На предыдущей лекции

- Что такое шаблоны (паттерны) проектирования
- Порождающие шаблоны проектирования – предназначены для создания объектов, позволяя системе оставаться независимой как от самого процесса порождения, так и от типов порождаемых объектов:
  - Factory Method (Фабричный метод)
  - Abstract Factory (Абстрактная Фабрика)
  - Builder (Строитель)
  - Prototype (Прототип)
  - Singleton (Одиночка)

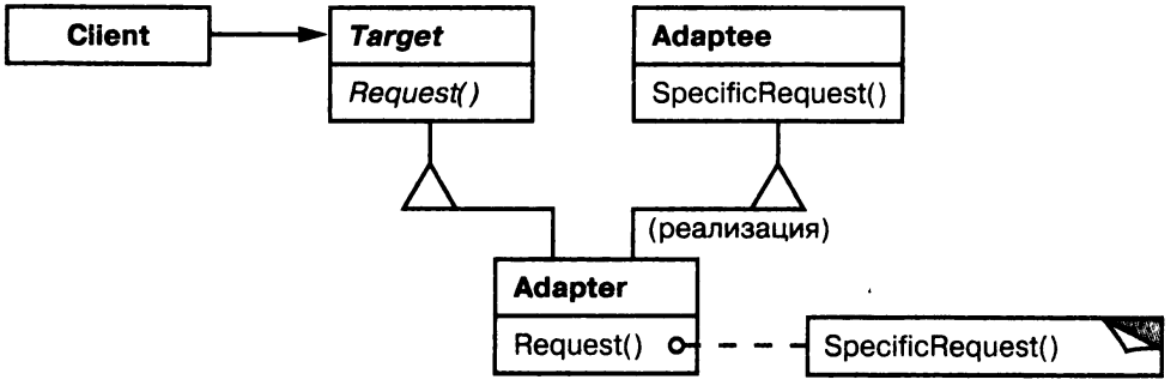
Структурные шаблоны

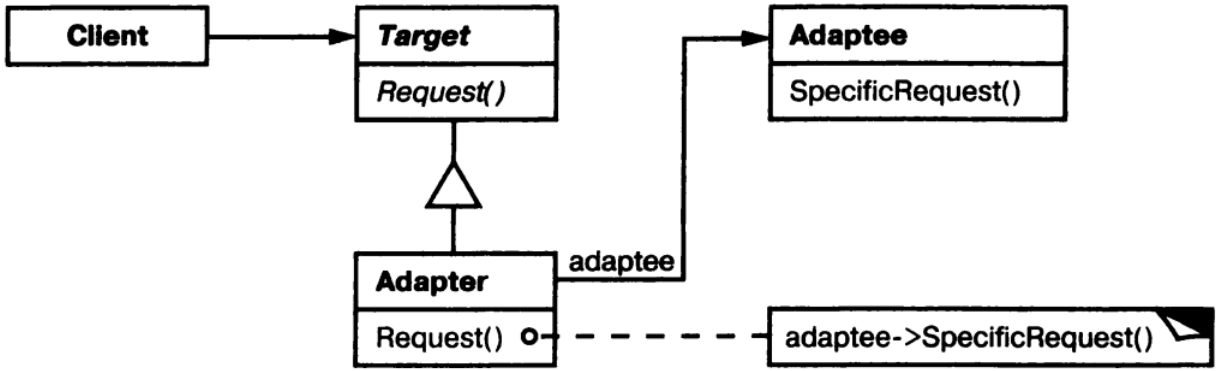
- Шаблоны, в которых рассматривается вопрос о том, как из классов и объектов образуются более крупные структуры.
- Действуют на уровне:
  - **Классов** - используют наследование для составления композиций из интерфейсов и реализаций.
  - **Объектов** – компонуют объекты для получения новой функциональности
- Шаблоны :
  - **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.

Адаптер

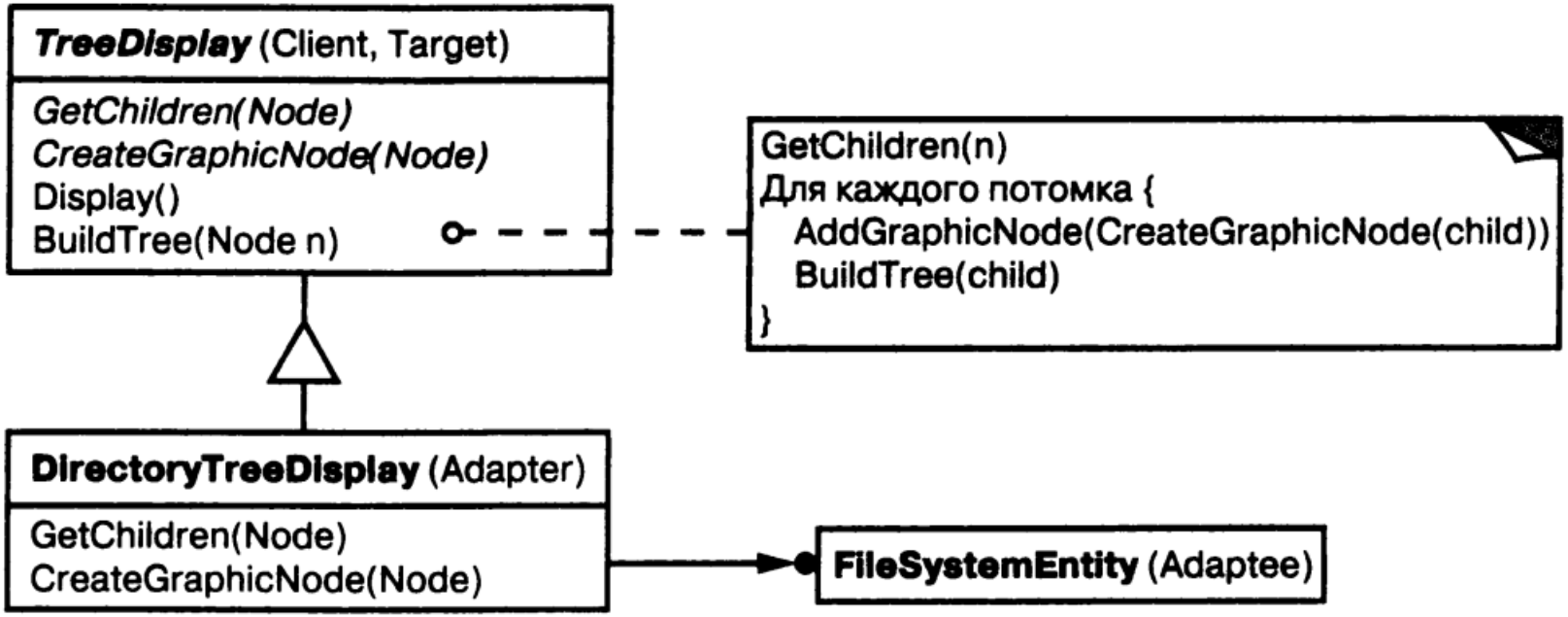
- Назначение:
  - Преобразует интерфейс одного класса в интерфейс другого, который ожидают клиенты. Адаптер обеспечивает совместную работу классов с несовместимыми интерфейсами, которая без него была бы невозможна.
- Применимость:
  - Если хотите использовать существующий класс, но его интерфейс не соответствует вашим потребностям;
  - Если собираетесь создать повторно используемый класс, который должен взаимодействовать с заранее неизвестными или не связанными с ним классами, имеющими несовместимые интерфейсы;
  - если нужно использовать несколько существующих подклассов, но непрактично адаптировать их интерфейсы путем порождения новых подклассов от каждого. В этом случае адаптер объектов может приспосабливать интерфейс их общего родительского класса.

Адаптер - структура





Адаптер - пример



Адаптер - примечания

- Клиенты вызывают операции Адаптера, а он транслирует их Адаптируемому, который и выполняет запрос, возвращая результат через Адаптер.
- Адаптер обычно применяется для изменения интерфейса существующего объекта, т.е. применяется на поздней стадии проектирования или при доработке существующей системы (в отличие от Моста).

Адаптер – пример

```
class Rectangle // Желаемый интерфейс
{
  public:
    virtual void draw() = 0;
};

//Хотим сделать так:
int main()
{
  Rectangle *r = new Rectangle (120, 200, 60, 40);  //startX, startY, length, width
  r->draw(); //Но реализации у нас нет!
}
```

Адаптер – пример

```
class LegacyRectangle // А есть вот такая старая реализация
{
  public:
    LegacyRectangle(Coordinate x1, Coordinate y1, Coordinate x2, Coordinate y2)
    {
      x1_ = x1; y1_ = y1;  x2_ = x2;  y2_ = y2;
      cout<<"LegacyRect: create. ("<< x1_<<","<<y1_<<")=>(“<<x2_<<","<<y2_<<")"<<endl;
    }
    void oldDraw()
    {
      cout<<"LegacyRect: oldDraw. ("<< x1_<<","<<y1_<<")=>("<<x2_<<","<<y2_<<")"<<endl;
    }
  private:
    Coordinate x1_;  y1_;  x2_;  y2_;
};
```

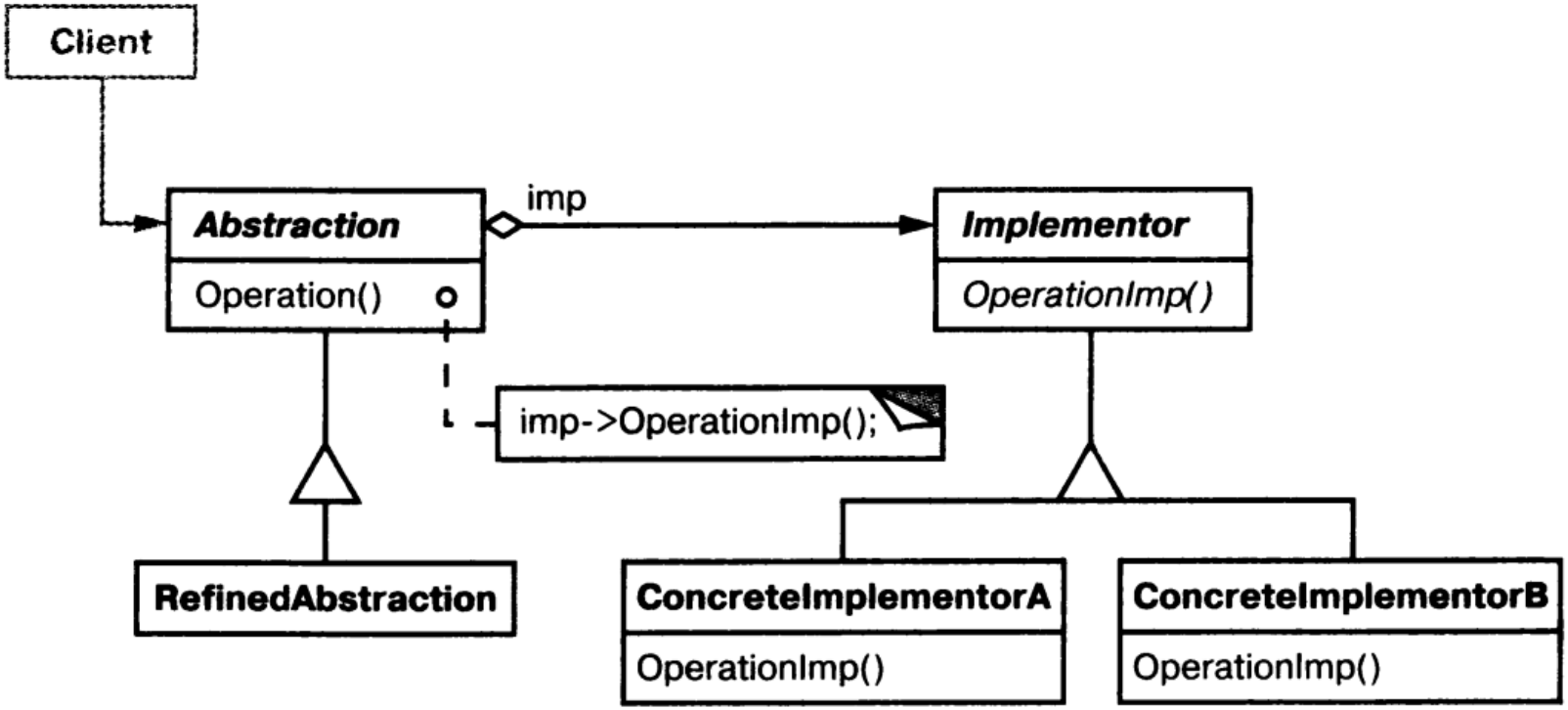
Адаптер – пример

```
class RectangleAdapter: public Rectangle, private LegacyRectangle { // Адаптер
  public:
    RectangleAdapter(Coordinate x, Coordinate y, Dimension w, Dimension h):
      LegacyRectangle(x, y, x + w, y + h) {
        cout << "RectAdapter: create.("<<x<<","<<y<<"),width="<<w<<",height ="<<h<< endl;
    }
    virtual void draw()  {
        cout << "RectangleAdapter: draw." << endl;
        oldDraw();
    }
};

int main() {
  Rectangle *r = new RectangleAdapter(120, 200, 60, 40);
  r->draw();
}
```

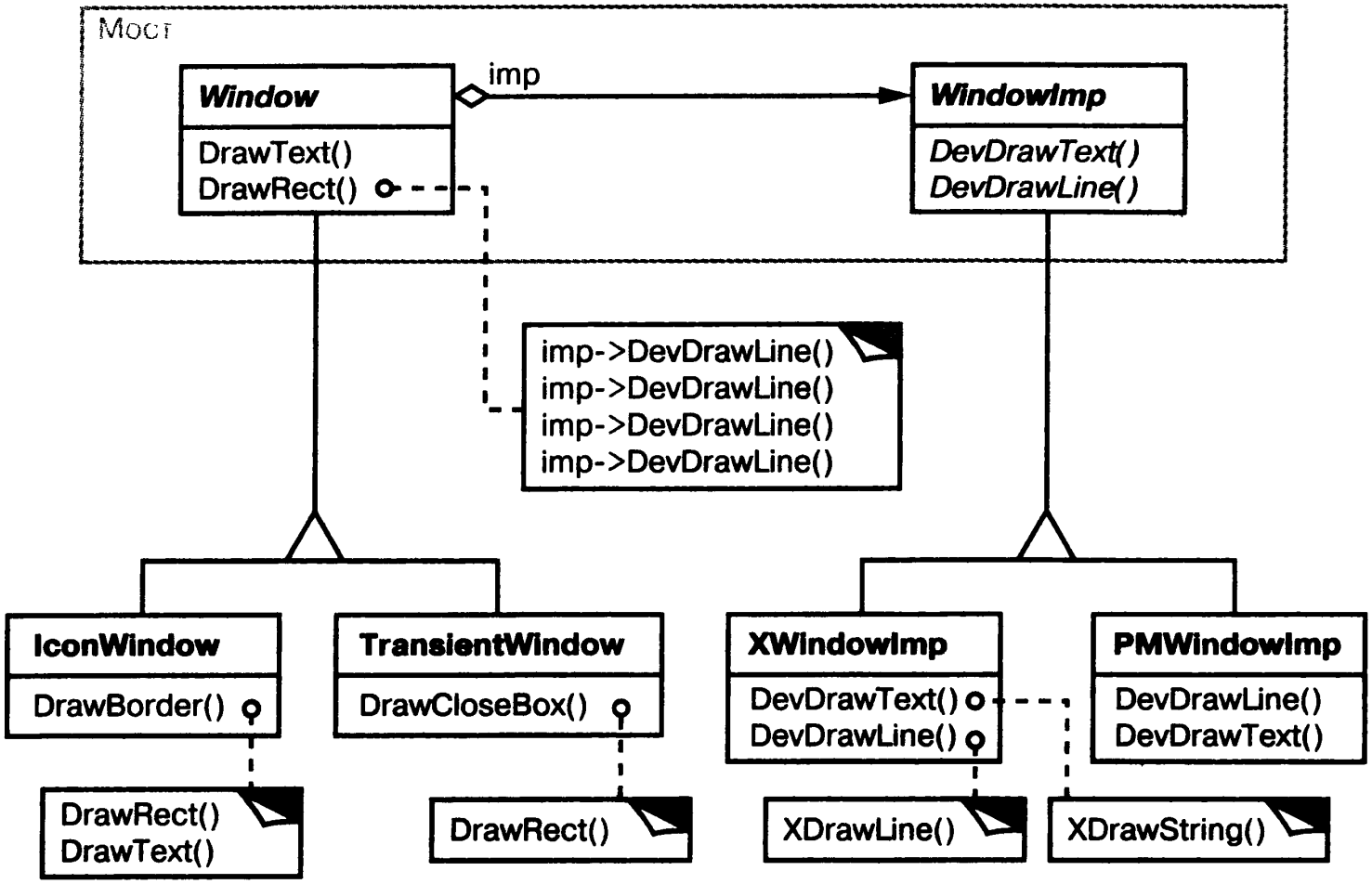
Мост

- Назначение:
  - Отделить абстракцию от ее реализации так, чтобы то и другое можно было изменять независимо
- Применимость:
  - Если хотите избежать постоянной привязки абстракции к реализации. Так, например, бывает, когда реализацию необходимо выбирать во время выполнения программы
  - Если и абстракции, и реализации должны расширяться новыми подклассами. В таком случае паттерн мост позволяет комбинировать разные абстракции и реализации и изменять их независимо
  - Если изменения в реализации абстракции не должны сказываться на клиентах, то есть клиентский код не должен перекомпилироваться
  - Если число классов начинает быстро расти (что создаёт проблему). Это признак того, что иерархию следует разделить на две части.
  - Если вы хотите разделить одну реализацию между несколькими объектами, и этот факт необходимо скрыть от клиента.

Мост - структура



Мост - пример


Мост
Window
DrawText()
DrawRect()
imp
WindowImp
DevDrawText()
DevDrawLine()
imp->DevDrawLine()
imp->DevDrawLine()
imp->DevDrawLine()
imp->DevDrawLine()
IconWindow
DrawBorder()
TransientWindow
DrawCloseBox()
XWindowImp
DevDrawText()
DevDrawLine()
PMWindowImp
DevDrawLine()
DevDrawText()
DrawRect()
DrawText()
DrawRect()
XDrawLine()
XDrawString()

Мост - примечания

- Абстракция транслирует запросы клиента Реализации.
- Удобно конструировать объекты, реализующие шаблон Мост, с помощью Абстрактной Фабрики.
- Мост обычно применяется на ранней стадии проектирования, закладываясь в «фундамент» системы.

Мост – пример

```
class UnitImpl //базовый класс реализации юнитов
{
  public:
    UnitImpl (int n) { id = n; x = 0; y = 0; }
    virtual void draw() = 0;
    virtual void move(int dx, int dy) {
        x += dx;
        y += dy;
    }
  protected:
    int id;
    int x,y;
};
```

Мост – пример

```
class WarriorImpl : public UnitImpl
{
  public:
    WarriorImpl(int n) : UnitImpl (n) {}
    void draw() { cout << " Warrior " << id << ": draw at " <<x<<","<<y<< endl; }
};

class RiflemanImpl : public UnitImpl
{
  public:
    RiflemanImpl(int n) : UnitImpl (n) {}
    void draw() { cout << " Rifleman " << id << ": draw at " <<x<<","<<y<< endl; }
};
```

Мост – пример

```
class Unit
{
   UnitImpl* unitImpl;
   public:
    Unit(UnitImpl* impl) {unitImpl = impl;}
    void move(char direction, int steps)
    {
       for(int i=0; i<steps; i++)
          switch (direction){
             case 'N':
               unitImpl->move(0, 1);
               unitImpl->draw();
               break;
             case 'S':
               unitImpl->move(0, -1);
               unitImpl->draw();
               break;
             case 'W':
               unitImpl->move(-1, 0);
               unitImpl->draw();
               break;
             case 'E':
               unitImpl->move(1, 0);
               unitImpl->draw();
               break;
          }
    }
};
```

Мост – пример

```
class PatrolUnit : public Unit
{
   public:
   PatrolUnit(UnitImpl* impl) : Unit(impl) {}
   void patrol(int times, int size)
   {
      for(int i=0; i<times; i++)
      {
         move('N', size);
         move('E', size);
         move('S', size);
         move('W', size);
      }
   }
};
```

```
class AttackUnit : public Unit
{
   public:
   AttackUnit(UnitImpl* impl) : Unit(impl) {}
   void attack(int distance, char direction)
   {
      move(direction, distance);
   }
};
```

Мост – пример

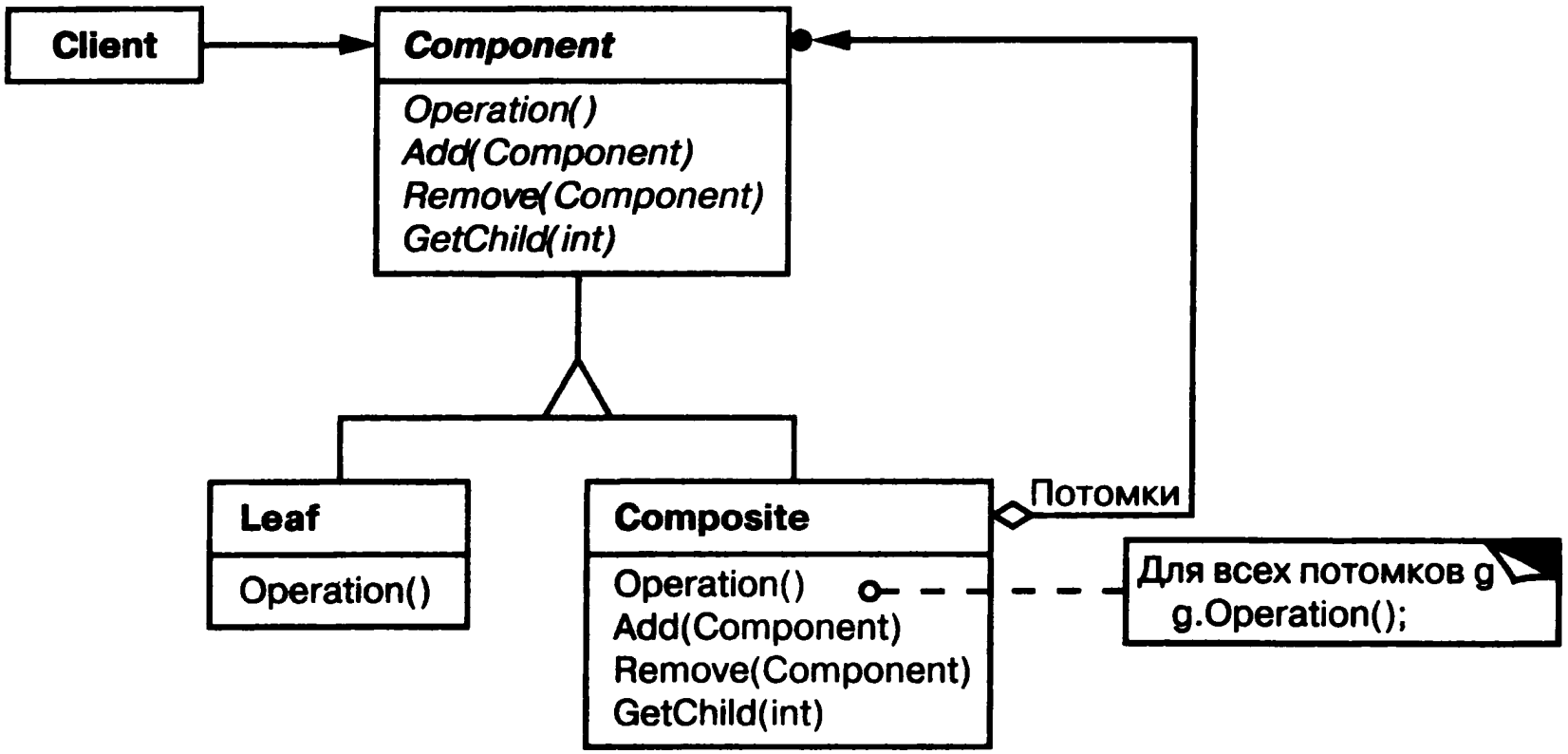
```
int main()
{
    PatrolUnit* patroller = new PatrolUnit(new WarriorImpl(1));
    AttackUnit* attacker1 = new AttackUnit(new RiflemanImpl(2));
    AttackUnit* attacker2 = new AttackUnit(new WarriorImpl(3));

    patroller->patrol(1, 3);
    attacker1->attack(5, 'N');
    attacker2->attack(7, 'E');

    return 0;
}
```

```
Warrior 1: draw at 0,1
Warrior 1: draw at 0,2
Warrior 1: draw at 0,3
Warrior 1: draw at 1,3
Warrior 1: draw at 2,3
Warrior 1: draw at 3,3
Warrior 1: draw at 3,2
Warrior 1: draw at 3,1
Warrior 1: draw at 3,0
Warrior 1: draw at 2,0
Warrior 1: draw at 1,0
Warrior 1: draw at 0,0
Rifleman 2: draw at 0,1
Rifleman 2: draw at 0,2
Rifleman 2: draw at 0,3
Rifleman 2: draw at 0,4
Rifleman 2: draw at 0,5
Warrior 3: draw at 1,0
Warrior 3: draw at 2,0
Warrior 3: draw at 3,0
Warrior 3: draw at 4,0
Warrior 3: draw at 5,0
Warrior 3: draw at 6,0
Warrior 3: draw at 7,0
```

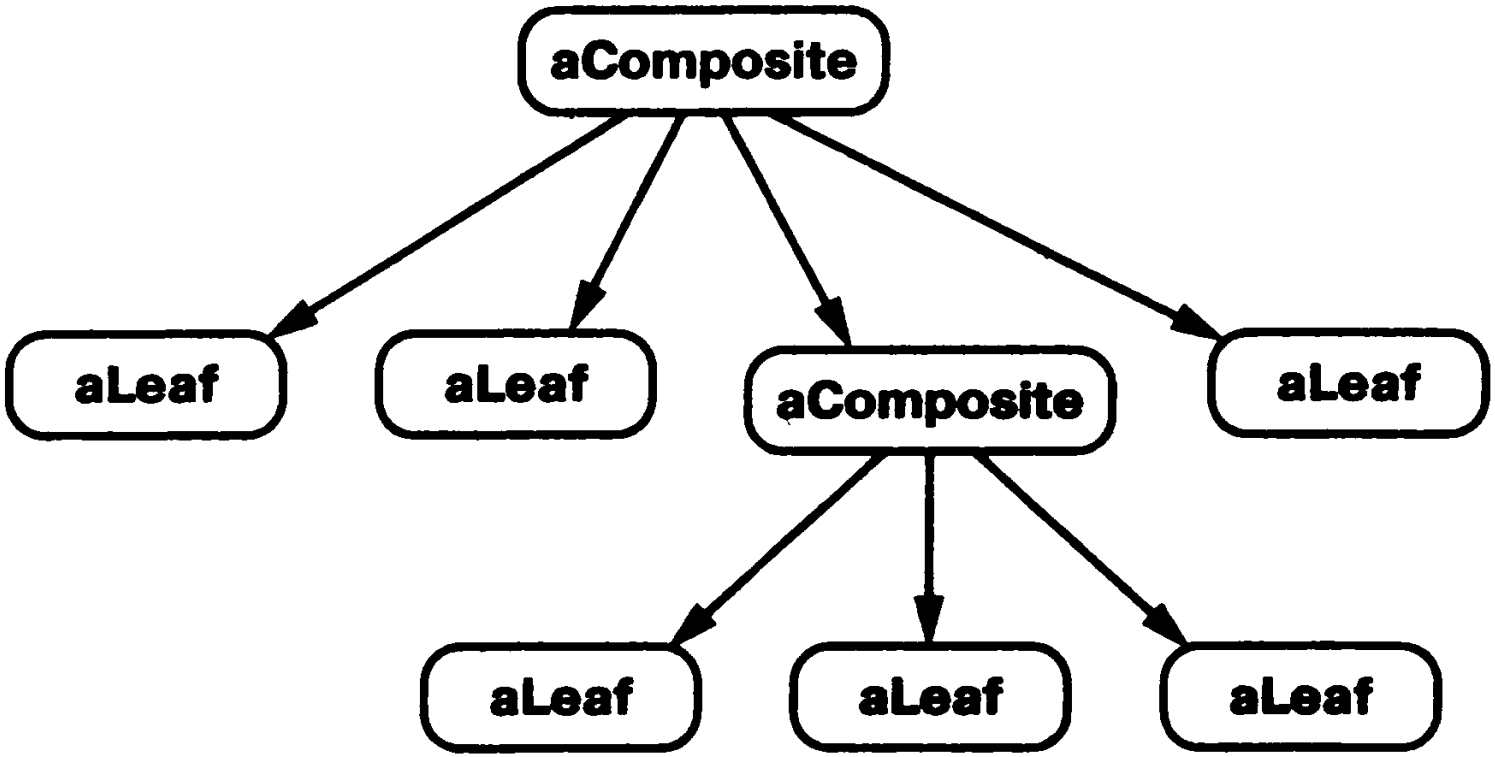
Компоновщик (Composite)

- Назначение:
  - Объединить объекты в древовидную структуру для представления иерархии от частного к целому.
- Применимость:
  - Если нужно представить иерархию объектов вида часть-целое
  - Если необходимо позволить клиентам обращаться к отдельным объектам и к группам объектов единообразно

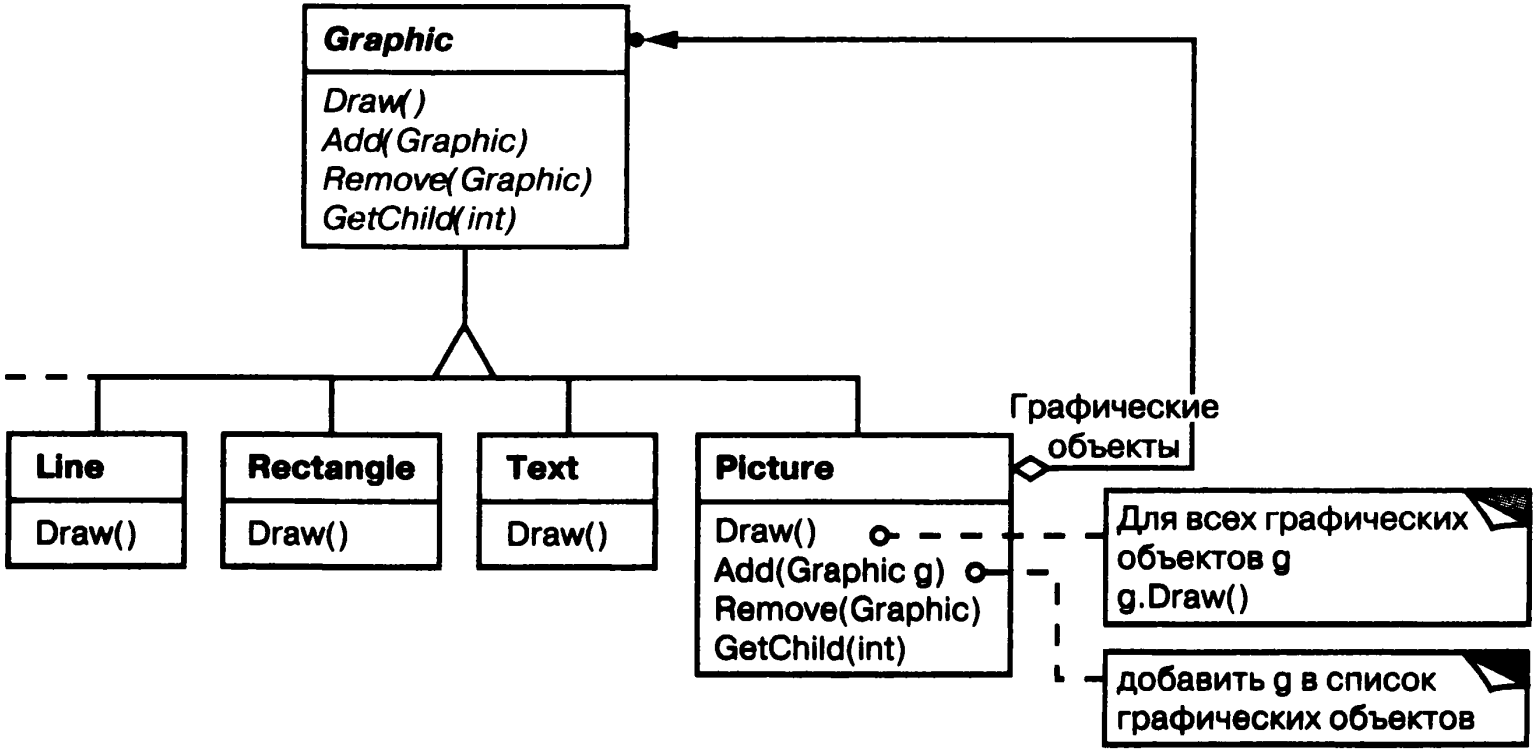
Компоновщик – структура классов



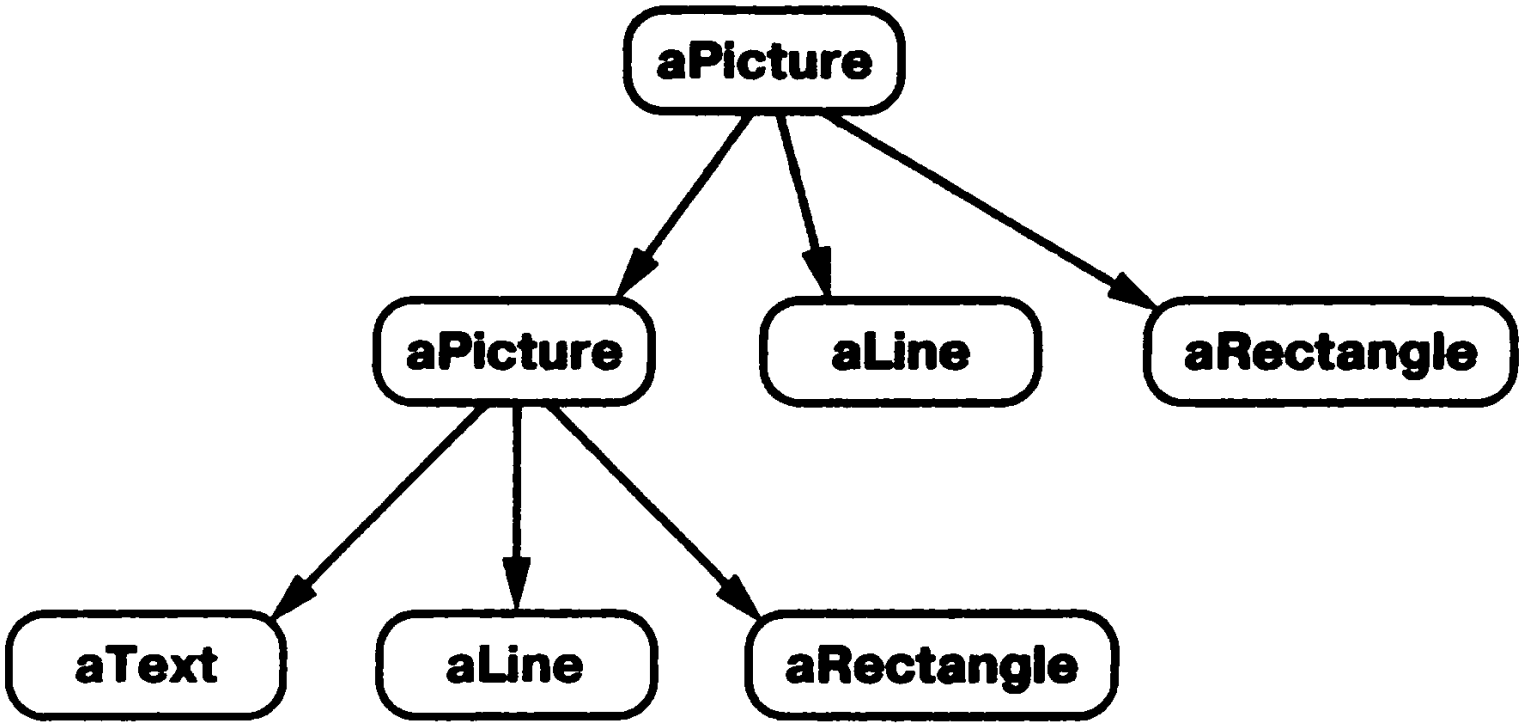
Компоновщик – структура объектов



Компоновщик - пример



Компоновщик - пример



Компоновщик - примечания

- Клиент используют интерфейс Component как для взаимодействия со всей структурой в целом, так и с ее отдельными составляющими.
- Если получателем является листовой объект Leaf, он и обрабатывает запрос.
- Если получатель – составной объект Composite, он передает запрос на обработку своим составляющим, возможно, добавляя что-либо от себя.

Компоновщик – пример

```
class Component{
  public:
    virtual void traverse() = 0;
};

class Leaf: public Component
{
    int value;
  public:
    Leaf(int val) { value = val;}
    void traverse() { cout << value << ' '; }
};
```

Компоновщик – пример

```
class Composite: public Component
{
    vector < Component * > children;
  public:
    void add(Component *ele) {
        children.push_back(ele);
    }
    void traverse() {
        for (int i = 0; i < children.size(); i++)
          children[i]->traverse();
    }
};
```

Компоновщик – пример

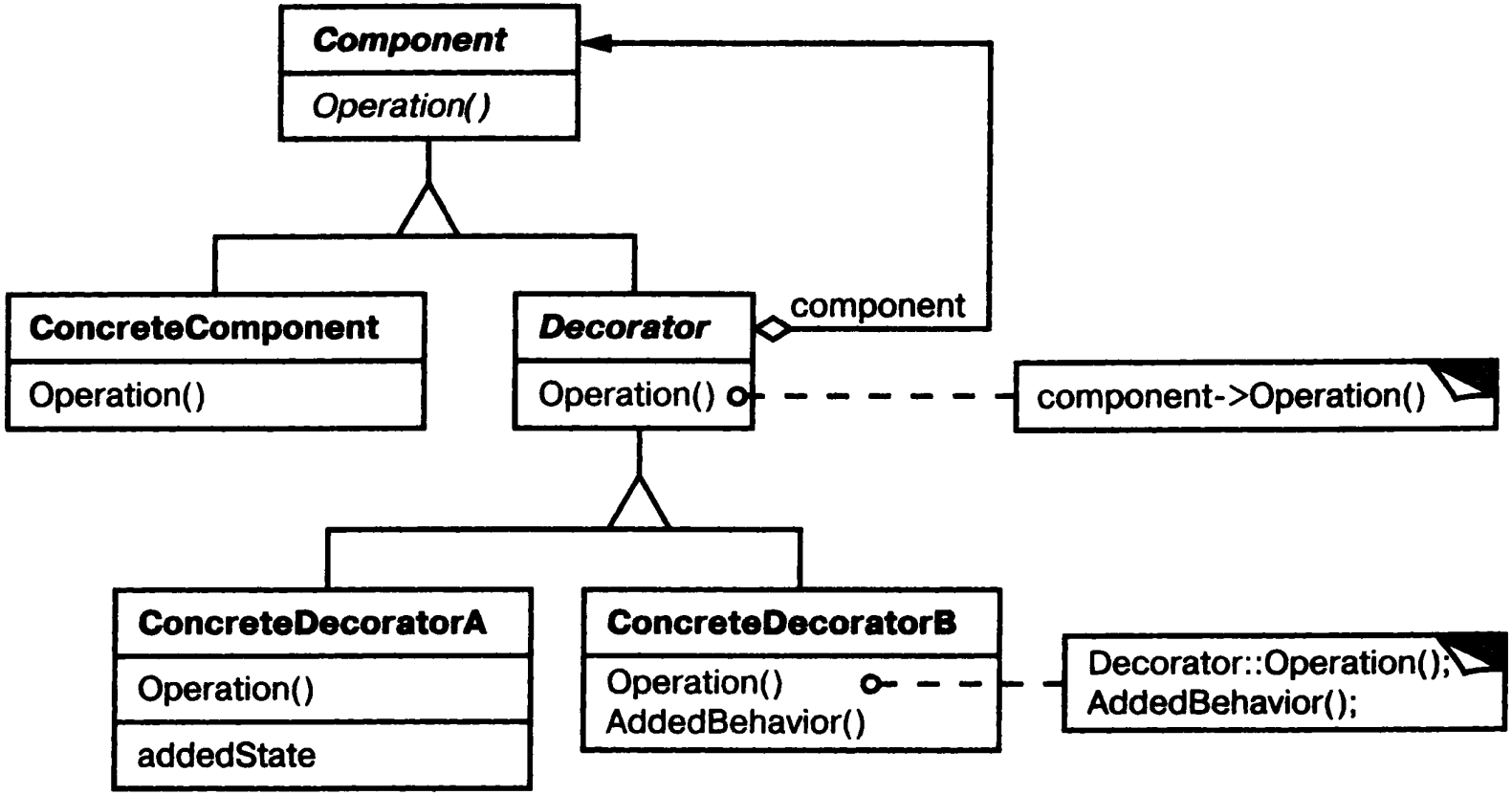
```
int main()
{
  Composite containers[4];

  for (int i = 0; i < 4; i++)
    for (int j = 0; j < 3; j++)
      containers[i].add(new Leaf(i *3+j));

  for (i = 1; i < 4; i++)
    containers[0].add(&(containers[i]));

  for (i = 0; i < 4; i++)  {
    containers[i].traverse();
    cout << endl;
  }
}
```

```
0 1 2 3 4 5 6 7 8 9 10 11
3 4 5
6 7 8
9 10 11
```

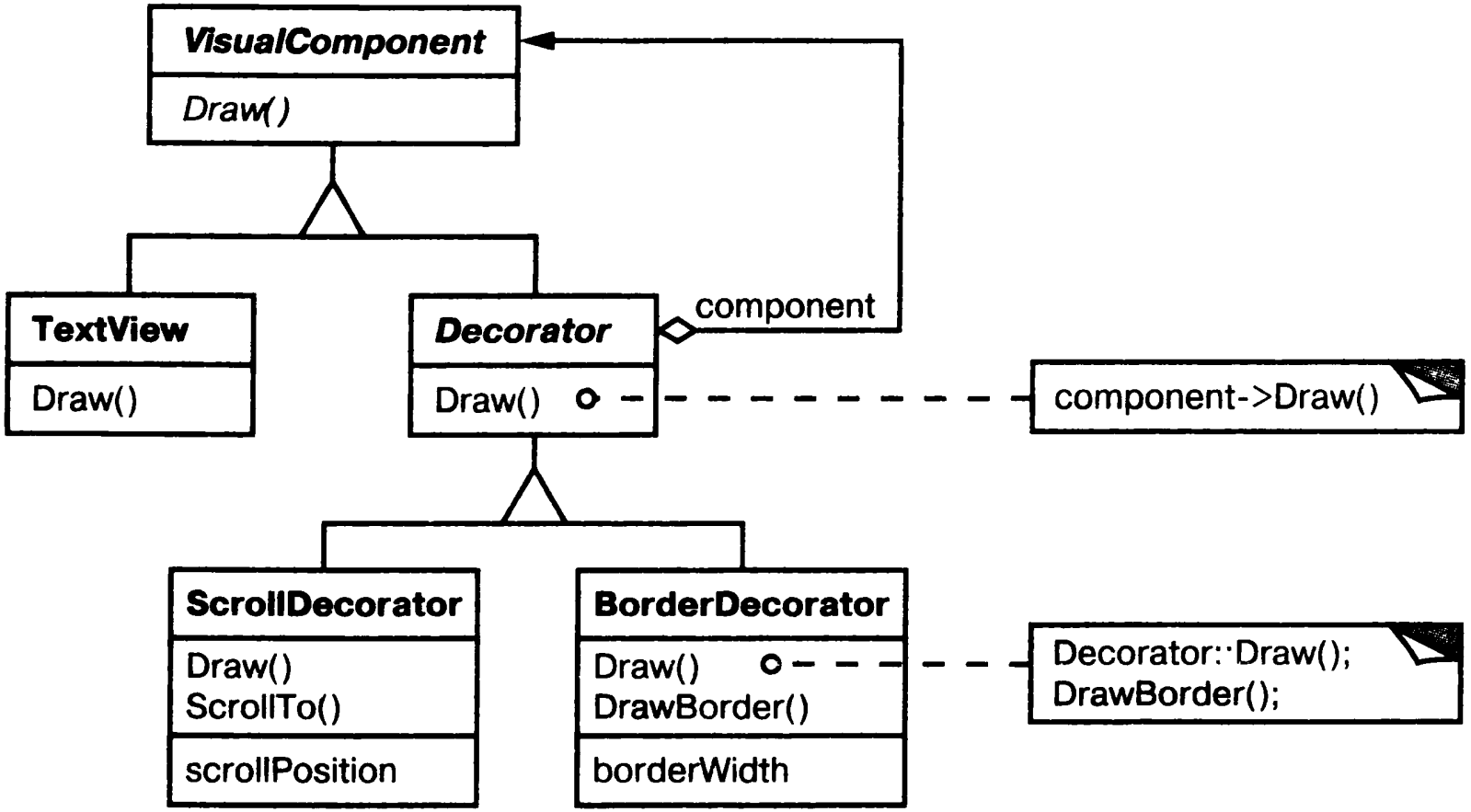
Декоратор

- Назначение:
  - Динамически добавлять объекту новые обязанности.
- Применимость:
  - Для динамического, прозрачного для клиента добавления обязанностей **объектам** (не классам!)
  - Для реализации обязанностей, которые могут быть сняты с объекта.
  - Когда наследование невозможно (sealed) или неудобно по каким-либо причинам.

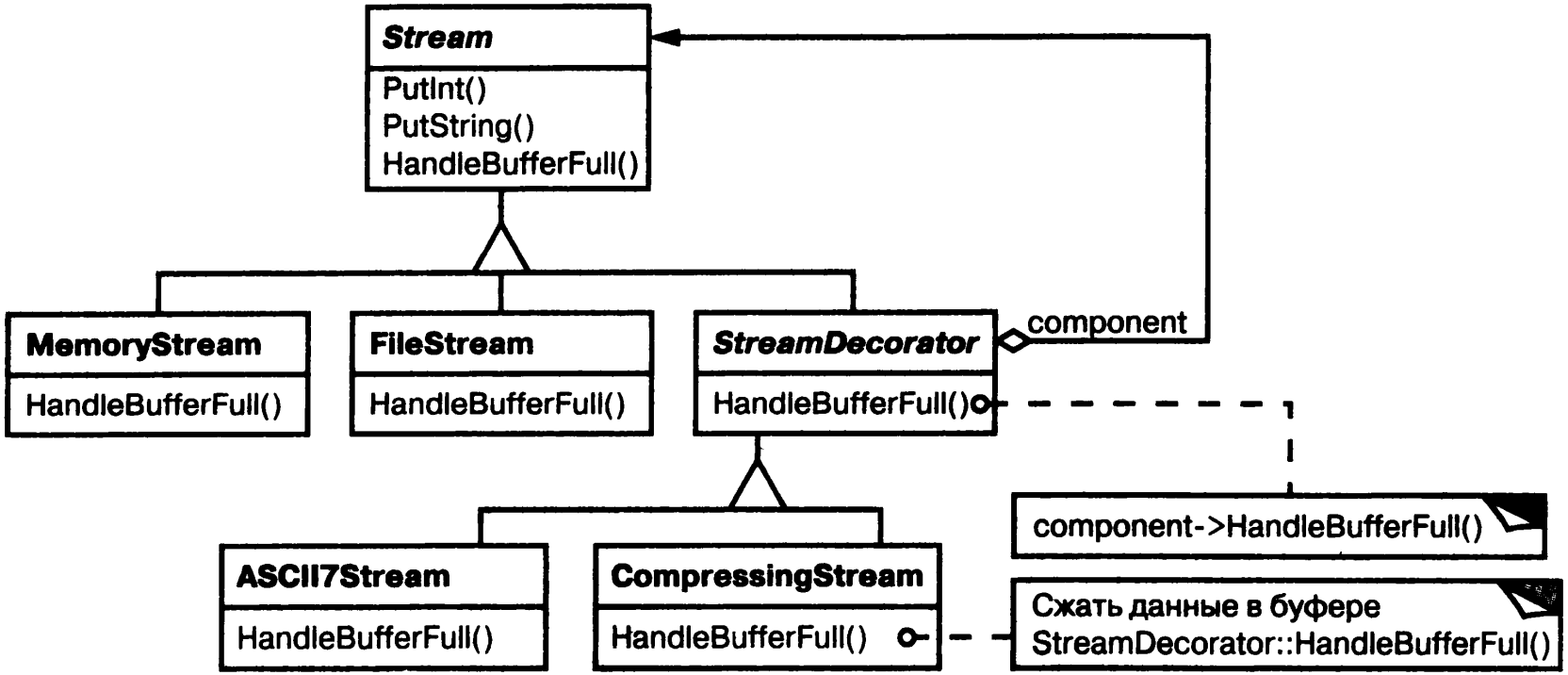
Декоратор - структура



Декоратор - пример



Декоратор - пример



Декоратор - примечания

- Под наследованием Декоратора от Компонента на диаграммах подразумевается исключительно наследование интерфейса. Доступ к функциональности компонента осуществляется через композицию, а не наследование.
- Декоратор переадресует запросы Компоненту, добавляя что-то от себя до или после вызова Компонента.
- В отличие от Адаптера сохраняет неизменным интерфейс объекта, меняя только поведение.

Компоновщик – пример

```
class Widget
{
  public:
    virtual void draw() = 0;
};
class TextField: public Widget
{
    int width, height;
  public:
    TextField(int w, int h) { width = w; height = h; }

    void draw()
    {
        cout << "TextField: " << width << ", " << height << '\n';
    }
};
```

Компоновщик – пример

```
class Decorator: public Widget
{
    Widget *wid;
  public:
    Decorator(Widget *w) { wid = w; }
     void draw() { wid->draw(); }
};
class BorderDecorator: public Decorator
{
  public:
    BorderDecorator(Widget *w): Decorator(w){}
    void draw() {
        Decorator::draw();
        cout << "   BorderDecorator" << '\n';
    }
};
```

Компоновщик – пример

```
class ScrollDecorator: public Decorator
{
  public:
     ScrollDecorator(Widget *w): Decorator(w){}
    void draw() {
       Decorator::draw();
       cout << "   ScrollDecorator" << '\n';
   }
};
int main()
{
Widget *aWidget = new BorderDecorator(new BorderDecorator(new ScrollDecorator
   (new TextField(80, 24))));
  aWidget->draw();
}
```

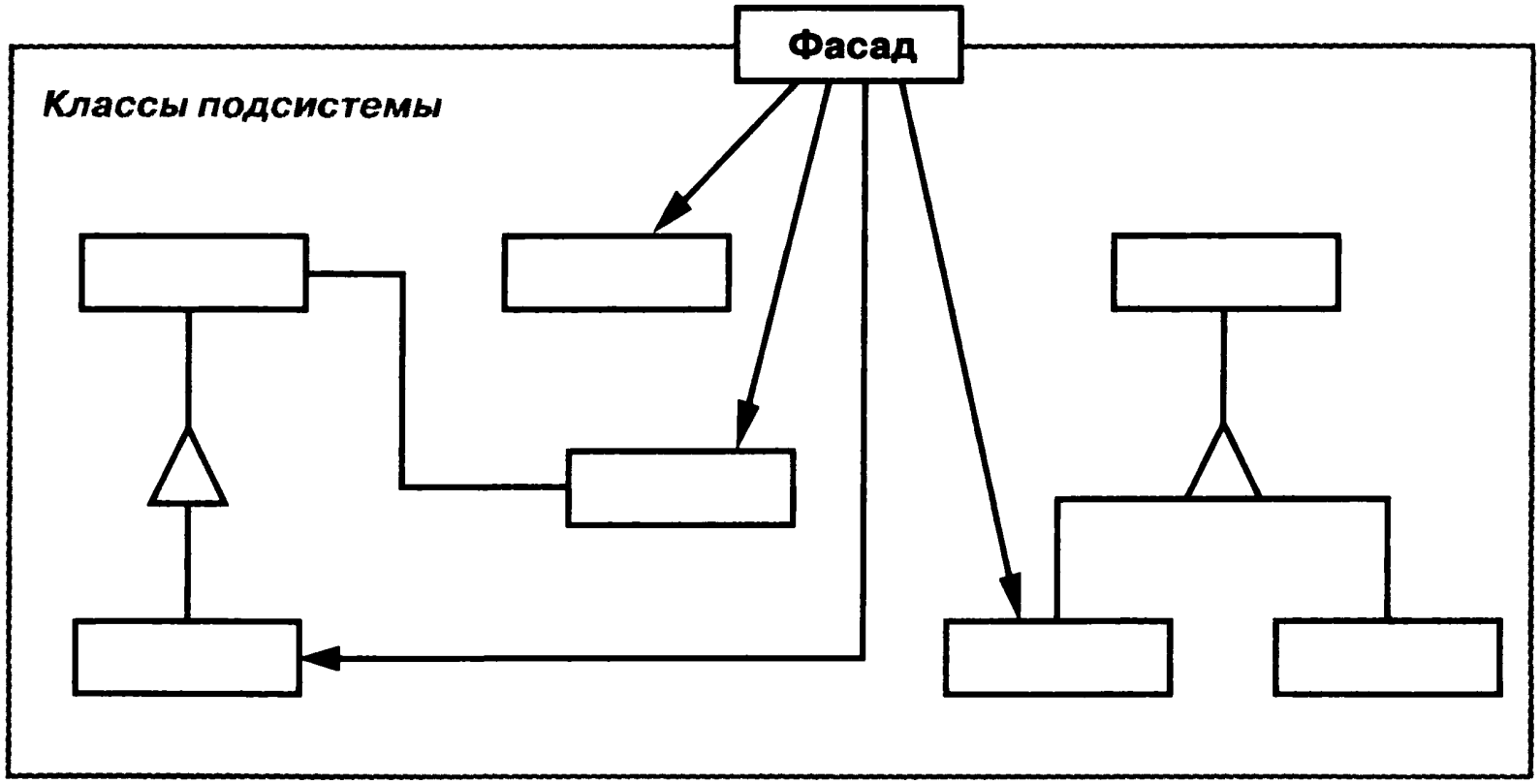
```
TextField: 80, 24
   ScrollDecorator
   BorderDecorator
   BorderDecorator
```

Фасад

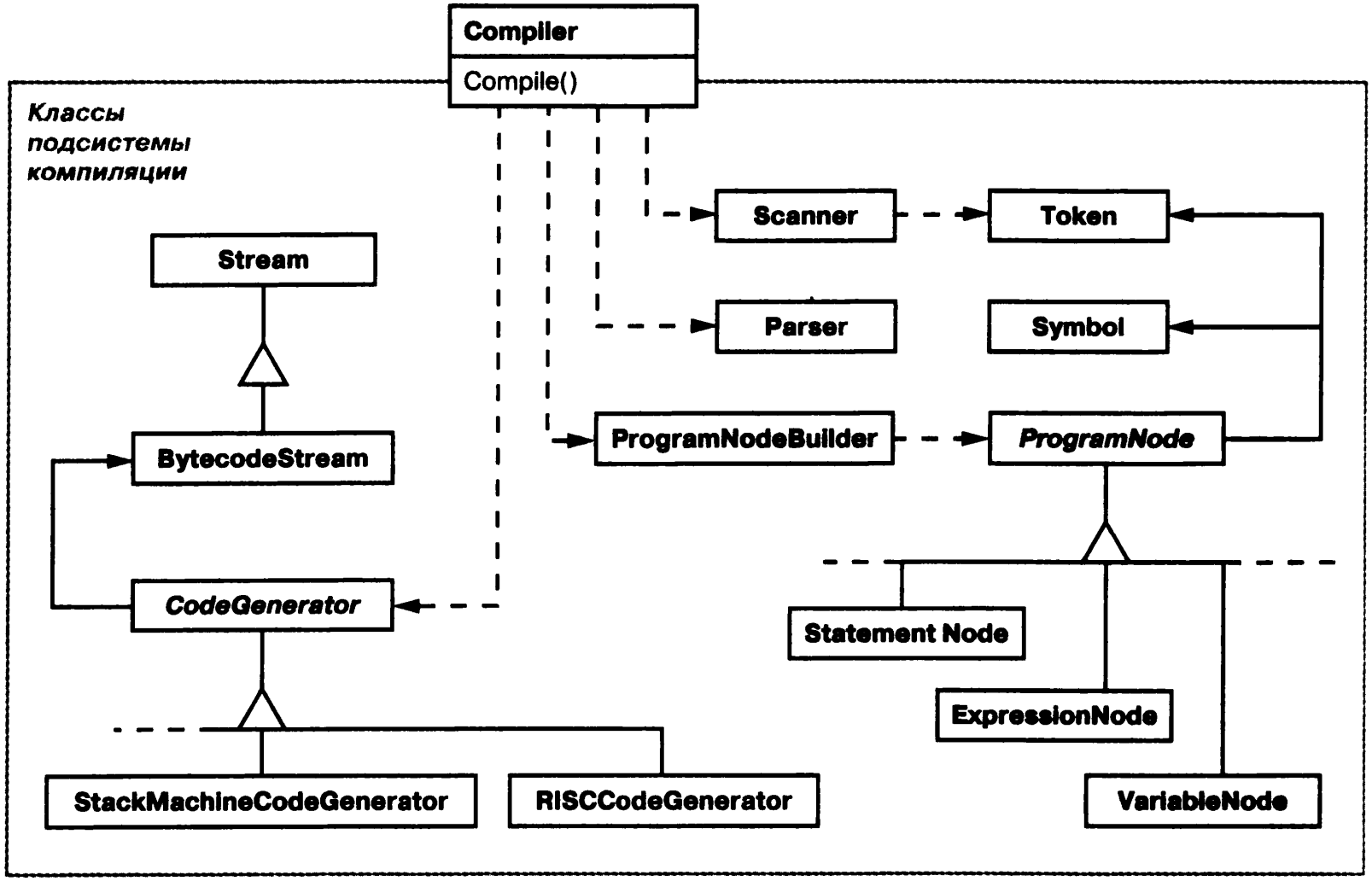
- Назначение:
  - скрыть сложность системы путем сведения всех возможных внешних вызовов к одному объекту, делегирующему их соответствующим объектам системы.
- Применимость:
  - Если хотим предоставить простой интерфейс к сложной системе
  - Если хотим уменьшить степень зависимости клиента от внутренней структуры системы
  - Если хотим разложить систему на отдельные слои

Фасад - структура

Фасад - структура



Фасад - пример



Фасад - примечания

- Клиенты общаются с подсистемой, посылая запросы Фасаду, который переадресует их нужным компонентам системы.
- Фасад может также осуществлять общее управление (“дирижирование”, orchestration) процессом исполнения запроса.
- Клиенты, пользующиеся фасадом, обычно не имеют доступа к компонентам подсистемы напрямую.

Фасад – пример

```
class Customer   // Клиент
{
   public:
   char Name[20];
   Customer(char* n) { strcpy(Name, n); }
};

class Bank   //Класс A в подсистеме
{
  public:
  bool HasSufficientSavings(Customer c, int amount) {
    cout << "Проверяем накопления " << c.Name << endl;
    return true;
  }
};
```

Фасад – пример

```
class Credit   //Класс Б в подсистеме
{
  public:
  bool HasGoodCredit(Customer c) {
    cout << "Проверяем кредит " << c.Name << endl;
    return true;
  }
};
class Loan //Класс В в подсистеме
{
  public:
  bool HasNoBadLoans(Customer c) {
    cout << "Проверяем долги " << c.Name << endl;
    return true;
  }
};
```

Фасад – пример

```
class Mortgage   // Facade
{
  Bank _bank;
  Loan _loan;
  Credit _credit;

  public:
  bool IsEligible(Customer cust, int amount)   {
    cout << cust.Name << " хочет получить " <<  amount << " денег в кредит" << endl;
    bool eligible = true;
    if (!_bank.HasSufficientSavings(cust, amount))   {   eligible = false;  }
    else if (!_loan.HasNoBadLoans(cust))   {  eligible = false;  }
    else if (!_credit.HasGoodCredit(cust))  {  eligible = false;  }
    return eligible;
  }
};
```

Фасад – пример

```
int main()
{
   Mortgage mortgage;
   Customer c("Вася");
   bool eligible = mortgage.IsEligible(c, 100500);
   cout << c.Name << " кредит " << (eligible ? "" : "не ") << "получил";
   return 0;
}
```

Вася хочет получить 100500 денег в кредит

Проверяем накопления Вася

Проверяем долги Вася

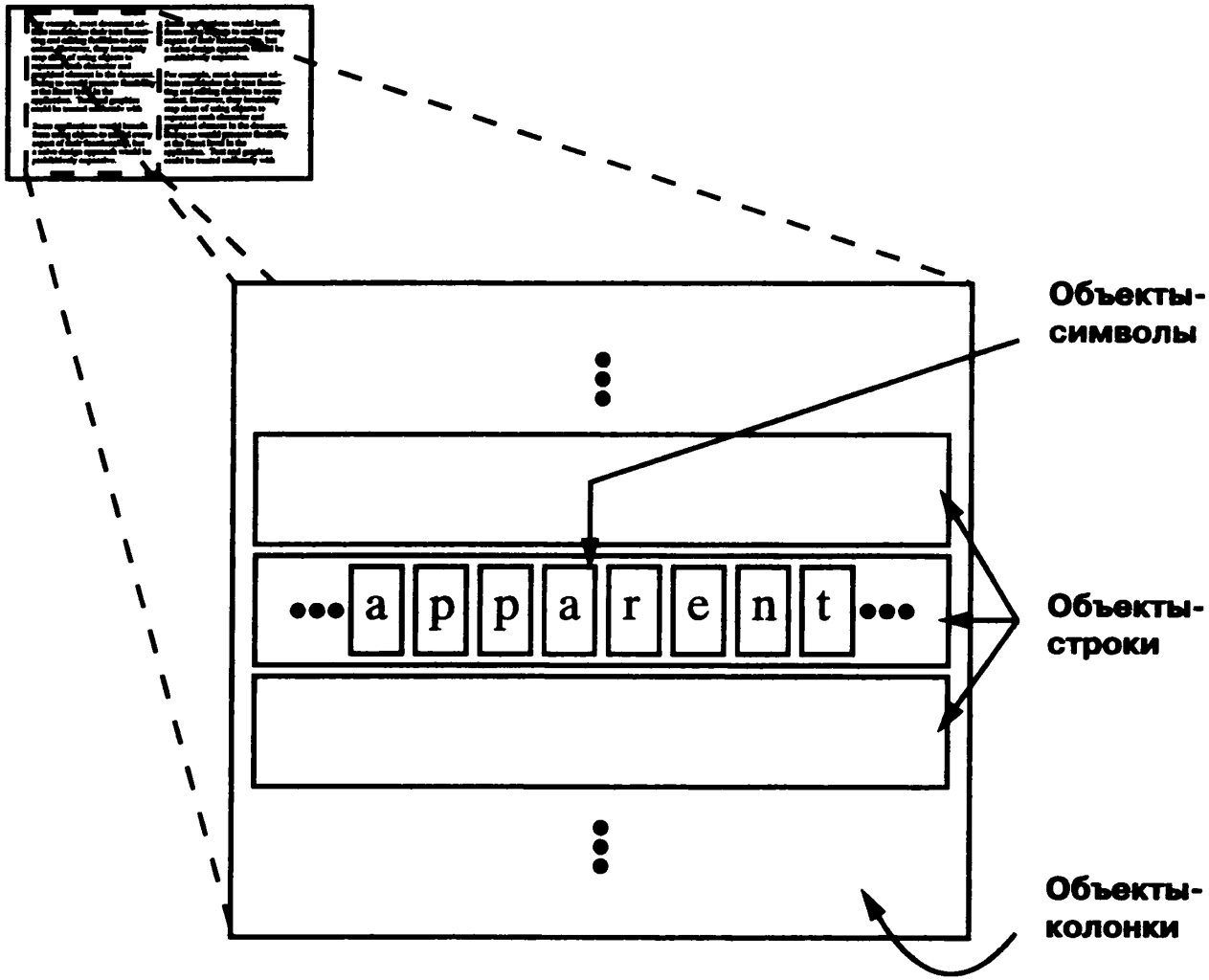
Проверяем кредит Вася

Вася кредит получил

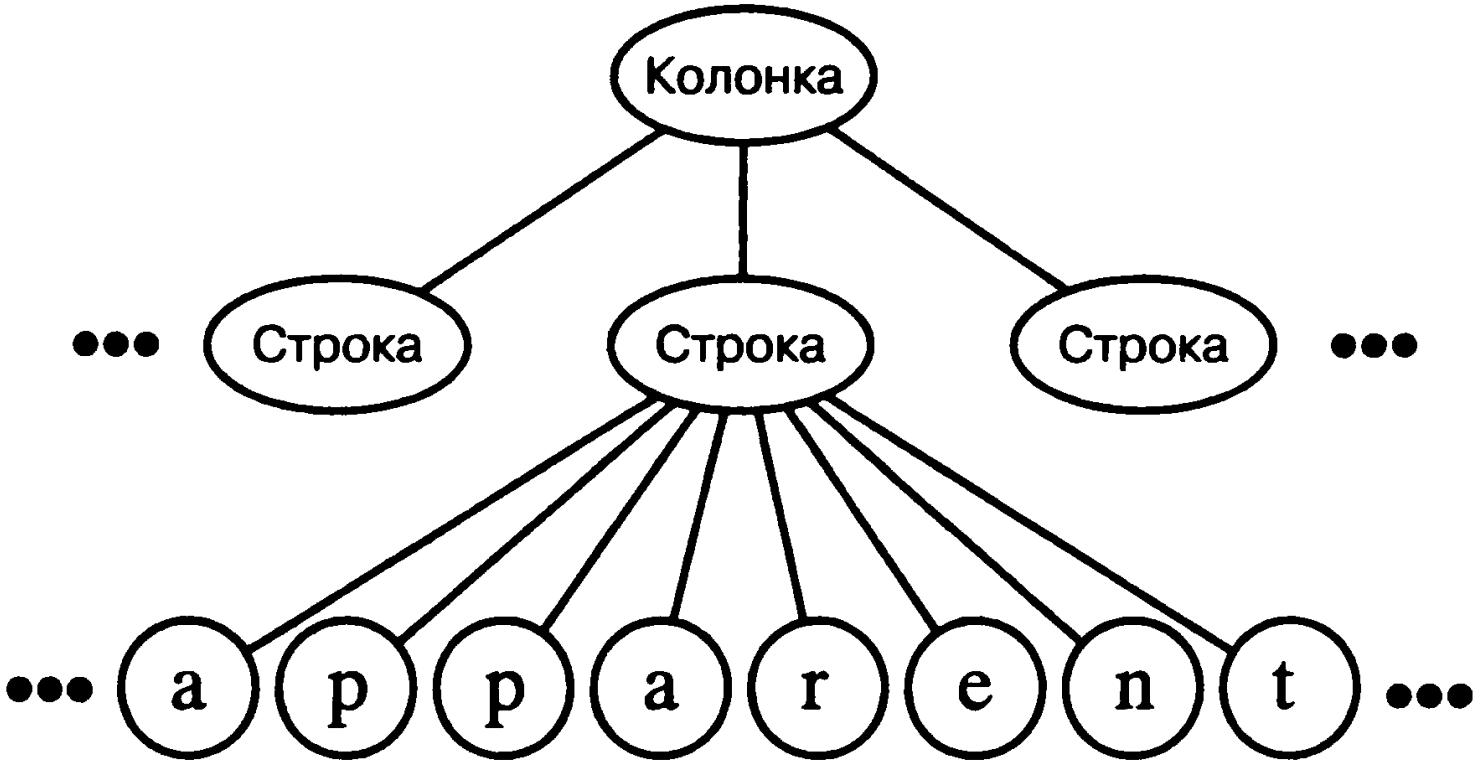
Flyweight (Приспособленец)

- Назначение:
  - Использовать разделяемые объекты для эффективной поддержки множества мелких объектов.
- Применимость (если выполняются все условия):
  - В приложении используется настолько большое число объектов, что накладные расходы на их хранение становятся чересчур высоки
  - Большую часть состояния объектов можно вынести вовне и заменить множество объектов ссылкой на один разделяемый объект, в который вынесено состояние
  - Идентичность объектов чаще всего не важна

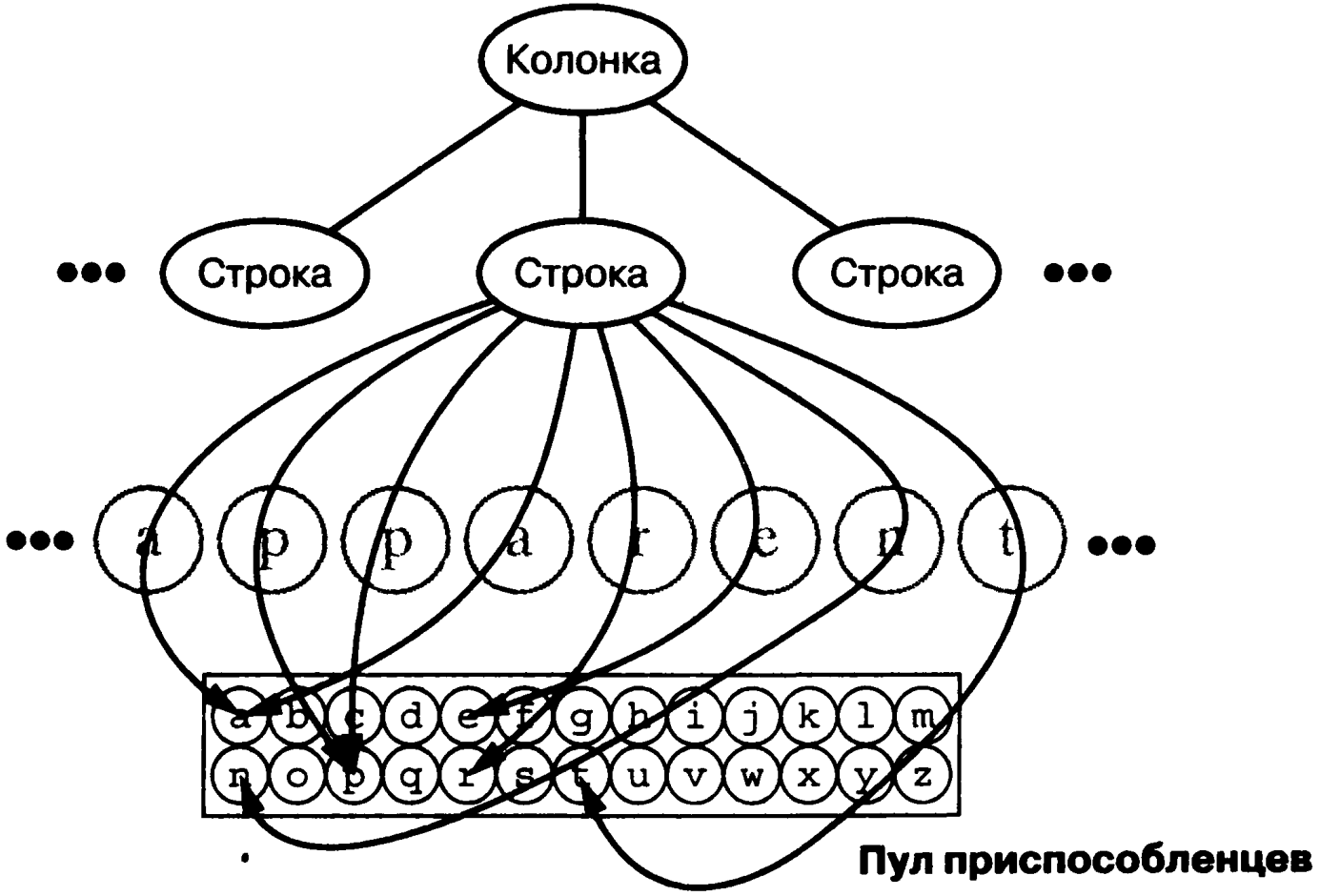
Flyweight - пример



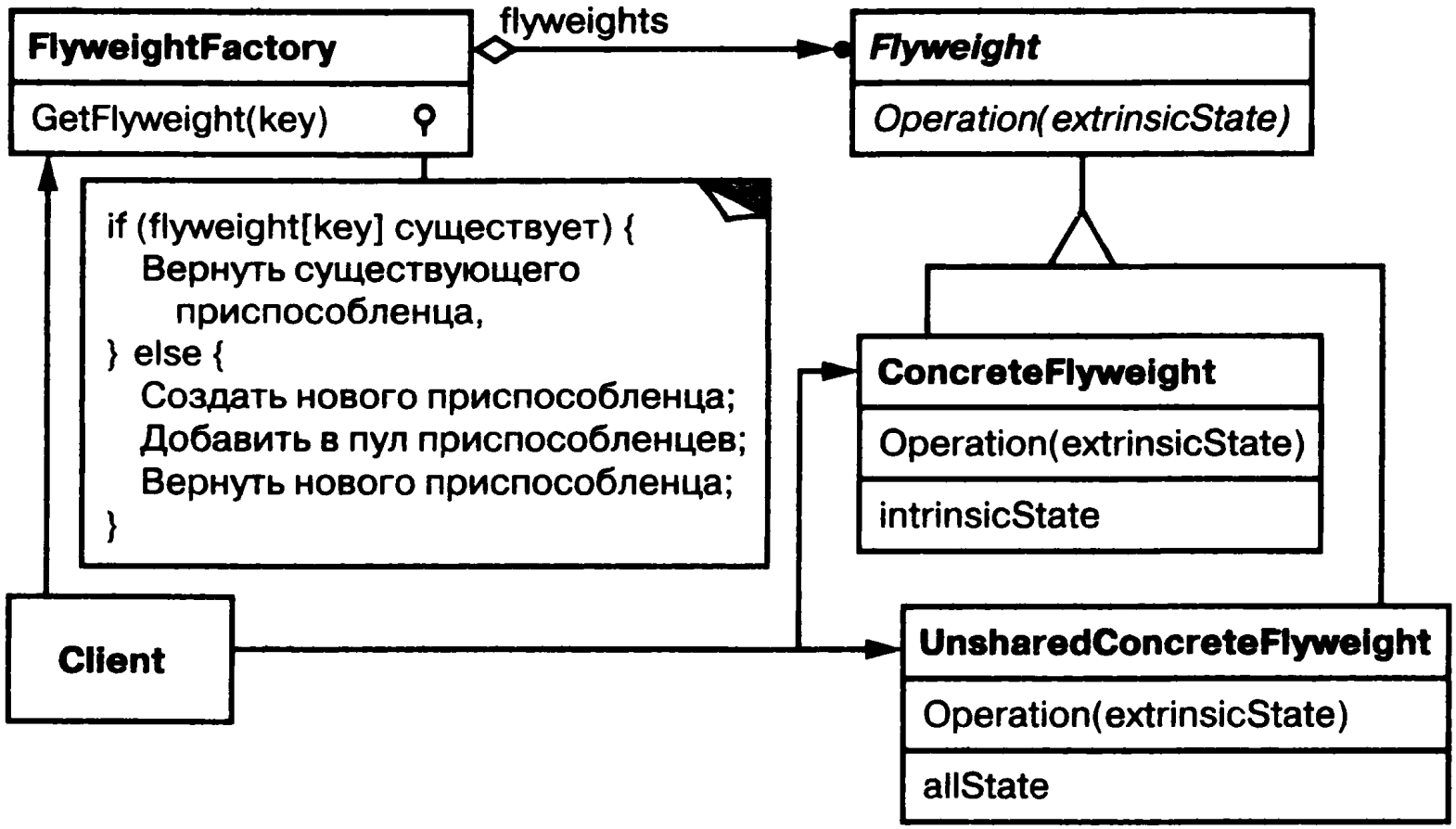
Flyweight - пример



Flyweight - пример



Flyweight - структура



Flyweight - примечания

- Состояние, необходимое Flyweight для работы делится на 2 части – внешнее и внутреннее. Внутреннее хранится в самом объекте, внешнее – передается клиентом.
- Клиенты не должны создавать экземпляры Flyweight напрямую, но только через Фабрику, что гарантирует корректное повторное использование объектов.

Flyweight – пример

```
class Icon
{
  public:
    Icon(char *fileName) {
        strcpy(_name, fileName); //имитация загрузки из файла
    }
    const char *getName() {return _name; }
    void draw(int x, int y) {
        cout << "   рисуем иконку " << _name << " на (" << x << "," << y <<  ")" << endl;
    }
  private:
    char _name[20];
    int _width;
    int _height;
};
```

Flyweight – пример

```
class FlyweightFactory {
 private:
   static int _numIcons;
   static Icon *_icons[5];
 public:
   static Icon *getIcon(char *name) {
      for (int i = 0; i < _numIcons; i++)
        if (!strcmp(name, _icons[i]->getName())) return _icons[i];
      _icons[_numIcons] = new Icon(name);
      return _icons[_numIcons++];
   }
   static void reportTheIcons() {
      cout << "Буфер приспособленцев: ";
      for (int i = 0; i < _numIcons; i++)  cout << _icons[i]->getName() << " ";
      cout << endl;
   }
};
```

Flyweight – пример

```
int FlyweightFactory::_numIcons = 0;
Icon *FlyweightFactory::_icons[];



class DialogBox
{
  public:
    DialogBox(int x, int y, int incr): _iconsOriginX(x), _iconsOriginY(y), _iconsXIncrement(incr){}
    virtual void draw() = 0;
  protected:
    Icon *_icons[3];
    int _iconsOriginX;
    int _iconsOriginY;
    int _iconsXIncrement;
};
```

Flyweight – пример

```
class FileSelection: public DialogBox
{
  public:
    FileSelection(Icon *first, Icon *second, Icon *third): DialogBox(100, 100, 100)
    {
        _icons[0] = first;
        _icons[1] = second;
        _icons[2] = third;
    }
    void draw()
    {
        cout << "отрисовка FileSelection:" << endl;
        for (int i = 0; i < 3; i++)
          _icons[i]->draw(_iconsOriginX + (i *_iconsXIncrement), _iconsOriginY);
    }
};
```

Flyweight – пример

```
class CommitTransaction: public DialogBox
{
  public:
    CommitTransaction(Icon *first, Icon *second, Icon *third): DialogBox(150, 150, 150)
    {
        _icons[0] = first;
        _icons[1] = second;
        _icons[2] = third;
    }
    void draw()
    {
        cout << " отрисовка CommitTransaction:" << endl;
        for (int i = 0; i < 3; i++)
          _icons[i]->draw(_iconsOriginX + (i *_iconsXIncrement), _iconsOriginY);
    }
};
```

Flyweight – пример

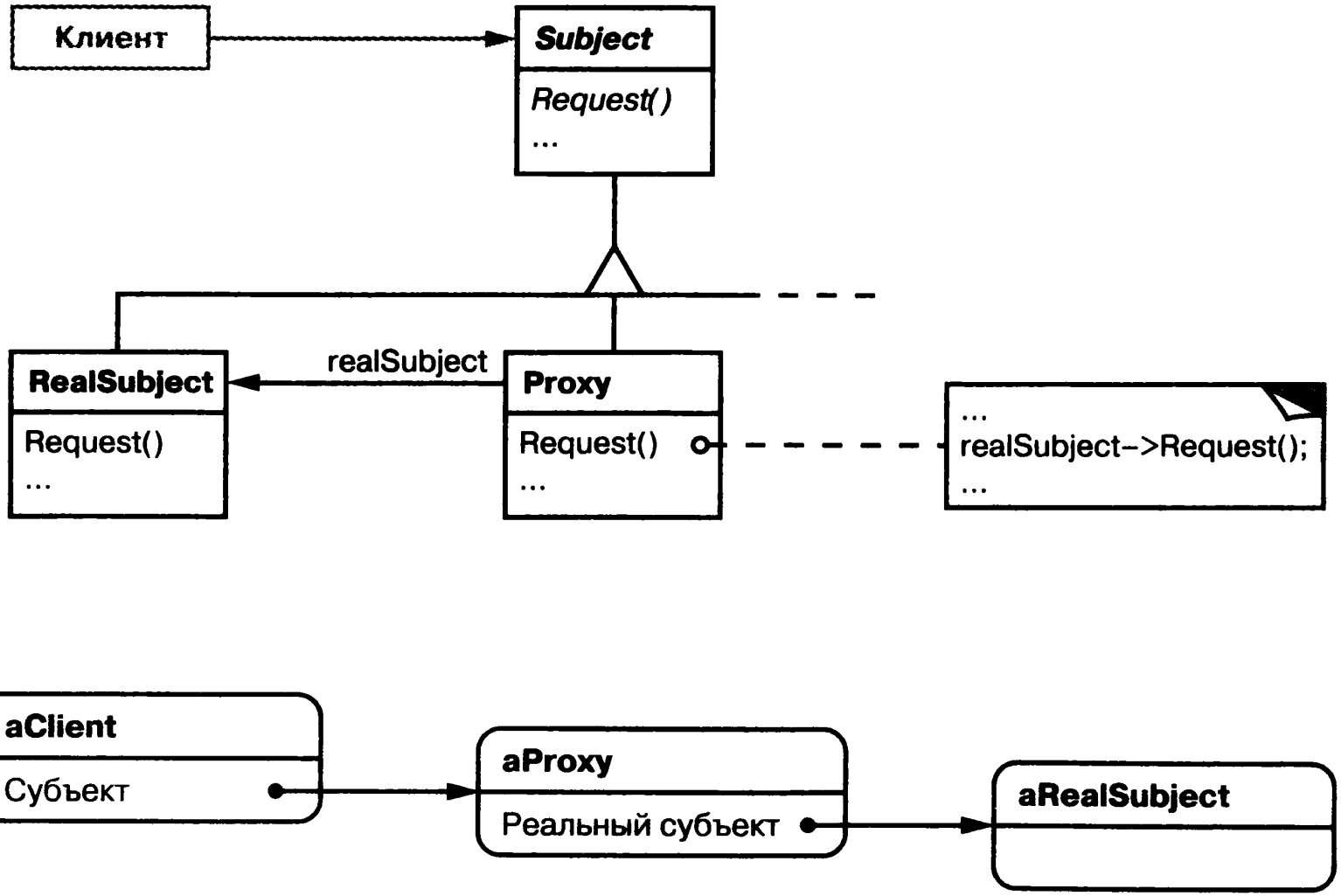
```
int main()
{
  DialogBox *dialogs[2];
  dialogs[0] = new FileSelection(FlyweightFactory::getIcon("go"),
    FlyweightFactory::getIcon("stop"), FlyweightFactory::getIcon("select"));
  dialogs[1] = new CommitTransaction(FlyweightFactory::getIcon("select"),
    FlyweightFactory::getIcon("stop"), FlyweightFactory::getIcon("undo"));

  for (int i = 0; i < 2; i++)
    dialogs[i]->draw();

  FlyweightFactory::reportTheIcons();
}
```

отрисовка FileSelection:
  рисуем иконку go на (100,100)
  рисуем иконку stop на (200,100)
  рисуем иконку select на (300,100)
отрисовка CommitTransaction:
  рисуем иконку select на (150,150)
  рисуем иконку stop на (300,150)
  рисуем иконку undo на (450,150)
Буфер приспособленцев: go stop select undo

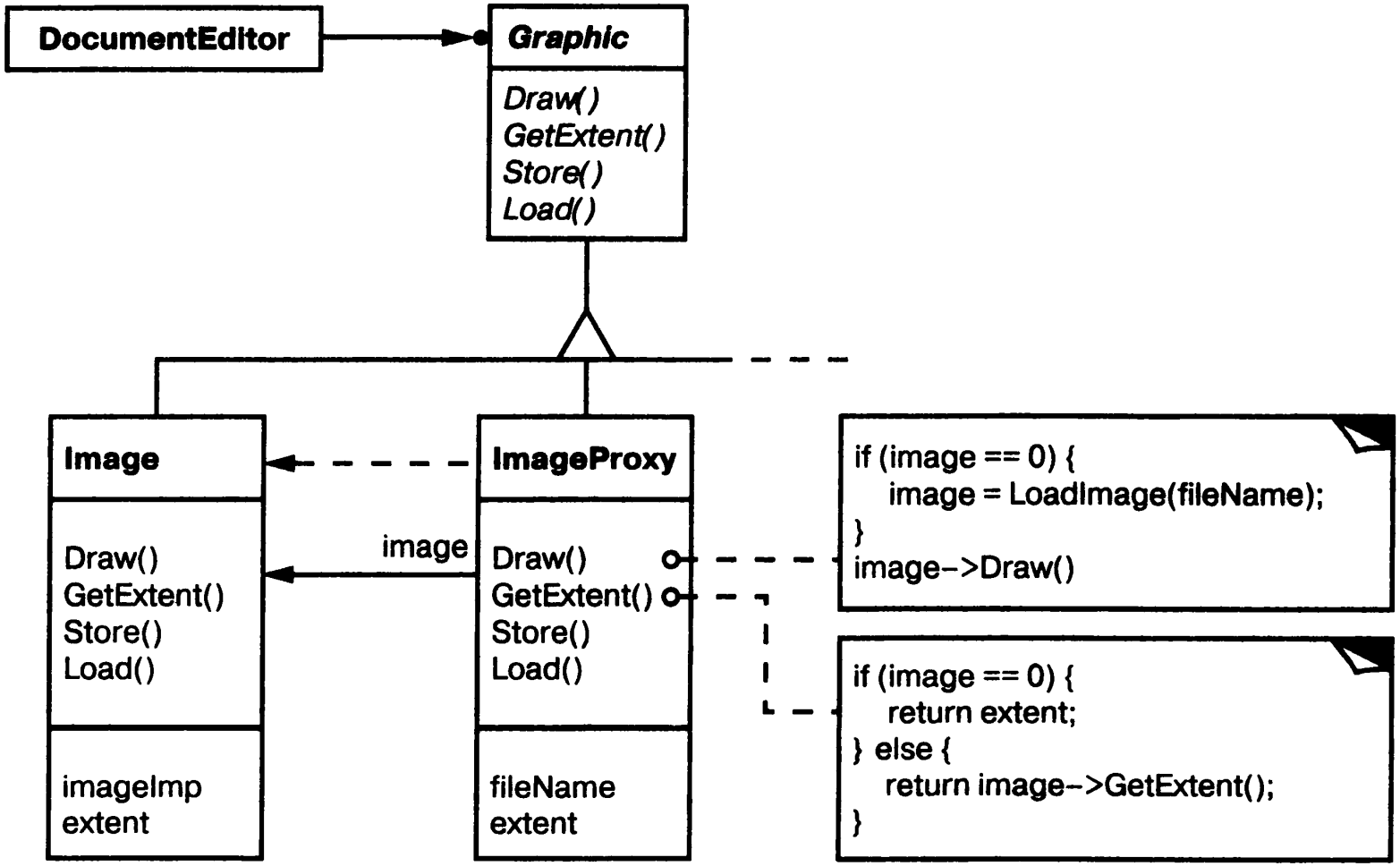
Заместитель (Proxy)

- Назначение:
  - Является заместителем другого объекта и контролирует доступ к нему
- Применимость:
  - Локальный заместитель удаленного объекта
  - Виртуальный заместитель «тяжелого» объекта
  - Защищающий заместитель, осуществляющий контроль доступа
  - «Интеллектуальный указатель», ведущий подсчет ссылок на объект

Заместитель - структура



Заместитель - пример



Заместитель - примечания

- Под наследованием Заместителя от Субъекта на диаграммах подразумевается исключительно наследование интерфейса. Доступ к функциональности Субъекта осуществляется через композицию, а не наследование.
- Заместитель либо отвечает на запросы сам, либо, если сам ответить не в состоянии, транслирует вызовы Субъекту.
- При этом Заместитель может выполнять дополнительную работу, контролировать доступ либо оптимизировать работу с Субъектом.
- В отличие от Декоратора, Клиент не имеет доступа к Субъекту и работает только с Заместителем.

Заместитель – пример

```
class IMath //Subject
{
   public:
     virtual long DoSomeTimeConsumingComputations(int x) = 0;
};

class Math : public IMath //RealSubject
{
   public:
   //притворимся, что тут тяжелые вычисления
     long DoSomeTimeConsumingComputations(int x) { return x * x; }
};
```

Заместитель – пример

```
class MathProxy : IMath //Proxy
{
   map<int, long> _cache;
   Math _math;
 public:
   long DoSomeTimeConsumingComputations(int x)
   {
      if (!(_cache.count(x)>0))
      {
         cout << "    Рассчитываю значение для " << x << endl;
         _cache[x] = _math.DoSomeTimeConsumingComputations(x);
      }
      else cout << "    Беру значение из кэша" << endl;;
      return _cache[x];
   }
};
```

Заместитель – пример

```
int main()
{
    MathProxy proxy; // Создаем кеширующий прокси
    cout << "F(1234) = " << proxy.DoSomeTimeConsumingComputations(1234)<< endl;
    cout << "F(456) = " << proxy.DoSomeTimeConsumingComputations(456)<< endl;
    cout << "F(1234) = " << proxy.DoSomeTimeConsumingComputations(1234)<< endl;
    return 0;
}
```

```
Рассчитываю значение для 1234
F(1234) = 1522756
    Рассчитываю значение для 456
F(456) = 207936
    Беру значение из кэша
F(1234) = 1522756
```

Резюме

- **Адаптер** помогает облегчить жизнь, когда система уже спроектирована, **Мост** – в процессе проектирования.
- **Мост** изначально проектируется для того, чтобы абстракция и реализация развивались независимо, **Адаптер** же нужен уже постфактум, для того, чтобы заставить несвязанные классы работать вместе.
- **Адаптер** дает субъекту новый интерфейс. **Заместитель** дает тот же интерфейс. **Декоратор** обеспечивает расширенный интерфейс.
- **Адаптер** изменяет интерфейс объекта, **Декоратор** расширяет его ответственность. Таким образом, **Декоратор** прозрачнее для клиента. Как следствие, декоратор поддерживает рекурсивную композицию, что невозможно с чистым **Адаптером**.
- **Компоновщик** и **Декоратор** имеют схожие структурные диаграммы, т.к. они оба полагаются на рекурсивную композицию чтобы организовать работу с неопределенным множеством объектов.
- **Компоновщик** может проходиться **Итератором**. **Посетитель** может применять операции к **Компоновщику**. **Компоновщик** может использовать **Цепочку Ответственности** чтобы дать компонентам доступ к свойствам более высокого уровня через своих родителей. Он также может использовать Декоратор чтобы перекрыть такие свойства на части композиции. Он может использовать **Наблюдателя** чтобы связать одну объектную структуру с другой и **Состояние** чтобы позволить компонентам менять их поведение при смене состояний.

Резюме

- **Компоновщик** может позволить компоновать Посредников из меньших компонентов через рекурсивную композицию.
- **Декоратор** позволяет менять внешность объекта. **Стратегия** позволяет менять его поведение.
- **Декоратор** позволяет добавлять обязанности объектам без наследования. **Компоновщик** – концентрируется не на добавлении нового, а на структурировании имеющегося. Эти цели различны, но взаимодополняющи, поэтому **Компоновщик** и **Декоратор** часто используются совместно.
- **Декоратор** и **Заместитель** имеют разные цели, но похожие структуры. Оба обеспечивают перенаправление вызова через другой объект, и реализация включает в себя ссылку на объект, которому перенаправляется вызов.
- **Фасад** определяет новый интерфейс, тогда как **Адаптер** использует существующий интерфейс. **Адаптер** заставляет два существующих интерфейса работать вместе, а не создает абсолютно новый новый.
- **Фасад**ные объекты – часто **Одиночки**, т.к. обычно требуется только один объект-**Фасад**.

Резюме

- **Посредник** похож на **Фасад** тем, что он абстрагирует функциональность существующих классов. **Посредник** абстрагирует/централизует коммуникацию между объектами-коллегами, он может добавлять какую-то свою функциональность и его знают (имеют ссылки на него) все объекты-коллеги. Напротив, **Фасад** определяет упрощенный интерфейс для подсистемы, не добавляет никакой собственной функциональности и о нем не знают классы подсистемы.
- **Абстрактная Фабрика** может использоваться как альтернатива **Фасаду** если необходимо скрыть набор платформо-зависимых классов.
- Тогда как **Flyweight** показывает, как сделать множество маленьких объектов, **Фасад** показывает ,как сделать один объект, представляющий целую подсистему.
- **Flyweight** часто комбинируется с Компоновщиком для реализации разделяемых листовых элементов.
- **Flyweight** демонстрирует, как могут быть разделены объекты **Состояния**.