Паттерны, детали реализации

Юрий Литвинов
yurii.litvinov@gmail.com

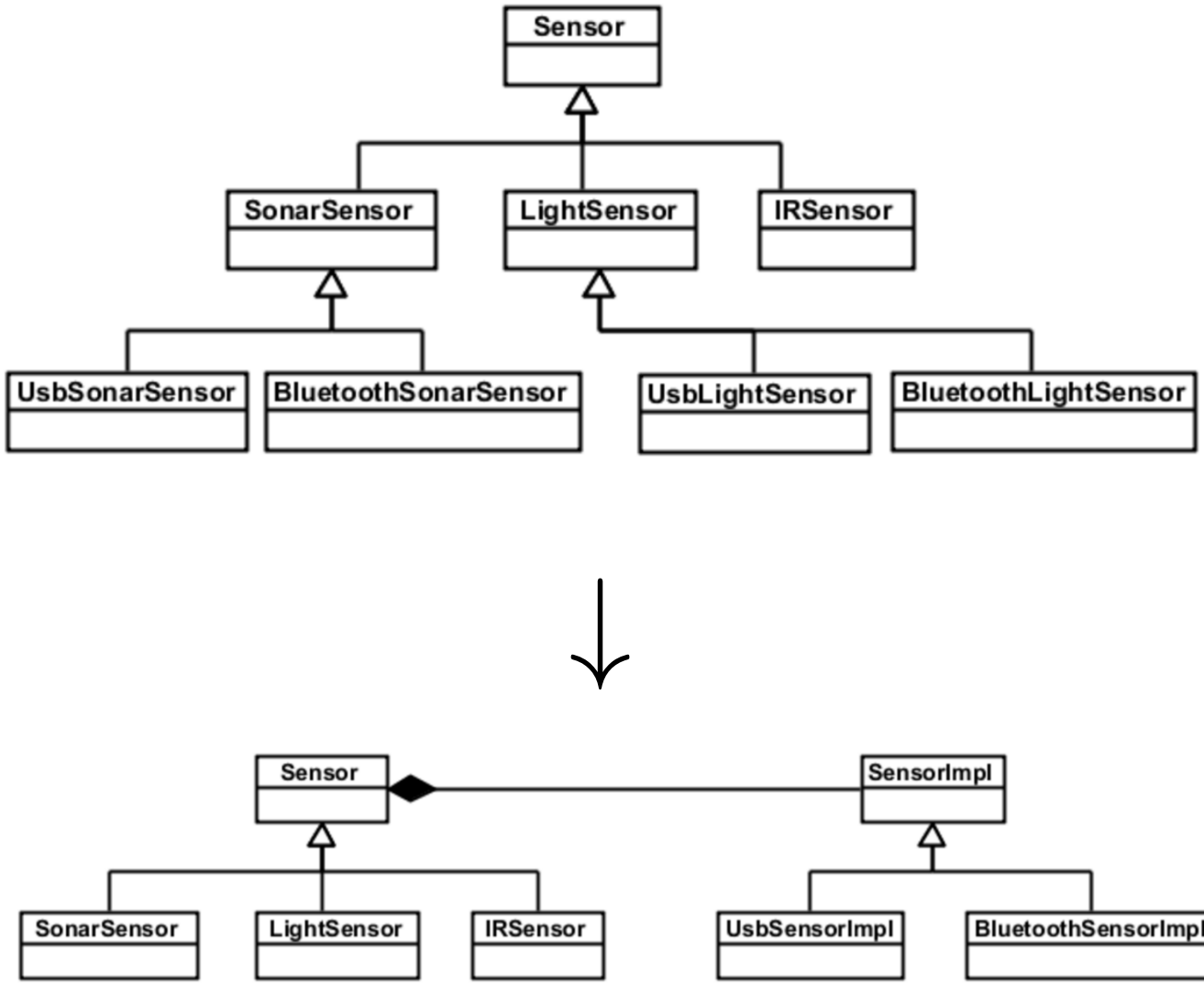
27.04.2018г

Паттерн “Мост” (Bridge)

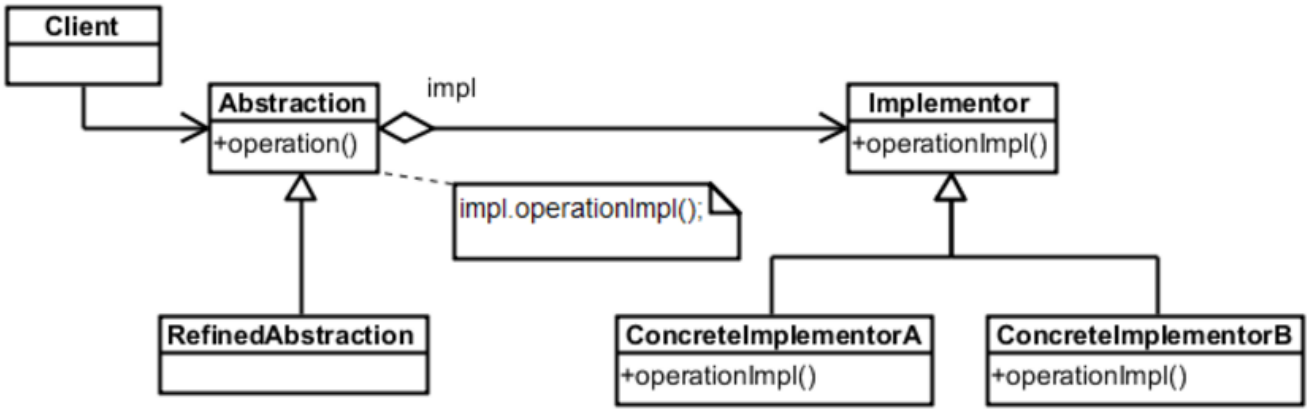
Отделяет абстракцию от реализации
Пример:

- Есть система, интерпретирующая программы для роботов
- Есть класс *Sensor*, от которого наследуются *SonarSensor*, *LightSensor*, ...
- Связь с роботом может выполняться по USB или Bluetooth, а может быть, программа и вовсе исполняется на симуляторе
- Интерпретатор хочет работать с сенсорами, не заморачиваясь реализацией механизма связи
- Рабоче-крестьянская реализация — *USBLightSensor*, *BluetoothLightSensor*, *USBSonarSensor*, *BluetoothSonarSensor*, ...
- Число классов — произведение количества сенсоров и типов связи

“Мост”, пример


Sensor
SonarSensor
LightSensor
IRSensor
UsbSonarSensor
BluetoothSonarSensor
UsbLightSensor
BluetoothLightSensor
Sensor
SensorImpl
SonarSensor
LightSensor
IRSensor
UsbSensorImpl
BluetoothSensorImpl

“Мост”, общая схема



- *Abstraction* — определяет интерфейс абстракции, хранит ссылку на реализацию
- *RefinedAbstraction* — расширяет интерфейс абстракции, делает полезную работу, используя реализацию
- *Implementor* — определяет интерфейс реализации, в котором абстракции предоставляются низкоуровневые операции
- *ConcreteImplementor* — предоставляет конкретную реализацию Implementor

Когда применять

- Когда хочется разделить абстракцию и реализацию, например, когда реализацию можно выбирать во время компиляции или во время выполнения
  - “Стратегия”, “Прокси”
- Когда абстракция и реализация должны расширяться новыми подклассами
- Когда хочется разделить одну реализацию между несколькими объектами
  - Как copy-on-write в строках

Тонкости реализации

Создание правильного Implementor-а

- Самой абстракцией в конструкторе, в зависимости от переданных параметров
  - Как вариант — выбор реализации по умолчанию и замена её по ходу работы
- Принимать реализацию извне (как параметр конструктора, или, реже, как значение в сеттер)
- Фабрика/фабричный метод
  - Позволяет спрятать платформозависимые реализации, чтобы не зависеть от них всех при сборке

Pointer To Implementation (PImpl)

Вырожденный мост для C++, когда “абстракция” имеет ровно одну реализацию, часто полностью дублирующую её интерфейс
Зачем: чтобы клиенты класса не зависели при сборке от его реализации

- Позитивно сказывается на времени компиляции программ на C++
- Позволяет менять реализацию независимо
  - Сохраняя бинарную совместимость

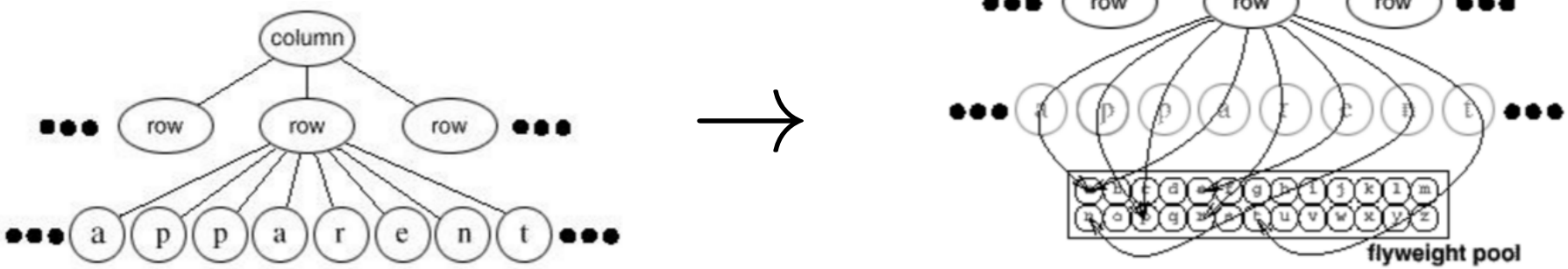
Как: предварительное объявление класса-реализации, полное определение — в .cpp-файле вместе с методами абстракции
Часто используется в реализации библиотек (например, Qt)

Паттерн “Приспособленец” (Flyweight)

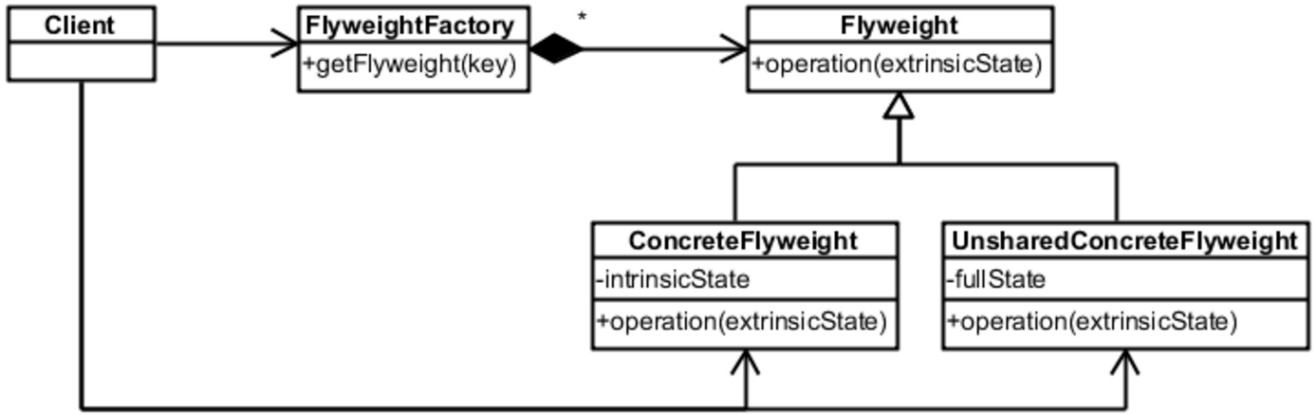
Предназначается для эффективной поддержки множества мелких объектов
Пример:

- Есть текстовый редактор
- Хочется работать с каждым символом как с объектом
  - Единообразие алгоритмов форматирования и внутренней структуры документа
  - Более красивая и ООПшная реализация
    - Паттерн “Компоновщик”, структура “Символ” $\rightarrow$ “Строка” $\rightarrow$ “Страница”
- Наивная реализация привела бы к чрезмерной расточительности по времени работы и по памяти, потому что документы с миллионами символов не редкость

“Приспособленец”, пример

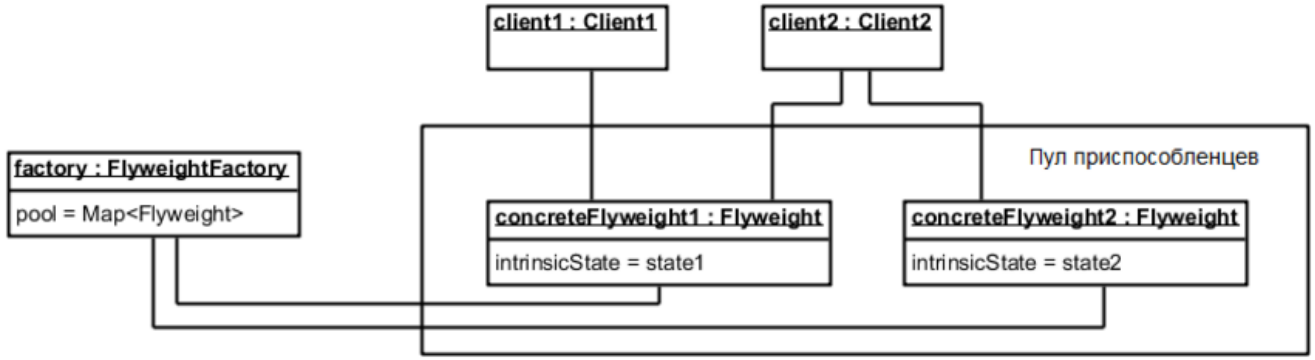


“Приспособленец”, общая схема



- *Flyweight* — определяет интерфейс, через который приспособленцы могут получать внешнее состояние
- *ConcreteFlyweight* — реализует интерфейс Flyweight и может иметь внутреннее состояние, не зависит от контекста
- *UnsharedConcreteFlyweight* — неразделяемый “приспособленец”, хранящий всё состояние в себе, бывает нужен, чтобы собирать иерархические структуры из Flyweight-ов (“Компоновщик”)
- *FlyweightFactory* — содержит пул приспособленцев, создаёт их и управляет их жизнью

“Приспособленец”, диаграмма объектов



- Клиенты могут быть разных типов
- Клиенты могут разделять приспособленцев
  - Один клиент может иметь несколько ссылок на одного приспособленца
- Во время выполнения клиенты имеют право не знать про фабрику

Когда применять

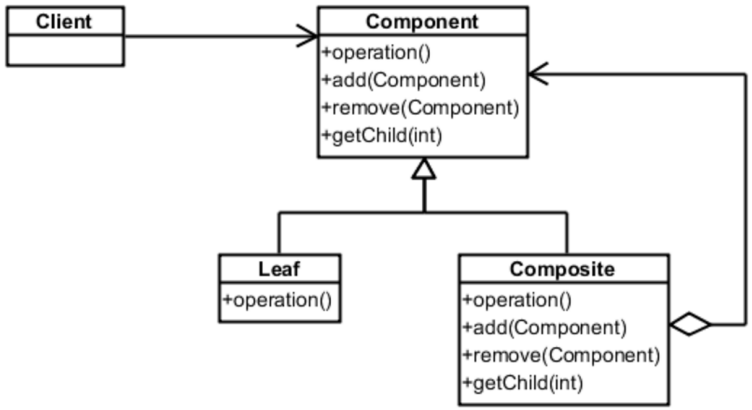
- Когда в приложении используется много мелких объектов
- Они допускают разделение состояния на внутреннее и внешнее
  - Желательно, чтобы внешнее состояние было вычислимо
- Идентичность объектов не важна
  - Используется семантика Value Type
- Главное, когда от такого разделения можно получить ощутимый выигрыш

Тонкости реализации

- Внешнее состояние — по сути, отдельный объект, поэтому если различных внешних состояний столько же, сколько приспособленцев, смысла нет
  - Один объект-состояние покрывает сразу несколько приспособленцев
    - Например, объект “Range” может хранить параметры форматирования для всех букв внутри фрагмента
- Клиенты не должны инстанцировать приспособленцев сами, иначе трудно обеспечить разделение
  - Имеет смысл иметь механизм для удаления неиспользуемых приспособленцев
    - Если их может быть много
- Приспособленцы немутабельны и Value Objects (с правильно переопределённой операцией сравнения)
  - Про hashCode() тоже надо не забыть

“Компоновщик” (Composite), детали реализации

- Ссылка на родителя
  - Может быть полезна для простоты обхода
  - “Цепочка обязанностей”
  - Но дополнительный инвариант
  - Обычно реализуется в Component



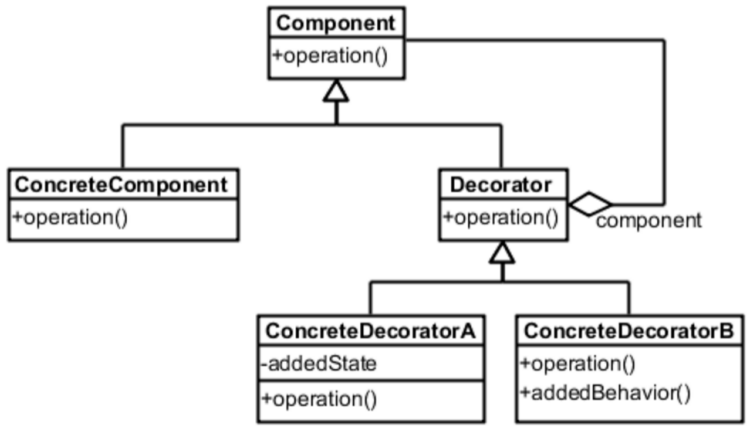
- Разделяемые поддеревья и листья
  - Позволяют сильно экономить память
  - Проблемы с навигацией к родителям и разделяемым состоянием
  - Паттерн “Приспособленец”
- Идеологические проблемы с операциями для работы с потомками
  - Не имеют смысла для листа
    - Можно считать Leaf Composite-ом, у которого всегда 0 потомков
  - Операции add и remove можно объявить и в Composite, тогда придётся делать cast
    - Иначе надо бросать исключения в add и remove

“Компоновщик”, детали реализации (2)

- Операция getComposite() – более аккуратный аналог cast-а
- Где определять список потомков
  - В Composite, экономия памяти
  - В Component, единообразие операций
  - “Список” вполне может быть хеш-таблицей, деревом или чем угодно
- Порядок потомков может быть важен, может нет
- Кеширование информации для обхода или поиска
  - Например, кеширование ограничивающих прямоугольников для фрагментов картинки
  - Инвалидация кеша
- Удаление потомков
  - Если нет сборки мусора, то лучше в Composite
  - Следует опасаться разделяемых листьев/поддеревьев

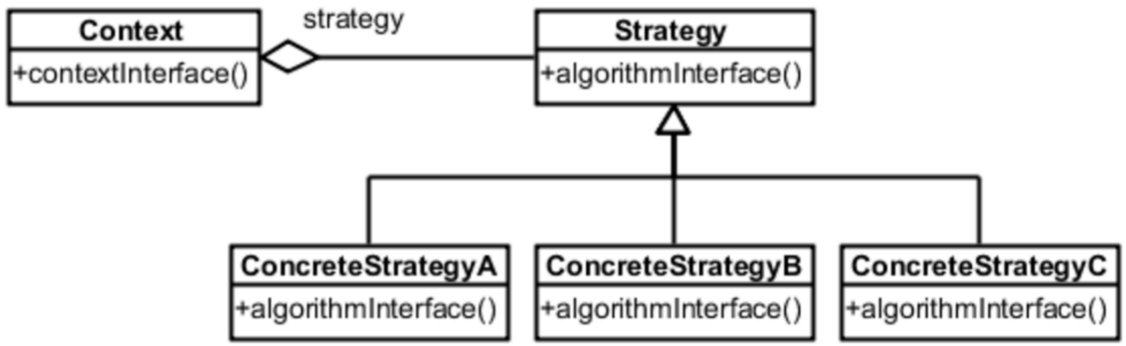
“Декоратор” (Decorator), детали реализации

- Интерфейс декоратора должен соответствовать интерфейсу декорируемого объекта
  - Иначе получится “Адаптер”
- Если конкретный декоратор один, абстрактный класс можно не делать



- Component должен быть по возможности небольшим (в идеале, интерфейсом)
  - Иначе лучше паттерн “Стратегия”
  - Или самодельный аналог, например, список “расширений”, которые вызываются декорируемым объектом вручную перед операцией или после неё

“Стратегия” (Strategy), детали реализации



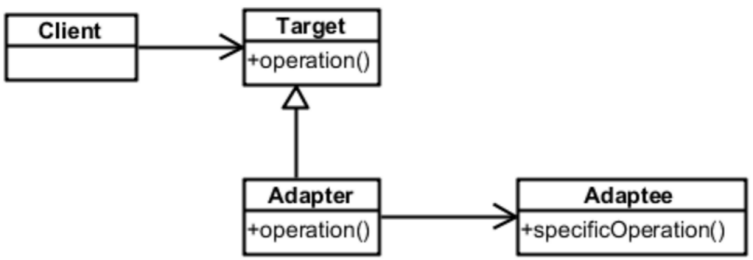
- Передача контекста вычислений в стратегию
  - Как параметры метода — уменьшает связность, но некоторые параметры могут быть стратегии не нужны
  - Передавать сам контекст в качестве аргумента — в Context интерфейс для доступа к данным

“Стратегия” (Strategy), детали реализации (2)

- Стратегия может быть параметром шаблона
  - Если не надо её менять на лету
  - Не надо абстрактного класса и нет оверхеда на вызов виртуальных методов
- Стратегия по умолчанию
  - Или просто поведение по умолчанию, если стратегия не установлена
- Объект-стратегия может быть приспособленцем

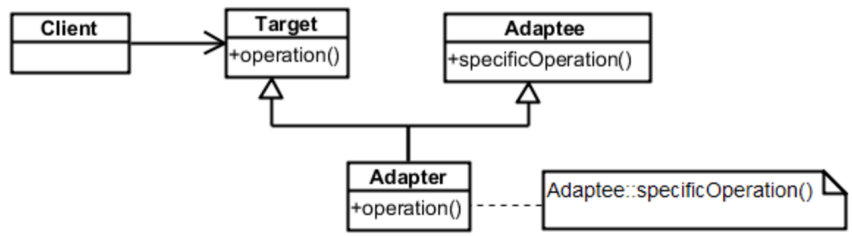
“Адаптер” (Adapter), детали реализации

- Адаптер объекта:



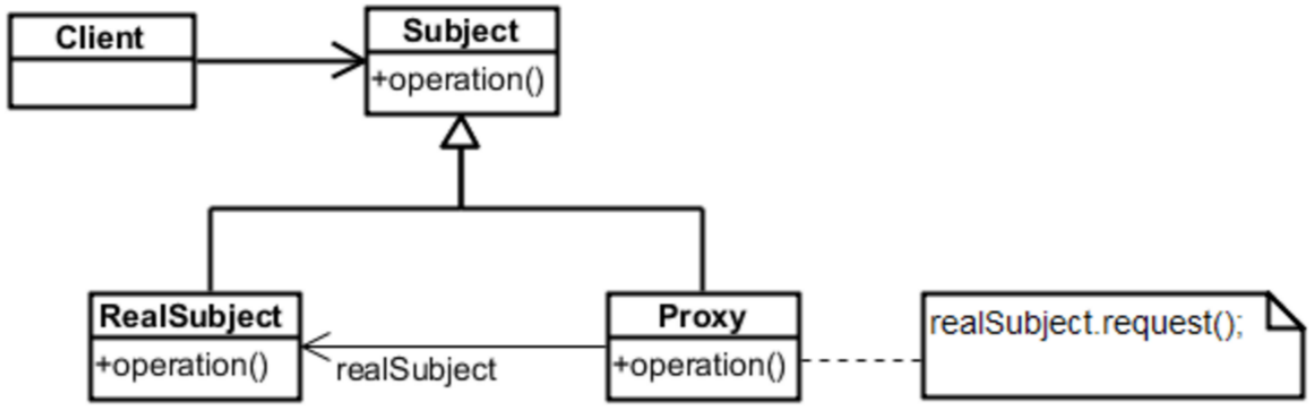
- Похоже на “Мост”

- Адаптер класса:



- Нужно множественное наследование
  - private-наследование в C++

“Заместитель” (Proxy), детали реализации



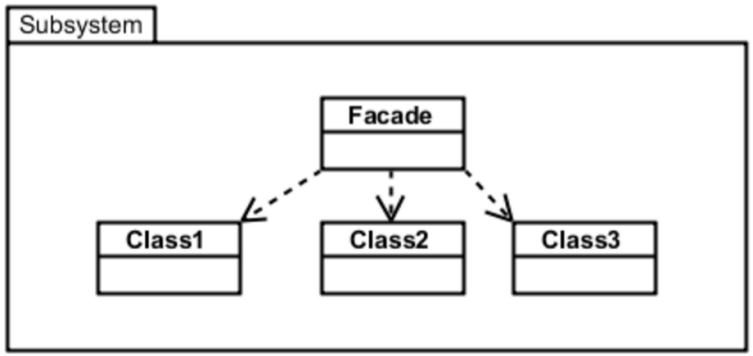
- Перегрузка оператора доступа к членам класса (для C++)
  - Умные указатели так устроены
  - C++ вызывает операторы -> по цепочке
    - object->**do**() может быть хоть ((object.**operator**->()).**operator**->()).**do**()
  - Не подходит, если надо различать операции

“Заместитель” (Proxy), детали реализации (2)

- Реализация “вручную” всех методов проксируемого объекта
  - Сотня методов по одной строчке каждый
  - C#/F#: **public void** do() => realSubject.**do**();
  - Препроцессор/генерация
    - Технологии наподобие WCF
- Проксируемого объекта может не быть в памяти

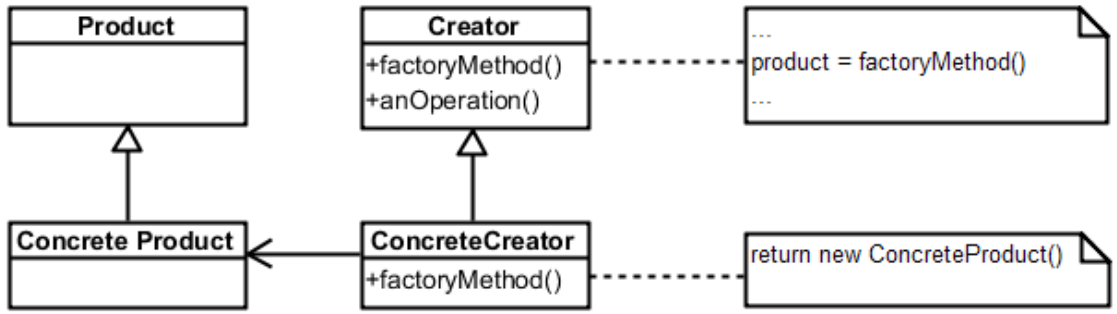
“Фасад” (Facade), детали реализации

- Абстрактный Facade
  - Существенно снижает связность клиента с подсистемой



- Открытые и закрытые классы подсистемы
  - Пространства имён и пакеты помогают, но требуют дополнительных соглашений
    - Пространство имён details
  - Инкапсуляция целой подсистемы — это хорошо

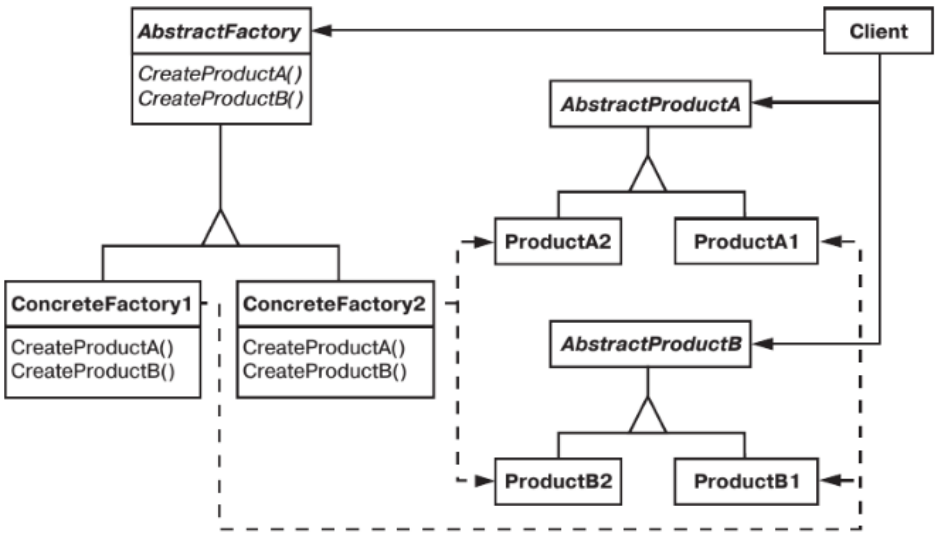
“Фабричный метод” (Factory Method), детали реализации



- Абстрактный Creator или реализация по умолчанию
  - Второй вариант может быть полезен для расширяемости
- Параметризованные фабричные методы
- Если язык поддерживает инстанциацию по прототипу (JavaScript, SmallTalk), можно хранить порождаемый объект
- Creator не может вызывать фабричный метод в конструкторе
- Можно сделать шаблонный Creator

“Абстрактная фабрика” (Abstract Factory), детали реализации

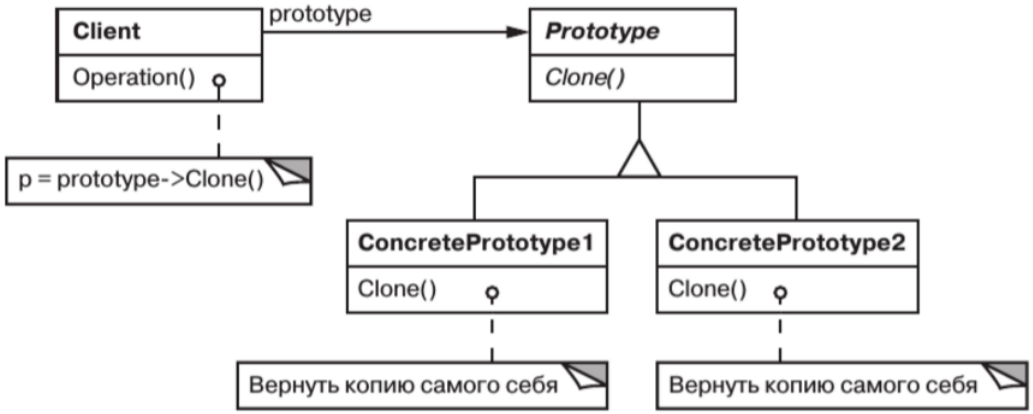
- Хорошо комбинируются с паттерном “Одиночка”
- Если семейств продуктов много, то фабрика может инициализироваться *прототипами*, тогда не надо создавать сотню подклассов



- Прототип на самом деле может быть классом (например, Class в Java)
- Если виды объектов часто меняются, может помочь параметризация метода создания
  - Может пострадать типобезопасность

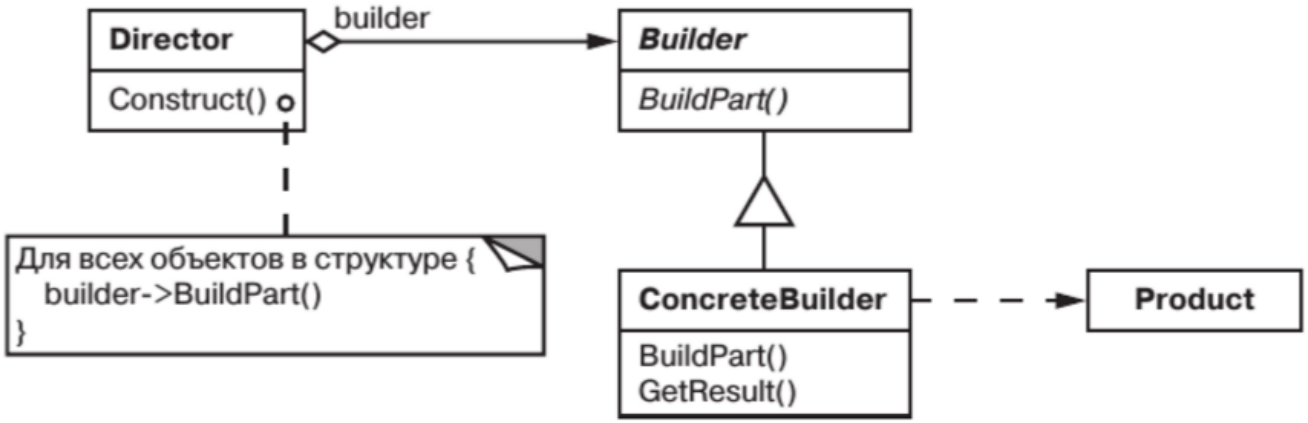
“Прототип” (Prototype), детали реализации

- Паттерн интересен только для языков, где мало runtime-информации о типе (C++)
- Реестр прототипов, обычно ассоциативное хранилище



- Операция Clone
  - Глубокое и мелкое копирование
  - В случае, если могут быть круговые ссылки
  - Сериализовать/десериализовать объект (но помнить про идентичность)
- Инициализация клона
  - Передавать параметры в Clone — плохая идея

“Строитель” (Builder), детали реализации



- Абстрактные и конкретные строители
  - Достаточно общий интерфейс
- Общий интерфейс для продуктов не требуется
  - Клиент конфигурирует распорядителя конкретным строителем, он же и забирает результат
- Пустые методы по умолчанию

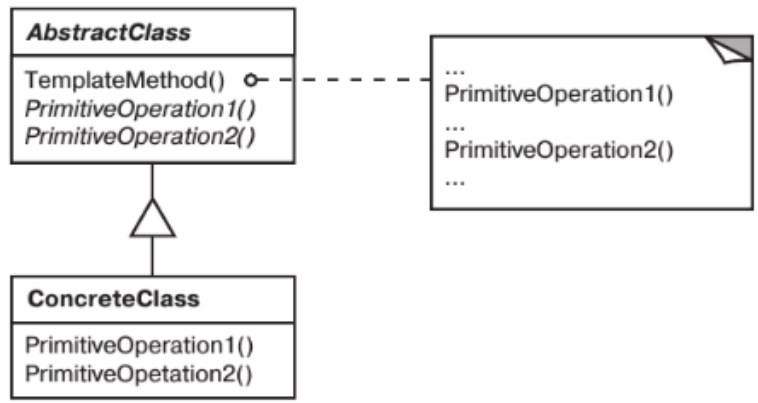
“Строитель”, примеры

- StringBuilder
- Guava, подсистема работы с графами

```
MutableNetwork<Webpage, Link> webSnapshot =
        NetworkBuilder.directed()
    .allowsParallelEdges(true)
    .nodeOrder(ElementOrder.natural())
    .expectedNodeCount(100000)
    .expectedEdgeCount(1000000)
    .build();
```

“Шаблонный метод” (Template Method), детали реализации

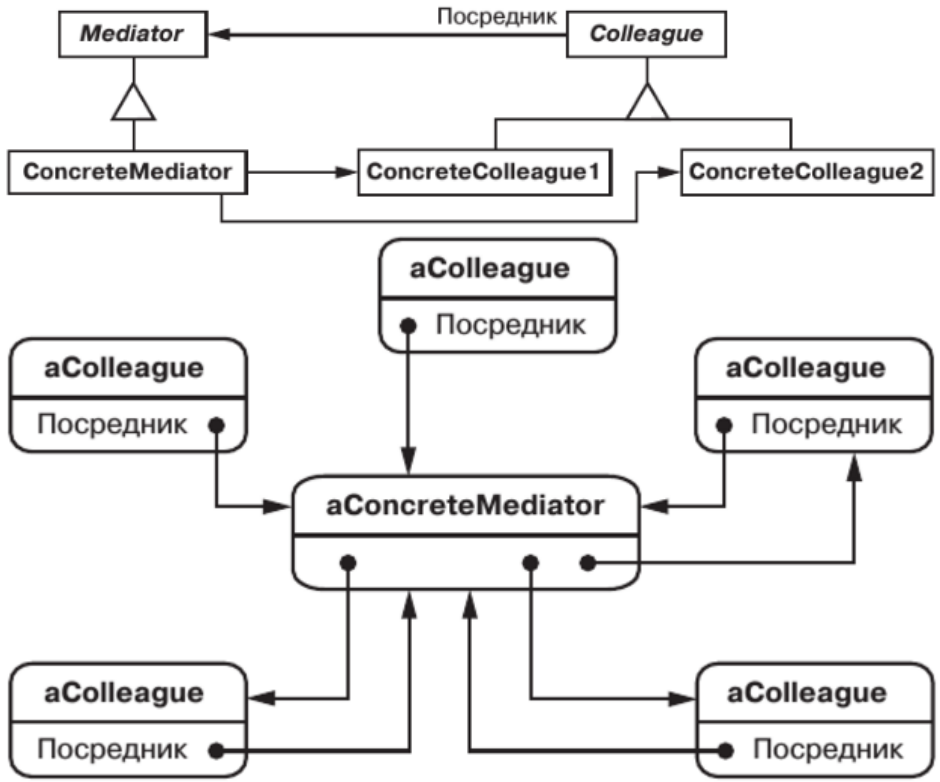
- Сам шаблонный метод, как правило, невиртуальный
- Лучше использовать соглашения об именовании, например, называть операции с Do



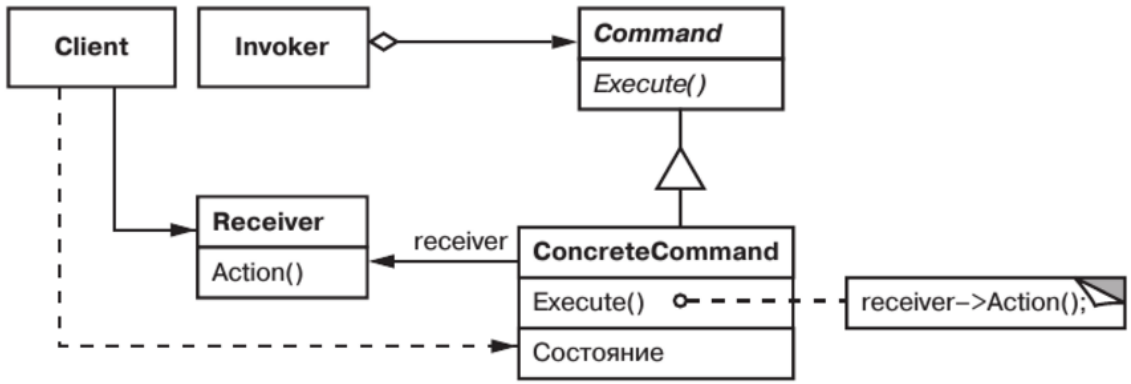
- Примитивные операции могут быть виртуальными или чисто виртуальными
  - Лучше их делать protected
  - Чем их меньше, тем лучше

“Посредник” (Mediator), детали реализации

- Абстрактный класс “Mediator” часто не нужен
- Паттерн “Наблюдатель”: медиатор подписывается на события в коллегах
- Наоборот: коллеги вызывают методы медиатора



“Команда” (Command), детали реализации



- Насколько “умной” должна быть команда
- Отмена и повторение операций — тоже от хранения всего состояния в команде до “вычислимого” отката
  - Undo-стек и Redo-стек
  - Может потребоваться копировать команды
  - “Искусственные” команды
  - Композитные команды
- Паттерн “Хранитель” для избежания ошибок восстановления

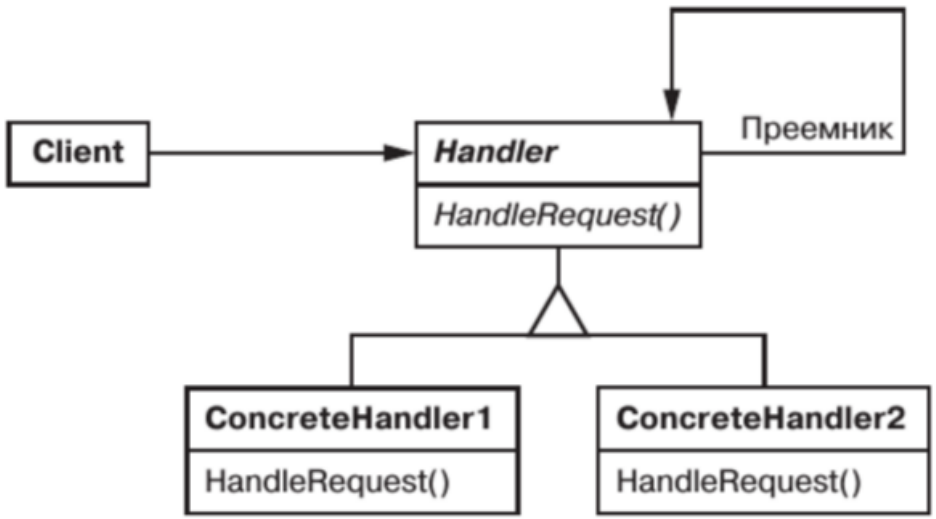
“Команда”, пример

- Qt, класс QAction:

```
const QIcon openIcon = QIcon(":/images/open.png");
QAction *openAct = new QAction(openIcon, tr("&Open..."), this);

openAct->setShortcuts(QKeySequence::Open);
openAct->setStatusTip(tr("Open an existing file"));

connect(openAct, &QAction::triggered, this, &MainWindow::open);

fileMenu->addAction(openAct);
fileToolBar->addAction(openAct);
```

“Цепочка ответственности” (Chain of Responsibility), детали реализации

- Необязательно реализовывать связи в цепочке специально
  - На самом деле, чаще используются существующие связи



- По умолчанию в Handler передавать запрос дальше (если ссылки на преемника всё-таки есть)
- Если возможных запросов несколько, их надо как-то различать
  - Явно вызывать методы — нерасширяемо
  - Использовать объекты-запросы

“Цепочка ответственности”, примеры

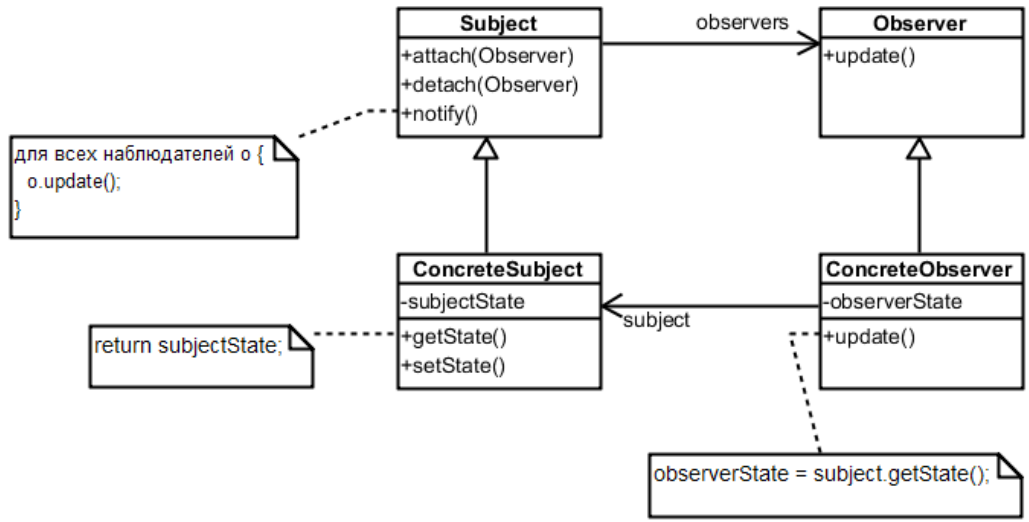
- Распространение исключений
- Распространение событий в оконных библиотеках:

```
void MyCheckBox::mousePressEvent(QMouseEvent *event)
{
    if (event->button() == Qt::LeftButton) {
        // handle left mouse button here
    } else {
        // pass on other buttons to base class
        QCheckBox::mousePressEvent(event);
    }
}
```

“Наблюдатель” (Observer), детали реализации

- В “нормальных” языках поддержан “из коробки” (через механизм событий)

- Могут использоваться хеш-таблицы для отображения субъектов и наблюдателей
  - Так делает WPF в .NET, есть даже языковая поддержка в C#



- Необходимость идентифицировать субъект
- Кто инициирует нотификацию
  - Операции, модифицирующие субъект
  - Клиент, после серии модификаций субъекта

“Наблюдатель” (Observer), детали реализации (2)

- Ссылки на субъектов и наблюдателей
  - Простой способ организовать утечку памяти в C# или грохнуть программу в C++
- Консистентность субъекта при отправке нотификации
  - Очевидно, но легко нарушить, вызвав метод предка в потомке
  - “Шаблонный метод”
  - Документировать, кто когда какие события бросает
- Передача сути изменений — pull vs push
- Фильтрация по типам событий
- Менеджер изменений (“Посредник”)

“Наблюдатель”, пример (1)

- События в C#:

```
internal class NewMessageEventArgs : EventArgs {
    private readonly string message;
    public MessageEventArgs(string message)
        => this.message = message;
    public string Message => message;
}
```

“Наблюдатель”, пример (2)

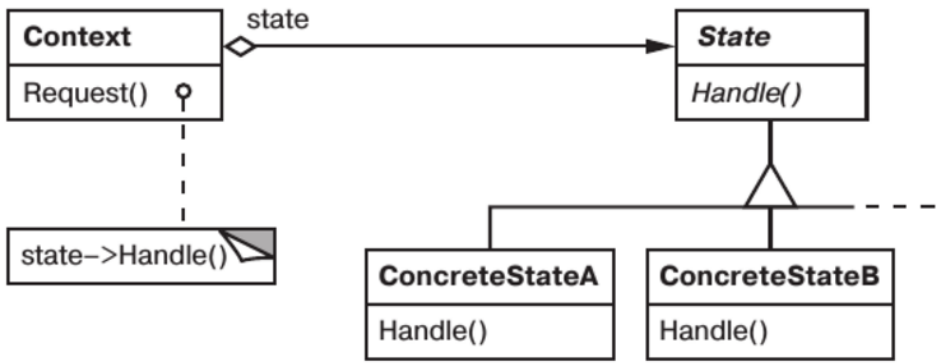
```
internal class Messenger {
    public event EventHandler<NewMessageEventArgs> NewMessage;
    protected virtual void OnMessage(NewMessageEventArgs e) {
        EventHandler<NewMessageEventArgs> temp
                = Volatile.Read(ref NewMessage);
        if (temp != null)
            temp(this, e);
    }
    public void SimulateMessage(String message) {
        NewMessageEventArgs e = new NewMessageEventArgs(message);
        OnMessage(e);
    }
}
```

“Наблюдатель”, пример (3)

```
internal sealed class Fax {
    public Fax(Messenger mm) => mm.NewMessage += FaxMsg;

    private void FaxMsg(object sender, NewMessageEventArgs e) {
        Console.WriteLine("Faxing message:");
        Console.WriteLine($"Message={e.Message}");
    }

    public void Unregister(Messenger mm)
            => mm.NewMessage -= FaxMsg;
}
```

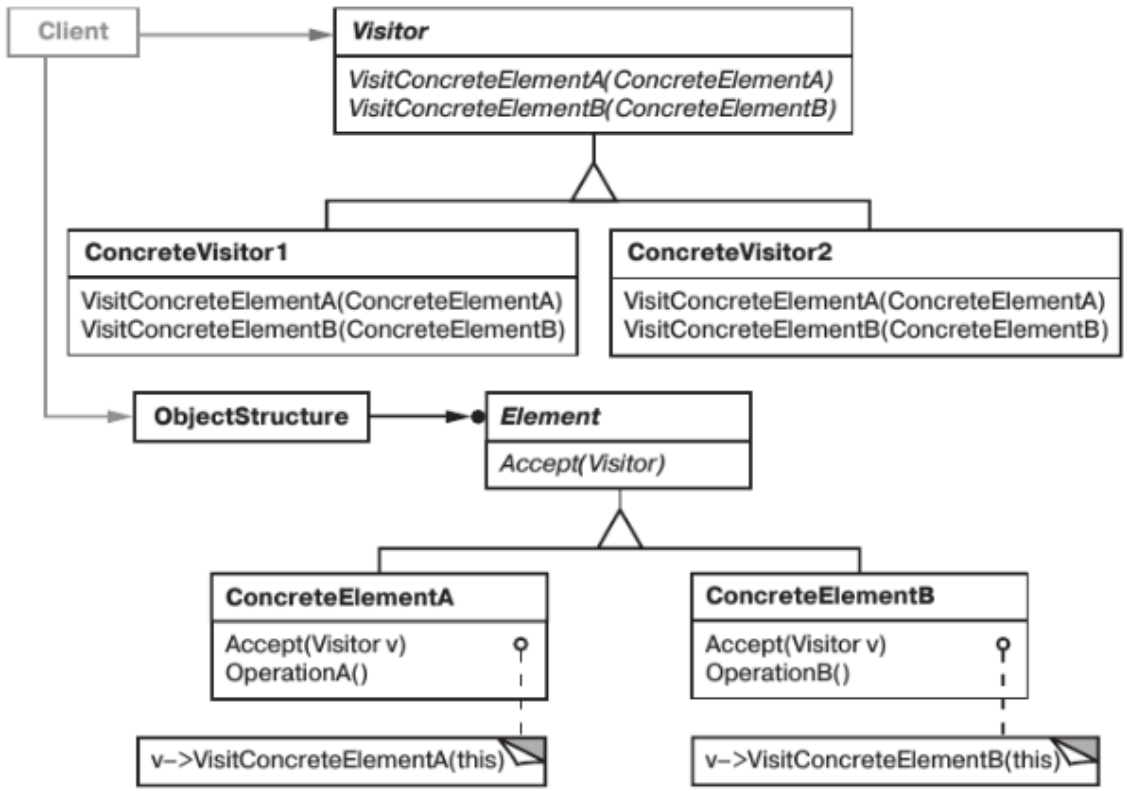
“Состояние” (State), детали реализации

- Переходы между состояниями — в Context или в State?
- Таблица переходов
  - Трудно добавить действия по переходу
- Создание и уничтожение состояний
  - Создать раз и навсегда
  - Создавать и удалять при переходах

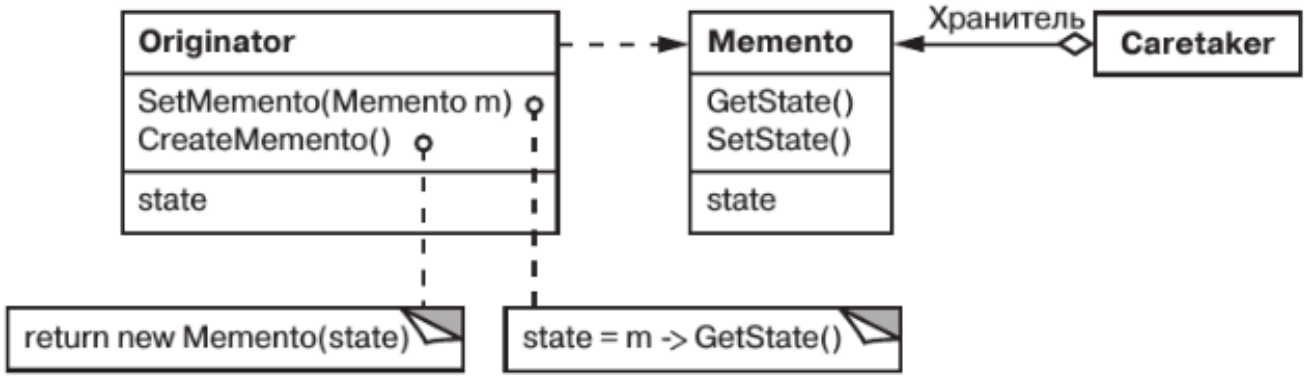


“Посетитель” (Visitor), детали реализации

- Использовать перегрузку методов Visit(...)
- Чаще всего сама коллекция отвечает за обход, но может быть итератор
- Может даже сам Visitor, если обход зависит от результата операции



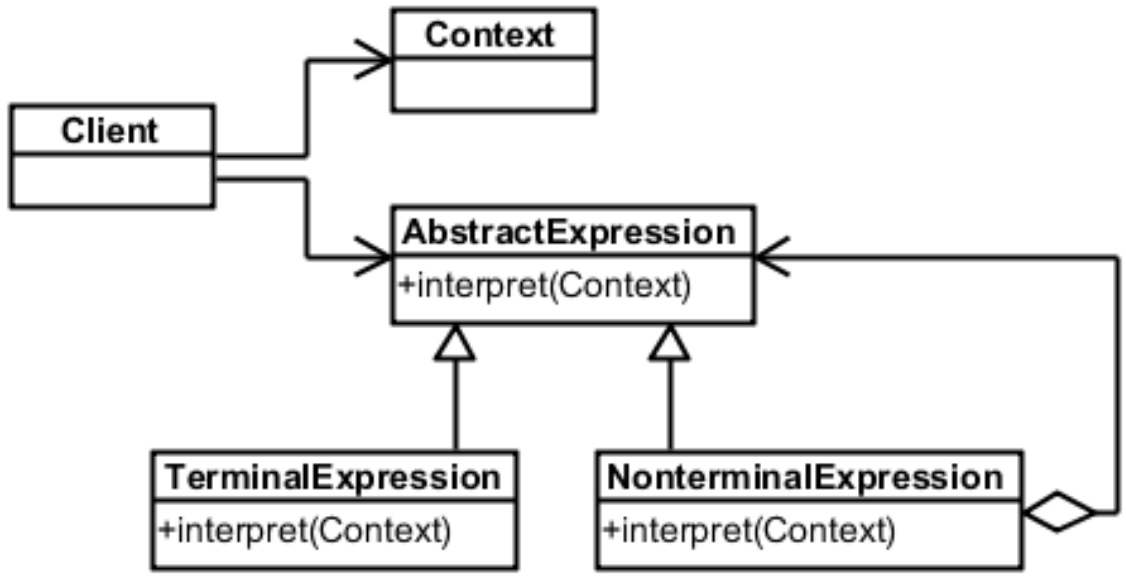
“Хранитель” (Memento), детали реализации



- Два интерфейса: “широкий” для хозяев и “узкий” для остальных объектов
  - Требуется языковая поддержка
- Можно хранить только дельты состояний

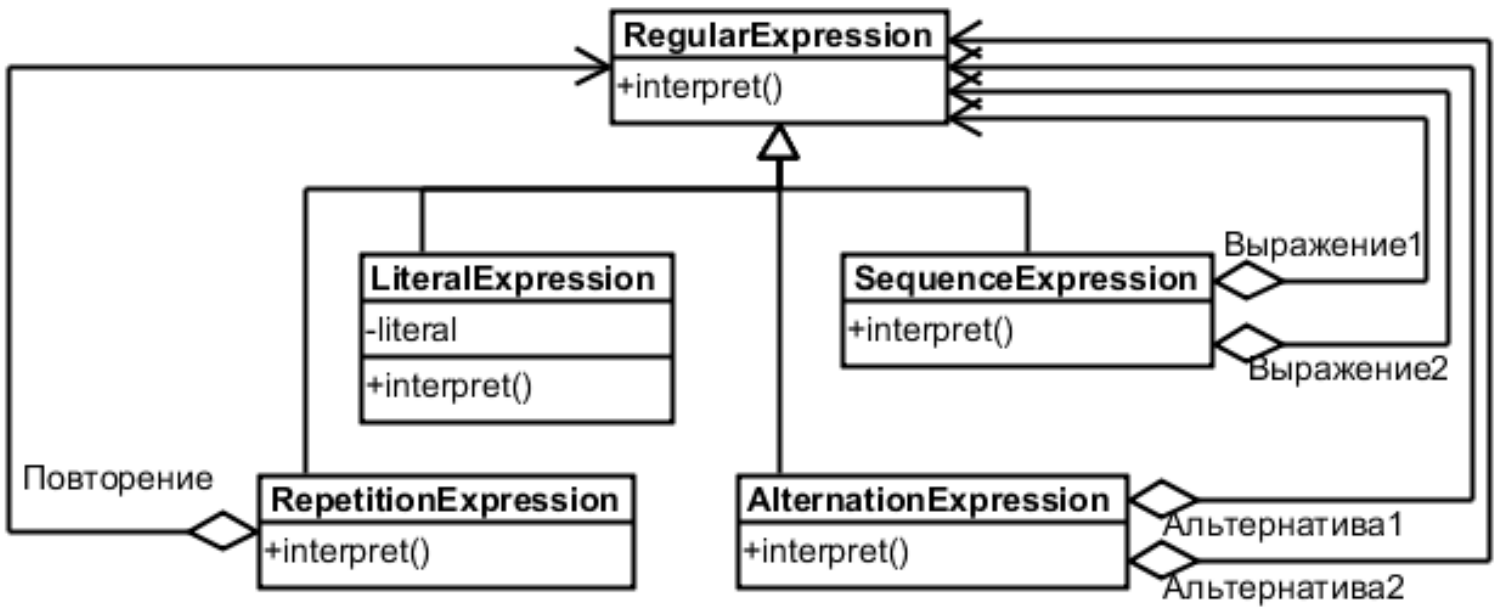
“Интерпретатор” (Interpreter)

Определяет представление грамматики и интерпретатор для заданного языка.



- Грамматика должна быть проста (иначе лучше “Visitor”)
- Эффективность не критична

“Интерпретатор”, пример



“Интерпретатор”, детали реализации

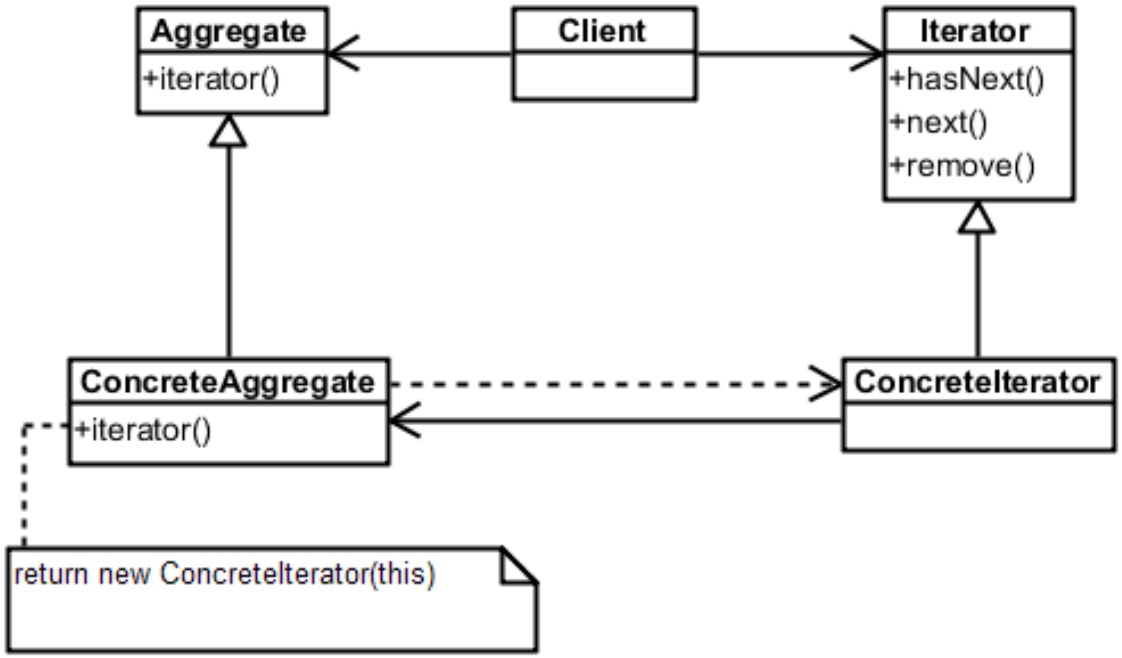
10-е правило Гринспена:

Любая достаточно сложная программа на Си или Фортране содержит заново написанную, неспецифицированную, глючную и медленную реализацию половины языка Common Lisp

- Построение дерева — отдельная задача
- Несколько разных операций над деревом — лучше “Visitor”
- Можно использовать “Приспособленец” для разделения терминальных символов

“Итератор” (Iterator)

Инкапсулирует способ обхода коллекции.



- Разные итераторы для разных способов обхода
- Можно обходить не только коллекции

“Итератор”, примеры

- Java-стиль:

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();
}
```

- .NET-стиль:

```
public interface IEnumerator<T>
{
    bool MoveNext();
    T Current { get; }
    void Reset();
}
```

“Итератор”, детали реализации (1)

- Внешние итераторы

```
foreach (Thing t in collection)
{
    Console.WriteLine(t);
}
```

- Внутренние итераторы

```
collection.ToList().ForEach(t => Console.WriteLine(t));
```

“Итератор”, детали реализации (2)

- Итераторы и курсоры
- Устойчивые и неустойчивые итераторы
  - Паттерн “Наблюдатель”
  - Даже обнаружение модификации коллекции может быть непросто
- Дополнительные операции
- В C++ итераторы — это сложно