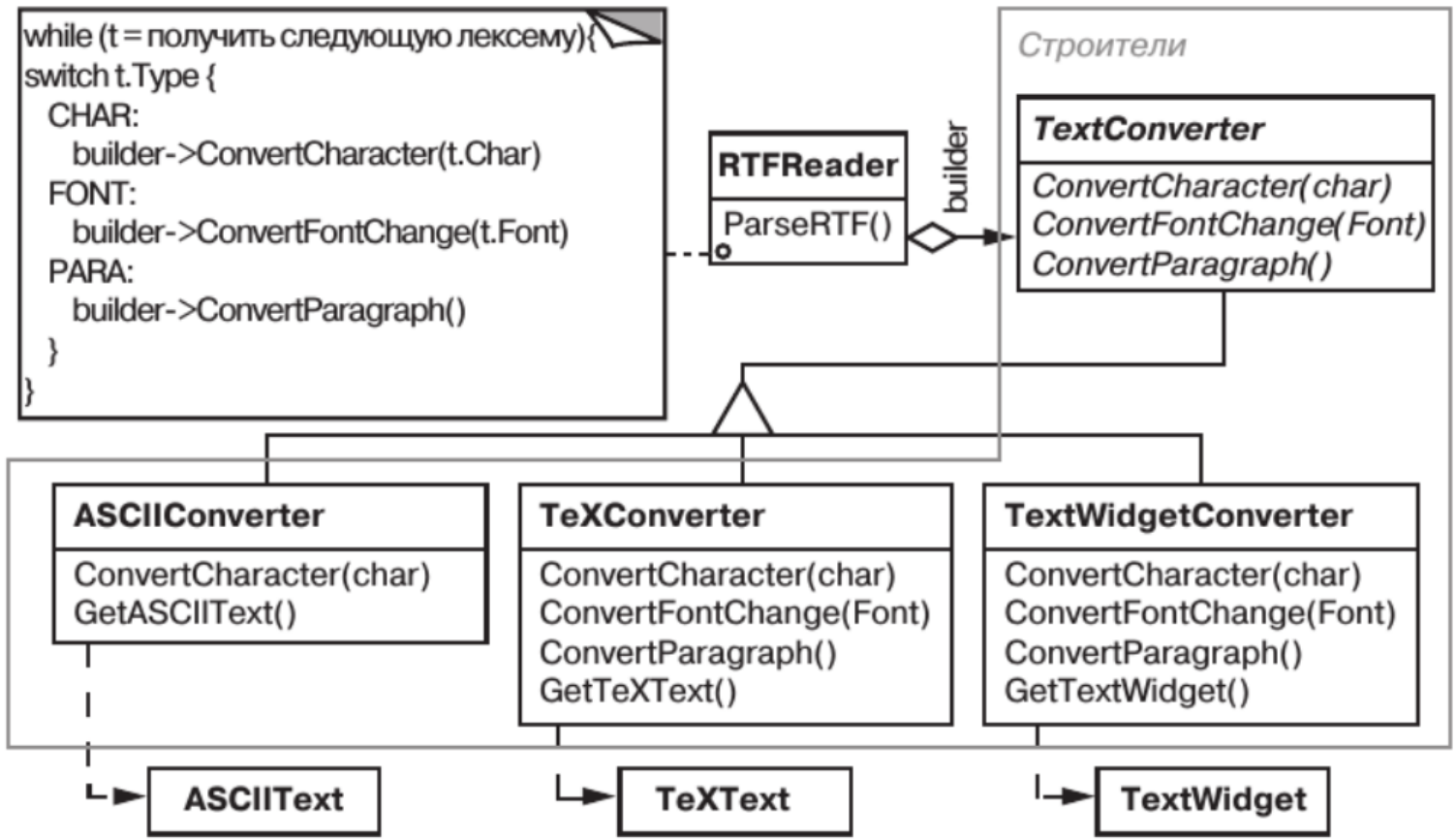
Лекция 8: Поведенческие шаблоны

Юрий Литвинов
y.litvinov@spbu.ru

03.04.2023

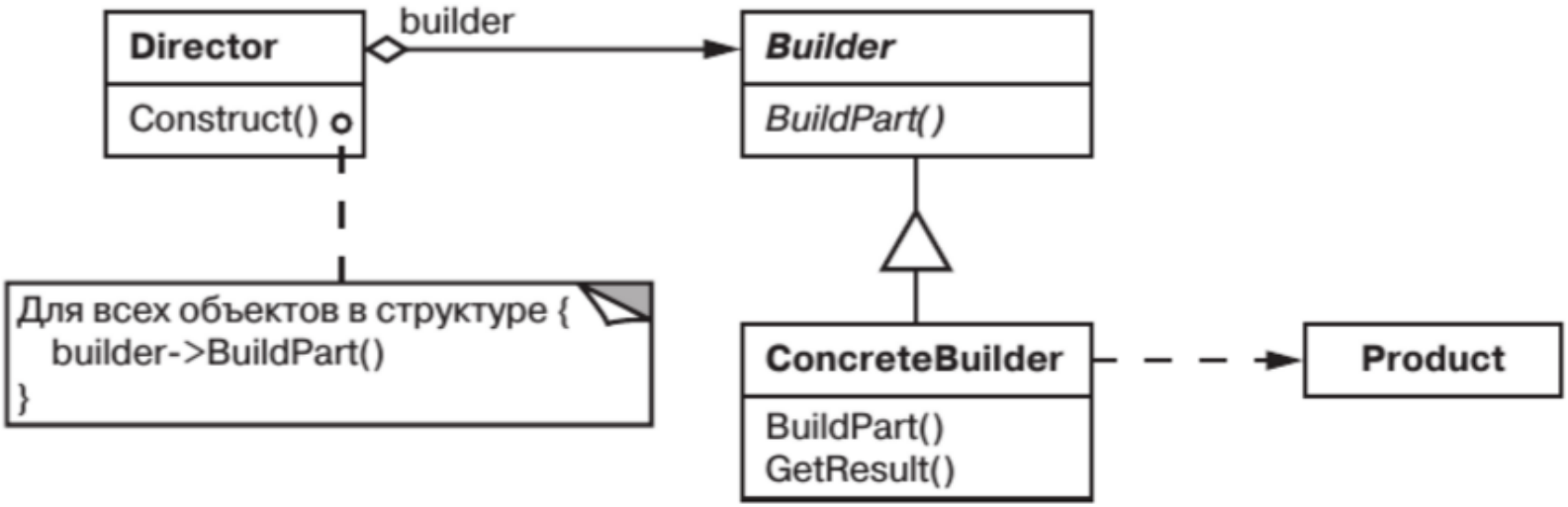
“Строитель”, мотивация

Конвертер текста

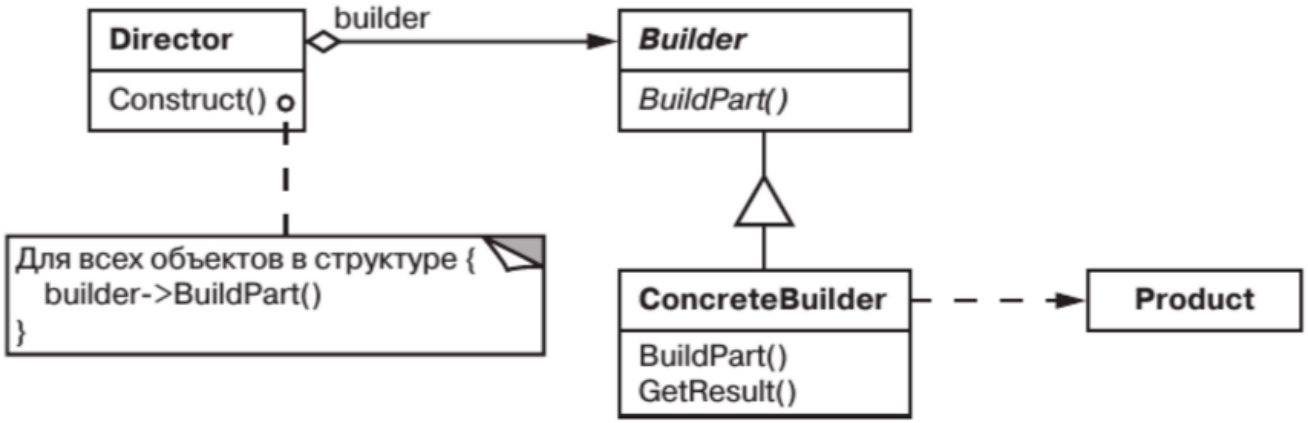


Патерн “Строитель”

Builder



“Строитель” (Builder), детали реализации



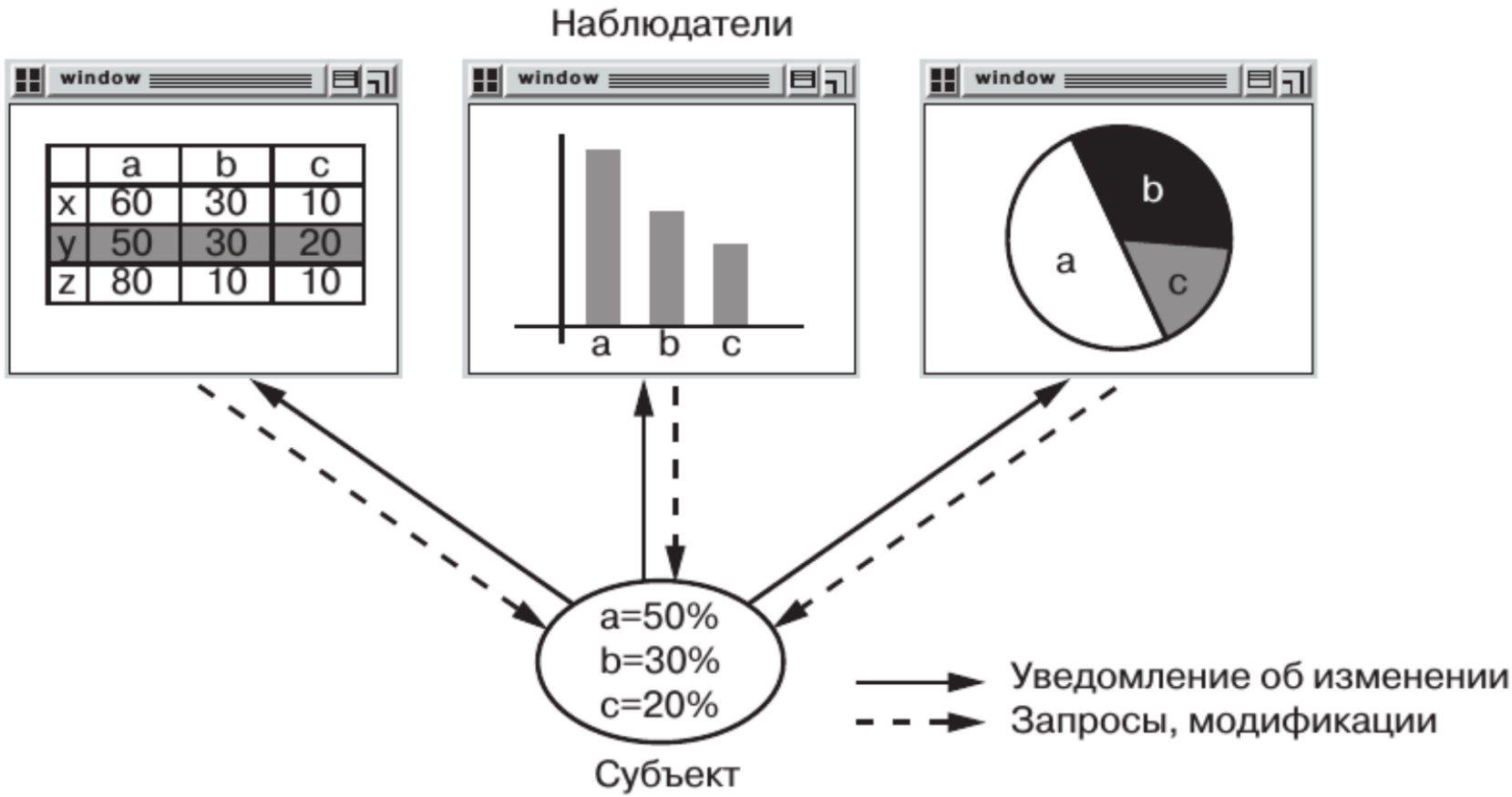
- Абстрактные и конкретные строители
  - Достаточно общий интерфейс
- Общий интерфейс для продуктов не требуется
  - Клиент конфигурирует распорядителя конкретным строителем, он же и забирает результат
- Пустые методы по умолчанию

“Строитель”, примеры

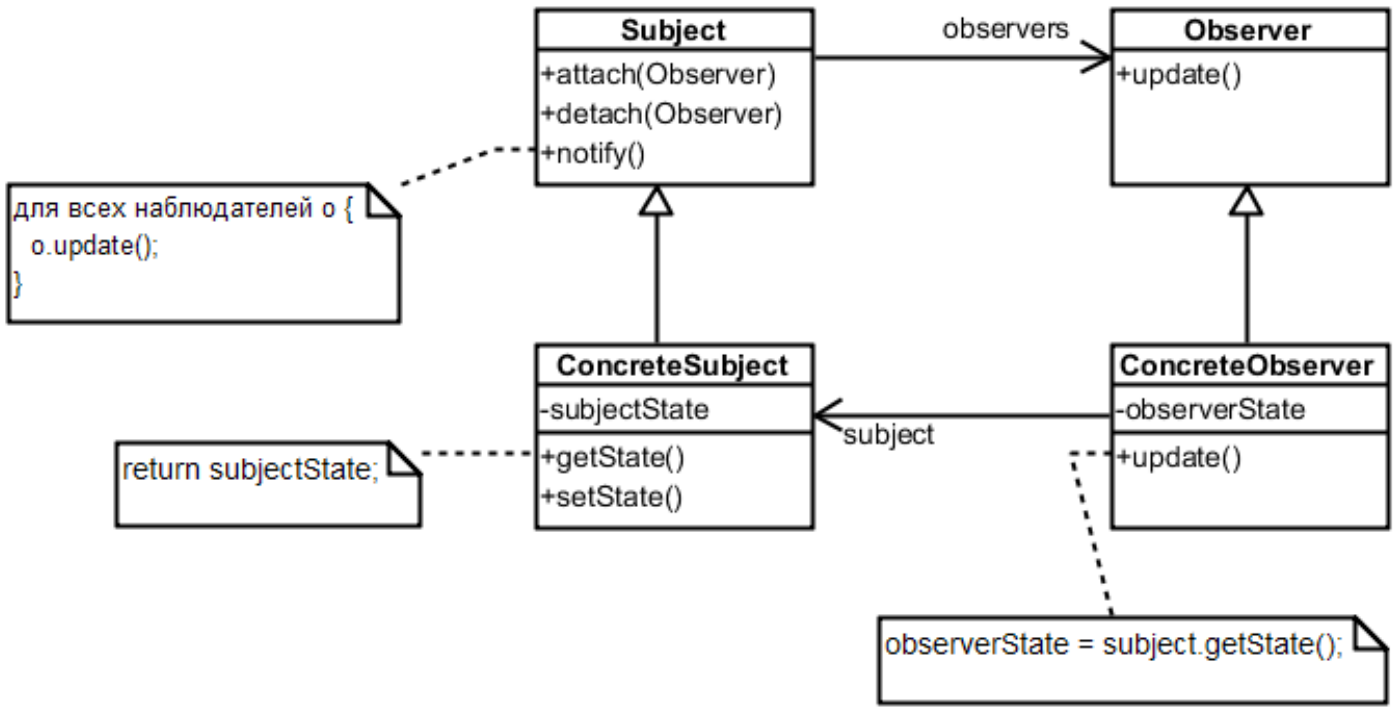
- StringBuilder
- Guava, подсистема работы с графами

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

Паттерн “Наблюдатель”, мотивация



Паттерн “Наблюдатель”



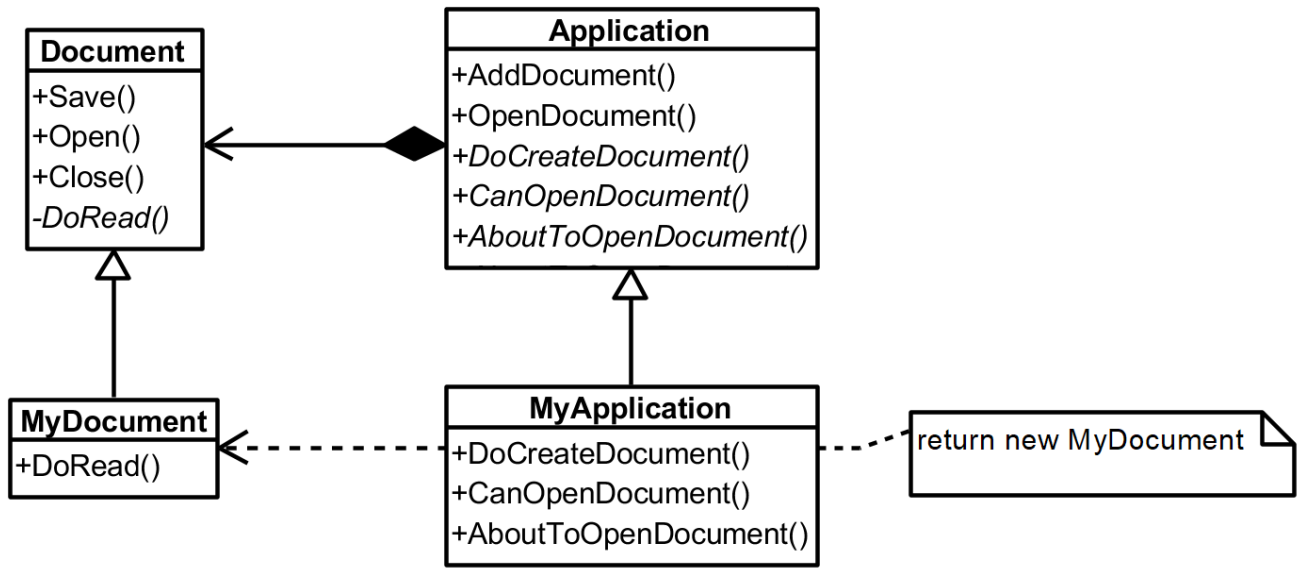
“Наблюдатель” (Observer), детали реализации

- Во многих 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”)

Паттерн “Шаблонный метод”, мотивация



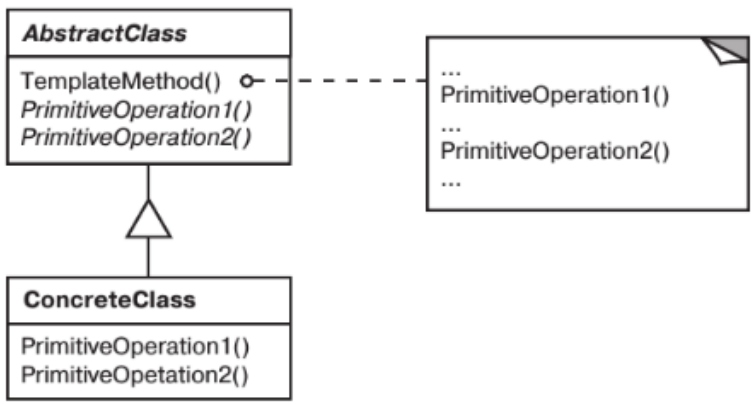
- Алгоритм, общий для всех потомков
- Детали реализации операций — в потомках
- Задание точек расширения

Шаблонный метод, пример

```
void Application::OpenDocument(const char* name) {
  if (!CanOpenDocument(name)) {
    return;
  }

  Document* doc = DoCreateDocument();

  if (doc) {
    _docs->AddDocument(doc);
    AboutToOpenDocument(doc);
    doc->Open();
    doc->DoRead();
  }
}
```

“Шаблонный метод” (Template Method), детали реализации

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

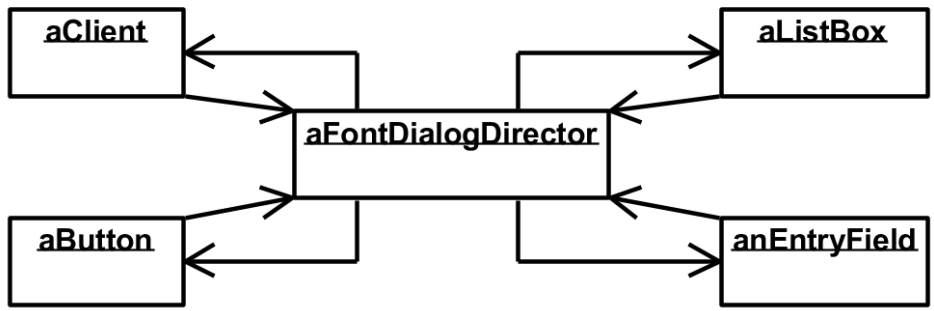


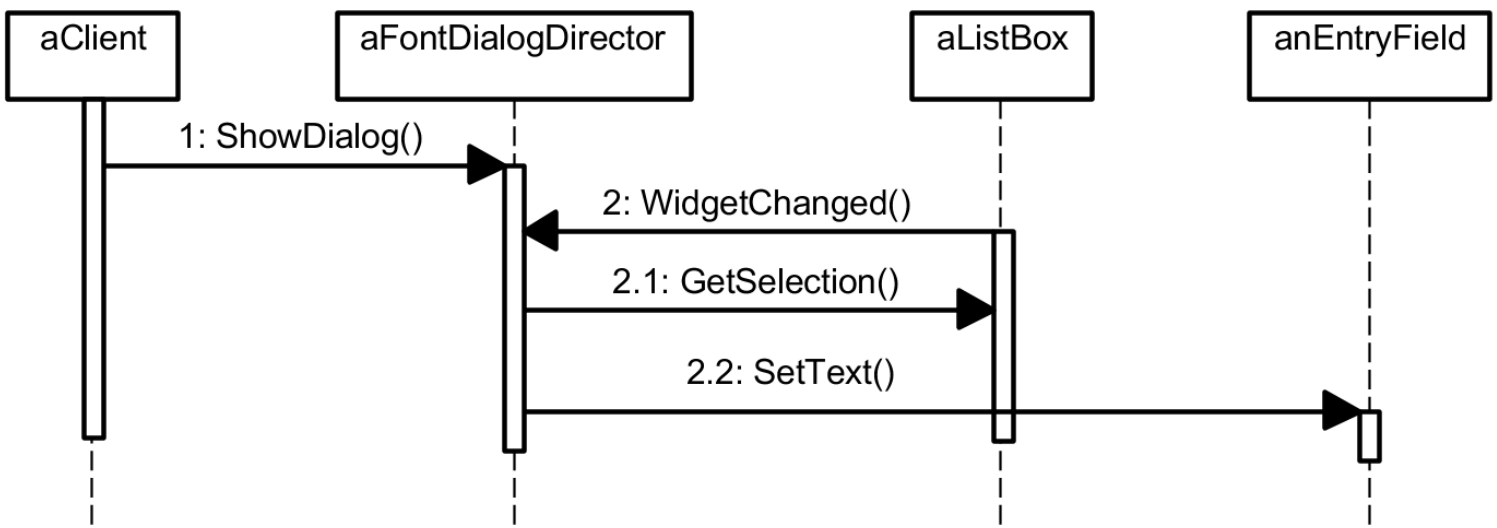
- Примитивные операции могут быть виртуальными или чисто виртуальными
  - Лучше их делать protected
  - Чем их меньше, тем лучше

“Посредник” (Mediator), мотивация

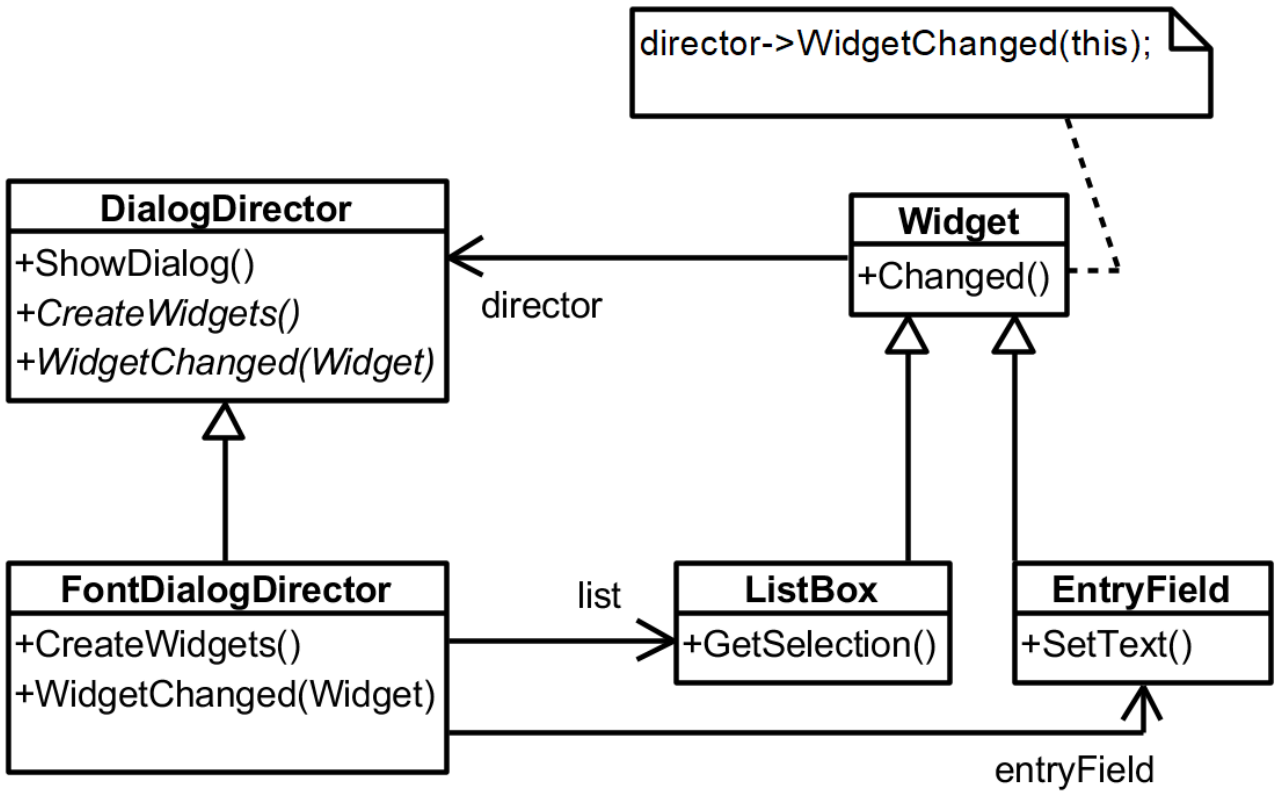
- Большое количество связей между объектами
- Объекты знают слишком много
- Снижается переиспользуемость

Решение: централизация



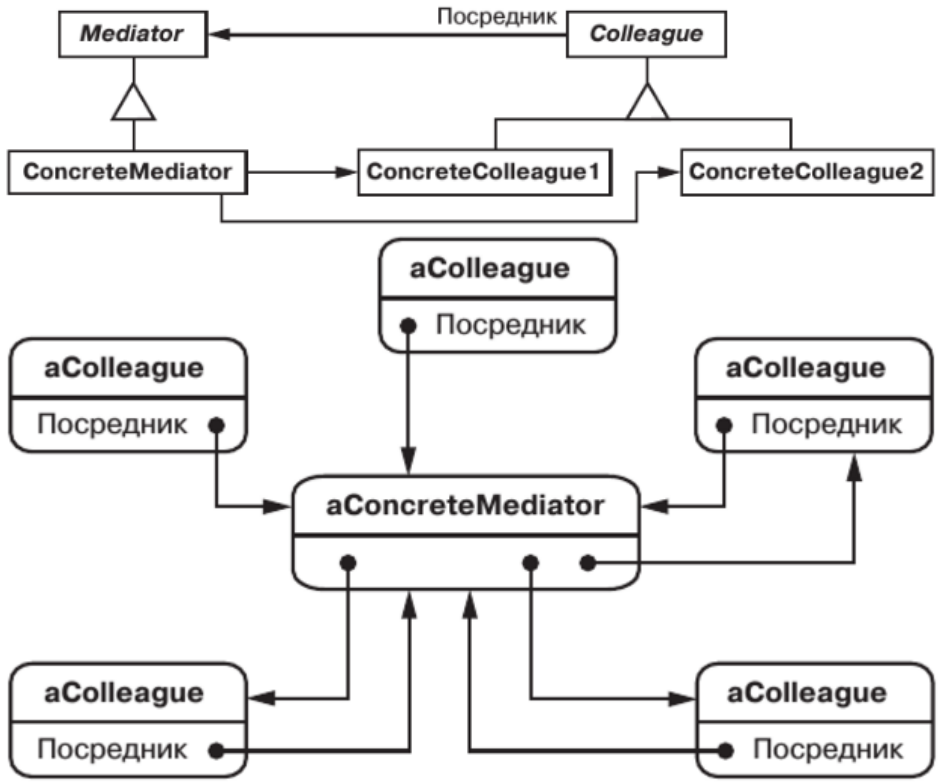


Что получилось



“Посредник” (Mediator), детали реализации

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



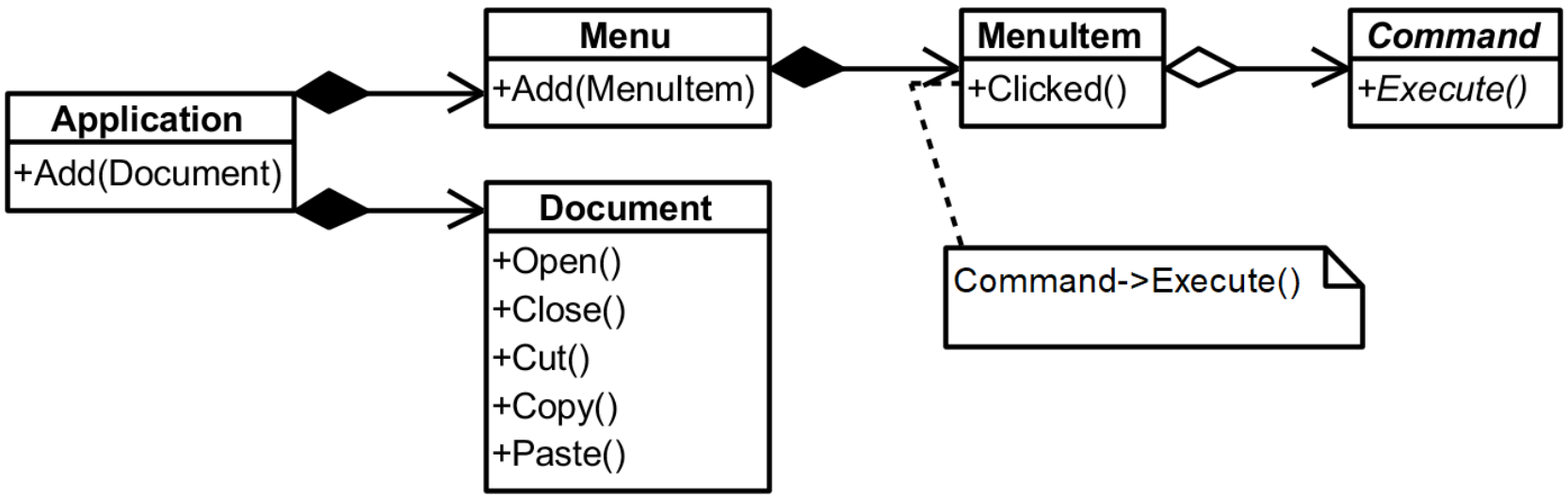
Посредник, достоинства и недостатки

- Устраняет связанность между классами-коллегами
- Повышает переиспользуемость классов-коллег
- Упрощает протоколы взаимодействия объектов
- Абстрагирует способ кооперирования объектов
- Централизует управление (потенциальный God Object!)

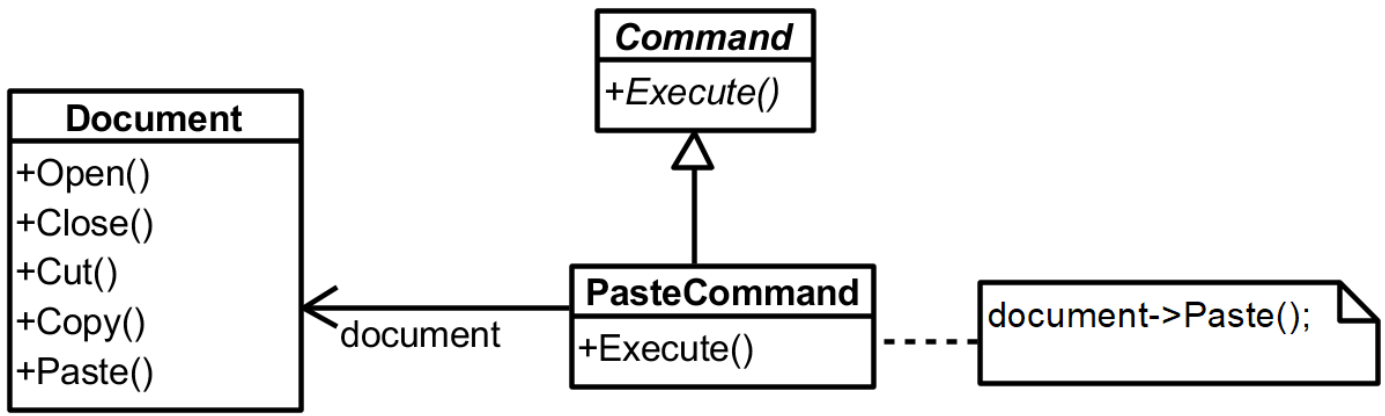
Паттерн “Команда”, мотивация

- Хотим отделить инициацию запроса от его исполнения
- Хотим, чтобы тот, кто “активирует” запрос, не знал, как он исполняется
- При этом хотим, чтобы тот, кто знает, когда исполнится запрос, не знал, когда он будет активирован
- Но зачем?
  - Команды меню приложения
  - Палитры инструментов
  - ...
- “Просто вызвать действие” не получится, вызов функции жёстко свяжет инициатора и исполнителя

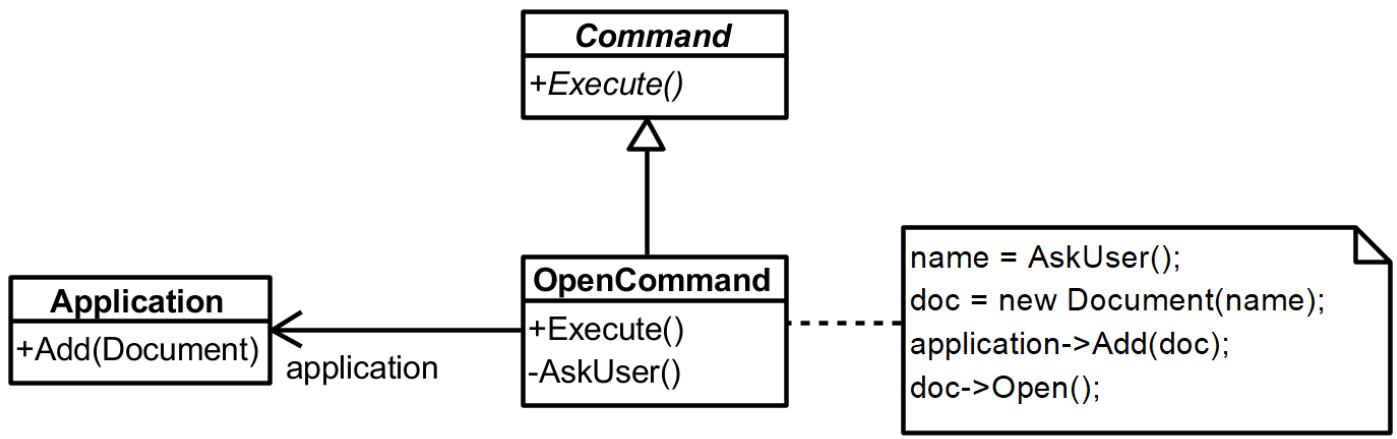
Решение: обернём действие в объект



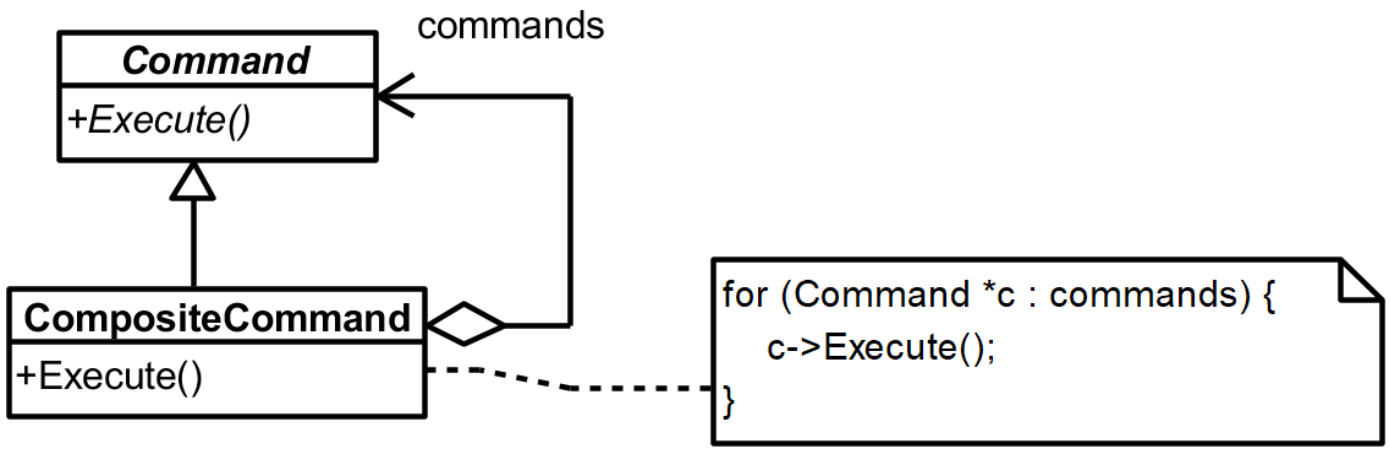
Команда вставки



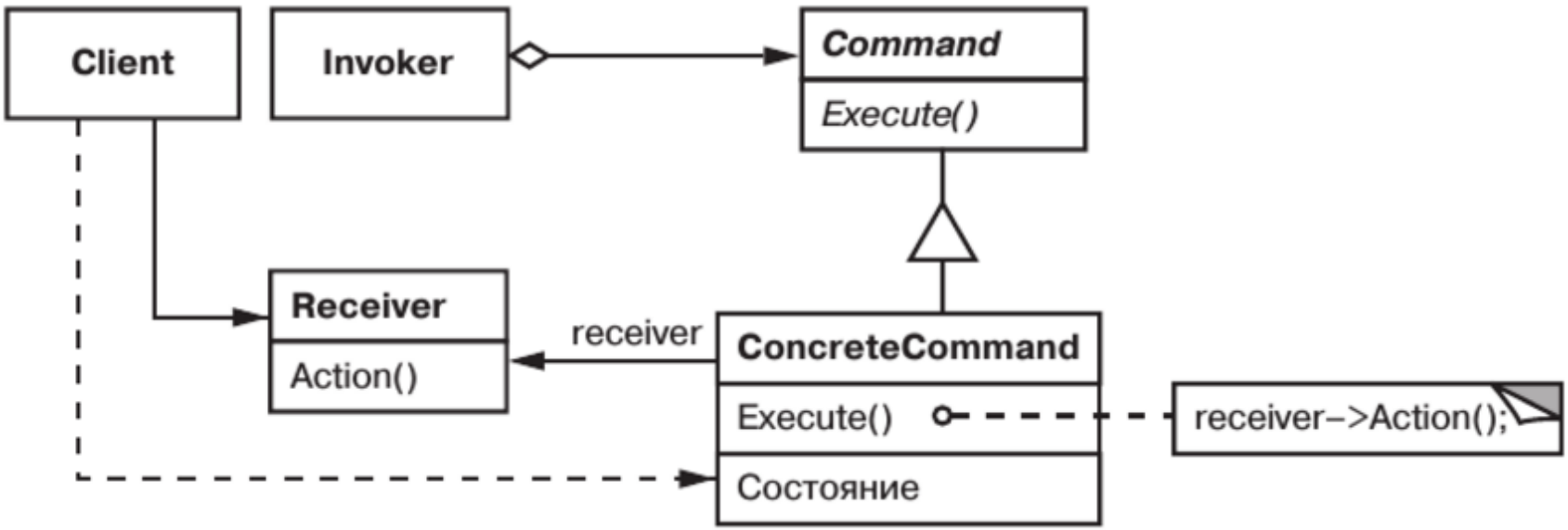
Команда открытия документа



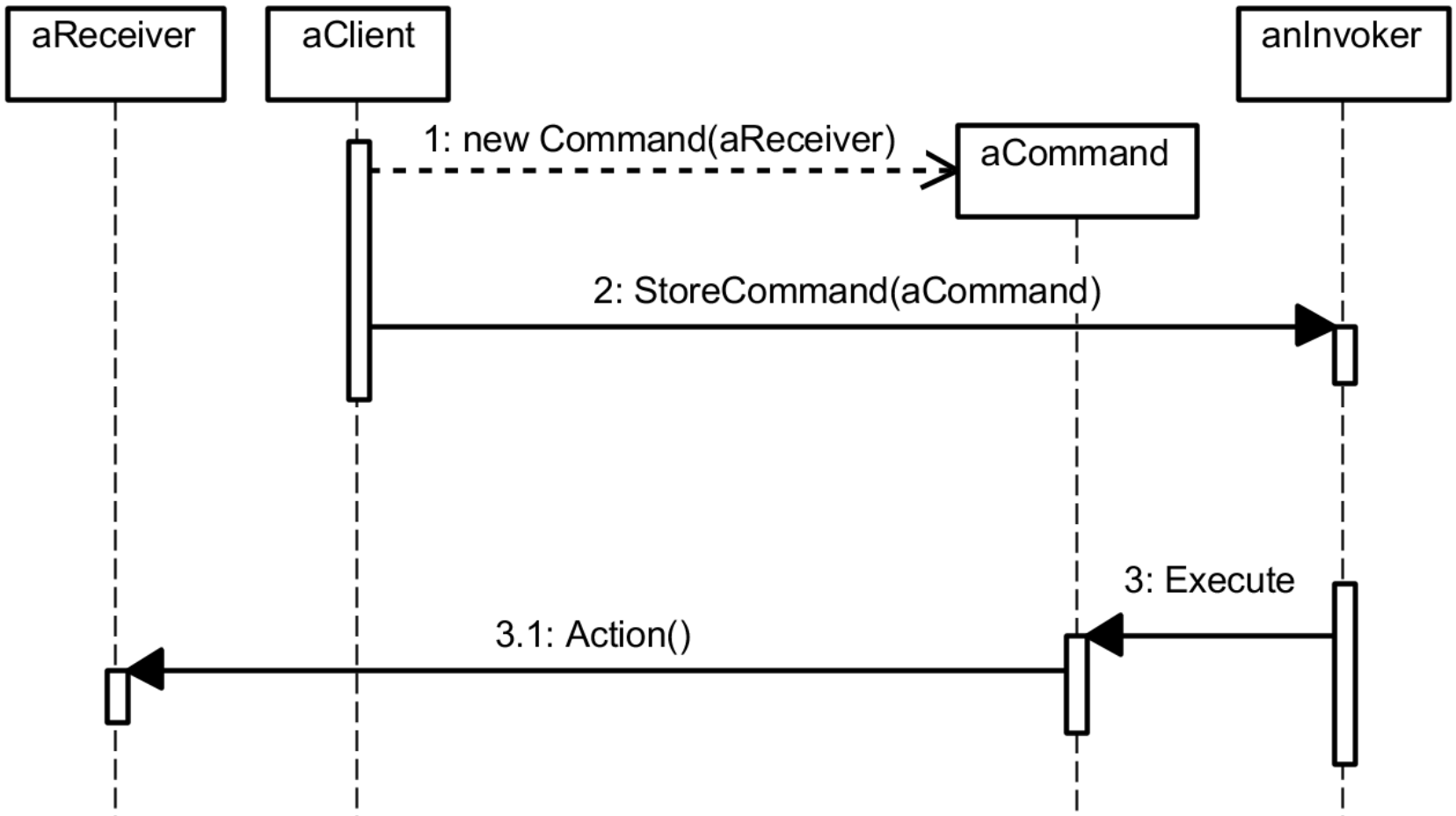
Составная команда



Паттерн “Команда”



Взаимодействие объектов



Команда, применимость

- Параметризовать объекты выполняемым действием
- Определять, ставить в очередь и выполнять запросы в разное время
- Поддержать отмену операций
- Структурировать систему на основе высокоуровневых операций, построенных из примитивных
- Поддержать протоколирование изменений

“Команда” (Command), детали реализации

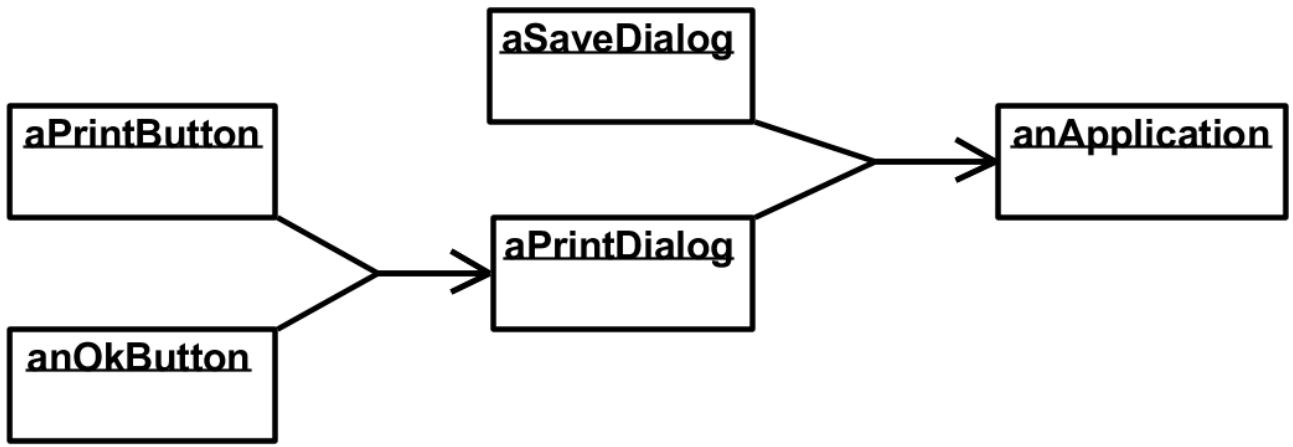
- Насколько “умной” должна быть команда
- Отмена и повторение операций — тоже от хранения всего состояния в команде до “вычислимого” отката
  - Undo-стек и Redo-стек
  - Может потребоваться копировать команды
  - “Искусственные” команды
  - Композитные команды
- Паттерн “Хранитель” для избежания ошибок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

“Команда”, пример

- Qt, класс QAction:

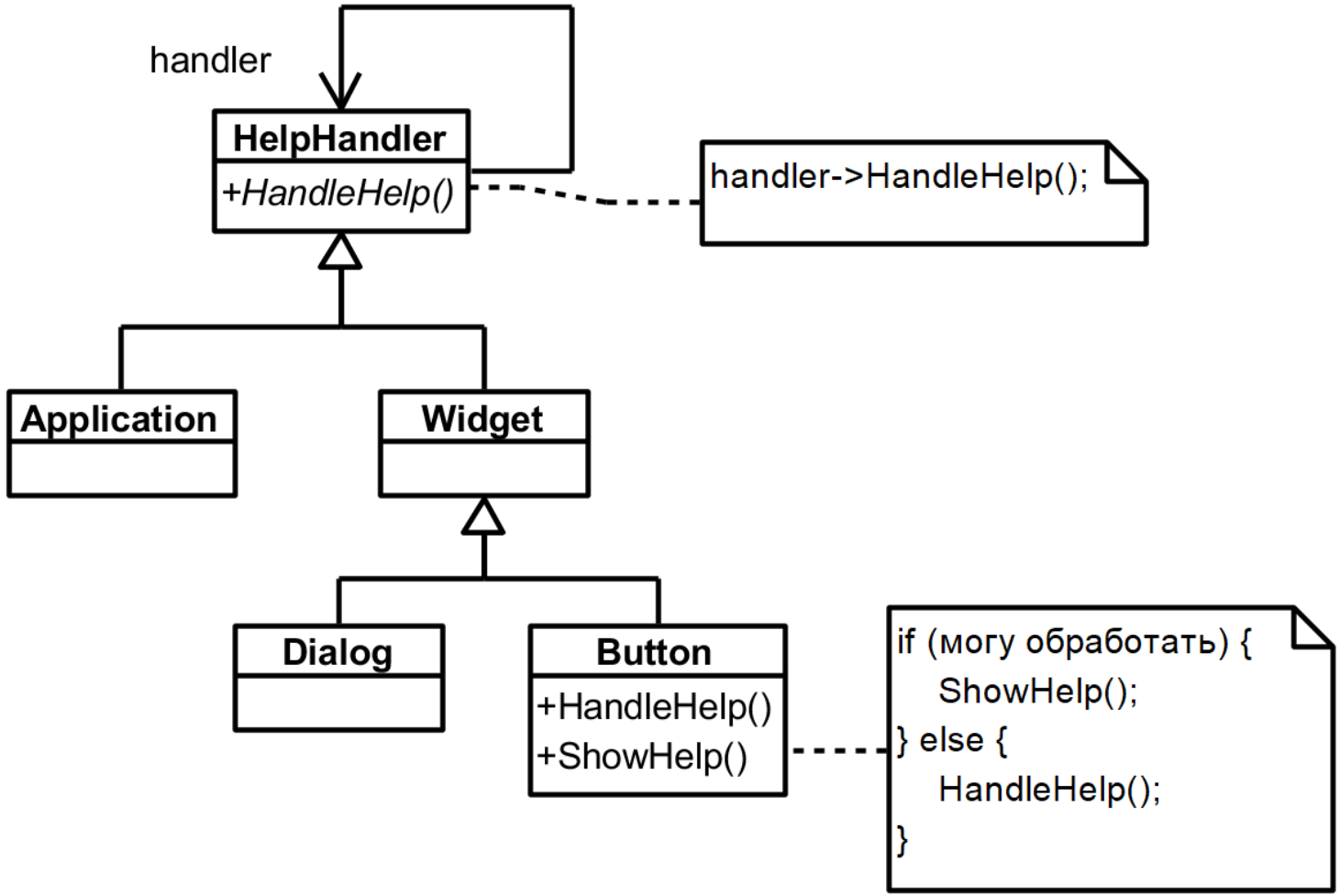
```
const QIcon openIcon = QIcon(":/images/open.png");
QAction *openAct
    = new QAction(openIcon, tr("&Open..."), this);

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

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

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

Паттерн “Цепочка ответственности”, мотивация



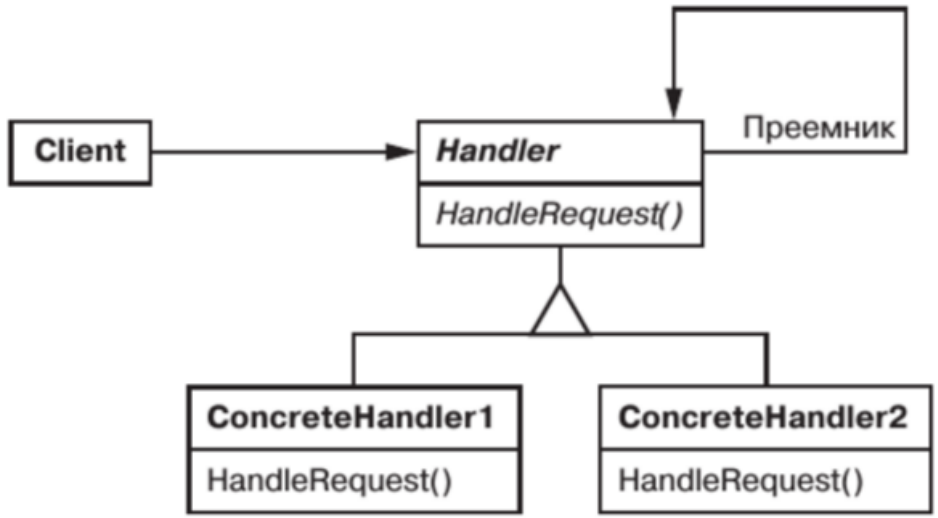
- Организация контекстной справки
- Если у элемента справки нет, запрос передаётся контейнеру
- Заранее неизвестно, кто в итоге обработает запрос

Как это выглядит на диаграмме классов



“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”, плюсы и минусы

- Ослабление связанности
- Дополнительная гибкость при распределении обязанностей
- Получение не гарантировано

Когда использовать:

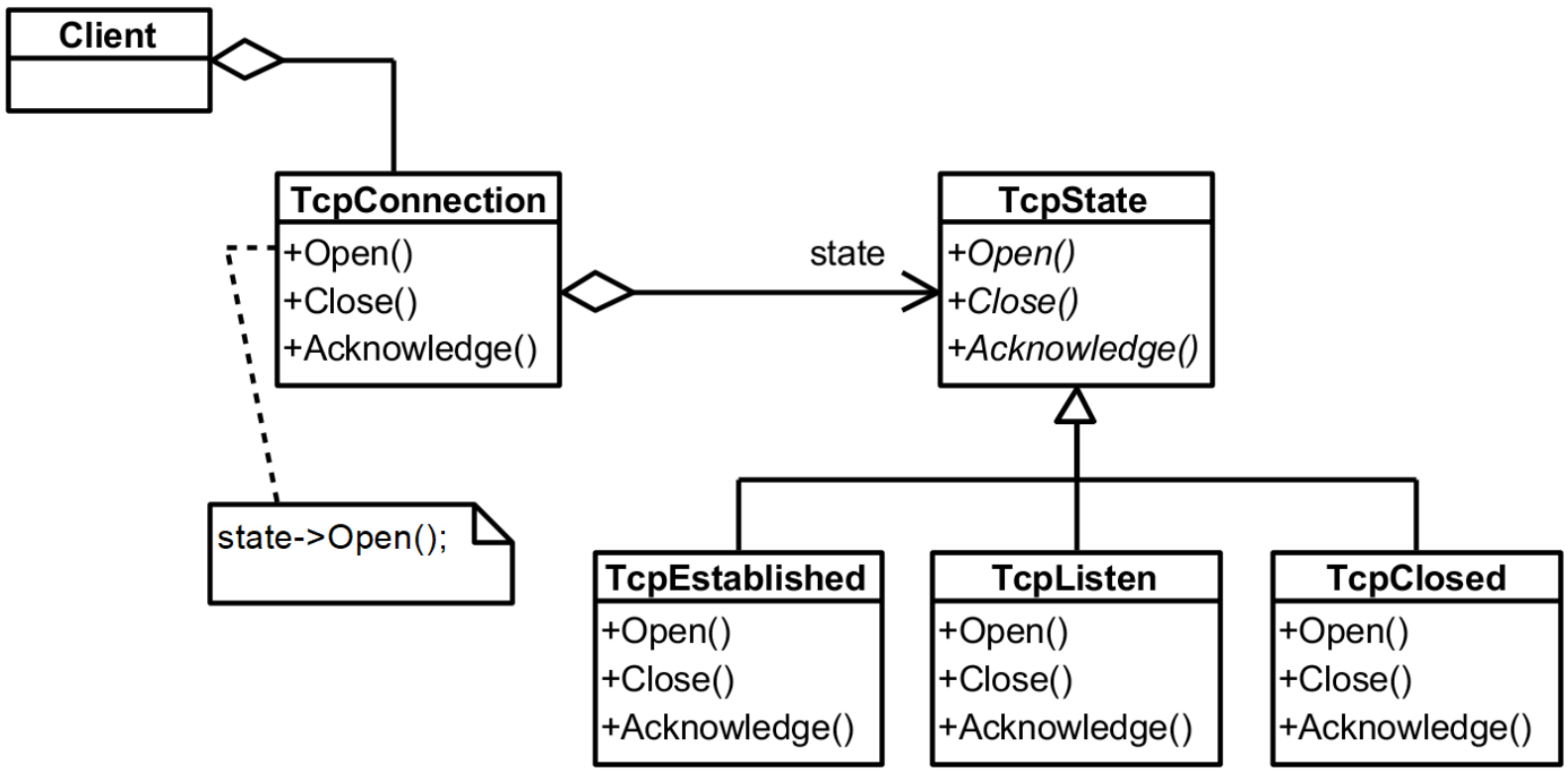
- Есть более одного объекта-обработчика запросов
- Конечный обработчик неизвестен и должен быть найден автоматически
- Хотим отправить запрос нескольким объектам
- Обработчики могут задаваться динамически

“Цепочка ответственности”, примеры

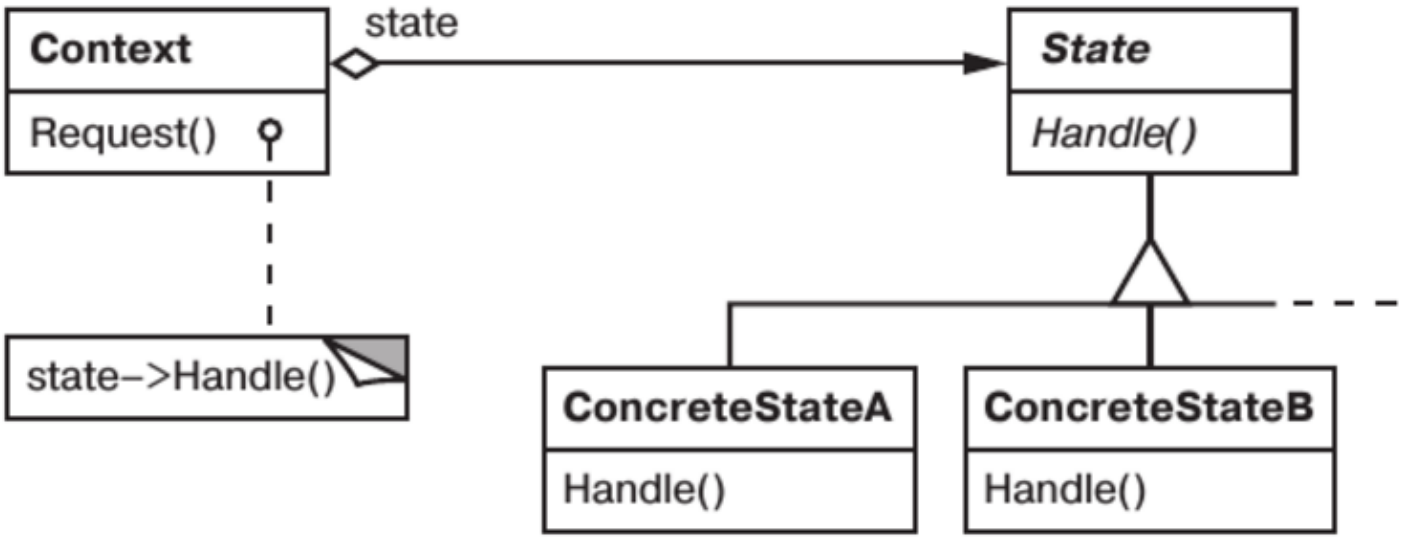
- Распространение исключений
- Распространение событий в оконных библиотеках:

```
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);
    }
}
```

Паттерн “Состояние”, мотивация



Паттерн “Состояние”



“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

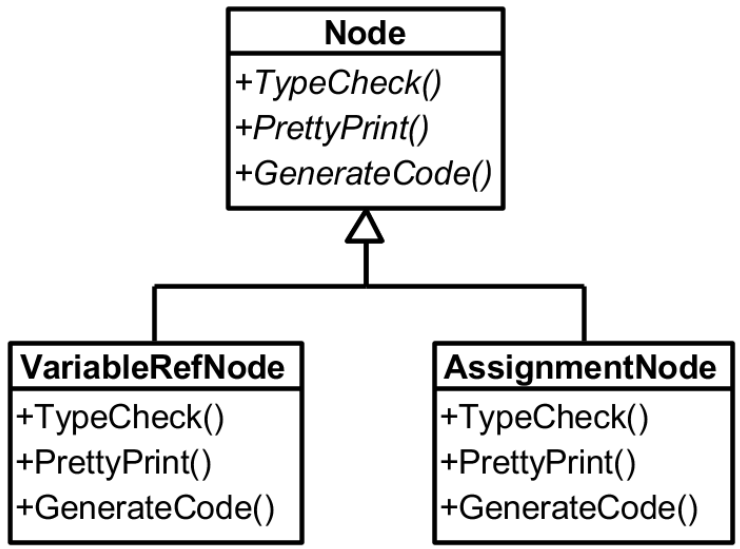
“Состояние” результаты

- Локализует зависящее от состояния поведение
- Делает явными переходы между состояниями
- Объекты состояния можно разделять

Когда применять:

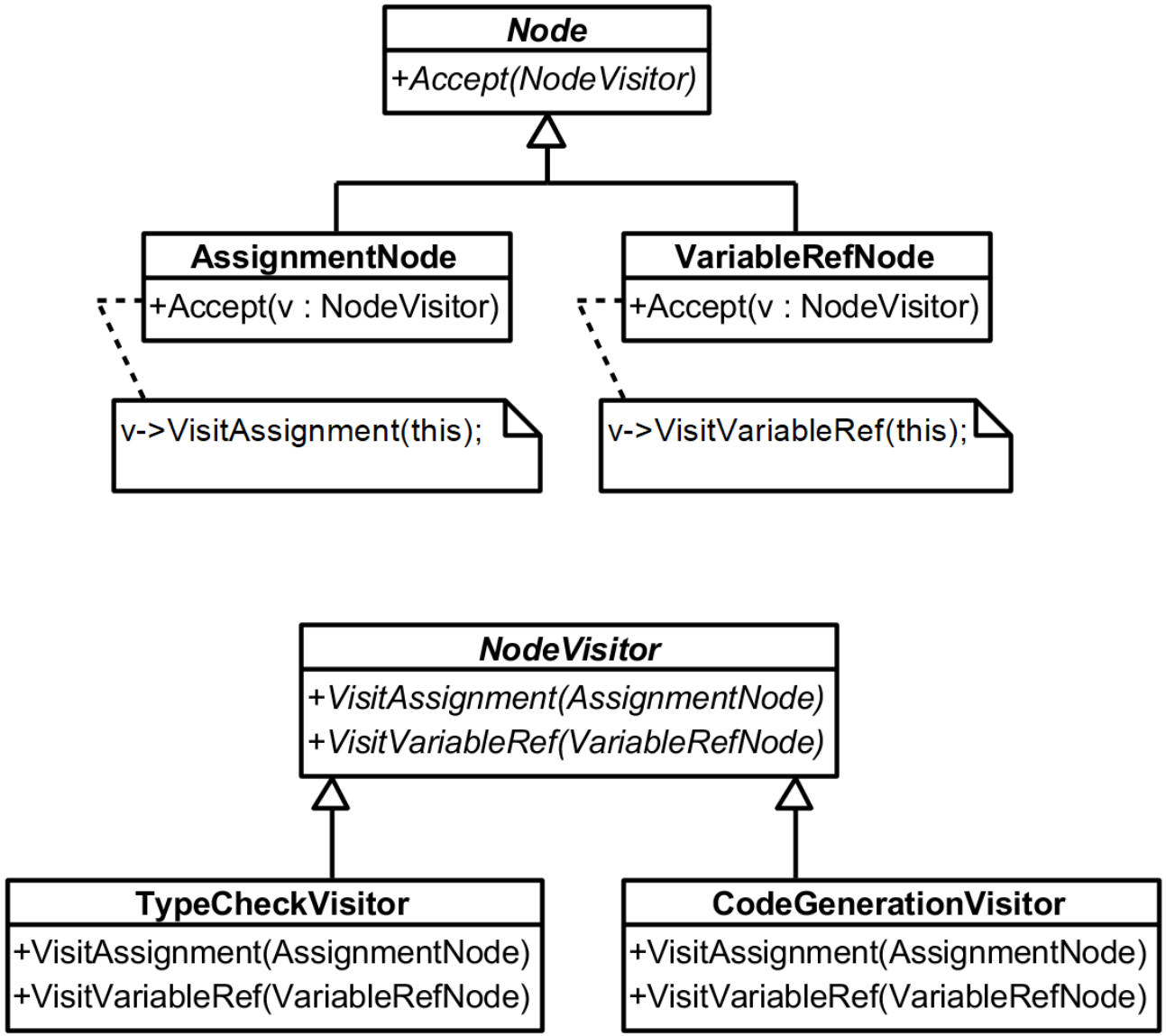
- Поведение объекта зависит от его состояния и должно изменяться во время выполнения
- Обилие условных операторов, в которых выбор ветви зависит от состояния

Паттерн “Посетитель”, мотивация

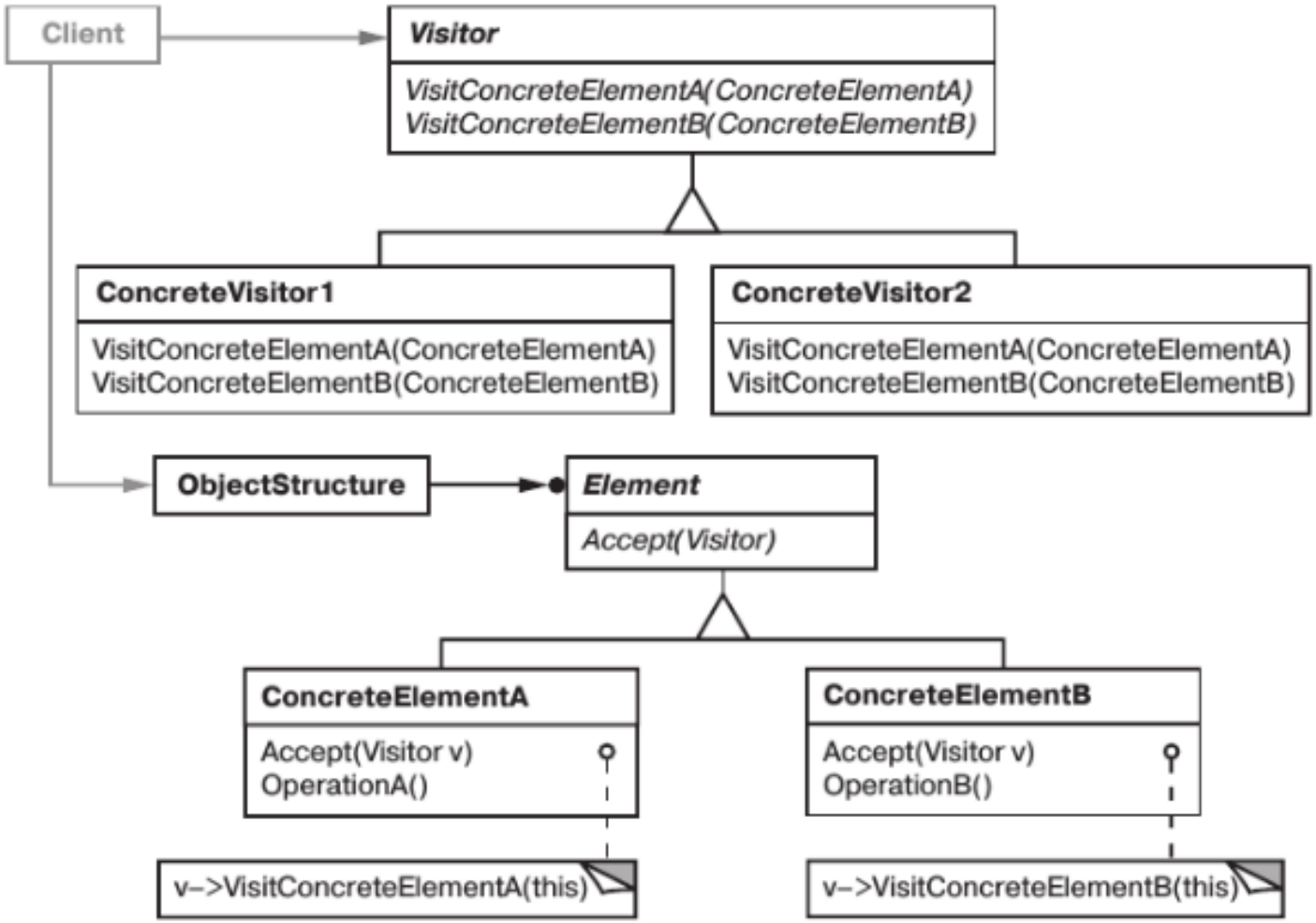


- Синтаксическое дерево
- Много разных типов узлов
- Много разных операций, которые над ними можно выполнять

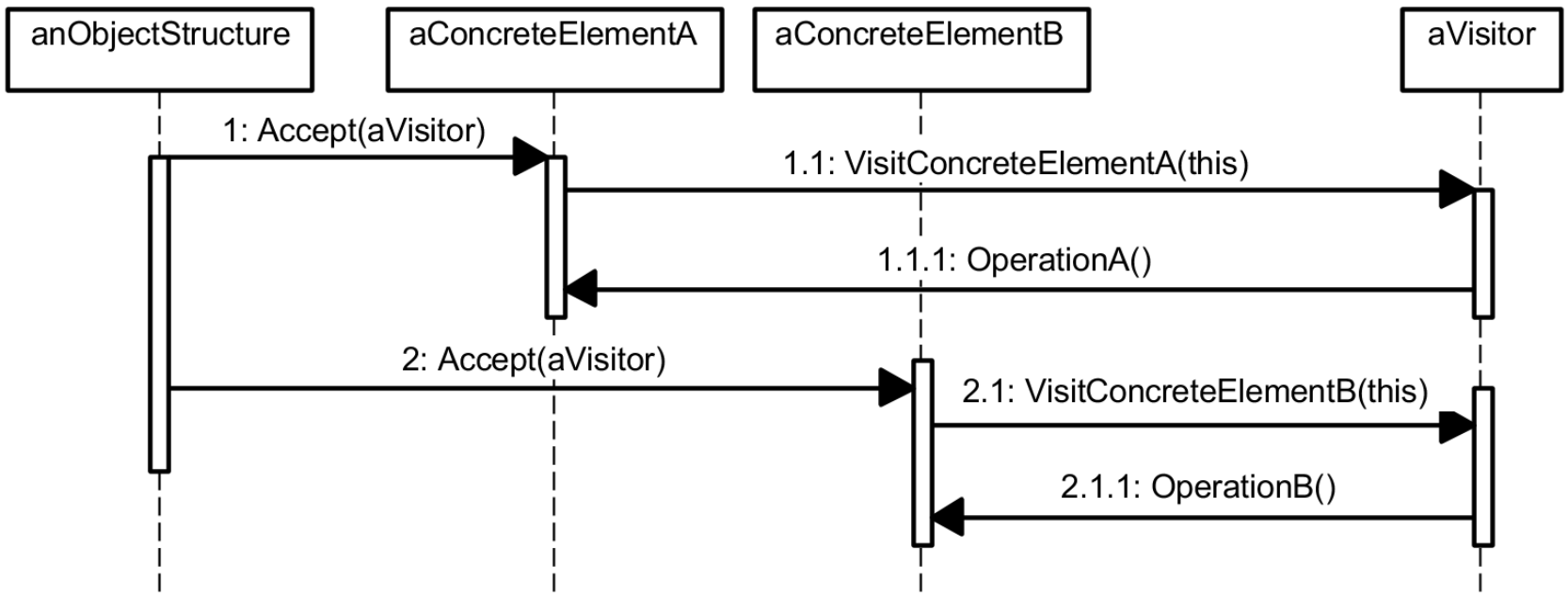
Паттерн “Посетитель”, решение



Паттерн “Посетитель”



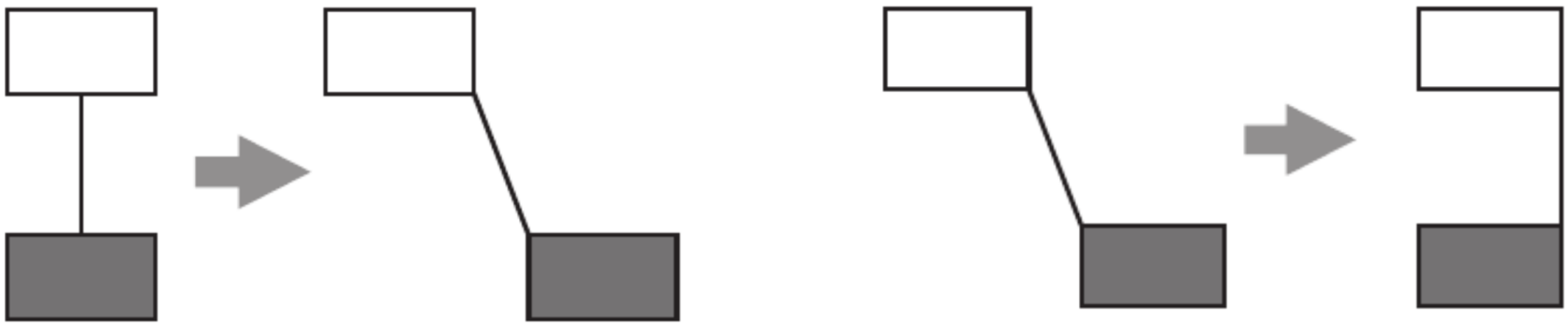
Двойная диспетчеризация



“Посетитель” (Visitor), детали реализации

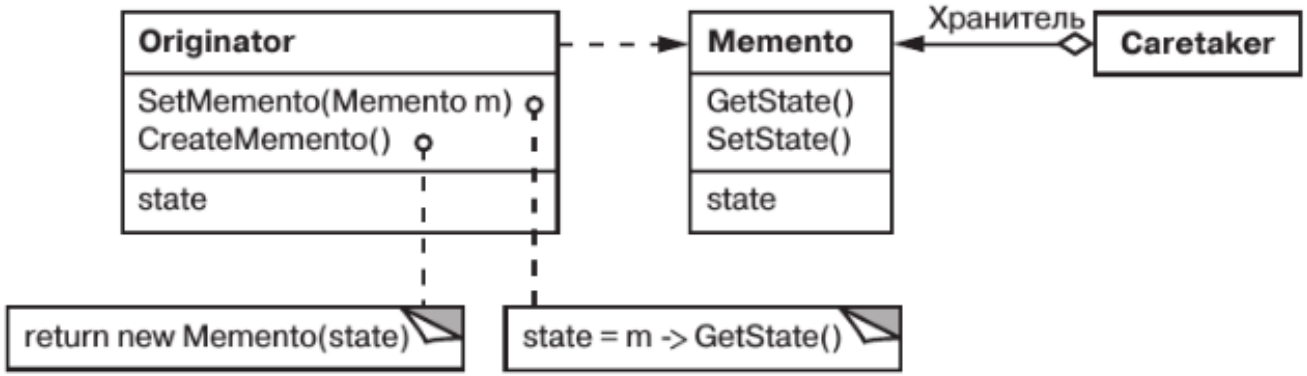
- Использовать перегрузку методов Visit(...)
- Чаще всего сама коллекция отвечает за обход, но может каждый элемент коллекции отдельно
- Может даже сам Visitor, если обход зависит от результата операции
- Аккумулирование состояния
- Несколько нарушает инкапсуляцию
- Просто добавлять новые операции, но сложно добавлять новые классы

Паттерн “Хранитель”, мотивация



- Хотим уметь фиксировать внутреннее состояние объектов
- И восстанавливать его при необходимости
- Не раскрывая внутреннего устройства объектов кому не надо

Паттерн “Хранитель”

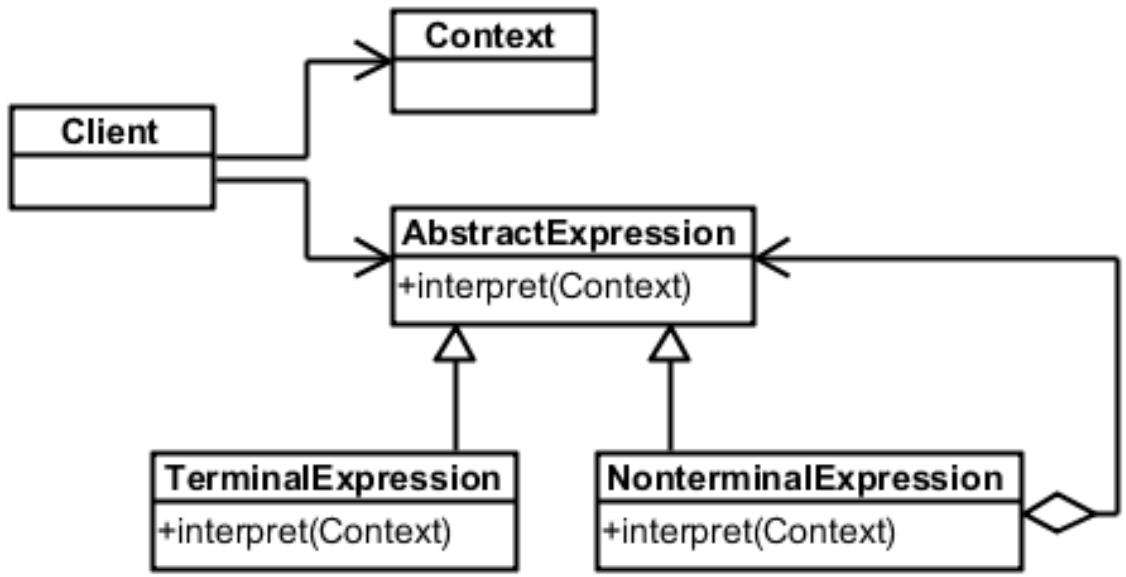


“Хранитель” (Memento), детали реализации

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

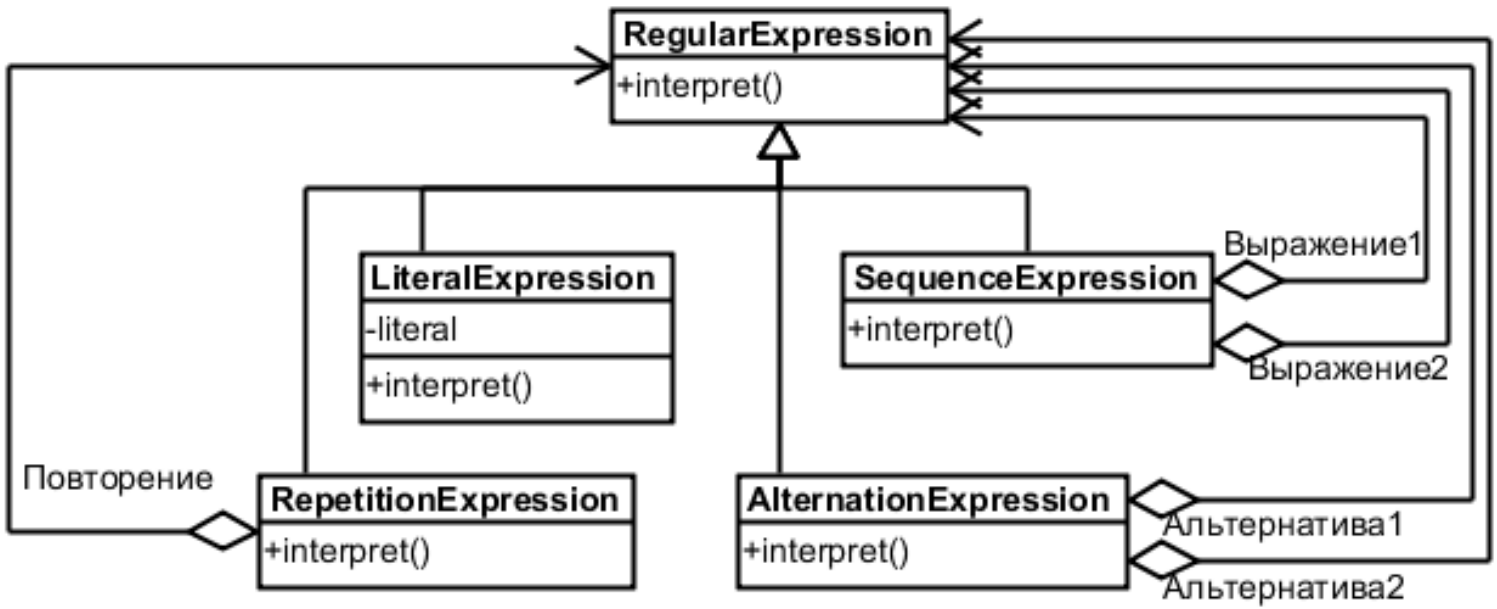
“Интерпретатор” (Interpreter)

Определяет представление грамматики и интерпретатор для заданного языка.



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

“Интерпретатор”, пример



“Интерпретатор”, детали реализации

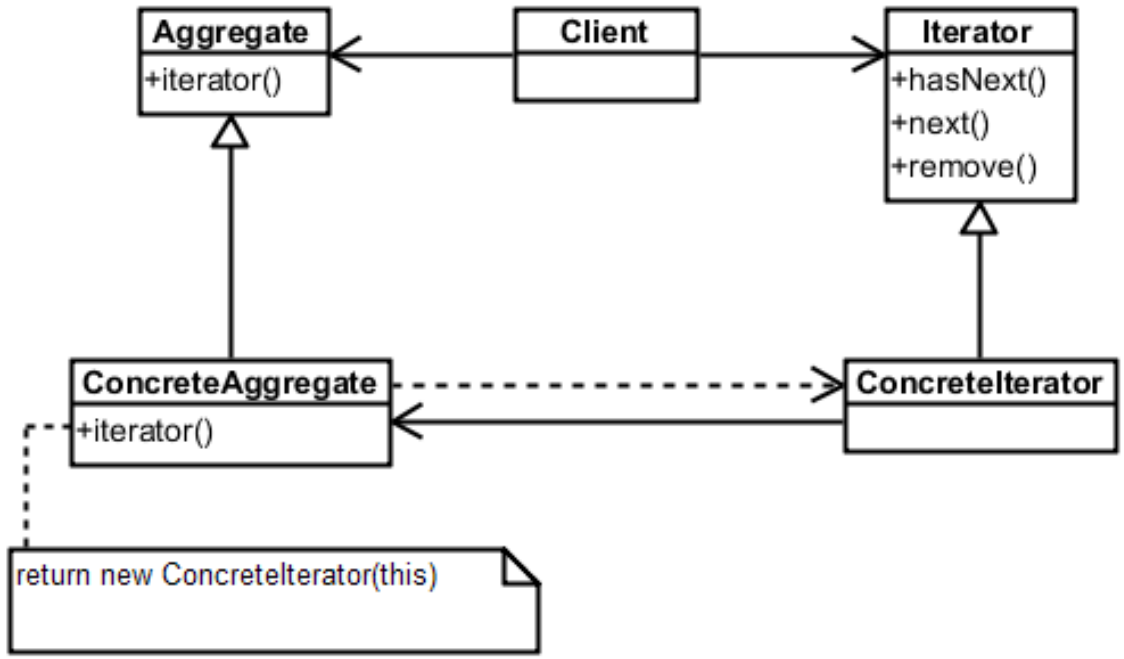
10-е правило Гринспена:

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