На предыдущей лекции

- **Структурные паттерны**, в которых рассматривается вопрос о том, как из классов и объектов образуются более крупные структуры:
  - **Адаптер (Adapter)** – стыкует интерфейсы различных классов
  - **Мост (Bridge)** – отделяет абстракцию от ее реализации
  - **Компоновщик (Composite)** – представляет сложный объект в виде древовидной структуры
  - **Декоратор (Decorator)** – динамически добавляет объекту новые обязанности
  - **Фасад (Facade)** – одиночный класс, представляющий целую подсистему
  - **Приспособленец (Flyweight)** – разделяемый объект, используемый для моделирования множества мелких объектов.
  - **Заместитель (Proxy)** – объект, представляющий другой объект.

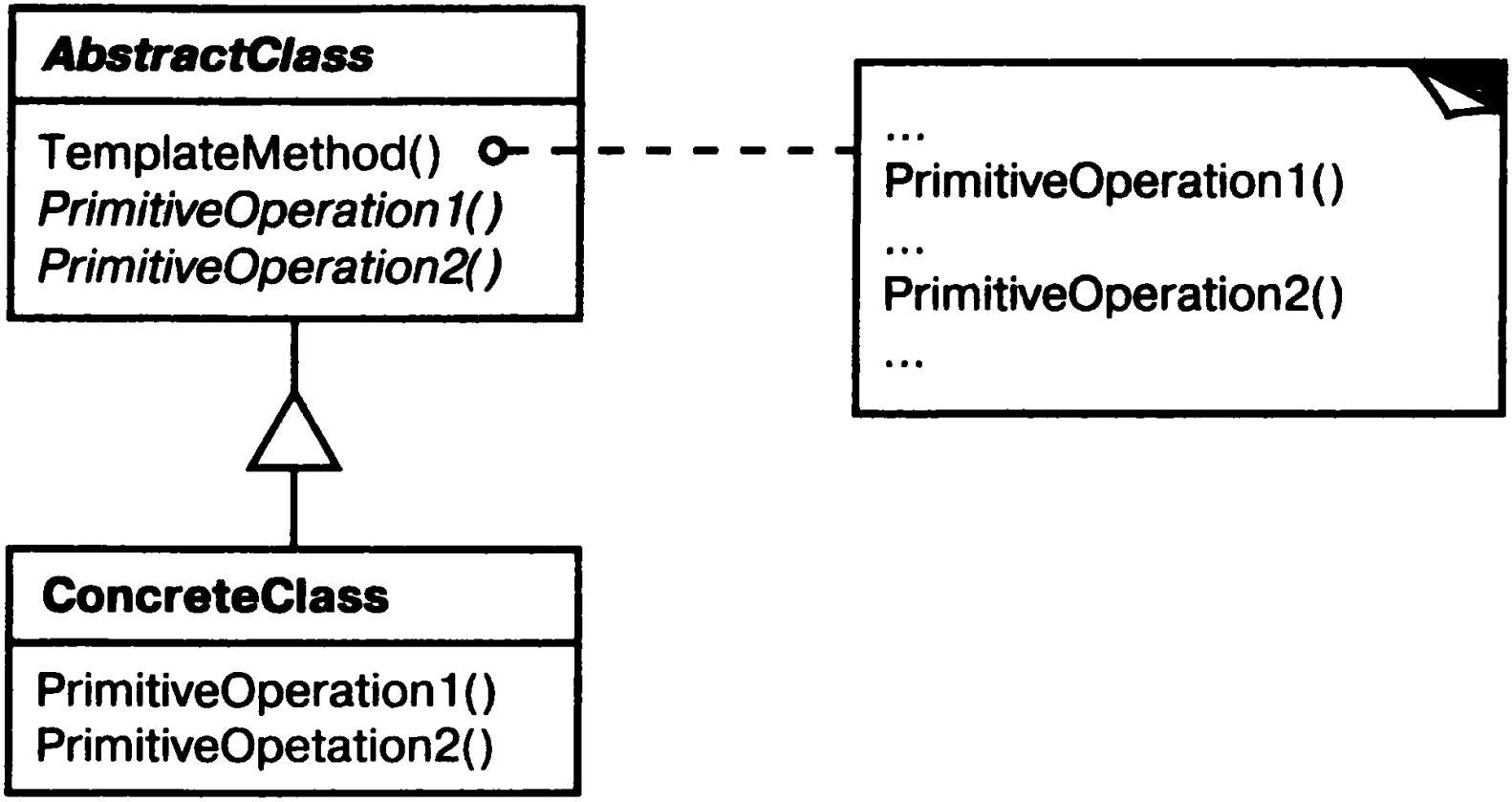
Паттерны поведения

- Паттерны, связанные с алгоритмами и распределением обязанностей между объектами.
- Действуют на уровне:
  - **Классов** - используют наследование чтобы распределить поведение между классами.
  - **Объектов** – используют композицию объектов для организации их совместной работы
- Паттерны:
  - **Шаблонный метод (Template Method)** – пошаговое определение алгоритма в подклассах;
  - **Итератор (Iterator)** – абстрагирует перебор объектов в контейнере;
  - **Наблюдатель (Observer)** – управляет зависимостями между объектами через события.
  - **Цепочка обязанностей (Chain of Responsibility)** – уменьшает степень связанности классов, посылая запросы не напрямую, а по цепочке кандидатов;
  - **Посредник (Mediator)** – уменьшает связанность классов через косвенные ссылки;
  - **Команда (Command)** – инкапсулирует запрос в виде объекта, который можно передавать, хранить и т.п.;
  - **Состояние (State)** - инкапсулирует состояние объекта так, что его изменение меняет поведение;
  - **Стратегия (Strategy)** – инкапсулирует алгоритм объекта, обеспечивая его хранение и замену;
  - **Хранитель (Memento)** – выносит во внешний объект внутренне состояние объекта;
  - **Посетитель (Visitor)** – инкапсулирует поведение, которое иначе пришлось бы распределять между классами;
  - **Интерпретатор (Interpreter)** – реализует грамматику языка в виде иерархии классов и реализует интерпретатор как последовательность операций над этими классами;

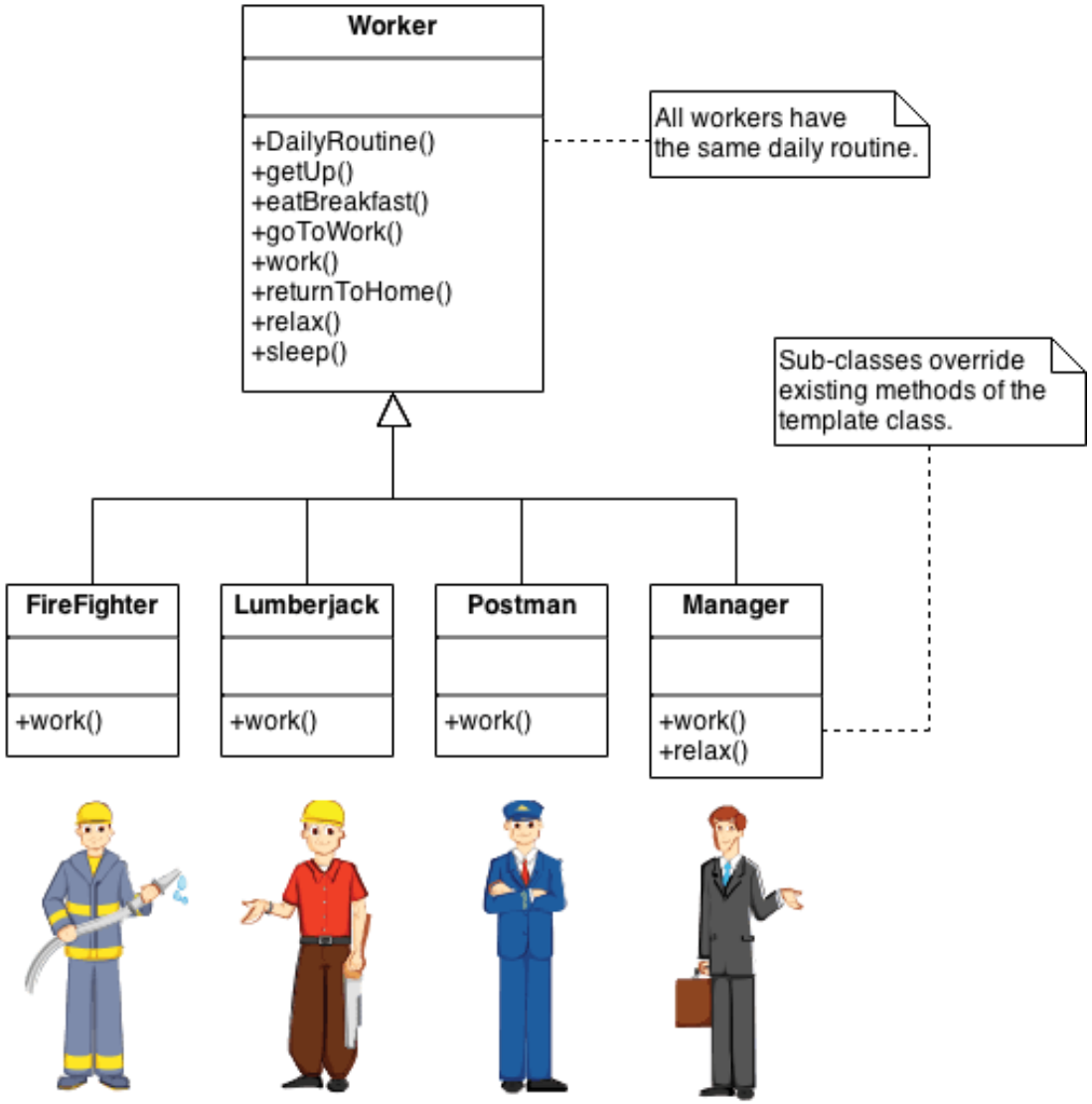
Шаблонный метод

- Назначение:
  - Определяет основу алгоритма и позволяет подклассам переопределить некоторые шаги алгоритма, не изменяя структуру в целом.
- Применимость:
  - Чтобы однократно использовать инвариантные части алгоритма, оставляя реализацию изменяющегося поведения на усмотрение подклассов;
  - Когда нужно вычленить и локализовать в одном классе поведение, общее для всех подклассов;
  - Для управления расширениями подклассов – можно определить шаблонный метод так, чтобы он вызывал операции-зацепки (hooks) в определенных точках, расширяя поведение именно в этих точках.

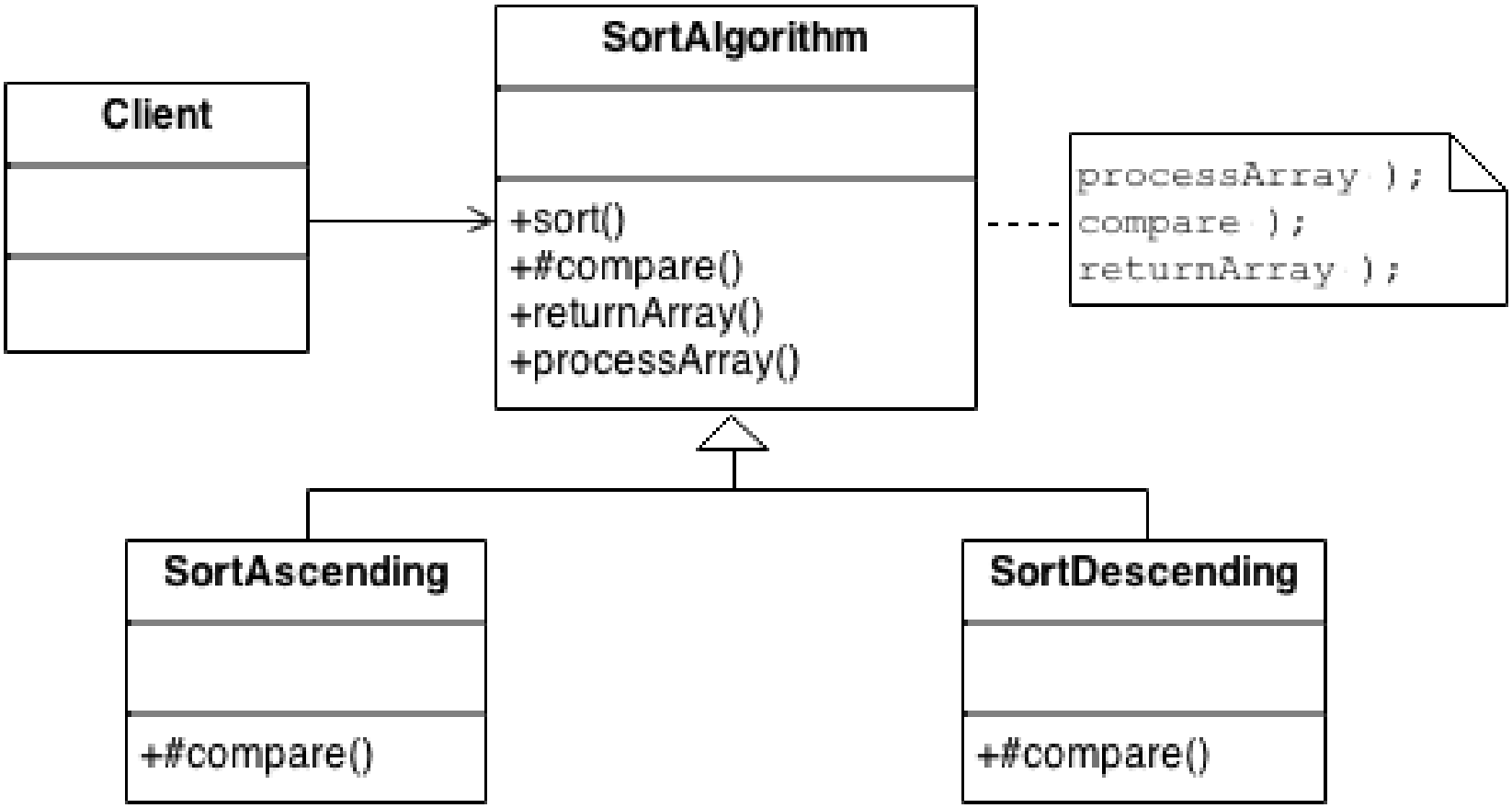
Шаблонный метод – структура



Шаблонный метод



Шаблонный метод – пример



Шаблонный метод – пример

```
class Base {
    void a() { cout << "a  "; }
    void c() { cout << "c  "; }
    void e() { cout << "e  "; }
    virtual void ph1() = 0; // точки вариативности алгоритма
    virtual void ph2() = 0;
  public:
    void execute() { //шаблонный метод
        a();
        ph1();
        c();
        ph2();
        e();
    }
};
```

Шаблонный метод – пример

```
class One: public Base { // уточнение точек вариативности
    void ph1() { cout << "b  "; }
    void ph2() { cout << "d  "; }
};
class Two: public Base {
   void ph1() { cout << "2  "; }
   void ph2() { cout << "4  "; }
};
int main() {
  Base *array[] =  { new One(), new Two() };
  for (int i = 0; i < 2; i++) {
    array[i]->execute();
    cout << '\n';
  }
}
```

```
a  b  c  d  e
a  2  c  4  e
```

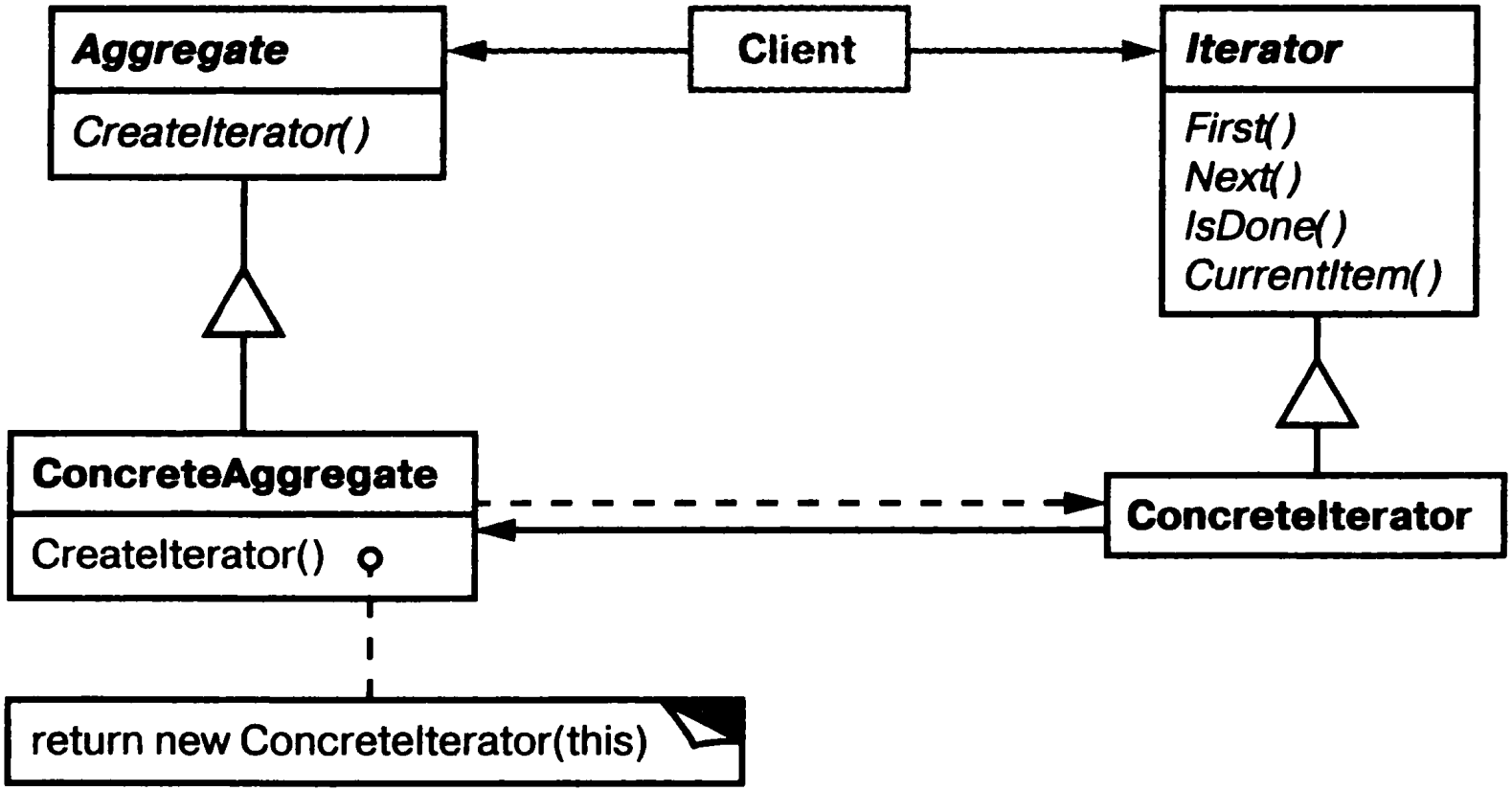
Шаблонный метод – примечания

- Инвертированная структура вызовов (т.н. «**Принцип Голливуда**» - «не звоните нам, мы сами Вам позвоним») – родительский класс вызывает методы подкласса, а не наоборот.
- Часть методов базового класса **можно** переопределить – если это операции-зацепки (hooks).
- Часть методов базового класса **необходимо** переопределить – если это абстрактные методы.
- Переопределяемые методы должны быть **виртуальными** для корректной полиморфной работы.
- Переопределяемые методы рекомендуется определять как **protected**, чтобы иметь возможность их переопределения потомками и, в то же время, избежать их использования напрямую.

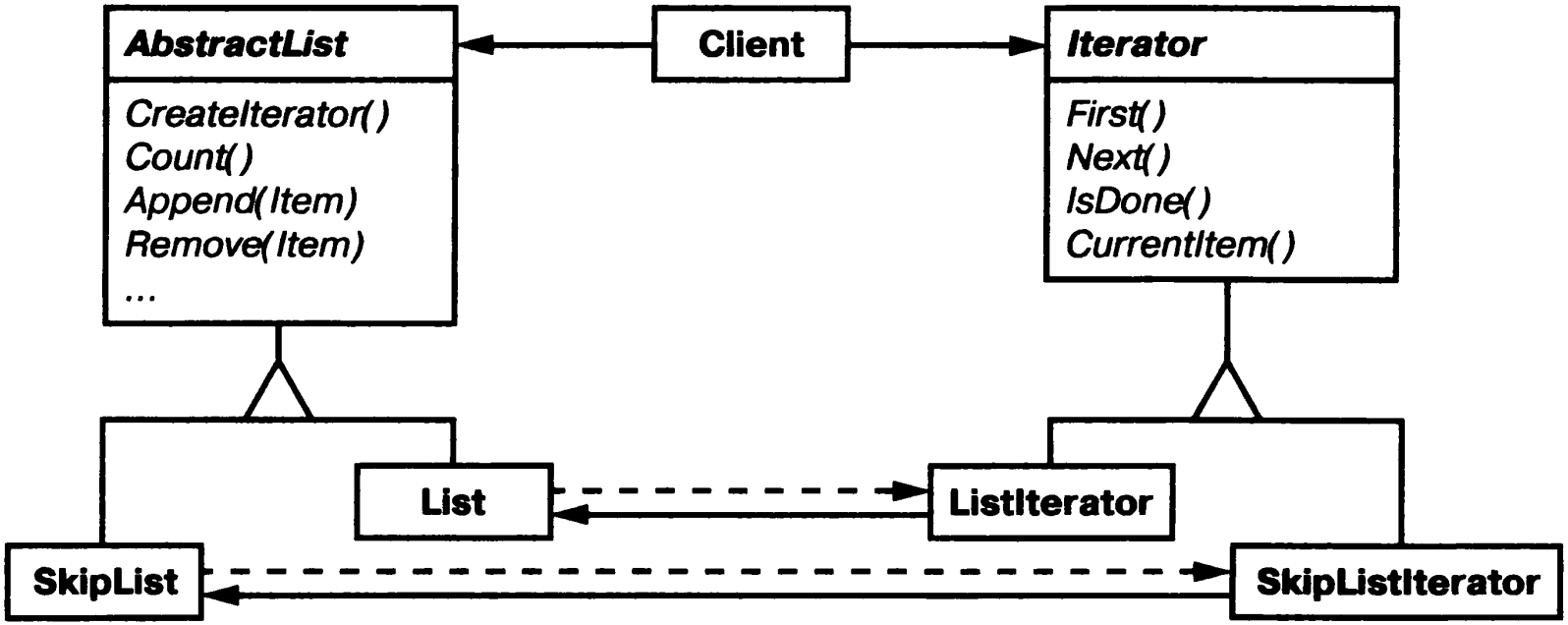
Итератор

- Назначение:
  - Предоставляет способ последовательного доступа ко всем элементам составного (контейнерного) объекта, не раскрывая его внутреннего представления.
- Применимость:
  - Для доступа к содержимому агрегированных объектов без раскрытия их внутреннего представления;
  - Для поддержки нескольких активных обходов одного и того же агрегированного объекта;
  - Для предоставления единообразного представления интерфейса для обхода различных структур данных (полиморфной итерации)

Итератор – структура



Итератор – пример



Итератор – пример

```
class Stack
{
    int items[10];
    int sp;
  public:
    friend class StackIter;
    Stack() { sp =  - 1; }
    void push(int in) { items[++sp] = in; }
    int pop() { return items[sp--]; }
    bool isEmpty() { return (sp ==  - 1); }
    StackIter *createIterator() {
return new StackIter(this);
    };
};
```

```
class StackIter
{
    const Stack *stk;
    int index;
  public:
    StackIter(const Stack *s) { stk = s; }
    void first() { index = 0; }
    void next() { index++; }
    bool isDone() { return index == stk->sp + 1; }
    int currentItem() {return stk->items[index]; }
};
```

Итератор – пример

```
bool operator == (Stack &l, Stack &r)
{
  StackIter *itl = l.createIterator();
  StackIter *itr = r.createIterator();
  for (itl->first(), itr->first(); !itl->isDone(); itl->next(), itr->next())
      if (itl->currentItem() != itr->currentItem()) break;
  bool ans = itl->isDone() && itr->isDone();
  delete itl;
  delete itr;
  return ans;
}
```

Итератор – пример

```
int main()
{
  Stack s1;
  for (int i = 1; i < 5; i++) s1.push(i);
  Stack s2(s1), s3(s1), s4(s1), s5(s1);
  s3.pop();
  s5.pop();
  s4.push(2);
  s5.push(9);
  cout << "1 == 2 is " << (s1 == s2) << endl;
  cout << "1 == 3 is " << (s1 == s3) << endl;
  cout << "1 == 4 is " << (s1 == s4) << endl;
  cout << "1 == 5 is " << (s1 == s5) << endl;
}
```

```
1 == 2 is 1
1 == 3 is 0
1 == 4 is 0
1 == 5 is 0
```

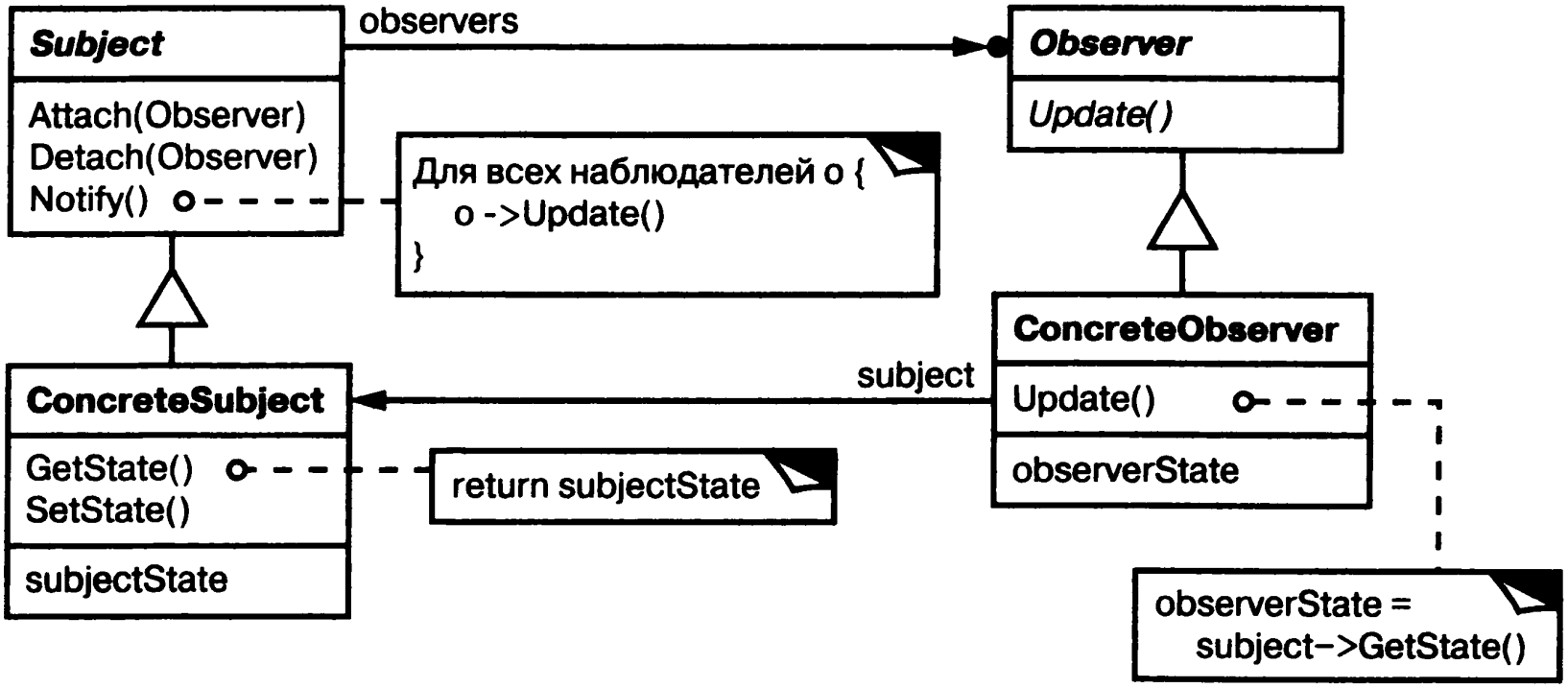
Итератор – примечания

- Итератор инкапсулирует алгоритм обхода и позволяет легко его изменять;
- Итератор облегчает интерфейс агрегата;
- На одном агрегате одновременно может работать несколько итераторов, т.к. состояние обхода инкапсулировано не в агрегате, а в итераторе;
- Устойчивость итератора – способность корректной работы при изменении агрегата (очень дорого, обычно от этого отказываются).

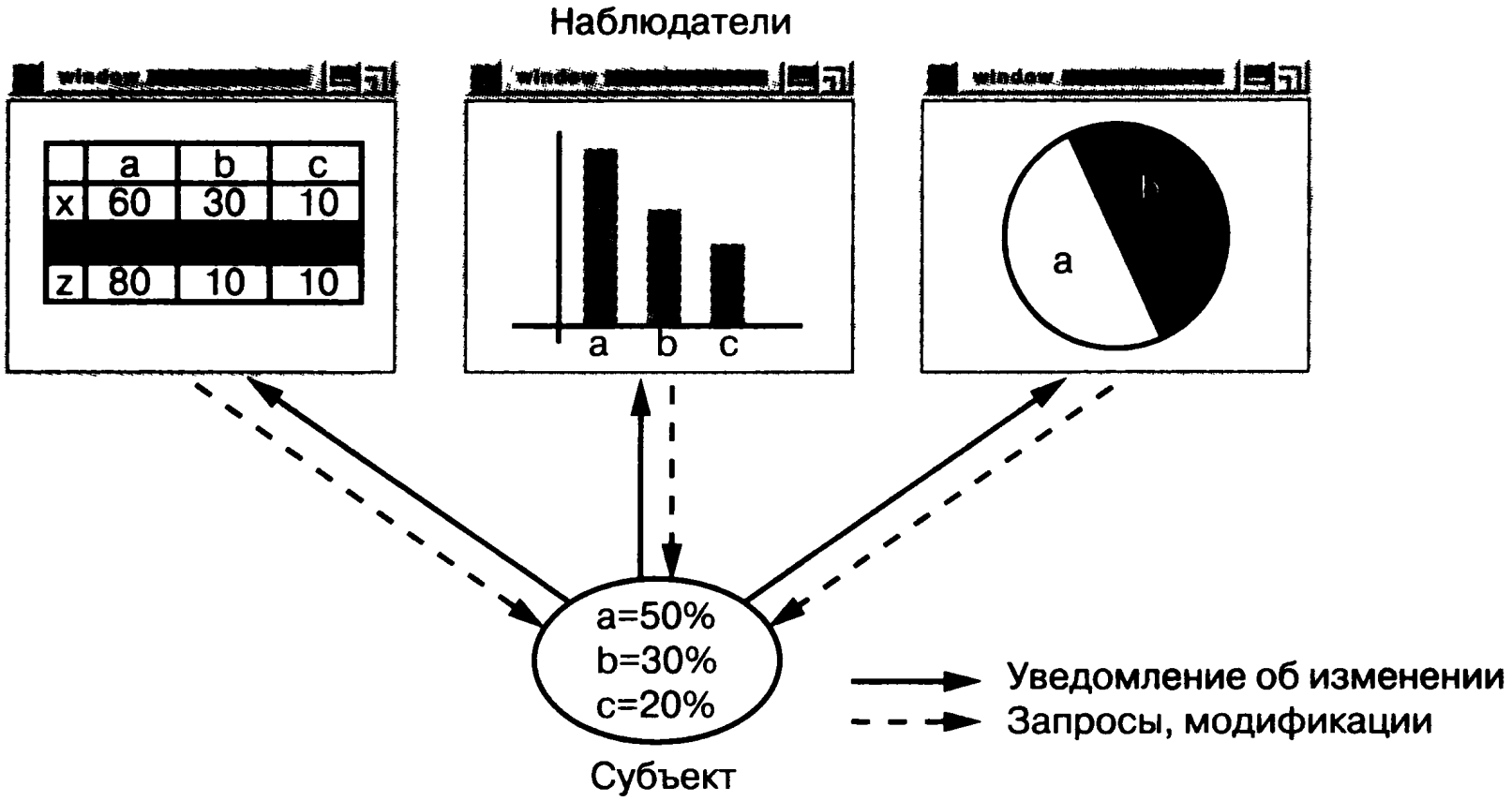
Наблюдатель

- Назначение:
  - Определяет зависимость 1:M между объектами таким образом, что при изменении состояния одного объекта все зависящие от него объекты оповещаются об этом и автоматически обновляются (реагируют).
- Применимость:
  - Если у абстракции есть несколько аспектов, зависящих друг от друга, инкапсуляция этих аспектов в разные объекты позволяет изменять и повторно использовать их независимо;
  - Когда при модификации одного объекта необходимо оповестить заранее неизвестное множество других объектов.
  - Когда уведомляющий объект ничего не знает об уведомляемых объектах.

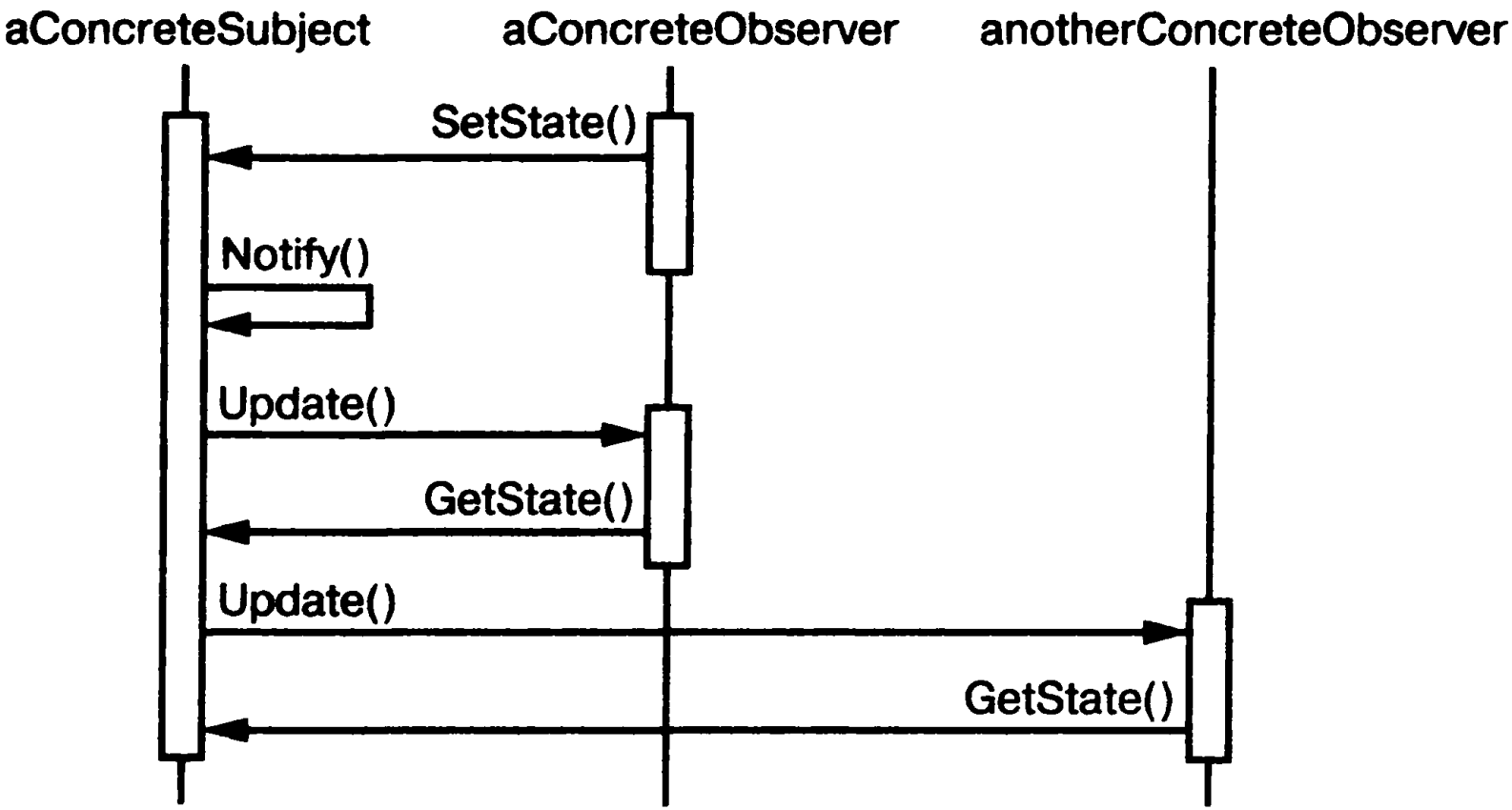
Наблюдатель – структура



Наблюдатель – пример



Наблюдатель – пример



Наблюдатель – пример

```
class Subject {
   vector < class Observer * > views;
   int value;
 public:
   void attach(Observer *obs) {
      views.push_back(obs);
   }
   void setVal(int val) {
      value = val;
      notify();
   }
   int getVal() { return value; }
   void notify() {
      for (int i = 0; i < views.size(); i++)
         views[i]->update();
   };
};
```

```
class Observer {
   Subject *model;
   int denom;
 public:
   Observer(Subject *mod, int div) {
      model = mod;
      denom = div;
      model->attach(this);
   }
   virtual void update() = 0;
 protected:
   Subject *getSubject() { return model; }
   int getDivisor() { return denom; }
};
```

Наблюдатель – пример

```
class DivObserver: public Observer {
  public:
    DivObserver(Subject *mod, int div): Observer(mod, div){}
    void update() {
        int v = getSubject()->getVal(), d = getDivisor();
        cout << v << " div " << d << " is " << v / d << '\n';
    }
};
class ModObserver: public Observer {
  public:
    ModObserver(Subject *mod, int div): Observer(mod, div){}
    void update() {
        int v = getSubject()->getVal(), d = getDivisor();
        cout << v << " mod " << d << " is " << v % d << '\n';
    }
};
```

Наблюдатель – пример

```
int main() {
  Subject subj;
  DivObserver divObs1(&subj, 4);  //создадим набор наблюдателей
  DivObserver divObs2(&subj, 3);
  ModObserver modObs3(&subj, 3);
  subj.setVal(14); //изменим наблюдаемый объект
}
```

```
14 div 4 is 3
14 div 3 is 4
14 mod 3 is 2
```

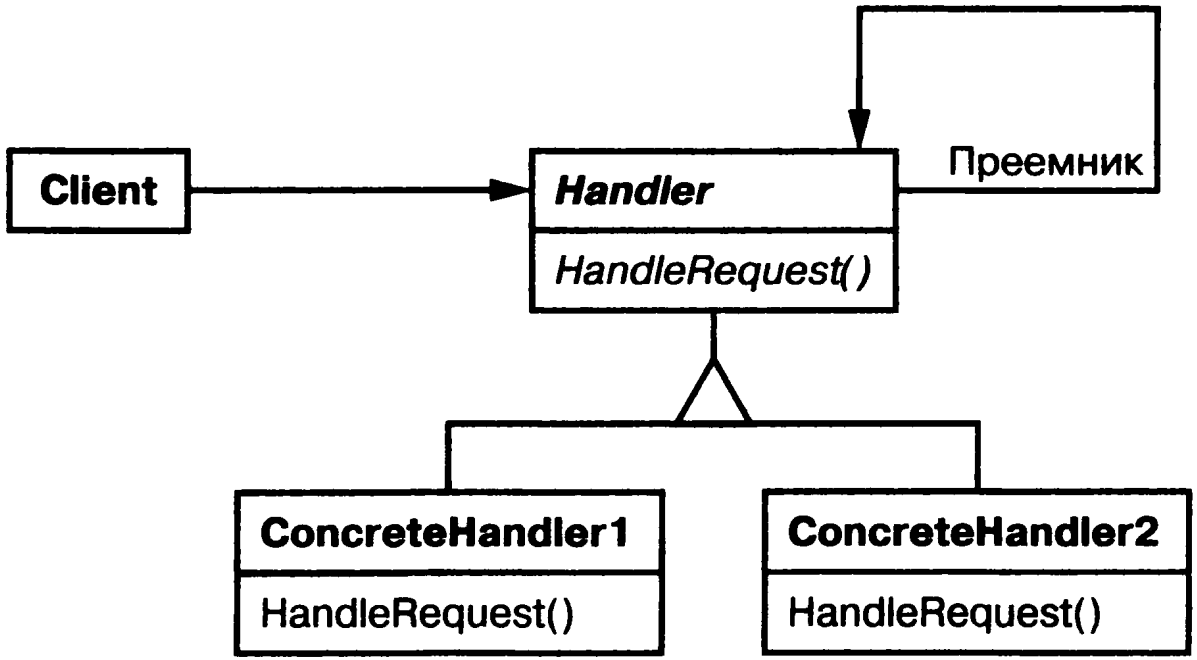
Наблюдатель – примечания

- Абстрактная связанность субъекта и наблюдателя – субъекту неизвестны конкретные классы наблюдателей;
- Поддержка широковещательной коммуникации – субъект может уведомлять любое количество наблюдателей;
- Неожиданные обновления – т.к. заранее неизвестно количество реагирующих объектов, и характер этой реакции, возможны сюрпризы.
- Возможность передачи информации (например, через EventArgs) непосредственно при нотификации (push-модель), вместо получения состояния отдельным вызовом (pull-модель)
- Необходимость отписки от наблюдения перед удалением наблюдателя.

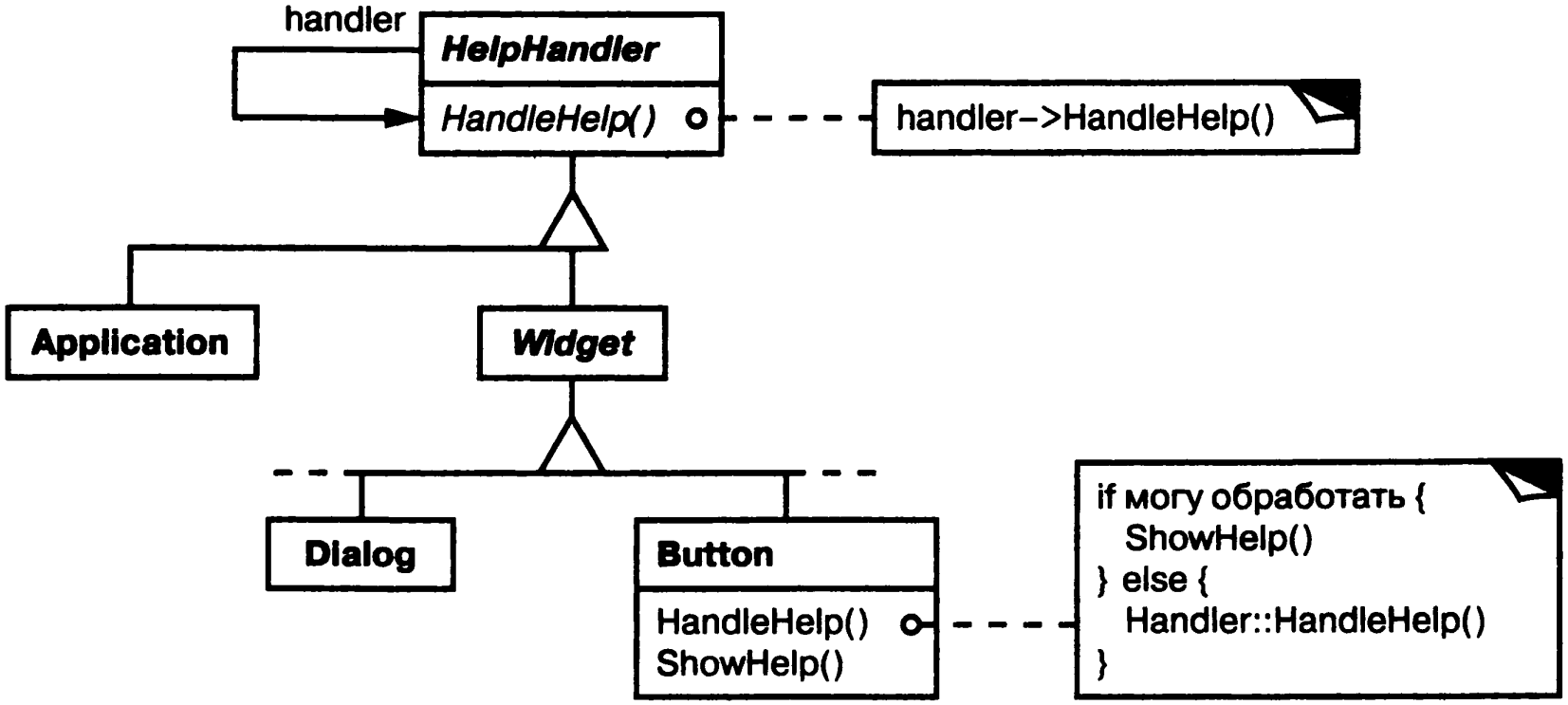
Цепочка обязанностей

- Назначение:
  - 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
- Применимость:
  - Есть более одного объекта, способного обработать запрос, причем нужный обработчик заранее неизвестен и должен быть найден в процессе обработки
  - Необходимо отправить запрос одному из нескольких объектов, не указывая, кому именно
  - Набор объектов-обработчиков должен задаваться динамически

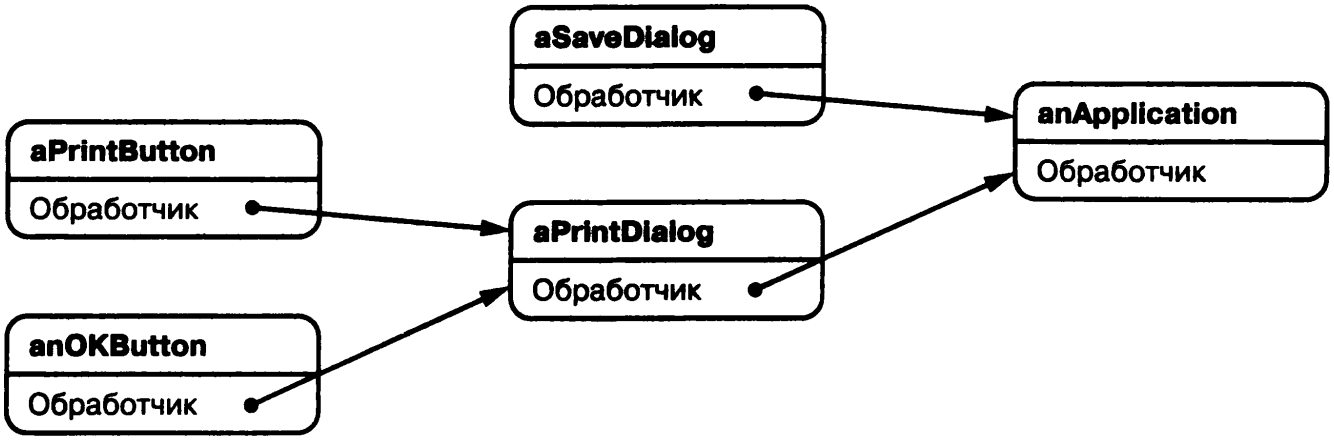
Цепочка обязанностей – структура

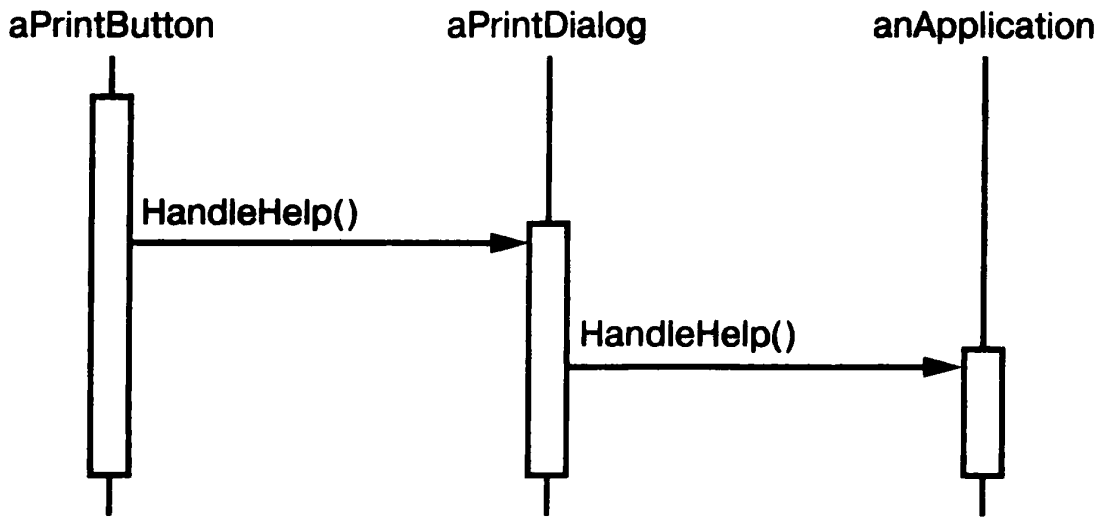


Цепочка обязанностей – пример



Цепочка обязанностей – пример





Цепочка обязанностей – пример

```
class Approver //Handler
{
   Approver *next; // указатель на следующего в цепочке
  public:
   Approver() { next = 0; }
   void setNext(Approver *n) { next = n; }
   void add(Approver *n) {
      if (next)
        next->add(n);
      else
        next = n;
   }
   virtual void approve(int i) { next->approve(i); } //Базовый класс всегда делегирует
};
```

Цепочка обязанностей – пример

```
class Manager: public Approver
{
  public:
    void approve(int i)
    {
        if (i > 10000) {
            cout << "Менеджер не  может утвердить сумму " << i << ", передает выше.  ";
            Approver::approve(i); // 3. Delegate to the base class
        }
        else
          cout << "Менеджер утвердил сумму " << i << endl;
    }
};
```

Цепочка обязанностей – пример

```
class Director: public Approver {
  public:
    void approve(int i)
    {
        if (i > 100000) {
            cout << "Директор не  может утвердить сумму " << i << ", передает выше.  ";
            Approver::approve(i);
        }
        else
          cout << "Директор утвердил сумму " << i << endl;
    }
};
```

Цепочка обязанностей – пример

```
class President: public Approver
{
  public:
    void approve(int i)
    {
        if (i > 1000000) {
            cout << "Президент отказал в выделении " << i << ", это уже чересчур.  ";
        }
        else
          cout << "Президент утвердил сумму " << i << endl;
    }
};
```

Цепочка обязанностей – пример

```
int main()
{
  Manager boss;
  Director bigboss;
  President biggestboss;
  boss.add(&bigboss);
  boss.add(&biggestboss);
  boss.approve(500);
  boss.approve(5000);
  boss.approve(50000);
  boss.approve(500000);
  boss.approve(5000000);
}
```

- Менеджер утвердил сумму 500
- Менеджер утвердил сумму 5000
- Менеджер не может утвердить сумму 50 000, передает выше. Директор утвердил сумму 50 000
- Менеджер не может утвердить сумму 500 000, передает выше. Директор не может утвердить сумму 500 000, передает выше. Президент утвердил сумму 500 000
- Менеджер не может утвердить сумму 5 000 000, передает выше. Директор не может утвердить сумму 5 000 000, передает выше. Президент отказал в выделении 5 000 000, это уже чересчур.

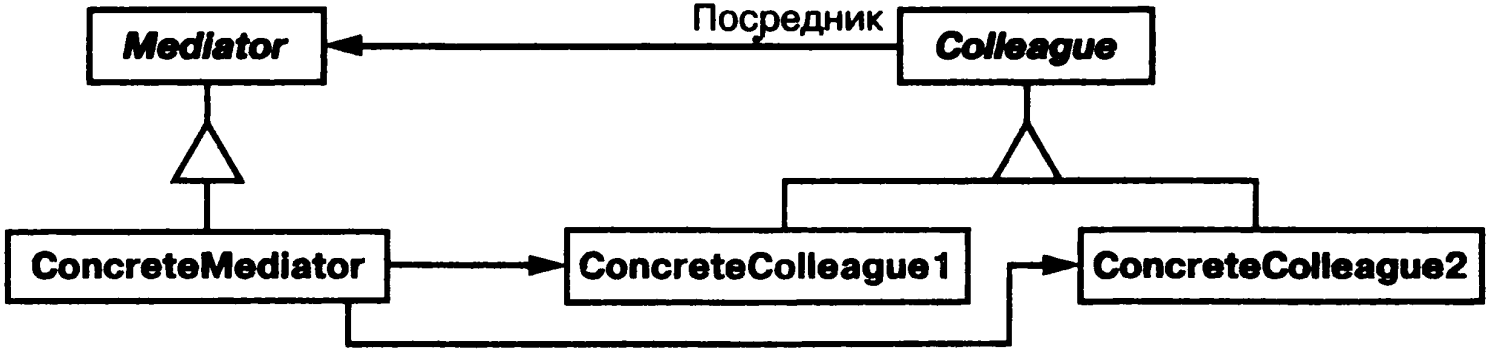
Цепочка обязанностей– примечания

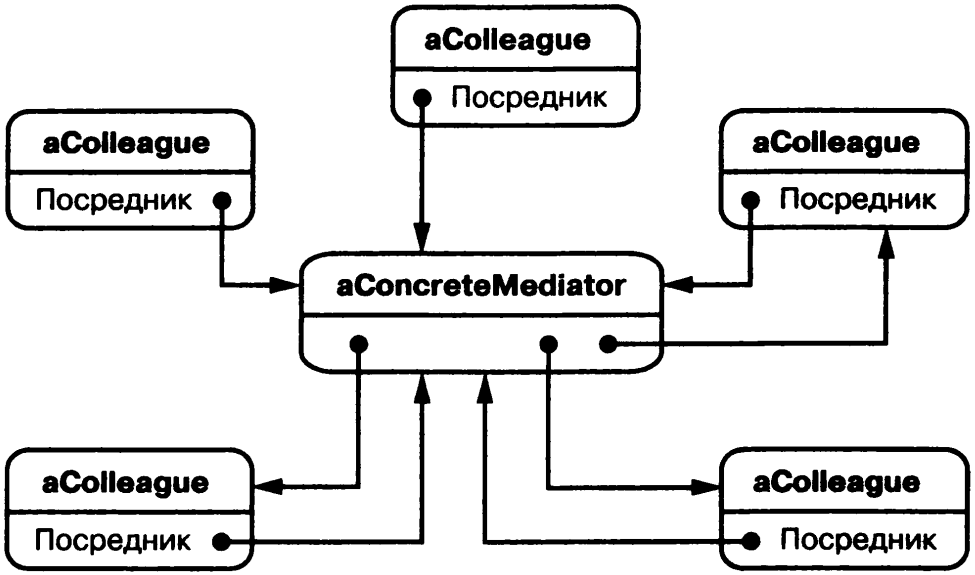
- Ослабление связанности – конкретный обработчик неизвестен отправителю
- Гибкость распределения обязанностей – в цепочку легко добавить нового участника
- Получение не гарантировано – запрос может пройти всю цепочку и остаться необработанным
- Запрос может быть инкапсулирован в объект и нести дополнительный данные, такие как флаг обработанности, например.
- Часто применяется в сочетании с Компоновщиком (см. пример), где Родитель является Преемником.

Посредник

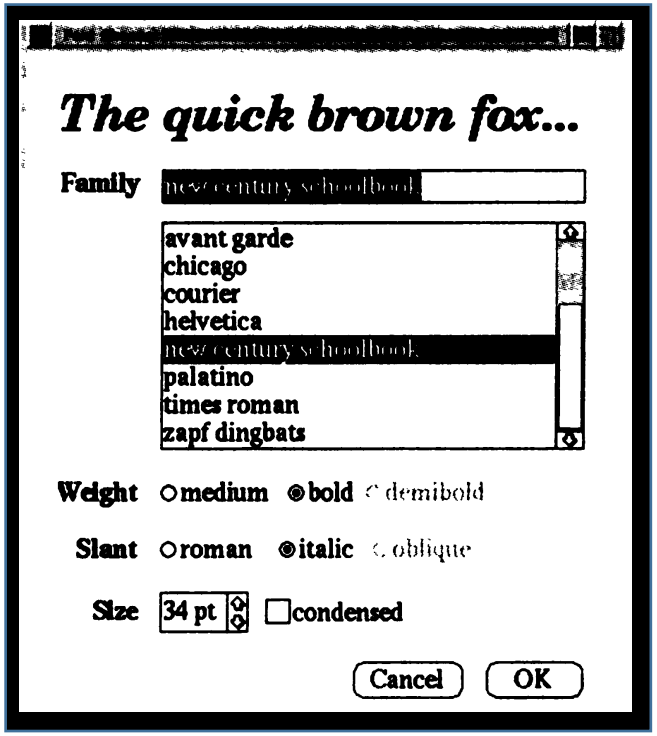
- Назначение:
  - Определяет объект, инкапсулирующий способ взаимодействия множества объектов.
  - Обеспечивает слабую связанность системы, избавляя объекты от необходимости знать друг о друге.
- Применимость:
  - Если имеются объекты, связи между которым сложны и четко определены, при этом, получающиеся зависимости не структурированы и сложны для понимания;
  - Нельзя повторно использовать объект, т.к. он обменивается информацией с множеством других объектов;
  - Поведение, распределенное между множеством классов, должно поддаваться настройке без порождения множества подклассов.

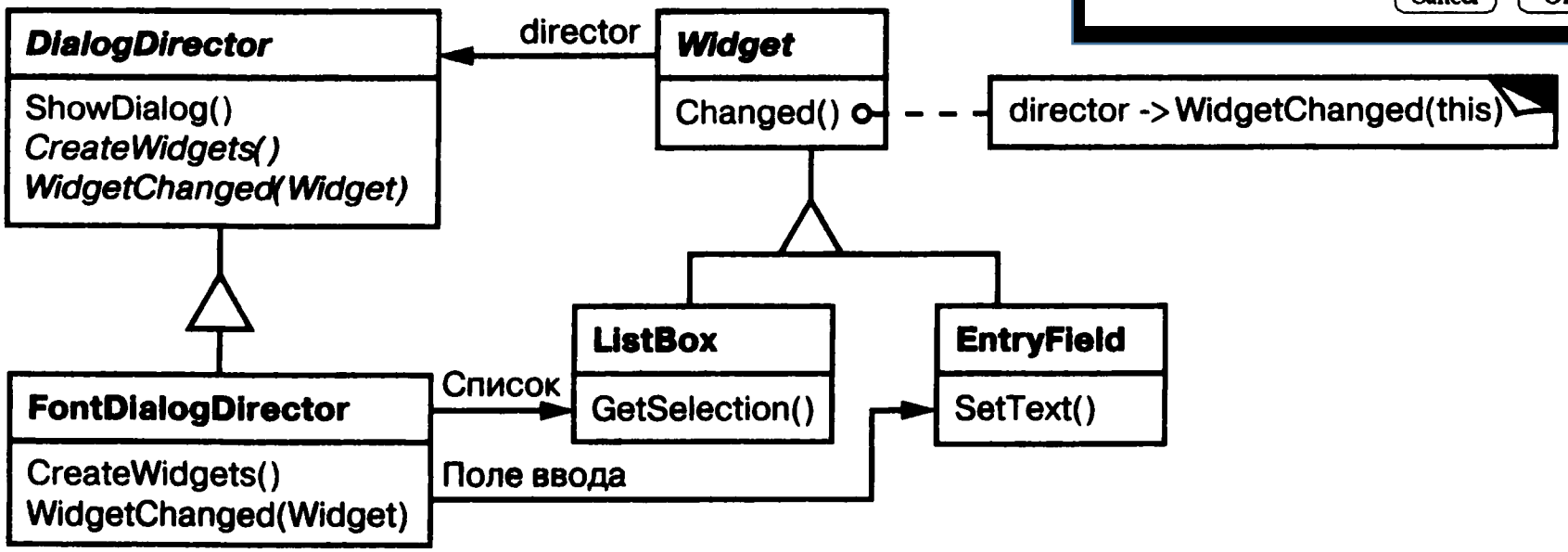
Посредник – структура



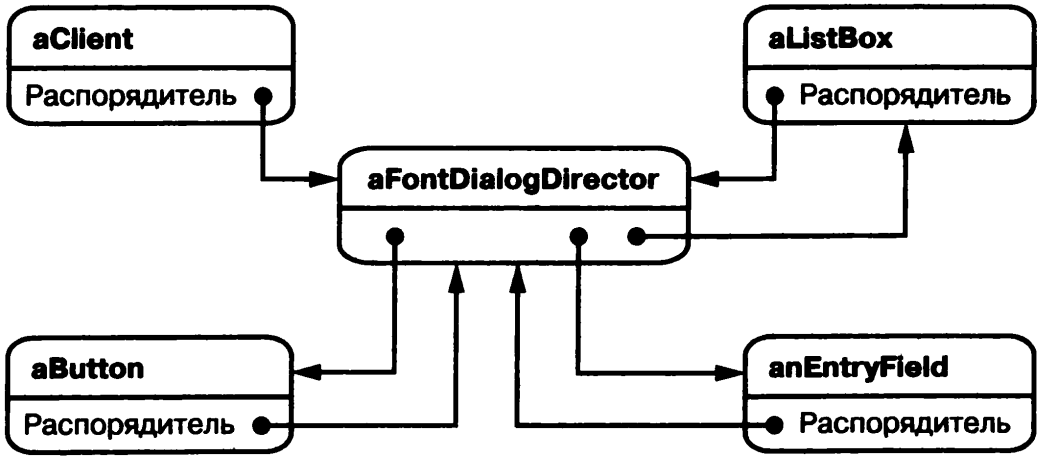


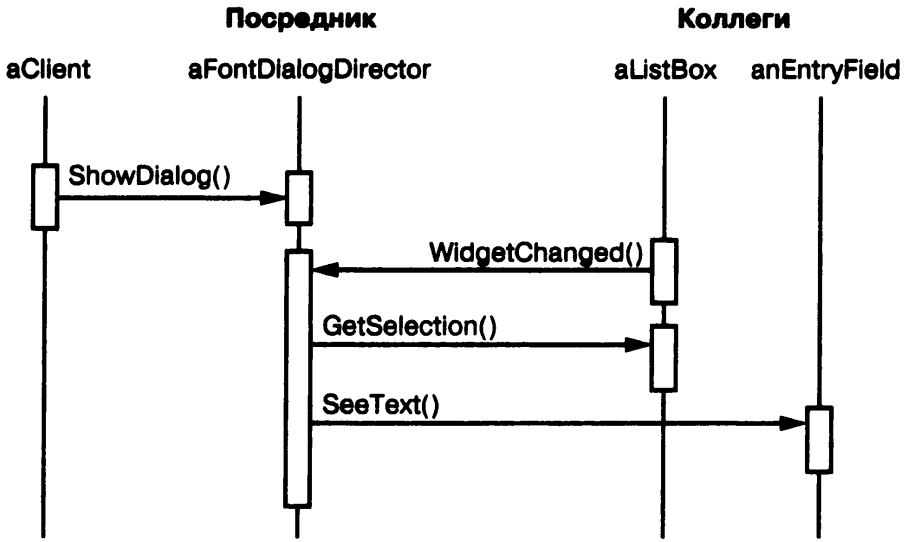
Посредник – пример





Посредник – пример





Посредник – пример

```
class FileSelectionDialog;
class Widget {
  public:
    Widget(FileSelectionDialog *mediator, char *name) {
        _mediator = mediator;
        strcpy(_name, name);
    }
    virtual void changed() { _mediator->widgetChanged(this); }
    virtual void updateWidget() = 0;
    virtual void queryWidget() = 0;
  protected:
    char _name[20];
  private:
    FileSelectionDialog *_mediator;
};
```

Посредник – пример

```
class List: public Widget
{
  public:
    List(FileSelectionDialog *dir, char *name):
Widget(dir, name){}
    void queryWidget()
    {
        cout<<"   "<<_name<<“ list?"<<endl;
    }
    void updateWidget()
    {
        cout<<"   "<<_name<<“ list!" << endl;
    }
};
```

```
class Edit: public Widget
{
  public:
    Edit(FileSelectionDialog *dir, char *name):
Widget(dir, name){}
    void queryWidget()
    {
        cout<<"   "<< _name <<" edit?"<< endl;
    }
    void updateWidget()
    {
        cout<<"   "<<_name<<"edit!" << endl;
    }
};
```

Посредник – пример

```
class FileSelectionDialog {
  public:
    enum Widgets { FilterEdit, DirList, FileList, SelectionEdit };
    FileSelectionDialog() {
        _components[FilterEdit] = new Edit(this, "filter");
        _components[DirList] = new List(this, "dir");
        _components[FileList] = new List(this, "file");
        _components[SelectionEdit] = new Edit(this, "selection");
    }
    virtual ~FileSelectionDialog() { for (int i = 0; i < 4; i++) delete _components[i]; } ;
    void handleEvent(int which) { _components[which]->changed(); }
    virtual void widgetChanged(); //реализация вынесена далее
  private:
    Widget *_components[4];
};
```

Посредник – пример

```
virtual void widgetChanged(Widget *theChangedWidget) {
    if (theChangedWidget == _components[FilterEdit]) {
        _components[FilterEdit]->queryWidget();
        _components[DirList]->updateWidget();
        _components[FileList]->updateWidget();
        _components[SelectionEdit]->updateWidget();
    } else if (theChangedWidget == _components[DirList]) {
        _components[DirList]->queryWidget();
        _components[FileList]->updateWidget();
        _components[FilterEdit]->updateWidget();
        _components[SelectionEdit]->updateWidget();
    } else if (theChangedWidget == _components[FileList]) {
        _components[FileList]->queryWidget();
        _components[SelectionEdit]->updateWidget();
    }  else if (theChangedWidget == _components[SelectionEdit]) {
        _components[SelectionEdit]->queryWidget();
        cout << "   file opened" << endl; }
  }
```

Посредник – пример

```
int main() {
  FileSelectionDialog fileDialog;
  int i;

  cout << "Exit[0], Filter[1], Dir[2], File[3], Selection[4]: ";
  cin >> i;

  while (i)  {
    fileDialog.handleEvent(i - 1);
    cout << "Exit[0], Filter[1], Dir[2], File[3], Selection[4]: ";
    cin >> i;
  }
}
```

```
Exit[0], Filter[1], Dir[2], File[3], Selection[4]: 1
  filter edit?
  dir list!
  file list!
  selection edit updated
Exit[0], Filter[1], Dir[2], File[3], Selection[4]: 2
  dir list?
  file list!
  filter edit!
  selection edit!
Exit[0], Filter[1], Dir[2], File[3], Selection[4]: 3
  file list?
  selection edit?
Exit[0], Filter[1], Dir[2], File[3], Selection[4]: 4
  selection edit?
  file opened
Exit[0], Filter[1], Dir[2], File[3], Selection[4]: 3
  file list?
  selection edit!
```

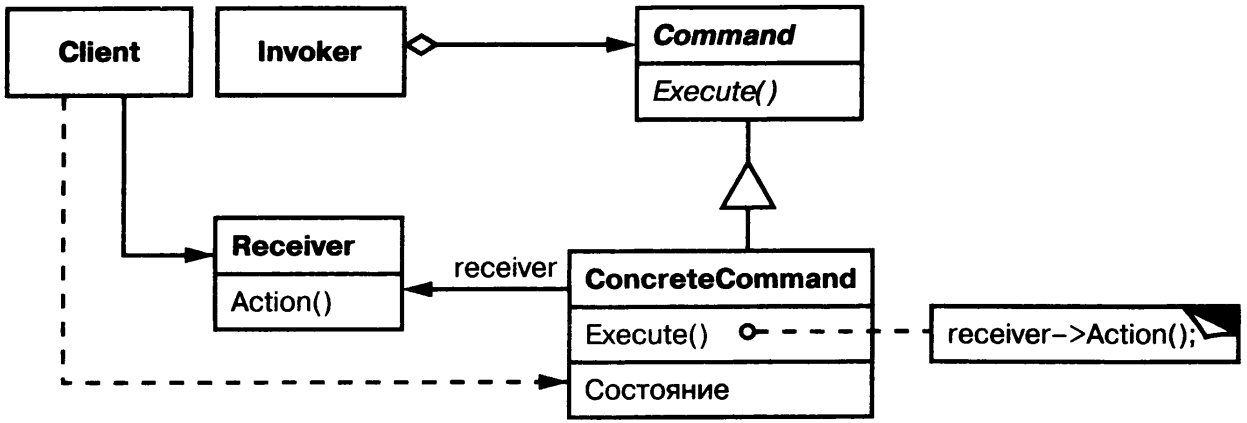
Посредник – примечания

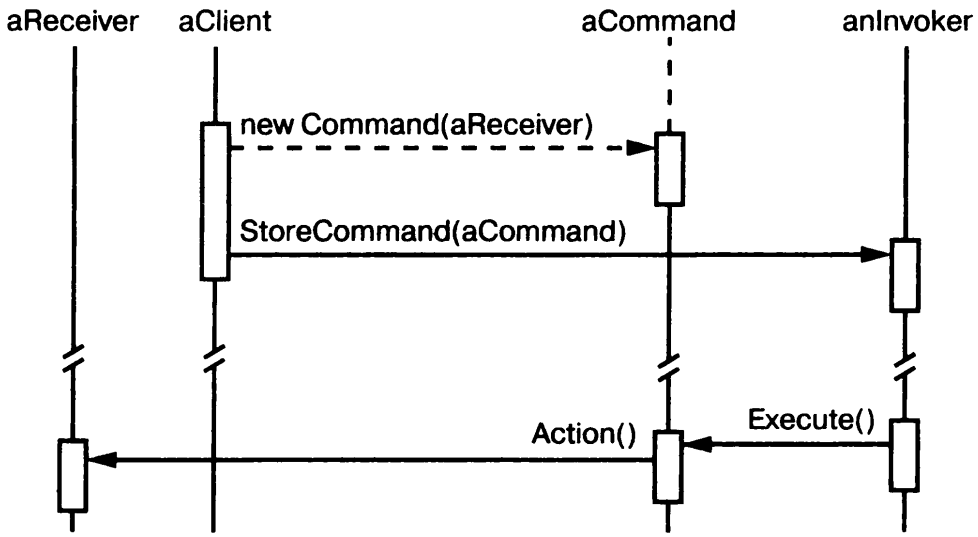
- Снижает чисто порождаемых подклассов – порождаем одного нового Посредника, а не подкласс для каждого Коллеги.
- Устраняет связность между Коллегами.
- Упрощает протокол взаимодействия объектов (1:M вместо M:M).
- Абстрагирует и централизует управление группой объектов.
- Хорошо сочетается с паттерном Наблюдатель для уведомления Посредника об изменениях Коллег.

Команда

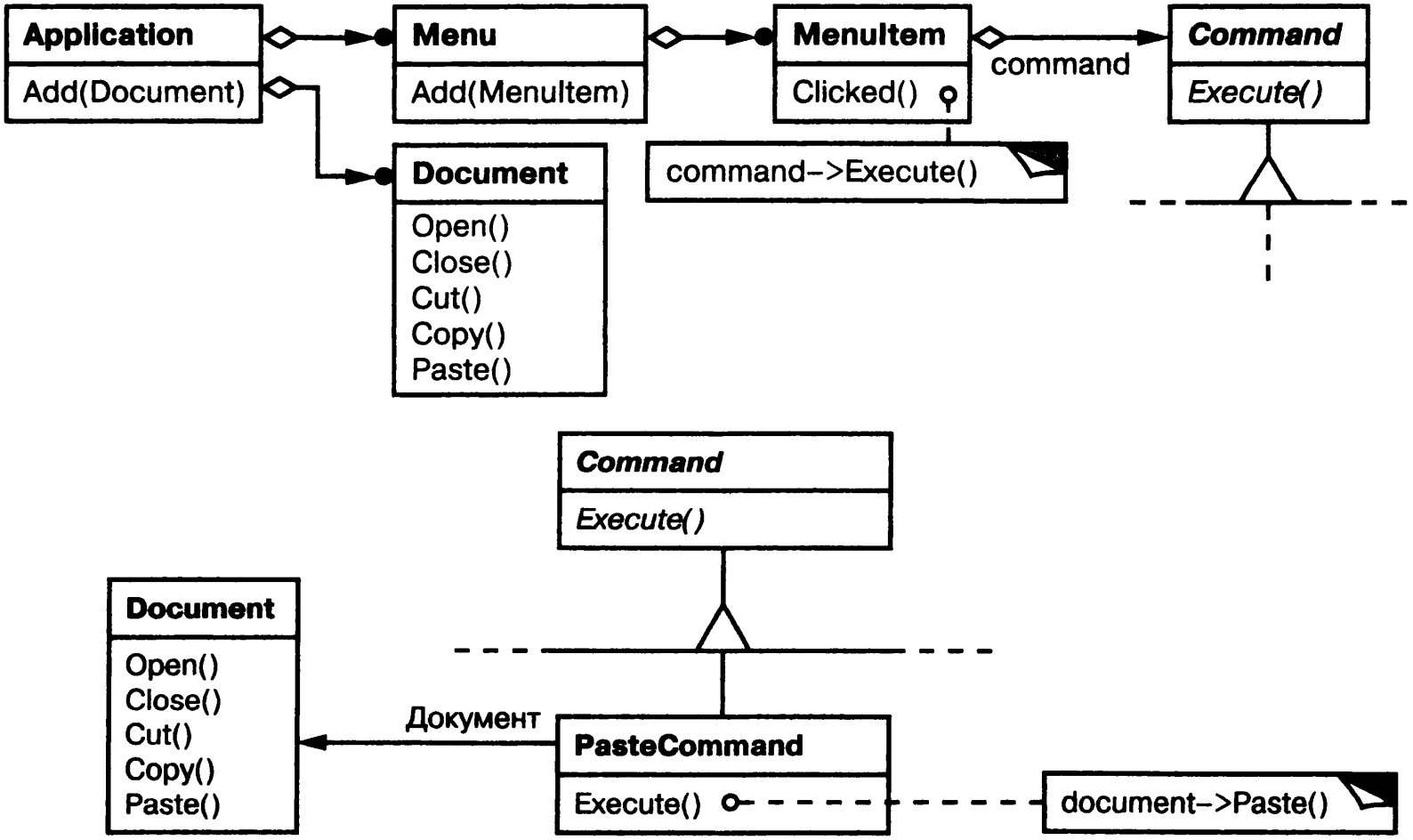
- Назначение:
  - Инкапсулирует запрос как объект, позволяя тем самым задавать параметры клиентов для обработки соответствующих запросов, ставить запросы в очередь и поддерживать отмену операций.
- Применимость:
  - Чтобы параметризировать объекты выполняемым действием (объектная обертка над указателем на функцию) – коллбэки, экшны и т.п.;
  - Чтобы определять, ставить в очередь и выполнять запросы в разное время;
  - Чтобы поддерживать отмену операций;
  - Чтобы поддерживать протоколирование операций;

Команда – структура





Команда – пример



Команда– пример

```
class CalculatorCommand : public Command //ConcreteCommand
{
   char _operator;
   int _operand;
   Calculator* _calculator;
   char Undo(char oprtr) {
      switch (oprtr) {
         case '+': return '-';
         case '-': return '+';
         case '*': return '/';
         case '/': return '*';
      }
   }
  public:
   CalculatorCommand(Calculator* calculator, char oprtr, int oprnd) {
      _calculator = calculator; _operator = oprtr; _operand = oprnd;
   }
   void Execute() { _calculator->Operation(_operator, _operand); }
   void UnExecute() {_calculator->Operation(Undo(_operator), _operand); }
};
```

```
class Command { //Command
  public:
   virtual void Execute() = 0;
   virtual void UnExecute() = 0;
};
```

Команда– пример

```
class User { //Invoker
    Calculator* _calculator = new Calculator();
    Command* _commands[10];
    int _current = 0;
    int _count = 0;
   public:
    void Redo(int levels)  {
        cout<< "\n---- Redo "<<levels<<" levels ";
        for (int i = 0; i < levels; i++) if (_current < _count - 1) _commands[_current++]->Execute();
    }
    void Undo(int levels)  {
        cout<< "\n---- Undo "<<levels<<" levels ";
        for (int i = 0; i < levels; i++) if (_current > 0)  _commands[--_current]->UnExecute();
    }
    void Compute(char oprtr, int oprnd)
    {
        Command* command = new CalculatorCommand(_calculator, oprtr, oprnd);
        command->Execute();
        _commands[_current++] = command; // Добавить команду в список отмены
        _count=_current;
    }
};
```

Команда– пример

```
class Calculator{ //Receiver
  int _curr = 0;
  public:
   void Operation(char oprtr, int oprnd) {
      switch (oprtr)  {
         case '+': _curr += oprnd; break;
         case '-': _curr -= oprnd; break;
         case '*': _curr *= oprnd; break;
         case '/': _curr /= oprnd; break;
      }
      cout<<"\nCurrent value = "<<_curr<<" ,following "<<oprtr<<" "<< oprnd;
   }
};
int main()
{
   User* user = new User(); //моделируем пользователя,
   user->Compute('+', 100); //работающего с
   user->Compute('-', 50);  //калькулятором
   user->Compute('*', 10);
   user->Compute('/', 2);
   user->Undo(4); //отменяем 4 операции
   user->Redo(3); //восстанавливаем 3 опрерации
}
```

```
Current value = 100 ,following + 100
Current value = 50 ,following - 50
Current value = 500 ,following * 10
Current value = 250 ,following / 2
---- Undo 4 levels
Current value = 500 ,following * 2
Current value = 50 ,following / 10
Current value = 100 ,following + 50
Current value = 0 ,following - 100
---- Redo 3 levels
Current value = 100 ,following + 100
Current value = 50 ,following - 50
Current value = 500 ,following * 10
```

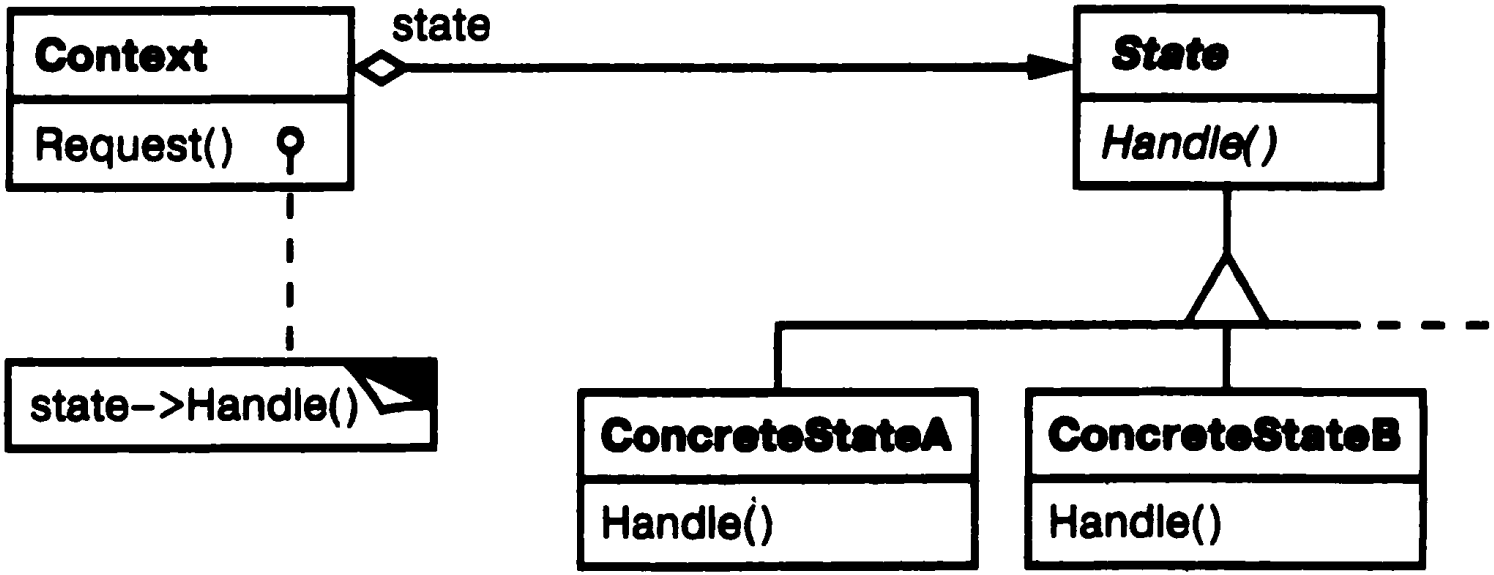
Команда – примечания

- Команда разрывает связь между объектом-инициатором и объектом-исполнителем.
- Команды – обычные объекты, которые можно хранить, передавать, расширять и т.п.
- Для добавления новой команды – достаточно добавить новый класс, и нет необходимости менять существующие.
- Из простых команд можно собрать составные, пользуясь паттерном Компоновщик
- Для отмены действия Команда должна иметь два метода – действия и его отмены (Do/Undo).

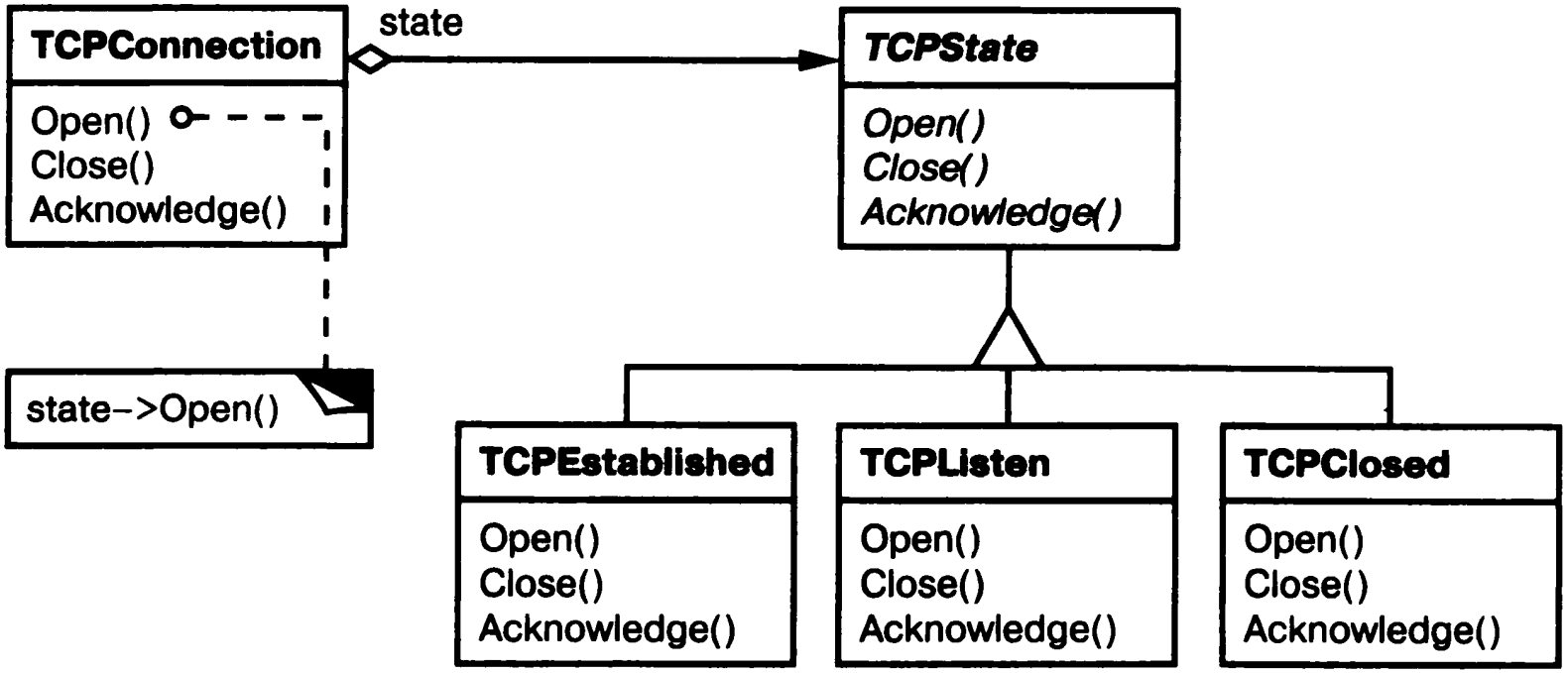
Состояние

- Назначение:
  - Позволяет объекту менять свое поведение в зависимости от состояния. Извне создается впечатление, что изменился класс объекта.
- Применимость:
  - Когда поведение объекта зависит от его состояния и должно изменяться во время выполнения;
  - Когда в коде операций встречаются состоящие из многих ветвей условные операторы, в которых выбор ветви определяется состоянием (часто разные операции при этом имеют схожую структуру ветвлений)

Состояние – структура



Состояние – пример



Команда– пример

```
class Machine {
  class State *current;
  public:
    Machine();
    void setCurrent(State *s) { current = s; }
    void on();
    void off();
};

class State {
  public:
    virtual void on(Machine *m) { cout << "   already ON\n"; }
    virtual void off(Machine *m) { cout << "   already OFF\n"; }
};

void Machine::on() { current->on(this); }
void Machine::off() { current->off(this); }
```

Команда– пример

```
class ON: public State{
  public:
    ON() { cout << "   ON-ctor "; };
    ~ON() { cout << "   dtor-ON\n"; };
    void off(Machine *m);
};

class OFF: public State {
  public:
    OFF() { cout << "   OFF-ctor "; };
    ~OFF() { cout << "   dtor-OFF\n"; };
    void on(Machine *m) {
        cout << "   going from OFF to ON";
        m->setCurrent(new ON());
        delete this;
    }
};

void ON::off(Machine *m) {
  cout << "   going from ON to OFF";
  m->setCurrent(new OFF());
  delete this;
}
```

Команда– пример

```
Machine::Machine() {
  current = new OFF();
  cout << '\n';
}

int main()
{
  Machine fsm;
  int num;
  while (1)
  {
    cout << "Enter 0/1: ";
    cin >> num;
    num ? fsm.on() : fsm.off();
  }
}
```

```
 OFF-ctor
Enter 0/1: 0
   already OFF
Enter 0/1: 0
   already OFF
Enter 0/1: 1
   going from OFF to ON   ON-ctor    dtor-OFF
Enter 0/1: 1
   already ON
Enter 0/1: 0
   going from ON to OFF   OFF-ctor    dtor-ON
Enter 0/1: 1
   going from OFF to ON   ON-ctor    dtor-OFF
Enter 0/1:
```

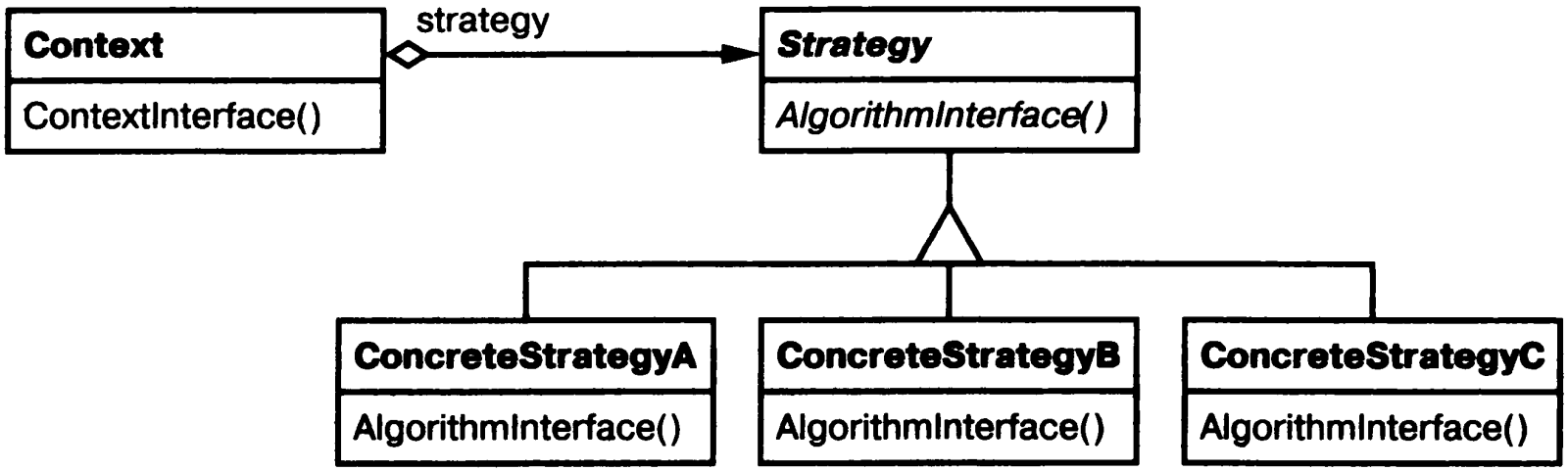
Состояние – примечания

- Паттерн Состояние локализует зависящее от состояния поведения в различных классах, описывающих каждое из состояний.
- Делает явными и консистентными (непротиворечивыми) переходы между состояниями.
- Объекты-состояния могут быть разделяемыми, а все экземплярные поля могут быть вынесены в Контекст.

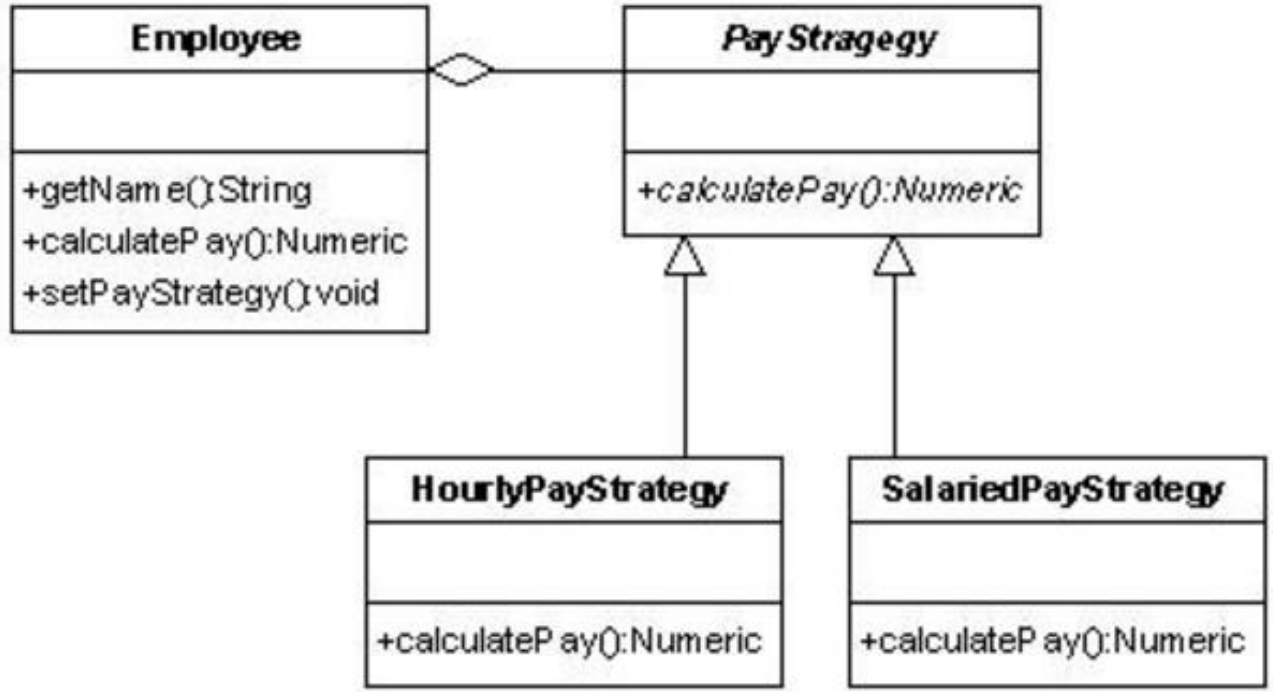
Стратегия

- Назначение:
  - Определяет семейство алгоритмов, инкапсулирует каждый из них и делает их взаимозаменяемыми. Стратегия позволяет изменять алгоритмы независимо от клиентов, которые ими пользуются.
- Применимость:
  - Имеется ряд родственных классов, отличающихся только поведением;
  - Необходимо иметь несколько вариантов алгоритма;
  - Необходимо скрыть детали реализации алгоритма от клиента
  - Класс определяет ряд вариантов поведения, представленных многочисленными ветвлениями, при этом проще перенести код из ветвей в классы Стратегии.

Стратегия – структура



Стратегия – пример



Стратегия– пример

```
class Strategy;
class TestBed
{
  public:
    enum StrategyType { Dummy, Left, Right, Center };
    TestBed() { strategy_ = NULL; }
    void setStrategy(int type, int width);
    void doIt();
  private:
    Strategy *strategy_;
};
class Strategy
{
  public:
    Strategy(int width): width_(width) {}
    void format() {
        char word[30];
        cin >> word;
        justify(word);
    }
  protected:
    int width_;
  private:
    virtual void justify(char *line) = 0;
};
```

Стратегия– пример

```
class LeftStrategy: public Strategy
{
 public:
   LeftStrategy(int width): Strategy(width){}
 private:
    /* virtual */void justify(char *line) {
       cout << line << endl;
       line[0] = '\0';
   }
};

class RightStrategy: public Strategy
{
 public:
   RightStrategy(int width): Strategy(width){}
 private:
    /* virtual */void justify(char *line) {
       char buf[80];
       int offset = width_ - strlen(line);
       memset(buf, ' ', 80);
       strcpy(&(buf[offset]), line);
       cout << buf << endl;
       line[0] = '\0';
   }
};
```

```
class CenterStrategy: public Strategy
{
 public:
   CenterStrategy(int width): Strategy(width){}
 private:
    /* virtual */void justify(char *line) {
       char buf[80];
       int offset = (width_ - strlen(line)) / 2;
       memset(buf, ' ', 80);
       strcpy(&(buf[offset]), line);
       cout << buf << endl;
       line[0] = '\0';
   }
};
```

Стратегия– пример

```
void TestBed::setStrategy(int type, int width){
  delete strategy_;
  if (type == Left)  strategy_ = new LeftStrategy(width);
  else if (type == Right) strategy_ = new RightStrategy(width);
  else if (type == Center) strategy_ = new CenterStrategy(width);
}

void TestBed::doIt() { strategy_->format(); }

int main(){
  TestBed test;
  int answer, width;
  cout << "Exit(0) Left(1) Right(2) Center(3): ";
  cin >> answer;
  while (answer) {
    cout << "Width: ";
    cin >> width;
    test.setStrategy(answer, width);
    test.doIt();
    cout << "Exit(0) Left(1) Right(2) Center(3): ";
    cin >> answer;
  }
  return 0;
}
```

```
Exit(0) Left(1) Right(2) Center(3): 1
Width: 30
aswerwerwer
aswerwerwer
Exit(0) Left(1) Right(2) Center(3): 2
Width: 30
sdfsdfsdf
                sdfsdfsdf
Exit(0) Left(1) Right(2) Center(3): 3
Width: 30
sdfsdfsdfsd
        sdfsdfsdfsd
Exit(0) Left(1) Right(2) Center(3): 0
```

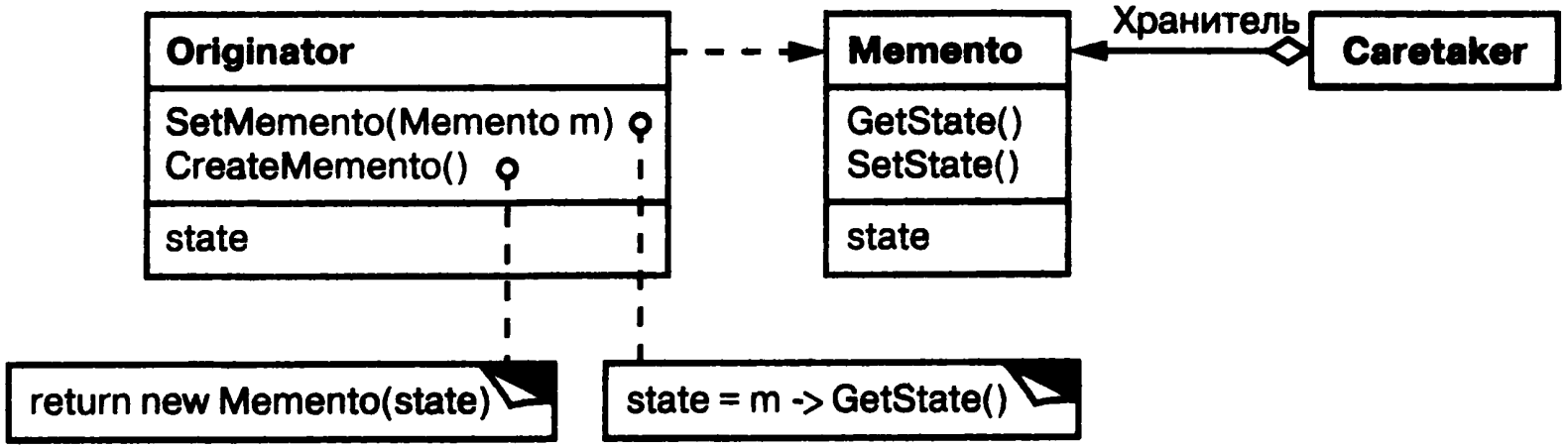
Стратегия – примечания

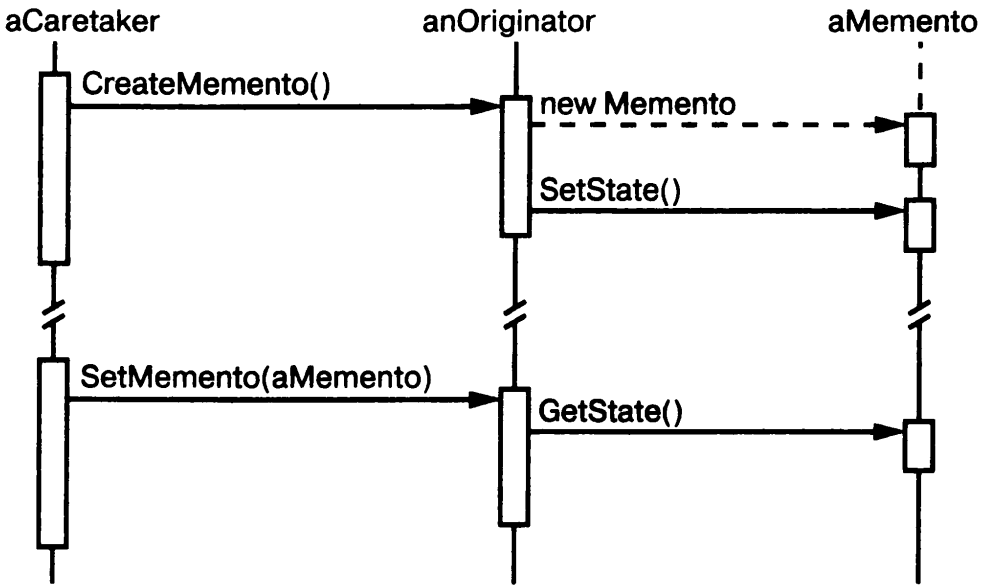
- Общая часть алгоритма может быть вынесена в базовый класс (расширение через Шаблонный Метод).
- Стратегия является альтернативой порождению подклассов Контекста.
- Стратегия упрощает логику операций Контекста.
- Стратегия дает возможность варьировать поведение, подставляя нужную стратегию. При этом клиент должен знать о имеющихся Стратегиях и их особенностях.
- Стратегии чаще всего являются разделяемыми объектами.

Хранитель

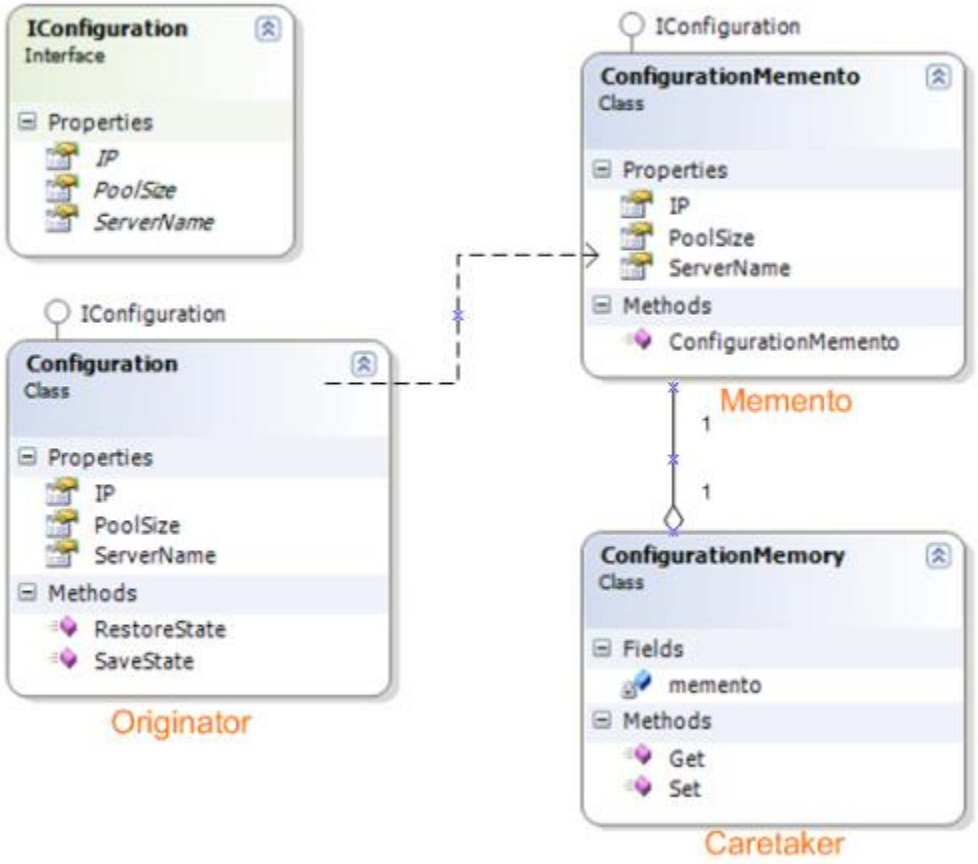
- Назначение:
  - Не нарушая инкапсуляции, фиксирует и выносит за пределы объекта его внутренне состояние так, чтобы позднее по нему можно было восстановить объект.
- Применимость:
  - Необходимо сохранить моментальный снимок объекта (или его части), чтобы впоследствии иметь возможность восстановить объект в том же состоянии.
  - Прямое получение этого состояния раскрывает детали реализации и нарушает инкапсуляцию.

Хранитель – структура





Хранитель – пример



Хранитель– пример

```
class Memento {
  private:
   int mVar1;
  private: //конструктор закрыт!
   Memento(){}
   void setVar1(int var) {mVar1 = var;}
   int getVar1() const {return mVar1;}
  friend class Orginator;
};

class Orginator {
  private:
   int mVar1;
  public:
   void setVar1(int var) {mVar1 = var;}
   int getVar1() const {return mVar1;}
   void print() {std::cout << mVar1 << "\n"; }
   Memento* saveState() {
      Memento* memento = new Memento;
      memento->setVar1(getVar1());
      return memento;
   }
   void restoreState(Memento* memento) {setVar1(memento->getVar1()); }
};
```

Хранитель– пример

```
int main(){
   Orginator* originator = new Orginator;
   originator->setVar1(10);
   originator->setVar1(11);

   originator->print();
   Memento* memento = originator->saveState();

   originator->setVar1(12);
   originator->print();

   originator->restoreState(memento);
   originator->print();
}
```

```
11
12
11
```

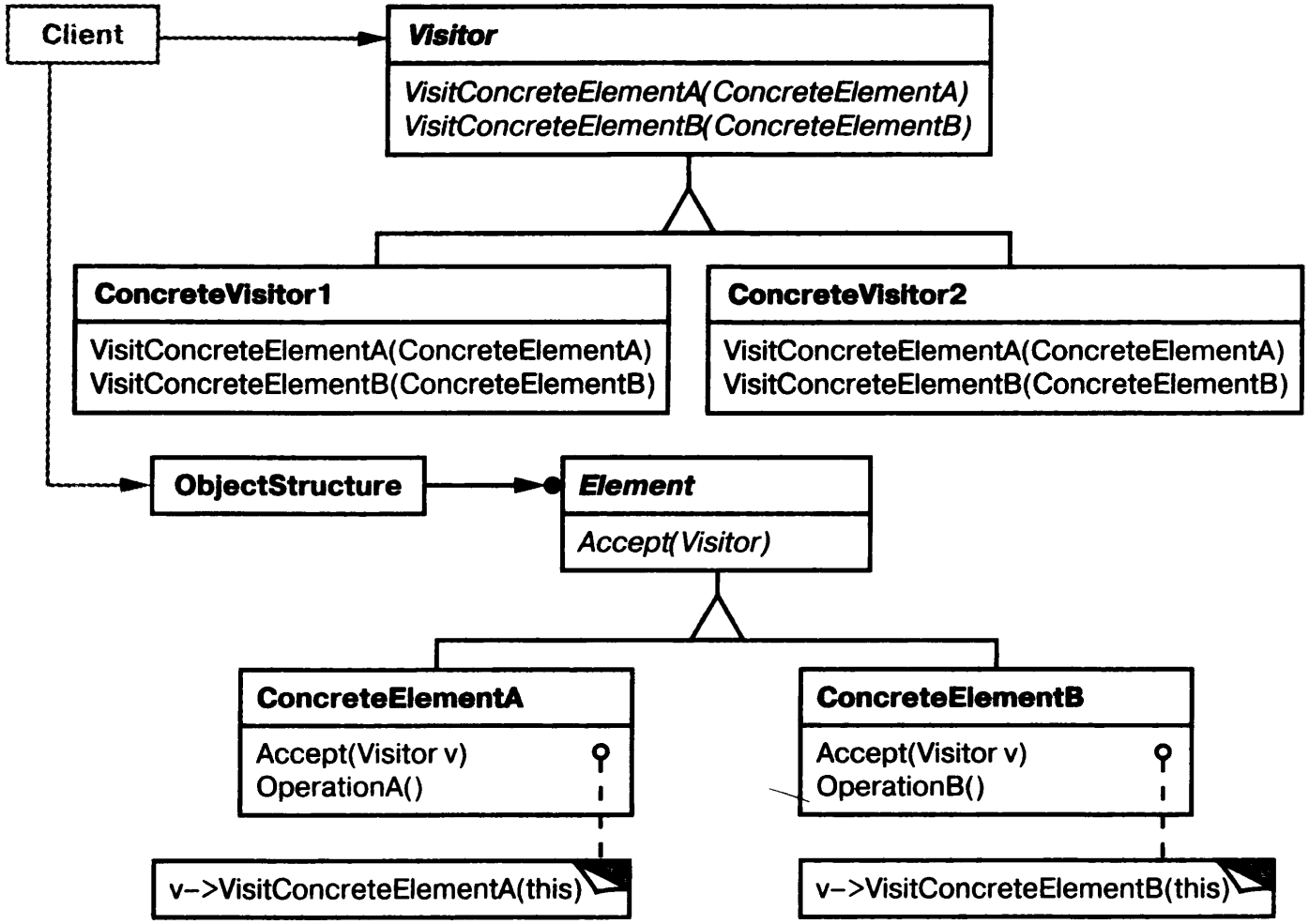
Хранитель – примечания

- В идеале, Хранитель должен обеспечивать доступ ко всей информации состояния только своему Хозяину. Для остальных классов, включая Посыльного эта информация должна быть закрыта.
- Возможно хранение инкрементных изменений состояния (используя паттерн Команда).

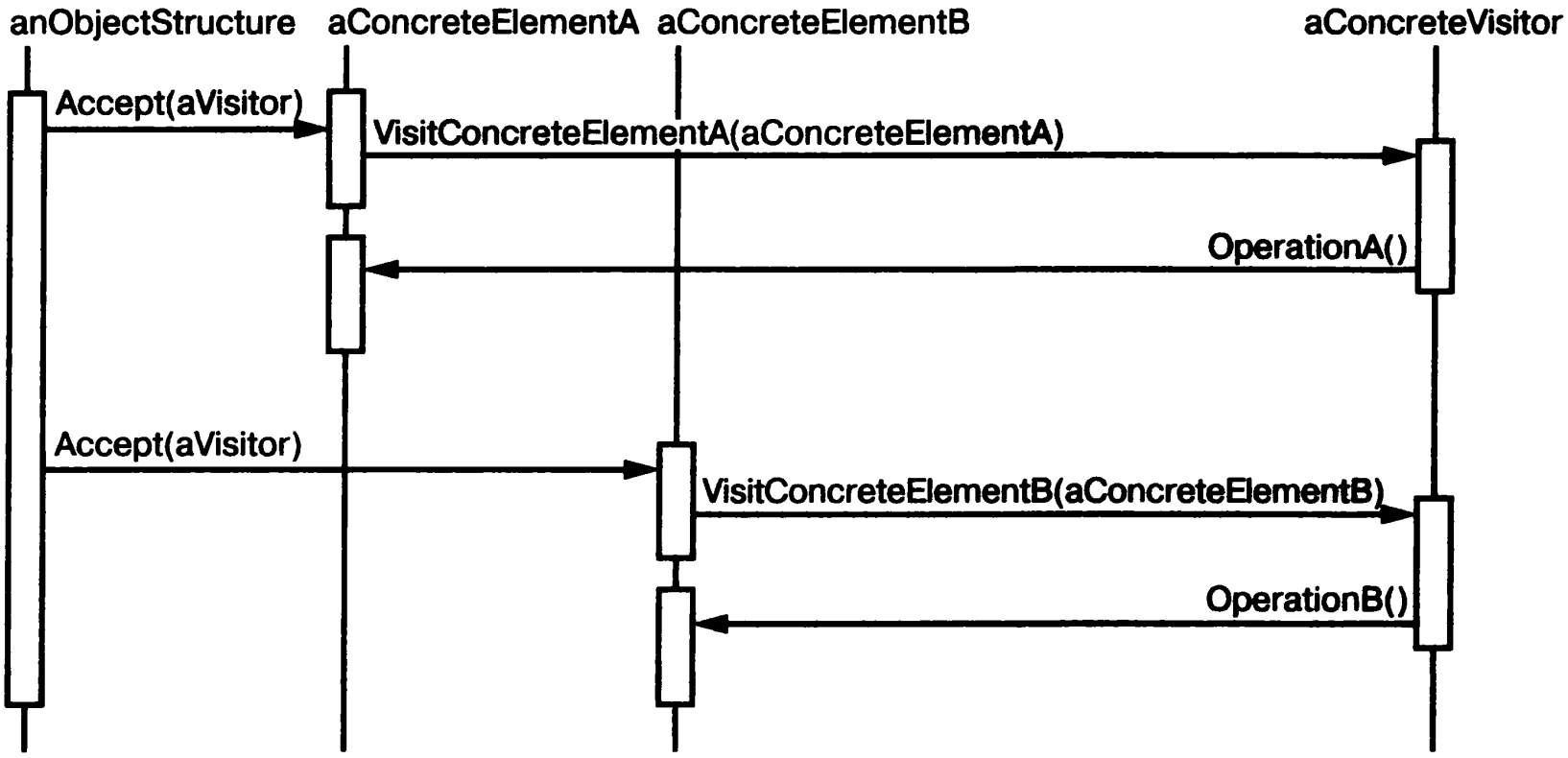
Посетитель

- Назначение:
  - Описывает операцию, выполняемую с каждым объектом из некоторой структуры, при этом не изменяя классы этих объектов.
- Применимость:
  - В структуре присутствуют объекты многих классов с различными интерфейсами и есть необходимость выполнять над ними операции, зависящие от конкретных классов;
  - Над данными объектам необходимо выполнять разнообразные, не связанные между собой операции и вы не хотите засорять такими операциями классы.
  - Классы, описывающие структуру, меняются редко, но новые операции добавляются часто.

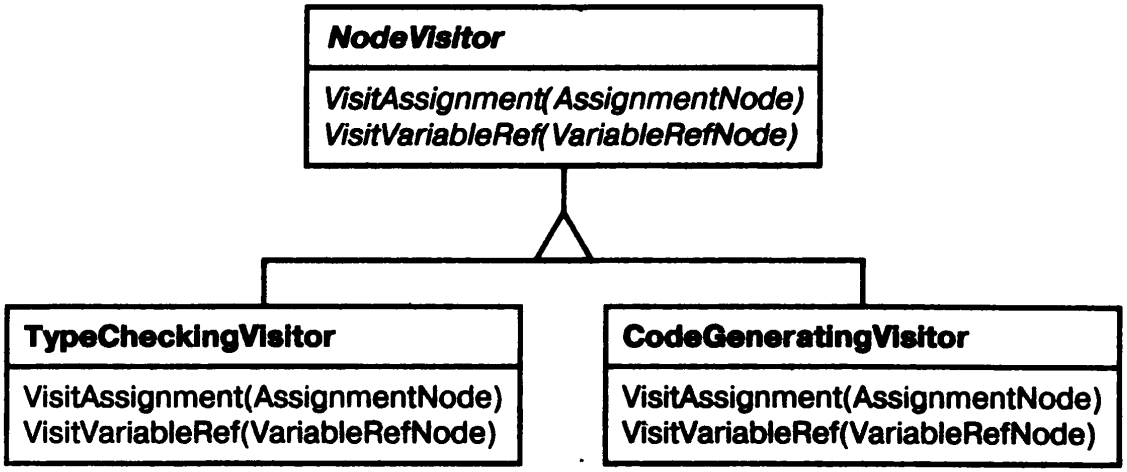
Посетитель – структура

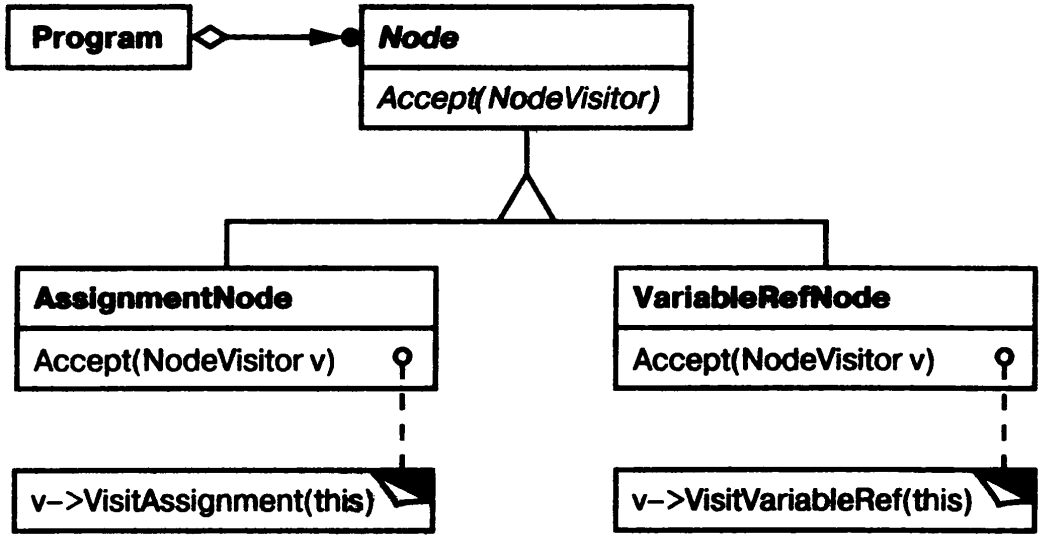


Посетитель – структура



Посетитель – пример





Хранитель– пример

```
class Element {
  public:
    virtual void accept(class Visitor &v) = 0;
};

class Student: public Element {
  public:
     void accept(Visitor &v);
    string thiss() { return "Student"; }
};

class Teacher: public Element {
  public:
     void accept(Visitor &v);
    string that() { return "Teacher"; }
};

class Technician: public Element {
  public:
     void accept(Visitor &v);
    string theOther() { return "Technician"; }
};
```

Хранитель– пример

```
class Visitor {
  public:
    virtual void visit(Student *e) = 0;
    virtual void visit(Teacher *e) = 0;
    virtual void visit(Technician *e) = 0;
};

 void Student::accept(Visitor &v) { v.visit(this); }
 void Teacher::accept(Visitor &v) { v.visit(this); }
 void Technician::accept(Visitor &v) { v.visit(this); }

class FiremanVisitor: public Visitor {
    void visit(Student *e) { cout << "do firefighting instruction for " + e->thiss() << '\n'; }
    void visit(Teacher *e) { cout << "do firefighting instruction for " + e->that() << '\n'; }
    void visit(Technician *e) { cout << "do firefighting instruction for " + e->theOther() <<
'\n'; }
};

class MedicVisitor: public Visitor
{
    void visit(Student *e) { cout << "do vaccination on " + e->thiss() << '\n'; }
    void visit(Teacher *e) { cout << "do vaccination on " + e->that() << '\n'; }
    void visit(Technician *e) { cout << "do vaccination on " + e->theOther() << '\n'; }
};
```

Хранитель– пример

```
int main()
{
  Element *list[] = {new Student(), new Teacher(), new Technician() };
  FiremanVisitor Vasya;
  MedicVisitor Masha;
  for (int i = 0; i < 3; i++) list[i]->accept(Vasya);
  for (int i = 0; i < 3; i++) list[i]->accept(Masha);
}
```

```
do firefighting instruction for Student
do firefighting instruction for Teacher
do firefighting instruction for Technician
do vaccination on Student
do vaccination on Teacher
do vaccination on Technician
```

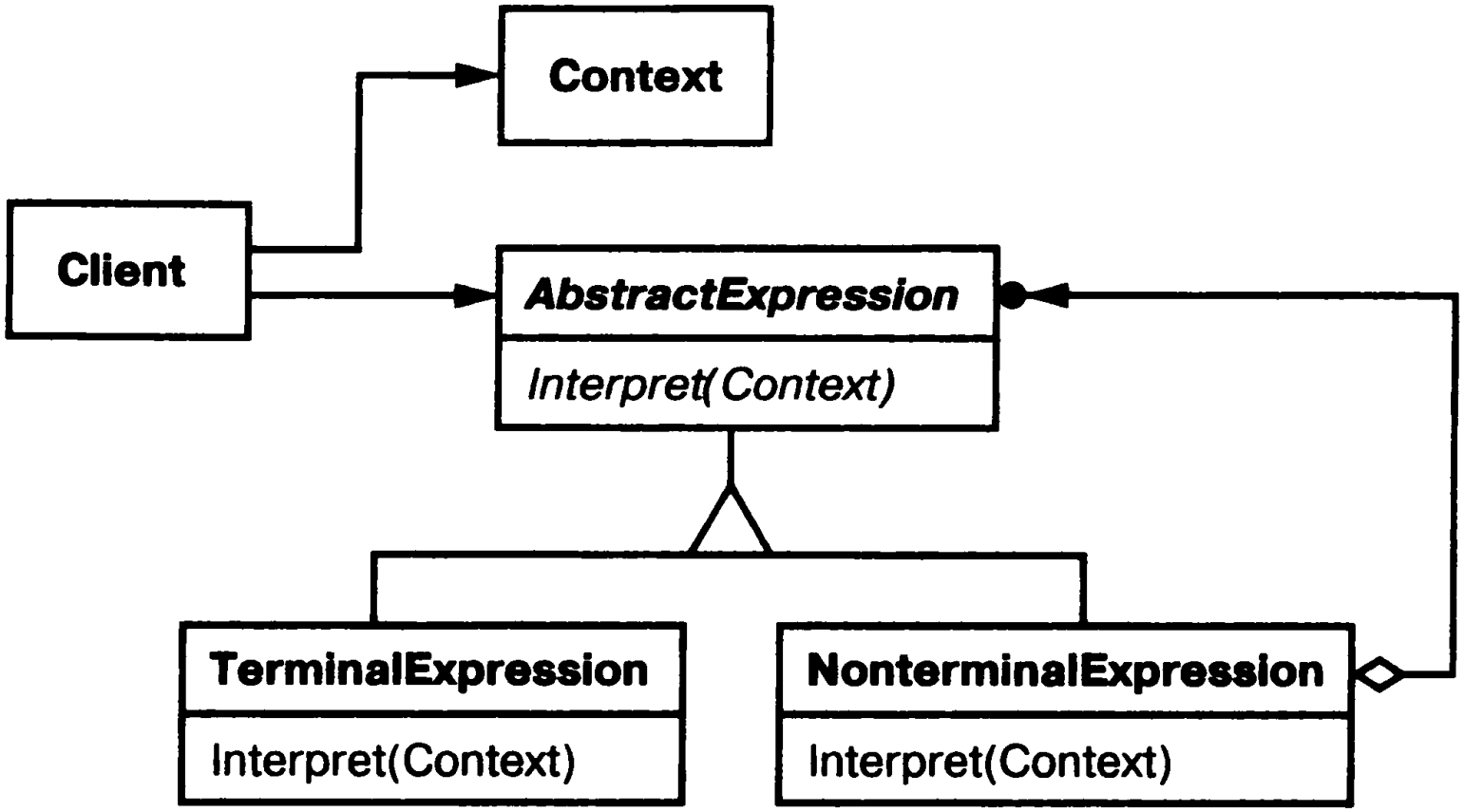
Посетитель – примечания

- Упрощает добавление новых операций, работающих на всей структуре объектов.
- Объединяет родственные операции и отделяет не имеющие отношения.
- Усложняет изменение структуры (добавление классов КонкретныхЭлементов)
- Таким образом, перед применением необходимо оценить, что будет меняться чаще – структура или алгоритмы работы с ней.
- Потенциальное нарушение инкапсуляции, т.к. Посетитель должен получить от Элемента достаточно информации для реализации своего алгоритма.
- Посетители могут аккумулировать состояние, что удобно для реализации агрегирующих алгоритмов.

Интерпретатор

- Назначение:
  - Для заданного языка определяет представление его грамматики, а также интерпретатор предложений этого языка.
- Применимость:
  - Если есть язык для интерпретации, предложения которого можно представить в виде абстрактных синтаксических деревьев;
  - Если грамматика языка достаточно проста (иначе решение сильно теряет в эффективности)
  - Эффективность не является главным критерием (это простой, но не самый эффективный способ работы)

Интерпретатор – структура



Интерпретатор – пример

- Римские числа:
  - Символ обозначает разряд:
    - I = 1
    - X = 10;
    - C = 100;
    - M = 1000.
  - Или полуразряд:
    - V = 5
    - L = 50
    - D = 500
  - Количество единиц в разряде:
    - 1 – 3 : N*{символ разряда} подряд
      - III, XX, C
    - 4 : {символ разряда}{символ следующего полуразряда}
      - IV, XL, CD
    - 5 : {символ следующего полуразряда}
    - 6-8 : {символ следующего полуразряда} N*{символ разряда}
      - VIII, LXX, DC
    - 9 : {символ разряда}{символ следующего разряда}
      - IX, XC, CM

Интерпретатор – пример

- Римские числа:
  - romanNumeral ::= {thousands} {hundreds} {tens} {ones}
  - thousands, hundreds, tens, ones ::= nine | four | {five} {one} {one} {one}
  - nine ::= "CM" | "XC" | "IX"
  - four ::= "CD" | "XL" | "IV"
  - five ::= 'D' | 'L' | 'V’
  - one ::= 'M' | 'C' | 'X' | 'I'

Интерпретатор – пример

```
class Thousand;
class Hundred;
class Ten;
class One;

class RNInterpreter {
  public:
    RNInterpreter();
    RNInterpreter(int){}
    int interpret(char*);
    virtual void interpret(char *input, int &total);
  protected:
    virtual char one(){}
    virtual char *four(){}
    virtual char five(){}
    virtual char *nine(){}
    virtual int multiplier(){}
  private:
    RNInterpreter *thousands;
    RNInterpreter *hundreds;
    RNInterpreter *tens;
    RNInterpreter *ones;
};
```

Интерпретатор – пример

```
void RNInterpreter::interpret(char *input, int &total){
   int index;
   index = 0;
   if (!strncmp(input, nine(), 2)) {
      total += 9 * multiplier();
      index += 2; }
   else if (!strncmp(input, four(), 2)) {
      total += 4 * multiplier();
      index += 2;}
   else {
      if (input[0] == five()) {
         total += 5 * multiplier();
         index = 1; }
      else
        index = 0;
      for (int end = index + 3; index < end; index++)
        if (input[index] == one())
          total += 1 * multiplier();
        else
          break;
   }
   strcpy(input, &(input[index]));
}
```

Интерпретатор – пример

```
class Thousand: public RNInterpreter {
  public:
    Thousand(int): RNInterpreter(1){}
  protected:
    char one() { return 'M'; }
    char *four() { return ""; }
    char five() { return '\0'; }
    char *nine() { return ""; }
    int multiplier() { return 1000; }
};

class Hundred: public RNInterpreter{
  public:
    Hundred(int): RNInterpreter(1){}
  protected:
    char one() { return 'C'; }
    char *four() { return "CD"; }
    char five() { return 'D'; }
    char *nine() { return "CM"; }
    int multiplier() { return 100; }
};
```

```
class Ten: public RNInterpreter {
  public:
    Ten(int): RNInterpreter(1){}
  protected:
    char one() { return 'X'; }
    char *four() { return "XL"; }
    char five() { return 'L'; }
    char *nine() { return "XC"; }
    int multiplier() { return 10; }
};

class One: public RNInterpreter{
  public:
    One(int): RNInterpreter(1){}
  protected:
    char one() { return 'I'; }
    char *four() { return "IV"; }
    char five() { return 'V'; }
    char *nine() { return "IX"; }
    int multiplier() { return 1; }
};
```

Интерпретатор – пример

```
RNInterpreter::RNInterpreter() {
  thousands = new Thousand(1);
  hundreds = new Hundred(1);
  tens = new Ten(1);
  ones = new One(1);
}
int RNInterpreter::interpret(char *input) {
  int total;
  total = 0;
  thousands->interpret(input, total);
  hundreds->interpret(input, total);
  tens->interpret(input, total);
  ones->interpret(input, total);
  if (strcmp(input, "")) return 0;
  return total;
}
int main() {
  RNInterpreter interpreter;
  char input[20];
  cout << "Enter Roman Numeral: ";
  while (cin >> input) {
    cout<<"   interpretation is "<<interpreter.interpret(input)<<endl<<"Enter Roman Numeral: ";
  }
}
```

```
Enter Roman Numeral: MMXVIII
  interpretation is 2018
Enter Roman Numeral: asd
  interpretation is 0
Enter Roman Numeral:
```