

01 Softwarové inženýrství

tags: *rssss-základ, swing*

> Softwarové inženýrství. Proces vývoje SW. Metodika Unified Process (UP). Agilní vývoj SW. Fáze testování a typy testů. Softwarové metriky, refactoring kódu. Odhadování úsilí. Údržba a znovupoužitelnost. Kvalita softwaru. (PA017)

> Stránka, kde je téma spracována o dost stručněji <http://statnice.dqd.cz/mgr-szz:in-gra:21-gra>

> https://docs.google.com/document/d/1JVC34-jBqKhnyxt9YNellD_LChUsvREdgS9NaEjvI/edit#heading=h.jnc4k9d4jod3

Softwarové inženýrství

- Softwarové inženýrství je samostatný inženýrský obor, který řeší systematický přístup k vývoji, provozování, údržbě a nahrazování software.
- Jako obor s certifikátem v USA uznáno v roce 1997.
- Chybí ucelená teorie, základní terminologie není jednotná.
- Praktici používají kolekce technik, které se zdají být funkční.

Dobře řešený software

- **Udržatelnost:** Měl by být umožněn vývoj SW podle měnících se potřeb zákazníka.
- **Spolehlivost:** Mezi atributy SW, na který je spolehnutí, patří spolehlivost, ochrana a bezpečnost. SW by neměl při výpadku systému způsobit fyzické, ani ekonomické škody.
- **Efektivita:** SW by neměl plýtvat prostředky systému (paměť, čas procesoru a pod.).
- **Použitelnost:** SW by měl mít přiměřené uživatelské rozhraní a odpovídající dokumentaci.

Kritické faktory SW produktivity

- **Složitost:** Lze ji charakterizovat nepřímo, některými viditelnými atributy programu (architektura, počet proměnných).
- **Velikost:** Programování v malém vs. programování ve velkém.
- **Komunikace:** Jednotlivec, malý tým, velký tým.
- **Čas, plán práci**
- **Neviditelnost SW**

Programování v malém

- Ověřené techniky.
- Shora-dolů, strukturované kódování, postupné zjemňování, zdokonalování (step-wise refinement).
- Inspekce logiky a kódu.
- Nástroje - překladače, odvívovače.

Programování ve velkém

- Plánovací mechanismy - dělení práce, harmonogram, zdroje.
- Dokumentovaná specifikace.
- Strukturovaný tým.
- Formalizované soubory testů, testovací příklady.
- Formalizované inspekce.

Proces vývoje SW

Vývoj software je proces, při němž jsou: - **uživatelské požadavky** - □ transformovány na - **požadavky na software** - □ transformovány na - **návrh** - □ implementován jako - **kód** - □ testován, dokumentován a certifikován pro - **operační použití**

Základní aktivity při vývoji SW

- **Specifikace:** Je třeba definovat funkcionality SW a operační omezení.
- **Vývoj:** Je třeba vytvořit SW, který splňuje požadavky kladené ve specifikaci.
- **Validace:** SW musí být validován („kolaudován“), aby bylo potvrzeno, že řeší právě to, co požaduje uživatel.
- **Evoluce:** SW musí být dále rozvíjen, aby vyhověl měnícím se požadavkům zákazníka.

Viditelné znaky výroby SW

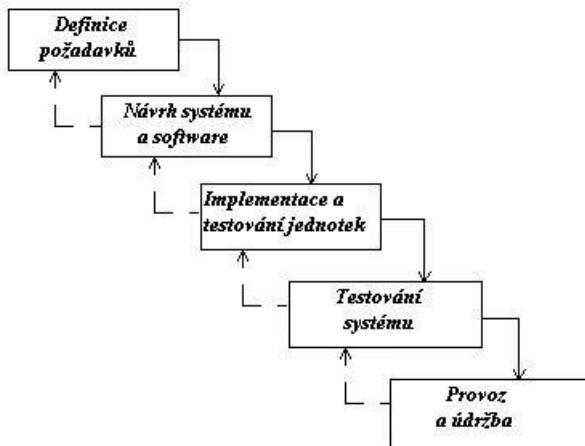
- **Artefakty**
 - Výpis programů
 - Dokumentace
 - Data
 - Zdrojové soubory
- **Procesy**
 - Pracovní postupy
 - Uznávaná pravidla (rules-of-thumb)
 - Interakce mezi členy týmu

Charakteristiky výrobního procesu SW

- **Srozumitelnost:** Je proces explicitně definován a je snadné porozumět definici procesu?
- **Viditelnost:** Výstupní procesní aktivity ve zřetelné výsledky tak, že postup procesu je viditelný zvenčí?
- **Spolehlivost:** Je proces navržen tak, že se lze chybám procesu vyhnout, nebo je zachytit dříve, než zaviní chyby ve výrobku?
- **Přijatelnost:** Je definovaný proces přijatelný a použitelný inženýry zodpovídajícími za produkci?
- **Robustnost:** Může proces pokračovat i v po výskytu neočekávaných problémů?
- **Udržovatelnost:** Může se proces vyvíjet tak, aby odrazil měnící se organizační požadavky nebo identifikovaná zlepšení procesu?
- **Rychlosť:** Jak rychle lze realizovat výrobní proces, který z dané specifikace vytvoří hotový systém předaný zákazníkovi?
- **Podporovatelnost:** Do jaké míry lze aktivity procesu podpořit nástroji CASE? (CASE nástroje primárně umožňují modelování IT systému pomocí diagramů)

Model životního cyklu „vodopád“

Všetky modely sú dôležité!!! **Manažeři ho majú rádi, pretože sa jednoducho riadi.** - Postupovat podľa vodopádového modelu znamená priebeházať od jednej fáze k nasledujúcim písne sekvenčným zpôsobom. Nejprve sa napríklad pripraví specifikace požadaviek, ktoré sú pevné dané. Teprve když sú požadavky úplne kompletné, prejde sa k návrhu. - Integrace systému zároveň s jeho testy - Problémy s cenou v prípade nezdaru v niektoré z pozdnejších etap - Vhodný skôr na malé projekty - Pokiaľ dokážeme vytvoriť jednotlivé etapy naraz, tak nám vznikne lepšia architektúra



Problémy

- špatná reakce na změnu - **velký problém!**
- Reálné projekty nedodržují jednotlivé kroky v předepsaném pořadí.
- Uživatel nedokáže v počátečních etapách formulovat požadavky na systém zřetelně a přesně.
- Zákazník musí být trpělivý.
- Pozdní odhalení nedostatků může vážně ohrozit celý projekt.

Viditelnost životního cyklu „vodopád“

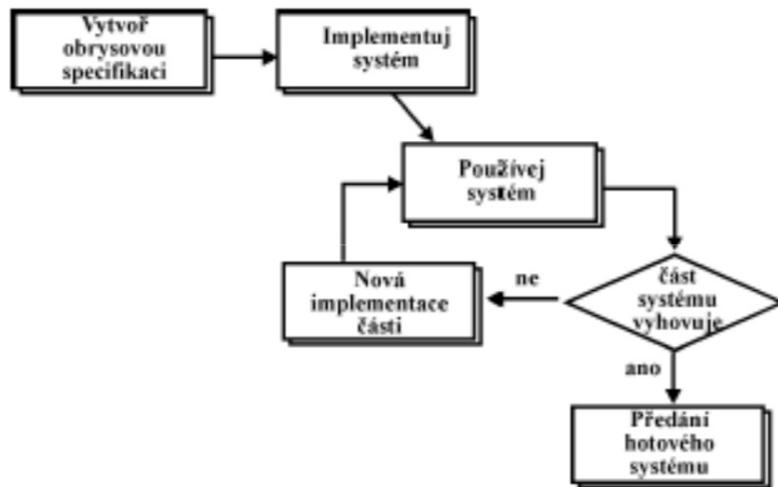
Aktivita	Výstupní dokumenty
Analýza požadavků	Studie proveditelnosti Obrysové požadavky
Definice požadavků	Dokument požadavků
Specifikace systému	Funkční specifikace Plán testování Návrh uživatelského manuálu
Návrh architektury	Specifikace architektury Plán testování systému
Návrh rozhraní	Specifikace rozhraní Plán integračních testů
Podrobný návrh	Specifikace návrhu Plán testování jednotek
Kódování	Kód programu
Testování jednotek	Protokol o testování jednotek
Testování modulů	Protokol o testování modulů
Integrační testování	Protokol integračních testů Konečný uživatelský manuál
Testování systému	Protokol testování systému
Přejímací testování	Konečný systém a dokumentace

Model „inkrementálního“ životního cyklu

Modifikace vodopádového modelu. Zjistilo se, že velké věci není možné dělat vodopádem. Finální projekt je rozdelen na dílčí verze, jednodušší části, vznikají tzv. inkrementy, na které postupně přidáváme (nabízíme) další funkcionality (princip verzování). Jednotlivé verze se vyvíjejí vodopádovým modelem, ale čas na jejich realizaci je menší a můžeme sledovat vývoj systému.

Je určený na velké projekty.

Princíp: Vytvorte mi systém s týmito 5 funkcionálitami. Najprv si vytvoríme nejaký základ a následne je funkcionálita dodávaná v inkrementoch v priebehu viacerých iterácií. Jedna iterácia vyzerá tak, že vyberieme do nej, akú funkcionálitu chceme pridať, tú prejdeme malým vodopádom(analýza, návrh, implementácia, otestovanie, integrácia so zvyškom), pridáme na konci iterácie túto funkcionálitu(hotový inkrement) a otestujeme či funguje celý systém. Inkrementálny model nesie podstatne menej rizík, keďže vytvárame softver po častiach tak pokiaľ niečo zlyhá, tak to zlyhá len v danom inkmente a nie ako celok ako je to v prípade Vodopádu. Produkt dodávame priebežne s obmedzenou funkcionálitou ktorá sa každou iteráciou rozrástá, to znamená že zákazník môže produkt používať v určitej podobe skôr, ako pri vodopáde, kde to môže začať používať až celé naraz.

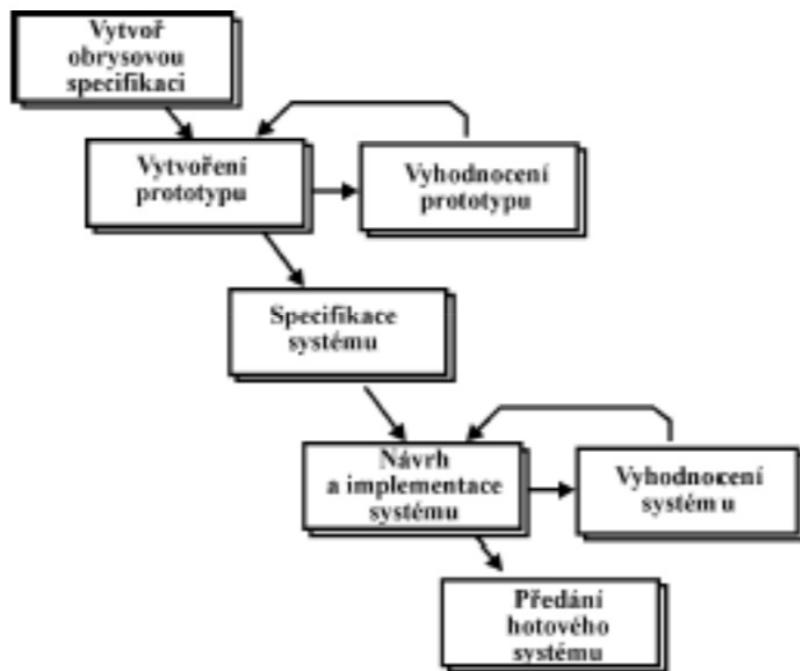


Problémy - V prípade že budeme pridávať jednotlivé inkrementy, tak nebude návrh v takej kvalite, ako keby ho spravíme naraz - Obrysová specifikace vs. skutečnosť - Dokumentace programu vs. specifikace - Údržba zvyšuje entropii - Z pohľadu manažmentu je zložitejší ako vodopád - Dokumentácia môže mierne zaostávať, neupdatne sa po každej iterácii ale povedzme raz za dlhšie časové obdobie, to záleží aj na zmluve - Zmenové požiadavky chcú taktiež určitú réžiu, toto dobre rieši Scrum

Vývoj podľa obrysovej specifikace

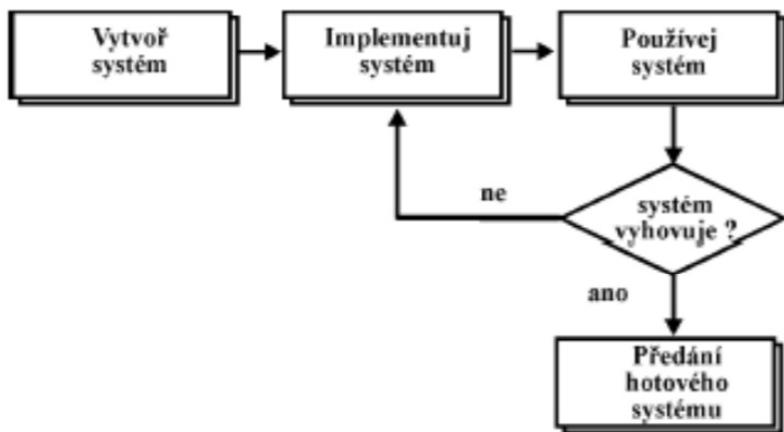
- **Jsou vyžadováni vysoce talentovaní pracovníci.** Průměrný tým nelze použít pro tento typ vývoje. Úspěšné produkty byly takto vytvořeny malými tímy vysoce talentovaných pracovníků.
- **Systémy jsou obvykle špatně strukturované.** Neustálé změny poškozují strukturu systému. Evoluce je obtížná a nákladná.
- **Proces není viditeľný.** Manažer potřebuje pravidelné výstupy, aby mohl řídit proces. Při rychlém vývoji není efektivní produkovať dokumenty, které přesně zachycují každou verzi.

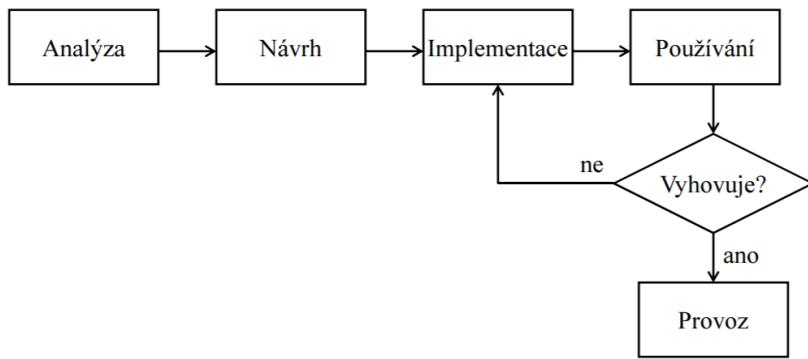
Model životného cyklu „prototypování“



- Zákazník nedokáže vyjadriť svoje požiadavky, vytvoríme mockup ktorý mu ukážeme, na ten mockup nám dá feedback, upravíme mockup, ukážeme zákazníkovi, ak je to ok, mockup zahadzujeme a ideme implementovať
- Tvorba prototypů **probíhá za účelem získávání poznatků, neslouží** ako predloha k vývoji
- Po specifikaci je prototyp zapomenut
- Pri 3 mockupe(prototype) vyzeráme ako dementi a zákazník od projektu pravdepodobne odstúpi

Model životného cyklu „výzkumník“





Problémy

- Obtížné manažerské řízení
 - Neexistující či neplatná dokumentace
 - Nenahraditelnost řešitelů

Nemáme požiadavky, snažíme sa vytvoriť lepšiu technológiu prípadne niečo nové.
Požiadavky typu: F1, nech vyvinú formulu, ktorá vyhrá šampionát.

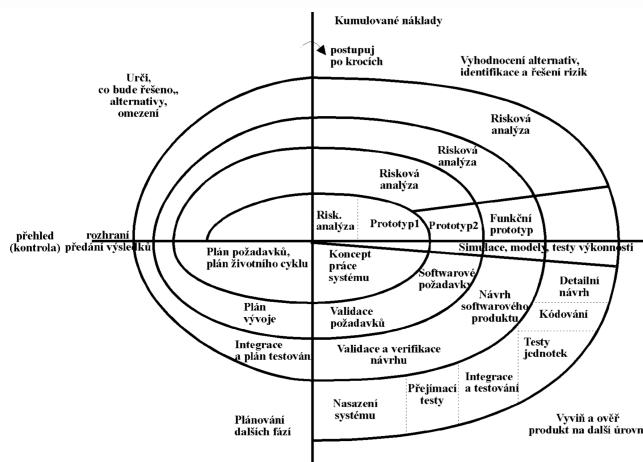
Tento typ nepatrí do komerčného sveta.

Typické pri výskume kde máme vyhradené zdroje a čas. Príklad: máme určité množstvo zdrojov a času, po tom čase s pozrieme čo sme výprodukovali a ak nič zmysluplné tak končíme.

Je to experimentování, u kterého často netušíte, jak dopadne. Metoda pokus/omyl, ve snaze zjistit od zákazníka co vlastně chce tím, že mu předkládáme hotové systémy.

Spirálový model životního cyklu (Boehm, 1988)

Vývoj v cykloch/iteráciach, prostredníctvom ktorých sa približujeme k danému cieľu. Dochádza zde k neustálemu opakovaniu jednotlivých stavov vývoja *navrhnu-naimplementuju-zkusím*. Mezi jednotlivé verze (inkrementy) sú vložené ďalší procesy ako zhodnocení verzie z pohľadu finálneho systému, či pridanie nových požiadaviek zákazníka. V priebehu iterácie nemusí vzniknúť vždy inkrement v zmysle novej funkcionality, môžme niečo opraviť, upraviť dokumentáciu, vylepšiť analýzu atď.



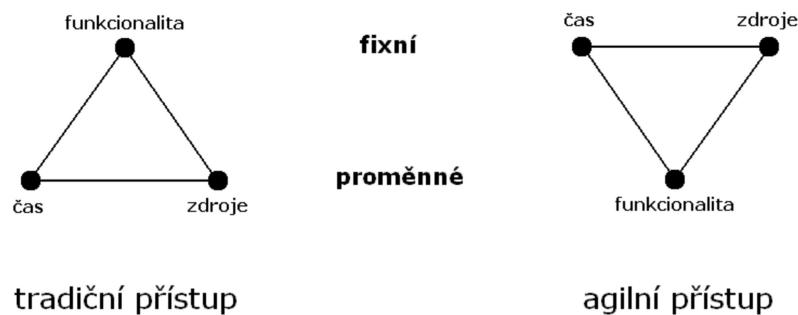
Základné prístupy k vývoju softveru

Prediktívny (Tradičný)

- Fixný funkcionality, čas a zdroje sa mohou meniť • Výstup projektu je dopredu stanovený a musí sa dodržať • Rigidný (těžce upravovateľný) • Zamiera na proces • Fixní požiadavky
- Dôkladné plánovanie dopredu • Príklad: **Unified Process** • Príklad použitia: Jadrová elektráreň

Agile

- Fixný čas a zdroje, inkrementy(funkcionality) sa pridávajú kym nevyčerpáme zdroje a čas
- Výstup projektu nie je vopred jasne stanovený, funkcionality sa pridáva po jednotlivých inkrementoch a priorita požiadaviek sa mení v priebehu životného cyklu • Flexibilný a adaptabilný • Kladie dôraz na ľudí a medziľudske vzťahy(soft skills), t.j. komunikácia v rámci tímu a komunikácia so zákazníkom • Pravidelne aktualizované požiadavky • Minimálne plánovanie vopred • Príklad: **SCRUM** • Príklad použitia: E-shop

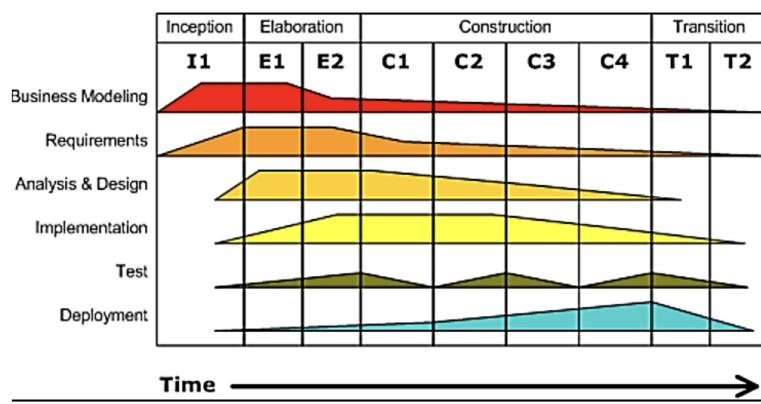


Metodika Unified Process (UP).

- Iteratívny a inkrementálny
- Prediktívny vývojový framework, vieme aký bude konečný výsledok. Pokiaľ nastanú komplikácie tak pridáme ďalšiu iteráciu, to nás sice bude stáť ďalšie zdroje ale projekt musí byť dokončený, to znamená že cieľ je jasné daný
- Risk driven, to znamená, že reagujeme na zmeny. Zmenový proces by mal byť vopred definovaný.
- Požiadavky sú zachytené prostredníctvom use case diagramov
- Je kladený dôraz na architektúru systému.
- Počas celého procesu sú využívané UML diagramy

Neexistuje žiaden univerzálny proces. UP je flexibilný a rozširiteľný

UP Lifecycle



Source: (Arlow, 2005)

V každej iterácii UP si prejdeme celým cyklom(business modeling, požiadavky, analýza a návrh, implementácia, testovanie a nasadenie), rozdiel je v tom, v ktorej iterácii bude aký pomer činností(nebudeme nasádzat' v prvej iterácii kedy ešte len rozmyšľame aký programovací jazyk použiť).

Dĺžka iterácií je pravidlne v rozmedzí mesiacov(1-3).

Jednotlivé fázy UP

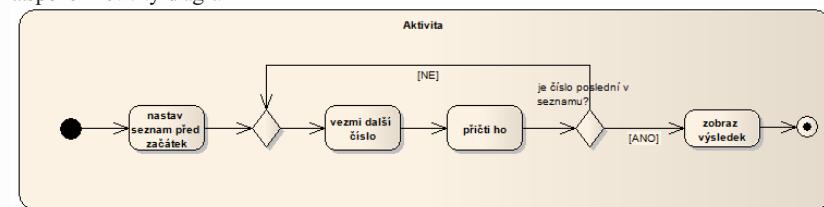
Doležité!!!

- **Zahájenie(Inception):** Takmer predprojektová fáza. Je to realizovateľné? Vznikajú obrysové požiadavky a bavíme sa skôr z biznisového hladiska čo to vlastne bude a čo by to malo robiť. Vytvoríme si zoznam požiadaviek. Identifikujeme si riziká ktoré sa budú postupne spresňovať
 - milník(výstup): Cieľe(Objectives)
 - UML diagramy: Activity diagram
- **Príprava(Elaboration):** Zberame požiadavky na daný produkt. Výkonávajú sa činnosti predchádzajúce programovaniu, to znamená analýza a návrh. Programátori začínajú vytvárať prototypy a rozmyšľať ako to bude asi vyzeráť, "prokopnout nějakou technologii". Chystaju sa klúčové veci, frameworky, interface s externými systémami atď. Vytvoríme si use case model, static a dynamic model
 - milník: základ architektury, zákazníkovi dávame podpísat' špecifikačný dokument, upresňujem si plány.
 - UML diagramy: Dynamický model(sequence diagram) statický model(class diagram)
- **Konštrukcia(Construction):** Vývoj produktu a testovanie. Vývojári vyvíjaný na svojom prostredí(dev) nasádzajú na test prostredie ktoré by sa malo čo najviac podobáť produkčnému a tam to testeri v tejto fáze testujú.
 - milník: počáteční funkcionality
 - UML diagramy: komponent diagram, class diagram, object diagram
- **Predávanie(Transition):** Předání produktu uživatelům, nasadenie produktu na produkčné prostredie a ďalšie otestovanie.
 - milník: release produktu

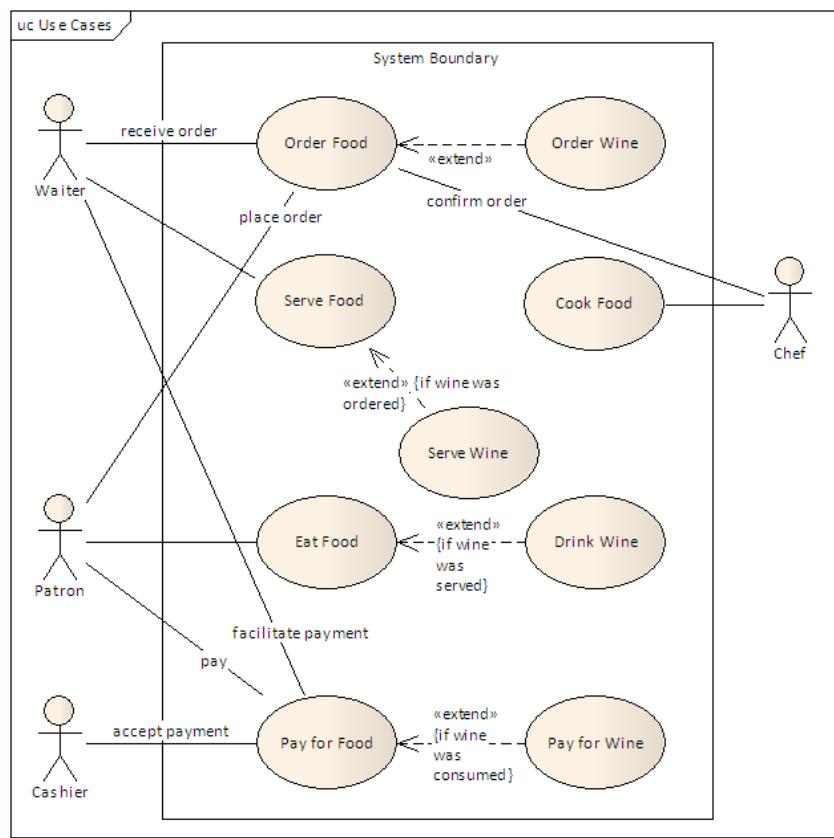
UML v Unified process

Každá z těchto fází by neměla trvat více než 3 měsíce, rozdíl mezi dvěma na sebe následujícími iteracemi se nazývá inkrement. Každá fáze musí obsahovat 6 workflows: - **Business Modeling** - Activity diagram(procesný model, chceme si namodelovať proces práce s daným systémom) - **Requirements** (Požadavky) - Use Case diagram(používá se pro funkčné požiadavky) - **Analysis & Design** (Analýza a design) - Class, Sequence, Collaboration diagram(similar to sequence) - **Implementation** (Implementace) - Class, Object, Component diagram - **Test** (Testování) - Use Case, Class, Activity diagram - **Deployment** - Deployment diagram

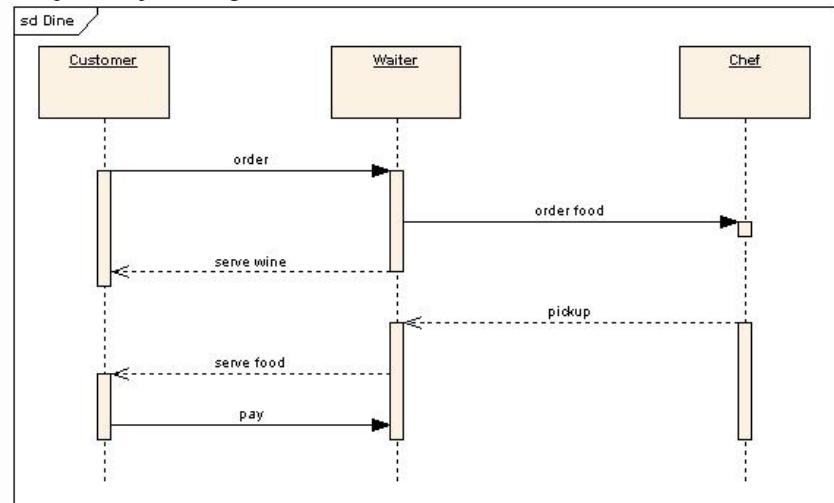
::spoiler Activity diagram



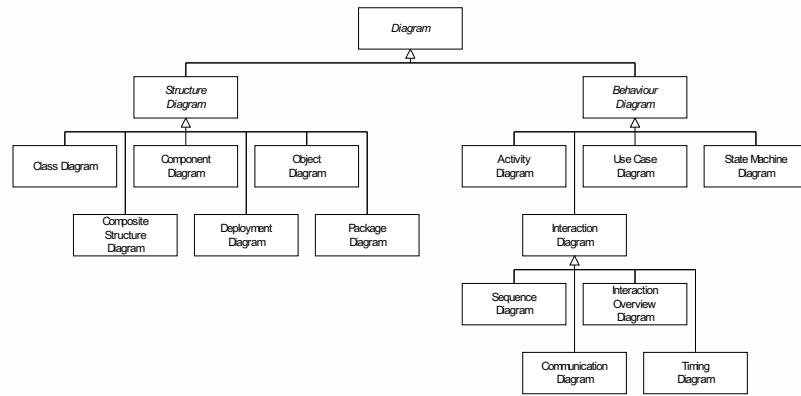
::::spoiler Use Case diagram



::::spoiler Sequence diagram

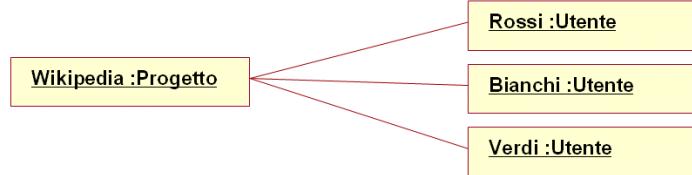
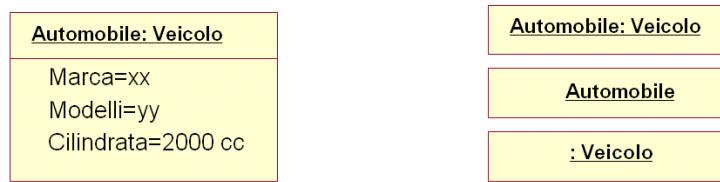


::::spoiler Class diagram

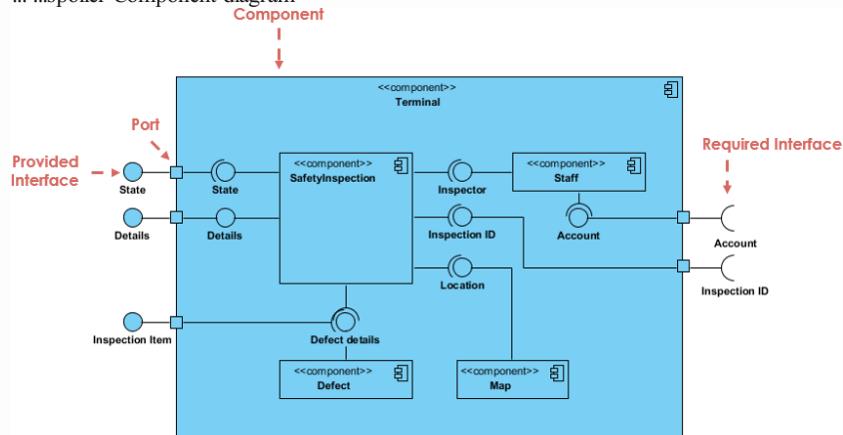


::::spoiler Object diagram

Object Diagram



::::spoiler Component diagram



::## Agilní vývoj SW Agile = hbitý, čílý, bystrý, svižný.

Hlavná priorita nie je dodať celý produkt v dohodnutej kvalite. Zákazník sa procesu zúčastňuje, prioritizuje požiadavky a hovorí čo chce aby bolo vyvinuté v ďalšej iterácii, má možnosť priority meniť atď. Zákazník nemusí dostať všetko čo chcel, ale projekt končí s pocitom že za daných okolností a za svoje finančie dostał toľko, koľko bolo možné.

Manifest agilního programování

Umožnit změnu je mnohem efektívnejší, než se ji snažit zabrániť.

Je třeba být připraven reagovat na nepředvídatelné události, protože ty nepochybňě nastanou.

- **Individuality a interakce** mají přednost před **procesy a nástroji**.
- **Fungující software** má přednost před **obsáhlou dokumentací**.
- **Spolupráce se zákazníkem** má přednost před **sjednáváním smluv**.
- **Reakce na změnu** má přednost před **plněním plánu**.

Společné rysy agilních metodik

Iterativní a inkrementální vývoj s krátkými iteracemi

Vývoj probíhá v krátkých fázích, takže celková funkcionalita je dodávána po částech.

Zákazník tak má možnost průběžně sledovat vývoj, muže se k němu vyjádřit a oponovat změny.

Zákazník má na konci jistotu, že nedostane něco, co neočekával.

Komunikace mezi zákazníkem a vývojovým týmem

V ideálním případě je zákazník přímo součástí vývojového týmu, má možnost okamžitě vidět průběžné výsledky a reagovat na ně.

Zákazník se účastní sestavování návrhu, spolurozhoduje o testech a poskytuje zpětnou vazbu pro vývojáře.

Průběžné automatizované testování

Díky krátkým iteracím se aplikace mění velice rychle, proto je nutné pro zajištění co nejvyšší kvality ověřovat její funkčnost průběžně.

Testy by měly být automatizované, předem sestavené a měly by být napsány ještě před samotnou implementací testované části.

Při každé změně musí být aplikována kompletní sada testů, aby nebyla porušena integrace jednotlivých částí

Konkrétní metodiky

Extreme Programming

Metodika prispôsobená programátorom, má ich motivovať, pretože najvyššia hodnota je vytvoriť čo najkvalitnejší kód.

Hodí se pro menší projekty a malé týmy, vyvíjející software podle zadání, které je nejasné nebo se rychle mění.

Jediným exaktním, jednoznačným, změřitelným, ověřitelným a nezpochybnitelným zdrojem informací je zdrojový kód.

Používá běžné principy a postupy, které dotahuje do extrému. Například: - Jestliže se osvědčují revize kódu, bude se neustále revidovat (myšlenka párového programování) - Pokud se vyplácí testování, bude se nepřetržitě testovat, a to i u zákazníka (testování jednotek, funkcionality, akceptační testy). - Osvědčuje-li se návrh, stane se součástí každodenní činnosti (refaktorizace).

Principy XP

- **Rychlá zpětná vazba**, která spočívá v rychlém zjištění stavu systému po provedené akci, vyhodnocení akce a uložení výsledku vyhodnocení co nejdříve zpět do systému.
- **Předpoklad jednoduchosti**, představuje v mnoha ohledech nejobjížnější princip, protože je v protikladu s tradičním pojeticím programování, kdy se vše plánuje a

navrhuje do budoucna tak, aby to bylo znova použito. XP naopak předpokládá u řešitelského týmu schopnost přidat složitost tam, kde to bude v budoucu účelné.

- **Přírůstková změna**, vychází z předpokladu, že velké změny provedené najednou nefungují. Veškeré změny v projektu se proto provádějí pomocí malých přírůstků.
- **Využití změny**, vychází z předpokladu, že nejlepší strategie je ta, která si zachová co nejvíce možností řešení nejhaléhavějších problémů projektu.
- **Kvalitní práce**, která představuje fixní proměnnou ze čtyř proměnných pro posouzení projektu (šíře zadání, náklady, čas a kvalita) s hodnotou vynikající, při horší hodnotě členy týmu práce nebude bavit a projekt může skončit neúspěchem.

Feature-Driven Development

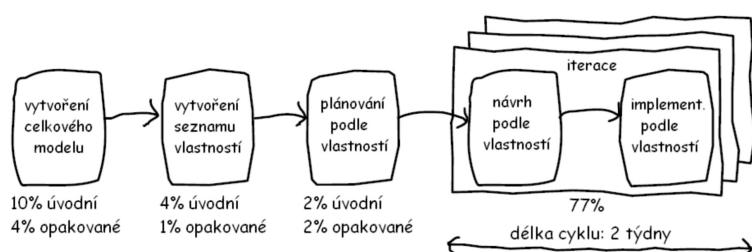
Vývoj po malých kouscích (vlastnostech, rysech), což jsou elementární části funkcionality přinášející nějakou hodnotu uživateli.

Vývoj probíhá v pěti fázích, první tři jsou sekvenční, poslední dvě pak iterativní. Iterace trvají zpravidla 2 týdny. Začíná se vytvořením modelu, ten se převede do seznamu vlastností, které se postupně implementují.

Měří pokrok ve vývoji projektu, FDD detailně plánuje a kontroluje vývojový proces.

Zaměření na dodávání fungujících přírůstku každé dva týdny

Fáze metodiky z pohledu rozdělení času:



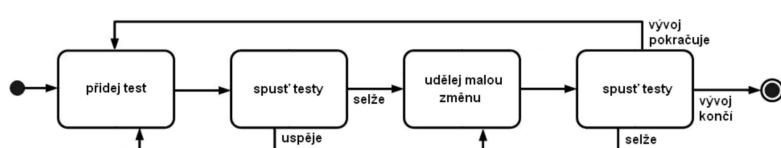
Test-Driven Development

Princíp: Čo prejde testom, to je v poriadku. Požiadavky neboli nejak definované. Keď nejaký vznikol, tak prvé čo sa spravilo, bolo vytvoriť test, pokiaľ daná funkcionálita prešla testom, tak je to v poriadku. Nezabývá se tvorbou specifikací, plánu a dokumentace, to si každý tým musí zvolit sám podle toho, jak mu to vyhovuje.

Doporučuje přistoupit k testům jako k hlavní fázi celého vývojového procesu.

Základním pravidlem je psát testy dříve než samotný kód a implementovat jen přesně ty části kódu, které projdou testem.

Postup metodiky:

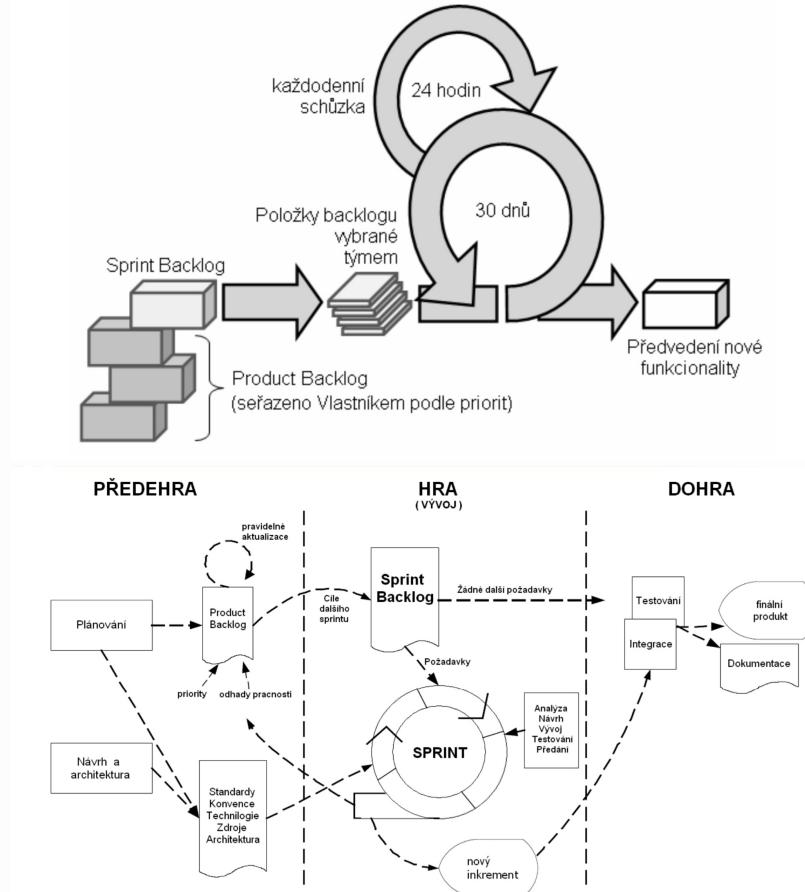


SCRUM Development Process

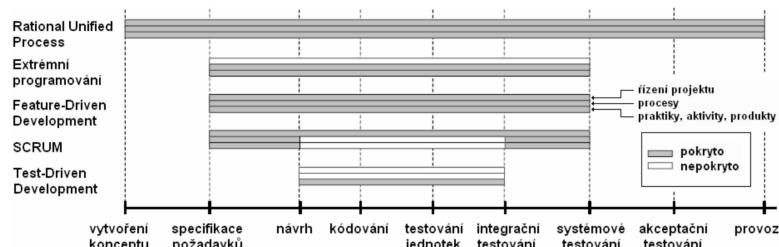
Doležité!!!

Iteratívny/inkrementálny vývoj, jedna iterácia trvá 2-4 týždne, nie viac. Na konci každej iterácie by mal vzniknúť nový, funkčný inkrement.

https://is.muni.cz/auth/el/fi/podzim2020/PA017/um/cz/02_ITPM.pdf



Srovnání metodík z pohľadu životného cyklu SW



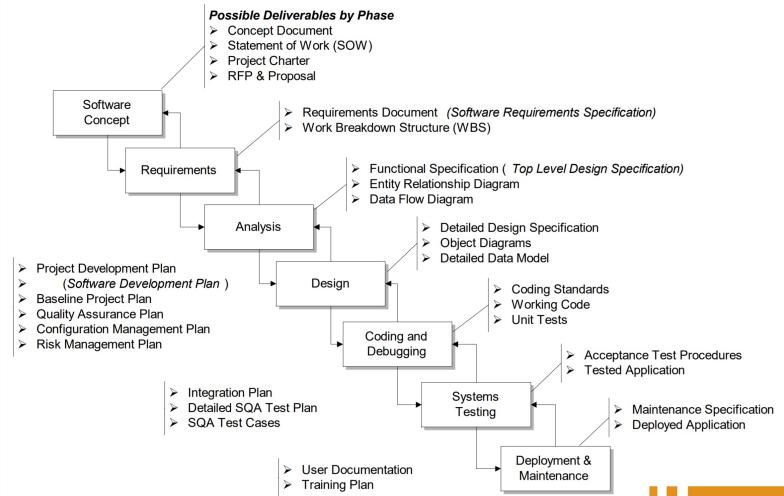
Fáze testovania a typy testu

- Testovanie je proces **spuštění** programu s cílem nalézt chyby.
- Dobrý testovací prípad má vysokou pravdepodobnosť nálezení dosud nenalezené chyby.
- Úspešný test je takový, ktorý odhalí dosud neodhalenou chybu.
- Je dôležité aby tester bola osoba, ktorá sa nepodieľala na predošlých fázach vývoja softveru(nemajú súčit, vyššia pravdepodobnosť že niečo nájdú)

- Test je úspešný, pokud zjistí prítomnosť jedné či viacej chyb v programu. (Myers, 1979)
- Je to destruktívna činnosť, ideálne pokiaľ máme na vývoj a testovanie dva rozdielne tímy.
- Detailná znalosť struktury programu usnadňuje hľadanie a opravu chyb.

Typy testov:

1. vizuálne testovanie
 2. unit testy
 3. integračné testy (pozn. robíme postupne tak, že po každej zaintegrovanej/pridanej komponente testujeme. Nerobíme to naraz!)
 4. systémové testovanie (performance, penetračné, užívateľské, akceptačné, beta testy (napr. na testovacom prostrední))
- ### Produkty podľa etap

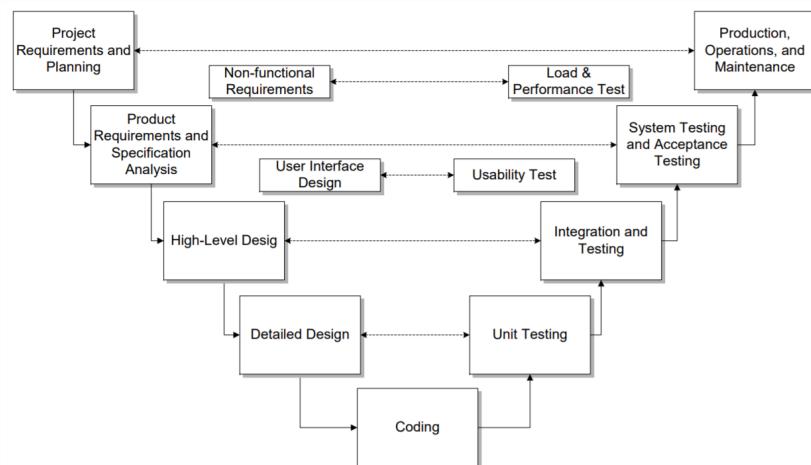


Co testování ukazuje?

- Testování nemôže ukázať nepôrdomnosť defektov, môže pouze ukázať, že v softwaru sú chyby.
- Testovanie také ukazuje funkcie a výkon.
- A je také ukazateľom kvality software.

V - procesný model

Extrémne dôležité!!! Oblubnená otázka na štátincích



- Na lavej strane V-procesného modelu sú zobrazené jednotlivé vývojové fáze vzniku softveru.

- Na pravej strane sú testy zodpovedajúce danej fázy (validačne-verifikačné činnosti ktoré overujú funkcionality)
- Jednotlivé položky sú chronologicky zoradené, čo znamená že najzásadnejšie chyby ktoré stojia najviac peňazí sú odhalené až ako posledné. Z toho dôvodu, majú zmysel inšpekcie, ktoré majú tieto chyby identifikovať skôr, ako bude ich oprava výrazne drahsia. Inšpekcie je možné robiť už vo fázach analýzy, či je všetko tak, ako má byť.
- Chyby najčastejšie vznikajú pri zbere požiadaviek a v návrhu.
- Testovanie ukazuje chyby, kol'ko ich tam je a je ukazateľom kvality daného softveru, avšak nie je garanciou bezchybnosti.
- Testovanie mimo chýb ukazuje aj výkonnosť (nefunkčné požiadavky, response na request musí byť menej ako 3 sekundy), softver musí byť nie len funkčný ale aj výkonný.
- upravená varianta vodopádu, ktorá zdôrazňuje verifikaci (studie proveditelnosti, revize, inspekce,...) a validaci (testování)
- testovanie produktu je plánované paralelně s jednotlivými fázemi vývoje
- účinnosť odhalení chyb se zvyšuje verifikací – môže byť až 80%!

Validace: test proti specifikovaným funkciám (“Dělat správné věci”) Čo to robí? (black-box) **Verifikace:** test proti vnitřní činnosti (“Dělat věci správně”) Ako to robí? (white-box) Spája sa s cyklomatickou zložitosťou!!! Ked'že technikou white-box chceme prejsť kód do hľbky a prejsť čo najviac priechodov, tak je dôležitá cyklomatická zložitosť, to znamená kol'ko takýchto priechodov máme a aká je celková zložitosť. V kontexte V-modelu, od Akceptácie(vrátané) smerom hore je to validácia(black-box), od Integračných testov(vrátané) smerom dole je to verifikácia(white-box)

Selektívni testy

- I tehy, kdy úplné testování není reálne (prakticky vždy!), testování „bílá skříňka“ by nemelo byt vynecháno.
- Dôležité logické cesty a cykly by mely byt testovány.
- Selektívni testování valídnuje rozhraní a vytváří důvěru ve vnitřní činnost software.

Dynamické testování

- Provedení počítačového programu s předem určenými vstupy.
- Porovnání dosažených výsledků s očekávanými výsledky.
- Testování je vlastně vzorkování, nemôže absolútne prokázať absenci defektov.
- Každý software má vši, testování nezarúčí odvšivení.

Testovací případy

- Klíčové položky plánu testování.
- Mohou obsahovať skripty, data, kontrolné seznamy.
- Mohou mít vztah k *Matici pokryti požadavků* (nástroj pro sledování)

Testování „černá skříňka“

- Funkční testování
- Program je „černá skříňka“
 - Nezajímá nás, jak to pracuje, ale co to dělá.
 - Zaměřeno na vstupy & výstupy
- Testovací případy založené na SRS (specifikacích požadavků)

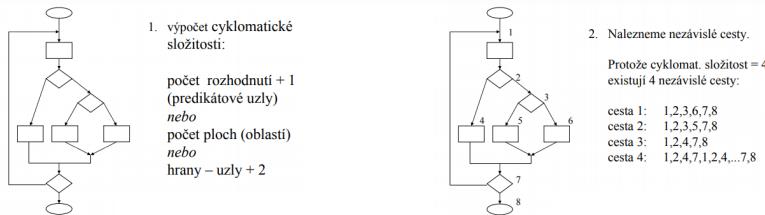
`s := Hledej (NejakePole,UlozenaHodnota)`

<i>velikost pole</i>	<i>hledaný prvek</i>
1.	existuje
2.	není
3.	
4.	je první
5.	je poslední
6.	není
7.	je první
8.	je poslední
9.	není
10.	je v obecné pozici
11.	je v obecné pozici

V tomto testu je obsažena zkušenost s mnoha verzemi vyhledávacích programů.

Testování „bílá skříňka“

- Zohledňuje strukturu programu
- Pokrytí
 - provedené příkazy
 - cesty průchodu kódem



- Vývojový diagram není nutný, ale obrázek pomůže vysledovat příslušné cesty.
- Testy základních cest by měly být provedeny u kritických modulů.

Testování jednotek, modulů

- Typ testování „bílá skříňka“
 - někdy ale jako „černá skříňka“
- Kdo testuje jednotky?
 - vývojáři
 - testy jednotek jsou programovány – stejný jazyk jako moduly – alt. název „Testovací drivery“
- Individuální testy mohou být seskupeny
 - „Kolekce testů“ (Test suites)
- Kdy se testují jednotky?
 - postupně během vývoje
 - po dokončení individuálních modulů

Integrace & Testování

- Vývoj/integrace/testování
 - nejčastější místo, kde dochází k překrývání aktivit
- Někdy je integrace/testování považováno za jednu etapu
- Postupně propojuje funkcionalitu
- QA tým pracuje souběžně s vývojovým týmem

Integrační postupy

Shora dolů (IDT)

- Nejprve je implementováno jádro (kostra) systému.
- Zkombinováno do minimální „skořápky“ systému.
- Pro doplnění neúplných částí se použijí „protézy“ nahrazované postupně aktuálními moduly.

použití „stubs“ (pahýly, protézy) - jednoduché náhražkové objekty se shodným rozhraním.

Testování shora-dolů odhaluje chyby analýzy a návrhu, je v souladu s prototypováním.

Nevýhody TDE - Složité objekty, moduly, nelze jednoduše zaměnit za „protézu“. - Výsledky testů na vyšších úrovních nemusí být přímo „viditelné“.

Zdola nahoru (BUT)

- Začne s individuálními moduly a sestavuje zdola.
- Individuální jednotky (po testování jednotek) jsou kombinovány do subsystémů.
- Subsystémy jsou kombinovány do celku.

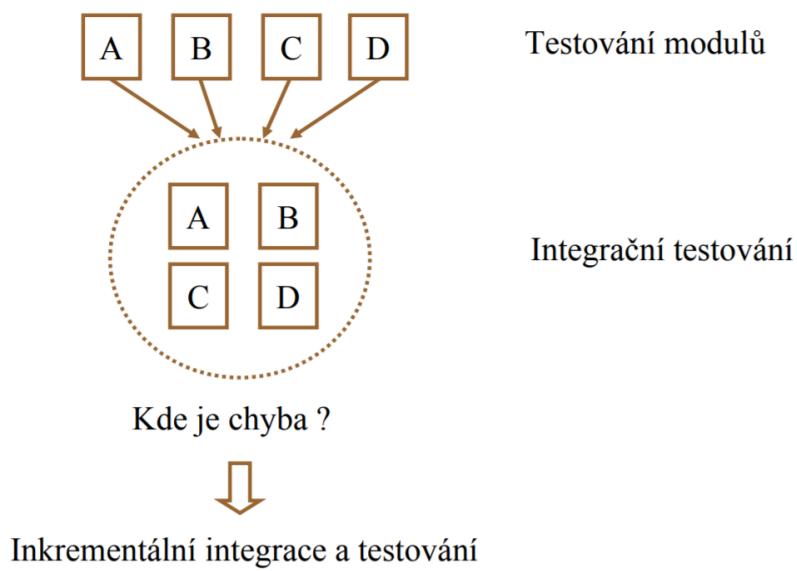
klasický testovací proces s nadřazenými testovacími objekty - „drivers“.

Nevýhody BUT - Čas a náklady na konstrukci „drivers“ pro testování jsou obvykle vyšší, než u „protéz“. - Až v závěru vznikne program použitelný pro předvedení, ve formě „prototypu“.

Obě metody mají své nevýhody, nelze říci, že jedna je nejlepší.

Atributy integrace

- Kdo dělá integrační testování?
 - vývojářský a/nebo QA tým
- Počet pracovníků a rozpočet jsou na vrcholu
- „Jde do tuhého“
- Problémy:
 - práce pod tlakem
 - blíží se datum odevzdání
 - neočekávaná selhání (vši), motivační problémy
 - konflikty při přejímání zákazníkem



Softwarové metriky

Dôležité!!! Software měříme primárně kvůli tomu, abychom dokázali určit kvalitu projektu/procesu. Taktež můžeme měřit velikost a složitost.

Řídit kvalitu znamená nemít tam chyby, snižovat složitost, snižovat velikost jednotlivých modulů. Chceme být schopni tyto jevy nějakým způsobem měřit, na základě těchto měření chceme predikovat jak to s tím softwarem bude vypadat v budoucnosti, odhalit slabé místa. Zajímavá metrika je i to, když je software natolik kvalitní, že přestaneme testovat. Celé je to o tom, že chceme kvalitu SW zvyšovat.

Příklad metriky: Metrika týkající se chybovosti, souvisí s testováním a s řízením kvality. Můžeme to měřit na daný softvér, modul alebo v rámci celého vývoja. Meraná budeme určitě vztažovat za určitú jednotku času. Chyby by mali byť kvantifikované a kategorizované. Kategorizované například podľa typu, prípadne podľa toho, kedy vznikli. Chyby môžu byť kategorizované podľa toho či sú malé, veľké, závažné alebo kritické, to hovorí o tom, ako to blokuje/ovplyvňuje biznis. ### Míra Kvantitatívny údaj o množství, rozmerach, kapacite, nebo velikosti nějakého atributu, produktu nebo procesu. (Počet chyb)

Dôležité!!! - **Přímá míra** (vnitřní vlastnosti): počet řádků kódu (LOC (spíše se používá KLOC - tisíc řádků)), rychlosť výpočtu, velikosť paměti, počet chyb za určitou dobu, ... - dokážeme ich jednoducho vyčíslit - **Nepřímá míra** (vonkajšie vlastnosti): funkčnosť, kvalita, složitosť, pracnosť, spolehlivosť, schopnosť údržby, ... - vieme porovnávať ale zložitejšie vyjadriť, napríklad kvalita ### Metrika Kvantitatívny (číselne vyjádrená) míra, tj. ukazatel do jaké míry se nějaký atribut vyskytuje v systému, komponente nebo procesu. (Počet chyb na 1000 řádků) Metrika může mít i kvalitatívny charakter, tj. nečíselné vyjádření.

Dôležité!!! - **Procesné metriky** - Činnosti súvisiace s procesom výroby softvéru - Koľko funkčných bodov prešlo programátorm rukami - koľko LOC bolo počas procesu napísané - milníky - Pracnosť(effort) - Poskytujú náhľad do procesných modelov, úloh softwarového inžinierstva, produktov alebo miľníkov - Vedú k dlhodobému zlepšovaniu procesov - činnosti spojené s vývojem, doba strávená na jednotlivých úlohách, pôvodní odhad a skutečná reálna doba - **Produktové metriky** - počet riadkov zdrojového kódu hotového produktu - počet strán dokumentácie - počet strán dokumentácie vzhľadom na počet riadkov zdrojového kódu - počet funkčných bodov daného produktu - Hodnotia stav projektu - Monitorujú riziká - Odhaľujú problematické miesta - Upravenie workflow - Vyhodnocujú schopnosť tímu kontrolovať kvalitu - **Metriky zdrojov** - počet lidí - vzdelanie ľudí - koľko ľudí má aké certifikáty - HW prostriedky pre daný projekt - licencie

Metriky založené na veľkosti kódu

Dôležité!!! - Veľkosť softwarového produktu - Lines Of Code (LOC) - 1000 Lines Of Code (KLOC) - Náročnosť v človekomesiach (E/MM) - Počet chyb/KLOC - Cena/KLOC - Počet stránok dokumentácie/KLOC - LOC je závislá na programátorovi a programovacom jazyku

LOC metriky

Dôležité!!! - Ľahké na použitie - Ľahké na porovnanie - Je možné vypočítať LOC existujúcich systémov, ale sledovateľnosť nákladov a požiadaviek môže byť stratená - Závislé na programátorovi a jazyku

Funkčne orientované metriky

- Metriky na základe funkčných bodov (FP)
- Odvodené pomocou empirických (založené na skúsenostach) vzťahov založených na spočítateľných vlastnostach systému

Function Points (FP) (viac rozpisane v Odhadovani usil)

- Počet vstupov
 - Rôzne vstupy od užívateľa
- Počet výstupov

- Reporty, obrazovky, error hlášky, ...
- Počet otázok
 - Vstup, ktorý generuje nejaký výstup
- Počet súborov
 - Logické súbory (databáza)
- Počet externých rozhraní
 - Pripojenie na iné systémy

$FP = \text{celkový počet} * (0.65 + 0.01 * \text{Sum}(F_i))$ Celkový počet vynásobený vähou získanou pre každú organizáciu na základe empirických dát F_i ($i = 1$ až 14) je hodnota na úpravu zložitosti Doležité!!! - **Funkční body produktu** - pri vývoji aplikace "na zelené louce" jsou tyto body takové body, které zůstanou v aplikaci na konci vývoje - **Funkční body projektu** - prošly týmu rukama, zaplatili jsme za ně, nemusí na konci vývoje zůstat (např. body vynaložené na vyzkoušení nějakého postupu a následné předělání zpět), musíme s nimi však nutně počítat - Například bychom měli projekt o 100 funkčních bodech, který bude měnit stávající software. A v rámci tohoto projektu bychom pouze umazali nějakou funkcionalitu (např. umazali 5 funkčních bodů), tj. **funkční body produktu** se snížily na 95, ale **funkční body projektu** = 5.

Metriky založené na FP: - Chyby/FP - Cena/FP - Počet stránok dokumentácie/FP - FP za človekomesiac

Metriky zložitosti

sú závislé na programátorovi a programovacom jazyku - Halsteadova metrika - Cyklomatická zložitosť #### Halsteadovy metrika Doležité!!! Softwarová metrika výpočtu složitosti na základě statistickej analýzy kódu. Používa tyto promenné: n_1 - počet unikátnych operátorov n_2 - počet unikátnych operandov N_1 - celkový počet operátorov N_2 - celkový počet operandov Pomocí nich podle specifických vzorečkov definiuje tyto metriky: Length of sentence: $N = N_1 + N_2$ Vocabulary: $n = n_1 + n_2$ **Estimated length:** $\tilde{N} = n_1 \log_2 n_1 + n_2 \log_2 n_2$ - optimálna dĺžka **Purity ratio:** $PR = \tilde{N} / N$ - pomer čistoty, pokiaľ je kód príliš zložitý tak je $PR < 1$, ak píšu veci zbytočne krátke tak $PR > 1$. Je dôležité aby bolo toto číslo podobné v rámci tímu kvôli zastupiteľnosti ľudí. Volume: $V = N \log_2 n$

Príklad:

```

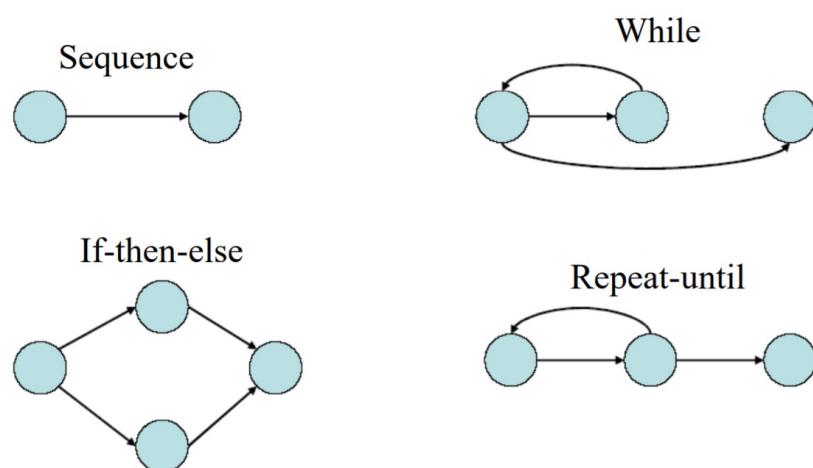
if (k < 2)
{
if (k > 3)
x = x*k;
}

```

- Distinct operators: if () { } > < = * ;
- Distinct operands: k 2 3 x
- $n_1 = 10$
- $n_2 = 4$
- $N_1 = 13$
- $N_2 = 7$

McCabe complexity measure - flow grafy

Jsou založeny na zobrazování kontrolních toků programu. Graf slouží k zobrazení kontrolního toku. Uzly představují úlohy zpracování. Hrany představují tok řízení mezi uzly.

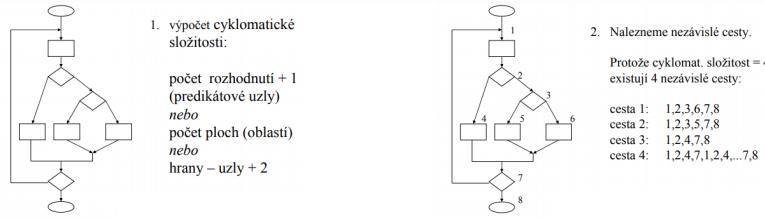


Cyklotomická složitost

Dôležité!!! Metrika zložitosti. *Kolik je možných průchodů grafem nebo kolik je tam nezávislých větví.* Funguje také jako indikátor spolehlivosti. Pokud $V(G)$ je větší než 10 má velký chyboucí potenciál a zle se testuje. $V(G)$ je v podstatě počet uzavřených oblastí v roviném grafu. Cyklotomická složitost vyjadřuje počet průchodů grafem.

Množina nezávislých cest v grafu - $V(G) = E - N + 2 - E$ - počet hran grafu - N - počet uzlů grafu - $V(G) = P + 1 - P$ - počet predikátových uzlů

Nutná dobrá dekompozice tj. složitost do 10. Čím více rozhodování tím složitější graf.



Cyklometrická zložitosť úzko súvisí s testovaním technikou white-box. Technikou white-box chceme pokryť všetky prieходy a ich kombinácie. S vysokou zložitosťou rastie chybový potenciál a klesá produktivita(programátorom sa v tom zle orientuje). Pri vysokej cyklometrickej zložitosti chceme modul dekomponovať.

Coupling

Extrémne dôležité!!! Coupling vyjadruje to, jak máme propojené dva moduly, jinak rečené vyjadruje míru integrace. Kolik to (sw) predáva dát směrem ven, kolik to přijímá, s kolika dalšími systémy to spolupracuje. Coupling metriky sa snaží zjistit, jak se chyba v jedné komponente odrazí na zbytku systému.

Čím je tato hodnota nižší tím lépe - nízká hodnota znamená, že nahrazení jednoho modulu jiným znamená malé integrační úsilí a malé rizika vzhledem na danou složitosť jak do sebe tyto moduly zapadají a jak spolu komunikují. Počítají se předávané parametry mezi moduly.
Datový a regulační tok - signálny, na které potřebujeme reagovat a datové položky(input/output) d_i - input data parameters c_i - input control parameters d_o - output data parameters c_o - output control parameters ##### Globální - globální proměnné používané moduly g_d - global variables for data g_c - global variables for control #####
Environmentální - množství modulů, které ten nás volá a které ho volají w - fan in number of modules called r - fan out number modules that call module ##### Metriky
 $M_c = k/m$, $k = 1$ $m = d_i + ac_i + d_o + bc_o + g_d + cg_c + w + r$ a, b, c, k can be adjusted based on actual data Vzorec vyjadruje šířku rozhraní, které se za těmi moduly skrývá.

Důvody pro měření metrik

- plánování projektu (odhad nákladů, pracnosti, času)
- kontrola kvality produktu
- odhad produktivity
- zdokonalení práce (růst výkonnosti organizace)

Příkladem může být tzv. Halsteadova metrika, která je založena na teorii informací. Výsledkem výzkumu SW metrik jsou metodiky vývoje (SOA, OO, strukturované programování, znalosti a vlivu kvality specifikací atd.) a metodiky odhadu pracnosti a doby řešení COCOMO, funkční body.

Použití SW metrik

- Výzkum: podklad pro hledání metod realizace softwarových produktů, které by přinesly podstatné zvýšení jeho kvality a snížení nákladů a doby vývoje a hlavně rozsahu prací při údržbě softwaru (výzkum metod a zákonitostí vývoje softwaru).
- Normy: základ pro stanovení technicko-ekonomických podkladů pro řízení prací při tvorbě softwaru (normy pracnosti, odhad takových metrik, jako je pracnost či doba řešení) a uzavírání smluv (cena, termíny), předpoklad CMM.
- Kontrola kvality: prostředek sledování spolehlivosti softwaru při provozu a podklad pro řídící zásahy během údržby, procesy zajistující kvalitu.
- Operativa: prostředek sledování průběhu prací při vývoji (dodržování termínů, procento testovaných komponent, trendy počtů chyb, počty nově zanesených chyb,

komponenty s největším počtem chyb, atd.).

Potíže s metrikami

- rozptyl hodnot
- druh SW, obtížně splnitelné termíny realizace
- kvalita zúčastněných
- omezení SW a HW

Refactoring kódu

Refaktorování je disciplinovaný proces provádění změn v softwarovém systému takovým způsobem, že nemají vliv na vnější chování kódu, ale vylepšují jeho vnitřní strukturu s minimálním rizikem vnášení chyb. Při refaktorování se provádí malé až primitivní změny, ale celkový efekt je velký, a to v podobě čistšího, průhlednějšího a čitelnějšího kódu, kód se také lépe udržuje a rozšířuje. Zlepšuje se také celková kvalita kódu a architektura, snižuje se počet chyb a tím i zvyšuje rychlosť vývoje programu. Refaktorování pomáhá pochopit a více si ujasnit kód, což je vhodné zejména upravování zdrojového kódu po někom jiném.

Existuje vazba mezi refaktorováním a návrhovými vzory, a to taková, že vzory popisují definovaný cíl a naopak refaktorování popisuje způsob, jak se k tomuto cíli dostat.

Kdy refaktorovat?

- pokud přidáváme funkcionality
- když potřebujeme opravovat chyby
- revize kódu
- důležitá součást iterativního vývoje (povinná)
- když kód „smrdí“

„Bad smells“

- **Duplicítní kód** – pokud je v kódu na více místech stejná struktura
- **Dlouhá metoda** – záleží na každém, kolik řádků už je pro něj dlouhá metoda, pokud se taková objeví, většinou se v takové metodě řeší více problémů najednou a měla by být rozdělena na metody menší, které řeší právě jeden problém. Pak je kód i čitelnější.
- **Velká třída** – pokud se třída snaží dělat více věcí najednou, není to z pohledu objektově orientovaného programování dobré, jelikož každá třída by správně měla řešit právě jeden problém, tudíž řešením je rozdělit třídu na více tříd
- **Dlouhý seznam parametrů** – metody s dlouhým počtem parametrů jsou většinou nesrozumitelné, pokud tyto předávané parametry obsahuje nějaký objekt, je řešením předat jako parametr tento objekt
- **Příkaz switch** – měl by být řešen pomocí polymorfismu a vyčleněn do samostatné metody
- **Komentáře** – pokud má programátor tendenci komentovat uvnitř metody svůj kód, je většinou lepší tento kód vyčlenit do samostatné metody, a pomocí vhodného názvu této metody odpadne nutnost komentáře
- **Shotgun Surgery** - abychom udělali jednoduchou změnu v kódu, je nutné sahat na mnoho míst, indikátor toho, že máme špatný model, že třídy mají špatnou zodpovědnost
- **Middle Man** - zprostředkování volání objektu zbytečně přes prostředníka
- **Lazy Class** - prázdná skořápka, třída, která nic nedělá
- **Spekulativní obecnost** - kód, který je v programu obsažen, aby sloužil někdy do budoucna

Systémový re-engineering

Znovu napsání celé části systému bez účelu změnit její funkcionality. Aplikovatelné, když některé, ale ne všechny subsystémy vyššího systému vyžadují častou údržbu. Re-engineering se praktikuje až potom, co byl systém nějakou dobu udržován a jeho cena za údržbu stoupá. K reengineeringu se používají automatizovaná zařízení, aby se vytvořil nový

systém, který by se lépe udržoval (i snížení ceny za údržbu). - Reverzní engineering - Vylepšení struktury programu - Modularizace programu - Re-engineering dat

Techniky refaktoringu

- Úpravy metod
 - nahradit algoritmus
 - nahradit dočasnou proměnnou dotazem (metodou, která spočítá její hodnotu)
 - nahradit metodu metodou objektu
 - odstranit přiřazení parametrům
 - přejmenovat metodu a přejmenovat položku – změna názvu na lepší a výstižnější
 - rozdělit dočasnou proměnnou
 - vložit metodu
 - vložit dočasnou proměnnou
 - vyjmout metodu – z dlouhé metody se vyjmě část kódu, který je vložen do nové metody
 - zavést vysvětlující proměnnou
- Přesouvání prvků mezi objekty
 - odstranit prostředníka
 - přesunout metodu a přesunout položku – přesun do vhodné třídy
 - skrýt delegátu
 - vyjmout třídu – vyjmě se část kódu třídy a vložit do třídy nové
 - vložit třídu
 - zavést cizí metodu
 - zavést místní rozšíření
- Organizace dat
 - nahradit datovou položku objektem
 - nahradit pole objektem
 - zapouzdřit soukromou položku – k přístupu k proměnné použít gettery a settery
 - změnit hodnotu na odkaz
 - změnit odkaz na hodnotu
 - zavést objekt null
 - zavést předpoklad
- Generalizace
 - nahradit dědičnost delegováním
 - nahradit delegování dědičností
 - přesunout metodu výš, přesunout položku výš – přesun do předka
 - přesunout metodu níž, přesunout položku níž – přesun do potomka
 - přesunout tělo konstruktoru výš – přesun do předka
 - vyjmout podtřídu
 - vyjmout rodičovskou třídu – vyjmout předka
 - vyjmout rozhraní – vyjmout rozhraní
 - vytvořit šablonovou metodu
 - zrušit hierarchii
- Zjednodušení volání metod
 - nahradit chybavý kód výjimkou
 - nahradit konstruktor tovární metodou
 - nahradit parametr explicitními metodami
 - nahradit parametr metodou
 - nahradit výjimku testem
 - oddělit dotaz a modifikátor
 - odstranit parametr
 - odstranit přístupovou metodu pro zápis
 - parametrisovat metodu
 - přejmenovat metodu a přejmenovat položku – změna názvu na lepší a výstižnější
 - přidat parametr
 - skrýt metodu
 - zachovat celý objekt
 - zapouzdřit přetykování na potomka
 - zavést objekt pro parametry
- Velké programování
 - roztrhnout dědičnost
 - převést procedurální návrh do objektů
 - vyjmout hierarchii
 - oddělit datový model od prezentace

- Ostatní techniky
 - duplikovat sledovaná data
 - nahradit kód typu podtřídami
 - nahradit kód typu třídou
 - nahradit magické číslo symbolickou konstantou
 - nahradit podtřídu položkami
 - nahradit vnořenou podmínu varovnými klausulemi
 - nahradit podmínu polymorfismem
 - nahradit kód typu stavem nebo strategií
 - odstranit příznak
 - zachovat celý objekt

Odhadování úsilí

COCOMO (Constructive Cost Model)

Doležité!!! ("Chcem aby ste si toto dali do hlavy") - **Model používaný k odhadování ceny SW. Idea je taková, že cena vývoje aplikace přímo závisí na velikosti SW.** - používá se na odhadování ceny/pracnosti softwaru. - dáva do vztahu velikost zamýšleného produktu(vyjadřujeme v LOC) s úsilím, které bude potřebné na jeho vytvoření (vyjadřujeme v MD nebo MM), díky tomu umíme zjistit i čas, teda kolik kalendářních dní bude potřebných na vytvoření softwaru. - hlavní indikátor velikosti a složitosti SW je LOC - Cena vývoje aplikace přímo závisí na velikosti SW. - Při odhadování pracnosti potřebujeme vědět, jak velký bude ten software - Přesnost odhadu velikosti SW(LOC) závisí na etapě vývoje. - V pozdějších etapách je odhad přesnejší. - Přesnost odhadu se může lišit až čtyřikrát (4:1) oběma směry. 25000 - 400000LOC

Empirické vztahy pro vyjádření $E=E(KSLOC)$ a $T=T(KSLOC)$ - E – effort (práce, člověk-měsíc) - T – doba vývoje (měsíc) - E aj T jsou funkce velikosti zdrojového kódu, závisí na nich

Základní problém při COCOMO je získat dobrou vstupní hodnotu(LOC), to znamená kolik řádků zdrojového kódu to bude.

Odhad sa snažíme zpřesnit i pomocí zkušenosťí s podobnými projekty, ať už se jedná o zkušenosťi programátorov nebo manažerov. Ze zkušenosťí s podobnými technologiemi, vyjadřují se k tomu senioři v týmu, případně manažerí, kteří se potkali s podobným projektem. Hodnoty od různých lidí dáme dohromady a vznikne nám jedna hodnota podobně jako při planning poker.

3 úrovně detailu:

je o presnosti výpočtu, výber záleží na tom, v ktorej fáze projektu som a ako presný chcem mať výpočet Na začiatku nemá zmysel robiť presné výpočty keď nemáme presné data - **Základní model:** hrubý odhad $E(KSLOC)$ a $T(KSLOC)$ založen na odhadu KSLOC. Má zmysel ho robit v úvodných fázach. Jednoduchý vzorec. Je to hrubý odhad. - **Střední model:** vliv jiných faktorů na $E(KSLOC)$ a $T(KSLOC)$. Podobný vzorec ako v základnom modeli, navyše obsahuje korektní faktor, ktorý zohľadňuje ďalšie parametre. Blížime sa k presnejším odhadom, sme v odchylke približne 0,5-1,5xLOC. - **Pokročilý model:** bere v úvahu vlivy vývojové etapy, ve které se projekt nachází. Funguje podobne ako stredný. Dáva zmysel pri väčších projektach, keď máme rôzne skúsenosti na rôznych pozíciah.

3 vývojové módy:

je o tom, ako náročný bude vývoj, kolko tam bude prekážok - **Organický mód** - jednodušší, dobře řešitelné projekty, zpravidla menšího rozsahu($SW < 50$ KSLOC) - úplne porozumenie požiadavkom - malá omezení, volnost při návrhu rozhraní - velké zkušenosťi při zpracování podobných projektů - malá závislost na speciálním HW - minimální potřeba nových algoritmů a architektur - minimum požadavků na zkrácení termínu dodání - příklad: vedecké aplikacie, jednoduchý skladový systém

- **Bezprostřední mód** – středně obtížné projekty
 - ($SW < 300$ KSLOC)

- dobré pochopení požadavků
- zřetelná omezení pro uživatelské rozhraní
- nezanedbatelná zkušenosť pri práci na podobných projektech
- strední závislosť na speciálnom HW
- strední potreba nových algoritmov a architektúr
- nezanedbatelný podnét pre ukončenie pred plánovaným termínem
- príklad: transakčné spracovanie, stredne zložitý skladový systém
- **Vázaný mód** – rozsáhlé projekty s vysokými nároky na řízení
 - projekty kde sa predpokladá veľká chybovost'
 - SW všetkých veľkostí
 - hrubá predstava o cieľoch projektu
 - těsná omezení, strikní požadavky na rozhraní
 - nezanedbatelná zkušenosť pri práci na podobných projektech
 - vysoká závislosť na speciálnom HW
 - extrémnní požadavky na nové algoritmy a architektury
 - vysoké podnety pro dokončení pred termínem
 - príklad: složité transakčné zpracování, složitý operační systém

Úsilí a čas

Výpočet E(KSLOC) - úsilie(effort) T(KSLOC) - time(čas)

$$E(\text{effort}) = a * (\text{KSLOC})^b \quad T(\text{time}) = c * E^d$$

KSLOC odhadnem a,b,c,d: najdeme v tabulkách, parametry volené podle úrovně modelu a vývojového módu, tieto hodnoty boli odvodene štatisticky na základe skúseností a prebehnutých projektoch - hodnoty v tabulkách sú špecifické pre daný model a mód - tabulky hodnot a,b,c,d pro všechny kombinace úrovně modelu/vývojové módy - základní model, bezprostrední mód: a=3.0, b=1.12, c=2.5, d=0.35 - střední model, vázaný mód: a=2.8, Fc, b=1.2, c=2.5, d=0.32

Od středního modelu vyšší se vyskytuje Fc - korekční faktor, spravidla číslo okolo 1. Korekční faktor se skládá z hodnot, které vyjadřují kvalitu tímu, technologií atd.

Intervaly hodnot parametrů: - a ∈ [2.4, 3.6] pro základní model - a ∈ [2.8 Fc, 3.2 Fc] pro střední a pokročilý model - b ∈ [1.05, 1.20] keď sa zväčší projekt, tak pracnosť rastie superlineárne - c = 2.5 ve všech případech - d ∈ [0.32, 0.38]

V základním modelu mají všechny parametry konstantní hodnoty. Od středního modelu vyšší se vyskytuje korekční faktor Fc je součinem hodnot 15 atributů (cost drivers) specifických pro vývojový proces. Sú v ňom zohľadnené aké máme skúsenosti, prostriedky, hw sw kapacity atd. Fc nabýva většinou hodnoty okolo 1. Fc zohľadňuje vyspelosť tímu a prostredia kde bude sw vznikat. Fc je v ideálnom prípade keď je všetko "normálne" 1. Pokial nemame dobre prostredie tak usilie bude vyssie takze Fc bude vyssie ako 1.

Atributy, mohou nabývat 6 možných hodnot ve stupnici: velmi nízký, nízký, normální, velký, velmi velký, extrémně velký

Atributy:

Dôležité!!! 1. **Atributy SW produktu - RELY** - požadovaná spolehlivosť (0.75 - 1.40) (velmi nízká: 0.75, extrémně velká: 1.40) - **DATA** - velikost databáze (0.94 - 1.16) - **CPLX** - složitosť produktu (0.70 - 1.65) (metrika cyklometrická složitosť) **příklad:** vysoké finanční riziko => RELY = velký 2. **HW atributy - TIME** - omezení času výpočtu (1.00 - 1.66) - **STOR** - využití paměti/disku (1.00 - 1.56) - **VIRT** - spolehlivosť (zranitelnost) virtuálních strojů, tj. HW + DBMS + OS + ... (0.87 - 1.30) - **TURN** - doba obrátky (0.87 - 1.15) **příklad:** využití paměti/disku < 50% => STOR = normální 3. **Atributy vývojového tímu - ACAP** - schopnost analytická (1.46 - 0.71) - **PCAP** - schopnost programátorská (1.42 - 0.70) - **AEXP** - zkušenosť s podobnými aplikacemi (1.29 - 0.82) - **VEXP** - zkušenosť se specifickým „virtuálním strojem“ (1.21 - 0.90) - **LEXP** - zkušenosť se specifickým programovacím jazykem (1.14 - 0.95) **příklad:** programátorská schopnosť v týmové práci = 35 (stupnice 0..100) => PCAP = vyssi(mame slabši tim, bude to stat viac usilia) 4. **Atributy projektu - MODP** - použití moderních programovacích technik (1.24 -

0.82) - TOOL - použití SW nástrojů (1.24 - 0.83) - SCED - přesné plánování (1.23 - 1.10)
příklad: občasné použití moderních programovacích technik => MODP = normální

Kroky při použití COCOMO

Určení nominálního úsilí En

- definujte (odhadně) úroveň modelu a vývojový mód
- nastavte odpovídající hodnoty a a b podle tabulky
- vypočtěte En ##### Určení korekčního faktoru F_C
- na základní úrovni není třeba řešit
- určete popisné hodnoty pro každý z 15 atributů
- převeďte na numerické hodnoty podle tabulky

$$F_C = \prod_{i=1}^{15} F_i$$

- výsledok je súčin týchto hodnôt

Určení aktuálního (zpřesněného) úsilí E

- na základní úrovni $E = En$
- E člověk-měsíc = $F_C * En$
- F_C vyjadřuje nárůst pracnosti En podle vlivu a významu jednotlivých atributů vývojového procesu ##### Určení doby vývoje T a dalších faktorů relevantních pro projekt
- nastavte odpovídající hodnoty c a d podle tabulky
- $T[měsíc] = c * E^d$
- COCOMO také umožňuje:
 - výpočet odhadovaných nákladů
 - rozložení práce a ceny v jednotlivých etapách řešení projektu
 - ...

Původní COCOMO

Hodnoty a, b, c, d jsou shodné pro střední a pokročilou úroveň modelu - pro střední úroveň se aplikuje výpočet na celý projekt - pro pokročilou úroveň se výpočet aplikuje pro jednotlivé etapy životního cyklu

COCOMO lze také použít pro odhad nákladů při modifikaci existujících aplikací

$$\text{ESLOC} = \text{ASLOC} \cdot (0,4 \text{ DM} + 0,3 \text{ CM} + 0,3 \text{ IM}) / 100$$

ESLOC - ekvivalentní počet SLOC ASLOC - odhadnutý počet modifikovaných SLOC DM - procento modifikace v návrhu CM - procento modifikace v kódu IM - integrační úsilí (procento původní práce)

Existují různé verze COCOMO pro různé účely a prostředí

COCOMO 2

Nie až tak dôležité, podstatné je rozumieť principu původnému COCOMO - nové softwarové procesy - nové jevy měření velikostí - nové jevy znovupoužití software - potřeba rozhodování na základě neúplné informace

3 různé modely, každý s jinou přesností.

- **ACM (Aplication Composition Model)** pro projekty s použitím moderních nástrojů a GUI, používame keď vieme ako budú asi vyžerať obrazovky a aké to asi bude veľké
- **EDM (Early Design Model)** pro hrubé odhady v úvodních etapách, kdy se architektura vyvíjí, používame keď máme hotový už nejaký prvotný návrh
- **PAM (Post Architecture Model)** pro odhady poté, co byla specifikována architektura

Předběžný návrh a Post-architektura

Úsilí = (multiplikátory okolí)^[velikost](faktory procesu)^[faktory procesu] Plán = (multiplikátor)^[Úsilí](faktory procesu)^[faktory procesu] - Okolí: výrobek, platforma, lidé, faktory projektu - Místo nasazení: nelineární znovupoužití a proměnlivost - Proces: omezení, riziko/architektura, tým, faktory vyspělosti

ED a PA modely

(ED-Early Design model, PA-Post-Arch model, PM - PersonMonth)

$$PM_{estimated} = A \times (Size)^{(SF)} \times \left(\prod_i EM_i \right)$$

Úsilí

několika přístupy - KSLOC (tis. rádek zdroj. kódů) - UFP (neupravené fukční body) - EKSLOC (ekvivalentní velikost zdroj. kódů) - SF: měřítkové faktory určené pomocí driverů exponentu - EM: multiplikátory úsilí (7 pro ED, 17 PA)

Nový přístup k měřítk. exponentu

$$PM_{estimated} = A \times (Size)^{(SF)} \times \left(\prod_i EM_i \right)$$
$$SF = 1.01 + 0.01 \sum (\text{hodnocení driverů exponentu})$$

A v rozsahu 1.01 - 1.26 SF - upravený součet 5 driverů s hodnocením 0 – 5

Drivery exponentu - návaznost na předchozí výsledky - flexibilita vývoje - rozhodnutí architektury/rizika - koheze týmu - vyspělost procesu (podle SEI CMM)

Nové atributy ovlivňující EM

- RUSE - požadovaný stupeň znovupoužitelnosti
- DOCU - souběžná úprava dokumentace při vývoji
- RCPX - složitost a spolehlivost produktu
- VMVH - proměnlivost virtuálního stroje – host
- VMVT - proměnlivost virtuálního stroje – periferie
- PVOL - proměnlivost HW platformy
- Pdif - složitost HW platformy
- PERS - personální schopnosti
- PREX - personální zkušenosti
- PCON - personální kontinuita na projektu
- PEXP - zkušenosť s platformou
- LTEX - zkušenosť s jazykem a nástroji
- SECU - bezpečnost
- SITE - vývoj ve více místech

Většina nových atributů vychází z kombinace dříve používaných atributů. Nové atributy mají 6 možných hodnocení, každému hodnocení odpovídá kladné číslo určené kalibrací z předchozích projektů

2 společné vlastnosti C a C2

- Oba způsoby při odhadu ceny zahrnují jistou množinu faktorů, která ji ovlivňuje
- Oba využívají stejný druh modelů na rozlišení výpočtu #### 3 rozdíly C a C2
- COCOMO 2 zahrnuje některé nové atributy na měření odhadu ceny, které vznikly kombinací předchozích z COCOMO.
- Modely v COCOMO 2 jsou na rozdíl od COCOMO zaměřené spíše na vývojovou etapu projektu
- Při odhadu nákladů na úpravu aplikace využívá COCOMO 2 i tzv. AA a SU koeficienty

Rozšírené a upravené modely

Dôležitý rozdiel medzi COCOMO a COCOMO 2 ESLOC = ASLOC . (AA+SU+0,4 DM+0,3 CM+0,3 IM)/100 - ESLOC, ASLOC, DM, CM, IM - stejné ako dříve - AA (Assessment and Assimilation) - práce potrebná pro určení, zda a v jakém rozsahu môže být existujúci modul použit bez změn, Je potrebné sa v danom systéme najprv zorientovať než ho začneme meniť nech vieme vlastne čo chceme/môžeme meniť - SU (Software understanding - pochopení SW) = čítelnosť a „uchopenie“ - to znamená ako dobré je daný kód čitateľný, či je to dobre zdokumentované, či kód obsahuje komentáre

Funkční body (FP) = normalizovaná metrika softwarového projektu

Dôležité!!! - Měří aplikační oblast, nezkoumá technickou oblast - Měří aplikační funkce a data, neměří kód - Zaměřuje se na pohled ze strany uživatele - Oproti COCOMO, obourává závislost na programátora a programovacím jazyku - Je to taky jednotka velikosti, která měří software ##### Princip - Předběžný odhad s použitím omezené informace - Měří vstupy, výstupy, dotazy, vnitřní paměti, vnější paměti - Je to jednotka výroby, cena práce za výrobu takové jednotky - Oproti COCOMO jsme schopní spočítat funkční body s poměrně vysokou přesností už ve fáze analýzy

Princip odhadu: velikost projektu(Fp) X složitost X rizikové faktory = odhad

Typy funkčních bodů

Dôležité!!! Funkční body vztažené k transakčním funkcím: - **Externí vstupy (EI - External Inputs)** - **Externí výstupy (EO - External Outputs)** - **Externí dotazy (EQ - External Enquiry)**

Funkční body vztažené k datovým funkcím: - **Vnitřní logické soubory (ILF - Internal Logical Files)** - dátá ktoré si systém uchováva sám. Je to veľká logická skupina užívateľských dát použitých pre riadene aplikácie. Započítavame každé logické zoskupenie z pohľadu používateľa. Nebudeme započítavať súbory ktoré sú tam z pohľadu technológie ale užívateľ o nich ani nevie. - **Soubory vnějšího rozhraní (EIF - External Interface Files)** - dátá ktoré si systém neuchováva a nespravuje sám ale pristupuje k nim prostredníctvom integrácie iného systému. Dátá sú uchovávané a spravované inou aplikáciou.

Externí vstupy (EI)

V SQL si to môžeme predstaviť ako UPDATE/DELETE/INSERT. Napríklad formulár kde užívateľ zadáva dátá. Započteme každá unikátní uživatelská data nebo zadání uživatelských povelů, ktorá vstoupí pries externí rozhraní do aplikace a přidá, mění, ruší nebo jinak pozmění data (např. přiřazení, přemístění, ...) v interním logickém souboru.

Započteme také řídící informaci, která vstoupí pries aplikační hranici a zajistí soulad s funkcí specifikovanou uživatelem.

Externí vstup by měl být považován za unikátní, pokud logický návrh vyžaduje logiku zpracování odlišnou od ostatních externích vstupů.

1. Datová obrazovka s přidáním, změnou a rušením (3 EI)
2. Více obrazovek pohromadě zpracovaných jako jedna transakce (1 EI)
3. Dvě datové obrazovky s odlišným uspořádáním dat, ale se shodnou logikou zpracování (1 EI)
4. Dvě datové obrazovky se shodným formátem, ale s odlišnou logikou zpracování (2 EI)
5. Datová obrazovka s více unikátními funkcemi (1 EI za každou funkcí)
6. Automatický vstup dat nebo transakcí z jiné aplikace (1 EI na každý typ transakce)
7. Vstup uživatelských povelů do aplikace (1 EI)
8. Vstupní formuláře (OCR) s jednou transakcí (1 EI)
9. Funkce úpravy dat, která následuje za dotazem (1 EI a 1 EQ)
10. Individuální výběry na obrazovce s menu (0 EI)
11. Oprava uživatelem udržované tabulky nebo souboru (1 EI)
12. Duplikát obrazovky, která již byla započtena jako vstup (0 EI)
13. Externí vstupy zavedené pouze kvůli technologii (0 EI)

14. Výběr položky ze seznamu (0 EI)

Externí výstupy (EO)

V SQL si to môžeme predstaviť ako SELECT. Prečíta sa niečo z databázy a zobrazí sa to v aplikácii. Napríklad vyhľadávanie, nie je to vstup pretože sa použije len ako filter na základe ktorého sa mi zobrazia určité dátá. Výstup teda môže byť zoznam položiek na základe daného filtra. Započteme každá unikátní uživatelská data nebo řídící data, která opouští externí hranici měřeného systému.

Externí výstup je považován za unikátní, pokud má odlišná data, nebo pokud vnější návrh (jiná aplikace) vyžaduje odlišnou logiku zpracování oproti jiným externím výstupům.

Externí výstupy se často skládají z hlášení, výstupních souborů zasílaných jiné aplikaci nebo zpráv pro uživatele.

1. Výstup dat na obrazovku (1 EO)
2. Souhrnná zpráva - dávkové zpracování (1 EO)
3. Automatická data nebo transakce směrem k jiným aplikacím (1 EO)
4. Chybové zprávy vrácené jako výsledek vstupní transakce (0 EO)
5. Záložní soubory (0 EO)
6. Výstup na obrazovku a na tiskárnu (2 EO)
7. Výstupní soubory vytvořené z technických důvodů (0 EO)
8. Výstup sloupcového a zároveň koláčového grafu (2 EO)
9. Dotaz s vypočtenou informací (1 EO, 0 EQ)

Externí dotazy (EQ)

Jako vnější dotaz započti každou unikátní vstupní/výstupní kombinaci, kde vstup je příčinou a generuje výstup. Napríklad zadám určité údaje na vstup ktoré sa uložia do databázy a následne sa zobrazí nejaký výstup z databázy. Vnější dotaz je považován za unikátní, pokud se od ostatních dotazů odlišuje typem výstupních datových elementů, nebo pokud vyžaduje odlišnou logiku zpracování v porovnání s ostatními externími dotazy.

1. On-line vstup a on-line výstup beze změny v datových souborech (1 EQ)
2. Dotaz následovaný změnovým vstupem (1 EQ/1 EI)
3. Vstup a výstup na obrazovce s nápovedou (na dané úrovni) (1 EQ)
4. On-line vstup s bezprostredním tiskem dat beze změny dat (1 EQ)
5. Výběr ze seznamu nebo umístění s dynamickými daty (1 EQ)
6. Výběr ze seznamu nebo umístění se statickými daty (0 EQ)
7. Požadavek na zprávu obsahující neodvozená data (1 EQ)

Interní logické soubory (ILF)

Každá velká logická skupina uživatelských dat nebo informací použitych pro řízení aplikace představuje jeden ILF.

Zahrneme každý logický soubor, nebo v případě DB, každé logické seskupení dat z pohledu uživatele, které je vytvořeno, používáno, nebo udržováno aplikací.

Spíše než fyzické soubory započteme každé logické seskupení dat tak, jak je viděno z pohledu uživatele a jak je definováno při analýze požadavků nebo návrhu dat.

Nezapočteme soubory, které nejsou přístupné uživateli prostřednictvím vnějšího výstupu nebo dotazu a které nejsou nezávisle udržovány.

1. Logická entita nebo skupina entit z pohledu uživatele. (1 ILF)
2. Logický interní soubor generovaný nebo udržovaný aplikací. (1 ILF)
3. Uživatelem udržovaná tabulka(y) nebo soubor(y). (1 ILF)
4. Datový soubor nebo soubor s řídící informací, který aplikace použije při sekvenčním zpracování a údržbě. (1 ILF)
5. Atributová entita udržovaná pouze prostřednictvím hlavní entity. (0 ILF)
6. Asociativní entity vytvořené průnikem nebo spojením obsahující pouze klíčový atribut (0 ILF)
7. Přechodný nebo trídící soubor (dočasný soubor) (0 ILF)

8. Soubor vytvořený proto, že byla použita určitá technologie (např. indexový soubor) (0 ILF)
9. Soubor s předlohou (vzorem), který aplikace pouze čte. (0 ILF, 1 EIF)

Soubory externího rozhraní (EIF)

Započteme každou velkou logickou skupinu uživatelských dat nebo řídící informace používané aplikací.

Tato informace musí být udržována jinou aplikací. Zahrňte každý logický soubor nebo logickou skupinu dat z pohledu uživatele.

Započteme každou velkou logickou skupinu uživatelských dat nebo řídící informace, která je extrahována aplikací z jiné aplikace ve formě souboru externího rozhraní.

Extrakce nemá mít za následek změnu v některém z interních logických souborů. Pokud ano, pak započteme do EI místo do EIF.

1. Soubory nebo záznamy extrahované z jiné aplikace (použité pouze jako odkazy) (1 EIF)
2. Databáze čtená pomocí jiné aplikace (1 EIF)
3. Vnitřní logický soubor jiné aplikace použitý jako transakce (0 EIF, 1 EI)
4. Systém HELP, bezpečnostní soubor, chybový soubor čtený nebo odkazovaný aplikací, který pochází z jiné aplikace, která soubory udržuje (2 EIF)

Výpočet funkčních bodů

Před výpočtem musíme EI, EO, EQ, ILF, EIF roztržit do skupin podle vah.

Váhy	nízká	průměrná	vysoká	celkem
EI	___ x 3+	___ x 4 +	___ x 6 =	___
EO	___ x 4 +	___ x 5 +	___ x 7 =	___
EQ	___ x 3 +	___ x 4 +	___ x 6 =	___
ILF	___ x 7 +	___ x 10 +	___ x 15 =	___
EIF	___ x 5 +	___ x 7 +	___ x 10 =	___

Neupravené funkční body celkem ___

Váhy určujeme na základě Matice zložitosti.

Matice složitosti

- FTR = File Types (User Data Groups) Referenced - počet dotknutých záznamov/tabuľiek/entít
- DET = Data Element Type (Attribute) - počet atribútov
- RET = Record Element Type (User View)

Matice složitosti vstupů (EI, EQ)

FTRs	1-4 DETs	5-15 DETs	16+DETs
0-1	nízká	nízká	průměrná
2-3	nízká	průměrná	vysoká
4+	průměrná	vysoká	vysoká

Matice složitosti výstupů (EO, EQ)

Rovnaka ako Matice složitosti vstupů

Matice složitosti souborů (ILF, EIF)

REIs	1-19 DEIs	20-50 DEIs	51+ DEIs
1	nízká	nízká	průměrná
2-4	nízká	průměrná	vysoká
5+	průměrná	vysoká	vysoká

Obecné charakteristiky systému

14 charakteristik hodnocených podle stupně vlivu na aplikaci(rychlosť, bezpečnosť, objemy dat...) Každý faktor je hodnocený ve stupni 0 – 5 takto: - 0 = bez vlivu - 1 = náhodný - 2 = mírný - 3 = průměrný - 4 = významný - 5 = podstatný

1. Vyžaduje systém spolehlivé zálohování a zotavení?
2. Jsou vyžadovány datové komunikace?
3. Existuje distribuované zpracování?
4. Je výkonnost kritická?
5. Poběží systém v stávajícím intenzivně využívaném operačním prostředí?
6. Systém požaduje on-line vstup dat?
7. Využaduje on-line vstup dat použití vstupní transakce přes více obrazovek nebo operací?
8. Jsou hlavní soubory opravovány on-line?
9. Jsou vstupy, výstupy, soubory a dotazy složité?
10. Je vnitřní zpracování složité?
11. Je kód navrhován s cílem znovupoužití?
12. Jsou konverze a instalace zahrnuty v návrhu?
13. Je systém navrhován pro násobné instalace u různých organizací?
14. Je aplikace navrhovaná tak, aby zajistila změny a snadné používání na straně uživatele?

Součtem dostaneme faktor technické složitosti

Počet funkčních bodů

Počet funkčních bodů = $[0.65 + (0.01 \times \text{Součet hodnocení charakteristik systému})] \times [\text{počet neupravených funkčních bodů}]$

Výsledné číslo hovorí o tom aká je veľkosť z pohľadu funkcionality, čo ten systém dokáže.

Nové a upravované projekty

Type of Project	Project Function Points	Application Function Points
Development Project	Project FP = New (Added) FP + Conversion FP	Installed Function Pts. (IFP) Application FP = New (Added) FP
Enhancement Project	Project FP = Added FP + Changed FP + Deleted FP + Conversion FP	Application FP = Original FP - Deleted FP + Added FP + Δ Changed FP

Funkčné body produktu vs. Funkčné body procesu(projektové) - funkčné body produktu vyjadrujú množstvo funkčných bodov daného SW - funkčné body procesu vyjadrujú množstvo úkonov ktoré boli vykonané počas daného procesu(mazanie kódu,

úprava, tvorba nového, prerábky kódu)

Postup výpočtu FP

1. Identifikujte a spočtěte ILF, EIF, EI , EO, EQ. Pro každou ILF a EIF identifikujte počet RET a počet DET. Pro každou EI, EO a EQ, identifikujte počet FTR a DET
2. S použitím matic složitosti spočtěte počet jednoduchých, průměrných a složitých položek EI, EO, EQ, ILF, EIF.
3. Spočtěte Počet neupravených funkčních bodů.
4. Určete hodnoty 14 charakteristik systému.
5. Sečtěte charakteristiky a určete Faktor technické složitosti systému.
6. Určete Počet upravených FP systému.

Odhady velikosti (Capers Jones)

1 Funkční bod = X příkazů - získame počet LOC - 320 - základní assembler - 213 - makro assembler - 128 - C - 107 - COBOL - 107 - FORTRAN - 80 - PL/I - 71 - Ada 83 - 64 - C++ - 54 - Ada 95 - 32 - Visual BASIC - 22 - Smalltalk - 16 - PowerBuilder - 13 - SQL

Tabuľka vyššie ukazuje aj to, že voľba programovacieho jazyka má vplyv na pracnosť v COCOMO. Prepočet Fp na LOC koreluje od približne 2000LOC. Pri 2000LOC koreluje aj doba projektu vypočítaná z COCOMO a Fp

Další odhad

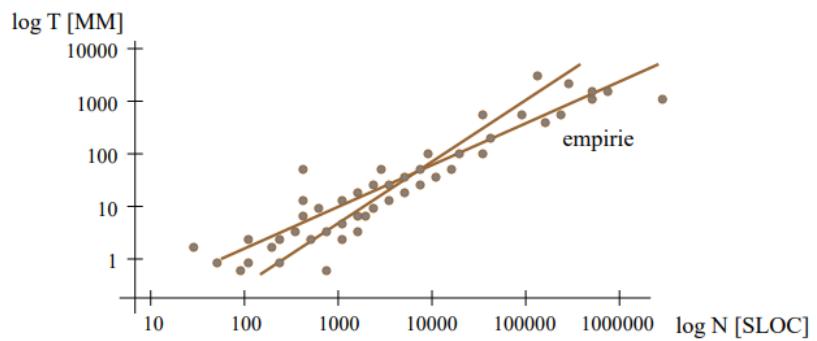
- FP^{1.15} predpovídá približný počet stran papírové dokumentace SW projektu
- FP^{1.2} predpovídá približný počet vytvorených testovacích prípadů
- FP^{1.25} predpovídá približný chybový potenciál u nových SW projektov
- FP^{0.4} predpovídá približný plán vývoje v kalendárnych mesiacích
- FP / 150 predpovídá približný počet pracovníkov potrebných pro řešení aplikace
- FP / 750 predpovídá približný počet pracovníkov údržby, kteří budou udržovat aplikaci v aktuálně požadovaném stavu

Približné velikosti aplikací

- Vstup objednávky 1,250 FP
- Zpracování daňového přiznání 2,000 FP
- Účtování telefónních služeb 11,000 FP
- Rezervace letenek 25,000 FP
- OS Windows 95 85,000 FP
- Telefonní přepojovací systém 12,000 FP

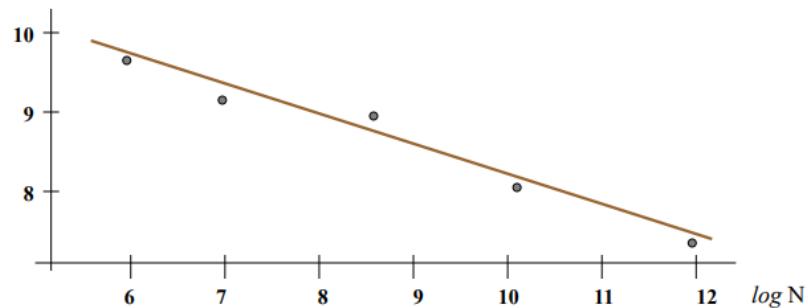
Softvérová fyzika

Vyjádření vztahů mezi základními veličinami (čas, pracnost, velikost FP nebo LOC (tyto hodnoty spolu korelují takže je to v podstatě skoro jedno) softwaru, kvalita) v softwarovém inženýrství. N - délka programu (počet řádek, SLOC) T - spotřeba práce (člověkoměsíce, MM) P - produktivita P=N/T D - doba realizace programu S - průměrný počet řešitelů ##### Práce a délka programu S množstvom riadkov kódov rastie pracnosť (tažie sa v ňom zorientovať, väčší tím, viac komunikácie)



Produktivita

$\log P$ [řádky/rok]



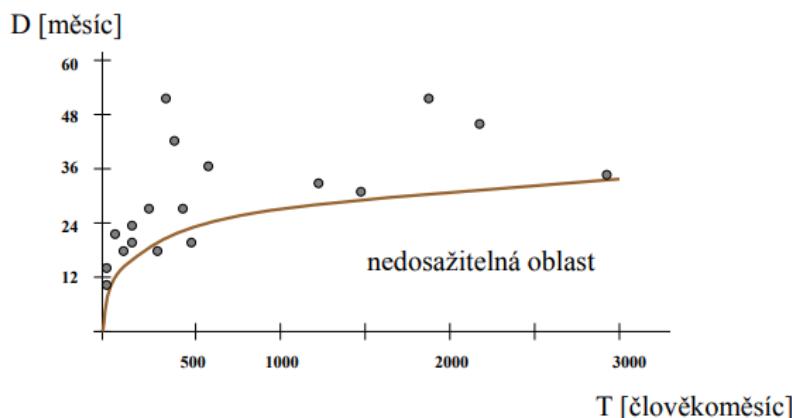
S rostoucí délkou programu klesá produktivita programátorů. ### Putnamova rovnica
Dôležité!!! Z Putnamovej rovnice je odvodený model COCOMO. Ukazuje balance v trojuholníku veľkosť SW, čas a pracnosť(pracnosť=cena-> je tam priama úmiera).

$$N \hat{=} c T^{1/3} D^{4/3}$$

dáva do vzťahu dĺžku programu, prácu a dobu riešenia

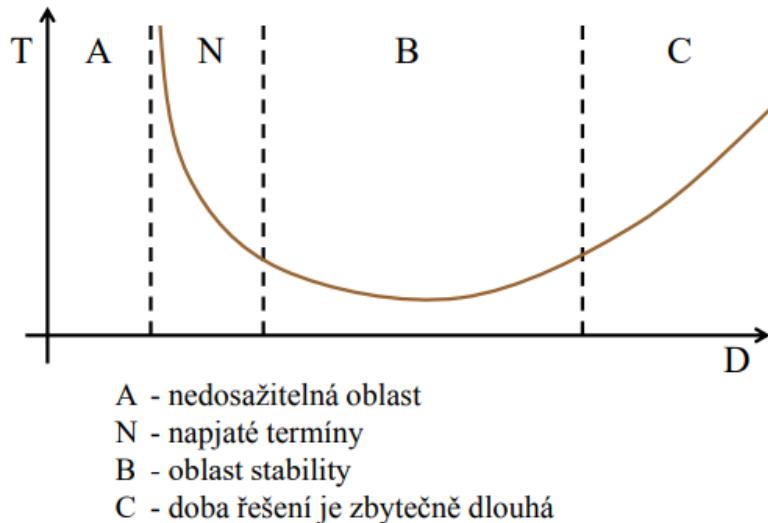
c - konštantá, vyjadrujúca produktivitu firmy(aké nástroje používa, ako skúsený máte tím, ako kvalitných majú programátorov... pozn. je možné pripojiť ku korekčnému faktoru v COCOMO) čím je team lepší a šikovnejší, tím je konštantá nízsia N - dĺžka programu T - práca D - doba riešenia

Dôsledky Putnamovy rovnice - Programy psané ve spěchu jsou delší. - Při zkrácení termínu na 83% je pracnosť dvojnásobná.



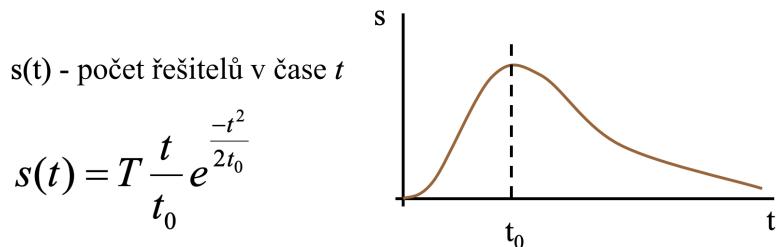
- pridaním programátora do rozbehnutého projektu nezrýchlime dobu dodania ale skôr spomalíme pretože je potrebné ho zaškoliť atď. - nedosažitelná oblasť - je to znázornenie

toho, že aj keby dám akékoľvek množstvo programátorov na projekt, tak ho nie je možné splniť pod určitý čas



Rozloženie riešiteľskej kapacity v čase. Dôležité!!!

Model rozložení aproximovaný pomocí vlny $s(t)$ (Rayleigh)



Celková spotřeba práce T :

$$T = \int_0^\infty s(t) dt$$

Softverový projekt je špecifický tím, že na ňom pracuje najviac ľudí približne v prvej polovici projektu (optimálny vrchol je zhruba 40%(t_0) času(t)), počas ktorej sa ale minie väčšina peňazí. Časom bude ľudí na projekte ubúdať kvôli tomu, že väčšina vecí je už hotová a začína sa s testovaním, ľadením chýb a nasádzaním. To znamená, že keď je približne 40% práce dokončenej, tak sa na projekte nachádza najväčšie množstvo ľudí, následne toto množstvo zasčne klesať. Pri HW je najväčšie množstvo ľudí na projekte skôru koncu projektu. Na konci projektu je dôležitá správa o záverečnej analýze hlavne z pohľadu projektového manažmentu aby sme sa mohli poučiť do ďalších projektov a zmapovať čo ako v skutočnosti vyzeralo (ako dopadlo nacenovanie, odhady, na čo sme zabudli a vypomstilo sa to, ako vyzerali meetingy, ako sa utrácal budget, aká práca bola vykonaná, aká je velkosť produktu, zhrnutie výkonu tímu)

Údržba a znovupoužitelnost

Údržba je modifikace SW produktu po predání zákazníkovi za účelem opravy chyb, zvýšení výkonnosti a prízpôsobení menícimu se okolí. Najčastejšie upravujeme software kvôli užívateľským novým/zmenovým požiadavkám. Údržba/servis projektu je veľmi nákladný a finančne zaujímavý pre poskytovateľa servisu.

Údržba je spíše u zakázkového SW. Snaží se reagovat na chyby a přizpůsobovat se měnímu se prostředí. Velmi nákladná v čase, finance odpovídají tomu jak dlouho ho chceme udržet. Náklady jsou 2x-100x větší než na vývoj. Na údržbu se mělo myslit již při vývoji.

- Oprava softwarových chyb
- Adaptace softwaru na jiné operační prostředí
- Přidat nebo měnit systémovou funkcionalitu → každá implementace degraduje kvalitu systému

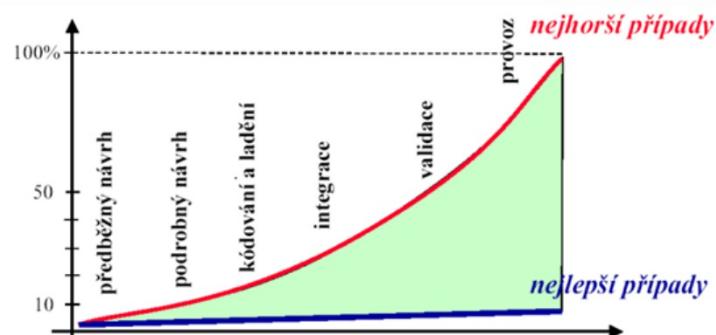
Faktory ceny údržby

- Stabilita týmu
- Smluvní zodpovědnost
- Um zaměstanců
- Věk a struktura programu

Vývoj relativní ceny údržby v závislosti na kvalitě produktu

Cena se odvíjí od toho jak dlouho se to udržuje a kolik je provedeno zákroků do systému. Do ceny zahrnuje také incident management, konzultace, změna/přidání funkcionality a školení.

Pokiaľ je softver kvalitne implementovaný, je cena údržby nižšia(napríklad modrá čiara). Na osi ypsilon vidíme percutálne vyjadrenie ceny základného softveru. To znamená že ak je sw naprogramovaný zle, tak údržba po istom čase bude rovnaká drahá ako vývoj daného sw. Zákazník zaplatí za sw a pokiaľ nie je dobre servisovateľný tak v priebehu 10 let zaplatí za servis trojnásobok.



Pokud si dobré rozmyslíte strukturu nového SW a vše uděláte „pořádně“, bude vývoj sice o něco dražší, ale vynaložené náklady se vrátí v období provozu SW systému, kdy děláte jeho údržbu a na přání zákazníka provádít modifikace.

Znovupoužitelnost

Hlavní výhodou je několikanásobné finanční ohodnocení jednou vyvinuté komponenty.
 ##### Úrovň znovupoužitnosti (reuse levels) - Abstrakce (The abstraction level):
 analytické prvky - Objekty (The object level): třídy - Komponenty (The component level):
 kolekce tříd - Systém (The system level): celý systém a různé platformy

Reuse-Oriented software engineering

Model vývoje SW, založen na systematickém znovuužití, kde jsou systémy integrovány z existujících komponent (commercial-off-the-shelf = COTS) - Zbylou funkcionalitu (kterou ještě nemáme) doprogramujme, především proxy, adaptéry a GUI. - Jedná se o standardní budování business systémů.

Lehmanovy „zákon“ (1980, 1985)

Dôležité!!!

Zákony se zabývají fází evoluce, popisují rovnováhu mezi novými požadavky a údržbou na straně jedné a zvyšující se složitostí, snižující se "business value" na straně druhé.

- **Z trvalé proměny:** Systém používaný v reálném prostředí se neustále mění, dokud není levnější systém restrukturalizovat, nebo nahradit zcela novou verzí.
- **Z rostoucí složitosti:** Při evolučních změnách je program stále méně strukturovaný a vzrůstá jeho vnitřní složitost. Odstranění narůstající složitosti vyžaduje dodatečné úsilí.
- **Z vývoje programu:** Rychlosť změn globálních atributů systému se může jevit v omezeném časovém intervalu jako náhodná. V dlouhodobém pohledu se však jedná o seberregulující proces, který lze statisticky sledovat a předvídat.
- **Z invariantní spotřeby práce:** Celkový pokrok při vývoji projektů je statisticky invariantní. Jinak řečeno, rychlosť vývoje programu je přibližně konstantní a nekoreluje s vynaloženými prostředky.
- **Z omezené velikosti přírůstku:** Systém určuje přípustnou velikost přírůstku v nových verzích. Pokud je limita překročena, objeví se závažné problémy týkající se kvality a použitelnosti systému.

Brooksův zákon

Přidání řešitelské kapacity u opožděného projektu může zvětšit jeho zpoždění. (Náklady na začlenění nového pracovníka do týmu jsou zpravidla větší než jeho přínos)

Kvalita softwaru

Vychází z požadavků z očekávání a ze standardů, dělí se na tři kategorie faktorů: -
Operation - činnost produktu - **Revision** - revize produktu - **Transition** - přemístění produktu - a k tomu konkrétní faktory, které tam patří - pozn. konkrétní faktory jsou zobrazené na obrázku pod nadpisem McCall... Quality model a McCall zobrazují to isté

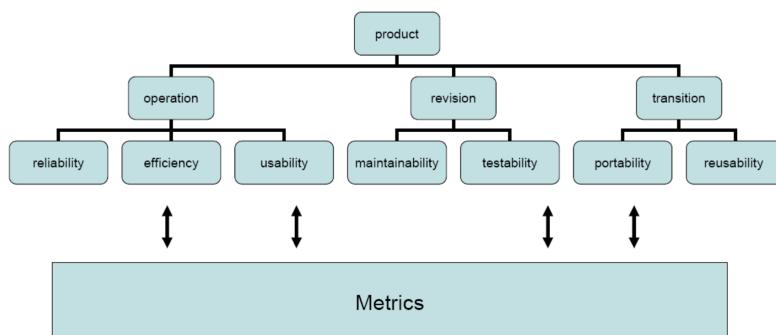
Každý program dělá něco správně; nemusí však dělat to, co chceme, aby dělal.

Dôležité!!! Presne túto definíciu **Kvalita:** Dopržení explicitně stanovených funkčních a výkonových požadavků, dopržení explicitně dokumentovaných vývojových standardů a implicitních charakteristik, které jsou očekávány u profesionálně vyrobeného software.

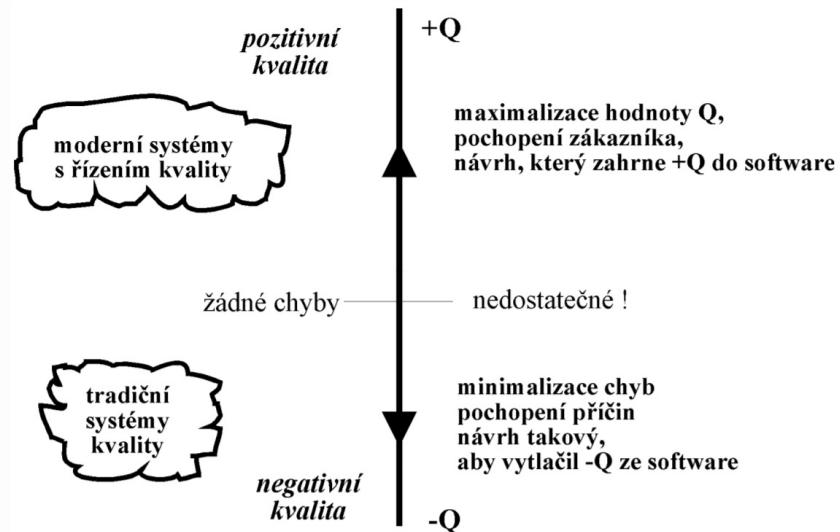
Dôležité!!! **Aspekty kvality:** - **odchyly od požadavků na software** - to, co je v požadavcích se nepřetaví do pozdější fáze vývoje SW systému - **nedopržení standardů** - standard při řízení projektu, standardy a best practices při programování, standard při dokumentaci - **odchyly od běžných zvyklostí a implicitních požadavků** - např. okno v prohlížeči neobsahuje close button

Extrémne dôležitý obrázok!!!

Quality Model



Spojité chápání kvality



Dôležité!!! #### Kvalita - IEEE Std. 610.12-1990 Stupeň, do jaké míry systém, komponenta nebo proces splňuje specifikované požadavky.

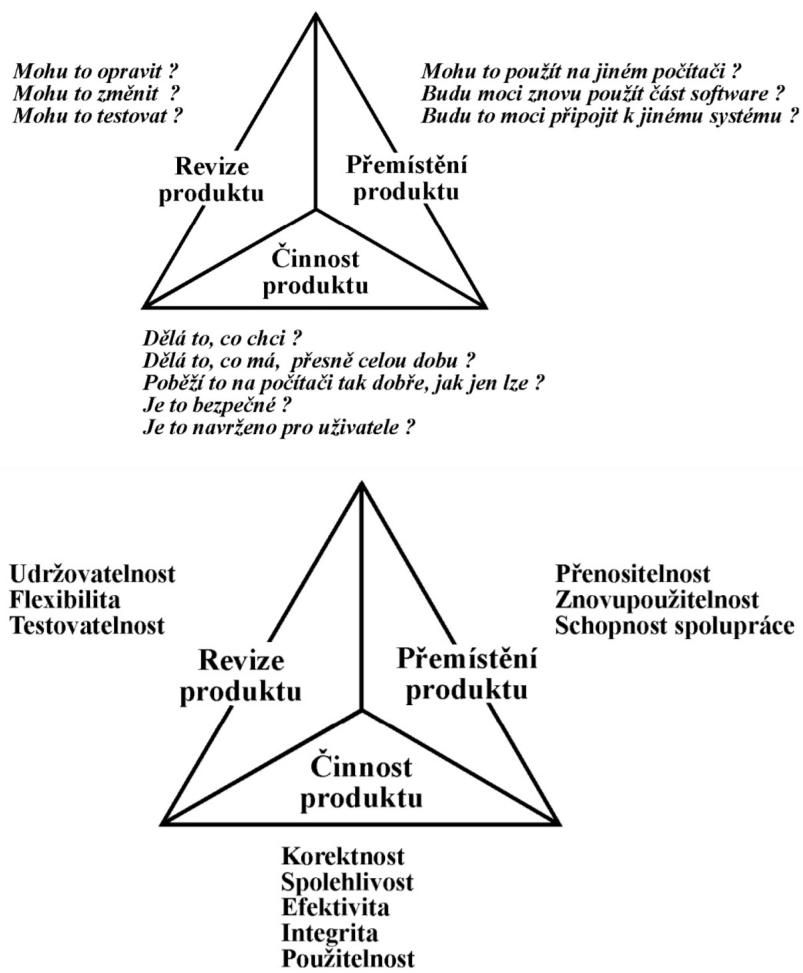
Stupeň, do jaké míry systém, komponenta nebo proces splňuje zákazníkovy nebo uživatelsovy potřeby nebo jeho očekávání.

Faktory kvality software

Dôležité!!! - Přímo měřitelné faktory - #chyb/KLOC/čas - Pouze nepřímo měřitelné faktory - použitelnost, udržovatelnost - Kategorie faktorů kvality: Dôležité!!! Toto sú jednotlivé položky z obrazka "Quality Model" - operační charakteristiky(operation) - funkčnosť, či sú tam chyby alebo nie príklad metriky: počet chyb na počet FP - schopnosť akceptovať zmény(revision) - revízia, testovateľnosť príklad metriky: cyklotestovanie - adaptibilita na nové prostredí(transition) - prenositeľnosť na iné prostredie, znovupoužiteľnosť príklad metriky: dokumentácia na počet LOC

Faktory kvality - McCall et al. (1977)

Dôležité!!! Vymenovať kategórie faktorov kvality(operation, revision, transition) a čoho sa týkajú, uviesť 2 príklady ku každému faktoru - **Korektnosť**: Rozsah toho, jak program splňuje specifikaci splňuje uživatelsovy zámery. - **Spolehlivosť**: V jakém rozsahu lze očekávať, že program bude plniť zamýšlené funkce s požadovanou presnosťí. - **Efektivita**: Množstvo výpočetných prostriedkov a kódu, ktoré program potrebuje na splnenie svých funkcií. - **Integrita**: V jakém rozsahu môžu byť program nebo data používána neoprávněnými osobami. - **Použitelnosť**: Úsilí vyžadované na učení, operovanie, prípravu vstupu a interpretáciu výstupu programu. - **Udržovateľnosť**: Úsilí vyžadované na vyhľadanie a opravu chyby v programu. - **Flexibilita**: Úsilí vyžadované na modifikáciu provozovaného programu. - **Testovateľnosť**: Úsilí potrebné na testovanie programu tak, aby sa uistili, že plní zamýšlené funkcie. - **Prenositelnosť**: Úsilí potrebné na prenosenie programu na iný HW/SW. - **Znovupoužiteľnosť**: Rozsah, v jakém lze program nebo jeho časti znova použiť v iné aplikaci (funkce a balení produktu). - **Schopnosť spolupráce**: Úsilí, ktoré je nutné vynaložiť pri pripojení daného systému k inému.



Globální hodnocení kvality výroby

Vyspělost organizace: model CMM Systémy kvality: norma ISO 9001 **Ocenění kvality:** cena MBNQA

CMM- Capability Maturity Model

Hodnotí vyspělost organizací podle stupně a kvality využívání SW procesů.

Úroveň 1: Výchozí

Chaotický proces, nepředvídatelná cena, plán a kvalita.

Úroveň 2: Opakovatelný

Intuitivní; cena a kvalita jsou vysoce proměnlivé, plán je pod vědomou kontrolou, neformální metody a procedury. Klíčové prvky: - řízené požadavky - plánování softwarového projektu - řízené subkontrakty na software - zajištění kvality software - řízení softwarových konfigurací

Úroveň 3: Definovaný

Orientován na kvalitu; spolehlivé ceny a plány, zlepšující se, ale dosud nepředvídatelný přínos (výkon) systému kvality. Klíčové prvky: - zlepšování organizačního procesu -

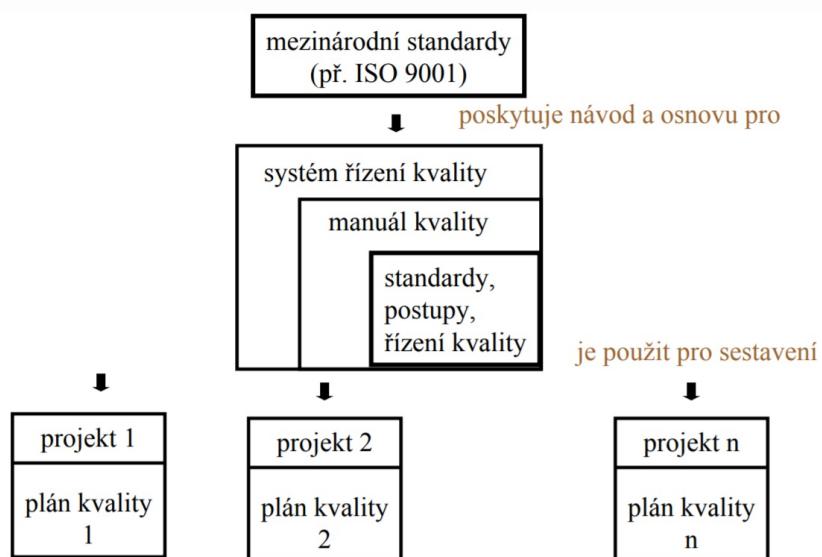
definice organizačního procesu - školicí program - řízení integrovaného software - aplikace inženýrských metod u softwarového produktu - koordinace mezi pracovními skupinami - detailní prověrky a oponentury

Úroveň 4: Řízený

Kvantitativní; promyšlená statisticky řízená kvalita produktu. Klíčové prvky: - měření a kvantitativní řízení procesu výroby - řízení kvality

Úroveň 5: Optimalizující

Kvantitativní základ pro kontinuální investice směřující k automatizaci a zlepšení výrobního procesu. Klíčové prvky: - prevence chyb - inovace technologie - řízené změny výrobních procesů



Principy systémů SQA (Software Quality Assurance)

- Definovaná a dokumentovaná politika kvality a manažerský podíl
- Definice zodpovědností, autorit a vztahů mezi všemi osobami, které svojí prací mohou ovlivnit kvalitu
- Dokumentované postupy a instrukce pro kvalitu
- Efektivní implementace dokumentovaného systému kvality na všech úrovních organizace
- Záznam všech aktivit SQA

Standardy ISO 900x

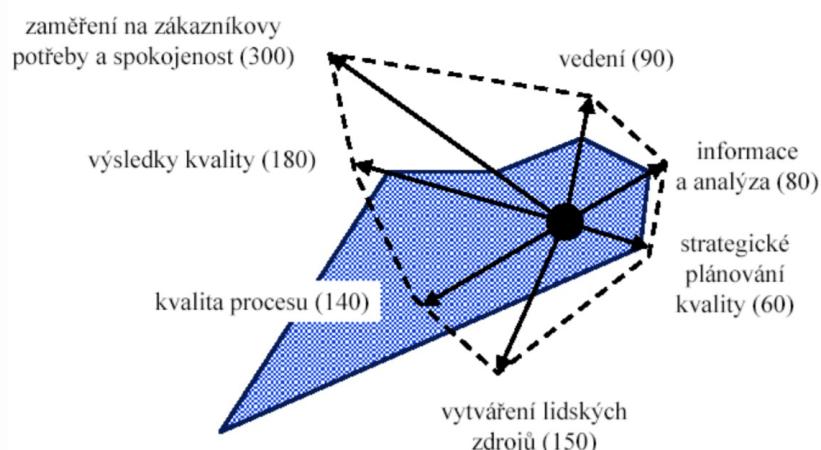
- **ISO 9001: 1994**
 - Systémy kvality - model zajištění kvality při návrhu, vývoji, výrobě, instalaci a servisu
- **ISO 9000 - 3: 1991**
 - Doporučení, jak aplikovat ISO 9001 při vývoji SW
- **ISO 9004 - 2: 1994**
 - Řízení kvality a prvky systému kvality - doporučení

ISO 9001 - Systémy kvality

1. Zodpovědnost vedení
2. Systém kvality
3. Přehled zakázek
4. Řízení návrhu

5. Řízení dokumentace
6. Nakupování
7. Zakoupené produkty
8. Identifikace a sledování produktu
9. Řízení procesu
10. Inspekce a testování
11. Inspekční, měřicí a testovací vybavení
12. Stav inspekce a testování
13. Zvládnutí nevyhovujícího produktu
14. Opravné akce
15. Manipulace, skladování, balení a doručení
16. Záznamy o kvalitě
17. Vnitřní prověrky kvality
18. Školení
19. Služby
20. Statistické techniky

Vztah mezi MBNQA a ISO 9001



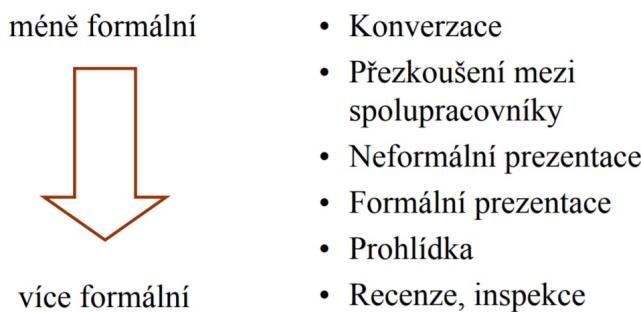
Jak začít SQA?

1. Formulace hypotézy
2. Pečlivý výběr vhodných metrik
3. Sběr dat
4. Interpretace dat
5. Iniciace akcí vedoucích ke zdokonalení
6. Iterace s vyhodnocením vlivu přijatých opatření, formulace dalších hypotéz

Přínos SQA

Když naleznou chybu v etapě požadavcích, tak cena bude jednoznačně několikanásobně nižší, než když jsi naleznou později. | Etapa | Cena nalezení a opravy || — | — | | Požadavky | 0.75 | | Návrh | 1.0 | | Kódování | 1.5 | | Testování | 3.0 | | Systémové testy | 10.0 | | Provoz | 60-100.0 | Pozn.: Cena normalizovaná vzhledem k ceně v etapě návrhu

Efektivita přezkoušení



V literatuře je argumentováno (např. Pressman), že efektivita roste se zvyšující se formálností.

Různé „review“ techniky

TYP METODY	TYPICKÉ CÍLE	TYPICKÉ VLASTNOSTI
Prohlídky	Minimální náročnost Školení vývojářů Krátká doba	Malá/žádná příprava Neformální proces Žádné měření Žádné FTR (FormalTechnical Review)
Odborné recenze	Zjištění požadavků Rozlišení nejednoznačností Školení	Formální proces Představení autora Rozsáhlá diskuse
Inspekce	Účinné a efektivní zjištění a odstranění všech defektů	Formální proces Kontrolní seznam Měření Fáze přezkoušení

	INSPEKCE, RECENZE	PROHLÍDKA
Požadavky	Výchozí požadavky	Detailní požadavky
Plány	Plány vývoje	
Vývoj	Návrh podrobného kódu	Návrh systému
Publikace		Pracovní & finální publikace
Testování		Implementace testu

Cíle formálního přezkoušení

- Odhalit chyby ve funkci, logice a implementaci software.
- Ověřit, že zkoumaná položka splňuje požadavky (odpovídá požadavkům).
- Zajistit, že položka byla prezentována s použitím předdefinovaných standardů.
- Zajistit jednotný vývoj.
- Zvýšit ředitelnost projektů.

Závažnost chyb, defektů

- Kritické
 - Defekty, které mohou způsobit pád systému, vznik chybných výstupů či chování nebo narušit uživatelská data. Není známa cesta, jak se těmto defektům vyhnout.
- Vážné
 - Defekty, které způsobují chybné výstupy či chování a je známa cesta, jak se těmto defektům vyhnout. Zasažena je významná část systému.
- Středně závažné
 - Defekty ovlivňující omezenou část funkcionality, které je možné se vyhnout nebo ji ignorovat.

- Málo závažné
 - Defekty, které mohou být opomenuty bez narušení funkčnosti

Indikátory kvalitních inspekcí jsou:

- Produkty jsou prověřovány až v době, kdy na to jsou připraveny.
- Termíny schůzek jsou přísné, ale zvládnutelné.
- Komentář autora je zařazen jen pokud je užitečný.
- Kontrolní seznamy a související materiály jsou užitečné.
- Schůzka recenze se zaměřuje na detekci problémů.
- Autor se nesmí cítit ohrožen.
- Úpravy jsou pečlivě prověřeny.
- Jsou odhaleny cesty na zlepšení inspekci a vývojového procesu.
- Účastníci považují metody za způsob efektivního zvýšení kvality.
- Každý se chce účastnit znova!

Chyby

Názvoslovie

Porucha je neschopnost systému nebo systémové komponenty provádět požadovanou funkci ve specifikovaných hranicích. Porucha může nastat, když se narazí na chybu, jejímž výsledkem je ztráta očekávané uživatelské služby.

Fault je chyba (defekt) v kódu, která může být příčinou jednoho nebo více selhání případně náhodná podmínka, která způsobuje, že funkční jednotka selhává při plnění požadované funkce (bug).

Error je chyba (omyl), kdy nesprávná nebo chybějící akce uživatele zapříčiní chybu (defekt) v programu.

- Čím později v životním cyklu detekujeme chybu, tím nákladnější bude její oprava.
- Mnoho chyb zůstane skryto a je odhaleno až po ukončení fáze, v níž byly udělány.
- V požadavcích je mnoho chyb.
- Chyby v požadavcích jsou především chybná fakta, opomenutí, rozporu a nejednoznačnosti.
- Chyby v požadavcích lze detektovat.

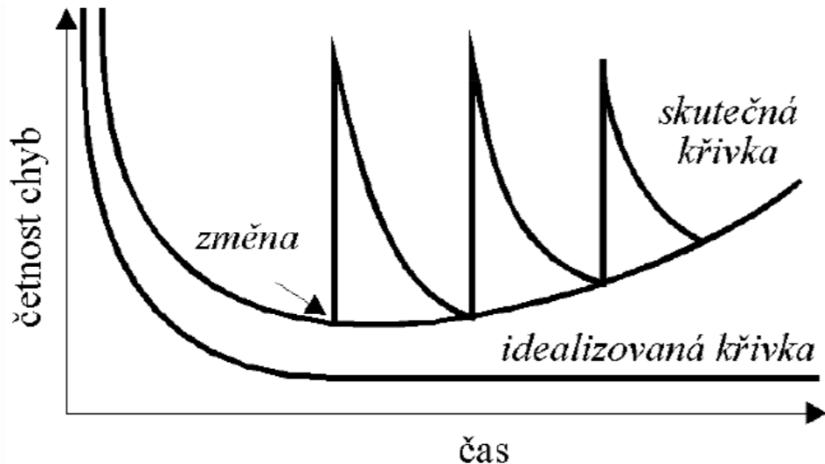
Chyby a opotřebení HW

- Na začátku je nový HW dostatečně výkonný, ale má poměrně dost chyb, které se ale zpravidla podaří časem odstranit.
- Na konci se začne projevovat opotřebení a HW začne zaostávat za svým výkonnějším okolím. Je třeba ho nahradit

Chyby a opotřebení SW

Je naivní si myslit, že s postupem času odstraníte v SW všechny chyby a zmizí tak veškeré problémy

Realita je taková, že zákazník požaduje během provozu nové a nové úpravy, což dělá systém složitějším, zanáší do systému další chyby. Jednou přijde den, kdy bude lepší to celé naprogramovat znova. (Lehmanovy zákony)

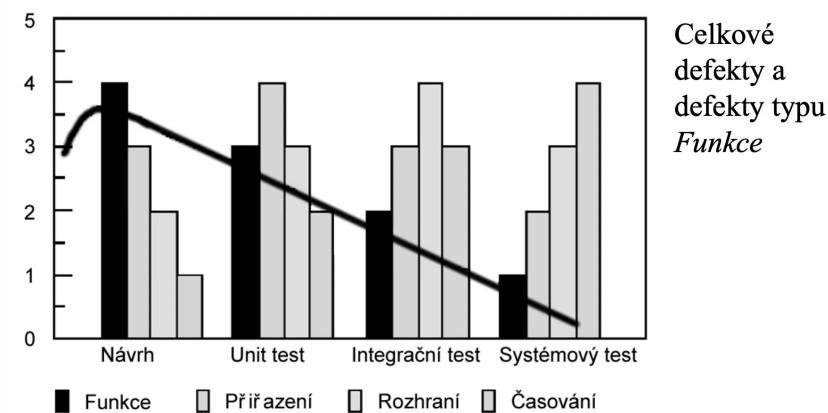


IBM ortogonální klasifikace defektů (ODC)

Dôležité!!! klasifikáciu defektov robíme kvôli tomu, že na základe kategórie dokážeme určite fázu vývoja kedy chyba vznikla - **funkce** – chyba ovlivňujúci schopnosti, rozhraní užívateľov, rozhraní výrobku, rozhraní s HW architekturou nebo globálnej datovou struktúrou. - **priřazení** – chyba pri inicializaci datovej struktury nebo bloku kódu. - **rozhraní** – chyba pri interakci s ostatnimi komponentami nebo ovladači pries volání, makra, riadicí bloky nebo seznamy parametrov. - **časování/serializace** – chyba, ktorá zahrnuje časovanie sdelených a RT prostredkov. - **ověřování** – chyba v logice programu, ktorá selže pri validaci dat a hodnot pred tím, než jsou použity. - **sestavení/balení/spojování** – chyba souvisejúca s problémami s repozitóriom projektu, změnami vedení, nebo správou verzí. - **dokumentace** – chyba, ktorá ovlivňuje publikace a návody pro údržbu. - **algoritmus** – chyba, ktorá sa týka efektivity alebo správnosti algoritmu alebo datové struktúry, ne však jejich návrhu.

Dôležité!!!

Funkční selhání podle etapy



- typy testov odhalujú rôzne typy chýb - príklad: typ chyby "Funkce" sa najviac objavuje v návrhu atď.