Objektově orientované programování

Filip Rosa

2025

Obsah

1 Modularita

1.1 Co je hlavním motivem pro vývoj programovacího paradigmatu od imperativního k objektovému?

Hlavným motívom pre prechod od imperatívneho programovania k objektovo orientovanému programovaniu je zjednodušenie riadenia komplexity pri vývoji a údržbe veľkých softvérových systémov. Tento prechod priniesol niekoľko kľúčových výhod, ktoré sa týkali najmä organizácie kódu, opakovateľnosti, rozšíriteľnosti a správy stavu aplikácií.

1.2 Co je imperativní programování?

Poznáme ho z bežne používaných jazykov. Môžeme ho rozdeliť na dva druhy:

- **procedurálne programovanie** - postupnosť krokov, ktorými meníme stav premenných programu
- **štruktúrované programovanie** - sekvencie, iterácie, vetvenie, skoky, abstrakcia

1.3 Co je modulární programování?

Je to návrh zhora-dolu. Rozdeľuje program na nezávislé, zameniteľné moduly, ktoré zaisťujú jednotlivé drobné funkčnosti. Modul obsahuje všetko potrebné pre zaistenie funkčnosti (dáta a algoritmy).

1.4 Jaké jsou hlavní faktory kvality software?

- vnútorné sú schované pred používateľom (AKO)
- vonkajšie popisujú správanie navonok - správnosť, robustnosť, rýchlosť, rozšíriteľnosť... (ČO)
- musíme byť schopní merať kvalitu software

1.5 Co je pochopitelnost modulu? Uveďte příklad.

Modul by mal vykonávať jednu jasne definovanú a pochopiteľnú úlohu, prípadne niekoľko málo jasne definovaných úloh.
Napr. modul pre prácu s dátumami funkciou is_leap_year jednoducho zisťuje, či je rok priestupný, a days_in_month vracia počet dní v danom mesiaci.

1.6 Co je samostatnost modulu? Uveďte příklad.

Každý modul musí byť relatívne samostatný a mal by mať čo najmenej väzieb na ostatné moduly. Nebolo by vhodné, aby všetky moduly programu boli navzájom prepojené a na sebe závislé.
Napr. modul pre základné operácie s textom. Vykonáva dve nezávislé operácie: konverziu textu na veľké písmená a počítanie samohlások. Nevyžaduje žiadne externé knižnice alebo konfigurácie. Môže byť použitý samostatne v akomkoľvek projekte.

1.7 Co je kombinovatelnost modulu? Uveďte příklad.

Moduly musia byť navzájom kombinovateľné. Musí byť možné modul vziať a použiť v inom kontexte alebo v inom projekte.
Napr. máme dva moduly: jeden na spracovanie textu a druhý na spracovanie čísiel. Oba moduly budú kombinovateľné v rámci väčšej aplikácie, kde budú spolupracovať. Každý z nich vykonáva jasnú úlohu a má jednoduché rozhranie.

1.8 Co je zapouzdření modulu? Uveďte příklad.

Moduly musia mať právo na isté súkromie: je prípustné a žiadúce, aby všetky informácie, ktoré nie sú potrebné pre klientov modulu zostaly skryté vnútri modulu. Väčšina funkcionality modulu je skrytá a len malá časť je viditeľná zvonku. Skrytej časti vravíme **implementácia** modulu a verejnej časti **rozhranie** modulu.
Napr. Máme triedu Car, ktorá reprezentuje automobil. Tento automobil má súkromné údaje, ako je aktuálny počet paliva v nádrži. Chceme, aby tieto údaje boli prístupné iba prostredníctvom metód triedy a neboli priamo meniteľné z vonkajšieho kódu.

1.9 Co je explicitní rozhraní modulu? Uveďte příklad.

Z deklarácie modulu musí byť všeobecne zrejmé, aké predpoklady pre vyokonávanie svojej úlohy potrebuje.
Napr. Vytvárame modul na spracovanie jednoduchých matematických operácií. Tento modul bude mať jasne definované funkcie pre sčítať, odčítať, vynásobiť a vydeliť čísla, ktoré tvoria jeho explicitné rozhranie.

1.10 Co je syntaktická podpora modularity?

Moduly počítačového programu musia byť jasne vymedzené syntaktickými jednotkami programu. Zo zápisu programovacieho jazyka musí byť zrejmé, kde končí a začína zápis modulu.

1.11 Co je pět kritérií pro dobrou modularitu?

- dekomponovateľnosť
- kombinovateľnosť
- pochopiteľnosť
- kontinuita
- ochrana

1.12 Co se rozumí pěti pravidly zajišťující dobrou modularitu?

- priame mapovanie
- pár rozhraní
- malé rozhrania

- explicitné rozhrania
- skrývanie informácií

1.13 Popište jednotlivá kritéria dobré modularity. Uveďte příklady.

- dekomponovateľnosť - schopnosť rozložiť veľký a komplexný systém na menšie, samostatné komponenty. Predstavme si, že vyvíjame webovú aplikáciu pre správu objednávok. Celkový systém môžeme rozdeliť na menšie moduly, ako sú: odul pre správu používateľov: Registrácia, prihlásenie, správa profilov. Modul pre spracovanie objednávok: Vytváranie objednávok, platby, správa stavu objednávky. Modul pre spracovanie platieb.
- kombinovateľnosť - schopnosť spájať rôzne moduly alebo komponenty do väčších celkov, ktoré spolupracujú, aby vykonávali zložitejšie úlohy. Zoberme si modul na spracovanie platieb v online obchode. Tento modul môže byť kombinovaný s rôznymi ďalšími modulmi ako: Modul pre spracovanie objednávok: Keď používateľ vytvorí objednávku, modul pre spracovanie objednávok využije modul na spracovanie platieb na dokončenie transakcie. Modul pre notifikácie.
- pochopiteľnosť - systém alebo modul je ľahko čitateľný a zrozumiteľný pre vývojárov alebo iných používateľov systému - premenné sú popisné, názov funkcie je jasný, kód je čitateľný
- kontinuita - systém alebo modul by mal fungovať bez prerušenia aj pri zmenách alebo aktualizáciách. Ak aktualizujeme databázový systém, musíme zabezpečiť, že staršie verzie systému budú fungovať aj po implementácii novej verzie.
- ochrana - zabezpečenie systému pred neautorizovaným prístupom, chybami alebo nepredvídanými situáciami. - autentifikácia, ochrana údajov, validácia vstupov

1.14 Popište jednotlivá pravidla pro dobrou modularitu. Uveďte příklady.

- priame mapovanie - existuje jednoznačný a priamy vzťah medzi internými komponentmi alebo dátovými štruktúrami a rozhraním modulu. Predstavme si triedu na reprezentovanie bankového účtu. Ak by sme použili priame mapovanie, poskytli by sme metódy, ktoré priamo manipulujú s internými hodnotami (ako je zostatok).
- pár rozhraní - modul poskytuje dve alebo viac rozhraní, ktoré sa vzájomne dopĺňajú a používajú sa spoločne. Predstavme si systém na správu súborov, ktorý poskytuje rozhranie na čítanie a zapisovanie do súborov, ale každé z týchto rozhraní môže byť prispôsobené pre rôznych používateľov (napr. pre čitateľov a zapisovačov).
- malé rozhrania - rozhranie modulu je minimalizované na čo najmenší počet metód alebo funkcií, ktoré vykonávajú jasne definovanú a konkrétnu úlohu. Predstavme si modul na validáciu používateľských údajov. Tento modul môže mať malé a špecifické rozhranie. Modul poskytuje len dve metódy, ktoré vykonávajú veľmi špecifické úlohy – validáciu e-mailu a validáciu hesla.

- explicitné rozhrania - rozhranie modulu je jasne definované a dokumentované. Rozhranie modulu je explicitné, pretože používateľ má jasne definované metódy na pridávanie položiek a získanie celkovej ceny. Modifikácia interných dát (napríklad priamy prístup k zoznamu položiek) nie je povolená.
- skrývanie informácií - interné detaily implementácie sú skryté pred používateľmi modulu. redstavme si triedu na správu bankového účtu, ktorá skrýva interný stav (ako je zostatok) a poskytuje len verejné metódy na interakciu.

1.15 K čemu je konstruktor? Uveďte příklad.

Konštruktor inicializuje dáta objektu hodnotami parametrov v konštruktori (naplní pamät dátami).

1.16 K čemu je destruktor, kdy ho potřebujeme a kdy ne? Uveďte příklad.

Deštruktor odstráni v pamäti dáta objektu (čistí pamäť). Nie je potrebný pokiaľ sú dáta objektu statické.

2 Třídy a objekty

2.1 Co je hlavními příčinami potřeby změn software?

1. **Technologický pokrok**
2. **Zmeny v užívateľkých požiadavkách**
3. **Bezpečnosť**
4. **Chyby a problémy**
5. **Zmeny v legislatíve**
6. **Zastaralosť softwaru**
7. **Zlepšenie užívateľského zážitku**
8. **Prechod na cloud alebo inú architektúru**
9. **Konkurenčný tlak**

2.2 Jaké jsou hlavní faktory ovlivňující objektovou orientovanost?

- metóda a jazyk
- implementácia a prostredie
- knižnice

2.3 Vysvětlete, co rozumíme pojmy objektově orientovaná metoda (přístup) a jazyk.

Nejde len o programovací jazyk a spôsob jeho použitia, ide aj o spôsob uvažovania a vyjadrovania a taktiež o záznamy v textovej alebo grafickej forme.

- trieda
- trieda ako modul
- trieda ako typ
- zasielanie správ
- skrývanie informácií
- statická kontrola typov
- dedičnosť, redefinícia, polymorfizmus, dynamická väzba
- generickosť
- správa pamäti a garbage collector

2.4 Vysvětlete, co rozumíme podporou objektově orientované implementace

Je to podpora vývoja. Zahŕňa vlastnosti a efektivitu nástrojov pre vývoj, nástroje pre podporu nasadenia nových verzií a nástroje pre podporu dokumentovania.

2.5 Vysvětlete, co rozumíme podporou opakované použitelnosti.

Znemená to navrhovanie a vytváranie komponént, modulov, kódu alebo systému takým spôsobom, aby boli ľahko použiteľné v rôznych kontextoch alebo projektoch bez nutnosti ich opätovného vytvárania alebo prepisovania.

Príklady:

- Knižnice a frameworky
- Kódové šablóny
- Cloudové služby a API

2.6 Vysvětlete pojmy třída a objekt a použijte správnou terminologii.

Trieda je časť software, ktorá popisuje abstraktný dátový typ a jeho implementáciu.
Objekt je abstraktný dátový typ so spoločným chovaním reprezentovaným zoznamom operácií, ktoré vie objekt vykonávať.

2.7 Zdůrazněte vlastnosti třídy z pohledu modularity.

Triedy nepopisujú len typy objektov, ale musia byť zároveň modulárnymi jednotkami. V čisto OOP by nemali byť iné samostatné jednotky než triedy.

2.8 Vysvětlete princip zapouzdření v OOP.

Zapúzdrenie (enkapsulácia) obmedzuje prístup k určitým častiam objektu a poskytuje kontrolu nad tým, ako sú dáta a funkcie v objekte využívané.
Cieľom je ochrana dát a zjednodušenie správy komplexivity. Pre interakciu s dátami objektu a jeho ovládanie sa používajú metódy **Getter** a **Setter**.

2.9 Vysvětlete princip zasílání zpráv.

Tento princíp slúži na komunikáciu medzi objektami aplikácie. Namiesto toho, aby objekty priamo pristupovali k internému stavu iných objektov alebo priamo volali ich metódy, komunikujú medzi sebou prostredníctvom správ.
Správy predstavujú požiadavky na vykonanie nejakej akcie, ktorú objekt vykonaná na základe prijatej správy. Tieto správy môžu byť volania metód, žiadosti o zmenu stavu objektu alebo žiadosť o dáta.

2.10 Vysvětlete principy deklarace a definice jednoduché třídy v C++.

Deklarácia je proces, pri ktorom definujeme štruktúru triedy, teda aké vlastnosti a chovanie trieda bude mať. class MyClass public: // Specifikuje, že členy jsou veřejné

(přístupné zvenčí) MyClass(); // Konstruktor void display(); // Veřejná metoda private: // Specifikuje, že členy jsou soukromé (nepřístupné zvenčí) int x; // Soukromý atribut ;
Definícia je proces, pri ktorom konkrétne určujeme ako budú metódy triedy implementované.

3 Návrh programu I

3.1 Vysvětlete, jak vznikají objekty třídy, pojem konstruktor a principy práce s ním v C++.

Objekt je inštanciou triedy. Je reprezentáciou nejakej entity, ktorá má stav reprezentovaný dátami a chovanie reprezentované metódami.
Vzniká konštruktorom, ktorý inicializuje objekty. Nemá návratovú hodnotu, pokiaľ nie je deklarovaný, vytvorí sa automaticky. Pri statickej deklarácii sa volá automaticky, inak je nutné použiť **new**. Môže ich byť viac, ale musia sa líšiť počtom alebo typom parametrov.

3.2 Vysvětlete, jak zanikají objekty třídy, pojem destruktor a principy práce s ním v C++.

Objekty zanikajú deštruktorom. Ten slúži pre dealokovanie dynamicky vytvorenej pamäti. Nemá návratovú hodnotu. Pokiaľ nie je definovaný, vytvorí sa automaticky. Pri statickej deklarácii sa volá automaticky, inak s použitím **delete**.

3.3 Vysvětlete rozdíl mezi statickou a dynamickou deklarací objektů v C++.

Statická deklarácia: objekt je na zásobníku, jeho životnosť je automatická a riadená blokom, v ktorom bol vytvorený.
Dynamická deklarácia: objekt je na halde, jeho životnosť je riadená manuálne pomocou new a delete.

3.4 Jak se dá postupovat, pokud chceme v zadání programu nalézt třídy, jejich metody a datové členy?

- **triedy** - často opakujúce sa podstatné mená
- **metódy** - slovesá
- **dáta** - podstatné mená

3.5 Kdy a proč potřebujeme použit více konstruktorů jedné třídy?

V prípade, že potrebujeme viacero inštancií danej triedy s rôznym počtom alebo typom parametrov.

3.6 Kdy potřebujeme deklarovat a definovat destruktor?

Keď sú dáta objektu dynamické.

3.7 Co jsou výchozí konstruktory a destruktory a k čemu je potřebujeme?

Východzí konštruktor je automaticky definovaný kompilátorom, pokiaľ nie je explicitne deklarovaný. Slúži k vytvoreniu objektu s východzími hodnotami.

Východzí deštruktor je automaticky definovaný kompilátorom, pokiaľ nie je explicitne deklarovaný. Slúži k zničeniu objektu bez špecifickej činnosti.

3.8 Jaké typy metod obvykle musíme deklarovat a definovat?

- konštruktor a deštruktor
- metódy poskytujúce informácie o stave objektu (getter)
- metódy meniace stav objektu (setter)

3.9 Co jsou objektové kompozice a k čemu jsou dobré?

Objekt môže byť súčasťou iného objektu a stáva sa jeho dátovou položkou.
Vznikajú tak komponované objekty s presne definovanými kompetenciami. Tie však môžu byť realizované prostredníctvom iterácie objektov, z ktorých sú komponované.

4 Objektová dekompozice a třída jako objekt

4.1 Jaký je rozdíl mezi funkční a objektovou dekompozicí programu?

Funkčná dekompozícia sa zameriava na rozdelenie programu do samostatných funkcií alebo procedúr, ktoré vykonávajú konkrétne úlohy.
Objektová kompozícia je charakteristická pre OOP. Program je v nej rozdelený objekty, ktoré sú inštanciami tried.

4.2 Proč preferujeme objektovou dekompozici a jaké jsou hlavní problémy funkční dekompozice?

Objektová dekompozícia nám umožňuje používať lepšie modelovať reálne objekty a ich interakcie. Môžeme používať enkapsuláciu, dedičnosť, polymorfizmus a umožňuje nám lepšiu rozšíriteľnosť.
Problémy funkčnej dekompozície sú zlá rozšíriteľnosť, nedostatok štruktúr pre zložitejšie systémy, obmedzená reprezentácia zložitých entít, horšie testovanie a debugging.

4.3 Za jakých podmínek můžeme považovat třídu za objekt a jak to implementovat v C++?

Trieda sa stáva objektom ak:

- obsahuje dáta, ktoré reprezentujú vlastnosti objektu
- má metódy
- má konštruktor
- môže mať prístupové modifikátory
- môže využívať dedičnosť a polymorfizmus

4.4 Vysvětlete rozdíl mezi členskými položkami třídy a instance a popište jejich dostupnost.

Členské položky triedy sú spojené s triedou ako celkom, môžeme k nim pristupovať bez potreby vytvárať inštanciu triedy, ale pristupujeme k nim prostredníctvom samostatnej triedy. Tieto položky sú zdieľané medzi všetkými inštanciami triedy.
Členské položky inštancie sú spojené s konkrétnymi inštanciami triedy. Každá inštancia má svoju vlastnú kópiu týchto atribútov a metód. K položkám pristupujeme pomocou objektu.

4.5 Jak můžeme v C++ důsledně odlišovat práci s členskými položkami tříd a instancí?

K **členským položkám triedy** pristupujeme pomocou názvu triedy - ClassName::staticMember.
K **členským položkám inštancie** pristupujeme prostredníctvom konkrétnej inštancie objektu - object.member.

4.6 Potřebuje třída v roli objektu konstruktor resp. destruktor a proč?

Áno, potrebuje. Konštruktor zaistí, že objekt je správne inicializovaný a pripravený k používaniu. Deštruktor zaisťuje správne uvoľnenie prostriedkov, aby sa predišlo únikom pamäti. Inak by to mohlo viesť k nesprávnemu chovaniu aplikácie.

5 Úvod do dědičnosti

5.1 Které dva klíčové požadavky řešíme pomocí dědičnosti?

Znovu-použiteľnosť a rozšíriteľnosť.

5.2 Jaké návrhové požadavky máme na použití tříd (co s nimi můžeme dělat)?

Kombinovanie s inými triedami, skladanie, rozšírenie o nové chovanie, pozmenenie existujúceho chovania.

5.3 Jaký je rozdíl mezi dědičností a skládáním? Co mají společného?

Skladaním docielime to, že objekt jednej triedy je zložený z objektov inej triedy. Jedná sa o vzťah „MÁ".
Dedičnosťou docielime to, že nová trieda je rozšírením alebo špeciálnym prípadom existujúcej triedy. Jedná sa o vzťah „JE".

5.4 V jakých rolích vystupují třídy v dědičnosti? Použijte správnou terminologii.

Predok - potomok, priamy predok - potomok. Rodič - dcéra (syn). Nadradená trieda - pod trieda.

5.5 Vysvětlete v jakém obecném vztahu je třída, ze které se dědí, se třídou, která dědí.

Trieda z ktorej sa dedí je rodičovská trieda. Potomok zdedí všetky metódy rodičovskej triedy. Môže dané chovanie rozšíriť alebo pozmeniť. Nemôže sa ho zbaviť.

5.6 Co všechno se dědí, co ne a proč?

Dedí sa všetko bez výnimky, rovnako aj miera skrývania informácie. Pretože predok definuje spoločné chovanie všetkých svojich potomkov. Potomkovia môžu toto chovanie rozšíriť či pozmeniť. Nemôžu sa ho zbaviť.

5.7 Co rozumíme jednoduchou dědičností a jak s tím souvisí hierarchie tříd v dědičnosti?

Každý potomok má práve jedného priameho predka. Predok môže mať viac priamych potomkov. Hierarchiou je strom.

5.8 Co je Liskové substituční princip a jak se projevuje v dědičnosti?

Potomok môže vždy zastúpiť predka a to preto, že majú spoločné chovanie. Opačne to neplatí.

5.9 V jakém pořadí se volají a vykonávají konstruktory při použití dědičnosti?

1. Volanie konštruktora objektu.
2. Volanie konštruktoru priameho predka.
3. Vykonanie konštruktoru priameho predka.
4. Vykonanie konštruktoru objektu.

6 Dědičnost – změna chování

6.1 Co rozumíme paradoxem specializace a rozšíření?

Vzťah dedičnosti je vzťahom obecný - špeciálny. Potomok je teda špeciálnym prípadom predka. Paradoxné je, že pri rozšírenní dochádza k tomu, že potomok vie viac ako ktorýkoľvek jeho predok. A teda čím bohatšie chovanie uvažujeme, tým menej tried ho poskytuje.

6.2 Uveďte správné a špatné příklady vztahu "generalizace-specializace".

Zlý príklad - vzťah bodu a kružnice. Mohli by sme potrebovať rozšíriť bod o prácu s polomerom, t.j. pridať nové chovanie. Avšak potreba rozšírenia sama o sebe nie je dostačujúca pre použitie dedičnosti. Nie je splnená potreba špecializácie (kružnica nie je špeciálnym prípadom bodu).

Dobrý príklad - vozidlo je všeobecná kategória reprezentujúca prostriedky na prepravu, takže bicykel môžeme považovať za špeciálny typ vozidla - má konrkétne vlasnosti.

6.3 Co rozumíme v dědičnosti změnou chování?

Pokiaľ je chovanie deklarované v predkovi, môžeme ho v potomkovi deklarovať znovu. Existuje potom viac metód rovnakého mena. Deklarované chovanie potom musíme v potomkovi implementovať.

6.4 Co rozumíme přetížením? Jedná se o rozšíření nebo změnu chování?

Preťaženie je situácia, keď daná metóda má rovnaké meno ale má iné parametre, iné typy parametrov, inú návratovú hodnotu. Preťaženie nie je zmena chovania, napriek tomu, že má rovnaké meno. Ide teda o rozšírenie chovania.

6.5 Uveďte různé typy přetížení.

- meno metódy zostáva rovnaké
- iný počet parametrov
- iné dátové typy parametrov
- iná návratová hodnota (nie v C++)

Tieto typy je možné kombinovať.

6.6 Co rozumíme překrytím? Jedná se o rozšíření nebo změnu chování?

Prekrytie je situácia, kedy metóda potomka má rovnakú deklaráciu ako metóda predka. Potomok dedí aj metódy predka. Má teda dve metódy s rovnakou deklaráciou. Je to zmena chovania. Typickým príkladom použitia sú konštruktory.

6.7 Jaký princip porušujeme, použijeme-li „protected“ a proč?

Porušujeme princíp zapúzdrenia. Pri zmene chovania môže vzniknúť potreba pracovať so súkromnou časťou predka.

6.8 Jaký problém přináší potřeba změny chování v dědičnosti?

Prekrytie metód, polymorfizmus, zhoršenie čitateľnosti

6.9 Popište, jak se prakticky projevuje různá míra přístupu k položkám třídy.

- **public** - položka triedy je dostupná odkiaľkoľvek
- **private** - položka triedy je prístupná iba v rámci triedy, v ktorej bola definovaná
- **protected** - položka je prístupná v triedach, ktoré dedia danú triedu, ale nie úplne voľne pre akýkoľvek kód

6.10 Jak se použití „protected“ projeví ve vztahu předka a potomka?

Prejaví sa to tak, že protected položky sú prístupné v podtriedach, aj keď sú inak skryté pred zbytkom programu. Potomok má priamy prístup k protected metódam a atribútom. Môže protected metódy prekryť a zmeniť ich chovanie. Pracuje s položkami akoby to boli jeho vlastné.