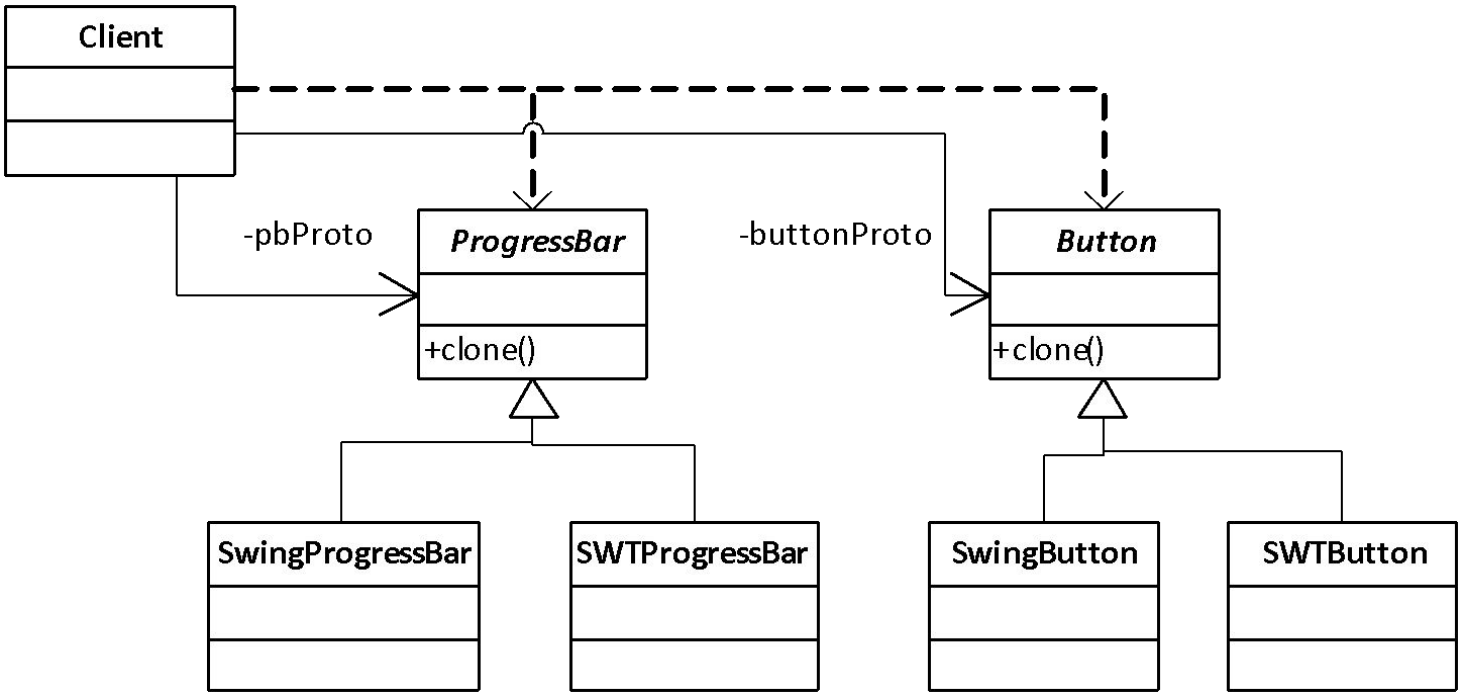
Фабрика имеет право выбросить исключение (в отличие от конструктора)

Проблема инстанцирования фабрики (кто побреет брадобрея?)

Prototype

Пример задачи: см. «Abstract Factory»

Решение: объекты создаются как клоны объектов-прототипов



Prototype: анализ

Преимущества:

- простой дизайн
- прототип обладает свойствами класса, его можно применять для создания динамических объектных моделей в статических языках

Недостатки:

- лишние, «мусорные» объекты

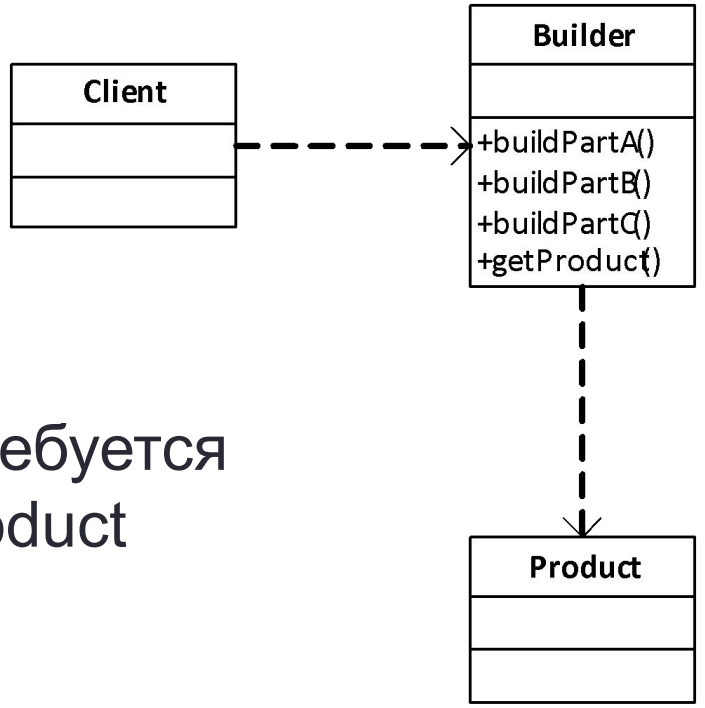
В JavaScript прототипы полностью заменяют классы.

Builder: задача

Необходимо сконструировать
сложный объект (например HTML-документ),
что невозможно сделать атомарным вызовом.

При этом необходимо, чтобы при получении доступа
к объекту он был консистентен.

Builder: образец



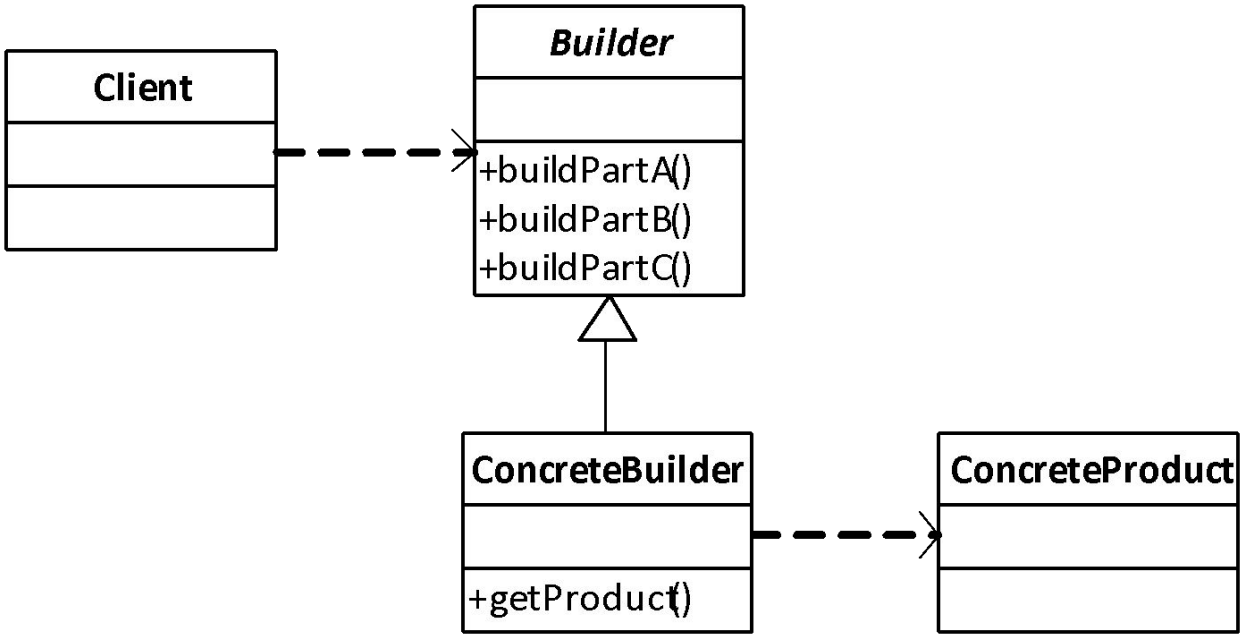
Решение:

Клиент использует build*-методы для инициализации объекта. Далее, объект востребуется атомарно посредством getProduct

Результаты:

Совместимость с RAII

Builder (GOF-вариант)



Задача:

Сконструировать сложный объект,
причем алгоритм конструирования
не должен зависеть от того, какой конкретно объект
будет получен в результате

IoC и жизненный цикл объекта

Garbage Collector: вместо явного уничтожения объект «забывается» и далее обрабатывается некоторой внешней сущность (GC)

Можно ли то же самое сделать с конструированием объекта?

Инициализация зависимостей без применения Dependency Injection

```
public interface ITool{ …}
public class Fork implements ITool{ … }

//non-DI class
public class Person1{
  ITool tool;
  public Person(){
     tool = new Fork(); //создаем вилку самостоятельно
  }
}
```

Тривиальный Dependency Injection (DI) на основе конструктора

```
//constructor-based DI class
public class Person2{
   ITool tool;
   public Person(ITool tool){ //требуем некоторый инструмент
       this.tool = tool;
   }
}
…
  public static void main(){
      ITool tool = new Fork();
     //выдаем инструмент (вилку) в пользование
      Person2 person = new Person2(tool);
  }
```

Преимущества DI

- Агрегирующие классы (т.е. те, для которых работает DI) не обязаны зависеть от конкретных классов включаемых объектов (соблюдается DIP)

- Можно явно указать какие зависимости будут общими для нескольких агрегирующие классов. При этом агрегирующие классы могут об этом не знать.

- Агрегирующие классы не зависят от «божественной» сущности, которая реализует DI

Проблемы с тривиальным DI

- DI реализует «божественная» сущность, зависящая от всей системы (нарушение ORR, LoD)
- Явно позволяет реализовать только статическое связывание. Что делать, если инструмент сломался и его нужно заменить?

DI на основе контейнера

- Контейнер универсален и не зависит от конкретных классов реализации. Зависимость устанавливается внешними конфигурационными файлами. Сам контейнер работает через интроспекцию (Java Reflection и т.д.)
- Зависимости могут внедряться опосредовано через proxy. Это позволяет подменять зависимости в соответствии с контекстом (поток, сессия и т.д.)
- Возможно конфигурировать жизненный цикл отдельных объектов (типично singleton, session, call)

DI: Spring framework example

```
//container-based DI class
public class Person3 implements IPerson{
   @Inject ITool tool;
}

//applicationContext.xml
…
<bean id=“person” class=“Person3”/>
<bean id=“tool” scope=“prototype” class=“Fork”/>
```

Поведенческие образцы

- Iterator
- Observer
- Immutable Object
- Memento
- Template method
- Strategy
- Command
- Chain of responsibilities

Iterator

Задача: имеется составной объект (список, дерево), необходимо обеспечить доступ к отдельным его частям и перемещение между ними.

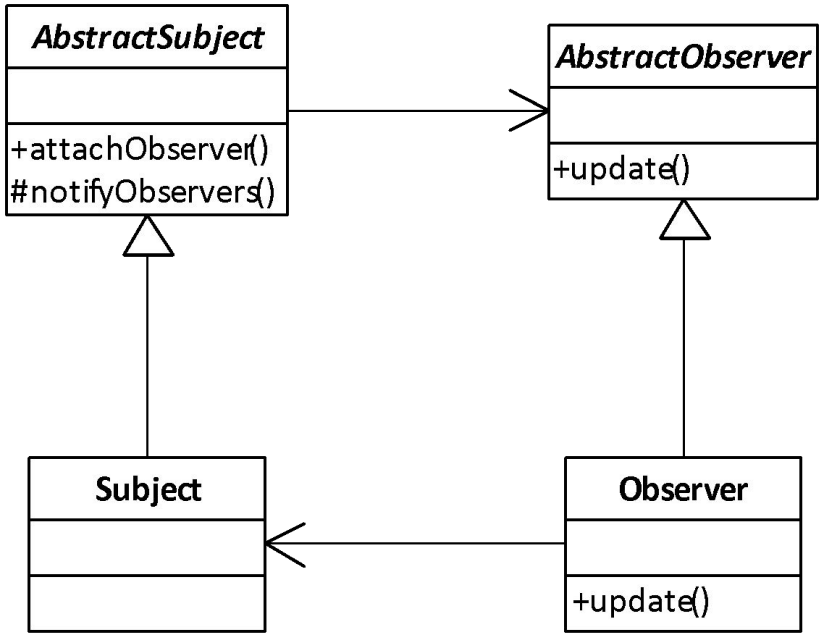
Решение: дополнительный класс, обеспечивающий:

- доступ к текущему элементу
- перемещение к следующему элементу
- удаление элемента (опционально)
- вставку элемента «после» (опционально)

Observer

Задача: автоматически оповещать объектов-наблюдателей об изменении состояния наблюдаемого объекта.

Решение:



Immutable Object

Неизменяемый объект: объект, получающий состояние при конструировании и не меняющий его.

Изменение состояния моделируется заменой самого объекта

Immutable Object: анализ

Преимущества:

- является необходимым условием для кода без побочных эффектов;
- существенно упрощает синхронизацию потоков/процессов;
- повышает логическую безопасность кода.

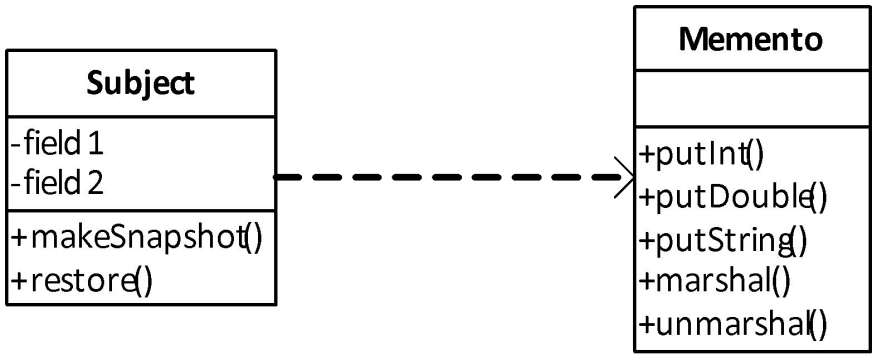
Недостатки:

- накладные расходы на копирование объектов;
- практически невозможно построить интерактивное приложение только на IO

Memento

Задача: обеспечить сохранение текущего состояния объекта и его последующее восстановления (для передачи объекта по сети, операций отката и т.д.

Решение:



Memento: анализ

Преимущества:

Обеспечивает сериализацию (маршалинг) объектов без привязки к конкретному формату/протоколу передачи.

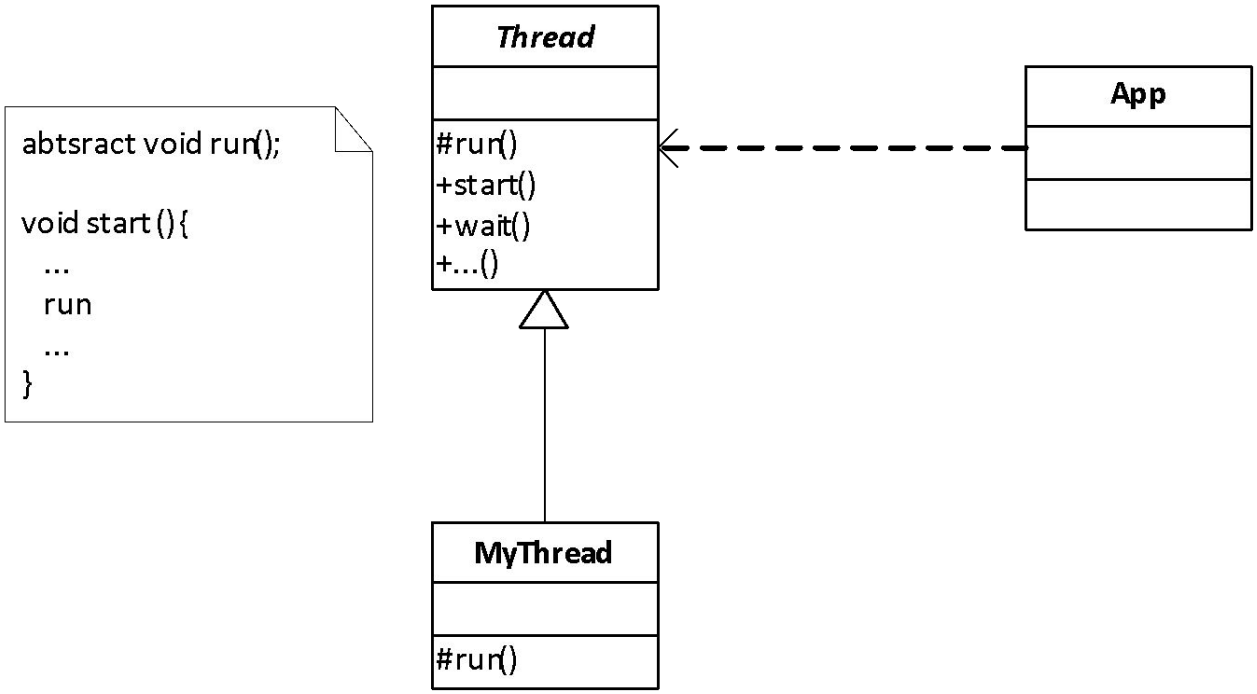
Недостатки:

Затруднено применение для сложных объектов с (например с циклическими внутренними зависимостями)

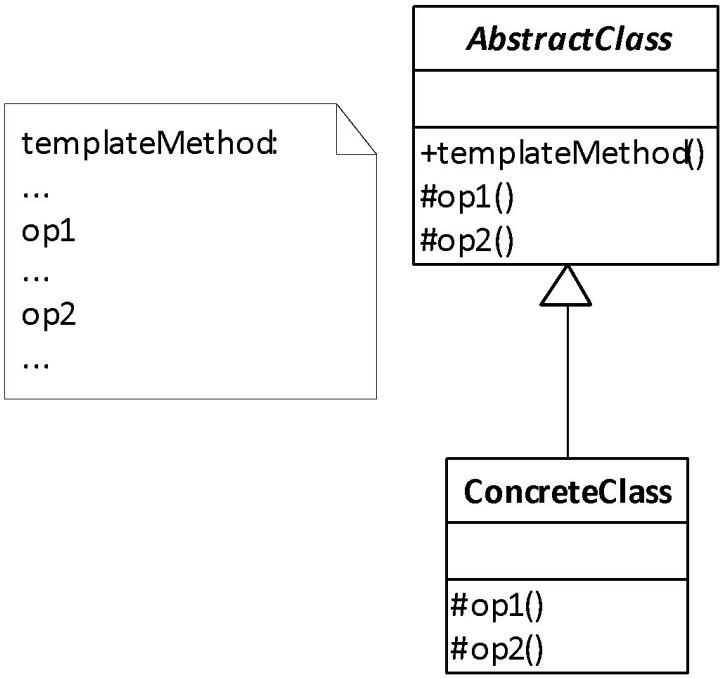
Template Method

Задача: реализовать универсальный класс Thread

Решение:



Template Method: образец



Template Method: анализ

Преимущества:

Позволяет реализовать универсальный абстрактный алгоритм, пропускающий некоторые детали своей реализации. Детали реализуются классе-потомке.

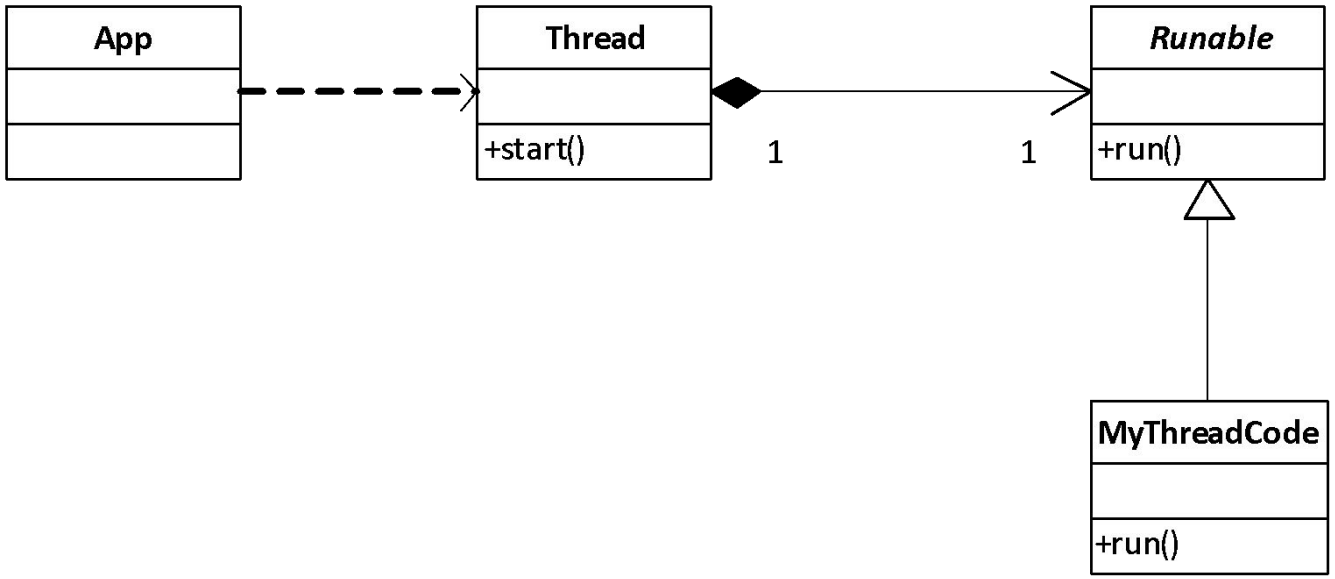
Недостатки:

Обеспечивает только статическое связывание с деталями реализации. Класс, реализующий детали зависит от всего алгоритма (потенциальное нарушение DIP)

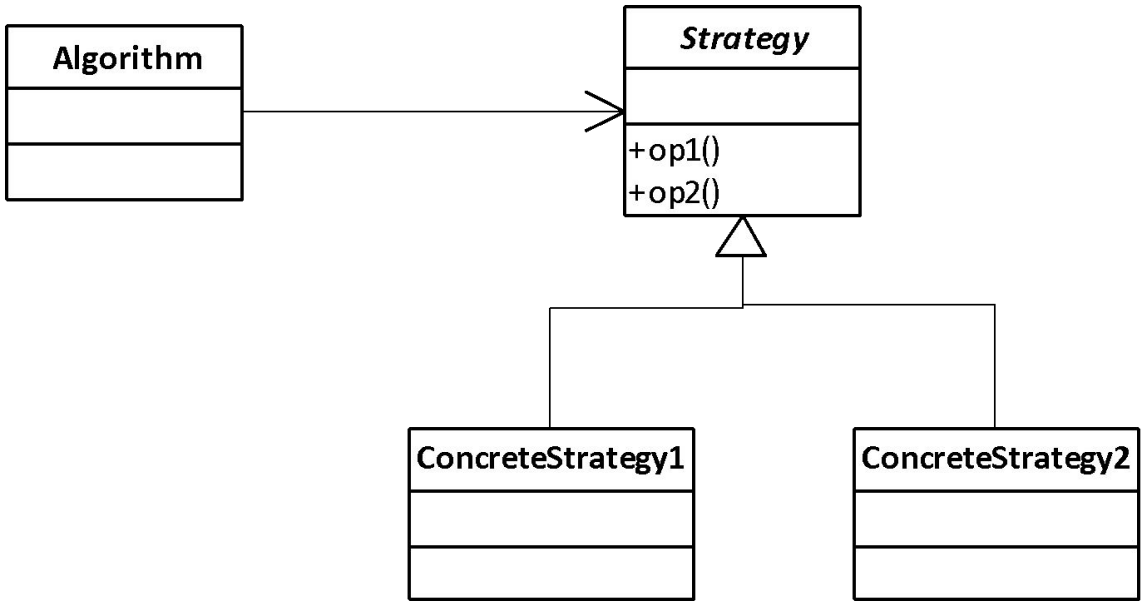
Strategy

Задача: см. Template Method

Решение:



Strategy: образец



Strategy: анализ

Преимущества:

Позволяет реализовать универсальный абстрактный алгоритм, пропускающий некоторые детали своей реализации. Детали реализуются в независимом классе, таким образом, разделяются абстракции.

Недостатки:

Несколько сложнее в реализации по сравнению с Template Method

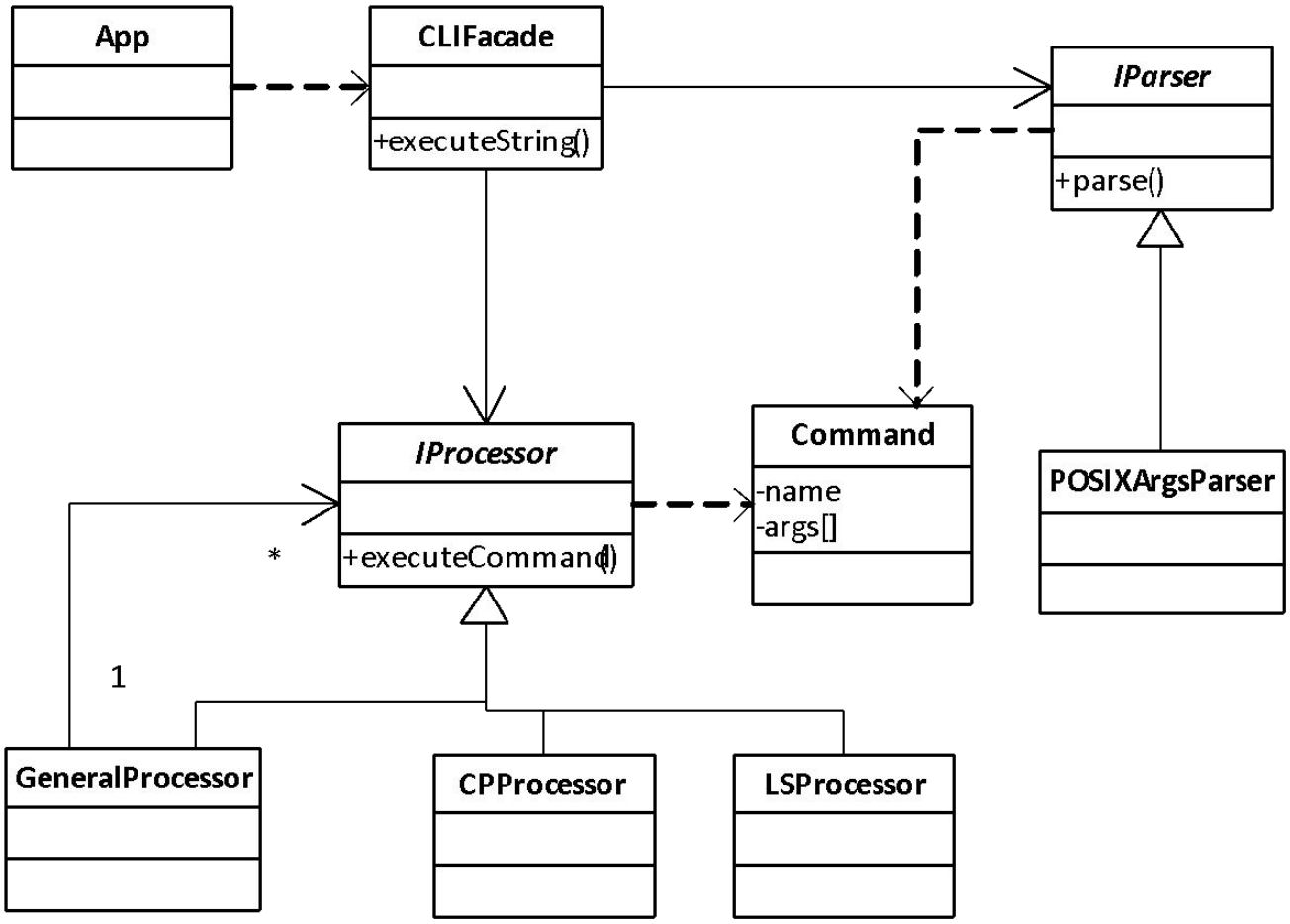
Задача: интерфейс командной строки

- все команды имеют вид
  command_name [args]

- набор команд должен быть расширяем

- добавление новых команд не должно изменять ядро (т.е. должен соблюдаться OCP)

Решение


App
CLIFacade
+executeString()
IParser
+parse()
IProcessor
+executeCommand()
Command
-name
-args[]
POSIXArgsParser
*
1
GeneralProcessor
CPProcessor
LSProcessor

GeneralProcessor

```
class GeneralProcessor{
//…
  public void executeCommand(Command command){
    boolean isProcessed = false;
    for (IProcessor p: procs){
      try{
        p.executeCommand();
      }
      catch (UnknownCommand ex){
        continue;
      }
      isProcessed = true;
      break;
    }
    if (!isProcessed) throw new UnknownCommand ();
  }
//…
}
```

Использованные шаблоны

- Strategy (2 раза)
- Chain of Responsibilities
- Command
- Facade

Command-Query Separation (CQS)

CQS – принцип построения интерфейсов.

Каждый метод является либо командой, либо запросом но не тем и другим одновременно.

Команда – метод, изменяющий состояние объекта и не возвращающий никакого значения

Запрос – метод без побочных эффектов, возвращающий значение.

Примеры «разумных» нарушений CQS

- Команда, возвращающая статус операции
- Запрос, выставляющий внешний признак ошибки (напр. в переданную по ссылке переменную)
- Кэширующий запрос