

Programovanie

1 Objektovo orientované programovanie – (zapúzdrenie, dedičnosť, polymorfizmus, trieda, modifikátory prístupu, konštruktory, abstraktné triedy, rozhrania), **vnorené triedy** (nested classes), **garbage collection**.

Zapuzdrenie:

Zapuzdrenie

Jedným z hlavných metodických princípov objektovo orientovaného programovania je **zapuzdrenie** (angl. *encapsulation*). Ide o „zabalenie“ dát a metód na manipuláciu s nimi do spoločného „puzdra“ – inštancie nejakej triedy.

- Kód z iných tried by mal s dátami „zabalenými“ v objekte manipulovať iba pomocou jeho metód na to určených.
- To sa obvykle zabezpečí tak, že sa modifikátor public priradí iba tým metódam, ktoré sú určené na použitie „zvonka“. Premenný a pomocný metódam sa priradí iný modifikátor, najčastejšie private.
- Verejné metódy tak tvoria akúsi „sadu nástrojov“, ktorú trieda poskytuje iným triedam na prácu s jej inštanciami. Napríklad trieda pre zásobník by mohla mať (okrem konštruktora) verejné metódy push, pop, isEmpty a peek, pričom jej premenné a prípadné pomocné metódy by boli súkromné.
- Vyhodou tohto prístupu je, že možno zmeniť vnútornú implementáciu triedy bez toho, aby to nejako ovplyvnilo ostatné triedy. Jediné, čo musí zostať zachované, je správanie verejných metód (čo zvyčajne ide zariadiť aj pri zmenenej implementácii zvyšku triedy). Napríklad v triede pre zásobník by sme mohli namiesto pola použiť spájaný zoznam a zvyšné triedy by to nijak neovplyvnilo.
- Zapuzdrenie tak umožňuje rozdeliť projekt na relatívne nezávislé logické celky s dobre definovaným rozhraním.

Dedenie:

Dedenie

Trieda môže byť podtriedou inej triedy. Napríklad trieda `Pes` môže byť podtriedou všeobecnejšej triedy `Zvieratko`: každý objekt, ktorý je inštanciou triedy `Pes` je potom súčasne aj inštanciou triedy `Zvieratko`. Tento vzťah medzi triedami vyjadrujeme klúčovým slovom `extends` v definícii triedy.

```
class Pes extends Zvieratko {  
    // ...  
}
```

Hovoríme tiež, že trieda `Pes` *dedi* od triedy `Zvieratko`, alebo že trieda `Pes` triedu `Zvieratko` *rozširuje*. V prípade vhodne zvolených prístupových modifikátorov (detaily neskôr) totiž inštancia triedy `Pes` zdedí metódy a premenné (nie konštruktory) definované v triede `Zvieratko` a tie sa potom správajú tak, ako keby boli priamo definované aj v triede `Pes`.

Dedenie umožňuje vynútiť sa nutnosti písť podobný kód viackrát. Namiesto implementácie podobných metód v niekoľkých triedach možno vytvoriť nadtriedu týchto tried a spoločne časti kódu presunúť tam.

Abstraktné triedy:

Abstraktné triedy a metódy

Aby sa metóda chovala v určitej skupine tried polymorfne, musí byť definovaná v ich spoločnej nadtriede. V tejto nadtriede však nemusí existovať jej zmysluplná implementácia.

- Uvažujme napríklad metódu `area()`, ktorá zráta plochu geometrického útvaru.
- Pre triedy `Rectangle`, `Circle`, resp. `Segment` je implementácia takejto metódy zrejmá. Zmysluplná implementácia v ich spoločnej nadtriede `Shape` by však bola prinajmenšom problematická.

Vzniknutú situáciu možno riešiť nasledovne:

- Metódu `area()` v triede `Shape`, ako aj triedu `Shape` samotnú, označíme za *abstraktnú* modifikátorom `abstract`.
- Abstraktná metóda pozostáva iba z hľavičky bez samotnej implementácie a je určená na prekrytie v podtriedach (musí ísť o nestickejúci metód).
- Abstraktná trieda je trieda, ktorá môže obsahovať abstraktné metódy. Zo zrejmých dôvodov z nej nemožno priamo tvoriť inštancie – napríklad v našom príklade by tieto inštancie „nevedeli, čo robia“ pri volaní metódy `area()` – ale za účelom volania z podtried môže obsahovať definície konštruktarov. Abstraktná trieda slúži iba na dedenie a ako taká nemôže byť typom žiadneho objektu. Stále však môže byť typom *referencie* na objekt.
- Podtriedy abstraktnej triedy, ktoré nie sú abstraktné, musia implementovať všetky abstraktné metódy svojho predka.

Dedenie a typy

- Typom objektu je trieda určená konštruktorm, ktorý bol objekt vytvorený – napríklad volanie konštruktora triedy `Circle` má za následok vytvorenie objektu typu `Circle`.
- Premenná, ktorej typom je trieda `T` však môže obsahovať aj referenciu na objekt, ktorého typom je podrieda triedy `T`. Napríklad premenná typu `Shape` tak môže obsahovať referenciu na objekt triedy `Shape` alebo jej ľubovoľnej podriedy. Vždy teda treba rozlišovať medzi typom referencie (premennej obsahujúcej referenciu na objekt) a samotným typom objektu.

```
Circle circle = new Circle(0, 0, 5);
Shape shape = circle;      // toto je korektné priradenie
// circle = shape;        // toto neskompileuje, keďže shape nemusí byť kružnica
circle = (Circle) shape;  // po pretypovaní to uz skomplilovat pojde; ak shape nie je instanciou Circle alebo null, vyhodi sa výnimka
```

- Istejší prístup je pri prirávaní premennej typu `Shape` do premennej typu `Circle` najprv overiť, či premenná typu `Shape` obsahuje referenciu na inštanciu triedy `Circle`. Na to slúži operátor `instanceof`. Platí pritom, že ak je objekt inštanciou nejakej triedy, je súčasne aj inštanciou ľubovoľnej jej nadriedy (samotný typ objektu je však daný iba najnižšou triedou v tomto usporiadani). Napríklad podmienka `shape instanceof Shape` je splnená kedykoľvek je splnená podmienka `shape instanceof Circle`. Pre ľubovoľnú triedu `Trieda` má výraz `null instanceof Trieda` vždy hodnotu `false` (a rovnako pre premenňu obsahujúcu `null`).

```
if (shape instanceof Circle) {
    circle = (Circle) shape;
}
```

- Keďže teda môžeme inštancie tried `Rectangle` alebo `Circle` považovať aj za inštancie ich spoločnej nadriedy `Shape`, môžeme rôzne typy útvarov spracúvať tým istým kódom. Napríklad nasledujúca metóda dostane pole útvarov a posunie každý z nich o daný vektor (`deltaX`, `deltaY`).

Dedenie a konštruktory

- Typickou úlohou konštruktora je správne nainicializovať objekt.
- Pri dedení si väčšinou každá trieda inicializuje „svoje“ premenné.
- Napríklad krajší spôsob realizácie konštruktov pre geometrické útvary je nasledovný: `Shape` inicializuje `x` a `y`, pričom napríklad `Circle` nechá inicializáciu `x` a `y` na `shape` a inicializuje už len `radius`.
- Prvý príkaz konštruktora môže pozostávať z volania konštruktora predka pomocou klúčového slova `super` (z angl. `superclass`, t. j. nadrieda).
- Ak nezavoláme konštruktor nadriedy ručne, automaticky sa zavola konštruktor bez parametrov, t. j. `super()`. To môže pri kompliovaní výstaviť v chybu v prípade, keď nadrieda nemá definovaný konštruktor bez parametrov (či už explicitne jeho implementáciou, alebo implicitne tým, že sa neuviedie implementácia žiadneho konštruktora nadriedy). Napríklad v horeuvedenom príklade je teda volanie konštruktora nadriedy nutnou podmienkou úspešného skomplilovania programu.
- Výnimkou je prípad, keď sa v rámci prvého príkazu volá iný konštruktor tej istej triedy pomocou `this(...)` – vtedy sa volanie konštruktora nadriedy nechá na práve zavolaný konštruktor.

Polymorfizmus:

Podrieda môže prekryť (angl. `override`) niektoré zdelené metódy, aby sa chovali inak ako v predkovi.

Uvažujme napríklad útvar `Segment` (úsečka), ktorý je zadaný dvoma koncovými bodmi a v metóde `move` treba posunúť oba. V triede `Segment` je teda potrebné metódu `move` prekryť. Metódu `move` z nadriedy `Shape` pritom možno zavolať ako `super.move()`, ale nemusí to byť v rámci prvého príkazu prekryvajúcej metódy `move` a metóda nadriedy sa tam prípadne ani nemusí zavolať vôbec (volanie metódy `super.move()` možno použiť aj v iných metódach a konštruktorkach triedy `Segment`). O prekrytie metódy z nadriedy ide samozrejme iba v prípade, že má prekryvajúca metóda rovnaký názov a rovnakú postupnosť typov parametrov (t. j. rovnakú signatúru) ako metóda nadriedy. V prípade rovnakého názvu metódy, ale rozdielnych typov parametrov ide len o preťaženie metódy, ako ho poznáme z minulej prednášky.

Návratový typ prekryvajúcej metódy sa musí buď zhodovať s návratovým typom prekryvanej metódy, alebo musí byť jeho „specializáciou“ (napr. môže ísť o podriedu triedy, ktorá slúži ako návratový typ prekryvanej metódy). Modifikátor prístupu prekryvajúcej metódy musí byť nastavený tak, aby bola táto metóda prístupná kedykoľvek je prístupná prekryvaná metóda (ak je teda napr. prekryvaná metóda verejná, musí byť verejná aj prekryvajúca metóda). Pokiaľ tiež vlastnosti prekryvajúcej metódy nie sú splnené, program neskompliluje.

Anotácia `@Override` je pri prekryvani metód nepovinná, ale odporúčaná. Ide o informáciu pre komplíta, ktorou sa vyjadruje snaha o prekrytie zdelenej metódy. Ak sa v predkovi nenachádza metóda s rovnakou signatúrou, komplíta vylási chybu. Tým sa dá predísť obzvlášť nepríjemným chybám, pri ktorých napríklad namiesto prekrytie metódy túto metódu neúmyselne preťažíme alebo napišeme metódu s úplne iným názvom (napr. `hashCode` namiesto `hashCode`).

S prekryvaním metód súvisí *polymorfizmus*, pod ktorým sa v programovaní (hlavne pri OOP) rozumie schopnosť metód chovať sa rôzne:

- S určitou formou polymorfizmu sme sa už stretli, keď sme mali viacero metód s rovnakým menom, avšak s rôznymi typmi parametrov (tzv. *preťažovanie metód*, angl. *overloading*).
- Pri dedení sa navýše môže metóda chovať rôzne v závislosti od triedy, ku ktorej táto metóda patrí.
- To, ktorá verzia metódy sa zavola, záleží od toho, akého typu je objekt, na akého typu je referencia naň.
- Taktôto ale funguje iba pri nestatických metódach (keďže statické metódy príslušia samotným triedam, rozdiel medzi typom referencie a typom inštancie tam nemožno využiť). Pri statických metódach preto nehovoríme o ich prekryvaní, ale o ich skryvaní; nemožno vtedy ani použiť anotáciu `@Override`.

Vo všeobecnosti sa pri volaní `o.f(par1, ..., parN)` pre objekt `o` typu `T` aplikuje nasledujúci princip:

- Ak má trieda `T` svoju implementáciu metódy `f` s vhodnými parametrami, vykoná sa táto verzia metódy.
- V opačnom prípade sa vhodná verzia metódy `f` hľadá v nadriede triedy `T`, v prípade neúspechu v nadriede nadriedy `T`, atď.

Polymorfizmus môže byť schovaný aj hlbšie – neprekrytá metóda z predka môže vo svojom tele volať prekryté metódy, čím sa jej správanie mení v závislosti od typu objektu.

Rozhrania:

Rozhrania

Rozhranie (angl. *interface*) je podobným konceptom ako abstraktá trieda. Existuje však medzi nimi niekoľko rozdielov, z ktorých najpodstatnejšie sú tieto:

- Rozhranie slúži predovšetkým ako zoznam abstraktívnych metód – klúčové slovo `abstract` tu netreba uvádzať. Pri triedach implementujúcich rozhranie je garantované, že na prácu s ním bude možné použiť metódy deklarované v rozhraní (odtiaľ aj termín „rozhranie“). Napríklad rozhranie pre zásobníky by mohlo deklarovať metódy ako `push`, `pop` a `isEmpty` a triedy pre zásobníky implementované pomocou polí resp. spájaných zoznamov by toto rozhranie mohli implementovať.
- Naopak implementované metódy musia byť v rozhraní označené klúčovým slovom `default`, prípadne musia byť statické (obidve tieto možnosti sa však typicky využívajú iba vo veľmi špeciálnych situáciach).
- Rozhranie nemôže definovať konštruktory, ani iné ako finálne premenné (t. j. konštanty).
- Kým od tried sa dedí pomocou klúčového slova `extends`, rozhrania sa *implementujú* pomocou klúčového slova `implements`. Rozdiel je predovšetkým v tom, že implementovať možno aj viaceré rozhrani. Jedno rozhranie môže navyše rozširovať iné (dopĺňať ho o ďalšie požadované metódy); v takom prípade používame klúčové slovo `extends`.
- Všetky položky v rozhraní sa chápú ako verejné (modifikátor `public` teda nie je potrebné explicitne uvádzať).
- Podobne ako abstraktá trieda, môže byť aj rozhranie typom referencie.
- Hoci nejde o prekrývanie v pravom slova zmysle, možno aj pri implementovaní metód z rozhraní použiť anotáciu `@Override`.

Modifikátory prístupu:

Prehľad niektorých modifikátorov tried, premenných a metód

Modifikátory prístupu:

- `public`: triedy, rozhrania a ich súčasti prístupné odvšadiaľ.
- (žiadnen modifikátor): viditeľnosť len v rámci balíka (`package`).
- `protected`: viditeľnosť v triede, jej podtriedach a v rámci balíka (len pre premenné, metódy a konštruktory; nedá sa aplikovať na triedu samotnú).
- `private`: viditeľnosť len v danej triede (len pre premenné, metódy a konštruktory; nedá sa aplikovať na triedu samotnú).

Iné modifikátory:

- `abstract`: neimplementovaná metóda alebo trieda s neimplementovanými metódami.
- `final`:
 - Ak je trieda `final`, nedá sa z nej ďalej dedit.
 - Ak je metóda `final`, nedá sa v podtriede prekryť.
 - Ak je premenná alebo parameter `final`, ide o „konštantu“, ktorú nemožno meniť (možno ju ale inicializovať aj za behu).
- `static`:
 - Statické premenné a metódy príslušia triede samotnej, nie jej inštanciam.
 - Statické triedy vo vnútri inej triedy nie sú viazané na jej konkrétnu inštanciu (viac neskôr).

Vnorené triedy:

Statické vnorené triedy

Trieda definovaná vo vnútri inej triedy sa v Java nazýva *vnorenou triedou* (angl. *nested class*). Takáto trieda môže alebo nemusí byť statická. V prípade, že statická je, správa sa táto trieda veľmi podobne ako ľubovoľná iná trieda – k statickej vnorenej triede StatickaVnorenaTrieda definovanej v triede VonkajsiaTrieda ale mimo triedy VonkajsiaTrieda pristupujeme cez VonkajsiaTrieda.StatickaVnorenaTrieda. Užitočnou novinkou je možnosť nastaviť vnoreným triedam prístupový modifikátor `private`, čím sa trieda stane viditeľnou iba z vonkajšej triedy, v ktorej je definovaná.

Staticosť vnorených tried znamená predovšetkým to, že k premenným a metódam inštancií triedy, v ktorej sú definované, pristupujú v podstate ako ktorakoľvek iná trieda (t. j. iba prostredníctvom tvorby inštancií vonkajšej triedy).

Vnútorné triedy

Nestatické vnorené triedy sa v Java nazývajú aj *vnútornými triedami* (angl. *inner classes*). Tieto triedy patria jednotlivým inštanciam vonkajšej triedy a majú tak prístup k ich premenným a metódam.

Garbage collection:

Celkovo ide o jazyk omnoho vyšej úrovne, než jazyk C: v oveľa väčšej miere sa tu abstrahuje od počítačovej architektúry. Java napríklad neumožňuje priamy prístup k pamäti počítača a o uvoľňovanie alokowanej pamäte sa stará JVM automaticky prostredníctvom mechanizmu tzv. *garbage collection*. Hoci teda jazyk nie je príliš vhodný na nízkoúrovňové programovanie, tvorba „bežných“ používateľských programov je tu podstatne pohodlnejšia, než napríklad v jazyku C. Okrem toho Java disponuje veľkou knižnicou štandardných tried (*Java Class Library*; skr. JCL), v ktorej je okrem iného implementované aj množstvo algoritmov a dátových štruktúr. Orientáciu v možnostiach ponúkaných touto knižnicou značne uľahčuje [dokumentácia knej](#).

2 Výnimky - vyhodnotenie výnimky, zachytenie a spracovanie výnimiek (try, catch, finally), vlastné triedy výnimiek, checked unchecked výnimky

Mechanizmus *výnimiek* (angl. *exceptions*) slúži v Java na spracovanie chýb a iných výnimočných udalostí, ktoré môžu počas vykonávania programu nastať. Dopolňajú sme v našich programoch takéto situácie viac-menej ignorovali – napríklad sme obvykle predpokladali, že vstup vždy spĺňa požadované podmienky, že súbor, z ktorého sa pokúšame čítať, vždy existuje, atď. Dôvodom bola predovšetkým prílišná prácnosť ošetrovania chýb pomocou podmienok a neprehľadnosť programov, ktoré takéto podmienky obsahujú.

Mechanizmus výnimiek v Java

Pod *výnimkou* (angl. *exception*) sa v Java rozumie inštancia špeciálnej triedy [Exception](#), prípadne jej nadriedy [Throwable](#), reprezentujúca nejakú výnimočnú udalosť. Trieda Exception má pritom množstvo podriedy reprezentujúcich rôzne typy chybových udalostí.

- Výnimka môže počas vykonávania nejakej metódy f vzniknúť buď tak, že ju vyhodí JVM (napríklad pri delení nulou alebo pri prístupe k neexistujúcemu prvku pola), alebo tak, že ju vyhodí sama táto metóda pomocou príkazu throw (detaily nižšie).
- Vzniknutá výnimka môže byť priamo odchytená a ošetrená v rámci danej metódy f. V opačnom prípade je vykonávanie metódy ukončené a výnimka je posunutá metóde g, ktorá metódu f volala (posunieme sa teda v zásobníku volaní metód o úroveň nižšie).
- V metóde g sa na celú situáciu dá pozerať tak, akoby výnimka vznikla pri vykonávaní príkazu, v rámci ktorého sa volala metóda f. Opäť teda môže byť výnimka buď odchytená a ošetrená, alebo rovnakým spôsobom predaná metóde, ktorá volala metódu g (aj v takom prípade hovoríme, že metóda g vyhodila výnimku, hoci reálne výnimka vznikla už v metóde f).
- Takto sa v zásobníku volaní metód pokračuje nižšie a nižšie, až kým je nájdená metóda, ktorá danú výnimku odchytí a spracuje.
- Ak sa takáto metóda na zásobníku volaní nenájde – čiže je výnimka vyhodená aj metódou main na jeho dne – typicky dôjde k ukončeniu vykonávania programu a k vypísaniu informácií o výnimke (vrátane zásobníka volaní metód) na štandardný chybový výstup.

Chytoranie a ošetrovanie výnimiek

Odchytanie a spracovanie výnimky možno v Java realizovať nasledujúcim spôsobom:

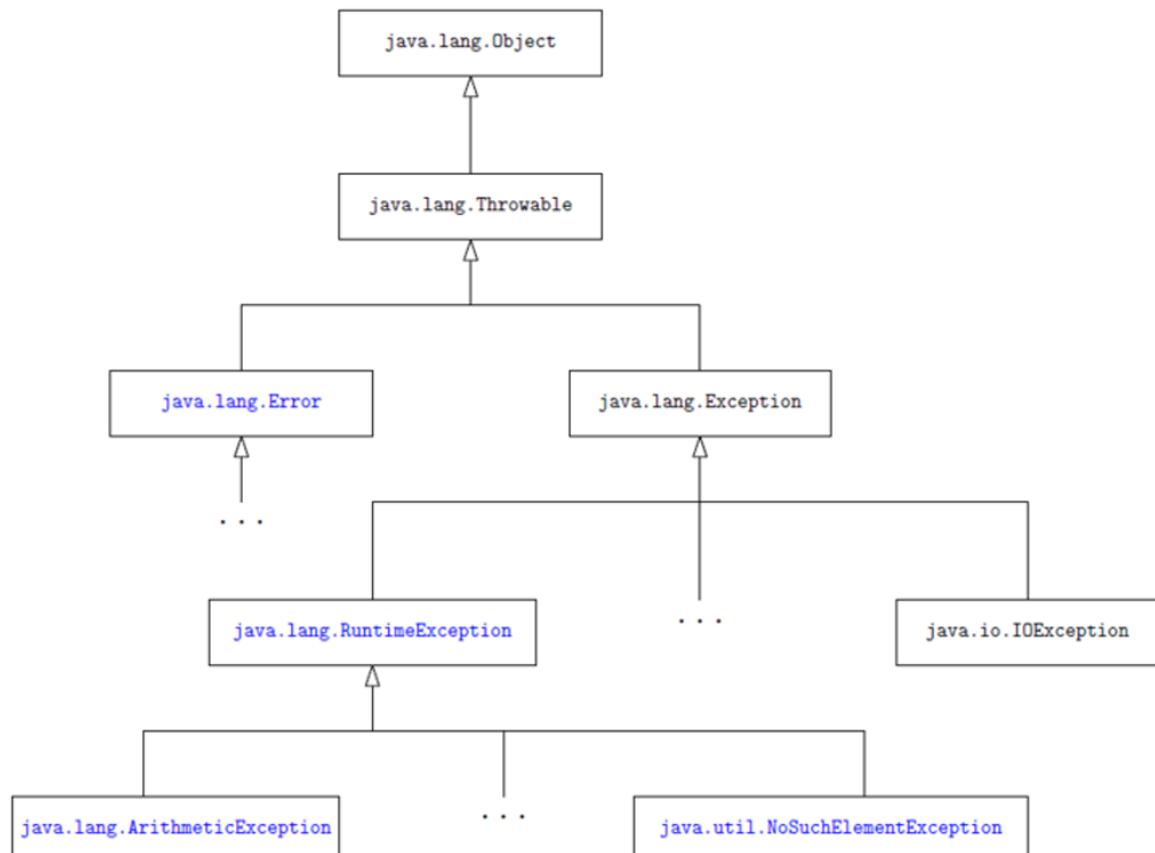
- Kód, v ktorom môže výnimka nastať, obalíme do bloku try.
- Kód spracúvajúci výnimku umiestnime do bezprostredne nasledujúceho bloku catch. Za samotným kľúčovým slovom catch nasleduje jeden argument reprezentujúci výnimku, ktorá sa má odchytiať. Napríklad blok catch (Exception e) odchytí ľubovoľnú výnimku, ktorá je inštanciou triedy Exception (alebo nejakej jej podtryedy).
- Kedykoľvek nastane v bloku try výnimka, okamžite sa vykonávanie tohto bloku ukončí. Ak je vzniknutá výnimka kompatibilná s argumentom bloku catch, pokračuje sa blokom catch a výnimka sa považuje za odchytanú. (Ak výnimka s týmto argumentom nie je kompatibilná, bude správanie rovnaké ako pri absencii bloku try.)

```
e.printStackTrace();
```

- V prípade vyhodenia výnimky nikdy nedôjde uzavoreniu vstupného súboru, pretože vykonávanie bloku try sa preruší ešte predtým, než príde na rad príslušný príkaz. O niečo vhodnejšou alternatívou by bolo presunutie príkazu zatvárajúceho scanner až za koniec príslušného bloku catch; samozrejme s ošetrením prípadu, keď scanner == null. Ale aj vtedy by sa mohlo stať, že sa tento príkaz nevykoná kvôli nejakej výnimke, ktorá nebola odchytaná (napríklad inštancia typu Throwable, alebo výnimka vyhodená v bloku catch). Riešením je pridať za bloky catch blok finally, ktorý sa vykoná bez ohľadu na to, či nastala výnimka a či sa nám ju podarilo odchytiať (dokonca sa vykoná aj v prípade, že sa v try bloku úspešne vykonal príkaz return).

Hierarchia výnimiek

V Java existuje množstvo tried pre rôzne druhy výnimiek. Malá časť hierarchie týchto tried je znázornená na nasledujúcom obrázku.



Všetky výnimky v Java dedia od triedy [Throwable](#). Tá má dve podtrydy:

- [Exception](#), od ktorej dedí väčšina „bežných“ výnimiek.
- [Error](#), od ktorej dedia triedy reprezentujúce závažné, v rámci aplikácie ľažko predpovedateľné systémové chyby (napríklad nedostatok pamäte). Jednou z najznámejších podtryd triedy Error je [StackOverflowError](#).

Podtrydy triedy Exception možno ďalej rozdeliť na dve základné kategórie:

- [RuntimeException](#) a jej podtrydy. Výnimky tohto typu obvykle reprezentujú rozličné programátorské chyby, ako napríklad prístup k neexistujúcemu prvku poľa ([ArrayIndexOutOfBoundsException](#)), delenie nulou ([ArithmeticException](#)), použitie kódu „očakávajúceho“ objekt na referenciu null ([NullPointerException](#)), atď. Do tejto kategórie patria aj výnimky typu [NoSuchElementException](#), hoci tie sú v príklade vyššie možno trochu zneužili na ošetrenie zlého formátu vstupného súboru. Vo všeobecnosti platí, že výnimky tohto typu by budú nemali vôbec nastáť (programátorské chyby, ktoré je nutné odladiť), alebo by mali byť ošetrené priamo v metóde, v ktorej vzniknú (ako napríklad [NoSuchElementException](#) v našom príklade vyššie).
- Zvyšné podtrydy triedy Exception. Tie často reprezentujú neočakávateľné udalosti, ktorým sa na rozdiel od výnimiek typu [RuntimeException](#) nedá úplne vyhnúť (napríklad [FileNotFoundException](#) a jej nadtryda [IOException](#)). Dobre napísaný program by sa mal vedieť z výnimiek tohto typu zotaviť (nie je napríklad dobré ukončiť program vždy, keď sa mu nepodarí otvoriť súbor).

S týmto rozdenením podľa druhov chýb reprezentovaných jednotlivými výnimkami súvisí aj nasledujúca nutnosť:

- Výnimka ľubovoľného typu okrem [RuntimeException](#), [Error](#) a ich podtryd musí byť v metóde, v ktorej môže vzniknúť, vždy buď odchytená, alebo v opačnom prípade ich musí táto metóda deklarovať vo svojej hlavičke ako neodchytené. Napríklad:
- Popri vykonaní príkazu `return` je totiž vyhodenie výnimky d'ľším možným spôsobom ukončenia vykonávania volanej metódy. Preto musí byť táto možnosť v hlavičke metódy explicitne špecifikovaná rovnako ako napríklad návratový typ.
- Pri inštanciach triedy [RuntimeException](#) a jej podtryd sa od tejto požiadavky upúšťa, pretože – ako bolo spomenuté vyššie – ide väčšinou o programátorské chyby, ktoré je nutné odladiť, alebo sú tieto výnimky odchytené priamo v metóde, v ktorej vzniknú. Pri inštanciach triedy [Error](#) a jej podtryd zas často ide o systémové chyby, zotavenie z ktorých principálne nie je možné. Často je teda najlepším riešením ukončiť samotný program.
- Hoci môže hŕdzané výnimky prostredníctvom `throws` deklarovať aj metóda `main` (s príkladmi sme sa už stretli), pri reálnych programoch sa to nepovažuje za dobrú prax – všetky výnimky, ktoré sú inštanciami [Exception](#), by mali byť ošetrené.

Checked unchecked

Z hľadiska použitia tak môžeme výnimky – presnejšie inštancie triedy [Throwable](#) – rozdeliť na dve základné kategórie:

- **Nestrážené výnimky** (angl. *unchecked exceptions*), ktorými sú inštancie tried [RuntimeException](#) a [Error](#) resp. ich podtryd. Vyhadzovanie týchto výnimiek nie je potrebné explicitne deklarovať (kompilátor nesleduje miesta ich vzniku).
- **Strážené výnimky** (angl. *checked exceptions*), ktorými sú inštancie triedy [Throwable](#) resp. jej podtryd, ktoré nie sú inštanciami (podtryd) triedy [RuntimeException](#) ani [Error](#). Vyhadzovanie týchto výnimiek je potrebné deklarovať explicitne (kompilátor si o ich vyhadzovanie udržiava prehľad).

Prehľad komplilátora o vyhadzovaní strážených výnimiek sa prejavuje napríklad aj nemožnosťou skompilovať program obsahujúci blok `catch` taký, že:

- Typ výnimky odchytávanej v tomto bloku `catch` zahŕňa *iba* strážené výnimky (t. j. ide o *vlastnú* podtrydu triedy [Exception](#) a súčasne nejde o triedu [RuntimeException](#) alebo jej podtrydu).
- V príslušnom bloku `try` sa výnimka tohto typu nikdy nevyhadzuje.

3 Vlákna (threads) – stav vlákna (new, runnable, blocked, waiting, timed waiting, terminated), životný cyklus vlákna (vytvorenie, spustenie, zastavenie) plánovanie vlákien (fixed priority scheduling, yield, time slicing), synchronizácia (kritické úseky, wait, notify, explicitné zámky, podmienkové premenné)

Vlákna

Moorov zákon, teda pozorovanie, že každý rok a pol sa výpočtová sila integrovaných obvodov zdvojnásobí, pričom cena ostáva konštantná, v posledných rokoch stráca silu - výroba integrovaných obvodov naráža na technické limity. Riešením, ktoré dovoľuje ďalej zvýšovať výkonnosť je **parallelizmus**. Podstatnou otázkou je, ako parallelizovať.

Možnosti sú viaceré:

- „rozsekáť“ dátu na niekoľko kusov, každý z nich spracovať zvlášť a na záver ich „zlepíť“ (čo však očividne nie je vhodné pre všetky typy dát a algoritmov)
- parallelizovať priamo algoritmus (tiež nevhodné pre niektoré algoritmy)
- „spekulatívne vykonávanie dopredu“ - predrátat si na dátach všetky operácie, ktoré môže používateľ vykonať (napríklad načítať všetky webové stránky, na ktoré boli linky v aktuálnej zobrazenej webovej stránke) a v momente, keď dá príkaz na ich vykonanie (keď užívateľ klikne na konkrétny link) už len zobraziť vopred vykonanú operáciu

Parallelizácia programov je uskutočniteľná pomocou **vlákien**, tiež nazývaných odľahčenými procesmi.

Rozdiel medzi vláknom a procesom je v tom, že správa zdrojov procesov je prevádzkovaná operačným systémom - ten dáva pozor, aby si procesy navzájom nesiahali do zdrojov, no pri vláknoch táto ochrana nie je zabezpečená: to, aby vlákna manipulovali len s im pridelenými zdrojmi, musí zabezpečiť programátor. Jednému procesu môže patriť viaceré vlákien. Výhodou vlákien oproti procesom je väčšia rýchlosť (kontrola procesov systémom a ich prepínanie zaberá istý čas).

Procesor dokáže naraz vykonávať len jedno vlákno/proces - aby vznikol dojem súčasného behu viacerých procesov/vlákien, musia sa rýchlo striedať. Striedanie vlákien môže zabezpečovať virtuálny stroj, no môže ho podporovať aj operačný systém. Striedanie vlákien je nepredvídateľné.

Existujú dva spôsoby, ako implementovať vlákna v Java: dedením od triedy Thread alebo implementáciou rozhrania Runnable.

Implementácia rozhrania Runnable sa používa, ak vytváraný objekt musí byť potomkom inej triedy ako Thread, prípadne ak by jediným dôvodom na dedenie od Thread bolo preddefinovanie metódy run().

Prieklady tejto stránky... | Slovenčina (SK)

Životný cyklus vlákna

Nové vlákno vznikne použitím "new T", kde T je trieda dediča od Thread (pričom samotná trieda Thread). Po vytvorení je vlákno v stave **NEW**. Na inštancii triedy T sa dá následne zavolať len metóda start(), hocičo iné (s výnimkou metódy setDaemon(), o ktorej bude reč nižšie) vyhodi výnimku. Zavolením metódy start() vlákno prejde zo stavu NEW do stavu pripravené - **RUNNABLE**: pridelia sa mu systémové zdroje a naplánuje sa pre beh. Vlákno v stave RUNNABLE buď beží, alebo čaká na priradenie procesoru.

Zo stavu RUNNABLE sa môže dostať do troch rôznych stavov, ktoré sa súhrnnne označujú ako **nepripravené**, v ktorých sa vláknu neprideľuje procesor, aj keby bol k dispozícii (čo je žiadane, pretože vlákno nemá čo robiť, a zbytočne by plynalo systémovými prostriedkami počas aktívneho čakania):

- **WAITING** - vlákno sa doň dostane zavolením metódy wait(). Do stavu RUNNABLE ho vráti zavolenie metódy notifyAll() iným vláknom, resp. zavolenie metódy notify() iným vláknom a následne vybraťe správcom procesov ako to vlákno spomedzi čakajúcich, ktoré bude upovedomené
- **TIMED_WAITING** - do tohto stavu sa vlákno dostane na M milisekúnd zavolením metódy sleep(M). Po uplynutí daného času sa samo „zobudí“ - prejde do stavu RUNNABLE
- **BLOCKED** - vlákno môže byť blokované vstupno-výstupnými operáciami alebo zámkom (Lock - bude spomínaný ďalej). Z tohto stavu späť do RUNNABLE ho „zobudí“ operačný systém

Vlákno skončí (prejde do stavu **TERMINATED**), keď skončí jeho metóda run(). Celý program skončí, keď skončia všetky jeho vlákna, ktoré nie sú démonmi (ako démoni sú označované vlákna, ktoré nebránia Java Virtual Machine v ukončení programu, aj keď bežia, teda vlákna bežiace „na pozadí“. Typickým príkladom je garbage collection. To, či je vlákno démonom, sa dedí (vlákno bude mať prednastavené, že je démonom, ak bolo démonom aj vlákno, čo ho vytvorilo) a dá sa to zmeniť metódou setDaemon() - pozor, len predtým, ako bola volaná metóda start()).

Na to, v akom stave vlákno je, sa dá spýtať metódou getState(). Tá vracia jednu zo šiestich konštant spomínaných vyššie (NEW, RUNNABLE, WAITING, TIMED_WAITING, BLOCKED, TERMINATED). Ďalšia metóda, ktorá sa môže zísť v súvislosti so stavom vlákna je isAlive() - vracia true, ak vlákno už začalo (bola na ňom zavolená metóda start()) a zároveň neskončilo jeho metóda run(), false v opačnom prípade.

Priorita a plánovanie vlákien

Priorita vlákna je celé číslo od 1 po 10 (v triede Thread sú definované konštanty MIN_PRIORITY ako 1, NORM_PRIORITY ako 5 a MAX_PRIORITY ako 10, teda väčšie číslo znamená väčšiu prioritu). Priorita sa dedí (podobne ako to, či je vlácko démonom). Zmeniť prioritu vlákna na P je možné jeho metódou setPriority(P), zistiť jeho prioritu metódou getPriority().

Plánovanie vlákien: keďže na väčšine počítačov naraz beží viac vlákien, ako má počítač procesorov, vlákna sa musia vo využívaní procesora striedať (predstavte si počítač, na ktorom by sa nestriedali). Nie len, že by sa nedalo pracovať s viacerými aplikáciami súčasne, navyše ani v rámci jednej aplikácie by nebolo možné vykonávať niekoľko vecí súčasne - vykonanie výpočtovo zložitejšej operácie (napr. doostrenie veľkej fotografie) by na istý čas odstavilo užívateľský interface aplikácie, tá by „zamrzla“. Preto má zmysel robiť aj aplikácie pre počítače s jedným jadrom viacvláknové.

To, kedy bude ktorému vláknu pridelený procesor určuje algoritmus nazývaný **plánovač**. Java podporuje jednoduchý deterministický plánovač známy ako fixed-priority scheduling. V ľubovoľnom čase, keď je k behu pripravených viaceré vlákien, plánovač spomedzi nich vyberie vlácko s najvyššou prioritou. Ak má niekoľko (všetky) vlákna rovnakú najvyššiu prioritu, vyberie ľubovoľné z nich. Vybrané vlácko beží, kým nenastane jedna z nasledujúcich situácií:

- skončí sa jeho metóda run()
- bežiace vlácko zavolá metódu yield(), čím sa dobrovoľne vzdá procesora a dá možnosť bežať inému vláknu (plánovač si vyberá ktorému)
- medzi pripravenými vláknami sa objaví vlácko s vyššou prioritou než má práve bežiace vlácko

Na systémoch podporujúcich **time-slicing** môže byť beh vlákna prerušený aj vypršením času, ktoré je vláknu pridelené - Java však time-slicing nešpecifikuje, a preto ho systém nemusi implementovať.

Napriek tomuto nie je zaručené, že v ľubovoľnom okamihu beží vlácko s najväčšou prioritou. Plánovač môže pre beh vybrať aj vlácko s nižšou prioritou, aby tým zabránil jeho „vyhladovaniu“ (čo je situácia, keď vlácko nie je schopné získať prístup k zdieľanému prostriedku a nie je schopné tento stav zmeniť).

Z týchto dôvodov nie je možné sa spoliehať na nejaké konkrétné poradie striedania vlákien: priorita môže byť použitá len na zvýšenie efektivity vykonávania vlákien, nie je možné na nej postaviť korektnosť algoritmu.

Sebeckým vláknom (selfish thread) sa označuje vlácko, ktorého beh je vždy zastavený až skončením jeho metódy run() alebo prerušením vláknom s väčšou prioritou. Takéto správanie je sebecké voči vláknam s rovnakou prioritou, ktoré nemajú šancu bežať, kým sebecké vlácko neskončí. Aby vlácko nebolo sebecké, malo by z času na čas zavolať svoju metódu yield() a tým umožniť beh iným vláknam s rovnakou prioritou. Toto samozrejme nie je nutné, ak systém implementuje time-slicing (tedy systém automaticky preruší vlácko vtedy, keď mu ubehne čas pridelený na beh), avšak keďže time-slicing nie je Javou vyžadovný, tak aplikácie, ktoré sa naň spoliehajú, nie sú prenositeľné.

Synchronizácia vlákien

Dosiaľ sme sa zaoberali len vláknami, ktoré sa nezaujímali o stav a aktivity iných vlákien. Každé vlácko taktiež obsahovalo všetky dátá a metódy potrebné pre svoj beh. Išlo teda o **nezávislé asynchónne vlákna**, ktoré sa o seba nijakým spôsobom nestarali.

Problémy môžu nastat, keď vlákna potrebujú pristupovať k zdieľaným prostriedkom, alebo chcú komunikovať.

Obsah
<ul style="list-style-type: none"> ▪ Kritické úseky <ul style="list-style-type: none"> • Ako to funguje? • wait(), notify() a notifyAll() • Explicitné zámky a podmienkové premenné

1. **priklad:** Dve vlákna chcú naraz vypisovať do súboru, vlácko A vypisuje reťazec „aa“, vlácko B vypisuje reťazec „bb“. V prípade, že sa nesynchronizujú, nemusí byť výsledkom „aabb“, resp. „bbaa“, ale môže to byť hocičo zmiešané, napr. „abba“ (vlácko A bol pridelený procesor, stihlo do súboru zapísať znak „a“, vzápäť mu vypršal čas, procesor bol pridelený vlácko B, to stihlo zapísať „bb“, následne bol procesor opäť pridelený vlácko A, ktoré dopísal druhé „a“). Ide o typický príklad **race conditions**, teda časovej závislosti - výsledok programu závisí od toho, ako budú naplánované vlácko na beh.
2. **priklad:** Vlácko komunikujú posielaním si dát - jedno vlácko je **producentom**, druhé **konzumentom**. Producent niečo vyrába (dátá, signály) a posielá konzumentovi, ten to prijima.

Kritické úseky

Ako **kritický úsek** sa označuje časť kódu, ktorá pristupuje k zdroju (dátu, štandardnému výstupu, ...), ku ktorému môže mať súčasne prístup aj iné vlácko. Je potrebné zabezpečiť, aby k zdieľanému zdroju malo naraz prístup len jedno vlácko - čo Java rieši cez označenie kritických úsekov kľúčovým slovom synchronized. To sa vždy viaže na nejaký objekt:

V druhom prípade sa synchronized viaže na objekt, ktorému patrí metóda, čiže this.

Je žiadúce písat **kritické úseky čo najkratšie**, a do kučeravých závieriek po synchronized uzatvárať len príkazy, ktoré tam nutne musia byť, nech sa zbytočne príkazmi netýkajúcimi sa zdieľaného zdroja neblokuje prístup k danému zdroju iným vláckom. Kľúčové slovo **synchronized sa nededí**: ak je metóda M v triede T synchronizovaná, tak predefinovaná metóda M v potomkovi triedy T synchronizovaná byť nemusí (samořejme, môže). Synchronizácia niečo stojí - **volanie synchronizovanej metódy je pomalšie ako volanie nesynchronizovanej**. Preto synchronizáciu treba používať len v metódach obsahujúcich kritické úseky.

Ako to funguje?

Každému objektu Java pridelí **zámek**, ktorý uzamkne, keď nejaké vlákno vstúpi do synchronizovanej časti kódu viažucemu sa k danému objektu. Keď je zámok uzamknutý, žiadne iné vlákno nemôže zavolať vykonávať žiadenský synchronizovaný kód viažuci sa na ten istý objekt. Pokial' sa o to pokúsi, bude blokované až kým pôvodné vlákno neopustí synchronizovanú časť kódu, čím sa zámok odomknie. Vlákno, ktoré objekt uzamklo, však môže volať synchronizované metódy tohto objektu a vykonávať iný synchronizovaný kód uzamknutý vzhľadom na tento objekt (aj rekurzívne).

Zamykanie a odomykanie Java vykonáva automaticky, ako **atomickú** (nedeliteľnú) operáciu. Vďaka tomu sa vyhneme problémom, ktoré by mohli vzniknúť ako dôsledok race conditions.

Vráťme sa teraz k príkladu producenta a konzumenta - ich činnosti musia byť synchronizované dvoma spôsobmi:

- Pristupovať k objektu Storage môže naraz iba jedno vlákno. Rieši sa uzamykaním objektu (synchronizované metódy).
- Producent musí oznámiť konzumentovi, že sú pripravené nové dátá a konzument zas musí oznámiť producentovi, že už dátá prečítal a v Storage je voľné miesto pre ďalšie dátá. Rieši sa metódami `wait()` a `notify()`.

`wait()`, `notify()` a `notifyAll()`

Treba si uvedomiť, že `wait()` znamená „zaspi a čakaj, kým ťa niečo nezobudí“ a nie „zaspi a čakaj, kým ťa nezobudí práve to, na čo čakáš“. Preto treba `wait()` volať v cykle: `while (!podmienka) wait();`

Podobne `notify()` znamená „upovedom niekoho o niečom“ - náhodne vyberie jedno z čakajúcich vlákien a to zobudí. Preto sa táto metóda používa len pri optimalizácii, keď všetky vlákna čakajú na tú istú podmienku a je jedno, ktoré sa aktivuje. (`notifyAll()` analogicky znamená „upovedom všetkých o niečom“.)

Keď vlákno čaká na príkaze `wait()`, odomknie sa zámok ku ktorému `wait()` patrí, avšak iné objekty uzamknuté týmto vláknom stále ostávajú uzamknuté.

`wait()`, `notify()` a `notifyAll()` sú použiteľné len v kritických úsekoch, mimo nich vyhadzujú výnimku.

`notify()` a `notifyAll()` **majú len aktuálny účinok**, ak nie je žiadne vlákno, ktoré by sa mohlo zobudit', účinok metódy `notify()` (resp. `notifyAll()`) sa nedoloží do budúcnosti, ale zanikne.

[Upraviť](#)

Explicitné zámky a podmienkové premenné

Ďalšia možnosť, ako získať výlučný prístup k časti kódu, je pomocou **explicitných zámkov**. Ich výhodou oproti synchronizovaným metódam je, že sa môžu zamknúť a odomknúť v rôznych častiach kódu (napríklad to nemusí byť v jednej metóde). Na vytvorenie explicitného zámku treba inšancovať triedu, ktorá implementuje rozhranie `Lock`, napr. `ReentrantLock`. Zámok sa potom zamkne zavolením metódy `lock()` a odomknie metódou `unlock()`. Keďže zámok sa automaticky neodomká pri ukončení metódy, je vhodné uvažovať `lock()` a `unlock()` do bloku `try/finally`, aby zámok neostal zamknutý pri vzniku výnimky, a neblokoval tým ostatné vlákna.

Pokiaľ je potrebné čakať na explicitnom zámku, vytvorí sa podmienková premenná pomocou metódy `Lock.newCondition()`. Podmienková premenná je inštancia triedy implementujúcej rozhranie `Condition`. Rozhranie poskytuje metódy `await()` (čakanie na podmienku, používa sa na mieste, kde by ste pri použití `synchronized` použili `wait()`) a `signal()` (zobudenie vlákien, ktoré čakali pomocou `await()` na danej podmienkovej premennej - podobne `notifyAll()`).

Pozor, aj pri použití explicitných zámkov (rovnako ako pri použití `synchronized`) **sa uzamykajú objekty, a nie kód**. Teda ten istý kód v bloku `synchronized` sa teda môže vykonávať pre rôzne objekty (inštancie).

Na jednom objekte môže čakať viacero vlákien, pričom ich čakanie mohlo vzniknúť na rôznych príkazoch `wait()`.

Každý objekt má **dve samostatné fronty**: v jednej sú **vlákna čakajúce na vstup do s týmto objektom synchronizovaného bloku** kód, v druhej **vlákna čakajúce na príkaze `wait()`**. Žiadne vlákno nikdy nie je v oboch frontách súčasne (keďže sa nemá ako dostať do oboch front súčasne). Čakajúce vlákna musí aktivovať iné bežlace vlákno. Metóda `notifyAll()` aktivuje všetky čakajúce vlákna, tj. presunie všetky vlákna z fronty tých, ktoré čakajú na metóde `wait()` do fronty tých, ktoré čakajú na vstup do synchronizovaného bloku.

4 Generics – formálne typové parametre, parametrizovaný typ, wildcards, ohraničené wildcards, generics methods)

[Upraviť](#)

Definícia jednoduchých Generics

Ako priklad si pozrime definiciu generických interface-ov List a Iterator (nájdeme ich v balíku java.util)

```
interface List<E> {
    void add(E x);
    Iterator<E> iterator();
}

interface Iterator<E> {
    E next();
    boolean hasNext();
}
```

Nové sú tu pre nás veci medzi < > zátvorkami. Sú to deklarácie **formálnych typových parametrov**. Typové parametre sa potom môžu v rámci rozhrania používať ako obyčajné dátové typy(napr. add(E x) definuje, že metóda add bude mať na vstupe jeden prvok x typu E). Pri konkrétnom použití (`List<Integer>`) sa vytvorí tzv. **parametrizovaný typ**. Všetky výskytu formálnych typových parametrov (E) sa nahradia aktuálnym typovým argumentom (Integer).

V skutočnosti sa nevytvárajú viaceré kopie kódu. Všeobecné typy sa komplilujú iba raz prevždy do jednej triedy ako ostatné triedy a rozhrania. Na pomenovávanie formálnych typových argumentov sa používajú jednopísmeňkové názvy. Na rozlišenie od ostatných názvov tried a rozhraní sa používajú v názve formálnych typových argumentov iba veľké písmená. Často sa používa E ako element.

Pozrime sa na nasledujúci kus kódu. Pôjde skompilovať?

```
List<String> stringList = new ArrayList<String>();
List<Object> objectList = stringList;
```

Na prvý pohľad by sa zdalo, že áno. Vedľ predsa String je potomok triedy Object, tak prečo by sme do Listu Object-ov nemohli priradiť List String-ov. Ak by však toto bolo možné, problém by nastal, ak by sme sa pokúsili o nasledovné:

```
objectList.add(new Object());
String s = stringList.get(0);
```

Prvý riadok je v poriadku (keďže objectList je typu <object>, môžeme doň pridať nejaký nový Object). V nasledujúcom riadku, však vyberáme tento prvok, ale už za pomocí stringList- ktorý je definovaný ako <String> a teda je v poriadku, že prvok, čo vyberáme chceme uložiť do premennej typu String. To sa nám, ale samozrejme nemôže podať, keďže daný prvok je typu Object.
Z tohto dôvodu teda ani úvodný program skompilovať nepôjde a dostaneme chybovú hlášku: *Type mismatch: cannot convert from List<String> to List<Object>*

Wildcards

Zamyslime sa nad metódou, ktorá vypíše všetky prvky kolekcie. Metóda **printCollection1** je implementáciou takejto metódy v jazyku Java, pred verziou 1.5. Metóda **printCollection2** je prvým pokusom o reimplementáciu tejto metódy použitím konštrukcií z jazyka Java 1.5.

```
void printCollection1(Collection c) {
    for (Iterator i = c.iterator(); i.hasNext();)
        System.out.println(i.next());
}

void printCollection2(Collection<Object> c) {
    for (Object e : c)
        System.out.println(e);
}
```

Vieme, že Collection<Object> nie je rodičom všetkých kolekcí. Nedá sa teda spoľahnúť na polymorfizmus a očakávať, že namiesto Collection<Object> vložím napríklad Collection<String>. Metóda **printCollection2** preto ako svoj argument môže prijať iba Collection<Object>, čím sa stáva podstatne menej univerzálnou, ako pôvodná implementácia. Pre špecifikovanie kolekcie neznaných objektov sa používa wildcard typ, v našom prípade Collection<?>. Môžeme teda ešte raz reimplementovať metódu **printCollection1** pomocou generics, tentokrát bez straty jej univerzálnosti:

```
void printCollection3(Collection<?> c) {
    for (Object e : c)
        System.out.println(e);
}
```

Prvky priradujeme do premennej typu **Object**. Využívame pritom polymorfizmus a to, že v jazyku Java je každý objekt potomkom triedy **Object**. Na druhej strane do kolekcie typu **Collection<?>** nie je možné nič pridať (okrem null).

```
Collection<?> c = new ArrayList<String>();
c.add(new Object());
// The method add(?) in the type collection<?> is not applicable for the arguments (object)
```

Metóda **add** má argument typu **E**, kde **E** je typ prvkov kolekcie. V tomto prípade znak ? znamená neznaný typ. Aby sme mohli dať nejaký objekt ako argument tejto metódy, musel by byť potomkom ľubovoľného objektu a taký objekt neexistuje.

Ohraničené wildcards

Majme jednoduchú aplikáciu pre kreslenie, ktorá dokáže kresliť tvary, ako napríklad kruhy alebo štvorce. Pre reprezentovanie týchto tvarov môžeme vytvoriť nasledovnú hierarchiu tried (**Shape**, **Circle**, **Rectangle**). Majme tiež nejakú kresiacu plochu **Canvas**, na ktorú budeme tieto tvary vykresľovať pomocou metódy **draw**. Väčšina kresieb pozostáva z viacerých tvarov a preto je dobré mať metódu, ako je **drawAll1**, ktorá vykreslí celý zoznam tvarov.

Generické metódy

Podobne ako generické triedy možno tvoriť aj jednotlivé generické metódy, pri ktorých sa typové parametre píšu *pred návratovým typom*. Tieto parametre sú „viditeľné“ iba v rámci danej metódy. Pri volaní metódy možno za tieto parametre buď explicitne dosadiť konkrétné triedy, alebo sa o to postará automatický mechanizmus inferencie typov.

5 Composit, Strategy:

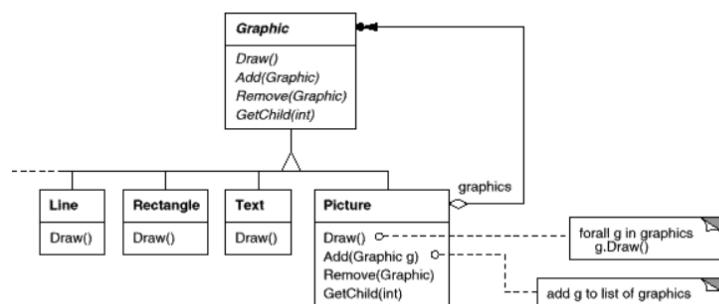
Composite pattern

Účel

Skladanie (komponovanie) objektov do stromovej štruktúry, pričom sa pracuje rovnako s jednoduchými a zloženými objektami.

Motívacia

Pri práci so zloženými objektami sa častokrát používajú iné metódy ako pre prácu so základnými objektami (primitívmi) a aj jednotlivé primitívy majú rozdielne metódy. Potom však vznikajú problémy pri používaní týchto objektov. Tieto problémy rieši **Composite** pattern.



Graphic je abstraktná trieda pre primitívy ale aj ich kontajnery.

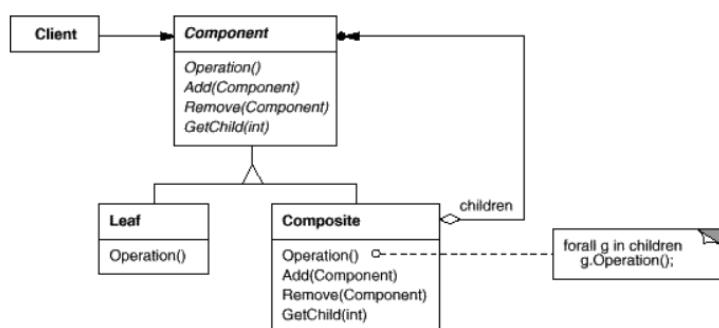
Použiteľnosť'

Návrhový vzor Composite sa používa, keď chceme:

- zobraziť hierarchie objektov
- pracovať so všetkými objektami v štruktúre jednotne a ignorovať rozdiely medzi primitívnymi objektami a kompozíciami

Štruktúra

Detailná štruktúra:



Účastníci

- **Component** (komponent)
 - deklaruje rozhranie (interface)
 - implementuje štandardné správanie
 - deklaruje rozhranie pre prístup k synom (a aj ich menežovanie)
 - deklaruje rozhranie pre prístup k rodičovi
- **Leaf** (list)
 - reprezentuje listový objekt, nemá deti
 - definuje správanie pre primitívne objekty
- **Composite** (uzol)
 - definuje správanie pre objekty, ktoré majú synov
 - uchováva referencie na svojich synov
 - implementuje operácie pre prácu so synmi
- **Client** (klient)
 - manipuluje s objektami pomocou rozhrania objektu Component

Vzťahy (spolupráca)

Client používa rozhranie triedy **Component** na prácu s objektami. Listy väčšinou vykonávajú požiadavku priamo, uzly ju väčšinou posielajú synom.

Dôsledky

Návrhový vzor **Composite**:

- definuje hierarchiu tried pozostávajúcu z primitívnych objektov a kompozícií
- uľahčuje klientovi manipuláciu s objektami
- uľahčuje pridávanie nových typov komponentov
- môže spraviť dizajn až príliš všeobecný
 - môže sa stať, že bude nutné obmedziť nejaké typy komponentov (treba riešiť kontrolami počas behu programu)

Implementácia

Pri implementácii si treba uvedomiť viaceru faktov:

- explicitná referencia na rodiča
 - je potrebná na prechádzanie štruktúrou
 - treba ju starostlivo menežovať (najlepšie iba pri add a remove operáciach v triede **Component**)
- zdieľanie komponentov
 - pre redukciu priestorovej náročnosti
 - problém s rodičmi a prechádzaním dokumentu
- maximalizovanie rozhrania objektu **Component**
 - objekt **Component** by mal pokryvať celú funkcionality
 - definuje sa všeobecné správanie, ktoré si jednotlivé konkrétné triedy prispôsobia
- deklarovanie operácií na prácu so synmi - kde sú definované operácie Add a remove?
 - ak sú deklarované v triede **Component**, zachováva sa uniformita, ale stráca sa bezpečnosť (niekto ich môže predefinovať v listoch)
 - ak sú deklarované v triede **Composite**, zachováva sa bezpečnosť, ale stráca sa uniformita
- má **Component** obsahovať zoznam synov?
 - áno, ak sú tam definované operácie add a remove
 - zbytočne to však zaberá miesto v listoch
- poradie synov
 - môže byť dôležité
 - rieši ho návrhový vzor Iterator
- Kešovanie
 - uchovávanie určitých informácií kvôli urýchleniu (napr. priestor, ktorý zaberá obrázok, aby sa nemusel vykreslovať, ak je neviditeľný)
 - netreba zabúdať na starostlivé aktualizovanie týchto údajov
- Mazanie komponentov
 - väčšinou robi rodič

Strategy pattern

Účel

[Upravit](#)

Definovať množinu algoritmov, zapúzdríť každý z nich a umožniť ich vzájomnú výmenu.

[Upravit](#)

Ďalšie pomenovania

Policy

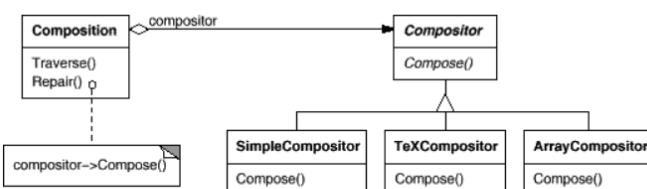
[Upravit](#)

Motívacia

Ak by sme chceli formátovací algoritmus implementovať priamo v kóde, narazíme na viaceré problémov:

- kód je zložitejší (hlavne ak máme viaceré algoritmy)
- možno nepoužijeme všetky algoritmy, zbytočne ich budeme mať v kóde
- ťažko sa pridávajú nové algoritmy

Všetky tieto problémy sa vyriešia definovaním tried, ktoré zapúzdrujú formátovacie algoritmy.



Trieda **Compositor** je abstraktná trieda pre formátovacie algoritmy. Konkrétné algoritmy sú z nej odvodené. **SimpleCompositor** jednoducho počíta, či už nie sме na konci riadku a ak áno, dà slovo do nového riadku. **TeXCompositor** robí krajšie (ale časovo náročnejšie) zarovnávanie do riadkov, príčom rovnomerne distribuuje medzery a môže aj podfarbovať text, čo signalizuje, že to ešte nie je zarovnané. **ArrayCompositor** dá do každého riadku rovnaký počet objektov (napr. ak usporiadavame ikony).

[Upraviť](#)

Použiteľnosť

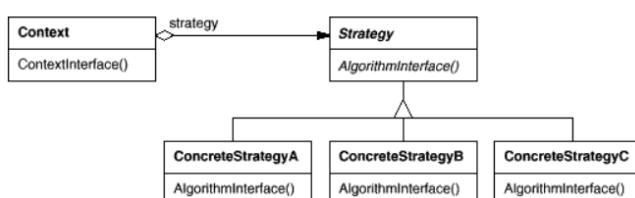
Návrhový vzor **Strategy** sa používa, ak:

- veľa tried sa líši iba v ich správaní
- potrebujeme veľa rôznych druhov algoritmov
- algoritmus používa dátá o ktorých nemá **Client** vedieť
- trieda definuje veľa typov správania, ktoré sú implementované pomocou viacnásobných podmienok

[Upraviť](#)

Štruktúra

Detail štruktúry:



[Upraviť](#)

Účastníci

- 1. **Strategy**
 - deklaruje rozhranie
- 2. **ConcreteStrategy**
 - implementuje algoritmus
- 3. **Context**
 - je konfigurovaný pomocou **ConcreteStrategy** objektu
 - uchováva referenciu na **Strategy** objekt
 - môže definovať rozhranie pomocou ktorého Strategy objekt pristupuje k jeho dátam

[Upraviť](#)

Vztahy (spolupráca)

- **Strategy** a **Context** spolupracujú pri implementácii algoritmu. **Context** môže poskytnúť potrebné dátá alebo referenciu na seba.
- **Context** posielá požiadavky od svojich klientov. Klienci zvyčajne vytvoria konkrétnu Strategy triedu, dajú ju **Context-u** a potom už pracujú výlučne s objektom **Context**.

Dôsledky

- Stratégie definujú triedy súvisiacich algoritmov. Často preto možno využiť dedičnosť pri implementovaní stratégií.
- Stratégie sú alternatívou pre dedičnosť (definovanu v triede **Context**). Mohli by sme aj priamo implementovať formátovací algoritmus v triede **Context** a modifikovať ho v odvodených triedach, je to však veľmi neprehľadné.
- Stratégie eliminujú podmienkové príkazy.
- Stratégie môžu ponúkať rôzne implementácie pre rovnaké správanie.
- Klient sa musí rozhodovať medzi rôznymi stratégiami, čo ho môže zaťažovať.
- Vzniká komunikačná rézia medzi **Strategy** a **Context** objektom. Vo väčšine prípadov je však zanedbateľná.
- Vzniká väčší počet objektov, možno ich však urobiť bezstavové a zdieľať ich.

[Upraviť](#)

Implementácia

Definovanie rozhrani objektov **Strategy** a **Context**.

Treba sa zamyslieť nad tým, ako bude najvhodnejšie odovzdávať potrebné informácie. **Context** môže poskytnúť potrebné dátu ako parametre metódy **Compose**, alebo poskytne referenciu na seba a konkrétna stratégia si už potom od neho vypýta to, čo potrebuje. To, ktoré riešenie je výhodnejšie, závisí od konkrétneho problému.

Definovanie **Strategy** objektu ako dobrovoľného.

Trieda **Context** môže byť naprogramovaná aj tak, že nemusí mať priradenú žiadnu stratégiu (t.j. pri vytváraní objektu typu **Context** sa mu nepriradi žiadna stratégia). V tomto prípade sa potom vykoná preddefinované správanie. Výhodou je, že klienti nie sú zaťažovaní výberom stratégie, ak nechcú.

6 Decorator, abstract factory:

Decorator pattern

[Prekiaľaj tejto stránky](#) · [Slovenčina \(SK\)](#)

Účel

[Upraviť](#)

Dynamicky pridať ďalšiu funkcionality objektu. **Decorator** predstavuje oproti dedeniu pružnejší spôsob pre rozšírenie funkcionality.

Ďalšie pomenovania

[Upraviť](#)

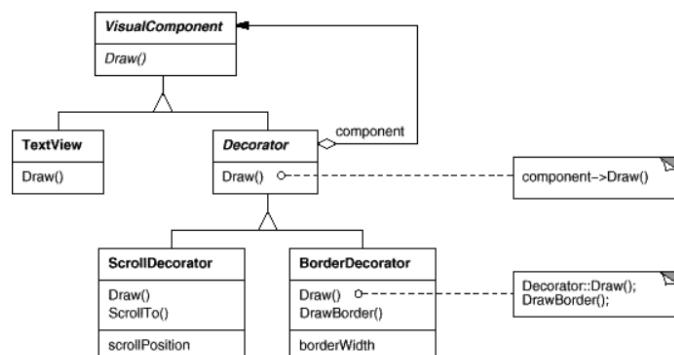
Wrapper

Obsah
<ul style="list-style-type: none">• Účel• Ďalšie pomenovania• Motívacia• Použiteľnosť• Štruktúra• Účastníci• Vzťahy (spolupráca)• Dôsledky• Implementácia• Jednoduchý kód• Súvisiace návrhové vzory

Motívacia

[Upraviť](#)

Niekedy chceme pridať funkcionality jednotlivým objektom a nie celej triede. Toto je aj prípad pridania posuvnej lišty a okraju pre vizuálne komponenty, ktorým sme sa zaoberali v prípadovej štúdii. Popísali sme aj, prečo je tento spôsob lepší ako priame dedenie.



[Upravit](#)

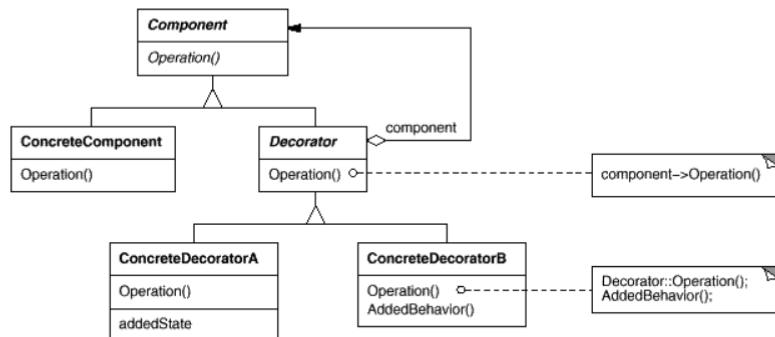
Použiteľnosť

Návrhový vzor Decorator sa používa:

- na pridanie funkcionality pre jednotlivé objekty dynamicky a transparentne (t.j. bez ovplyvnenia ostatných objektov)
- na pridanie funkcionality, ktorá je voliteľná
- ak rozšírenie funkcionality pomocou dedenia je nepraktické; ak napr. máme veľa typov rozšírení a tak by nám exponenciálne narastal počet odvodnených tried

[Upravit](#)

Štruktúra

[Upravit](#)

Účastníci

1. **Component** = deklaruje rozhranie (interface) objektov, ktoré môžu mať dynamicky pridanú funkcialitu
2. **ConcreteComponent** = deklaruje objekt, ktorému chceme pridať funkcialitu (konkrétny komponent, napríklad TextView)
3. **Decorator** = uchováva referenciu na objekt typu **Component** (alebo jeho potomka), ktorému pridáva funkcialitu; definuje rozhranie, ktoré zodpovedá rozhraniu objektu **Component**
4. **ConcreteDecorator** = pridáva konkrétnu funkcialitu objektu typu **Component**, alebo jeho potomkovi (konkrétny dekorátor, napríklad ScrollDecorator?, BorderDecorator)

[Upravit](#)

Vzťahy (spolupráca)

Decorator posielá väčšinu požiadaviek priamo objektu **Component**, niekedy však môže predtým alebo potom vykonať aj svoju vlastnú funkcialitu.

[Upravit](#)

Dôsledky

Návrhový vzor Decorator má nasledovné dve výhody:

1. Je viac flexibilnejší ako dedičnosť. **Decorator** pridáva funkcialitu za behu (pri dedičnosti by sme museli vytvoriť nový objekt), dovoľuje ľuboľne kombinovať pridanú funkcialitu (v poradí, aké potrebujeme), pričom môžeme aj dvakrát pridať tú istú funkcialitu (dvojitý rámk).
2. Obmedzuje vytváranie zbytočne zložitých objektov. **Decorator** ponúka prístup pláť iba za to, čo využívaš. Namiesto možnosti podporovať všetky možné funkciality vo veľkej konfigurovateľnej triede definuje radšej jednoduchú triedu, ktorej pridáva funkcialitu pomocou objektov **Decorator**. Výsledkom toho je, že aplikácia nemusí platiť za funkcialitu, ktorú nepoužíva. Navyše sa ľahko definujú nové typy dekorátorov nezávisle na triedach, ktorým pridávajú funkcialitu.

Návrhový vzor **Decorator** má nasledovné dve nevyhody:

1. **Decorator** aj jeho **Component** nie sú identické. Hoci **Decorator** sa správa ako transparentný uzáver, v skutočnosti to nie je objekt **Component**, čo môže spôsobovať problémy, keď sa spoliehame na identitu objektu.
2. Pri použíti tohto návrhového vzoru vzniká počas behu programu veľa malých objektov, ktoré častokrát vyzerajú rovnako, môžu sa lísiť iba ich vzájomným poprepájaním. Toto môže robiť problém pri debugovaní ľuďom, ktorí systém nepochopili.

Implementácia

Pri implementácii si treba uvedomiť viaceré faktov:

- Prispôsobenie rozhrani. Rozhranie objektu **Decorator** musí zodpovedať rozhraniu objektu, ktorému funkcionality pridáva.
- Vynechanie abstraknej triedy **Decorator**. Ak chceme pridať iba jednu jedinú funkcionality, môžeme vyniechať abstraktnej triedy **Decorator** a môžeme priamo pracovať s nejakou triedou **ConcreteDecorator**.

Toto riešenie je vhodné najmä ak nemôžeme rozširovanú triedu modifikovať a tak do nej priamo doplniť požadovanú jednu funkcionality.

- Ľahká trieda **Component**. V abstraktej triede **Component** by sme sa mali zamierať na zadefinovanie iba potrebných metód. Nemala by obsahovať zbytočné metódy a ani premenné, aby nezaťažovala triedu **Decorator** (a ani ostatné od nej odvodené triedy).

Ak je však trieda Component priliš ťažká, niekedy je lepšie nepoužiť návrhový vzor Decorator, ale radšej nejaký iný, napríklad Strategy.

Abstract Factory pattern

Účel

Obsah
<ul style="list-style-type: none">• Účel• Ďalšie pomenovania• Motívacia• Použiteľnosť• Struktúra• Vzťahy (spolupráca)• Dôsledky• Implementácia• Jednoduchý kód• Súvisiace návrhové vzory

Poskytuje rozhranie na vytváranie rodín príbuzných objektov bez špecifikovania ich konkrétnych tried.

Obsah

Ďalšie pomenovania

Kit

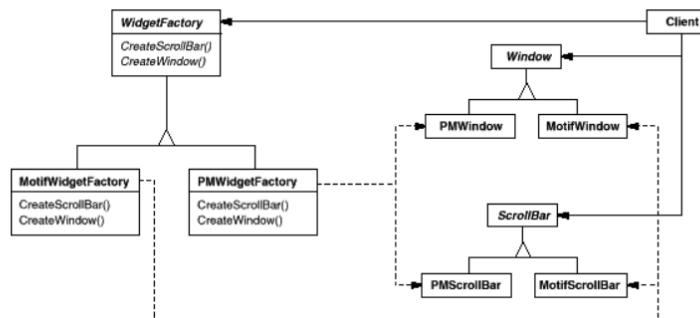
Obsah

Motívacia

Upraviť

Predpokladajme, že máme vývojové prostredie podporujúce rôzne vzhľady (look-and-feel). Ak by sme pri vytváraní nových komponentov priamo volali konštruktor konkrétnej triedy (napr. PMButton), t'azko by sa neskôr menil vzhľad aplikácie.

Vytvorime preto abstraktnej triedu **WidgetFactory**, ktorá obsahuje rozhranie na vytvorenie ľubovoľného komponentu (napr. **Button**, **Menu**). Musíme mať však aj abstraktnej triedu pre každý komponent (napr. **Button**) a od nej odvodené konkrétné triedy (napr. **PMButton**, **MotifButton**, ...). Pre každý vzhľad. Triedy **PMWidgetFactory**, **MotifWidgetFactory**, ..., ktoré sú odvodené od abstraktej triedy **WidgetFactory** potom vrátia konkrétnu implementáciu daného objektu (napr. **PMButton**, **MotifButton**, ...). Klient sa tak nemusí starat o to, aký objekt má vytvárať, iba si najprv vytvorí továren (WidgetFactory) pre požadovaný vzhľad (napr. **PMWidgetFactory**) a potom si už od nej pôta jednotlivé komponenty (t.j. napr. pre **PMWidgetFactory** vráti funkcia **CreateButton** objekt typu **PMButton**). Získame tým to, že sa nemusíme starat o to, akú verziu (vzhľadovú) objektu máme vytvoriť a navyše tým máme aj zabezpečené, že všetky komponenty budú rovnakého vzhľadu a nemôžeme sa pomýliť.



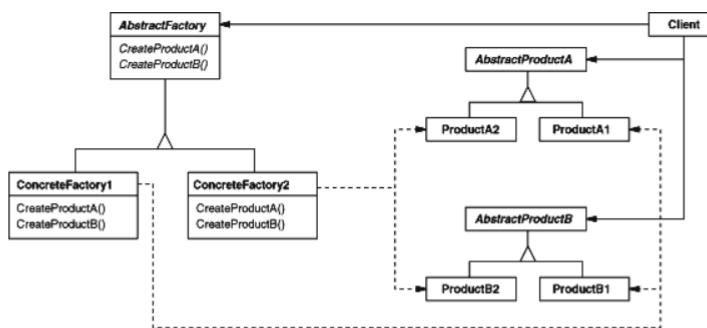
Použiteľnosť

Návrhový vzor **Abstract factory** sa používa, ak:

- systém má byť nezávislý na tom, ako sú jeho produkty vytvárané
- systém má byť nakonfigurovaný s jednou z viacerých rodín produktov
- rodina príbuzných objektov produktov má byť používaná spolu a chceme pomôcť zabezpečiť toto obmedzenie
- chceme poskytovať knižnicu tried produktov, pričom chceme sprístupniť iba ich rozhrania a nie ich implementáciu

Štruktúra

[Upravit](#)



[Upravit](#)

Vzťahy (spolupráca)

- Za normálnych okolností sa počas behu programu vytvára iba jediná všeobecne dostupná inštancia triedy **AbstractFactory**. Táto konkrétna továreň potom vytvára všetky komponenty (objekty majúce požadované správanie). Ak potrebujeme mať objekty viacerých typov, musíme si vytvoriť ďalšiu továreň (inštanciu indej triedy **ConcreteFactory**)
- AbstractFactory** presúva vytváranie konkrétneho objektu na niektorú zo svojich odvodených tried **ConcreteFactory**.

[Upravit](#)

Dôsledky

Návrhový vzor **AbstractFactory** má nasledovné výhody:

- Odstraňuje vytváranie konkrétnych tried z kódu aplikácie. Zodpovednosť za vytvorenie správneho typu objektu preneháva **AbstractFactory**, pričom klient pracuje iba s abstraktími rozhraniami a netrapi sa s konkrétnymi typmi.
- Dovoľuje jednoduchú výmenu rodín objektov. Keďže výber celej rodiny objektov je uskutočnený iba na jednom mieste, tam kde sa inicializuje **AbstractFactory**, dá sa tento výber ľahko zmeniť pomocou konfiguračného súboru. Dá sa však zmeniť aj počas behu programu. V našom prípade stačí vybrať novú **ConcreteFactory** a opäťovne vytvoriť užívateľské rozhranie.
- Pomáha zvyšovať konzistenciu medzi produktami. Keď sú jednotlivé objekty rodín produktov navrhnuté tak, že majú fungovať spolu, je dôležité, aby boli používané naozaj iba objekty z jednej rodiny. Návrhový vzor **AbstractFactory** veľmi jednoducho túto podmienku zabezpečuje.

Návrhový vzor **AbstractFactory** má nasledovné nevýhody:

- Podpora nových typov produktov je zložitejšia. Ak by sme chceli pridať nový typ produktu (napr. **SpecialButton**), museli by sme zmeniť rozhranie triedy **AbstractFactory** a teda aj všetky triedy **ConcreteFactory**.

[Upravit](#)

Implementácia

Ako sme už spominali, je ľahšie pridať nové typy produktov. Mohli by sme zmeniť návrh tak, že by sme pridali nový parameter do metód, ktoré vytvárajú objekty. Tento parameter by špecifikoval typ objektu (menu, tlačidlo, ...), ktorý treba vytvoriť. Mohol by byť identifikátor triedy, číslo, string alebo hocičo iné, čo by identifikovalo typ produktu. Potom by sme v triede **AbstractFactory** mali iba jednu metódu **Make** s parametrom určujúcim typ produktu. Takéto riešenie je jednoduchšie použiteľné v jazykoch bez striknej typovej kontroly (napr. Smalltalk). V C++ môžeme toto riešenie použiť iba v prípade, ak všetky objekty majú spoločnú abstraktívnu triedu (alebo môžu byť pretypované na jednu triedu). Objekt tejto triedy potom dostávame ako výsledok funkcie, čím vzniká problém, že nemáme prístupné metódy špecifické pre daný typ produktu. Preto ho musíme väčšinou pretypovať, čo zas nemusí byť bezpečné. Je to však cena za toto flexibilnejšie riešenie.

7 Bridge, memento:

Bridge pattern

Účel

Oddeliť abstrakciu od implementácie tak, aby sa mohli meniť nezávisle na sebe.

... | Nový | Otvorit | Znovuformu | ...

Obsah
<ul style="list-style-type: none">• Účel• Ďalšie pomenovania• Motívacia• Použiteľnosť• Štruktúra• Účastníci• Vzáhy (spolupráca)• Dôsledky• Implementácia• Jednoduchý kód• Súvisiace návrhové vzory

Ďalšie pomenovania

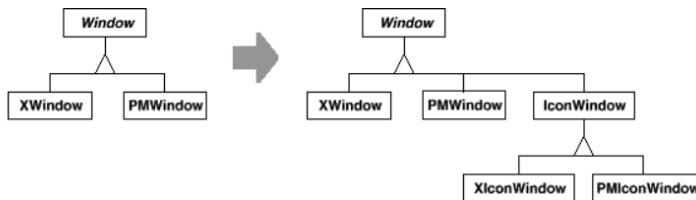
Handle/Body

Upravit

Motivácia

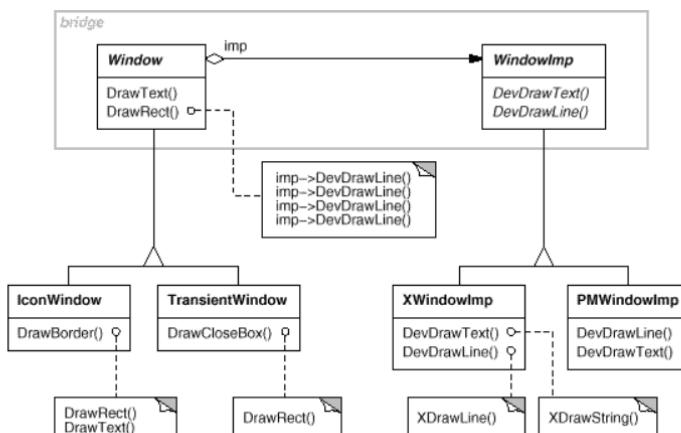
Ked' má abstrakcia viaceru možných implementácií, zvyčajná cesta, ako to naprogramovať je použitím dedenia. Abstraktná trieda definuje rozhranie a konkrétné podtrydy ju implementujú rôznymi spôsobmi. Tento prístup však nie je dostatočne flexibilný. Predstavme si implementáciu triedy **Window**, ktorú majú používať aplikácie, ktoré majú byť prenositeľné na viaceré oknové systémy (**XWindow**, **Macintosh**, ...). Môžeme vytvoriť abstraktívnu triedu **Window** a od nej pomocou dedenia odvodené triedy **XWindow** a **MacWindow**, ktoré implementujú danú triedu pre konkrétny systém. Toto riešenie má však nasledovné nevýhody:

- Ak by sme mali triedu **IconWindow** odvodenú od triedy **Window**, museli by sme urobiť pre ňu dve nové podtrydy **XIconWindow** a **MacIconWindow**. V konečnom dôsledku by sme museli vytvoriť jednu podtrydu pre každú dvojicu: typ systému, podtryda triedy **Window**. Počet nových podtryd by bol preto súčinom počtu podtryd triedy **Window** a počtu systémov.



- Zmena typu systému je príliš zložitá, keďže pri vytváraní okna už vytváram okno konkrétneho typu pre daný typ systému a zmena tohto typu sa nedá jednoducho urobiť niekde na jednom mieste v kóde. (Bližšie sme sa týmto problémom zaoberali v [pripradovej štúdií](#) v bode 5).

Návrhový vzor **Bridge** rieši tento problém oddelením hierarchií tried pre abstrakciu a implementáciu okna. Máme hierarchiu triedy **Window** (rôzne typy okien) a hierarchiu triedy **WindowImp** (implementácie pre rôzne oknové systémy).



Všetky operácie na podtrydach triedy **Window** sú implementované pomocou abstraktných operácií definovaných v triede **WindowImp**. Tento návrhový vzor sa nazýva **Bridge**, lebo premostuje abstrakciu a jej implementáciu, príčom sú však tieto dve hierarchie úplne nezávislé.

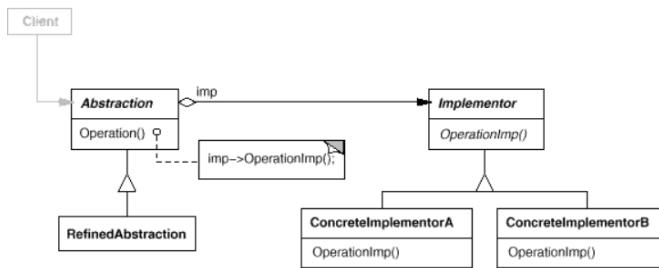
Použiteľnosť

Upravit

Návrhový vzor **Bridge** sa používa, keď:

- chceme zabrániť pevnému prepojeniu medzi abstrakciou a implementáciou; napríklad ak chceme meniť implementáciu počas behu programu
- abstrakcia a aj implementácia majú byť nezávisle rozširiteľné pomocou dedenia
- zmeny v implementácii nemajú mať vplyv na klienta, t.j. jeho kód sa nemusí prekompliovať
- chceme pred klientom kompletne schovať implementáciu

Štruktúra



[Upravit](#)

Účastníci

1. **Abstraction** (napríklad Window) = deklaruje rozhranie abstrakcie ; spravuje referenciu na objekt typu **Implementor**
2. **RefinedAbstraction** (napríklad IconWindow) = rozširuje rozhranie definované triedou **Abstraction**
3. **Implementor** (napríklad WindowImpl) = Definuje rozhranie pre implementačné triedy. Toto rozhranie sa nemusí byť rovnaké ako rozhranie triedy **Abstraction**. Väčšinou toto rozhranie poskytuje iba primitívne operácie, zatiaľ čo rozhranie triedy **Abstraction** poskytuje operácie na vyššej úrovni.
4. **ConcreteImplementor** (XWindowImpl, MacWindowImpl) = Implementuje rozhranie triedy **Implementor** a definuje jeho konkrétnu implementáciu

[Upravit](#)

Vzťahy (spolupráca)

[Upravit](#)

Abstrakcia posielá požiadavky klienta príslušnému implementátorovi.

Dôsledky

[Upravit](#)

Návrhový vzor Bridge má nasledovné dôsledky:

- **Oddelenie abstrakcie a implementácie** = Implementácia nie je pevne zviazaná s abstrakciou a preto môže byť menená aj počas behu programu. Toto oddelenie zároveň eliminuje závislosti pri komplikovaní, preto nemusíme prekomplikovať abstraktnú triedu ak sa zmenila implementácia. Navyše toto oddelenie prispieva k lepšej štrukture systému.
- **Zvyšená rozšíritelnosť** = Môžeme nezávislo rozširovať hierarchie tried abstrakcie a implementácie.
- **Ukrytie implementačných detailov pred klientom**.

Implementácia

[Upravit](#)

Pri implementácii si treba uvedomiť viacero skutočností:

- Iba jeden **ConcreteImplementor** = V situácii, keď bude iba jedna trieda **ConcreteImplementor**, nepotrebjeme abstraktnú triedu **Implementor**. Ide o degenerovaný prípad návrhového vzoru **Bridge**. Toto sa môže stať, ak chceme iba oddeliť implementáciu od abstrakcie, alebo chceme skryť nejaké implementačné záležitosti pred klientom.
- Vytvorenie správneho objektu **Implementor**. Kto vytvára objekt **Implementor**? Ak abstrakcia vie o všetkých triedach **ConcreteImplementor**, tak ho môže vytvárať sama. Takisto sa môže určiť preddefinovaná implementácia, ktorá sa potom môže meniť. Ďalšou možnosťou je prenechanie zodpovednosti inému objektu - definovanie **AbstractFactory**, ktorá bude vraciať správnu implementáciu.
- Zdieľanie objektov **Implementor**. Jednotlivé objekty Implementor sa môžu zdieľať, pričom sa uchováva počet referencií, ktoré naň odkazujú.

Memento pattern

Účel

[Upraviť](#)

Bez narušenia zapuzdrenia zachytiť a externe uchovať interný stav objektu tak, aby mohol byť tento objekt neskôr obnovený do tohto stavu.

[Upraviť](#)

Ďalšie pomenovania

Obsah

- Účel
- Ďalšie pomenovania
- Motívacia
- Použiteľnosť
- Struktura
- Účastníci
- Vzťahy (spolupráca)
- Dôsledky
- Implementácia
- Súvisiace návrhové vzorce

Token

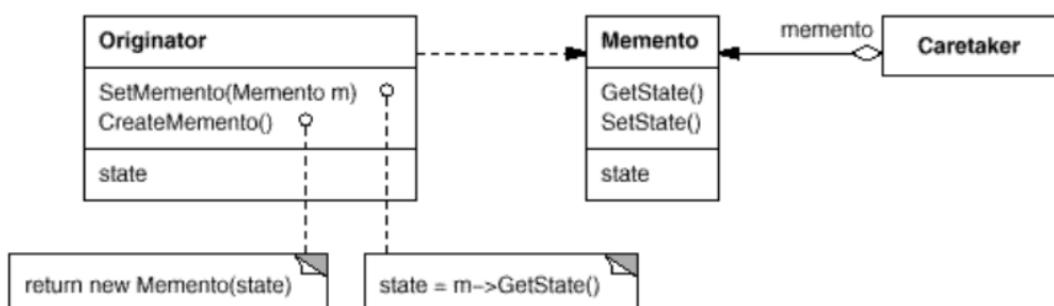
Motívacia

- Zapuzdrenie časti stavu objektov znemožňuje jeho externé uloženie.
- Porušenie zapuzdrenia narúša spoľahlivosť a rozširiteľnosť.

Použiteľnosť

- Keď je potrebné uložiť snímku stavu (alebo jeho časti) nejakého objektu tak, aby mohol byť tento stav neskôr obnovený.
- Keď priame rozhranie pre získanie stavu by zviditeľnilo implementačné detaily a narušilo zapuzdrenie objektu.

Štruktúra



Účastníci

1. Memento

- Pamäta si interný stav objektu *Originator*
- Chráni stav pred prístupom od iného objektu ako *originator*

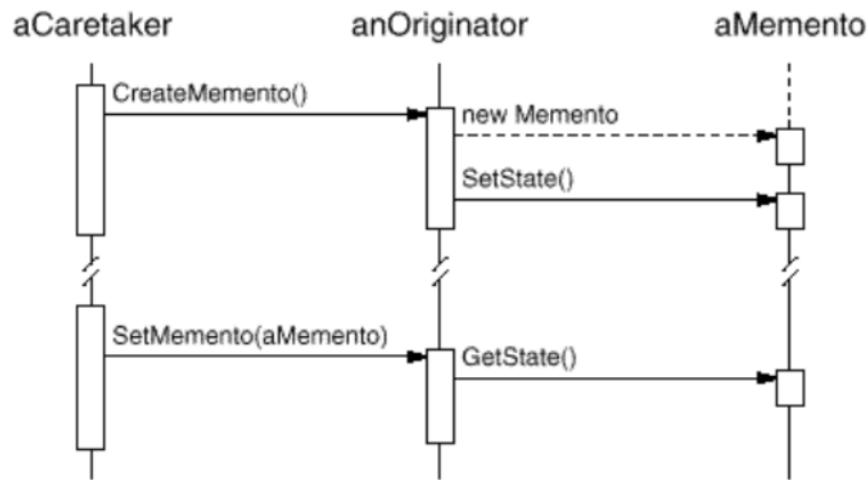
2. Originator

- Vytvorí *memento* obsahujúce snímku jeho aktuálneho stavu
- Použije *memento* na obnovenie svojho stavu
- Používa široké rozhranie mementa

3. Caretaker

- Zodpovedá za uloženie mementa
- Nikdy nepracuje s jeho obsahom
- Používa úzke rozhranie mementa

Vzťahy (spolupráca)



1. Caretaker

- Požaduje memento od *originátora*
- Nejaký čas si ho drží
- Pošle ho naspäť *originátorovi*, alebo ani nikdy nemusí

2. Memento

- Je pasívne
- Iba *originátor*, ktorý ho vytvoril bude do neho zapisovať, alebo z neho čítať

Dôsledky

- Zachovanie hraníc zapúzdrenia
- Zjednodušuje originátora
- Používanie mement môže byť drahé
 - Ak sa má kopírovať veľké množstvo pamäte
- Definovanie úzkeho a širokého rozhrania
 - V niektorých jazykoch môže byť problém
- Skrytá cena starania sa o mementá
 - Caretaker nevie koľko informácie je v memente uložené

Implementácia

UMLAVSKA

C++

```
class State;

class Originator {
public:
    Memento* CreateMemento();
    void SetMemento(const Memento*);
    // ...
private:
    State* _state;
    // internal data structures
    // ...
};

class Memento {
public:
    // narrow public interface
    virtual ~Memento();

private:
    friend class Originator;
    // private members accessible only to Originator
    Memento();
    State* _state;
    void SetState(State*);
    State* GetState();
    // ...
}
```

Java

```
class State { /* ... */ }

class Originator {
    public Memento createMemento() { /* ... */ }
    public void setMemento(Memento m) { /* ... */ }
    // ...

    private State state;
    // internal data structures
    // ...

    public class Memento {
        private final State state;
        private Memento (State state) { this.state = state; }
        private State getState () { return state; }
    }
}
```

V Java kóde sa používa vnorená trieda s privátou metódou `getState` ku ktorej môže pristupovať len rodičovská trieda `Originator`. V C++ sa na rovnaký účel používajú spriateľené triedy (`friend class`).

Použitie zo strany klienta (`Caretaker`) je rovnaké, na ukážku v Jave:

```
originator originator = new Originator();

/* Uložíme si počiatočný stav */
Memento m1 = originator.createMemento();

/* Nejaká práca s originatom, ktorá zmení stav */
originator.changeState();

/* Obnovíme pôvodný stav */
originator.setMemento(m1);

/* Originátor je teraz v rovnakom stave ako pred .changeState() */
```

8 Iterátor, Visitor:

Iterator pattern

[Pridať do tejto stránky](#) [Súťaž o článok](#)

Účel

[Upraviť](#)

Poskytuje spôsob, ako postupne pristupovať k prvkom štruktúry bez odkrycia vnútornej reprezentácie.

Ďalšie pomenovania

[Upraviť](#)

Cursor (špecialný prípad)

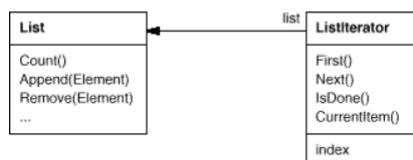
Obsah
<ul style="list-style-type: none">• iterator pattern• Účel• Ďalšie pomenovania• Motívacia• Použiteľnosť• Štruktúra• Účastníci• Vzťahy (spolupráca)• Dôsledky• Implementácia• Súvisiace návrhové vzory

[Upraviť](#)

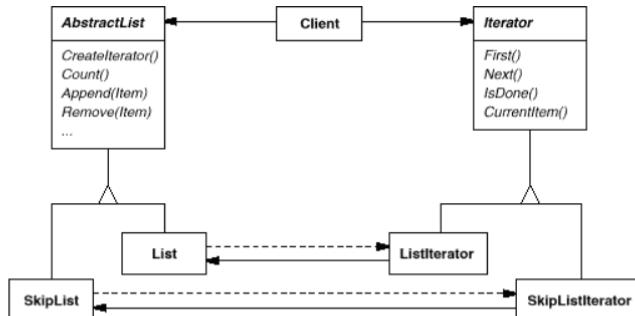
Motívacia

Štruktúrované objekty (napr. listy) by nám mali poskytovať možnosť prístupu k ich prvkom bez exponovania vnútornej štruktúry. Navyše môžeme chcieť prechádzať štruktúrou rôznymi spôsobmi. Nechceme však príslušné triedu zaplavíť množstvom metód na všetky možné spôsoby prechádzania, aké si len dokážeme vymyslieť (a možno by aj tak niekto chcel ešte iný spôsob). Riešením je vytvoriť nový objekt, ktorý prevezme zodpovednosť za prechod štruktúrou. Tento objekt si bude pamätať, v ktorých objektoch už bol. Trieda **Iterator** definuje rozhranie pre objekty zodpovedné za prechod štruktúrou.

ListIterator bude zodpovedný za prechod štruktúrou **List**



Konkrétny **List**, ktorý má prechádzať, sa poskytne objektu **ListIterator** už pri jeho vytváraní. Oddelením algoritmu na prechod štruktúrou od štruktúry môžeme jednoducho bez zasahovania do existujúceho kódu pridať nové algoritmy. Všimnime si, že iterátor a list sú zviazané. T.j. klient pracuje s konkrétnym typom štruktúry. Radi by sme však zmenili typ štruktúry bez zmeny klientovho kódu. Majme napríklad **SkipList**, čo je štruktúra s vlastnosťami podobnými balancovaným stromom. Radi by sme vytvorili kód, ktorý bude schopný pracovať s oboma týmito štruktúrami. Definujeme si preto triedu **AbstractList**, ktorá poskytuje spoločné rozhranie pre manipuláciu s listami. Potom môžeme definovať podtrydy triedy **Iterator** pre každú podtrydu triedy **AbstractList**.



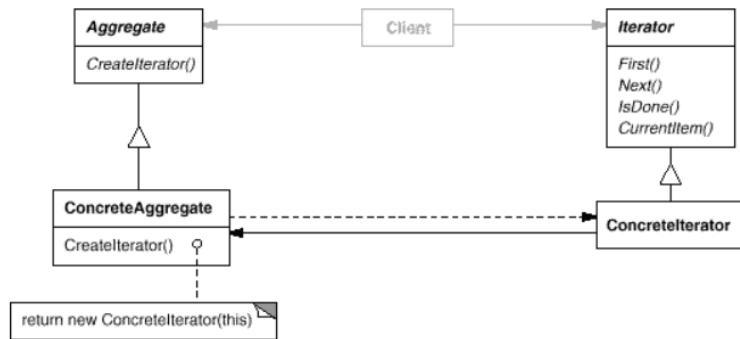
Pri vytváraní konkrétnego iterátora nám pomôže samotný list. V triede **AbstractList** si zadeklarujeme funkciu **CreateIterator** a necháme už na príslušné podtrydy, aby vytvorili iterátor vhodný pre ich štruktúru (hovoríme o polymorfickom iterátori).

Použiteľnosť

Návrhový vzor **Iterator** sa používa na:

- prístup k objektom zloženej štruktúry bez toho, aby sme sa zaoberali vnútornou reprezentáciu štruktúry
- podporu viacerých algoritmov na prechod štruktúrou
- poskytnutie jednotného rozhrania na prechádzanie rôznymi typmi štruktúr

Štruktúra



Účastníci

1. **Iterator**
 - definuje rozhranie pre prechod štruktúrou
2. **Concreteliterator**
 - Implementuje rozhranie triedy **Iterator**
 - Uchováva si aktuálnu pozíciu počas prechodu štruktúrou
3. **Aggregate**
 - definuje rozhranie pre vytvorenie objektu **Iterator**
4. **ConcreteAggregate**
 - implementuje funkciu na vytvorenie konkrétneho objektu **Concreteliterator**

Vzťahy (spolupráca)

- Objekt **Concreteliterator** prechádza dokumentom, pričom vie, ktorý je aktuálny prvok a vie vypočítať nasledujúci prvok.

Dôsledky

Návrhový vzor **Iterator** má nasledovné dôsledky:

1. Poskytuje možnosť jednoduchého zmenenia spôsobu prechodu štruktúrou.
2. **Iterator** zjednodušuje rozhranie triedy **Aggregate** (lebo metódy na prechod štruktúrou sú vybrané preč).
3. Jedna štruktúra môže byť prechádzaná viacerými iterátormi naraz v tom istom čase.

Implementácia

úprava

Iterator má veľa variant a alternatív. Niektoré programovacie jazyky priamopodporujú niektoré z nich.

1. Kto kontroluje prechod? Základným problémom je, či prechod štruktúrou kontroluje klient - vtedy ide o externý iterátor, alebo ho kontroluje iterátor sám - vtedy ide o interný iterátor. Pri externých iterátoroch si klient pýta aktuálny prvk a pomocou operácie *Next* (alebo podobnej) sa posúva na ďalší prvk. Pri interných iterátoroch sa iterátoru odovzdá operácia a on už sám prechádza štruktúrou a vykonáva danú operáciu. V jazykoch, ktoré nepodporujú anonymné funkcie (napr. C++) vzniká problém s implementáciou interných iterátorov. Externé iterátry sú navyše flexibilnejšie, hoci interné iterátry sú zasa jednoduchšie (na niektoré problémy v niektorých jazykoch).
2. Kto definuje prechodový algoritmus? **Iterator** nie je jediným miestom, kde môže byť definovaný algoritmus prechodu. Môže byť definovaný aj v triede **Aggregate** a **Iterator** sa použije iba na uchovanie stavu iterácie. Vtedy hovoríme iterátoru aj **Cursor**. Klient potom volá operáciu *Next* na objekte **Aggregate**, pričom **Cursor** je argument. Operáciu *Next* mení stav objektu **Cursor**. Tento spôsob môže byť užitočný, ak iterátor používa privátne premenné objektu **Aggregate**, ktoré nechceme (alebo nemôžeme) zprístupniť.
3. Ako robustná je iterácia? Ak počas prechodu štruktúrou túto štruktúru meníme (pridávame alebo mažeme prvky), môže sa stať, že niektorý prvk vynecháme alebo prejdeme viackrát, prípadne sa zacyklíme. Jednoduchým riešením by bolo zkopírovať celú štruktúru a prechádzať kópiou, je to však vo všeobecnosti príliš drahé. Je viacero spôsobov, ako zabezpečiť, aby bol iterátor robustný (t.j. prešiel správne štruktúrou aj ak sa bude meniť). Môže napríklad **Aggregate** pri každom pridávaní a mazaní prvkov prispôsobiť vnútorný stav iterátorov, ktoré vytvoril, alebo si potrebnú informáciu ukladá u seba.
4. Prídavné operácie iterátora. Medzi základné operácie iterátora patria: *First*, *Next*, *IsDone*, *CurrentItem*. Avšak aj niektoré ďalšie môžu mať svoj význam (napr. *Previous*, *SkipTo*).
5. Iterátori môžu mať privilegovaný prístup. Na iterátor môžeme pozerať ako na rozšírenie objektu **Aggregate**, ktorý ho vytvoril. **Iterator** a **Aggregate** sú úzko zviazané. Môžeme preto triedu **Iterator** definovať ako zpriateľenú(friend) triedu **Aggregate**.
6. Iterátori pre kompozitné objekty.
7. Null iterátor. **NullIterator** je degenerovaný iterátor, ktorý sa používa pre ošetrenie hraničných hodnôt. Jeho funkcia *IsDone* vracia vždy *true*, čo vlastne znamená, že už prešiel všetkými svojimi deťmi. **NullIterator** je používaný napríklad v listoch stromu, čím nám umožňuje jednotný prístup na prechádzanie štruktúrou.

Visitor pattern

Prednádaj tejto stránky: Slovenská verzia

Účel

Upraviť

Reprezentuje operáciu, ktorá má byť vykonaná na prvkoch štruktúry. **Visitor** nám dovoľuje definovať nové operácie bez zmeny tried elementov, na ktorých majú byť vykonané.

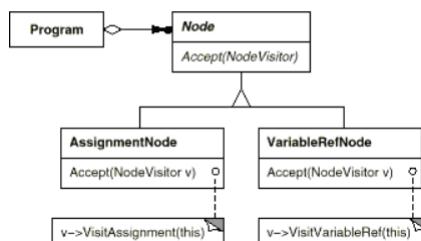
Upraviť

Ďalšie pomenovania

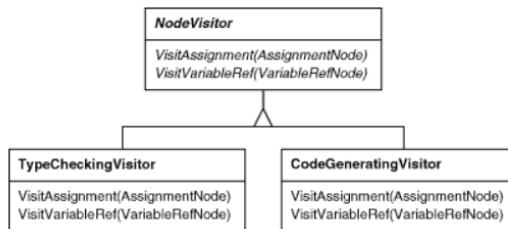
Upraviť

Motivácia

Majme komplítor, ktorý reprezentuje programy pomocou abstraktných syntaktických stromov. Potrebuje vykonávať operácie statickej analýzy, ako napr. kontrola, či sú všetky premenne definované, či je premennej priradená hodnota skôr, ako je použitá, kontrola typov, optimalizácia kódu, ... Navyše môže využívať syntaktické stromy na krajšie zobrazenie programu (odsadenie, ...), počítanie rôznych metrik, ... Väčšina z týchto operácií potrebuje, aby sa k jednotlivým typom uzlov pristupovalo rôzne (t.j. inak sa správať k priradeniu, inak k uzlom reprezentujúcim premenne, aritmetické výrazy, ...) . Na obrázku je časť hierarchie triedy **Node**.



Problémom je, že tým, že všetky operácie budeme implementovať vo všetkých podriedach triedy **Node**, dostaneme kód, ktorý je veľmi ťažko zrozumiteľný, ťažko sa udržiava a mení. Je to spôsobené tým, že metódy jednotlivých operačí sú pomiešané a vznikajú dlhé a neprehľadné triedy. Navyše pridanie novej triedy zvyčajne vyžaduje prekompilovanie všetkých tried. Bolo by lepšie, keby sa nová operácia mohla pridať bez toho, aby sa robili zmeny v triede **Node** a v jej podriedach. Obe tieto požiadavky môžeme splniť tak, že jednotlivé operačí zapuzďime do samostatného objektu nazvaného **Visitor**, ktorý budeme poskytovať uzlom syntaktického stromu, keď ním budeme prechádzať. Keď uzol 'akceptuje' daný objekt **Visitor** (t.j. v metóde **Accept**), pošle požiadavku objektu **Visitor**, v ktorej je zakomponovaný aj typ uzla. Navyše obsahuje daný uzol ako parameter. Objekt **Visitor** potom vykoná operáciu pre daný typ objektu. Napríklad komplátor, ktorý by nepoužíval objekt **Visitor** by mohol kontrolu typov implementovať pomocou metódy **TypeCheck** na uzloch syntaktického stromu. Ak by komplátor používal návrhový vzor **Visitor**, vytvoril by objekt **TypeCheckingVisitor** a volal by metódu **Accept** na uzloch syntaktického stromu. Argumentom metódy **Accept** by bol práve **TypeCheckingVisitor**. Každá trieda odvodnená od triedy **Node** by v metóde **Accept** volala naspäť metódu objektu **TypeCheckingVisitor**, ktorá je priadená danému typu uzla. Napríklad trieda **VariableRefNode** by volala metódu **VisitVariableRef**. To, čo zvyklo byť naimplementované v metóde **TypeCheck** v triede **VariableRefNode** bude teraz naimplementované v metóde **VisitVariableRef** v triede **TypeCheckingVisitor**. Aby sme nezostali iba pri kontrole typov, pridáme abstraktnú triedu **NodeVisitor**, ktorú potom budeme môcť používať aj na ostatné analýzy. Trieda **NodeVisitor** bude musieť deklarovať metódu pre každú triedu odvodnenú od triedy **Node**. Pre nové typy analýzy už len odvodíme novú triedu od triedy **NodeVisitor** a nemusíme meniť existujúce triedy.



Návrhový vzor **Visitor** definuje dve hierarchie tried. Jednu pre objekty, na ktorých sa operačie vykonávajú (hierarchia **Node**) a druhú pre objekty, ktoré definujú operačie na týchto objektoch (hierarchia **NodeVisitor**).

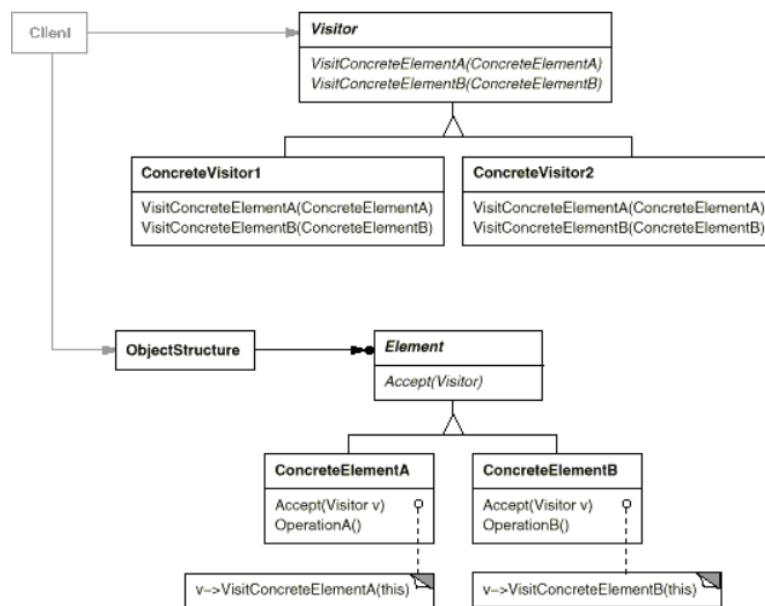
[Upravit](#)

Použiteľnosť

Návrhový vzor **Visitor** sa používa, keď:

- máme štruktúru objektov, ktorá obsahuje veľa rôznych tried objektov s rozdielnym rozhraním, pričom chceme vykonávať operačie na týchto objektoch, ktoré závisia od typu objektu.
- veľa rôznych a navzájom nesúvisiacich operačí sa má vykonať na objektoch v štruktúre objektov, pričom však nechceme, aby sme triedy týchto objektov 'zneprehľadnili' týmito operačiami.
- triedy reprezentujúce štruktúru objektov sa menia iba veľmi zriedkavo, ale často chceme definovať nové operačie nad touto štruktúrou.

Štruktúra



[1. Visitor \(NodeVisitor\)](#)

- deklaruje operáciu `Visit` pre každú triedu **ConcreteClass** v štruktúre objektov. Meno operácie a jej argumenty určujú triedu, ktorá posieľa požiadavku `Visit` triede **Visitor**.

[2. ConcreteVisitor \(TypeCheckingVisitor\)](#)

- implementuje každú operáciu `Visit` deklarovanú v triede **Visitor**.

[3. Element \(Node\)](#)

- deklaruje operáciu `Accept`, ktorá má ako argument triedu **Visitor**.

[4. ConcreteElement \(VariableRefNode\)](#)

- implemenuje operáciu `Accept`.

[5. ObjectStructure \(Program\)](#)

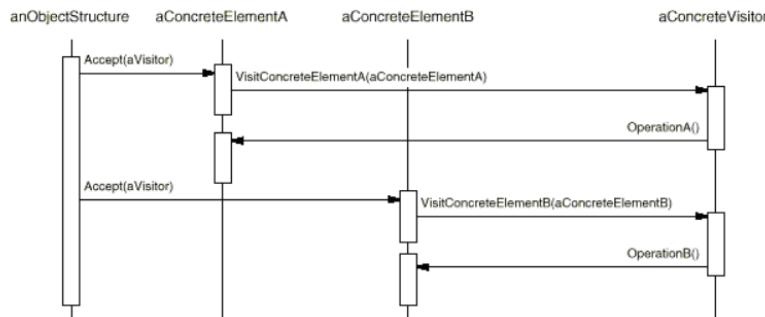
- vie vymenovať svoje elementy
- môže poskytovať 'vysoko-úrovňové rozhranie' na prístup k svojim elementom
- môže byť objektom typu **Composite**, alebo kolekcia typu list, množina, ...

[Upraviť](#)

Vzťahy (spolupráca)

- Klient, ktorý používa návrhový vzor **Visitor** musí vytvoriť objekt **ConcreteVisitor** a potom prechádzať štruktúrou objektov, pričom každý element navštíví s objektom **Visitor**.
- Keď je element navštivený, volá operáciu triedy **Visitor**, ktorá prislúcha jeho triede. Daný element poskytne sám seba ako argument tejto operácie, aby **Visitor** mohol pristupovať k jeho stavu, ak je to potrebné.

Na obrázku je znázornená spolupráca medzi štruktúrou objektov, objektom **Visitor** a dvoma elementami.



[Upraviť](#)

Dôsledky

Návrhový vzor **Visitor** má nasledovné výhody a nevýhody:

- Jednoduché pridanie novej operácie. Pri pridávaní novej operácie nemusíme zasahovať do tried reprezentujúcich štruktúru, ale iba vytvoríme novú podtriedu triedy **Visitor**.
- Združuje príbuzné operácie a oddeľuje nesúvisiace. Metódy vykonávajúce jednu operáciu nie sú roztrúsené v triedach reprezentujúcich štruktúru, ale sú sústredené v objekte **Visitor**. Naopak, metódy vykonávajúce rôzne operácie sú oddelené tým, že sú každá v inej triede odvodenej od triedy **Visitor**. Navýše všetky dátá špecifické pre nejaký algoritmus môžu byť ukryté v objekte **Visitor**.
- Pridanie novej triedy **ConcreteElement** je zložitejšie. Ak chceme pridať novú podtriedu triedy **Element (Node)**, musíme pridať novú metódu do triedy **Visitor** a teda aj do každej triedy **ConcreteVisitor**. Ak by sme v triede **Visitor** definovali preddefinované správanie, nemuseli by sme meniť každú triedu **ConcreteVisitor**, avšak vo väčšine prípadov sa takéto preddefinované správanie dá uplatniť iba na nepatrné percento tried.
- Prechádzanie hierarchiami tried. Na prechádzanie štruktúrou objektov môže byť použitý objekt **Iterator**. Objekt **Iterator** má obmedzenie v tom, že jednotlivé elementy musia mať spoločnú abstrakttnú triedu. Objekt **Visitor** toto obmedzenie nemá, keďže si môžeme zadefinovať operáciu `Visit` pre ľubovoľný typ objektu.
- Akumulovanie stavu. Objekt **Visitor** môže pri prechádzaní štruktúrou akumulovať stav. Bez tohto objektu by sme si museli stav odovzdávať pomocou argumentov, alebo by bol uložený v globálnych premenných.
- Porušenie zapúzdrenia. Návrhový vzor **Visitor** predpokladá, že rozhranie triedy **ConcreteElement** poskytuje dostatok informácie na to, aby mohol objekt **Visitor** vykonávať svoju činnosť. To častokrát speje k tomu, že **ConcreteElement** poskytuje vo svojom rozhraní prístup k svojmu vnútornému stavu, čo môže kompromitovať zapúzdrenie.

Implementácia

Pri implementácii si treba uvedomiť viacero faktov:

1. Double dispatch. Návrhový vzor **Visitor** používa techniku *double-dispatch*. Túto techniku priamo používajú niektoré jazyky (napr. CLOS). Jazyk C++ podporuje *single-dispatch*. V jazykoch podporujúcich *single-dispatch* určujú operáciu, ktorá bude vykonaná dve kritéria: meno operácie a typ prijimatelia. *Double-dispatch* jednoducho znamená, že operácia, ktorá bude vykonaná závisí od mena operácie a dvoch typov prijimatelov. Napríklad *Accept* závisí od typu dvoch objektov: **Visitor** a **Element**.
2. Kto je zodpovedný za prechod štruktúrou? Zodpovednosť za prechod štruktúrou môžeme umiestniť na tri miesta:
 - do štruktúry objektov (ak máme napríklad kolekciu)
 - do objektu **Visitor**
 - do samostatného objektu **Iterator**