Паттерны проектирования GOF (порождающие и структурные)

Технологии разработки программного обеспечения

Виноградова М.В.
МГТУ им. Н.Э. Баумана
Кафедра СОИУ (ИУ5)

Кооперация

- Кооперация (сотрудничество) - это средство представления комплексных решений в разработке ПО на внешнем, архитектурном уровне.
  - это цельная спецификация ПО
  - это реализация потоков управления и структур данных
  - это реализация прецедента
- статическая составляющая – структура совместно работающих классов (диаграмма классов – участников)
- динамическая составляющая - поведение совместно работающих элементов (диаграммы последовательностей)

Паттерны (шаблоны) проектирования

- Паттерн – это параметризированная настраиваемая кооперация для решения типичной проблемы в определенном контексте.
- Дает:
  - уменьшение затрат на анализ и проектирование ПО,
  - повышение качества и правильности разработки на логическом уровне.
- Паттерны — это наборы готовых решений с возможностью повторного использования.
- Создаются профессионалами.
- Отражают проверенные и оптимизированные решения.

Паттерны GOF – Gang of four

- «Команда четырех»
- **Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.**
- «Приемы объектно-ориентированного проектирования. Паттерны проектирования»
- 23 паттерна на высоком уровне абстракции
- Виды паттернов:
  - Порождающие
  - Структурные
  - Поведенческие

Описание паттерна

Название	**Описание**
Имя	Выразительное имя паттерна дает возможность указать проблему проектирования, ее решение и последствия ее решения. Использование имен паттернов повышает уровень абстракции проектирования
Задача/ проблема	Формулируется проблема проектирования (и ее контекст), на которую ориентировано применение паттерна. Задаются условия применения
Решение	Описываются элементы решения, их отношения, обязанности, сотрудничество. Решение представляется в обобщенной форме, которая должна конкретизироваться при применении. Фактически приводится шаблон решения — его можно использовать в самых разных ситуациях
Результат	Перечисляются следствия применения паттерна и вытекающие из них компромиссы. Такая информация позволяет оценить эффективность применения паттерна в данной ситуации

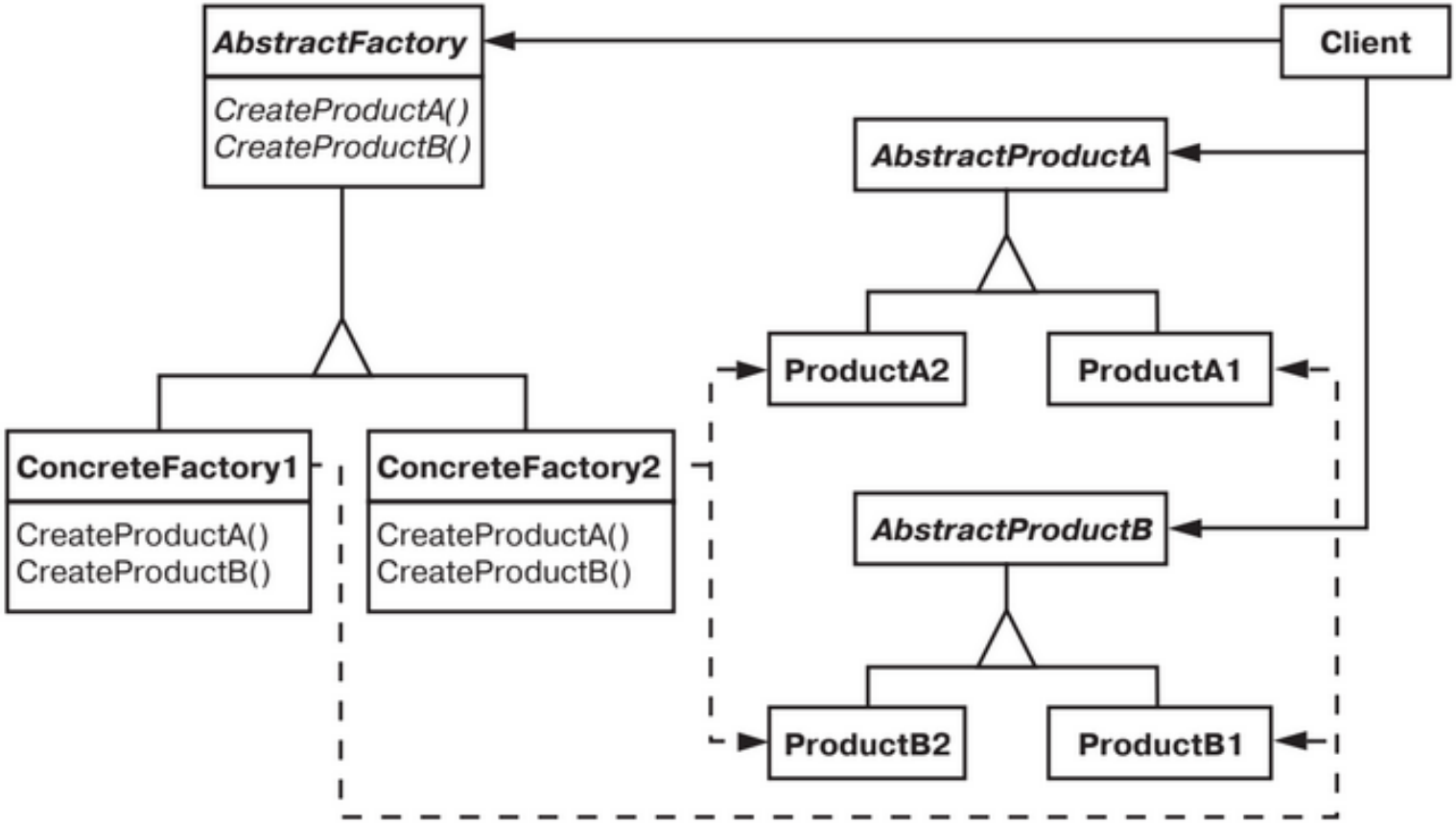
Порождающие паттерны

- **Абстрактная фабрика (Abstract factory)** - класс, который представляет собой интерфейс для создания семейства компонентов системы.
- **Фабричный метод (Factory method)** - определяет интерфейс для создания объекта, но оставляет подклассам решение о том, какой класс инстанцировать.
- **Прототип (Prototype)** - определяет интерфейс создания объекта через клонирование другого объекта вместо создания через конструктор.
- **Строитель (Builder)** - класс, который представляет собой интерфейс для создания сложного объекта.
- **Одиночка (Singleton)** - класс, который может иметь только один экземпляр.

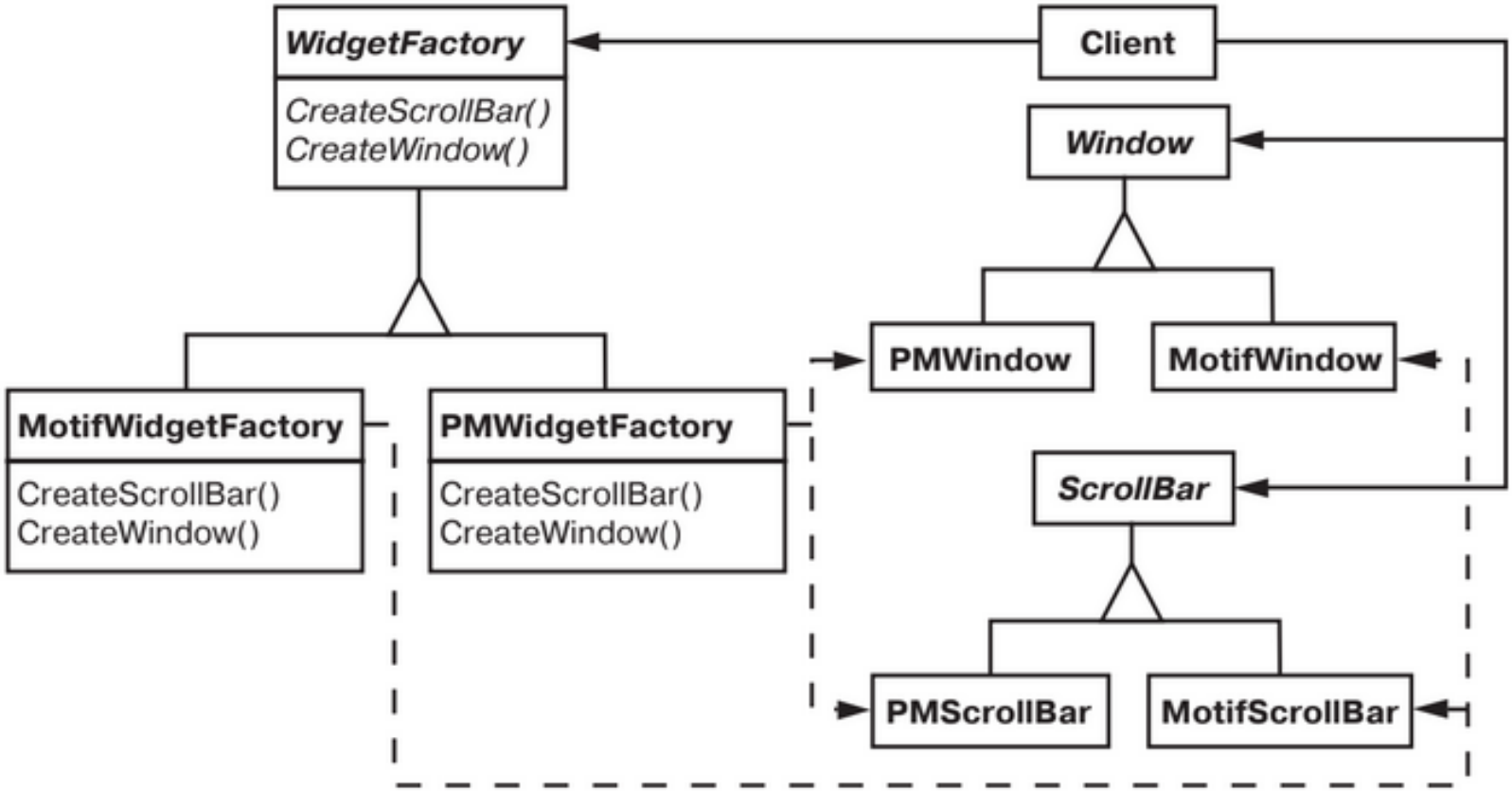
Абстрактная фабрика (Abstract Factory)

- Предоставляет интерфейс для создания семейства взаимосвязанных или взаимозависимых объектов, не специфицируя их конкретных классов.
- Проблема:
  - Система не должна зависеть от того, как создаются, компонуются и представляются входящие в неё объекты.
  - Семейство объектов должно использоваться совместно.
  - Система должна конфигурироваться одним из семейств объектов.
  - Требуется создавать библиотеку объектов через интерфейсы (сокрытие реализации).

Абстрактная фабрика - решение



Абстрактная фабрика - пример



Абстрактная фабрика - результат

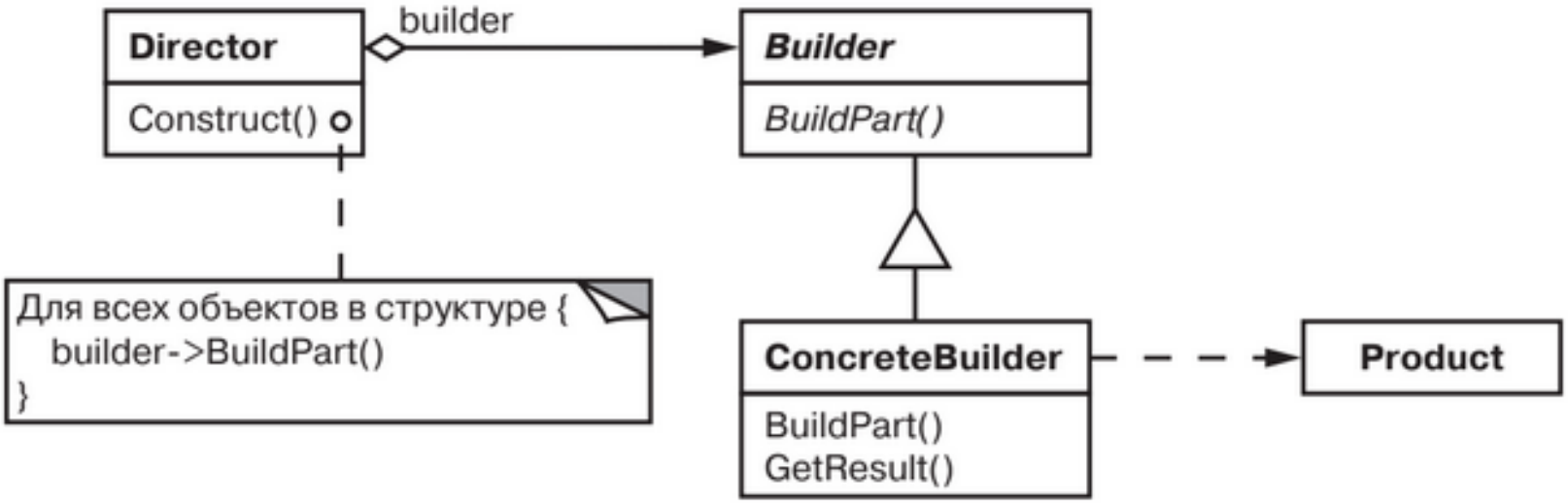
- (+) Изолирует конкретные классы (имена неизвестны клиенту).
- (+) Гарантирует сочетаемость продуктов (в одном семействе).
- (+) Упрощает замену семейств продуктов.
- (-) Сложно расширять фабрику.

- Использование:
  - конкретная фабрика - как Одиночка;
  - абстрактная фабрика - через фабричный метод/ прототип

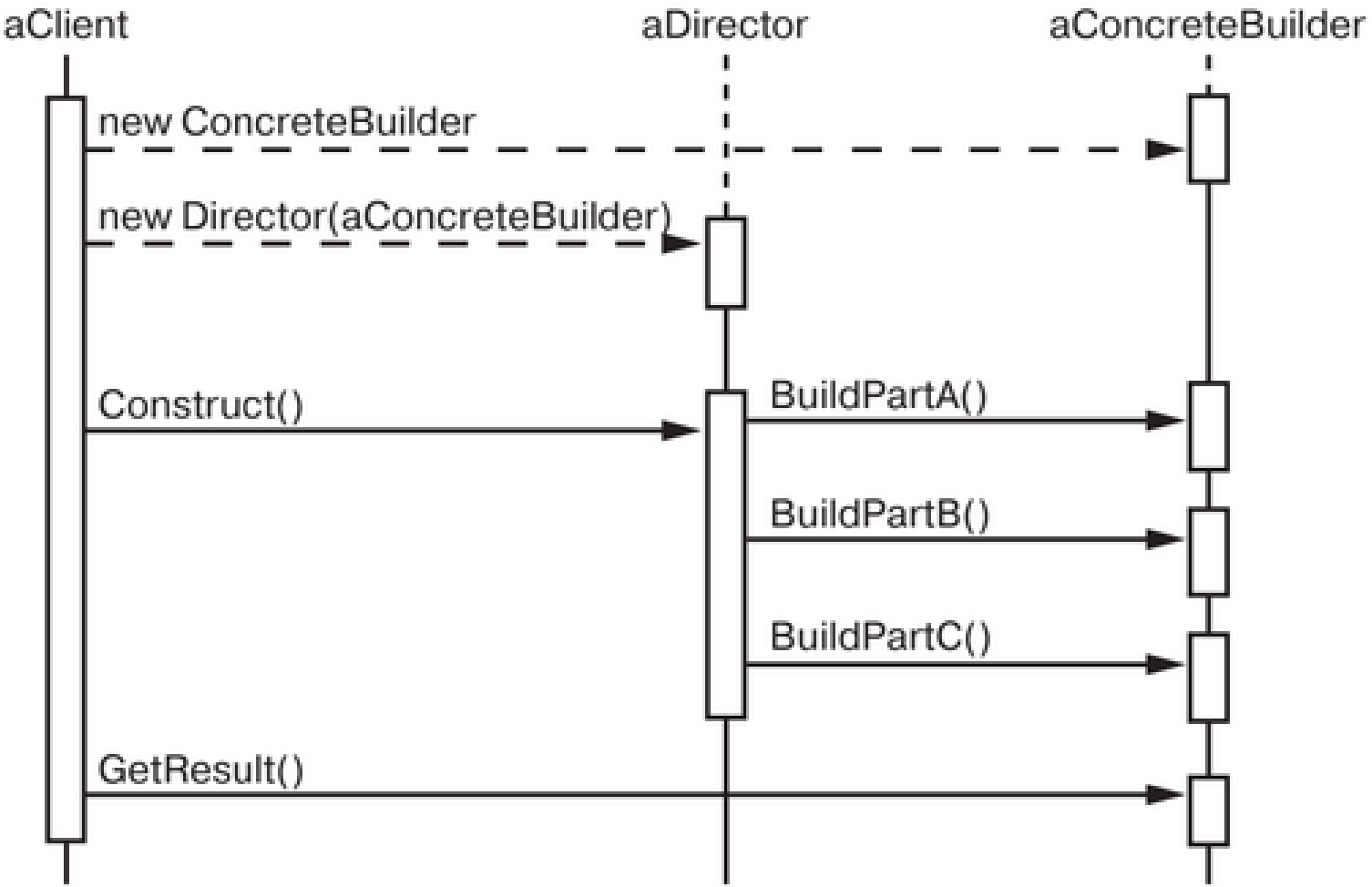
Строитель (Builder)

- Отделяет конструирование сложного объекта от его представления, т.ч. в результате одного процесса конструирования могут быть созданы различные представления.
  Проблемы:
  - алгоритм создания сложного объекта не должен зависить от того, из каких частей он состоит и как они стыкуются;
  - конструирование должно обеспечить различные представления объекта.

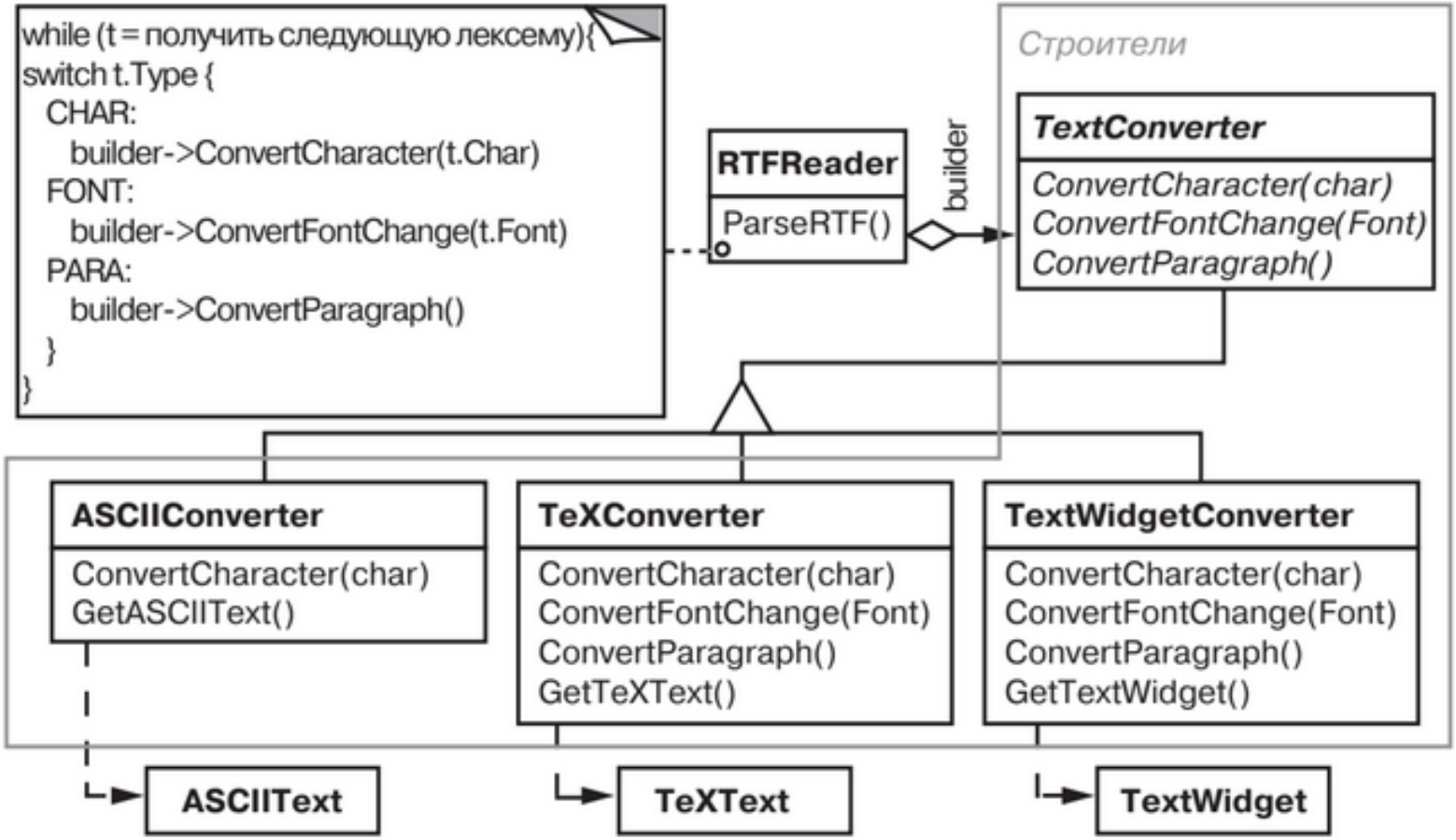
Строитель - решение



Строитель - решение



Строитель - пример



Строитель - результаты

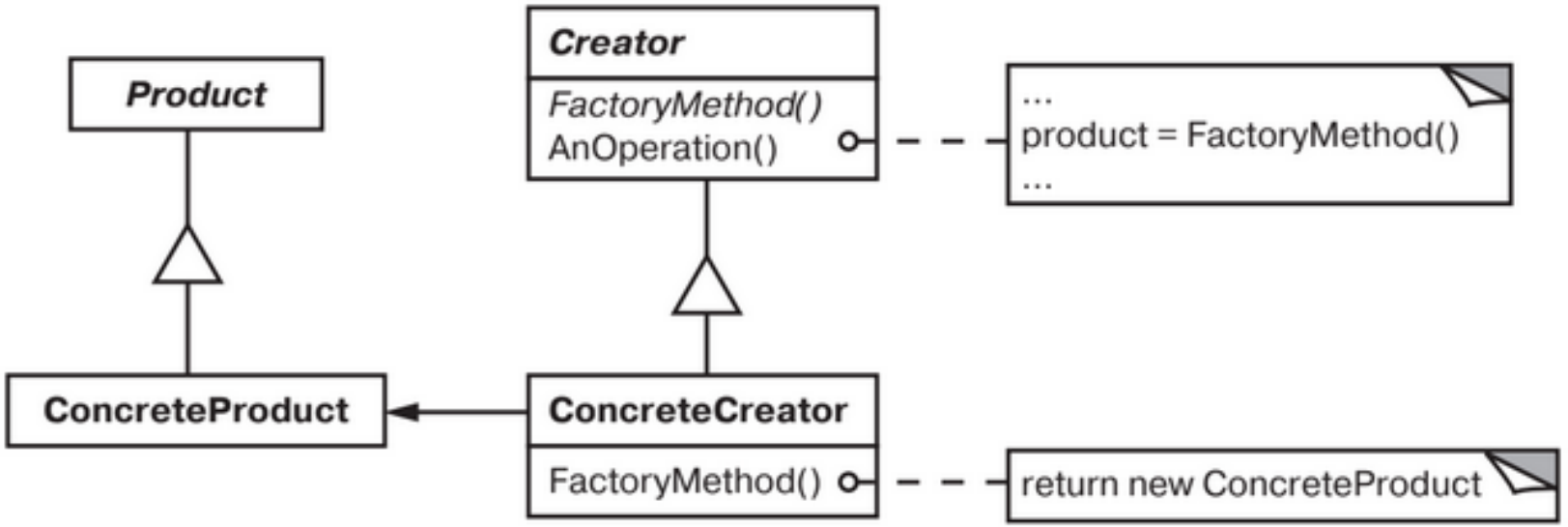
- Позволяет изменять внутреннее представление продукта (распорядитель не знает классы внутри продукта).
- Изолирует код, реализующий конструирование, и код, представление.
- Тонкий контроль за строительством (по шагам).

- Абстрактная фабрика -> сразу создается семейство, возвращает результат сразу.
- Строитель -> создается по шагам, возвращает продукт в конце.
- Часто строитель делает компоновщика.

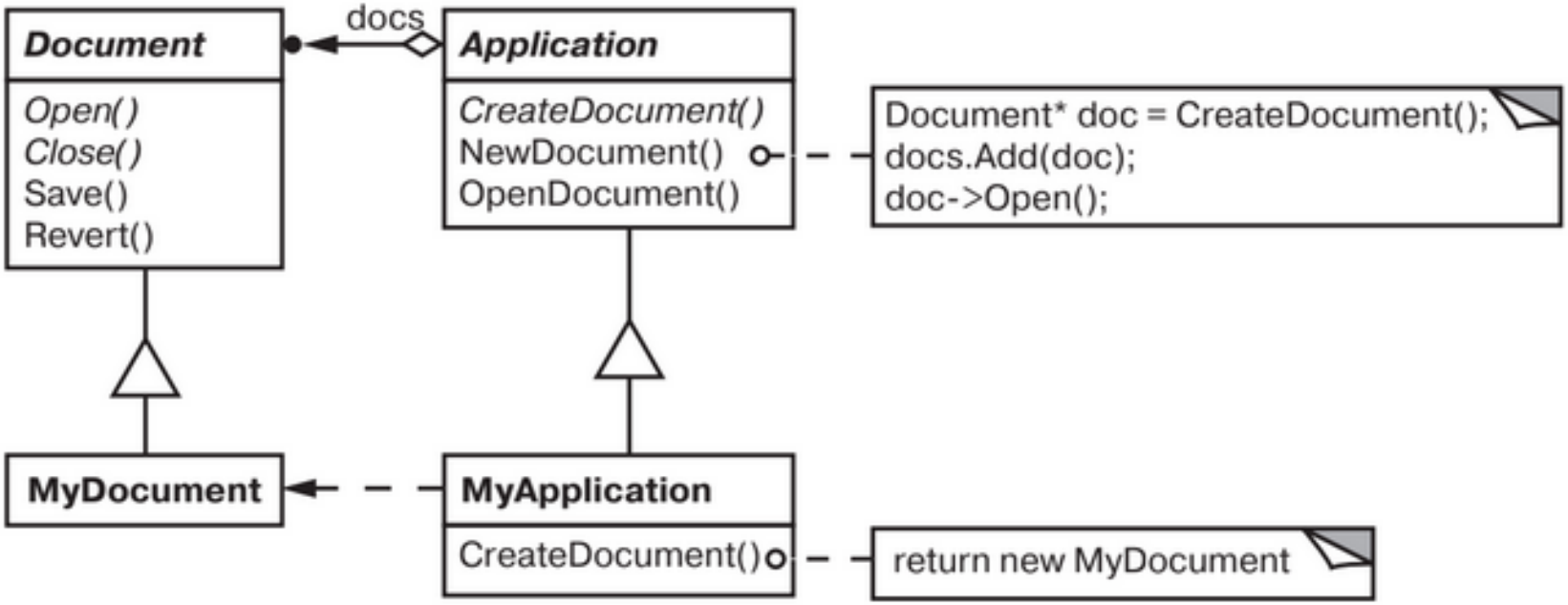
Фабричный метод (Factory Method)

- Определяет интерфейс для создания объекта; оставляет подклассам решение о том, какой класс инициализировать. Позволяет классу делегировать инстанцирование подклассам.
- Задача:
  - класс не знает о классе созданного объекта;
  - класс создан так, что создает объекты, а их класс определяется его подклассами.
- Применение:
  - Каркас с абстрактными классами (документы; приложения); создает объект, но не знает какого класса.

Фабричный метод – решение



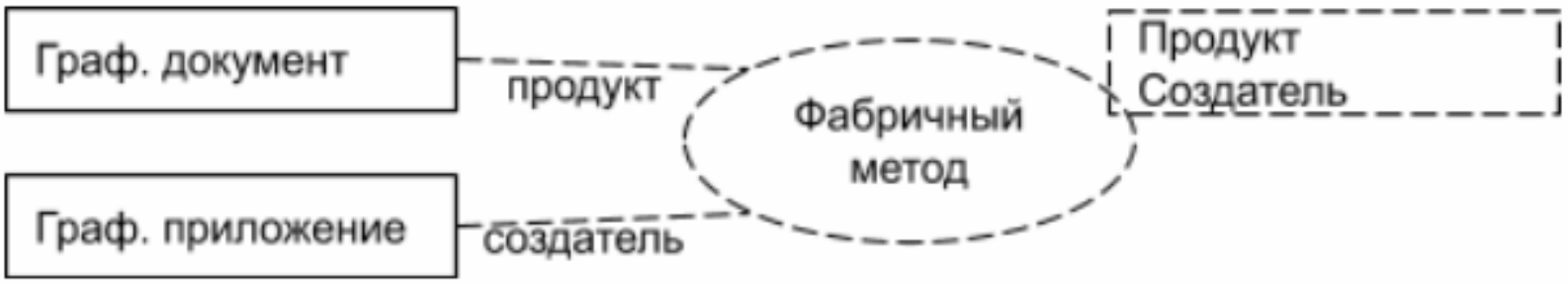
Фабричный метод – пример



Фабричный метод - результат

- (+) Не надо встраивать в код зависящие от приложения классы (только интерфейс) => легко добавить новые классы продукта.
- (+) Возможность создания параллельных иерархий классов (фигуры и манипуляторы ими)
- (-) Для добавления нового продукта нужен новый создатель.

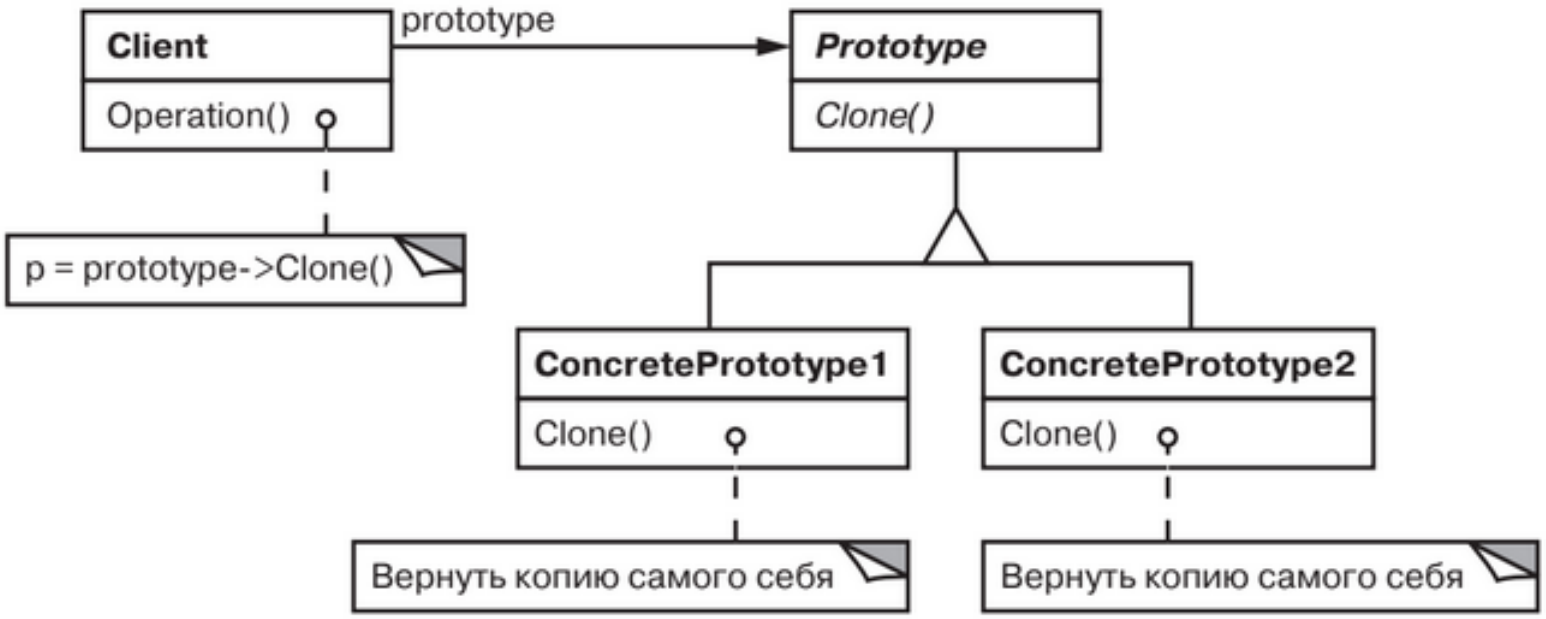
- Используется совместно с абстрактной фабрикой.
- В отличие от прототипов не требуется функция инициализации.



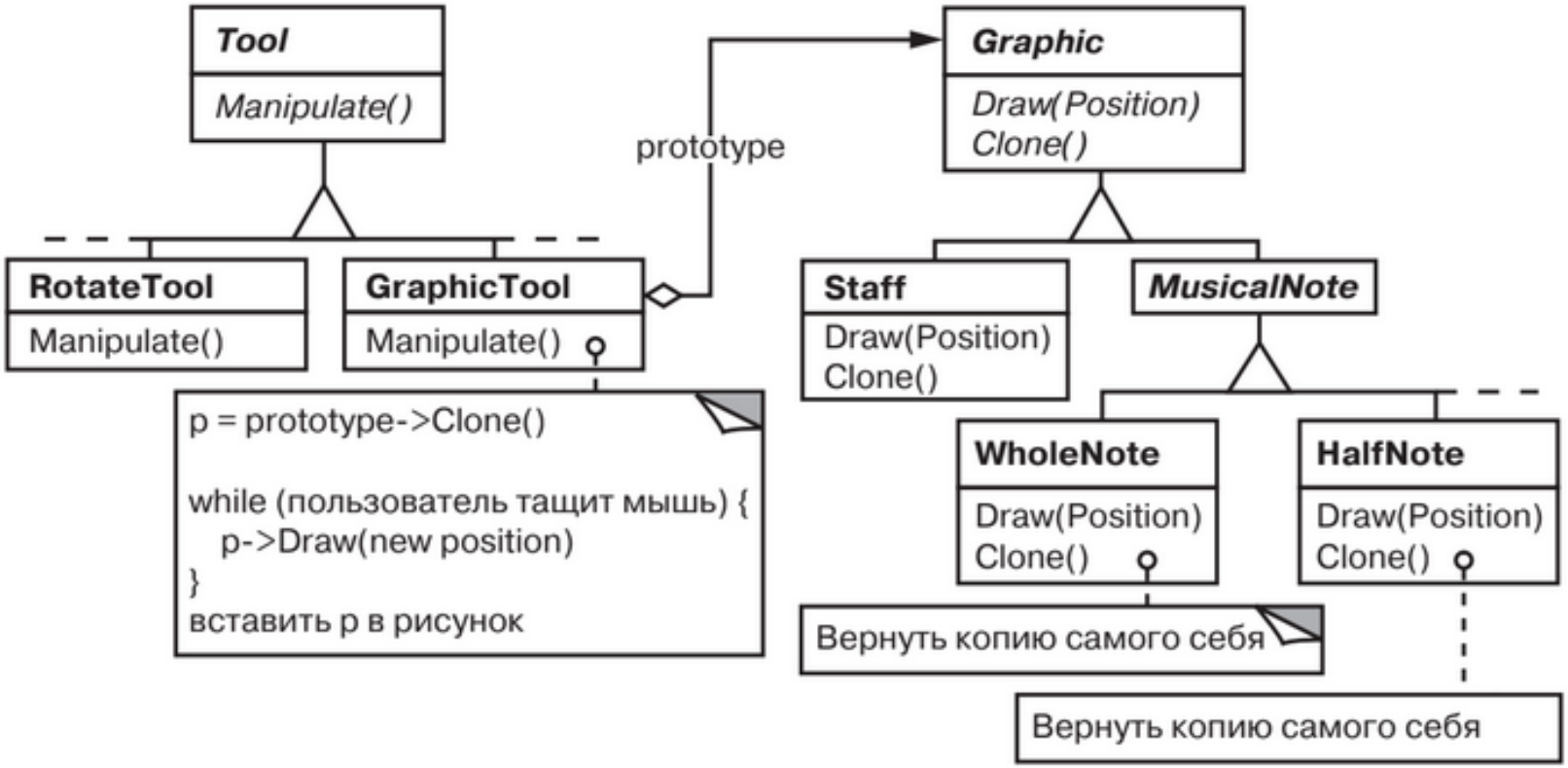
Прототип (Prototype)

- Задает виды созданных объектов с помощью экземпляра-прототипа и создает новые объекты путем копирования этого прототипа.
- Проблема:
  - инстанцированные классы задаются во время выполнения (динамическая загрузка);
  - без параллельной иерархии классов/фабрик для создания объектов;
  - экземпляр класса может быть в большом количестве состояний => удобнее клонировать, чем инстанцировать в определенное состояние.

Прототип - решение



Прототип - пример



Прототип - результат

- (+) Скрывает от клиента названия классов продукта => работает с различными классами без модификации кода.
- (+) Добавление/удаление продуктов во время выполнения.
- (+) Спецификация новых объектов путем изменения значений (клонирование вместо инстанцирования).
- (+) Спецификация новых объектов путем изменения структуры (составные объекты, глубокое копирование).
- (+) Уменьшение количества подклассов (без параллельной иерархии классов).
- (+) Динамическое конфигурирование приложения классами.
- (-) Любой подкласс должен реализовать операцию Clone() - это может быть сложно, если есть круговые ссылки.

- Может использоваться с абстрактной фабрикой, но вообще является ее конкурентом.

Одиночка (Singleton)

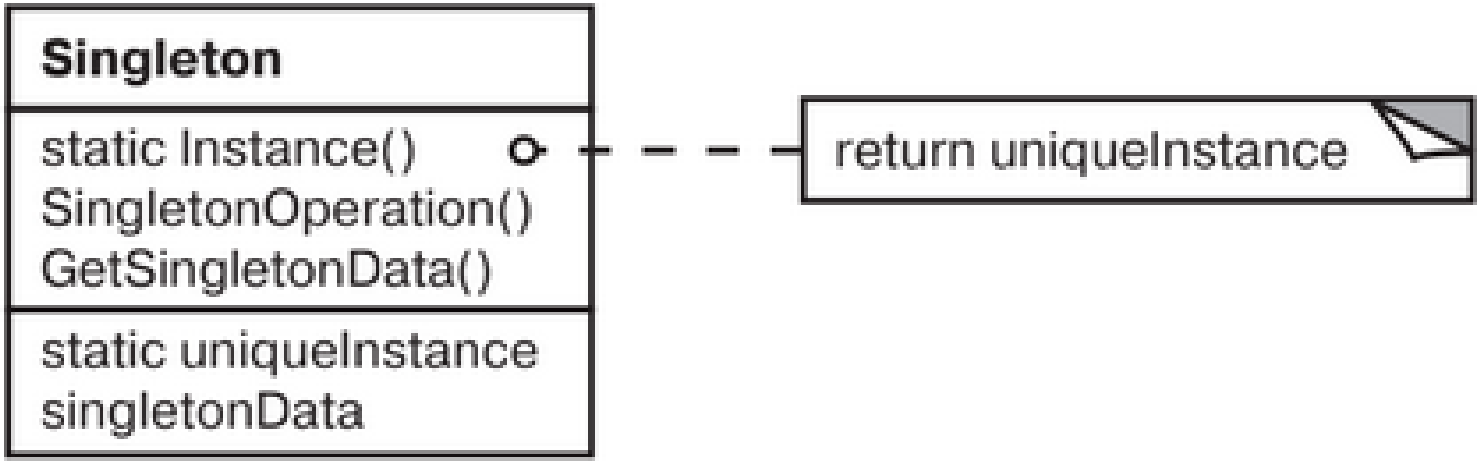
- Гарантирует, что у класса есть единственный экземпляр, и предоставляет глобальную точку доступа к этому экземпляру.
- Задача:
  - должен быть ровно один экземпляр некоторого класса, доступный всем;
  - единственный экземпляр должен расширяться путём порождения подклассов, и клиентам нужно иметь возможность работать с расширенным экземпляром без модификации своего кода.

Одиночка - код

```
class Singleton {
    public:
        static Singleton*
    Instance();
        void do();
    protected:
        Singleton();
    private:
        static Singleton* _instance;
};

Singleton* Singleton::_instance = 0;

Singleton* Singleton::Instance ()
{
    if (_instance == 0)
        _instance = new Singleton;

    return _instance;
}
```

```
void main()
{
    Singleton *s;
    s = Singleton::Instance();
    s->do();
    …

    Singleton *b;
    b = Singleton::Instance();
    b->do();
}
```

Одиночка – решение



Одиночка - результат

- Контролирует доступ к единственному экземпляру.
- Уменьшает количество имен (вместо глобальной переменной).
- Может быть более одного экземпляра без модификации приложения.
- Может быть расширение операций и представлений без модификации приложения.
- С помощью одиночки могут быть реализованы:
  - абстрактная фабрика
  - строитель
  - прототип

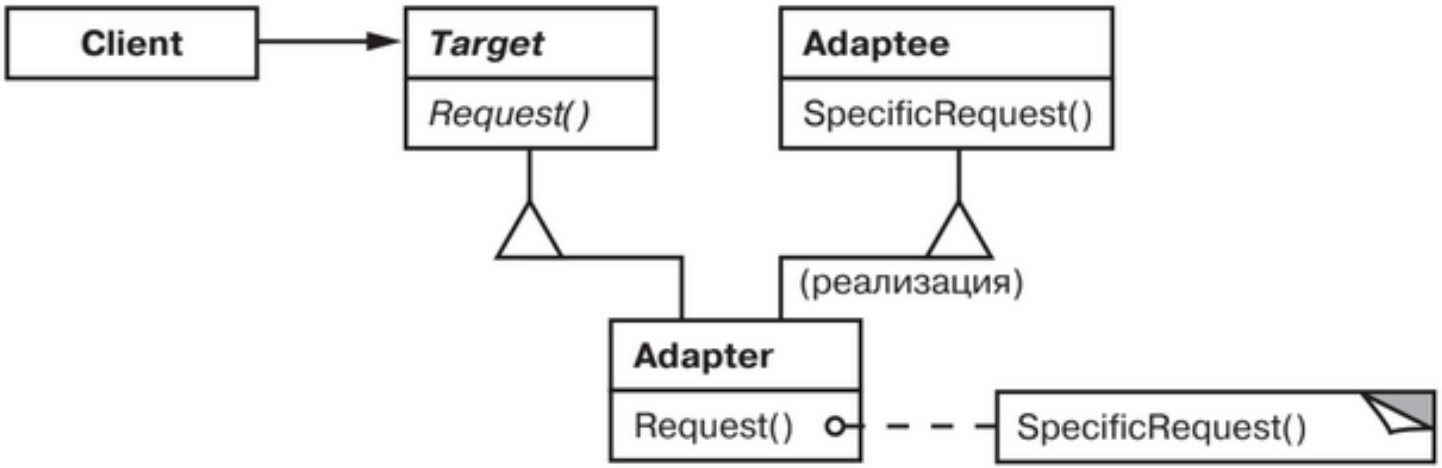
Структурные паттерны

- **Адаптер (Adapter)** - объект, обеспечивающий взаимодействие двух других объектов, один из которых использует, а другой предоставляет несовместимый с первым интерфейс.
- **Мост (Bridge) - о**тделяет абстракцию от реализации, благодаря чему появляется возможность независимо изменять то и другое.
- **Компоновщик (Composite)** - объект, который объединяет в себе объекты, подобные ему самому.
- **Декоратор (Decorator)** - класс, расширяющий функциональность другого класса без использования наследования.
- **Фасад (Facade)** - объект, который абстрагирует работу с несколькими классами, объединяя их в единое целое.
- **Приспособленец (Flyweight) -** использует разделение для эффективной поддержки большого числа мелких объектов.
- **Заместитель (Proxy)** - объект, который является посредником между двумя другими объектами, и который реализует/ограничивает доступ к объекту, к которому обращаются через него.

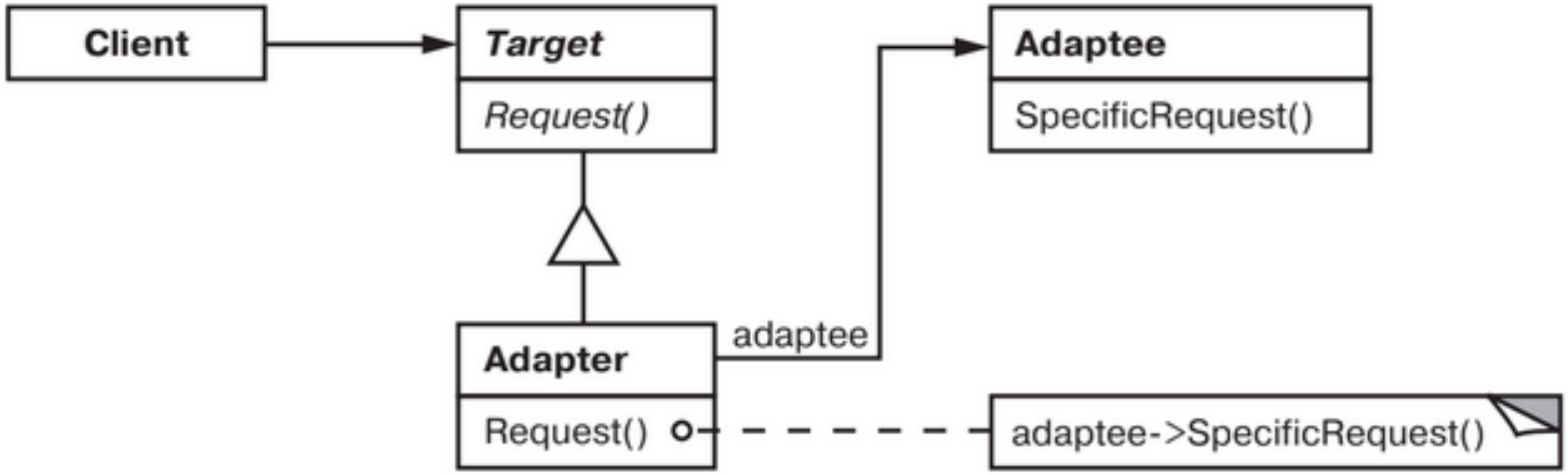
Адаптер (Adapter)

- Преобразует интерфейс одного класса в интерфейс другого, который ожидают клиенты. Обеспечивает работу классов с несовместимыми интерфейсами
- Проблема:
  - необходимо использовать существующий класс, интерфейс которого не соответствует потребностям;
  - создать повторно используемый класс для взаимодействия с заранее неизвестным классом, с другим интерфейсом;
  - (для адаптера объектов) необходимо использовать n подклассов, но неудобно порождать от них новые классы.

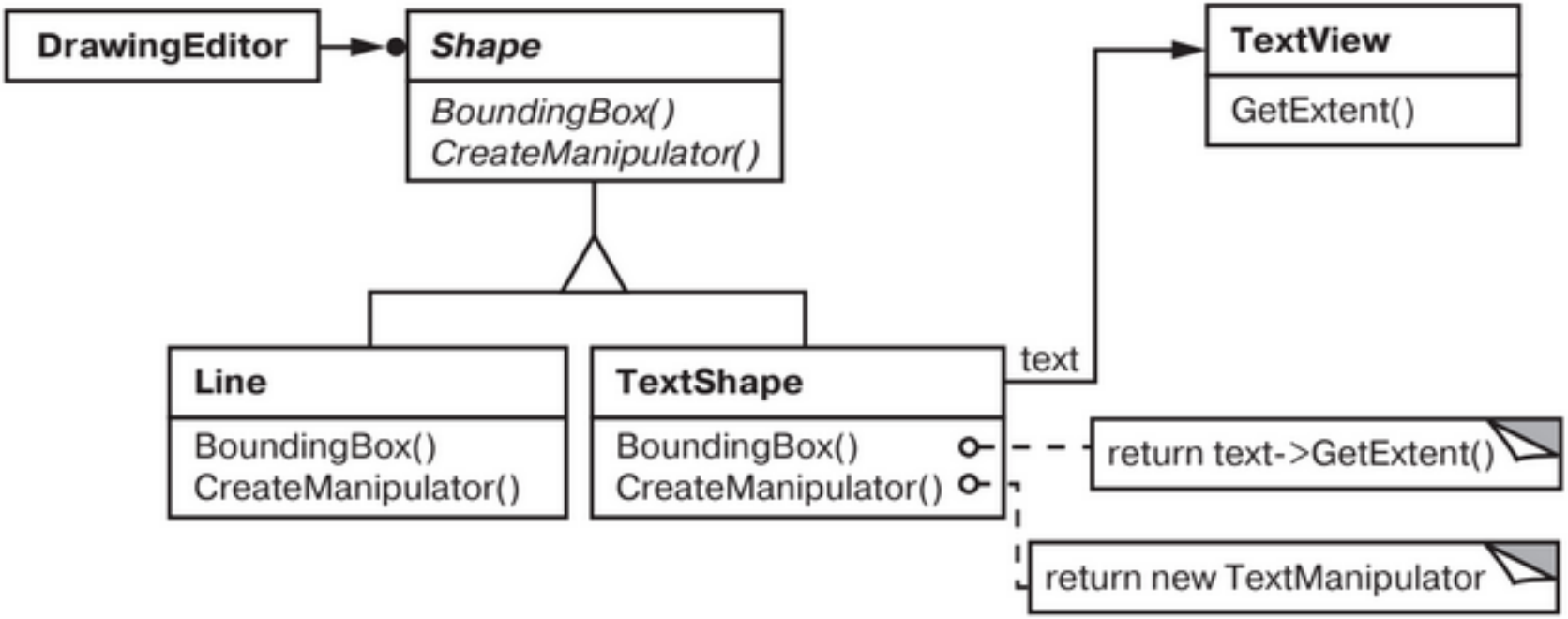
Адаптер класса - решение



Адаптер объекта - решение



Адаптер - пример



Адаптер - результат

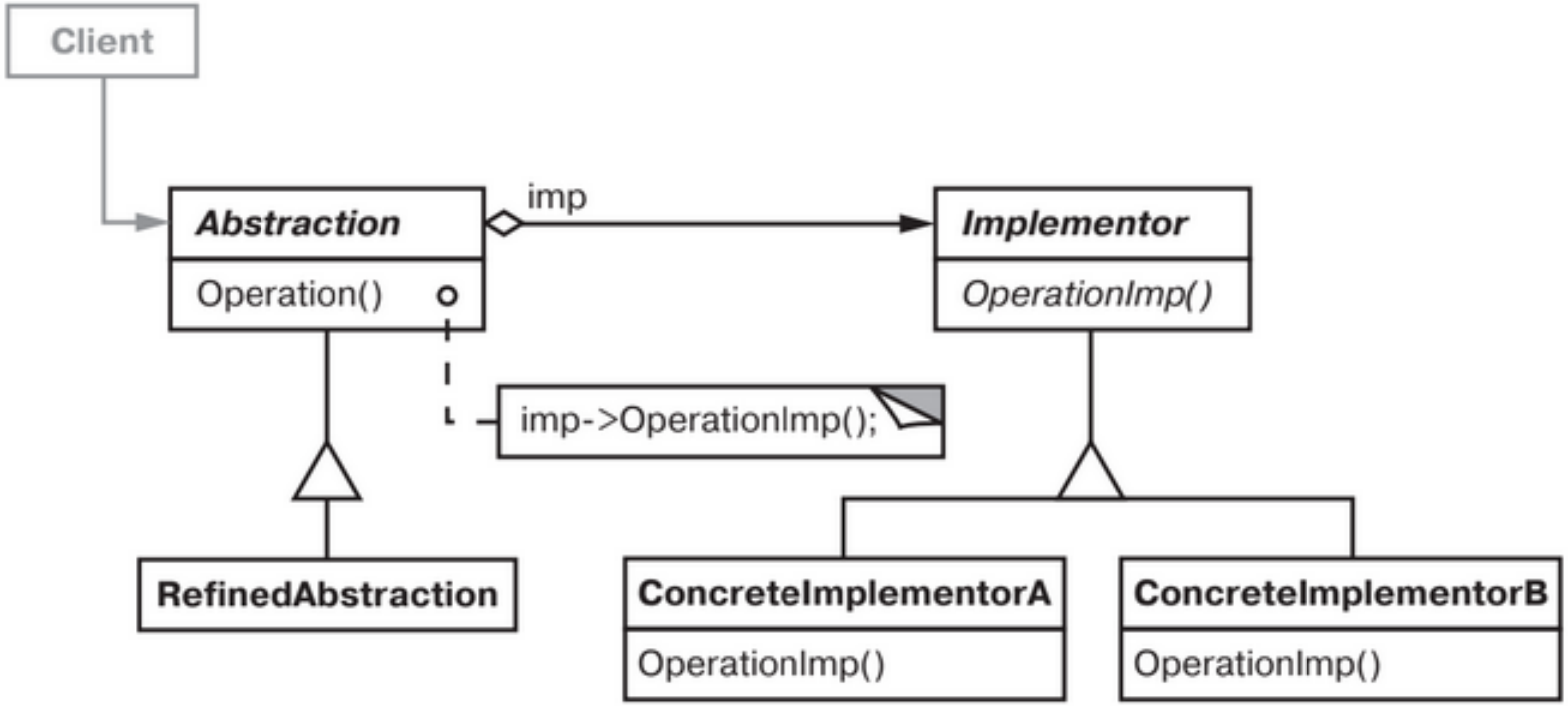
- Адаптер класса:
- (+) адаптер может изменять часть операций в целевом или адаптируемом;
- (-) не используется для адаптеров подклассов.

- Адаптер объекта:
- (+) если 1 адаптер и >1 адаптируемых, то можно при подключении адаптируемого добавить функциональность;
- (-) трудно заменить операции в адаптируемых.

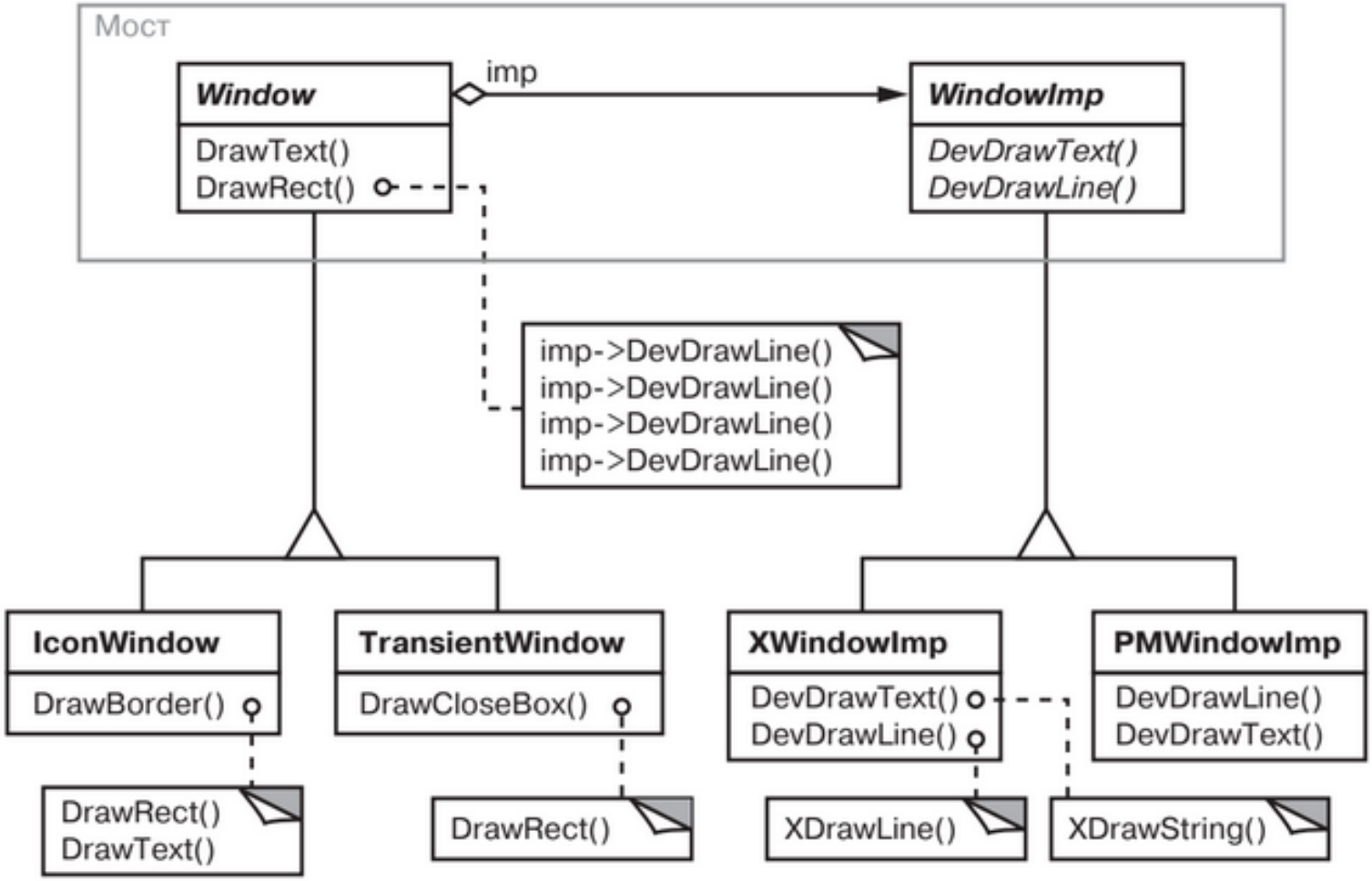
Мост (Bridge)

- Отделяет абстракцию от реализации так, что их можно изменять независимо.
- Проблема:
  - без постоянной привязки абстракции к реализации (изменение реализации во времени выполнения);
  - абстракция и реализация должны быть расширены подклассами;
  - изменение абстракции ≠> изменение реализации;
  - сокрытие от класса интерфейса реализации;
  - скрытое разделение реализации среди n объектов.

Мост - решение



Мост - пример


Мост
Window
DrawText()
DrawRect()
imp
WindowImp
DevDrawText()
DevDrawLine()
imp->DevDrawLine()
imp->DevDrawLine()
imp->DevDrawLine()
imp->DevDrawLine()
IconWindow
DrawBorder()
TransientWindow
DrawCloseBox()
XWindowImp
DevDrawText()
DevDrawLine()
PMWindowImp
DevDrawLine()
DevDrawText()
DrawRect()
DrawText()
DrawRect()
XDrawLine()
XDrawString()

Мост - результат

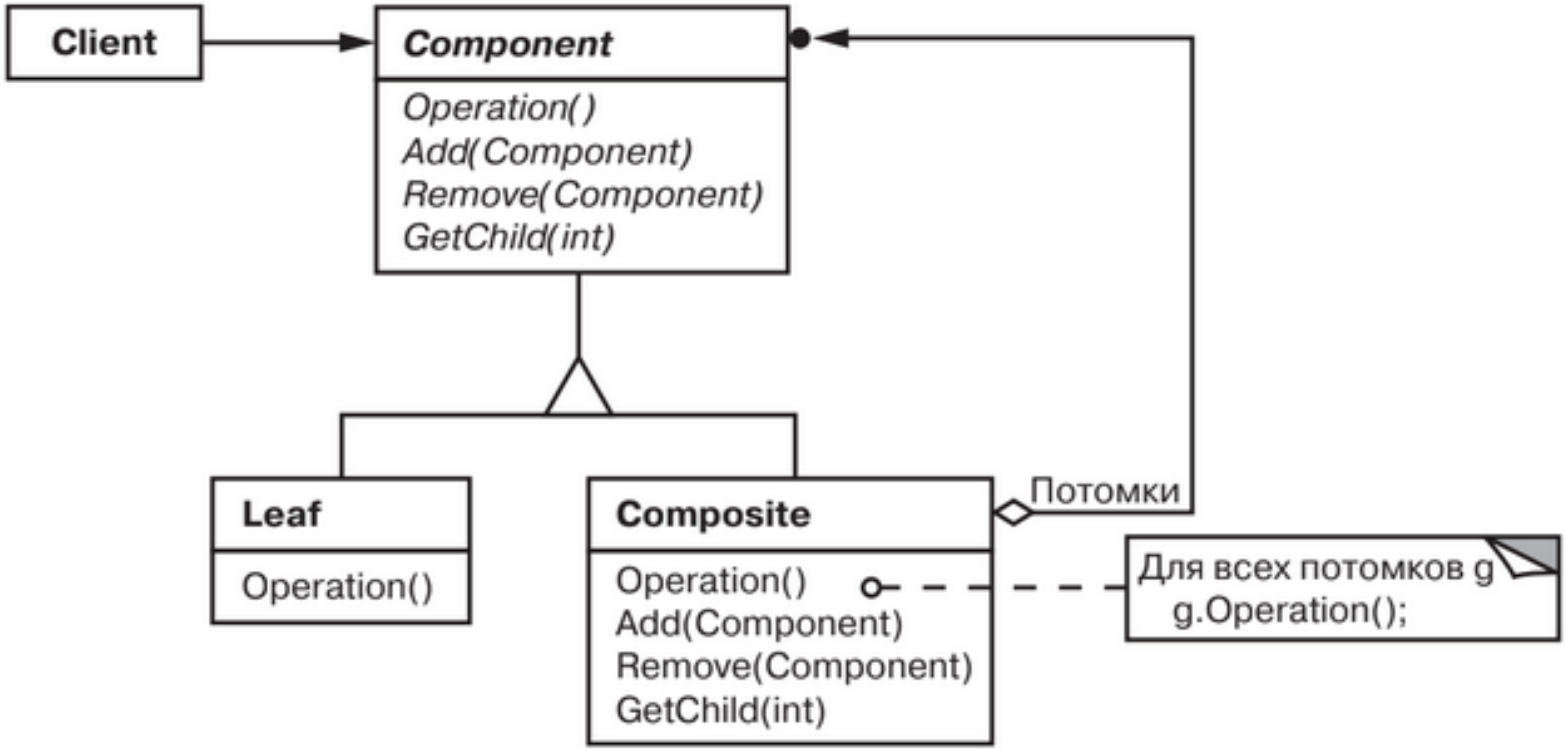
- Отделяет абстракцию и реализацию.
- Увеличивает степень расширяемости.
- Сокрытие деталей реализации от клиента.

- Мост создается изначально.
- Адаптер соединяет существующие.
- Для создания/конфигурации моста используется абстрактная фабрика.

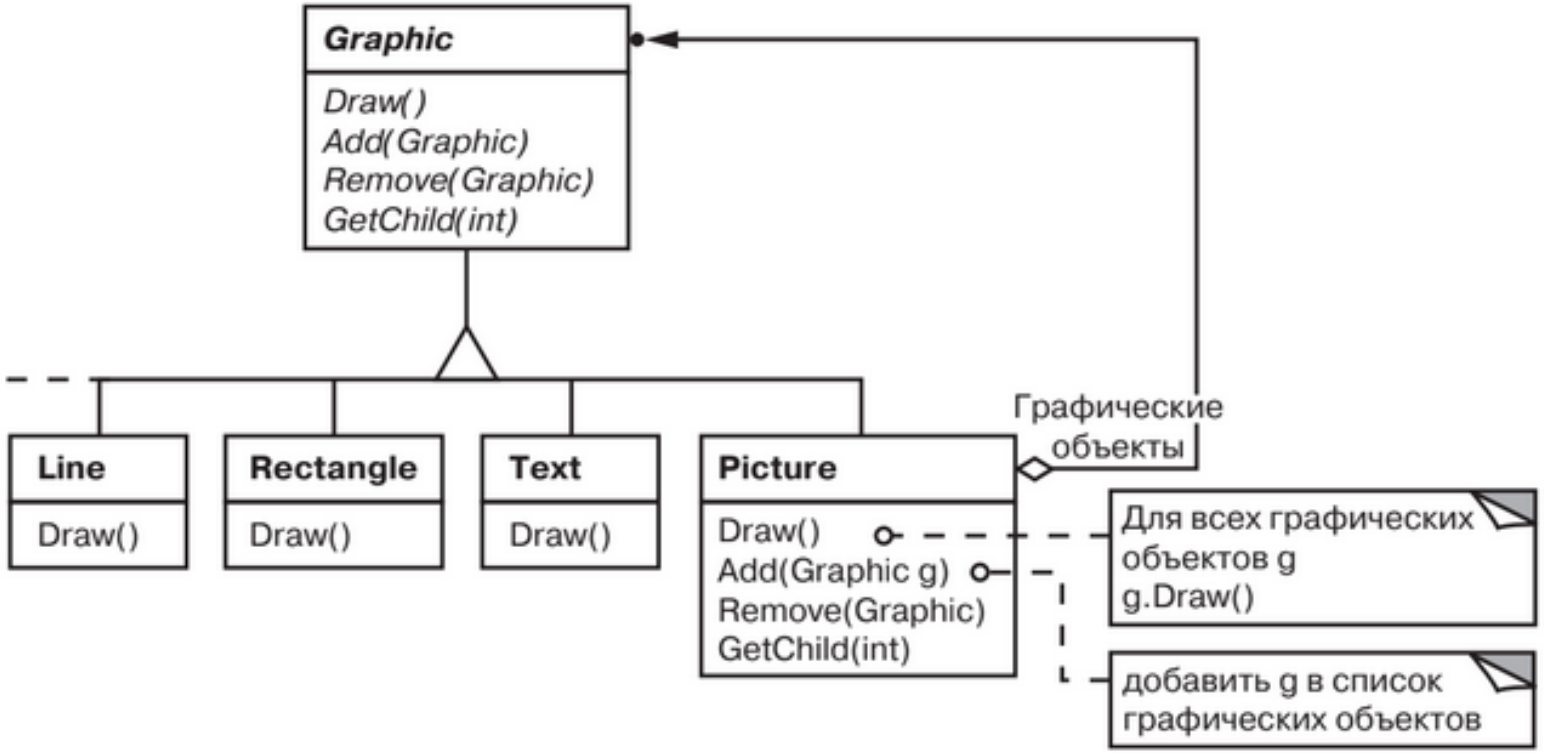
Компоновщик (Composite)

- Обеспечивает представление иерархий часть/целое, объединяя объекты в древовидные структуры
- Проблема:
  - объединение маленьких компонентов в крупные (контейнеры), объединение компонентов в большие и в огромные;
  - клиент по-разному работает с этими объединениями.
- Применяется:
  - построение иерархии целое-часть;
  - унификация использования составных/ индивидуальных объектов.

Компоновщик - решение



Компоновщик - пример



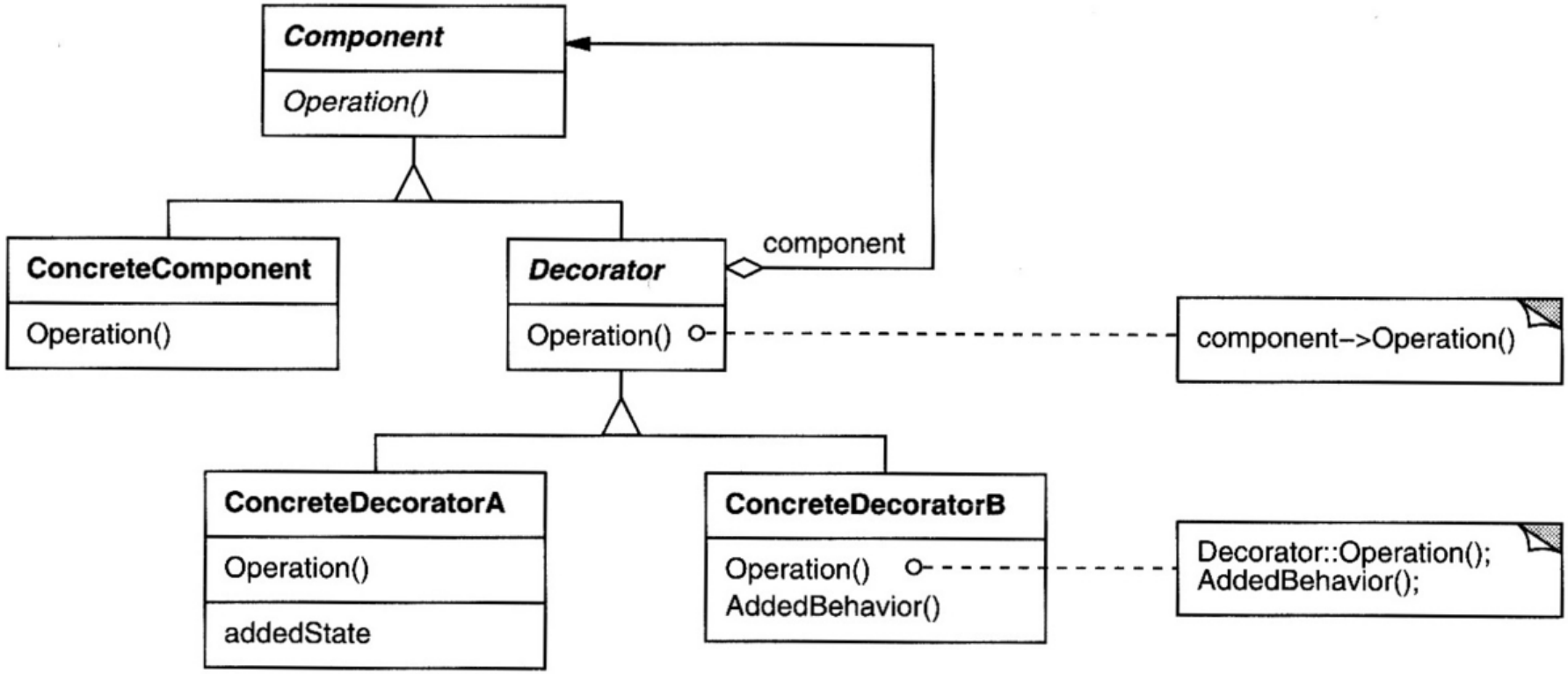
Компоновщик - результат

- (+) иерархия из примитивов и контейнеров;
- (+) облегчает добавление новых разновидностей компонентов;
- (+) упрощает организацию клиента (унифицирует обращения);
- (-) сложно накладывать ограничения на объекты в композиции.

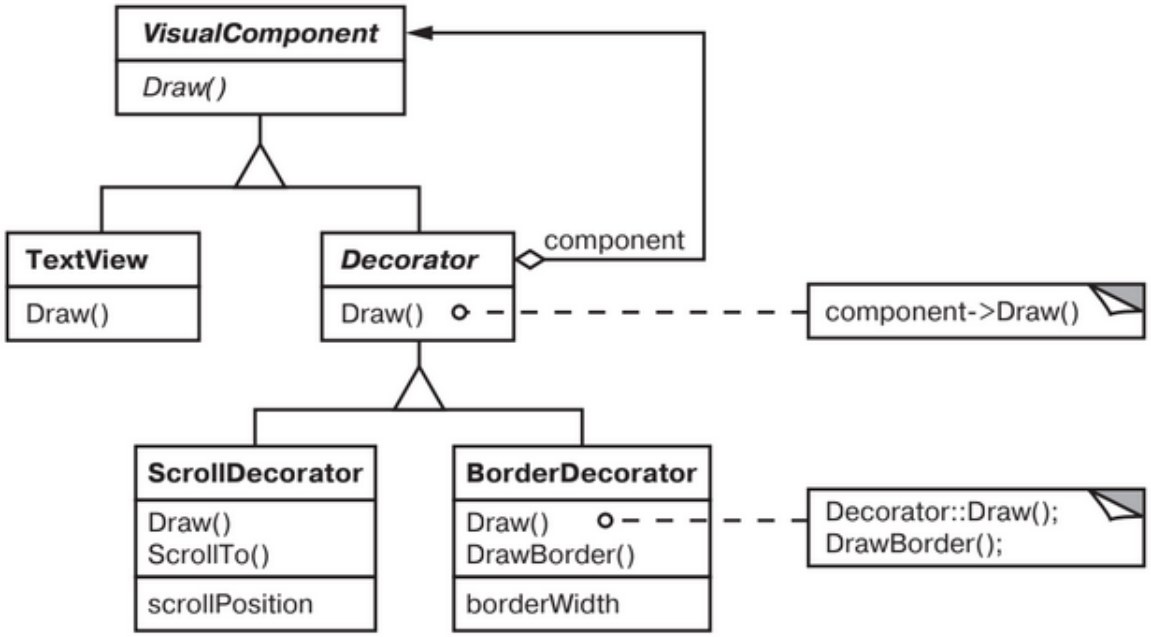
Декоратор (Decorator)

- Динамически добавляет объекту новые обязанности. Является гибкой альтернативой порождению подклассов для расширения функциональности.
- Проблема:
  - динамическое/прозрачное для класса добавление функций объектам;
  - реализация функций, которые могут быть сняты с клиента;
  - для избежания иерархии классов (очень много).

Декоратор - решение



Декоратор - пример



Декоратор - результат

- (+) увеличение гибкости по сравнению с наследованием (может быть 1 свойство > 1 раза);
- (+) без перегруженных функциями классов на верхнем уровне иерархии классов;
- (+) декоратор ≠ его компоненты;

- (-) много мелких объектов (сложно проектировать/ тестировать/ разбирать).

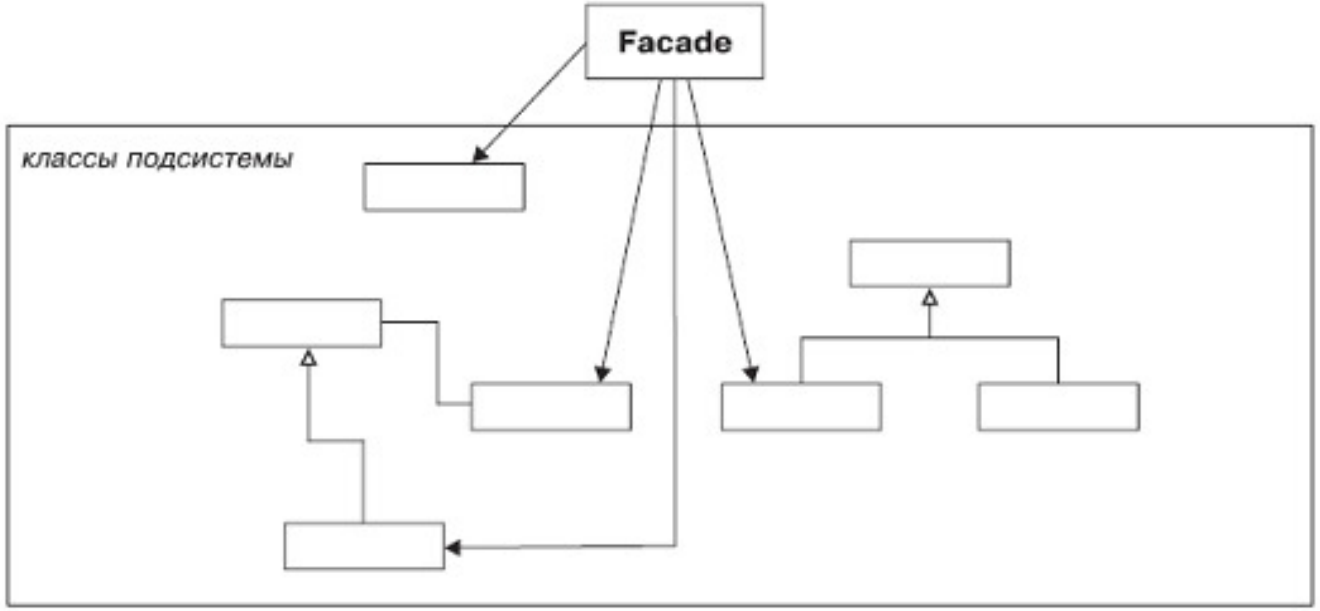
- Компоновщик - строит иерархию.
- Декоратор - новые обязанности без подклассов.
- Для декоратора компоновщик - это компонент.
- Для компоновщика декоратор - это лист.

Фасад (Facade)

- Представляет унифицированный интерфейс вместо набора интерфейсов некоторой подсистемы. Определяет интерфейс более высокого уровня, который упрощает исполнение подсистемы.
- Проблема:
  - простой интерфейс к сложной подсистеме (сокрытие деталей; вид системы по умолчанию с возможностью обратиться прямо);
  - увеличение независимости и переносимости (отдельные подсистемы и классы);
  - разделение подсистем на слои. Фасад для входа на любой уровень.

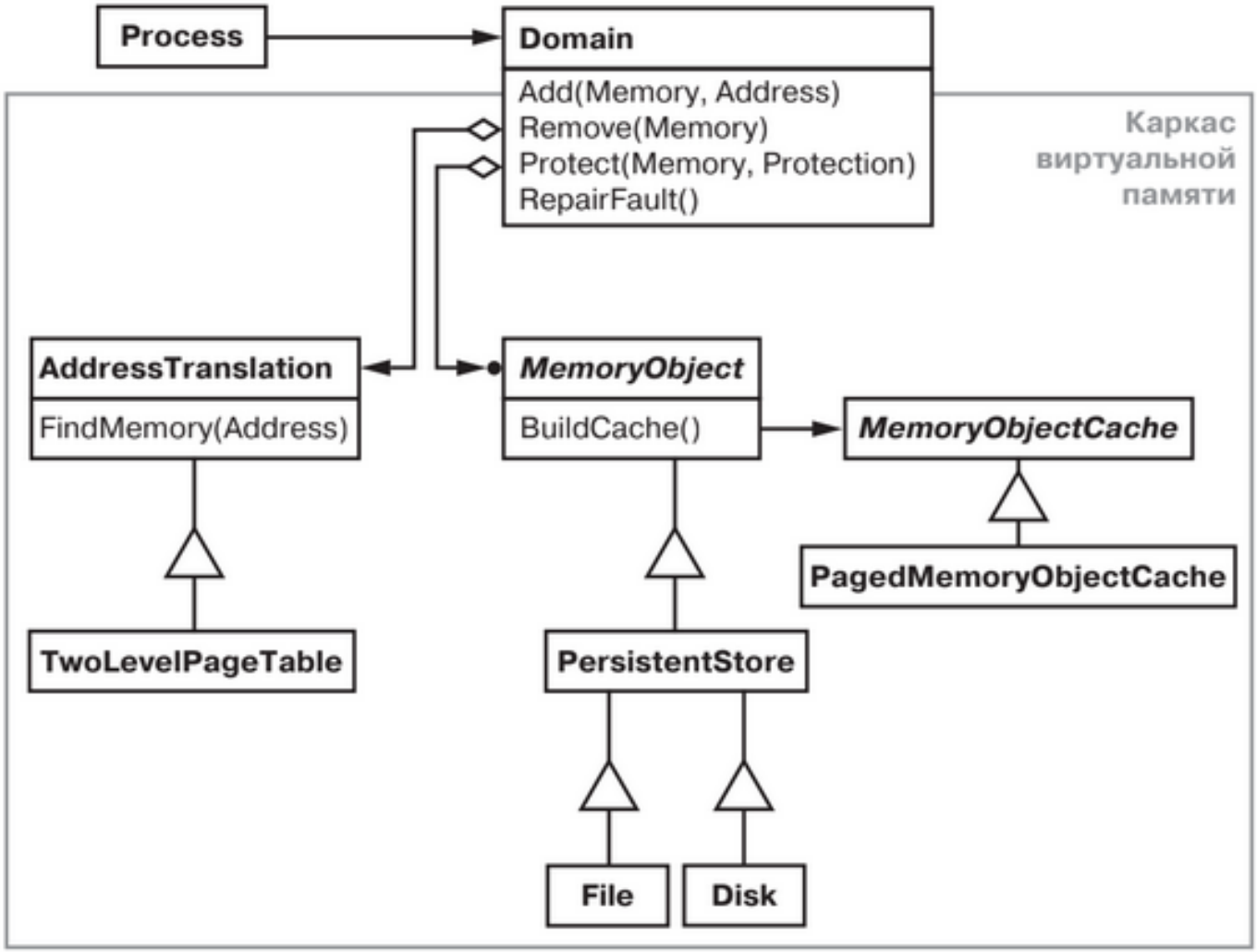
Фасад - идея

Фасад - решение



- Фасад - знает класс подсистемы, делегирует запрос клиента классам подсистемы.
- Класс подсистемы: реализует функциональность подсистемы, не знает о фасаде.

Фасад - пример



Фасад - результат

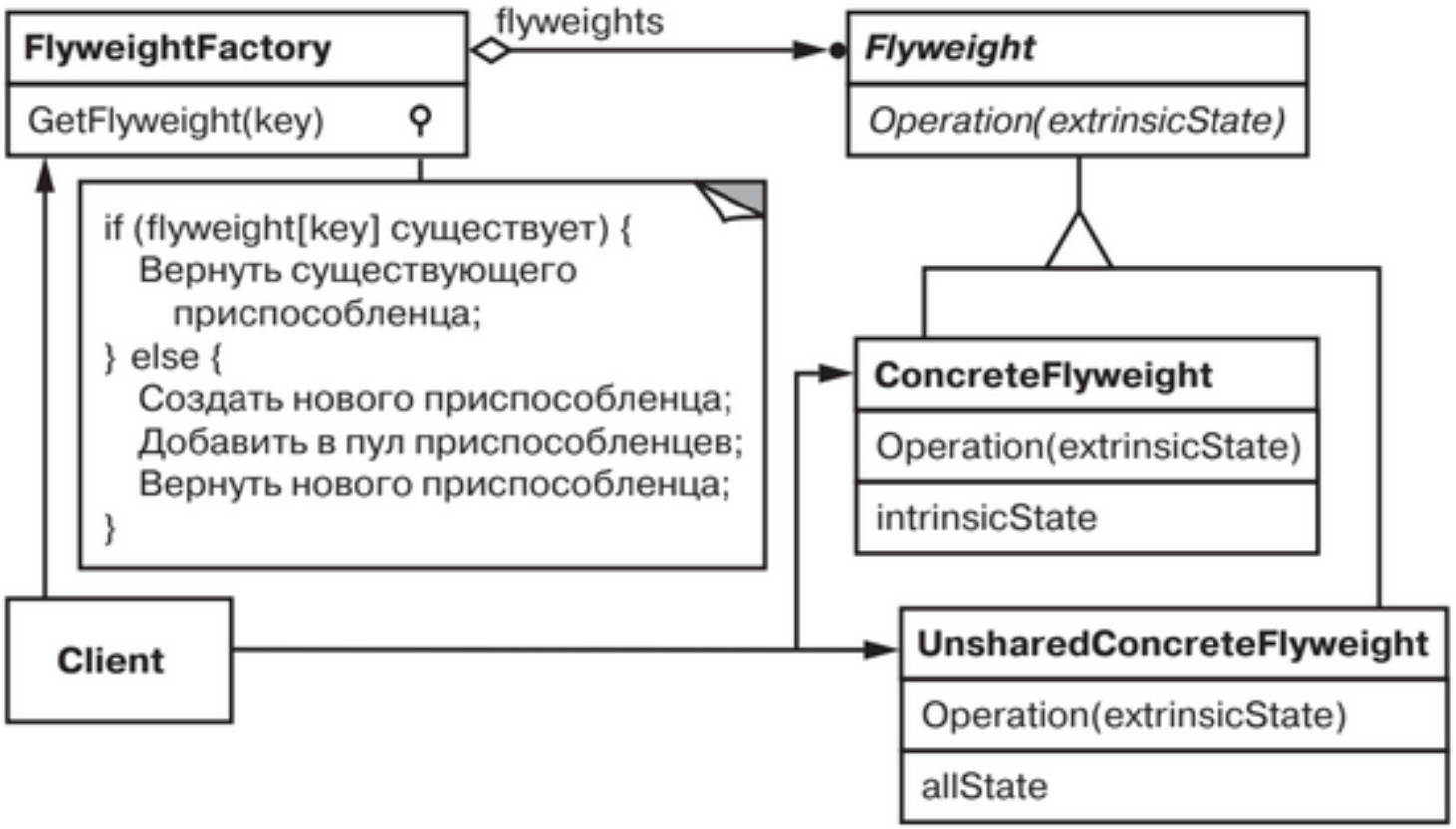
- Разделение классов и подсистем => уменьшение сцепления подсистем.
- Упрощение переноса на другую платформу.
- Может быть прямое обращение к классам подсистемы.

- Фасад - альтернатива абстрактной фабрики.
- Фасад может быть создан Абстрактной фабрикой (создание и исполнение элементов подсистемы).
- Фасад может быть одиночкой.

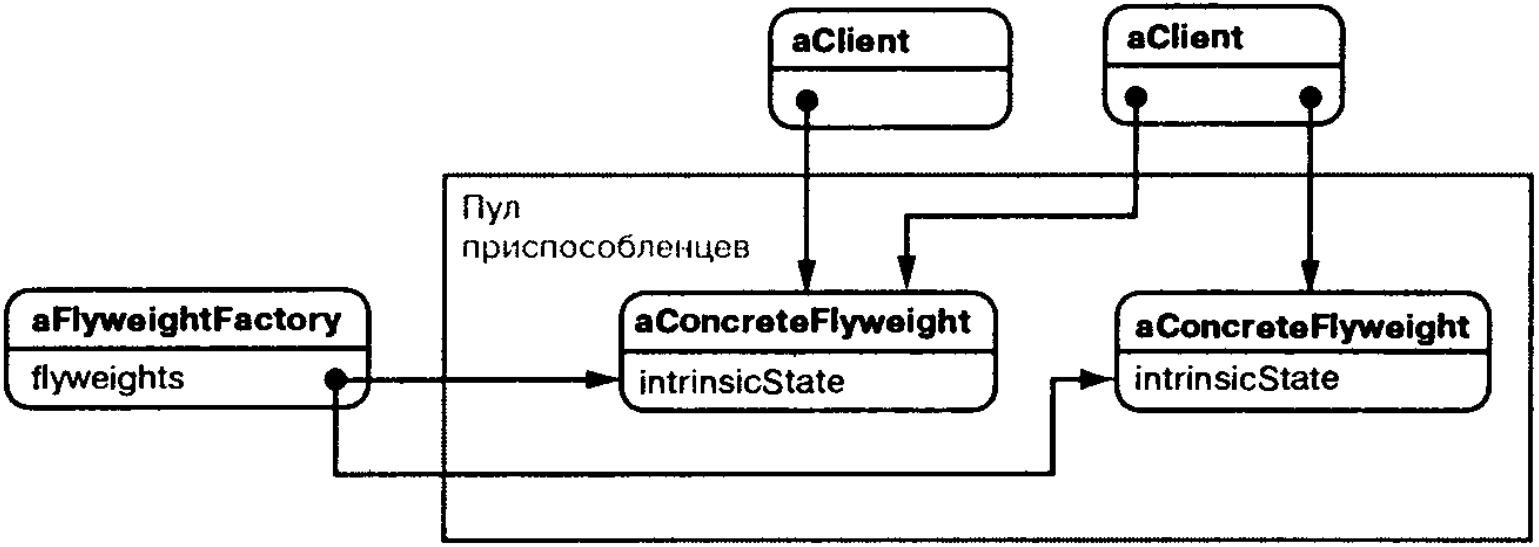
Приспособленец (Flyweight)

- Использует разделение для поддержки множества мелких объектов.
- Проблема:
  - много объектов => увеличение накладных расходов на хранение;
  - большую часть составных объектов можно вынести вовне => группу объектов можно заменить на разделяемый объект;
  - приложение не зависит от идентичности объектов.

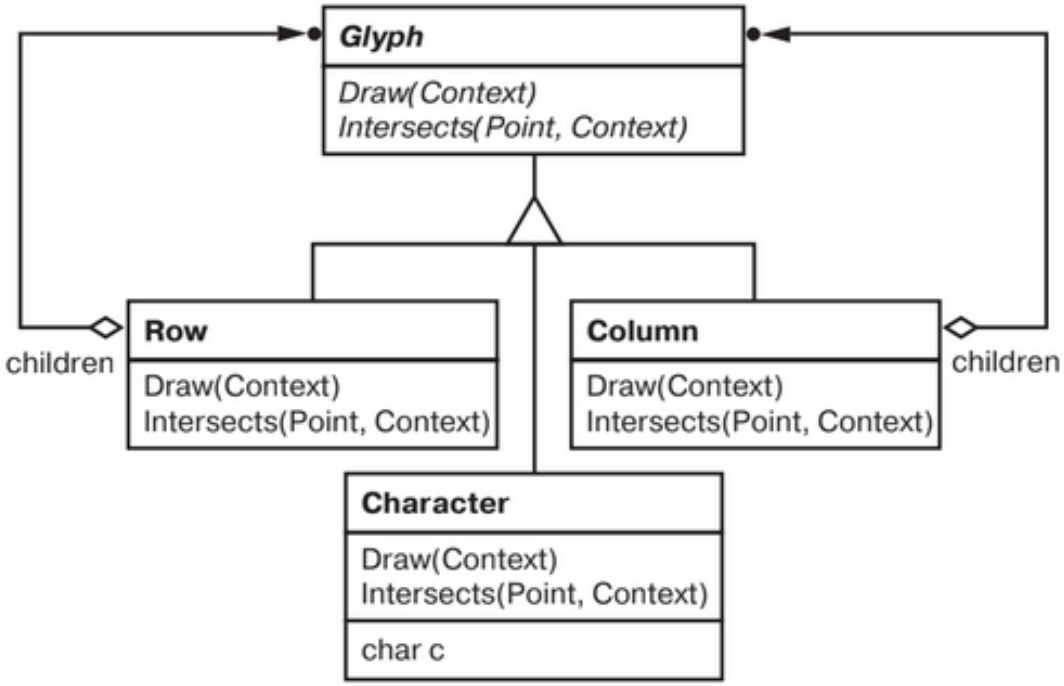
Приспособленец - решение

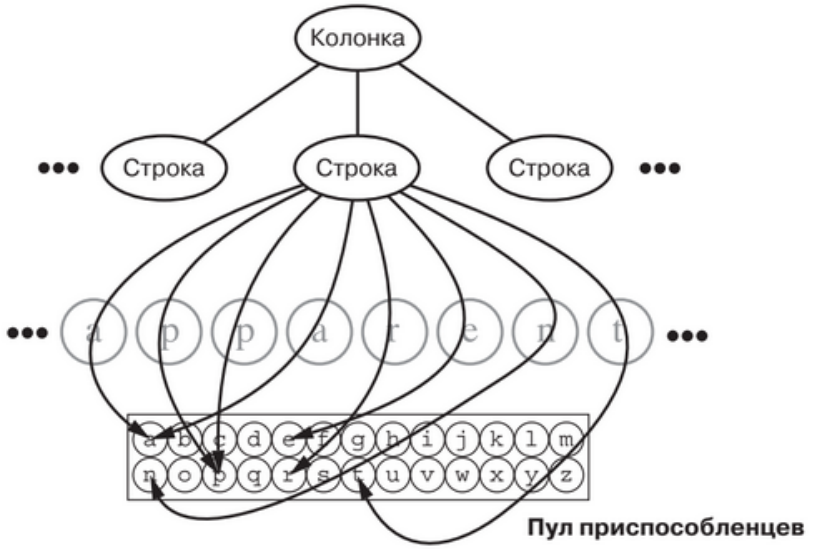


Приспособленец - решение



Приспособленец - пример





Приспособленец - результат

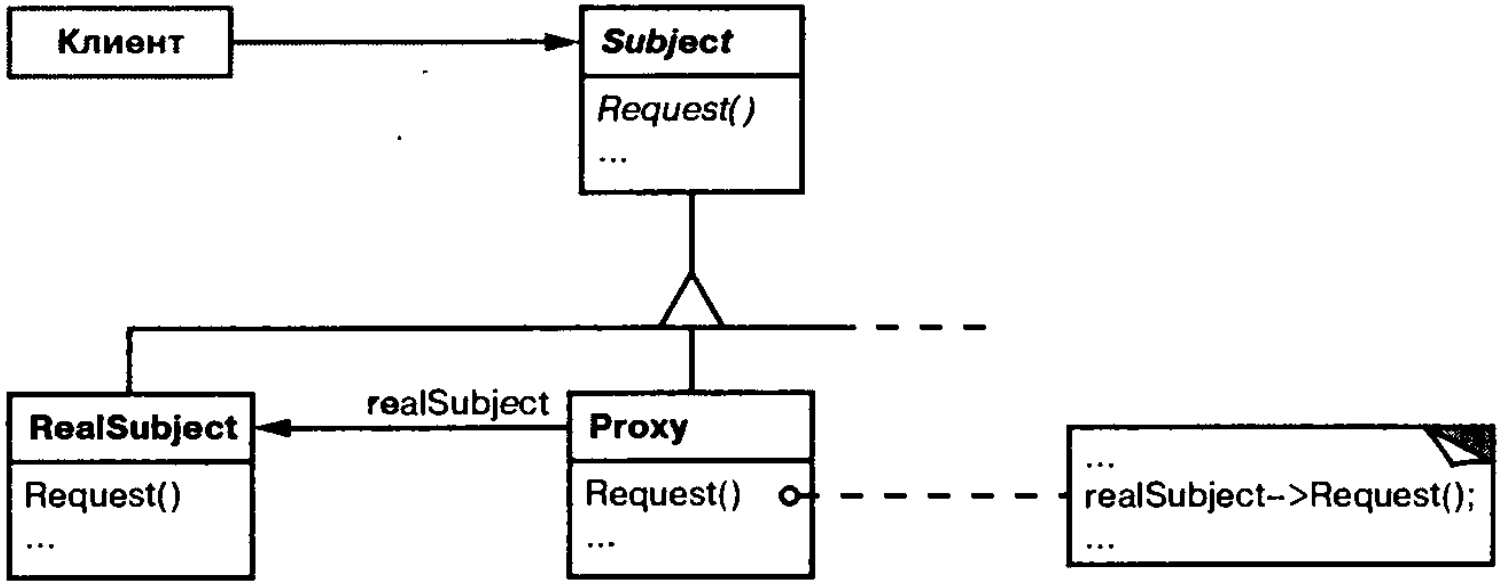
- Увеличение затрат на передачу, вычисление внутренних составляющих.
- Уменьшение количества экземпляров.
- Уменьшение ресурсов (ОП).
- Вычисление вместо хранения.

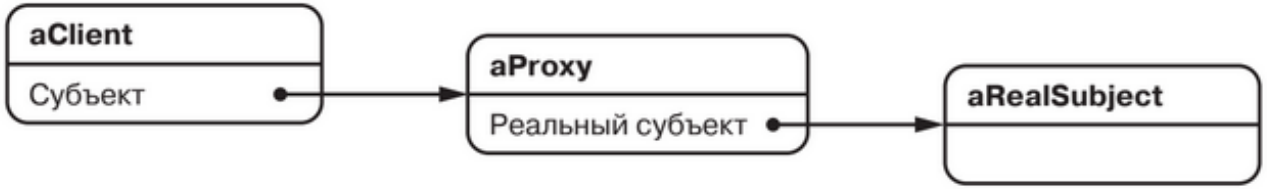
- Используется совместно с компоновщиком для создания ациклического орграфа с разделяемыми листьями.

Заместитель (Proxy)

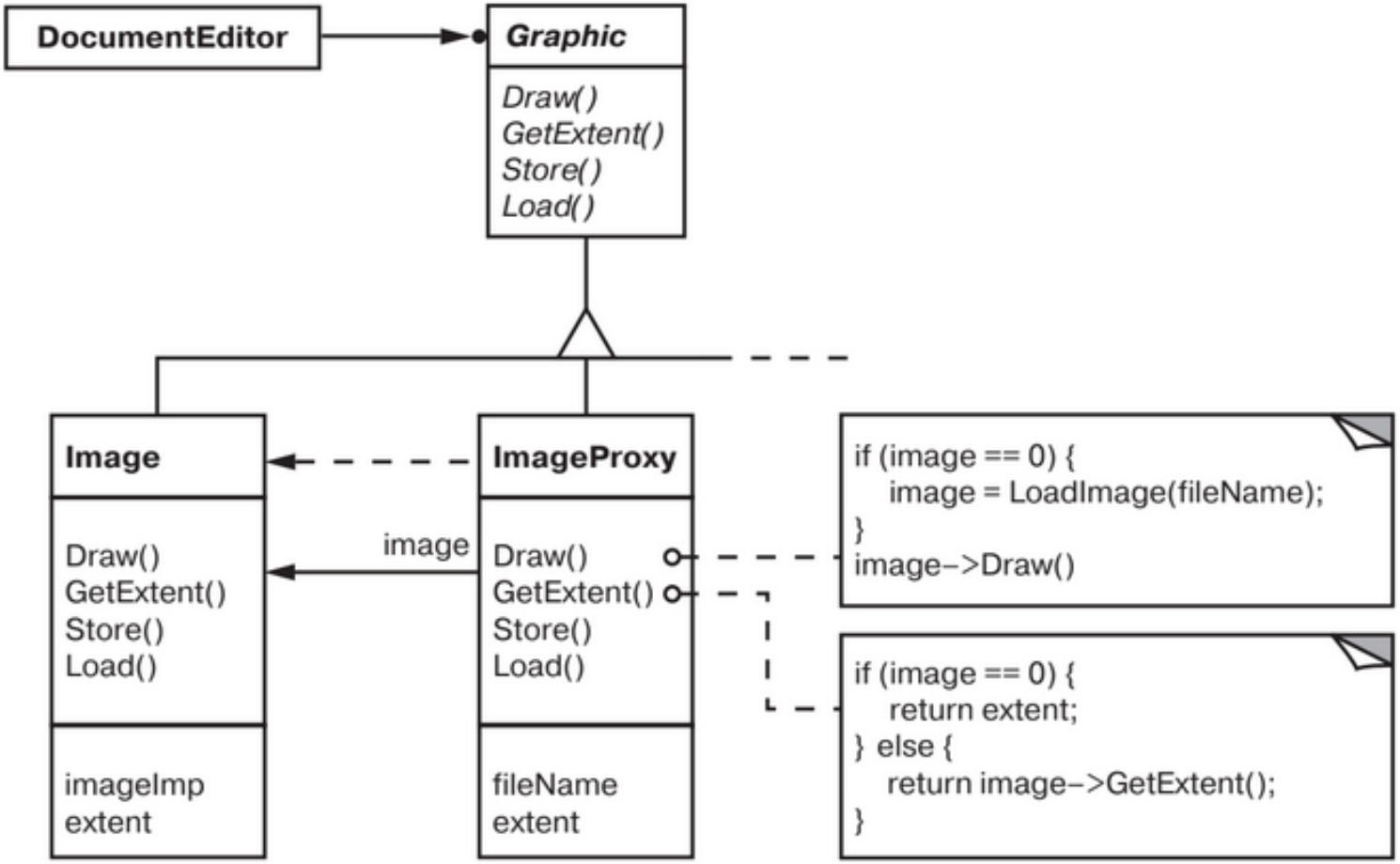
- Является суррогатом другого объекта и контролирует доступ к нему.
- Проблема:
  - удаленный заместитель (в другом адресном пространстве);
  - виртуальный заместитель (загрузка объектов по требованию);
  - защищающий заместитель (права доступа);
  - “умная ссылка” (дополнительные действия).

Заместитель - решение





Заместитель - пример



Заместитель - результат

- Удаленный заместитель (сокрытие расположения).
- Виртуальный заместитель (оптимизация времени/ОП).
- Дополнительные задачи при доступе к объекту (счетчик ссылок запроса в ОП; блокировка ресурсов системы).
- Заместитель создает “виртуальную” ссылку на ресурс системы (или узла).

- Декоратор - часть функциональности, которая изменчива (динамическая рекурсия).
- Заместитель - разрешает/запрещает доступ к функциональности субъекта (статическая связь).