Kapitola 5

Objektový návrh, hygiena programování

5.1 Objektový návrh

Pracujeme-li na větším projektu ideálně ve více osobách, je potřeba rozmyslet, jak si rozdělit práci. K tomu slouží dekompozice programu a také objektový návrh. V tomto návrhu popíšeme, jakou strukturu bude program mít (tedy jak bychom to dělali, kdybychom na to měli čas).

Z objektového návrhu musí být patrné, jaké třídy v programu plánujete mít, co bude v jaké třídě (atributy a metody) a kdo s kým bude komunikovat. Můžete k tomu využít například kód v C#, ze kterého vynecháte těla všech funkcí (a zůstane tedy popis namespaců, tříd, hlavičky funkcí a definice atributů). V této chvíli ještě není jasné, kdo koho bude volat. Proto si povíme o některých vhodných formalismech:

5.1.1 UML diagramy

Jedním formalismem se širokým využitím jsou UML-diagramy. UML je zkratka Unified Modeling Language. UML diagramy jste patrně již někde viděli. K jejich kreslení existují různé nástroje. Ze všech padají obrázky, ve kterých jsou krabičky pospojované šipkami. Šipky říkají, kdo s kým komunikuje (kdo koho volá, tedy kdo na kom závisí).

Každá krabička má název (nahoře) a položky (obvykle vypsané pod sebou). Toto je formalismus jak dělaný na zápis tříd s metodami a atributy. U metod je vhodné si rozmyslet, jaké budou mít argumenty (abyste si ověřili, že požadovaná data budete schopni předat).

5.1.2 Data Flow Diagramy

Vypadá jako olabelovaný orientovaný graf. Co je orientovaný graf, víte z Diskrétní matematiky. Olabelovaný znamená, že hrany olabelujeme (tedy jim přidáme nějaký atribut – nějakou informaci). Vrcholy reprezentují jednotlivé operace (úkony, funkce či moduly) a (orientované) hrany říkají, jaká data odkud kam putují. Tedy šipkou označíme, odkud kam data poputují a label řekne, co za data to bude (například "setříděný seznam studentů"může putovat ze Studijního oddělení jednomu každému zkoušejícímu).

Data Flow Diagram poměrně jasně použijete tam, kde proudí data (na vstupu máme data, na výstupu je chceme vidět také, ale nějak vkusně zpracovaná). UML-diagram použijete, pokud chcete popsat, kde bude jaká funkce a kde jaká metoda. U Data Flow Diagramu není patrná struktura kódu programu, v UML-diagramu zase zaniká proudění dat. Záleží tedy především na tom, co chceme v tuto chvíli vizualizovat.

Při přípravě objektových návrhů myslete na to, že tyto návrhy musejí být především realizovatelné. Ačkoliv děláme, co můžeme, času k procvičení implementace objektových návrhů moc nemáme. Ve vlastním zájmu si to však zkuste.

5.2 Hygiena programování

5.2.1 Zdravotní aspekty

Na našince všude číhá plno nástrah. Nejen zkoušející pátrající po tom, co neumíte, ale informace o zdravotních rizicích je poslední dobou čím dál důležitější. Školení BOZP (tedy bezpečnosti a ochrany zdraví při práci) jste absolvovali (nejspíše na Albeři) a od teď vás BOZP bude pronásledovat celý (pracovní) život (v životě osobním máte též Občanským zákoníkem uloženu všeobecnou prevenční povinnost, ale nikdo nemá povinnost vás o tom poučovat).

Ačkoliv rizika nevypadají (podle slidů) pěkně, myslete na to, že aby nastal malér, museli byste doporučení porušovat poměrně vyzývavě (dlouhodobě či intenzívně). To se občas někomu stane (a je na to ošklivý pohled). Rizika však je nutné sledovat a vyhodnocovat průběžně. Nečekat, až ve čtyřiceti už ruce nad klávesnicí neudržíte (a pak řešit, jak budete programovat, když už nehnete prsty na rukou).

Hardware, na kterém fungujeme, nám dlouho hlásí, pokud něco není v pořádku. Důležité je všimnout si toho včas. Udržovat se ve vhodné kondici je důležité z mnoha důvodů. Jedním z nich je, že není denně posvícení. Některé dny přijdou třeba viry věnčité, a pokud člověk za běžného počasí sotva dýchal, zajímavě to dopadne.

Matfyzáctví má plno specifik. Jedním z nich je, že studium je i základním způsobem duševně náročné. Jeden z mých někdejších studentů (bývalý – nepochybně statečný – voják zvláštních jednotek) situaci zhodnotil: "Matfyz is harder than special forces!"Myslete tedy na to, že zdraví máte jedno. Až si ho zničíte (ať to znamená, co chce), bude po něm. U nás je zvykem udržovat se

celkově v dobrém stavu (odbornice se poměrně obdivně vyjádřila o tom, v jakém stavu je u nás péče o duševní zdraví). Pokud vás nápadně ostatní začnou upozorňovat, že vám hrozí nějaký problém, je dobře vzít jejich připomínky v úvahu (i když samozřejmě pravdu mít nemusejí). Ostatně Matfyzáctví není soutěží v rituálech a ani soutěží v masochismu, ale mistrovství v uvedení hardwaru, na kterém fungujeme, do vysoce výkonného stavu (který musí být dlouhodobě udržitelný).

5.2.2 Odborné aspekty

Kód, který napíšeme, se obecně má používat řadu let. Je proto důležité, aby byl co nejdéle udržovatelný. K tomu bychom měli upírat své úsilí při jeho psaní. Zatím jste na začátku cesty a pokoušíte se především překladač přesvědčit, ať udělá, co chcete. Tohoto cíle se vám (pokud to nevzdáte) podaří dříve či později dosáhnout. Pak bude potřeba pracovat na tom, abyste psali kód kultivovaně. S kulturou programování ovšem není proč otálet. Kultivovaně navržený kód se vám bude lépe ladit. Ve slidech tedy najdete některá doporučení týkající se implementace kódu.

Tyto rady od nás průběžně slýcháte od začátku roku. Tedy kód máte vhodně komentovat. Komentáře mají popisovat to, co v kódu není na první pohled patrné. Tedy zinicializujeme-li proměnnou i nulou, vhodný komentář říká, proč jsme to udělali (třeba proto, že budeme hledat minimální přirozené číslo s nějakou vlastností – a toto číslo se nám objeví právě v proměnné i). Špatný komentář říká to, co je očividné, tedy že *do i přiřadíme nulu.*

Zdrojové kódy je potřeba vhodně dekomponovat do součástí, které spolu souvisí. K tomu slouží dekompozice do modulů (souborů), tříd (objektů) a funkcí. Rozdělení musí být účelné a funkční. Opět programování není mistrovstvím v masochismu (kdy budeme muset pracně přemýšlet, jak se funkce dostane k datům, která potřebuje – od toho děláme objektový návrh, abychom se ujistili, že každou funkci jsme schopni implementovat s těmi parametry, co předáváme – a že každý volající dotyčné údaje má). Ve slidech najdete popisy některých konkrétních situací, jakož i návrhové principy SOLID.

5.3 Verzovací systémy

Systémy pro správu verzí souvisí s kulturou programování. Je vhodné je používat při implementaci větších projektů a při týmové práci jsou nutné (alternativy jsou příliš komplikované). Tento systém umožňuje udržovat projekt (potažmo kód s ním související) v synchronizovaném stavu. Tedy když rozkopete nějakou funkci, rychle se o tom dozvědí ostatní (stáhnou si vaši rozkopanou funkci a kód jim přestane kompilovat). Tak se podívají, co se stalo, a zjistí, že jste si konečně pořádně přečetli objektový návrh a opravili jste předávané parametry. Zajásají a přepíší všech 500 volání. :-) Anebo udělají fallback na předchozí verzi. Ne nadarmo se ty systémy jmenují verzovací. Udržují totiž i údaje o všech předchozích (tzv. commitnutých) verzích.

Verze nevznikají samočinně, protože by bylo těžké říci, kdy jste už napsali jednu jednotku programu (kterou má smysl ukládat). Napíšete tedy kus kódu a řeknete *ted'!* Verzovací systém vaše změny nahraje do repozitáře (a tak vznikne nová verze).

Verzovacích systémů je (či bylo) několik. Konkrétně a například RCS, CVS, SVN nebo Git. Zpravidla jde o nějaké zkratky (Revision Control System, Concurrent Versions System, Subversion). Akorát Git se jmenuje po svém autorovi, Linusu Torvaldsovi.

Kromě různých advancovaných klikátek mají verzovací systémy obvykle i výkonné řádkové rozhraní (kterým systém ovládáte z příkazové řádky a máte tak přehled, co se vám děje pod rukama).

Normální člověk používá verzovací systém tak, že někde zprovozní repozitář (v nejhorším případě si založí server, vy můžete použít již zprovozněný `gitlab.mff.cuni.cz`), nastuduje základní instrukční repertoir umožňující vytáhnout současnou verzi, tuto verzi updatovat (proběhly-li nám změny pod rukama) a uložit změny do repozitáře. Přednášející dosud používá CVS, kde se tyto rituály dělaly takto:
Nastavení cesty k repozitáři:
`export CVSROOT=cesta_k_repozitáři`
Založení nového repozitáře:
`cvs init`
Založení nového projektu (import z primárních zdroják):
`cvs import ...`
Získání projektu v současném stavu (tzv. vycheckoutování)
`cvs checkout jmeno_modulu`
Zapsání změn do repozitáře
`cvs commit`
Update (vhodné udělat před commitem, kdy je nutné vypořádat případné konkurentně provedné změny:
`cvs update -PRd`
Kdo to ten soubor, s..., tak rozhrabal a proč:
`cvs annotate file_name` (vypíše jednotlivé řádky současné verze s informací kdo který řádek přidal ve které verzi, pak si můžete přečíst commitovací komentář a zkusit odhadnout, proč to najednou nefunguje).
Co jsem to vyvedl?!:
`cvs diff file_name`
(ukáže, co jsme od minulé commitnuté, nebo jinak popsané verze změnili).

Každý verzovací systém nějakým způsobem implementuje tyto operace. Konkrétně dnes nejpoužívanější Git občas jeden rituál nahrazuje více rituály. Při commitu nedojde k zapsání dat do repozitáře, ale příkazem `commit` jen připíšeme poznámky, co jsme změnili. Samotný upload se totiž provede až pomocí `git push`. Taktéž je matoucí příkaz `add`. Ten v CVS přidá soubor do repozitáře, kdežto v Gitu se jím označují soubory pro commit (což CVS poznala sama, které soubory se změnily a rituál commitu tedy reprezentoval jeden příkaz namísto tří, ale pokrok nezastavíme).

Běžný člověk se tedy naučí zmíněný instrukční repertoir (pro verzovací systém

svého mládí), ten pak používá (a na jiný si těžko zvyká). Gitlab má i pěkné webové rozhraní, které umožňuje vést další agendu. Například todo-list, potažmo popis featur (které máte implementovat) a bugů (které máte opravit). Můžete si tam tak vést poznámky o tom, v jakém stavu projekt je (co už jste implementovali, co čeká, kde je jaká díra a podobně). Povšimněte si, že ReCodEx je doma na GitHubu, což je zase jen zamaskovaný Git (také s webovým interfacem) – `github.com/ReCodEx`.

Tento týden, jak vidíte, probíráme poněkud méně technicky náročná témata. Čas ovšem využijte ke zlepšování vlastních schopností.