Лекция 8: Поведенческие шаблоны

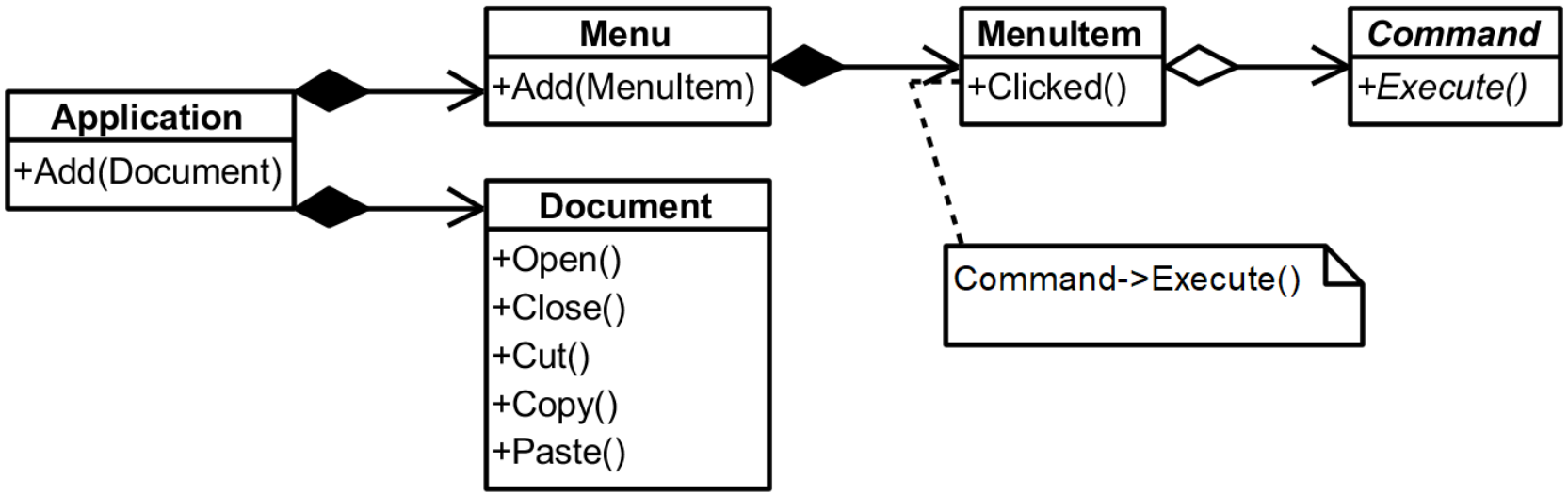
Юрий Литвинов
yurii.litvinov@gmail.com

21.04.2020г

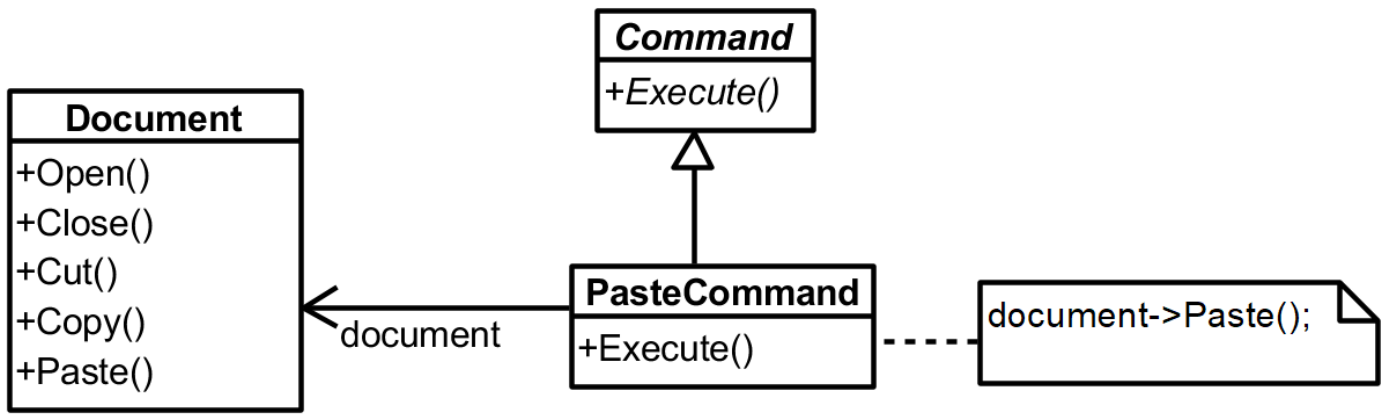
Паттерн “Команда”, мотивация

- 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

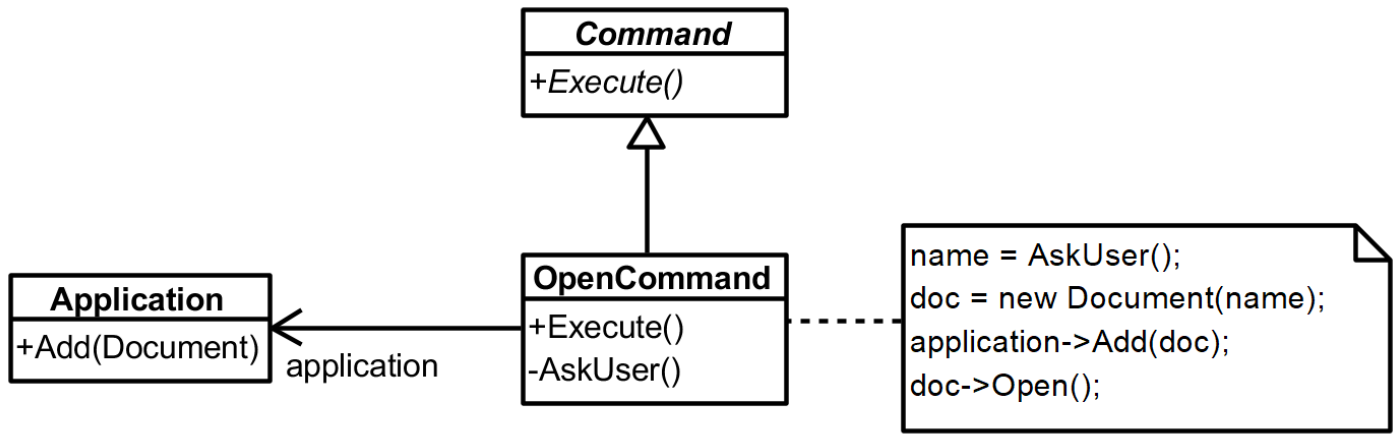
Решение: обернём действие в объект



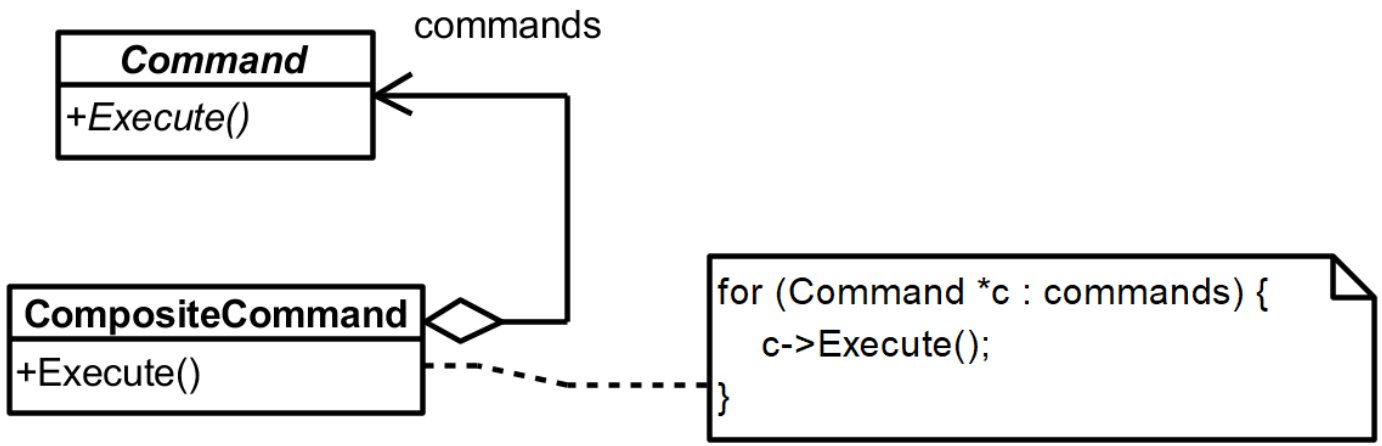
Команда вставки



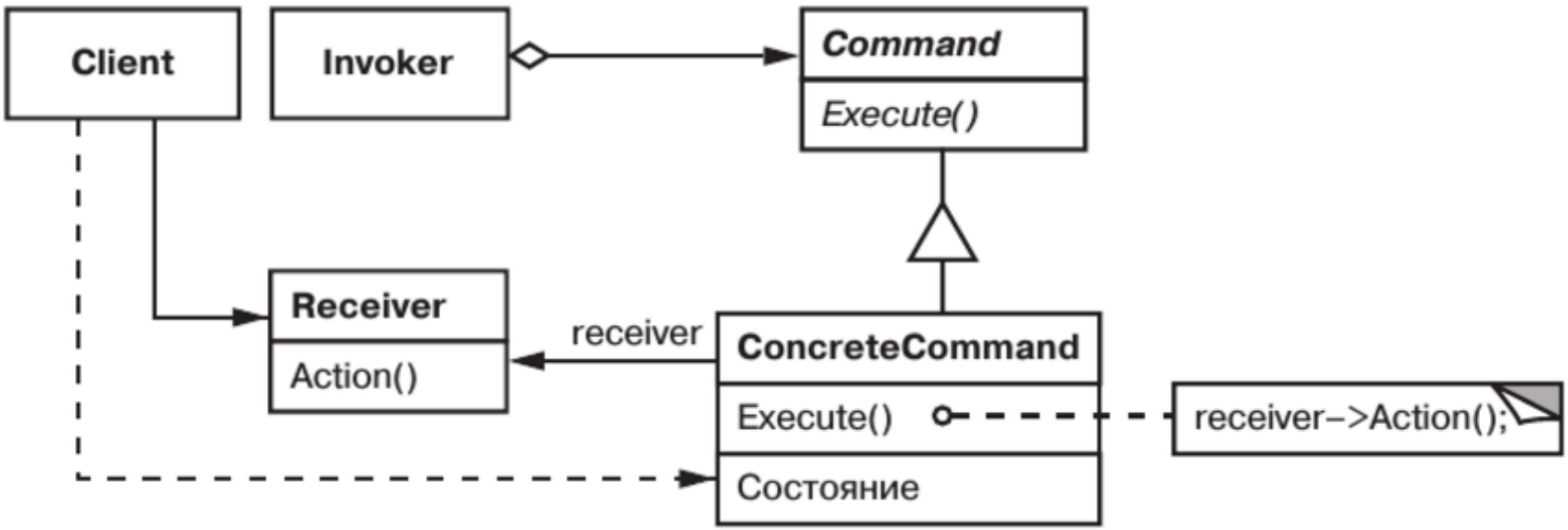
Команда открытия документа



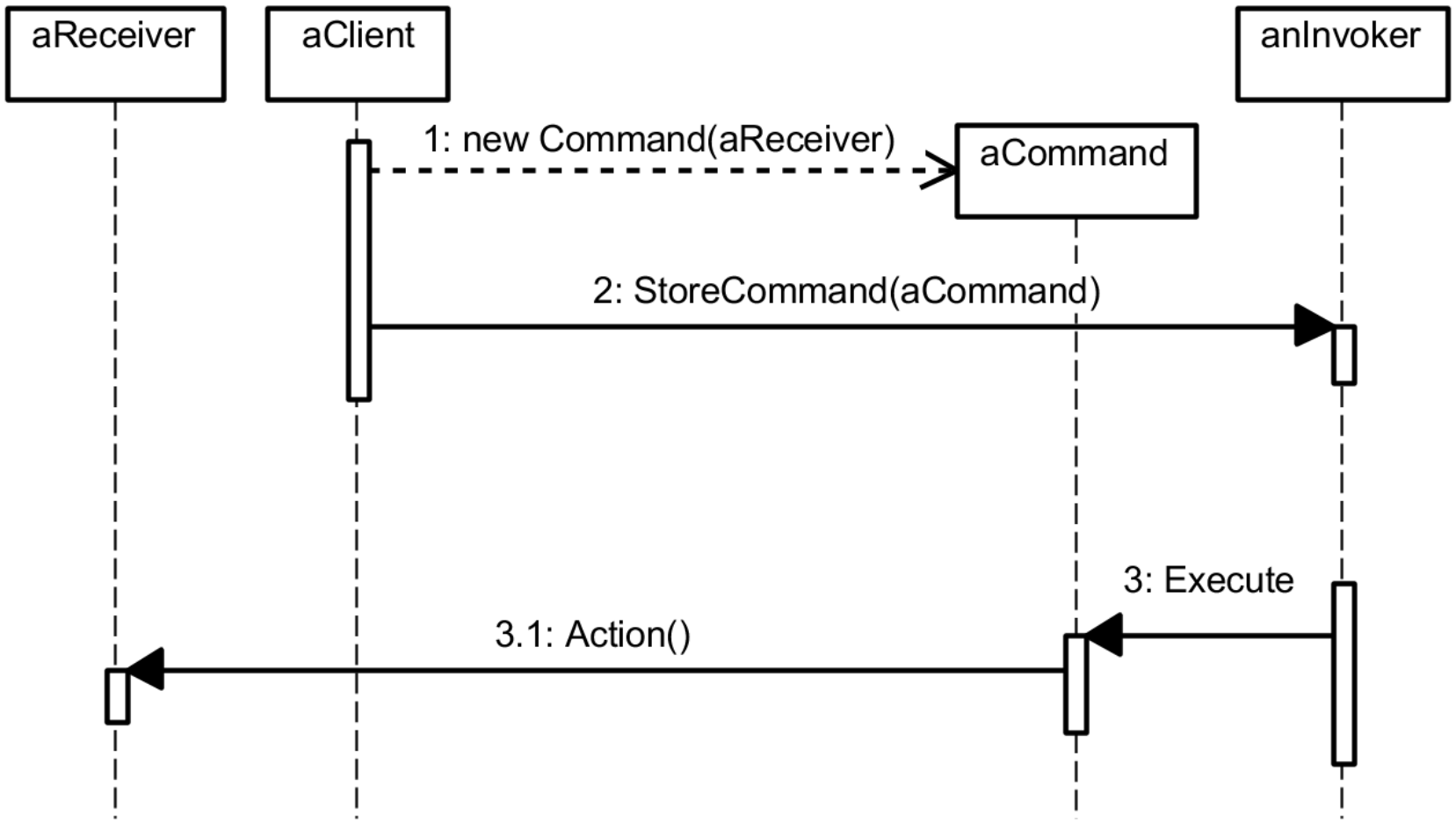
Составная команда



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

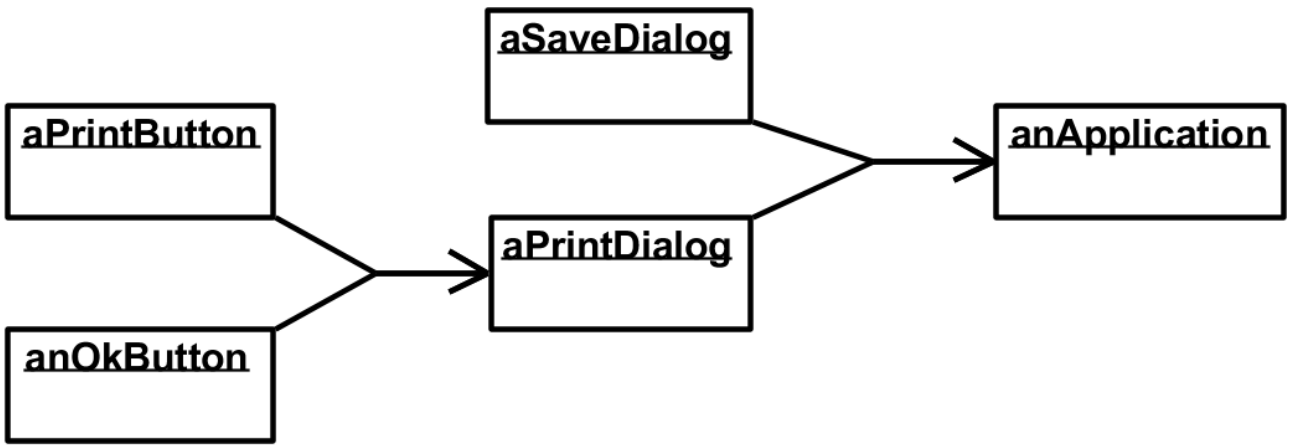
- Насколько “умной” должна быть команда
- Отмена и повторение операций — тоже от хранения всего состояния в команде до “вычислимого” отката
  - Undo-стек и Redo-стек
  - Может потребоваться копировать команды
  - “Искусственные” команды
  - Композитные команды
- Паттерн “Хранитель” для избежания ошибок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

“Команда”, пример

- Qt, класс QAction:

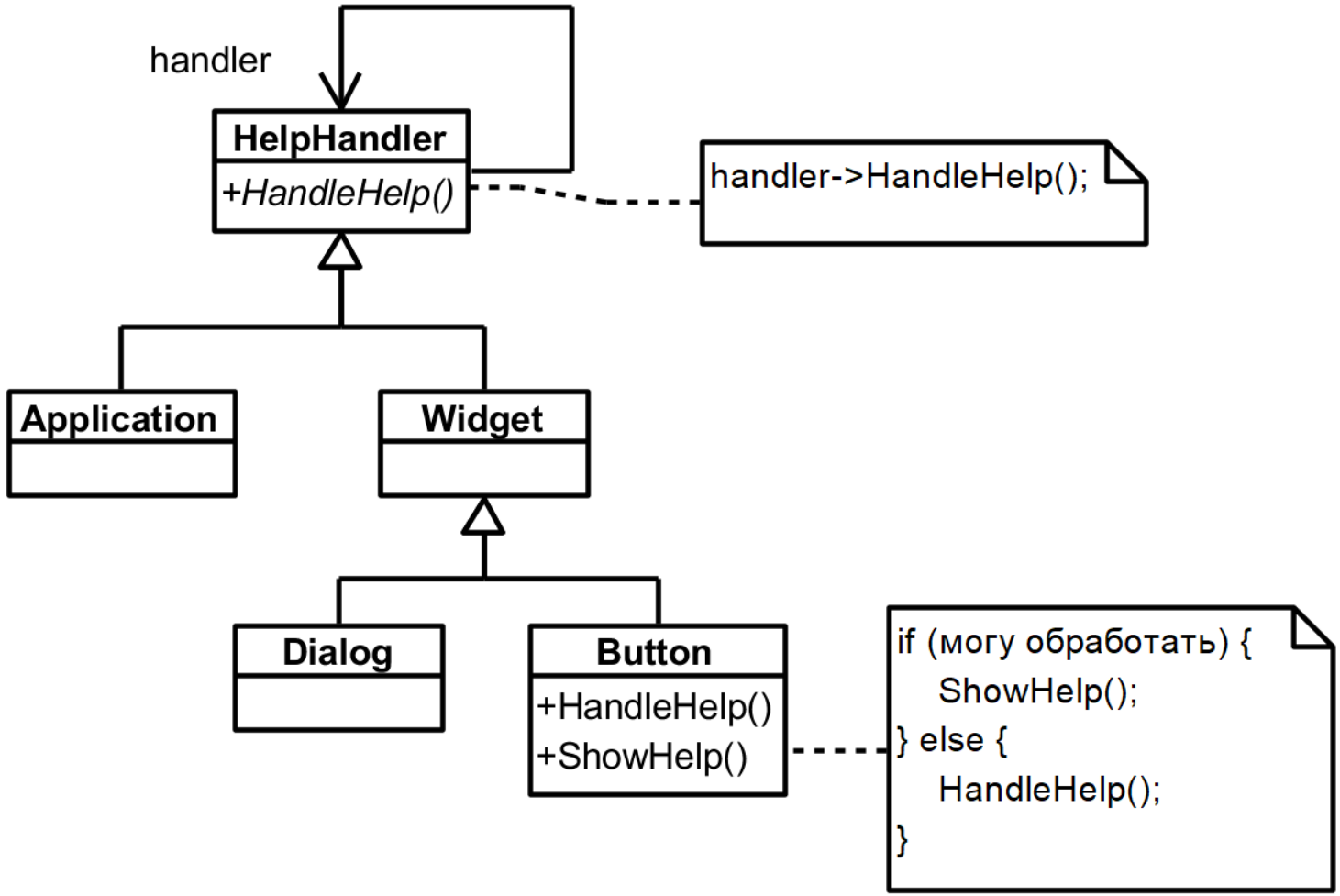
```
const QIcon openIcon = QIcon(":/images/open.png");
QAction *openAct = new QAction(openIcon, tr("&Open..."), this);

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



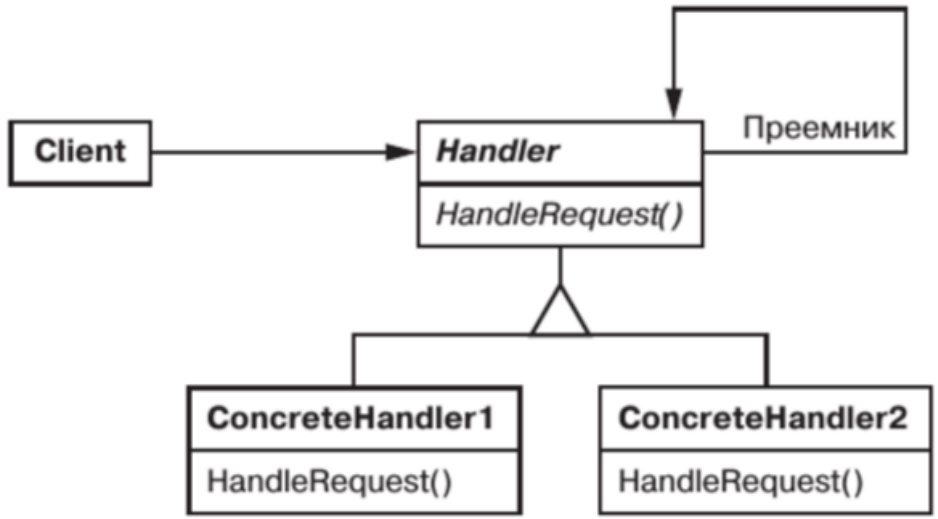
- Организация контекстной справки
- Если у элемента справки нет, запрос передаётся контейнеру
- Заранее неизвестно, кто в итоге обработает запрос

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:

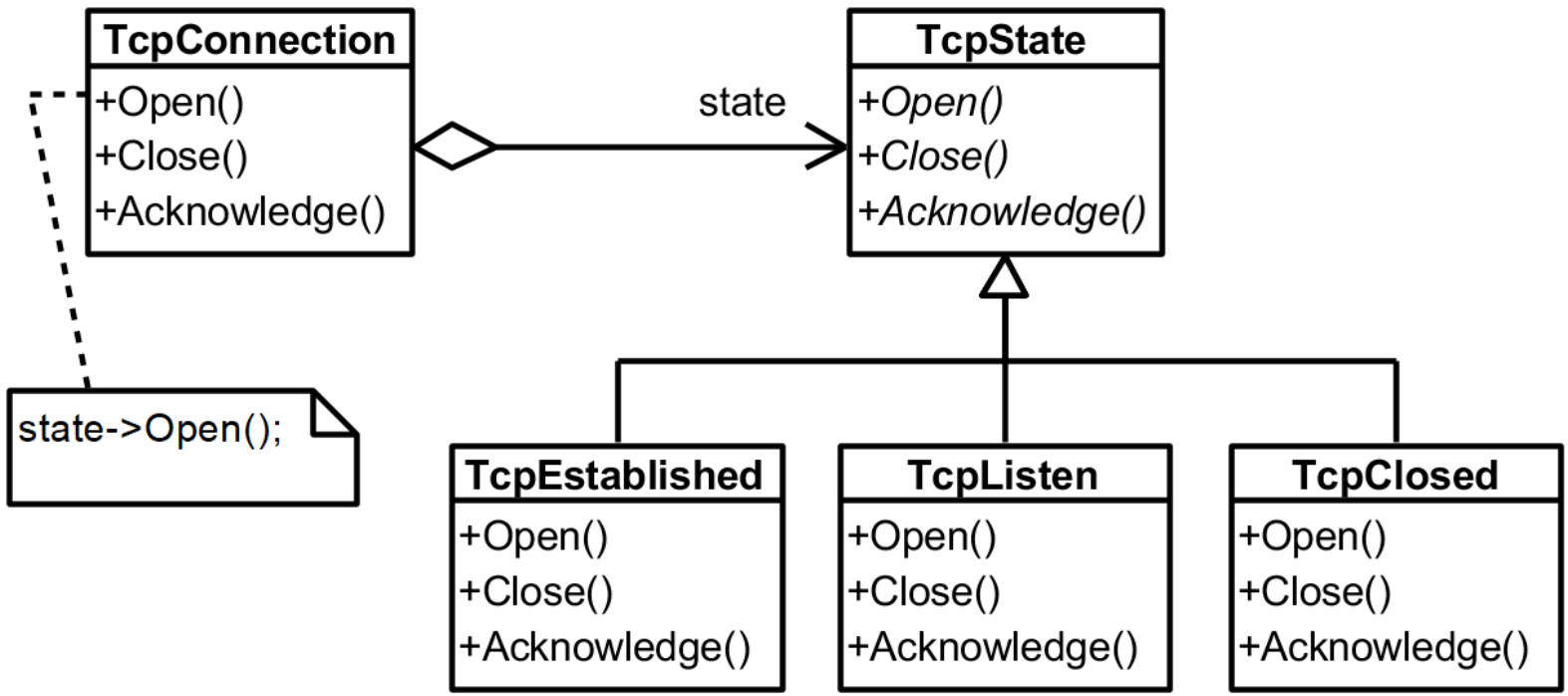
- Есть более одного объекта-обработчика запросов
- Конечный обработчик неизвестен и должен быть найден автоматически
- Хотим отправить запрос нескольким объектам
- Обработчики могут задаваться динамически

“Цепочка ответственности”, примеры

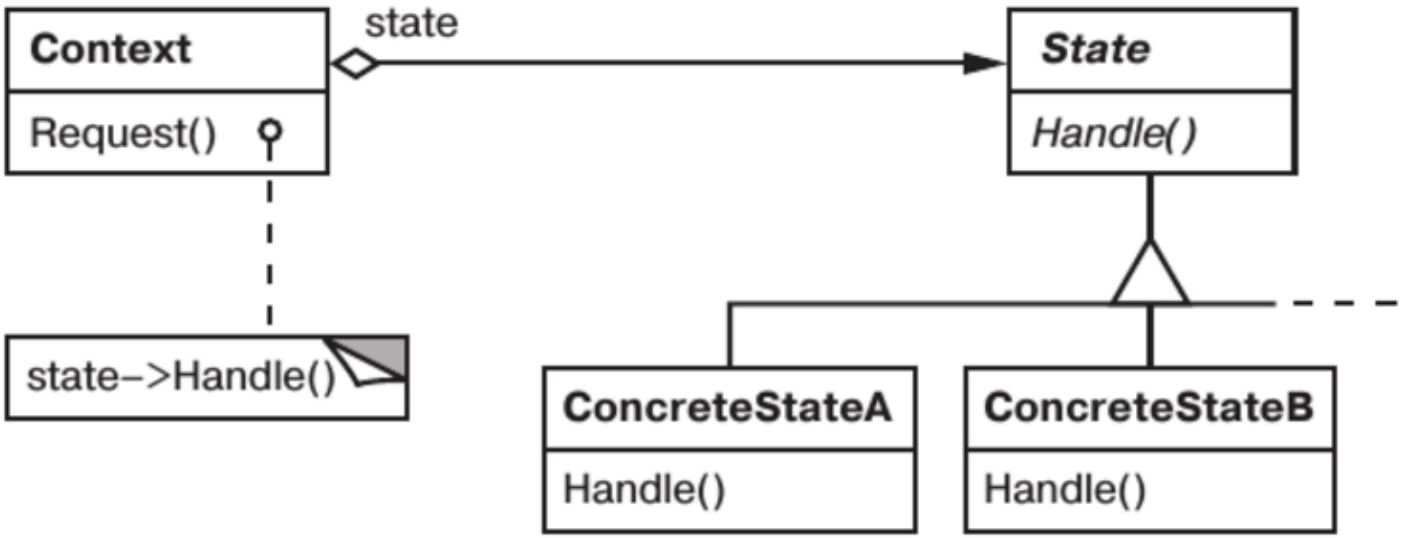
- Распространение исключений
- Распространение событий в оконных библиотеках:

```
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