ООП, лекция 3-4

Паттерны проектирования

Соломатин Д.И., ПиИТ, ФКН, ВГУ
solomatin@cs.vsu.ru

Что такое «хороший дизайн»

- Точного универсально ответа нет и быть не может (скорее можно сказать, что такое «плохой дизайн»)
- Есть стандартные решения для определенных задач – **паттерны проектирования**
  - Предложены хорошими специалистами
  - Проверены временем
  - Составляют удобный словарь для общения

Канонические паттерны (GoF-паттерты)

- GoF – Gang of Four («банда четырех») – **Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес**
  - Книга

    «Design Patterns: Elements of Reusable Object-Oriented Software»

    («Приемы объектно-ориентированного проектирования. Паттерны проектирования»)
    - Вышла в 1995 году
    - Выделены 23 паттерна проектирования (получили название GoF-паттернов или канонических паттернов)
    - Примеры на C++, кое-что на SmallTalk

Определения

- Паттерны проектирования – описание взаимодействия объектов и классов, адаптированных для решения общей задачи проектирования в конкретном контексте
  @ GoF

- Под классическими паттернами проектирования понимаются повторяющиеся элементы дизайна приложений для объектно-ориентированных языков программирования со статической типизацией (C++, C#, Java, Object Pascal и др.)

  - для объектно-ориентированных языков с динамической типизацией (JavaScript и др.) тоже можно говорить о паттернах проектирования, но немного в другом контексте

Что НЕ является паттерном

- Готовые классы (в том числе и повторно используемые)
  - Структуры данных
  - Потоки
  - И т.д.
- Архитектура в целом
  - Клиент-Сервер
  - Трехзвенная

Элементы паттерна (GoF)

- **Имя**
  - Как устоявшийся профессиональный термин может использоваться для четкого описания архитектурного решения в документации, при обсуждении с коллегами и т.п. Присваивание паттернам имен позволяет в дальнейшем проектировать на более высоком уровне абстракции.
- **Задача**
  - Описание того, когда следует применять тот или иной паттерн. В задаче может описываться конкретная проблема проектирования, например, способ представления алгоритмов в виде объектов. В задаче может отмечаться, какие структуры классов или объектов свидетельствуют о негибком дизайне, также может приводится перечень условий, при которых имеет смысл применять данный паттерн.
- **Решение**
  - Описание элементов дизайна, отношений между ними, функций каждого элемента. Речь идет не о конкретном дизайне или реализации для конкретной задачи, а о шаблоне решения, применимом в различных, но схожих ситуациях.
- **Результаты**
  - Следствия применения паттерна и разного рода компромиссы. Описание результатов позволяет оценить преимущества и недостатки того или иного паттерна и выбрать решение между различными вариантами дизайна.

Описание паттерна (GoF)

- Название и классификация паттерна
- Назначение
- Известен также под именем
- Мотивация
- Применимость
- Структура
- Участники
- Отношения
- Результаты
- Реализация
- Пример кода
- Известные применения
- Родственные паттерны

Для чего применяются паттерны

- Для создания «хорошего дизайна»
  - Инкапсуляция
  - Повторное использование
  - Снижение зависимостей
  - И т.д.

Механизмы повторного использования

- **Наследование** («прозрачный ящик»)
  - Наследование класса определяется статически на этапе компиляции
  - Нарушение инкапсуляции родителя для потомков
  - Тесная связь родителя и потомков
  - Нельзя изменить унаследованную от родителя реализацию во время выполнения программы
- **Композиция** («черный ящик»)
  - Композиция объектов – это альтернатива наследованию класса
  - Для композиции требуется, чтобы объединяемые объекты имели четко определенные интерфейсы
  - Композиция объектов определяется динамически во время выполнения, следовательно, **реализация может меняться во время выполнения**
  - **Композиция объектов в большинстве случаев предпочтительнее наследованию классов**
- **Обобщенное программирование**

Классификация GoF-паттернов

- **Порождающие паттерны**
  - Отвечают за создание объектов
- **Структурные паттерны**
  - Организуют структуру классов (на этапе разработки) или объектов (на этапе выполнения программы)
- **Паттерны поведения**
  - Характеризуют, как классы и объекты взаимодействуют между собой

Пространство GoF-паттернов

Цель / Уровень	Порождающие паттерны	Структурные паттерны	Паттерны поведения
Класс	Фабричный метод	Адаптер (класса)	Интерпретатор
			Шаблонный метод
Объект	Абстрактная фабрика	Адаптер (объекта)	Итератор
	Одиночка	Декоратор	Команда
	Прототип	Заместитель	Наблюдатель
	Строитель	Компоновщик	Посетитель
		Мост	Посредник
		Приспособленец	Состояние
		Фасад	Стратегия
			Хранитель
			Цепочка обязанностей

Отношения между GoF-паттернами

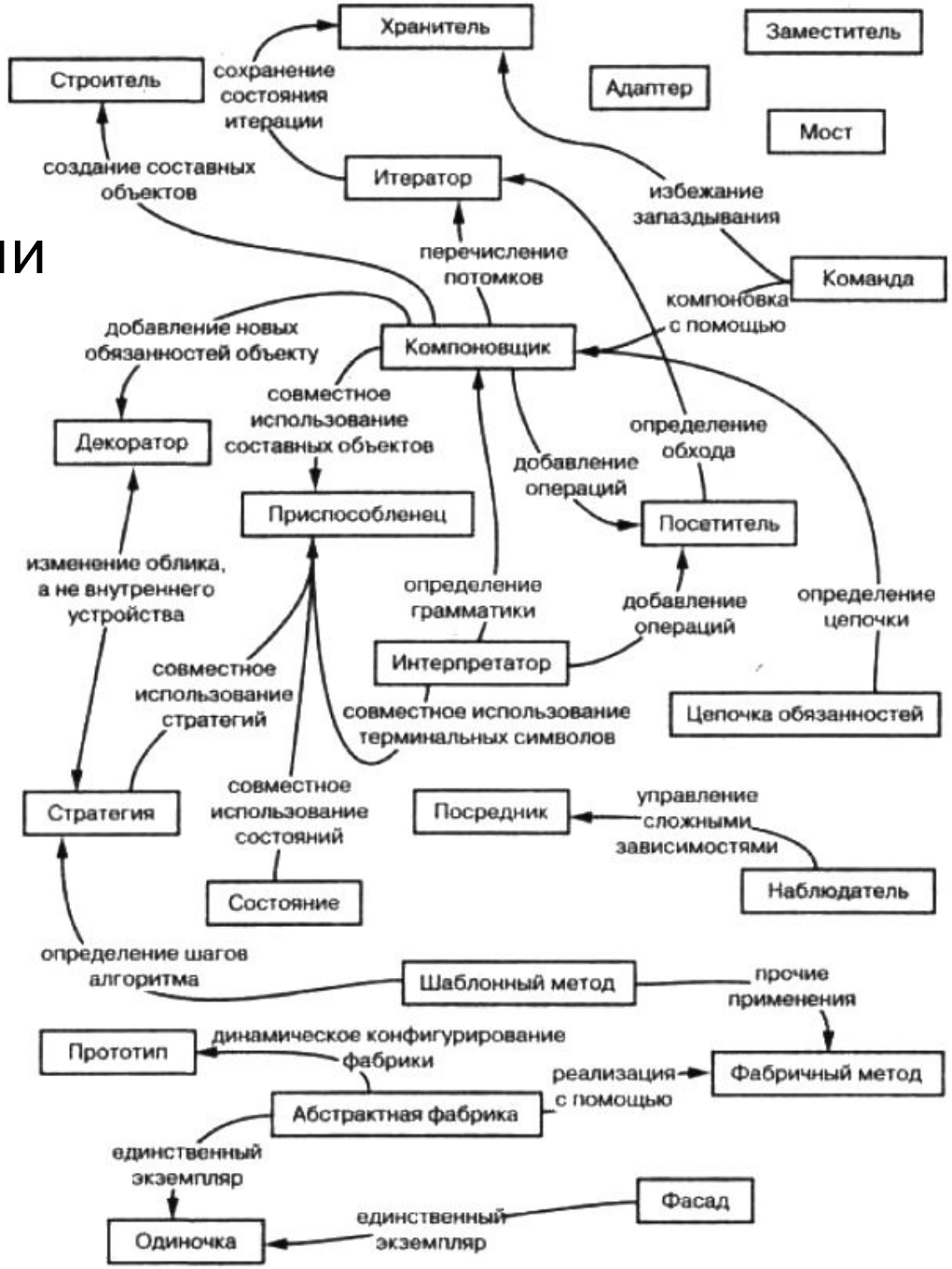


Рис. 1.1. Отношения между паттернами проектирования

Как выбирать паттерн проектирования

- Подумайте, как паттерны решают проблемы проектирования
- Пролистайте разделы каталога, описывающие назначение паттернов
- Изучите взаимосвязи паттернов
- Проанализируйте паттерны со сходными целями
- Разберитесь в причинах, вызывающих перепроектирование
- Посмотрите, что в вашем дизайне должно быть изменяющимся

Как пользоваться паттерном проектирования

- Прочитайте описание паттерна, чтобы получить о нем общее представление
- Вернитесь назад и изучите разделы «Структура», «Участники» и «Отношения»
- Посмотрите на раздел «Пример кода», где приведен конкретный пример использования паттерна в программе
- Выберите для участников паттерна подходящие имена
  - Обычно имена участников паттерна слишком абстрактны, но иногда бывает удобно включить имена участников паттерна в имена элементов программы (SimpleLayoutStrategy, TeXLayoutStrategy)
- Определите классы
  - Объявите их интерфейсы, установите отношения наследования и определите переменные экземпляра, которыми будут представлены данные объекты и ссылки на другие объекты.
- Определите имена операций, встречающихся в паттерне
  - Будьте последовательны при выборе имен. Например, для обозначения фабричного метода можно было бы всюду использовать префикс Create-.
- Реализуйте операции, которые выполняют обязанности и отвечают за отношения, определенные в паттерне

Когда использовать паттерны

- Необоснованное использование паттернов может существенно усложнить программу, поэтому применять паттерны нужно, или когда проблема уже возникает или когда ее появление можно предсказать
- Существует мнение, что применять паттерны надо начинать только тогда, когда программист сам начинает выделять паттерны в своих проектах; до этого момента в большинстве случаев будет иметь место необоснованное их использование
- Но в любом случае познакомиться с ними необходимо хотя бы для того, чтобы как можно раньше появился материал для размышлений

Графическая нотация OMT (Object Modeling Technique)

- OMT (Object Modeling Technique) – один из методов ООАП и одновременно одна из общепризнанных систем графических обозначений Д. Румбаха (James Rumbaugh)
  - Используется для иллюстраций к паттернам проектирования в книге «банды четырех»
- Существуют и другие графические нотации, например UML

OMT, диаграммы классов

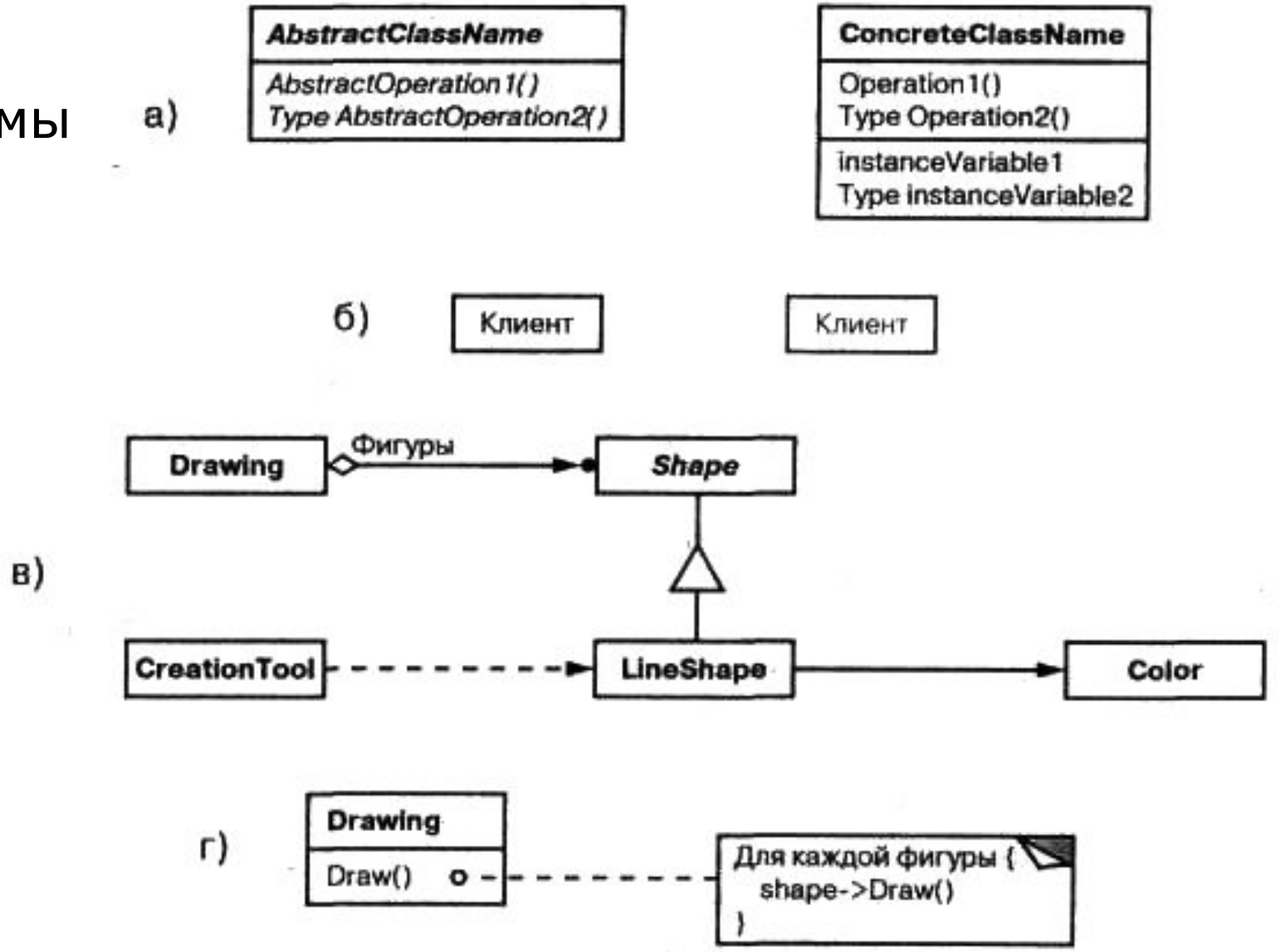


Рис. В.1. Нотация диаграмм классов: а) абстрактные и конкретные класс
б) класс клиента участника (слева) и класс неявного клиента (справа);
в) отношения между классами; г) аннотация на псевдокоде.

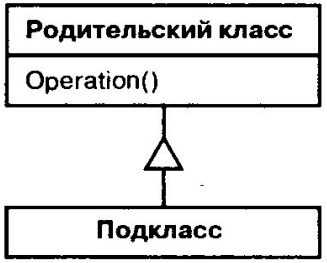
OMT, описание классов

- Класс обозначается прямоугольником
- В верхней части напечатано имя класса
- Описание переменных располагается ниже описания методов
- Можно ставить имя типа перед методом, переменной экземпляра или фактического параметра
- Курсивом в имени обозначаются абстрактные классы (соответственно и интерфейсы) и методы
- При описании паттернов проектирования бледным шрифтом часто обозначают клиентов, которые не входят в состав участников паттерна

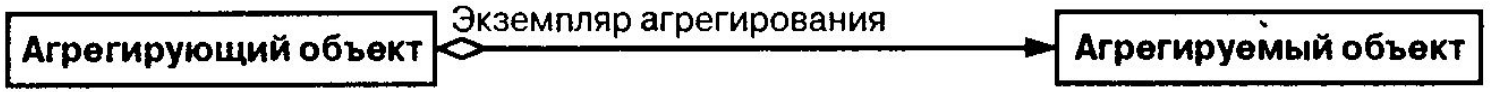
OMT, связи между классами

- Инстанцирование

- Наследование



- Агрегирование

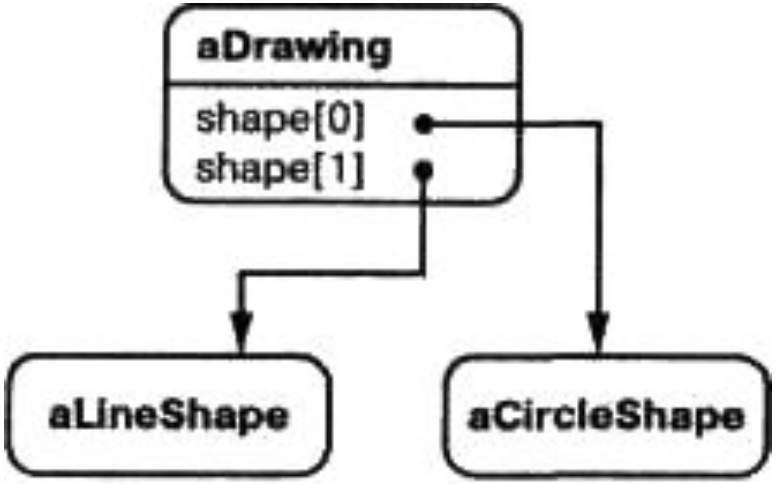


- Осведомленность (ассоциация, использование)

Агрегирование и осведомленность

- И агрегирование и осведомленность – способы композиции объектов и делегирования функциональности
- Агрегирование
  - Один объект владеет другим или несет за него ответственность
  - Агрегат и его составляющие имеют одинаковое время жизни
- Осведомленность
  - Одному объекту известно о другом объекте
  - Осведомленные объекты не несут никакой ответственности друг за друга
- Осведомленность более слабое отношение, чем агрегирование

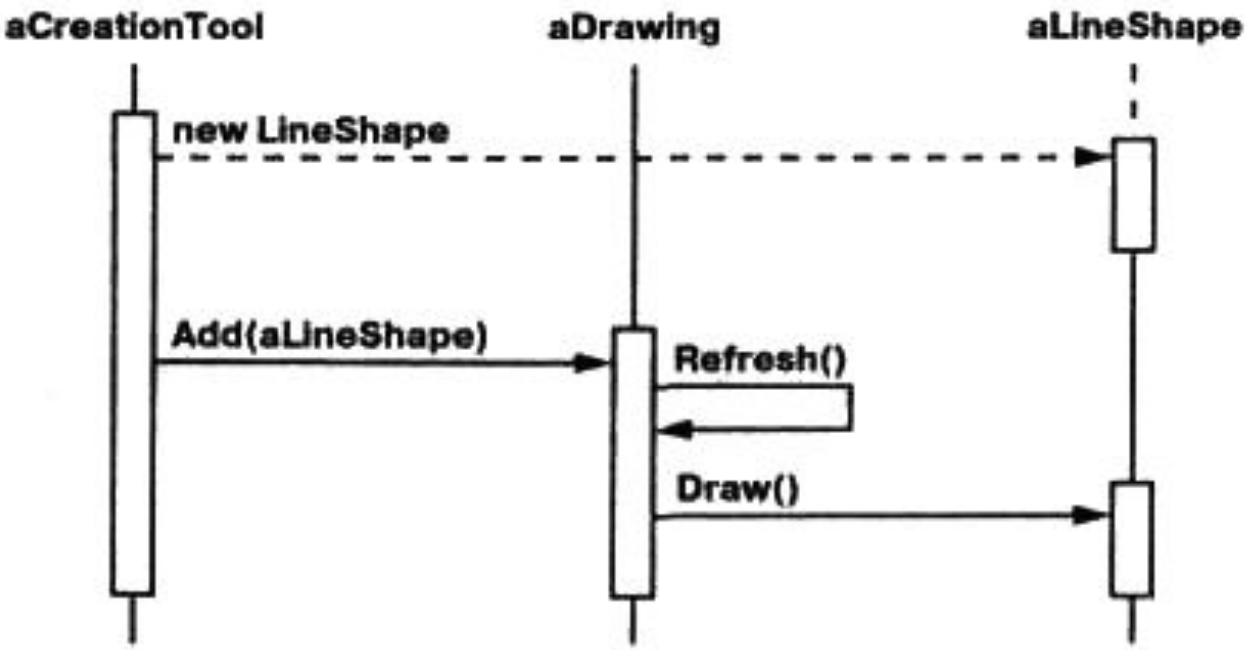
OMT, диаграммы объектов



OMT, описание объектов

- В диаграммах объектов приводятся только экземпляры в какой-то определенный момент времени
- Для обозначения объектов используются прямоугольники со скругленными углами
- Объекты именуются «aSomething», где Something – класс объекта
- Стрелки ведут к объектам, на которые ссылается данный

OMT, диаграммы взаимодействия



OMT, описание взаимодействий

- Время на диаграммах взаимодействий откладывается сверху вниз
- Сплошная вертикальная черта означает время жизни объекта, до момента создания объекта вертикальная линия идет пунктиром
- Вертикальный прямоугольник говорит о том, что объект активен, т.е. обрабатывает какой-либо запрос (выполняет какой-либо метод)
- Соглашение об именовании объектов такое же, как для диаграмм объектов
- Запросы, посылаемые другим объектам (вызовы методов), обозначаются горизонтальной стрелкой, указывающей на объект получатель; имя запроса (метода) показывается над стрелкой
- Запрос на создание объекта обозначается горизонтальной пунктирной стрелкой
- Запрос объекта самому себе изображается стрелкой на этот же объект

Порождающие паттерны

- Фабричный метод (Factory Method)
- Абстрактная фабрика (Abstract Factory)
- Прототип (Prototype)
- Строитель (Builder)
- Одиночка (Singleton)

Порождающие паттерны, причина появления

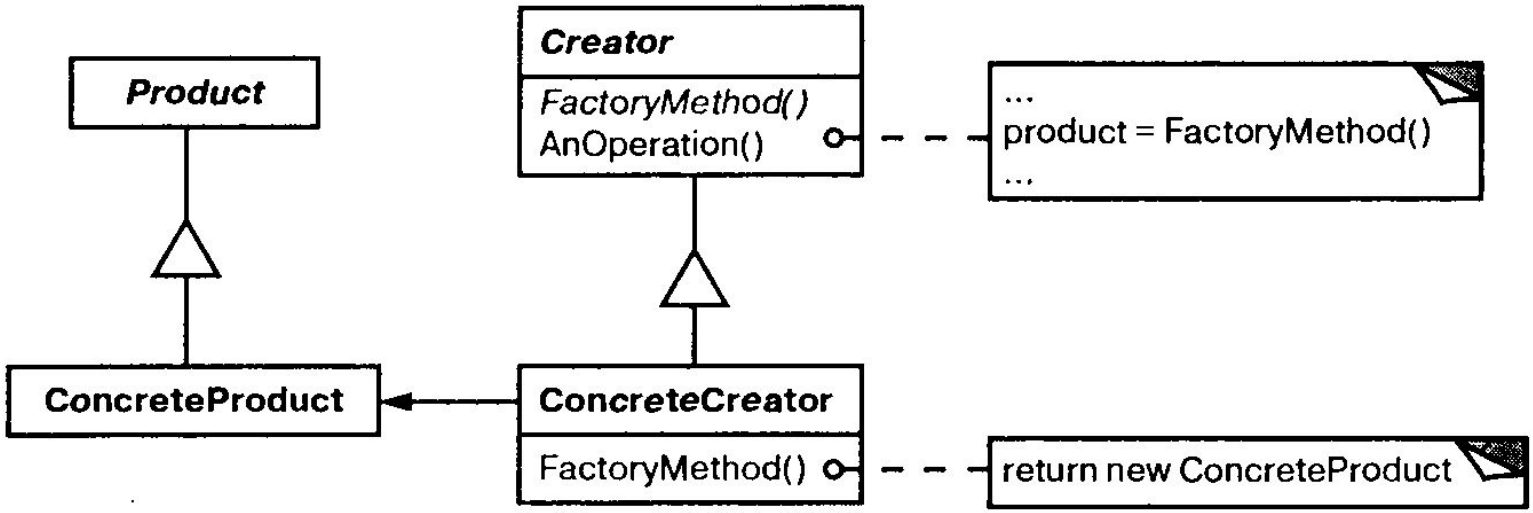
- Казалось бы, все классы создаются вызовом своего конструктора, зачем вводить порождающие паттерны?
- Объяснение заключается в принципе ***программирования в соответствии с интерфейсом, а не с реализацией***
  - Взаимодействие объектов должно проходить строго через интерфейс, т.к. клиенту не нужно знать информацию о конкретных типах объектов
  - Такой подход может кардинально уменьшить число зависимостей между подсистемами
  - Но, т.к. клиенты не знают, объекты какого конкретного класса будут созданы, следовательно самостоятельно создать эти объекты не могут; здесь приходят на помощь порождающие паттерны

Фабричный метод (Factory Method), задача

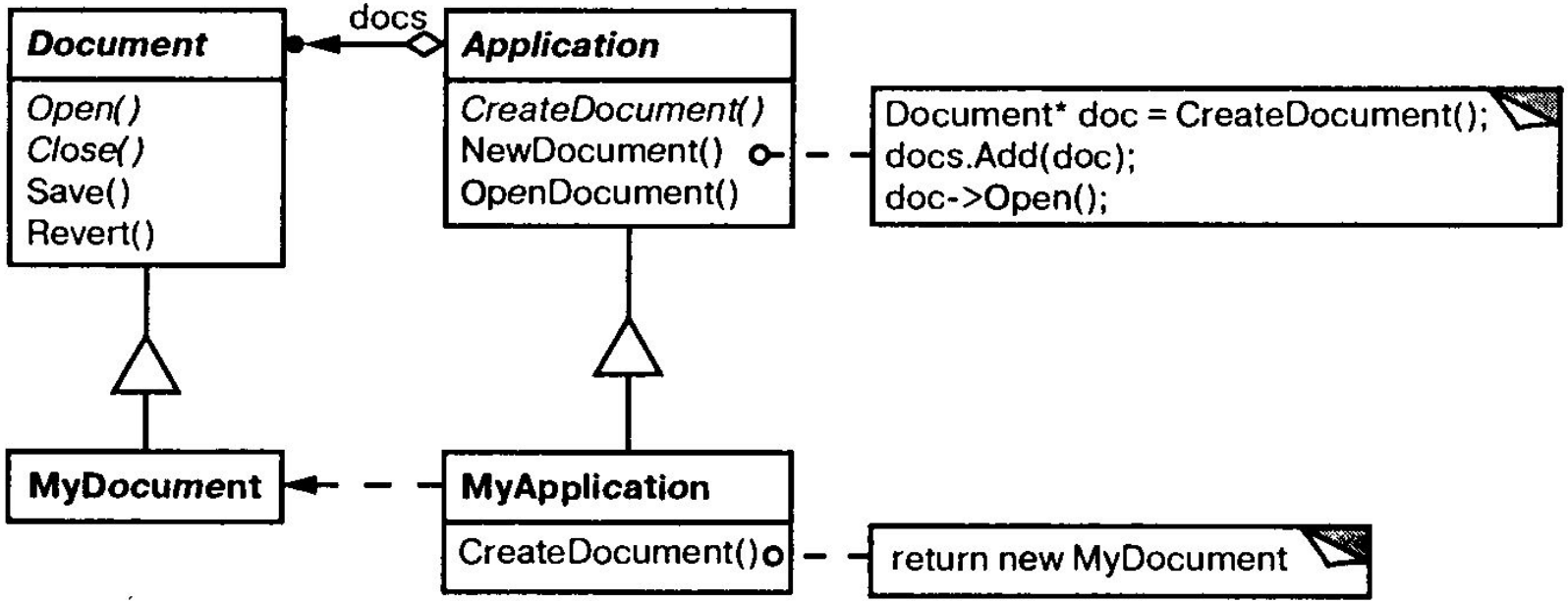
- Определяет интерфейс для создания объекта абстрактного (редко конкретного) типа, но скрывает способ создания
- Фабричный метод позволяет классу делегировать инстанцирование подклассам

Фабричный метод (Factory Method), диаграмма классов

- Делегирование инстанцирования подклассам
  - В данном конкретном случае является частным случаем абстрактной фабрики с одним видом продукта



Фабричный метод (Factory Method), пример с созданием документов



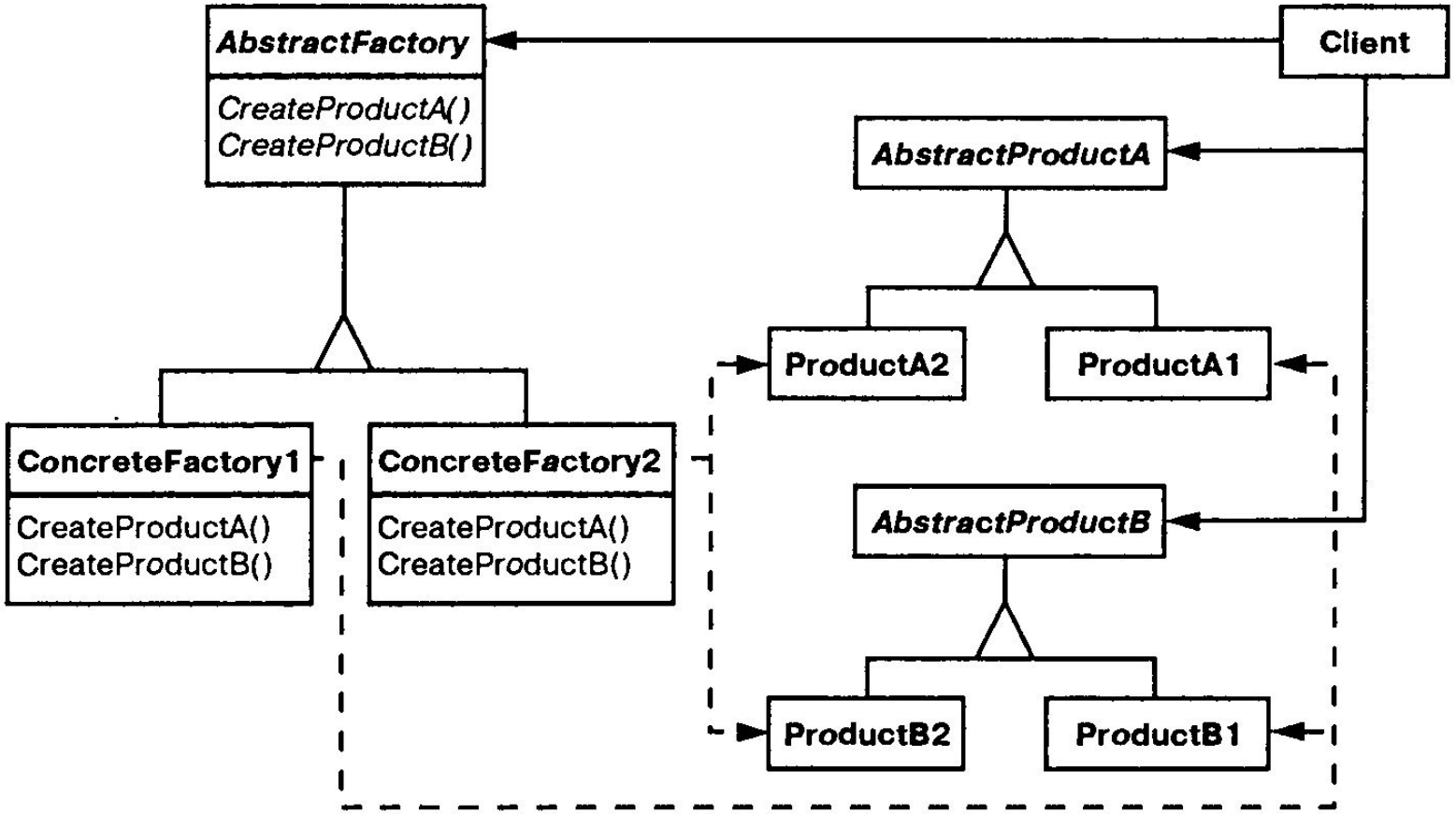
Абстрактная фабрика (Abstract Factory), задача

- Предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, не специфицируя их конкретных классов (варьирует создаваемые классы, сохраняя при этом их интерфейсы)

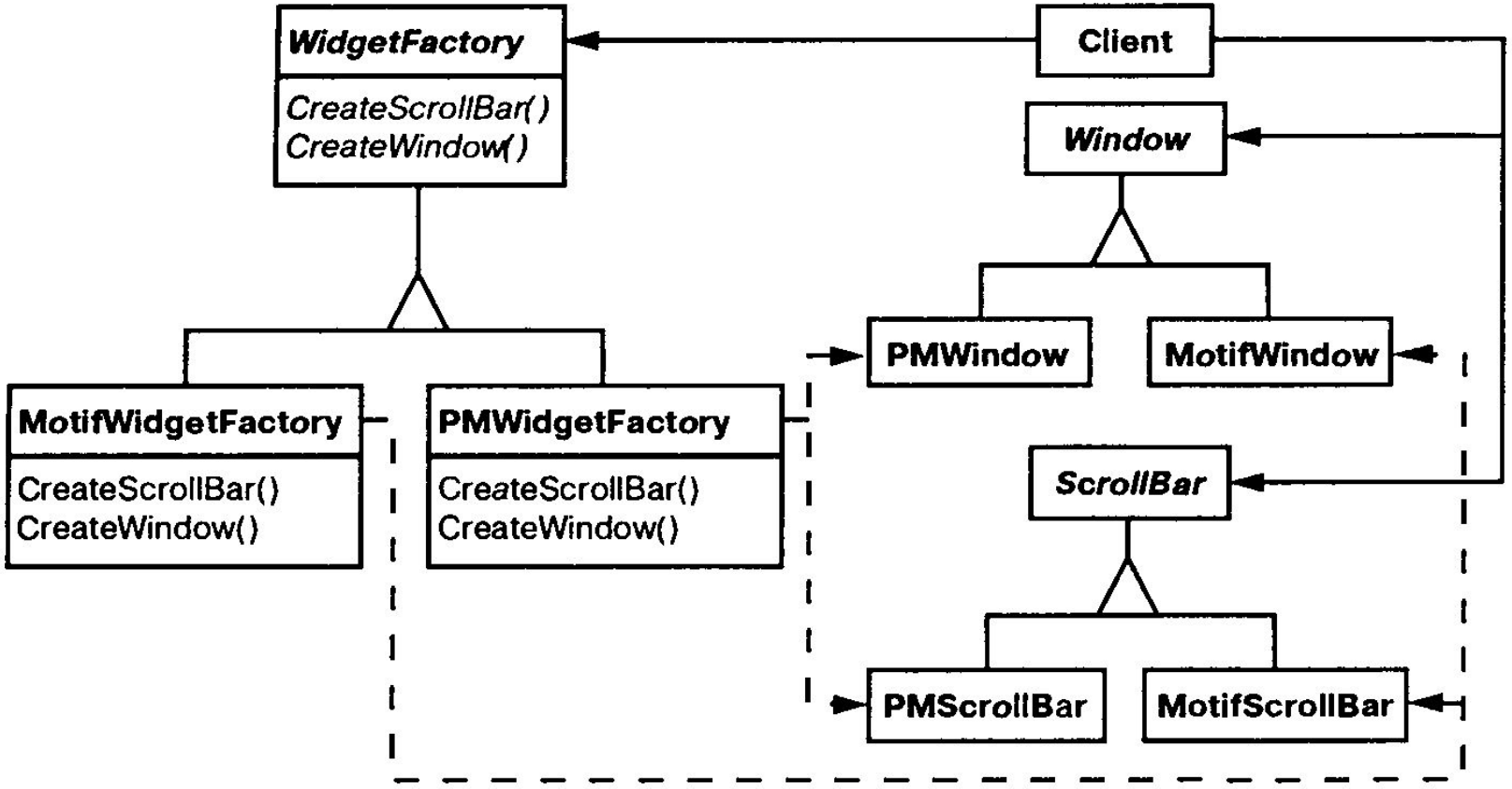
- Используется когда:

  - система не должна зависеть от того, как создаются, компонуются и представляются входящие в нее объекты
  - входящие в семейство взаимосвязанные объекты должны использоваться вместе и вам необходимо обеспечить выполнение этого ограничения
  - система должна конфигурироваться одним из семейств составляющих ее объектов
  - необходимо предоставить библиотеку объектов, раскрывая только их интерфейсы, но не реализацию

Абстрактная фабрика (Abstract Factory), диаграмма классов



Абстрактная фабрика (Abstract Factory), пример с различн. оконными системами



Абстрактная фабрика (Abstract Factory), результаты

- изолирует конкретные классы
- упрощает замену семейств продуктов
- гарантирует сочетаемость продуктов
- поддержать новый вид продуктов трудно
  - Надо или тщательно проектировать интерфейсы, так, чтобы потребности в новых продуктах не возникало (это не всегда возможно)
  - Или каким-либо образом параметризировать методы фабрики, создающие объекты, чтобы указывать, продукт какого типа (абстрактного) нужно создавать (при этом клиент должен уметь преобразовывать базовый тип продуктов к конкретному типу продукта, что небезопасно);
    - Если будет только один параметризируемый метод, условно CreateProduct(productType), совершенно не ясно, относить ли такое решение к абстрактной фабрике или же к фабричному методу

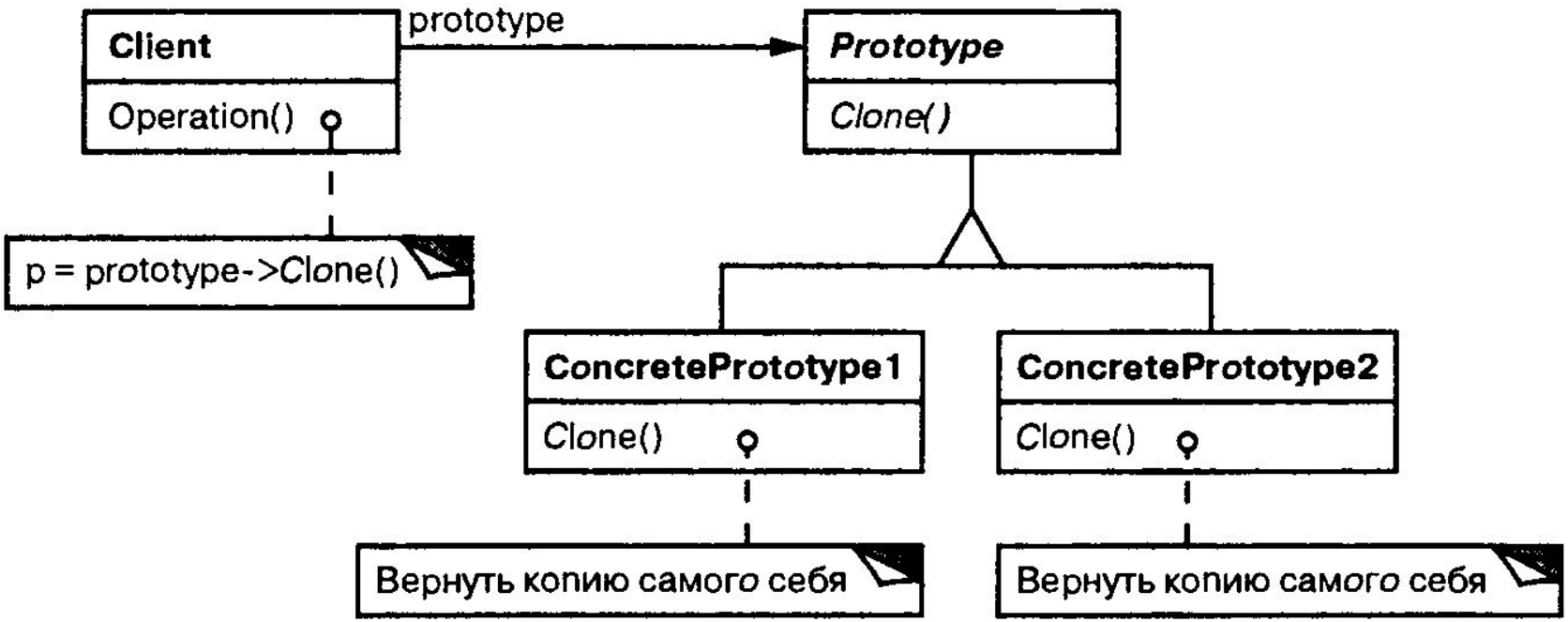
Абстрактная фабрика (Abstract Factory), замечания

- Нетрудно заметить, что абстрактная фабрика реализуется с помощью фабричных методов

Прототип (Prototype), задача

- Задает виды создаваемых объектов с помощью экземпляра-прототипа и создает новые объекты путем копирования этого прототипа
- Используется когда/для:
  - инстанцируемые классы определяются во время выполнения, например с помощью динамической загрузки
  - экземпляры класса могут находиться в одном из не очень большого числа различных состояний; может оказаться удобнее установить соответствующее число прототипов и клонировать их, а не инстанцировать каждый раз класс вручную в подходящем состоянии.

Прототип (Prototype), диаграмма классов



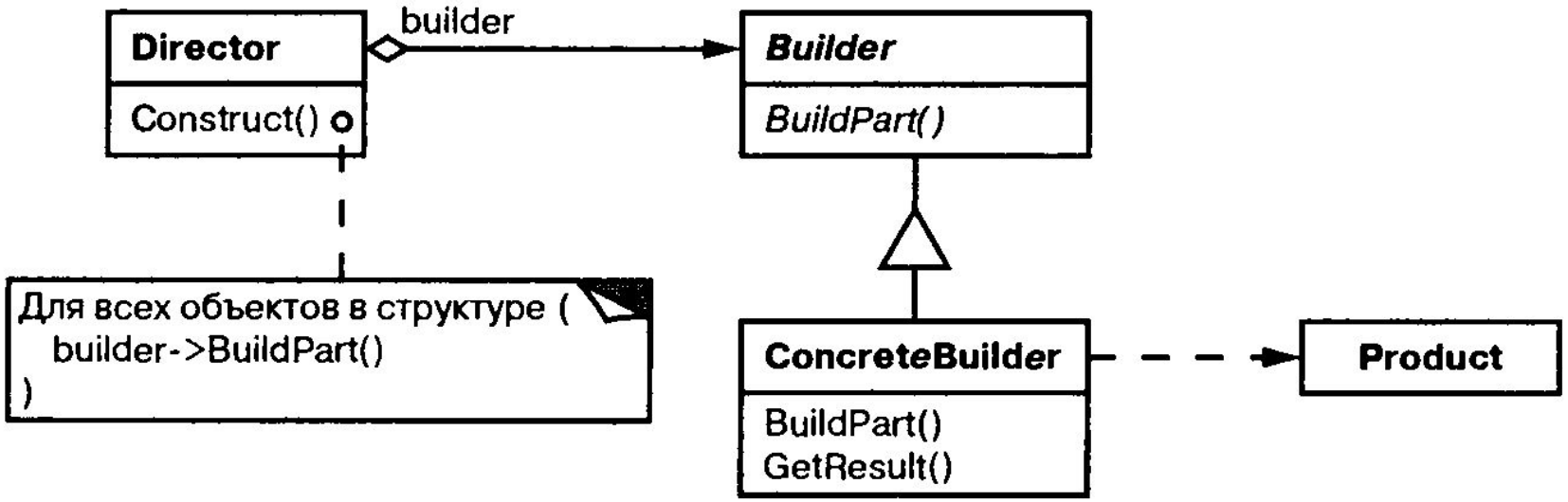
Прототип (Prototype), замечания

- Каждый подкласс Prototype должен реализовывать метод Clone()
  - В .NET FCL существует интерфейс IClonable, однако метод IClonable.Clone возвращает Object вместо Prototype
  - Простым способом реализовать метод Clone в C# иногда можно путем сериализации/десериализации объекта

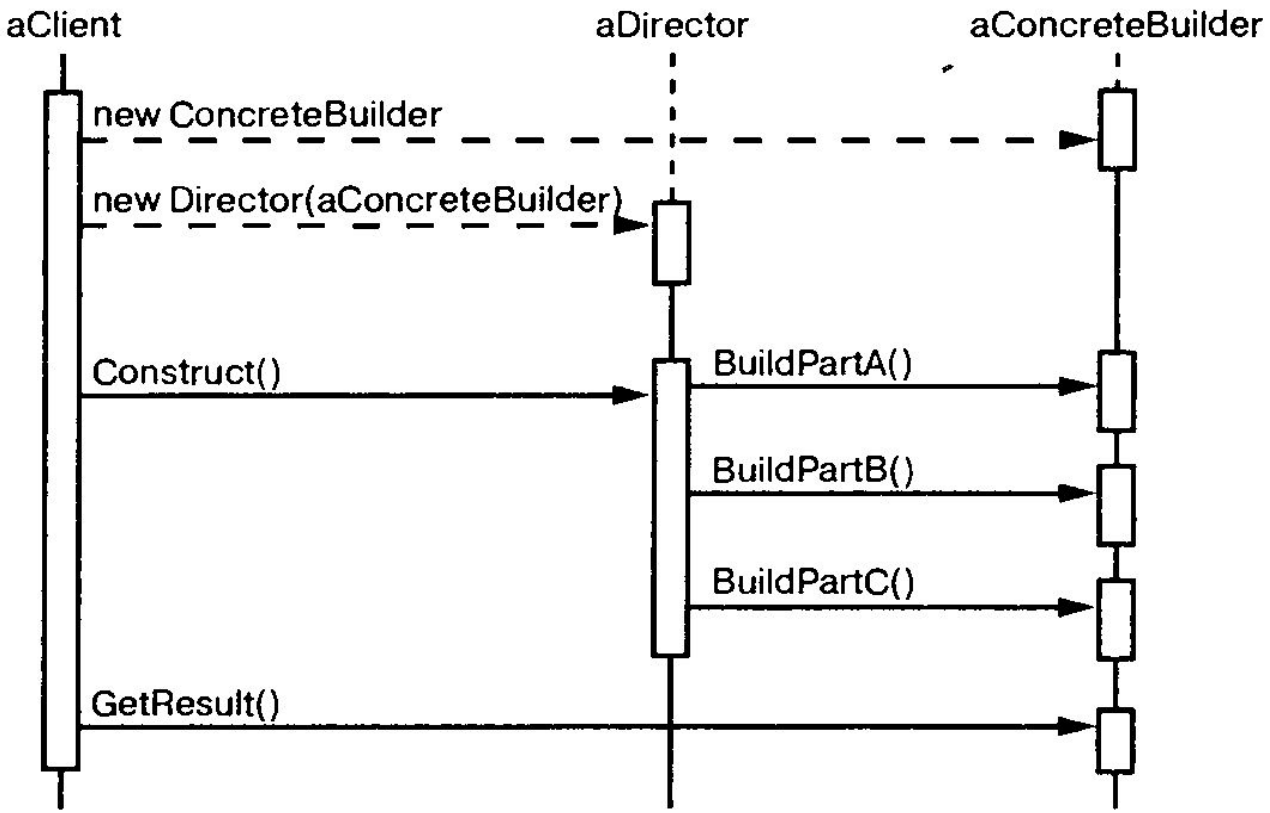
Строитель (Builder), задача

- Отделяет конструирование сложного объекта от его представления, так что в результате одного и того же процесса конструирования могут получаться разные представления
- Используется когда:
  - алгоритм создания сложного объекта не должен зависеть от того, из каких частей состоит объект и как они стыкуются между собой
  - процесс конструирования должен обеспечивать различные представления конструируемого объекта

Строитель (Builder), диаграмма классов



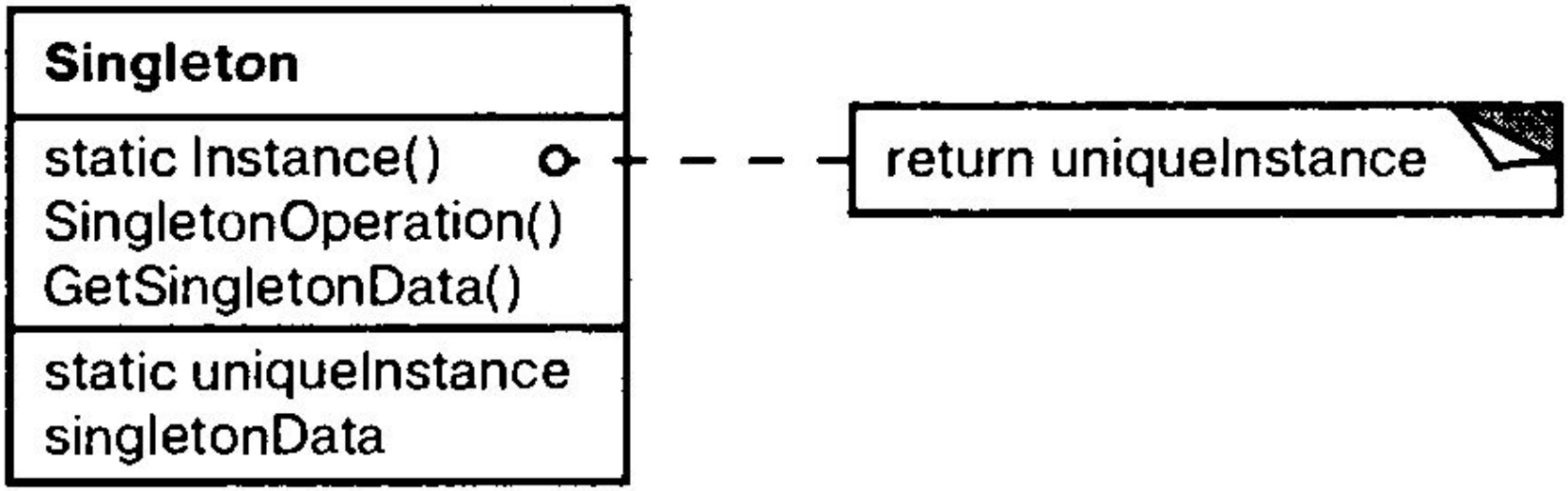
Строитель (Builder), диаграмма взаимодействия



Одиночка (Singleton), задача

- Используется, чтобы гарантировать единственность экземпляра класса
  - Например, в системе может быть много принтеров, но возможен лишь один спулер

Одиночка (Singleton), диаграмма классов



Одиночка (Singleton), пример кода

```
public class Singleton {
  private static Singleton instance;

  private int singletonData=10;


  protected Singleton() { // protected
  }

  public static Singleton Instance() {
     if(instance==null)
       instance=new Singleton();

     return instance;
  }


  public int SingletonData {
    get { return singletonData; }
  }
}
```

Одиночка (Singleton), замечания

- С помощью данного паттерна могут быть реализованы другие, например, абстрактная фабрика, строитель

Структурные паттерны

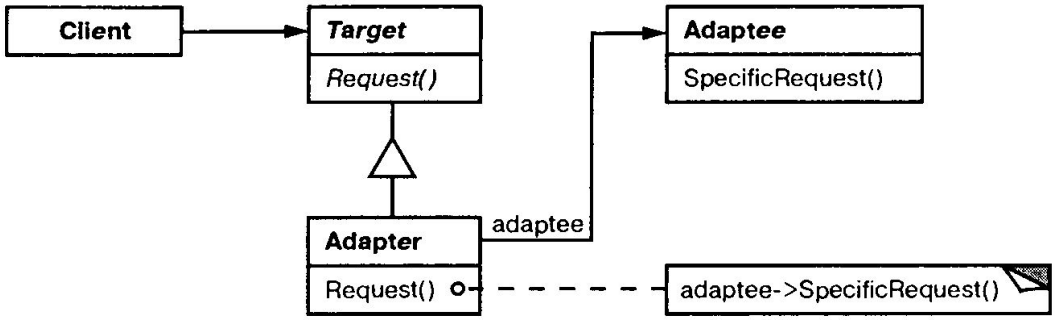
- Адаптер (Adapter)
- Декоратор (Decorator)
- Заместитель (Proxy)
- Компоновщик (Composite)
- Мост (Bridge)
- Приспособленец (Flyweight)
- Фасад (Facade)

Адаптер (Adapter), задача

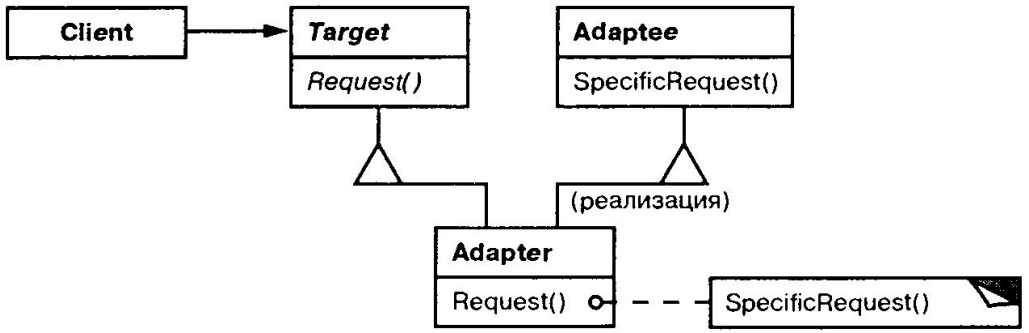
- Преобразует интерфейс одного класса в интерфейс другого, который ожидают клиенты
- Адаптер обеспечивает совместную работу классов с несовместимыми интерфейсами, которая без него была бы невозможна

Адаптер (Adapter), диаграмма классов

- 1-ый вариант (адаптер объекта) – композиция объектов



- 2-ой вариант (адаптер класса) – множественное наследование



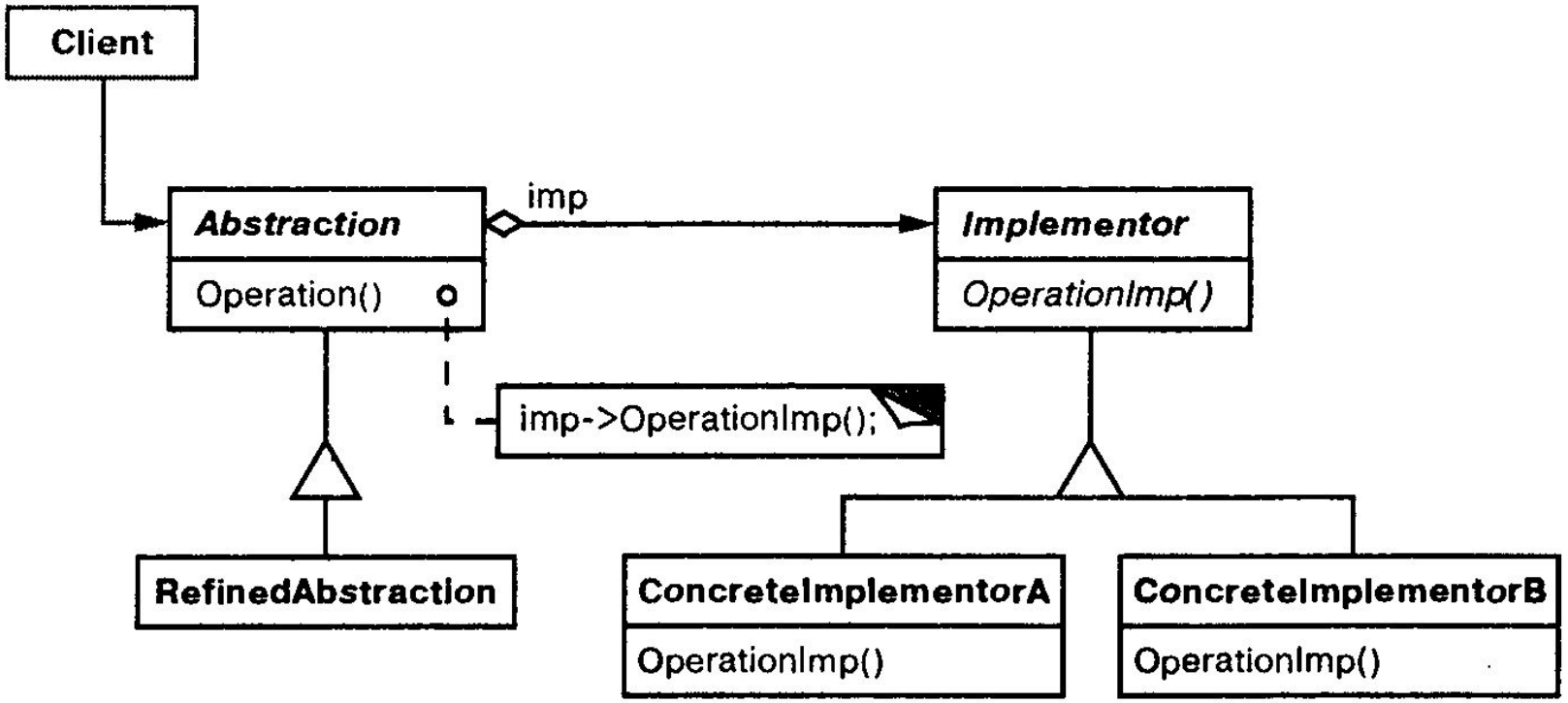
Адаптер (Adapter), пример: TextReader, StreamReader, Stream из FCL

- TextReader – Target
  - string ReadLine()
  - string ReadToEnd()
- StreamReader – Adapter
- Stream – Adaptee
  - int Read(byte[] buffer, int offset, int count)

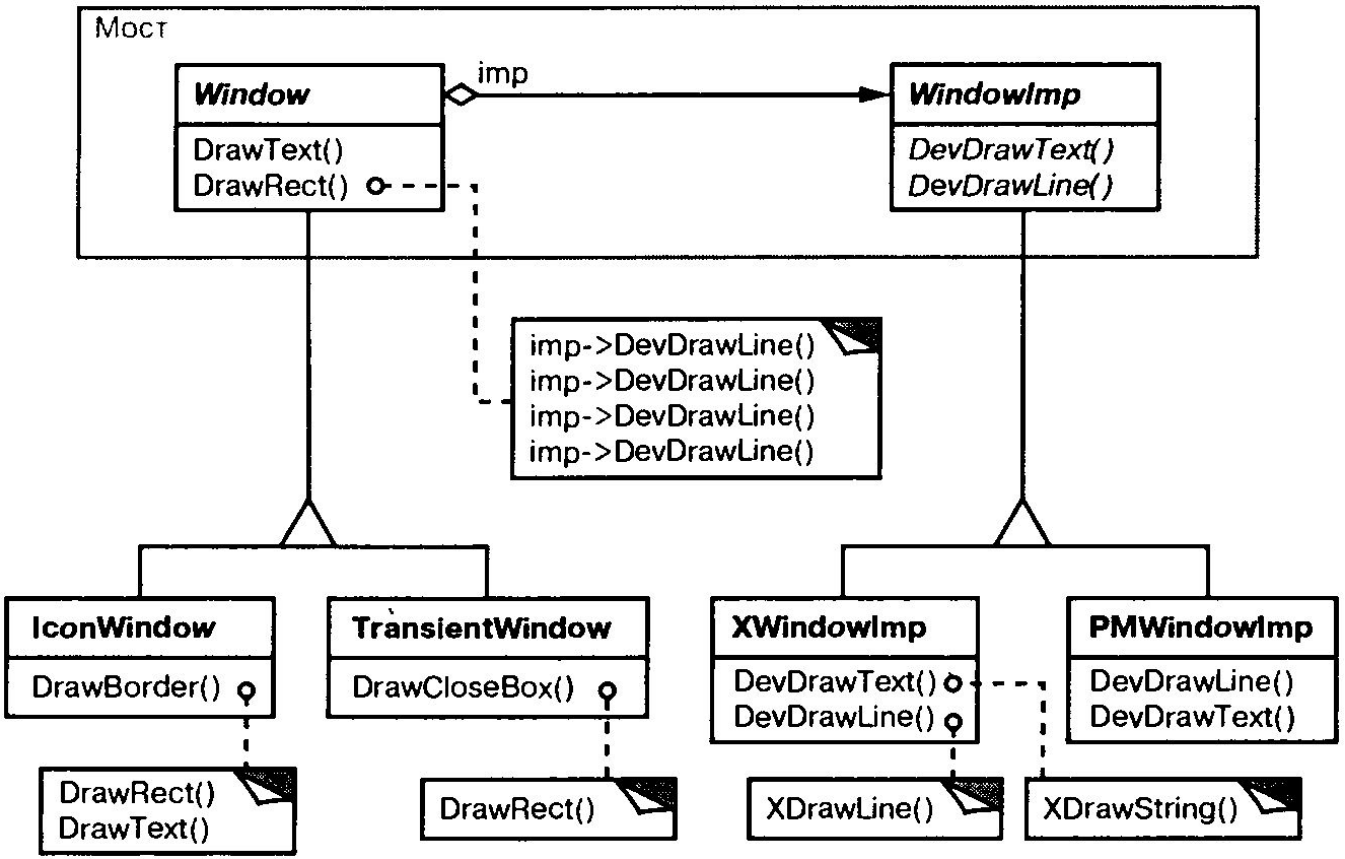
Мост (Bridge), задача

- Отделить абстракцию от ее реализации так, чтобы то и другое можно было изменять независимо
- Используется когда:
  - необходимо избежать постоянной привязки абстракции к реализации; так, например, бывает, когда реализацию необходимо выбирать во время выполнения программы
  - изменения в реализации абстракции не должны сказываться на клиентах, то есть клиентский код не должен перекомпилироваться
  - необходимо разделить одну реализацию между несколькими объектами и этот факт необходимо скрыть от клиента; простой пример – класс String, в котором разные объекты могут разделять одно и то же представление строки (StringRep)

Мост (Bridge), диаграмма классов



Мост (Bridge), пример с графическими библиотеками



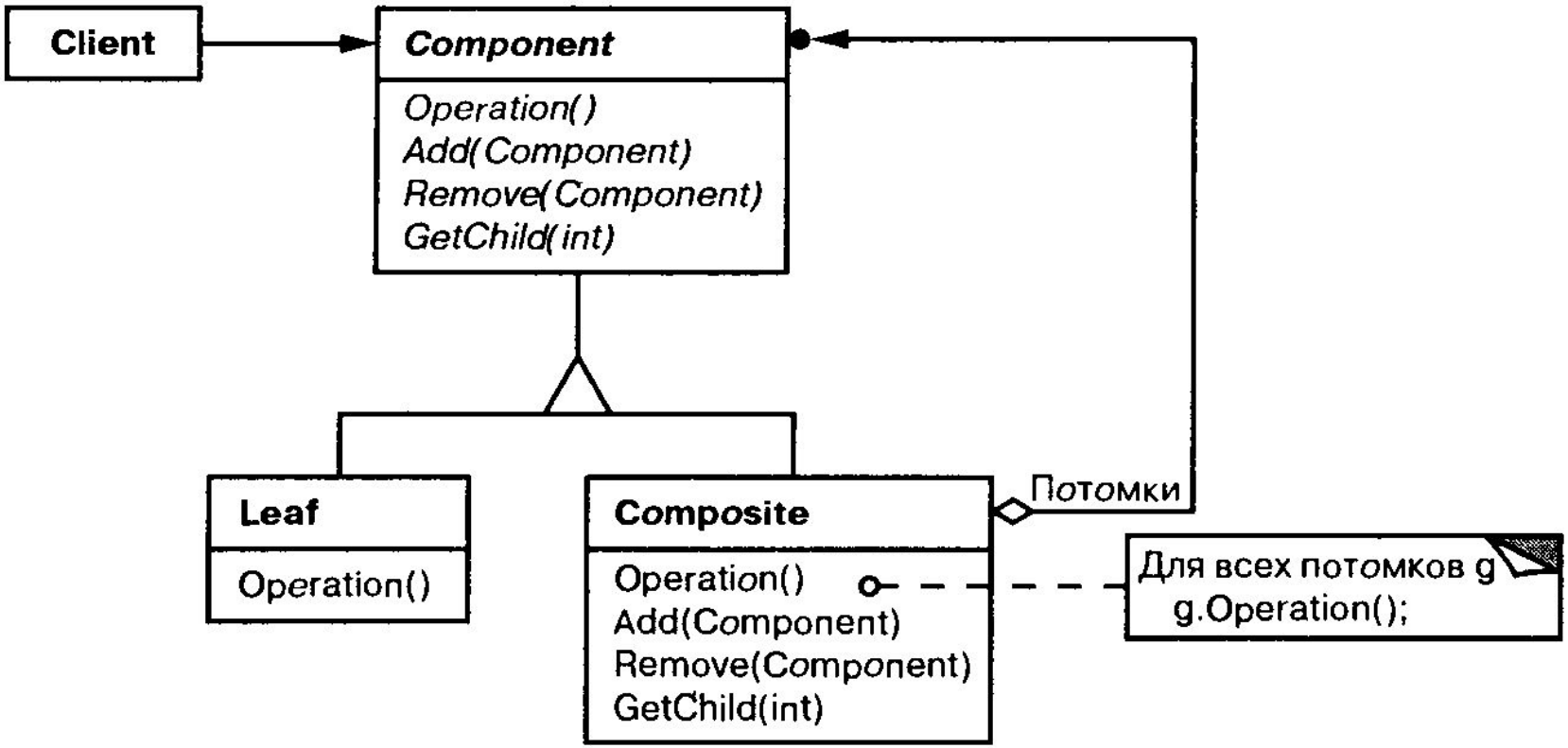
Мост (Bridge), замечания

- Конкретная реализация, как правило, будет порождаться одним из порождающих паттернов

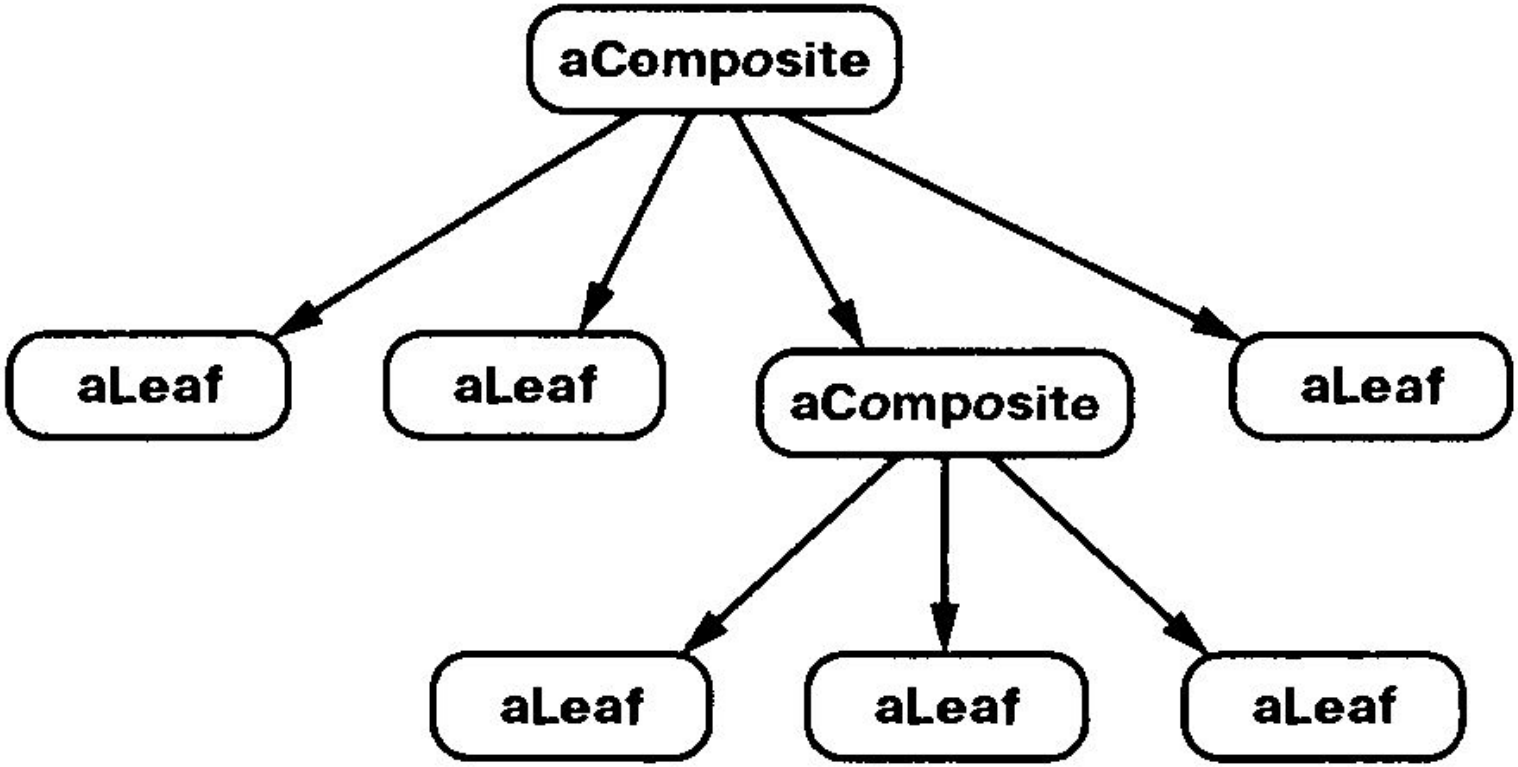
Компоновщик (Composite), задача

- Компонует объекты в древовидные структуры для представления иерархий часть-целое
- Позволяет клиентам единообразно трактовать индивидуальные и составные объекты
- Наиболее часто встречается в таких приложения, как графические редакторы, редакторы схем и т.п., где пользователь может объединить несколько объектов в единое целое и трактовать это целое как новый объект

Компоновщик (Composite), диаграмма классов



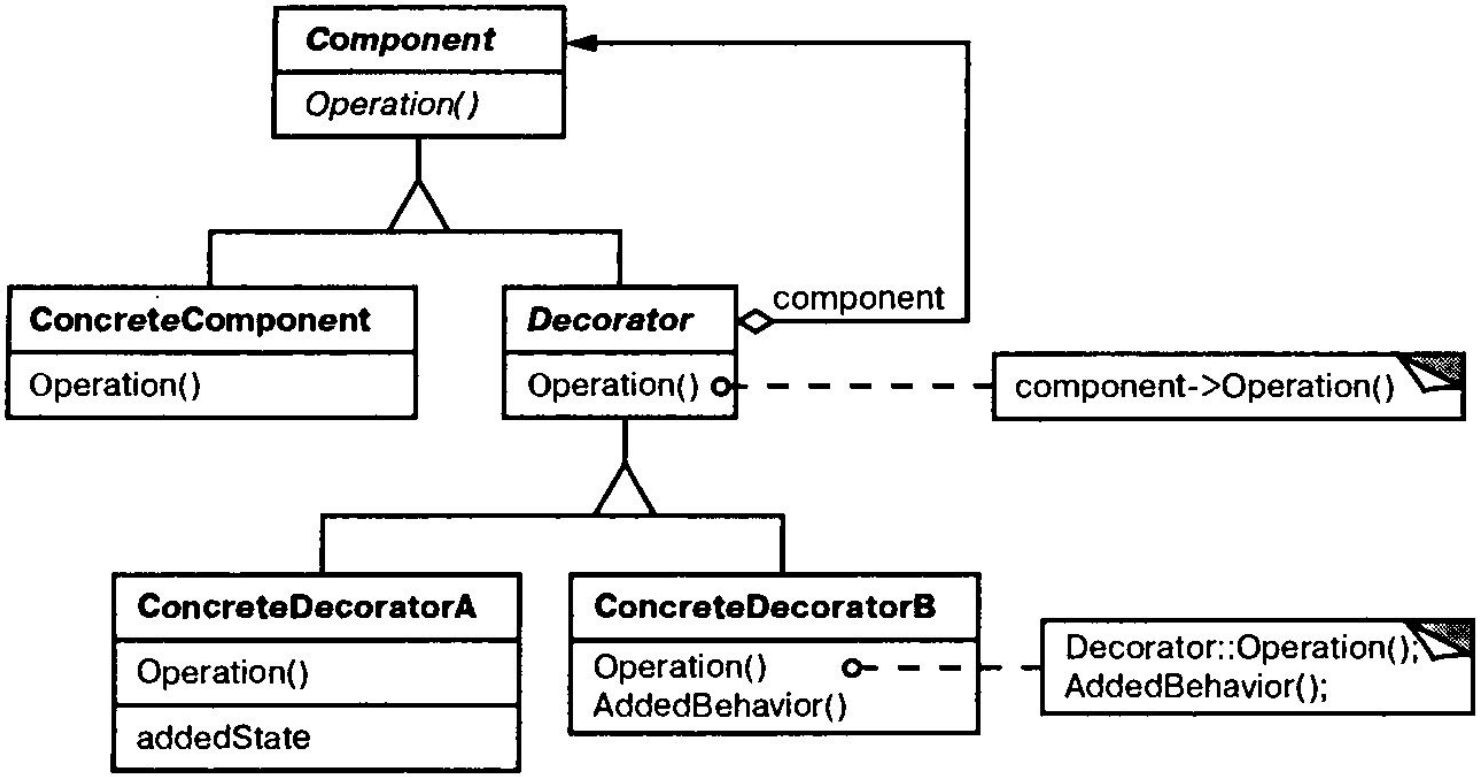
Компоновщик (Composite), диаграмма объектов



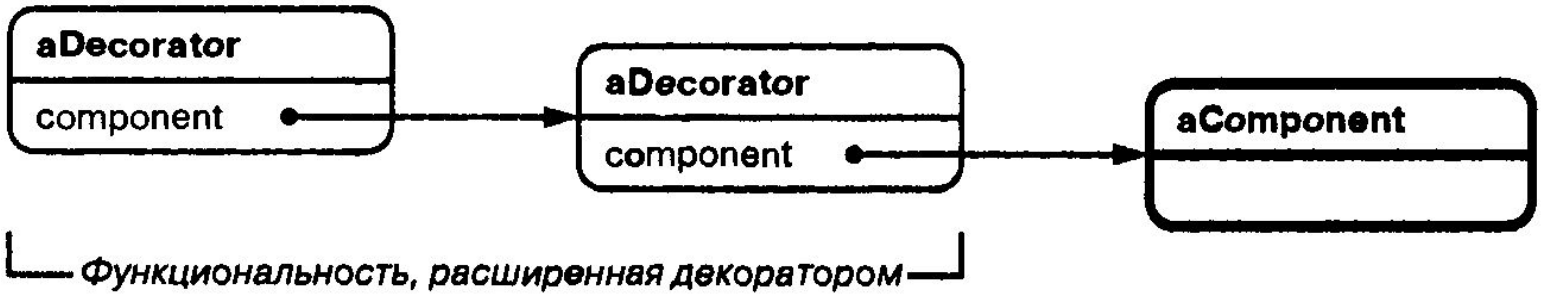
Декоратор (Decorator), задача

- Динамически добавляет объекту новые обязанности
- Является гибкой альтернативой порождению подклассов с целью расширения функциональности
- Используется когда/для:
  - динамического, прозрачного для клиентов добавления обязанностей объектам
  - реализации обязанностей, которые могут быть сняты с объекта
  - расширение путем порождения подклассов по каким-то причинам неудобно или невозможно

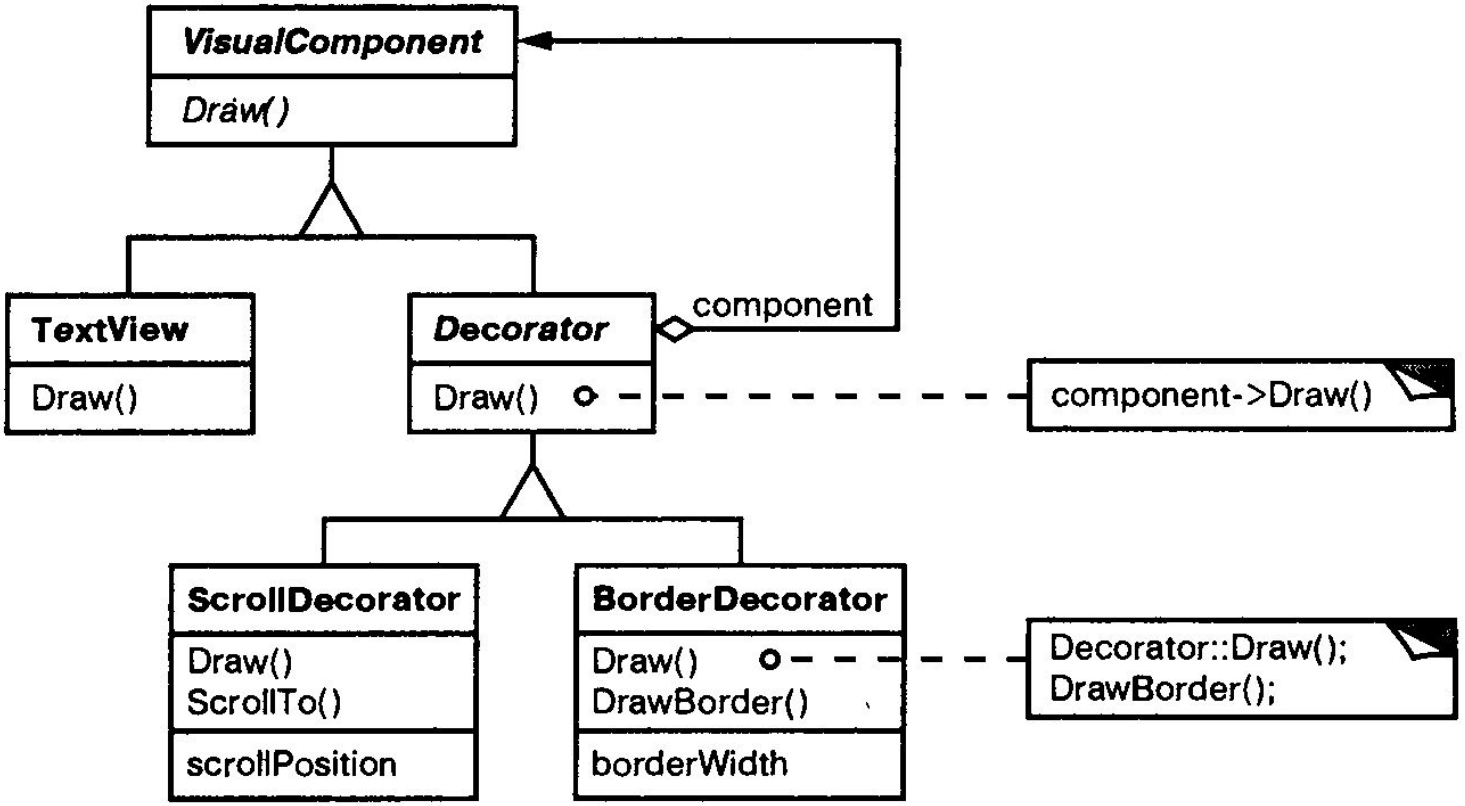
Декоратор (Decorator), диаграмма классов



Декоратор (Decorator), диаграмма объектов (цепочка декораторов)



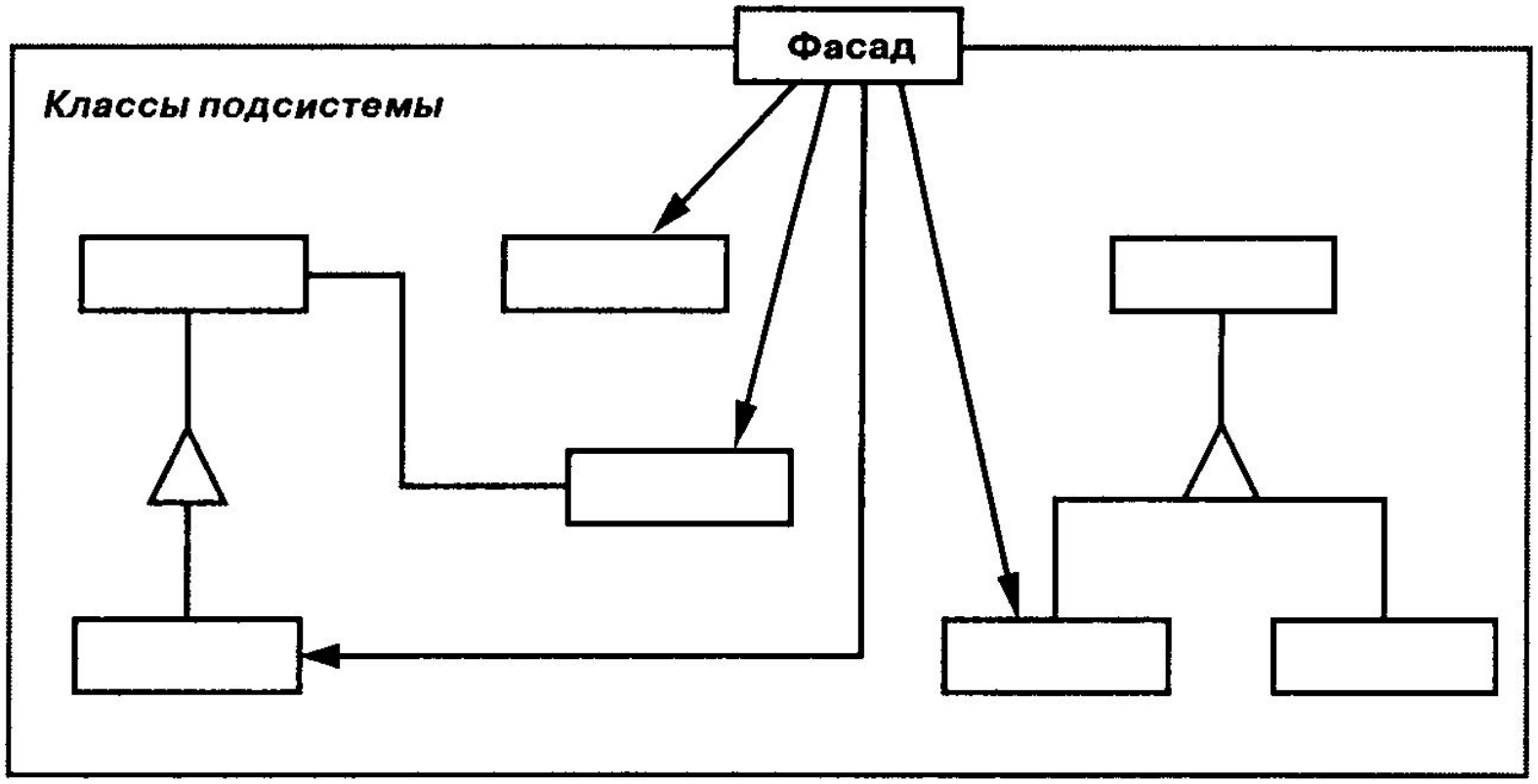
Декоратор (Decorator), пример с полосами прокрутки



Фасад (Facade), задача

- Предоставляет унифицированный интерфейс вместо набора интерфейсов некоторой подсистемы
- Определяет интерфейс более высокого уровня, который упрощает использование подсистемы

Фасад (Facade), возможная диаграмма классов



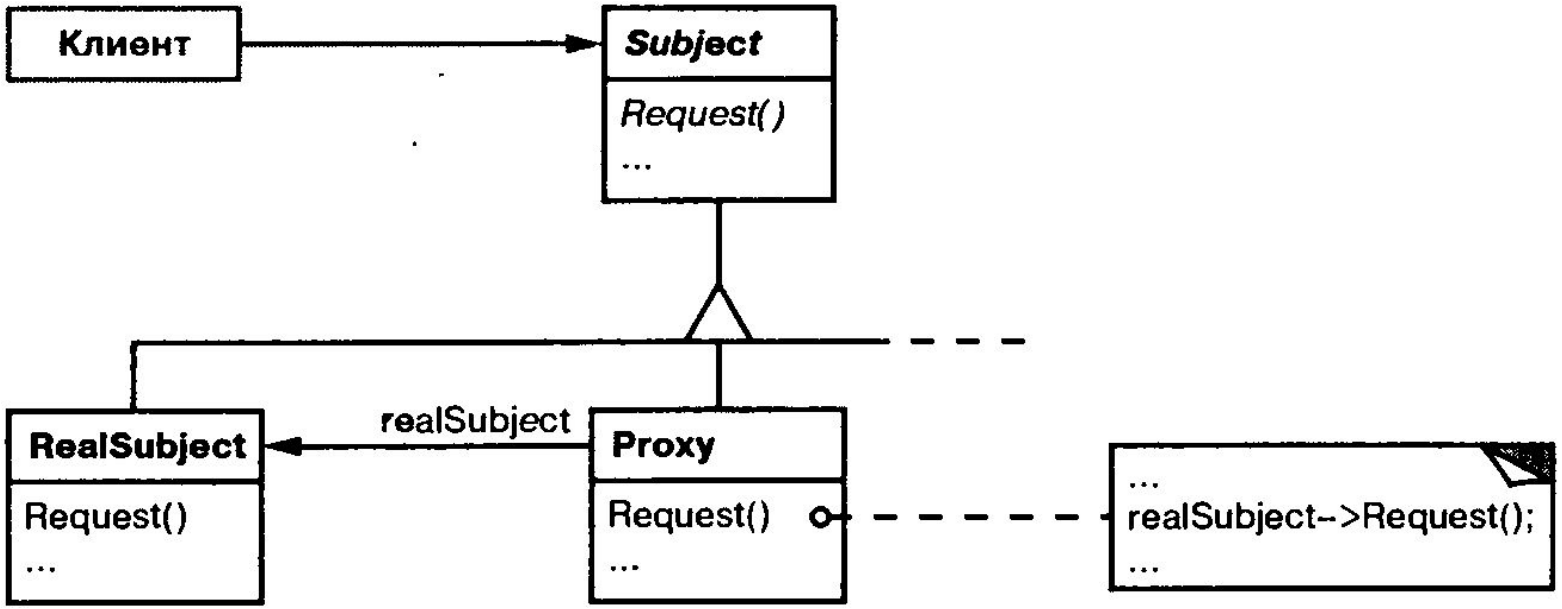
Приспособленец (Flyweight)

- Использует разделение для эффективной поддержки множества мелких объектов

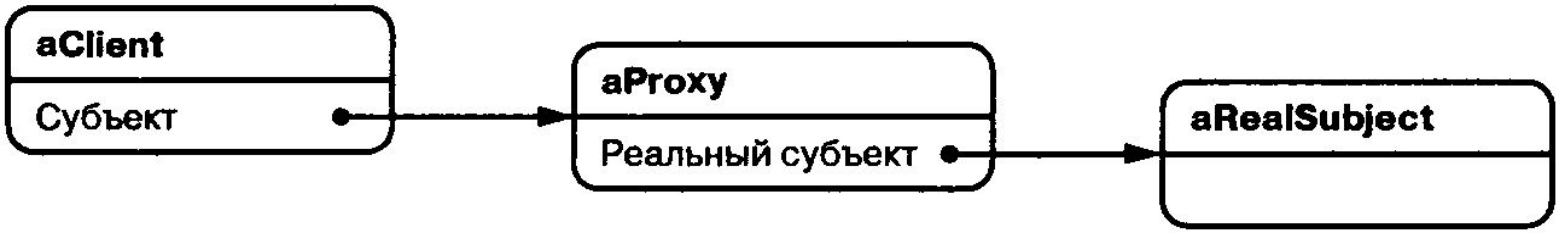
Заместитель (Proxy), задача

- Является суррогатом другого объекта и контролирует доступ к нему
- Используется в случаях:
  - удаленный заместитель предоставляет локального представителя вместо объекта, находящегося в другом адресном пространстве
  - виртуальный заместитель создает «тяжелые» объекты по требованию
  - защищающий заместитель контролирует доступ к исходному объекту
  - «умная» ссылка – это замена обычного указателя; позволяет выполнить дополнительные действия при доступе к объекту

Заместитель (Proxy), диаграмма классов



Заместитель (Proxy), диаграмма объектов



Заместитель и классы UnmutableList, UnmutableMap и т.д.

- Классы UnmutableList, UnmutableMap и т.д. из 1-го практического задания – типичные заместители
  - Контролируют доступ к используемому объекту

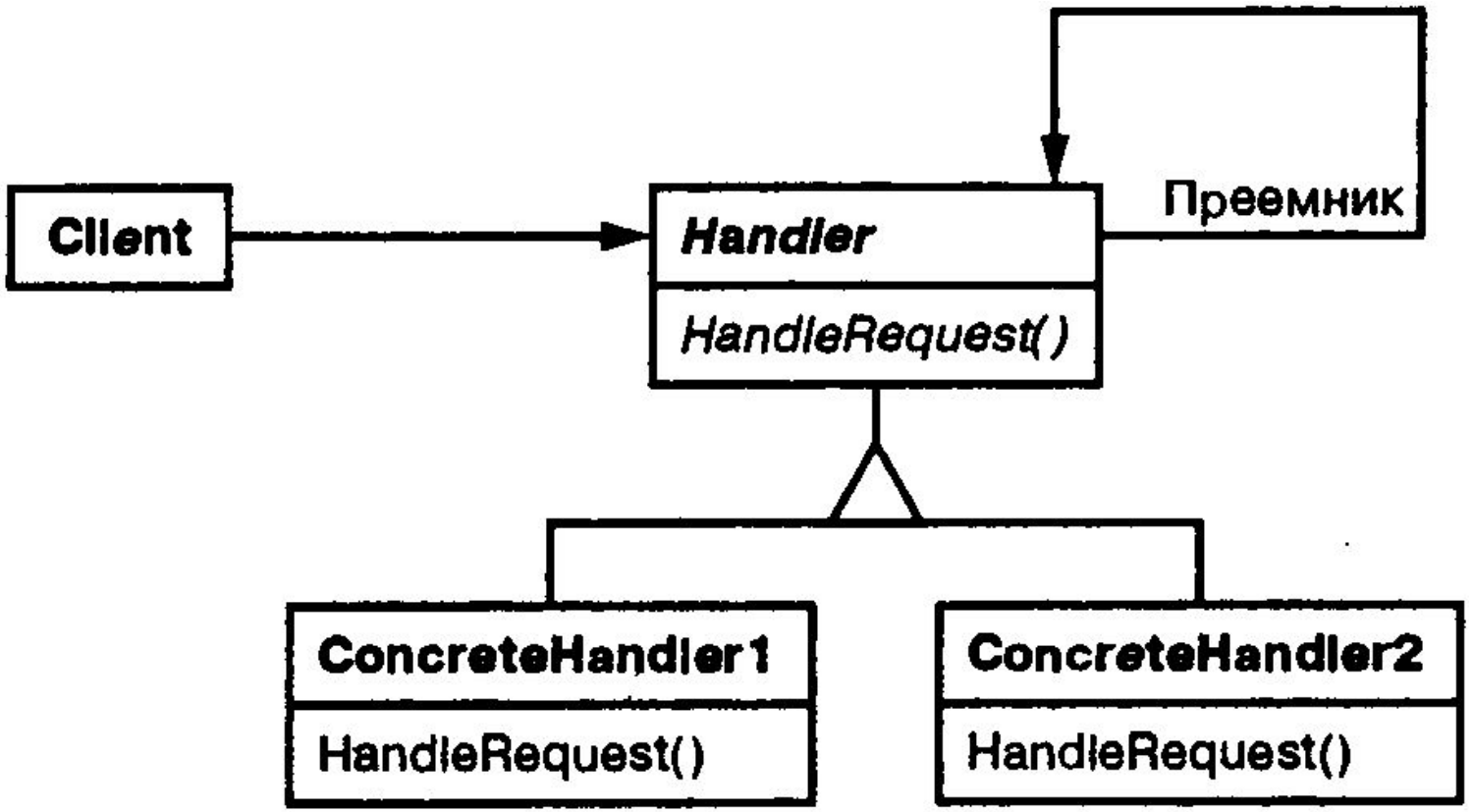
Паттерны поведения

- Интерпретатор (Interpreter)
- Итератор (Iterator)
- Команда (Command)
- Наблюдатель (Observer)
- Посетитель (Visitor)
- Посредник (Mediator)
- Состояние (State)
- Хранитель (Memento)
- Цепочка обязанностей (Chain of Responsibility)
- Шаблонный метод (Template Method)

Цепочка обязанностей (Chain of Responsibility), задача

- Позволяет избежать привязки отправителя запроса к его получателю, давая шанс обработать запрос нескольким объектам
- Связывает объекты-получатели в цепочку и передает запрос вдоль этой цепочки, пока его не обработают

Цепочка обязанностей (Chain of Responsibility), диаграмма классов



Цепочка обязанностей (Chain of Responsibility), диаграмма объектов

- Следует учитывать, что объекты, участвующие в паттерне, могут быть скомпонованы и другим способом, например, просто храниться в массиве

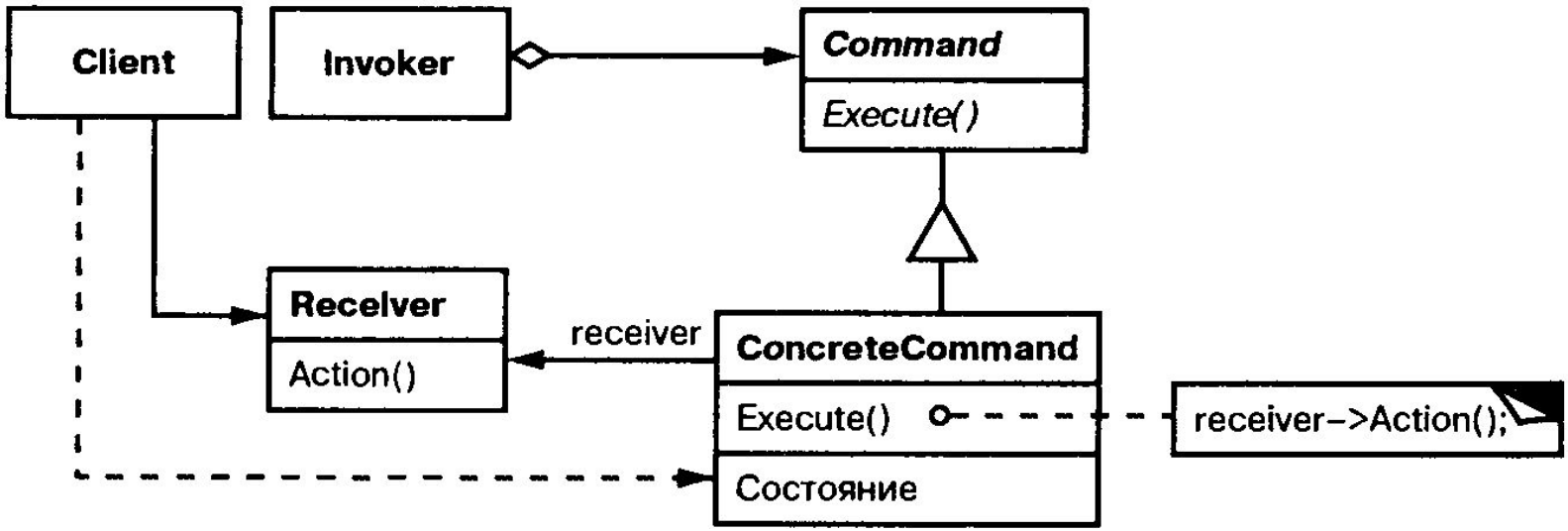
Цепочка обязанностей (Chain of Responsibility), результаты

- ослабление связанности
- дополнительная гибкость при распределении обязанностей между объектами
- получение не гарантировано

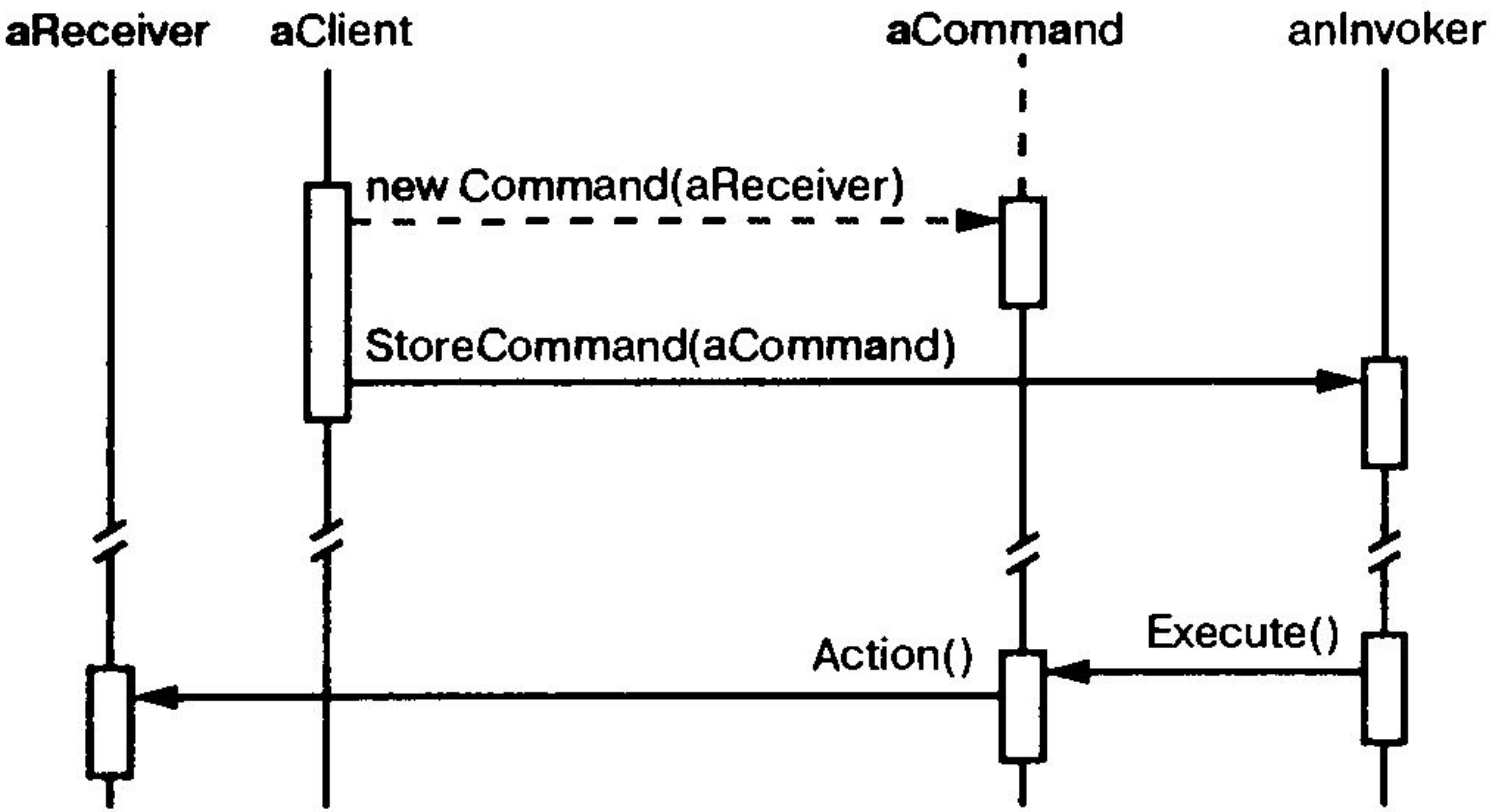
Команда (Command), задача

- Инкапсулирует запрос как объект, позволяя тем самым задавать параметры клиентов для обработки соответствующих запросов, ставить запросы в очередь или протоколировать их, а также поддерживать отмену операций
- Известен также под именем Action

Команда (Command), диаграмма классов



Команда (Command), диаграмма взаимодействия



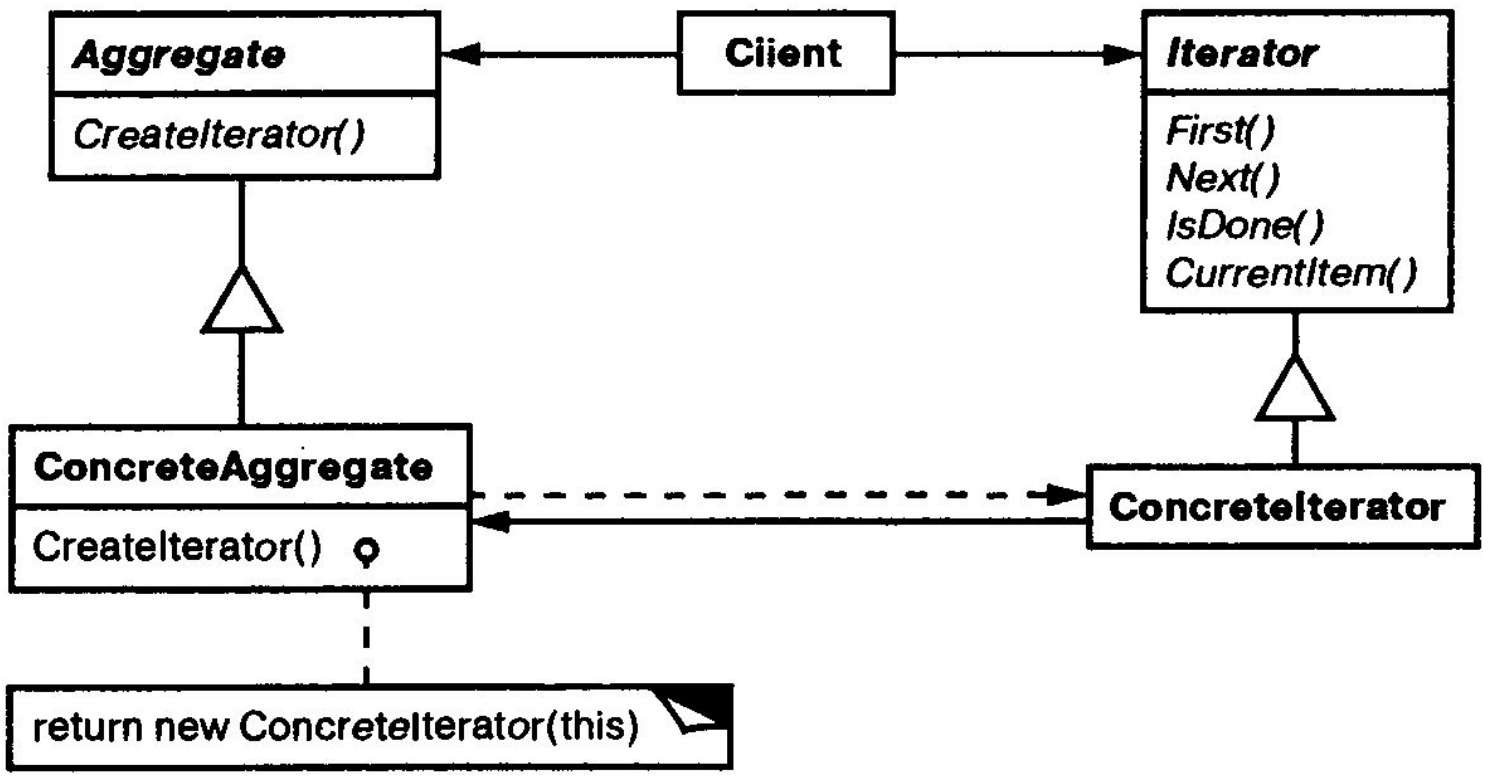
Интерпретатор (Interpreter), задача

- Для заданного языка определяет представление его грамматики, а также интерпретатор предложений этого языка

Итератор (Iterator), задача

- Предоставляет способ последовательного доступа ко всем элементам составного объекта, не раскрывая его внутреннего представления
- Используется для:
  - для предоставления единообразного интерфейса с целью обхода различных агрегированных структур (то есть для поддержки полиморфной итерации)
  - для поддержки различные виды обхода агрегата
  - для поддержки нескольких активных обходов одного и того же агрегированного объекта
- Известен также под именем Cursor

Итератор (Iterator), диаграмма классов



Итератор (Iterator), возникающие вопросы

- Кто из участников управляет итерацией?
  - Клиент (внешний или пассивный итератор)
  - Агрегат (внутренний или активный итератор)
    - В этом случае часто можно говорить о паттерне Посетитель (Visitor)
- Что определяет алгоритм обхода?
- Насколько итератор устойчив?
  - Различные проблемы могут возникнуть при модификации агрегата (добавлении, удалении элементов и т.п.) во время активности итератора
- И т.д.

Итератор (Iterator), реализация в C#

- Итераторы могут быть обобщенными - IEnumerable<T>
- Языковые средства для итерации по объектам итератора

```
foreach(Student student in studentGroup) {
  …
}
```

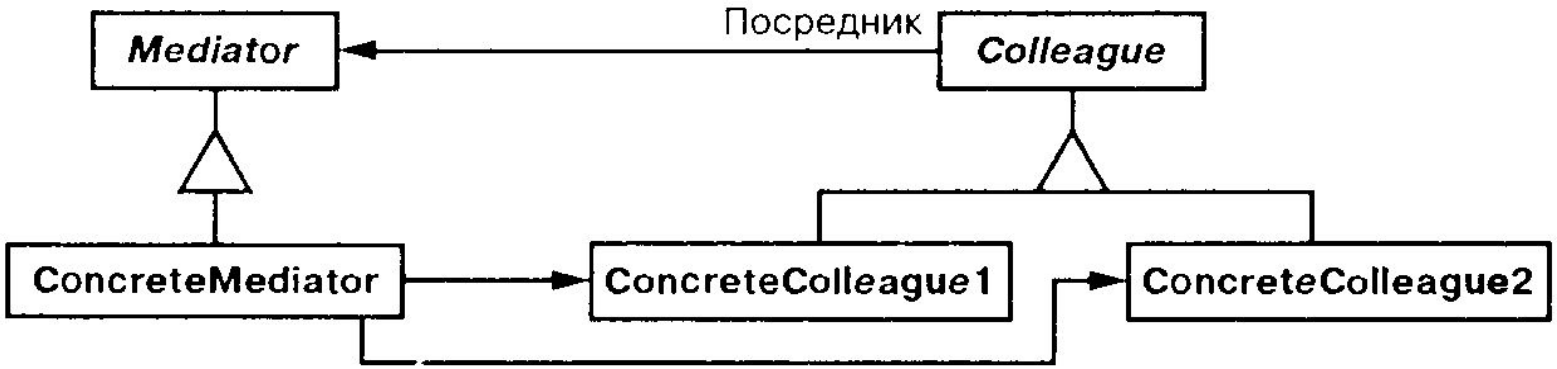
- Языковые средства для упрощения создания итераторов

```
public class StudentGroup {
  …
  public IEnumerable<Student> Students {
    get {
      for(int i=0; i<count; i++)
        return yield students[i];
    }
  }
```

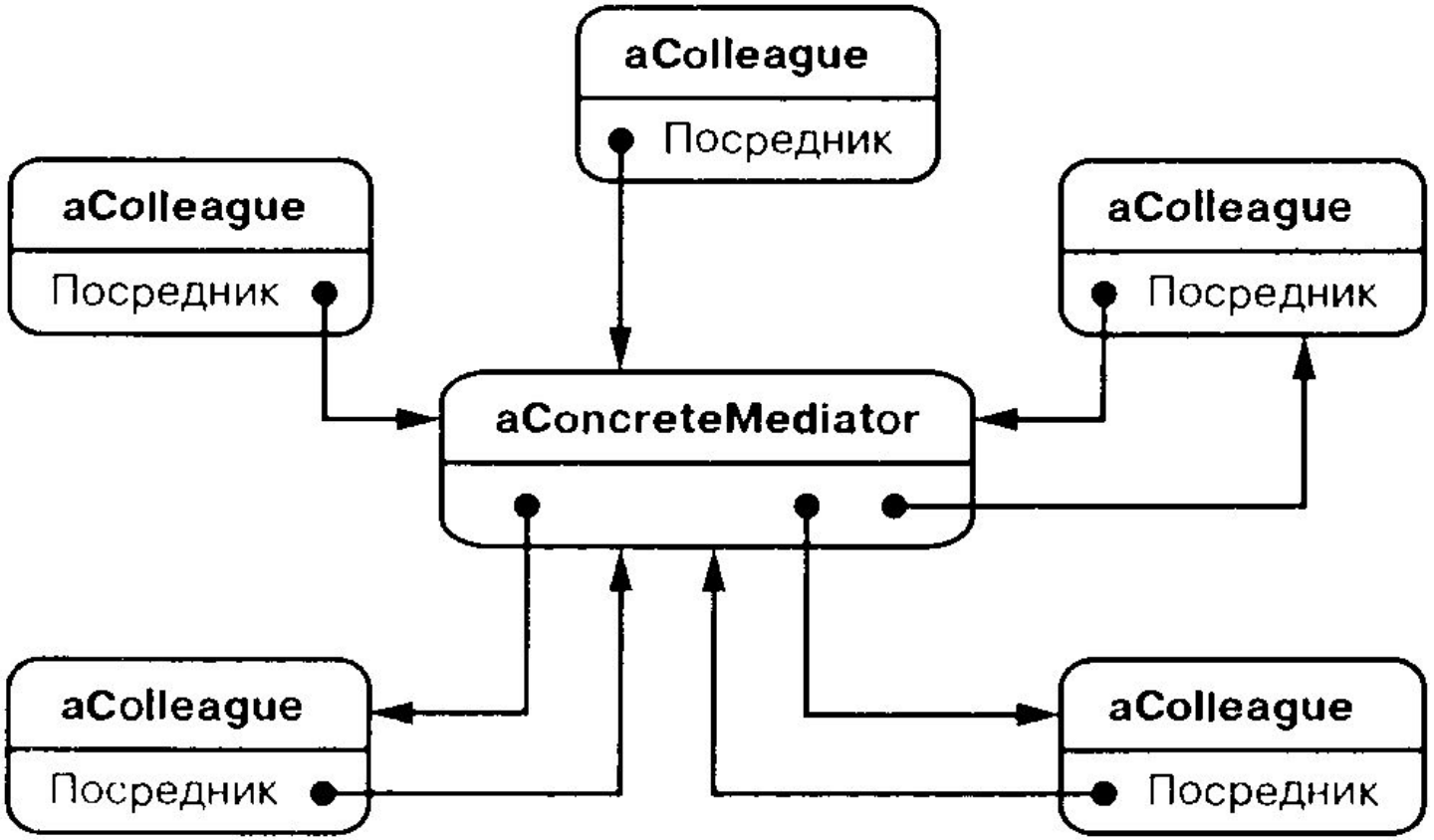
Посредник (Mediator), задача

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

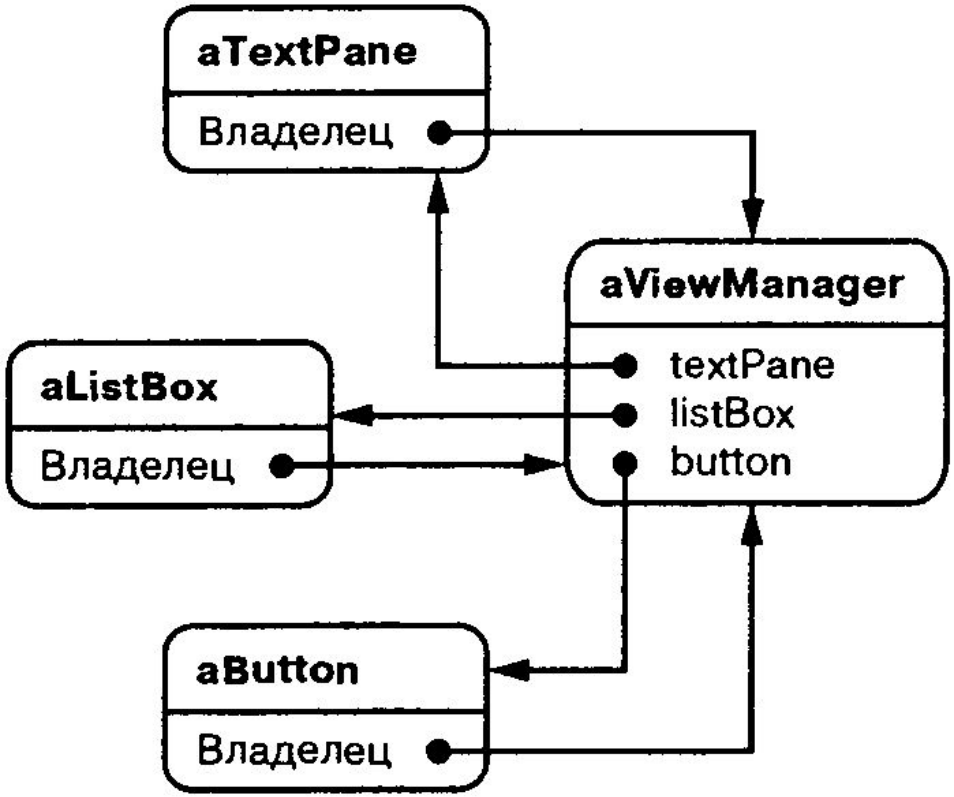
Посредник (Mediator), диаграмма классов



Посредник (Mediator), диаграмма объектов



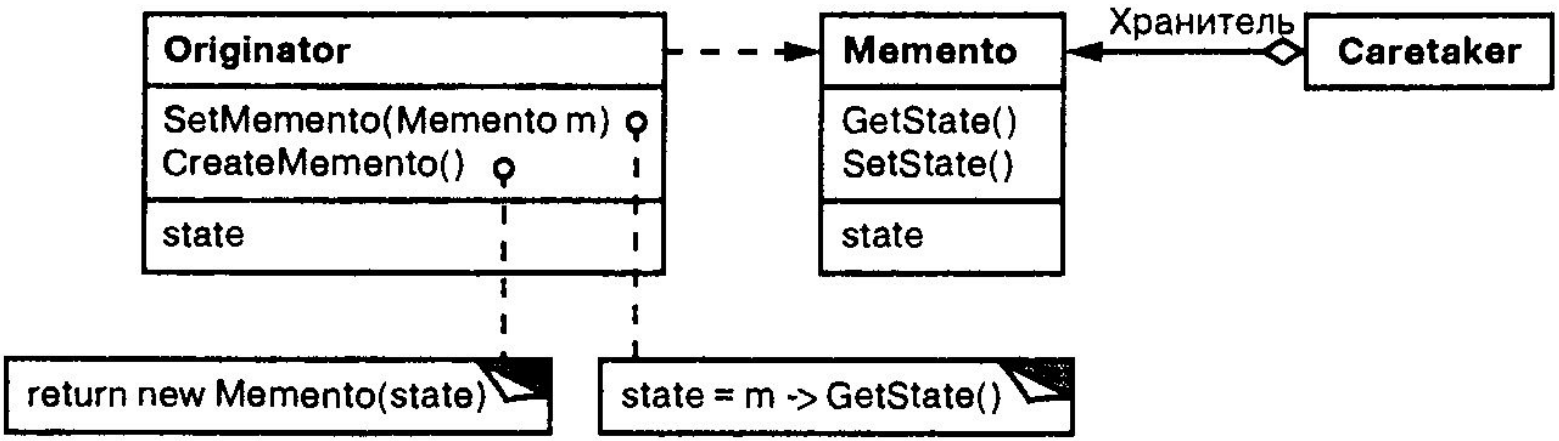
Посредник (Mediator), пример с диалоговым окном



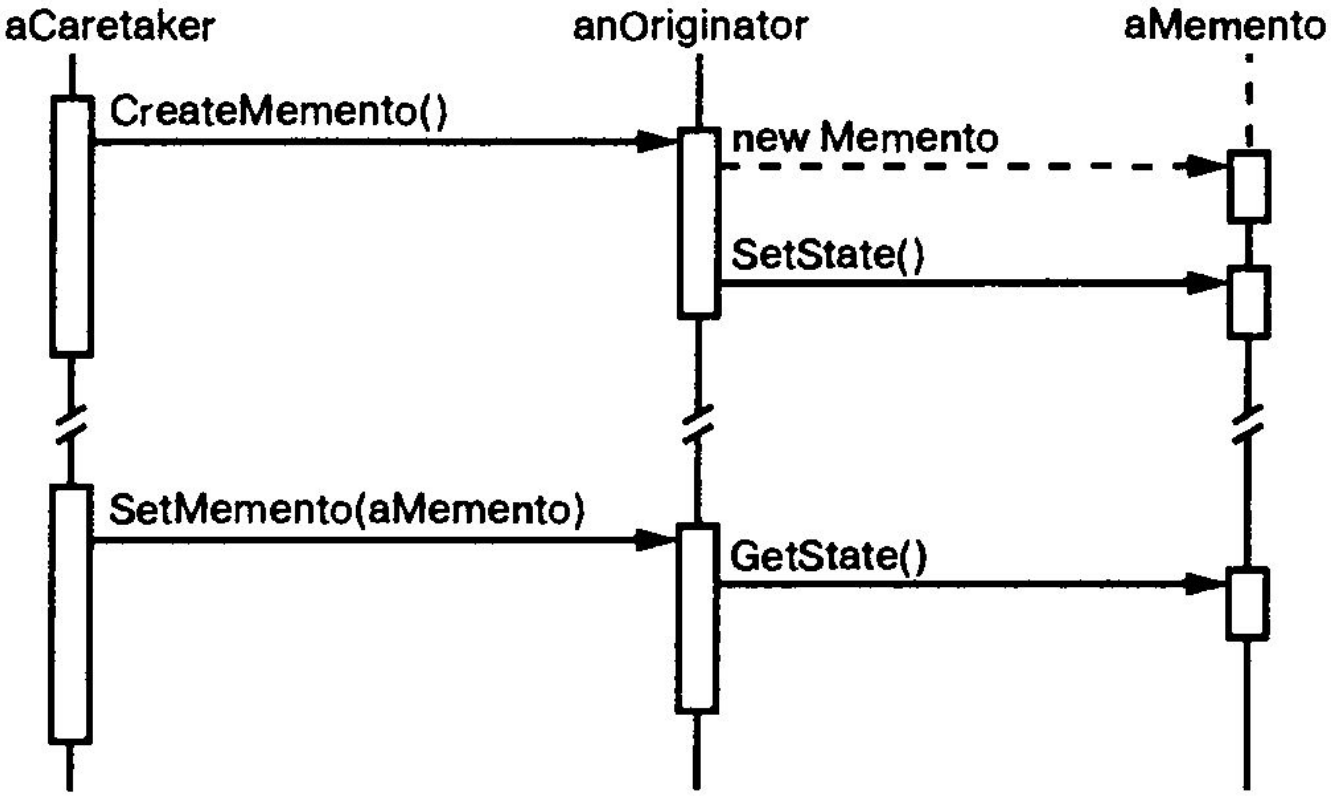
Хранитель (Memento), задача

- Не нарушая инкапсуляции, фиксирует и выносит за пределы объекта его внутреннее состояние так, чтобы позднее можно было восстановить в нем объект

Хранитель (Memento), диаграмма классов



Хранитель (Memento), диаграмма взаимодействия



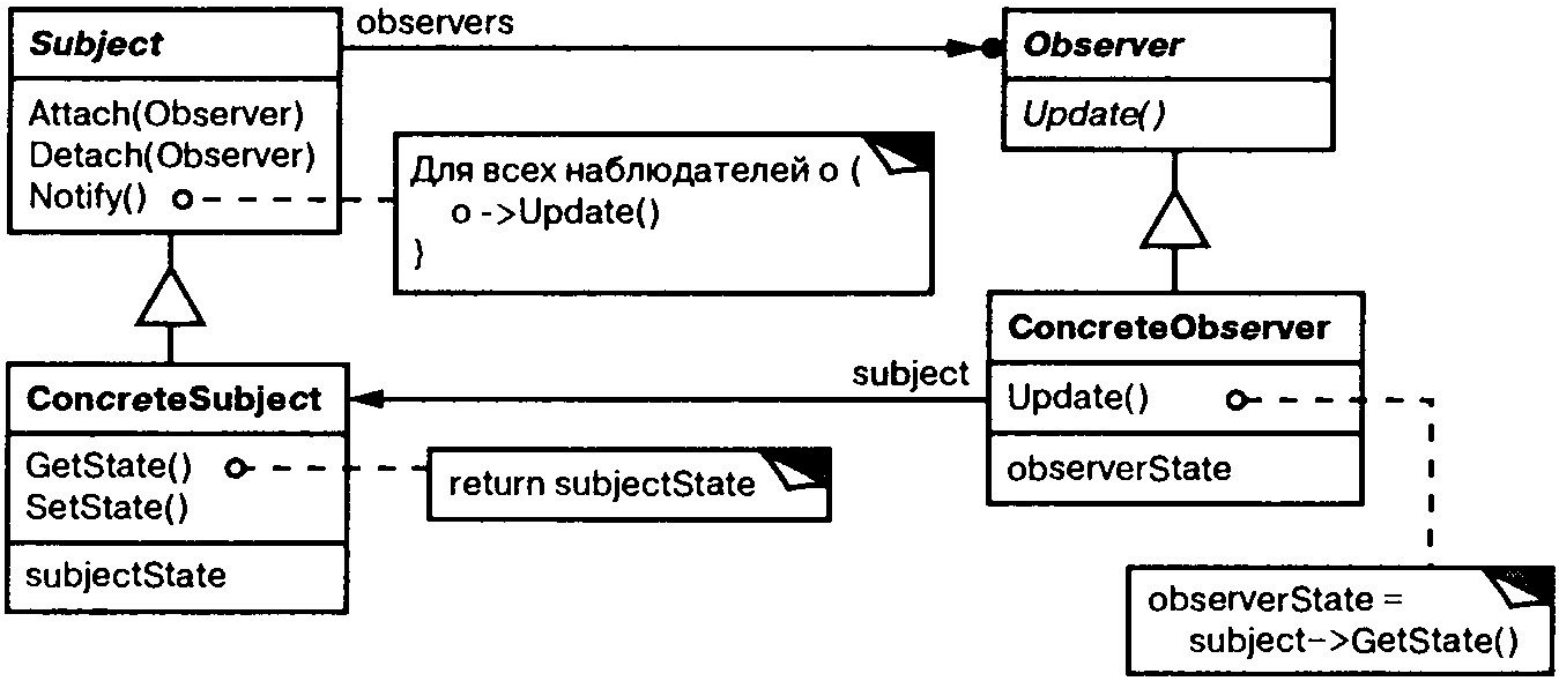
Хранитель (Memento), Реализация в .NET

- Часто вместо паттерна Хранитель можно использовать механизм сериализации/десериализации

Наблюдатель (Observer), задача

- 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
- Известен также под именем Listener

Наблюдатель (Observer), диаграмма классов



Наблюдатель (Observer), диаграмма взаимодействия