**ISO / IEC 9126 Kategorie**

- Functionality
  - -Existence soubor funkcí a jejich vlastnostmi
- Reliability
  - - Schopnost softwaru udržet úroveň výkonu za stanovených podmínek pro uvedené období
- Usability
  - -Úsilí potřebné pro použití
- Efficiency
  - - Vztah mezi úrovní výkonu softwaru a množstvím zdrojů, používán na základě stanovených podmínek.
- Maintainability
  - - Nutné usilovat o to, aby šel software dále upravovat
- Portability
  - - Převoditelnost z jednoho prostředí do druhého

**Reliability**

- Maturity
  - – ... Frekvence selhání softwaru.
- Fault Tolerance
  - –Odolnost software proti selhání.
- Recoverability
  - –Schopnost přivést zpět selhaný systém do plného provozu, včetně dat a síťových připojení.
- Reliability Compliance
  - –"... dodržuje normy, konvence nebo předpisy"

**Efficiency**

- Time Behavior
  - –Doba odezvy pro dané transakce.
- Resource Utilization
  - – Charakterizuje použité zdroje, tj. paměť, CPU, disk a síťové využití.
- Scalability
  - –Vztah mezi Time Behavior a Resource Utilization

**Maintainability**

- Analyzability
  - – Schopnost identifikovat základní příčinu selhání v softwaru.
- Changeability
  - – Charakterizuje množství úsilí změnit systém.
- Stability
  - – Charakterizuje citlivost na změnu určitého systému, který může mít negativní dopad, který může být způsoben systémovými změnami.
- Testability
  - – Charakterizuje úsilí potřebné k ověření (test) systémové změny.

**Portability**

- Adaptability
  - – Schopnost systému měnit se dle nových předpisů nebo provoznímu prostředí.
- Replacability
  - – Plug and Play aspekt softwarových komponent, aby bylo snadné je přenést do stanoveného prostředí.

**Interpreted vs. machine-code**
»(+) Nezávislost na platformě - architektura (RISC / CISC), OS
»(+) Reflexe - sledovat, upravovat vlastní strukturu za běhu
»(+) Dynamické psaní - při spuštění (JAVA nemá dynamické typování!)
»(+) Malé rozměry
»(+) Dynamický rozsah - (Perl)
» (-) Pomalejší provedení - interpretovat režimu JIT latence

**Techniky ochrany**
»Chraňte svůj kód
- »Žádný fyzický přístup k programu - např. klient-server modelu
- »Šifrování kódu - pokud není v hardwaru, může být zachycena
- »Nativní kód místo byte-kódu
- » code obfuscation

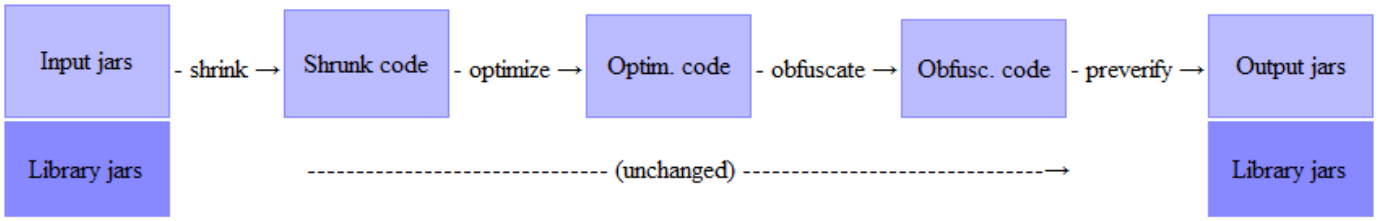
**Obfuscator**
Je to konverzní softwarová pomůcka, která převádí zdrojový kód konkrétního programovacího nebo skriptovacího jazyku do téhož zdrojového kódu v témže jazyku, ale provede v něm několik změn. Typicky:

- odstraní komentáře a dokumentaci uvnitř kódu
- zruší formátování kódu tím, že vymaže veškeré „bílé místo“ (white space)
- přejmenuje identifikátory proměnných popř. i konstant, někdy i uživatelských funkcí
- popř. další nadstandardní zásahy do kódu (definování a použití vlastních funkcí pro potřeby obfuskace)

Účelem obfuskátoru je zatemnit daný zdrojový kód, t.j. co nejvíce znesnadnit jeho „čitelnost“ pro člověka – odmazáním komentářů vysvětlujících, co kód dělá; zničení formátování kódu, zrušení odsazení, indikující hierarchii jednotlivých příkazů a jejich příslušnost do syntaktických struktur kódu, naruší způsob, kterým oko člověka znalého syntaxe daného programovacího jazyka je zvyklé daný kód číst. Přejmenování proměnných je obdobným krokem – názvy proměnných u programů s dobrou štábní kulturou naznačují, na co jsou tyto proměnné používány.

Podmínkou smyslu existence obfuskátoru je současně to, aby zdrojový kód byl i po konverzi čitelný pro překladač nebo interpret svého respektivního jazyku.

Důvod, proč se obfuskátory používají, je zamezení třetí osobě, která by ke zdrojovému kódu mohla získat přístup; aby tento program mohla snadno rozvíjet, dělat na něm úpravy (například i ty za účelem odstranění různých ochran heslem nebo licenčními klíči) apod.




**Java Virtual Machine**
Java Virtual Machine (JVM) je sada počítačových programů a datových struktur, která využívá modul virtuálního stroje ke spuštění dalších počítačových programů a skriptů vytvořených v jazyce Java. Úkolem tohoto modulu je zpracovat pouze tzv. mezikód, který je v Javě označován jako Java bytecode. Odhaduje se, že od roku 2006 byl JVM nainstalován na více než čtyřech miliardách stanic po celém světě.

Java Virtual Machine umí zpracovat mezikód (Java bytecode), který je obvykle vytvořen ze zdrojových kódů programovacího jazyka Java. Mezikód však může být vytvořen i z jiných jazyků než je Java. Příkladem zdrojového kódu, který může být přeložen do Java bytecodu je zdrojový kód jazyka Ada. Virtuální stroj JVM mohou využívat kromě Sun (firma vyvíjející programovací jazyk Java) i jiné společnosti, které se zabývají vývojem programovacích jazyků. JVM je sice zahrnutý v ochranné známce „Java“, ale může být vyvíjen i jinými společnostmi, pokud budou dodrženy podmínky a související smluvní závazky vydané společností Sun.

Java Virtual Machine je důležitý pro práci programů vytvořených v Javě. Díky tomu že je JVM k dispozici na mnoha platfomách, je možné aplikaci v Javě vytvořit pouze jednou a spustit na kterékoliv z platforem, pro kterou je vyvinut JVM (např. Windows, Linux). JVM umožňuje automatické zpracování výjimek, díky kterým dokáže určit hlavní příčinu chyby nezávisle na zdrojovém kódu.

JVM je dodáván spolu se sadou standardních knihoven, které jsou nazývány Java API (Application Programming Interface). Application Programing Interface je systém, který umožňuje programu pracovat s funkcemi a třídami knihoven. JVM a API společně tvoří celek, který je poskytován jako Java Runtime Environment (JRE).

*Spouštěcí prostředí*

- Aby mohla být aplikace spuštěna v JVM musí být zkompilována do standardizovaného a přenosného binárního formátu, který je obvykle ve formě .class souborů. Aplikace může být složena z mnoha různých druhů souborů, proto je možné pro snadnější distribuci rozsáhlých aplikací, zabalit více Javovských tříd do jednoho souboru typu .jar.

- Spustitelné jsou v JVM soubory s příponou .class nebo .jar. JVM je takzvaný interpret umožňující vykonávat zdrojový kód programu přeložený do mezikódu a zprostředkovat tak komunikaci s platformou na které je JVM spuštěn. V pozdějších verzích Javy nebyl mezikód přímo interpretován, ale před prvním svým spuštěním dynamicky zkompilován do strojového kódu daného počítače (tzv. just in time compilation - JIT). V současnosti se převážně používají technologie zvané HotSpot compiler, které mezikód zpočátku interpretují a na základě statistik získaných z této interpretace později provedou překlad často používaných částí do strojového kódu včetně dalších dynamických optimalizací (jako je např. inlining krátkých metod atp.).

**JIT (akronym pro Just In Time)**
Je v informatice označení pro speciální metodu překladu využívající různé techniky pro urychlení běhu programů přeložených do mezikódu (např. CIL). Používá se například pro programovací jazyk Java. Program, který je spuštěn a prováděn, je v době provádění přeložen přímo do nativního strojového kódu počítače, na kterém je prováděn, čímž dochází k urychlení jeho běhu. Negativem této techniky je prodleva, kterou JIT kompilátor (nikoli interpret) stráví překladem do nativního kódu, a proto se do nativního kódu často překládají jen mnohokrát (řádově 10 000×) volané úseky programu. Hlavním problémem JIT je, že má málo času na provedení své práce. Tyto nevýhody lze eliminovat použitím trvalé cache. Naopak výhodou je, že je možné lépe optimalizovat pro daný procesor a využít jeho rozšířených instrukcí.

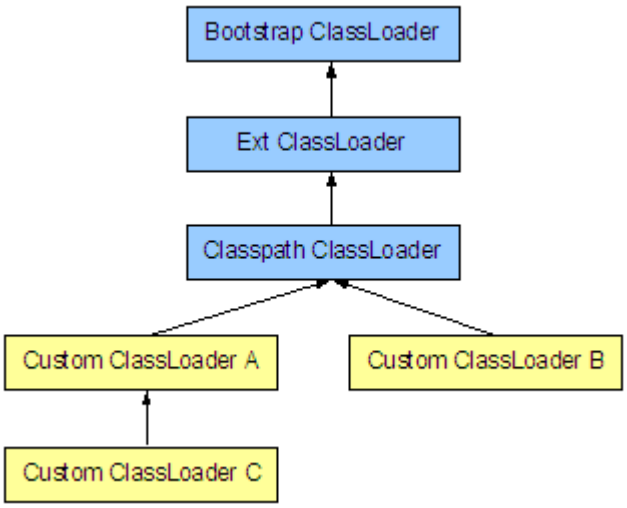
**Classloader**
Classloader (abstraktní třída java.lang.ClassLoader), jak již jeho název napovídá, je zodpovědný za nahrávání tříd. Pokud si kladete otázku, proč je vlastně classloader potřeba a proč to nedělá Java tak nějak automaticky, tak odpověď je, že dělá. Nicméně aplikace mají různé potřeby, představte si například aplikační server, který musí umět deploynout aplikaci za běhu. Jedním z požadavků, který classloader umožňuje realizovat, je například dynamické nahrávání tříd za běhu aplikace.

Dalším případem užití classloaderu může být nahrání třídy z různých úložišť, disku, sítě a nebo z databáze. Z tohoto důvodu java.lang.ClassLoader definuje metodu defineClass, která dostává definici třídy jako pole bytes.

JVM zaručuje, že jednou nahraná třída nebude vícekrát nahraná. Každá třída je uvnitř JVM identifikovaná plně kvalifikovaným jménem, tedy package a název třídy a classloaderem, který ji nahrál. To je velice důležité, pokud budeme mít například třídu Foo v package hoo, kterou nahrál classloader C1 a třídu Foo v package hoo, kterou nahrál classloader C2, jedná se o dvě rozdílné třídy. Pokud bych jejich instance zkusili přetypovat, dostali bychom ClassCastException.

Z historického hlediska je zajímavostí, že Java 1.1 nepoužívala pro identifikaci plně kvalifikované jméno třídy a classloader, který ji nahrál, ale pouze plně kvalifikované jméno. Jak se záhy ukázalo, bylo to typově nebezpečné, protože mohlo docházet podvržení tříd viz článek Java is not type-safe.

Hierarchie classloaderu je stromová a díky tomu umožňuje delegování nahrání tříd. Hierarchie classloaderu také zaručuje omezenou viditelnost třídy v prostoru, který classloader definuje. Classloader umožňuje v daném prostoru vidět třídy, které nahrál on sám a nebo některý z jeho předků a naopak nevidí na třídy, které nahrál kterýkoliv z jeho potomků. Vztah předek-potomek není myšlen na úrovni dědičnosti, ale na úrovni vazeb mezi objekty.




*Bootstrap classloader* - boostrapový classloader je zodpovědný za nahrání základních tříd Javy

*Ext classloader* - classloader pro Java rozšíření, který nahrává třídy z extension adresáře Javy

*Classpath classloader* - classloader nahrává třídy uvedené na classpath

*Custom Classloader* - implementace classloaderu, který může nahrávat třídy z libovolného uložište a defacto libovolným způsobem

**Immutable Object (neměnný objekt)**
Neměnnost třídy úzce souvisí se seznamem atributů a hodnot v nich uložených. Neměnné třídy se vyznačují tím, že žádné jejich metody nemění hodnoty atributů. Jakmile se do atributu neměnného objektu v konstruktoru přiřadí nějaká výchozí hodnota, zůstává tato hodnota stejná během celé doby existence objektu, tedy až do jeho odstranění z paměti. Pokud se má některá hodnota neměnného objektu změnit, je nutné vytvořit novou instanci objektu.

Neměnnost je takřka nutností pro kvantity v matematických výrazech a objekty, které mají být uloženy v kolekci využívající metody hashCode() a equals(). Neočekávané komplikace totiž mohou přijít, je-li rovnost dvou objektů závislá na hodnotě měnitelného atributu a tento atribut se změní.

Výhodou neměnnosti je spolehlivost, stabilita a implicitní vláknová bezpečnost. Vlákna se nemusí v přístupu k atributům a metodám objektu nijak synchronizovat, protože se žádné atributy nemění. Další výhodou je možnost bezpečného sdílení všech instancí neměnných tříd v rámci celé aplikace a to bez obav z toho, že by někdo tyto instance modifikoval.

Nevýhodou je naopak nutnost vytvářet mnoho instancí velmi malých objektů. S tím ale moderní platformy nemají žádný větší problém.

Neměnné třídy často obsahují různé tovární metody, které vytváří odvozené instance, ve kterých jsou provedeny požadované změny nebo představují výsledek provedené operace. Atributy původní instance neměnné třídy se přitom nikdy nemění.

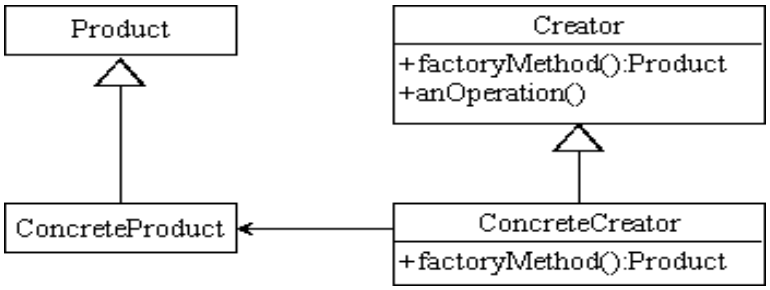
*Postup implementace neměnné třídy*
- nadeklarovat třídu a její atributy
- všechny atributy označit modifikátorem final
- vygenerovat konstruktor, který všechny atributy inicializuje
- podle potřeby doplnit metodu hashCode() a equals()

*Příklad*
- Třída String
  Třída String ve standardní knihovně jazyka Java je neměnná. V řetězci není možné změnit ani jeden znak. Všechny operace které v řetězci provádí změny ve skutečnosti vrací nové instance třídy String a původní řetězec nechávají nedotčený.

```
String hello = "hello";
String helloUC = hello.toUpperCase();
```

**Factory Method**

- Patří mezi tvořící návrhové vzory
- Slouží k vytvoření jedné z několika možných tříd
- Třídy, které vrací, mívají stejné rozhraní a nadtřídu, ale každá úkoly provádí jiným způsobem
- Rozhodnutí o tom, kterou třídu vrátit, nechává tvořící třída na podtřídě, nebo o konkrétním produktu rozhoduje tvořící metoda na základě poskytnutých dat




*Příklad*

product interface

```
public interface Trace {
      // turn on and off debugging
      public void setDebug( boolean debug );
      // write out a debug message
      public void debug( String message );
      // write out an error message
      public void error( String message );
}
```

product A

```
public class SystemTrace implements Trace {
      private boolean debug;
      public void setDebug( boolean debug ) {
            this.debug = debug;
      }
      public void debug( String message ) {
            if( debug ) {  // only print if debug is true
                  System.out.println( "DEBUG: " + message );
            }
      }
      public void error( String message ) {
            // always print out errors
            System.out.println( "ERROR: " + message );
      }
}
```

**Lazy initialization**

Již z názvu lze postřehnout, že se jedná o odložení vytváření objektu, počítání hodnoty nebo provádění nějakého procesu, až do chvíle, kdy ho budeme poprvé potřebovat.

*Účel*

- V programování se často setkáváme se situacemi, kdy na jedné straně pracujeme s několika velkými objekty a na druhé potřebujeme program zoptimalizovat. Je sice možnost ty objekty inicializovat již od začátku, ale co když ho budeme potřebovat jenom jednou a někdy ke konci programu? Nebo svými podmínkami ten objekt celý program přeskočí a nevyužije vůbec? Tudíž vytváření takového objektu hned od začátku je nevhodné a východiskem z této situaci je tak zvaná odložená inicializace.

- Odložená inicializace je pak často využívá s jinými návrhovými vzory, jako jsou Tovární metoda (Factory Method), Jedináček (Singleton) nebo Proxy.

*Plusy*

- Inicializace objektu probíhá ve chvíli kdy je opravdu nutná
- Zrychluje se počáteční inicializace

*Minusy*

- Není možné vytvořit pořadí inicializovaných objektů
- Vzniká odezva při dotazování se na objekt

*Příklad*

```
// Řádně synchronizovaná odložená inicializace
private static Foo foo = null;

public static synchronized Foo getFoo() {
        if (foo == null)
                foo = new Foo();
        return foo;
}
```

**Singleton**

Je název pro návrhový vzor, používaný při programování. Využijeme ho při řešení problému, kdy je potřeba, aby v celém programu běžela pouze jedna instance třídy. Tento návrhový vzor zabezpečí, že třída bude mít jedinou instanci a poskytne k ní globální přístupový bod. Singleton je také často využíván jako součást jiných návrhových vzorů jako jsou například Flyweight nebo Facade.

*Účel*

- Nutnost existence jediné instance se objevuje například tam, kde potřebujeme, aby se nějaké objekty pohybovaly jen ve vymezeném prostředí – hráči fotbalu hrající na jednom hřišti. Třída definující hřiště vytváří svou instanci jako jedináček. Dalším příkladem mohou být dialogová okna nebo ovladače zařízení[1]. Známým příkladem ze světa Windows je schránka, která může existovat jen jednou, aby se nám data získaná v jedné aplikaci neztratila někde po cestě do druhé aplikace.

*Implementace*

```
public class Singleton {

     private static Singleton instance;

     //Vytvorime soukromy konstruktor
     private Singleton() { }

     //Metoda pro vytvoreni objektu jedinacek
     public static Singleton getInstance() {
     //Je-li promenna instance null, tak se vytvori objekt
         if (instance == null) {
             instance = new Singleton();
         }
         //Vratime jedinacka
         return instance;
     }

     public void pozdrav() {
         System.out.print("Ahoj ja jsem jedinacek.");
     }

     //Pouziti
     public static void main(String[] args) {
         Singleton objekt = Singleton.getInstance();
         objekt.pozdrav();
     }
 }
```

**Multiton**

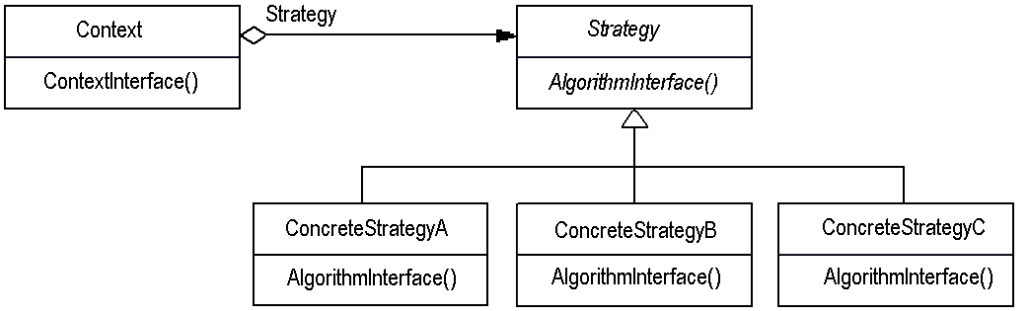
Návrhový vzor Multiton je obdobou návrhového vzoru Singleton. Zásadní rozdíl spočívá v tom, že zatímco u Singletonu existuje pouze jediná instance třídy, u Multitonu existuje jedna instance pro každý klíč (každý rozlišitelný stav). Výhodou tohoto přístupu jsou nižší nároky na pamět a vytvoření vždy jen jedné instance pro každý stav (vytváření může být nákladné).

*Příklad*

```
public static class MyMultiton {
    private static final Map<Object, MyMultiton> instances = new HashMap<Object, MyMultiton>();

    private final Object attribute;

    private MyMultiton(final Object attribute) {
        this.attribute = attribute;
    }

    public static MyMultiton getInstance(Object attribute) {
        synchronized (instances) {
            MyMultiton instance = instances.get(attribute);
            if (instance == null) {
                instance = new MyMultiton(attribute);
                instances.put(attribute, instance);
            }
            return instance;
        }
    }
}
```

**Strategy**

Návrhový vzor Strategy zapouzdřuje nějaký druh algoritmů nebo objektů, které se mají měnit, tak aby byly pro klienta zaměnitelné.

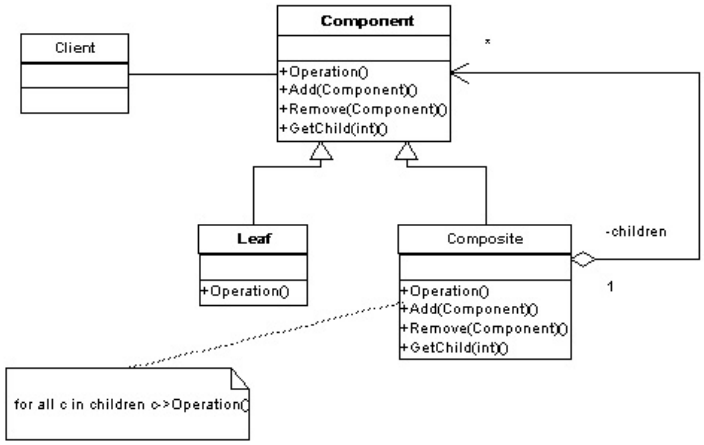



*Příklad*

```
public class Main {

    public static void main(String[] args) {
        Bytost b = new Bytost(new Clovek());
        b.promluv();

        Bytost p = new Bytost(new Pes());
        p.promluv();
    }

}
interface Strategy{
    void promluv();
}
class Clovek implements Strategy{

    public void promluv() {
        System.out.println("člověk
promluvil");
    }

}
class Pes implements Strategy{

    public void promluv(){
        System.out.println("pes zaštěkal");
    }
}
class Bytost{
    private Strategy strategy;

    Bytost(Strategy st){
        this.strategy = st;
    }

    public void promluv() {
        strategy.promluv();
    }
}
```

**Composite**

Návrhový vzor Composite představuje řešení, jak uspořádat jednoduché objekty a z nich složené (kompozitní) objekty. Snahou vzoru je, aby k oběma typům objektů (jednoduchým a složeným) bylo možné přistoupit jednotným způsobem.




**Iterator**

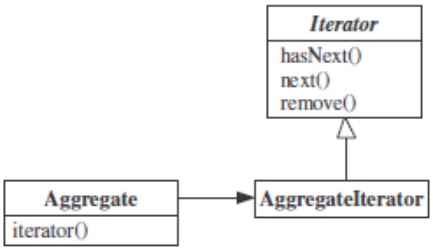
Návrhový vzor Iterator zajišťuje možnost procházení prvků bez znalosti jejich implementace. Lze implementovat pomocí pole nebo jiných datových struktur. Pomocí datové struktury ArrayList je implementace daleko snazší.

*Motivace*

- Iterátor použijeme v několika částečně se překrývajících situacích.

*Průchod složitého objektu*

- První z nich je, chceme-li procházet objekt, jehož struktura není na první pohled zřejmá. Příkladem může být třída Rodina, ve které existuje velké množství vazeb (rodič, potomek, švagr, teta, sestřenice...). Rozhraní iterátoru nám zaručí, že projdeme systematicky všechny členy rodiny (žádného nevynecháme, ani nezpracujeme vícekrát).

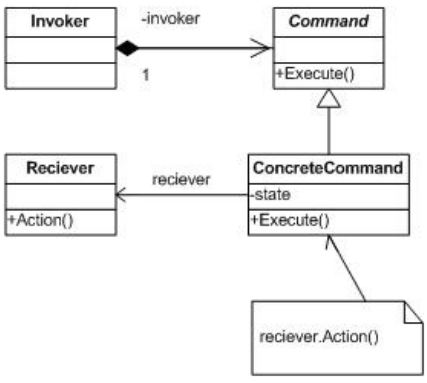



*Jednotné rozhraní*

- Druhou situací je, když chceme vytvořit jednotné rozhraní pro mnoho různých kolekcí. Toto rozhraní pak implementujeme například pro spojový seznam, strom, dynamické pole a podobně. V programu má pak uživatel možnost procházet všechny kolekce jednotným způsobem, aniž by se musel zajímat o to, jak jsou data uložená.

**Command**

Zapouzdřete požadavek jako objekt a tím umožněte parametrizovat klienty s různými požadavky, frontami nebo požadavky na log a podporujte operace, které jdou vzít zpět. – např.: Thread Pool

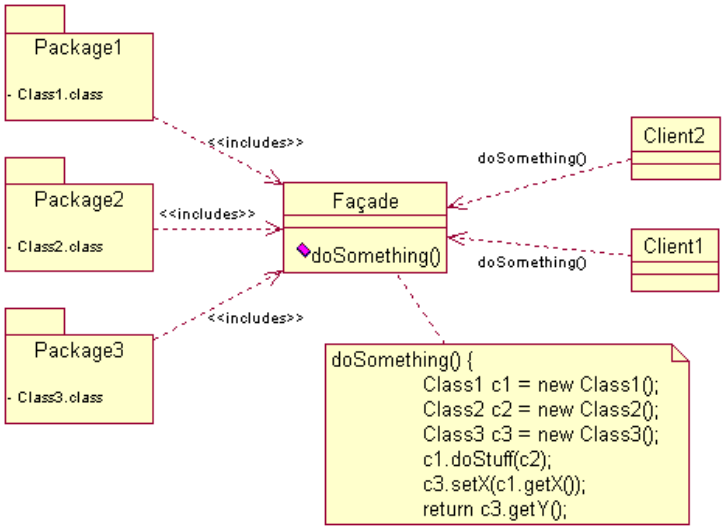


**Facade**
Fasáda je objekt, který nabízí zjednodušené rozhraní pro větší skupiny kód, jako je knihovně tříd .

Fasáda může:

- vytvořit softwarové knihovny snadněji používatelné, pochopit a vyzkoušet, protože fasáda má pohodlné metody pro běžné úkoly;
- Knihovny lépe čitelnější;
- snížení závislosti na venkovním kódu na vnitřním fungování knihovny, protože většina kódu používá fasádu, což umožňuje větší flexibilitu při vývoji systému;
- zabalit špatně navržený soubor API s jedním dobře navržené rozhraní API (podle potřeby úkolu).

Adaptér se používá, když wrapper musí respektovat určité rozhraní a musí podporovat polymorfní chování. Na druhé straně se fasáda používá, když chce někdo jednodušší a přehlednější rozhraní pro práci.




*Facade Vs. Interface*

- Rozhraní (ve smyslu Java) je mnohem omezenější než Facade
- Pouze definuje contract mezi library/implementing class a pouzivá ho programátor
- Vzor fasáda umožňuje aktivní ovládání složitějších problémů

**Wrapper Facade**
Návrhový vzor zapouzdřuje funkce a údaje,
které nejsou objektově orientované rozhraní API do více stručných, robustní, přenosné, udržovatelné a soudržné objektově orientované rozhraní třídy.

**Proxy**
Proxy pattern využijeme, jestliže potřebujeme zajistit kontrolu nad přístupem k jinému objektu.

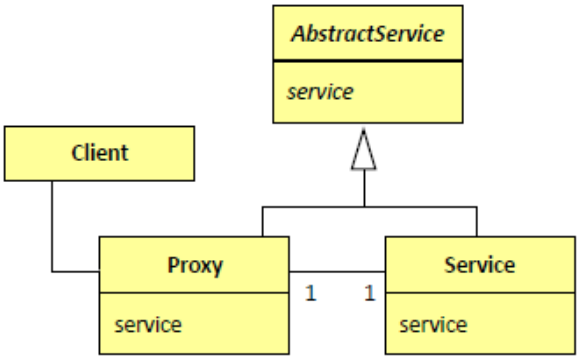
Můžeme identifikovat čtyři obecné situace, kdy potřebujeme kontrolovat přístup k jinému objektu:

- *Virtual proxy* - jestliže je vytvoření reálného (kontrolovaného objektu) moc náročné, je tento vytvářen, až je to skutečně nutné. Proxy implementuje logiku, která rozhodne o nutnosti vytvoření reálného objektu.

- *Remote proxy* - poskytuje lokální reprezentaci objektu, který je umístěn na vzdáleném stroji respektive v jiném adresním prostoru.

- *Protective proxy* - kontroluje přístup k zapouzdřenému(kontrolovanému) objektu. Před přesměrováním volání reálného objektu, jsou zkontrolována práva na užití tohoto objektu.

- *Smart proxy* - poskytuje dodatečné operace před volání kontrolovaného objektu. Jedná se například o zjišťování počtu referencí na vzdálený objekt. Jestliže na objekt neexistuje reference, může být uvolněn z paměti. Pokud je objekt uchován na perzistentním médiu, může proxy zajisti jeho nahrání do paměti. Dále proxy může zajišťovat správu zámků, aby objekt nemohl být používán více klienty zároveň.



**Active Object**
Návrhový vzor Active Object patří do skupiny návrhových vzorů pro paralelní programování. Odděluje volání metody od vlastního provedení metody. Typickou situací pro jeho použití je, když je třeba z jednoho vlákna (či více vláken) volat metody objektu, který běží v jiném (svém vlastním) vláknu, tedy volat je asynchronně a zároveň zajistit, aby byly provedeny ve správném pořadí a ve správné chvíli – tj. když nastanou podmínky umožňující jejich spuštění.

*Použití*

- Tento návrhový je vhodné použít, když z různých vláken chceme volat metody jednoho objektu tak, aby tyto metody nezdržovaly nebo nezastavovaly běh těchto vláken, a zároveň chceme mít možnost ovlivňovat pořadí, v jakém objekt metody „zpracuje" a samozřejmě se při tom vyhnout konfliktům jako race condition nebo deadlock.

**Reactor**
Reaktor architektonický vzor umožňuje událostmi řízené aplikace demultiplexovat a odeslání požadavků na služby, které jsou dodávány do aplikace z jednoho nebo více klientů.

**Proactor**
Proactor architektonický vzor umožňuje události řízené aplikace efektivně demultiplexovat a odeslání žádostí o službu, která si vyžádalo dokončení asynchronní operace, aby se dosáhlo výkonu výhody souběžnosti aniž by některé z jejích závazků.

**Threads**
»Procesy vs vlákna
- »Oba podporují souběžné provádění
- »Jeden proces má jedno nebo více vláken
- »Vlákna mají stejný adresní prostor (data a kód)
- »Přepínání kontextu mezi vlákny je levnější
- »vzájemná komunikace je poměrně účinná

»Vlastní lokální proměnné
»Vlastní metoda parametry
»Vytvoření vlákna
- »Podtřída java.lang.Thread
- »Realizace java.lang.Runnable

- » new
  - » po vytvoření
- » runnable
  - » start()
- » blocked
  - » Čekání na zámek, znovu synchronizovat metodu/blok
- »Čeká ní(může být přerušen)
  - » o.wait(), t.join(), LockSupport.park()
- »Časované čekání (může být přerušen)
  - » Thread.sleep(x), o.wait(x), t.join(x)
  - » LockSupport.parkNanos(x), LockSupport.parkUntil(time)
- » Ukončení
  - » finished t.run() method, Runtime.exit(), t.stop()

**Strong**
Intuitivní a přirozený pro programátora zvyklého na jednovláknové aplikace, všechny zápisy do paměti jsou viditelné v pořadí, v jakém proběhly. Přerovnání instrukci kompilátorem nemají vliv na sémantiku programu, pro vícevláknové aplikace je ale velmi nevýhodný, protože výrazně omezuje optimalizační možnosti procesoru nebo JVM a tím i výkon.

**Weak**
Využívá ho Java, od verze 5 je revidovaný. V případě jednovláknového programu se chová jako strong. Pro více vláken je situace složitější, např. se může stát, že zápis do proměnných A a B jedním threadem je v druhém threadu viděn v opačném pořadí nebo dokonce vůbec. Proto jsou k dispozici nástroje pro zajištění konzistence (bariéry, kritické sekce – synchronized bloky, atomické instrukce ).

**Synchronizace**
Každý objekt v Javě má se sebou svázaný monitor. Ten má tři funkce:

- Zámek – zajišťuje exkluzivitu přístupu.

- Wait condition – jde o čekání wait() (výstup z wait je opět read-barrier) a notifikace threadů notify() a notifyAll().

- Memory barrier. – při použití synchronized bloku se před něj do bytekódu vkládá instrukce monitorenter (chová se jako read-barrier, což znamená, že po této instrukci veškerá čtení z paměti musí číst aktuální hodnoty) a za něj monitorexit (chová se jako write-barrier, což znamená, že veškeré v kritické sekci do paměti zapsané hodnoty budou od teď viditelné ostatním threadům).

Synchronizace obvykle stačí pro většinu použití.

**Volatile**
Volatile proměnné mají několik vlastnosti:

- Kompilátor je nesmí držet v registru, musí je vždy propsat do hlavní paměti.
- Čtení volatile fieldu implikuje provedení read-barrier před tímto čtením, zápis do volatile fieldu implikuje provedení write-barrier po tomto zápisu.
- Kompilátor sice může přerovnávat přístupy k paměti, nicméně toto nesmí provést přes hranici přístupu k volatile fieldu (to je novinka od Javy 5).
- Čtení a zápis volatile fieldu je vždy atomický (i pro long a double).

Tvoří happens before hrany. Pokud zapíšu do volatile fieldu v jednom threadu a z druhého threadu přečtu již tuto novou hodnotu, pak vše co se stalo v prvním threadu před tímto zápisem se určitě stalo před čtením z druhého threadu a je tedy z tohoto druhého threadu po tomto čtení viditelné.

**Final proměnné**
Lze je číst bez synchronizace z jakéhokoliv threadu a je zaručen zisk správné hodnoty (před tím ovšem musí doběhnout konstruktor objektu s daným final fieldem, aniž by dal někomu referenci sám na sebe, tzn. reference na this nesmí opustit konstruktor. Toho nejsnáze dosáhneme vytvořením Factory a jednoduchým konstruktorem).

Immutable objekty mají mnoho dalších výhod. Mohu s nimi pracovat z libovolného threadu bez synchronizace, lze je znovu použít.

**MapReduce**
MapReduce (někdy též Map/Reduce nebo MapRed) je programovací model pro zpracování velkých množin dat pomocí paralelního zpracování, a současně knihovna v jazyce C++, která jej implementuje.

*Princip*

- Mějme cluster databázových nebo jiných serverů
- Jeden ze serverů (říkejme mu master, ale v některých modelech to může být libovolný uzel z clusteru) přijme požadavek Map/Reduce od klienta.
- Uzel master rozešle funkci Map (nebo více zřetězených funkcí) všem ostatním uzlům clusteru, ty provedou kód této funkce a vrátí masteru výsledky, které mohou být i duplicitní (více stejných výsledků z několika uzlů — to je žádoucí kvůli odolnosti proti výpadkům). Master může také zpracovat funkci Map a také to v některých implementacích dělá.
- V okamžiku, kdy má master dostatek výsledků z fáze Map od ostatních uzlů (a sám od sebe), nebo vyprší časový limit pro odpověď od těchto uzlů, provede master nad navrácenou množinou dat funkci Reduce. Fáze Reduce odstraní duplicitní data a provede operace, které je možné provést jen v případě, že máme kompletní množinu výsledků ze všech uzlů.
- Na konci fáze Reduce je možné navrátit výsledek klientovi, který si tuto operaci zadal.

**Phantom reference**
Je označení odkazu v jazyce Java, která je vytvořena pomocí třídy java.lang.ref.PhantomReference a dovoluje zabalit objekt, na který ukazuje. Na rozdíl od normální silné (strong) reference dovoluje, aby objekt, na který ukazuje referent mohl být, uvolněn garbage collectorem a poté smazán z paměti. Phantom reference je vhodná ke sledovnání, odkazovaného objektu jestli se stal nedostupný a neoživitelný, k provedení různých operací jako jsou nahrátí dalších velkých souborů, aby nedošlo k chybě OutOfMemoryError nebo k provedení závěrečných operací po životě objektu (uvolňování neuvolněných zdrojů) mnohem efektivněji a lépe než je tomu u funkce finalize(). Třída java.ref.PhantomReference je rozšíření jeho předka java.ref.Reference a tak z ní dědí i funkci get(), která by měla vrátit odkaz na odkazovaný objekt PhantomReference (referenta), ale jelikož je tento objekt už nedostupný a neoživitelný vrací null. Phantom reference je tedy použitelná jen s Reference Queue (Odkazovací frontou). Díky této frontě poznáme, že se referent dostal do Phantom dostupného stavu (není oživitelný a je určen k smazání z paměti), tím že bude do této fronty přidána pomocí garbage collectoru na referenta odkazující Phantom reference. K Phantom referenci je možné přiřadit vždy jen jeden odkaz a vždy jen jednu frontu a to vždy při jeho vytvoření přidáním odkazů na referenta a na frontu do jeho konstruktoru. Phantom reference se automaticky nečistí. Třída java.lang.ref.PhantomReference není finální, takže jí lze libovolně rozšiřovat.

**Návrhové vzory**
*Creational patterns (Vzory týkající se tvorby objektů)*
- **Abstract Factory** (Abstraktní továrna) – Definuje rozhraní pro vytváření rodin objektů, které jsou na sobě závislé nebo spolu nějak souvisí bez určení konkrétní třídy.
- **Factory Method** (Tovární metoda) – Definuje rozhraní pro vytváření objektu, které nechává potomky rozhodnout o tom, jaký objekt bude fakticky vytvořen. *Tovární metoda nechává třídy přenést vytváření na potomky.
- **Builder** (Stavitel) – Odděluje tvorbu komplexu objektů od jejich reprezentace tak, aby stejný proces tvorby mohl být použit i pro jiné reprezentace.
- **Lazy Initialization** (Odložená inicializace) – Odkládá vytváření objektu, počítání hodnoty nebo provádění nějakého procesu, až do okamžiku, kdy je ho poprvé potřeba.
- **Object pool** (Fond, lidově bazén) – Umožňuje vyhnout se drahému vytváření a uvolňování zdrojů recyklováním objektů, které už se nepoužívají.
- **Prototype** (Prototyp, Klon) – Specifikuje druh objektů, které se mají vytvořit použitím prototypového objektu. Nové objekty se vytváří kopírováním tohoto prototypového objektu.
- **Singleton** (Jedináček) – Potřebujete-li, aby měla třída maximálně jednu instanci.
- **Multiton** – Potřebujete-li, aby měla třída maximálně jednu instanci ke každému klíči.
- **Resource aquisition is initialization** – Zajišťuje, že zdroje budou správně uvolněny, tím, že je připoutá k životní době odpovídajících objektů.

*Structural Patterns (Vzory týkající se struktury programu)*
- **Adapter** (Adaptér) – Potřebujete-li, aby spolu pracovaly dvě třídy, které nemají kompatibilní rozhraní. Adaptér převádí rozhraní jedné třídy na rozhraní druhé třídy.
- **Bridge** (Most) – Oddělí abstrakci od implementace, tak aby se tyto dvě mohly libovolně lišit.
- **Composite** (Strom, Složenina) – Komponuje objekty do stromové struktury a umožňuje klientovi pracovat s jednotlivými i se složenými objekty stejným způsobem.
- **Decorator** (Dekorátor) – Použijeme jej v případě, že máme nějaké objekty, kterým potřebujeme přidávat další funkce za běhu. Nový objekt si zachovává původní rozhraní.
- **Facade** (Fasáda) – Nabízí jednotné rozhraní k sadě rozhraní v podsystému. Definuje rozhraní vyšší úrovně, které zjednodušuje použití podsystému.
- **Flyweight** (Muší váha) – Je vhodná pro použití v případě, že máte příliš mnoho malých objektů, které jsou si velmi podobné.
- **Proxy** – Nabízí náhradu nebo zástupný objekt za nějaký jiný pro kontrolu přístupu k danému objektu.

*Behavioral Patterns (Vzory týkající se chování)*
- **Chain of responsibility** (Zřetězení zodpovědnosti) – Vyhněte se závislosti odesílatele požadavku na příjemci tím, že dáte více než jednomu objektu možnost poradit si s tímto požadavkem. Zřetězte příjemce a přeposílejte požadavek po řetězu, dokud nebude vyřešen.
- **Command** (Příkaz) – Zapouzdřete požadavek jako objekt a tím umožněte parametrizovat klienty s různými požadavky, frontami nebo požadavky na log a podporujte operace, které jdou vzít zpět.

- **Interpreter** (Interpret) - Vytváří jazyk, což znamená definování gramatických pravidel a určení způsobu, jak vzniklý jazyk interpretovat.
- **Inversion of control** (Dependency Injection, Obrácení řízení) – Vztahuje se k poskytování externích závislostí softwarové komponentě.
- **Iterator** (Iterátor) – Nabízí způsob, jak přistupovat k elementům skupinového objektu postupně bez toho, abyste vystavovali vnitřní reprezentaci tohoto objektu.
- **Mediator** (Prostředník) – Umožňuje zajistit komunikaci mezi dvěma komponentami programu, aniž by byly v přímé interakci a tím musely přesně znát poskytované metody.
- **Memento** (Memento) – Bez porušování zapouzdření zachyťte a uložte do externího objektu interní stav objektu tak, aby ten objekt mohl být do tohoto stavu kdykoliv později vrácen.
- **Null Object** (Prázdný objekt) – Jde o objekt, který má fungovat jako základní stav objektu a který je v podstatě náhradou stavu null.
- **Observer** (Pozorovatel) – V případě, kdy je na jednom objektu závislých mnoho dalších objektů, poskytne vám tento vzor způsob, jak všem dát vědět, když se něco změní.
- **Servant** (Služebník) – Definuje společnou funkcionalitu pro skupinu tříd, které většinou nemají společnou rodičovskou třídu.
- **State** (Stav) – Umožňuje objektu měnit své chování, pokud se změní jeho vnitřní stav. Objekt se tváří, jako kdyby se stal instancí jiné třídy.
- **Strategy** (Strategie) – Zapouzdřuje nějaký druh algoritmů nebo objektů, které se mají měnit, tak aby byly pro klienta zaměnitelné.
- **Specification** (Specifikace) – Rekombinovatelná obchodní logika ve stylu booleanů.
- **Template method** (Šablonová metoda) – Definuje kostru toho, jak nějaký algoritmus funguje, s tím, že některé kroky nechává na potomcích. Umožňuje tak potomkům upravit určité kroky algoritmu bez toho, aby mohli měnit strukturu algoritmu.
- **Visitor** (Návštěvník) – Reprezentuje operaci, která by měla být provedena na elementech objektové struktury. Visitor vám umožní definovat nové operace beze změny tříd elementů na kterých pracuje.

*Concurrent patterns (Vzory řešící problémy vzniklé spouštěním programů ve vláknech a tudíž při souběžném řešení úlohy)*

- **Active object** – Tento návrhový vzor odděluje spouštění metod od provádění metod, přičemž spouštění metod může být ve svém vlastním vlákně. Cílem je přidat souběžnost použitím asynchronních volání metod a plánovače, který obsluhuje požadavky.
- **Event-based asynchronous** – Na událostech založený asynchronní návrhový vzor řešící problémy s Asynchronním vzorem, které nastávají ve vícevláknových programech.
- **Balking** – Tento vzor je softwarovým vzorem, který na objektu vykoná nějakou akci, pouze pokud je objekt v určitém stavu.
- **Double checked Locking** – Tento vzor je také znám jako „optimalizace zamykání s dvojnásobnou kontrolou“. Návrh je vytvořen tak, aby zredukoval zbytečné náklady na získávání zamčení tím, že nejdříve otestuje kritérium pro zamčení nezabezpečeným způsobem ('lock hint'). Pouze pokud uspěje, pak se opravdu zamkne.Tento vzor může být nebezpečný, pokud je implementován v některých kombinacích programovacích jazyků a hardwaru. Proto je někdy považován také za proti-vzor.
- **Guarded** – Jde o vzor obstarávající operace, které požadují uzamčení a navíc mají nějakou podmínku, která musí být splněna předtím, než může být operace provedena.
- **Monitor object** – Monitor je přístup k synchronizaci dvou nebo více počítačových úloh, které používají sdílené zdroje, zpravidla hardwarové zařízení nebo sadu proměnných.
- **Read write lock** – Tento vzor, také známý jako RWL, je vzor, který umožňuje souběžný přístup k objektu pro čtení, ale vyžaduje exkluzivní přístup pro zápis.
- **Scheduler** – Jde o souběžný vzor, který se používá pro explicitní kontrolu, kdy mohou vlákna vyvolávat jednovláknový kód.
- **Thread pool** – V bazénku vláken je vytvořen nějaký počet vláken pro řešení nějakého množství úloh, které jsou organizovány ve frontě. Zpravidla je výrazně více úloh než vláken.
- **Thread-specific storage - Thread-local storage** (TLS) je programovací metoda, která používá statickou nebo globální paměť lokálně pro vlákno.
- **Reactor** – Jde o vzor používaný pro vyřizování požadavků na službu, které jsou z jednoho nebo více vstupů doručovány správci služeb. Správce služeb rozdělí příchozí požadavky a přidělí je synchronně přidruženým vyřizovačům požadavků.