Шаблоны (паттерны) проектирования

Содержание

1. Шаблоны и Каркасы
2. Классификация шаблонов проектирования
3. Порождающие шаблоны проектирования
4. Структурные шаблоны проектирования
5. Шаблоны поведения проектирования
6. Анти-паттерны
7. Общая классификация паттернов по категориям их применения
8. Принципы хорошего проектирования SOLID.

При разработке систем постоянно возникают вопросы, которые являются определяющими при моделировании:

- как распределить обязанности между классами и объектами?
- как должны взаимодействовать объекты?
- какие функции выполняют конкретные классы?

Некоторые проверенные временем решения таких вопросов, возникающих во время разработки, были сформулированы в виде шаблонов (паттернов) проектирования именованных формул решения проблем, позволяющих систематизировать процесс разработки.

Шаблоны проектирования позволяют не решать каждую новую задачу с нуля, а воспользоваться предыдущими удачными решениями

- *Образец*, или *паттерн* (Pattern), - это типичное решение типичной проблемы в данном контексте.
- *Механизм* (Mechanism) - это образец проектирования, применимый к сообществу классов.
- *Каркас* (Framework) - это архитектурный образец, предлагающий расширяемый шаблон для приложений в одной конкретной области.

- Любая хорошо структурированная система изобилует образцами на разных уровнях абстракции.
- Образцы проектирования описывают
  - структуру и поведение сообщества классов,
- Архитектурные образцы
  - структуру и поведение системы в целом.
- Образцы входят в UML просто потому, что являются важной составляющей словаря разработчика. Явно выделение образцов в системе делает ее более понятной и простой для развития и сопровождения.

Хорошо структурированный образец обладает следующими свойствами:

- решает типичную проблему типичным способом;
- включает структурную и поведенческую составляющие;
- раскрывает элементы управления и стыковки, с помощью которых его можно настроить на разные контексты;.
- является атомарным, то есть не разбивается на меньшие образцы;
- охватывает разные индивидуальные абстракции в системе.

- По словам Кристофера Александра, «любой паттерн описывает задачу, которая снова и снова возникает в нашей работе, а также принцип ее решения, причем таким образом, что это решение можно потом использовать миллион раз, ничего не изобретая заново».
- Хотя имелись в виду паттерны, возникающие при проектировании зданий и городов, но его слова верны и в отношении паттернов объектно-ориентированного проектирования.

Паттерн (образец, шаблон)

Под паттернами проектирования понимается описание взаимодействия объектов и классов, адаптированных для решения общей задачи проектирования в конкретном контексте.

Паттерн проектирования именует, абстрагирует и идентифицирует ключевые аспекты структуры общего решения. Он вычленяет участвующие классы и экземпляры, их роль и отношения, а также функции.

При описании каждого паттерна внимание акцентируется на конкретной задаче объектно-ориентированного проектирования. Анализируется,- когда следует применять паттерн, можно ли его использовать с учетом других проектных ограничений, каковы будут последствия применения метода.

Определение паттерна или шаблона проектирования

- *Шаблон проектирования* – это формализованное описание определенного функционального аспекта объектов реального мира, выполненное, чаще всего, с использованием нотации языка моделирования предметной области UML и представляющее собой удачное решение задачи проектирования этого аспекта в терминах объектно-ориентированной парадигмы.

Основные элементы паттерна

- Имя
- Задача
- Решение
- Результат

Наименование или Имя

Сославшись на него, мы можем сразу описать проблему проектирования; ее решения и их последствия.

Присваивание паттернам имен позволяет проектировать на более высоком уровне абстракции. С помощью словаря паттернов можно вести обсуждение с коллегами, упоминать паттерны в документации, в тонкостях представлять дизайн системы.

Нахождение хороших имен было одной из самых трудных задач при составлении каталога.

Задача

Описание того, когда следует применять паттерн.

Необходимо сформулировать задачу и ее контекст:

- Может описываться конкретная проблема проектирования, например способ представления алгоритмов в виде объектов.
- Иногда отмечается, какие структуры классов или объектов свидетельствуют о негибком дизайне.
- Также может включаться перечень условий, при выполнении которых имеет смысл применять данный паттерн.

Решение

Описание элементов дизайна, отношений между ними, функций каждого элемента.

Конкретный дизайн или реализация не имеются в виду, поскольку паттерн - это шаблон, применимый в самых разных ситуациях. Просто дается абстрактное описание задачи проектирования и того, как она может быть решена с помощью некоего весьма обобщенного сочетания элементов (в нашем случае классов и объектов).

Результаты

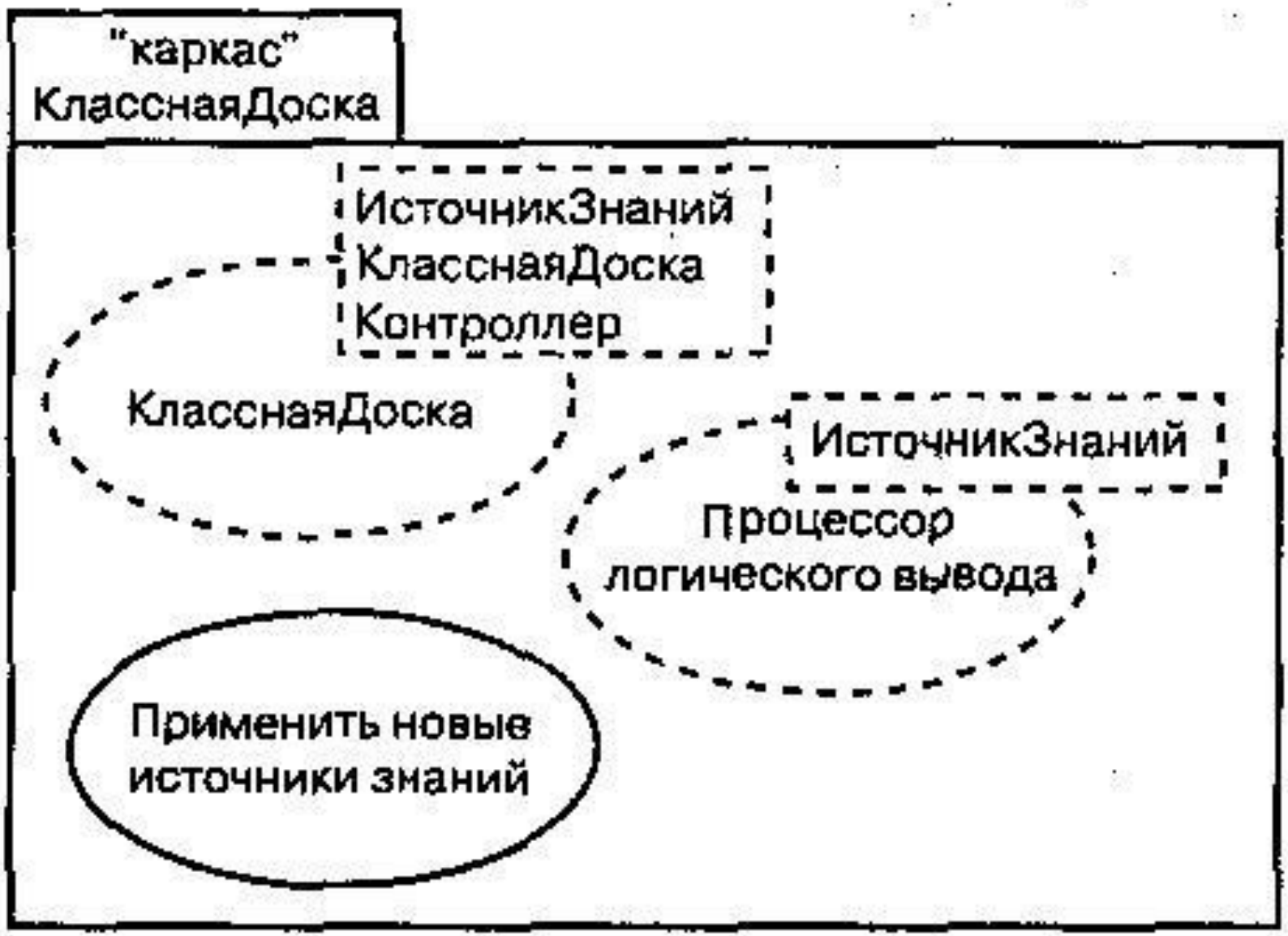
Это следствия применения паттерна и разного рода компромиссы.

Хотя при описании проектных решений о последствиях часто не упоминают, знать о них необходимо, чтобы можно было выбрать между различными вариантами и оценить преимущества и недостатки данного паттерна. Здесь речь идет и о выборе языка и реализации.

Поскольку в объектно-ориентированном проектировании повторное использование зачастую является важным фактором, то к результатам следует относить и влияние на степень гибкости, расширяемости и переносимости системы.

Каркас

- Каркас - это архитектурный образец, предлагающий расширяемый шаблон для приложений в некоторой конкретной области. Выбор такого образца вместо управляемой событиями архитектуры оказывает влияние на всю систему. Этот образец (равно как и его альтернатива) является настолько общим, что имеет смысл назвать его каркасом.
- Каркас - это больше чем механизм. Фактически можно считать, что каркас - это род микроархитектуры, включающий в себя множество механизмов, совместно работающих для разрешения типичной для данной предметной области проблемы. Специфицируя каркас, вы описываете скелет архитектуры со всеми управляющими органами, которые раскрываются пользователям, желающим адаптировать этот каркас для применения в нужном контексте.


"каркас"
КласснаяДоска
ИсточникЗнаний
КласснаяДоска
Контроллер
КласснаяДоска
ИсточникЗнаний
Процессор
логического вывода
Применить новые
источники знаний

Классификация паттернов

Паттерны проектирования различаются степенью детализации и уровнем абстракции.

Рассмотрим классификацию паттернов по двум критериям.

1. Цель - отражает назначение паттерна. В связи с этим выделяются порождающие паттерны, структурные паттерны и паттерны поведения. Первые связаны с процессом создания объектов. Вторые имеют отношение к композиции объектов и классов. Паттерны поведения характеризуют то, как классы или объекты взаимодействуют между собой.

2. Второй критерий - уровень - говорит о том, к чему обычно применяется паттерн: к объектам или классам. Паттерны уровня классов описывают отношения между классами и их подклассами. Такие отношения выражаются с помощью наследования, поэтому они статичны, то есть зафиксированы на этапе компиляции.

Паттерны уровня объектов описывают отношения между объектами, которые могут изменяться во время выполнения и потому более динамичны. Почти все паттерны в какой-то мере используют наследование. Поэтому к категории «паттерны классов» отнесены только те, что сфокусированы лишь на отношениях между классами. Большинство паттернов действуют на уровне объектов.

Таблица 1. Классификация паттернов

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

- Порождающие паттерны классов частично делегируют ответственность за создание объектов своим подклассам, тогда как порождающие паттерны объектов передают ответственность другому объекту.
- Структурные паттерны классов используют наследование для составления классов, в то время как структурные паттерны объектов описывают способы сборки объектов из частей.
- Поведенческие паттерны классов используют наследование для описания алгоритмов и потока управления, а поведенческие паттерны объектов описывают, как объекты, принадлежащие некоторой группе, совместно функционируют и выполняют задачу, которая ни одному отдельному объекту не под силу.

Порождающие паттерны

В свою очередь эту категорию делят еще на два типа

- паттерны порождающие объекты
- паттерны порождающие классы.

Первые создаются с помощью другого объекта, вторые с помощью наследования изменяют класс создаваемого объекта.

Основная идея порождающих паттернов заключается в том, что инстанцирование объектов происходит «за кадром», они скрывают в себе какие именно классы используются в приложении и детали их реализации, оставляя только интерфейсы к ним. В идеале это позволяет собрать полностью рабочее приложение из различных заготовленных заранее объектов, но обычно это практически невозможно без использования других типов шаблонов.

Порождающие паттерны

- Абстрактная фабрика (Abstract Factory) позволяет изменять поведение системы, варьируя создаваемые объекты, при этом сохраняя интерфейсы
- Одиночка (Singleton) Позволяет классу контролировать существование только одного экземпляра и обеспечивает доступ к нему.
- Прототип (Prototype) Позволяет создавать новые объекты за счет клонирования специального объекта - прототипа
- Строитель (Builder) позволяет абстрагировать процесс создания комплексных систем, путем выделения и обобщения классов, отвечающих за создание частей
- Фабричный метод (Factory Method) Определяет интерфейс для разработки объектов, при этом объекты данного класса могут быть созданы его подклассами

Абстрактная фабрика

- **Абстрактная фабрика (англ. *Abstract factory*)** — порождающий шаблон проектирования, позволяющий изменять поведение системы, варьируя создаваемые объекты, при этом сохраняя интерфейсы. Он позволяет создавать целые группы взаимосвязанных объектов, которые, будучи созданными одной фабрикой, реализуют общее поведение. Шаблон реализуется созданием абстрактного класса Factory, который представляет собой интерфейс для создания компонентов системы (например, для оконного интерфейса он может создавать окна и кнопки). Затем пишутся наследующиеся от него классы, реализующие этот интерфейс.

Абстрактная фабрика

Цель

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

Плюсы

- изолирует конкретные классы
- упрощает замену семейств продуктов;
- гарантирует сочетаемость продуктов.

Минусы

- сложно добавить поддержку нового вида продуктов.

Абстрактная фабрика

Применимость

- 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.
- Требуется предоставить библиотеку объектов, раскрывая только их интерфейсы, но не реализацию.

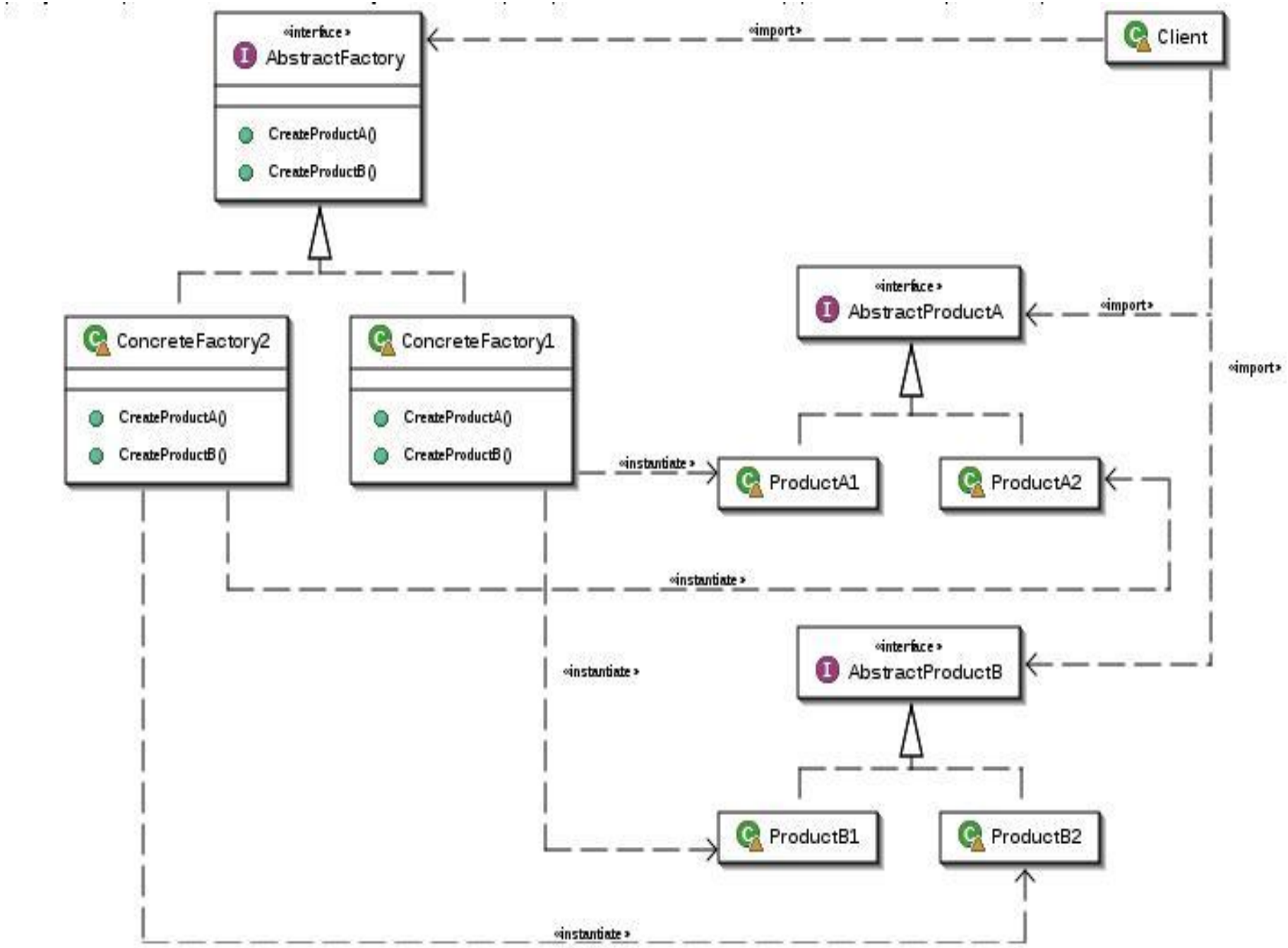
- При рассмотрении порождающих паттернов, обычно используется такое понятие как — объекты-продукты. Фабрика должна иметь операции, которые создают новые объекты, (иначе это не фабрика), и эти самые возвращаемые фабрикой объекты, называют продуктами.

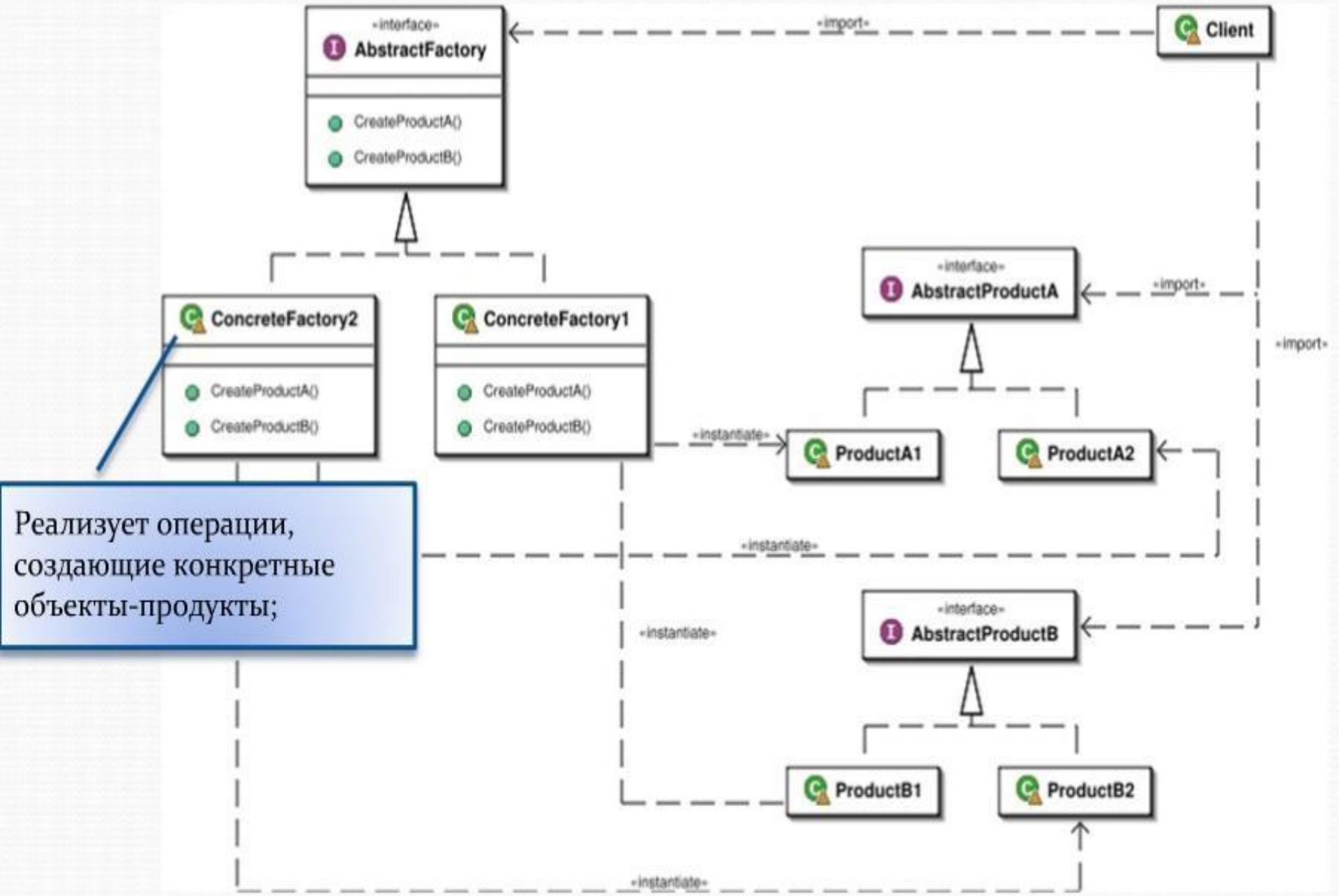
  Сама по себе абстрактная фабрика содержит только интерфейс для операций создающих объекты, работа с этим паттерном заключается в создании конкретных фабрик, которые будут реализовывать эти интерфейсы. Если пойти дальше, то зачастую для достижения большей гибкости, требуется так же писать абстрактные классы и для объектов-продуктов.

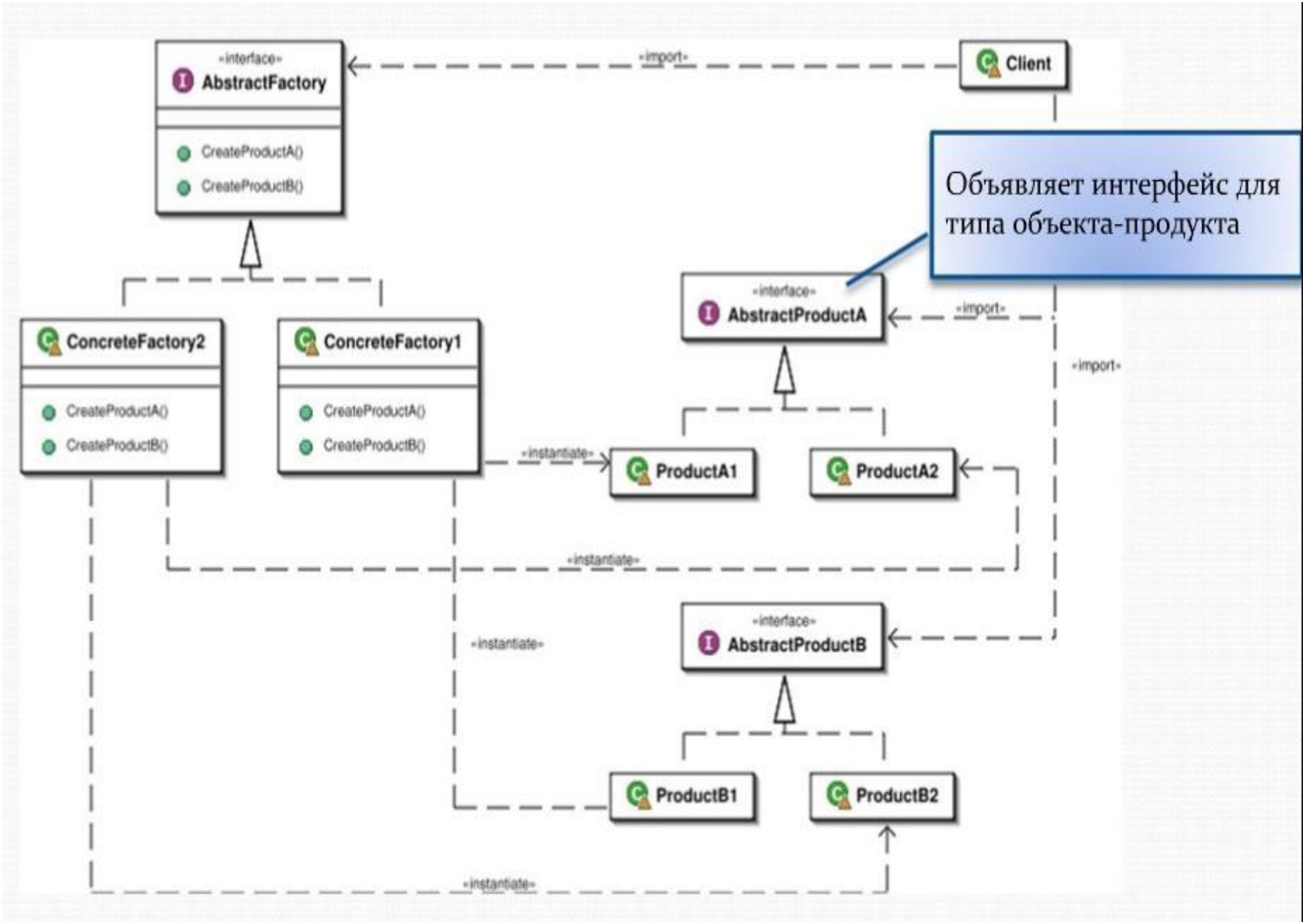
При разработке необходимо создать:

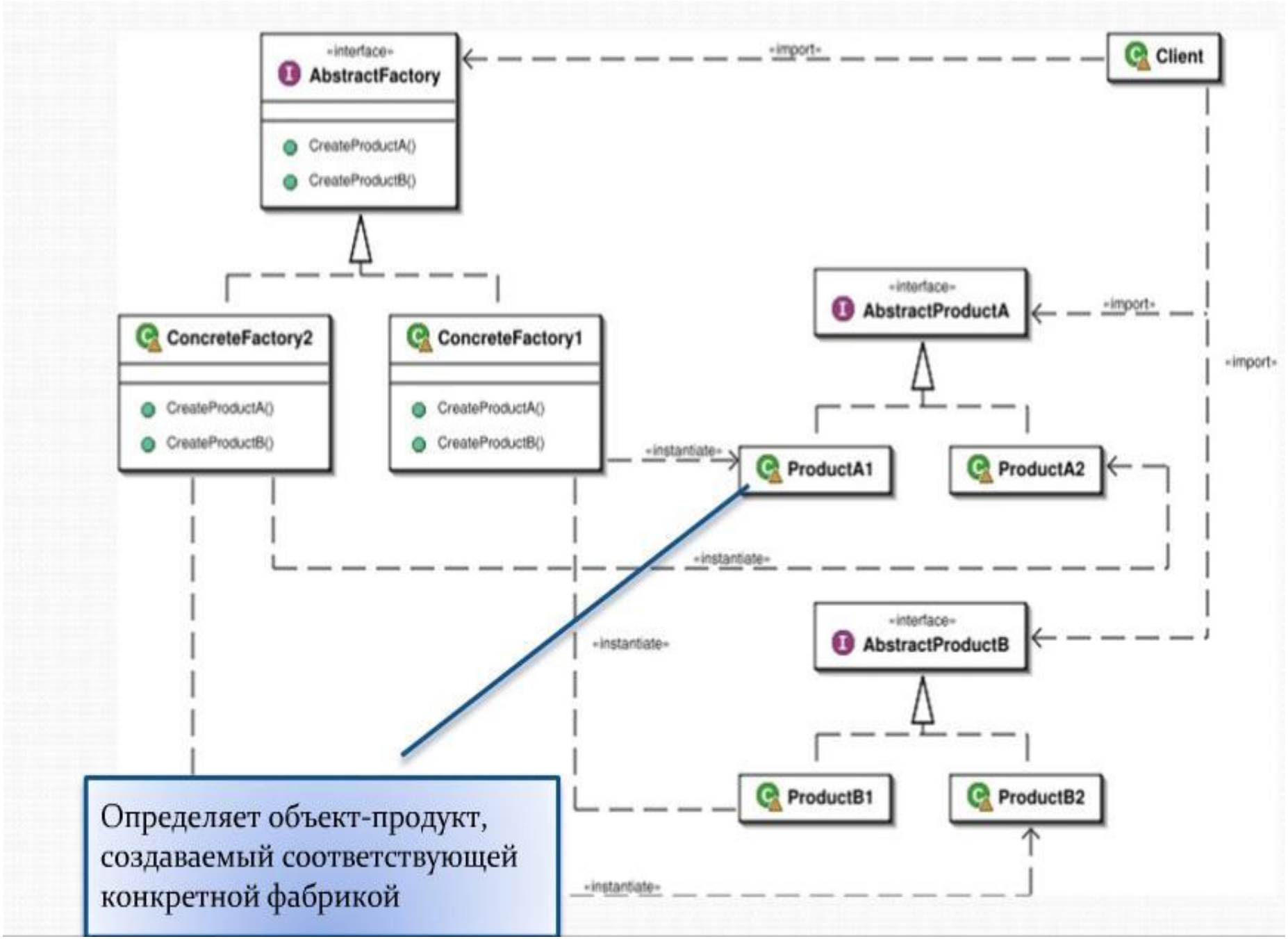
- абстрактный класс — фабрику;
- абстрактные классы — объекты, которые она будет создавать (это может быть и один объект);
- конкретный класс(-ы), который(-ые) будет(-ут) реализовывать методы и/или дополнять их, это будут фабрики производящие семейства объектов;
- создать реальные классы продуктов (в некоторых случаях они могут быть пустыми и наследовать функционал от своих абстрактных родителей);

Первые два пункта относятся к проектированию, следующие уже к реализации. Главное хорошо продумать проектирование, что бы получить должный эффект от любого паттерна.

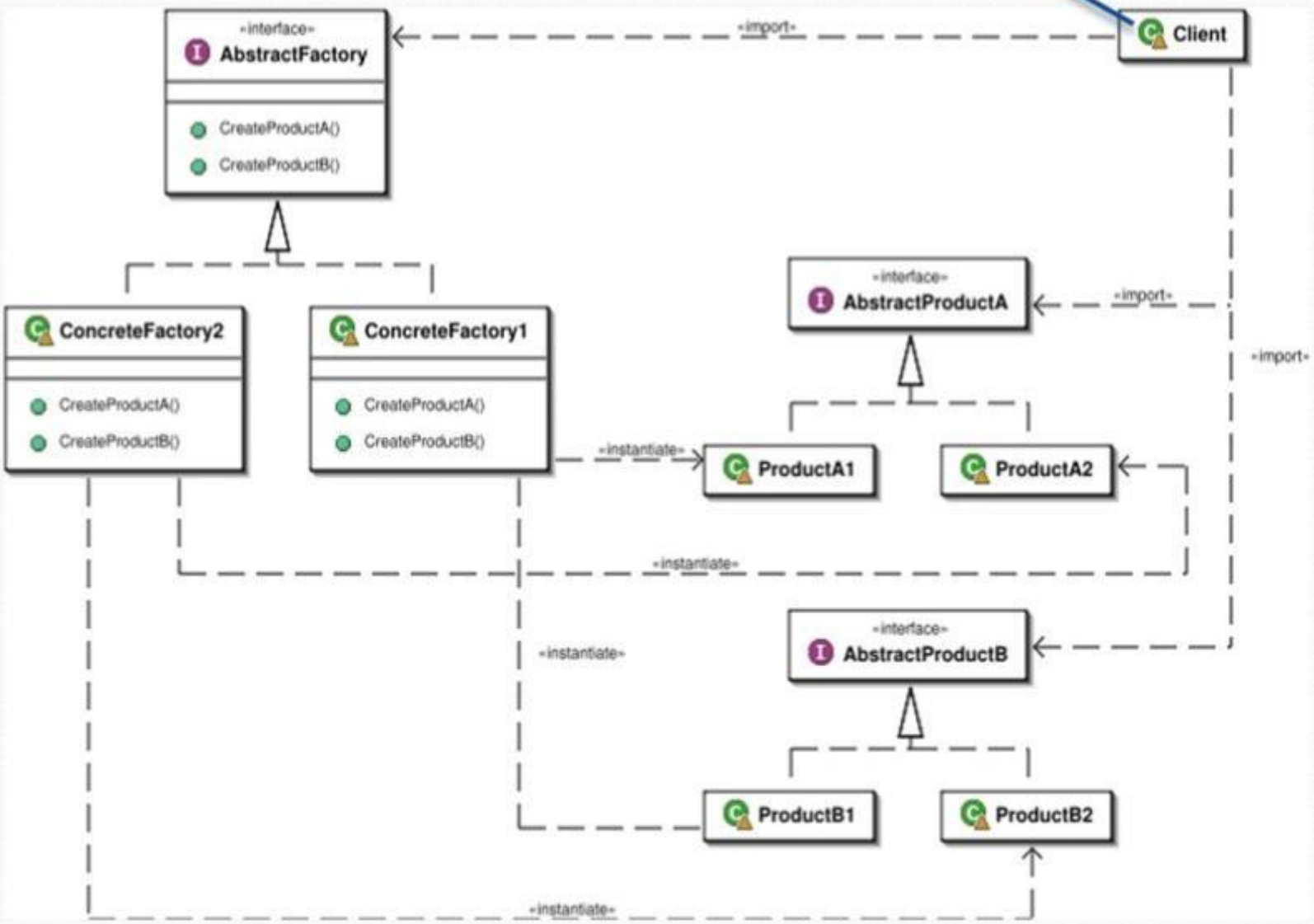

«interface»
AbstractFactory
CreateProductA()
CreateProductB()
«import»
Client
ConcreteFactory2
CreateProductA()
CreateProductB()
ConcreteFactory1
CreateProductA()
CreateProductB()
«interface»
AbstractProductA
«import»
«import»
«instantiate»
ProductA1
ProductA2
«instantiate»
«instantiate»
«interface»
AbstractProductB
ProductB1
ProductB2
«instantiate»


«interface»
AbstractFactory
CreateProductA()
CreateProductB()
«import»
Client
«interface»
AbstractProductA
«import»
«import»
ConcreteFactory2
CreateProductA()
CreateProductB()
ConcreteFactory1
CreateProductA()
CreateProductB()
«instantiate»
ProductA1
ProductA2
Реализует операции, создающие конкретные объекты-продукты;
«instantiate»
«instantiate»
«interface»
AbstractProductB
ProductB1
ProductB2
«instantiate»


«interface»
AbstractFactory
CreateProductA()
CreateProductB()
«import»
Client
Объявляет интерфейс для типа объекта-продукта
«interface»
AbstractProductA
«import»
«import»
ConcreteFactory2
CreateProductA()
CreateProductB()
ConcreteFactory1
CreateProductA()
CreateProductB()
«instantiate»
ProductA1
ProductA2
«instantiate»
«instantiate»
«interface»
AbstractProductB
ProductB1
ProductB2
«instantiate»


«interface»
AbstractFactory
CreateProductA()
CreateProductB()
«import»
Client
«interface»
AbstractProductA
«import»
«import»
ConcreteFactory2
CreateProductA()
CreateProductB()
ConcreteFactory1
CreateProductA()
CreateProductB()
«instantiate»
ProductA1
ProductA2
«instantiate»
«instantiate»
«interface»
AbstractProductB
ProductB1
ProductB2
«instantiate»
Определяет объект-продукт, создаваемый соответствующей конкретной фабрикой

Пользуется исключительно интерфейсами, которые объявлены в классах AbstractFactory и AbstractProduct



```cpp
#include <iostream>

// AbstractProductA
class ICar
{
public:
	virtual void printName() = 0;
};

// ConcreteProductA1
class Ford : public ICar
{
public:
	virtual void printName()
	{
		std::cout << "Ford" << std::endl;
	}
};

// ConcreteProductA2
class Toyota : public ICar
{
public:
	virtual void printName()
	{
		std::cout << "Toyota" << std::endl;
	}
};

// AbstractProductB
class IEngine
{
public:
	virtual void printPower() = 0;
};
```

```cpp
#include <iostream>

// AbstractProductA
class ICar
{
public:
	virtual void printName() = 0;
};

// ConcreteProductA1
class Ford : public ICar
{
public:
	virtual void printName()
	{
		std::cout << "Ford" << std::endl;
	}
};

// ConcreteProductA2
class Toyota : public ICar
{
public:
	virtual void printName()
	{
		std::cout << "Toyota" << std::endl;
	}
};
```

```
// AbstractProductB
class IEngine
{
public:
        virtual void printPower() = 0;
};

// ConcreteProductB1
class FordEngine : public IEngine
{
public:
        virtual void printPower()
        {
                std::cout << "Ford Engine 4.4" << std::endl;
        }
};

// ConcreteProductB2
class ToyotaEngine : public IEngine
{
public:
        virtual void printPower()
        {
                std::cout << "Toyota Engine 3.2" << std::endl;
        }
};
```

```
// AbstractFactory
class ICarFactory
{
public:
        virtual ICar*      createCar()    = 0;
        virtual IEngine*   createEngine() = 0;
};

// ConcreteFactory1
class FordFactory : public ICarFactory
{
public:
        // from ICarFactory
        virtual ICar* createCar()
        {
                return new Ford();
        }

        virtual IEngine* createEngine()
        {
                return new FordEngine();
        }
};

// ConcreteFactory2
class ToyotaFactory : public ICarFactory
{
public:
        // from ICarFactory
        virtual ICar* createCar()
        {
                return new Toyota();
        }

        virtual IEngine* createEngine()
        {
                return new ToyotaEngine();
        }
};
```

```
void use(ICarFactory* f)
{
        ICar*     myCar      =  f->createCar();
        IEngine*  myEngine   =  f->createEngine();

        myCar->printName();
        myEngine->printPower();

        delete myCar;
        delete myEngine;
}


int main()
{
        ToyotaFactory   toyotaFactory;
        FordFactory     fordFactory;

        use (&toyotaFactory);
        use (&fordFactory);

        return 0;
}
```

Структурные паттерны

- Здесь также есть два типа — паттерны уровня класса и паттерны уровня объекта.
- Ярким примером первых является «Адаптер». Общий смысл его в том, что если есть класс и его интерфейсы не совместимы с другими библиотеками в нашей системе, то что бы разрешить этот конфликт, мы не изменяем код этого класса, а пишем для него адаптер. Очень часто этот паттерн применяется при написании библиотек, которые позволяют работать с различными СУБД.
- Паттерны уровня объекта позволяют достичь большей гибкости в приложения во время его выполнения. Наиболее популярный из них - «Декоратор».

Структурные паттерны

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

Паттерн Фасад

Общая цель всякого проектирования - свести к минимуму зависимость подсистем друг от друга и обмен информацией между ними. Один из способов решения этой задачи - введение объекта фасад, предоставляющий единый упрощенный интерфейс к более сложным системным средствам.

Паттерн Фасад предназначен для замены нескольких разнотипных интерфейсов доступа к определенной подсистеме некоторым унифицированным интерфейсом, что существенно упрощает использование рассматриваемой подсистемы.

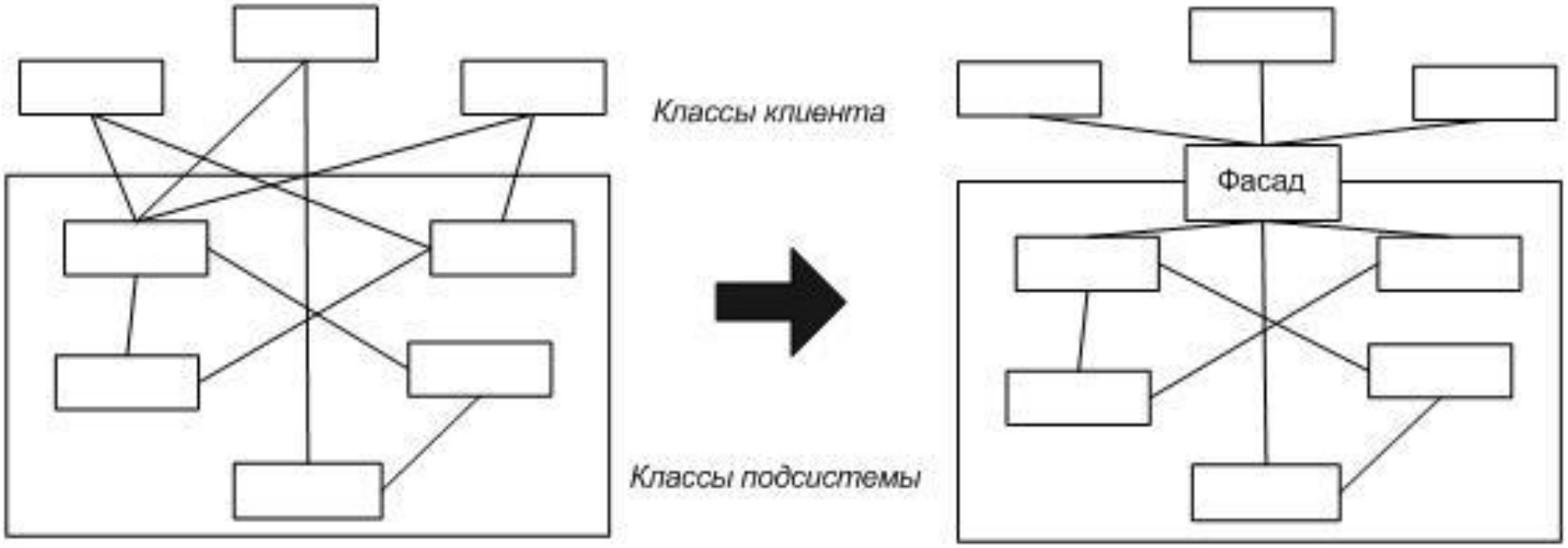
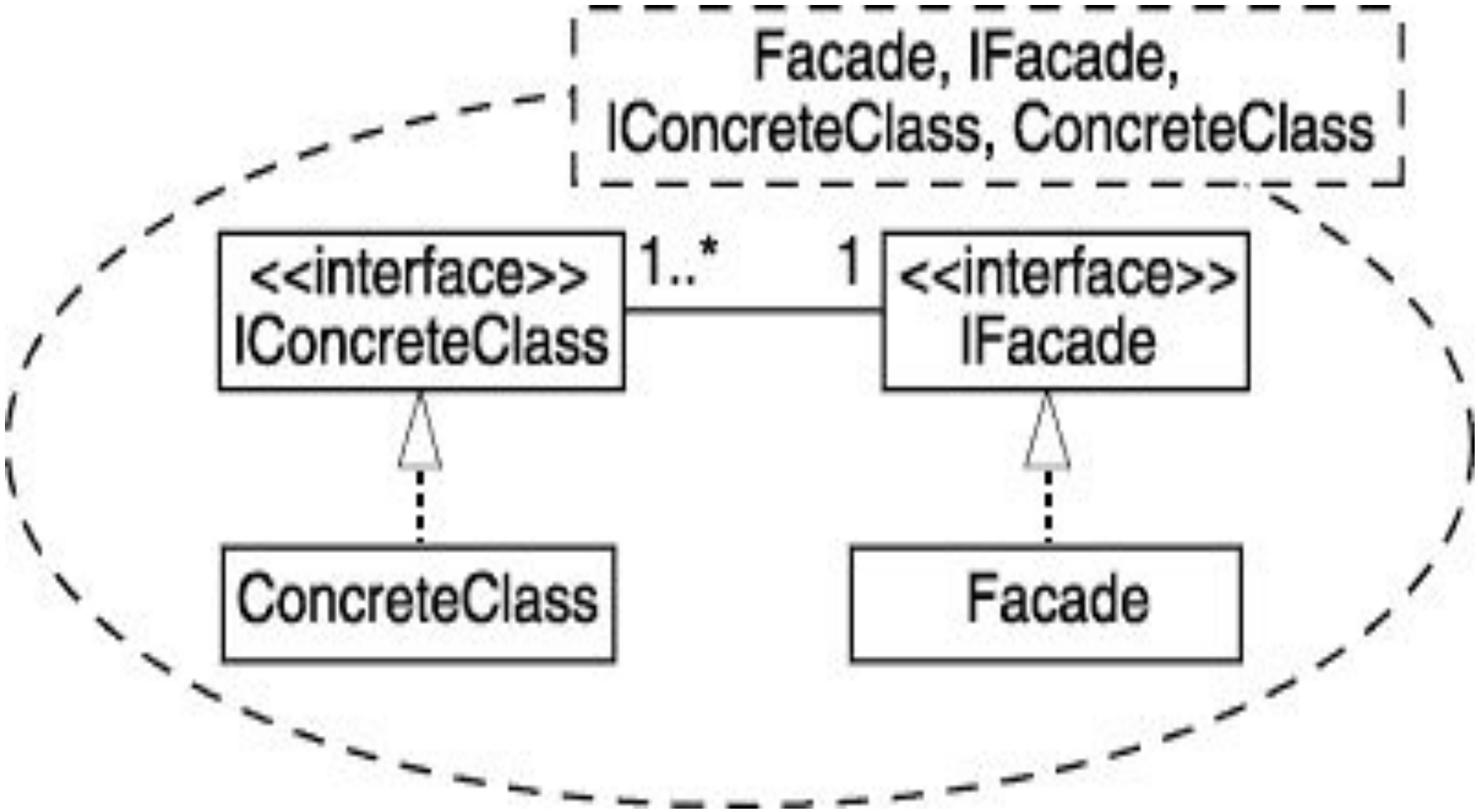

Классы клиента
Фасад
Классы подсистемы

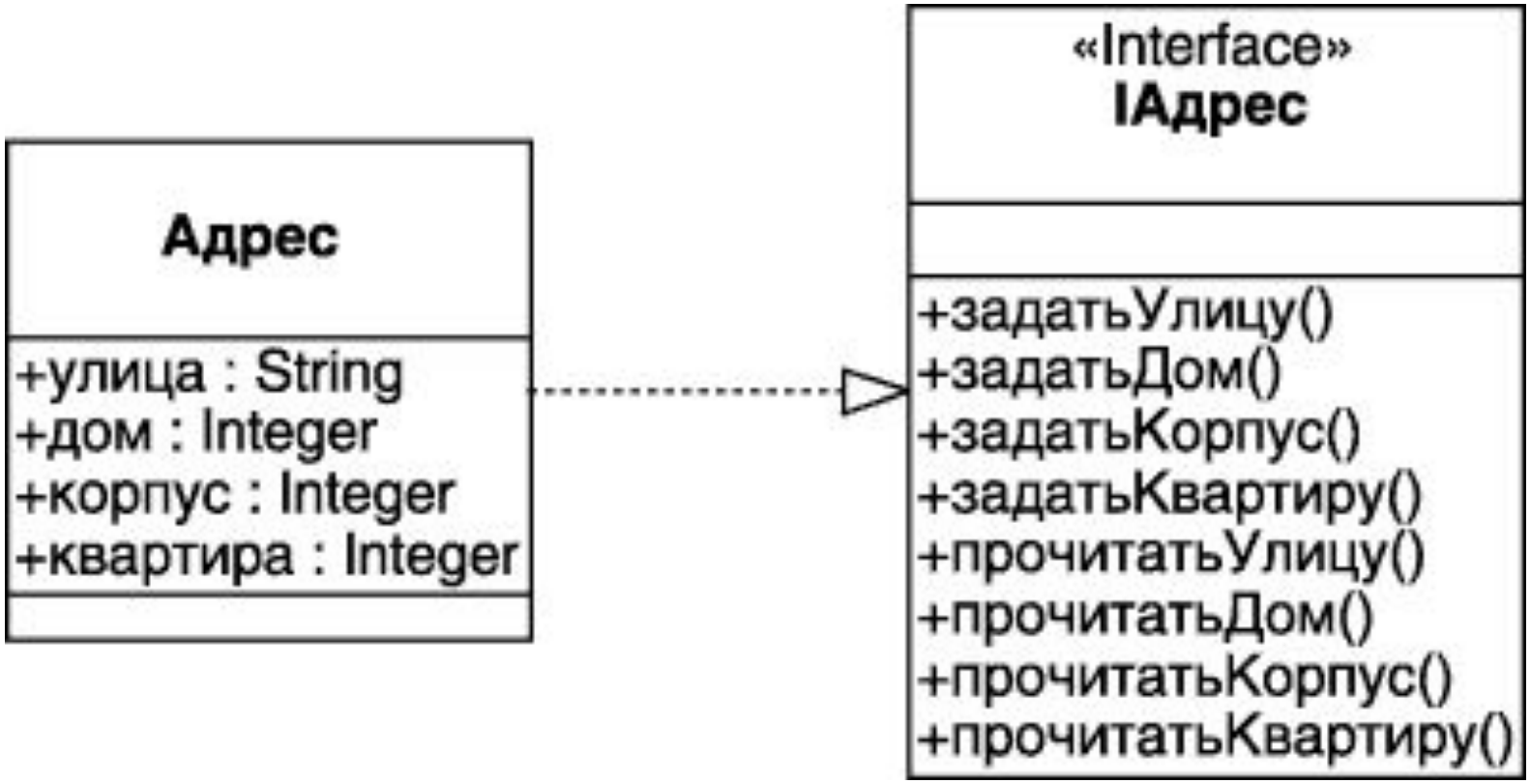
Диаграмма параметризованной кооперации



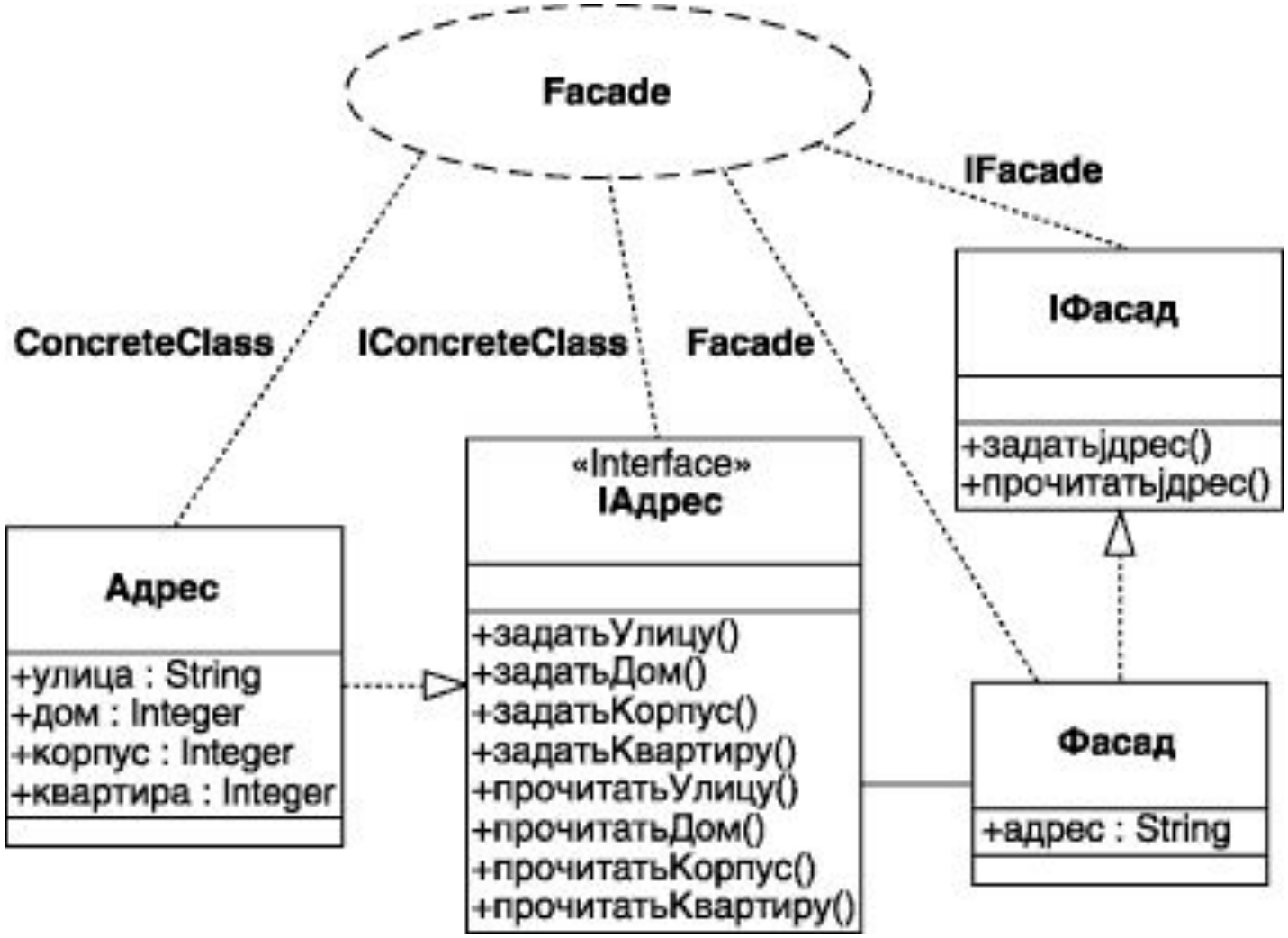
- Изображенная параметризованная кооперация содержит 4 параметра: класс Facade (Фасад), интерфейс IFacade, интерфейсы IConcreteClass и конкретные классы ConcreteClass, в которых реализованы интерфейсы IConcreteClass. Пунктирная линия со стрелкой в форме треугольника служит для обозначения отношения реализации (не путать с отношением обобщения классов).
- При решении конкретных задач проектирования данный паттерн может быть конкретизирован. В этом случае вместо параметров изображенной кооперации должны быть указаны классы, предназначенные для решения отдельных задач.

Пример паттерна Фасад

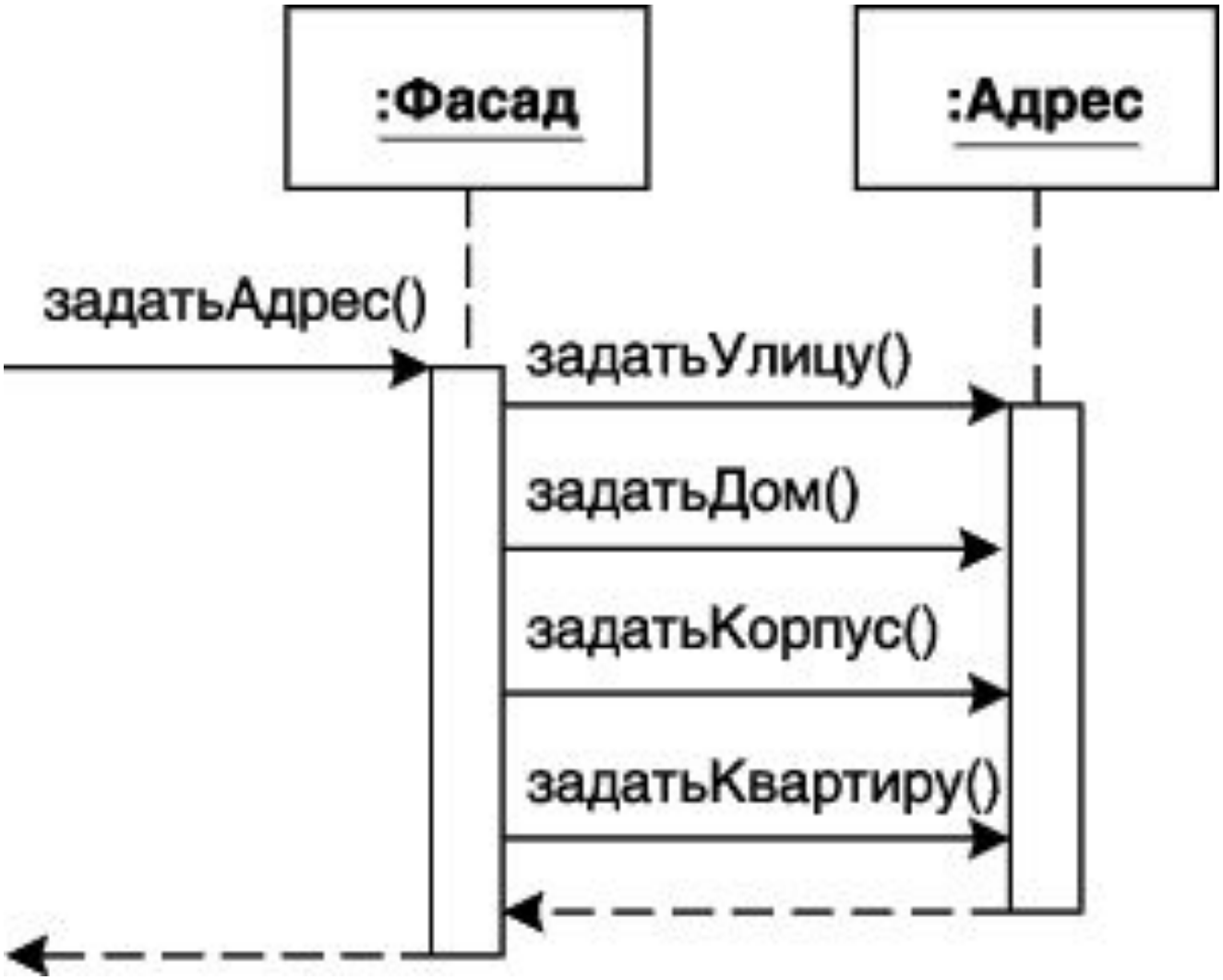
- Пример, который иллюстрирует использование паттерна Фасад для выполнения операций по заданию и считыванию адресов из базы данных сотрудников.
- Фрагмент соответствующей диаграммы классов содержит 2 класса: Адрес и интерфейс к операциям этого класса IАдрес
- При задании адреса нового сотрудника необходимо обратиться к этому интерфейсу и последовательно выполнить операции: задатьУлицу(), задатьДом(), задатьКорпус(), задатьКвартиру(), используя в качестве аргумента идентификационный номер нового сотрудника.
- Для получения информации об адресе сотрудника, необходимо также обратиться к этому интерфейсу и последовательно выполнить операции: прочитатьУлицу(), прочитатьДом(), прочитатьКорпус(), прочитатьКвартиру(), используя в качестве аргумента идентификационный номер интересующего сотрудника.


«Interface»
IАдрес
+задатьУлицу()
+задатьДом()
+задатьКорпус()
+задатьКвартиру()
+прочитатьУлицу()
+прочитатьДом()
+прочитатьКорпус()
+прочитатьКвартиру()
Адрес
+улица : String
+дом : Integer
+корпус : Integer
+квартира : Integer

- Очевидно, отслеживать при каждом обращении правильность выполнения этих последовательностей операций неудобно. С этой целью к данному фрагменту следует добавить еще один интерфейс, реализацию паттерна Фасад для рассматриваемой ситуации. Соответствующий фрагмент модифицированной диаграммы классов будет содержать 4 класса, изображенные таким образом, чтобы иллюстрировать реализацию параметрической кооперации.


Facade
IFacade
ConcreteClass
IConcreteClass
Facade
IФасад
+задатьjдрес()
+прочитатьjдрес()
«Interface»
IАдрес
Адрес
+улица : String
+дом : Integer
+корпус : Integer
+квартира : Integer
+задатьУлицу()
+задатьДом()
+задатьКорпус()
+задатьКвартиру()
+прочитатьУлицу()
+прочитатьДом()
+прочитатьКорпус()
+прочитатьКвартиру()
Фасад
+адрес : String

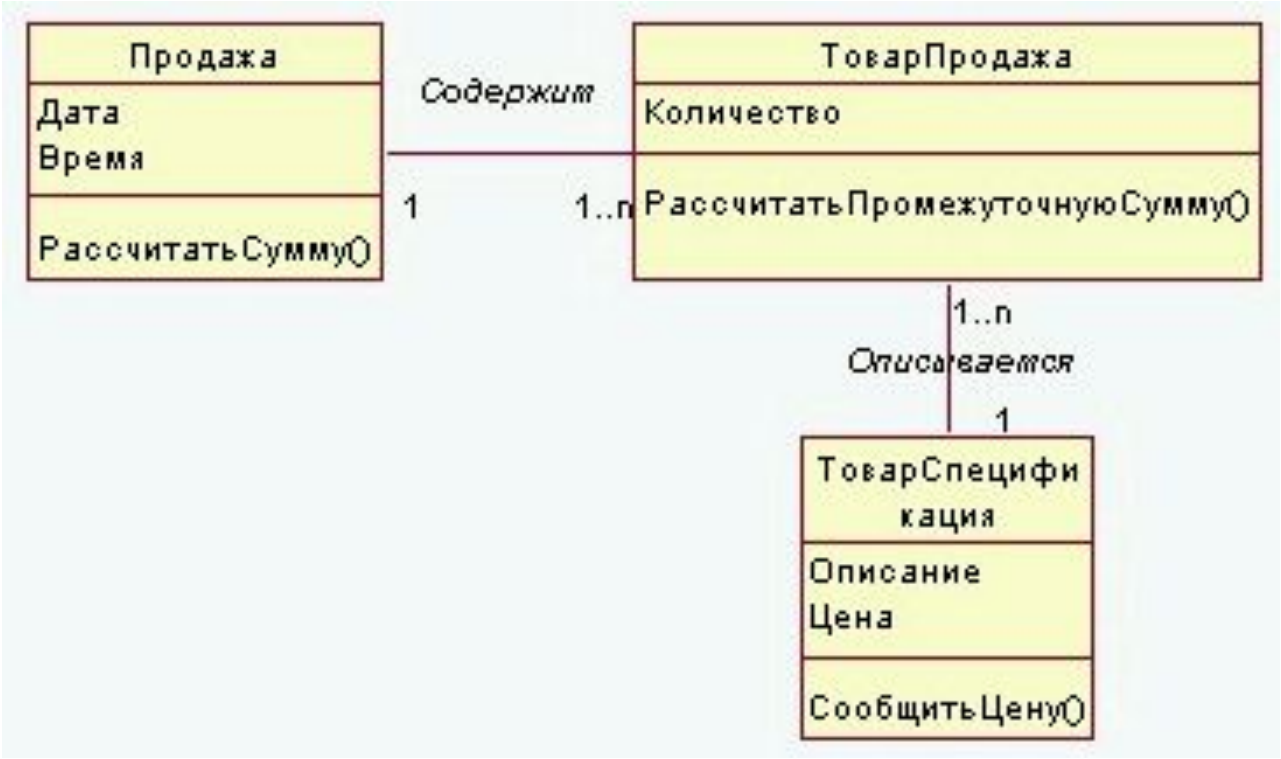
- При задании адреса нового сотрудника в этом случае достаточно обратиться к интерфейсу IФасад и выполнить единственную операцию: задатьАдрес(), используя в качестве аргумента идентификационный номер нового сотрудника. Для получения информации об адресе сотрудника также достаточно обратиться к этому интерфейсу и выполнить единственную операцию: прочитатьАдрес(), используя в качестве аргумента идентификационный номер интересующего сотрудника. Реализацию данных операций следует предусмотреть в классе Фасад. Взаимодействие объектов этих классов может быть представлено с помощью диаграммы последовательности


:Фасад
:Адрес
задатьАдрес()
задатьУлицу()
задатьДом()
задатьКорпус()
задатьКвартиру()

- Аналогичная диаграмма последовательности может быть построена для выполнения операции по чтению адреса.
- Использование паттерна Фасад обеспечивает для клиента не только простоту доступа к информации об адресах, но и независимость представления объектов класса Адрес от запросов клиентов. Это обстоятельство особенно актуально при изменении формата представления информации или смене соответствующей базы данных. В этом случае потребуется внести изменения только в реализацию операций класса Фасад.

Информационный эксперт (Information Expert)- GRASP

- Проблема: В системе должна аккумулироваться, рассчитываться и т. п. необходимая информация
- Решение: Назначить обязанность аккумуляции информации, расчета и т. п. некоему классу (информационному эксперту), обладающему необходимой информацией.
- Информационным экспертом может быть не один класс, а несколько.


Продажа
Дата
Время
РассчитатьСумму()
Содержит
1
1..n
ТоварПродажа
Количество
РассчитатьПромежуточнуюСумму()
1..n
Описывается
1
ТоварСпецификация
Описание
Цена
СообщитьЦену()


1: РассчитатьСумму( )
2: РассчитатьПромежуточнуюСумму( )
: Продажа
: ТоварПродажа
3: СообщитьЦену( )
: ТоварСпецификация

Преимущества и недостатки

- Поддерживает инкапсуляцию, то есть объекты используют свои собственные данные для выполнения поставленных задач.
- Если объект, обладающий наиболее полной информацией, например, о продаже (*см. пример* - класс "Продажа"), будет отвечать и за сохранение этой информации в базе данных, то получится, что логика приложения (моделирование продажи) и логика связи с базой данных "помещаются" в один класс (нарушение принципа разделения обязанностей основных объектов системы, и, кроме того, логика связи с базой данных будет дублироваться во многих других классах.

Пример: Необходимо рассчитать общую сумму продажи

- Имеются классы проектирования "Продажа", "ТоварПродажа" (продажа отдельного вида товара в рамках продажи в целом), "ТоварСпецификация" (описание конкретного вида товара).
- Необходимо распределить обязанности по предоставлению информации и расчету между этими классами. Объект "Продажа" должен передать сообщение "Рассчитать промежуточную сумму" каждому экземпляру класса "ТоварПродажа" (которые, в свою очередь, передают сообщения "СообщитьЦену" объектам "ТоварСпецификация", с целью получения информации о цене экземпляра товара), и, затем, просуммировать полученные результаты. Промежуточную сумму рассчитывает объект "Товар Продажа". Таким образом, все три объекта являются информационными экспертами

Паттерны поведения

- Основная идея паттернов этого типа — взаимодействие объектов и классов между собой.
- Они также делятся на два уровня — паттерны поведения уровня класса и паттерны поведения уровня объекта.
- Здесь самое сложное это добиться наименьшей степени связанности компонентов системы друг с другом, потому что почти все объекты должны знать о существовании других и нести в себе эту информацию (сложные паттерны «Посредник» и «Цепочка обязанностей»)

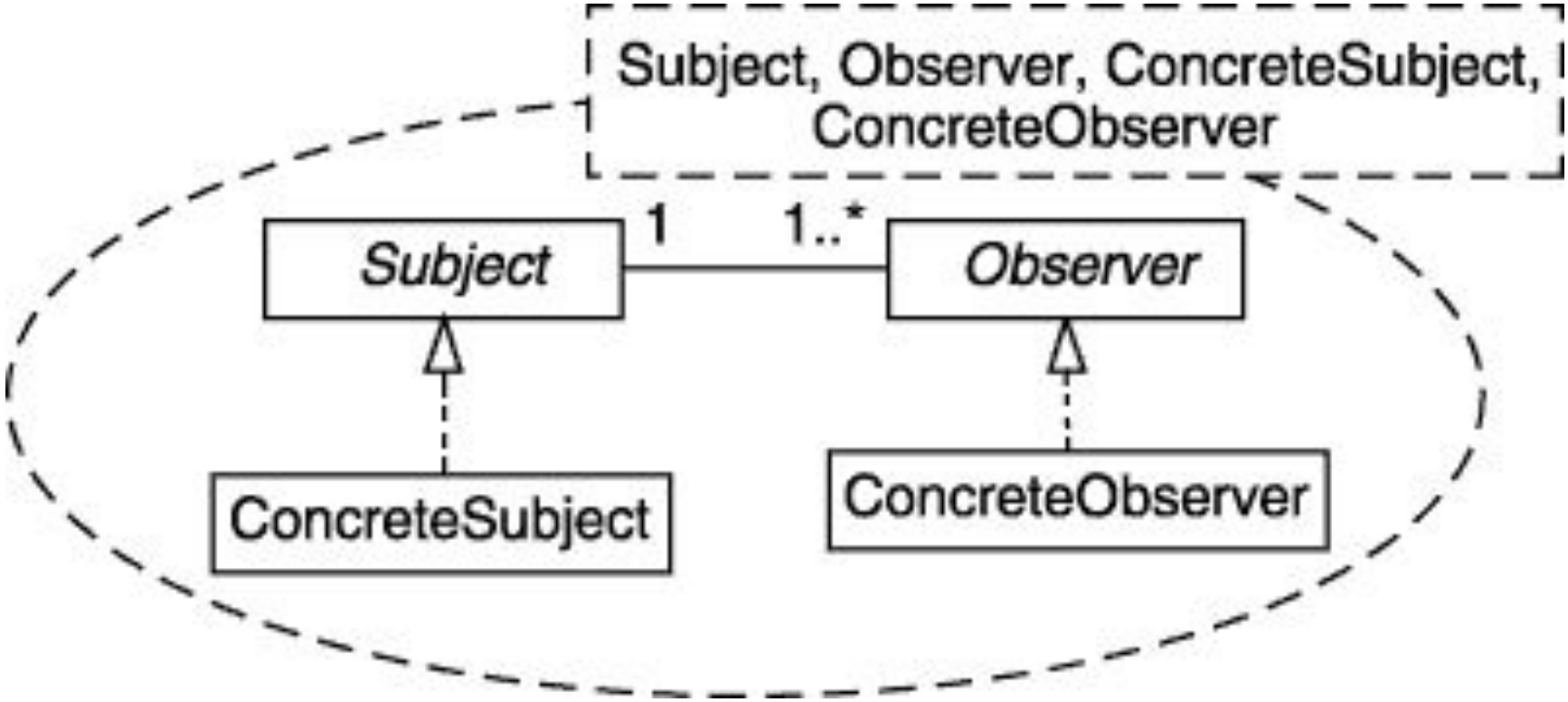
Паттерны поведения

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

Паттерн Наблюдатель

- Паттерн Наблюдатель предназначен для контроля изменений состояния объекта и передачи информации об изменении этого состояния множеству клиентов.
- В общем случае паттерн Наблюдатель также может быть изображен в виде параметризованной кооперации

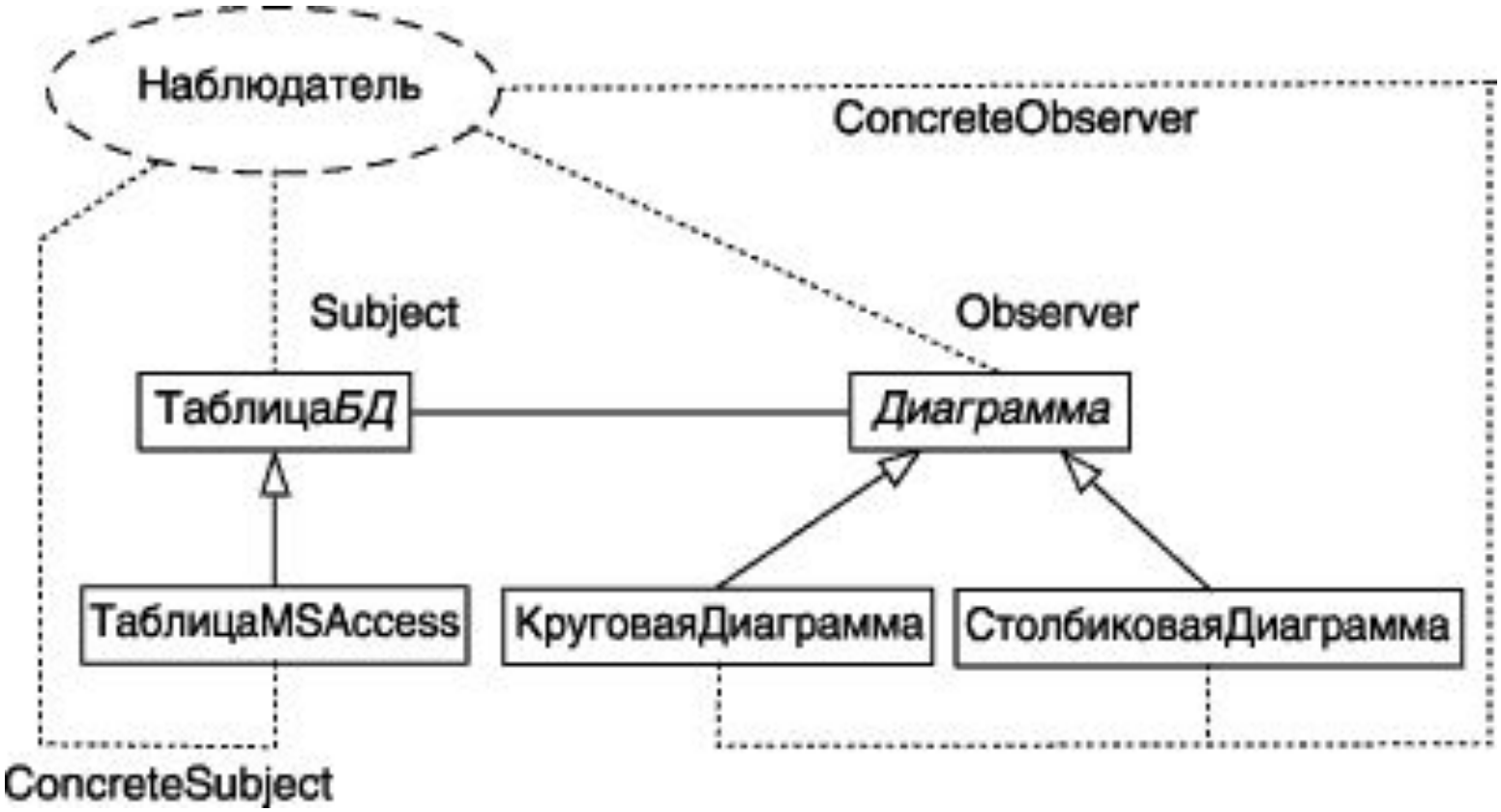
Паттерн Наблюдатель



- Изображенная параметризованная кооперация содержит 4 параметра:
  - абстрактный класс Subject (Субъект),
  - класс ConcreteSubject (Конкретный Субъект),
  - абстрактный класс Observer (Наблюдатель)
  - класс ConcreteObserver (Конкретный Наблюдатель).
- Пунктирная линия со стрелкой в форме треугольника служит для обозначения отношения обобщения классов.
- При решении конкретных задач проектирования данный паттерн также может быть конкретизирован. В этом случае вместо параметров изображенной кооперации должны быть указаны классы, предназначенные для решения отдельных задач.

Пример использования паттерна Наблюдатель

- Отслеживание изменений в таблице БД и отражении этих изменений на диаграммах. Для определенности можно использовать таблицу БД MS Access и две диаграммы - круговую и столбиковую. Фрагмент соответствующей диаграммы классов содержит 5 классов
- За субъектом Таблицей MS Access может "следить" произвольное число наблюдателей, причем их добавление или удаление не влияет на представление информации в БД.


Наблюдатель
ConcreteObserver
Subject
Observer
ТаблицаБД
Диаграмма
ТаблицаMSAccess
КруговаяДиаграмма
СтолбиковаяДиаграмма
ConcreteSubject

- Класс Таблица MS Access реализует операции по отслеживанию изменений в соответствующей таблице, и при их наличии сразу информирует абстрактного наблюдателя. Тот в свою очередь вызывает операции по перерисовке соответствующих диаграмм у конкретных наблюдателей, в качестве которых выступают классы Круговая Диаграмма и Столбиковая Диаграмма.
- Использование паттерна Наблюдатель не только упрощает взаимодействие между объектами соответствующих классов, но и позволяет вносить изменения в реализацию операций классов субъекта и наблюдателей независимо друг от друга. При этом процесс добавления или удаления наблюдателей никак не влияет на особенности реализации класса субъекта.

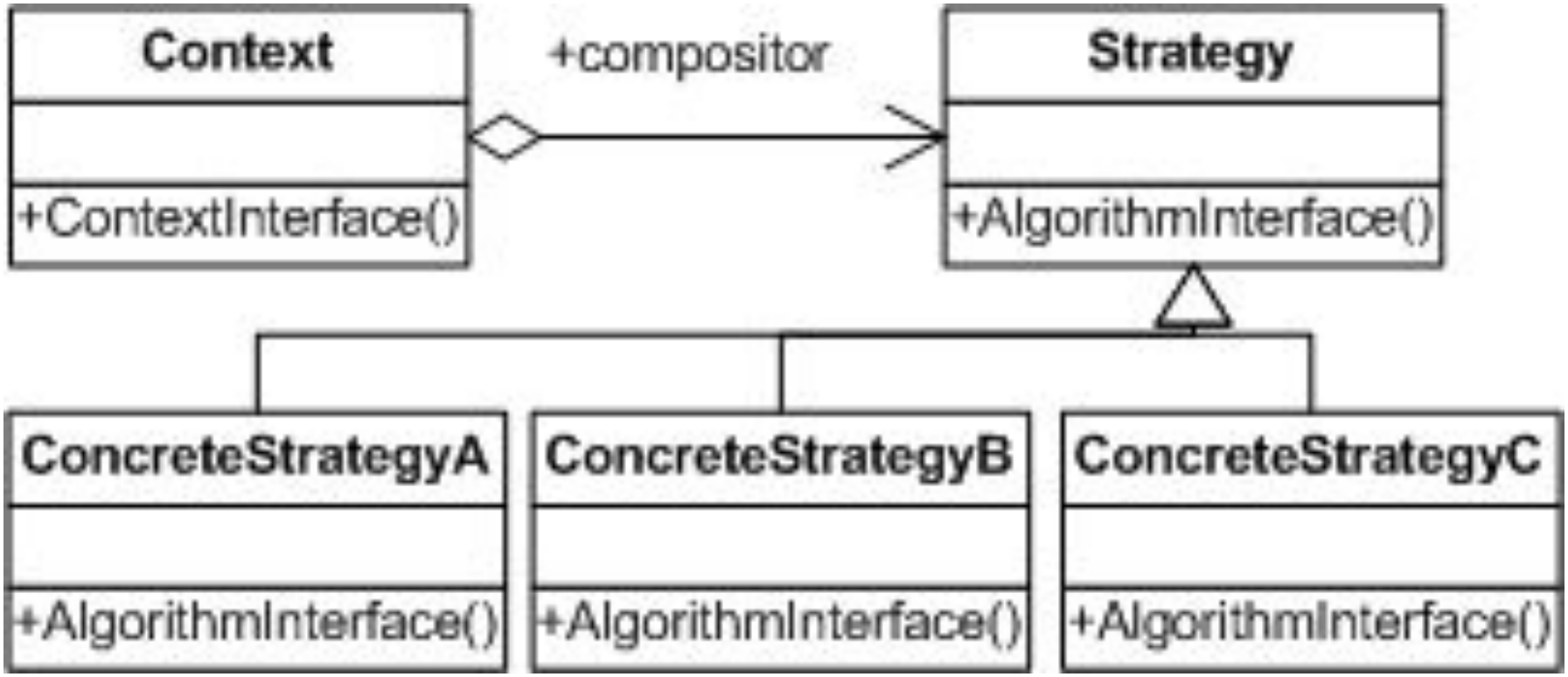
Паттерн Стратегия

- Имеется много родственных классов, отличающихся только поведением. Стратегия позволяет сконфигурировать класс, задав одно из возможных поведений;

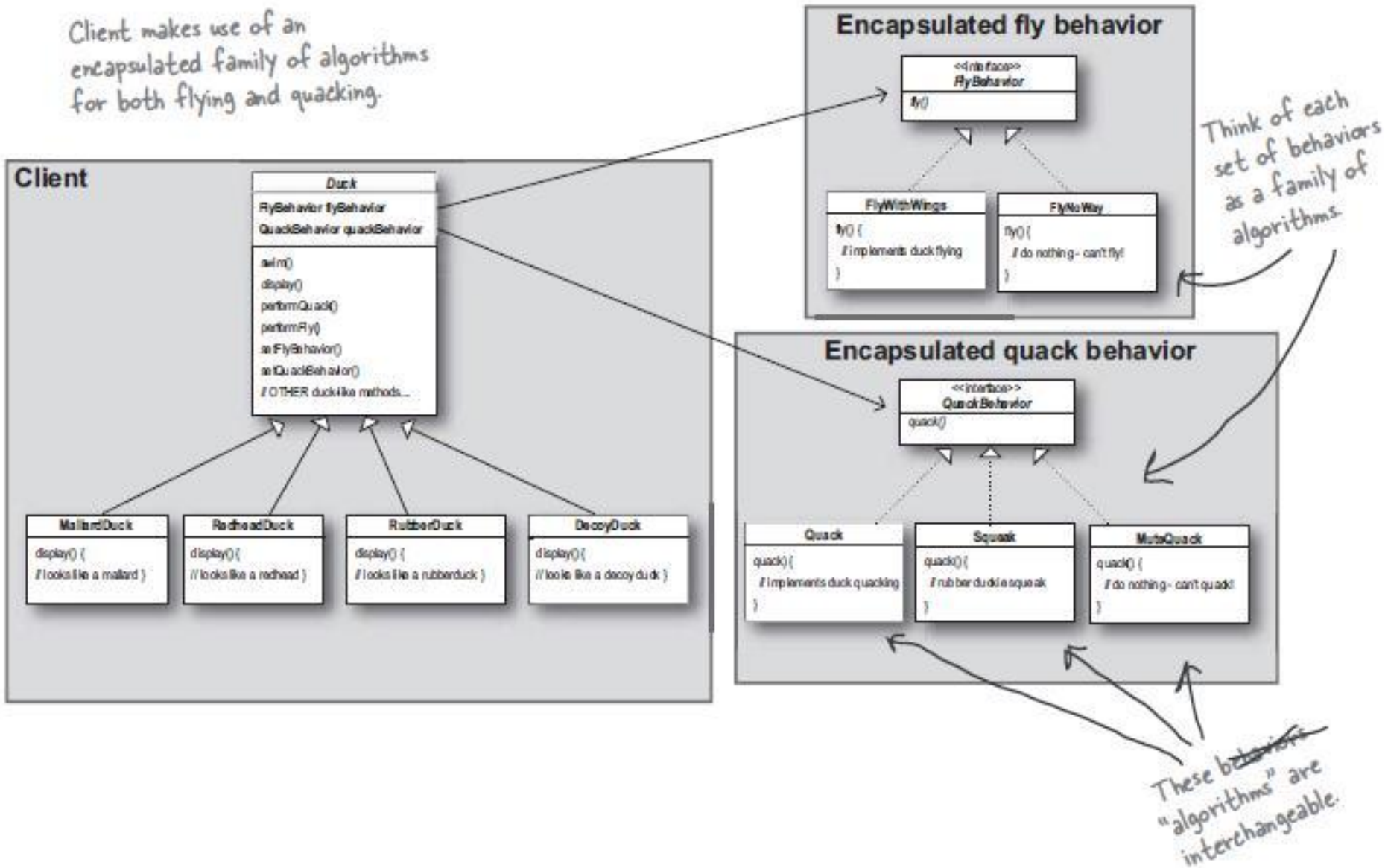
- Необходимо иметь иметь несколько разных вариантов алгоритма. Например, можно определить два варианта алгоритма, один из которых требует больше времени, а другой - больше памяти. Стратегии разрешается применять когда варианты алгоритмов реализованы в виде иерархии классов;

- В алгоритме содержатся данные, о которых клиент не должен знать. Используйте паттерн стратегия, чтобы не раскрывать сложные, специфичные для алгоритма структуры данных;

- В классе определено много поведений, что представлено разветвленными условными операторами. В этом случае проще перенести код из ветвей в отдельные классы стратегий.

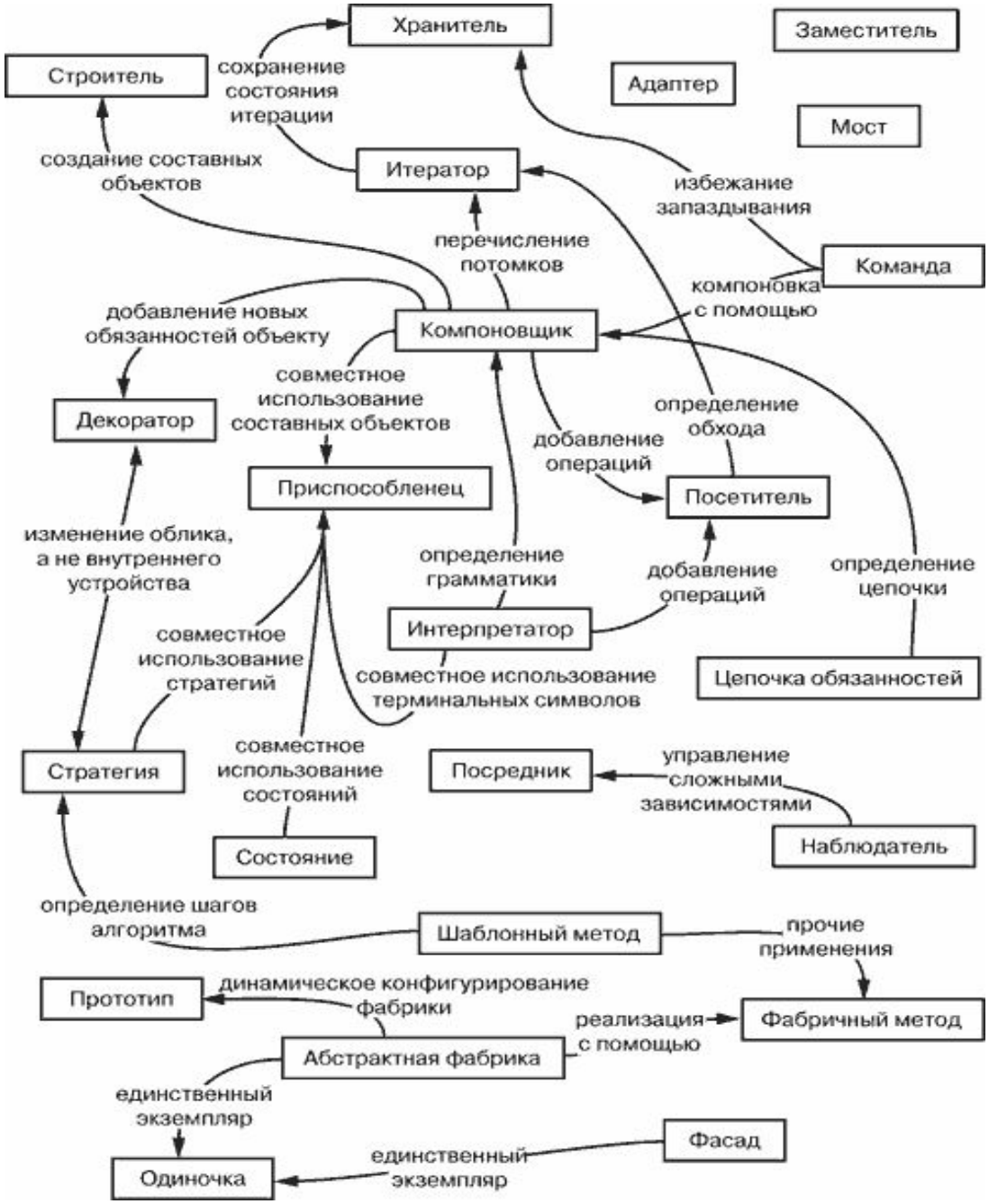

Context
+compositor
Strategy
+ContextInterface()
+AlgorithmInterface()
ConcreteStrategyA
ConcreteStrategyB
ConcreteStrategyC
+AlgorithmInterface()
+AlgorithmInterface()
+AlgorithmInterface()

- *Strategy* (Compositor) - стратегия:
  - объявляет общий для всех поддерживаемых алгоритмов интерфейс. Класс Context пользуется этим интерфейсом для вызова конкретного алгоритма, определенного в классе ConcreteStratecjy;
- *ConcreteStrategy* (SirnpleCompositor, TeXCompositor, ArrayCompositor) - конкретная стратегия:
  - реализует алгоритм, использующий интерфейс, объявленный в классе Strategy;
- *Context* (Composition) - контекст:
  - конфигурируется объектом класса ConcreteStrategy;
  - храпит ссылку на объект класса Strategy;
  - может определять интерфейс, который позволяет объекту strategy получить доступ к данным контекста


Client makes use of an encapsulated family of algorithms for both flying and quacking.
Client
Duck
FlyBehavior flyBehavior
QuackBehavior quackBehavior
swim()
display()
performQuack()
performFly()
setFlyBehavior()
setQuackBehavior()
// OTHER duck-like methods...
MallardDuck
display() {
// looks like a mallard }
RedheadDuck
display() {
// looks like a redhead }
RubberDuck
display() {
// looks like a rubberduck }
DecoyDuck
display() {
// looks like a decoy duck }
Encapsulated fly behavior
<<interface>>
FlyBehavior
fly()
FlyWithWings
fly() {
// implements duck flying
}
FlyNoWay
fly() {
// do nothing - can't fly!
}
Think of each set of behaviors as a family of algorithms.
Encapsulated quack behavior
<<interface>>
QuackBehavior
quack()
Quack
quack() {
// implements duck quacking
}
Squeak
quack() {
// rubber duckie squeak
}
MuteQuack
quack() {
// do nothing - can't quack!
}
These behaviors "algorithms" are interchangeable.

Классификация паттернов по ссылкам

- Некоторые паттерны часто используются вместе. Например, компоновщик применяется с итератором или посетителем. Некоторыми паттернами предлагаются альтернативные решения. Так, прототип нередко можно использовать вместо абстрактной фабрики. Применение части паттернов приводит к схожему дизайну, хотя изначально их назначение различно. Например, структурные диаграммы компоновщика и декоратора похожи.
- На следующем рисунке такие отношения изображены графически


Хранитель
Заместитель
Строитель
сохранение состояния итерации
Адаптер
Мост
создание составных объектов
Итератор
избежание запаздывания
перечисление потомков
Команда
компоновка с помощью
добавление новых обязанностей объекту
Компоновщик
совместное использование составных объектов
определение обхода
Декоратор
добавление операций
Посетитель
Приспособленец
изменение облика, а не внутреннего устройства
определение грамматики
добавление операций
определение цепочки
Интерпретатор
совместное использование стратегий
совместное использование терминальных символов
Цепочка обязанностей
совместное использование состояний
Стратегия
Посредник
управление сложными зависимостями
Состояние
Наблюдатель
определение шагов алгоритма
Шаблонный метод
прочие применения
Прототип
динамическое конфигурирование фабрики
реализация с помощью
Фабричный метод
Абстрактная фабрика
единственный экземпляр
Фасад
Одиночка
единственный экземпляр

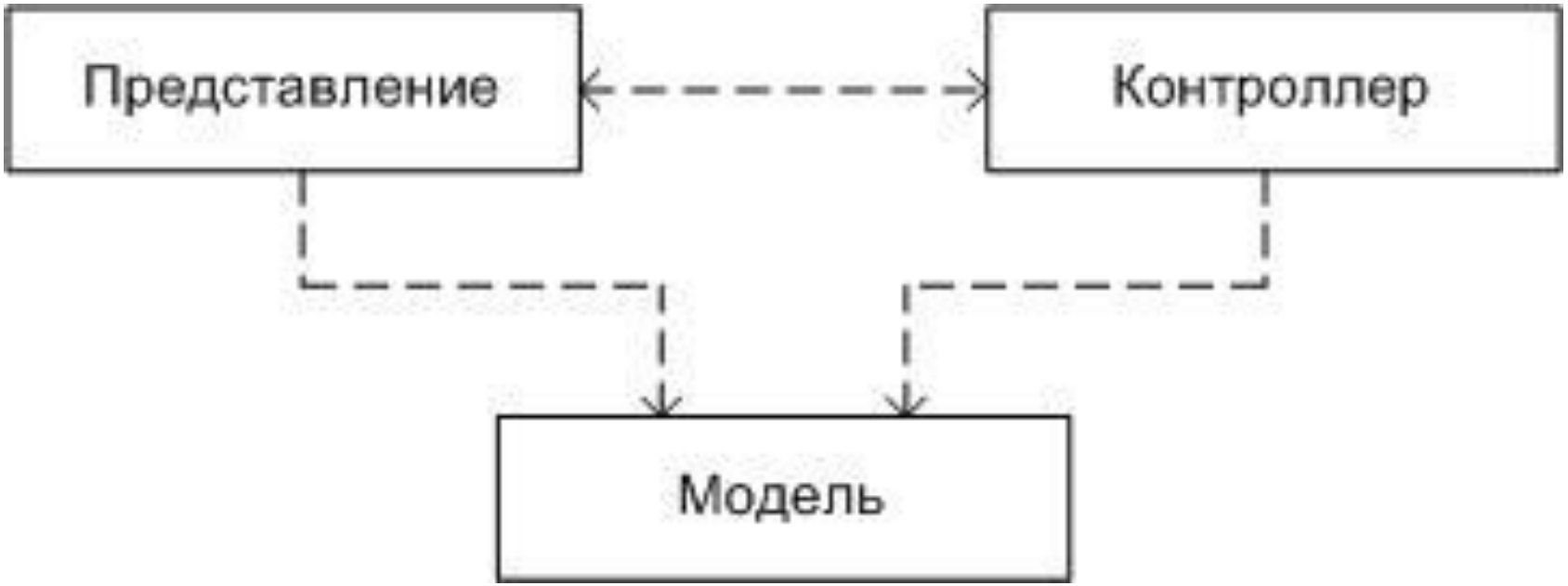
Вариант классификации шаблонов проектирования

- Порождающие паттерны
- Системные паттерны
- Структурные паттерны
- Паттерны распределения обязанностей
- Паттерны домена
- Паттерны слоя источника данных
- Паттерны моделирования поведения
- Паттерны для представления данных в Web

Паттерны для разработки Web-приложений

- *Контроллер приложения (Application Controller)*
- *Контроллер запросов (Front Controller)*
- *Контроллер страниц (Page Controller)*
- *Двухэтапное представление (Two Step View)*
- *Представление с преобразованием (Transform View)*
- *Представление по шаблону (Template View)*
- *Модель-представление-контроллер (Model View Controller)*

Модель-представление-контроллер (Model View Controller)



- Типовое решение **модель-представление-контроллер** подразумевает выделение трех отдельных ролей.
- Модель - это объект, предоставляющий некоторую информацию о домене. У модели нет визуального интерфейса, она содержит в себе все данные и поведение, не связанные с пользовательским интерфейсом.
- В объектно-ориентированном контексте наиболее "чистой" формой модели является объект *модели предметной области (Domain Model)*. В качестве модели можно рассматривать и *сценарий транзакции (Transaction Script)*, если он не содержит в себе никакой логики, связанной с пользовательским интерфейсом.

- Представление отображает содержимое модели средствами графического интерфейса. Функции представления заключаются только в отображении информации на экране.
- Все изменения информации обрабатываются третьим “участником” нашей системы - контроллером. Контроллер получает входные данные от пользователя, выполняет операции над моделью и указывает представлению на необходимость соответствующего обновления.
- В этом плане графический интерфейс можно рассматривать как совокупность представления и контроллера.

- Ключевым моментом в отделении представления от модели является направление зависимостей: представление зависит от модели, но модель не зависит от представления.
- Программисты, занимающиеся разработкой модели, вообще не должны быть осведомлены о том, какое представление будет использоваться. Это существенно облегчает разработку модели и одновременно упрощает последующее добавление новых представлений. Кроме того, это означает, что изменение представления не требует изменения модели.

Другие типы шаблонов

- **аналитические шаблоны**, описывают основной подход для составления требований для программного обеспечения (requirement analysis) до начала самого процесса программной разработки
- **коммуникационные шаблоны**, описывают процесс общения между отдельными участниками/сотрудниками организации
- **организационные шаблоны**, описывают организационную иерархию предприятия/фирмы
- **Анти-паттерны** (Anti-Design-Patterns) описывают как не следует поступать при разработке программ, показывая характерные ошибки в дизайне и в реализации.

Анти-паттерны

- **Анти-паттерны** (anti-patterns), также известные как **ловушки** (pitfalls) — это классы наиболее часто внедряемых плохих решений проблем. Они изучаются, как категория, для того , чтобы избежать их в будущем.
- Термин происходит из информатики, из книги «Банды четырёх» *Шаблоны проектирования*, которая заложила примеры практики хорошего программирования. Авторы назвали эти хорошие методы «шаблонами проектирования», и противоположными им являются «анти-паттерны». Частью хорошей практики программирования является избегание анти-паттернов.

Анти-паттерны в управлении разработкой ПО

- Дым и зеркала (Smoke and mirrors): Демонстрация того, как будут выглядеть ненаписанные функции (название происходит от двух излюбленных способов, которыми фокусники скрывают свои секреты).
- Раздувание ПО (Software bloat): Разрешение последующим версиям системы требовать всё больше и больше ресурсов.
- Функции для галочки: Превращение программы в конгломерат плохо реализованных и не связанных между собой функций (как правило, для того, чтобы заявить, что функция есть).

Анти-паттерны в объектно-ориентированном программировании

- **Базовый класс-утилита** (BaseBean): Наследование функциональности из класса-утилиты вместо делегирования к нему
- **Вызов предка** (CallSuper): Для реализации прикладной функциональности методу класса-потомка требуется в обязательном порядке вызывать те же методы класса-предка.
- **Божественный объект** (God object): Концентрация слишком большого количества функций в одной части системы (классе)
- **Полтергейст** (компьютер) (Poltergeist): Объекты, чьё единственное предназначение — передавать информацию другим объектам

- При реализации проектов по разработке программных систем и моделированию бизнес-процессов встречаются ситуации, когда решение проблем в различных проектах имеют сходные структурные черты. Попытки выявить похожие схемы или структуры в рамках объектно-ориентированного анализа и проектирования привели к появлению понятия паттерна, которое из абстрактной категории превратилось в непременный атрибут современных CASE-средств
- Паттерны ООАП различаются степенью детализации и уровнем абстракции.

Общая классификация паттернов по категориям их применения

- Архитектурные паттерны
- Паттерны проектирования
- Паттерны анализа
- Паттерны тестирования
- Паттерны реализации

Архитектурные паттерны (Architectural patterns)

- Архитектурные паттерны(Architectural patterns) - множество предварительно определенных подсистем со спецификацией их ответственности, правил и базовых принципов установления отношений между ними.

- Архитектурные паттерны предназначены для спецификации фундаментальных схем структуризации программных систем.
- Наиболее известными паттернами этой категории являются паттерны GRASP (General Responsibility Assignment Software Pattern).
- Эти паттерны относятся к уровню системы и подсистем, но не к уровню классов. Как правило, формулируются в обобщенной форме, используют обычную терминологию и не зависят от области приложения.
- Паттерны этой категории систематизировал и описал К. Ларман.

Структурные архитектурные паттерны

- Для организации классов или обьектов системы в базовые подструктуры (в подсистемы)
  - Репозиторий
  - Клиент/сервер
  - Обьектно - ориентированный, Модель предметной области (Domain Model), модуль таблицы (Data Mapper)
  - Многоуровневая система (Layers) или абстрактная машина
  - Потоки данных (конвейер или фильтр)

паттерн "Многоуровневая система"

- В соответствии с паттерном "Многоуровневая система" структурные элементы системы организуются в отдельные уровни со взаимосвязанными обязанностями таким образом, чтобы на нижнем уровне располагались низкоуровневые службы и службы общего назначения, а на более высоких - объекты уровня логики приложения.
- При этом взаимодействие и связывание уровней происходит сверху вниз.
- Связывания объектов снизу вверх следует избегать.


Представление
Домен (предметная область, бизнес - логика)
Источники данных

- Слой представления охватывает все, что имеет отношение к общению пользователя с системой. К главным функциям слоя представления относится отображение информации и интерпретация вводимых пользователем команд с преобразованием их в соответствующие операции в контексте домена (бизнес - логики) и источника данных.
- Источник данных - подмножество функций, обеспечивающих взаимодействие со сторонними системами, которые выполняют
- В отличие от архитектурного паттерна "Клиент - сервер" слои вовсе не обязательно должны располагаться на разных машинах.

Пример

- Примером данного подхода может служить модель взаимодействия открытых систем (OSI - Open System Interconnection - международная программа стандартизации обмена данными между компьютерными системами на основе семиуровневой модели протоколов передачи данных в открытых системах).

Преимущества и недостатки

- Многоуровневая система может быть разработана пошагово (итеративно).
- Изменение исходного кода влечет за собой переделку всех элементов системы, поскольку все элементы системы тесно связаны друг с другом. Логика приложения тесно связана с интерфейсом пользователя - затруднительно менять интерфейс или принципы реализации логики. Из-за высокой связанности, работу по реализации системы сложно разделить между разработчиками и, кроме того, сложно модифицировать функции приложения или переходить на новые технологии

Паттерны управления

- Для обеспечения взаимодействия отдельных архитектурных элементов системы
  - централизованного управления (одна из подсистем полностью отвечает за управление, запускает и завершает работу остальных подсистем)
  - децентрализованное реагирование на события (согласно этим паттернам на внешние события отвечает соответствующая подсистема.).

- Проектирование взаимодействия подсистем с реляционной базой данных является неотъемлемой частью разработки корпоративных информационных систем, среди паттернов управления выделена большая группа паттернов, описывающих организацию связи с базой данных

Паттерны проектирования (Design patterns)

- Паттерны проектирования (Design patterns) - специальные схемы для уточнения структуры подсистем или компонентов программной системы и отношений между ними.

- Паттерны проектирования описывают общую структуру взаимодействия элементов программной системы, которые реализуют исходную проблему проектирования в конкретном контексте.
- Наиболее известными паттернами этой категории являются паттерны GoF (Gang of Four), названные в честь Э. Гаммы, Р. Хелма, Р. Джонсона и Дж. Влиссидеса, которые систематизировали их и представили общее описание.
- Паттерны GoF включают в себя 23 паттерна. Эти паттерны не зависят от языка реализации, но их реализация зависит от области приложения.

Паттерны анализа (Analysis patterns)

- Паттерны анализа (Analysis patterns) - специальные схемы для представления общей организации процесса моделирования.
- Паттерны анализа относятся к одной или нескольким предметным областям и описываются в терминах предметной области.
- Наиболее известными паттернами этой группы являются паттерны бизнес-моделирования ARIS (Architecture of Integrated Information Systems), которые характеризуют абстрактный уровень представления бизнес-процессов. В дальнейшем паттерны анализа конкретизируются в типовых моделях с целью выполнения аналитических оценок или имитационного моделирования бизнес-процессов.

- Наиболее известными паттернами этой группы являются паттерны бизнес-моделирования ARIS (Architecture of Integrated Information Systems), которые характеризуют абстрактный уровень представления бизнес-процессов.
- В дальнейшем паттерны анализа конкретизируются в типовых моделях с целью выполнения аналитических оценок или имитационного моделирования бизнес-процессов.

Паттерны тестирования (Test patterns)

- Паттерны тестирования (Test patterns) - специальные схемы для представления общей организации процесса тестирования программных систем.

- К этой категории паттернов относятся такие паттерны, как тестирование черного ящика, белого ящика, отдельных классов, системы.
- Паттерны этой категории систематизировал и описал М. Гранд. Некоторые из них реализованы в инструментальных средствах, наиболее известными из которых является IBM Test Studio. В связи с этим паттерны тестирования иногда называют стратегиями или схемами тестирования.

Паттерны реализации (Implementation patterns)

- Паттерны реализации (Implementation patterns) - совокупность компонентов и других элементов реализации, используемых в структуре модели при написании программного кода.

- Эта категория паттернов делится на следующие подкатегории:
  - паттерны организации программного кода,
  - паттерны оптимизации программного кода,
  - паттерны устойчивости кода,
  - паттерны разработки графического интерфейса пользователя и др.
- Паттерны этой категории описаны в работах М. Гранда, К. Бека, Дж. Тидвелла и др.
- Некоторые из них реализованы в популярных интегрированных средах программирования в форме шаблонов создаваемых проектов. В этом случае выбор шаблона программного приложения позволяет получить некоторую заготовку программного кода.

Мартин Фаулер:

- Не бойтесь потратить время на изучение паттернов;
- Хорошо подумайте, когда лучше всего применить паттерн;
- Хорошо подумайте, как лучше всего реализовать паттерн в его наипростейшей форме, а уже потом вносите дополнения;
- Если вы применили паттерн, а потом поняли, что без него было бы лучше - убирайте, не сомневайтесь