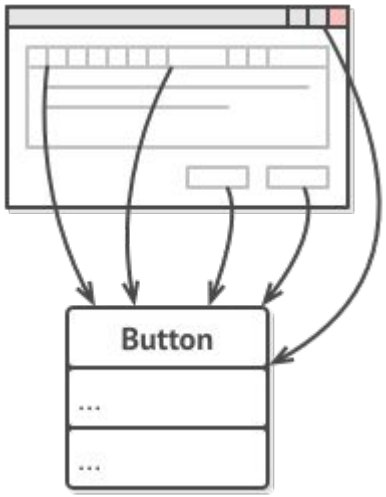
# Команда&Снимок

- **Команда** - поведенческий паттерны 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.
- **Снимок** — поведенческий паттерн проектирования, который позволяет делать снимки состояния объектов, не раскрывая подробностей их реализации. Затем снимки можно использовать, чтобы восстановить прошлое состояние объектов.

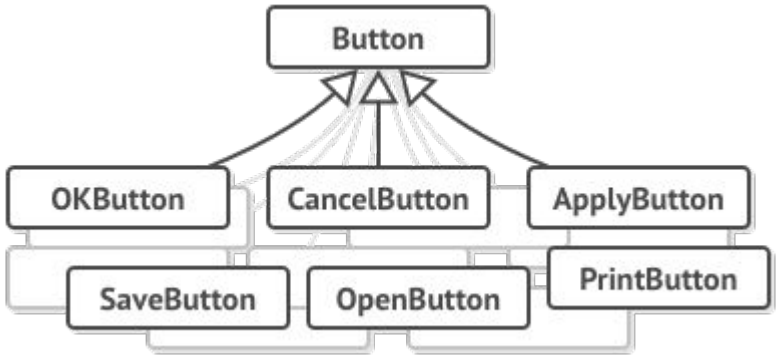
# Содержание

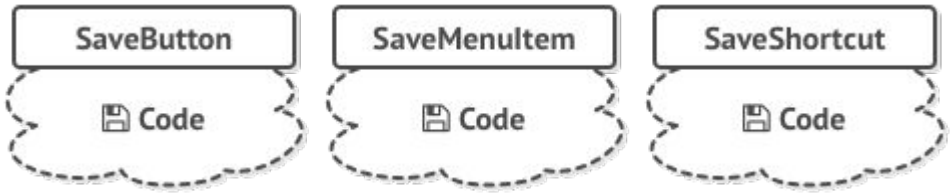
- Команда
- Пример реализации Команды
- Снимок
- Пример реализации снимка

# Проблема



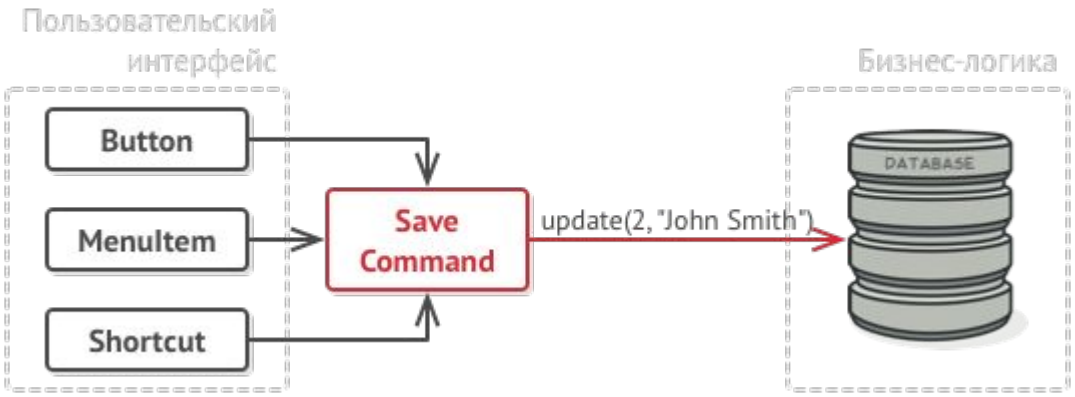

Некоторые операции, например «сохранить», можно вызывать из нескольких мест — нажав кнопку на панели управления, вызвав контекстное меню или просто нажав клавиши Ctrl+S



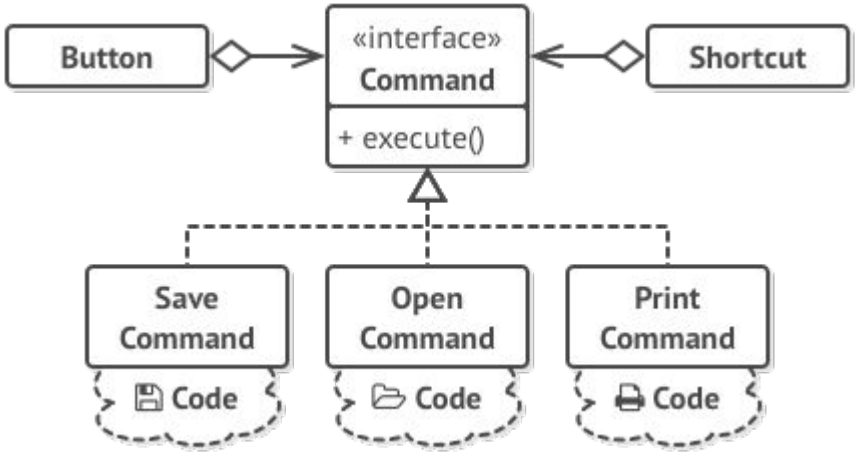
# Решение

- Программы обычно структурированы в виде слоёв. Самый распространённый пример — слои 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.
- Паттерн Команда предлагает не отправлять такие вызовы напрямую, а «завернуть» их в отдельные объекты с единственным методом, который приводит вызов в действие.



# Вариант решения

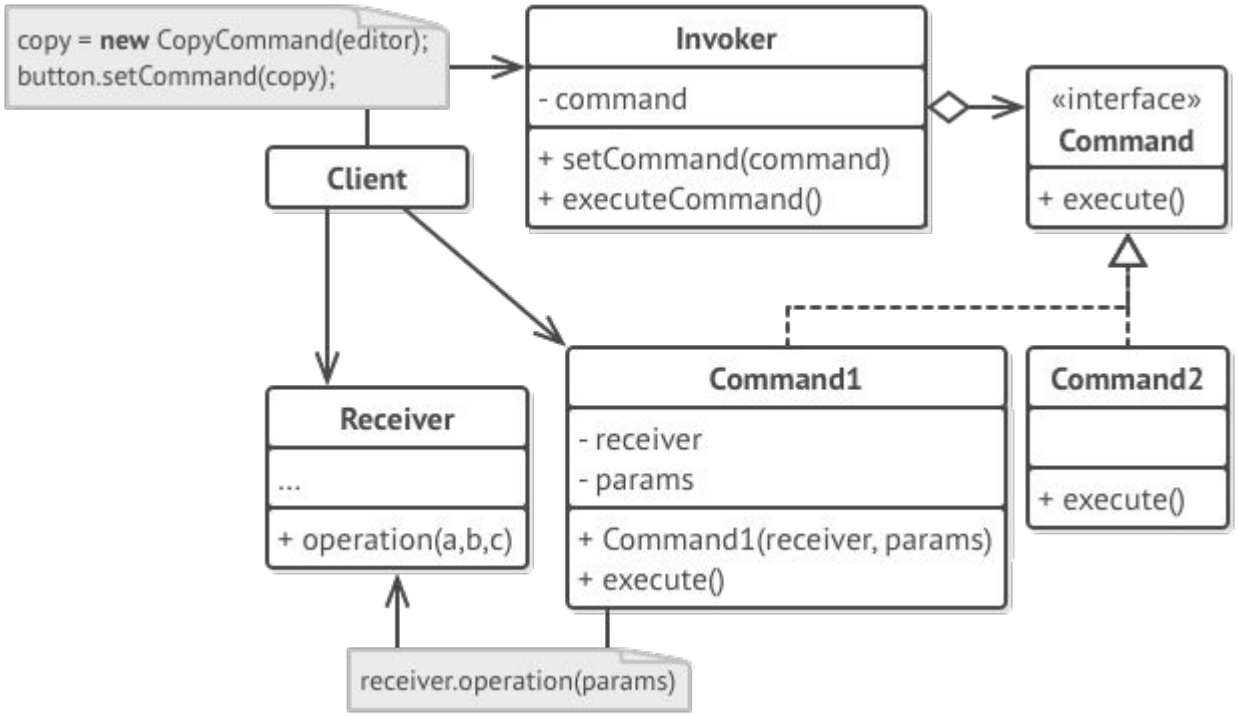
- Параметры, с которыми должен быть вызван метод объекта получателя, можно загодя сохранить в полях объекта-команды. Благодаря этому, объекты, отправляющие запросы, могут не беспокоиться о том, чтобы собрать необходимые для получателя данные. Более того, они теперь вообще не знают, кто будет получателем запроса. Вся эта информация скрыта внутри команды.
- Классы команд можно объединить под общим интерфейсом, с единственным методом запуска команды. 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, изменяя итоговое поведение отправителей.



# Структура

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 — определённому объекту бизнес-логики.
4. **Параметры**, с которыми команда обращается к получателю, следует хранить в виде полей. В большинстве случаев, объекты команд можно сделать неизменяемым, предавая в них все необходимые параметры только через конструктор.
5. **Получатель** содержит бизнес-логику программы. В этой роли может выступать практически любой объект. Обычно, команды перенаправляют вызовы получателям. Но иногда, чтобы упростить программу, вы можете избавиться от получателей, слив их код в классы команд.

# Схема


copy = new CopyCommand(editor);
button.setCommand(copy);
Invoker
- command
+ setCommand(command)
+ executeCommand()
«interface»
Command
+ execute()
Client
Receiver
...
+ operation(a,b,c)
Command1
- receiver
- params
+ Command1(receiver, params)
+ execute()
Command2
+ execute()
receiver.operation(params)

Паттерн **Команда** служит для ведения истории выполненных операций, позволяя, отменять их, если потребуется

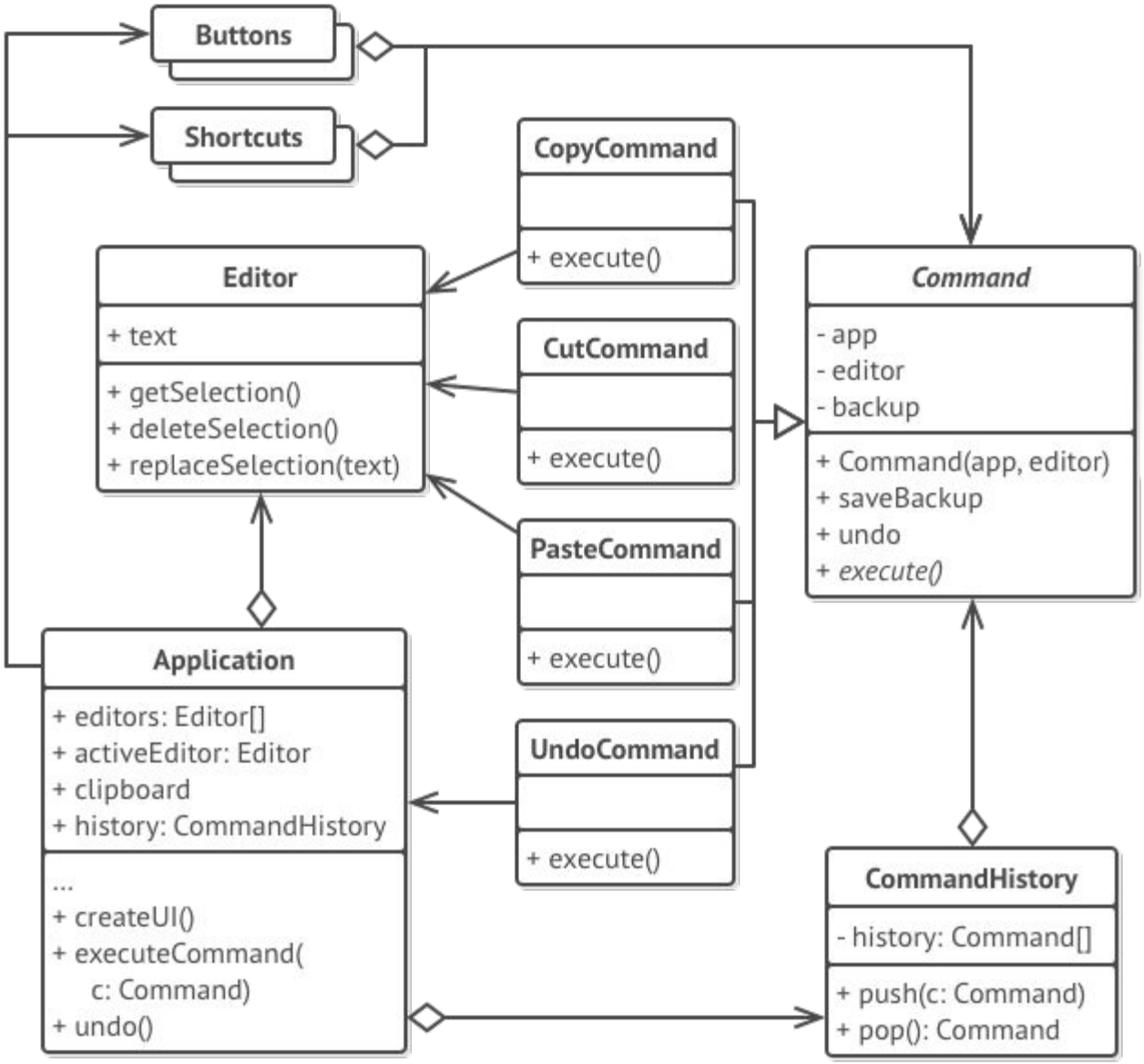
Команды, которые меняют состояние редактора (например, команда вставки текста из буфера обмена), сохраняют копию состояния редактора перед выполнением действия.

Копии выполненных команд помещаются в историю команд, откуда они могут быть доставлены, 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.

# *Пример реализации оттмены в текстовом редакторе*



# Код класса Command

```
// Абстрактная команда задаёт общий интерфейс для всех команд.
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()
```

# Команды

```
// Конкретные команды.
class CopyCommand extends EditorCommand 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 EditorCommand is // Команды, меняющие состояние редактора, сохраняют состояние редактора
// перед своим действием и сигнализируют об изменении, возвращая true.
method execute() is
 saveBackup()
app.clipboard = editor.getSelection()
editor.deleteSelection() return true
class PasteCommand implements Command is
 method execute() is
saveBackup()
editor.replaceSelection(app.clipboard) return true // Отмена это тоже команда.
class UndoCommand implements Command is
method execute() is
 app.undo()
return false // Глобальная история команд — это стек.
class CommandHistory is
private field history: array of Command // Последний зашедший...
method push(c: Command) is
Push command to the end of history array. // ...выходит первым.
method pop():Command is Get the most recent command from history.
```

# Класс Редактор

```
// 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
Return selected text.
method deleteSelection() is
 Delete selected text.
 method replaceSelection(text) is
Insert clipboard contents at current position.
// 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 editors: array of Editors
field activeEditor: Editor
field history: CommandHistory
```

# Класс приложения

```
// Код, привязывающий команды к элементам интефрейса 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.**

Команда превращает операции в объекты. А объекты можно передавать, хранить и взаимозаменять внутри других объектов.

Для библиотеки графического меню и нужно, чтобы пользователи могли использовать меню в разных приложениях, не меняя каждый раз код ваших классы. 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 как это делается, паттерн Команда является, пожалуй, самым популярным.

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

# Шаги реализации

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.
3. Кроме этого, команда должна иметь поля для хранения параметров, которые нужны при вызове методов получателя. Значения всех этих полей команда должна получать через конструктор.
4. И наконец, реализуйте основной метод команды, вызывая в нём те или иные методы получателя.
5. Добавьте в классы отправителей поля для хранения команд. Объект-отправитель должен принимать готовый объект команды извне через конструктор, либо через сеттер команды.
6. Измените основной код отправителей так, чтобы они делегировали выполнение действия команде.
7. Порядок инициализации объектов должен выглядеть так:
    1. Создаём объекты получателей.
    2. Создаём объекты команд, связав их с получателями.
    3. Создаём объекты отправителей, связав их с командами.

# Преимущества и недостатки

- Убирает прямую зависимость между объектами, вызывающими операции и объектами, которые их непосредственно выполняют.(+)
- Позволяет реализовать простую отмену и повтор операций.(+)
- Позволяет реализовать отложенный запуск команд.(+)
- Позволяет собирать сложные команды из простых.(+)
- Соблюдает *принцип открытости/закрытости*.(+)
- Усложняет код программы за счёт дополнительных классов.(-)

# Отношения с другими паттернами

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 отображать выполненные действие над объектом, снимки — хранить копию состояния этого объекта до того, как команда была выполнена.

**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 сделать 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**.

**Посетитель** это более мощный аналог **Команды**, которую можно выполнить сразу над объектами нескольких классов.

# Снимок

**Проблема**

Предположим, что вы пишете текстовый редактор. Помимо обычного редактирования, ваш редактор позволяет менять форматирование текста, вставлять картинки и прочее.

В какой-то момент вы решили сделать все эти действия отменяемыми. Для этого вам нужно сохранять текущее состояние редактора перед тем, как выполнить любое действие..

Чтобы сделать копию состояния объекта, достаточно скопировать значение его полей.

Если вы решите провести рефакторинг — убрать или добавить парочку полей в класс редактора — то придётся изменять код всех классов, которые могли копировать состояние редактора.

Чтобы сделать копию состояния, вам нужно записать значения всех этих полей в некий «контейнер».

Скорее всего, вам понадобится хранить массу таких «контейнеров», поэтому удобней всего сделать их объектами одного класса.

Этот класс должен иметь массу полей и практически никаких методов. Чтобы другие классы смогли записывать и читать из него данные, вам придётся сделать его поля публичными.

Другие классы станут зависимыми от любых изменений в классе редактора.

Получается, нам придётся либо открывать классы для всех желающих, испытывая массу хлопот с поддержкой кода, либо делать классы закрытыми, но отказаться от идеи отмены операций.

# Решение

- Все проблемы, описанные выше, возникают из-за нарушения инкапсуляции. Это когда одни объекты пытаются сделать работу за других, влезая в их приватную зону, чтобы собрать необходимые для операции данные.
- Паттерн **Снимок** поручает создание копии состояния объекта самому объекту, который этим состоянием владеет. Вместо того чтобы делать снимок «извне», наш редактор сам сделает копию своих полей — ведь ему доступны все поля, даже приватные.
- Паттерн предлагает держать копию состояния в специальном объекте «*снимке*» с ограниченным интерфейсом, позволяющим, например, узнать дату изготовления или название снимка. Но с другой стороны, снимок должен быть открыт для своего «*создателя*», позволяя прочесть и восстановить его внутреннее состояние.
- Эта схема позволяет создателям производить снимки и отдавать их для хранения другим объектам, называемым ***«опекунами».***
- Опекунам будет доступен только ограниченный интерфейс снимка, поэтому они никак не смогут повлиять на внутренности самого снимка. В нужный момент, опекун может попросить создателя восстановить своё состояние, передав в него соответствующий снимок.
- В нашем примере с редактором, **опекуном** можно сделать отдельный класс, который будет хранить список выполненных операций.
- Если пользователь решит откатить операцию, класс истории возьмёт последний снимок из стека и отправит его в редактор для восстановления.

# Структура

### Классическая реализация на вложенных классах

Классическая реализация паттерна полагается на механизм вложенных классов, которые доступны только в некоторых языках программирования (C++, C#, Java).

1. Создатель делает снимки своего состояния по запросу, а также воспроизводит прошлое состояние, если подать в него готовый снимок.
2. Снимок — это простой объект данных, содержащий состояние создателя. Надёжней всего сделать объекты снимков неизменяемыми и передавать в них состояние только через конструктор.
3. Опекун должен знать, когда и зачем делать снимок создателя, а также когда его нужно восстанавливать.
4. Опекун может хранить историю прошлых состояний создателя в виде стека из снимков. Если понадобится сделать отмену, он возьмёт последний снимок и передаст его создателю для восстановления.
5. В этой реализации снимок — это внутренний класс по отношению к классу создателя, поэтому тот имеет полный доступ к его полям и методам, несмотря на то, что они объявлены приватными. Опекун же не имеет доступа ни к состоянию, ни к методам снимков и может всего лишь хранить ссылки на эти объекты.

### Реализация с промежуточным пустым интерфейсом

Подходит для языков, не имеющих механизма вложенных классов (PHP).

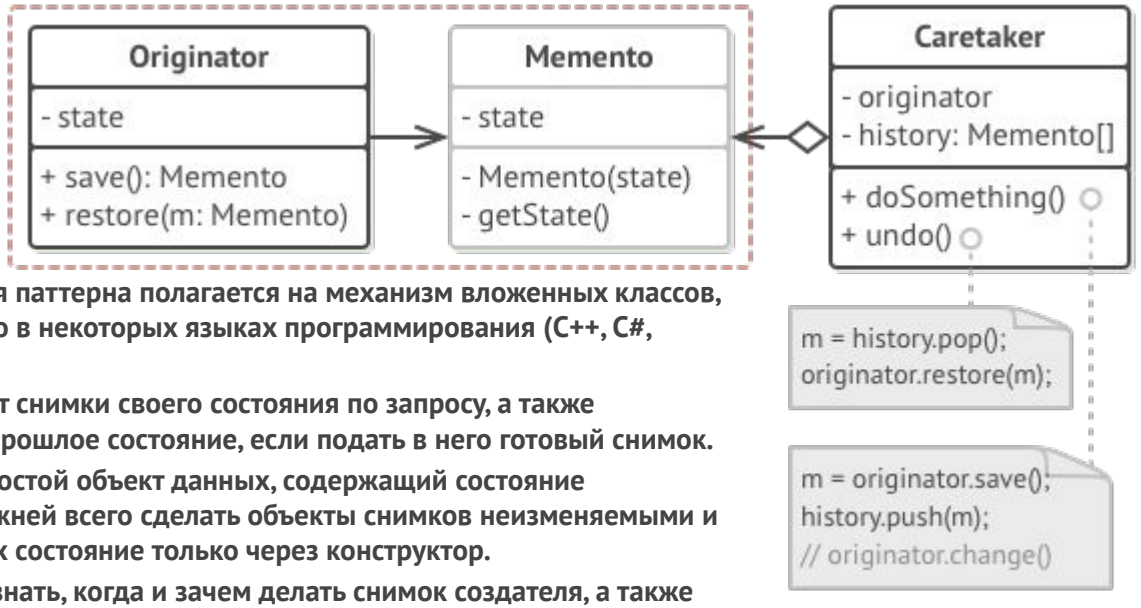
1. В этой реализации создатель работает напрямую с конкретным классом снимка, а опекун — только с его ограниченным интерфейсом.
2. Благодаря этому достигается тот же эффект, что и в классической реализации. Создатель имеет полный доступ к снимку, а опекун — нет.

### Снимки с повышенной защитой

Когда нужно полностью исключить доступ к состоянию Создателей и Снимков.

1. Реализация разрешает иметь несколько видов создателей и снимков. Каждому классу создателей соответствует собственный класс снимков. Ни создатели, ни снимки не позволяют прочесть их состояние.
2. Здесь опекун ещё более жёстко ограничен в доступе к состоянию создателей и снимков. Но с другой стороны, опекун становится независим от создателей, так как метод восстановления теперь находится в самих снимках.
3. Снимки теперь связаны с теми создателями, из которых они сделаны. Они по-прежнему получают состояние через конструктор. Благодаря близкой связи между классами, снимки знают, как восстановить состояние своих создателей.

# Классическая реализация на вложенных классах



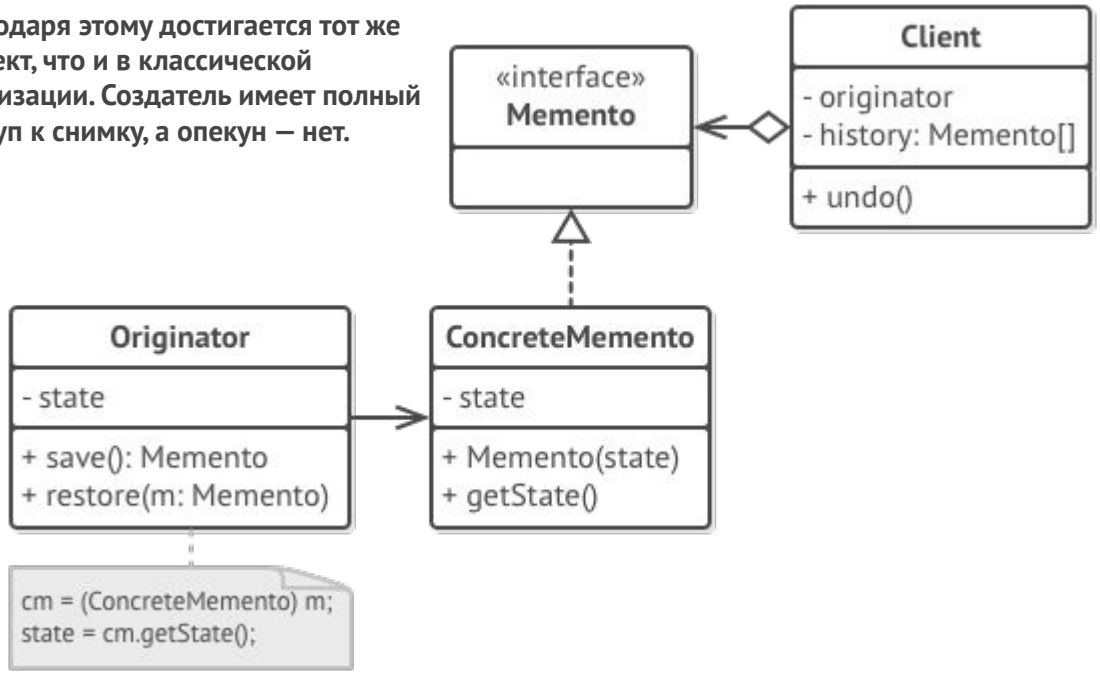

**Классическая реализация паттерна полагается на механизм вложенных классов, которые доступны только в некоторых языках программирования (C++, C#, Java).**

1. **Создатель делает снимки своего состояния по запросу, а также воспроизводит прошлое состояние, если подать в него готовый снимок.**
2. **Снимок — это простой объект данных, содержащий состояние создателя. Надёжней всего сделать объекты снимков неизменяемыми и передавать в них состояние только через конструктор.**
3. **Опекун должен знать, когда и зачем делать снимок создателя, а также когда его нужно восстанавливать.**
4. **Опекун может хранить историю прошлых состояний создателя в виде стека из снимков. Если понадобится сделать отмену, он возьмёт последний снимок и передаст его создателю для восстановления.**
5. **В этой реализации снимок — это внутренний класс по отношению к классу создателя, поэтому тот имеет полный доступ к его полям и методам, несмотря на то, что они объявлены приватными. Опекун же не имеет доступа ни к состоянию, ни к методам снимков и может всего лишь хранить ссылки на эти объекты.**

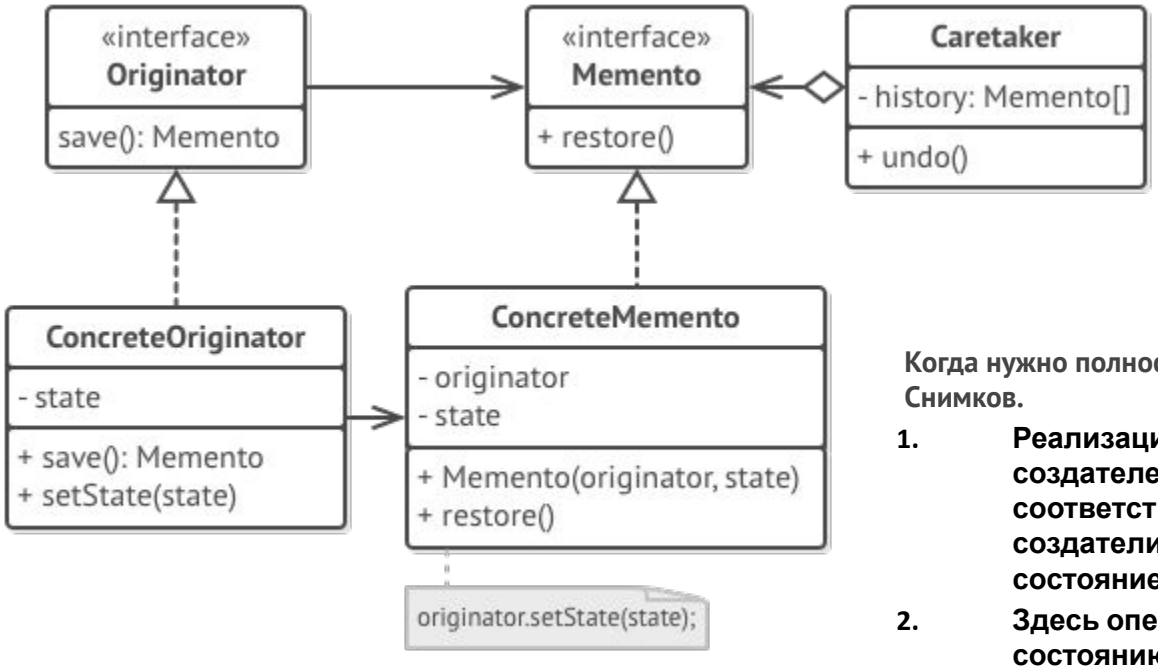
# Реализация с промежуточным пустым интерфейсом

**Подходит для языков, не имеющих механизма вложенных классов (PHP).**

1. **В этой реализации создатель работает напрямую с конкретным классом снимка, а опекун — только с его ограниченным интерфейсом.**
2. **Благодаря этому достигается тот же эффект, что и в классической реализации. Создатель имеет полный доступ к снимку, а опекун — нет.**



# Снимки с повышенной защитой




**Когда нужно полностью исключить доступ к состоянию Создателей и Снимков.**

1. **Реализация разрешает иметь несколько видов создателей и снимков. Каждому классу создателей соответствует собственный класс снимков. Ни создатели, ни снимки не позволяют прочесть их состояние.**
2. **Здесь опекун ещё более жёстко ограничен в доступе к состоянию создателей и снимков. Но с другой стороны, опекун становится независим от создателей, так как метод восстановления теперь находится в самих снимках.**
3. **Снимки теперь связаны с теми создателями, из которых они сделаны. Они по-прежнему получают состояние через конструктор. Благодаря близкой связи между классами, снимки знают, как восстановить состояние своих создателей.**

# Пример

- В этом примере паттерн Снимок используется совместно с паттерном **Команда** и позволяет хранить резервные копии сложного состояния текстового редактора и, если потребуется, восстанавливать его.
- Объекты команд выступают в роли опекунов и запрашивают снимки у редактора перед тем, как выполнить своё действие. Если потребуется отмена операции, команда сможет восстановить состояние редактора, используя сохранённый снимок.
- При этом снимок не имеет публичных полей, поэтому никакой объект не сможет получить доступа его данным. Снимки связаны с определённым редактором, которых их создал и сам восстанавливает его состояние. Это позволяет программе иметь одновременно несколько объектов редакторов, например, разбитых по вкладкам.

# Класс Редактор

```
// Класс создателя должен иметь специальный метод, который сохраняет состояние
 // создателя в новом объекте-снимке.
 class Editor is
 private field text: string
 private field cursorX, cursorY, selectionWidth
method setText(text) is
 this.text = text
 method setCursor(x, y) is
 this.cursorX = cursorX this.cursorY = cursorY
 method selectionWidth(width) is
 this.selectionWidth = width
 method saveState():EditorState is
// Снимок — неизменяемый объект, поэтому Создатель передаёт все своё
// состояние через параметры конструктора.
return new EditorState(this, text, cursorX, cursorY, selectionWidth)
```

# Состояние редактора

```
// Снимок хранит прошлое состояние редактора.
 class EditorState is
 private field editor: Editor
private field text: string
private field cursorX, cursorY, selectionWidth
constructor EditorState(editor, text, cursorX, cursorY, selectionWidth) is this.editor =
editor
 this.text = text
this.cursorX = cursorX
= cursorY this.selection
Width = selectionWidth
// В нужный момент, владелец снимка может восстановить состояние
редактора.
 method restore() is
editor.setText(text) editor.setCursor(cursorX, cursorY)
editor.selectionWidth(selectionWidth)
```

# Опекун

```
// Опекуном может выступать класс команд (см. паттерн Команда). В этом случае,
 // команда сохраняет снимок получателя перед тем, как выполнить действие. А при
// отмене, возвращает получателя в предыдущее состояние.
class Command is
private field backup: EditorState
 method backup() is
 backup = editor.saveState()
method undo() is
 if (backup != null)
backup.restore() // ...
```

# Применимость

- **Когда вам нужно сохранять мгновенный снимок состояния объекта (или его части), чтобы впоследствии объект можно было восстановить в том же состоянии.**

Паттерн Снимок позволяет делать любое количество снимков объекта и хранить их независимо от объекта, с которого делают снимок.

Снимки часто используют не только для реализации операции отмены, но и для транзакций, когда состояние объекта нужно откатить, если операция не удалась.

- **Когда прямое получение состояния объекта раскрывает детали его реализации и нарушает инкапсуляцию.**

Паттерн предлагает изготовить снимок самому исходному объекту, так как ему доступны все поля, даже приватные.

# Шаги реализации

1. Определите класс создателя, объекты которого должны создавать снимки своего состояния.
2. Создайте класс снимка и опишите в нём все те же поля, которые имеются в оригинальном классе-создателе.
3. Сделайте объекты снимков неизменяемыми. Они должны получать начальные значения только один раз, через свой конструктор.
4. Если ваш язык программирования это позволяет, сделайте класс снимка вложенным в класс создателя.
5. Добавьте в класс создателя метод получения снимков. Создатель должен создавать новые объекты снимков, передавая значения своих полей через конструктор.
6. Сигнатура метода должна возвращать снимки через ограниченный интерфейс, если он у вас есть. Сам класс должен работать с конкретным классом снимка.
7. Добавьте в класс создателя метод восстановления из снимка..
8. Опекуны, будь то история операций, объекты команд или нечто иное, должны знать о том, когда запрашивать снимки у создателя, где их хранить, и когда восстанавливать.
9. Связь опекунов с создателями можно перенести внутрь снимков. В этом случае каждый снимок будет привязан к своему создателю и должен будет сам восстанавливать его состояние. Но это будет работать либо если классы снимков вложены в классы создателей, либо если создатели имеют сеттеры для установки значений своих полей.

# Преимущества и недостатки

- Не нарушает инкапсуляции исходного объекта.(+)
- Упрощает структуру исходного объекта. Ему не нужно хранить историю версий своего состояния.(+)
- Требует много памяти, если клиенты слишком часто создают снимки.(-)
- Может повлечь дополнительные издержки памяти, если объекты, хранящие историю, не освобождают ресурсы, занятые устаревшими снимками. (-)
- В некоторых языках (например, PHP, Python, JavaScript) сложно гарантировать, чтобы только исходный объект имел доступ к состоянию снимка. (-)

# Отношения с другими паттернами

- **Команду** и **Снимок** можно использовать сообща для реализации отмены операций. В этом случе объекты команд будут отображать выполненные действие над объектом, снимки — хранить копию состояния этого объекта до того, как команда была выполнена.
- **Снимок** можно использовать вместе с **Итератором**, чтобы сохранить текущее состояние обхода структуры данных и вернуться к нему в будущем, если потребуется.
- **Снимок** иногда можно заменить **Прототипом**, если объект, чьё состояние требуется сохранять в истории, довольно простой, не имеет активных ссылок на внешние ресурсы, либо их можно легко восстановить.

# Литература


- https://refactoring.guru/ru/design-patterns
- **Погружение в ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ** v2 020-2.26 Стр.216

**Тепляков С.** Т34 Паттерны проектирования на платформе NET. — СПб.: Питер, 2015. — 320 с.: ил. ISBN 978-5-496-01649-0