Nástroje pro tvorbu programů

Ing. Marek Běhálek
Katedra informatiky FEI VŠB-TUO
A-1018 / 597 324 251
http://www.cs.vsb.cz/behalek
marek.behalek@vsb.cz

Obsah kapitoly

- V této kapitole se budeme zabývat:
  - editory zdrojových kódu;
  - nástroji pro strávu aplikací;
  - nástroji pro sestavování aplikace;
  - testováním aplikací;
  - laděním aplikací;
  - …
- Nebudeme se zabývat jak vyvíjet software.
  - Předměty pokrývající oblast softwarové inženýrství.

Životní cyklus softwarového produktu

- Rozhodnutí o vytvoření produktu
- Specifikace požadavků
- Analýza a návrh aplikace
- Implementace
- Testování a ladění
- Dokumentace
- Instalace
- Marketing, prodej a podpora
- Údržba
- Ukončení prodeje a podpory

Specifikace požadavků a analýza

- Analytické nástroje
  - Obvykle grafické – diagramy UML
  - Požadavky
    - Funkční
    - Nefunkční
  - Datová analýza – statický pohled
  - Funkční analýza – dynamický pohled
- Úloha a náplň předmětu Úvod do softwarového inženýrství

Návrh aplikace

- Architektura aplikace
- Znovupoužitelnost
  - Knihovny
  - Třídy a objekty
  - Rozhraní
- Návrhové vzory
  - Osvědčená řešení často se vyskytujících situací
  - Kompozit, Pozorovatel, Továrna, …
- Úloha a náplň předmětu Úvod do softwarového inženýrství

Implementace

- Zápis ve formě programu
  - Nemusí jít o text – grafický jazyk
- Prototypová implementace
  - Pro rychlé ověření vlastností aplikace
  - Jazyky pro rychlé prototypování
- Verzování
  - Alfa-, Beta-verze, stabilní verze
  - Vývojová verze, „nightbuild“
  - Nástroje pro správu aplikací

Testování a ladění

- Jednotkové testy (unit tests)
  - Testování souladu s požadavky na úrovni jednotlivých programových jednotek – metod, tříd, modulů
- Integrační testy
  - Testování větších celků
- Ladění
  - Na úrovni zdrojového textu
  - Na strojové úrovni

Dokumentace

- Důležitá!
  - Týmová spolupráce, nahraditelnost
  - Slouží i pro autora
- Programátorská dokumentace
  - Často bývá součástí zdrojového kódu
- Uživatelská dokumentace
  - Návod k instalaci, konfiguraci, použití

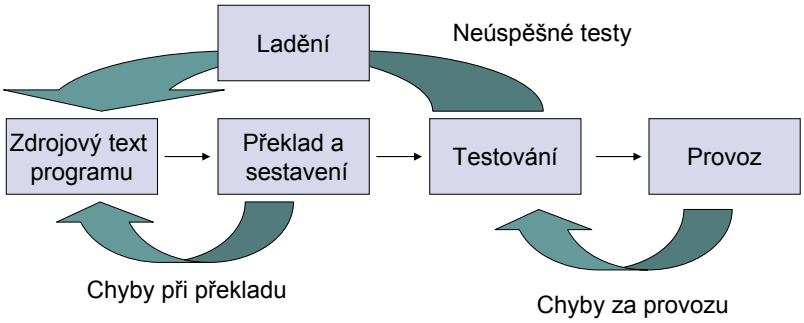
Instalace

- Může jít o náročný proces
  - Konfigurace okolního prostředí
  - Nastavení parametrů aplikace
  - Propojení s jinými aplikacemi
- Zvyšuje náklady na vlastnictví aplikace
  - Distribuce nových verzí – často přes Internet
  - Vzájemné ovlivňování různých aplikací

Údržba

- Oprava chyb
- Reakce na změny požadavků
  - Neúplné nebo nesprávně definované požadavky
  - Legislativní požadavky
- Rozšiřování aplikací
- Kdy již nemá smysl aplikace dále udržovat?
  - Cena údržby / cena nové aplikace

Tvorba programu

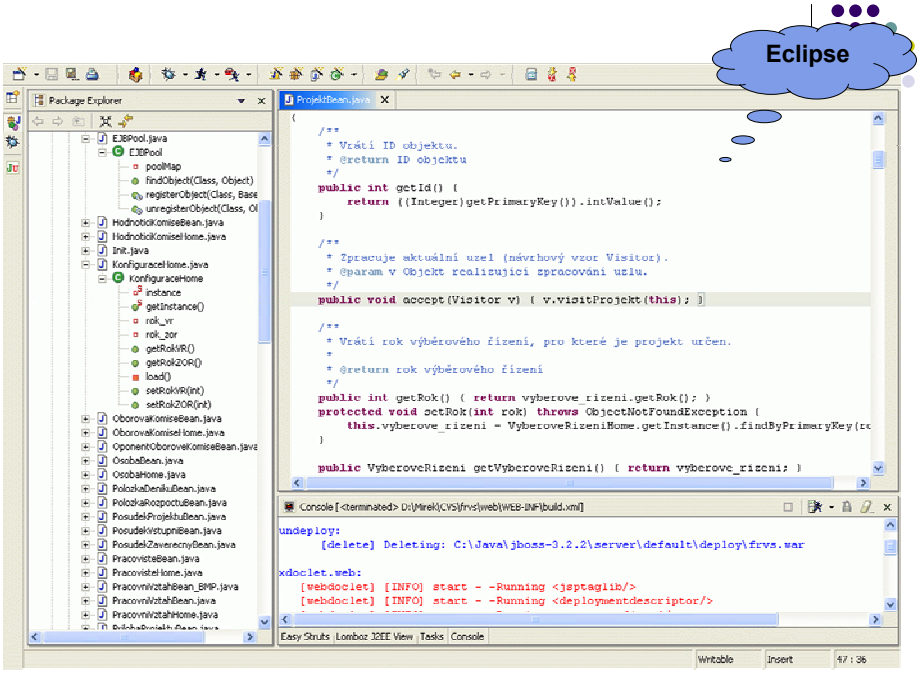


Nástroje pro programátory

- Editor
- Překladač / interpret x zpětný překladač
- Spojovací program
- Správa verzí
- Nástroje pro sestavení projektu
- Testovací nástroje, generátory testů
- Ladicí programy
- Nástroje pro ladění výkonu
- Nástroje pro tvorbu dokumentace
- Tvorba instalačních balíků
- Další nástroje:
  - Internacionalizace (i18n)
  - ...

Integrovaná vývojová prostředí (IDE)

- Poskytují více uvedených funkcí současně
  - Orientované na určitý jazyk
    - Borland Pascal, C++, JBuilder, C#Builder
    - SharpDeveloper, JCreator, NetBeans
  - Univerzální prostředí
    - Eclipse (Java, C++, C#, …)
    - MS Visual Studio (C++, C#, Jscript, VB, …)
- Rozšiřitelná prostředí
  - Připojování dalších funkcí k definovaným rozhraním (Eclipse, NetBeans)

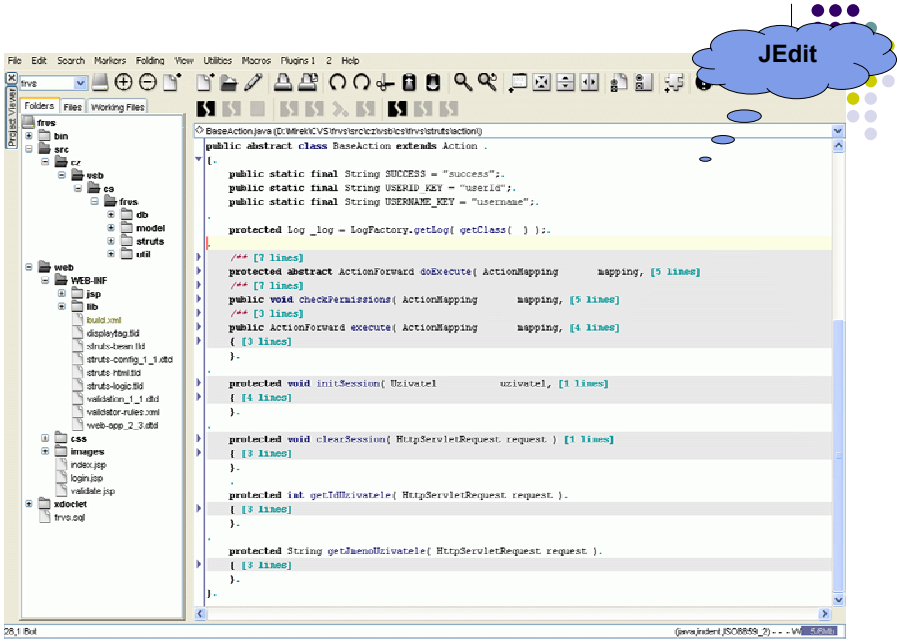


Preconstructed Development Environments

- Jiné používané jméno – Collaborative Development Environments
- Předpřipravená ucelená sada nástrojů pro vývoj aplikace.
  - správa projektu
  - sestavení aplikace
  - dokumentace
  - …
- Nejznámější PDE je SourceForge (http://www.sourceforge.net).
  - Open source projekty
  - V roce 2005 okolo 100 000 projektů a 1 milión uživatelů.
- Další PDE: GFroge, CollabNet, Savane,BerliOS,

Editor

- Programátorská podpora
  - zvýraznění syntaxe (syntax highlighting)
  - kontrola závorek
  - vyhledávání a nahrazování – soubor, projekt
  - šablony a makra
  - sbalení textu (folding)
  - spolupráce se správou verzí
- Příklad: PSPad, gvim, emacs, jEdit, …

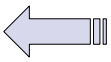


Překladač - Úloha překladače

- **Překlad jednoho jazyka na druhý**
  - Co je to jazyk?
    - Přirozený jazyk – složitá, nejednoznačná pravidla
    - Formální jazyk - popsán *gramatikou*
  - Co je to překlad?
    - Zobrazení T : L1 → L2
    - L1: zdrojový jazyk (např. C++)
    - L2: cílový jazyk (např. strojový kód P4)

Překladač - Zdrojový jazyk

- **Přirozený jazyk**
  - Předmět zájmu (počítačové) lingvistiky
- **Programovací jazyk**
  - C, C++, Java, C#, Prolog, Haskell



- **Speciální jazyk**
  - Jazyky pro popis VLSI prvků (VHDL)
  - Jazyky pro popis dokumentů (LaTeX, HTML, XML, RTF)
  - Jazyky pro popis grafických objektů (PostScript)

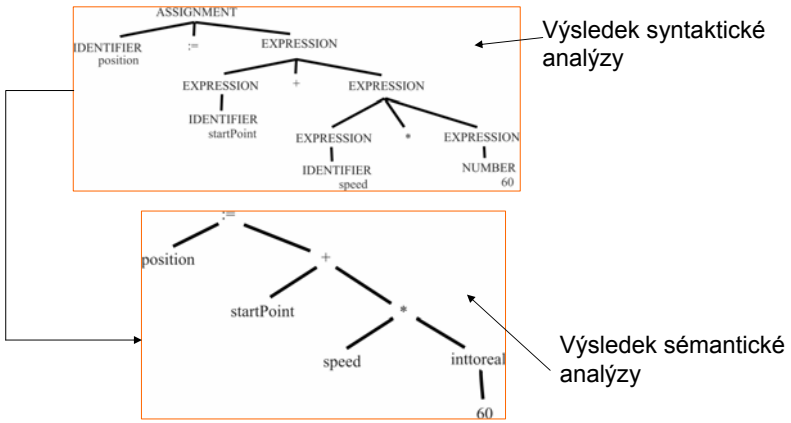
Překladač - Cílový jazyk

- **Strojový jazyk**
  - Absolutní binární kód
  - Přemistitelný binární kód (.obj, .o)
  - Jazyk symbolických instrukcí
- **Vyšší programovací jazyk (např. C)**
- **Jazyk virtuálního procesoru**
  - Java Virtual Machine
  - MSIL pro .NET

Překladač - Modely zdrojového programu (1)

- Vstup: Zdrojový program
  - `position := startPoint + speed * 60;`
- Lexikální analýza
  - `<ID,position> <:=,> <ID,startPoint> <+,> <ID,speed> <*,> <INT,60>`
- Syntaktická analýza
- Sémantická analýza

Překladač - Modely zdrojového programu (2)



Výsledek syntaktické analýzy

Výsledek sémantické analýzy

Překladač - Modely zdrojového programu (3)

- Generování mezikódu
  - `temp1 := inttoreal(60)`
  - `temp2 := speed * temp1`
  - `temp3 := startPoint + temp2`
  - `position := temp3`
- Optimalizace
  - `temp1 := speed * 60.0`
  - `position := startPoint + temp1`
- Generování cílového programu

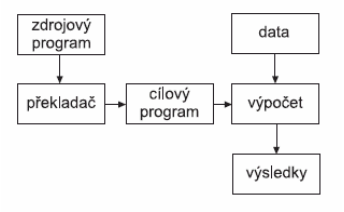
```
ld      qword ptr [_speed]
fmul    dword ptr [00B2]          ;60.0
fadd    qword ptr [_startPoint]
fstp    qword ptr [_position]
```


Překladač – Funkce překladače

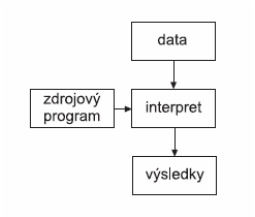
- *Analýza* zdrojového textu, vyhledání chyb
  - Základní stavební prvky – identifikátory, čísla, řetězce, operátory, oddělovače, …
  - Programové konstrukce – deklarace, příkazy, výrazy
  - Kontextové vazby – definice/užití, datové typy
- *Syntéza* cílového programu / interpretace
  - Strojový jazyk (nebo JSI)
  - Jazyk virtuálního procesoru (JVM, CLR)

Překladač - Typy překladače (1)

- Kompilační překladač



- Interpretační překladač



Překladač – Typy překladače (2)

- Inkrementální překlad
  - Umožňuje po drobné opravě přeložit jen změněnou část
  - Možnost provádění drobných změn během ladění programu
- Just-in-time překlad
  - Generování instrukcí virtuálního procesoru (Java VM - .class, .NET CLR – jazyk IL)
  - Překlad až v okamžiku volání podprogramu
  - Optimalizace podle konkrétního procesoru

Překladač – Typy překladače (3)

- Zpětný překladač
  - Umožňuje získat zdrojový program z cílového (např. z .exe, .class)
    - disassembler (např. ILDASM v prostředí .NET)
    - decompiler (např. DJ Java Decompiler)
  - V některých státech (USA) není dovoleno (u nás **ano** – viz § 66 autorského zákona)
  - *Obfuscation* („zmatení“, také „duševní pomatenost“ – viz http://slovniky.seznam.cz/)
  - Transformace cílového programu komplikující zpětný překlad

Sestavovací program (linker)

Vstup:

- Přemístitelné moduly (.obj, .o)
  - Relativní adresy
  - Tabulka adres určených pro *relokaci*
- Knihovny modulů (.lib, .a)

Výstup:

- Spustitelný program (.exe)
- Dynamicky zaváděné knihovny (.dll, .so)

Správa verzí - Problémy při vývoji aplikace

- Oprava chyb
  1. Byl vydán produkt (verze 1.0)
  2. Produkt je dále rozšiřován, aktuální verze je 1.3
  3. Je objevena zásadní chyba ve verzi 1.0
  4. Verze 1.3 není připravena k vydání.
- Paralelní vývoj
  - Na projektu pracuje najednou několik programátorů.
- Ladění chyb
  - Při ladění objevíme chybu a chceme zjistit, kdy se daná chyba do produktu dostala.

Správa verzí (1)

- **Systém pro správu verzí (SCM – Software Configuration Management) uchovává změny projektu jak probíhaly v čase při jeho vývoji.**
- **Vývoj v týmu**
  - sdílení zdrojových textů
  - ochrana před současnými modifikacemi
- **Vývoj různých verzí**
  - Verze pro různá cílová prostředí
  - Vývojové verze – 0.1, 1.1beta, 1.2RC3, ...
  - Možnost mít různé větve při vývoji
- **Archivace**
  - Možnost návratu po nevhodných změnách
  - Zjištění rozdílů ve verzích
- **Bezpečnost**

Správa verzí (2)

- **Rozdělení podle uložení dat**
  - centralizované
  - distribuované
    - někdy obtížné striktně rozdělit, mohou například podporovat replikace pro urychlení práce
- **Rozdělení podle způsobu přístupu**
  - sériový model – právě jeden uživatel může měnit soubory
  - konkurenční model – více uživatelům je povolen přístup k souborům

Správa verzí (3)

- **RCS** (Walter F. Tichy, Purdue University) - jen pro jednotlivé soubory, jeden uživatel
- **CVS** (Concurrent Version System)
  - architektura klient-server
  - pojmy zavedené CVS jsou běžné i v jiných nástrojích
  - pořád nejčastěji používaný systém
- **Subversion,MS Visual SourceSafe, IBM Rational ClearCase, Perforce, BitKeeper, Arch, …**

Správa verzí – Cena použití SCM

- **Velikost uložených dat**
  - Projekt zabírá mnohonásobně více místa než je nutné.
  - Naivní přístup – uložení každé nové verze.
  - Typicky ukládány pouze změny.
- **Výkon**
  - Získání aktuální verze projektu z SCM je mnohem náročnější než posté „zkopírování“ z adresáře.
- **Konektivita**
  - Nutnost připojení k serveru, kde jsou uložená data.
- **Znalost aplikace**
  - Lidé používající SCM musí mít nějaké základní znalosti jak se systémem pracovat.
  - Nemusí být jednoduché.
- **Cena za provoz SCM**
- **Riziko poškození**
  - Vše umístěno na jednom místě.

Správa verzí - Concurrent Version Systém (1)

- Jeden ze systémů pro správu verzí.
- Postaven na architektuře klient-server.
  - Možnost práce více uživatelů najednou.
- Neklade nároky na jednotné vývojové prostředí.
- Projekt je sada souborů uložených v systému pro správu verzí.
  - U CVS je nazýván *modul*.

Správa verzí - Concurrent Version Systém (2)

- Místo, kam se ukládají data je v CVS nazýváno ***repository***.
  - originální soubory
  - změnové soubory + komentáře (diff)
    - Zabere menší prostor k uložení.
    - Získání souboru a nebo jeho úprava mohou být časově náročné (může být nutné projít všechny verze).
  - může být na sdíleném serveru nebo v síti
- Nestará se například o vytváření (build) projektu, jen ukládá a spravuje verze kolekce souborů.

Správa verzí – Práce s CVS (1)

- Předpokládejme, že v CVS je uložena kompletní verze vyvíjeného projektu projekt.

1. ***Checkout***
   - Vývojář se rozhodne pracovat na nějaké části projektu.
   - Operací: `cvs checkout` získá svou osobní pracovní verzi (uložena lokálně u vývojáře).
   - Obvykle je automaticky inkrementováno číslo verze: z 1.5 na 1.6.
   - Doplněny další informace, jako kdo, kdy,… soubor získal.

Správa verzí – Práce s CVS (2)

2. ***Edit***
   - Vývojář pracuje na své lokální verzi (v CVS označována jako *sandbox*).
   - Může přidávat soubory, měnit jejich obsah.
   - Sestavuje a spouští tuto svou lokální verzi.
3. ***Diff***
   - Zjištění změn v pracovní verzi oproti verzi, která je uložená v *repository*.
   - Výsledek je jaké změna a kde se udály.

Správa verzí – Práce s CVS (3)

4. ***Update***
   - Obsah *repository* se mohl změnit v průběhu práce programátora.
   - Vývojář získá aktuální verzi z *repository* a snaží se jí sladit s jeho pracovní verzí.
     - Možnost, že soubor paralelně modifikovali dva vývojáři.
     - Je potřeba vyřešit potencionální konflikty.
5. ***Commit***
   - Programátor ukládá změny provede ve své pracovní verzi zpět do *repository* (operace: `cvs commit`).

Správa verzí - Slučování změn (1)

1. V repozitory je uložena verze 1.5.
2. Programátoři Alice a Bob získají soubory z této verze (oprace *checkout*).
3. Oba dva provedou změny.
4. Alice uloží změny na CVS (operace *commit*).
5. Bob chce uložit změny.
   - V systému je nyní verze 1.6.
   - Bobova verze je 1.5
   - Bob neměnil aktuální verzi!
6. Bob musí provést *udpade své verze.*
   - Nemusí uspět!

Správa verzí - Slučování změn (2)

- Operace update uspěje když:
  - apply(A, apply(B,1.5)) = apply(B, apply(A,1.5))
  - Nezáleží na pořadí.
  - V případě neúspěchu CVS ohlásí chybu, verze nelze sloučit.
- V případě, že CVS neohlásí chybu, ještě nemuselo být sloučení v pořádku! – **testování**
- V případě, že CVS ohlásí chybu, jsou Bobovi označena místa konfliktů ve zdrojových souborech.
- Bob tyto konflikty musí vyřešit, pokud chce umístit svou verzi do CVS.

Správa verzí - Slučování změn (3)

- Chyba, která je CVS rozpoznána
  - 1.5: `a = b;`
  - A(1.6): `a = ++b;`
  - B(1.7): `a = b++;`
- Chyba, kterou CVS nerozpozná
  - 1.5: `int f(int a) {…}`
  - A(1.6): `int f(int a, int b) {…}`
  - B: `f(5);`
- Kontrola prováděna na základě porovnávání textu (*diff3*)

Správa verzí – Tags and Branches

- ***Tag***
  - V jisté fázi vývoje projektu pojmenujeme aktuální verzi.
- ***Branches*** – dvě revize souboru, větvení
  - Dva lidé požádají o verzi 1.5
  - Vytvoří se dvě verze 1.5.1 a 1.5.2
  - Tyto dvě verze jsou dále uchovávány najednou.
- Normálně nejsou vytvářeny dvě větve projektu, ale změny se „slučují (*merging*)“.
- Je udržována jedna hlavní vývojová linie.
- Musíte explicitně vytvořit další větev.

Správa verzí – Další vlastnosti CVS (1)

- CVS logs
  - Pro každý soubor jsou dostupné informace o změnách.
- CVS je volně dostupné, existuje celá řada klientů.
- Nevýhody
  - Operace checkout a commit jsou atomické jen na úrovni adresářů né transakcí.
    - Pokud operace commit není korektně dokončena, jsou aktualizovány jen některé soubory.
    - Pokud někdo čte a zároveň někdo jiný zapisuje, může získat verzi, kde jsou jen některé změny.
  - Další problémy spojené s přejmenováním adresářů, nemožností připojit poznámky k pojmenovaným verzím, pojmenováním souborů a adresářů…
    - Odráží skutečnost, že CVS se vyvíjelo a ne vytvářelo jako celek.

Správa verzí - Subversion

- Subversion je vydáván v licenci Apace Software Fundation.
- Nástupce CVS
  - podobný „styl“
  - umožňuje konkurenční a centralizovanou správu verzí
- Hlavní změny oproti CVS
  - Číslování verzí souboru
  - Přejmenování adresářů a souborů
  - Atomické operace
    - Operace buď uspějí celé a nebo se neprojeví vůbec.
  - Metadata jsou také „verzována“.
  - Plná podpora binárních souboru.

Správa verzí - Arch

- Oproti centralizovaným SCM jako je CVS nebo Subversion umožňuje Arch distribuovaný přístup ke správě verzí.
  - Umožňuje podobný systém distribuce dat jako BitTorrent.
  - Můžete pracovat se svou „lokální“ repository.
  - V případě že to chcete, je pak tato repository synchronizována s ostatníma.
- Jde o open source a volně dostupný nástroj.
- Nástroj je aktivně vyvíjen.
- Hlavním nedostatkem je, že pro velký objem dat je poměrně pomalý.

Správa projektů – Proč používat nástroje pro správu (1)

- Sestavení projektu – kompilace zdrojových kódů do formy, kterou lze provádět na počítači.
- Sestavení projektu může být poměrně složité, pokud velikost projektu roste.
  - Rozdělení zdrojových kódů na části (GUI, interface databáze,…)
  - Jednotlivé části jsou závislé na jiných (ne na všech).
  - Může být složité definovat závislé části (reflexe v Javě).
  - Může být definováno nějaké netriviální pořadí pro kompilaci.
  - Sestavení celé aplikace může být časově náročné.
    - Při změně nějakého zdrojového souboru chceme zkompilovat pouze nezbytně nutné zdrojové soubory.

Správa projektů – Proč používat nástroje pro správu (2)

- Správa projektu nezahrnuje jen vlastní sestavení aplikace. Typické činnosti:
  - inicializace prostředí (adresáře, parametry, ...)
  - překlad a sestavení programů
  - systematické testování
  - generování dokumentace
  - odstranění pracovních souborů
  - vytváření archivů a distribucí
  - instalace

Správa projektů – Proč používat nástroje pro správu (2)

- Některé studie uvádějí 10 % – 30 % času při vývoji komplexních aplikací zabere:
  - práce na skriptech, které sestavují aplikaci;
  - čekání na pomalé sestavování aplikace;
  - hledání chyb, které způsobuje nekonzistentní sestavování aplikace.

Správa projektů – Hlavní cíle

- **Nezávislost na prostředí**
  - umístění knihoven, programů
  - verze a varianty nástrojů (např. javac/jikes)
- **Udržení konzistence**
  - sledování závislostí
- **Optimalizace budování projektu**
  - vyhledání nejkratší cesty – zpracování pouze změněných a na nich závisejících souborů

Správa projektů – Typické schéma použití (1)

- Definování cílů
  - Obvykle předán jako parametr při spuštění.
- Načtení skriptu pro sestavení aplikace – „build file“
  - Načtení souboru a jeho kontrola.
- Konfigurace
  - Jeden skript může být použitý na více platformách.
  - Specifické nastavení může být definováno přímo při použití programátorem (další parametry příkazové řádky).

Správa projektů – Typické schéma použití (2)

- Zohlednění závislostí
  - Zohlední možné chyby jako jsou například cyklické závislosti
- Definice cílů pro sestavení
  - Sestaví posloupnost kroků, kterou je nutné provést k úspěšnému sestavení aplikace.
- Vytvoření příkazů, které sestaví aplikaci
  - Zohlední další informace poskytované programátorem, vlastnosti cílové platformy,…
- Provedení vytvořených příkazů
  - Mohou nastat různé chyby.

Správa projektů – Typické problémy (1)

- Maximální délka textu použité v příkazové řádce
  - absolutní cesty
- Formát jmen souborů
  - Rozdíly nejen mezi platformami ale například i mezi verzemi jedné platformy.
- Jednotkové testy pro nástroje pro správu aplikací
  - „Programování“ skriptu pro nástroje pro správu projektu je také vytváření aplikace.
  - Obtížné hledání chyb.

Správa projektů – Typické problémy (2)

- Pomalé sestavení aplikace
  - Profilace jednotlivých sestavení
  - Provádět činnosti pouze jednou
    - Špatně definované závislosti
  - Použití serveru pro sestavování aplikace
  - Rozdělení sestavení aplikace na stupně
    - Různě stupně sestavení pak mohou být použity jako startovní.
  - Použití cache
  - Použití paralelního či distribuovaného zpracování
    - Podporují jen některé nástroje.
    - V zásadě mnohem obtížnější sestavit či udržovat.

Správa projektů - Dávkové zpracování (1)

preloz.sh

```
yacc –o synt.cpp –d synt.y
lex  –o lex.cpp lex.l
gcc  –o prekl synt.cpp lex.cpp main.cpp
```

- Nejjednodušší možnost pro správu projektu.
- Výhody
  - Je jednoduché a rychlé je sestavit.
  - Jednoduché zjistit, jaké příkazy se mají provést.

Správa projektů - Dávkové zpracování (2)

- Nevýhody
  - Provede všechny příkazy, ne jen nutné.
  - Detekce chyb
    - Provádění skriptu pokračuje i po chybě.
    - Chyba může znehodnotit další sestavování aplikace.
  - Ladění
    - Ladění je u dávkových souboru realizováno hlavně textovými výpisy.
    - Někdy je možné provést „dry run“ – příkazy jsou pouze vypsány, né provedeny.
  - Přenositelnost
    - Obvykle je obtížné (nemožné) přenášet dávkové soubory mezi platformami.

Správa projektů - Program `make` (1)

- První nástroj pro sestavování aplikací
  - Pořád jeden z nejpoužívanějších programátory v C/C++.
- Skript pro sestavení se obvykle jmenuje „`makefile`“.
- Celá řada implementací – make (1977), gmake, nmake,…
- Různé produkty postavené na konceptu `make` (například cake, cook - použitý s CVS Aegis).

Správa projektů - Program `make` (2)

- Statická definice závislostí
- Sestavení *cílových objektů* na základě *předpokladů*
  - prog.o: prog.cpp lex.cpp synt.cpp
    gcc –c prog.cpp lex.cpp synt.cpp
  - na začátku je tabelátor!
- Využití *implicitních pravidel*
  - prog.cpp -> prog.o -> prog
- Makrodefinice
  - SRCS = prog.cpp lex.cpp synt.cpp
    prog: $(SRCS)
    gcc –o prog $(SRCS)

Správa projektů - Příklad souboru `Makefile`

```
all: p4

par.o: par.c lex.c

p4: par.o
  $(CC) -o p4 par.o

clean:
  $(RM) par.c par.o lex.c

allclean: clean
  $(RM) p4

dist:
  tar -czf p4.tgz Makefile lex.l par.y
```

Správa projektů –Výhody a nevýhody `make` (1)

- Výhody
  - Make je rozšířený a široce používaný.
- Nevýhody
  - Nekompletní analýza závislostí, cyklické závislosti
    - Závislosti jsou definovány staticky.
    - Rekurzivní volání mezi `makefile` sobory.
  - Přenositelnost
    - Různé chování různých variant `make` na různých platformách.
    - Použití jednoho nástroje (například kompilátoru) se může na jiné platformě lišit.

Správa projektů –Výhody a nevýhody `make` (2)

- Nevýhody
  - Rychlost
  - Ladění
    - Můžeme použít parametr –n pro „dry run“.
    - Pořád může být obtížné určit, proč některé soubory byly či nebyly použity.
  - Nutnost rekompilace je detekovány na základě časových razítek souborů.
  - Syntaxe `makefile` souborů
- Řešením může být „další vrstva“ – generátor `makefile` souborů (nejznámější Automake)

Správa projektů - GNU Autotools

- Nejčastěji používané pro „open source“ C/C++ projekty.
- Skládá se z:
  - **Autoconf**
    - Vytváří skripty pojmenované `configure`.
    - Tyto skripty zjistí, jak daný systém splňuje požadavky aplikace na něj kladené.
  - **Automake**
    - Ze skriptu `Makefile.am` vytváří `Makefile.in`. Ten je potom použit nástrojem Autoconf k sestavení zdrojového souboru pro **GNU gmake**.
  - **Libtool**
    - Vytváří knihovny pro programy v C.

Správa projektů – JAM

- Vytvořen Perforce (volně k dispozici)
- Just Another Make – JAM
- Určen pro sestavování aplikací v C/C++
- Oproti nástroji `make` je rychlejší.

Správa projektů – Ant (1)

- Ant – zkratka: Another neat tool
- Produkt vytvořený v licenci Apache Fundation.
- Implementován v prostředí Java
  - Platformě nezávislý
- Nejčastěji používaný nástroj pro sestavování aplikací v Javě.
- Můžeme provést vše, co „umí“ Java
- Nyní implementováno více než 100 funkcí (http://ant.apache.org).
- Možnost rozšiřování
  - definované programátorské rozhraní
  - přidávání dalších akcí
- Integrován do mnoha vývojových prostředí
  - Eclipse, NetBeans, JBuilder, jEdit, ...

Správa projektů – Ant (2)

- Činnost se řídí souborem v XML
  - Musí dodržovat všechny běžné konvence pro XML dokument.
  - build.xml
  - ant xxx – zpracování cíle s názvem xxx
- Hlavní struktura skriptu pro ANT
  - Hlavní element je element `<project>`
  - V těle tohoto elementu jsou umístěny elementy `<target>`
    - Definují jednotlivé cíle
  - Cile jsou složeny z `<task>` elementů. Ty definují jednotlivé operace, které se mají provést.
  - Možnost použití proměnných - `<property>`

Správa projektů – První příklad skriptu pro ANT (1)

```
<?xml version="1.0"?>
<project name="Test" default="compile" basedir=".">
  <property name="dir.src" value="src"/>
  <property name="dir.build" value="build"/>

  <target name="prepare">
    <mkdir dir="${dir.build}">
  </target>

  <target name="clean" description="Remove all">
    <delete dir="${dir.build}">
  </target>

  <target name="compile" depends="prepare">
    <javac srcdir="${dir.src}" destdir="${dir.build}">
  </target>
</project>
```


Správa projektů – První příklad skriptu pro ANT (2)

- Obvykle v souboru *build.xml*, ale jméno může být libovolné.
- Spuštění:
  - ant clean
  - ant –buildfile MyBuildFile.xml clean
  - ant
  - ant clean compile

Správa projektů – Pomocné výpisy v ANTu

```
<target name="help">
    <echo message="Toto je nejaky text">
    <echo>
      Tento text bude vypsan taky!
    </echo>
    <echo><![CDATA[
    tento text
    bude na dva radky]]>
    </echo>
</target>

ant -projecthelp
ant -verbrose
```


Správa projektů – Systémové proměnné v ANTu

```
<!-- Abort if TOMCAT_HOME is not set -->
 <target name="checkTomcat"
 unless="env.TOMCAT_HOME">
     <fail message="TOMCAT_HOME must be set!">
 </target>
```

- `env` – odpovídá volání metody `System.getEnv()`
- Vlastnosti lze umístit do externího souboru
  - `dir.buid=build`
  - `<property file="local.properties">`

Správa projektů – Argumenty příkazové řádky v ANTu (2)

```
<?xml version="1.0"?>
<project name="properties" default="run" basedir=".">
  <property name="prop1" value="Property 1 value">
  <target name="run">
    <echo message="prop1 = ${prop1}">
    <echo message="prop2 = ${prop2}">

    <java classname="ShowProps">
      <classpath path="./">
      <sysproperty key="prop1" value="${prop1}">
    </java>
  </target>
</project>

ant -Dprop2="Hello world">
```


Správa projektů – Argumenty příkazové řádky ANTu (2)

```
public class ShowProps {
  public static void Main(String[] args) {
      System.out.println(“prop1 = ”
  +System.getProperty(“prop1”));
      System.out.println(“prop2 = “
  +System.getProperty(“prop2”));
  }
}

Výstup:
[echo]prop1 = Property 1 value
[echo]prop2 = Hello world
[java]prop1 = Property 1 value
[java]prop2 = null
```

Správa projektů – Definice proměnné *classpath* v ANTu

```
<path id="project.classpath">
  <pathelement location="${dir.src}"/>
  <fileset dir="${tomcat}/common/lib">
    <include name="*.jar"/>
  </fileset>
  <fileset location="${dir.src}">
    <include name="*.jar">
  </fileset>
</path>

<pathconvertor targetos="windows" property="windowsPath"
 refid="projec.classpath">

<javac destdir="${dir.build}">
  <src path="${dir.src}">
  <classpath refid="project.classpath">
</javac>
```


Správa projektů – Definice cest k souborům

- include = “scr/lib/cviceni1/*.java”
- include = “src/**/*.java”
- exclude = “src/**/Test*.java”
- include = “**/a/**”
- include = “Test?.java”

Správa projektů – Definice cest - *fileset*

```
<target name="clean"
  description="Clean project">
     <delete file="uloha3.jar"/>
     <delete>
       <fileset dir=".">
         <include name="**/*.class">
       </fileset>
     </delete>
  </target>
```


Správa projektů – Jar archivy a dokumentace v ANTu

```
<jar jarfile="${dir.dist}/cviceni3.jar">
   <fileset dir="${bir.build}"
     includes="**/*.class"
     exludes="**/Test*.class"/>
 </jar>

 <mkdir dir="apidoc"/>
 <javadoc packagenames="gramar.*"
 destdir="apidoc">
   <sourcepath>
     <pathelement location="."/>
   </sourcepath>
 </javadoc>
```

Správa projektů – Další podporované funkce v ANTu

- Umožňuje implementovat logiku (podmínky)
- Testování pomocí *JUnit*
- Spolupráci s CVS, FTP, Telnet
- Vytváření archivů
- Práce se soubory – změna práv, kopírování,…
- Validace XML dokumentů
- a mnoho dalších…

Správa projektů – Slabosti ANTu (1)

- Omezení pro XML dokumenty
  - Velké projekty budou mít rozsáhlé soubory pro sestavení.
  - Speciální znaky jako <. Nutno používat `<`.
- Složité řetězce závislostí
  - Lze rozdělit do více zdrojových souborů a volat je pomocí úkolu `antcall`.
    - Může značně zpomalit sestavení aplikace.
  - Od verze 1.6 lze používat úkol `import`, což ulehčuje modulární vytváření skriptů pro ANT.

Správa projektů – Slabosti ANTu (2)

- Omezené použití `<property>`
  - Nemají vlastnosti proměnných z programovacích jazyků.
  - Jakmile je jednou nastavena hodnota nemůže už být změněna.
  - Nelze použít `property`, která by obsahovala název další `property` a tak se dostat k její hodnotě.
  - XML editory často neumí pracovat s proměnnými ANTu
- Paralelní zpracování a „dry run“
- Pomalý start – použití JVM
- Platformě závislé problémy
  - Lze jim předcházet, například použitím úkolu `PathConvertor`.

Správa projektů – Další projekty navazující na ANT

- Další úlohy pro ANT
  - AntContrib – podpora kompilování zdrojových souborů v C/C++ na různých plaformách
- Generátory zdrojových skriptů pro ANT
  - Antelope – UI pro vytváření skriptů pro ANT, pomáhá také v profilaci a nebo ladění skriptů.
- Další varianty ANTu
  - nant – nástroj pro správu projektu na platformě .NET

Správa projektů – SCons

- Skripty sestavovány v jazyce Python
  - Lze využít všech možností jazyka Python.
- Hlavní vlastnosti
  - Přenositelné soubory pro sestavení
  - Automatická detekce závislostí
  - K detekci zda došlo ke změně používá MD5 signaturu.
  - Podpora paralelního sestavování
  - Rozšiřitelnost a modularita
  - Integrace nástrojů jako například nástroje pro správu verzí.

Ladění programů

- „Hledání chyb je proces ověřování mnoha věcí, v jejichž platnost věříme, až po nalezení toho, co není pravda.“
  - V určitém bodě programu má proměnná *x* hodnotu *v*.
  - V konkrétním příkazu *if-then-else* provedeme právě větev else.
  - Funkce *f* se volá se správnými parametry.
- Vše je třeba ověřit – jak?

Ladění programů - Strategie ladění programů (1)

- **„Binární vyhledávání“**
  - Omezujeme úsek programu, ve kterém se hledaná chyba může vyskytovat
  - Příklad: Hledáme místo, kde se nastavila nesprávná hodnota nějaké proměnné.
  - Lze použít i na nějaké problémy při překladu.

Ladění programů - Strategie ladění programů (2)

- **Ladící výpisy**
  - v principu nevhodné
  - pomocí: printf/count/System.out.write
  - odstranění z odladěné verze
    - komentáře
    - podmíněný překlad
- **Logovací nástroje (např. log4j)**
  - možnost konfigurace
  - požadavky na nízkou režii – ponechává se v hotovém programu

Ladění programů - Strategie ladění programů (3)

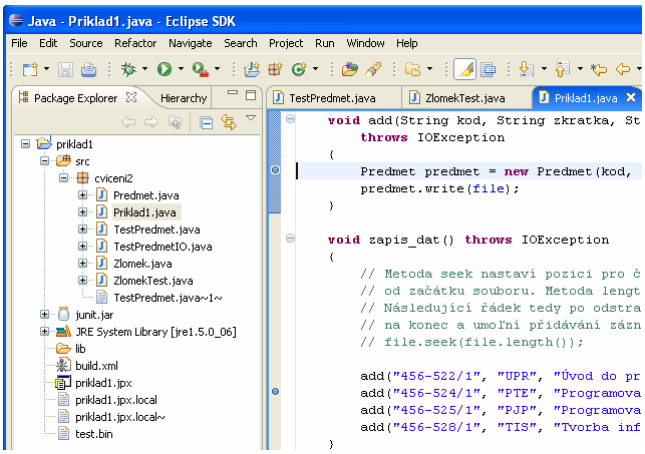
- **Sledování stopy programu (trace)**
  - Výpis posloupnosti zpracovaných řádků
  - Výpis volání podprogramů
- **Analýza obsahu paměti po chybě**
  - Uložení obrazu paměti do souboru (core v Unixu)
  - Statistika analýzy bodu, kde došlo k chybě
  - Propojování se zdrojovým programem

Ladění programů - Strategie ladění programů (4)

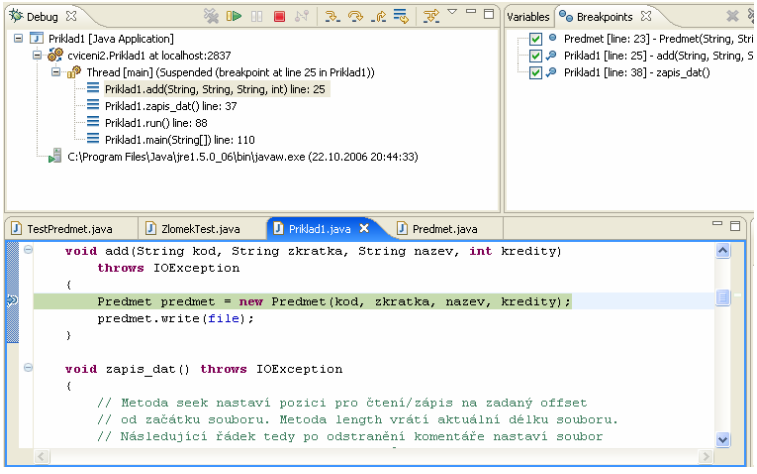
- **Využití ladícího programu**
  - Součást většiny integrovaných vývojových prostředí
  - Existuje celá řada různých nástrojů.
  - Na zdrojové nebo instrukční úrovni

1. Definice bodů zastavení (breakpoint)
2. Spuštění programu
3. Kontrola stavu v bodech zastavení
4. Krokování do bodu zastavení (step into, step over)

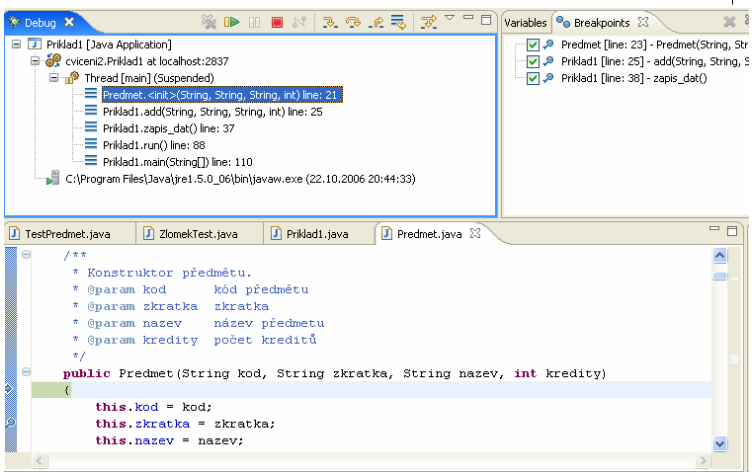
Ladění programů – Příklad nastaveni „breakpointů“



Ladění programů – Příklad běhu v ladícím režimu



Ladění programů – Příklad použití funkce step-in



Testování programů - Co je to chyba?

- Jakýkoliv problém, snižující kvalitu programu.
  - Funkcionalita
  - Užitná hodnota
  - Spolehlivost
  - Výkon
  - Požadavky uživatele

Testování programů (1)

- **Verifikace**
  - Ověřování interní konzistence produktu (zda produkt odpovídá návrhu, návrh analýze, analýza požadavkům)
  - Formální verifikace – často obtížná
- **Validace**
  - Ověřování (externím nezávislým zdrojem), zda celé řešení splňuje očekávání uživatelů nebo klientů.

Testování programů (2)

- Cíl: **Snížení rizika výskytu chyby**
- Nutný pesimismus!
  - Výskyt chyby je třeba očekávat.
- Nejdůležitější pravidlo pro testování je dělat ho. (B. Keringham a R. Pike – The Practice of Programming)
- Opakované testování (re-testing)
  - Kontrola, zda jsme chybu odstranili.
- Regresní testování
  - Kontrola, zda jsme úpravou nevnesli nove chyby

Testování programů – Typy testů (1)

- Rozdělení na testy, které:
  - jsou součástí výsledného produktu;
  - jsou odděleny od výsledného produktu (*náplní této kapitoly*).
- Rozdělení dle množství informací, které máme pro testování.
  - black box testing – osoba, která vytváří test nemusí mít informace o tom, jak funguje aplikace na úrovni, na které je test vytvářen.
  - white box testing – pro vytváření testů je nutné znát informace o fungování testované části

Testování programů – Typy testů (2)

- „Ruční“ testování
  - náročné
  - není opakovatelné
  - snadno se přehlédnou chyby
- Automatické testování
  - generování testů – zajištění maximálního pokrytí zdrojového textu

Testování programů – Úrovně testování (1)

1. **Jednotkové testy (Unit tests)**
   - Jsou vytvářeny pro malé části produktu – „jednotky“.
   - Co to je jednotka závisí na konkrétním produktu, programovacím jazyce,… (třída, metoda třídy, funkcionalita tvořené aplikace,…)
   - Při testování testují jen konkrétní jednotku.
     - Neočekává se, že k testování bude použit zbytek aplikace.
     - Simulují například činnost databáze, síťové zdroje a podobně.
   - Obvykle pojmenované jako `TestXXX` (kde XXX je jméno testované jednotky).

Testování programů – Úrovně testování (2)

2. **Integrační testy**
   - Testují větší moduly vytvářené aplikace.
3. **Systémové testy**
   - Obvykle vytvářeny „testry“
   - Testují systém jako by byl nainstalován uživateli.
   - Očekává se, že jsou přítomny všechny prostředky nutné pro běh aplikace (databáze, síťové zdroje,…).
   - Testy funkcionality, uživatelského rozhraní, bezpečnosti,…
4. **Zákaznický test** (Customer tests, Acceptance tests)
   - Testují hotový systém.
   - „Black-box testing“ celého produktu, výsledkem je zda je produkt možno předat zákazníkovi.

Testování programů – Úrovně testování (3)

- Alfa
  - Provádí se před zveřejněním produktu
    - Vývojáři
    - „Výstupní kontrola“
- Beta
  - Poskytnutí produktu vybrané skupině externích uživatelů, získání zpětné vazby.
- Gama
  - Kompletní dílo, které zcela neprošlo interní kontrolou kvality

Testování programů – Funkce prostředí pro testování

- Prostředí pro testování by minimálně mělo:
  - umět spustit série testů;
  - rozhodnout, zda testy proběhly úspěšně. Jednotlivé testy by měly být prováděny nezávisle na ostatních;
  - v případě, že test skončil chybou určit proč;
  - sumarizovat získané výsledky.
- V ideálním případě by prostředí pro testování by mělo být nezávislé na vlastních testech.

Testování programů – Příprava před testováním

- Plánování testů
  - Rozdělení do skupin podle toho, jaké části aplikace testují.
  - Rozdělení do skupin tak, aby mohly být testy prováděny paralelně.
- Příprava dat
  - Simulace různých zdrojů – databáze,…
  - Generování náhodných dat.
- Příprava prostředí pro testování.
- Definování zodpovědnosti za části tvořené aplikace.

Testování programů – Spuštění testů

- Provedení jednoho, nějaké skupiny, všech testů.
- Jsou-li testy prováděny paralelně a nebo jsou některé činnosti prováděny na pozadí může být nutné tyto činnosti synchronizovat.
- Pokud používáme více počítačů a nebo více platforem může nám prostředí pro testování pomoct při řízení těchto strojů a nebo stírá rozdíly mezi platformami.
- Uchování výstupu nebo vstupu testu.

Testování programů – Po skončení testování

- Vygenerování zprávy, která shrnuje výsledky testu.
  - Přehledný, úplný, flexibilní,…
  - Měl by poskytovat přehled, které soubory a jak byly testovány.
- Měli bychom být schopni rozlišit mezi neúspěšně provedeným testem a chybě při testování.
- Měli bychom být schopni uměle vytvořit nalezenou chybu.

Testování programů – JUnit (1)

- prostředí Java, varianty i pro jiné jazyky (NUnit, CPPUnit,…)
- programátorské rozhraní pro tvorbu testů
- různé nástroje pro automatické provádění testů
  - grafické rozhraní
  - příkazový řádek, akce pro Ant
  - podpora v řadě IDE
- využití reflexe pro vyhledání testů uvnitř tříd
- rozšíření pro testování webových aplikací, servletů, databázových aplikací...

Testování programů – JUnit (2)

- `import junit.framework.TestCase;`
- `public class TestXYZ extends TestCase { }`

- `void setUp() {…}` — *inicializace*
- `void tearDown() {…}` — *finalizace*
- `void testX() {…}` — *krok testu*

- `assertTrue(`*`podmínka`*`), assertFalse(…)`
- `assertEquals(x,y), assertNotNull(x)`
- …

Testování programů – JUnit (3)

```
package cviceni3;
import junit.framework.TestCase;

public class TestZlomek extends TestCase {
   protected Zlomek z1 = new Zlomek(1, 3);
   protected Zlomek z2 = new Zlomek(2, 6);
   protected Zlomek z3 = new Zlomek(2, 3);

   protected void setUp()  { }
   protected void tearDown()  { }
   public void testEquals()
   {
      assertEquals(z1, z1);
      assertEquals(z1, z2);
   }
   public void testAdd()
   {
      Zlomek result = Zlomek.plus(z1, z2);
      assertEquals(result, z3);
   }
}
```

Testování programů – Jak napsat testy v JUnit(1)

- Obecně chceme, aby testy ověřovaly funkcionalitu nějaké části vyvíjeného produktu.
  - Informace co testovat získáme spíše z popisu funkcionality než ze zdrojových kódů.
- Test by měl odhalit, pokud testovaná část neimplementuje „popsané“ funkce.

Testování programů – Jak napsat testy v JUnit(2)

- Příklad: Testujeme metodu `isEmpty()` třídy `Vector`.
  - Metoda vrátí `true` v případě, že vektor je prázdný a `false` v případě že není.
- Můžeme test napsat takto:

```
public void testIsEmpty () {
  Vector vector=new Vector();
  assertTrue(vector.isEmpty());
}
```

  - Špatné řešení! Takováto implementace metody `isEmpty` by testem prošla a přitom neodpovídá popisu!

```
public boolean isEmpty() { return true;}
```

  - Lepší řešení by bylo otestovat obě varianty výstupů.

Testování programů – Jak napsat testy v JUnit(3)

- Konkrétní řešení závisí na testovaném problému!
- Obecné rady
  - Pokud testovaná metoda vrací víc „typů“ výsledku (například metoda compareTo() třídy Integer vrací -1, 0, 1) otestujte všechny varianty.
  - Obvykle nejsme schopni otestovat všechny varianty vstupů a výstupu.
    - Ověříme nejčastěji používané.
    - Otestujeme „krajní meze“ vstupů a výstupů.

Testování programů – Další vlastnosti JUnit (1)

- Testy lze „združovat“ pomocí třídy `TestSuite`

```
public static Test suite() {
    TestSuite suite = new TestSuite();
    suite.addTestSuite(cviceni2.TestPredmet.class);
    suite.addTestSuite(cviceni2.TestPredmetIO.class);
    return suite;
}
```

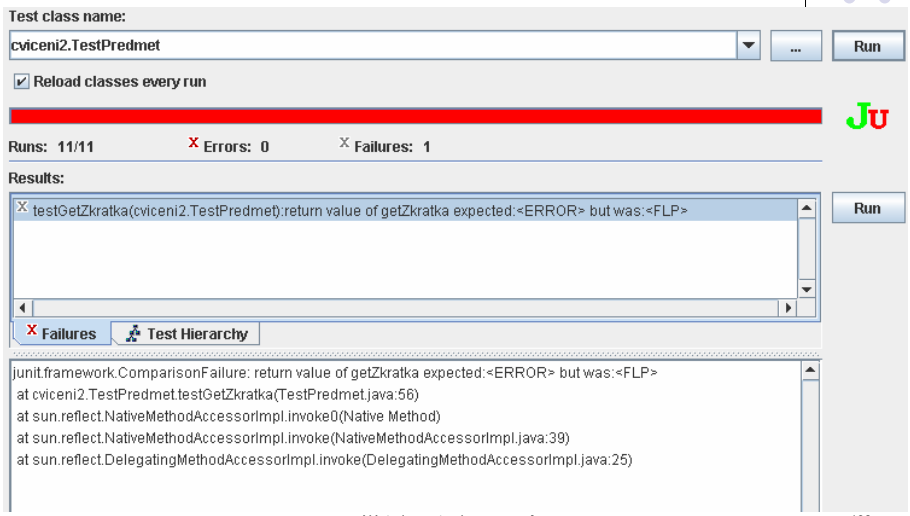
- Výsledek testu lze získat pomocí třídy `TestResult`

```
TestResult result = new TestResult();
suite.run(result);
result.wasSuccessful();
```

- Další informace naleznete na:http://junit.sourceforge.net/javadoc/
- Aplikace ke stažení na: http://xprogramming.com/software.htm

Testování programů – Další vlastnosti JUnit (2)

Testování programů – Další vlastnosti JUnit (3)



Testování programů – Další vlastnosti JUnit(4)

- JUnit verze 4.x
  - Java 1.5
  - Použity anotace
  - Testy
    - @Test public void testVyhledej() { … }
  - Inicializace
    - @Before protected void setUp() throws Exception {…}
    - @After protected void tearDown() throws Exception {…}

Testování programů – Další vlastnosti JUnit (5)

```
import static org.junit.Assert.*;
public class PoleTest {
   @Before protected void setUp() throws Exception {
      pole = new Pole();
   }
   @After protected void tearDown() throws Exception {
      pole = null;
   }
   @Test public void vratPocet() {
      …
      assertEquals("return value", expectedReturn, actualReturn);
   }
}
```

Testování programů – Další nástroje pro testování aplikace (1)

- Analyzátory paměti
  - Uchovává informace o tom, jak a kolik paměti bylo při běhu aplikace použito.
  - Purify, Electric Fence, …
- Coverage tools
  - Nástroje zjišťující jak velká část aplikace je použita při testování.
  - Výsledek může být zdrojový kód, který nebyl při testování použit.
  - Další možností (branche coverage) je, které části podmínek nebyly provedeny respektive, které větve programu nebyly použity.

Testování programů – Další nástroje pro testování aplikace (2)

- Testování výkonnosti
  - Sledování počtu volání určitých funkcí
  - Sledování času stráveného výpočtem různých částí programu
    - podklad pro optimalizaci změnou algoritmu
  - Nástroje označované jako **profilery.**
  - Bývají součástí vývojových prostředí. Profilaci může podporovat překladač a nebo lze použít samostatný program jako: gprof.

Testování programů – Další nástroje pro testování aplikace (3)

- Analyzátory kódu (*static code analyzers*)
  - Testují různé statické vlastnosti zdrojových kódů
    - Jak přesně splňují normy pro daný programovací jazyk (ANSI C).
    - Bezpečnost – hledá potencionálně nebezpečné příkazy.
    - Korektnost – některé jazyky umožňují matematicky dokazovat vlastnosti (funkcionální jazyky) nebo hledají vzory častých chyb (FindBug)
    - Velikost případně komplexnost
    - Analýzy dokumentace, která je součástí zdrojových kódů.
    - „Stabilita“ API – jak často se v čase mění

Testování programů – Vývoj řízený testy (TDD)

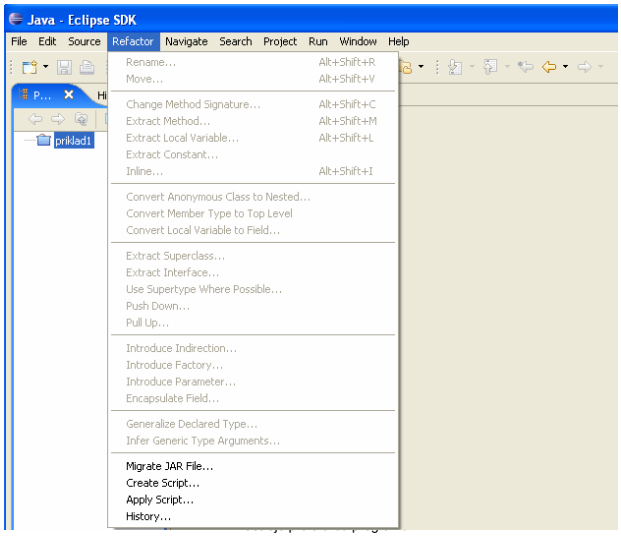
1. Napíšeme testy
2. Napíšeme program
3. Spustíme automatizované testování
4. Provedeme refaktorizaci
5. Opakujeme až do odstranění všech chyb

- Hlavním cílem je dosáhnout toho, aby všechny testy prošly
  - Mohou být ale chybné testy!

Testování programů – Refaktorizace (1)

- Transformace zdrojového kódu, která vede ke zlepšení jeho čitelnosti nebo struktury, avšak bez změny významu nebo chování.
  - Přejmenování proměnné
  - Vytvoření podprogramu ze zadaného úseku programu.
  - Nahrazení posloupnosti příkazu *if* polymorfismem.
  - …
- Důležitá součást metodiky TDD, XP
- Nástroje pro refaktorizaci bývají součástí IDE.

Testování programů – Refaktorizace (2)



Sledování chyb – Proč používat

1. Najdeme chybu v aplikaci.
2. Výskyt chyby je třeba zaznamenat tak, aby se dala reprodukovat
   - Jaké kroky k výskytu chyby vedly?
   - Jaké bylo očekávané chování?
   - Jaké bylo skutečné chování?
   - Jaké jsou důsledky a závažnost chyby?
3. Je potřeba evidovat:
   - které chyby jsou aktuálně řešeny (nevyřešeny);
   - kdo je zodpovědný za odstranění chyby;
   - …

Sledování chyb (1)

- Nástroj, který umožňuje uložit informace o chybách a rozliší jednotlivé chyby (jednoznačně je identifikuje).
- Nástroj využívají všichni členové týmu.
  - Nástroj pomáhá celé skupince lidí pracovat na řadě malých problémů (jednotlivých chybách).
- Celá řada nástrojů
  - Bugzilla, GNATS, FogBugs, JIRA, TestTrack,…
  - Většina nástrojů jsou informační systémy, které pro uložení dat používají databázi a nejčastěji komunikují přes webové rozhraní.

Sledování chyb (2)

- Dobrý nástroj pro sledování chyb by měl:
  - uchovávat informace o chybě, včetně stavu ve kterém je (nevyřešena, řešena, …);
  - umět pracovat se skupinami chyb (odstranění komplexnějšího problému);
  - vyhledat chyby a sledovat aktuální změny;
  - generovat statistiky a komplexnější zprávy o sledovaných chybách;
  - podporovat historii jednotlivých chyb a být schopen spojit chyby s příslušnou verzí dané aplikace (integrace s SCM);
  - rozlišovat závažnost chyby;
  - …

Sledování chyb – Bugzilla

- Asi nepoužívanější volně dostupný nástroj pro sledování chyb.
- Napsána v Perlu, komunikuje přes internetový prohlížeč a používá email ke komunikaci.
- Implementuje běžné funkce očekávané od nástroje pro sledování změn.
  - Vyhledávání - regulární výrazy, boolovské výrazy
  - LDAP, historie změn každé chyby, podpora závislostí mezi chybami,
  - „hlasování“ pro určení „otravných“ chyb
  - …

Generování dokumentace

- Dokumentace je vytvářena současně s aplikací
  - Oddělená dokumentace
    - problémy s aktualizací
    - nutnost uvádět kompletní specifikace
- Dokumentace určená pro:
  - uživatele aplikace;
  - pro potřeby tvůrců aplikace.
- Dokumentace jako součást zdrojového textu
  - snadnější údržba (např. včetně verzování)
  - literární programování (D. Knuth)
  - dokumentační značky - javadoc

Generování dokumentace - Program javadoc

- Dokumentace ve speciálních poznámkách

```
/**
 * Dokumentační poznámka
 */
```

- Dokumentační značky + HTML

  @author Joe
  @param x Popis parametru x
- Rozšíření – generování zdrojových textů pomocí šablon - XDoclet

Generování dokumentace - Program javadoc

```
/**
 * Konstruktor zlomku.
 * Naplní čitatele a jmenovatele a převede
 * zlomek do normalizovaného tvaru.
 * @param citatel    Čitatel zlomku.
 * @param jmenovatel Jmenovatel zlomku.
 */
public Zlomek(int citatel, int jmenovatel)
{
   this.citatel = citatel;
   this.jmenovatel = jmenovatel;
   normalizuj();
}
```


Nasazení aplikace

- Může jít o náročný proces
  - Konfigurace okolního prostředí
  - Nastavení parametrů aplikace
  - Propojení s jinými aplikacemi
- Generátory instalačních balíčků
  - Mohou být platformě závislé i nezávislé.
  - Nástroje pro automatické nasazení aplikace.
  - Antigen, Advanced Installer, IzPack,…

Tvorba aplikací pro mezinárodní prostředí

- úprava programů pro mezinárodní prostředí
- Internacionalizace (i18n)
  - Zajištění takových vlastností aplikace, aby byla potenciálně použitelná kdekoliv.
- Lokalizace (l10n)
  - Přizpůsobení aplikace konkrétnímu jazykovému a kulturnímu prostředí.

Tvorba aplikací pro mezinárodní prostředí - **Internacionalizace (i18n)**

- Formát data a času
- Zápis čísel
- Měna
- Jazyk (abeceda, číslice, směr psaní, …)
- Telefonní čísla, adresy, PSČ
- Míry a váhy

Tvorba aplikací pro mezinárodní prostředí - Lokalizace

- Jazykové verze, překlady
- Zvyklosti
- Symbolika
- Estetika
  - Barvy
  - Ikony
- Kulturní hodnoty, sociální kontext

Tvorba aplikací pro mezinárodní prostředí - Nástroje pro i18n/l10n

- překlad textových řetězců
  - extrakce textů – překlad – vložení zpět
  - oddělení textů od programu
    - např. soubory .properties v různých jazykových verzích
  - knihovna gettext – pro různé programovací jazyky
- Přístup k informacím o prostředí
  - setlocale(), getlocale(), …