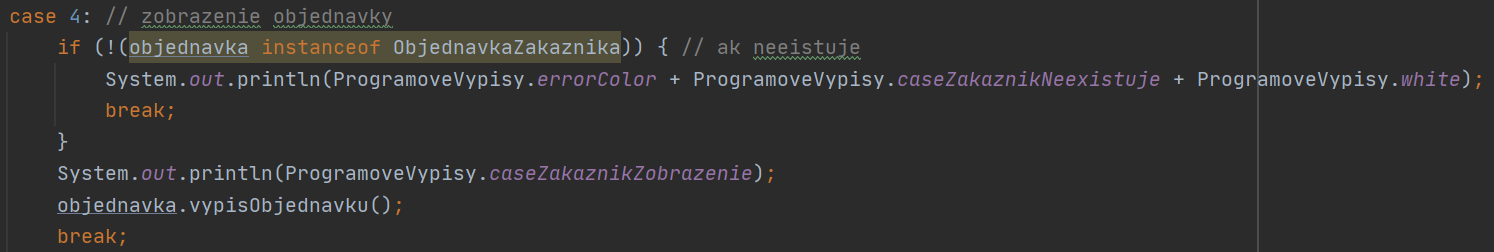
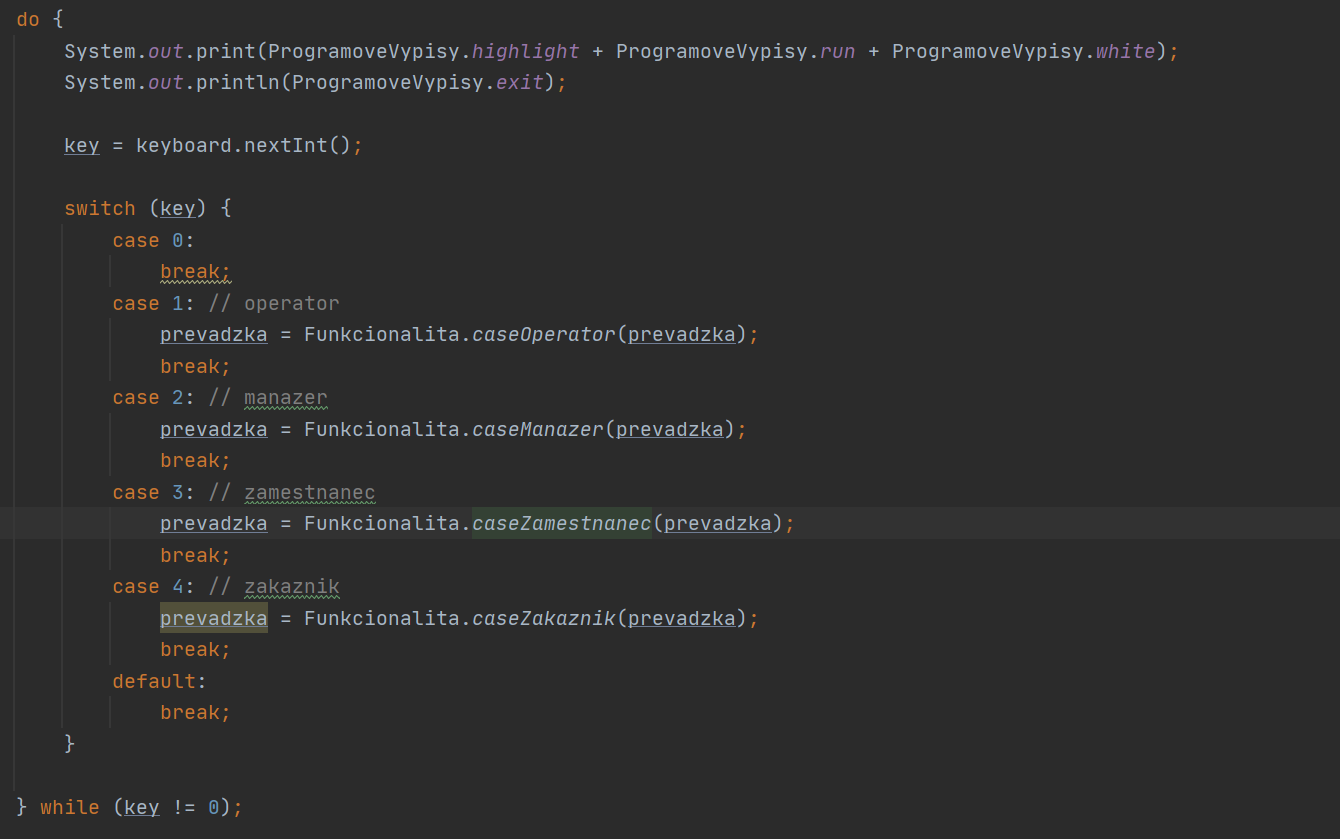


V mojom projekte je vnútorná logika a používeteľské rozhranie – konzolová aplikácia – rozdelené pomocou balíkov a modalu MVC.

Messages (trieda ProgramoveVypisy) predstavuje rozhranie pre pouzívateľa – view, a jeho funkcionalitu zabezpečuje v balíku core (controller) trieda **Funkcionalita**.

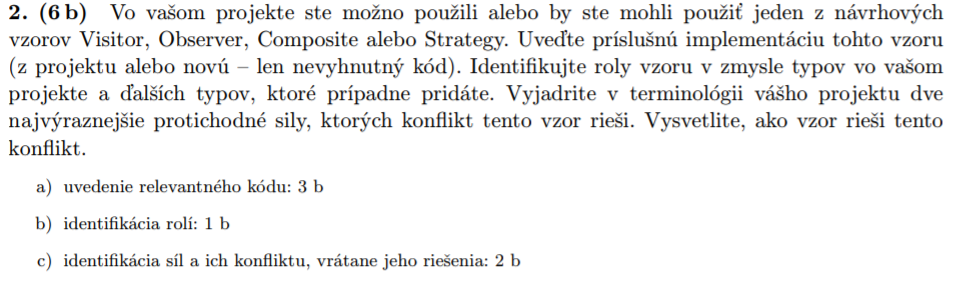
(využitie triedy **ProgramoveVypisy** triedou Funkcionalita)

O takéto oddelenie som sa snažila v celom kóde – všetky výpisy aj s formátovaním textu spravuje trieda ProgramoveVypisy, čím som chcela dosiahnuť prehľadnejší kód a jednoduchšiu úpravu používateľského rozhrania pri zmene funkcionality kódu.



(hlavný cyklus v triede main využívajúci ProgramoveVypisy a volajuci triedu Funkcionalita)

Dostatočné oddelenie je zabezpečené zordelením do balíkov a správnym použitím enkapsulácie.



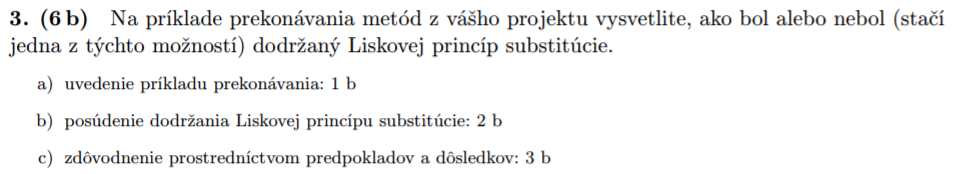
Vzor Visitor - účelom vzoru Visitor je definovať novú operáciu bez zavedenia úprav existujúcej objektovej štruktúry. Ide o pridanie novej funkcionality bez z meny existujúcich tried. Teda použijeme princíp otvorenosti a uzavretosti – nebudeme upravovať kód, ale budeme môcť ozšíroť funkcionalitu pridaním novej implementácie Visitora.

Konflikt spočíva v tom, že tento návrhový vzor potrebuje stálu údržbu pri pridávaní nových prvkov do štruktúry. Teda keď pridáme element musíme pridať Visitorovi novú metódu, ktorou ho spracuje.

Silná stránka je v oddelení algoritmu od tried v ktorých pracuje a uľahčuje pridávanie nivých operácií.

**Implementácia:**

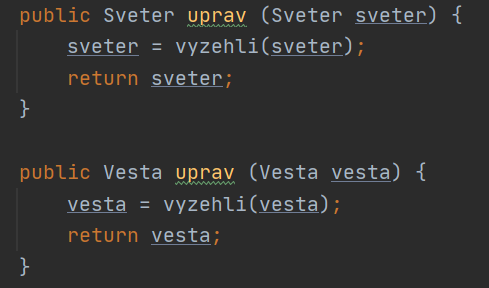
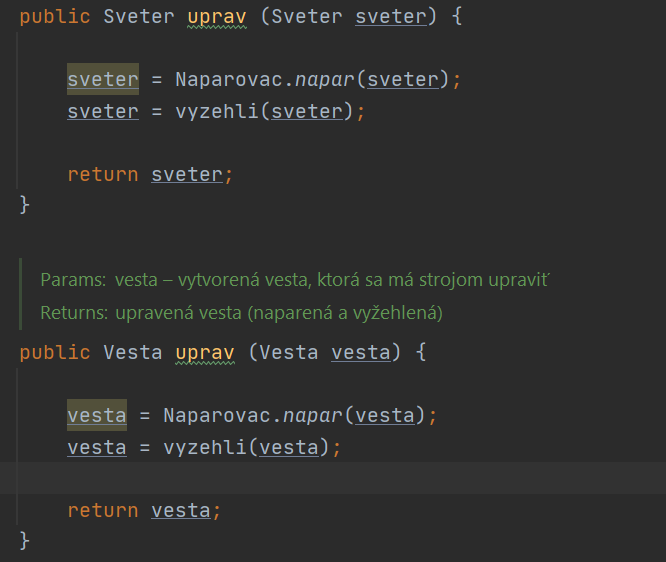
|  |  |
| --- | --- |
| // vytvorenie interface  public interface Visitable{  public void accept(Visitor visitor);  }  // konkrétny element  public class Rukavice implements Visitable{  private double cena;  private double spotrebaMaterialu;  // akceptovanie Visitora  public void accept(Visitor vistor) {  visitor.visit(this);  }  public double getCena() {  return cena;  }  public double getSpotrebaMaterialu() {  return spotrebaMaterialu;  }  } | // pridanie metód do Visitora pre každý element  public interface Visitor{  public void visit(Rukavice rukavice);    // ostatné elementy  public void visit(Sveter sveter);  public void visit(Vesta vesta);  } |



Pri Liskovej princípe substitúcie sledujeme to, či objekty uvažovanej nadtriedy je možné nahradiť objektami jej podtried.

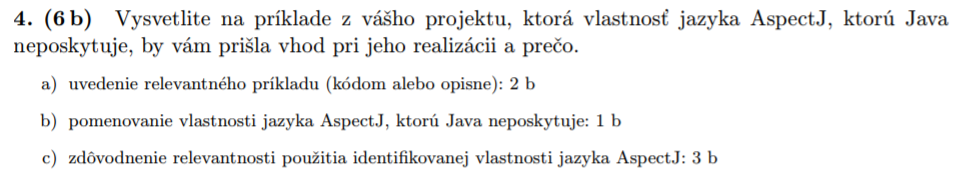
V mojom projekte je použité prekonávanie napríklad v tomto prípade:

V balíku **machines**, všetky stroje obsahujú metódu **uprav( ),** ktorá sa správa podľa toho, o aký konkrétny stroj ide. Napríklad trieda NaparovaciStrojNaZehlednie dedí od triedy StrojNaZehlenie metódu vyzehli, okrem ktorej pri volaní metódy uprav využije aj metódu napar.

(metóda uprav v triede StrojNaZehlenie) (metóda uprav v triede NaparovaciStrojNaZehlenie)

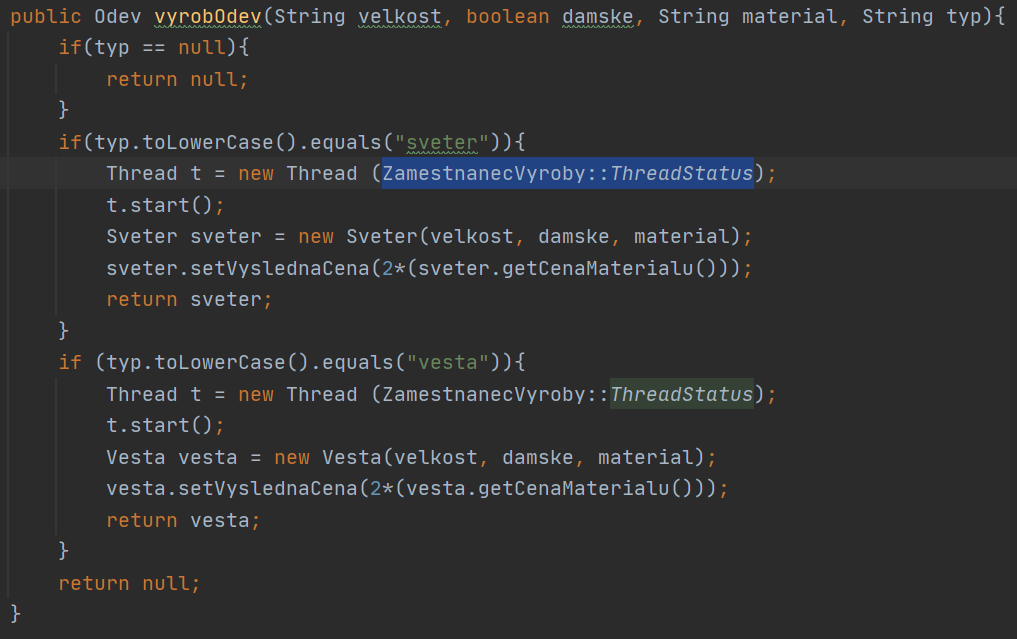
Tým pádom je možné nahradiť objekt nadriedy jej podtriedou, nie však naopak, nakoľko podtrieda je rozšírená a funkcionalitu, ktorú nadtrieda neobsahuje.

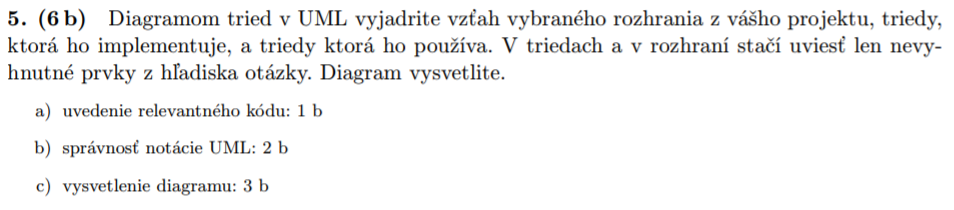
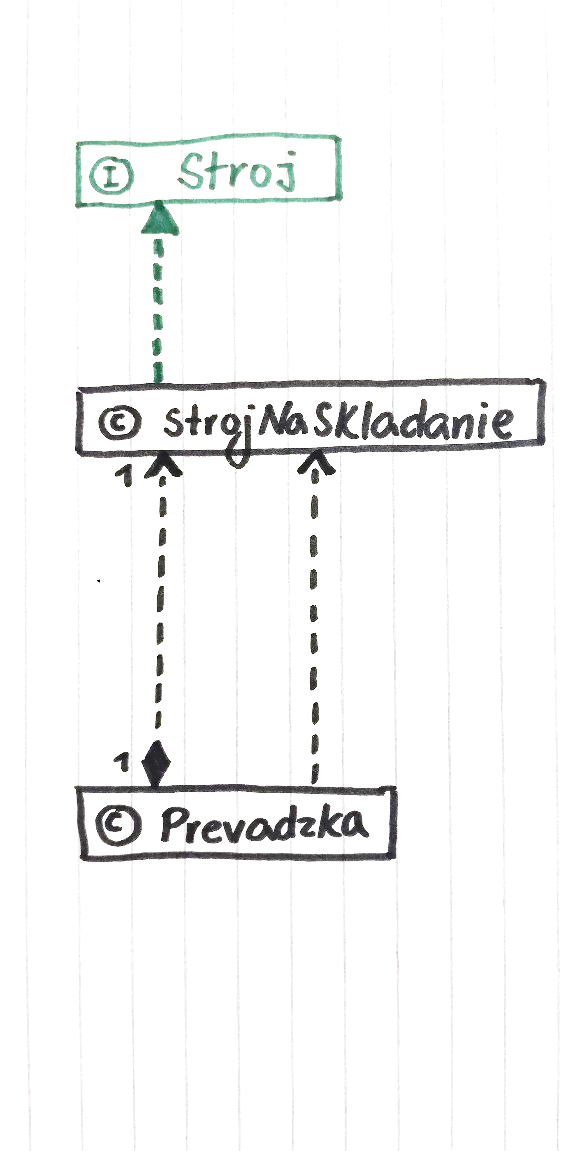


Problematika pracujúceho objektu – Java nepodporuje priamo prenesenie odkazu na metódu, ktorou ju možno spustiť.

Využíva sa na to implementovanie rozhrania Runnable a metódy run.

V triede SijaciStroj  používam preddefinované funkčné rozhranie Runnable na odkazovanie na statickú metódu. Rovnako sa pri tom využívajú vlákna (threads) pri vytváraní produktov.





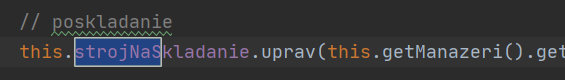
* Rozhranie: **Stroj**,



* trieda, kt. ho implementuje: **StrojNaSkladanie**,



* trieda, kt. ho používa: **Prevádzka**.



(v metóde odosliDoSkladu)

Rozhranie Stroj predstavuje predpis pre triedu StrojNaSkladanie, a trieda modelu Singleton Prevadzka vytvára inštanciu triedy StrojNaSkladanie, ktorú využíva.