Vybrané návrhové vzory mimo GoF

Návrhové vzory

Creational

- Simple factory method
- Lazy initialization
- Library class
- Multiton
- Object pool
- Dependency injection
- RAII

Simple factory method

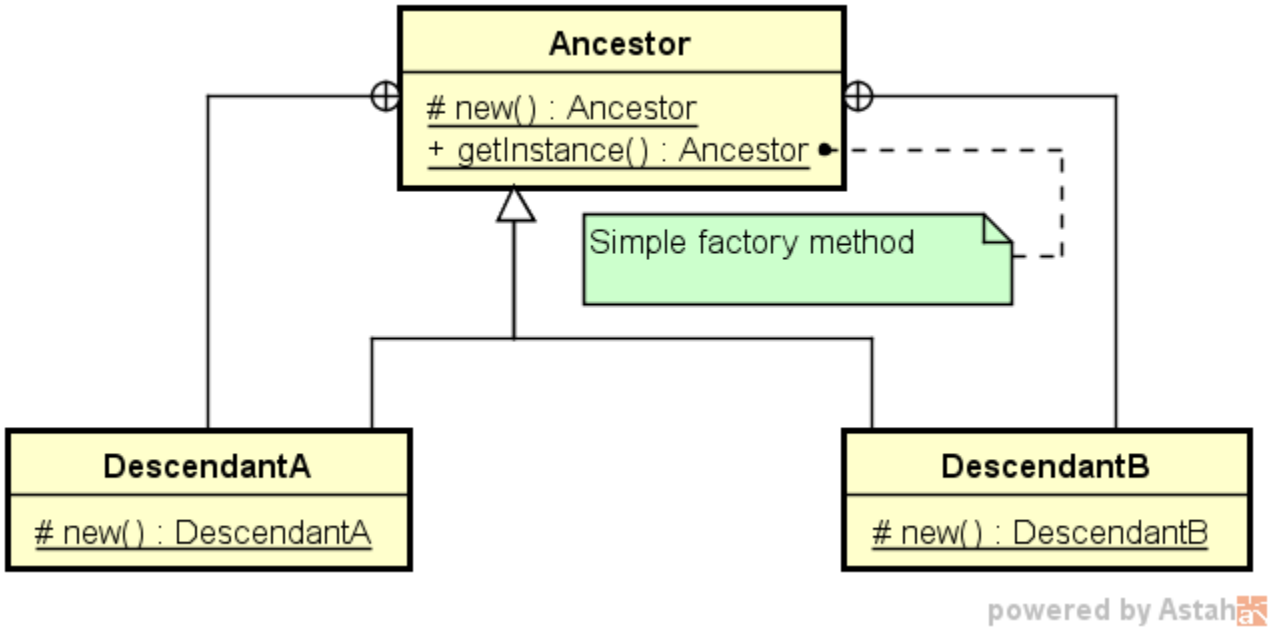
Jednoduchá továrenská metóda

Zavádza namiesto konštruktoru metódu triedy, ktorá vracia jej inštancie. Používa sa tam, kde nie je výhodné priame použitie konštruktoru.

Simple factory method

- Zjednodušenie vzoru `Factory Method`.
- Obmedzenia konštruktora:
  - **Vždy** vytvorí inštanciu – SFM môže „zrecyklovať“ existujúcu inštanciu.
  - Vracia iba inštancie **vlastnej** triedy – SFM dokáže vytvoriť aj potomkov.
  - Prvý príkaz **musí byť** volanie rodičovského konštruktora.
  - **Nesmie** používať virtuálne metódy – SFM môže po vytvorení inštancie automaticky zavolať virtuálnu metódu

Simple factory method



Lazy initialization

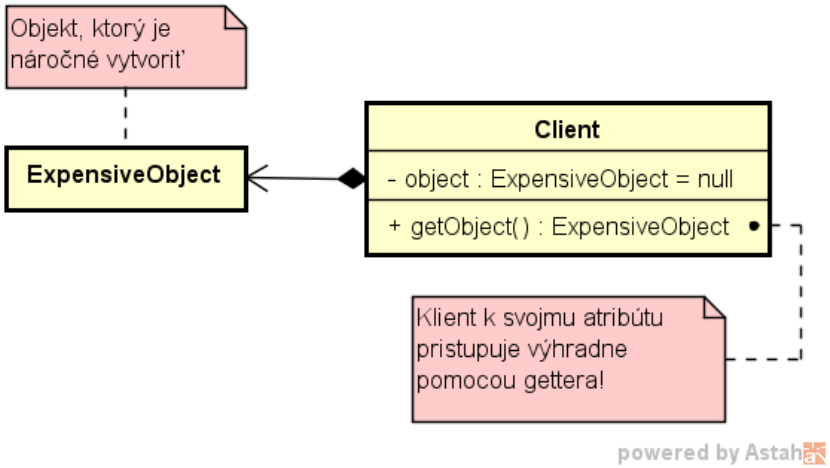
Lenivá inicializácia

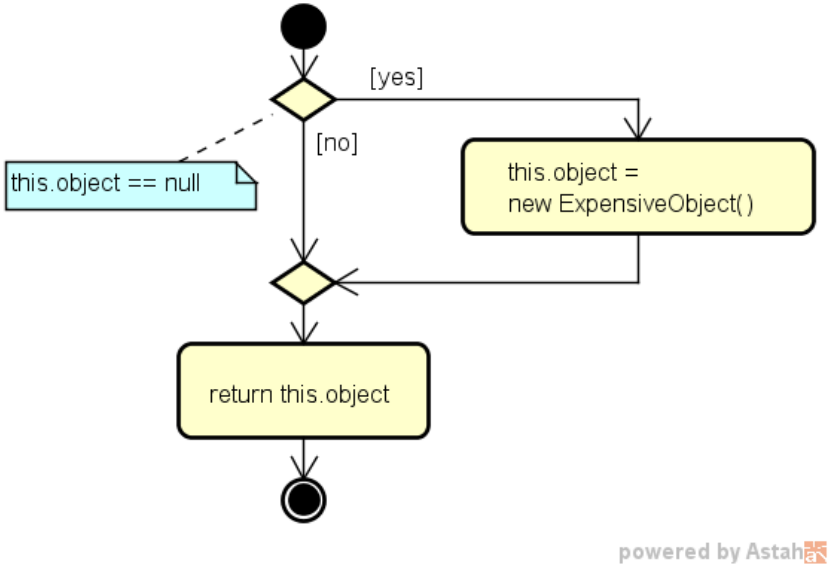
Pozdržuje výpočtovo náročné vytvorenie objektu až do doby, kedy nie je prvý krát potrebný.

Lazy initialization

- Objekt získava pomocou (jednduchej) továrenskej metódy. Referencia na vytvorenú inštanciu je následne uložená (napr. do atribútu, tabuľky, ..).
- Pomáha šetriť zdroje – ak inštanciu triedy `ExpensiveObject` nebude nikdy potrebná, tak sa nikdy nevytvorí.
- Pozor na viacvláknové aplikácie!

Lazy initialization





Library class

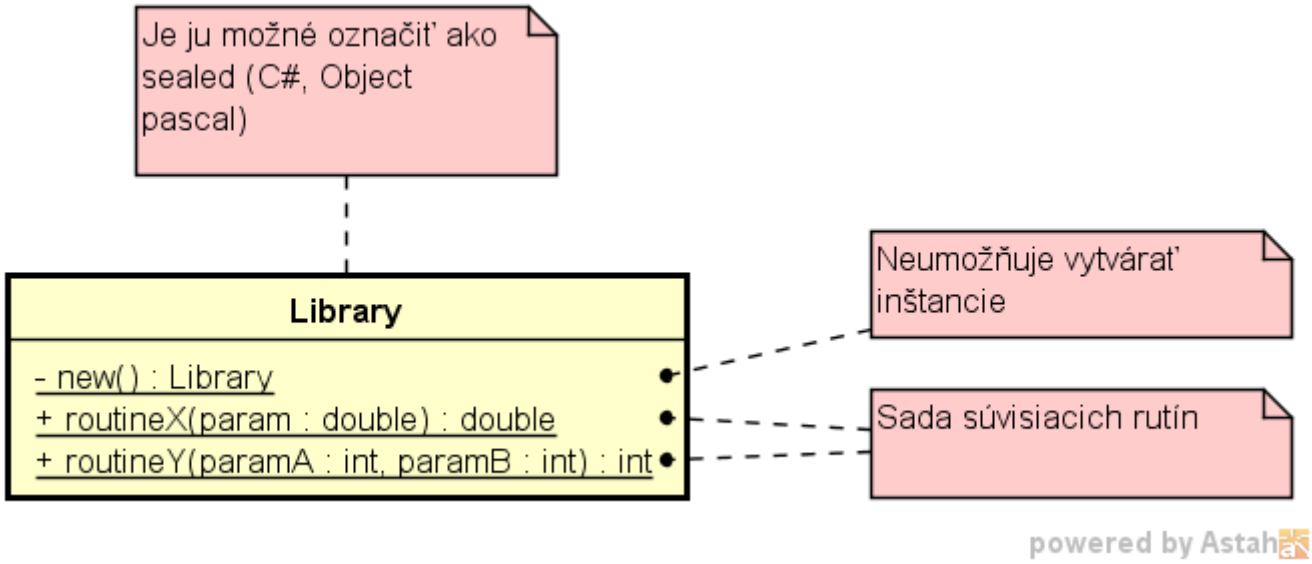
Utility, Knižničná trieda

Zjednocuje súvisiace rutiny ako statické metódy. Nepovoľuje vytváranie vlastných inštancií.

Library class

- Znemožňuje vytvárať vlastné inštancie (súkromný bezparametrický konštruktor).
- Jedna z možných implementácií návrhového vzoru Jedináčik.
- `java.lang.Math`, `java.lang.System`

Library class



Multiton

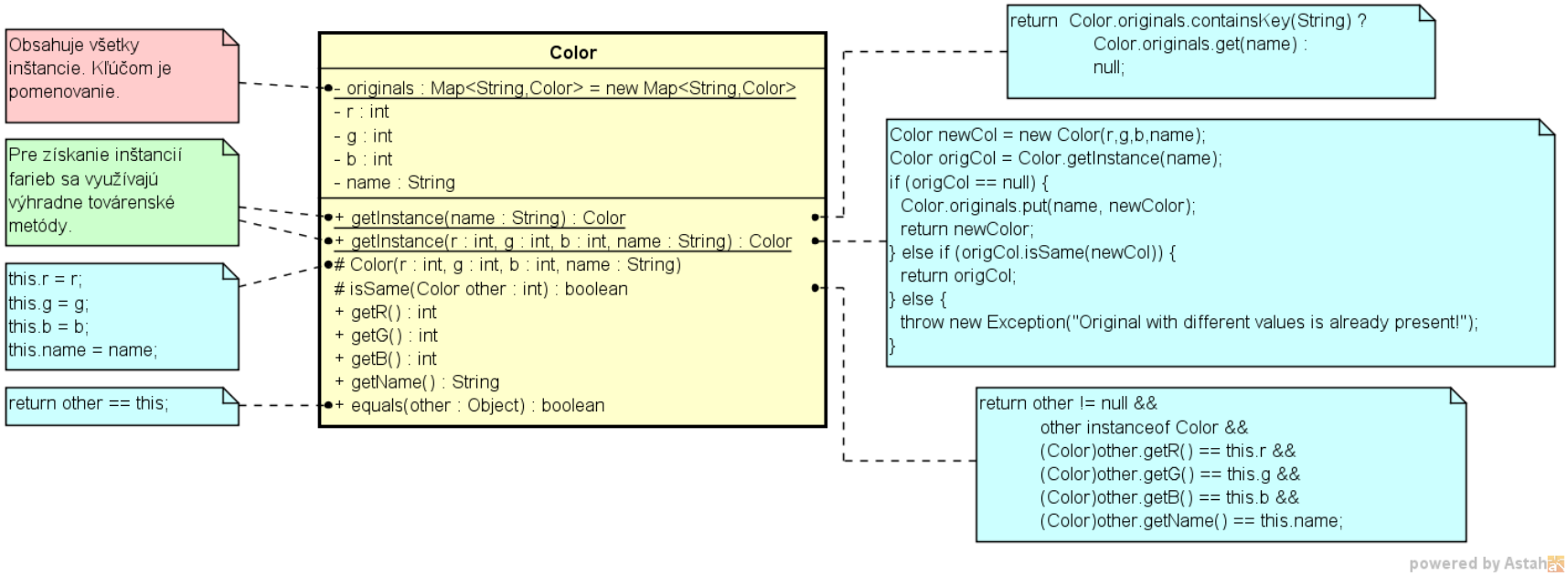
Originál

Umožňuje efektívne využívať na rôznych miestach vopred známe malé množstvo inštancií danej triedy.

Multiton

- Zovšeobecnenie jedináčika.
- Jednoduchšia verzia fondu – nevracia použité inštancie.
- Zabezpečuje, aby v aplikácií neboli dve inštancie s rovnakými hodnotami atribútov.
- Vytvárané inštancie originálu sa ukladajú do mapy – pri požiadavke na danú inštanciu sa najskôr hľadá v mape – to môže byť pomalé.
- Na druhú stranu, porovnanie objektov nie je nutné riešiť pomocou `equals`, stačí porovnať referencie.
- Pozor na serializáciu!

Multiton


Obsahuje všetky inštancie. Kľúčom je pomenovanie.
Pre získanie inštancií farieb sa využívajú výhradne továrenské metódy.
this.r = r;
this.g = g;
this.b = b;
this.name = name;
return other == this;
Color
- originals : Map<String,Color> = new Map<String,Color>
- r : int
- g : int
- b : int
- name : String
+ getInstance(name : String) : Color
+ getInstance(r : int, g : int, b : int, name : String) : Color
# Color(r : int, g : int, b : int, name : String)
# isSame(Color other : int) : boolean
+ getR() : int
+ getG() : int
+ getB() : int
+ getName() : String
+ equals(other : Object) : boolean
return Color.originals.containsKey(String) ?
Color.originals.get(name) :
null;
Color newCol = new Color(r,g,b,name);
Color origCol = Color.getInstance(name);
if (origCol == null) {
Color.originals.put(name, newColor);
return newColor;
} else if (origCol.isSame(newCol)) {
return origCol;
} else {
throw new Exception("Original with different values is already present!");
}
return other != null &&
other instanceof Color &&
(Color)other.getR() == this.r &&
(Color)other.getG() == this.g &&
(Color)other.getB() == this.b &&
(Color)other.getName() == this.name;
powered by Astah

Object pool

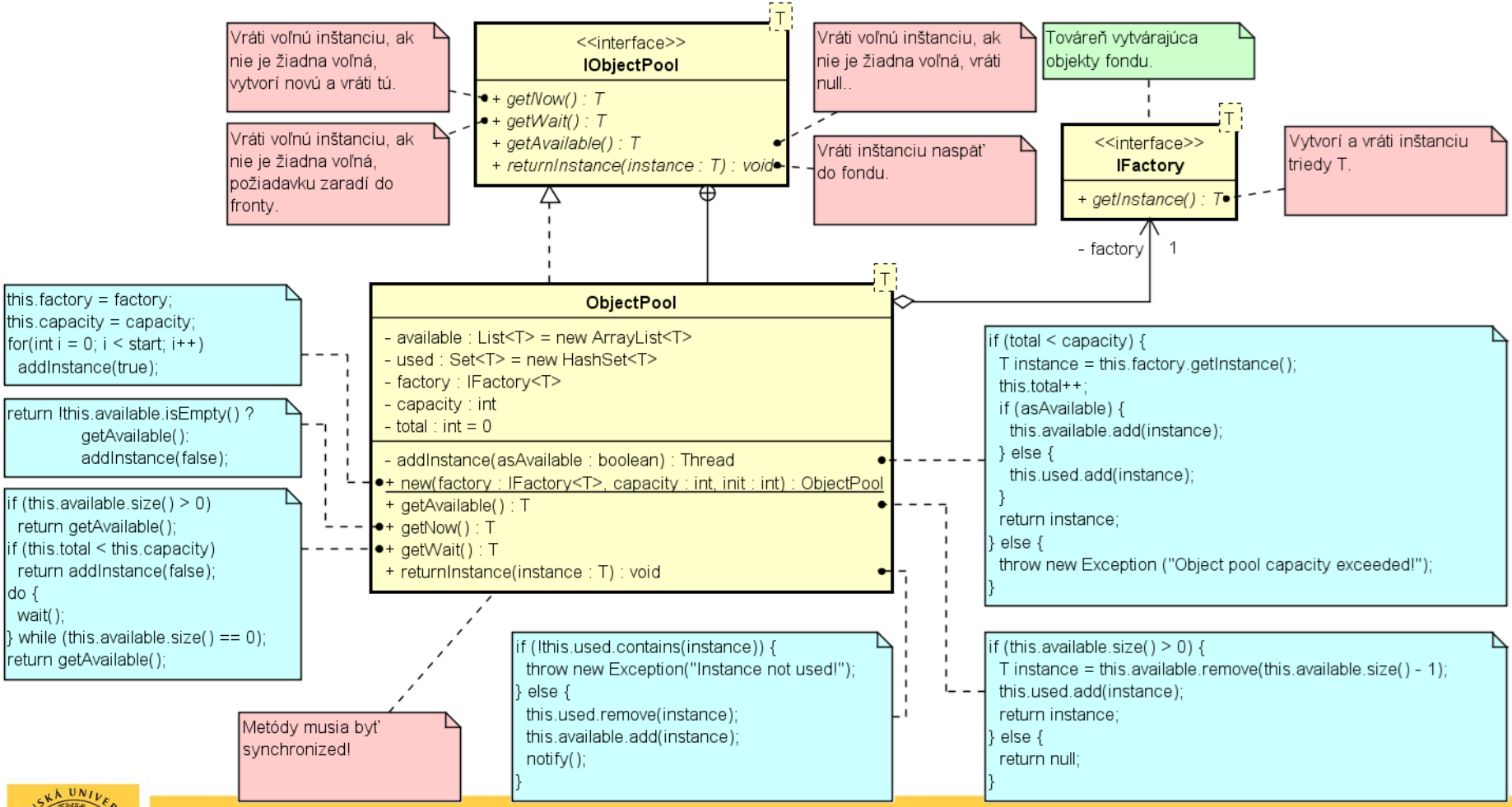
Fond

Definuje množinu vopred pripravených inštancií, ktoré sú klientom podľa potreby sprístupňované namiesto toho, aby boli zakaždým vytvárané a mazané.

Object pool

- Ak aplikácia obsahuje objekty, ktoré je náročné zostrojiť, často sa zostroja raz a následne sa na požiadanie prideľujú.
- Ak nie je vo fonde voľná inštancia:
  - tak sa požiadavka zaradí do fronty a čaká, kým sa nejaká inštancia neuvoľní (statická implementácia) alebo
  - sa automaticky vytvorí nová inštancia, ktorá požiadavku preberie (dynamická implementácia).

Object pool


Vráti voľnú inštanciu, ak nie je žiadna voľná, vytvorí novú a vráti tú.
Vráti voľnú inštanciu, ak nie je žiadna voľná, požiadavku zaradí do fronty.
<<interface>>
IObjectPool
+ getNow() : T
+ getWait() : T
+ getAvailable() : T
+ returnInstance(instance : T) : void
T
Vráti voľnú inštanciu, ak nie je žiadna voľná, vráti null..
Vráti inštanciu naspäť do fondu.
Továreň vytvárajúca objekty fondu.
<<interface>>
IFactory
+ getInstance() : T
Vytvorí a vráti inštanciu triedy T.
- factory
1
ObjectPool
- available : List<T> = new ArrayList<T>
- used : Set<T> = new HashSet<T>
- factory : IFactory<T>
- capacity : int
- total : int = 0
- addInstance(asAvailable : boolean) : Thread
+ new(factory : IFactory<T>, capacity : int, init : int) : ObjectPool
+ getAvailable() : T
+ getNow() : T
+ getWait() : T
+ returnInstance(instance : T) : void
this.factory = factory;
this.capacity = capacity;
for(int i = 0; i < start; i++)
addInstance(true);
return !this.available.isEmpty() ?
getAvailable():
addInstance(false);
if (this.available.size() > 0)
return getAvailable();
if (this.total < this.capacity)
return addInstance(false);
do {
wait();
} while (this.available.size() == 0);
return getAvailable();
if (total < capacity) {
T instance = this.factory.getInstance();
this.total++;
if (asAvailable) {
this.available.add(instance);
} else {
this.used.add(instance);
}
return instance;
} else {
throw new Exception ("Object pool capacity exceeded!");
}
if (!this.used.contains(instance)) {
throw new Exception("Instance not used!");
} else {
this.used.remove(instance);
this.available.add(instance);
notify();
}
if (this.available.size() > 0) {
T instance = this.available.remove(this.available.size() - 1);
this.used.add(instance);
return instance;
} else {
return null;
}
Metódy musia byť synchronized!

Dependency injection

Vkladanie závislostí

Zabezpečuje, aby objekt, ktorý vyžaduje nejakú funkčnosť, bol nezávislý na objekte, ktorý funkčnosť poskytuje, čím umožňuje tvoriť aplikácie s minimálnym previazaním tried.

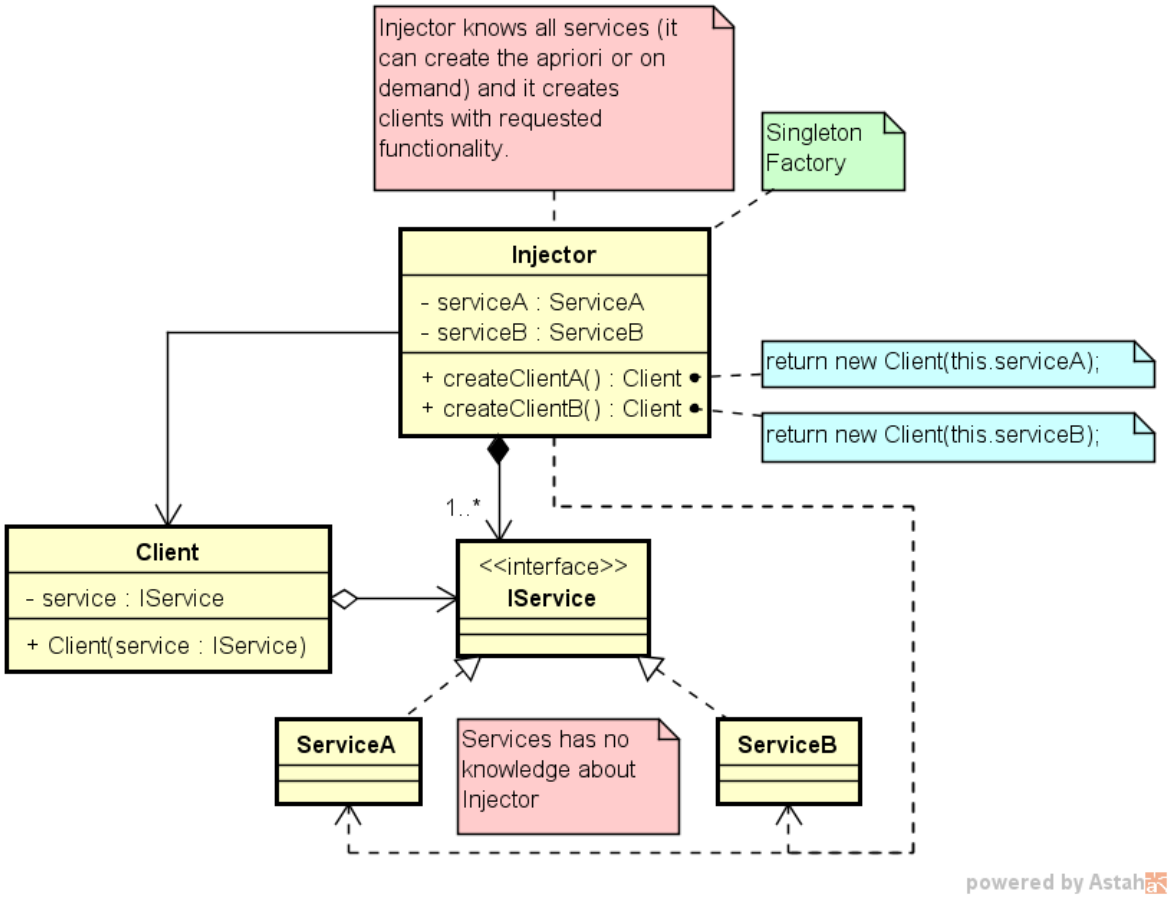
Dependency injection

- Oddeľuje proces vytvárania a používania objektov - **klient**, ktorý chce použiť **službu**, nie je zodpovedný za jej vytvorenie a pozná iba jej rozhranie.
- Základ pre nepreviazanosť triedy (loosely coupling).
- Jednoduchá tvorba modulárneho softvéru a jeho konfigurácií.
- Inversion of controll (hollywoodský princíp):
  - Template method cez dedičnosť.
  - Dependency injection cez kompozíciu.
- Podobnosť so vzorom Strategy (robí podobnú vec, ale v zmysle behaviorálnych vzorov, DI je kreačný vzor).

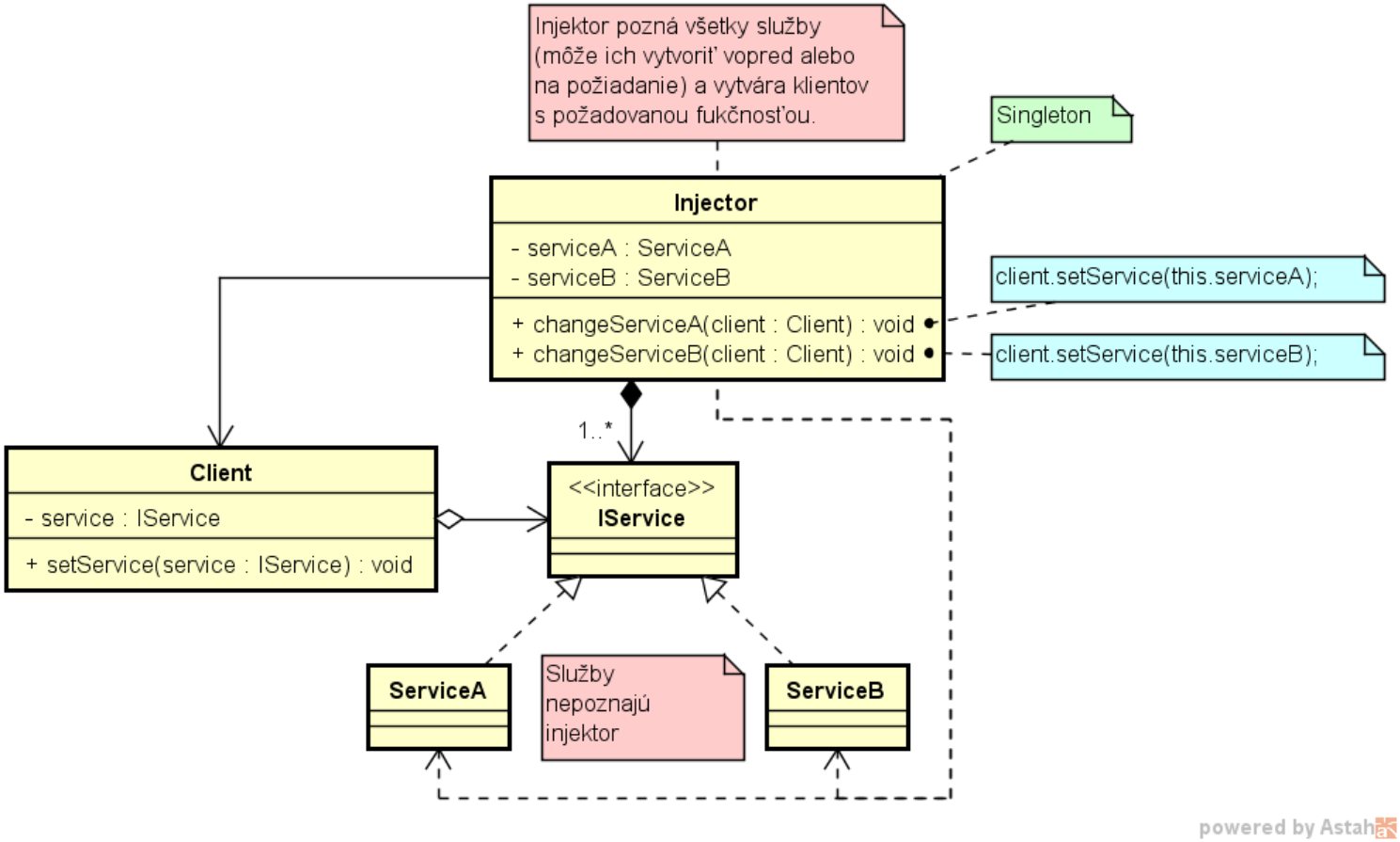
Dependency injection

- Prepojenie klienta so službou realizuje **injektor** (nesmie to byť klient), ktorý má znalosť o dostupných službách:
  - **Constructor injection** - služba je do klienta poslaná pri jeho vzniku ako parameter konštruktora.
  - **Setter injection** - služba je nastavená pomocou settera.
  - **Interface injection** - rozhranie služby poskytuje metódu, ktorú zavolá klient a služba seba vloží do klienta (služba je injektor).
- Injektor uchováva informácie o
  - službách (typicky v kontajneri, kde sa registrujú) a o
  - konfigurácií.
- Registrácia služieb do injektora môže byť
  - explicitná - injektor pozná všetky služby a vytvorí a zaregistruje ich inštancie.
  - implicitná – injektor poskytuje registračné rozhranie a služba sa do neho sama zaregistruje pri svojom vzniku.

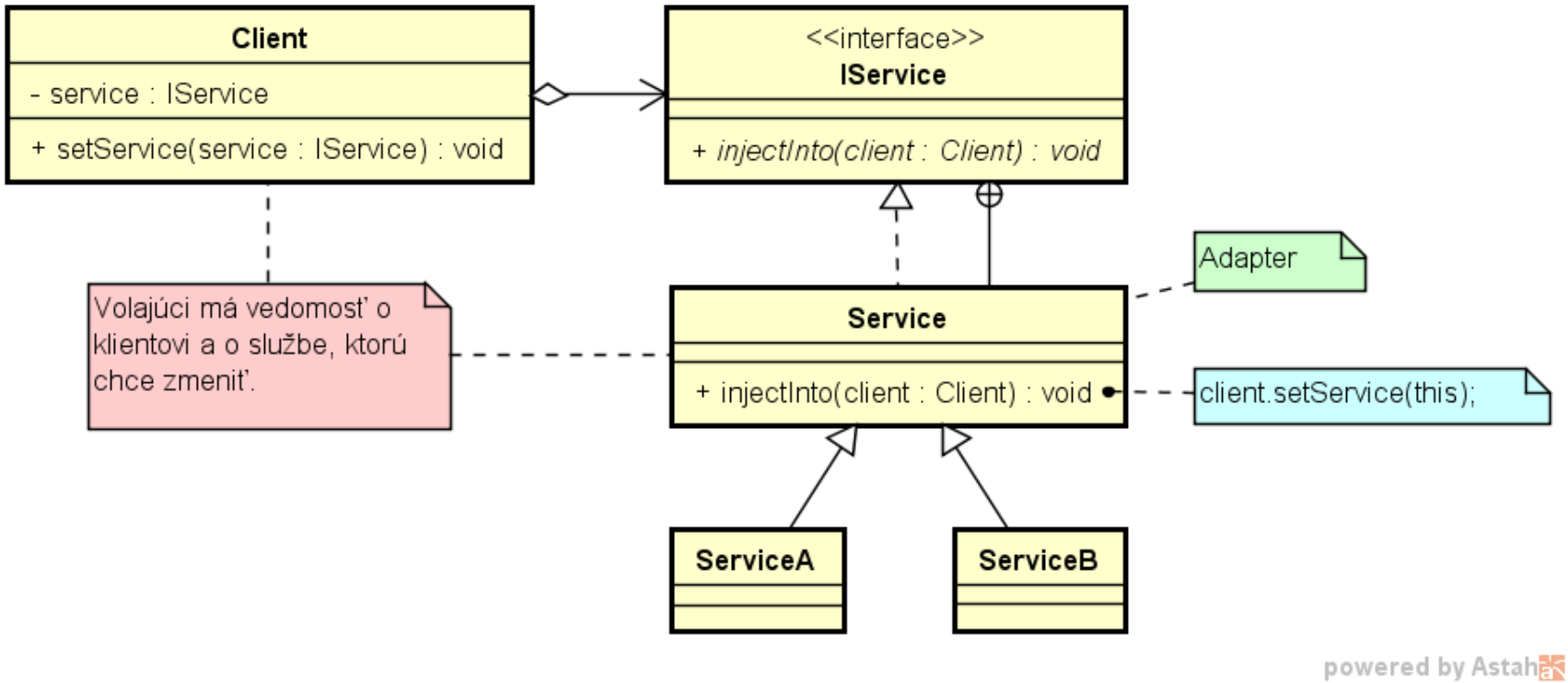
Constructor injection



Setter injection

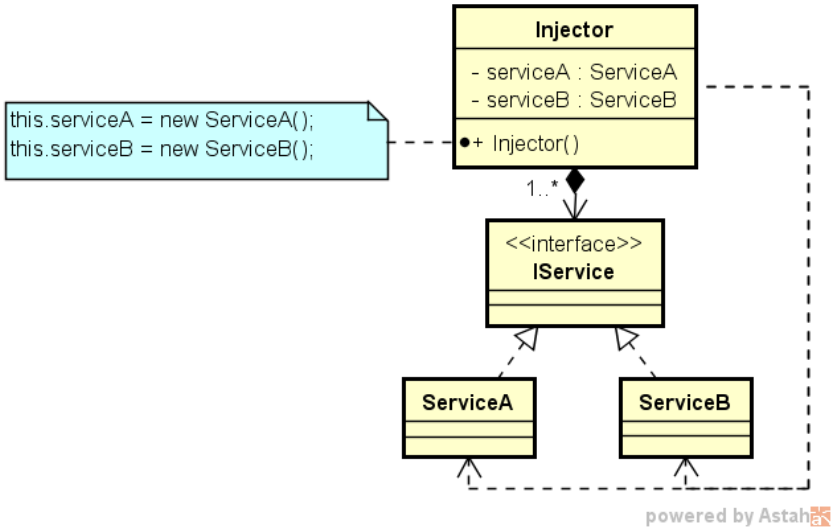


Interface injection

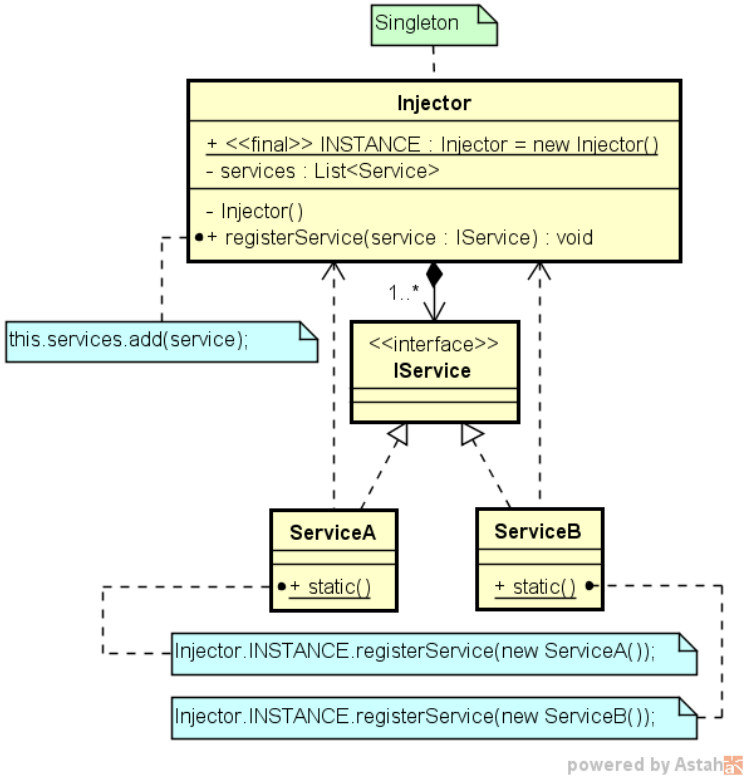


Injektor vs Služba

Injektor pozná služby



Služby poznajú injektor



Resource Acquisition is Initialization

RAII, Resource Release Is Finalization, Scope-Bound Resource Management, Pridelenie zdroja je inicializácia

Predstavuje automatický mechanizmus manažovania konca životného cyklu zdrojov (pamäť, sockety, pripojenia, súbory,..) po opustení ich rozsahu platnosti.

Resource Acquisition is Initialization

- Vznik v C++.
- Obaľuje párové funkcie vyžiadania a uvoľnenia zdroja (new/delete, alloc/free, file-open/file-close) – často sa v kóde zabúda na časť s uvoľnením.
- Zabezpečuje uvoľnenie zdroja na konci bloku (rozsahu platnosti).
- Poskytuje bezpečnosť pri výnimočných stavoch (exceptions).

Resource acquisition is initialization

```
template <class T>
public class AutoDelete {
  public:
    AutoDelete(T* ptr): ptr_(ptr)
    {}
    ~AutoDelete() {
      delete ptr_;
    }
  private:
    T* ptr_;
}
```

Resource acquisition is initialization

```
template <class T>
public class AutoRelease {
  public:
    AutoRelease(T& lock): lock_(lock){
      lock_.acquire();
    }
    ~AutoRelease() {
      lock_.release();
    }
  private:
    T& lock_;
}
```

Resource acquisition is initialization

```
void testBad() {
   X* x = new X();
   L l;
   // ak vyhodí výnimku, x sa určite neuvoľní
   x.operation();
   // ak vyhodí výnimku, l sa určite neodomkne
   l.operation();
   delete x; // čo ak zabudnem na deštruktor?
   x = null;
   l.release();
}
```

Resource acquisition is initialization

```
void testOK() {
   X* x = new X();
   L l;
   AutoDelete<X> safeDel(x);
   AutoRelease<L> safeRel(l);
   // môže vyhodiť výnimku, x sa určite uvoľní
   x.operation();
   // môže vyhodiť výnimku, l sa určite odomkne
   l.operation();
   x = null;
}
```

Structural

- Crate
- Marker
- Delegation
- Twin

Crate

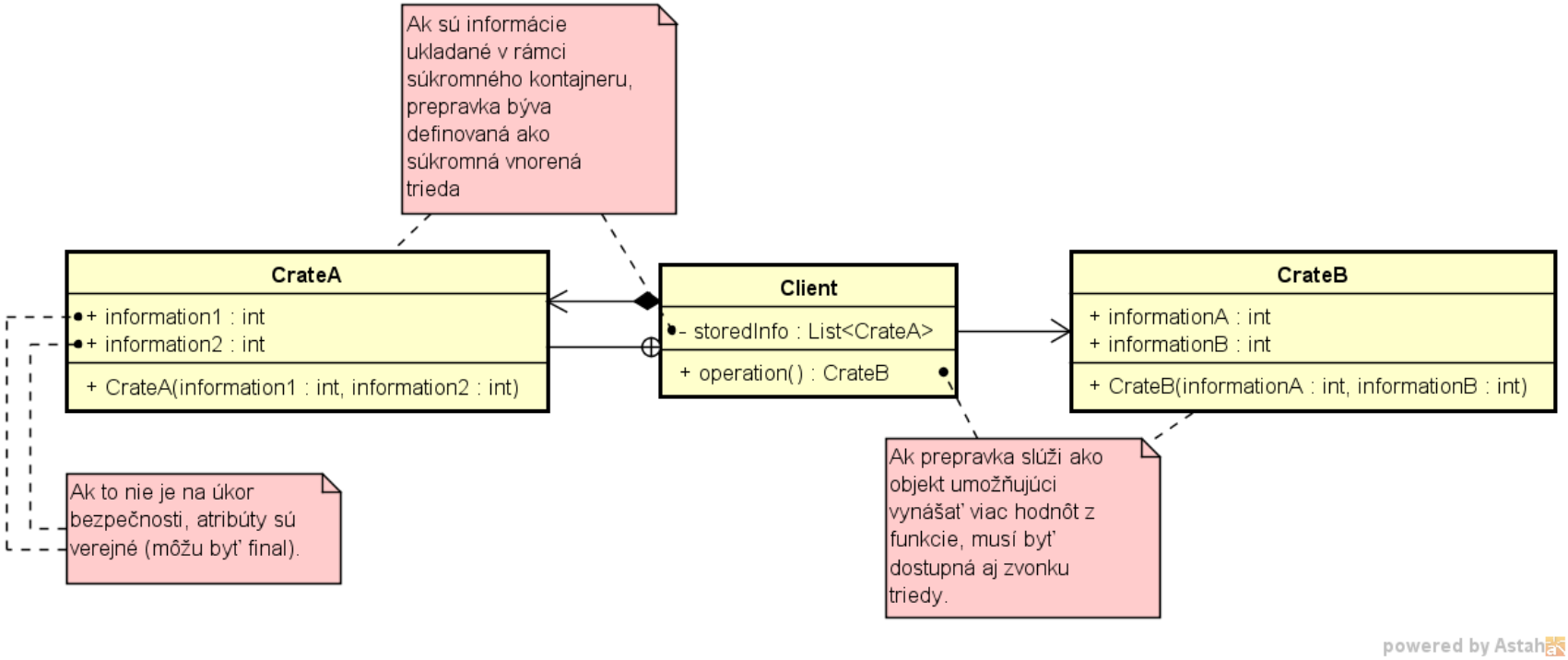
Messenger, Prepravka

Zlučuje samostatné informácie do jedného objektu, čo umožňuje ich jednoduché ukladanie a prenos.

Crate

- Každá informácia predstavuje jeden atribút.
- Definujú sa výhradne **jednoúčelové** prepravky!
- Niekedy sa rozširujú o ďalšiu funkčnosť (pribudnú metódy), potom už objekt často prestáva byť obyčajnou prepravkou!
- `java.awt.Point`, `java.awt.Dimension`

Crate


Ak sú informácie ukladané v rámci súkromného kontajneru, prepravka býva definovaná ako súkromná vnorená trieda
CrateA
+ information1 : int
+ information2 : int
+ CrateA(information1 : int, information2 : int)
Client
- storedInfo : List<CrateA>
+ operation( ) : CrateB
CrateB
+ informationA : int
+ informationB : int
+ CrateB(informationA : int, informationB : int)
Ak to nie je na úkor bezpečnosti, atribúty sú verejné (môžu byť final).
Ak prepravka slúži ako objekt umožňujúci vynášať viac hodnôt z funkcie, musí byť dostupná aj zvonku triedy.
powered by Astah

Marker

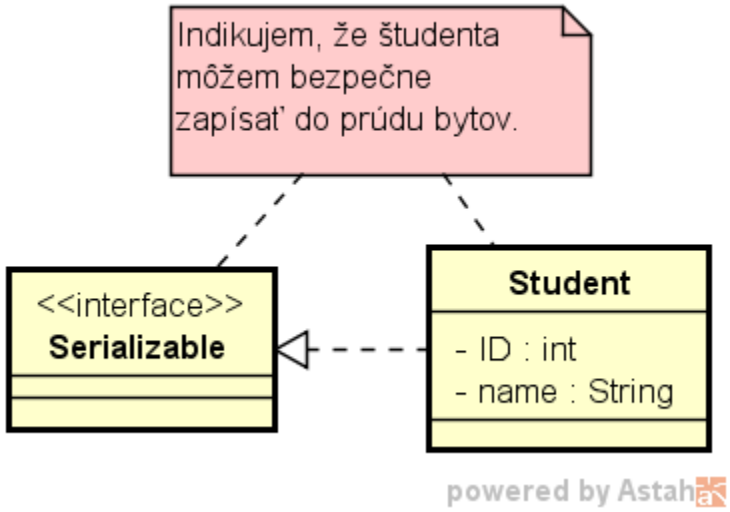
Označenie

Umožňuje pridať metainformáciu triede, ak na to jazyk nemá iné výrazové prostriedky.

Marker

- Rozhranie, ktoré nedefinuje žiadne údaje.
- Indikuje špecializované použitie triedy, ktorú označuje.
- Forma metaúdajov – pozor, potomkovia sa ho nevedia zbaviť.
- V C# môžete využiť *custom attributes*.
- Využíva ho vzor memento.
- `Serializable`

Marker


Indikujem, že študenta
môžem bezpečne
zapísať do prúdu bytov.
<<interface>>
Serializable
Student
- ID : int
- name : String
powered by Astah

Delegation

Delegát

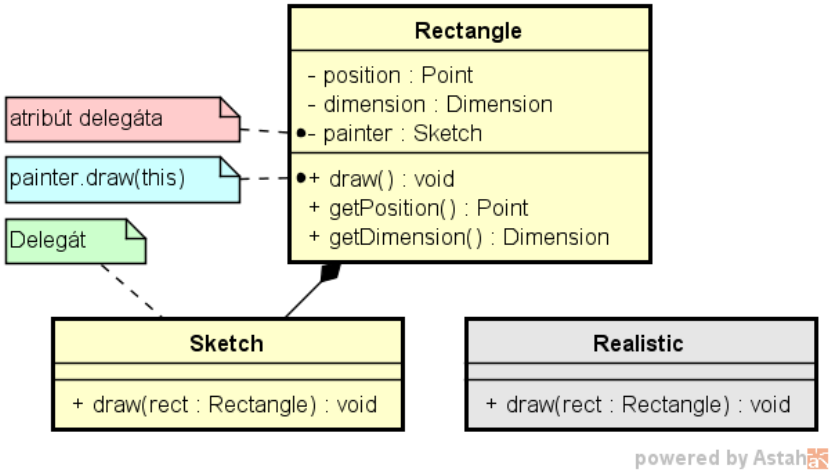
Umožňuje pomocou kompozície získať rovnakú funkčnosť a znovupoužiteľnosť kódu ako pomocou dedičnosti.

Delegation

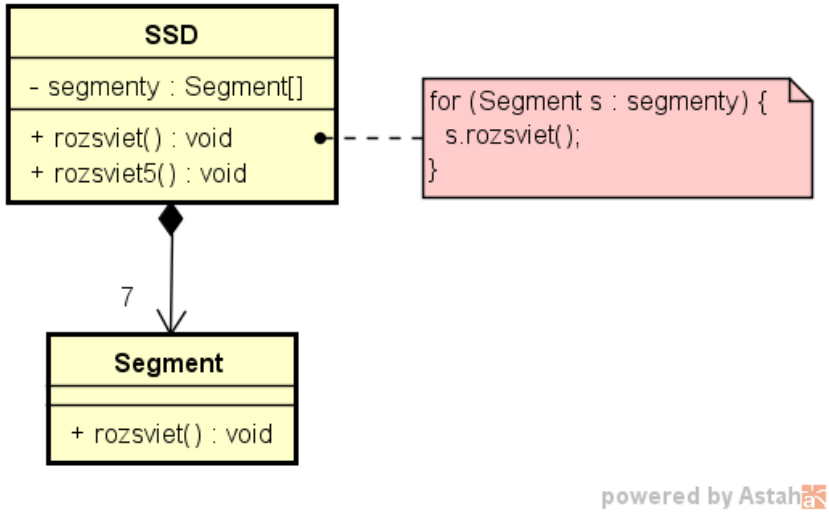
- Požiadavka nie je spracovávaná objektom, na ktorý bola smerovaná (**prijímateľom**), ale nejakým jeho komponentom, na ktorý ju objekt deleguje (**delegát**).
- Dôležité je rozlíšiť kontext!
  - Je to kontext **prijímateľa**?
    delegácia → filozoficky **dedičnosť**
  - Je to kontext **delegáta**?
    preposlanie (forwarding) → filozoficky **kompozícia**
- Niektoré programovacie jazyky vedia delegáciu vyjadriť pomocou svojich výrazových prostriedkov (`helper`, `implements`, `by`..).

Delegation

Delegácia



Preposlanie



Twin

Dvojča

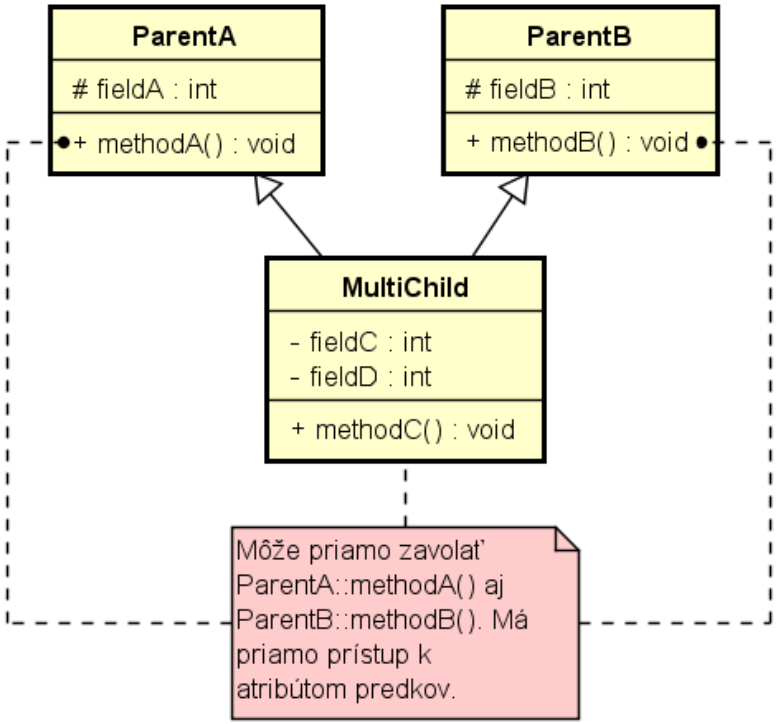
Umožňuje využiť koncept viacnásobnej dedičnosti v jazykoch, kde nie je možná viacnásobná dedičnosť a taktiež predchádzať problémom s ňou spojenou.

Twin

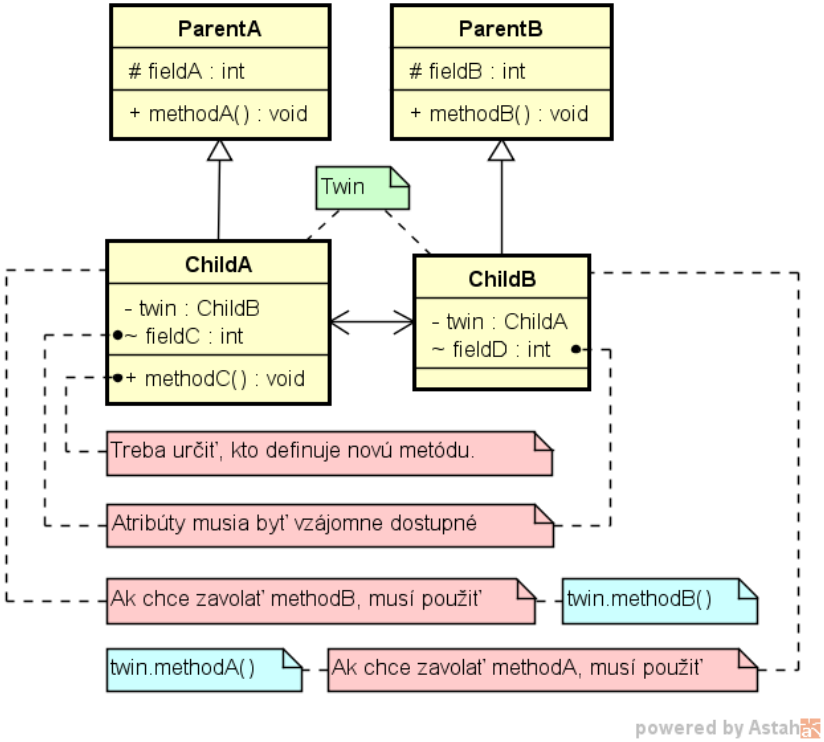
- Namiesto jednej triedy zdedenej z dvoch rodičov pozostáva z dvoch **úzko previazaných** tried, kde každá má jediného rodiča.
- Musia byť jasné zodpovednosti tried.
- Pozor na výkon – môže byť menej efektívna ako dedičnosť, lebo bude potrebné preposielať správy medzi komponentami.

Twin

Viacnásobná dedičnosť



Twin



Behavioral

- Null object
- Servant

Null object

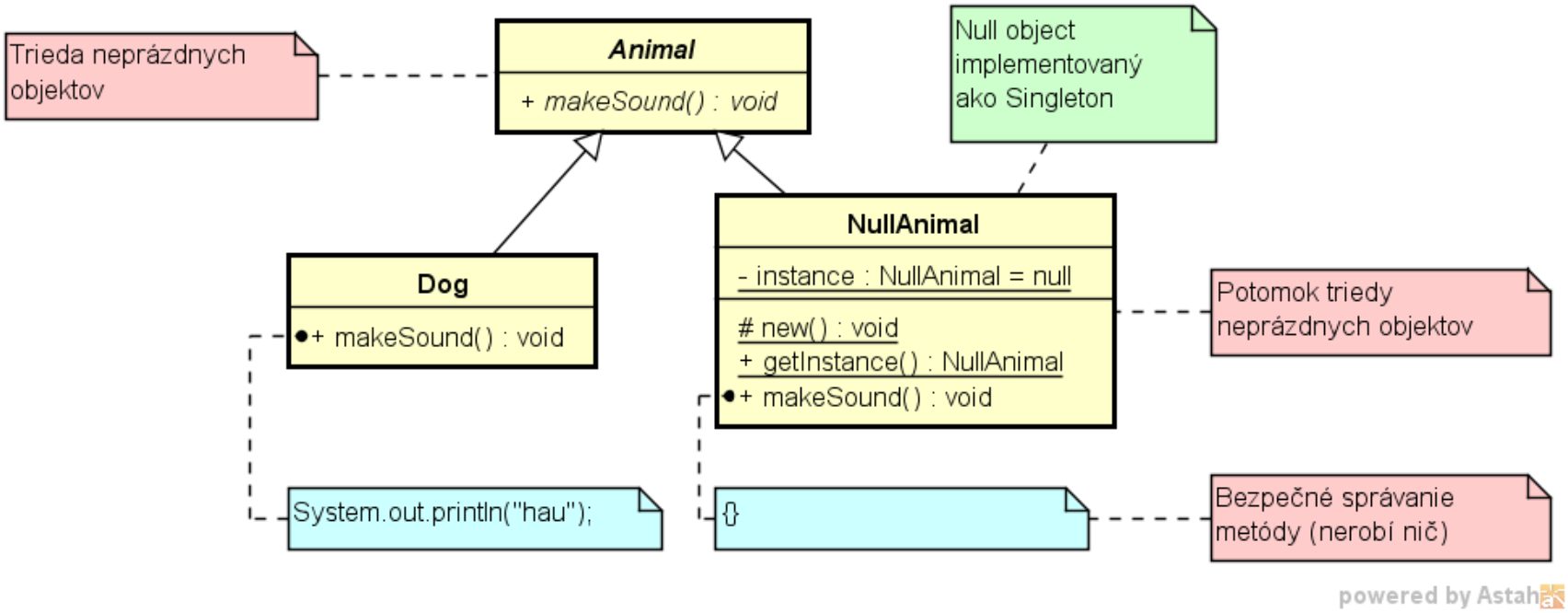
Prázdny objekt

Predstavuje skutočný objekt, ktorý nahrádza literál „null“ v situáciách, ktoré by viedli k problémom v kóde – napr. k častému testovaniu na existenciu skutočného objektu alebo k výnimkám NullPointerException.

Null object

- Prázdny objekt je definovaný ako inštancia triedy, ktorá je potomkom triedy neprázdnych objektov.
- Často je to `Jedináčik`.
- Definuje bezpečné „prázdne“ správanie (nerobí nič).
- Ak referencia ukazuje na `null objekt`, tak pri volaní metód nie je potrebné ošetrovať existenciu objektu a napriek tomu nehrozí `NullPointerException`.

Null object


Trieda neprázdnych objektov
Animal
+ makeSound() : void
Null object implementovaný ako Singleton
Dog
+ makeSound( ) : void
NullAnimal
- instance : NullAnimal = null
new( ) : void
+ getInstance( ) : NullAnimal
+ makeSound( ) : void
Potomok triedy neprázdnych objektov
System.out.println("hau");
{}
Bezpečné správanie metódy (nerobí nič)
powered by Astah

Null object

Bez využitia null object

```
// volanie metódy je vždy
// potrebné ošetriť
if (animal != null) {
  animal.makeSound();
}

// priradenie null
animal = null;

// test na null
animal == null;
```

S využitím null object

```
// volanie metódy je vždy
// bezpečné
animal.makeSound();

// priradenie null
animal = NullAnimal.getInstance();

// test na null
animal == NullAnimal.getInstance();
```

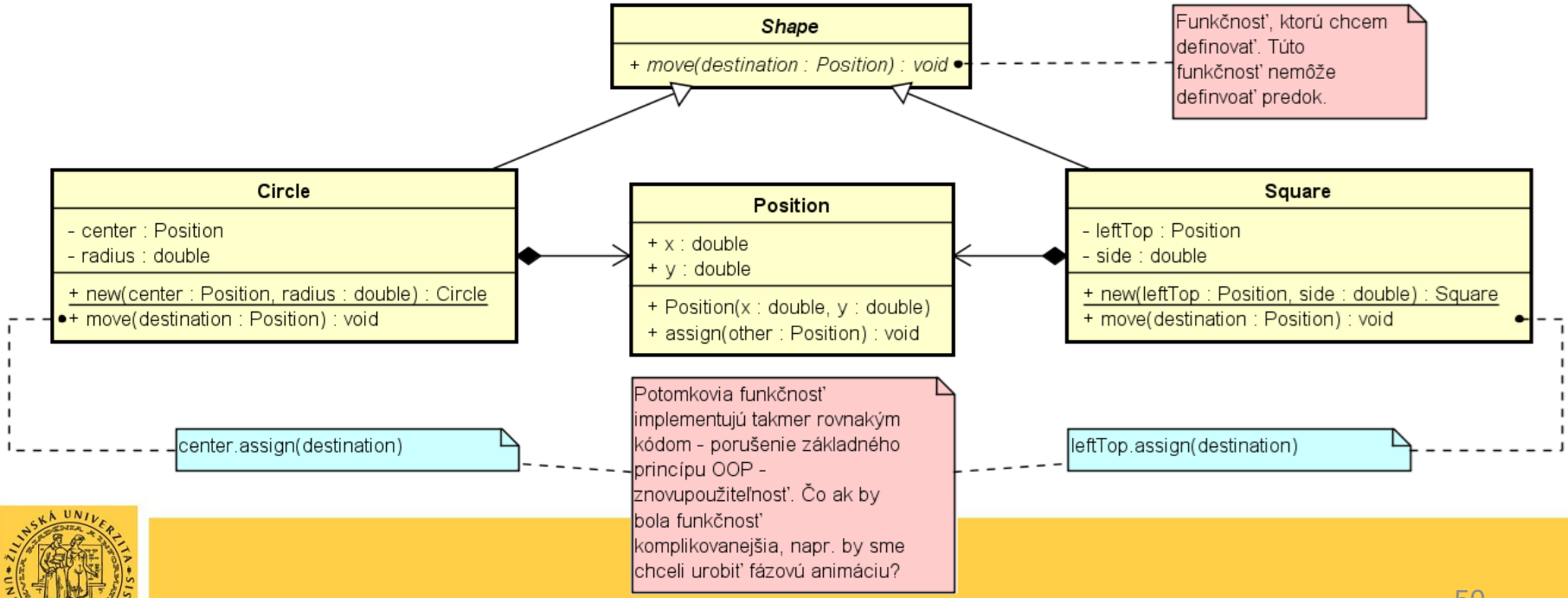
Servant

Služobník

Predstavuje triedu, ktorej inštancie (prípadne ona sama) poskytujú metódy zabezpečujúce špecifickú službu. Objekty, ktoré chcú túto službu využívať ju potom využívajú prostredníctvom Služobníka - nemusia funkčnosť implementovať sami.

Servant - motivácia

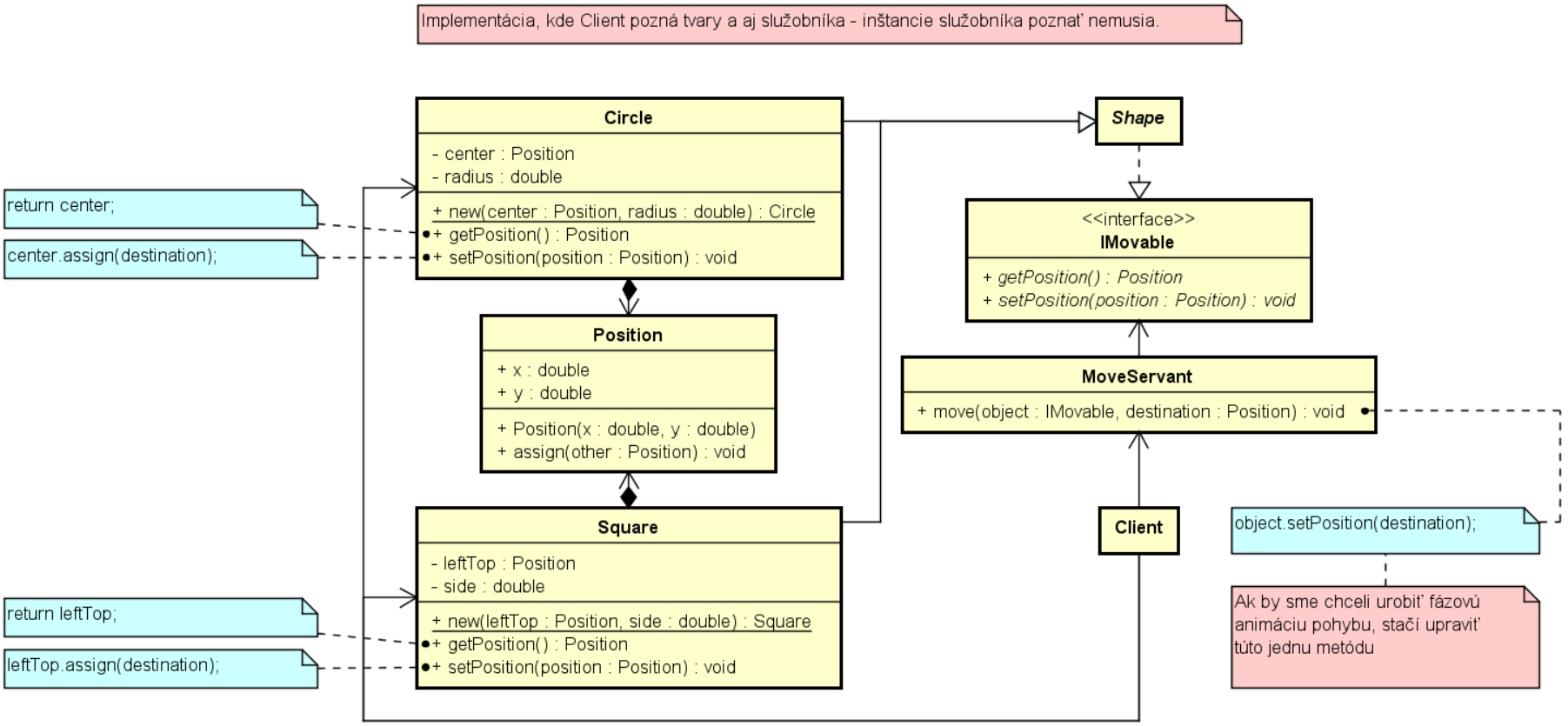
- Majme projekt s geometrickými tvarmi.
- Do tvarov chceme pridať novú funkcionalitu – ako to urobiť pružne?



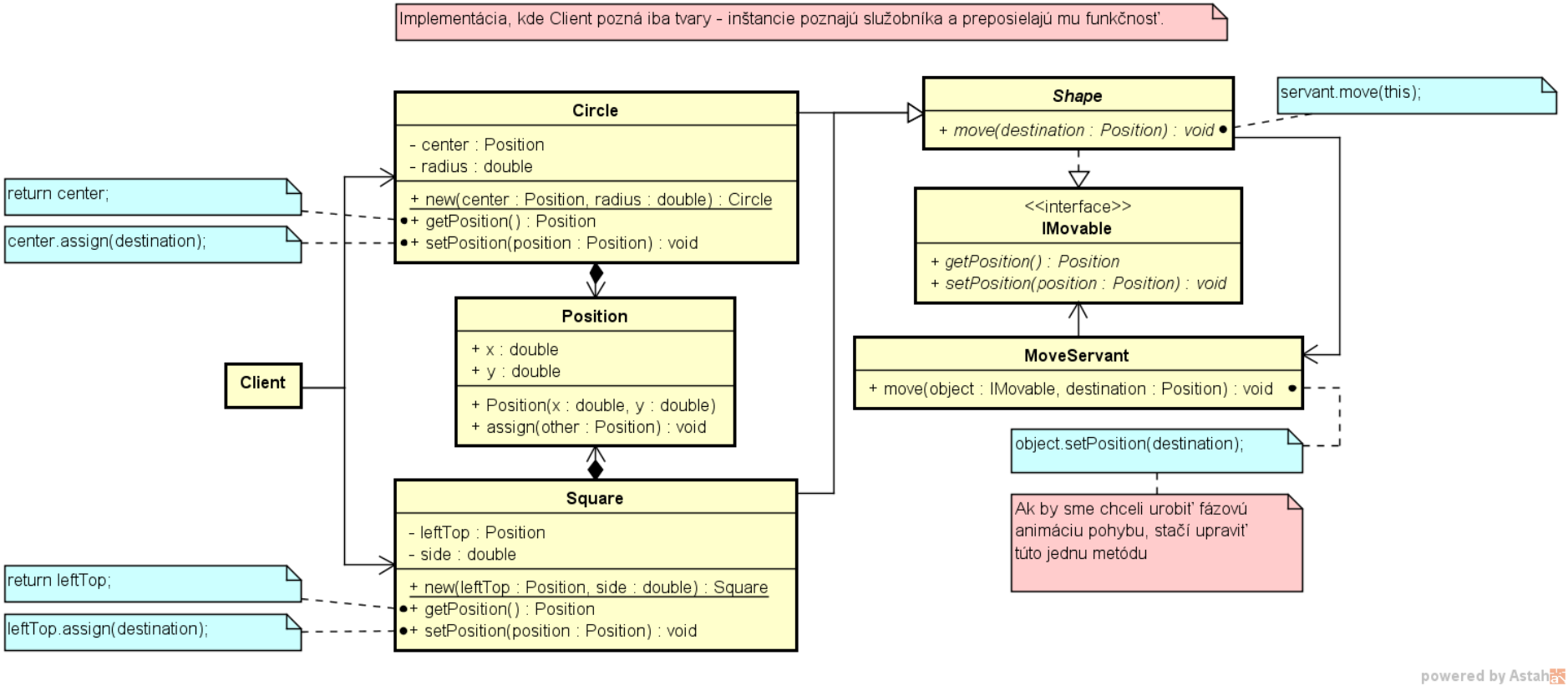
Servant - implementácia

1. Najskôr je potrebné zistiť, čo má mať služobník na starosti.
2. Potom sa definuje `interface`, ktoré obsahuje všetky metódy, ktoré bude služobník pre svoju funkčnosť vyžadovať.
3. Vytvorí sa služobník (nová trieda, rozšírenie existujúcej triedy), ktorý definuje príslušnú funkčnosť nad vytvoreným rozhraním.
4. Nakoniec všetky triedy, ktoré chcú využívať službu, musia implementovať rozhranie definované v druhom kroku.

Servant – implementácia1



Servant – implementácia2



Servant vs Command

- ***Servant*** – máme objekty, ktorým chceme ponúknuť funkčnosť. Vytvorí sa
  - trieda (služobník), ktorá poskytuje službu a
  - rozhranie, ktoré musia obsluhované objekty implementovať.

Obsluhované objekty sú potom posielané služobníkovi, spravidla ako parametre jeho metódy.

- ***Command*** – máme objekty, ktorých funkčnosť má byť modifikovaná. Vytvorí sa
  - rozhranie predpisujúce danú funkčnosť (daný príkaz), a ďalej sa vytvoria
  - objekty (príkazy), ktoré dané rozhranie implementujú.

Príkazy sa potom posielajú objektom ako parametre ich metód.

Upozornenie

- Tieto študijné materiály sú určené výhradne pre študentov predmetu 5II132 Návrhové vzory (Design Patterns) na Fakulte riadenia a informatiky Žilinskej univerzity v Žiline.
- Reprodukovanie, šírenie (i častí) materiálov bez písomného súhlasu autora nie je dovolené.

Ing. Michal Varga, PhD.
Katedra informatiky
Fakulta riadenia a informatiky
Žilinská univerzita v Žiline
Michal.Varga@fri.uniza.sk