Softwarové inženýrství
(SWI)
Software Engineering
(SWE)

Cíle a význam softwarového inženýrství

Co je to SWE (několik definic)?

- Disciplína zabývající se praktickými problémy vývoje velkých softwarových systémů

- Aplikace inženýrských přístupů na software

- Definice z IEEE Standard Computer Dictionary (1990): Aplikace systematického, disciplinovaného, měřitelného přístupu na vývoj a údržbu software

- Encyklopedia Wikipedia (http://en.wikipedia.org/wiki/Category:Software_engineering):

Software engineering is the application of computer sciences, project management, and other techniques to create software applications.

Čím se zabývá SWE?

- Celkové technické řešení problému zahrnující hardwarové prostředky a softwarové vybavení
- Znalosti z oblasti sběru a specifikace požadavků, analýzy problému, návrhu SW řešení, implementace, testování a údržby SW
- Znalosti z oblasti ekonomie, plánování a řízení projektů, organizace firmy atd.
- Znalosti z oblasti psychologie, lidských zdrojů, motivace, struktury řešitelských týmů apod.
- Celkové využití znalostí a postupů uvedených výše -> efektivní tvorba výsledného produktu požadované kvality

Proč vzniklo SWI?

- Aplikace od vzniku prvních počítačů a softwarových produktů stále složitější
- Zpoždění (často i několikanásobné proti původním předpokladům) při tvorbě rozsáhlých systémů, zvýšená nákladnost projektů
- Způsoby a postupy vývoje aplikované na malé systémy nepoužitelné pro rozsáhlé projekty
- „Softwarová krize“ (1968 konference NATO)
- Brooks (1975) – na základě zkušeností kniha „The Mythical Man-Month“ – téma: proč je obtížné vytvářet velké SW systémy

Brooks a Mythical Man-Month – několik postřehů , komentářů, otázek

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

Proč?

- Příchozí lidé se nejprve musí s projektem seznámit (řada z nich se musí seznámit např. i novou technologií) – po nějaký čas neprodukují žádnou „přidanou hodnotu"
- Stávající lidé zaučují nové lidi a mají méně času na vlastní projektovou práci
- Práce musí být znovu přerozdělována – to zabere jednak čas, jednak přerozdělení znamená ztrátu jisté části práce

A jak to řešit?

- Jedinečné, vždy použitelné a vždy fungující řešení neexistuje (velmi záleží na osobě „šéfa“ – na jeho zkušenostech, profesních a lidských kvalitách), problém je možné předvídat a snažit se mu předejít, to však neznamená, že nenastane
- Plánování – řešení nedostatečného počtu lidí hned, co tuto situaci rozpoznáme (tj.nejlépe na začátku projektu)
- Motivace pracovníků, aby začali pracovat na projektu, a aby neopouštěli projekt před jeho dokončením (dnes spíše „cukr“ než „bič“)
- Dočasné přetížení pracovníků na projektu
    - záleží na délce stavu přetížení – krátkodobé je možné a bývá produktivní dlouhodobé je z hlediska firmy i zaměstnanců neefektivní, ¨
    - člověk podává nejlepší výkony na hranici mírného stresu, přílišná pohoda i velký stres výkon snižují)
    - tvořivá práce potřebuje čas (myšlenky musí uzrát), manuální práce vyžaduje zachování míry pozornosti (konec projektu bývá spíše manuální než-li tvořivý)

…

2. **Není možné zaměnit čas a počet lidí**
   Tj. pokud projekt trvá 10 lidem 1 rok, pak není možné, aby 40 lidem trval 3 měsíce)

Proč?

- Práce není plně paralelizovatelná – např. dokud není hotová analýza, návrh a rozdělení systému do modulů, není možné začít s kódováním
- S rozdělením systémů na podsystémy, které řeší jiný programátor nebo programátoři (rozdělujeme, aby každý programátor měl práci) roste i počet interakcí mezi podsystémy (každý podsystém může potenciálně komunikovat s ostatními – možné interakce rostou s druhou mocninou)
- Ladění a testování systému je obtížně paralelizovatelné (10 lidí nenajde chybu 10x rychleji než 1 člověk, skutečně spotřebovaný čas závisí na skutečném počtu chyb a obtížnosti jejich hledání , tj. přeneseně na zkušenosti, pozornosti a motivaci programátorů)

A jak to řešit:

- Opět platí body viz výše
- Programátoři nejsou jen lidé sedící před počítačem a píšící kód, potřebují mezi sebou komunikovat (ve vzrůstající míře platí pro návrháře, analytiky a šéfy všeho druhu) – vyšší nároky na emoční kvalitu lidí - dlouhodobě se nevyplatí zaměstnat na projektu výborného programátora s mizernou sociální interakcí s okolím (schopnosti na bázi rozumově postihnutelných znalostí je možné poměrně snadno měnit a kultivovat, stojí jen čas, emoční schopnosti a chování lze měnit podstatně hůře (emoční centra jsou vývojově starší) , stojí nejen čas, ale i velkou ochotu a vytrvalost pracovat na osobním růstu,
- Z dlouhodobého pohledu je firemně prospěšnější zaměstnat člověka s širším odborným rozhledem a emočně zralým chováním nežli výborného odborníka v daném odboru s problematickými rysy osobnosti, první se v dané oblasti velmi pravděpodobně zorientuje a jeho přínos v čase poroste, druhý může jisté výsledky přinášet téměř ihned, ale jeho přínos bude po krátkém počátečním vzestupu v čase klesat, své problematické chování velmi pravděpodobně nezmění)

3. Programátoři jsou schopni napsat cca 1000 řádků/rok, tj. 5,5 řádku/den

Proč?

- Protože čekají na analýzu, návrh, výsledky testů…, domlouvají se, … viz výše

A jak to řešit?

Těžko,

- často lze ještě zefektivnit organizaci práce (firemní systémy však nikdy nemohou být vyladěny na 100%, jistý chaos vždy nastává a nastávat bude, v konečné fázi opět závisí na kvalitě lidí)
- smířit se s tímto faktem

4. **Efekt druhého systému – člověk, který navrhuje celkově druhý systém, tento návrh někdy až „katastrofálně“ přežene**

Proč?

- Na prvním systému se učí a získané zkušenosti v kombinaci se znalostmi se snaží na druhý pokus dovést k „dokonalosti“

Jak to řešit?

- Šéf analytik (návrhář) má za sebou již zkušenosti alespoň ze třemi projekty
- Člověk řešící druhý systém o tomto ví, a dokáže sám sebe „krotit“, či je schopen akceptovat situaci, kdy „je krocen“

5. **Sledování pokroku ve vývoji - akumulace malých termínových skluzů nakonec vede k významnému zpoždění projektu**

Proč?

- Skluzy na sobě nejsou nezávislé, navzájem se „podporují“

Jak to řešit?

- Stanoveni tzv. milníků (kontrolních bodů) (3. generace time managementu – propracovaný systém co, kdy a jak řešit, dělat - logický, rozumově dobře přijatelný systém, přesný v oblasti výsledků – dobré řešení pro firemní rámec, nikoli však na osobní úrovni -> 4. generace time managementu)
- Motivace jednotlivce + další viz výše

6. **Integrita systému – mnohé možnosti příliš komplikovaného systému nebudou využity, protože nikdo nemá čas naučit se je používat**

Proč?

- Přirozená lenost, pohodlnost + „časová komprese“ současné doby

Jak to řešit?

- Systém bude tzv. user-friendly, tj. nebude zahrnovat všechny možné funkčnosti, ale pouze ty, které budou v souladu s celkovým designem ,funkčností a cílovou skupinou uživatelů softwaru (př. vývojové prostředí, když jste začínali s Javou, raději nejprve jednodušší , pak přechod ke složitějšímu)

7. **Pilotní systém – pokud vytváříme typově nový systém, zákazníkovi neposkytneme jakkoli přepracovanou, doplňovanou a opravovanou pilotní verzi, ale druhou verzi systému budovanou prakticky znovu od začátku**

Proč?

- Na pilotní verzi se tým naučí nové techniky a technologie, které ve výsledku vedou k téměř novému návrhu architektury sytému
- Snaha opravit pilotní systém se většinou časově ani kvalitativně nevyplácí a končívá špatně (slepovaný systém, rozladění tvůrci i zákazník)

Řešení

- Viz výše

8. **Formální dokumenty a manuály – specifikované cíle projektu, způsob jejich dosažení, analýzy, návrhy implementace, kalkulace, vyhodnocení rizik, uživatelské manuály, atd.…**

Otázky:
Kdy mají být dokumenty psány a jak mají být dokumenty rozsáhlé,

- aby v nich bylo vše podstatné,
- aby nemusely být často přepisovány,
- aby se odborní pracovníci nestali nemotivovanými úředníky na úkor tvořivé práce,
- aby je někdo četl a byly mu užitečné

9. Komunikace – všichni členové týmu spolu potřebují komunikovat

Jakým způsobem?

1. Osobní komunikace
   - Výhody –nenahraditelná forma komunikace při kontaktu se zákazníkem, analýze problémů, tvořivém návrhu řešení (také „více hlav více ví“, kolegové přijdou na o, co by vás nikdy nenapadlo, protože mají více či méně jiný pohled na svět, jiné zkušenosti, jiné myšlení – tzv.brainstorming), forma sociální interakce
   - Nevýhody - nebezpečí v dlouhých a neplodných schůzích, rušení při náročné duševní práce (programátoři nebývají ve výsledku placení za časté vyrušování, ale za kvalitně odvedený kód, naopak např. v popisu práce obchodníka, sekretářky, pracovníka hot-line je právě rychlá reakce na aktuální vyrušení)
   - Analytici, a šéfové všeho druhu by měli zvládat lépe než např. programátoři

2. Technické prostředky (telefon, e-mail, sms,…)
   - Výhody – odpadá nejistota v sociálním kontaktu, často není okamžité rušení (e-mail nemusíte okamžitě číst, ani na něj odpovídat, pokud pracujete na důležitějších věcech), poskytování textů, které přečteme, když na ně máme klid, poskytování jednoduchých technických řešení
   - Nevýhody – zdlouhavé řešení zejména tvořivých problémů, chybí okamžité doplňování a rozvíjení nápadů, omezené komunikační možnosti

10. **Chirurgický tým – dobří programátoři jsou obecně 5-10x výkonnější než průměrní programátoři** (avšak po měření ve skupině zkušených programátorů, Sommerville uvádí rozdíl až 25x)

Problém

- těžko seženu větší počet dobrých programátorů (kvalitních lidí) - musím počítat s tím, že v týmu budou lidé různých kvalit

A jak to řešit?

- dobrý programátor vyvíjí, ostatní mu pomáhají (provádějí, co je třeba v daném čase)
- pro větší projekt vznikají malé týmy, každý veden šéfprogramátorem, tyto týmy koordinovány manažerem (zkušenost, každý člověk, kterého řídíte, vás stojí 10% času -> vznik hierarchických úrovní řízení organizace)
- avšak nebezpečí - touto hierarchickou úrovní špatně procházejí špatné zprávy, vrcholový management se pak těžko dovídá o skutečném stavu projektu (opět záleží na kvalitě lidí a kvalitě komunikace)
- zkušenost a její role

11. Většina chyb není v kódu, ale v návrhu

Proč?

- Návrh se provádí dříve, není možné okamžitě ověřit, jestli je návrh v pořádku

Jak to řešit?

- návrh napsat hlavní program, který bude volat top-level procedury (které jsou na začátku prázdné), postupně se přidávají moduly do celého systému

Pozn: Programátoři správně naprogramují to, co se jim řeklo, ale co se jim řeklo chybně (anebo taky vůbec neřeklo), je chybně

12. Odhad protrvání projektu

Napsat překladač je 3x náročnější než aplikační programy
Napsat systémový program je3x náročnější než napsat překladač

Celkovou práci na projektu lze rozdělit cca takto

- 1/3 plánování¨
- 1/6 kódování
- 1/4 testování modulů
- 1/4 testování systému

Z toho plyne, že kódování je časově nejméně náročná část, obtížnější je rozdělit systém do modulů a zajistit jejich komunikaci

13. „Zmrazení“ kódu a verzování – aplikaci je nutné v určité fázi „zmrazit“

Proč?

- software je „neviditelný“, některé skutečnosti se ukáží pouze na určitém stupni vývoje, pokud poskytujeme budoucímu uživateli zpětnou vazbu, pak nové skutečnosti mohou změnit jeho náhled a požadavky, poté bude muset být změněn i samotný software

A jak to řešit?

- Budoucí uživatel se k funkčnosti systému vyjadřuje jen do určitého stavu vývoje (jinak by nebylo možné software dokončit), poté je funkčnost softwaru zmražena a další změny jsou zpracovávány až v další verzi

14. Specializované nástroje – tým by měl používat vzájemně sladěné nástroje pro vývoj softwaru

15. Možnosti snižování nákladů

- Programátoři by měli být najmuti až po vytvoření architektury systému, to může trvat i několik měsíců a v tomto čase by neměli co na práci (avšak zákoník práce, etika apod.)
- Software si koupit
- Univerzální zásada softwarového inženýrství: Nedělejte znovu to, co již bylo uděláno. (Samozřejmě za předpokladu, že to, co bylo uděláno, je kvalitní a prodává se za rozumnou cenu – a to bývá většinou splněno)

Brooks a aplikace jeho myšlenek:

Manažeři dělají stále stejné chyby jako Brooks popisuje -> Brooks: „nazývají tuto knihu biblí softwarového inženýrství a …protože ji každý čte, ale nikdo se tím neřídí“

Brooksův článek „No Silver Bullet“ (1986) (legenda říká, že vlkodlaka lze zasřelti pouze stříbrnou kulkou)

Tvrdí, že žádná technologie nebo technika managementu nezpůsobí do 10 let řádové zlepšení produktivity vývoje SW, a…měl pravdu

Za silver bullet byly považovány např.

- vysokoúrovňové jazyky
- OOP
- Vývojová prostředí pro tvorbu programů
- Umělá inteligence
- Expertní systémy
- Grafické systémy
- Verifikace programů

Místo toho „pouze“ postupná vylepšení

A proč to nefunguje?

- Vývoj softwaru žádá prakticky dokonalý nebo téměř dokonalý výsledek, aby byl tento software použitelný,
- dosáhnout dokonalosti u velkého systému vyvíjeného od začátku je věc velmi náročná až „nelidská“, v každém případě však velmi drahá,
- větší šanci bychom měli, kdyby měl pravdu Descartes: (považující svět za mechanicko-materialistický celek, s vlastními zákony, kde se nevyskytují zásadní rozdíly mezi přírodou živou a neživou)

Myslím, tedy jsem.

Ale spíše se zdá, že :

Nejsme myslící systémy, které mají emoce, jsme vysoce emoční systémy, které někdy myslí.

Aneb pokud se s tím smíříme (boj se zdá být po několik dalších tisíce generací bojem don-Quijota s větrnými mlýny), pak můžeme hledat cesty a zlepšení, jak vyvíjet velký softwarový celek bez duševního strádáni, ještě lépe však s příjemným pocitem

A tak v následujících kapitolách se budeme věnovat věcem převážně rázu technickému, přesto však občas zabloudíme i k věcem netechnickým.