Objektový návrh

přednáška NPRG031 Programování 2 18. 3. 2020

Proč

Dosud jsme viděli různé významy slova „programování", jako řešení nějaké úlohy, hledání algoritmu, implementace… To všechno jsou části činnosti nazývané **vývoj softwaru.**

Současné programy svou velikostí a složitostí patří k nejsložitějším lidským výtvorům. Velká část projektů vývoje softwaru nedokáže dodržovat termíny, rozpočet, případně práci vůbec nedokončí. Proto hledáme způsob, jak software vyvíjet nějak organizovaně a předvídatelně. Zabývá se tím celá disciplína – **softwarové inženýrství,** výsledek jsou rady, návody, doporučené postupy; s některými se ještě setkáme.

Jedním z typických rysů softwaru je **nekončící potřeba změn a úprav,** ať jde o opravy chyb, rozšiřování funkčnosti nebo přizpůsobování měnícím se potřebám zákazníka. Problém je v tom, že když ve fungujícím systému nějakou část změníme, může se stát, že tato změna ovlivní (až překazí) fungování jiných částí.

Proto se snažíme o **dekompozici** – rozdělení softwaru na části, které spolu komunikují pouze prostřednictvím předem daných **rozhraní** (interface). Podle toho, co jsou tyto **části**, mluvíme o dekompozici **funkční, modulární** nebo **objektové.**

Rozhraní je přitom to jediné, co o sobě jednotlivé části musí vědět, takže pokud se nezmění rozhraní, změna uvnitř jedné části nenaruší funkčnost ostatních částí.

Objektový návrh

Cílem objektového návrhu je rozdělení softwaru do **objektů a tříd** a návrh jejich komunikace (návrh jejich **rozhraní**).

Abychom se o takových návrzích mohli bavit, hodil by se nám nějaký **společný jazyk.** V současnosti je takovým jazykem **UML 2.0.**

UML 2.0

Jazyk **Unified Modeling Language (UML)** je domluvený systém, jak pomocí obrázků („diagramů“) zachycovat software z různých pohledů

(podobně jako třeba rodinný dům vypadá jinak z pohledu bytového architekta, elektrikáře, hasiče, instalatéra, lektora feng-šhui nebo bytového zloděje).

O vývoj UML se stará Object Management Group (OMG), formální specifikace zde: https://www.omg.org/spec/UML/2.0 .

Diagram tříd

Diagram tříd (class diagram) je jeden z diagramů UML, znázorňuje jednak **třídy** a a jednak **vztahy** mezi objekty a mezi třídami.

U každé třídy potom znázorňuje jejich **atributy** a **operace** s vyznačením přístupnosti (+ pro **public**, - pro **private** a řada dalších).

Vztahy mezi třídami a mezi objekty

Rozlišujeme následující druhy vztahů:

1 mezi objekty
   1.1 asociace
   1.2 agregace
   1.3 kompozice
2 mezi třídami
   2.1 závislost

2.2 generalizace

3 mezi třídou a rozhraním

3.1 realizace

Asociace

Základní vztah mezi objekty, které mají něco společného, třeba objekt **Student** má vztah s objektem **Předmět.**

Kreslí se obyčejnou čárou.

Může mít **orientaci** (třeba **Auto** potřebuje znát svůj **Model**), opačně to neplatí.

Může mít **název**, třeba **Student** ---studuje---> **Předmět.**

Může mít **násobnost** (1:1, 1:N, M:N), třeba Auto a Model…

Téměř všechny asociace jsou **binární**, má smysl i aby byly ternární (třeba potomek a jeho rodiče), dá se vyjádřit pomocí binárních.

Agregace

Vztah vyjadřující, že jeden objekt je částí jiného objektu.

Objekt **Auto** je částí objektu **Událost**, ale nepatří mu, existuje i bez něj. Jiný příklad: **Adresa** je součástí objektu **Firma**, ale více firem může mít stejnou adresu.

Kreslí se čárou **s prázdným** kosočtvercem na straně CELKU (pokud jsou objekty ČÁST a CELEK).

Kompozice

Neboli „silná agregace" část nemá smysl bez celku. **Auto** má čtyři **Kola. Model** má **Kalendář**, na začátku ho vyrobí a na konci zničí.

Kreslí se čárou **s plným** kosočtvercem na straně CELKU.

Závislost

Orientovaný vztah, též „server-klient". **Server** má něco, bez čeho **Klient** nemůže fungovat.

Kreslí se čárkovanou čárou.

Generalizace

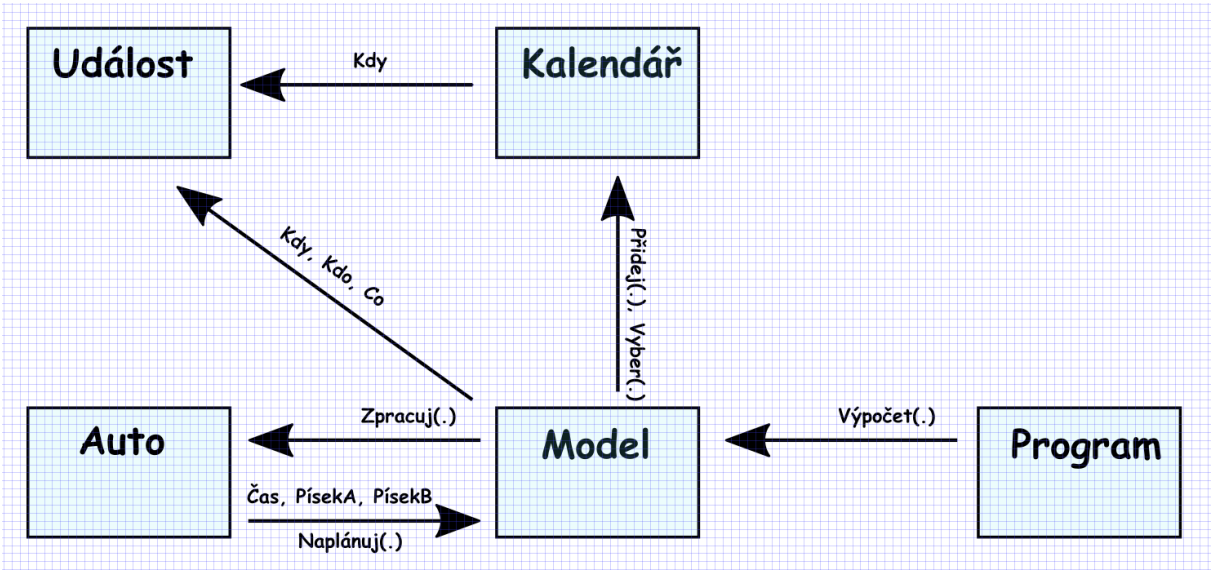
Třída **Kočka** je zobecněním třídy **Tygr** nebo obráceně „Tygr je Kočka".
Někdy nazývaný **isa vztah** protože „a tiger **is a** cat".

Kreslí se čárou s prázdnou šipkou na konci.

Realizace

Vztah mezi třídou a rozhraním. Třída realizuje určité rozhraní.

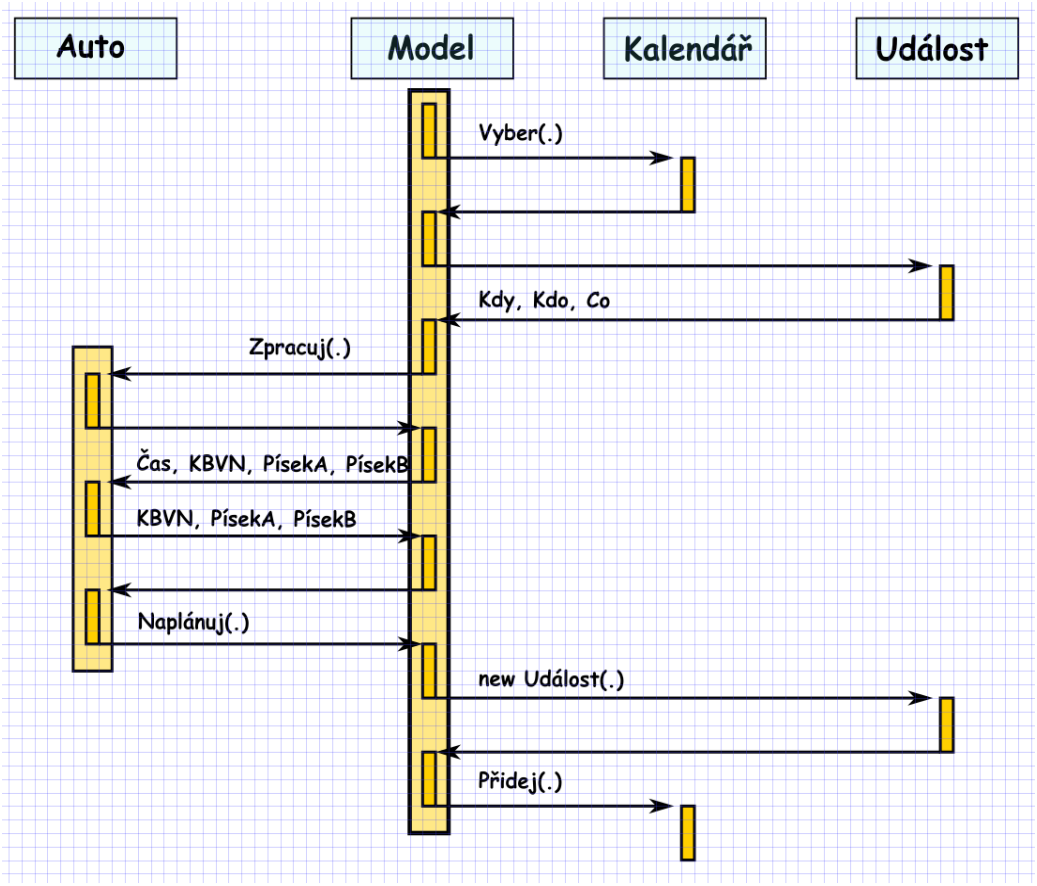
Vztahy



Tohle není přesně Diagram tříd, ale šipky znázorňují, co který objekt potřebuje vědět o jiném objektu, tedy co vše se může pokazit (proti směru šipek), pokud něco změníme.

Sekvenční diagram

...znázorňuje průběh spolupráce objektů.



Jak probíhá jeden krok cyklu zpracování událostí objektem Model.

Jak navrhovat

Společný jazyk (UML, ale i jiné) potřebujeme k tomu, abychom se mohli s někým domlouvat a rozuměli si. Na druhou stranu i obrázek, který nevyužívá a nesplňuje všechny možnosti a pravidla jazyka, může posloužit tomu, abychom si něco lépe rozmysleli a lépe viděli.

Osobní poznámka:
Myslím, že pozoruji opakující se vzorec:

- *na začátku je jednoduchá myšlenka, se kterou všichni souhlasí*
- *potom někdo položí neodolatelnou otázku* ***„Jak to mám dělat správně?"***
- *na to se vyrojí spousta lidí ochotných vysvětlovat, jak se to má dělat správně*
- *tato vysvětlení se vzájemně liší*
- *začnou vznikat skupiny, fan-kluby, názorové proudy, frakce a hnutí, které se začnou hádat, pomlouvat, vylučovat, věznit a jinak si ubližovat.*

Mohli jsme to pozorovat v náboženství, v politice, v umění… i jinde. Proto zmíněné diagramy i následující pravidla berte spíše jako úhel pohledu, ze kterého se můžete podívat na svůj objektový návrh a zeptat se, jestli jste na něco nezapomněli.

Princip jedné zodpovědnosti (Single Responsibility Principle)

Jeden objekt by měl mít na starosti jednu věc. Fronta by měla přidávat a vybírat prvky, ale pokud je s nimi potřeba dělat něco dalšího, ať to dělá někdo jiný (další objekt/třída).

Princip otevřenosti a zavřenosti (Open–closed principle)

Třída by měla být otevřená pro rozšiřování (třeba tak, že odvodíme novou třídu, do které přidáme data a funkce), ale uzavřená ve smyslu, že má definitivní rozhraní, které mohou ostatní používat a které se nebude měnit.

Zákon substituce (Liskov substitution principle)

Pokud je někde potřeba objekt typu A, musí být místo něj možné použít i objekt typu B, pokud B je odvozený od A (neboli opravdu platí že Tygr je taky Kočka).

Princip oddělených rozhraní (Interface segregation principle)

Klient by neměl být nucen záviset na metodách, které nepoužívá, takže místo velkého společného rozhraní je lepší mít několik oddělených malých rozhraní určených pro různé klienty (třída v C# může implementovat více různých rozhraní).

Princip obrácení závislosti (Dependency Inversion Principle)

Vyšší části by neměly záviset na (konkrétních) nižších částech, ale na abstrakci, protože implementace se mění často, abstrakce by měla být trvalejší. Konkrétnější musí záviset na abstraktnějším.

Deméteřin zákon (Law of Demeter)

Funkce FFF() ve třídě TTT by měla volat pouze:

1. funkce třídy TTT
2. funkce objektů předaných jako parametry funkci FFF
3. funkce objektů vytvořených funkcí FFF
4. funkce objektů patřících třídě TTT
5. globální funkce

Protipříklad:

```
seznam.vyber().split()[0].ToItem().Name.ToString()
```

DRY (Don‘t Repeat Yourself)

Každá informace by měla být v programu jen jednou – jediný zdroj pravdy (Single Source Of Truth). Pokud je ji potřeba ve více formách, je správné ji generovat z jednoho zdroje.

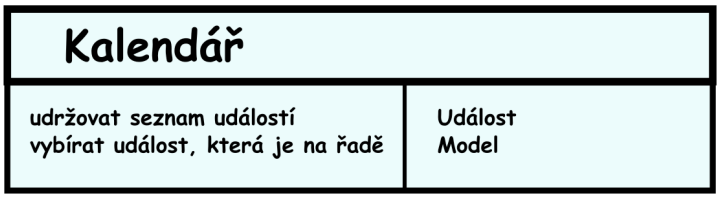
Opak se nazývá **WET (Write Everything Twice!).**

High cohesion, Low coupling…

...a mnoho dalších pravidel. Nízká (nebo vysoká) provázanost je něco, co můžeme dobře vidět na diagramu tříd, i v té zjednodušené formě na obrázku výše.

CRC kartičky

Class-Responsibility-Cooperators. Jedna z pomůcek pro přemýšlení



Návrhové vzory

Myšlenka, že v mnoha programech se opakují stejné (pod)problémy a že by bylo možné vydestilovat vzorová řešení těchto podproblémů. Původní kniha **Design Patterns: Elements of Reusable Object-Oriented Software** autorů Gamma, Helm, Johnson a Vlissides, přezdívaných „Gang of Four (GoF)“. Učí se v předmětu **NPRG024 Návrhové vzory.**

Guruové, teoretici a další čtení

Martin Fowler https://martinfowler.com/
Robert C. Martin https://blog.cleancoder.com/
Kent Beck, třeba http://c2.com/doc/oopsla89/paper.html
Bertrand Meyer, třeba
http://se.ethz.ch/~meyer/publications/computer/taxonomy.pdf
Paul Graham http://www.paulgraham.com/nerds.html
Joel Spolsky https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/

(Jestli se Vám nechce číst, tak alespoň ten předposlední…
(není to o objektech).)

Závěr

Smyslem této přednášky bylo připomenout, že tvorba programu by neměla začínat zdrojovým kódem, ale přemýšlením a také že na to přemýšlení existují určité návody a postupy (i když si někdy odporují).

A pokud někdy nevíte kudy kam, nakreslete si obrázek!