

Вопрос: Образцы (паттерны) проектирования, их классификация и способ описания. Пример образца.

Образец (или паттерн) – это типовое проектное решение конкретной задачи проектирования, описанное специальным образом, чтобы облегчить его повторное применение.

Фактически, каждый паттерн является формализованным опытом лучших разработчиков в индустрии создания ПО.

Основные составляющие части описания образца:

Имя. Идентифицирует образец. Хорошее имя характеризует решаемую проблему и способ ее решения.

Задача. Описание ситуации, в которой следует применять образец. Это описание включает в себя: постановку проблемы, контекст проблемы, перечень условий, при выполнении которых имеет смысл применять образец.

Решение. Описание элементов архитектуры, связей между ними, функций каждого элемента. Включает в себя UML-диаграммы.

Результаты. Следствия применения паттерна и компромиссы. Преимущества и недостатки образца. Влияние использования образца на гибкость, расширяемость и переносимость системы.

Каталог образцов «банда четырёх» содержит более 20 образцов, сгруппированных на 3 части:

- порождающие образцы (способы создания экземпляров классов);
- структурные образцы (способы задания статических связей между проектными классами);
- образцы поведения (способы организации взаимодействий между объектами).

Рассмотрим по одному примеру из каждой группы образцов. (При ответе на вопрос достаточно привести один из трёх примеров и дать по нему пояснения.)

Мост (Bridge)

Классификация: структурный образец.

Назначение: отделить абстракцию от реализации.

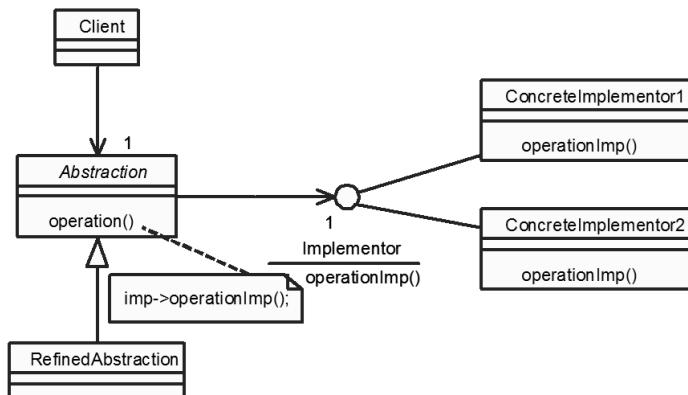
Мотивация: наследование жестко привязывает реализацию к абстракции, поэтому лучше иметь иерархию наследования для интерфейсов и отдельно их реализаций.

Ситуации применимости:

- обеспечение независимости абстракции и реализации;
- необходимо расширять подклассами как интерфейсы, так и их реализации;
- изменения в реализации не должны влиять на клиента;
- необходимо разделить большую иерархию наследования на части.

Участники:

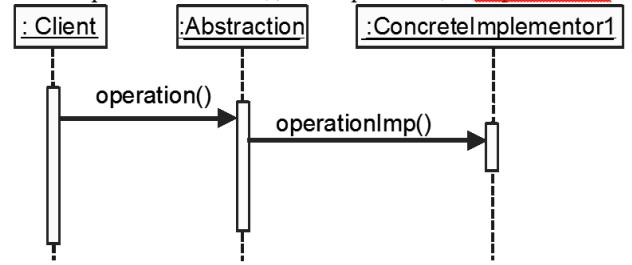
- Abstraction – абстракция, в которой определен интерфейс требуемый клиенту;
- RefinedAbstraction – уточненная абстракция с расширенным интерфейсом;
- Implementor – интерфейс для классов-реализаций;
- ConcreteImplementor – конкретный реализатор.



Отношения: Абстракция перенаправляет запросы клиента к одной из реализаций Implementator.

Результаты:

- реализация отделяется от интерфейса;
- чтобы заменить реализацию нет необходимости перекомпилировать абстракцию и ее клиента;
- система становится более легко модифицируемой.



Пример: Пусть есть абстракция Shape (форма), в ней есть операция draw(), отвечающая за отрисовку. В каждой конкретной форме (Rectangle, Circle) отрисовка реализуется с помощью примитивов drawLine(), drawCircle(), описанных в интерфейсе Drawing, реализуемом разными графическими утилитами DrawingV1, DrawingV2, рассчитанными на работу с разными графическими устройствами Driver1, Driver2. Диаграмма классов:

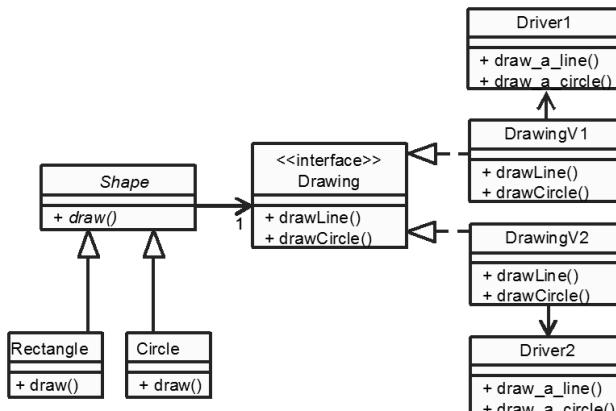
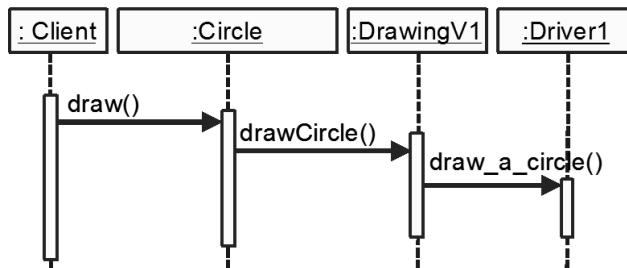
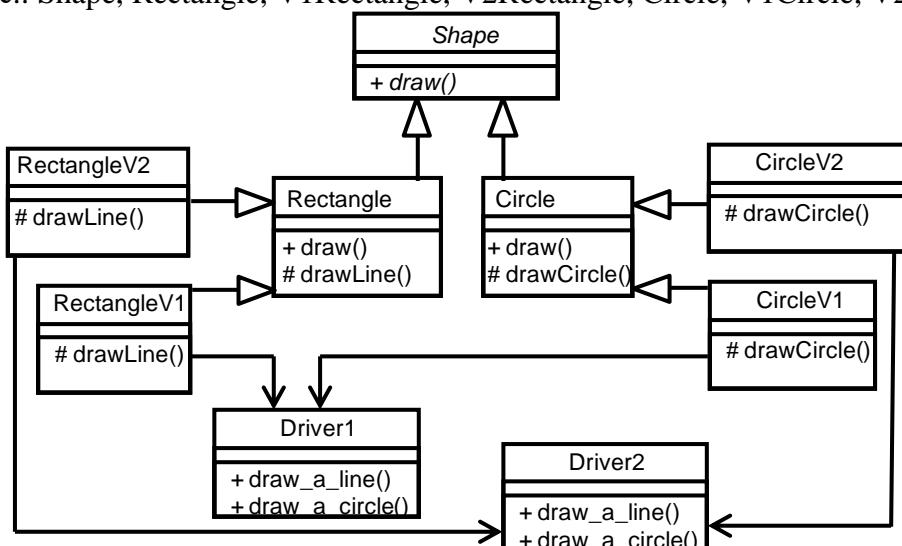


Диаграмма взаимодействия:



Если не применять образец, то у Rectangle и Circle могли бы быть два наследника, каждый из которых рассчитан на работу с одним из двух вариантов графики. Т. е. в иерархии форм было бы 7 классов (см на рис.: Shape, Rectangle, V1Rectangle, V2Rectangle, Circle, V1Circle, V2Circle).



Если добавить ещё формы – наследницы Shape – Triangle, PolyLine, то в первом случае при их отрисовке дополнительные классы не нужны, так как можно воспользоваться реализациями Drawing. Во втором случае иерархия разрастается, в ней становится 13 классов (добавляются Triangle,

TriangleV1, TriangleV2, PolyLine, PolyLineV1, PolyLineV2).

Аналогично применение паттерна Мост выгодно при добавлении поддержки еще одного графического устройства Driver3. Будет достаточно добавить новую реализацию интерфейса Drawing – класс DrawingV3, вместо того, чтобы заводить каждой конкретной фигуре наследника с реализацией отрисовки для нового устройства (RectangleV3, CircleV3, TriangleV3, PolyLineV3).

Strategy (Стратегия)

Классификация: образец поведения.

Назначение: Определяет семейство алгоритмов, инкапсулирует каждый из них и делает их взаимозаменяемыми.

Мотивация: есть несколько алгоритмов решения одной задачи, которые нежелательно «зашивать» в клиентский класс.

Ситуации применимости:

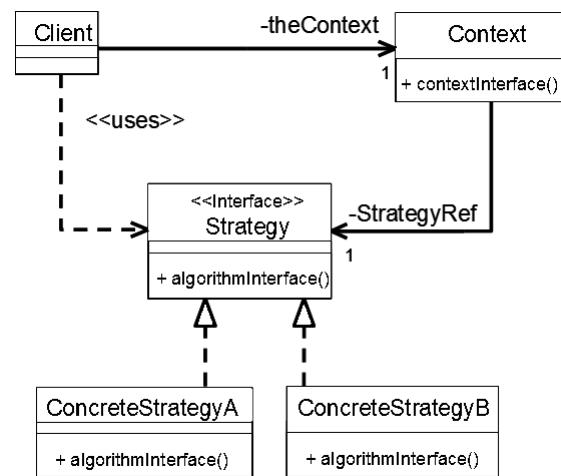
- Имеется много родственных классов, отличающихся только поведением.
- Необходимо иметь несколько разных реализаций одной операции.
- Нужно скрыть от клиента сложные, специфичные для алгоритма структуры данных.
- Упрощение кода метода, представляющего собой длинное ветвление или switch.

Участники:

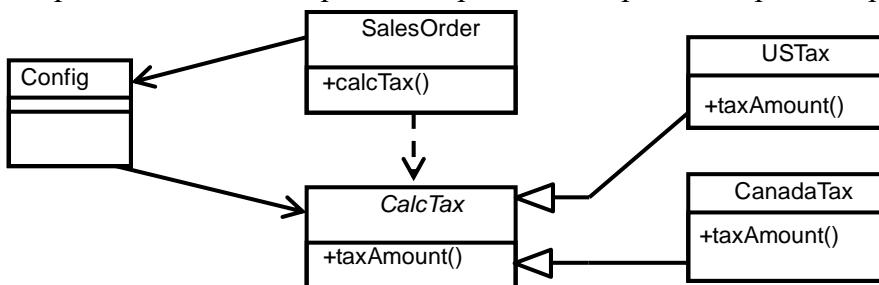
- Strategy – интерфейс общий для семейства алгоритмов;
- ConcreteStrategy – конкретная стратегия, реализующая интерфейс;
- Context – контекст, направляющий запросы клиента стратегиям;
- Client – клиентский класс.

Результаты:

- Иерархия классов стратегий определяет семейство алгоритмов или поведений, которые можно повторно использовать.
- Инкапсуляция алгоритма в отдельный класс позволяет изменять его независимо от контекста.
- Избавляемся от if и switch (улучшаем читаемость кода).
- Интерфейс класса Strategy общий для всех подклассов ConcreteStrategy – неважно, сложна или тривиальна их реализация. Поэтому вполне вероятно, что некоторые стратегии не будут пользоваться всей передаваемой им информацией, особенно простые.



Приведем пример использования образца для реализации разных стратегий расчета налогов:



Предполагается, что объект класса Config сообщает SalesOrder ссылку на объект-алгоритм расчета налогов (либо экземпляр USTax, пригодный для США, либо CanTax, пригодный для Канады). Если потребуется добавить новые способы расчета, достаточно добавить подклассы CalcTax. Обратите внимание, что в примере вместо интерфейса и реализации используется абстрактный класс и связи обобщения.

Альтернативой предложенному решению является внесение внутрь SalesOrder::calcTax() логики выбора схемы расчета и реализация расчетов в отдельных операциях SalesOrder. Модифицируемость такого решения ниже, чем при использовании образца.

Абстрактная фабрика (Abstract Factory)

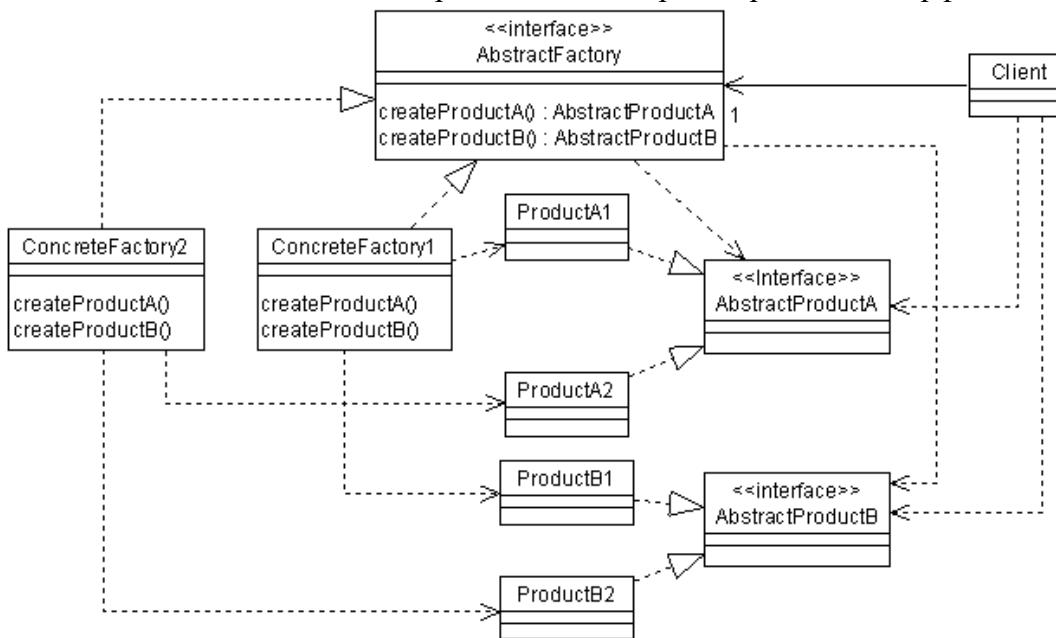
Классификация: образец порождения объектов.

Назначение: предоставляет интерфейс для создания взаимосвязанных и взаимозависимых объектов, не определяя их конкретных классов.

Мотивация: часто встает задача проектирования программной системы независимой от конкретной реализации GUI.

Ситуации применимости:

- Система не должна зависеть от того как создаются, компонуются и представляются входящие в нее объекты;
- Входящие в семейство объекты должны использоваться вместе и необходимо обеспечить выполнение этого ограничения;
- Система должна конфигурироваться одним из семейств составляющих ее объектов;
- Предоставляется библиотека классов, реализация которых скрыта за интерфейсом.

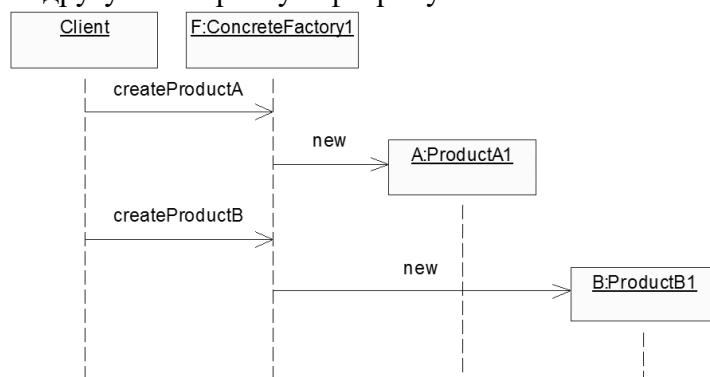


Участники:

- **AbstractFactory** – интерфейс с операциями для порождения экземпляров абстрактных классов-продуктов.
- **ConcreteFactory** – реализация порождения экземпляров конкретных классов.
- **AbstractProduct** – интерфейс с операциями класса-продукта.
- **ConcreteProduct** – реализация абстрактного продукта, объекты которой порождаются одной из конкретных фабрик.
- **Client** – класс, использующий интерфейсы **AbstractFactory** и **AbstractProduct**.

Отношения:

Обычно, во время выполнения создается один экземпляр **ConcreteFactory**, который создает экземпляры конкретных продуктов одного из семейств. Для использования объектов другого семейства нужно породить другую конкретную фабрику.



Результаты:

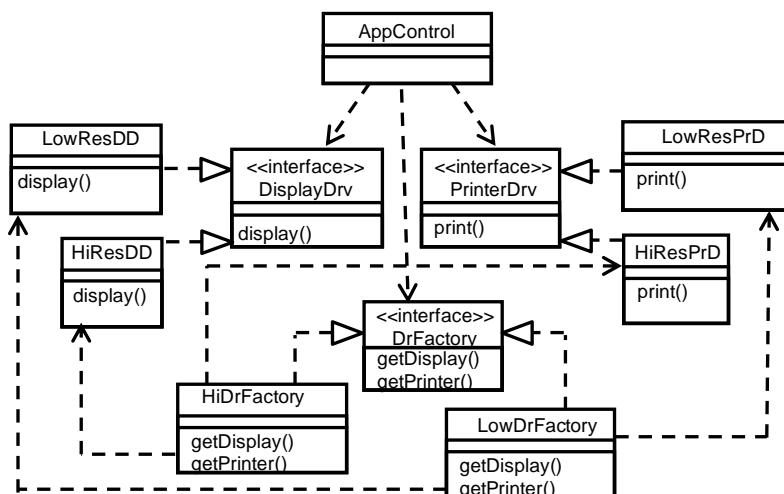
- изоляция клиента от деталей реализации классов-продуктов (их имена известны только конкретной

фабрике);

- упрощение замены семейств продуктов;
- набор классов-продуктов фиксирован, добавлять новые продукты в семейства трудно.

Представим, что нужно добавить третий класс продуктов. Потребуется добавить иерархию из 3-х классов и дополнительный метод в каждую фабрику, что довольно затратно.

Пример: две фабрики обеспечивают производство семейств классов-драйверов, работающих с низким или высоким разрешением. Предполагается, что разрешение драйвера принтера должно соответствовать дисплейному.



Без применения образца пришлось бы связывать класс AppControl прямыми зависимостями с классами LowResDD, HiResDD, LowResPrD, HiResPrD. На диаграмме, приведённой выше, предполагается, что классы LowResPrD и HiResPrD могут иметь общий интерфейс (или общий суперкласс). Если это не так, совместно с образцом Абстрактная фабрика следует применить образец Адаптер.

Литература к вопросу

1. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. – СПб.: Питер, 2016.
2. Фримен Э., Фримен Э. и др. Паттерны проектирования – СПб: Питер, 2016.
3. Материалы по курсу ООАП/МАППО. Конспект лекции №9 (conspect09.pdf) и слайды к лекции №9 (slides09.pdf):
<https://drive.google.com/drive/folders/1he5B9iNXlbhpkW-96GOKItWGm1HRSwle?usp=sharing>