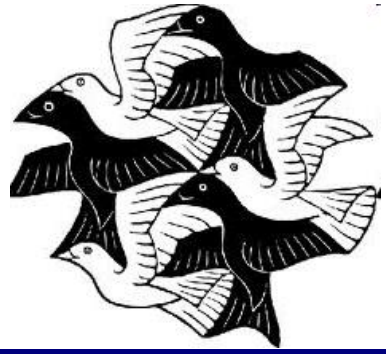


4. Структурни шаблони - 2

Софтуерни шаблони за проектиране

Боян Бончев,
ФМИ – Софийски университет

Анотация

- Структурни шаблони
- Определения
- Свойства
- Декоратор, Фасада, Мини-обект, Пълномощник - цел, мотивация, структура, участници, кооперирane, последствия, въпроси по прилагането им
- 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 използват агрегацията от класове
  - структурните шаблони, ориентирани към обекти, използват агрегацията 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 вариираме 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.

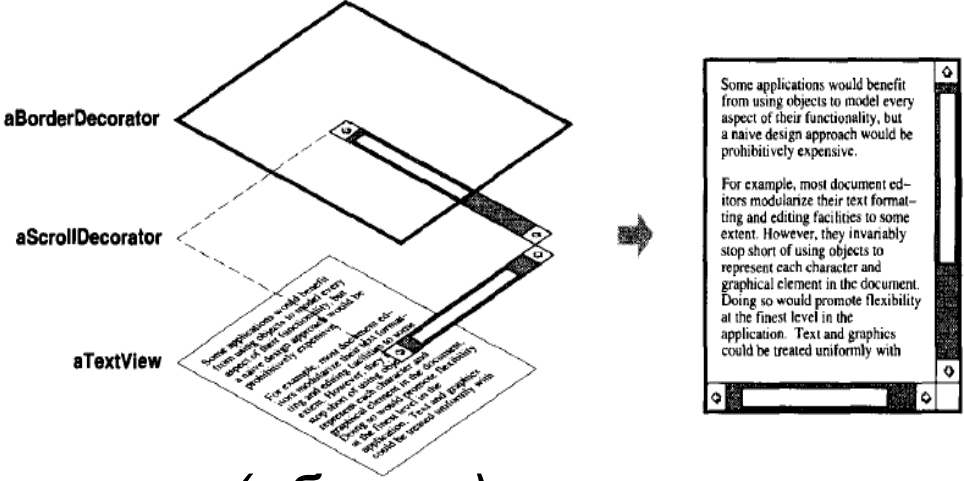
- Пълномощник (Proxy, GoF 207) - действа като контейнер за друг обект, като местен представител за обекта в отдалечено адресно пространство. Може да представлява тежък обект, който се зареждат при поискване, или да защитава достъп до чувствителен обект.

Шаблон Декоратор (Decorator) [3]

- **Цел** – добавя допълнителни отговорности към обект, и то ***динамично***. Декораторът предоставя гъвкава алтернатива на наследяването за разширяване на функционалността.
- **Познат още като** – Обвивка - Wrapper (<>Adapter!)
- **Мотивация** - понякога искаме да добавим отговорности към отделни обекти, а не върху целия клас. Даден инструментариум за графичен потребителски интерфейс трябва да ви позволи да добавяте свойства като ивица (border) или поведение като превъртане към някои от компонентите на потребителския интерфейс.

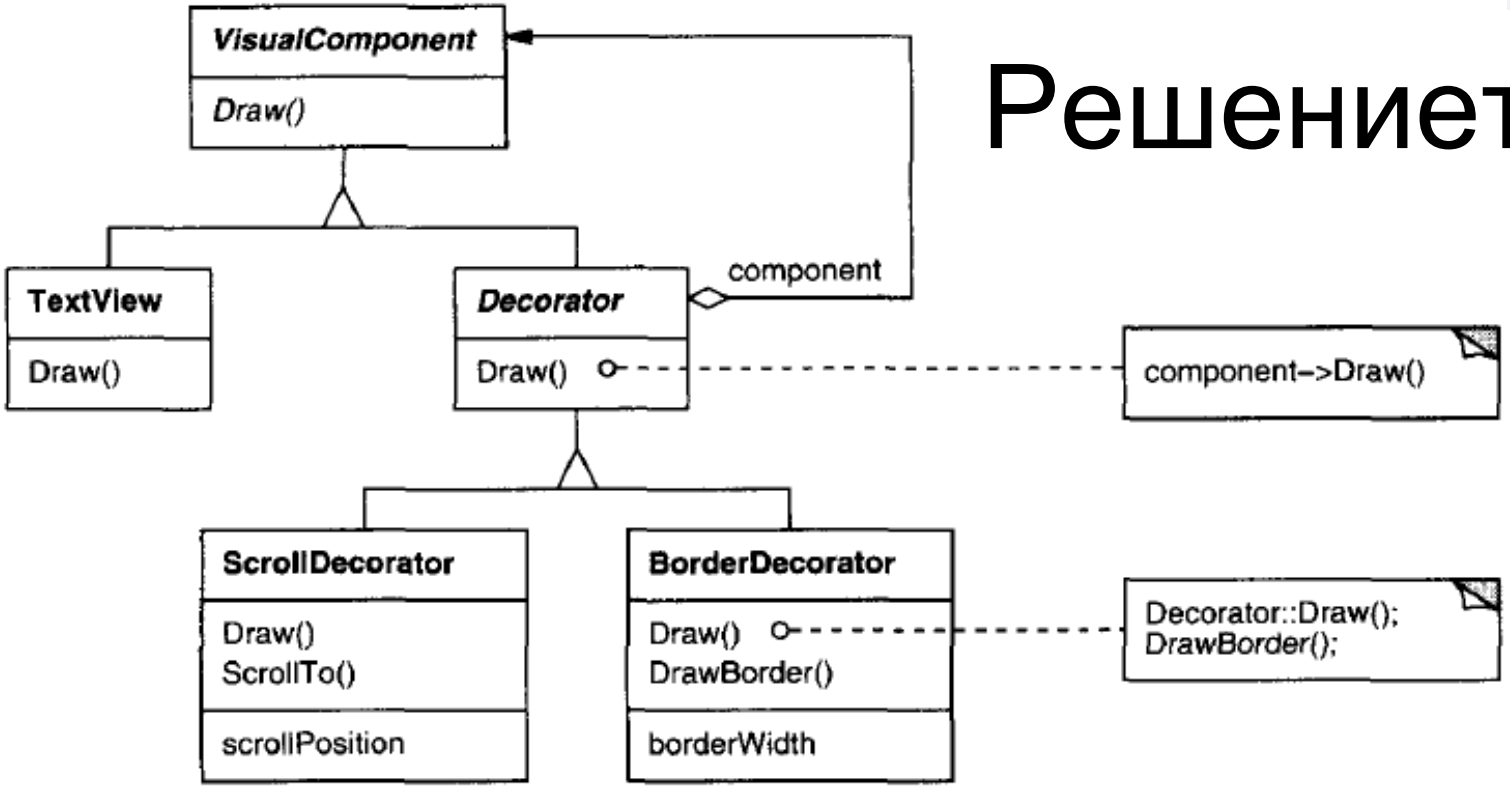
  Един начин да добавяне отговорности е използването на подкласове - *inheritance*. Наследяването на ивица от друг клас обаче разполага ивицата върху всеки екземпляр на класа, т.е. ***статично*** – клиентът няма контрол над декорацията с ивица.

Мотивация

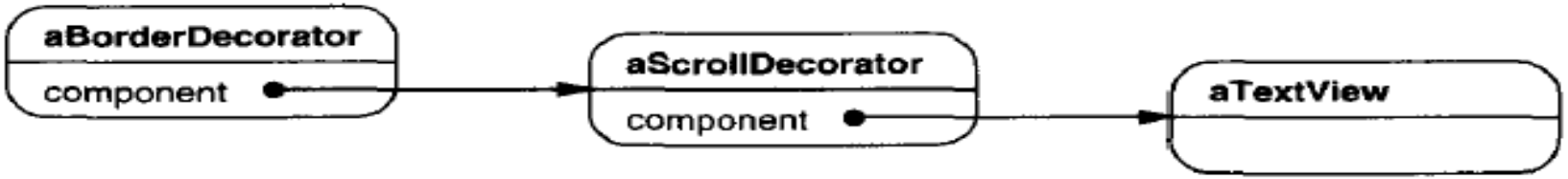


- По-гъвкав подход е *да обхванем (обвием) компонента в друг обект, който добавя ивица*.
- Обхващащият обект се казва **декоратор**. Декораторът отговаря на интерфейса на компонента и го декорира така, че неговото присъствие е *прозрачно за клиентите на компонента*.
- Декораторът предава заявките към този компонент и може да извършва допълнителни действия (като поставяне на ивица) преди или след предаването.
- Прозрачността позволява декораторите да се вграждат *рекурсивно*, и така да добавяме много отговорности.

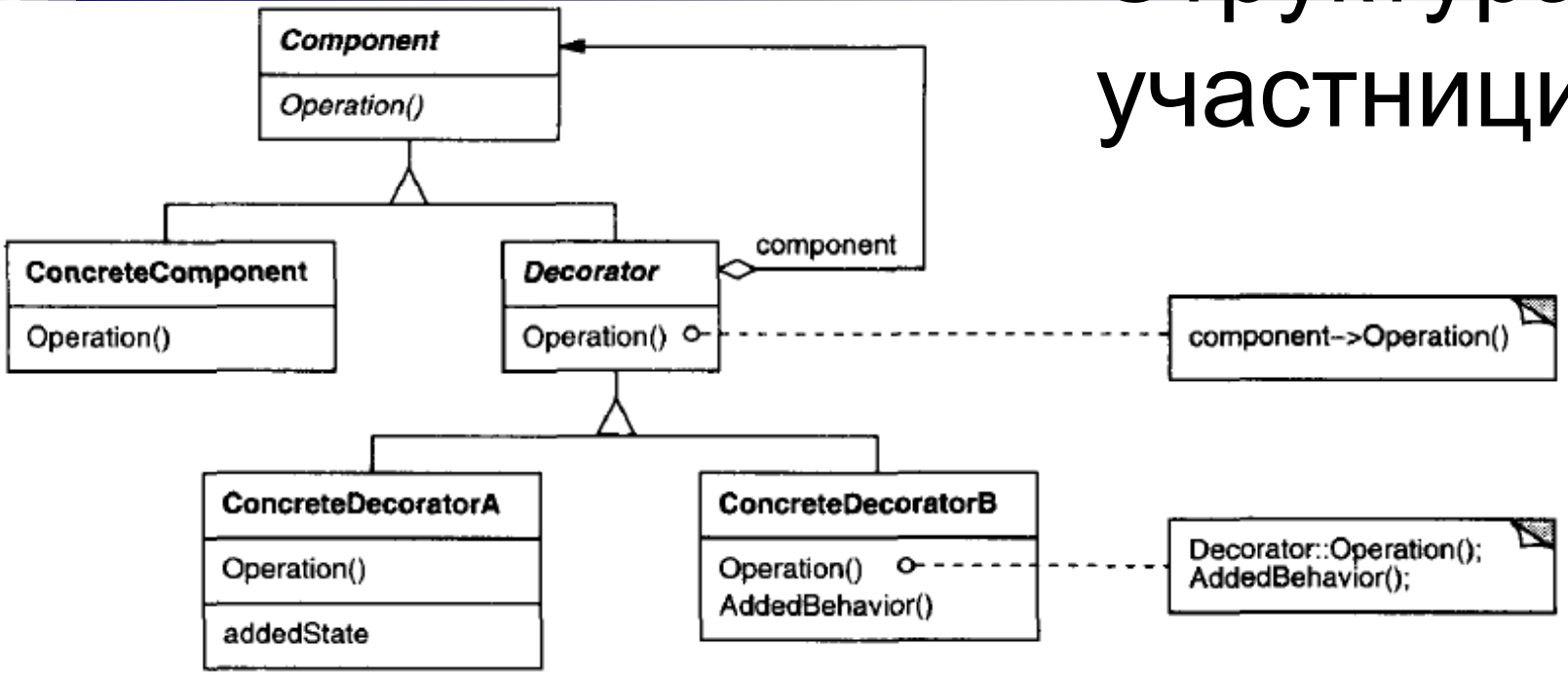
Решението



- Композираме декоратори с **TextView**, за да получим желания резултат.
- Класовете **ScrollDecorator** и **BorderDecorator** са подкласове на ***Decorator*** - абстрактен клас за визуални компоненти, които декорират други визуални компоненти.



Структура и участници



- ***Component*** (*VisualComponent*) - дефинира интерфейс за обекти, които могат да имат отговорности, добавени към тях динамично.
- **ConcreteComponent** (TextView) - определя обект, към който могат да бъдат прикрепени допълнителни отговорности .
- ***Decorator*** - поддържа референция към Component обект и се придържа към интерфейса на Component.
- **ConcreteDecorator** (BorderDecorator, ScrollDecorator) – добавя отговорности към компонента.

Взаимодействия

Взаимодействия - *Decorator* препредава заявките към своя *Component* обект. Той може евентуално да изпълни допълнителни операции *преди или след* предаването на заявката.

Използваме Decorator:

- За да добавяме отговорности към отделни обекти *динамично и прозрачно*, т.е. без да ги засягаме
- Отговорности, които може да бъдат оттеглени
- Ако разширяването с подкласове е непрактично:
  - Когато голям брой от независими разширения биха довели до експлозия от класове, или
  - Ако дефиницията на класа е скрита или той не подлежи на разширения

Последствия 1/2

- 1. Декораторът осигурява:
  - по-гъвкав начин да добавим отговорности на обекти, отколкото може да се постигне със статично (множествено) наследяване.
  - Отговорности могат да се добавят и премахват по време на изпълнение, просто като се прибавя и маха декоратор.
  - За разлика от това, наследяването изисква създаването на нов клас за всяка допълнителна отговорност (напр. BorderedScrollableTextView или BorderedTextView). Това води до много класове и увеличена сложност.

Последствия 2/2

- 2. *Pay-as-you-go* подход към добавянето на отговорности. Приложенията не трябва да плащат за функционалности, които не използват.
- 3. Декораторът и неговия компонент не са идентични. Декораторът действа като прозрачна кутия (*transparent enclosure*). Но от гледна точка на идентичността на обекта, декорираният компонент не е идентичен на самия компонент.
- 4. Много малки обекти.

Имплементация

- *Interface conformance* – интерфейсът на декоратора трябва да отговаря на интерфейса на декорирания обект;
- *Omitting the abstract Decorator class* – няма нужда да дефинираме абстрактен Decorator клас, *когато дабавяме само една отговорност*. Това е често срещан случай при работа със съществуваща йерархия (не създаваме нова). В този случай можем да вкараме в ConcreteDecorator отговорността на декоратора за предаване на заявката към обекта.
- *Keeping Component classes light-weight* – важно е да се поддържа общия клас лек, т.е. той трябва да се съсредоточи върху определянето на интерфейса, а не върху съхраняването на данни.
- *Changing the skin of an object versus changing its guts* - декораторът е обвивка над обект, която променя поведението му, за разлика от вътрешността му - стратегия - Strategy (GoF315).

Java пример – *декориране на* JButton [3]

- GoF предполага, че Декоратор трябва да бъде получен от общ Visual Component клас и всяко съобщение за действителния компонент (напр. бутон) трябва да бъде препратено от декоратора. Тук получаваме нашия декоратор от класа JComponent и използваме неговия контейнер, за да предаде всички извиквания към бутона, той ще съдържа.

- GoF предлага класове като Decorator да бъдат **абстрактни** и от такъв абстрактен клас да се извлича поведението на конкретните декоратори (има смисъл за добавяне на много отговорности). В този случай Java Decorator клас наследява JComponent.

```
//this class decorates a CoolButton with borders invisible when the mouse
//is not over the button
public class CoolDecorator extends Decorator {
    boolean mouse_over; //true when mouse over button
    JComponent thisComp;
    public CoolDecorator(JComponent c) {
        super(c);
        mouse_over = false;
        thisComp = this; //save this component
        //catch mouse movements in inner class
        c.addMouseListener(
          new MouseAdapter(){
            public void mouseEntered(MouseEvent e) {
              mouse_over=true; //set flag when mouse over
              thisComp.repaint();
            }
            public void mouseExited(MouseEvent e) {
              mouse_over=false; //clear if mouse not over
              thisComp.repaint();
            }
         });
}
```

```
public class Decorator extends
JComponent {
    public Decorator(JComponent c) {
    setLayout(new BorderLayout());
    //add component to container
    add("Center", c);
  }
}
```

```
//paint the button
public void paint(Graphics g){
  super.paint(g); //first draw the parent button!!!
  if(! mouse_over) {
    //if the mouse is not over the button,
    //then erase the borders!
    Dimension size = super.getSize();
    g.setColor(Color.lightGray);
    g.drawRect(0, 0, size.width-1, size.height-1);
    g.drawLine(size.width-2, 0, size.width-2,
                    size.height-1);
    g.drawLine(0, size.height-2, size.width-2,
                    size.height-2);
   }
  }
} //end of CoolDecorator
```

Използване на Decorator

```
….
super ("Deco Button");
JPanel jp = new JPanel();
getContentPane().add(jp);
jp.add( new CoolDecorator(new JButton("Cbutton")));
jp.add( new CoolDecorator(new JButton("Dbutton")));
jp.add(quit = new JButton("Quit"));
quit.addActionListener(this);
```

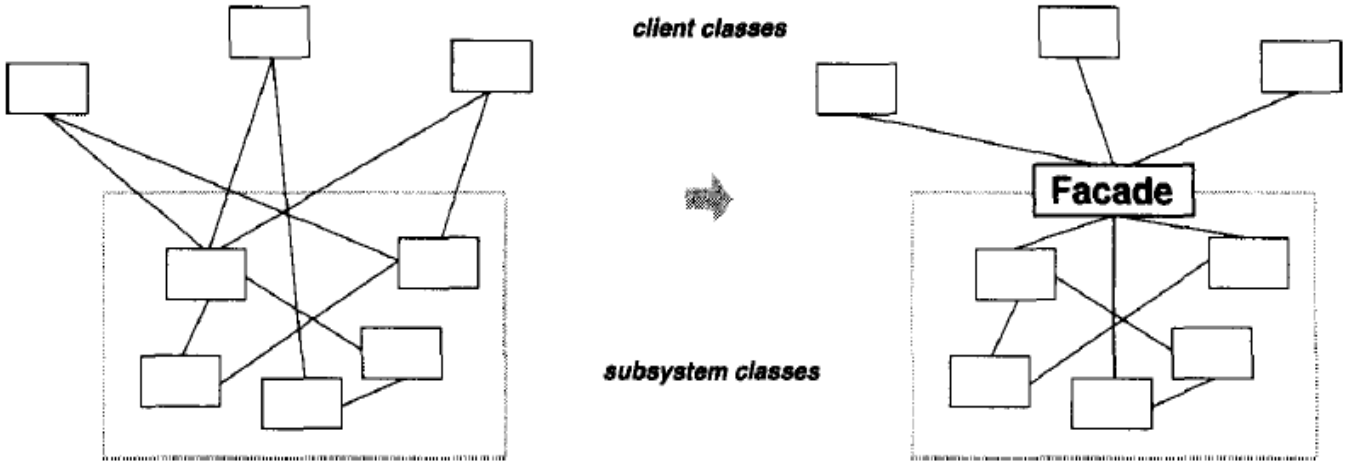
Многократно декориране


```
public class SlashDecorator extends Decorator {
    int x1, y1, w1, h1; //saved size and pos'n
    public SlashDecorator(JComponent c) {
      super(c);
      …..
    }
    //-----------------------------------------
    public void setBounds(int x, int y, int w, int h) {
      x1 = x; y1= y; //save coordinates
      w1 = w; h1 = h;
      super.setBounds(x, y, w, h);
    }
    //-----------------------------------------
    public void paint(Graphics g) {
      super.paint(g); //draw button
      g.setColor(Color.red); //set color
      g.drawLine(0, 0, w1, h1); //draw red line
    }
}
…
```

jp.add(new SlashDecorator(new CoolDecorator(new JButton("Dbutton"))));

Шаблон Фасада (Façade,Facade) [1]

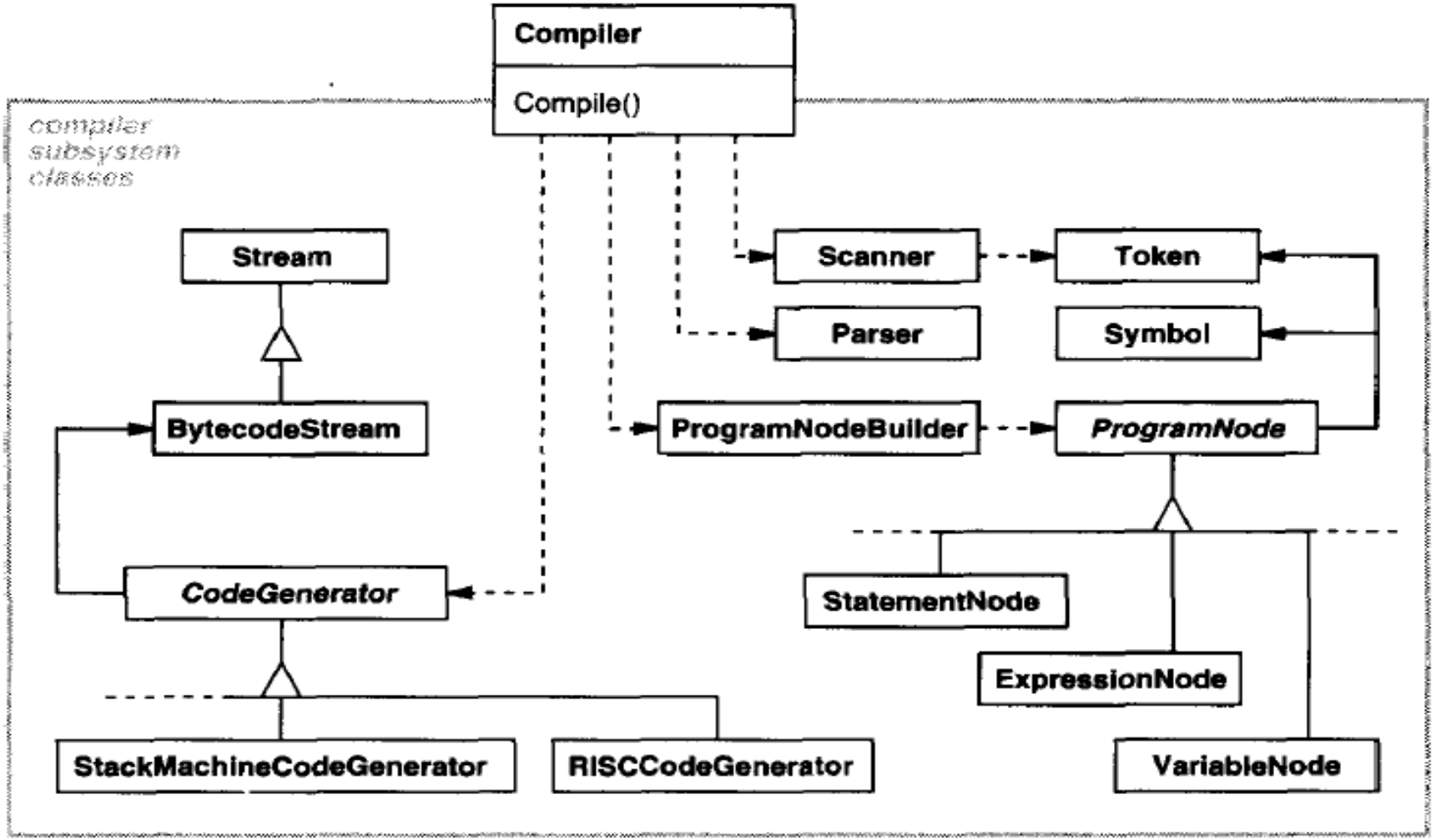
- **Цел -** осигурява единен интерфейс към набор от интерфейси в подсистема - определя интерфейс от по-високо ниво с цел по-лесно използване на подсистемата.
- **Мотивация –** цел на дизайна е да се сведе до минимум комуникацията и зависимости между подсистеми. Един от начините за постигане на това е да се въведе обект-фасада, с единен, опростен интерфейс на по-общите свойства на подсистемата.



Мотивация – Compiler Interface

- Разглеждаме приложения с достъп до подсистема-компилатор. Тя съдържа класове като Scanner, Parser, ProgramNode, BytecodeStream, и ProgramNodeBuilder, които имплементират компилатора.
- Някои специализирани приложения трябва да достъпват тези класове директно. Но за клиентите на компилатора детайли като разбор на входния текст и генериране на код не са важни - те просто искат да компилират код.
- За да предостави интерфейс от по-високо ниво, който предпазва клиентите от тези класове, компилаторната подсистема включва Compiler клас, имащ единен интерфейс към функционалността на подсистемата и по този начин действащ като фасада: тя предлага на клиентите един опростен интерфейс към компилатора като обединява заедно класове от компилатор, без да ги скрива напълно.

Решението


Compiler
Compile()
compiler subsystem classes
Stream
Scanner
Token
Parser
Symbol
BytecodeStream
ProgramNodeBuilder
ProgramNode
CodeGenerator
StatementNode
ExpressionNode
StackMachineCodeGenerator
RISCCodeGenerator
VariableNode

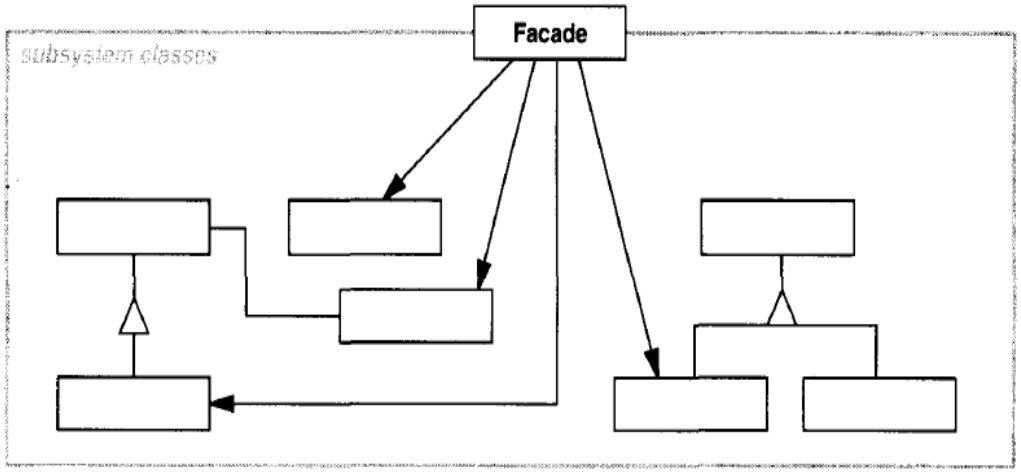
Приложимост

Използваме шаблона Фасада (Façade), ако:

- имаме нужда от прост интерфейс до комплексна под-система. Повечето шаблони при прилагането им водят до *повече и по-малки класове*. Това прави подсистемата лесна за *многократна употреба и за персонализация*, трудна за използване от клиенти, които не трябва да я персонализират. Една фасада може да осигури *прост изглед по подразбиране на подсистемата*, и то достатъчно добър за повечето клиенти. Само клиенти, които се нуждаят повече адаптивност, ще трябва да погледнат отвъд фасадата.
- има много зависимости между клиента и имплементационните класове на абстракцията. Фасадата отделя подсистемата от клиента и други подсистеми, и оттук - независимост и преносимост.
- имаме една входна точка за всяко подсистемно ниво; ако подсистемите са зависими, то те си комуникират само чрез техните фасади

Участници и структура

- Façade (Compiler)
  - - знае кои подсистемни класове ще изпълнят заявката;
  - - делегира клиентските заявки към тези класове.
- Подсистемни класове (Scanner, Parser, ProgramNode, etc.)
  - - имплементират системната функционалност;
  - - извършват работата, за1вена през обекта Façade;
  - - нямат знания за фасадата,, т.е. референция към нея.



Взаимодействия

- Клиентите комуникират с подсистемата чрез изпращане на заявки към фасадата, която ги изпраща на съответния обект(и) от подсистемата.
- Въпреки че подсистемните обекти изпълняват действителната работа, фасадата може да се наложи да работи допълнително, за да преобразува интерфейса си към подсистемните интерфейси.
- Клиенти, които използват фасадата, не трябва да достъпват директно обектите на подсистемата

Последствия и ползи

- Фасадата предпазва клиента от подсистемните компоненти, като по този начин намалява броя на обектите, с които се занимава клиентът, и прави подсистемата лесна за използване. Примери от живота ...
- Тя насърчава слабата връзка между подсистемата и нейните клиенти. Слабото свързване позволява да се варират компонентите на подсистемата, без това да повлияе на клиентите. Фасадният слой премахва сложни или циклични зависимости:
  - Намалява компилационните зависимости и така ограничава повторното компилиране.
  - Фасадата може да опрости пренасяне на системи към други платформи, защото е по-малко вероятно изграждането на една подсистема да изисква това на останалите.I
- Тя не забранява работа директно с подсистемните класове . Компромис: универсалност с/у опростяване?

Детайли на имплементацията 1/2

- 1. Намалява свързаността м/у клиента и подсистемата:
  - Тази свързаност може да се намали още чрез **представяне на Facade като абстрактен клас с конкретни подкласове с различни имплементации на подсистемата** – такава абстрактна Facade прави клиента независим от конкретната имплементация на подсистемата;
  - Алтернатива на тези подкласове е **конфигурирането на обекта Facade с различни подсистемни обекти (делегация)** . Така променяме фасадата само със замяна на тези обекти.

Детайли на имплементацията 2/2

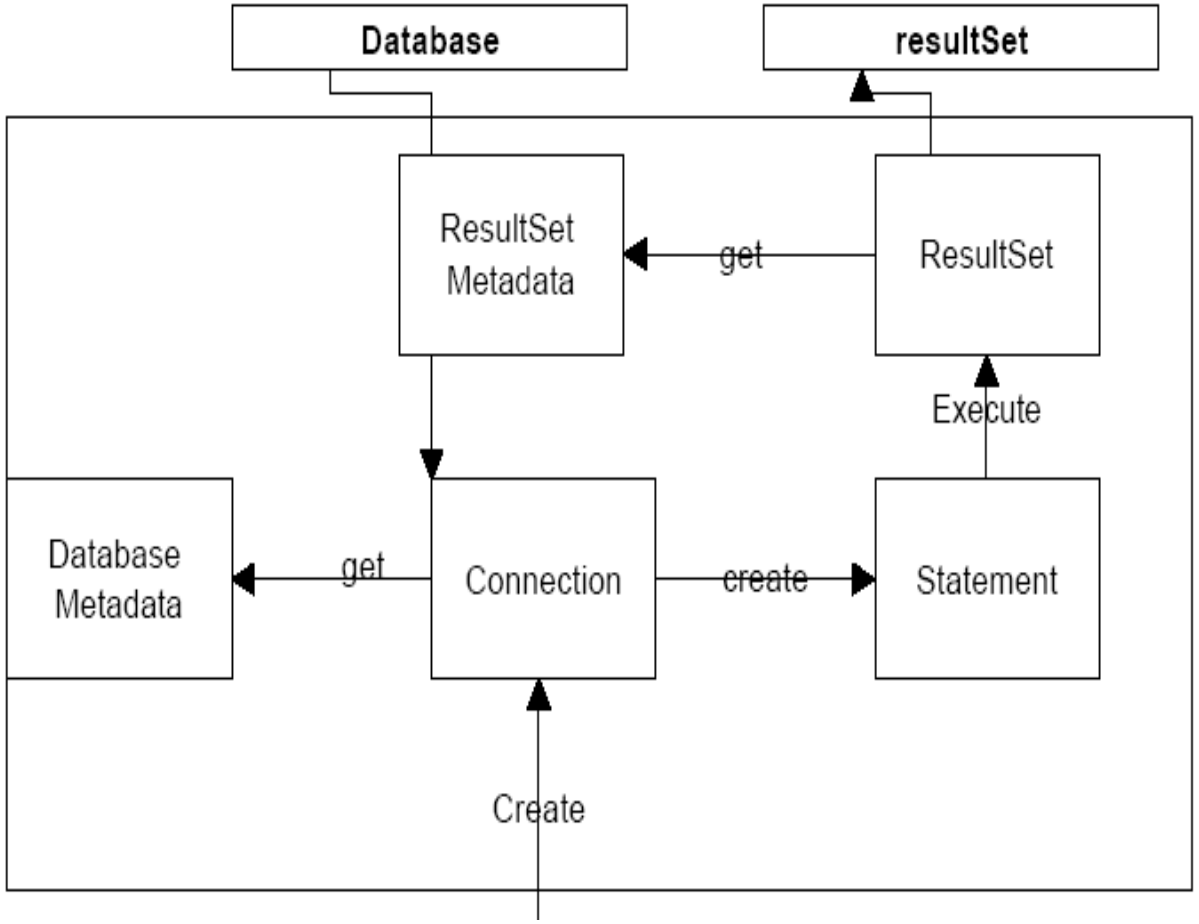
- 2. *Публични с/у частни подсистемни класове.* Подсистемата е аналогична на класа – класът капсулира състояние и операции, а подсистемата капсулира класове. Публични и частни интерфейси на подсистемата:
  - Публичният интерфейс за подсистема се състои от класове, които всички клиенти могат да достъпват; частният интерфейс е само за нейни разширения;
  - Фасадният клас е част от публичния интерфейс, но той не е единствен. Например, класът Parser от компилаторната подсистема също в част от публичния интерфейс.

Java **Façade** за JDBC класове [3]

• java.sql пакетът предоставя пример на набор от доста класове, които си взаимодействат по сложен начин
• да се свържем с базата данни, можем да използваме инстанция на класа **Connection**. След това, за да разберем имената на таблиците в базата данни и полетата им, трябва да получим копие на **DatabaseMetadata** от **Connection**.
• да създаване на заявка, ние композираме SQL query string и използваме **Connection**, за да създадем **Statement**. Чрез изпълнението на **Statement**, получаваме екземпляр на **ResultSet** клас, и за да разберем имената на колоните редове в този **ResultSet**, трябва да се получим екземпляр от класа **ResultsetMetadata**.

Решението

Чрез въвеждане на Façade състояща се от класовете **Database** и **resultSet** (забележете малката буква “r”), можем да създадем по-удобна система.



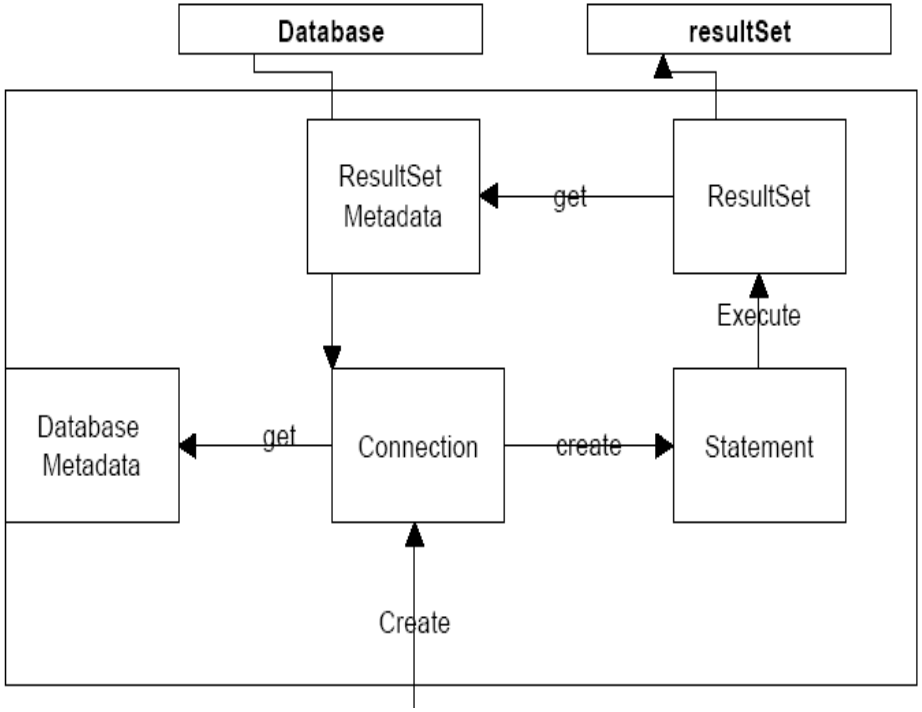
Вместо…

```
try{
        Class.forName(driver);
} //load the Bridge driver
 catch (Exception e) }
     System.out.println(e.getMessage());
}
try {      con = DriverManager.getConnection(url);
        DatabaseMetaData dma =con.getMetaData(); //get the meta data
} catch (SQLException e) {
        System.out.println(e.getMessage());
}
Vector tname = new Vector();
try {
        ResultSet results = dma.getTables( …. ));
} catch (Exception e) {System.out.println(e);}
while (results.hasMoreElements()) //gets the
        tname.addElement ( results.getString  ("TABLE_NAME") );
//next – get columns for each table…
```

… Два прости обхващащи класа, които съдържат всички значими методи на класовете Connection, ResultSet, Statement и Metadata:

```
Class Database {
    public Database(String driver()) //constructor
    public void Open(String url, String cat);
    public String[] getTableNames();
    public String[] getColumnNames(String table);
    public String getColumnValue(String table,
                                 String columnName);
    public String getNextValue(String columnName);
    public resultSet Execute(String sql);
}
```

Продължение



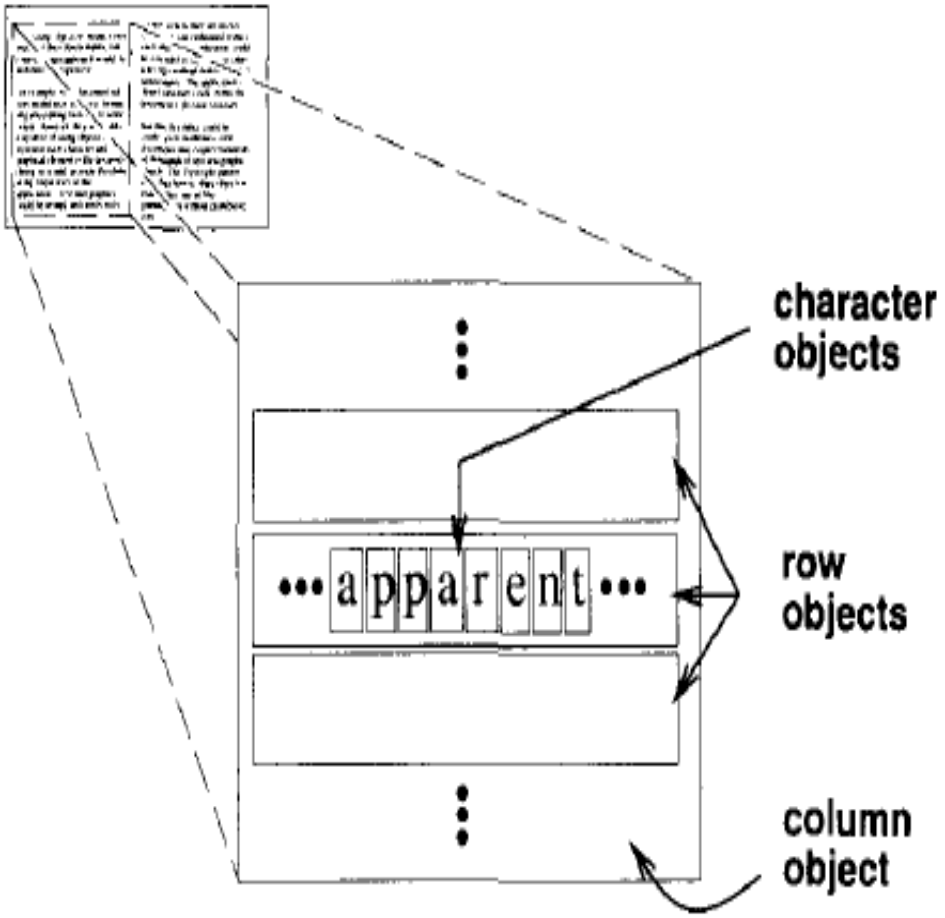

```
class resultSet {
        public resultSet(ResultSet rset) //constructor
        public String[] getMetaData();
        public boolean hasMoreElements();
        public String[] nextElement();
        public String getColumnValue(String columnName);
        public String getColumnValue(int i);
}
```

Шаблон Миниобект (Flyweight) [1]

- **Цел -** използва споделяне с цел ефективно да поддържа голям брой малки обекти.
- **Мотивация** – има случаи в програмирането, когато трябва да генерираме много голям брой екземпляри на малки класове за представяне на данните:
  - Понякога може значително *да се намали броят на различните класове*, които трябва да инстанцираме, ако забележим, че *екземплярите са едни и същи, с изключение на няколко параметъра*.
  - Ако можем да преместим тези променливи извън екземпляра на класа и да ги предаваме при извикването на метод, то броят на отделните екземпляри може да бъде значително намален.

Мотивиращ пример

- ОО редактори обикновено използват обекти, които представят вградени елементи като таблици и фигури.
- Въпреки това, те обикновено не използват обекти за всеки знак в документа, въпреки че това би повишило гъвкавостта.
- Символите и вградените елементи биха могли да бъдат третирани еднакво по отношение на това как да се изготвят и форматират.



Решението

- *ВСИЧКО Е ОБЕКТ!* - недостатък на такъв дизайн е нейната цена. Дори умерено големи документи може да изисква стотици хиляди символни обекти, които ще консумират много памет и оттам неприемлив *runtime overhead*.
- Шаблонът Миниобект (Flyweight) описва как да се споделят малки обекти, за да позволи използването им без прекомерни разходи.
- Един Flyweight е *споделен обект*, който може да се използва в различни контексти едновременно:
  - Flyweight действа като самостоятелен обект в даден контекст и е неразличим от инстанция на обект, която не е споделена.
  - Flyweight обектите не могат да правят предположения за контекста, в който оперират.

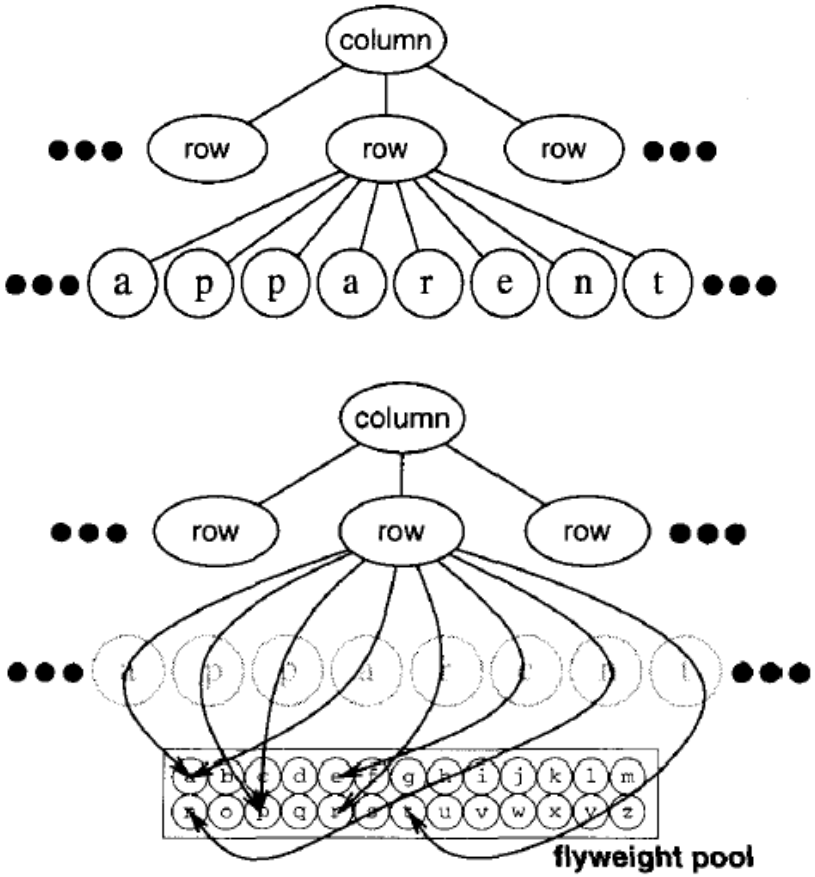
Вътрешно (intrinsic) и външно (extrinsic) състояние

- Ключовата концепция е разграничението между вътрешно (**intrinsic**) и външно (**extrinsic**) състояние.
  - Вътрешното (присъщо) състояние се съхранява в обекта Flyweight и се състои от информация, която е независима от контекста на Flyweight, като по този начин е споделено.
  - Външното (неприсъщо) състояние зависи и варира в зависимост от контекста на миниобекта и следователно не може да се споделя. Клиентските обекти са отговорни за предаване на външното състояние на Flyweight, когато има нужда от него.
- Например, редактор на документи може да създаде Flyweight за всяка буква от азбуката. Всеки Flyweight съхранява символен код на буквата (присъщо състояние), но неговата позиция в документа и типографският му стил (външно състояние) може да се определят, когато се появи.

Пример

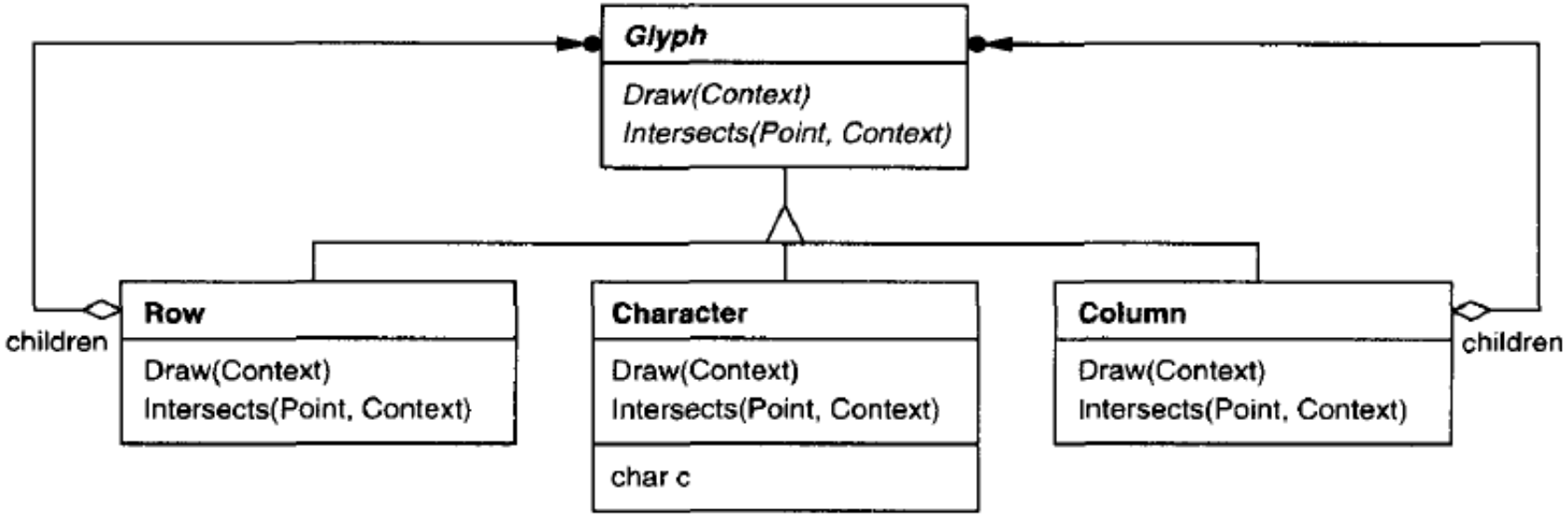
- Логически имаме обект за всяка поява на даден знак в документа →

- Физически обаче, има един общ Flyweight обект за даден символ, и той се появява в различен контекст в структурата на документа. Всяка поява на даден обект характер се отнася до същата инстанция в общ пул от Flyweight обекти →



Резултат

- Glyph е абстрактен клас за графични обекти, *<u>някои от които могат да бъдат миниобекти</u>*. Операциите, които може да зависят от външно състояние, го предават към тях като параметър. Например, Draw и Intersects трябва да знаят в кой контекст е глифът, преди да могат да си вършат работата. Клиентите предоставят контекстна информация, от която зависи изчертаването на Flyweight.
- Тъй като броят на различните символи-обекти е далеч по-малко от броя на символите в документа, общият брой на обектите е значително по-малко от този на наивното изпълнение.

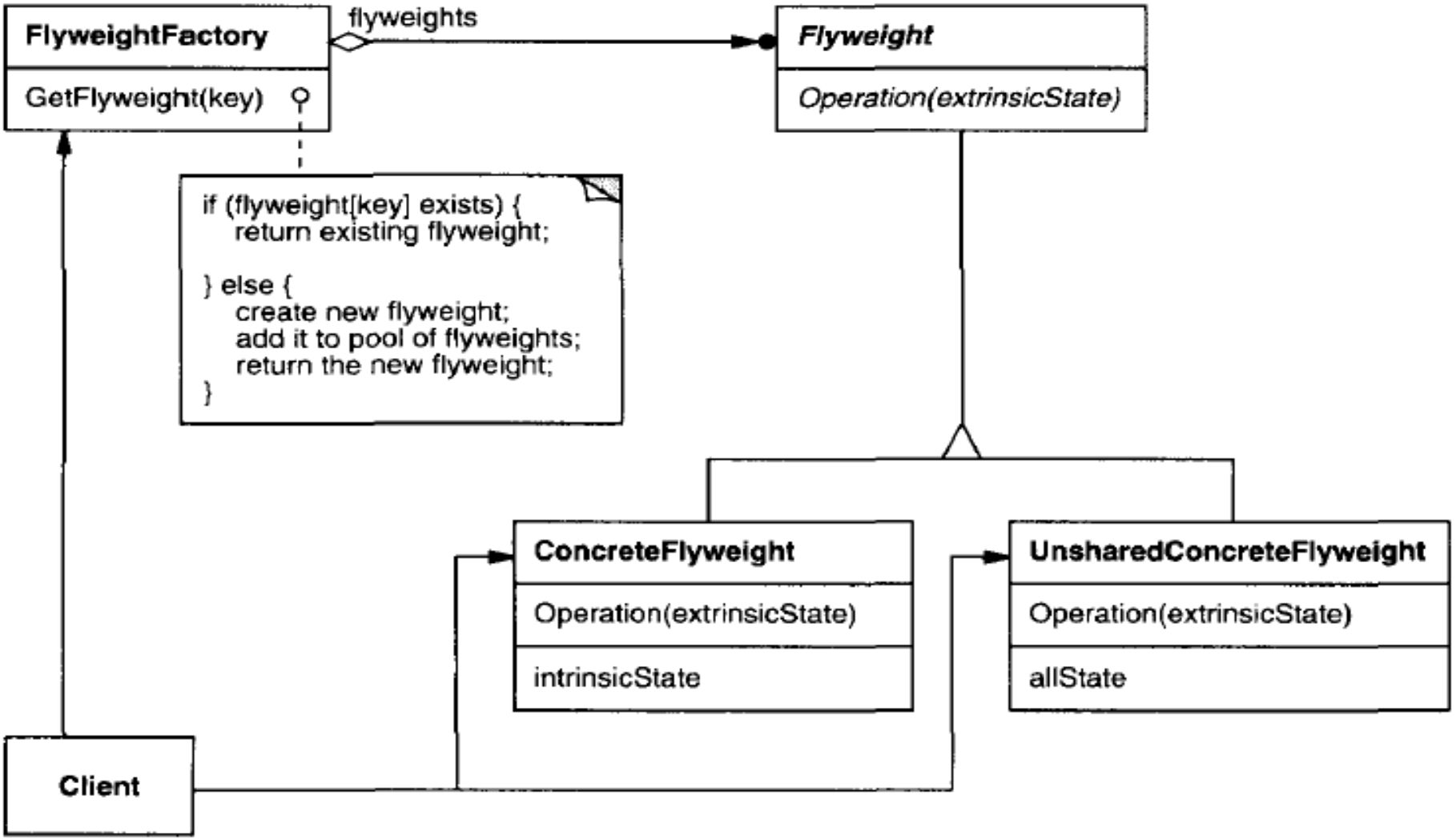


Приложимост

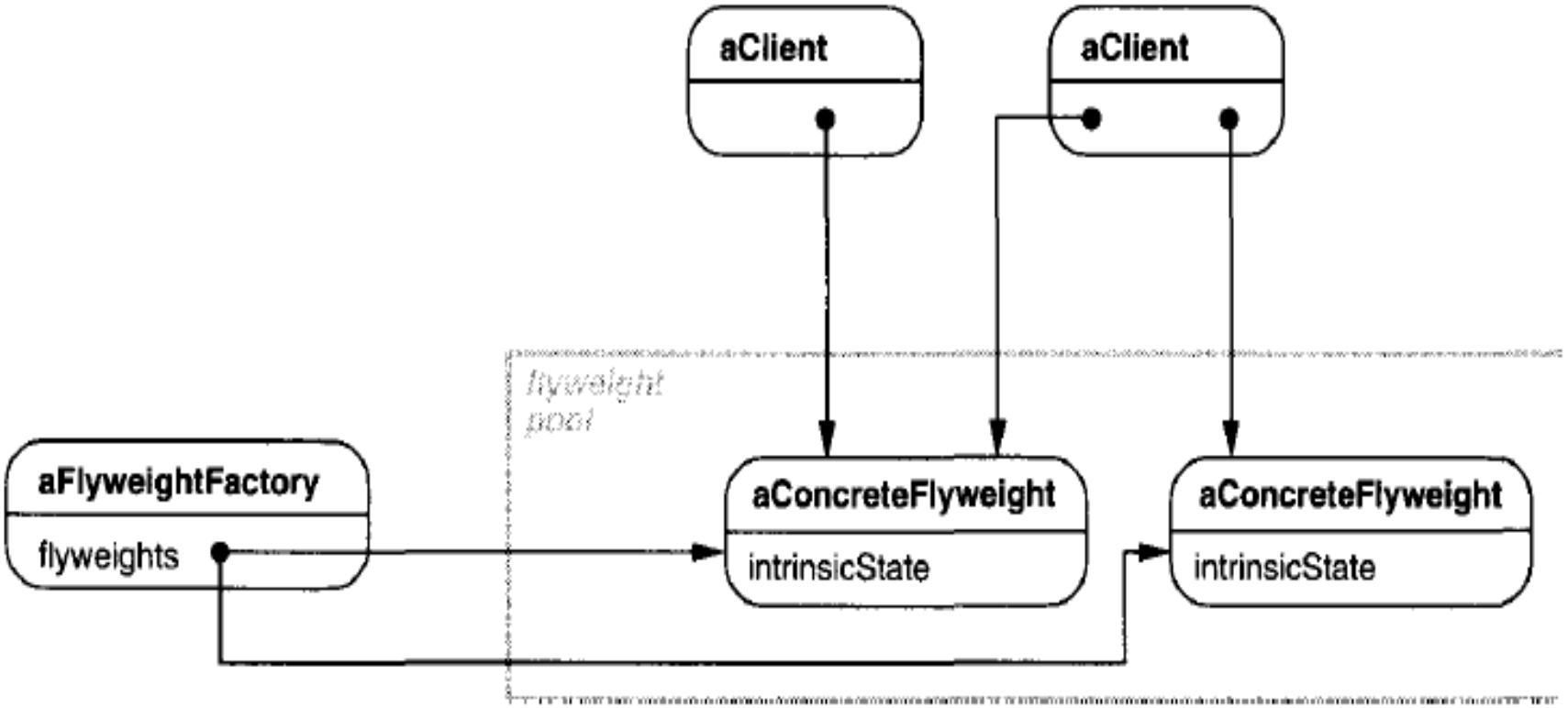
Ефективността на Flyweight зависи силно от това къде и как се използва. Използвайте шаблона Миниобект, когато са изпълнени ВСИЧКИТЕ условия по-долу:

- Приложението използва голям брой обекти.
- Разходите за съхранение на обектите са високи.
- Повечето от състоянието на обект може да се направи външно.
- Много групи от обекти могат да бъдат заменени от сравнително малко споделени обекти, веднъж след като се обособи външното състояние.
- Приложението не зависи от идентичността на обекта. Тъй като Flyweight обектите могат да бъдат споделени, тестовете за идентичност ще се върнат истина за концептуално различни обекти.

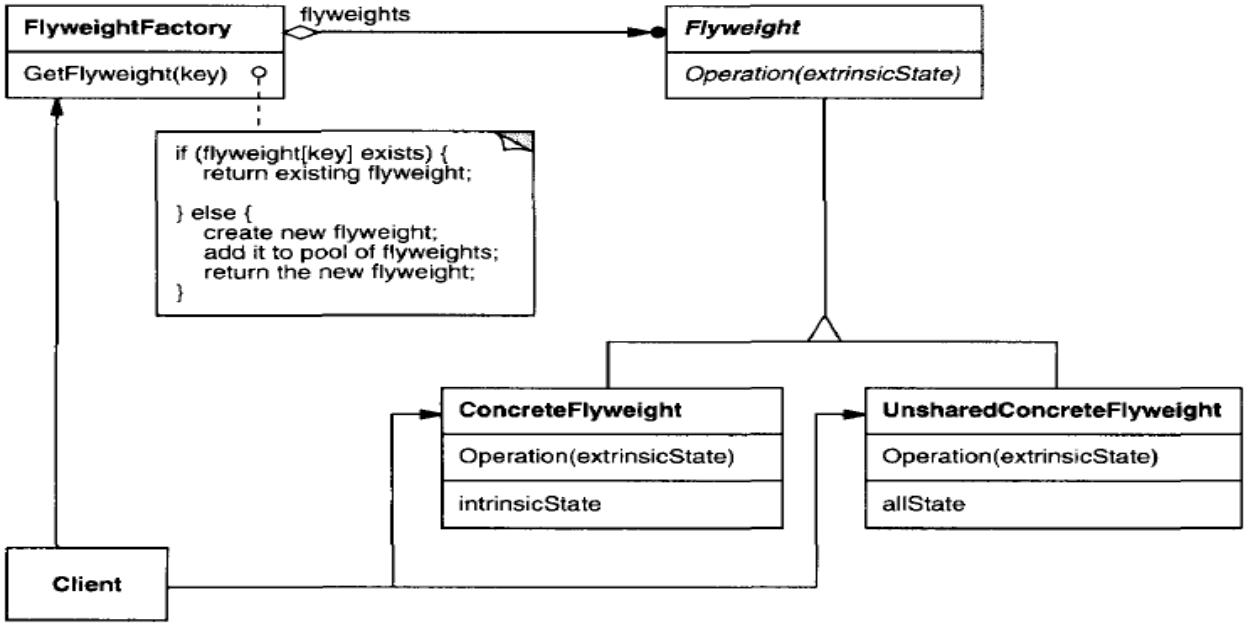
Структура


FlyweightFactory
GetFlyweight(key)
flyweights
Flyweight
Operation(extrinsicState)
if (flyweight[key] exists) {
return existing flyweight;
} else {
create new flyweight;
add it to pool of flyweights;
return the new flyweight;
}
ConcreteFlyweight
Operation(extrinsicState)
intrinsicState
UnsharedConcreteFlyweight
Operation(extrinsicState)
allState
Client

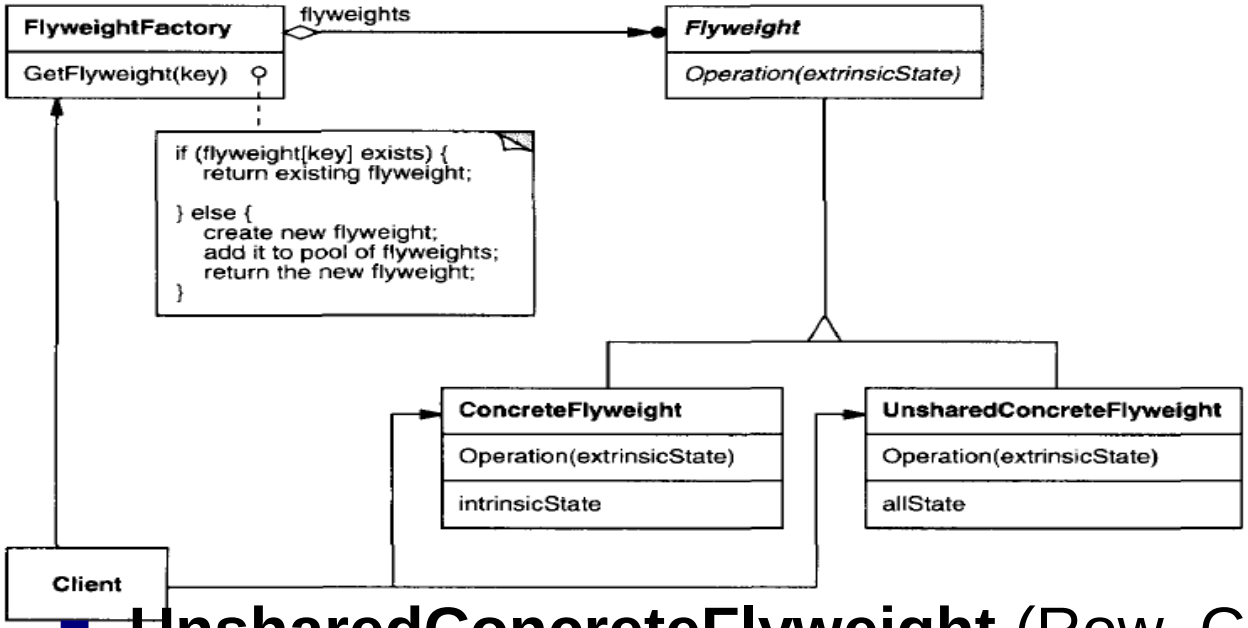
Споделяне


aClient
aClient
flyweight pool
aFlyweightFactory
flyweights
aConcreteFlyweight
intrinsicState
aConcreteFlyweight
intrinsicState

Участници 1/2



- **Flyweight** (Glyph)
  - декларира интерфейс, чрез които мини-обектите могат да получават външно състояние и действат според него.
- **ConcreteFlyweight** (Character)
  - изпълнява Flyweight интерфейса и съхранява вътрешното състояние (ако има такова). ConcreteFlyweight обектът трябва да бъде споделен. Съхраненото вътрешно състояние е независимо от контекста на ConcreteFlyweight обекта.



Участници 2/2

- **UnsharedConcreteFlyweight** (Row, Column)
  - Не всички Flyweight подкласове трябва да са споделени - Flyweight интерфейсът го разрешава, но не го налага! Често UnsharedConcreteFlyweight обектите имат за деца в структурата ConcreteFlyweight обекти (като Row и Column класове).
- **FlyweightFactory**
  - Създава и управлява миниобекти;
  - Осигурява правилното им споделяне – когато клиентът заяви миниобект, FlyweightFactory подава съществуващ екземпляр или създава такъв.
- **Client**
  - Поддържа референция към миниобектите;
  - Изчислява/запазва външното им състояние.

Взаимодействия

- Състоянието на Flyweight се характеризира като **вътрешно (присъщо) или външно**:
  - присъщото (вътрешно) състояние се съхранява в ConcreteFlyweight обекта;
  - неприсъщото (външното) състояние се съхраняват или изчислява от клиентските обекти. Клиентите предават това състояние на Flyweight, когато извикват неговите операции.

- Клиентите **не трябва директно да инстанциират ConcreteFlyweights**. Те трябва да получават ConcreteFlyweight обекти изключително от FlyweightFactory обект, за да се гарантира правилното им споделяне.

Последствия

- Миниобектите могат да ни въведат в разходи, свързани с прехвърляне, намиране или изчисляване на външни състояния, особено ако те по-рано са съхранявани като присъщи (вътрешни) състояния. Тези разходи все пак се компенсират от пестенето на памет, което се увеличава, ако все повече flyweights са споделени.
- Пестенето на памет е функция от няколко фактора:
  - **Намаляване на общия брой екземпляри (те са споделени)**
  - **Обем на вътрешното състояние на обект**
  - **Дали външното състояние се изчислява или съхранява**
- Колкото повече flyweights са споделени, толкова са по-големи икономиите за съхранение. Спестяванията се увеличават с размера на споделеното състояние.

Детайли на имплементацията 1/2

1. Изваждане на външното състояние от споделените обекти. Приложимостта на шаблона се определя от това доколко е лесно идентифицирането на външно (extrinsic) състояние и как то се премахва от споделените обекти. Премахването му не води до икономии, ако има много видове външни състояния.

2. *Управление на споделените обекти* – те не се създават директно, защото са споделени. FlyweightFactory позволява локализирането им от клиента, често с асоциативна памет за look up. В примера за документния редактор фабриката може да поддържа *таблица с миниобекти, индексирани по символни кодове.* Мениджърът връща обект по кода му или го създава, ако той не съществува.

Детайли на имплементацията 2/2

3. Споделяемостта предполага някаква форма на референтно броене (reference counting) или освобождаване на паметта (garbage collection), когато миниобектът вече не е необходим. Това не е нужно, ако броят на flyweights е фиксиран и малък (напр. flyweights за ASCII набор от символи) – тогава си заслужава да ги пазим постоянно.

4. Често Flyweight шаблонът се комбинира с Composite (GoF163), за да представи йерархична структура като граф с общи възли-листа. Резултат на споделянето е, че Flyweight възлите-листа не могат да съхраняват указател към родителя. На Flyweight тази референция се предава като част от външното му състояние.

Java пример [3]

- Необходимо е да се направи малка икона на папка с име под нея за всяко лице в една организация. Ако това е голяма организация, може да има голям брой такива икони, но те всъщност са едно и също графично изображение. Дори ако имаме две икони, една за "*is Selected*" и един за "*not Selected*", броят на различните икони е малък.
- В такава система, отделна икона-обект за всеки човек със собствените му координати, име и избрано състояние е загуба на ресурси.
- Вместо това, ние ще създадем *FolderFactory*, който връща класа с икона за избрана или неизбрана папка, но не създава допълнителни екземпляри на тези икони, след като всяка една е създадена. Тъй като това е най-обикновен случай, ще ги създадем в самото начало.

Класът FolderFactory

```
class FolderFactory {
    Folder unSelected, selected;
    public FolderFactory() {
        Color brown = new Color(0x5f5f1c);
        selected = new Folder(brown);
        unSelected = new Folder(Color.yellow);
    }
    //------------------------------
    public Folder getFolder(boolean isSelected) {
        if (isSelected)
                return selected;
        else
                return unSelected;
    }
}
```

Споделяне на класа Folder

- За случаите, когато може да съществуват повече екземпляри, фабриката може да поддържа таблица с тези, че вече е създала и да създава нови само ако те не са били вече в таблицата.

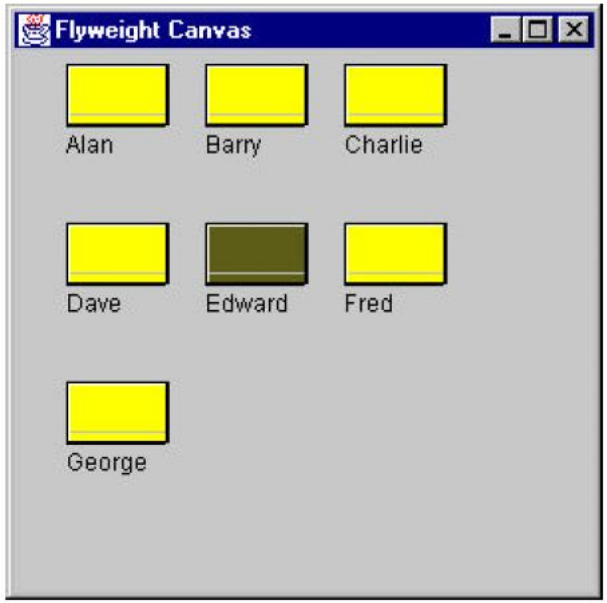
- Специфичното при използването на Flyweights тук обаче е, че предаваме координатите и името на служителя, за да бъдат показани в папката, когато я изчертаваме. Тези координати са външни данни, които ни позволяват да споделим две папки-обекти. Пълният клас Folder просто създава папка с даден цвят на фона и има публичен метод Draw, който изчертава папката на указаното място.

За да използваме Flyweight класа, основната програма трябва да изчисли позицията на всяка папка като част от paint метода и след това да предаде координатите на екземпляра на папката.

```
class Folder extends JPanel {
    private Color color;
    final int W = 50, H = 30;
    public Folder(Color c) {
        color = c;
    }
    //------------------------------
    public void Draw(Graphics g, int tx, int ty, String name) {
        g.setColor(Color.black);                    //outline
        g.drawRect(tx, ty, W, H);
        g.drawString(name, tx, ty + H+15);          //title
        g.setColor(color); //fill rectangle
        g.fillRect(tx+1, ty+1, W-1, H-1);
        g.setColor(Color.lightGray);                //bend line
        g.drawLine(tx+1, ty+H-5, tx+W-1, ty+H-5);
        g.setColor(Color.black);                    //shadow lines
        g.drawLine(tx, ty+H+1, tx+W-1, ty+H+1);
        g.drawLine(tx+W+1, ty, tx+W+1, ty+H);
        g.setColor(Color.white);                    //highlight lines
        g.drawLine(tx+1, ty+1, tx+W-1, ty+1);
        g.drawLine(tx+1, ty+1, tx+1, ty+H-1);
    }
}
```

```
public void paint(Graphics g) {
        Folder f;
        String name;
        int j = 0; //count number in row
        int row = Top; //start in upper left
        int x = Left;
        //go through all the names and folders
        for (int i = 0; i<names.size(); i++) {
                name = (String)names.elementAt(i);
                if(name.equals(selectedName))
                        f = fact.getFolder(true);
                else
                        f = fact.getFolder(false);
                //have that folder draw itself at this spot
                f.Draw(g, x, row, name);
                x = x + HSpace; //change to next posn
                j++;
                if (j >= HCount) { //reset for next row
                        j = 0; row += VSpace;
                        x = Left;
                }
        }
}
```

Можем да генерираме масив или вектор с папките в самото начало и просто да сканираме през масива. Това не е толкова разточително, защото това е всъщност масив с референции към един от екземплярите на двете папки.



Шаблон Пълномощник (Proxy) [1]

- **Цел –** предоставя заместител (*surrogate*) или контейнер (*placeholder*) за друг обект с цел контрол на достъпа до него.
- **Познат още като -** Surrogate
- **Мотивация** - една от причините за контролиране на достъпа до даден обект е да се отложи неговото създаване и инициализация, докато момента на използването му. Да разгледаме документен редактор, който вгражда графични обекти в документа. Някои графични обекти, като големи растерни изображения, се създават/отварят бавно. Но отварянето на документа трябва да бъде бързо, така че трябва да се избегне създаването на всички «скъпи» обекти наведнъж, още повече ако това не е нужно.

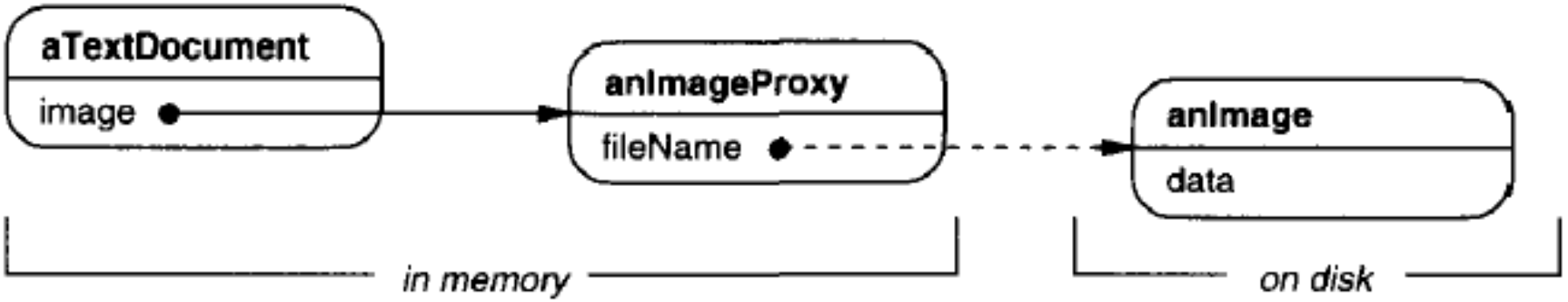
Предложение

Тези ограничения ще предположат създаването на всеки «скъп» обект *при поискване*, което в този случай е тогава, когато изображението се зареди.

- Но какво ще се постави в документа на мястото на изображението?
- И как можем да скрием факта, че образът е създаден *при поискване*, така че да не се усложни изпълнението на редактора? Тази оптимизация не трябва да повлияе на кода за представяне и форматиране на обекта.

Решение 1/2

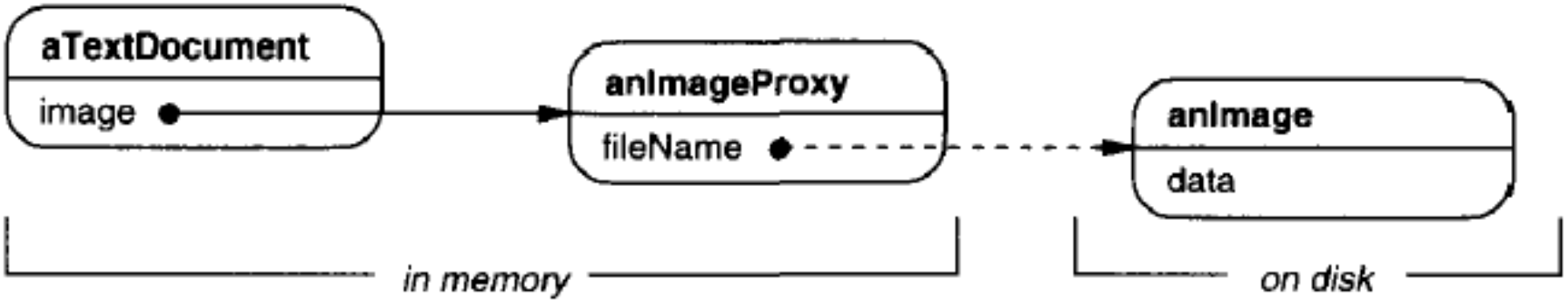
- Решението е да се използва друг обект, **пълномощник** на изображението, който действа като заместител на реалния образ. Пълномощникът действа точно както изображението и се грижи да го инстанциира, когато това е необходимо.
- Пълномощникът (прокси) създава реално изображение само когато документният редактор изисква от него да се покаже, като за целта извиква Draw операция. Пълномощникът препредава следващите заявки директно към изображението. Следователно той трябва да запази референция към изображението, след като го създаде.



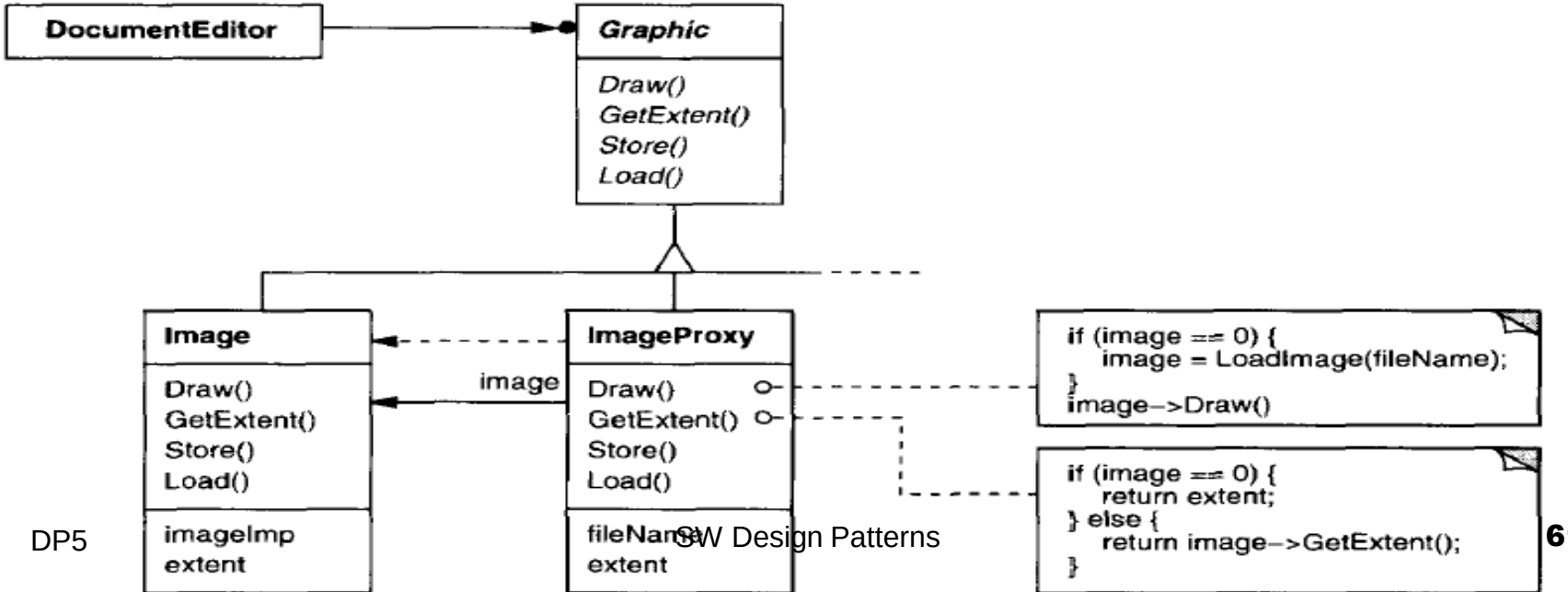
Решение 2/2

- Ако изображенията се съхраняват в отделни файлове, можем да използваме името на файла като препратка към реален обект.

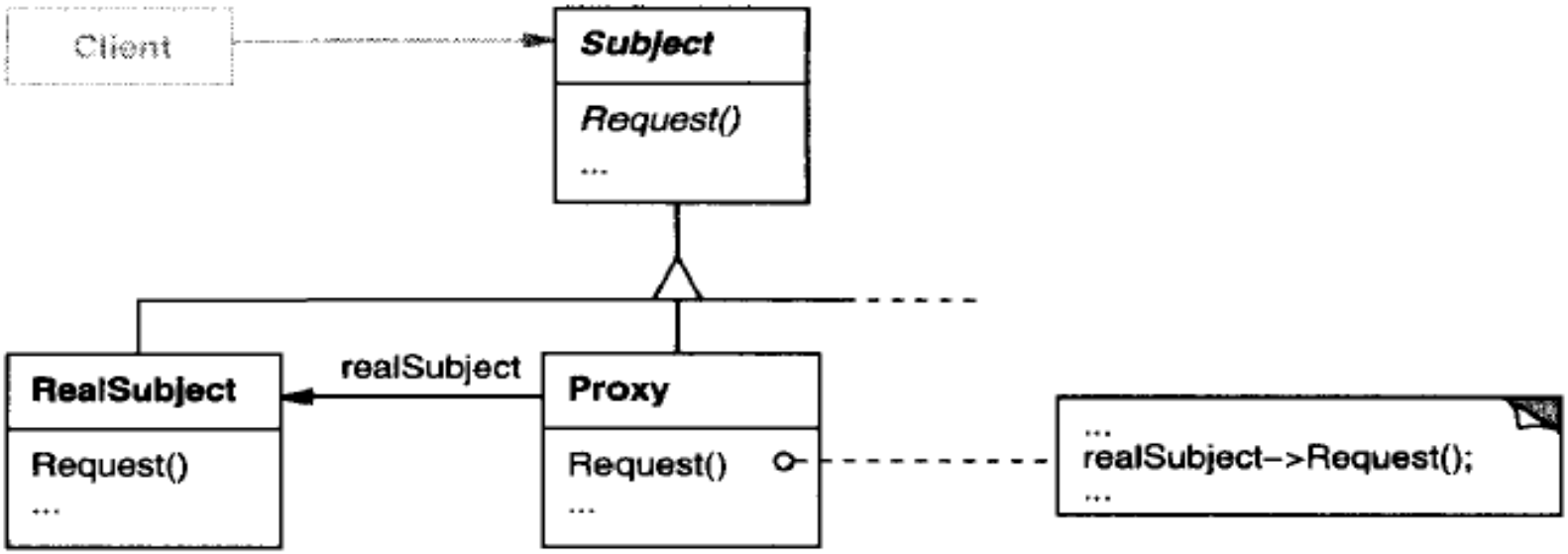
- Пълномощникът съхранява още и неговия размер (**extent**) - ширина и височина. Размерът позволява на прокси да отговори на заявки за размера на изображението, без всъщност да инстанциира картинката.



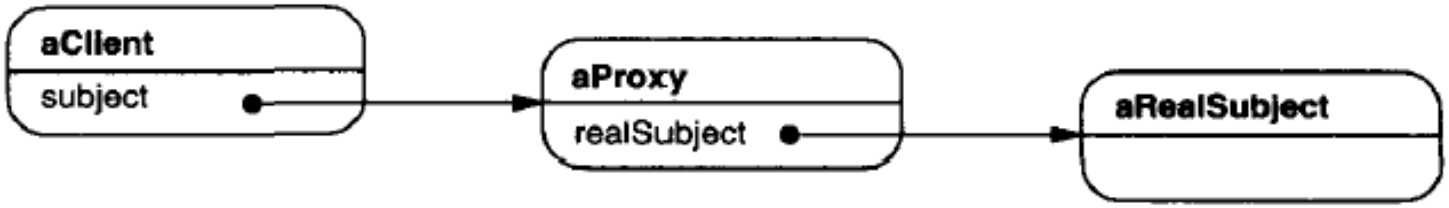
- Редакторът достъпва вградените изображения чрез интерфейса, определен от абстрактния клас ***Graphic***.
- **ImageProxy** е клас за изображения, които са създадени по поръчка (on demand). ImageProxy поддържа името на файла като референция към изображението на диска - предадено като аргумент на конструктора на ImageProxy. **ImageProxy** съхранява очертаващ правоъгълник на изображението и референция към реалния екземпляр на изображение - тя няма да бъде валидна, докато прокси не създаде реалния образ. **Draw** операцията осигурява създаването на изображението преди предаване на заявки към него.
- **GetExtent** препредава заявки само ако изображението е инстанциирано; иначе **ImageProxy** връща extent от екрана.



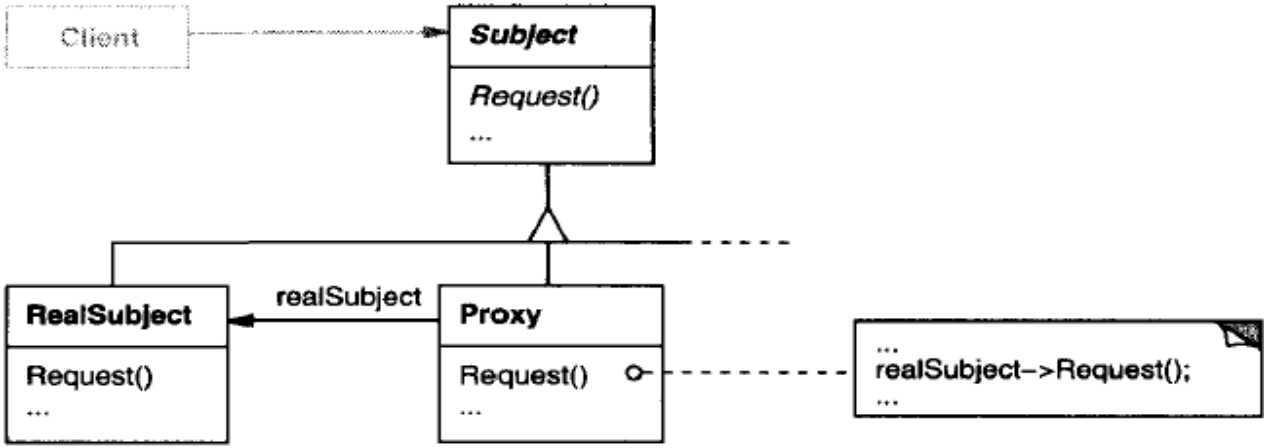
Структура



Обектна диаграма на прокси структура (run-time) ↓ :



Участници



- **Proxy** (ImageProxy) - поддържа връзка, която позволява на прокси да достъпва реалния субект, като:
  - може да реферира **Subject**, ако **RealSubject** и **Subject** имат едни и същи интерфейси;
  - предоставя интерфейс идентичен на **Subject**, така че прокси да може да бъде заместен от реалния субект;
  - контролира достъпа до реалния обект и може да бъде отговорен за създаването и изтриването му.
- **Subject** (Graphic) – дефинира общ интерфейс за RealSubject и Proxy, така че Proxy да може да се ползва навсякъде, където се очаква RealSubject.
- **RealSubject** (Image) – дефинира реален обект, който се представя от пълномощника.

Отговорности на пълномощника

Прокси предава заявките към RealSubject, когато е необходимо, в зависимост от вида си. Отговорностите на пълномощника също зависят от вида на прокси :

- ***remote proxies*** са отговорни за кодиране на заявката и нейните аргументи и за изпращане на кодираната заявка към реалния субект в *чуждо адресно пространство*.
- ***virtual proxies*** *може да кешират допълнителна информация за реалния субект, така че те могат да отложат достъпа до него. Например, ImageProxy кешира* extent на *реалното изображение.*
- ***protection proxies*** проверяват дали извикващият обект има разрешенията за достъп, необходими за изпълнение на заявката.

Приложимост 1/2

Proxy е приложим, когато е необходимо по-гъвкаво или по-сложно позоваване на обект, отколкото обикновения указател:

- **1. Отдалеченият пълномощник (remote proxy)** предоставя локален представител на обекта в различно адресно пространство.
- **2. Виртуалният пълномощник (virtual proxy) създава** скъпи обекти при поискване. ImageProxy от мотивацията е пример за такова proxy.

Приложимост 2/2

3. Защитен пълномощник (protection proxy) контролира достъпа до оригиналния обект. Защита на пълномощника е полезна, когато обектите трябва да имат различни права за достъп - например, за защитен достъп до обекти на операционната система.

- **4. Умна референция (*Smart reference)*** - заменя обикновения указател с допълнителни действия по време на достъп до обекта, напр.:
  - Броене на референциите до обекта с цел garbage collection
  - Зареждане на персистентен обект в паметта при първото му рефериране;
  - Проверка, че истинският обект е заключен преди достъп до него, за да се гарантира, че никой друг обект може да го промени – critical sections

Последствия

A) Proxy шаблонът въвежда **опосредстван достъп до обекта**:

1. Дистанционното прокси може да скрие факта, че обектът се намира в различно адресно пространство.
2. А виртуалният пълномощник може да извършва оптимизации като например създаване на обекта при поискване.
3. Както защитното прокси, така и умните референции позволяват допълнителни задачи по поддръжката, когато един обект е достъпен.

B) Има и друга оптимизация, която Proxy шаблонът може да скрие от клиента. Тя се нарича **copy-on-write**, и това е свързано със създаването по поръчка. Копирането на голям и сложен обект може да бъде скъпа операция. Ако копието никога не се променя, тогава няма нужда да поема този разход. Чрез използването на прокси да отложи процес на копиране, ние гарантираме, че сме платили цената за копиране на обект, само ако той се модифицира.

Детайли на имплементацията

- *Proxy не винаги трябва да знае вида на реалния обект.* Ако Proxy клас може да се справи със своя субект единствено чрез абстрактен интерфейс, тогава няма нужда да се прави Proxy клас за всеки клас RealSubject; пълномощникът може да третира всички RealSubject класове унифицирано. Но ако Proxies ще инстанцииат RealSubjects (като при виртуалното прокси), а след това те трябва да познават конкретния клас => отделни прокси.

- *Как да се позове на субекта, преди той да е инстанциран* - някои прокси трябва да се отнасят до субекта им независимо дали е на диска или в паметта. Това означава, че те трябва да използват някаква форма за адресиране в пространство – напр. независим идентификатор на обекта (използвахме името на файла за тази цел в мотивацията).

Java пример [3]

В този пример създаваме програма за показване на изображение върху **JPanel** тогава, когато то се зареди. Вместо зареждане на изображение директно, ние използваме клас, наричен ImageProxy, да се отложи зареждането и очертае правоъгълник около изображението, докато приключи зареждането.

```
public class ProxyDisplay extends JxFrame {
    public ProxyDisplay() {
        super("Display proxied image");
        JPanel p = new JPanel();
        getContentPane().add(p);
        p.setLayout(new BorderLayout());
        ImageProxy image = new ImageProxy(this, "elliott.jpg",
        2321, 2271);
        p.add("Center", image);
        setSize(2400,2400);
        setVisible(true);
} …..
```

Класът ImageProxy подготвя зареждането на образа и създава MediaTracker обект, за да следи процеса на зареждане в конструктора:

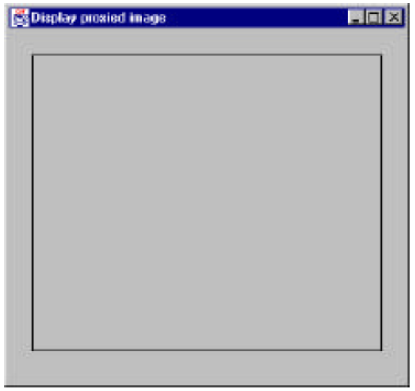
```
public ImageProxy(JFrame f, String filename, int w, int h) {
    height = h;
    width = w;
    frame = f;
    tracker = new MediaTracker(f);
    img = Toolkit.getDefaultToolkit().getImage(filename);
    tracker.addImage(img, 0); //watch for image loading
    imageCheck = new Thread(this);
    imageCheck.start(); //start 2nd thread monitor
    //this begins actual image loading
    try{
        tracker.waitForID(0,1);
    }
    catch(InterruptedException e){ … }
}
```

Методът *waitForID* на MediaTracker всъщност започва зареждането. В този случай, ние задаваме минимално време за изчакване от 1 милисекунда, така че можем да намалим видими забавяния на програмата.
Конструкторът създава отделна imageCheck нишка, която проверява състоянието на зареждането на всеки няколко милисекунди.

```
public void run() {
    //this thread monitors image loading
    //and repaints when the image is done
    try{
        Thread.sleep(1000);
        while(! tracker.checkID(0))
                Thread.sleep(1000);
    }
    catch(Exception e){ … }
    repaint();
}
```

За целите на тази илюстрация програма, polling time е забавено до 1 секунда, така че можем да видим как програмата чертае правоъгълник и след това обновява окончателното изображение.

Накрая, Proxy е дериват на JPanel компонент и поради това има *paint* метод. При този метод, чертаем правоъгълник, ако изображението не е заредено, а в противен случай – изтриваме правоъгълника със самото изображение.




```
public void paint(Graphics g) {
    if (tracker.checkID(0)) {
        height = img.getHeight(frame); //get height
        width = img.getWidth(frame); //and width
        g.setColor(Color.lightGray); //erase box
        g.fillRect(0,0, width, height);
        g.drawImage(img, 0, 0, frame); //draw image
    } else {
        //draw box outlining image if not loaded yet
        g.drawRect(0, 0, width-1, height-1);
    }
}
```

Вижте статията:

- **Non-Software Examples of Software Design Patterns,** by Michael Duell, in AG Communication Systems e-zine:

  http://www2.ing.puc.cl/~jnavon/IIC2142/patexamples.htm