Паттерны проектирования GOF (поведенческие)

Технологии разработки программного обеспечения

Виноградова М.В.
МГТУ им. Н.Э. Баумана
Кафедра СОИУ (ИУ5)

Поведенческие паттерны -1

- **Команда (Command)** - представляет действие. Объект команды заключает в себе само действие и его параметры.
- **Стратегия (Strategy)** - предназначен для определения семейства алгоритмов, инкапсуляции каждого из них и обеспечения их взаимозаменяемости.
- **Шаблонный метод (Template method)** - определяет основу алгоритма и позволяет наследникам переопределять некоторые шаги алгоритма, не изменяя его структуру в целом.
- **Наблюдатель (Observer)** - определяет зависимость типа «один ко многим» между объектами таким образом, что при изменении состояния одного объекта все зависящие от него оповещаются об этом событии.
- **Цепочка обязанностей (Chain of responsibility)** - предназначен для организации в системе уровней ответственности.
- **Итератор (Iterator)** - дает возможность последовательно обойти все элементы составного объекта, не раскрывая его внутреннего представления.

Поведенческие паттерны - 2

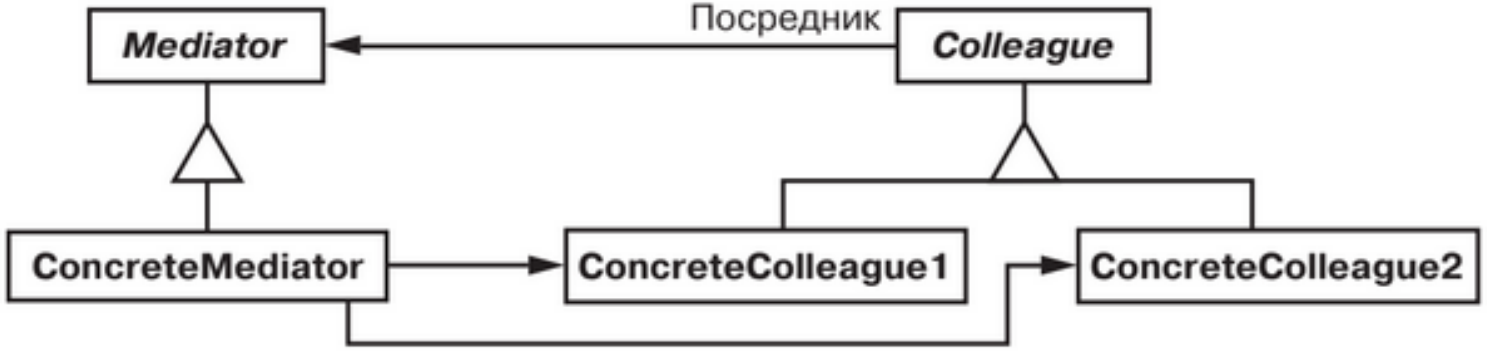
- **Интрерпретатор (Interpreter)** - для заданного языка определяет представление его грамматики, а также интерпретатор предложений языка, использующий это представление.
- **Посредник (Mediator)- о**пределяет объект, в котором инкапсулировано знание о том, как взаимодействуют объекты из некоторого множества. Способствует уменьшению числа связей между объектами, позволяя им работать без явных ссылок друг на друга. Это, в свою очередь, дает возможность независимо изменять схему взаимодействия.
- **Состояние** (**State) - п**озволяет объекту варьировать свое поведение при изменении внутреннего состояния. При этом создается впечатление, что поменялся класс объекта.
- **Хранитель** (**Memento) -** позволяет, не нарушая инкапсуляции, получить и сохранить во внешней памяти внутреннее состояние объекта, чтобы позже объект можно было восстановить точно в таком же состоянии.
- **Посетитель** (**Visitor)** - представляет операцию, которую надо выполнить над элементами объекта. Позволяет определить новую операцию, не меняя классы элементов, к которым он применяется.

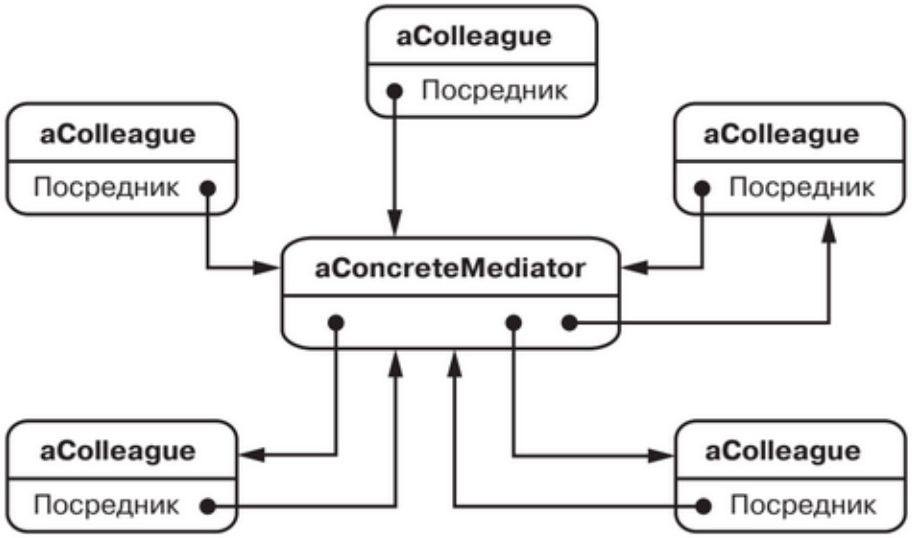
Посредник (Mediator)

- Определяет объект, содержащий способ взаимодействия множества объектов. Обеспечивает слабое сцепление системы, избавляя объекты от необходимости явно ссылаться друг на друга и позволяя тем самым независимо изменять взаимодействие между ними.

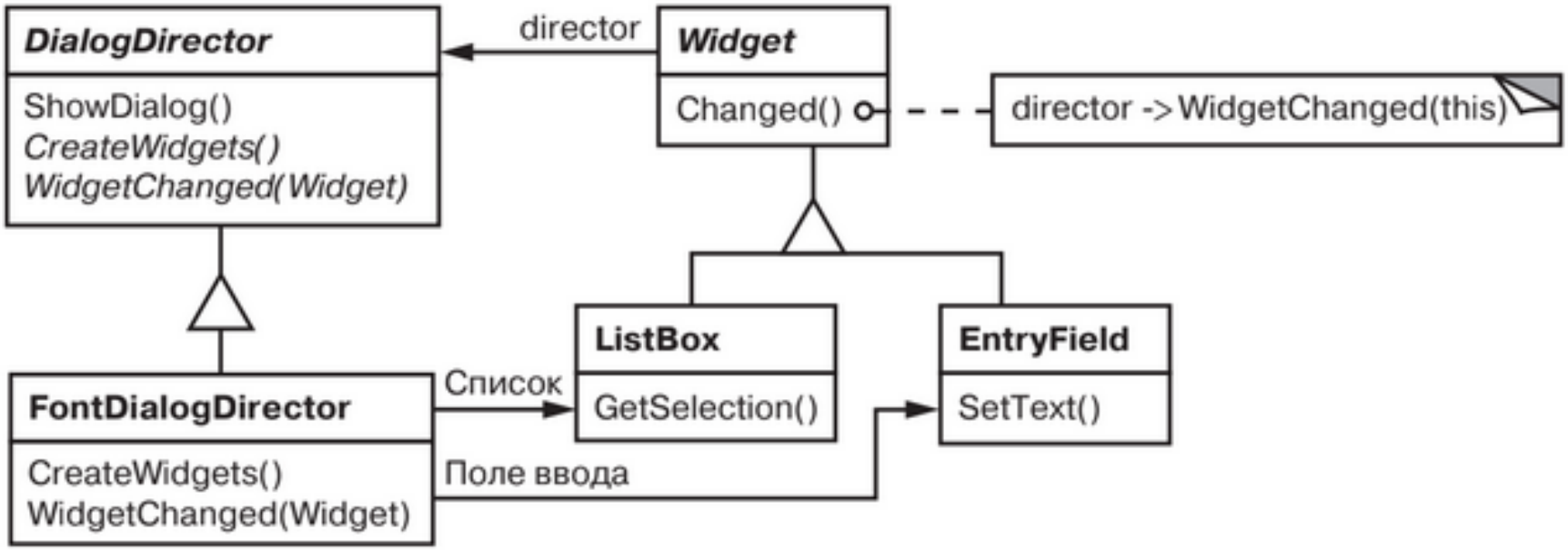
- Проблемы:
  - связи между объектами сложны, четко определены, не структурированы, сложны для понимания;
  - нельзя повторно использовать бъект, т.к. он сильно сцеплен;
  - поведение между классами должно быть настраиваемо без порождения подклассов;
- Применение:
  - Приложение -> элемент управления -> сложное взаимодействие (при вводе в 1 поле => активация других элементов и появление определенных данных).

Посредник - решение

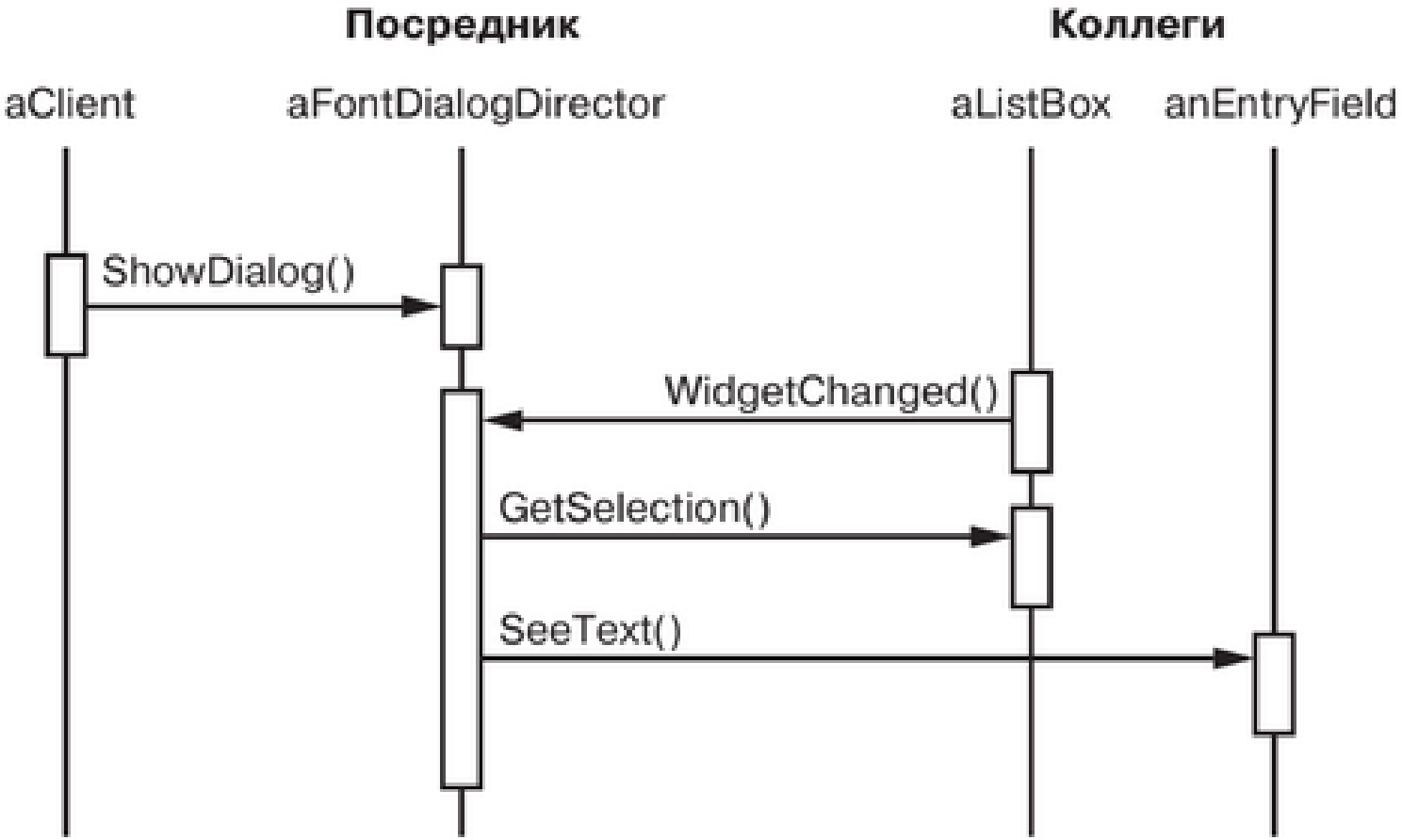




Посредник - пример



Посредник - пример



Посредник - результат

- (+) уменьшается количество подклассов;
- (+) уменьшается сцепление коллег -> 0 (независимое изменение коллег);
- (+) упрощаются протоколы взаимодействия объектов (M-M) => (1-M);
- (+) абстракция и инкапсуляция кооперации объектов;
- (-) централизация управления (монолит, сложен).

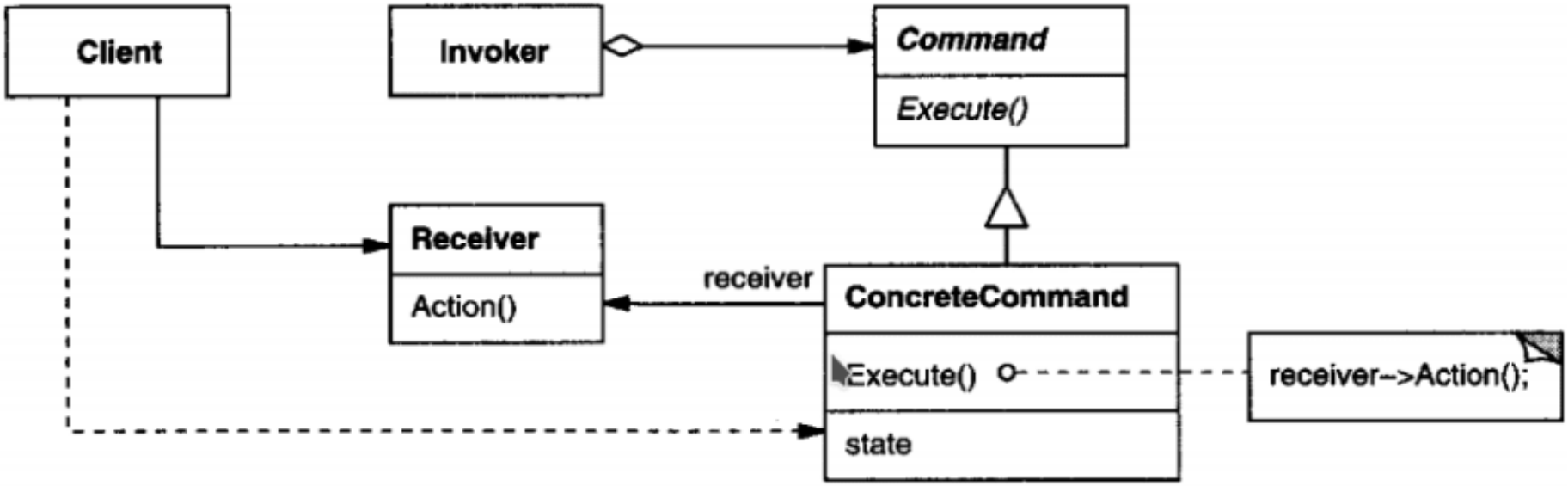
- Коллеги взаимодействуют с посредником через наблюдателя;
- Фасад прозрачен, а Посредник двунаправлен.

Команда (Command)

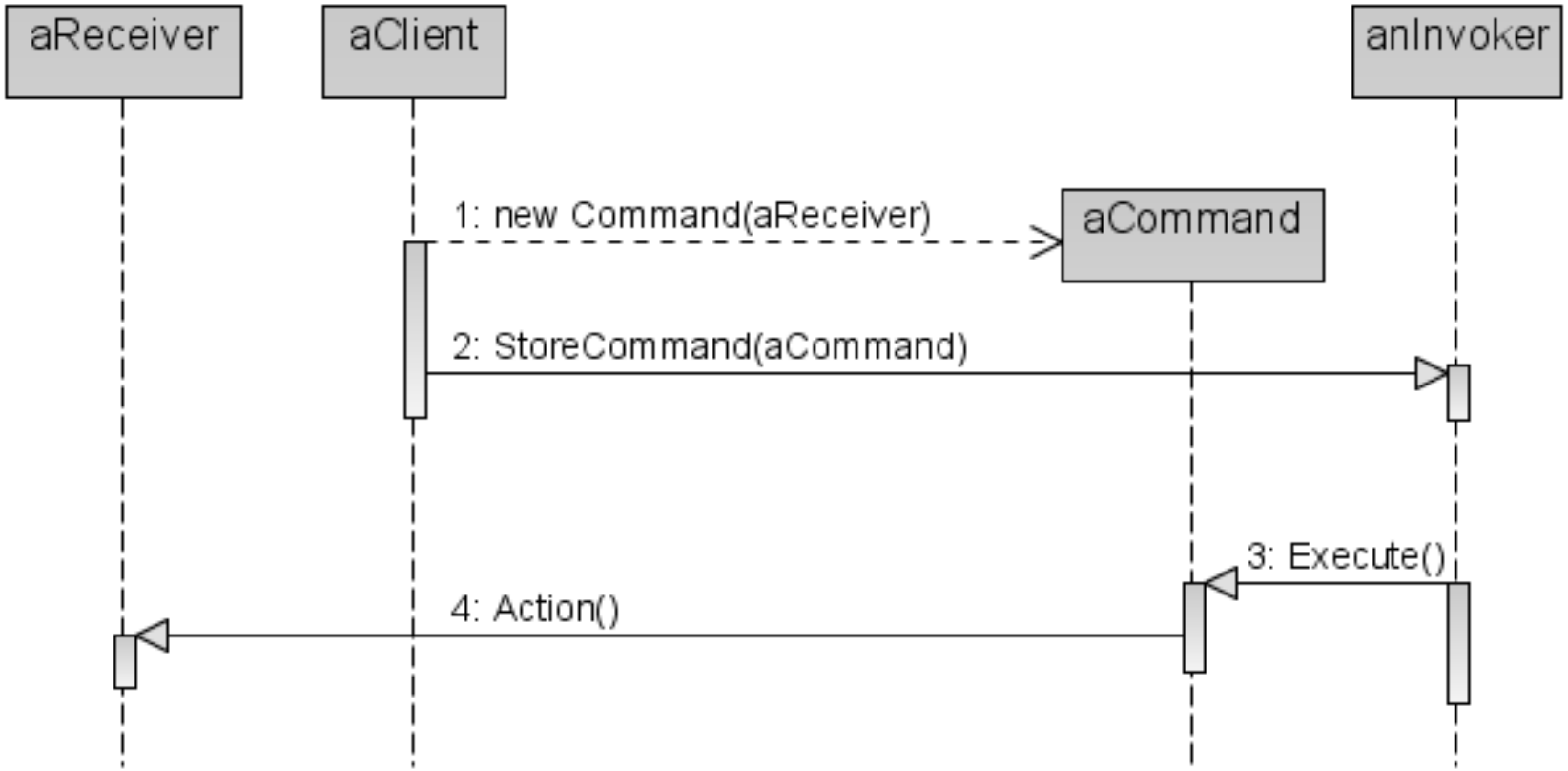
- Выполняет преобразование запроса в объект, обеспечивая:
  - параметризацию клиентов в различных запросах;
  - постановку запросов в очередь и их регистрацию;
  - поддержку отмены операций.

- Проблема:
  - Разделение: объект, посылающий запрос, и объект, выполняющий запрос => увеличивается гибкость GUI (пункт меню -> функция, динамическое изменение)
- Применение:
  - замена функции обратного вызова;
  - история операций и отмена операции;
  - регистрация изменений состояния для восстановления после сбоя;
  - создание сложных операций на базе простых.

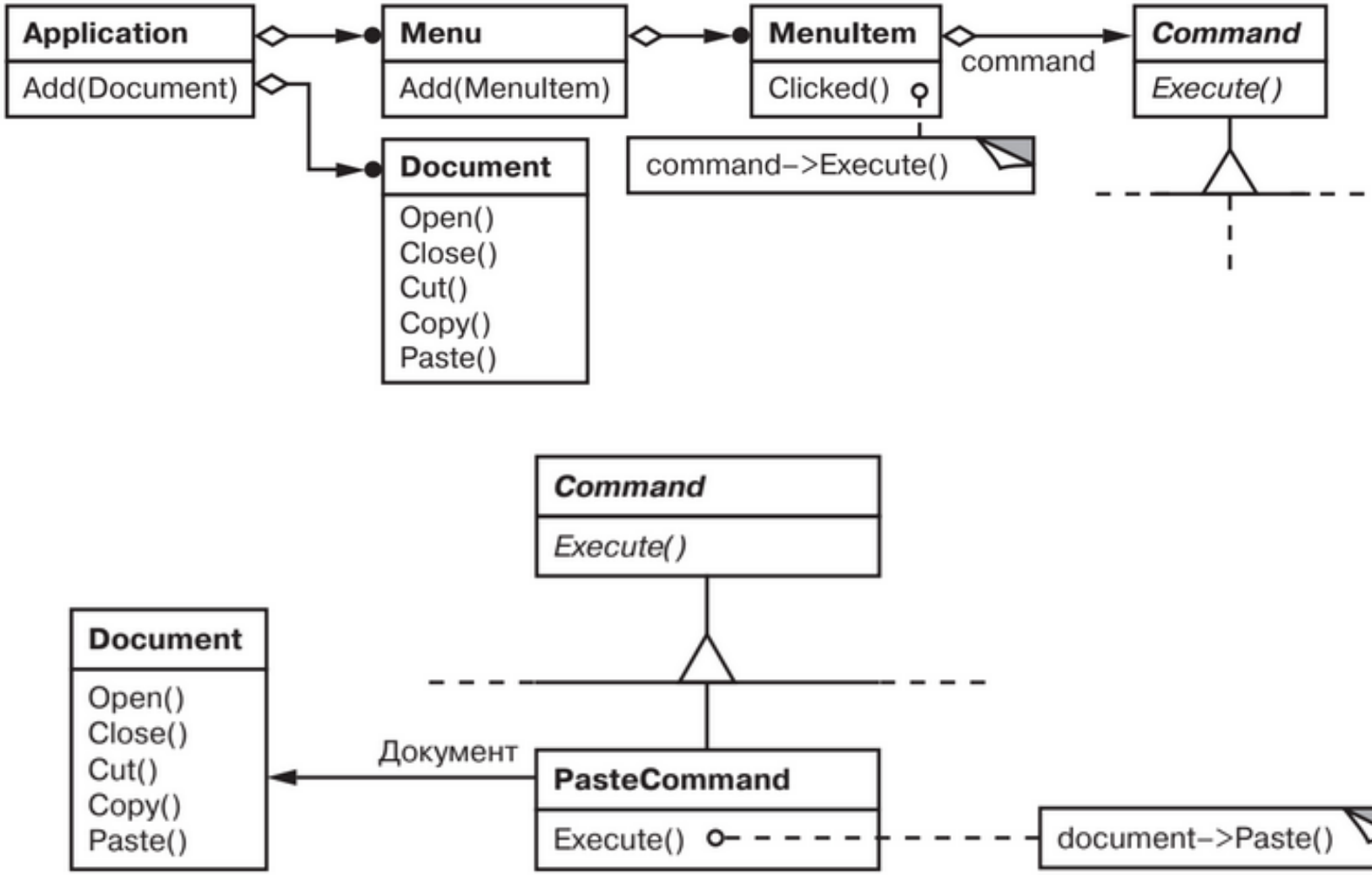
Команда - решение



Команда - решение



Команда - пример



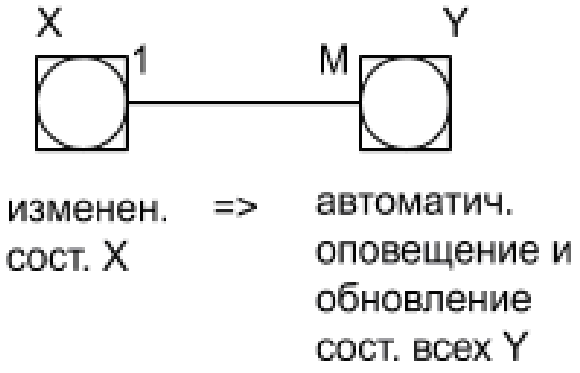
Команда - результат

- Разделение инициатора и получателя.
- Команды - полноценные объекты (с возможностью расширения).
- Компоновка составных команд; легко добавлять новые команды.

Наблюдатель (observer)

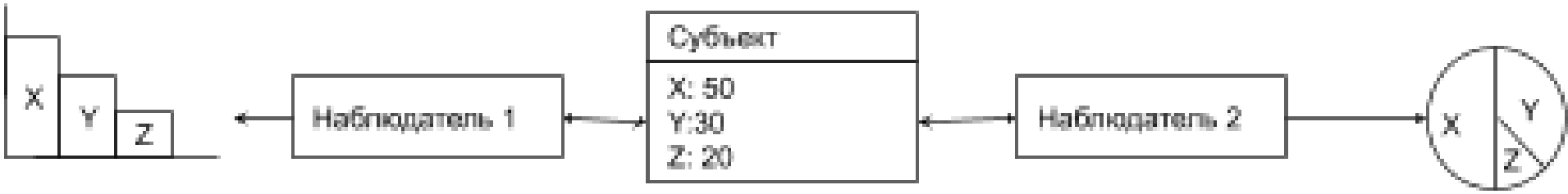
- Определяет зависимость 1-M между объектами таким образом, чтобы при изменении 1 объекта все зависящие от него оповещались об этом и автоматически обновлялись

- Проблема:
  - X не знает об Y и их количестве и не делает предположений об Y => уменьшение сцепления;
  - Может быть применено, если непрямое взаимодействие объектов уровня логики приложения и GUI => уменьшение сцепления уровней.

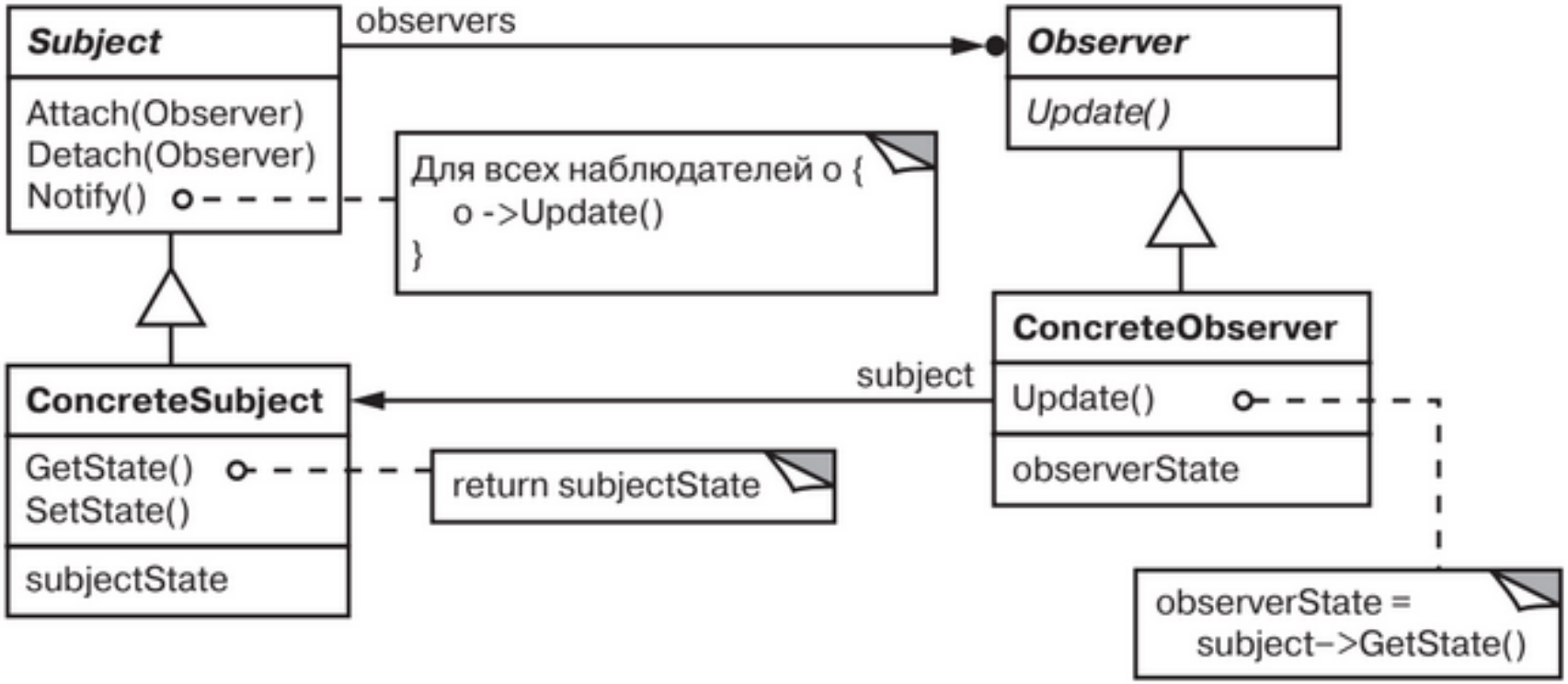


Наблюдатель - идея

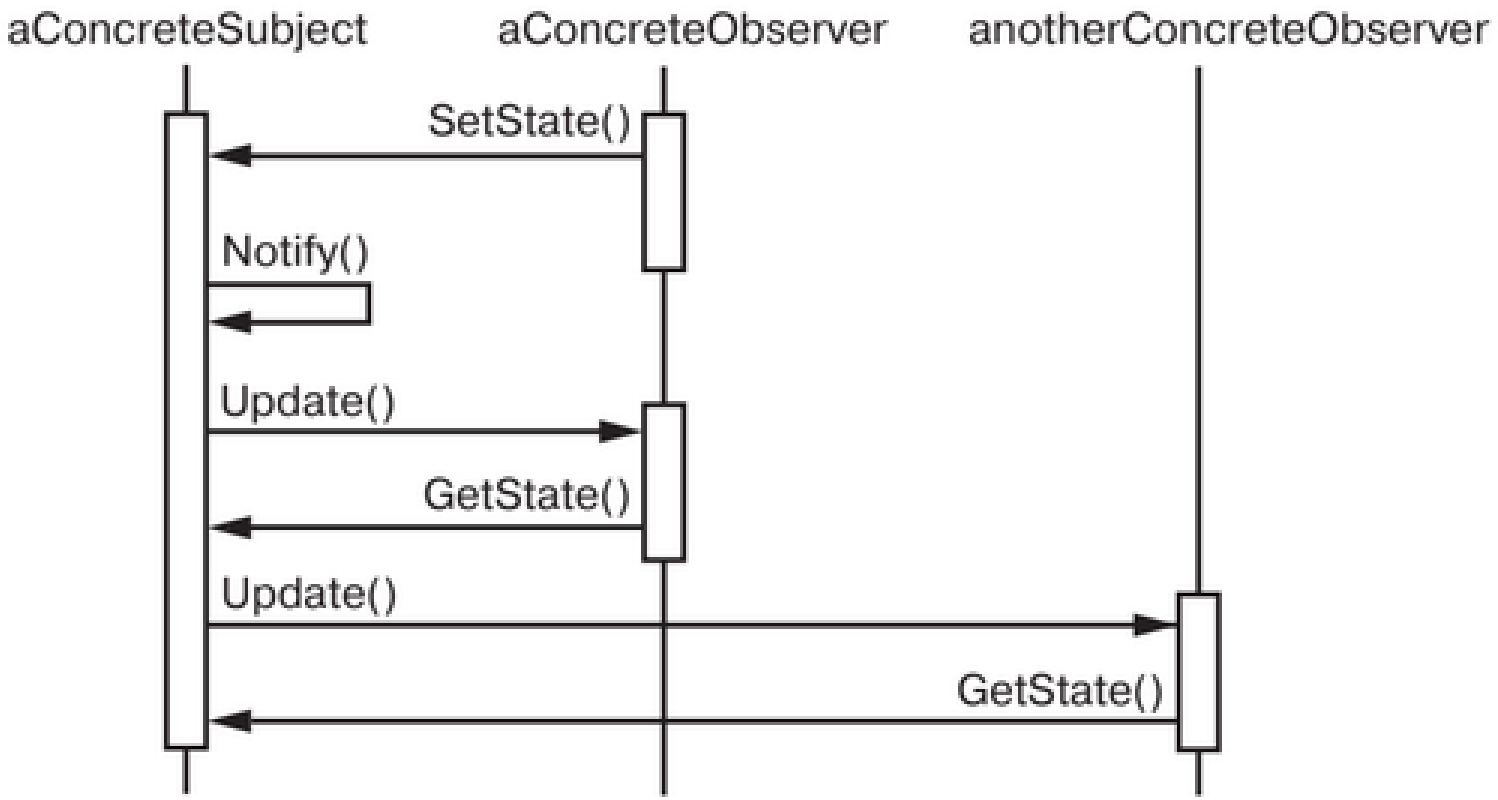
- Схема издатель-подписчик (1-N);
- Издатель не знает о подписчике;
- Подписчик автоматически уведомляется об изменении издателя;
- Подписчик опрашивает субъект (издателя) для синхронизации состояний.



Наблюдатель - решение



Наблюдатель - решение



Наблюдатель - результат

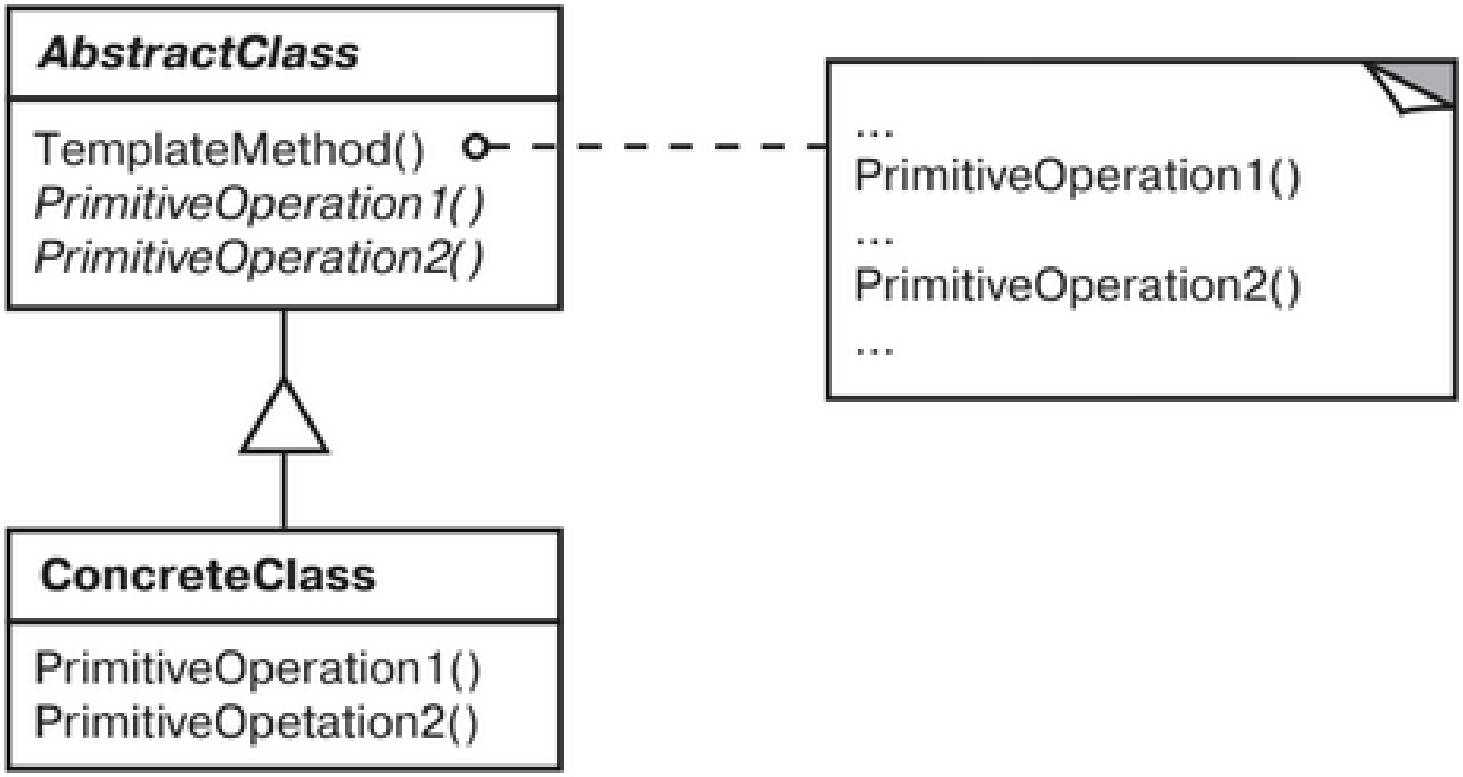
- (+) субъект знает об абстрактном наблюдателе, но не о конкретном => минимальное сцепление;
- (-) изменения в субъекте => очень много лишних изменений в наблюдателях.

Шаблонный метод (Template Method)

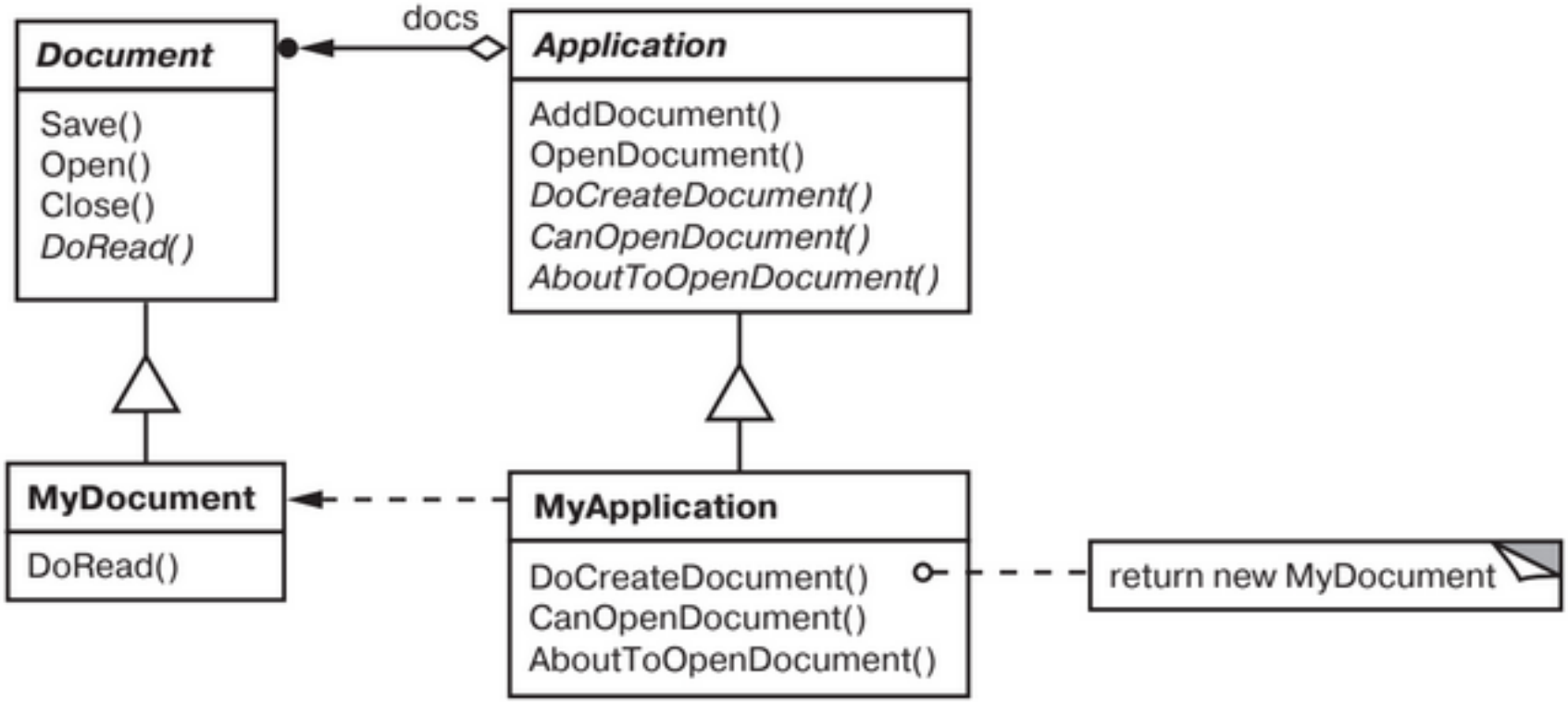
- Определяет основу алгоритма и позволяет подклассам переопределить некоторые шаги алгоритма, не изменяя его структуру в целом.

- Проблема:
  - Однократное использование общей части алгоритма (изменяется только часть алгоритма в подклассах);
  - Выделение общей части кода (для избежания дублирования в подклассах);
  - Управление расширениями подклассов (шаблонный метод вызывает операции-зацепки в определенных точках).
- Применение:
  - Каркас: приложение (абстр+шабл.) + документ (абстр.) -> наследование для конкретных документов;
  - Шаблон описывает алгоритм в терминах абстрактных функций.

Шаблонный метод - решение



Шаблонный метод - пример



Шаблонный метод - результат

- Основа повторного использования (особенно для библиотеки классов).
- Шаблонный метод вызывает:
  - конкретные операции подклассов/классов клиента;
  - операции базового класса;
  - примитивные операции (абстракции) -> нужно заменить;
  - фабричный метод;
  - операции-зацепки (поведение по умолчанию) -> можно заменить.

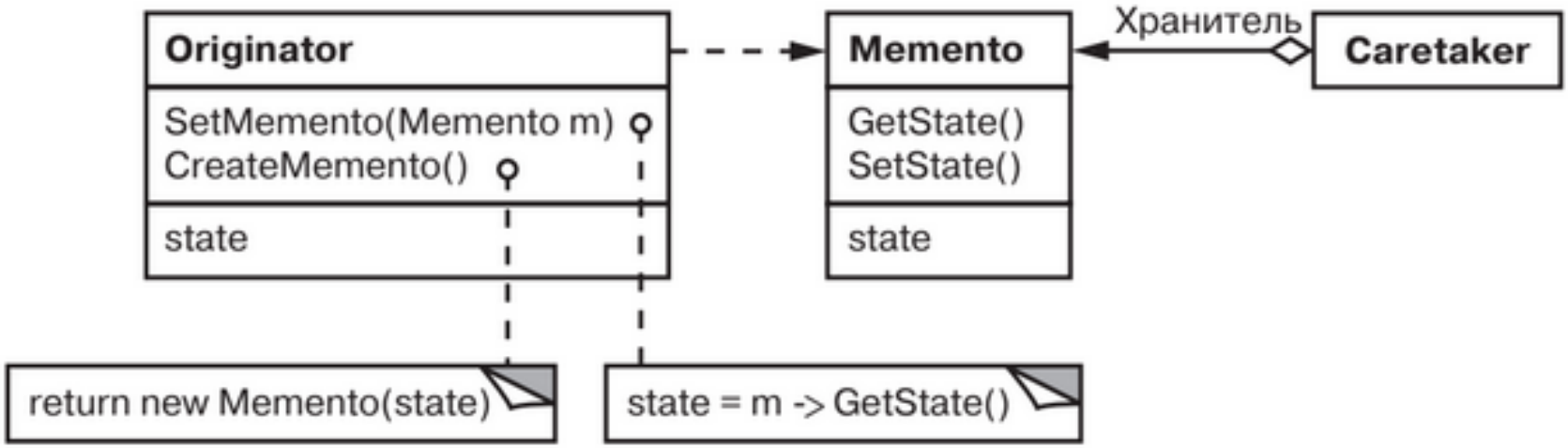
- Фабричный метод вызывается шаблонным методом.
- Шаблонный метод использует наследование для модификации части алгоритма, а Стратегия - делегирование .

Хранитель (memento)

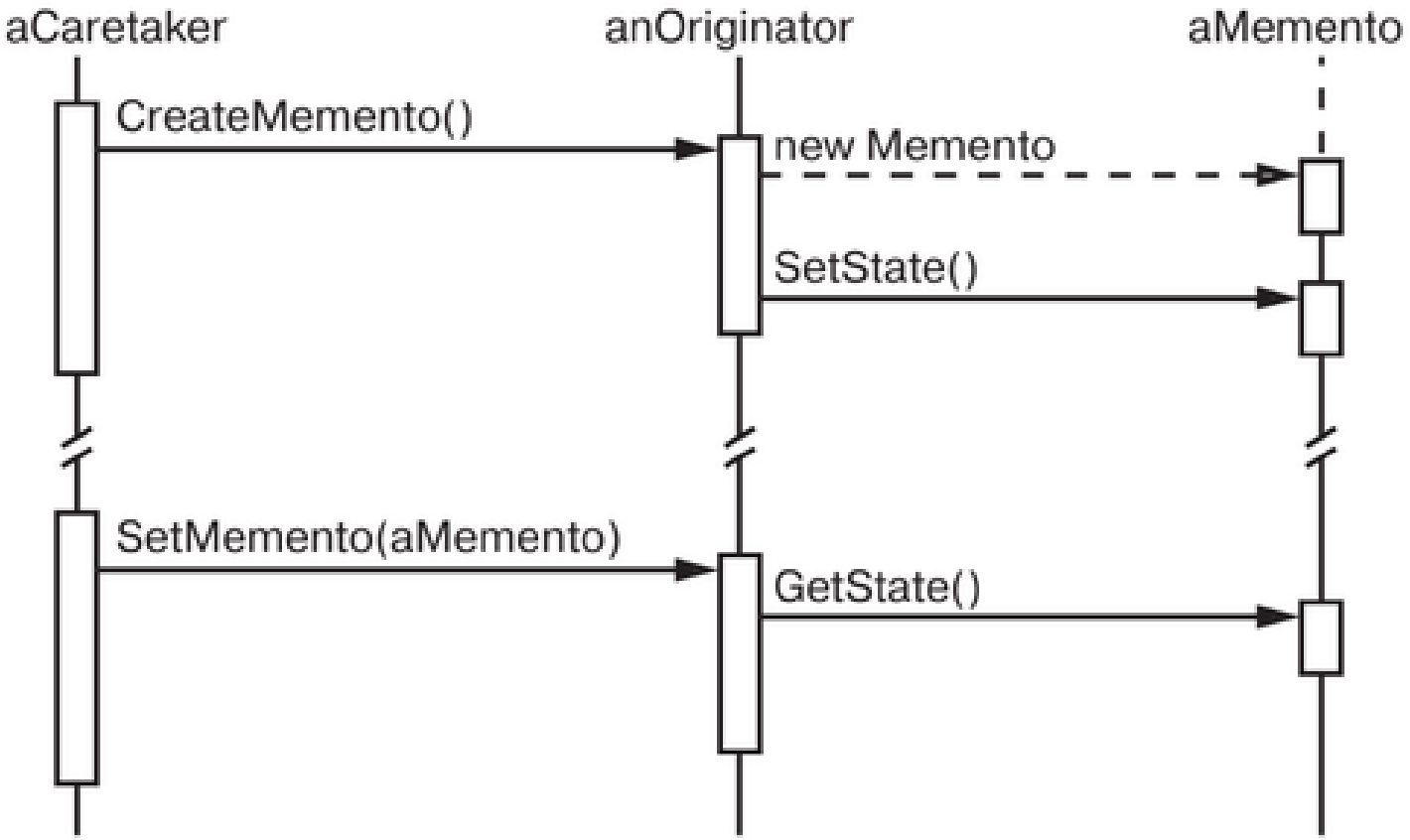
- Не нарушая инкапсуляции, фиксирует и выносит за пределы объекта его внутреннее состояние так, чтобы позже можно было восстановить в нем объект.

- Проблема:
  - необходимо сохранить мгновенный снимок состояния объекта/части для последующего восстановления;
  - если прямое получение состояния объекта открывает детали реализации и нарушает инкапсуляцию.

Хранитель - решение



Хранитель - решение



Хранитель - результат

- (+) Сохранение границ инкапсуляции.
- (+) Упрощение структуры хозяина (использование ресурсов для хранения вне хозяина).
- (-) Большие издержки для использования хранителей.
- (-) “Узкий” и “широкий” интерфейсы сложны для реализации.
- (-) Скрытие информации об ОП хранителя от посыльного.

- Команда может быть хозяином хранителя (откат изменений).
- Хранитель + итератор (переход к следующему действию).

Цепочка обязанностей (Chain of Responsibility)

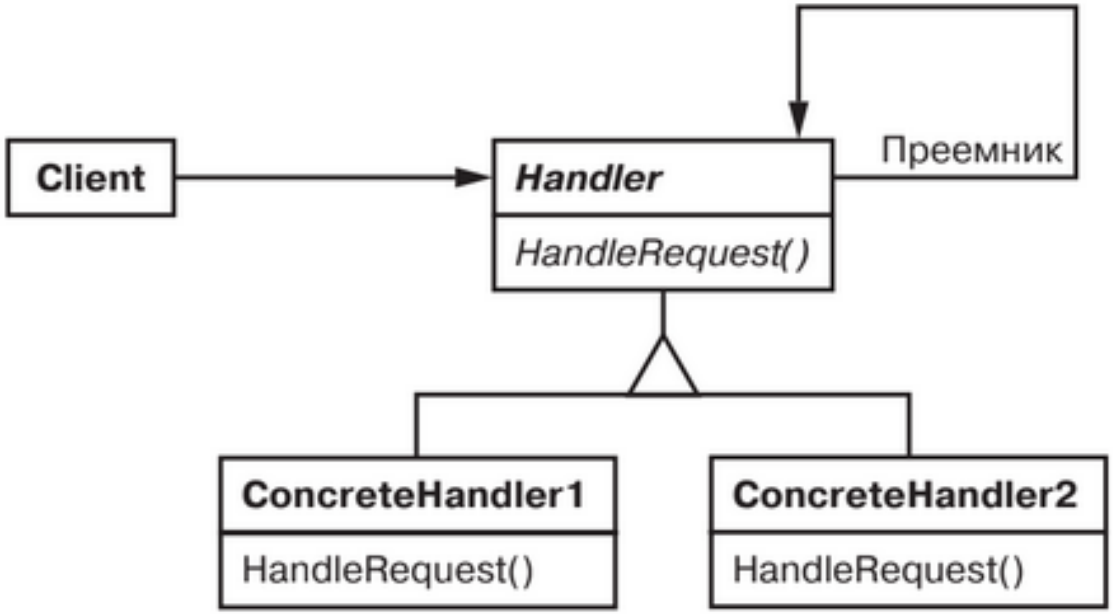
- Позволяет избежать привязки отправителя запроса к его получателю, давая шанс обработать запрос нескольким объектам. Связывает объекты получатели в цепочку и передает запрос вдоль этой цепочки, пока его не обработают.

- Проблемы:
  - несколько обработчиков запроса: конкретный получатель неизвестен и должен быть найден автоматически;
  - отправка запроса к 1 из n объектов, не указывая явно кому;
  - набор объектов, могущих обработать запрос, должен определяться автоматически.
- Применение:

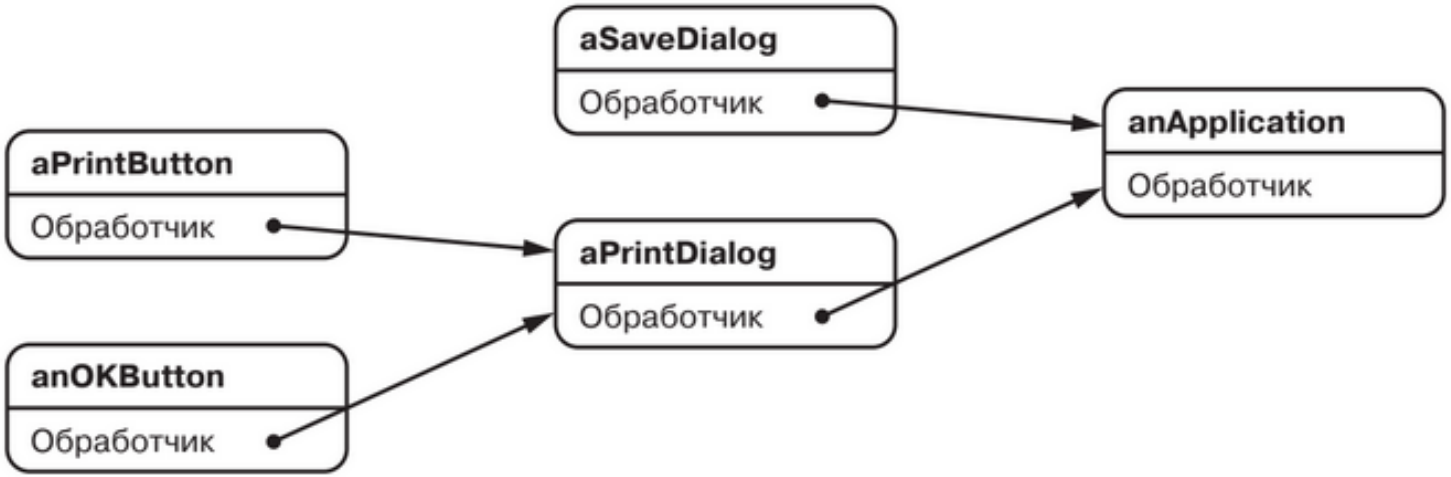
  (Приложение -> окно -> элемент;

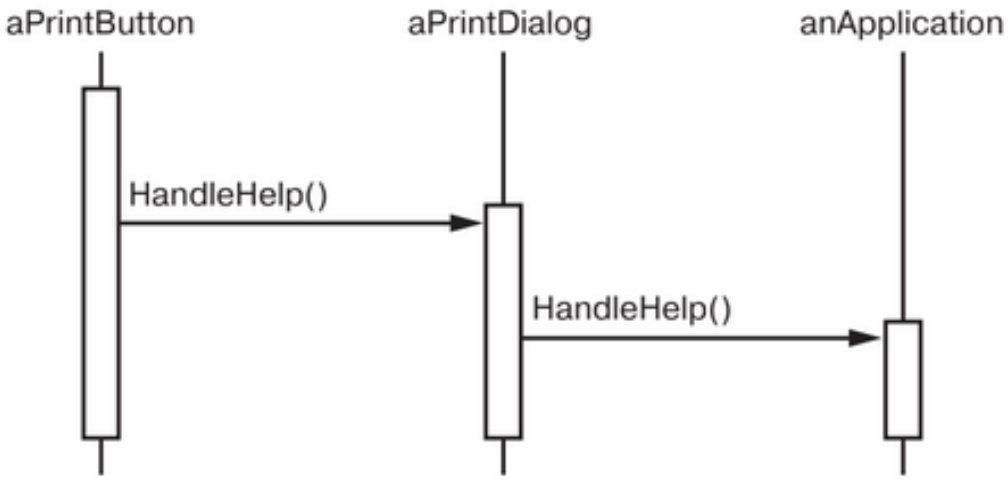
  справка по элементу -> да/ окно -> да/ приложение)

Цепочка обязанностей - решение



Цепочка обязанностей - пример





Цепочка обязанностей - результат

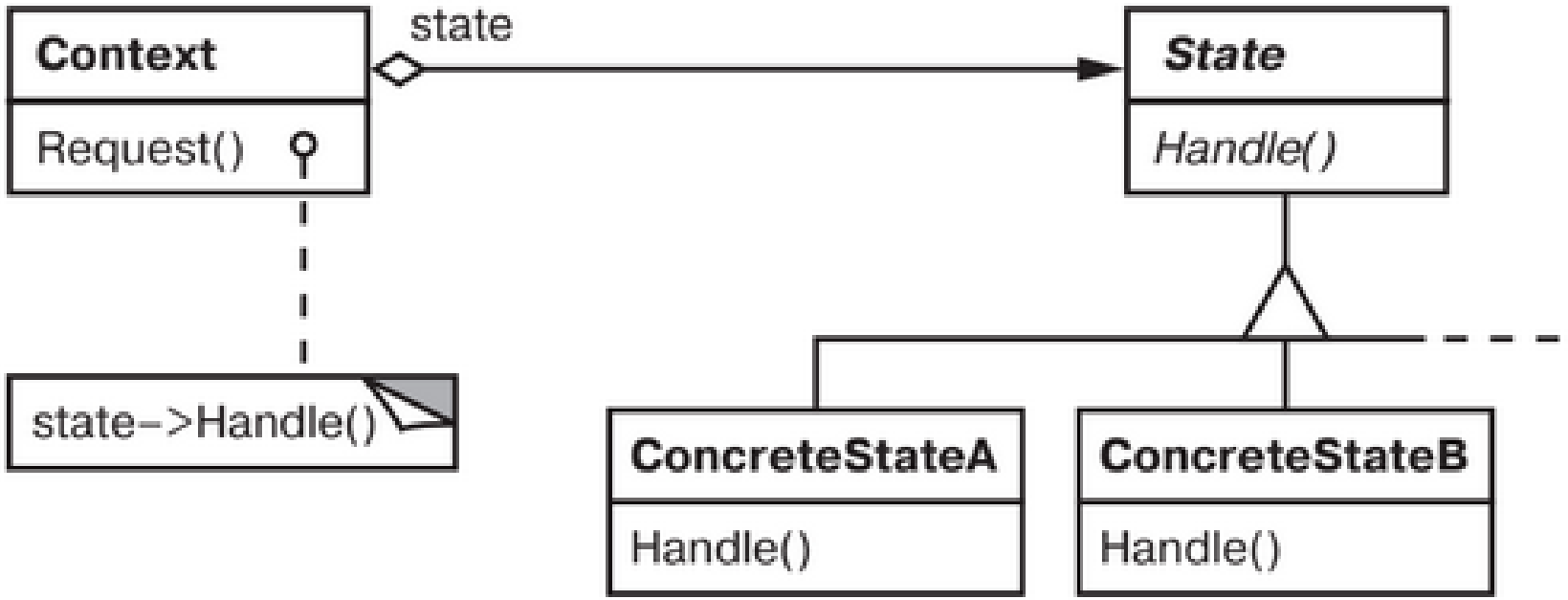
- Уменьшение сцепления клиента и обработчика запросов;
- сокрытие структуры цепочки (отправителя и получателя).
- Увеличение гибкости при распределении обязанностей между объектами (динамическое добавление/удаление в цепочке)
- (-) получение не гарантировано.
  - запрос никем не обработался и пропал,
  - неправильная конфигурация цепочки.

Состояние (State)

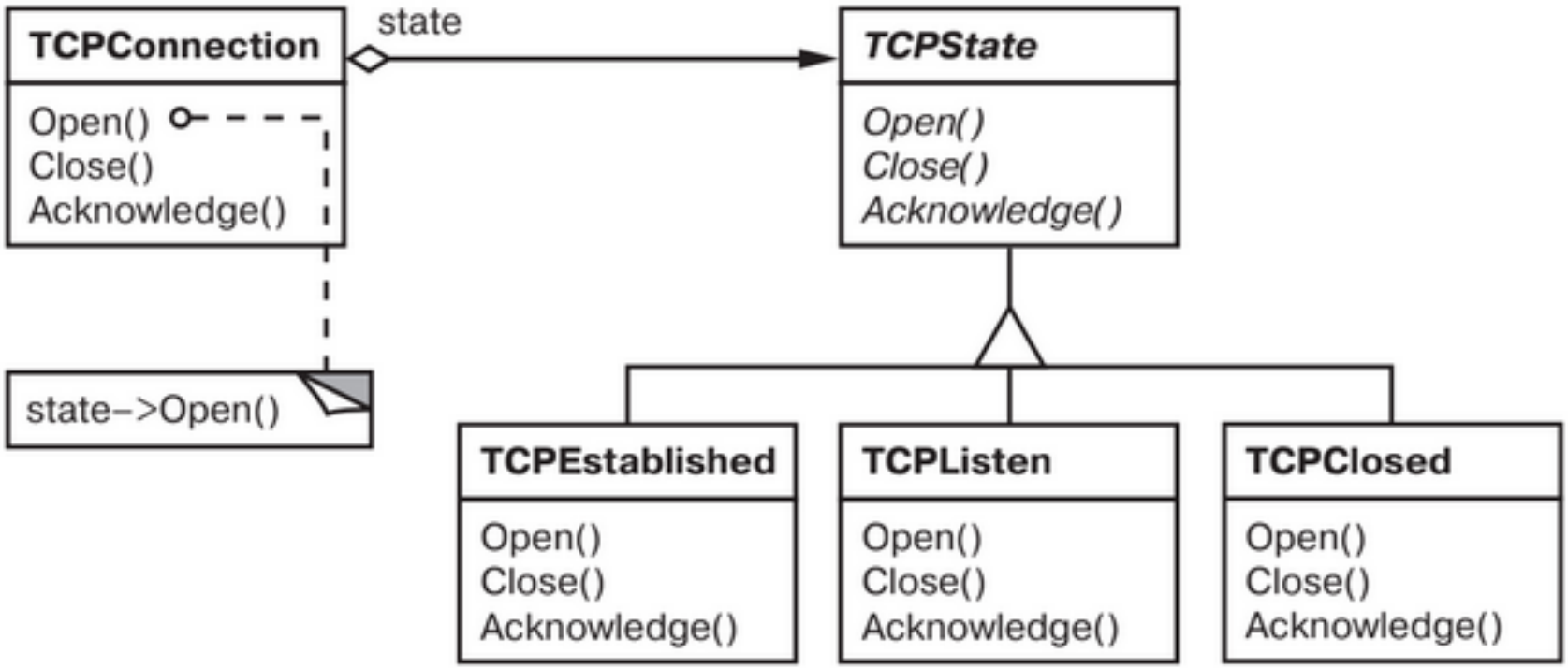
- Позволяет объекту варьировать свое поведение в зависимости от внутреннего состояния. Извне создается впечатление, что изменился класс объекта.

- Проблема:
  - поведение объекта зависит от его состояния и должно изменяться во времени,
  - если в коде операций есть условный оператор из многих ветвей (выбор ветви зависит от состояния – перечисление констант) => любая ветка выносится в отдельный класс, который может изменяться независимо.

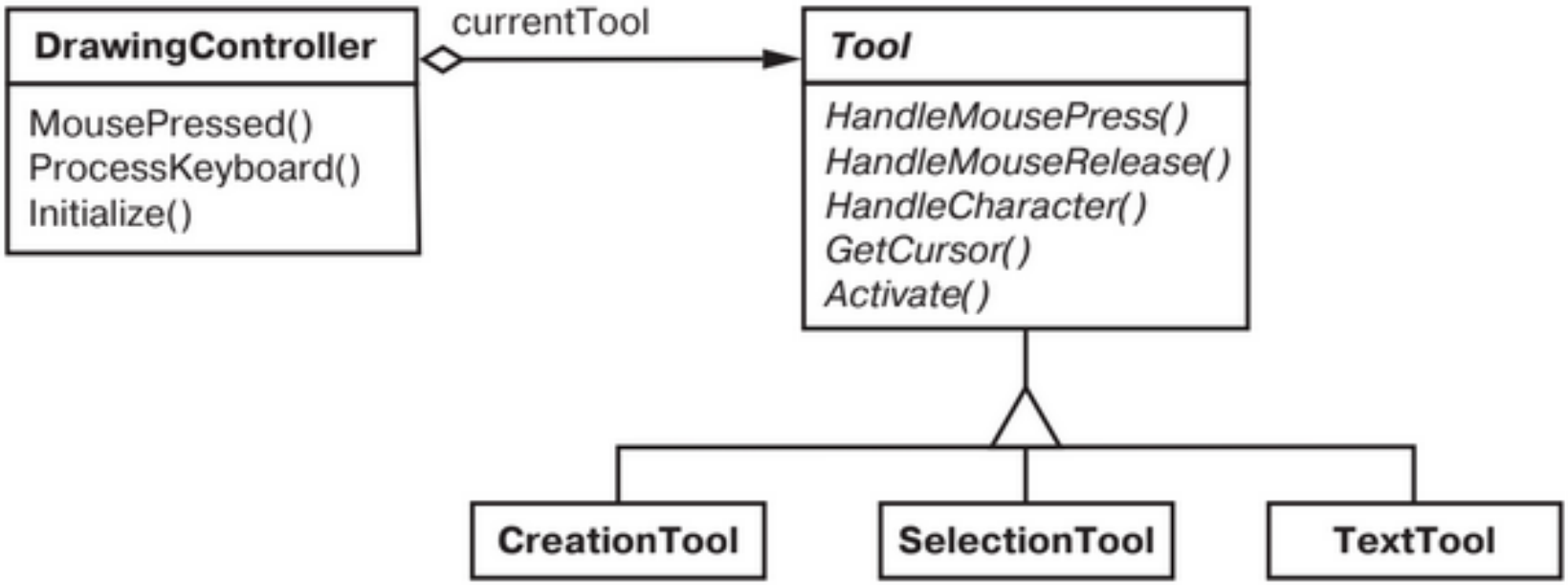
Состояние - решение



Состояние - пример



Состояние - пример



Состояние - результат

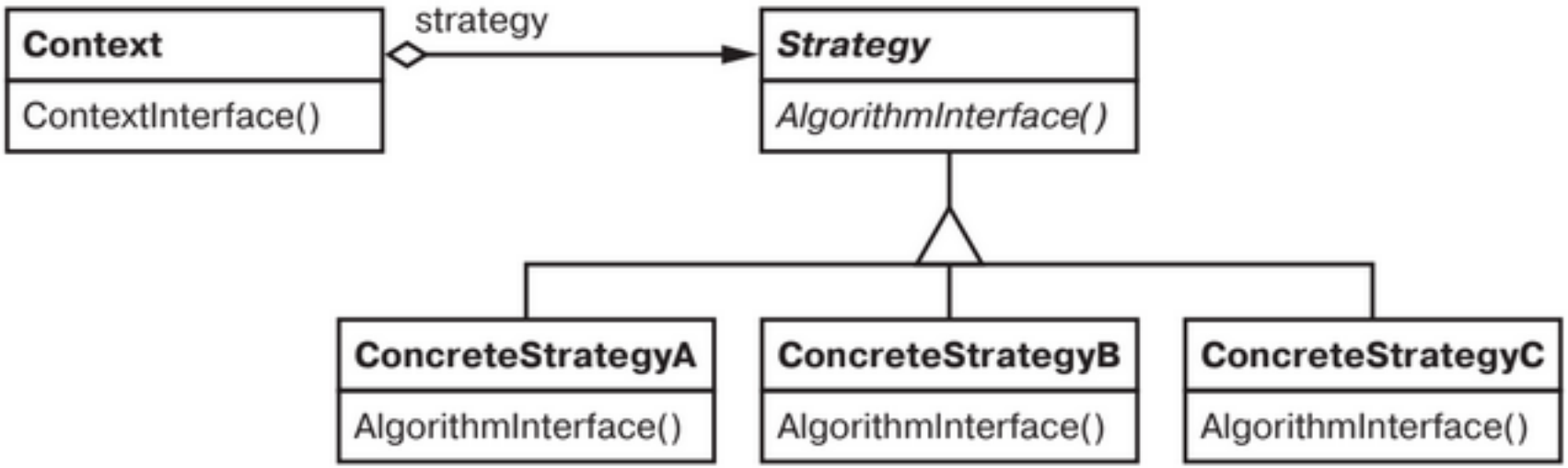
- (+) локализует поведение, зависящее от состояния и делит его на части, соответствующие состояниям (состояние -> объект).
- (+) упрощение кода; легко добавить новые состояния.
- (+) явные переходы между состояниями (переход - изменение экземпляра).
- (+) объекты состояний можно изменить (состояние может быть приспособленцем).
- (-) увеличение количества классов.

Стратегия (Strategy)

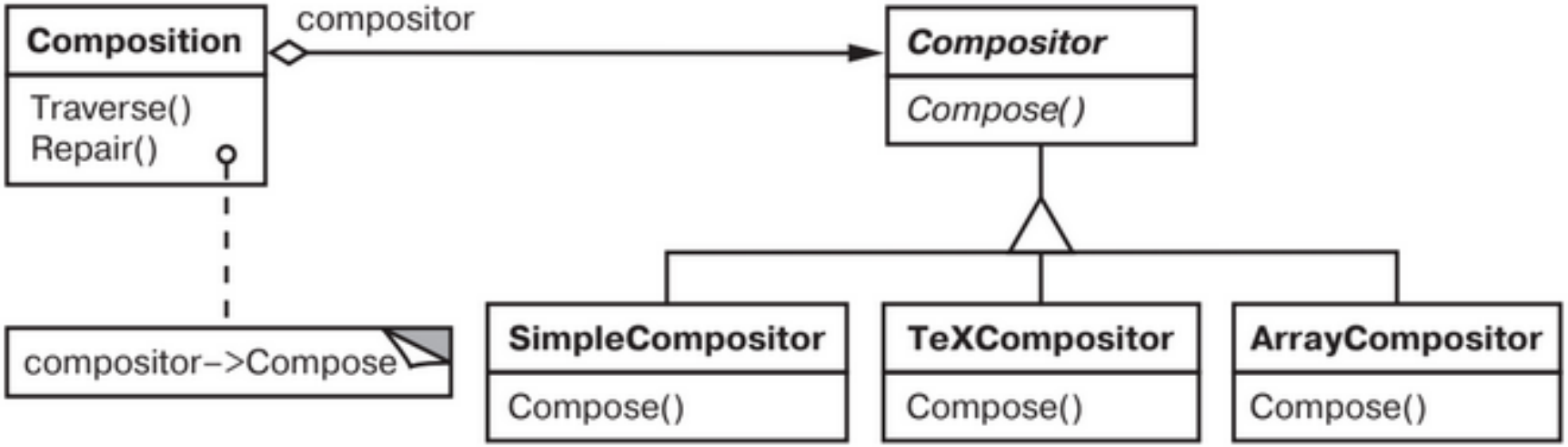
- Определяет семейство алгоритмов, инкапсулирует каждый из них и делает взаимозаменяемыми. Позволяет изменять алгоритмы независимо от клиентов, которые ими пользуются.

- Проблемы:
  - n возможных классов с различным поведением; надо задать 1 из них;
  - нужно n вариантов алгоритмов (может быть иерархия классов);
  - надо скрыть действие алгоритма от клиента;
  - много поведений с условием => любая ветка в отдельный класс.
- Применение:
  - Текст в окне -> разбиение на строки (просто; по количеству символов; с оптимизацией)

Стратегия - решение



Стратегия - пример



Стратегия - результат

- (+) Семейство родственных алгоритмов (иерархия классов общих алгоритмов).
- (+) Альтернатива подклассам контекста.
- (+) Избавление от условных операторов.
- (+) Набор реализаций (выбор).
- (-) Клиент должен знать о стратегиях для их выбора.
- (-) Увеличение количества объектов.
- (-) Избыток параметров от контекста к конкретной стратегии (из-за унификации).

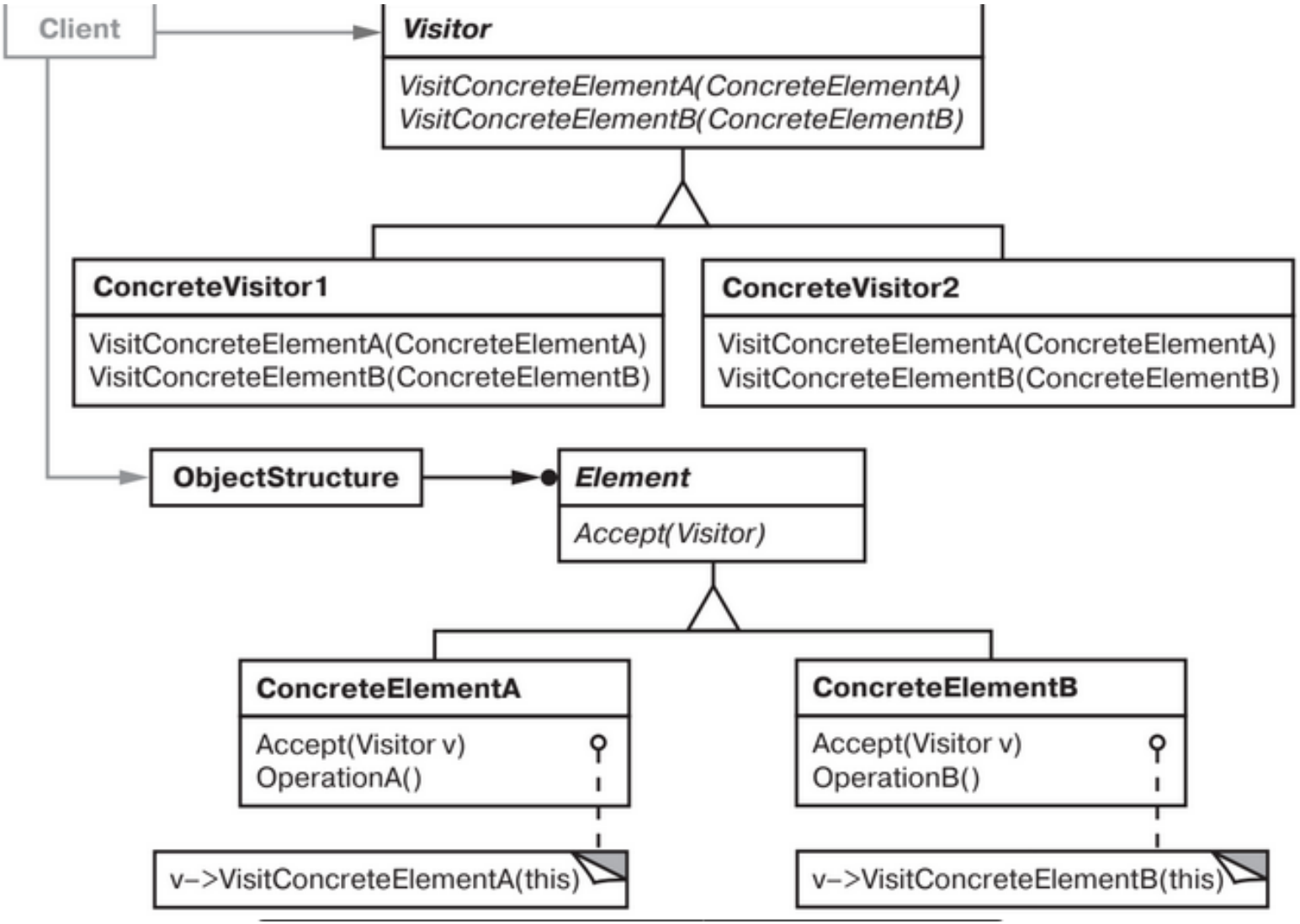
- Пример: проверка введенных данных (окно ввода; функция проверки правильности)
- Сочетание: стратегия-объект в большинстве случаев может быть приспособленцем.

Посетитель (Visitor)

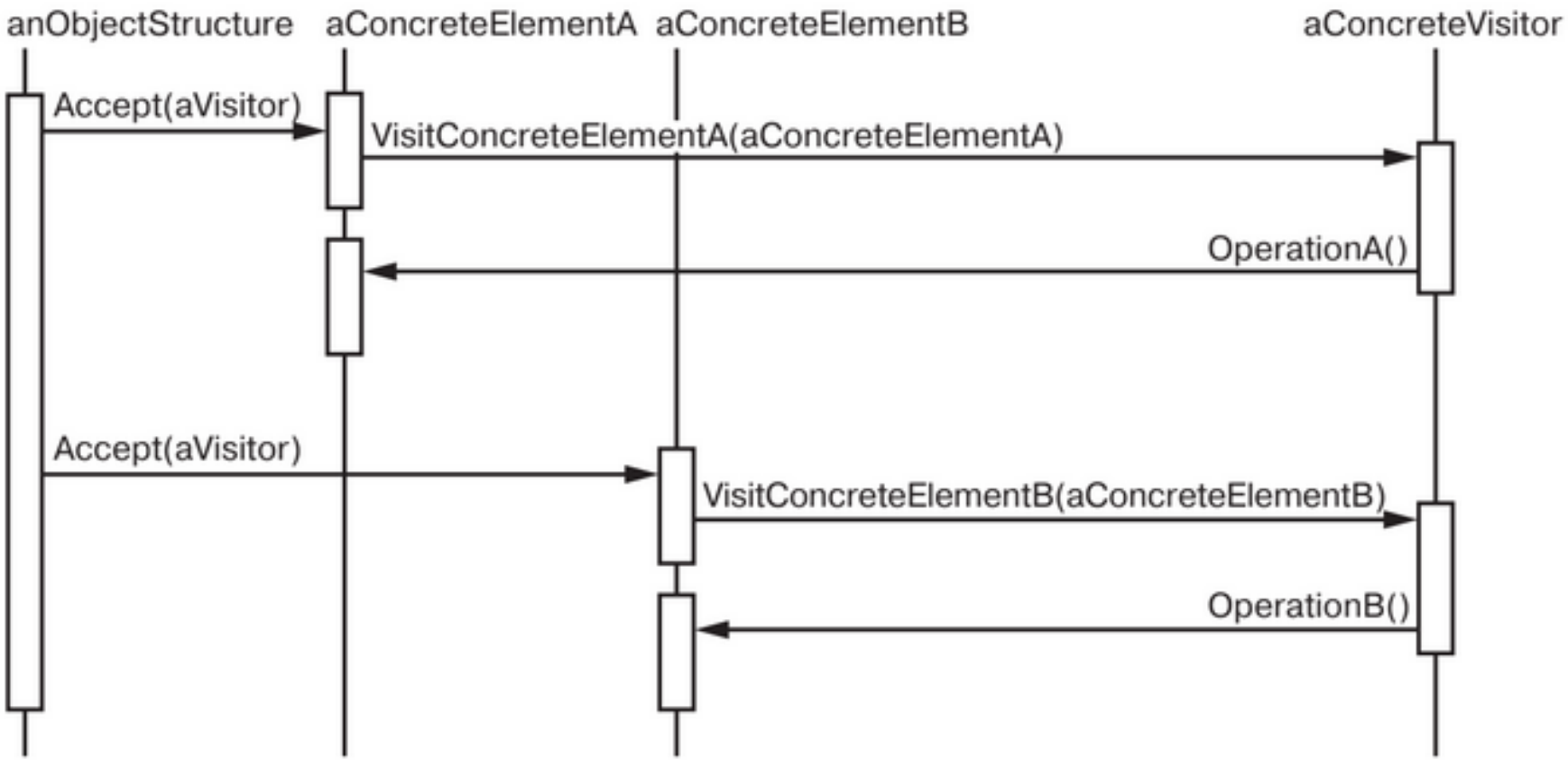
- Описывает операцию, выполняемую с каждым объектом из некоторой структуры. Позволяет определять новую операцию, не изменяя классы этих объектов.

- Проблема:
  - в структуре множество объектов классов с различными интерфейсами и должна быть операция над объектами, зависящими от классов;
  - множество операций над объектами классов:
    - объединение родственных операций в 1 класс,
    - вынос операций из классов объектов (не усложнять),
    - разделение операций между приложениями;
  - классы в структуре объектов изменяются редко, а операции над объектами добавляются часто.
- Применение:
  - 3Д-графика + операции (поиск/изображение/сохранение/…) -> иерархия геометрических фигур;
  - Семантическое дерево: переменные/операции/const + операции(проверка типа; преобразование кода; печать). Цель: разделить.

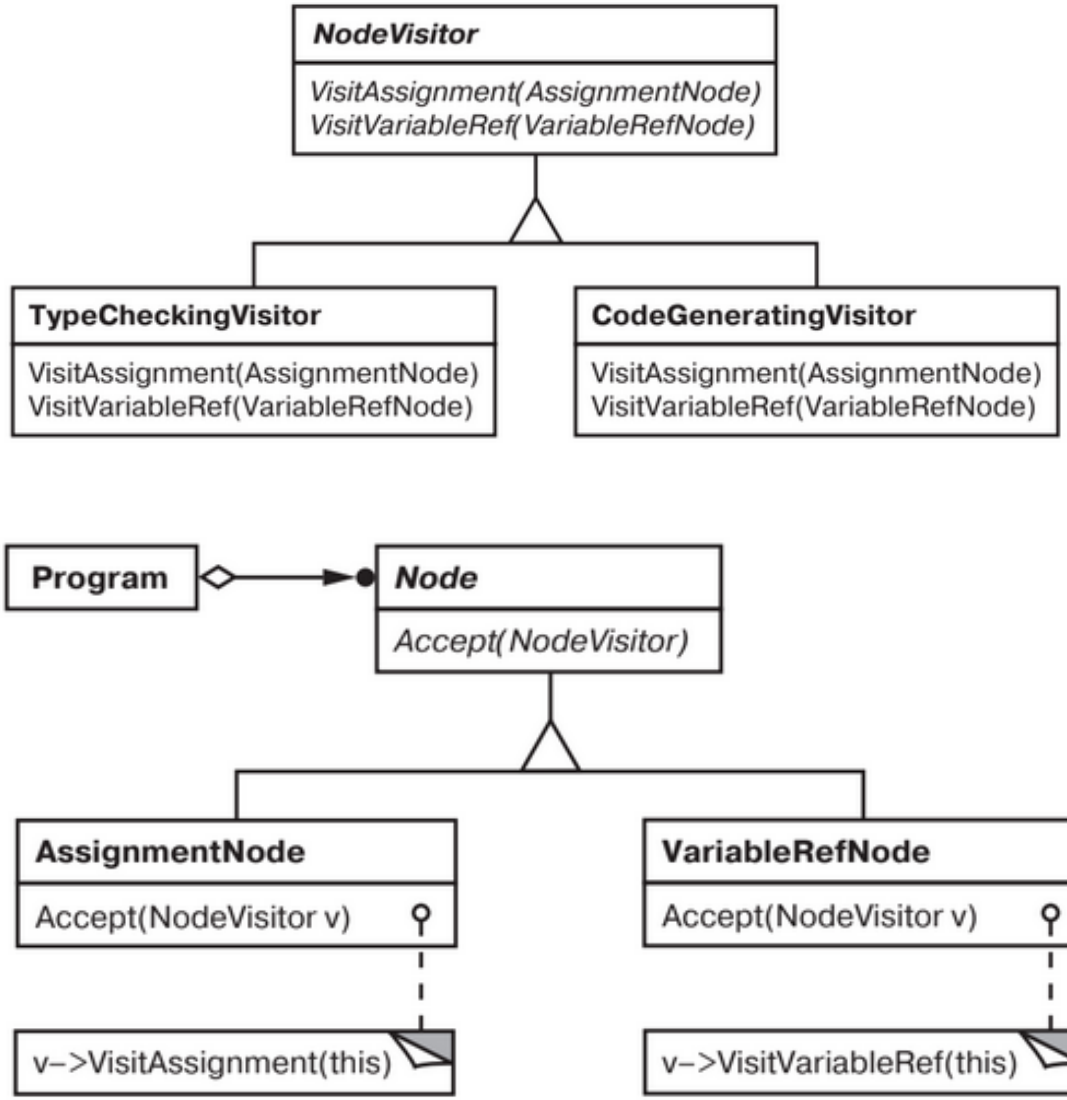
Посетитель - решение



Посетитель - решение



Посетитель - пример



Посетитель - результат

- (+) Упрощение добавления операции (новый посетитель).
- (+) Объединение родственных функций.
- (+) Скрытие данных алгоритма обработки.
- (-) Добавление новых элементов затруднено (необходимо переопределить интерфейс посетителя).
- (-) Нарушение инкапсуляции.

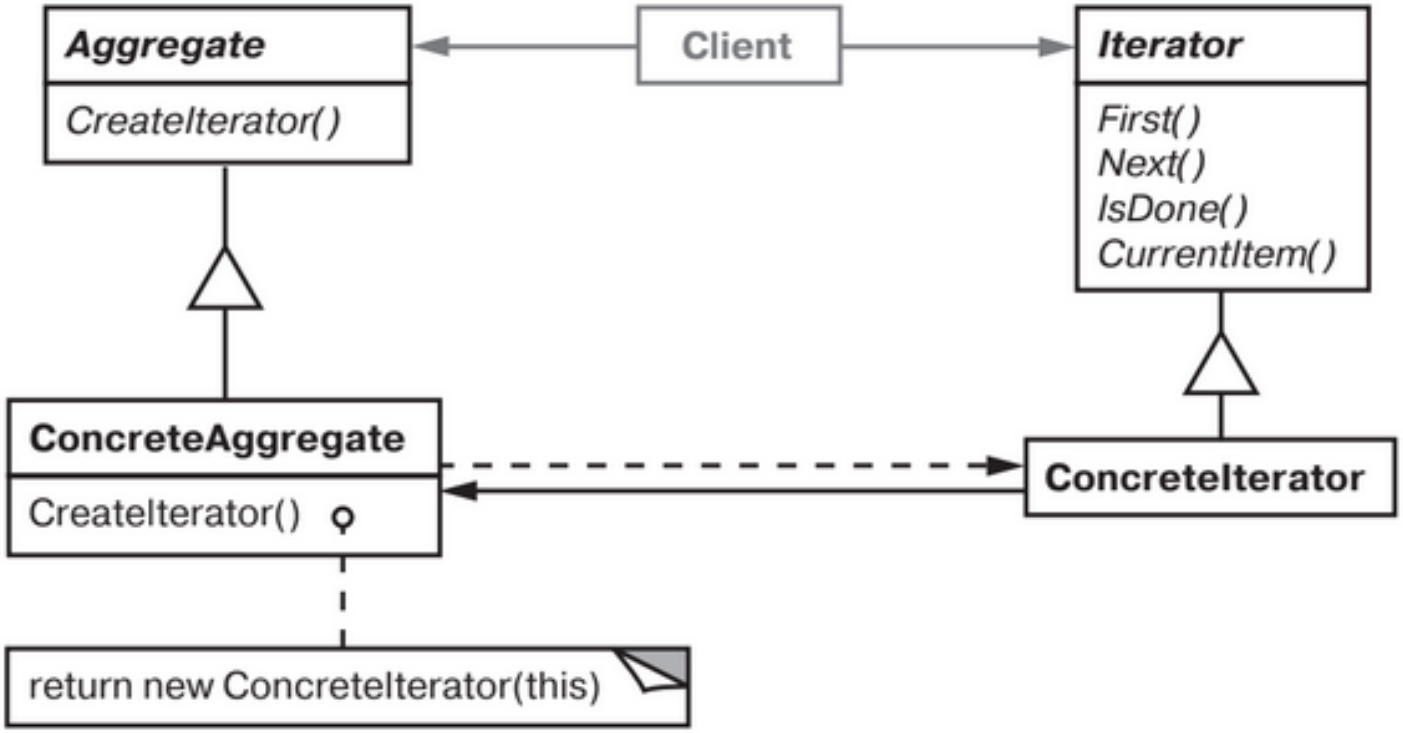
- Итератор - обход объектов с 1 базовым классом.
- Посетитель - обход структуры различных классов.

Итератор (Iterator)

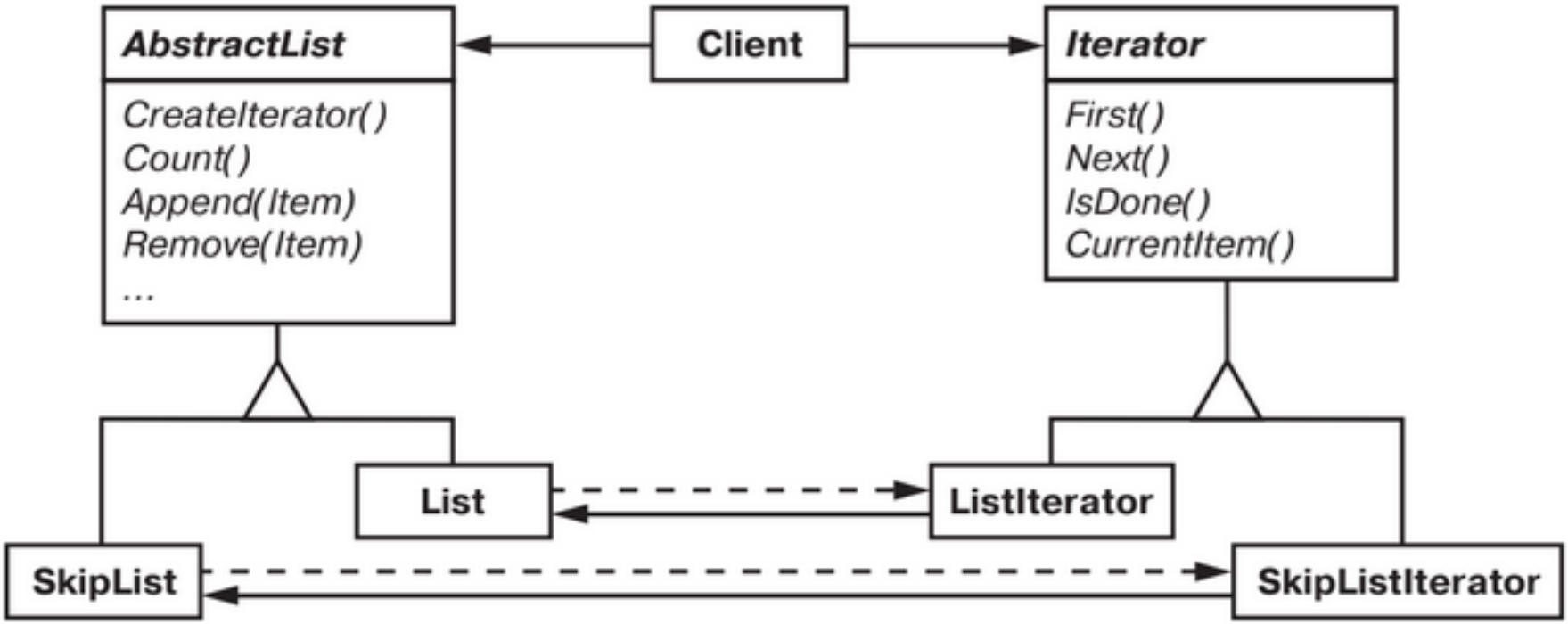
- Предоставляет способ последовательного доступа ко всем элементам составного объекта, не раскрывая его внутреннего представления.

- Проблема:
  - доступ к элементам коллекции без раскрытия ее структуры и внутреннего представления;
  - n активных обходов 1 коллекции;
  - единый интерфейс для обхода различных структур (полиморфная итерация);
- Применение:
  - Список различных элементов + различные алгоритмы обхода; алгоритм обхода знает свой список и зависит от него

Итератор - решение



Итератор - пример



Итератор - результат

- Поддержка различных видов обхода коллекции (новый вид -> новый класс итератора).
- Упрощенный интерфейс коллекции.
- Может быть n одновременных активных обходов для 1 объекта коллекции.

- Итератор обходит компоновщика.
- Через фабричный метод инстанцируют полиморфные итераторы.
- Хранитель хранит состояние итератора и содержится в итераторе.

Интерпретатор (Interpretator)

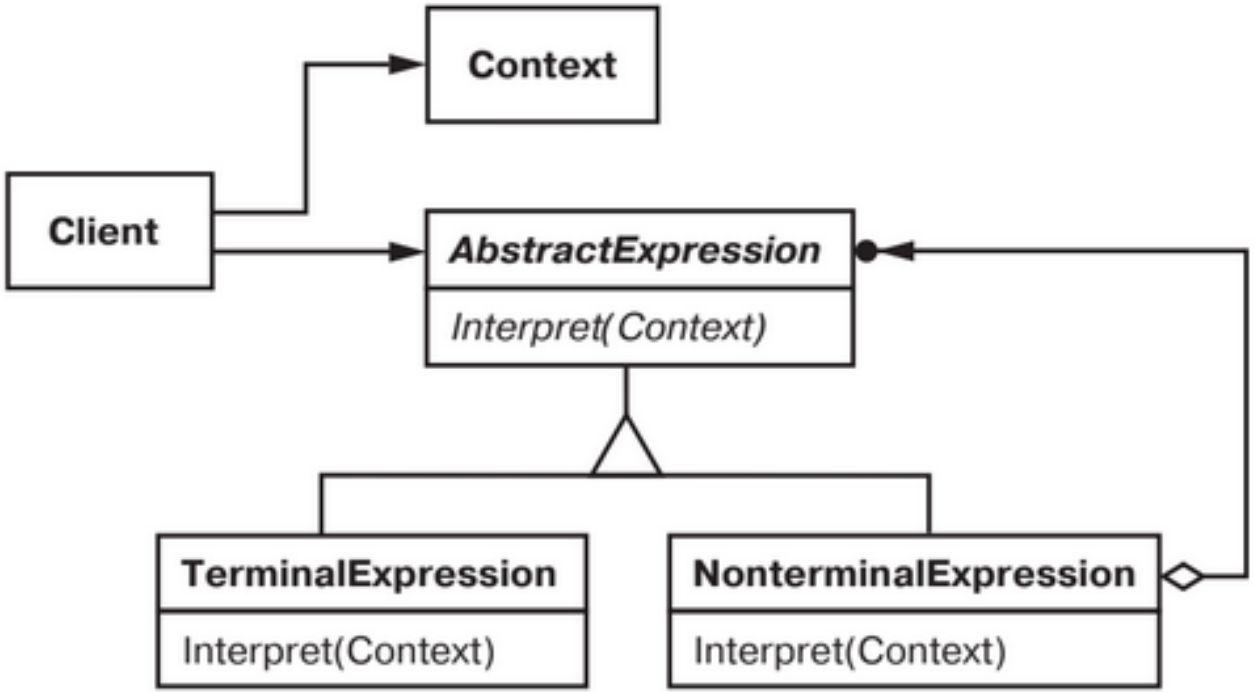
- Для заданного языка определяет представление его грамматики, а также интерпретатор предложений этого языка

- Проблемы:
  - Поиск по обработчику -> регулярное выражение -> грамматика;
  - язык для интерпретации можно представить в виде абстрактных синтаксических деревьев:
    - простая грамматика (сложная грамматика -> генерация синтаксических аналогов), т.к. больше и сложнее иерархия классов,
    - эффективность не является главным критерием (эффективность интерпретации -> преобразование дерева в другую форму, например, конечный автомат).

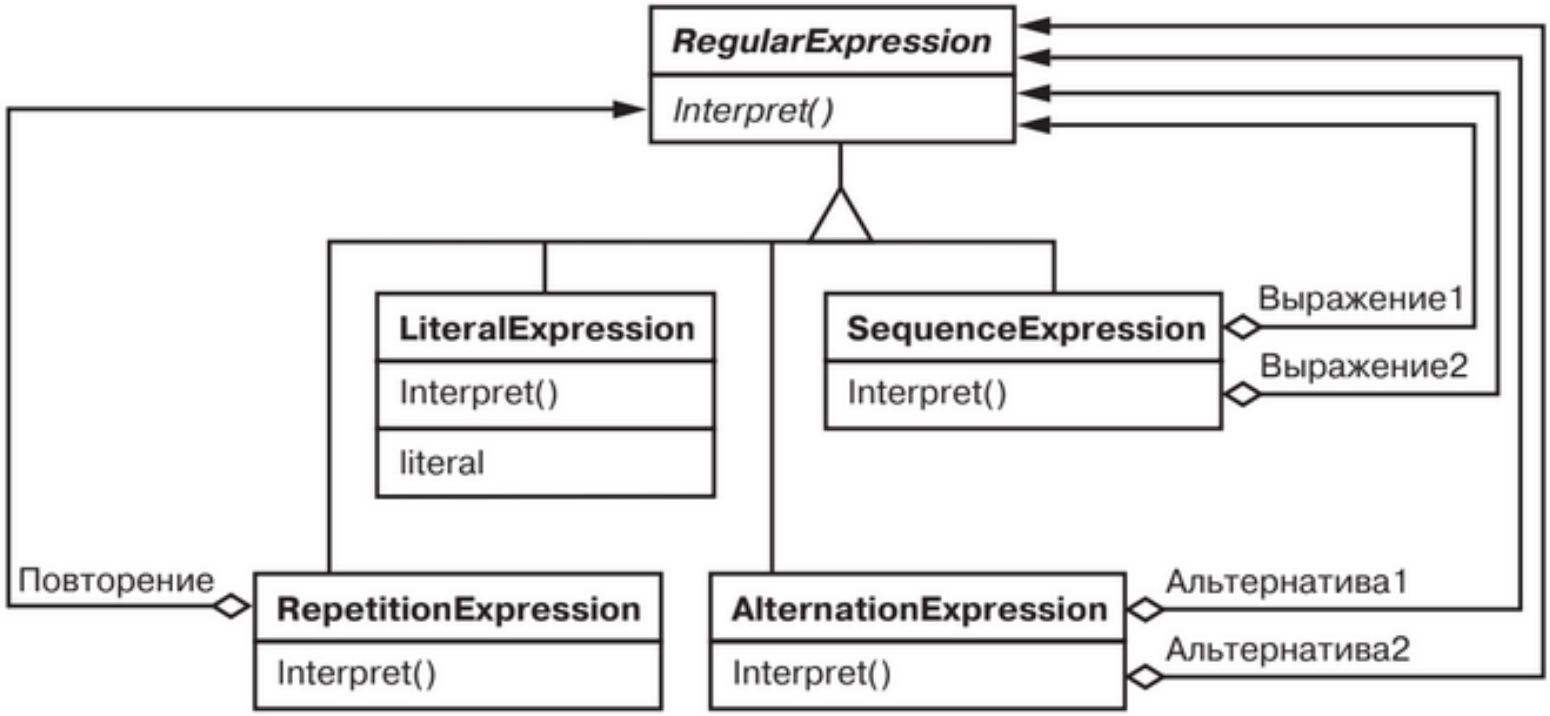
Интерпретатор - пример

```
expression ::= literal | alternation | sequence | repetition |
            '(' expression ')'
alternation ::= expression '|' expression
sequence ::= expression '&' expression
repetition ::= expression '*'
literal ::= 'a' | 'b' | 'c' | ... { 'a' | 'b' | 'c' | ... }*
```

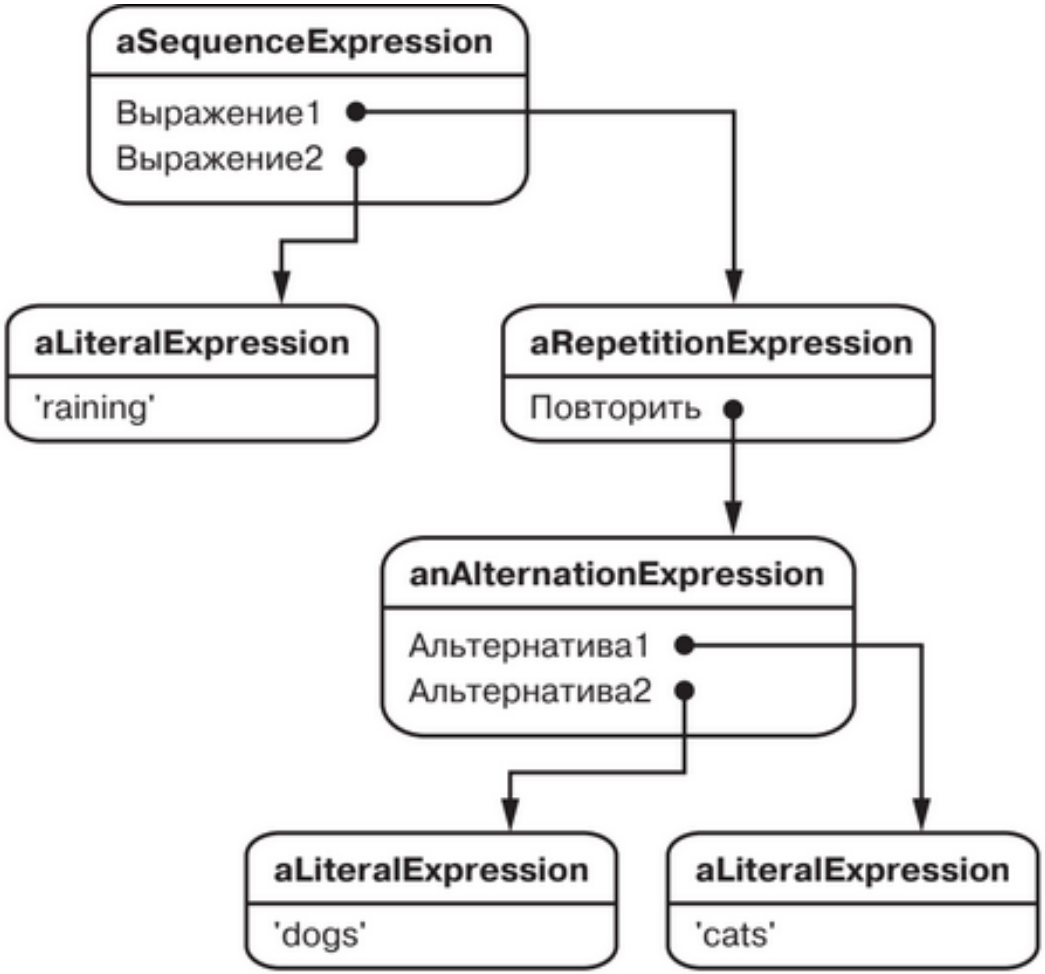
Интерпретатор - решение



Интерпретатор - пример



Интерпретатор - пример



Интерпретатор - результат

- (+) Группировку легко изменять/расширять.
- (+) простая реализация группировки.
- (+) добавление новых способов интерпретации.
- (-) сложные группировки трудно сопровождать.

- Абстрактное дерево может быть компоновщиком.
- Приспособленец для разделения терминальных символов.
- Итератор для обхода структуры.
- Посетитель для инкапсуляции в 1 класс поведения любого узла дерева.