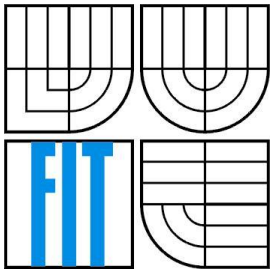


FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

MODELOVÁNÍ PROCESŮ V JAZYCE PYTHON

SEMINÁRNÍ PRÁCE - VYBRANÉ PROBLÉMY INFORMAČNÍCH SYSTÉMŮ

AUTOR PRÁCE Ing. LUKÁŠ MÁČEL

BRNO 2010

Obsah

1 Úvod

V době psaní této práce existuje hned několik standardů, jejichž cílem je přesně definovaná specifikace modelu procesu. Některé ze standardů se zaměřují především na co nejjednodušší a dobře pochopitelné zachycení řídící logiky procesu a následnou přenositelnost takového popisu (BPMN, UML), jiné jsou navrženy tak, aby podporovaly přímou interpretaci popisu procesu pomocí workflow engine (BPEL, XPDL). Podle mého názoru zde ale stále chybí modelovací jazyk, který by na jedné straně dovolil vytvořit přehledný a dobře udržovatelný popis i komplexních procesů a na druhé straně umožnil jeho přímou interpretaci pomocí workflow engine bez nutnosti další transformace. V následujících odstavcích se pokusím uvést problémy, které vidím na soudobých prostředcích pro modelování procesů.

Diagramy popsané v notaci BPMN vychází z vývojových diagramů (flow charts), které už jednou v minulosti byly neúspěšně používány pro popis obecných algoritmů. Přestože BPMN přichází s řadou vylepšení, především zápis dovolující používat události a výjimky, základní nevýhody jejich předchůdce stále zůstávají. Diagramy jsou přehledné a snadno pochopitelné, ale pouze, pokud je řídící logika dostatečně jednoduchá. Jakmile začne stoupat komplexnost popisovaného procesu, stává se výsledný graf velmi chaotický a komplikovaný. Nehledě na fakt, že s rostoucí složitostí také rostou prostorové nároky na diagram. To lze sice částečně kompenzovat zavedením hierarchie, ale i tak je popis složitějšího procesu obvykle rozsáhlý. Tento problém připouští i samotný standard BPMN, který zavadí speciální propojovací události, které slouží pro přechod mezi grafy na různých stránkách. Další nevýhodu představuje složitá modifikace již vytvořeného grafu. Často je při zapracování změny nutné celý graf překreslit, protože se musí jednotlivé uzly přeskupit, aby se dalo lépe využít vyhrazený prostor diagramu.

Na druhé straně stojí jazyky jako BPEL nebo XPDL, které si kladou za hlavní cíl, aby byl výsledný model přímo interpretovatelný. Jejich vytváření však není příliš snadné. Výsledné XML je velmi komplikované, využívá řadu jmenných prostorů a pro jeho editaci je tak často nezbytné využít některé z dostupných vývojových prostředí. Formát XML lze využít pro vyjádření obecně strukturované hodnoty. Pokud ale množství značek významně převyšuje celkový obsah dokumentu, stává se výsledný zápis značně nepřehledný. XML také není vhodné například pro vyjadřování cyklů nebo přiřazení hodnoty z jedné proměnné do druhé, které se v popisu procesu běžně vyskytuje kvůli mapování vstupních a výstupních dat. Při modelování procesu se proto obvykle postupuje tak, že logika řízení se vyjadřuje pomocí BPMN a potom se provádí transformace do BPEL nebo XPDL, kde se dále doplňují informace potřebné pro interpretaci modelu ve workflow engine. Bližší popis nejdůležitějších standardů a jejich vzájemné kompatibility je v [1].

Předkládaný text si klade dva důležité cíle. Jednak obsahuje bližší popis standardu BPMN a na příkladech popisuje krok za krokem, jak lze notaci použít pro zachycení řídící logiky procesu.

Zároveň však souběžně usiluje o návrh nového modelovacího jazyka využívající koncepty objektově orientovaného paradigmatu, který by vyřešil problémy popsané výše. Výsledný model by potom nebyl definován ani pomocí grafické notace, ani deklarativním XML, které využívají všechny současné standardy, ale prostřednictvím obecného programovacího jazyka.

Pokusím se postupně ukázat, že lze nalézt takové mapování mezi standardem a použitým jazykem, že všechny důležité konstrukce BPMN půjde pomocí tohoto jazyka nejen vyjádřit, ale nový popis procesu přinese oproti BPMN navíc některé další užitečné vlastnosti. Za nejdůležitější atributy považuji integraci dat do modelu procesu a především schopnost interpretace modelu pomocí workflow engine bez nutnosti vytvářet další popis ve formátu XML.

Jako jazyk pro popis procesu jsem se rozhodl vybrat již existující objektově orientovaný jazyk. Bylo by pochopitelně možné se pokusit navrhnout jazyk úplně nový, který by přesně obsahoval všechny potřebné konstrukce. Podle mého názoru jsou však současné objektově orientované jazyky naprosto dostatečné, aby mohly být použity pro popis procesu, který bude dobře čitelný, jednoduše interpretovatelný a snadno rozšiřitelný. Pokud bude potřeba, vytvořím knihovnu důležitých tříd a funkcí, aby šlo použít všechny obraty z BPMN, a vytváření modelu bylo tak co nejjednodušší a přímočaré.

Pro popis procesu budu využívat skriptovací jazyk **Python**. K této volbě mě vedlo několik důležitých vlastností jazyka:

- Jedná se o interpretovaný jazyk – schopnost pozastavit provádění kódu je užitečné při následné interpretaci modelu pomocí workflow engine,
- obsahuje všechny důležité koncepty objektově orientovaného paradigmatu,
- má kompaktní a přehlednou syntax – výsledný model je dobře čitelný.

Python představuje podle mého názoru vhodný jazyk pro modelování procesů, není však samozřejmě jediný možný kandidát. Na jeho místo je možné dosadit jiný jazyk, který bude splňovat uvedená kritéria. Důležité je především to, jakým způsobem půjde vyjádřit důležité pojmy a konstrukce potřebné pro popis procesu.

Návrh jazyka pro modelování procesů je rozdělen do následujících kapitol:

Kapitola 2.1 obsahuje shrnutí základních prvků modelu procesu. Nejdříve je přiblížena definice úkolu, události a brány podle standardu BPMN a potom demonstruji jejich použití na konkrétním příkladu. Dále je proveden návrh, jakým způsobem je možné tyto prvky vyjádřit v jazyce Python. V závěru kapitoly zmiňuji využití dekorátoru při řešení asynchronního úkolu.

Kapitola 2.2 představuje podprocesy a jejich význam při modelování. Popíši definici podprocesů v BPMN a navrhnu vyjádření pomocí zanořovaní funkcí v jazyce Python.

V kapitole 2.3 se zaměřím na základní řídící konstrukce, mezi které patří sekvence, selekce, paralelizace a iterace. Vysvětlím jejich sémantiku a uvedu odpovídající konstrukce v Pythonu.

Obsahem kapitoly 2.4 je modelování aktérů. Jednotlivé úkoly jsou definovány jako metody participantů procesu. Na příkladě objednávky je ukázáno, jak lze využít objektovou orientaci při modelování organizačních jednotek i jednotlivých pracovníků.

Kapitola 2.5 popisuje způsob, jakým lze modelovat datové objekty a jejich vazby. Dále je diskutována potřeba dat pro interpretaci procesu a jejich dostupnost v nově navrženém popisu.

Závěrečná kapitola 2.6 pojednává o interpretaci nově navrženého popisu. Podrobněji je objasněno pozastavení procesu při vykonávání asynchronního úkolu a omezení na práci s daty, která s tím souvisejí.

1.1 Značení v textu

Názvy tříd, metod a funkcí jsou označeny kurzívou. Důležité pojmy a tvrzení budou zvýrazněny tučným písmem. Obrázky pro výklad BPMN notace byly vytvořeny v aplikaci pro modelování procesů *BizAgi*.

2 Návrh jazyka pro modelování procesů

2.1 Základní prvky modelu

Nejdříve se zaměřím na základní stavební prvky modelu, které vyjadřují všechny důležité situace, k nimž během provádění procesu dochází. Model procesu v BPMN představuje orientovaný graf složený z uzlů a hran. Standard rozlišuje tři základní typy uzlů, mezi které patří:

- úlohy,
- události,
- a brány.

Uzel typu **úloha (task)** představuje logickou jednotku práce, která musí být během provádění procesu realizována. V grafu je znázorněn pomocí obdélníku se zakulacenými rohy a příslušným popiskem charakterizujícím náplň této úlohy. Standard dále nabízí možnost upřesnit, o jaký typ úlohy se jedná. Jednotlivé typy ukazuje obrázek Obr. 2-1.

Obr. 2-1 Typy úkolů

Uživatelská úloha (user task) vyjadřuje, že práci má provést nějaká osoba. Pro její realizaci musí workflow engine alokovat patřičné lidské zdroje. Úloha tedy není automatizovaná, musí se počkat, až ji příslušný člověk provede, a proto lze předpokládat, že aktivita spojená s řešením úlohy může trvat delší časový úsek. Příkladem je fyzické dodání balíku se zbožím, kdy přepravce musí balík odvést na dané místo a úloha je dokončena, až potvrdí workflow systému, že balíček doručil.

Úloha služby (service task) generuje při řešení naopak automatizovanou aktivitu, kdy je obecně požádána nějaká služba, aby vyřešila popsaný úkol. V [2, s. 65] se doporučuje využívat tento typ úlohy pro synchronní žádost. To znamená, že workflow engine počká na odpověď služby a potom ihned pokračuje dál. Z toho vyplývá, že je vhodné takto modelovat úlohy, jejichž provedení trvá krátkou dobu, protože jinak by bylo blokováno zpracování procesu.

Úloha poslání/přijetí (send/receive task) představuje konstrukci, jak požádat o provedení služby a přijmout odpověď asynchronně. V modelu lze tuto situaci vyjádřit rovněž pomocí události uprostřed procesu (intermediate event.

Skriptovací úloha (script task) uzavírá výčet základních typů úkolů. V tomto případě se jedná o úlohu určenou pro samotný workflow engine. Ten automaticky provede příslušný skript. Podle normy by se mělo jednat pouze o spuštění jednoduchých funkcí, které se provedou rychle.

Uzel typu **událost (event)** slouží v modelu procesu k identifikaci toho, že proces se nachází ve význačném stavu důležitém z pohledu jeho řízení. Každý proces by měl mít definované okolnosti svého začátku, protože workflow engine musí rozpoznat, kdy má vytvořit a spustit jeho instanci. V BPMN se modeluje okamžik spuštění procesu pomocí **počáteční události** (start event). Standard rozlišuje několik způsobů zahájení. První variantou je start v důsledku přijmutí zprávy (message start). Často je potřeba také vyjádřit, že proces začne v závislosti na časové podmínce (time start). Další možností je zahájení na základě splnění určité datové podmínky (conditional start). Norma dovoluje také použít počáteční událost bez udání typu (none start). Takové zahájení se užívá především u podprocesů (viz kapitola 2.2). Grafickou reprezentaci uvedených typů počátečních událostí ukazuje Obr. 2-2.

Obr. 2-2 Typy počátečních událostí

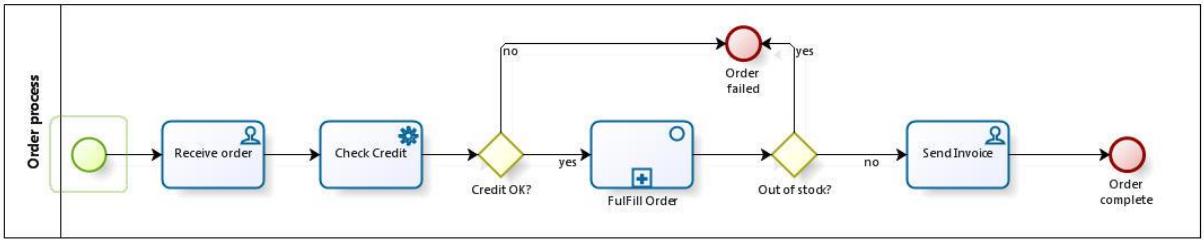
Podobně je nutné modelovat okamžik, že proces skončil. Zatímco počáteční událost je pouze jedna, koncových událostí se v modelu vyskytuje obvykle více, protože cílem není zachytit pouze úspěšné ukončení, ale naopak pečlivě dokumentovat všechny neúspěšné stavy, do kterých může proces dospět. V [2] se výstižně píše, že jedním z důvodů, proč modelujeme procesy, je právě snaha odhalit, co vede v praxi k neúspěchu procesu a dokázat takovéto situace rozlišit, správně na ně zareagovat a příště se jim snažit předcházet. BPMN využívá pro vyjádření faktu, že proces dospěl do koncového stavu, **koncovou událost (end event).** Ukončení může být provázeno odesláním zprávy (message end). Jinou variantou je ukončení v důsledku chyby (error end), která se potom propaguje na nejbližší vyšší úroveň modelu procesu. Proces lze také explicitně ukončit (terminate end), aniž by byl informován případný nadřazený proces. BPMN dovoluje také vyjádřit blíže nespecifikovaný způsob ukončení (none end). Obrázek níže demonstruje různé typy koncových událostí.

Obr. 2-3 Typy koncových událostí

Posledním důležitým typem uzlu je **brána**, která vyjadřuje kontrolní bod v řízení procesu. Uzel brány má jednu vstupní hranu a dvě a více hran výstupních, přičemž volba výstupu při interpretaci procesu je dána typem brány a případně strážnou podmínkou. Graficky je uzel znázorněn pomocí diamantu. Není-li blíže specifikován typ, potom je chování brány podle BPMN definováno jako XOR. Při provádění prostřednictvím workflow engine se tedy vyhodnotí asociovaná podmínka a na jejím základě se vybere pouze jedena výstupní sekvence úloh.

Zbývá ještě vyložit sémantiku orientované hrany (v normě označené jako spojovací objekt), která propojuje jednotlivé uzly grafu. Standard rozlišuje více typů hran, zatím uvedu pouze hranu pro **sekvenční tok,** v diagramu znázorněnou pomocí plné orientované šipky. V relaci sekvence může být jak uzel typu úloha, tak rovněž uzel události nebo brány. Význam interpretace diskutované hrany z pohledu workflow engine je takový, že po ukončení zpracování zdrojové uzlu nastane bezprostředně zpracování uzlu cílového.

Po vyložení základních modelovacích prvků již mohu představit praktický příklad procesu objednávky (order process) a ukázat, jakým způsobem je možné takovýto proces popsat v BPMN. Výsledný model znázorňuje Obr. 2-4.



Obr. 2-4 Proces objednávky - převzato z [2]

Proces je zahájen startovací událostí, v tomto případě nebudu specifikovat její typ. První úkol, který se musí uskutečnit, je přijetí objednávky (receive order). Ikona vyjadřuje, že úkol bude proveden člověkem. Po dokončení úlohy se začne okamžitě provádět úkol kontroly účtu (check credit), tentokrát workflow engine iniciuje automatickou aktivitu. Následně proces vstupuje do exkluzivní brány. Pokud se nachází na účtu peníze, bude se pokračovat realizací objednávky (fulfill order), jinak proces končí chybným stavem reprezentovaným pomocí dále nespecifikované koncové události. Úspěšná větev ještě pomocí druhé exkluzivní brány zkontroluje, zdali je požadované zboží aktuálně na skladě a pokud ano, zpracuje se úkol poslání objednávky (send invoice). Proces poté končí úspěchem. V opačném případě dojde ke koncové události označující selhání procesu.

Nyní se pokusím vyjádřit uvedený příklad v jazyku Python. V úvodu kapitoly jsem naznačil, že budu při modelování procesu usilovat o maximální využití konstrukcí, které už jazyk přímo nabízí, a proto pro popis procesu i jednotlivých úkolů použiji obecnou funkci. Proces objednávky je obsahem příkladu Př. 1.

Př. 1 Vyjádření aktivity, sekvence a podmínky

```
#úlohy
@userTask
def receiveOrder(process): pass
@serviceTask
def checkCredit(process): pass
@userTask
def fullFillOrder(process): pass
@userTask
def sendInvoice(process): pass

#výjimky
```

```
class ProcessFailed(WkfException): pass
class orderFailed(ProcessFail): pass

#definice procesu objednávky
def OrderProcess(process):
      receiveOrder(process)
      if checkCredit(process):
            if fullFillOrder(process):
                  sendInvoice(process)
            else:
                  raise orderFailed
      else:
            raise orderFailed
```

V diagramu BPMN se nachází postupně 4 úlohy: přijetí objednávky, kontrola účtu, realizace objednávky a poslání faktury. V novém popisu jim odpovídají čtyři stejně pojmenované funkce. Pro jednoduchost zatím nebudu definovat jejich tělo. Samotný proces je potom opět popsán prostřednictvím funkce implementující jeho řídící logiku. Realizaci úkolů reprezentuje zavolání příslušné funkce úkolu a pořadí, v jakém jsou funkce postupně volány, odpovídá sekvenčnímu toku vyjádřenému v diagramu pomocí orientovaných hran. Exkluzivní brána koresponduje s konstrukcí pro větvení **if – else**. Ta zajistí požadovanou selekci, a provede se tak pouze příslušná sekvence úloh.

Pro vyjádření situace, že proces objednávky selhal, tj. dospěl do výjimečného stavu, který musí být identifikován, jsem využil objektu výjimky. Koncovou událost reprezentující neúspěch procesu lze tedy jednoduše modelovat vyvoláním výjimky. V příkladu je vidět, že výjimka nesoucí informaci o neúspěchu objednávky (orderFailed) je specializací obecné výjimky vyjadřující jakýkoliv neúspěšný konec procesu (processFailed) a ta dědí zase z obecné výjimky workflow engine (WkfException). V procesu je tedy možné úspěšně využít systém výjimek, který je významným konceptem objektově orientovaného programování umožňujícím propagovat nastání výjimečné události přes více úrovní procesu a různě reagovat podle toho, jakého typu je aktuálně zachycená výjimka.

Základní kostra modelu procesu je tedy velmi jednoduchá. Na tomto místě je však ještě nutné uvést několik důležitých poznámek. Zatím jsem neřekl, jakým způsobem bude reprezentována počáteční událost procesu. Z navrženého popisu je patrné, že proces se začne provádět v okamžiku, kdy workflow engine zavolá funkci, která ho modeluje. Na rozdíl od BPMN tedy ve zkoumaném modelu nebude explicitně vyjádřeno pomocí nějakého příkazu jazyka Python, že nastala počáteční událost. V případě blíže nespecifikované počáteční události to nijak nevadí, pokud by ale spuštění procesu záviselo na datové/časové podmínce, nebo skutečnosti, že dorazí zpráva od jiného procesu, nemám aktuálně způsob, jak takovéto spuštění popsat. Řešení není ovšem složité. Podmínky zahájení procesu popíši pomocí registrace modelu procesu do workflow engine, jejíž součástí bude také specifikace spouštěcí podmínky. Tento přístup je podle mého názoru flexibilnější, protože registraci lze dynamicky upravit a měnit například způsob nastartování procesu podle určité konfigurace. Podobně také koncová událost signalizující úspěšné dokončení procesu není v ukázkovém popisu

vyjádřena přímo, ale vyplývá z toho, že funkce procesu skončí provedením posledního příkazu. Součástí registrace procesu může být také definování ukončovací aktivity, jako například odeslání zprávy, která odpovídá koncové události zprávy.

Druhým důležitým faktem je povinný parametr *process*, který musí mít každá funkce implementující proces a který je nutný také předat každé funkci úkolu. Tento parametr obsahuje referenci na instanci procesu. Připomínám, že funkce procesu představuje **model procesu**. Na první pohled se může zdát tento parametr nadbytečný a zbytečně komplikující výsledný kód modelu, ale jeho přítomnost je nutná, protože každý úkol musí mít při interpretaci modelu přidělenou konkrétní instanci procesu, v rámci které je realizován. Situace je úplně stejná jako při definici obecné třídy, kde se využívá klíčové slovo this (v Pythonu self) pro přístup k jejím vlastnostem a metodám. Při vytvoření instance třídy je potom reference na tuto instanci předávána jako první parametr všem metodám (v Pythonu explicitně jako první povinný parametr), aby mohly přistupovat k datům právě aktuální instance. Stejným způsobem tedy funkce úkolu potřebuje přistupovat k aktuální instanci procesu.

Poslední poznámka se týká využití speciální konstrukce jazyka Python označované jako dekorátor. Dekorátor může být funkce, ale také celá třída, která je aplikována na funkci nebo metodu uvedením *@nazev_dekoratoru* na řádku bezprostředně předcházejícím její definici. Tímto způsobem se interpretu jazyka sdělí, že se při zavolání funkce nemá provést její tělo, ale místo toho se má spustit dekorátor, kterému je předán jako parametr objekt funkce, jež se měla spustit. Dekorátor tedy umožňuje obalit samotné volání funkce a provést nějaký kód před a po ukončení jejího provádění.

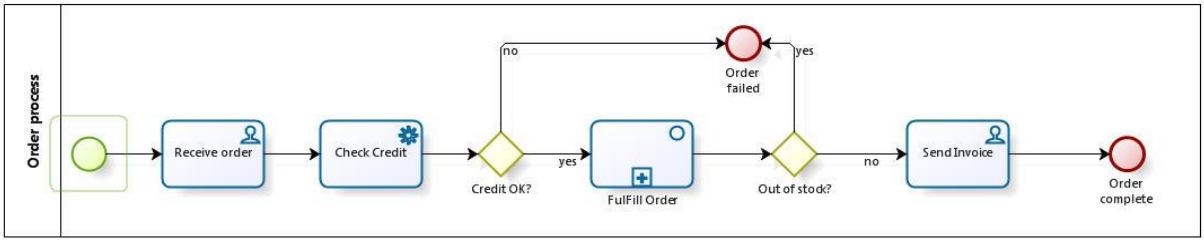
V příkladě Př. 1 využívám dekorátor pro každou funkci úkolu. Cílem je deklarovat typ úkolu a především skrýt rozdíly v synchronním a asynchronním volání funkce úkolu. Zatímco v případě uživatelského úkolu je potřeba vykonávání procesu pozastavit a počkat, než vybraný člověk provede aktivitu spojenou s daným úkolem a zaznamená do workflow systému odpověď, u synchronního dotazu na službu je po vykonání funkce možné hned pokračovat. Tento zásadní rozdíl, který má dopad na interpretaci procesu, však nechci explicitně ošetřovat v algoritmu procesu, a proto zde pouze volám funkci úkolu a technické detaily volání a získání odpovědi nechám vyřešit příslušnému dekorátoru. Bližší informace o způsobu interpretace navrhovaného popisu procesu jsou obsahem kapitoly 2.6.

2.2 Podprocesy

Důležitý vyjadřovací prostředek, který nabízí standard BPMN, představuje možnost vytvářet hierarchický model za pomoci podprocesů. Podproces reprezentuje detailní pohled na úkol rodičovského procesu, ve kterém je průběh tohoto úkolu modelován opět pomocí procesu složeného z dalších úkolů. Tento přístup lze opakovaně aplikovat a jednotlivé úkoly podprocesů vyjadřovat opět pomocí podprocesů až do okamžiku, kdy je popis dostatečně podrobný a jeho úkoly jsou již

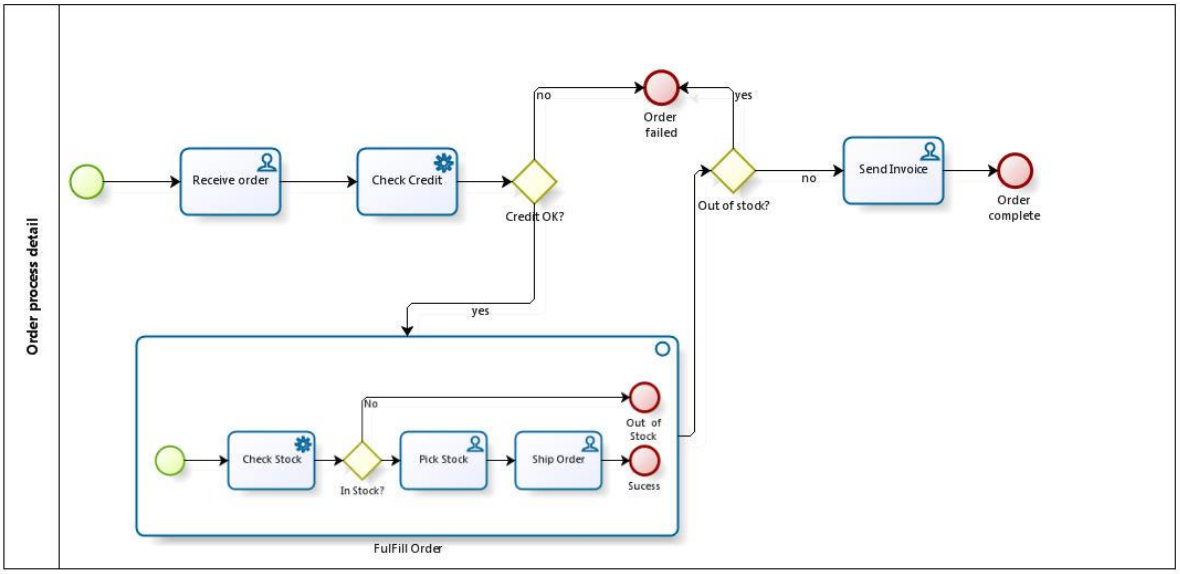
znázorněny jako atomické. Používání podprocesů tedy úzce souvisí s volbou úrovně pohledu na popis procesu. Jedná se koncept, prostřednictvím něhož lze redukovat složitost modelu, a tak zvýšit jeho přehlednost.

BPMN používá pro vyjádření podprocesu symbolu "+" uvedeného v obdélníku úkolu, který značí, že úkol lze dále expandovat a zobrazit tak jeho vnitřní strukturu pomocí podprocesu. Situaci zachycuje Obr. 2-5 procesu objednávky. V něm je naznačeno, že úkol *realizace objednávky* není atomický a je možné prozkoumat jeho detail.



Obr. 2-5 Proces objednávky - převzato z [2]

Většina současných modelovacích nástrojů dovoluje jednoduše zvýšit granularitu popisu pomocí kliknutí myší na příslušný úkol. Po jeho rozbalení se objeví celý podproces, který je znázorněn na obrázku níže.



Obr. 2-6 Proces objednávky - detail

Zkoumaný podproces je opět zahájen počáteční událostí. Standard BPMN požaduje, aby se neuváděl žádný typ této události, protože k spuštění podprocesu dochází automaticky v okamžiku, kdy workflow engine začne zpracovávat úkol *realizace objednávky*. Podobně musí odpovídat koncové události podprocesu. Podle diagramu končí úkol buď chybějícím zbožím ve skladu, nebo úspěšným nalezením zboží, které je vyskladněno (pick stock) a následně odvezeno zákazníkovi (ship order).

Používání podprocesů je velmi důležité, a proto je nutné, aby také objektově orientovaný popis obsahoval způsob, jak definovat tento pojem. Naštěstí se jedná o jednoduchý úkol, jelikož proces vyjadřuji pomocí funkce a funkce lze jednoduše zanořovat až do požadované hloubky. V demonstračním příkladu tedy pouze přidáme funkce pro úkoly podprocesu a úkol realizace objednávky nahradíme funkcí podprocesu, nyní už bez dekorátoru. Definici úkolu s využitím podprocesu demonstruje příklad Př. 2.

Př. 2 Podproces

```
#úkoly podprocesu
@serviceTask
def checkStock(process): pass
@userTask
def pickStock(process): pass
@userTask
def shipOrder(process): pass


#podproces modelující strukturu úkolu
def fulFillOrder(process):
 if checkStock(process):
       pickStock(process)
       shipOrder(process)
       return true
 return false

#Hlavní proces objednávky
def OrderProcess(process):
 receiveOrder(process)
 if checkCredit(process):
       if fullFillOrder(process):
             sendInvoice(process)
       else:
             raise orderFailed
 else:
       raise orderFailed
```

Zatím se snažím důsledně kopírovat model procesu v BPMN a ukázat, že stejného popisu jde dosáhnout také v navrhovaném modelovacím jazyce. Kód procesu je ale možné zjednodušit díky využití systému výjimek. Namísto vracení pravdivostní hodnoty z podprocesu *realizace objednávky* (fulFilOrder), použiji přímo vyvolání výjimky *outOfStock*, která se automatiky předá nadřazenému procesu. Ten ji dále zpracuje a vyvolá výjimku udávající, že se objednávka nezdařila. Modifikovaný příklad je uveden níže.

Př. 3 Použití výjimek

```
#podproces modelující strukturu úkolu
def fulFillOrder(process):
 if checkStock(process):
       pickStock(process)
       shipOrder(process)
 else:
       raise outOfStock

#Hlavní proces objednávky
```

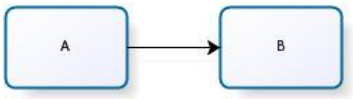
```
def OrderProcess(process):
 try:
       receiveOrder(process)
       if checkCredit(process):
             fullFillOrder(process):
             sendInvoice(process)
       else:
             raise badCredit
 except: outOfStock, badCredit
       raise OrderFailed
```

2.3 Řízení toku práce

Pro řízení toku práce existují ustálené šablony, jejichž použití je nezávislé na vybraném modelovacím jazyku. Mezi nejdůležitější z nich patří sekvence, selekce, paralelizace a iterace. V následujícím textu vždy uvedu způsob, jakým popisuje vzor řízení standard BPMN, a navrhnu vyjádření v jazyce Python. Příklady budou využívat abstraktní úkoly *A, B, C* a *D*.

Nejjednodušší způsob řízení toku práce představuje spojení úkolů do **sekvence**. V BPMN diagramu vyjadřuje sekvenci orientovaná hrana, vedoucí od zdrojového úkolu k cílovému. V tomto jediném případě není využito uzlu brány. Ve skutečnosti je možné bránu mezi úkoly vložit, ale její použití zbytečně komplikuje diagram, a proto se užívá přímé propojení úkolů orientovanou hranou. Příklad Př. 4 demonstruje sekvenci dvou úkolů.

Př. 4 Sekvence

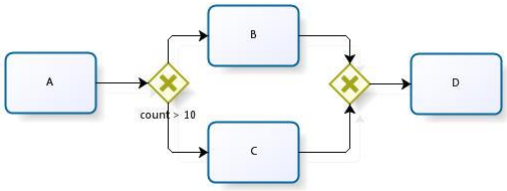



```
A()
B()
```

Nejdříve se realizuje úkol *A*. Po jeho bezprostředním ukončení workflow engine iniciuje provádění úkolu *B*. V Pythonu odpovídá sekvenčnímu provedení úkolů zavolání funkce A následované voláním funkce úkolu *B*.

Další důležitý vzor představuje **selekce**, kterou jsem již použil v příkladu Př. 1. V BPMN diagramu je reprezentována pomocí brány typu XOR, která obsahuje v diamantu symbol *X*. Stejnou funkci plní rovněž brána bez udání typu. Sémantika brány odpovídá selektivnímu výběru jedné výstupní sekvence. Proces tedy může probíhat ve dvou větvích, ale pouze jedna je při interpretaci vybrána. Pro opětovné spojení do jedné sekvence bez synchronizace se rovněž využije brána typu XOR (viz Př. 5).

Př. 5 Selekce

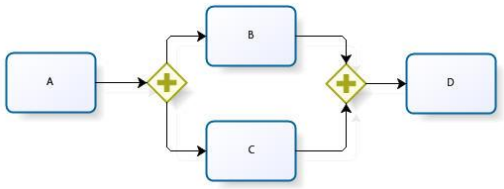



```
A()
if count > 10:
      B()
else:
      C()
D()
```

Zdůrazňuji, že podmínka v jazyce Python představuje plnohodnotný výraz, který workflow engine může bez problémů vyhodnotit, zatímco v BPMN se jedná o pouhou anotaci brány, která nemá definovanou sémantiku a její zápis nejde při provádění modelu přímo použít.

Třetím vzorem řízení je **paralelizace**. Jedná se o velmi důležitou řídící konstrukci, protože z důvodu efektivnější provádění procesu je často potřeba, aby se úkoly realizovaly souběžně. Standard využívá pro vytvoření paralelních sekvenčních toků uzel brány typu AND, která se graficky znázorňuje pomocí diamantu se symbolem „+“ uvnitř (and-split). Situace je zachycena v příkladu Př. 6.

Př. 6 Paralelizace



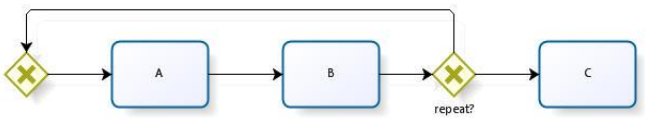

```
def p1(p):
 B(p)
def p2(p):
 C(p)

def testParallel(p):
 A(p)
 p.newProcess(p1)
 p.newProcess(p2)
 p.join(p1, p2)
 D(p)
```

Význam diagramu je následující. Nejdříve se provede úkol *A*. Potom se souběžně budou provádět úkoly *B* a *C* v libovolném pořadí. Úkol *D* se provede okamžitě po dokončení úkolu *B* a *C*. Druhá brána typu AND (And join) tedy zajišťuje synchronizaci a zabrání provedení úkolu *D*, dokud nejsou oba úkoly *B, C* hotovy.

V jazyce Python není přímo podpora pro konstrukci paralelního procesu, a proto je potřeba implementovat potřebnou funkcionalitu, kterou jsem umístil do objektu instance procesu *p*. V příkladu je vidět, že nejdříve je nutné definovat funkce procesů obou sekvenčních toků, které vykonají úkoly *B* a *C*. Samotný hlavní proces nejdříve provede úkol *A* a pak vytvoří instance definovaných procesů pomocí metody ***newProcess()***. Synchronizaci zajistí volání metody ***join()*** instance procesu. Následně se vykoná závěrečný úkol *D*.

Poslední šablonu řízení, kterou představím, je **iterace.** Ta zajišťuje, že jeden nebo více úkolů může být provedeno opakovaně. Př. 7 demonstruje iteraci sekvence úkolů *A, B*. Vzor používá exkluzivní OR pro modelování rozhodnutí, jestli bude sekvence znovu opakována, nebo se provede úkol *C*.

Př. 7 Iterace

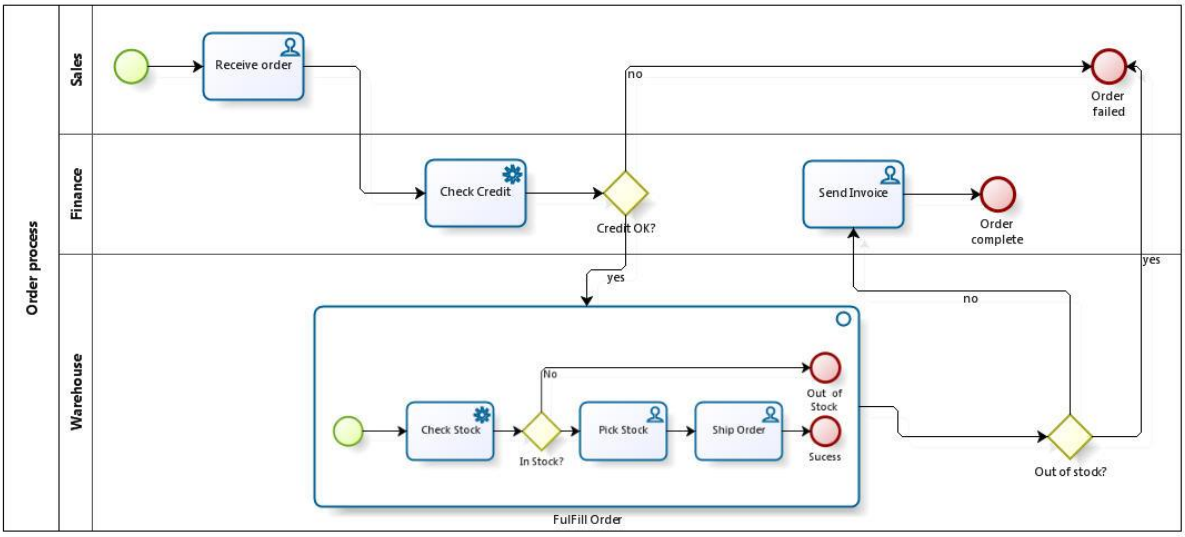



```
while repeat:
    A()
    B()
C()
```

Zápis cyklu s podmínkou *repeat* je v navrhovaném popisu opět velmi jednoduchý. Stačí použít konstrukci *while*, která bude opakovat úkoly *A*, *B*, dokud bude podmínka platit. Na závěr se provede úkol *C*.

2.4 Modelování aktérů

Zatím jsem se v rámci procesu zabýval pouze úkoly a logikou řízení jejich provádění. Nevěnoval jsem se vůbec specifikaci aktérů, kteří budou jednotlivé úkoly provádět. Standard BPMN přináší prostředek, jak vyjádřit, že za realizaci úkolů je zodpovědný určitý člověk, organizační jednotka nebo jiná entita. V diagramu k tomuto účelu slouží prvky **bazén** (pool) a **plavecká dráha** (swimlane). Jedná se o oficiální terminologii, kterou zavádí standard BPMN. Názvy prvků vychází z grafického znázornění, ve kterém je celý proces uzavřen v obdélníku (bazén) a ten je potom rozdělen na několik drah, podobně jako v reálném bazénu. Každá dráha je označená jménem entity zodpovídající za úkoly, které jsou v ní umístěné. Plavecké dráhy tedy slouží pro organizaci diagramu podle aktérů a přináší důležitou informaci pro interpretaci modelu. Jejich využití budu demonstrovat na příkladu procesu objednávky. Nový diagram obsahující údaje o aktérech ukazuje Obr. 2-7.



Obr. 2-7 Proces objednávky s aktéry - převzato z [2]

Výsledný model je segmentován pomocí tří aktérů. Obchodní oddělení (sales) má na starosti přijmutí objednávky. Finanční oddělení (finance) bude provádět kontrolu peněz na účtu a odešle výslednou fakturu zákazníkovi a konečně sklad (warehouse) bude zajišťovat realizaci samotné objednávky.

Jakým způsobem ale zavést aktéry do popisu procesu v Pythonu? Využiji k tomu objektovou orientaci jazyka. Jednotlivé participanty procesu budu modelovat jako objekty a úkoly, které mají realizovat, budou představovat metody těchto objektů. Od ostatních metod objektu je odliším jednoduše pomocí dekorátoru. Musím proto přehodnotit dosavadní příklad Př. 2 a definovat několik nových tříd.

Nejdříve zavedu organizační jednotky. Vytvořím třídu obchodního oddělení, která zahájí proces objednávky, alokuje patřičné zdroje pro příjem objednávky a předá finančnímu oddělení podklady pro kontrolu peněz na účtu. Třída je uvedena v příkladu níže.

Př. 8 Obchodní oddělení

```
class Sales(object):

    def __init__(self, salesManagers, finance):
        self.salesManagers = salesManagers
        self.finance = finance

    def orderProcess(self, process, salesManager):
        salesManager.receiveOrder(process)
        self.finance.processOrder(process)

    def processOrder(self, process):
        self.initProcess(process, self.salesManagers[0])
```

Zavoláním metody *processOrder()* dojde ke spuštění procesu objednávky. Obchodní oddělení vybere jednoho ze svých pracovníků (salesManager) a zahájí samotný proces, jehož algoritmus je popsán v metodě *orderProcess()*. Jako první úkol se má provést přijmutí objednávky, a proto zavolám metodu objektu pracovníka *receiveOrder()*. Po úspěšném provedení se předá zpracování procesu finančnímu oddělení, na jehož instanci musí mít třída referenci, zavoláním její metody *processOrder()*.

Př. 9 Finanční oddělení

```
class Finance(object):
    def __init__(self, accountants, warehouse):
        self.accountants = accountants
        self.warehouse = warehouse

    def orderProcess(self, process, accountant):
        if accountant.checkCredit(process):
            self.warehouse.fullfillOrder(process)
            accountant.sendInvoice(process)

    def processOrder(self, process):
            self.orderProcess(process, self.accountants[0])
```

Příklad Př. 9 demonstruje třídu finančního oddělení. Zavolání metody *orderProcess()* způsobí výběr účetního (accountant), který provede úkol kontroly účtu (metoda objektu *checkCredit()*). Pokud je všechno v pořádku, zavolá se metoda skladu pro realizaci objednávky *fulfillOrder()* a v třetím kroku pošle opět objekt účetního finální fakturu zákazníkovi *sendInvoice()*.

Poslední organizační jednotku představuje sklad. Jeho kód je obsahem příkladu Př. 10. Sklad má k dispozici automatizovanou službu pro vyhledání zboží a dále několik pracovníků, kteří zajišťují vybavení zásilky a rozvoz zboží k zákazníkovi. Proces je popsán opět metodou *orderProcess()*. Její parametry přináší referenci na instance aktérů, které budou potřeba. Nejdříve se zavolá služba pro kontrolu, zdali se zboží nachází na skladu. Potom skladník (stockPicker) vybaví objednaný produkt a přepravce (orderShipper) doručí balík zákazníkovi.

Př. 10 Sklad

```
class Warehouse(object):
    def __init__(self, workers):
        self.workers = workers
        self.service = Warehouse.Service()

    def orderProcess(self, process, service, stockPicker,
orderShipper):
        if service.checkStock(process):
            stockPicker.pickStock(process)
            orderShipper.shipOrder(process)
        else:
            raise OutOfStock

    def fullfillOrder(self, process):
             self.orderProcess(process, self.service, self.workers[0],
self.workers[0])
```

Zbývá ještě uvést třídy modelující samotné aktéry. Pro jednoduchost popíši pouze definici třídy *SalesManager*. U ostatních entit by se postupovalo podobně. Potřebný kód je v příkladu Př. 11.

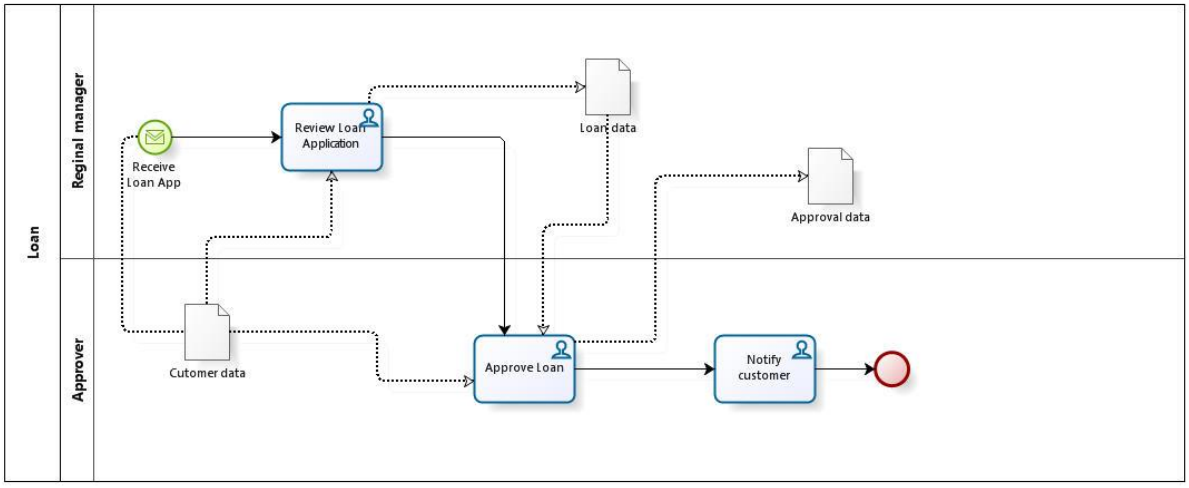
Př. 11 Definice aktéra

```
class Person(WKFParticipant):
    def __init__(self, name, email):
        WKFParticipant.__init__(self, email)
        self.name = name
        self.email = email

class SalesManager(Person):
      @userTask
      def receiveOrder(self, process):
            self.task(process, "receive Order")
```

Pro lidské aktéry definuji nejdříve třídu *Person*, která bude obsahovat například informaci o jméně a emailu příslušné osoby. Tato třída představuje specializaci obecného participanta workflow systému (WkfParticipant). Nyní již definuji třídu *SalesManager* dědící z obecné osoby. Její zatím jediná metoda *receiveOrder()* reprezentuje úkol, za který je manažer zodpovědný. Metoda musí mít uvedený dekorátor, který zajistí technické detaily zpracování asynchronní odpovědi. Implementace metody potom způsobí, že na uvedený email aktéra bude poslána výzva, aby zpracoval objednávku od uživatele.

2.5 Integrace dat

Prozatím se podařilo definovat, jakým způsobem je možné popsat algoritmus procesu včetně základních řídících konstrukcí, používání podprocesů a zapojení modelu aktérů. Poslední nezbytný krok spočívá v integraci dat, bez kterých není možné realizovat řídící logiku a která jsou také nepostradatelná pro vykonávání jednotlivých úkolů. BPMN na úrovni 3 umožňuje popsat datové objekty a jejich vazby na vstupy/výstupy úkolů. Opět se však jedná o abstraktní popis, který musí být posléze doplněn o přesnou specifikaci pomocí definice v XML.



Obr. 2-8 Modelování dat v BPMN - převzato z [2]

Použití datových objektů ukazuje obrázek výše. V diagramu se pro jejich vyjádření užívá speciálního typu uzlu graficky reprezentovaného pomocí dokumentu s ohnutým rohem. Vazby se potom popisují pomocí orientovaných hran, které jsou čárkované a mají prázdnou šipku. V ukázkovém modelu procesu přináší počáteční událost zprávu od zákazníka, jejíž obsah se modeluje pomocí datového objektu *customer data*. Tyto data potom využívá úkol *přezkoumání nároku na půjčku* (review loan application). Vazba je specifikována orientovanou hranou. Následný úkol *schválení půjčky* využívá jak data od zákazníka, tak výstup z přezkoumání. Výsledkem je dokument o schválení půjčky (approval data).

V nově navrhovaném popisu se data modelují snadno, protože datové vstupy procesů a úkolů jsou dány argumenty odpovídajících funkcí a výstupy potom představují návratové hodnoty těchto funkcí. Zápis vzorového modelu v jazyce Python ukazuje příklad Př. 12.

Př. 12 Proces půjčky

```
def loan(process, regionalManager, approver, customerData):
    loanData = regionalManager.review(process, customerData)
    approver.approve(process, customerData, loanData)
    approvalData = approver.notify(process)
```

Funkce procesu je volána s parametry instancí dvou aktérů a vstupní zprávou, která způsobila spuštění procesu, obsahující informace o zákazníkovi. Data jsou potom předána metodě *review()*

manažera, která vrací informace o požadované půjče. Oba dva datové objekty se potom využijí pro metodu schválení *approve()* a ta nakonec vrací nový záznam o poskytnuté půjčce.

Výhodou uvedeného popisu je přítomnost reálných dat přímo v algoritmu procesu, takže je možné přistupovat k atributům objektu, modifikovat je nebo vytvářet data nová a ty potom následně ukládat do databáze. Veškeré záznamy uložené v informačním systému organizace mohou být v rámci procesů jednoduše použity.

2.6 Interpretace modelu

Důležitým přínosem navrhovaného popisu procesu je možnost ho přímo spustit pomocí workflow engine. Na rozdíl od deklarativního zápisu pomocí XML, které používají současné standardy, je model popsán procedurálně. S tím je však spojen jeden významný problém. Provedení funkce procesu je prakticky okamžité. Jednotlivé úlohy ale naopak často trvají delší časový úsek, dokonce může jít o týdny i měsíce. Stačí uvážit například schvalovací proces, ve kterém se musí postupně vyjádřit více různých lidí. Provádění funkce procesu tedy musí být v okamžiku zpracování asynchronní úlohy pozastaveno a až po jejím splnění může engine pokračovat interpretací následujícího příkazu. Při přerušení je nutné uložit veškerá data procesu, aby bylo zaručeno, že po opětovném spuštění bude stav veškerých datových struktur stejný jako před jeho zastavením.

Implementace zastavení funkce úkolu je skryta v jeho dekorátoru. Při psaní definice modelu tedy není nutné tento problém nijak ošetřovat. Dekorátor provede tělo funkce vybraného úkolu a potom ukončí provádění funkce procesu. Workflow engine si přitom uloží úkol jako provedený a případně uschová jeho výstup. V okamžiku, kdy aktér splní svoji činnost a sdělí tento fakt systému, musí dojít k obnovení procesu. Toho lze docílit opětovným spuštěním funkce procesu. Tentokrát však nejde o reálný výpočet, protože všechny úkoly včetně posledně dokončeného se ve skutečnosti neprovedou. Jelikož workflow engine eviduje proběhnuté úkoly a jejich výstupy, může prostřednictvím dekorátoru provedení funkce úkolu zrušit a vrátit přímo uloženou návratovou hodnotu. Takovýmto způsobem se projde algoritmus procesu až po první neprovedený úkol a od něj pokračuje normální provádění. Synchronní volání služeb, provádění obslužných skriptů nebo zanoření se do funkce podprocesu probíhá standardním způsobem.

Mechanismus opakovaného spouštění dokáže simulovat pozastavení vykonávání procesu. Stále je však mít na paměti, že musí být dodržena konzistence dat procesu, a proto v rámci procesu je možné využívat pouze hodnoty, které si proces sám vytvořil pomocí svých úkolů nebo vstupní parametry. Řízení nesmí být tedy založeno na datech mimo kontext provádění procesu ani nesmí přímo využívat například čtení dat z databáze, protože její stav se může během opakovaného přehrávání scénáře procesu měnit. Vždy proto musí být získání takovýchto dat provedeno skrze funkci úkolu.

Navržený způsob interpretace sice klade omezení na tvůrce modelu, zároveň ale nabízí veškeré výhody spojené s objektově orientovaným programováním a to je podle mého názoru významný přínos. Za nezbytné ale považuji, aby se omezující podmínky definovaly formálně a následně se popsal jejich dopad na vyjadřovací sílu modelu.

3 Závěr

V této práci jsem se zabýval modelováním procesů pomocí jazyka Python. Navrhnul jsem nový popis procesu, který využívá plné síly obecného objektově orientovaného jazyka. Pro všechny důležité konstrukce standardu BPMN se mi podařilo vytvořit ekvivalentní vyjádření v jazyce Python. Neudělal jsem zatím formální důkaz ekvivalence vyjadřovací síly obou popisů, ale na příkladech jsem demonstroval, že specifikace procesu v Pythonu je použitelná a nabízí některé další zajímavé vlastnosti, které diagram v BPMN nemá. Model může být přímo interpretován prostřednictvím workflow engine bez jakékoliv podpůrného XML. Dále lze v popisu procesu využívat rovnou data organizace potřebná pro rozhodnutí v rámci řídící logiky procesu. Při modelování je také možné použít systému výjimek nebo modelovat participanty procesu jako objekty.

Model procesu je v navrženém popisu reprezentován funkcí, jejíž tělo implementuje algoritmus procesu. Jednotlivé úkoly, které se musejí realizovat, jsou potom opět funkce, respektive metody jednotlivých aktérů, kteří nesou zodpovědnost za jejich provedení. Asynchronní volání je zajištěno prostřednictvím dekorátoru, který skrývá technické detaily čekání na dokončení úkolu.

Navrhnul jsem také základní princip interpretace modelu pomocí workflow engine. Ten je založen na postupném provádění funkce reprezentující proces. Realizace asynchronních úkolů, která vyžaduje pozastavení provádění procesu do doby, kdy je k dispozici výsledek úkolu, je řešena opakovaným pouštěním funkce procesu s tím, že již splněné úkoly se přeskočí a jejich návratové hodnoty doplní workflow engine. Tento způsob interpretace klade bohužel jisté omezení na používání dat vyplývající z opakovaného provádění algoritmu. Bezpečně lze používat pouze datové struktury získané přes parametry procesu nebo prostřednictvím úkolu v rámci procesu. Do budoucna by bylo vhodné provést formální důkaz toho, jestli navržený způsob interpretace nemá nějaký další negativní vliv na funkčnost modelu.

Literatura

[1] Havey, M.: Essential BusinessProcess Modeling.O'Reilly Media, 2005. 333 s. ISBN 0-596-00843-0

[2] Silver, B.: BPMN Method & Style. Cody-Cassidy Press, 2009. 213 s. ISBN 978-0-9823681-0-7.