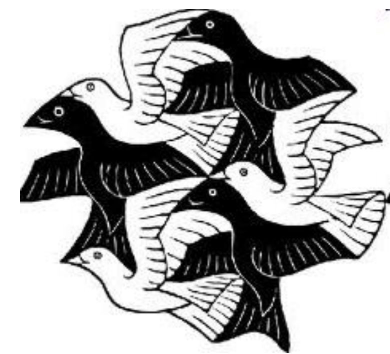


4. Структурни шаблони - 1

Софтуерни шаблони за проектиране

Боян Бончев,
ФМИ – Софийски университет
© 2006/2016

Анотация

- Структурни шаблони
- Определения
- Свойства
- Адаптер, мост, композит - цел, мотивация, структура, участници, коопериране, последствия, въпроси по прилагането на шаблоните
- Примери на Java

Използвана литература

- Gamma, Helm, Johnson, Vlissides **("Gang of Four“ - GoF**) *Design Patterns: Elements of Reusable Object-Oriented Software*, 1995
- *THE DESIGN PATTERNS JAVA COMPANION*, by James W. Cooper, Addison-Wesley, October 2, 1998
- *Design Patterns Explained*, by Allan Shalloway and James Trott, Prentice Hall, 2001
- *Thinking in Patterns*, by Bruce Eckel, Revision 0.9, 5-20-2003
- James O. Coplien (June 1996). *Software Patterns*. ISBN 978-1884842504

Шаблони за проектиране, каталог GoF – оригинални имена

		Цел		
		Създаващи	**Структурни**	**Поведенчески**
Обхват	**Клас-базирани**	•Factory Method	•Adapter	•Interperter
	Обектно-базирани	•Abstract Factory •Builder •Prototype •Singleton	•Bridge •Composite •Decorator •Facade •Flyweight •Proxy	•Chain of Responsibility •Command •Iterator •Mediator •Template method •Memento •Observer •State •Strategy •Visitor

Шаблони за проектиране, каталог GoF – превод на български език

		Цел		
		Създаващи	**Структурни**	**Поведенчески**
Обхват	**Клас-базирани**	•Метод фабрика	•Адаптер	•Интерпретатор
	Обектно-базирани	•Абстрактна фабрика •Строител •Прототип •Сек	•Мост •Композиция •Декоратор •Фасада •Мини-обект •Пълномощник	•Верига от отговорности •Команда •Итератор •Посредник •Шаблонен метод •Спомен •Наблюдател •Състояние •Стратегия •Посетител

Класификация на шаблони за проектиране

- Структурни шаблони - за формиране на големи структури от данни
  - структурните шаблони, ориентирани към класове, изполват йерархия от класове
  - структурните шаблони, ориентирани към обекти, изполват агрегация от обекти

7 структурни шаблона 1/3

- Адаптер (Adapter, GOF 139) - може да се използва за преобразуване на един интерфейс на клас до друг, с цел по-лесно програмиране.
- Мост (Bridge, GOF 151) - разделя абстракция на даден обект от неговата имплементация, така че да можем да ги варираме независимо.
- Композит (Composite, GOF 163) - описва как да се изгради клас-йерархия на обекти, всеки от които може да бъде прост или съставен обект. Съставният обекти ви позволяват да композирате примитивни и композитни обекти в произволно сложни структури.

7 структурни шаблона 2/3

- Декоратор (Decorator, GoF 175) - описва как динамично да добавяме отговорности към обекти; композира обектите рекурсивно, за да позволи отворен брой добавени допълнителни отговорности. Например, декоратор обект, съдържащ компонент от потребителския интерфейс, може да добави декорация като граница или сянка на компонента, или пък функционалност като превъртане и мащабиране.

- Фасада (Facade, GoF 185) - показва как един обект може да представлява цялата подсистема. Фасадата е представител за набор от обекти. Фасадата изпълнява своите отговорности чрез изпращане на съобщения до обектите, които тя представлява.

7 структурни шаблона 3/3

- Миниобект (Flyweight, GoF 195) - определя структура за споделяне на обекти. Обекти се споделят най-малко по две причини: ефективност и съгласуваност. Flyweight се фокусира върху споделяне с цел *ефективно използване на паметта*. При използване на много обекти, трябва да се обърне особено внимание на цената на всеки обект. Пестим чрез споделяне на обекти, вместо чрез репликирането им.

- Пълномощник (Proxy, GoF 207) - действа като контейнер за друг обект, като местен представител за обекта в друго адресно пространство. Може да представлява тежък обект, който се зареждат при поискване, или да защитава достъп до чувствителен обект.

Адаптер (Adapter) [1]

- **Цел -** преобразува интерфейса на даден клас в друг интерфейс, какъвто клиентът очаква. Адаптерът позволява на двата класа да работят заедно, което иначе не е възможно, защото са с несъвместими интерфейси.
- **Познат още като -** Wrapper
- **Мотивация –** разгледаме графичен редактор, който позволява на потребителите да изготвят и организират графични елементи (линии, многоъгълници, текст и т.н.). Интерфейсът за графични обекти се определя от абстрактен клас, наречен Shape. Класовете за елементарни геометрични форми като LineShape и PolygonShape са доста лесни за изпълнение, защото те са с ограничени възможности за изчертаване и редактиране. Но TextShape подкласът за показване и редактиране на текст е значително по-труден за изпълнение, тъй като освен редактирането на текст включва опресняване на екрана и управление на буферирането.

Мотивация (прод.)

Междувременно, дадена рамка с набор от инструменти за потребителски интерфейс може да предлага сложен клас TextView за показване и редактиране на текст. В идеалния случай бихме искали да използваме повторно TextView за имплементация на TextShape, но рамката не е проектирана за Shape класове. Бихме могли да променим класа TextView, така че да съответства на интерфейс Shape, но:

1. сорс кодът на рамката може да не е наличен;
2. безсмислено е да променяме TextView; *рамките не трябва да се преправят за всяко приложение*.

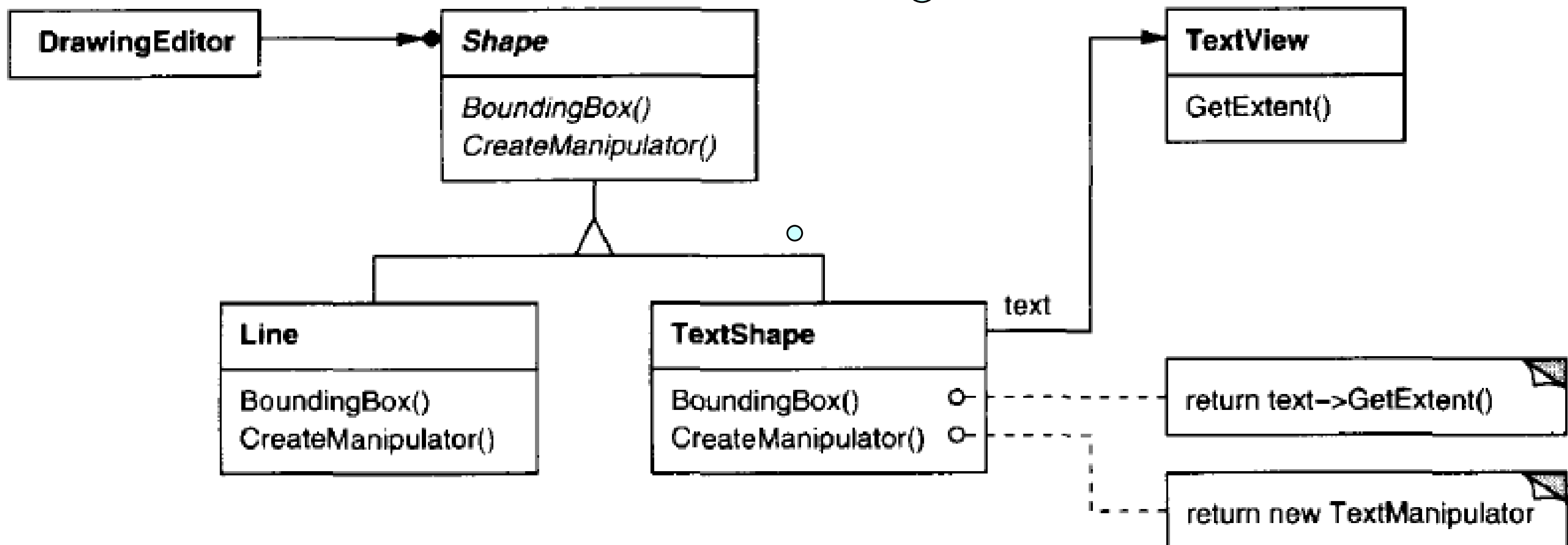
Решението

Понеже TextShape адаптира Text View към интерфейса на Shape, редакторът може да използва иначе несъвместимия клас TextView.

Можем да дефинираме TextShape така, че да адаптира интерфейса на TextView към Shape, по два начина:

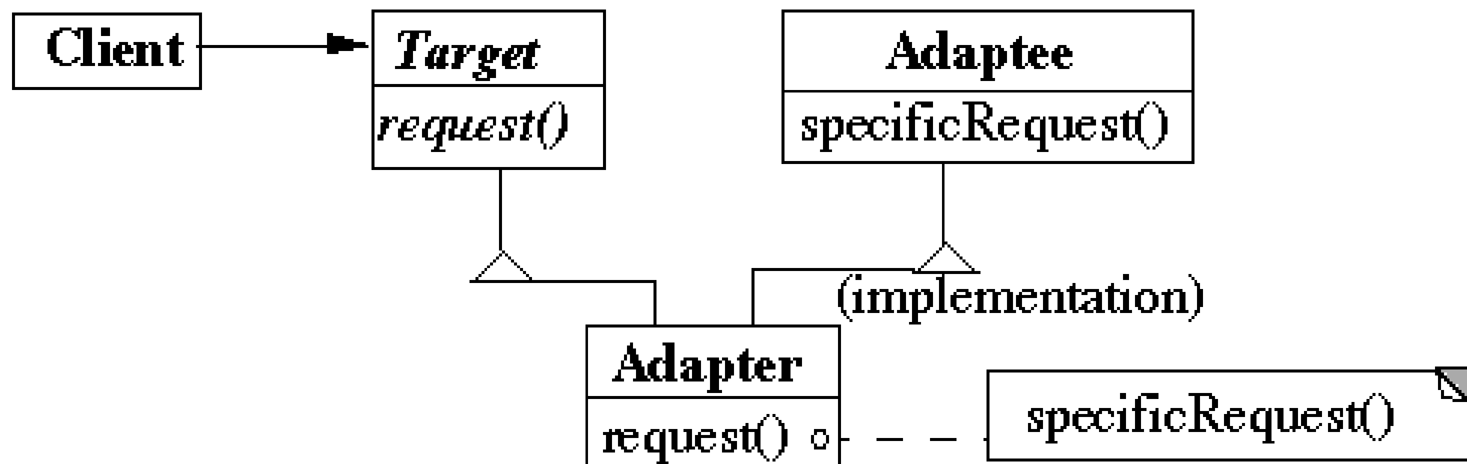
- 1) чрез наследяване на интерфейса на Shape's и на имплементацията на TextView, или
- 2) чрез композиране на екземпляр на TextView в TextShape.

Тези два подхода отговарят на клас-ориентираната и на обектната версии на шаблона Adapter. TextShape е ***adapter***.



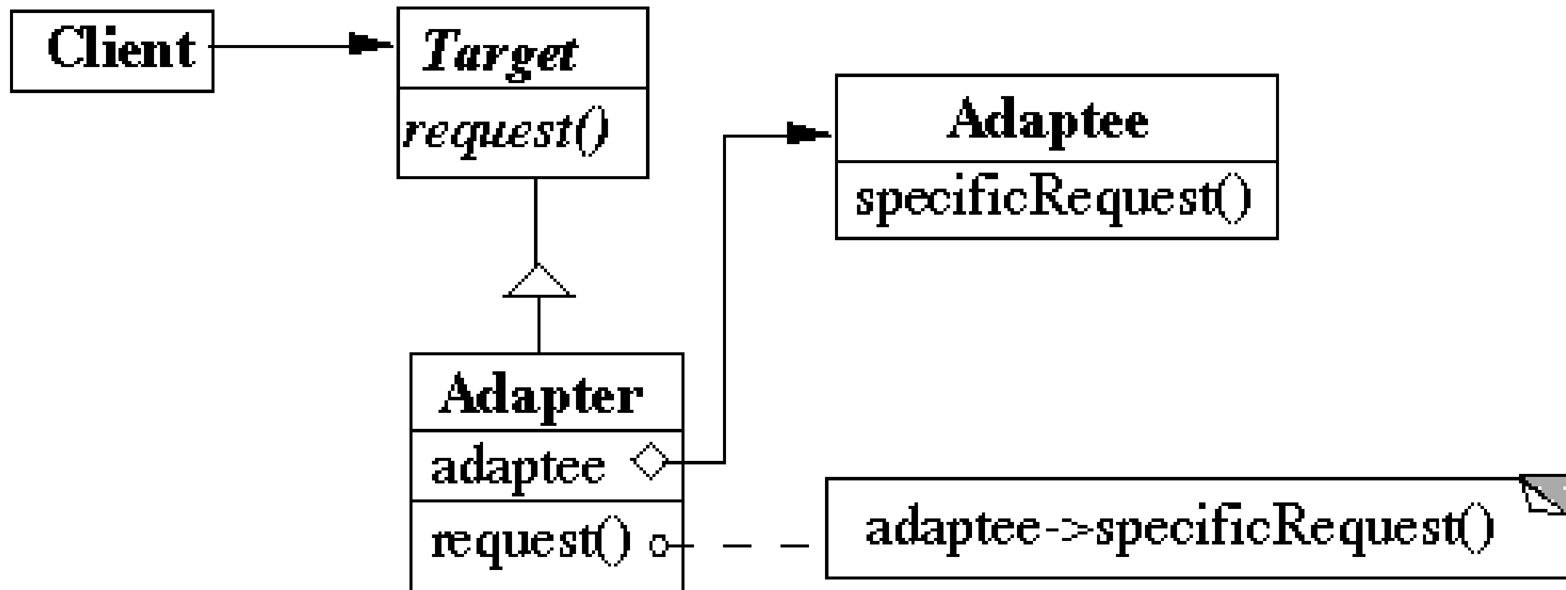
Участници и структура (Class Adapter)

- **Target** (Shape) - дефинира домейн-специфичен интерфейс за ползване от клиента (Client).
- **Client** (DrawingEditor) – коопериран с обекти, ориентирани към интерфейса Target.
- **Adaptee** (TextView) – дефинира съществуващ интерфейс, изискващ адаптиране.
- **Adapter** (TextShape) – адаптира интерфейса Adaptee към този на Target.



Участници и структура (Object Adapter + Delegation)

- **Участници –** същите както на предходния слайд.
- **Комуникация –** клиентите извикват операция на екземпляр на Adapter. На свой ред, адаптерът извиква операции на Adaptee, които изпълняват заявката.



Приложимост

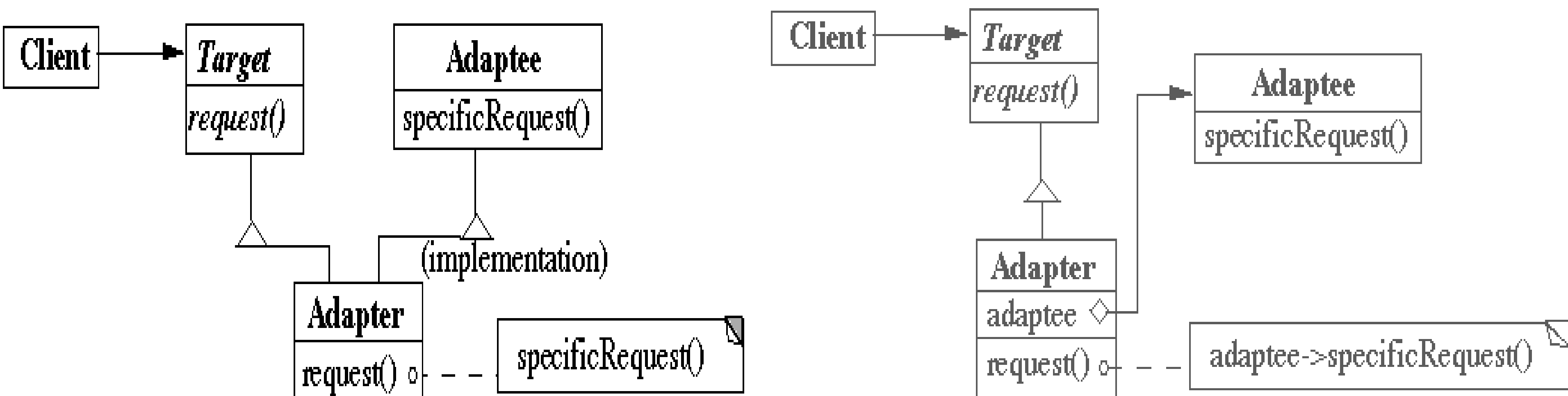
Използваме шаблона Adapter, когато:

- искаме да използваме съществуващ клас и интерфейсът му не съвпада с този, който ни е нужен.
- искаме да създадем клас за многократна употреба, който си сътрудничи с несвързани или непредвидени класове, т.е. с класове, които не е задължително да имат съвместими интерфейси.
- (*само за обектен адаптер*) трябва да използваме няколко съществуващи подкласа, но е непрактично да адаптираме техните интерфейси чрез наследяване от всеки един. Адаптер-обектът може да адаптира интерфейса на родителския клас.

Последствия и компромиси 1/2

При използване на ***class adapter***:

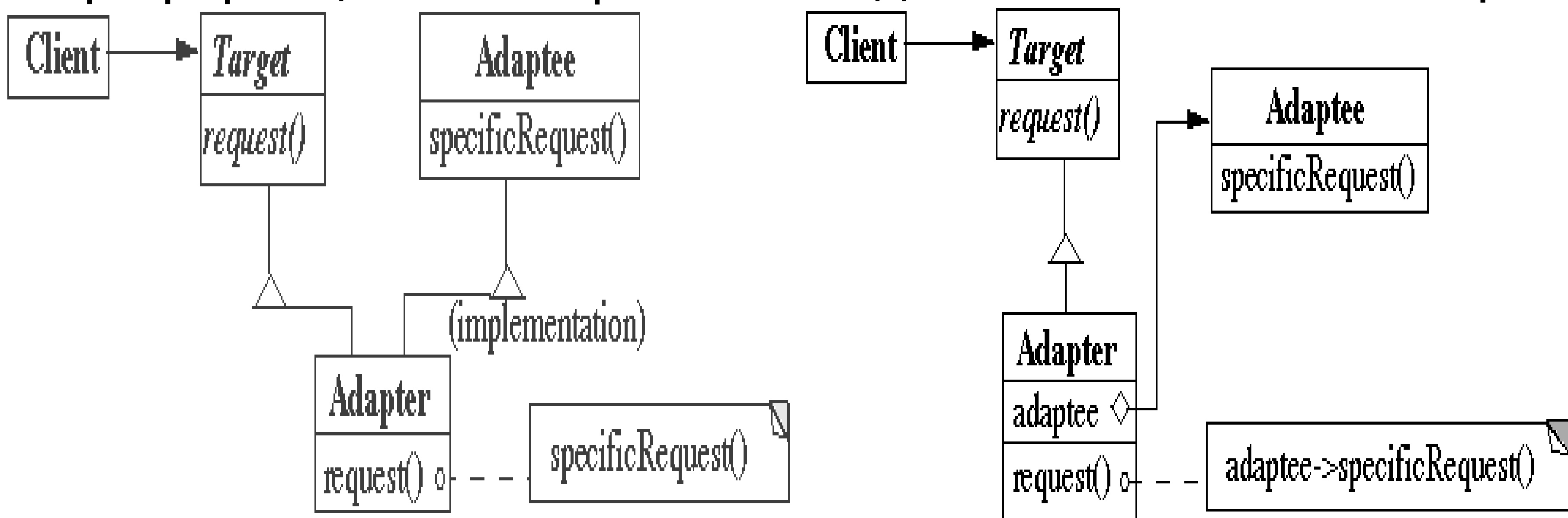
- Адаптираме Adaptee до Target, като се ангажираме с конкретен клас Adapter. В резултат на това, такъв адаптер клас няма да работи, когато искаме *да се адаптира клас и всички негови подкласове*.
- Adapter може да пренапише част от поведението на Adaptee, понеже Adapter е подклас на Adaptee.
- Използва се само един обект, без нужда от указател/референция към Adaptee.



Последствия и компромиси 2/2

При използване на ***object adapter***:

- Разрешаваме на отделния Adapter *<u>да работи с много Adaptees</u>* – т.е. със самия Adaptee и всичките му евентуални подкласове. Adapter може да добави функционалност към всички Adaptees наведнъж;
- *<u>Пренаписването на поведението на Adaptee е по-трудно</u>* – то би изисквало наследяване на Adaptee с референция от Adapter към подкласа вместо към Adaptee.



До каква степен да адаптираме?

- Адаптерът може да работи много малко или пък много, за да се адаптира Adaptee до целта.
- Адаптерът може просто съпостави една операция към друга

```
class PegAdapter: public RoundPeg {
    private: OldSquarePeg* square;
    public: PegAdapter() { square = new OldSquarePeg;}
    void roundPegOperation() { square->squarePegOper(); }
}
```

- Може да се наложи много работа по адаптирането, ако *<u>операциите в Target нямат сравними аналози в Adaptee</u>*

Java **Object** Adapter – преправяме Swing **JList** така, че да изглежда като **awtList**

```
public interface awtList {
//Interfaces are important in Java, because of no multiple inheritance as in C++
        public void add(String s);
        public void remove(String s);
        public String[] getSelectedItems()
}

public class JawtList extends JScrollPane implements awtList {
  private JList listWindow;
  private JListData listContents;
  //-----------------------------------------
  public JawtList(int rows) {
     listContents =
       new JListData();
     listWindow =
       new JList(listContents);
     getViewport().add(listWindow);
}
```

addListDataListener(l)	Add a listener for changes in the data.
removeListDataListener(l)	Remove a listener
fireContentsChanged(obj, min,max)	Call this after any change occurs between the two indexes min and max
fireIntervalAdded(obj,min,max)	Call this after any data has been added between min and max.
fireIntervalRemoved(obj, min, max)	Call this after any data has been removed between min and max.

```
//----------------------------------------
public void add(String s) {
   listContents.addElement(s);
}
//----------------------------------------
public void remove(String s) {
   listContents.removeElement(s);
}
//----------------------------------------
public String[] getSelectedItems() {
   Object[] obj = listWindow.getSelectedValues();
   String[] s = new String[obj.length];
   for (int i =0; i<obj.length; i++)
     s[i] = obj[i].toString();
   return s;
}
}
```

Забележете, че истинската обработка на данните става в класа **JListData**. Този клас е дериван от **AbstractListModel ------------->**

removeListDataListener(l)	Remove a listener
addListDataListener(l)	Add a listener for changes in the data.
	max
fireIntervalAdded(obj,min,max)	Call this after any data has been added between min and max.
fireIntervalRemoved(obj, min, max)	Call this after any data has been removed between min and max.

Клас-ориентиран адаптер

Ако сме създали класа JawtClassList като дериват на JList, то трябва да създадем JScrollPane:

```
leftList = new JclassAwtList(15);
JScrollPane lsp = new JScrollPane();
pLeft.add("Center", lsp);
lsp.getViewport().add(leftList);
//and so forth.
```

Тогава нашият клас-ориентиран адаптер ще е:

```
public class JclassAwtList extends JList implements awtList {
private JListData listContents;
//-----------------------------------------
public JclassAwtList(int rows) {
    listContents = new JListData();
    setModel(listContents);
    setPrototypeCellValue("Abcdefg Hijkmnop");
}
```

```
public class JawtList extends JScrollPane implements awtList {
    …
    //---------------------------------------
      public void add(String s) {
          listContents.addElement(s);
     }
    //---------------------------------------
    public void remove(String s) {
          listContents.removeElement(s);
    }
    //---------------------------------------
    public String[] getSelectedItems() {
          Object[] obj = getSelectedValues();
          String[] s = new String[obj.length];
          for (int i =0; i<obj.length; i++)
          s[i] = obj[i].toString();
          return s;
    }
}
```

Шаблон Мост (Bridge) [GoF]

- **Цел -** Bridge има структура, подобна на обектов адаптер, но с различно намерение: *да се отдели интерфейсът от неговата имплементация, така че те да могат да се променят лесно и независимо* (докато Адаптер е променя интерфейса на вече съществуващ обект!).
- **Познат още като -** Handle/Body
- **Мотивация -** Когато една абстракция може да има една от няколко възможни реализации, обикновено се използва наследяване. Абстрактен клас дефинира интерфейс към абстракцията и конкретни подкласове го имплементират по различни начини. Но това не е достатъчно гъвкаво. Наследяването се свързва с имплементацията на абстракцията постоянно, което го прави трудно за модифициране, разширяване и повторна употреба.

Мотивация

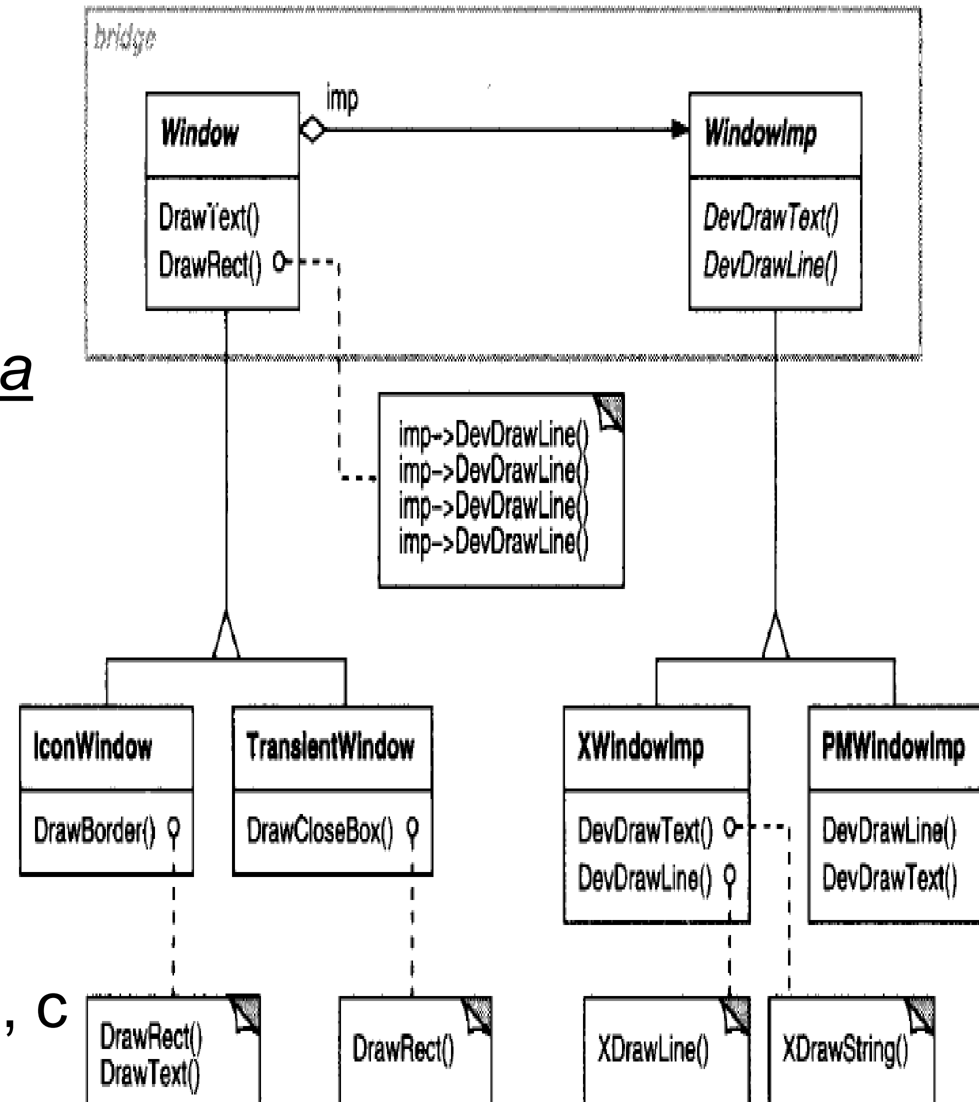
Преносима приложна рамка за потребителски интерфейс ще бъде използвана за приложения, работещи както с X Window System, така и с IBM's Presentation Manager (PM). Чрез наследяване можем да дефинираме абстрактен клас Window и подкласове XWindow и PMWindow, имплементиращи Window интерфейса за различни платформи. Но с два недостатъка:

1. Това е неудобно да се разшири абстракцията *Window* за покриване на различни видове прозорци или нови платформи (например, за IconWindow подклас на Window)
2. Това прави *клиентският код платформено-зависим*. Всеки път, когато клиентът създава прозорец, той инстанциира конкретен клас, който има конкретна имплементация.

- *Клиентите трябва да бъдат в състояние да създават прозорец, без да се ангажират с конкретна имплементация.*.

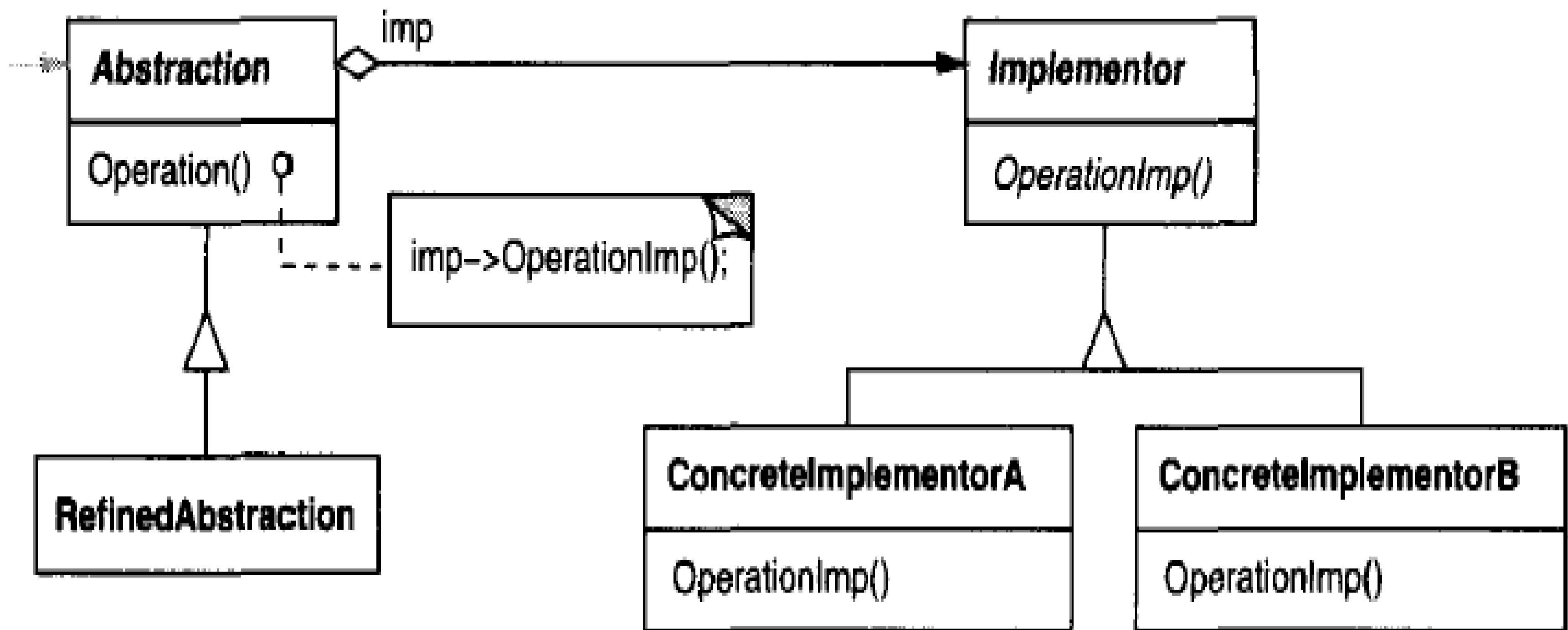
Решение

- Шаблонът Мост (Bridge) решава тези проблеми чрез *разделяне на абстракцията Window и нейните имплементации в отделни йерархии от класове*.
- Отделна йерархия за прозоречните интерфейси (Window, IconWindow, TransientWindow) и друга йерархия за платформено-зависимите имплементации, с корен WindowImp.



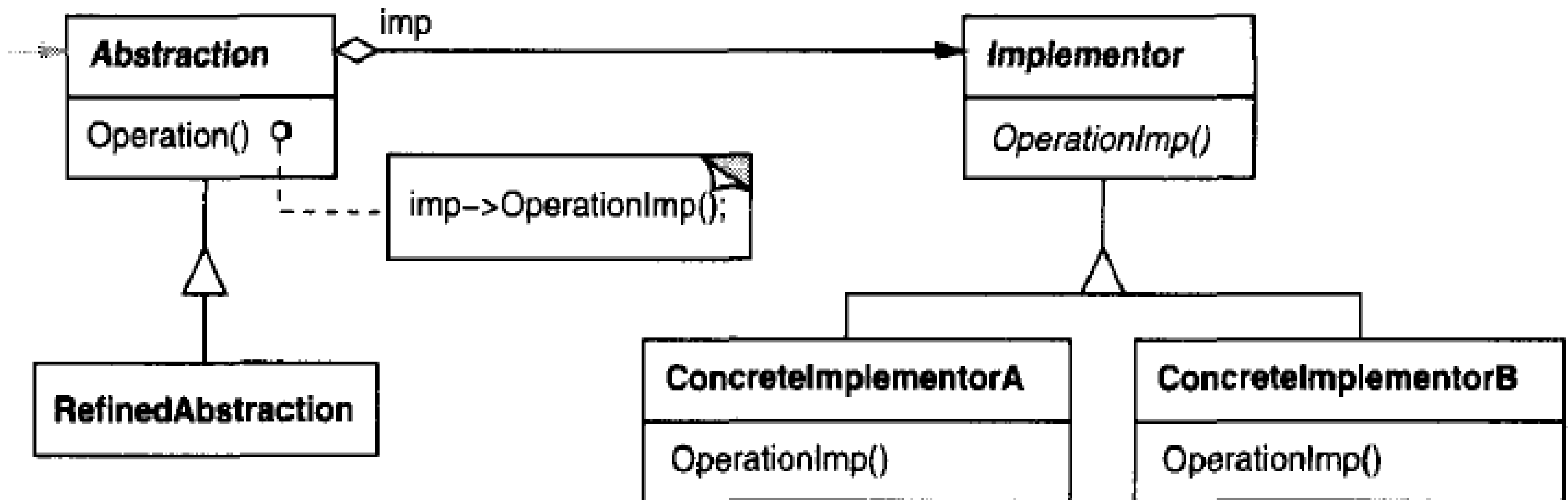
Структура и участници 1/2

- **Abstraction** (Window)
  - - дефинира интерфейса на абстракцията;
  - - поддържа препратка към обект от тип Implementor.
- **RefinedAbstraction** (IconWindow)
  - - разширява интерфейса, дефиниран от Abstraction.



Структура и участници 2/2

- **Implementor** (Windowimp)
  - Дефинира интерфейса на имплементационните класове, **който не е задължително да отговаря точно на интерфейса на Abstraction**.
  - Типично е Implementor да задава само примитивни операции, а Abstraction да дефинира операции от високо ниво, използващи тези примитиви.
- **ConcreteImplementor** (XWindowImp, PMWindowImp)
  - Имплементира интерфейса Implementor interface и така дефинира конкретна имплементация.



Същност

- **Обвързване между абстракция и имплементация**:
  - Една абстракция може да използва различни имплементации
  - Една абстракция може да се използва от различни имплементации

- **Скрива имплементацията от клиентите** – използвайки наследяване

- Клиентският код в този шаблон няма достъп до **имплементациятга**

Приложимост

Използваме шаблона Мост (Bridge), когато:

- искаме да се избегне трайното обвързване между абстракция и неговото прилагане (имплементация)
- абстракциите и техните имплементации трябва да бъдат независимо разширяеми чрез наследяване
- промените в имплементация на абстракция трябва да нямат влияние върху клиентите, т.е. не трябва да трябва се налага да се прекомпилира клиентският код
- искаме да скрием изцяло изпълнението на абстракция от клиентите (потребители)
- искаме да споделим имплементацията сред множество обекти, и то скрито от клиента.

Последствия

- *Отделяне на интерфейс от изпълнение.*
  - Имплементацията не е свързана постоянно с даден интерфейс.
  - Изпълнението на абстракция може да бъде конфигурирано и променяно по време на изпълнение.
  - Също така се елиминират компилационните зависимости за имплементацията. На високо равнище система трябва да знае само за Abstraction и Implementor.
- *Подобряване на разширяемостта.* Можем да разширяваме самостоятелно йерархиите на Abstraction и Implementor.
- *Скриване на подробностите по изпълнението от клиентите* (като напр. споделянето на имплементационни обекти).

Java пример – мост към JList и JTable имплементации [3]

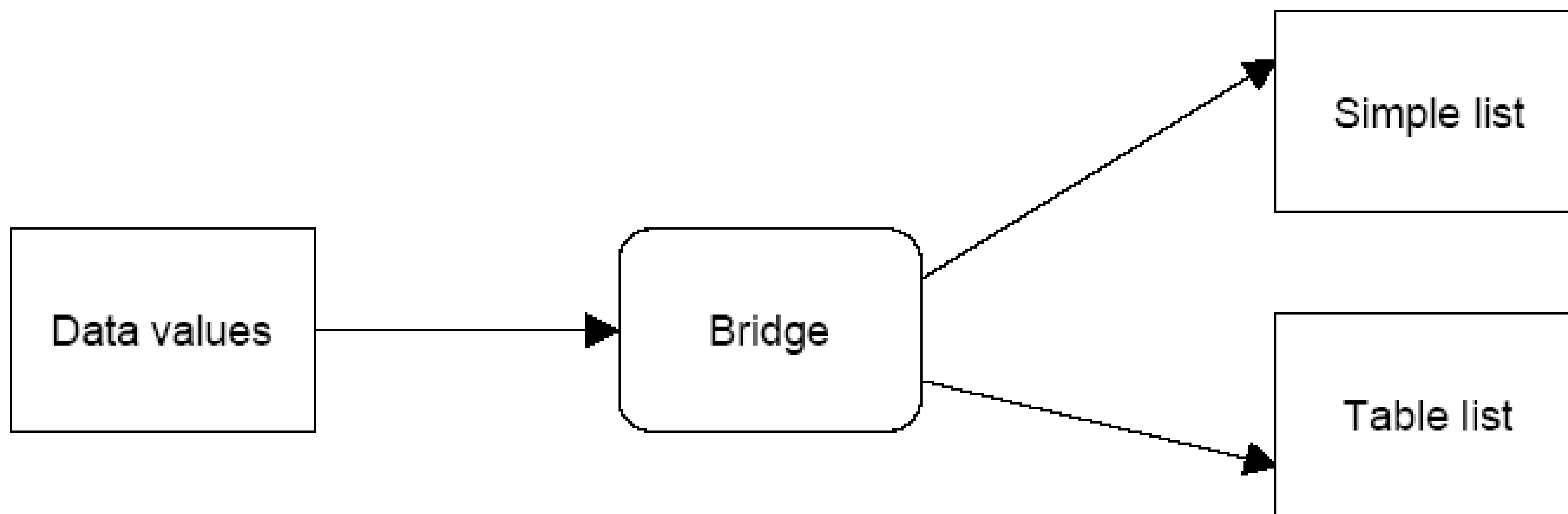
Трябва да се произведем два вида дисплеи от нашите продуктови данни:

- **customer view – клиентски изглед -** това е просто JList на продуктите
- **executive view -** показва броя на доставени единици.

```
public class productList extends JawtList {
    public productList(Vector products) {
        super(products.size()); //for compatibility
        for (int i = 0; i < products.size(); i++) {
          //take each string apart and keep only
          //the product names, discarding the quantities
          String s = (String)products.elementAt(i);

          //separate qty from name
          int index = s.indexOf("--");
          if(index > 0)
             add(s.substring(0, index));
          else
             add(s);
        }
    }
}
```

- Сега предполагаме, че трябва да се направят някои промени в начина, по който тези списъци показват данните - т.е. продукти, показани в азбучен ред. За да се продължи с този подход, ще трябва или да променим тези класове, или да ги наследим...
- Вместо тази кошмарна поддръжка – използваме мост:

public class listBridge extends JscrollPane



- Когато можем да създадем клас мост, ние трябва да решим как мостът ще определи кой от няколкото класа да инстанциира:

```
static public final int TABLE = 1, LIST = 2;
```

- Добавяме:

```
pleft.add("North", new JLabel("Customer view"));
pleft.add("Center",
        new listBridge(prod, listBridge.LIST));
//adds the execute view as table
pright.add("North", new JLabel("Executive view"));
pright.add("Center",
        new listBridge(prod, listBridge.TABLE));
```

- Конструктор за класа listBridge:

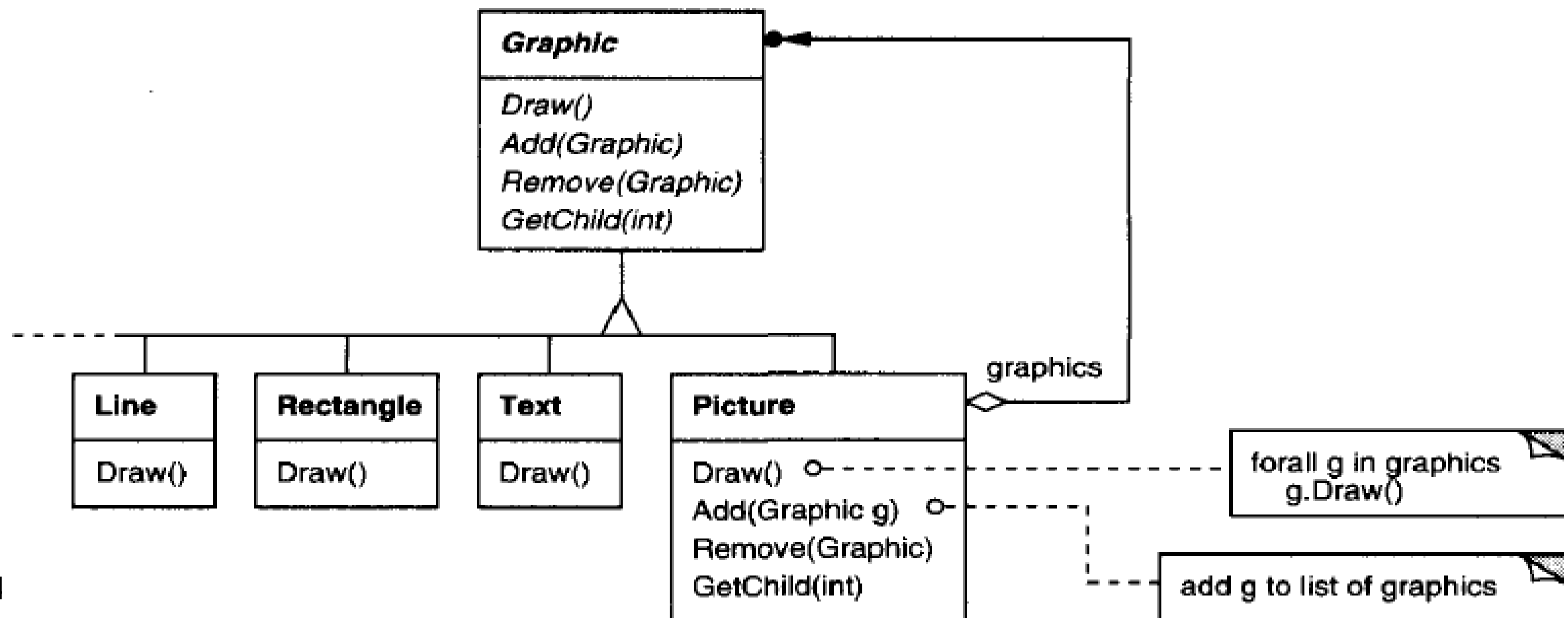
```
public listBridge(Vector v, int table_type) {
        Vector sort = sortVector(v); //sort the vector
        if (table_type == LIST)
            getViewport().add(makeList(sort)); //make table
        if (table_type == TABLE)
            getViewport().add(makeTable(sort)); //make list
        }

private JList makeList(Vector v) {
    return new JList(new BridgeListData(v));
}

private JTable makeTable(Vector v) {
    return new JTable(new prodModel(v));
}
```

Шаблон Композит (Composite)

- **Цел –** композира обекти в дървовидна структура, за представяне на йерархична структура от тип част-цяло. *Composite оставя клиентите да третират индивидуалните обекти и композициите по унифициран начин.*
- **Мотивация** – графичните приложения като редактори и др. изграждат комплексни диаграми от прости компоненти. Проста имплементация би дефинирала ***graphical primitives*** - такива като Text и Lines плюс други класове, служещи като ***containers*** за тези примитиви.
  Възниква обаче проблем с такъв подход: Кодът, който използва тези класове, *трябва да третира примитивните и контейнеровите обекти по различен начин*. Така приложението се усложнява.

Решение

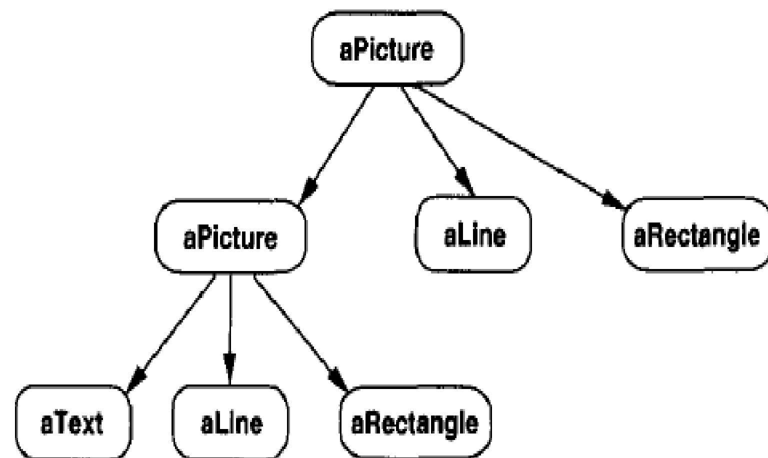
- Шаблонът Композит описва как да използваме рекурсивна композиция, така че клиентите да не трябва да правят тази разлика м/у примитивни и композитни обекти.
- Ключът на Composite шаблона е абстрактен клас, който представлява както примитиви, така и техните контейнери. За графичната система, този клас е Graphic.
- Graphic декларира операции като Draw, които са специфични за графичните обекти, както и операции за всички съставни обекти, напр. за достъп до децата.

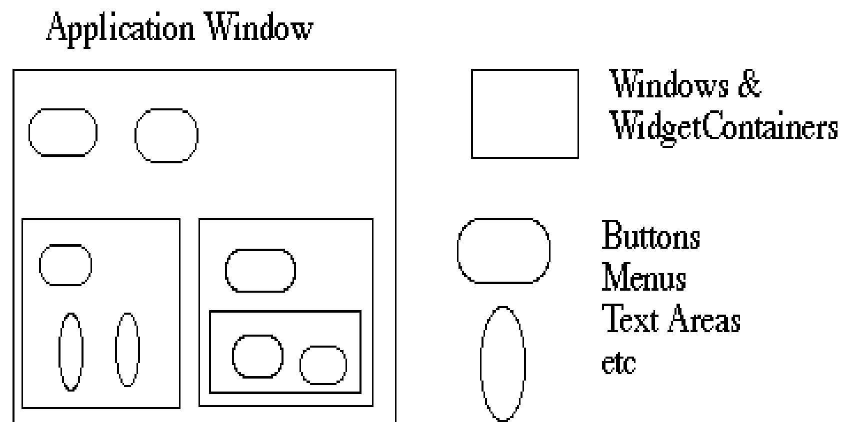


Приложимост

Използваме шаблон Composite, когато:

- искаме да представим йерархии от обекти от тип *част-цяло*.
- искаме клиентите да могат да пренебрегват разликата между композити от обекти и отделни обекти. Клиентите ще третират еднакво всички обекти в композитната конструкция.

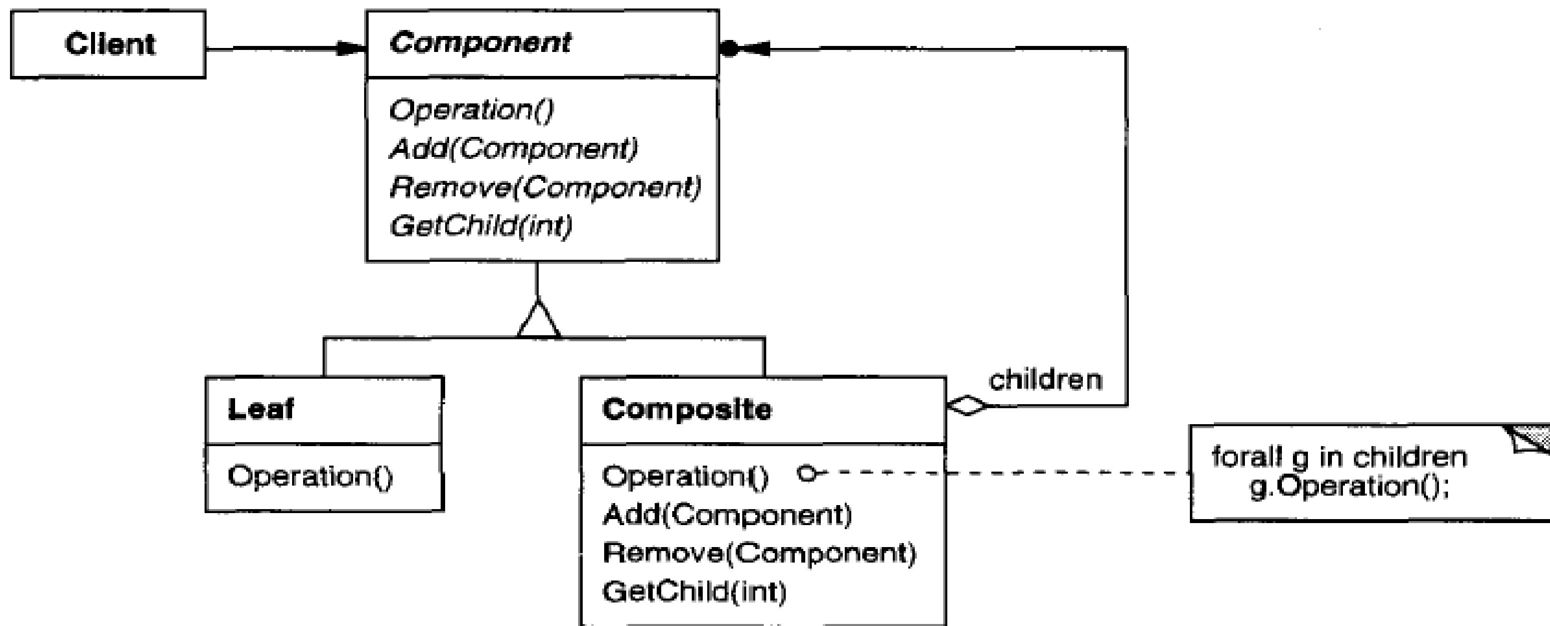




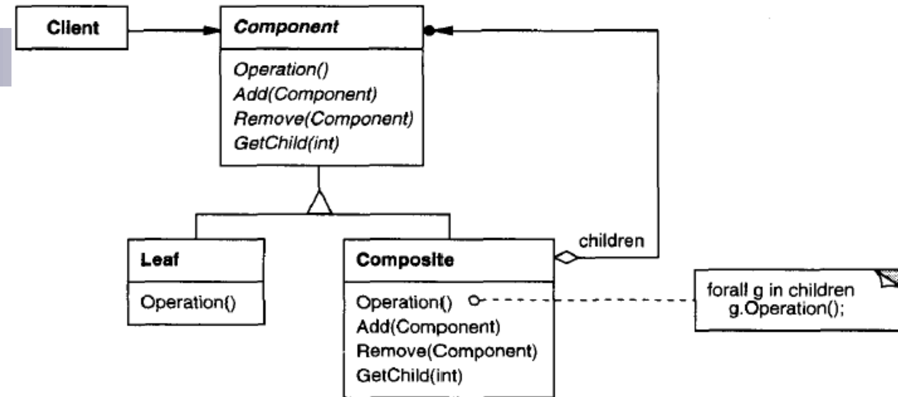
Структура и комуникация

Сътрудничество - клиентите използват интерфейса Компонент, за да взаимодействат с обекти в композитната конструкция:

- Ако получателят е листо, тогава заявката се обработва директно.
- Ако получателят е композит, то тогава обикновено заявката се изпраща на децата компоненти, евентуално с допълнителни операции преди и/или след изпращане.



Участници



- **Component** (Graphic)
  - декларира интерфейс за обекти в композицията.
  - реализира поведението по подразбиране за общ интерфейс за всички класове, в зависимост от случая.
  - декларира интерфейс за достъп и управление на компонент-дете
  - (по избор) дефинира интерфейс за достъп до родителски компонент в рекурсивна структура, и евентуално го имплементира.
- **Leaf** (Rectangle, Line, Text, etc.)
  - представя обект-листо (без деца) в композицията;
  - дефинира поведение за примитивните обекти.
- **Composite** (Picture)
  - дефинира поведение за компонентите, имащи деца;
  - съхранява децата;
  - имплементира операции за децата в интерфейса Component.
- **Client**
  - управлява обектите в композицията чрез интерфейса Component.

Последствия от Композит

- дефинира клас йерархии, състоящи се от примитивни обекти и композити. Примитивните обекти може да се вграждат в по-сложни обекти, което от своя страна се композират отново, и така нататък рекурсивно.
- прави клиента прост. Клиентите могат да третират композитни структури и отделни обекти по един и същи начин. Клиентите обикновено не знаят дали те се занимават с листо или композитен компонент (и не трябва да се грижат). Това опростява кода на клиента.
- прави по-лесно добавянето на нови видове компоненти. Новоопределените Composite или Leaf подкласове работят автоматично със съществуващите структури и клиентски код. Клиентите не трябва да бъдат променяни за нов компонент.
- Недостатък - по-трудно се ограничават компонентите на композитната структура – дизайнът е прекалено общ.

Детайли на имплементацията 1/3

- 1. Експлицитни родителски референции. Поддържането на референции *от децата към техния родител* може да опрости прекосяването и управлението на композитната конструкция.
  - *Обичайното място за определяне на родителска референция е в Component класа*. Leaf и Composite могат да наследят препратката и операциите, които го управляват.
  - Чрез родителски референции можем да поддържаме инварианта, че *всички деца от една композиция имат за родител композит, който от своя страна ги има като деца*.
- 2. Споделяне на компоненти - често е полезно с цел намаляване на заетата памет. Но когато компонент може да има не повече от един родител, споделянето на компоненти става трудно. Едно възможно решение за децата е да съхраняват множество родители.

Детайли на имплементацията 2/3

- 3. Максимизиране на интерфейса Component. Една от целите на шаблона е да направи клиентите нечувствителни от специфични Leaf или Composite, които използват.
  - За да се постигне тази цел, клас Компонент следва да определи колкото се може повече общи операции за Composite и Leaf класове.
  - Component класът обикновено осигурява по подразбиране реализации за тези операции, и Leaf и Composite подкласовете могат да ги предефинират. Въпреки това, тази цел понякога е в противоречие с принципа на дизайна на йерархията на класовете, който казва, че класът трябва да определя само операции, които са от значение на всички негови подкласове.

Детайли на имплементацията 3/3

- 4. Деклариране на операции за управление на децата – макар че Composite имплементира Add и Remove операции, то важно е къде те се декларират в йерархията.

  Трябва ли да се декларират тези операции в компонента и да ги направим смислени за Leaf класове, или трябва да се декларират и да ги определят само в Composite и неговите подкласове?

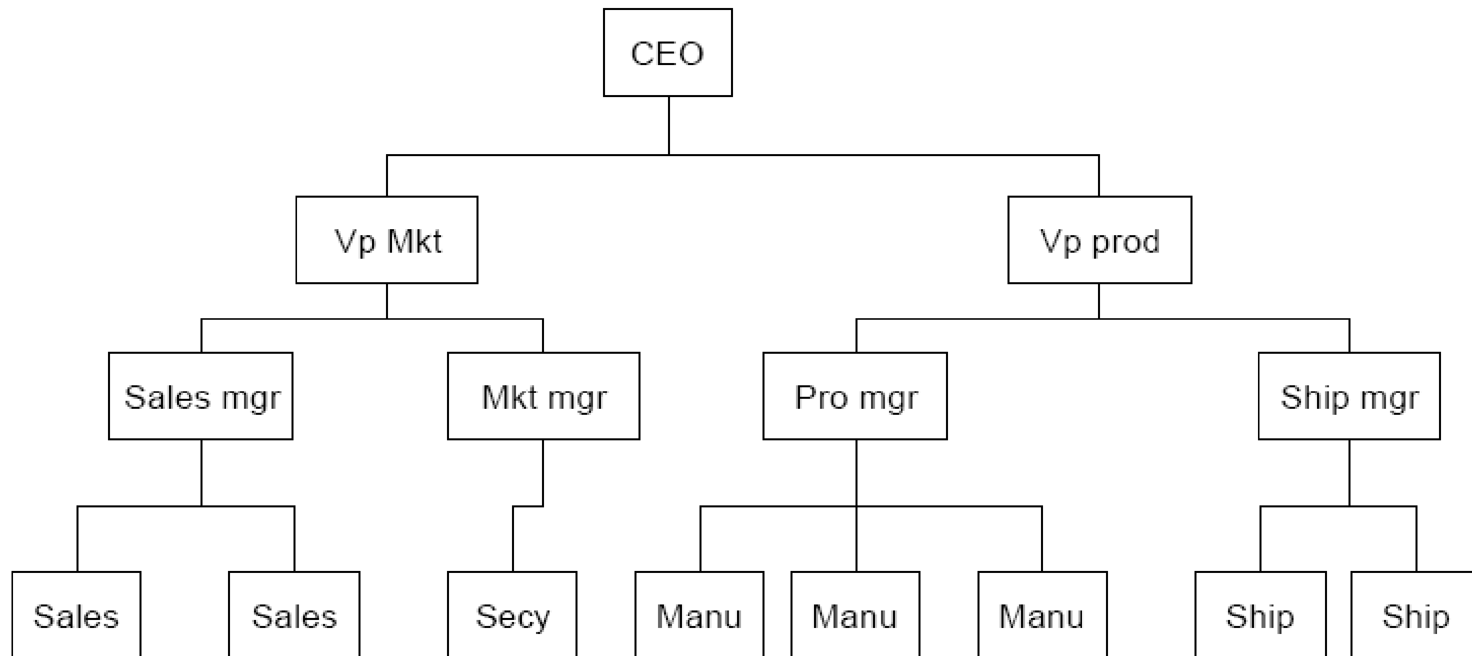
  Дискусия

Детайли на имплементацията 3/3

- 4. Деклариране на операции за управление на децата – макар че Composite имплементира Add и Remove операции, то важно е къде те се декларират в йерархията. *Трябва ли да се декларират тези операции в компонента и да ги направим смислени за Leaf класове, или трябва да се декларират и да ги определят само в Composite и неговите подкласове?*
- Решение - компромис между ***сигурност*** и ***прозрачност***:
  - Дефиниране на интерфейса за управление на дете в корена на йерархията дава *прозрачност*, защото можем да се отнасяме към всички компоненти еднакво. Това обаче ще ни намали *безопасността*, тъй като клиентите могат да се опитат да правят безсмислени неща като премахване на обекти от листата
  - Дефиниране на управление на дете в Composite класа дава *безопасност*, защото всеки опит да добавяме или премахваме обекти от листа ще бъдат уловени по време на компилация. Но губим *прозрачност*, защото листата и композитите имат различни интерфейси

Пример на Java – композиране на компания

- Цената на отделен служител е просто заплатата му и надбавките към нея.
- Цената на един служител, който оглавява отдел, е неговата заплата плюс тези на всичките си подчинени.



```
public class Employee {
    String name;
    float salary;
    Vector subordinates;
    //------------------------------------
    public Employee(String _name, float _salary) {
      name = _name;
      salary = _salary;
      subordinates = new Vector();
    }
    //------------------------------------
    public float getSalary() {
      return salary;
    }
    //------------------------------------
    public String getName() {
      return name;
    }
```

```
    public void add(Employee e) {
            subordinates.addElement(e);
            }
    //--------------------------------------------
    public void remove(Employee e) {
            subordinates.removeElement(e);
    }
    public Enumeration elements() {
            return subordinates.elements();
    }
    public float getSalaries() {
      float sum = salary; //this one’s salary
      //add in subordinates salaries
      for(int i = 0; i < subordinates.size(); i++) {
            sum += ((Employee)subordinates.elementAt(i)).getSalaries();
      }
      return sum;
    }
}
```

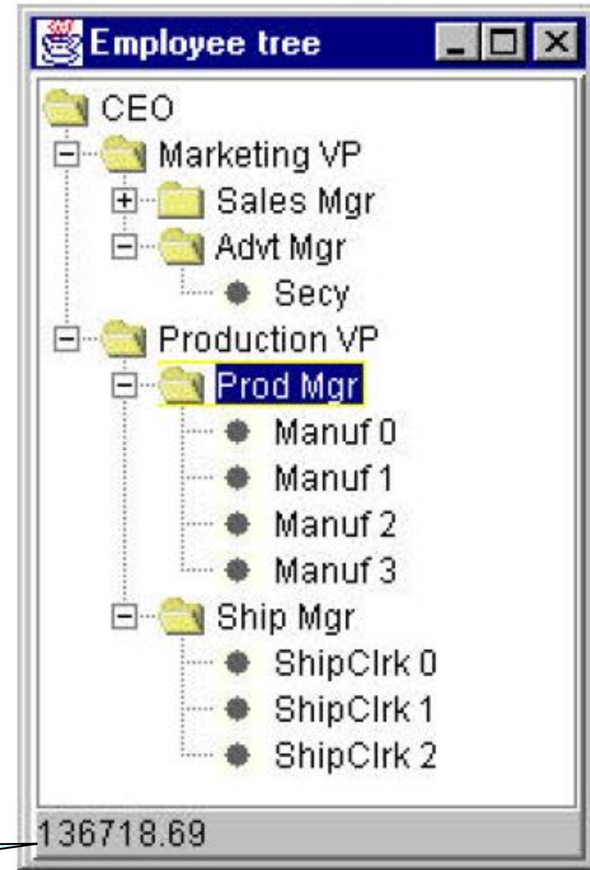
//Building the Employee Tree

```
boss = new Employee("CEO", 200000);
boss.add(marketVP = new Employee("Marketing VP", 100000));
boss.add(prodVP = new Employee("Production VP", 100000));
marketVP.add(salesMgr = new Employee("Sales Mgr", 50000));
marketVP.add(advMgr = new Employee("Advt Mgr", 50000));
//add salesmen reporting to Sales Manager
for (int i=0; i<5; i++)
    salesMgr .add(new Employee("Sales "+ new Integer(i).toString(),
    30000.0F+(float)(Math.random()-0.5)*10000));
….
```

//Building a JTree list

```
private void addNodes(DefaultMutableTreeNode pnode, Employee emp) {
    DefaultMutableTreeNode node;
    Enumeration e = emp.elements();
    while(e.hasMoreElements()) {
      Employee newEmp = (Employee)e.nextElement();
      node = new DefaultMutableTreeNode(newEmp.getName());
      pnode.add(node);
      addNodes(node, newEmp);
    }
}
```

```
public void valueChanged(TreeSelectionEvent evt) {
  //called when employee is selected in tree list
  TreePath path = evt.getPath();
  String selectedTerm =
    path.getLastPathComponent().toString();
   //find that employee in the composite
   Employee emp = boss.getChild(selectedTerm);
   //display sum of salaries of subordinates(if any)
   if(emp != null)
     cost.setText(new
          Float(emp.getSalaries()).toString());
}
```



Заплата за избраното поддърво (employee + subordinates)