Návrhový vzor *Factory* v JAVA API

Martin Kot
Katedra informatiky
VŠB - Technická univerzita Ostrava
martin.kot.fei@vsb.cz

Abstrakt

V třídách API jazyka JAVA je použito mnoho návrhových vzorů. Najdeme zde i tvořící vzory *Factory* a *Factory Method*. V tomto textu je uvedeno několik konkrétních případů. Aby o nich čtenář získal lepší představu, je ke každému příkladu vyobrazen diagram tříd a stručný popis vysvětlující, k čemu továrna v dané implementaci slouží. Největší prostor je věnován třídě `java.awt.Toolkit` a objektům, které generuje.

1 Úvod

1.1 Návrhové vzory

Objektová orientace se v programování prosazuje již řadu let. Za tu dobu se zjistilo, že se často stejné problémy opakují pouze v malých obměnách. Na řadu z nich se našla vhodná řešení. Tato řešení nazýváme návrhové vzory.

Každý návrhový vzor tvoří množina objektů a tříd, které jsou propojeny vazbami. Když se při návrhu konkrétního systému objeví problém, který by mohl nějaký návrhový vzor vyřešit, nahradíme třídy návrhového vzoru třídami z našeho systému. Přitom potřebujeme, aby navzájem zaměněné třídy měly v řešené situaci stejné chování. Protože užitím návrhového vzoru řešíme jen jeden problém, mohou se v těchto třídách vyskytovat i další metody, řešící jiné problémy. Všechny metody vyžadované návrhovým vzorem musí být k dispozici, aby výsledný objektový návrh mohl plnit svou funkci. Obdobně musí zůstat zachovány vazby požadované návrhovým vzorem, ale můžeme přidat další pro jiné potřeby.

Návrhové vzory dělíme podle účelu do tří skupin:

Tvořící vzory (Creational patterns) - Přenechávají tvorbu některých objektů podtřídám nebo jiným objektů. Příklady jsou (ponechávám užívané anglické názvy, i když se občas objevují pokusy o překlad):

- *Factory* (někdy také *Abstract Factory*) - Poskytuje rozhraní pro tvoření rodin příbuzných objektů bez specifikace konkrétních tříd.

- *Factory Method* - Definuje rozhraní pro tvorbu objektů, ale nechá podtřídu rozhodnout, kterou třídu konkrétně instanciovat.
- *Prototype* - Specifikuje druhy objektů, které tvoří kopírováním prototypu.

Strukturální vzory (Structural patterns) - Popisují obecné struktury objektů a tříd. Příklady jsou:

- *Composite* - Skládá objekty do stromové struktury. Klient může zacházet s jednotlivými prvky stejně jako se složeninou z více prvků.
- *Adapter* - Převádí rozhraní třídy na takové, jaké potřebuje klient.
- *Decorator* - Připojuje k objektu dynamicky další funkčnost.
- *Proxy* - Poskytuje pro nějaký objekt zástupce, který řídí přístup k tomuto objektu.

Vzory chování (Behavioral patterns) - Popisují algoritmy nebo spolupráci objektů. Příklady jsou:

- *Chain of Responsibility* - Snaží se zabránit, aby žádost uživatele mohlo zpracovat více objektů. Objekty se zřetězí a zpráva postupně putuje po řetězci, dokud ji některý objekt nezpracuje.
- *Observer* - Definuje závislost *n* objektů na jednom jiném. Pokud tento jeden objekt změní stav, jsou o změně informovány všechny závislé objekty.
- *Iterator* - Umožňuje sekvenční přístup k prvkům agregovaného objektu, aniž by klient musel znát vnitřní strukturu agregace.
- *State* - Umožňuje objektu měnit chování na základě vnitřního stavu.
- *Strategy* - Umožňuje změnu algoritmu nezávisle na klientovi, který algoritmus užívá.

1.2 API jazyka JAVA

Aplikační programátorské rozhraní (API - Application Program Interface) jazyka JAVA je soubor tříd, které jsou k dispozici programátorům v tomto jazyce. V třídách API nalezneme implementaci několika návrhových vzorů. Například grafické uživatelské rozhraní (GUI - Graphic User Interface) umožňuje vnořování komponent do sebe podle návrhového vzoru *Composite*. V takto vnořených objektech GUI se události zpracovávají podle vzoru *Chain of Responsibility*. Při implementaci návrhového vzoru *Observer* stačí sledovaný objekt zdědit z třídy `Observer` a pozorovatelé musí implementovat rozhraní `Observable`.

Samozřejmě, že použití návrhových vzorů v JAVě není omezeno jen na ty, které jsou již přímo v API. Každý uživatel si může vlastní třídy navrhnout s využitím některého vzoru. Tato práce má ale za cíl zmapovat použití návrhového vzoru *Factory* v API jazyka JAVA a proto se možnostmi užití návrhových vzorů aplikačními programátory ve vlastních návrzích zabývat nebude. V názvech API tříd se slovo „factory“ vyskytuje poměrně často, ale jde spíše o implementace návrhového vzoru *Factory Method*. Proto bude sekce textu označená 3 věnována tomuto vzoru. V sekci 2 bude popsán vzor *Abstract Factory* a jeho dvě implementace v API.

2 Abstract Factory

2.1 Abstraktní popis

Pro představu, v čem je užití návrhového vzoru vhodné, si představme následující příklad. Mějme aplikaci, která používá komunikaci přes síť. Můžeme mít požadavek na zabezpečenou komunikaci i nezabezpečenou. Aplikace se současně může chovat buď jako klient nebo jako server. Nezabezpečenou komunikaci provádí třídy `UnsecureSocket` na straně klienta a `UnsecureServerSocket` na straně serveru. Komunikaci zabezpečenou protokolem SSL obdobně zajišťují třídy `SSLSocket` a `SSLServerSocket`. Pokud bychom aplikaci psali bez použití *Factory*, museli bychom při přepsání aplikace z nezabezpečené komunikace na zabezpečenou změnit v kódu konstruktory pro tvorbě socketů na jiné třídy.

Použitím *Abstract Factory* si situaci můžeme zjednodušit. `SSLSocket` a `UnsecureSocket` zdědíme ze společné třídy `Socket` nebo je necháme implementovat interface `Socket`, abychom zajistili jejich jednotné rozhraní. Pro `SSLServerSocket` a `UnsecureServerSocket` obdobně vytvoříme nadtřídu nebo interface `ServerSocket`. Přidáme třídu `SocketFactory`, která bude mít dvě metody:

```
createSocket():Socket
createServerSocket():ServerSocket
```

Z ní zdědíme třídy `SSLSocketFactory` tvořící v implementaci obou metod instance tříd pro zabezpečené sockety a `UnsecureSocketFactory` tvořící instance tříd pro nezabezpečené sockety. Místo přímého volání konstruktorů budeme sockety tvořit prostřednictvím `SocketFactory`. Převod z nezabezpečené komunikace na zabezpečenou se provede jednoduše v kódu na jednom místě nahrazením konstruktoru třídy `UnsecureSocketFactory` za `SSLSocketFactory`.

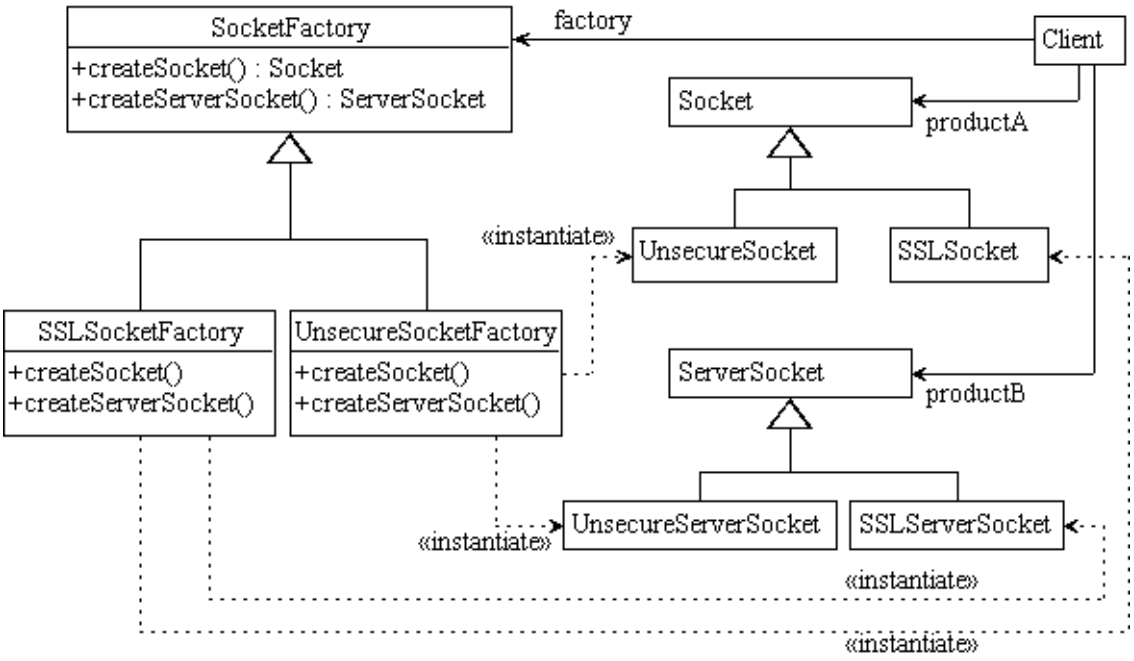
Výhoda se může projevit i v budoucnu. Až se objeví nový protokol pro zabezpečenou komunikaci, vytvoříme se novou podtřídu třídy `SocketFactory` a dvě třídy implementující sockety zabezpečené novým protokolem zděděné z našich obecných tříd pro sockety. Nebudeme muset procházet celý rozsáhlý kód a měnit staré třídy za nové. Pouze zase v jednom místě nahradíme starou podtřídu `SocketFactory` za novou. Celá situace je znázorněna na obrázku 1.

Podobných situací vhodných pro použití vzoru *Abstract Factory* je mnoho. Uvidíme je i v API jazyka JAVA. Abstraktní zobrazení návrhového vzoru je na obrázku 2.

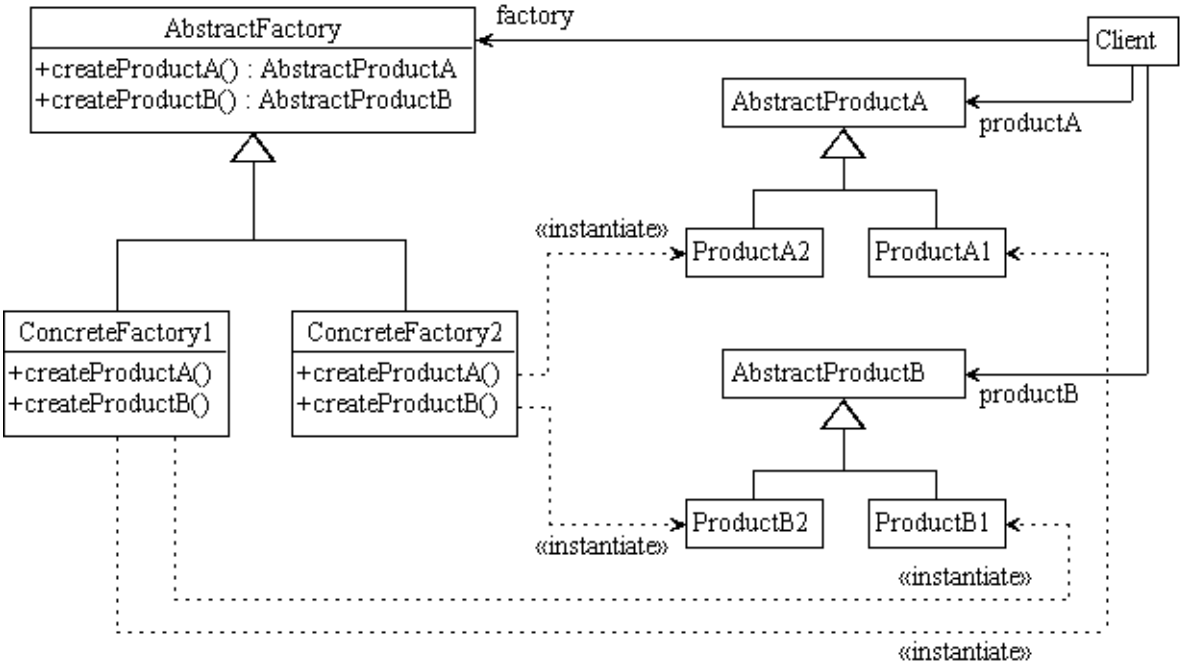
2.2 Toolkit

Nejdůležitějším použitím návrhového vzoru *Factory* v JAVě je třída `java.awt.Toolkit`. Programátor s jejím využitím může psát programy vybavené grafickým uživatelským rozhraním (GUI) přenositelné mezi platformami. Na každé platformě potom GUI může mít odlišný vzhled a případně i chování, ale z hlediska spolupráce s dalšími objekty má GUI pořád stejné rozhraní.

`java.awt.Toolkit` má celou řadu metod tvořících různé druhy objektů. Na obrázku 3 je zobrazen diagram tříd znázorňující tvorbu dvou produktů - `java.awt.peer.ButtonPeer`

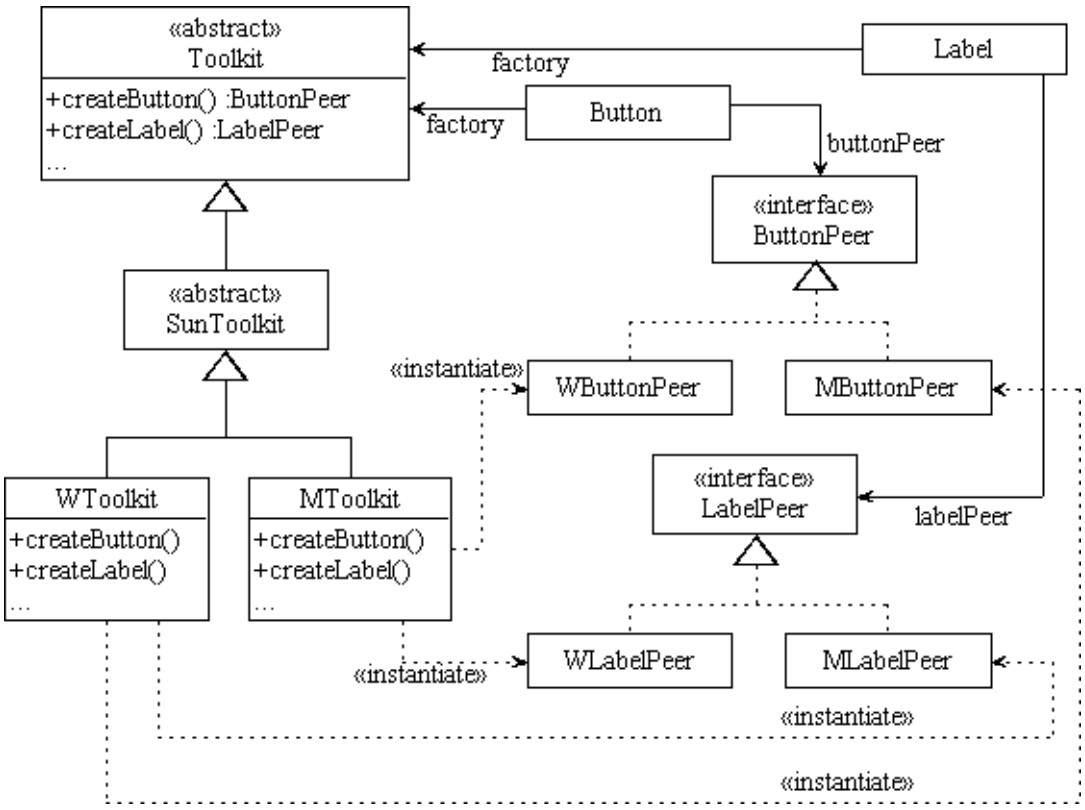


Obrázek 1: Diagram tříd motivačního příkladu užití *Abstract Factory*



Obrázek 2: Diagram tříd abstrakce návrhového vzoru *Abstract Factory*

a `java.awt.peer.LabelPeer`. Konkrétní továrna `sun.awt.windows.WToolkit` je implementací firmy Sun pro platformu Windows a `sun.awt.motif.MToolkit` pro XWindows v unixových systémech.



Obrázek 3: Diagram tříd zobrazující propojení Toolkit se dvěma druhy produktů

Každou tvořící metodu z `java.awt.Toolkit` používá objekt jiné třídy, protože si tím tvoří vlastní vzhled. Na obrázku 3 je `java.awt.peer.ButtonPeer` vytvořen a užíván objektem třídy `java.awt.Button` a obdobně `java.awt.peer.LabelPeer` objektem třídy `java.awt.Label`. V tabulce 1 je seznam všech tvořících metod třídy `java.awt.Toolkit` a tříd, které využívají volání těchto metod k vytvoření svého vzhledu.

V diagramu tříd na obrázku 3 byly jen dva abstraktní produkty a jim příslušné konkrétní produkty. Obrázky 4, 5 a 6 zobrazují celou hierarchii abstraktních produktů a konkrétních produktů pro platformu Windows. Abstraktními produkty zde jsou všechny interface z balíku `java.awt.peer`, které jsou návratovou hodnotou některé metody v tabulce 1. Konkrétní produkty představují všechny objekty implementující tato rozhraní. Na zmíněných obrázcích to jsou ty, jejichž názvy začínající ’W’ a nacházejí se v balíku `sun.awt.windows`. Právě ony jsou továrnou `sun.awt.windows.WToolkit` ve skutečnosti tvořeny.

Třídy v balíku `sun.awt.motif` mají hierarchii téměř stejnou jen s malými odlišnostmi (např. `MFileDialogPeer` dědí z `MDialogPeer`). Také ony implementují rozhraní z balíku

Metoda	**Klient**
createButton():java.awt.peer.ButtonPeer	java.awt.Button
createCanvas():java.awt.peer.CanvasPeer	java.awt.Canvas
createCheckbox():java.awt.peer.CheckboxPeer	java.awt.Checkbox
createCheckboxMenuItem(): java.awt.peer.CheckboxMenuItemPeer	java.awt.CheckboxMenuItem
createChoice():java.awt.peer.ChoicePeer	java.awt.Choice
createComponent():java.awt.peer.LightweightPeer	java.awt.Component
createDialog():java.awt.peer.DialogPeer	java.awt.Dialog
createFileDialog():java.awt.peer.FileDialogPeer	java.awt.FileDialog
createFrame():java.awt.peer.FramePeer	java.awt.Frame
createLabel():java.awt.peer.LabelPeer	java.awt.Label
createList():java.awt.peer.ListPeer	java.awt.List
createMenu():java.awt.peer.MenuPeer	java.awt.Menu
createMenuBar():java.awt.peer.MenuBarPeer	java.awt.MenuBar
createPanel():java.awt.peer.PanelPeer	java.awt.Panel
createPopupMenu():java.awt.peer.PopupMenuPeer	java.awt.PopupMenu
createScrollbar():java.awt.peer.ScrollbarPeer	java.awt.Scrollbar
createScrollPane():java.awt.peer.ScrollPanePeer	java.awt.ScrollPane
createTextArea():java.awt.peer.TextAreaPeer	java.awt.TextArea
createTextField():java.awt.peer.TextFieldPeer	java.awt.TextField
createWindow():java.awt.peer.WindowPeer	java.awt.Window

Tabulka 1: Tabulka tvořících metod třídy Toolkit a tříd využívajících tyto metody

`java.awt.peer` a představují konkrétní produkty. Jiné firmy mohou vytvořit další konkrétní továrnu i produkty pro další platformy i alternativní pro stejné platformy.

Běžný programátor v jazyce JAVA však tvořící metody třídy `Toolkit` přímo nikdy volat nebude. Na rozdíl od běžných tříd JAVA API, zdrojové kódy balíku `sun.awt.windows` nejsou součástí distribuce JDK a tím ani API dokumentace. Jsou v distribuci obsaženy jen jako binární class soubory.

Objekty grafického rozhraní, uvedené v tabulce 1 jako klienti, si vytvářejí svůj objekt „peer“. Tento objekt se stará o vzhled a chování objektu grafického rozhraní po celou dobu jeho životního cyklu. Příkladem situace, kdy se „peer“ objekt vytvoří, je přidání nějaké komponenty (např. tlačítka) do kontejneru (např. okna). Diagram sekvencí zobrazující tuto situaci je na obrázku 7. Je zde zanedbáno rušení objektů, které není pro pochopení funkce továrny podstatné.

Nalezení konkrétní podtřídy třídy `java.awt.Toolkit` je zda vykonáno statickou metodou `getProperty()` třídy `java.lang.System`. V této třídě jsou systémové vlastnosti nastaveny nativními metodami. Metodě `getProperty()` se dá jako parametr název vlastnosti („java.awt.Toolkit“) a návratovou hodnotou je řetězec se jménem „toolkitu“ pro aktuální systém, na kterém aplikace běží. Statickou metodou `forName()` třídy `Class`, které se jako parametr předá získaný název „toolkitu“, se vytvoří nový objekt `Class` a z něj již je možné vytvořit objekt nalezené podtřídy `java.awt.Toolkit`. Na konci sekvence je tak vytvořen systémově závislý „peer“ objekt systémově nezávislými metodami. Prvky rozhraní v tělech svých metod provádějících systémově závislé operace (`show()` apod.) volají příslušnou metodu svého „peer“ objektu, který akci provede.

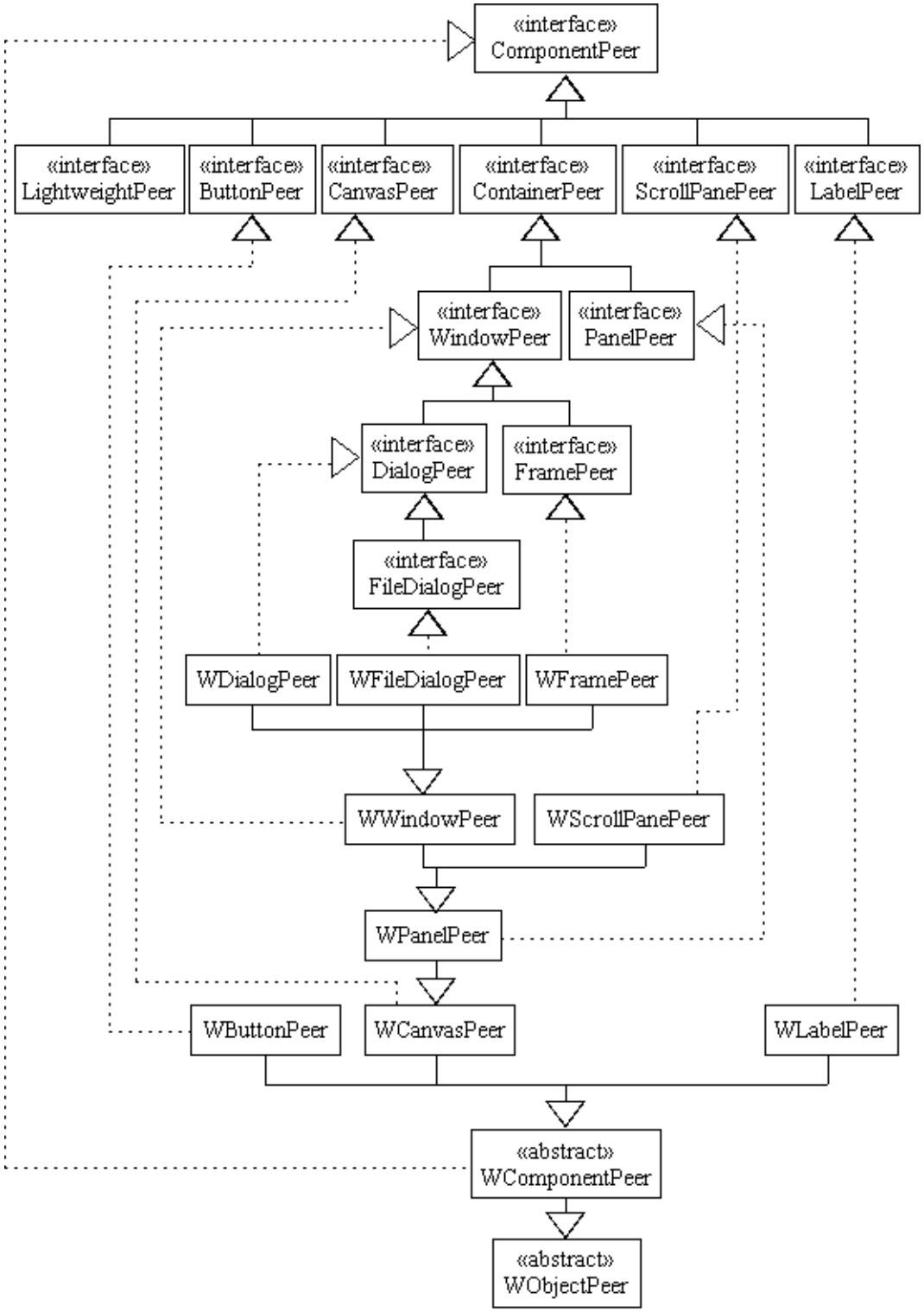
2.3 EditorKit

Dalším příkladem užití *Factory* v Javě je `javax.swing.text.EditorKit`. Roli klienta zde hraje `javax.swing.JEditorPane`. Objekty této třídy si prostřednictvím `EditorKit` vytvářejí objekty `Document` a `ViewFactory` z balíku `javax.swing.text`. Situace je znázorněna diagramem tříd na obrázku 8.

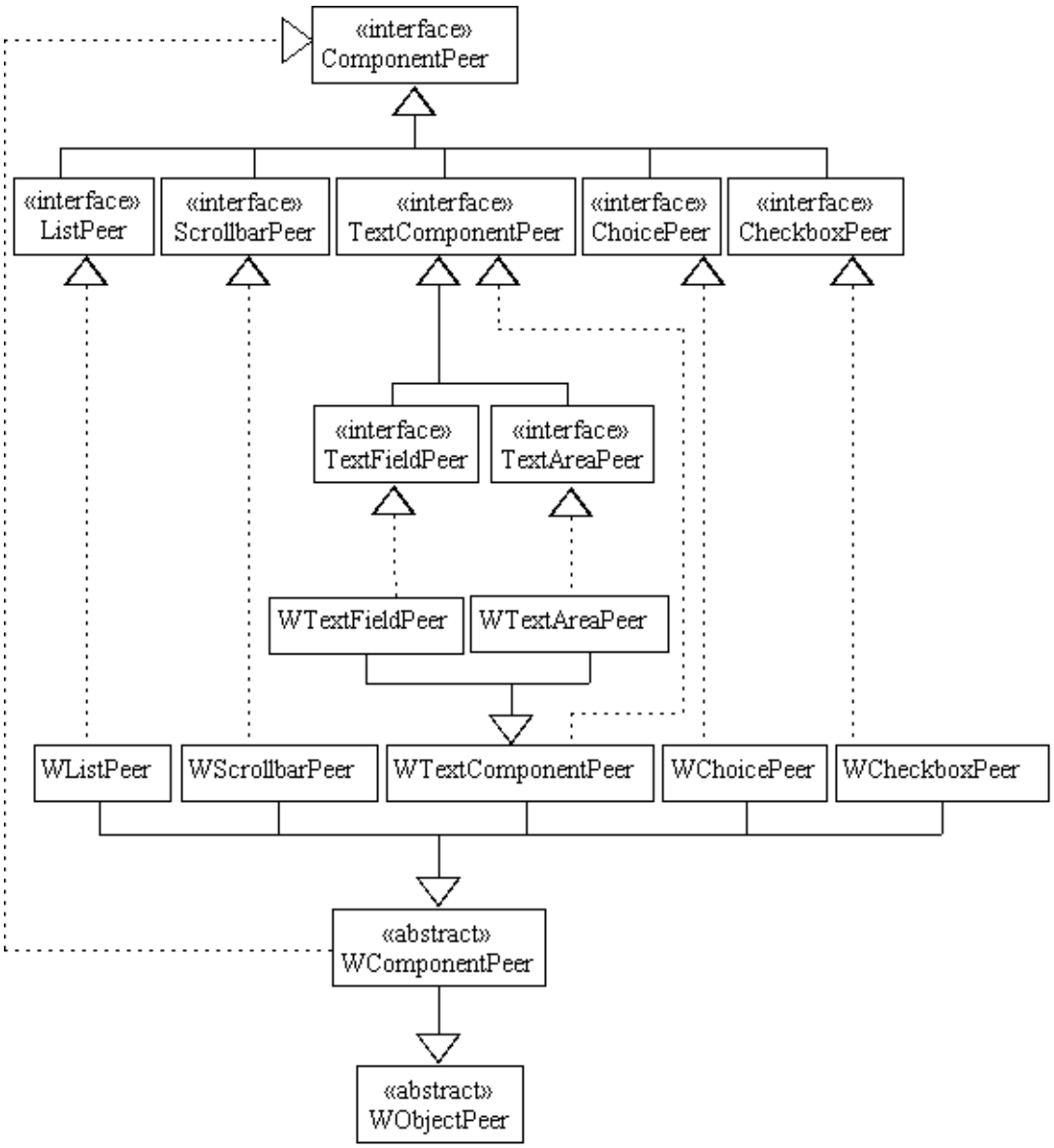
`ViewFactory` zde i přes své jméno hraje roli produktu. Jméno této třídy souvisí s tím, že jde o implementaci vzoru *Factory Method* pro tvoření objektů typu `javax.swing.text.View` pro nějakou komponentu. Pro různé typy dokumentů se vytvoří objekty různých podtříd `ViewFactory`, aby potom generovaly jen vhodné objekty typu `View`. Objekty `View` jsou odpovědné za zobrazení dokumentu.

`Document` slouží jako kontejner na text pro textové prvky rozhraní SWING. Standardně poskytuje např. metody pro vracení celého nebo části textu, nahrazení nebo vymazání části textu atd. Konkrétní implementace mohou přidat prvky, jako například dělení na kapitoly, sekce apod. a další vlastnosti různých formátů dokumentů.

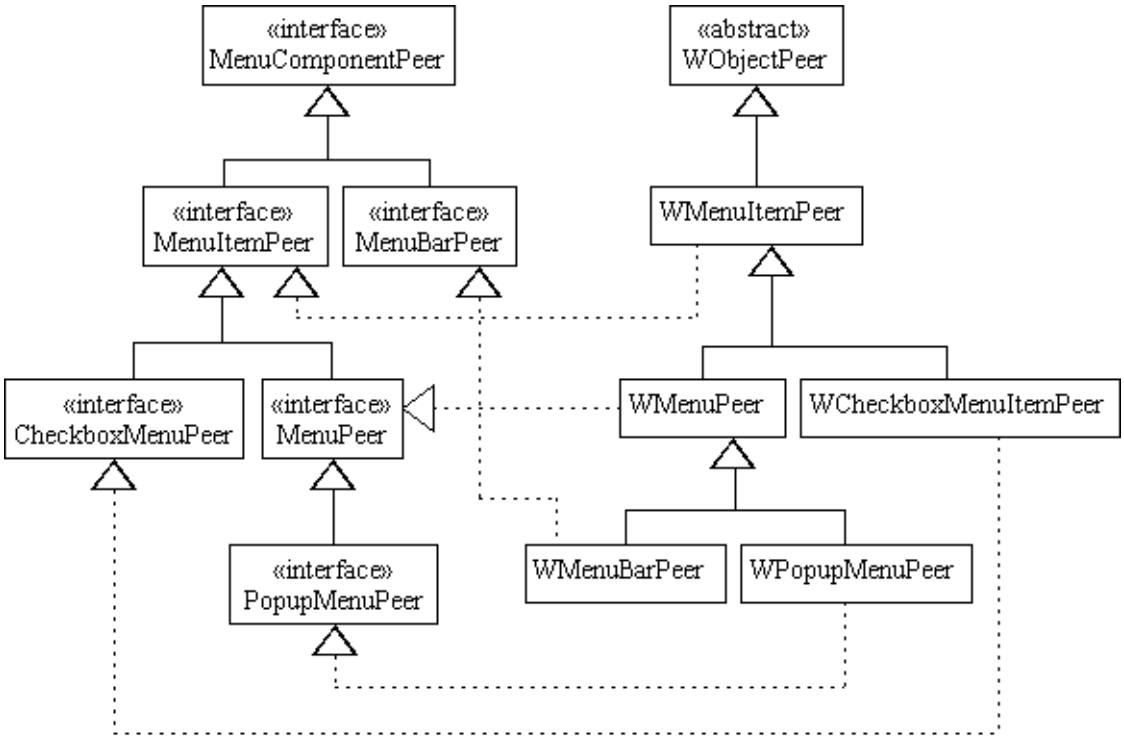
Konkrétní produktová třída `HTMLFactory` je vnitřní třídou `HTMLEditorKit`, obdobně `StyledViewFactory` je vnitřní třídou `StyledEditorKit` a `PlainEditorKit` je v `JEditorPane`. `PlainEditorKit` implementuje rozhraní `ViewFactory` a dědí z třídy `DefaultEditorKit`. To mu umožňuje se chovat jako továrna i produkt současně. Pokud je na takový objekt zavolána metoda `getViewFactory()`, vrací sám sebe. Nevzniká tak ani nová instance.



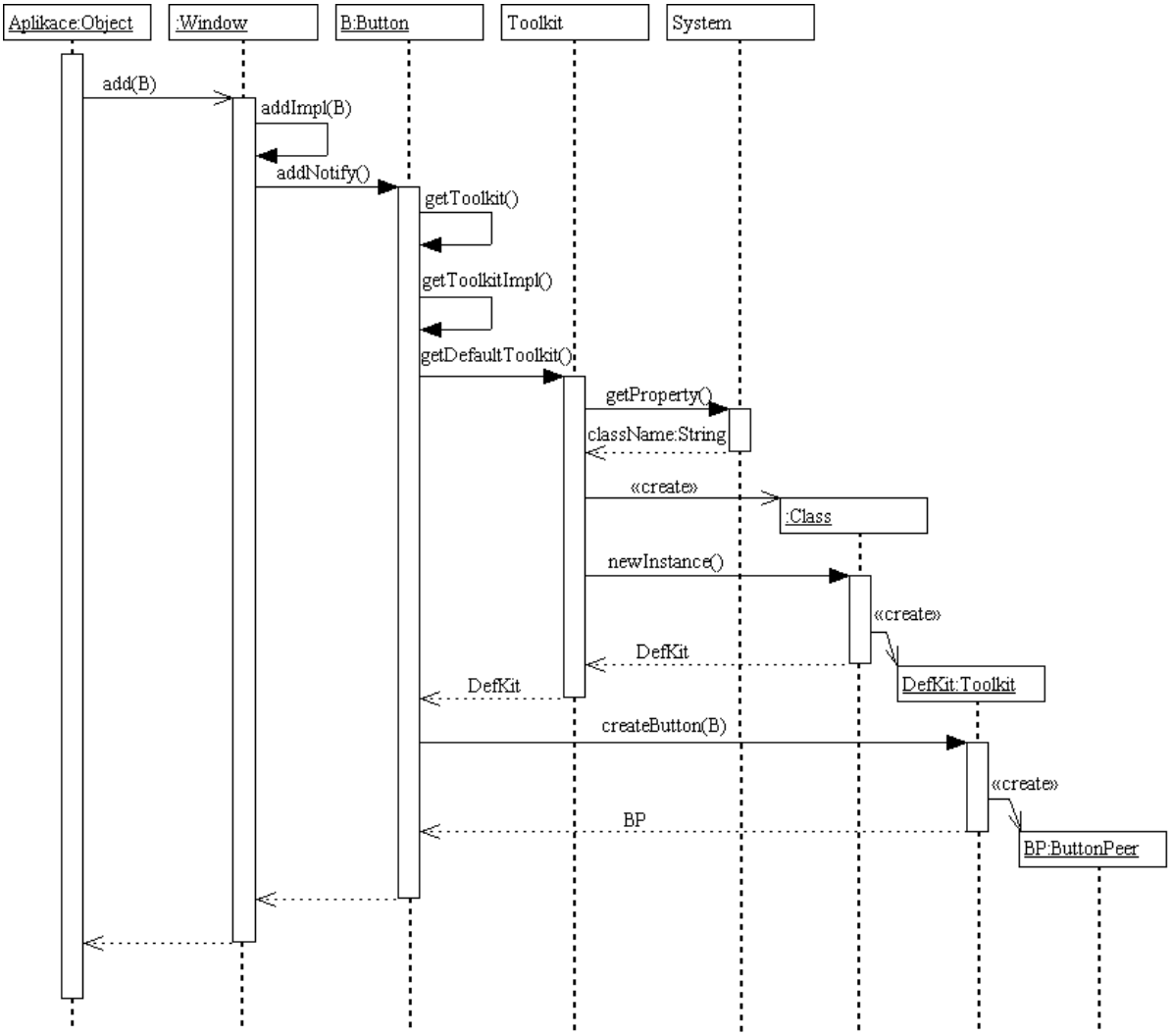
Obrázek 4: Diagram tříd - 1.část hierarchie produktů tvořených továrnou `Toolkit`



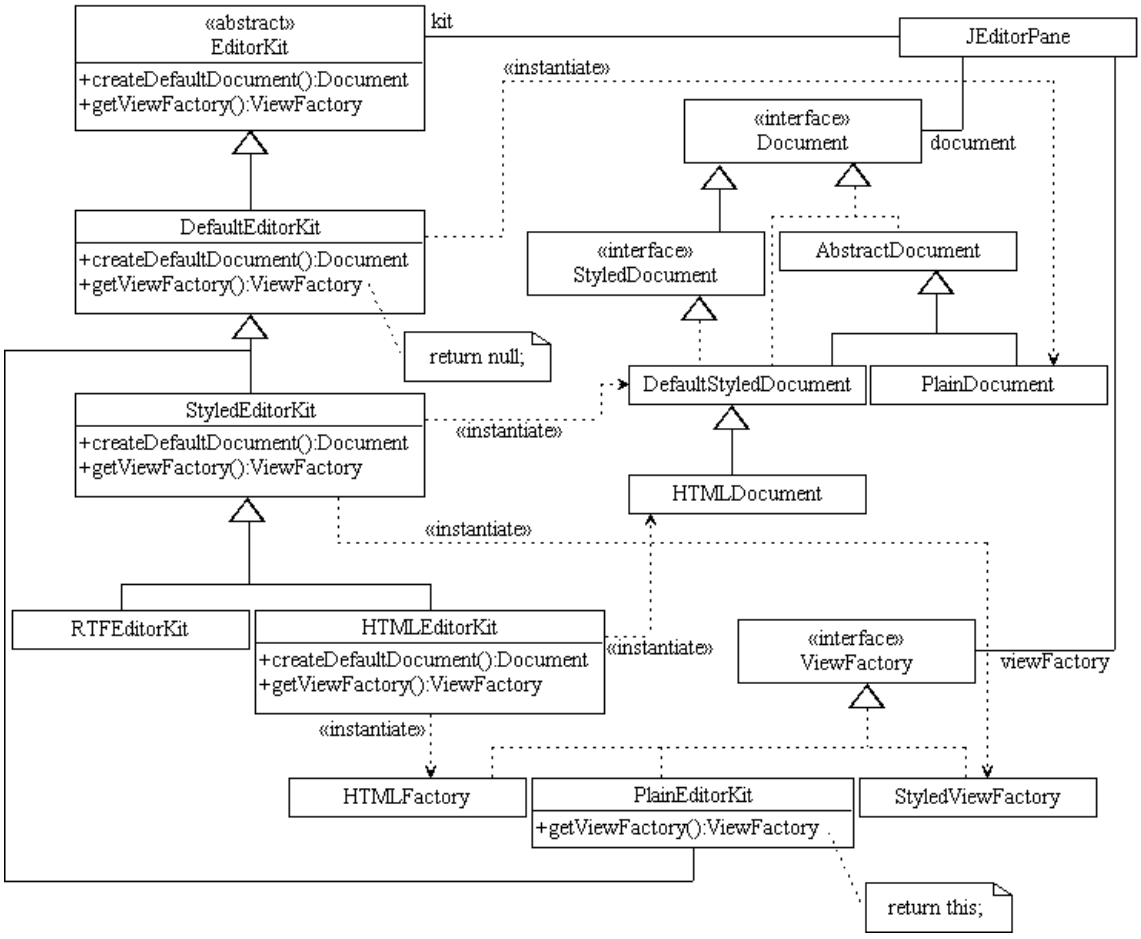
Obrázek 5: Diagram tříd - 2.část hierarchie produktů tvořených továrnou `Toolkit`



Obrázek 6: Diagram tříd - 3.část hierarchie produktů tvořených továrnou `Toolkit`



Obrázek 7: Diagram sekvencí pro vložení tlačítka do okna

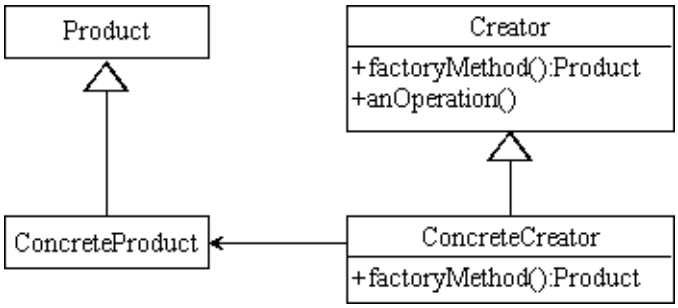


Obrázek 8: Diagram tříd pro EditorKit

3 Factory Method

3.1 Abstraktní popis

Návrhový vzor *Factory Method* patří stejně jako *Abstract Factory* do skupiny tvořících vzorů. Slouží k vytvoření instance jedné z několika možných tříd, často v závislosti na poskytnutých datech. Obvykle všechny třídy, které vrací, mají shodnou nadtřídu a metody, ale každá z nich provádí úkol jiným způsobem a je optimalizována pro jiný typ dat. Rozhodnutí o tom, kterou třídu vrátit nechává tvořící třída na své podtřídě a sama pracuje nezávisle na zvolené produktové třídě. Na obrázku 9 je abstraktní zobrazení tohoto vzoru.



Obrázek 9: Diagram tříd - abstrakce vzoru Factory Method

Někdy se za použití vzoru *Factory Method* považuje i situace, kdy o konkrétním produktu nerozhoduje podtřída, ale tvořící metoda tvořící třídy na základě dodaných dat. To je případ několika tříd v JAVA API obsahujících slovo „factory“. Rovněž v několika případech je v API definováno jen rozhraní tvořícího objektu, ale není k dispozici žádná implementace.

3.2 PopupFactory

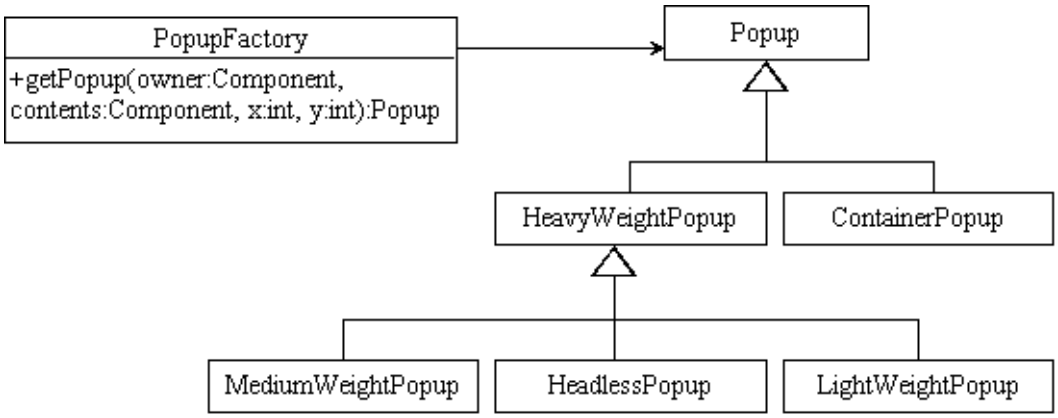
`PopupFactory` se nachází v balíku `javax.swing` a, jak jméno napovídá, je používána k získávání instancí třídy `Popup`. Ty jsou užívány k zobrazení prvku (instance `Component`) nad všemi ostatními prvky.

`Popup getPopup(Component owner, Component contents, int x, int y)` rozhoduje o konkrétní podtřídě třídy `Popup` na základě svých parametrů. Všechny podtřídy jsou vnitřními třídami `Popup`.

Situace je zobrazena diagramem tříd na obrázku 10.

3.3 KeyFactory

Třída `KeyFactory` se nachází v balíku `java.security`. Slouží k převodu specifikací kryptografických klíčů (instance `KeySpec`) na objekty reprezentující klíče samotné (instance `Key`) a opačně. Metody



Obrázek 10: Diagram tříd - `PopupFactory`

```
PrivateKey generatePrivate(KeySpec keySpec)
PublicKey generatePublic(KeySpec keySpec)
```

tedy rozhodují o konkrétní návratové třídě klíče podle dodané specifikace v argumentu. Diagram tříd zobrazující třídy generované touto továrnou je na obrázku 11.

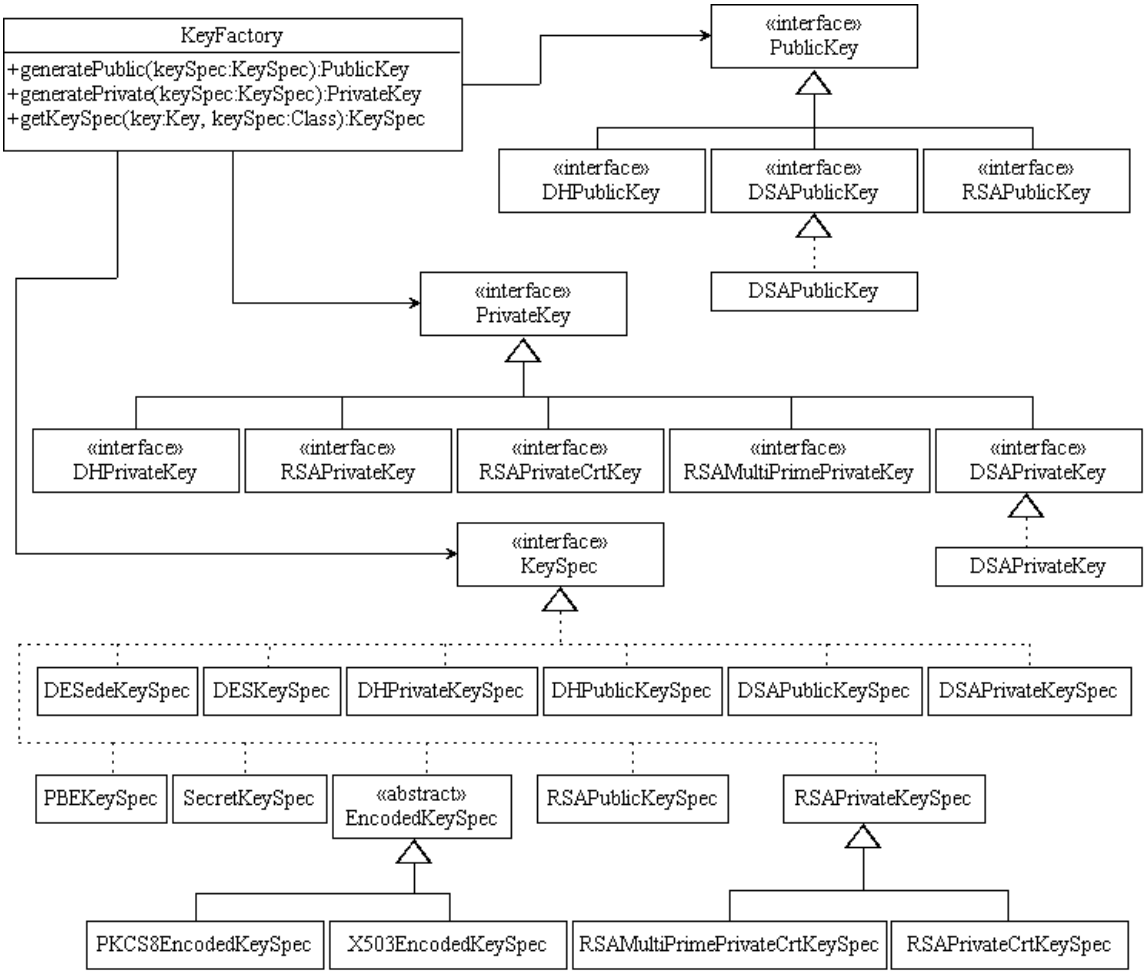
Třída `KeyFactory` při tvorbě klíčů využívá `KeyFactorySpi`. Tato třída je abstraktní a ve zdrojových kódech obsažených v distribucí JDK a tím ani v API dokumentaci není žádná její podtřída. Také tam není žádná implementace rozhraní odvozených z `PrivateKey` a `PublicKey`. Mezi binárními třídami jsou však v balíku `sun.security.provider` implementace rozhraní `DSAPublicKey` a `DSAPrivateKey` pod stejným jménem. Také tam existuje podtřída `KeyFactorySpi` se jménem `DSAKeyFactory`. Pro ostatní protokoly by musely při použití být podobné třídy vytvořeny.

3.4 SocketImplFactory

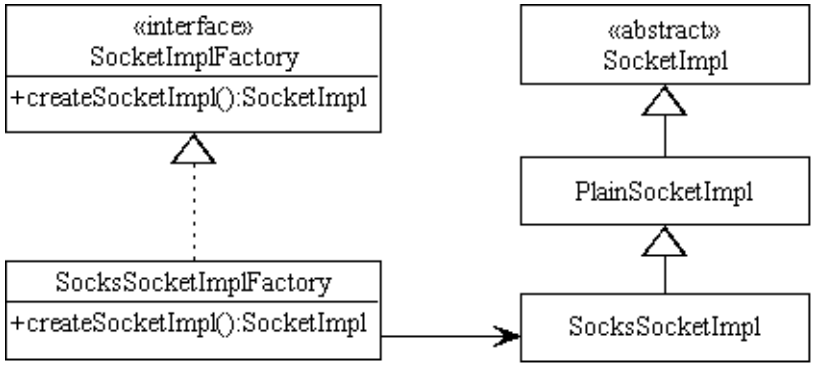
Třída `Socket` reprezentuje klientský konec síťového spojení. Skutečnou práci ale provádějí objekty tříd implementujících `SocketImpl`. `Socket` si takový objekt vytvoří pomocí metody `SocketImpl createSocketImpl()` továrny `SocketImplFactory`. Diagram tříd pro tuto aplikaci *Factory Method* je na obrázku 12.

3.5 DocumentBuilderFactory

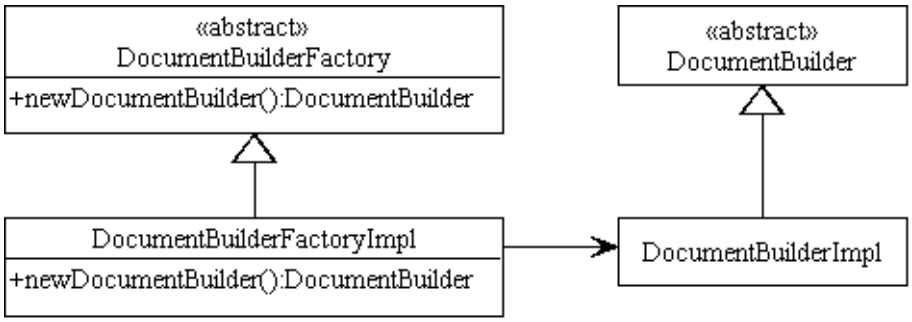
Třída `DocumentBuilderFactory` definuje továrnu, která umožňuje aplikacím získat parser. Ten potom vytváří stromy DOM objektů z XML dokumentů. Jak je vidět na diagramu tříd na obrázku 13, k dispozici je jedna neabstraktní implementace tvořícího objektu i produktu. Obě se nacházejí v balíku `org.apache.crimson.jaxp`.



Obrázek 11: Diagram tříd - `KeyFactory`



Obrázek 12: Diagram tříd - `SocketImplFactory`



Obrázek 13: Diagram tříd - DocumentBuilderFactory

4 Závěr

V tomto textu je uvedeno jen několik příkladů použití návrhových vzorů *Factory* nebo *Factory Method* v API jazyka JAVA. Ostatní aplikace tohoto vzoru jsou obdobné. Často tovární třídu využívají jiné třídy z API a aplikační programátor tvořící metody přímo neužije. Velký význam má hlavně `Toolkit`, který také pracuje skrytě před programátory a zajišťuje možnost psát platformově nezávislé aplikace s grafickým uživatelským rozhraním.