KIV/ZSWI 2003/2004
Přednáška 13

Testování systému

* účelem otestovat celý systém, jehož je SW součástí
* mívají různý účel, např. otestovat vlastnosti jako je výkonnost, kompatibilita, bezpečnost, instalovatelnost, spolehlivost apod.

Testování výkonnosti

* v mnoha typech systémů je nepřípustné, aby SW nesplňoval požadavky na výkonnost (zejména v řídících systémech)
* v takovém případě by se výkonnost měla testovat ve všech krocích včetně jednotkových testů
* pomocné procedury monitorují dobu vykonání apod.

Zátěžové testování

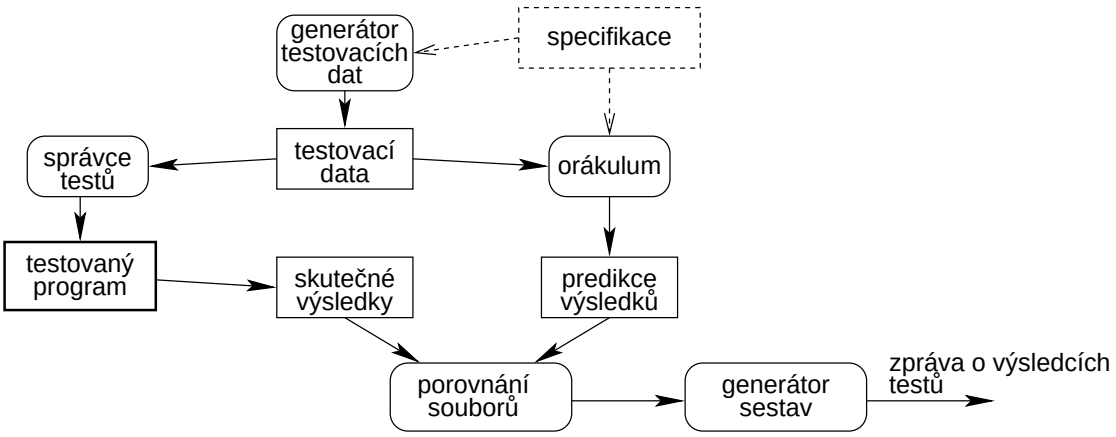
* obvykle se používají testy, kde se zátěž postupně zvyšuje, dokud není výkonnost systému neakceptovatelná nebo dokud systém nehavaruje
  - zátěž = množství dat, frekvence požadavků, data která jsou extrémně náročná na zpracování
  - je vhodné určit části kódu, které mohou být problematické při velké zátěži, zátěžové testy navrhnout tak aby pokrývaly především tyto části kódu
  - ověření zda havárie systému nepoškodí data apod.
  - může odhalit některé defekty, které se normálně neprojeví
  - důležité zejména u internetových aplikací, v distribuovaných systémech apod., kde se vysoce zatížené systémy mohou zahltit, protože si vyměňují čím dál více koordinačních dat, čímž se opět zvyšuje zátěž systému atd.

Testování zotavení systému po havárii

* mnoho systémů se musí být schopno zotavit po havárii v předepsaném čase, jinak hrozí značné finanční ztráty
  - při testování zotavení systému způsobujeme různé havárie systému a ověřujeme, zda se systém zotavil správně a v časovém limitu

Nástroje pro testování SW

* v rozsáhlých systémech může cena testování dosáhnout 50% ceny vývoje, mohou existovat stovky až tisíce testovacích případů
  - proto potřeba automatizace testů
  - automatizace testů také snižuje cenu změn - programátoři se nemusejí tolik obávat, že změnou zanesou do kódu defekt, protože testy by defekt (s určitou pravděpodobností) odhalily
  - obvyklé je následující uspořádání:



* jednotlivé programy mají následující fce:
 - správce testů (test manager) - řídí běh testů
 - generátor testovacích dat (test data generator) - generuje testovací vstupní data pro testovaný SW
 - orákulum (oracle) - generuje předpokládané výstupní hodnoty
  . orákulum lze vyvinout jako nový program (obsahující podmnožinu funkčnosti, co nejméně pracná implementace)
  . často lze využít prototyp SW, předchozí verze SW, SW vytvořený konkurencí apod.
  . pokud se jako orákulum používá předchozí verze SW, používá se název regresivní testování (regression tests = porovnáváme výsledky staré a nové verze, rozdíly znamenají potenciální problém v nové verzi)
 - program pro porovnání souborů (file comparator) - porovná výsledky skutečného běhu s předpokládanými hodnotami vygenerovanými orákulem; často lze použít univerzální programy jako cmp(1) a diff(1) v UNIXu apod.
 - generátor zpráv (report generator) - umožňuje definovat a generovat zprávy o výsledcích testů

Poznámka (regresivní vs. progresivní testování)

 Proces testování má svou progresivní i regresivní fázi. Progresivní fáze testování přidává a testuje nové fce (nově přidané nebo modifikované moduly a jejich rozhraní s již integrovanými moduly). Regresivní fáze testuje důsledky změn v již integrovaných částech.

[]

* jako jednoduchou ukázku uvedu část příkazového souboru pro příkazový interpret (shell) v UNIXu, který provádí testování aplikace:

```
test-mkdata > data.test                  # Vytvoří testovací data ”data.test”.
test-oracle < data.test > result1.test   # Orákulum předpoví výsledky ”result1.test”.
aplikace    < data.test > result2.test   # Testujeme aplikaci.
if cmp result1.test result2.test; then   # Porovnáme skutečné a předpovězené výsledky.
   echo Test 1: PASSED                   # Soubory stejné -> test prošel.
else
   echo Test 1: FAILED                   # Soubory rozdílné -> test neprošel.
fi
```

* nástroje pro zachycení a pozdější přehrání testů (capture-replay tool)
 - informace protékají SW systémem - na vhodné místo toku dat aplikací vložíme nástroj, který tekoucí data zaznamená
 - tester spustí/vykoná a zaznamená testovací případy
 - později může zaznamenané testovací případy spustit znovu (regresivní testování)
 - existují komerční capture-replay nástroje pro zachycení/přehrání stisků kláves a pohybů myši, obsahující i nástroje pro zachycení a porovnání výstupů na obrazovku - používané pro GUI aplikace
  . pro většinu ostatních účelů (testování zapouzdřených systémů apod.) je většinou nutné vytvořit si vlastní SW pro podporu testování

* kolik defektů lze očekávat a kolik z nich lze najít testováním?
 - podle kvality vývoje lze očekávat asi 10 až 50 defektů na 1000 řádek kódu před testováním
  . např. Microsoft Application Division 10-20 defektů/1000 řádek kódu (Moore 1992)
 - testy najdou obvykle méně než 60% defektů, proto je vhodné je kombinovat s inspekcemi
 - někteří autoři uvádějí, že defekty jsou častěji v testovacích případech než v testovaném kódu (McConnell 1993)
 - pro kritické systémy je vhodné používat kombinaci formálních metod vývoje, inspekcí a testování (např. pro Cleanroom metodiku v průměru 2.3 defektů/1000 řádek kódu, pro některé systémy až 0 defektů)

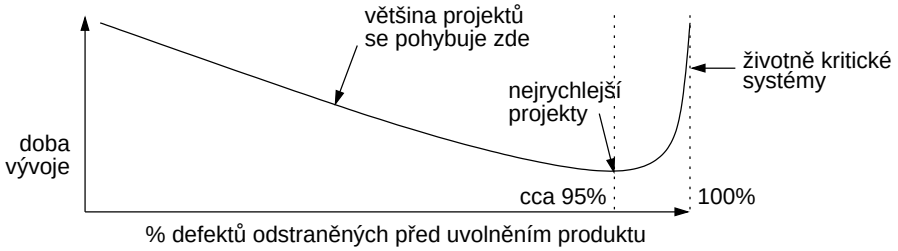
Poznámka (kvalita SW a rychlost vývoje)

 Většina managerů se snaží zkrátit dobu vývoje omezením času věnovanému inspekcím návrhu a kódu, testování apod. Zejména testování bývá často

obětováno, protože je na konci vývojového cyklu (tj. blízko deadline).

Podle dostupných studií (shrnutých např. v DeMarco a Lister 1987, McConnell 1996 aj.) tento chybný přístup celkovou dobu vývoje naopak prodlužuje. Jinými slovy, kvalitnější produkt bude dříve dokončen. Pokud obětujeme kvalitu, vývoj se tím prodlouží a prodraží.

Ve skutečnosti vypadá vztah mezi dobou vývoje a počtem defektů přibližně následovně:



[]

Ladění

* testováním nalezneme defekty, následuje proces ladění (angl. debugging)
* ladění = identifikace příčiny defektu (90% času) a její oprava (10% času)
* mělo by probíhat v 5 krocích - (1) stabilizace symptomu, (2) nalezení příčiny, (3) oprava, (4) otestování opravy defektu, (5) vyhledání obdobných defektů

* stabilizace symptomu - potřebujeme, aby se defekt projevoval spolehlivě
  - proto nejprve hledáme testovací případ, který symptom reprodukuje
  - testovací případ co nejvíce zjednodušíme, aby se defekt ještě projevoval
  - při zjednodušování testovacího případu často již můžeme vytvářet hypotézu, proč defekt nastává
  - existují defekty, které se neprojevují spolehlivě, například neinicializované proměnné, neplatný ukazatel, chyby časového souběhu
    . některé z těchto defektů můžeme zviditelnit; například neinicializované proměnné - před spuštěním programu paměť zaplnit náhodnými hodnotami apod.
    . chyby časového souběhu lze zviditelnit pomocí vložení yield(), případně náhodného prokládání (o chybách časového souběhu - viz předmět ZOS)

* nalezení příčiny symptomu
  - např. hledání zúžením podezřelé části kódu: systematicky vynecháváme kód (i na úkor funkčnosti), testujeme zda se symptom ještě vyskytuje
    . adaptace metody binárního vyhledávání - vynechá se přibližně polovina kódu, pokud se symptom projevuje opět rozdělíme na poloviny atd.
    . vynechávání volání podprogramu
  - použití ladícího programu nebo ladících výpisů - sledujeme kde nastane symptom
    . přeskakujeme ty části programu které nejsou relevantní, můžeme použít obdobné metody jako výše (aniž bychom vynechávali kód)
  - někdy pomůže se vyspat (podvědomí pracuje za nás)

* oprava defektu
  - pokud jsme defekt nalezli, bývá jeho oprava poměrně jednoduchá, ale je vysoké nebezpečí zanesení dalšího defektu (podle některých studií více než 50%)
  - podle studie z 1986 mají větší šanci provést opravu správně programátoři s celkovou znalostí programu (oproti programátorům kteří se seznámí pouze s opravovanou částí programu)
  - proto je před opravou třeba rozumět jak problému, tak opravovanému programu

* opravu je třeba otestovat
  - defekty je nutné opravovat po jednom, opravy po jedné otestovat
  - poté program otestovat jako celek, nejlépe regresivními testy, aby se ukázaly případné vedlejší efekty opravy

- pro případ defektu v opravě je vhodné mít uchovánu předchozí verzi (ručně, SCM), porovnat obě verze (např. v UNIXu programem diff(1)), z toho je často možné zjistit problém.

* hledáme obdobné defekty

Defenzivní programování

* defenzivní programování = zabezpečení definovaného chování při chybných vstupech, zamezení propagace chyb z podprogramu ven

* nástroje:
 - kontroly vstupních parametrů
 - makro assert() v C, aserce v Javě), preconditions/postconditions v Eiffelu (programming by contract)
  . pokud nejsou, snadno si naprogramujeme jejich ekvivalent, např. v Pascalu:

```
procedure Assert(Condition: boolean; Message: string);
begin
     if (not Condition) then
     begin
          writeln('Assertion ', Message, ' failed. Aborting the program');
          halt;
     end
end;
```

  . použití např.:

```
Assert(delitel <> 0, 'delitel <> 0');  (* dělitel nesmí být roven 0 *)
```

 - výjimky v Javě, C++, Delphi, Pythonu apod. - konstrukce typu try-catch a try-catch-finally:

```
try {
    foo();                                  // zde může nastat výjimka
} catch (SomethingWentWrongException e) {   // pokud nastane výjimka,
    System.out.println("some error");       //  provede se toto
} finally {                                 // nakonec se vždy provede
    dispose();                              //  toto
}
```

 - program může při spuštění ověřovat své datové struktury, volání fcí apod.

* reakce na chybu
 - fce vrátí speciální návratový kód
 - programová výjimka, způsob propagace - podle stanovených konvencí
 - nastavení defaultní hodnoty vstupu nebo defaultního stavu
* o způsobu reakce na chybu se by mělo rozhodnout na úrovni architektury, aplikace by se měla ve všech svých částech chovat konzistentně

Poznámka (rozdíl mezi vývojovou verzí produktu a dodanou verzí)

Zatímco během vývoje potřebujeme, aby defekty byly co nejlépe viditelné, ve výsledném produktu se naopak snažíme, aby defekty co nejméně rušily. Ve výsledném produktu proto:

* ponecháváme kód který kontroluje významné defekty; pokud aplikace hlásí interní chybu, měla by také oznámit, jakým způsobem jí uživatel může ohlásit
* zrušíme kód testující nepodstatné defekty,
* zrušíme kód který způsobuje havárie
* ponecháme kód který umožní přijatelné ukončení aplikace při chybě (např. s uložením dat)

Rušení kódu neprovádíme fyzicky (při ladění ho budeme opět potřebovat), ale např. pomocí preprocesoru vynecháme tělo procedury Assert, využijeme verzování apod.

Např. v jazyce C definicí makra NDEBUG změníme všechny výskyty assert()
na prázdný příkaz.

[]

Údržba SW systémů

* údržba SW = aktivity, které jsou prováděny po uvolnění programu, resp. po jeho dodání zákazníkovi

* údržba zahrnuje především tyto tři typy aktivit:
 - oprava chyb SW (corrective maintenance)
 - přizpůsobení SW změnám prostředí, ve kterém běží - OS, periferie apod. (adaptive maintenance)
 - přidávání nové funkčnosti nebo změna funkčnosti na základě požadavků uživatele (perfective maintenance)

* v průměru cca 17% údržby se týká opravy chyb, 18% přizpůsobení SW změnám prostředí a 65% přidávání nebo změny funkčnosti
* údržba tvoří cca 50% až 75% vývoje, pro obtížně změnitelné systémy (jako jsou zapouzdřené systémy reálného času) až 80%
 - studie ukazují že cena údržby systému postupně stoupá
 - proto je efektivní systém navrhnout a implementovat tak, aby se cena údržby snížila

* proces údržby je spuštěn množinou požadavků na změny od uživatelů systému, managementu nebo od zákazníka
 - provedeme vyhodnocení ceny a dopadu změn
 - navržené změny jsou schváleny nebo neschváleny; některé jsou odloženy
  . rozhodnutí o schválení/neschválení změn je do určité míry ovlivněno udržovatelností komponenty, ve které změnu provádíme
 - změny jsou implementovány a ověřeny
  . v ideálním případě: změna specifikace systému, návrhu, implementace, otestování systému
 - je dodána nová verze
* už bylo probíráno, viz ”Správa požadavků” na třetí přednášce

Nestrukturovaná údržba

* výše uvedené by ovšem platilo v ideálním případě
* pracnost a cenu údržby ve skutečnosti zvyšují tyto faktory:
 - po dodání produktu je tým obvykle rozpuštěn, lidé jsou přiřazeni jiným projektům; údržba je přenechána jinému týmu nebo jednotlivcům, kteří systém neznají => většina jejich úsilí musí být věnována pochopení existujícího systému
 - smlouva bývá obvykle pouze na dodání systému, o údržbě se nemluví => tým nemá motivaci vytvářet udržovatelný SW (zvláště pokud se předpokládá, že údržbu bude provádět někdo jiný)
 - údržba je obvykle přenechána nejméně zkušeným programátorům, navíc systémy mohou být vytvořeny v zastaralých programovacích jazycích (Cobol), které se tým provádějící údržbu musí teprve naučit
 - změnami se snižuje strukturovanost kódu - tím roste ”entropie” SW, dokumentace starých systémů může být ztracena nebo může být nekonzistentní atd.

* v mnoha případech je jediným dostupným prvkem SW konfigurace zdrojový text
 - proto proces údržby začíná (obtížným) procesem vyhodnocení kódu
  . kód obsahuje pouze implementaci, nikoli záměr => obtížná interpretace
  . pokud neexistují testy, není možné regresivní testování
 - takovému případu říkáme nestrukturovaná údržba (unstructured maintenance), produktivita klesá na 40:1 oproti vývoji (podle Boehma)
 - tomuto stavu se snažíme předejít, pokud je to možné

* v zásadě dvě možnosti jak se k problému postavit:
 - předpokládat vodopádový model - systém vyvinout, udržovat dokud se údržba stane ekonomicky neúnosná, pak nahradit novým systémem

- evoluce systému - předpokládat např. spirálový model
  . systém by měl být navržen tak, aby ho bylo možné přizpůsobovat novým požadavkům
  . pokud systém nevyhovuje (= nízká udržovatelnost), musíme systém přepracovat (přestrukturovat apod.)

* udržovatelnost systému lze měřit následujícími metrikami:
- počet požadavků na opravy chyb
  . pokud počet požadavků na opravy stoupá, může to znamenat, že do programu je procesem údržby vnášeno více chyb než je jich odstraňováno údržbou, tj. indikuje snížení udržovatelnosti
- průměrná doba potřebná pro vyhodnocení dopadu změny
  . odráží rozsah (např. počet komponent) které jsou ovlivněny požadavkem na změnu
  . pokud naroste, udržovatelnost se snižuje
- průměrná doba implementace požadavku na změnu
- počet nevyřízených požadavků na změnu

* pokud je program špatně udržovatelný, jsou možné následující kroky pro zlepšení:
- konverze SW do moderního programovacího jazyka (nebo do modernější varianty použitého programovacího jazyka); například z Fortranu do jazyka Java nebo C#
- zpětné inženýrství = analýza programu s cílem najít specifikaci a návrh SW
  . obvykle na základě zdrojových textů, v některých případech jsou ztraceny a je nutné vycházet ze spustitelného kódu
- vylepšení struktury programu s cílem zlepšit jeho srozumitelnost
  . pro automatickou transformaci nestrukturovaného kódu na strukturovaný existují nástroje (při transformace ale ztratíme původní komentáře)
- modularizace programu = sdružení souvisejících částí programu, v některých případech včetně transformace architektury SW
  . stejná motivace jako při vytváření modulů při vývoji, tj. abstrakce dat, abstrakce řízení HW, sdružení příbuzných fcí
- přizpůsobení zpracovávaných dat změněnému programu

O postupech při přepracovávání existujících systémů hovoří kniha Martina Fowlera ”Refactoring: Zlepšení existujícího kódu. Grada, Praha 2003.”

Metriky

* metrika (metrics) = jakékoli měření atributů SW produktu nebo SW procesu
- např. počet řádků kódu, počet defektů, defekty na 1000 řádek kódu apod.
- jsou potřebné, abyste věděli co se s projektem opravdu děje, pro zlepšování SW nebo SW procesu, pro odhady o budoucích projektech, pro informaci kam zaměřit testování atd.
- např. před zavedením testovacího nástroje můžeme změřit počet defektů nalezených za časovou jednotku, totéž po zavedení nástroje
- jiný příklad - nejvíce defektů bývá v procedurách/metodách s vysokou cyklomatickou složitostí - tam bychom měli zaměřit své úsilí při testování

* nejdůležitější metriky se týkají následujících oblastí:
- metriky analytického modelu (např. kvalita specifikace)
- metriky návrhu (např. složitost komponent)
- metriky zdrojových textů (např. níže uvedené LOC a cyklomatická složitost)
- metriky pro testování (např. pokrytí logiky programu, efektivita testování)

* nejjednodušší a nejčastější metrika zdrojových textů - počet řádek kódu (Lines of Code, LOC)
- otázka - co máme počítat jako řádek kódu?
- pokud se snažíme měřit množství práce do programu vložené, nebudeme započítávat komentáře, prázdné řádky a automaticky generovaný kód
- někdy se nazývá NCLOC (Non-comment LOC) nebo ELOC (Effective LOC)
- je základ metodiky COCOMO pro odhad ceny SW

* McCabeho metrika ”cyklomatická složitost” počítá rozhodovací body v podprogramu
- vysoká cyklomatická složitost má korelaci s chybovostí, s obtížností číst a testovat podprogram (jak už bylo řečeno, prostá délka podprogramu je

nemá korelaci s chybovostí)
- cyklomatickou složitost spočteme následovně:

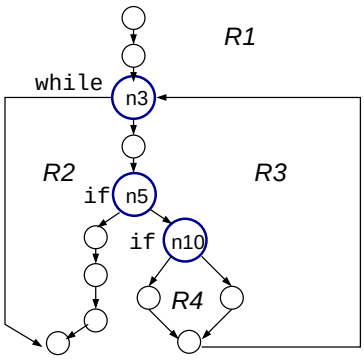
1. Pro přímou cestu podprogramem započteme 1.
2. Pro každé z následujících klíčových slov nebo jejich ekvivalentů přičteme jedničku: if, while, do-while, for, a pro ”and a ”or” ve složených podmínkách.
3. Pro každý případ v switch/case přičteme 1.

- pokud je >10, je příznak že je vhodné podprogram zjednodušit (např. část podprogramu vložit do dalšího podprogramu volaného z původního)

Poznámka pro zajímavost (cyklomatická složitost a graf toku řízení)

Pokud v rozhodovacích příkazech nejsou složené podmínky, odpovídá cyklomatická složitost počtu regionů v grafu toku řízení podprogramu. Pojmem region se myslí oblast ohraničená hranami a uzly grafu; oblast mimo graf se počítá jako samostatný region.

Jako příklad uvádím graf toku řízení pro příklad binárního vyhledávání, uvedený na minulé přednášce:



Pro kontrolu si můžeme spočíst cyklomatickou složitost pomocí výše uvedené metody: započteme přímou cestu, while, if a if, tj. výsledek je opět 4.

[]

Některé užitečné metriky
.....

* velikost zdrojových textů
 - celkový počet řádek (LOC)
 - celkový počet komentářových řádek (CLOC)
 - počet deklarací dat
 - počet prázdných řádek

* produktivita
 - E-faktor (environmental factor) = počet nepřerušených hodin / celková pracovní doba; optimální je cca 0.4
 - pracovní doba strávená na projektu
 - pracovní doba strávená na každém podprogramu
 - počet změn v podprogramu
 - náklady projektu
 - náklady projektu na řádek kódu
 - náklady na opravu defektu

* sledování defektů
 - vážnost defektu
 - místo defektu
 - způsob opravy defektu
 - osoba zodpovědná za defekt
 - počet řádek změněných při opravě defektu
 - pracovní doba strávená opravou defektu
 - průměrná doba potřebná na nalezení defektu
 - průměrná doba potřebná na opravu defektu
 - počet pokusů opravit defekt
 - počet nových chyb zanesených opravou defektu

* celková kvalita

```
 - celkový počet defektů
 - počet defektů v podprogramu
 - průměrný počet defektů na 1000 řádek kódu
 - střední doba mezi haváriemi
 - chyby detekované překladačem

* udržovatelnost
 - počet parametrů předávaných každému podprogramu
 - počet lokálních proměnných použitých každým podprogramem
 - počet podprogramů volaných každým podprogramem
 - počet rozhodovacích bodů v každém podprogramu
 - složitost řídících konstrukcí v každém podprogramu
 - počet řádek kódu v každém podprogramu
 - počet komentářů v každém podprogramu
 - počet deklarací dat v každém podprogramu
 - počet prázdných řádek v každém podprogramu
 - počet příkazů goto v každém podprogramu
 - počet vstupně/výstupních příkazů v každém podprogramu

* objektově orientované metriky
 - vážená složitost třídy
 - hloubka stromu dědičnosti
 - počet přímých potomků třídy
 - počet operací předefinovaných v každé podtřídě
 - stupeň provázanosti mezi třídami


Práce v týmech
==============

* většina profesionálních programátorů pracuje v týmech, týmy od 2 do
  několika set lidí
 - pokud je tým velký, je zřejmá potřeba aby byl nějak strukturován
   . rozdělení do skupin, každá skupina je zodpovědná za podprojekt
   . skupiny by neměly mít více než 8 členů
   . malý počet = zmenšení komunikačních problémů

Constantine (1993) popisuje čtyři paradigmata pro organizaci týmů:

* uzavřené paradigma - tým má pevně stanovenou hierarchii
 - příkladem je na první přednášce zmíněný ”chirurgický tým”
 - tyto týmy pracují dobře, pokud je SW podobný předchozím projektům, obvykle
   bývají méně inovativní

* náhodné paradigma - tým má volnou strukturu, role závisejí na iniciativě
  jednotlivých členů týmu (”tvořivá nezávislost”)
 - může mít vysokou výkonnost, pokud si členové mohou vzájemně důvěřovat,
   jednotliví členové mají přiměřené dovednosti a pokud neobsahuje rebely
 - vhodné pro inovativní projekty, často problémy pokud je vyžadována ”běžná
   práce”

* otevřené paradigma - tým založený na spolupráci, typicky značná komunikace
  a rozhodování založené na konsensu
 - vhodné pro řešení obtížných problémů, obvykle nebývá tak efektivní jako
   jiné typy týmů

* synchronní paradigma - závisí na možnosti rozdělit problém na nezávislé
  části
 - členové týmu pracují na jednotlivých podproblémech, členové mezi sebou
   nemusejí příliš komunikovat

Nezávisle na typu organizace týmu ovlivňují práci v týmu především 4 faktory:

* složení týmu: má tým vyvážené technické schopnosti, zkušenosti,
  osobnostní charakteristiky?
* koheze týmu: je tým množinou jednotlivců nebo má ”skupinového ducha”
  (skupina o sobě uvažuje jako o týmu)
* skupinová komunikace: dokáží spolu členové efektivně komunikovat?
* organizace týmu: má každý přiměřenou roli v týmu?
```

Složení týmu
.....

* v psychologické studii motivace (Bass & Dunteman) se ukázalo, že profesionály lze v zásadě rozdělit podle jejich motivace do tří kategorií:
 - úkolově orientovaní - motivací je jim práce, kterou vykonávají
  . při vytváření SW je motivací intelektuální výzva vytvořit SW
  . platí pro velkou část vývojářů
 - orientovaní na sebe - v zásadě motivováni osobním úspěchem a uznáním
  . vývoj SW je jim prostředkem k dosažení vlastních cílů
 - orientovaní na interakci - jsou motivovaní přítomností a činností spolupracovníků

* lidé orientovaní na interakci pracují raději ve skupinách, zatímco úkolově orientovaní a na sebe orientovaní obvykle raději pracují sami
* u žen je pravděpodobnější orientace na interakci než u mužů, často jsou efektivnější komunikátoři

* motivace jednotlivce se skládá ze všech tří kategorií, jedna z nich ale většinou převažuje
* osobnosti nejsou statické, motivace se může měnit
 . například pokud má úkolově orientovaný člen týmu pocit, že není přiměřeně odměňován, může se jeho motivace změnit na ”orientovaný na sebe”

* pro skupinu je dobré, pokud obsahuje doplňující se typy osobností:
 - úkolově orientovaní bývají obvykle nejsilnější technicky
 - na sebe orientovaní obvykle tlačí tým na dokončení práce (výsledky)
 - orientovaní na interakci napomáhají komunikaci uvnitř skupiny

* Proč optimalizovat složení týmu [převzato od P. Brady]

 Dobře vyvážený tým je velmi silná zbraň s enormní kapacitou k tvořivé práci - a proto je také drahý a musí být zatěžován odpovídajícími úkoly. Optimalizovat složení je tedy vhodné při vytváření nových skupin za účelem ambiciózních a náročných úkolů, a také pro týmy, které musí obstát v prostředí velkých změn, silné konkurence, potřeby rychlé inovace a akčnosti.

* Kdy složení týmu není kritické [převzato od P. Brady]

 Není vždy nutné snažit se optimalizovat složení týmu. Optimalizace není na místě v případech rutinních operací a úloh pod intelektuální úrovní ideálního týmu - takový tým by pro daný účel byl velmi drahý nehledě na to, že by práce neposkytovala jeho členům motivaci a uspokojení.

 Někdy ji nelze aplikovat z praktických či logistických důvodů - ne vždy je možnost vybrat lidi s požadovanými vlastnostmi aniž by bylo potřeba nabírat nové zaměstnance; je také možné že jsou k dispozici lidé s potenciálem, ale bez technických znalostí.

 Je vždy lepší se o vyvážení složení týmu pokusit částečně než vůbec. Pokud přesto není vyhovující, mohou se členové jeho nedostatky snažit nahradit bděním nad slabými aspekty s použitím ”nejlepších ze všech špatných” lidí a postupnou změnou.

* důležitá role je vedoucí skupiny
 - obvykle technické směřování a administrace projektu
 - sledují práci týmu, efektivitu

* vedoucí obvykle určeni managementem
 - problém - určení vedoucí nemusejí být vůdci skupiny po technické stránce
 - ve skutečnosti si skupina může najít ve svém středu např. technicky nejschopnějšího, nejlepšího motivátora
 - někdy je proto výhodné oddělit technické vedení od administrace projektu

Koheze skupiny
.....

* kohezivní skupina = pro členy jsou jejich individuální zájmy méně důležité než zájmy skupiny

- tj. členové se cítí být členy skupiny, snaží se skupinu chránit, pomáhat si navzájem apod.
- lze podpořit poskytováním informací a důvěry skupině

* výhody kohezivní skupiny:
- konsensem mohou vzniknout standardy kvality
- členové skupiny těsně spolupracují - mohou se od sebe navzájem učit apod.
- členové znají navzájem svojí práci (výhoda pokud např. člen skupiny onemocní)
- programy mohou být chápány jako skupinové vlastnictví (egoless programming) - usnadňuje provádění inspekcí, přijímání kritiky a vylepšování programu skupinou

* kohezivní skupiny mají náchylnost ke dvěma problémům:
- iracionální rezistence ke změně vedoucího - pokud je vedoucí nahrazen někým mimo skupinu, skupina se může sjednotit proti novému vedoucímu => snížení produktivity
  . pokud je to možné měl by být vedoucí zvolen z členů skupiny
- tzv. skupinové myšlení - kritické myšlení je potlačeno ve prospěch loajality vůči skupině (resp. skupinovým normám, skupinovým rozhodnutím)

Komunikace uvnitř skupiny

* dobrá komunikace mezi členy skupiny je podstatná
* podstatné faktory:
- velikost skupiny
- struktura skupiny
- složení
- fyzické pracovní prostředí

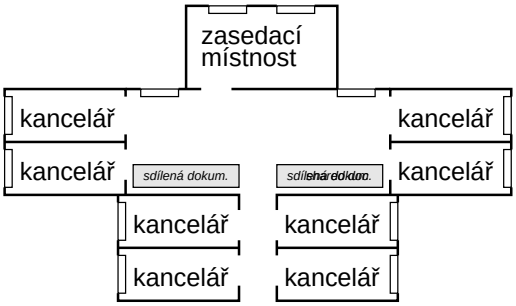
* klíčovým faktorem je velikost skupiny
- počet možných komunikačních cest je n * (n-1)
- tj. pro sedmičlennou skupinu (42 cest) je pravděpodobné že někteří členové spolu budou komunikovat velmi zřídka

* dalším faktorem struktura skupiny
- v neformálně strukturovaných skupinách efektivnější komunikace než ve formálně (hierarchicky) strukturovaných skupinách
- v hierarchicky strukturovaných skupinách mají informace tendenci putovat nahoru a dolů v hierarchii, ale lidé na některé úrovni spolu nekomunikují - problém velkých skupin

* složení skupiny
- pokud se skupina skládá z příliš mnoha lidí stejného osobnostního typu, nastávají konflikty a komunikace uvázne
- komunikace je obvykle lepší ve skupinách kde jsou muži i ženy než ve skupinách složených pouze z mužů nebo pouze z žen
- ženy jsou častěji interakčně orientované => mohou být prostředníci

* fyzické pracovní prostředí
- velmi podstatné pro chování a výkonnost skupiny
- podle (DeMarco & Lister 1985) rozdíly ve výkonnosti týmů až 1:11
- pro výkonnost nejdůležitější vlastnosti:
  . soukromí - potřeba prostoru, kde se mohou soustředit na práci bez vyrušování
  . přirozené světlo, viditelnost vnějšího prostředí (= okna)
  . možnost individuálních úprav prostředí podle způsobu práce

* pro komunikaci je podstatné, aby skupina mohla diskutovat projekt formálně i neformálně
- (Weinberg 1971) cituje případ, kdy organizace chtěla zabránit programátorům ”ztrácet čas” povídáním u automatu na kafe; po odstranění automatu se okamžitě dramaticky zvýšil počet formálních požadavků na výpomoc
- vyplatí se mít neformální místo pro setkávání, stejně jako konferenční místnost pro formální sezení



Softwarové profese
....

V rámci týmů vykonávají různí členové různou práci; všichni jsou stejně potřební, někteří ale nesou větší zodpovědnost.

Jako příklad uvedu rozdělení na profese podle (Paleta 2003):

* zadavatel nebo manager produktu - formálně není součástí týmu
 - sestavuje požadavky na vytvářenou aplikaci, zodpovídá otázky týmu, přebírá aplikaci
 - u systémů vytvářených na zakázku je zástupcem zadavatele

* vedoucí projektu - řídí vlastní vývoj, je zodpovědný za splnění požadavků, termínu a rozpočtu
 - komunikuje se zadavatelem nebo managerem produktu

* technický leader nebo architekt - ostatní se na něj obracejí, pokud mají technický dotaz
 - navrhuje celkovou strukturu aplikace, vybírá technologie a vývojové prostředky

* databázový specialista - návrh databáze
 - příprava výkonnostních testů a na základě jejich výsledků optimalizace databáze

* analytik - rozpracovává specifikaci, vytváří konceptuální model aplikace
 - navrhuje posloupnost obrazovek apod.
 - role může být kombinována s pozicí programátora nebo tvůrce dokumentace

* návrhář uživatelského rozhraní - navrhuje obrazovky aplikace tak, aby byly přehledné a snadno použitelné

* programátor - vytváří kód na základě specifikace a konceptuálního modelu

* tester - řídí nebo provádí testování

* tvůrce dokumentace - zpracovává technickou dokumentaci vytvářenou ostatními členy týmu, vytváří uživatelskou dokumentaci (manuály, nápověda)

Konfigurační management
=======================

* v SW projektech se mění požadavky na systém, design systému, kód, dokumentace systému atd.
 - v průběhu času vědí všichni zúčastnění více (o tom co potřebují, jak by se to nejlépe udělalo, atd.)
 - požadavky na změny budou přicházet ve všech fázích tvorby SW
 - pro řízení změn v projektech byly vyvinuty procesy, nazývané souhrnně konfigurační management (angl. software configuration management, SCM)
 - úkolem SCM definovat procedury pro provádění změn, eliminuje některé problémy vznikající zejména pokud je mnoho vývojářů a mnoho verzí SW

* představte si tým vyvíjející SW
 - úspěšný SW => tisíce požadavků na opravy a vylepšení
 - kód ve sdíleném adresáři - co když dva programátoři provádějí změnu ve

```
   stejném modulu?
 - oprava chyby v produkční verzi, měla by se promítnout zároveň ve vývojové
   verzi do které jsou ale mezitím přidávány další vlastnosti
 - vylepšení zavleklo chyby - jak se vrátit ke staré verzi?
 - jak zjistit, z čeho se která verze skládá?
 - obvykle kombinace těchto + dalších problémů
* pro úspěch projektu podstatná schopnost řídit změny tak, aby si systém mohl
  zachovat integritu v čase

Tradiční SCM proces
-------------------

* v tradičním procesu vývoje SW založeném na vodopádovém modelu je SW předán
  SCM týmu po dokončení vývoje a po otestování jednotlivých komponent
 - SCM tým přebírá odpovědnost za sestavení úplného systému a za vedení testů
 - chyby objevené při testu systému jsou předány zpět k opravě vývojovému týmu
 - tento přístup ovlivnil tvorbu standardů; např. IEEE Std. 828 nevysloveně
   předpokládá vodopádový model, tj. obtížně se přizpůsobují např.
   inkrementálnímu vývoji
 - proto také SCM patří k nejhůře zpracovaným tématům v literatuře o SW
   inženýrství
 - napřed popíšu tradiční SCM proces (tak jak ho popisují standardy), pak
   zmíním přizpůsobení SCM pro inkrementální model SW procesu

* tradiční SCM definuje 4 procedury, které musejí být pro SW projekt definovány
  pokud má být definován dobrý SCM proces
 - identifikace konfigurace
 - řízení konfigurace
 - vytváření záznamů o stavu konfigurace
 - autentizace konfigurace

Identifikace konfigurace
........................

* informace, které jsou výstupem jednotlivých fází SW procesu můžeme rozdělit
  do tří velkých kategorií: (1) programy (jak zdrojové texty tak
  spustitelné), (2) dokumentace programů, (3) data
 - na počátku máme specifikaci systému, z ní vznikne DSP, později design atd.
 - informace vytvořená v důsledku SW procesu a reprezentující určitou podobu
   daného SW systému se nazývá konfigurace SW

* konfigurace sestává z tzv. ”konfigurovatelných položek” (configurable item,
  CI), které jsou atomické z hlediska změn a označování verzí
 - CI bude např. DSP, ERA model, jeden .java soubor, jedna .dll knihovna,
   množina testovacích případů apod.
 - mezi CI existují závislosti (kompozice, generování, master-dependent, ...)
 - každá CI je jednoznačně identifikovatelná (typ CI, identifikátor projektu,
   identifikátor změny nebo verze)

* konzistentní konfigurace = SW konfigurace, jejíž prvky jsou navzájem
  bezrozporné
 - obsahuje např. zdrojové texty, makefiles, konfigurační soubory,
   dokumentace, testů atd. v příslušných verzích
 - bezrozporná = např. zdrojové soubory lze přeložit, knihovny přilinkovat

* baseline = konzistentní konfigurace tvořící stabilní základ pro produkční
  verzi nebo další vývoj (startovací bod pro řízenou evoluci)
 - příklad: milník beta verze aplikace stabilní: vytvořená, otestovaná, a
   schválená managementem
 - pro baseline předpokládáme následující vlastnosti:
  . dokumentovaná funkčnost, tj. vlastnosti SW pro každou baseline jsou dobře
    známé
  . známá kvalita: např. známé chyby budou dokumentovány, SW prošel
    testováním před tím, než je definován jako baseline
  . baseline je nezměnitelná a znovu vytvořitelná: po definici nemůže být
    baseline změněna, všechny CI tvořící baseline můžeme kdykoli znovu
    vytvořit
 - změny prvků baseline jen podle schváleného postupu
 - při problémech návrat k baseline
```

```
 - každá nová baseline je předchozí baseline + souhrn schválených změn CI

* proces identifikace konfigurace definuje baseline, z jakých CI se skládá

* další užitečné pojmy:
 - delta = množina změn CI mezi dvěma po sobě následujícími verzemi
  . v některých systémech jednoznačně identifikovatelná
 - repository (úložiště, databáze projektu) = centrální místo, kde jsou
   uloženy všechny CI projektu
  . řízený přístup (udržení konzistence)
 - workspace (pracovní prostor) = soukromý datový prostor, v němž je možno
   provádět změny prvků konfigurace, aniž by byla ovlivněna jejich podoba v
   repository
  . akce ”zkopírování CI z repository” a ”uložení CI do repository”

Řízení konfigurace
..................

* problém ve všech fázích životního cyklu:
 - jak zvládat množství požadavků na úpravy produktu (opravy, vylepšení)?
 - jak poznat kdy už jsou vyřešeny?

* nutný striktní postup akcí: klasifikace a vyhodnocení navrhovaných změn,
  jejich schválení nebo neschválení, koordinace schválených změn,
  implementace změn na příslušnou baseline, dokumentace a ověření

* požadavek obsahuje: název a verzi produktu / subsystému, kterého se týká;
  popis chyby či požadované změny (co nejpřesnější) + indikaci priority;
  pro chybu: jak vznikla, jak je možné ji znovu reprodukovat;
  informace o použitém software (konfigurace, OS, knihovny)
 - požadavek prochází stavy: nový -> převzatý -> přiřazený ->
   vyřešený/zrušený/duplicitní -> uzavřený

* postup zpracování požadavku
 - přijetí požadavku
  . přidělení ID
  . nastavení závažnosti, priority (kritická chyba - problém - vada na kráse
    - vylepšení)
 - v tradičním procesu schválení, neschválení, odložení změny řídí ”komise
   pro řízení konfigurace” (angl. configuration control board, CCB)
  . chyba -> nutno ověřit že chyba je reálná
 - zpracování požadavku
  . pověřený člověk (dle zodpovědnosti za části systému)
  . lokalizace změn v produktech procesu
  . oprava v lokálním workspace, testování a validace
  . schválení, nová baseline

* SW podpora - systémy pro správu změn (bug tracking systems)
 - evidence a archivace požadavků, sledování stavu požadavku, případně
   statistiky
 - často emailové, webové
 - např. Gnats + Gnatsweb, Bugzilla, JitterBug apod.

Vytváření záznamů o stavu konfigurace
.....................................

* zajištění sledovatelnosti změn SW
* zaznamenávání informací o každé verzi SW a o změnách oproti předchozí
  baseline, které k této verzi vedly
* záznam pomůže zodpovědět otázky jako
 - ”Byla chyba XYZ opravena?”
 - ”Kdo je zodpovědný za tuto modifikaci?”
 - ”Čím přesně se tato verze liší od baseline?”
* konkrétní nástroje uvedeme později

Autentizace konfigurace
.......................

* proces který zajišťuje
```

- aby v nové baseline byly zahrnuty všechny plánované a schválené změny
- aby součástí dodaného systému byly všechny požadované programy, dokumentace a data

SCM pro inkrementální vývoj

* příklad procesu:
 - celý systém se sestavuje často (např. denně)
 - organizace určí čas, do kdy musí vývojáři doručit své komponenty (např. 14h)
 - komponenty mohou být neúplné, ale musejí poskytovat základní funkčnost, která může být otestována

* z komponent se sestaví nová verze systému
 - systém je předán testovacímu týmu, který provede předdefinované testy systému
 - vývojáři zatím dále pracují na svých komponentách, přidávají funkčnost a opravují chyby objevené v předchozích testech
 - testovací tým zdokumentuje objevené chyby, předá je vývojářům - vývojáři chybu opraví v další verzi komponenty

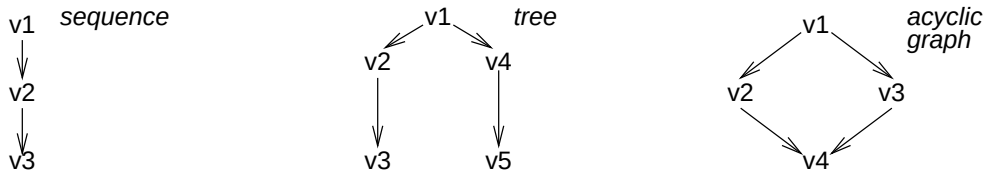
* hlavní výhodou denního sestavování je brzké nalezení problémů vzniklých interakcí komponent
* vývojáři pociťují tlak aby jejich komponenty nezpůsobily havárii systému - důsledkem je lepší testování jednotek

* SCM proces musí někdo řídit, musí ustanovit podrobné procedury, musí zajistit aby všechny změny proběhly správně
* příklad velkého projektu - jádro OS Linux:
 - většinu skutečného vývoje jiní vývojáři, ale jaké vlastnosti bude obsahovat jádro určuje jeden člověk - Linus Torvalds
 - všichni vývojáři mu posílají změny které mají být začleněny do jádra
 - hraje roli SCM procesu:
  . řízení konfigurace (začleňování/nezačleňování změn ostatních vývojářů)
  . vytváření záznamů o stavu konfigurace (ChangeLog)
  . autentizace konfigurace (zaručuje že jádro má všechny části)

Verzování

* účel: udržení přehledu o podobách CI
 - verze popisuje stav CI, nebo postup jeho změn
 - extenzionální verzování: každá verze má jednoznačné ID
   např. 1.5.1 = základní verze pro DOS, 1.5.2 pro UNIX,
   1.6.1 oprava pro DOS, 2.1.1 = nový release pro DOS
  . přístup v často používaných nástrojích (rcs/cvs, SourceSafe)
  . jednoduchá implementace
  . nepoužitelné při větším počtu verzí
 - intenzionální verzování: verze je popsána souborem atributů
  . např. OS=DOS and UmiPostscript=YES
  . nutné pro větší prostory verzí
  . potřeba vhodných nástrojů (Adele, částečně cpp)

* prostor verzí je často reprezentován grafem
 - uzly = verze, hrany = vazby mezi verzemi, nejčastěji relace následnictví
 - větvení (branch) - často nahrazuje varianty (rcs/cvs)
 - operace vytvoření větve (branch-off, split) a spojení (merge)



* nástroje pro verzování
 - ruční verzování = dohody a konvence o značení verzí v názvech (dokument,

```
   soubor, adresář), baseline pomocí zálohy všech souborů v daném čase
 - základní - správa verzí souborů
  . obvykle extenzionální verzování modulů
  . ukládání všech verzí v zapouzdřené úsporné formě
  . příklady nástrojů: rcs, cvs
 - pokročilé - integrované do CASE
  . obvykle kombinace extenzionálního a intenzionálního verzování
  . automatická podpora pro ci/co prvků z repository do nástrojů
  . příklad nástroje: ClearCase, Adele

rcs: Revision Control System
............................

* správa verzí pro textové soubory; UNIX, Windows, DOS
 - extenzionální stavové verzování komponent
 - části systému - utility spouštěné z příkazového řádku:
  . ci, co, rcs, rlog, rcsdiff, rcsmerge
 - ukládá (do foo.c,v souboru)
  . historii všech změn v textu souboru
  . informace o autorovi a času změn
  . textový popis změny zadaný uživatelem
  . další informace (stav prvku, symbolická jména)
 - používá diff(1) pro úsporu místa
  . poslední revize uložena celá
  . předchozí pomocí delta vygenerované programem diff(1)
 - funkce
  . zamykání souborů, poskytování R/O kopií
  . symbolická jména revizí, návrat k předchozím verzím
  . možnost větvení
  . informace o souboru a verzi lze včlenit do textu pomocí klíčových slov
    $Author$, $Date$, $Revision$, $State$, $Log$ (popis poslední změny
    zadaný uživatelem), $Id$ (kombinace filename revision datum author state)
  . typické použití v C: static char rcsid[] = "$Id$"
    při "co" expanduje na "$Id: soubor.c,v 1.1 2003/05/16 03:17:16 luki Exp $"

CVS (Concurrent Versioning System)
..................................

* nadstavba nad rcs => umí vše co umí rcs (zejm. klíčová slova)
* optimistický přístup ke kontrole paralelního přístupu
 - místo zamkni-modifikuj-odemkni (RCS) pracuje systémem zkopíruj-modifikuj-sluč
* práce s celými konfiguracemi (projekty) najednou
* sdílené repository + soukromé pracovní prostory
* repository lokální nebo vzdálená (rsh, p-server)
* možnost definovat obsah a strukturu konfigurace
* zjišťování stavu prvků, rozdílů oproti repository
* příkazová řádka, grafické nadstavby (UNIX, Windows, web)

* rcs i cvs jsou volně šířené
* množství informací on-line, viz např. http://www.loria.fr/~molli/cvs-index.html
* existují další podobné nástroje, např. PRCS, Subversion apod.

cpp: Realizace variant
......................

* cpp = C preprocessor, umožňuje intenzionální stavové verzování
 - např. chceme variantu foo.c pro případ OS=DOS and UmiPostscript=YES
 - definice atributů pro popis variant
 - hlavičkový soubor (centrální místo def. varianty celého systému)
   parametry příkazové řádky gcc -DOS_DOS (např. v Makefile)


Automatizace překladu a sestavení projektu: make
------------------------------------------------

* program "make" pochází ze systémů UNIX, původně souvislost s jazykem C
* účelem automatizace překladu a sestavení projektu, minimalizace času
  spotřebovaného vytvářením aktuálních verzí objektových a dalších "strojově
  vytvářených" souborů
```

* spustitelný program a objektové soubory se typicky vytvářejí ze zdrojových textů
* popis pravidel překladu umístíme do souboru ”Makefile” (případně ”makefile”)
* překlad spustíme příkazem ”make”

Příklad (projekt v jazyce C v systému UNIX)

 * program p bude sestaven ze dvou objektových souborů a.o a b.o
 * ty se vytvářejí překladem z odpovídajících zdrojových textů a.c a b.c, oba používají společný hlavičkový soubor inc.h:

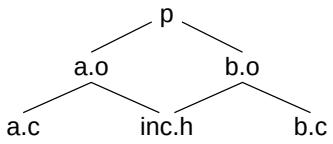
 * pro překlad projektu vytvoříme soubor Makefile, obsahující tři pravidla:

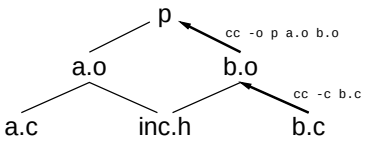
```
p: a.o b.o
        cc -o p a.o b.o

a.o: a.c inc.h
        cc -c a.c

b.o: b.c inc.h
        cc -c b.c
```

 * poslední pravidlo znamená:
  - soubor ”b.o” závisí na souborech ”b.c” a ”inc.h”
   . pokud bude soubor ”b.c” nebo soubor ”inc.h” mladší než ”b.o”, je třeba ”b.o” znovu vytvořit pomocí příkazu ”cc -c b.c”
   . b.c bude mladší např. pokud v něm provedu změnu textovým editorem
  - ostatní pravidla obdobná, tj. pokud dojde ke změně, provede se minimální počet příkazů, která zajistí, aby výsledek byl aktuální





[]

* pravidla mají tvar

```
      cíl: prerekvizity
              příkaz1
              příkaz2
```

 - cíl (cílový soubor, co se má vytvořit)
 - volitelně prerekvizity (soubory, ze kterých se cíl vytváří)
 - volitelně příkazy, které se spustí, pokud je některá z prerekvizit mladší než cíl (cíl je ”zastaralý”, měl by se vytvořit z prerekvizit; pro zjištění stáří se používá čas modifikace souboru)
  . pozor, v UNIXovém ”make” musí být příkazy uvozeny tabulátorem

* provedení souboru Makefile - spustíme příkaz ”make”
 - make najde první pravidlo v souboru Makefile
 - před provedením pravidla rekurzivně zajistí, aby jeho prerekvizity nebyly zastaralé
  . každou prerekvizitu bude považovat za cíl, najde příslušné pravidlo
  . pokud je cíl zastaralý, vytvoří ho znovu pomocí příkazů pravidla
 - chyba (nenulová návratová hodnota příkazu) způsobí ukončení programu make (lze potlačit uvedením ”-”, tj. ignoruje případnou chybu)

* falešné (phony) cíle
 - cíl je ve skutečnosti ”návěští podprogramu”, nikoli vytvářený soubor
 - pravidlo nemá prerekvizity
 - používá se např. pro automatizaci úklidových akcí, instalaci, provedení testů, vytváření distribučních archivů apod. (phony cíle clean, install, test apod.)

```
       clean:
               -rm *.o core
```

* proměnné (v terminologii programu make nazývané makra)
 - definice: jméno=řetězec
 - použití: $(jméno)
 - příklad:

```
      CC=gcc  # kterým překladačem jazyka C budeme překládat

      p: a.c
              $(CC) -o p a.c
```

* další vlastnosti: vestavěná pravidla, inferenční pravidla

* soubory Makefile najdete ve většině volně šířených programů (jádro OS Linux apod.)
* protože ”make” je velmi používáno, mnoho firem apod. má vlastní rozšíření (podpora paralelního běhu, ”include” a ”ifdef” podobně jako v C, apod.)
* gcc -M umí vytvořit závislosti pro objektové soubory, jikes pro .class

* existují další nástroje se obdobným účelem, např. Ant pro automatizaci překladu projektů v jazyce Java

Etické a právní aspekty tvorby SW

* stejně jako v jiných oborech i v SW inženýrství existují určitá etická pravidla, jejichž nedodržování je považováno za neslušné a neprofesionální
* profesionální organizace jako ACM a IEEE definovaly ”etické zásady SW inženýra” (CS Code of Ethics), viz http://www.acm.org/serving/se/code.htm

* některé základní zásady:
 - při vytváření SW máte možnost být někomu prospěšní nebo mu způsobit škodu (nebo dát možnost jiným aby byli prospěšní nebo ublížili)
  . například pokud jste zodpovědní za vývoj systému kritického pro bezpečnost lidí a čas vás tlačí - bylo by neetické prohlásit systém za otestovaný, pokud není
  . pokud je pravděpodobná účast na vojenských, nukleárních nebo jiných projektech, na které existují různé pohledy z etického hlediska, je třeba to předem mezi zaměstnavatelem a zaměstnancem vyjasnit
 - důvěrnost - měli byste respektovat důvěrnost informací o klientech nebo zaměstnavateli
 - způsobilost - měli byste si být vědomi své úrovně a nepřijímat vědomě práci, která je nad vaše schopnosti
 - dodržovat autorská práva, patenty apod. - porušováním můžete uvést do potíží nejen sebe
 - nezneužívat cizí počítače např. pro provozování programů, se kterými by vlastník nesouhlasil

Autorský zákon

* pokud se živíte SW, je životně nutné znát autorský zákon (121/2000 Sb. ”Zákon o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů”) viz http://www.nkp.cz/o_knihovnach/00-121.htm
 - z § 2 vyplývá, že počítačový program (”je-li původní v tom smyslu, že je autorovým vlastním duševním výtvorem”) i jeho jednotlivé vývojové fáze a části jsou předmětem ochrany podle AZ; podle § 65 je chráněn jako dílo literární
 - § 5, § 11 a § 26: autorem je fyzická osoba, která dílo vytvořila, autorství nelze převést nebo se ho vzdát
* s AZ souvisí § 152 trestního zákona: ”Kdo neoprávněně zasáhne do zákonem chráněných práv k autorskému dílu ... bude potrestán odnětím svobody až na dvě léta nebo peněžitým trestem nebo propadnutím věci.”

* licence
 - § 12 a § 46: autor má právo své dílo užít a udělit jiné osobě smlouvou

licenci k jednotlivým způsobům nebo ke všem způsobům užití (užití podle AZ = rozmnožování, rozšiřování atd.; rozmnožování je podle § 66 i ”vytvoření rozmnoženiny (nezbytné) k zavedení ... programu do paměti počítače”)
. § 49: licence může být výhradní nebo nevýhradní (výhradní = autor nesmí poskytnout licenci třetí osobě)
. § 49: povinnou náležitostí licence je výše odměny nebo způsob jejího určení
. § 50: není-li v licenci řečeno jinak, platí pouze na území České republiky
. § 50: není-li určeno jinak, platí max. jeden rok (!!!)
- § 58: zaměstnavatel vykonává svým jménem a na svůj účet autorova majetková práva k dílu, které autor vytvořil ke splnění svých povinností k zaměstnavateli (není-li sjednáno jinak)
. není-li sjednáno jinak, zaměstnavatel může dílo zveřejnit pod svým jménem, upravovat atd.
. počítačové programy se považují za zaměstnanecká díla i tehdy, byla-li vytvořena na objednávku

* licence platná v právním řádu jiné země nemusí být platnou licencí podle českého AZ a naopak (zejména pokud v licenci chybí některé ze zákona povinné ustanovení)

* podle českého AZ vzniká právo autorské k dílu v okamžiku, ”kdy je dílo vyjádřeno v jakékoli objektivně vnímatelné podobě”
- naproti tomu v jiných jurisdikcích je požadováno uvést informaci o copyrightu ve tvaru: Copyright <rok zveřejnění> <držitel autorských práv>

* příklad licence platné v ČR:

Copyright 2004 Západočeská univerzita v Plzni

Západočeská univerzita v Plzni tímto poskytuje nabyvateli rozmnoženiny tohoto počítačového programu a jeho dokumentace (dále jen ”Software”) bezúplatně celosvětovou a časově neomezenou nevýhradní licenci ke všem způsobům užití Software včetně zejména práva Software rozmnožovat a rozšiřovat, s právem upravovat Software a měnit jeho název, spojovat Software s jinými díly a zařazovat Software do děl souborných.

❇