EDICE
myslíme v ...

knihovna programátora

- Učebnice pro ty, kteří nechtějí zůstat obyčejnými kodéry, ale chtějí se stát špičkovými architekty
- Postupuje podle metodiky Architecture First
- Soustředí se na návrh programů a osvojení klíčových architektonických zásad
- Vysvětluje a procvičuje návrhové vzory, refaktoraci kódu, vývoj řízený testy a další oblasti, které běžné učebnice ignorují
- Vše průběžně procvičuje na příkladech řešených spolu se čtenářem
- Doporučená učebnice na řadě středních škol i univerzit

RUDOLF PECINOVSKÝ

Java 7

učebnice objektové architektury pro začátečníky

GRADA

EDICE
myslíme v ...

O autorovi

Rudolf Pecinovský patří ke špičkovým odborníkům na výuku programování. Publikoval již 43 učebnic, které byly přeloženy do pěti jazyků, a nepřeberné množství článků a příspěvků na odborných konferencích. Je autorem metodiky výuky programování *Karel*, navazující metodiky *Baltík* a moderní metodiky výuky objektově orientovaného programování známé pod anglickým názvem *Architecture First*. Učí programování na VŠE a FIT ČVUT. Současně pracuje jako Senior EDU Expert ve firmě ICZ a.s., kde má na starosti doškolování profesionálních programátorů a organizaci kurzů, které si objednávají jiné firmy.

O knize

Tato kniha je třetím vydáním populární učebnice programování, která je na našem trhu zcela ojedinělá. Na rozdíl od ostatních učebnic, které se soustředí na výuku syntaxe jazyka a práce s knihovnami, se tato kniha soustředí především na výklad architektonických konstrukcí. Neučí čtenáře kódovat, ale snaží se jej naučit, jak programy navrhovat. Učí jej, jak má při programování myslet. Reaguje tak na známou skutečnost, že kodérů je hodně, ale dobrých softwarových architektů je proklatě málo (proto také mají několikanásobně vyšší platy).

Kniha je sice primárně určena začátečníkům, ale ohlasy na předchozí vydání ukázaly, že v ní najdou poučení i zkušení programátoři. Většina učebnic a kurzů programování totiž vyvolává falešnou představu, že objektově programovat znamená používat třídy a dědění. Tato kniha je první, která ukazuje, že objektově orientované programování přináší především jiný způsob myšlení. Jak výstižně napsal jeden čtenář: „*Myslel jsem si, že nejsem žádné programátorské ucho. Když jsem ale přečetl vaši učebnici, otevřel jsem oči a hubu. Konečně jsem pochopil věci, které mi ostatní učebnice nedokázaly vysvětlit.*“

Kniha vznikla na základě dlouholetých autorových zkušeností se školením profesionálních programátorů, výukou programování na univerzitě i vedením žákovských programátorských kroužků. Autor v ní uvádí čtenáře krok za krokem do tajů objektově orientovaného programování a ukazuje mu, jak možnosti této moderní technologie co nejlépe využít a kde si dát naopak pozor na její úskalí.

Výklad je postaven na příkladech, které autor spolu s čtenářem postupně řeší a přitom čtenáře učí nejenom základním programátorským návykům a dovednostem, ale předvede mu i nejrůznější užitečné triky, z nichž mnohé nikde jinde vysvětlené nenajdete. Současně upozorňuje na nejčastější začátečnické chyby, které před svými čtenáři ostatní učebnice většinou tají. Navíc probírá i řadu témat (např. návrhové vzory), která jsou většinou probírána až v pokročilých, nebo dokonce nadstavbových kurzech, přestože patří do základní výbavy objektového programátora.

knihovna programátora

RUDOLF PECINOVSKÝ

Java 7

učebnice objektové architektury pro začátečníky

GRADA
Publishing

Java 7
učebnice objektové architektury pro začátečníky

Rudolf Pecinovský

Vydala Grada Publishing, a.s. U Průhonu 22, Praha 7
jako svou 4971. publikaci

Odborní lektoři:
doc ing. Alena Buchalcevová, Ph.D., doc. ing. Pavel Herout, Ph.D.,
doc. MUDr. Jiří Kofránek, CSc., doc. ing. Vojtěch Merunka,
Ph.D., prof. RNDr. PhDr. Antonín Slabý, CSc., doc ing. Miroslav Virius, CSc.

Odpovědný redaktor: Martin Vondráček
Návrh vnitřního layoutu: Rudolf Pecinovský
Zlom: Rudolf Pecinovský
Počet stran 496
Vydání 1., 2012

Vytiskla Tiskárna PROTISK, s.r.o., České Budějovice

ISBN 978-80-247-3665-5 (tištěná verze)
ISBN 978-80-247-8325-3 (elektronická verze ve formátu PDF)
ISBN 978-80-247-8326-0 (elektronická verze ve formátu EPUB)

Mé ženě Jarušce a dětem
Štěpánce, Pavlínce, Ivance a Michalovi

Stručný obsah

Podrobný obsah

Skrytí spoluautoři

Knihu bych nemohl dokončit v předepsaném čase, kdyby mi s ní nepomohli moji studenti, kteří se podíleli především na přípravě doprovodných programů a dalšího podpůrného programového vybavení a přispěli i řadou připomínek k obsahu a stylu výkladu. Dovolte mi proto abecedně uvést alespoň ty nejzasloužilejší.

Daniel Bartoň se podílel na odladění některých knihovních programů, převodu knihoven do angličtiny pro druhý díl učebnice a úpravách speciální třídy umožňující definici jediné testovací třídy pro celou skupinu tříd se společným rodičem.

Sergej Bobuskyy měl hlavní podíl na vývoji pluginu BJ2NB, který doplňuje možnosti prostředí *NetBeans*, v němž budeme pracovat v druhém dílu, o schopnost průběžného zobrazování diagramu tříd a jeho provázání se zdrojovým kódem, na které si zvyknete u prostředí *BlueJ*, jež se používá v dílu prvním.

Martin Fiala vyvinul pro plugin BJ2NB editor kopenogramů.

David Král pomáhal s přípravou podkladů pro generátor projektů a s úpravami tohoto generátoru v rámci jeho průběžného zdokonalování.

Filip Malý doplnil a upravil sadu maker, která automatizují některé činnosti spojené s přípravou rukopisu a jeho následného převodu do finální podoby. Podílel se i na vývoji knihovny pro automatizované testování vyvinutých programů.

Vladimír Oraný v rámci své diplomové práce vyvinul (a před vydáním knihy na poslední chvíli ještě upravil) speciální knihovnu umožňující výrazně rozšířit možnosti deklarace požadavků, kterým musejí vyhovovat testované programy. Setkáte se s ní v druhém dílu.

Jarmila Pavlíčková a **Luboš Pavlíček** nejsou mí studenti, ale kolegové na VŠE. Mnohé z formulací použitých ve výkladu se vytříbily na základě našich četných (a mnohdy i vášnivých) debat o výuce programování. Jarmile bych chtěl navíc poděkovat za to, že mne osvobozuje od řady administrativních úkonů, které jsou pro mne téměř nezvládnutelnou překážkou.

Martin Vondráček podrobně pročetl celý rukopis a přispěl řadou poznámek k zvýšení jeho čitelnosti, srozumitelnosti a odborné přesnosti.

Na závěr pak musím vyjádřit svůj velký dík firmě ICZ a veškerému osazenstvu oddělení Realizace. Bez jejich podpory by kniha nevznikla.

Předmluva k prvnímu vydání

Rudu Pecinovského jsem poprvé potkal v době, kdy jsme oba studovali na Jaderné fakultě ČVUT v Praze. Doopravdy jsme se ale poznali až mnohem později, když jsme na počátku devadesátých let spolupracovali na překladu manuálů k jistému dodnes populárnímu programovému prostředí. Brzy jsme zjistili, že máme jeden společný zájem – učit lidi, jak kvalitně psát programy.

V současné době dominuje při tvorbě aplikací objektově orientované programování. Moderní vývojové nástroje, které jsou na trhu k dispozici, jeho znalost předpokládají, aplikační knihovny z něj vycházejí, softwarové firmy ho vyžadují, nově vznikající programovací jazyky jsou čistě objektové. A když už jsme u těch jazyků: Java je dnes asi nejpoužívanější jazyk pro vývoj nových aplikací a zcela určitě to je jazyk, který se nejdynamičtěji rozvíjí. Přesto téměř všechny učebnice Javy, které na trhu najdete, začínají procedurálním programováním a k objektově orientovanému programování se dostanou až ke konci. Objekty pak často vypadají jako nepříliš pohodlná nadstavba nad procedurálním programováním.

Řekl jsem, že tak vypadají *téměř* všechny knihy. Kniha Rudy Pecinovského je totiž velmi příjemnou výjimkou. Je to učebnice, která objekty opravdu začíná a prvních několik kapitol se ani ničím jiným nezabývá. Teprve poté, co zvládnete základní pojmy a dovednosti objektově orientovaného programování, se začne zabývat konstrukcemi, jako je cyklus nebo podmínka. Tento postup, který si autor vyzkoušel na začínajících programátorech v programátorských kroužcích a který používá při výuce profesionálů, vás naučí od počátku myslet objektově. Ukazuje objekty jako něco opravdu přirozeného, jako něco, co výrazně usnadňuje přemýšlení o řešené úloze.

Při čtení Rudovy knihy jsem občas litoval, že už umím programovat, a tak jen doufám, že slibované další díly budou stejně dobré.

M. Virius

Úvod

Otevíráte třetí vydání knížky, která vás chce naučit programovat moderním, objektově orientovaným stylem. Stylem, jímž se v dnešní době vyvíjí drtivá většina klíčových aplikací, ale k jehož výuce ještě řada škol nedospěla. Po nastudování této knížky budou proto mnozí z vás vědět o moderním programování víc než leckterý z vašich učitelů.

Co je nového ve 3. vydání

Oproti předchozímu vydání je kniha od základů přepracovaná. Kniha je nyní rozdělena do dvou dílů. První díl se soustředí především na výklad základních architektonických principů, které by si měl čtenář osvojit předtím, než se pustí do kódování složitějších projektů. Jeho cílem je, aby čtenáři přešly tyto principy do krve dřív, než se začne soustředit na kód.

Druhý díl pak získané návyky prohloubí a seznámí čtenáře s řadou dalších programátorských technik. Vysvětlí hlubší souvislosti, na něž v běžných učebnicích již nezbývá místo, a doplní čtenářovy znalosti syntaxe jazyka. Současně učí práci s profesionálním vývojovým nástrojem a seznámí čtenáře s jeho základními vlastnostmi.

Druhý díl je možno použít jako samostatnou učebnici pro ty, kteří již zvládli základní architektonické principy návrhu objektových aplikací z jiných učebnic.

Komu je kniha určena

Tato knížka je určena těm, kteří to se svojí touhou naučit se moderně programovat myslí vážně a chtějí se naučit programovat dobře. Zaměřuje se na ty, kteří nechtějí zůstat obyčejnými kodéry, jejichž práci postupně přebírají nejrůznější nástroje, ale chtějí se propracovat mezi špičkové architekty, kteří umějí navrhnout optimální řešení splňující požadavky zákazníka (a jsou podle toho také ohodnoceni).

Knížka je sice primárně určena těm, kteří ještě nikdy neprogramovali, ale ukázalo se, že se z ní dozvědí hodně nového i poměrně zkušení programátoři. První vydání jsem psal na základě zkušeností z mnoha kurzů. Při vedení těchto kurzů jsem si uvědomil, že to, co děti (a částečně i dospělí, kteří s programováním teprve začínají) pochopí poměrně snadno, zvládají programátoři s předchozími zkušenostmi z neobjektového programování obtížně. Spoustu úsilí totiž musí věnovat

tomu, aby se nejprve odnaučili mnohé z toho, co si před tím pracně osvojili. Teprve pak začnou pomalu vstřebávat jiný způsob programátorského myšlení, jejž objektově orientované programování vyžaduje[1].

Od čtenářů předchozích vydání jsem dostal řadu mailů, v nichž mi psali, že jsou sice zkušenými programátory, ale teprve zde pochopili některé věci, které jim ostatní učebnice a kurzy nedokázaly vysvětlit. Uvědomil jsem si přitom, že knihu mohou s výhodou použít i ti, kteří již nějakou dobu programují a potřebují se přeškolit z klasického, strukturovaného programování na programování objektově orientované (80 % účastníků mých kurzů).

Tady bych ale chtěl případné zkušenější čtenáře upozornit na jedno úskalí: dospělý člověk již, na rozdíl od dětí, nedokáže přijmout nové informace, aniž by je okamžitě nespojoval se svými dosavadními znalostmi a zkušenostmi. Dozví-li se proto informaci, která neodpovídá zcela jeho dosavadním zkušenostem, tak ji podvědomě „ohne“, aby se s nimi dostala do souladu. To je nejčastějším zdrojem problémů zkušenějších programátorů při přechodu na nové paradigma, protože to, co si zapamatují, bývá v důsledku výše zmíněného „ohnutí“ něco trochu jiného, než co se jim snažil přednášející sdělit.

Co se naučíte

Musím vás upozornit na to, že kniha, kterou držíte v ruce, se od běžných učebnic poněkud liší. Ostatní učebnice jsou totiž většinou především učebnicemi nějakého programovacího jazyka. Jejich autoři se proto ve svém výkladu soustředí hlavně na výklad vlastností popisovaného jazyka a jeho knihoven. Bohužel se v nich ale nedozvíte skoro nic o tom, jak při návrhu programů přemýšlet, aby vás nezaskočily náhlé změny zadání, kterými je současné programování pověstné. Takovéto učebnice proto nevychovávají návrháře, kteří by uměli program navrhnout, ale pouze kodéry, kteří umějí zanalyzované zadání zakódovat.

Vypadá to, jako když autoři předpokládají, že se při čtení jejich knihy naučíte programovat nějak sami od sebe obdobně, jako se to museli naučit oni. Zkušenosti s programátory, kteří navštěvují moje kurzy ve firmě či na univerzitě, však ukazují, že tohoto výsledku bývá dosaženo jen zřídka. Většina účastníků mých kurzů zná poměrně dobře konstrukce nějakého objektově orientovaného programovacího jazyka. Bohužel, skoro nikdo z nich v něm neumí objektově programovat[2], neumí přepnout na objektový způsob uvažování.

---

[1] Statistiky uvádějí, že typická doba přechodu na objektové paradigma je 12 až 18 měsíců, přičemž čím je programátor zkušenější, tím déle mu přerod trvá.

[2] Výzkum z přelomu století ukázal, že pouze 10 % programů psaných v objektově orientovaných jazycích je navrženo opravdu objektově. (Goddard, D. 1994. Is it really object oriented?

Pokračování na další stránce

Tady si neodpustím vzpomínku na jednoho studenta, který mi na konci semestru vysvětloval, že už profesionálně programuje, takže si myslel, že předmět udělá hravě. Když na začátku semestru viděl, že učím jiné programování, než to, které jej učili na střední škole, řekl si, že to je pouhé hraní a že si počká, až začneme programovat tak, jak je zvyklý, a pak chybějící body hravě dožene. V průběhu semestru však zjistil, že se mu stále vzdalujeme, a když se rozhodl, že se nás pokusí dohnat, odhalil, že řadu konstrukcí, které běžně používáme, vůbec nechápe. Nezbylo mi, než mu zopakovat, že k programování, které bylo hlavním proudem v sedmdesátých letech (a které se na řadě středních škol stále učí), se nevrátím a že většina účastníků přichází do mých kurzů právě proto, aby se tento jiný způsob programátorského myšlení naučila.

Znovu proto opakuji: toto není učebnice programovacího jazyka, toto je učebnice objektového programování. Většina učebnic vás učí, jak to či ono zakódovat v daném programovacím jazyku. My se primárně nebudeme učit, jak program zapsat, ale jak jej navrhnout. Mým cílem není vychovat z čtenářů kodéry v Javě, ale připravit je na to, aby z nich mohli vyrůst schopní architekti. Nebudu vás proto učit specialitám použitého programovacího jazyka, ale pokusím se vás naučit efektivně navrhovat a vytvářet spolehlivé a snadno udržovatelné programy. Jinými slovy: chci vás naučit dovednostem, které budete používat, ať už budete programovat v jakémkoliv objektově orientovaném jazyku. Jazyky přicházejí a odcházejí. Základní programátorské techniky a způsob myšlení však žijí daleko déle než jazyky, se kterými byly zavedeny.

Díky tomu, že se místo na programovací jazyk soustředím spíše na vlastní programování, vznikl v knize prostor pro výklad řady programátorských zásad a technik, o nichž se klasické učebnice vůbec nezmiňují, a to dokonce ani učebnice pro zkušené programátory. Tyto zásady a techniky se většinou přednášejí až v nadstavbových kurzech, které učí vytvářet programy tak, aby s vámi mohly růst a aby vás nezaskočily rychle se měnící požadavky zákazníků (a připravte se na to, že tyto měnící se požadavky vás potkají i v případě, kdy vaším zákazníkem jste vy sami). Přitom vůbec nejde o techniky složité, které by začátečník nezvládl pochopit. Ostatní učebnice je pomíjejí pouze proto, že nesouvisejí přímo se syntaxí programovacího jazyka, ale týkají se obecného programování.

Data Based Advis. 12, 12 (Dec. 1994), 120-123.) Od té doby se situace trochu zlepšila, nicméně na svých školeních stále pozoruji, že strukturovaně navržené programy psané v objektových jazycích v mnoha softwarových firmách převažují.

Tato učebnice je naopak vysvětluje od samého počátku výkladu, protože vím, že byste si je měli v co největší míře osvojit ještě před tím, než se seznámíte s kódem. Ti, kteří se nejprve učí kódovat, a teprve následně se dozvídají, jak lze návrh programu zefektivnit, bývají příliš v zajetí svých zkušeností s kódováním a při návrhu programů se pak zbytečně silně soustředí na některé nepodstatné detaily.

Styl výuky

Tuto knihu sice najdete mezi doporučenou literaturou na řadě univerzit, ale není to odborným stylem psaná vysokoškolská učebnice. Naopak. Snažil jsem se ji psát tak, aby se dobře četla i středně bystrému středoškolákovi či středoškolačce a aby, jak s oblibou říkám, vykládaná látka „dobře tekla do hlavy".

Už podle tloušťky knihy jste nejspíše odhadli, že není určena těm, kteří hledají dvousetstránkovou rychloučebnici, s jejíž pomocí se naučí programovat za víkend. Jestli jste tuto knihu otevřeli, tak asi patříte k těm, kteří vědí, že taková učebnice neexistuje. Dvousetstránková knížka bude možná levná, ale může věci pouze naznačovat, takže se její čtenář dostane ke skutečnému poznání až po následném dlouhém a usilovném samostudiu. Předpokládám proto, že už víte, že tenké učebnice patří ve skutečnosti k těm nejdražším, protože to, co ušetříte při jejich koupi, mnohonásobně ztratíte při následném zdlouhavém osvojování si stručně a náznakově probrané látky.

Tato kniha se od ostatních učebnic a kurzů programování liší ještě v jedné věci: její výklad je koncipován podle metodiky *Architecture First*, která doporučuje vštípit studentům základní principy návrhu architektury před tím, než začnou psát vlastní programy. Má-li se vám dostat objektově orientované myšlení pod kůži, musíme s jeho výkladem začít hned a nezatěžovat vás napřed klasickými konstrukcemi, které by vaše myšlení směřovaly trochu jinam, takže byste se museli po chvíli zase přeorientovávat.

Zpočátku proto budeme nějakou dobu pracovat v interaktivním režimu, v němž počítači pouze vysvětlujeme, co má udělat, a počítač za nás příslušný program vytvoří. Nebudete se proto muset rozptylovat syntaktickými pravidly použitého programovacího jazyka, ale budete se moci soustředit na architekturu daného programu a vlastnosti jeho součástí. Na klasické konstrukce samozřejmě nezapomeneme, ale dojde na ně řada až v době, kdy již budete mít za sebou několik objektově orientovaných programů, které budete moci pomocí těchto konstrukcí dále vylepšovat.

Většina učebnic demonstruje vykládanou látku na jednoduchých úlohách, jejichž hlavním účelem je demonstrovat vysvětlovanou vlastnost jazyka. Takovéto programy bývají často označovány jako AHA-příklady, protože jejich hlavním účelem je, aby si student po jejich pročtení řekl: „AHA, takhle to funguje!" Bohužel, z AHA-příkladů ne vždy také pochopí, jak se daná konstrukce v praxi používá.

I když se AHA-příkladům nebudu vyhýbat, budu se vám snažit předkládat i úlohy složitější, a to i za cenu toho, že část úlohy, která bude používat doposud nevysvětlené konstrukce, za vás budu muset předem vyřešit sám. Takovéto úlohy daleko lépe odpovídají těm, s nimiž se budete v praxi setkávat. Typickou úlohou programátora totiž není navrhnout kompletní řešení nějakého jednoduchého problému, ale naopak doplnit stávající, většinou velmi složitý a někým jiným napsaný program o nějakou novou funkci. Při práci na takovýchto příkladech si navíc vyzkoušíte další potřebnou dovednost, kterou je schopnost orientovat se v programu, který je mnohem složitější, než byste sami dokázali v daném okamžiku naprogramovat.

Programovací jazyk

Sliboval jsem, že se nebudu soustředit na výuku jazyka, ale na výuku programování. Výuce programovacího jazyka se však nevyhneme. Budete-li si chtít vyzkoušet to, co jste se naučili, nezbyde vám, než program v nějakém programovacím jazyku vytvořit. Pro demonstraci látky vysvětlované v této učebnici budu používat programovací jazyk Java. Zvolil jsem jej z několika důvodů:

- V současné době je to nejrozšířenější programovací jazyk, a to jak z hlediska obliby, tak především z hlediska jeho použití (studie firmy Evans Data uvádí, že v roce 2011 používalo Javu 68 % programátorů).
- Je to moderní programovací jazyk, na němž lze demonstrovat použití všech důležitých postupů.
- Oproti jiným jazykům je doopravdy jednoduchý, takže se jej snadno naučíte (jak s oblibou říkám: je to jediný univerzální mainstreamový jazyk, který se dá zcela naučit za semestr).
- Překladač i ostatní vývojové nástroje je možné získat zdarma. Java je v současné době jediná platforma, pro niž existují profesionální, a přitom volně dostupné nástroje pro vývoj programů všech velikostí, od malých programů pro čipové karty až po rozsáhlé systémy běžící na několika vzájemně komunikujících počítačích rozmístěných po celém světě.
- Vytvořené programy nejsou omezeny na jediný operační systém, ale můžete je přenášet mezi různými operačními systémy. Znám programátorské týmy pracující na společném projektu, jejichž členové pracují na různých systémech (*Windows – Linux – MacOS*) a na všech musí vyvíjený program chodit. Java je v současnosti navíc jediná platforma, jejíž programy můžete spustit prakticky na všech telefonech.
- Je to jediný jazyk, k němuž existuje několik zdarma dostupných profesionálních vývojových nástrojů a spolu s nimi i několik nástrojů specializovaných pro výuku (některé z nich dokonce lokalizované do češtiny a slovenštiny).

Jak jsem již řekl, v této učebnici se chci soustředit spíše na to, jak programovat, a programovací jazyk používám pouze jako prostředek k tomu, abychom si mohli vysvětlené věci hned také vyzkoušet. Zkoušet ale budeme hodně, takže se v průběhu výuky naučíte nejenom programovat, ale zároveň získáte jistou praxi při řešení nejrůznějších úloh v programovacím jazyku Java.

Uspořádání

Jak jsem již naznačil, kniha vychází ve dvou dílech. První díl vás má naučit základům objektově orientovaného architektonického myšlení, druhý se pak bude soustředit spíše na techniky a technologie, s jejichž pomocí budete své geniální architektonické nápady realizovat. Přitom prohloubíme i architektonické poznatky z prvního dílu.

První díl je rozdělen do tří částí. V první části využijeme schopnosti použitého vývojového prostředí umožňujícího práci v interaktivním režimu, v němž uživatel představuje jeden z objektů aktuálního projektu, který posílá ostatním objektům různé zprávy. Využijeme toho, že v tomto režimu můžeme vytvářet programy, aniž bychom napsali jediný řádek kódu. Jenom se budeme snažit si osvojit základní principy práce objektově orientovaného programu. Použitému vývojovému prostředí vždy pouze vysvětlíme, co má program dělat, a vývojové prostředí vytvoří daný program za nás. Budeme se proto moci soustředit na klíčové vlastnosti objektového programu, aniž bychom se rozptylovali pravidly psaní kódu.

Nejprve se seznámíme s objekty a třídami objektů. Naučíme se objektům posílat zprávy a analyzovat jejich reakce. Ukážeme si, jak definovat program, aniž bychom museli zapsat nějaký kód. Podíváme se do nitra objektů a představíme si první návrhové vzory. Pak se seznámíme s rozhraním, které je základní konstrukcí moderního programování. Vysvětlíme si, kdy je vhodné je zavést a jaké výhody nám může jeho zavedení přinést. První část ukončí hovory o dědičnosti, jejích možných podobách a vlastnostech včetně představení abstraktních tříd a metod.

V druhé části budeme postupně opakovat vše, co jsme se v první části naučili. Přitom si budeme průběžně ukazovat, jak se jednotlivé konstrukce probrané v první části zapisují v kódu. Vynecháme pouze dědičnost implementace, která s sebou nese tolik záludností, že její výklad odložím až do druhého dílu. V prvním dílu ale probereme řadu jiných základních programátorských technik, které budeme potřebovat pro efektivní definice našich programů. V závěru druhé části si pak budete moci naprogramovat první samostatnou aplikaci – hru, ve které budete ve virtuálním vesmíru přistávat s UFO na předem připravené přistávací rampy.

Třetí část prohloubí některé vaše znalosti a dovednosti. Seznámíme se s pachy v kódu, předvedeme si techniky refaktorace a ukážeme si, jak co nejbezpečněji upravovat hotový kód. Na závěr se naučíte rozdělovat program do balíčků.

Většina úloh, na jejichž řešení vykládanou látku demonstruji, jsou grafické simulace, protože na nich můžete nejlépe pochopit podstatu vysvětlovaných konstrukcí. Říkal jsem si, že pro vás budou zajímavější než obsluha bankovních účtů a další praktické úlohy, s nimiž se v učebnicích často setkáváme.

Úlohy v prvním dílu budou, pravda, celkem triviální, protože toho ještě nebudete moc umět, i když i tady si zkusíte naprogramovat část jednoduché hry. Druhý díl už přinese zajímavější úlohy.

Žádná z úloh zadaných v této knize není pouze zadána. **Všechny zadané úlohy jsou vyřešeny, abyste si mohli porovnat své řešení se vzorovým** a abyste se v případě, kdy se dostanete do těžkostí a nebudete si vědět rady, měli kam obrátit pro inspiraci. Budete-li se však chtít opravdu naučit programovat, doporučuji vám vyřešit všechny tyto doplňkové úlohy vlastní hlavou.

Jazyk programů

Někteří čtenáři mne občas napadají za to, že ve svých textech používám důsledně českou terminologii a české identifikátory v programech. Za dlouhou dobu své učitelské praxe jsem si však vyzkoušel, že používání původních anglických termínů v začátečnických kurzech není dobré řešení. Začátečníci mívají problémy s pochopením vlastní látky a přidání termínů, kterým nerozumějí (znalost angličtiny u nás stále není na takové úrovni, jakou bychom rádi viděli), jim situaci pouze ztěžuje.

Praxe pak ukazuje, že mnozí „anglicky hovořící“ programátoři schovávají za anglické termíny to, že problému sami nerozumějí a při používání slangu působí na neznalé okolí jako odborníci. Když na začátečníka vybafnu např. název *singleton*, málokterý bude vědět, co to slovo znamená, a nezbyde mu, než si je zapamatovat jako nějaký nový, cizí termín. Když se pak po pár týdnech výuky zeptám, jaké vlastnosti má návrhový vzor singleton, začnou žáci nejprve tápat, který z probraných vzorů to je, a v řadě případů jej zamění s nějakým jiným.

Když naproti tomu použiji pro daný návrhový vzor termín *jedináček*, všichni si jej ihned pevně spojí se svojí představou jedináčka. Nejenom že jej pak i po týdnech správně vyloží, ale navíc i lépe pochopí jeho podstatu.

Prosím proto čtenáře, kteří jsou hrdí na svoji znalost angličtiny, aby se smířili s tím, že budu vycházet vstříc většině, která konstrukce označené českými termíny lépe pochopí a daleko lépe si je zapamatuje. Ti, kteří můj počeštěný výklad nepotřebují, se jistě již dávno poučili z některé anglicky psané učebnice.

Odmítám přistoupit na psaní programů s „ceskymi“ termíny bez diakritiky. To je zoufalá berlička těch, kteří neumějí anglicky a schovávají to za pomlouvání identifikátorů s diakritikou. Programovacímu jazyku je opravdu jedno, jestli píšete s diakritikou nebo bez ní, tak proč bychom se měli nechávat znásilňovat počítačem

a nutit se programovat tak, jak se programovalo v minulém století, kdy znalost národních abeced nebyla v programovacích jazycích běžná.

Všichni však víme, že programátorský svět je veskrze anglický. Jak s oblibou říkám: „Pracujete-li jako softwaroví vývojáři, máte jen dvě možnosti: buď se naučíte anglicky, nebo změníte zaměstnání.“ V druhém dílu již nebudete naprostí začátečníci, a proto vyměníme používanou knihovnu za její mutaci s anglickými identifikátory (vysvětlující komentáře zůstanou české) a spolu s knihovnou začneme používat anglické identifikátory i v programech v učebnici a v úlohách, které budete řešit, abyste se o svých geniálních programech mohli bavit s kýmkoliv na světě.

Potřebné vybavení

Pro úspěšné studium prvního dílu budete potřebovat čtyři věci:

- dostatečně výkonný počítač,
- základní vývojovou sadu Javy,
- vývojové prostředí *BlueJ*,
- programy používané v příkladech,
- internetový prohlížeč, který je schopen přehrát flashové animace použité pro část výkladu.

Sada JDK (Java Development Kit)

K vývoji programů budete potřebovat vývojovou sadu JDK, kterou můžete zdarma stáhnout na adrese http://www.oracle.com/technetwork/java/javase/downloads. Potřebujete ale sadu JDK 7 nebo mladší. Na starších verzích Javy naše programy pracovat nebudou. Počítejte s tím, že instalační soubory zabírají téměř 100 MB a po instalaci bude zabírat okolo 400 MB.

K vývojové sadě si nezapomeňte stáhnout a nainstalovat i dokumentaci. Najdete ji na stejné webové adrese, jenom je na stránce skoro až na konci. Je přibližně stejně rozsáhlá jako samotná vývojová sada (nebojte se, nemusíte ji číst celou), ale při jakémkoliv programování za hranicemi příkladů z učebnice se bez ní neobejdete. Musíte se však smířit s tím, že dokumentaci seženete pouze anglicky. ZIP soubor s dokumentací zabírá asi 60 MB a po rozbalení bude zabírat necelých 300 MB.

Vývojové prostředí

S JDK při vývoji programů teoreticky vystačíte (některé učebnice ani nic jiného nepoužívají), nutí vás však starat se o řadu konfiguračních detailů, které odvádějí vaši pozornost od vlastního programování. Převážná většina programátorů proto používá vývojové prostředí, které tyto detaily vyřeší za ně, a navíc jim pomůže i v řadě dalších oblastí.

Já budu při výkladu používat v prvním dílu vývojové prostředí *BlueJ*, které můžete (rovněž zdarma) stáhnout na adrese http://bluej.org/download/download.html. Ve srovnání s vlastní Javou je toto prostředí nenáročné. Instalační soubory mají necelých 7 MB a po instalaci zaberou zhruba 10 MB. V druhém dílu pak přejdeme k používání profesionálního vývojového prostředí.

Standardní instalace však nevyužívá všechny výhody, které prostředí *BlueJ* nabízí, a na druhou stranu se občas stane, že poslední publikovaná verze obsahuje chyby, které vám brání v provedení některých akcí, o nichž v učebnici píšu. Doporučuji vám proto stáhnout verzi tohoto prostředí, kterou najdete na mojí webové stránce http://vyuka.pecinovsky.cz/bluej_config/. Na této stránce najdete vedle anglické verze i verze lokalizované do češtiny a slovenštiny. Oproti původní konfiguraci budou naše programy navíc i trochu barevnější, což vám umožní se v nich lépe orientovat. Současně v nich najdete připravené šablony souborů, které budeme v učebnici vytvářet. Poslední výhodou takto zkonfigurovaných souborů je, že přepnou celé prostředí do kódování UTF-8, takže přestanete mít problémy s přenášením souborů mezi jednotlivými platformami a uživatelé používající *Windows* si budou moci vesele vyměňovat své programy s uživateli systémů *Linux* či *Macintosh*, a to nezávisle na tom, používají-li diakritiku.

Proč právě *BlueJ*

Na webových programátorských konferencích někteří programátoři kritizovali, že v učebnici nepoužívám některé profesionální vývojové prostředí. Přiznávám, že *BlueJ* se od těchto prostředí opravdu odlišuje – můžete se je naučit používat tak za 20 minut, kdežto naučit se dokonale některé profesionální vývojové prostředí vám dá víc práce, než se naučit Javu.

Hlavní výhodou prostředí *BlueJ* však není jeho jednoduchost (i když je nesmírně důležitá), ale to, že je optimalizované pro výuku ve vstupních kurzech programování. Jak jsem již řekl, tato kniha se vás nebude snažit naučit kódovat, ale bude se vás snažit naučit „myslet objektově“. K tomu je důležité, aby vám prostředí umožňovalo přemýšlet na úrovni architektury programů, a nepohybovat se pouze kdesi v suterénu na úrovni kódu.

BlueJ je v době psaní knihy jediné zdarma dostupné vývojové prostředí, které vaše zásahy do architektury okamžitě promítá do kódu, a naopak vaše zásahy do

kódu okamžitě promítne do zobrazení architektury, a které současně umožňuje práci v interaktivním režimu, v němž pouze ukazujete, co má program umět, a *BlueJ* vše naprogramuje za vás. To jiná prostředí neumožňují. Toho budeme využívat v první části, kdy se budeme soustředit na objektové vlastnosti programů, aniž bychom se museli rozptylovat nějakými pravidly pro zápis kódu.

Doprovodné programy

Na stránce knihy na adrese http://knihy.pecinovsky.cz/mojj najdete také soubor s generátorem projektů použitých v knize. Abych vyšel vstříc těm, kteří mají odpor k používání diakritiky, je pro ně na webu připraven program `Konverze`, který zkopíruje soubory do zadané složky a při tom změní jejich kódování. Při převodu do kódování ASCII zbaví jejich názvy i obsah veškeré diakritiky. Tento program je napsaný v Javě, takže by měl chodit na všech počítačích.

Doprovodné animace

Sami jistě ze zkušenosti víte, že dobrý příklad je v řadě situací lepší než řada slov. Na stránce http://vyuka.pecinovsky.cz/animace je proto připravena celá řada animovaných ukázek, v nichž krok za krokem demonstruji některé vysvětlované postupy. Tyto animace využijete hlavně při čtení první části knihy, protože demonstrují především práci v interaktivním režimu a tvorbu základních konstrukcí. Ukazují, jak tyto konstrukce použít, a předvádějí chování programů, v nichž jsou vysvětlené konstrukce použity.

Animace však nejsou navázány na konkrétní text kapitol této knihy. Vznikly jako doprovod k mým přednáškám a k jiným knihám. Vedle českých animací najdete na zmíněné stránce i řadu anglických, které zase slouží jako doprovod k mému kurzu programování, jenž běží na stránkách *NetBeans*.

Použité konvence

K tomu, abyste se v textu lépe vyznali a také abyste si vykládanou látku lépe zapamatovali, používám několik prostředků pro odlišení a zvýraznění textu.

Objekty První výskyt nějakého termínu a další texty, které chci zvýraznit, vysazuji **tučně**.

Názvy Názvy firem a jejich produktů vysazuji *kurzivou*. Kurzivou vysazuji také názvy kapitol, podkapitol a oddílů, na které se v textu odkazuji.

Citace Texty, které si můžete přečíst na displeji, např. názvy polí v dialogových oknech či názvy příkazů v nabídkách, vysazuji **tučným bezpatkovým písmem**.

Adresy Názvy souborů a internetové adresy vysazuji obyčejným bezpatkovým písmem.

`Program` Texty programů a jejich částí vysazuji `neproporcionálním písmem`.

Kromě částí textu, které považuji za důležité zvýraznit nebo alespoň odlišit od okolního textu, najdete v textu ještě řadu doplňujících poznámek a vysvětlivek. Všechny budou v jednotném rámečku, který bude označen ikonou charakterizující druh informace, kterou vám chce poznámka či vysvětlivka předat.

Symbol jin-jang bude uvozovat poznámky, s nimiž se setkáte na počátku každé kapitoly. Zde vám vždy prozradím, co se v dané kapitole naučíte.

Otevřená schránka s dopisy označuje informace o projektu, s nímž budeme v dalším textu pracovat, nebo v němž najdete vzorové řešení.

Obrázek knihy označuje poznámku týkající se používané terminologie. Většinou upozorňuje na další používané termíny nebo na konvence, které se k probírané problematice vztahují. Seznam všech terminologických poznámek najdete v rejstříku pod heslem „terminologie“.

Obrázek počítače označuje zadání úkolu, který máte samostatně vypracovat. Seznam všech úloh najdete v rejstříku pod heslem „úloha“.

Píšící ruka označuje obyčejnou poznámku, která pouze doplňuje informace z hlavního proudu výkladu o nějakou zajímavost.

Ruka s hrozícím prstem upozorňuje na věci, na které byste si měli dát pozor, protože jejich zanedbání vás většinou dostane do problémů.

Usměváček vás bude upozorňovat na různé tipy, kterými můžete vylepšit svůj program nebo zefektivnit svoji práci.

Mračoun vás naopak bude upozorňovat na různá úskalí programovacího jazyka nebo programů, s nimiž budeme pracovat, a bude vám radit, jak se těmto nástrahám vyhnout či jak to zařídit, aby vám alespoň pokud možno nevadily.

Brýle označují tzv. „poznámky pro šťouraly“, ve kterých se vás snažím seznámit s některými zajímavými vlastnostmi probírané konstrukce nebo upozorňuji na některé souvislosti, avšak které nejsou k pochopení látky nezbytné.

Symbol znamení raka označuje poznámku popisující doprovodnou animaci (animovaný výklad), která se vás bude snažit naučit nějakou dovednost, která se špatně popisuje slovy.

Odbočka

Občas je potřeba vysvětlit něco, co nezapadá přímo do okolního textu. V takových případech používám podšeděný blok se silnou čarou po straně. Tento podšeděný blok je takovou drobnou odbočkou od ostatního výkladu. Nadpis podšeděného bloku pak najdete i v podrobném obsahu mezi nečíslovanými nadpisy.

Vaše poznámky a připomínky

Kniha vznikala pomalu a dlouho. Přes veškeré úsilí, které jsme jí já i moji spolupracovníci věnovali, nemohu vyloučit, že v textu či doprovodných příkladech zůstaly skryté nějaké chyby. Předem se za ně omlouvám a prosím vás, abyste mi o nich napsali na adresu rudolf@pecinovsky.cz. Já bych je opravil a opravu vystavil na webové stránce této knihy http://knihy.pecinovsky.cz/mojj.

Budete-li mít jakékoliv dotazy k vysvětlovanému textu nebo budete-li mít s něčím problémy, využijte konference na adrese http://www.pandora.cz/list/vyuka_oop. Budete-li do ní chtít poslat příspěvek, musíte se na serveru nejprve zaregistrovat. Číst cizí příspěvky však můžete i bez registrace.

Část 1: Interaktivní režim

Kapitola 1
Seznamujeme se s nástroji

Co se v kapitole naučíte

V této kapitole se seznámíte s nástroji, které budete při studiu dalších částí knihy potřebovat. Nejprve vám povím o historii a současných trendech v programování a prozradím vám, proč jsem pro výuku vybral programovací jazyk Java. Pak vám ukážu vývojové prostředí *BlueJ* a naznačím, jak je máte nainstalovat na svůj počítač. Poté vám vysvětlím, jak jsou organizovány doprovodné projekty k učebnici, a ukážu vám, jak je připravit, abyste s nimi mohli v průběhu studia učebnice snadno pracovat. Na závěr vám předvedu, jak otevřít projekt ve vývojovém prostředí *BlueJ*, prozradím vám, co je to diagram tříd, a naučím vás, jak je možno tento diagram v prostředí *BlueJ* upravovat.

1.1 Trocha historie

První počítače

Historie počítačů (tj. strojů, u nichž je postup výpočtu řízen programem) a programování se začala psát již v devatenáctém století. Charles Babbage tehdy dostal zakázku od anglického námořnictva na vylepšení svého počítacího stroje pro výpočet navigačních tabulek, které se hemžily chybami. V roce 1848 dokončil návrh počítacího stroje, který byl řízený programem zadávaným na děrných štítcích. Zbytek života se pak věnoval jeho konstrukci. Stroj byl mechanický a měl být poháněn parním strojem, ale z programátorského hlediska již umožňoval značnou část operací, kterými se honosí současné počítače.

Zajímavé je, že prvním publikujícím programátorem byla žena – Augusta Ada Lovelance (mimochodem dcera známého romantického básníka lorda Byrona), která se s Babbagem spřátelila a pomáhala mu s některými výpočty. Jednou se na výletě v Itálii setkala s poručíkem Menabreou (pozdějším generálem a předsedou vlády sjednocené Itálie), který pro místní smetánku přednášel o Babbageově stroji. Aby Babbageův stroj zpopularizovala, přeložila přednášku, která popisovala hardware tohoto počítače, do angličtiny a na Babbageův popud překlad doplnila „poznámkami překladatele", kde vysvětlila možnosti stroje ze softwarového hlediska. Svůj výklad doplnila i krátkým programem na výpočet Bernoulliho čísel – prvním publikovaným počítačovým programem na světě[3]. (Na její počest byl v osmdesátých letech minulého století pojmenován programovací jazyk Ada.)

Babbageovi se jeho počítač nepodařilo rozchodit, protože vyžadoval součástky z materiálu s pevností pancéřové oceli opracované s hodinářskou přesností, a to bylo pro tehdejší technologii velké sousto[4]. Rozchodil jej až jeho vnuk, který byl americkým generálem a při odchodu do penze se rozhodl, že všem ukáže, že dědeček nebyl blázen a jeho stroj by byl schopen provozu.

První funkční počítač postavil až v roce 1938 v Německu Konrád Zuse. Nabídl jej armádě, ale tehdejší německá generalita prohlásila, že takový nesmysl nebude armáda nikdy potřebovat. Bůh ví, kam by se vývoj ubíral, kdyby generálové nebyli tak krátkozrací a počítač pomáhal německým inženýrům za druhé světové války při výpočtech.

Další počítače se objevily na konci druhé světové války. Nejslavnějším z nich byl ENIAC, který byl vyroben v roce 1944 a byl prvním čistě elektronickým počítačem (ostatní ještě používaly relé[5], či dokonce mechanické prvky). Skutečné počítačové (a tím i programátorské) orgie však začaly až v padesátých letech, kdy se začaly počítače vyrábět sériově a počítačová věda (computer science) se zabydlela na všech univerzitách.

Problémem prvních počítačů byla jejich vysoká poruchovost. Antonín Svoboda tehdy přišel s řadou konstrukčních vylepšení umožňujících vytvořit z nespolehli-

---

[3] Poznámky překladatele doplnila až po Babbageově přemlouvání. Babbage ji chtěl ušetřit práci a program pro ni napsal, jenže ona mu v něm našla chybu a definitivní, publikovaná verze je již její dílo. Není tedy prvním člověkem, který program napsal, ale je prvním člověkem, který svůj program publikoval.

[4] Takhle se to alespoň dlouho tvrdilo. V roce 1991 však Doron Swade, kurátor londýnského muzea vědy, sestavil Babaggeův počítač pouze s použitím technologií dostupných v polovině devatenáctého století.

[5] Relé je elektromechanický prvek, kde průchod proudu cívkou zapříčiní sepnutí nebo rozpojení kontaktů. Rychlá relé dokázala sepnout i 100krát za sekundu – to byla také maximální rychlost tehdejších počítačů.

vých součástek spolehlivý počítač[6]. Konstruktéři se postupně naučili vytvářet vysoce spolehlivé počítače. Nejslabším článkem celého komplexu se tak stával program.

Co je to program

Program bychom mohli charakterizovat jako v nějakém programovacím jazyku zapsaný předpis, popisující, jak má procesor, pro nějž je program určen (v našem případě počítač), splnit zadanou úlohu.

Cílovým procesorem nemusí být vždy počítač. Oblíbeným příkladem programů jsou např. kuchařské předpisy. V předpočítačových dobách zaměstnávaly některé instituce (např. armáda) velké skupiny počtářů[7], které na mechanických kalkulačkách řešily podle zadaných programů výpočty, které dnes předhazujeme počítači.

Na programu je důležité, že musí být napsán v nějakém programovacím jazyku, kterému rozumí programátor. Napsaný program je pak jiným programem (překladačem) převeden do „jazyka“, kterému rozumí počítač.

Vybrat jazyk pro kuchařský předpis je jednoduché, vybrat jazyk pro počítač je mnohem složitější. Vlastnosti použitého jazyka totiž naprosto zásadně ovlivňují jak rychlost vývoje programu, tak i rychlost a kvalitu výsledných programů. Proto také prošly programovací jazyky i metodiky jejich používání celou řadou revolučních změn.

Program musí být především spolehlivý

Počítače byly postupně nasazovány v dalších a dalších oblastech a programátoři pro ně vytvářeli dokonalejší a dokonalejší programy. Programy byly čím dál rafinovanější a složitější, a to začalo vyvolávat velké problémy. Programátoři totiž přestávali být schopni své programy rozchodit, a když je vítězně rozchodili, nedokázali z nich v rozumném čase odstranit chyby, které uživatelé v programu objevili.

Tato krize vedla k zavádění nejrůznějších metodik, které měly jediný cíl: pomoci programátorům psát spolehlivé a snadno upravovatelné programy. V padesátých letech minulého století se tak prosadily vyšší programovací jazyky, v šedesátých

---

[6] V tehdejší době (tj. v padesátých letech) byla naše republika na špičce světového vývoje počítačů. Svobodovy myšlenky byly později uplatněny v počítačích řídících americké kosmické lodě. Jenže to už bylo v době, kdy byl z naší republiky vypuzen, emigroval do USA a tam působil jako špičkový počítačový odborník.

[7] Přiznejme si, že to tehdy byly většinou počtářky, protože muži dělají při práci podobného druhů příliš mnoho chyb. Takovéto „lidské počítače“ pomáhali např. v Sovětském svazu s vývojem atomové pumy či prvními lety do vesmíru.

letech modulární programování, v sedmdesátých letech na ně navázalo strukturované programování a v průběhu osmdesátých a zejména pak devadesátých let ovládlo programátorský svět objektově orientované programování, jehož vláda pokračuje dodnes. (Jednotlivé termíny si za chvíli probereme podrobněji.)

Hlavním cílem programátorů v počátcích programování bylo, aby jejich programy spotřebovaly co nejméně paměti a byly co nejrychlejší. Tehdejší počítače totiž měly paměti málo, byly z dnešního hlediska velice pomalé a jejich strojový čas byl drahý. Se stoupající složitostí programů však byly takto psané programy stále méně stabilní a stále hůře udržovatelné. Současně s tím, jak klesala cena počítačů, jejich strojového času i paměti, začínal být nejdražším článkem v celém vývoji člověk.

Cena strojového času a dalších prostředků spotřebovaných za dobu života programu začínala být pouze zlomkem ceny, kterou bylo nutno zaplatit za jeho návrh, zakódování, odladění a následnou údržbu. Začal být proto kladen stále větší důraz na produktivitu programátorů i za cenu snížení efektivity výsledného programu.

Prakticky každý program zaznamená během svého života řadu změn. Požadavky zákazníka na to, co má program umět, se většinou průběžně mění, a program je proto nutno průběžně upravovat, rozšiřovat a vylepšovat. Celé současné programování je proto vedeno snahou psát programy nejenom tak, aby pracovaly efektivně, tj. rychle a s minimální spotřebou různých zdrojů (operační paměť, prostor na disku, kapacita sítě atd.), ale aby je také bylo možno kdykoliv jednoduše upravit a vylepšit.

Předchozí zásady krásně shrnul Martin Fowler ve své knize Refactoring:

„Napsat program, kterému porozumí počítač, umí i hlupák.
Dobrý programátor píše programy, kterým porozumí i člověk.“

Mějte při tvorbě svých programů tuto větu neustále na paměti.

1.2 Objektově orientované programování – OOP

Vývoj metodik programování

Jak jsem řekl, v průběhu doby se prosadilo několik metodik, které doporučovaly, jak programovat, abychom byli s programem co nejdříve hotovi a výsledný program byl co nejkvalitnější. Tyto metodiky na sebe postupně navazovaly. Každá z nich těžila ze zkušeností získaných při aplikaci předchozí metodiky a stavěla na nich.

První revolucí bylo zavedení vyšších programovacích jazyků, které programátory osvobodily od zapisování programů v podobě, která přesně odpovídala použitým instrukcím použitého počítače, a zavedly vyjadřování, které bylo daleko bližší zvyklostem lidí. To pak umožnilo vědcům a technikům nečekat s každou prkotinou na programátora a naprogramovat si spoustu jednodušších výpočtů sami.

Zavedení vyšších programovacích jazyků umožnilo tvorbu složitějších programů. Vyvíjené programy byly zanedlouho tak složité, že se v nich programátoři přestali vyznávat. Modulární programování ukazovalo, že rychlost vývoje i kvalitu výsledného programu zvýšíme, když vhodně rozdělíme velký projekt do sady menších, rozumně samostatných modulů.

Produktivita programátorské práce se zvýšila, ale záhy se začalo ukazovat, že programátoři mají problémy nejenom s rozchozením obřích programů, ale i jednotlivých modulů. V roce 1968 publikoval E. W. Dijkstra článek[8], který kritizoval v té době převažující způsob psaní programů a prosazoval tzv. *strukturované programování*. Jeho článek rozpoutal obrovské diskuse, které krásně charakterizoval jiný článek nazvaný *Real Programmers Don't Use Pascal*[9] (v češtině vyšel pod názvem *Opravdoví programátoři nepoužívají Pascal*[10]) a publikovaný v červnu 1983 v časopise *Datamation*.

Strukturované programování se ponořilo do hloubky kódu a ukazovalo, že dalšího zvýšení produktivity vývoje i kvality výsledných programů dosáhneme dodržením několika jednoduchých zásad při vlastním psaní kódu. Jeho hlasatelé předváděli, že opuštěním nejrůznějších fint, kterými se programátoři snažili o optimalizaci kódu, a maximálním zpřehledněním vytvářeného programu dosáhneme nejenom vyšší produktivity programátora, ale v mnoha případech i vyšší efektivity výsledného programu.

Se stále rostoucí složitostí vyvíjených programů se začalo ukazovat, že jednou z největších překážek v efektivní tvorbě kvalitních programů je tzv. *sémantická mezera* mezi tím, co chceme vytvořit a tím, co máme k dispozici. Naše programy mají řešit široké spektrum úkolů od řízení mikrovlnné trouby přes nejrůznější kancelářské a grafické programy a hry až po složité vědecké úlohy, kosmické lety či protivzdušnou obranu kontinentu. Ve svém rozletu jsme ale odkázáni na stroje, které si umějí pouze hrát s nulami a jedničkami. Čím budou naše vyjadřovací možnosti blíže zpracovávané skutečnosti, tím rychleji a lépe dokážeme naše programy navrhnout, zprovoznit a udržovat.

---

[8] Dijkstra, E. W. "*Letters to the editor: go to statement considered harmful*". Communications of the ACM 11 (3): 147–148. DOI:10.1145/362929.362947. ISSN 0001-0782.

[9] Najdete jej např. na adrese http://www.webcitation.org/659yh1oSh.

[10] Přeloženou verzi článku najdete např. na http://www.logix.cz/michal/humornik/Pojidaci.Kolacu.xp.

Jinými slovy: Kvalita programu a rychlost jeho tvorby je velice úzce svázána s hladinou abstrakce, kterou při jejich tvorbě používáme. Budeme-li např. programovat ovládání robota, bude pro nás výhodnější programovací jazyk, v němž můžeme zadat příkazy typu „zvedni pravou ruku“, než jazyk, v němž musíme vše vyjadřovat pomocí strojových instrukcí typu „dej do registru A dvojku a obsah registru A pak pošli na port 27“.

Objektově orientované programování (v dalším textu budu využívat zkratku OOP) se proto obrátilo do vyšších hladin abstrakce a ukázalo, že vhodným zvýšením abstrakce dokážeme rychle a spolehlivě navrhovat a vyvíjet ještě větší a komplikovanější projekty. Studie dokonce ukazují, že programy obsahující více jak cca 100 000 příkazů již není v lidských silách naprogramovat dostatečně kvalitně a efektivně.

OOP přichází s výrazovými prostředky, které nám umožňují maximálně zvýšit hladinu abstrakce, na které se „bavíme“ s našimi programy, a tím maximálně zmenšit onu sémantickou mezeru mezi tím, co máme k dispozici a co bychom potřebovali. Umožňuje programovat efektivněji a vytvářet spolehlivější programy. Chceme-li však jeho výhod plně využít, nesmíme zapomínat na nic z toho, s čím přišly předchozí metodiky.

OOP je považováno za hlavní proud současného programování. Všechny moderní programovací jazyky se honosí přídomkem „objektově orientované“. Tím se nám snaží naznačit, že nabízejí konstrukce, které nám umožní programovat objektově. Podpora OOP byla postupně doplněna i do těch jazyků, které vznikly dávno před tím, než se začalo o nějakém OOP hovořit. Prakticky žádný z nově vznikajících jazyků si již nedovolí nepodporovat OOP.

1.3 Překladače, interprety, platformy

Tato podkapitola je určena těm, kteří se nespokojí jen s tím, že věci fungují, ale chtějí také vědět, jak fungují. Naznačíme si v ní, jak je to v počítači zařízeno, že programy pracují.

Operační systém a platforma

Operační systém je sada programů, jejímž úkolem je zařídit, aby počítač co nejlépe sloužil zadanému účelu. Operační systémy osobních počítačů se snaží poskytnout co největší komfort a funkčnost jak lidským uživatelům, tak programům, které operační systém nebo tito uživatelé spouští. (Teď nehodnotím, jak se jim to daří.)

Operační systém se snaží uživatele odstínit od hardwaru použitého počítače. Uživatel může střídat počítače, avšak dokud bude na všech stejný operační systém, bude si se všemi rozumět.

Při obsluze lidského uživatele to má operační systém jednoduché: člověk komunikuje s počítačem pomocí klávesnice, obrazovky, myši a případně několika dalších zařízení. Ty všechny může operační systém převzít do své správy a zabezpečit, aby se nejrůznější počítače chovaly vůči uživateli stejně.

U programů to má ale složitější. Programy totiž potřebují komunikovat nejenom s operačním systémem (např. když chtějí něco přečíst z disku nebo na něj zapsat), ale také přímo s procesorem, kterému potřebují předat své instrukce k vykonání. Problémem ale je, že různé procesory rozumí různým sadám instrukcí.

Abychom věděli, že náš program na počítači správně poběží, musíme vědět, že počítač bude rozumět té správné sadě instrukcí a že na něm poběží ten správný operační systém. Kombinaci *operační systém + použitý hardware* budeme v dalším textu označovat jako **platforma**.

Nejrozšířenější platformou současných osobních počítačů je operační systém *Windows* na počítačích s procesory kompatibilními s procesorem *Intel Pentium*. Další vám známou platformou bude nejspíš operační systém *Linux* na počítačích s procesory kompatibilními s Pentiem. Oba zmíněné operační systémy však mají své verze běžící na počítačích s jinými procesory. Třetím do mariáše, který se v poslední době stále více prosazuje i v naší republice, je systém *MacOS* počítačů *Apple*.

Programovací jazyky

Jak asi všichni víte, pro zápis programů používáme nejrůznější programovací jazyky. Ty jsou vymýšleny tak, aby v nich mohl člověk co nejlépe popsat svoji představu o tom, jak má počítač splnit požadovanou úlohu.

Program zapsaný v programovacím jazyku pak musíme nějakým způsobem převést do podoby, které porozumí počítač. Podle způsobu, jakým postupujeme, dělíme programy na překládané a interpretované.

U **překládaných programů** se musí napsaný program nejprve předat zvláštnímu programu nazývanému překladač (někdo dává přednost termínu kompilátor), který jej přeloží (zkompiluje), tj. převede jej do podoby, s níž si již daná platforma ví rady, tj. musí jej přeložit do kódu příslušného procesoru a používat instrukce, kterým rozumí použitý operační systém. Přeložený program pak můžeme kdykoliv na požádání spustit.

Naproti tomu **interpretovaný program** předáváme v podobě, v jaké jej programátor vytvořil, programu označovanému jako interpret. Ten obdržený program prochází a ihned jej také provádí.

Výhodou překládaných programů je, že většinou běží výrazně rychleji, protože u interpretovaných programů musí interpret vždy nejprve přečíst kus programu, zjistit, co má udělat, a teprve pak může tento požadavek vykonat.

Výhodou interpretovaných programů bývá na druhou stranu to, že jim většinou tak moc nezáleží na tom, na jaké platformě běží. Stačí, když na daném počítači běží potřebný interpret. Mohli bychom říci, že platformou těchto programů je právě onen interpret. Vytvoříte-li pro nový počítač interpret, můžete na něj vzápětí přenést i všechny vytvořené programy. Kdykoliv tyto programy po systému něco chtějí, požádají o to svoji platformu (interpret) a ta jim příslušné služby zprostředkuje. Takovým programům pak může být jedno, na jakém procesoru a pod jakým operačním systémem běží, protože se beztak „baví" pouze se svou platformou.

Naproti tomu překládané programy se většinou musí pro každou platformu trochu (nebo také hodně) upravit a znovu přeložit. Při implementaci programu pro více platforem bývá pracnost přizpůsobení programu jednotlivým platformám srovnatelná s pracností vývoje jeho první verze.

Vedle těchto základních druhů programů existují ještě **hybridní programy**, které jsou současně překládané i interpretované a které se snaží sloučit výhody obou skupin. Hybridní program se nejprve přeloží do jakéhosi mezijazyka, který je vymyšlen tak, aby jej bylo možno co nejrychleji interpretovat. Takto přeložený program je potom interpretován speciálním interpretem označovaným často jako virtuální stroj[11].

Hybridní programy spojují výhody obou kategorií. K tomu, aby v nich napsané programy mohly běžet na různých platformách, stačí pro každou platformu vyvinout potřebný virtuální stroj. Ten pak vytváří vyšší, mnohem univerzálnější platformu. Je-li tento virtuální stroj dostatečně „chytrý" (a to jsou v současné době prakticky všechny), dokáže odhalit často se opakující části kódu a někam stranou je přeložit, aby je nemusel pořád kolem dokola interpretovat.

Na překládané, interpretované a hybridní bychom měli dělit programy, avšak často se takto dělí i programovací jazyky. Je sice pravda, že to, zda bude program překládaný, interpretovaný nebo hybridní není závislé na použitém jazyku, ale je to především záležitostí implementace daného jazyka, nicméně každý z jazyků má svoji typickou implementaci, podle které je pak zařazován.

Prakticky všechny jazyky sice mohou být implementovány všemi třemi způsoby a řada jich opravdu ve všech třech podobách existuje (jako příklad bychom mohli uvést jazyky Basic, Java nebo Pascal), ale u většiny převažuje typická implementace natolik výrazně, že se o těch ostatních prakticky nemluví. Z vyjmenované trojice je např. klasický Basic považován za interpretovaný jazyk, Java za hybridní a Pascal za překládaný.

[11] Virtuální stroj se mu říká proto, že se vůči programu v mezijazyku chová obdobně, jako se chová procesor vůči programu v čistém strojovém kódu.

Hybridní implementace jazyků se v posledních letech výrazně prosadily a jsou dnes králi programátorského světa. Vyvíjí v nich převážná většina programátorů a procento implementací v těchto jazycích neustále vzrůstá.

Ty nejchytřejší virtuální stroje dokonce sledují, co program dělá, a zjistí-li, že je to výhodné, přeloží rychle část programu znovu tak, aby využila některých právě platících speciálních podmínek, a mohla běžet ještě rychleji. Na těchto strojích pak může běžet hybridní program skoro stejně rychle jako překládaný a v některých speciálních případech dokonce i rychleji. Přitom si stále udržuje svoji nezávislost na hardwarové platformě a použitém operačním systému.

1.4 Java a její zvláštnosti

O splnění zásad objektově orientovaného programování se snaží tzv. objektově orientované jazyky. Mezi nimi je v současné době nejpopulárnější programovací jazyk Java, který se narodil v roce 1995 a hned po svém vzniku zaznamenal velký ohlas. Ještě v průběhu devadesátých let se stal nejpoužívanějším programovacím jazykem a tuto pozici si od té doby stále udržuje. V současné době v něm vyvíjí (alespoň podle firmy Oracle) přes 9 miliónů programátorů. Na většině univerzit se Java používá jako vstupní jazyk pro výuku programování.

Klíčové vlastnosti Javy

Jaké jsou vlastnosti tohoto programovacího jazyka, že v tak krátké chvíli tak významně ovlivnil programátorský svět? Důvodů je celá řada. Zde uvedu jen ty z nich, které se výhodně projeví při výuce programování.

Objektově orientovaná

Java plně podporuje objektově orientované programování. Na rozdíl od C++ a dalších jazyků „klasického ražení“ již neumožňuje napsat program, který by nebyl (alespoň formálně) objektově orientovaný. Její autoři se přitom do ní snažili začlenit převážnou většinu zásad objektově orientovaného programování.

Předchozí tvrzení musím trochu zlehčit. Platí obecná zásada, že jako čuně mohu programovat v jakémkoliv programovacím jazyku. To, že překladač nedovoluje zapsat program, který by nebyl alespoň formálně objektově orientovaný, ještě nic neříká o tom, bude-li objektově orientovaný doopravdy, tj. i svou architekturou a duchem. To už záleží na programátorovi, to za vás žádný programovací jazyk neudělá.

Jednoduchá

Java je velice jednoduchý jazyk. Základy jeho syntaxe může člověk, který umí programovat, zvládnout během několika hodin. Pak už se může soustředit na poznání a pochopení funkce klíčových knihoven a na jejich co nejefektivnější využití.

Jednoduchost jazyka však neznamená jeho omezenost. Jak jsem již řekl, Java je v současnosti nejpoužívanějším programovacím jazykem. Programují se v ní aplikace všech rozměrů od drobných programů pro čipové karty, přes programy pro mobilní telefony a nejrůznější zabudovaná zařízení, až po obří projekty rozprostřené na řadě vzájemně komunikujících počítačů.

Multiplatformní

Java se řadí mezi hybridní jazyky. To znamená, že je současně překládaná i interpretovaná. Programy v jazyku Java se nejprve přeloží do speciálního tvaru nazývaného *bajtkód*, který pak analyzuje a interpretuje speciální program nazývaný *virtuální stroj Javy* (Java Virtual Machine – používá se pro něj zkratka VM).

Virtuální stroj umožňuje, aby jeden a týž program běhal na různých počítačích a operačních systémech. Pro každou konfiguraci hardwaru a operačního systému lze definovat její vlastní virtuální stroj. Ten se pak stará o správný běh programů, takže se program (a s ním i uživatel) vůbec nemusí starat o to, na jakém hardwaru a operačním systému zrovna běží.

Java běhá pod systémy *Windows*, *Unix*, *Linux*, *MacOS*, *Solaris* a řadou dalších operačních systémů. Vytvoříte-li program v Javě, můžete jej spustit téměř na libovolném počítači. Jak mnozí z vás vědí, programy v Javě je možné spouštět i na řadě mobilních telefonů, a dokonce ovládají i miliony čipových karet.

Java je jazyk i platforma

Jedním z klíčových záměrů autorů Javy bylo vytvořit nejenom jazyk, ale celou platformu. Tuto platformu realizuje výše zmíněný virtuální stroj spolu se základní knihovnou nejpoužívanějších funkcí.

Asi bych se zde měl zmínit o tom, že Java není jediná platforma, ale jsou to hned čtyři platformy:

- J2SE (Java 2 Standard Edition) označuje základní platformu určenou pro vývoj desktopových aplikací a jednodušších verzí serverových aplikací. Tuto edici budeme používat i my.
- J2EE (Java 2 Enterprise Edition) označuje nadstavbu nad J2SE obsahující další knihovny specializované pro tvorbu rozsáhlých distribuovaných aplikací.
- J2ME (Java 2 Micro Edition) označuje zjednodušenou verzi určenou pro vývoj aplikací pro malá zařízení, jejichž typickými představiteli jsou mobilní telefony.

- Java Card je ještě osekanější verze používaná při vývoji aplikací určených pro čipové karty. Tato platforma bývá někdy zařazována pod J2ME.

Marketingové oddělení firmy Sun se při uvádění Javy rozhodlo pojmenovat jazyk i platformu stejně (obávám se, že ani netušili, že jsou to dvě různé věci). Z toho ale vycházela řada nedorozumění. Musíme vždy rozlišovat, kdy hovoříme o platformě a kdy o jazyku. Na platformě Java běží nepřeberné množství nejrůznějších programovacích jazyků[12]. Jazyk Java je pouze jedním z nich. Programy napsané v různých jazycích jsou však většinou schopny komunikovat a spolupracovat, takže můžete část svého programu napsat např. v Javě a druhou část v nějakém jazyku, který je pro daný úkol výhodnější.

Vývojářská sada

Jako vývojáři musíte rozlišovat dvě verze programových balíků, které je možno stáhnout:

- Jednodušší **JRE** (Java Runtime Environment – běhové prostředí Javy) poskytuje vše potřebné pro běh programů.
- Komplexnější **JDK** (Java Development Kit – sada pro vývoj v Javě) označovaná někdy také SDK (Software Development Kit – sada pro vývoj softwaru) je vlastně sada JRE doplněná o základní vývojové nástroje (překladač, generátor dokumentace, ladící program a další) a poskytuje tak vše potřebné pro vývoj programů.

Vy se chcete naučit pomocí této učebnice programovat, tak budete potřebovat mít instalované JDK (připomínám, že se musí jednat o verzi 7 či vyšší). Uživatelům, kteří pak budou spouštět vaše programy, stačí JRE. V dalším textu budu předpokládat, že máte JDK nainstalováno podle pokynů v pasáži *Sada JDK (Java Development* Kit) na straně 26.

1.5 Vývojové prostředí *BlueJ*

Mnozí autoři učebnic vystačí při výkladu s nástroji z JDK doplněnými o nějaký editor, ve kterém píší zdrojové texty programů. Dříve obdobným způsobem pracovala i nemalá část programátorů, ale v současné době jsou to již pouze výjimky.

[12] Jistou představu můžete získat na stránce http://en.wikipedia.org/wiki/List_of_JVM_languages, další možností je např. stránka http://www.is-research.de/info/vmlanguages/category/jvm-language/. Na obou najdete výběr jazyků vyvinutých pro tuto platformu, ale nejsou to zdaleka všechny.

Většina programátorů dává přednost speciálním vývojovým prostředím označovaným souhrnně zkratkou **IDE** – Integrated Development Environment (integrované vývojové prostředí), což je sada programů, která nabízí řadu funkcí, jež vývoj programů výrazně zefektivňují.

IDE dáme přednost i my. V této učebnici budu používat vývojové prostředí *BlueJ* (čtěte blúdžej), které bylo vyvinuto speciálně pro výuku objektově orientovaného programování a oproti jiným IDE nabízí několik vlastností, jež se nám budou při výkladu základů OOP hodit.

- ☞ Je maximálně jednoduché, takže se začátečníci mohou soustředit na své programy a nejsou rozptylováni záplavou možností, kterými je zahrnují profesionální vývojová prostředí a které na počátku beztak neumí využít.
- ☞ Je názorné, protože slučuje možnosti klasického textového zápisu programů s možností definice jeho architektury v grafickém prostředí. Tato koncepce se stává základním postupem návrhu programů. *BlueJ* je schopno vytvořit na základě grafického návrhu kostru programu a průběžně zanášet změny v programu do jeho grafické podoby a naopak změny v grafickém návrhu do textové podoby programu, aniž by ovlivnilo ty části programu, které programátor zadal „ručně“.
- ☞ Je interaktivní – umožňuje přímou práci s objekty. Ostatní prostředí vám většinou povolují pouze vytvořit program, který můžete spustit. Přímou komunikaci s jednotlivými součástmi programu, tj. vytváření objektů a zasílání zpráv řízené interaktivně uživatelem, však většinou neumožňují.

K používání tohoto prostředí ve vstupních kurzech programování se veřejně hlásí více než tisícovka univerzit a školicích středisek po celém světě. Jeho výhodné vlastnosti přivedly řadu programátorů k tomu, že je začali používat i při vývoji profesionálních aplikací. Tito programátoři oceňují zejména jeho malou paměťovou náročnost a originální, jinde nenabízené možnosti interaktivního vývoje.

V dalším textu budu předpokládat, že máte toto prostředí nainstalováno ze stránek zmiňovaných v pasáži *Vývojové prostředí* na straně 27.

1.6 Projekty a *BlueJ*

V současných vývojových prostředích již nepracujeme s programy, ale s projekty. Projekt může obsahovat jeden program (tj. něco, co spustíme a ono to běží) nebo několik programů – to podle toho, jak se nám to zrovna hodí.

Doporučuji vám, abyste si pro projekty zřídili novou složku. Do ní pak budete umisťovat jak svoje projekty, tak projekty, které budou součástí tohoto seriálu.

Windows a substituované disky

V operačním systému *Windows* můžete používat 26 logických disků – pro každé písmeno abecedy jeden. Většina uživatelů však používá pouze zlomek tohoto počtu. Disky A: a B: bývaly donedávna vyhrazeny pro mechaniky pružných disků, ale v současné době jsou většinou nepoužívané. Na disku C: má většina uživatelů operační systém. Pak na počítači najdete ještě pár dalších jednotek vyhrazených pro další oddíly velkého disku, pro DVD a případně ještě nějaké síťové disky či zařízení, která se vůči počítači tváří jako další disk (např. paměť Flash) a tím výčet končí. Průměrný uživatel tak má většinu písmen abecedy nevyužitých.

Operační systém umožňuje použít volná písmena pro tzv. **substituované disky**, což jsou složky, které se rozhodnete vydávat za logický disk. Protože o této možnosti většina uživatelů neví, a přitom je to funkce velice užitečná, ukážu vám, jak ji můžete využít.

Substituované disky se definují pomocí příkazu:

`SUBST` *název_disku substituovaná_složka*

Nejjednodušší způsob, jak definovat ve *Windows* např. substituovaný disk `J:`, je vložit do složky, kterou budete chtít substituovat jako disk `J:`, dávkový soubor s příkazem k substituci. (Písmeno `J` se pro Javu hodí nejlépe, ale můžete si vybrat jakékoliv jiné.)

Pokud jste ještě nepracovali s dávkovými soubory, tak věřte, že to jsou obyčejné textové soubory, do nichž zapisujete příkazy pro operační systém. Jejich název může být libovolný, ale musí mít příponu bat (zkratka ze slova batch – dávka).

V dávkovém souboru budou následující příkazy (na velikosti písmen nezáleží):

```
SUBST J: /d
SUBST J: .
```

První příkaz má za úkol zrušit případnou doposud nastavenou substituci disku `J:` (není-li v daném okamžiku označený disk substituován, systém vypíše chybovou zprávu, ale jinak se nic nestane), druhý příkaz pak substituuje aktuální složku jako disk `J:`.

Soubor umístíte do složky, z níž budete chtít udělat substituovaný disk. Kdykoliv pak tento dávkový soubor spustíte, dávka substituuje složku, v níž je umístěna, jako příslušný disk. Dávka se přitom spouští obdobně jako aplikace – např. poklepáním na ikonu jejího souboru v *Průzkumníku*.

Kdykoliv od této chvíle budete pracovat s diskem `J:`, budete ve skutečnosti pracovat s obsahem substituované složky. A naopak: cokoliv uděláte s obsahem substituované složky, uděláte zároveň s obsahem disku `J:`.

Budete-li chtít mít danou substituci nastavenu trvale, můžete umístit zástupce dávkového souboru do položky Po spuštění ve startovní nabídce. Protože je v každé verzi operačního systému jinde, bude nejlepší, když ji ve startovní nabídce najdete, klepnete na ni pravým tlačítkem a v následně otevřené místní nabídce zadáte **Otevřít**. Tím otevřete okno průzkumníka s touto složkou. Pak v druhém okně průzkumníka otevřete složku s příslušným dávkovým souborem, uchopíte jeho ikonu PRAVÝM tlačítkem myši, přesunete ji do složky nabídky a pustíte. Otevře se místní nabídka (ta se otevře, pouze pokud přesouváte soubor pravým tlačítkem), ve které zadáte, že zde chcete vytvořit zástupce, a tím celý proces končí.

Abyste mohli složku substituovat jako nějaký disk, nesmí váš operační systém používat disk označený tímto písmenem. Písmeno může být použito nejvýše pro jiný substituovaný disk, protože tuto substituci můžete před nastavením nové substituce zrušit (pro tento případ je v dávkovém souboru první příkaz s parametrem `/d` – delete).

Substituované složky umožňují sjednotit prostředí několika počítačů. První verzi této knihy jsem díky svému neustálému putování připravoval na několika počítačích, přičemž každý z nich měl jinak uspořádané složky (vadilo mi to, ale nemohl jsem s tím nic dělat). V každém z nich jsem ale měl definovány následující substituované disky:

- `J:` pro složku s vývojovými nástroji Javy,
- `S:` pro složku, ve které je text knihy,
- `V:` pro složku, ve které jsou programy pro knihu.

Po této úpravě mi již nutnost přecházení mezi různými počítači nevadila, protože jsem věděl, že na každém z nich najdu na discích `J:`, `S:` a `V:` potřebné nástroje a dokumentaci.

Používáte-li operační systém *Windows*, můžete urychlit budoucí vyhledání složky s projekty právě tím, že pro ni zřídíte zvláštní substituovaný disk. Kdykoliv se pak obrátíte na příslušný disk, obrátíte se ve skutečnosti k příslušné složce. Pomocí substituce si tak můžete zkrátit cestu k často používaným složkám.

Jednou z možností je zřídit na datovém disku složku Java s podsložkami Texty a Projekty. Do složky Texty si můžete vložit své poznámky, články stažené z internetu a další texty týkající se programování a Javy, ve složce Projekty si pak zřiďte pro každý projekt novou složku, ve které budou umístěny všechny soubory, které k němu budou patřit.

Jakmile složku pro své projekty zřídíte, můžete do ní hned stáhnout generátor projektů k této učebnici zmiňovaný v pasáži *Doprovodné programy* na straně 28, a s jeho pomocí pak získávat potřebné projekty.

Vyhledání a otevření projektu

V následujícím textu budeme pracovat s projektem **101a_Tvary**, který je určen pro první seznámení s prostředím BlueJ, třídami a objekty a s nímž budeme pracovat celou příští kapitolu.

Předpokládám, že máte instalováno vše, o čem jsem se zmiňoval v úvodu v pasáži *Potřebné vybavení* na straně 26. Spusťte *BlueJ*, zadejte příkaz **Projekt → Otevřít**. Po zadání příkazu se otevře dialogové okno **Otevřít projekt** (viz obrázek 1.1), v němž vyhledáte a označíte složku s projektem **101a_Tvary** a stisknete tlačítko **Otevřít**. Po otevření projektu by okno *BlueJ* mělo vypadat obdobně jako na obrázku 1.2.

Vraťme se ještě na chvilku k obrázku 1.1. Všimněte si, že ikony projektů (přesněji ikony složek, v nichž je uložen projekt) vypadají jinak než ikony obyčejných složek. *BlueJ* totiž do složky, ve které budou soubory jeho projektu, přidá svůj vlastní soubor `package.bluej`, do nějž si ukládá informace o poloze a velikosti aplikačního okna a o grafickém uspořádání objektů v zobrazovaném diagramu. Podle přítomnosti tohoto souboru pak pozná, zda se jedná o složku s jeho projektem nebo o nějakou obyčejnou složku.

1.7 Diagram tříd

Velký obdélník zabírající většinu okna projektu obsahuje tzv. **diagram tříd** našeho projektu. Diagram tříd popisuje strukturu našeho programu a vzájemné vazby mezi jeho částmi.

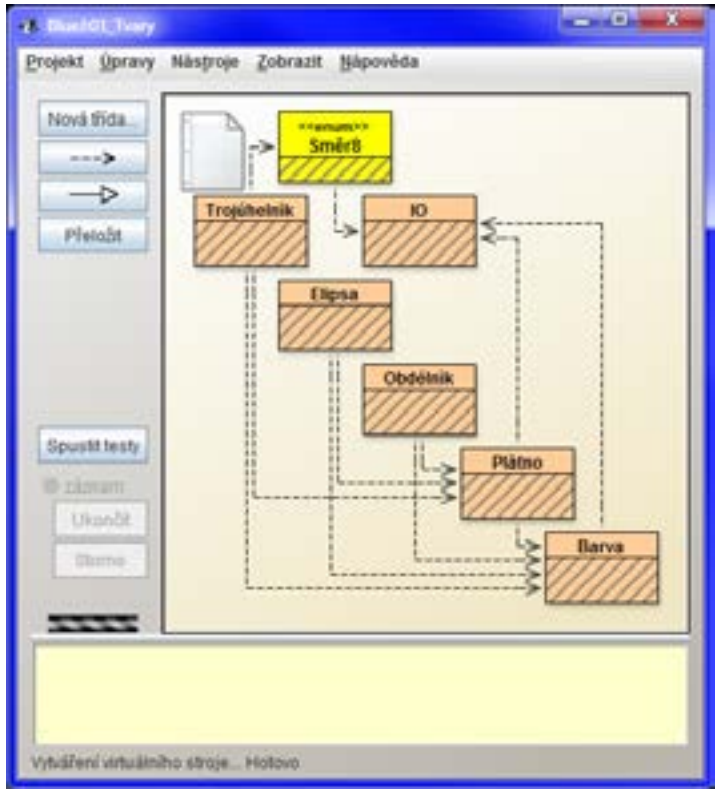
Malé obdélníky v diagramu tříd představují části programu, které nazýváme **třídy** (za chvíli si o nich budeme povídat podrobněji). Jsou znázorněny podle konvencí grafického jazyka UML[13] (vybarvení obdélníků do konvence nepatří, ale zvyšuje přehlednost a názornost diagramu). Čárkované šipky prozrazují, kdo koho používá – např. šipky vedoucí od tříd `Obdélník`, `Trojúhelník` a `Elipsa` ke třídě `Plátno` naznačují, že tyto třídy třídu `Plátno` používají (jak se vzápětí dozvíte, tyto obrazce se na plátno kreslí).

Bílý obrázek listu papíru v levém horním rohu diagramu představuje textový soubor se základním popisem celého projektu. Když na něj poklepete nebo když v jeho místní nabídce zadáte povel **Otevři**, otevře se okno editoru, v němž si budete moci tyto informace přečíst a v případě potřeby i upravit či doplnit.

---

[13] UML je zkratkou z anglického *Unified Modeling Language* – sjednocený modelovací jazyk. Je to grafický jazyk, ve kterém programátoři navrhují své aplikace, než začnou psát program.

Obrázek 1.1
Otevření existujícího projektu



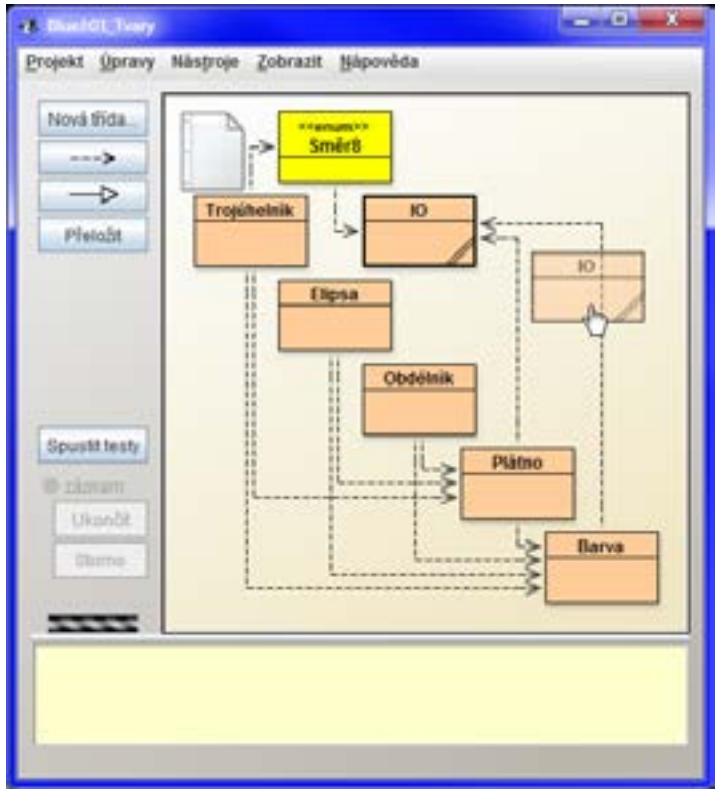
Obrázek 1.2
Okno BlueJ po otevření projektu 101a_Tvary

Šrafování spodní části obdélníků není součástí jazyka UML. Tímto způsobem se autoři *BlueJ* rozhodli symbolizovat to, že vyšrafované třídy ještě nejsou přeloženy. To lze snadno napravit: stiskněte tlačítko **Přeložit** v levé části aplikačního okna. Uvidíte, jak třídy ztmavnou (tím prostředí naznačuje, že se překládá), aby pak opět zesvětlely a jejich šrafování zmizelo (již jsou přeloženy). Jakmile je třída přeložena, můžete ji používat.

Manipulace s třídami v diagramu

Polohu jednotlivých tříd v diagramu můžete měnit. Najedete-li ukazatelem myši na obdélník představující danou třídu, změní se podoba kurzoru ze šipky na ukazující ruku. V tomto okamžiku můžete obdélník uchopit a přesunout do požadované pozice, kde jej upustíte.

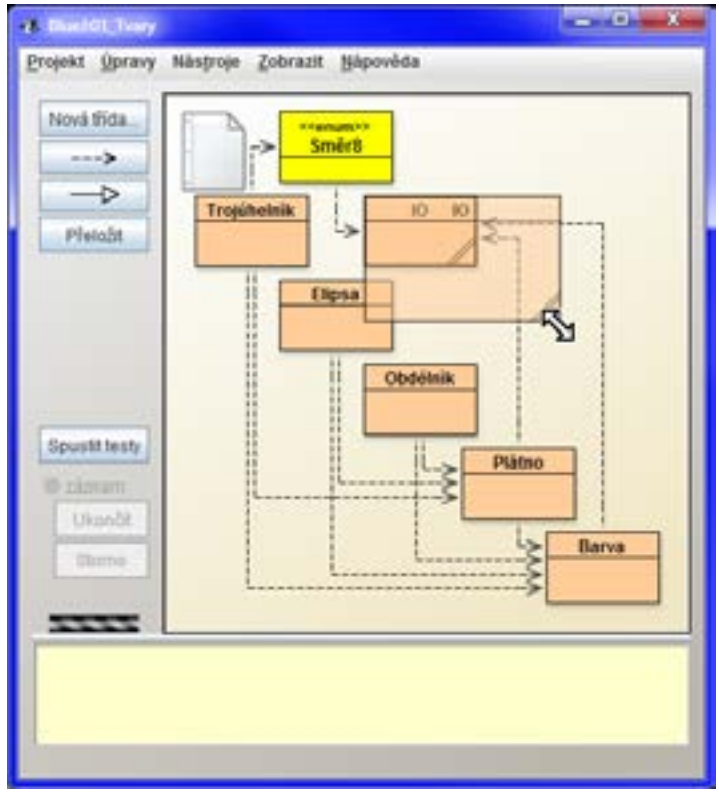
Po stisku tlačítka myši rám obdélníku ztuční. V průběhu přesunu se s ukazatelem myši bude přesouvat průsvitná kopie příslušného objektu, takže budete průběžně znát původní i nově nastavovanou pozici obdélníku (viz obrázek 1.3).



Obrázek 1.3
Posunutí ikony třídy v diagramu

Jsou-li zástupci tříd navzájem spojeni šipkami, bude *BlueJ* vaše přesuny sledovat a po umístění obrázku třídy do nové pozice šipky vždy příslušně překreslí. Můžete si tak diagram upravit do podoby, v níž vám bude připadat nejpřehlednější a nejnázornější.

Obdélníky zastupující třídy můžete nejenom přesouvat, ale i měnit jejich rozměr. Všimněte si, že poté, co na obdélník klepnete, objeví se u pravého dolního rohu ztučnělého rámu dvojité přeškrtnutí. Najedete-li ukazatelem myši do oblasti označené tímto přeškrtnutím, změní se jeho podoba na dvojitou šikmou šipku. Nyní můžete uchopit roh obdélníku, přesunout jej do nové pozice a upravit tak velikost obdélníku (viz obrázek 1.4).



Obrázek 1.4
Změna velikosti ikony třídy v diagramu

Někdy se stane, že se nám projekt rozrůstá a rozrůstá, a najednou bychom potřebovali posunout více tříd najednou, abychom udělali místo pro třídy, které se chystáme do projektu přidat. Postup je jednoduchý, ale nejprve se musíte naučit vybírat skupiny tříd.

Třídy jde zařazovat a vyřazovat ze skupiny vybraných tříd několika způsoby, které můžete kombinovat: