Česká zemědělská univerzita v Praze
Provozně ekonomická fakulta
Katedra informačího inženýrství

Odhad manažerských charakteristik vývoje IS v etapě specifikace požadavků

Doktorská disertační práce

Doktorand: Ing. Josef Pavlíček
Školitel: Prof. RNDr. Jiří Vaníček, CSc.
Obor: Informační mamagement

Praha, srpen 2006

Děkuji svému školiteli Prof. RNDr. Jiřímu Vaníčkovi, CSc. za vedení této práce.

Abstrakt:

Práce se zabývá navržením manažerských charakteristik vývoje informačních systémů v etapě specifikace požadavků. Jde především o vymezení pojmu Celková Složitost softwarového Produktu (*CSP)*. Ta je odhadována na základě funkční specifikace na software, případů užití, scénářů a faktorů ovlivňující vznik software. Odhad Celkové Složitosti softwarového Produktu se stává klíčový pro manažerská rozhodnutí typu zda realizovat softwarový projekt, jehož výsledkem je hotový softwarový produkt, či jaké zdroje pro vývoj alokovat, jak vytvořit časový plán pro jeho vývoj a pro úvahy o ceně výsledného produktu.

V práci jsou analyzovány známé přístupy k odhadu složitosti softwaru. Autor se zabývá analýzou metod Funkčních jednic - Function point, metodou odhadu nákladů COCOMO, ale i relativně moderní metodou odhadu pracnosti software na základě bodů případů užití - UseCase points a řadou dalších. Na základě poznatků získaných studiem a analýzou těchto metod autor navrhuje postup pro odhad složitosti softwaru tak, aby odhad byl co nejpřesnější. V práci je vymezen pojem složitost software, faktory složitosti ovlivňující vznik software, základní složitost softwarového produktu, a celková složitost softwarového produktu. Jednotlivé dílčí složitosti se stávají základními mamažerskými charakteristikami vývoje informačních systémů.

Existující metody odhadu složitosti softwaru jsou podrobeny kritické analýze. Je ukázáno, že z pohledu teorie měření je nelze teoreticky zdůvodnit. Proto autor navrhuje alternativní postup odhadu složitosti, při kterém jsou ty kroky odhadu, které jsou založeny jen na subjektivním hodnocení svěřeny neuronovým sítím.

Pro účel odhadu celkové složitosti softwarového produktu byla autorem navržena neuronová síť PETRA. V textu je uveden popis příslušné neuronové sítě a je zde zdůvodněno, jakým způsobem je možné ji pro účel odhadu využít. Tato část práce je původním přístupem autora k problému.

Práce sítě PETRA je ověřena na 21 úlohách se známými výsledky pro adaptaci sítě a je predikována složitost čtyřech projektů, které dosud nebyly dokončeny. Je ukázáno, že získané odhady jsou nejméně tak přesné, spíše přesnější, než jsou údaje o přesnosti odhadu známých postupů, uvedené v literatuře.

Klíčová slova:

Softwarové inženýrství, Metoda funkčních jednic, Metoda bodů Případu užití, Složitost, Projekt, Případ užití, Diagram užití, Unified Process, Rational Unified Process, Unified Model Language, Perceptron, Neuron, Neuronová síť, Zpětné šíření chyby, Jakost, Atribut,PETRA.

Abstract:

The work deals with the system complexity prediction. For mangers is very useful to be able to predict the system complexity on the beginning of the software project. To know the costs necessary for realizing of a new software, to know number of programmers, analytics, testers etc. is useful before we start programming. It is very early. It usually is in the first phase of the software project live cycle. To have some method, which is able to predict the system complexity from the function requirements on the new software in the first phase of the software live cycle is useful. The author analyzes the familiar methods like COCOMO, Function Point or UseCase Point. On the results and knowledge from the analysis of them the author designs the new method of software complexity prediction. There are determinate the most important terms like as software complexity, complexity factors influencing the software originating, basic complexity of the software product and the whole complexity of the software product. The particular complexities are the base for the managers characteristic for the development process of information systems.

The obtaining methods for software complexity prediction are analyzed and criticized. The author shows, that from the point of view of the theory of measurement the obtaining methods for software prediction don't give the reasons. On the base of this fact the author designs the alternative method, where are the steps of the prediction method, which are based only on the subjective classification, given to keep to the artificial intelligence - neural networks.

For the purpose of the prediction method the author designs the neural network PETRA. In the paper is a description of PETRA and there is the reasoning, how is possible to use this neural network for the method purpose. This part of the dissertation work is an original author approach for this problem.

The functionality of PETRA is tested on 21 tasks. For the each task is known the right result. The network is adapted on the specific company environment and is tested how is the network able to adapt on it. Instead of that the network is tested to predict the software complexity for not finished projects. The tests show that the predictions of

the network are very accurate and the results are as good as other methods, rather better than the results of the known approaches mentioned in literature.

Keywords:

Software engineering, Function Point method, UseCase Point method, Use Case, Software complexity, UseCase diagram, Unified Process, Rational Unified Process, Unified Model Languare, Perceptron,Neuron, Neural network, Error backpropagation, Quality, Attribute, PETRA.

Seznam zkratek:

- UP – Unifiled Process – metoda složící pro řízení softwarových projetků viz [7].
- RUP – Rational Unified Process – komerčně (firmou IBM) vyvíjený klon metody UP viz [16].
- UML – Unified Model Language viz [4,7] – modelovací jazyk určený pro modelování softwaru.
- Use Case – Případ užití.
- UCP – Use Case Point – metoda odhadu složitosti navržená G. Karnerem viz [24].
- FP – Function point – metoda funkčích jednic navržená A. Albrechtem viz [11,35].
- COCOMO – COnstructive COsts MOdel, metoda odhadu nákladů potřebných na tvorbu software navržená panem Boehmem viz [34].
- Složitost - obecně je viz [3] „*chápána jako míra úsilí, které potřebuje člověk vynaložit k tomu, aby produkt navrhl, vyvinul, implementoval, provozoval a udržoval v průběhu jeho životního cyklu“*.
- ZS – Základní složitost diagramu užití.
- ZSP – Základní složitost softwarového produktu.
- FS – Faktory složitosti ovlivňující složitost softwarového produktu.
- FSk – Faktor složitosti klasifikovatelný.
- Fso – Faktor složitosti obtížně klasifikovatelný.

Obsah

1 Úvod

Cílem mojí disertační práce je vypracovat metodu odhadu složitosti softwaru, která by byla použitelná již v etapě specifikace požadavků na software – tedy v okamžiku, kdy už jsou známy požadavky na funkčnost požadovaného softwaru, ale ještě před tím, než začneme softwarový produkt programovat. Navržené odhady jsou získávány postupy, které byly inspirovány výsledky mezinárodního vědeckého týmu SQuaRE[1], který navrhl normy pro hodnocení jakosti softwaru. Kromě nepochybné souvislosti mezi jakostí a složitostí, tedy i pracností software je zde také důvod, proč se jakostí zabývat a modelem hodnocení jakosti se inspirovat.

Oba tyto důležité atributy software a systémů obsahujících software je totiž velmi důležité odhadovat co nejdříve, jakmile je to jen možné. Nejlépe již v etapě specifikace. Pro odhady (tak zvané prediktory) potřebujeme z podkladů, které jsou v dané etapě k dispozici, určitou množinu vstupních dat, tak zvaných primitiv pro měření, neboli základních měr. Tyto množiny sice nemusí být shodné, do značné míry se však překrývají.

Pro měření a hodnocení jakosti je k dispozici model podpořený mezinárodními normami a množstvím postupů jak jakost hodnotit a měřit viz [12]. Pro odhady jsou navrženy také vnitřní míry. Pro složitost takový model dosud není a ani postupy pro odhad nejsou popsány. Nabízí se tedy cesta pokusit se to, co je vytvořeno pro odhad jakosti modifikovat tak, aby to bylo použitelné i pro odhad složitosti.

1.2 Cíle disertační práce

Softwarové inženýrství je disciplínou, která se zabývá plánováním a řízením vývoje softwaru. Pro plánování jakéhokoliv procesu[2] je třeba znát odhad nákladů na jeho provedení a minimální čas, potřebný pro tento proces. Znát náklady a minimální čas umožní rozhodnout o budoucnosti softwarového projektu[3].

1 Software QUality Requirements and Evaluation

2 Dle [2] „Proces je soubor vzájemně souvisejících nebo vzájemně působících činností, které přeměňují vstupy na výstupy“.

3 Dle [2] “Projekt je jedinečný **proces** sestávající z řady koordinovaných a řízených činností s daty zahájení a ukončení, prováděný k dosažení cíle, který vyhovuje specifickým **požadavkům**, včetně omezení daných časem, náklady a zdroji“.

Pro odhad nákladů a minimální potřeby času navrhuji zavést „odhad složitosti“. Ten provádím za pomoci vstupních dat získaných z diagramů užití vytvořených dle postupů popsaných v metodě Unified Process viz [7]. Diagramy užití jsou součástí průmyslového standardu UML viz [18] (Unified Model Language).

O standardu UML viz [4,5] a metodě Unified Process viz [1,7,17] hovořím v další kapitole disertační práce.

1.2.1 Hlavní cíl disertační práce:

Hlavním cílem disertační práce je vytvořit metodu odhadu složitosti softwaru a navrhnout manažerské charatkeristiky vývoje informačních systémů, které by byly použitelné v etapě specifikace požadavků na software. Tato metoda by měla být obecně použitelná pro libovolné softwarové produky vytvořené dle postupů popsaných v metodě Unified Process viz [7] tak, aby bylo by možné odhad složitosti provádět již v okamžiku, kdy je hotová specifikace požadavků na software.

1.2.2 Dílčí cíle disertační práce:

- Vypracovat kromě odhadu složitosti i metodu pro rozpoznání klíčových faktorů, které se podílejí zásadním způsobem na složitosti celého řešení.
- Navrhnout využití inovativních prostředků pro odhad složitosti. Především pak využití neuronové sítě typu Perceptron s Error backprogation učebním algoritmem.
- Navrhnout vhodnou architekturu neuronové sítě pro daný účel a dát jí k dispozici široké veřejnosti.

1.3 Východiska řešení

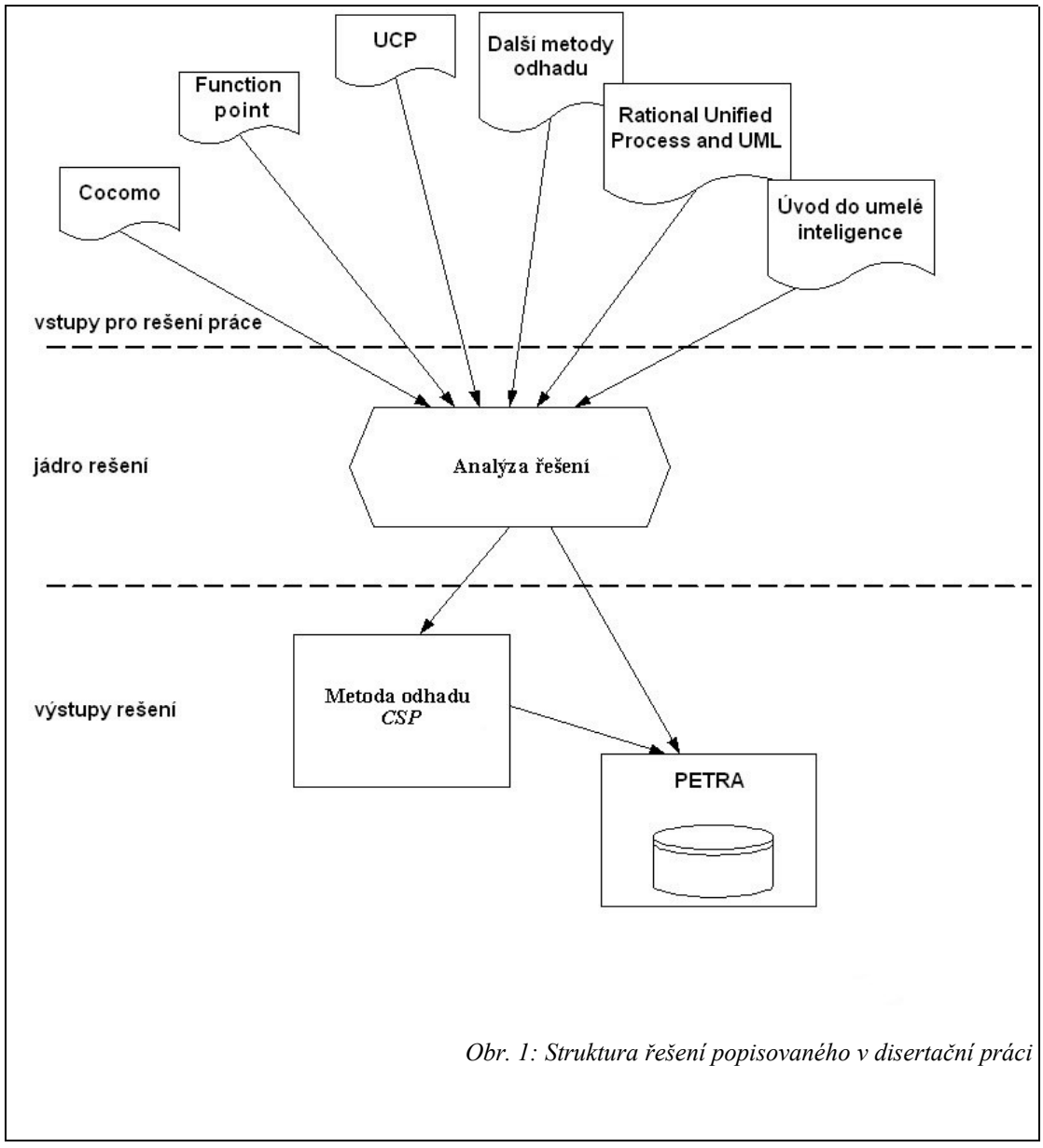
V disertační práci vycházím ze svých dřívějších zkušeností publikovaných jak v České republice, tak v zahraničí a dále ze:

- Zkušeností získaných z mezinárodních vědeckých konferecní (Kypr 2003, USA - Orlando 2003, USA - San Francisco 2005).
- Zkušeností získaných během pětiletého studia na České zemědělské univerzitě, Katedře informačního inženýrsví viz seznam publikovaných prací.
- Zkušeností získaných ze své praxe:
    - programátor v jazyce Java ve firmě AIP Safe, s.r.o.,
    - vedoucí týmu vývojářů ve firmě AIP Safe, s.r.o..

- Z analytické praxe získané v Komerční bance a.s.:
    - analýza a nasazení Intranetového portálu,
    - analýza softwarového řešení pro správu interních předpisů.
- Ze zkušeností získaných ve firmě Sun Microsystems:
    - sběr uživatelských požadavků na softwarovou komponentu Profiler viz [A.18].
    - práce na softwarovém designu pro nástroj NetBeans.
- Ze zkušeností s vedením projektů a z certifikací, které jsem získal během své praxe ve firmách:
    - AIP Safe s.r.o.,
    - Komerční banka a.s.,
    - Sun Microsystems.

1.4 Struktura řešení a návaznost jednotlivých kapitol práce

Struktura řešení, kterou jsem použil k dosažení cílů práce, je uvedena na Obr 1.



Obr. 1: Struktura řešení popisovaného v disertační práci

Struktura řešení vychází z obecných principů a směřuje k návrhu a praktické aplikaci výsledků mé práce. Jednotlivé kapitoly je možné rozdělit do tří skupin:

1. analýza dostupných zdrojů řešení (vstupy do řešení práce),
2. analýza navrhovaného řešení (jádro řešení),
3. návrh a implementace metody odhadu složitosti (výstupy řešení).

1.4.1 Analýza zdrojů – vstupy pro řešení

Tato část práce obsahuje kapitoly, v nichž analyzuji současný stav jednotlivých technik odhadu složitosti. Obsahuje také kapitolu, ve které analyzuji možnost vyžití umělé inteligence při odhadu složitosti softwaru. Na základě závěrů z těchto kapitol pak provádím analýzu a návrh vhodné metody odhadu složitosti.

Kapitola "COCOMO" *(metoda COnstructive COsts MOdel)* - obsahuje analýzu metod[4] COCOMO. Kriticky se zamýšlím nad použitelností metod v praxi. Metody COCOMO jsou vyvíjeny a rozšiřovány na podnikové standardy. Ač není jejich nasazení masové, je dobré znát jejich přednosti. Proto jsem studoval a analyzoval kladné stránky metod. Cílem této kapitoly je nalézt základní třídu atributů měření.

Kapitola "Metoda Funkčních jednic" *(Function point)* - obsahuje anlýzu této metody odhadu složitosti softwaru. Stejně jako metody z rodiny COCOMO, je metoda Funkčních jednic metodou nepřehlédnutelnou. Proto se jí podrobně věnuji. Cílem této kapitoly je analyzovat možnost použití funkčních bodů ve prospěch odhadu složitosti.

Kapitola "UCP" *(metoda UseCase Points)* - obsahuje anlýzu této metody odhadu složitosti softwaru. Stejně jako metoda Funkčních jednic, je UCP metodou významnou. Proto se jí věnuji. Cílem této kapitoly je analyzovat možnost jejího použití ve prospěch odhadu složitosti.

Kapitola "Další metody odhadu složitosti, pracnost, dělba práce" obsahuje výčet a anlýzu dalších metod odhadu složitosti. Teoreticky se věnuji problematice pracnosti, dělby práce, vlivu napjatých termínů atd. Cílem této kapitoly je analyzovat a rozhodnout o případném použití některých technik ve prospěch odhadu složitosti.

4 Metoda COCOMO se stala celosvětově uznávanou metodou pro odhad nákladů potřebných na realizaci softwarového produktu. Od základní COCOMO někdy označované COCOMO 81 navržené Dr. Barrym Boehmem vznikla celá řada variant COCOMO. Proto v disertaří práci hovořím o COCOMO v množném čísle. Mám tím na mysli metody navržené na základech COCOMO.

Kapitola "Rational Unified Process and UML" obsahuje zhodnocení a analýzu objektově orientovaného návrhu softwaru popsaného v Rational Unified Process viz [1,7,17]. Tato metoda využívá k návrhu software modelovací jazyk UML viz[4,5]. Rational Unified Process (zkracujeme RUP) se nezabývá odhadem složitosti software. Jeho cílem je umožnit řízený vývoj software. Cílem této kapitoly je zhodnocení RUPu a využití úspěšných technik RUPu pro účel odhadu složitosti software.

Kapitola "Úvod do umělé inteligence a její praktické využití" obsahuje popis a zhodnocení několik typů neuronových sítí. V této kapitole se zabývám především perceptronovou neuronovou sítí. Tato síť je složena z neuronů jejichž přenosovou funkcí je nejčastěji sigmoida. Systém učení sítě je řešen za pomoci zpětného šíření chyby. Cílem této kapitoly je odhalit možnosti využití umělé inteligence pro účely odhadu složitosti software.

1.4.2 Analýza řešení

V této části práce, na základě získaných poznaktů z předchozích kapitol analyzuji zvolený cíl – navrhnout metodu odhadu složitosti software ve fázi jeho specifikace.

Kapitola "Analýza řešení" obsahuje analýzu řešení odhadu složitosti software. V této kapitole uvádím návrh řešení odhadu složitosti. Upozorňuji na úskalí návrhu. Na základě získaných poznatků uvedených v předchozích kapitolách navrhuji nejvhodnější řešení.

1.4.3 Návrh a implementace

V této části práce navrhuji a implementuji novou metodu odhadu složitosti, kterou jsem na základě znalostí získaných studiem pramenů předložených a analyzovaných v předchozích kapitolách navrhl. Tato metoda je podpořena výsledky, které jsem ověřil v existujícím prostředí reálné firmy Sun Microsystems.

Kapitola " Metoda odhadu *CPS*" obsahuje návrh a ukázku jak postupovat při užití metody odhadu složitosti softwarového produktu, kterou jsem pro tento účel navrhl.

Kapitola "PETRA" obsahuje detailní popis neuronové sítě PETRA. Obsahuje popis její architektury i její topologie. V této kapitole se zabývám problematikou učení neuronové sítě. Ukazuji, jak je možné neuronovou síť adaptovat na prostředí libovolné firmy a jak využít její schopnosti predikce k odhadu složitosti softwarového produktu.

Kapitola "Praktické ověření odhadu celkové složitosti softwarového produktu s využitím neuronové sítě PETRA" obsahuje návrh a ukázku implementace navržené metody. Metoda je kriticky zhodnocena. Upozorňuji na její silné i slabé stránky. Byť se jedná o metodu obecnou, je vždy nutné ji adaptovat na prostředí, ve kterém bude použita. V této kapitole presentuji naměřené výsledky metody odhadu složitosti, které jsem získal z uvedených experimentů.

1.5 Metody dosažení cílů práce

Postup dosažení zvoleného cíle jsem rozdělil do tří uvedených dílčích cílů. Ty určují základní strukturu disertační práce.

První, nejobsáhlejší část práce, je výsledkem mého téměř pětiletého studia. Problematikou odhadu složitosti jsem se aktivně zabýval (a zabývám se stále). Teze uváděné v disertační práci a výsledky z nich odvozené jsou získány studiem literatury

jak tuzemské tak i zahraniční. Řadu poznatků jsem získal diskusí na vědeckých konferencích jak vnitrostátních tak mezinárodních viz. Přehled publikovaných prací, např. [A.1,A.2,A.3,A.7,A.9].

Ve své práci se snažím respektovat moderní trendy řízení tvorby software, které jsou popsány v metodě UP viz [7] a jejím klonu RUP viz [16]. Z tohoto důvodu také využívám terminologii definovanou v Unified Process. Unifikovaný modelovací jazyk UML viz.[18] z části využívám pro účely odhadu složitosti.

V práci jsem využil své praktické zkušenosti s řešením odhadu složitosti software v Software eXperience Design Teamu[5] firmy Sun Microsystems a své účasti v sdružení OMG *(Object Management Group)*[6].

5 Software eXperience Design Team – tým,ve kterém pracuji jako zaměstnanec firmy Sun Microsystems. Podílím se na vývoji softwarových komponent pro programovací nástroj NetBeans. V seznamu publikovaných prací viz [A.18].

6 Jsem aktivním členem tohoto sdružení a zastupuji zde firmu Sun Microsystems. V OMG sdružení se jedná napaříklad o uzavření strandardu pro jazyk UML a další viz [17].

Během práce na jednotlivých kapitolách disertační práce jsem řešil řadu dílčích problémů vzhledem k stanoveným cílům. Jednalo se především o problém ordinárních hodnot, při volení vhodných atributů měření. Jejich řešení na základě dostupných pramenů (literatura, konference, konzultace, Internet), zkušeností nebo rozhodnutím mě vedlo k navržení metody odhadu složitosti softwarového produtku.

Ta je založena na výpočtu tak zvané základní složitosti softwarového produtku. Výpočet je prováděn na základě funkční specifikace, která popisuje chování budoucího softwarového produktu. Protože vznik software je ještě ovlivněn řadou faktorů, které jeho vznik ovliňují nepřímo, ale mohou mít na jeho vznik zásadní vliv, není možné hovořit pouze o složitosti software. Proto jsem se rozhodl odhad nazývat Celkovou Složitostí softwarového Produktu *(CSP)*. Číselnou hodnotu získanou tímto odhadem navrhuji považovat za **základní manažerskou charakteristiku vývoje IS**. Odhad *CSP* pak v sobě zahrnuje jak odhad složitosti vznikajícího software, tak i klasifikaci jednotlivých faktorů *FS*, které vznik software ovlivňují. Ty navrhuji považovat za **manažerské charakteristiky složitosti vývoje IS.**

Obdobný odhad složitosti se mi nepodařilo v světové literatuře, ani při praxi v podnicích, nalézt. Z návrhů jak složitost softwaru odhadnout je zjevné, že vhodná metoda pro objektově orientované programování chybí. Byť ve světě existuje řada návrhů jak složitost odhadovat viz. COCOMO, tak se zdá jejich použití velmi svízelné. Z mých zkušeností, praxe i diskusí to jednoznačně vyplývá. Na základě této zkušenosti navrhuji novou metodu.

Hlavním problémem práce bylo popsat podmínky aplikace metody, aby ji bylo možné přijmout, jako možný přístup k odhadu složitosti softwaru. Dalším problémem bylo metodu prakticky implementovat a následně ověřit její použitelnost v praxi.

Mojí snahou bylo zformovat problematiku odhadu složitosti softwarového produtku co nejsrozumitelněji. V případě, že v práci uvádím nějaké zobecnění, vždy je uveden odkaz na podkladové prameny. Problematika odhadu složitosti už vzhledem k její podstatě, nemohla být pojmuta komplexně. Proto jsem soustředil na principy

vedoucí k dosažení cíle disertační práce.

1.6 Stav řešení zvolené problematiky v České republice a ve světě

Problematika odhadu složitosti softwaru je souhrnně popsána v definici metod COCOMO viz [34], FP viz [14], UCP viz [24]. Ani zde není problematika odhadu složitosti softwaru popsána absolutně. Velkým problémem těchto metod je možnost interpretace výsledků výpočtů, jejichž vstupem jsou míry získané měřením v měřících stupnicích ordinálního typu. Z teorie měření plyne, že tyto výsledky nejsou obecně invariantní vzhledem k volbě měření. Metody se buď částečně nebo úplně vyhýbají popisu prostředí, které se na tvorbě softwarové komponenty nepřímo podílí. Tyto nedostatky se snažím řešit navržením alternativní metody odhadu složitosti softwarového produktu a navržením dvou základních tříd faktorů ovlivňujících složitost. Jedná se o faktory *FSk (Faktory Složitosti klasifikovatelné)*, které jsou klasifikovány na základě případů užití vytvořených v etapě specifikace požadavků na software podle metody UP a faktory *FSo (Faktory Složitosti klasifikovatelné obtížně)*, které jsou klasifikovány v závislosti na prostředí vzniku software.

Tyto třídy faktorů *FSk* a *FSo* navrhuji považovat za:

- *FSk* - **Manažerské charakteristiky zadání požadavků na IS,**
- *FSo* - **Manažerské charakteristiky prostředí vývoje IS.**

Pokud obě třídy faktorů spojím do jedné třídy (provedu jejich zobecnění), pak hovořím o obecných faktorech složitosti *FS (Faktory Složitosti)*. Ty, jak jsem již uvedl, navrhuji považovat za obecné **Manažerské charakteristiky složitosti vývoje IS.**

Metoda odhadu složitosti softwarového produktu a třídy faktorů jsou uvedeny v kapitolách 3. a 4. v disertační práci.

1.7 Oblasti řešené problematiky, které nejsou v práci obsaženy

V disertační práci neřeším:

- Problematiku operací s naměřenými atributy získanými pomocí stupnic ordinárního typu.
- Problematiku učení neuronové sítě metodou učení zpětným šířením chyby.
- Problematiku více vrstvých neuronových síti (s větším množstvím skrytých vrstev).
- Problematiku neuronových sítí s větším množstvím výstupních neuronů.

1.8 Předpokládané přínosy disertační práce

Přínosy práce pro teorii:

- Práce ukazuje možnost propojení vědeckých disciplín jako je matematika a informatika v prakticky použitelný celek.
- Využití umělé inteligence je stále zastřeno jistou rouškou tajemství. Právě její praktické využití v případě odhadu složitosti tvorby software otevírá nové možnosti jejího praktického využití.
- Metoda, kterou popisuji v disertační práci, je nová, doposud nebyla nikde publikována.

Přínosy pro praxi:

- Veřejnost dostává k dispozici alternativní metodu odhadu celkové složitosti softwarového produktu.
- Komerční i neziskové organizace dostávají k použití způsob, jak složitost softwaru odhadnout. Tento odhad se může stát součástí strategického řízení podniku či vývojářského týmu.
- Díky jednoduchosti metody odhadu složitosti je na snadě její praktické využití. Bez velkého úsilí zadavatele je metoda schopna provádět řadu analýz (odhad složitosti softwarového produktu, rozpoznání klíčových faktorů ovlivňujících vznik softwarového produktu). Ty opět mohou být účelné pro praktické využití v řízení vývoje softwaru.
- Metoda odhadu je k dispozici zdarma včetně neuronové sítě PETRA. Je tudíž dostupná široké veřejnosti. Ta ji může dle potřeby vylepšovat či měnit.

2 Analýza zdrojů – vstupy pro řešení práce

Na začátku této kapitoly vymezím pojem „etapa specifikace požadavků“. Ikdyž se touto etapou zabývám podrobně v kapitole 2.8, pro účel kritického zhodnocení metod COCOMO, Funkčních jednic, UCP a dalších, uvedených v disertační práci, je to nutné udělat dříve.

Pod pojmem „etapa specifikace požadavků“ chápejme časový interval od zadání prvních požadavků na software, až po okamžik, kdy je již funční specifikace na software hotová a schválena zadavatelem. Tudíž známe uživatelské požadavky na chování a funkčnost požadovaného softwarového produktu. Protože však různé metody řízení tvorby softwaru, jako je například metoda Unified Process viz [7] nebo BORM viz [23], mají různý počet fází (UP má čtyři) a v jednotlivých fázích může být různé množství různých etap, na jejichž konci jsou nějakým způsobem shrnuty požadavky na vznikající software, navrhuji tento časový interval obecně nazývat **etapa specifikace požadavků**.

Pro účel metody odhadu složitosti pak navrhuji využít metody řízení tvorby softwaru Unified Process, která má čtyři fáze a etapa specifikace požadavků na software začíná v první fázi Unified Processu – ve fázi **Založení** a končí v druhé fázi nazývané **Rozpracování požadavků na software**. V metodě Unified Process se tato etapa nazývá Requirements, což lze přeložit jako seznam požadavků či analýza požadavků.

2.1 COCOMO

Viz [3],„*COCOMO (COnstructive COsts MOdel) publikoval Boehm na počátku osmdesátých let. Mělo jít o cíl, ke kterému se měl blížit vývoj software při dodržení zásad softwarového inženýrství a využívání moderních metodik tvorby programů v imperativním prostředí.“mam t*

2.1.1 Matematický model odhadu COCOMO

Pro odhad COCOMO zavádí Boehm viz [34] tři základní pojmy viz[3,34]:

- “***E -*** *celková pracnost projektu v člověkoměsících za předpokladu, že se řešitel vlastní práci na projektu věnuje plně asi 150 a více hodin týdně.“*
- ***„T -*** *minimální “rozumná” doba realizace projektu v měsících práce*“
- ***„V*** *- modifikovaný počet tisíců zdrojových řádků kódu tvořícího softwarové řešení*“.

Viz [3]:

Pak platí odhady:

$$E = a \cdot V^b$$

$$T = c \cdot E^d$$

kde a, b, c, d jsou konstanty. Velikost konstant závisí na řadě různých okolností jako je volba zdrojového jazyka a vývojového prostředí.

Viz [3]: „*Základní model COCOMO rozlišuje tři úrovně projektů, zvané módy*“.

1. **Organický mód** – relativně malý softwarový tým pracuje na známé aplikaci. Tým využívá známé algoritmy.

$$a = 3.2, b = 1.05, c = 2.5, d = 0.38$$

2. **Přechodný mód** – je přechodem mezi organickým módem a třetím “vázaným” módem. Týká se projektů, kde je kladen zvýšený nárok na komunikaci.

$$a = 3.0, b = 1.12, c = 2.5, d = 0.35$$

3. **Vázaný mód** – je definován pro velké projekty, jenž jsou řešeny za obtížných podmínek, při častých změnách požadavků v průběhu řešení. Operační systém a hardware je nestabilní. Jsou kladeny vysoké nároky na stabilitu a rychlou časovou odezvu.

$$a = 2.8, b = 1.20, c = 2.5, d = 0.32$$

2.1.2 Analýza odhadu COCOMO

Viz [3] “*Boehmův návrh konstant je na prvý pohled poněkud zvláštní. U krátkých programů (například délky tisíce řádků) dává totiž pro program v obtížném vázaném módu nižší pracnost, než pro program stejného objemu, realizovaný v snažším organickém módu”*. Tento paradox lze odstranit tak, že konstantu *a* budeme volit vždy stejnou. Například rovnou 3 bez rozdílu módu.

Autor skript Měření a hodnocení jakosti informačních systémů, Vaníček viz [3] uvádí “*že uvedenou opravu je třeba provést a rozdělení projektů do módu nelze chápat dogmaticky*”. Vaníček dále uvádí jiný, konkureční návrh, odhadu COCOMO. Volí konstantu *a* rovnu 3. Viz [3] “ *konstanta 3 odpovídá produktivitě práce cca 17*

řádků zdrojového kódu malého programu za den, ovšem včetně návrhnu, dokumentace, zkoušení atd". Konstantu *c* volí Vaníček rovnu 2.5. Konstanty *b* a *d* pak navrhuje volit v závislosti na tak zvaných třídách programových systémů.

Tento přístup, při kterém je odhad COCOMO proveden nad třídami programových systémů, ukazuje omezení této metody odhadu. Omezení spočívá jednoznačně v nejasnosti vymezení tříd "programových systémů" , pro které je odhad použitelný.

Boehm viz [34] uvádí tři módy (organický, přechodný, vázaný). Vaníček viz [3] uvádí pět tříd programových systémů. Boehem uvádí základní čtyři konstanty (empiricky zjištěné). Praxe však ukazuje případy, kdy je vhodné mít konstantu *a* stejnou pro všechny tři módy. A to v případě krátkých programů.

Vaníček ve své "konkurenční" implementaci COCOMa navrhuje pro svých pět tříd programových systémů zachovat mimo konstanty *a* i konstantu *c* stejnou pro všechny.

Viz [3] "*Třída 2: Ekonomické a vědeckotechnické úlohy se složitými algoritmy zpracování, bez interakce, úlohy pracující s rozsáhlými nedistribuovanými bázemi dat bez paralelního přístupu*:"

$$b \in \langle 1.08 , 1.13 \rangle \quad , \quad d \in \langle 0.35 , 0.38 \rangle$$

Jak je z uvedeného příkladu vidět, pro účel odhadu složitosti je třeba nalézt vhodnou třídu softwarových projektů. Čím přesněji bude možno hodnocený projekt zařadit do této třídy, tím bude odhad přesnější. V případě Boehma jsou tyto třídy (módy) relativně jednoduché. Zvolený softwarový projekt popisují relativně snadno. Problém spočívá v tom, že jsou velmi nepřesné a proto i výsledek odhadu nemůže být přesný. Vaníček se tomuto zjednodušení vyhýbá zavedením přesnějších tříd. Jeho návrh se ale stává méně použitelným.

2.1.3 Kritika odhadu COCOMO

U odhadů COCOMO je podstatné, že pracnost E je vzhledem k rozsahu V superaditivní $(b>1)$. To odpovídá známé skutečnosti, že dvojnásobně rozsáhlý úkol si vyžádá úsilí, které je více než dvojnásobné. Bude-li totiž úkol řešit jeden člověk, bude muset neustále vést v patrnosti větší množství informací a to jej nuntě zdrží. Pokud úkol rozdělíme mezi více řešitelů, nevyhneme se zase ztrátám na jejich nutnou vzájemnou komunikaci.

Naproti tomu nejkratší „rozumný“ čas T je silně subaditivní $(d\simeq 0.35)$. To ovšem platí jen v případě, že počet řešitelů, které lze na úkol nasadit není omezen. Ani zde však čas není nepřímo úměrný počtem řešitelů. Potom odhad stanovený metodou, lze stačit jen za cenu extrémního růstu nákladů.

Pro účel odhadu složitosti softwaru ve fázi specifikace se odhad COCOMO nehodí. Především nejsme schopni stanovit počet řádků zdrojového kódu v okamžiku, kdy software neexistuje. Těžko budeme vypočítávat E – celkovou pracnost projektu. I kdybychom se počet řádků zrojového kódu pokusili odhadnout z “podobně” složitého projetku, narazíme na řadu dalších úskalí. Je to například volba vhodných konstant.

Mimo to, v odhadu COCOMO nejsou zahrnuty některé faktory, které se domnívám, mohou zásadně ovlivnit celkovu pracnost. Mohou to být jak nároky na produktovou jakost, faktory odrážející kvalifikaci řešitelů, tak faktory popisující způsob řízení. Idkyž byla navržena metoda odhadu korekce, má svá omezení. Zavádí do odhadu třídu možných faktorů, které mohou ovlivnit odhad. Faktory jsou klasifikovány dle vlivu na odhad v ordinální stupnici a to:

- velmi nízký,
- nízký,
- normální,
- vysoký,
- velmi vysoký.

V závislosti na této klasifikaci odhad COCOMO násobíme hodnotou blízkou jedné. Je-li faktor hodnocen jako normání, násobíme jej číslem jedna. Je-li vliv hodnocen jako nízký, bude činitel, kterým odhad násobíme menší než 1 a naopak.

Problém v klasifikaci těchto faktorů spočívá v tom, že faktory nejsou disjunktní. Jedna okolnost se tedy může odrazit ve více faktorech a být tedy započítána násobně. Při korekci odhadu COCOMO je pak nutné uplatnit pouze ty skutečnosti, které jsme ještě jinde nepoužili.

Odhady COCOMO mají jako svůj vstup „rozsah“ kódu, representujícího řešení. Z toho plyne, že jsou použitelné „ex-post“. Tedy po té, co je úkol vyřešen. V počátečních etapách, kdy je takový odhad mimořádně cenný je nutné tento „rozsah“ nějak odhadnout na základě požadavků (na fuknčnost, na jakost, na použitelnost,...). K tomu slouží jiné metody (např. metoda Funkčních jednic).

V prvých verzích COCOMO odhadů byl rozsah popisován počtem řádků zdrojového kódu. Záhy se ukázalo, že toto číslo je třeba modifikovat. Například s ohledem na užitý programovací jazyk (různé jazyky jsou různě „hutné“), na složitost algoritmů, apod. Při užití moderních metod tvorby softwaru, jako je například objektový návrh, rozsáhlé užití knihovních tříd objektů, návrhových vzorů a pod., je třeba jako vstup pro „rozsah“ softwaru brát i jiná data než pouhý zdrojový kód programu.

Krom výše uvedených negativ odhadu COCOMO je nutné ukázat na jeho orientaci na strukturované programování. Pro objektově orientované programování je tento odhad nevhodný. Pro odhad složitosti například uživatelského rozhraní, kde je zásadní důraz kladen na vzhled a ergonomii, je zcela nepoužitelný.

Jako pozitivní v odhadu COCOMO pro účel odhadu složitosti software považuji metodu korekce odhadu. Tento způsob klasifikace faktorů ovlivňujících pracnost považuji za použitelný ve své práci.

2.2 Metoda Funkčních jednic

Ve světe existují přístupy k odhadu složitosti ve fázích specifikace záměru. Ve fázi specifikace se vyjasní, co vlastně má být předmětem řešení, to jaké požadavky má vytvářený produkt splňovat viz [9,14,15]. Jedním z předních přístupů k odhadu složitosti (ve vztahu k jakosti softwaru) je Function point (metoda Funkčních jednic) viz [11]. Autorem této metody je A. Albrecht, který se na jejím vývoji podílel v laboratořích IBM v 70 letech minulého století. Vznikala za účelem odhadu složitosti projektů informačních systémů.

Tato metoda není jediným uceleným návodem, ale rozsáhlou skupinou návodů, které se navzájem velmi liší, v závislosti na tom, jaký produkt má být vytvářen a pomocí jakých nástrojů. Viz [3] „*Společným rysem metod Funkčních jednic je to, že složitost odhadují jako součin dvou faktorů. Prvý z nich je založen na funkcích, které jsou od produktu vyžadovány, druhý na podmínkách, které pro vytvoření produktu objektivně jsou*“.

Na první pohled je zjevné, že je to dosti neúplný základ pro odhad složitosti. Složitost nepochybně jestě závisí na mnoha dalších faktorech. Některé z nich již byly popsány v předchozí kapitole COCOMO. V metodě funkčních jednic se tyto faktory uplatňují nepřímo. Problém je ovšem v tom, že v etapě specifikace mnoho informací nemáme. Vaníček viz [3] v Měření a hodnocení jakosti software trefně uvádí: “T*en kdo chce přesto odhad složitosti (a tedy i pracnosti) provést, se dostává do situace popsané anegnotou, ve které nešťastník hledá v noci před svým domem ztracené klíče pod lucernou elektrického osvětlení ne proto, že si myslí že je vytrousil právě tam, ale proto, že je to jedinné místo, kde na to vidí”*.

2.2.1 Matematický model metody Funkčních jednic

Ve světe existuje celá řada popisů metody Funkčních jednic (v agličtině Function point, proto zkracujeme na FP). Například A.Albrech viz [35], Vaníček viz [3], Konsorcium Function Point viz [36]. V disertační práci popis přebírám od svého školitele, protože se domnívám, že je z výčtu popisů nejpřehlednější.

Vaníček viz [3] přehledně shrnuje metodu výpočtu Funkčních jednic jako:

- *V*- je obecně označován objem funkcí, které potřebujeme zjisit.

- *IN (vstup)* – je označován počet logicky různých vstupů do zpracování. Každý vstup se počítá tolikrát, kolik různých typů či formátů dat je zpracováváno.

- *OUT (výstup)* – počet logicky různých výstupů zpracování (počítáme obdobně jako IN).

- *FILE (soubor)* – počet interních logických souborů.

- *INTF (vázané vstupy)* – počet logických souborů, které systém sdílí s jinými softwarovými celky.

- *INQ* – počet logicky různých výstupů s čekáním na vstup (dotazů), zabezpečovaných systémem.

Během vývoje metody FP docházelo k jejímu vylepšování. Na snadě je každý prvek FP:

- IN,
- OUT,
- FILE,
- INTF,
- INQ,

ještě ohodnotit "nějakou" váhou "*w"*. Typickým příkladem může být klasifikace jednotlivých prvků podle stupnice ordinálního typu viz [3]:

jednoduchý – průměrně složitý – složitý

Kdy stanovíme pro každý (IN,OUT atd..) prvek jednoté měřítko.

Viz [3]:

- ***jednoduchý*** *– Vstupní věta obsahuje málo (např. méně než 10) položek, je přehledně strukturovaná, vstup je prováděn do jedinného souboru.*

- ***složitý*** *– Vstup do více než 3 soborů nebo složitá struktura (4 a více úrovní).*

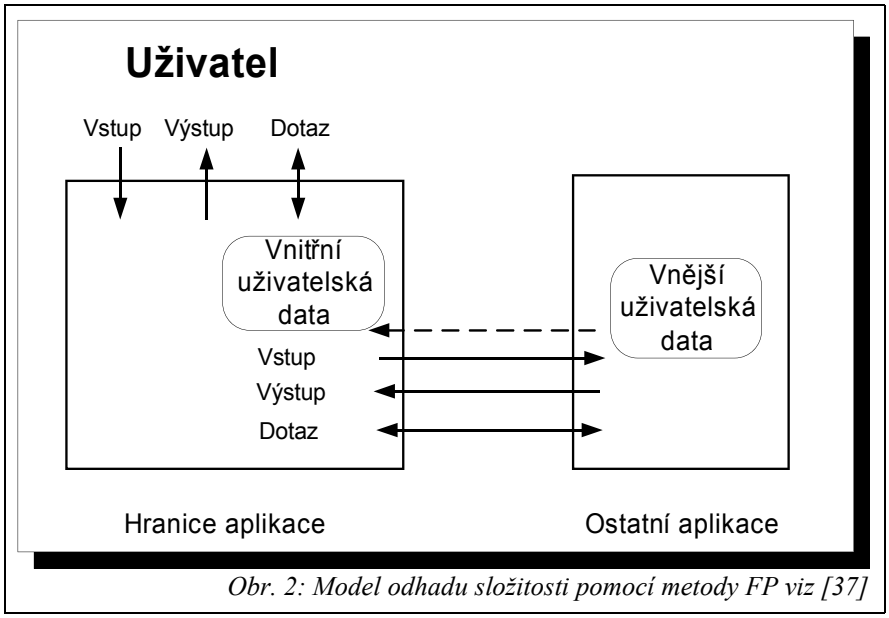
- ***průměrně složitý*** *– Nepatří do jednoduchých ani složitých vstupů.*

Výsledný výpočet složitosti V pak získáme jako součet vah všech takto vymezených funkčních jednic" viz Tabulka 1.

Funkce / váha *w*	jednoduchý	průměrně složitý	složitý	Celkem
IN	3	4	6	
OUT	4	5	8	
FILE	4	5	8	
INTF	2	5	8	
INQ	3	7	9	
Celkem V				***V***

Tabulka 1: Tabulka výpočtu složitosti FP

Schéma výpočtu odhadu složitosti pomocí metody Funkčních jednic lze vidět na obrázku Obr. 2.



Obr. 2: Model odhadu složitosti pomocí metody FP viz [37]

V druhém kroku metody Funkčních jednic hodnotíme podmínky realizace projetku.

Faktory, které ovlivňují podmínky realizace se liší od projektu a mohou vypadat například takto:

Obecné charakteristiky systému		Krátký popis
1	Datová komunikace	Kolik komunikačních zařízení podporuje přenos nebo výměnu informací s aplikací nebo systémem?
2	Distribuované zpracování dat	Jak jsou řízena, distribuovaná data a zpracování funkcí?
3	Výkon	Byla časová odezva nebo výkon v souladu s požadavky uživatele?
4	Intenzita využití konfigurace	Jaká je intenzita využití současné HW platformy, na které budou aplikace vykonávány?
5	Transakční míra	Jak často jsou transakce zpracovávány – denně, týdně, měsíčně, atd.?
6	On-line vkládání dat	Jaké procento informací je vkládáno on-line?
7	Výkonnost koncového uživatele	Byla aplikace navržena, aby zlepšila pracovní výkon koncového uživatele?
8	On-line aktualizace	Kolik vnitřních logických souborů je aktualizováno on-line?
9	Složité zpracování	Disponuje aplikace rozsáhlým logickým a matematickým zpracováním?
10	Znovupoužitelnost	Byla aplikace vyvinuta, aby uspokojila jednu nebo více uživatelských potřeb?
11	Jednoduchost instalace	Jak složitá je úprava a instalace?
12	Provozní jednoduchost	Jak účinně a/nebo automatizovaně je aplikace startována, zálohována a obnovena?
13	Multifunkční využití	Byla aplikace speciálně navržena, vyvinuta a podporována, aby mohla být instalována pro multifunkční využití v rámci organizací?
14	Ulehčení změn	Byla aplikace speciálně navržena, vyvinuta, aby podporovala lehké zavedení změn?

Tabulka 2: Převzato z Struska viz [37]

Poznámka: *Na první pohled je zřejmé, že se jedná o „nějaké“ manažerské charatkeristiky vývoje IS. Jednotlivé charakteristiky se však liší projekt od projektu. Pro odhad složitosti je tudíž vhodné je nějak formalizovat. Z tohoto důvodu zavádím třídu faktorů složitosti FS, které představují mnou navržené základní manažerské charakteristiky vývoje informačních systémů. Ty slouží k odhadu složitosti softwarového produktu a tudíž i k odhadu složitosti celého informačního systému (který se skládá z dílčích softwarových produktů). Jejich popis je uveden v kapitole 3.*

Každý ze zvolených faktorů opět klasifikujeme podle ordinální stupnice:

- 0 - Faktor nemá žádný vliv nebo neexistuje.
- 1 - Faktor má vliv malý.
- 2 - Vliv existuje, je o něco menší než průměrný.
- 3 - Faktor má vliv průměrný.
- 4 - Faktor má vliv významný.
- 5 - Faktor je velmi významný a ovlivní podstatně architekturu řešení i realizaci.

Jednotlivé ohodnocené faktory sečteme a získáme číslo např. "*PF"* viz [3]. *„Pro zvolený případ definujeme (na základě zkušeností nebo poučením se z literatury) například vzorec"*:

$$F \text{ (hodnocení realizace)} = \quad 0.7 + 0.01 \cdot PF$$

Výsledná složitost ve funkčních jednicích se vypočítá dle vzorce:

$$FP = V \cdot F \cdot \beta$$

V - je objem funkcí,

F - je hodnocení realizace podmínek (dle zvolených vah),

β - je konstanta závislá na jednotkách měření složitosti.

2.2.2 Kritika metody Funkčních jednic

Kritika metody Funkčních jednic spočívá ve skutečnosti, viz [3] *„že zde přetrvává značná subjektivnost hodnocení a jsou hrubě porušována pravidla o tom, co je a není smysluplné při práci s hodnotami získanými měřením v měřicí stupnici daného typu."*

V současnosti lze metodě vytknout její orientaci na „dávkové zpracování". Alespoň ty její modifikace, které jsou publikovány a uváděny v literatuře orientaci na dávkové zpracování hromadných dat, navržené metodami strukturovaného návrhu, nemohou zapřít.

Těmto problémům se někteří autoři variantních metod FP snaží předejít různými úpravami. Další odvozené metody jsou například:

- International Function Point Users Group (IFPUG).
- Software Productivity Research Feature/Function Point (SPRF Feature/Function Point).
- Metoda Use Case Point.
- a další.

2.3 Metoda Use Case Point

Kritika metody FP a jejích odvozenin vedla v roce 1993 Gustava Karnera viz [24] k vyvinutí metody, založené na tak zvaných "Use Case Points".

Use Case je anglický název pro případ užití. Případ užití je způsob jak zachytit požadavek na funcionalitu budoucího systému, za dodržení pravidel definovaných v metodě Unified Process. Use Case Points lze zjednodušeně přeložit jako body případů užití.

Výsledná metoda byla diplomovou prací Gustava Karnera na švédské University of Linköping viz [24]. Práva k užívání a kopírování na ní vlastní firma Rational Software. Tato firma mimo jiné je aktivním rozšiřovatelem sjednoceného (unifikovaného) modelovacího jazyka UML[4,22] a tvůrcem Rational Unified Process. Těmito produkty se detailně zabývám v dalších kapitolách 3. a 4.

Karner založil metodu odhadu bodů případů užití na základě předpokladu, že funkčnost systému vychází z uživatelem definovaných případů užití – Use Cases [4,22]. Tyto případy užití jsou základem pro odhad velikosti informačního (nebo jiného softwarového) systému.

2.3.1 Matematický model UCP

Pro výpočet odhadu složitosti na základě bodů případů užití je neprve potřeba zjistit počet aktorů a počet případů užití. Viz Tabulka 3 je jedním z možných případů, jak fuknční body případů užití počítat. Slouží jako vzorový příklad.

Role	Případ užití	Počet případů užití
Administrátor	Administrace_1	1
Uživatel	Zadávání dat_1, Čtení sestav_1	2
Manager	Čtení sestav_1,Čtení sestav_2	1
Celkem		4

Tabulka 3: Vzorový příklad výpočtu bodů případu užití

V tabulce jsou uvedeni aktoři a jejich role v jednom daném ukázkovém příkladu. Popis rolí není pro výpočet bodů případů užití důležitý. Je ale důležité vědět, kolik aktorů má vazbu na jeden případ užití. Body jsou případu užití udělovány na základě počtu aktorů, které jej využívají. V našem případě má případ užití Čtení_sestav_1 dva aktory. To znamená, že je patrně složitější, než ostatní případy užití. Karner navrhuje klasifikovat vznikající případy užití za pomoci ordinální stupnice s běžně využívanými hodnotícími stupni:

Typ případu užití	Váha w
Jednoduchý	1
Středně složitý	3
Složitý	6

Tabulka 4: Příklad přiřazení váh

jednoduchý – středně složitý – složitý

Podle “složitosti” se pak jednotlivému případu užití přidělují váhy obdobně jako na vzorovém příkladu v Tabulce 4.

Dále, podobně jako v metodě FP, je nutné ohodnotit jednotlivé dílčí technické faktory prostředí od 0 – nemá vliv, do 5 – má zásadní vliv. Po dosazení do obdobných vzorců viz Karner [24] získáme počet bodů případu užití.

Karner viz [24] dále navrhuje zavést míru – pracnost v člověkohodinách na jeden bod případu užití. Karner doporučuje tuto míru rovnou 20. Z vlastních zkušeností při vedení projektů i dle zahraniční literatury viz [25] navrhuji zvolit hodnotu v intervalu <15, 25> na jeden bod případu užití.

Nakonec je potřeba vynásobit počet bodů případu užití zvolenou mírou a sečíst výsledky. Vyjde celková odhadovaná složitost pro softwarový produkt.

2.3.2 Kritické zhodnocení metody UCP

Metoda UCP, stejně tak jako metoda Funkčních jednic, hrubě porušuje pravidla přípustných transformací s hodnotami měřenými pomocí ordinálních stupnic. Jistě - jak uvádí Vaníček viz [3] “ *v době specifikace požadavků na systém nemáme mnoho podkladů pro měření.”* Z mé zkušenosti využití jiných než ordinálních stupnic je v době specifikace požadavků na systém prakticky nemožné. Tudíž Karner správně použil ordinálni stupnici ke klasifikaci složitosti jednotlivých případů užití. Otázkou, kterou si dovolím položit je:

“*Nestačilo by pouze klasifikovat případy užití na složitý, středně složitý a jednoduchý? Pak, porovnáním naměřených hodnot (i sčítání hodnot naměřených v ordinálních stupnicích nemá valný smysl) určit: objektivně máme nejvíc případů užití středně složitých a nejméně máme složitých. Tudíž se složitost systému pohybuje od středně složitého k jednoduchému a dle zkušenosti z předešlých projektů je doba na realizaci X člověkohodin?*

Odpadlo by relativně složité vážení funkčních bodů, které již z principu přijatelných transformací v ordinálních stupnicích nemá smysl.“

2.4 Další metody odhadu složitosti

Odhad složitosti softwaru nedá spát celé řadě lidí. I já jsem prodiskutoval mnoho hodin na toto téma se svými kolegy viz [A.2,A.3,A.18]. Jako vhodná cesta k odhadu složitosti softwaru se zdá dekomponovat celý problém na dílčí softwarové komponenty. Ty by měly být tak jednoduché, že by mělo být snadné při projektovém řízení jejich složitost odhadnout.

2.4.1 Složitost projektování v malém

"*Jedná se o odhad složitosti programového modulu, jenž je jako celek navržen jedním autorem nebo jedním koletivem autorů.*" viz [3].

S*ložitost v malém lze zjišťovat:*

- *Na základě fyzikálních údajů o software, jako je počtu příkazů v programu, počtu řádků programu, počtu užitých operandů atd.*

- *Na základě grafu řízení algoritmu.*

- *Na základě strukturálního rozkladu algoritmu"* viz [3].

První způsob odhadu není ideální. Patrně počet příkazů v programu, či počet řádků máme k dispozici, až když je program hotový. Složitost však můžeme odhadovat na základě analýzy grafu řízení.

2.4.2 Složitost na základě analýzy grafu řízení

Způsob odhadu složitosti na základě grafu řízení je patrně obecně použitelný. Je také teoreticky nejpropracovanější a variantu grafu řízení používají nejrůznější metody jako Unified Process viz [7] nebo RUP viz [16] a jiné. Má však některé nevýhody.

2.5.2.1 Definice grafu řízení

Orientovaný graf $G = (V,H,\Phi)$ se nazývá grafem řízení, pokud jsou splněny následující podmínky:

1. Množina vrcholů V se skládá z pěti navzájem disjunktních podmnožin

$$V = Z \vee K \vee P \vee R \vee S \quad .$$

a) Z má jediný uzel z, který má vstupní řád 0 a výstupní řád 1. Jde o uzel vyjadřující začátek programu. Nemá žádný vstup a vystupuje z něj jediná hrana.

b) K má jediný uzel k, který má vstupní řád 1 a výstupní řád 0. Jde o uzel vyjadřující konec programu. Nemá žádný výstup a vstupuje do něj jediná hrana.

c) P obsahuje uzly s jedním vstupem a výstupem. Mají vstupní i výstupní řád roven 1. Tyto uzly znázorňují výkonné kroky programu.

d) R obsahuje uzly s jediným vstupem ale dvěma výstupy. Tyto uzly znázorňují rozhodovací bloky.

e) S obsahuje uzly s dvěma vstupy a jediným výstupem. Tyto uzly znázorňují slučovací bloky.

2. Každý uzel grafu G leží aspoň na jedné cestě, která spojuje uzel z s uzlem k.

2.4.2.2 Kritika grafu řízení jako nástroje pro dohad složitosti

Kritika grafu řízení jako nástroje pro odhad složitosti spočívá v tom, že různé vstupy pro automatické generování programů u prostředků pro podporu jejich návrhu mohou vést na stejný graf řízení algoritmu. Problém spočívá v tom, že jednoho cíle lze dosáhnout různými cestami. Zatím co v jednom případě je program složen ze snadno pochopitelných a přehledných struktur, v druhém případě stejně funkční program může být navržen pomocí skokových príkazů jako je *goto.* Toto řešení ja samozřejmě mnohem složitější než řešení prvé. Vede však ke stejnému grafu řízení algoritmu.

Kritika vyčítá grafu řízení i to, že je v podstatě jen zjednodušeným vývojovým diagramem. Dnes se vývojové diagramy, často ke škodě vznikajících softwarových produktů, již nepoužívají. Máme k dispozici řadu jiných, nových nástrojů. Firma Rational Software v její metodě řízení vývoje software používá modelovací jazyk UML. Diagramy aktivit jsou velmi podobné grafu řízení a jsou také zjednodušeninou vývojových diagramů. Tyto prostředky jsou však často dostupné až v okamžiku implementace a to zpravidla bývá pozdě. Je proto třeba hledat vstupy pro odhad složitosti, jenž jsou k dispozici co možná nejdříve.

2.4.2.3 Míry vycházející z grafu řízení

2.4.2.3.1 McCabeovy míry

Tato míra je založena na **počtu rozhodovacích uzlů** v grafu řízení. Navrhl ji McCabe v roce 1976.

Označíme-li v grafu

$G = (V,H, \varphi)$,

card (V) počet vrcholů a card (H) je počet jeho hran, pak je McCabe (označujeme ji *MCC*) míra definována vztahem:

$$MCC = (G) = \text{card}\ (H) - \text{card}\ (V) + 2.$$

Této míře se někdy také říká "*cyklomatická složitost grafu řízení"* viz [3]. Ta udává maximální počet "lineárně nezávislých" cest od počátečního uzlu „*z*" do koncového uzlu „*k*" v grafu řízení. Viz [3] "*Každá cesta od **z** do **k** prochází některými hranami grafu (některými i vícekrát). Lze ji tedy přiřadit celočíselný vektor o počtu složek card (H), který bude udávat, kolikrát byla při této cestě užita daná hrana. Tvrzením, že jsou cesty lienárně nezávislé budeme rozumět to, že jsou lineárně nezávislé vektory, které jim tímto způsobem přísluší*" viz [3].

2.4.2.3.2 Kritika McCabeovy míry

Kritika McCabeovým mírám vyčítá že pro odhad složitosti nejsou o nic výhodnější, než triviální míra – počet rozhodovacích bloků v grafu řízení. Ta je nazývána logickou složitostí grafu řízení.

Další námitkou je to, že nebere ohled na hloubku vnoření struktur do sebe. Patrně platí, že vnořené struktury jsou složitější, než struktury řazené za sebou. Konečně nebere vůbec ohled na porušení zásad strukturovaného programování. Například míra dvou cyklů v sekvenci za sebou je táž jako míra vnořených cyklů a dokonce stejná jako míra zcela nevhodné konstrukce dvou překrývajících se cyklů nebo skoků uvnitř cyklu.

Tyto námitky se snaží řešit další míry.

Jedná se o:

- Harrisonovy,
- Magelovy,
- Piwowarského,
- Zuseho,

modifikace McCabeho měr. Čtenář se s jejich popisem může lépe seznámit v dostupné litaratuře Měření a hodnocení jakosti informačních systémů viz [3].

Další kritikou je viz [3], "*že míry vycházející z grafu řízení jen těžko odrážejí prohřešky proti zásadám strukturovaných a dobře navržených programů. To je však nutné posoudit případ od případu.*"

Další námitkou proti užití míry vycházející z grafu řízení je skutečnost, že je velmi snadné ji "zneužít". Bude-li založena například na množství řádků zdrojových kódů programu a bude-li podle toho hodnocena produktivita programátora, tak snadno může bez změny funkčnosti programu program "přizpůsobit", aby hodnocení bylo pro něj co nejpříznivější. Zabránit takovým úpravám je takřka nemožné.

Ze svých praktických zkušeností potvrzuji oprávněnost kritik McCabeho měr. Domnívám se však, že i přes výše stanovená omezení je způsob odhadu složitosti na základě grafu řízení, účinným a vhodným způsobem. Narozdíl od metod FP nebo UCP minimálně operuje s hodnotami naměřenými v ordinální stupnici.

V moderních přístupech řízení tvorby software se využívá téměř zásadně případů užití (Use Cases), které popisují chování systému z pohledu uživatele. Jejich inverzí je pak scénář. Ten popisuje chování systému z "pohledu" systému. Scénář je možné zobrazit pomocí grafu řízení. Využití triviální McCabeovy míry je tu na snadě.

Samozřejmě toto zobrazení naráží na skutěčnost že jedna řádka ve scénáři např.

„Systém zobrazí uživateli dialogové okno s nabídkou"

může být chápána jako jedna akce a být tudíž triviální. Pod pojmem Dialogové okno s nabídkou se ale může skrývat složitý dotaz do databáze. Zobrazíme-li scénář pomocí grafu řízení, nepostihneme tuto "skrytou" složitost a odhad bude nepřesný.

Obhajobou měr založených na grafu řízení je podle mě skutečnost, že v relativně malém, uzavřeném, prostředí na relativně malém množství projektů je analytik či designer tvořící případy užití (Use Casy) a scénáře po čase schopen psát bez skrytých složitostí typu „dialogového okna". Analytik se totiž praxí naučí odhalit složitost skrytou v uživatelském zadání. Domnívám se, že případy užití a scénáře jím tvořené pak mohou být dostatečně přesné, aby na ně bylo možné applikovat odhad složitosti na základě grafu řízení.

2.4.2.4 *Ordinální míry strukturovanosti*[7]

Zásady strukturovaného programování ihned vedou k ordinální stupnici viz [3]:

- složitost 1 = *strukturovaný* návrh či program. Bylo použito pouze sekvence, selekce a iterace, hierarchicky v potřebném počtu úrovní.

- složitost 2 = *dobře navržený* návrh či program. Bylo užito pouze sekvence, selekce a cyklů s více výskoky, hierarchicky v potřebném počtu úrovní.

- složitost 3 = *nestrukturovaný* ale intervalově reducibilní návrh či program.

- složitost 4 = *intervalově* nereducibilní návrh či program.

2.4.2.4.1 Kritika ordinální míry strukturovanosti

Na základě této míry, lépe stupnice, není možné odhadnout pracnost. Může být však indikátorem jakosti provedení. V případě strukturovanosti návrhnu může být účelná pro rozpoznání kvalitního návrhu od návrhu "horkou jehlou", nebo amatérského pokusu o řešení metodou „jak mě to právě napadlo".

7 Tuto kapitolu volně cituji od svého školitele Prof. Jiřího Vaníčka [3]. Uvedena je z důvodu ucelení problematiky. V současné době se již téměř vždy programuje objektově. Míra strukturovanosti však může být použita ve fázích specifikace, kdy seznam požadavků je strukturován do případů užití a scénářů.

2.4.3 Složitost na základě přehledného hierachického rozkladu[8]

Vaníček viz [3] navrhuje postup, který by umožnil odhad složitosti návrhu realizace softwaru, který se, místo o graf řízení, opíra o tak zvaný hierarchický rozklad úlohy. Jeho hlavní výhody by měly být především:

- Skutečnost, že se opírá o postup návrhu shora dolů, který je v moderních přístupech k projektování informačních systémů a software jednoznačně převládající.

- Skutečnost, že není příliš snadné přizpůsobit uměle návrh tak, aby při hodnocení bylo dosaženo výsledků podle přání řešitele.

- Vaníček viz [3] však uvádí: „*nevýhodou navrhovaného postupu je skutečnost že: metodu lze užít jen pro strukturovaný návrh nebo jeho jistá rozšíření, mezi která patří i dobře navržené programy. Užití pro systémy, které tento požadavek nesplňují, je možné jen nepřímo a za cenu značných komplikací.*“

2.4.3.1 Kritika odhadu složitosti na základě přehledného hierachického rozkladu

Stejně tak jako analýza grafu řízení vychází v podstatě ze znázornění software vývojovým diagramem, je i hierachický rozklad založen na filosofii a technice strukturogramů. Tento přístup však neodráží objektový přístup k návrhu systému.

8 Kapitola je volnou citací – viz [3].

2.4.4 Složitost projektování ve velkém

V předchozích kapitolách jsem hovořil především o postupech měření složitosti jednoho programového modulu. Většinou se však softwarový produkt skládá z většího množství modulů. Ty spolu vzájemně spolupracují. To, jak spolu moduly spolupracují, může zásadně ovlivnit celkovou složitost softwarového produktu. Tuto složitost nazýváme složitost programování ve velkém nebo také intermodulární složitost.

Obecně platí pravidlo, že spolupráce jednotlivých modulů je tím jednodušší, čím více platí následující pravidla viz [3]:

- *„Každý modul řeší pouze vzájemeně svázané úkoly, které spolu úzce souvisejí.*
- *Dva různé moduly spolu spolupracují jednoduchým a snadno srozumitelným způsobem“.*

Prvou vlastnost nazýváme soudržností, druhou nazýváme spřaženost.

2.4.4.1 Soudržnost

Yordon a Constantin v roce 1979 navrhli měřící stupnici pro hodnocení míry soudržnosti. Zdokonalil ji Fenton. Stupnice je ordinálního typu a má následující charakter[9]:

- **Funkční soudržnost:** Modul realizuje jedinnou funkci.
- **Sekvenční soudržnost:** Modul realizuje několik funkcí, avšak postupně.
- **Komunikační soudržost:** Modul realizuje více funkcí, avšak na různých strukturách.
- **Procedurální soudržnost:** Modul realizuje několik funkcí, které navzájem souvisejí pomocí jedné spojené procedury, spravující společná data.
- **Logická soudržnost:** Modul realizuje několik navzájem logicky provázaných funkcí nad společnými daty, pomocí "vedlejších efektů" na sdílených datech.
- **Náhodná soudržnost:** Není jasně definován vztah mezi více funkcemi, které modul realizuje.

Vaníček uvádí viz [3] "*Existují i přesnější způsoby měření soudržnosti. Například míra založená na tak zvaných datových řetězech, navržená Biemanem a Ottem. Jsou však příliš složité a ne zcela uspokojivě odrážejí intuitivní požadavky*".

9 Opět se jedná o volnou citaci mého školitele Prof. Jiřího Vaníčka, viz [3]

2.4.4.1.1 Kritika míry pro odhad soudržnosti

Kritika této míry spočívá především v nejasnosti užitých pojmů a tím i navrženého dělení. Kromě toho jde o ordinální stupnice, což omezuje interpretaci výroků získaných výpočtem ze získaných měr.

Ze mých zkušeností je jakákoliv složitá míra v praxi velmi špatně nasazována. Získané výsledky často nejsou zcela přesvědčivé. Často pak v praxi stojíme před rozhodnutím:

- Realizujeme-li jednoduchou míru s horší schopností odhadu a přičteme k ní nejakou rezervu:

$$Celková\ složitost = míra\ složitosti + REZERVA$$

- nebo realizujeme-li složitou míru.

2.4.4.2 Spřaženost modulů

Pro hodnocení zpřažeností modulů navrhuje Fenton následující dvou úrovňovou ordinální stupnici:

- Moduly spolu nespolupracují.
- Moduly si předávají pouze data a to pouze pro čtení.
- Moduly si předávají data pro čtení i zápis pomocí parametrů.
- Moduly si pomocí parametrů předávají nejen data, ale i příznaky, pomocí kterých modifikují svoji práci.
- Moduly pracují nad vymezenými společnými (globálními) daty.

- Moduly mají možnost vzájemně měnit jakákoliv svoje data.

- Moduly mají možnost vzájemně zasahovat do svých programových kódů.

Pokud se nepodaří rozhodnout podle vyšší úrovně navrhuje Fenton rozhodnout podle druhé, nižší úrovně.

Druhá a nižší úroveň hodnotí objem komunikace.

Fenton navrhl pro spřaženost modulů X a Y míru viz[3]:

$$FEN(X,Y) = T - 1 + N / (N+1)$$

Kde T je největším číslem ohodnocený typ vzájemné komunikace a N je počet, kolikrát je tohoto typu užito.

2.4.4.2.1 Kritika Fentonovy míry pro spřaženost modulů

Tato klasifikace je použitelná pro rozlišení jednotlivých stupňů spřaženosti. Není možné ji využít k výrokům typu: komunikace typu 6 je třikrát tak složitá jako komunikace typu 2. Tak tomu zjevně není. Jedná se o typickou ordinální stupnici s jejími omezeními.

2.4.5 Složitost v objektovém prostředí[10]

"*Problémy s komunikací mezi moduly vedly k požadavku tuto komunikaci maximálně zjednodušit. Vývoj metodik pro návrh a vývoj software vedl k požadavku chápat modul nejen jako program, ale jako nedílný objekt, tvořený daty a procedurami, jenž tato data mění. Postupně se ukázalo, že některé datové abstrakce mají společné rysy a některé funkce, které mají být jimi zajišťovány, není nutné zabezpečovat duplicitně, ale jednou, společně*" viz [3,38].

Citovaným způsobem se došlo ke konceptu objektu a posléze k objektovému programování.

Důležitým rozdílem o proti strukturálnímu programování je v tom, že program již není chápán jako sled instrukcí, ale jako model světa řízený požadavky objektů. To značně komplikuje zavedení měr složitosti. Zatím co návrh složitosti jednotlivých objektů, pokud je nemůžeme použít z knihoven, je třeba stanovit klasickými mírami pro programování v malém, problém intramodulární složitosti zůstává.

Z měr použitelných pro odhad složitosti objektového návrhu můžeme ukázat některé míry navržené Chidamberem a Kemerervizem viz [38]:

2.4.5.1 Hloubka stromu dědičnosti – DIT (depth of inheritance tree)

Tato míra *DIT (C)* je pro každou třídu definována jako maximální délka cesty od kořene stromu dědičnosti k vrcholu, který v hierarchii tříd odpovídá dané třídě.

DIT objektového návrhu pak určíme jako maximum z hodnot *DIT* (*C*) pro jednotlivé třídy. *DIT* je ordinálního typu. Není tedy smysluplné uvažovat součet.

Význam míry spočívá v tom, že čím je tato délka větší, tím více metod může třída zdědit a tím obtížnější je určit její chování.

10 Kapitola je volnou citací mého školitele Vaníčka viz [3] a Chidamberera a Kemerera viz [38]

DIT je jedna z šesti měr, které Chidamber a Kemerer navrhli jako míry složitosti ve velkém při objektovém návrhu. Složitost ve velkém je potřeba odhadovat užitím všech šesti těchto měr, protože některé jsou „protichůdné“ (jednu lze snížit, na úkor zvětšení jiné). Jak tyto míry sumarizovat ale autoři nenavrhli. Jejich popis uvádím pro ucelení kapitoly.

2.4.5.2 Počet synů – NOC (number of children)

Pro danou třídu *C* udává *NOC* (C) počet synů (to je přímých potomků) vrcholu, který v stromu dědičnosti představuje třídu *C*. *DIT* celého návrhu je pak součtem příspěvků *DIT* (*C*) všech tříd *C*.

Protože synové dědí metody, znamená vyšší hodnota *DIT* (*C*) vyší úroveň abstrakce návrhu. Může vést k potřebě testů metod ve třídách, které je dědí a tedy k vyšší složitosti realizace.

Míry *DIT* a *NOC* si vzájemně „konkurují“. Snaha k snížení hodnoty jedné z nich může vést k zvýšení hodnoty druhé a opačně.

2.4.5.3 Spřažení mezi třídami – CBO (coupling between object classes)

Pro každou třídu *C* udává míra *CBO* (*C*) počet tříd, na jejichž historii třída *C* závisí tím, že volá jejich metody nebo používá jejich instanční proměnné.

CBO celého návrhu lze pak stanovit jako počet příspěvků *CBO* (*C*) všech tříd v návrhu.

Existence spřažení mezi třídami narušuje modularitu tříd a znemožňuje jejich užití v jiné aplikaci. Vysoký stupeň spřažení tříd znesnadňuje testování celku a zvyšuje citlivost systému ke změnám. Opakem spřažení je tak zvané „zapouzdření“, jehož dosažení nebo se k němu alespoň přiblížení, bývá u objektového návrhu cílem.

2.4.5.4 Odezva třídy – RFC (response of class)

Odezva třídy *RFC* (*C*) je deifnována jako počet metod, které jsou buď metodami tříd (*C*), nebo paří do jiné třídy, ale některá metoda třídy (*C*) je volá. Jde tedy o počet metod, jejichž provedení může být potencionálně vyvoláno tím, že některý objekt třídy (*C*) přijal zprávu. Přitom se počítají pouze metody vyvolané buď přímo přijatou zprávou nebo aktivací metody, kterou zpráva vyvolala. Metody případně volané na dalších úrovních se již nezapočítávají. Budou totiž započteny do míry *RFC* u jiné třídy.

Pokud se čtenář disertační práce opravdu dostal až k této kapitole, má můj obdiv. Vážím si jej pro jeho pečlivost a obdivuji jej. Já osobně mám sklon obdobně nezáživnou práci prolistovat, proletět a tudíž přehlédnout obdobný odstavec, jako je tento. Řešit problém odhadu složitosti software či celého projektu s využitím umělé inteligence mě opravdu baví. Psát ale hromady teorie do práce podobného typu je pro mě značně ubíjející. Snad to na kvalitě práce není tak moc znát. Nebýt však Prof. Jiřího Vaníčka, který mi práci připomínkuje, asi bych ji do hromady nedal. Jiří je pro mne člověk neuvěřitelně moudrý a snad jen tímto způsobem mohu zvěčnit můj obdiv k němu.

Pokud čtenáře cokoliv z práce zaujalo, včetně tohoto odstavce, rád si s ním o problematice popovídám a pozvu jej na nějakého panáčka či pivečko. Podobný odstavec zakomponovat do mé diplomové práce se nepodařilo. Byl jsem moc zbabělý. Dnes k mým třicátým narozeninám se již nebojím a hrdě se k němu hlásím. Kdo se dočetl až sem, má u mě panáka a obdiv.

RFC (*C*) celého návrhu je opět součtem příspěvků *RFC* (*C*) všech tříd návrhu.

Velký počet volaných metod zvyšuje náročnost pochopení návrhu a znesnadňuje testování. Na druhé straně, vyšší hodnota *RFC* znamená kompaktnější software, což je významné zvlášť u tříd, které mají být užívány standardně, jako opětovně použitelné objekty.

Míry *CBO* a *RFC* se též do značné míry překrývají. Zdá se, že míra *RFC* vystihuje spřažení metod lépe. Vzniká tedy otázka, zda existují důvody pro zachování míry *CBO* pro spřažení měr v souboru a zda jsou dostatečně závažné.

2.4.5.5 Nedostatečná soudržnost tříd v metodách – LCOM (lack of cohesion metric)

Tato míra si klade za cíl postihnout, zda jsou třídy dostatečně soudržné, jinými slovy, zda návrh nelze zjednodušit tím, že třídu rozdělíme na dvě či několik tříd, které budou realizovat jednotlivé metody odděleně a budou vzájemně spřaženy.

Dvě metody M_1 a M_2 téže třídy pokládáme za spřažené, pokud průnik $I_1 \wedge I_2$ jejich instančních proměnných je neprázdný.

Označme *p* počet dvojic metod třídy *C*, které jsou spřažené a *q* počet dvojic metod třídy *C,* které spřažené nejsou. Za dostatečnou soudržnost třídy *C* je považován případ, kdy počet spřažených dvojic je alespoň tak velký, jako nespřažených. Další zvyšování soudržnosti již není považováno za přínos.

Míra *LCOM* je tedy definována takto:

$$LCOM(C) = 0, \quad \text{je-li} \quad p \leqslant q,$$
$$LCOM(C) = p - q, \quad \text{je -li} \quad p > q.$$

Míra je pouze ordinálního typu. Proto stanovení celkového nedostatku soudržnosti návrhu vede k potížím. Definovat jej jako součet by znamenalo porušení pravidel teorie měření. Východiskem může být definice pomocí maxima. Celkový nedostatek soudržnosti je roven nejvyšší hodnotě *LCOM* (*C*) pro „nejméně soudržnou“ třídu *C*.

Soudržnost je pochopitelně u objektového návrhu žádanou vlastností. Její nedostatek signalizuje nepřehledný návrh a větší nebezpečí chyb při implementaci.

Míra do jisté míry „konkuruje“ jiným. Neodstatek soudržnosti lze „léčit“

rozdělením málo soudržné třídy na několik tříd. To má samozřejmě své dopady. Může znamenat zvýšení míry *DIT* a nebo *NOC,* a protože některé metody mohou být spřažené, zvýší patrně i míry *CBO* a *RFC*.

2.4.5.6 Váha metod pro třídu – WMC (weight of method per class)

Tato míra je definována pro každou třídu *C* vzorcem:

$$WMC(C) = \sum_{j=1,} \ldots, {}_{n}c_{j},$$

jako součet složitostí c_j všech metod třídy *C*. Tyto složitosti je třeba stanovit jako složitosti imperativních programů stanovených některými mírami navržených pro programování v malém.

Souhrnná složitost objektového návrhu je pak součtem příspěvků *WMC* (*C*) všech tříd, které se v návrhu vyskytují.

Význam této míry pro čistě objektovou složitost je mimo jiné v tom, že větší počet množství metod a jejich vyšší složitost má za následek obvykle větší aplikační specifičnost třídy a tím menší možnost použití metod při dědění.

2.4.5.7 Kritika měr složitosti v objektovém prostředí

Navržené míry pro odhad složitosti v objektovém prostředí není jasné, jak je kombinovat na vzájem. Míry lze zjistit, až když jsou navrženy třídy objektů a typy zpráv, které si mají objekty posílat a metody, které třídy mají. To je až v okamžiku implementace, což je pro náš cíl příliš pozdě.

2.6 Pracnost vývoje

Pro každé plánování a řízení prací na informačních systémech je nutné odhadovat náklady, jenž bude potřeba vynaložit. Odhad je možné provádět v okamžiku, kdy víme co budeme vyvíjet. Tento okamžik lze nalézt mezi *Incepční* (překládáme „Založení projektu“) a *Elaborační* (překládáme „Rozpracování požadavků na software“) fází životního cyklu softwarového produktu, pokud postupujeme podle metody Unified Process viz [7]. Ikdyž, jak víme z předchozích kapitol, neobejde se to bez řady problémů a nepřesností.

Problémem odhadu složitosti je v tom, že je jej nezbytné neustále korigovat. Korekce probíhá v závislosti na známých datech, vyplývajících z probíhajícího vývoje. To samozřejmě značně ovlivňuje jeho přesnost. Jsou-li na začátku vývoje, ve fázi specifikace, odhadnuty náklady nereálné a jsou-li skutečné mnohonásobně vyšší, vede tento druh odhadu nákladů k celkem oprávněné kritice. Ta kritizuje jeho počáteční nepřesnost a téměř kontinuální upravování, na základě vznikajících částí informačního systému. Odhad pokulhává ruku v ruce s vývojem a je jen málo platný. Je pak zcela pochopitelené, že se často ceny softwarových produktů realizovaných formou softwarových projeků neodvíjí od skutečného odhadu na základě funkční sepcifikace, ale že si firmy informační systémy objednávají za fixní cenu. Záleží pak na dodavateli, má-li odvahu za tuto cenu požadovanou funkčnost realizovat.

I přes řadu nevýhod tohoto postupu odhadu nákladů je jistě smysluplné se jím zabývat. Zkoumání všech dostupných řešitelských podkladů (funkčních specifikací z fází projektu a zdrojového kódu) vede k odhadu složitosti produktu. Vaníček viz [3] uvádí “*Zůstává otázka, jaký je vztah mezi složitostí a náklady, respektive složitostí, počtem řešitelů a časem, který je na realizaci projektu potřeba.”*

Na první pohled by se zdálo, že vynásobíme-li počet řádků zdrojového kódu nějakou konstantou, která bude zohledňovat typ použitého programovacího jazyka a faktory ovlivňující vývoj, bude vztah lineární.

To je omyl. Vztah mezi rozsahem software a náklady na jeho vývoj zdaleka lineární není. „*V dobách, kdy neexistovala žádná pravidla pro tvorbu software (v padesátých letech) se zdálo, že je tento vztah kvadratický*“ viz [3].

Viz [3] v sedmdesátých letech dospěl Brooks na základě experimentálních dat k empirickému závěru, že u velkých programových celků lze závislost pracnosti na objemu programu aproximovat funkcí:

$$E = m \cdot V^{3/2}$$

Kde:

- E - je pracnost měřená například v člověkoměsících.
- V - je objem programu, měřený v modifikovaných tisících řádcích zdrojového kódu a m je vhodná konstanta viz [3,39].

I tento odhad lze v dnešní době považovat za překonaný. Lze tvrdit, že v současné době, díky moderním programovacím jazykům a novým pravidlům vývoje software, již není tak strmý. Viz [3] „*Nic méně, pro imperativní programování se zřejmě nikdy nepodaří, aby tato závislost byla lineární. To plyne z podstaty věci. Při tvorbě rozsáhlého systému je třeba neustále hlídat více souvislostí a vazeb než u systému malého rozsahu a tuto práci je třeba vynakládat v každém kroku. Počet těchto kroků sice roste s objemem lienárně, každý jednotlivý krok je však náročnější. Celková pracnost roste tedy nutně strměji než přímá úměra.“*

Z Brooksova odhadu však plyne násedující viz [39]:
Brooks při svém odhadu zahrnul do pracnosti E pracnost potřebnou pro:

- Vypracování specifikace.
- Realizaci specifikace (návrh a kódování).
- Její testování a ověřování až do okažiku předání uživatelům, včetně uživatelské a řešitelské dokumentace.

Nezahrnul do ní:

- Náklady na údržbu (opravy chyb, zdokonalování produktu a jeho realizaci v prostředí).

- Náklady na školení uživatelů.

Velmi podobnou zásadou se řídí i všechny další odhady pracnosti. Vývoj je zahrnován vždy úplně, včetně dokumentace. Údržba, rozšiřování funkcí a školení uživatelů se musí zahrnout zvlášť, dle konkrétní situace.

2.6.1 Pracnost a dělba práce

Již víme, že pracnost roste v závislosti na objemu produktu rychleji než lineárně. Viz [3] „*Ve vzorcích odhadů COCOMO se to projevilo tak, že hodnota konstanty* ***b*** *byla větší než 1 u všech módů, respektive všech tříd produktů. Nabízí se tedy možnost snížit celkovou pracnost tím, že práci rozdělíme na části řešené samostatně, které neboudou tak rozsáhlé. Platí totiž, že je li* ***b > 1****, je pro kladná* ***x*** *a* ***y*** *vždy:*"

$$(x+y)^b > x^b + y^b$$

Krom výše uvedeného, je nereálné aby jeden řešitel řešil projekt s vysokou celkovou pracností. Doba na realizaci by byla neúnosně dlouhá.

Dělba práce má ale svá uskalí. Mezi řešiteli v týmu je řeba vymezit určité rozhraní, pomocí něhož budou jimi realizované subsystémy vzájemě komunikovat. Čím je soudržnost celku vyšší, tím musí být složitější rozhraní. Toto rozhraní je potřeba nejen vymezit, ale také kontrolovat a po odděleném odzkoušení obou systémů provést kompletaci a znovu ověřit funkčnost celku. To však představuje doplňkovou práci. Při tom se jedná o práci velmi náročnou na kvalifikaci. Účelná dekompozice systému, vymezení jasných rozhraní vyžaduje značné znalosti a zkušenosti. Dekomponovaný systém přináší nutnost vzájemné komunikace řešitelů subsystému. Ta stojí opět čas a

zvyšuje celkovou pracnost. Viz [3] "*Souhrn těchto doplňkových prací, které si dekompozice vyžádá, je možno chápat jako pracnost programování ve velkém.*"

Dekompozice tudíž přináší dvě úskalí. Na jedné straně je dělba práce nutná pro zkrácení celkového termínu dokončení vývoje a příznivě se odrazí v celkové pracnosti. Na druhé straně však pracnost zvýší, protože je nutné definovat komunikační rozhraní mezi jednotlivými subsystémy.

2.7 Vliv napjatých termínů

Je-li projekt rozdělen na vhodný počet subsystémů, jenž realizují různí řešitelé, dospějeme k hodnotě, která je z hlediska celkových nákladů optimální. Problém nastává, pokusíme-li se počet řešitelů zvýšit. Děláme to zpravidla ve snaze zkrátit dobu vývoje. To se projeví na nárůstu nákladů. Tento nárůst je z počátku nepatrný, ale záhy se stává citelným a od určitého počtu pracovníků je již strmý tak, že další posilovájní týmu se nevyplácí. Dokonce může vést i k vzrůstu doby realizace.

F.P. Brooks na základě své zkušenosti s řešením rozsáhlých softwarových systémů v IBM uvádí že: „*je-li vývoj v časovém skluzu, pak posílení řešitelského týmu o další členy je nejjistější cestou, jak zpoždění ještě zvýšit* " viz [39].

Je tedy nutné při odhadu složitosti softwarových produktů vést v patrnosti i to, že počet (respektive měnící se počet řešitelů během projektu) řešitelů se výrazně projevuje na době realizace. Příslušný model odhadu složitosti musí tuto skutečnost nějak zohlednit.

2.8 Unified Process and UML

V této kapitole se zaměřím na první část metody vývoje softwaru navrženou firmou Rational Software – Unified Process (zkráceně UP). Někteří autoři metodu Unified Process nazývají „Rational Unified Process“ a zkracují ji na RUP. „R“ pak symbolizuje jméno firmy Rational.

UP viz [7] je ucelený návod jak postupovat při vývoji softwaru od fáze sběru uživatelských požadavků přes fáze vývoje až k fázi nasazení do provozu. Firma Rational Software pro účel metody navrhla také modelovací jazyk Unified Model Language (zkracujeme UML). Ten slouží k modelování software. Snahou firmy Rational Software je aby se stal univerzálním modelovacím nástrojem, který by byl využíván během všech, dle UP čtyřech, fází tvorby softwaru. Za jeho pomoci je možné (většinou formou grafů) popisovat požadavky kladené na vznikající software. Na jeho strukturu, architekturu, vzhled, jakost atd. Pouhý UML by však byl pouze množstvím grafických modelů. UP dává uživateli návod, jak postupně jednotlivé grafické modely používat, aby na konci jeho práce byl skutečný softwarový produkt.

UP není metodou odhadu složitosti software. Uvádím jej proto, že se jeho použití stalo velmi rozšířené v praxi. Byť není standardizován žádnou oficiální autoritou, je za neoficiální standard považován. Mnoho softwarových firem využívá UP pro řízení projektů. Je pak samozřejmostí, že se firma Rational Software snaží nalézt nějakou metodu odhadu složitosti. Ta, jak jsem již uvedl, jednoznačně chybí.

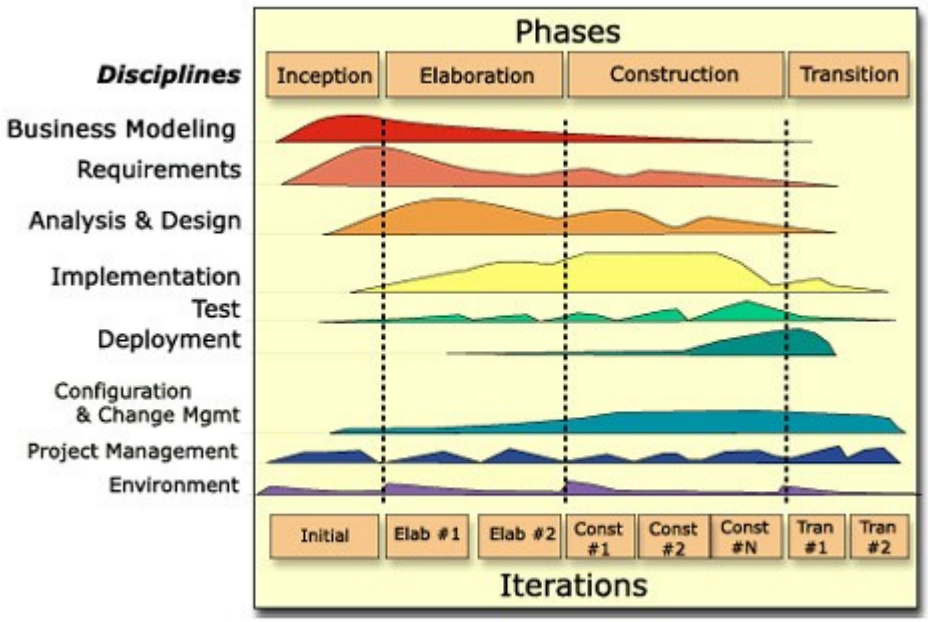
Protože UP lze v dnešní době považovat za standard, grafické modely s využitím UML také, je vhodné se pokusit je nějak využít pro účel odhadu složitosti, respektive pro návrh odhadu složitosti, který má být příspěvkem této práce, orientovat na metodu UP.

2.8.1 Vznik metody UP (Unified Process) a její hodnocení

Dle [1] *"Metoda Unified Process (dále již jen zkratka UP) je založena na metodách Ericsson (Ericsson approach, 1967)"*viz [18]*, Rational (Rational Objectory Process, 1996-1997) - a na dalších zdrojích"*. Na jejím vzniku se podíleli Ivar Jacobson viz [4,5,7,22], G. Booch viz [4,5,7,20] a řada dalších autorů. Metoda je souhrn doporučených postupů jak tvořit software.

Unified Process rozděluje životní cyklus softwarového projektu na čtyři fáze (etapy) viz. Obr 2.8.1:

- **Založení** (Inception).
- **Rozpracování požadavků na software** (Elaboration).
- **Budování** (vývoj) **software** (Construction).
- **Zavedení software do použití** (Transition).



Obr 2.8.1. Čtyři hlavní fáze životního cyklu projektu dle IBM - RUP

Zdůrazňuje se prolínání jednotlivých pracovních procesů - etap (v RUP - Processes) napříč fázemi. Obchodní modelování (na obr 2.8.1 - Business Modeling) prostupuje všemi fázemi. Největší úsilí je mu ale věnováno na začátku. Stejně tak je to s tvorbou seznamu uživatelských požadavků (na obr 2.8.1 – Requirements).

Protože odhad složitosti softwartového produktu má být proveden co nejdříve, tedy ve chvíli, kdy jsou známy požadavky na funkčnost softwaru (to je na konci procesu specifikace požadavků), a dle UP má být funkční specifikace hotová v druhé fázi - Rozpracování požadavků na software (v Elaborační fázi) životního cyklu softwarového produktu, provádím jej na konci této fáze. Funkčí specifikace na softwarový produkt se řídí doporučením definovaným v metodě UP viz [7]. Vziká v etapě nazývané Requirements – seznam požadavků. Tuto etapu navrhuji považovat za **etapu specifikace požadavků**, kterou jsem ji již vymezil na zčátku kapitoly 2. Postup, jak uživatelské požadavky na software sbírat v etapě specifikace požadavků, uvedený v metodě UP využívám pro mnou navrženou metodu odhadu složitosti softwarového produktu.

Součástí metody UP je modelování chování systému pomocí standardního modelovacího jazyka UML viz [1,4,5,6,].

Dle [1] "*Metodologie UP stojí na třech základech, kterým je potřeba věnovat pozornost*:

- *Architektura.*
- *Případ (případy) užití.*
- *Iterace a přírůstek.*

2.8.1.1 Architektura

Příručka *The Unified Modeling Language Reference Manual* viz [5] definuje architekturu systému jako viz [5] „organizační strukturu systému. UML umožňuje čtyři základní pohledy na systém:

- Logický pohled.
- Pohled procesů.
- Pohled implementace.
- Pohled nasazení.

A obecný pátý pohled, ve kterém jsou tyto čtyři pohledy integrovány. Nazýváme jej **pohled případů užití**.

O **Pohledu případu užití** uvádí viz [5] „*Všechny jiné pohledy jsou odvozeny od pohledu případů užití. Tento pohled zachycuje základní požadavky kladené na systém jako na množinu případu užití a utváří tak základ tvorby všech dalších pohledů*“.

Pohled případu užití (zkracujeme na případ užití – anglicky Use Case) umožňuje popsat požadavky uživatele na systém. Využívá se ve etapě specifikace požadavků.

2.8.1.2 Případy užití a riziko:

Dle [1] "*Případy užití jsou způsobem, jak zachytit požadavky. Docela přesně bychom mohli tvrdit, že metoda UP je řízena požadavky*[11]. Tento rys je plně v souladu o vymezení pojmu jakost (kvalita) viz [2,12] jako stupně splnění požadavků. Je tedy přirozené využít pro odhad pracnosti postupy úspěšně prověřené pro měření jakosti softwarových produktů. Případ užití (UseCase) pak zachycuje logicky ucelenou skupinu požadavků uživatele na chování systému. Je psán z pohledu uživatele. Je základním nástrojem používaným ve etapě specifikace požadavků. V jazyce UML je případ užití modelován pomocí diagramu užití viz [1,4,5,6,7,8]. O diagramu užití hovořím v další kapitole UML.

2.8.1.3 Iterace a přírůstek:

Dle [7] "*Metoda UP je založena na iterativním a přírůstkovém procesu"*. Základní myšlenka spočívá v rozložení velkého softwarového projektu na řadu „miniprojektů". Každý „miniprojekt" má svůj jasně definovaný začátek a předpokládaný konec. Miniprojekt má vždy čtyři fáze, o kterých jsem hovořil v kapitole 2.8.1

Cílem je vytvoření procesu přeměňujícího vstupy ve výstupy.

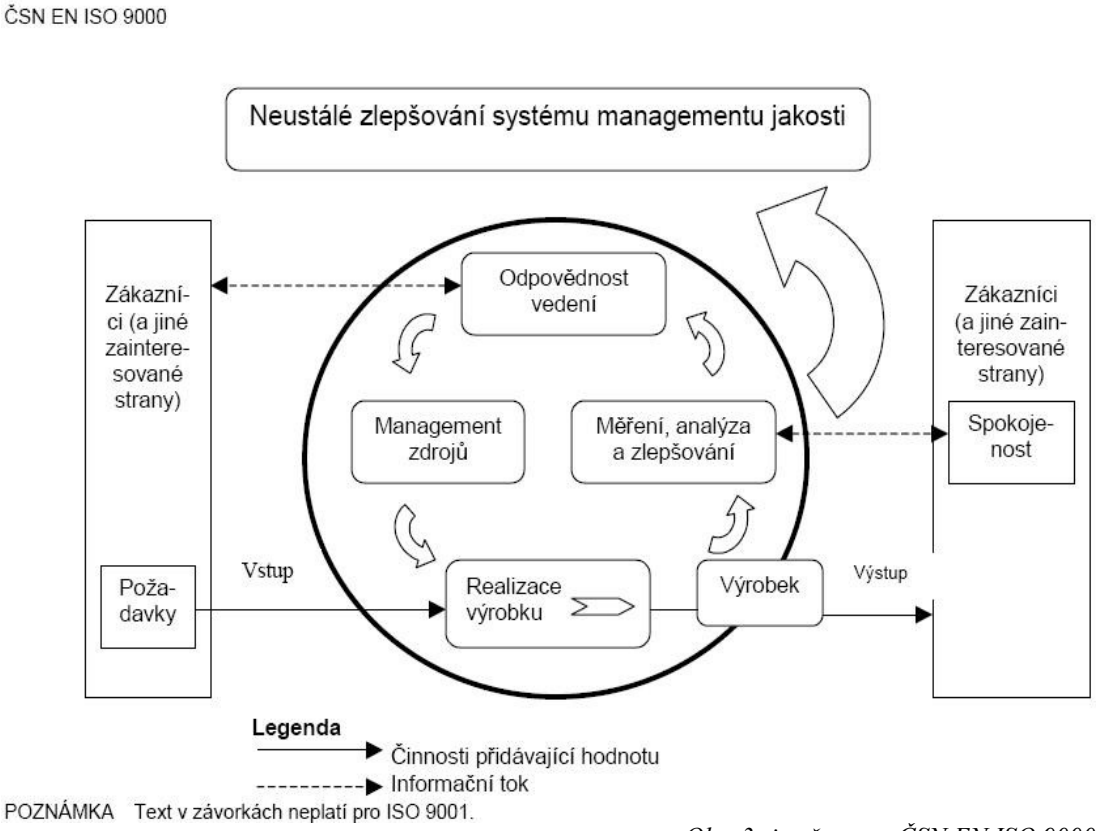
Pokud na konci „miniprojektu" nebylo dosaženo požadovaných cílů a nebo se nalezly nějaké další skutečnosti, jenž realizaci „miniprojektu" nějak komplikují, proběhne celý miniprojekt znovu. Samozřejmě zase prochází všemi definovanými fázemi jako je Založení (Incepční fáze), Rozpracování požadavků na systém (Elaborační fáze) atd.

Každá iterace generuje zvolený produkt (jenž je téměř vždy určitou částí finálního systému) a příslušnou projektovou dokumentaci. Rozdíl mezi výsledky jednotlivých iterací je označován za přírůstek. V cyklu pak vylepšujeme produkt, dokud nedosáhneme finálního „cíle" viz [2].

---

11 Viz [2] „je ***požadavek*** *potřeba nebo očekávání, které jsou stanoveny,obecně se předpokládají nebo jsou závazné"*

Myšlenka iterativního přístupu není nová. Iterativní přístup je například použit při neustálém zlepšování modelu jakosti v normě ČSN EN ISO 9000. Viz [2] Obr 3.



Obr. 3: je převzat z ČSN EN ISO 9000

Norma ČSN EN ISO 9000 definuje projekt: viz [2] „*Jedinečný proces sestávající se z řady koordinovaných a řízených činností s daty zahájení a ukončení, prováděný pro dosažení cíle, který vyhovuje specifickým požadavkům, včetně omezení daných časem, náklady a zdroji*“.

Viz [2] „*POZNÁMKA 1 - Jednotlivý projekt může být součástí struktury většího projektu*“. Tato poznámka koresponduje s myšlenkou rozdělení projektu do „miniprojektů či podprojektů“.

Viz [2] „*POZNÁMKA 2 - U některých projektů jsou cíle propracovány a znaky produktu jsou definovány současně s postupem prací na projektu*“. V první fázi metody UP není přesně jasné, jak bude projekt rozsáhlý a jaké specifické požadavky

jím budou realizovány. Dochází k jeho upřesňování v dalších fázích.

Viz [2] „*POZNÁMKA 3 - Výstupem projektu může být jeden produkt nebo několik jednotek produktu "*. Zde se UP shoduje. Výstupem projektu řízeného podle UP může být jeden nebo více produktů.

2.8.2 Vznik UML a jeho zhodnocení

Rozšíření počítačů vede k stále propracovanějším přístupům k problematice tvorby softwaru. Dle [1]: *„Do roku 1994 ve světě existovalo několik soupeřících jazyků pro modelování a také několik metodik“* viz [9,11,19,20,21,22]. *„Ačkoliv s sebou každá metodika přinesla něco nového“* viz [18] *„– a obvykle též novou notaci – rozmanité inovace jen výjimečně znamenaly i novou kvalitu“*. Více jak polovinu „tehdejšího trhu“ si mezi sebe rozdělily metody Booch viz [20] a OMT (Object Model Technique – Rumbaugh) viz [21].

Podle [1]: „*Na straně metodik si jasně nejlépe vedla metodika Objectory“* viz [22]. Pokus o sjednocení metodik a jejich postupů vedl v roce 1994 ke vzniku metodiky Fusion viz [19] jejímž autorem byl Derek Coleman. Do její přípravy však nebyli zapojeni autoři Rumbaugh, Booch, Jacobson. Metoda Fusion skončila v okamžiku, kdy se Rumbaugh, Booch a Ivar Jacobson spojili ve firmě Rational Corporation (dnes Rational Software), která pracovala na tvorbě jazyka UML (Unified Model Language) viz [4,5,6,7,8].

V roce 1997 sdružení OMG (Object Management Group) viz [17] jazyk UML přijalo a vznikl první průmyslový standard pro vizuální modelování. Viz [1]: „*Jazyk UML spojuje mnoho nejlepších myšlenek přejatých z „prehistorických“ metod“*. Domnívám se, že jde o velmi dobře použitelný modelovací jazyk. Přesto je UML podroben kritice viz [6].

V současné době je k dispozici verze UML 2.0 která je publikovaná sdružením Object Management Group viz. [17], jejímž jsem členem. Zde je k dispozici podrobný popis otevřeného standardu UML a řady dalších. V případě UML 2.0 jde o rozšíření původního UML. I ten je podroben kritice. Podle [8]: „*UML není rozhodně nástrojem, který laik za rozumně krátkou dobu (třeba během 15 minut na začátku schůzky s analytiky) pochopí tak, že je schopen číst a rozumět diagramům“*.

V knize The *Unified Modeling Language User Guide (průvodce uživatele jazykem UML, Booch)* viz [4] Booch uvádí jak je jazyk UML sestaven z pouhých tří stavebních bloků:

- Z předmětů (things).

- Vztahů (relationships).

- Diagramů (diagrams) - což jsou pohledy na modely. Viz [1] *„Vyprávějí příběh" o softwarovém systému a jsou naším způsobem vizualizace toho, „co" systém bude dělat (analytické diagramy, analysis-level diagrams), a toho, „jak" to bude dělat (návrhové diagramy, design-level diagrams)".*

V UML je k modelování definována celá řada diagramů. S jejich pomocí je možné modelovat vznikající software ve všech fázích jeho životního cyklu. Odhad složitosti je zapotřebí provádět pokud možno co nejdříve - ve etapě specifikace požadavků na vznikající software. Etapa je ukončena v druhé fázi Unified Processu – Rozpracování požadavků na software. V této etapě se využívá především (je samozřejmě možné využít i jiných diagramů) případů užití, které je možné graficky zobrazit pomocí diagramů užití.

- **Diagramy užití** - Viz [1] „*Diagramy užití James Rumbaugh definuje jako „specifikaci posloupností činností, včetně proměnných posloupností a chybových posloupností, které systém, podsystém nebo třída může vykonat prostřednictvím interakce s vnějšími (externími) účastníky".*

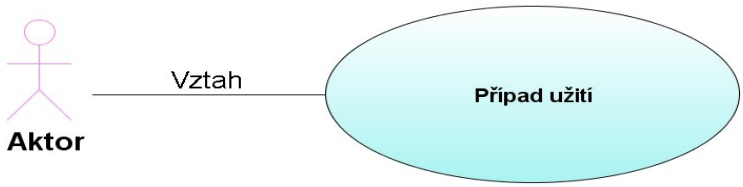
2.8.2.1 Detailní popis diagramu užití

Diagram užití slouží k grafickému modelování případů užití. Zatím co případ užití je slovní popis popisující chování uživatele pracujícího se systémem, samotný diagram musí nějak graficky zohlednit vztah uživatele (aktora) a jeho požadavků (případ užití) k systému.

Za tímto účelem jsou pro diagram užití definovány základní tři symboly viz. Obr.4:

- Uživatel – aktor (piktogram znázorňující panáčka) může jím být:
  - člověk,
  - nebo autorizovaný proces prováděný programem:
    - člověk i „proces“ je ještě popsán jeho chováním a vztahem k případu užití. Toto chování nazýváme rolí aktora. Aktor „čtenář sestavy“ se bude v systému chovat jinak než aktor „administrátor informačního systému“. Aktor „relační databáze“ bude mít jistě naprosto rozdílnou roli od ostatních, „živých“ aktorů.

- Případ užití znázorňovaný elipsou.
- Vztah (většinou neorinetovaná hrana spojující uživatele s případem užití).



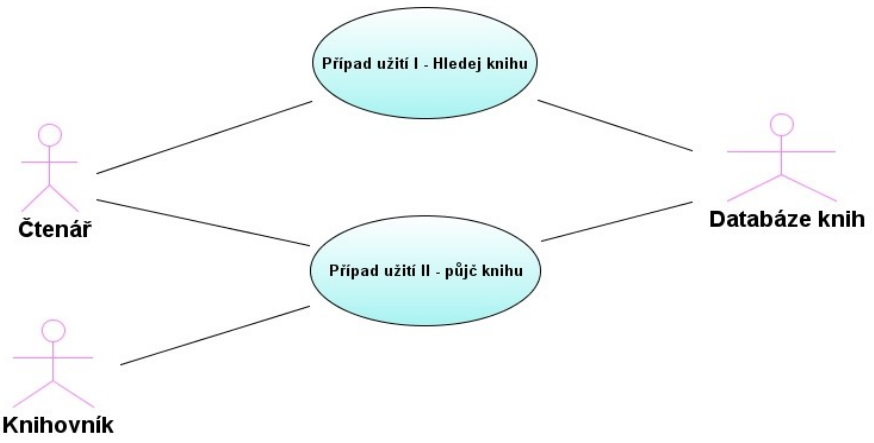
Obr. 4: Diagram užití

Poznámka: Notace UML umožňuje upřesnit „vztah“ na několik dalších typů (Extend[12], Generalization, Association, Include a jiné). Mimo to je možné diagram

12 Uvádím názvy anglicky, protože překlad do češtiny buď chybí, nebo není výstižný

užití rozšířit o řadu „upřesňujících“ prvků (Package a jiné). Z mé zkušenosti je ale jejich využití kontraproduktivní. Diagram užití má jednoduchou formou popsat podstatné požadavky na systém a jeho vztahy k uživatelům. Ostatní upřesnění často ve etapě specifikace požadavků chybí. Není tedy vhodné „za každou cenu“ upřesňovat typy vztahů a rozlišovat typy případů užití.

Jak je z Obr. 4 patrné, lze z diagramu užití na první pohled poznat, kolik uživatelů pracuje s jedním případem užití. V diagramu užití však může být větší množství případů užití. I vazby mezi uživateli a případy užití mohou být složitější viz. Obr. 5: Složitější diagram užití.



Obr. 5: Složitější diagram užití

V tomto případě máme v diagramu užití dva případy užití:

- Hledej knihu.
- Půjč knihu.

a tři aktory s různými rolemi:

- Čtenář.
- Knihovník.
- Databáze knih.

Máme zde i počet vazeb:

- Čtenář má dvě vazby.
- Knihovník jednu.
- Databáze knih dvě.

Z výše uvedeného popisu je zřejmé, že diagram užití v sobě nese objektivně zjistitelné prvky (body), které mohou sloužit k odhadu složitosti.

Jedná se především o:

- Počet případů užití v diagramu užití.
- Počet uživatelů (aktorů) případu užití.
- Role uživatelů (aktorů) případu užití.
- Vazby mezi aktory a jednotlivými případy užití.

2.8.2.2 Ostatní UML diagramy

Ostatní UML diagramy jsou použity v dalších fázích, respektive v jejich etapách, UP. Pro odhad složitosti je nevyužívám. Proto se nebudu zabývat jejich podrobným popisem.

Jsou to:

- Diagramy tříd.
- Diagramy aktivit.
- Stavové diagramy.
- Sekvenční diagramy.
- Diagramy spolupráce

Jejich seznam zde uvádím pro ucelení kapitoly.

Poznámka: Jak jsem již uvedl v kapitole 2.8.1. Unified Process nelze brát dogmaticky. Etapy fází životního cyklu softwaru v UP se mohou, a z mé zkušenosti plyne že je to účelné, prolínat. Etapy fáze Založení projektu se prolínají do fáze Rozpracování požadavků na software atd. Proto ve fázi Rozpracování požadavků na software je možné využití i jiných diagramů, např. Diagramů tříd.

2.8.3 Kritické zhodnocení Unified Process

Unified Process není v rozporu s normou ČSN EN ISO 9000 viz [2]. U Unified Processu postrádám metodu odhadu složitosti v počátečních fázích projektu, to je v Incepční a Elaborační fázi projektu viz [7] . Odhad složitosti, pomocí kterého by mohl vedoucí projektu odhadovat potřebný čas a náklady na realizaci požadovaných cílů projektu, jednoznačně chybí. Ikdyž existuje metoda UCP viz [24], domnívám se, že není použitelná. Jak jsem ukázal v kapitole 2.3., z pohledu teorie měření ji není možné teoreticky důvodnit.

Složitost, rozumějme míra složitosti, je známa až na konci fáze Budování software (dle Unified Process). To je v okamžiku, kdy je většinou software už naprogramovaný a na případné změny je již pozdě.

Proto navrhuji provádět odhad složitosti ve fázi Rozpracování požadavků na software na základě měřitelných prvků v diagramu užití. Diagram užití, jak jsem uvedl, je používán ke grafickému zobrazení případů užití. Ty jsou dle metody UP využívány právě ve fázi Rozpracování požadavků na software, v etapě Requirements, kterou jsem navrhl považovat za **etapu specifikace požadavků**. Metoda odhadu složitosti na základě diagramů užití je pak platná pro metodu UP a všechny její klony, jež zachovávají diagramy užití tak, jak je popsáno v předchozích kapitolách.

2.8.3.1 Dovětek k UP

Metoda Unified Process dostala také svou komerční podobu ve formě Rational Unified Process. Zatím co Unified Process je otevřenou metodou, Rational Unified Process je produktem[13] firmy Rational Software, která je dnes součástí firmy IBM.

13 Viz [2] "*produkt je výsledek procesu*
POZNÁMKA 1 Existují čtyři generické kategorie výrobků (produktů), a sice:
− služby (např. přeprava);
− software (např. počítačový program, slovník);
− hardware (např. mechanická část motoru);
− zpracované materiály (např. mazivo)."

Rational Unified Process (zkráceně RUP) je rozšířením UP. Viz [1] „*Tento produkt nabízí nástroje, které nejsou obsaženy v metodě UP“.* Její součástí je i souhrn dokumentace k metodě RUP. Firma Rational Software vyvinula pro RUP dnes již velmi známý CASE nástroj Rational Rose.

Ani RUP není metodou, která by neměla slabá místa. RUP je metoda zaměřená především na projekty vznikající „na zelené louce“. Pro projekty, jejichž výsledkem má být změna existujícího softwarového produktu, není dle kritiků vhodná. Mimo to se lze setkat s názorem, že nedostatečně modeluje obchodní procesy (tak zvané Business modelování).

S námitkou nedostatečného modelování obchodních procesů se neztotožňuji. Domnívám se, že je možné obchodní procesy modelovat pomocí diagramů užití. Přesto modelování obchodních procesů a zaměření „RUPu“ na projekty vznikající „na zelené louce“ vede některé autory k tvorbě nových metod.

Proto je možné setkat se víceméně s dvěma typy nových metod:

- Prvním typem jsou metody rozšiřující UP:
  - Viz [16] „*Enterprise Unified Process*“.
- Druhým typem jsou pak metody využívající standardu UML (UML často rozšiřují), ale nevychází z UP:
  - Viz [10] „*TOGAF (The Open Group Architectural Framework)“* ,
  - Viz [23] „*BORM (Business and Object Relations Modeling)“.*

2.9 Úvod do umělé inteligence a její praktické využití

Tato kapitola se zabývá klasifikací a základním popisem neuronových sítí. Uvádím zde nezbytné základní informace potřebné pro konstrukci a aplikaci neuronové sítě, pro odhad složitosti vývoje softwaru. Tuto problematiku jsem studoval a často konzultoval s Doc. Arnoštem Veselým viz [30]. Na základě získaných znalostí jsem navrhl nástroj určený pro odhad složitosti softwarového produktu - PETRA (PErcepTRon Artificial intelligence).

Nástroj jsem naprogramoval v programovacím jazyce Java viz [40]. Jeho popisu se podrobněji věnuji v stejnojmenné kapitole - PETRA.

2.9.1 Vznik umělého neuronu

Pojmem neuron označujeme nervovou buňku živých organismů.

Neuronová buňka se skládá z:

- **Těla** - Soma.
- Z těla vybíhá velké množství kratších výběžků – **Dendritů** (Dendrites).
- Z jednoho dlouhého výběžku, který nazýváme **Axon** (Axon).

U složitějších organismů jako jsou savci, ptáci atd. Se bohatě rozvinula tzv. nervová soustava. Nervová soustava je tvořena obrovským množstvím vzájemně propojených neuronových buněk. Například Novák a kolektiv viz [26] uvádí, že lidský mozek je složen asi z 10^{10} neuronů. Jednotlivé neurony jsou mezi s sebou propojeny ještě o řád vyšším počtem vazeb!

Dokonalost neuronových sítí u živých organismů vedla v 50 letech minulého století ke snaze je napodobit. Základem neuronových sítí se stal neuron, který byl hardwarovou a později softwarovou zjednodušeninou neuronu živých organismů. Studium živých organismů přineslo celou řadu topologií umělých neuronových sítí.

2.9.2.1 Matematický model neuronu

Každý neuron přijímá z okolí kladné nebo záporné podměty. Když souhrn vnějších podmětů překročí prahovou hodnotu neuronu, neuron se aktivuje. Viz [29] "*Výstupní hodnota neuronu je pak nějakou nelineární transformací souhrnu podmětů"*. Z tohoto pohledu vycházejí matematické modely neuronů.

Jako prvním byl McCullochův a Pittsův "logický neuron". Pracoval s binárními vstupními a výstupními hodnotami 0 a 1.

Vhodnějším modelem neuronu než byl McCulochův a Pittsův „logický neuron" byl neuron, který má na svém vstupu (vstup - podmět nazýváme někdy Excitačním vstupem) jakékoliv reálné číslo v intervalu od <-1, 1>. Výstupní hodnotou neuronu pak mohlo být opět nějaké reálné číslo, ze stejného intervalu, nebo pouze binární výstup 0 nebo 1.

Veselý viz [30] ve svých pracích symoblicky zobrazuje model neuronu jako jakýsi objekt, jenž má n vstupů $\boldsymbol{x}(t)$ $(x_1(t), \ldots, x_n(t))$ a jeden výstup $y(t)$. Každý vstup je představován vstupní hranou $\boldsymbol{x}(t)$ a má ještě váhu $w_i(t)$.

Každý vstup $\boldsymbol{x}(t)$ je násoben váhou hrany $\boldsymbol{w}(t)$. Suma všech vstupních hodnot $h(t)$ se nazývá „ *postsynapticka excitace"*.

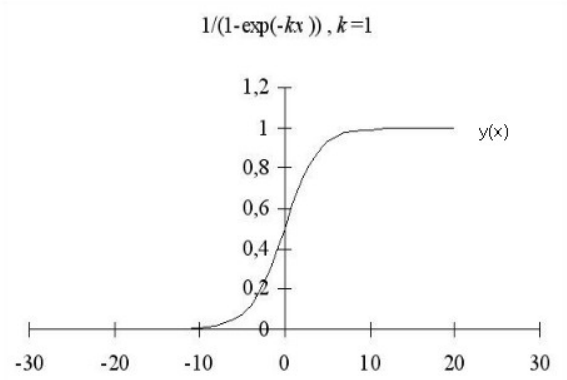
$$h(t) = \sum_{j=1}^{n} w_j(t) \cdot x_j(t)$$

Výstupní funkcí neuronu $y(t)$ je pak:

$$y(t) = f(h(t))$$

Tuto funkci nazýváme přenosovou funkcí. Dle typu přenosových funkcí f je pak neuron nazýván. Nejčastější přenosová funkce je sigmoidální:

$$f(x) = \frac{1}{1+\exp^{(-kx)}} \quad , \quad f'(x) = k f(x)\,(1 - f(x))$$



Obr. 6: Průběh sigmiodální funkce, Veselý viz [30]

Neurony se sigmoidální přenosovou funkcí využívám při konstrukci neuronové sítě PETRA.

2.9.3 Neuronové sítě

Teprve spojením neuronů do neuronových sítí vzniká stroj, schopný řešit složité úlohy. Může umět rozpoznávat předložené vzory, umí si je zapamatovat - je prostě částečně „inteligentní“.

První neuronová síť vznikla v roce 1957. Jednalo se o Rosenblattův Perceptron. „*Rosenblatt jej navrhl jako model zrakové soustavy*.“ viz [29,31].

Percepron je třívrstvá síť. První (vstupní) vrstvu nazývá Rosenblatt sítnice. Slouží pro přijímání vstupů z prostředí. Vstupní vrstvou jsou vstupní neurony nazývané receptory. Viz [29] „*Hodnoty na vstupu nabývaly hodnot 0 nebo 1 podle toho, zda byly excitovány:*

1- ano, 0 – ne.

Výstupy receptorů byly přivedeny na asociativní elementy. Asociativní element připomínal adaptivní lineární neuron s tím, že všechny váhy w_i *měly pevné hodnoty +1 nebo -1. Asociativní element se aktivoval (vydal hodnotu 1), pokud souhrn jeho vstupů překročil zadaný práh. Počet asociativních elementů byl řádově desítky tisíc. Vstupy z asociativních elementů byly náhodně zvolenými vazbami propojeny na reagující elementy, jejichž počet odpovídal počtu tříd, do kterých byly předložené vzory klasifikovány* .“

Poslední vrstva sítě je blok, určený pro výběr maxima. Ten vybere pro daný obraz ten reagující element, který má nejvyšší výstup a který odpovídá třídě, do níž je obraz zařazen.

Rozenblatt zamýšlel percepron jako model lidského mozku. Mozek je však mnohem složitější. **Myšlenkou Perceptronu jsem byl inspirován a využil jsem ji při konstrukci neuronové sítě PETRA**.

V roce 1969 se objevila kritika Minského a Paperta viz [29]. Ti kritizovali klasifikační schopnosti neuronu. Dokázali, že jeden neuron není schopen realizovat jednoduchou logickou funkci, tzv. vylučovací disjunkci (XOR). Toto tvrzení je samozřejmě opávněné. Platí však pouze pro jeden neuron. Jednoduchá síť složená ze tří perceptronů již XOR realizovat dovede. V tehdejší době však nebyl znám vhodný algoritmus pro učení vícevrstvých sítí. Díky absenci učebního algoritmu a kritice Minského a Paperta se vývoj neuronových sítí zastavil na zhruba dvacet let.

Nezastavil se však úplně. Autoři jako Hopfield, Nielsen a Kohonen viz [29,30] pracovali na neuronových sítích dál. Vydali se však jinou cestou. Volili jinou architekturu a topologii sítí, než je vícevrstvý perceptron.

Dokud však nebyl znám vhodný algoritmus učení neuronových sítí, jejich vývoj fakticky stagnoval.

O průlom tohoto trendu se zasloužil v roce 1985 tým pracující na MIT (Massachusetts Institute of Technology). Podařilo se mu najít algoritmus učení neuronových sítí nazvaný error backpropagation (zkracujeme na back propagation) – učení pomocí zpětného šíření chyby. Od tohoto okamžiku se neuronové sítě opět začaly rozvíjet.

Metodu učení neuronové sítě pomocí zpětného šíření chyby využívám pro učení mnou setrojené neuronové sítě PETRA.

V současné době se neuronové sítě již používají pro řešení různých teoretických i praktických úloh.

Jsou to především úlohy:

- Pro rozpoznávání vzorů.
- Filtrování a čištění zašuměných signálů.
- Predikování hodnot dle neúplného či zašuměného zadání.

Je zřejmé, že pro každou úlohu je vhodné volit jiný typ sítě. Typ sítě se liší

architektonickým návrhem (jaká je přenosová funkce neuronu, jak jsou neurony mezi s sebou propojeny, kolik je vstupních neuronů, kolik skrytých vrstev atd.) a způsobem uspořádání neuronů v síti – tak zvanou topologií.

2.9.3.1 Učení neuronových sítí

Základním problémem neuronových sítí je však způsob jejich adaptace na úlohu, kterou s jejich pomocí chceme řešit. Chceme-li například predikovat trend růstu ceny určitého zboží v závislosti na růstu cen několika dalších kokurenčních výrobků, je neuronová síť typu perceptron s jednou skrytou vrstvou vhodným nástrojem. Problém spočívá v tom, jak neuronovou síť „donutit“ trend predikovat. Adaptovat ji na zvolenou úlohu.

Způsobu adaptace se říká učení neuronové sítě. Lze si je představit obdobně, jako když se sami učíme slovíčka cizího jazyka. Učíme se je tak dlouho, až jsme schopni téměř každé slovíčko použít a znát jeho význam. Setkáme-li se s podobným slovíčkem či slovní složeninou, jsme schopni predikovat význam, ačkoliv jsme se s ním ještě nikdy nesetkali. Obdobně se neuronové sítě učí z předložených vzorů (např. časových řad) rozpoznat trend. V případě, že neuronovou síť můžeme kontrolovat – předkládáme učební soubor a známe odpověď, ke které se má odpověď sítě blížit, hovoříme o učení s učitelem.

Nemáme-li soubor požadovaných odpovědí a síť se učí – adaptuje, na základě jen vstupních parametrů, nazýváme tento proces učení bez učitele. Jde například o Hopfieldovu neuronovu síť viz [29].

Algoritmy učení neuronových sítí jsou složité a mají mnoho úskalí. Teprve metoda učení vícevrstvé perceprtonové neuronové sítě pomocí zpětného šíření chyby, dosáhla slušných výsledků. Tento algoritmus učení pracuje na základě zpětného ovlivňování vah neuronů v síti. Používá se na to gradientní metoda a postup je zjednodušeně následující viz [29,30]:

- Spočítá se rozdíl (chyba) mezi požadovaným a vypočteným výstupem sítě.

- Po té se šíří chyba z neuronů výstupní vrstvy. Upraví váhy na vstupech do výstupní vrstvy tak, abychom dostali požadované hodnoty na výstupu. Pak se chyba šíří do první skryté vrstvy. Zde si síť podle zabudovaného algoritmu změní váhy na vstupech do neuronů.

- Tak se postupuje dál, až se chyba dostane až na vstupní neurony - receptory. Na těch se váhy nemění.

Síť si své váhy upravuje postupně. Vždy změní své váhy tak, aby výsledná chyba byla co nejmenší od požadovaného výstupu pro předložený vzor. Potože se však síti předkládá řada úloh, které mohou být od sebe velmi odlišné, musí síť měnit váhy tak, aby „nezapomněla“ co se již naučila z předchozích vzorů a současně, aby se naučila rozpoznávat nový, předložený vzor.

Touto metodou se neuronová síť přizpůsobuje učebnímu souboru. Adaptuje se na zvolenou úlohu. Neuronové síti je v cyklech předládán učební soubor. Výsledek sítě je porovnán s požadovaným výsledkem a je-li chyba na výstupu sítě zanedbatelná, považujeme síť za naučenou. Je - li síť naučena, je schopna řešit úlohy, na které byla adaptována.

Poznámka: Tento způsob učení – back propagation – **využívám pro účely odhadu složitosti za pomoci neuronové sítě PERTA**. Proto mu věnuji pozornost. Existují i jiné způsoby učení, např. učení Kohonenových map [29,30,31]. Tento způsob není pro účely odhadu složitosti využíván. Nebudu se jím tedy podrobně zabývat.

2.9.4 Kritické zhodnocení neuronových sítí

Neuronové sítě, a především pak sítě Perceptron s metodou učení Backpropagation, jsou jistě vhodným nástrojem pro řešení toho typu úloh, kde neznáme algortimus výpočtu, ale známe správný výsledek. **Z tohoto důvodu využívám vícevrstvou perceptronovou neuronovou síť pro odhad složitosti softwarového produktu.** Neuronová síť má schopnost adaptace na zvolenou úlohu a po určitém množství učebních cyklů je schopna odpovídat na předložené vzory s určitou chybou.

Uvedu příklad neuronové sítě, která má za úkol rozpoznat typ operace na základě tří vstupních parametrů:

Vstupní vektor číslo:	Vstup 1	Vstup 2	Vstup 3	Odpověď sítě
1	5	5	25	Násobení
2	5	5	0	Odečítání
3	5	5	10	Sčítání

Tabulka 2.9.4 Učební soubor neuronové sítě

Kromě spolehlivosti odpovědí je síť také schopna zobecnění. Dostane-li na vstup nějaký „zašuměný“ vzor, umí jej opravit.

Příklad: Protože jsem síť naučil rozpoznávat pouze operace sčítání, odečítání a násobení, pouze na základě učebních dat uvedených v tabulce 2.9.4, bude na vstupní vektor (5, 4.5, 25) odpovídat zcela jistě „násobení“. Má totiž schopnost zobecnění a protože třetí parametr je 25, je pravděpodobné, že číslo 4.5 je zadáno chybně a má místo něj být uvedeno číslo 5. Síť provádí toto zobecnění na základě své naučenosti. Protože učební soubor obsahoval jen 3 vstupní vektory a síť tedy zná pouze tyto varianty vstupních vzorů, bude každý jiný vzor zobecňovat podle toho, jak se bude nejlépe k blížit některému ze tří naučených vzorů.

Neuronové sítě mají však „nectnost“. Nikdy nevíme jak se síť rozhoduje a proč se tak rozhodla. Zatím co u klasických elektronických obvodů pracujících na základě známých algoritmů jsme vždy schopni na základně vstupních hodnot určit výsledek, u neuronových sítí tomu tak není. Její vnitřní funkce, která na základě učení nakonec vygeneruje odpověd, byť je dána konečným množstvím vah a přenosových funkcí, nám zůstává utajena. Tudíž nevíme, odpoví-li síť vždy správně. Můžeme operovat jen s pravěpodobností, že odpověď bude z X procent špatná. To značně omezuje její možnost použití v oblastech, kde je nutné vědět, že dané rozhodnutí je stoprocentně správné.

Tudíž systémy u nichž je zásadní požadavek na stoprocentní bezchybovost, v současnosti nejsou řízeny neuronovými sítěmi. V době vzniku této disertační práce, na základě současných znalostí vyplývajících ze studia umělých neuronových sítí, nelze zaručit, že síť bude vždy fungovat správně.

2.10 Závěrečné shrnutí kapitol

Analýza dostpupných zdrojů jednoznačně ukazuje dva trendy v odhadu složitosti softwaru. Buď je odhad složitosti založen na funkcích, které jsou od vznikajícího softwaru požadovány a na podmínkách, které pro vytvoření produktu objektivně jsou. Jde o metody odhadu složitosti na základě funkčních jednic. Nebo jde o metody odhadu, které operují s již hotovým zdrojovým kódem. Odhad složitosti pak provádějí na základě informací dostupných ze zdrojového kódu.

Provedl jsem analýzu především metod FP a UCP. Tyto metody jsou založené na funkčních jednicích pracujích s ordinálními stupnicemi. Porušují pravidla operací s hodnotami naměřenými v ordinální stupnici. Z tohoto důvodu musí být tyto metody odhadu vždy do určité míry nepřesné. Ač se autoři metod snaží korigovat výsledky metod pomocí různých opravných koeficientů, zaváděním konstant atd., nemůže být jejich snaha odměněna dostatečně dokonalou metodou.

Metody Funkčních jednic nijak nezatracuji. Je jistě vhodné a pravděpodobně jediné možné využívat funkčních jednic k odhadu složitosti. Domnívám se však, že je potřeba se pečlivě vyhýbat jakýmkoliv operacím, jenž budou pro hodnoty naměřené v ordinální stupnici neinterpretovatelné.

Odhad složitosti prováděný na základě znalosti zdrojového kódu či strukturovaného zadání je smysluplný. Pokud se podaří zajistit kvalitní tým programátorů, jenž budou programovat ukázněně za předem domluvených konstrukcí, bude možné odhad složitosti provádět. Problém spočívá v tom, že to bude již pozdě. Odhad složitosti je potřeba provádět co nejdříve. Na to ale nemáme potřebný zdrojový kód.

Naopak, odhad složitosti na základě grafu řízení, jenž může být zobrazením scénáře zadání, se mi zdá být smysluplný. I zde samozřejmě platí, že scénář může být napsán úmyslně zjednodušeně. Jeho zobrazení grafem řízení bude jednoduché, složitost bude vycházet nízká. O tomto problému jsem již hovořil v předchozích

kapitolách.

Důležitým poznatkem, který jsem při analýze dostupných zdrojů získal je:

Na složitosti softwarového návrhu se zásadně projevuje vliv prostředí. Proto, vzniká-li jakákoliv nová metoda odhadu složitosti, měla by už ze své podstaty s vlivem prostředí počítat a nějak jej zohlednit ve svých výpočtech.

3 Analýza řešení

Pro plánování jakéhokoliv projektu jehož výsledkem má být nějaký softwarový produkt je třeba znát odhad nákladů na jeho provedení a minimální čas, potřebný pro jeho realizaci.

Odhadnout čas a náklady na softwarový projekt je účelné pokud možno co nejdříve. To je v tom okamžiku, kdy již máme dostatečně dobře shrunuty požadavky na software. Díky překrývání jednotlivých etap ve fázích dle UP viz [7] není možné zcela přesně říci, jestli k tomu dochází v první (Založení) či druhé fázi (Rozpracování požadavků na software) životního cyklu projektu. Dokud však nezačneme „naplno programovat" je vhodné složitost budoucího software nějak odhadnout. Nejprve je ale nutné vymezit pojem „složitost".

Složitost obecně je viz [2] „*chápána jako míra úsilí, které potřebuje člověk vynaložit k tomu, aby produkt navrhl, vyvinul, implementoval, provozoval a udržoval v průběhu jeho životního cyklu"*.

Za účelem odhadu složitosti softwarového produktu navrhuji chápat složitost jako míru úsilí, které potřebuje člověk vynaložit k tomu, aby produkt navrhl a vyvinul. Při čemž pod pojmem vyvinul budeme chápat stav, kdy bylo dosaženo zvoleného cíle. V případě tvorby programu pro evidenci knih v knihovně budeme pod pojmem vyvinul chápat okamžik, kdy je program:

- naprogramován,
- otestován,
- s uživatelskou dokumentací,
- připraven k předání uživateli.

Samotné nasazení softwaru do provozu a jeho údržbu už do odhadu složitosti nezahrnuji. Složitost odhaduji za účelem rozhodnutí, zda projekt, jehož cílem je finální softwarový produkt, realizovat, jak dlouho realizace bude tvrat a kolik bude zhruba stát.

Míra úsilí je ovlivněna faktory vnějšími a vnitřními. Student připravující se na zkoušku z matematiky bude ovlivněn jednak dostupností materiálů, počtem cvičení, kvalitou cvičícího (faktory vnější), ale i vlastní leností, nebo neschopností udržet pozornost (faktory vnitřní). Proto je nutné tyto faktory při odhadu složitosti zohlednit.

3.1 Využití diagramu užití k odhadu složitosti softwaru

V předchozích kapitolách jsem uvedl jak diagram užití může graficky znázornit případy užití. Případy užití jsou využívány v počátečních(první a druhá) fázích životního cyklu softwarového produktu, v etapě specifikace požadavků na softwarový produkt. Slouží pro specifikaci požadavků kladených na vznikající software. Je-li specicifikace požadavků kompletní, pak jsou i kompletní všechny případy užití (podle RUP). Protože pro grafické znázornění případů užití může být použit digram užití (jeden nebo více) nabízí se myšlenka využít jej nějak k odhadu složitosti softwaru.

Dovolte ilustrační příklad. Předpokládejme, že jsme firma specializující se na tvorbu systémů pro knihovny. Dostali jsme zakázku vyvinout jednoduchý program pro evidenci půjčených knih.

Zadání projetku:

V podnikové knihovně „*K*“ je udržována databáze knih. Data do databáze jsou vkládána prostřednictvím informačního systému „*I*“. S tímto informačním systémem pracuje vyškolený pracovník „*Knihovník*“. Knihovník má přístup k seznamu knih, jenž jsou v knihovně uloženy. Jejich umístění v příslušných regálech je uvedeno v databázi knih. Pokud však přijde čtenář do knihovny a potřebuje si vypůjčit knihu, dochází ke komplikacím.

Knihovník musí příslušnou knihu vyhledat v informačním systému. Po té, co ji nalezne, jde k příslušnému regálu a zde knihu fyzicky vyhledá. Pokud si však již někdo knihu půjčil, knihovník ji v regálu nenajde. Musí se podívat do zvláštního sešitu, který vede. V něm nalezne jméno, komu byla kniha vypůjčena a datum jejího vrácení.

Tento posup je nepohodlný protože:

- Čtenář nemá seznam knih.
- Čtenář, pokud je požadovaná kniha vypůjčena, si nemůže nalézt knihu podobnou, která vypůjčena není:
  - o to musí žádat knihovníka.

- Knihovník musí vést speciální sešit kde si píše seznam vypůjčených knih.
    - Knihovník nemá rychle k dispozici seznam vypůjčených knih.

Úkol v ilustračním příkladu je náseledující :

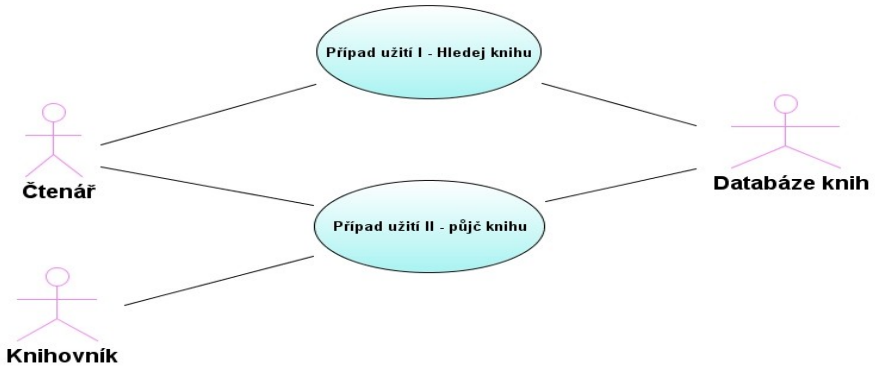
- Umožnit čtenáři vyhledávat v seznamu knih.
- Zobrazit čtenáři knihy k dispozici.
- Zobrazit čtenáři knihy vypůjčené.
- Umožnit čtenáři si knihy vypůjčit.
- Umožnit knihovníkovi sledovat seznam vypůjčených knih.
- Využít stávajícího databázového systému pro sledování knih.

Případy užití v ilustračním případu pak mohou být následující:

1. Případ užití - **Hledej knihu:**
    - Čtenář má k dispozici seznam knih, ve kterém může hledat.
    - Nalezne-li požadovanou knihu, požaduje od systému informaci o tom, zda byla kniha vypůjčena.
    - Databáze knih poskytuje Čtenáři potřebná data.
    - atd.
2. Případ užití - **Půjč knihu:**
    - Čtenář si chce vypůjčit knihu:
        - Čtenář potřebuje mít možnost knihu rezervovat.
        - Čtenář potřebuje uvést, kdy má být kniha k dispozici aby si ji mohl vyzvednout.
    - Knihovník potřebuje vidět seznam knih:
        - rezervovaných k vypůjčení,
        - které mají již být vráceny, ale nejsou.
    - Databáze knih poskytuje Čtenáři a Knihovníkovi potřebná data.
    - atd.

Poznámka: Jde o modelový příklad. Je úmyslně zjednodušen. Je samozřejmě možné jej rozebírat do mnohem větších detailů. To ale není účelem této kapitoly. Účelem je ukázat jak mohou diagramy užití posloužit při odhadu složitosti software.

Diagram užití pak může mít podobu zobrazenou na Obr. 3.1.



Obr. 3.1 Modelový příklad diagramu užití

Je z něj patrné násedující:

- Požadovaná funkcionalita je realizována pomocí dvou případů užití:
  - Hledej knihu,
  - Půjč knihu.
- Čtenář má vazbu na dva případy užití.
  - Hledej knihu,
  - Půjč knihu.
- Knihovník má vazbu na jeden užití:
  - Půjč knihu.
- Databáze knih má vazbu na dva případy užití:
  - Hledej knihu,
  - Půjč knihu.

Pro odhad složitosti diagramu užití je možné využít:

- Počtu případů užití v diagramu užití,
- Počtu aktorů v diagramu užití,
- Rolí aktorů v diagramu užití:
  - Knihovník,
  - Čtenář,

- Databáze knih,
- atd.

- Počtu vazeb v diagramu užití.

3.2 Odhad složitosti softwarového produktu na základě diagramu užití

V této kapitole vymezuji pojem základní složitost diagramu užití, základní složitost softwarového produktu, faktory složitosti a celková složitost softwarového produktu. Jak již bylo uvedeno, složitost chápejme jako míru úsilí nutnou vynaložit za účelem dosažení zvoleného cíle. Tato složitost je ovlivněna faktory vnitřními a vnějšími. Pokud se konečně (zpravidla jednou týdně) odhodlám jít běhat, pak je moje míra úsilí dána cílem, který jsem si zvolil = uběhnout 5 km. Mimo to se na realizaci tohoto cíle podílejí faktory vnitřní – celková únava, lenost, obezita ale i faktory vnější – počasí, nebo nepředpokládaně rozkopaná silnice na trati. Proto nemůže být míra úsilí vždy stejná.

Proto navrhuji pro účel odhadu složitosti softwarového produktu definovat:

- Základní složitost diagramu užití - *ZS.*
- Základní složitost softwarového produktu - *ZSP*.
- Faktor (faktory) Složitosti ovlivňující složitost celého softwarového produktu - *FS.* Ty navrhuji považovat za **Manažerské charakteristiky složitosti vývoje IS**. Za účelem jejich klasifikace je navrhuji dělit na:
  - Faktory Složitosti klasifikovatelné *FSk.* Ty navrhuji považovat za **Manažerské charakteristiky zadání požadavků na IS.**
  - Faktory Složitosti klasifikovatelné obtížně *FSo.* Ty navrhuji považovat za **Manažerské charakteristiky prostředí vývoje IS**.

- Celkovou složitost softwarového produktu - *CSP*, která je „nějakou" funkcí základní složitosti softwarového produktu a faktorů ovlivňujících složitost softwarového produktu $CSP = f(ZSP,FS)$. Tu navrhuji považovat za **Základní manažerskou charateristiku vývoje IS.**

3.2.1 Základní složitost diagramu užití

Základní složitostí diagramu užití budeme chápat počet případů užití v diagramu užití. Případ užití musí být minimálně jeden v jednom diagramu užití.

3.2.2 Základní složitost softwarového produktu

Během procesu tvorby případů užití a jejich grafického zaznamenávání do diagramu užití je téměř vždy potřeba využít většího počtu diagramů užití. Většina softwarových produktů totiž není tak jednoduchá, aby všechny případy užití mohly být zaznamenány v jednom diagramu užití. Proto je vhodné zavést pojem základní složitost softwarového produktu *ZSP*, který je roven součtu základních složitostí všech vzniklých diagramů užití pro zvolený softwarový produkt *ZSP*.

$$ZSP = \sum_{i=1}^{n} ZS_i$$

Pro jednoduchý příklad zobrazený na Obr. 3.1 bude *ZSP* = 2. V **jednom** diagramu užití jsou pouze **dva případy užití**.

3.2.3 Faktory ovlivňující složitost softwarového produktu

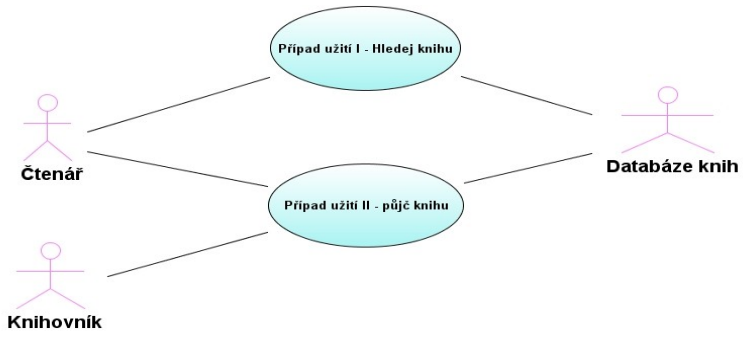
Každý softwarový produkt je ovlivněn některými faktory, které jeho realizaci zesložiťují nebo usnadňují. Tyto faktory navrhuji rozdělit na faktory klasifikovatelné - ty se dají vypočítat z diagramů užití a na faktory obtížně klasifikovatelné.

3.2.3.1 Faktory složitosti klasifikovatelné

Každý diagram užití obsahuje kromě případů užití také aktory a vazby mezi aktory a případy užití. Typ aktorů, počet aktorů a počet vazeb mezi případy užití pak jednoznačně nějak ovlivňuje složitost diagramu užití a tudíž i celého softwarového produktu. Proto je vhodné je při výpočtu složitosti nějak zohlednit. Nazývám je Faktory složitosti klasifikovatelné – *FSk*. Jejich klasifikací dostaneme **Manažerské charakteristiky zadání požadavků na IS.** Tyto charakteristiky pak slouží k rozohodování o realizaci softwarového produktu, o jeho ceně atd.

Navrhuji následující způsob výpočtu *FSk,* viz příklad na obrázku Obr. 3.2.3.1:

Na Obr. 3.2.3.1 vidíme nám už známý příklad diagramu užití.



Obr. 3.2.3.1

Vidíme, že aktor Čtenář má dvě vazby s dvěma případy užití. Stejně tak na tom je i aktor Databáze knih. Knihovník má vazbu jen na jeden případ užití. V diagramu máme tři typy aktorů. Samozřejmě i typ aktora nějak ovlivňuje složitost případu užití, tím diagramu užití a tudíž i celého softwaru. Protože aktoři mohou mít různá jména a různé role v diagramu užití stojíme před problémem, jak roztřídit jednotlivé role. Toto třídění navrhuji udělat podle toho, jak jsou si role navzájem podobné.

Vhodným způsobem se zdá být navržení nějaké třídy rolí aktorů viz Tabulka 5. Lze totiž předpokládat, že lze nalézt a vytvořit „nějakou“ třídu rolí aktorů, která by obsahovala základní typy rolí, s kterými se lze v projektu setkat. V případě odhadu složitosti by se pak každé specifické roli v projektu, přidělila nějaká zobecněná role z třídy rolí.

Každému aktoru (ve vzorovém příkladu Knihovníkovi nebo Databázi knih atd.) zobrazenému v diagramu užití pro daný softwarový produkt, bude přiřazena ta role z třídy rolí, jenž pro něj bude nejvýstižnější.

Pro účel odhadu složitosti softwarového projektu jsem navrhl následující tabulku rolí viz Tabulka 5. Jednotlivé role jsou jednotlivými faktory složitosti *FSk*. Ty se stávají „manažerskými charakteristikami zadání požadavků na IS“. Jsou vypočítatelné z diagramu užití. Tudíž jsou klasifikovatelné na základě funkční specifikace na softwarový produkt.

Proto je nazývám **Manažerské charakteristiky zadání požadavků na IS.**

Tato třída obsahuje následující typy rolí viz Tabulka 5:

Třída rolí aktorů		
Číslo role	Jméno role	Popis role (jednotlivé role je možné upravit dle potřeby vznikajícího sotwarového produktu)
1	Uživatel	Tato role je přiřazena každému aktoru, který má v systému základní práva typu číst veřejně dostupné dokumenty, vyhledávat nad nimi atd. Zástupce této role je například běžný uživatel operačního systému Windows bez práv instalace programů na disk, s předpřipravenými programy k použití.
2	Uživatel s rozšířenými právy	Tato role je přiřazena každému aktoru, který má povolený částečný nebo úplný přístu k datům, která vyžadují nějakou autorizaci. Zástupce této role je například uživatel operačnho systému Windows, který vlastní přístupová práva k některým programům jako je elektronická pošta atd.
3	Administrátor s omezenými právy	Tato role je přiřazena každému aktoru, který má povoleno udržovat a spravovat vybranou část informačního systému. Nemá však práva administrovat celý informační systém. Zástupcem této role je například uživatel systému Windows, který má práva instalovat soubory na disk, přidávat jiné uživatele pod svůj uživatelský účet. Zde může nastavovat přístup jiným uživatelům nižší nebo stejná přístupvá práva, jako má on.
4	Administrátor	Tato role je přiřazena každému aktoru, který má absolutní práva správy informačního systému. Zástupce této role je například Administrátor v operačním systému Windows, který má úplná práva číst, mazat, zapisovat a přídělovat práva ostatním uživatelům v celém operačním systému.
5	Relační databáze	Tato role je přiřazena každému programu - aktoru, který zastupuje relační systém správy dat.
6	Jiný druh databáze	Tato role je přiřazena každému programu – aktoru, který zastupuje jiný než relační systém pro správu dat.
7	File systém	Tato role je přiřazena každému programu – aktoru, který pracuje (otevírá, přepisuje, ukládá, maže) se soubory ve stromové stuktuře.
8	Jiný program	Tato role je volně využitelná pro specifické účely vznikajícího softwarového produktu a je jí přiřazen jakýkoliv jiný program – aktor, kterému nelze přiřadit některou z navržených rolí.
9	Operační systém	Tato role zastupuje program – aktora, který je operačním systémem.
10	Nespecifikovaná role	Tato role je volně využitelná pro specifické účely vznikajícího softwarového produktu, je možné ji využít dle potřeby odhadu.

Tabulka 5: Třída rolí FSk [Pavlíček]

Role aktorů jsou stejné pro všechny role vystupující v diagramech užití pro navrhovaný softwarový produkt. Vyskytuje-li se příslušná role v specifikaci pro softwarový produkt, musí mít minimálně jednu vazbu s minimálně jedním případem užití v specifikaci na produkt. Upravením příslušné tabulky pro daný softwarový produkt získáme faktory složitosti klasifikovatelné *FSk,* které jsou manažerskými charakteristikami zadání požadavků na IS,

3.2.3.1.1 Příklad nalezení faktorů složitosti klasifikovatelných

Dle příkladu zobrazeném na obrázku Obr. 3.2.3.1 v předchozí kapitole navrhuji pro výpočet *FSk* postupovat následujícím způsobem:

1. Nejprve přiřadíme aktorům příslušné role z tabulky rolí.
   - Může se stát, že navržená granularita rolí neodpovídá potřebám, specifikovaným v požadavcích na softwarový produkt. Z tohoto důvodu jsou v tabulce předpřipraveny dvě nespecifikované role (jedna pro aktora typu program, jedna pro aktora typu člověk). Ty je možné využít pro specifikcé účely vznikajícího softwarového produktu. Pokud ani to nestačí, je nutné tabulku rozšířit o další role. Tím se zvětší počet položek v třídě *FSk*. Pro účel odhadu složitosti softwarového produktu to není omezení. Jen je nutné v budoucnu dodržovat jednou stanovenou třídu rolí a neměnit ji pro jednotlivé softwarové produkty. Pro odhad složitosti musí být vždy zachován stejný počet položek třídy rolí, které pak zapisujeme ve vektoru $\vec{FSk}$. Tento vektor představuje již ohodnocené faktory složitosti *FSk*.
2. Pak spočítáme počet vazeb každé role.
3. Zapíšeme je do tabulky nebo do vektoru.

Add bod 1:

- Čtenáři je přiřazena role „Uživatel“.
- Knihovník má roli „Uživatel s rozšířenými právy“.
- Databáze knih má roli „Relační databáze“.

Add bod 2:

- Role „Uživatel“ má 2 vazby.
- Role „Uživatel s rozšířenými právy“ má 1 vazbu.
- Role „Relační databáze“ má 2 vazby.

Add bod 3:

Tabulka přiřazených rolí		
Číslo role	Jméno role	FSk
1	Uživatel	2
2	Uživatel s rozšířenými právy	1
3	Administrátor s omezenými právy	0
4	Administrátor	0
5	Relační databáze	2
6	Jiný druh databáze	0
7	File systém	0
8	Jiný program	0
9	Operační systém	0
10	Nespecifikovaná role	0

Tabulka 6: Vzor výpočtu FSk [Pavlíček]

Získané hodnoty pak zapíšeme do příslušného vektoru:

$$\vec{FSk} = (2,1,0,0,2,0,0,0,0,0).$$

Tím získáme ohodnocené **Manažerské charakteristiky zadání požadavků na IS**.

3.2.3.2 Faktory složitosti obtížně klasifikovatelné

Každý projekt (slovem projekt budeme chápat plánování, sběr uživatelských požadavků na požadovaný software, jejich analýzu, řízení vzniku softwarového produktu včetně jeho otestování, s vzniklou uživatelskou dokumentací) je ještě ovlivněn řadou dalších faktorů, které se klasifikují obtížně. Navrhuji je značit *FSo*. Na rozdíl od *FSk*, nejsou tyto faktory patrné z diagramu užití. Jde o faktory zohledňující například:

- Prostředí firmy, která projekt realizuje.
- Typ software, který vzniká.
- Použitý programovací jazyk.
- Druh vývojářkého týmu.
- Požadavek na stabilitu vyvíjeného software.
- A řada dalších.

Pro tyto faktory je třeba také zavést nějakou třídu faktorů *FSo*. Potíž spočívá v tom, že se tato třída bude patrně lišit u každé firmy. Tuto skutečnost beru v metodě odhadu složitosti softwarového produktu v úvahu. Protože součástí odhadu složitosti je adaptace neuronové sítě PETRA (podrobněji o ní hovořím v kapitole PETRA) na dané prostředí následujícím způsobem:

1. Vytvořme z již realizovaných softwarových produktů učební soubor pro PETRu.
   - Nalezněme základní složitosti softwarových produktů *ZSP*.
     - Ke každému softwarovému produktu nalezněme *FSk*.
     - Zjistěme kolik času (např. člověkodní) vývoj softwarového produktu spotřeboval.
   - Pro všechny realizované softwarové produkty nalezněme Faktory složitosti obtížně klasifikovatelné - *FSo*.
2. Naučme PETRu predikovat čas dle předložených učebních souborů.
3. Předložme požadavky na nový softwarový produkt, které jsme nalezli:
   - *ZSP*.
   - *FSk* (počet položek vektoru, který *FSk* představuje, musí být stejný jako v

učebních souborech).

- *FSo* (počet položek vektoru, který *FSo* představuje musí být stejný jako v učebních souborech).

4. Odhadněme složitost softwarového produktu.

Je možné *FSo* považovat za volitelné a závislé na každé firmě. Problém s *FSo* může nastat v okamžiku, kdy firma nemá dostatečná data k naučení sítě PETRA.

Dalším problémem je to, že tyto faktory by měly být voleny s rozvahou. Proto je pro účel metody důležité navrhnout třídu nějakých základních – obecných faktorů, které každá firma bude nějak splňovat. Tím fakticky navrhnout a vytvořit **Manažerské charakteristiky prostředí vývoje IS,** které jsou pro účel výpočtu zastupovány faktory *FSo*.

Nabízí se pak myšlenka použít pro definování těchto faktorů stejný postup, který využívají i jiné metody odhadu, jmenovitě metoda COCOMO.

1. Faktory *FSo* chápejme jako podmínky realizace software. Navrhuji zavést následující třídu faktorů složitosti *Fso* viz Tabulka 7.

Třída faktorů FSo	
Číslo faktoru	Název faktoru
Faktory lidského typu	
1	Analýzu prováděl zkušený analytik
2	Tým má zkušenosti s podobným projektem
3	Kapacita týmu pro realizaci požadavků je dostatečná
4	Programátoři umí programovat v daném programovacím jazyce
5	Fluktulace jednotlivých členů je značná
6	Členové týmu mají dostatek zkušeností s potřebnými nástroji
Faktory produktového typu	
7	Požadavek na polehlivost software je vysoký
8	S použitou databází jsou dostatečné zkušenosti
9	Požadavek na znovupoužitelnost softwaru je vysoký
10	Požadavek na programovou dokumentaci je vysoký
11	Požadavek na rychlost odezvy je vysoký
Faktory nasazení	
12	Software musí zabírat co nejméně místa na discích
13	Software musí zabírat co nejméně místa v paměti
14	Operační systém, pod jehož řízením bude software, nasazen je dostatečně stabilní
15	Software musí být multiplatformní
Faktory projektu	
16	Požadavky na funkčnost softwaru se často mění
Další, nespecifikované faktory	
17	Faktor 1 (v případě potřeby lze doplnit)
18	Faktor 2 (v případě potřeby lze doplnit)
19	Faktor 3 (v případě potřeby lze doplnit)

Tabulka 7: Třída faktorů FSo [Pavlíček]

2. Každý faktor navrhuji klasifikovat podle ordinální stupnice**:**
   - 0 - Faktor neexistuje nebo nemá vliv.
   - 1 – Faktor existuje má střední vliv.
   - 2 – Faktor existuje, jeho vliv je zásadní.

Dle typu softwarového produktu může být potřeba do odhadu složitosti ještě zahrnout nějaké další faktory ovlivňující složitost softwarového produktu, které nejsou v Tabulce 7 uvedeny a rozšířit tak třídu faktorů *FSo*. Proto jsou poslední tři řádky (faktory 17 až 19) uvedeny jako nespecifikované faktory. Na jejich místo je pak vhodné umístit ty faktory, které nejsou v třídě faktorů *FSo* navrženy.

Protože se může jednat o nějaké faktory, které jsou svou povahou odlišné od navržených faktorů a nelze je klasifikovat podle stejné ordinální stupnice, která je využita pro klasifikaci navržených faktorů, je možné pro ně vytvořit jinou stupnici pro jejich klasifikaci. To není v rozporu s metodou odhadu složitosti. Uděláme to následujícím způsobem:

Předpokládejme, že je zapotřebí do odhadu složitosti softwarového produktu nějak zohlednit i to, který analytik se podílel na vzniku jednotlivých případů užití popisujících funkční požadavky na budoucí software. Jde jistě o nějaký faktor složitosti. Proto na místě faktoru 17 budeme uvádět autora případů užití. Autorů případů užití může být ale větší množství. Proto navrhuji vytvořit například následující tabulku (Tabulka 8), podle které budeme jednotlivé autory klasifikovat.

Tabulka autorů případů užití	
Číslo autora	Jméno autora
1	Josef Pavlíček
2	Martin Vodák
3	Ivan Kučera
4	Josef Kadeřavý
5	Alexander Dubnik

Tabulka 8: Tabulka autorů případů užití [Pavlíček]

Faktor 17 – jméno autora pak budeme klasifikovat podle Tabulky 8. Každému

autorovi, který se podílel na analýze, přiřadíme číslo uvedené v tabulce a vložíme jej na sedmnácté místo v tabulce *FSk.*

Tento způsob je možný. Obecně lze říci, že pro odhad složitosti není důležité aby faktory složitosti *FSo* byly klasifikovány pomocí jednotné stupnice. Každý faktor v třídě faktorů *FSo* může být klasifikován podle jedinečné stupnice. Jedinné a postačující pro zajištění správnosti odhadu složitosti je, aby každý faktor *FSo* měřený pro každý jednotlivý softwarový produkt byl vždy klasifikován podle stupnice, která mu byla na navržena. Toto pravidlo je nutné stále dodržovat. Od okamžiku, kdy vytvoříme učební soubor je nutné jej neměnit.

Nestačí-li počet faktorů navržených v třídě faktorů *FSo,* pak lze třídu rozšířit o více položek. Toto řešení není proti pravidlům metody odhadu složitosti. Rozšíření musí splňovat pravidla definovaná v předchozí kapitole - faktory složitosti klasifikovatelné *FSk.* Z mých praktických zkušeností však vyplývá, že to není nutné.

3.2.3.2.1 Příklad nalezení faktorů složitosti obtížně klasifikovatelných

Pro modelový softwarový produkt uvedený v příkladu v kapitole 3.1 ukáži, jak nalézt a ohodnotit faktory *FSo*. Každý faktor ohodnotím dle ordinální stupnice uvedené v předchozí kapitole: (0 nemá vliv, 1 – má střední vliv atd.) viz Tabulka 9.

Třída faktorů FSo		
Identifikátor faktoru	Název faktoru	FSo
Faktory lidského typu		
1	Analýzu prováděl zkušený analytik	2
2	Tým má zkušenosti s podobným projektem	2
3	Kapacita týmu pro realizaci požadavků je dostatečná	2
4	Programátoři umí programovat v daném programovacím jazyce	2
5	Fluktulace jednotlivých členů je značná	0
6	Členové týmu mají dostatek zkušeností s potřebnými nástroji	2
Faktory produktového typu		
7	Požadavek na polehlivost software je vysoký	0
8	S použitou databází jsou dostatečné zkušenosti	2
9	Požadavek na znovupoužitelnost softwaru je vysoký	0
10	Požadavek na programovou dokumentaci je vysoký	0
11	Požadavek na rychlost odezvy je vysoký	0
Faktory nasazení		
12	Software musí zabírat co nejméně místa na discích	0
13	Software musí zabírat co nejméně místa v paměti	0
14	Operační systém, pod jehož řízením, bude software nasazen je dostatečně stabilní	2
15	Software musí být multiplatformní	0
Faktory projektu		
16	Požadavky na funkčnost softwaru se často mění	0
Další, nespecifikované faktory		
17	Faktor 1 (v případě potřeby lze doplnit)	0
18	Faktor 2 (v případě potřeby lze doplnit)	0
19	Faktor 3 (v případě potřeby lze doplnit)	0

Tabulka 9: Tabulka faktorů FSo - [Pavlíček]

Pokud jsem nalezl všechny faktory *FSo*, přepíši je do vektoru:

$$\vec{FSo} = (2,2,2,2,0,2,0,2,0,0,0,0,0,2,0,0,0,0)$$

Tím získáme ohodnocené **Manažerské charakteristiky prostředí vývoje IS**.

3.2.4 Celková složitost softwarového produktu

Jak jsem již uvedl, celková složitost softwarového produktu *CSP* je "nějakou" funkcí základní složitosti softwarového produktu *ZSP* a faktorů složitosti *FS*. Slovo "nějakou" uvádím zcela úmyslně v uvozovkách.

Není to proto, že bych snad chtěl zavádět nějaký vágní pojem do mé práce, ale prostě proto, že tato funkce je neznámá. Ze svých zkušeností usuzuji, že není možné nalézt obecně platný vzorec, který by po dosazení vhodných konstant mohl celkovou složitost vyčíslit. Je to dáno jednoznačně tím, že veškeré atributy měření jsou ordinálního typu. Tudíž již od samého počátku je velmi problematické s nimi provádět nějaké operace. **Proto navrhuji celkovou složitost softwarového produktu *CSP* chápat jako funkci základní složitosti softwarového produtku *ZSP* a faktorů složitosti**. *CSP* pak navrhuji považovat za **základní manažerskou charakteristikou vývoje IS**, protože dává možnost se na jejím základě rozhodnout, zda realizovat softwarový produkt, jaké zdroje pro jeho vývoj alokovat, či uvažovat o jeho výsledné ceně.

Nalezení vhodné, ale specifické pro daný problém, funkce je úkol pro neuronovou síť PETRA. Ta je pomocí dat, naměřených z již realizovaných softwarových produktů, adaptována k řešení tohoto úkolu. Její nespornou výhodou je její adaptace na zvolené prostředí. Jedinnou podmínkou její úspěšné adaptace je vstupní vektor hodnot a vhodně zvolený učební soubor. Ten musí být předkládán v cyklech. Je-li učební soubor kompletně zpracován a síť nedává dobré výsledky, předloží se ten samý soubor k učení znovu. To se automaticky provádí tak dlouho, dokud není síť naučena.

Celková složitost softwarového produktu pak bude následující:

$$CSP = f(ZSP, FS) \quad ,$$

$$CSP = f(ZSP, (\vec{FSk}, \vec{FSo})) \quad .$$

Poznámka: Neuronová síť PETA, jako každá neuronová síť podobného typu, se chová zjednodušeně řečeno tak, jak jsme jí to naučili. Učení probíhá pomocí učebního souboru, kde na jedné straně je vstupní vektor hodnot, a na druhé straně je správný výsledek viz:

jeden řádek učebního souboru souboru sítě PETRA je tvořen:

CSP = 'správná hodnota´

- na levé části Celkovou složitostí softwarového produktu *CSP,*
- a v pravé části je správná hodnota (například v člověkodnech).

Bezpodmínečně nutné pro naučení sítě, je zachovat stejnou strukturu (velikost vstupního vektoru musí být stejná) učebního souboru a všechny předkládané hodnoty musí být ve stejných jednotkách. Síť se je schopna po určitém množství učebních cyklů (epoch) adaptovat na předkládané stupní vektory – vzory. Pak je schopna, na základě naučení, predikovat hodnoty odhadu složitosti v závislosti na předložených vstupních datech.

Celková složitost softwarového produktu je pak nejčastěji predikce člověkohodin, nebo člověkodní, které budou potřeba na vyvinutí softwarového produktu s požadovanou funkčností.

Nespornou výhodou mého řešení je, že se síť fakticky adaptuje na skutečné prostředí skutečné firmy. Jednotlivé faktory složitosti pak ovlivňují *CSP* v kontextu na prostředí, kde budou funkčnosti realizovány.

Síť je tedy schopna využít vztahů existujících mezi faktory složitosti právě v rámci jedné firmy. Tyto vztahy jsou ve skutečnosti velmi těžce popsatelné. Téměř vůbec je nejsme schopni měřit. Můžeme je popsat slovy. Slovům můžeme přiřadit nějaká čísla. Ale s těmito čísly bychom neměli dělat další operace, jako je vážení a průměrování. Právě pouhý popis typu – faktor má zásadní vliv 2, má střední nebo malý vliv – 1, neexistuje 0, je pravděpodobně ideální.

To ale naprosto nestačí v klasické metodě Funkčních jednic, nebo Use Case Points. Naopak v případě metody, kterou navrhuji, s použitím umělé inteligence, je tento popis dostačující. Stroj sám pomocí učebních příkladů nalezne pro dané prostředí ten nejlepší "aproximační" vzorec.

Nevýhodou stále zůstává, že tento "vzorec" je nám skryt[14] a nikdy jej nebudeme moci znát.

14 Teoreticky je vzorec samozřejmě možné nalézt. Protože počet neuronů v síti je vždy znám, je známa i jejich přenosová funkce, je možné zjistit váhy na vstupech do jednotlivých neuronů, teoreticky je možné funkci odhadu složitosti nalézt. Je to však tak náročné, že to postrádá reaálný význam viz [29].

3.3 Kritika odhadu celkové složitosti softwarového produktu

Kritika odhadu *CSP* spočívá v nepřesnosti výpočtu základní složitosti diagramu užití *ZS*. Jde opět o diskutabilní způsob, jak získat alespoň nějaká data k výpočtu.

Bohužel, lépe se pravděpodobně ve fázi Rozpracování požadavků na software popsat počet požadovaných funkčností na software nedá. Vždy stojíme před problémem, zda se jeden případ užití nedá dekomponovat na víc dílčích případů užití. Tohoto nebezpečí si ve své práci všímám a snažím se jej odstranit pomocí faktorů složitosti.

Podobným problémem se zabývá celá řada výzkumných týmů. V metodě Use Case Point se tomuto problému věnují a snaží se jej korigovat za pomoci různých vah. Těmi pak váží příslušný bod případu užití. Tento způsob nepovažuji za vhodné řešení.

Stejně tak jako hodnota *ZS* v mojí metodě, i bod v *UCP* je číslo získané měřením za pomoci ordinální stupnice. Násobení jej nějakým koeficientem (vahou) nemá tudíž smysl.

Domívám se však, že existuje jiná cesta, která může popsat složitost případu užití relativně přesně. Jedna velká sotwarová firma (úmyslně není uvedena) ve svých pracech používá[15] zajímavý způsob zpřesnění odhadu počtu funkcí na základě funkční specifikace. Celý princip spočívá v tom, že odhad počtu funkcí provádí vždy jeden vyškolený pracovník.

15 Bohužel se mi nepodařilo získat tištěný dokumet. Tuto informaci jsem získal na konferneci v San Francisku v září 2005.

Pracovník si po určité době vytvoří:

- znalost prostředí, pro které odhad provádí, například:
  - specializuje se na bankovní software,
  - specializuje se na obchodní domy,
  - atd.
- znalost vývojářského týmu, ve kterém pracuje, například:
  - spolupracuje s Františkem, Ivanem atd.
- znalost vývojových nástrojů, které při vývoji software využívají, například:
  - Java,
  - Oracle databáze.
- a další.

Jeho zkušenosti vytváří jistý kontext firmy, pro kterou jsou jeho odhady:

- relativně přesné,
- mají podobnou granularitu.

Zjednodušeně řečeno, můžeme se spolehnout, že daný pracovník po čase vytváří případy užití stejné – stejně složité. Toto samozřejmě není myšlenka nová. Na první pohled je možné pracovníka ohodnotit nějakou konstantou a vždy jeho výsledky jí násobit. Tak byl presentován odhad složitosti výše zmiňovanou firmou. Tento způsob je samozřejmě též problematický. Nic méně, vede mě k přesvědčení, že musí existovat závislost mezi autorem a případem užití, jím vytvořeným.

Tato závislost se dá modelovat pomocí neuronové sítě na základě znalostí z předchozích vyvinutých softwarových produktů. Stane-li se autor (je-li zahrnut do *FSo*) součástí *FSo*, pak se síť adaptuje i na autora a je relativně přesně schopna predikovat složitosti na základě znalostí z předchozích vyvinutých softwarových produktů, právě pro tohoto autora.

Toto řešení má tu výhodu, že **neprovádím žádné násobení čísel získaných z ordinálních stupnic**. Při tom řešení zachovává (rozpoznává) autora a dle zanlostí o něm predikuje složitost. Mimo to, je zde zachována relativně velká přesnost popisu ZS^{16}.

Z těchto závěrů usuzuji, že **za výše uvedeného postupu s neuronovou sítí PETRA, je celková složitost softwarového produktu *CSP* odhadnutelná.**

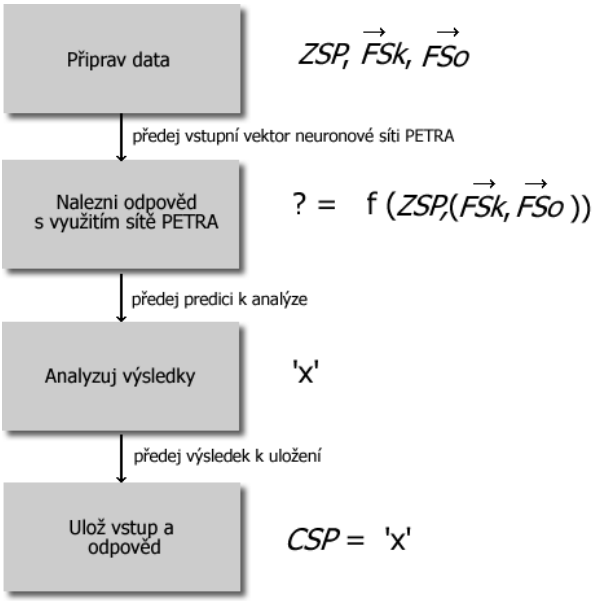
16 Samozřejmě za předpokladu schopného autora. Ne začátečníka.

4. Metoda odhadu CSP

4.1 Metoda CSP – postup

V této kapitole popisuji moji metodu odhadu složitosti softwaru na základě výpočtu *ZSP* a stanovením příslušných faktorů složitosti *FS*.

Tato metoda odhadu má následující charakter, viz Obr. 9.



Obr. 7: Metoda predikce [Pavlíček]

1. Prvním krokem odhadu složitosti je nejprve připravit vhodný vstupní vektor naměřených dat. Tím je:
   - *ZSP*
   - $\vec{FSk}$
   - $\vec{FSo}$

   $$\vec{vstup} = (ZSP, \vec{FSk}, \vec{FSo}).$$
2. Druhým krokem je předložení vstupního vektoru neuronové síti. Ta odpoví jednou číselnou hodnotou. Záleží na způsobu naučení sítě, má-li tato hodnota představovat člověkoměsíce, člověkohodiny, počet dní atd.

3. Třetím krokem je analyzovat výsledek. Výsledky lze rozdělit do tří skupin:
   - Výsledek je pravděpodobný. Výsledek uložíme do nějaké báze dat v libovolném formátu.
   - Výsledek je nepravděpodobný. Uložíme jej do báze dat a porovnáme jej po čase se skutečností.
   - Výsledek je nesmyslný. Znamená to, že je síť špatně naučena, nebo jsme zadali špatně vstupní vektor, nebo se jedná o nějaký další problém, který je vhodné zkoumat. Nejčastěji k tomuto problému dochází, je-li síť přeučena.

4. Uložíme data do "databáze zanalostí", kterou nazývám "**znalostní báze**". Tato báze dat v sobě uchovává informaci o příslušném realizovaném softwarovém produktu. Je součástí mého návrhu metody odhadu složitosti softwarového produktu. Z této báze dat jsou pak vytvářeny nové učební soubory pro neuronovou síť PETRA. Čím více máme popsaných realizovaných softwarových produktů v znalostní bázi, tím lépe můžeme adapovat síť na prostředí firmy a tudíž můžeme i zpřesnit její odhady.

Pro znalostní bázi navrhuji použít následující formát:

vstupní vektor = výsledek,

$$\vec{vstup} = výsledek\,,$$

$$\vec{vstup} = (ZSP\,, (\vec{FSk}\,, \vec{FSo})).$$

Výsledky ukádáme do znalostní báze proto, aby se nám neustále rozrůstal učební soubor pro neuronovou síť PETRA. Tím vlastně vytváříme znalostní bázi pro budoucí odhad složitosti.

5 PETRA

Tato kapitola pojednává o využití neuronové sítě PETRA, kterou jsem navrhl za účelem odhadu složitosti softwarového produktu.

5.1 Architektura a topologie neuronové sítě PETRA

Neuronová síť PETRA (PErcepTRon Artificial[17] inteligence) je perceptronová neuronová síť s jednou skrytou vrstou, s jedním výstupním neuronem a se sigmoidální přenosovou funkcí jednotlivých neuronů, kterou jsem navrhl a naprogramoval v jazyce Java viz [40] pro účel odhadu *CSP*. Architekturu a topologii sítě jsem navrhl na základě studia odborné literatury viz [29,30] a na základě testů, které jsem během vývoje sítě a návrhu metody odhadu složitosti softwarového produktu provedl. Odborné problémy, které jsem řešil při konstrukci sítě PETRA, jsem konzuloval s Doc. Veselým.

Ukázalo se, že síť se sigmoidální přenosovou funkcí, s jednou skrytou vrstvou, ve které je vždy dvojnásobný počet neuronů než je neuronů ve vstupní vrstvě, které jsou spojeny každý s každým neuronem ve vstupní vrstvě a stejně tak je každý neuron skryté vrstvy spojen s jedním neuronem výstupní vrstvy, dává nejlepší výsledky. Tato architektura a topologie sítě je dostačujícící k tomu, aby se síť byla schopna naučit rozpoznávat vstupní vzory viz [29,30]. Proto jsem se ji rozhodl použít pro odhad složitosti *CSP*.

PETRA se učí pomocí mechanismu zpětného šíření (error backpropagation) chyby viz [29,30]. Tento mechanismus by viz [29,30] měl být dostatečně účinný pro její adaptaci na zvolené prostředí firmy.

17 PETRA – Byl jsem inspirován svou ženou Petrou. Během vývoje této sítě jsem se střetnul s řadou problémů především pak při procesu učení sítě. Tato „tvrdohlavost“ mi přišla tak povědomá, že jméno PETRA bylo již na snadě.

Navrhl jsem toto základní nastavení sítě PETRA:

- 30 neuronů na vstupní vrstvě (počet neuronů je stejný jako počet prvků ve vstupním vektoru).
- 60 neuronů v skryté vrstvě.
- 1 neuron ve výstupní vrstvě.
- Počáteční váhy neuronů jsou vždy voleny náhodně.

Protože však pro účely metody odhadu složitosti softwarového produktu může být potřeba zvětšit vstupní vektor (rozšířit třídu *FSk* nebo *Fso*), je PETRA kofigurovatelná. Uživatel má možnost si dle své potřeby určovat počet vstupních neuronů. Zachovává se však pravidlo, že počet neuronů v skryté vrstvě je vždy dvojnásobný.

5.2 Vytvoření znalostní báze a naučení sítě

Výše uvedená metoda odhadu složitosti je jednoduchá a lze jí takřka realizovat ihned. Pouze však za předpokladu, že je PETRA adaptována na prostředí firmy. To je však ve skutečnosti zásadní problém, jemuž je věnována tato kapitola.

Nebudu přesně uvádět, jak se neuronová síť adaptuje. To je problematika teorie neuronových sítí, která je obecně dostupná například v literatuře viz [26,29,30]. Popíši však postup, pomocí kterého je možné adaptovat síť PETRA.

5.2.1 Vytvoření znalostní báze

Pro naučení neuronové sítě PETRA je zapotřebí vytvořit znalostní bázi dat. Tato báze se skládá z vstupního vektoru na jedné straně a z požadované odpovědi na straně druhé.

$$\vec{vstup}_1 = výsledek_{1,}$$

$$\vec{vstup}_2 = výsledek_{2,}$$

…

…

…

…

$$\vec{vstup}_n = výsledek_n.$$

Vektory ve znalostní bázi by měly popisovat již realizované softwarové produkty. Velikost jednotlivých vektorů musí být vždy stejná. Obecně platí pravidlo, že čím více jich máme, tím lépe. Bohužel se primární data o projektech získávají zpětně velmi špatně. Tudíž je zapotřebí je vybírat velmi pečlivě. Lépe je mít menší množství kvalitních dat. Často je vhodné zavést ještě nějakou klasifikaci pro rozlišení typů softwarových produktů.

Proto navrhuji softwarové produkty dělit dle typu. Pro ty pak vytvářet různé znalostní báze. Například:

- Návrh a vytvoření softwaru či softwarové komponenty, u které je kladen důraz na vzhled a použitelnost uživatelského rozhraní.

- Návrh a vytvoření softwaru či softwarové komponenty, s využitím databázového systému.

- Návrh a vytvoření servisně orientované aplikace.

- a další.

Druhá možnost je nechat rozdělení projektů na síti. To navrhuji udělat tak, že ve vstupním vektoru $\vec{FSo}$ využijeme posledních položek. Jde o položky s názvem "Další, nespecifikované faktory 17,18,19". Ty jsem, na základě svých zkušenotí, určil k tomu, aby si je mohl uživatel metody libovolně přizpůsobit k vlastním účelům. Zde můžeme využít např. Položku číslo 17 k odlišení jednotlivých typů projektů. Na její místo pak vložíme číslo, klasifikující typ softwarového produktu.

Pro tento účel navrhuji zavést vzorovou třídu typů softwarových produktů. V této třídě budeme mít například následující položky viz Tabulka 9:

Třída typů softwarových produktů	
Číslo produktu	Jméno typu produktu
1	Návrh a vytvoření softwaru či softwarové komponenty, u které je kladen důraz na vzhled a použitelnost uživatelského rozhraní.
2	Návrh a vytvoření softwaru či softwarové komponenty, s využitím databázového systému.
3	Návrh a vytvoření servisně orientované aplikace.

Tabulka 10: Třída typů softwarových produktů [Pavlíček]

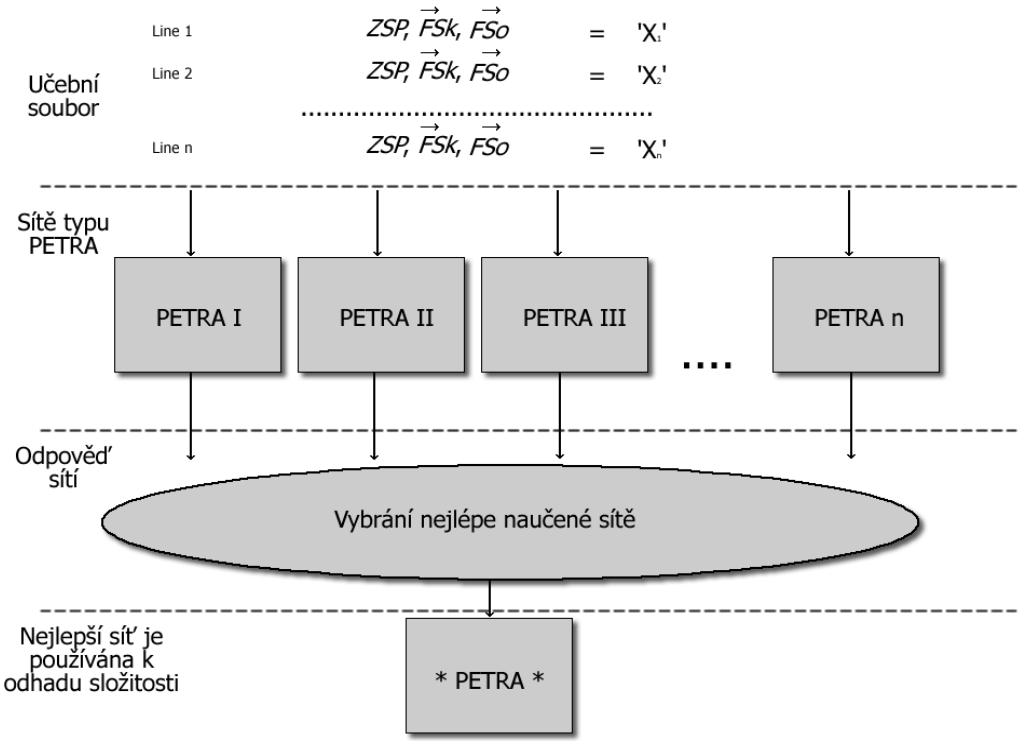
Číselná hodnota 1 až *N* neslouží pro výpočet. Pouze tím, že do vstupního vektoru sítě zadáme toto číslo rozlišíme jednotlivé projekty od sebe. Neuronová síť se pak naučí rozpoznávat projekty od sebe. Tento postup má tu výhodu, že se nám tak podaří zvětšit znalostní bázi. Budeme mít více učebních dat. Tím se nám lépe podaří adaptovat síť na prostředí firmy.

Nasazujeme - li metodu odhadu ve firmě, pak většinou nemáme dostatek vhodných dat k naučení sítě. Během jejího používání by nám měla vznikat navržená znalostní báze. Tu navrhuji rozšiřovat vždy, když vyvineme nový softwarový produkt tak, že si zaznamenáme *CSP*, *FS* a kolik člověkodní, nebo člověkohodin tvorba softwarového produktu spotřebovala.

Když se tato báze rozroste o určité množství nových (například o pět) řádků, je vhodné síť znovu naučit. Tím vylepšit její předpovídací schopnost.

5.2.2 Učení sítě PETRA

Postup učení neuronové sítě uvádím na obrázu Obr.10: Učení sítě PETRA.



Obr. 8: Učení sítě PETRA [Pavlíček]

Máme - li připraven vhodný učební soubor, vytvoříme několik instancí sítě PETRA[18]. Na základě testů, které jsem provedl viz přílohy, navrhuji učit vždy pět sítí. Učit větší množství sítí již nemá význam. Každá síť se učí definovaným počtem učebních epoch. Počet epoch záleží především na různorodosti softwarových produktů, které jsou zaznamenány v učebním souboru. Čím rozmanitější jsou, tím více je potřeba učebních epoch.

Skončí - li všechny sítě proces učení, je třeba vybrat z nich tu nejlepší. Ne vždy se totiž podaří síť naučit (adaptovat ji na prostředí). Hodnoty vah sítě jsou na začátku učení voleny náhodně. To může mít za následek, že se jedna síť mnohem lépe učí dopovídat na předkládané učební vzory, než druhá. Proto, po proběhnutí všech

18 Většinou stačí pět instancí sítě PETRA

učebních epoch, může jedna síť vykazovat lepší výsledky na předkládané vzory, než síť druhá. Je tudíž vhodné učit větší množství sítí a pak vybrat tu s nejlepšími vlastnostmi.

Vybereme-li favoritní síť, uložíme její konfirguraci (hodnoty vah všech neuronů) do konfiguračního souboru a můžeme ji využívat k predikci složitosti.

5.3 Návrh metody rozpoznání klíčových faktorů složitosti

Expertní systém založený na bázi neuronových sítí přináší uživateli řadu možností. Jednou z nejzajímavějších je možnost rozpoznání klíčových faktorů realizace softwarového produktu. Do této kapitoly jsem hovořil o tom, že faktory složitosti *FS* nějak ovlivňují realizaci softwarového produktu. Tudíž ovlivňují jeho složitost. V některých případech je vhodné vědět, které faktory vznik softwarového produktu ovlivňují zásadně. V případě malých typů softwarových produktů, jako je návrh a implementace uživatelského rozhraní, tato vlastnost není potřeba. V případě velkého softwarového produktu, kdy do odhadu složitosti vstupuje velké množství případů užití a na celkové realizaci softwarového produktu se podílí řada vývojářských týmů, konzultantů atd, je tato vlastnost účelná.

Proto jsem pro neuronovou síť PETRA vytvořil funkci, pomocí které je možné jednotlivé faktory sledovat. Je-li faktor pro realizaci projektu klíčový, síť signalizuje jeho důležitost.

5.3.1 Rozpoznání klíčových faktorů složitosti – postup

Rozpoznávání klíčových faktorů složitosti je funkce, která umožní uživateli rozpoznat, který faktor se na realizaci softwarového produktu projevuje zásadním způsobem.

Postup je následující:

- Nejprve předložíme síti vstupní vektor obsahující (*ZSP,FS*) a provedeme odhad *CSP*.
- Proběhne-li odhad a je -li odhad rozumný (nevyjde nesmyslná hodnota), přistoupíme k druhému kroku. Zapneme automatickou funkci rozpoznávání klíčových faktorů složitosti, která je zabudována v nástroji PETRA. Dá se ale udělat i mechanicky. Navrhuji následující postup:
  - Každý prvek vstupního vektoru má svůj vlastní neuron ve vstupní vrstvě. Má - li vstupní vektor 30 položek, pak má neuronová síť PETRA na vstupu 30 vstupních neuronů. Každý vstupní neuron je spojen se všemi neurony ve skryté vrstvě. Každý spoj mezi neuronem vstupní a skryté vrstvy má určitou váhu. Pro každý prvek vstupního vektoru pak provedeme analýzu tak, že postupně měníme váhy na opačné (zachováme absolutní hodnotu váhy, ale změníme její znaménko) pro jeden prvek vstupního vektoru a sledujeme, zda se po změně vah nějak zásadně nezměnil výstup.
  - Zůstává - li hodnota výstupu podobná s původně provedeným odhadem, faktor se nijak zásadně neprojevuje na složitosti softwarového produktu.
  - Změní - li se hodnota výstupu, například změnou znaménka, nebo výrazně vroste či klesne (je - li původní odhad 300 člověkodní a hodnota se změní o 50 % nahoru či dolů), podílí se faktor na celkové složitosti softwarového produktu zásadně.

Tato funkce neříká, jakým způsobem se faktor na celkové složitosti softwarového produktu projevuje. Pouze na něj upozorňuje. Je na uživateli, jak s touto informací

naloží.

Tato funkce je výhodná pro odhalování nám skrytých vztahů, které mohou existovat například mezi týmy, vyvíjejícími nějakou společnou softwarovou komponentu. Teprve další analýza může odhalit, jak tento vztah vypadá. Přínosem této funkce je, že umožnuje poznat i ty vztahy, které nám nejsou na první pohled zřejmé.

5.4 Závěr kapitoly

Výsledky experimentů[19], které jsem provedl na základě metody odhadu složitosti, ukazují její obecnou použitelnost. Metoda sama je relativně jednoduchá. To mě vede k přesvědčení, že tato metoda má budoucnost. Dává zájemcům možnost odhadovat složitost vznikajícího softwaru ve vztahu na prostředí, kde je realizován.

Nová je myšlenka popsat vliv prostředí na vznikající software pomocí umělé inteligence. Tím se vyhýbám použití různých operací s hodnotami naměřenými v ordinální stupnici. Mimo to, znalost uložená v neuronech sítě PETRA, je využitelná i pro řadu jiných úkolů, než je jen predikce složitosti. Vytvoření znalostní báze pro naučení neuronové sítě PETRA není složité. Za dodržení předepsaných postupů může znalostní báze vznikat rychle. Tím, stejně jako v účetnictví firmy, je možné nalézt odpovědi na řadu nezodpovězených otázek. Proč se vývoj zdržuje? Co je klíčové pro tvorbu softwarového produktu atd.

Nevýhodou tohoto řešení je skutečnost, že nevíme, proč PETRA predikovala takový výsledek a ne jiný. Protože však PETRA nerozhoduje o životech lidí, ani o poškození nějakých dat, lze se s touto vlastností smířit.

19 Kapitola 6

6. Praktické ověření odhadu celkové složitosti softwarového produktu s využitím neuronové sítě PETRA

Tato kapitola je praktickým ověřením vlastností neuronové sítě PETRA. Popisuje její:

- Adaptaci na prostředí firmy.
- Prakticky ověřuje její schopnost odhadu celkové složitosti softwarového produktu.

6.1 Adaptace sítě PETRA v prostředí reálné firmy

Ačkoliv je využití neurovné sítě pro odhad *CSP* teoreticky možné, stál jsem před otázkou, jak se síť bude schopna adaptovat na reálné prostředí v reálné firmě. Proto jsem provedl řadu testů v prostředí firmy Sun Microsystems, které tuto schopnost měly ověřit.

Nejprve jsem vytvořil neuronovou síť PETRA. Ta měla na vstupu 30 vstupních neuronů, 60 neuronů ve skryté vrstvě a jeden výstupní neuron. Její počáteční váhy byly nastaveny náhodně. Vstup byl realizován vstupním vektorem o stejné velikosti – 30 položek.

Tato velikost odpovídá již dřívě navrženým velikostem pro:

- *ZSP* – 1 položka vektoru.
- $\vec{FSk}$ - 10 položek vektoru.
- $\vec{FSo}$ - 19 položek vektoru.

Data jsem získal z projektů realizovaných ve firmě Sun Microsystems. Popis projektů je volně dostupný na webové stránce http://ui.netbeans.org. Zde jsou uloženy projekty, které splňují podmínky pro:

- Výpočet *ZSP*.
- Vytvoření vektoru *FSk*.
- Vytvoření vektoru *Fso*.
- Je u nich uveden čas v člověkodnech, potřebný na jejich realizaci.

Specifikem daných projektů (na jejichž základě vznikal vždy nějaký softwarový produkt – v tomto případě používejme raději pojem projekt, než softwarový produkt) je, že byly provedeny v rámci jedné softwarové firmy Sun Microsystems. Byly realizovány na základě detailních funkčních specifikací, ve kterých byl kladen důraz na kvalitu popisu jednotlivých požadovaných funkčností budoucího softwarového produktu. Na vývoji se podíleli programátoři s patřičnými zkušenostmi.

Sebraná data jsem nijak nefiltroval a do učebního souboru jsem vložil i takové softwarové produkty, které se nějak odlišovaly od ostatních. Například velkým počtem člověkodní potřebných na jejich realizaci, nebo velkým množstvím autorů funkčních specifikací.

Je nutné uvést, že veškeré softwarové produkty (byly zaměřeny především na tvorbu nových uživatelských rozhraní pro nástroj NetBeans) vznikaly paralelně. Na jejich specifikaci i realizaci se podíleli autoři vždy jen určitým procentem své denní kapacity. Nebylo však možné nijak přesně postihnout, kolik hodin přesně softwarový produkt spotřeboval, protože jsem neměl k dispozici přesný rozpis prací, na kterých dotyčný pracovník pracoval, během projektu. Musel jsem se s paralelismem při tvorbě softwarového produktu smířit. Ukázalo se, že to není na škodu, protože tento jev je reálný.

Z realizovaných softwarových produktů, které byly realizovány v rámci ucelených projektů, jsem vytvořil učební soubor[20]. Na jeho základě jsem testoval schopnost neuronové sítě PETRA se adaptovat na prostředí dané firmy V rámci testování byly provedeny testy vždy s pěti neuronovými sítěmi.

Jak jsem již uvedl, neuronové sítě na začátku učení měly nastaveny vnitřní váhy jednotlivých neuronů náhodně. Teoreticky se mohlo stát, že jedna síť se naučí adaptovat na prostředí mnohem lépe než síť jiná. Proto jsem při testech testoval vždy pět neuronových sítí. Snažil jsem se ověřit, zda se všechny sítě dostatečně přesně naučí rozpoznávat předložené vzory (adaptovat se na prostředí). Zajímalo mě, jaké budou rozdíly v přesnosti odpovědí jednotlivých sítí.

20 Vzor učebního souboru je uveden v příloze.

6.1.1 Detailní popis adaptace sítě PETRA

Adaptace sítě proběhla s násedující konfigurací sítě PETRA:

1. 30 vstupních neuronů,
2. 60 neuronů ve skryté vrstvě,
3. s jedním výstupním neuronem,
4. s náhodně nastavenými počátečními váhami neuronů,
5. v testu bylo paralelně učeno pět sítí.

Sítě byly učeny metodou zpětného šíření chyby:

1. učební soubor měl 21 řádků (obsahoval 21 realizovaných softwarových produktů, které byly popsány formou projektové dokumentace),
2. sítě se učily pomocí 100 000 učebních epoch,
3. proces učení probíhal dokud dokud:
   - změna vah neuronů sítí byla v absolutní hodnotě větší než 0,001,
   - nebo dokud nebyl vyčerpán počet učebních epoch (to se ukázalo jako zbytečné. Sítě se vždy zastavily dříve než po 100 000 epochách).

V okamžiku, kdy všechny sítě byly naučeny, jsem provedl test. Všem sítím jsem předložil všechny naučené projekty. Sledoval jsem, jak přesně je každá síť schopna odpovědět na předložený vzor. Výsledky jsem porovnal a uložil do příslušných tabulek. V této práci uvádím jen podstatné množství základních tabulek a grafů popisujících schopnost adaptace sítí.

Prvním testem, pomocí kterého jsem se snažil rozpoznat adaptaci sítí na prostředí firmy Sun Microsystems, bylo zjištění odchylek jednotlivých sítí při, předložení vzorů, které se sítě měly naučit. Sítě obdržely soubor 21 vstupních vektorů představujících jednotlivé softwarové produkty a jejich úkolem bylo odhadnout čas potřebný na jejich realizaci v člověkodnech.

V prvním sloupci Tabulky 11 je seznam softwarových produktů realizovaných ve firmě Sun Mircrosystems. V sloupcích Odchylka sítě 1 až Odchylka sítě 5 jsou pak uvedeny odchylky od skutečného počtu člověkodní v absolutní hodnotě.

Soft.produkt	Odchylka sítě 1	Odchylka sítě 2	Odchylka sítě 3	Odchylka sítě 4	Odchylka sítě 5
1	1	0	1	1	1
2	0	0	0	0	0
3	0	0	0	0	0
4	0	0	0	0	0
5	0	0	0	0	0
6	0	1	1	1	1
7	0	1	1	1	1
8	0	2	0	1	3
9	0	1	0	1	1
10	17	16	16	16	14
11	3	2	3	2	0
12	3	3	3	1	1
13	3	2	2	1	1
14	2	1	1	1	1
15	7	7	7	8	10
16	0	0	1	1	1
17	4	4	4	2	2
18	1	1	1	0	0
19	3	3	3	5	7
20	1	1	1	1	2
21	1	0	0	1	3
Průměrná odchylka	2,19	2,14	2,14	2,1	2,33
Medián	1	1	1	1	1

Tabulka 11: Odchylky sítí od požadovaných hodnot

Z hodnot zobrazených v Tabulce 11 je vidět, že přesnost odhadu člověkodní pro jednotlivé softwarové produkty kolísala v rozmezí od 0 do 17 člověkodnů.

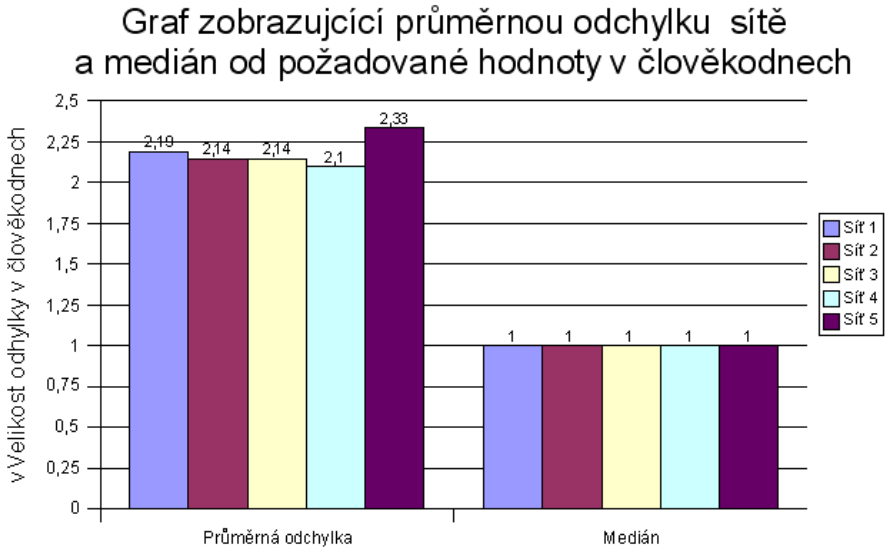
Průměrná odchylka sítí v člověkodnech je zobrazena na předposledním řádku Tabulky 11. Medián odhadu všech sítí v člověkodnech je roven číslu 1 a je zobrazen v posledním řádku tabulky.

Zajímavý je řádek 10, ve kterém všechny sítě chybovaly. Jedná se o realizovaný softwarový produkt, který byl realizován za obdobných podmínek, jako ostatní produkty, ale jeho doba realizace byla mnohonásobně vyšší, než u produktů

podobných. Tento produkt bylo samozřejmě možno ze souboru odstranit a síť na něj neadaptovat. Protože však jednotlivé vztahy mezi faktory složitosti *FS* (vždyť jich je 30 a mohou se ovlivňovat každý s každým) nejsou známy, rozhodl jsem se tento softwarový produkt v modelu odhadu zachovat.

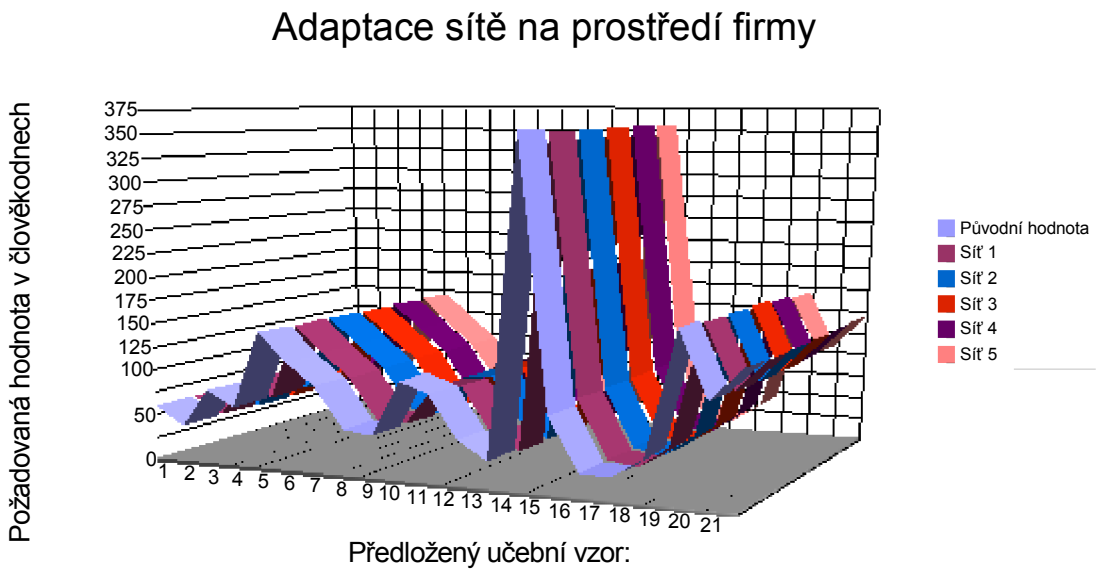
Sítě, i přes to, že počet člověkodní na realizaci softwarového produktu 10 byl zcela netypický, odhadly potřebu času na jeho realizaci relativně přesně. Průměrná odchylka sítí v člověkodnech se pohybovala kolem 2,1 až 2,3 člověkodne na softwarový produkt. Což v praxi znamená dva dny na softwarový produkt (jehož průměrná doba realizace, ve které není započten extrémně odlišný softwarový produkt 10, je kolem 100 člověkodní).

V grafu Graf 1 je možné vidět grafické zobrazení jednotlivých průměrných odchylek sítí v člověkodnech (sloupce vlevo) a medián sítí v člověkodnech (sloupec vpravo).



Graf 1: Průměrná odchylka a medián odchylky sítí od požadovaných hodnot

V grafu Graf 2 vidíme přesnost adaptace jednotlivých sítí na prostředí firmy. Na ose *Y* je zobrazena požadovaná odpověď v člověkodnech a na ose *X* jsou zobrazeny jednotlivé softwarové produkty. První křivka představuje skutečné hodnoty v člověkodnech pro jednotlivé softwarové produkty a ostatní křivky zobrazují, jak přesně jednotlivé sítě odpovídaly na jednotlivé předložené vzory.



Graf 2: Adaptace sítí na prostředí firmy

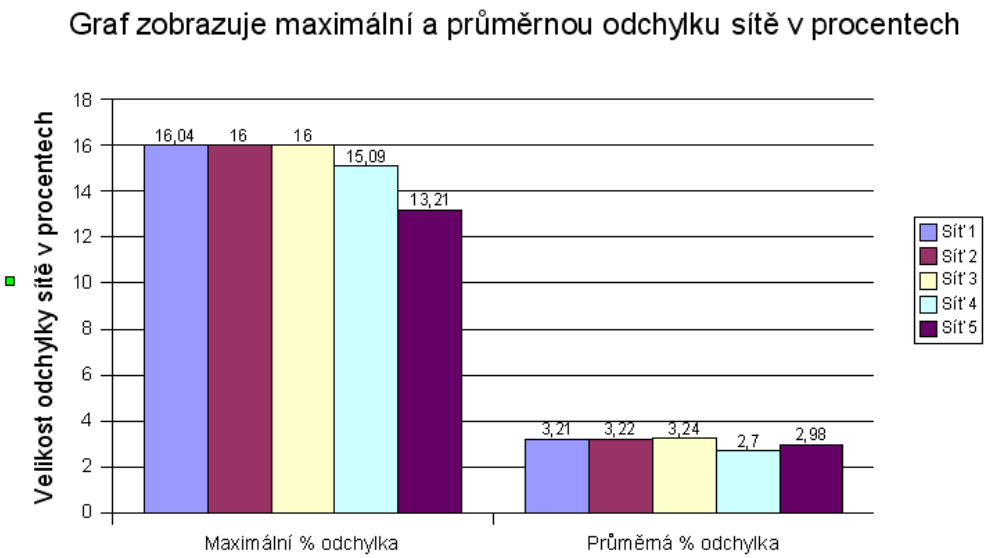
Z grafu je patrné, že se sítě uměly přizpůsobit prostředí firmy s minimální odchylkou.

Na základě odchylek odpovědí jednotlivých sítí v člověkodnech byla vypočtena maximální a průměrná procentuelní odchylka sítí. Je možno ji vidět vypočtenu pro každou síť v následující Tabulce 12. Sloupce Síť 1 až Síť 5 znázorňují jednotlivé sítě. V prvním řádku je uvedena maximální ochylka od požadované hodnoty v procentech. V druhém řádku je zobrazena průměrná odchylka sítí v procentech.

	Síť 1	Síť 2	Síť 3	Síť 4	Síť 5
Maximální % odchylka	16,04	16	16	15,09	13,21
Průměrná % odchylka	3,21	3,22	3,24	2,7	2,98

Tabulka 12: Maximální a průměrná procentuelní odchylka sítě

Graf 2 zobrazuje maximální a průměrnou odchylku jednotlivých sítí v procentech. Na ose *Y* jsou zobrazena procenta. Na ose *X* jsou umístěny jednotlivé výsledky sítí. Vlevo je maximální odchylka sítí, vpravo průměrná odchylka sítí v procentech.



Graf 2: Zobrazuje maximální a průměrnou odchylku jednotlivých sítí v procentech

6.1.2 Závěr kapitoly

Z provedených testů je patrné, že se neuronová síť PETRA adaptovala na prostředí firmy a na předložené projekty s maximální procentuelní odchylkou 16 %. Průměrná procentuelní odchylka činila zhruba 3,2 %. Pro provedení experimentu se tedy prokázalo, že výběr a volba parametrů byla pro adaptaci sítě postačující.

Experimenty též ukázaly, že mezi jednotlivými sítěmi byl minimální rozdíl v jejich schopnosti adaptovat se na prostředí firmy.

6.2 Odhad celkové složitosti projektu s využitím neuronové sítě PETRA v prostředí reálné firmy

Druhým krokem pro uznání metody odhadu *CSP* za použitelnou je zjistit, jak přesně je PETRA schopna, na základě naučení, odhadovat složitosti projektů. Z tohoto důvodu jsem upravil učební soubor použitý při předchozím testu. Z učebního souboru jsem vybral čtyři projekty, pro které jsem se rozhodl odhadovat *CSP*. Každý projekt byl realizován jiným autorem. Podmínkou pro odhad složitosti bylo, aby v učebním souboru sítě byl vždy alespoň jeden projekt realizován autorem, pro kterého je složitost odhadována.

To znamená:

- Autor 1, pro kterého byla predikována složitost v člověkodnech musel mít 1 nebo více realizovaných softwarových produktů v učebním souboru.

Projekty, pro které byla odhadována složitost byly vždy rozdílné od projektů v učebním souboru. Nebyly identické.

Opět bylo učeno pět sítí. Každá síť pak odhadovala složitost v člověkodnech pro čtyři různé softwarové produkty. V Tabulce 13 vidíme průměrnou odchylku odhadu složitosti jednotlivých sítí v člověkodnech od skutečné složitosti v člověkodnech a medián odchylky v člověkodnech.

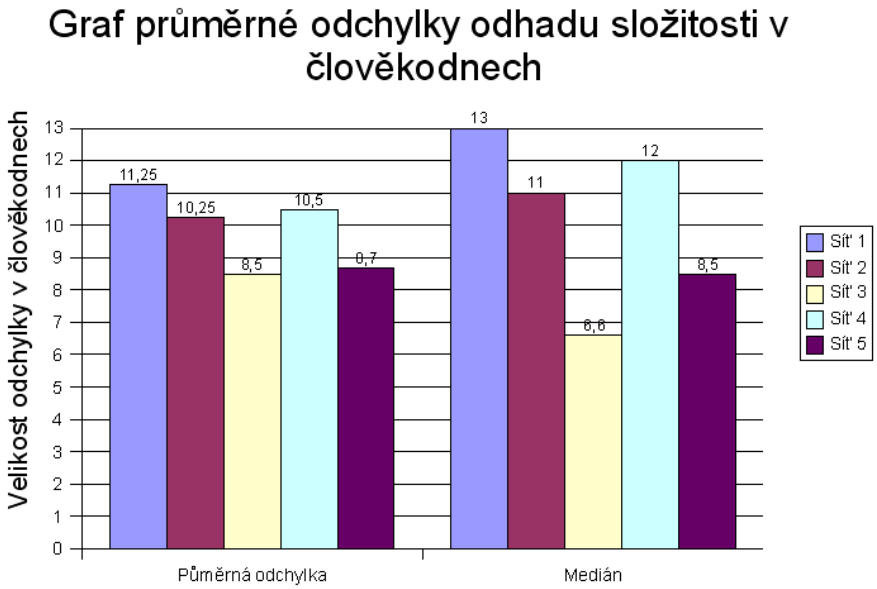
Soft. Produkt	Odchylka 1	Odchylka 2	Odchylka 3	Odchylka 4	Odchylka 5
1	13	13	20	14	1
2	13	9	3	10	2
3	16	16	10	15	3
4	3	3	1	3	4
Průměrná ochylka	11,25	10,25	8,5	10,5	2,5
Medián	13	11	6,5	12	2,5

Tabulka 13: Odchylka sítí při predikci CSP pro nezámé softwarové produkty

V sloupcích Odchylka 1 až Odchylka 5 vidíme odchylky v odhadu složitosti sítí v člověkodnech pro 4 neznámé softwarové produkty. V řádku průměrná odchylka

vidíme průměrnou odchylku sítí v člověkodnech pro všechny čtyři softwarové produkty. V poslední řádce je zobrazen medián odchylek jednotlivých sítí v člověkodnech.

Graf 4 zobrazuje průměrnou odchylku sítí při odhadu složitosti v člověkodnech (vlevo) a medián odchylek v člověkodnech (vpravo).



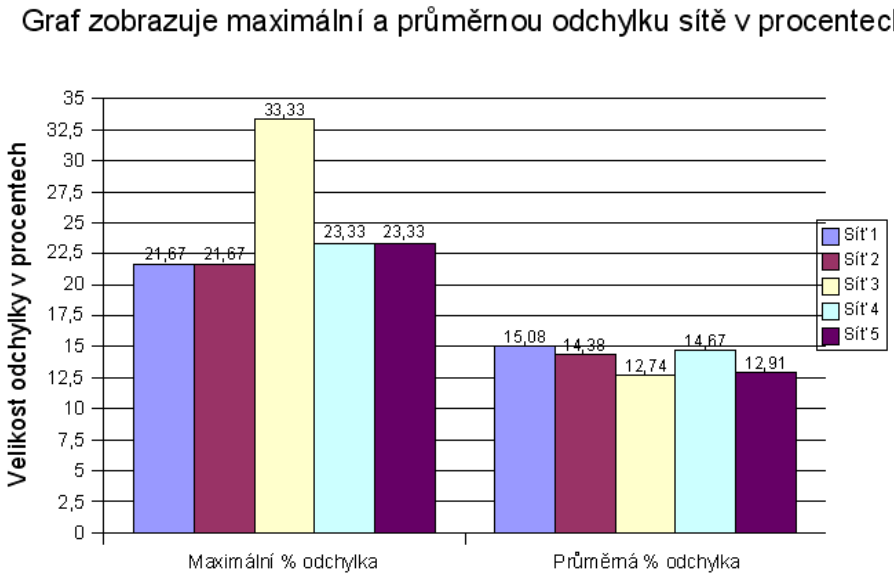
Graf 4: Graf jednotlivých odchylek sítí v člověkodnech

Na základě naměřených hodnot jsem vypočítal maximální a průměrnou procentuelní odchylku sítí. Tabulka 14 zobrazuje Maximální a průměrnou odchylku sítí 1 až 5 v procentech.

	Síť 1	Síť 2	Síť 3	Síť 4	Síť 5
Maximální % odchylka	21,67	21,67	33,33	23,33	23,33
Průměrná % odchylka	15,08	14,38	12,74	14,67	12,91

Tabulka 14: Maximálni a průměrná odchylka sítí v procentech

Graf 5 zobrazuje maximální (levá část grafu) a průměrnou procentulení (vpravo) odchylku odhadu složitosti. Na ose *Y* jsou uvedena procenta odchylky, na ose *X* jsou uvedeny jednotlivé odchylky sítí v procentech.



Graf 5: Graf maximální a průměrné procentuelní odchylky od požadované hodnoty

6.2.3 Závěr kapitoly

Neuronová síť PETRA byla schopna predikovat *CSP* s 15 % průměrnou odchylkou. Při tom největší odchylka činila 33 %. Experimenty dokázaly, že neuronová síť pro odhad složitosti software s využitím předepsaných faktorů složitosti je použitelným nástrojem.

6.3 Závěr kapitol

Výsledky získané experimentem ukázaly, že průměrná odchylka sítí je srovnatelná s odchylkou metod COCOMO nebo FP, u kterých literatura uvádí odchylku přesnosti odhadu v rozmezí 20 až 30 %.

Výsledky experimentu jsou relativně přesné a ukazují, že se sítě byly schopny adaptovat na prostředí firmy relativně přesně - 3 % odchylka.

Výsledky odhadu *CSP* jsou velmi přesné - 15 %. V budoucnu bude vhodné odzkoušet schopnost odhadu sítí na větším vzorku softwarových projektů. Předložený vzorek, byť se jednalo o realné projekty, byl velmi specifický. Firma Sun Microsystems má velmi přesně řízený vývoj softwarových komponent. To většinou neplatí u běžných softwarových firem. Mimo to, vzorek zahrnoval vývoj softwarových komponent pro prostředí NetBeans. Ty jsou samy o sobě specifické a jistě jejich složitost vývoje je mnohem menší, než tvorba velkých informačních systémů. Data o takových projektech však nejsou volně k dispozici. Proto v současné době využívám dat naměřených ve firmě IBM. Získaná data nemohu volně uvádět v disertační práci.

Studiem možností odhadu složitosti softwarových produktů s využitím umělé inteligence se budu zabývat i v budoucnosti. O výsledcích své práce budu informovat veřejnost formou vědeckých konferencí a odborných publikacích.

7 Závěr práce

Každá firma, stojící před úkolem vyvinout „nějaký“ softwarový produkt, řeší otázku, jak dlouho bude daný softwarový produkt vyvíjet a kolik bude třeba vynaložit nákladů na jeho realizaci. Na základě této znalosti, která je základní manažerskou charatkeristikou vývoje IS, se může vedení firmy rozhodnout zda požadovaný softwarový produkt realizovat či nikoliv.

Odhadnout potřebné náklady na požadovaný software však vyžaduje mít k odhadu dostatečné množství informací. To ale v době rozhodování zda softwarový produkt vyvíjet či nikoliv chybí.

V současné době neexistuje vhodná metoda, která by na základě požadavků na software umožnila odhadovat složitost požadovaného softwarového produktu.

Relativně přesné odhady složitosti software jsou založeny na počtu řádků zdrojového kódu (COCOMO viz kapitola 2.1). Tato data jsou známa ale již v okamžiku, kdy je softwarový produkt již téměř hotov. To je zpravidla pozdě.

Ostatní metody, jako je například metoda UseCase Points (viz kapitola 2.3), provádějí odhady složitosti na základě hodnot naměřených pomocí ordinálních stupnic a hrubě porušují pravidla teorie měření. Ukázal jsem, že možnost interpretace výsledků výpočtů, jejichž vstupem jsou míry získané měřením v měřících stupnicích ordinálního typu, je problematická. Z teorie měření plyne, že tyto výsledky nejsou obecně invariantní vzhledem k volbě měření ordinálního typu.

7.1 Shrnutí hlavního cíle disertační práce

Za účelem odhadu složitosti softwarového produktu *CPS,* který navrhuji považovat za **základní manažerskou charateristiku vývoje informačních systémů** jsem sestrojil novou metodu odhadu. Ta využívá alternativní přístup, kde se vliv faktorů *FS* ovlivňujících složitost softwarového produktu, které navrhuji považovat za **manažerské charateristiky složitosti vývoje IS,** nestanovuje předem, ale je svěřen neuronové síti.

Síť jsem sestrojil a provedl testování odhadu. Testování prokázalo (viz kapitola 6, přílohy), že dává dostatečně přesné výsledky pro odhad složitosti softwarového produktu.

Hlavním výsledkem mé práce je následující **seznam manažerských charakteristik vývoje IS**:

- *CSP* - Základní manažerská charateristika vývoje IS,
- *FS* - Manažerské charakteristiky složitosti vývoje IS,
- *FSk* - Manažerské charakteristiky zadání požadavků na IS,
- *Fso* - Manažerské charakteristiky prostředí vývoje IS.

A **jejich odhad**, který je prováděn:

- Pomocí **nové metody odhadu složitosti**, kterou jsem za tímto účelem sestrojil,
- **Ve fázi Rozpracování požadavků na software**, která je popsána metodou Unified Process,
- **V etapě specifikace požadavků na software**, která je po obchodním modelování druhou etapou vývoje informačních systémů podle metody Unified Process.

Metoda odhadu složitosti, i manažerské charateristiky jsou nové a nebyly doposavad nikde publikovány.

7.2 Shrnutí dílčích cíle disertační práce

Dílčím cílem disertační práce bylo navrhnout metodu pro rozpoznání klíčových faktorů, které se zásadním způsobem podílejí na složitosti řešení. Ty navrhuji považovat za **manažerské charatkeristiky složitosti vývoje IS**. Jejich rozpoznání provádím na základě adaptace neuronové sítě PETRA, kterou jsem sestrojil za účelem odhadu *CSP*. Tohoto cíle se mi podařilo dosáhnout a podrobně jej popisuji v kapitole 5.3 Rozpoznání klíčových faktorů složitosti.

Dalším dílčím cílem disertační práce bylo využití inovativních prostředků pro odhad složitosti. Tento cíl se mi podařilo dosáhnout. Pro odhad složitosti *CSP,* který navrhuji považovat za **základní manažerskou charakteristiku vývoje IS***,* jsem sestrojil neuronovou síť PETRA (PErcepTRon Artificial intelligence). Podrobně ji popisuji v kapitole 5. PETRA.

Posledním dílčím cílem disertační práce bylo navrhout vhodnou architekturu neuronové sítě pro daný účel a dát jí k dispozici široké veřejnosti. Tento cíl se mi podařilo dosáhnout téměř úplně. Neuronovou síť s vhodnou architekturou jsem navrhl, předvedl její využití pro účel odhadu. Veřejnost na základě informací uvedených kapitole 5. PETRA může odhad složitosti *CSP* provádět. Nepodařilo se mi síť dát veřejnosti k dispozici jako ucelenou softwarovou komponentu, která by byla naprogramována a byla volně dostupná. Na dokončení tohoto cíle pracuji ve firmě Sun Microsystems. O výsledcích mé práce budu veřejnost informovat na vědeckých konferencích a ve vědeckých publikacích.

7.3 Závěrečné poděkování

Na závěr si dovoluji poděkovat:

- Prof. RNDr. Jiřímu Vaníčkovi, CSc., za vedení disertační práce a za vědomosti, které mi během pětiletého studia na Katedře informačního inženýrství předal, jako můj školitel a kamarád.
- Doc. Arnoštu Veselému, CSc., za odborné konzultace při návrhu neuronové sítě PETRA.
- Doc. Vojtěchu Merunkovi, PhD., za odborné konzultace a pomoci při studiu metod řízení vývoje informačních systémů.
- Katedře informačního inženýrsví České zemědělské univerzity v Praze.
- Zuzaně Husákové za obětavou pomoc při odevzdávání disertační práce.
- Firmě Sun Microsystems za pomoc a pochopení.
- Mé manželce Petře, za pomoc a inspiraci.
- Přírodě za existenci kofeinu, bez jehož pomoci bych asi disertační práci nedokončil.

8 Použitá litertura

[1] Arlow J., Neustat I., UML a unifikovaný vývoj aplikací, Brno, Computer Press, 2003, ISBN 80-7226-947-X.

[2] Česká technická norma ČSN EN ISO 9000, Systémy managementu jakosti, 2001.

[3] Vaníček J., Měření a hodnocení jakosti informačních systémů, Praha, ČZU PEF, 2., přepracované vydání, ISBN 80-213-1206-8.

[4] Booch G., ,Jacobson I., Rumbaugh J., The Unified Modeling Language User Guide,Addison-Wesley 1998, ISBN 0201571684.

[5] Booch G., ,Jacobson I., Rumbaugh J., The Unified Modeling Language Reference Manual,Addison-Wesley 1998, ISBN 020-130-998-X.

[6] Anthony J., Simons H., 30 Things that go wrong in object modelling with UML 1.3 University of Sheffield, UK Ian Graham,Ian Graham Associates.

[7] Jacobson I.,Booch G., Rumbaugh J., Unified Software Development Process, ISBN 020-1571-692; Published: Feb 4, 1999; Copyright 1999.

[8] Merunka V., Objektový přístup a UML v návrhu informačního systému, konference Objekty 2004.

[9] Garmus D.,Herron D., Function Point Analysis: Measurement Practices for Successful Software Projects, Addison Wesley Professional, Published: Nov 16, 2000; Copyright 2001, ISBN 0201699443.

[10] Harrison D.,TOGAF, http://www.developer.com/design/article.php/10925_3374171_3, 10.9.2006.

[11]Garmus D.,Herron D., Function Point Analysis: Measurement Practices for Successful Software Projects, Addison Wesley Professional, ISBN 0201699443; Published: Nov 16, 2000; Copyright 2001.

[12] ISO/IEC JTC1: http://www.jtc1.org/,10.9.2006.

[13]Harrison D., www.tcrug.org/Documents/PPM02%20final.ppt,10.9.2006.

[14] Keller R., An Introduction to Function Point Analysis,http://www.qpmg.com/fp-intro.htm, 10.9.2006.

[15] Lipton D., Function Points and the SEI Capability Maturity Model, http://www.qpmg.com/seicmm2.htm, 10.9.2006.

[16]Enterprise Unified Project, http://www.enterpriseunifiedprocess.com/, Ronin International, Inc., 10.9.2006.

[17] Object Management Group, http://OMG.ORG, 10.9.2006.

[18] ETSI EG 201 872 V1.2.1,Methodological approach to the use of object-orientation in the standards making process, 10.9.2006.

[19] Coleman D., The Fusion Method, Prentice Hall; 1st edition (September 24, 1993), ISBN – 0133388239.

[20] Booch G., Object-Oriented Analysis and Design with Applications,Addison-Wesley Professional; 2 edition (September 30, 1993), ISBN 0805353402.

[21] Rumbaugh J., Blaha M., Lorenzem W., Eddy F., Premerlani W., Object-Oriented Modeling and Design, Prentice Hall; 1st edition (October 1, 1990), ISBN 0136298419.

[22] I. Jacobson I., Object Oriented Software Engineering: A Use Case Driven Approach,Published by Addison Wesley Professional, ISBN 0201544350.

[23] Merunka V, Polák J., Carda A., Umění systémového návrhu, Grada 2003, ISBN 80-247-0424-X.

[24] Karner, G, 1993, "Metrics for Objectory". Diploma thesis, University of Linköping, Sweden. No. LiTHIDA - Ex-9344:21. December 1993.

[25] http://www.idi.ntnu.no/emner/tdt4290/docs/faglig/uml2001-anda.pdf,8.3.2006, Estimating Software Development Effort based on Use Cases – Experiences from Industry.

[26] Novák a koletiv: Umělé neuronové sítě. C.H. Beck 1998.

[27] Dr. Mong Suan Yee,Radial basis function network based burst-by-burst adaptive transceivers http://www-mobile.ecs.soton.ac.uk/comms/Res_Int_transceiver.htm, 8.3.2006.

[28] Obrázek neuronu, copyright – fig.cox.miami.edu, http://fig.cox.miami.edu/~cmallery/150/neuro/neurophysiology.htm, 8.3.2006.

[29] Berka P.,, Dobývání znalostí z databází, Adastra s.r.o., 2003, ISBN 80-200-1062-9.

[30] Veselý A., – Artificial Inteligence – Hand out.

[31] Kotek Z., Chalupa V., Brůha I., Jelínek J., Adaptivní učící se systémy. SNTL, Praha 1980.

[32] Server firmy Sun Microsystems, pod sekce Human Interface Desing, ui.netbeans.org, 10.9.2006.

[33] Radecký M.,Vondrák I., Štolfa S, Byznis modelování a jeho možnosti využití v praxi, http://mr.hyperlink.cz/files/radecky_technologie04.pdf,10.9.2006.

[34] Software Cost Estimation with Cocomo II (with CD-ROM) by Barry W.Boehm, Ellis Horowitz, Ray Madachy, and Donald Reifer (Hardcover - Jan 15, 2000), ISBN: 0130266922.

[35] A. Albrecht. Measuring application development productivity. in Proc. Joint SHARE/GUIDE/IBM Applications Development Symposium, Monterey, CA, 1979.

[36] http://www.ifpug.org/, 8.3.2006.

[37] Struska Z., Metody odhadu složitosti – Feature Point a Funtion Point, sborník konference Objekty 2005.

[38] Chidamber, S.R. And Kemerer, Ch. F.: A Metric Suite for Object Oriented Design, IEEE Transaction on Software Engineering, Vol. SE – 14, No. 9. pp 1366 – 1372, 1988.

[39] Brooks F.: The Mythical Man-Month: Essays on Software Engineering,ISBN 0-201-83595-9, 1995.

[40] Java.Sun.com,http://java.sun.com/,10.9.2006.

9 Přehled publikovaných prací

[A.1] Pavlíček J., Nevřivová P., Beránková M., Modelování integrovaných podnikatelských subjektů v zemědělství in MVD2002 - Ekonomika a manažment podnikov v procese globalizacie, IV zvatok, 1.diel, Ekonomika, ISBN 80-8069-030-8.

[A.2] Pavlíček J., Běžné problémy s odhadem a měřením pracnosti vývoje programů, Brno 2001, ISBN 80-7302-022-X.

[A.3]Pavlíček J., Vaníček J., Nevřivová P., Opomíjené problémy při zjišťování a měření efektivnosti výuky. Pedagogický SW 2002 - Sborník příspěvků a programů, ISBN 80-85645-46-7.

[A.4] Pavlíček J., Ekonomické a jiné výhody využití webových portálů, Praha 2002 – SunNew Sun Microsystems.

[A.5] Pavlíček J., Nevřivová P., Objekty ve službách outsorcingu IS in Objekty 2002 - sborník 7.ročníku celostátní odborné konference, ISBN 80-213-0947-4.

[A.6] Pavlíček J., Beránková M., Nevřivová P., Patka J., Dömeová L., ADJUSTABLE ENVIRONMENT FOR CREATIVE TOOL IN SIMULATION LEARNING in Information and Communication Technology in Education Procedings, ISBN 80-7042-888-0.

[A.7] Pavlíček J., Merunka V., Nevřivová P., The development of the information systems in Proceedings of the International Symposium on Information and Communication Technologies, ISBN 0-9544145-2-7.

[A.8] Pavlíček J., Nevřivová P., Beránková M., VÝUKOVÉ SYSTÉMY VE VIRTUÁLNÍM PROSTŘEDÍ in Informace na dlani, Inforum 2003, ISSN 1214-1429.

[A.9] Pavlíček J., Nevřivová P., The development methodologies of the information systems in Sborník prací z mezinárodní vědecké konference Agrární perspektivy XII. Nová ekonomika a rozšíření EU - Díl II., ISBN 80-213-1056-1.

[A.10] Beránková M.,Nevřivová P.,Patka J.,Pavlíček J., Učební pomůcka pro výuku simulací, in eLearning ve vysokoškolském vzdělávání 2003, ISBN 80-7318-138-X.

[A.11] Beránková M.,Nevřivová P., Pavlíček J., Výuka simulací na Provozně ekonomické fakultě in MendelNET 2003 Sborník příspěvků z konference studentů doktorského studia, ISBN 80-7157-719-7.

[A.12] Pavlíček J., Nevřivová P., The Electronic Systems for education for students (The basic problems with the implementation) in Computer Based Learning in Science, Conference proceedings 2003, Volume I, ISBN 9963-8525-1-3.

[A.13] Nevřivová P., Beránková M., Pavlíček J., SIMULATION AS IMPORTANT TOOL FOR DECISION MAKING – IN EDUCATIONAL PROCESS in Proceeding of the Winter International Symposium on Information and Communication Technologies, ISBN 0-9544145-3-5.

[A.14] Nevřivová P., Pavlíček J., P.: Grafická tvorba simulací in Sborník příspěvků z doktorandského semináře, ČZU, Praha, 2004, ISBN 80-213-1150-9.

[A.15] Beránková, M., Nevřivová, P., Pavlíček, J.: ICT podpora výuky simulací: model zásob, Proceedings of papers, Efficiency and Responsibility in Education, 2004, Praha, ISBN 80-213-1175-4.

[A.16] Beránková M., Domeová, L., Nevřivová, P.,Pavlíček, J.: Significant Topics for Decision Making: Operational Research and ICT Support in Information Systems, Technologies and Applications, Volume I, Proceedings 2004, Orlando, ISBN 980-6560-13-2.

[A.17] Beránková, M., Nevřivová, P., Pavlíček, J.: Simulation of Inventory Systems: Cognitive Context in Information and Communication Technology in Education Proceedings, 2004, Ostrava,ISBN 80-7042-993-3.

[A.18] Pavlíček, J., Franc J., Nevřivoá P., Sun Profiler – Uživatelsky zaměřený přístup -k tvorbě nových softwarových produktů, Objekty 2004, ISBN 80-248-0672-X

[A.19] Pavlíček, J., Franc J., Nevřivoá P., Sun Profiler – Uživatelsky zaměřený přístup -k tvorbě nových softwarových produktů, Objekty 2004, ISBN 80-248-0672-X

10 Přílohy

Kapitola přílohy obsahuje vzor učebního soubrou neuronové sítě PETRA.