Poznámky k textu přednášek

Pokud bude v zápisu přednášek něco nesrozumitelné nebo pokud najdete překlepy apod., prosím dejte mi to vědět!

Věci, které jsou uvedeny s poznámkou typu ”pro zajímavost” nebudu požadovat u zkoušky, ale bylo by si je alespoň přečíst.

Nejčastější zkratky, použité v tomto textu:

* DSP = dokument specifikace požadavků
* HW = hardware
* SW = software
* SWE (SW Engineering) = softwarové inženýrství

Předběžný plán přednášek:

1. Úvod. Co je SW inženýrství. Proces vývoje SW.
2. Specifikace požadavků.
3. Získávání požadavků.
4. Notace objektově orientované analýzy.
5. Objektově orientovaná analýza a design.
6. Strukturovaná analýza.
7. Moderní strukturovaná analýza. Terminologie objektově orientované analýzy.
8. Strukturovaný design. Architektonický návrh.
9. Architektonické styly. Programátorský styl a dokumentace kódu.
10. Programátorský styl a dokumentace kódu. Nástroje.
11. Implementace. Verifikace a validace. Ladění a testování jednotek.
12. Integrace a testování systému.
13. Konfigurační management. Údržba SW systémů. Metriky. Prototypování. Práce v týmech. Etické a právní aspekty tvorby SW.

(Někdy bude část tématu dokončena v následující přednášce.)

Jak a proč vzniklo SW inženýrství

* DOTAZ: kolik funkčních řádků programu jste schopni napsat za den?

* první počítače programovány jednotlivci nebo malými týmy
* často VT výpočty, jazyky FORTRAN a assembler

* příchod počítačů III. generace (1965-1980) - integrované obvody (oproti počítačům sestaveným z jednotlivých tranzistorů apod.)
* o několik řádů výkonnější, větší paměť apod.
* možnosti nových aplikací (banky, pojišťovny, letecké společnosti...)
* výsledné aplikace o několik řádů rozsáhlejší než předchozí SW systémy

* výsledek - důležité systémy často léta zpoždění, cena několikanásobek původních předpokladů, chybovost atd. - termín ”softwarová krize”
* ukázalo se, že způsoby vývoje pro malé SW projekty se nedají použít pro velké projekty, zapotřebí nové techniky a metody
* termín ”softwarové inženýrství” navržen v 1968 na konferenci NATO o ”softwarové krizi”

* nejznámější případ systému OS/360 od IBM
  - jeden z prvních velmi rozsáhlých projektů operačního systému
  - na začátku 200, maximum přes 1000 lidí
  - více o něm na ZOS

* Brooks na základě zkušeností napsal knihu ”The Mythical Man Month” (Brooks 1975)
* téma - proč je obtížné vytvářet velké SW systémy

* Brooks např. tvrdí, že programátoři jsou schopni napsat jenom cca 1000 řádků odladěného kódu za rok (tj. v průměru cca 5.5 řádku za den)

```
* Brooks vs. odpovědi na DOTAZ - jak je to možné?

* velké projekty (= stovky programátorů) jsou úplně jiné než malé
  projekty - zkušenosti z malých projektů se nedají na velké přenést

* velké projekty - dlouhou dobu spotřebuje:
 - plánování jak rozdělit projekt do modulů
 - specifikace činnosti a rozhraní modulů
 - získání představy o interakci modulů
 - to všechno před tím, než začne vlastní psaní programu (”kódování”)

* následuje kódování a odladění jednotlivých modulů
* integrace (sestavení) modulů do výsledného systému
* výsledný systém je třeba otestovat (i když jednotlivé moduly
  odladěny, po sestavení částí obvykle nefunguje napoprvé)

* tato posloupnost se nazývá ”vodopádový model” vývoje SW:
1. Plánování
2. Kódování
3. Test modulů
4. Test systému
5. Nasazení
(Ještě se k tomu vrátíme; teď zmiňuji pouze pro informaci.)

* Brooks odhadl, že
 - 1/3 celkové práce je plánování
 - 1/6 kódování
 - 1/4 testování modulů
 - 1/4 testování systému

* jinými slovy
 - kódování je ta nejsnazší část
 - obtížné je:
  . rozdělit projekt do modulů
  . zajistit, aby modul A správně komunikoval s modulem B

* tj. pokud 1 programátor píše malý program, zůstává mu ta nejjednodušší část
  (jeho efektivita je také vyšší, podle současných měření cca 20 řádků/den)
* většina úloh, které jste zatím řešili, byly malé úlohy
* cílem tohoto předmětu je získat základní představu i o ostatních
  částech procesu vývoje SW

Poznámka (co se míní pojmy software, SW systém a SW produkt)

 * většina lidí si pod pojmem SW představuje pouze programy, pro praxi
   příliš omezující pohled
 * SW = programy + dokumentace + konfigurační data
 * SW systém sestává obvykle z několika programů, konfiguračních
   souborů, systémové dokumentace (popisuje strukturu systému) a
   uživatelské dokumentace (vysvětluje jak systém používat)
 * SW produkt = SW které se dá prodat zákazníkovi

 * existují 2 základní typy SW produktů:
  - generické produkty
   . vyvíjen např. na základě analýzy potřeb trhu
   . samostatné systémy prodávané na otevřeném trhu každému, kdo je
     schopen si je koupit (shrink-wrapped SW)
   . např. operační systémy, textové procesory, kreslící programy,
     databáze, překladače programovacích jazyků atd.
  - produkty vyvíjené na zakázku (customised products)
   . SW vyvíjený pro konkrétního zákazníka na základě jeho požadavků
   . např. informační systém pro konkrétní firmu, řídící systémy pro
     elektronická zařízení apod.
  - nejpodstatnější rozdíl - kdo určuje specifikaci

[]
```

Co je SW inženýrství?
....

* pojem software - viz poznámka výše

* pojem inženýrství (engineering) - slovníková definice je: praktická
  aplikace teorie, metod a nástrojů při návrhu strojů, mostů apod.
 - praktické řešení je ale třeba najít i když odpovídající teorie (ještě)
   neexistuje
 - problémy je třeba řešit v rámci daných finančních a organizačních omezení

* SW inženýrství je aplikace inženýrských metod na software
 - zabývá všemi aspekty tvorby SW: specifikací, vývojem, testováním, údržbou,
   managementem, atd. především rozsáhlých SW systémů, vývojem teorie, metod
   a nástrojů pro vývoj SW

Definice z IEEE Standard Computer Dictionary (1990):

 Aplikace systematického, disciplinovaného, měřitelného přístupu na vývoj a
 údržbu software; jinými slovy, aplikace inženýrských principů na software.

* rozdíly mezi SW inženýrstvím (SWE) a informatikou (computer science, CS):
 - CS se zabývá algoritmy, způsobem práce atd. počítačů a SW systémů -
   existuje exaktní popis
 - SWE řeší praktické problémy tvorby SW
  . příliš složité, často nutné používat ad hoc metody
  . na rozdíl od CS se v SWE většinou nedozvíte porovnání jednotlivých metod
    apod. - porovnání je obtížné a drahé (dokonce čím dál dražší)
  . často poskytuje obecné koncepce, je na uživateli aby je naplnil
    jednotlivinami

Cíle předmětu ZSWI:

* hlavní cíl: zkusit si práci v týmu, naučit a vyzkoušet základní postupy,
  které by měl znát každý programátor (základy analýzy, návrhu, testování
  modulů a systému; na ZSWI navazuje ASWI, kde důkladněji)
* na přednáškách: vysvětlit význam SW inženýrství, dozvědět se o dalších
  oblastech, které SW inženýrství pokrývá

Potenciální problém:

* ”programátorský prvňáček si obvykle uvedené problémy neuvědomuje a zmíněné
  pravdy může považovat za zbytečnou teorii” [Bořík 1999]
* proto týmová semestrální práce, na které si budete moci některé problémy
  alespoň uvědomit

Ale zpátky k Brooksově knížce:

Proč Mythical Man Month
....

* dodneška můžeme najít různá vyjádření náročnosti vývoje SW v
  ”člověkoměsících” (resp. ”člověkorocích”) = počet lidí*čas
* titul Brooksovy knihy vychází z tvrzení, že čas a počet lidí nejsou
  zaměnitelné

* pokud projekt trvá 15 lidem 2 roky, není možné aby 15*24=360 lidem
  trval měsíc (a asi ani to, aby 60 lidem trval 6 měsíců)

* je to ze 3 důvodů:
1) práce není plně paralelizovatelná
  - dokud není dokončeno plánování a dokud není určeno rozdělení systému do
    modulů a definováno rozhraní modulů, nemůžeme začít s kódováním
  - např. pro dvouletý projekt může plánování trvat 8 měsíců
2) abychom plně využili velký počet programátorů, musíme systém rozdělit na
   velký počet částí (aby každý programátor měl práci)
  - každý podsystém ale může potenciálně komunikovat se všemi ostatními
    => počet uvažovaných interakcí mezi podsystémy roste s druhou mocninou
    počtu podsystémů

3) ladění a testování systému jsou obtížně paralelizovatelné
  - 10 lidí nenajde chybu 10x rychleji než 1
  - skutečně spotřebovaný čas závisí na počtu chyb a obtížnosti jejich hledání

Brooks shrnul svou zkušenost do tzv. Brooksova zákona:

”Přidáním lidí ke zpožděnému projektu projekt ještě více zpozdíme.”
(Adding manpower to a late software project makes it later.)

* proč?
 - přidaní lidé se musejí s projektem seznámit, což zabere čas i stávajícím lidem (místo aby programovali, učí nové členy týmu)
 - práce musí být přerozdělena tak, aby to odpovídalo většímu počtu členů týmu (přerozdělení práce zabere čas, také tím přijdeme o část již udělané práce)

Struktura týmu
....

* malé skupiny programátorů (do 10 lidí) jsou obvykle organizovány celkem neformálně
 - vedoucí skupiny se spoluúčastní vývoje
 - ve skupině se může najít ”technický vedoucí” skupiny, který určuje technické směřování
 - práce je diskutována skupinou jako celkem (demokratické konsensuální rozhodování), práce je rozdělována podle schopností a zkušeností

* velké projekty - velké týmy (OS/360 - ve špičce přes 1000 lidí)
* jak tým strukturovat?

* hodně důležitá kvalita lidí
* je známo (Sackman et al 1968), že špičkoví programátoři jsou 10x výkonnější než špatní programátoři (ve skutečnosti měření probíhalo ve skupině zkušených programátorů; Sommerville uvádí rozdíl až 25x)
* problém - pokud potřebuji 200 programátorů, těžko zaměstnám 200 špičkových - jsem nucen zaměstnat lidi různých kvalit

* ve velkých projektech je důležitá tzv. ”architekturální koherence”
* nejlépe aby jeden člověk měl pod kontrolou návrh (design) systému
* jinak může vzniknou slepenice, která se nikomu nebude líbit

* kombinace myšlenky ”někteří programátoři jsou mnohem lepší než jiní” a ”potřeba architekturální koherence” (Mills asi 1970)
 - organizace týmu typu ”chirurgický tým” (surgical team)
 - výkonný programátor = šéf týmu, měl by mít možnost 100% pracovat na návrhu a kódování
 - ostatní členové týmu mu pomáhají, aby se nemusel zabývat rutinními věcmi
 - pro 10 členný tým např. tyto role:
  . šéfprogramátor - navrhuje architekturu a píše kód
  . kopilot - pomáhá šéfprogramátorovi a slouží coby hlásná trouba
  . administrátor - management lidí, peněz, prostoru, zařízení atd.
  . redaktor - rediguje dokumentaci, kterou musí psát šéfprogramátor
  . sekretářky - administrátor i redaktor je potřebují
  . archivář (program clerk) - archivuje kód i dokumentaci
  . toolsmith - vytváří jakékoli nástroje, které šéfprogramátor potřebuje
  . tester - testuje kód šéfprogramátora
  . jazykový právník (language lawyer) - externista, který může šéfprogramátorovi poradit s programovacím jazykem (moderní programovací jazyky jsou jednodušší než PL/1 - v současnosti asi nebude zapotřebí jazykový právník, ale znalec knihoven by se mohl uplatnit)
 - dnes už je mnoho uvedených fcí automatizovatelných - je možný menší tým, ale základní myšlenka stále platí

* co když máme větší projekt?
 - musíme organizovat jako hierarchii
 - na nejnižší úrovni mnoho malých týmů, každý veden šéfprogramátorem
 - skupiny týmů musejí být koordinovány managerem

- ze zkušenosti vyplývá, že každý člověk kterého řídíte vás stojí 10% času => manager na plný úvazek pro každou skupinu 10 týmů
- tito manažeři musejí být řízeni... až 3 úrovně

Poznámka k zápočtové úloze:

Ustanovení efektivního chirurgického týmu vyžaduje talentovaného programátora (kterých je málo) a čas (který nemáte) a i poté má určitá rizika (týkají se zejména rolí ostatních členů týmu). Proto prosím pracujte jako neformální skupina.

[]

* Brooks popisuje pozorování, že výše uvedeným stromem neprocházejí dobře špatné zprávy (tzv. ”bad news diode”)
  - žádný šéfprogramátor ani manager nebude chtít říci nadřízenému, že má zpoždění 4 měsíce a deadline není možné stihnout (nositelé špatných zpráv obvykle nejsou vítáni)
  - výsledek - vrcholový management se obtížně dozvídá o skutečném stavu projektu

Role zkušenosti
....

* pro projekt je důležité mít zkušené návrháře
* Brooks ukazuje, že většina chyb není v kódu, ale v návrhu
  - programátoři správně naprogramují to, co se jim řeklo, ale co se jim řeklo (někdy ani neřeklo) řeklo je chybně
* proto navrhuje místo klasického vodopádového modelu:
  - napsat hlavní program, který bude volat top-level procedury (které jsou na začátku prázdné)
  - postupně přidávat moduly do celého systému
  - výsledek - testování integrace systému se děje kontinuálně, chyby v návrhu se projeví dříve

* nějaká (malá) zkušenost týmu je nebezpečná - viz tzv. efekt druhého systému (second systems effect)
  - první produkt týmu obvykle bývá minimální, protože návrháři mají obavy, aby produkt vůbec vytvořili
  - produkt pracuje, na členy týmu to udělá dojem
  - při vytváření druhého systému zahrnou vše, co při vytváření prvního produktu vynechali
  => výsledek - druhý systém je nafouklý, nevýkonný atd.
  - při vytváření třetího systému už si opět dávají pozor

Příklad (v OS):

* první systém se sdílením času CTSS - minimální funkčnost
* následník MULTICS - příliš ambiciózní, verze kterou se po letech podařilo dokončit se příliš neujala
* třetí systém UNIX - pečlivý výběr vlastností, úspěšnější.

”No Silver Bullet”
....

* Brooks napsal také důležitý článek nazvaný ”No Silver Bullet” (1986) (legendy tvrdí, že vlkodlaka lze zastřelit pouze stříbrnou kulkou)
* tvrdí, že žádná technologie nebo technika managementu nezpůsobí do 10 let řádové zlepšení produktivity vývoje SW - a měl pravdu

* různé věci, které byly považovány za ”silver bullet”:
  - lepší vysokoúrovňové jazyky
  - objektově orientované programování
  - umělá inteligence a expertní systémy
  - grafické programování
  - verifikace programů
  - prostředí pro tvorbu programů
* spíše se zdá, že budou postupná vylepšení

(Konec povídání o Brooksově knížce.)

Softwarový proces

* SW proces = množina aktivit a (mezi)výsledků (artefaktů), jejichž výsledkem je SW produkt

* existují různé SW procesy
 - velká různorodost vytvářeného SW => pro různé typy systémů vhodné rozdílné procesy
 - do úvahy je třeba vzít různé schopnosti (znalosti, dovednosti) zúčastněných lidí
  . například pro Boehmův spirálový model potřebujeme lidi, kteří mají dostatek zkušeností pro provádění analýzy rizik

* nicméně ve všech SW procesech se objevují 4 základní aktivity:
 - specifikace SW = určení požadované funkčnosti a definice omezení
 - vývoj SW = tvorba SW splňujícího specifikaci
 - validace SW = ověření že SW dělá co požaduje zákazník
 - evoluce SW = přizpůsobení SW měnícím se požadavkům zákazníka

* různé SW procesy organizují tyto 4 aktivity různým způsobem (různé časování apod.)

Modely SW procesu

* model neboli paradigma SW procesu = zobecněný popis procesu, resp. popis procesu z určitého pohledu
* nyní uvedeme nejpoužívanější obecné modely SW procesu

* vodopádový model (waterfall model)
 - výše uvedené 4 aktivity chápe jako samostatné fáze procesu
 - po splnění se fáze ukončí a přechází se na další

* evoluční vývoj
 - aktivity specifikace, vývoje a validace jsou smíšeny
 - ze specifikace je velmi rychle vyvinut prvotní systém
 - na základě požadavků zákazníka je systém dále upravován
 - pak může být systém předán nebo reimplementován strukturovanějším způsobem (pro snazší údržbu)

* formální transformace
 - na počátku je vytvořena abstraktní formální (matematická) specifikace systému
 - tato specifikace je postupně transformována do funkčního programu
 - transformace zachovávají správnost, tj. na konci víme, že program odpovídá specifikaci

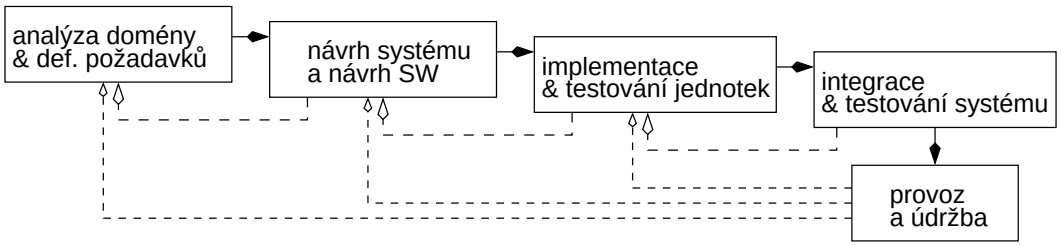
* sestavení systému ze znovupoužitelných (reusable) komponent
 - předpokládá, že části systému již existují
 - vývoj systému se zaměřuje na integraci těchto částí (místo toho, aby se vyvíjely znovu)

* iterativní modely - inkrementální vývoj a spirálové modely
 - hybridní modely, umožňují použít různé přístupy pro různé části systému

Neexistuje nějaký ”nejsprávnější” model, různé modely jsou vhodné pro různé situace. V praxi se nejčastěji používají SW procesy založené na vodopádovém modelu a evolučním vývoji. Formální metody byly v několika projektech úspěšně použity, zatím ale nejsou příliš rozšířené - rozšíření lze čekat spíše v budoucnu pro vývoj distribuovaných systémů. Většina SW procesů zatím není explicitně orientována na vyváření systémů z existujících komponent; i v tom se dá očekávat změna. Zajímavé jsou hybridní modely, protože umožňují realizovat podsystémy podle nejvhodnějšího modelu (odpovídají potřebě tvorby velmi rozsáhlých systémů).

Vodopádový model vývoje SW

* nejstarší publikovaný model (Royce 1970)



Základní aktivity jsou:

* analýza domény, získávání (sběr) a definice požadavků
  - analýza domény = seznámení s širším (např. obchodním) kontextem systému
    . pojem ”doména” zde znamená obor, kterého se problém týká
  - získávání požadavků = konzultací s uživateli systému se zjistí cíle, požadované služby a omezení kladená na systém
  - definice požadavků = výše uvedené cíle, služby atd. jsou podrobně definovány v dokumentu specifikace požadavků (DSP), který slouží jako specifikace vytvářeného systému

* návrh systému a návrh SW (angl. system & software design)
  - návrh systému rozdělí celkové požadavky na požadavky na HW a požadavky na SW, určí celkovou architekturu systému
  - návrh SW zahrnuje identifikaci a popis základních abstrakcí a jejich vztahů (často pomocí grafické notace, např. UML)

* implementace a testování modulů
  - design je realizován jako množina modulů, tříd, případně programů
  - testování jednotek = ověření, že každý modul odpovídá specifikaci modulu

* integrace a testování systému
  - jednotlivé moduly jsou sestaveny do výsledného systému
  - úplný systém je otestován na shodu se specifikací
  - po otestování je systém předán zákazníkovi

* provoz a údržba
  - nejdelší fáze životního cyklu - systém se prakticky používá
  - údržba = oprava chyb programu a designu, které nebyly odhaleny v předchozích fázích + rozšiřování systému podle nových požadavků + zlepšování implementace

Poznámka (pojem údržba)

Pojmem ”údržba” (angl. maintenance) se v SW inženýrství neníní administrace systému, ale změny software vynucené provozem systému.

[]

Teoreticky by další fáze procesu neměla začít, dokud není předchozí fáze dokončena; v praxi:

* během designu jsou odhaleny problémy s požadavky, během kódování problémy s designem apod.
* proto SW proces není lineární sekvence - ve skutečnosti se musíme vracet (na obrázku naznačeno čerchovaně)
  - většinou se vracíme o 1 fázi
  - údržba způsobuje změny v určité fázi, dále se pokračuje podle vodopádového modelu

* výše zmíněné iterace jsou drahé a pracné (zahrnují přepracování a schvalování dokumentů)
* v praxi proto často dojde po několika iteracích ke zmražení příslušné

části vývoje (např. zmražení specifikace) a přejde se na další fázi
- problémy jsou tím "odloženy na později", čili fakticky ignorovány
- zmražení požadavků často vede k tomu, že systém nedělá co uživatel chce
- zmražení designu obvykle vede ke špatně strukturovaným systémům (problémy designu se pak při implementaci obcházejí různými triky, které ztíží další vývoj - "informační skleróza")

Problémy vodopádového modelu:

* rozdělení projektu do fází je málo flexibilní
 - je obtížné reagovat na změny požadavků ze strany zákazníka
 - proto je vodopádový model vhodný pouze pokud je problém dobře známý (= jsme schopni získat korektní specifikaci)
 - problémy s vodopádovým modelem vedly k formulaci alternativních modelů

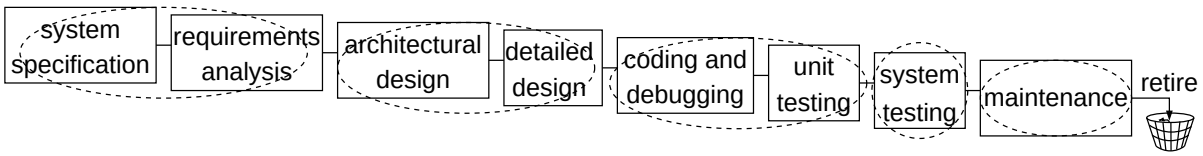
Poznámka (zápočtová úloha a přednášky podle vodopádového modelu)

Vytváření zápočtové úlohy je strukturováno podle klasického vodopádového modelu; stejně budou strukturovány i přednášky, abyste se dozvěděli co potřebujete před tím, než to po vás budu chtít. Na druhou stranu to povede k odsunutí některých témat z jejich logického místa na později (např. strukturovaný návrh bude uveden až po objektovém návrhu, i když logicky patří před něj).

[]

Poznámka (odlišnosti v literatuře)

Protože SW inženýrství je relativně mladý obor, najdeme v různé literatuře popisující totéž téma značné odlišnosti (vyjma oblastí, které jsou již standardizovány). Např. porovnejte první zmínku o vodopádovém modelu uvedenou v souvislosti s Brooksovo knihou s výše uvedeným a s následujícím obrázkem:



Rozdělení do jednotlivých fází můžeme nazvat různě, principy ovšem zůstávají tytéž.

[]

Evoluční modely vývoje SW
....

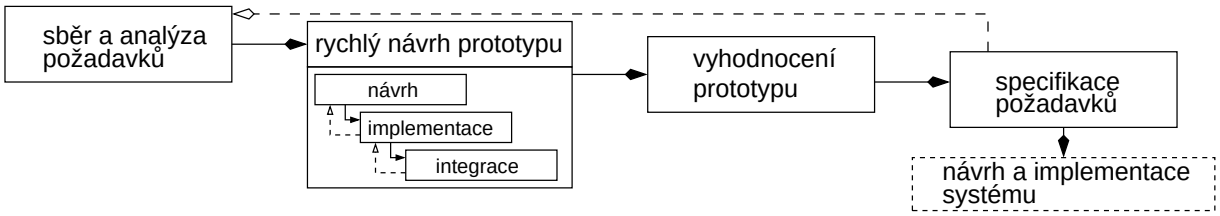
* základní myšlenka:
 - vytvořit počáteční implementaci
 - vystavit jí komentářům uživatele
 - postupně vylepšovat přes meziverze dokud nedostaneme adekvátní výsledek

* fáze specifikace, vývoje a validace neprobíhají vždy sekvenčně ale mohou probíhat i paralelně, mezi fázemi je zpětná vazba

* existují 2 základní typy modelů evolučního vývoje SW:

1. model výzkumník (exploratory development, tj. průzkumný vývoj)
  - cílem je spolupracovat se zákazníkem na zjištění jeho požadavků abychom mu nakonec dodali požadovaný systém
  - vývoj obvykle začíná dobře srozumitelnými částmi systému
  - systém se vyvíjí přidáváním nových vlastností navrhovaných zákazníkem

2. model prototyp (throw-away prototyping, tj. doslova tvorba prototypu který bude zahozen)
   - cílem je lepší pochopení zákazníkových požadavků a v důsledku vytvoření lepší definice požadavků na systém
   - tvorba prototypu se zaměřuje na ty části požadavků zákazníka, kterým moc nerozumíme



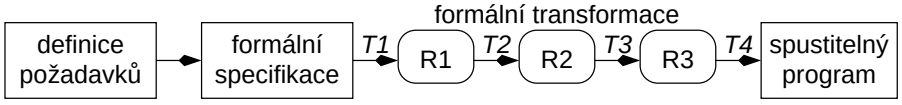
* výhody evolučního vývoje SW
  - často vede efektivněji než vodopádový model k systému splňujícímu bezprostřední požadavky zákazníka
  - specifikace může být vytvářena postupně
  - zákazník vidí že se něco děje (na rozdíl od vodopádu, kde produkt vidí až na konci)

* problémy:
  - proces není viditelný: manažeři nemohou měřit postup vývoje (na rozdíl např. od vodopádového modelu)
  - systémy vyvářené evolučně jsou často špatně strukturované
    . neustálé změny vedou k porušení struktury systému
    . vývoj dalších verzí se postupně stává obtížnější = dražší
  - prototypování může vyžadovat speciální nástroje a techniky
    . výstup může být problematický, např. nekompatibilní s jinými nástroji a technikami

* tj. vhodné buď pro malé systémy (do 100 000 řádků kódu) nebo pro středně velké systémy (do 500 000 řádků) s krátkou dobou života
* ve velkých systémech by evoluční vývoj působil zřetelné problémy, ale je možné použít smíšený přístup:
  - špatně srozumitelné části - throw-away prototyping => zpřesnění DSP
  - dobře srozumitelné části - vodopádový model
  - uživatelské rozhraní - model výzkumník

Formální modely vývoje SW

* určitým způsobem podobné vodopádovému modelu
* na počátku je vytvořena specifikace požadavků
* ta je zpřesněna do podrobné formální specifikace systému (pojmem formální se zde míní ”matematická”)
* postupné zpřesňování specifikace formálními transformacemi
* konečným výsledkem má být spustitelný program



* v každém kroku se přidávají podrobnosti, ale můžeme dokázat, že odpovídá reprezentaci systému z předchozího kroku
* pokud neuděláme chybu, pak výsledek (dokazatelně) odpovídá specifikaci
* proces designu, implementace a testování jednotek je tedy nahrazen formálními transformacemi

* dokázat korespondenci transformací je obtížné a lze při tom udělat chybu
* proto jsou v praxi mnohem oblíbenější formální metody, pro které existuje podpůrný SW - model checkers, theorem provers (”dokazovače hypotéz”)

* příklady metod formálního vývoje SW: Cleanroom (IBM cca 1987), metody založené na jazycích B a Z

* příklad části jednoduché formální specifikace v modulárním specifikačním jazyce založeném na temporálních logikách:

module *PříkladČítače*

EXTENDS *Naturals*
VARIABLES *cnt,* Čítač.
retval Návratová hodnota.

$TypeInvariant_C \triangleq cnt \in Nat$

$Init_C \triangleq cnt = 0$ Nový čítač se nastaví na nulu.

INTERFACE

cntget $\triangleq$ Zvětší čítač a vrací jeho původní hodnotu.
$\wedge\ cnt' = cnt + 1$ Zvětšíme čítač
$\wedge\ retval' = cnt$ a nastavíme návratovou hodnotu.

THEOREM $Spec_C \Rightarrow \Box TypeInvariant_C$ Typová správnost specifikace.

* využití zejména tam, kde je třeba dokázat zákazníkovi bezpečnost, spolehlivost systému apod.
* cena vývoje srovnatelná s jinými přístupy

Nevýhody:

* vyžaduje specializované znalosti - výběr transformace vyžaduje určitou zkušenost, která se liší od klasického programování
* někdy vyžaduje transformaci do speciálního programovacího jazyka (často čím nižší programovací jazyk, tím obtížnější převod)
* z uvedených důvodů se v praxi příliš nepoužívá

Vývoj orientovaný na znovupoužitelné komponenty

* ve většině SW projektů se stane, že vývojáři znají design nebo kód podobný tomu, který se požaduje
* vezmou, upraví a začlení do svého systému
* děje se nezávisle na modelu vývoje SW

* v posledních letech se objevuje nový způsob vývoje - component-based SWE
* vývoj se spoléhá na velkou množinu SW komponent a nějaký rámec pro jejich integraci
 - komponentou může být např. knihovní fce, třída, podsystém, aplikace (např. Mozilla pro prohlížení nápovědy)
 - rámec - např. příkazové jazyky (Tcl/tk), distribuované objektové architektury (CORBA, JavaBeans) apod.

* komponentově orientovaný proces vývoje postupně:
 - specifikace požadavků
 - analýza komponent
 - modifikace požadavků
 - design systému s využitím komponent
 - vývoj a integrace
 - validace systému

* zatímco první a poslední fáze obdobná jako v ostatních procesech, ostatní fáze se liší

* analýza komponent
 - podle specifikace požadavků vyhledáváme komponenty, které specifikaci splňují
 - obvykle jí ale nesplňují přesně, resp. poskytují pouze část požadovaných funkcí
 - výsledkem je informace o komponentách

* modifikace požadavků
 - analyzujeme požadavky a modifikujeme je tak, aby odrážely dostupné komponenty
 - pokud modifikace není možná, vrátíme se k analýze komponent se snahou najít alternativní řešení

* design systému s využitím komponent
 - během této fáze je navržen rámec systému nebo se využije už existující rámec
 - pokud neexistuje požadovaná komponenta, musí se navrhnout

* vývoj a integrace
 - vytvoříme komponenty, které nemáme (a nemůžeme koupit atd.)
 - komponenty jsou integrovány do systému

* tento model vývoje SW omezuje množství SW, který musíme vyvinout => snížení ceny a omezení nebezpečí, že nastanou potíže
* obvykle také rychleji získáme výsledný SW

* nevýhody:
 - kompromisy vůči požadavkům jsou nevyhnutelné => systém nakonec nemusí splňovat požadavky zákazníka
 - vývoj není zcela v našich rukách, protože nové verze komponent vyvíjí někdo jiný než my (vývoj může být ukončen, nové verze komponenty nemusejí být kompatibilní se starými verzemi) => údržba systému se může prodražit
 - rámce a knihovny komponent bývají tak rozsáhlé, že seznámení s nimi může vyžadovat několik měsíců (ve velkých organizacích bývají určeni specializovaní pracovníci)

❇