České vysoké učení technické v Praze
Fakulta Elektrotechnická

Bakalářská práce

Metriky a modely složitosti
Complexity Metrics and Models

Hana Klimešová

Vedoucí práce : Doc.Ing. Karel Richta, CSc.

Studijní program : Elektrotechnika a informatika strukturovaný bakalářský

Obor: Informatika a výpočetní technika

srpen 2007

Poděkování:

Chtěla bych na tomto místě poděkovat všem, kteří mi jakýmkoli způsobem pomáhali při vzniku této práce. Zvláště děkuji vedoucímu mé bakalářské práce Doc.Ing. Karlu Richtovi, CSc. za odbornou pomoc.

Prohlášení:

Prohlašuji, že jsem svou bakalářskou práci vypracovala samostatně a použila jsem pouze podklady uvedené v přiloženém seznamu.
Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č.121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze .....

Anotace:

Tato bakalářská práce se zabývá přehledem metrik používaných v informačních technologiích se zaměřením na metriky japonského odborníka Kaoru Ishikawy. Nejprve jsou rozebrány jednotlivé metriky používané v současné době a v dalších částech je pozornost soustředěna na návrh metodiky pro hodnocení složitosti softwarových produktů a na její aplikaci.

Summary:

This final project deals with metrics used in IT with a view to japan specialist Kaoru Ishikawa. There is a description of metrics used at the present time in the first part of this final project. There is design of methodics for complexity assessment of software products and application in the following parts.

Obsah

Seznam obrázků

Seznam tabulek

1 Úvod

V důsledku vstupu České republiky do EU, jak uvádí [1], se naskytla velká šance pro mnoho domácích firem prosadit se na jednotném evropském trhu. Ovšem k tomu je potřeba nabízet opravdu vysoce kvalitní produkty. Pro zvýšení konkurenceschopnosti podniku je nutné pravidelně analyzovat procesy v podniku probíhající a pomocí vhodných statistických metod dosáhnout jejich dlouhodobého zefektivňování. Většina firem buduje své systémy pomocí norem ISO 9000 a ty výslovně vyžadují aplikaci statistických metod a stanovení přesné specifikace jejich použití.

„Organizace musí plánovat a uplatňovat procesy monitorování, měření, analýzy a zlepšování, které jsou potřebné pro prokázání shody výrobku, pro zajištění shody systému managementu jakosti a pro neustálé zlepšování efektivnosti systému managementu jakosti. To musí zahrnovat přesné určení příslušných metod, včetně statistických metod a rozsahu jejich použití."

S problémy se zaváděním statistických a grafických metod se potýká velké množství firem. Největší problémy si však působí odpovědní lidé sami, jelikož mnoho z nich si pod pojmem „statistické metody" představuje komplex složitých vzorců a výpočtů, které jsou ve svém důsledku pro chod firmy naprosto zbytečné. Není však nutné aplikovat soustavu složitých statistických a grafických nástrojů a metod pro zlepšení vlastní výkonnosti firmy. Navíc k jejich aplikaci je možno využít různý statistický software, který dokáže ušetřit uživateli mnoho práce a poskytuje přehledné výstupy.

Práce seznamuje s různými metrikami, které je možno použít pro hodnocení kvality softwarových produktů v různých firmách, zároveň shrnuje klady a zápory jednotlivých metrik.

2 Popis problému, specifikace cíle

S obrovským „boomem“ softwarových aplikací vznikla poptávka po skutečně kvalitních produktech, které by splňovaly všechny požadavky zákazníků a cenově odpovídaly vynaloženému úsilí pro vývoj, návrh, implementaci a testování daných SW produktů.

S tím ovšem zákonitě také musela přijít otázka, který produkt je ten nejkvalitnější, jak se to pozná a později také jestli se to dá nějak změřit, popř. jak. Naskytla se tedy nutnost určit jednotná pravidla, podle kterých by se kvalita SW produktů dala měřit. Více než 30 let vývoje přineslo obrovské množství výzkumů, ale vzhledem k tomu, že mluvíme o novodobých SW produktech, kde musíme počítat s velkým množstvím vlastností a přídomků, které mít může, ale také nemusí, stále neexistuje konkrétní jednotný model pro stanovení kvality produktu.

Jedna z prvních metod byla LOC (Lines of Code), ta byla dále využita v metodě COCOMO 1.1 (1981), po které následovala COCOMO 2.0 . Z jiného soudku pochází metoda funkčních bodů (Albrecht 1979). Mezi nejznámější patří Halsteadova SW metrika (Halstead 1977) a McCabeova metrika cyklomatické složitosti (McCabe 1976). Práce dále hovoří o metrikách delfské věštírny, metodě analogie, ale především se zabývá metrikami navrženými Kaoru Ishikawou. Tyto metriky jsou nejprve rozebrány teoreticky, v další části je navržena metodika pro hodnocení kvality softwarových produktů, která je založena na kombinaci s dalšími metrikami uvedenými výše a v poslední části práce je tato metodika aplikována na hotový softwarový produkt za pomocí grafického statistického nástroje Statgraphics.

3 Přehled metrik používaných v současnosti

3.1 O metrikách obecně

Začněme tím, co je to vůbec metrika. Metrika je definována jako hodnotící funkce. SW metriky jsou podle [2] soustavy veličin používaných ke kvantitativnímu ohodnocení (hlavně produktivity a kvality) procesu vývoje SW a samotného produktu. Metrika jakosti softwaru je praktickým způsobem popsána v takto: „kvantitativní stupnice a metoda, která může být použita k určení dosažené hodnoty význačného rysu u konkrétního softwarového produktu." Znamená to, že metriky se používají pro reálné zjištění měřených hodnot atributů jakosti určitým prakticky proveditelným způsobem.

Měření provádíme zejména proto, abychom mohli ohodnotit kvalitu produktu, produktivitu tvůrců, přínosy nových nástrojů a metod, abychom vytvořili základ pro odhady a zdůvodnili požadavky na nové nástroje, vybavení a pod. Měření rozdělujeme na měření přímé a nepřímé:

- přímé (náklady, pracnost, doba, velikost kódu –LOC, rychlost, počet chyb, ...)

- nepřímé (funkčnost, kvalita, složitost, efektivnost, spolehlivost, udržovatelnost, ...)

Podle různých kritérií rozlišujeme mnoho metrik, uveďme si několik příkladů: technické metriky, metriky kvality, produktivity, metriky orientované na objem, na funkci a na lidi. Metriky orientované na objem jsou např. KLOC (KLines of Code) a LOC, metriky orientované na funkci se zaměřují na funkčnost nebo užitečnost programu, patří tam např. metoda funkčních bodů – FP. Metriky kvality jsou měřeny během procesu (složitost programu, modularita, ...) nebo po předání uživateli (počet chyb, udržovatelnost, ...). Samotné slovo kvalita představuje poměrně široký a neurčitý pojem, proto se jej někteří snažili lépe specifikovat. Zde jsou faktory kvality podle McCalla (1977):
- korektnost: Rozsah toho, jak program splňuje specifikaci a splňuje uživatelovy záměry.
- spolehlivost: V jakém rozsahu lze očekávat, že program bude plnit zamýšlené funkce s požadovanou přesností.
- efektivita: Množství výpočetních prostředků a kódu, které program potřebuje na splnění svých funkcí.
- integrita: V jakém rozsahu mohou být program nebo data používána neoprávněnými osobami.
- použitelnost: Úsilí vyžadované na učení, operování, přípravu vstupu a interpretaci výstupu programu.
- flexibilita: Úsilí vyžadované na modifikaci provozovaného programu.
- udržovatelnost: Úsilí vyžadované na vyhledání a opravu chyby v programu.
- testovatelnost: Úsilí potřebné na testování programu tak, abychom se ujistili, že plní zamýšlené funkce.
- přenositelnost: Úsilí potřebné na přemístění programu na jiný HW/SW.
- znovupoužitelnost: Rozsah, v jakém lze program nebo jeho části znovu použít v jiné aplikaci (funkce a balení produktu).
- schopnost spolupráce: Úsilí, které je nutné vynaložit pro připojení daného systému k jinému.

Hrubý odhad počtu LOC na 1 FP pro různé progr. jazyky:

Jazyk	LOC/FP (průměr)
Asembler	320
C	128
COBOL	105
FORTRAN	105
Pascal	90
Ada	70
OO jazyky	30
4GL	20
generátory kódu	15
Tab.procesory	6
graf.jazyky	4

Tabulka 3.1: Vztah metrik orientovaných na objem a na funkci - hrubý odhad

3.2. LOC

Tato metrika patří mezi metriky orientované na objem a slouží k odhadu velikosti SW. Je výsledkem SW projektů, snadno měřitelná, řada modelů pro odhady ji používá jako vstup. Jedná se o sečtení počtu řádků kódu programu. Existuje několik variant, zde jsou varianty podle Jonese (1986), uvedené v [8]:

1. Sčítáme pouze proveditelné řádky.
2. Sčítáme proveditelné řádky plus definice datových typů.
3. Sčítáme proveditelné řádky, definice datových typů a komentáře.
4. Počítáme proveditelné řádky, definice datových typů, komentáře a jazyk kontroly práce (např. ošetření výjimek apod.)
5. Počítáme řádky jako fyzické řádky na výstupu obrazovky.
6. Počítáme řádky jako řádky ukončené logickými oddělovači.

Nevýhody LOC

i. špatný SW návrh může způsobit nadměrnou hodnotu LOC
ii. běžným uživatelům těžce srozumitelný
iii. závisí na použitém prog. jazyku, nevhodné pro neprocedurální jazyky
iv. při odhadech neznáme předem velikost

Na internetu – www odkaz [4] se dokonce lze dočíst o drobných rozepřích, kdy se programátoři přetahují, kdo je schopen napsat víc řádek kódu za den. Při bližším zkoumání však zjistíme, jak je toto „přetahování“ pošetilé. Jak bylo řečeno, záleží na používaném programovacím jazyce, a také na tom, jaké je zadání programátora. Zda pouze programuje a návrh a analýzu mu dělá někdo jiný, nebo si vše dělá sám. Pokud programátor ví, co má programovat, může být několikanásobně rychlejší než programátor, který vše musí vymýšlet. Počet řádků napsaného kódu tedy nemůže být dostatečně objektivní metrikou pro hodnocení kvality či složitosti SW produktů, ani

objektivním měřítkem výkonnosti vývojáře software. Na druhé straně, pokud kód dělá co by měl dělat, jde ve větších týmech asi o jediné měřítko, které ukáže, že vývojář v práci nespí, nebo minimálně nespí celou pracovní dobu.

Například *Software Productivity Consortium* uvádí ve svých průmyslových standardech 26 řádek zdrojového kódu za měsíc. Nepřímé zdroje uvádějí, že společnost IBM dělala zpětný interní průzkum u svých programátorů a došla k neuvěřitelným 14 řádkům kódu na den z celoročního průměru. Jde o řádky, které nakonec v produktu zůstávají, ale přesto jde o neuvěřitelně nízké číslo.

Další zajímavé informace:

- NASA: znovupoužití zdrojového kódu beze změny představuje 20% námahy vytvoření porovnatelného nového kódu.
- *D.N.Card, D. V. Cotnoir, and C. E. Goorevich:* cena vymazání řádky zdrojového kódu je stejná jako cena napsání této řádky.
- *R.B. Grady:* programátor udělá průměrně 2.5 chyby za hodinu.
- *aR. Dion:* 40-45 % úsilí programátora je strávena na opravách kódu.
- *R.B. Grady:* průměrně 25% chyb je do programu zaneseno při opravách jiných chyb a při údržbě.

Pro zajímavost několik operačních systémů a počet řádků zdrojového kódu (LOC) v milionech. Operační systémy MS Windows jsou uvedeny dle *Garyho McGrawa*. Mělo by jít o celkovou distribuci, nejen o jádra operačních systémů:

Rok	**Operační systém**	**LOC**
1995	MS Windows NT 4.0	16
1999	MS Windows 98	18
2002	MS Windows XP	40
2000	Linux Debian Potato	55
2001	Linux Red Hat 7.1	30

Tabulka 3.2: Příklady operačních systémů podle celkového počtu řádků zdrojového kódu

"Měření produktivity vývoje software pomocí počítání množství řádek zdrojového kódu (*Lines Of Code - LOC*) je jako posuzovat letadla podle toho, kolik váží."

Bill Gates

Měření programu podle počtu řádků zdrojového kódu zřejmě není ten správný způsob, neboť pak by Windows musel být nejlepší program na světě.

3.3 COCOMO – (Constructive Cost Model)

Původní metoda Cocomo 1.1 byla vyvinuta na přelomu sedmdesátých a osmdesátých let minulého století Hary Boehmem (někdy je také označována jako COCOMO'81 podle roku jejího vzniku - 1981). Je to jedna z nejznámějších a nejpopulárnějších metod pro odhad nákladů. V průběhu vývoje v oblasti tvorby informačních systémů (IT) byl

původní Cocomo nahrazen v polovině devadesátých let novou verzí, která je známa pod označením Cocomo 2.0 (nebo také COCOMO II). Nová verze odráží vývoj v oblasti IT ve vztahu s postupných přechodem na objektové prostředí a měla by odstranit handicap způsobený nutností znát počet zdrojových řádků. Cocomo 2.0 je také nastaveno pro znovupoužití SW a reinženýring, jsou zde použity automatické nástroje pro překlad existujícího SW. Cocomo 1.1 poskytovalo malou přizpůsobitelnost pro tyto faktory.

Cocomo 2.0 využívá object points pro provedení odhadu. Objekty zahrnují obrazovky, reporty a moduly v programovacích jazycích třetí generace. Object points nejsou nutně navázány na objekty v objektově orientovaném programování. Počet nezpracovaných objektů je odhadnutý, složitost každého objektu je odhadnuta a je vypočítán vážený součet (Object-Point count). Procento užití a předpokládaná produktivita jsou odhadnuty také. S touto informací může být spočítán i odhad pracnosti.

Vzorec pro výpočet odhadu pracnosti:

$$Effort = \frac{NOP}{PROD}$$

$$NOP = OP\frac{(100 - \%reuse)}{100}$$

kde *NOP* je *New Object Points*

PROD je *Productivity Rate*

PROD = 4 - very low, 7 - low, 13 - normal, 25 - high, 50 - very high

reuse – značí opětovné použití

3.4 FP – metoda funkčních bodů

Vychází z empirického vztahu mezi počitatelnými veličinami a ohodnocením složitosti, navržena pro aplikace obchodních informačních systémů, příp. systémové a VT aplikace. Je to metrika, která slouží k odhadování pracnosti vyvíjeného softwaru metodou výpočtu funkčních bodů, které se vypočítají z rozsahu požadavků na systém (např. počet datových struktur, uživatelských vstupů a výstupů, ...). Z výsledného počtu funkčních bodů lze odhadnout velikost budoucího programu, protože pro různé programovací jazyky je na základě statistických měření známo, kolik příkazů je třeba na pokrytí jednoho funkčního bodu. Metoda „feature points“ je variantou této metody pro případ objektové tvorby softwaru.

Vzhledem k tomu, že je nezávislá na jazyku, lze ji poměrně dobře využít pro metody odhadu, např. pro stanovení časového harmonogramu (pozn.: podle nezávislých studií prováděných například analytickou organizací Gartner Group však i ten „nejlepší“ časový harmonogram bývá dodržen jen v několika málo procentech případů).

Metoda funkčních bodů patří sice mezi ty složitější, ale poskytuje poměrně správné hodnoty. Princip vymyslel Allan Albrecht a spočívá v rozdělení cílového systému do kategorizovaných prvků (těmi jsou například externí vstupy či výstupy). Následně se každému prvku přiřadí složitost, která se vynásobí tzv. váhovým faktorem (ke stanovení složitosti jsou k dispozici tabulky zohledňující vlastnosti prvků; stejně tak jsou pevně stanoveny váhové faktory). Součet takto získaných hodnot pak určuje tzv. neupravený počet funkčních bodů. Dalším krokem je přechod k upravenému počtu funkčních bodů (zahrne se vliv dalších faktorů, které nejsou obsaženy v kategorizovaných prvcích). A konečně se stanoví celková doba trvání realizace projektu v tzv. člověkodnech (opět jsou k dispozici převodní tabulky dle použitých vývojových nástrojů a dalších prostředků). Hlavní nevýhodou této metody je nutnost mít co možná nejpřesnější představu o výsledku našeho snažení.

Analýza funkčních bodů

Lze ji uplatnit na konci detailního návrhu, kdy jsou již specifikovány transakce v budovaném systému a je jasná struktura a obsah databáze, jak uvádí [4]. Metoda se skládá ze dvou základních kroků. Prvním je specifikace složitosti systému. Druhým je převod tzv. funkčních bodů definujících složitost systému na člověkodny pracnosti. Primárně sloužila metoda právě k měření složitosti systému. Vychází se z úměry, že čím je systém složitější, tím je také pracnější. Z toho se odvozuje i jeho pracnost. Metoda je poměrně objektivní.

Odhad se provádí na úrovni celého projektu. Odhadnutou hodnotu celkové pracnosti je třeba rozdělit mezi jednotlivé etapy, fáze a činnosti. Postup je poměrně komplikovaný, v prvním kroku se hodnotí 5 typů funkcí:

- vstupní transakce (jména souborů)
- dotazy (interaktivní vstup s nutností odpovědi)
- výstupní transakce (zprávy, záznamy)
- externí vazby (soubory sdílené jiným SW systémem)
- logické datové soubory (neviditelný externí systém, databáze)

• Každý typ funkce je buď jednoduchý, průměrný nebo složitý. Pro hodnocení jsou vydány tzv. matice složitosti. Každému prvku je podle složitosti přiřazen určitý váhový počet bodů (od 3 pro jednoduché vstupní transakce až po 15 pro složité log. datové soubory). Celkový počet bodů je nazýván součtem tzv. neupravených funkčních bodů (UFP Unadjusted function points).
• Ve druhém kroku se zjišťují hodnoty „opravných faktorů„. Tyto faktory charakterizují prostředí, ve kterém je projekt realizován, např. distribuovanost systému, interaktivita dialogů apod. Na jejich základě je vypočítán „upravující koeficient„ nebo také „vyrovnávací faktor“, kterým se neupravené funkční body vynásobí .

Vypočítané upravené funkční body se pak podle koeficientu produktivity převedou na pracnost. Jedné člověkohodině odpovídá určitý počet bodů.

Váhové faktory podle Albrechtova originální metody, uvedené v [6] takto:

Typ funkce	Jednoduchý	Průměrný	Složitý
Vstupní transakce	x3	x4	x6
Výstupní transakce	x4	x5	x7
Logické dat.soubory	x7	x10	x15
Externí vazby	x5	x7	x10
Dotazy	x3	x4	x6

Tabulka 3.3: Váhové faktory podle Albrechtovy originální metody

Klasifikace na jednoduchý, průměrný nebo složitý váhový faktor se řídí následující tabulkou:

	1-5 Datových typů	6-19 Datových typů	20+ Datových typů
0-1Typů referenčních souborů	Jednoduchý	Jednoduchý	Průměrný
2-3Typů referenčních souborů	Jednoduchý	Průměrný	Složitý
4+Typů referenčních souborů	Průměrný	Složitý	Složitý

Tabulka 3.4: Klasifikace váhových faktorů

Pro nalezení FP se UFP násobí „upravujícím koeficientem“ (TCF Technical complexity factor) podle následující rovnice:

TCF = 0.65 + (součet faktorů Fi) / 100 ,
kde F_i mají hodnotu 0-5 pro i = 1..14 (složitost zpracování), hodnoty: 0 nemá vliv, 1 nahodilý, 2 mírný, 3 průměrný, 4 významný, 5 podstatný vliv

1. Požaduje systém zálohování a obnovu? (Jednoduchá výměna)
2. Jsou požadovány datové přenosy? (Přenášení dat)
3. Obsahuje funkce distribuovaného zpracování? (Celkové zpracování)
4. Je požadován kritický výkon? (Výkonnost)
5. Bude systém pracovat za silném provozu? (Silně užívané konfigurace)
6. Požaduje systém přímý vstup dat? (Online datové vstupy)
7. Požadují vstupní transakce přímé vstupy dat prostřednictvím více
8. obrazovek nebo operací? (Rychlost transakce)
9. Jsou hlavní soubory aktualizovány přímo? (Online update)
10. Jsou vstupy, výstupy, soubory a dotazy složité? (Jednoduché operace)
11. Je složité vnitřní zpracování? (Rozmístěné funkce)
12. Je kód navržen pro opakované použití? (Znovupoužitelnost)
13. Jsou v návrhu zahrnuty konverze a instalace? (Jednoduchá instalace
14. Je systém navržen pro vícenásobné instalace na různých místech? (Rozmanitá poloha)

15. Je aplikace navržena tak, aby usnadnila změny a snadné uživatelské ovládání (Efektivita koncového uživatele)

Výsledný počet FP se tedy vypočítá : FP = UFP x TCF

Výhody FP

i. Nezáleží na množství kódu .
ii. Není závislý na použitém prog. jazyku.
iii. Specifikace potřebných dat již na začátku projektu. Potřebujeme pouze detailní specifikaci.
iv. Přesnější než odhady metodou LOC .

Nevýhody FP

i. Možnost subjektivního posuzování.
ii. Těžké pro výpočet.
iii. Ignorace kvality výstupu.
iv. Orientace na tradiční data zpracovávaných aplikací.
v. Fyzikální význam, vyžaduje zkušenost.

Organizace jako International Function Point Users Group IFPUG se snaží najít pravidla pro FP, aby zabezpečovaly, že jejich počet bude porovnatelný navzdory různým společnostem. Organizace IFPUG navázala na koncept A.Albrechta, který vyvinul základní koncept metody FP v laboratořích IBM na konci 70.let min. století.

3.5 Metoda analogie a metoda delfské věštírny

Mezi další známé metody odhadu pro stanovení časového harmonogramu patří Metoda analogie či Metoda delfské věštírny, uvedené v [5].

První z nich, Metoda analogie, je použitelná v případech, kdy již nějaký ten úspěch máme za sebou. Další podmínkou je co nejpřesnější zaznamenání průběhu předchozích projektů (tedy popis jednotlivých kroků, atypické problémy ...). Odhad se pak provádí na základě podobnosti, kterou lze vyjádřit například pomocí tzv. indexu shodnosti prvků (CCI – Component Commonality Index). Ten může nabývat hodnot z intervalu <0,10>, kde spodní hranice znamená naprostou rozdílnost a každý krok směrem výše zlepšení o 10 %. Nevýhodou této metody je právě sama její podstata – výsledek je ovlivněn zejména vlastnostmi, které se v původních projektech nevyskytovaly.

Metoda delfské věštírny je jakousi sázkou do loterie. Externím specialistům specifikujete co nejpřesněji váš cíl a požádáte je o jejich názor na časovou náročnost. Získané odhady se dále zpracovávají běžnými statistickými metodami. Na první pohled se může jednat o metodu v praxi nepoužitelnou, avšak v některých případech (zejména nejste-li schopni splnit podmínky u jiných metod) může posloužit pro alespoň částečně podložený odhad.

3.6 Halsteadova metrika velikosti programu

Je založena na předpokladu, že všechny programy se skládají z konečného počtu programových jednotek, tzv. tokenů, které jsou rozeznatelné v syntaktické fázi překladačem. Počítačový program je tedy považován za sbírku tokenů, které mohou být klasifikovány jako operátory a operandy.

Jednotlivé komponenty důležité pro měření :

- n_1 = počet unikátních nebo rozdílných operátorů v implementaci
- n_2 = počet unikátních nebo rozdílných operandů v implementaci
- N_1 = celkový počet ze všech unikátních nebo rozdílných operátorů použitých v implementaci (tedy i těch, které se opakují)
- N_2 = celkový počet ze všech unikátních nebo rozdílných operandů použitých v implementaci (tedy i těch, které se opakují)

Příklady operátorů : "+" , "*" , index "[...]" nebo středník " ; "

Pro tuto metriku Halstead definoval:

i. Délku slovníku n jako $n = n_1 + n_2$
ii. Délku programu N jako $N = N_1 + N_2$

Jednoduchý příklad uvedený ve [7] :

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

• Rozdílné operátory: if ( ) { } > < = * ;
• Rozdílné operandy: k 2 3 x
• n1 = 10
• n2 = 4
• N1 = 13
• N2 = 7

Na základě těchto jednoduchých měření Halstead vynalezl systém rovnic, které vyjadřují celkovou délku slovníku, celkovou délku programu, odhadnutý minimální objem pro algoritmus, skutečný objem, úroveň programování (metrika složitosti SW), programová náročnost a další rovnice, např. pro programátorské úsilí a předpokládaný počet chyb v SW :

Délka programu: N = N1 + N2

Délka slovníku: n = n1 + n2

Odhadnutá délka: EN = n1 log2 n1 + n2 log2 n2
– dobrý odhad délky dobře strukturovaných programů

Koeficient čistoty programování (purity ratio):
PR = EN /N

Objem: V = N log2 n
– počet bitů potřebných pro rozhodnutí při volbě každé z ***n*** položek programového slovníku
Úroveň programování: L = V*/V

V* je objem nejkompaktnější implementace, tzv. minimální objem

L *je dobrá metrika srozumitelnosti programu*

Programová náročnost (difficulty): D = V/V*

Programátorské úsilí: E = V/L

Předpokládané chyby (B) : B = V/ S*
S* je číslo udávající počet rozhodnutí mezi chybami, podle Hasteada S* = 3000.

Výhody Halsteadovy metriky :

i. Nevyžaduje hloubkovou analýzu struktury programu.
ii. Předpovídá námahu na údržbu.
iii. Je výhodná v oblasti plánovacích projektů.
iv. Měří celkovou kvalitu programu.
v. Jednoduchá na výpočty.
vi. Může být použita pro všechny programovací jazyky.
vii. Četné studie v průmyslových odvětvích podporují užití Halsteadovy metriky pro předpovídání náročnosti programu a také počtu programových chyb.

Nevýhody Halsteadovy metriky :

i. Závisí na úplném kódu.
ii. Má malé nebo žádné využití v modelech odhadu a předpovědí – pro stanovení odhadnuté délky EN musí být program zcela nebo téměř dokončen.
iii. Nepřizpůsobuje se tak dobře aplikaci z hlediska návrhu jako McCabeův model.

3.7 Cyklomatická složitost – metrika McCabe

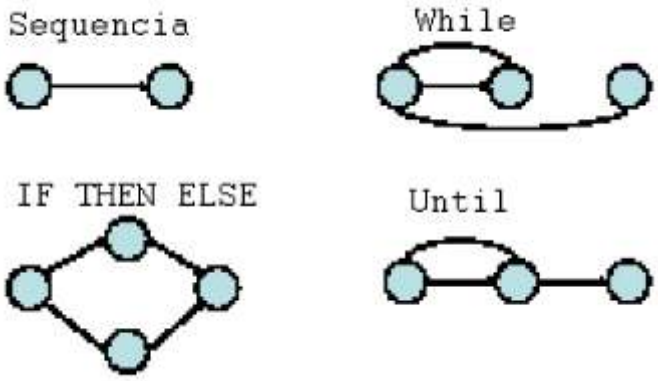
Měření složitosti programu vynalezl v roce 1976 Thomas McCabe. Vynalezl systém, který nazval cyklomatická složitost programu. Tento systém měří počet nezávislých cest v programu, což udává číselnou hodnotu složitosti. V praxi je to součet čísel testovacích podmínek programu.

Cyklomatická složitost (The cyclomatic complexity CC) grafu (G) se může vypočítat následují rovnicí :

CC(G) = Number (edges) - Number (nodes) + 1

CC(G) = Počet (hran) - Počet (uzlů) + 1

Uzly reprezentují jeden nebo více příkazů v kódu (úkoly v běhu programu) a hrany reprezentují kontrolní toky mezi uzly.



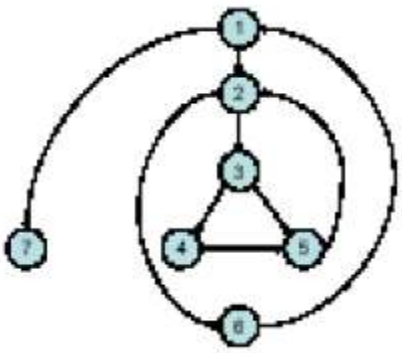
Obr. 3.1: Grafová notace toků

• Stanovení nezávislých cest grafu (basis set)
• V(G) = E – N + 2
– E je počet hran v toku grafu
– N je počet uzlů
• V(G) = P + 1
– P je číslo uzlu predicate

V(G) je počet uzavřených oblastí planárního grafu. Počet oblastí se zvětšuje s počtem cest a cyklů. Slouží ke kvantitativnímu měření testované složitosti a k indikaci konečné spolehlivosti. Experimentální data ukazují, že hodnota V(G) by neměla být větší než 10. Testovaní nad tuto hodnotu je pak velmi obtížné.

Příklad A:

```
i = 0;
while (i<n-1) do
j = i + 1;
while (j<n) do
if A[i]<A[j] then
swap(A[i], A[j]);
end do;
i=i+1;
end do;
```

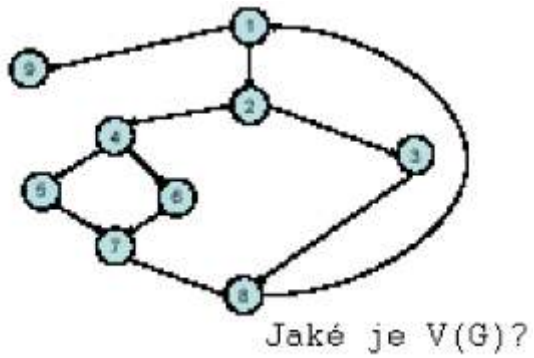


Obr. 3.2: Graf k příkladu A

Výpočet V(G)
• V(G) = 9 – 7 + 2 = 4
• V(G) = 3 + 1 = 4
• Basis Set
- 1, 7

– 1, 2, 6, 1, 7
– 1, 2, 3, 4, 5, 2, 6, 1, 7
– 1, 2, 3, 5, 2, 6, 1, 7

Příklad B:



Obr. 3.3: Graf k příkladu B

Výsledky četných experimentů (G.A. Miller) naznačují, že modulový přístup nula odpadá, když McCabeova cyklomatická složitost je v rozmezí 7 ± 2. Díky studiím programů psaných v jazyce pascal nebo fortran (Lind a Vairavan 1989) se zjitilo, že cyklomatická složitost mezi 10 a 15 minimalizuje počet modulových změn.

Thomas McCabe, vynálezce cyklomatické složitosti, založil společnost McCabe & Associates zabývající se také jinými metrikami, které na cyklomatickou složitost navazují.

Výhody McCabeovy cyklomatické složitosti :

i. Může být použita jako metrika jednoduchá pro údržbu.
ii. Pokud je použita jako metrika kvality, dává relativní složitost různých návrhů.
iii. Může být použita dříve v životu cyklu než Healsteadova metrika.
iv. Je vhodná pro měření minimální náročnosti a nejlepších oblastí koncentrace pro testování.
v. Dává návod pro testovací procesy díky logickému limitování programu během jeho vývoje.
vi. Jednoduchá k použití.

Nevýhody McCabeovy cyklomatické složitosti :

i. Cyklomatická složitost je měření řídící složitosti (control complexity) a ne datové složitosti (data complexity)
ii. Tu samou váhu má umístění jak vnořených tak ne-vnořených cyklů. Nicméně, důkladně vnořená podmínková struktura je těžší pro pochopení než ne-vnořená struktura.
iii. Může dávat matoucí schéma týkající se velkého počtu srovnání a velké rozhodovací struktury.

Výše uvedené metriky jsou pouze přehledem nejznámějších metrik. V současné době jich existuje mnohem více a další se neustále vyvíjejí.

4 Podrobný popis srovnávaných systémů / řešení.

4.1 Použití sedmi základních nástrojů na měření kvality SW

Se sedmi základními nástroji pro řízení a zlepšování jakosti byla česká odborná veřejnost seznámena prvně K. Ishikawou v roce 1973 při jeho návštěvě Prahy. Jejich původní název byl *„Seven Tools“* a jejich obsah byl formován v průběhu padesátých a šedesátých let minulého století v Japonsku právě K. Ishikawou a E.Demingem, který v té době v Japonsku dlouhodobě působil. Společným rysem těchto nástrojů je požadavek na trvalou týmovou práci, tedy požadavek, který přežil všechny vývojové fáze řízení jakosti až po současný přístup formulovaný v normách ISO řady 9000 z roku 2000.

Základní statistické prostředky ke kontrole kvality podpořené Ishikawou (1989) jsou široce užité v průmyslové výrobě. Podle [8] by se měli stát základní částí literatury, která se zabývá kontrolou kvality, a jsou známy jako „Ishikawových sedm základních nástrojů“. Existuje mnoho cest, jak analyzovat softwarové produkty, aplikace nástrojů Ishikawy představuje pouze sadu základních operací. Musíme mít na paměti, že tyto statistické nástroje mohou být velmi užitečné pro vedoucí projektů a pro výrobní manažery hlavně ve středních a větších vývojových organizacích. V kontrastu tedy stojí, že neposkytují důležité informace softwarovým vývojářům, jak by se dala zlepšit kvalita jejich návrhů nebo realizace.

Vzhledem k tomu, že u malých projektů jsou méně zřejmé statistické vzory parametrů vývojového procesu, nelze zde využít výhody statistik a ne všechny zmíněné nástroje jsou pro malé projekty užitečné. Navíc, ačkoli výhody těchto nástrojů byly dokázány ve dlouhodobých výrobních operacích, jejich použití a role v softwarovém vývoji nebyly zatím širším okruhem lidí rozpoznány.

Například, použití regulačních diagramů ve výrobě může zajistit jistou kvalitu koncového produktu, jakmile je proces definovaný a kontrolní limity tvoří sadu. Nicméně v softwarovém vývoji je proces komplex a zahrnuje vysoký stupeň tvořivosti a duševní aktivity. Je to extrémně těžké, pokud ne přímo nemožné, definovat ve statistických termínech proces schopný softwarového vývoje. Proto dosáhnutí statistického procesu kontroly v softwarovém vývoji může znamenat mnohem víc než kontrola při sestavování diagramů. To může vyžadovat například vývoj nových technologií, případně CASE nástrojů a použití modelů chyb a spolehlivosti, které odhadují technici. Správné použití sedmi základních nástrojů může však vést k pozitivním dlouhodobým výsledkům pro procesové zlepšení a kvalitnímu vedení v softwarovém vývoji, Kaoru Ishikawa dokonce tvrdí, že 95% problémů podniku lze řešit použitím sedmi základních nástrojů řízení jakosti.

Následující sekce začnou s krátkým popisem nástrojů, následované diskuzí o každém nástroji s příklady jeho aplikací, kde jsou také popsány vlivy těchto nástrojů na procesovém zlepšení a rozhodování.

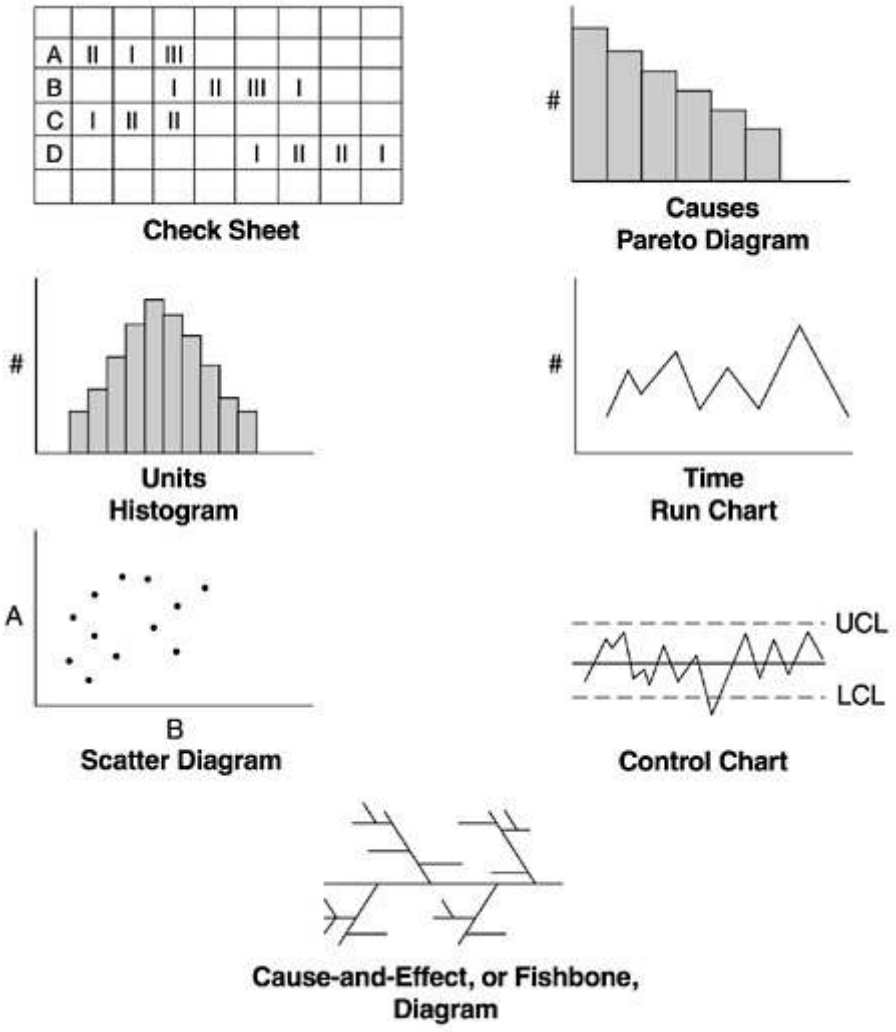
Příklady jsou jednak z literatury o softwarovém inženýrství nebo ze softwarových projektů rozvinutých u IBM Rochester v Minnesotě.

Kromě sedmi základních nástrojů diskutujeme také o vztahových diagramech, které jsou efektivní pro malý tým a částečně užitečný v zobrazování vztahů v diagramech příčin a následků (cause-and-effect diagram).

4.2 Ishikawových sedm základních nástrojů

Těmito základními nástroji se myslí: formulář (checklist, kontrolní formulář – check sheet), Paretův diagram (Pareto diagram), histogram (histogram), bodový diagram (scatter diagram), průběhový diagram (run chart), regulační diagram (control chart) a diagram příčin a následků (cause-and-effect diagram). Obr 4.1 ukazuje jednoduchou reprezentaci těchto nástrojů.

Kontrolní formulář je papírová forma s tištěnými položkami, které mají být zaškrtnuty. Jeho hlavní účely jsou, aby snadno sbíral a uspořádal data, aby takto nasbíraná data mohly být snadno využitelná i později. Jiným typem kontrolního formuláře může být takzvaný kontrolní formulář s potvrzováním. To se týká hlavně vlastností týkajících se kvality procesu nebo produktu. Abychom odlišili tento potvrzující kontrolní formulář od kontrolního formulář, který shromažďuje data, používáme termín formulář – checklist. V mnoha prostředích vyvíjejících software hledisko shromažďování dat je elektronicky automatizované a přispívá k rozvoji formulářů, které jsou použity ve výrobě. Naše diskuze o nástrojích se tedy soustředí na formuláře (check-listy).



Obr. 4.1: Ishikawových sedm základních nástrojů pro řízení kvality

Paretův diagram je jednoduchá grafická metoda pro seřazení položek od nejčetnější po nejméně četnou. Paretův diagram je založen na Paretově principu, podle kterého na většinu následků působí pouze několik položek.

Největšího zlepšení s vynaložením nejmenšího úsilí se dosáhne odlišením nejdůležitějších položek od méně důležitých.

Paretův diagram zobrazuje v klesajícím sledu příspěvek každé položky na celkovém následku. Relativní příspěvek může spočívat v počtu výskytů, v nákladech nebo jiných mírách dopadu na následek.

Ke znázornění relativního příspěvku každé položky se používají sloupce, pro znázornění kumulativního příspěvku položek se používá kumulativní křivka četnosti.

Postup tvorby Paretova diagramu, včetně Paretovy analýzy, jsou popsány v kapitole 4.2.2. Paretův diagram je většinou používaný v klesajícím vyobrazení, ve kterém nejčetnější položky jsou spojovány s určitým problémem. Je pojmenován podle italského ekonoma z 19. století Vilfreda Pareta (1848 - 1923), který vysvětlil v odborných názvech svůj princip - rozdělení majetku - toto rozsáhlé dělení majetku se týká jen malého procenta populace. V roce 1950 Juran (Joseph Moses Juran – americký průmyslový inženýr, známý jako obchodní průmyslový guru) aplikoval princip na problém kvality a dokázal, že většina následků (80 - 90%) problémů s jakostí je způsobena malým počtem příčin (5 – 20%). Tyto příčiny Juran nazval „životně důležité“, zbylá procenta příčin označuje jako „příčiny zajímavé“(popř. triviální). Ve vývoji softwaru jsou na ose X Paretova diagramu nežádoucí příčiny a na ose Y nežádoucí počet. Paretův diagram může identifikovat několik příčin, které vysvětlují převahu chyb. Ukazuje, které problémy by měly být řešeny nejdříve pro zmenšení chybovosti a zvětšení výkonu. Pareto analýza většinou uvádí 80-20 (20% nejdůležitějších příčin způsobuje 80% chyb). Soustředěním pozornosti na tyto příčiny a jejich řešením lze dosáhnout nejlepšího zlepšení.

Histogram je grafická reprezentace dat, která poskytne rychlou informaci o tvaru rozdělení statistického souboru (soubor hodnot, znaků či parametrů procesu) a o jeho statistických charakteristikách. Histogram je grafická reprezentace četnosti vzorků nebo populace. Na osu X se vynáší interval parametrů (například stupeň závažnosti nebo softwarové vady), řazené ve vzestupném pořadí zleva doprava a osa Y obsahuje číselná vyjádření. V histogramu jsou sloupce četností řazeny v pořadí X-ové proměnné, zatímco u Paretova diagramu jsou sloupce četností řazeny v pořadí počtu četností. Cílem histogramu je ukázat distribuční charakteristiky parametrů jako celkový tvar, centrální tendence, rozptylování, nesouměrnost a tak dále. To přinese lepší porozumění parametru zájmu.

Průběhový diagram sleduje provedení parametru zájmu v závislosti na času. Osa X je čas a osa Y představuje hodnotu parametru. Průběhový diagram se nejlépe využívá u analýzy nových směrů, zvláště jestliže jsou dostupná dřívější data pro porovnání s nynějším trendem. Ishikawa vložil do své knihy (1989) také množství různých grafů jako je kruhový diagram, sloupcový diagram apod. do sekce průběhové diagramy. Příkladem průběhového diagramu v softwaru je týdenní číslo problémů, u kterých se započalo s řešením v nahromaděných objednávkách – ukazuje množství softwarových fixních nákladů pro vývojový tým.

Bodový diagram je nástroj ke zjištění či ověření vzájemné závislosti dvou jevů.

Základní přínosy:

- odhalí případnou existenci závislosti mezi zkoumanými jevy
- znázorní charakter a těsnost případné závislosti
- potvrdí nezávislost

• přispívá ke snížení rizik při eventuálních změnách hodnot jedné proměnné

Bodový diagram názorně ukazuje vztah mezi dvěma proměnnými. Každý bod v bodovém diagramu reprezentuje sledování obou závislých či nezávislých proměnných. Bodové diagramy pomáhají při rozhodování (např. – jestliže je na ose X naplánovaná určitá akce, pak její následek je očekáván na ose Y). Sledujeme-li u statistických jednotek dva kvantitativní znaky *x, y*, obdržíme celkem *n* dvojic hodnot x_i, y_i, *i = 1, 2, …., n*. První představu o závislosti obou znaků lze získat tak, že zjištěná data znázorníme bodovým diagramem, v němž je každá dvojice x_i, y_i znázorněna jako bod v pravoúhlé souřadné soustavě, kde na vodorovné ose (osa X) je umístěna stupnice hodnot znaku *x* a svislé (osa Y) stupnice hodnot znaku *y*. Vynesené body tvoří jakýsi „roj“, z něhož můžeme vystopovat charakteristické rysy závislosti obou znaků. Bodový diagram nám tedy poskytuje informaci o průběhu závislosti a také o její těsnosti. Bodový diagram bychom měli volit v případě, pokud máme vyjádřen koeficient úměry dvou proměnných. Nejpoužívanější koeficient úměry je Pearsonův koeficient momentu výrobku, který předpokládá lineární závislost. Jestliže je závislost nelineární, Pearsonův koeficient nemůže ukázat žádnou závislost, proto může sdělovat i chybné informace.

Regulační diagram je nástroj, který zobrazuje vývoj sledovaného jevu. Základní informací je posloupnost výběrů v čase a výběry jsou reprezentovány:
• Střední hodnotou
• Variabilitou

Zásah musí zahrnovat čtyři kroky:
• Prověření jakosti výstupů od předchozího provedení výběru
• Vyšetření příčin změny, například pomocí diagramu příčin a následků
• Odstranění nalezené příčiny
• Opatření do budoucna - zamezení opakovanému působení oné příčiny

Regulační diagramy jsou hlavními nástroji statistické regulace procesu. Autorem je americký odborník W.A.Shewhart. Jde o grafickou pomůcku umožňující oddělit identifikovatelné (zvláštní) příčiny od náhodných (obecných) příčin variability procesu. Konstrukce regulačních diagramů má matematicko-statistický základ. Regulační diagramy využívají provozních údajů pro stanovení mezí, uvnitř kterých lze očekávat budoucí pozorování, není-li proces ovlivňován vymezitelnými nebo zvláštními příčinami.

Na ose *x* se vynášejí pořadová čísla podskupin, na ose *y* hodnoty výběrových charakteristik sledovaného znaku jakosti či parametru procesu, které vypočteme z chronologicky za sebou jdoucích hodnot znaku jakosti získaných při provádění pravidelných výběrových kontrol. Minimální doporučovaný počet podskupin pro stanovení regulačních mezí je 20 – 25 podskupin s rozsahem min. 4 – 5 jednotek (měření).

Regulační diagram dále obsahuje střední přímku (SL), horní a dolní regulační meze (USL, LSL). Regulační meze vymezují pásmo, v němž leží s předem zvolenou pravděpodobností hodnoty výběrových charakteristik jednotlivých podskupin. Za předpokladu, že na zkoumaný proces působí v daném časovém úseku jen náhodné příčiny variability procesu, lze při stanovení regulačních mezí vycházet z pravděpodobnostního rozdělení příslušných výběrových charakteristik. Nejčastěji se uvedená pravděpodobnost volí na úrovni 0,9973, což znamená, že regulační meze jsou vzdáleny od centrální čáry 3 směrodatné odchylky z dané výběrové charakteristiky na obě strany.

Princip využívání regulačních diagramů:
- v pravidelných časových intervalech provádíme náhodný odběr předem stanoveného pevného počtu produktů tvořících tzv. podskupinu
- u odebraných produktů (stejného druhu, vyrobených za stejných podmínek) se měří či zjišťuje stejný znak jakosti *X* (např. určitý rozměr)
- u naměřených či jinak zjištěných hodnot znaku jakosti se vypočte pro každou podskupinu jedna nebo více výběrových charakteristik
- hodnoty vypočtených výběrových charakteristik se chronologicky zakreslí do regulačního diagramu a provede se analýza diagramu.

Regulační diagram může být považován za vyspělou formu průběhového diagramu pro případy, kde se nám podaří vyjádřit kapacitu procesu. Skládá se ze středové čáry, dvou kontrolních limitů (a někdy dvou varovných limitů uvnitř kontrolních limitů) a hodnot parametrů zájmu vnesené do diagramu, které reprezentují stav procesu. Osa x znázorňuje reálný čas. Jestliže všechny hodnoty parametru jsou uvnitř kontrolních limitů a mimo všechny zvláštní tendence, uvádí se, že proces je v regulovaném stavu. Jestliže hodnoty padnou vně kontrolních limitů nebo ukazují zvláštní situaci, proces je považován za neovladatelný. V takových případech následuje analýza pro zjištění příčiny a opravné akce.

Diagram příčin a následků, taktéž znám jako diagram „rybí kost“, byl vynalezen Ishikawou v padesátých letech v Japonsku. Prvně byl využit k vysvětlení faktorů ovlivňujících výrobu železa. Byl přičleněn k Japonské průmyslové standardní terminologii pro řízení kvality (Kume, 1989). Znázorňuje vztah mezi charakteristikou kvality a faktorem ovlivňující tuto charakteristiku. Návrh se podobá rybí kosti, charakteristika kvality zájmu je vyznačována na pozici „rybí hlavy“ a faktor ovlivňující charakteristiku je umístěn v místech „rybích kostí“. Zatímco bodový diagram popisuje v detailu dvojrozměrný vztah, diagram příčin a následků identifikuje v diagramu všechny faktory ovlivňující kvalitu.

4.2.1 Formulář (Checklist)

Formulář hraje důležitou roli ve vývoji softwaru. Formulář shrnuje klíčové body procesu a je efektivnější než zdlouhavý proces důkazů (Bernstein, 1992). V IBM Rochester proces vývoje softwaru obsahuje četné fáze, např. požadavky, systémovou architekturu, návrh na vysoké úrovni, návrh na nízké úrovni, vývoj kódu, testování, zapojení a konstrukce, testování součástí, testování systému a prvotní klientský program. Každá fáze má sadu úkolů a vstupních a výstupních kritérií. Formulář pomáhá vývojářům/programátorům zabezpečit, že všechny úkoly budou dokončeny a že pro každý úkol budou zohledněny všechny faktory.

Formuláře lze využít téměř všude. Formuláře, užívané denně celou vývojářskou komunitou, jsou vyvíjeny a přepracovávány na základě nahromaděných zkušeností. Formuláře jsou často částí technické dokumentace. Jejich denní užívání také udržuje činnost procesů.

Jiným typem formuláře je běžný chybový formulář, jehož částí je období zahájení procesu prevence proti chybám (DPP). DPP proces zahrnuje tři kroky: (1) analýza chyby k určení hlavní příčiny, (2) specializované týmy implementují navrhované akce a (3) období zahájení schůzky jako hlavní zpětná vazba. Tato schůzka je vedená

technickými týmy na začátku každé vývojové fáze. Hodnotící formulář běžných chyb a diskuze na téma jak jim zabránit je jedna z ohniskových oblastí.

Asi nejvýznamnější formulář v IBM Rochester v sekci vývoje softwaru je PTF formulář. PTF je zkratka pro Program Temporary Fix, který je fixně doručen zákazníkovi, když narazí na chyby v softwarovém systému. Chybný PTF je nevýhodný pro uspokojení zákazníka a měla by vždycky existovat silná oblast zájmu o toto téma uvnitř IBM. Díky realizování automatizovaného PTF formuláře a jiných položek IBM Rochester redukovalo už tak malé procento chyb na jednociferné číslo za celý rok. Pokud máme více než 20 milionů řádků kódu v AS/400 softwarovém systému a více než 250 000 licencí, je to velmi významný úspěch. Všimněte si, že PTF formulář je právě jednou částí zlepšení v oblasti kvality, což hraje důležitou roli v IBM Rochester.

PTF formulář byl vyvinut na základě analýzy mnohých zkušeností nahromaděných po mnoho let, mnohokrát přezkoumaných a korigovaných. Začalo to on-line formulářem, který se nyní vyvinul do automatizovaného expertního systému, který je hluboko zakořeněný v softwarovém procesu. Když je zavolán proces, expertní systém automaticky poskytne radu a krok za krokem vede softwarové vývojáře. Následuje výčet některých položek z formuláře:

- Q_n (Y,N) je kritické v tom, že opravuješ přesný problém popsaný v APAR/PTR popisu problému. Jestliže popis problému je nepřesný, nemůžeš si být jistý, že opravdu opravuješ daný problém, který ohlásil zákazník. Je problém popsaný v APAR/PTR přesný a opravuješ tento problém?
- Q_n (Y,N) Chtělo PTF změnu? Jestliže ano, UPOZORNĚNÍ: neměňte velikost. Je tam pravděpodobně pevně zapsaná reference na proměnnou offsetu a můžeš způsobit chybu.
- *a*. (Y,N) Jedná se o změnu běžné používaného elementu - CUE – Common Use Element?
- Každá námaha musí být znát na vývoji PTF, ale tak, že si nevyžádá změny na CUE.
- Jestliže je změna požadována, musí být udělána tak, aby pouze změněný modul potřeboval znovu zkompilovat a všechny ostatní moduly fungovaly správně s nezměněným CUE.
- Jestliže je požadována změna CUE, tak nemůže být změněn pouze modul, a v tom případě dočasné omezení by mělo být zváženo podle APAR náročnosti a dostupnosti obcházení.

Jestliže musíš mít stále v PTF jeden CUE
1. (Y,N) Byla tato oprava uvolněna a schválena CUE koordinátorem?
Kontaktujte XXX.

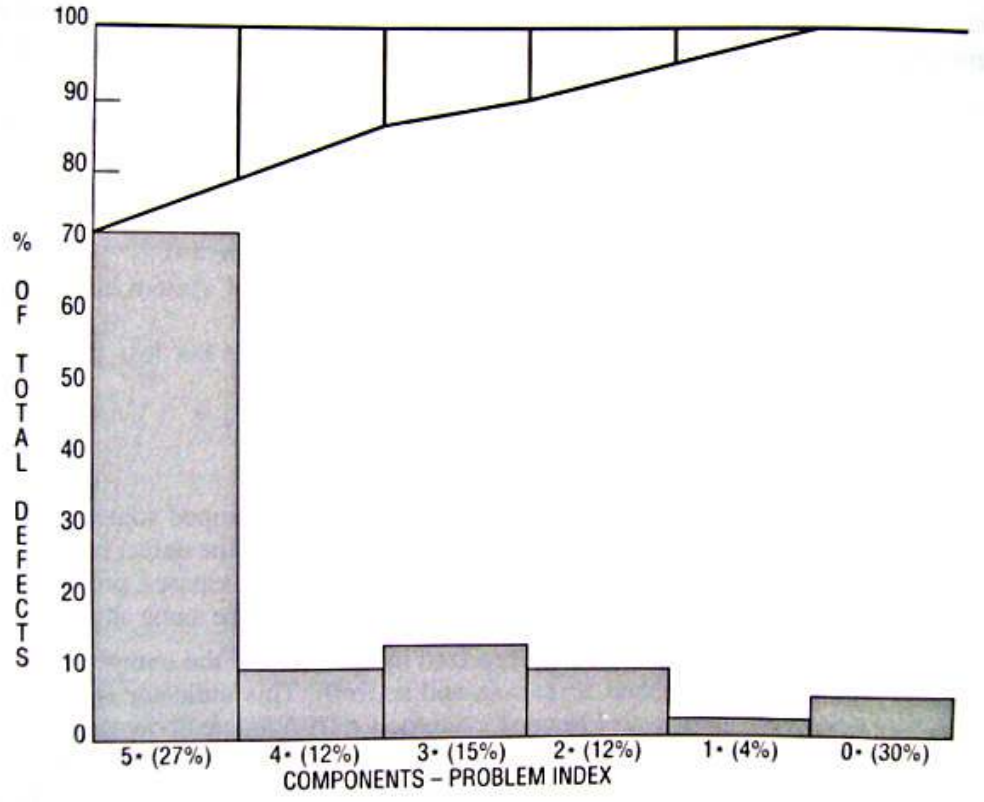
- *b*. (Y,N) Je měněná funkce používaná více než jedním modulem/ komponentou?

Aby sis pomohl, stanov jestli je daná funkce používána více než jedním modulem/ komponentou a udělej následující:
- Jestliže je to možné, zeptej se tvůrce na kód.
- Užij YYY příkazu k vyhledání výskytů. Ujisti se, že jsi našel všechny požadované verze pro uvolnění.
- Užij YYY1 příkazu, abys viděl, které jiné moduly používají tuto funkci.

Jestliže ano, pak
- 1. (Y,N) Je nutné znovu zkompilovat všechny moduly, které užívají tuto funkci pro toto PTF?
- 2. (Y,N) Umístil jsi všechny požadované moduly se správným PTR číslem?
- 3. (Y,N) UPOZORNĚNÍ: Měl jsi souhlas ode všech vlastníků komponent, že změna je bez problémů a nezpůsobíš problém? Pošli poznámku všem vlastníkům komponent, která vysvětluje změnu a která se ptá, jestli je to bez problémů. Když uvažuješ o tom, jestli změníš funkci a tato změněná část se dostane na úroveň budování, všechny budoucí PTF obsahující moduly, které užívají tuto funkci, dostanou tvůj kód a nebudou nic vědět, dokud jim to neřekneš. Toto může způsobit vadu v PTF.

4.2.2 Paretův diagram

Paretova analýza pomáhá tím, že pozná oblasti, které způsobují nejvíce problémů, což normálně znamená, že, když je opravíte, dostanete nejlepší návrat na investici. To je nejlépe využitelné v oblasti kvality softwaru, protože softwarové chyby nebo chybová hustota nikdy nepochází z jedné distribuce. Proto není překvapující, když vidíme Paretovy diagramy v literatuře o softwarovém inženýrství. Například Daskalantonakis (1992) ukazuje příklad Paretovy analýzy v Motorole pro rozpoznání hlavního zdroje požadovaných změn, které aktivovaly požadované akce v procesu. Grady a Caswell (1986) ukazují Paretovu analýzu softwarových chyb ve čtyřech softwarových projektech pro firmu Hewlett-Packard. Vrchol tvoří tři typy (nová funkce nebo různá požadovaná zpracování, existující data, která potřebují být organizována/uvedena rozdílně, a další pole dat pro potřeby uživatele), které odpovídají za víc než jednu třetinu chyb. Díky soustředění se na tyto převládající chybové typy, určení pravděpodobných příčin a zavedení zlepšujícího procesu, Hewlett-Packard byl schopný dosáhnout významných zlepšení v oblasti kvality. Obr. 4.2 ukazuje Paretovu analýzu příčin chyb na výrobku AS/400. Bylo zjištěno, že hlavní důvody k chybám u takového výrobku tvoří problémy rozhraní (INTF) a problémy s inicializací dat (INIT). Díky zaostření na tyto dvě oblasti přes návrh, realizaci, testovací procesy a řídící technické vzdělávání expertů, bylo zpozorováno významné zlepšení. Na obrázku je ukázána další chyba, která zahrnuje komplexní logické problémy (CPLX), národní jazykové problémy související s překladem (NLS), problémy vztažené k adresám (ADDR) a problémy s definicí dat (DEFN).



Obr 4.2: Paretova analýza chyb softwaru

Konstrukce Paretova diagramu

- Definování následku a shromáždění informací o všech možných příčinách
- Číselná kvantifikace jednotlivých příčin - tzv. četností
- Sestavení tabulky
- Absolutní a kumulativní absolutní četnosti
- Relativní a kumulativní relativní četnosti
- Sestrojení diagramu podle četností

Tento proces získávání Paretova diagramu se také někdy označuje jako Paretova analýza.

V oblasti jakosti je Paretův princip jedním z nejfrekventovanějších nástrojů, jak uvádí [9]. Umožňuje oddělit podstatné faktory od méně podstatných a ukázat, kam zaměřit úsilí při odstraňování nedostatků v procesu zabezpečování jakosti. Výsledek se zpravidla zpracovává do histogramu, seřazeného podle výšky sloupců a doplněného o Lorenzovu kumulativní křivku – tzv. Paretův graf.

Oblasti uplatnění Paretovy analýzy:
- organizace nákupu a výdaje materiálu
- snížení nákladů na údržbu a opravy
- snížení počtu opožděných dodávek
- redukce nepotřebných zásob
- snížení absence apod.

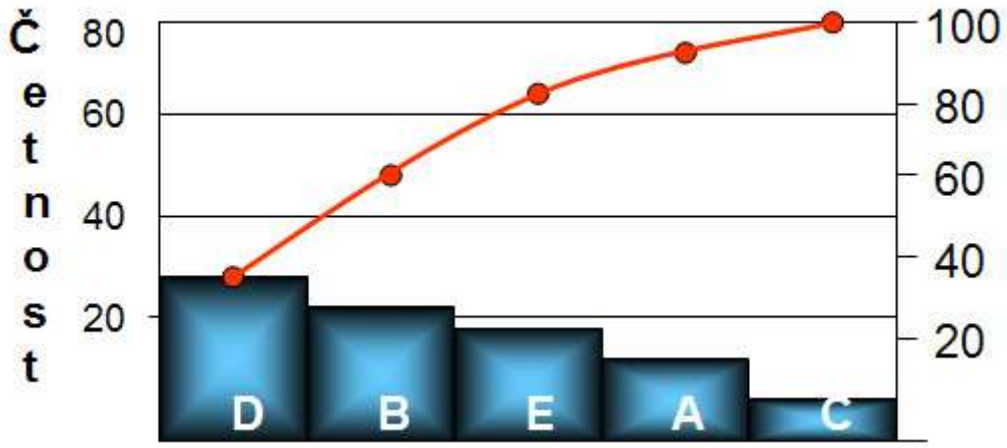
Postup při Paretově analýze:
1) Rozhodnutí, které položky budeme sledovat a jak provádět sběr dat. Určení kategorií dat (vady). Volba období, za které budeme provádět sběr dat, podle období, ve kterém se objevují problémy. Vlastní sběr dat.
2) Tabelace dat a výpočet kumulativních četností. Setřídíme údaje dle hodnot zvoleného ukazatele (např. dle počtu vad, dle výše nákladů spojených s jednotlivými vadami, dle počtu bodů přiřazených experty v brainstormingu jednotlivým příčinám neshod, apod.). Výpočet kumulativních součtů hodnot ukazatele. Princip výpočtu zobrazuje tabulka.
3) Sestavení Paretova diagramu
- osu x rozdělíme na stejné intervaly tak, že jejich počet odpovídá počtu druhů neshod ($\sum vad$);
- levou vertikální osu (y) označíme stupnicí od 0 do ∑čet (tj. celkový počet odhalených neshod);
- na pravou vertikální osu (y) vyznačíme stupnici relativních kumulovaných součtů od 0% do 100%;
- sestrojíme sloupcový graf (1 sloupec = druh vady, výška sloupce odpovídá četnosti daného druhu neshody);
- sestrojíme křivku kumulovaných četností v procentním vyjádření (tzv. Lorenzovu křivku). Tato křivka je spojnicí bodů, jejichž souřadnice odpovídají hodnotě pravé hranice daného intervalu a hodnotě kumulované četnosti v procentech odpovídající danému intervalu.

Příklad :

Neshoda	Absolutní četnost	Kumulovaná abs. četnost	Relativní četnost	Kumulovaná rel. četnost
D	26	26	32,10	32,10
B	21	47	25,93	58,02
E	18	65	22,22	80,25
A	11	76	13,58	93,83
C	5	81	6,17	100
Celkem	81		100	

Tabulka. 4.1: Tabulka pro konstrukci Paretova diagramu

Absolutní četnost - četnost jednotlivých příčin, tzn. jak se která podílí na následku
Kumulovaná abs. četnost – četnost, které se dosáhne postupným přičítáním jedné absolutní četnosti ke druhé
Relativní četnost – četnost získaná jako podíl jednotlivých četností vyjádřených v procentech
Kumulovaná rel. četnost – četnost získaná postupným přičítáním jedné relativní četnosti ke druhé



Obr. 4.3: Paretův diagram k příkladu

Po sestrojení diagramu následují tyto fáze:

1) Označení diagramu, zápis důležitých údajů (název diagramu, období sběru dat, název procesu, jméno tvůrce diagramu apod.).
2) Vyhodnocení Paretova diagramu. Na základě volby kritéria pro výběr „životně důležitých“ neshod stanovíme, na které neshody (obecné příčiny neodpovídající jakosti) je třeba zaměřit pozornost a provést jejich hlubší analýzu s cílem snížit počet neshodných výrobků. Volba kritéria se řídí zejména účelem analýzy a možnostmi (finančními, technickými, personálními) realizace nápravných opatření. Pro prvotní orientaci postačí kritérium 50% (tj. 50%-ní podíl kumulovaných součtů sestupně seřazených hodnot ukazatele). Nejpoužívanější kritérium je 80%.

pozn.: Chceme-li provést analýzu pečlivěji a jme-li schopni zaměřit své síly na více příčin, volíme tzv. kritérium průměrné hodnoty zvoleného ukazatele (např. průměrný počet neshod na jeden druh neshody). Hranici „životně důležitých" neshod stanovíme v tomto případě tak, že postupně porovnáváme hodnotu zvoleného ukazatele u jednotlivých příčin s průměrnou hodnotou tohoto ukazatele. Příčina, kde je hodnota daného ukazatele menší než průměrná hodnota, již nepatří do „životně důležitých" neshod.

Pro případ průměrného počtu neshod na jeden druh neshody platí:
průměrný počet neshod na jeden druh neshody ∑čet/ počet vad
vada1: a <> ∑cet / počet vad
vada2: b <> ∑cet / počet vad
vada3: c <> ∑cet /počet vad
atd.

Vady, u nichž je četnost výskytu neshody větší než průměrná hodnota (tj ∑cet / počet vad), zařazujeme do skupiny „životně důležitých" příčin.

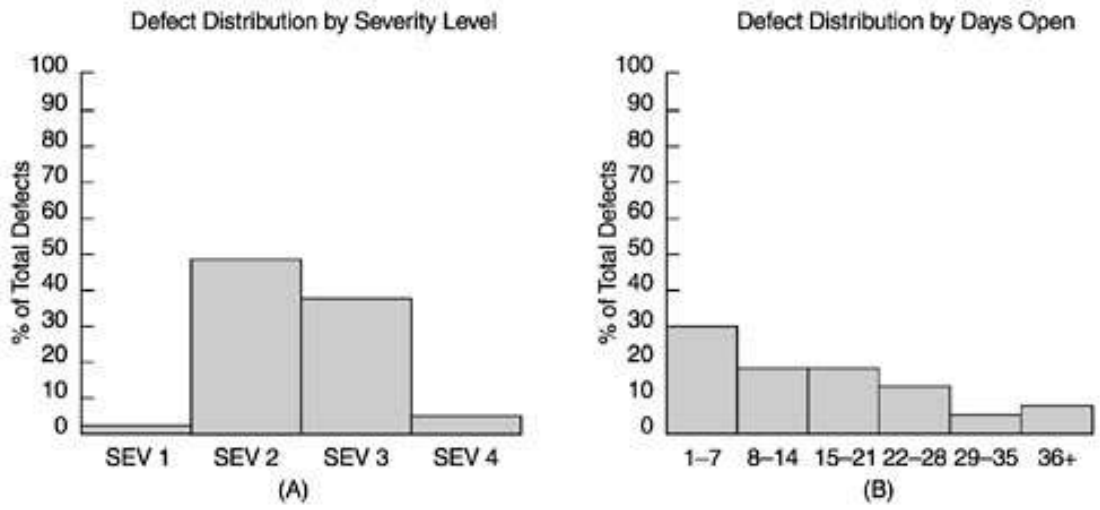
Následná analýza by měla být zaznamenána na omezení vlivu těchto příčin (tj. na jejich odstranění).

Celý postup analýzy by se měl realizovat vícenásobně. Volíme postupně různá zkoumání. Toto opatření nám umožní postihnout možné souvislosti mezi příčinami. Závěrečnou fází Paretovy analýzy je všestranné posouzení dílčích závěrů a zhodnocení jejich vzájemného působení. U „životně důležitých" nositelů negativních jevů je zapotřebí provést další hluboké šetření, odhalit příčiny a připravit nápravná opatření. K tomu je nutno vytvořit tým, v němž budou zúčastněni zástupci všech profesí, jejichž činnost s daným jevem souvisí.

Poř. číslo	Název vady	Počet vad (četnost)	Kumulativní četnosti	Relativní kumulovaná četnost (%)
1	vada1	A	a	a / (Σcet/100)
2	vada2	B	a+b	(a+b) / (Σcet/100)
3	vada3	C	a+b+c	(a+b+c) / (Σcet/100)
	atd.			
	Σvad	Σčet		

Tabulka 4.2: Tabulka s obecnými vztahy pro sestrojení Paretova diagramu

4.2.3 Histogram

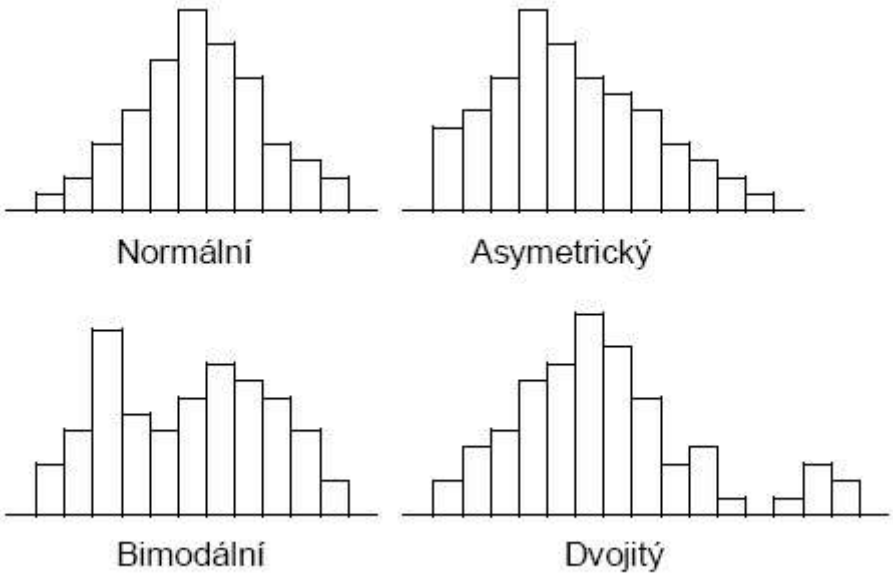


Obr. 4.4: Dva příklady histogramů

Obrázek 4.4 ukazuje dva příklady histogramů používané pro softwarové projekty a řízení kvality. Panel (A) ukazuje četnost chyb produktu podle stupně závažnosti (od 1 do 4, přičemž 1 značí největší stupeň a 4 nejmenší). Chyby s různým stupněm závažnosti se liší v jejich dopadu na zákazníky. Chyba v nejnižším stupni znamená pro zákazníky nepříjemnosti. Na rozdíl od toho chyba v nejvyšším stupni může způsobit nečinnost systému a ovlivňuje zákazníkovy obchody. Proto, pokud máme stejnou rychlost vzniku chyb (stejný počet chyb), histogram nám toho říká o kvalitě softwaru poměrně dost.

Panel (B) ukazuje četnost chyb během testování stroje vyjádřením počtu dnů, kdy byly chyby sledovány (1.-7. den, 8.-14. den, 15.-21. den, 22.-28. den, 29.-35. den a 36.+). Vyjadřuje časovou reakci ve stanovení chyb během testovacích fází, proto také znázorňuje množství vykonané práce.

Údaje v histogramu se zobrazují jako řada obdélníků o shodné šířce *(h)* a proměnné výšce. Šířka představuje určitý interval z celkového rozpětí údajů. Výška znázorňuje počet hodnot údajů (četnost) v daném intervalu. Seskupení měnících se výšek představuje rozdělení hodnot údajů.



Obr. 4.5: Typy histogramů

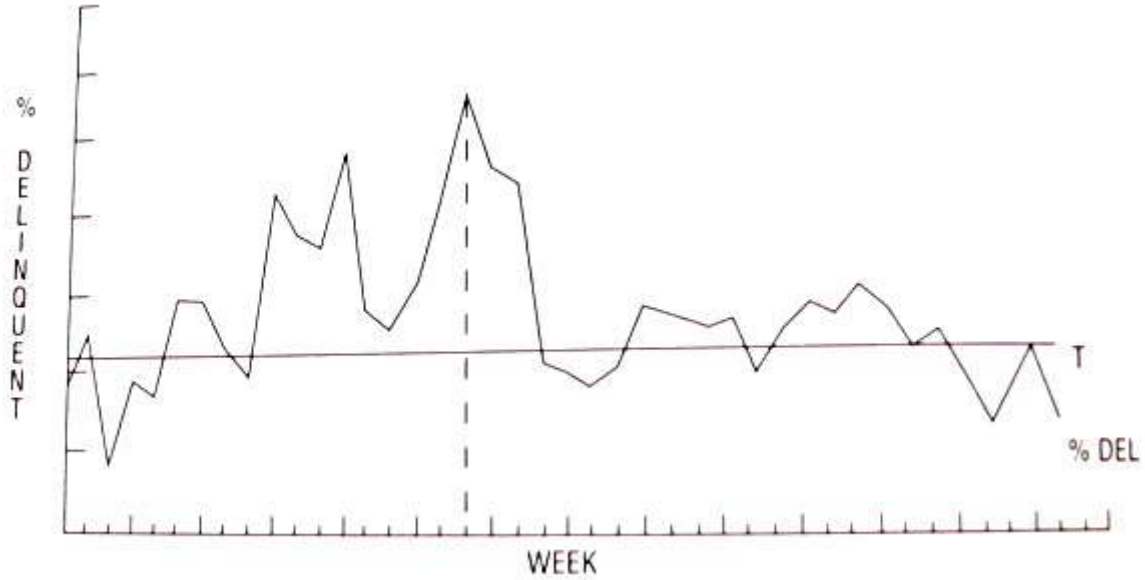
4.2.4 Průběhový diagram

Průběhový diagram je také hojně využíván pro řízení softwarových projektů – v knihách a časopisech o softwarovém inženýrství najdeme mnoho příkladů z reálného života.

Například týdenní počet chyb a chybové rezervy během jednotlivých fází při testování stroje mohou být znázorněny pomocí průběhového diagramu. Tyto diagramy slouží jako přehled kvality v reálném čase stejně dobře jako množství práce. Často jsou tyto diagramy srovnávány s modely plánování a data získána v dřívějším období mohou být tak touto reprezentací umístěny do správného hlediska. Další příklad sleduje procento oprav softwaru, které překročily časová kritéria na opravu. Cílem je zavčas zabezpečit doručení oprav zákazníkům.

Obr. 4.6 ukazuje průběhový diagram počtu chybových zprávy za týden, u kterých se již započalo s opravou, ale které ještě nebyly uzavřeny opravou podle časových kritérií. Horizontální osa označuje podíl nevyřízených oprav. Přerušovaná čára označuje čas, kdy speciální opravná akce odsunula boj proti vysokému podílu nevyřízených oprav. Pro každou chybnou nevyřízenou zprávu byla provedena náhodná analýza a odpovídá realizovaným akcím.

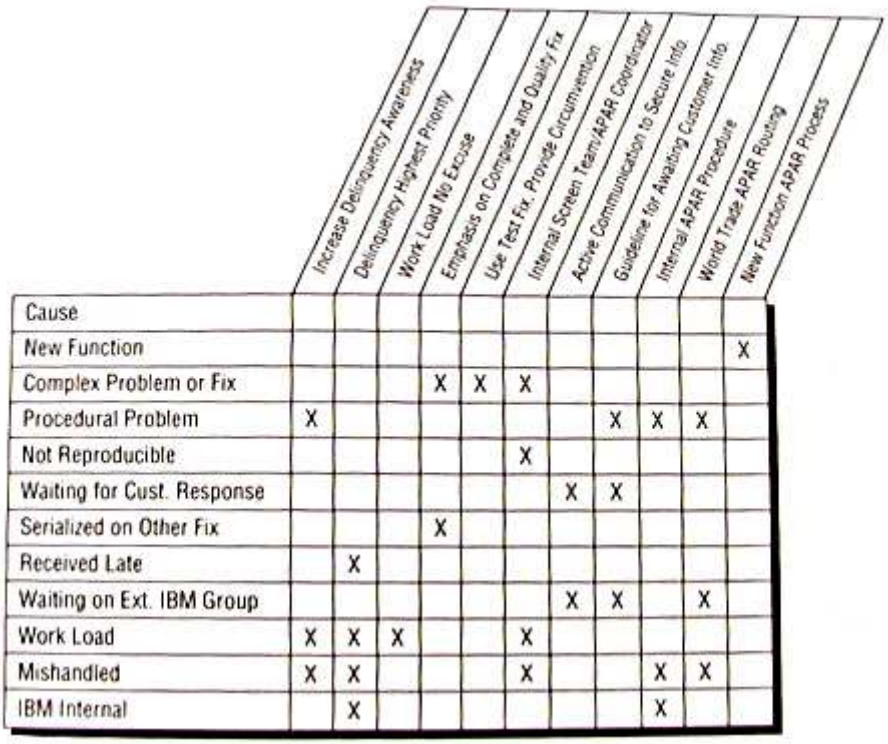
Na obr. 4.7 je ukázka příčin a realizovaných akcí. Výsledkem je snížení podílu nevyřízených chybových zpráv za měsíc. Podíl kolísal kolem plánu na 4 měsíce a dočasně se vymkl kontrole (zkratka APAR znamená Authorized Programming Analysis Report).



Obr. 4.6: Průběhový diagram % vyjádření týdenních nevyřízených oprav

Jiný typ průběhového diagramu je využit téměř ve všech organizacích zabývajících se vývojem softwaru. Plánované řízení je oblouk S, který označuje narůstající vývoj parametru zájmu vzhledem k času versus to, co bylo plánováno. V IBM Rochester parametry, které byly sledovány v každém projektu v aktuálním čase versus čas plánovaný, zahrnují:

- Dokončení zhodnocení návrhu
- Dokončení kontroly kódu
- Dokončení začlenění kódu
- Dokončení testování komponent úspěšně
- Ostatní parametry související s projektem a řízením kvality.



Cause	Increase Delinquency Awareness	Delinquency Highest Priority	Work Load No Excuse	Emphasis on Complete and Quality Fix	Use Test Fix, Provide Circumvention	Internal Screen Team/APAR Coordinator	Active Communication to Secure Info	Guideline for Awaiting Customer Info	Internal APAR Procedure	World Trade APAR Routing	New Function APAR Process
New Function											X
Complex Problem or Fix				X	X	X					
Procedural Problem	X							X	X	X	
Not Reproducible						X					
Waiting for Cust. Response							X	X			
Serialized on Other Fix				X							
Received Late		X									
Waiting on Ext. IBM Group							X	X		X	
Work Load	X	X	X			X					
Mishandled	X	X				X			X	X	
IBM Internal		X							X		

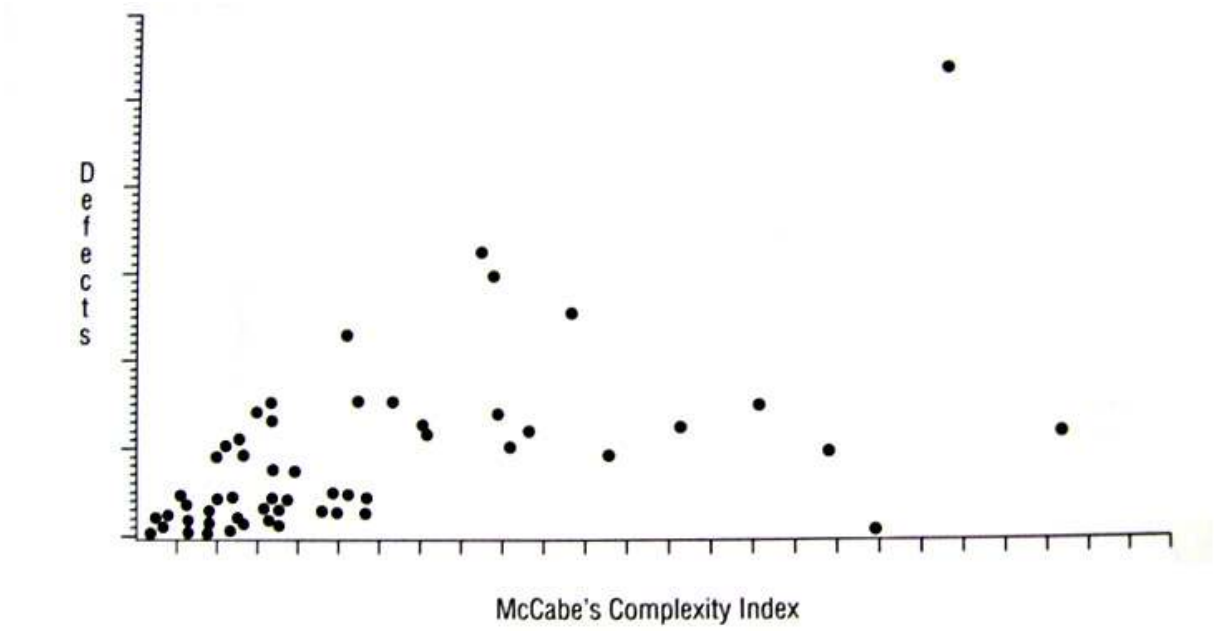
Obr. 4.7: Příčiny a zlepšující akce ke snížení nevyřízených oprav

4.2.5 Bodový diagram

V porovnání s ostatními nástroji je bodový diagram nejtěžší k aplikaci. Obvykle souvisí s výzkumnou prací a vyžaduje přesná data. Často je použit spolu s jinými technikami jako jsou korelační analýza, regresní analýza a statistické modelování.

Obr. 4.8 ukazuje příklad bodového diagramu znázorňující vztah mezi McCabeovým indexem složitosti a mírou chybovosti. Každý bod reprezentuje modul programu, kde osa x znázorňuje index složitosti a osa y míru chybovosti. Přestože složitost programu může být měřena nejdříve po dokončení programu a chyby jsou objevovány v průběhu dlouhé časové periody, pozitivní vzájemný vztah mezi dvěma veličinami nám dovolí použít složitost programu k předpovídání míry chybovosti. Kromě toho můžeme snížit složitost programu, (měřeno McCabeovým indexem), a tím snížit šanci k vytvoření chyb.

Snížení složitosti nám umožňuje také programy lépe udržovat. Některé týmy z operačního systému AS/400 si osvojily tento přístup za svoji strategii na zlepšení kvality a údržbu. Moduly programu s vysokým indexem složitosti jsou cíle pro analýzu a pro možný modulový rozpad, shrnutí, mezimodulární vymazání a jiné akce. U modulů s nízkým indexem složitosti ale vysokým počtem chybových zpráv je samozřejmě nutno vymazat chybný návrh nebo jeho realizaci a měl by být prozkoumán.

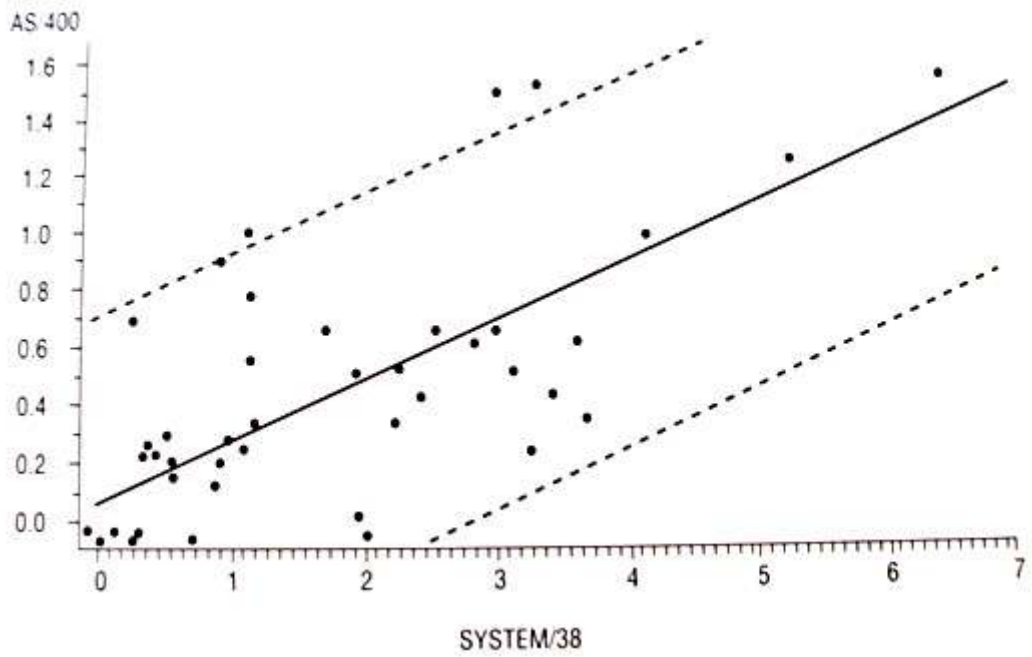


Obr. 4.8: Bodový diagram závislosti složitostí programu a stupněm chybovosti

Další příklady bodových diagramů zahrnují vztahy mezi chybami, indexy kvality stejných komponent v současných a předchozích vydáních, vztahy mezi rychlostí testování chyb a rychlostí vzniku chyb atd. Díky zkoumání těchto vztahů jsme získali pohledy na řízení kvality softwaru.

Ve vývoji softwaru je možnost opětovného použití možná nejdůležitější faktor ve zlepšení produktivity. Kvalita softwaru je nicméně často omezována utajenými chybami nebo nedostatky v návrhu předchozího kódu. Pro softwarový systém AS/400 některé komponenty byly vyvinuty díky opětovnému použití komponent existujících výrobků na platformě IBM System/38. Ke zkoumání vztahů rychlosti vzniku chyb ve znovu použitých komponentách mezi dvěma platformami používáme bodový diagram. Obr. 4.9 ukazuje příklad jednoho výrobku. V obrázku každý bod reprezentuje komponentu, osa x udává její rychlost vzniku chyb na platformě System/38 a osa y znázorňuje rychlost vzniku chyb na platformě AS/400.

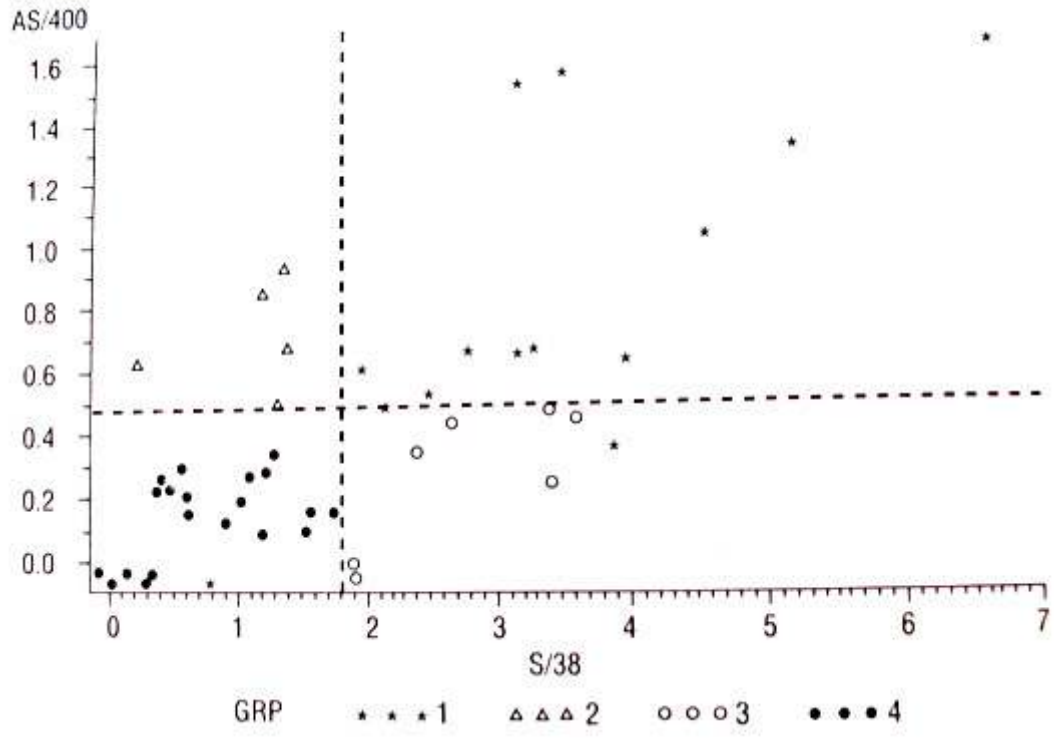
Ačkoliv existují změny a úpravy ve výrobku AS/400 a byly zařízeny dodatečné revize a testování, přesto vzájemný vztah (0.69) je poměrně silný. Vidíme, že obě dvě křivky jsou lineární (diagonální křivky), a 95%-ní interval spolehlivosti (oblast mezi dvěma přerušovanými čarami).



Obr. 4.9: Vztah rychlosti vzniku chyb mezi dvěma platformami

Dále pokračujeme v rozdělení bodového diagramu do čtyř kvadrantů podle střední hodnoty rychlosti vzniku chyb u komponent na platformě AS/400 a System/38 (Obr. 4.10). Toto rozdělení poskytuje různé analýzy a zlepšující strategie k tomu, aby mohly být použity různé druhy komponent.

- Komponenty v pravém horním kvadrantu (označeny hvězdičkami) jsou trvale problémové komponenty. Skutečnost, že tyto komponenty utrpěly navzdory letům stárnutí na platformě System/38 naznačuje, že na významné akce (jako například zkoumání návrhové struktury, přepsání chybám náchylných modulů) je nutno brát ohled.
- Komponenty v levém horním kvadrantu (označeny trojúhelníky) jsou komponenty s nízkou rychlostí vzniku chyb v System/38 ale vysokou v AS/400. Strategie zlepšení by se měla soustředit na přirozené zvětšování AS/400 a na vývoj procesu.
- Komponenty v pravém dolním rohu (označeny prázdnými kolečky) jsou komponenty, které mají vysokou rychlost vzniku chyb v System/38 ale nízkou v AS/400. Změny provedené u komponent AS/400 a akce obdržené vývojem by měly být prozkoumány.
- Komponenty v levém dolním rohu (označeny plnými kolečky) jsou komponenty, které mají nízkou rychlost vzniku chyb v obou platformách. Ohnisko analýzy by mělo být v jejich použití v případě, že jsou použity v návrhové struktuře ve značné míře.



Obr. 4.10: Seskupování znovu použitých komponent podle rychlosti vzniku chyb

4.2.6 Regulační diagram

Regulační diagram je nejsilnější nástroj pro dosažení statistického řízení procesu (SPC statistical process control). Ve vývoji softwaru je nicméně těžké použít regulační diagram ve formálním SPC způsobu. Zkusit definovat kapacitu procesu je velmi těžký téměř nemožný úkol ve vývoji softwaru. Kapacita procesu je základní odchylka procesu ve vztahu k daným limitům. Nejmenší odchylka procesu znamená nejlepší kapacitu procesu. Chybovou částí jsou části, které jsou zaznamenány s hodnotami specifických parametrů mimo dané limity. Menší odchylky procesu jsou tedy známkou lepší kvality produktu.

Ve statistických termínech je kapacita procesu dána vztahem:

$$C_P = \frac{|USL - LSL|}{6sigma},$$

kde *USL* a *LSL* jsou horní a dolní limit, *sigma* je standardní odchylka procesu a *6sigma* reprezentuje celkovou odchylku procesu.

Jestliže je k některým charakteristikám připojena jednostranná specifikace, kapacitní index může být definován jako:

$$C_P = \frac{|USL - u|}{3sigma},$$

kde *u* znamená procesový průměr, nebo

$$C_P = \frac{|u - LSL|}{3sigma}$$

V prostředí průmyslové výroby, kde je denně vyráběno mnoho částí, odchylka a kapacita procesu může být počítána pomocí statistických vzorců a regulační diagram může být použit na bázi reálného časového úseku. Software se liší v průmyslové výrobě v různých hlediscích a takové odlišnosti činí téměř nemožné odhadnout kapacitu procesu v organizaci zabývající se vývojem softwaru.

- Specifikace pro většinu stanovených metrik téměř nesouvisí nebo souvisí špatně se skutečnými potřebami zákazníků. Dobře definované specifikace založené na požadavcích zákazníka, které mohou být vyjádřeny ve vzorcích nebo metrikách chybí pro prakticky všechny softwarové projekty (více aktuální, extrémně těžké pro odvození).
- Software je vývoj (ne výroba) a skládá se z různých fází činnosti (architektura, návrh, kód, testování atd.) a je za potřebí hodně času k dokončení projektu.
- Spolehlivé modely jsou takové, které mohou kvantitativně určit vztahy mezi různými aktivitami v každé fázi vývoje ve stupni kvality, kdy je vyvíjen koncový produkt.
- Uvnitř organizace na vývoj softwaru jsou často využívány vícenásobné procesy.
- Technologické a vývojové procesy jsou rychle měněny.

Navzdory těmto názorům jsou regulační diagramy užitečné pro zlepšení kvality softwaru – pokud jsou použity s volnějšími pravidly. To znamená, že regulační diagram v softwaru nelze použít ve formálních termínech statistického řízení procesu a kapacity.
Přesněji řečeno – je použit jako nástroj pro zvýšení konzistence a stability. V mnoha případech není použit na bázi reálného časového úseku a je vhodně nazýván – pseudo regulační diagram.
Existuje mnoho typů regulačních diagramů. Nejběžnějším je *X* a *S* diagram, které znázorňují průměrné hodnoty a standardní odchylky. Existují také střední diagramy, diagramy pro jednotlivce, *p* diagram pro neodpovídající podíl, *np* diagram pro neodpovídající číslo, *c* diagram pro neshodné číslo, *u* diagram pro neodpovídající čísla za jednotku atd. Nejvhodnější pro softwarové aplikace jsou *p* diagramy, pokud jsou zahrnuty procenta, a *u* diagram v případě, že je použita rychlost vzniku chyb. Kontrolní limity jsou počítány jako hodnota parametru zájmu (například *X* nebo *p*) plus/mínus tři standardní odchylky. Můžeme také zvýšit závislost diagramu přidáním dvou varovných limitů, které jsou počítány jako hodnota parametru plus/mínus dvě standardní odchylky. Tak jako se liší výpočet standardní odchylky díky typům parametrů, vzorce pro kontrolní limity (a varovné limity) se také liší. Například kontrolní limity pro rychlost vzniku chyb (*u* diagram) mohou být vypočítány jako:

$$\text{Horní limit} = \mu + 3\sqrt{\frac{\mu}{n}}$$

$$\text{Dolní limit} = \mu - 3\sqrt{\frac{\mu}{n}},$$

kde μ, je narůstající rychlost vzniku chyb získaná z předchozích dat (vážený průměr jednotlivých rychlostí) a *n* je průměrný počet řádků zdrojového kódu.

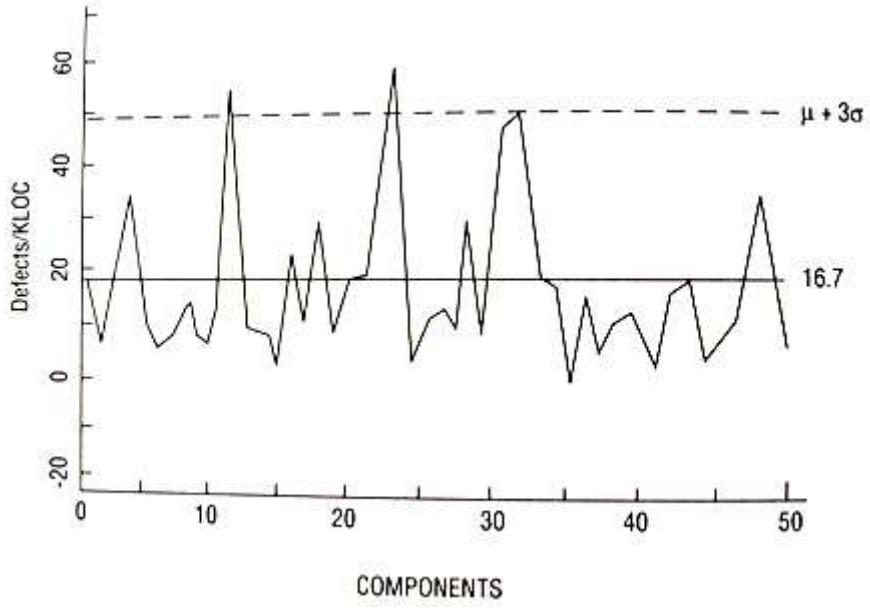
Kontrolní limity vyjádřeny v procentech (např. metrika pro účinnost) mohou být vypočítány:

$$\text{Horní limit} = p + 3\sqrt{\frac{p(1-p)}{n}}$$

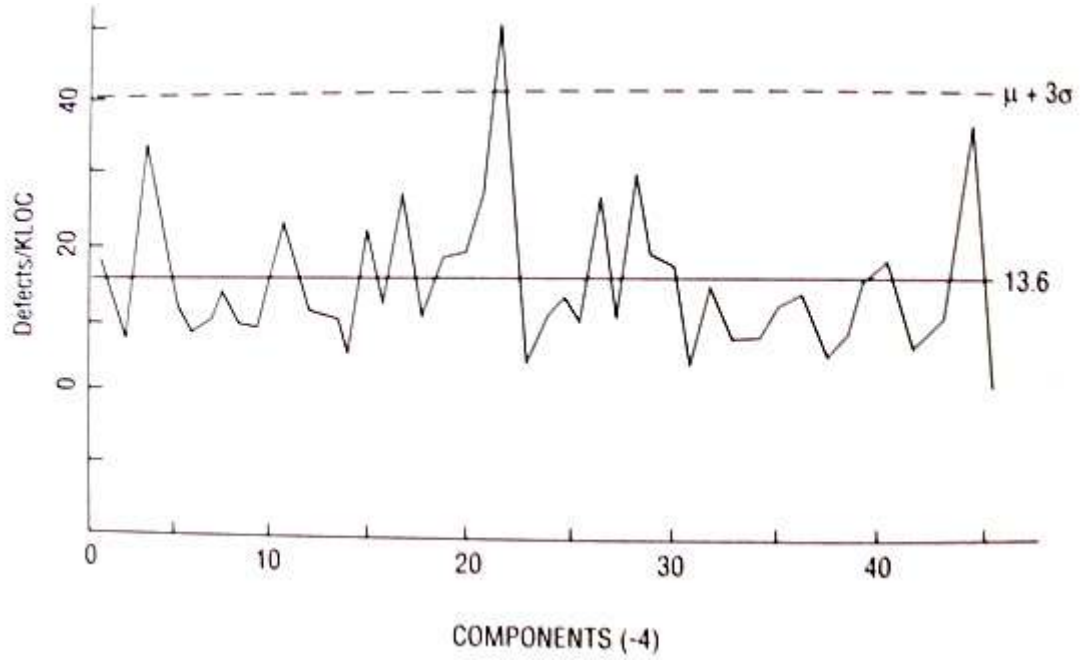
$$\text{Dolní limit} = p - 3\sqrt{\frac{p(1-p)}{n}},$$

kde p je vážený průměr z procent získaných z předchozích dat a n je průměrná velikost vzorku.

Různé metriky zabývající se procesem vývoje softwaru mohou být zmapovány regulačními diagramy, například kontrola chyb pomocí KLOC, testování chyb pomocí KLOC, fáze účinnosti, index řízení neopravených chyb atd. Obr. 4.11 ukazuje příklad pseudo regulačního diagramu, který zpracovává testování chyb pomocí KLOC u komponent pro projekt IBM Rochester, ve kterém byly označeny komponenty náchylné k chybám pro budoucí důkladnou analýzu. V tomto případě použití regulačního diagramu zahrnovalo více než jednu iteraci. V první iteraci jsou identifikovány komponenty s rychlostmi vzniku chyb vně kontrolní limity (tedy poměrně vysokými). (Měli bychom si povšimnout, že v tomto příkladu je regulační diagram jednostranný s pouze horním kontrolním limitem). V druhé iteraci komponenty, které byly identifikovány dříve, jsou odstraněny a data jsou znovu vyhodnoceny s novými kontrolními limity (Obr. 4.12). Tento proces „loupání cibule“ dovolil seznámení s další sadou potenciálních komponent náchylných k chybám, některé mohou být označeny jako počáteční sady diagramů. Tento proces může pokračovat několika dalšími iteracemi. Prioritou zlepšující akce, se kterou třeba souvisí dostupnost zdrojů, může být také určení pořadí iterací, ve kterých jsou definovány problémové komponenty. V každé iteraci body, které se vymknou kontrole, by měly být odstraněny z analýzy pouze v tom případě, pokud se porozumělo jejich příčinám a byly učiněny kroky, aby se předešlo jejich dalšímu vzniku.

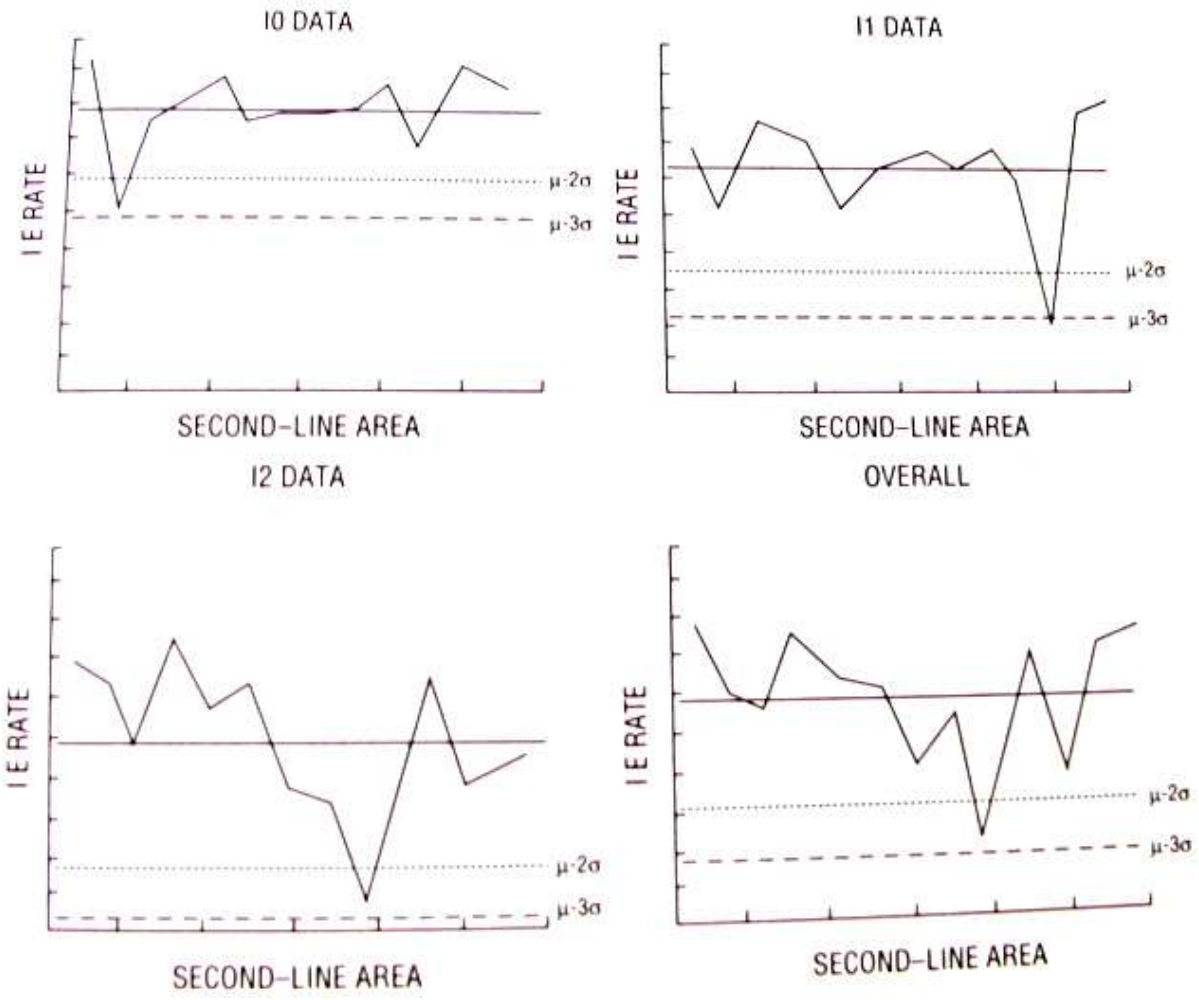


Obr. 4.11: Pseudo regulační diagram při testování rychlosti vzniku chyb – 1. iterace



Obr. 4.12: Pseudo regulační diagram při testování rychlosti vzniku chyb – 2. iterace

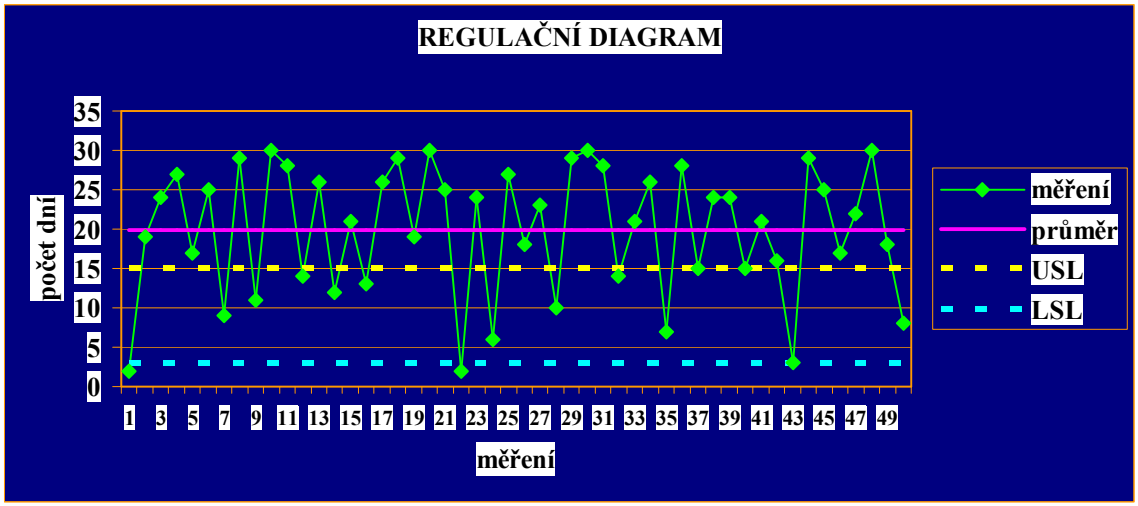
Další příklad, také z IBM Rochester, je efektivita kontrol při sestavování diagramů v oblasti několika fází hodnocení a kontrol, jak ukazuje Obr. 4.13. Efektivita je relativní měření v procentech, kdy v čitateli je počet odstraněných chyb ve vývojové fázi a ve jmenovateli je celkový počet chyb nalezených v této fázi plus chyby nalezené později. Na obrázku každý bod reprezentuje efektivitu kontroly a druhořadý stupeň oblasti vývoje. Čtyři panely ukazují vysoký stupeň kontroly návrhu (I0), nižší stupeň kontroly návrhu (I1), kontrola kódu (I2) a celková efektivní kombinace všech tří fází (vpravo dole). Oblasti s nízkou efektivitou (označují varovné a kontrolní limity) stejně jako s vysokou efektivitou byly prostudovány a určeny přispívající činitelé. Jako výsledek těchto regulačních diagramů a dodatečné práce byla zlepšena kontrola efektivity vývojového procesu AS/400.



Obr. 4.13: Pseudo regulační diagram kontroly efektivity

Konstrukce regulačního diagramu při regulaci měřením:

- Provede se 25 výběrů o konstantním rozsahu
- Z každého výběru se vypočítají statistické charakteristiky
- Vypočítá se průměr těchto statistik
- Vypočítají se regulační meze
- Meze se vyznačí do regulačního diagramu
- Diagram se použije při řízení procesu

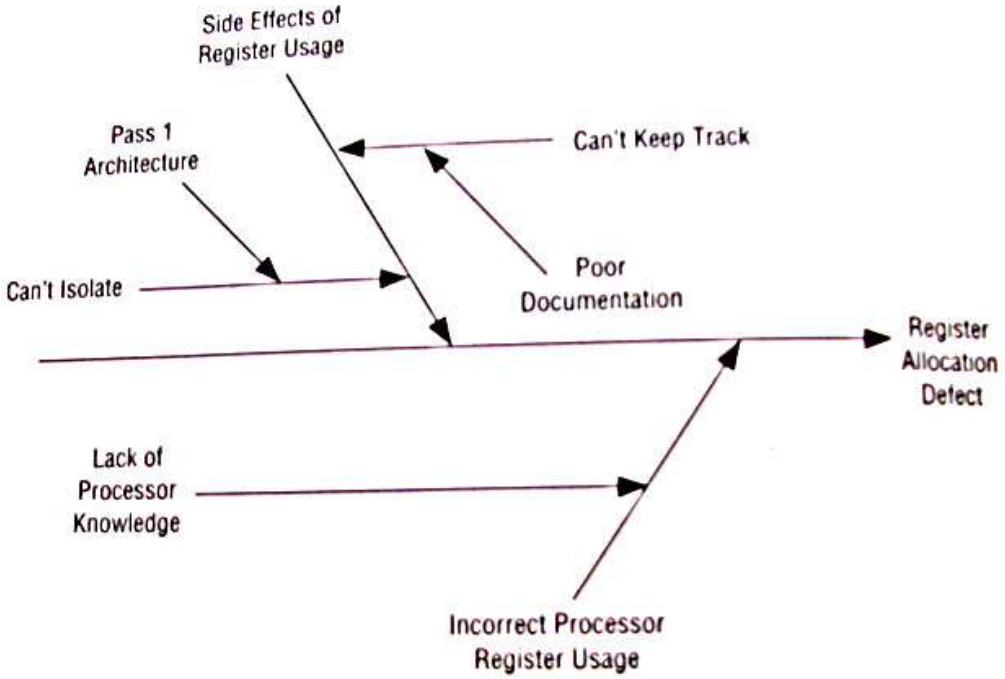


Obr. 4.14: Jednotlivé parametry charakteristické pro regulační diagram

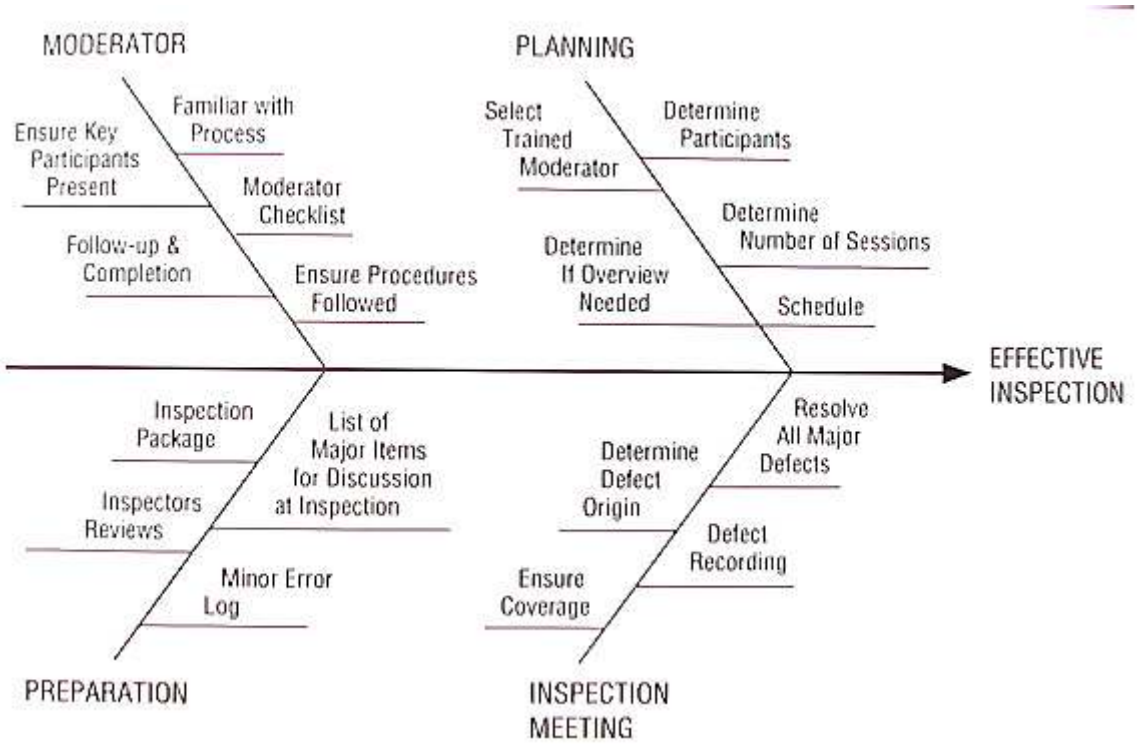
4.2.7 Diagram příčin a následků

Diagram příčin a následků, kvůli svému tvaru nazývané též „rybí kost“, je asi ten nejméně používaný nástroj pro vývoj softwaru. Mezi několika diagramy, se kterými jsme se setkali, asi nejlepší příklad je jeden určený Gradym a Caswellem (1986) v projektu firmy Hewlett-Packard. Vývojový tým, v jehož zájmu bylo zlepšení kvality, nejdříve použil Paretův diagram a shledal, že chyby vyvolané přidělením registru byly v jejich projektu nejčetnější. S pomocí diagramu příčin a následků pak provedli „brainstormingovou“ schůzi, ve které projednávali daný problém. Jak ukazuje Obr. 4.15, shledali stranu týkající se použití registru a chybné použití registru jako dvě první příčiny. Nakonec obě byly způsobeny nedostatečnými znalostmi operací v registrech. Po tomto shledání HP skupina učinila před dalšími projekty aktivní kroky k zajištění řádného cvičného registru a registru týkajícího se dokumentace a procesorů.

Obr. 4.16 ukazuje další příklad diagramu „rybí kost“, který se týká klíčových faktorů efektivních kontrol. Jako diagram vzali část zabývající se materiálním vzděláním procesu pro projekt, se kterým jsme již měli zkušenosti.



Obr. 4.15: Diagram příčin a následků



Obr. 4.16: Diagram příčin a následků u kontroly návrhu

4.2.8 Shrnutí

V nedávných letech existovaly v softwarovém průmyslu neustále se objevující trendy používat vědecké metody k dosažení zpřesnění v plánovaných softwarových projektech. Mnoho statistických nástrojů a nástrojů k řízení kvality bylo široce použito v průmyslové výrobě a získaly přijetí v prostředí vývoje softwaru. V této sekci jsme se zabývali možným použitím sedmi základních nástrojů v příkladech z reálného života. V mnoha případech analýza založená na těchto nástrojích vynesla významné výsledky ve zlepšení softwarových procesů.

Možnost aplikace jednotlivých nástrojů se různí. Zatímco některé mohou být hojně použity v podstatě denně (jako například formulář (checklist) nebo průběhový diagram), jiné slouží pouze k prvotní orientaci (jako třeba diagram příčin a následků). Jednotlivé nástroje mohou být použity společně (s ostatními) nebo s jinou vyspělou metodou. Např. Paretův diagram, diagram příčin a následků a bodový diagram mohou být použity dohromady pro určení hlavního problému a jeho hlavní příčiny. Regulační diagram může být použit v případě sledování stability procesu nebo k určení, zdali skutečně došlo ke zlepšení po uskutečnění opravných akcí.

Sedm základních nástrojů opravdu jsou základními nástroji. Nemusí být přitažlivé z hlediska výzkumníků. Jejich skutečné hodnoty leží v důsledném a vše prostupujícím použití vývojovými týmy pro zlepšení procesu. Jejich dopad může být enormní, speciálně pokud jsou zautomatizovány a staly se hluboce zakořeněné v procesu pro vývoj softwaru, jak to bylo ukázáno na příkladu automatického fixního checklistu v IBM Rochester.

A konečně, Ishikawových sedm základních nástrojů jsou v poslední době nazývány jako sedm starých nástrojů nebo sedm nástrojů řídících kvalitu. V posledních letech se objevilo sedm nových nástrojů řídících/plánujících kvalitu (které jsou většinou kvalitativní), kterými jsou: diagram příbuznosti, poměrný diagram, stromový diagram, metrický diagram, maticový diagram, orientovaný diagram (v síťovém grafu šipkový diagram) a PDPC (Process decision Program Chart). Ačkoliv diskuze o těchto nových nástrojích již není v zadání, nemohu si odpustit zmínku, že tyto nástroje jsou čím dál tím častěji používány v řízení softwaru.

5 Specifikace kritérií pro srovnání

Funkce popsaných sedmi nástrojů:

Formulář (Checklist) umožňuje:
- Zaznamenávat informace o jakosti SW
- Utřídit je, aby poskytly jasný obraz o situaci
- Umožňuje další zpracování
- Pro konstrukci neexistuje standardizovaný formát
- Konstrukce je vždy podřízena účelu
- Užíván také jako prevence proti chybám

U některých aplikací je speciální checklist dodáván k programu a pokud dojde k chybě programu je automaticky kontaktován vývojový tým a na základě checklistu je veden krok za krokem k opravě programu.
Na principu checklistu funguje v Microsoft Windows – Poradce při potížích, kdy je uživatel dotazován, jaký má problém a krok za krokem je naváděn na možné řešení, což v případě nalezení a odstranění problému vede k vyšší spokojenosti zákazníka a tím i ke zvýšení kvality softwaru. Jiná věc je, nakolik je pomoc takového druhu účinná, kolik % ze zákazníků zde skutečně najde pomoc.

Paretův diagram bývá často kombinován s diagramem příčin a následků, neboť oba se zabývají hledáním příčin určitého stanoveného problému. V diagramu příčin a následků se obvykle používá jako první, pomocí brainstormingu se určí všechny příčiny, které působí na následek (tedy určitý problém) a do Paretova diagramu se pak vyznačí tyto příčiny od nejčetnější po nejméně četnou. Následně se sestrojí Lorenzova křivka a určí se 20% příčin, které mají za následek 80% chyb. Vývojový tým by se dále měl soustředit především na odstranění těchto 20% příčin daného problému, což vede k efektivnímu zvýšení kvality produktu.

Histogram znázorňuje četnost měřenou pro veličiny stejného druhu, např. četnost chyb v jednotlivých dnech, kdy probíhalo testování produktu. Pokud nedochází k postupnému snižování počtu chyb úměrně s narůstajícími dny testování, je to pro vývojový tým signál pro zahájení opatření, které by vedlo k nápravě. Pokud se tak stane, je to známka zvýšení kvality.

Průběhový diagram použijeme, pokud sledujeme určitou vlastnost v určitém časovém intervalu. Během tohoto intervalu provádíme pravidelná měření, naměřené hodnoty vneseme do grafu a danou charakteristiku porovnáme se stejným diagramem, který byl sestaven v minulosti. Pozorujeme novou tendenci a v případě výskytu anomálií je potřeba se zabývat možným vznikem problému. Následné odstranění problému vede opět ke zvýšení kvality.

Bodový diagram – určuje závislost mezi dvěma faktory (např. počtem chyb a složitostí programu), podle Pearsonova koeficientu závislosti je možno předvídat počet chyb a pokud se vývojový tým soustředí na snižování složitosti programu, a tím i na snížení počtu chyb, vede to ke zlepšení kvality produktu. Dále je možno použít bodový diagram k určení vzájemného vztahu mezi dvěma programy – v případě, že se jeden program částečně vyvinul z druhého (samozřejmě v rámci jedné firmy), pak jde o důležitý faktor - znovupoužitelnosti. Pokud se použije to, co se v minulém projektu povedlo, kde jsou již chyby odladěny, dá se předpokládat, že nový produkt bude mít vyšší kvalitu.

Regulační diagram – umožňuje oddělit v procesu vážné příčiny od náhodných. Někdy může být také vyspělou formou průběhového diagramu. Sledujeme celý proces, vneseme do grafu naměřené veličiny, které jsme sbírali po určitou dobu, vypočítáme horní a dolní meze a určíme, zda-li se proces nevychyluje mimo dané limity. Menší odchylky procesu jsou známkou lepší kvality produktu.

Podniky mohou využít kombinace těchto nástrojů : regulačního diagramu pro celkové sledování procesu (např. vývoje produktu), v případě vychýlení mimo meze následuje hledání příčiny pomocí diagramu příčiny a následků, a konečně Paretova analýza pro určení hlavních příčin problému. Místo regulačního diagramu je v některých případech vhodnější bodový diagram.

Pro sestavení jednotlivých diagramů je zapotřebí mít k dispozici vhodná data - nejlépe hodnoty interních nebo externích metrik, jak popisuje [10].

Interní metriky:

- *Prac* – spotřeba práce, obvykle v člověkoměsících
- *Doba* – doba provádění
- *Prod* – jednotky délky za člověkoměsíc (řádky za měsíc)
- *Fail* – počet selhání za týden (den)
- *Prod* – produktivita, počet jednotek délky vytvořených za člověkoměsíc.
- *team(t)* – velikost týmu (počet osob) v čase *t*, měřeno od začátku prací. Tato metrika umožňuje postupné zpřesňování odhadů pracnosti a doby řešení během vývoje projektu.
- *Team* – průměrná velikost týmu
- *Fail(t,p)* – počet selhání systému /části *p* detekovaných při testování či provozu v čase *t*. Při provozu hlásí selhání zákazníci. Obvykle se udává po dnech nebo týdnech. Tato metrika je důležitou mírou kvality. Při inspekcích je hodnota metriky *Fail* dána počtem chyb zjištěných při inspekci.
- *Defect(t,p)* – počet míst v programech, která bylo nutno opravit pro odstranění selhání nebo pro nápravu selhání v čase *t* (den, týden) v části *p*, normalizovaný pro 1000 řádků (1000 _ počet defektů_délka)
- *Defect1(e1,e2,t,p)* – počet defektů v části *p* vzniklých v etapě řešení *e*1 a zjištěných v etapě *e*2 v době *t*. Tato metrika a metriky z ní odvozené jsou účinnou mírou efektivnosti inspekcí a testů. Důležitá externí metrika pro vyhodnocování kvality produktu
- *Satisf(t)* – průměrná míra spokojenosti zákazníků včase *t*. Spokojenost se udává ve stupnici 1 až 5 (nejlepší). Lze vyhodnocovat trendy. Existují metody odhadu vývoje úspěšnosti produktu na trhu z aktuálních hodnot metrik *Satisf* (Babich, 1992).
- *DobaOpr* – průměrná doba opravy selhání.
- *Zmeny(f,t)* – počet změněných míst souboru *f* včase *t* (týdnu/dni). Tato metrika se snadno zjišťuje a její trendy mohou v průběhu prací poskytnout cenné informace.
- *MTBF(t)* – (Mean Time Between Failures): střední doba mezi poruchami (v určitém období, např. týdnu, *t*).

Vzhledem k tomu, že data pro interní metriky jsou většinou známy pouze týmu během vývoje na základě sledování vývojových procesů, budeme se orientovat spíše na externí metriky, které se dají zjistit na hotovém produktu.

Externí metriky:

- *Del* – délka produktu v řádcích. U programů se nepočítají komentáře. *Del* programů se někdy udává v lexikálních atomech. Do metriky *Del* se někdy nezahrnují deklarace proměnných a záhlaví podprogramů.
- *Srnd* – rozsah slovníku operandů. Tato metrika se týká programů. Operand je buď konstanta (např. celé číslo 10, nebo řetězec znaků „ xyz“, v terminologii programování literál), nebo proměnná, např. *x*. *Srnd* je pak počet logicky odlišných operandů vyskytujících se v programu.
- *Nrnd* – počet výskytů operandů v programech
- *Noper* – Počet výskytů delimiterů a znaků operací a podprogramů v programech.
- *Soper* – rozsah slovníku operací a delimiterů. Tato metrika udává, kolik program obsahuje významem různých znaků operací (*, C, atd.), jmen podprogramů (sin, tan, put), delimiterů (: **if begin real** atd.).
- *Users* – Maximální počet uživatelů, pro které je systém plánován.
- *McCabe* – počet podmíněných příkazů (if), příkazů cyklu (for, while, . . .) a přepínačů (case) v programu. Metrika *McCabe* je dobrým indikátorem složitosti programů. Jejím nedostatkem je, že není citlivá na hloubku a způsob vloženosti podmíněných příkazů (tvar rozhodovacího stromu (případně lesa).
- *In, Out, Qer, File, Filee* – složitost příkazů vstupu, výstupu, dotazů na terminál a operací se soubory interními a se soubory společnými s jinými aplikacemi. U databází složitost SQL dotazů. Tyto metriky se používají v odhadech pracnosti a doby řešení v metodě funkčních bodů.
- *FanIn, FanOut* – míry indikující složitost rozhraní tříd, modulů a aplikací. *FanIn* udává počet logicky různých typů dat vstupujících do dané entity (např. modulu).
- *FanOut* je obdoba *FanIn* pro vystupující datové toky.
  Často se používá metrika $Fan1 = \sum modul\ i\ (FanIni * FanOuti)$ 2.

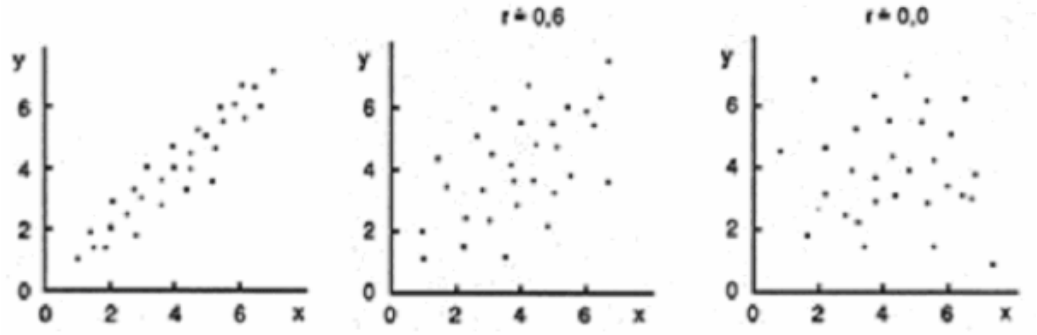
6 Návrh a popis srovnávací metodiky, metriky, testů

Pro hodnocení si vybereme tento hotový softwarový produkt : internetový prohlížeč Mozilla Firefox a jeho dvě verze Mozilla Firefox 0.8 a Mozilla Firefox 2.0.0.6. Program je k dispozici na adrese [11], dostupný přes SourceForge.net, kde jsou dostupné všechny open-source programy. Využijeme bodový diagram k určení vzájemné závislosti mezi verzemi, k čemuž využijeme některé z výše uvedených externích metrik. Pro určení vzájemné závislosti využijeme Pearsonův výsledný momentový korelační koeficient, vyjádřený vztahem, uvedený v [12]:

$$r_{xy} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\left[\sum_{i=1}^{n}(x_i - \bar{x})^2\right]}\sqrt{\left[\sum_{i=1}^{n}(y_i - \bar{y})^2\right]}} = \frac{n\sum_{i=1}^{n} x_i y_i - \left(\sum_{i=1}^{n} x_i\right)\left(\sum_{i=1}^{n} y_i\right)}{\sqrt{\left[n\sum_{i=1}^{n} x_i^{\,2} - \left(\sum_{i=1}^{n} x_i\right)^2\right]}\sqrt{\left[n\sum_{i=1}^{n} y_i^{\,2} - \left(\sum_{i=1}^{n} y_i\right)^2\right]}}$$

kde x_i jsou hodnoty pro verzi Mozilla Firefix 0.8 a y_i jsou hodnoty pro verzi 2.0.0.6 pro i = 1,2,....n, kde n je celkový počet provedených měření.

Podle hodnoty tohoto Pearsonova koeficientu rozlišujeme silnou kladnou závislost (pro r = *0.90*), slabou kladnou závislost (pro r = *0.2*), žádnou závislost (pro r = *0*), silnou zápornou závislost (pro r = - *0.9*) a slabou zápornou závislost (pro r = - *0.2*). Jednotlivým hodnotám Pearsonova koeficientu odpovídá charakteristické rozdělení jednotlivých bodů v bodovém diagramu, obrázky uvedené podle [13]:



Obr. 6.1: Bodové diagramy pro charakteristické hodnoty Pearsonova koeficientu

Hodnoty koeficientů odpovídají v prvním případě r = 0.9 tedy silné kladné závislosti, v druhém případě se jedná o slabou kladnou závislost a ve třetím případě se nejedná o žádnou závislost. V případě záporných závislostí jsou jednotlivé body v grafech inverzní.

Na osu X bodového diagramu budeme vynášet parametry pro verzi Mozilla Firefox 0.8 a na osu Y parametry pro verzi Mozilla Firefox 2.0.0.6. Vstupní data pro diagram získáme pomocí externích metrik:
- LOC (lines of code, kapitola 3.2) ve variantě, kdy se počítají fyzické řádky kódu, které dostaneme na výstupu obrazovky,
- Halsteadovy metriky velikosti programu (kapitola 3.6).

Tyto metriky použijeme pro jednotlivé komponenty se stejnými názvy v obou verzích. Pro bodový diagram je důležité provést alespoň 30 výběrů, spočítáme tedy řádky kódu u 30-ti komponent v obou verzích Mozilla Firefox. V případě použití Halsteadovy metriky velikosti programu spočítáme u jednotlivých komponent tyto čtyři veličiny:

- n_1 = počet unikátních nebo rozdílných operátorů v implementaci
- n_2 = počet unikátních nebo rozdílných operandů v implementaci
- N_1 = celkový počet ze všech unikátních nebo rozdílných operátorů použitých v implementaci (tedy i těch, které se opakují)
- N_2 = celkový počet ze všech unikátních nebo rozdílných operandů použitých v implementaci (tedy i těch, které se opakují)

Na základě těchto veličin vypočítáme pro každou komponentu její objem, tedy počet bitů potřebných pro rozhodnutí při volbě každé z ***n*** položek programového slovníku

Objem: $V = N \log_2 n$, kde $n = n_1 + n_2$ a $N = N_1 + N_2$

Z výsledných bodových grafů a koeficientů zjistíme vzájemnou závislost, popř. potvrdíme nezávislost, což ale nepředpokládáme. Zároveň porovnáním počtů řádek kódu a porovnáním počtu operandů a operátorů jednotlivých komponent u obou verzí můžeme určit, která z verzí je složitější.

K výpočtu Pearsonova koeficientu využijeme program Microsoft Excel, který zahrnuje funkci RKQ, která vrací druhou mocninu Pearsonova výsledného momentového korelačního koeficientu pomocí zadaných datových bodů. Ke znázornění bodového diagramu použijeme grafický program Statgraphics Plus 3.1, který po zadání dvojic x_i, y_i vytvoří pro zadané hodnoty bodový diagram. Program Statgraphics je klasický statistický systém, jehož výhodou je zejména statistická grafika. Je možno různými metodami grafické výstupy upravovat. Výhodou jsou také textové části, které lze různě kopírovat a tisknout. Statgraphics je rozdělen do šesti základních modulů, přičemž bodový diagram nalezneme v kategorii Quality Control. V této verzi programu Statgraphics se nachází také další ze sedmi nástrojů: histogram, Paretova analýza a regulační diagram.

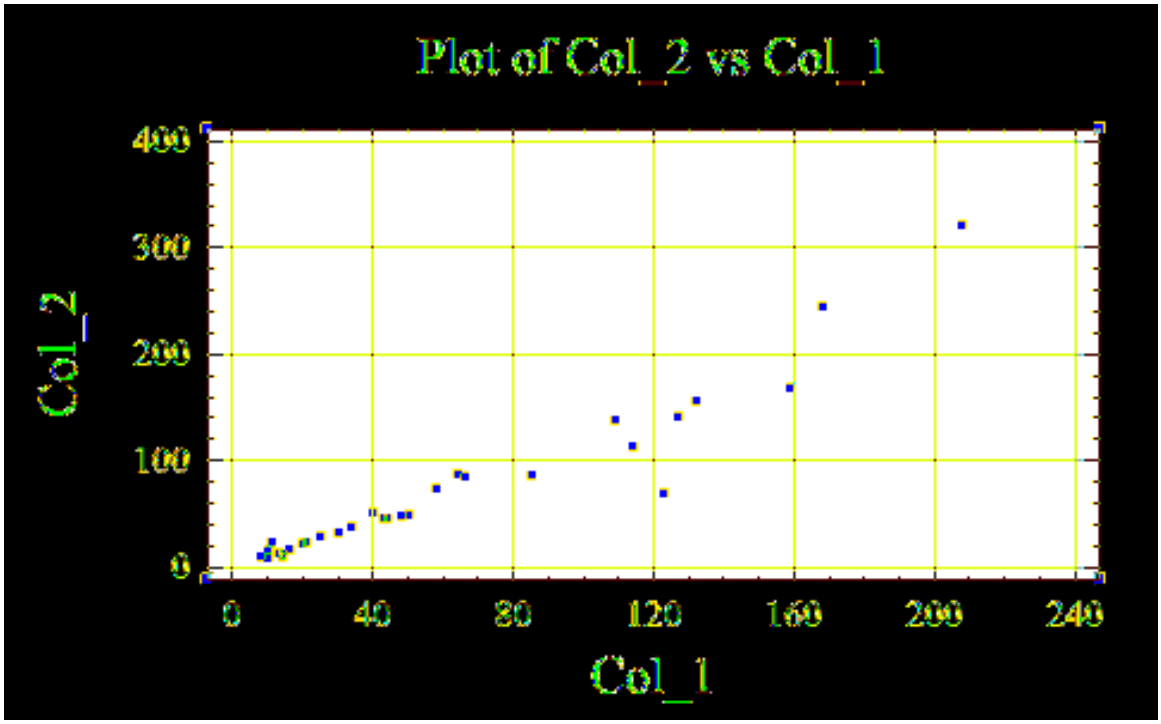
7 Podrobný popis výsledků srovnání, testů, měření

Z naměřených hodnot jsme získali následující tabulku:

č. měření	Komponenty	Mozilla Firefox 0.8	Moz. Fire. 2.0.0.6
1	nsBarProps	10	9
2	nsDomClassInfo	208	322
3	nsGlobalWindow	168	245
4	nsJSEnvironment	58	74
5	leaky	16	17
6	nsScrollPortView	21	23
7	nsViewManager	132	156
8	nsView	25	29
9	nsProfile	85	86
10	nsProfileAccess	50	49
11	js	64	87
12	jsapi	109	138
13	jsarray	40	52
14	jsstr	159	168
15	jsregeep	127	142
16	jquant2	48	48
17	nsChromeRegistry	123	69
18	nsPrintOptionsImpl	44	46
19	nsRegion	34	38
20	nsStackFrameWin	11	23
21	nsMemoryImpl	14	12
22	nsDebugImpl	13	13
23	nsCore	10	15
24	nsXPComInit	30	32
25	nsCategoryManager	20	22
26	nsComponentManager	114	113
27	nsStaticComponentLeader	8	10
28	xcDll	14	10
29	nsDirectoryService	43	46
30	nsLocalFileWin	66	85
Celkem LOC		1864	2179

Tabulka 7.1:Naměřené hodnoty LOC pro Mozilla Firefox 0.8 a 2.0.0.6

Bodový diagram v programu Statgraphics :



Obr. 7.1: Bodový diagram pro verze programu Mozilla Firefox

Vypočtený Pearsonův koeficient :

$r_{xy} = 0{,}952595$, který jsme vypočítali pomocí funkce RKQ a výsledek jsme ověřili dosazením naměřených veličin do vzorce uvedeného v kapitole 6. V obou případech jsme dostali stejný výsledek. Z vypočteného Pearsonova koeficientu vyplývá, že vzájemný vztah je velmi silný a jedná se o silnou kladnou závislost, čemuž odpovídá i tvar bodového diagramu. Můžeme tedy říct, že vyšší verze Mozilly Firefox 2.0.0.6 se vyvinula z nižší verze 0.8. Pokud srovnáme celkový počet řádků kódu u jednotlivých verzí, vyšší verze má více řádků, je tedy složitější. Za předpokladu, že vývojový tým si ze starší verze vzal to nejlepší pro vývoj verze nové, můžeme říct, že nová verze je také kvalitnější a neplatí zde obecné pravidlo, že čím více řádků kódu, tím je větší pravděpodobnost vzniku chyb. V tomto případě nám jinak ne příliš spolehlivá metoda LOC poskytla poměrně přesné výsledky a to bylo způsobeno především tím, že u obou verzí byl použit stejný programovací jazyk (C++) a tím, že struktura programů byla podobná.

V případě použití Halsteadovy metriky pro získání vstupních dat jsou výsledky analogické.

Verze Mozilla Firefox 0.8 je verze z roku 2004 a je to vůbec první verze, u které se začal používat název Mozilla Firefox, jenž je užíván dodnes. U verze 2.0.0.6 se objevily zprávy, že tato nová verze opravuje závažné bezpečností chyby předchozích verzí a zde je hodnocení nové verze Mozilla Firefox 2.0.0.6 z portálu www.chip.cz:

„Nová verze přináší proti předchozí řadu vylepšení. Zejména se jedná o rychlejší start a lepší práci s operační pamětí, vylepšení práce s panely, integrovaný nástroj pro opravu překlepů, velmi šikovné automatické doplňování vyhledávaných výrazů, znovuotevření uzavřeného webu, anti-phishingový filtr, správu vyhledávačů a přepracovanou funkci automatické aktualizace.“

8 Zobecnění výsledků rešerše, výsledků testů, závěr

Největší přínos sedmi základních nástrojů nastává, pokud jsou zabudovány a pevně svázány s vývojovým procesem, který se skládá z návrhu, realizace a testování softwarových produktů. V případě zautomatizovaného používání lze dosáhnout značných zlepšení v oblasti kvality a vývoje softwarových produktů.

Checklist může být použit ve všech fázích vývoje jako nástroj pro kontrolu plnění jednotlivých částí a zohlednění všech faktorů. Paterův diagram, diagram příčin a následků pomáhají nalézt nejdůležitější příčiny nějakého problému, kterým mohou být např. vysoké náklady nebo vznik chyby. Histogram patří mezi nástroje pro rychlé a jednoduché srovnání určitých parametrů podle četnosti např. porovnání produktů podle počtu nalezených chyb. Bodový diagram potvrdí závislost popř. nezávislost dvou programů nebo parametrů, např. závislosti počtu chyb na složitosti programu. Regulační a průběhový diagram sledují buď celý proces nebo jeho části po určitou dobu a slouží ke kontrole, jestli nedošlo k neočekávaným jevům. Tento popsaný postup je nejlepší způsob, jak využít tyto nástroje, zvyšování kvality je jejich hlavní účel.

Pokud chceme metodiku pro hodnocení hotových SW produktů, je vhodné použít bodový diagram, pomocí něhož můžeme hodnotit dvě verze jednoho programu, nebo dva různé programy, které mají mezi sebou vzájemný vztah, popř. dva parametry jednoho programu ve vzájemném vztahu. Jako hodnotící metriku můžeme zvolit LOC (při srovnávání dvou programů pouze v případě, že jsou oba programy napsány ve stejném programovacím jazyce, aby byly výsledky objektivní) nebo Halsteadovu metriku, která je vhodná pro všechny programovací jazyky, popř. McCabeovu metriku, která je jednoduchá k použití.

V této práci se s úspěchem podařilo navrhnout a aplikovat metodiku, kterou je možno hodnotit SW produkty, pro autorku bylo velkým přínosem zejména poznání jednotlivých metrik, možností jejich použití a především výhod a výsledků, které plynou v případě jejich vhodného použití. Nyní pokud by nastala situace, že by autorka byla součástí vývojového týmu a aktivně se podílela na vývojovém procesu, nebyl by pravděpodobně problém navrhnout a aplikovat některé ze sedmi nástrojů pro zlepšení kvality produktů v daném podniku. Zároveň jsme si ověřili, že použití takovýchto nástrojů není vůbec komplikované. Pro aplikaci nástrojů pro řízení kvality jsme nepotřebovali ani složité vzorce ani kalkulačku, práci udělaly za nás programy jakými jsou např. Microsoft Excel a Statgraphics.

9 Seznam použité literatury a internetových zdrojů

[1] Praktické využití statistických a grafických metod ve firmě a jejich aplikace za využití statistického software : http://www.fs.vsb.cz/akce/2004/asr2004/Proceedings/Papers/031.pdf

[2] Úvod do plánování projektu: http://www.fit.vutbr.cz/study/courses/PPS/public/pdf/14_2.pdf

[3] Rychlost programování: http://www.neo.cz/~tomas/java.net/2004/06/rychlost-programovn.html

[4] Odhadování pracnosti: http://cevis.datis.cdrail.cz/MRKP/5/5416OdhadPracnosti.html

[5] Jak dlouho trvá vývoj aplikace: http://www.zive.cz/h/Programovani/AR.asp?ARI=7415

[6] Software Measurement: http://yunus.hacettepe.edu.tr/~sencer/research.html

[7] Software Measurement: http://www.fi.muni.cz/~sochor/PA104/Slajdy/Mereni.pdf

[8] Stephen H.Kan, „Metrics and Models in Software Quality Engineering“, Addison-Wesley Publishing Company, 1995, str. 86, 127-150.

[9] Nástroje jakosti: www.ft.utb.cz/czech/uvi/czech/staff/shejbalova/JM/4.pdf

[10] Od specifikaci k předání, metriky: http://kocour.ms.mff.cuni.cz/~kral/Prednpodzim05/

[11] Index pro ftp://ftp.mozilla.org/pub/mozilla.org/firefox/releases/ : ftp://ftp.mozilla.org/pub/mozilla.org/firefox/releases/

[12] Jednoduché nástroje řízení jakosti: http://www.businessinfo.cz/files/2005/061019_nastroje-rizeni-jakosti-1.pdf

[13] Technická univerzita v Liberci – Řízení jakosti a spolehlivosti, Eva Pelantová a Pavel Fuchs, prezentace, slide 20

A Seznam použitých zkratek

ADDR	Problems Related to Addresses
APAR	Authorized Programming Analysis Report
CC	Cyclomatic Complexity
CCI	Component Commonality Index
COCOMO	Constructive Cost Model
CPLX	Complex Logical Problems
CUE	Common Use Element
DEFN	Data Definition Problems
DPP	Defect Prevention Process
FP	Function Points
HP	Hewlett Packard
IBM	International Business Machines Corporation
INIT	Initialization Problems
INTF	Interface Problems
IT	Informační technologie
KLOC	KLines of Code
LOC	Lines of Code
LSL	Lower Specification Limit
NASA	National Aeronautics and Space Administration
NLS	National Languages Problems
NOP	New Object Point
MS	Microsoft
OP	Object Point
PDPC	Process Decision Program Char

PROD	Produktivity Rate
PTF	Program Temporary Fix
RKQ	Funkce pro Pearsonův výsledný momentový korelační koeficient
SL	Specification Limit
SPC	Statistical Process Control
SW	Software
TCF	Technical Complexity Factor
UFP	Unadjusted Function Points
USL	Upper Specification Limit
VT	Výpočetní technologie

B Obsah přiloženého CD

bak_prace_Klimesova_5.pdf - bakalářská práce ve formátu pdf

bak_prace_Klimesova_5.doc - bakalářská práce ve formátu doc

tabulka.xls – tabulka v Microsoft Excel pro výpočet Pearsonova korelačního koeficientu

adresář materialy - materiály, ze kterých jsem čerpala, číslo v hranatých závorkách v názvu souboru značí referenci na položku v seznamu použité literatury