Softwarová architektura

(Nezúžený "klasický" úvod)

Tomáš Smolík
Profinit, s.r.o.
tomas.smolik@profinit.eu
http://www.profinit.eu

Obsah

1. Úvod

Tento text pojednává o velmi důležitém aspektu návrhu softwarového systému a to o softwarové architektuře. Velmi zjednodušeně řečeno, softwarová architektura se zabývá[1] problematikou struktury a organizace softwarového systému, tedy jeho architekturou. Velmi zjednodušeně lze říci, že s rostoucí velikostí a složitostí softwarových systémů se návrh a specifikace celkové struktury systému stává významnějším problémem, než volba algoritmů a datových struktur[2] [Shaw94].

Z historických důvodů je tento text koncipován tak, aby kromě vlastního výkladu poskytl výklad dostatečný pro splnění nároků běžných standardů pro proces vývoje software. Např. v standardu ISO 9000-3:1991 [ISO9000-3] jsou to klauzule 5.6.1 a 5.6.2; přesně řečeno tento text odpovídá duchu[3] těchto nároků. Na druhou stranu tento text avšak není vyčerpávajícím pojednáním o problematice softwarové architektury. Je zde však poskytnut dostatek referencí[4] pro další studium.

1.1 Příčina zájmu o softwarovou architekturu

Teoretické práce, studie a zkušenosti ukázaly, že struktura systému, resp. jeho architektura, je velmi významná. Uveďme si pár příkladů. Clements shrnuje odkaz prací Parnase [Parnas72, Parnas74, Parnas76, Parnas79] a Dijkstry [Dijkstra68] na poli zájmu o strukturální otázky softwarových systémů takto: „Struktura je důležitá a dosažení správné struktury přináší užitek“ [Clements96]. Kazman v technické zprávě zabývající se odvozením softwarové architektury z nároků na kvalitativní atributy softwarového systému uvádí: „ ... čím větší je systém, tím více dosažení nefunkčních kvalitativních nároků záleží na softwarové architektuře systému“ [Kazman94]. Dále Clements tvrdí: „Volba vhodné struktury s vhodnými koordinačními mechanismy mezi strukturálními částmi umožňuje ekonomickou produkci bez obětování nároků na výkon či korektnost“ [Clements96b].

Je tedy třeba před vlastním návrhem stanovit nároky na výslednou architekturu softwarového systému. Stanovení nároků na architekturu musí být činěno s vědomím všech důsledků pro budoucí život navrhovaného systému. Nároky na architekturu musí být dokumentovány a tvoří jeden z nejzávažnějších dokumentů produktu[5]. Zde poznamenejme, že stanovení nároků a kriterií na strukturu programového systému vlastně znamená jisté vědomé dobrovolné omezení. Omezení jinak téměř neomezených možností, jak realizovat celou řadu momentů při návrhu a implementaci programového systému. Zde je dobré si uvědomit paralelu s tím, co znamenají koncepty strukturovaného programováním v oblasti programování v malém. Na tento fakt nap ř. upozorňuje [Lane90].

Vědomé stanovení a nakládání s nároky na architekturu vyvíjeného systému a jejich dokumentace umožňuje (následuje jakýsi úvodní přehled, problematika je podrobněji diskutována v následujících částech textu):

- Dosáhnout architektonickou jednotnost (a pokud možno čistotu) v průběhu vývoje a následné údržby softwarového systému: Toto má obrovský dopad jak na jakost těchto procesů, tak na jejich výsledek. Dále je zde nezanedbatelný dopad na náročnost procesu údržby z hlediska zdrojů[6].
- Příslušné zohlednění požadovaných nároků na vyvíjený softwarový systém: Je třeba si uvědomit, že dobrá softwarová architektura sice automaticky nezaručuje jakost výsledného softwarového produktu, ale špatná[7] může být překážkou. Proto dobrá softwarová architektura je nutná, avšak je třeba si být vědom, že to není podmínka postačující.
- Usnadnění distribuce vývojového procesu, samostatného vývoje komponent, přebírání komponent splňujících příslušné definované strukturální nároky: Zde se tedy dostáváme do oblasti tzv. vývoje založeného na komponentách („Component-based development“, resp. „Component-oriented development“ [Clements96b, Nierstrasz92]), do oblasti znovupoužívání, do oblasti vývoje produkčních řad, atd.

[1] Tento zájem jde na úroveň jednotek dekompozice softwarového systému na nejjemnější úrovni granularity, které jsou uvažovány při designu (úroveň abstrakce modelování designu).
[2] Tvrzení však v žádném případě neříká, že toto není vážný problém.
[3] Zde je míněn ten fakt, že norma ISO 9001:1994, resp. ISO 9000-3:1991 je velmi stručná a relativně málo detailní. Z toho plyne nutnost aplikovat ducha a nejen literu těchto norem. Na tento fakt je např. upozorňováno ve srovnávacích studiích CMM s ISO 9001 [Paulk94, Paulk95b].
[4] Ne všechny reference jsou ve vlastním textu referovány.
[5] Boehm píše: „Pokud projekt nedosáhl architektury systému obsahující její zdůvodnění, projekt by pak neměl přikročit k dalšímu vývoji. Specifikování architektury, jako jednoho z produktů projektu, umožní její použití v průběhu procesu vývoje a údržby“ [Boehm95].
[6] Zde je dobré si uvědomit proporce mezi nároky vývoje a údržby.
[7] Špatná může znamenat to samé co žádná v tom smyslu, že rozhodnutí týkající se struktury vznikají jako nutný „vedlejší efekt“ vlastního návrhu bez jasného vědomého záměru zabývat se touto otázkou.

1.2 Definice softwarové architektury

V současné době zatím neexistuje žádná oficiální definice softwarové architektury. Proto je též poskytnuta sekce Problematika definice softwarové architektury (sekce č. 2.1) dále v textu. Na druhou stranu všechny definice mají společné jádro a liší se buď svým záběrem, důrazem na určitý aspekt či detailností. Pro tento text byla jako „oficiální" vybrána definice, která je výsledkem diskusní skupiny věnované problematice softwarové architektury v Institutu softwarového inženýrství při Carnegie Mellon University v Pittsburghu. Tato definice byla např. použita v [Garlan95, Clements96, Chastek96]. Následuje definice:

Softwarová architektura je struktura komponent programu/systému, jejich vzájemné vazby, principy a předpisy určující jejich návrh a vývoj v průběhu času. (Původní znění: The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time.)

1.3 Hrubé vymezení našeho zájmu

Z pohledu našeho zájmu lze činnosti prováděné při návrhu programového systému rozdělit do několika skupin. Jednu skupinu tvoří činnosti, které provádějí dekompozici systému do dekompozičních jednotek (míněno z pohledu návrhu) v potřebném počtu úrovní dekompozice. Činnosti, které provádějí specifikaci těchto dekompozičních jednotek. Činnosti navrhující spolupráci těchto dekompozičních jednotek. Tedy činnosti jednoznačně zřejmé, když se hovoří o návrhu. Pracovně tyto činnosti nazvěme vlastní návrh. Vlastní návrh má jednoznačnou návaznost na analýzu vyvíjeného systému v tom smyslu, že navrhuje realizaci analytického modelu vyvíjeného systému.

Další skupinu tvoří činnosti, které specifikují, jakým způsobem jsou dekompoziční jednotky systému na jednotlivých úrovních dekompozice reprezentovány a realizovány v daném vývojovém prostředí. Činnosti, které specifikují, jakým způsobem je realizována spolupráce[8] dekompozičních jednotek na jednotlivých úrovních dekompozice. Činnosti, které specifikují další nároky na strukturální aspekty programového systému v průběhu jeho vývoje a dalšího života. Tyto činnosti se obecně týkají nároků na strukturu, resp. architekturu vyvíjeného programového systému. Pracovně tedy tyto činnosti nazvěme nároky na architekturu.

Vlastní návrh by obecně nikdy neměl začít bez jasně stanovených nároků na architekturu. Architektura programového systému má zásadní vliv jak na jeho fungování, tak na jeho vývoj, údržbu a další rozvoj. Proto stanovení nároků na architekturu musí být činnost vědomá a nikoli jen nutný „vedlejší efekt" vlastního návrhu. Předmětem této kapitoly je tedy problematika nároků na architekturu. Nároky na architekturu programového systému přirozeně značným způsobem předurčují konečnou architekturu softwarového systému. V tomto smyslu je tedy předmětem této kapitoly softwarová architektura a v tomto smyslu bude termín softwarová architektura používán, nebude-li řečeno či z kontextu nevyplyne jinak. Pro úplnou jasnost ještě na vysvětlení zrekapitulujme: Konečnou architekturou je např. myšleno, že na nejvyšší úrovni dekompozice má systém pět dekompozičních jednotek (tato problematika není předmětem této kapitoly - je předmětem vlastního návrhu). Nárokem na architekturu je např. myšleno, že všechny dekompoziční jednotky, které se dále člení, jsou vzhledem k dekompozičním jednotkám na stejné úrovni dekompozice reprezentovány tzv. třídou zabezpečující rozhraní („interface class"). Třídou zabezpečující rozhraní se myslí to a to, atd. (tato problematika je předmětem této kapitoly).

1.4 Struktura textu (kapitoly Softwarová architektura)

Sekce č. 2 popisuje „co to je softwarová architektura?" a tím umožňuje i její hlubší pochopení. Sekce 3 popisuje, proč je softwarová architektura důležitá. Sekce 4 popisuje vývoj při použití konceptu softwarové architektury. Sekce 5 přináší ilustrační příklady umožňující pochopit důležitost vědomého uchopení a používání konceptu softwarové architektury. Sekce 6 předestírá některé související práce se softwarovou architekturou. Sekce 7 definuje minimální hranice chápání a naplnění konceptu softwarové architektury, která je vyžadována minimální dobrou praxí softwarového inženýrství.

[8] Zahrnuje veškerou nutnou spolupráci dekompozičních jednotek systému, což obecně zahrnuje jejich: komunikaci, koordinaci, kooperaci, synchronizaci, atd. Způsobem realizace spolupráce je míněn mechanismus, což např. zahrnuje: vyvolání procedury s předáním parametrů přes společnou oblast, službu poskytovat přes SVC (supervisor call), pro synchronizaci aktivních objektů (míněno obecně) použít mechanismus rendezvous, pro synchronizaci pasivních objektů použít mechanismus semaforu s původní sémantikou operací P a V, atd.

2. Vymezení problematiky softwarové architektury - Co to je softwarová architektura?

2.1 Problematika definice softwarové architektury

Jako součást vymezení problematiky softwarové architektury budou uvedeny některé její definice. Ačkoli pro tento text byla vybrána a již výše uvedena „oficiální“ definice, tak uvedení dalších definic není samoúčelné. Naopak obsahuje mnoho podnětů k zamyšlení a pomáhá ukazovat (potenciální) šíři záběru probírané problematiky.

Nejprve si však uveďme dvě vhodně vybrané civilní definice termínu architektura:

1. Architektura je technické řešení nebo vůbec uspořádání, skladba nějakého (složitějšího) celku; odvozeno z Akademického slovníku cizích slov [Academia95].
2. Architektura je struktura čehokoli; výklad č. 6 z [Webster94].

Následuje několik vybraných definic softwarové architektury:

1. Softwarová architektura je studium struktur většího měřítka a výkonu softwarového systému. Důležité aspekty architektury systému zahrnují rozdělení funkcí mezi moduly systému, prostředky komunikace mezi moduly a reprezentaci sdílených informací [Lane90].
2. Abstraktně řečeno, softwarová architektura zahrnuje popis elementů, ze kterých jsou postaveny systémy, interakce mezi těmito elementy, předepsané vzorové způsoby, které předepisují jejich kompozici a omezující podmínky na tyto vzorové způsoby. Obecně, daný systém je definován pomocí souboru komponent a interakcí mezi těmito komponentami. Takový systém může být obratem použit jako (kompozitní) element při návrhu většího systému [Shaw94].
3. Softwarová architektura je množina definic a pravidel, které definují komponenty softwarového systému, jejich rozhraní a pravidla pro jejich interakce [Sha95].
4. Softwarová architektura je, zhruba řečeno, pohled na systém, který zahrnuje hlavní komponenty systému, chování těchto komponent, jak se jeví zbytku systému a způsoby, kterými komponenty interagují a koordinují své chování k dosažení cílů systému. Architektonický pohled je abstraktní pohled přinášející s sebou porozumění vyšší úrovni, a potlačení a odsunutí detailů vlastních většině abstrakcí [Clements96b].
5. Softwarová architektura sestává z: (i) soubor komponent software a systému, propojení a omezujících podmínek; (ii) soubor definovaných potřeb zainteresovaných stran na systém; (iii) zdůvodnění, které demonstruje, že komponenty, propojení a omezující podmínky definují systém, který, pokud je implementován, by splnil soubor definovaných potřeb zainteresovaných stran na systém. Definice z roku 1995 pochází od Boehm a jeho studentů [SEISADef97].
6. Softwarová architektura má přinejmenším čtyři odlišná ztělesnění: V každé kategorii, struktury popisují systém z jiné perspektivy: (i) Konceptuální architektura popisuje systém pomocí jeho hlavních elementů návrhu a vztahů mezi nimi; (ii) Architektura propojení modulů zahrnuje dvě ortogonální struktury: funkční dekompozici a vrstvy; (iii) Architektura provádění („execution“) popisuje dynamické struktury systému; (iv) Architektura kódu popisuje způsob, jakým jsou zdrojový kód, přeložený kód a knihovny organizovány ve vývojovém prostředí. Tento pohled z roku 1995 pochází od Soni, Nord a Hofmeister ze Siemens Corporate Research. Autoři píší, že je založen na studiu převažujících a významných struktur zjištěných ve vývojových prostředích průmyslových projektů, které studovali [SEISADef97].

Shrneme-li, dolní mez vnímání problematiky softwarové architektury je následující: Softwarová architektura je o strukturálních otázkách systému. Strukturální otázky mohou být vyjádřeny pomocí komponent, vzájemných vazeb, principů a předpisů, co se jejich použití týče. Přesné vymezení uvažovaných strukturálních otázek a způsob jejich reprezentace se liší podle důrazu uživatele softwarové architektury.

Rekapitulujeme-li náš důraz, pak je vyjádřen „oficiální“ definicí softwarové architektury uvedené v sekci Definice softwarové architektury (sekce č. 1.2), sekcí Hrubé vymezení našeho zájmu (sekce č. 1.3) a akcentem definice č. 3 uvedené výše. Poznamenejme ale, že tímto důrazem nejsme v žádném případě limitováni. Dále poznamenejme, že termíny „vzájemné vazby“, „propojení“, „interakce“, „koordinace“, atd. vztahující se ke komponentám, entitám, elementům, atd. jsou míněny obecně a nic dopředu nepředjímají.

Nakonec uveďme, že na The First International Workshop on Architectures for Software Systems 1995, Shaw poskytla následující souhrn z definic a pohledů (implicitních i výslovných) prezentovaných na tomto workshopu: Všechny strukturální modely splňují, že softwarová architektura je představována komponentami, propojeními mezi těmito komponentami, plus (obvykle) některými dalšími aspekty, které zahrnují: (i) konfiguraci, styl; (ii) omezující podmínky, sémantiku; (iii) analýzy, vlastnosti; (iv) důvody, požadavky, potřeby zainteresovaných stran [SEISADef97].

2.2 Původ softwarové architektury

Studium softwarové architektury je v podstatě studium struktury programových systémů. Podstatné je, že je to výslovné a vědomé studium struktury s vědomím všech následků strukturálních rozhodnutí. Následně velmi stručně zmíníme významné práce a myšlenkové koncepty, které značnou měrou přispěly k systematickému studiu strukturálních otázek, resp. stály u jejich počátků. Poznamenejme, že jejich autoři jsou dnes žijící legendy oboru počítačů a softwarového inženýrství.

2.2.1 Dijkstra - struktura operačního systému T.H.E.

Dijkstra ve svém článku [Dijkstra68] ukázal, že se vyplatí zabývat se tím, jak je softwarový systém členěn a strukturován oproti pouhému programování za účelem dosažení správných výsledků. Dijkstra toto ilustroval na operačním systému, který byl členěn pomocí vrstev („layered structure“). Jednotlivé části operačního systému byly sdružovány do těchto vrstev a části a dané vrstvy mohly komunikovat pouze se sousedními vrstvami. Dijkstra poukázal na zisky při vývoji a údržbě takového systému.

Poznamenejme, že výše naznačená architektura obecně není pro operační systémy v žádném případě všelék. Nicméně nastolení tématu architektury softwarových systémů a důsledků architektonických rozhodnutí bylo a je velmi cenné a přínosné. Jmenovitě oblast architektury, struktury, strukturálních omezení, architektury vzhledem k paralelismu, atd. operačních systémů je dodnes a napořád bude velmi podnětná a živá. Zvláště po nástupu objektově orientovaných konceptů. Za všechny zajímavé projekty na poli operačních systémů a jejich architektury jmenujme pouze několik: *Choices* [Campbell93], CHORUS [Gien92], Mach [Loepere92], PEACE [Shröder92a], Apertos [Yokote92]. Některé zajímavé momenty z architektury výše uvedených operačních systémů budou uvedeny jako příklad či poslouží jako srovnání v sekci č. 5 níže v textu. Přehled architektonických otázek a různých řešení v oblasti operačních systémů s doslova stovkami referencí lze nalézt v [Mukherjee93, Mukherjee94].

2.2.2 Parnas - moduly, struktury software, rodiny programů

Článek [Parnas72] přispívá k problematice dekompozice systému do modulů. Pojednává též o konceptu zapouzdření. Článek [Parnas74] je o struktuře software. Článek [Parnas76] je o konceptu rodiny programů („Program Families“). Tento koncept říká, že jistou skupinu programů (potenciálně možných) je vhodné vnímat ve vzájemné souvislosti. Tímto je myšleno, že při návrhu prvních členů této skupiny je třeba činit jen taková rozhodnutí, která vyhoví všem následným (potenciálně možným) členům této skupiny. Pokud toto zobecníme, lze uvažovanou skupinu programů vnímat jako strom, kde čím blíže kořeni, tím zásadnější rozhodnutí návrhu pro celou skupinu - rodinu programů. Raná rozhodnutí u kořene lze vnímat *de facto* jako stanovení architektury pro celou rodinu programů. Je třeba nezaměňovat koncept rodiny programů s doménou problému a její podporou programovým systémem. Další práce [Parnas79] se týká problematiky návrhu systému, který je možno snadno rozšiřovat a redukovat. Jak je vidět, všechny zde zmíněné Parnasovy práce se týkají témat úzce spjatých se softwarovou architekturou. Poznamenejme, že filosofie návrhu operačního systému PEACE [Shröder92b] je ovlivněna pozdějšími Parnasovými pracemi.

2.2.3 Brooks - konceptuální integrita

„Tvrdím, že konceptuální integrita je nejdůležitější otázka při návrhu systému. Je lepší mít systém bez určitých anomálních rysů a vylepšení, ale s dodržením jedné množiny konceptů návrhu, nežli mít systém, který obsahuje mnoho dobrých, ale nezávislých a nekoordinovaných konceptů“[9] [Brooks95]. Dodržení jedné množiny konceptů návrhu však znamená stanovit si tyto koncepty a vlastně nároky na návrh a tedy i na strukturu. Brooks na úvod do tématu uvádí příklad, kdy nás dodnes uvádí v obdiv architektonická jednota katedrály v Remeši, kde osm generací stavitelů katedrály respektovalo původní architektonické zásady a zřeklo se vlastních nápadů a pokusů o vylepšení. Tato sebekázeň se vyplatila. Softwarové systémy se sice nebudují desetiletí a staletí po sobě jdoucími mistry návrháři. Avšak je třeba zabránit, aby návrh byl činěn nekoordinovaně po mnoha kouscích mnoha lidmi, bez respektování předem stanovených zásad softwarové architektury.

[9] Bylo citováno z kapitoly Aristocracy, Democracy, and System Design.

2.3 Co se vše myslí softwarovou architekturou - možné pohledy a interpretace

2.3.1 Problematika začlenění konceptu softwarové architektury do návrhu

Návrhem (resp. designem) rozumíme tu část vývojového procesu software, jejímž hlavním smyslem je rozhodnout, jak bude systém implementován. Koncept softwarové architektury se v návrhu jednak projeví tím, že před vlastním návrhem je třeba stanovit nároky na softwarovou architekturu a po té dbát na jejich plnění. Takže výsledný systém vykazuje takovou celkovou architekturu, která odpovídá dříve stanoveným nárokům. Tedy téma softwarové architektury průběžně doprovází celý návrh. Stanovení nároků na softwarovou architekturu před vlastním návrhem lze též chápat jako učinění první sady rozhodnutí týkajících se návrhu. Jak je vidět v následující sekci, softwarovou architekturu systému, resp. strukturální otázky je možno a třeba vidět z několika vzájemně se doplňujících perspektiv. Pro každou z těchto perspektiv ovšem platí, že může být interpretována jako popis systému, který byl vyvinut, ale též i jako předpis, co má být při vývoji respektováno, aby bylo dosaženo jistých záměrů.

2.3.2 Pohledy a možné vnímání struktury softwarového systému

Jak již bylo řečeno výše, problematika softwarové architektury se týká otázek struktury softwarového systému a je součástí návrhu. Zaměřme se nyní blíže, co je míněno strukturálními otázkami a jaké z toho plynou důsledky. Dle [Shaw94, Clements96] strukturální otázky zahrnují:

- „hrubá“ organizace systému;
- globální řídící struktury;
- komunikační protokoly;
- synchronizace a přístup k datům;
- přiřazení funkcionality k elementům návrhu;
- fyzická distribuce;
- skladba elementů návrhu;
- rozšiřování a výkon;
- výběr mezi alternativami návrhu.

Jako každý jiný složitý systém vykazuje složitý softwarový systém mnoho struktur[10]. Je proto se třeba vyvarovat neurčitým popisům typu „celková struktura“. A vždy je třeba přesně stanovit, co je termínem struktura míněno, která struktura je na mysli. Těmto jednotlivým zdůrazněním se říká architektonický pohled či model. Různé architektonické modely nejsou v protimluvu, naopak se doplňují a dohromady tvoří softwarovou architekturu. Jinak řečeno, u složitých systémů lze těžko jednou sadou strukturálních elementů a vazeb popsat všechny aspekty jejich architektury. Dále je nutno uvažovat vztahy mezi architektonickými pohledy.

Každý architektonický pohled (resp. pohled na architekturu) se zaměřuje na určitou množinu zájmů. Pohledy jsou proto abstrakce, každý vzhledem k různým kriteriím. Každá abstrakce odstraní detaily, které jsou nezávislé na zájmech, pro které je tato abstrakce tvořena. Každý pohled může používat vlastní notaci, může definovat, co v něm znamená: komponenta, vzájemná vazba, důvod, princip a předpis. Jednotlivé pohledy však nejsou zcela nezávislé. Elementy v jednom se mohou vztahovat k elementům v jiném. Takže stejně, jako nám pomůže uvažovat je zvlášť, je též nutné pečlivě a přesně uvažovat o vzájemných závislostech mezi jednotlivými pohledy.

Uveďme příklad č.1: Je systém implementován v jazyku C. Jednotlivé moduly systému budou realizovány jazykovým konstruktem funkce jazyka C. Moduly, které spolu souvisí, budou uloženy v jednom souboru systému souborů (míněno ve vývojovém prostředí). Překladem jednoho takového souboru systému souborů vznikne relativní modul vzhledem k linkovacímu programu. Vazba mezi relativními moduly je zabezpečena mechanismem ENTRY & EXTRN. Systém je na nejvyšší úrovni dekompozice (nejhrubší úrovni granularity) dekomponován do subsystémů. Subsystém je tvořen několika relativními moduly. Se subsystémem je spřažen jeden proces a subsystém se vzhledem k ostatním subsystémům chová jako aktivní objekt a komunikuje s nimi pomocí mechanismu synchronního zasílání zpráv. Subsystém vždy obsluhuje právě jeden požadavek, ostatní čekají ve frontě. Nyní se podrobněji podívejme na tuto architekturu[11] a uveďme si náznaky třech příkladů architektonických pohledů:

- logická hierarchická dekompozice za účelem zvládnutí složitosti systému: systém **à** subsystém **à** skupina modulů **à** modul; předávání řízení mezi moduly: volání C funkce; spolupráce subsystémů: synchronní zasílání zpráv;
- dekompozice vzhledem k uspořádání systému při vývoji a překladu: soubory systému souborů obsahující skupinu modulů a obsahující klauzule ENTRY & EXTRN;

[10] Např. struktura zdrojového kódu vzhledem k překládání, struktura systému vzhledem k dekompozičním jednotkám, struktura systému vzhledem k testování, struktura systému vzhledem ke konfiguračnímu řízení, struktura systému vzhledem k dekompozici v případě distribuovaného prostředí, struktura systému při běhu vzhledem k procesům, resp. vláknům („thread“), atd.

[11] Poznamenejme, že jde o zjednodušený popis architektury operačního systému MINIX [Tanenbaum87], který slouží výukovým účelům.

– uspořádání systému při běhu: subsystémy spřažené s jedním procesem tvořící aktivní objekt spolupracující pomocí synchronního zasílání zpráv (pozn. nevyřízené žádosti čekají ve frontě).

Existence popisu softwarové architektury umožňuje mimo jiné její analýzu a tím pádem i možnost dopředu říci něco o vlastnostech softwarového systému, který ji bude vykazovat. Například zaměříme-li se na architektonický pohled „uspořádání systému při běhu“ ve výše uvedeném příkladu č. 1, lze říci následující: granularita synchronizace je hrubá, jmenovitě synchronizace se děje vzhledem k subsystémům. Dopad je příznivý na náročnost programování, protože *de facto*[12] odpadnou nároky na starosti se synchronizací. Dopad je nepříznivý na průchodnost, resp. výkon systému. Na vysvětlenou dodejme, že v příkladu č. 1 např. systém souborů je subsystém.

Příklad 1

Dle [Clements96] architektonické pohledy mohou být kategorizovány následně:

– Zda jsou struktury, které pohled reprezentuje, rozlišitelné v době běhu (runtime) systému. Například v příkladu č. 1 uvedeném výše, je subsystém viditelný, modul ne.
– Zda struktury popisují produkt, proces vývoje produktu, či proces používání produktu. (Pozn. všechny pohledy zde zmiňované popisují produkt).

V současné době neexistuje ustálená standardní sada architektonických pohledů či termínů, jak je referovat. Následuje sada dle [Clements96] typických a užitečných pohledů. Poznamenejme, že jejich pojmenování není ustálené a jednotné a že to není jediná možná sada a ani vyčerpávající. Architektonické pohledy[13] dle [Clements96]:

– Konceptuální (logický) pohled: Konceptuální (nebo logický) architektonický pohled zahrnuje množinu abstrakcí potřebnou k vyjádření funkčních požadavků na systém na abstraktní úrovni. Tento pohled je užitečný při komunikaci architekta s expertem na doménu problému. Konceptuální pohled je nezávislý na implementačních rozhodnutích a místo toho zdůrazňuje interakce mezi entitami v doméně problému. V případě objektově orientovaného přístupu může být tento pohled vyjádřen diagramy tříd, resp. kategoriemi tříd (koncept seskupení tříd umožňující hierarchický rozklad).
– Modulární (vývojový) pohled: Modulární (nebo vývojový) pohled je často používaná architektonická struktura. Zaměřuje se na organizaci skutečných programových modulů. Tento pohled může mít různé formy v závislosti na tom, jak jsou moduly v systému organizovány. Moduly lze organizovat a slučovat do identifikovatelných podsystémů (pokud to složitost vyžaduje podsystémy, zas do podsystémů, atd.) podle různých kriterií. Lze je organizovat na základě četnosti jejich vzájemné komunikace, lze je organizovat na základě konceptu zapouzdření pro ulehčení údržby systému. Moduly lze organizovat vzhledem k logickému uspořádání systému (např. do vrstev). Moduly lze organizovat vzhledem k organizaci zdrojového kódu a překládání. Poznamenejme, že jedna konkrétní fyzická organizace modulů nevylučuje mnoho různých ,,a pro daný účel vhodných, „logických“ organizací, tedy pohledů. Protože tento pohled je nejčastější, udělejme shrnutí: Modul považujme za základní dekompoziční jednotku při vývoji systému. Moduly jsou organizovány a slučovány do podsystémů a to hierarchicky. Moduly lze organizovat vzhledem k logickému uspořádání systému. Možné vztahy a závislosti mezi moduly, vztah datových abstrakcí a modularity, atd. jsou diskutovány v [Rajlich77].
– Procesní (koordinační) pohled: Tento pohled se zaměřuje na chování systému během jeho běhu („runtime“). Strukturální komponenta v tomto pohledu obvykle bývá proces. Ovšem obecně jí může být jakýkoli koncept sloužící jako prostředek pro vykonávání strojových instrukcí z pohledu kódu softwarového systému. V této souvislosti se lze kromě termínu proces setkat s termíny[14]: „těžký“ proces („heavy-weight process“), „lehký“ proces („light-weight process“), vlákno („thread“), „muší“ proces („feather-weight process“). V tomto architektonickém pohledu jde hlavně o to, jak procesy vzájemně spolupracují, jak se synchronizují, jak vznikají, zanikají, jaké je mapování mezi procesním pohledem a strukturami jiných pohledů, např. vývojového. Jde o to, zda části systému poskytující určité služby[15] budou fungovat jako pasivní objekty, či aktivní objekty. Zda umožní své paralelní využití (tj. budou-li reentrantní). Budou-li reentrantní, tak zda vzhledem k vnějším procesům, či vnitřně k dynamicky generovaným vláknům, atd.[16]. Tento pohled pomáhá

---

[12] Například subsystém systém souborů se musí chovat vzhledem k příchozím požadavkům jako monitor. To jest aktivní proces je v něm jenom jeden (obsluhuje se jeden požadavek), ale některé požadavky v něm mohou být zároveň zablokovány (např. čtení z pipe; na vysvětlenou OS MINIX implementuje sémantiku UNIX v7).

[13] Následující popisy jednotlivých pohledů je třeba brát spíše jako podnět k přemýšlení, než přesnou definici. Např. při používání termínu modul velmi záleží, co se tím vlastně přesně myslí, záleží na vývojovém prostředí, atd. Dále poznamenejme, že příklady a ilustrace použité v jednotlivých popisech nemusí představovat nejlepší odborné řešení z pohledu softwarového inženýrství, ale že slouží pro vymezení toho kterého pohledu.

[14] Jde o míru přepínání kontextu (pozn. kontextem je míněna paměť, registry procesoru, kontext file systému, atd.).

[15] Tento případ je typický pro operační systémy, databázové stroje, a paralelní systémy vůbec a ovšem též pro jejich používání z pohledu systémového programátora.

[16] Problematika části systému (objektu), který nabízí služby okolí, je vzhledem k paralelismu, synchronizaci a v případě objektově orientovaného prostředí i vzhledem k dědičnosti rozebrána např. v [Kosvancova94].

při přemýšlení o výkonu systému a o jeho provozních charakteristikách. Poznamenejme, že problematika paralelních procesů je rozebrána např. v [Andrews83, Andrews91].

- Fyzický pohled: Fyzický pohled znázorňuje mapování software na hardware. Toto mapování musí být flexibilní a mít co nejmenší dopad na zdrojový kód.

Jednotlivé pohledy poskytují různé struktury systému, jednotlivé pohledy mají cenu samy o sobě. Ačkoli jednotlivé pohledy dávají různé perspektivy struktury systému, nejsou nezávislé. Strukturální elementy z jednoho pohledu mají vztahy s elementy z dalších pohledů. Tyto vztahy je třeba znát a vědět, co znamenají. Toto je v náznaku vidět např. na příkladu č. 1 výše.

3. Přínos softwarové architektury - Proč je softwarová architektura důležitá?

3.1 Úvodní shrnutí

Vědomé stanovení nároků na softwarovou architekturu, které má vyvíjený softwarový systém vykazovat, před vlastním návrhem má mnoho kladných přínosů. Od zřejmých po ne zcela zřejmé, od zcela automatických po potenciálně možné. Dále, vědomé uvažování o architektonických otázkách a stanovení softwarové architektury přináší mnoho možností. Opět od zřejmých po ne zcela zřejmé, od jednoduchých po rafinované, od lehce využitelných po možnosti nárokující si dlouhodobé úsilí.

Před stručným rozborem těchto přínosů a možností si aspoň některé naznačme:

- dokumentace softwarové architektury představující jednotný přístup k strukturálním otázkám je nesmírně přínosná pro celý další život systému, tj. jeho vývoj a (hlavně) údržbu;
- softwarová architektura by měla systematicky odrážet kvalitativní nároky na vyvíjený systém (např. udržovatelnost, rozšiřitelnost) a naopak analýzou softwarové architektury lze o systému mnohé říci, tedy jde o jednoznačný přínos k zajišťování jakosti softwarového procesu a produktu;
- vědomé stanovení softwarové architektury nutí k jednotnému a vnitřně konzistentnímu učinění těch nejzásadnějších rozhodnutí designu;
- stanovená a dokumentovaná softwarová architektura umožní její sdělení všem zainteresovaným stranám při vývoji a údržbě softwarového systému.

3.2 Klasifikace přínosů softwarové architektury

Níže uvedená klasifikace čerpá z [Clements96, Clements96b]. Uveďme, že níže uvedená klasifikace přínosů a důvodů, proč je softwarová architektura důležitá, jistě není úplná a jistě některé aspekty jsou diskutabilní, avšak přesto je velmi podnětná a ilustrativní.

3.2.1 Hrubší klasifikace

Dle [Clements96b] zaměření na problematiku softwarové architektury přislibuje lepší zvládnutí níže vyjmenovaných oblastí. Kde pokrok v každé z nich může vést k významnému zlepšení ve vývoji a aplikaci velkých a složitých softwarových systémů. Oblasti to jsou následující:

- Vývoj založený na komponentách („Component-based development“): Systémy mohou být postaveny rychlým a efektivním způsobem převzetím (nebo vygenerováním) velkých externě vyvinutých komponent. Známá architektura umožňuje samostatný a nezávislý vývoj skladebních komponent, kde integrace se stává stěžejním problémem.
- Včasná předpověď jakosti („Early quality prediction“): Studiem architektury systému je možno do předu stanovit jeho určité vlastnosti i při neexistenci detailního návrhu a kódu. Jedná se jak o vlastnosti, které systém vykazuje při běhu, tak i o statické. Například výkon je do velké míry dán frekvencí a způsobem komunikace mezi komponentami, udržovatelnost je do velké míry dána lokalizací (resp. lokalizovatelností) změn. Vzájemný vztah mezi softwarovou architekturou a kvalitou softwarového produktu je obousměrný. Tedy požadavky na kvalitativní atributy vyvíjeného systému je možno a třeba zohlednit při tvorbě či výběru softwarové architektury. Tímto se např. zabývá [Kazman94]. Naopak problematika analýzy a hodnocení softwarové architektury je např. probírána v [Abowd97, Barbacci97]. Nakonec poznamenejme, že špatná softwarová architektura může být úplnou překážkou k dosažení jistých kvalitativních atributů vyvíjeného systému, avšak dobrá architektura je sice nezaručuje, ale otvírá k nim cestu.
- Vývoj produkčních řad („Product line development“): Celé produkční řady mohou být vyvinuty sdílením společné architektury. Znovupoužití ve velkém měřítku je možné skrze plánování na úrovni architektury. Produkční řada je skupina příbuzných systémů, které dohromady zaplňují poptávku v určité oblasti. Produkční řadu si lze představit jako rodinu programů [Parnas76], kde výběr architektury odpovídá rozhodnutím blízko kořene Parnasova rozhodovacího stromu (viz sekce č. 2.2.2 výše).
- Oddělení funkcionality a propojení („Separation of functionality from interconnection“): Funkcionalita komponent může být oddělena od mechanismů propojení komponent. Toto má své dobré důvody. Mechanismy propojení mohou být různé, např. vyvolání procedury s parametry, vyvolání procedury s globálními daty, implicitní vyvolání při splnění události, atd. Ovšem volba určitého mechanismu má jisté důsledky. Pokud volba mechanismu proběhne bez rozmyslu jako „vedlejší“ efekt vlastního návrhu a programování, může se ukázat špatná a většinou se již nedá měnit. Je proto například vhodné zavést určité abstrakce pro propojení a konkrétní mechanismus použít, až když je to opravdu třeba.

– Omezení prostoru návrhu („Constraining the design space“): Méně je více: omezit prostor návrhu se vyplatí. Předem stanovit určitá omezení na strukturu programového systému a řešení určitých momentů, které se často opakují, se vyplatí. Vyplatí se to vzhledem k vývoji, znovupoužitelnosti, čitelnosti, testovatelnosti, udržovatelnosti, rozhodování mezi alternativami, atd. Jedním z úsilí na tomto poli je oblast tzv. vzorů (lze se setkat s termíny „design patterns“, „object-oriented patterns“, „software patterns“). Problematice vzorů je např. věnováno několik článků v [CACM9610] a článek [Coad92]. Přibližně řečeno, to, co strukturované programování znamená pro oblast programování v malém, znamenají strukturální omezení definovaná softwarovou architekturou pro oblast návrhu.

3.2.2 Jemnější klasifikace

Tato sekce přináší jemnější klasifikaci důvodů a názorů, proč je softwarová architektura důležitá. A proč praktikování vývoje založeného na definované softwarové architektuře je cenné. Klasifikace pochází z [Clements96].

3.2.2.1 Architektura je prostředek pro komunikaci zainteresovaných stran

Architektura představuje prostředek, jak i (a hlavně) u velkých systémů mohou různé strany diskutovat své zájmy. Architektura ovlivňuje celou řadu vlastností systému a přitom i pro rozsáhlé systémy je myšlenkově uchopitelná a tím pomáhá k pochopení systému a k činění rozhodnutí v ranných stadiích návrhu, která velmi ovlivňují výsledek.

3.2.2.2 Architektura obsahuje sadu prvotních rozhodnutí o návrhu systému

Architektura reprezentuje soubor prvotních rozhodnutí o návrhu systému. Tato prvotní rozhodnutí jsou nejsložitěji napravitelná, nejobtížněji změnitelná a mají největší dopad na další vývoj[17]. Některé zmíněné dopady si uveďme:

– Architektura definuje omezení na realizaci systému: O realizovaném (implementovaném) systému řekneme, že vykazuje jistou architekturu, pokud je v souladu se strukturálními nároky na návrh, které jsou popsány architekturou. Systém musí být tedy rozdělen do předepsaných komponent, komponenty musí navzájem interagovat předepsaným způsobem, komponenty musí naplňovat svoji specifikaci, jak předepisuje architektura. Tvůrci architektury nemusí být experty na návrh algoritmů či speciality programovacího jazyka. Naopak realizátoři komponent nemusí znát problematiku celkové struktury, učiněných kompromisů na úrovni celého systému, atd.
– Architektura určuje organizační strukturu pro projekt vývoje a údržby: Architektura nejen že předepisuje nároky na strukturu systému, ale též má vliv na rozdělení a organizaci práce při vývoji a údržbě.
– Určitá architektura umožňuje či zabraňuje dosažení cílových kvalitativních atributů systému: Obousměrný vztah mezi architekturou a jakostí byl diskutován již výše v sekci č. 3.2.1. Zde se zaměřme na problém volby architektury, za účelem umožnění naplnění požadovaných kvalitativních atributů. Kvalitativní atributy mohou být rozděleny do dvou kategorií. První kategorie zahrnuje ty, které mohou být zjištěny měřením běžícího systému (např. výkon, spolehlivost, bezpečnost). Druhá kategorie zahrnuje ty, které nemohou být zjištěny měřením běžícího systému, ale naopak pozorováním jeho vývoje a údržby. Modifikovatelnost např. záleží na modularizaci systému a respektování konceptu zapouzdření. Vliv architektury je především na nefunkční požadavky. Práce [Kazman94] identifikuje šest tzv. základních operací („unit operations“), kterými je možno formovat architekturu: separace („Separation“), abstrakce (Abstraction“), komprese („Compression“), uniformní kompozice (Uniform Composition“), replikace („Replication“), sdílení zdrojů („Resource Sharing“). Dále je zkoumán vztah mezi těmito základními operacemi a souborem osmi nefunkčních kvalitativních atributů: přizpůsobení rozsahu („Scalability“), modifikovatelnost („Modifiability“), integrovatelnost („Integrability“), přenositelnost („Portability“), výkon („Performance“), spolehlivost („Reliability“), snadnost vytvoření („Ease of Creation“), znovupoužitelnost („Reusability“). Vliv základních operací na kvalitativními atributy je prezentován v tabulce 1. Tabulka 1 přináší identifikaci nejvýznamnějšího vlivu a předpokládá správné použití základních operací. Tabulka je převzata z přílohy A [Kazman94].

	přizpůsobení rozsahu	modifikovatelnost	integrovatelnost	přenositelnost	výkon	spolehlivost	snadnost vytvoření	znovupoužitelnost
separace	+	+	+	+	+/-		+/-	+
abstrakce	+	+	+	+	-		+	+
komprese	-	-	-	-	+		+/-	-

[17] Konstatování odpovídá poučce o exponenciálním růstu ceny chyby vzhledem k počátku životního cyklu.

unif. kompozice	+		+				+	
replikace	-	-		-	+/-	+	-	-
sdílení zdrojů		+	+	+	+/-	-	+	+/-

Tabulka 1

- Studiem architektury je možno predikovat určité kvalitativní vlastnosti systému: Studium architektury umožní provádět predikce o systému. Studium, analýza, hodnocení navržené, resp. vybrané architektury před její aplikací je velmi důležité, protože má dalekosáhlý vliv. Viz sekce č. 3.2.1 výše.
- Architektura může být základ pro školení: Popis základních zásad struktury systému je velmi vhodný podklad pro nové členy projektu jak vývoje, tak (hlavně) údržby.
- Určitá architektura pomáhá uvažování o změnách a jejich zvládání: Většinu života softwarového systému často představuje údržba. Při údržbě je systém modifikován. Jde tedy o to, aby softwarová architektura umožnila vhodný přístup k provádění změn. Dále změny z pohledu architektury lze klasifikovat na: lokální, ne-lokální a architektury. Při změnách lokálních se změna týká jedné komponenty. Při změnách ne-lokálních je modifikováno více komponent. Při změnách architektury je nutno měnit např. způsob spolupráce komponent a pravděpodobně se takové změny budou týkat celého systému. Pochopitelně je záhodno, aby změny byly lokální. Proto je na architektuře, aby zajistila, že nejpravděpodobnější a nejčastější změny budou snadno proveditelné.

3.2.2.3 Architektura jako přenositelný model

Softwarová architektura formuje relativně malý, myšlenkově uchopitelný model toho, jak je systém strukturován a jak jeho komponenty dohromady pracují. Tento model je přenositelný skrze systémy. Obzvláště může být použit pro systémy, které vykazují podobné požadavky[18] a tento model může podpořit znovupoužití ve velkém měřítku. Čím dříve se v životním cyklu podaří aplikovat znovupoužití, tím větší efekt. Tedy použití softwarové architektury pro systémy s podobnými požadavky má velký přínos. Uvažování o architektuře jako přenositelném modelu lze členit:

- Celé produkční řady sdílí společnou architekturu: Tento bod byl již popsán v sekci č. 3.2.1 výše.
- Systémy mohou být postaveny za použití rozsáhlých externě vyvinutých komponent, které jsou kompatibilní s předdefinovanou architekturou: K popisu uvedenému v sekci č. 3.2.1 výše ještě dodejme: Pro možnost efektivního externího vyvíjení komponent je třeba mít opravdu dobře definované architektonické požadavky. Pro ilustraci, efektivní použití externě (míněno mimo projekt) vyvinuté komponenty vyžaduje kromě základních syntaktických (vstupní body, typ a počet parametrů, atd.) a sémantických (vliv na globální data, možné vyvolání výjimek, nějaký neformální popis, co komponenta dělá) informací následující: informace týkající se výkonu, informace týkající se bezpečnosti, informace týkající se spolehlivosti, předpoklady komponenty o používání vzhledem k procesům (je aktivní, pasivní, reentrantní, atd.), předpoklady komponenty o svém okolí během kompilace, linkování, či běhu systému, a další nutné informace o komponentě.
- Architektura umožňuje oddělení funkcionality komponent od mechanismů propojení komponent: K popisu v sekci č. 3.2.1 ještě dodejme: Interakce komponent (koordinace, kooperace, komunikace) zahrnuje širokou škálu otázek: volání, zasílání zpráv, způsob předávání dat, formáty dat, je komponenta aktivní/pasivní?, je komponenta reentrantní?, komunikace je synchronní/asynchronní?, používá se aktivní čekání?, atd. Důležité je, aby vlastní funkcionalita komponent byla od těchto problémů oddělena. To poté umožňuje volit konkrétní mechanismus dle vhodnosti. Dále je zde příležitost pro automatickou generaci tímto ovlivněného tzv. spojovacího kódu („glue code“). Oddělení koordinačního modelu od poskytovatelů služeb (tj. komponent) je v [Peterson94] označeno za klíčový atribut dobré architektury.
- Méně je více: omezení možných alternativ návrhu se vyplatí: Tento bod byl již popsán v sekci č. 3.2.1 výše.
- Určitá architektura umožňuje vývoj komponent založený na generických řešeních („Template-Based“): Řada momentů plynoucí ze strukturálních nároků může být lokalizována a naprogramována jen jednou. A poté vhodným mechanismem poskytnuta pro použití v komponentách. Vhodným mechanismem může obecně být: „include“ mechanismus, generičnost, dědičnost, použití služby, atd.

[18] Zde upozorněme obzvláště na nefunkční požadavky. Například průmyslové systémy, u kterých je vysazení z provozu velmi nákladné, vyžadují naprosto bezvadné zvládnutí přechodu na vyšší verzi (což je dáno buď jejich vývojem anebo modifikováním obecně). Toto je velmi náročný požadavek, který vykazují též vojenské systémy. Řešení problému se velmi týká softwarové architektury. Příkladem takové architektury je Simplex architektura [Sha95].

4. Vývoj založený na softwarové architektuře

Při používání vědomě formulované softwarové architektury během vývoje a údržby softwarového systému je třeba se vzhledem k softwarové architektuře zaměřit na následující okruhy problémů:

- Vývoj či výběr architektury: Architekturu je třeba vyvinout, resp. vybrat. Obecně se ukazuje, že vývoj softwarové architektury není jednorázová záležitost, ale spíše záležitost iterativního charakteru [Clements96]. Což zahrnuje prototypování, testování, měření a analýzu. Toto poskytuje prostředek pro včasné zajišťování splnění nároků na vyvíjený systém. Architektura je obecně ovlivněna třemi kategoriemi faktorů: (i) požadavky na systém, (ii) prostředí, ve kterém bude systém vyvíjen (např. organizace chce zhodnotit předchozí návrhy, používá se objektově orientované prostředí, atd.), (iii) předchozí zkušenosti tvůrců architektury.
- Reprezentace a sdělení architektury: Aby architektura mohla plnit svůj účel, je třeba ji vhodně reprezentovat a sdělovat všem zainteresovaným stranám. Problematika popisu softwarové architektury je rozebrána nap ř. v [Shaw94].
- Analýza a hodnocení architektury: Hodnocení a přezkoumání softwarové architektury potvrzuje, že vybraná či vyvinutá softwarová architektura je schopná uspokojit nároky systému předtím, než je systém implementován.
- Zajištění aplikace předepsané architektury: Zajištění aplikace předepsané softwarové architektury během celého následného života systému zajišťuje jeho konceptuální integritu. Bez této konceptuální integrity se rozhodnutí učiněná návrháři v době tvorby architektury znehodnotí a celá snaha přijde nazmar ve chvíli, kdy se předepsaná architektura přestane aplikovat. Protože potom definice architektury už neslouží jako popis systému na jisté úrovni abstrakce.

5. Ilustrační příklady některých momentů

V této sekci jsou uvedeny čtyři příklady ilustrující některé momenty a rozhodnutí, které se týkají struktury softwarového systému. Příklady mají formu prezentace dvou řešení určitého problému. Záměrem příkladů je ilustrovat úroveň abstrakce problémů softwarové architektury, nikoli jednotlivá řešení vyvyšovat či naopak. Příklady jsou vybrány z oblasti operačních systémů.

5.1 Příklad 1 - „o mechanismu propojení komponent“

V objektově orientovaném systému souborů operačního systému *Choices* [Madany92] (dále fs1 jako file system 1) a objektově orientovaném systému souborů prezentovaném v [Smolik95] (dále fs2 jako file system 2) jsou komponenty systému reprezentovány jazykovým konstruktem class jazyka C++. V systému fs1 je na všech úrovních dekompozice propojení mezi komponentami realizováno pomocí jazykového konstruktu ukazatel. V systému fs2 je tento mechanismus použit na první (tj. vyšší) úrovni dekompozice. Na druhé (tj. nižší) úrovni dekompozice, resp. jemnější úrovni granularity, je pro propojení mezi komponentami použit jazykový konstrukt pro dědičnost. Řešení fs1 je standardní univerzální a nekontroverzní, umožňující např. jednoduší rekonfiguraci systému. Řešení fs2 přináší přehlednější a jednoduší kód pěkně zrcadlící strukturování systému na dané úrovni dekompozice. Na druhou stranu se řešení fs2 hodí pouze pro nižší úroveň dekompozice. Dále je třeba si být vědom, že jazykový konstrukt dědičnosti se v tomto případě používá pro zcela nestandardní účel a proto je třeba znát všechny důsledky tohoto použití a schopnost ho obhájit např. proti [Rumbaugh93].

V této souvislosti poznamenejme, že problém architektury je též rozhodnout a předepsat způsob reprezentace a používání dále se členící komponenty.

5.2 Příklad 2 - „o mechanismu komunikace komponent“

Operační systém Mach 3.0 [Loepere92] nemá file system, je to jádro. Za tímto účelem existuje mnohovláknový UNIX server („Multi-threaded UNIX Server“) a rozhraní k tomuto file systému je označováno jako emulační knihovna („Emulation Library“). Toto rozhraní je nahráváno do adresového prostoru procesů používající výše uvedený UNIX server [Dean93]. Za účelem umožnění řešení některých systémových volání emulační knihovně bez standardního kontaktování UNIX serveru, emulační knihovna a server sdílí dvě stránky paměti. Toto řešení přináší elegantní zpracování některých volání file systému, avšak problém začíná, když se chce, aby procesy s emulační knihovnou v paměťovém prostoru začaly migrovat. Proto například operační systém CHORUS/MiX V.4 [Gien92, Dean93] takovéto sdílení stránek v obdobné situaci neumožňuje.

5.3 Příklad 3 - „o způsobu používání komponent“

Systém souborů operačního systému MINIX [Tanenbaum87] je aktivní objekt, který obsluhuje vždy jednu žádost, přičemž obsluha některých žádostí může být zablokovaná (viz příklad č. 1 v sekci č. 2.3.2 výše). Tento file system se vůči žádostem o své použití chová *de facto* jako monitor (např. viz [Andrews83]). Systém souborů [Smolik95] je pasivní objekt umožňující své mnohonásobné volání. V podstatě jde o přístup k paralelismu a synchronizaci. První případ je snadnější udělat, druhý případ umožňuje větší propustnost a své větší využití.

5.4 Příklad 4 - „o synchronizaci“

Podívejme se na problém přístupu k synchronizaci naznačený v bezprostředně předchozí sekci. V případě file systému MINIX se procesy při jeho použití v podstatě synchronizují na úrovni celého file systému, což je velmi hrubá granularita synchronizace a toto není typické, protože to vede k omezení možnosti využití změn ěného file systemu. Jemnější granularita synchronizace byla v klasických operačních systémech použita, např. vzhledem ke globálním datovým strukturám typu PCB („Process Control Block“, datová struktura reprezentující kontext procesu, co proces, to jedna položka v tabulce) nebo analogickým např. pro file system. Použití jazykového konstruktu třída (resp. hierarchie tříd) k popisu kontextu procesu a definování jeho rozhraní k operačnímu systému umožňuje reprezentování procesu v operačním systému jako instanci této třídy. Zde dochází k vyčlenění dat, ke kterým se není třeba synchronizovat při plnění jisté služby operačního systému [Smolik95] anebo se synchronizace provádí ještě na jemnější úrovni granularity.

6. Některé související práce

S problematikou softwarové architektury souvisí celá řada těsně svázaných témat. O dvou důležitých z nich se v následujících sekcích stručně zmíníme.

6.1 Skládání systémů z velkých celků

Skládání systémů z velkých celků je lákavá věc. V celé své šíři bohužel není tak snadná, jak se může jevit. Ovšem protože potenciální přínosy jsou značné, zabývá se touto problematikou celá řada projektů a skupin. Například Composable Software Systems (Carnegie Mellon University), Software Systems Generator Research Group (University of Texas), Software Composition Group (University of Berne), Object Systems Group (University of Geneva). Uveďme několik prací z tohoto okruhu [Batory92, Mey92, Tsichritzis92].

6.2 Frameworks

Objektově orientovaný framework[19] (OOF) je návrh architektury pro objektově orientované systémy. Popisuje komponenty systému a způsob jakým interagují. Tyto komponenty jsou realizovány jako abstraktní třídy a jsou integrovány v OOF. Interakce v OOF jsou definovány pomocí tříd, instancí, omezujících podmínek, dědičnosti, polymorfismu a pravidel kompozice [DeBaud93]. OOF může být použit pro zachycení předepsané softwarové architektury. OOF může být výchozím bodem pro celou třídu systémů (předpokládá to ovšem, že je vyvinut na základě dokonalých znalostí domény problému). OOF je velmi mocný koncept a byl použit např. pro vývoj objektově orientovaného operačního systému *Choices* [Campbell93, Madany92, Campbell92].

[19] Překlad by mohl být např. „základní struktura“, „rámec“, atd. V textu bude dále používaná zkratka OOF.

7. Minimální hranice chápání a naplnění konceptu softwarové architektury

Následuje popsaná minimální hranice chápání a naplnění konceptu softwarové architektury, která je vyžadována minimální dobrou praxí softwarového inženýrství:

1. Musí být vyhotovena definice softwarové architektury. Tato se musí stát součástí dokumentace produktu.
2. Definovaná softwarová architektura musí být aplikována v následném vývoji a údržbě vyvíjeného softwarového produktu.
3. Definice softwarové architektury musí být prováděna osobou (osobami) znalou problematiky aspoň na úrovni tohoto stručného pojednání.
4. Definice softwarové architektury musí minimálně obsahovat:
   a) definici realizace a reprezentace možných typů dekompozičních jednotek na všech úrovních dekompozice;
   b) definici realizace a reprezentace možných typů vzájemných vazeb mezi možnými typy dekompozičních jednotek;
   c) pravidla pro použití a omezující podmínky na vzájemnou kombinaci 4.a) a 4.b);
   d) procesní pohled vzhledem k 4.a) a 4.b); má-li smysl;
   e) fyzický pohled; má-li smysl;
   f) obsahuje-li bod 4.a) až 4.e) prohřešek oproti konceptu ADT, konceptu zapouzdření, či jakémukoli jinému elementárnímu konceptu softwarového inženýrství, je třeba to výslovně uvést, vysvětlit a zabezpečit proti chybnému použití.
5. Definice softwarové architektury musí být vhodná vzhledem k požadavkům na vyvíjený softwarový produkt.
6. Definice softwarové architekturu musí být před aplikací přezkoumána. Výsledky přezkoumání musí být uchovávány i s definicí softwarové architektury.

8. Závěr

Na závěr lze konstatovat, že téma softwarové architektury zajišťuje příslušnou pozornost důležitým problémům, které doprovázejí softwarové inženýrství od jeho počátku (třeba ne tolik zdůrazňované). Problematice softwarové architektury je třeba při projektu věnovat patřičnou pozornost aspoň na minimální hranici jejího chápání.

9. Reference

Abowd97 Abowd, G., Bass, L., Clements, P., *et al*. *Recommended Best Industrial Practice for Software Architecture Evaluation*. CMU/SEI-96-TR-025, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, January 1997.

Academia95 Petráčková, V., Krous, J., a kol. *Akademický slovník cizích slov*. Academia, nakladatelství AV ČR, Praha. 1995.

Andrews83 Andrews, G., and Schneider, F. „Concepts and Notations for Concurrent Programming," In *Computing Surveys*, Vol. 15, No. 1, March 1983.

Andrews91 Andrews, B. *Concurrent Programming: Principles and Practice*. The Benjamin Cummings Publishing Company, Inc., Redwood City, CA, 1991.

Barbacci97 Barbacci, M., Klein, M., and Weinstock, Ch. *Principles for Evaluating the Quality Attributes of a Software Architecture*. CMU/SEI-96-TR-036, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, March 1997.

Batory92 Batory, D., and O'Malley, S. *The Design and Implementation of Hierarchical Software Systems With Reusable Components*. (To Appear, ACM Transactions on Software Engr. and Methodology, October 1992), Department of Computer Sciences, The University of Texas, Austin, TX, 1992.

Boehm95 Boehm, B. „Engineering Context (for Software Architecture)," Invited talk, *First International Workshop on Architecture for Software Systems*. Seattle, Washington, April 1995.

Brooks95 Brooks, F. *The Mythical Man-Month - Essays on Software Engineering, Anniversary Edition*. Addison-Wesley, Reading, MA, 1995.

CACM9610 CACM. *Software Patterns, In Communications of the ACM*. Vol. 39, No. 10, October 1996.

Campbell92 Campbell, R., Islam, N., and Madany, P. „*Choices*, Frameworks and Refinement," In *Computing Systems*, Vol. 5, No. 3, 1992.

Campbell93 Campbell, R., and Islam, N. *Choices A Parallel Object-Oriented Operating System*. The MIT Press, 1993.

Chastek96 Chastek, G., and Brownsword, L. *A Case Study in Structural Modeling*. CMU/SEI-96-TR-035, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, December 1996.

Clements96 Clements, P., and Northrop, L. *Software Architecture: An Executive Overview.* CMU/SEI-96-TR-003, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, February 1996.

Clements96b Clements, P. *Coming Attractions in Software Architecture*. CMU/SEI-96-TR-008, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, January 1996.

Coad92 Coad, P. „Object-Oriented Patterns," In *Communications of the ACM*. Vol. 35, No. 9, September 1992

Dean93 Dean, R., and Armand, F. *Data Movement in Kernelized Systems*. Technical Report, Carnegie Mellon University, CHORUS Systemes CEDEX-FRANCE, 1993.

DeBaud93 DeBaud, J. *From Domain Analysis to Object Oriented Framework, A Reuse Oriented Software Engineering Methodology*. A Ph.D. Thesis Proposal Presented to The Academic Faculty, Georgia Institute of Technology, December 1993.

Dijkstra68 Dijkstra, E. „The Structure of the 'T.H.E.' Multiprogramming System," In *Communications of the ACM*, Vol. 11, No. 5, May 1968.

Garlan95 Garlan, D., and Perry, D. „Introduction to the Special Issue on Software Architecture (Guest Editorial)," In *IEEE Transactions on Software Engineering*. April 1995.

Gien92 Gien, M., and Grob, L. „Micro-kernel Based Operating Systems: Moving UNIX onto Modern Systems Architectures," In *Proceedings of the UniForum'92 Conference*. San Francisco, January 1992.

ISO9000-3 ISO 9000-3:1991 *Quality management and quality assurance standards - Part 3: Guidelines for the application of ISO 9001 to the development, supply and maintenance of software.* 1991.

Kazman94 Kazman, R., and Bass, L. *Toward Deriving Software Architectures From Quality Attributes.* CMU/SEI-94-TR-10, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, August 1994.

Kosvancova94 Košvancová, V. „Analysis of the Synchronization of Objects with Respect to Inheritance," In *Proceedings of CTU Seminar 94* - Part B, Czech Technical University, Praha, January 1994.

Lane90 Lane, T. *Studying Software Architectures Through Design Spaces and Rules.* CMU/SEI-90-TR-18, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, November 1990.

Loepere92 Loepere, K. *Mach 3 Kernel Principles*. Open Software Foundation and Carnegie Mellon University, July 1992.

Madany92 Madany, P. *An Object-Oriented Framework for File Systems*. Ph.D. thesis, Report No. UIUCDCS-R-92-1751, University of Illinois at Urbana-Champaign, Urbana, IL, June 1992.

Mey92 Mey, V. „Experimenting with Component-Oriented Software Development," In *Object Frameworks.* (ed. Tsichritzis). Centre Universitaire d'Informatique, Université de Gen`eve, Gen`eve, Switzerland, 1992.

Mukherjee93 Mukherjee, B., Schwan, K., and Gopinath, P. *A Survey of Multiprocessor Operating System Kernels (DRAFT).* GIT-CC-92/05, College of Computing, Georgia Institute of Technology, Atlanta, Georgia, November 1993.

Mukherjee94 Mukherjee, B., and Schwan, K. *Operating Systems for Parallel Machines*. College of Computing, Georgia Institute of Technology, Atlanta, Georgia, 1994.

Nierstrasz92 Nierstrasz, O., Gibbs, S., and Tsichritzis, D. „Component-Oriented Software Development," In *Communications of the ACM*, Vol. 35, No. 9, September 1992.

Parnas72 Parnas, D. „On the Criteria for Decomposing Systems into Modules," *In Communications of the ACM*. Vol. 15, No. 12, December 1972.

Parnas74 Parnas, D. „On a 'Buzzword': Hierarchical Structure," In *Proceedings IFIP Congress 74*. North Holland Publishing Company, 1974.

Parnas76 Parnas, D. „On the Design and Development of Program Families," In *IEEE Trans. on Software Engineering*. SE-2, March 1976.

Parnas79 Parnas, D. „Designing software for ease of extension and contraction," In *IEEE Trans. on Software Engineering*. SE-5, March 1979.

Paulk94 Paulk, M. *A Comparison of ISO 9001 and the Capability Maturity Model for Software*. CMU/SEI-94-TR-12, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, July 1994.

Paulk95b Paulk, M. „How ISO 9001 Compares with the CMM," In *IEEE Software*. January 1995.

Peterson94 Peterson, S., and Stanley, J. *Mapping a Domain Model and Architecture to a Generic Design.* CMU/SEI-94-TR-8, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1994.

Rajlich77 Rajlich, V., Souček, J., *et al*. „Datové Abstrakce," In *Proceedings of SOFSEM'77*. Výskumné výpočtové stredisko, Bratislava, 1977.

Rumbaugh93 Rumbaugh, J. „Disinherited! Examples of misuse of inheritance," In *JOOP'93*, Vol. 5, No. 3, February 1993.

SEISADef97k) SEI. *What is software architecture?*. <http://www.sei.cmu.edu/technology/architecture/>, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, (stav duben 1997.

Sha95 Sha, L., Rajkumar, R., and Gagliardi, M. *A Software Architecture for Dependable and Evolvable Industrial Computing Systems.* CMU/SEI-95-TR-005, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, July 1995.

Shaw94 Shaw, M., and Garlan, D. *Characteristics of Higher-level Languages for Software Architecture.* CMU/SEI-94-TR-23, School of Computer Science and Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, December 1994.

Shröder92a Shröder-Preikschat, W. „Scalable Operating System Design," In *PEACE - The Evolution of a Parallel Operating System*. Arbeitspapiere der GMD 646, Gesellschaft für Mathematik und Datenverarbeitung mbH, Germany, 1992.

Shröder92b Shröder-Preikschat, W. ed. *PEACE - The Evolution of a Parallel Operating System*. Arbeitspapiere der GMD 646, Gesellschaft für Mathematik und Datenverarbeitung mbH, Germany, 1992.

Smolik95 Smolík, T. „An Object-Oriented File System - an Example of Using the Class Hierarchy Framework Concept," In *Operating Systems Review*. ACM SIGOS, Vol. 29, No. 2, ACM Press, NY, April 1995.

Tanenbaum87 Tanenbaum, A. *Operating Systems: Design and Implementation*. Prentice Hall, 1987.

Tsichritzis92 Tsichritzis, D., Nierstrasz, O., and Gibbs, S. „Beyond Objects: Objects," In *Object Frameworks*. (ed. Tsichritzis). Centre Universitaire d'Informatique, Université de Gen`eve, Gen`eve, Switzerland, 1992.

UML1097 Booch, G., Rumbaugh, J., Jacobson, I., *et al*. *Unified Modeling Language*. version 1.0, Rationale Software Corporation, CA, 13 January 1997.

Webster94 *Webster's encyclopedic unabridged dictionary of the English language*. Gramercy Books, dilithium Press, Ltd., NY, 1994.

Yokote92 Yokote, Y. „The Apertos Reflective Operating System: The Concept and Its Implementation," In Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications, 1992.

Bass98 Bass, L., Clemments P., et al. *Software Architecture in Practice*. The SEI Series in Sooftware Engineering, Addison-Wesley, 1998.

Clements02a Clements, P. et al. *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley, 2002.

Fowler03 Fowler, M., et al. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003.

Clements02b Clements, P. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 2002.

Hohpe04 Hohpe, G. *Enterprise Integration Patterns : Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley 2004.

Rozanski05 Rozanski, N. et al. *Software System Architecture: Working with Stakeholders Using Viewpoints and Perspectives*. Addison-Wesley, 2005.

Chevance05 Chevance, R. *Server Architectures: Multiprocessors, Clusters, Parallel Systems, Web Servers, and Storage Solutuions*.Elsevier Digital Press, 2005.