Технология разработки программного обеспечения (вторая часть)

Структурные шаблоны проектирования ПО

2. Структурные паттерны

- Описывают способы построение сложных структур из классов и объектов.

1. **Adapter**
2. Bridge
3. **Façade**
4. **Composite**
5. Decorator
6. Flyweight
7. Proxy

Паттерн Adapter (Адаптер)

- ***Цель паттерна*** Adapter (адаптер) – привести (адаптировать) интерфейс некоторого адаптируемого класса к интерфейсу, который ожидается клиентом.
- Основные понятия:
  - *клиент (client)* -- класс, который использует (в общем случае агрегирует) некоторый класс, который мы называем *адаптируемым (adaptee).*
  - *адаптер (adapter)* -- класс, выполняющий приведение интерфейса адаптируемого класса к интерфейсу, ожидаемому клиентом.

Причина возникновения паттерна

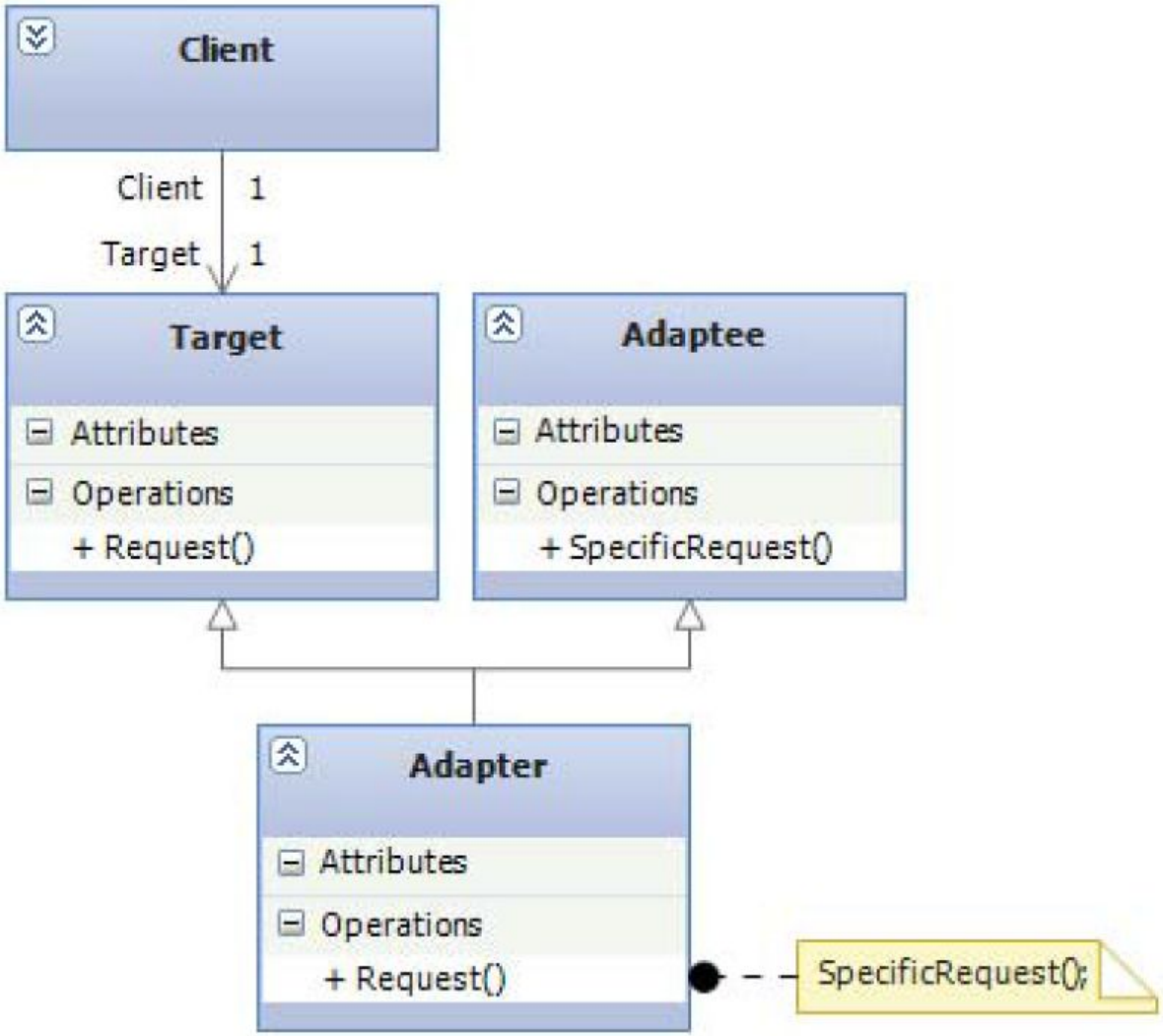
- Проблема: имеется некоторый класс, который нужно использовать в необычной для его структуры задаче.
- Например, есть тип данных, описывающий понятие сетевого устройства (IPEndPoint), который имеет такими свойствами как
  - IP- адрес,
  - мак-адрес и
  - имя хоста.
- Его нужно использовать для решения некоторой задачи (например, трассировки перемещения пакетов).

Причина возникновения паттерна

- Решение выполняется классом NetView, который агрегирует множество объектов типа IPEndPoint.
- Нужно
  - выполнить графическое представление для процесса и результата анализа,
  - вывести его в окно приложения.

- Проблема в том, что класс NetView
  - не имеет интерфейса, специфичного для объекта графической подсистемы
  - поэтому не может быть использован оконным классом для выполнения прорисовки.
- Нельзя изменить исходный текст (структуру) класс NetView
  - он является частью, используемого нами набора типов из dll-библиотеки предоставленной сторонними разработчиками;
  - или нельзя «изменять» структуру класса, т.к. она используется другими задачами приложения.

Структура паттерна Adapter



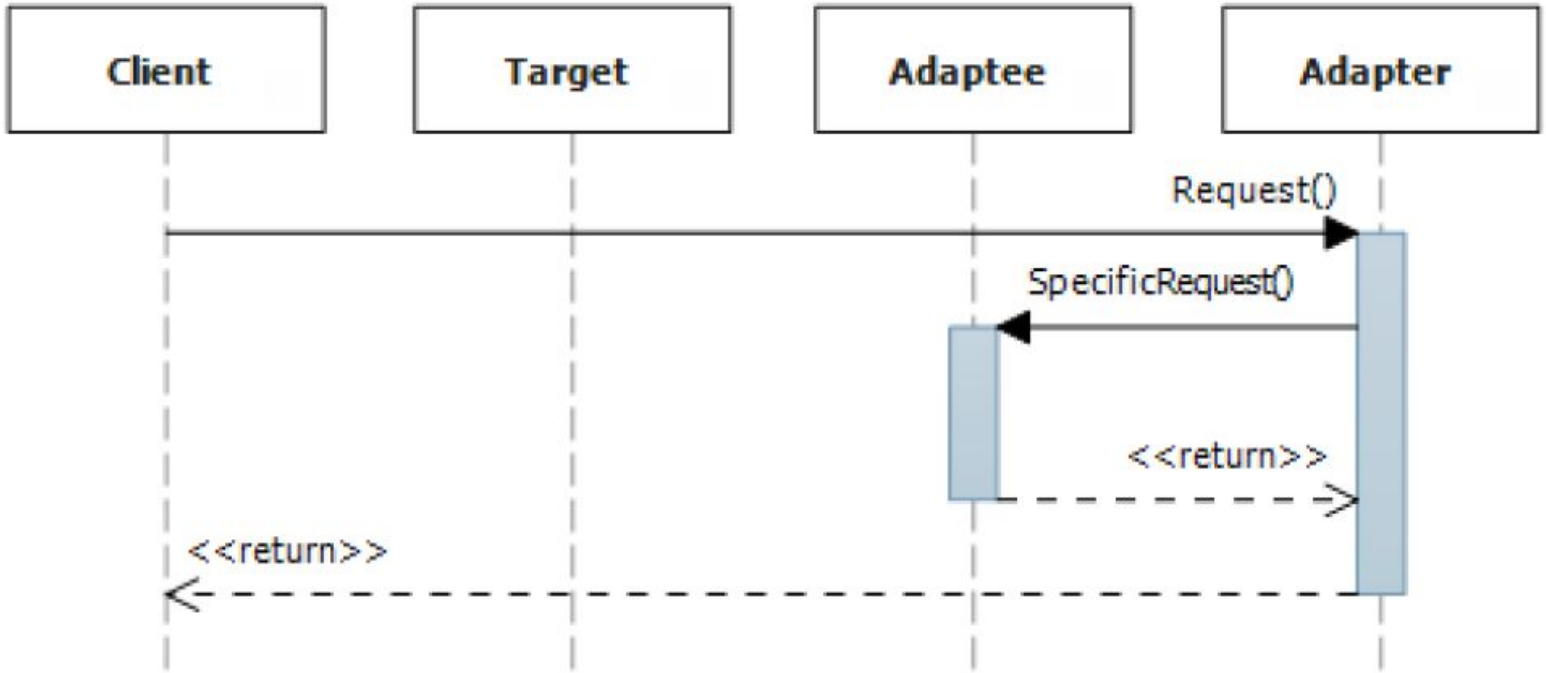
Участвующие элементы

- *Client* - класс, который использует некоторые вспомогательные типы данных и ожидает, что они имеют стандартный интерфейс взаимодействия (использования) описанный классом Target.
- *Target* - класс, имеющий интерфейс, ожидаемый клиентом.
- *Adaptee* - класс, необходимый для работы клиента, но имеет интерфейс, отличный от того, который ожидается клиентом.
- *Adapter* - класс, выполняющий приведение интерфейса класса Adaptee, к интерфейсу класса Target.

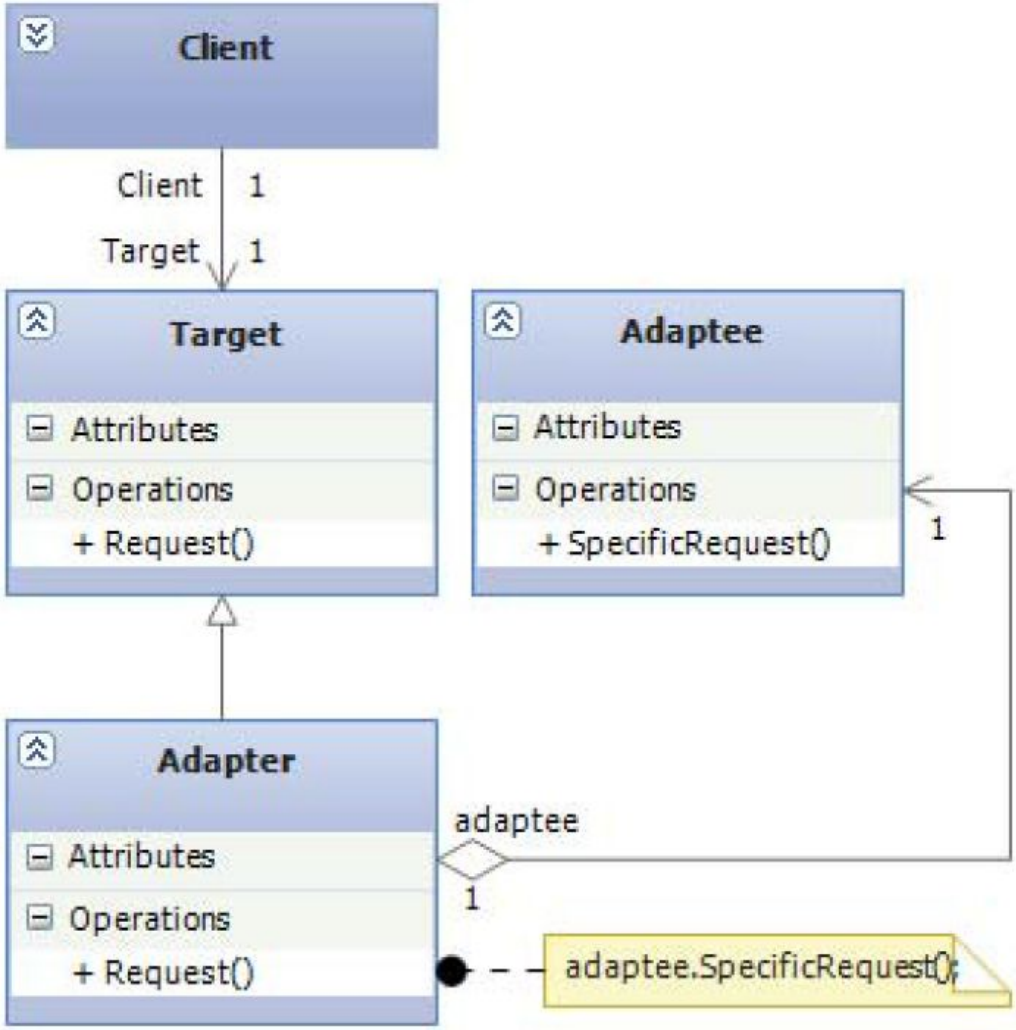
- Приведение интерфейса выполняется за счёт того, что класс Adapter наследует оба класса Adaptee и Target
  - значит, обладает интерфейсами обоих этих классов.

- Затем класс Adapter приводит вызовы методов специфичных для интерфейса класса Target к вызовам соответствующих методов интерфейса класса Apadtee.

Диаграмма последовательности

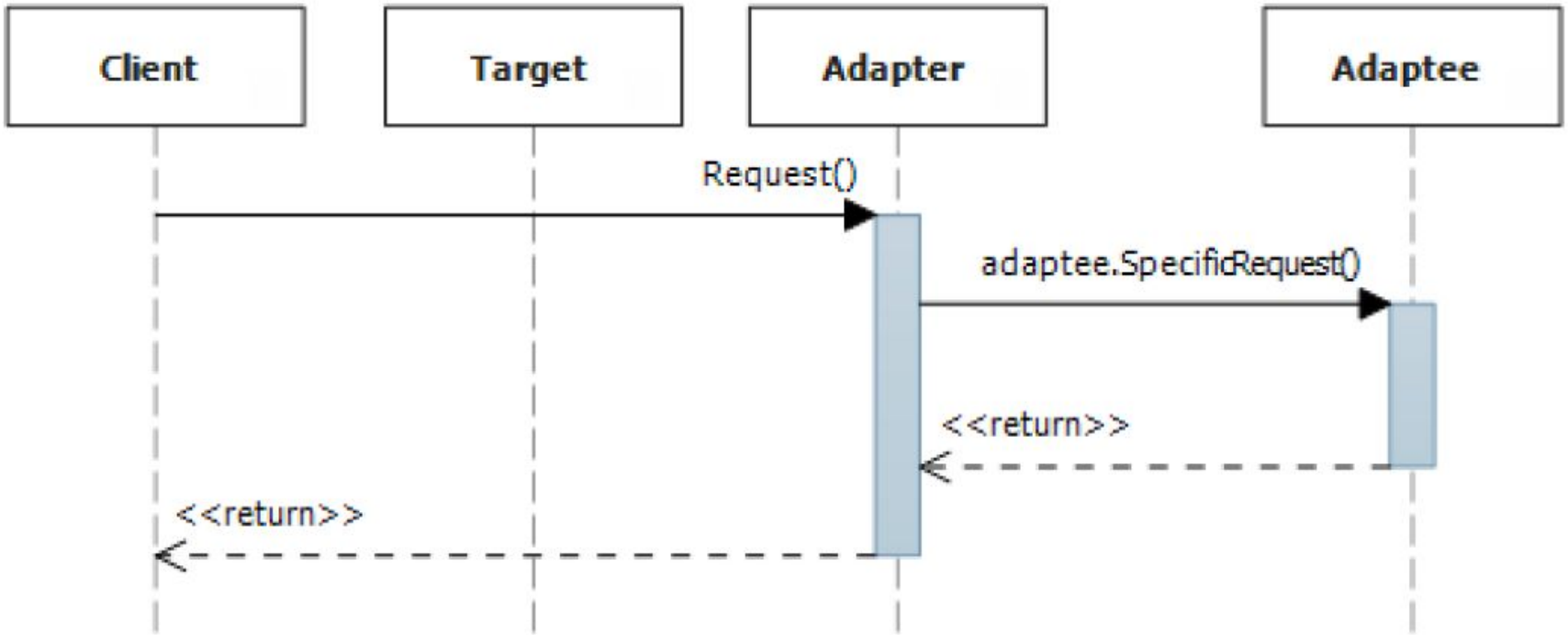


Альтернативный способ



Отличие от предыдущей структуры

- Классы Adapter и Adaptee находятся не в отношении родства, а в отношении ассоциации, то есть класс Adapter агрегирует класс Adaptee.
- Приведение интерфейса класса Adaptee к интерфейсу класса Target выполняется следующим образом:
  - вызовы методов объекта класса Adapter, специфичные для интерфейса класса Target
  - приводятся к вызовам соответствующий методов объекта класса Adaptee, инкапсулированного в классе Adapter.


Client
Target
Adapter
Adaptee
Request()
adaptee.SpecificRequest()
<<return>>
<<return>>

Основной результат использования паттерна Adapter

- Позволяет гибко преобразовать интерфейс некоторого класса к интерфейсу, ожидаемому приложением без изменения структуры самого класса.
  - для устранения избыточности структуры типов,
  - повышения модульности создаваемых приложений.
- Избыточность играет отрицательную роль тогда, когда необходимо повторно использовать написанный нами код (например, в другом приложении).
  - такой код называется reusable-кодом, создание reusable-кода считается хорошей практикой, поскольку уменьшает стоимость и увеличивает скорость разработки

Основной результат использования паттерна Adapter (2)

- Увеличивает гибкость и масштабируемость создаваемых приложений за счёт модульной структуры готового приложения.
- Расширять приложение путём добавления новых типов значительно проще, чем полностью заново создавать некоторые модули.

Пример использования паттерна

- Есть приложение, выполняющее управление товарооборотом предприятия;
- Используется тип данных Product (продукт/товар), который будет организован в коллекцию товаров при помощи класса ProductsCollection;
- мы решили сделать так, чтобы можно было все данные экспортировать в отдельные xml-файлы.
- Для этого мы реализовали класс XmlIO, которые осуществляет запись и чтение Xml-документа, и объект которого инкапсулируется оконным классом нашего приложения.

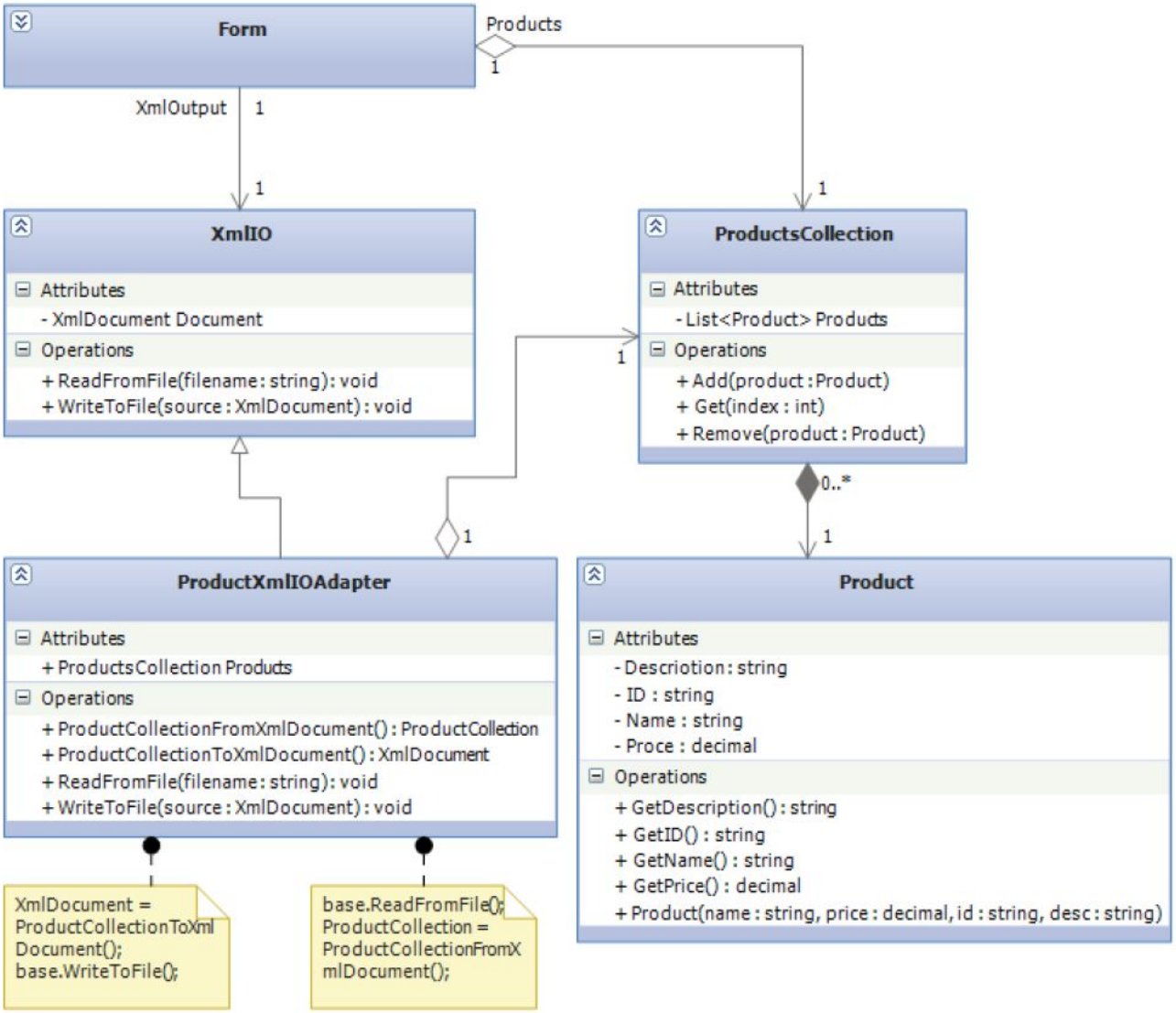
Класс ProductXmlIOAdapter,

- Для записи в файл коллекции продуктов создается класс ProductXmlIOAdapter
  - инкапсулирует коллекцию продуктов,
  - выполняет приведение этой коллекции к XML-документу,
  - выполняет приведение XML-документа к коллекции.
- При этом
  - не меняется структура исходного типа данных и
  - устраняется избыточность структуры, которая могла появиться вследствие наполнения класса Product дополнительным функционалом.

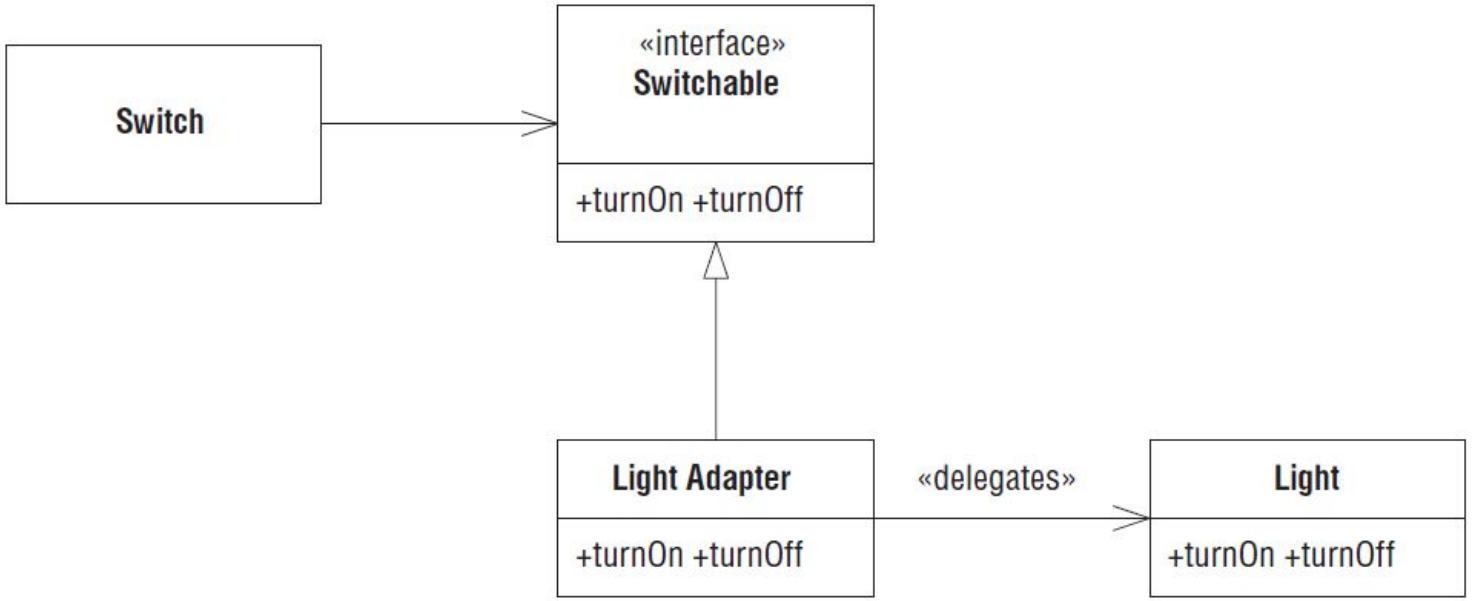
Устранение избыточности

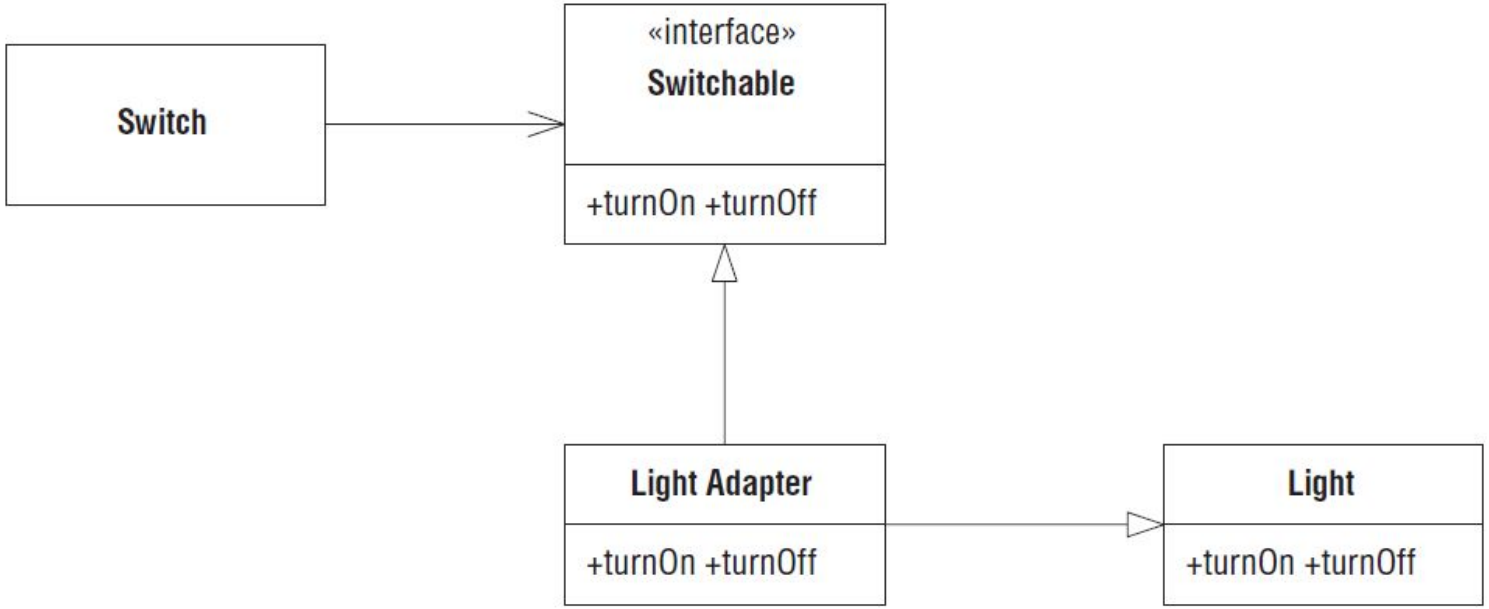
- Устранение избыточности необходимо постольку, поскольку класс Product может использоваться при приведении информации о продуктах, полученной из некоторой базы данных, к объектному представлению, для выполнения последующего анализа и так далее.
- При выполнении всех этих действий избыточность структуры типа может создавать дополнительные сложности.
- Также, подобная модульная структура добавляет гибкости при сопровождении проекта и использовании (создании) reusable-

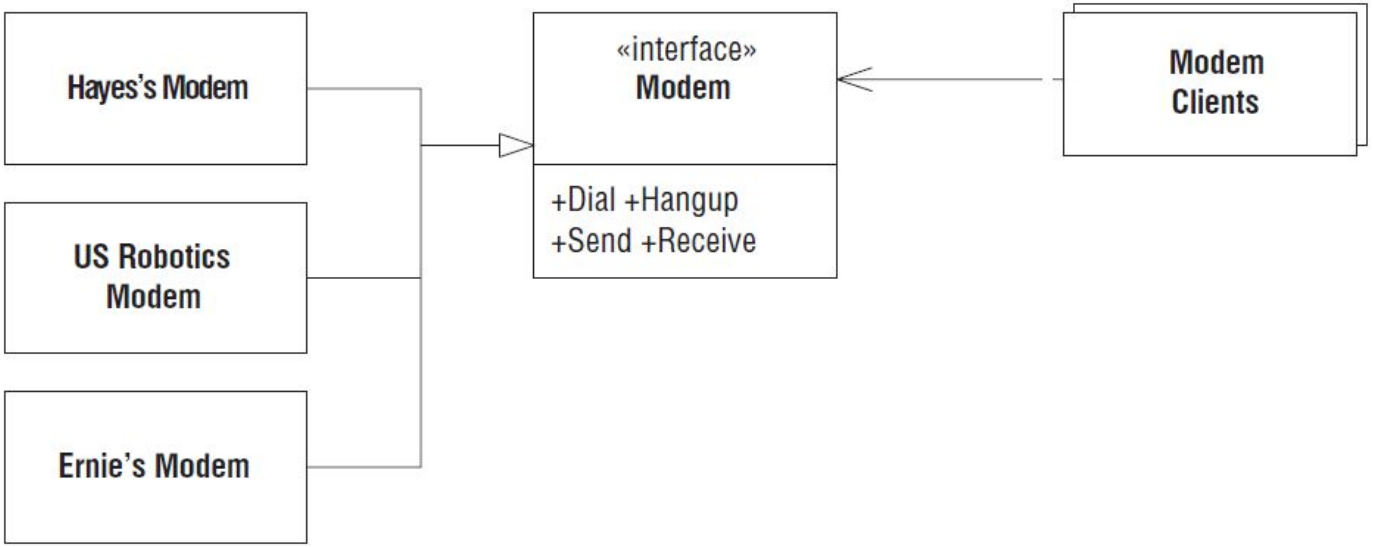
Диаграмма классов, иллюстрирующая приложение



Прототип Адаптер


Switch
«interface»
Switchable
+turnOn +turnOff
Light Adapter
+turnOn +turnOff
«delegates»
Light
+turnOn +turnOff


Switch
«interface»
Switchable
+turnOn +turnOff
Light Adapter
+turnOn +turnOff
Light
+turnOn +turnOff

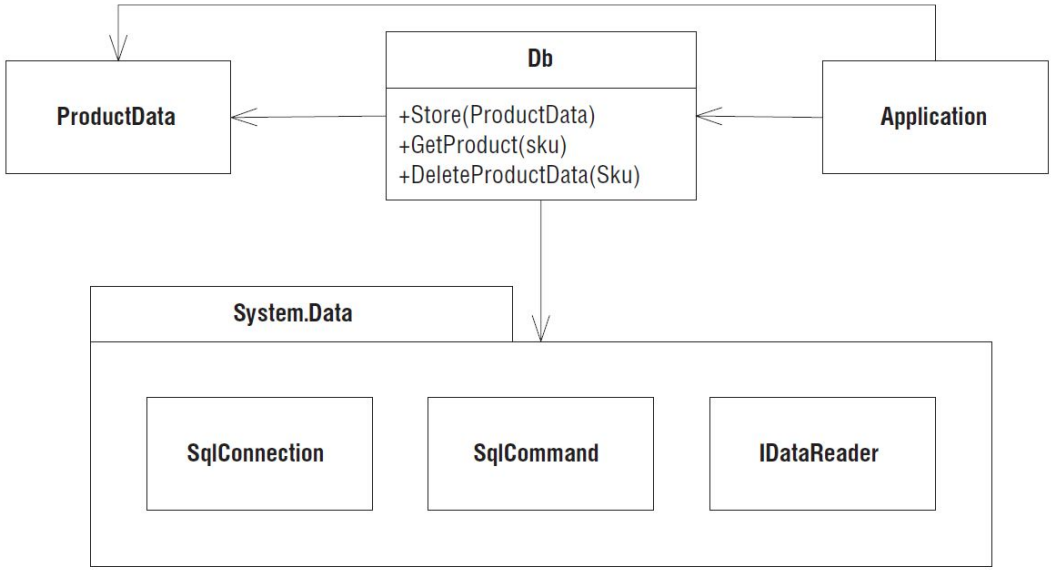

Hayes's Modem
US Robotics Modem
Ernie's Modem
«interface»
Modem
+Dial +Hangup
+Send +Receive
Modem Clients

Паттерн Façade (Фасад)

- Паттерн Façade (Фасад) применяется, когда нужно предоставить простой специализированный интерфейс к группе объектов, имеющих сложный общий интерфейс.
- Пример:
  - есть интерфейсы классов из пространства имен System.Data (ADO.Net).
  - Нужно сделать простой интерфейс, специфичный для данных ProductData.

Класс Db

- Накладывает очень простой интерфейс, специфичный для ProductData, на сложные общие интерфейсы классов из пространства имен System.Data.
  - избавляет Application от необходимости вникать в тонкости пространства имен System.Data.
  - скрывает общность и сложность System.Data за простым специализированным интерфейсом



- Класс DB, являющийся частным случаем Фасада – определяет политику использования System.Data.
- Класс DB описывает:
  - как открыть и закрыть соединение с базой данных,
  - как установить соответствие между переменными-членами ProductData и полями базы данных,
  - как строить запросы для манипулирования данными.

- С точки зрения Application пространства имен System.Data вообще не существует, оно скрыто за Фасадом.
- Использование паттерна Фасад подразумевает следующее:
  - разработчики согласны с тем, что все обращения к базе данных должны производиться только через класс DB.

Паттерн Mediator (Посредник)

- Например, класс QuickEntryMediator находится за сценой и привязывает текстовое поле ввода к списку.
- Когда вы вводите текст в поле, первый элемент списка, начинающийся с введенной строки, подсвечивается.
- Это позволяет набирать только начало текста и затем производить быстрый выбор из списка.

Класс QuickEntryMediator

- Принимает объекты TextBox и ListBox.
- Предполагается, что пользователь будет вводить в TextBox префиксы строк, находящихся в ListBox
- Класс автоматически выбирает первый элемент ListBox, который начинается с префикса, введенного в TextBox.
- Если значение в поле TextBox равно null или префикс не соответствует никакому элементу ListBox, то выделение в ListBox снимается.
- В этом классе нет открытых методов.
- Нужно просто создать объект класса и QuickEntryMediator и забываете о его существовании.
- Например:

```
TextBox t = new TextBox();
ListBox l = new ListBox();
QuickEntryMediator qem = new QuickEntryMediator(t, l);
```

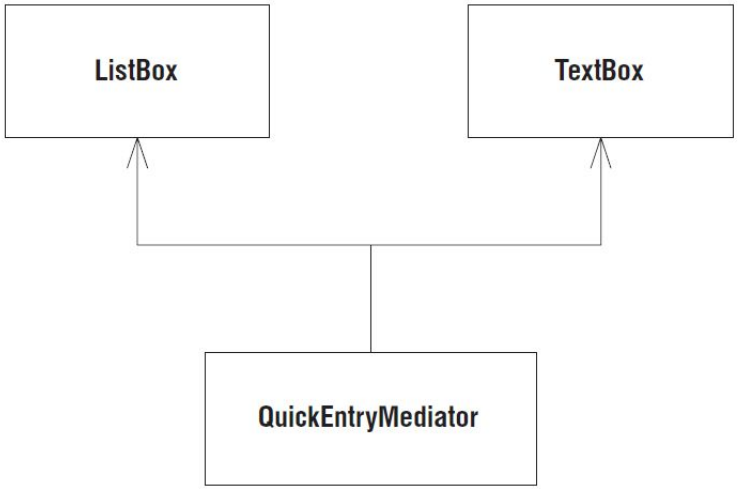
Класс QuickEntryMediator

```
using System;
using System.Windows.Forms;
public class QuickEntryMediator {
  private TextBox itsTextBox;
  private ListBox itsList;
  public QuickEntryMediator(TextBox t,
     ListBox l) {
    itsTextBox = t;
    itsList = l;
     itsTextBox.TextChanged += new
     EventHandler(TextFieldChanged);
  }
```

```
private void TextFieldChanged(object source,  EventArgs
     args) {
   string prefix = itsTextBox.Text;
   if (prefix.Length == 0)
   { itsList.ClearSelected();  return; }
   ListBox.ObjectCollection listItems = itsList.Items;
   bool found = false;
   for (int i = 0; found == false && i < listItems.Count; i++)
      object o = listItems[i];
       string s = o.ToString();
      if (s.StartsWith(prefix)) {
       itsList.SetSelected(i, true);
       found = true;
      } }
  if (!found) { itsList.ClearSelected(); }
} }
```

Структура класса QuickEntryMediator

- Конструктору экземпляра QuickEntryMediator передаются ссылки на ListBox и TextBox.
- QuickEntryMediator назначает обработчик события TextChanged для объекта TextBox.
- при любом изменении текста вызывает метод TextFieldChanged, который ищет в списке ListBox элемент, начинающийся с текущего значения текстового поля, и выделяет его.
- Пользователи классов ListBox и TextField понятия не имеют о существовании этого Посредника.
- Он находится в сторонке и незаметно накладывает свою политику на объекты, не спрашивая у них разрешения и даже не ставя их в известность.



Выводы

- Накладывать политику можно
  - сверху, используя паттерн Фасад, если эта политика должна быть явной.
  - если необходима скрытость, то больше подойдет паттерн Посредник.
- Фасады обычно служат предметом соглашения.
- Все должны быть готовы использовать Фасад вместо скрывающихся за ним объектов.
- Посредник, напротив, скрыт от пользователей.
- Его политика – это свершившийся факт, а не предмет договоренностей.

Заместитель и Шлюз: управление сторонними API

Паттерн **Заместитель**

Паттерн Bridge (мост)

- **Цель** паттерна Bridge («мост») – отделить абстракцию от её реализации, чтобы они могли изменяться независимо друг от друга.

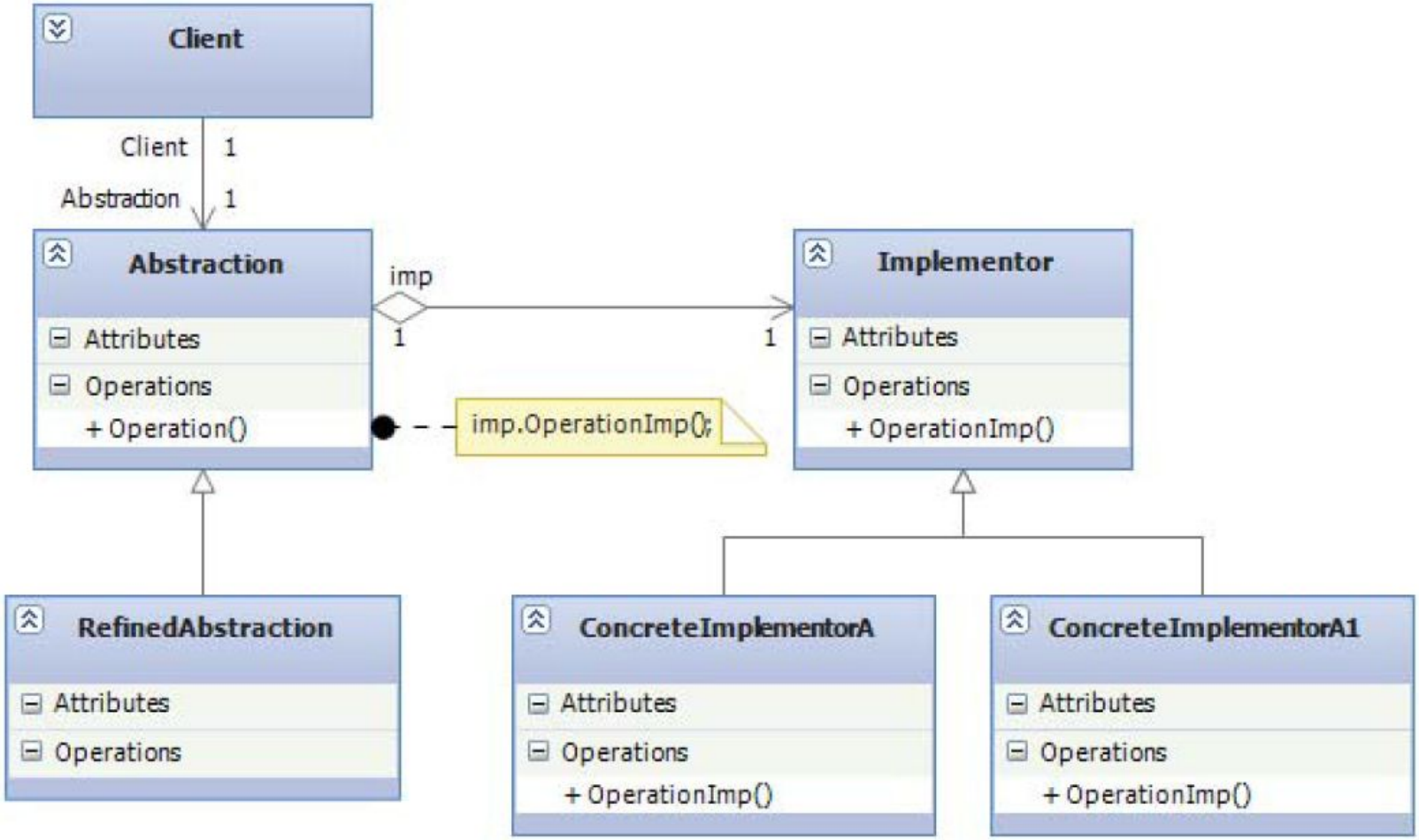
Причины возникновения паттерна

- Обычно, когда некоторая абстракция (обычно абстрактный класс) может иметь несколько конкретных реализаций, используют ***наследование*** для определения множества классов, с похожим (в общем случае говорят одинаковым или совместимым) интерфейсом.
- Абстрактный класс определяет интерфейс для своих потомков, который они реализуют «различными» способами.
- Такой подход является не всегда достаточно гибким и имеет недостатки, способные привести
  - к избыточности кода,
  - создать дополнительные трудности при сопровождении проекта, что значительно увеличит его стоимость.

Причины возникновения паттерна (2)

- Прямое наследование интерфейса *абстракции* некоторым конкретным классом связывает реализацию с абстракцией напрямую
  - создаёт трудности при дальнейшей модификации реализации (её расширении)
  - не позволяет повторно использовать абстракцию и её реализацию отдельно друг от друга.
- Реализация, как бы, становиться «жёстко связанной» с абстракцией.
- Паттерн проектирования мост предполагает помещение интерфейса и его реализации в различные иерархии
- Это позволяет
  - отделить интерфейс от реализации и использовать их независимо
  - комбинировать любые варианты реализации с различными уточнёнными вариантами абстракции.

Структура паттерна Bridge



Участники паттерна Bridge

- *Abstraction (абстракция)* - определяет интерфейс абстракции, а также содержит объект исполнителя, который определяет интерфейс реализации.
- *Implementor (исполнитель)* – определяет интерфейс для классов реализации.
  - интерфейс исполнителя не обязательно должен соответствовать интерфейсу абстракции.
  - интерфейсы, определённые абстракцией и исполнителем, могут быть совершенно разными, что является достаточно гибким.
  - В целом, исполнитель должен определять базовые операции, на которых впоследствии базируется высокоуровневая логика абстракции.
- *RefinedAbstraction (уточнённая абстракция)* - расширяет интерфейс определённый абстракцией.
- *ConcreteImplementor (конкретизированный исполнитель)* - класс, который реализует интерфейс исполнителя и определяет его частную реализацию.
- Абстракция и исполнитель совместно образуют «мост», который связывает уточнённую абстракцию с конкретной

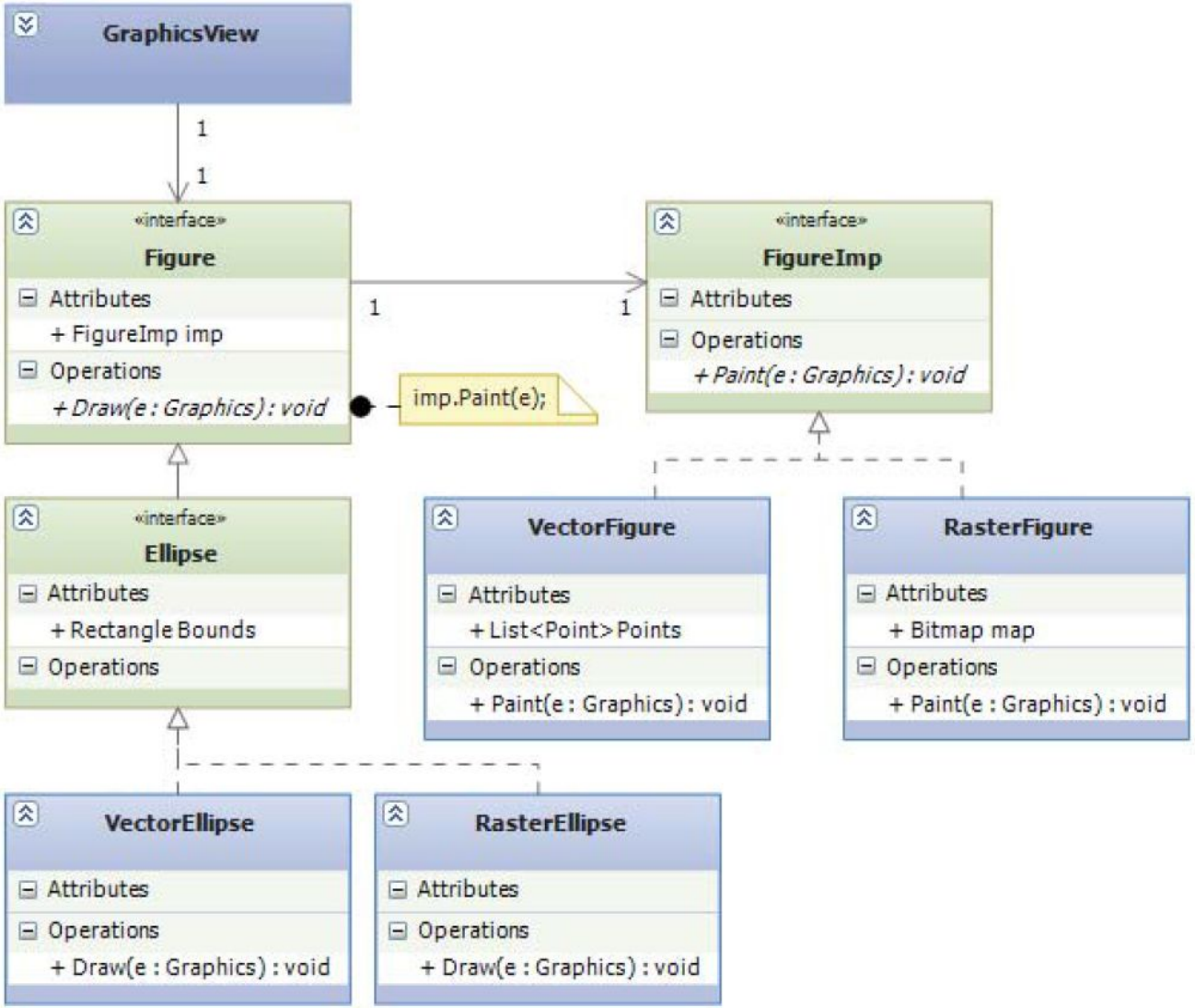
Преимущество использование паттерна Bridge (Мост)

- Выполняется логическое и структурное разделение абстракции от её реализации, что делает код более гибким.
- Улучшает расширяемость кода, т.к. абстракция и исполнитель находятся в различных иерархических структурах, а значит становиться возможным расширение реализации независимо от абстракции, и наоборот.
- Позволяет скрывать детали реализации от клиента (приложения, которое использует абстракцию), что делает клиент независимым от используемой реализации.

Пример использования паттерна Bridge

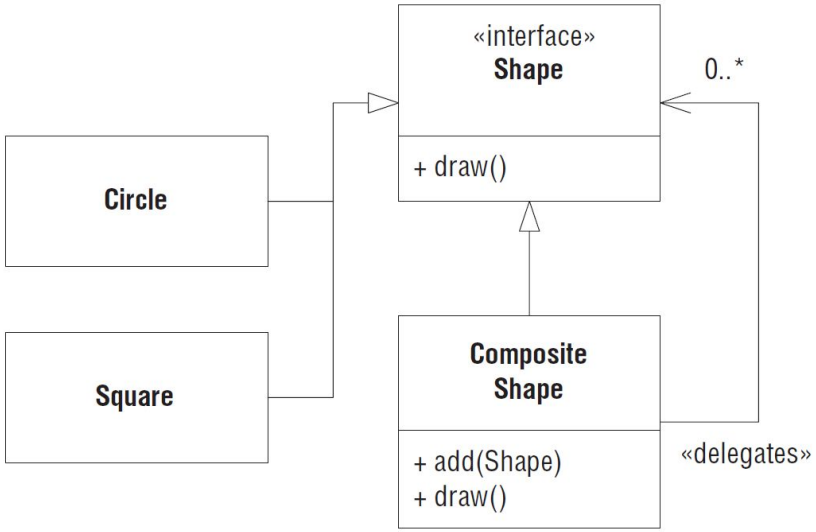
- Смешанный графический редактор – позволяет совместно, в рамках одного представления, редактировать растровую и векторную графику.
- Приложение работает с некоторыми абстрактными фигурами, использующими интерфейсом Figure.
- Интерфейс исполнителя определяется интерфейсом FigureImp, от которого мы наследуем классы VectorFigure и RasterFigure - соответственно, описывающих реализации прорисовки векторной и растровой фигур.
- После определения реализации мы можем уточнить абстракцию, описав интерфейса некоторой конкретной фигуры (например, эллипса).
- После этого можно связать уточнённую абстракцию с конкретной реализацией, например, определив классы VectorEllipse и RasterEllipse, описывающие соответственно векторный и растровый эллипсы.

Модель описанного выше приложения



Паттерн Composite (Компоновщик)

- Паттерн Composite (Компоновщик) – очень простой паттерн, имеющий широкое применение.
- Например, есть иерархия классов геометрических фигур.
  - У базового класса Shape есть два подкласса: Circle и Square.
  - Третьим подклассом является компоновщик.



Пример паттерна Composite (Компоновщик)

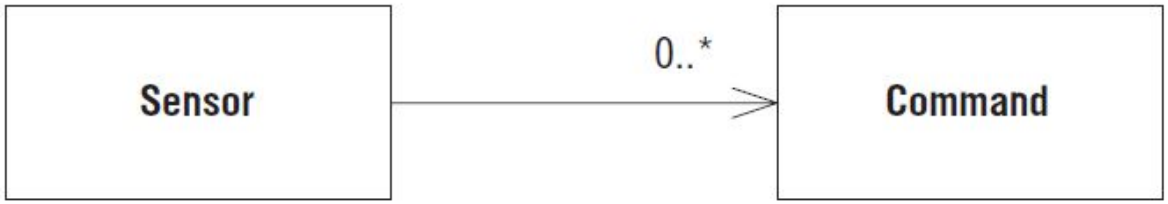
- В классе CompositeShape хранится список объектов типа Shape.
  - метод Draw() в этом классе последовательно вызывает метод Draw() каждого объекта в списке.
- Экземпляр CompositeShape выглядит для системы как один объект Shape.
  - Его можно передать любому методу, принимающему Shape, и он будет вести себя, как Shape.
- Однако это заместитель группы объектов Shape.

Реализация класса CompositeShape

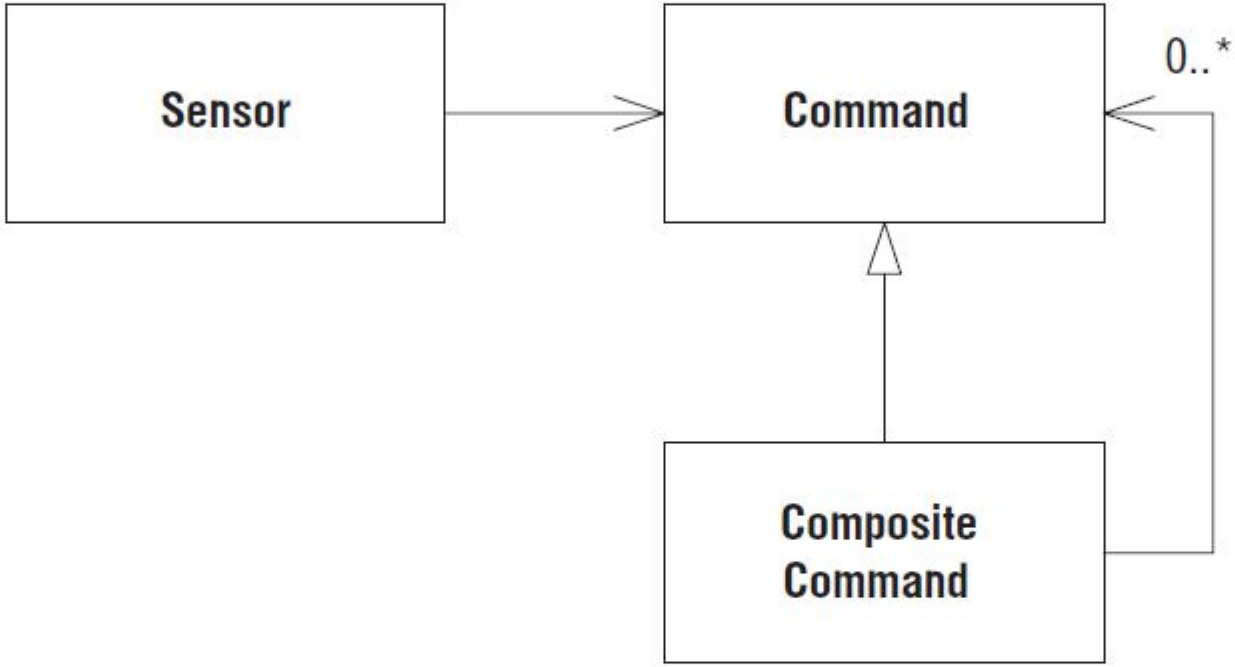
```
public interface Shape {
  void Draw();
}
using System.Collections;
public class CompositeShape : Shape {
  private ArrayList itsShapes = new ArrayList();
  public void Add(Shape s) {
    itsShapes.Add(s);
  }
  public void Draw() {
     foreach (Shape shape in itsShapes) shape.Draw();
  }
}
```

Составные команды

- Ранее рассматривались объекты Sensor и Command



- Обнаружив событие, объект-датчик Sensor вызывал метод Do() ассоциированного с ним объекта Command.
- Часто Sensor должен выполнять несколько команд.
- Например,
  - дойдя до определенного места на тракте подачи, лист бумаги закрывает оптический датчик.
  - в этот момент датчик останавливает один двигатель, запускает другой и включает определенную муфту.


Sensor
Command
0..*
Composite
Command