VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

ZPŘÍSTUPNĚNÍ SYSTÉMU PROALPHA EXTERNÍMU KLIENTOVI

EXTRENAL ACCESSS TO THE PROALPHA SYSTEM

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. ONDŘEJ SUCHÝ**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. MARTIN KRČMA**
SUPERVISOR

BRNO 2017

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačových systémů Akademický rok 2016/2017

Zadání diplomové práce

Řešitel: **Suchý Ondřej, Bc.**
Obor: Bezpečnost informačních technologií
Téma: **Zpřístupnění systému proAlpha externímu klientovi**
Extrenal Access to the proAlpha System
Kategorie: Informační systémy

Pokyny:

1. Seznamte se s Enterprise Resource Planning (ERP) systémy, především se systémem proAlpha.
2. Prozkoumejte možnosti vnějšího přístupu k tomuto systému.
3. Zvolte si jednu z částí systému a navrhněte způsob vnějšího přístupu k této části.
4. Navržené řešení realizujte.
5. Vytvořte jednoduchého klienta pro přístup do dané části systému a demonstrujte na něm funkčnost vytvořeného řešení.
6. Zhodnoťte dosažené výsledky a navrhněte další možné směřování práce.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Splnění bodů 1 až 3 zadání.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Krčma Martin, Ing.**, UPSY FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 24. května 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačových systémů a sítí
612 66 Brno, Božetěchova 2

L.S.

prof. Ing. Lukáš Sekanina, Ph.D.
vedoucí ústavu

Abstrakt

V této práci je rozebírána problematika ERP systémů a jejich hlavních cílů a principů. V návaznosti na tento rozbor je představen ERP systém proALPHA, který je dále analyzován z pohledu možností přístupu k jeho funkcionalitě z externích klientů, což je také hlavním cílem této práce. Jako část tohoto systému, jejíž funkcionalitu se budeme snažit zpřístupnit, byl zvolen modul materiálového hospodářství. Do něj budeme přistupovat přes modul INWB, který je implementován s využitím nástroje Sonic ESB, jehož problematika je rovněž předmětem této práce. Všech vytyčených cílů se podařilo úspěšně dosáhnout.

Abstract

This work deals with main goals and principles of ERP systems. On the basis of the presented principles it presents the proALPHA ERP system and examines it in order to determine external access possibilities. The external access into the proALPHA system is the main goal of this work. The materials management module has been chosen as the part of the system to be accessed. The access will be realized via proALPHA INWB module based on Sonic ESB which is described as well. All set goals of this work were reached.

Klíčová slova

ERP systém, Enterprise Resource Planning, proALPHA, podniková sběrnice služeb, Java Messaging Service, Sonic, externí přístup

Keywords

ERP system, Enterprise Resource Planning, proALPHA, Enterprise Service Bus, Java Messaging Service, Sonic, external accesss

Citace

SUCHÝ, Ondřej. *Zpřístupnění systému proALPHA externímu klientovi.* Brno, 2017. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Krčma Martin.

Zpřístupnění systému proALPHA externímu klientovi

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci na téma Zpřístupnění systému proALPHA externímu klientovi vypracoval samostatně pod vedením pana Ing. Martina Krčmy. Další informace mi poskytli zaměstnanci firmy SPC solutions, spol. s r.o. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Ondřej Suchý
9. května 2017

Poděkování

Na tomto místě bych chtěl poděkovat Ing. Ivanu Pagáčovi za konzultace praktické části této práce a pomoc při řešení problémů, které se v jejím průběhu objevily, Ing. Martinu Šebestovi za vytvoření zadání práce a Ing. Martinu Hollemu za poskytnutí informací o systému proALPHA z uživatelského pohledu.

Obsah

Kapitola 1

Úvod

Jako klíčovým prostředkem pro správu celopodnikových zdrojů se v současné době uplatňují *Enterprise Resource Planning* (ERP) systémy. Základy těchto systémů byly položeny již v 60. letech minulého století, v podobě, v jaké je známe dnes, se však začaly objevovat až na počátku 90. let. [11, 31] Na přelomu tisíciletí pak ERP systémy začaly výrazněji pronikat i do oblasti malých a středních podniků. [16] Jak ukazuje například studie [4], pro malé podniky je velmi důležitým požadavkem možnost přistupovat k informacím a funkcionalitě systému odkudkoliv včetně mobilního a webového rozhraní. Vzdálený přístup do ERP systému je hlavním předmětem této práce.

Cílem jejího zadání je nastudovat a představit problematiku ERP systémů a na základě získaných poznatků zanalyzovat ERP systém proALPHA. Konkrétně je třeba zjistit, jaké jsou možnosti externího přístupu do tohoto systému a vybrat vhodnou část, jejíž funkcionalitu se pokusíme zpřístupnit. Další náplní práce je pak navrhnout implementaci externího přístupu do vybrané části, realizovat ji a demonstrovat její funkčnost jednoduchým testovacím klientem. Iniciátorem tohoto zadání je firma SPC solutions, spol. s r.o., která se dlouhodobě stará o českou a slovenskou lokalizaci systému proALPHA a která vznesla požadavek na jeho analýzu z hlediska možností zpřístupnění jeho funkcionality externím klientům.

Přehled vývoje ERP systémů, jejich základních konceptů a klíčových cílů bude rozebrán v kapitole 2, na kterou navazuje kapitola 3 zabývající se podrobněji systémem proALPHA. V té je rovněž identifikována vhodná oblast pro otestování možností externího přístupu do systému a jeho preferovaná cesta.

Kapitoly 4 a 5 pak popisují mechanismy a principy, na kterých je vybraná cesta externího přístupu založená. Konkrétně se jedná o problematiku ESB a EMS. O jejich implementaci se v tomto případě stará nástroj Sonic ESB respektive SonicMQ. Rozbor relevantních technologií doplňuje kapitola 6 prezentující hlavní implementační jazyk této práce OpenEdge ABL.

Na základě všech těchto poznatků následně v rámci kapitoly 7 navrhneme celkovou podobu systému pro externí přístup a definujeme rozhraní mezi jeho jednotlivými částmi. Implementačními detaily navrženého systému se budeme zabývat v kapitolách 8 a 9. Dosažené výsledky a směr, kterým by se mohl případný další vývoj této práce ubírat budou shrnuty v její závěrečné části v kapitole 10.

Kapitola 2

ERP systémy

Aby byl podnik schopný čelit výzvám moderního, rychle se měnícího ekonomického prostředí a mohl být konkurencí pro ostatní podniky na trhu, je nutné, aby všechny jeho části fungovaly co nejefektivněji, nejspolehlivěji a s minimálními náklady. Jedině tehdy totiž podnik může nabízet v dostatečně krátkém čase kvalitní a levné služby svým zákazníkům.

Tohoto stavu docílíme maximálním využitím potenciálu veškerých zdrojů, které má podnik k dispozici, a optimalizací jeho běžných obchodních procesů.[1] Právě za tímto účelem vznikly komplexní podnikové informační systémy, pro které se používá označení ERP (*Enterprise Resource Planning*). Pojem ERP definuje [8] následovně:

> „**ERP** je charakterizován jako typ aplikačního software, který umožňuje řízení a koordinaci všech disponibilních podnikových zdrojů a aktivit. Mezi hlavní vlastnosti ERP patří schopnost automatizovat a integrovat klíčové podnikové procesy, funkce a data v rámci celé firmy.“

Kvůli výše uvedeným požadavkům během posledních desetiletí firmy upustily od svých starých informačních systémů a začaly místo nich využívat ERP systémy integrující klíčové aktivity podniku. ERP systémy se využívají ke správě všech podnikových dat, díky čemuž mohou poskytovat aktuální informace tehdy, kdy je jich třeba. [5]

Cílem této kapitoly je podrobněji představit pojem ERP systému, udělat stručný přehled jeho historického vývoje a prezentovat jeho klíčové prvky, základní úlohy a výhody včetně rizik spojených s jeho nasazením.

2.1 Historický vývoj

V této části bude probrán průběh vývoje informačních systémů, které měly za úkol starat se o zdroje podniku. Přehled zachycuje období od 60. let minulého století, kdy dominovaly systémy zajišťující plánování a správu výroby, až po současné ERP a ERP II systémy, pro něž je typická vysoká míra integrace dílčích podnikových systémů.

2.1.1 60. a 70. léta

Jak již bylo naznačeno v úvodu této sekce, ERP systémům předcházely systémy orientované spíše na plánování a správu výroby než podnikových zdrojů obecně. [11] Tento software

[1] *Obchodní proces* chápeme jako kolekci aktivit s jedním nebo více vstupy, které jsou transformovány na výstupy mající nějakou hodnotu pro zákazníka. Zákazník může být buď osoba platící za cílový produkt, nebo pracovník jiného oddělení, který výstup dále zpracovává.

zaznamenal v průběhu 60. a 70. let vývoj od jednoduchých systémů pro sledování zásob až k pokročilejším MRP (*Material Requirement Planning*) systémům. [16]

V období 60. let si mohly společnosti dovolit udržovat poměrně velké množství zásob, aniž by tím klesala jejich konkurenceschopnost. Systémy se tedy zaměřovaly především na kontrolu zásob podniku. [31]

Během 70. let se však začalo ukazovat, že tento přístup postupně přestává být přípustný. To vedlo na vznik systémů MRP [31], které umožňovaly vhodně plánovat produkci v dostatečném množství a s tím spojené objednávky zdrojů nutných pro realizaci této produkce. Vytvoření plánu probíhalo na základě analýzy predikce budoucích prodejů. [16]

2.1.2 Zvyšování integrace systémů

Díky zvýšení možností a dostupnosti nových technologií bylo v 80. letech možné provést propojení pohybů zásob s odpovídajícími finančními aktivitami, což vedlo na vznik systémů MRP II (*Manufactoring Resource Planning*). Tyto systémy v sobě zahrnovaly finanční účetnictví a správu financí společně s řízením výroby a správou materiálů. [31] Kromě toho do sebe také začlenily oblast plánování kapacit výrobních zdrojů. [8]

Pokračující technologické pokroky na počátku 90. let umožnily další rozšiřování systémů MRP II. Postupně se do nich začlenily části pro správu a řízení veškerých zdrojů společnosti, jako například lidské zdroje, projektové řízení, materiálové plánování apod. [11, 31] S rostoucím počtem oblastí, které tyto systémy pokrývaly, se pro ně začal zavádět název ERP (*Enterprise Resource Planning*). První společností, která vyvinula software pro ERP systémy, je společnost SAP. [16]

2.1.3 Moderní ERP systémy

Díky velkému počtu oblastí, které ERP pokrývají, se již tyto systémy nemusí orientovat pouze na výrobní podniky, ale i na množství jiných průmyslových odvětví. [11] To znamená, že nemusí být nutně použity jen pro výrobní společnosti, ale může je použít kterákoliv firma, která chce zvýšit svou konkurenceschopnost efektivním řízením svých zdrojů. [31]

Od svých předchůdců se odlišují silnější integrací výrobních a finančních modulů, díky čemuž je možné lépe vyhodnocovat ekonomické efekty a případná rizika zakázek. Dále se také projevuje lepší řízení personálních zdrojů, řízení majetku a propojení výrobního a finančního plánování. [8]

K velkému nárůstu počtu implementací[2] ERP systémů došlo na konci 90. let, a to mimo jiné i kvůli blížícímu se problému Y2K.[3] Aby firmy tento problém vyřešily, mohly buď vynaložit nemalé finanční náklady na opravu svých zastaralých systémů, nebo investovat do zavedení moderního ERP systému, který měl navíc potenciál zlepšit správu obchodních procesů společnosti a tím i její konkurenceschopnost. [16, 31]

Ve stejné době se také začali dodavatelé ERP systémů více orientovat na systémy pro malé a střední podniky, jelikož oblast velkých podniků se již postupně zaplňovala. [16]

V posledních letech se můžeme setkat také s pojmem ERP II. Systémy tohoto typu pokračují dále v trendu integrace a kromě jádra tvořeného systémem ERP v sobě zahrnují i množství dalších prvků jako například Business intelligence, e-Businnes, workflow apod. ERP II jsou typicky určeny pro střední a větší podniky. [8]

[2] V kontextu ERP systémů rozumíme pod pojmem *implementace* nasazení systému do provozu v konkrétním podniku.

[3] Systémy ze 70. a 80. let z důvodu optimalizace používaly pro reprezentaci roku jen jeho dvě poslední číslice. V roce 2000 by tak došlo k přetečení této hodnoty vedoucí na nekonzistenci v systémech.

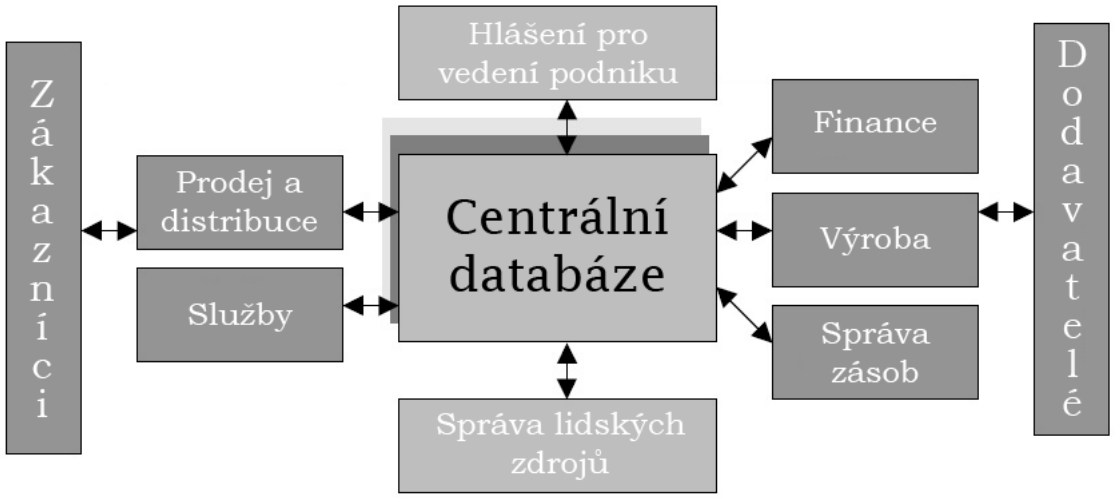
2.2 Principy ERP systémů

V době, kdy ERP systémy nebyli tak rozšířené, jako je tomu v posledních letech, měla většina společností neintegrované informační systémy, které se staraly pouze o činnost konkrétních oblastí podniku, jako například marketing, produkce atd. Každý z těchto systémů fungoval nezávisle na ostatních s vlastním hardwarem, softwarem a metodami zpracování dat. [16, 11]

Nevýhodou tohoto přístupu je absence kvalitního sdílení dat napříč celým podnikem vedoucí na neefektivitu fungování společnosti, přestože všechny systémy pracují kvalitně a spolehlivě ve své individuální oblasti. Aby bylo možné sdílet data mezi jednotlivými odděleními, bylo typicky nutné vyexportovat data z jednoho systému a znovu je zadat do dalšího. [16]

Tento proces je jednak časově náročný, navíc také zvyšuje pravděpodobnost vzniku chyby a tudíž nekonzistence v systému. Automatizací převodu dat mezi systémy je sice možné riziko chyby snížit, změny ale obvykle probíhají periodicky a aktuální data tudíž nejsou dostupná v dostatečně krátkém čase. [16] Dále například není možné sledovat průchod zákaznického požadavku napříč jednotlivými odděleními, což opět vede na nutnost opakovaného zadávání stejné informace na různých úrovních procesu do často neslučitelných databází. Tím se snižuje efektivita podnikových dat a operací a naopak stoupá pravděpodobnost chyby a nekonzistence v datech. [8]

Jak již bylo naznačeno v sekci 2.1, ERP systémy se snaží sjednotit dílčí podnikové funkce na úrovni celého podniku do jedné aplikace se společným zdrojem dat (viz obrázek 2.1). Programy v nich integrované vyhovují potřebám jednotlivých oddělení a pracovníků. Jejich úkolem je tedy vytvořit jedinou konzistentní aplikaci, která bude efektivně poskytovat podporu běžným podnikovým procesům. [12, 8]



Obrázek 2.1: Zjednodušená architektura ERP systému. (Převzato z [2])

ERP pokrývá techniky pro správu podniku jako celku z pohledu efektivního nakládání s podnikovými zdroji za účelem zvýšení efektivity celé organizace. Jsou navrženy tak, aby modelovaly a automatizovaly základní obchodní procesy s cílem integrace informací napříč podnikem a eliminovaly nákladné propojení mezi jednotlivými systémy, u kterých se

vzájemná komunikace nikdy nepředpokládala. [11] Pro ERP systémy je charakteristický vysoký počet uživatelů, typicky jsou transakčně orientované a téměř výhradně pracují nad relační databází. [8]

Při nasazování ERP systému je velmi důležitá jeho *customizace* neboli přizpůsobení. ERP systémy jsou obecné aplikace a je tedy nutné je přizpůsobit potřebám konkrétních ekonomických subjektů. Jako příklady customizace uvádí [8] mimo jiné

- úpravy uživatelského rozhraní a funkcí,
- nastavení defaultních hodnot pro jazyk, měnu apod.,
- úpravy a naplnění číselníků,
- přizpůsobení standardních výpočtů.

Podle [8] v našich podmínkách obvykle rozlišujeme následující typy ERP systémů:

- **velké systémy** – pro podniky s více než 500 zaměstnanci a obratem nad 800 mil. Kč,
- **střední systémy** – pro podniky s 25 – 500 zaměstnanci a obratem 100 – 800 mil. Kč,
- **malé systémy** – pro podniky do 25 zaměstnanců a obratem do 100 mil. Kč.

2.2.1 Modulární architektura

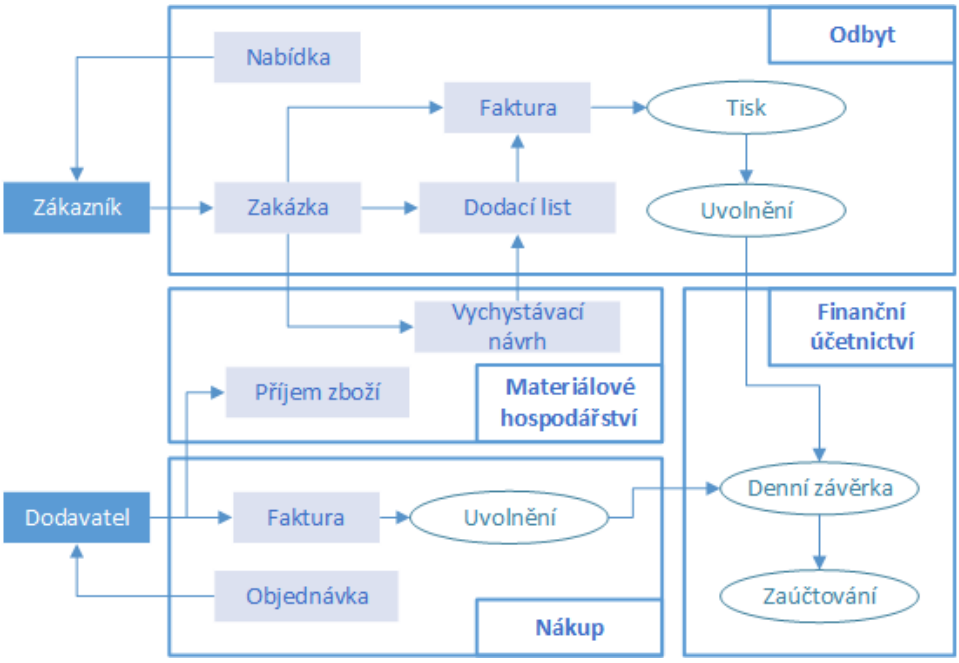
Aby byla v ERP systému zachována rovnováha mezi provázaností a nezávislostí jednotlivých částí, je celý systém založen na modulární struktuře. [8] Softwarové moduly jsou individuální programy, které lze zakoupit, instalovat a provozovat samostatně, všechny moduly ale budou k práci využívat data ze sdílených databází. [16]

Díky tomu je možné lépe vyhovět potřebám konkrétních podniků a nekupovat nepotřebné moduly. Je zbytečné, aby například firma, která nevyrábí žádné zboží, měla instalovaný výrobní modul apod. [8]

Moduly ERP systému si mezi sebou předávají data buď přes sdílené databáze, nebo prostřednictvím datových vstupů a výstupů. Důsledkem této komunikace je, že vykonání operace v jednom modulu přímo vede na automatické generování příslušné operace v jiném modulu. Takovéto operace jsou pak vzájemně konzistentní a jejich důsledky jsou dohledatelné napříč systémem. [8] Tento přístup je demonstrován na obrázku 2.2.

Jako typické příklady modulů ERP systému si uvedeme následující zástupce: [8, 12, 16]

- **Modul výroby** – udržuje informace o produkci, provádí její plánování a zaznamenává produkční aktivity. Obvykle nebývá omezený jen na jednoduché operace jako výroba do zásoby nebo výroba na zakázku, ale umožňuje kombinovat škálu výrobních a plánovacích metod v rámci operace a dosáhnout tak vysoké flexibility.
- **Modul finančního účetnictví** – shromažďuje data z oblasti financí, zaznamenává transakce do hlavní účetní knihy, generuje finanční výkazy pro další použití atd. Finanční data jsou důležitá pro strategické plánování.
- **Modul řízení zásob** – optimalizuje všechny nákupní procesy, umožňuje automatizovanou evaluaci dodavatelů, snižuje náklady na nákupy a uskladnění, eviduje příjmy, výdaje a stav zásob.



Obrázek 2.2: Typický tok dat mezi moduly ERP systému při realizaci styku se zákazníkem a dodavatelem. Jednotlivé dokumenty reprezentované modrošedým obdélníkem jsou systémem vytvářeny na základě již existujících dokumentů. Operace nad dokumenty jsou znázorněny bílým oválem. Pojmenování modulů, dokumentů a operací uváděných na obrázku koresponduje s pojmenováním v systému proALPHA (viz kapitola 3).

- **Modul lidských zdrojů** – usnadňuje nábor nových zaměstnanců s vhodnými schopnostmi a jejich zaškolení, spravuje zaměstnance podniku a jejich data, kontroluje cestovní výdaje, zaměstnanecké výhody a platy, plánování směn a další.
- **Modul prodeje a distribuce** – zpracovává objednávky, stará se o procesy spojené s objednávkami a jejich dodáním, spravuje informace o zákaznících.
- **Modul správy zařízení** – zajišťuje plánování preventivních údržbářských prací za účelem minimalizace poruch zařízení v podniku.

ERP architektura navíc kromě výše uvedených modulů, které se taky označují jako aplikační, obvykle nabízí i řadu dalších podpůrných modulů. Mezi tyto moduly mohou patřit například dokumentační moduly obsahující dokumentaci jednotlivých aplikačních modulů, implementační moduly pomáhající při nasazení systému v daném podniku, moduly pro customizaci systému nebo vlastí vývojové prostředí. [8]

2.2.2 Přínosy a rizika ERP

ERP systémy pomáhají společnosti centralizováním informací z různých geografických lokací lépe kontrolovat její zásoby, nákupy, finance, výrobu a lidské zdroje. [5] Dále nabízejí ucelený pohled na podnik a jeho fungování, který pokrývá všechny jeho funkce a oddělení. [31]

Díky integraci jednotlivých oblastí podniku získáváme optimalizované obchodní procesy, které jsou méně nákladné než v případě neintegrovaných systémů. [16] Zavedení ERP

systémů rovněž vede ke snížení ceny práce a zvýšení kvality zákaznických služeb vedoucí k minimalizaci ztráty zakázek a celkovému zvýšení obratu. [12]

ERP systémy jsou také schopné provádět simulaci různých scénářů využití dostupných kapacit a zdrojů, což umožňuje udržovat stav zásob a prostoje při práci výrobních zařízení na minimální úrovni. [11] Díky tomu můžeme pozorovat pokles nákladů na skladování a logistiku a snížení pravděpodobnosti zastarávání nebo poškození zásob. [5, 12]

Z nehmotných přínosů můžeme zmínit, že díky společné databázi ERP systému není nutné duplikovat soubory a udržovat redundantní záznamy. Finanční výkazy mohou být navíc snadno přizpůsobené pro konkrétní účely a odhady jsou založeny na detailních ERP kalkulacích. Vysoká integrace a schopnost automaticky aktualizovat data mezi souvisejícími činnostmi vedou také na zlepšené rozhodování. [11, 12]

Na druhou stranu je ale nutné zmínit, že implementace ERP systému je velmi nákladná, a to jak finančně, tak i časově. Hlavní faktory ovlivňující cenu jsou podle [5, 16] následující:

- rozsah ERP softwaru, který je závislý na velikosti společnosti,
- pořízení hardwaru, který odpovídá potřebám komplexního ERP systému,
- náklady na konzultanty a analýzy,
- zaškolení pracovníků a následná podpora.

Kromě toho, výběr vhodného systému pro danou organizaci a jeho efektivní implementace jsou náročné procesy s vysokou pravděpodobností neúspěchu. [12] Vždy je nutné mít na paměti, že ne všechny organizace získají od stejného ERP systému stejné výhody a ne každý systém je dostatečně vhodný pro každou firmu. Jako časté příčiny neúspěchu si můžeme uvést tyto faktory:

- zvolení nevhodného ERP systému, jehož technologické schopnosti neodpovídají existujícím obchodním procesům a procedurám dané společnosti,
- lidský faktor, tj. nekvalitní implementační tým a vedení projektu, zaměstnanci neochotní přijmout nový systém,
- nedostatečné plánování a řízeni implementace,
- nedostatek podpory ze strany řízení podniku vedoucí na nekvalitní analýzu a přepracování obchodních procesů,
- neochota přizpůsobit stávající obchodní procesy a organizační strukturu podniku zvolenému ERP systému. [5, 31]

Kapitola 3

Systém proALPHA

ProAPLHA je ERP systém vyvíjený německou společností proALPHA Business Solutions GmbH. Hlavní cílová skupina tohoto systému jsou malé a střední podniky v oblastech výroby, obchodu a služeb. [1] Hlavní jádro systému je napsané v jazyce OpenEdge ABL, další externí komponenty jsou psané v jazycích C# a Java.

Cílem této kapitoly je seznámit čtenáře s podobou tohoto systému, jeho základními funkcemi a operacemi a představit jeho klíčové moduly. Z těchto modulů také určíme jeden, který v práci dále poslouží jako cílová oblast pro externí přístup do systému. V závěru kapitoly také stručně prodiskutujeme preferovaný způsob přístupu do vybrané části.

Jelikož firma SPC solutions nemá přístup k vývojářské dokumentaci systému proALPHA a nebyly nalezeny ani jiné dostupné a dostatečně kvalitní materiály, musely být při psaní této kapitoly využity jiné alternativy. Jako zdroj informací pro tuto kapitolu sloužily kromě oficiálních webových stránek systému [1] převážně osobní konzultace s pracovníky SPC, rozpracovaná uživatelská dokumentace firmy SPC a nápověda v uživatelském rozhraní systému proALPHA.

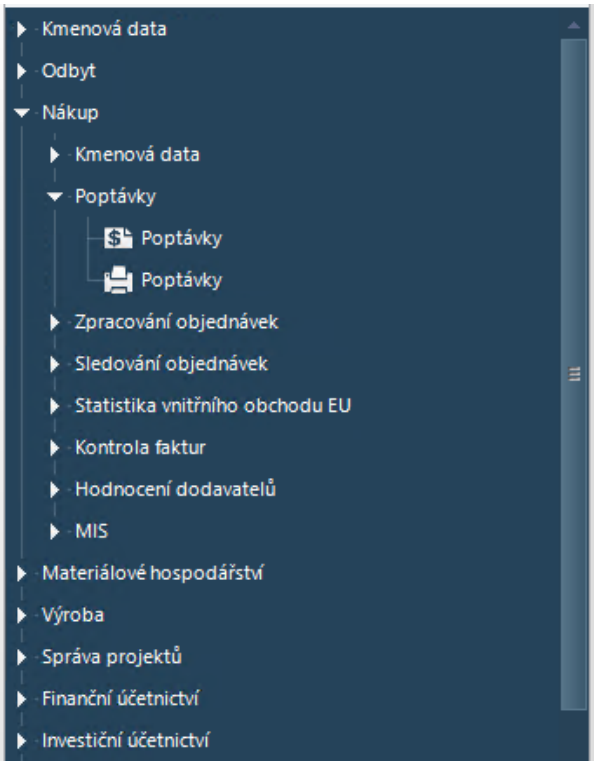
3.1 Uživatelské rozhraní

Ještě než se budeme podrobněji zabývat jednotlivými částmi a funkcionalitou systému, ukážeme si základní prvky jeho uživatelského rozhraní. Po zadání přihlašovacích údajů a spuštění proALPHA můžeme na levé straně hlavní obrazovky najít navigační panel (obrázek 3.1). Jednotlivé části systému jsou v tomto panelu organizovány do stromové struktury, přičemž nejvyšší úroveň reprezentuje jednotlivé moduly, nižší úrovně oblasti, které pod ně spadají, případně data, která jsou relevantní v dané oblasti.

Zbylá část obrazovky slouží ke zobrazení aktuálně otevřených oken umožňujících prohlížení a úpravu dat. Při práci s dokumenty v systému proALPHA se nejčastěji setkáme se dvěma základními typy oken:

- hlavní okno (obrázek 3.2),
- závislé okno (obrázek 3.3).

Hlavní okno slouží ke zobrazení jednoho záznamu z databáze. Typicky se jedná o dokument jako je faktura, dodací list, zakázka aj. V horní části okna jsou uvedeny základní údaje o dokumentu. Mezi ně spadá například zákazník, který je s dokumentem spojen, datum vytvoření apod. Doplňující údaje pak nalezneme ve spodní části okna rozdělené do přepínacích panelů.



Obrázek 3.1: Panel pro navigaci v systému proALPHA.

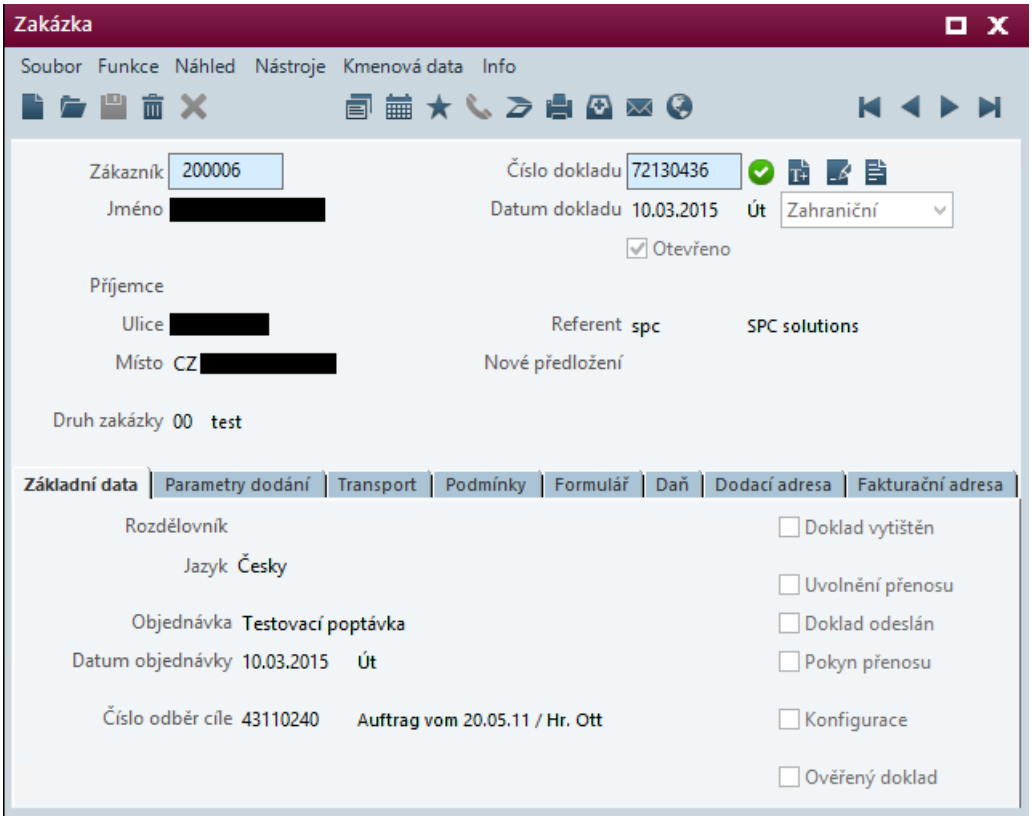
Pro zobrazení závislých záznamů, jako například jednotlivých položek zakázky, slouží závislé okno. V horní části okna se nachází tabulka s jednotlivými položkami a základními údaji, konkrétně objednané množství, cena nebo termín dodání, ve spodní části pak opět můžeme nalézt detaily vybrané položky.

V některých případech se můžeme v tomto okně setkat i se strukturovanými položkami, kde rozlišujeme hlavní a závislé řádky položek. Jako příklad uveďme položky vychystávacích návrhů, kde hlavní řádek reprezentuje skladovou položku, která má být vychystána v požadovaném množství. Vychystávání může probíhat postupně, a položky lze odebírat z různých skladů. Každý závislý řádek tedy představuje záznam o jednom dílčím vychystání části požadovaného množství.

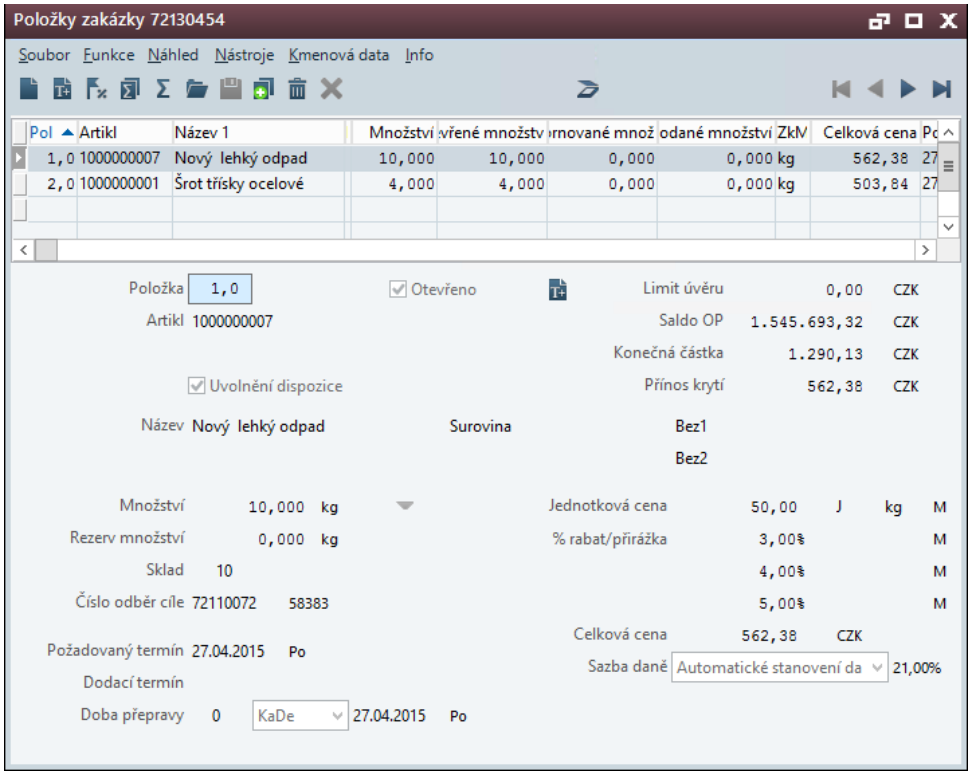
Obě okna sdílejí také společné menu s následujícími položkami:

- **Soubor** – obecné operace se systémem a záznamy,
- **Funkce** – přístup k podřízeným nebo nadřízeným záznamům,
- **Náhled** – přepínání podrobností zobrazovaných ve spodní části okna,
- **Nástroje** – operace, které jsou specifické pro daný typ záznamu, například fakturu,
- **Kmenová data** – přístup do číselníků a k nadřízeným záznamům,
- **Info** – přístup k dalším datům doplňujícím informace zobrazované v okně.

Kromě menu máme v záhlaví každého okna k dispozici také *toolbar* s tlačítky pro navigaci mezi záznamy v databázi, zpřístupnění editace záznamu, jeho smazání a další.



Obrázek 3.2: Hlavní datové okno systému proALPHA.



Obrázek 3.3: Okno se závislými daty systému proALPHA.

3.2 Moduly a části systému

Díky využití modulární struktury, rozebírané v části 2.2.1, umožňuje proALPHA přizpůsobení systému pro potřeby konkrétního podniku, zároveň však dodržuje koncept integrace jednotlivých částí, čímž zajišťuje konzistentní tok dat napříč podnikem. V této části probereme jednotlivé moduly systému a stručně vysvětlíme, jakou v něm hrají roli.

3.2.1 Kmenová data

Kmenová data představují typ dat, který je naprosto nezbytný pro běh celého systému. Pokud nebudou správně vyplněna, nebude možné provádět požadované operace. Přístup ke kmenovým datům je možný buď přes příslušnou položku v navigaci systému nebo přes jednotlivé moduly, u kterých jsou vždy k dispozici odkazy na data, která se jich týkají.

Mezi kmenovými daty obvykle najdeme především číselníky, tj. uspořádané seznamy záznamů, ve kterých je každému záznamu přiřazena identifikace. Konkrétně se jedná o databázové tabulky, ve kterých jsou uloženy

- státy,
- měny,
- číselné řady,
- banky a banovní spojení,
- daňové klíče,
- ceníky
- a monoho dalších.

Kromě výše uvedených si ještě z kmenových dat zvlášť vyzvedněme seznamy

- zákazníků,
- dodavatelů,
- artiklů (zboží a služby),
- pracovníků.

3.2.2 Finanční účetnictví

Modul finančního účetnictví má na starosti zaznamenávání obchodních transakcí, správu informací o bankovních spojeních apod. Dále také umožňuje generovat hlášení v podobě rozvahy a výkazu zisků a ztrát. Zároveň poskytuje všechny nutné prostředky pro strategické plánování.

Zaúčtování faktury z odbytu nebo nákupu typicky probíhá tak, že nejprve provedeme její tisk a následně uvolnění v příslušném modulu. Uvolněná faktura je připravena k zaúčtování a po provedení denní závěrky pak z faktury vzniká otevřená položka. Ta říká, že nám zákazník dluží peníze za vystavenou fakturu nebo naopak dluží peníze podnik dodavateli za přijaté zboží. Otevřenou položku uspokojíme například zaplacením požadované částky dodavateli.

3.2.3 Odbyt

Odbytový modul se stará prakticky o všechny operace týkající se styku se zákazníky. Je možné v něm vytvářet zákazníkům nabídky, na základě nabídek zadávat zakázky a dodací listy a pro ně pak vystavovat faktury. V případě vráceného zboží je zde také možné generovat dobropis. Dále je v tomto modulu možné předpovídat budoucí odbyt artiklů.

V průběhu celého procesu obsluhy zákazníka jsou data pro dokumenty postupně přebírána z existujících dokumentů jak zachycuje obrázek 2.2. Díky tomu tedy lze přes jednotné rozhraní spravovat data z oblasti prodejů od nabídek až po faktury bez nutnosti opakovaného zadávání stejných údajů, což eliminuje možnost vzniku chyby při přepisu. Tyto údaje navíc mohou být později použity také ve výrobním modulu pro plánování výroby. Další výhodou přebírání dokumentů je možnost vysledovat pro každý dokument jeho předchůdce. Tyto údaje lze získat přes položku *Info* v menu odpovídajícího okna.

3.2.4 Nákup

Jestliže se odbytový modul staral o veškerý kontakt se zákazníkem, pak modul nákupu pokrývá pro změnu všechny aspekty vztahů mezi podnikem a jeho dodavateli. Ti mohou dodávat jak zboží, tak služby. Podobně jako v odbytech, i zde se výrazně uplatňuje přebírání jednotlivých dokumentů, tj. poptávek, objednávek, dodacích listů a faktur.

Další z funkcí tohoto modulu je kvalitativní hodnocení dodavatelů. Hodnocení probíhá na základě uživatelem definovaných skupin ohodnocení. Tyto skupiny jsou definované volně a jejich definice záleží tedy především na konkrétních potřebách uživatele. Jako příklad hodnotícího kritéria můžeme uvést hodnocení na základě dodržování dodacích termínů respektive zpoždění dodávek. Hodnocení probíhá vždy pro období specifikované uživatelem.

3.2.5 Materiálové hospodářství

Modul materiálového hospodářství má primárně za úkol řešit logistické záležitosti podniku. Konkrétně se jedná o

- plánování poptávek pro dodavatele,
- určení množství objednávaných položek,
- uskladnění a transport zboží,
- optimalizovanou správu skladů,
- sledování veškerých pohybů zboží.

Důležitými dokumenty v tomto modulu jsou *vychystávací návrhy* (*staging suggestions*, *picking*). Ty nesou mimo jiné informace o tom, jaké skladové položky byly v jakých množstvích rezervovány k odebrání ze skladu. Nejčastěji se položky ze skladu odebírají, aby mohly být vyexpedovány k zákazníkovi na základě zakázky, může se ale také jednat o odebrání surovin za účelem výroby nebo přeskladnění na jiný sklad.

Práce s těmito dokumenty probíhá stejným způsobem jako s dokumenty zmiňovanými dříve. Na druhou stranu ale vychystávací návrhy nejsou tak silně vázány na ostatní moduly a operace nad nimi mají menší celkové dopady na systém. Díky tomu se jeví tato oblast materiálového hospodářství jako optimální část systému proALPHA pro otestování možností externího přístupu do systému. Další motivací pro využití této oblasti může být i fakt,

že právě při práci s těmito dokumenty vzniká nejintenzivnější potřeba využití mobilních zařízení.

3.2.6 Ostatní moduly

Přestože ERP systém proALPHA nabízí více modulů než jen výše uvedené, nebudeme se jimi v této práci zabývat podrobně, ale uvedeme si pouze následující přehled:

- **Investiční účetnictví** – stará se o hmotný majetek společnosti a jeho odpisy.
- **Vnitropodnikové účetnictví** – umožňuje dělit náklady na střediska podle potřeb podniku.
- **Výroba** – zahrnuje všechny funkce pro potřeby plánování a řízení výroby produktů.
- **MIS** – nabízí manažerský pohled na data bez možnosti úprav.
- **Správa systému** – spravuje šablony a umožňuje z nich generovat formuláře.
- **Správa projektů** – umožňuje definovat a spravovat projekty, jejich strukturu, činnosti, milníky atd.
- **Vyhodnocení** – provádí analýzy odchylek plánovaných činností od skutečnosti.
- **Výměna dat** – viz 3.3

3.3 Možnosti externího přístupu

Interakci systému proALPHA s okolním světem zajišťuje modul výměny dat označovaný také jako *integration workbench* (INWB). Ten v systému plní úlohu *podnikové sběrnice služeb* (ESB), jejíž koncepty budou podrobně rozebrány v kapitole 5. INWB je realizován s využitím softwaru Sonic ESB.

Důvod pro zavedení INWB do systému proALPHA byla optimalizace výrobních plánů, které byly výpočetně náročné a nebylo možné je zpracovávat synchronně. Tento problém byl vyřešen použitím asynchronního *enterprise messaging systému* (viz kapitola 4) právě v rámci modulu INWB. V následujících verzích proALPHA se funkce tohoto modulu rozšířily a v současnosti je možné jej použít například pro tyto operace:

- replikace dat v systému,
- přenos dokumentů jak mezi dvěma instancemi proALPHA, tak mezi externími systémy,
- ověřování adres obchodních partnerů za účelem udržení činnosti podniku v souladu s korporátní compliancí
- a další.

Všechny operace se systémem jsou silně závislé na uživatelském kontextu, který nese informace o přihlášeném uživateli a úrovni jeho oprávnění. Kontext je možné získat mimo jiné s využitím standardních komponent, které poskytuje modul INWB. Tento modul tedy plní také funkci primární cesty pro přístup z vnějšího prostředí do systému proALPHA.

Kapitola 4

Enterprise messaging a JMS

Enterprise messaging systém (EMS) můžeme podle [6] definovat jako asynchronní rozhraní, které umožňuje zasílání zpráv mezi programy. Tyto zprávy jsou uloženy do fronty a následně ve vhodnou dobu zpracovány příjemcem. V prostředí podnikových systémů se setkáváme také s označením *Message-Oriented Middleware* (MOM) [6, 28]. Smyslem EMS je naplnění následujících cílů:

- propojení činnosti existujících systémů, které pracovaly nezávisle na sobě, prostřednictvím výměny zpráv,
- spolehlivé doručení informací z koncových zařízení, jako například z externích senzorů, výrobních strojů v rámci podniku apod.,
- garantované doručení dat ze vzdálených oblastí podniku do jeho centra,
- podávání hlášení pro vedení podniku o obchodních aktivitách v distribuovaných podnikových systémech,
- bezpečný a spolehlivý přenos dat od obchodních partnerů a zákazníků do obchodních portálů. [17]

V této kapitole nejprve v sekci 4.1 představíme základní myšlenky a koncepty využité v EMS, dále se v sekci 4.2 budeme věnovat popisu specifikace JMS a na závěr (sekce 4.3), probereme SonicMQ, konkrétní implementaci JMS.

4.1 Hlavní principy EMS

Enterprise messaging systém je ve své podstatě software, který umožňuje dvěma aplikacím komunikovat prostřednictvím odesílání a přijímaní zpráv bez nutnosti lidského zásahu. Tyto aplikace mohou běžet zcela nezávisle na široké škále hardwarových zařízení. [17]

Hlavní myšlenkou v EMS je asynchronní doručování zpráv přes síť od jedné aplikace ke druhé. Odesilatel tak nemusí čekat na příjemce, až zprávu příjme a zpracuje, ale může ihned po odeslání zprávy pokračovat v činnosti. Na druhou stranu existují také případy, kdy je vyžadována synchronní komunikace. Z tohoto důvodu je důležité, aby systém pro zasílání zpráv dokázal podporovat jak asynchronní, tak synchronní komunikaci. [17, 28]

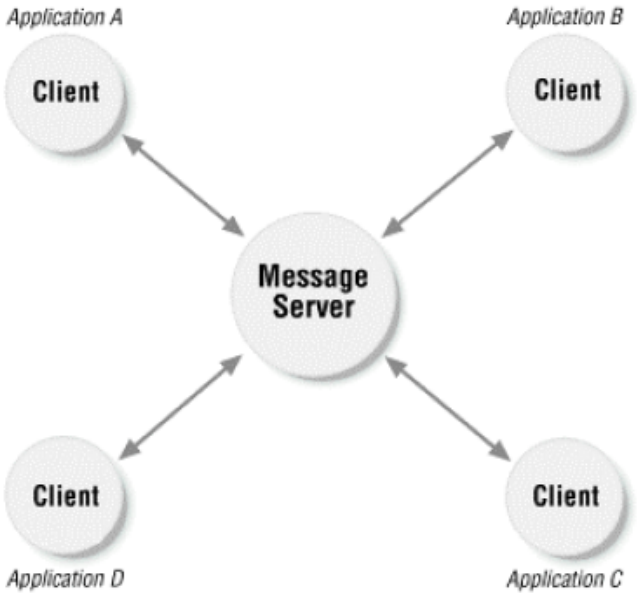
Zpráva má v EMS podobu samostatného balíku dat a směrovacích hlaviček. Data obsažná ve zprávě mohou být jakéhokoliv charakteru. Tyto zprávy obecně informují systém, že v některém jiném systému došlo k události, na kterou je třeba reagovat. O doručení

a správnou distribuci zpráv mezi aplikace se stará MOM. Ten mimo jiné dále zprostředkovává také odolnost proti chybám, vyvažování zátěže, rozšiřitelnost a podporu transakcí pro systémy, které si potřebují spolehlivě vyměňovat velké počty zpráv. [28]

Výměna zpráv v moderních EMS probíhá přes virtuální kanály, pro které se používá označení *destinace* (viz 4.2.2). Pokud chce jedna aplikace komunikovat s jinou, neadresuje konkrétní zprávu přímo své protistraně, místo toho ji zašle do příslušné destinace. Každá aplikace, která má o příjem zpráv zájem, se registruje jako odběratel dané destinace. Odesilatel a příjemce tak nejsou pevně svázáni a mohou přijímat i odesílat zprávy podle vlastní potřeby. [28]

4.1.1 Architektura EMS

V praxi se můžeme setkat s centralizovanou i distribuovanou architekturou. Centralizovaná architektura je závislá na *message serveru* (message router, message broker), který je zodpovědný za doručování zpráv mezi klienty. Klient vždy vidí pouze server, kterému předává zprávy k doručení, nikoliv klienta, se kterým komunikuje. [22, 28] Tato architektura je znázorněna na obrázku 4.1.



Obrázek 4.1: Centralizovaná architektura EMS, pro kterou se typicky používá topologie *hvězda*, tj. centrální server, k němuž jsou připojení všichni klienti. (Převzato z [28])

V případě decentralizované architektury jsou úkoly message serveru delegovány na klientské stanice. Pro doručení zpráv na síťové vrstvě v současné době všechny decentralizované architektury používají IP multicast. [28]

4.2 Java Message Service

Java Message Service (JMS) je API pro EMS vytvořené společností Sun Microsystems. Jedná se spíše o abstrakci rozhraní a tříd nutných pro výměnu zpráv mezi klienty než

o samotný messaging systém. Smyslem tohoto API je poskytnout Java aplikacím standardní programovací model, který je nezávislý na messaging systémech. Aplikace mohou tyto systémy využívat k zasílání notifikací o událostech nebo vzájemné výměně dat. [10, 28] Využívání možností JMS není striktně omezeno jen na Javu. Podporu pro práci s tímto API dnes nalezneme pro většinu běžných programovacích jazyků jako například C/C++, C# nebo i OpenEdge ABL. [17]

JMS API nabízí dva standardní modely pro komunikaci: *publish-and-subscribe* a *point-to-point*. [10, 17, 22, 28] Tyto modely a jejich specifické přínosy budou podrobněji popsány v části 4.2.2.

4.2.1 Pojem zprávy

JMS zprávy mohou buď nést důležitá data, nebo sloužit pouze jako notifikace o události v systému. Každá zpráva má tři základní části:

1. *tělo zprávy* nesoucí její data,
2. *hlavičky* s důležitými metadaty o zprávě,
3. *vlastnosti* nastavované programátorem se stejnou strukturou jako hlavičky.

Hlavičky jsou v principu realizované jako dvojice klíč-hodnota, přičemž klíč je pevně definován. O jejich nastavení se obvykle stará implementace JMS. Typicky v nich nalezneme informace nutné pro směrování zprávy, jejího autora a ID, typ apod. [10, 17, 28]

Na základě hodnot uložených v hlavičkách a vlastnostech zprávy (nikoliv v jejich těle) je také možné zprávy filtrovat s využitím tzv. *message selectorů*. JMS konzument se tak může rozhodnout, jestli danou zprávu zkonzumuje či nikoliv. [17, 28]

Typy JMS zpráv

V JMS rozlišujeme podle typu nesených dat celkem šest typů zpráv, které musí být podporovány každým JMS poskytovatelem. Popis těchto typů nalezneme ve zdrojích [10, 17, 28]. V praxi se však můžeme setkat i s různými rozšířeními JMS o nové typy ze strany konkrétních implementací JMS (viz 4.3). Standardní typy jsou následující:

- **Message** – nejjednodušší typ zprávy, který slouží jako základní rozhraní a ze kterého jsou odvozeny ostatní typy. Zprávy tohoto typu obsahují pouze hlavičky a vlastnosti a mohou tedy být použity jako JMS zprávy bez těla. Obvykle se používají pouze pro notifikaci.
- **TextMessage** – tento typ se používá pro zprávy nesoucí řetězec textových dat, tj. typ `String`, který je typicky nestrukturovaný. Je však možné jej použít i pro výměnu komplexnějších dat jako například XML dokumentů.
- **ObjectMessage** – datovým obsahem této zprávy je serializovatelný Java objekt. Díky tomuto typu je možná výměna objektů mezi aplikacemi, avšak jak odesilatel, tak příjemce musí mít přístup k definici třídy, jejíž instancí přenášený objekt je. Aby tedy byla výměna možná, obě komunikující strany musí být programy v Javě.
- **ByteMessage** – pokud potřebujeme zaslat data v nativním formátu aplikace, který není kompatibilní s žádným standardním typem, použijeme zprávu ByteMessage, jejímž obsahem jsou neinterpretované byty. Obvykle se jedná o situace, kdy JMS využíváme čistě jako prostředek pro přenos dat mezi dvěma systémy.

- **StreamMessage** – je typ nesoucí sérii primitivních datových typů, jako například `int`, `char`, `double` atd. Tyto typy jsou ze zprávy čteny ve stejném pořadí jako byly zapsány a je prováděna i typová kontrola.
- **MapMessage** – jako svůj obsah nese zpráva tohoto typu množinu dvojic jméno-hodnota. Hodnoty těchto dvojic mohou být primitivní datové typy nebo typ `String`.

4.2.2 Modely komunikace

Jak již bylo zmíněno v úvodu této sekce, při práci s JMS se setkáváme se dvěma modely zasílání zpráv:

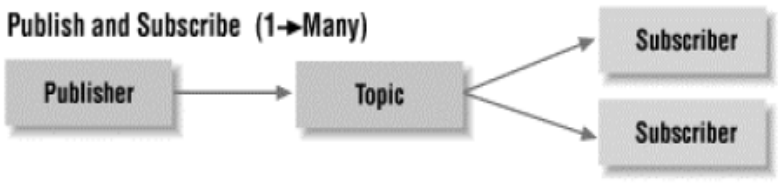
- *publish-and-subscribe* (pub/sub),
- *point-to-point* (ptp)

U obou modelů platí, že mezi komunikujícími klienty nejsou žádné pevné vazby. Jak odesilatelé, tak příjemci mohou být dynamicky přidáváni a odebírání, díky čemuž se může v průběhu času zvětšovat nebo zmenšovat komplexnost celého systému. Modely a jejich charakteristiky jsou popisovány na základě informací z [10, 17, 28].

Publish-and-subscribe

Princip modelu *publish-and-subscribe* spočívá v tom, že jeden producent může zaslat zprávu více konzumentům s využitím virtuálního kanálu neboli destinace (viz obrázek 4.2). V případě pub/sub modelu se pro destinaci používá označení *topic*. Významné charakteristiky tohoto modelu jsou následující:

1. Zprávy zaslané do topicu jsou doručeny všem klientům, kteří jsou zaregistrovaní k odběru.
2. Pro doručení zpráv je použit *push model*, což znamená, že zprávy jsou automaticky zaslány příjemcům, aniž by si o ně museli zažádat.
3. Každý klient přihlášený k odběru topicu obdrží svou vlastní kopii zprávy, která do něj byla odeslána.

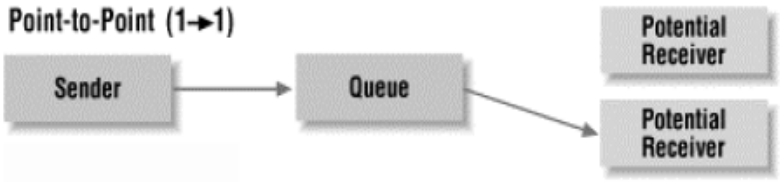


Obrázek 4.2: Komunikační model *publish-and-subscribe*. (Převzato z [28])

Point-to-point

Jak je ukázáno na obrázku 4.3, model *point-to-point* implementuje pro změnu komunikaci jeden s jedním, tj. každá zpráva má maximálně jednoho příjemce. Virtuální kanál využívaný v tomto modelu se nazývá *fronta*. Hlavní charakteristiky tohoto modelu jsou tedy tyto:

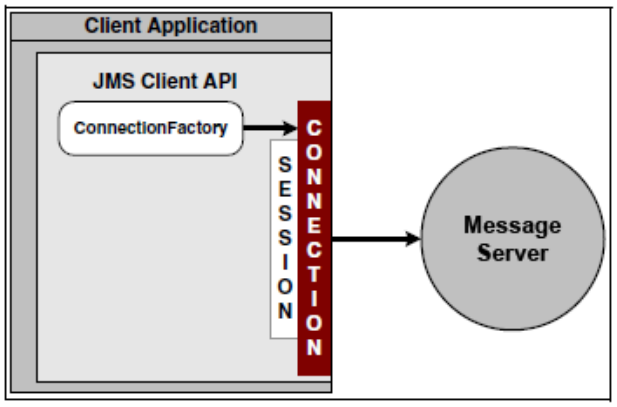
1. Přestože k frontě může být připojeno více klientů, zpráva je doručena vždy právě jednomu z nich.
2. Komunikace může být synchronní nebo asynchronní.
3. Zprávy jsou doručovány buď na žádost klienta s využitím *pull modelu*, nebo automaticky v případě *push modelu*.
4. Fronta zprávy doručuje ve stejném pořadí, v jakém jsou do ní zasílány. Toto pořadí může být ovlivněno prioritami zpráv.



Obrázek 4.3: Komunikační model *point-to-point*. (Převzato z [28])

4.2.3 Praktické aspekty JMS API

Pokud pro zasílání zpráv používáme JMS, budeme nejprve vytvářet spojení (*connection*), které identifikuje aplikaci a určuje, jak bude server s klientem nakládat, a v rámci něj následně vytvoříme jedno nebo více sezení (*session*). Po vytvoření sezení může klientská aplikace produkovat nebo konzumovat zprávy. [17, 22] Jakým způsobem je spojení se serverem zapouzdřeno, ukazuje obrázek 4.4.



Obrázek 4.4: Spojení aplikace s message serverem. O zapouzdření spojení se stará objekt *ConnectionFactory*. (Převzato z [22])

Po vytvoření sezení odesilatelé zasílají své zprávy do virtuálních kanálů (*destinací*), příjemci je pak z těchto destinací získávají. [17, 22]

Pro implementaci synchronní komunikace může klientská aplikace použít mechanismus *Request/Reply*, což znamená, že po odeslání zprávy čeká producent na odpověď. Aby bylo možné odpověď doručit, vytvoří pro ni producent tzv. *dočasnou destinaci* a její identifikátor odešle v hlavičce zprávy. [17, 22] Tyto destinace jsou vytvářeny dynamicky a jejich životnost je spojená se spojením, do kterého patří sezení, v rámci něhož byly vytvořeny. Je také zaručena unikátnost destinace napříč všemi spojeními. [28]

4.3 SonicMQ

Progress SonicMQ představuje certifikovanou implementaci JMS, která umožňuje využívat komunikační modely ptp a pub/sub. Díky velké škálovatelnosti umožňuje rychlé zvětšování topologie a prudké výkyvy v objemu zasílaných zpráv. Dále také zavádí nové typy zpráv, konkrétně *XML message* a *Multipart message*. [17]

SonicMQ je založen na topologii hvězda, která chápe všechny entity připojené do topologie jako klienty s výjimkou centrálního serveru neboli *SonicMQ brokeru*, ke kterému se všichni klienti připojují a přes který probíhá výměna zpráv. Pro komunikaci se severem lze využít následující protokoly:

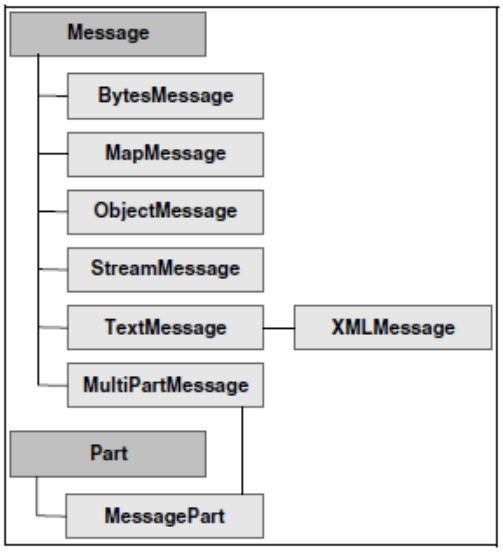
- **TCP** – implicitní typ socketu pro SonicMQ,
- **SSL** – SonicMQ podporuje kódování na úrovní spojení s využitím SSL,
- **HTTP** – kromě navázaní a udržování spojení klienta se serverem může být HTTP využito i pro interakci SonicMQ s čistě webovými aplikacemi,
- **HTTPS** – zabezpečená verze služeb HTTP. [22]

Destinace jsou reprezentovány objektem `Destination`. Jedná se o pojmenované lokace, do kterých mohou klienti zasílat zprávy a které jsou vždy buď typu fronta, nebo topic. Je možné poslat zprávu do dynamicky definovaného topicu i staticky definované fronty se stejným jménem. Server totiž rozlišuje, jakého typu je cílové destinace, tj. buď topic, nebo fronta. [17]

4.3.1 Zprávy v SonicMQ

SonicMQ zpráva je balík dat, který zapouzdřuje tělo zprávy a vystavuje navenek metadata, která identifikují cílovou destinaci, prioritu, časové razítko a další. Jak můžeme vidět na obrázku 4.5, kromě pěti typů zpráv odvozených od společného předka Message, které jsou dané JMS specifikací, nabízí SonicMQ také rozšíření v podobě následujících dvou typů:

1. **XMLMessage** – rozšiřuje typ TextMessage, její tělo obsahuje standardní TextMessage s XML tagy, která je zpracovatelná buď jako validní XML DOM nebo s využitím SAX,
2. **MultipartMessage** – tělo této zprávy obsahuje jednu nebo více částí. Každá část může představovat například standardní zprávu (objekt Message), primitivní typy jako XML, HTML nebo data v libovolném jiném MIME formátu. Každá část má svoji hlavičku, ve které jsou dvojice jméno/hodnota určující například typ obsahu, a obsah.



Obrázek 4.5: Hierarchie typů zpráv v SonicMQ. (Převzato z [22])

Pokud zprávě vyprší její platnost nebo ji SonicMQ označí za nedoručitelnou, nazveme ji *mrtvou zprávou*. Pro tyto zprávy nabízí SonicMQ tzv. *Dead Message Queue* (DMQ), která usnadňuje jejich obsluhu.

Co se týče struktury samotné zprávy, můžeme zde rozlišit standardní prvky definované v JMS specifikaci, konkrétně hlavičky, uživatelem definované vlastnosti a tělo. Kromě těchto polí využívá SonicMQ ještě poskytovatelem definované vlastnosti, tj. vlastnosti definované SonicMQ, které nesou informace potřebné k využívání rozšířených vlastností nabízených touto implementací. Jako příklad můžeme uvést hlavičku definující typ zprávy včetně nově zavedených typů nebo hlavičky informující jak nakládat s nedoručitelnými zprávami.

Kapitola 5

Podniková sběrnice služeb Sonic ESB

Abychom se mohli zabývat *podnikovou sběrnicí služeb* (enterprise service bus, ESB) a jejím významem, je nutné nejprve stručně vysvětlit pojem *servisně orientovaná architektura*, který s ní úzce souvisí. Servisně orientovaná architektura (SOA) je model architektury založený na konceptu služeb. [15, 30] Službou v tomto kontextu rozumíme samostatnou bezstavovou funkci, která je dostupná přes standardizované rozhraní nezávislé na implementaci. [7, 15]

Primárním cílem SOA je převést různorodé IT zdroje v obchodním prostředí na množinu centrálně spravovaných služeb. Tento přístup vytváří základ pro tvorbu aplikací, které umožňují znovupoužití a kombinaci existujících služeb za účelem tvorby komplexnější logiky. S využitím SOA je možné vytvořit distribuované prostředí, ve kterém spolupracuje velké množství aplikací bez ohledu na jejich geografické umístění, platformu, použitý datový model nebo jazyk, ve kterém jsou napsány. [7, 15, 30]

V typické SOA aplikaci poskytovatelé registrují své služby u centrální služby *Naming Service*. Klientské aplikace pak u ní vyhledávají dostupné služby a získávají informace o tom, jak s nimi komunikovat. [15]

V následujících částech této kapitoly představíme praktickou realizaci servisně orientované architektury prostřednictvím podnikové sběrnice služeb. Základní principy ESB budou vysvětleny a demonstrovány na produktu Sonic ESB, implementaci ESB od Progress Software.

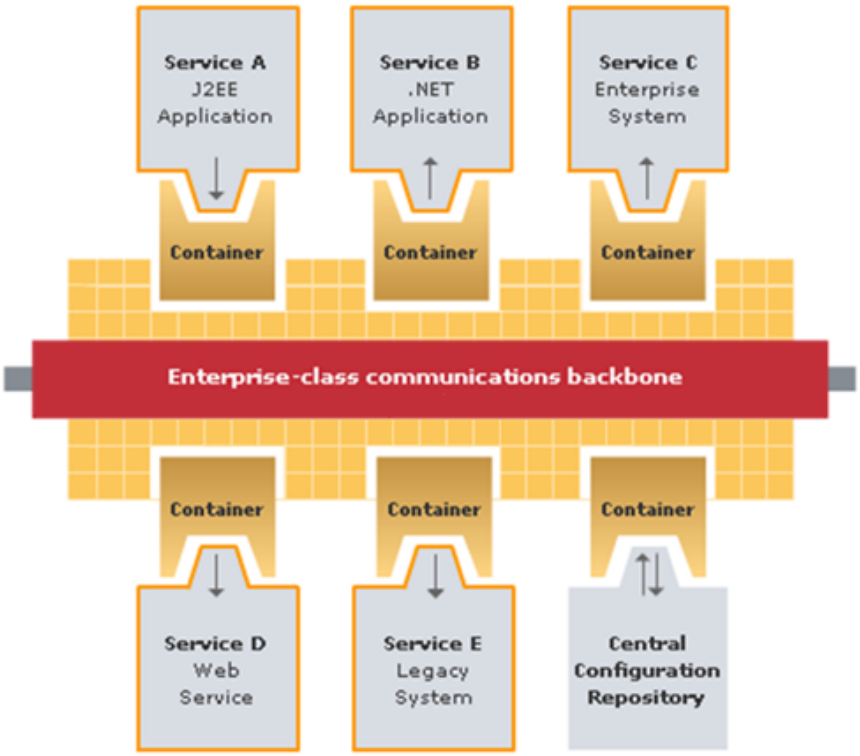
5.1 SOA implementovaná sběrnicí ESB

ESB nabízí distribuovanou, událostmi řízenou servisně orientovanou architekturu, která v sobě kombinuje zasílání zpráv, webové služby, datové transformace a inteligentní směrování za účelem spolehlivého propojení a koordinace interakce mezi různými aplikacemi napříč společností a jejími obchodními partnery. [7, 15, 27]

Jak můžeme vidět na obrázku 5.1, ESB na rozdíl od tradiční topologie hvězda využívá flexibilní topologii založenou na sběrnici. Jakmile jsou jednou aplikace vystaveny na sběrnici, je možné je snadno vzájemně propojovat s ostatními. [7, 15, 30]

Zároveň si také můžeme všimnout čtyř základních komponent v architektuře ESB:

1. **Komunikační páteř** – zajišťuje konektivitu mezi službami.



Obrázek 5.1: Architektura ESB. Služby jsou připojeny ke sběrnici (komunikační páteř) přes jednotné rozhraní (kontejner) bez ohledu na jejich implementaci. (Převzato z [30])

2. **Konfigurační repozitář** – centralizované úložiště všech metadat potřebných v sytému. V Sonic ESB se tento repozitář označuje jako *Directory Service*. [30]

3. **Kontejnery** – budou probrány v 5.2.2.

4. **Služby** – budou probrány v 5.2.3.

Komunikační páteř leží na nejnižší, fyzické vrstvě, která bývá obvykle reprezentována messaging systémem. Tato vrstva je pak řízena pravidly definovanými ve vyšších vrstvách. [7, 15, 27, 34] V případě Sonic ESB je jako komunikační páteř použita implementace JMS SonicMQ [27, 30], která byla probrána v sekci 4.3.

Nad fyzickou vrstvou pak leží ESB framework, jehož cílem je vytvářet, distribuovat, spouštět a spravovat služby. Kromě uvedených vrstev je možné se v ESB setkat i s dalšími nadstavbovými vrstvami. [34] Tyto vrstvy však nejsou důležité pro účely této práce, a proto se jimi nebudeme podrobněji zabývat.

O celý běh Sonicu, tj. jak SonicMQ, tak Sonic ESB se stará tzv. *Domain Manager*. [27] Ten má na starost řízení Directory Service a je zodpovědný za veškerou konfigurační komunikaci s ním. Prostřednictvím Domain Manageru je možné přes nástroj *Sonic Management Console* vytvářet kontejnery, řídit jejich životní cyklus, přiřazovat jim služby, spravovat zdroje Sonicu apod.

5.1.1 Komunikace v ESB

Komunikace mezi jednotlivými službami připojenými k ESB je založena na zasílání zpráv. Tyto zprávy přenášejí požadavky a následné odpovědi od jedné služby k druhé. [3] Každá zpráva v Sonic ESB je typu `XQMessage`, což je rozhraní v jazyce Java podobné typu MultipartMessage v SonicMQ. Toto rozhraní umožňuje přístup k hlavičkám a jednotlivým částem zprávy. [27, 30]

Obsah zprávy může být soubor XML nebo obecná binární data. [3, 7] Preferovaným formátem v ESB je však XML. Data, která jsou v systému produkována a konzumována mohou existovat ve velkém množství různých formátů. Přestože je možné přenášet přes ESB data v jakékoliv formě, využití jednotného formátu XML s sebou nese spoustu výhod jako například možnost využívat speciální ESB služby pro kombinaci dat z různých zdrojů, jejich transformaci, vložení nových hodnot nebo jejich přesměrování (viz 5.3). [7]

Služby v ESB vytvářejí ESB zprávy, které jsou následně zasílány na stanovené ESB adresy. ESB adresa reprezentuje pojmenovanou destinaci, na které příjemce čeká na příchozí zprávu. Pokud odesilatel požaduje odpověď na svoji zprávu, je součástí metadat zprávy i adresa pro zaslání odpovědi. [3]

Každá ESB adresa může být jednoho z následujících typů, jak udávají [3, 30]:

- **ESB endpoint** – abstrakce pro destinace transportní vrstvy (viz 5.2.1).
- **ESB služba** – odkazuje přímo na jméno služby, má stejný význam jako odkaz na vstupní endpoint této služby.
- **ESB proces** – posloupnost služeb zakomponovaná do komplexnějšího procesu (viz 5.2.4).

5.2 Klíčové prvky v Sonic ESB

V následující části kapitoly si vysvětlíme jednotlivé komponenty, se kterými se setkáme při návrhu a tvorbě ESB aplikací. Informace o nich byly čerpány primárně ze zdrojů [3, 7, 27, 30]. Jako doplňující zdroje byly dále použity [18] pro endpointy a kontejnery, [20] pro služby a [34] pro procesy.

5.2.1 ESB Endpointy

ESB endpointy umožňují komunikaci mezi jednotlivými službami. Endpoint představuje pojmenovanou destinaci, přes kterou ESB služba odesílá a přijímá zprávy. Jedná se tedy o její primární komunikační rozhraní. Aplikace přistupují ke službám odesíláním požadavků do jejich vstupních endpointů. Díky využití tohoto mechanismu je možné přesunout část transportní logiky ze služeb do lépe konfigurovatelného ESB.

Za každým endpointem leží na nižší vrstvě některá z JMS destinací, tj. topic nebo fronta, ke kterým ESB framework přistupuje přes SonicMQ broker. Rozlišujeme čtyři typy endpointů:

- **Vstupní endpoint** – přijímá zprávy, které jsou následně zpracovány službou. Každá přijatá zpráva způsobí jedno spuštění implementace služby.
- **Výstupní endpoint** – představuje poslední destinaci při vykonávání ESB procesu nebo služby, do které je vygenerována odpověď na příchozí zprávu. Zprávy z tohoto endpointu mohou být zaslány jak další službě, tak odesilateli původního požadavku.

- **Chybový endpoint** – volitelný endpoint služby nebo procesu, jehož úkolem je obsluha odstranitelných chyb a přerušení, které je nutné řešit na aplikační úrovni. O zaslání zprávy do chybového endpointu se rozhoduje na úrovni služby.
- **Rejected endpoint** – Zachytává zprávy, které jsou nevalidního typu pro daný proces, způsobily výjimku v průběhu vykonávání služby nebo je z nějakého důvodu není možné doručit. O zaslání zprávy do chybového endpointu se rozhoduje na úrovni ESB frameworku.

5.2.2 ESB kontejnery

ESB kontejnery jsou základní stavební prvky architektury v Sonic ESB. Jedná se o samostatnou komponentu, jejíž primární cíle jsou:

- řídit životní cyklus služeb,
- poskytovat službám běhové prostředí.

Kontejner dále také navazuje spojení s jednotlivými JMS destinacemi, tj. endpointy, čímž izoluje služby od zbytku ESB infrastruktury. Konfigurační informace získává od centralizovaného Directory Service, který v případě změn konfigurace zašle kontejneru notifikaci. ESB kontejner dále poskytuje podporu pro přístup k logovacím souborům služeb, které pod něj spadají.

Jednou ze základních komponent každého kontejneru je také logika tzv. *dispečera*, jehož úkolem je zajištění příjmu a odesílání ESB zpráv, spuštění služby, na jejíž vstupní endpoint dorazila zpráva, a interpretování *itineráře* zprávy (viz 5.2.4) při zpracování ESB procesu. Tento přístup umožňuje soustředit se při vytváření služby především na její logiku, aniž by bylo třeba starat se zpracování itineráře a směrování odchozích zpráv nebo jejich transformace, případně zotavení z chyb.

5.2.3 ESB Služby

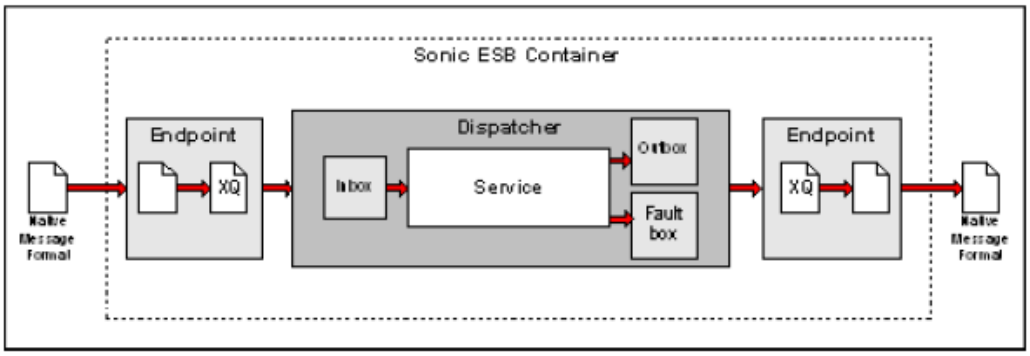
V Sonic ESB je služba chápána jako klient přijímající a odesílající zprávy, který může být použit samostatně nebo jako součást jednoho nebo více procesů. Veškerá aplikační logika zůstává uvnitř služby.

Konkrétní běžící služba je instance některého typu služby. Typ služby se skládá z třídy v jazyce Java, která definuje konkrétní operace prováděné službou, a XML soubor, který popisuje inicializační a běhové parametry služby. Konfigurace služby je pak specifické použití jejího typu s nakonfigurovanými endpointy a inicializačními parametry.

Každá služba je spravována ESB kontejnerem, který zajišťuje její konektivitu s messaging systémem na nižší úrovni a stará se o její životní cyklus. Vstup je službě předán ve formě `XQMessage` zprávy, její výstup je pak opět ve formě `XQMessage`.

Na obrázku 5.2 je ukázáno, jakým způsobem probíhá příjem a zpracování zprávy službou. Postupně proběhnou následující kroky:

1. Zpráva je přijata vstupním endpointem příslušné služby, kde je převedena do formátu `XQMessage`.
2. Dispečer kontejneru se postará o její přesunutí do *inboxu* adresované služby, čímž dojde k její iniciaci.



Obrázek 5.2: Cesta zprávy přes ESB kontejner.

3. Služba příjme zprávu a vykoná nad ní požadované operace.
4. Na základě výsledku prováděných operací jsou vygenerovány příslušné odchozí zprávy do *outboxu*, respektive *faultboxu* služby.
5. Dispečer umístí odchozí zprávy do příslušných endpointů na základě jejich adres.
6. Zprávy jsou v endpointech převedeny do vhodného formátu a odeslány do svých destinací.

Pokud mluvíme o *inboxu*, *outboxu* a *faultboxu*, máme vždy na mysli kolekci zpráv spravovanou dispečerem, ze které si služba vyzvedává příchozí zprávy a kam umisťuje odchozí.

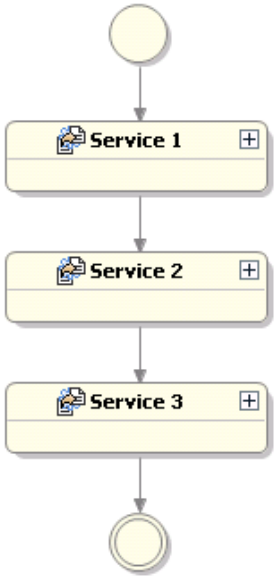
5.2.4 ESB procesy

Pojmem *proces* v ESB rozumíme posloupnost služeb, přes které přijatá zpráva postupně projde. Řízení procesu v ESB může představovat jak jednoduchou sekvenci o několika krocích, tak sofistikovaný obchodní proces s paralelním vykonáváním několika větví procesů využívající podmíněné větvení a opětovné spojování. Příklad jednoduchého procesu je ukázán na obrázku 5.3. Nejběžnější případy, co může udělat jeden krok, jsou:

- spustit jednu nebo více služeb,
- spustit jeden nebo více jiných procesů,
- vybrat jednu z více větví procesu, která bude vykonána,
- rozdělit proces na více paralelních větví.

Celý proces je distribuovaný na sběrnici, avšak jeho definice je centralizovaná v Directory Service a kterákoliv služba nebo endpoint běžící v ESB kontejneru jej může adresovat. Proces může být typicky vystaven navenek jedním z následujících způsobů:

- jako adresa na sběrnici,
- přes JMS požadavky,
- jako webová služba,
- přes HTTP požadavky.



Obrázek 5.3: Jednoduchý ESB proces v nástroji Sonic Workbench. Zpráva postupně putuje službami Service 1 až 3. Kruhy na začátku a na konci označují vstupní a výstupní endpointy procesu. (Převzato z [27])

Stejně jako služby musí být i procesy umístěny do ESB kontejneru. Zpráva do procesu vstupuje vždy přes vstupní endpoint jeho prvního kroku. Po zpracování zprávy dojde k jejímu přeposlání na vstupní endpoint následující služby nebo kroku. Pokud zpráva dorazí do poslední destinace, je odeslána do výstupního endpointu celého procesu.

Itinerář procesu

Jak již bylo zmíněno dříve, proces je definován posloupností kroků. Tato posloupnost se nazývá *itinerář*. Itinerář je připojen ke zprávě v momentě, kdy dorazí do daného ESB procesu. Zpráva jej pak nese s sebou po dobu celé její cesty k cílové destinaci.

Vyhodnocení itineráře provádí příslušný ESB kontejner v každém kroku ESB procesu, na základě čehož je pak spuštěna vhodná další služba nebo vnořený proces. Tento přístup eliminuje nutnost existence centrální jednotky, se kterou by musela každá služba navazovat kontakt a zjišťovat, kam má být zpráva přeposlána. Řízení procesu je tak kompletně distribuované.

Mapování polí zpráv

Mapování zpráv určuje, jak běžící proces čte data například z příchozí `XQMessage`, podle potřeby je transformuje a následně výsledná data mapuje na cíl, což může být opět `XQMessage`. Díky tomu je možné řešit tyto problémy na úrovni ESB procesů místo uvnitř implementace služeb, čímž se zjednodušuje proces jejich vytváření a integrace.

Jako možné akce, které lze s daty při mapování provést, si uvedeme následující zástupce:

- **XPath** – transformace zdrojových dat s využitím XPath výrazu,

- **Přidání částí** – zkopírování částí nebo hlaviček ze zdrojové `XQMessage` do cílové bez jejich modifikace. Tato akce je použitelná jen pokud mapováním vytváříme zcela novou `XQMessage`.
- **Odebrání částí** – části původní `XQMessage` jsou odebrány a neobjeví se v cílové. Akci lze použít, pokud jako odpověď přebíráme původní zprávu.

5.3 Přehled vestavěných služeb

V poslední části kapitoly o podnikové sběrnici služeb Sonic ESB budou představeny základní typy vestavěných služeb, které Sonic nabízí. Služba *Sonic Connect* umožňující volání webových služeb a zpřístupnění ESB procesů jako webové služby bude podrobněji popsána v části 7.4.1. Informace uváděné v této sekci byli čerpány především ze zdrojů [20, 30], jako vedlejší zdroje byli dále použity [7, 27, 34].

5.3.1 Směrování a transformace

Směrování podle obsahu (content based routing, CBR) umožňuje definovat službu, která prochází obsah přijatých zpráv, a následně směruje tyto zprávy k různým procesům, službám nebo endpointům na základě informací získaných z jejich hlaviček a těl. Mezi základní charakteristiky CBR typicky patří následující:

- Směruje zprávy na jednu nebo více ESB adres.
- Směrování je založené na komplexních pravidlech.
- Nemění obsah směrované zprávy.

Pravidla, která tato služba využívá ke směrování zpráv, mohou být definována jako jednoduchý XML dokument využívající XPath výrazy, nebo v případě náročnějšího směrování jako soubor v jazyce JavaScript. Pravidla CBR podporují výrazy a podmínky. Pokud je podmínka vyhodnocena jako splněná, je daná zpráva odeslána odpovídající cestou.

Kromě směrování zpráv podporuje Sonic ESB také jejich transformaci. Služba *XML transformace* slouží k převodu obsahu zprávy na zprávu s jiným formátem na základě XSLT souboru, který je službě předán jako její parametr. Stejným způsobem je možné modifikovat, vytvářet, případně mazat i hlavičky transformované zprávy. XSLT dále umožňuje vytvoření více výstupních XML zpráv z jedné vstupní a jejich odeslání do příslušných destinací.

5.3.2 Split and Join

Principem služby *Split and Join* je udělat z jedné vstupní zprávy více výstupních, nad každou z nich provést potřebné operace a jejich výsledek opět spojit do jedné zprávy. V Sonic ESB jsou k dispozici dva typy této služby:

1. *Split and Join Parallel*,
2. *Split and Join ForEach*.

Split and Join Parallel vytvoří několik kopií příchozí zprávy a každou z těchto kopií odešle na jednu adresu, přičemž každá adresa reprezentuje jinou ESB službu nebo proces. Po obdržení odpovědí od všech adresátů provede složení výsledků do jedné výsledné

zprávy. Tuto službu je vhodné použít v situacích, kdy je třeba nad zprávou provést několik nezávislých operací a jejich výsledek pak spojit do jediné zprávy.

Split and Join ForEach na rozdíl od předchozí varianty nekopíruje příchozí zprávu, ale rozděluje ji na několik částí. Jejich počet je možné určit dynamicky. Každá z těchto částí je následně odeslána na jednu stejnou adresu ke zpracování. Výsledky pro jednotlivé části zprávy jsou poté opět shromážděny a spojeny do jedné výsledné zprávy. Tento přístup se používá v situacích, kdy je třeba vykonat stejnou operaci nad různými částmi zprávy.

5.3.3 Služby pracující nad soubory

Pro manipulaci se soubory nabízí Sonic ESB dvě jednoduché služby:

- *File pickup*,
- *File drop*.

Přestože je možné setkat se i s dalšími službami pracujícími nad soubory, v rámci této práce stručně probereme jen uvedené dvě.

Služba *File pickup* slouží ke zkopírování obsahu zadaného souboru do odchozí zprávy vhodného typu. Kromě samotného obsahu souboru jsou do vlastností zprávy vloženy také jeho metadata. File pickup může zahájit svou činnost buď na základě přijetí zprávy, nebo provádí kontrolu souborů určených ke zpracování periodicky v pravidelných intervalech.

Služba *File drop* pak provádí inverzní činnost ke službě File pickup, tj. kopíruje obsah příchozí zprávy do nového souboru na specifikované místo na lokálním souborovém systému. Službu je možné nakonfigurovat tak, aby nevytvářela nové soubory, ale přidávala data na konec existujícího souboru.

Kapitola 6

OpenEdge ABL

Programovací jazyk *OpenEdge Advanced Business Language* (ABL), dříve *Progress 4GL* nebo jen *PROGRESS* [32], je vysokoúrovňový procedurální a objektově orientovaný programovací jazyk, umožňující výstavbu aplikací od uživatelského rozhraní až po přístup k databázi. Jazyk ABL je používán v rámci *Progress OpenEdge*, což je sada vývojových nástrojů pro vytváření dynamických, jazykově nezávislých aplikací a jejich nasazení na libovolné platformě. [19, 25] V mírně pozměněném prostředí OpenEdge je vytvořen i ERP systém proALPHA. Z toho důvodu bude jazyk ABL také hlavním implementačním jazykem této práce.

Základní charakteristiky jazyka uváděné v [19, 25] jsou následující:

- **Procedurální jazyk** – program je tvořen množinou příkazů, které mohou být uloženy v samostatných procedurách. Příkazy se skládají z jednoho nebo více klíčových slov jazyka ABL a případně parametrů.
- **Blokově strukturovaný** – příkazy mohou tvořit bloky, tj. sekvence jednoho nebo více příkazů, případně zanořených bloků, které sdílejí určité zdroje.
- **Interpretovaný** – zdrojový kód programu je přeložen do platformově nezávislého *runtime kódu* (r-code), který je následně interpretován *ABL Virtual Machine* (AVM).

Další důležitou součástí OpenEdge je také *AppServer*, který se stará o spouštění ABL procedur jako odpověď na žádost klientů přes síť. Díky němu je možné vytvářet komplexní distribuované aplikace tím, že umožníme klientům spouštět ABL procedury a funkce vzdáleně stejným způsobem, jako kdyby běžely na lokální stanici klienta. Vzdálenou procedurou, respektive funkcí tedy rozumíme proceduru nebo funkci spouštěnou na AppServeru odděleně od klienta a typicky na jiné stanici, než na které běží klient. [26]

V dalších částech této kapitoly probereme praktické aspekty jazyka OpenEdge ABL, které budou relevantní pro tuto práci. Konkrétně se jedná o vysvětlení pojmu procedur a funkcí a manipulace s daty, které čerpají ze zdrojů [19, 25], dále bude následovat představení datových struktur *temporální tabulka* a *ProDataSet* popsaných v [25, 29], možností práce s XML dokumenty, kterou se podrobně zabývá [24] a nakonec zmíníme adaptéry uváděné v [23], které umožňují komunikaci ABL se Sonic ESB a SonicMQ.

6.1 Procedury a funkce

Na rozdíl od většiny běžně používaných programovacích jazyků jazyk ABL striktně rozlišuje pojmy procedura a funkce. Zatímco funkce transformuje hodnoty parametrů na návratovou

hodnotu nějakého datového typu, procedura pouze vykoná určitý úsek kódu na základě parametrů, které jsou jí předány.

O spouštění procedur se stará příkaz `RUN`, kterému jsou také předávány jejich parametry. Každý parametr má datový a typ a jeden ze tří typů parametru:

- `INPUT` – je předán do procedury hodnotou.
- `OUTPUT` – je proměnná, do níž bude zapsána hodnota při vykonávání procedury. Původní hodnota parametru se nepředává.
- `INPUT-OUTPUT` – je kombinací předchozích dvou typů. Jedná se o klasické předání parametru odkazem.

Přestože hlavní komunikace procedury s jejím okolím probíhá přes parametry, je možné využít také její návratovou hodnotu. V případě ABL procedruy se ale jedná pouze řetězcovou hodnotu vracenou příkazem `RETURN`, kterou je ve volající proceduře možné získat příkazem `RETURN-VALUE`. Typické použití je pro indikaci úspěchu a neúspěchu procedury.

Existují dva typy procedur:

- externí procedury,
- interní procedury.

6.1.1 Externí procedura

Blok externí procedury je tvořen textovým souborem s libovolným počtem příkazů, který lze zkompilovat jako samostatnou jednotku spustitelnou příkazem `RUN`. Jedná se o nejzákladnější blok jazyka ABL a lze v něm použít kterýkoliv typ příkazu nebo bloku. Externí proceduře je možné definovat jeden nebo více vstupních bodů v podobě interních procedur a funkcí.

Na externí procedury můžeme nahlížet jako na určitý typ knihovny. Typicky je použijeme pro zapouzdření množiny souvisejících funkcí a procedur nebo rozsáhlé komplexní procedury, která funguje jako samostatný celek.

Abychom mohli externí proceduru využívat jako knihovnu funkcí, je nutné ji zavolat perzistentně pomocí možnosti `PERSISTENT` příkazu `RUN` jak demonstruje následující příkaz:

```
RUN h-FuncProc.p PERSISTENT SET hFuncProc.
```

Při tomto volání proběhne inicializace proměnné `hFuncProc` datového typu `HANDLE`. Ten můžeme chápat jako ukazatel, respektive popisovač objektu jazyka ABL, například právě externí procedury. Perzistentně zavolaná procedura a její datový kontext zůstávají uloženy v paměti.

6.1.2 Interní procedura

Pro definici bloku interní procedury použijeme příkaz `PROCEDURE` následovaný jejím jménem. Za ním je pak uveden výčet parametrů procedury, jejich typů (`INPUT`, `OUTPUT`, ...) a datových typů.

Interní procedura je vždy kompilována jako část externí procedury a může obsahovat kterýkoliv typ příkazu nebo bloku kromě další interní procedury. Pokud chceme zavolat interní proceduru definovanou externě, uvedeme v příkazu `RUN` handle externí procedury, ve které se její definice nachází:

```
RUN myProc IN hFuncProc (par1, par2).
```

6.1.3 Funkce

Funkce jazyka ABL se ničím neliší od funkcí jiných jazyků, tj. volá se uvedením jejího jména, za kterým následuje seznam parametrů. Ty jsou pak funkcí transformovány na návratovou hodnotu. Díky tomu je možné uvést volání funkce na libovolném místě, kde je očekávána proměnná nebo výraz stejného datového typu.

Funkce mohou být definovány lokálně, v externí proceduře nebo vzdáleně. Pokud funkci definujeme jinak než lokálně, je třeba uvést na začátku procedury její deklaraci. Následující příklad deklaruje funkci `myFunc`, která je definována v objektu (například externí proceduře) jehož handle je uložen v proměnné `hFuncProc`:

```
FUNCTION myFunc RETURNS DECIMAL (INPUT par1 AS DECIMAL) IN hFuncProc.
```

6.2 Přístup k databázi

Jazyk OpenEdge ABL nabízí sadu příkazů a funkcí pro co nejjednodušší manipulaci se záznamy databáze. Jako nejběžnější příklad se uvádí konstrukce, která má obdobný význam jako SQL příkaz `SELECT * FROM Customer`, konkrétně

```
FOR EACH Customer:
  DISPLAY Customer.
END.
```

Příkaz `FOR EACH` otevírá blok kódu, ve kterém proběhne dotaz na databázi a získání všech záznamů zadané tabulky. V každé iteraci tohoto bloku máme k dispozici jeden ze získaných záznamů a pomocí `DISPLAY` provedeme jeho zobrazení uživateli. `FOR EACH` je možné doplnit také klauzulí `WHERE` specifikující podmnožinu záznamů. Jako alternativu ke klíčovému slovu `EACH` lze použít `FIRST` a `LAST`, které z tabulky získají první, respektive poslední odpovídající záznam.

Další možnost přístupu k položkám databáze je s využitím příkazu `FIND`, který vrací jeden záznam tabulky. Tento příkaz lze kombinovat se zmiňovanými klíčovými slovy `FIRST` a `LAST` a dále také s `NEXT` a `PREV` pro přechod na sousední záznam. Stejně tak je možné aplikovat na výběr omezení v podobě klauzule `WHERE`. Jestli bylo hledaní úspěšné či nikoliv, můžeme zjistit vestavěnou funkcí `AVAILABALE`.

Kdykoliv se procedura snaží získat záznam z tabulky databáze, vytvoří pro ni AVM tzv. *record buffer* a záznam zpřístupní skrz něj. Record buffer je tedy datová struktura nesoucí jeden záznam databázové tabulky, se kterým se aktuálně pracuje. Pro každou odkazovanou tabulku v proceduře je vytvořen jeden implicitní record buffer se stejným jménem, jako je jméno tabulky. Record buffery je možné také definovat explicitně v případě potřeby práce s více záznamy z jedné tabulky najednou.

6.3 Temporální tabulky a ProDataSety

Pro zapouzdření a práci s většími bloky dat poskytuje jazyk ABL dvě datové struktury, které se chovají stejně jako data v databázi. Tyto struktury jsou

- *temporální tabulka* (`TEMP-TABLE`),
- *ProDataSet* (`DATASET`).

Každá z nich může být vytvořená přímo na základě dat uložených databázi, definovaná programátorem nebo může vzniknout kombinací obou přístupů. Tyto datové struktury také nabízejí několik metod pro usnadnění práce s nimi. Volání metod nad objekty jazyka ABL se realizuje operátorem `":"`, tj. například `myTempTable:WRITE-XML()`.

6.3.1 Temporální tabulky

Temporální tabulka poskytuje programátorovi téměř všechny vlastnosti databázové tabulky a je možné ji použít na většině míst, kde je vyžadována práce s databázovou tabulkou. Na rozdíl od databázové však temporální tabulka není perzistentní a tudíž se nikde trvale neukládá.

Tyto datové struktury se typicky používají v případech, kdy potřebujeme dočasně uložit několik řádků tabulky nebo předávat velké objemy dat mezi procedurami. Tabulky jsou viditelné jen v rámci sezení, které je vytvořilo nebo obdrželo jako parametr. Kromě operací souvisejících s perzistencí dat a víceuživatelským přístupem je možné s nimi pracovat identickým způsobem jako s databázovými tabulkami.

Temporální tabulku je možné vytvořit

- nezávisle na tabulkách v databázi,
- jako kopie databázových tabulek (klíčovým slovem `LIKE`),
- podle databázové tabulky, ale s přidanými nebo přejmenovanými poli apod.

Pokud je tabulka založená na databázové tabulce přebírá všechna její pole a indexy.

6.3.2 ProDataSety

ProDataSety jsou objekty, které dále rozšiřují funkcionalitu temporálních tabulek. Jedná se databázi uloženou v paměti, která je naplněna množinou souvisejících záznamů potenciálně v několika tabulkách. Ve skutečnosti tedy představují kolekce jedné nebo více temporálních tabulek, které mohou volitelně obsahovat také informace o jejich vzájemných vztazích.

Součástí definice ProDataSetu může být i část představující množinu vztahů, označovaných jako *Data-Relations*, z nichž každý definuje vztah mezi nadřízenou a podřízenou tabulkou této struktury. Vztahy mají podobu dvojic, které udávají jména polí v obou tabulkách tvořící primární a cizí klíče.

Díky definici vztahů mohou být závislé tabulky vyplněny automaticky na základě dat v nadřazené tabulce bez nutnosti vytváření explicitního dotazu na databázi. Ke každému nadřazenému záznamu jsou získány všechny jeho podřízené. Každá temporální tabulka v ProDataSetu může být získána z jiného zdroje dat. V nejjednodušším případě se jedná o OpenEdge databázi případně databázi jiného typu, ke které se přistupuje s využitím prostředí OpenEdge.

6.4 Zpracování XML

Standardní instalace OpenEdge zahrnuje prostředky pro čtení a zápis XML. Zároveň je možné provádět validaci dokumentu proti DTD nebo XML Schema. Jazyk ABL nabízí dvě API pro práci s XML dokumenty:

- Document Object Model (DOM),

- Simple API for XML (SAX).

Kromě těchto dvou metod ABL umožňuje i serializaci temporálních tabulek a ProDataSetů do XML a jejich opětovné načtení. K tomu slouží jejich metody `READ-XML` a `WRITE-XML`. Dále je možné exportovat i XML Schema tabulky nebo ProDataSetu a následně podle něj vytvořit prázdnou datovou strukturu.

6.4.1 SAX API

Zatímco DOM reprezentuje dokument jako množinu uzlů uspořádaných do stromové struktury, SAX API jej rozloží do posloupnosti volání procedur. Jedná se tedy o model proudového zpracování, který vždy v daném okamžiku pracuje pouze s jediným elementem a umožňuje na něj reagovat pomocí callback procedury. Díky tomu má SAX menší paměťové nároky, na druhou stranu ale není možný náhodný přístup k elementům.

Jazyk ABL poskytuje podporu pro SAX API prostřednictvím těchto objektů:

- `SAX-reader` – načítá a analyzuje XML dokument,
- `SAX-writer` – umožňuje generování XML dokumentu jako proudu znaků,
- `SAX-attributes` – obsahuje hodnoty atributů, které se mohou v XML vyskytnout.

Než začneme načítat XML dokument s využitím objektu `SAX-reader`, je nutné vytvořit externí proceduru, která obsahuje jednotlivé callback procedury. Ta se uloží do jeho atributu `HANDLER`. Po nastavení XML vstupu, je pak možné zahájit zpracování dokumentu metodou `SAX-PARSE`. V průběhu zpracování aplikace reaguje na události vyskytující se v XML dokumentu pomocí zadaných callback procedur. Jako příklady nejčastějších událostí si uvedeme následující:

- `StartDocument` – detekován začátek XML dokumentu,
- `StartElement` – detekován začátek XML elementu,
- `Characters` – zpracovávají se řetězcová data,
- `EndElement` – detekován konec XML elementu,
- `EndDocument` – detekován konec XML dokumentu.

Případné atributy se předávají proceduře `StartElement` přes automaticky vytvořenou instanci objektu `SAX-attributes`.

Vytváření nového XML dokumentu pomocí `SAX-writer` probíhá posloupností volání jeho metod, které otevírají, zavírají a plní jednotlivé XML elementy. Může se jednat například o následující metody:

- `START-DOCUMENT` – vytvoří nový XML dokument
- `START-ELEMENT` – vloží otevírající XML značku,
- `END-ELEMENT` – vloží ukončující XML značku,
- `END-DOCUMENT` – ukončí vytvářený dokument,
- `WRITE-DATA-ELEMENT` – zapíše kompletní element včetně jeho obsahu.

Při zpracování XML dokumentů v rámci této práce není nutný náhodný přístup k jejich elementům ani udržování zpracovávaného dokumentu v paměti. Vzhledem k nižší paměťové náročnosti tak bylo pro práci s XML upřednostněno SAX API.

6.5 Sonic adaptéry

Součástí OpenEdge je také podpora pro připojení k messagingovému systému SonicMQ a pro vystavení ABL kódu jako služby na podnikové sběrnici služeb Sonic ESB, které byly rozebírány v částech 4.3 a 5.

6.5.1 SonicMQ adaptér

Adaptér pro SonicMQ je v ABL k dispozici v podobě tří externích procedur:

- **ptpsession.p** – sezení využívající model ptp,
- **pubsubsession.p** – sezení využívající model pub/sub,
- **jmssession.p** – sezení umožňující využít současně ptp i pub/sub.

Tyto procedury implementují objekty využívané v JMS API, konkrétně spojení, sezení a zprávy.

Nezbytnou součástí JMS komunikace v ABL je také objekt `MessageConsumer`. Jeho úkolem je přijímat zprávy z JMS destinací, případně asynchronní chybové zprávy. Při práci s tímto objektem aplikace typicky implementuje proceduru, která se stará o obsluhu příchozích zpráv, poté vytvoří instanci `MessageConsumer` a tuto proceduru mu předá. `MessageConsumer` se následně buď přihlásí k odběru topicu, nebo přijímá zprávy z fronty.

6.5.2 Sonic ESB adaptér

Sonic ESB adaptér umožňuje, aby OpenEdge služba (např. program v jazyce ABL) umístěná na sběrnici Sonic ESB mohla být zavolána třeba jako součást Sonic ESB procesu. Existují dva přístupy jak toho dosáhnout:

- **Nativní invokace** (Native Invocation),
- **Webové služby** – Sonic ESB zavolá aplikaci na AppServeru jako webovou službu.

V případě nativní invokace volá Sonic ESB přímo aplikaci na AppServeru prostřednictvím služby *OpenEdge Native service*. Ta bere jako své běhové parametry tzv. invokační soubory s příponou `.esboe` vytvářené v rámci prostředí OpenEdge. Ve své podstatě se jedná se o XML dokumenty specifikující podrobnosti o tom, jakou ABL proceduru spustit a jaké parametry tato procedura očekává. Úkolem OpenEdge Native service je rovněž postarat se o namapování hodnot na tyto parametry. Tento přístup bude podrobněji probrán v 7.4.2.

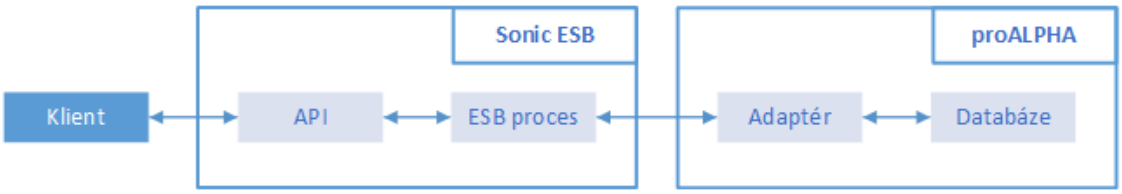
Kapitola 7

Návrh systému pro externí přístup

V předcházejících kapitolách této práce jsme nejprve rozebrali problematiku a základní principy ERP systémů a následně jsme na základě těchto poznatků zanalyzovali systém proALPHA, a to především z hlediska možností zpřístupnění jeho funkcionality pro externí klientské aplikace. V rámci této analýzy jsme rovněž vybrali správu vychystávacích návrhů jako vhodnou oblast, jejíž funkcionalitu budeme zpřístupňovat, a cestu, po které budeme do systému přistupovat, konkrétně modul INWB.

Dále práce popisovala technologie, které budou pro řešení problému externího přístupu relevantní, tj. nástroje SonicMQ a Sonic ESB, na kterých je vystavěn modul INWB, a hlavní implementační jazyk systému proALPHA OpenEdge ABL. V této kapitole se budeme zabývat tím, jak tyto technologie spojit a využít pro vytvoření systému umožňujícího provádění operací nad daty v proALPHA prostřednictvím vzdálených klientů.

Princip, na němž bude celý systém fungovat, je zachycen na obrázku 7.1. Klientské aplikace zasílají své požadavky na systém prostřednictvím aplikačního rozhraní (API), které bude tyto požadavky transformovat do zpráv typu `XQMessage` využívané pro komunikaci mezi službami na sběrnici Sonic ESB (viz 5.1.1). Na základě `XQMessage` ESB proces identifikuje operaci, která má být provedena, a její parametry a postará se o spuštění příslušných ABL procedur. Po skončení operace proces rovněž vrátí klientovi vhodnou odpověď.



Obrázek 7.1: Schéma navrhovaného systému pro externí přístup do systému proALPHA. Přestože Sonic ESB lze považovat za součást modulu INWB, v tomto schématu na něj nahlížíme jako na externí komponentu.

Na straně systému proALPHA se pak nachází adaptér, jehož interní procedury a funkce nabízejí podporu pro manipulaci s daty v systému. Jeho úkolem je rovněž podat vhodným způsobem informace o tom, jak požadovaná operace dopadla, případně k jakým chybám při ní došlo. Část označená jako *Databáze* pak představuje oblast dat systému vybranou v sekci 3.2, ke kterým budeme externě přistupovat, tj. správa vychystávacích návrhů v modulu materiálového hospodářství. Vychystávací návrhy se nacházejí v databázové tabulce `MMT_Picking`.

7.1 Operace nad vychystávacími návrhy

Ještě před tím, než se začneme podrobně zabývat návrhem jednotlivých částí systému pro externí přístup, probereme, jaké operace jsou možné nad vychystávacími návrhy v systému proALPHA a které z nich umožníme klientům provádět přes API.

Vychystávací návrhy umožňují rezervovat skladové položky v požadovaném množství a přesunout je do vychystávacího skladu, kde jsou připraveny k převzetí. Typicky se používají v situacích, kdy je třeba vychystat zboží na základě zákazníkovy objednávky, odebrat zdroje nutné pro výrobní činnost nebo při přesunu položek z jednoho skladovacího místa na jiné.

Do správy vychystávacích návrhů se dostaneme s využitím navigačního panelu systému proALPHA přes položky *Materiálové hospodářství* → *Vychystávání* → *Vychystávací návrhy*. Zde se zobrazuje hlavní okno, popisované v sekci 3.1 s detaily jednoho vychystávacího návrhu. Na závislé okno s jeho položkami se dostaneme přes *Funkce* → *Položky*.

Vychystávací návrh může být v jednom z následujících stavů vychystávání:

- **Nový** – stav bezprostředně po vytvoření vychystávacího návrhu, dosud neproběhla žádná manipulace s jeho daty,
- **Ve zpracování** – na návrhu se pracuje (tj. byl změněn některý z jeho údajů),
- **Částečně vychystaný** – některé položky návrhu jsou již vychystány, ostatní na vychystání teprve čekají,
- **Vychystaný** – všechny položky návrhu jsou vychystány.

Ze zobrazených údajů o vychystávacím návrhu umožňuje hlavní okno editaci následujících polí:

- **komisionářský sklad** – identifikátor vychystávacího skladu, kam budou položky návrhu přesunuty,
- **vychystávací místo** – oblast vychystávacího skladu, do které jsou položky umístěny,
- **referent** – uživatel zodpovědný za vychystávací návrh,
- **stav vychystávání** – viz výše,
- **rozdělovník** – umožňuje uživateli podle potřeby definovat klasifikaci návrhů,
- **číslo formuláře** – identifikace šablony, která je použita pro tisk vychystávacího návrhu,
- **počet formulářů** – počet kopií k vytištění.

Editace všech uvedených položek bude umožněna i v rámci demonstrace externího přístupu do systému proALPHA. Prostřednictvím položky menu *Nástroje* je možné přistoupit ještě k dalším operacím nad vychystávacími návrhy jako například zaúčtování návrhu, tj. vygenerování jeho následujícího dokumentu. Podpora pro tyto operace nebude v rámci této práce implementována.

V závislém okně pak lze editovat kromě položek komisionářský sklad, vychystávací místo a stav vychystávání, jejichž význam je obdobný jako u hlavního okna, ještě následující pole:

- **odebrané množství** – množství, které již bylo odebráno ze skladu,

- **PDo** – příznak dodávky popisující detaily dodání, např. jestli se jedná o kompletní dodávku nebo jen částečnou.

I v tomto případě bude umožněna vzdálená editace všech těchto polí.

Je důležité zmínit, že změny těchto hodnot rozhodně nejsou vzájemně nezávislé. Pokud například změníme stav jednoho závislého řádku položky na *Vychystáno*, bude jeho odebrané množství automaticky nastaveno na hodnotu rovnu rozdílu požadovaného a dosud odebraného množství. Stav *Vychystáno* pak bude nastaven také hlavnímu řádku odpovídající položky. Obdobným způsobem se projeví i změna některého z polí samotného vychystávacího návrhu. Při změně vychystávacího místa tak například bude nastavena stejná hodnota také pro vychystávací místa všech položek návrhu.

Kromě operací editujících uvedená datová pole vychystávacích návrhů a jejich položek bude v rámci této práce umožněno také vytvoření vychystávacího návrhu na základě objednávky a mazání vychystávacích návrhů respektive jejich jednotlivých položek.

7.2 Návrh aplikačního rozhraní

Při vytváření vhodného aplikačního rozhraní systému pro externí přístup je nutné brát ohled na to, že pokud mluvíme v této práci o externích klientech, máme na mysli primárně mobilní zařízení s omezenými výpočetními zdroji. Aplikace budou k systému přistupovat jako k webové službě a jak uvádějí práce [9, 33], preferovaným přístupem pro komunikaci s mobilními aplikacemi je v tomto případě využití REST arichtektury.

Důvod upřednostnění RESTful webových služeb před těmi založenými na protokolu SOAP je právě otázka výkonu. Použití SOAP webových služeb může u mobilních aplikací vést na nepřijatelně vysokou spotřebu výpočetních zdrojů, která může být způsobena například zpracováváním objemných XML zpráv, přes které SOAP komunikace probíhá. V důsledku toho je pak výkonnost celé klientské aplikace velmi nízká. Dalším důvodem pro využití REST je možnost jeho snadné a intuitivní implementace prostřednictvím ESB procesu v Sonic ESB. [21]

7.2.1 Architektura REST

Ještě než se v této sekci budeme věnovat samotnému návrhu podoby aplikačního rozhraní, stručně probereme základní principy a vlastnosti, na kterých jsou RESTful webové služby založeny. Uváděné informace jsou čerpány ze zdrojů [9, 14, 21, 33].

Architektura REST nahlíží na data a funkcionalitu jako na tzv. *zdroje*, ke kterým je umožněn přístup přes jejich *Uniform Resource Identifikátory* (URI). Pro komunikaci klientů se serverem je použit bezstavový protokol, typicky HTTP. Díky použití standardních protokolů a rozhraní pro výměnu zdrojů jsou aplikace založené na REST jednoduché, odlehčené a nabízejí vysokou výkonnost.

RESTful webové služby jsou založeny na REST architektuře, tj. zdroje jsou vystaveny klientům prostřednictvím URI a pro manipulaci s nimi jsou použity čtyři základní HTTP metody:

- `GET` – slouží pro získání zdroje, a to jak jednoho konkrétního, tak celé množiny zdrojů,
- `PUT` – metoda umožňující editaci zdroje s daným identifikátorem, v těle požadavku je nový obsah zdroje,
- `POST` – vytváří zdroj s obsahem, který je zaslán v těle požadavku,

- `DELETE` – smazání zdroje se zadaným identifikátorem.

Každá URI zdroje může mít až čtyři různé typy částí:

- **neměnná část** – část URI která zůstává stejná, např. `/books` specifikuje seznam všech knih v databázi,
- **proměnné URI** – liší se podle toho, k jakému konkrétnímu zdroji přistupujeme, např. `/books/hamlet` a `/books/othello`,
- **query parametry** – umisťují se na konec URI oddělené otazníkem, specifikují detaily vybraných záznamů, např. `/books?year=2010` pro všechny knihy vydané v roce 2010,
- **matrix parametry** – se oddělují středníkem, mají podobný význam jako query parametry, ale vztahují se k aktuální úrovni, na které jsou uvedeny, např. pro vybrání všech knih autorů, jejichž křestní jméno je *Jack*, můžeme použít `/authors;name=jack/books`.

S RESTful webovými službami se typicky jako formát pro výměnu dat pojí JSON. Jeho výhodou oproti XML je především jeho stručnost, díky čemuž se minimalizuje velikost přenášených dat. I přes výhody formátu JSON je však třeba myslet na to, že preferovaným formátem pro komunikaci v rámci Sonic ESB je značkovací jazyk XML, jak bylo zmíněno v části 5.1.1, a proto bude XML primární formát pro komunikaci s navrhovaným systémem pro externí přístup.

Jako výhody XML můžeme zmínit například jeho lepší vyjadřovací schopnosti, konkrétně možnost vytváření atributů pro jednotlivé elementy dokumentu nebo definování jmenných prostorů. Podrobnosti ohledně jazyka XML je možné nalézt v jeho specifikaci [13].

7.2.2 Definice rozhraní

V sekci 7.1 jsme definovali, jaké operace bude klientům umožněno vykonávat prostřednictvím aplikačního rozhraní. V této části se budeme zabývat konkrétní podobou jednotlivých URI, které tyto operace zpřístupňují, HTTP stavovými kódy vracenými jako součást odpovědí na klientské požadavky a nastíníme podobu dat, která se budou přes API posílat.

V navrhovaném REST API budeme mít celkem tři různé URI, kterými budeme implementovat vybrané operace nad vychystávacími návrhy:

- `/staging/suggestions` – umožňuje pracovat nad množinou vychystávacích návrhů, případně vytvořit nový návrh,
- `/staging/suggestions/{sid}` – zprostředkovává přístup k jednomu vychystávacímu návrhu identifikovanému hodnotou proměnného pole `sid`,
- `/staging/suggestions/{sid}/lines/{lid}` – dále specifikuje jeden konkrétní řádek daného vychystávacího návrhu prostřednictvím proměnné části `lid`,

Nad jednotlivými URI aplikačního rozhraní jsou definovány různé HTTP metody. Tyto metody a operace, které jsou za nimi skryty, jsou shrnuty v tabulce 7.1. Stavové kódy vracené jako součást odpovědí na požadavky klientů a případy, kdy jsou vráceny, jsou pak uvedeny v tabulce 7.2.

Vychystávací návrhy budou vraceny vždy i se všemi svými položkami. Jelikož jsou položky v systému proALPHA vždy chápány jako neodmyslitelná část vychystávacího návrhu, nemá smysl implementovat metodu `GET` pro získání jediné položky některého z návrhů.

URI	Metoda	Význam
`/suggestions`	GET	Vrátí seznam všech vychystávacích návrhů v systému proALPHA.
	POST	Vytvoří nový vychystávací návrh na základě zakázky v systému. Identifikátor zakázky je zaslán v těle požadavku. Při úspěšném provedení operace bude nově vzniklý návrh vrácen v těle odpovědi.
`/suggestions/{sid}`	GET	Získá jeden konkrétní vychystávací návrh identifikovaný proměnnou `sid` včetně všech jeho položk.
	PUT	Upraví vybraný vychystávací návrh. Jeho nová podoba bude součástí těla HTTP požadavku. Při úspěšném provedení operace je aktualizovaný návrh vrácen v těle odpovědi.
	DELETE	Smaže vybraný vychystávací návrh.
`/suggestions/{sid}/lines/{lid}`	PUT	Upraví vybraný řádek vychystávacího návrhu daný proměnnou `lid`. Ovlivněný návrh bude vrácen v těle odpovědi.
	DELETE	Smaže vybraný řádek vychystávacího návrhu. Při úspěšném provedení operace bude celý ovlivněný návrh vrácen v těle odpovědi.

Tabulka 7.1: Přehled HTTP metod, které je možné volat nad URI aplikačního rozhraní a jejich význam. URI jsou uvedeny bez společné neměnné části `/staging`.

Stavový kód	Navrácen
`200 OK`	• Požadavek `GET` proběhl v pořádku. • Požadavek `PUT` proběhl v pořádku. • Řádek vychystávacího návrhu byl úspěšně smazán.
`201 Created`	• Vychystávací návrh byl úspěšně vytvořen.
`204 No Content`	• Vychystávací návrh byl úspěšně smazán.
`400 Bad Request`	• Neznámá HTTP metoda. • Požadavek způsobil chybu v adaptéru.
`405 Method Not Allowed`	• Požadovanou metodu nelze aplikovat na danou URI.
`501 Not Implemented`	• Pokus o provedení neexistující operace.

Tabulka 7.2: Přehled HTTP stavových kódů vracených aplikačním rozhraním a situace, ve kterých budou vráceny.

V případě, že dotaz na systém pro externí přístup způsobí chybu na straně adaptéru, bude v těle odpovědi vrácen seznam chyb, které byly při zpracování zachyceny. Tato situace obvykle nastává například jako důsledek zadání nepovolené hodnoty do některého z editovatelných polí návrhu, pokus o přístup k neexistujícímu návrhu apod. Příslušné odpovědi mají typicky stavový kód `400 Bad Request`, případně `501 Not Implemented`.

Řešení závislostí mezi editovatelnými poli

V závěru této části práce ještě zmiňme, že navrhované REST API na rozdíl od systému proALPHA nepodporuje možnost současné editace několika datových polí položek vychystávacího návrhu. Vzhledem ke vzájemným závislostem jednotlivých polí, které byly nastíněny v části 7.1, může tento přístup způsobovat komplikace při editaci jako například uvedení systému do nekonzistentního stavu.

Z výše uvedeného důvodu je třeba umožnit při editaci položky vychystávacího návrhu přes API změnu pouze jedné hodnoty v daném okamžiku. Tyto změny se ihned uloží do databáze včetně automatických změn ostatních polí, které tato akce může vyvolat. Proto není v tomto případě možné implementovat editace obvyklým přístupem orientovaným na zdroje, při kterém metoda `PUT` předá serveru novou podobu daného zdroje, nýbrž je nutné ji chápat spíše jako funkci, které vždy předáme pole, jež chceme měnit, a jeho novou hodnotu. Tento problém není nutné řešit u editace dat samotného vychystávacího návrhu, jelikož jeho datová pole nejsou vzájemně závislá.

Podobným způsobem je nutné přistupovat i k metodě `POST` vytvářející nový vychystávací návrh. Jelikož vychystávací návrhy nejsou generovány uživateli, ale vytvářeny systémem na základě existujících dokumentů, je třeba i v tomto případě specifikovat metodě parametry pro vykonání operace místo konkrétního těla zdroje, který se má vytvořit. Předávané parametry specifikují operaci, která vychystávací návrh vytváří a data nutné k jejímu správnému provedení. Jelikož v rámci této práce umožňujeme pouze generování návrhu na základě existující objednávky, bude tato operace vždy představovat vychystání objednávky s daným identifikátorem.

7.3 Adaptér a jeho funkce

V předchozí sekci jsme definovali podobu REST API, přes které bude klientům umožněna interakce se systémem proALPHA. Nyní se budeme zbývat tím, jak vhodně vytvořit část označenou na schématu 7.1 jako *Adaptér*, která bude přímo manipulovat s vychystávacími návrhy systému a na jejíž funkce a procedury budeme mapovat jednotlivé zdroje navrženého API. Funkcionalita adaptéru rovněž zahrnuje podání vhodné zprávy o výsledku operace.

Adaptér, který představuje stěžejní část celého systému pro externí přístup, bude realizován jako externí procedura jazyka ABL, požadované operace budou v rámci adaptéru implementovány jako jeho interní procedury. Adaptér bude dále nabízet i některé podpůrné procedury a funkce pro usnadnění práce s ním.

Po skončení činnosti adaptéru bude možné získat informace o tom, jestli při běhu volaných procedur došlo k chybě nebo ne. Jelikož těchto chyb mohlo při zpracování vzniknout více, jsou chyby postupně zaznamenávány do XML zprávy, která pak může být předána zpět klientovi a zobrazena uživateli.

Aplikace využívající adaptér by tedy měla vždy po vykonání požadavků od klienta zkontrolovat zda nedošlo k chybě prostřednictvím funkce `lGetAnyError`, která vrací hodnotu

`True` v případě alespoň jedné chyby nebo `False`, pokud vše proběhlo v pořádku.[1] Pro získání XML s hlášením o chybách bude v adaptéru k dispozici funkce `clGetErrorList`.

V následujících částech této sekce si postupně rozebereme jednotlivé procedury nabízené adaptérem pro implementaci obsluhy požadavků na API, jejich vstupní a výstupní parametry a situace, kdy v nich může vzniknout chyba. Přehled těchto funkcí je uveden v tabulce 7.3.

Procedura/funkce	Význam
`getSuggestions`	Získá všechny dostupné vychystávací návrhy.
`stageOrder`	Vychystá zadanou zakázku.
`getSuggestionById`	Získá jeden vychystávací návrh.
`updateSuggestionById`	Upraví hodnoty zadaného vychystávacího návrhu.
`deleteSuggestionById`	Smaže zadaný vychystávací návrh.
`updateLineAdpotionCode`	Upraví hodnotu odpovídajícího pole řádku návrhu.
`updateLinePickStatus`	
`updateLinePickedQty`	
`updateLinePickingLocation`	
`updateLinePickingStorage`	
`deleteLine`	Smaže zadaný řádek vychystávacího návrhu.
`lGetAnyError`	Informuje o vzniku chyby při zpracování.
`clGetErrorList`	Vrátí seznam chyb.

Tabulka 7.3: Přehled procedur a funkcí adaptéru, zajišťujících implementaci vybraných operací nad vychystávacími návrhy a obsluhu možných chyb.

7.3.1 Získání vychystávacích návrhů

Abychom mohli pracovat s vychystávacími návrhy přes navržené REST API, je v prvé řadě nutné dát klientovi možnost návrhy získat z databáze. Jak již bylo zmíněno dříve, získání návrhů bude probíhat prostřednictvím metody `GET` nad příslušnou URI, přičemž mohou nastat dvě situace:

- získání všech vychystávacích návrhů,
- získání jednoho konkrétního vychystávacího návrhu.

Pro získání množiny všech otevřených vychystávacích návrhů, tj. těch, které dosud nebyly archivovány nebo jinak vyřízeny, bude adaptér poskytovat proceduru `getSuggestions`, která má jediný výstupní parametr:

- `opclMMT_Picking` – XML reprezentace množiny vychystávacích návrhů.

Tato procedura nikdy neskončí s chybou. Pokud nejsou v systému žádné otevřené vychystávací návrhy, je klientovi vráceno prázdné XML.

Získání jediného vychystávacího návrhu bude adaptér realizovat prostřednictvím procedury `getSuggestionById`, která očekává následující parametry:

[1] V rámci této práce budeme pro booleovské hodnoty používat běžné označení `True` a `False`, přestože jazyk OpenEdge ABL ve skutečnosti používá hodnoty `yes` a `no`.

- `piPickingId` – identifikátor vychystávacího návrhu,
- `opclMMT_Picking` – výstupní parametr, XML reprezentace získaného návrhu.

K chybě v této proceduře dojde v případě, kdy neexistuje vychystávací návrh s uvedeným identifikátorem.

7.3.2 Vytváření a mazání záznamů

Přestože vychystávací návrhy mohou vznikat různými způsoby, v sekci 7.1 jsme uvedli, že v rámci této práce bude implementována pouze možnost vytvoření prostřednictvím operace vychystání objednávky. Pro její vykonání bude adaptér nabízet proceduru `stageOrder` s parametry

- `piOrderID` – identifikátor objednávky k vychystání,
- `opiPickingId` – výstupní parametr, identifikátor vzniklého vychystávacího návrhu,

Chyba u této procedury nastane, pokud neexistuje objednávka se zadaným identifikátorem nebo pokud z nějakého důvodu vychystání neumožnil systém proALPHA.

V případě mazání záznamů z databáze máme možnost buď

- smazat celý vychystávací návrh včetně všech jeho závislých záznamů nebo
- smazat pouze jeden z řádků mezi jeho závislými položkami.

Pro mazání celých záznamů bude v adaptéru k dispozici procedura `deleteSuggestionById` s jediným vstupním parametrem `piPickingId` představujícím identifikátor mazaného návrhu. Kromě pokusu o smazání neexistujícího vychystávacího návrhu může v tomto případě vzniknout chyba pouze tehdy, pokud operaci z nějakého důvodu zamítne samotná proALPHA.

V případě mazaní jedné ze závislých položek návrhu procedurou `deleteLine` je rozdíl pouze v tom, že procedura bere jeden parametr navíc, konkrétně `pcInternalID`, který identifikuje jeden řádek závislých položek a je unikátní v rámci příslušného vychystávacího návrhu.

Chyby zde může mimo pokusu o smazání neexistujícího řádku vyvolat např. snaha o vymazání jiného než hlavního řádku položky nebo smazání poslední položky vychystávacího návrhu. Veškeré mazaní položek je pak dále podmíněno povolením částečné dodávky.

7.3.3 Editace vychystávacích návrhů

Stejně jako u mazání dat musíme i v tomto případě rozlišovat, zda chceme měnit hodnoty datových polí u samotného vychystávacího návrhu nebo u jednoho řádku některé z jeho položek. Dále je třeba vzít v úvahu i problém vzájemné závislosti datových polí probíraný v 7.1.

Jak již bylo dříve řečeno, tyto závislosti nemusíme řešit v případě údajů o vychystávacím návrhu. Při jeho editaci si tedy vystačíme s jedinou procedurou `updateSuggestionById`. Ta bude navíc podporovat editaci takovým způsobem, jaký se běžně používá pro editaci zdrojů přes REST API, tj. přijímá novou verzi zdroje a aktualizuje starou. Z toho plynou následující parametry této procedury:

- `piPickingID` – identifikace upravovaného vychystávacího návrhu,

- `pclSuggestion` – XML s novou podobnou upravovaného vychystávacího návrhu.

Změny needitovatelných hodnot zaslané v XML s novou podobou daného návrhu budou ignorovány stejně jako jakékoliv změny v týkající se jeho položek. Jelikož jeden ze vstupů této procedury je dokument ve formátu XML, bude nutné implementovat i jeho zpracování a získání potřebných údajů. Chyby zde můžou vzniknout zadáním neplatného identifikátoru vychystávacího návrhu nebo pokusem o nastavení nepovolené hodnoty některého z polí.

O něco komplikovanější je editace jedné položky vychystávacího návrhu. Kvůli zmiňovaným datovým závislostem bylo určeno, že nebude možné upravovat více hodnot jediným příkazem. Proto nebude editace položek implementovaná jednou procedurou, jako tomu bylo v předchozím případě, nýbrž pro změnu každého z editovatelných polí bude existovat samostatná procedura. Zde je uveden jejich přehled:

- `updateLineAdpotionCode` – mění hodnotu příznaku dodávky,
- `updateLinePickStatus` – mění stav vychystávání řádku položky,
- `updateLinePickedQty` – mění hodnotu odebraného množství,
- `updateLinePickingLocation` – mění vychystávací místo,
- `updateLinePickingStorage` – mění komisionářský sklad položky.

Všechny uvedené procedury budou očekávat na vstupu stejnou trojici parametrů:

- `piPickingID` – identifikátor vychystávacího návrhu, pod který editovaný řádek spadá,
- `pcInternalID` – identifikace řádku položky v rámci daného vychystávacího návrhu,
- *nová hodnota* – parametr nesoucí novou hodnotu pole, jeho pojmenování se liší.

Jako vždy, i v tomto případě může být zdrojem chyby úprava neexistujícího vychystávacího návrhu nebo jeho položky, dále je třeba hlídat, aby nedošlo k zadání neplatné hodnoty pro dané datové pole.

7.4 Propojení API s adaptérem

V sekcích 7.2 a 7.3 jsme navrhli podobu aplikačního rozhraní, k němuž budou přistupovat vzdálené klientské aplikace, a adaptéru manipulujícího s daty systému proALPHA. Nyní je třeba, abychom tyto dvě části propojili a namapovali zdroje nabízené aplikačním rozhraním na interní procedury adaptéru. Toto propojení bude realizované prostřednictvím podnikové sběrnice služeb Sonic ESB (viz kapitola 5) v rámci modulu INWB.

7.4.1 RESTful webová služba jako ESB proces

Sonic ESB může spolupracovat s RESTful webovými službami prostřednictvím služby *Sonic Connect* [21], která umožňuje jak kontaktování webové služby v rámci kroku v ESB procesu, tak vystavení ESB procesu jako RESTful webové služby. Druhá ze zmiňovaných možností bude hlavní náplní této části práce. Úkoly služby Sonic Connect jsou:

- transformovat data mezi `XQMessage` a XML zprávou,

- starat se o detaily HTTP spojení, které je nutné pro obsluhu příchozích a odchozích zpráv.

Jako RESTful webové služby mohou být nasazeny pouze ESB procesy typu REST. Tyto procesy mají předdefinované rozhraní, které se stará o mapování dat mezi HTTP požadavkem a odpovědí a ESB procesem. Zpráva vstupující do REST procesu má následující pole:

- **URI** – udává celou URI zdroje,
- **Method** – udává operaci, která se provádí nad zdrojem, v podobě jednoho z HTTP sloves,
- **URITemplate** – nese proměnnou část URI zdroje,
- **PostData** – obsahuje data odeslaná na adresu zdroje,
- **Accept** – udává MIME typ, který je odesilatel zprávy ochotný přijmout jako odpověď.

Všechna tato pole s výjimkou `PostData` jsou umístěna v hlavičce zprávy. V ní se dále nacházejí také hodnoty všech proměnných částí a query parametrů dané URI.

Při vygenerování REST ESB procesů je prvním krokem vždy směrování podle XPath, který směruje zprávu podle její hlavičky `Method` udávající typ operace, který se provede nad zdrojem. Toto směrování vytváří v procesu větve pro každé ze čtyř primárních HTTP sloves a pátou pro zprávy s neznámým požadavkem. Šablona REST procesu je uvedena v příloze A.1.

Rozhraní REST procesů dále určuje, že odchozí zprávy budou mít tato pole:

- **Response** – tělo zprávy, které obsahuje odpověď určenou odesilateli požadavku, může obsahovat libovolná data,
- **StatusCode** – hlavička zprávy nesoucí hodnotu HTTP stavového kódu vrácenou odesilateli.

Proces musí po skončení vyplnit hlavičku zprávy `StatusCode` platným HTTP stavovým kódem. Odesilateli požadavku je dále z odchozí zprávy jako odpověď vrácen pouze obsah pole `Response` v těle zprávy.

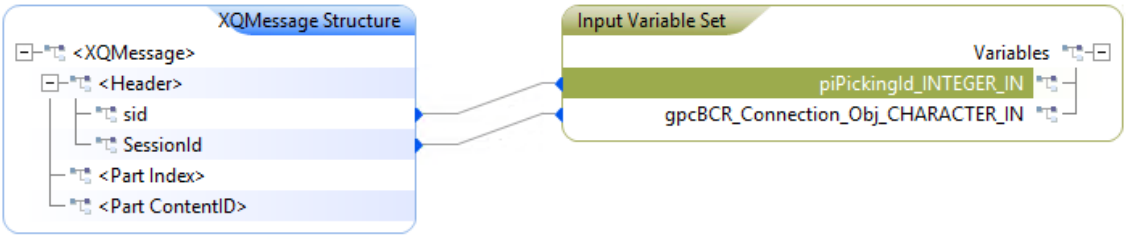
7.4.2 Volání ABL procedury z ESB služby

Jak již bylo dříve zmíněno v části 6.5.2, pro spolupráci programovacího jazyka OpenEdge ABL a Sonic ESB nabízí OpenEdge *Adaptér pro Sonic ESB* [23]. S tímto adaptérem budeme v našem případě pro vystavení ABL kódu jako služby na sběrnici ESB využívat přístup *nativní invokace* prostřednictvím služby *OpenEdge Native Service*.

Nejdůležitějším krokem pro úspěšné spuštění ABL kódu jako služby tímto způsobem je vygenerování jeho invokačního souboru (`.esboe`), což by za normálních okolností mělo být možné v prostředí *Progress Developer Studio for OpenEdge*. V mírně modifikované verzi tohoto prostředí pro systém proALPHA, které bylo k dispozici při implementaci praktické části této práce, však tato možnost nebyla funkční, a proto musel být invokační soubor generován prostřednictvím externího nástroje *Proxy Generator*.

Po vygenerování invokačního souboru je třeba jej předat jako běhový parametr vybrané instanci služby OpenEdge Native Service. Na základě tohoto souboru pak služba umožňuje

definovat mapování dat mezi `XQMessage` a parametry ABL procedury (viz obrázek 7.2). Kromě konkrétních hodnot z `XQMessage` je možné na parametry mapovat také konstanty nebo data z XML specifikované výrazem XPath.



Obrázek 7.2: Mapování polí `XQMessage` na vstupní parametry ABL procedury spouštěné přes OpenEdge Native Service. Obrázek byl získán z prostředí Sonic Workbench.

Jakmile má OpenEdge Native Service přiřazenou ABL proceduru, která se má volat, a namapovány její vstupní a výstupní parametry, je možné tuto instanci služby spouštět v rámci libovolného ESB procesu.

7.4.3 REST procesy a komunikace s adaptérem

Při implementaci procesu zprostředkovávajícím vrstvu mezi REST API a adaptérem využijeme službu Sonic Connect, která bude spouštět námi definované REST procesy. Tyto procesy budou vycházet ze šablony uvedené v příloze A.1 a každý z nich odpovídá jedné ze tří URI navržených v části 7.2.2. Všechny tyto procesy je možné najít v příloze A.

V těchto procesech budou implementovány vždy pouze ty větve, které korespondují s podporovanými HTTP metodami jednotlivých URI. Všechny ostatní větve budou vracet klientovi stavový kód `405 Method Not Allowed` a prázdné tělo odpovědi, aniž by došlo k jakékoliv interakci se systémem proALPHA. Speciální větev pro odchycení neexistující HTTP metody označená jako `FAULT` pak bude vracet stavový kód `400 Bad Request`.

Každá z implementovaných větví v sobě bude mít kromě kroku volání adaptéru také kroky `Login` a `Logout`, které se postarají o přihlášení, respektive odhlášení uživatele a získání uživatelského kontextu. Ten se při volání procedur přes appserver vždy předává jako jeden z parametrů a tudíž bez něj není možné spuštění žádné procedury na straně systému proALPHA. Z toho plyne, že tyto větve budou mít vždy právě tři kroky:

1. `Login` – přihlášení do systému proALPHA a získání uživatelského kontextu,
2. *Operace* – provedení operace požadované klientem,
3. `Logout` – odhlášení ze systému.

Jednotlivé kroky reprezentující operace nad systémem budou vykonány prostřednictvím služby OpenEdge Native Service popsané v předchozí části. Za každým z těchto kroků tedy bude stát externí procedura jazyka ABL fungující následujícím způsobem:

1. obdrží parametry získané z HTTP požadavku od klienta,
2. perzistentně spustí proceduru adaptéru,
3. na základě parametrů spustí příslušné interní procedury a funkce adaptéru,

4. podle jejich výsledku vrátí ESB procesu odpovídající stavový kód a odpověď.

Vstupní parametry těchto procedur se budou lišit podle toho, jakou operaci daná externí procedura realizuje. Typicky se bude jednat o identifikátory vychystávacího návrhu respektive jeho řádku, případně určení, které datové pole se má aktualizovat a na jakou hodnotu. Vstupní parametry těchto procedur budou podle potřeby popsány v části 9.1.2. Každá procedura bude mít také dva společné výstupní parametry:

- `opclResponse` – XML dokument s odpovědí pro klienta,
- `opiStatusCode` – stavový kód určený na základě výsledku běhu adaptéru.

Všechny tyto procedury budou dále vycházet ze standardní šablony systému proALPHA pro procedury volané vzdáleně na appserveru. Tato šablona jim přidává jako vstupní parametr `gpcBCR_Connection_Obj`, který slouží pro předání uživatelského kontextu.

Zdrojové kódy jazyka ABL stojící za modulem INWB, v jehož prostředí je implementována tato část práce, jsou z velké části zakryptovány a k procesům, které implementují, nebyla k dispozici dostatečná dokumentace. Z toho důvodu nebyly komponenty dostupné v tomto modulu využity při implementaci ESB procesů, ale sloužily pouze jako ideový vzor pro jejich návrh.

Kapitola 8

Implementace adaptéru pro externí přístup

Klíčovou částí celého systému pro externí přístup do ERP systému proALPHA je adaptér uložený v souboru `y_mpic00.p`, který poskytuje sadu procedur umožňující manipulaci s daty, konkrétně vychystávacími návrhy. Tyto procedury byly implementovány na základě analýzy zdrojových kódů, které potřebnou funkcionalitu zajišťují v systému proALPHA.

Jednou z největších komplikací při práci na adaptéru byla skutečnost, že proALPHA je z větší části vystavěná na dvouvrstvé architektuře a rozlišuje pouze *klienta* s uživatelským rozhraním a *server*, který se stará o databázi a k němuž klient přistupuje. Tyto vrstvy navíc nejsou příliš vhodně odděleny a z toho důvodu je téměř veškerá potřebná funkcionalita systému implementována přímo v souborech definujících uživatelské rozhraní, na které jsou také navázány jednotlivé proměnné.

Při implementaci adaptéru tedy hlavní náplň práce spočívala v identifikaci relevantních bloků zdrojových kódů, objasnění jejich významu a přerušení veškerých vazeb na uživatelské rozhraní tak, aby mohly být použity v rámci adaptéru. Ten v konečném důsledku reprezentuje prostřední vrstvu třívrstvé architektury, tj. zapouzdřuje logiku práce s databází a umožňuje s ní pracovat libovolnému klientovi. Úplný přehled procedur a funkcí, které k tomu adaptér poskytuje, je uveden v tabulce 7.3.

Analýzou uživatelského rozhraní systému proALPHA bylo zjištěno, že hlavní okno pro práci s vychystávacím návrhem je implementováno procedurou `mmwpic00.w`,[1] zdrojový kód okna se závislými položkami návrhu pak najdeme v proceduře `mmbpic01.w`. Tyto dvě externí procedury sloužily jako hlavní zdroj pro implementaci adaptéru. Přehled všech souborů a jejich relevantních částí, ze kterých vychází interní procedury a funkce adaptéru, je shrnut v tabulce 8.1. Tabulka 8.2 pak uvádí procedury a funkce, které lze přímo volat z adaptéru a propojit jej tak s nižšími úrovněmi systému proALPHA.

Při přerušování vazby mezi logikou pracující s daty a uživatelským rozhraním bylo nutné řešit, jakou hodnotou nahradit proměnnou čerpající z některého z polí hlavního, případně závislého okna, jak implementovat ošetření nevalidních vstupů řešené na úrovni uživatelského rozhraní a jak zachytit volání dialogů s chybovými hlášeními a přesměrovat je do seznamu vzniklých chyb generovaného adaptérem.

Jako příklad typického odstranění vazby na uživatelské rozhraní uveďme předělání hodnoty odkazující na uživatelské rozhraní prostřednictvím atributu `screen-value`:

```
ttMMT_Picking.PickStatus:screen-value in frame {&FRAME-NAME}.
```

[1] Přípona `.w` (`widget`) se používá pro označení ABL kódu implementujícího uživatelské rozhraní.

Soubor	Relevantní části
mmwpic00.w	local-assign-record
	ON LEAVE OF ttMMT_Picking.PickingStorage[2]
	ON LEAVE OF ttMMT_Picking.Specialist
mmbpic01.w	local-delete-prepare
	invokeSaveChanges
	setStatusOfPickLineToWorking
	checkUpdateOfField
	ON LEAVE OF ttMMT_PickingDetailView.AdoptionCode
	ON LEAVE OF ttMMT_PickingDetailView.PickStatusShortCut
	ON LEAVE OF ttMMT_PickingDetailView.PickedQty
	ON LEAVE OF ttMMT_PickingDetailView.PickingLocation
	ON LEAVE OF ttMMT_PickingDetailView.PickingStorage
v_webl00.w	Kommissionieren
MMCPickingSvcStd.cls	checkPickWflWorkGroupID
	checkFormNumber
b__alp00.cdf	definice globálních konstant

Tabulka 8.1: Přehled zdrojových kódů, které implementují logiku vybraných operací nad vychystávacími návrhy. Jejich význam bude podle potřeby upřesněn v dalších částech této kapitoly.

Volané funkce a procedury	Umístění
cMessageText	DMCMessageSvcStd.cls
oLastPAMessage	DMCErrorFrwStd.cls
vuvpic00.p	-
cDispValueByRefParamVal	DCCAppConfigSvcStd.cls
fillDataset	mmjpic00.p
checkAdoptCodeAndDocType	mmrpic00.p
applyFilterValues	obecné metody nad datasety
modifyTrackingChanges	
saveChanges	
emptyDataset	

Tabulka 8.2: Přehled funkcí a procedur systému proALPHA volaných z adaptéru a jejich umístění. Jejich význam bude podle potřeby upřesněn v dalších částech této kapitoly.

Tento zápis obvykle z našeho pohledu reprezentuje novou hodnotu zadanou uživatelem, která se nachází v zadávacím poli uživatelského rozhraní. V adaptéru ji tedy musíme na-

[2]Klíčová slova `ON LEAVE OF` definují *trigger* spouštěný při opuštění příslušného datového pole v uživatelském rozhraní.

hradit hodnotou obdrženou v rámci požadavku od klienta. V tomto konkrétním případě bychom použili zápis

```
integer(pcStatus).
```

Mimo těchto úprav bylo také nutné identifikovat kód, který souvisí pouze uživatelským rozhraním, tj. není pro adaptér relevantní, a vypustit ho. Jakým způsobem probíhalo odstranění další vazby, konkrétně potlačování dialogových oken, bude popsáno v části 8.1.1

Části původních zdrojových kódů uváděné v této kapitole nejsou referencovány do odpovídajících zdrojových souborů pomocí čísel řádků, jelikož jsou vytvořeny pro starší vývojové prostředí jazyka ABL AppBuilder. Ten externí proceduru zobrazuje po jejích blocích (interní procedury, funkce apod.) bez číslování řádků. Pro analýzu zdrojového kódu a určení jeho bloků zároveň používá tzv. *pokyny pro preprocesor* začínající symbolem `"&"`. Ty jsou vložené přímo do zdrojového kódu, což ještě dále komplikuje určení odpovídajícího čísla daného řádku. Definice procedury tedy má ve skutečnosti následující podobu:

```
&ANALYZE-SUSPEND _UIB-CODE-BLOCK _PROCEDURE procName Procedure
PROCEDURE getSuggestions :
  ...
END PROCEDURE.

/* _UIB-CODE-BLOCK-END */
&ANALYZE-RESUME
```

8.1 Generování seznamu vzniklých chyb

V této sekci představíme způsob, jakým budou ošetřovány chyby, ke kterým dochází v rámci adaptéru. Zároveň také bude podrobně popsána procedura `writeError` nahrazující zobrazování chybových dialogů.

Jak již bylo naznačeno v sekci 7.3, součástí adaptéru je také odchytávání chyb, které se vyskytly při práci v systému proALPHA a jejich následné předání aplikaci využívající služeb adaptéru. Tyto chyby jsou zaznamenávány do *seznamu chyb*, který je reprezentován jednoduchým XML se strukturou odpovídající schématu 8.1.

```
<xs:schema xmlns:xs = "http://www.w3.org/2001/XMLSchema">
  <xs:element name="errorList">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="error" maxOccurs="unbounded" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Schéma 8.1: Struktura XML se seznamem chyb.

Generování tohoto XML probíhá prostřednictvím SAX writeru, k jehož inicializaci dojde ihned po perzistentním spuštění externí procedury adaptéru. Jako cíl pro zápis seznamu je určena globální proměnná `gclErrorList`. O obsluhu chyb se starají tyto procedury a funkce:

- `writeError` – přidá zachycenou chybu do seznamu,
- `lGetAnyError` – vrátí hodnotu proměnné indikující, zda došlo k chybě,
- `clGetErrorList` – vrátí XML se seznamem chyb.

Implementace funkcí `lGetAnyError` a `clGetErrorList` je triviální a vzhledem k tomu, že slouží pouze pro získání odpovídajících proměnných, tedy `glAnyError` a `gclErrorList`, není důvod se jimi podrobně zabývat. U funkce `clGetErrorList` pouze zmiňme, že její součástí je také uzavření XML seznamu chyb vložením ukončovací značky `</errorList>`. Kvůli tomu není po zavolání této funkce možné přidávat do seznamu chyb žádné další položky a měla by tedy být volána jako poslední funkce adaptéru.

8.1.1 Procedura writeError

O něco komplikovanější je situace v případě procedury `writeError`. Než však začneme probírat detaily její implementace, podíváme se nejprve na obsluhu chyb v systému proALPHA. Z důvodu usnadnění lokalizace celého systému do požadovaného jazyka jsou všechna jeho chybová a jiná hlášení uložená v tzv. repozitářích, které jsou součástí jeho databáze. K vyvolání dialogového okna se pak použije pouze identifikátor hlášení, reprezentovaný řetězcem o pěti písmenech a pěti číslicích.

Obsluhu těchto hlášení a dialogů implementuje třída `DMCMessageSvcStd.cls`, se kterou systém pracuje jako se statickou knihovnou. Pokud v systému dojde k chybě a je třeba uvědomit uživatele, volá se její metoda `showError`. Ta obdrží jako parametr identifikátor chyby, případně další parametry, které určují proměnné části hlášení a postará se o vykreslení dialogového ona. Toto volání může vypadat například takto:

```
adm.method.cls.DMCMessageSvc:prpoInstance:showError('mmpic00123').
```

Abychom mohli chybová hlášení akumulovat v seznamu chyb, je nutné všechna tato volání nahradit procedurou `writeError`. Zároveň je také třeba dát možnost zapsat do seznamu vlastní chybu, neboť přesunutí logiky prováděných operací do námi implementovaného adaptéru rozšiřuje množinu možných chyb. Z toho plynou požadavky na parametry procedury `writeError`, které budou následující:

- `pcMessageCode` – identifikátor chybového hlášení, případně text vlastní chyby,
- `pcSubstitutionList` – seznam hodnot pro proměnná pole hlášení, předává se metodě `cMessageText` (viz dále),
- `plCustom` – indikátor, zda jde o vlastní chybu nebo se má její text vyhledat v repozitáři.

Součástí `writeError` je také nastavení příznaku `glAnyError` informujícího o tom, že při zpracování došlo k alespoň jedné chybě. Po něm následuje vytvoření řetězce, který bude zapsán do seznamu chyb, v závislosti na parametru `plCustom`. Podobu této procedury přibližuje pseudokód 8.1.

Analýzou metody `cMessageText`, která se stará o získáni textu chyby z repozitáře (řádek 8 uvedeného pseudokódu), bylo zjištěno, že požadovaný formát pro `pcSubstitutionList` je řetězec jednotlivých parametrů oddělených čárkou. Volitelně je možné specifikovat jiný oddělovač pomocí fráze `Delimiter=` na začátku celého řetězce. Jako příklad `pcSubstitutionList` uveďme řetězec `"Delimiter=/,first/second/third"`. Parametry chybového

Pseudokód 8.1 Vnitřní realizace procedury `writeError`.

```
1: glAnyError ← True
2: if zadán vlastní řetězec then
3:     e ← pcMessageCode
4: else
5:     if pcMessageCode je prázdný then
6:         e ← "Unknown error."
7:     else
8:         získej z repozitáře text pro chybu s kódem pcMessageCode
9:         e ← text chyby z repozitáře
10:    end if
11: end if
12: zapiš e do seznamu chyb
```

hlášení typicky použijeme, pokud v něm chceme zobrazit například uživatelem zadanou neplatnou hodnotu datového pole.

Na závěr popisu zachytávání chyb vzniklých během činnosti adaptéru ještě zmiňme, že existují případy, kdy jsou chybová hlášení generována na nižší úrovni než jsou soubory `mmwpic00.w` a `mmbpic01.w`. V takovýchto případech není možné vytvoření dialogu nahradit vlastní obsluhou, volání proběhne a hlášení je zapsáno do logů. Jediný způsob jak se k němu dostat, je uzavřít volání nižší úrovně do bloku a zachytit prostřednictvím příkazu `CATCH` skutečnost, že v něm došlo chybě. Následně získáme metodou `oLastPAMessage` třídy `DMCErrorFrwStd.cls` poslední zalogovaný záznam a ten vložíme do seznamu chyb na úrovni adaptéru.

8.2 Manipulace se záznamy

Pokud chceme v systému proALPHA pracovat se záznamem z databáze, není možné si jej pouze vytáhnout například příkazem `FIND` a provést požadované změny. Pokud bychom takto postupovali, dostali bychom databázi do nekonzistentního stavu, protože na nejnižší úrovni neexistuje žádné ošetření, které by implementovalo například ovlivnění závislých záznamů. Veškerá manipulace se záznamy z databáze by měla probíhat prostřednictvím ProDataSetů a operací nad nimi, které jsou v systému k dispozici. I tak je stále nutné určité kontroly implementovat zvlášť, jelikož proALPHA v některých případech spoléhá na omezení daná uživatelským rozhraním.

V případě vychystávacích návrhů, kterými se v této práci zabýváme, budeme pracovat s globálně definovaným datasetem `dsMMT_Picking`. Ten obsahuje dvě temporální tabulky, přičemž mezi nimi nejsou v rámci datasetu definovány žádné vztahy. Jedná se o tyto tabulky:

- `ttMMT_Picking` –je založená na databázové tabulce `MMT_Picking` obsahující vychystávací návrhy a nese hlavní informace o návrhu.
- `ttMMT_PickingDetailView` – reprezentuje jednotlivé řádky položek vychystávacího návrhu.

Všechny nestandardní operace nad datasetem definované systémem proALPHA jsou volány prostřednictvím *includu* souboru `call`, který svým chováním zastupuje operátor `":"`

(viz sekce 6.3). Include v jazyce ABL představuje doslovné vložení obsahu uvedeného souboru na dané místo zdrojového kódu. Include souboru je proveden uvedením jeho názvu ve složených závorkách `"{ }"`. Při tomto vložení je mu také možné předat dodatečné parametry. Nejdůležitější operace nad `dsMMT_Picking`, které budeme tímto způsobem volat jsou:

- `fillDataset` – naplnění datasetu záznamy podle definovaných filtrů (při vícenásobném volání se nové záznamy přidávají na konec),
- `applyFilterValues` – definice filtrů pro výběr záznamů, typicky nastavení identifikátoru pro získání jednoho konkrétního záznamu,
- `modifyTrackingChanges` – zapnutí sledování změn prováděných v datasetu (pokud není sledování zapnuto, žádné změny nebudou promítnuty do databáze),
- `saveChanges` – promítnutí zaznamenaných změn do databáze,
- `emptyDataset` – vyprázdnění datasetu.

Tento výčet nezachycuje všechny operace využité při implementaci adaptéru. Další operace nad datasetem budou vysvětleny až podle potřeby.

Naplněný dataset budeme z adaptéru vracet v rámci odpovědi na klientské požadavky. Jak jsme již dříve uvedli, výměna dat v implementovaném systému pro externí přístup bude probíhat výhradně přes formát XML. Pro export záznamů datasetu tedy budeme používat jeho metodu `WRITE-XML` zmiňovanou v sekci 6.4. Jelikož datasety systému proALPHA obsahují velké množství polí, nebudeme uvádět kompletní schéma jeho XML reprezentace, ale jen přibližně nastíníme jeho podobu bez detailů na nižší úrovni prostřednictvím schématu 8.2.

```
<xs:schema xmlns:xs = "http://www.w3.org/2001/XMLSchema">
  <xs:element name="dsMMT_Picking">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="ttMMT_Picking"
                    type="fullPicking"
                    maxOccurs="unbounded"/>
        <xs:element name="ttMMT_PickingDetailView"
                    type="fullPickingDetailView"
                    maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Schéma 8.2: Zjednodušené schéma XML reprezentujícího ProDataSet `dsMMT_Picking`.

8.3 Vytváření a získávání záznamů

Vytváření nových vychystávacích návrhů je externím klientům umožněno prostřednictvím vychystávání zakázek v systému. U nich je proto také nutné hledat zdrojové kódy, které

se starají o provedení této operace. Konkrétně nás bude zajímat hlavní okno *Odbyt* → *Zakázky* → *Zakázky* a v něm položka menu *Nástroje* → *Vychystávat zakázku*. Abychom identifikovali proceduru, na kterou je mapována tato položka menu, bylo nutné prohledat repozitář *Menu Designer*, ve kterém jsou tyto informace uloženy.

V repozitáři bylo zjištěno, že hledaný kód se nachází v proceduře `Kommissionieren` v souboru `v_webl00.w`. Kvůli vazbě na uživatelské rozhraní však není možné tuto proceduru volat přímo a její implementaci tedy bylo nutné replikovat v proceduře adaptéru `stageOrder` s následujícími úpravami:

- Vyhledání zakázky v databázi – původní procedura předpokládá, že záznam je již nalezen prostřednictvím uživatelského rozhraní.
- Potlačení uživateské interakce – odstranění volání dialogu `vupic00001` žádající potvrzení operace.
- Nahrazení zobrazení chybového hlášení voláním procedury `writeError`.
- Doplnění chybového hlášení v případě zadání neexistující objednávky, které přes uživatelské rozhraní nebylo možné.

Dále je ještě třeba naplnit požadavek na proceduru `stageOrder` z části 7.3.2, aby vracela identifikátor nově vzniklého návrhu. Ten vyhledáme příkazem `FIND LAST` jako poslední vychystávací návrh, jehož předek odpovídá zadané zakázce.

Pokud pak chceme prostřednictvím adaptéru získat jeden nebo více vychystávacích návrhů, máme k dispozici procedury `getSuggestionById` a `getSuggestions`. Ty jsou implementovány s využitím operací nad datasetem uvedených v části 8.2.

Jelikož je načtení vychystávacího návrhu se zadaným identifikátorem v adaptéru často používaná operace, byla její implementace vyčleněna do samostatné funkce `lFillDataset`. Ta očekává jako svůj vstup identifikátor návrhu `piPickingId` a vrací booleovskou hodnotu indikující úspěšnost hledání. Její podobu můžeme charakterizovat pseudokódem 8.2.

Pseudokód 8.2 Vnitřní realizace funkce `lFillDataset`.

```
1:  vyprázdni dataset dsMMT_Picking
2:
3:  vyhledej v databázi vychystávací návrh identifikovaný piPickingId
4:  if záznam nebyl nalezen then
5:      writeError("Invalid ID.", "", True)
6:      return False
7:  end if
8:
9:  načti záznam do dsMMT_Picking
10: return True
```

V proceduře `getSuggestionById` pak po zavolání této funkce už jen zbývá provést export datasetu metodou `WRITE-XML()` a vrátit výsledek.

Načítání záznamu do datasetu `dsMMT_Picking` je realizováno jeho metodou `fillDataset`, která vybere záznamy podle filtrů nastavených metodou `applyFilterValues`. Jelikož pro načtení všech všech vychystávacích návrhů nestačilo pouze nezadat filtry a nebyl zjištěn ani jiný způsob jak toho docílit, bylo nutné v proceduře `getSuggestions` projít všechny vychystávací návrhy a postupně je nahrát do datasetu, jehož XML reprezentace je vrácena jako výsledek. Tento postup shrnuje pseudokód 8.3.

Pseudokód 8.3 Vnitřní realizace procedury `getSuggestions`.

```
1: for each vychystávací návrh do
2:     vlož vychystávací návrh do dsMMT_Picking
3: end for
4: return XML export datasetu dsMMT_Picking
```

8.4 Mazání vychystávacích návrhů a jejich položek

V uživatelském rozhraní slouží pro smazání jednoho záznamu tlačítko *delete* v toolbaru, jehož implementaci sdílejí všechna hlavní okna v systému. Tato implementace vychází se standardního chování jazyka ABL. Díky tomu je možné jednoduše realizovat mazání vychystávacího návrhu prostřednictvím adaptéru procedurou `deleteSuggestionById` způsobem uvedeným v pseudokódu 8.4.

Pseudokód 8.4 Vnitřní realizace procedury `deleteSuggestionById`.

```
1: načti mazaný záznam do datasetu dsMMT_Picking
2: zapni sledování změn nad dsMMT_Picking
3: smaž nalezený záznam z tabulky ttMMT_Picking datasetu dsMMT_Picking
4: try
5:     promítni změny do databáze
6: catch chyba na nižší úrovni zpracování
7:     e ← text vzniklé chyby
8:     writeError(e, "", True)
9: end try
```

Všimněme si, že v této proceduře mažeme pouze záznam samotného vychystávacího návrhu (temporální tabulka `ttMMT_Picking`), nikoliv jeho jednotlivé položky. O smazání závislých položek se postará metoda `saveChanges` volaná v rámci promítnutí změn do databáze na 5. řádku uvedeného pseudokódu. Toto volání musí být rovněž ošetřeno příkazem `CATCH`, aby bylo možné odchytit případné chyby, jak bylo popsáno v 8.1.1.

U procedury `deleteLine` mazající jednu položku vychystávacího návrhu (temporální tabulka `ttMMT_PickingDetailView`) postupujeme obdobným způsobem jako v předchozím případě. Jediný rozdíl je ve vložení následujícího chování mezi řádky 1 a 2 pseudokódu 8.4:

- dohledání mazaného řádku v rámci nalezeného vychystávacího návrhu na základě vstupního parametru `pcInternalID`,
- provedení kontrol, zda je možné smazání provést.

Tyto kontroly se nacházejí v souboru `mmbpic01.w` v proceduře `local-delete-prepare`, která představuje přetížení standardní callback procedury volané před smazáním záznamu prostřednictvím toolbaru. Jelikož vazba mezi touto logikou a uživatelským rozhraním není tak silná jako v jiných případech, mohly být kontroly převzaty bez větších úprav. Proběhlo pouze předělání obsluhy chyb na volání `writeError`, přičemž každá chyba je adaptérem považována za fatální, tzn. po libovlné chybě procedura ihned končí a smazání položky neproběhne.

8.5 Editace vychystávacího návrhu

V této části budeme podrobně rozebírat pouze implementaci `updateSuggestionById` modifikující data samotného vychystávacího návrhu. Editace jeho jednotlivých položek bude z důvodu velkého rozsahu a odlišného přístupu rozebrána samostatně.

Jak již bylo řečeno v části 7.3.3, na vstupu procedury `updateSuggestionById` očekáváme XML dokument reprezentující modifikovaný vychystávací návrh. Ve skutečnosti však není nutné, aby se přenášela všechna data, nýbrž stačí přenést pouze ta pole, jejichž hodnoty hodláme měnit. Plné funkcionality tak můžeme dosáhnout i s dokumenty respektujícími minimalistické schéma 8.3.

```
<xs:schema xmlns:xs = "http://www.w3.org/2001/XMLSchema">
  <xs:element name="dsMMT_Picking">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="ttMMT_Picking" type="fullPicking" maxOccurs="1"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>

<xs:complexType name="fullPicking">
  <xs:complexContent>
    <xs:all>
      <xs:element name="PickingStorage"  type="xs:integer"/>
      <xs:element name="PickingLocation" type="xs:string"/>
      <xs:element name="FormularNr"      type="xs:integer"/>
      <xs:element name="FormularAnzahl"  type="xs:integer"/>
      <xs:element name="PickStatus"      type="xs:integer"/>
      <xs:element name="Specialist"      type="xs:string"/>
      <xs:element name="WflWorkGroupID"  type="xs:string"/>
    </xs:all>
  </xs:complexContent>
</xs:complexType>
```

Schéma 8.3: Doporučené schéma XML nesoucího nové hodnoty editovatelných datových polí vychystávacího návrhu.

Toto XSD nelze brát zcela závazně, neboť adaptér je velmi benevolentní ohledně přijímaného formátu. Všimněme si, že z komplexního typu `fullPicking` jsou díky `<xs:all>` všechna pole volitelná a na jejích pořadí nezáleží. Jakékoliv nadbytečné elementy, které nejsou ve schématu uvedeny, budou jednoduše ignorovány, a to včetně `ttMMT_PickingDetailView` s řádky vychystávacího návrhu nebo dalších vychystávacích návrhů. Při vícenásobném uvedení některého z elementů `fullPicking` se použije poslední uvedená hodnota. Mapování elementů na editovatelná datová pole popsaná v sekci 7.1 je zachyceno v tabulce 8.3. Samotnou proceduru `updateSuggestionById` pak můžeme shrnout pseudokódem 8.5.

Identifikátor záznamu v globální proměnné `gcCurrentPicking` je využíván při zpracování vstupního XML (viz část 8.5.1) spouštěného na řádku 8 uvedeného pseudokódu.

Datové pole	XML element
Komisionářský sklad	`PickingStorage`
Vychystávací místo	`PickingLocation`
Referent	`Specialist`
Stav vychystávání	`PickStatus`
Rozdělovník	`WflWorkGroupID`
Číslo formuláře	`FormularNr`
Počet formulářů	`FormularAnzahl`

Tabulka 8.3: Přiřazení XML elementů jednotlivým datovým polím vychystávacího návrhu, jejichž editace je možná prostřednictvím adaptéru.

Pseudokód 8.5 Vnitřní realizace procedury `updateSuggestionById`.

```
1: načti upravovaný vychystávací návrh do datasetu dsMMT_Picking
2: gcCurrentPicking ← identifikátor nalezeného záznamu
3:
4: vytvoř nový objekt sax-reader
5: inicializuj vytvořený sax-reader
6:
7: zapni sledování změn nad dsMMT_Picking
8: proveď zpracování vstupního XML pomocí sax-reader
9: try
10:     promítni změny do databáze
11: catch chyba na nižší úrovni zpracování
12:     e ← text vzniklé chyby
13:     writeError(e, "", True)
14: end try
```

Přestože uvádíme, že editace vychystávacích návrhů je implementována procedurou `updateSuggestionById`, hlavní logika spojená s touto operací je ve skutečnosti vykonávána v rámci analýzy vstupního XML, tj. v callback procedurách, které využívá `SAX-READER`. Tyto procedury jsou součástí adaptéru, aby nebylo nutné rozdělovat jeho zdrojový kód do více externích procedur. Při zpracování jsou využity callback procedury

- `startElement` – pouze nuluje proměnnou pro akumulaci textových dat,
- `characters` – akumuluje textová data uvnitř elementu do proměnné,
- `endElement` – implementuje logiku editace (viz dále).

8.5.1 Procedura endElement

Callback procedura `endElement` se volá vždy při detekci koncové značky ve zpracovávaném XML. V této fázi už tedy máme načtený celý její obsah, představující hodnotu, která se má nastavit pro pole dané názvem uzavíraného elementu. Na nejvyšší úrovni je tato procedura implementována s využitím konstrukce *switch - case*.

Odpovídající větev kódu pro vykonání příslušné operace je vybírána na základě vstupního parametru `pcQName` nesoucího název XML elementu, který byl právě ukončen. Každá

větev je implementována přibližně stejným způsobem daným pseudokódem 8.6. Je-li globální proměnná `gcCurrentPicking` (viz sekce 8.5) prázdná, editace již byla dříve ukončena a všechny další načítané hodnoty jsou tudíž ignorovány. Ukončení editace a vynulování této proměnné proběhne při přečtení značky `</ttMMT_Picking>` ukončující vychystávací návrh.

Pseudokód 8.6 Přibližná podoba jedné větve procedury `endElement`.

```
1: case název editovaného pole
2:     if gcCurrentPicking není prázdná then
3:         if nová hodnota je validní then
4:             nastav příslušné pole na novou hodnotu
5:         end if
6:     end if
```

Pokud je třeba, aby se změna pole projevila i v položkách editovaného návrhu, použije se volání operace `applyChangeToMates` nad datasetemn jako součást logiky implementující řádek 4 pseudokódu 8.6. Ta bere jako parametry mimo jiné název pole, jehož hodnotu měníme, a hodnotu, na kterou jej nastavujeme. Tato hodnota je téměř ve všech případech získávána z uživatelského rozhraní prostřednictvím atributu `screen-value` a bylo nutné ji nahradit hodnotou načtenou z XML.

Téměř veškerou editaci polí a s ní spojené kontroly implementuje proALPHA prostřednictvím procedury `local-assign-record` volané při potvrzení změn přes toolbar v hlavním okně, tj. v souboru `mmwpic00.p`. V této proceduře bylo nutné identifikovat kód související s jednotlivými datovými poli a jeho funkcionalitu vhodně rozdělit mezi jednotlivé větve procedury `endElement`. V případě polí `PickingStorage` a `Specialist` se pak ještě část logiky nachází v triggerech obsluhujících opuštění zadávacího pole příslušné hodnoty.

V následujících částech se podíváme podrobněji na jednotlivé větve procedury `endElement`, přičemž jejich náročnost se liší.

PickingLocation a FormularAnzahl

V poli `PickingLocation` je povolená libovolná řetězcová hodnota, díky čemuž není nutné řešit žádné kontroly validity. Načtený řetězec se pouze přiřadí do odpovídající proměnné a změny se promítnou do závislých záznamů voláním `applyChangeToMates`.

Stejně jednoduché je i nastavení hodnoty počtu formulářů `FormularAnzahl`, kde není nutné řešit ani aplikaci změny na položky návrhu. Jediný požadavek na tuto hodnotu je, aby se jednalo o číslo od 0 do 9.

Specialist a PickingStorage

V případě změny pole `Specialist` je před samotným přiřazením nutné provést kontrolu metodou `checkPickingSpecialist` třídy `MMCPickingSvcStd.cls`, která se provádí v příslušném triggeru. Validaci implementuje interní procedura adaptéru `lIsSpecialistValid`.

Identická je i situace v případě pole `PickingStorage` pouze s tím rozdílem, že změnu je nutné promítnout do závislých záznamů.

WflWorkGroupID a FormularNr

Změny rozdělovníku `WflWorkGroupID` a čísla formuláře `FormularNr` probíhají stejným jednoduchým způsobem, který přesně koresponduje s pseudokódem 8.6. Při přiřazení nové hodnoty do jednoho z těchto polí není nutné provádět žádné dodatečné operace.

Problém u tvorby validačních funkcí volaných v rámci 3. řádku pseudokódu 8.6 byl v tom, že na úrovni souboru `mmwpic00.p` neprobíhá žádná kontrola jejich validity. Analýzou volaných procedur a funkcí bylo zjištěno, že tyto kontroly probíhají na nižší úrovni s využitím třídy `MMCPickingSvcStd.cls` a jejích metod `checkPickWflWorkGroupID` a `checkFormNumber`. Tyto metody tedy byly použity pro implementaci kontroly v adaptéru.

Přestože nejsme schopni potlačit volání chybových dialogů na této úrovni, byl využit alespoň fakt, že v každé z nich může dojít pouze k jednomu typu chyby. Při ošetření volání metod `CATCH` blokem tak mohl být proceduře `writeError` předán odpovídající identifikátor chyby a nebylo nutné definovat vlastní chybový řetězec.

PickStatus

Asi nekomplikovanější ze všech větví byla editace pole `PickStatus`. Samotné přiřazení nové hodnoty je implementováno interní procedurou adaptéru `updateStatus`, která jako svůj parametr bere novou hodnotu stavu vychystávání `pcStatus`. Tuto proceduru bychom mohli zjednodušeně popsat pseudokódem 8.7.

Pseudokód 8.7 Přibližná podoba procedury `updateStatus`.

```
1: if pcStatus představuje stav "Nový" nebo "Ve zpracování" then
2:     writeError("mmpic00042", "", no)
3: else
4:     if lIsStatusValid(pcStatus) then
5:         stav vychystávání ← pcStatus
6:         checkIfStaged()
7:     end if
8: end if
```

Procedura `checkIfStaged`, jejíž volání následuje za kontrolou validity, se stará o korektní provedení změn ve zbytku systému v případě zadání stavu *"Vychystáno"*. Pokud byl zadán jakýkoliv jiný stav, proběhne v systému proALPHA kontrola, zda jsou všechny položky návrhu vychystány, a pokud ano, uživatel je prostřednictvím dialogu dotázán, jestli nechce zadat vychystání celého návrhu. Jelikož v adaptéru potřebujeme odstranit přímou uživatelskou interakci, předpokládáme implicitně negativní odpověď a tudíž tento kód nebyl v adaptéru vůbec použit.

Funkce `lIsStatusValid` kontroluje, zda hodnota `pcStatus` odpovídá jedné z hodnot stavu vychystávání, které jsou uvedeny v tabulce 8.4. V systému proALPHA je tato kontrola řešena na úrovni uživatelského rozhraní rozbalovací nabídkou, která neumožní zadání nevalidní hodnoty. Abychom ji mohli korektně implementovat v adaptéru, bylo nutné najít úsek kódu, který toto umožní.

Analýzou spouštěných procedur a funkcí bylo zjištěno, že zadání nevalidní hodnoty stavu způsobí výjimku v metodě `cDispValueByRefParamVal` třídy `DCCAppConfigSvcStd.cls`, která vrací odpovídající řetězcovou reprezentaci zadaného klíče (např. pro hodnotu `10` v našem případě vrátí řetězec *"Nový"*). Funkce `lIsStatusValid` tedy předloží novou hodnotu

Hodnota pole	Stav
10	*Nový*
20	*Ve zpracování*
30	*Částečně vychystaný*
40	*Vychystaný*

Tabulka 8.4: Možné hodnoty elementu `PickStatus` a odpovídající stavy vychystávání. Definice těchto hodnot se nachází v souboru `b__alp00.cdf`.

stavu této metodě a na případnou výjimku reaguje voláním `writeError` a návratovou hodnotou `False`.

8.6 Editace položek vychystávacího návrhu

Jak jsme již uvedli a zdůvodnili v sekci 7.2.2, editaci řádků položek vychystávacích návrhů nebudeme implementovat jedinou procedurou, nýbrž definujeme jednu proceduru pro každé editovatelné pole. V této sekci se budeme postupně zabývat implementačními detaily všech těchto procedur.

Vzhledem ke způsobu, jakým proALPHA umožňuje editace těchto dat, tj. vše lze editovat současně, je efektivní kód zprostředkovávající tuto funkcionalitu roztroušen mezi interními procedurami souboru `mmbpic01.w` (viz tabulka 8.1). Tento kód zároveň obsahuje jen malé množství volání použitelných operací na nižší úrovni zpracování, což znamená, že je velká část logiky implementována na této úrovni a je nutné ji v adaptéru rekonstruovat.

Její hlavní část je umístěna v triggerech reagujících na opuštění odpovídajících prvků uživatelského rozhraní a proceduře `invokeSaveChanges` volané při potvrzení provedených změn. Identifikace všech těchto částí, jejich správná interpretace a následná rekonstrukce byla časově jednou z nejnáročnějších činností celé práce.

U všech procedur, které implementují editaci jednoho řádku vychystávacího návrhu je použita přibližně stejná posloupnost prováděných akcí (pseudokód 8.8), která byla vytvořena na základě analýzy zdrojového kódu `mmbpic01.w`.

Identifikátor nadřízeného řádku získaný na řádku 8 uvedeného pseudokódu hraje roli při vykonávání některých akcí v rámci editace. V případě závislého řádku, tj. typ `StorageLine`, je tato hodnota nastavena na identifikátor hlavního řádku, pod který závislý spadá. Pokud ale pracujeme s hlavním řádkem (typ `Line`), je uložen jeho vlastní identifikátor. Princip hlavních a závislých řádků vychystávacích návrhů byl vysvětlen v sekci 3.1

ProALPHA rozlišuje ještě další dva typy řádků, konkrétně `SetLine` a `OnHand`. Význam těchto typů se však nepodařilo objasnit a ani při experimentech se systémem se nepovedlo vytvořit situaci, kdy by některý z těchto řádků vznikl. Z toho důvodu v adapatéru předpokládáme, že řádek bude vždy buď typu `StorageLine`, nebo `Line`. Jiný typ způsobí chybu ošetřenou voláním `writeError`.

Pseudokód 8.8 Obecný postup při editaci vybraného pole řádku vychystávacího návrhu.

```
1: načti upravovaný vychystávací návrh do datasetu dsMMT_Picking
2: zapni sledování změn nad dsMMT_Picking
3: najdi upravovaný řádek v rámci získaného návrhu
4: if řádek nebyl nalezen then
5:     writeError("Invalid line ID.", "", True)
6:     return
7: end if
8: podle typu řádku získej jeho nadřízený řádek
9:
10: proveď všechny potřebné kontroly validity
11: editované pole ← nová hodnota
12:
13: promítni změny do případných závislých řádků
14: proveď případné dodatečné operace a kontroly
15: stav vychystávání ← "Ve zpracování"
16: try
17:     promítni změny do databáze
18: catch chyba na nižší úrovni zpracování
19:     e ← text vzniklé chyby
20:     writeError(e, "", True)
21: end try
```

8.6.1 setStatusOfPickLineToWorking

Než se začneme zabývat analýzou jednotlivých procedur realizujících editaci položek, rozebereme proceduru `setStatusOfPickLineToWorking` implementující řádek 15 pseudokódu 8.8. Tato procedura vychází ze stejnojmenné procedury v souboru `mmbpic01.w`.

Jejím úkolem je nastavit při změně některé z položek stav celého vychystávacího návrhu na *"Ve zpracování"* a tím dát najevo, že s návrhem již byly provedeny nějaké operace. Stejná změna stavu proběhne i u nadřízeného řádku položky, pokud pracujeme se závislým.

Prvním krokem procedury je kontrola, zda má vůbec smysl změnu provádět. Abychom tuto podmínku pochopili a mohli ji řádně zrekonstruovat, bylo nutné dohledat význam nezdokumentované globální proměnné `gcPickStatusShortOnRowEntry`. Na základě získaných poznatků bylo dosaženo závěru, že část podmínky pracující s touto proměnnou ověřuje, zda již dříve během editace nedošlo ke změně stavu. Tato část byla nahrazena booleovským parametrem obdrženým od volající procedury.

Dále bylo také nutné řešit chybu v této podmínce, konkrétně v následujícím porovnání:

```
  ttMMT_PickingDetailView.PickStatusShortCut:screen-value
= {&pa_MM_PickStatus-New}
```

Zatímco hodnota `PickStatusShortCut` v sobě nese řetězcovou reprezentaci stavu vychystávání, hodnota `{&pa_MM_PickStatus-New}` je definována v `b__alp00.cdf` jako konstanta `10`. Tato část tudíž zákonitě nikdy nemůže nabýt hodnoty `True`. Jelikož nelze s jistotou určit její pravý význam a nepovedlo se ani vyvolat využitím této chyby nepředvídané chování, byla část nahrazena konstantou `False`, což vzhledem k jejímu umístění znamenalo možnost ji úplně vypustit.

V dalších částech procedury `setStatusOfPickLineToWorking` se již nevyskytují žádné komplikace a pokud je splněna úvodní podmínka, proběhne následující posloupnost úkonů:

1. změna stavu zadaného řádku,
2. vyhledání nadřazeného řádku,
3. změna stavu nadřazeného řádku,
4. změna stavu celého vychystávacího návrhu.

8.6.2 updateLinePickStatus

Jelikož procedury zajišťující editaci jednoho řádku vychystávacího návrhu vykazují určité podobnosti a jejich popis je poměrně obsáhlý, budeme se podrobně zabývat pouze procedurou `updateLinePickStatus`, která je z nich nejkomplikovanější. Všechny ostatní procedury jsou popsány v příloze B.

Při implementaci `updateLinePickStatus` bylo nutné brát v úvahu, že proALPHA tuto operaci provádí prostřednictvím pole `PickStatusShortCut` s řetězcovou reprezentací stavu, zatímco adaptér pracuje s polem `PickStatus` nesoucím jeho identifikátor. Tuto skutečnost bylo nutné zohlednit při rekonstrukci relevantního kódu.

Hlavní část procedury čerpá z odpovídajícího triggeru, v němž můžeme nalézt i *retry* blok realizující anulování změn v případě chyby. Jelikož adaptér změny ukládá do systému až po bezchybném provedení celé procedury, není nutné se tímto blokem zabývat.

Prvním krokem editace je kontrola prostřednictvím procedury `checkUpdateOfField`, z jejíž příslušné větve čerpá funkce adaptéru `lCanUpdateStatus`. Kontrolu představují celkem dvě podmínky, jestli je v daném poli přípustná změna na požadovaný stav. Tyto podmínky mohly být převzaty bez větších změn. Dále do `lCanUpdateStatus` přibyla ještě kontrola, zda se uživatel nepokusil zadat stav *"Nový"*, jenž není nikdy přípustný jako nová hodnota.

Po této kontrole následuje blok podmíněný výrazem

```
IF      gcOriginDocType <> 'PPA'
   AND last-event:function <> 'CHOOSE'
   ...
```

Z tohoto výrazu byla odstraněna část `last-event:function <> 'CHOOSE'`, která souvisí výhradně s uživatelským rozhraním. Následné volaní operace `Package` nad datasetem bylo ošetřeno blokem `CATCH`. Globální proměnná `gcMMTPickLineObj` byla identifikována jako předek editovaného řádku a nahrazena odpovídající hodnotou.

Přestože jsme již dříve uvedli, že jakýkoliv jiný typ řádku než `StorageLine` nebo `Line` považujeme za chybu, byla do adaptéru převzata i kontrola

```
IF ttMMT_PickingDetailView.PickLineType = {&pa_MM_PickLineType-SetLine}.
```

Pokud je tato podmínka splněna, editace bude přerušena.

Není-li nová hodnota stavu *"Vychystáno"*, následuje volání `setStatusOfPickLineToWorking`. V opačném případě proběhne vyhodnocení podmínky obsahující mimo jiné test, zda je editovaný řádek typu `OnHand`:

```
IF     ttMMT_PickingDetailView.PickLineType = {&pa_MM_PickLineType-OnHand}
   AND NOT CAN-FIND(...
```

Vzhledem k jeho umístění by, stejně jako v předchozím případě, v adaptéru nikdy nemělo dojít k jejímu splnění, i přes to je však její blok pro úplnost převzat. Jeho rekonstrukce vyžadovala potlačení uživatelských dialogů navrhujících úpravy dalších hodnot v systému. U obou dialogů předpokládáme zápornou odpověď, abychom minimalizovali automatické změny, které by nemuseli být žádoucí.

Stejný přístup je aplikován i v případě dialogu navrhující nastavení nadřazeného řádku na stav *"Vychystáno"*, pokud jsou již vychystány všechny jeho položky. Zamítnutí této operace znemožní kladné vyhodnocení celé komplikované podmínky, která tento test provádí, a je tedy možné ji zcela vypustit včetně jejího těla.

V následujícím bloku je opět použita kontrola, zda je editovaný řádek typu `OnHand`. I v tomto případě je celý blok převeden obvyklým způsobem do adaptéru. V jeho podmínce bylo dále nutné předělat testovaní hodnoty `PickStatusShortCut` na `PickStatus`.

Operace které následují za tímto blokem jsou opět podmíněny dialogem navrhujícím změnu stavu celého vychystávacího návrhu na *"Vychystáno"*, který je potlačen. Použita je až část volající metodu `applyChangeToMates` včetně výrazu, kterým je podmíněna. Poslední akce triggeru, která je z našeho pohledu významná, je volání `checkUpdateOfField` pro pole `AdoptionCode`. Adaptér jej implementuje voláním funkce `lCanUpdateAdoptCode` s klíčem aktuální hodnoty tohoto pole.

V proceduře `invokeSaveChanges` pak byly identifikovány pouze dvě kontroly vztahující se na novou hodnotu stavu vychystávání. Výrazy, které je implementují, jsou poměrně komplikované a nebudeme se podrobně zabývat jejich významem. Uvedeme pouze, že při rekonstrukci v adaptéru na ně byly aplikovány obvyklé transformace pro odstranění vazby na uživatelské rozhraní. Po těchto kontrolách již následuje promítnutí provedených změn do databáze voláním `saveChanges`, které je ošetřeno blokem `CATCH`.

Kapitola 9

Realizace externího přístupu

V kapitole 7 jsme definovali celkovou podobu systému pro externí přístup a nastínili rozhraní jeho jednotlivých částí. Nyní se budeme podrobněji věnovat jeho realizaci, tj. postupně probereme implementační detaily navržených částí, a to především procedur představujících mezivrstvu mezi adaptérem a Sonic ESB. V závěru této kapitoly také navrhneme a implementujeme jednoduchého externího klienta, jehož prostřednictvím budeme demonstrovat funkčnost celého systému. Implementace adaptéru byla vzhledem ke svému velkému rozsahu probrána samostatně v rámci kapitoly 8.

9.1 ESB procesy a jejich interakce s okolím

Hlavním úkolem ESB procesu v systému pro externí přístup je implementovat navržené aplikační rozhraní REST API a transformovat klientské požadavky na něj do podoby volání jednotlivých procedur a funkcí adaptéru na straně systému proALPHA. K tomu bude použita služba Sonic Connect a šablona pro REST procesy, které byly rozebírány v části 7.4.1.

9.1.1 Realizace potřebných procesů

V našem případě budeme při vytváření RESTful webové služby implementující externí přístup do systému proALPHA vycházet z dříve navržených URI, tj. využijeme postup *shora dolů* popsaný v [21].

Prvním krokem tohoto postupu je vytvoření nového projektu ve vývojovém prostředí Sonic Workbench, který bude typu Sonic Connect. Tento projekt obsahuje všechny soubory a jiné prvky potřebné k realizaci webové služby prostřednictvím ESB procesu, které jsou generovány ze standardních šablon. Zároveň také obsahuje jednu instanci služby Sonic Connect, v níž můžeme dále definovat jednotlivé webové služby a klienty, jež bude mít na starosti.

Každou ze tří URI, které je třeba implementovat, budeme chápat jako jednu RESTful webovou službu. Při přidávání služeb do Sonic Connect přístupem shora dolů definujeme následující paramtery:

- **Base URL** – neměnná část URI specifikující na jakém rozhraní služba poběží,
- **Display Name** – pojmenování služby v rámci Sonic Connect,
- **Resource Template** – URI šablona daného zdroje (viz tabulka 7.1),

- **ESB Process Name** – ESB proces typu REST, který bude zdroj implementovat.

Base URL je stejná v případě všech tří služeb, konkrétně `http://0.0.0.0:port/staging`. Adresa `0.0.0.0` značí, že po spuštění bude služba naslouchat na všech IP adresách, přes které je možné přistupovat ke stanici, na níž běží. Hodnoty Display Name byly pro přehlednost nastaveny tak, aby korespondovaly s názvy ESB procesů, které je implementují.

Vytvořením webové služby v rámci Sonic Connect dojde k vygenerování REST procesu implementujícího URI vytvořené služby (jeden pro každou URI) a souboru `.xcbr`, který se stará o směrování `XQMessage` v REST procesu podle HTTP metody příchozího požadavku. V tabulce 9.1 je uveden přehled všech procesů, které ve vytvářeném systému existují a jakou URI implementují. Výsledná URL zdroje, na kterou budou klienti generovat požadavky pak vznikne spojením Base URL a URI šablony. Výsledek může mít například podobu `http://127.0.0.1:18787/staging/suggestions`

ESB proces	Implementovaná URI
`suggestions.esbp`	`/suggestions`
`suggestionsID.esbp`	`/suggestions/{sid}`
`linesID.esbp`	`/suggestions/{sid}/lines/{lid}`

Tabulka 9.1: ESB procesy v systému pro externí přístup a URI, které jsou jimi implementovány.

Po vygenerování procesů následuje implementace jednotlivých větví realizující potřebné operace. Abychom v případě neexistující nebo nepovolené HTTP metody docílili vrácení požadovaného stavového kódu, jak bylo definováno v tabulce 7.2, jsou všechny odpovídající výstupní endpointy nahrazeny chybovými. V chybových endpointech je pak provedeno mapování konstantní hodnoty stavového kódu na pole `StatusCode` v hlavičce odchozí zprávy definované rozhraním procesu typu REST.

Komplikace se Sonicem v INWB

Na závěr definice procesů v Sonic ESB ještě zmiňme problémy, které bylo nutné řešit při implementaci této části práce. Sonic ESB byl, stejně jako Progress Developer Studio, přizpůsoben potřebám systému proALPHA, což vedlo na komplikace při vývoji.

Jedním ze závažných problémů bylo odebrání standardních komponent Sonicu, které nebyly nezbytně nutné pro běh samotného INWB. To zahrnovalo jak většinu standardních služeb popisovaných v sekci 5.3, tak i šablony potřebné ke generování REST procesů a dalších souborů souvisejících s projektem typu Sonic Connect. Aby bylo možné realizovat navrhované ESB procesy, bylo nutné všechny potřebné komponenty manuálně vrátit na Domain Manager a podle potřeby pro ně vytvořit odpovídající endpointy.

Dále se negativně projevilo například odlišné chápání adresářové struktury Domain Manageru a zavedení vlastních jmenných konvencí, které se liší od konvencí doporučených manuálem nástroje Sonic. Tyto odlišnosti nebyly reflektovány v dostupném prostředí Sonic Workbench, což komplikovalo například aktualizace konfiguračních souborů projektu na Domain Manageru.

V rámci této práce byla snaha dodržovat konvence Sonicu, a proto je tato část systému implementována bez většího ohledu na prostředí vytvořené na Domain Manageru modulu INWB. Celý projekt včetně všech svých zdrojů byl nahrán do adresáře Domain Manageru

workspace. Pro systém byl vytvořen zvláštní kontejner pojmenovaný `ct_proalpha_REST` a v něm byl podle konvence definován ESB kontejner se shodným názvem. V rámci něj pak běží instance služby Sonic Connect `RESTstaging` implementující RESTful webovou službu pro externí přístup do systému proALPHA.

9.1.2 Propojení ESB procesu a adaptéru

Když máme vytvořen projekt s nakonfigurovanou instancí Sonic Connect a všechny potřebné ESB procesy pro realizaci navržené RESTful webové služby, je dalším krokem této práce vhodným způsobem implementovat jednotlivé větve těchto procesů. Konečná podoba všech tří REST procesů se nachází v příloze A.

Jak již bylo zmíněno v části 7.4.3, každá z těchto větví bude mít tři kroky:

1. přihlášení do systému proALPHA,
2. provedení požadované operace,
3. odhlášení ze systému.

Přihlášení do systému je provedeno stejným způsobem, jaký je používán v ostatních procesech modulu INWB. S využitím OpenEdge Native Service je na appserveru systému proALPHA zavolána externí procedura `paasconn.p`, která klientovi přidělí na základě přihlašovacích údajů uživatelský kontext, jímž se identifikuje ve všech procedurách, které hodlá spouštět na appserveru. Tato procedura se volá v rámci kroku `Login` a její vstupní parametry jsou

- `gpcServiceID` – identifikace služby, která do systému přistupuje,
- `gpcUserID` – uživatelské jméno,
- `gpcPassword` – heslo uživatele,
- `gpcInfo`.

Jako hodnota `gpcServiceID` je nastavena konstanta `BDE`, která se používá pro externí služby. Vzhledem k tomu, že zabezpečení navrhovaného systému je nad rámec zadání této práce, byl v systému proALPHA pro jednoduchost vytvořen speciální uživatel *ExternalUser*, jehož uživatelské jméno a heslo jsou mapovány jako konstanty na parametry procedury `paasconn.p`. Proměnná `gpcInfo` není relevantní a je jí přiřazen prázdný řetězec.

Výsledkem této procedury je uživatelský kontext předaný prostřednictvím výstupního parametru `gopcSessionID`, který se dále předává jako jeden z parametrů každé proceduře volané vzdáleně na appserveru. Kontext se také předává jako jediný parametr proceduře `paasdisc.p` zprostředkovávající odhlášení ze systému. Její volání probíhá v rámci kroku `Logout`.

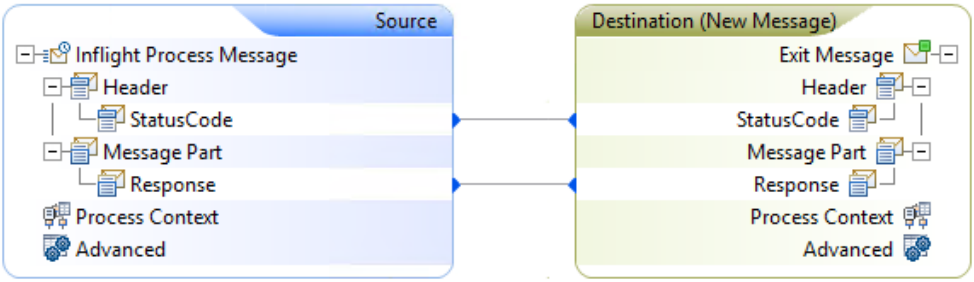
Mezi kroky `Login` a `Logout` se nachází asi nejdůležitější krok každé z implementovaných větví jednotlivých REST procesů, a to konkrétně vykonání požadované operace. Tento krok vždy volá prostřednictvím OpenEdge Native Service odpovídající externí proceduru zprostředkovávající mezivrstvu mezi REST procesem a adaptérem. Přehled těchto procedur je uveden v tabulce 9.2.

Na konci každé z větví pro podporované HTTP metody je potřeba namapovat hodnoty zprávy `XQMessage` uvnitř procesu na zprávu, která bude proces opouštět a která musí splňovat požadavky rozhraní REST procesu. Jelikož rozhraní dodržuje už zpráva, se kterou

Procedura	Proces	Větev	Význam
y_nsug00.p	suggestions.esbp	GET	Vrací všechny vychystávací návrhy.
y_nsug03.p		POST	Vytváří vychystávací návrh.
y_nsug01.p	suggestionsID.esbp	GET	Vrací zadaný vychystávací návrh.
y_nsug02.p		DELETE	Maže zadaný vychystávací návrh.
y_nsug04.p		PUT	Upravuje vychystávací návrh.
y_nsul00.p	linesID.esbp	DELETE	Upravuje a maže řádky položek.
		PUT	

Tabulka 9.2: Přehled procedur realizujících komunikaci ESB procesů implementujících jednotlivé URI a jejich význam. Procedury jsou pojmenovány podle konvencí zavedených v systému proALPHA.

se pracuje uvnitř procesu, bude mít mapování ve výstupních endpointech za úkol pouze vytvoření nové zprávy opouštějící proces a zkopírování odpovídajících polí, jak ukazuje obrázek 9.1.



Obrázek 9.1: Mapování polí mezi zprávou uvnitř ESB procesu a novou zprávou opouštějící proces skrz výstupní endpoint. Obrázek byl získán z prostředí Sonic Workbench.

Procedury vrstvy mezi ESB a adaptérem

Jelikož jsou všechny tyto procedury implementovány velmi podobným způsobem a liší se pouze v detailech, nemá smysl se podrobně zabývat každou z nich. Obecné chování těchto procedur zahrnuje:

1. Perzistentní spuštění externí procedury implementující adaptér.
2. Výběr vhodné interní procedury adaptéru.
3. Zavolání interní procedury adaptéru.
4. Získání informací o chybách při zpracování.
5. Nastavení těla pro HTTP odpověď.
6. Nastavení HTTP stavového kódu.
7. Uvolnění perzistentně zavolané procedury.

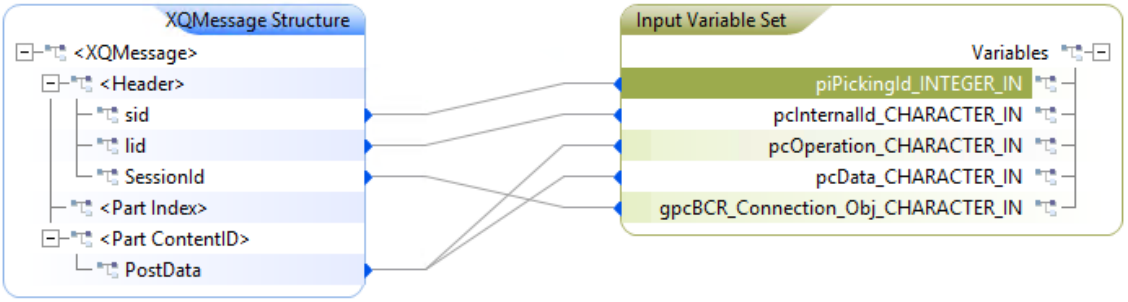
Pro další popis implementačních detailů těchto procedur si je rozdělíme na dvě skupiny podle principu, na kterém jsou postaveny, a jaké očekávají parametry:

1. Získané parametry jsou přímo předány konkrétní proceduře adaptéru.
2. Na základě těla HTTP požadavku se vybere procedura adaptéru, kterou je třeba spustit.

Do první skupiny patří procedury `y_nsug00.p`, `y_nsug01.p`, `y_nsug02.p` a `y_nsug04.p`. Jejich hlavní společnou charakteristikou je, že bez ohledu na to, jaké parametry obdrží od ESB procesu, vždy spustí jednu a tu samou proceduru adaptéru a hodnoty parametrů jí pouze přepošlou. Druhý bod uvedeného obecného chování se jich tedy netýká.

Nastavení těla odchozí zprávy, tj. výstupního parametru `opclResponse`, a stavového kódu po skončení činnosti interní procedury probíhá na základě toho, jestli došlo k chybě nebo ne. V případě chyby je do těla zprávy vloženo hlášení o chybách prostřednictvím funkce `clGetErrorList` a nastaven stavový kód `400 Bad Request`. Pokud k žádné chybě nedošlo, jsou prostřednictvím `opclResponse` vrácena požadovaná data (např. množina vychystávacích návrhů nebo návrh ovlivněný editací) získaná procedurou adaptéru `getSuggestionById` a nastaven stavový kód `200 OK`, případně `204 No Content` (viz tabulka 7.2).

Procedury druhé skupiny, tzn. `y_nsug03.p` a `y_nsul00.p` se liší v tom, že v době jejich volání není jasné, která interní procedura adaptéru jimi bude spuštěna. Ta je vybrána až v rámci druhého bodu obecného chování na základě vstupního parametru `pcOperation`, který je součástí XML v těle HTTP požadavku. Z něj je tento parametr vyseparován ještě před samotným voláním procedury prostřednictvím mapování, jak ukazuje obrázek 9.2.



Obrázek 9.2: Mapování polí XQMessage na parametry procedury `y_nsul00.p`. Pole `PostData` obsahující XML se mapuje na dva parametry, přičemž každému z nich je přiřazena hodnota jiného elementu, který je určen XPath transformací. Obrázek byl získán z prostředí Sonic Workbench.

Všimněme si, že v tomto případě mapujeme jedno pole příchozí `XQMessage` na dva parametry volané procedury. Co už však na tomto obrázku vidět není, je akce XPath trasformace aplikovaná na XML v poli `PostData`, jehož formát je dán schématem 9.1.

Odpovídající XPath výrazy pro získání požadovaných dat z tohoto XML budou tedy mít podobu:

- `request/operation/string(.)` – získání hodnoty elementu `<operation>` s názvem operace, která se má provést,
- `request/data/string(.)` – získání hodnoty elementu `<data>` s hodnotou parametru pro příslušnou proceduru adaptéru.

```
<xs:schema xmlns:xs = "http://www.w3.org/2001/XMLSchema">
  <xs:element name="request">
    <xs:complexType>
      <xs:sequence>
        <xs:element type="xs:string" name="operation"/>
        <xs:element type="xs:any" name="data"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Schéma 9.1: Schéma XML pro specifikaci operace prováděné nad řádkem vychystávacího návrhu a případné nové hodnoty ovlivněného datového pole.

U dalších bodů obecného chování procedur je jediný rozdíl oproti prvnímu typu v tom, že ještě navíc přibude stavový kód `501 Not Implemented`, pokud v požadavku přišla neznámá operace. Přehled možných operací a tedy i hodnot elementu `<operation>` procedury `y_nsul00.p` je uveden v tabulce 9.3. Procedura `y_nsug03.p` podporuje pouze operaci `stageOrder` (viz sekce 7.1).

Operace	Význam
delete	Smazání řádku vychystávacího návrhu.
updateAdoptionCode	Změna příznaku dodávky.
updatePickedQty	Změna hodnoty odebraného množství.
updatePickingLocation	Změna vychystávacího místa.
updatePickingStorage	Změna vychystávacího skladu.
updatePickStatus	Změna stavu vychystávání.

Tabulka 9.3: Přehled operací podporovaných procedurou `y_nsul00.p`

Přestože REST API nahlíží na editaci a smazání řádku vychystávacího návrhu jako na dvě odlišné operace, obě jsou implementovány procedurou `y_nsul00.p`. V případě mazání záznamu je element `<data>` jednoduše ignorován.

9.2 Testovací klient

Součástí zadání této práce bylo také vytvořit jednoduchého testovacího klienta, jehož prostřednictvím budeme demonstrovat funkčnost celého implementovaného systému pro externí přístup. Tento klient by měl být schopen realizovat všechny operace nad vychystávacími návrhy uvedené v sekci 7.1 s využtím nabízeného REST API.

Jako implementační jazyk klienta byl zvolen jazyk PHP, a to především kvůli možnosti snadné realizace požadovaného chování. Jeho uživatelské rozhraní je vytvořeno s využitím HTML5 formulářů a celou klientskou aplikaci je tedy možné spouštět přes webový prohlížeč. Požadavky na REST aplikační rozhraní jsou realizovány prostřednictvím HTTP klienta pro PHP *Guzzle*.[1]

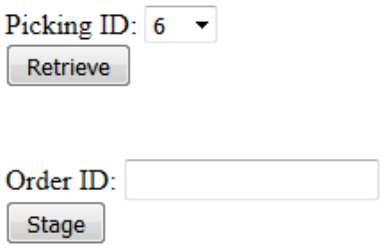
[1] Pro více informací viz http://docs.guzzlephp.org.

Testovací klient se bude skládat z celkem tří zdrojových souborů s následujícími významy:

- `index.php` – domovská stránka klienta umožňující výběr a vytvoření vychystávacího návrhu,
- `picking.php` – logika pro veškerou manipulaci s vybraným návrhem a jeho prezentaci,
- `utils_rest.php` – knihovna funkcí zprostředkovávajících komunikaci s REST API.

9.2.1 Podoba klienta a jeho funkce

Domovská stránka, která se zobrazí jako první po spuštění klientské aplikace, umožňuje uživateli vybrat jeden z vychystávacích návrhů dostupných v systému proALPHA a zobrazit jeho editovatelný detail, stejně jako zadání identifikátoru zakázky, která má být vychystána. Její implementaci realizuje soubor `index.php`. Jak ukazuje obrázek 9.3, na stránce se nacházejí dva formuláře, jeden pro každou z uvedených operací.



Obrázek 9.3: Podoba domovské stránky testovacího klienta.

Jelikož získání všech zakázek dostupných k vychystání je mimo rozsah implementovaného REST API, je nutné identifikátor zakázky opsat ze systému ručně.

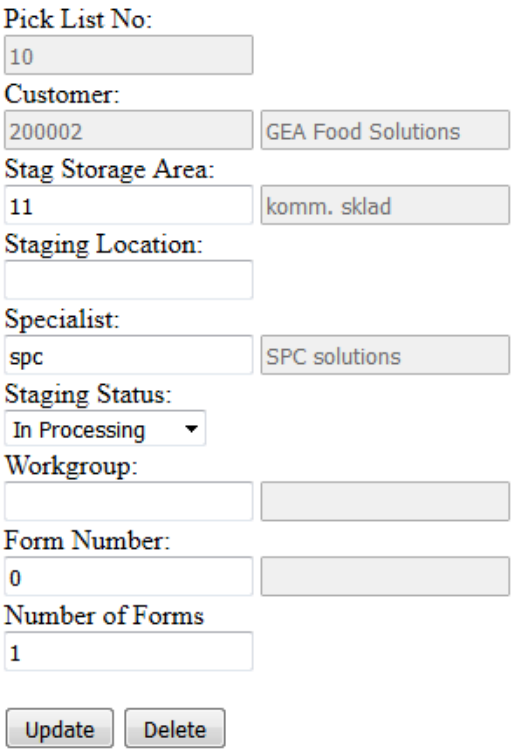
Po vybrání, případně vytvoření jednoho z návrhů, je uživatel přesměrován na stránku `picking.php`, kde je dostupná převážná většina funkcionality testovacího klienta. Tato stránka zobrazuje jeden vychystávací návrh obdobným způsobem, jako je tomu v případě hlavního a závislého okna v systému proALPHA.

Na obrázku 9.4 můžeme vidět formulář, zobrazující editovatelné údaje z hlavního okna návrhu. Kromě těchto polí jsou pro přehlednost přidány také needitovatelná pole s identifikátorem návrhu a informacemi o zákazníkovi, který je s ním spojen. Dále si můžeme všimnout ještě polí s dodatečnými informacemi o editovatelné hodnotě jako například název komisionářského skladu s identifikátorem `11`.

Po odeslání formuláře tlačítkem `Update` se všechny aktuální hodnoty editovatelných polí zapíšou do databáze systému proALPHA a zobrazí se nová podoba vychystávacího návrhu. Pokud se uživatel rozhodne návrh smazat tlačítkem `Delete`, bude v případě úspěchu vrácen na domovskou stránku klienta a vychystávací návrh bude trvale odebrán z databáze.

Závislé okno je reprezentováno tabulkou pod tímto formulářem, jejíž zjednodušená podoba je v příloze C.1. Hlavní řádky mají vždy v prvních dvou sloupcích zobrazeno číslo řádku a název produktu, kterého se řádek týká. Všechny řádky s prázdnými prvními dvěma sloupci jsou závislé.

Pro editaci jednotlivých datových polí je v odpovídající buňce vždy k dispozici zadávací pole nebo rozbalovací nabídka a tlačítko `Update`, které se vztahuje pouze k této buňce. Změněná hodnota se zapíše do databáze až po stisknutí tohoto tlačítka. Zároveň také dojde



Obrázek 9.4: Formulář umožňující editaci a smazání vychystávacího návrhu

k aktualizaci zobrazovaného vychystávacího návrhu a všechny neodeslané změny tak budou přepsány. Smazání řádku je vždy řešeno tlačítkem `Delete` v jeho posledním sloupci.

Použité zobrazení položek se může jevit jako nepraktické a těžkopádné, přičemž některé operace nemají smysl, jelikož jejich provedení povede vždy k chybě (např. `Delete` u závislého řádku). Na druhou stranu ale poskytuje potřebnou flexibilitu a volnost pro plnou demonstraci funkčnosti implementovaného systému.

9.2.2 Vnitřní realizace klienta

V předchozí části jsme probrali, jak má aplikace testovacího klienta vypadat z uživatelského pohledu a jaké od ní očekáváme chování, nyní se budeme zabývat tím, jak bylo této funkcionality dosaženo.

Domovská stránka, tj. skript `index.php` je na realizaci velice jednoduchá a obsahuje pouze dva formuláře, jeden pro výběr vychystávacího návrhu a druhý pro vychystání zadané zakázky. Oba formuláře jsou definovány stejným způsobem:

```
<form method="get" action="picking.php">.
```

Po jejich odeslání je tedy uživatel přesměrován na stránku `picking.php`. Té je metodou `GET` předána vždy jedna z následujících dvojic parametrů:

- `submit=Retrieve` a `MMT_Picking_ID` s identifikátorem vybraného vychystávacího návrhu v případě zobrazení,
- `submit=Stage` a `orderId` s identifikátorem zakázky v případě vychystávání.

Komunikaci klienta s aplikačním rozhraním systému zprostředkovávají funkce definované v samostatném souboru `utils_rest.php`. Všechny tyto funkce jsou implementovány podle stejného vzorce:

1. vytvoření instance *Guzzle* klienta,
2. nastavení těla HTTP požadavku,
3. zaslání požadavku na REST API,
4. získání stavového kódu a těla obdržené odpovědi,
5. v případě chyby (stavový kód větší než 300) zobrazení vhodné chybové stránky,
6. v opačném případě vytvoření objektu `SimpleXMLElement` z XML obdrženého v těle odpovědi reprezentující aktuální podobu daného návrhu
7. vrácení tohoto objektu volající funkci.

Součástí `utils_rest.php` je také funkce `displayErrorPage` pro zobrazení zmiňované chybové stránky. Ta obsahuje kromě HTTP stavového kódu odpovědi také obdržený seznam chyb. Pro převod XML reprezentace tohoto seznamu do podoby HTML je k dispozici funkce `errorListToHtml`.

Samotný skript `picking.php` pak implementuje především generování a obsluhu všech formulářů umožňujících editaci vybraných datových polí. Tato obsluha je implementována v jeho úvodní části a obsahuje následující větve kódu:

- **Delete staging suggestion** – smaže vychystávací návrh a vrátí uživatele na domovskou stránku.
- **Stage order** – zařídí vychystání zakázky zadané na domovské stránce.
- **Update staging suggestion** – vygeneruje XML pro úpravu vychystávacího návrhu (viz sekce 8.5) a zajistí její provedení.
- **Update suggestion line** – vygeneruje XML pro úpravu řádku vychystávacího návrhu (viz 9.1.2) a zajistí její provedení.
- **Delete suggestion line** – smaže řádek vychystávacího návrhu.
- **Get staging suggestion** – získá vychystávací návrh se zadaným identifikátorem.

Každá z těchto větví zahrnuje přístup k REST API přes některou z funkcí `utils_rest.php` a po jejím provedení je vždy k dispozici objekt `SimpleXMLElement` s aktuální podobou vychystávacího návrhu daného hodnotou `MMT_Picking_ID`, kterou skript obdržel metodou `GET`. Jedinou výjimkou je větev mazající záznam.

Formulář zastupující hlavní okno systému proALPHA po potvrzení předává metodou `GET` parametry odpovídající jednotlivým editovatelným položkám, ke kterým přidává identifikátor zobrazovaného návrhu. Názvy parametrů korespondují se elementy XML rozebíraného v sekci 8.5.

Tabulka pro zobrazení přehledu položek vychystávacího návrhu je generována příkazem `foreach` nad elementy `ttMMT_PickingDetailView` získaného z objektu `SimpleXMLElement`. Každá buňka reprezentující editovatelné pole je realizována jako samostatný formulář o jednom zadávacím prvku a jednom tlačítku. Při jeho odeslání se s parametry předávají také

hodnoty identifikující vychystávací návrh a jeho řádek a název odpovídající operace, která bude mít jednu z hodnot uváděných v tabulce 9.3 kromě hodnoty `delete`.

Pro podporu generování tabulky s položkami jsou v souboru `picking.php` k dispozici ještě následující funkce:

- `print_delete_form` – vygeneruje mazací formulář do poslední buňky řádku,
- `print_update_form` – vygeneruje formulář pro úpravu jednoho datového pole,
- `print_status_select` – vygeneruje rozbalovací nabídku pro stav vychystávání,
- `print_adoption_select` – vygeneruje rozbalovací nabídku pro příznak dodávky.

Jelikož REST API neimplementuje podporu pro získání stavových kódů a příznaků dodávky z číselníků systému proALPHA, jsou tyto hodnoty zapsány v generujících funkcích jako konstanty.

Kapitola 10

Závěr

V úvodu této práce jsme se zabývali problematikou ERP systémů, v rámci níž byly vysvětleny základní principy modulární architektury a integrace jednotlivých oblastí podniku. Dále jsme také probrali typické moduly, se kterými se můžeme v ERP systémech setkat, a jejich význam.

Tyto poznatky byly následně prakticky demonstrovány na informačním systému proALPHA, k jehož datům se v rámci této práce snažíme přistupovat přes externího klienta. Jako cílová oblast, ke které přistupujeme byla zvolena správa vychystávacích návrhů spadající do modulu materiálového hospodářství. Pro přístup do systému byl využit jeho modul INWB, který slouží jako primární nástroj pro tyto účely.

V následujících kapitolách pak byly podrobně představeny principy ESB a jeho transportní vrstvy EMS. Právě ty jsou totiž využity pro implementaci zmiňovaného modulu INWB. Dále jsme se zabývali také principy a relevantními datovými strukturami hlavního implementačního jazyka této práce OpenEdge ABL. S ohledem na tyto technologie jsme pak navrhli celkovou podobu systému pro externí přístup a následně jej úspěšně implementovali. Za účelem demonstrace jeho funkčnosti byl rovněž vytvořen i jednoduchý klient využívající služeb implementovaného systému.

Při práci se systémem proALPHA vyvstaly jisté problémy, které komplikovaly zpřístupnění jeho funkcionality externím klientům. Bylo by možné dosáhnout lepších výsledků, pokud by byla proALPHA lépe dokumentovaná a její zdrojové kódy více korespondovaly se zásadami moderního softwarového inženýrství, tj. pokud by například bylo vhodněji oddělené uživatelské rozhraní od funkcionality systému.

Vývojové nástroje třetích stran využívané systémem proALPHA jsou obvykle přizpůsobené podle jeho potřeb a jejich chování se může lišit od standardního. Tyto změny pak mohou způsobit komplikace při používání jejich standardní funkcionality.

Otázky, které v této práci řešeny nebyly, se týkají především zabezpečení celého implementovaného systému. Pro jeho praktické použití by v bylo nutné v prvé řadě implementovat podporu pro přihlašování uživatelů do systému přes vytvořené REST API a řešit problémy s tím spojené. Stejně tak by bylo vhodné se podrobněji zabývat situacemi, jež mohou nastat v Sonic ESB, a řešit chyby, k nimž dochází na úrovni serveru zprostředkovávajícího mezivrstvu mezi klienty a systémem proALPHA. Řešení tohoto problému by zahrnovalo implementaci vhodných ESB procesů ošetřujících vzniklé chyby a jejich propojení s implementovanými REST procesy.

Dále, vzhledem k nedostatečné dokumentaci zdrojových kódů, na jejichž základě byl implementován adaptér pro systém proALPHA, by také bylo vhodné podrobněji studovat tyto kódy za účelem získání hlubších poznatků o jejich celkovém významu a logice. S jejich

využitím by pak bylo možné provést verifikaci adaptéru a případně jej upravit tak, aby se eliminovalo co nejvíce scénářů, které by potenciálně mohly dostat systém do nekonzistentního stavu.

Modul INWB, který byl v práci použit pro komunikaci s externími klienty, je v systému proALPHA viditelně nový a je stále ve vývoji. Tento stav se ve zdrojových kódech odráží ve formě komplikovaných, relativně málo dokumentovaných a částečně zakryptovaných úseků kódu. I přes tyto problémy se však podařilo dosáhnout všech cílů vytyčených v zadání práce. Na základě výsledků dosažených v této práci se za současného stavu modulu INWB jeví navrhovaný způsob jako jedna z nejkomplikovanějších cest, jak přistupovat k datům systému proALPHA z externích klientů. Vzhledem k výše zmíněnému intenzivnímu vývoji tohoto modulu však lze předpokládat, že se situace v blízké budoucnosti změní.

Literatura

[1] ERP Solution for SMEs | proALPHA ERP. [Online; navštíveno 17. 12. 2016].
URL https://www.proalpha.com

[2] Evolution of ERP Systems. [Online; navštíveno 7. 11. 2016].
URL http://what-when-how.com/information-science-and-technology/evolution-of-erp-systems/

[3] *Sonic ESB: An Architecture and Lifecycle Definition*. 2005.

[4] Adam, R.; Kotze, P.; Van der Merwe, A.: Acceptance of enterprise resource planning systems by small manufacturing Enterprises. 2011.

[5] Arik Ragowsky, T. M. S.: Enterprise resource planning. *Journal of Management Information Systems*, ročník 19, č. 1, 2002: s. 11–15.

[6] Beal, V.: Enterprise messaging system. [Online; navštíveno 11. 11. 2016].
URL http://www.webopedia.com/TERM/E/enterprise_messaging_system.html

[7] Chappell, D.: *Enterprise service bus*. O'Reilly Media, Inc., 2004.

[8] Gála, L.; Pour, J.; Toman, P.: *Podniková informatika*. Grada Publishing as, 2006.

[9] Hamad, H.; Saad, M.; Abed, R.: Performance Evaluation of RESTful Web Services for Mobile Devices. *Int. Arab J. e-Technol.*, ročník 1, č. 3, 2010: s. 72–78.

[10] Hapner, M.; Burridge, R.; Sharma, R.; aj.: Java message service. *Sun Microsystems Inc., Santa Clara, CA*, 2002.

[11] Leon, A.: *Enterprise Resource Planning*. Tata McGraw-Hill Publishing Co. Ltd, 1999.

[12] Leon, A.: *ERP demystified*. Tata McGraw-Hill Education, 2008.

[13] Maler, E.; Paoli, J.; Sperberg-McQueen, C. M.; aj.: Extensible Markup Language (XML) 1.0 (Fifth Edition). Technická zpráva, W3C, Listopad 2008.
URL https://www.w3.org/TR/REC-xml/

[14] Malý, M.: REST: architektura pro webové API. [Online; navštíveno 18. 01. 2017].
URL https://www.zdrojak.cz/clanky/rest-architektura-pro-webove-api/

[15] Menge, F.: Enterprise service bus. In *Free and open source software conference*, ročník 2, 2007, s. 1–6.

[16] Monk, E.; Wagner, B.: *Concepts in enterprise resource planning*. Cengage Learning, 2012.

[17] Progress Software Corporation: *Getting Started with SonicMQ*. Srpen 2011.

[18] Progress Software Corporation: *Sonic ESB Configuration and Management Guide*. Srpen 2011.

[19] Progress Software Corporation: *Sonic ESB Developer's Guide*. Srpen 2011.

[20] Progress Software Corporation: *Sonic ESB Working with Built-in ESB Services*. Srpen 2011.

[21] Progress Software Corporation: *Sonic ESB Working with RESTful Web Services*. Srpen 2011.

[22] Progress Software Corporation: *SonicMQ Application Programming Guide*. Srpen 2011.

[23] Progress Software Corporation: *OpenEdge Development: Messaging and ESB*. Srpen 2014.

[24] Progress Software Corporation: *OpenEdge Development: Working with XML*. Srpen 2014.

[25] Progress Software Corporation: *OpenEdge Getting Started: ABL Essentials*. Srpen 2014.

[26] Progress Software Corporation: *OpenEdge Getting Started: Application and Integration Services*. Srpen 2014.

[27] Progress Software Corporation: *OpenEdge Getting Started: Guide for New Developers*. Srpen 2014.

[28] Richards, M.; Monson-Haefel, R.; Chappell, D. A.: *Java message service*. "O'Reilly Media, Inc.", 2009.

[29] Sadd, J.: *OpenEdge Development: ProDataSets*. Progress Software Corporation, Srpen 2014.

[30] Sharma, S.; Sengupta, A.: *Aurea Sonic SOA Overview*. Aurea Software Inc., 2013.

[31] Umble, E. J.; Haft, R. R.; Umble, M. M.: Enterprise resource planning: Implementation procedures and critical success factors. *European journal of operational research*, ročník 146, č. 2, 2003: s. 241–257.

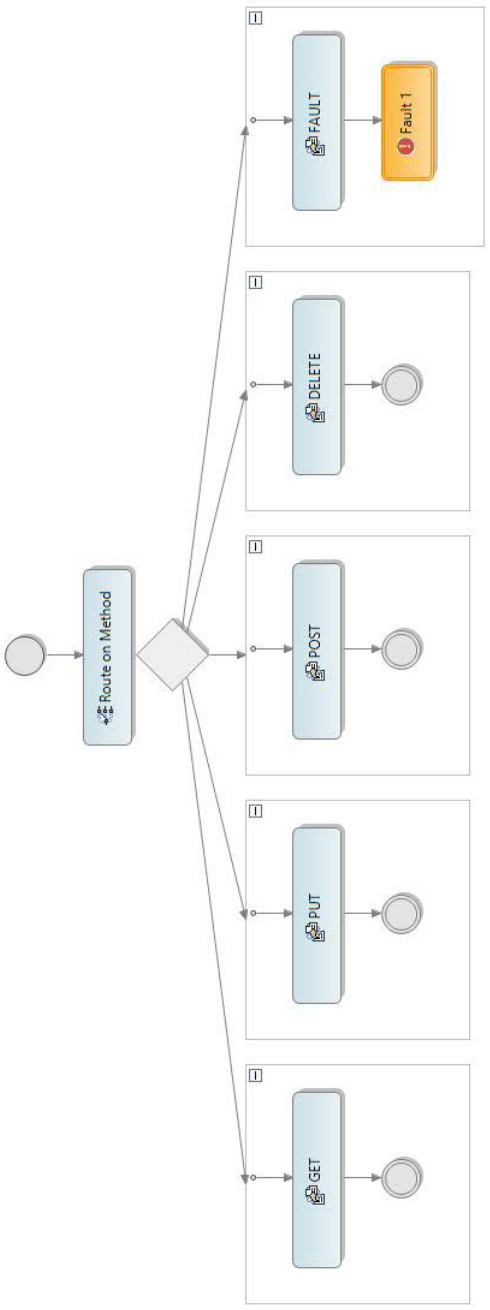
[32] Viñals, S.: Introducing OpenEdge Advanced Business Language (ABL). Technická zpráva, Progress Software Corporation, 2007.

[33] Wagh, K.; Thool, R.: A comparative study of soap vs rest web services provisioning techniques for mobile host. *Journal of Information Engineering and Applications*, ročník 2, č. 5, 2012: s. 12–16.

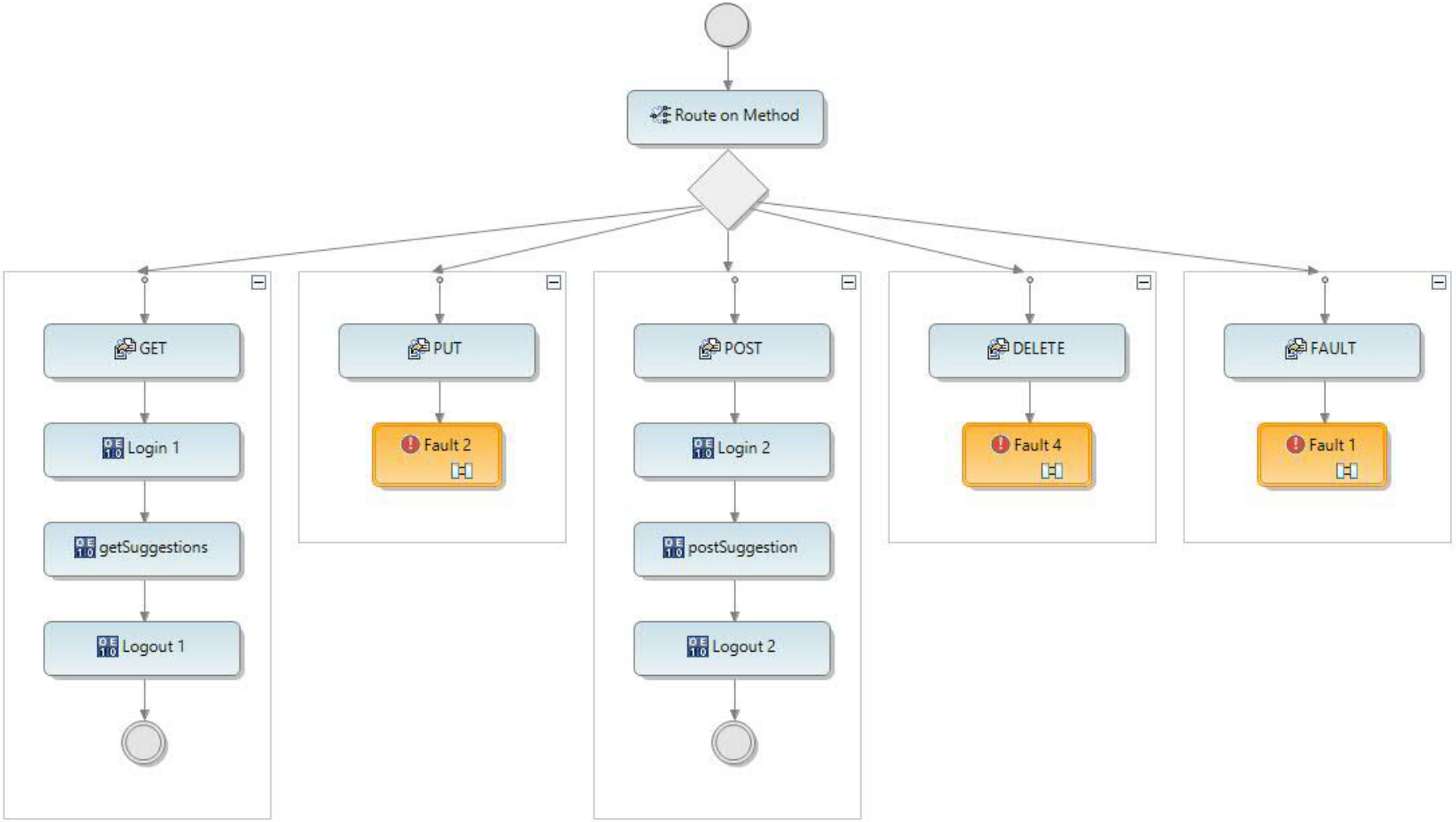
[34] Štumpf, J.: *Podniková sběrnice služeb*. Progress Software Corporation, 2006.

Příloha A

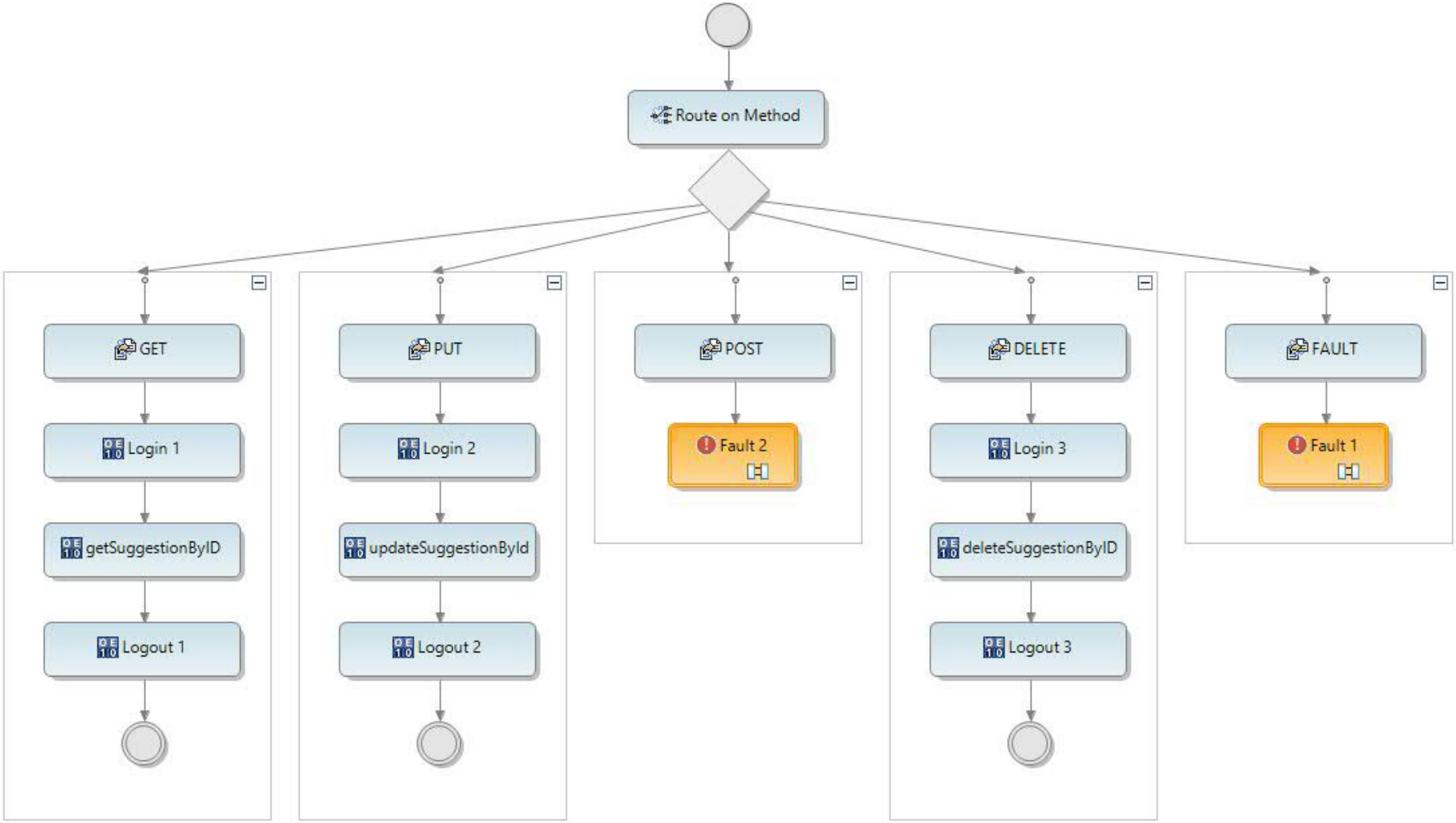
ESB procesy



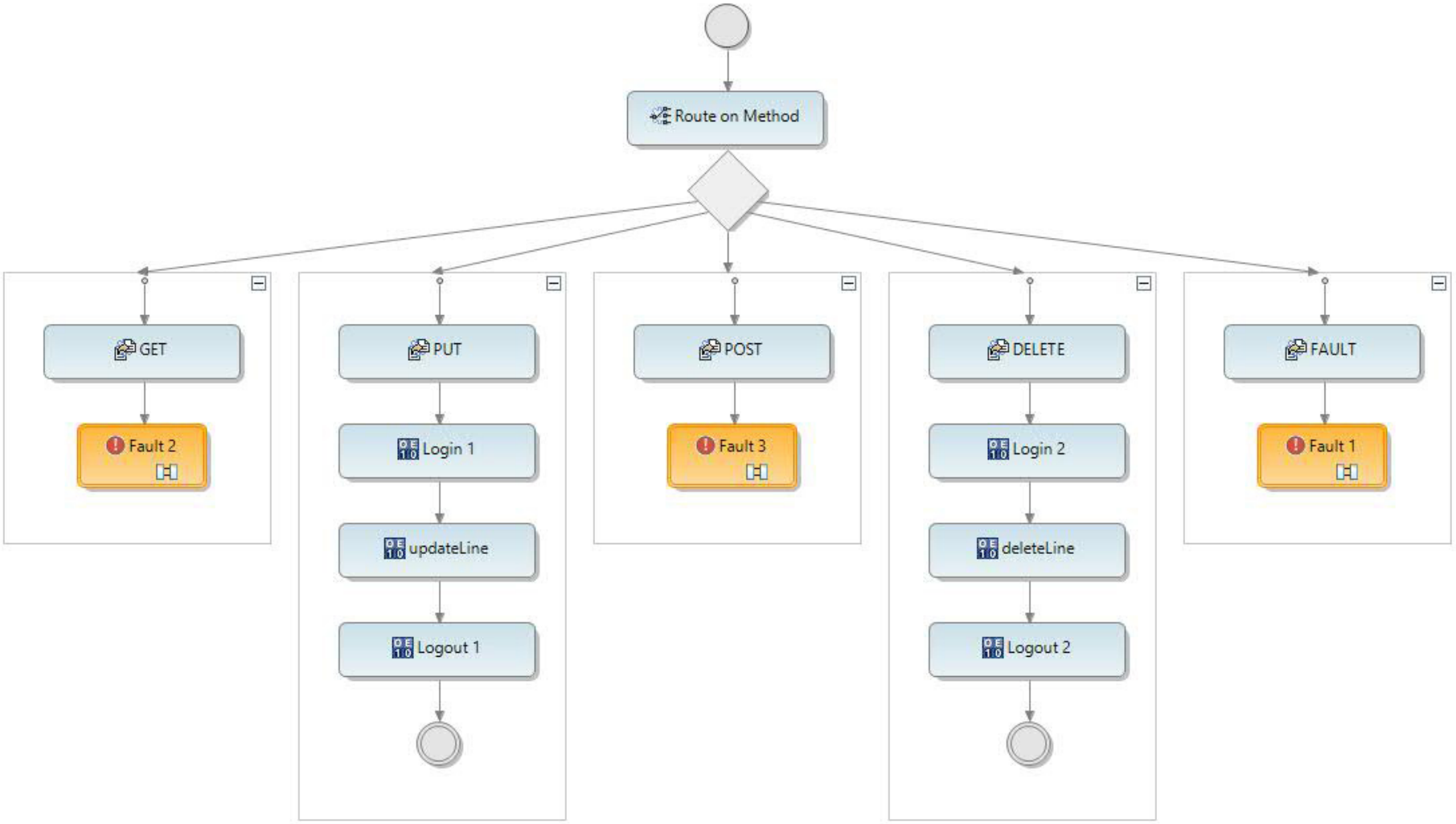
Obrázek A.1: Šablona ESB procesu implementujícího RESTful webovou službu. První krok procesu `Route on Method` typu *Směrování podle obsahu* určí na základě metody v HTTP požadavku, která větev procesu bude vykonána. Při neznámé metodě skončí proces chybovým krokem `Fault`. Obrázek byl získán z prostředí Sonic Workbench.



Obrázek A.2: ESB proces `suggestions` pro získání množiny vychystávacích návrhů a vychystání zakázky. Obrázek byl získán z prostředí Sonic Workbench.



Obrázek A.3: ESB proces `suggestionID` pro získání, editaci a smazání jednoho vychystávacího návrhu. Obrázek byl získán z prostředí Sonic Workbench.



Obrázek A.4: ESB proces `linesID` pro editace a mazání řádků vychystávacích návrhů. Obrázek byl získán z prostředí Sonic Workbench.

Příloha B

Procedury pro editaci řádků návrhu

updateLinePickingLocation

Stejně jako v případě editace samotného vychystávacího návrhu, i zde lze změnu vychystávacího místa považovat za nejjednodušší operaci. Všechen kód, který se stará o její realizaci se nachází v odpovídajícím triggeru.

Jeho prvním krokem je podmínka testující, zda došlo ke změně hodnoty `pickinglocation`, jejíž splnění můžeme považovat v adaptéru vzhledem k volání editační procedury za samozřejmé a tudíž není nutné se jí zabývat. Dále následuje volání `applyChangeToMates` a `setStatusOfPickLineToWorking`. Žádné kontroly validity zde nejsou nutné.

Ve zdrojovém kódu procedury `updatelinepickinglocation` si můžeme všimnout ošetření operace `saveChanges` blokem `CATCH`. Všechna tato volání musejí být takto ošetřena z důvodu uváděného v části 8.1.1.

updateLinePickingStorage

Stejným způsobem byla vytvořena i procedura `updateLinePickingStorage` měnící hodnotu komisionářského skladu. Jediný rozdíl oproti `updateLinePickingLocation` je v tom, že přibyla kontrola zadané hodnoty prostřednictvím funkce adaptéru `lCanUpdateStorage`.

Ta byla implmentována na základě interní procedury `checkUpdateOfField` v souboru `mmbpic01.w`, která kontroluje validitu prováděných změn. Jelikož tato procedura implementuje kontrolu všech polí, bylo v ní nutné identifikovat ty části, které se týkají pole `PickingStorage`. Konkrétně se jedná o podmínku na samém začátku procedury a příslušnou větev příkazu `CASE`.

Tyto části bylo možné v `lCanUpdateStorage` snadno rekonstruovat pomocí dříve uváděných kroků pro přerušení vazby na uživatelské rozhraní. Celá funkce je pak ošetřena blokem `CATCH` kvůli možnosti chyby při volání metody `checkPickingStorage` třídy `MMCPickingSvcStd.cls`.

updateLineAdpotionCode

Procedura `updateLineAdpotionCode` je implementovaná stejným způsobem jako dříve uvedená `updateLinePickingStorage` pouze s tím rozdílem, že je zde po změně hodnoty nutné

řešit závislost dalších datových polí. Dále je třeba také ještě před zahájením obecných akcí uvedených v přehledu v úvodu sekce 8.6 řešit, zda je zadaný příznak dodávky vůbec platný, neboť proALPHA v tomto případě opět spoléhá, že uživatelské rozhraní nedovolí chybné zadání. Možné hodnoty a jejich význam uvádí tabulka B.1.

Hodnota pole	**Řetězcová reprezentace**
`B`	*Objednat dle potřeby*
`K`	*Kompletní dodávka*
`M`	*Vyšší dodávka*
`N`	*Následná dodávka*
`R`	*Zbytek převzít*
`T`	*Dílčí dodávka*
`U`	*Snížená dodávka*

Tabulka B.1: Možné hodnoty pole `AdoptionCode` a jejich význam.

Při implementaci kontroly validity nové hodnoty bylo, stejně jako v případě hodnoty `PickStatus` v části 8.5.1, nutné dohledat, kde dojde k výjimce při chybném zadání. I v tomto případě je generována chyba v metodě `cDispValueByRefParamVal`, která tudíž byla použita pro implementaci funkce adaptéru `cGetAdoptionCode`. Ta podle zadaného identifikátoru vyhledá a vrátí řetězcovou reprezentaci pro daný příznak dodávky. Při zadání neplatného identifikátoru je vrácena nedefinovaná hodnota, kterou jazyk ABL reprezentuje otazníkem `"?"`.

Kontrola, zda lze na danou položku aplikovat zadaný příznak dodávky, probíhá v rámci funkce `lCanUpdateAdoptCode`. Ta opět vychází z větve procedury `checkUpdateOfField`, která se stará o hodnotu `AdoptionCode`. Tato větev obsahuje pouze volání procedury `checkAdoptCodeAndDocType`, jejíž definice se nachází v `mmrpic00.p`. Díky tomu, že tato externí procedura musí být nastavena kvůli práci s datasetem jako nadřazená procedura (super procedura) adaptéru, je `checkAdoptCodeAndDocType` možné volat jednoduše příkazem `RUN`. Její volání bylo převzato pouze s odstraněním vazeb na pole uživatelského rozhraní a ošetřením blokem `CATCH`.

Změnou `AdoptionCode` může být v některých případech ovlivněna i hodnota `PickStatus`, a to konkrétně při zadání, hodnot `B`, `N` nebo prázdného příznaku. Logika zajišťující tuto změnu se nachází v příslušném triggeru. Při její rekonstrukci v adaptéru v rámci akcí následujících nastavení nové hodnoty bylo kromě obvyklých úprav nutné potlačit dialog žádající potvrzení změny, u něhož předpokládáme implicitní kladnou odpověď a tudíž jej můžeme vypustit.

updateLinePickedQty

Implementace procedury `updateLinePickedQty` je ve srovnání s dříve popsanými procedurami podstatně komplikovanější, a to především kvůli větším dopadům na ostatní datová pole a zbytek systému. Některé bloky kódu realizující změnu této hodnoty se nepodařilo plně objasnit a byly tudíž převzaty jen s nejnutnějšími úpravami, aby mohly fungovat v kontextu adaptéru. Kromě odpovídajícího triggeru sloužila jako zdroj kódu také procedura `invoke-`

`SaveChanges`, která se volá vždy v rámci `local-assign-record`, tj. při potvrzení úprav v systému proALPHA.

Jako v předchozích případech, i zde se na začátku nachází kontrola implementovaná funkcí `lCanUpdatePickedQty`, která vychází z příslušné větve `checkUpdateOfField`. Ta se stará o to, aby nedošlo k zadání odebraného množství do pole hlavního nebo již vychystaného řádku. Dále byla také přidána kontrola, zda nebylo zadáno záporné množství.

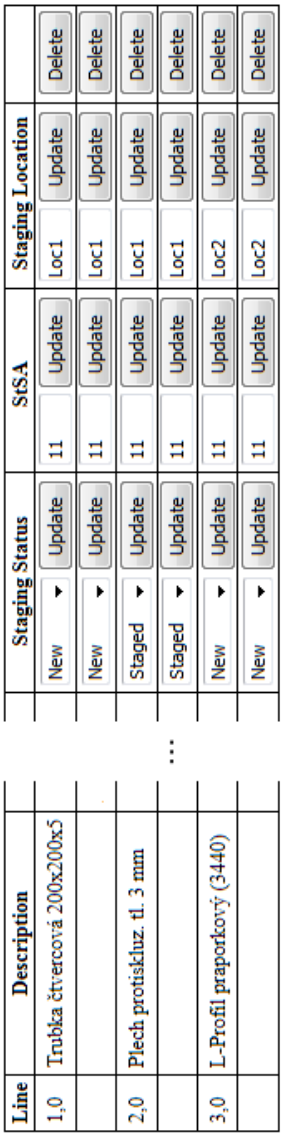
Po úvodní kontrole proveditelnosti operace následuje podmíněný blok, jehož význam se nepodařilo zjistit a byl tedy převzat bez větších změn. Tento blok je ve zdrojovém kódu adaptéru vhodně vyznačen a jeho zdrojem je kromě triggeru také procedura `invokeSaveChanges`, která je v původním zdroji explicitně volána.

Za tímto blokem následuje, kromě samotného nastavení nového množství, také výpočet rozdílu staré a nové hodnoty `PickedQty` a její promítnutí do nadřazených řádků voláním `updatePickedQty` z externí procedury `mmrpic00.p`.

Na konci `updateLinePickedQty` se pak nachází relevantní logika z `invokeSaveChanges` kontrolující například zda nedošlo k pokusu odebrat větší než rezervované množství. V této části rovněž proběhne samotné uložení změn. Volání procedury `AskUserForChangeOfReservation` bylo zcela ignorováno, jelikož vyžaduje přímou interakci s uživatelem. Změna rezervovaného množství navíc ani nespadá do vybraných operací, které adaptér podporuje.

Příloha C

Uživatelské rozhraní testovacího klienta



Obrázek C.1: Zjednodušená tabulka pro editaci a mazání řádků vychystávacího návrhu.

Příloha D

Obsah CD

- `client` – složka se zdrojovými kódy testovacího klienta.
  - `index.php` – vstupní bod klientské aplikace.
- `RESTstaging` – projekt exportovaný z prostředí Sonic Workbench.
  - `resources`
    - `esboe` – složka s invokačními soubory `.esboe`.
    - `processes` – složka s definicí ESB procesů implementujících REST API.
  - `SonicConnectService.scsvc` – instance služby Sonic Connect s definicí REST API.
- `src`
  - `y`
    - `proc` – složka se zdrojovými kódy adaptéru.

  Adresářová struktura koresponduje se strukturou v systému proALPHA, kde složka `src` obsahuje zdrojové kódy systému, `y` určuje úroveň customizace, na které se nacházejí, a `proc` označuje kódy reprezentující spustitelné procedury.
- `tex` – složka se zdrojovými soubory pro vygenerování technické zprávy.
  - `obrazky` – složka obsahující obrázky použité v technické zprávě.
- `Technická zpráva.pdf` – elektronická verze technické zprávy.