Проектирование архитектур программного обеспечения

лекция 3

Зозуля А.В.

КСПТ

2016 г.

Ранее..

- Элементы модели: объекты, отношения, службы, модули
- Расслоение системы
- Типовые решения организации бизнес-логики

Содержание

- Паттерны проектирования
- Порождающие, структурные и поведенческие паттерны
- Антипаттерны

Типовые решения проектирования Design Patterns

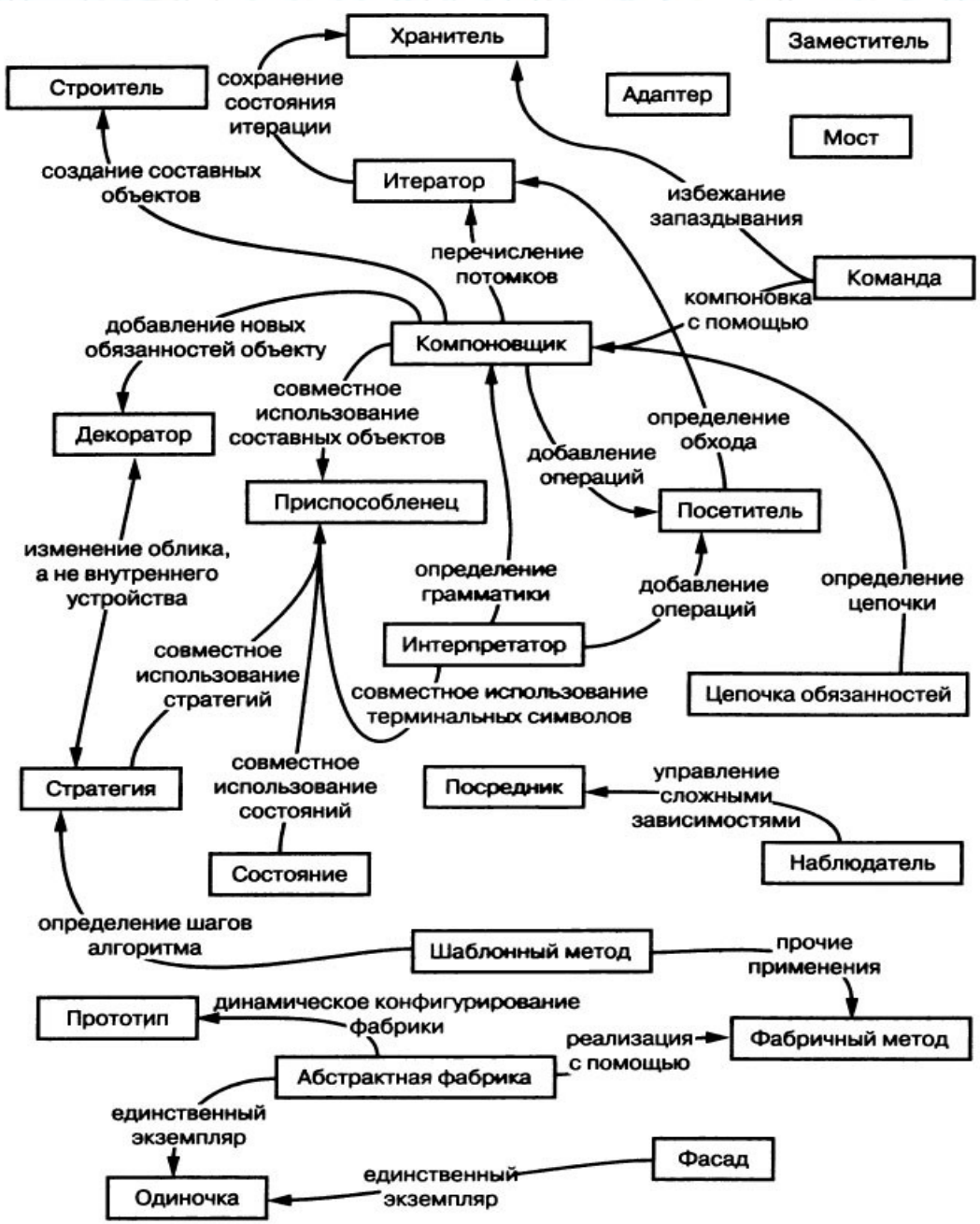
- Различают:
  - Архитектурные паттерны
  - Аналитические паттерны
  - **Паттерны проектирования**
  - Анти-паттерны
- Паттерны проектирования:
  - Применяются во всех слоях
  - «Кирпичики» архитектуры
  - Общепринятый каталог
  - Облегчают сопровождение

Классификация паттернов проектирования

Иллюстрации: Приемы объектно-ориентированного проектирования. Э. Гамма, Р. Хелм,..

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

Отношения между паттернами



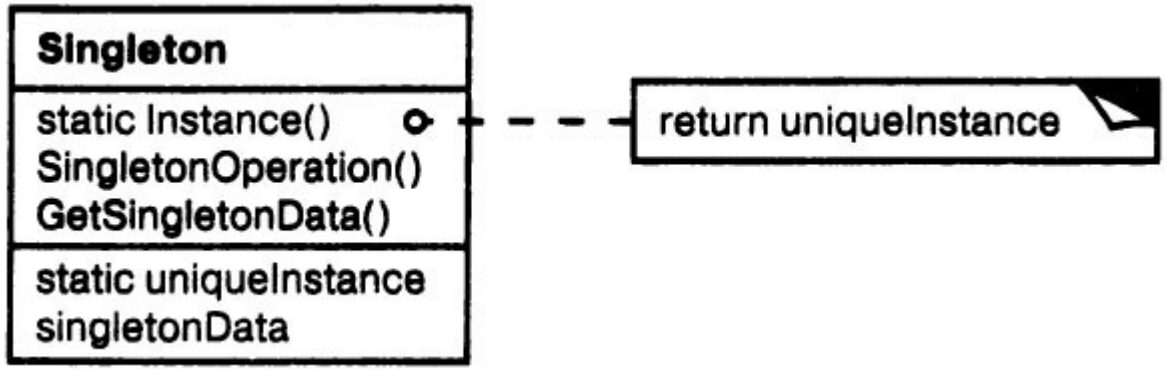
Порождающие паттерны

- Абстрагируют процесс инстанциирования
- Инкапсулируют знания о конкретных классах
- Скрывают детали создания и композиции классов
  - Одиночка
  - Абстрактная фабрика
  - Строитель
  - Фабричный метод

Паттерн Одиночка (Singleton)

- Паттерн, порождающий объекты
- Гарантирует, что у класса есть только один экземпляр
- Предоставляет к экземпляру глобальную точку доступа
- Глобальная переменная — не подходит
- Единственный экземпляр должен расширяться путем порождения подклассов

Паттерн Одиночка



- Одиночка — определяет операцию Instance(), которая позволяет клиентам получать доступ к единственному экземпляру
- Instance() - статический метод класса
- Одиночка несет ответственность за создание собственного уникального экземпляра

Паттерн Одиночка

```
class CSingleton
{
public:
    static CSingleton* GetInstance (void);
private:
    static CSingleton* m_instance;
protected:
    CSingleton(){};
};

CSingleton* CSingleton::m_instance = NULL;

CSingleton* CSingleton::GetInstance() {
    if (!m_instance)
        m_instance = new CSingleton;
    return m_instance;
}
```

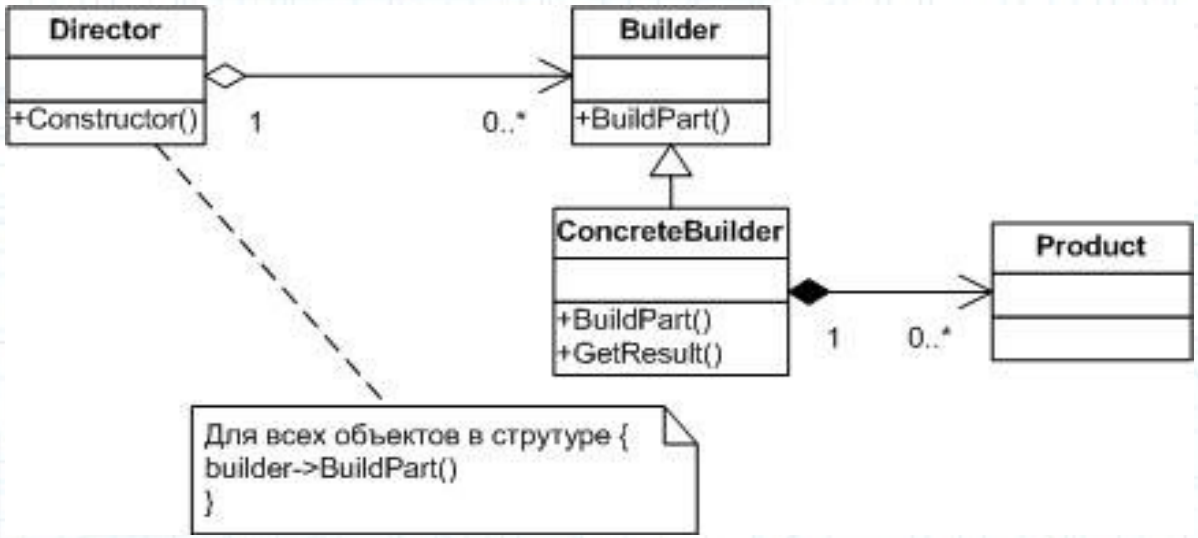
Паттерн Одиночка

- Преимущества:
  - Контролируемый доступ к единственному экземпляру
  - Допускает уточнение операций и представления
  - Допускает переменное число экземпляров
- Недостатки:
  - Стоит хорошо подумать перед «заточением» объекта

Паттерн Строитель (Builder)

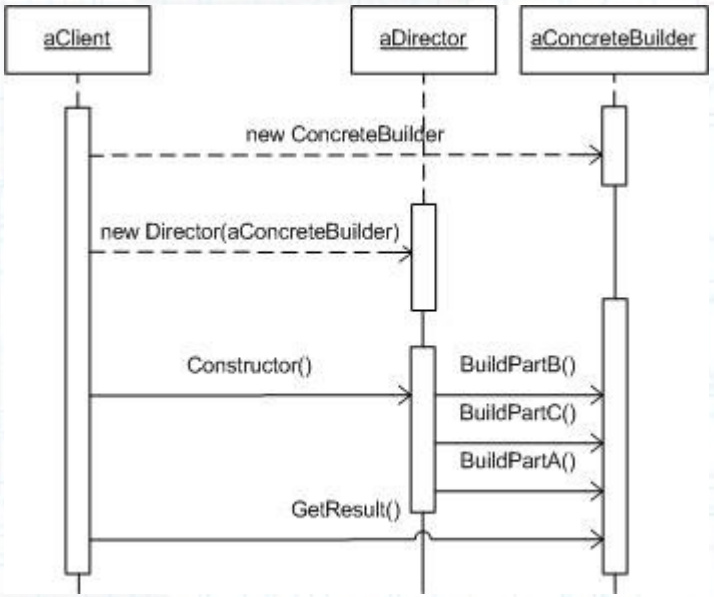
- Паттерн, порождающий объекты
- Отделяет конструирование сложного объекта от его представления
- Алгоритм создания сложного объекта не зависит от того, из каких частей состоит объект
- Процесс конструирования обеспечивает различные представления конструируемого объекта

Паттерн Строитель



- Builder — строитель — задает абстрактный интерфейс для создания частей объекта Product
- ConcreteBuilder — конкретный строитель: создает и собирает части продукта
- Director — распорядитель
- Product - продукт

Паттерн Строитель



- Клиент создает распорядителя Director и конфигурирует его строителем Builder
- Director уведомляет Builder о необходимости создания продукта
- Builder конструирует продукт
- Клиент получает продукт у Builder

Паттерн Строитель

```
class MazeBuilder {
public:
    virtual void BuildMaze() { }
    virtual void BuildRoom(int room) { }
    virtual void BuildDoor(int roomFrom, int roomTo) { }
    virtual Maze* GetMaze() { return 0; }

protected:
    MazeBuilder();
};

Maze* MazeGame::CreateMaze (MazeBuilder& builder) {
    builder.BuildMaze();
    builder.BuildRoom(1);
    builder.BuildRoom(2);
    builder.BuildDoor(1, 2);

    return builder.GetMaze();
}
```

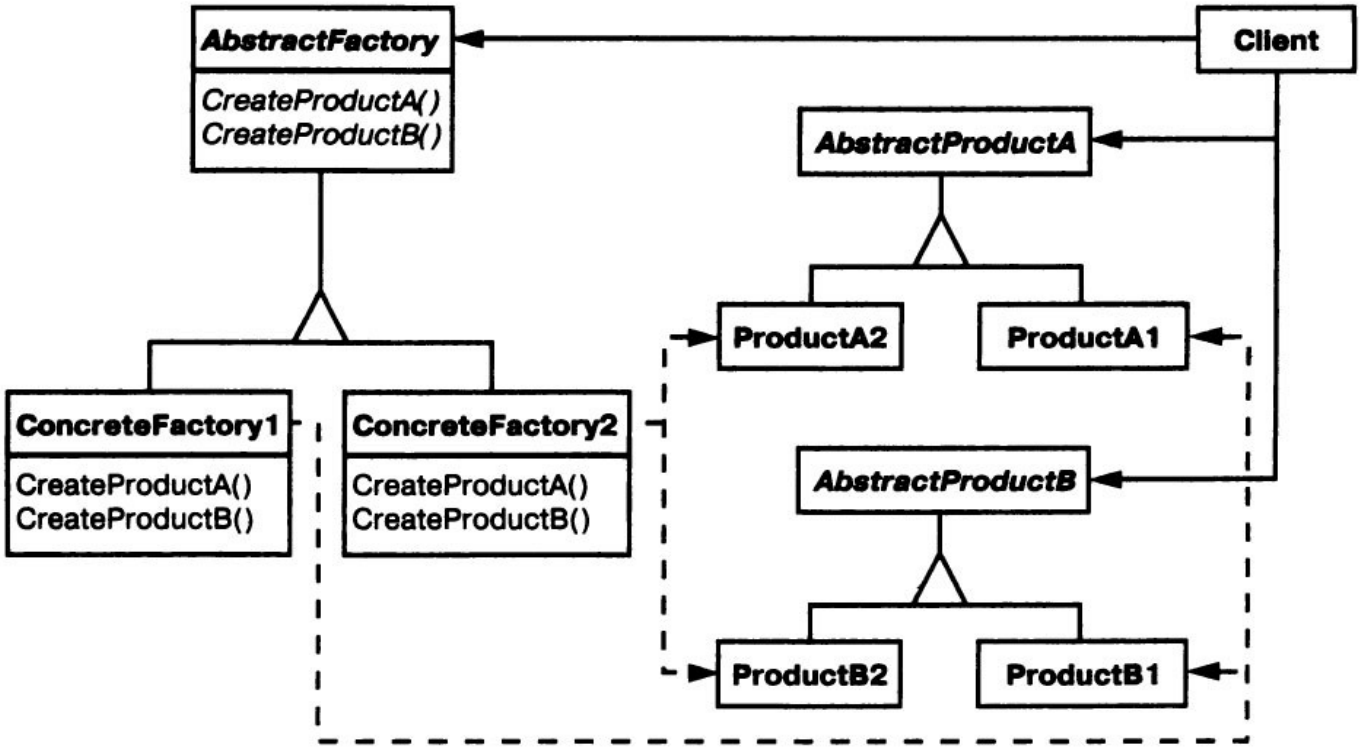
Паттерн Строитель

- Преимущества:
  - Позволяет изменять внутреннее представления продукта
  - Изолирует код, реализующий конструирование и представление
  - Дает более тонкий контроль над процессом конструирования
- Недостатки:
  - При существенных различиях типов конструируемых объектов может усложняться интерфейс строителя

Паттерн Абстрактная фабрика (Abstract Factory)

- Паттерн, порождающий объекты
- Предоставляет интерфейс для создания семейств взаимосвязанных объектов
- Клиент не определяет конкретных классов объектов

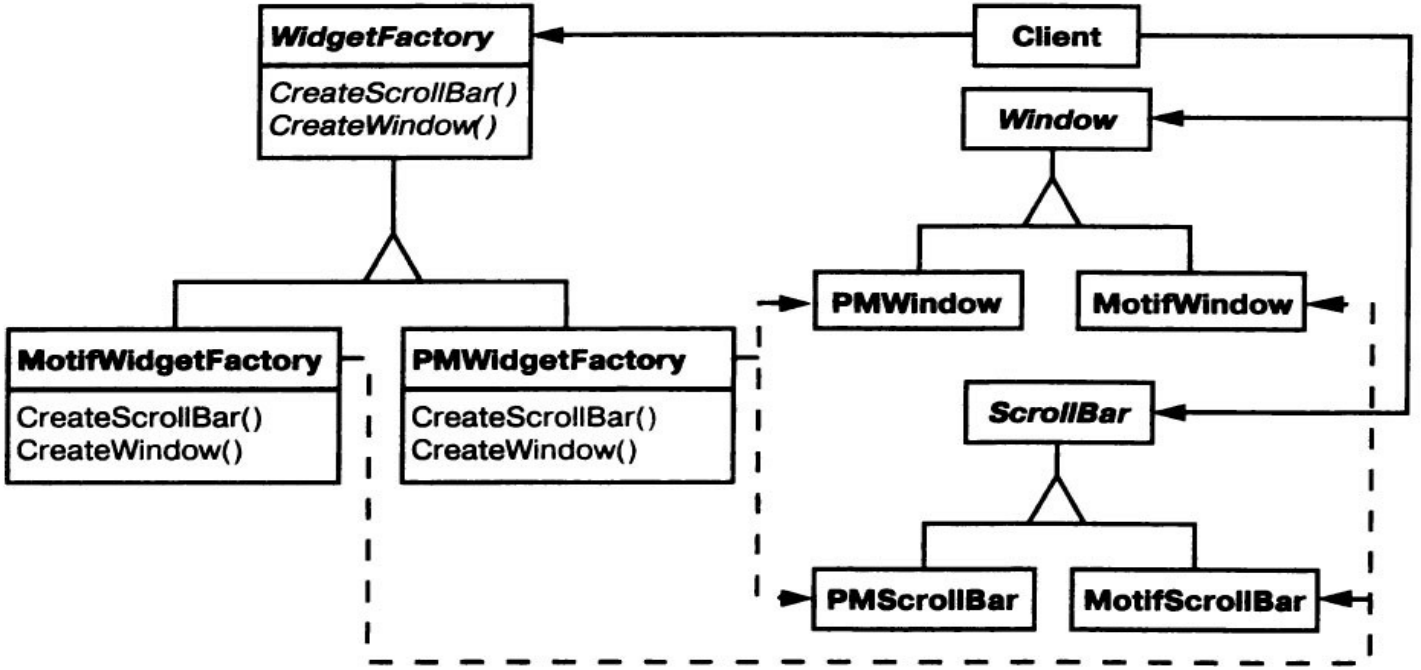
Паттерн Абстрактная фабрика



- Абстрактная фабрика — объявляет интерфейс для операций, создающих абстрактные объекты-продукты (паттерн «Одиночка»)
- Конкретная фабрика — реализует операции, создающие конкретные объекты-продукты (содержит «Фабричные методы»)
- Абстрактный продукт — объявляет интерфейс для типа объекта-продукта
- Конкретный продукт — создается конкретной фабрикой

Пример паттерна Абстрактная фабрика

- Пользовательский интерфейс, поддерживающий разные стандарты внешнего вида: PM и Motif
- Клиент не знает, какие классы использует, а придерживается абстрактного интерфейса
- Конкретные фабрики отвечают за согласование типов создаваемых объектов



Пример паттерна Абстрактная фабрика

```
class MazeFactory {
public:
    virtual Maze* MakeMaze() const { return new Maze; }
    virtual Wall* MakeWall() const { return new Wall; }
    virtual Room* MakeRoom(int n) const { return new Room(n); }
    virtual Door* MakeDoor(Room* rl, Room* r2) const
        { return new Door(rl, r2); }
};

Maze* MazeGame::CreateMaze (MazeFactory& factory) {
    Maze* aMaze = factory.MakeMaze();
    Room* rl = factory.MakeRoom(l);
    Room* r2 = factory.MakeRoom(2);
    Door* aDoor = factory.MakeDoor(rl, r2);

    aMaze->AddRoom(rl);
    aMaze->AddRoom(r2);
    rl->SetSide(North,factory.MakeWall()); rl->SetSide(East,aDoor);
    rl->SetSide(South,factory.MakeWall()); rl->SetSide(East,aDoor);
    rl->SetSide(South,factory.MakeWall());
    rl->SetSide(West,factory.MakeWall());
    r2->SetSide(North,factory.MakeWall());
    r2->SetSide(East,factory.MakeWall());
    r2->SetSide(South,factory.MakeWall()); r2->SetSide(West,aDoor);
    return aMaze;
}
```

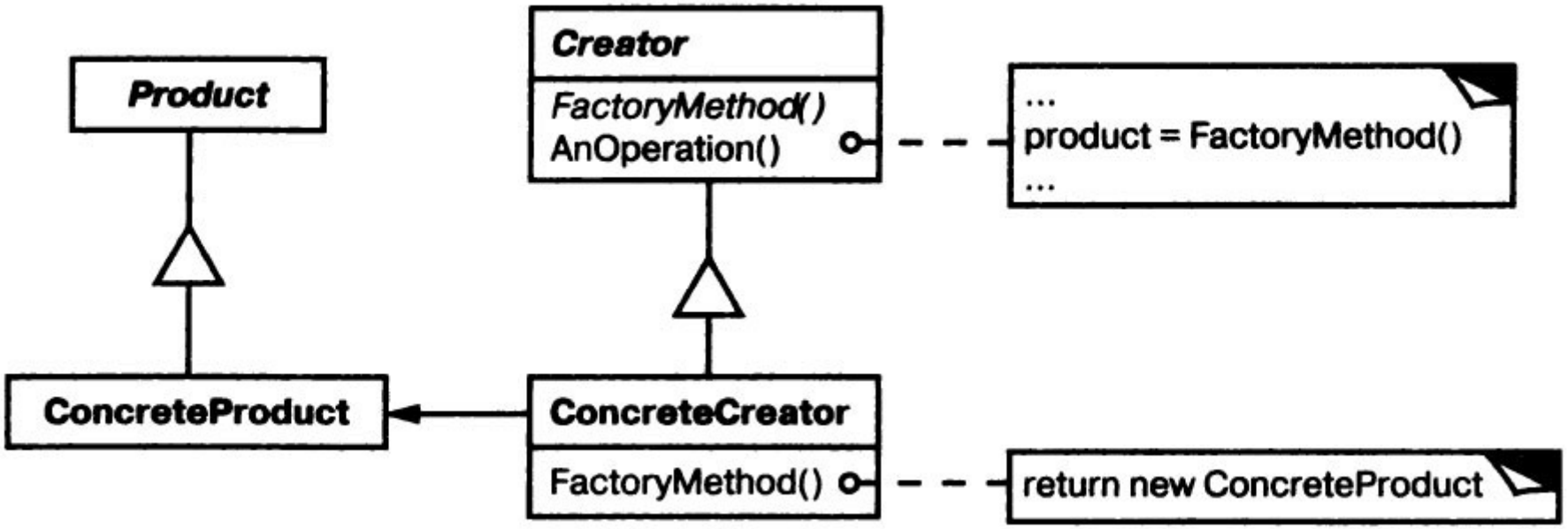
Паттерн Абстрактная фабрика

- Преимущества:
  - Независимость клиентов от процесса создания и компоновки семейства объектов
  - Гарантирует сочетаемость продуктов
  - Упрощает замену семейств продуктов
- Недостатки:
  - Сложность расширения при создании новых видов продуктов

Паттерн Фабричный метод (Factory Method) / Виртуальный конструктор (Virtual Constructor)

- Паттерн, порождающий классы
- Определяет интерфейс для создания объекта, но оставляет подклассам решение о том, какой класс инстанциировать
- Позволяет классу делегировать инстанциирование подклассам
- Класс не знает, объекты каких классов ему нужно создавать, но знают подклассы

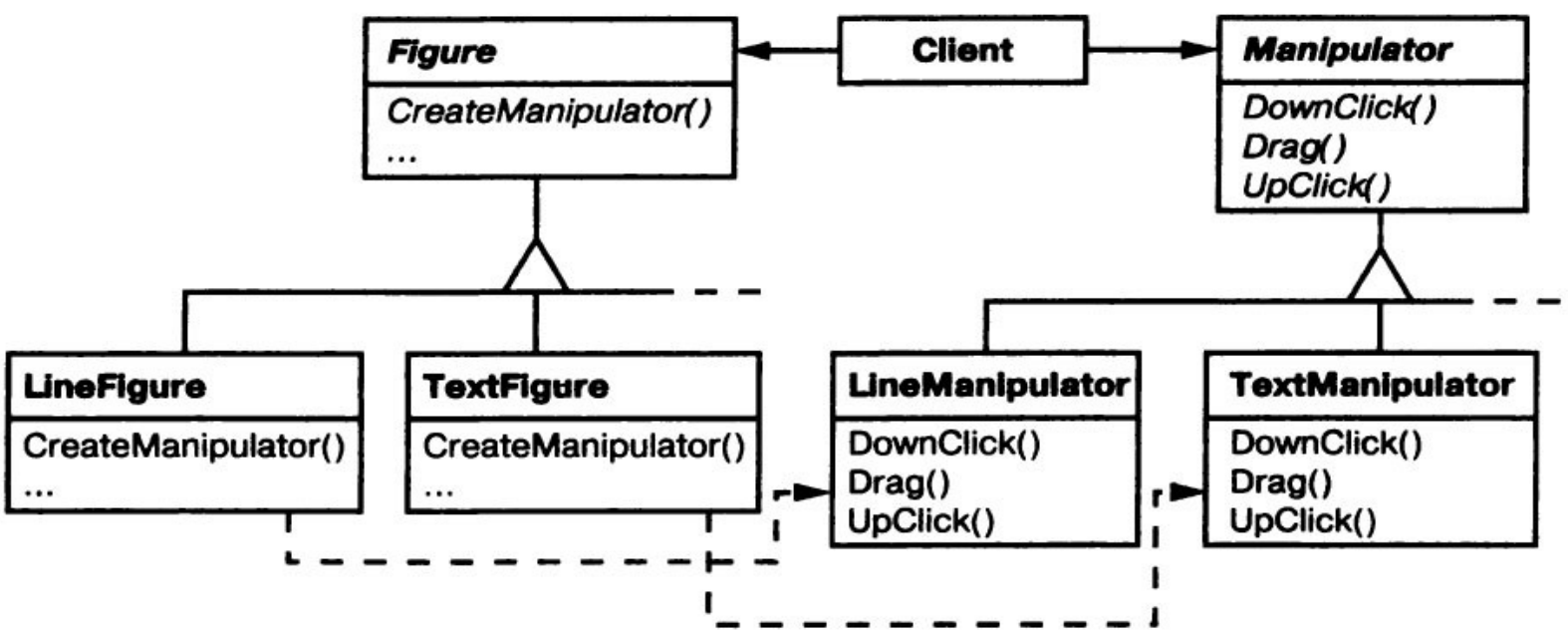
Паттерн Фабричный метод



- Продукт — определяет интерфейс объектов, создаваемых фабричным методом
- Конкретный продукт — реализует интерфейс Продукта
- Создатель — объявляет фабричный метод, может определять реализацию по умолчанию
- Конкретный создатель — замещает фабричный метод, возвращающий Конкретный продукт

Пример паттерна Фабричный метод

- Графические фигуры, которыми можно манипулировать
  - Манипуляция — не часть объекта фигуры
  - Возникает параллельная иерархия
  - Разным фигурам может соответствовать один и тот же алгоритм манипуляции



Пример паттерна Фабричный метод

```
class Creator {
    public:
    virtual Product* Create(ProductId);

};

Product* Creator::Create (ProductId id) {

    if (id == MINE) return new MyProduct;
    if (id == YOURS) return new YourProduct;
    return 0;
}

Product* MyCreator::Create (ProductId id) {

    if (id == YOURS) return new MyProduct;
    if (id == MINE) return new YourProduct;
    if (id == THEIRS) return new TheirProduct;
    return Creator::Create(id);
}
```

Паттерн Фабричный метод

- Преимущества:
  - Логика взаимодействия с объектами абстрагирована от процесса их инстанциирования
  - Фабричный метод может быть параметризируемым
- Недостатки:
  - Для создания Конкретного продукта нужно обязательно создавать Конкретного создателя

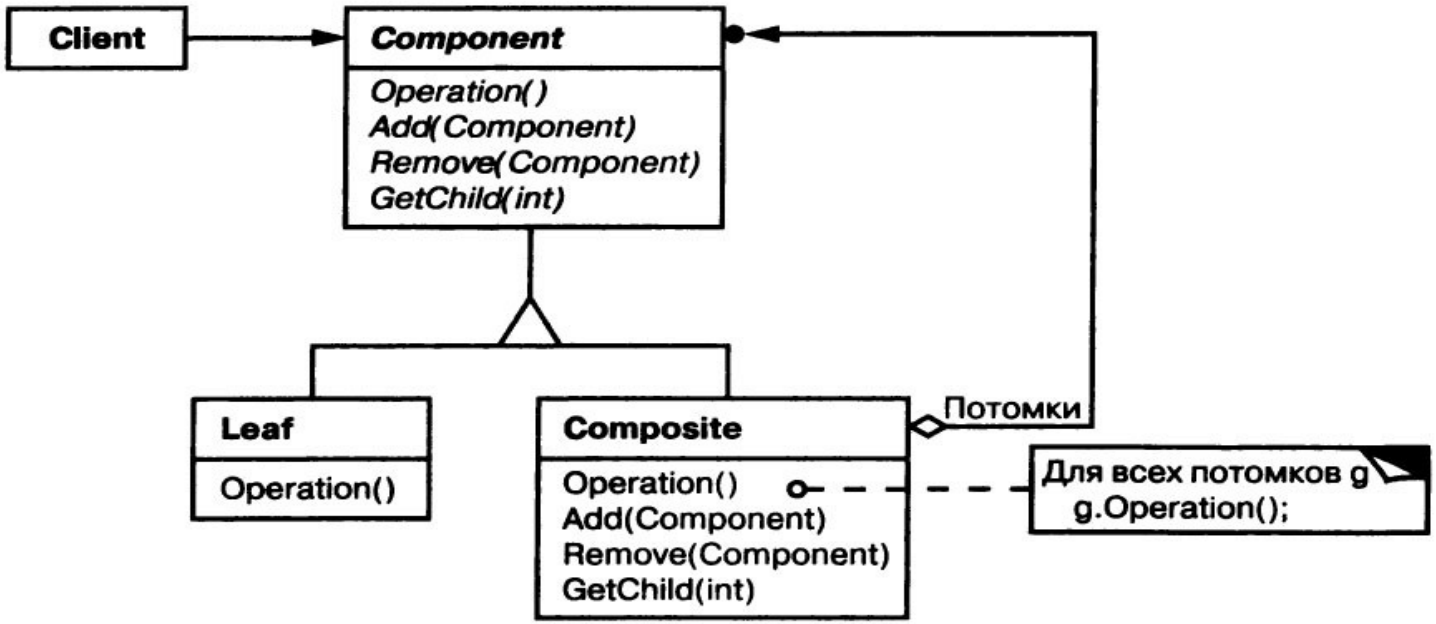
Структурные паттерны

- Отвечают за композицию классов и объектов в более крупные структуры
- Паттерны уровня класса используют наследование
- Паттерны уровня объекта используют агрегирование и композицию
- Композиция может изменяться во время выполнения

Паттерн Компоновщик (Composite)

- Паттерн, структурирующий объекты
- Компонует объекты в древовидные структуры для представления иерархий часть-целое
- Позволяет клиентам единообразно трактовать индивидуальные и составные объекты
- Основа паттерна — абстрактный класс, представляющий и примитивы, и контейнеры

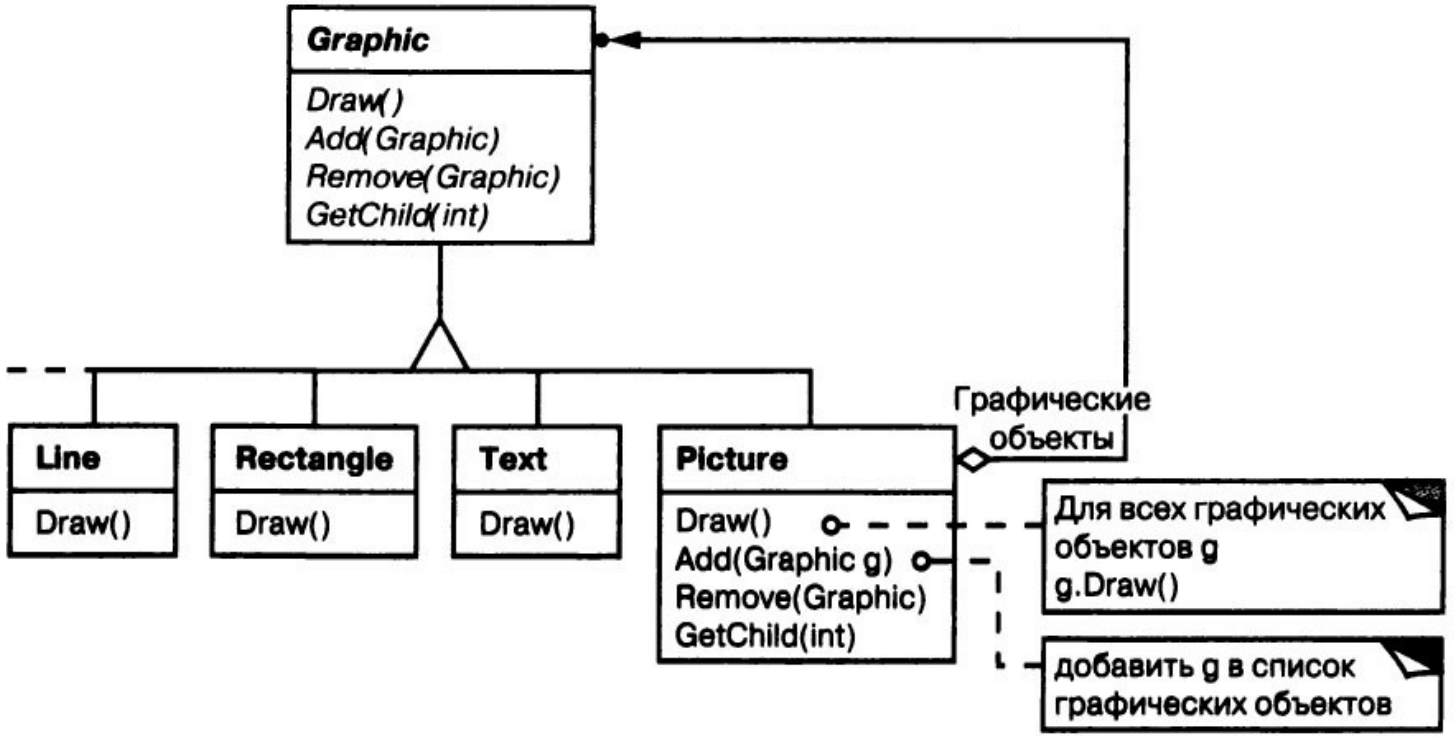
Паттерн Компоновщик



- Компонент — интерфейс для компонуемых объектов, реализует общие операции, интерфейс для доступа к потомкам/родителю и управлению ими
- Лист — не имеет потомков
- Составной объект — определяет поведение компонентов, у которых есть потомки, хранит компоненты-потомки, реализует операции управления потомками
- Клиент — манипулирует объектами через Компонент

Пример паттерна Компоновщик

- Графическая система:
  - Графический элемент — Компонент
  - Графические примитивы: линия, прямоугольник, текст — Листья
  - Картинка — составной объект



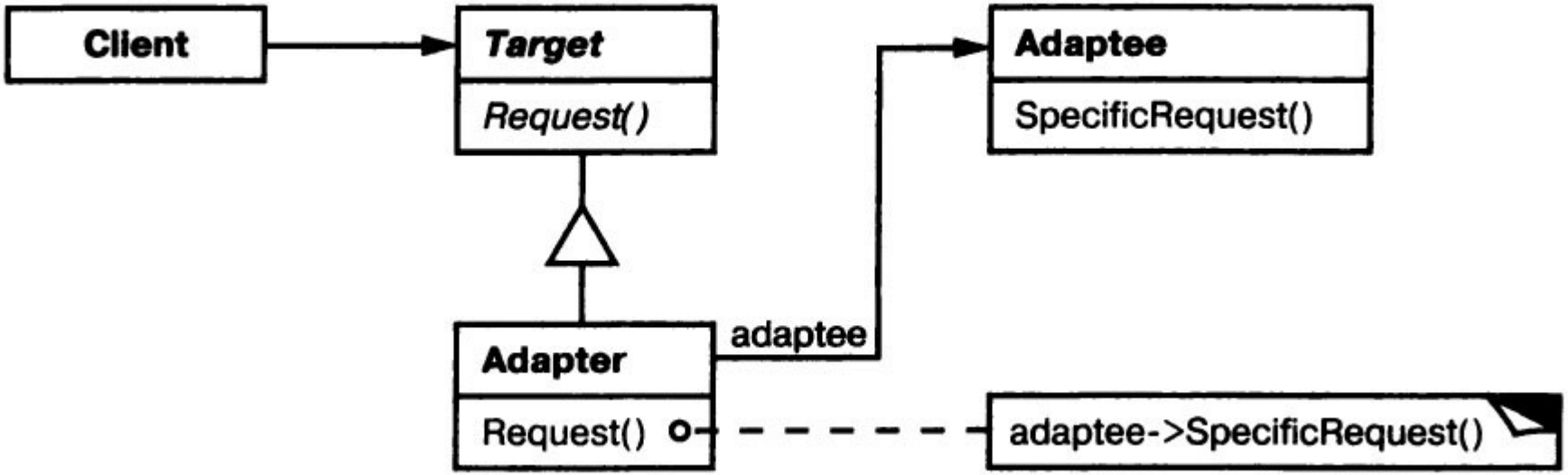
Паттерн Компоновщик

- Преимущества:
  - Унификация композиции объектов
  - Упрощение архитектуры клиента: нет ветвления в зависимости от типа узла
  - Простота добавления нового компонента
- Недостатки:
  - Сложность накладывания ограничений на состав композиции
  - Разнородные объекты «раздувают» интерфейс Компонента

Паттерн Адаптер (Adapter)

- Паттерн, структурирующий классы и объекты
- Преобразует интерфейс одного класса в другой
- Обеспечивает совместимость классов

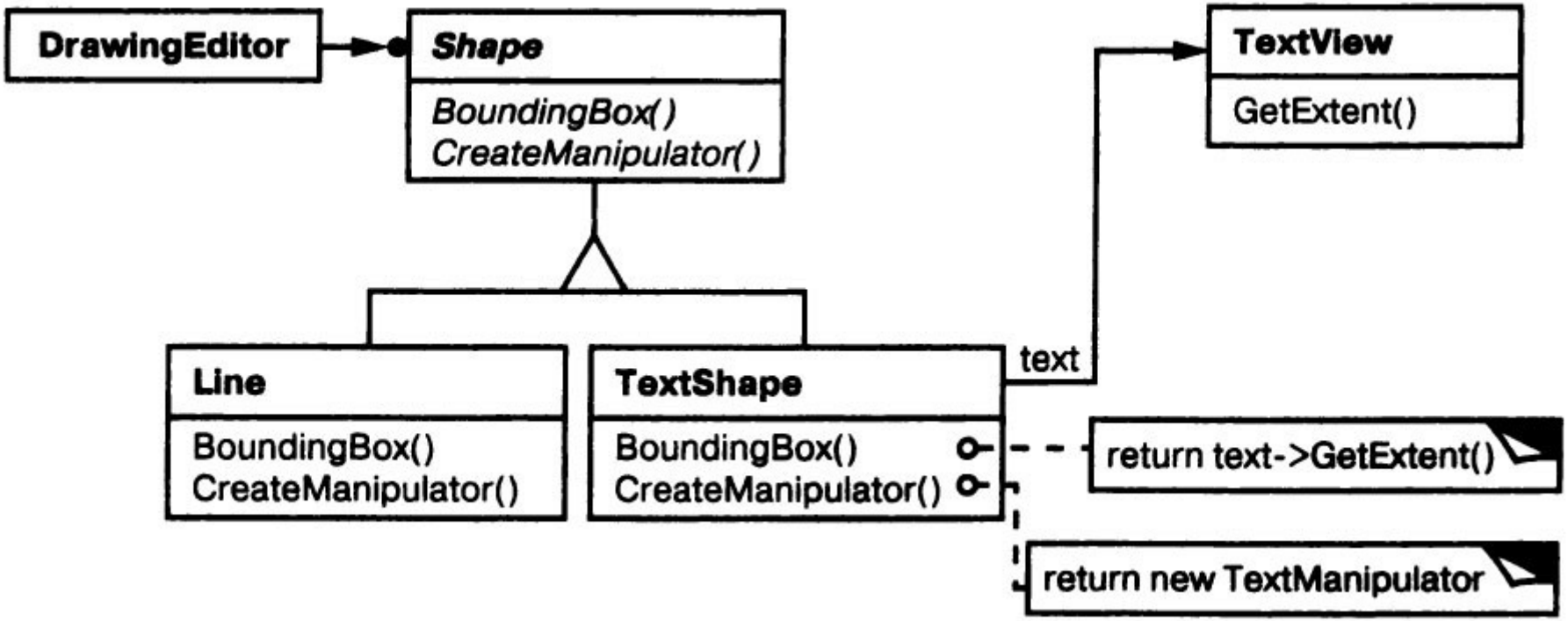
Паттерн Адаптер (Adapter)



- Целевой — определяет зависящий от предметной области интерфейс, которым пользуется клиент
- Клиент — вступает во взаимоотношения с объектами, удовлетворяющими целевому интерфейсу
- Адаптируемый — определяет существующий интерфейс, который нуждается в адаптации
- Адаптер — адаптирует интерфейс Адаптируемого к Целевому

Пример паттерна Адаптер

- Графические объекты, имеющие изменяемую форму и отображающие сами себя
- Задача: повторно использовать редактор текста TextView
- TextShape адаптирует TextView для редактора



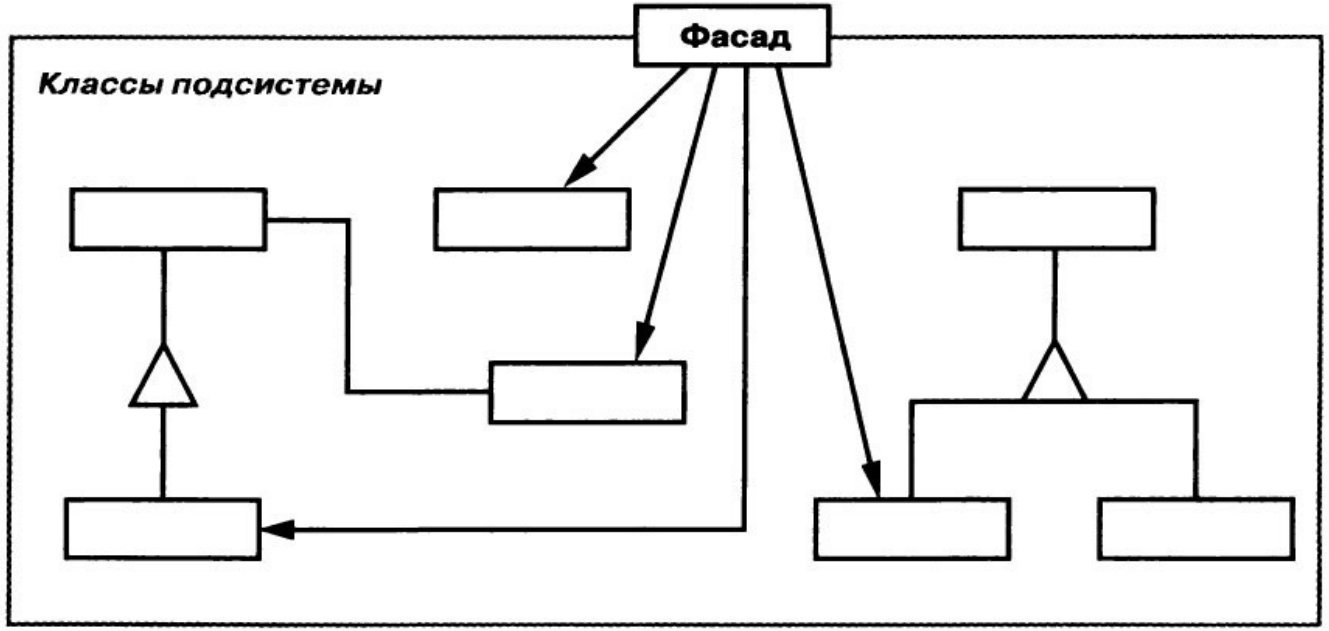
Паттерн Адаптер (Adapter)

- Преимущества:
  - Повторное использование классов без их изменения
- Недостатки:
  - Усложняется при сильном отличии адаптируемого объекта
  - Затруднено замещение операций адаптируемого класса

Паттерн Фасад (Facade)

- Паттерн, структурирующий объекты
- Предоставляет унифицированный интерфейс вместо набора нескольких интерфейсов
- Определяет упрощенный интерфейс более высокого уровня
- Снижает зависимость подсистем и обмен информацией
- Не содержит бизнес-логики

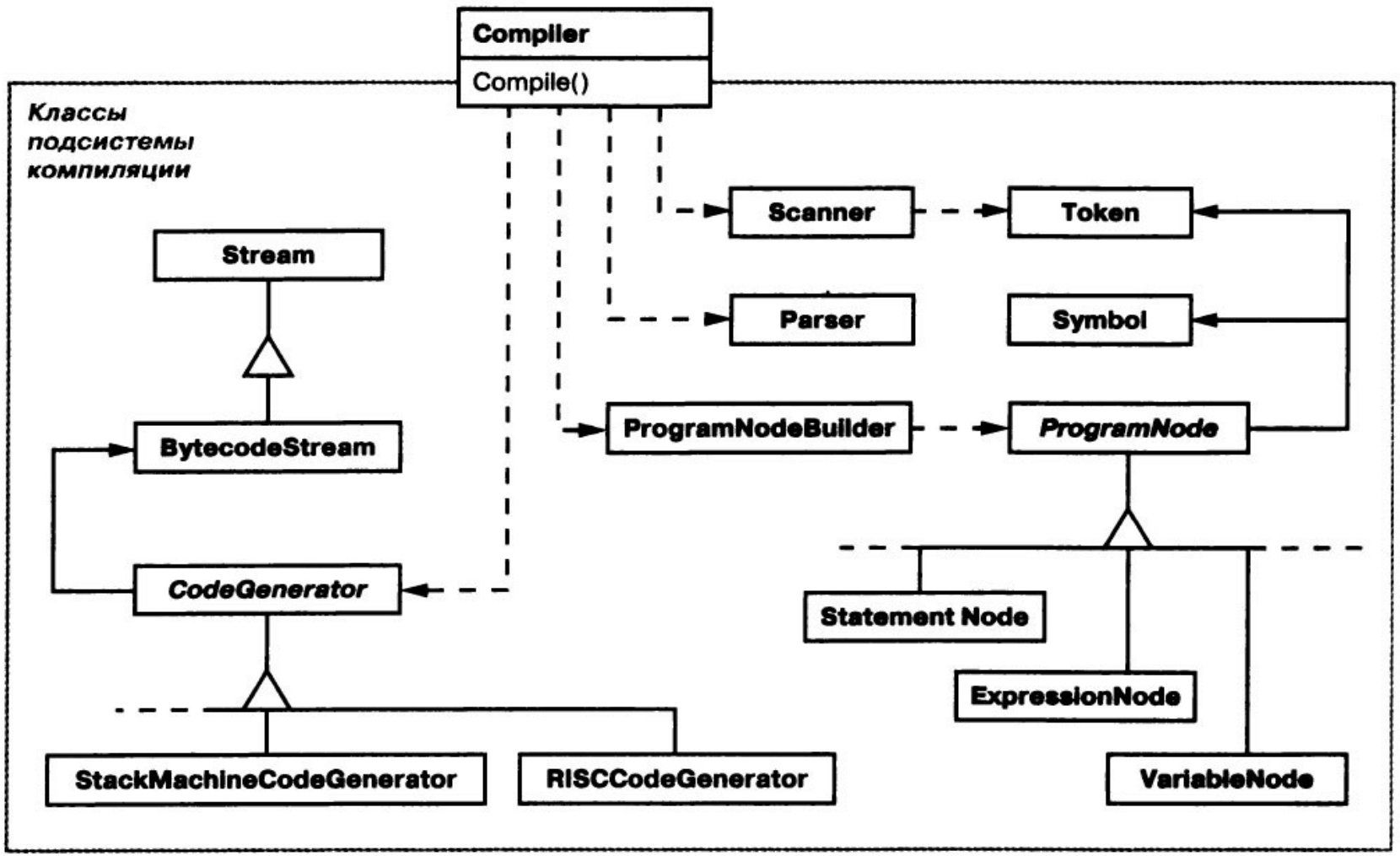
Паттерн Фасад



- Фасад - «знает», каким классам подсистемы адресовать запрос, делегирует запросы клиентов подходящим объектам внутри подсистемы
- Классы подсистемы — реализуют функциональность подсистемы, ничего не «знают» о существовании фасада, скрыты от клиента

Пример паттерна Фасад

- Среда программирования, предоставляющая приложениям доступ к системе компиляции



Паттерн Фасад

- Преимущества:
  - Изолирует клиентов от компонентов подсистемы
  - Ослабляет связность между подсистемами
  - Инструмент расслоения
  - Не запрещает обращения «напрямую»
- Недостатки:
  - Возможно «раздувание» фасада

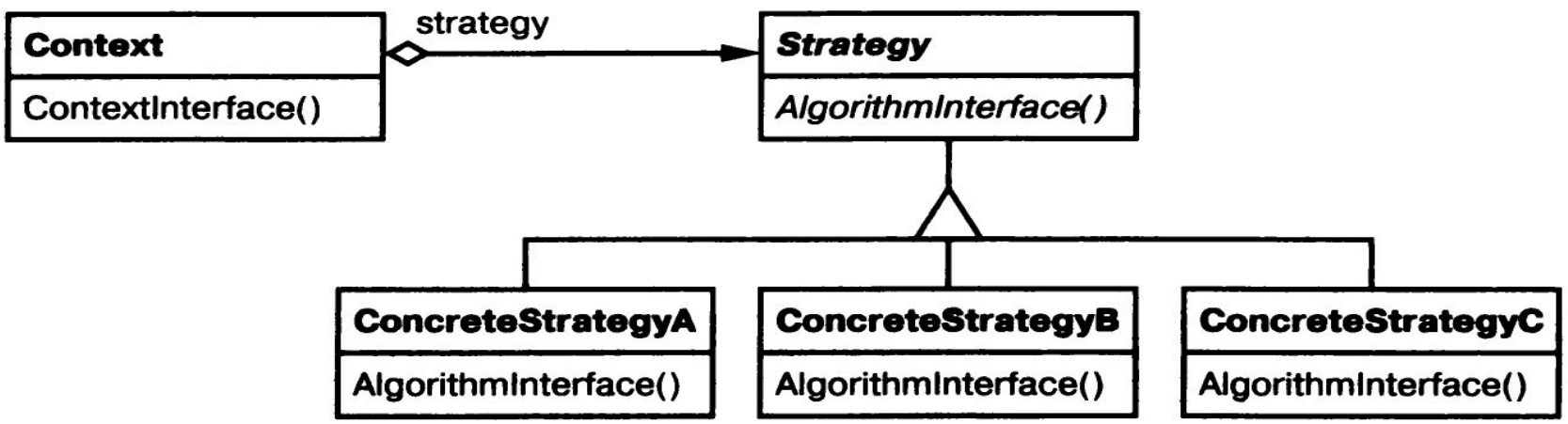
Паттерны поведения

- Связаны с алгоритмами и распределением обязанностей между объектами
- Характеризуют сложный поток управления
- В паттернах уровня класса используется наследование
- В паттернах уровня объекта используется композиция

Паттерн Стратегия (Strategy) / Политика (Policy)

- Паттерн поведения объектов
- Определяет семейство алгоритмов
- Инкапсулирует реализацию алгоритма за интерфейсом
- Делает алгоритмы взаимозаменяемыми
- Делает реализацию независимой от клиента

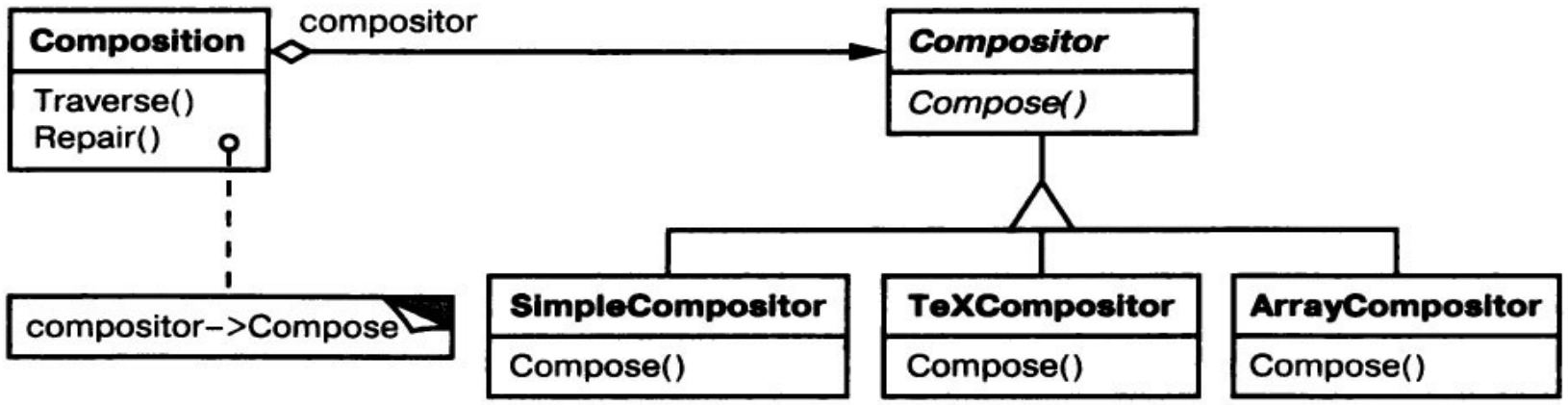
Паттерн Стратегия



- Стратегия — объявляет общий для всех алгоритмов интерфейс
- Конкретная стратегия — реализует алгоритм, использующий интерфейс Strategy
- Контекст — конфигурируется объектом Конкретная стратегия, хранит ссылку на объект Стратегия, может определять интерфейс, предоставляющий данные для Стратегии

Пример паттерна Стратегия

- Задача разбиения текста на строки
- Разные стратегии разбиения
- Composition — хранит ссылку на Compositor и делегирующий ему обязанность по переформатированию текста
- Клиент параметризирует Composition объектом Compositor



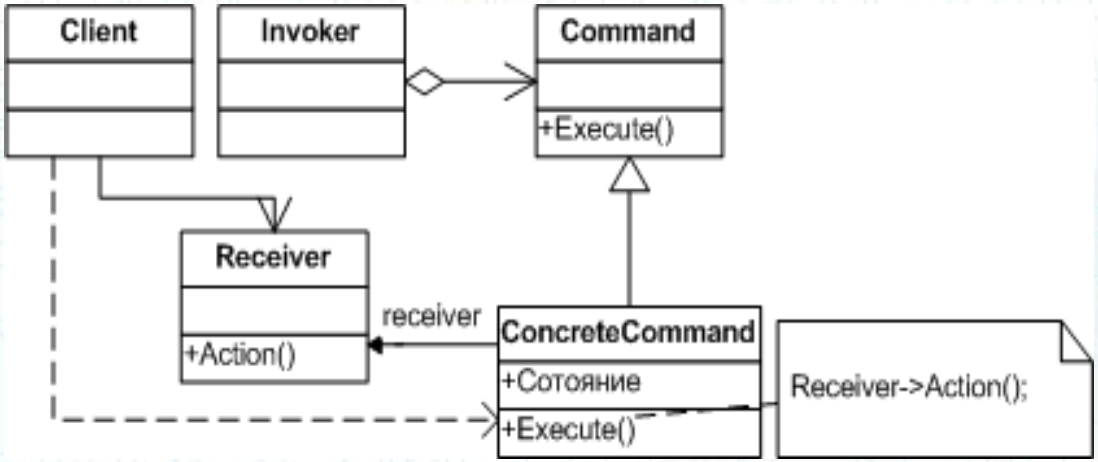
Паттерн Стратегия

- Преимущества:
  - Повторное использование семейства родственных алгоритмов
  - Отделение реализации алгоритма от данных
  - Избавление от условных операторов
- Недостатки:
  - Клиент должен знать о стратегиях
  - Обмен излишней информацией между контекстом и стратегией
  - Увеличение числа объектов

Паттерн Команда (Command) / Действие (Action) / Транзакция (Transaction)

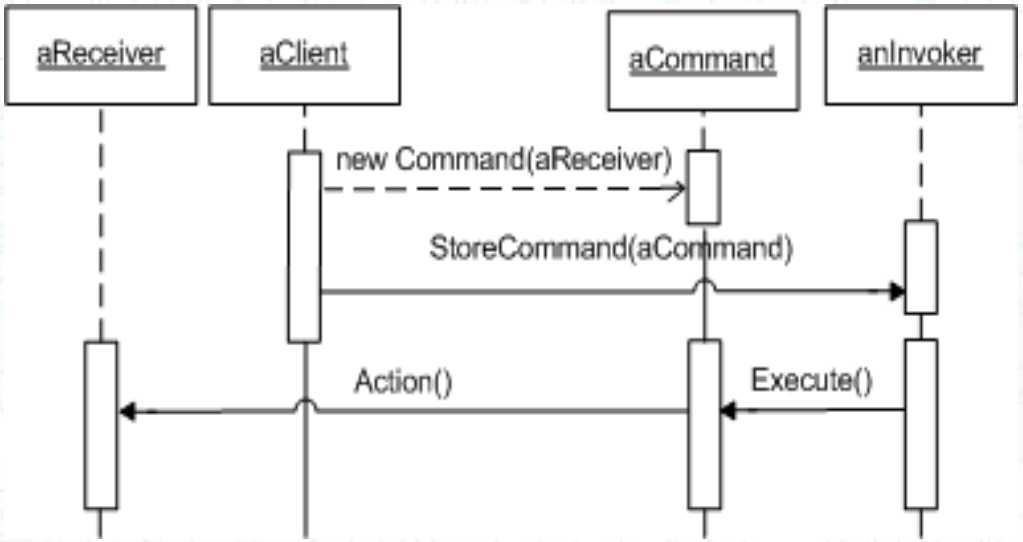
- Паттерн поведения объектов
- Инкапсулирует запрос как объект
- Поддерживает: параметризацию запросов, очередь, протоколирование
- Позволяет объектам отправлять запросы неизвестным объектам, преобразовав запрос в объект
- Отделяет объект, инициирующий операцию, от объекта, который «знает», как ее выполнить

Паттерн Команда



- Command — команда — объявляет интерфес для выполнения операции
- ConcreteCommand — конкретная команда — определяет связь между получателем и действием, реализует операцию Execute путем вызова операций получателя
- Client — клиент — создает объект ConcreteCommand и устанавливает получателя
- Invoker — инициатор — обращается к команде для выполнения запроса
- Receiver — получатель — знает о способах выполнения операций, необходимых для удовлетворения запроса

Паттерн Команда



- Клиент создает объект ConcreteCommand и устанавливает для него получателя
- Инициатор Invoker сохраняет объект ConcreteCommand
- Инициатор отправляет запрос, вызывая операцию команды Execute. Если поддерживается отмена выполненных действий, то ConcreteCommand перед вызовом Execute сохраняет информацию о состоянии, достаточную для выполнения отката
- Объект ConcreteCoimand вызывает операции получателя для выполнения запроса

Паттерн Команда

- Команда разрывает связь между объектом, инициирующим операцию, и объектом, имеющим информацию о том, как ее выполнить
- Допускается расширять команды, как в случае с любыми другими объектами
- Из простых команд можно собирать составные. В общем случае составные команды описываются паттерном Компоновщик
- Добавлять новые команды просто, поскольку никакие существующие классы изменять не нужно

Антипаттерны

- **Базовый класс-утилита** (BaseBean) — наследование функциональности из класса-утилиты вместо делегирования
- **Вызов предка** (CallSuper) — методу класса-потомка требуется в обязательном порядке вызывать те же методы класса-предка
- **Божественный объект** (God object) — слишком большое количество сложных методов в классе
- **Полтергейст** (Poltergeist) — объекты, единственное предназначение которых передавать информацию другим объектам
- **Одиночество** (Singletonitis) — неуместное использование паттерна одиночка
- **Приватизация** (Privatisation) — сокрытие функциональности в приватной секции, что затрудняет его расширение в классах-потомках

далее..

Типовые решения источников данных