



## НОВОГОДНЯЯ РАСПРОДАЖА!



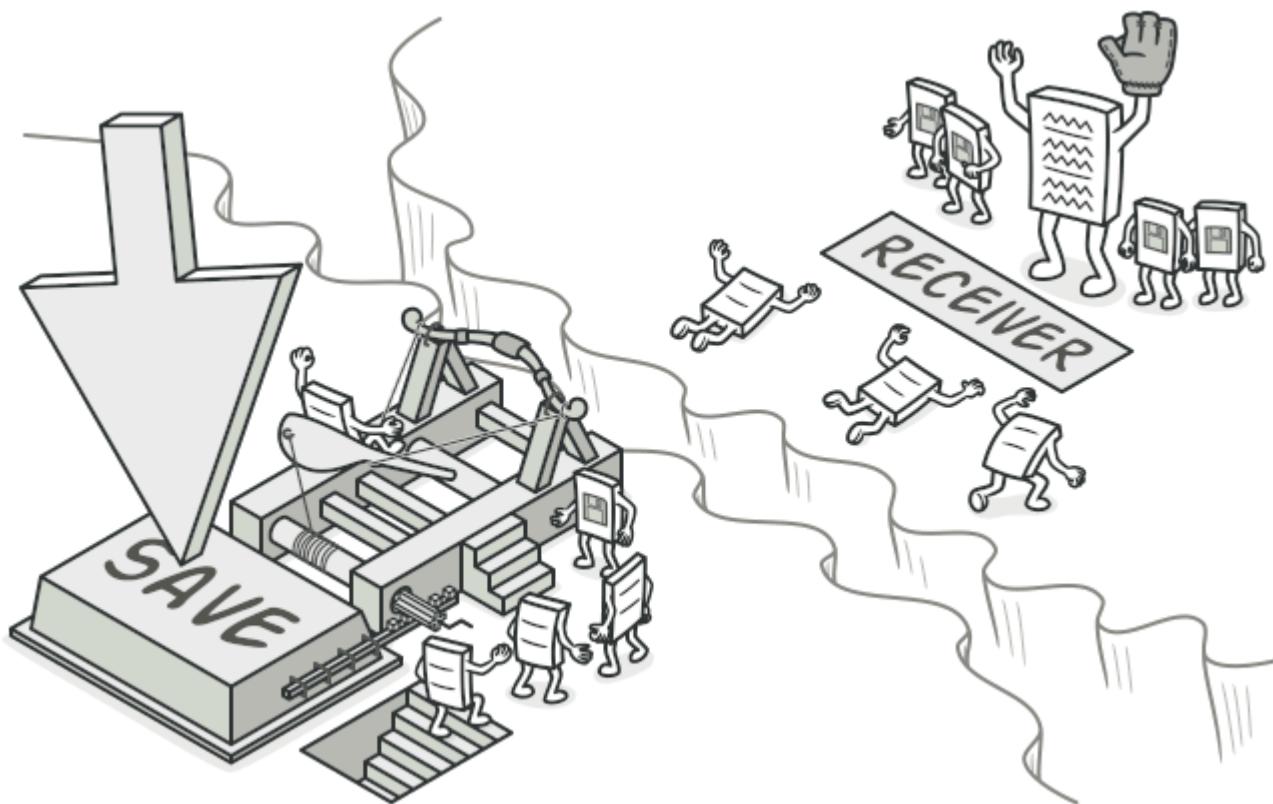
[Home](#) / Паттерны проектирования / Поведенческие паттерны

# Команда

Также известен как: Действие, Транзакция, Action, Command

## 💬 Суть паттерна

**Команда** – это поведенческий паттерн проектирования, который превращает запросы в объекты, позволяя передавать их как аргументы при вызове методов, ставить запросы в очередь, логировать их, а также поддерживать отмену операций.

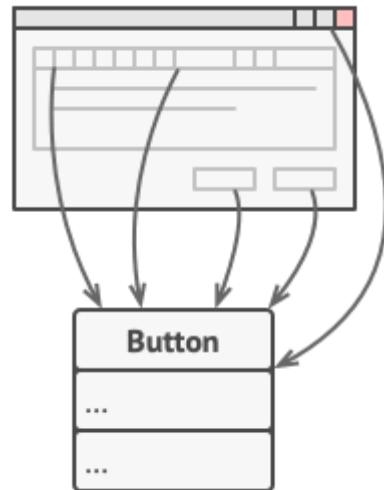


## 😢 Проблема

Представьте, что вы работаете над программой текстового редактора. Дело как раз подошло к разработке панели управления. Вы создали класс красивых [Кнопок](#) и хотите использовать

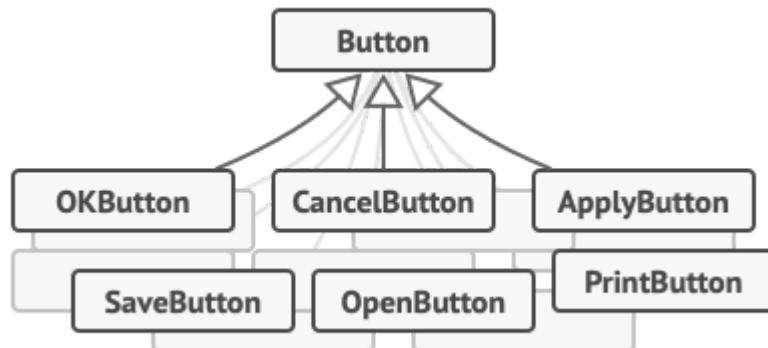


# НОВОГОДНЯЯ РАСПРОДАЖА!



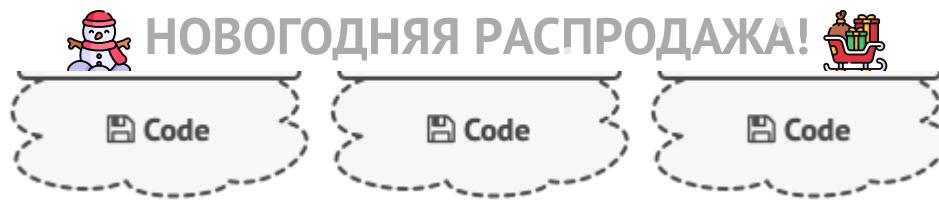
*Все кнопки приложения унаследованы от одного класса.*

Все эти кнопки, хоть и выглядят схоже, но делают разные вещи. Поэтому возникает вопрос: куда поместить код обработчиков кликов по этим кнопкам? Самым простым решением было бы создать подклассы для каждой кнопки и переопределить в них метод действия под разные задачи.



*Множество подклассов кнопок.*

Но скоро стало понятно, что такой подход никуда не годится. Во-первых, получается очень много подклассов. Во-вторых, код кнопок, относящийся к графическому интерфейсу, начинает зависеть от классов бизнес-логики, которая довольно часто меняется.



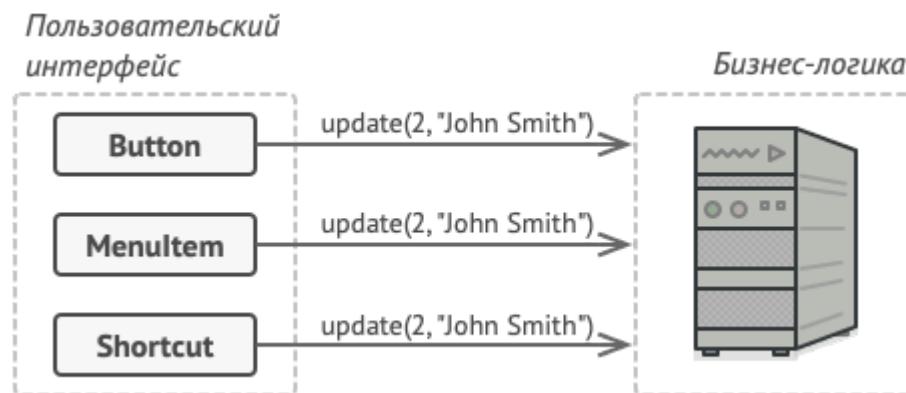
Несколько классов дублируют одну и ту же функциональность.

Но самое обидное ещё впереди. Ведь некоторые операции, например, «сохранить», можно вызывать из нескольких мест: нажав кнопку на панели управления, вызвав контекстное меню или просто нажав клавиши `Ctrl+S`. Когда в программе были только кнопки, код сохранения имелся только в подклассе `SaveButton`. Но теперь его придётся продублировать ещё в два класса.

## 😊 Решение

Хорошие программы обычно структурированы в виде слоёв. Самый распространённый пример – слои пользовательского интерфейса и бизнес-логики. Первый всего лишь рисует красивую картинку для пользователя. Но когда нужно сделать что-то важное, интерфейс «просит» слой бизнес-логики заняться этим.

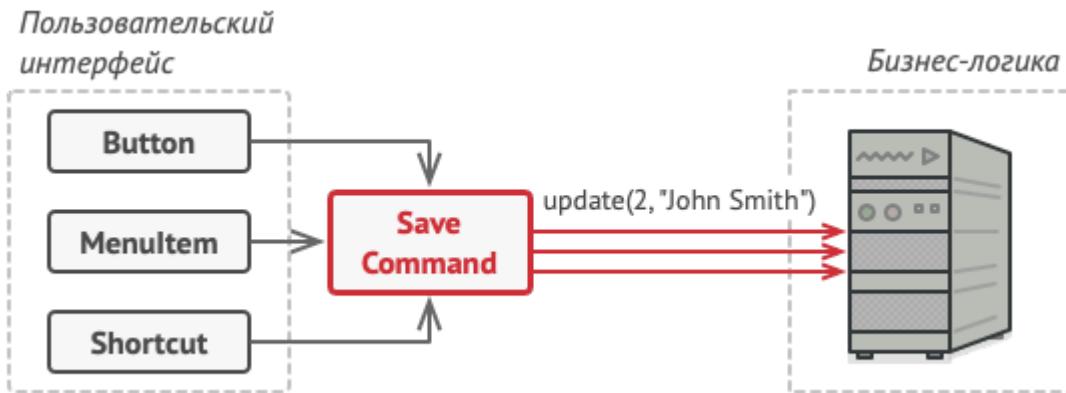
В реальности это выглядит так: один из объектов интерфейса напрямую вызывает метод одного из объектов бизнес-логики, передавая в него какие-то параметры.



Прямой доступ из UI в бизнес-логику.

Паттерн Команда предлагает больше не отправлять такие вызовы напрямую. Вместо этого каждый вызов, отличающийся от других, следует завернуть в собственный класс с единственным методом, который и будет осуществлять вызов. Такие объекты называют *командами*.

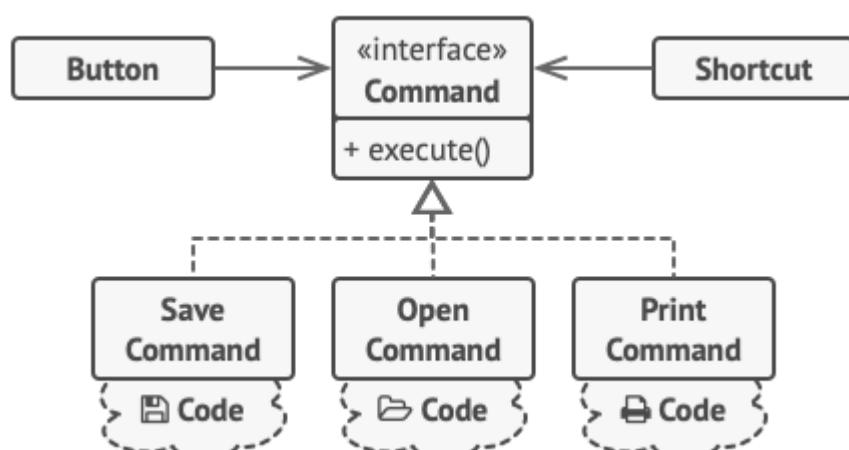
 **НОВОГОДНЯЯ РАСПРОДАЖА!**   
 Каким видят следующие отправители запросы. Когда объект интерфейса будет готов передать запрос, он вызовет метод команды, а та — позаботится обо всём остальном.



Доступ из UI в бизнес-логику через команду.

Классы команд можно объединить под общим интерфейсом с единственным методом запуска. После этого одни и те же отправители смогут работать с различными командами, не привязываясь к их классам. Даже больше: команды можно будет взаимозаменять на лету, изменения итоговое поведение отправителей.

Параметры, с которыми должен быть вызван метод объекта получателя, можно загодя сохранить в полях объекта-команды. Благодаря этому, объекты, отправляющие запросы, могут не беспокоиться о том, чтобы собрать необходимые для получателя данные. Более того, они теперь вообще не знают, кто будет получателем запроса. Вся эта информация скрыта внутри команды.



Классы UI делегируют работу командам.



# НОВОГОДНЯЯ РАСПРОДАЖА!



потребуется создавать уйму подклассов кнопок под разные действия. Будет достаточно единственного класса с полем для хранения объекта команды.

Используя общий интерфейс команд, объекты кнопок будут ссылаться на объекты команд различных типов. При нажатии кнопки будут делегировать работу связанным командам, а команды — перенаправлять вызовы тем или иным объектам бизнес-логики.

Так же можно поступить и с контекстным меню, и с горячими клавишами. Они будут привязаны к тем же объектам команд, что и кнопки, избавляя классы от дублирования.

Таким образом, команды станут гибкой прослойкой между пользовательским интерфейсом и бизнес-логикой. И это лишь малая доля пользы, которую может принести паттерн Команда!

## Аналогия из жизни



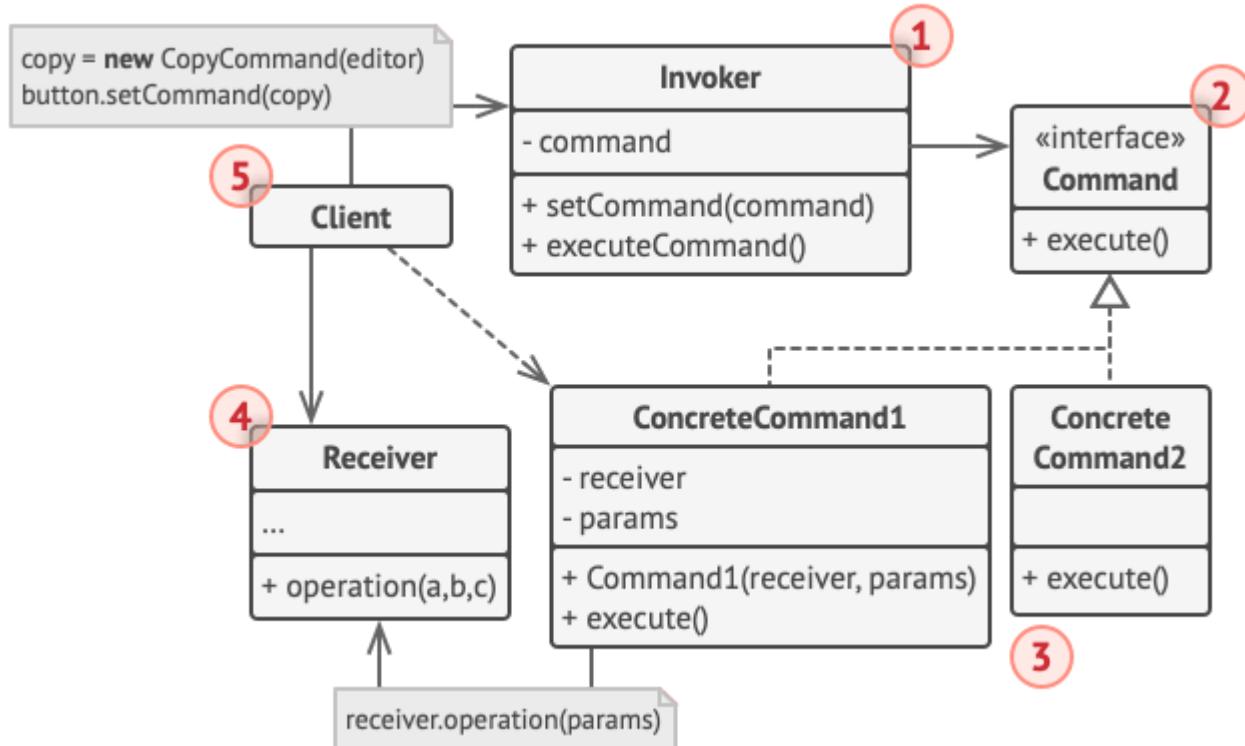
Пример заказа в ресторане.

Вы заходите в ресторан и садитесь у окна. К вам подходит вежливый официант и принимает заказ, записывая все пожелания в блокнот. Откланявшись, он уходит на кухню, где вырывает лист из блокнота и клеит на стену. Далее лист оказывается в руках повара, который читает содержание заказа и готовит заказанные блюда.

В этом примере вы являетесь  *отправителем*, официант с блокнотом —  *командой*, а повар —  *получателем*. Как и в паттерне, вы не соприкасаетесь напрямую с поваром. Вместо этого вы отправляете заказ с официантом, который самостоятельно «настраивает» повара на работу.



## STRUCTURE



1. **Отправитель** хранит ссылку на объект команды и обращается к нему, когда нужно выполнить какое-то действие. Отправитель работает с командами только через их общий интерфейс. Он не знает, какую конкретно команду использует, так как получает готовый объект команды от клиента.

2. **Команда** описывает общий для всех конкретных команд интерфейс. Обычно здесь описан всего один метод для запуска команды.

3. **Конкретные команды** реализуют различные запросы, следуя общему интерфейсу команд. Обычно команда не делает всю работу самостоятельно, а лишь передаёт вызов получателю, которым является один из объектов бизнес-логики.

Параметры, с которыми команда обращается к получателю, следует хранить в виде полей. В большинстве случаев объекты команд можно сделать неизменяемыми, передавая в них все необходимые параметры только через конструктор.

4. **Получатель** содержит бизнес-логику программы. В этой роли может выступать практически любой объект. Обычно команды перенаправляют вызовы получателям. Но



# НОВОГОДНЯЯ РАСПРОДАЖА!

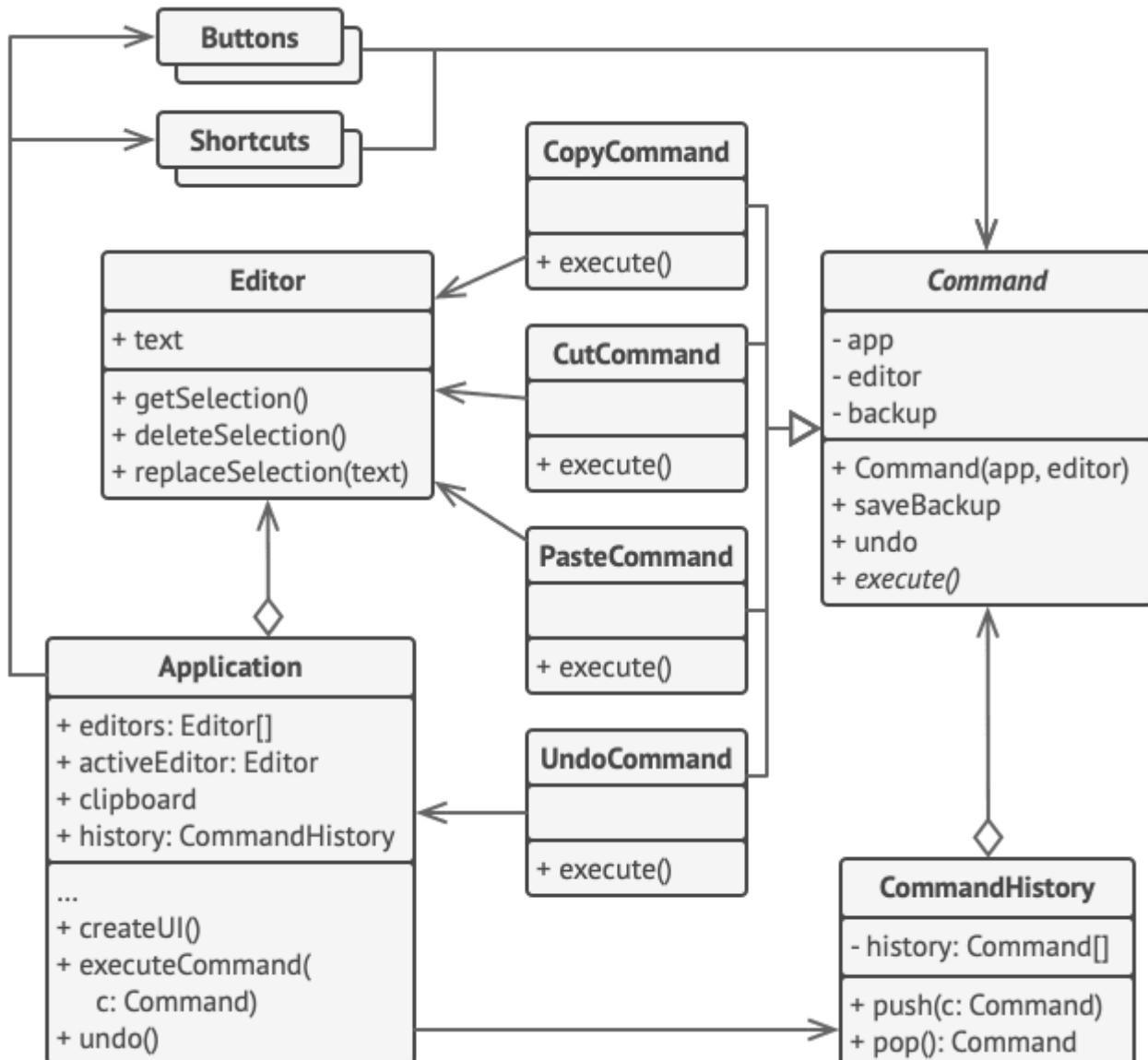


в классы команд.

5. **Клиент** создаёт объекты конкретных команд, передавая в них все необходимые параметры, среди которых могут быть и ссылки на объекты получателей. После этого клиент связывает объекты отправителей с созданными командами.

## # Псевдокод

В этом примере паттерн **Команда** служит для ведения истории выполненных операций, позволяя отменять их, если потребуется.



Пример реализации отмены в текстовом редакторе.



# НОВОГОДНЯЯ РАСПРОДАЖА!



Суффера оюменај, Сохраняюти копию состояния редактора перед выполнением действия.

Копии выполненных команд помещаются в историю команд, откуда они могут быть получены, если нужно будет сделать отмену операции.

Классы элементов интерфейса, истории команд и прочие не зависят от конкретных классов команд, так как работают с ними через общий интерфейс. Это позволяет добавлять в приложение новые команды, не изменяя существующий код.

```
// Абстрактная команда задаёт общий интерфейс для конкретных
// классов команд и содержит базовое поведение отмены операции.

abstract class Command is

    protected field app: Application
    protected field editor: Editor
    protected field backup: text

    constructor Command(app: Application, editor: Editor) is
        this.app = app
        this.editor = editor

    // Сохраняем состояние редактора.
    method saveBackup() is
        backup = editor.text

    // Восстанавливаем состояние редактора.
    method undo() is
        editor.text = backup

    // Главный метод команды остаётся абстрактным, чтобы каждая
    // конкретная команда определила его по-своему. Метод должен
    // возвратить true или false в зависимости о того, изменила
    // ли команда состояние редактора, а значит, нужно ли её
    // сохранить в истории.
    abstract method execute()

// Конкретные команды.

class CopyCommand extends Command is
    // Команда копирования не записывается в историю, так как
    // она не меняет состояние редактора.
    method execute() is
        app.clipboard = editor.getSelection()
        return false

class CutCommand extends Command is
    // Команды, меняющие состояние редактора, сохраняют
    // состояние редактора перед своим действием и сигнализируют
    // об изменении, возвращая true.
    method execute() is
```


**НОВОГОДНЯЯ РАСПРОДАЖА!**


```

editor.deleteSelection()
return true

class PasteCommand extends Command is
method execute() is
    saveBackup()
    editor.replaceSelection(app.clipboard)
    return true

// Отмена – это тоже команда.
class UndoCommand extends Command is
method execute() is
    app.undo()
    return false

// Глобальная история команд – это стек.
class CommandHistory is
private field history: array of Command

// Последний зашёдший...
method push(c: Command) is
    // Добавить команду в конец массива-истории.

// ...выходит первым.
method pop():Command is
    // Достать последнюю команду из массива-истории.

// Класс редактора содержит непосредственные операции над
// текстом. Он отыгрывает роль получателя – команды делегируют
// ему свои действия.
class Editor is
field text: string

method getSelection() is
    // Вернуть выбранный текст.

method deleteSelection() is
    // Удалить выбранный текст.

method replaceSelection(text) is
    // Вставить текст из буфера обмена в текущей позиции.

// Класс приложения настраивает объекты для совместной работы.
// Он выступает в роли отправителя – создаёт команды, чтобы
// выполнить какие-то действия.
class Application is
field clipboard: string

```



**field** history: CommandHistory

// Код, привязывающий команды к элементам интерфейса, может  
// выглядеть примерно так.

**method** createUI() **is**

// ...

```
copy = function() {executeCommand(  
    new CopyCommand(this, activeEditor))}  
copyButton.setCommand(copy)  
shortcuts.onKeyPress("Ctrl+C", copy)
```

```
cut = function() { executeCommand(  
    new CutCommand(this, activeEditor))}  
cutButton.setCommand(cut)  
shortcuts.onKeyPress("Ctrl+X", cut)
```

```
paste = function() { executeCommand(  
    new PasteCommand(this, activeEditor))}  
pasteButton.setCommand(paste)  
shortcuts.onKeyPress("Ctrl+V", paste)
```

```
undo = function() { executeCommand(  
    new UndoCommand(this, activeEditor))}  
undoButton.setCommand(undo)  
shortcuts.onKeyPress("Ctrl+Z", undo)
```

// Запускаем команду и проверяем, надо ли добавить её в  
// историю.

**method** executeCommand(command) **is**

```
if (command.execute())  
    history.push(command)
```

// Берём последнюю команду из истории и заставляем её все  
// отменить. Мы не знаем конкретный тип команды, но это и не  
// важно, так как каждая команда знает, как отменить своё  
// действие.

**method** undo() **is**

```
command = history.pop()  
if (command != null)  
    command.undo()
```

## 💡 Применимость

💡 Когда вы хотите параметризовать объекты выполняемым действием.



# НОВОГОДНЯЯ РАСПРОДАЖА!

взаимозаменять внуки других объектов.



Скажем, вы разрабатываете библиотеку графического меню и хотите, чтобы пользователи могли использовать меню в разных приложениях, не меняя каждый раз код ваших классов. Применив паттерн, пользователям не придётся изменять классы меню, вместо этого они будут конфигурировать объекты меню различными командами.

---

 **Когда вы хотите ставить операции в очередь, выполнять их по расписанию или передавать по сети.**

 Как и любые другие объекты, команды можно сериализовать, то есть превратить в строку, чтобы потом сохранить в файл или базу данных. Затем в любой удобный момент её можно достать обратно, снова превратить в объект команды и выполнить. Таким же образом команды можно передавать по сети, логировать или выполнять на удалённом сервере.

---

 **Когда вам нужна операция отмены.**

 Главная вещь, которая вам нужна, чтобы иметь возможность отмены операций, – это хранение истории. Среди многих способов, которыми можно это сделать, паттерн Команда является, пожалуй, самым популярным.

История команд выглядит как стек, в который попадают все выполненные объекты команд. Каждая команда перед выполнением операции сохраняет текущее состояние объекта, с которым она будет работать. После выполнения операции копия команды попадает в стек истории, все ещё неся в себе сохранённое состояние объекта. Если потребуется отмена, программа возьмёт последнюю команду из истории и возобновит сохранённое в ней состояние.

Этот способ имеет две особенности. Во-первых, точное состояние объектов не так-то просто сохранить, ведь часть его может быть приватным. Но с этим может помочь справиться паттерн [Снимок](#).

Во-вторых, копии состояния могут занимать довольно много оперативной памяти. Поэтому иногда можно прибегнуть к альтернативной реализации, когда вместо восстановления старого состояния команда выполняет обратное действие. Недостаток этого способа в сложности (а иногда и невозможности) реализации обратного действия.



# НОВОГОДНЯЯ РАСПРОДАЖА!



1. Создайте общий интерфейс команд и определите в нём метод запуска.
2. Один за другим создайте классы конкретных команд. В каждом классе должно быть поле для хранения ссылки на один или несколько объектов-получателей, которым команда будет перенаправлять основную работу.

Кроме этого, команда должна иметь поля для хранения параметров, которые нужны при вызове методов получателя. Значения всех этих полей команда должна получать через конструктор.

И, наконец, реализуйте основной метод команды, вызывая в нём те или иные методы получателя.

3. Добавьте в классы отправителей поля для хранения команд. Обычно объекты-отправители принимают готовые объекты команд извне – через конструктор либо через сеттер поля команды.
4. Измените основной код отправителей так, чтобы они делегировали выполнение действия команде.
5. Порядок инициализации объектов должен выглядеть так:

- Создаём объекты получателей.
- Создаём объекты команд, связав их с получателями.
- Создаём объекты отправителей, связав их с командами.

## ⊕⊖ Преимущества и недостатки

- |  |  |
|--|--|
| <ul style="list-style-type: none"> <li>✓ Убирает прямую зависимость между объектами, вызывающими операции, и объектами, которые их непосредственно выполняют.</li> <li>✓ Позволяет реализовать простую отмену и повтор операций.</li> <li>✓ Позволяет реализовать отложенный запуск операций.</li> <li>✓ Позволяет собирать сложные команды из простых.</li> </ul> | <ul style="list-style-type: none"> <li>✗ Усложняет код программы из-за введения множества дополнительных классов.</li> </ul> |
|--|--|



# НОВОГОДНЯЯ РАСПРОДАЖА!



## ↔ Отношения с другими паттернами

- **Цепочка обязанностей**, **Команда**, **Посредник** и **Наблюдатель** показывают различные способы работы отправителей запросов с их получателями:
  - *Цепочка обязанностей* передаёт запрос последовательно через цепочку потенциальных получателей, ожидая, что какой-то из них обработает запрос.
  - *Команда* устанавливает косвенную одностороннюю связь от отправителей к получателям.
  - *Посредник* убирает прямую связь между отправителями и получателями, заставляя их общаться опосредованно, через себя.
  - *Наблюдатель* передаёт запрос одновременно всем заинтересованным получателям, но позволяет им динамически подписываться или отписываться от таких оповещений.

- Обработчики в **Цепочке обязанностей** могут быть выполнены в виде **Команд**. В этом случае множество разных операций может быть выполнено над одним и тем же контекстом, коим является запрос.

Но есть и другой подход, в котором сам запрос является *Командой*, посланной по цепочке объектов. В этом случае одна и та же операция может быть выполнена над множеством разных контекстов, представленных в виде цепочки.

- **Команду** и **Снимок** можно использовать сообща для реализации отмены операций. В этом случае объекты команд будут отвечать за выполнение действия над объектом, а снимки будут хранить резервную копию состояния этого объекта, сделанную перед самым запуском команды.
- **Команда** и **Стратегия** похожи по духу, но отличаются масштабом и применением:
  - *Команду* используют, чтобы превратить любые разнородные действия в объекты. Параметры операции превращаются в поля объекта. Этот объект теперь можно логировать, хранить в истории для отмены, передавать во внешние сервисы и так далее.
  - С другой стороны, *Стратегия* описывает разные способы произвести одно и то же действие, позволяя взаимозаменять эти способы в каком-то объекте контекста.
- Если **Команду** нужно копировать перед вставкой в историю выполненных команд, вам может помочь **Прототип**.



## </> Примеры реализации паттерна



### Не втыкай в транспорте

Главная | Рефакторинг | Паттерны | Премиум контент | Книги | Партнеры | Проектирования.

Курс по Git | Форум | Связаться

Теперь это удобно делать даже во время поездок в общественном транспорте.



Узнать больше...

© 2014-2025 Refactoring.Guru.

Условия использования

Все права защищены.

Политика конфиденциальности

Иллюстрации нарисовал Дмитрий Жарт

Использование контента | About us

#### Ukrainian office:

FOP Olga Skobeleva

Abolmasova 7

Kyiv, Ukraine, 02002

Email:

support@refactoring.guru

#### Spanish office:

Oleksandr Shvets

Avda Pamplona 64

Pamplona, Spain, 31009

Email:

support@refactoring.guru