Vysoká škola ekonomická v Praze
Fakulta informatiky a statistiky


# Návrh modelové aplikace pro řízení distribuce zboží pro supermarkety

BAKALÁŘSKÁ PRÁCE

Studijní program: Aplikovaná informatika
Studijní obor: Aplikovaná informatika

Autor: Vojtěch Pavlů
Vedoucí práce: Ing. Tomáš Bruckner, Ph.D.

Praha, Březen 2020

## Prohlášení

Prohlašuji, že jsem bakalářskou práci *Návrh modelové aplikace pro řízení distribuce zboží pro supermarkety* vypracoval samostatně za použití v práci uvedených pramenů a literatury.

V Praze dne 11. května 2020 .....

Podpis studenta

## Poděkování

Děkuji Ing. Tomáši Brucknerovi, Ph.D. za obrovskou pomoc, trpělivost, cenné rady a ochotu při vedení mé bakalářské práce. Dále bych chtěl poděkovat své rodině za podporu při jejím psaní. V neposlední řadě pak všem třem konzultantům z oboru maloobchodního prodeje, kteří si nepřáli být z důvodu profesní mlčenlivosti jakkoliv jmenováni, za jejich rady, poznámky a pomoc při snaze pochopit problematiku.

# Abstrakt

V této práci je popsán celý vývoj komunikačního nástroje pro správu výměny dat mezi supermarketem (či jiným obdobným prodejcem) a zákazníkem. Tento vývoj počíná od popisu odvětví a podniku v tomto odvětví, stanovením důležitých oblastí, které je třeba při tvorbě nástroje zpracovat, samotným vývojem a testováním vytvořených komponent.

Konkrétní analýza se sestává z marketingových nástrojů určených k mapování silných a slabých stránek podniku, faktorů působících na podnik v odvětví a základního zkoumání vzorců chování spotřebitele.

Formulace rámce systému je tvořena pomocí procesů v notaci EPC, z nichž je formulován návrh řešení včetně univerzální klient-server platformy pro umožnění prakticky libovolné komunikace v rámci podniku. Ta je tvořena v jazyce Java a nad ní je sestaven soubor nástrojů pro naplnění potřeb samotné komunikace mezi stanovenými subjekty, a to s podporou pro desktop. Při vývoji autor uvažoval i jistý druh zabezpečení a to především pomocí protokolu SSL/TLS pro předejití bezpečnostním trhlinám typu Man in the Middle. V rámci práce je také navrženo několik testů, jejichž úspěšné splnění lze považovat za naplnění cílů této práce.

### Klíčová slova

Supermarket, Java, Vývoj, Klient-server, SSL/TLS, Business strategie

# Abstract

This paper contains the whole customer-supermarket (or any other similar vendor) communication instrument development from analysis of the market and a general business in the market, main scope of the business to the code transformation and the created components testing.

The market analysis consists of the result of marketing tools for indication and definition of business' strengths and weaknessess, the business influencing factors and general consumer behavior description.

The process chains of the main business system scope are defined using EPC notation. These are used for versatile client-server platform assembly to provide arbitrary enterprise communication. The platform is created in programming language of Java (for desktop by default) and is used for building higher-level communication tools meant to be used by entities of the business. This thesis also contains a set of basic tests of which the successful goals completion of this paper indicates.

While developing, the author reflects needs of basic and general security systems and develops communication channels establishing concept using SSL/TLS protocol for the Man in the Middle type of attack prevention.

## Keywords

Supermarket, Java, Development, Client-Server, SSL/TLS, Business strategy

# Obsah

# Seznam obrázků

# Seznam výpisů kódu

# 1. Úvod

Práce pojednává o vývoji systému pro zefektivnění distribuce zboží mezi prodejcem (supermarketem) a spotřebitelem. S rozvojem informačních technologií je patrný silný důraz na použití výpočetní techniky s cílem dosažení vyššího zisku. Pomocí chytrých zařízení jsme schopni silně ovlivnit křivku učení v podniku a potažmo i v celém odvětví, maximalizovat úspory z rozsahu obecně a optimalizovat automatizaci v často opakovaných a snadno strukturovaných řešeních problémů. Tím se deleguje zodpovědnost za provedení akce na stroj, který zpravidla předepsaný úkon provede rychleji a s nižší mírou chybovosti.

## 1.1 O práci

Touto prací se snaží autor demonstrovat a především navrhnout modelové řešení problému, který spatřuje v oblasti nakupování zboží v kamenných prodejnách. Konkrétně pak mluvě o nákupních centrech, respektive o supermarketech. Autor spatřuje největší nedostatek tohoto způsobu distribuce zboží (myšleno při běžném nakupování) v časové neefektivitě a neefektivitě využití lukrativních (a tedy jistě velmi nákladných) prostor vyhrazených pro prodejnu. Protiargumentem by mohla být jistě podpora marketingu a propagace, na což lze reagovat argumentem užití e-shopu, který svou úlohu v rámci podpory prodeje naplňuje přinejmenším uspokojivě.

Touto genezí úvah lze najít východisko v kombinaci obojího s cílem využít výhod obojího a minimalizovat negativní externality v tomto vztahu vzniklé.

## 1.2 Cíle práce

Cílem práce je vyvinout takové řešení, jenž by splňovalo dostatečně podmínku navýšení efektivity využívání zdrojů, které prodejce má, ovšem není schopen je plně zúročit. Konkrétně se autor prací snaží popsat celý proces vývoje konceptu souboru softwarových komponent, které jsou schopny se starat o vztah mezi zákazníkem a obchodním řetězcem, respektive jeho pobočkou. K tomu autor vyžaduje pochopení business logiky a role, kterou podnik v odvětví prodeje zboží zastává. Tento soubor komponent je třeba vytvořit co možná nejuniverzálněji, aby bylo možné snadno provádět úpravy nad defaultní business logikou.

Jinými slovy má práce za cíl analyzovat odvětví a podnik typu supermarket, popsat vztah mezi zákazníkem a prodejcem, navrhnout základní platformu určenou pro tento vztah a tuto posléze testovat.

Je mimo jiné důležité apelovat na modení trendy a zvyklosti[1]

Tyto cíle lze strukturovaně a konkrétněji vydefinovat následovně:

1. **Pochopit odvětví** - Pro vytvoření spojnice (na jakékoliv úrovni a za použití jakýchkoliv nástrojů) mezi zákazníkem a prodejcem je vhodné pochopit náplň komunikace, cíle, za jakými je komunikace vedena a případně hodnotu jejích dílčích částí. Jedním z cílů práce je tedy analyzovat vztah mezi prodejcem a spotřebitelem pro potřeby návrhu zefektivnění tohoto vztahu pomocí výpočetní techniky. Výstupem tohoto dílčího cíle by měl být strukturovaný výstup zjištěných poznatků.
2. **Vytvoření základní platformy pro komunikaci** - V návaznosti na zjištěné nedostatky stávající komunikace by autor měl navrhnout platformu umožňující jistou formu kontaktu mezi zákazníkem a prodejcem tak, aby byl efektivnější, výnosnější, snazší, pohodlnější a/nebo rychlejší, tedy dojít znatelného zlepšení či alespoň navrhnout a připravit komponenty k dosažení tohoto zlepšení.
3. **Návrh konceptu nad platformou** - Autor zastává stanovisko, že tímto postupem vrstvení lze dosáhnout delšího životního cyklu konceptu a lze ho snadněji editovat.
4. **Test komplexu** - Testování navržených funkcionalit, zda naplňují správnost i business logiku

## 1.3 Výstup práce

Výstupem této práce je analýza podniku, ze které autor při vyhotovování dalších cílů vycházel. Dalším výstupem je funkční platforma umožňující komunikaci mezi spotřebitelem a prodejcem o libovolném obsahu a nad ní postaven systém v rámci komunikace mezi zákazníkem a prodejcem.

- **Analýza podniku a podniku v odvětví** - Analytické zaměření se na podnik pro potřeby vývoje platformy.
- **Platforma pro komunikaci** - Funkční platforma umožňující komunikaci o libovolném obsahu. Jinými slovy takový nástroj, který obsahuje abstraktní rozhraní pro definici zpráv, nehledě na jejich struktuře.[2] Tato platforma by dále měla být schopna tyto zprávy přenášet v rámci systému (pokud možno bezpečně) a řešit standardní typy chyb a problémů.
- **Komplex funkcionalit rozšiřujících platformu pro použití v supermarketu či jiném obdobném podniku.**

[1] *Best practices* - dobré návyky v oblasti návrhu a vývoje

[2] Analogickým příkladem z reálného světa bychom si mohli představit jako kombinaci řečového ústrojí a sluchu. Samotná slova, věty či jiné formulace myšlenek chápejme jako business logiku, nástroje umožňující přenos (sluchové a řečové ústrojí) pak jako platformu a konečně zprávy (pomocí kterých komunikujeme v rámci business logiky) jako zvuky, které vydáváme či slyšíme a ze kterých děláme nějaké výstupy a závěry.

## 1.4 Úvodní ustanovení

V této části autor definuje a popisuje ustanovení, jenž se práce přímo týkají. Lze zde nalézt například použitá sada pojmů či užitý software.

### 1.4.1 Terminologické deviace

V práci se autor občas musí odchýlit od standardního chápání, respektive pojmenování, různých entit, procesů či objektů. Koná takto za účelem upřesnění vyjádření a s cílem vhodnějšího vymezení pojmů, k čemuž ostatně slouží i tato podkapitola.

V následujících kapitolách je užíváno několika pojmů, které se mohou zdát v porovnání s ostatními, jako synonyma, či že nejsou použity zcela šťastně. Proto autor zde vypsal přesný použitý význam pro tyto termíny s nutným odůvodněním, proč takto usoudil.

#### Architektura, Struktura, Topologie

Tyto termíny lze do jisté míry považovat za podobné, ne-li shodné. Důležité je však uvážit, že je nutné tyto použít v mnoha kontextech a to z mnoha různých úhlů pohledů.

**Architektura** je pojem používaný především ve stavitelství, kdy se toto čistě materiální pojetí výseče reality snoubí s i uměleckým náhledem na budovu. Lze ho ovšem přirozeně použít (mimo jiné) i v mnoha oblastech informatiky pro popis vazeb mezi jednotlivými abstraktními prvky.

V této práci je používáno tohoto pojmu ve významu popisu vazeb mezi komponentami v rámci kódu a popisu návaznosti jednotlivých virtuálních entit.

**Struktura** lze chápat jako abstraktní model systému[3], který lze nějakým způsobem škálovat, kategorizovat, třídit.

V práci je tohoto termínu užito za účelem pojmenování **business vazeb** mezi **business entitami**, tedy chápání celku z pohledu reálného nahlížení na podnik, jako na ekonomický subjekt, jako subjekt se sociální vazbou, jako na právnickou osobu, atd.

**Topologie** je v práci pojmenován model, kde chápeme jednotlivá informační stanoviště jako subjekty a elementy v elektronické, virtuální síti. Přijměme proto podřízených pojmů z teorie grafů a nazývejme tyto uzly sítě.

[3] libovolného, nikoliv nutně z oblastí informatiky

Tento termín je velmi často užíván pro vědy zkoumající morfologii, tedy tvar zkoumaných objektů. Můžeme se s tímto setkat například v oborech matematiky.

### Třída, Objekt, Instance

**Třída** je předpisem (kódem), dle kterého jsou interpretovány objekty v souladu s touto třídou tvořeny. Sestává se z parametrů, jenž jsou součástí tvořených objektů, statických parametrů (náležící třídě), instančních metod (funkcí nad objektem), konstruktorů a statických metod (funkcí nad třídou, respektive statickými parametry). Mluvíme-li tedy o třídě, chápeme tak **typ** či **druh** instance[1, strana 57][2, strana 207-211].

**Objekt** v práci chápeme jako samotnou interpretaci nějakého definovaného druhu, tedy popsaného v nějaké konkrétní třídě.[2, strana 206]

**Instance** vystupuje v práci jako již konkrétní, vytvořený objekt, s nímž pracujeme.[4]

### Rozhraní, Interface

Rozhraní a Interface v práci chápejme jako propojovací uzel mezi implementacemi. Když tvrdíme, že dvě entity mají stejné rozhraní, chápejme toto tvrzení jako že mají stejné funkce, stejnou definici, shodné signatury a obdobné kontrakty. Pokud budeme tvrdit, že mají stejný interface, chápejme to jako že existují konkrétní, definovaná rozhraní.[5]

### Server, Služba

Chápejme rozdíl mezi **Serverem** a **Službou** v této práci tak, jak je v této podkapitole uvedeno.

Server chápejme jako softwarové vybavení či jako instanci obsahující nástroje pro obsluhu služeb a samotné služby. Oproti tomu službu chápejme jako samotný nástroj plnící business logiku a je uložena a náleží danému serveru. Server tedy chápejme jako kontejner služeb.

V této práci se autor uchýlil k tomuto netradičnímu pojetí, neboť usnadňuje pochopení struktury komunikace mezi jednotlivými prvky systému.

[4] Tato definice se odlišuje od obecně chápané definice instance.[2, strana 207]

[5] Obvykle je chápáno rozhraní a interface jako pojmy se shodným významem, jen s rozdílnou etymologií. My se však přikloňme k definici rozdílu, kdy každá třída má vlastní druh rozhraní a vedle toho může mít volitelně i rozhraní v podobě interface.[2, strana 215] Zároveň se obvykle obecně chápe, že rozhraní (tedy interface) jen přislibuje schopnost funkcionality, samotná implementace je odpovědná za její provedení.[2, strana 213]

## 1.5 Psaní práce

Autor se rozhodl tuto práci psát v typografickém jazyce TEX[3], k čemuž použil zdrojových kódů vyvěšených na intranetu katedry IT. TEX je volně dostupný typografický systém se zaměřením na elektronickou sazbu.[3] Ač byl vytvořen již v 70. letech, dodnes je z mnoha úhlů pohledu náročné najít rovnocenného konkurenta.[4]

# 2. Metoda

V rámci vývoje systému se autor rozhodl rozdělit celou práci do několika dílčích úkolů, jejichž zkoumání a plnění je věnováno několik dalších kapitol.

Celý systém lze jen těžko pochopit samostatně a bez použití nástrojů či alespoň nějakým způsobem standardizovaného či pevně stanoveného myšlenkového procesu. Za jeden z těchto jednoduchých procesů lze uvažovat například stanovení si co možná nejpřesnějších úkolů a cílů, kterých je třeba dosáhnout.

Vhodným nástrojem by v případě rozsáhlé týmové práce byly například metody **CPM**[1] nebo **PERT**[2], tedy metody ke zkoumání a řízení naplňování cílů. Třetí významnou metodou pro grafické znázornění jednotlivých činností je **Ganttův diagram**[3] Ovšem pro potřeby práce autor usoudil, že není nutné použít ani jednu z těchto jmenovaných metod - místo toho si stanovil jasné cíle pro potřeby sledování naplnění závazků k této práci.

## 2.1 Stanovení hlavních pilířů práce

Hlavním smyslem celé této práce a celého autorova snažení je popis struktury informačního systému pro tranzit informací v rámci odvětví maloobchodního řetězce, nebo lépe řečeno **supermarketu**. Tento projekt by měl naplňovat aktuální potřeby rychle se vyvíjejích trendů v oblasti obsluhy spotřebitele.

Pro práci si tedy autor stanovil následující opěrné pilíře, které tvoří jeho kodex pro implementaci řešení:

- **Znovupoužitelnost a univerzalita** - Zajistit tvorbu systému s co nejširším polem působnosti. Každý podnik má v rámci odvětví mnoho důvodů pro to se odlišit, každý má vlastní přístup a postup k plnění svých plánů, výsledný produkt by ho tedy neměl nijak omezovat či nutit tyto zvykolsti měnit. Toto ustanovení je úzce provázáno s přístupem **TASW**[4]. Autor také považuje za vhodné usnadnit budoucí vývoj do co nejvyšší možné míry, aby tak zajistil co nejdelší životní cyklus systému. Cestou k tomuto cíli je silná aplikace známých a prověřených návrhových vzorů, silná modularita, variabilita a co nejjednodušší provázanost a vzájemná závislost mezi komponentami. Teoreticky lze

[1] *„Critical Path Method“*, což lze přeložit jako *„Metoda kritické cesty“* - metoda hledající kritické cesty (nejdůležitější milníky v rámci vývoje projektu) s cílem plánování a sledování; silný důraz na teorii grafů

[2] *„Program evaluation and review technique“*, což lze přeložit jako *„Technika programového ohodnocení a sledování“* - metoda obdobného cíle, jako CPM, s výhodou přidání pravděpodobnostního ohodnocení v rámci normálního rozdělení s potenciálem získat představu o budoucím vývoji řízeného projektu a to co do splnění cílů ve stanoveném časovém horizontu

[3] Systém grafické notace pro plánování a řízení projektů či programů pomocí naplánované posloupnosti procesů.[5]

[4] *„Typový aplikační software“*, tedy celý balík víceméně univerzálních funkcionalit[6, kapitola 5]

poupravit původní význam zkratky **DRY**[5], s cílem naplnění výše uvedených cílů tak, aby byla zajištěno co nejširší a nejkvalifikovanější uplatnění.

- **Bez speciálních znalostí** - Udržovat při vývoji na mysli, že osoba, která bude zajišťovat integraci a deployment, nemusí být nutně obeznámena s používáním speciálních nástrojů a „berliček“. Řešení by mělo být tedy co nejjednodušší na používání, správu a nemělo by využívat složité frameworky, mnoho externích knihoven či nestandardních způsobů použití. Zároveň je klíčem k úspěchu v očích tohoto pilíře podrobná dokumentace. Nespornou výhodou pak je i fakt, že minimalizací používání mnoha knihoven se potenciálně sníží náklady na používání a na provoz, neboť hotové knihovny mohou býti zpoplatněny, a to fixně (tedy na nasazení) či variabilně (cena na měsíc, cena na procesor, kombinace předchozích, či jinak).

- **Bezpečnost** - Veškerý kód by měl být co nejvíce bezpečný. K tomu by měl využívat základních zásad bezpečného systému:

  - **Důkazovost** - bezustavné zaznamenávání akcí a stavů za účelem nepopiratelnosti
  - **Důvěrnost** - ochrana dat, uživatelů a systému jako takového před útočníky, tedy zachování tajemství
  - **Integrita** - zajištění, aby implementátor měl možnost dosažení celistvosti jak systému, tak i dat

  Zároveň by však neměl svazovat implementátora v rámci jeho snažení, neboť by tím byl porušen předchozí pilíř.

## 2.2 Analýza úkolů

Uvažujeme-li vytvářet systém pro podnik z určitého odvětví, je nutné si nejdříve zmapovat pole působnosti. Zjistit hlavní cíle v odvětví, způsob práce, pochopit úlohu jednotlivých aktérů z vnitřku a z vnějšku na podnik působících, identifikovat a popsat události a procesy, které na ně navazují. Zkráceně lze popsat následující část jako pochopení prostředí.

V druhém bodě je vhodné začít mluvit o systému a začít uvažovat o jeho návrhu, tzn. identifikovat a stanovit možná řešení a způsoby řešení a vybrat nejvhodnější alternativy.

Dále musíme vytvořit návrh - vymodelovat podnik jako množinu abstraktních entit v zájemných vztazích, teprve poté lze začít vytvářet systém. Neopomenutelnou součástí je průběžné testování, kdy procházíme nejen samostatné kusy, ale co možná nejobsáhlejší tématicky provázané komplexy z důvodů zajištění bezchybného chodu.

[5] *„Don't repeat yourself!“*, což lze přeložit jako *„Neopakuj se!“* - tzv. suchý princip[7]

## 2.3 Detailní rozpis úkolů

V následujícím bloku je vypsán celkový rozpis úkolů včetně jejich dílčích částí. Jsou zde zahrnuty podstatné milníky a je jim nadále přizpůsobena struktura této práce. Uvažujme zde myšlenkovou genezi od absolutní neznalosti problematiky až po nahlížení na problematiku z mnoha úhlů, na mnoha úrovních a nebo v mnoha dimeních.

1. **Analýza prostředí** - V tomto bodě je nutné prozkoumat specifické oblasti odvětví včetně jejich úskalí a omezení. S tímto bodem autorovi vypomáhaly osoby v této problematice zainteresované - buďto tak, že v oboru pracovaly či stále pracují, a to na různých pozicích a úrovních řízení. Je však nutné úvážit, že vyšší pozice (a od nich tedy i relevatnější, komplexnější a potenciálně obecně přínostnější informace) jsou často vázány mlčenlivostí.
   (a) **Osvojení si mikroprostředí podniku** - pochopit vztahy mezi jednotlivými potenciálními uživateli. Tím je myšlena specifikace procesů, jenž probíhají uvnitř podniku, obecnou organizační strukturu, role, které jednotliví zaměstnanci zastávají či například vztahy s okolními filiálkami.
   (b) **Osvojení si vnitřního makroprostředí** - identifikace „blízkých“ stakeholderů[6], konkrétně pak například odběratelé (zákazníci) a dodavatelé. Jakým vlivem vstupují do procesů, jaké události a stavy inicializují, a jaký mají vliv na chod podniku?
   (c) **Osvojení si vnějšího makroprostředí** - definice vztahů k subjektům s nepřímým vlivem na podnik. Jmenovitě lze uvažovat stát a jeho působení v pozici regulátora, legislativní omezení.

   Výstupem tohoto úkolu jsou analýzy **PEST**[7] a **SWOT**[8].

2. **Popis a konkretizace řešení** - Zde je nutné uchopit zjištěné role, spárovat je s pozicí v organizační struktuře a přiřadit příslušné procesy, které jí z tohoto titulu náleží. Následně je třeba vydefinovat poptávané řešení.
   (a) **Modelování procesů** - procesy je vhodné vymodelovat dle zjištěného schématu práce v příslušném nástroji. Jako nejvhodnější pro potřeby této práce se zdá být model **EPC**[9]. Alternativním by mohl být model **BPMN**.
   (b) **Detailní definice řešení** - uchopení požadavků, nároků a definice funkcionalit.
   (c) **Definice přípustných řešení a výběr nejvhodnějších** - výběr technologií a přístupů, pomocí kterých lze naplnit potřeby a vybrat z nich nejpříslušnější problematice

[6] rozuměj zájmové skupiny

[7] *„Political, Economical, Social, Technological (influence)“*, tedy kdo (potažmo co) může z různých úhlů pohledu ovlivnit naše snažení - jmenovitě pak vlivy *„politické, ekonomické, společenské a technologické“*.

[8] *„Strengths, Weaknesses, Opportunities, Threats“*, což můžeme přeložit jako *„Silné a slabé stránky, příležitosti a hrozby“*. Tato analýza slouží k identifikaci majoritních faktorů - vnitřních a vnějších pozitiv i negativ, s cílem je posléze uvážit při modelování.

[9] *„Event-driven process chain“*, neboli *„Řetěz procesů řízených událostmi“*, což je metoda/model pro zachycení jakékoliv činnosti navazující na událost, stav či jeho změnu, která logicky vyústí opět v stav či událost.[6, kapitola 12][30]

3. **Tvorba návrhu aplikace** - Definice jednotlivých aspektů a úkolů před samotným procesem programování, ze kterého se následně vychází. Vedle toho je nutné identifikovat neměnná jádra aplikace, stejně jako identifikovat ty soubory funkcionalit, které budou podléhat průběžným úpravám nejčastěji a předpřipravit je na to.
   (a) **Definice topologie systému** - formální definice sítě zařízení, která mezi sebou mají komunikovat, návrh infrastruktury
   (b) **Definice použitelných návrhových vzorů** - určení vzorů, které mají být použity a implementovány, možné odůvodnění
   (c) **Tvorba komunikačních protokolů** - popis stanovení, jak by měla probíhat komunikace mezi jednotlivými uzly v informační síti.
   (d) **Tvorba modelů a diagramů** - závazné modely a diagramy, pomocí kterých lze standardizovat vývoj. Příkladně k tomuto dílčímu cíli vede možnost užití následujících modelů:
      i. Specifikace aktérů
      ii. Konceptuální model databáze
      iii. Usecase model
      iv. Diagram tříd
      v. Další modely zaznamenávající relace

4. **Tvorba kódu** - Je nutné implementovat vymodelované abstraktní konstrukty, průběžně je testovat pomocí jednotkových testů či manuálních testů a operativně řešit průběžně objevené problémy.

5. **Tvorba výsledné implementace a testování** - Návrh elementárního grafického rozhraní a testování funkce systému jako celku - tedy testování nejen dílčích atomických funkcí, ale komplexní plnění business logiky.

# 3. Analýza prostředí

V této kapitole autor popisuje zjištěná specifika odvětví, jenž modeluje, a to s důrazem na případné potřeby implmenetace do systému.

## 3.1 Makroprostředí podniku a fungování v odvětví

Nyní uvažujme podnik coby instanci v prostředí, kde je jich podobných nepočitatelně a zahrnuje mimo jiné i ostatní ovlivňující prvky, jenž můžeme chápat jako například dodavatele či zákazníka, a je ovlivňována z vnějšku nejen okolními instancemi (podniky podobného zaměření), ale i svými vzdálenými stakeholdery, kterými jsou například stát, média či trestná činnost kybernetická[1].

### 3.1.1 Vnitřní makroprostředí

Hlavní kategorie subjektů v tomto prostředí si můžeme vymezit následovně:

- **Zákazník** - osoba v přímém kontaktu s podnikem, s relativně (vůči množině zákazníků) významným vlivem na podnik - jeho způsob fungování, jeho efektivitu a silnou mírou i na jeho preference (například zisk). **Je tedy nutné uvažovat i jeho preference**
- **Dodavatel** - aktér s přímou ekonomickou vazbou na sledovaný hypotetický podnik. Ve vzájemné komunikaci se obvykle využívá specializovaných systémů. Je vhodné vytvořit podporu komunikace s dodavatelem.
- **Konkurence** - aktér, kterého chceme naopak cíleně ze systému vyloučit, alespoň do míry vlivu na konkrétní implementaci. Ovšem pro potřeby této práce musíme na konkurenty nahlížet z vnějšího postavení pro zajištění pilíře *Znovupoužitelnost a univerzalita* z podkapitoly *Stanovení hlavních pilířů práce*. Proto jsou také předmětem zkoumání, ale s cílem pochopit drobné odlišnosti a zaznamenat je ve smyslu přizpůsobit systém co nejvyšší variabilitě.

**Zákazník** je pro podnik jedním z klíčových stakeholderů a tím pádem významně ovlivňuje chod celé organizace. Kromě mnoha jiného, v roli racionálně uvažujícího ekonomického subjektu může zásadně ovlivňovat cenu zboží (a tím de facto do jisté míry určovat zisky podniku) a to buďto substitucí nakupovaného zboží (záměnou zboží $A$ za zboží $B$) - tedy změnou struktury nákupního koše, nebo přechodem ke konkurenci (odchod od prodejce $A$ k prodejci $B$). Motivace k prvnímu zmíněnému je obvykle především cena, nicméně například i rozdílná kvalita (relativně k ceně) může hrát významnou roli. Vedle toho druhou pro podnik nepříz-

[1] To je taková, kterou si pracovně můžeme vydefinovat jako s cílem mařit a narušovat činnost podniku či systému, této se snažíme předcházet dle pilíře *Bezpečnost* v podkapitole *Stanovení hlavních pilířů práce*

nivou alternativu (tedy záměnu prodejců) zákazník volí v případě, kdy mu nevyhovuje zboží (cenou, kvalitou či z jiných důvodů - obdobně jako v rámci substituce zboží) nebo z důvodu nevyhovujících komplementárních „produktů“, které podnik (ne)nabízí.[2] K systematickému řízení vztahu se zákazníky se využívá služeb tzv. **CRM**[3] a v koncepci **SOA**[4] je zahrnut.

Shrneme-li tuto úvahu, dostáváme, že **zákazník je citlivý na poměr ceny a kvality a na kvalitu poskytovaných služeb**.

**Dodavatel** je dalším klíčovým hráčem. Zatímco zákazník zboží odebírá, za což podnik dostává úplatu, dodavatel je odpovědný za dodávku zboží s cílem dosažení výnosu. Tento akt předání s sebou nese pro podnik i mnoho problémových oblastí.

1. **Cena** - Přirozeným vlivem, na který je podnik citlivý je právě cena. Abstrahujeme-li od fixních nákladů, získáváme ty variabilní - tedy náklady rostoucí při „spotřebě“. Podnik musí efektivně řídit i tyto své náklady, aby byl finančně provozuschopný. Máme-li zákazníka, jenž je citlivý na vývoj ceny, musí být pak i prodejce cenově citlivý směrem ke svému dodavateli.
2. **Kvalita** - U kvality je to obdobné jako u ceny, tedy přenos citlivosti na kvalitu od zákazníka k dodavateli.
3. **Skladové řízení a skladová logistika** - Každý sklad má své pevně stanovené bezpečnostní kapacity, lze tedy jen obtížně zajistit dostatečnou zásobu v dostatečném sortimentu, není-li sklad neúměrně kapacitně vybavený vůči potřebám podniku. Přihlédnuv k trvanlivosti spotřebního zboží, i to je nutné řídit, v ideálním případě metodou **FIFO**[5], což ovšem samo o sobě nutně nezajistí uspokojení těchto nároků. V praxi se pro tuto problematiku a mnoho dalších, používají systémy **ERP**[6], jejichž součástí bývají i systémy **SCM**[7]

[2] Tím jsou myšleny vedlejší služby, jenž se mnohdy velmi často dají ekonomicky intrepretovat jen s obtížemi. Sem bychom mohli zařadit služby jako dlouhá otevírací doba, vhodné řazení produktů, dostupnost prodejny, stabilní dostupnost zboží, snadná orientace v prodejně, či například familiární a nápomocný personál.

[3] *„Customer Relationship Management“*, což můžeme přeložit jako *„Řízení vztahu se zákazníky“*. Používá se například v rámci marketingových operací, product promotion či k tvorbě statistik (s cílem budování či sledování naplňování podnikové strategie)

[4] *„Service oriented architecture“*, což lze přeložit jako *„Architektura orientovaná na služby“*, je jeden ze základních konceptů návrhu architektury informačního systému, kdy komplexitu (a tedy nesnadné mapování) překomponujeme na služby.

[5] *„First in, first out“*, což se dá přeložit jako *„První dovnitř, první ven“*, je zásobníková metoda řízení prvků v úložišti. Je postavena na principu fronty, kdy první, kdo vstoupí do úložiště také první úložiště opustí. V metodikách účetnictví je to ostatně z důvodů trvanlivosti spotřebního zboží minimálně doporučováno. Opačným principem je LIFO, tedy „Last in, first out“.

[6] *„Enterprise resource planning“*, což lze přeložit jako *„Plánování podnikových zdrojů“*, zahrnuje mnoho dalších systémů, přístupů a metodik pro zajištění efektivní správy víceméně libovolného vyčíslitelného podnikového zdroje.[8]

[7] *„Supply chain management“*, což lze přeložit jako *„Systém správy dodavatelského řetězce“*, pomocí kterého lze řídit mimo jiné i skladové položky (respektive dodávky zboží) z mnoha úhlů pohledu, a to víceméně automatizovaně či alespoň velmi intuitivně. Lze takto sledovat logistiku, tok informací či plánovat vztahy s dodavateli.[9]

Zjišťujeme tedy, že vztah podniku a dodavatelů je také nutné řídit a lze k tomu využít již existujících standardů (přístupů), kterými jsou ERP, respektive SCM.

**Konkurence** má v rámci tohoto odvětví poměrně specifickou strukturu. Konkurenty nelze chápat nutně jen jako soubor značek[8], ani tolik na úrovni divizí starajících se o různé velikosti prodejen jednotlivých poskytovatelů služeb[9]. Mnohem lepší obraz dostaneme, uvážíme-li konkurenci v rámci:

- **působnosti jednotlivých filiálek** (provozoven) - Pro příklad uvažujme dvě provozovny rozdílných (ale můžeme uvažovat i shodné) značek a rozdílných velikostí v poměrně snadno dosažitelé vzdálenosti od sebe. Tyto musíme brát jako naprosto validní vzájemné konkurenty. Naopak prodejci shodných velikostí v relativně významné vzdálenosti od sebe (například více než 20 km) již jako konkurenty lze považovat omezeně.[10] Přijmeme-li tu premisu za určující kritérium, lze z toho vyvodit důsledek, že konkurenci chápeme z pohledu preferencí a přípustnosti varianty pro zákazníka. Jinými slovy lze popsat míru vnímání konkurence (z pohledu podniku) prakticky identicky tak, jak ji vnímá zákazník, který je omezován vlastními možnostmi. Zákazník může například přejíždět za účelem levnějšího zboží (protože spotřebitelský přebytek bude vyšší), ale pouze do té míry, dokud je rozdíl spotřebitelských přebytků v obou (ve všech) variantách kladný. Pokud by klesl pod 0, racionální spotřebitel by variantu s cestováním za levnějším začal považovat za nevýhodnou a z uvažování by ji vyňal - a to samozřejmě za předpokladu jinak nezměněných užitkových veličin.
- **cenové struktury nabídky** - Racionální spotřebitel vždy volí výhodnější produkt (zpravidla je hlavním určujícím faktorem cena, ale může jím být i rozdílná kvalita produktu). Prodejci s diametrálnímě rozdílnými cenovými rozloženími jsou si mnohem méně konkurující, než ti s podobnou cenovou strukturou.

Vzájemně si mezi sebou nejbližší konkurenti (co do lokality nebo cenové struktury) určují ceny, panuje zde silná rivalita a silný boj o zákazníka a jeho loayalitu.

### 3.1.2 Vnější makroprostředí

Nyní poodhalme problematiku vnějšího makroprostředí s cílem identifikovat co nejlépe aktéry či regulátory. Standardním regulátorem je přirozeně stát a příslušná legislativa. Hlavními všeobecně platnými ustanoveními, která jsou zároveň v jistém smyslu platná i pro tento systém, jsou následující:

[8] Příkladně vztah mezi prodejcem Tesco, Kafuland, Lidl, Globus, atd.

[9] V rámci řetězce Tesco pak například Tesco Express, Tesco Supermarket, Tesco Hypermarket, atd.

[10] V závislosti na obecné relativní dostupnosti maloobchodních služeb v okolí. V rámci urbanistické oblasti je tento faktor v nižším rozsahu, než v méně osídlených oblastech, například na venkově.

- **Ochrana osobních úrajů**[11], kde musíme uvažovat zabezpečení informací a dat, s nimiž je v rámci organizace nakládáno. Týká se tak především ochrany dat jistým způsobem citlivých o všech lidských osobách, tedy zaměstnanců, zákazníků a případně dodavatelů. **Systém (respektive implementující osoba) by měl (respektive měla) zajistit zabezpečení těchto dat před jejich zneužitím a sám by je neměl využívat jinak, než je (kumulativně)**

  – smysl podnikání
  – umožněno zákonem

  Zjednodušeně se za splnění těchto podmínek dá považovat stav, kdy citlivé informace (data) nejsou zneužívána a ani není předpokladatelně možné takovéhoto porušení dosáhnout, tedy jsou implementována dostatečná opatření pro zajištění této bezpečnosti, osoba, jíž data náleží, je upozorněna na tyto skutečnosti[12], a zároveň není ukládáno dat nad nutný rámec svého ekonomického působení. Tato potenciálně kompromitující data je nutné mít možnost v případě dožádání vymazat. [10][11]

- **Zákon o ochraně spotřebitele** [12] - Tento zákon definuje vztah (co do práv a povinností) mezi prodejcem a spotřebitelem. Především pak vymezuje práva spotřebitele, respektive povinnosti prodejce, což ostatně vyplývá již z názvu zákona. Kupříkladu předepisuje řešení pochybení na straně prodejce, kdy nakoupené zboží nenaplňuje kvalitativně očekávané (respektive očekavatelné) vlastnosti.

- **Zákon o obchodních korporacích** [13] - Tento zákon definuje strukturu, vztahy a podmínky fungování korporace na území České republiky.

- **Zákoník práce** [14] - Neboť systém uvažuje podnik v relaci se zaměstnanci a se zaměstnanci nějakým způsobem pracuje, musíme uvažovat také právní úpravu tohoto vztahu, tedy vztah **Zaměstnavatel-Zaměstnanec**. Stejně jako jakýkoliv podnik, i podnik v tomto odvětví musí ctít tento zákon.

- **Zákon o kybernetické bezpečnosti** [15] - Ač tento zákon není plně spjat s tímto systémem, je vhodné tento použít alespoň jako soubor praktik pro inspiraci. Vedle tohoto je vhodné uvažovat při implementaci i standardy **ISO 27001** a **ISO 27002**, neboli **ISMS**[13].

- **Ochrana hospodářské soutěže** [16] - Tento zákon je uveden především pro úplnost,

[11] Je dobré si uvědomit, že původní Zákon o ochraně osobních úrajů *č. 101/2000 Sb.* se zrušuje zákonem *č. zákon č. 110/2019 Sb.*, a který je věcně doplněn *nařízením EU 2016/679 (GDPR)*[10]. Je velmi pravděpodobné, že subjekty podléhající této úpravě a splňují požadované podmínky, splňují z velké části i podmínky řčeného nařízení.[18]

[12] respektive je požádána o svolení

[13] *„Information Security Management System“*, což lze přeložit jako *„Systém řízení bezpečnosti informací“*, je standard metodik a přístupů s účelem zajištění bezpečnosti informací a její řízení, stejně tak jako řízení potenciálních rizik.[19]

neboť se týká ochrany hospodářské soutěže. Nadále v této práci uvažujeme, že podnik se nesnaží či jinak nenaplňuje skutkovou podstatu pokusu o nepovolený konkurenční boj, o monopolizaci trhu, nekalou soutěž či jiné porušování této právní normy.

- **Občanský zákoník** [17] - Tento zákon mimo jiné vymezuje člověka jako osobu, stanovuje její základní práva a povinnosti, a tím se zařazuje mezi primární postuláty právního chápání evidence lidí (rozuměj občanů). Pro potřeby podniku lze uvažovat tento (v kombinaci se zákonem o ochraně osobních údajů nebo GDPR) v souvislosti například s **evidencí zaměstnanců**.

## 3.2 Existující řešení

Jakožto existující řešení musíme uvažovat systémy pro správu výše definovaných oblastí, jež se v praxi používají. Sem bychom mohli zařadit několik majoritních typů systémů:

- **SOA** - Systém orientovaný svojí architekturou na služby, čímž se usnadňuje pochopení jinak tak komplexního celku. Krátký výpis (obvykle používaných) službeb lze rozdělit následovně:
  - Zmíněné **SCM v rámci ERP** pro řízení vztahu s dodavateli v návaznosti na řízení podnikových zdrojů
  - Zmíněné **CRM** pro řízení vztahu se zákazníky
  - **Finanční a účetní systémy**
  - **PLM**[14] a **PDM**[15] - Lze v následujících kapitolách práce abstrahovat, neboť se hodí především pro podniky výrobní.
  - **Skladové a skladově logistické systémy**[16] - Systémy pro řízení skladů a to jak z pohledu řízení skladových položek, tak z pohledu managementu v oblasti logistiky ve skladu (od plánování optimální cesty ve skladu až ke globálnímu logistickému sledování a plánování skladové politiky)
  - **HR systémy** - systémy pro správu lidských zdrojů

## 3.3 PEST

V této analýze autor popisuje konkrétnější vlivy ze strany faktorů **politických**, **ekonomických**, **sociálních** a **technických** s bližším zaměřením na podnik v odvětví maloobchodního prodeje.

[14] *„Product Lifecycle Management“*, což můžeme přeložit jako *„Řízení životního cyklu produktu“*

[15] *„Product Data Management“*, což můžeme přeložit jako *„Řízení dat o produktu“*, alternativně lze mluvit o správě dokumentace.

[16] Často bývá součástí již zmíněného SCM, není však podmínkou naplňovat tento vztah obousměrně.

### 3.3.1 Politické vlivy

Do této kategorie spadají všechny vlivy, které můžeme očekávat ze strany státu, vlády či legislativy; spadají sem obvykle i faktory zdanění.

- **Legislativní omezení marketingu** - Podnik naráží na problém, jak cenově efektivně cílit svoji reklamu tak, aby neporušoval legislativu platnou na území ČR. Podnik má poměrně silně omezené možnosti v rámci online propagace sebe či svých produktů. Nelze se spoléhat na „sledování“ a „měření“ svých (potenciálních) zákazníků, a to z titulu GDPR. V praxi to podniky často řeší členstvím v omezené skupině věrnostního klubu, kdy při každém nákupu se prokazují kartou s evidenčním číslem. Za takto předané informace podnik poskytuje specializované služby, obvykle vyjímečné slevy.

- **Právní ochrana spotřebitele** - Spotřebitel (zákazník) je chráněn zákonem proti zneužití svého postavení prodejcem, pokud se naplní příslušná skutková podstata. Zákazník pak má právo reklamovat obchod (rozumněj směnu), který nesplňuje podmínky smluvené, či obchodníkem klamavě propagované.

- **Prodej daňově zatíženého zboží** - Kromě tradičně daňově zatíženého zboží podnik obvykle prodává zboží, jehož prodejní cenu významně ovlivňuje **spotřební daň**[17]

- **Zboží se speciální právní úpravou** - Prodejce může mínit prodávat zboží se zvláštní právní úpravou. Z obvyklého může jít o prodej zboží se stanovenou minimální věkovou hranicí kupujícího. Standardně lze mluvit o alkoholických nápojích, tabákových výrobcích či třeba o zábavní pyrotechnice.

### 3.3.2 Ekonomické vlivy

- **Silná závislost na ekonomickém cyklu** - Toto odvětví silně podléhá ekonomickému cyklu. Zákazníci v době ekonomické dekadence volí levnější zboží (přesněji lepší poměr ceny ku užitku) s vidinou spořivějšího žití. Naopak v době ekonomického růstu lze očekávat i silný nárůst zájmu zákazníků o „luxusní“ zboží - takové, které má tento poměr podstateně horší.

- **Náročné upevňování pozice na trhu** - Podniky v tomto odvětví mají problém se zásadně odlišit v pozitivním slova smyslu. Je pro ně náročné vytvořit přidanou hodnotu, neboť je jejich cílem minimalizace ceny (na kterou je zákazník citlivý) za účelem alespoň stabilizovat zisky a tím zvyšovat svůj tržní podíl. Mají tedy velmi úzký finanční prostor pro přidávání hodnoty ke každému nákupu.

- **Vysoké náklady na vstup do odvětví** - Lze očekávat pouze slabé pokusy o vstup na tento trh, neboť náklady na něj jsou velmi vysoké.

[17] obvykle například tabák a tabákové výrobky, víno, destiláty a jiné lihoviny

### 3.3.3 Sociální vlivy

- **Levná pracovní síla** - Na nižších úrovních lidských zdrojů v tomto odvětví nejsou nijak zvlášť specifické podmínky pro vykonávání dané profese. Plyne z toho jednak (v jisté podobě) „solidarita“, že podnik poskytuje pracovní pozice pro jinak obtížně zaměstnatelné, stejně jako z toho plynou teoreticky nižší náklady, aplikujeme-li základní ekonomické axiomy.

- **Tendence ke slevám** - V tomto odvětví podniky volí často strategii silného „slevového přehlcení“, a to z důvodů správy skladových zásob (například likvidace zboží s krátkou trvanlivostí), či z důvodu „sezónního marketingu“, kdy podnik nabízí zboží s vazbou na období v roce[18]. Tímto aktem se mimo jiné podnik snaží upevnit svoji pozici na trhu, respektive upevnit svůj **brand** v povědomí zákazníků.

- **Čas zákazníků** - Uvažujeme-li zákazníka jako racionálně ekonomicky uvažující subjekt, musíme uvážit i jeho preference v oblasti jeho nákladů na veličinu času. Zákazník si uvědomuje strávený čas při nákupech a měl by být dle jeho soudu úměrný užitku sníženému o cenu nákupu, který z celého procesu nakupování má.

### 3.3.4 Technologické vlivy

- **Zastarávání technologií** - Jednotlivé technologie jsou zatíženy postupným zastaráváním. Tento jev může být způsoben vznikem potřeby vývoje tranzitu informací. Týká se však především sofistikovaných a hlubokých infratruktur.[19]

- **Cílení reklamy pomocí znalostního inženýrství** - Existuje mnoho způsobů, jak s jistou pravděpodobnostní přesností identifikovat, kvantifikovat a kvalifikovat preference spotřebitelů. Jejich využití je právně omezeno co do sběru, uchovávání a zpracovávání dat. Výsledky těchto analýz však lze velmi silně využít na mnoha polích působnosti podniku v tomto odvětví.

- **Rostoucí nároky na zabezpečení IS** - Lze pozorovat jistou korelaci mezi objemem dat v systému a významností systému jakožto potenciálního cíle útoku. Lze také uvažovat neustálý růst datových skladů a je tedy třeba zvyšovat úroveň svého zabezpečení, jakožto ochranu dat, systému, fyzické i virtuální infrastruktury, dobrého jména firmy (důvěryhodnosti) a ochrany obchodního tajemství (konkurenční výhody).

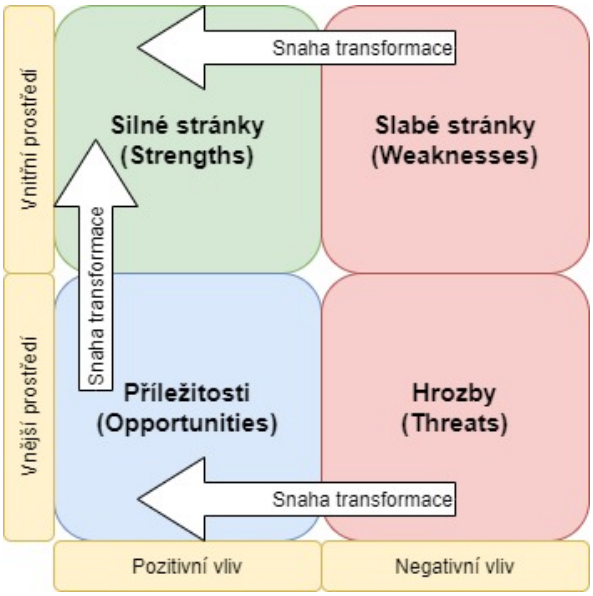
[18] Očekává silný nárůst poptávky po tomto statku, proto se před- a nadzásobí, čímž mu vzniknou úspory z rozsahu, o ty pak může produktu snížit prodejní cenu. Příkladně může posloužit zásobování podniku zábavní pyrotechnikou před koncem roku či květinářské produkty před svátkem sv. Valentýna.

[19] Příkladem může být nemožnost implementace nových logistických trendů či potřeb.

## 3.4 SWOT analýza

Cílem SWOT analýzy je identifikovat a kategorizovat efekty, které se objevují či mohou objevit. Tyto efekty se obvykle zapisují do tabulky (matice) tvořené čtveřicí prostor seřazených dle rostoucí negativity (horizontální osa) a míře vlivu z vnější (vertikální osa).

Na následujícím obrázku je vyobrazena struktura SWOT matice a snaha podniku transformace jednotlivých aspektů.




Obrázek 3.1: Ukázka struktury SWOT matice

Na obrázku *Ukázka struktury SWOT matice* je patrná kromě struktury i snaha podniku o transformaci negativních aspektů (slabé stránky a hrozby) do těch pozitivních (silné stránky a příležitosti), stejně jako přeměna příležitostí do silných stránek, tedy přenos pozitiv z odvětví do svého vlastního portfolia. Pro příklad bychom mohli uvést postupný přechod prodejců k poskytování služeb bezkontaktního a bezhotovostního placení.

Tato SWOT analýza především přeformulovává výstupy předchozí analýzy *PEST*.

### 3.4.1 Strengths

- **Silná vyjednávací síla vůči dodavatelům** - Pomocí systematického řízení vztahů (a obecné povaze vztahu) podnik disponuje schopností vyjednávat o ceně za dodávané zboží.
- **Informační asymetrie** - Zákazník není schopen efektivně porovnávat všechny produkty mezi všemi konkurenty.

- **Nízké nároky na personál** - Na nižších pozicích v podniku jsou minimální kvalifikační nároky pro uchazeče. Tím se zvyšuje nabídka na potenciálním trhu práce, čímž se zvyšuje vyjednávací síla prodejce a může si dovolit snížit své mzdové náklady pro obsazovanou pozici.
- **Významné úspory z rozsahu** - Při obchodování ve velkém lze obvykle dosáhnout množstevní slevy. Stejně tak lze uvažovat nižší komplementární náklady.[20]
- **Vysoké náklady na vstup do odvětví** - Vznik nového konkurenta je prakticky vylučitelné riziko, čímž vzniká velmi silná pozice relativně nízkého počtu poskytovatelů těchto služeb na tomto trhu. Negativem může být vznik živné půdy pro **oligopol**, což lze chápat jako negativum z pohledu subjektivního i objektivního.

### 3.4.2 Weaknesses

- **Rychle se měnící nároky na IS** - Stejně jako v mnoha jiných odvětvích podléhají informační systémy postupnému zastarávání a postupně se snižující schopnosti reagovat na potřeby jeho uživatelů. V tomto odvětví, kde je informace instancí páteřního hybatele podniku, je patrné ještě významnější riziko nedostatečnosti systému. Podniky by tedy měly volit systémy s co nejvyšší variabilitou a schopností adaptace nových přístupů a potřeb pro minimalizace nákladů na vývoj a provoz.
- **Silná závislost na preferencích zákazníka** - Málokterý podnikatelský obor se tak silně specializuje na uspokojování potřeb zákazníka tak, jako maloobchodní sítě. Svědčí o tom relativní ziskovost podniku, kdy obchodní marže vznikající přidanou hodnotou za naskladnění a redistribuci je vůči nákladům a celkové ceně velmi nízká. Stejně tak můžeme mluvit o silné konkurenci, kdy nelze považovat konkurenta jen na bázi porovnávání brandu - musíme uvažovat jednotlivé filiálky co do jejich tržeb, lokality, počtu prodávaných druhů produktů a mnoho dalších parametrů.
  Minimalizace této slabé stránky je možná:
  1. **diferenciací** - poskytnutí unikátních produktů a služeb, jež ostatní poskytnout nemohou
  2. **prostou lokální monopolizací** - stát se majoritním poskytovatelem agregovaných produktů a služeb o lokální působnosti

  Tyto výše uvedené cesty jsou však velmi náročné a to nejen co do nákladů. Kupříkladu uvážujme, že lze všechny potenciální zákazníky zahrnout do jedné spotřebitelské skupiny. Dále uvažujme, že tuto skupinu lze zpětně dekomponovat a kategorizovat dle doby, za kterou si daný zákazník zvykne na novinku v přístupu či službě, a začne ji využívat. Za jinak běžných předpokladů by měla křivka odpovídat křivce limitně se blížící normálnímu rozdělení[21], z čehož vyplývá, že většina spotřebitelů se nachází v okolí středu sledovaného období.[22]

[20] například náklady na přepravu

[21] Přesněji obsah plochy mezi křivkou Gaussova rozdělení (pro stejné vstupní parametry) a křivkou reprezentující sledovanou skupinu subjektů se limitně blíží nule.

[22] tzn. znatelně zpožděnou reakci na zavedení nového standardu

### 3.4.3 Opportunities

- **Minimalizace neefektivních a redundatních časových nároků na nákup** - Rozdělíme-li si čas, který zákazník stráví nakupováním, do kategorií, můžeme identifikovat a zaměřit se na ty nesoucí nejmenší užitek pro podnik. Tyto jsou následující:
  - **Výběr zboží** - Zákazník při nakupování porovnává jednotlivé produkty v několika různých dimenzích:
    * prodejní cena
    * sleva
    * kvalita
    * užitek z nákupu produktu
    * věrnost ke značce

    Dále pak porovnává jednotlivé kusy produktů dle jejich data spotřeby - racionálně upřednostňuje takové zboží, které vydrží déle, čímž ovšem mohou vznikat podniku druhotné náklady z již neprodejného zboží.
    Z tohoto titulu lze tvrdit, že čas, který zákazník tráví výběrem zboží je pro podnik silně neefektivní. Vedle toho pro zákazníka můžeme tyto náklady chápat jako náklady obětované příležitosti, kde příležitostí rozumíme zboží lépe vyhovující jeho preferencím[23].

  - **Fronty** - Čas, který zákazník stráví čekáním ve frontách. Pro zákazníka je to doba z drtivé většiny naprosto neefektivní, neboť jen čeká a nevyvíjí žádný užitek ani pro podnik. Jedinými dvěma užitnými hodnotami, jenž zákazník potenciálně vytváří, je uvažování nad nákupem něčeho, co zapomněl či co právě vidí, nebo sledováním reklam a jiných podpůrných propagačních objektů v dohledu.

  - **Hledání zboží** - Kumulace dob, které zákazník stráví procházením prodejny a snaží se najít konkrétní (nebo obecný) produkt. Povaha této doby má dvě strany - pozitivní, coby propagačního nástroje, kdy zákazník nakupuje zboží, které neměl v plánu zakoupit, a negativní, kdy kvalita podniku v očích zákazníka klesá, neboť takto ztrácí svůj čas.
    Možností řešení je například zlepšení navigace pro snazší orientaci v prodejně.

  - **Prostorová nedostatečnost** - Agregátní doba čekání na průchod, tedy souhrn časových intervalů, které zákazník tráví čekáním, neboť není schopen projít (respektive projet s vozíkem) z důvodu prostorového přehlcení prodejny zákazníky. Tento čas lze chápat jako silně negativní pro obě strany vztahu. Zákazníkovi nepřináší prakticky žádný měřitelný užitek a podnik ztrácí schopnost sledování zákazníků.[24]

  Redukcí hodnot těchto časových kategorií můžeme potenciálně snížit nadbytečné náklady. Cenou za to je nutnost nalezení jiného marketingového nástroje pro propagaci

[23] neuvažujeme-li faktor, kdy se zákazník z části uspokojuje samotným nakupováním bez nutného přičinění prodejce

[24] A to za účely marketingu nebo i kontroly proti krádežím

vlastní značky, značky prodejců a pro nepřímé řízení skladových zásob. Zároveň takto uspokojíme hypotetického racionálně uvažujícího zákazníka, zvýší se kapacita potenciálu obsloužených osob a zvýší se plynulost prodeje.

- **Zvyšování efektivity skladového řízení pomocí IT** - Podnik z odvětví masového maloobchodu je silně závislý na řízení skladů, což lze popsat jako poměrně složitou optimalizační úlohou pracující s mnoha úrovněmi a mnoha dimenzemi. Ovšem při úspěšném managementu zásob je podnik schopen se stát **cenovým šampionem**[25]. K tomu je velmi vhodné využít informačních technologií s cílem provádění snadných, ale velmi často opakovaných operací nad zbožím.

### 3.4.4 Threats

- **Silně cyklicky zaměřené odvětví** - Jak bylo zmíněno výše, toto odvětví je silně cyklické, tedy ziskovost je silně závislá na ekonomické situaci. Je tudíž třeba silně plánovat počínání podniku do budoucnosti.[26]

- **Riziko kybernetických útoků** - A to za účelem nelegálního získávání dat, získání kontroly nad zařízením či s cílem **DoS**.[27] Nejen, že může dojít k porušení vícero právních předpisů či vznik významných škod (a to nejen na informatických zdrojích), podnik může ztratit konkurenční výhodu, a to vyzrazením obchodního tajemství nebo poškozením dobrého jména firmy. [20, 21, 22]

- **Přechod zákazníka k lokálním a specializovaným prodejcům** - Vyvíjí se trend zpětného návratu zákazníků do prodejen drobných podnikatelů s konkrétními produkty, velmi často vlastní výroby. Konkrétně jde například o pekárny, masny a uzenářství, vinárny nebo cukrárny. Velcí hráči[28] vytlačili z trhu drobné prodejce. K těm je však trend zákazníků se navracet, motivují ho k tomu následující důvody:
  - **Vyšší kvalita** - Zákazník očekává od podniku s úzkou specializací mnohem vyšší kvalitu, zvláště jde-li o prodejnu lokálního výrobce. Zákazník pak akceptuje i vyšší cenu.
  - **Vysoce specializovaný produkt** - Zákazník není schopen potřeby naplnit u masového prodejce, neboť takových produktů nevede (například z nedostatečné poptávky).
  - **Osobnější přístup** - Zákazník očekává osobnější přístup, případně pomoc s výběrem, čehož se mu dostává v prodejně supermarketu jen omezeně.
  - **Osobní styl** - Lze považovat za společenský trend vymykat se moderním způ-

[25] Díky úsporám z rozsahu, kdy je schopen podnik ušetřit velké finanční prostředky a tím i poskytnout významně nižší cenu než menší podniky v obdobném odvětví.[23]

[26] Například plánování skladových položek dle očekávaných spotřebitelských preferencí a finančních možností.

[27] *„Denial of Service“*, což můžeme přeložit jako *„Odepření (odmítnutí) služby“*, je útok s cílem narušit IT službu tak, aby ji nikdo nemohl používat. Další variantou je **DDoS**, tedy distribuovaný DoS, kdy velmi často jednotliví uživatelé ani nemusí vědět, že se pomocí škodlivého kódu jejich zařízení pokouší skupinově o útok. [20]

[28] rozumněj velké samoobslužné prodejny - supermarkety

sobům, z ekonomického hlediska jde o značně „luxusní“ způsob uvažování.

– **Pocit podpory lokálního výrobce** - Motivace kupovat zboží, u kterého zákazník ví, že je to územně limitovaný výrobek, a cítí jistou sounáležitost a příslušnost. Tento nákup pak realizuje pro dosažení další přidané hodnoty, u níž je velmi náročné ji kvantitativně[29] ohodnotit.

[29] rozumněj ekonomicky

# 4. Specifikace procesů podniku

Nahlížíme-li na podnik jako na sociálně-ekonomický subjekt, uvažujeme, že požívá nějaké **vstupy**, z nichž generuje patřičné **výstupy**. Ve zjednodušené podobě pak můžeme vstupy chápat například informace (resp. data) či materiální entity (např. dodávky materiálu, finance), a za výstupy pak obvykle produkt (tedy službu či statek k prodeji, z čehož vznikají další finanční prostředky) a další informace (resp. data). K této **transformaci z vstupů do výstupů** pak dochází pomocí mnoha různých procesů, na nichž může participovat mnoho různých aktérů, či mnoha různých aktérů se jejich provádění týká. Zkoumáním těchto procesů se tato kapitola zabývá.

Autor v této části volí postup pochopení organizační struktury podniku včetně základního rozdělení rolí, dále formulaci základních událostí pro podnik typických (ve vztahu k této práci) a konečně procesy, kterými se podnik řídí a jejichž naplňováním směřuje ke své majoritní strategii.

Autor se k modelování podniku v tomto odvětví opírá o konzultace shrnuté v příloze *A Záznamy z odborných konzultací* počínaje stranou *109*.

## 4.1 Organizační struktura

Jako organizační strukturu chápejme model podniku, který obsahuje vlastní jmenné prostory tématicky propojených pojmů (oddělení), v němž existuje vícero druhů rolí, jež může zastávat více osob (obvykle zaměstnanců). V takto ohraničeném prostoru můžeme modelovat procesy, odpovědnosti za provedení procesů, sledování událostí a přerozdělení rolí.

V podniku typu supermarket uvažujme role následující:

- **Provoz** - Funkce podniku z materiální stránky
  - **Vedoucí skladu** - Osoba odpovědná za fungování skladu, dále se stará o objednávání zboží.
  - **Vedoucí prodejny** - Osoba odpovědná za plynulý chod prodejny. Obvykle řeší a kontroluje nedostatečně naskladněné zboží na prodejně.
  - **Vedoucí pokladen** - Stará se o plynulý chod na pokladnách a řeší problémy zde vzniklé.
  - **Skladník ve skladě** - Provádí jednoduché operace se zbožím na skladě. Připravuje zboží na naskladnění v prodejně a naskladňuje na sklad zboží přivezené dodavatelem.
  - **Skladník v prodejně** - Doplňuje zboží v prodejně.
  - **Kontrolor kvality zboží** - Kontroluje kvalitu zboží od dodavatele, zda splňuje kvalitou a kvantitou dodávka objednávku.
  - **Logistik** - Plánuje objednávky a způsob jejich doručení. Neformálně i plánuje

fungování skladu.

- **Administrativa** - Fungování podniku z formální stránky[1]
  - **Účetní oddělení** - Řeší události finančního charakteru
  - **Management** - Řeší plnění strategie podniku. Může obsahovat i role HR a PR, zpravidla je tomu tak v závislosti na velikosti provozu.
  - **PR oddělení** - Stará se o tvorbu a plnění PR strategie[2]
  - **HR oddělení** - Stará se o provoz z pohledu pracovní síly.

## 4.2 Analýza aktérů

Jako aktéra můžeme chápat takovou osobní jednotku, jíž náleží právo provádět předem stanovený úkol v předem stanoveném rozsahu. Tento úkon pak chápejme jako proces. K modelování aktérů použijme základní řazení dle povahy jejich působení v podniku. Pro potřeby práce uvažujme pouze provozní část podniku pro potřeby této práce relevantní, tedy tu starající se o materiální stranu fungování.

Z tohoto pohledu pak máme aktéry a k nim přiřazené akce následující:

- **Zákazník**
  - Zadává objednávku
  - Platí za zboží
  - Přebírá zboží
- **Skladník**[3]
  - Kontroluje přijaté zboží
  - Přemisťuje zboží z místa na stanovené místo
  - Kontroluje stav zboží
  - Připravuje zboží na další transport
  - Provádí namátkové kontroly zboží
- **Supervisor** - vedoucí zastávající roli kontrolora
  - Provádí kontrolu naplňování stanovených úkolů zaměstnanci
- **Administrátor** - vedoucí zastávající exekutivní roli
  - Upravuje stanovené úkoly

### 4.2.1 Informační systém

Informační systém není zvykem považovat za aktéra, neboť zastává roli nástroje pro dosahování cílů. Pro naše potřeby ho však mezi ně zařadíme, jelikož chceme, aby některé operace

[1] Touto oblastí se zabývejme jen okrajově a pro úplnost.

[2] Často toto oddělení mít provozovny tohoto typu nemusí, neboť jsou úlohy náležící PR řešeny centrálně; a to na úrovni kraje, státu či regionu. Lze se setkat i se situací, kdy je tato role syntetizována s rolí managementu.

[3] Syntetizujme skladníka prodejny a pracujícího ve skladě, neboť jejich role jsou prakticky identické a pro potřeby této práce je hodnota jejich zásadního rozlišování prakticky nulová.

prováděl víceméně autonomně, bez významného přičinění člověka. Informační systém chápejme, že může vystupovat pro potřeby podniku ve dvojí roli:

1. **Nástroj** - poskytuje (drobné) služby s cílem pomoci svým uživatelům v dosažení maximální možné efektivity a produktivity.
2. **„Dirigent“** - řídící orgán, jenž kontroluje dodržování pravidel příslušným procesům vlastní.

V případě obecného informačního systému obecného podniku pak stavíme na několika obecných stavebních kamenech, které by každý systém měl mít integrované:

- **Zajištění důkazovosti** - IS by si měl vést vlastní evidenci operací, které provedl.
- **Zajištění komunikace** - IS by měl být schopen komunikovat a nechat okolní uzly provádět operace v souladu s posledním jmenovaným.
- **Zajištění důvěrnosti** - Neměl by umožnit přístup nepovolané osobě a neměl by umožnit ani odposlouchávání komunikace.

## 4.3 Události

Jako první, co je úvodem této sekce nutné udělat, je co nejpřesnější definice pojmu „událost“. Napříč obory, které se odvětví mohou týkat, je druhů událostí mnoho a mnohdy se liší nejen co do povahy, ale třeba i smyslu, přístupu ze strany podniku a výsledného efektu.

Jako událost tedy chápejme obecně takovou změnu stavu s potenciálem spustit nějakou reakci - **proces** (přirozeně z business pohledu). Tato událost může být jak vzniklou, velmi specifickou situací, tak i rutinním efektem, s nímž se potýkají zaměstnanci takřka neustále. Cílem modelování událostí pak je podchytit co největší objem na úrovni abstrakce, aby bylo možné jednoznačně určit, jak se bude za běžného provozu na cokoliv reagovat.[4]

Nyní si vypíšeme základní typové události, které v běžném podniku v odvětví maloobchodního řetězce ve vztahu k zákazníkovi mohou vzniknout, nehledě na jejich příznivost, kdo by na ni měl reagovat či jak by na ně mělo být reagováno. Nadále uvažujme tyto za (pro tuto práci) stěžejní součást business logiky podniku.

1. Zákazník vytvořil objednávku
2. Objednávka je vyhotovena
3. Zaměstnanci byl přiřazen úkol
4. Zaměstnanec splnil úkol
5. Zaměstnanec nesplnil úkol
6. Objednávku není možné vyhotovit

[4] V praktických podmínkach je toto nejen velmi obtížné, je to takřka nemožné, neboť se nelze připravit na vše. Jde však o to pochopit, co se může stát (s nějakou rozumnou mírou pravděpodobnosti), a ošetřit to navěšením příslušného procesu náležícího příslušné osobě - roli.

## 4.4 Procesy

Chápejme, že na jmenované události reagujeme procesem k nim spjatým - jak tuto událost řešit. Tyto procesy pak udávají rámec funkcí a procedur, které uvádí systém do chodu a naplňují business logiku podniku.

Tyto procesy jsou modelovány tak, aby odpovídaly reálému, nynějšímu stavu. Jsou však autorem poupraveny, aby sloužily i jako materiál k tvoření platformy. Lze tedy uvažovat například to, že tvorbou objednávky zákazníkem je myšleno předpřipravení si seznamu položek, které chce nakoupit. V modelované výseči světa by tomu pak bylo přidávání položek do virtuálního košíku, jenž by byl následně validován.

Procesy jsou formulovány pomocí grafického jazyka **EPC**[5], který byl zvolen pro jeho snadnou čitelnost, jasnost, snadnou syntaxi a relativně vysokou vyjadřovací schopnost. Alternativou by mohl být například jazyk **BPMN**[6, strana 321-322].

### 4.4.1 Zákazník vytvořil objednávku

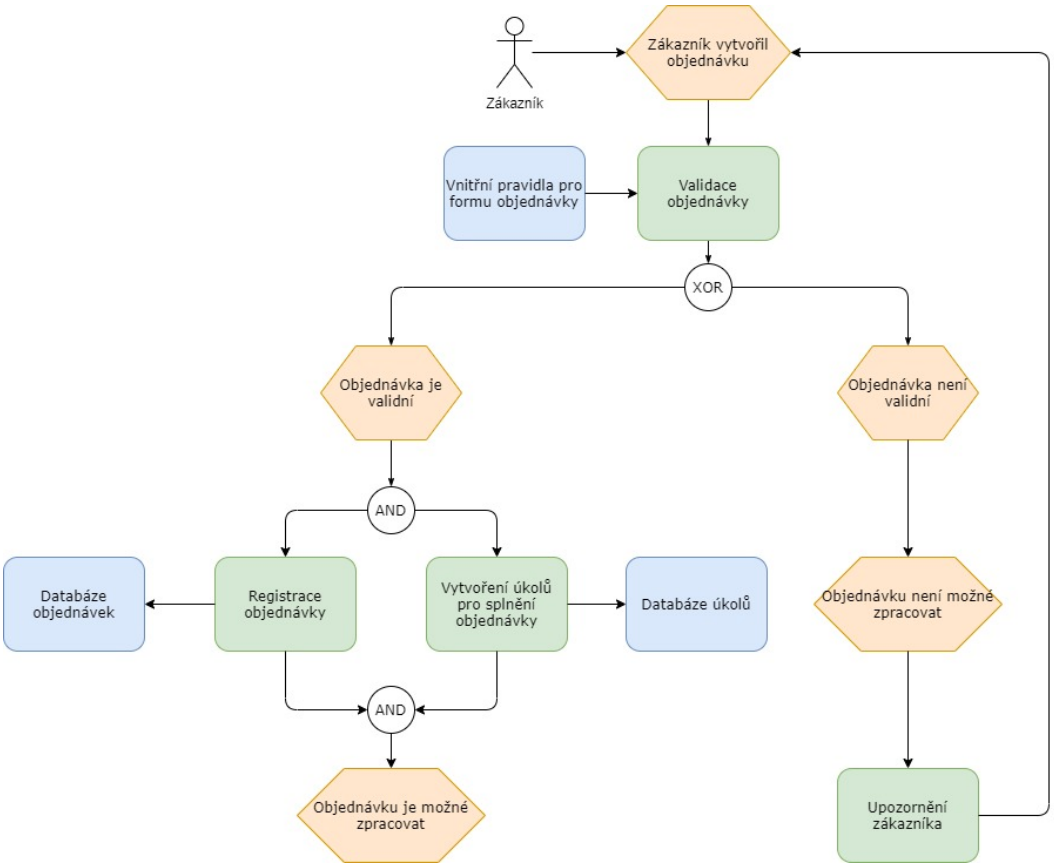
Na událost „Zákazník vytvoří objednávku“ reagujeme procesem její zpracování, viz *Upravený proces události „Zákazník vytvoří objednávku“*. Tento proces popisuje ověření zadané objednávky a úkoly k ní spjaté. Tato objednávka je v této fázi jen ověřována co do validity její formy. Neověřují se zde faktické údaje - například zda zboží je v dané kvantitě na skladě. Pokud tato objednávka je shledána nevalidní, je na to zákazník upozorněn a je mu dán prostor pro napravení příslušných pochybení. V opačném případě se zaeviduje a vytvoří se soubor úkolů pro tuto objednávku.

Pro práci bezpečnějším způsobem však může být jiný proces, který definuje proces tvorby objednávky pomocí předobjednávky. Tato sekvence úkonů má hlavní pozitivum v tom, že systém je stále upozorněn o změně poptávaného nákupního koše, má však možnost s tímto ještě stále pracovat. Zákazník si vložením zboží do košíku de facto rezervuje dané množství a lépe odpovídá i modelu reality - aktuálního stavu nákupu.

Na obrázku *4.2 Upravený proces události „Zákazník vytvoří objednávku“* na straně *42*je podrobněji posán proces tvorby předobjednávky. Ta slouží především k zajištění integrity. V reálném světě si zákazník rezervuje zboží tím, že ho vloží do nákupního vozíku. V momentě vložení mu nemůže již nikdo do ukončení nákupu z vozíku cokoliv vyjmout.[6] Objednávku posléze stvrzuje tím, že u poklady zaplatí svůj nákup.

[5] *„Event-driven process chain“*, tedy *„Řetězec procesů řízených událostmi“* je notace, která umožňuje znázornit proces jako soubor střídajících se událostí (případně stavů) a akcí, které na ně reagují. Dále jazyk umožňuje kromě těchto objektů a vazeb mezi nimi vkládat i další informační vstupy (dokumenty, informace) a aktéry.[6, strana 318-320][30]

[6] Neuvažujme nekalou praktiku, kdy jiný zákazník spatří v košíku něco, co by musel složitě hledat v prodejně a převede toto zboží do svého depozitáře.




Obrázek 4.1: Proces události „Zákazník vytvoří objednávku“

Pro potřeby práce tento proces nákupu dekomponujme (přijmeme-li tuto analogii) na situaci před dovezením košíku k pokladně a další nutné operace poté.

Při porovnání procesů bez předobjednání a s ním je patrné, že na procesu jako takovém se příliš velkých změn nedocílí. Po stránce datových toků je však změna zásadní. Provedením tohoto rozšíření tvoříme další koncept ukládání zboží. Výhody nám z toho plynou například v tom, že jsme schopni lépe sledovat průběh plnění procesu[7], a pro pochopení business logiky pak uvažujeme, že předobjednávka je stále ještě založena explicitně na komunikaci mezi zákazníkem a podnikem. V momentě transformace předobjednávky v objednávku již podnik závisí jen sám na sobě.[8]

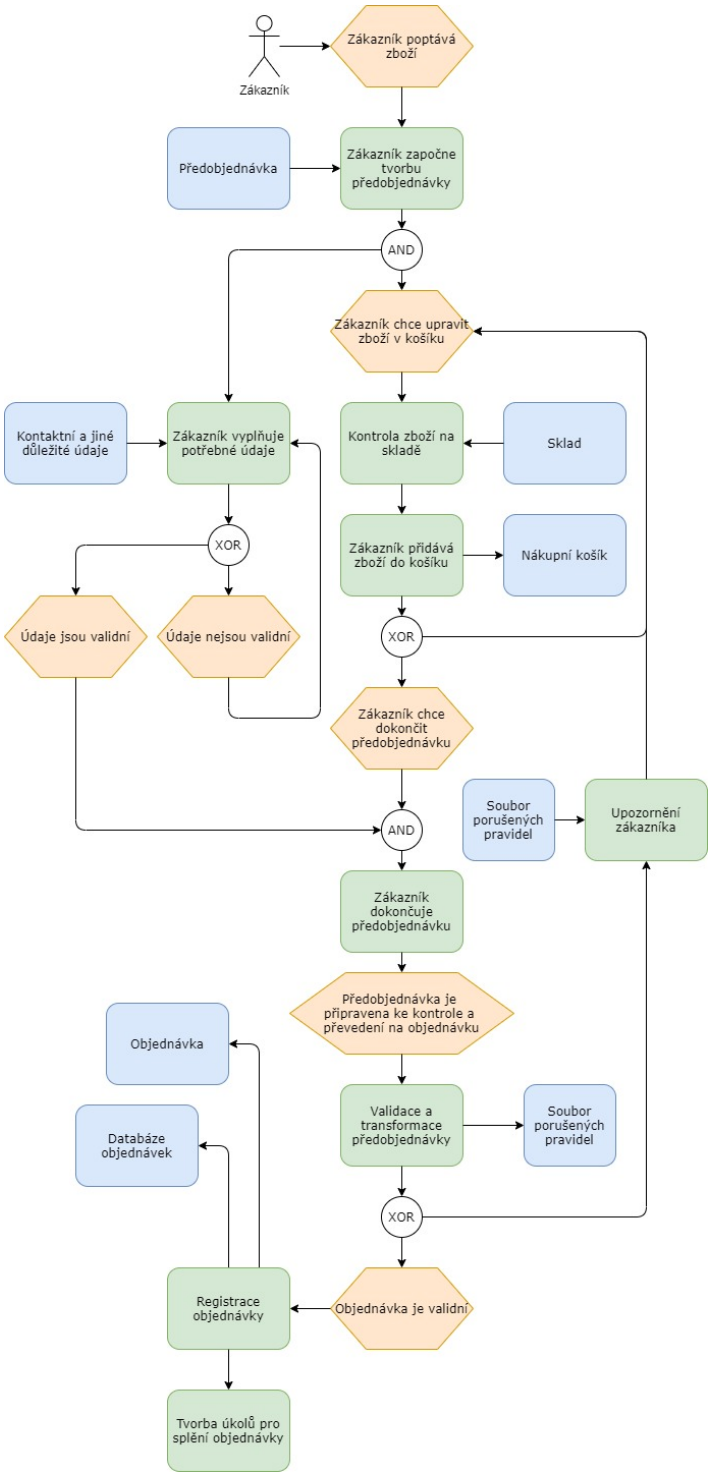
### 4.4.2 Objednávka je vyhotovena

V momentě vyhotovení objednávky je nutné, aby byl zákazník upozorněn. Předpokládejme, že ji vždy zaplatí a vždy vyzvedne.[9] Na diagramu *Proces události „Objednávka je vyhotovena“*

[7] V souladu s poučkou „Co nelze měřit, není možné ani řídit“

[8] Zásadní dopad tento akt má i například z pozice účetní, kdy vzniká povinnost (a přirozeně i nárok) až v momentě závazného objednání. Dále vytváří živnou půdu i pro rozšíření konceptu o „objednání se zpožděným dodáním“, tedy objednání ve smyslu, že zákazník by chtěl tuto objednávku realizovat v budoucnu.

[9] V opačném případě je nutné řešit problém ad hoc, a to například odstraněním objednávky z evidence či opětovné kontaktování zadavatele - zákazníka. Přirozeně řešení nestandardních situací závisí na podmínkách,

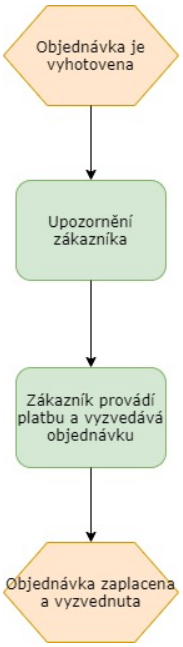



Obrázek 4.2: Upravený proces události „Zákazník vytvoří objednávku“

je tento proces znázorněn.

Kontaktování zadavatele může probíhat více způsoby - pomocí:

které si stanoví podnik. V návrhu by však k tomuto faktoru mělo být přihlédnuto a v implementaci by měl být připraven defaultní soubor administrátorských nástrojů pro úpravu aktuálního (a případně nepříznivého) stavu.




Obrázek 4.3: Proces události „Objednávka je vyhotovena“

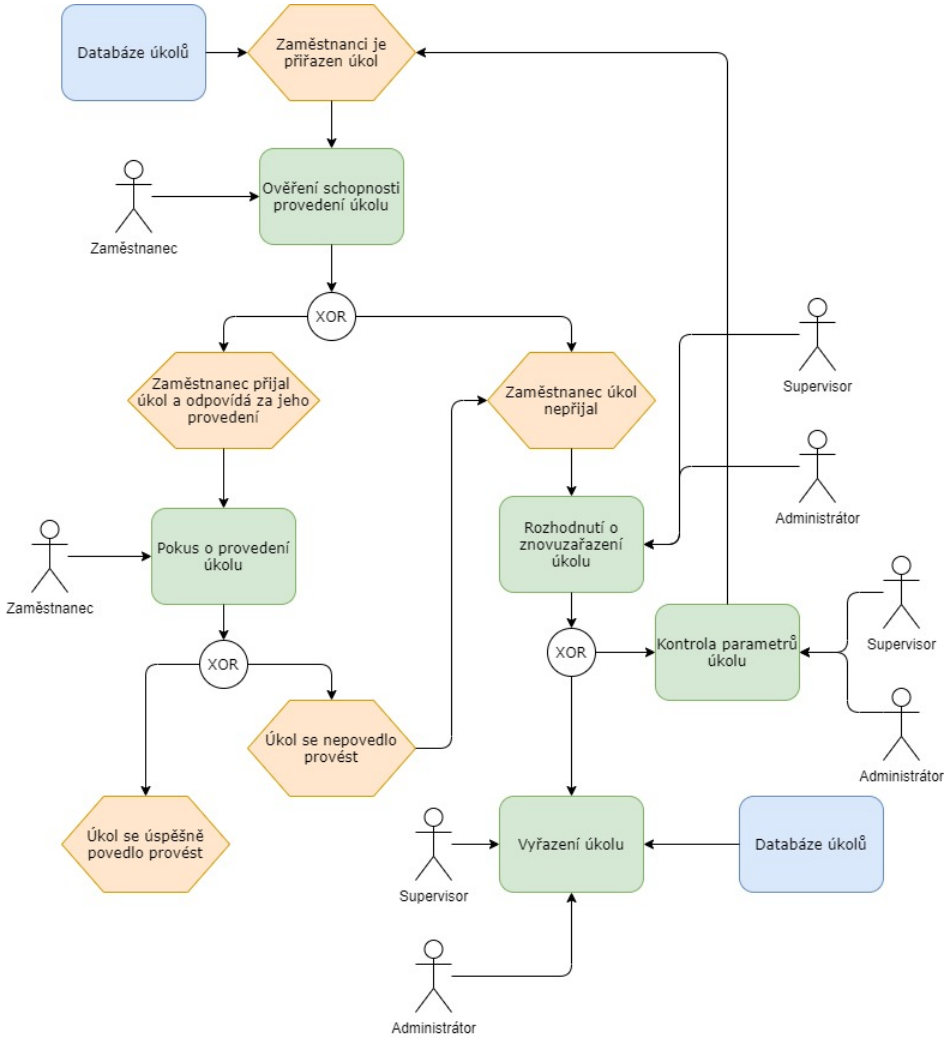
- **SMS zprávy** - upozornění pomocí zprávy v mobilu. Zákazník by měl k tomuto účelu zadat telefonní číslo. Výhodou je prakticky okamžité upozornění zákazníka o aktuálním stavu. Negativní vlivy jsou, že zákazníkovi není možné předat žádný propagační materiál[10], dále omezený rozsah zprávy, placený přístup k této technologii a nutnost dalšího zabezpečení.
- **Webové aplikace** - uživatel zadá kód objednávky do aplikace, aby viděl průběh vyhotovování. K tomu však je třeba implementovat sdílení těchto dat pro tuto aplikaci. Lze využít jako vhodné médium pro propagaci vlastní značky či k zlepšení služeb zákazníkům.
- **Objednávka na jméno** - uživatel se prokazuje svými personáliemi. To však může být za jistých okolností v rozporu s literou zákona, konkrétně pak s oblastí ochrany osobních údajů[11]. Dále musíme uvažovat i problém s unikátností identifikátoru. Jméno a příjmení není jednoznačným identifikátorem, zvláště pak v oblastech, ve kterých uvažujeme například 200 osob najednou.
- **Email** - notifikace pomocí emailu. Jednoduchý nosič dat, využití lze považovat za beznákladové. Zároveň jeho nasazení je snadné. Při vyvarování se zneužití emailové adresy k různým druhům nekalé soutěže či k porušování jiných právních předpisů, lze tento považovat za naprosto regulerní řešení. Nevýhodou je možná pomalá odezva. Chápeme, že doba mezi vytvořením objednávky a upozorněním zákazníka o jejím vyhotovení se může být záležitostí minut.

Při zhodnocení kladů a negativ jednotlivých alternativ, pro práci je nejvhodnějším řešením forma emailové komunikace. V budoucnu by bylo možné propojit tuto metodu s webovou aplikací pomocí unikátního odkazu.

[10] Myšleno například logo prodejce
[11] viz podkapitola *Vnější makroprostředí* na straně *27*.

### 4.4.3 Zaměstnanci byl přiřazen úkol

V momentě vytvoření objednávky je vhodné ji zpracovat a vytvořit k ní spjaté úkoly. Tyto úkoly jsou následně přerozdělovány mezi aktivní zaměstnance. Každý zaměstnanec prochází při přidělení úkolu hned dvěma po sobě jdoucími rozhodovacími procesy (viz *Proces události „Zaměstnanci byl přiřazen úkol“*. Nejdříve uvažuje, zda je možné splnit další úkol s ohledem na kvantum mu již přidělených úkolů. Při kladném rozhodnutí postupuje do druhého, kdy při plnění úkolu porovnává, zda je možné úkol splnit[12]. Tyto jeho kroky ověřuje (myšleno sleduje) administrátor a supervisor - v případě negativního rozhodnutí mohou poupravit parametry úkolu (a znovu zařadit úkol k přidělení), či úkol jako takový vyřadit a prohlásit ho za nesplnitelný.

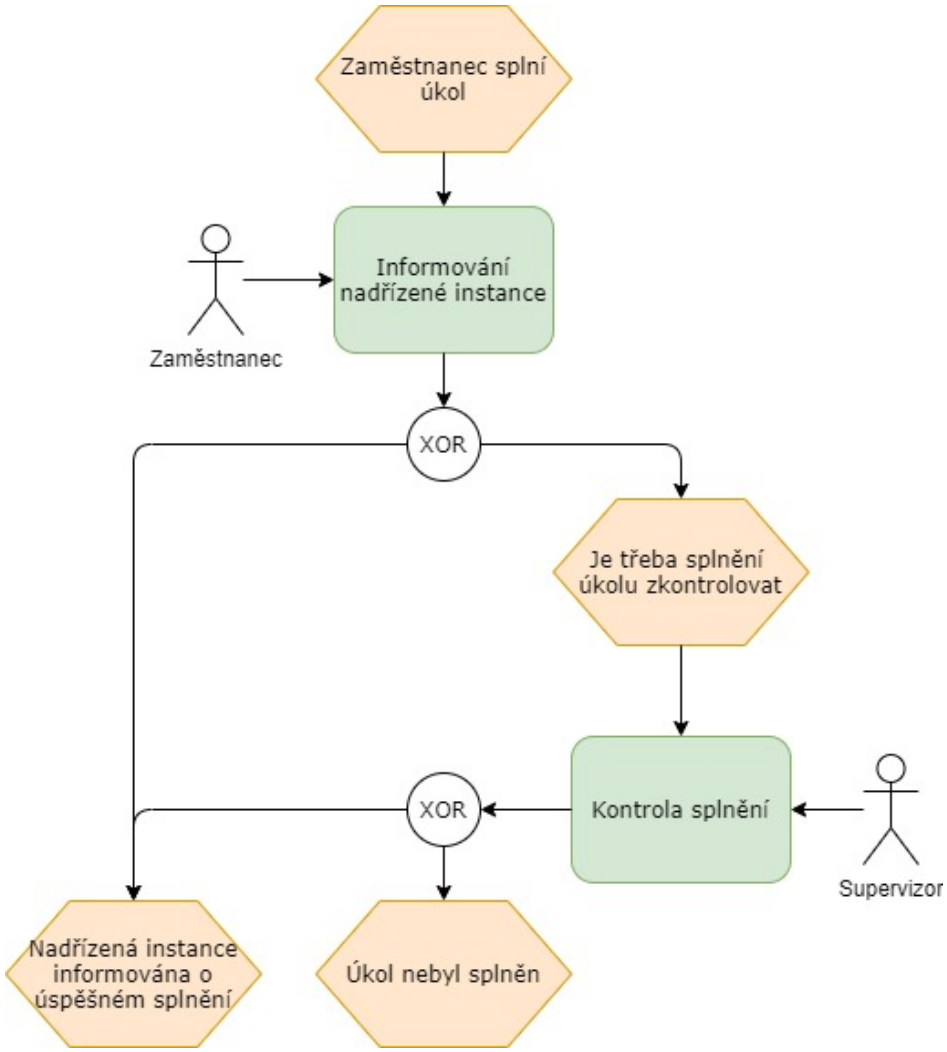



Obrázek 4.4: Proces události „Zaměstnanci byl přiřazen úkol“

[12]Například při přípravě zboží narazí na situaci, kdy není dostatek produktů na skladě.

### 4.4.4 Zaměstnanec splnil úkol

V momentě, že zaměstnanec splní úkol, musí být řídící orgán (supervisor) o této skutečnosti informován. Ten poté zkontroluje, zda ostatní úkoly k objednávce spjaté jsou také splněny. Především však kontroluje, zda byl úkol doopravdy splněn v dané kvalitě a způsobem, jakým měl - nemusí být však podmínkou, viz diagram *Proces události „Zaměstnanec splnil úkol“*.

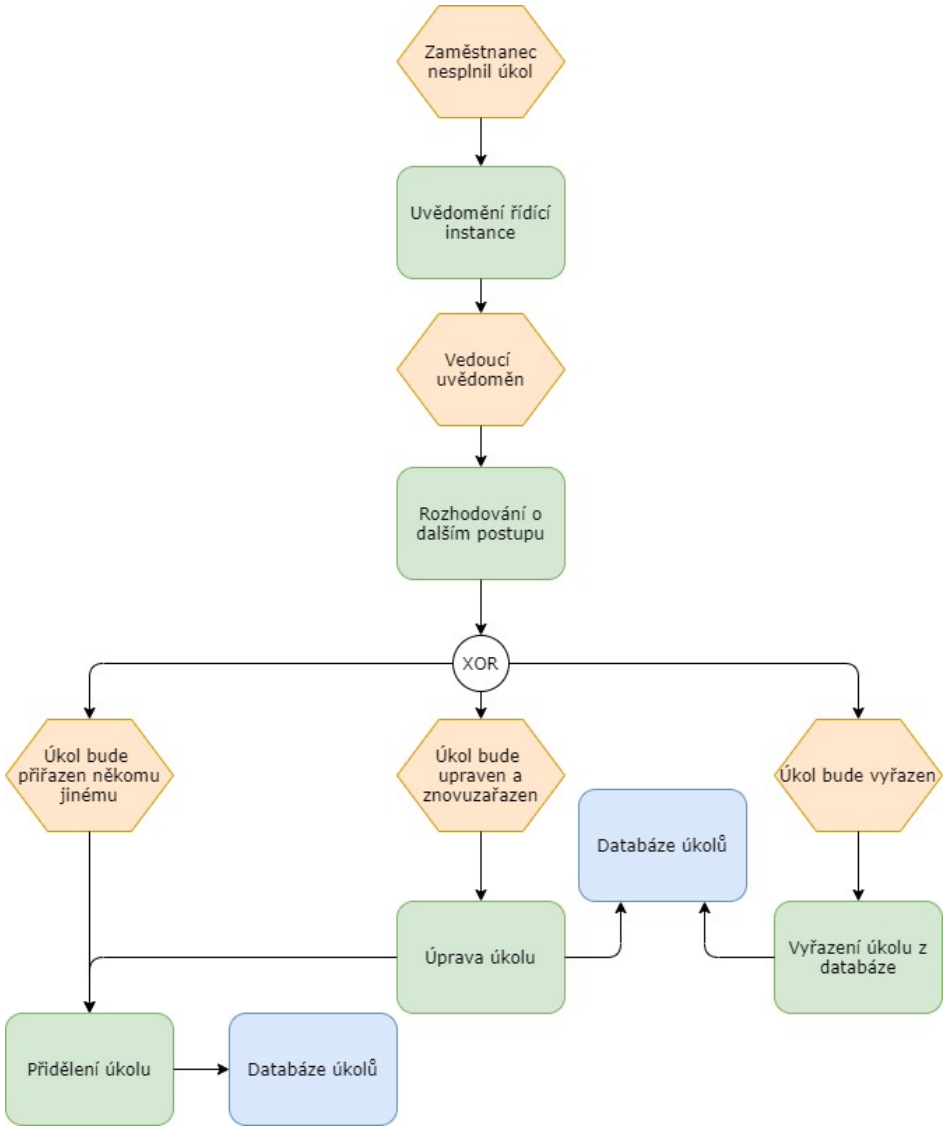



Obrázek 4.5: Proces události „Zaměstnanec splnil úkol“

### 4.4.5 Zaměstnanec nesplnil úkol

V systému musíme uvažovat i situaci, kdy zaměstnanec nesplní úkol, a to z libovolného důvodu. Například může dojít na situaci, kdy systém nesprávnou manipulací eviduje na skladě jiné množství (respektive *nějaké* množství), zatímco v realitě skladník zjistí, že již žádné zboží na skladě není. V ten moment musí řešit situaci operativně a obeznámit o tom svého nadřízeného (respektive osobu s dostatečnou kompetencí toto řešit). Na té pak závisí rozhodnutí založené na povaze problému o vyřazení úkolu z databáze, jeho úpravě či o pouhém předání úkolu jinému pracovníkovi.

Proces je graficky zpracován v jazyce EPC do diagramu na obrázku *4.6 Proces události „Zaměstnanec nesplnil úkol“*.

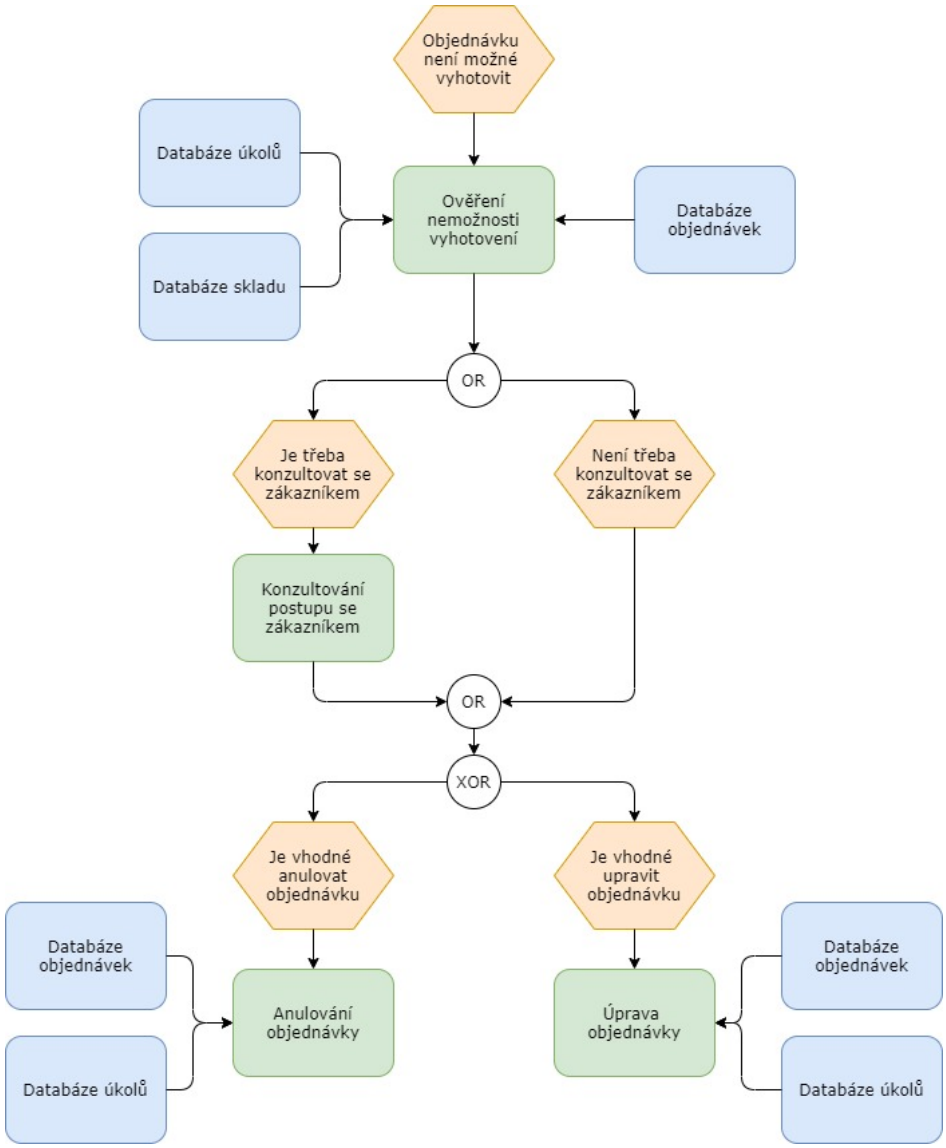



Obrázek 4.6: Proces události „Zaměstnanec nesplnil úkol“

V případě vyřazení úkolu by bylo možné napojit automatické upozornění zákazníka na změnu objednávky, případně kontaktování dodavatele o dodání chybějícího zboží, upozornění administrátora či přidělení úkolu o inventuře zboží některému ze zaměstnanců. Pro potřeby práce však toto nutně nepovažujme.

### 4.4.6 Objednávku není možné splnit

Dále chápejme, že může nastat v systému i situace, kdy objednávku není možné splnit. Taková může nastat například v nedostatku zboží na skladě či kritickém selhání systému. Proto autor navrhuje systém řešení takových situací pomocí diagramu procesu *4.7 Proces události „Objednávku není možné vyhotovit“*.




Obrázek 4.7: Proces události „Objednávku není možné vyhotovit“

Na tomto diagramu je patrné, že je proces sestaven s poměrně výrazným důrazem na ad hoc řešení - nelze kategorizovat přímo objednávky dle jejich nutnosti konzultace se zákazníkem v případě objevení problémů. Podniku musí být umožněno řešení dle svých vlastních preferencí. Z objektivního hlediska je přijatelnou alternativou jak dokončení objednávky se změnou (například nedodání všeho požadovaného zboží v poptávané kvantitě), odmítnutí celé objednávky, tak případná změna po konzultaci se zákazníkem (například přidání substitučního produktu).

Obecně lze tedy vznik tohoto stavu popsat u objednávky, u které je alespoň jeden úkol prohlášen za nesplnitelný a nadřízenou instancí je potvrzena jeho nesplnitelnost - úkol je vyřazen ze seznamu úkolů přiřazených k objednávce.

### 4.4.7 Model VAC

Model VAC autor využil pro znázornění propojení procesů. V této notaci lze tyto procesy kompilovat do kompaktního celku, z něhož jasně vyplývá vztah modelovaných procesů dle jejich vzájemné provázanosti. Notace je postavena na konceptu šipek jdoucích zleva doprava ve směru návaznosti procesu, kde v pravé části modelu chápeme zákazníka.[6, strana 320]

Obecně platí, že model zleva doprava udržuje směr úspěšného scénáře, případné odchylky jsou znázorněny okolo hlavní (nejdelší) linie. Tyto vedlejší subscénáře nemusí na sebe nutně navazovat, mohou znamenat pouze odchylku od původní sekvence procesů hlavního proudu, či jeho předčasné ukončení[13].

V tomto modelu *4.8 Model přidané hodnoty* jsou seřazeny procesy tak, jak autor chápe jejich řazení v business logice podniku.


Obrázek 4.8: Model přidané hodnoty

[13] To přirozeně závisí na samotném modelovaném procesu

# 5. Specifikace řešení

V této kapitole se hlouběji podívejme na rámec problému, jehož řešení je smyslem této práce, s cílem vybrat základní model z možných dostupných tak, aby potřeby systému naplňoval co možná nejvhodněji, a to v návaznosti na předchozí kapitolu.

Ze všeho nejdříve se podíváme na rámec celého řešení, následně na vhodné alternativy, pomocí kterých problém řešit a na závěr autor uvádí výběr samotné technologie.

## 5.1 Analýza rámce

V dosavadním řešení jsme seznali, že je nutné zajistit v první řadě komunikaci mezi více aktéry najednou. Pro plynulý chod podniku je nutné, aby mohli vykonávat své role paralelně. Uvažujme tedy, že každý k tomu má konkrétní prostředky (například své vlastní zařízení, pomocí kterého bude schopen se připojit do systému).

Podíváme-li se na dosavadní poptávané řešení, je zřejmé, že funkcionalita celého systému je založena na poměrně komplexní komunikaci. Je tedy na místě uvážit požadavky na systém zvnějšku. Naším rámcem z pohledu podniku pak tedy může být, že chceme, aby zákazník nepřímo komunikoval se zaměstnancem, respektive skupinou zaměstnanců, a to za použití výpočetní techniky. Tímto de facto dostáváme zadání souboru navzájem komunikujících aplikací s rozdílným cílem, způsobem užití a i jiným smyslem. **Neopomenutelnou součástí se také stává potřeba zabezpečnení.**

Zadáním pak je propojení několika různých zařízení tak, aby spolu byla schopna komunikace, přenosu dat a zpracovávání informací konkrétní vrstvě náležící. Takto lze víceméně popsat aplikaci řešení **klient-server** interakce.

Jako cíl tedy chápejme centralizaci digitálních zdrojů, rozvrstvení odpovědnosti a to vše tak, aby bylo možné komunikovat v síti s centralizovanými prvky - servery.

## 5.2 Specifikace poptávaných funkcionalit

V této kapitole si rozebereme funkce, které očekáváme, že bude možné provádět v rámci příslušné role.

### 5.2.1 Business logika

To jsou takové funkce, které lze považovat za nutné pro naplňování strategie podniku.

- **Zákazník**
  - Přidat produkt do košíku v daném množství
  - Odebrat produkt z košíku
  - Přidat kontaktní údaj pro potvrzení objednávky
  - Přidat poznámku k objednávce
  - Potvrdit objednávku
- **Skladník**
  - Přijetí úkolu
  - Odmítnutí úkolu
  - Odmítnutí úkolu z důvodu nemožnosti úkol splnit
  - Přestat naslouchat přidělování úkolů
  - Začít naslouchat přidělování úkolů
  - Dokončit úkol
- **Admin/Supervisor**
  - Přidat zaměstnance
  - Upravit zaměstnance
  - Upravit objednávku
  - Odebrat objednávku
  - Odebrat úkol
  - Upravit úkol

### 5.2.2 Uživatelská logika

Tato logika se stará o interakci mezi uživatelem a zbytkem systému.

- **Zaměstnanec**
  - Zobrazit svůj košík
  - Zobrazit všechny nabízené produkty
  - Vyhledávat v produktech
- **Skladník**
  - Zobrazit své úkoly
  - Řídit své úkoly
- **Admin/Supervisor**
  - Zobrazit aktuální stavy skladníků a jejich úkolů
  - Zobrazit aktuální stavy objednávek
  - Zobrazit aktuální stavy skladu

### 5.2.3 Strukturální funkcionality

Strukturální funkcionality můžeme popsat jako takové, které umožňují průběh business logiky a uživatelských funkcionalit.

Naším předpokladem je, že poptávané řešení je jistou formou klient-server aplikace, tedy souborem souběžně běžících programů se schopností mezi sebou komunikovat a to přes síť - i na velkou vzdálenost. Musíme tedy uvážit fakt, že je třeba zajistit bezpečnost v systému, který modelujeme.

Tím v tomto systému chápejme jistotu, že každé zařízení, které nějakým způsobem komunikuje, přesně a s jistotou ví, kdo je na druhé straně spojení. K tomu můžeme využít autentizaci a jednu z jejích tří metod, tedy ověření toho:

- **co subjekt má**
- **co subjekt zná**
- **čím subjekt je**[24]

Cílem práce je zaměřit se na první dvě, tedy **co subjekt má** (například ověření zařízení) a **co subjekt zná** (například kombinaci přihlašovacího jména a hesla). Třetí zmíněná se zabývá například biometrickými údaji a není smyslem práce tuto implementovat.

## 5.3 Rozbor možností a alternativ řešení

### 5.3.1 Topologie systému

Jak bylo naznačeno v sekci *Analýza rámce*, jedná se co do hypotetické poptávky o soubor spolu kooperujících aplikací skrz síť zařízení.

Samotný **klient-server** přístup poskytuje tři hlavní způsoby komunikace mezi zařízeními:

1. **Monolitická topologie** - Uživatel přistupuje k serveru pomocí terminálu bez vlastní logiky, jenž by rozuměla datovým tokům skrz něj procházející.
2. **Čistý klient-server** - Uživatel pracuje s klientskou aplikací, která poptává odpovědnost za provedení služeb konkrétního služebníka (server).
3. **P2P**[1]

Pro potřeby práce je první prakticky nepoužitelná, neboť neumožňuje prostor pro paralelní vykonávání své role v rámci podniku. V úvahu pak připadají přístupy 2 a 3.

Druhý je snazší na implementaci, neboť direktivně stanoví, co, jak a kdy bude provedeno. Vidíme zde ovšem riziko - přestane-li správně fungovat tento server, celý systém není schopen chodu. vedle toho P2P poskytuje mnohem vyšší bezpečnost (co do pravděpodobnosti vyřazení z provozu celého systému), nicméně jeho implementace je mnohem složitější.

Ideální by tedy bylo „zprůměrovat“ tyto dvě technologie - vytvořit systém, kde jsou uzly,

[1] *„Peer-to-Peer“*, tedy přístup, kdy jsou si všichni rovni. Tento předpoklad pramení z možnosti, kdy oba druhy uzlů (tedy klientská aplikace i serverová logika) může vstupovat v roli toho, kdo poptává funkcionalitu, i toho, kdo ji poskytuje.

které zastávají čistě direktivní autoritu (čistě jen server), uzly, které zastávají roli obojí (klient i server) a ty, které zastávají roli pouze klienta.

Tento koncept trojjedinosti volí autor z toho důvodu, že je třeba (kumulativně):

- **Komunikovat v rámci business logiky** - provádět operace v souladu s podnikem
- **Zajistit bezpečnost této komunikace** - komunikace týkající se zajištění bezpečnosti business logiky

Toho lze dosáhnout způsobem zvolení autority, která bude sloužit výhradně jako autentizační autorita. Prohlašme tedy tuto jako **Notářskou službu**[2].

### 5.3.2 Výběr hlavního jazyka

Pro potřeby psaní kódu hlavních částí aplikací se objevuje mnoho různých alternativ pro vytvoření dostatečných řešení problému. Výsledný kód musí být schopen umělého prodlužování svého životního cyklu (drobnými) úpravami tak, aby vyhovoval logice byznysu co možná nejlépe.

Dnes se používá mnoho různých přístupů k architektuře kódu a ještě více jazyků. **TIOBE index**[25] nám nabízí slušnou představu o využití programovacích jazyků napříč celým spektrem užití. Tabulka *5.1 Tabulková reprezentace TIOBE indexu[25]* nám ukazuje pořadí programovacích jazyků v rámci analýzy srovnání jejich popularity dle vyhledávání v internetových vyhledávačích. V této tabulce je zahrnuto hned několik jazyků s hned několika možnými využitími. Jsou zde zastoupeny vysokoúrovňové programovací jazyky (například Java či C#), nízkoúrovňové jazyky (například C či C++), skriptovací jazyky (např. JavaScript, R), či jazyky s velmi specifickým užitím (SQL). V uvedené tabulce jsou zvýrazněny možné alternativy, ze kterých je vhodné vybrat jeden takový (nebo jejich kombinaci), jenž by naplňoval potřeby tvořeného systému pokud možno co nejlépe.

Neboť je cílem systém s vysokou variabilitou, snadnou údržbou a se snadným pochopením programátora, za tento standard lze považovat ze zúženého výběru jazyky **Java** a **C#**. Kupříkladu JavaScript (včetně jeho frameworků a dalších rozšíření) nemá tak významný smysl pro architekturu, která je těmto dvěma nativním předpokladem. Java v porovnání se C# jsou si velmi podobné jazyky - oba nativně podporují **OOP** paradigma, oba jsou multiparadigmatické (tzv. hybridní) a oba jsou staticky typované[3]. Java však oproti C# vede v rozumné použitelnosti na nespočtu různých platforem, což je další kritický faktor v rozhodování.

[2] [26, strana 90] je popsána jako autorita stvrzující dokumenty. Pro naše potřeby tyto prohlašme jako přenášené zprávy obecně. Každá zpráva sice nemusí nutně procházet přes tuto službu (to by u větších datových toků mohlo být značně redundantní), ale měla by být krom vlastního obsahu i nosičem jistého oprávnění o komunikaci.

[3] Z toho plyne sice nepatrně více práce pro programátora, ale odchytí se mnoho chyb ještě před spuštěním programu. Dále pak je program stabilnější, neboť statickým přístupem (oproti tomu dynamickému) se předchází nestandardnímu chování programu.

| **Pořadí** | **Název jazyka** | **Rating** |
|---|---|---|
| **1** | **Java** | **17.358%** |
| 2 | C | 16.766% |
| **3** | **Python** | **9.345%** |
| 4 | C++ | 6.164% |
| **5** | **C#** | **5.927%** |
| 6 | Visual Basic .NET | 5.862% |
| **7** | **JavaScript** | **2.060%** |
| 8 | PHP | 2.018% |
| 9 | SQL | 1.526% |
| 10 | Swift | 1.460% |
| 11 | Go | 1.131% |
| 12 | Assembly language | 1.111% |
| 13 | R | 1.005% |
| 14 | D | 0.917% |
| 15 | Ruby | 0.844% |
| 16 | MATLAB | 0.794% |
| 17 | PL/SQL | 0.764% |
| 18 | Delphi/Object Pascal | 0.748% |
| 19 | Perl | 0.697% |
| 20 | Objective-C | 0.688% |

Tabulka 5.1: Tabulková reprezentace TIOBE indexu[25]

### 5.3.3 Výběr jazyků pro periferní úkoly

Periferními úkoly chápejme takové úkoly, které nejsou primárním cílem ani business logiky, ani nutně infrastrukturních vztahů mezi uzly v síti.

Příkladem nám může být kód databáze. Pro komunikaci s databází se užívá standardně jazyka SQL, stejně tak jako pro definování vnitřní struktury a vztahů[4].

Pro potřeby práce uvažujme relační databázi určenou pro persistenci dat s podporou SQL[5].

Další oblastí pak je konfigurace. Dnes se užívají především dva druhy konfigurace programu, a to pomcí **konfiguračních souborů** a **přes rozhraní**[6]. V situaci, kdy bychom chtěli konfigurovat pomocí první zvolené metody, vyvstávají zde dva hlavní směry - **s pevnou strukturou** a **bez pevné struktury**. Ty s pevnou strukturou jsou obvykle zastoupeny jazykem XML či JSON, ty bez pevné struktury pak jednoduchým programovacím jazykem, který de facto předpřipraví zdroje pro spuštění hlavní aplikace.

Pro potřeby naší práce zvolme formu XML pro jeho univerzálnost, poměrně dobrou čitenost, podporu mnoharozměrných dat a snadnou kontrolu[7].

Další jazyk je (pro výběr jazyka Javy) i jazyk pro definici GUI (grafického rozhraní). Je přirozeně možné užít standardních knihoven Swing, ovšem od nich se upouští a dává se přednost JavaFX. Tento jazyk sice také podporuje tvorbu a vykreslování grafického rozhraní bez použití externích konfiguračních souborů, kód se však stává silně těžkopádným. Pro tyto potřeby pak volí autor jazyk *FXML* jako v podstatě univerzální definici rozložení grafiky pro ovládání aplikace založeným na jazyce XML[34].[8]

[4] Mluvě o standardním **relačním** databázovém modelu založeném na SQL. Alternativně by bylo možné užít **NoSQL** databáze, v zastoupení například **MongoDB**.

[5] viz podkapitola *6.6 Databáze* na straně *71*

[6] například grafické

[7] pomocí XSD - schémat definujících strukturu souboru

[8] viz kapitola *8 Grafické rozhraní* na straně *83*

# 6. Návrh řešení

V této kapitole se podívejme na hlavní aspekty řešení problému. Rozvineme tedy potřebnou problematiku z předchozích kapitol a vydefinujeme a rozebereme si aspekty, u nichž lze prohlásit, že jsou stěžejními.

## 6.1 Rozdělení do modulů

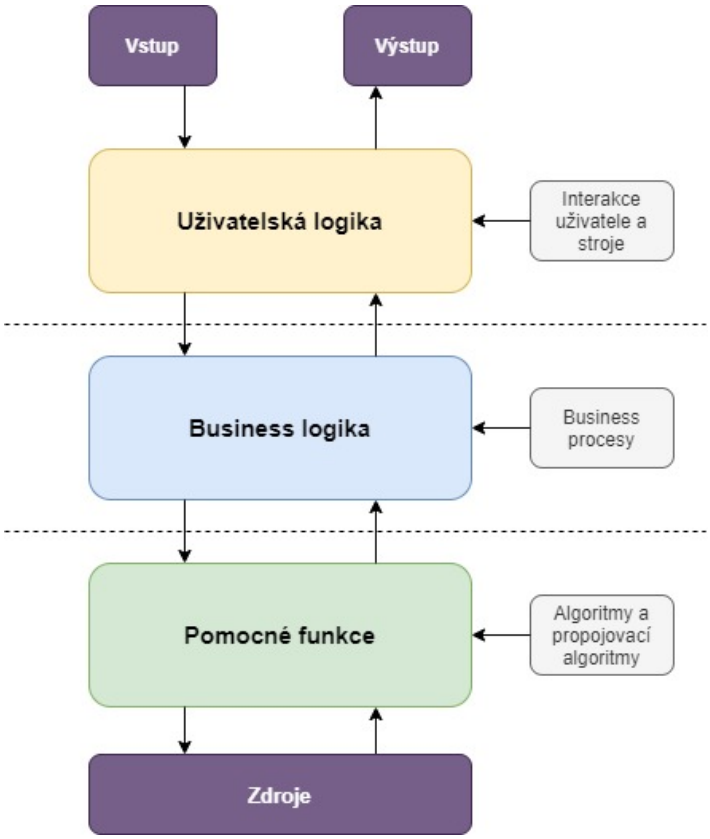
Pokud-že chceme tvořit informační systém, musíme správně pochopit nejdříve logiku byznysu. Poté teprve můžeme fakticky nahlédnout pod pokličku toho, co by měl dělat informační systém a u jakých procesů by měl být zodpovědný, za jejich efektivnější provedení. Autoři knihy *Tvobra informačních systémů: principy, metodiky, architektury* popisují informační systém dle principu **BITA**, tedy **Business IT alignment**. V podstatě je zde popsána **nutná** korelace mezi strategií byznysu a informační strategií.[6] V obou případech (po značném zjednodušení) můžeme dojít k poznatku, že každá strategie v jakémkoliv oboru se snaží o maximalizaci efektivity, tedy využití vstupů tak efektivně, aby agregované výstupy byly pokud možno co nejvyšší. Můžeme tedy (alespoň pro naše potřeby) chápat tuto vlastnost jako predikát každé uváženíhodné strategie. Informační systém pak chápejme jako spojovník či rozhraní těchto dvou strategií.

Vyvstávají zde dvě alternativy, jak tohoto spojovníku dosáhnout:

- **Kompletní propojení (merge) obou strategií** - Velmi poutavé řešení, neboť nevyžaduje nijak zvláštní podmínky pro tvorbu rozhraní, tedy IS. Lze dosáhnout velmi rychlého vývoje systému, vývojáři stačí přemýšlet a uvažovat jen na jedné jediné úrovni. To s sebou ovšem nese neopomenutelná úskalí:
  - Takovýto systém (nezávisle na své komplexitě a robustnosti) je těžkopádný, jakákoliv úprava je náročná časově (a tedy i finančně)
  - Rychlost vývoje je vykoupena neefektivní správou
  - Životní cyklus IS je prakticky nemožné uměle prodlužovat
  - Velmi úzká provázanost byznysu a informatiky vede k jednoúčelovosti. Rozšíření je přinejmenším velmi náročné.

- **Provedení víceúrovňového propojení strategií** - Tento způsob řešení má oproti první alternativě nevýhodu v náročném a zdlouhavém vývoji. Existuje zde riziko, že ještě před nasazením systému dojde k jeho zastarávání. Kvůli silně rigorózním přístupu je náročné přistupovat k vývoji agilně.
  Ovšem tyto nevýhody jsou vykoupeny tím, že je systém vytvářen se silným důrazem na dodržování vrstev, čímž v případě změny libovolné strategie je nutné něco měnit maximálně na jedné polovině řetězce. Tento systém je jednak v souladu s pilířem o znovupoužitelnosti a univerzalitě[1], tak i v souladu s poznatkem o nedostatečnosti rozšiřitelnosti stávajících řešení[2]

[1] stanoveným v kapitole *2.1 Stanovení hlavních pilířů práce.*
[2] zjištěným z konzultace s osobou P. v příloze *Záznamy z odborných konzultací.*




Obrázek 6.1: Vztah mezi uživatelem, byznys logikou a hlubokými strukturami

Na obrázku *6.1 Vztah mezi uživatelem, byznys logikou a hlubokými strukturami* můžeme pozorovat vazby mezi jednotlivými úrovněmi přechodu. Procházíme-li odshora, všimneme si, že do systému máme nějaké vstupy a výstupy. Cílem **uživatelské logiky** je tyto zpracovávat a předávat uživateli. Přímo komunikuje s **byznys logikou**, jenž se stará o naplňování business procesů. Ta opět reaguje s **pomocnými funkcemi**, hlubokými strukturami systému, a s různými zdroji - databáze, kooperace IS a OS, atd.

### 6.1.1 Infrastrukturní hierarchie

Pro splnění zásady *Znovupoužitelnost a univerzalita* z kapitoly *Stanovení hlavních pilířů práce* musíme apelovat na silnou hierarchii v architektuře celého výsledného systému. Jelikož je naším cílem systém, který bude mít možnost umělého prodlužování svého životního cyklu drobnými úpravami, musíme nutně oddělit různé funkcionality a vytvářet příslušná rozhraní tak, aby bylo dosaženo co nejvyšší modularity a variability. Toho lze dosáhnout třízením těchto funkcionalit do bloků s podobnou tématikou a dle podobné **infrastrukturní návaznosti**[3]

Vydefinujme si tedy předpoklad, že můžeme různé funkcionality (na sobě v nějakém kontextu závislé) rozřadit do modulů tak, že jedna je infrastrukturou druhé, která je infrastrukturou další, čímž nám vzniká řetězec vzájemně provázaných infrastrukturních vztahů, kdy můžeme uplatnit pravidlo

[3]tzn. seřazení těchto bloků dle návaznosti - navzájem jsou na svých součástech závislé, proto ty nejméně závislé by bylo vhodné zařadit co nejníže, vedle toho ty spadající do business logiky a uživatelského rozhraní se snažit vtěsnat naopak co nejvýše, neboť pracujeme s předpokladem, že tyto funkcionality budou nejvíce závislé na ostatních.

**tranzitivity závislosti**.[4] Zároveň platí, že funkcionalita nadřazená nemusí (a pro zachování pevné architektury ani nesmí) rozumět procesům, které probíhají na nižší úrovni. V případě potřeby obměny nižších funkcionalit by tedy měly mít příslušné **rozhraní** nastavené tak, aby vyšší struktury poptávaly provedení úkonu s pevně stanovenými vstupy a výstupy.[5]

Musíme však zajistit, aby nedocházelo v tomto řetězci závislostí k cyklickým vztahům[6]. Proto možnými dvěma řešeními pro budování architektury se zdá být **strom** symbolizující vztahy funkcionalit, či idealistická jediná **linearní** vazba.

### 6.1.2 Utility

Začneme-li od těch nejnižších úrovní, uvažujeme tedy přímou komunikaci systému se strojem. Neboť by daná implementace měla být co nejvíce univerzální, musíme využít silně parametrického prostředí. Zároveň platí, že tato část by měla být velmi pečlivě testována, měla by být algoritmicky velmi efektivní a neměla by být závislá na ničem jiném, než je samotný stroj, operační systém, případně vstupní data.

Tato rozhraní a tyto funkcionality můžeme obecně pojmenovat standardním slovem **Utility**, či v této práci používané **Pomocné funkce**. Za ty považujeme takové funkce, které pracují na velmi abstraktní úrovni a nevyžadují více, než je popsáno výše. Sem bychom mohli zařadit následující funkcionality:

- **Základy práce s XML** - Pro zapisování, čtení, úpravu, dotazování a validaci dokumentů. Je zde nutné uvažovat také to, že v rámci architektury by měly být sice tyto funkce definovány nejníže, neboť nejsou business logikou, uživatelskou, ani tou funkční. Nicméně implementace závisí na vyšších entitách, je tedy vhodné v pomocných funkcích vydefinovat rozhraní, které bude společné pro ostatní, a oddělit ho tak od implementace, jenž bude vytvořena tam, kde jí bude třeba.

- **Kryptografie** - V rámci bezpečnosti je nutné implementovat alespoň základní kryptografické principy. Pro potřeby práce je vhodné implementovat alespoň hashové funkce s obměnitelným algoritmem[7]. Dále by tato funkce měla umět podporovat bezpečnostní přístupy k hashování. Prvním přístupem je užití soli (například pro ukládání hashů uživatelských hesel), či pepře.[8]

[4] Příkladně nám může posloužit běžná obytná stavba. Pro osobu bydlící v tomto stavení je spuštění vody z vodovodního kohoutku funkcí, která mu přímo náleží. Ovšem pro tuto funkci je infrastrukturou rozvod vody v trubkách zabudovaných ve stěnách, pro ty zase rozvody vodovodního potrbubí v obci, a takto můžeme pokračovat až k podzemním zásobám vody.

[5] Ilustrovat lze tento přístup na příkladu pracovní pozice. Máme-li pracovní pozici *(rozhraní)* s odpovědností za provedení nějakých úkonů, zaměstnavatele *(vyšší infrastrukturní entita)* ve skutečnosti nezajímá, jakým způsobem zaměstnanec *(konkrétní instance implementující rozhraní)* úkon provede, zajímá se pouze o to, zda zvládne provést stanovený úkon *(zda implementuje toto rozhraní)* s danými vstupními a výstupními parametry *(signatura + kontrakt)*. Lze tedy zaměstnat libovolnou osobu, která splňuje předpoklady definované pracovní pozicí *(implementace jednoho rozhraní vícero třídami entity)*

[6] Dostali bychom se do situace, kdy nižší struktura by vyžadovala strukturu vyšší a tím by se celá integrita (že každá vrstva je sama o sobě funkční) rozsypala.

[7] Uvažujme algoritmy z rodin MD a SHA, jmenovitě MD5, SHA1, SHA256, SHA348 a SHA512

[8] Užití soli i pepře je technika, kdy se k heslu před provedením hashování přidá další textový řetězec, celek je následně hashován. Tím je zajištěno, že byť pro identická hesla je (za předpokladu náhodně vygenerovaného řetězce) hash rozdílný. Rozdíl mezi solí a pepřem spočívá v tom, že sůl je po celou dobu procesu hashování známa. V příkladě s ověřením heslem, server porovnává hash hesla v kombinaci se solí kdy provádí jeden hash na iteraci. Vedle toho pepř chápejme jako náhodně vygenerovaný řetězec přidaný před hashováním. Server

- **Random generátory** - Generování náhodných hodnot pro různé účely.
  - **Náhodná čísla** - například číslo z daného intervalu
  - **Náhodné textové hodnoty** - například řetězce pro potřeby generování náhodných hesel
  - **Náhodné strukturované texty** - například náhodné emailové adresy pro potřeby testování či tvorba náhodných hesel se stanovenou formou (například pomocí regulárních výrazů).

- **Správce komunikace s operačním systémem** - příkladně zjišťování informací o síťovém rozhraní, zápis do souborů či jejich čtení.

Tyto služby je nutné psát (tedy kódovat) se silným apelem na jejich správnou funkci (tedy silně testovat) a zároveň zajistit jejich co nejvyšší algoritmickou efektivitu. Tyto drobné podprogramy slouží systému jako pevné stavební kameny, neočekávejme u nich významné změny a měly by proto být stabilní, rychlé a hlavně správné. Zároveň jsou to takové procesy, u nichž očekáváme případné obměny jen co do způsobu provedení, nikoliv nutně jejich vstupů a výstupů. Je tedy třeba důsledně dbát na architekturu a to pomocí rozhraní, abstrakce a víceúčelnosti a znovupoužitelnosti.

### 6.1.3 Správci síťové komunikace a business logika systému

Uvažujeme-li klient-server aplikaci (systém), musíme pracovat s propojením více zařízení. Abychom setrvali při práci v souladu se stanovenými pilíři práce, je vhodné tyto služby začlenit do níže uvedených sfér působnosti programu a nevázat business logiku již nutně na tuto implementaci, nýbrž této využít jako infrastrukturu business logice.

Cílem dalšího rozvrstvení je tedy tvorba univerzální platformy pro přenos prakticky libovolných zpráv a samotné zprávy. Za předpokladu, že tato platforma funguje tak, jak má, lze uvažovat, že v případě i zásadní změny v business logice není nutné nijak tuto platformu měnit.

Tvoříme-li spojení mezi více počítači pomocí jazyka **Java** a snažíme-li se vyhnout užití nadbytečné integrace frameworků, pravděpodobně užijeme tříd **Socket** a **ServerSocket**. Tyto třídy nám umožňují vytvořit (šifrovaný) kanál mezi dvěma zařízeními v síti a připravit instance tříd **InputStream** a **OutputStream** pro čtení vstupu a odesílání výstupu.

Zůstaneme-li poplatní tomu, že chceme užít **objektového rozhraní komunikace**, pak k tomu Java poskytuje například třídy **ObjectOutputStream** a **ObjectInputStream**.[9] Tyto třídy dovedou z obyčejných vstupních a výstupních datovodů skládat serializovatelné[10] objekty, s nimiž lze plnohodnotně pracovat. **Je však podmínkou, že třídy těchto objektů musí být v naprosto shodné podobě na obou stranách komunikace**. Pokud-že tato podmínka není splněna, vrací program

pak zkouší všechny kombinace pepře, dokud nenalezne odpovídající kombinaci. Tento způsob však způsobuje problém s exponenciálně rostoucími počty provedených hashů, samotné generování těchto znaků je pomocí algoritmu použitelného pro brute-force crackování hesel. Z toho vyplývá vzorec pro počet kombinací nutných hashů následovně: počet nutných vygenerovaných hashů $= n^m$, kde $n$ je velikost znakové sady a $m$ je maximální délka náhodných znaků. Při použití znakové sady o délce 62 a pepře o délce 5 znaků, počet testovaných hashů se pak rovná bezmála 920 milionům, přičemž o navýšené bezpečnosti oproti použití soli lze úspěšně polemizovat.

[9] Alternativními řešeními komunikace pro textové rozhraní pomocí příkazů by bylo možné využít kombinaci tříd **PrintWriter** pro výstup a **BufferedReared** pro vstup.[27]

[10] implementující rozhraní **Serializable**

chybu[11], což je pochopitelné, neboť program neví, jak s objektem nakládat a není způsob, jak mu to kulantně objasnit.

Dále v tomto souboru funkcionalit musíme uvažovat samotné zprávy, strukturu komunikačního protokolu a způsob, jakým spolu moduly komunikují obecně. Tyto jmenované oblasti lze nalézt v podkapitole *6.2 Klient-server řešení* na straně *59*.

V neposlední řadě je zde obsažena i business logika. Ta by měla být částečně provázána s komunikací na bázi socketů - uvažujeme, že systém komunikace by měl mít co nejmenší vliv na používání a tedy by implementované konstrukty pro podporu síťové komunikace měly být skryty. Toho lze dosáhnout například prohlášením této vrstvy za infrastrukturu business logiky.[12]

### 6.1.4 Grafické rozhraní

Jako grafické rozhraní chápejme nástroj pro efektivní vytváření příkazů určených aplikaci. Toho je obecně obvykle dosaženo pomocí ovládacích prvků zařazených do tzv. ovládacího panelu. Mezi základní ovládací prvky můžeme zařadit tlačítka (s různým způsobem použití), textová pole či seznamy hodnot.

Každá „sub-aplikace“ by měla mít vlastní ovládací panel a měl by být schopen ovládat prvky v rámci logiky případu užití. Hlavními kusy tohoto grafického rozhraní by pak měly být panely pro řízení objednávek a pro správu úkolů stanovených zaměstnanci. Pomocí těchto přehledových souborů ovládacích prvků by mělo být dosaženo možnosti zefektivnit prodej[13] či efektivně přerozdělit úkoly zaměstnancům[14].

Podrobněji se lze o této problematice dočíst v kapitole *8 Grafické rozhraní* na straně *83*

## 6.2 Klient-server řešení

Jak bylo již popsáno, řešením by měl být klient-server systém, tedy systém zařízení v síti propojených tak, aby bylo možné ovládat různé uzly z různých geografických pozic. Java umožňuje hned několik způsobů, jak tohoto dosáhnout, kdy umožňuje využít schopností tříd **Socket** nebo pomocí celého frameworku **Spring**. První zmíněné je určeno spíše menším rozměrům systému, vedle toho framework Spring disponuje hned několika možnými řešeními z mnoha úhlů pohledů.

### 6.2.1 Socket

Služby, které třída Socket (s příslušnou vedlejší implementací) poskytuje, jsou poměrně rozsáhlé. Umožňuje přenos dat mezi zařízeními (respektivě běžícími programy) po virtuální síti (pomocí TCP/IP

[11] respektive příslušnou vyjímku - ClassNotFoundException

[12] Formálně lze hovořit o umělém vytvoření další sub-vrstvy business logiky - viz model *6.1 Vztah mezi uživatelem, byznys logikou a hlubokými strukturami* na straně *56*

[13] místo hledání produktů v prodejně stačí zákazníkovi například vyhledávat v produktech ze seznamu skladových položek.

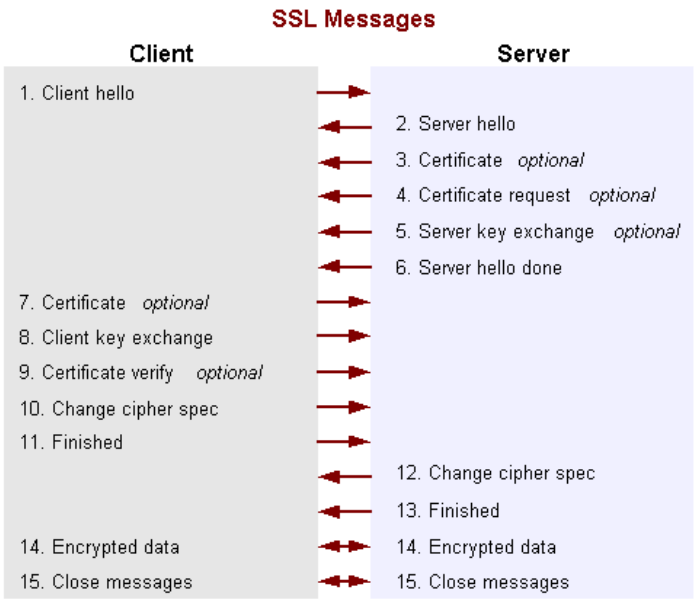
[14] Každý zaměstnanec by měl být schopen vidět všechny jemu udělené úkoly a těm přidělit úroveň splnění, viz proces *4.4, Proces události „Zaměstnanci byl přiřazen úkol“* na straně *44*

protokolu), a to ať prostých dat strukturovaných do pole bytů, čistý datový tok, text či samotné objekty.

Velkou výhodou je jeho poměrně široká možnost implementace a fakt, že je součástí „čisté“ Javy a není proto potřeba připouštět další knihovny a frameworky. Nevýhodou je pak složitější implementace, prakticky nutnost použití vláken a vlastní řešení bezpečnosti[15].

Jako sockety je vhodné použití socketů bezpečnějších, než jsou standardní. Tím v Javě myslíme SSLSockets[16], tedy sockety s nativní podporou SSL/TLS.[17]

Proces autentizace a následné komunikace lze rozdělit do souboru patnácti kroků v požadované sekvenci, jak je vidět na obrázku *6.2 Sequence of Messages Exchanged in SSL Handshake.*




Obrázek 6.2: Sequence of Messages Exchanged in SSL Handshake[29]

V práci autor použil **TLS** ve verzi 1.3, které poskytuje významně vyšší bezpečnost než jeho předchůdci, při nastavení povolených šifer `TLS_AES_128_GCM_SHA256` a `TLS_AES_256_GCM_SHA384`[18]

Toto spojení lze navázat jak pro použití v rámci lokální sítě (pomocí běžné, privátní IP adresy získané od DHCP serveru, tedy obvyklou součástí routeru), tak i v rámci komunikace s veřejnou částí sítě, tedy například i pro podporu HTTPS[28] v rámci komunikace s webovými servery.

[15]Bezpečnost v Javě lze řešit i pomocí jí příslušejících nativních knihoven. Jejich propojení však vyžaduje mnohem složitější přístup, než je tomu například u komponent frameworku **Spring**

[16]Odkaz na dokumentaci: `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/javax/net/ssl/SSLSocket.html`

[17]Autor si k těmto vytvořil keystore a truststore s certifikátem, který vygeneroval pomocí nástroje *keytool* v balíku Javy. Defaultní nastavené heslo k němu je "changeit"(bez uvozovek) a měl by být používán výhradně k testování a debugování.

[18]Pro jiné šifry se autorovi při TLS v1.3 nepovedlo dlouhodobě stabilní spojení navázat.

### 6.2.2 Spring

V rámci enterprise užití se framework Spring ukázal v kombinaci s Hibernate jako velmi mocný nástroj pro velmi rychlé vytváření aplikací. Dále pak snadno umožňuje tvorbu webových aplikací[19]. Tato nesporná výhoda je vykoupena nutností pochopit logiku celého frameworku, vazeb mezi jednotlivými komponentami a náročnějšímu dodržování architektury systému včetně požadavku na infrastrukturní nezávislost. Dále jde o další nadstavbu jazyka Java, není tedy splněn náš předpoklad o užití co nejjednodušších nástrojů pro snadnou správu.

### 6.2.3 Komunikační protokol

Uvážíme-li řešení pomocí socketů, musíme si navrhnout komunikační protokol, tedy způsob, jakým se budou zařízení v síti dorozumívat.

Jako přípustná řešení se zdají být následující:

1. **Textové rozhraní** - snadná implementace, minimální přenosová režie. Zato vysoké riziko prozrazení obsahu zprávy, nelze využít ochranných nástrojů při psaní kódu[20].
2. **Souborová komunikace** - Složitější implementace, vysoká přenosová režie, nutná implementace čistících nástrojů[21], zato snazší evidence a logování - vše stačí jen uložit na pevný disk. Formát dokumentů by mohl být například XML či JSON.
3. **Objektový přenost** - složitá implementace, zachování architektury, střední přenosová režie. Při komunikaci pomocí objektů lze dosáhnout (oproti ostatním způsobům) usnadnění implementace pomocí návrhových vzorů, zajištění bezpečnostní integrity a případné řešení problémů rozložit i přímo do daných zpráv. Obecně lze libovolnou funkcionalitu přímo připojit do implementace zprávy (návrhový vzor **Command**), což mimo jiné zásadně zpřehlední výsledný kód[22].

Díky těmto zjištěním pak autor vybral jako objekt komunikačního protokolu přenos objektový, tedy variantu číslo 3.

### 6.2.4 Návrh komunikace

Snažíme-li se vytvořit klient-server aplikaci, musíme uvážit i přístup, jakým mezi sebou budou zařízení komunikovat, tedy kdo, kdy a jak získá slovo a jak své právo „mluvit“ bude využívat.

Jak bylo popsáno na straně *51* v podkapitole *Topologie systému*, vidíme, že je třeba vytvořit komunikační systém pro umožnění ověřování zpráv. Zároveň zde byl elementárně popsán pojem **Notářská služba**, která předává strvrzení o oprávnění.

[19] Například v kombinaci s Tomcat

[20] Při kompilaci se u jiných druhů komunikace může velká část chyb odchytit - například překlepy. Zde to nelze zaručit, neboť stroj lidsky čitelným zprávám nerozumí.

[21] Buďto by se nesměl obsah souborů dlouhodobě evidovat nebo by se musel v případě potřeby mazat, neboť s každou zprávou by vznikla další spotřeba úložiště.

[22] Teoreticky lze uvažovat i zahrnutí modelů textového rozhraní i souborové komunikace do tohoto typu, neboť při správné implementaci nemá obsah ani velikost zprávy prakticky žádný vliv na samotné komunikační rozhraní.

V našem systému pak tímto může být textový řetězec, který tato služba vygeneruje a odešle oběma koncům. Ten v nižší pozici (v pozici klienta) pak prokazuje, že k této komunikaci má oprávnění, a server ověřuje, zda takový podpis eviduje.

Forma tohoto podpisu může být libovolná hodnota s vysokou entropií. Kupříkladu by tuto roli zastoupilo vysoké číslo či náhodně vygenerovaný textový řetězec.[23]

Je tedy rozumné, aby klientu a příslušnému serveru bylo explicitně uděleno oprávnění o vzájemné komunikaci formou předání daného klíče[24]. Tento klíč by zároveň měl dosahovat dostatečné délky - v práci implementováno na náhodnou délku mezi 30 a 50 znaky.

### Struktura zpráv

Máme-li zvolenu komunikaci objektovou, pak musíme formulovat strukturu zprávy. Každá zpráva by měla nést několik hlavních atributů. Prvním je již zmíněný token, kterým se ověřujeme. Při komunikaci s notářem (dále jen **Router**) však součástí není, neboť pro toto spojení nebyl udělen. Proto zde musí být stanovena jiná autentizační metoda.

Strukturu obecné zprávy autor sestavil následovně:

- **Token** - pomocí kterého lze ověřit oprávnění o odeslání zprávy
- **Otisk zařízení** - ze kterého lze zjistit, které zařízení zprávu odeslalo[25]
- **Typ zprávy** - typ, který jasně definuje další strukturu a způsob nakládání s ní.
- **Samotný obsah zprávy** - samotná *ne-meta* data, která mají být přenesena
- **Způsob, jakým by měla být zpracována** - způsob či procedura, funkce. Dává příjemci přesné instrukce, jak má být se zprávou nakládáno.[26]

### Způsob zpracování zprávy

Zpráva při přijetí by měla být nějakým způsobem zpracována. Je proto nutné stanovit přesné instrukce, jak se zprávou nakládat. Pokud-že bychom zvolili kategorickou přesnost pro zpracování zpráv, musíme u všech implementovat vzor Command, který se nemusí vždy nutně hodit.

Zvolme proto volnější způsob, kdy nadefinujeme kategorizaci zpráv dle jejich vnějšího typu na:

- **Rozkaz** - zprávy tohoto typu mají přesně stanovenou strukturu a mají především jasně vydefinován způsob, jak s nimi má být nakládáno. Zároveň je tato implementace skryta do momentu, kdy se na druhém konci zpráva začne provádět. Příkladem užitečného užití může být rozkaz o ukončení komunikace, když kupříkladu na jedné straně dojde k chybě a je vhodné komunikaci ukončit.

[23] Při použití znakové sady o malých i velkých písmenech standardní anglické abecedy v kombinaci s číslicemi (0-9), získáváme velikost znakové sady 62. Při kombinaci těchto znaků s opakováním pak máme při $n$ znacích počet kombinací hodnotu $62^n$, což při pouhých 6 znacích určuje počet kombinací na bezmála 57 miliard, kdy při vygenerování a otestování hodnoty v čase 1 $ms$ se dostáváme na dobu takřka dvou let.

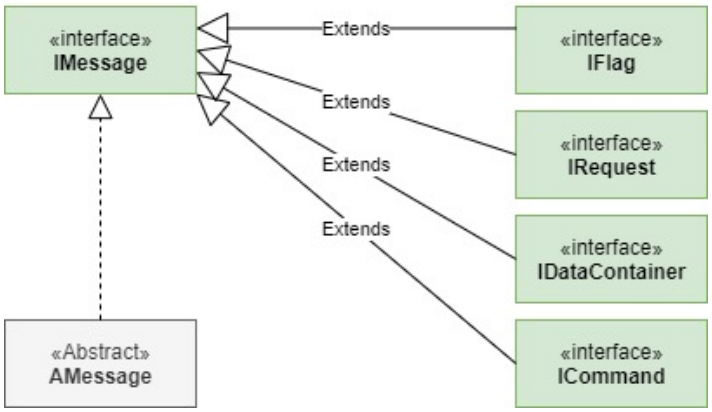
[24] Dále používejme pojmu **Token**

[25] Pro potřeby práce autor zvolil užití IP adresy, MAC adresy a názvu zařízení. Tyto lze sice měnit a MAC adresy nemusí být unikátní, jako se obecně vzato za to má, to však v práci neuvažujme. Nadále tuto substrukturu nazývejme pro zjednodušení **Fingerprint**

[26] viz návrhový vzor **Command** - rozkaz

- **Datový nosič** - je-li nutné odeslat zprávu obsahující data s předpokladem, že nezáleží na způsobu zpracování, může být tento druh zprám zpracováván v zákrytu rozhraní pro datový nosič. Smyslem této zprávy je přenést data, nikoliv nutně provést operaci.
- **Požadavek** - dotaz na další zprávu, na kterou by mělo být reagováno. Druhem nakládání s touto zprávou může být zaměněna s rozšířeným rozkazem.
- **Indikátor** - tento druh zprávy obsahuje hodnotu z předem stanovené škály. Bude-li tato hodnota například interpretována výčtovým typem, lze snadno zaměnit za datový nosič.

Na obrázku *Vztah mezi druhy zpráv* máme ukázánu možnou strukturu, jak vytvořit koncept zpráv. Vidíme zde hlavní interface (rozhraní), které zastřešuje všechny ostatní zprávy. Je to nejobecnější typ zprávy a je vhodné, aby definoval signatury a kontrakty metod pro nakládání s tokenem a s fingerprintem, neboť tyto mají všechny společný. Dále zde vidíme rozhraní **IFlag**, **IRequest**, **IDataContainer** a **ICommand**. Ty dědí tyto parametry z rozhraní IMessage a přidávají si dle potřeby vlastní. Následuje zde ještě entita **AMessage**, tedy abstraktní třída implementující rozhraní IMessage. Ta má za cíl fungovat již čistě jako nosič povinných atributů zprávy (metadat fingerprint a token). Zároveň metody pro správu těchto atributů ve zprávě vlastní, nemusí tedy ostatní potomci dědící z tohoto rozhraní nijak tvořit redundantí a duplicitní kód.




Obrázek 6.3: Vztah mezi druhy zpráv

Pro každou další zprávu pak platí, že stačí, aby dědila z třídy AMessage a je již regulerním typem zprávy. Je však vhodné, aby implementovala jedno z určených standardizovaných rozhraní, které dedí signaturu od IMessage a definuje si vlastní. Zároveň tento model nevylučuje použití více těchto rozhraní najednou - zprávu můžeme chápat jako rozkaz i jako nosič dat, pak stačí implementovat obojí najednou. Zpráva pak v rámci polymorfismu vystupuje v pozici datového nosiče i jako rozkaz.[27]

Zároveň je nutné poznamenat, že každá zpráva (i její obsah) musí být označena značkou **Serializable**, tedy dědit toto rozhraní.[28] To je vhodné, aby bylo provedeno již v typu IMessage.

Samotné zpracování tohoto konstruktu zpráv pak lze provádět dvěma směry. Prvním, který by okamžitě při přijetí začal zpracovávat zprávu a nebo pomocí příslušného zpracovatele. První způsob by nás zavázal k tomu, aby všechny zprávy dědily z rozhraní ICommand a tedy nutně měly nadefinován způsob zpracování již uvnitř. Druhý způsob nás od této povinnosti osvobozuje, ale svazuje nás zase v tom, že musíme mít třídu, která by daný druh zprávy uměla zpracovat.

[27] Příkladem tohoto užití by mohl být rozkaz typu „odešli konkrétní data na adresu uvedenou v této zprávě“

[28] Tím de facto jen virtuálnímu stroji dáváme najevo, že tuto třídu a její instance lze dekomponovat a odeslat přes socket, stejně jako tyto zprávy zpět složit.

Autor si vybral druhý typ a hned ho také navrhl. Uvažujeme-li zpracovatele zpráv (dále v práci **Handler**), měl by tento umět převzít zprávu, ověřit, zda mu její zpracování náleží, a provést předem stanovenou proceduru nad touto zprávou. Jinými slovy v momentě přijetí zprávy je tato předána zpracovateli, který ji zpracuje. V případě potřeby obměny není pak nutné měnit zprávu, stačí upravit či nahradit tuto komponentu zpracovatele.[29]

Takto zároveň odpadá problém s ověřením, zda tento typ zprávy může být na daný typ serveru odeslán - to je restriktivně omezeno typy těchto handlerů. Z toho nepřímo vyplývá, že každá strana spojení by pak měla mít svoji vlastní přepravku těchto handlerů a měla by při zpracovávání zprávy vybrat ten správný. Chápeme-li problematiku takto, je vhodné, aby při provedení zpracovávání byl postup přepravky následující:

1. Přijetí zprávy a postoupení přepravce handlerů
2. Vybrání handleru v pořadí a postoupení mu zprávy
   (a) Ověření příslušnosti zprávy
      - Přísluší tomuto handleru zpracovat tento typ zprávy.
      - Nepřísluší tomuto handleru zpracovat tuto zprávu. Přepravka pokračuje znovu od kroku 2.
   (b) Zpracování zprávy
3. Vrácení zprávy o tom, zda přepravka nalezla či nenalezla příslušný handler a tedy zda zpráva byla zpracována.

### 6.2.5 Autentizace uzlů

Mluvíme-li o autentizaci pomocí spojení pomocí tranzitu objektů, uvažujeme, že tento proces autentizace je de facto také jen organizovanou procedurou odesílání a příjmu zpráv. Navrhněme tedy primární, obecný model.

Jakýkoliv uzel v síti systému se musí autentizovat a musí být podroben sérii testů. Pro úspěšnou autentizaci pak musí projít kumulativně všemi. Tyto zprávy však nepatří mezi ostatní (například určené pro přenost citlivých dat) a zároveň nechceme aby potenciální útočník mohl tyto zprávy odesílat ještě před provedením autentizace.

Autor proto navrhl způsob, kdy přepravka handlerů (dále jen **HandlerContainer**) je děděna a překryta dalším druhem přepravky, tedy autentizačním kontejnerem. Tomu přísluší handlery autentizační povahy (například příjem zpráv obsahující uživatelské přihlašovací údaje) a žádné jiné. Zároveň po dokončení těchto procedur je ověřovaná strana obeznámena o výsledku autentizace a je jí dáno slovo ohledně zpráv, které má připravené k odeslání.[30]

[29] Toto je vhodné právě pro tento druh aplikací. Odesílaná zpráva musí mít shodně napsanou třídu na obou koncích spojení, jinak může dojít k nepříznivým důsledkům - Můžeme to chápat i jako ochranu, změnou struktury zprávy na jednom konci spojení (či v průběhu přenosu) by za jinak nezměněných okolností mohla u typu ICommand způsobit obrovské bezpečnostní hrozby. Takto může stroj sám rozpoznat nepříslušnost zprávy a vyhodnotit jako nezpracovatelnou sám. V případě, že chceme tyto zprávy použít i k jiným účelům, stačí jen nadefinovat jiný handler a ten se o zbytek postará relativně autonomně.

[30] Ve skutečnosti autor zvolil způsob obrácený, kdy u nadřazené strany spojení jsou dva druhy kontejnerů - jeden pro autentizační zprávy a druhý pro zprávy business logiky. pouze po úspěšném ověření všemi testovanými procedurami se přepne kontejner „do režimu business“ a umí zpracovat i jiný druh zpráv. Pokud by se podřízená strana komunikace pokusila odeslat takovouto zprávu ještě před dokončením autentizace, nebude přijata a ani zpracována - s výjimkou žádosti o ukončení spojení.

Příkladně by tyto procedury mohly být následující:

- Odeslání požadavku na autentizaci z klienta
- Server odešle požadavek na uživatelské jméno. Klient má na tento druh zprávy připraven handler a odešle automaticky své uživatelské jméno.
- Server odešle požadavek na uživatelské heslo. Klient opět v reakci odpoví na tuto zprávu odesláním hesla v předem stanovené podobě.
- Server odešle výsledek autentizačního scénáře.

Výhodou oddělení těchto HandlerContainerů spočívá v tom, že by mohla umožňovat zpracování běžných zpráv tak, že při prvním zpracování zprávy již další handler hledán nebude. V případě autentizačních scénářů však neumožňuje kombinaci těchto testů. Je tedy vhodné, aby zpráva *„Chci být autentizován“* byla zpracována více handlery najednou a tím bylo dosaženo komplexnější autentizace.

#### Automatické uzavírání spojení

Může být potřeba některá spojení uzavírat nikoliv v návaznosti na nějaké operaci, ale po uplynutí nějaké doby. A to například z bezpečnostních důvodů či k zajištění plynulé komunikace a nepřetěžování kapacit.

Uzavírat spojení v návaznosti na čase je vhodné ve dvou situacích:

- Při dlouhé autentizaci - Není-li schopen se uzel po určený časový úsek přihlásit, automaticky spojení uzavřít.
- Při dlouhé prodlevě mezi odeslanými zprávami - Není-li spojení dlouho využíváno, je rozumné jej uzavřít a otevřít pro další zařízení ve frontě.

Automatizací tohoto procesu odpadá velké množství problémů s přerozdělováním zdrojů, napomáhá k plynulému chodu celého systému a snižuje počet aktuálně běžících vláken na zařízení.

Tvorbu tohoto řešení je možné zajistit pomocí nadefinování odpočtu nastaveného na časový úsek. V případě změny (například provedení úspěšné autentizace) by tento odpočet měl být resetován. Tento systém si nadefinujme třídou **ConnectionAutoCloser**.

### 6.2.6 Shrnutí návrhu komunikace

Chápeme tedy zprávu jako objekt příslušné třídy, který má být odeslán po spojení mezi dvěma uzly - sockety. Handler chápeme jako zpracovatel zprávy, který ověřuje příslušnost zprávy (zda je zpráva daného typu) a v případě, že ano, pokusí se ji zpracovat. HandlerContainer pak vnímáme jako soubor těchto handlerů, v němž jsou zprávy zpracovávány.

## 6.3 Servery a služby

Jak je uvedeno na straně *18* v podkapitole *Server, Služba*, chápejme tyto pojmy odlišně. Mluvíme-li tedy o serveru, ten by měl být jedinou instancí v běžící aplikaci (přirozeně těchto zároveň běžících aplikací uvažujeme neomezené množství), měla by tedy být v souladu s návrhovým vzorem Singleton.

Třída definující server by měla obsahovat jistý kontejner pro služby, které při spuštění serveru také spouští. Tyto služby pak naplňují business logiku.

### 6.3.1 Služba Router

První službou, kterou uvažujeme, je **Router**. Ta plní v systému roli dvojí:

- **notářskou** - tedy ověření a vygenerování tokenu
- **přepojovací** - spojení dvou zařízení mezi sebou

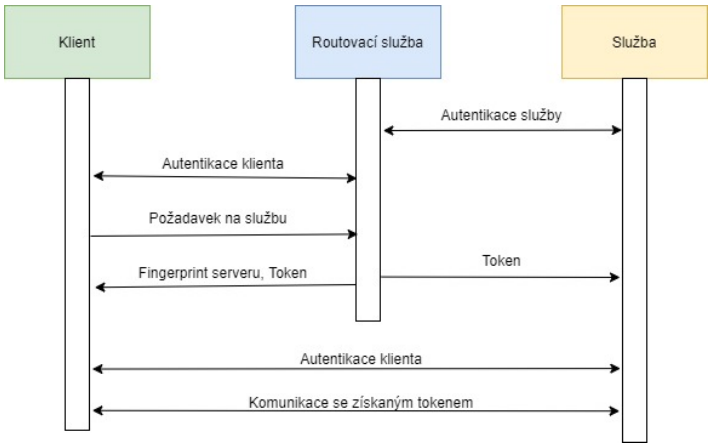
První roli jsme již popsali na straně *61* v podkapitole *Návrh komunikace*.

Tou druhou rolí je forma propojení. Je přirozeně možné, aby klient měl v sobě uloženy všechny adresy se všemi proty, není to však kýžený stav, neboť při změně na straně serveru by muselo dojít i ke změně na straně klienta.

Sekvenci komunikace mezi klientem a službou pak můžeme popsat takto:

1. Klient se ověří na routeru
2. Klient se dožaduje připojení k službě *xxx*
3. Router odešle IP adresu a port, na který se má klient připojit
4. Router službě *xxx* a klientovi odešle vygenerovaný token
5. Klient se připojí k službě a ověří se tokenem

Tento proces je graficky znázorněn na obrázku *6.4 Návrh komunikace*. Na hypotetické ose $y$ chápejme čas, který plyne.




Obrázek 6.4: Návrh komunikace

### 6.3.2 Služba

Vedle routeru chápeme, že existuje služba. Ta funguje jako zpracovatel uživatelských požadavků v rámci business logiky. Zde spočívá kritická oblast této práce - připravit platformu pro volné rozšiřování tak, aby bylo možné s co nejmenším úsilím upravit, změnit či doplnit aplikaci k business modelu dle vlastní potřeby. Je zde potřeba připraveného rozhraní, které by bylo pro všechny druhy služeb společné. Dále pak uvažujme hierarchicky níže - do jakých vztahů mohou tyto služby vstupovat?

Rozdělme tyto do dvou hlavních kategorií. Prvním vztahem je přirozeně vztah s Routerem. V tom je služba podřízena Routeru, je tedy v této komunikaci klientskou stranou. Zároveň však chceme, aby tato služba byla nadřízena klientovi (ve smyslu zaměstnance).

Ač se router sice jeví v mnoha ohledech jako jakákoli jiná služba, hlavní rozdíl spočívá v tom, že router není v žádném ze vztahů klientem, vedle toho služba ano, a to právě s routerem. Mohou tedy míti ne-li shodné, pak obdobné atributy.

Příkladem shodného atributu je udržení typu služby, aby se jím mohla prokazovat. Sdílí tedy s routerem nejen signatury ze společného rozhraní, měly by tyto dvě entity sdílet i společného předka. Toto rozhraní nazvěme **IService** a předka (abstraktního) pojmenujme **AService**. Tímto formalizujeme společné atributy a samy se mohou lišit vcelku libovolně.

Dalším článkem řetězce mezi službou a třídou AService (uchovávající typ služby, kterým se autorizuje), je článek **AGeneralService**. Na této úrovni se již nutně musí lišit od třídy Router, neboť zde uvažujme již náznaky odlišné logiky a hlavně jiný přístup ke komunikaci s okolím.

Zůstaneme-li poplatní paradigmatu o univerzálnosti, zde pak je nutné podotknout, že jediným omezením, které případné rozšíření musí splňovat, je nutnost dědění třídy AGeneralService a uvažování komunikace s klientem i routerem.

### 6.3.3 Klient

Klient je vedle Routeru a obecné služby posledním z tohoto souboru vzájemných vztahů nejmenovaný. Jde o takovou entitu, kdy oproti routeru je ve všech komunikačních kanálech na nižší úrovni - podřízené serveru. Zároveň však zastává roli samotného zařízení. Neboť je v jeho vlastní režii kdy se kam připojí, může si dovolit víceméně samostatné působení. Dále předpokládejme, že nebude existovat v rámci jedné spuštěné instance aplikace více instancí klienta, lze ji tedy nastavit jako **Thread-safe** jedináčka (tedy dle návrhového vzoru Singleton).

Předpokládejme, že v rámci spojení i v rámci zpráv může docházet k chybám. Bylo by tedy vhodné formalizovat všechny očekávatelné chyby a popsat způsoby jejich řešení. Tímto zajistíme, že rutinní chyby může klient sám řešit za předpokladu, že danou chybu eviduje a má nastavenou metodu řešení. Prohlásíme-li instanci klienta jako řešitele problémů, pak musíme stanovit jednotlivé chyby. Jako chybu považujme například chybné přihlašovací údaje či například neschopnost Routeru poskytnout adresu služby (z důvodu, že není spuštěna). Na tuto situaci musí klient provést nějakou proceduru, aby nedošlo k nezastavitelné kaskádě dalších problémů. Tento soubor možných řešení přitom může zůstat shodný pro více druhů chyb.

Na první zmíněnou chybu (tedy chybné přihlašovací údaje) by mohlo být reagováno například postupným ukončením daného spojení. Dalším možným řešením by mohlo být okamžité odpojení, ukončení celé aplikace, pokus o znovuprovedení akce či třeba i nedělání ničeho, tedy ignorování chyby. Tyto chyby definujme ve výčtovém typu **ESolveMethod**, samotné odchytávače chyb definujme vždy příslušnou třídou - objektem[31].

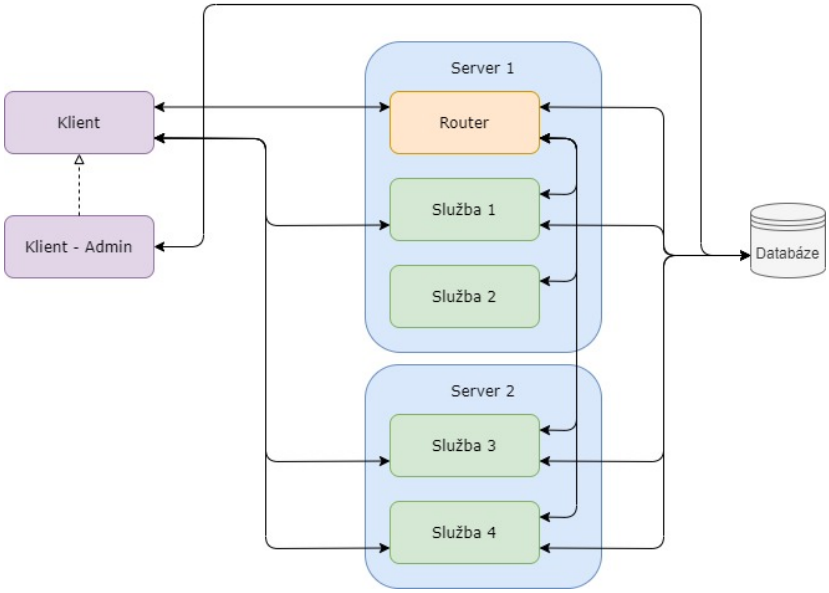
Uvědomíme-li si existenci pravidla, že služba (i router) jsou schopné v rámci zabezpečení zpracovat

[31] Lze si povšimnout podobného řešení ve vztahu mezi přijatou zprávou a příslušným handlerem. Také v tomto případě chceme, aby byly tyto chyby odchytávány a předávány do vlastních řešitelů problémů. Tyto však mají navíc volitelnou metodu, pomocí které bude problém řešen.

zprávu jen v případě, že druhou stranu považuje za dostatečně autenizovanou, narážíme na další omezení. Klient by musel být explicitně obeznámen s tím, že zpráva nebyla přijata, což by byl problém redundantního zahlcování sítě zprávami, dále by to mohlo být i riziko bezpečnostní - postupným zahlcováním služby nestíhající odpovídat by mohlo dojít (v extrémním případě) až k vyřazení služby z funkce. Zároveň však, aby nebyly zprávy smazány, je vhodné je uložit a odeslat v momentě, kdy je spojení autentizováno. Při abstraktnějším pohledu zde spatříme návrhový vzor **Observer**, kdy chceme, aby posluchač (observer) byl informován v momentě, kdy se objekt naslouchání (vydavatel; případně i subjekt) pozmění. Toho lze dosáhnout pomocí rozhraní **IObserver** a **ISubject**, kde první zmíněné má pouze metodu k tomu *být notifikován*, ISubject pak metody na přidání posluchače a upozornění všech posluchačů.[32] Pomocí tohoto přístupu lze vytvořit frontu zpráv, které čekají na odeslání. Aby se předešlo hromadění, jsou tyto zprávy (zakryté rozhraním IMessage) ještě obaleny v objektu typu **MessageTask** s vnitřní informací příjemce, ale především expirační dobou. Dojde-li zpráva expirační doby, je z této fronty vyřazena jako neodeslatelná, v praxi to znamená, že spojení se nepovedlo autentizovat či vůbec navázat.

### 6.3.4 Návrh topologie

V této podkapitole se podívejme na rozdělení topologie poněkud podrobněji a v souladu s poznatky z předchozích kapitol.[33] Nyní chápeme, že systém musí mít entitu typu **klient**, která má za cíl komunikovat se **službami**, ty zase komunikují s databází. Podrobněji lze vidět tyto vztahy na obrázku *6.5 Topologický model systému* na straně *68*.




Obrázek 6.5: Topologický model systému

Na poukázaném obrázku máme vidět modelový návrh, jak by topologie systému mohla vypadat. V první řadě zde máme entitu Klient. Ta má za cíl komunikovat a priori se službami, jimž zadává úkoly, přičemž nemusí mít navázané spojení se všemi - na obrázku je naznačeno záměrně spojení

[32] V abstraktním modelu to funguje tak, že subjekt při sledované změně upozorní všechny, kteří jsou přihlášeni k poslechu. Kouzlo celého tohoto vztahu je, že jednotliví pozorovatelé mohou mít rozdílné implementace a nemusí se repetetivně dotazovat změny, stejně jako nemusí mít subjekt implementovánu metodu pro provedení změny nad všemi pozorovateli.

[33] Především pak *5.3.1 Topologie systému* na straně 51 a *6.2.4 Návrh komunikace* na straně 61

jen se službami 1, 3 a 4. Dále má přímé spojení i k routeru pro potřeby přepojení na novou službu. Vedle klienta (respektive pod ním) je vidět i entita **Klient - Admin**, jejímž cílem je vlastně „dědit“ vlastnosti klasického klienta, zároveň však přidat schopnost připojení se přímo k databázi.[34]

Serverová strana spojení je patrná na objektech *Server 1* a *Server 2*, kde můžeme mluvit především o jednom velkém rozdílu - první zmíněný obsahuje *Router* a služby se k němu připojují přímo (v rámci serveru), přičemž druhý obsahuje pouze služby a ty se k routeru připojují v rámci sítě.[35] Tyto služby (jak již bylo naznačeno) mohou komunikovat s klienty, musí komunikovat s routerem (z důvodů správy spojení) a mohou komunikovat s databází - to je patrné u *Služby 2*, která komunikuje pouze s routerem.

## 6.4 Spojení

Spojení rozdělme dle uzlů, od které role systému je komunikováno a kdo je příjemcem. To je ilustrováno v tabulce *6.1 Matice definic spojení mezi uzly v síti*, kde na ose $y$ najdeme lokální komunikátor a na ose $x$ cílové spojení. Toto rozdělení bylo zvoleno pro snazší orientaci v atributech, které jednotlivé mechanismy udržující spojení mají.

Tabulka 6.1: Matice definic spojení mezi uzly v síti

| | **Router** | **Služba** | **Klient** |
|---|---|---|---|
| **Router** | x | R2SConnection | R2CConnection |
| **Služba** | S2RConnection | x | S2RConnection |
| **Klient** | C2RConnection | C2SConnection | x |

### 6.4.1 Atributy spojení

Pro každé spojení můžeme mluvit o několika hlavních atributech, které mu náleží. Začněme tedy tím, co mají všechna spojení společné.

Každé spojení by mělo mít vlastní nástroj pro komunikaci - v našem případě instanci třídy SSLSocket. Dále pak nástroje pro ovládání tohoto socketu - pro vpisování a čtení ze spojení - instance třídy ObjectInputStream a ObjectOutputStream[36] Pomocí těchto pak můžeme vpisovat a číst zprávy ze spojení. To nám definuje i dvě hlavní metody, které každé spojení musí mít - číst a psát zprávy. Dále musí mít nástroj pro přidání handleru[37]. Z toho vyplývá, že musí mít každé spojení i příslušný HandlerContainer, pomocí kterého zpracovává příchozí zprávy.

[34] Toto dědení dále v práci uvažujme jen na bázi přidání funkcionality pro roli *Administrátor*.

[35] Umožnění tohoto řešení autor přijal z důvodu možnosti rozložení nároků na výkon mezi více zařízení. **Tím autor zaručuje nezávislost na zdrojích podniku** a de facto se snaží, aby bylo možné (ad absurdum) spustit celý systém z jednoho mobilního zařízení i na mnoha serverech (v běžném slova smyslu) zároveň. Více je možné se dočíst v kapitole *10 Konfigurace a nasazení* začínající stranou *95*.

[36] Pro zjednodušení dále tyto nazývejme pojmy **input** a **output**.

[37] Spojení služby s klientem může mít pro každý druh služby odlišné handlery, užijeme-li lazy initialization a přídáme-li tyto až v nejzazší možný moment (při inicializaci zdrojů v konstruktoru), zajistíme tím i univerzalitu a znovupoužitelnost jedné třídy pro více účelů jen vložením dalších parametrů.

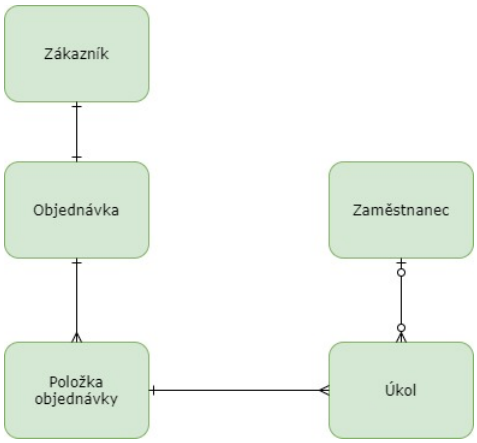
Z těchto společných atributů můžeme složit základ hlavního předka - interfejsu **IConnection**. To má za cíl nejen býti předkem všem ostatním spojením, ale také formalizovat a kompilovat signatury, které všechny instance implementující toto rozhraní musí povinně musí umět.

To, co rozlišujeme je pak rozdíl mezi spojeními na straně nadřazené entity a té podřízené. Nadřazeným (tedy v pozici serveru v obecném slova smyslu) náleží vlastní HandlerContianer, který mají určen výhradně pro autnetizační scénáře. Po úspěšné autentizaci pak přepnou užití do druhého, toho pro běžné zprávy business logiky.

## 6.5 Objednávání

V této podkapitole autor popisuje návrh business logiky aplikace s cílem osvětlit návrh celého systému nastíněného procesy z kapitoly *4 Specifikace procesů podniku*. Zároveň se zde zaměřuje především na hlavní část business logiky, tedy na sestavení a vyhotovení objednávky.

K tomu, aby se dal vytvořit kód aplikace, je nutné míti zprvu stanovenou abstraktní rovinu problému (tedy jakýsi průběh řešení situací - lze do jisté míry chápat i jako algoritmus v běžném světě) a chápat entity, které v systému vystupují. Na modelu *6.6 Model entit hlavní business logiky* je vidět vztah mezi entitami v konceptuální úrovni chápání světa.




Obrázek 6.6: Model entit hlavní business logiky

Máme zde několik entit s patrnými vztahy mezi nimi. Prvním vztahem je relace mezi objendávkou a zákazníkem. Zákazník je zde ilustrován jen pro názornost a kompletnost, v rámci práce vystupuje jen jako aktér, případně textová hodnota (například číslo zákaznické karty v kombinaci s kontaktním údajem). Vztah mezi nimi je 1:1, neboť uvažujeme, že takto interpretovaný zákazník bude plnit právě jednu objednávku, stejne jako každá objednávka v systému musí mít nutně právě jednoho zadavatele.

Další relací je vztah mezi objednávkou a položkou objednávky. Objednávku chápejme jako kompilaci položek objednávky ve stanovené kvantitě. Každá položka musí být unikátní, byť by obsahovala produkt shodný s produktem jiné položky. Zároveň každá taková položka musí mít právě jednu objednávku, které náleží, nemůže existovat sama o sobě a nemůže mít více evidovaných objednávek (tím by byla narušena unikátnost).

V neposlední řadě autor dává do souvislosti položku objednávky a úkol. Tento vztah pak chápe jako 1:N, kdy k jedné položce může být přiřazeno více úkolů, ale vždy alespoň jeden. Nelze vyhotovit tuto

položku bez připravení alespoň jednoho kusu zboží, musí tedy alespoň jeden existovat. Existence více těchto úkolů je umožněna pomocí tzv. **SideTasks**. V systému chceme, aby bylo možné provádět i vedlejší úkoly, meta-úkoly chcete-li. Těmi můžeme chápat například drobnou inventuru daného kusu zboží na skladě za účelem odhalení případných chyb v rámci skladového systému.

Konečně zde máme vztah mezi zaměstnancem a úkolem - přirozeně chápeme, že zaměstnanci může být přiřazeno více úkolů najednou, stejně jako nemusí existovat žádný takový. V obrácené situaci zase vidíme, že může (byť přechodně) existovat úkol, který nemá přiřazeného zpracovatele, ale když, tak maximálně jednoho. Vztah tedy interpretujme jako oboustranně nepovinné 0:N.

## 6.6 Databáze

Pro potřeby ukládání dat užíváme různých druhů úložišť s různou datovou strukturou. Autor práce uvažuje následující struktury a přístupy pro ukládání vygenerovaných dat, aby z nich vybral pro tuto práci nejvhodnější:

- **Sdílený soubor** - Způsob existence souboru jakožto zdroje, ke kterému má přístup více entit (aplikací či modulů aplikace) naráz. To může být považováno za vhodné pro drobné aplikace, kde neuvažujeme praktickou možnost kolize. Zástupci mohou být dokumenty ve formátech s tabulkovou strukturou (např. **CSV**) či se stromovou strukturou (převážně XML či JSON).
- **Relační databáze** - Obvykle mluvíme o souboru navzájem propojených tabulek vztahy 1:1, 1:n či m:n. Obvykle jde o tzv. SQL Databáze.
  - Silná závislost na schématu
  - Tradiční způsob uchovávání dat
  - Zástupci jsou **MySQL**, **Microsoft SQL Server** či **PostgreSQL**[38]
- **Dokumentová databáze** - (Pro zjednodušení) chápejme takovou databázi jako soubor dokumentů, kde jeden dokument reprezentuje objekt jako abstrakci z reálného světa.[39]
  - Volná a jednoduše rozšiřitelná struktura
  - Moderní forma ukládání velkých objemů dat
  - Majoritními zástupci jsou **MongoDB**, **Apache Cassandra** či například **RavenDB**

První alternativu lze ze smyslu použití automaticky vyloučit, neboť by implementace ochrany samotného souboru před přepisem jednotlivými komponentami byla pro potřeby práce zbytečná. Dále by nesplňoval systém možnost infrastrukturní nezávislosti. Lze tedy uvažovat, že alternativami zůstávají relační databáze a dokumentově orientované databáze. Autor pro potřeby práce volí relační databáze pro snadné použití SQL dotazy nad daty.

### 6.6.1 Konceptuální model databáze

Konceptuální model databáze (v relačně orientovaném kontextu) lze popsat jako soubor entit navzájem propojených svými atributy a explititními vztahy 1:1, 1:N a M:N.

Tento model autor vyhotovil a je umístěn v příloze *B Konceptuální model databáze* na straně *112*

[38] Má podporu i „ne-relačních“ dotazů pro JSON.
[39] Na rozdíl od relačních databází, kde takový objekt chápeme jako „řádek tabulky“.

### 6.6.2 SQL kód pro nastavení databáze

Kód v SQL pro vzorové nastavení základní databáze a jejích tabulek je uveden ve výpise v příloze *C Zdrojový kód SQL pro spuštění databáze* na straně *113*.

### 6.6.3 Komunikace s databází

Pro komunikaci s databází v rámci aplikace fungující na bázi Javy existují hlavní dva směry. Oba mají svá pozitiva a svá negativa. Cílem této podkapitoly je vybrat co nejvhodnější.

#### JDBC API

Vývojářské vybavení **Java Database Connectivity** poskytuje potřebné nástroje k přístupu k relačním databázím pomocí aplikace. Poskytuje solidní a jednoduché funkce pro tvorbu, exekuci a commit SQL příkazů, přičemž vyniká optimalizací rychlostí transferu dat. Dále umožňuje použití tzv. **PreparedStatement**, tedy předpřipravení příkazu dle stanovených pravidel. Toho lze využít jako obrana proti SQLInjection, tedy vložení (škodlivého) kódu do parametru dotazu.

```
1
2  String firstName = "Alice";
3  Connection connection = // Inicializace spojeni
4
5  String query = "SELECT LASTNAME FROM PEOPLE WHERE FIRSTNAME LIKE ?;"
6
7  PreparedStatement preparedStatement = connection.prepareStatement(query);
8  preparedStatement.setString(1, firstName);
9
10 ResultSet resultSet = preparedStatement.executeQuery();
11
12 while(resultSet.next()) {
13
14   System.out.println(resultSet.getString(1));
15 }
```

Výpis kódu 6.1: Ukázka použití konstruktu PreparedStatement

Ve výpisu kódu *6.1 Ukázka použití konstruktu PreparedStatement* vidíme ukázkový způsob, jak vypsat všechna příjmení osob s křestním jménem "Alice"pomocí objektu PreparedStatement. S jeho využitím lze prakticky abstrahovat od SQL injection v pravém slova smyslu, neboť vložená hodnota bude databázi předána jako čistě textová hodnota. Obdobně by tomu mohlo být i u hodnot typu `Long`, resp. `BIGINT` či například u `LocalDateTime`, resp. `TIMESTAMP`.

Výhodou je, že toto rozhraní není absolutně jakkoliv závislé na platformě či struktuře systému. Volitelně lze přepojovat mezi adresami databáze real-time či okamžitě měnit uživatelské údaje pro připojení. Další výhodou je již zmíněná rychlost propojení, jelikož je JDBC záležitostí „čisté“ Javy a nepoužívá nadměrných konstruktů.

Nevýhodou je podstatně složitější zacházení s kódem. Je nutné ho napsat mnohem více a pravděpodobně se nelze vyhnout u větších projektů ani opakování kódu.

Formální nevýhodou je de facto popírání objektového přístupu, což může být z pohledu teoretické a kategorické přesnosti přinejmenším zavádějícím, zvláště pak při práci se silně

staticky typovaném jazyce, jakým Java je. A to přirozeně za předpokladu, že se nad JDBC nepostaví persistentní vrstva.

#### Hibernate

Pro komunikaci s databází lze využít i frameworku **Hibernate**[40] spravujícího veškerý transfer dat mezi aplikací a databází. Obecně se chápe tento framework jako **Objektově relační mapování** (zkráceně **ORM**)[41].

Tento framework pracuje na kombinaci předdefinování entit (v aplikaci pomocí objektů) a entitního manažera (instance třídy **EntityManager**).

Obvykle se struktura aplikace (s uvážením pesistence) pak dělí do několika hlavních úrovní:

1. **Business logika** - samotný řešený problém
2. **„Servisní logika“** - přístupový nástroj pro komunikaci mezi business logikou a nižšími vrstvami
3. **DAO** - Data Access Object
4. **Entity**
5. **Databáze**

Tento framework nejen usnadní práci při vývoji, ale dokáže si poradit i s kvantem standardně ošetřitelných chyb. Nevýhodou však je náročná real-time konfigurace.

#### Výběr směru

Autor v této práci upřednostňuje užití JDBC API pro jeho univerzální použití a vysokou přenosovou efektivitu. Zavazuje se však k poskytnutí nutných kroků, aby bylo možné o Hibernate celou aplikaci rozšířit a komunikaci s databází na tomto přístupu postavit.

### 6.6.4 Návrh persistence

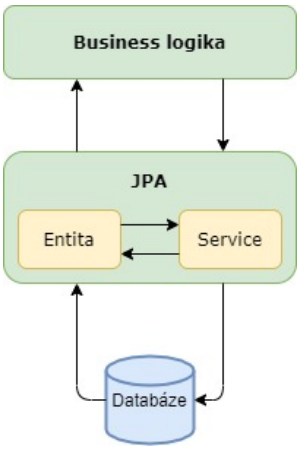
Persistenční model autor navrhl s ohledem na osvědčené metody z implementace persistence pomocí Hibernate.[42] Model si ilustrujme na diagramu *6.7 Návrh JPA* - je zde vidět rozdělení aplikace do 3 hlavních spolu navzájem komunikujících celků.

V horní části je možné vidět entitu Business logika - tu v nynější situaci chápejme jako zastřešující orgán působnosti aplikace - nyní si tam pro zjednodušení začleňme i funkce jakými

[40] Dokumentaci lze nalézt pod tímto odkazem: `https://hibernate.org/orm/documentation/5.4/`

[41] Tento přístup je z teoretického hlediska mnohem vhodnější pro použití při používání objektově orientovaného programování, neboť programátor může využít účelové fikce a výsledky z tohoto spojení pak považuje za plnohodnotné objekty. U JDBC musí takové zacházení zřídit sám.

[42] Mimo jiné i pro snazší pozdější reformulaci požadavků na kombinaci aplikace a databáze.




Obrázek 6.7: Návrh JPA

je například grafické rozhraní[43]. Tato komunikuje s úrovní **JPA**[44]. Ta se sestává především ze samotné entity (například zaměstnanec) a z jí náležící „služby“[45]. Zatímco entita vystupuje v programu v roli objektu jako jakéhokoliv jiného, služba je jeho (s přimhouřením oka) „tovární třídou“[46].

Zatímco entita je reprezentována jako objekt z virtuálně-reálného světa (například má tzv. gettery, settery či instanční metody definovány v souladu se smyslem instance), servisní (pomocné) třídy slouží k přístupu k entitám (uloženým) v databázi. To znamená, že musí mít implementovány metody pro výběr konkrétní entity vybrané dle předem známého identifikátoru, pro získání všech instancí, pro smazání, upravení a vytvoření instance entity. Chápejme, že signatury těchto metod jsou společné pro všechny služby (service) komunikace s databází. Definujme si tedy rozhraní **IPersistor** pro určení všech společných rysů těchto služeb, předpis pro entitu definujme jako rozhraní **IEntity**.

V poslední řadě je zde databáze, se kterou komunikujeme. Předpokládejme, že je pro vyvíjenou aplikaci správně nakonfigurovaná a data přístupná. O konzistenci dat se při správném nastavení všech relací stará databáze sama[47].

Je vhodné udržovat uživatelské přístupy zabezpečené. Nelze považovat za *best practice* uložit tyto přímo do kódu v plain-text podobě. Stejně jako není vhodné, aby více přistupujících užívalo jeden uživatelský účet či aby všechny účty měly stejná práva k manipulaci s daty.

Předpokládejme, že k tabulce v databázi přistupuje více různých komponent systému. Uvažujme například entitu zaměstnance, pro níž autor navrhuje dvě příkladná omezení pro zajištění co nejvyšší datové integrity a bezpečnosti systému:

[43] Obvykle bychom ho chápali jako součást jiných úrovní

[44] *„Java Persistence API“*

[45] Nemá nic společného s dřívějšími užitími slova „služba“, autor pouze dodržuje konvence o pojmenovávání takových objektů, entit a tříd, které umožňují převod mezi údaji v databázi na datové nosiče v podobě objektů v kontextu OOP.

[46] Nesprávně se považuje za koncept návrhového vzoru **Tovární třída** jakákoliv třída mající metodu, jenž volá konstruktor instance třídy tvořené. My se však k tomuto omylu krátkodobě uchýlme.

[47] Přirozeně při běžném a vhodném používání

1. Správa zaměstnanců musí mít povoleno z této tabulky mazat, přidávat nové zaměstnance, upravovat stávající a vybírat jednotlivce i skupiny (či celky).
2. Správa skladu může potřebovat vypsat například zaměstnance, kteří zastávají v podniku nějakou roli. Je však nežádoucí, aby měla oprávnění k mazání, editaci či přidávání nových zaměstnanců.

V případě nedodržení tohoto pravidla může celý systém docházet zdánlivě nedeterministického chování, chyby nebude možné sledovat a vystavuje celý systém zbytečnému riziku pokusu o napadení. Vhodné by tedy intutitivně mohlo být rozdělit tyto „účty“ dle jejich potřeb v databázi. Autor proto postuluje řešení pomocí třídy dle vzoru jedináček, který kompiluje daný druh uživatelského spojení s databází[48] závisícího na smyslu užití. Pojmenujme tuto třídu **DatabaseConnectionContainer**.

### 6.6.5 Zabezpečení databáze

V rámci zabezpečení databáze či komunikace mezi databází a systémem nebylo implementováno žádné řešení. Komunikace s databází byla zkoušena výhradně při použití defaultního nastavení zabezpečení databáze **H2**.

[48] rozuměj spojení s databází pod uživatelským učtem

# 7. Psaní kódu

## 7.1 Konvence pro psaní kódu

V této podkapitole autor popisuje použité konvence pro psaní kódu, konvence pro použité názvy a obecně závazné konvence použití, a to identifikátorů, názvů tříd, interface, výčtových typů, abstraktních tříd či metod. Takto činí pro potřeby čtenářů, aby bylo možné se lépe orientovat v kódu.

### 7.1.1 Názvy tříd

V první řadě se podívejme na názvy tříd a obdobných entit v rámci OOP. První zásadou je používat tzv. „Velbloudí notaci“[1]. Další zásady jsou uvedeny v následujícím výpise:

- Obecné třídy nemají pro názvy zvláštních specifik vyjma dosud zmíněných.
- Abstraktní třídy vždy začínají písmenem `A`, následující znak je vždy kapitálka.
- Název interface začíná vždy písmenem `I`, následující znak je vždy kapitálka.
- Třída výčtového typu užívá jako první písmeno `E` (z anglického Enum), následující znak je vždy kapitálka.

### 7.1.2 Názvy identifikátorů

V další fázi se podívejme na identifikátory. To pro naše potřeby chápejme jako úložiště `pointeru`[2]. Tento koncept Java *a priori* nevyužívá a lze pro zjednodušení i předpokládat, že ani nepodporuje - vzniká zde fikce, že libovolná proměná má v sobě uloženu prakticky celou instanci třídy shodné s typem proměnné.

Mluvíme-li o názvech proměnných, musíme rozlišovat zda jde o proměnné instanční (tedy náležící instanci, respektive objektu), či o proměnné statické (náležící třídě). Pro první zmíněné platí obdobné pravidlo jako pro obecnou třídu (tedy velbloudí notace), jen s tím rozdílem, že první písmeno celého názvu necháváme psané malým písmenem. Vedle toho název statické proměnné zapisujeme vždy kapitálkami celý, jednotlivá slova rozdělujeme podtržítkem). Lokální proměnné mají stejná pravidla jako instanční atributy.

[1] Tedy že každé slovo, ze kterého se název skládá začíná velkým písmenem, ostatní jsou malými. Jednotlivá slova nejsou rozdělena mezerou, ani žádným jiným znakem, jednotlivá písmena zkratek jsou považována automaticky za samostatná slova, jsou tedy vždy kapitálkami.

[2] reference na instanci

### 7.1.3 Názvy metod

Pro názvy metod volíme opět konvenci o velbloudí notaci, identickou s notací pro názvy instančních proměnných.

V práci se používá mnoho tzv. „getterů“ a „setterů“[3]. Pro ty platí, že vždy začínají slovem `get` a následují názvem proměnné, které se týká, pro gettery. Pro settery platí obdobné pravidlo za použití slova `set`.

Speciálním případem je, když mluvíme o getteru proměnné typu `boolean`. Pak lze použít místo slova `get` slova `is`, čímž zpřesňujeme uvažování nad modelem - ptáme se pak totiž zda je nějaký konkrétní boolovský stav nastaven na hodnotu pravda (`true`). Pokud že tomu tak je, pak vrací `true`, jinak vrací `false` (nepravda).

### 7.1.4 Další konvence

Pro názvy balíčků platí shodné pravidlo, jako pro pro instanční proměnné, tedy velbloudí notace s prvním písmenem malým. Dále je „cesta“ od kořenového adresáře k cílovému uzlu popsána řetězcem názvů těchto balíčků oddělených tečkou.

Další konvencí, kterou autor zastává, je používání při programování výhradně anglického jazyka. Je zvykem ho takto používat mimo jiné proto, že se tím eliminují rizika s použitím speciálních znaků.[4]

Za zmínění pak stojí pojmenovávání již autorem vykonstruovaného komplexu - handlery. Pro snazší vyhledávání příslušné zprávy a jejího zpracovatele je konvence založena na přidání slova „Handler“ na konec názvu zprávy[5].

## 7.2 Dokumentace

Dokumentace je psána v rámci kódu programu a to díky **JavaDoc**[6]. Autor tímto dbá na to, aby existoval vedle interpretace v jazyce Java také slovní popis v přirozeném jazyce. Je dobrým zvykem zvolit jako tento přirozený jazyk angličtinu.

Java jako taková má vlastní systém psaní komentářů[7]. Má tedy 3 hlavní druhy psaní komen-

[3] Používají se například pro proměnné s priváním modifikátorem přístupu (například aby nebylo možné libovolně měnit obsah proměnné bez provedení kontrolní procedury)

[4] Java a vývojová prostředí by si pravděpodobně díky nastavenému kódování dovedly s českou abecedou poradit, ovšem je historickou konvencí používat co nejvíce znakovou sadu ASCII.

[5] Příkladně pak `TaskStateChange` a `TaskStateChangeHandler`

[6] Systém dokumentačních komentářů, ze kterých umí vývojová prostředí vygenerovat kompletní dokumentaci kódu.

[7] kód, který se však nekonotroluje[8] a neprovádí, slouží pouze pro komunikaci mezi autorem a čtenářem kódu

tářů - jeden pro psaní dokumentace (dokumentační komentář) a další dva pro zakomentování nějaké textové zprávy. A to buďto jednoho řádku či bloku řádků.

```
1
2  /**
3   * Dokumentacni komentar metody. Je zde vhodne popsat
4   * funkci a zpusob pouziti metody vcetne jejich parametru,
5   * pripadne jak resi nestandardni situace
6   *
7   * @param textovyParametr vstupni parametr metody
8   *
9   * @return text rozdeleny 2. znakem v poradi mezerou
10  *
11  * @throws IllegalArgumentException
12  *    kdyz pocet znaku neni alespon 3.
13  */
14 public String splitSecond(String textovyParametr) {
15
16   if(textovyParametr.length() < 3) {
17
18     throw new IllegalArgumentException("Delka musi byt alespon 3!");
19   }
20
21   String text1 = textovyParametr.substring(0, 2);
22   String text2 = textovyParametr.substring(2, textovyParametr.length()-1);
23
24   return String.format("%s %s", text1, text2);
25 }
```

Výpis kódu 7.1: Ukázka dokumentačního komentáře

Na ukázce *7.1 Ukázka dokumentačního komentáře* je vidět dokumentační komentář začínaje sekvencí znaků `/**` a konče `*/`, dále je poznat podle zelené barvy.

Javadoc je dále možné vygenerovat do komplexní dokumentace v podobě navzájem propojených HTML stránek. Toho autor využil a uložil ji do repozitáře na adrese `https://github.com/cz-vse-java-bp/bp/`. Je rozdělena do dvou celků v souladu s rozdělením do prvních dvou modulů.

## 7.3 Vývojové prostředí

K psaní kódu v jazyce Java je takřka nutností využít vhodného **IDE**[9], které slouží nejen jako vizuální podpora programování (například barevně odlišené různé důležité pojmy) či jako „síto chyb“, kdy kontroluje programátora, aby se předešlo zanášením chyb do kódu kupříkladu překlepem. U Javy totiž slouží i jako iniciátor a „vlastník“ instancí **virtuálního stroje**, jenž výsledné příkazy programu provádí.

Pro usnadnění vývoje je na trhu mnoho vývojových prostředí, která poskytují všechny výše zmíněné funkce, ba dokonce přidávají mnohé další. Předními jsou pak například **refaktorace**, či průběžná odlehčená rekompilace, jenž svede odchytat i drobné chyby v architektuře kódu. Uvažovaná prostředí jsou následující:

- **BlueJ** - Vývojové prostředí určené především pro začátečníky. Jeho nespornou výhodou je schopnost bezprostředního zobrazení architektury v odlehčeném UML diagramu.

[9] *„Integrated Development Enviroment“*, což lze přeložit jako *„Integrované vývojové prostředí“* - program sloužící k editaci kódu.

- **NetBeans** - O poznání komplexnější (a již profesionální) vývojové prostředí s pokročilými funkcemi. Existuje pro něj mnoho pluginů s cílem zjednodušit programování.
- **Eclipse** - Velmi robustní profesionální prostředí určené k vývoji nad vícero programovacími jazyky.
- **IntelliJ IDEA** - Prostředí určené pro již pokročilé programátory s podporou mnoha programovacích jazyků, s nativní podporou mnoha přístupů a paradigmat. Nespornou výhodou je snadné používání předdefinovaných funkcí, poměrně velmi inteligentní generování kódu a široká paleta customizace prostředí.

Pro potřeby této práce si autor vybral poslední zmíněné, tedy **IntelliJ IDEA** od společnosti **JetBrains**. Dále, použil následující softwarové vybavení:

- **Základní knihovny JDK** - Java Development Kit ve verzi 11.0.2. Pro samotnou tvorbu kódu nepostradatelná komponenta
- **Knihovny JavaFX** - pro definici grafického rozhraní

## 7.4 Návrhové vzory

**Návrhový vzor** je pojem z teorie programování, respektive **softawového inženýrství**, kdy abstrahujeme od konkrétnosti a na této úrovni abstrakce pak vytváříme vztah mezi objekty (rolemi). V rámci OOP pak modelujeme relace mezi třídami a objekty tak, aby nezávisely na konkrétní instanci.

**Návrhový vzor je jedním ze základních kamenů efektivního kódu psaného v paradigmatu OOP.**

V následujícím výpise jsou uvedeny potenciálně v práci nejhojněji využívané návrhové vzory.

- **Singleton** - **Jedináček**, je takový návrhový vzor, díky kterému je zaručeno, že v celém programu bude instance třídy prohlášené za Singleton pouze jednou. Libovolné části programu se pak na ni mohou odvolávat pomocí statické metody, obvykle s názvy *getInstance()* nebo *getReference()*.
- **Command** - Návrhový vzor **Příkaz** slouží k popsání vztahu mezi dvěma a více objekty, kdy jeden potřebuje provést metodu objektu skupiny v tomto vztahu podřízené. Aby bylo možné zaručit, že tyto podřízené objekty budou všechny reagovat stejně na konkrétní příkaz (zavolání metody), je nutné, aby implementovaly stejné rozhraní. Nadřízený objekt pak s nimi pracuje jako s libovolnou implementací daného rozhraní.
- **Lazy initialization** - **Odložená (líná) inicializace** je návrh, kdy se tvorba instance či provádění (zpravidla náročného) kusu kódu odloží až na moment, kdy je to doopravdy potřeba.
- **Observer** - Občas je nutné zajistit, aby se nějaká operace provedla v momentě, kdy vznikne nějaká událost. Prvním možným řešením by bylo nekonečné dotazování, zda tato situace nastala. Vhodnějším způsobem je však svým „posluchačům“ oznámit, že

se tak stalo a že mohou začít. Tomu se říká návrhový vzor **Pozorovatel** a vnitřně funguje tak, že se pozorovatel zaregistruje k odběru novinek u objektu, jemuž náleží pozorované. V momentě změny pak tento upozorní všechny své pozorovatele a ti se pak individuálně postarají o provedení svých úkonů.
- **Facade** - **Fasáda**, jehož smyslem je zjednodušeně sjednocení architektur pomocí rozhraní.

Pomocí těchto je pak tvořena architektura celého systému.

## 7.5 Kód aplikací

Samotný kód systému je přiložen v *git* repozitáři na adrese `https://github.com/cz-vse-java-bp/bp`. Rozdělení tříd do balíčků je uvedeno v dělení dle „projektů“ v přílohách, konkrétně pak v *E Výpisy seznamů tříd* počínaje stranou *117*.

# 8. Grafické rozhraní

Míníme-li tvorbu aplikace, je více než vhodné k ní přiložit i jistý ovládací prvek. Hlavními dvěma směry, kterými se v této situaci vydat jsou použití **příkazového rozhraní**[1] a **grafického rozhraní**[2]. Autor zvolil druhou možnost, tedy grafické rozhraní pro uživatele.

## 8.1 Nástroje

Pro tvorbu grafického rozhraní v Javě autor použil knihoven **JavaFX**[3]. V souvislosti s tím využil nástroje **Scene Builder** od společnosti Gluon pro snazší práci s jazykem FXML.

## 8.2 Scény

V této podkapitole autor popisuje jednotlivé scény grafického rozhraní, tedy okna s ovládacími prvky.

### 8.2.1 Přihlášení

Pro přihlášení bylo autorem navrženo jednoduché okno s možností zadání uživatelského jména a hesla. Dále pak zadání IP adresy a portu routeru, od kterého po přihlášení je možno získat informace o dalším postupu. Dalšími vstupy pak je tlačítko pro výběr souboru s Trust store, tedy certifikáty, kterým věří, včetně pole k heslu k tomuto kontejneru.[4] Návrh tohoto okna lze vidět na obrázku *8.1 Okno pro přihlášení*.

### 8.2.2 Výběr role

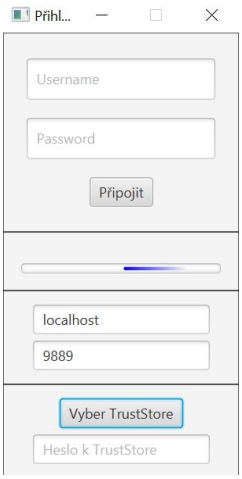
Pro výběr role uživatel musí být přihlášen. Odešle zprávu s požadavkem na role, které může zastávat, na router, jenž tato data přečte z databáze a vrátí seznam rolí definovaný ve výčto-

[1] Ovládání pomocí příkazů. Jednoduché na implementaci, velmi nízké nároky na stroj, neboť nemusí vykreslovat grafiku. Ovšem má velmi náročnou obsluhu pro uživatele. Tento přístup se dnes pro aplikace příliš nepoužívá.

[2] Zkráceně GUI; uživatelsky mnohem přívětivější, ovšem s náročnější implementací a se složitějšími výpočty kvůli vykreslování grafických prvků.

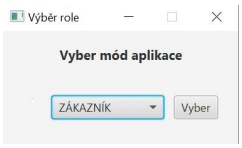
[3] dříve nebylo nutné explicitně vkládat do projektů, neboť Java knihovny pro grafické rozhraní s podporou FXML měla. Od novějších verzí však součástí není.[31, 32]

[4] Vstupy jako IP adresa, port, Trust store či heslo k němu lze vyřešit konfiguračním souborem, aby uživatel zadával pouze svou přezdívku a své heslo.




Obrázek 8.1: Okno pro přihlášení

vém typu **ERole**. Následně pak uživatel může vybrat roli, ve níž chce komunikovat a vybere tím aplikaci k obsluze.[5] Na obrázku *8.2 Okno výběru role* je návrh tohoto nástroje patrný.




Obrázek 8.2: Okno výběru role

### 8.2.3 Zákazník

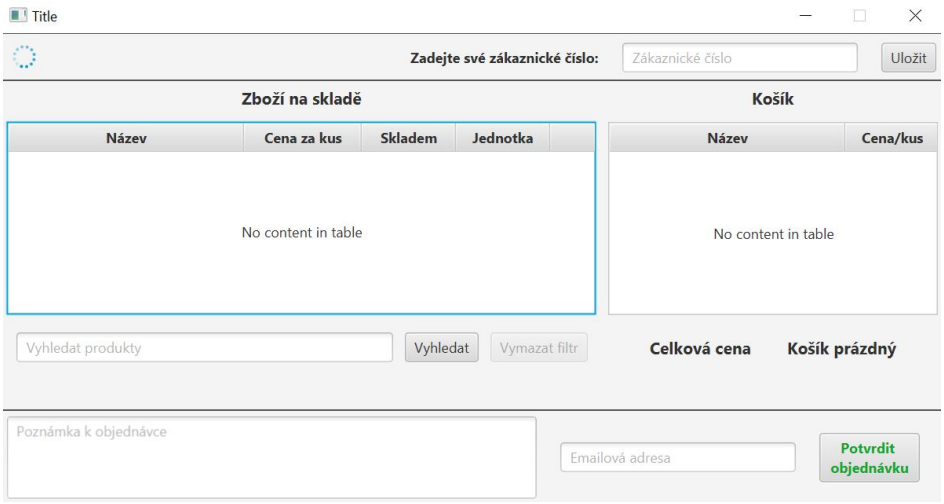
Pro zákazníka je připravena jednoduchá aplikace s možností kompletního vyhotovení objednávky. Na obrázku *8.3 Okno zákazníka* je pak tato subaplikace vidět s grafickým rozhraním. V pravé střední části může zákazník vybírat produkty, které jsou naskladněné v daném množství. V levé střední části pak vidí produkty, které má v košíku. Při rozkliknutí položky má zákazník možnost provést v daném prostředí nad touto položkou několik základních operací. V horní části panelu zákazník zadává své identifikační číslo. Tím je simulována situace, kdy zákazník je například vlastníkem věrnostní karty. Pod tabulkou naskladněného zboží je textové pole pro vyhledávání - implementováno pouze na úrovni porovnávání názvů s absolutní či relativní přesností. Lze doplnit jiný scénář filtrování produktů novým objektem implementujícím rozhraní **ISearchEngine**.

V dolní části aplikace vidíme pole pro poznámku k objednávce. Sem lze uvést doplňující požadavky, které všichni skladníci vyhotovující dílčí části objednávky uvidí. Dále je zde pole pro uvedení emailové adresy. V databázovém modelu je omezeno pouze na hodnoty typu varchar(255). V tomto okně je nastavena funkce, aby došlo k ověření validity emailové adresy pomocí autorem navrženého validátoru této adresy, zda splňuje základní stanovená pravidla

[5] Těmito je myšlena aplikace pro objednání zboží (Zákazník), pro skladníka (Skladník) či aplikaci administrátorskou.

pomocí regulárního výrazu ve třídě **EmailAddress**[6]. Použitý regulární výraz:

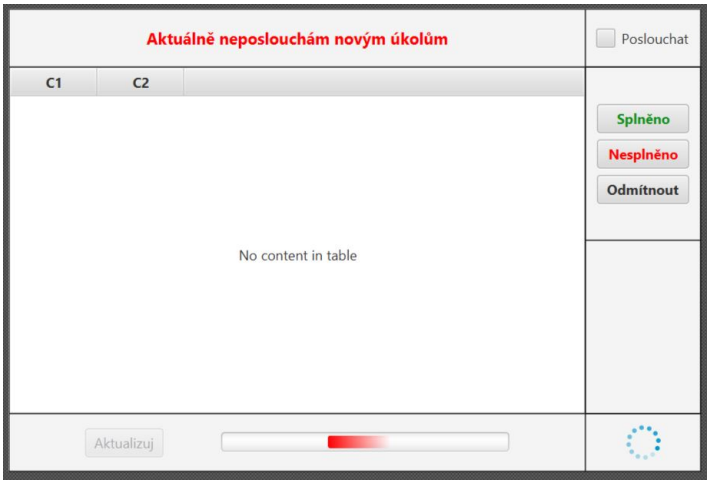
```
([0-9a-zA-Z]([-.\w]*[0-9a-zA-Z])*@([0-9a-zA-Z][-\w]*[0-9a-zA-Z]\.)+[a-zA-Z]{2,9})
```




Obrázek 8.3: Okno zákazníka

### 8.2.4 Skladník

Dalším ovládacím panelem je „sub-aplikace“ Skladník. Ta se stará o správu přidělených úkolů skladníkovi. Na obrázku *8.4 Okno skladníka* je vidět jednoduchý návrh okna, pomocí kterého lze zobrazovat a vyhotovovat úkoly.




Obrázek 8.4: Okno skladníka

Na obrázku je vidět několik jednoduchých ovládacích prvků - primárně zde máme **CheckBox** pro indikaci poslouchání. Při zaškrtnutí se odešle zpráva na server o tom, že skladník naslouchá novým úkolům. Ty se mu tam přidělí a odešlou zpět. V momentě jejich přijetí se potvrdí zpět na server a vypíší se do tabulky.

[6] Více informací v dokumentaci kódu

S vypsanými lze následně manipulovat označením a příslušným tlačítkem v panelu po pravé straně. Těmito tlačítky je možné úkol „vyhotovit“[7], prohlásit, že úkol nelze splnit[8], či ho odmítnout - tím se úkol nastaví do pozice **NEZADÁNO** a je znovu připraven k přidělení dle zvoleného **IAssignScenario** - přidělovacího scénáře[9].

V dolní části lze nalézt tlačítko **Aktualizuj**, pomocí kterého lze zařadit novou zprávu k odeslání[10], konkrétně jde o zprávu zajišťující požadavek na kompletní aktualizaci všech skladníkovi přidělených úkolů.

Obvykle je panel pro změny připraven v pravé části obrazovky.

### 8.2.5 Admin a Supervisor

Autor pro zjednodušení zvolil sloučení ovládacích funkcí rolí **Admin** a **Supervisor**. Tyto lze užít pomocí grafického rozhraní, jehož obrázky návrhu lze vidět v této podkapitole.

Grafické rozhraní pro sub-aplikaci administrativy autor rozdělil do částí, které slouží nezávisle na vybraném úkolu, a na ty, které jsou spjaty s výkonem konkrétního úkonu v daném rámci. Tyto rámce lze stanovit jako řízení:

- Zaměstnanců
- Úkolů
- Objednávek
- Skladů

Pro tyto potřeby tak okno rozdělil do dolního panelu, ve kterém lze najít především tlačítko pro aktualizaci dat, a `TabPane` z balíků `JavaFX`. V tomto kontejneru záložek jsou pak jednotlivé tyto „sub-role“ sestaveny.

Dalším specifikem této role je, že je vhodné, aby komunikovala primárně s databází, a to například pro potřeby nápravy chyb vzniklých špatnou manipulací či vznikem chyby až na službě.

#### Připojení k databázi

Tato záložka stanovuje, kam se mají nastavit přístupy v databázi. Na obrázku *8.5 Okno pro přihlášení k databázi* je možné vidět především tři instance třídy `TextField`, tedy textová pole. Do nich uživatel zadává adresu databáze, uživatelské jméno a heslo. Tlačítkem poté

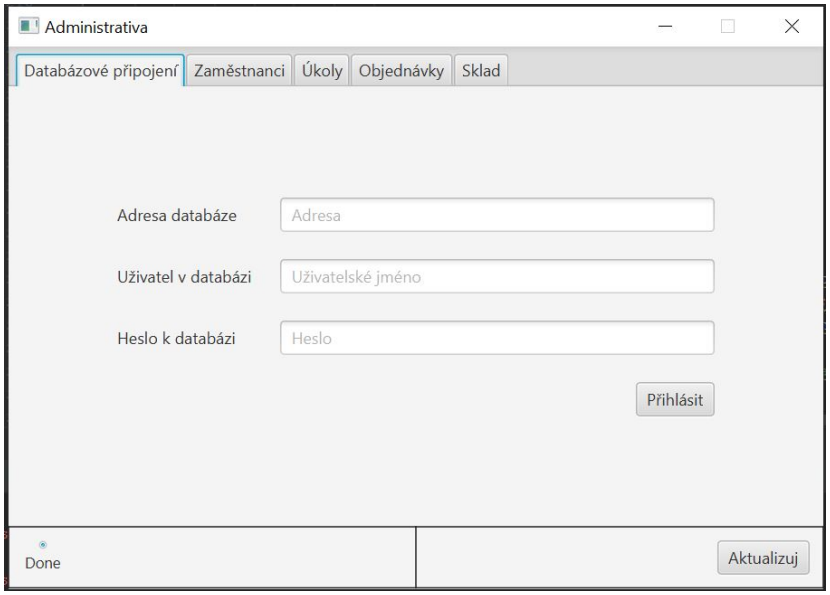
[7] V této verzi implementace se pouze odešle informace o změně stavu úkolu na **VYHOTOVENO**, kde to služba **TaskManagement** zpracuje a uloží v databázi.

[8] Odeslat na server požadavek na změnu stavu na **NEVYHOTOVENO**

[9] Defaultně je nastaveno **RandomAssign**, tedy scénář náhodného přidělení úkolu online skladníkovi. V momentě, kdy bude přihlášen pouze jeden jediný skladník, pak nebude mít tento akt přirozeně žádný pozorovatelný vliv, neboť tomuto zaměstnanci bude úkol přirozeně znovu přidělen.

[10] Pomocí instance třídy `cz.vse.java.messages.utils.future.MessageTask`

potvrdí svoje údaje, které se ověří pokusem o získání čísla ze sekvence připravené jen k tomuto úkonu.

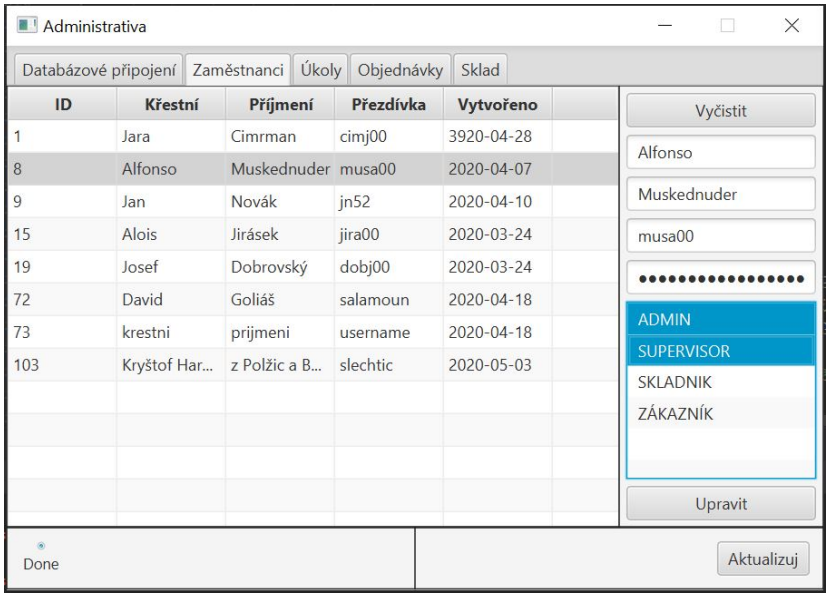



Obrázek 8.5: Okno pro přihlášení k databázi

### Správa zaměstnanců

V této části může uživatel měnit údaje o zaměstnancích. Příkladně změna hesla, jmen či rolí, které zaměstnanec zastává.

V levé části okna z obrázku *8.6 Okno pro správu zaměstnanců* lze vidět výpis zaměstnanců sestavených do tabulky typu `TableView` typu `UserSealer`, tedy třídy obalující instance entitní třídy `User` pro potřeby logiky tabulky.

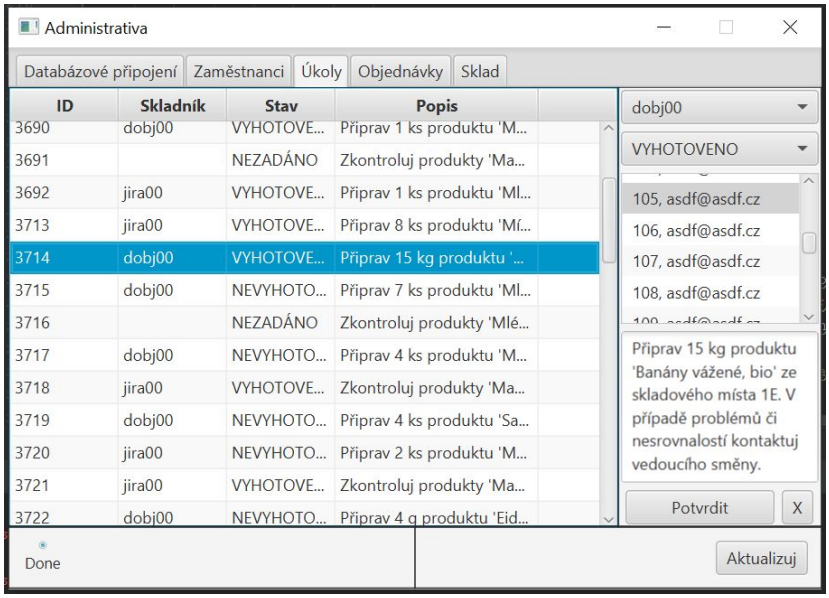



Obrázek 8.6: Okno pro správu zaměstnanců

### Správa úkolů

Tato část je jednou z kritických celého systému - správa úkolů. Tyto úkoly je nutné mít možnost měnit, tvořit a případně i mazat.

O to se stará právě záložka **Úkoly**, obsahující v levé části tabulku se všemi úkoly, a v pravé části pole pro prvky pro správu úkolů. Obrázek okna lze vidět na obrázku *8.7 Okno pro správu úkolů*.




Obrázek 8.7: Okno pro správu úkolů

### Správa objednávek

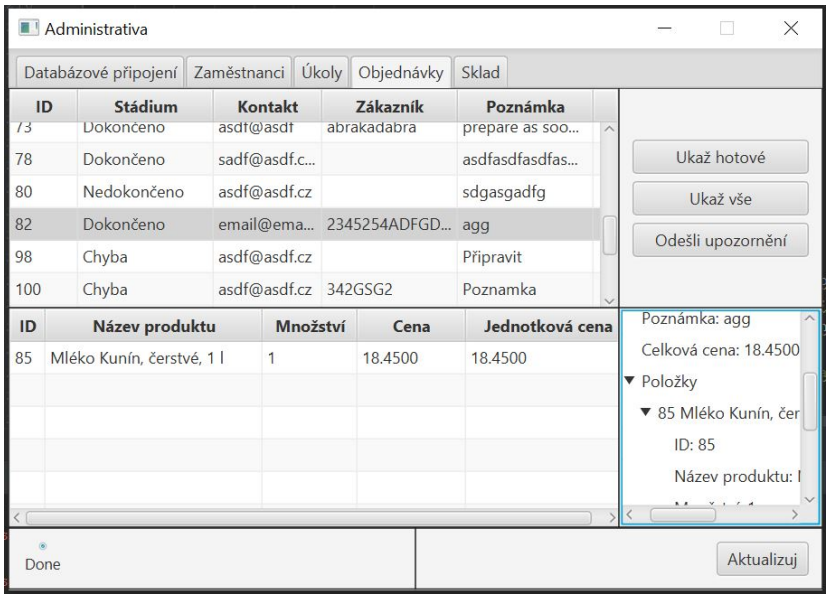
Pod kartou **Objednávky** je vidět návrh okna pro správu objednávek (viz obrázek *8.8 Okno pro správu objednávek*).

Tato záložka obsahuje především tabulku objednávek s popisem jejich stavu (ve sloupci „Stádium“), v pravé horní části pak vidíme základní prvky pro filtrování těchto úkolů („Ukaž hotové“ a „Ukaž vše“), včetně tlačítka pro odeslání emailu, které otevře správu emailů, viz podkapitola *8.2.5 Odesílání emailů* na straně *90*

V dolní části okna je další tabulka, v níž jsou uvedeny všechny položky náležící objednávce. Vedle ní je instance `TreeView` s textovou interpretací celé objednávky se všemi potřebnými atributy.

### Správa skladu

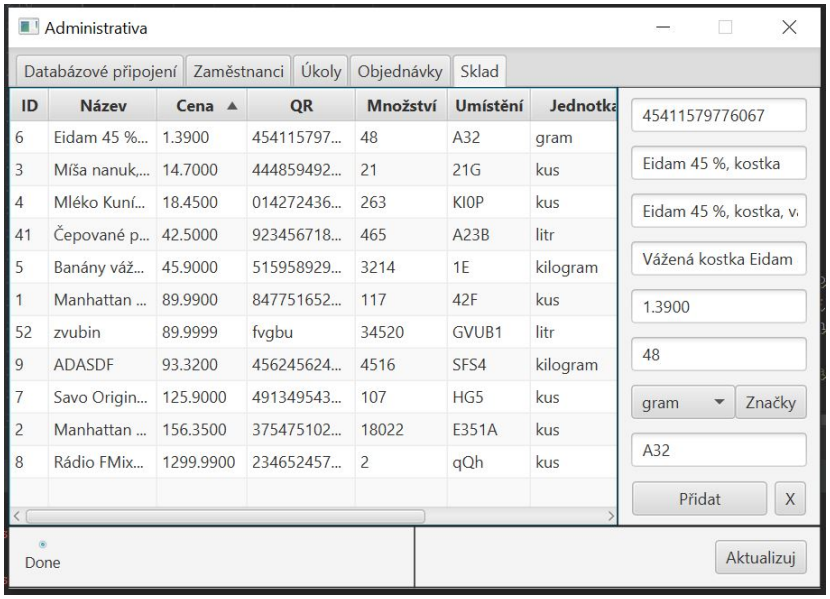
V této záložce je možné vidět především tabulka s daty o skladových položkách - produktech. Instance entitní třídy `Product` je zde převedena do obalující třídy `ProductSealer`, která se




Obrázek 8.8: Okno pro správu objednávek

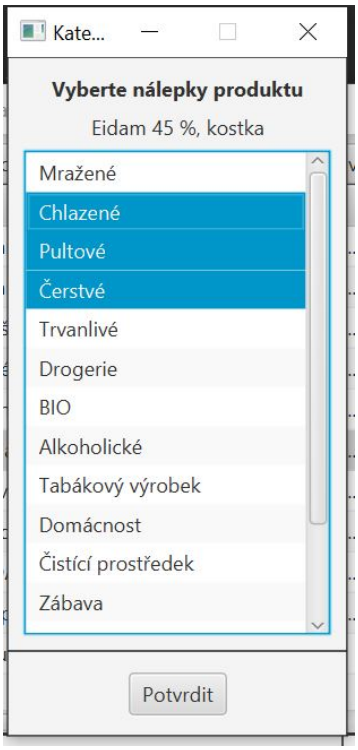
stará o správnou interpretaci dat v tabulce.

Na obrázku *8.9 Okno pro správu skladových položek* je vidět i pravá strana okna, tedy pole pro editaci. Kromě textových polí zde lze vidět i tlačítko „Značky“ otevírající pole pro výběr všech značek[11], které lze k produktu přiřadit. To je vidět na obrázku *8.10 Okno pro editaci značek*.




Obrázek 8.9: Okno pro správu skladových položek

[11] nálepek, známek či kategorií; v kódu interpretováno výčtovým typem `EProductStamp`.




Obrázek 8.10: Okno pro editaci značek

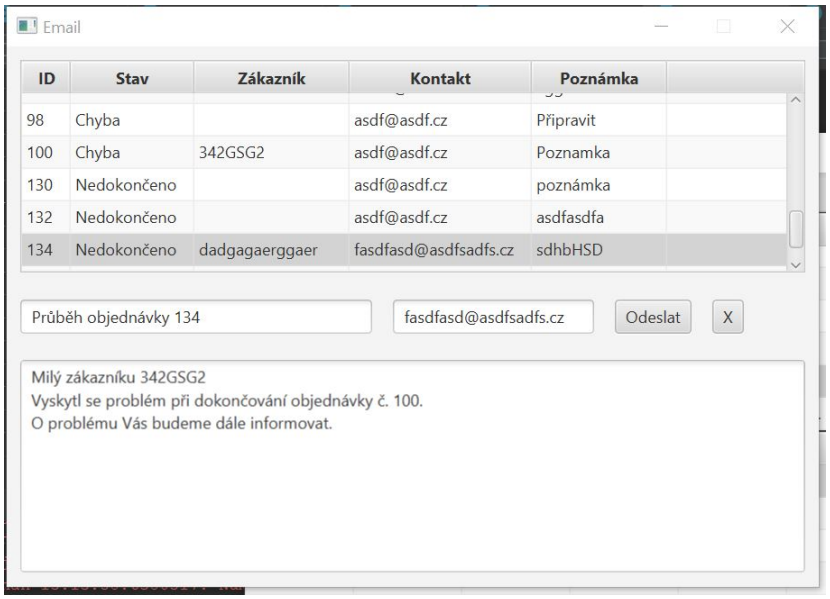
### Odesílání emailů

Pro potřeby komunikace pomocí emailu využil autor knihovny `javax.mail.mail` ve verzi `1.5.0-b01` a pseudo-smtp serveru *mailtrap.io*.

Samotné okno odesílání emailu o stavu objednávky je vidět na obrázku *8.11 Okno pro upozornění zákazníka*.

V horní části je tabulka s výpisem objednávek. V dolní části pak potřebné údaje pro odeslání emailu, tedy textová pole pro předmět emailu, pro adresu příjemce a pro obsah zprávy. Všechna tato pole jsou přirozeně automaticky vyplněna a to kliknutím na příslušnou položku z tabulky.

Pomocí tlačítka „Odeslat“ se provede odeslání emailu, pomocí tlačítka „X“ se obsah těchto polí vymaže.




Obrázek 8.11: Okno pro upozornění zákazníka

# 9. Testování softwarového produktu

Tato kapitola se zabývá průběhem testování souborů komponent, které autor navrhl. Dále pak se zabývá výsledky testování a řešením daných problémů.

Pro testování správné funkcionality bylo použito autorem následujících přístupů:

- frameworku **JUnit** pro potřeby jednotkového testování
- manuální testy testy komplexnějších funkcí pro kontrolu správného průběhu
- autorem navrženého grafického rozhraní pro potřeby testování komplexních funkcí pro kontrolu naplnění business logiky

## 9.1 Testování pomocí JUnit

*„Dostatečné pokrytí testy umožňuje pokračovat ve vývoji dalších funkcionalit bez ručního návrhu mnoha testů.“*[1] Tímto se autor snažil řídit při návrhu scénářů testování, aby se vyvaroval nadbytečnému kódu, který by musel poté složitě upravovat či dokonce mazat.

První otázkou, kterou si autor pokládal, bylo co přesně testovat pomocí JUnit. Testování jednotkovými testy má smysl jen tam, kde entita (objekt, třída či instance) může vystupovat sama za sebe.

Testovány proto byly převážně komponenty z balíku nejzákladnějších funkcí v balíku **Utils**. Návrh testů probíhal tak, aby byla protestována funkcionalita v návaznosti na smysl komponenty (třídy). Příkladem pak jsou náhodné generátory (třídy RandomStringGenerator, RandomNumberGenerator či RandomEmailAddressGenerator), dále třeba funkce hashování.

## 9.2 Manuální testy a testování v grafickém rozhraní

Testování manuálními testy a pomocí grafického rozhraní spočívalo ve zkoušení následujících funkcí navrženého grafického rozhraní a sledování výpisů z Loggeru z balíčku *java.util.logging*. Pomocí úrovní důležitosti záznamu pak bylo rozřazeno, zda jde o chybu (*Level.SEVERE*), o běžnou zprávu (*Level.INFO*) či o pouhé sdělení stavu (*Level.FINE*)

[1] V originále *„Having a high test coverage of your code allows you to continue developing features without having to perform lots of manual tests.“*[33]

| Název testu | Výsledek | Komentář |
|---|---|---|
| Přihlášení se správnými údaji - povolit | Úspěch | Uživateli umožněn přístup |
| Přihlášení se nesprávnými údaji - zakázat | Úspěch | Uživateli odepřen přístup |
| Předčasné ukončení aplikace | Úspěch | Žádná nestandardní aktivita nezaznamenána |
| Pokus o výběr role, kterou uživatel nezastává | Úspěch | Model upraven, netřeba testovat |
| Pokus o přidání do košíku zboží, které není skladem | Úspěch | Není umožněno z logiky GUI |
| Pokus o přidání záporného množství zboží do košíku | Úspěch | Není umožněno z logiky GUI |
| Pokus o potvrzení objednávky bez kontaktních údajů | Úspěch | Není umožněno z logiky GUI |
| Pokus o vypnutí aplikace před potvrzením objednávky | Úspěch | Na serveru se objednávka resetuje |
| Spuštění aplikace skladníka bez spuštění poslouchání úkolů | Úspěch | Žádné úkoly mu nejsou přiděleny |
| Spuštění aplikace skladníka se spuštěním poslouchání | Úspěch | Jsou mu přiděleny úkoly |
| Spuštění aplikace skladníka a následné vypnutí poslouchání | Úspěch | Nové úkoly mu přiděleny nejsou a stávající jsou přerozděleny |
| Náhlé vypnutí aplikace skladníka | Úspěch | Úkoly jsou přerozděleny |

## 9.3 Objevené chyby

Objevené a řešené chyby jsou v přiložené tabulce *bugs.xlsx* (viz *G Externí přílohy* počínaje stranou *124*). Testovány byly pomocí knihoven JUnit, manuálních testů a testování funkcionality pomocí zadávání příkazů v GUI.

# 10. Konfigurace a nasazení

Tato kapitola se zabývá konfigurací jednotlivých komponent, které tvoří celkový systém.

Hlavním cílem této kapitoly je osvětlení návrhu konfigurace systému pomocí konfiguračních souborů. Autor se rozhodl pro nastavení konfigurace použít jazyka XML kvůli jeho jednoduché syntaxi (čímž je vhodný a čitelný pro prakticky libovolného uživatele), díky snadnému přepisu a také kvůli široké paletě možností snadného ověření správnosti.

Autor kvůli těmto potřebám navrhl **XSD** schéma[1], které je uvedeno v příloze *D Schéma pro konfiguraci* počínaje stranou *115*. Toto schéma je schopno odchytit velké množství chyb (například překlepy), dále pak nedodržení stanovené struktury dokumentu či nesprávné hodnoty uživatelem do dokumentu zanesené. Autor neuvažoval potřebu ověření unikátnosti (tedy neduplicity) portů nutných k propojení zařízení.

V následujících podkapitolách se podívejme na dva hlavní konfigurační aspekty, které je třeba řídit.

## 10.1 Struktura konfiguračního souboru

Autor sestavil soubor pravidel konfigurace, tedy co by konfigurační soubor měl v této fázi obsahovat.

Nejdříve musíme nakonfigurovat spojení - když využíváme SSL/TLS, využíváme i certifikátů. Ty jsou uloženy ve speciálních souborech, konkrétně pak tzv. **Key Store** a **Trust Store**. Je možné je mít uloženy na libovolném místě v úložišti, ale musí k nim mít systém přístup. Dále musíme uvažovat, že tyto soubory jsou z bezpečnostních důvodů zaheslované. Proto nám stačí, aby bylo možné zadat absolutní cestu k těmto souborům včetně hesla[2]. Ty uložme do uzlu `certs`.

Dále je vhodné, aby bylo možné rozpoznávat různé druhy konfigurace. K tomu autor navrhl uzel `metaInfo`, ve kterém jsou uloženy informace o tvůrci a kdy byl soubor vytvořen.

Také zde máme potřebu, aby v konfiguračním souboru byly uvedeny informace o připojení k databázím. To nalezneme v uzlech `DBConns`, respektive `access`, kde jsou přístupy děleny dle

[1] Schéma napsané v jazyce XML za použití jmenného prostoru `http://www.w3.org/2001/XMLSchema`, které definuje strukturu samotného XML dokumentu jako datového nosiče. Dále restriktivně omezuje možné hodnoty či dokonce vztahy mezi nimi. Alternativními (či doplňujícími) řešeními by mohlo být použití technologií **DTD**, **RelaxNG** či **Schematron**. Tyto nástroje pro validaci XML souborů lze do systému přidat implementací autorem vytvořeného rozhraní `IXMLValidator` (tedy především překrytím metody `validate()`, vracející stav validace či výjimku) a nasazením obdobně jako je tomu u třídy `cz.vse.java.utils.xml.XMLSchemaValidator` v modulu `Utils`.

[2] Za normálních okolností jde o obrovskou bezpečnostní trhlinu - uložit heslo v *plain-text* podobě, to však v práci neuvažujme.

způsobu užití.

V poslední řadě konfigurační soubor musí obsahovat samotnou konfiguraci služeb, tu autor popisuje v následujících podkapitolách.

## 10.2 Konfigurace serveru s routerem

Uvažujeme-li model, kdy servererová část systému může běžet na libovolném počtu zařízení [3], pak máme dva hlavní typy instance serveru. První je ta, kde je spuštěn tento router (a souběžně mohou probíhat operace ostatních služeb), a druhou je zástupce čistě „služebních“ instancí.

Nyní se zaměřme na první zmíněnou variantu.

Stanovené schéma umožňuje konfigurovat router v rámci jednoho zařízení včetně libovolného počtu služeb. To je zajištěno stanovenou strukturou.

Příkladem konfigurace routeru lze vidět ve výpisu *10.1 Konfigurace routeru jako služby.*

```
<router>
  <routerConfig>
            <clientsPort>888</clientsPort>
            <servicePort>999</servicePort>
            <policy>
                <maxClients>10</maxClients>
                <maxServices>10</maxServices>
          </policy>
  </routerConfig>
</router>
```

Výpis kódu 10.1: Konfigurace routeru jako služby

Stanovené schéma nás limituje, že nelze nedodržet strukturu (tedy nelze měnit názvy elementů), nelze se vymanit ani omezení o velikosti čísla portů či například o maximálním počtu klientů (defaultně 1-100, včetně).

V elementech `clientsPort` a `servicePort` vidíme čísla portů, na kterých je možné se k routeru připojit. Dále zde vidíme omezení v rámci uzlu `policy`, kolik těchto spojení maximálně evidovat. Předpokládejme tedy, že nelze tuto hodnotu překročit.

Jelikož máme takto stanoveno, na který port se mají služby připojit, nemusíme již stanovovat pro každou službu tuto hodnotu zvlášť. Dále chápeme, že služby konfigurované v rámci tohoto konfiguračního souboru poběží na právě jednom zařízení, IP adresa je pak tedy automaticky stanovena na hodnotu `localhost`.

Na výpise *10.2 Konfigurace služby* zase vidíme, jak lze konfigurovat služby.

```
<service>
        <serviceName>STORAGE_MANAGEMENT</serviceName>
        <clientsPort>1010</clientsPort>
        <policy>
            <maxClients>10</maxClients>
```

[3] pomocí užití socketových připojení a topologie s notářskou službou - routerem

```
6         </policy>
7 </service>
```

Výpis kódu 10.2: Konfigurace služby

Na tomto výpise pro změnu vidíme, že služba se konfiguruje pomocí názvu služby (omezeno výčtovým typem v konfiguračním schématu, viz *D Schéma pro konfiguraci*), dále pomocí portu pro klienty a maximálním počtem klientu.

## 10.3 Konfigurace serveru bez routeru

Druhým přístupem je oproti konfiguraci serveru včetně routeru právě tento. Celé se mění přirozeně jen daty týkajícími se právě routeru. Ten není nutné konfigurovat takto precizně, neboť předpokládáme, že je konfigurován co do funkcí již na zařízení, na které budeme odkazovat[4]. Nyní nám tedy stačí pouze poukázat na adresu s portem, o zbytek se služby „bez routeru“ postarají autonomně.

Na výpise *10.3 Konfigurace routeru jako odkazu* je ukázka, jak autor tento problém s odkazováním řešil. Na tomto výpise je patrné, že je mnohem jednodušší, ovšem obsahuje všechny potřebné údaje pro spuštění služeb - IP adresu a port.

```
1  <router>
2         <connTo>
3               <ip>192.168.1.235</ip>
4               <port>999</port>
5         </connTo>
6  </router>
```

Výpis kódu 10.3: Konfigurace routeru jako odkazu

Aby bylo předejito chybným konfiguracím, je to ve vzorovém schématu definováno, že lze zvolit právě jeden z těchto přístupů a to právě jednou na celou konfiguraci. Pro služby se nic nemění, může jich být v konfiguračním souboru definováno teoreticky neomezené množství. Dále autor navrhl validaci uzlu `ip` pomocí restriktivního omezení nad typem `string` pomocí regulárního výrazu

```
(((1?[0-9]?[0-9]|2[0-4][0-9]|25[0-5])\.){3}(1?[0-9]?[0-9]|2[0-4][0-9]|25[0-5]))|(localhost)
```

který zajišťuje, že nebude možné vložit neplatnou IP adresu (konkrétně pak rozsah `0.0.0.0` až `255.255.255.255` a hodnoty `localhost`).

## 10.4 Ukázková konfigurace

Na výpise *10.4 Ukázka konfigurace serveru* na straně *98* můžeme vidět ukázku konfigurace, kterou autor při testování používal. Je však nutné si dát pozor na řádky *38* a *40*, kam je nutné uvést platnou a celou cestu k souborům Keystore a Truststore.

[4] tedy viz předchozí podkapitola a výpis *10.1 Konfigurace routeru jako služby.*

```xml
<serverConfiguration>
    <metaInfo>
        <createdBy>User</createdBy>
        <date>2020-03-09</date>
    </metaInfo>
    <router>
        <routerConfig>
            <clientsPort>888</clientsPort>
            <servicePort>999</servicePort>
            <policy>
                <maxClients>10</maxClients>
                <maxServices>10</maxServices>
            </policy>
        </routerConfig>
    </router>
    <service>
        <serviceName>STORAGE_MANAGEMENT</serviceName>
        <clientsPort>1010</clientsPort>
        <policy>
            <maxClients>10</maxClients>
        </policy>
    </service>
    <service>
        <serviceName>TASK_SERVICE</serviceName>
        <clientsPort>1111</clientsPort>
        <policy>
            <maxClients>10</maxClients>
        </policy>
    </service>
    <service>
        <serviceName>ORDER_MANAGEMENT</serviceName>
        <clientsPort>1212</clientsPort>
        <policy>
            <maxClients>10</maxClients>
        </policy>
    </service>
    <certs>
        <keyStore>...keyStore.jks</keyStore>
        <keyStorePass>changeit</keyStorePass>
        <trustStore>...trustStore.jts</trustStore>
        <trustStorePass>changeit</trustStorePass>
    </certs>
    <DBConns>
        <access>
            <use>USER_AUTHENTICATION</use>
            <url>jdbc:h2:tcp://localhost/~/test</url>
            <user>sa</user>
            <password></password>
        </access>
        <access>
            <use>FINGERPRINT_AUTHENTICATION</use>
            <url>jdbc:h2:tcp://localhost/~/test</url>
            <user>sa</user>
            <password></password>
        </access>
        <access>
            <use>EMPLOYEE_MANAGEMENT</use>
            <url>jdbc:h2:tcp://localhost/~/test</url>
            <user>sa</user>
            <password></password>
        </access>
        <access>
            <use>STORAGE_MANAGEMENT</use>
            <url>jdbc:h2:tcp://localhost/~/test</url>
            <user>sa</user>
            <password></password>
        </access>
        <access>
            <use>TASK_MANAGEMENT</use>
            <url>jdbc:h2:tcp://localhost/~/test</url>
            <user>sa</user>
            <password></password>
        </access>
        <access>
            <use>ORDERS_MANAGEMENT</use>
            <url>jdbc:h2:tcp://localhost/~/test</url>
            <user>sa</user>
            <password></password>
        </access>
    </DBConns>
</serverConfiguration>
```

Výpis kódu 10.4: Ukázka konfigurace serveru

## 10.5 Nasazení

Abychom mohli nasadit a uvést v provoz systém, musíme ho nakonfigurovat. Rozdělme toto do dvou částí - konfirugace, která ovlivňuje, jak a za jakých podmínek má být systém spuštěn, a jak má fungovat. První část máme již stanovenu v předchozích kapitolách, tedy konfigurace pomocí XML souboru.

Nyní se podívejme na spuštění. Použijemeli konfiguraci z výpisu *10.4 Ukázka konfigurace serveru*, spustí se především dva se 3 služby a router. Ty spolu musí komunikovat a musí se tedy vzájemně autentizovat. Jelikož je defaultně router nastaven, aby vyžadoval ověření pomocí instance **FingerPrint**[5], musí se nastavit data v databázi. Relevantní pak jsou entity `IP_ADDRESS_ACCESS` a `MAC_ACCESS`[6], do kterých je nutné zadat příslušná data - tedy IP adresu, MAC adresu a oboje k příslušnému typu služby, na kterou má být zařízení oprávněno se připojit.

[5] ta se ověřuje vůči datům v databázi

[6] viz *C Zdrojový kód SQL pro spuštění databáze* na straně *113*

# 11. Rozšiřitelnost platformy

V této kapitole se zabývejme rozšiřitelností systému, tedy přidáváním a případně upravováním funkcionalit.

Podíváme-li se na kapitoly *4 Specifikace procesů podniku* a *6 Návrh řešení*, pak seznáme, že celý systém funguje především jako funkční platforma zpracovávající hlavní procesy pro komunikaci mezi zákazníkem a podnikem (v podobě zaměstnanců). Nasazení tohoto systému v podobě, v jaké se tato platforma nachází by mohlo být problematické.

Autor proto volil vývoj ze strany univerzálního použití a snadného rozšíření pro snadnou customizaci funkcí a procesů, které v systému lze provádět. Neměl by být tedy existovat větší problém s přidáním těchto za účelem snazšího nasazení do již působícího podniku.

## 11.1 Přidání funkcionality

Za hlavní způsob přidání funkcionality volí autor možnost přidáním nové služby (o tuto funkci se starající). Pro rozšíření o danou službu musí vývojář provést sérii základních kroků:

0. **Vyvinout službu** - tato služba musí být přirozeně schopná sama o sobě fungovat a musí mít základních signatur, které stanovuje systém. K tomu lze využít implementaci metod z rozhraní **IService** a **Runnable**[1], dále pak děděním z abstraktní třídy **AGeneralService**. V neposlední řadě by služba měla již v konstruktoru mít nastaveny nutné handlery zpráv pro zajištění plynulého chodu aplikace. Další implementace je již na vývojáři.
1. **Registrace služby** - základním způsobem, jak registrovat službu a zajistit, že na ni bude možné správně odkazovat, je přidání „podpisu“ do výčtového typu **EServiceType**. Zde jsou uloženy jednotlivé typy služeb, pomocí kterých je jednoznačně identifikována potřeba provedení nějakého úkonu.
2. **Upravením konfiguračních souborů** - aby bylo možné službu spustit, je třeba, aby byly upraveny i spouštěcí konfigurační soubory, třídy s nimi pracující a případná schémata ověřující formální správnost dokumentů (obvykle ve formátu *XSD*).
3. **Úprava klientské části aplikace** - aby bylo možné pracovat s novou částí aplikace, musí být poskytnuty nástroje pro obsluhu. Chápejme to jako migrace handlerů a zpráv i na klientskou část aplikace. Dále je pak nutné upravit grafické rozhraní (například přidáním nové scény pro ovládání nových funkcí).
4. **Úprava databáze** - především pak přidání nových typů služeb do příslušných tabulek, rozšíření oprávnění pro připojení se ke službě a k databázi samotné.

Pomocí těchto kroků lze přidat nové funkce do stávající platformy a tím zajisit nové užití a

[1] ze základního balíku Javy - pro potřeby vláknového přístupu

prodloužit tak žitovní cyklus systému.

## 11.2 Rozšíření klienta

V momentě, kdy chceme, aby bylo možné upravit (případně rozšířit) funkce klienta, pak je nutné uvažovat nad rozsahem. Přirozenou alternativou je rozšíření na nejvyšší úrovni, tedy úprava logiky uživatelské - grafického rozhraní; a tu tímto zásahem dekomponovat na dvě podúrovně. Přirozeně toto řešení autor připouští a řešení to umožňuje.[2] Toto řešení však nemusí vývojář považovat za optimální. Je tedy možné upravit tyto třídy (respektive přidat nové) do nižších úrovní projektu, obvykle pak do třídy starající se o klienta tak, že se vyvine funkcionalita, která se pouze v rámci potřeby zakomponuje do tohoto jedináčka[3].

[2] V sub-projektu GUI (v té nejvyšší vrstvě) je uložen **JAR** soubor obsahující infrastruktury (nižší vrstvy). Lze tyto pak použít pro remodeling systému - jako knihovny.

[3] Chápej instanci po vzoru **Singleton**

# Závěr

V této kapitole autor popisuje závěrečné zhodnocení práce co do splnění cílů.

## Splnění cílů

Podíváme-li se na podkapitolu *1.2 Cíle práce*, můžeme se dočíst o stanovených cílích práce a čtenář je po přečtení schopen ohodnotit jejich naplnění. V následujících sub-sekcích autor hodnotí naplnění cíle.

### Pochopení odvětví

Autor zvolil pro zmapování důležitých úkazů odvětví a vztahů mezi jeho aktéry metodu kombinace PEST analýzy[4], SWOT matice[5], mapování hodnot, které tyto vztahy pro jednotlivé aktéry mají a za jakých podmínek, pomocí zkoumání podniku v odvětví (viz kapitola *3 Analýza prostředí* na straně *25*).

Autor dále v práci popisuje procesní náhled na business, hlouběji však a priori na zájem této práce, tedy zkoumaje vztahy mezi spotřebitelem a podnikem, respektive jeho relevantní zdroje, viz kapitola *4 Specifikace procesů podniku* na straně *37*.

Dále autor zkoumá právní předpisy, které mají na podnik významný vliv se stručným popisem jejich restriktivních omezujících faktorů či důvodu jejich zahrnutí. Toho se lze dočíst v kapitole *3.1.2 Vnější makroprostředí* počínaje stranou *27*.

### Návrh platformy

Pro vytvoření platformy autor využil co nejzabezpečenějšího řešení pro zajištění možného dlouhodobého chodu bez vážnějších bezpečnostních hrozeb. Není však v práci hlouběji uvažována možnost použití výrazně nižšího zabezpečení či dokonce žádného. Autor však k tomu poskytuje základní nástroje a rozhraní, aby v případě potřeby bylo minimalizováno nutné úsilí pro editaci.

Platforma dále aspiruje na univerzální a snadno rozšiřitelný koncept pro podnikovou komunikaci s primárním zaměřením na tu mezi spotřebitelem a zaměstnanci. Po drobnějších úpravách však lze uvažovat použití pro komunikaci s víceméně univerzálním rozměrem.

Platforma byla vyvíjena s použitím osvědčených technologií s moderními přístupy[6] pro za-

[4] v podkapitole *3.3 PEST* na straně 29
[5] v podkapitole *3.4 SWOT analýza* na straně 32
[6] Příkladně pak kombinace JDBC a perzistentní vrstvy.

jištění snadného a efektivního použití (bez nadbytečné rozbustnosti) a univerzálního použití, nezávisle na infrastruktuře IT podniku; nejrelevantněji je toto shrnuto v kapitole *6 Návrh řešení*.

### Vývoj konceptu nad platformou

Cílem bylo vytvořit konkrétní užití navržené platformy v rámci business logiky supermarketu, respektive komunikace zákazníka a podniku. Autor apeloval na stanovené pilíře a na procesní přístup. Nerozšířil však koncept o jeden z významných argumentů tohoto vztahu, tedy operativní změna ceny pomocí slevy. Autor neimplementoval v konceptu nativní podporu se dvěma odůvodněními:

1. Každý podnik v tomto odvětví má stanovena pravidla pro slevy rozdílně
2. Lze v budoucnu doimplementovat

Ze stejných důvodů není implementována nativní podpora obrázků (například fotografií produktů či uživatelů). Obě tyto kategorie lze dotvořit pomocí vytvořením nového atributu u databázových entit a drobnou úpravou tříd persistentní vrstvy.

Pro potřeby práce autor volil silně architektonický přístup pro nasazení platformy (konkrétní implementace), aby bylo zajištěno snadné rozšíření před použitím.

V práci je naplnění cíle popsáno především v kapitole *6 Návrh řešení*, soubor vytvořených tříd je uveden v příloze *E Výpisy seznamů tříd* počínaje stranou *117*.

### Testování

Posledním významným cílem se stalo testování funkcionalit, a to z pohledu zda „dělají, co mají“ a zda „to, co dělají je simulací výseče reality“.

První stranou testování prošly všechny testované entity, v případě že ne, bylo odchyceno automatizovaným testováním ve formě **JUnit** testů, případně manuálními testy. Druhou fázi autor testoval pomocí zhodnocení do jaké míry naplňuje funkce svůj cíl, neboť jde již o komplexní funkce s velkým kvantem proměnných, a jednotkový či manuální test byl použit v případě objevení nestandardního či neočekávaného chování funkce, viz kapitola *9 Testování softwarového produktu*.

Konkrétního výpisu nápravy chyb se lze dočíst v externě přiložené tabulce v souboru *bugs.xlsx*, z níž jsou vypsány chyby, se kterými se autor potýkal, stav jejich řešení a popis vyřešení.

# Literatura


[1] ROUBALOVÁ, Eliška. *Java bez předchozích znalostí.* Brno: Computer Press, 2015. ISBN 978-80-251-4572-2.

[2] PECINOVSKÝ, Rudolf. *Java 9: kompletní příručka jazyka.* Praha: Grada Publishing, 2018. Knihovna programátora (Grada). ISBN 9788027107155.

[3] LaTeX. *A document preparation system* [online]. [cit. 15. 01. 2020]. Dostupné z: `https://www.latex-project.org/help/documentation/`

[4] OLŠÁK, Petr. TEX pro pragmatiky: TEX - plainTEX - CSplain - OPmac. Brno: CSTUG, 2016. ISBN 80-901950-1-1

[5] ManagementMania.com. *Ganttův diagram (Gantt Chart).* [online]. Copyright © 2011 [cit. 03.02.2020]. Dostupné z: `https://managementmania.com/cs/ganttuv-diagram`

[6] BRUCKNER, Tomáš. *Tvorba informačních systémů: principy, metodiky, architektury.* Praha: Grada, 2012. Management v informační společnosti. ISBN 978-80-247-4153-6.

[7] Přispěvatelé Wikipedie. *Wikipedie: Otevřená encyklopedie: Don't repeat yourself* [online]. [cit. 30. 01. 2020]. Dostupné z: `https://cs.wikipedia.org/w/index.php?title=Don%27t_repeat_yourself&oldid=13034216`

[8] Oracle Česká Republika. *Co je ERP?* [online]. [cit. 04. 02. 2020]. Dostupné z: `https://www.oracle.com/cz/applications/erp/what-is-erp.html`

[9] Oracle Česká Republika. *Co je systém řízení dodavatelského řetězce?* [online]. [cit. 31.01.2020]. Dostupné z: `https://www.oracle.com/cz/applications/supply-chain-management/what-is-supply-chain-management-system.html`

[10] EUR-Lex. *NAŘÍZENÍ EVROPSKÉHO PARLAMENTU A RADY (EU) 2016/679* [online]. [cit. 20. 1. 2020]. Dostupné z: `https://eur-lex.europa.eu/legal-content/CS/TXT/?uri=CELEX:32016R0679`

[11] Česká republika. *110/2019 Sb. Zákon o zpracování osobních údajů* [zákon]. [cit. 31.01.2020]. Dostupné z: `https://www.zakonyprolidi.cz/cs/2019-110`

[12] Česká republika. *634/1992 Sb. Zákon o ochraně spotřebitele* [zákon]. [cit. 31.01.2020]. Dostupné z: `https://www.zakonyprolidi.cz/cs/1992-634`

[13] Česká republika. *90/2012 Sb. Zákon o obchodních korporacích* [zákon]. [cit. 31.01.2020]. Dostupné z: `https://www.zakonyprolidi.cz/cs/2012-90`

[14] Česká republika. *262/2006 Sb. Zákoník práce* [zákon]. [cit. 31.01.2020]. Dostupné z: `https://www.zakonyprolidi.cz/cs/2006-262`

[15] Česká republika. *181/2014 Sb. Zákon o kybernetické bezpečnosti* [zákon]. [cit. 31.01.2020]. Dostupné z: `https://www.zakonyprolidi.cz/cs/2014-181`

[16] Česká republika. *143/2001 Sb. Zákon o ochraně hospodářské soutěže* [zákon]. [cit. 31.01.2020]. Dostupné z: https://www.zakonyprolidi.cz/cs/2001-143

[17] Česká republika. *89/2012 Sb. Občanský zákoník* [zákon]. [cit. 31.01.2020]. Dostupné z: https://www.zakonyprolidi.cz/cs/2012-89

[18] Legislativa - Ochrana osobních údajů. *Úvodní strana - Ministerstvo vnitra České republiky* [online]. Copyright © 2019 Ministerstvo vnitra České republiky. Všechna práva vyhrazena. [cit. 31.01.2020]. Dostupné z: https://www.mvcr.cz/gdpr/clanek/gdpr-web-legislativa-legislativa.aspx

[19] WhatIs.com. *What is ISO 27001?* [online]. [cit. 03. 02. 2020]. Dostupné z: https://whatis.techtarget.com/definition/ISO-27001

[20] NÚKIB. *Doporučení pro případ napadení DDoS útokem - jak se zachovat a jak postupovat* [online]. [cit. 01. 02. 2020]. Dostupné z: https://www.govcert.cz/cs/informacni-servis/doporuceni/2150-doporuceni-pro-pripad-napadeni-ddos-utokem-jak-se-zachovat-a-jak-postupovat/

[21] Český statistický úřad. *S kybernetickým útokem se v roce 2018 setkaly dvě pětiny velkých firem v ČR* [online]. [cit. 27. 01. 2020]. Dostupné z: https://www.czso.cz/csu/czso/s-kybernetickym-utokem-se-v-roce-2018-setkaly-dve-petiny-velkych-firem-v-cr

[22] NÚKIB. *Zpráva o stavu kybernetické bezpečnosti za rok 2017.* Copyright © [cit. 02.02.2020]. Dostupné z: https://www.nukib.cz/download/publikace/zprava-o-stavu-kyberneticke-bezpecnosti-cr-2017.pdf

[23] Přispěvatelé Wikipedie. *Generické strategie* [online]. c2019 [cit. 02. 02. 2020]. Dostupné z: https://cs.wikipedia.org/w/index.php?title=Generick%C3%A9_strategie&oldid=16891998

[24] ManagementMania.com *Autentizace, ověření, identifikace (Authentication)* [online]. Wilmington (DE) 2011-2020, 13.02.2018 [cit. 25.03.2020]. Dostupné z: https://managementmania.com/cs/autentizace-identifikace

[25] TIOBE Software. *TIOBE Index* [online]. Copyright © 2020 TIOBE Software BV [cit. 16.02.2020]. Dostupné z: https://www.tiobe.com/tiobe-index/

[26] HANÁČEK, Petr a STAUDEK, Jan. *Bezpečnost informačních systémů: metodická příručka zabezpečování produktů a systémů budovaných na bázi informačních technologií.* Praha: Úřad pro státní informační systém, 2000. ISBN 80-238-5400-3

[27] Orcale. *Writing the Server Side of Socket.* [online]. Copyright ©. [cit. 15. 03. 2020] Dostupné z: https://docs.oracle.com/javase/tutorial/networking/sockets/clientServer.html

[28] Oracle. *Sample Code Illustrating a Secure Socket Connection Between a Client and a Server.* [online]. 13. 03. 2018. [cit. 20. 04. 2020]. Dostupné z: https://docs.oracle.co

m/javase/10/security/sample-code-illustrating-secure-socket-connection-client-and-server.htm

[29] Oracle. *JSSE Reference Guide.* [online]. [cit. 22. 02. 2020]. Dostupné z: https://docs.oracle.com/javase/8/docs/technotes/guides/security/jsse/JSSERefGuide.html

[30] VONDRÁK, Ivo. *Průvodce kurzem Metody byznys modelování: eLearning : distanční forma studia : vzdělávací řídící systém MOODLE.* Ostrava: VŠB - Technická univerzita, Centrum eLearningu VIRTUNIV, 2004. ISBN 80-248-0729-7.

[31] KRILL, Paul. *Removed from Java JDK 11, JavaFX 11 arrives as a standalone module.* [online]. 20. 9. 2018. [cit. 12.04.2020]. Dostupné z: https://www.infoworld.com/article/3305073/removed-from-jdk-11-javafx-11-arrives-as-a-standalone-module.html

[32] PALMER, Bethan. *Using JavaFX with Java 11 or Higher.* [online]. 1. 5. 2019. Dostupné z: https://blog.idrsolutions.com/2019/05/using-javafx-with-java-11/

[33] VOGEL, Lars. *Unit Testing with JUnit - Tutorial.* [online]. 21. 06. 2016, [cit. 25. 3. 2020] Dostupné z: https://www.vogella.com/tutorials/JUnit/article.html

[34] Oracle. *Getting Started with JavaFX: Using FXML to Create a User Interface | JavaFX 2 Tutorials and Documentation.* [online]. Copyright ©. [cit. 15.03.2020]. Dostupné z: https://docs.oracle.com/javafx/2/get_started/fxml_tutorial.htm

# Přílohy

## A Záznamy z odborných konzultací

V této příloze jsou uvedeny záznamy z konzultací se stávajícími či bývalými pracovníky různých pozic v rámci supermarketu.

### Konzultace s osobou P.

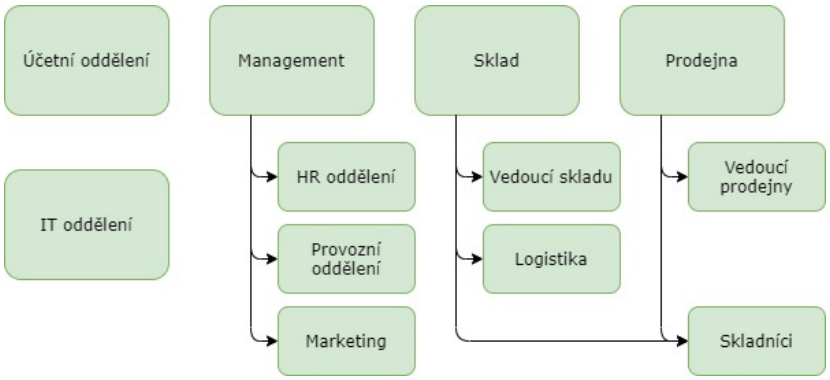
**Pozice** - Skladový logistik
**Stav** - Stávající zaměstnanec
**Řetězec** - Kaufland

**Datum konzultace** - 5. února 2020

- Prodejna si řeší a priori dodávky sama, pomocí SAP systému je plánuje. Sami se na prodejně ptají hlavního skladu, zda není nějaké zboží, které je třeba upřednostnit.
- Slevové letáky jsou řízeny na vyšší úrovni - centrálně, ale s působností na okresy, výjimečně na úrovni krajů.
- Neberou rozdíl mezi zvýhodněným a slevněným zbožím v úvahu.
- Každý produkt má na skladě pevné místo, občas je třeba ho pozměnit.
- Podnik trpí tím, že jejich systémy jsou těžko upravitelné a zároveň není dostatečně vzdělaný pracovní personál pro jeho správu a obměnu.

**Datum konzultace** - 7. března 2020

- Struktura podniku je následující:




Obrázek A.1: Nastínění organizační struktury

- Zaměstnanci často rotují, skladníci občas vykonávají několik funkcí najednou - příjem zboží, příprava zboží, dokonce i úklid.

- Skladová inventura se neprovádí pravidelně, především se provádí drobné kontroly, zda je všeho tak, jak by mělo.
- Problémy nekonzistence dat v systému oproti skutečnosti se zpravidla řeší ad hoc.
- Čerstvé zboží od lokálních dodavatelů, řádně zkontrolováno a předáno do prodejny.

## Konzultace s osobou H.

**Pozice** - Skladník
**Stav** - Bývalý zaměstnanec
**Řetězec** - Tesco

**Datum konzultace** - 18. prosince 2019

- Sklad je řízen centrálně, dodávka několikrát denně. Skladníci mají pevně stanovené pozice a nemění své role.
- Sklad je zpracováván tak, aby byly jeho kapacity optimalizovány, velmi vysoký důraz na logistiku ve skladě a minimalizace potřebných prostor.
- Personál silně proškolen k vykonávání své úlohy.
- Minimální kvalifikační nároky na pozici, často ani jazykové schopnosti (čeština).
- Typ systému pro správu skladu neznámý.
- Specifika organizační struktury neznámá.

### Konzultace s osobou D.

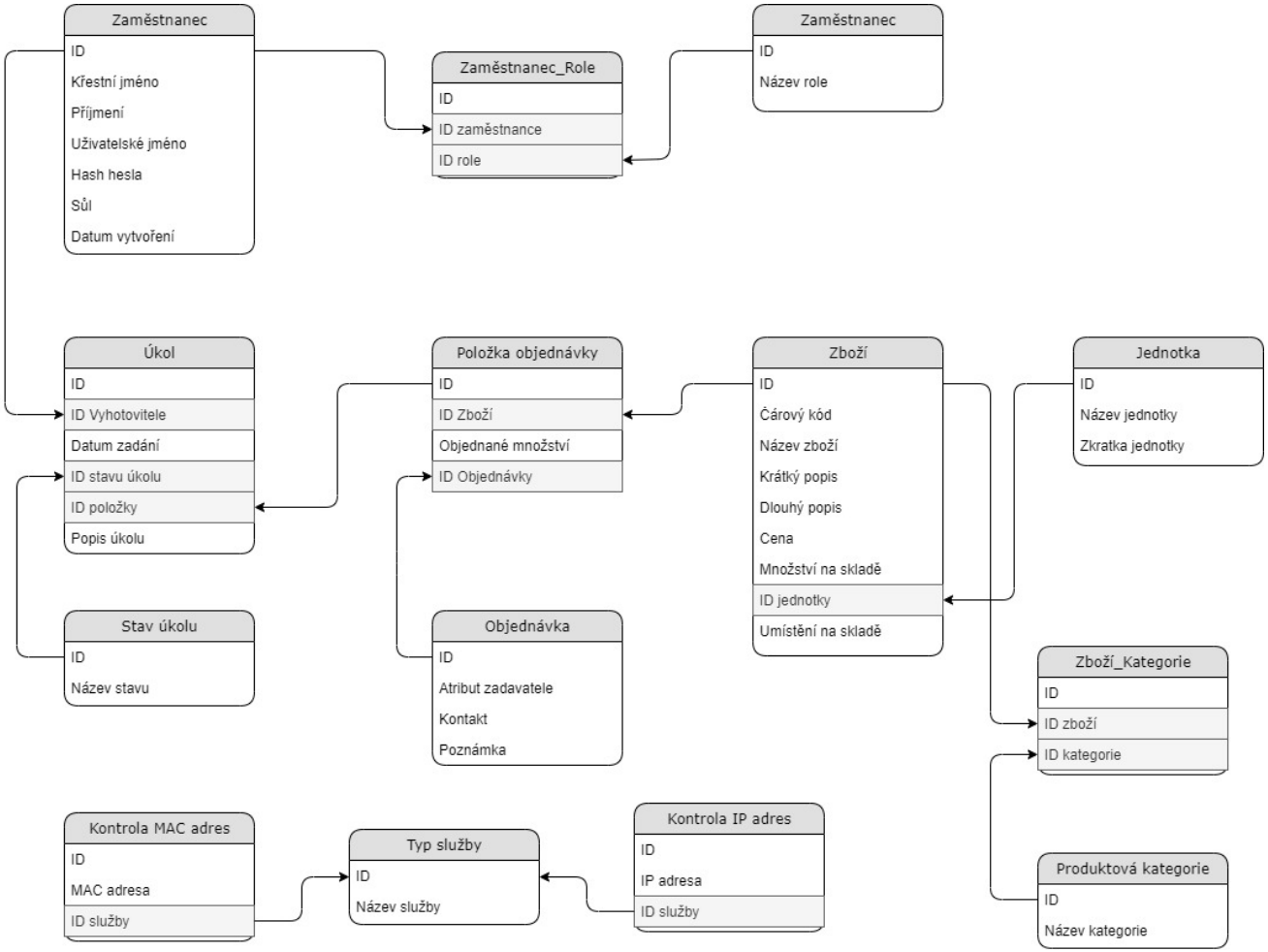
**Pozice** - Zástupce manažera provozovny
**Stav** - Bývalý zaměstnanec
**Řetězec** - Tesco

**Datum konzultace** - 20. listopadu 2019

- Náplní pozice je zajištění zdrojů pro správnou funkci prodejny.
- Organizace proškolování zaměstnanců, řízení provozních záležitostí (zpracovávání inventur, řízení HR, ...)
- Sklad i letáky jsou řízeny centrálně, na úrovni krajů a na úrovni velikosti prodejny.
- Organizační struktura je proměnlivá (prodejna od prodejny).
- Čerstvé zboží dováženo od lokálních dodavatelů, několikrát denně a většinou okamžitě převezeno na prodejnu. Pečivo je řešeno formou polotovarů.

# B Konceptuální model databáze

Na obrázku *B.1 Konceptuální model databáze* na straně *112* je vidět konceptuální model databáze na bázi entitních vztahů.




Obrázek B.1: Konceptuální model databáze

# C Zdrojový kód SQL pro spuštění databáze

V této příloze je uveden vzorový kód pro konfiguraci entit v databázi.

```
1
2   CREATE SEQUENCE TEST_CONN_SEQ;
3
4
5  CREATE TABLE EMPLOYEES (
6  ID INT PRIMARY KEY,
7  FIRSTNAME VARCHAR(255) NOT NULL,
8  LASTNAME VARCHAR(255) NOT NULL,
9  USERNAME VARCHAR(255) NOT NULL,
10 PASS_HASH VARCHAR(255) NOT NULL,
11 HASH_SALT VARCHAR(255) NOT NULL,
12 DATE_OF_CREATION TIMESTAMP NOT NULL
13 );
14
15 CREATE SEQUENCE EMPLOYEE_SEQUENCE;
16
17 create table ROLE (
18 ID bigint primary key,
19 ROLE_NAME varchar(255) not null unique,
20 );
21
22 CREATE SEQUENCE ROLE_SEQUENCE;
23
24 create table EMPLOYEES_ROLE (
25 ID bigint primary key,
26 EMP_ID bigint NOT NULL,
27 ROLE_ID bigint NOT NULL,
28
29
30 CONSTRAINT FK_EMPLOYEE FOREIGN KEY (EMP_ID) REFERENCES EMPLOYEES(ID),
31
32
33 CONSTRAINT FK_ROLE FOREIGN KEY (ROLE_ID) REFERENCES ROLE(ID)
34 );
35
36 create table IP_ADDRESS_ACCESS (
37 ID bigint primary key,
38 IP_ADDRESS varchar(255) not null,
39 ID_SERVICE bigint not null,
40
41 CONSTRAINT FK_SERVICE_TYPE FOREIGN KEY (ID_SERVICE) REFERENCES SERVICE_TYPE(ID),
42 );
43
44 CREATE SEQUENCE IP_ADDRESSES_SEQUENCE;
45 CREATE SEQUENCE SERVICE_SEQUENCE;
46 CREATE SEQUENCE MAC_SEQUENCE;
47
48
49
50 CREATE TABLE SERVICE_TYPE (
51 ID bigint primary key,
52 SERVICE_NAME varchar(255) unique not null
53 );
54
55 create table MAC_ACCESS (
56 ID bigint primary key,
57 MAC varchar(255) not null,
58 ID_SERVICE bigint not null,
59
60 CONSTRAINT FK_SERVICE_TYPE_MAC FOREIGN KEY (ID_SERVICE) REFERENCES SERVICE_TYPE(ID),
61 );
62
63 create table UNITS (
64 ID bigint primary key,
65 NAME varchar(255),
66 ABBR varchar(30)
67 );
68
69 insert into units values (1, 'kus', 'ks');
70 insert into units values (2, 'kilogram', 'kg');
71 insert into units values (3, 'gram', 'g');
72 insert into units values (4, 'litr', 'l');
73
74 create table PRODUCT (
75 ID bigint primary key,
76 BARCODE varchar(50),
77 NAME varchar(80),
```

```
78 SHORT_DESC varchar(255),
79 LONG_DESC varchar(1000),
80 PRICE numeric(10,4),
81 QUANTITY int,
82 UNIT_ID bigint,
83
84 CONSTRAINT FK_PRODUCT_UNIT FOREIGN KEY (UNIT_ID) REFERENCES UNITS(ID),
85 );
86
87 create table PRODUCT_STAMP (
88 ID bigint primary key,
89 NAME varchar(255)
90 );
91
92
93 insert into PRODUCT_STAMP values (1, 'Mražené');
94 insert into PRODUCT_STAMP values (2, 'Chlazené');
95 insert into PRODUCT_STAMP values (3, 'Pultové');
96 insert into PRODUCT_STAMP values (4, 'Čerstvé');
97
98 create table PRODUCT_TO_PRODUCT_STAMP (
99 ID bigint primary key,
100 PRODUCT_ID bigint,
101 PRODUCT_STAMP_ID bigint,
102
103 CONSTRAINT FK_PTPS_2_P FOREIGN KEY (PRODUCT_ID) REFERENCES PRODUCT(ID),
104 CONSTRAINT FK_PTPS_2_PS FOREIGN KEY (PRODUCT_STAMP_ID) REFERENCES PRODUCT_STAMP(ID),
105 );
106
107 create sequence PRODUCT_SEQ;
108
109 create sequence P2PS_SEQ start with 7;
110
111 alter table PRODUCT add LOCATION varchar(20);
112
113 CREATE SEQUENCE TASK_SEQ;
114
115 CREATE TABLE TASK_STATE(
116 ID BIGINT PRIMARY KEY,
117 STATE_NAME VARCHAR(50)
118 );
119
120 CREATE TABLE ORDERS (
121 ID BIGINT PRIMARY KEY,
122 SUBMITTER VARCHAR(255),
123 CONTACT VARCHAR(50),
124 NOTE VARCHAR(255),
125 );
126
127 CREATE TABLE ORDER_ITEM (
128 ID BIGINT PRIMARY KEY,
129 PRODUCT_ID BIGINT NOT NULL,
130 QUANTITY INT NOT NULL,
131 ORDER_ID BIGINT NOT NULL,
132
133 CONSTRAINT FK_OI_2_PRODUCT FOREIGN KEY (PRODUCT_ID) REFERENCES PRODUCT(ID),
134 CONSTRAINT FK_OI_2_ORDER FOREIGN KEY (ORDER_ID) REFERENCES ORDERS(ID)
135 );
136
137
138 CREATE TABLE TASK (
139 ID BIGINT PRIMARY KEY,
140 EMP_ID BIGINT,
141 CREATED TIMESTAMP NOT NULL,
142 TASK_STATE_ID BIGINT NOT NULL,
143 ORDER_ITEM_ID BIGINT NOT NULL,
144 DESCRIPTION VARCHAR(255),
145
146 CONSTRAINT FK_T_2_E FOREIGN KEY (EMP_ID) REFERENCES EMPLOYEES(ID),
147 CONSTRAINT FK_T_2_TS FOREIGN KEY (TASK_STATE_ID) REFERENCES TASK_STATE(ID),
148 CONSTRAINT FK_T_2_ORDER_ITEM FOREIGN KEY (ORDER_ITEM_ID) REFERENCES ORDER_ITEM(ID)
149 );
150
151
152 CREATE SEQUENCE ORDER_SEQ;
153 CREATE SEQUENCE ORDER_ITEM_SEQ;
154
155
```

Výpis kódu C: SQL pro vytvoření tabulek a dalších objektů v databázi

# D Schéma pro konfiguraci

V této příloze je přiložen zdrojový kód vzorového XSD schématu určeného pro kontrolu uživatelského vstupu před spuštěním serveru.

```
<?xml version = "1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

    <xs:element name="serverConfiguration" type="confType" />

    <xs:complexType name="confType">
        <xs:sequence>
            <xs:element name="metaInfo" type="metaType" />
            <xs:element name="router" type="routerType" />
            <xs:element name="service" type="serviceType" maxOccurs="unbounded" />
            <xs:element name="certs" type="certsType" />
            <xs:element name="DBConns" type="dbConnsType" minOccurs="0"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="metaType">
        <xs:sequence>
            <xs:element name="createdBy" type="xs:string" />
            <xs:element name="date" type="xs:date" />
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="routerType">
        <xs:choice>
            <xs:element name="routerConfig" type="routerConfig" />
            <xs:element name="connTo" type="connToType" />
        </xs:choice>
    </xs:complexType>


    <xs:complexType name="routerConfig">
        <xs:sequence>
            <xs:element name="clientsPort" type="portType" />
            <xs:element name="servicePort" type="portType" />
            <xs:element name="policy" type="policyType" />
        </xs:sequence>
    </xs:complexType>


    <xs:complexType name="connToType">
        <xs:sequence>
            <xs:element name="ip" type="ipType" />
            <xs:element name="port" type="portType" />
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="serviceType">
        <xs:sequence>
            <xs:element name="serviceName" type="serviceNameType" />
            <xs:element name="clientsPort" type="portType" />
            <xs:element name="policy" type="policyType" />
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="certsType">
        <xs:sequence>
            <xs:element name="keyStore" type="xs:string" />
            <xs:element name="keyStorePass" type="xs:string" />
            <xs:element name="trustStore" type="xs:string" minOccurs="0"/>
            <xs:element name="trustStorePass" type="xs:string" minOccurs="0"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="policyType">
        <xs:sequence>
            <xs:element name="maxClients" type="maxConns" />
            <xs:element name="maxServices" type="maxConns" minOccurs="0"/>
        </xs:sequence>
    </xs:complexType>

    <xs:simpleType name="maxConns">
        <xs:restriction base="xs:integer">
            <xs:minExclusive value="0" />
            <xs:maxInclusive value="100" />
        </xs:restriction>
```

```
76      </xs:simpleType>
77
78      <xs:simpleType name="portType">
79          <xs:restriction base="xs:integer">
80              <xs:minExclusive value="0" />
81              <xs:maxInclusive value="65535" />
82          </xs:restriction>
83      </xs:simpleType>
84
85      <xs:simpleType name="ipType">
86          <xs:restriction base="xs:string">
87              <xs:pattern value="(((1?[0-9]?[0-9]|2[0-4][0-9]|25[0-5])\.){3}(1?[0-9]?[0-9]|2[0-4][0-9]|25[0-5]))|(
     localhost)" />
88          </xs:restriction>
89      </xs:simpleType>
90
91      <xs:simpleType name="serviceNameType">
92          <xs:restriction base="xs:string">
93              <xs:enumeration value="EMPLOYEE_MANAGEMENT" />
94              <xs:enumeration value="TEXT_PRINTER" />
95              <xs:enumeration value="STORAGE_MANAGEMENT" />
96              <xs:enumeration value="TASK_SERVICE" />
97              <xs:enumeration value="ORDER_MANAGEMENT" />
98          </xs:restriction>
99      </xs:simpleType>
100
101     <xs:complexType name="dbConnsType">
102         <xs:sequence>
103             <xs:element name="access" type="accessType" maxOccurs="unbounded"/>
104         </xs:sequence>
105     </xs:complexType>
106
107     <xs:complexType name="accessType">
108         <xs:sequence>
109             <xs:element name="use" type="dbUseType" />
110             <xs:element name="url" type="xs:string" />
111             <xs:element name="user" type="xs:string" />
112             <xs:element name="password" type="xs:string" />
113         </xs:sequence>
114     </xs:complexType>
115
116     <xs:simpleType name="dbUseType">
117         <xs:restriction base="xs:string">
118             <xs:enumeration value="USER_AUTHENTICATION" />
119             <xs:enumeration value="FINGERPRINT_AUTHENTICATION" />
120             <xs:enumeration value="EMPLOYEE_MANAGEMENT" />
121             <xs:enumeration value="STORAGE_MANAGEMENT" />
122             <xs:enumeration value="TASK_MANAGEMENT" />
123             <xs:enumeration value="ORDERS_MANAGEMENT" />
124         </xs:restriction>
125     </xs:simpleType>
126
127 </xs:schema>
```

Výpis kódu D: Vzorová ukázka XSD schématu pro kontrolu konfiguračního souboru serveru

# E Výpisy seznamů tříd

V těchto výpisech jsou uvedené názvy tříd (v rámci balíčků) autorem naprogramovaných, rozdělených do modulů.

## E.1 Modul Utils

- `cz.vse.java.utils.device.LocalDeviceProp`
- `cz.vse.java.utils.emailServices.EmailAddress`
- `cz.vse.java.utils.emailServices.EmailsContainer`
- `cz.vse.java.utils.emailServices.XMLEmailContainer`
- `cz.vse.java.utils.lists.IRestrictionsList`
- `cz.vse.java.utils.lists.ListingMethod`
- `cz.vse.java.utils.lists.stringLists.IStringRestrictionsList`
- `cz.vse.java.utils.lists.stringLists.StringBlackList`
- `cz.vse.java.utils.lists.stringLists.StringWhiteList`
- `cz.vse.java.utils.random.Charsets`
- `cz.vse.java.utils.random.RandomEmailAddressGenerator`
- `cz.vse.java.utils.random.RandomNumberGenerator`
- `cz.vse.java.utils.random.RandomStringGenerator`
- `cz.vse.java.utils.random.RegexBasedStringGenerator`
- `cz.vse.java.utils.xml.IXMLReader`
- `cz.vse.java.utils.xml.IXMLValidator`
- `cz.vse.java.utils.xml.IXMLWriter`
- `cz.vse.java.utils.xml.ListOfNodes`
- `cz.vse.java.utils.xml.XMLSchemaValidator`

## E.2 Modul Connections

Z tohoto výpisu tříd je patrné, že největší podíl (co do množství) mají třídy z balíčků `cz.vse.java.messages` a `cz.vse.java.handlers`. Je to dáno tím, že jinak je platforma víceméně univerzální, v závislosti právě na logice těchto zpráv a těchto jejich zpracovatelů.

- `cz.vse.java.App`
- `cz.vse.java.authentication.AuthenticationHandlerContainer`
- `cz.vse.java.authentication.EAuthenticationScenarioType`
- `cz.vse.java.authentication.FingerPrintDBAuthenticator`
- `cz.vse.java.authentication.IAuthenticationScenario`
- `cz.vse.java.authentication.IPDBAuthenticator`
- `cz.vse.java.authentication.TokenContainer`
- `cz.vse.java.authentication.TokenValidator`
- `cz.vse.java.authentication.UserAuthenticator`
- `cz.vse.java.connections.clientSide.C2RConnection`
- `cz.vse.java.connections.clientSide.C2SConnection`
- `cz.vse.java.connections.routerSide.R2CConnection`
- `cz.vse.java.connections.routerSide.R2SConnection`
- `cz.vse.java.connections.serviceSide.S2CConnection`
- `cz.vse.java.connections.serviceSide.S2RConnection`
- `cz.vse.java.connections.utils.IConnection`

- cz.vse.java.connections.utils.ICSConnection
- cz.vse.java.connections.utils.ISSConnection
- cz.vse.java.connections.utils.ISSConnectionManager
- cz.vse.java.connections.utils.management.AConnectionManager
- cz.vse.java.connections.utils.management.AConnectionWithRouter
- cz.vse.java.connections.utils.management.ClientRouterManagement
- cz.vse.java.connections.utils.management.ClientServiceManagement
- cz.vse.java.connections.utils.management.RouterClientsManagement
- cz.vse.java.connections.utils.management.ServiceClientsManagement
- cz.vse.java.connections.utils.management.ServiceRouterManagement
- cz.vse.java.connections.utils.management.ServicesManagement
- cz.vse.java.connections.utils.problemSolvers.CannotAcceptMessage
- cz.vse.java.connections.utils.problemSolvers.CannotAuthenticate
- cz.vse.java.connections.utils.problemSolvers.CannotSendMessage
- cz.vse.java.connections.utils.problemSolvers.ConnectionInterrupted
- cz.vse.java.connections.utils.problemSolvers.NoDBRecordFound
- cz.vse.java.connections.utils.problemSolvers.NullMessageContent
- cz.vse.java.connections.utils.problemSolvers.ServiceReferenceNotFound
- cz.vse.java.connections.utils.problemSolvers.WrongFingerPrint
- cz.vse.java.connections.utils.problemSolvers.WrongMessage
- cz.vse.java.connections.utils.problemSolvers.WrongPassword
- cz.vse.java.connections.utils.problemSolvers.WrongServiceType
- cz.vse.java.connections.utils.problemSolvers.WrongToken
- cz.vse.java.connections.utils.problemSolvers.WrongUsername
- cz.vse.java.connections.utils.problemSolvers.utils.AProblemSolver
- cz.vse.java.connections.utils.problemSolvers.utils.ESolveMethod
- cz.vse.java.connections.utils.problemSolvers.utils.IProblemSolver
- cz.vse.java.connections.utils.problemSolvers.utils.ProblemSolverContainer
- cz.vse.java.handlers.AddPreOrderItemHandler
- cz.vse.java.handlers.AddTaskMessageHandler
- cz.vse.java.handlers.AskForTaskCommandHandler
- cz.vse.java.handlers.AuthResultContainerHandler
- cz.vse.java.handlers.ChangeProductQuantityHandler
- cz.vse.java.handlers.ErrorMessageHandler
- cz.vse.java.handlers.FingerPrintContainerRequestHandler
- cz.vse.java.handlers.GiveMeMyOrderHandler
- cz.vse.java.handlers.GiveMeRolesHandler
- cz.vse.java.handlers.ListeningForTasksContainerHandler
- cz.vse.java.handlers.OrderContainerMessageHandler
- cz.vse.java.handlers.OrderTransformationResultHandler
- cz.vse.java.handlers.PasswordRequestHandler
- cz.vse.java.handlers.PreOrderChangeResultHandler
- cz.vse.java.handlers.PreOrderContainerMessageHandler
- cz.vse.java.handlers.ProductAllContainerHandler
- cz.vse.java.handlers.ProductAllRequestHandler
- cz.vse.java.handlers.ProductByIDContainerHandler
- cz.vse.java.handlers.ProductByIDRequestHandler
- cz.vse.java.handlers.QuitMessageHandler
- cz.vse.java.handlers.RolesContainerMessageHandler
- cz.vse.java.handlers.ServiceReferenceContainerHandler
- cz.vse.java.handlers.ServiceReferenceRequestHandler
- cz.vse.java.handlers.ServiceTypeContainerHandler
- cz.vse.java.handlers.ServiceTypeRequestHandler

- cz.vse.java.handlers.SetContactToOrderMessageHandler
- cz.vse.java.handlers.SetNoteToOrderMessageHandler
- cz.vse.java.handlers.SetSubmitterToOrderHandler
- cz.vse.java.handlers.TaskStateChangeHandler
- cz.vse.java.handlers.TextMessageHandler
- cz.vse.java.handlers.TokenRequestHandler
- cz.vse.java.handlers.TryToGenerateOrderHandler
- cz.vse.java.handlers.UniqueOrderIdentRequestHandler
- cz.vse.java.handlers.UsePreOrderIdentHandler
- cz.vse.java.handlers.UserNameRequestHandler
- cz.vse.java.handlers.UseTokenHandler
- cz.vse.java.handlers.utils.AHandler
- cz.vse.java.handlers.utils.HandlerContainer
- cz.vse.java.handlers.utils.IHandler
- cz.vse.java.messages.AddPreOrderItem
- cz.vse.java.messages.AddTaskMessage
- cz.vse.java.messages.AskForTaskCommand
- cz.vse.java.messages.AuthenticationResultContainer
- cz.vse.java.messages.AuthMeMessage
- cz.vse.java.messages.ChangeProductQuantity
- cz.vse.java.messages.ErrorMessage
- cz.vse.java.messages.FingerPrintContainerMessage
- cz.vse.java.messages.FingerPrintContainerRequest
- cz.vse.java.messages.GiveMeMyOrder
- cz.vse.java.messages.GiveMeRoles
- cz.vse.java.messages.IPAddressContainer
- cz.vse.java.messages.ListeningForTasksContainer
- cz.vse.java.messages.OrderContainerMessage
- cz.vse.java.messages.OrderTransformationResult
- cz.vse.java.messages.PasswordContainerMessage
- cz.vse.java.messages.PasswordRequest
- cz.vse.java.messages.PreOrderChangeResult
- cz.vse.java.messages.PreOrderContainerMessage
- cz.vse.java.messages.ProductAllContainer
- cz.vse.java.messages.ProductAllRequest
- cz.vse.java.messages.ProductByIDContainer
- cz.vse.java.messages.ProductByIDRequest
- cz.vse.java.messages.QuitMessage
- cz.vse.java.messages.RolesContainerMessage
- cz.vse.java.messages.ServiceReferenceContainer
- cz.vse.java.messages.ServiceReferenceRequest
- cz.vse.java.messages.ServiceTypeContainer
- cz.vse.java.messages.ServiceTypeRequest
- cz.vse.java.messages.SetContactToOrderMessage
- cz.vse.java.messages.SetNoteToOrderMessage
- cz.vse.java.messages.SetSubmitterToOrder
- cz.vse.java.messages.TaskStateChange
- cz.vse.java.messages.TextMessage
- cz.vse.java.messages.TokenContainerMessage
- cz.vse.java.messages.TokenRequest
- cz.vse.java.messages.TryToGenerateOrder
- cz.vse.java.messages.UniqueOrderIdentRequest
- cz.vse.java.messages.UsePreOrderIdent

- cz.vse.java.messages.UserNameContainerMessage
- cz.vse.java.messages.UserNameRequest
- cz.vse.java.messages.UseToken
- cz.vse.java.messages.utils.AMessage
- cz.vse.java.messages.utils.EErrorType
- cz.vse.java.messages.utils.ICommand
- cz.vse.java.messages.utils.IDataContainer
- cz.vse.java.messages.utils.IMessage
- cz.vse.java.messages.utils.IRequest
- cz.vse.java.messages.utils.future.MessageTask
- cz.vse.java.messages.utils.future.MessageTaskContainer
- cz.vse.java.messages.utils.past.ReceivedMessageContainer
- cz.vse.java.services.clientSide.Client
- cz.vse.java.services.clientSide.ISearchingEngine
- cz.vse.java.services.clientSide.ProductNameSearch
- cz.vse.java.services.clientSide.ProductsContainer
- cz.vse.java.services.references.EReferenceFor
- cz.vse.java.services.references.ServiceReference
- cz.vse.java.services.references.ServiceReferenceContainer
- cz.vse.java.services.references.SRCContainer
- cz.vse.java.services.serverSide.AGeneralService
- cz.vse.java.services.serverSide.AService
- cz.vse.java.services.serverSide.EmployeeManagement
- cz.vse.java.services.serverSide.EServiceType
- cz.vse.java.services.serverSide.IService
- cz.vse.java.services.serverSide.OrderManagement
- cz.vse.java.services.serverSide.Router
- cz.vse.java.services.serverSide.Server
- cz.vse.java.services.serverSide.StorageService
- cz.vse.java.services.serverSide.TaskManagement
- cz.vse.java.services.serverSide.TextPrinterService
- cz.vse.java.services.serverSide.config.ServerConfiguration
- cz.vse.java.services.serverSide.config.ServiceConfiguration
- cz.vse.java.utils.ConnectionAutoCloser
- cz.vse.java.utils.FingerPrint
- cz.vse.java.utils.SSLClientConfigManager
- cz.vse.java.utils.SSLServerConfigManager
- cz.vse.java.utils.Token
- cz.vse.java.utils.database.DatabaseConnectionContainer
- cz.vse.java.utils.database.DBConnection
- cz.vse.java.utils.database.EDBUse
- cz.vse.java.utils.database.EmployeeDBConnection
- cz.vse.java.utils.observerDP.IObserver
- cz.vse.java.utils.observerDP.ISubject
- cz.vse.java.utils.persistance.Storage
- cz.vse.java.utils.persistance.entities.EProductStamp
- cz.vse.java.utils.persistance.entities.EUnit
- cz.vse.java.utils.persistance.entities.IEntity
- cz.vse.java.utils.persistance.entities.OrderItem
- cz.vse.java.utils.persistance.entities.Product
- cz.vse.java.utils.persistance.entities.User
- cz.vse.java.utils.persistance.entities.orders.Order
- cz.vse.java.utils.persistance.entities.orders.OrdersContainer

- cz.vse.java.utils.persistance.entities.orders.PreOrder
- cz.vse.java.utils.persistance.entities.orders.PreOrderItem
- cz.vse.java.utils.persistance.entities.tasks.ETaskState
- cz.vse.java.utils.persistance.entities.tasks.ISideTaskAssigner
- cz.vse.java.utils.persistance.entities.tasks.RandomCheckSideTask
- cz.vse.java.utils.persistance.entities.tasks.Task
- cz.vse.java.utils.persistance.entities.tasks.TaskContainer
- cz.vse.java.utils.persistance.entities.tasks.TaskDescriptionGenerator
- cz.vse.java.utils.persistance.service.AEntityService
- cz.vse.java.utils.persistance.service.IPersistor
- cz.vse.java.utils.persistance.service.OrderItemService
- cz.vse.java.utils.persistance.service.OrderService
- cz.vse.java.utils.persistance.service.ProductService
- cz.vse.java.utils.persistance.service.TaskService
- cz.vse.java.utils.persistance.service.UserService
- cz.vse.java.utils.userData.ERole
- cz.vse.java.utils.userData.UserProperties
- cz.vse.java.utils.userTaskAssignment
- cz.vse.java.utils.userTaskAssignment.IAssignScenario
- cz.vse.java.utils.userTaskAssignment.RandomAssign
- cz.vse.java.utils.userTaskAssignment.TaskSolver
- cz.vse.java.utils.userTaskAssignment.TaskSolverContainer

## E.3 Modul GUI

V této části je uveden výpis tříd autorem vytvořených pro potřeby správy GUI.

- cz.vse.java.mode.ModePicker
- cz.vse.java.Launcher
- cz.vse.java.orders.Orders
- cz.vse.java.tasks.Tasks
- cz.vse.java.admin.OrderEmailForm
- cz.vse.java.admin.StampsPicker
- cz.vse.java.admin.Admin
- cz.vse.java.tasks.TaskFormat
- cz.vse.java.sign.SignGuiController
- cz.vse.java.App
- cz.vse.java.orders.ProductDetail

# F Použité cizí dependencies

V této příloze je uveden výpis cizích dependencies (závislostí)[7], které autor v práci použil. Pro snazší orientaci autor tyto vypisuje v podobě, jak je do modulů zařazoval[8].

Ve všech modulech je importován defaultně následující kód, přidávající závislost knihoven JUnit, viz výpis *F Defaultní import knihovny JUnit*

```
1
2 <dependency>
3     <groupId>junit</groupId>
4     <artifactId>junit</artifactId>
5     <version>4.11</version>
6     <scope>test</scope>
7 </dependency>
```

Výpis kódu F: Defaultní import knihovny JUnit

Na výpise je dále vidět struktura zadávání těchto importů - „organizace“ vydávající rozšíření, název artefaktu, dále verze artefaktu a rámec užití, v tomto případě testování.

## F.1 Grafické rozhraní

Pro definici grafického rozhraní autor použil knihoven JavaFX od vydavatele *Gluon*. Konkrétní balíky pak jsou uvedeny ve výpise *F Knihovny pro definici grafického rozhraní*.

```
1     <dependency>
2         <groupId>org.openjfx</groupId>
3         <artifactId>javafx-controls</artifactId>
4         <version>11</version>
5     </dependency>
6     <dependency>
7         <groupId>org.openjfx</groupId>
8         <artifactId>javafx-graphics</artifactId>
9         <version>11</version>
10    </dependency>
11    <dependency>
12        <groupId>org.openjfx</groupId>
13        <artifactId>javafx-fxml</artifactId>
14        <version>11</version>
15    </dependency>
16    <dependency>
17        <groupId>org.openjfx</groupId>
18        <artifactId>javafx-media</artifactId>
19        <version>11</version>
20    </dependency>
```

Výpis kódu F: Knihovny pro definici grafického rozhraní

Na tomto výpise je vidět import knihoven pro ovládací prvky, grafiku, pro podporu jazyka FXML a pro média.

Tyto závislosti jsou použity v modulu **Connections**.

[7] Chápejme jako knihovny

[8] Pomocí nástroje *Apache Maven*; oficiální web: `http://maven.apache.org/`

## F.2 Mailing

Zde je uvedena závislost pro potřeby odesílání mailů.

```
    <dependency>
         <groupId>javax.mail</groupId>
         <artifactId>mail</artifactId>
         <version>1.5.0-b01</version>
     </dependency>
```

Výpis kódu F: Knihovna pro správu odesílání emailů

Tato závislost je použita v modulu **GUI**.

## F.3 Komunikace s databází

Pro komunikaci s databází je v práci užito knihovny pro práci s **H2** databází, viz výpis *F Knihovna pro komunikaci s databází*.

```
   <dependency>
     <groupId>com.h2database</groupId>
     <artifactId>h2</artifactId>
     <version>1.4.200</version>
   </dependency>
```

Výpis kódu F: Knihovna pro komunikaci s databází

Tato závislost je použita v modulu **Connections**.

# G Externí přílohy

V této příloze je uveden výpis příloh, které svojí povahou či rozsahem nelze zařadit přímo do práce, ovšem jsou součástí. Na následujícím seznamu jsou uvedeny tyto přílohy.

- **Kód aplikací** - Samotný kód, který autor vytvořil pro naplnění cílů této práce. Ten je uložen v *Github* repozitáři dostupném na adrese `https://github.com/cz-vse-java-bp/bp/`.
- **Tabulka objevených a řešených chyb** - Tabulka s výpisem objevených chyb, způsobem objevení, jejich stavu a jejich způsobu řešení. Ta byla odevzdána s touto prací do školního informačního systému jako příloha.
- **Dokumentace** - Vygenerovaná JavaDoc dokumentace prvních dvou modulů, ta je uložena v *Github* repozitáři dostupném na adrese `https://github.com/cz-vse-java-bp/bp/`.