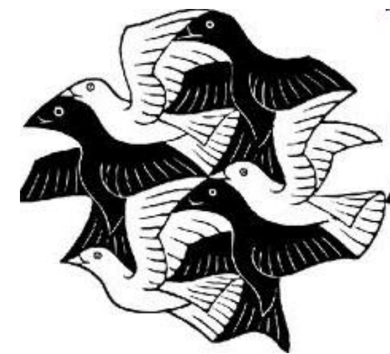


3. Създаващи шаблони

Софтуерни шаблони за проектиране

Боян Бончев,

ФМИ – Софийски университет

© 2006/2017

Анотация

- Създаващи шаблони
- Определения
- Свойства
- 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

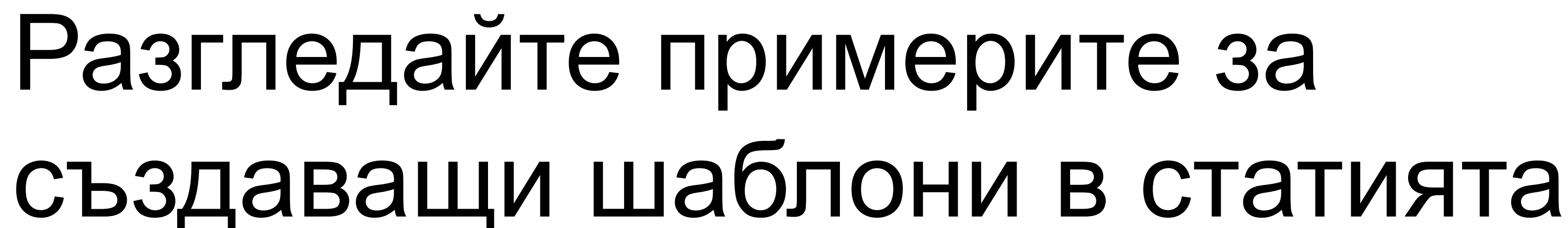
Класификация на шаблоните за проектиране

- Създаващи шаблони
  - Резюмират процеса на инстанцииране на обекти
  - Правят използваната система по-малко зависима от нейната реализация
  - Създаващите шаблони, ориентирани към класове, използват наследяване, за да променят инстанция на класа
  - Създаващите шаблони, ориентирани към обекти, делегират инстанциирането към друг обект

*В Java, най-лесният начин за създаването на обект е чрез директното използване на оператора **new**.*

```
Fred  f = new Fred(); //instance of Fred class
```

Това обаче води до т.нар. hard coding, в зависимост от това как създаваме обекта в рамките на програмата. В много случаи, точното естество на обекта, който се създава, може да варира в зависимост от нуждите на програмата. Така абстрахирането на процеса на създаване на специален клас "създател" може да направи кода по-гъвкав и общ.



- **Non-Software Examples of Software Design Patterns,** от Michael Duell, в AG Communication Systems e-zine, 2007

Пет създаващи шаблона

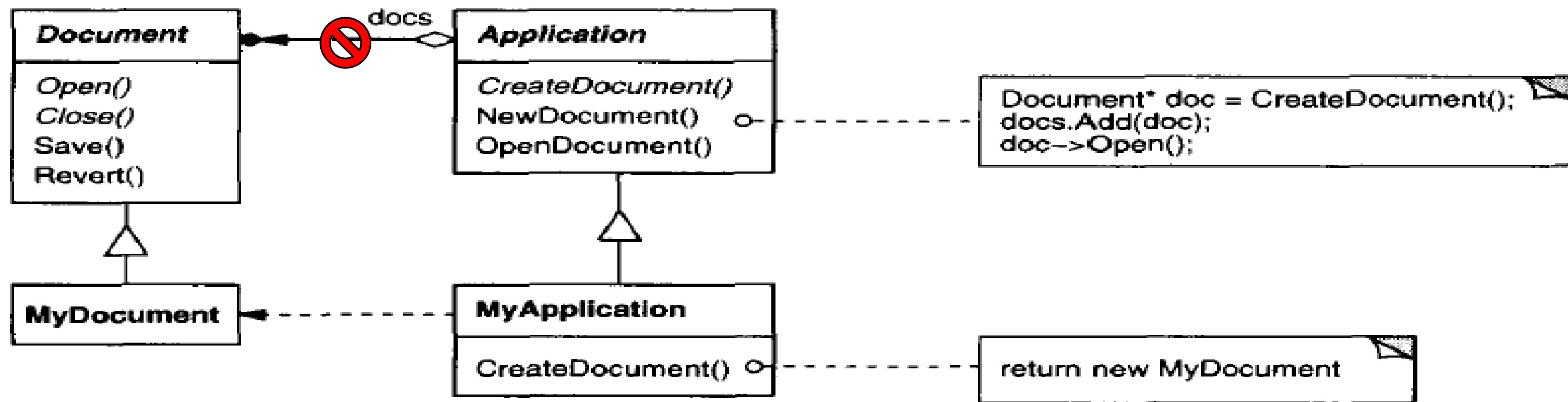
- **Метод фабриката** осигурява прост клас за вземане на решения, който връща един от няколко възможни подкласове на абстрактния базов клас, в зависимост от данните, които се използват.
- **Абстрактната фабрика** предоставя интерфейс за създаване и връщане на едно от няколко семейства от свързани обекти.
- **Шаблонът строител (Builder)** разделя изграждането на сложен обект от неговото представяне, така че могат да бъдат създадени няколко различни представяния в зависимост от нуждите на програмата.
- **Шаблонът Прототип** започва с една инициализирана инстанция (екземпляр) на класа, като я копира или клонира, за да създаде нови екземпляри.
- **Шаблонът Сек (Singleton)** е клас, от които може да има не повече от един екземпляр. Той осигурява единна глобална точка на достъп до тази инстанция.

Метод фабрика (Factory Method) [1]

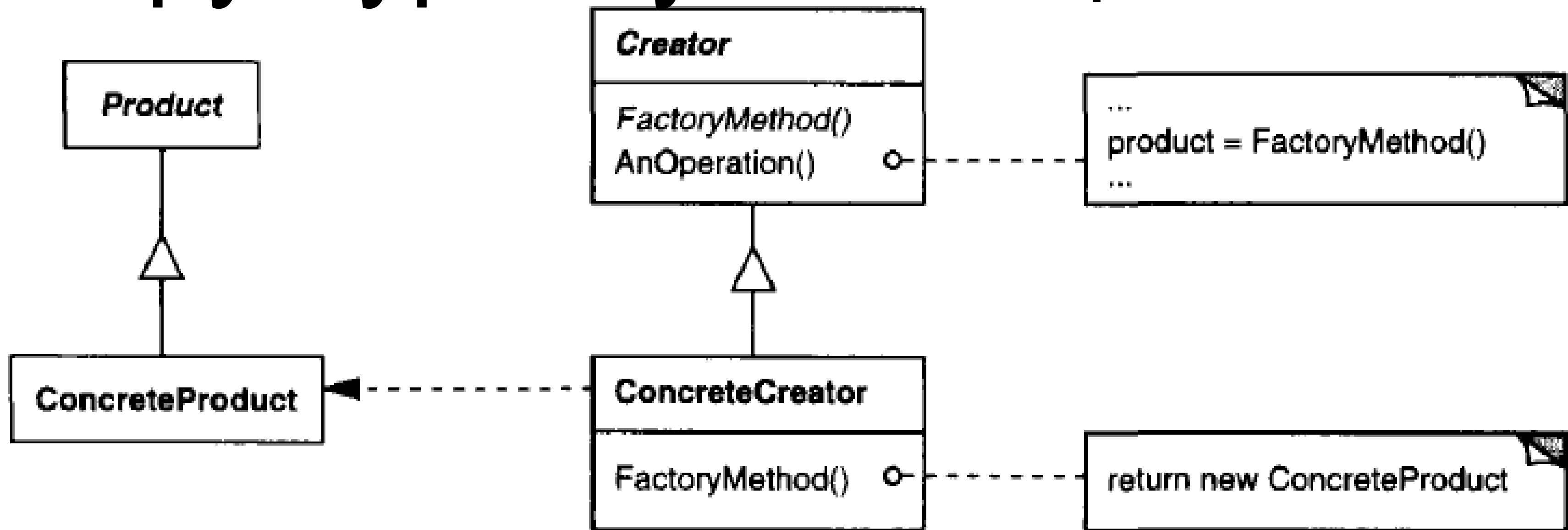
- **Цел:** дефинира интерфейс за създаване на обект, но позволява на подкласовете да решат кой клас трябва да инстанциира обекта. Метод фабрика позволява един клас отложи инстанциирането до подкласове.
- **Познат още като:** Virtual Constructor
- **Мотивация:** Разглеждаме *приложна рамка* която презентира на потребителя различни документи. Двете главни абстракции на рамката са *abstract* класовете **Application** и **Document**. Класът Application е отговорен за управляване на документи и може да ги създава по заявка. Понеже създаването и инстанциирането на определен под-клас на Document е зависимо от приложенията, класът Application не може да предскаже какъв точно под-клас на Document трябва да се инстанциира.

Дилемата и нейното решение

- Дилема: рамката трябва да инстанциира класове, но тя познава само абстрактни класове, които не може да инстанциира.
- Шаблонът Метод фабрика предлага решение на проблема. **Той капсулира знанието за това какъв подклас Документ трябва да се създаде, и премества това знание извън рамката.**

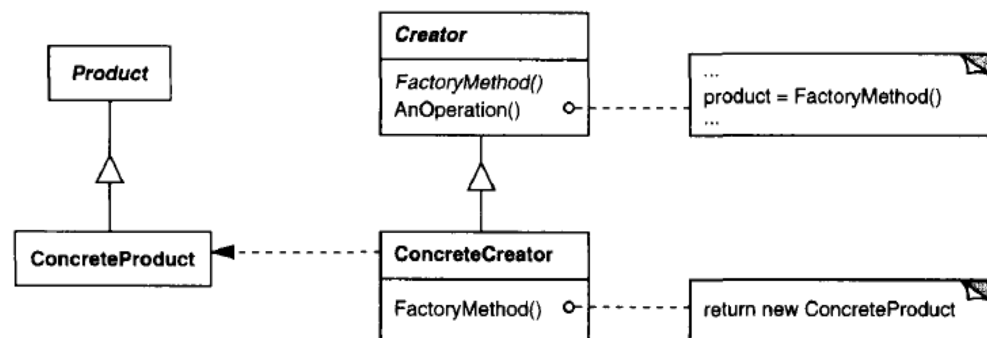


Структура и участници



- **Product** (Document) - дефинира интерфейса на обекти, който метод фабриката създава .
- **ConcreteProduct** (MyDocument) – имплементира интерфейса Product.
- **Creator** (Application) – (1) декларира метод фабрика, връщащ обект от тип Product; *може също да дефинира имплементация по подразбиране на метод фабриката, която връща ConcreteProductобект по подразбиране; (2) може да извика* метод фабриката, за да получи обект от тип Product.
- **ConcreteCreator** (MyApplication) – предефинира метод фабриката, за да върне инстанция на ConcreteProduct.

Приложимост



Използвайте шаблона *Factory Method* когато:

- даден клас не може да предвиди класа на обектите, които трябва да се създаде;
- даден клас желае неговите подкласове да определят обектите, които той създава;
- класове делегират отговорността към един от няколко помощни подкласа, и искаме да локализираме знанието за това от какъв помощен подклас е делегатът.

Последици 1/2

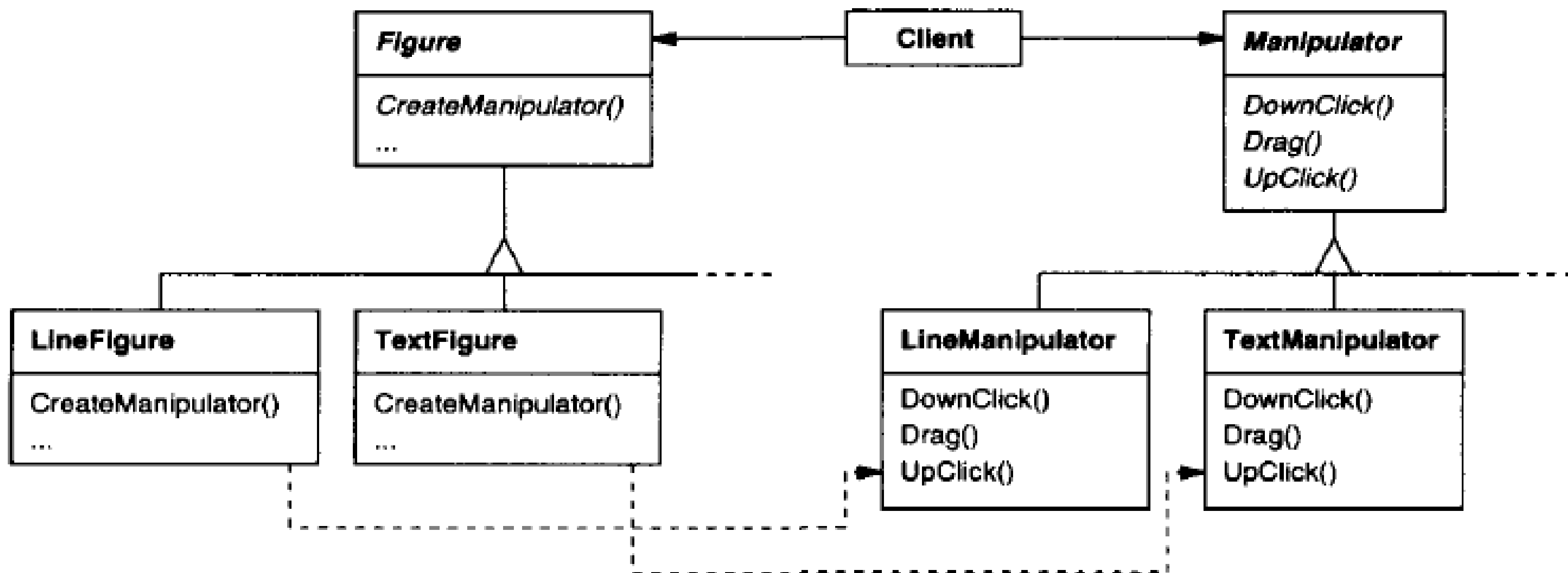
- Factory method премахва *необходимостта да се обвързваме в кода ни със специфични за приложението класове*. Кодът се занимава само с *интерфейса* Product, следователно, той може да работи с всеки дефиниран от потребителя ConcreteProduct клас.

- Потенциален недостатък на метод фабриката е, че на клиентите да се наложи да направят подклас на класа *Creator просто за да създадат конкретен ConcreteProduct обект*. Създаването на подкласове е приемливо, когато на клиента му трябва подклас на Creator така или иначе, но в противен случай клиентът трябва да се справи по друг начин => параметризация!

- *Осигурява т. нар. "куки" (hooks) за подкласове. Създаването на обекти вътре в клас с метод фабрика е винаги по-гъвкаво от създаването на обект директно.* Метод фабриката дава "кука" за подкласове за предоставяне на разширена версия на даден обект.

Последици 2/2

- *Свързва паралелни йерархии класове* - клиент може да намери фабричните методи за полезни, особено в случай на паралелни йерархии на класове. Например, различни фигури могат да използват различни подкласове на Manipulator, за да се справят с конкретни взаимодействия.



Имплементация 1/2

- *Два варианта:*
  - (1) случай, когато ***Creator класът е абстрактен и не предоставя имплементация за метод фабриката,*** която той декларира,

 и

  - (2) случай, когато ***Creator класът е конкретен и предоставя имплементация по подразбиране за метод фабриката.***

Имплементация 2/2

- *Параметризиран фабричен метод - друг вариант на модела позволява метод фабриката да създава множество видове продукти. Метод фабриката получава параметър, който определя вида на обекта с цел да се създаде. Всички обекти, които метод фабриката създава, ще споделят интерфейса Product.*

```
class Creator {
    public: virtual Product* Create(Productid) ;
};
Product* Creator::Create (Productid id) {
    if (id == MINE) return new MyProduct;
     if (id == YOURS) return new YourProduct;
     // repeat for remaining products...
     return 0;
};
```

Java пример [3]

```
class Namer {
    //a simple Product class to take a string apart into two names
    protected String last; //store last name here
    protected String first; //store first name here
    public String getFirst() {
            return first; //return first name
    }
    public String getLast() {
            return last; //return last name
    }
}
```

```
class FirstFirst extends Namer { //split first last
    public FirstFirst(String s) {
     int i = s.lastIndexOf(" "); //find sep space
     if (i > 0) {
       //left is first name
       first = s.substring(0, i).trim();
       //right is last name
       last =s.substring(i+1).trim();
     } else {
       first = “”; // put all in last name
       last = s; // if no space
     }
    }
}
```

```
class LastFirst extends Namer {
    //split last, first
  public LastFirst(String s) {
    //find comma
    int i = s.indexOf(",");
    if (i > 0) {
      //left is last name
      last = s.substring(0, i).trim();
      //right is first name
      first = s.substring(i + 1).trim();
    } else {
      last = s; // put all in last name
      first = ""; // if no comma
    }
  }
}
```

Продължение

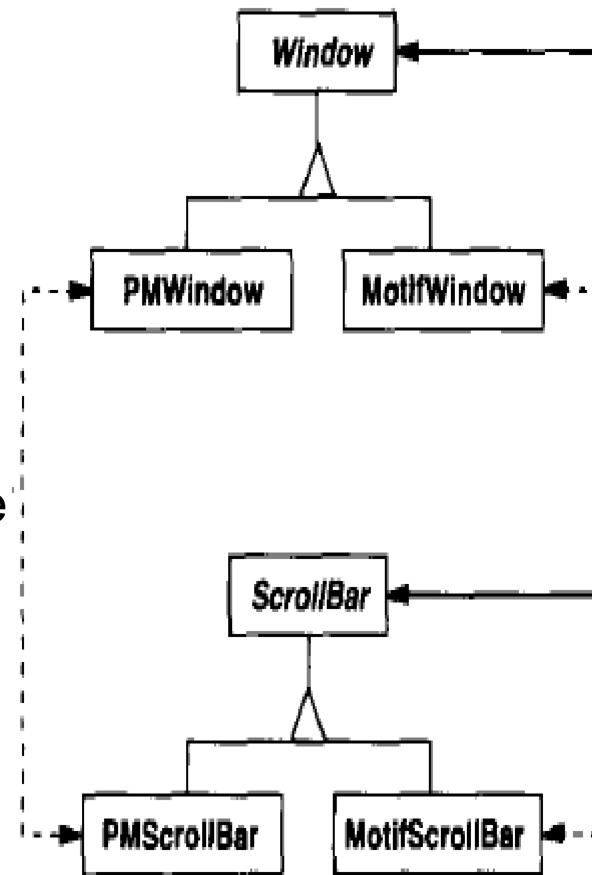
Фабриката е изключително проста. Ние просто тестваме за съществуването на запетая и след това се връщаме инстанция на единия или другия клас:

```
class NameFactory {
    //returns an instance of LastFirst or FirstFirst
    //depending on whether a comma is found
    public Namer getNamer(String entry) {
        int i = entry.indexOf(","); //comma determines name order
        if (i>0)
            return new LastFirst(entry); //return one class
        else
            return new FirstFirst(entry); //or the other
    }
}
```

```
//usage:
NameFactory nfactory = new NameFactory();
namer = nfactory.getNamer(entryField.getText());
//compute the first and last names using the returned class
txFirstName.setText(namer.getFirst());
txLastName.setText(namer.getLast());
```

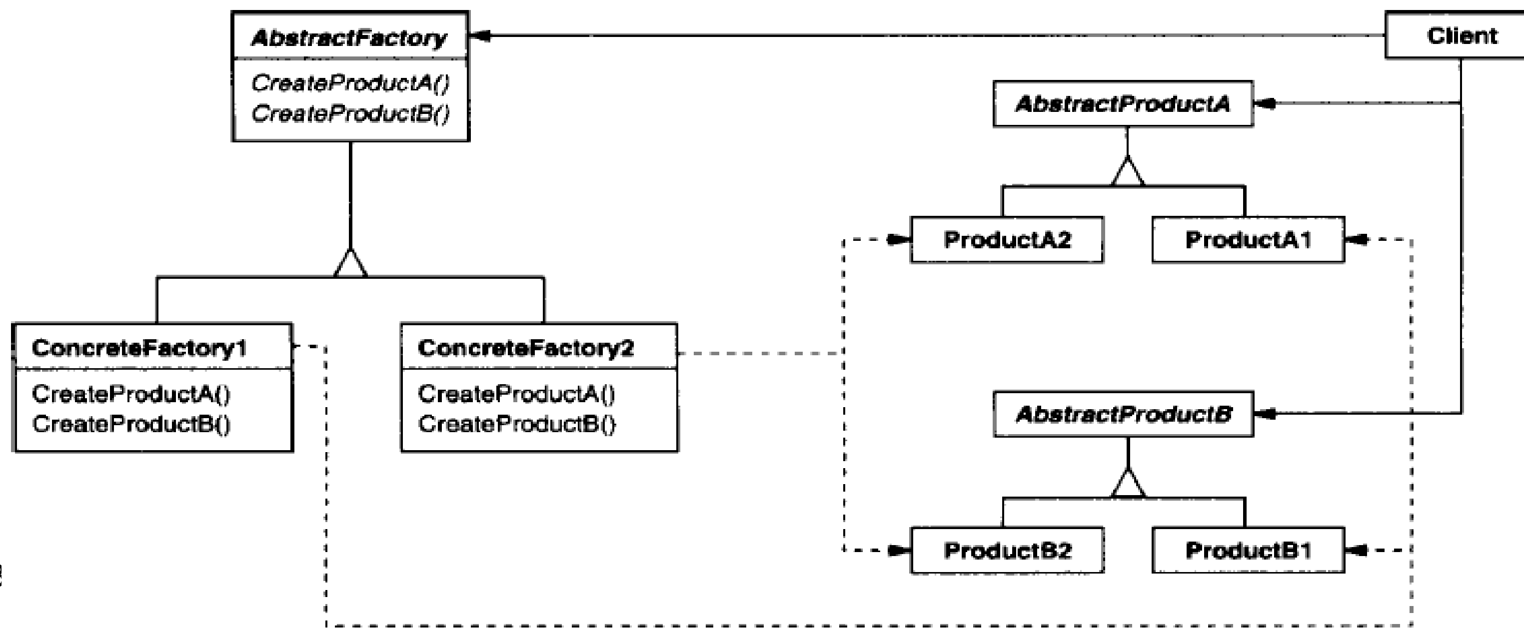
Абстрактна фабрика (Abstract Factory) [1]

- **Цел**: предоставя интерфейс за създаване на семействата на свързани или зависими обекти, без да уточнява конкретните им класове .
- **Познат още като**: комплект (Kit)
- **Мотивация**: помислете за набор от инструменти за потребителски интерфейс, който поддържа множество стандарти, като напр. Motif и Presentation Manager. Различните възприятия (look-and-feels) определят различни изяви и поведение за примитивите (widgets) на потребителския интерфейс, като плъзгачи, бутони и прозорци. *С цел преносимост до различни стандарти и платформи, приложението не трябва да фиксира в кода графични примитиви от определен вид.*



Структура и участници

- **AbstractFactory** (WidgetFactory) - декларира интерфейс за операции, които създават обекти на абстрактни продукти.
- **ConcreteFactory** (MotifWidgetFactory, PMWidgetFactory) - имплементира операции за създаване на обекти на конкретни продукти.
- **AbstractProduct** (Window, ScrollBar) - декларира интерфейс за вида на продуктов обект.
- **ConcreteProduct** (MotifWindow, MotifScrollBar) – (1) определя как продуктовият обект се създава от съответната конкретна фабрика; (2) имплементира AbstractProduct интерфейса.
- **Client** – използва само интерфейсите, декларирани от класовете AbstractFactory и AbstractProduct.



Приложимост 1/2

Използваме шаблона Abstract Factory, когато:

- Дадена система следва да бъде независима от начина, по който нейните продукти са създадени, композирани и представени.
- Системата трябва да бъде конфигурирана с едно от множество семейства от продукти.
- Семейство от свързани помежду си продукти е предназначено те да бъдат използвани заедно, и имаме нужда да фиксираме това ограничение.
- Искаме да предоставим библиотека от класове на продукти, като обаче се разкриват само техните интерфейси, но не и имплементациите им.

Приложимост 2/2

Коопериране:

- Обикновено по време на изпълнение се създава една единствена инстанция на класа ConcreteFactory. Тази конкретна фабрика създава продуктови обекти, които имат определена имплементация. За да създадат различни обекти на продукта, клиентите трябва да използват различни конкретни фабрики.
- AbstractFactory отлага създаването на продуктови обекти до нейния ConcreteFactory подклас.

Последици от прилагането на абстрактната фабрика 1/2

- *Тя изолира конкретни класове, като капсулира отговорността и процеса на създаване на продуктови обекти.* Клиентите манипулират екземплярите им чрез абстрактни интерфейси. Имена на класовете-продукти са изолирани в имплементацията на конкретната фабрика, и затова те не се появяват в клиентския код.
- *Това улеснява обмена на продуктови семейства.* Класът на конкретната фабрика се появява само веднъж в приложението - там, където се инстанциира. Той може да използва различни конфигурации на продукти, просто чрез смяна на конкретната фабрика – така цялото продуктово семейството се променя наведнъж.

Последици от прилагането на абстрактната фабрика 2/2

- *Тя насърчава съгласуваността между продукти.* Когато обекти от едно семейство са проектирани да работят заедно, много е важно приложението да използва обекти само от едно семейство в даден момент.
- *Поддръжката на нови видове продукти е трудна - защото AbstractFactory интерфейсът определя набор от продукти, които могат да бъдат създадени.* Поддръжката на нови видове продукти изисква разширяване на интерфейса на фабриката, което включва промяна на AbstractFactory класа и на всички негови подкласове.

Имплементационни техники 1/2

- *Фабриките като singletons.* Приложението обикновено се нуждае от един екземпляр на ConcreteFactory за продуктова фамилия. Затова най-удачната ѝ имплементация е като Сек (Singleton) - GoF стр. 127.

- *Създаване на продукти.* AbstractFactory само декларира интерфейс за създаване на продукти. Подкласовете на ConcreteProduct трябва реално да се създават от конкр. фабрика. Най-разпространеният начин да направим това е чрез метод фабрика (виж Factory Method) за всеки един продукт. Конкретната фабрика ще определи своите продукти чрез пренаписване на factory method за всеки продукт.

Имплементационни техники 2/2

- *Дефиниране на разширяеми фабрики.*
  - AbstractFactory обикновено дефинира различна операция за всеки вид продукт, който може да произвежда. Добавяне на нов вид продукт изисква промяна на AbstractFactory интерфейса и на всички класове, които зависят от него.
  - По-гъвкав, но по-малко безопасен дизайн е **да се добави параметър за операциите, които създават обекти**. Този параметър определя вида на обекта за създаване. Така, AbstractFactory има нужда **само от една "*Make*" операция с параметър**, показващ вида на обекта за създаване. Това е техника, използвана в Prototype и базираните на класове абстрактни фабрики.

Пример - Garden [3] 1/2

```
public abstract class Garden {
    public abstract Plant getCenter(); //What are good border plants?
    public abstract Plant getBorder(); //What are good center plants?
    public abstract Plant getShade(); //What plants do well in partial shade?
}
```

```
public class Plant {
    String name;
    public Plant(String pname) {
      name = pname; //save name
    }
    public String getName() {
      return name;
    }
}
```

```
public class VegieGarden extends Garden {
  public Plant getShade() {
    return new Plant("Broccoli");
  }
  public Plant getCenter() {
    return new Plant("Corn");
  }
  public Plant getBorder() {
    return new Plant("Peas");
  }
}
```

Пример – Garden [3] 2/2

Можем лесно да конструираме нашата абстрактна фабрика, за да се върнем един от тези Garden обекти, базирани на текстовия аргумент:

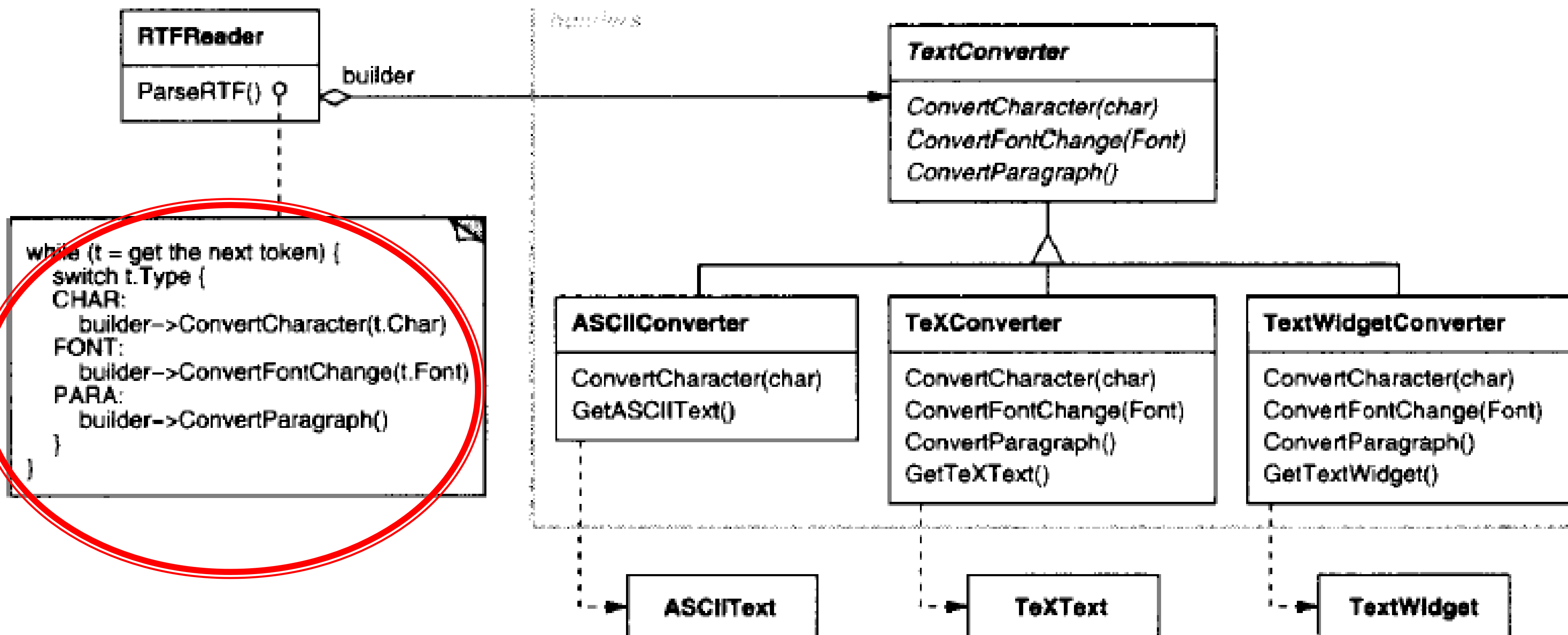
```
class GardenMaker {
    //Abstract Factory which returns one of three gardens
    //it creates the whole family of products at once… Comments….?
    private Garden gd;
    public Garden getGarden(String gtype) {
        if(gtype.equals("Perennial")) //for the whole year
            gd = new PerennialGarden();
        else if(gtype.equals("Annual"))
                gd = new AnnualGarden();
            else
                gd = new VegieGarden(); //default
        return gd;
    }
}
```

Шаблон Строител (Builder) [1]

- **Цел**: да се отдели изграждането на сложен обект от неговото представяне, така че един и същ процес на конструкция да може да създават различни представяния.
- **Мотивация**: четец (reader) на документи в RTF (Rich Text Format) трябва да може да конвертира RTF до много текстови формати. Трябва да може да се добавя лесно ново конвертиране, без да се модифицира четеца. Друг пример: XML parser + XLS converter
- Решението е да конфигурираме класа *RTFReader с обект TextConverter, който конвертира RTF до друго текстово представяне*. Понеже RTFReader парсва RTF документ, то той ще използва *TextConverter, за да извърши конверсията*.

Продължение

- Всеки път, когато RTFReader разпознае RTF маркер (или обикновен текст или RTF контролна дума), той издава заявка към TextConverter за конвертиране на маркера. TextConverter обектите са отговорни както за извършване на преобразуването на данни, така и за представяне на знака в определен формат.
- Подкласовете на TextConverter се специализират в различни преобразувания и формати.

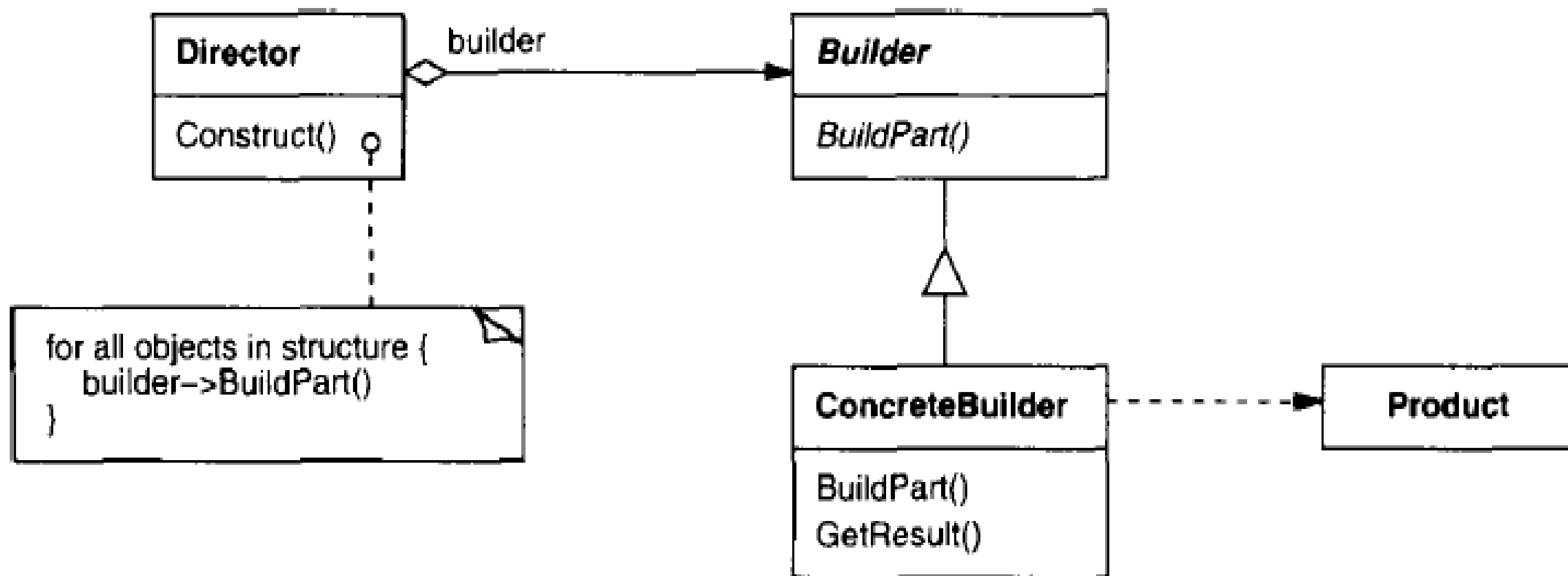


Използване

- Шаблонът Строител обхваща отношенията:
  - Всеки конвертиращ клас се нарича **builder** и
  - Четецът (reader) се нарича **director.**
- В примера, шаблонът Строител **разделя алгоритъма** за интерпретиране на текстовия формат (т.е. парсера на *RTF документи*) **от създаването и представянето на конвертирания формат**.
- Това ни дава възможност за многократно използване (**reuse**) на RTFReader алгоритъма за създаване на различни текстови презентации от RTF документи – просто с конфигуриране на RTFReader с различни подкласове на TextConverter.

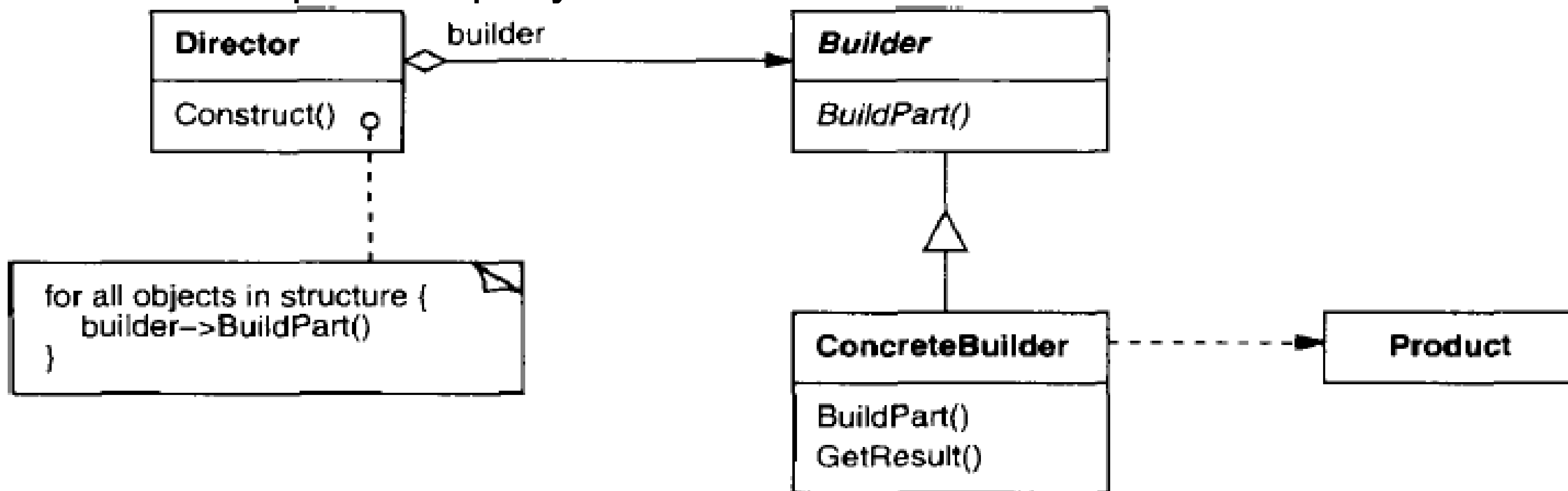
Структура и участници 1/2

- **Builder** (TextConverter) - задава абстрактен интерфейс за създаване на части на даден обект Product.
- **ConcreteBuilder** (ASCIIConverter, etc.) – (1) конструира и сглобява части на продукта чрез прилагането на интерфейса Builder; (2) определя и следи представянето, което той създава; (3) предоставя интерфейс за изтегляне на продукта (напр. GetASCIIText, GetTextWidget).



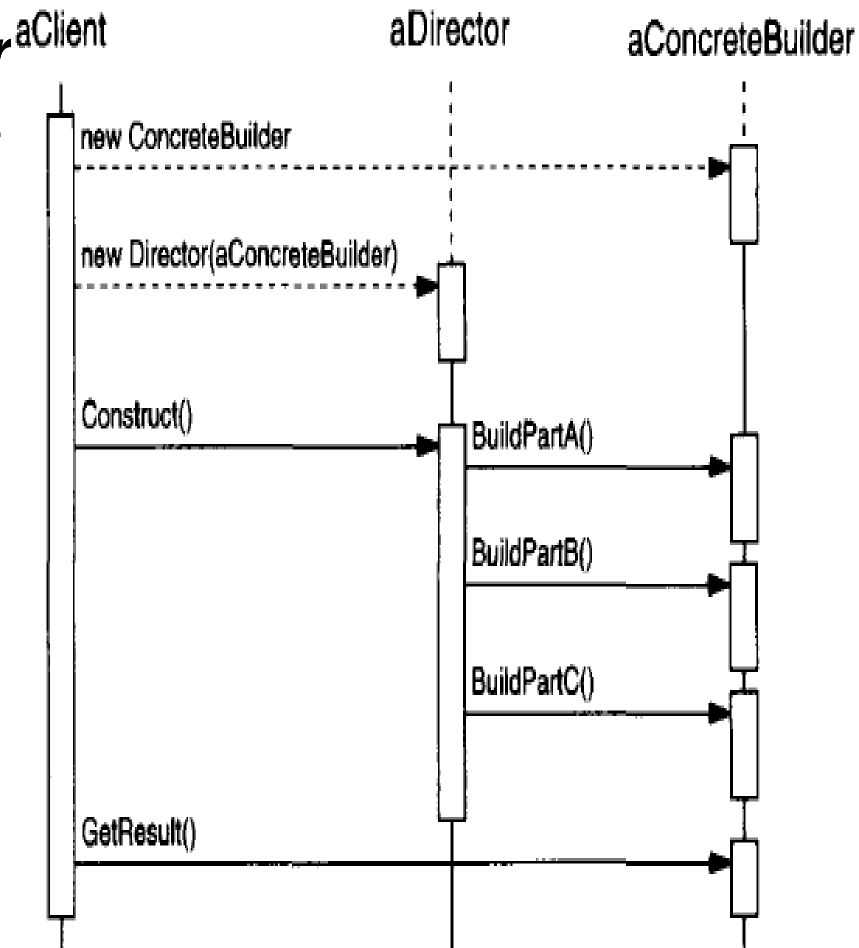
Структура и участници 2/2

- **Director** (RTFReader) – конструира обект, използвайки интерфейса Builder.
- **Product** (ASCIIText, TeXText, TextWidget) – (1) представлява сложен обект в процес на изграждане. ConcreteBuilder изгражда вътрешното представяне на продукта и определя процеса, чрез който той се сглобява; (2) включва класове, които определят съставните части, включително интерфейсите, за сглобяване на частите в крайния резултат.



Сътрудничество

- Клиентът създава конкр. ***Builder***
- Клиентът създава ***Director*** обект и го ***конфигурира със желания Builder*** обект.
- ***Director*** известява ***Builder*** всеки път, когато част от продукта трябва да се изгради.
- ***Builder*** обработва заявки от ***Director*** и добавя части към продукта.
- Клиентът извлича продукта от строителя ***Builder***.



Последици 1/2

- *Шаблонът ви позволява да варирате вътрешното представяне на продукта.* Обектът Builder предоставя на режисьора абстрактен интерфейс за изграждане на продукта. Интерфейсът позволява на строителя да скрие представянето и вътрешната структура на продукта. Той също така крие начина, по който продуктът се сглобява. Тъй като продуктът е конструиран чрез абстрактен интерфейс, всичко, което трябва да направите, за да промените вътрешното представяне на продукта, е да дефинирате нов вид конкретен строител.
- *Той изолира кода за конструиране и представяне.* Шаблонът Builder подобрява модулността чрез капсулиране на начина, по който се изгражда и представя сложен обект. Клиентите не трябва да знаят нищо за класовете, които определят вътрешната структура на продукта; такива класове не се появяват в интерфейса Builder. Всеки ConcreteBuilder съдържа кода за създаване и асемблиране на определен вид продукт.

Последици 2/2

- *Резултат от изолацията на кода за изграждане и представяне:* *кодът се пише веднъж, и после различни Directors могат да го използват многократно за изграждане на варианти на Product от същия набор от части*. В предния пример с RTF документи, можем да дефинираме четец за различен от RTF формат, напр. SGMLReader, и да ползваме същия TextConverter за генериране на ASCIIText, и т.н.
- *Той ни дава по-прецизен контрол над строителния процес.* За разлика от други създаващи шаблони, с които продуктът се създава "с един изстрел", Builder шаблонът изгражда продукта стъпка по стъпка, под контрола на режисьора. *Само когато продуктът е завършен, режисьорът може да го изтегли от строителя*. Така интерфейсът Builder отразява процеса на изграждане на продукта повече, отколкото други създаващи шаблони. Това ни дава по-прецизен контрол върху процеса на строителството, а оттам и над вътрешната структура на получения продукт.

Въпроси на имплементацията

- *Assembly & construction interface.* Строителите конструират своите продукти в стъпков режим, затова интерфейсът на класа Builder трябва да бъде достатъчно общ, за да позволи изграждането на продукти за всички видове конкретни строители.
- Ключов въпрос за дизайна се отнася до *модела на процеса на изграждане и сглобяване*. Обикновено е достатъчен модел, при който резултатите на заявките за конструиране на продукта просто се добавят към продукта (*append*).
- *Защо няма абстрактен клас за продукти?* В общия случай, продуктите, произведени от конкретните строители, се различават толкова много по тяхното представяне, че биха спечелили много малко от различни продукти с общ супер-клас.
- *Празните методи са тези по подразбиране в Builder.*

Builder пример [3]

```
abstract class multiChoice { //abstract Builder
    //This is the abstract base class
    //that the listbox and checkbox choice panels
    //are derived from.

    Vector choices; //array of labels
    //----------------------------------------
    public multiChoice(Vector choiceList) {
        choices = choiceList; //save list
    }
    //to be implemented in derived classes
    abstract public Panel getUI(); //return a Panel of components
    abstract public String[] getSelected(); //get list of items
    abstract public void clearAll(); //clear selections
}
```

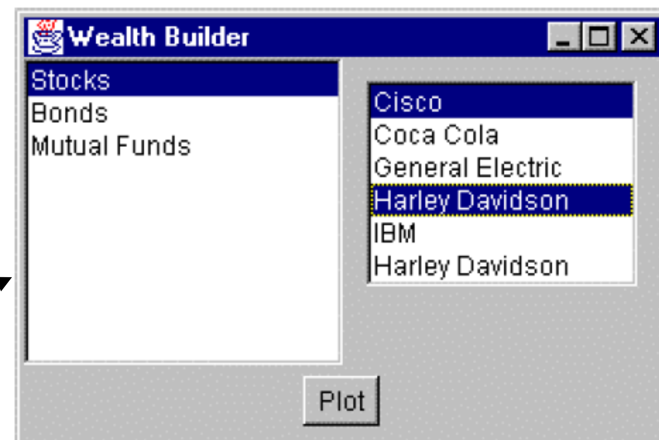
Продължение

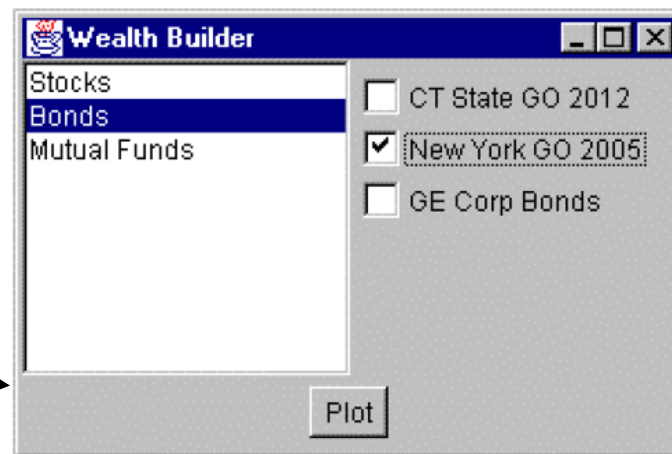
Методът getUI връща Panel контейнер е дисплей multiple-choice. Двата дисплея, които използваме тук - checkbox panel или list box panel - са получени от този абстрактен клас :

- **class listboxChoice extends *multiChoice***

или

- **class checkBoxChoice extends *multiChoice***





Прост Factory клас решава кой от двата класа да върне

```
class choiceFactory {
    multiChoice ui;
    //This class returns a Panel containing
    //a set of choices displayed by one of
    //several UI methods.

    public multiChoice getChoiceUI(Vector choices) {
        if(choices.size() <=3) {
            //return a panel of checkboxes
            ui = new checkBoxChoice(choices);
        } else {
            //return a multi-select list box panel
            ui = new listboxChoice(choices);
        }
        return ui;
    }
}
```

На езика на шаблоните за проектиране, този фабричен клас се нарича **Director**, а класовете, разширяващи *multiChoice,* са конкретни **Builders**.

Прототип (Prototype) [1]

- **Цел:** определя вида на обектите за създаване чрез използване на прототипен екземпляр, като създава нови обекти посредством копиране на този прототип.

- **Мотивация:** създаваме софтуерен редактор за музикални партитури чрез персонализиране на обща рамка за графични редактори и добавяне на нови обекти, които представляват ноти, паузи и групи от ноти. Рамката предоставя абстрактен клас Graphic за графични компоненти (напр. ноти) и абстрактен клас Tool за определяне на инструменти като тези в палитрата. Рамката предефинира GraphicTool подклас за инструменти, които създават екземпляри на графични (Graphic) обекти и да ги добавят към документа.

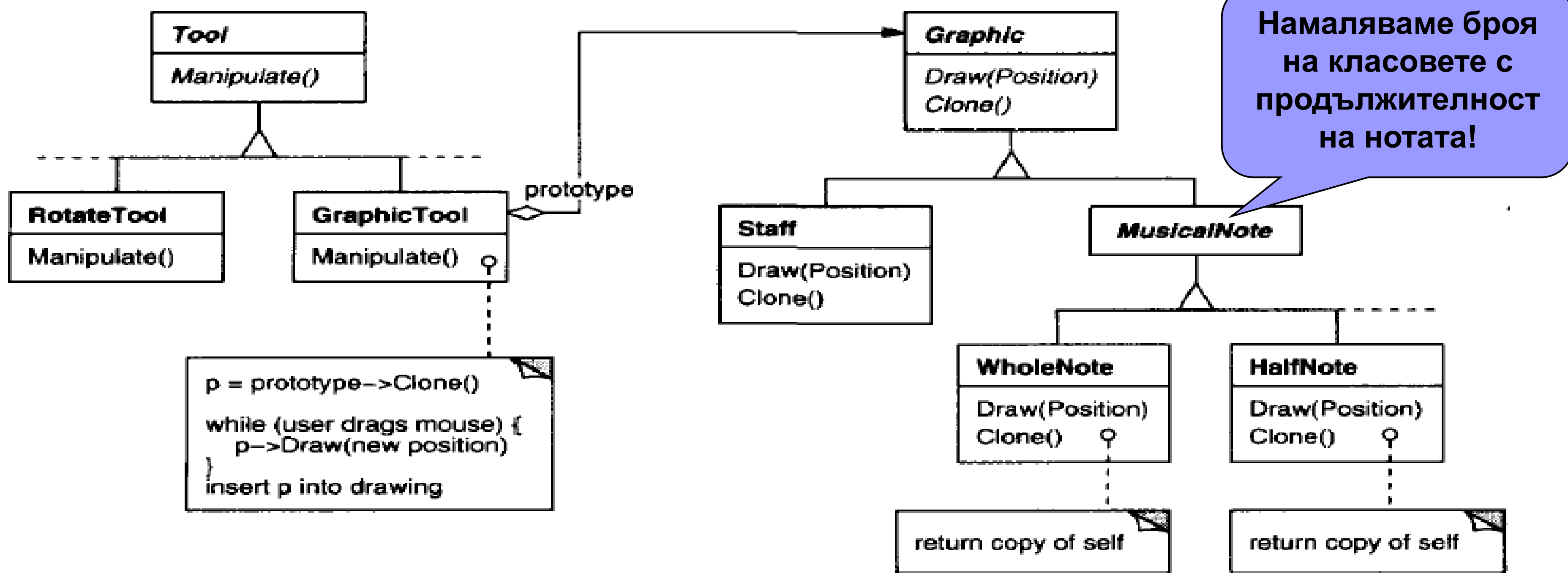
Проблемът

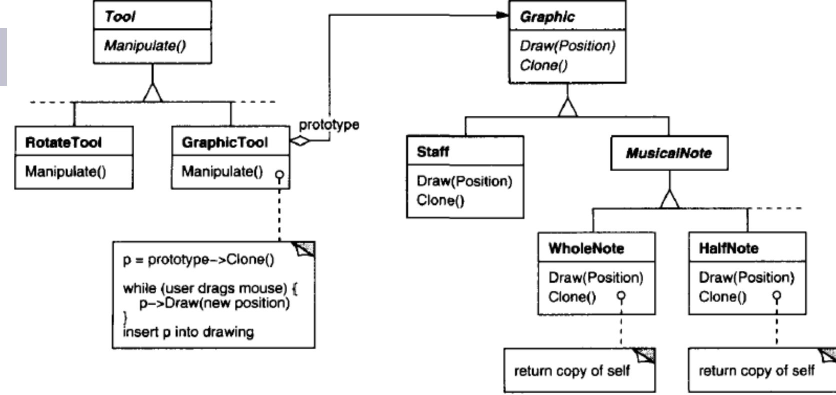
Но GraphicTool представлява проблем за дизайнера на рамката: GraphicTool не знае как да се създаде екземпляри на нашите класове за описание на музика, за да ги добави към резултата.

- Бихме могли да създадем подклас GraphicTool за всеки вид музикален обект, но това ще произведе много подкласове, които се различават само по вида на музикалния обект, който създават
- Вече знаем, че обектовата композиция (делегация) е гъвкава алтернатива на наследяването
- Въпросът е, как може рамката да я използва с цел задаване на параметризирани екземпляри на GraphicTool, за създаване на нови обекти?

- Решение – *<u>нека GraphicTool да създава нови Graphic обекти чрез копиране („клониране“) на екземпляр на подклас на Graphic</u>*. Наричаме този екземпляр прототип (**prototype**). GraphicTool е параметризиран с прототипа, който трябва да клонира и чийто клонинг да добави към документа. Ако всички подкласове на Graphic поддържат операцията Clone, то GraphicTool ще може да клонива всеки вид Graphic.

  Така в нашия редактор всеки инструмент за създаване на музикален обект е екземпляр на GraphicTool, инициализиран с различен прототип с цел клонирането му!





Участници и структура

- **Prototype** (Graphic) – декларира интерфейс за клониране.
- **ConcretePrototype** (Staff, WholeNote, HalfNote) – имплементира операцията за клониране на себе си.
- **Client** (GraphicTool) – създава нов обект чрез извикване на операция за клониране на прототипа.

Client
Operation()
prototype
Prototype
Clone()
p = prototype->Clone()
ConcretePrototype1
Clone()
ConcretePrototype2
Clone()
return copy of self
return copy of self

Приложимост

- когато класовете на създаваните обекти се определят по време на изпълнение, например чрез динамично зареждане;
- за да се избегне изграждането на йерархия от класовете на фабрики, които създават йерархия от продукти, *или*
- когато инстанции на клас може да имат една от няколко различни комбинации от състояния. Може да бъде по-удобно да се инсталира съответния номер на прототипи и ги клонираме, отколкото да създаваме екземпляри на класа ръчно, всеки път със съответното състояние.

Последици 1/3

- Както при Abstract Factory и Builder:
  - скрива конкретни продуктови класове от клиента, като по този начин намалява броя на имената, които клиентът познава;
  - позволява клиентът да работи със специфични класове за приложението без нужда от промяна в клиента.

- *Добавяне и премахване на продукти по време на изпълнение* – позволява вкарването на нови продукти само чрез регистриране на екземпляр на прототипа при клиента динамично!

Последици 2/3

- *Задаване на нови обекти чрез вариране на стойности -* динамичните системи определят ново поведение чрез обектна композиция – т.е. чрез стойностите за променливи на обекта и не чрез създаване на нови класове. Значително намалява броят на необходимите класове;
- *Задаване на нови обекти чрез вариране на структура* – за удобство, такива приложения често ви позволяват да инстанциирате комплексни структури, дефинирани от потребителя. Доколкото съставният обект имплементира Clone като дълбоко копие, то прототипи могат да бъдат различни структури.

Последици 3/3

- *Динамично конфигуриране на приложение с класове* - такова приложение не може да извиква конструктори, указани по време на компилация. Вместо това, по време на изпълнение околната среда създава инстанция на всеки клас автоматично, когато е необходимо, и я регистрира например в мениджър на прототипите.

Въпроси на имплементацията 1/3

Използване на мениджър на прототипите - когато броят на прототипите в системата не е фиксиран (т.е. те могат да бъдат създадени и унищожени динамично), водим регистър на наличните прототипи. Клиентът ще поиска от регистъра прототип, преди да го клонира. Наричаме този регистър **prototype manager:**

- Той може да връща прототип по ключ;
- За целта регистрира прототипа под този ключ;
- Клиентът управлява и разглежда регистъра по време на изпълнение.

Въпроси на имплементацията 2/3

- *Имплементиране на операцията Clone* - как да стане коректно при циклични референции? *Плиткото копие* (shallow copy) е просто и често достатъчно,

  НО

  клониране на прототипи със сложни структури обикновено изисква дълбоко копие, защото клонингите, както и оригиналът, трябва да бъдат независими.

Въпроси на имплементацията 3/3

- *Инициализиране на клонингите* – често трябва да се инициализират някои или всички параметри от вътрешното състояние на клиента. По принцип не може да предаваме тези стойности в Clone операцията, тъй като броят им ще варира между класовете на прототипите. Някои прототипи може да изискват множество параметри за инициализация, а за други да няма нужда. *<u>Предавайки параметри в Clone операцията, ние изключваме единния интерфейс за клониране</u>*.

Клониране в Java

/* Cloning in Java - getting a copy of any Java object using the **clone** method:

•It is a **protected** method and can only be called from within the same class or the module that contains that class.

•You can only clone objects which are declared to implement the **Cloneable** interface.

•Objects that cannot be cloned throw the **CloneNotSupported** Exception.*/

```
public class SwimData implements Cloneable {
	public Object clone() {
		try{
				return super.clone();
		} catch(Exception e) {
				System.out.println(e.getMessage());
				return null;
		}
	}
}
```

Пример [2]

- Проста програма чете данни за плувци (Swimmers) от файл и после клонира резултатния обект
- Класът **Swimmer** съдържа информацията, а друг клас **SwimData** поддържа вектор от Swimmers четени от файла

```
class Swimmer {
      String name;
      int age;
      String club;
      float time;
      boolean female;
      …..
```

```
public class SwimData implements Cloneable {
    Vector swimmers;
    public SwimData(String filename) {
     String s = "";
     swimmers = new Vector();
     //open data file
     InputFile f = new InputFile(filename);
     s= f.readLine(); //read in and parse each line
     while(s != null) {
          swimmers.addElement(new Swimmer(s));
          s= f.readLine();
     }
     f.close();
    }
```

Продължение

- Създаваме още и *getSwimmer* метод в SwimData и още *getName*, *getAge* и *getTime* методи в класа **Swimmer**.
- След като сме прочели данните в SwimInfo, можем да ги покажем в списък.

```
swList.removeAll(); //clear list
for (int i = 0; i < sdata.size(); i++) {
    sw = sdata.getSwimmer(i);
    swList.addItem(sw.getName()+" "+sw.getTime());
}
```

Продължение

- Тогава, когато потребителят задейства бутона Clone, ще клонираме класа и сортираме данните в новия клас различно.
- Ще клонираме данните, защото това е по-бързо, вместо да създаваме нов екземпляр на класа.

```
sxdata = (SwimData)sdata.clone();
sxdata.sortByTime(); //re-sort
cloneList.removeAll(); //clear list

//now display sorted values from clone
for(int i=0; i< sxdata.size(); i++) {
  sw = sxdata.getSwimmer(i);
  cloneList.addItem(sw.getName()+" "+sw.getTime());
}
```

Продължение

- В оригиналния клас, имената са сортирани по пол и после по време, докато в клонирания клас са сортирани само по време.
- Примерът използва плитко клониране (***shallow copy***) на оригиналния клас – клонират се референциите към обекти, но самите те сочат към същите обекти! => всяка операция над копираните данни ще се отрази и на оригиналните данни в прототипа.
- За други случаи е нужно дълбоко копиране - ***deep copy*** – може да използваме serializable interface.

Шаблон Сек (Singleton)

- **Цел** - да се гарантира даден клас да има само един екземпляр, и да се предостави глобална точка за достъп до него.
- **Мотивация** - за някои класове е важно да имат точно една инстанция (екземпляр) – например един контролер или мениджър. Как ще се гарантира, че един клас ще има само един екземпляр и че той ще е лесно достъпен? Глобалната променлива прави обекта достъпен, но не ни предпазва от инстанцииране на множество обекти.
- **Решение** – това да се направи в самия клас, който да отговаря за следенето на своя екземпляр и да осигури начин за достъп до него. Това е шаблонът Сек (Singleton).

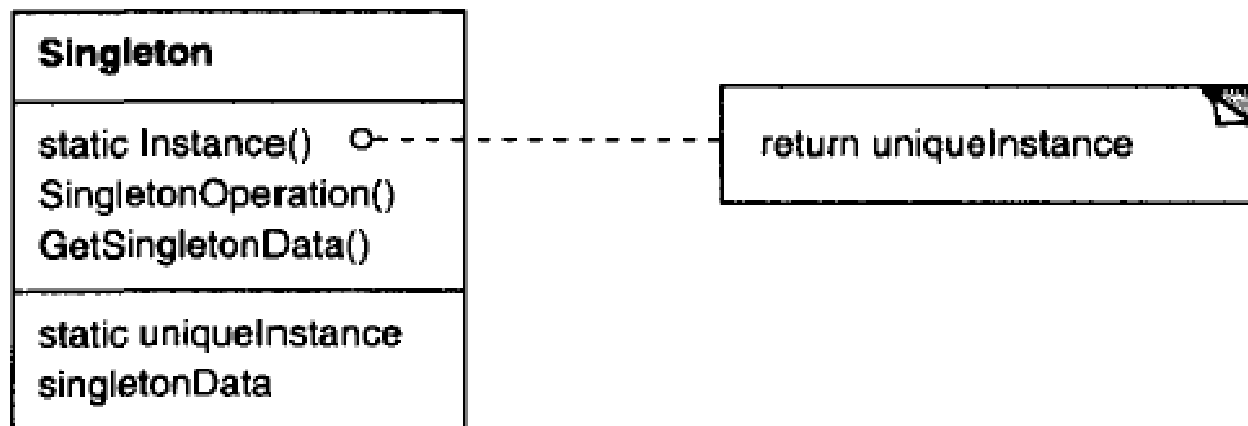
Приложимост

Използваме Singleton, ако:

- трябва да има точно един екземпляр на даден клас, и той трябва да бъде достъпен за клиента от добре позната точка за достъп;
- когато такъв клас с една единствена инстанция трябва да бъде наследяван, без да се променя първоначалният код.

Structure, Participants & Collaborations

- **Участници**
  - Дефинира за контейнера операция за достъп до екземпляра – Instance() – която е static.
  - Може да създава сам своя уникален екземпляр
- **Взаимодействия**
  - Клиентът има достъп до екземпляра само през статичната операция Instance().



Последствия 1/2

- *Контролиран достъп до единствена инстанция.* Класът Сингълтън капсулира строг контрол над това как и кога клиентите да имат достъп до него.
- *Намалено пространство на имената.* Сингълтън е подобрение в сравнение с глобалните променливи, които съхраняват единични екземпляри*.*
- *Позволява усъвършенстване на операциите и представянето.* Клас Сингълтън може да бъде наследяван и използван многократно по време на изпълнение.

Последствия 2/2

- *Позволява променлив брой екземпляри* – шаблонът разрешава повече от един екземпляр на Singleton клас. Освен това, можете да използвате един и същ подход, за да се контролира броя на копията, която използва приложението.
- *По-гъвкав от клас операции -* друг начин за пакетиране на функционалността на Сингълтън е да се използват клас операции (тоест, статични член-функции в C++ или клас методи в Java) – по-трудно за повече от една инстанция на даден клас.

Пример

```
class iSpooler {
    //this is a prototype for a printer-spooler class
    //such that only one instance can ever exist
    static boolean instance_flag = false; //true if 1 instance
    //the constructor is private!
    private iSpooler() { }
    //static Instance method returns one instance or null
    static public iSpooler Instance() {
        if (! instance_flag) {
                instance_flag = true;
                return new iSpooler(); //only callable from within
        }
        else
                return null; //return no further instances – no exceptions!
    }
    //-----------------------------------------
    public void finalize() {
        instance_flag = false;
    }
}
```

Задача

Модифицирайте предходният пример така, че:

- класът да работи със статична референция към екземпляра си (с първоначална стойност Null) вместо с instance_flag и
- да връща екземпляра при всяко обръщение към статичния метод.