FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Analýza datových toků v nástroji SAP Business Objects
Student:	Bc. Jan Potočiar
Vedoucí:	Ing. Lucie Svitáková
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2020/21

Pokyny pro vypracování

Cílem práce je navrhnout a implementovat funkční prototyp modulu, který provede syntaktickou a sémantickou analýzu reportů a jiných datově relevantních objektů v nástroji SAP Business Objects a následně její výsledek využije pro analýzu datových toků. Modul bude zařazen do projektu Manta.

1. Seznamte se s projektem Manta a s nástrojem SAP Business Objects.
2. Navrhněte modul v projektu Manta pro zpracování úloh v SAP Business Objects. Využijte existující infrastrukturu projektu a obecný model pro reportovací nástroje navržený v diplomové práci Petra Košvance, Analýza datových toků v reportovacích nástrojích.
3. Implementujte prototyp, řádně ho zdokumentujte a otestujte.

Seznam odborné literatury

Košvanec, Petr. Analýza datových toků v reportovacích nástrojích. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2019.
Další literaturu dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 31. října 2019

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Analýza datových toků v nástroji SAP BusinessObjects

Bc. Jan Potočiar

Katedra softwarového inženýrství
Vedoucí práce: Ing. Lucie Svitáková

27. května 2020

Poděkování

Rád bych poděkoval všem, kteří mi jakýmkoli způsobem pomohli s dokončením diplomové práce. V prvé řadě Lucii Svitákové za vedení práce a mnoho dobrých postřehů. Dále Petru Košvancovi, pod jehož vedením jsem pracoval na většině praktické části. Dále Jaroslavu Kotrčovi, Lukáši Hermannovi a všem ostatním kolegům z Manty, kteří byli vždy nápomocní. Na závěr bych chtěl poděkovat své rodině a přítelkyni za jejich neustálou podporu.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů (dále jen „autorský zákon“), především § 35 a § 60 autorského zákona upravující školní dílo.

V případě počítačových programů, jež jsou součástí mojí práce či její přílohou, a veškeré související dokumentace k počítačovým programům (dále jen „software“), uděluji v souladu s ust. § 2373 zákona 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, nevýhradní a neodvolatelné oprávnění (licenci) k užití software, a to všem osobám, které si přejí software užít. Tyto osoby jsou oprávněny software užít jakýmkoli způsobem a za jakýmkoli účelem v neomezeném rozsahu (včetně užití k výdělečným účelům), vč. možnosti software upravit či měnit, spojit jej s jiným dílem a/nebo zařadit jej do díla souborného. Toto oprávnění je časově, teritoriálně i množstevně neomezené a uděluji jej bezúplatně.

V Praze dne 27. května 2020 . . . . . .

Odkaz na tuto práci

Potočiar, Jan. *Analýza datových toků v nástroji SAP BusinessObjects*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2020.

Abstrakt

Cílem této práce bylo zanalyzovat datové toky v reportovacím nástroji SAP BusinessObjects a na základě této analýzy navrhnout a implementovat modul, který provede jejich integraci do nástroje Manta. V rámci práce byla také zanalyzována a použita metodika programování řízené testy (test-driven development). Pro implementaci byl použit programovací jazyk Java a několik podpůrných nástrojů jako Maven, JUnit či ANTLR. Výsledkem je funkční prototyp modulu, který byl úspěšně integrován do nástroje Manta.

Klíčová slova Manta, SAP BusinessObjects, test-driven development, Java, business intelligence, datové toky, report, reportovací nástroj

Abstract

The goal of this thesis was a data flow analysis in the reporting tool SAP BusinessObjects, a design, and implementation of a module, which integrates the data flows into Manta. The methodology test-driven development was analyzed and used during the work on the module as well. The implementation is written in Java; several other tools were used, such as Maven, JUnit, and ANTLR. A functional prototype of the desired module was successfully created and integrated into Manta.

Keywords Manta, SAP BusinessObjects, test-driven development, Java, business intelligence, data lineage, report, reporting tool

Obsah

Seznam obrázků

Seznam tabulek

Úvod

S rozvojem informačních technologií hrají data v naší společnosti stále důležitější roli. Data jsou sbírána, ukládána do databází, kopírována, přenášena, upravována, vizualizována, agregována, zabezpečována. S daty se provádí mnoho různých operací, jejichž složitost místy dosahuje už takových rozměrů, že je velmi obtížné se v nich vyznat. Zde přichází na pomoc Manta, která umožňuje operace s daty a jejich původ vizualizovat v podobě datových toků.

S daty se v rámci organizace manipuluje pomocí různých technologií – databází, skriptů, ETL či reportingových nástrojů a mnoha dalších. Manta se snaží pokrýt co největší portfólio těchto nástrojů, a do tohoto portfólia se rozhodla zařadit i technologii SAP BusinessObjects.

Cílem práce je navrhnout a implementovat konektor pro reportingový nástroj SAP BusinessObjects a integrovat jej s Mantou takovým způsobem, aby datové toky v tomto nástroji bylo možné vizualizovat v Mantě.

Práce je členěna do tří kapitol. V první kapitole se věnuji analýze souvisejících technologií (především Mantě a SAP BusinessObjects) a metodice, kterou jsem výsledný konektor implementoval – metodice zvané Test-Driven Development (TDD). Druhá kapitola obsahuje popis návrhu a implementace jednotlivých modulů, ze kterých se výsledný konektor skládá. Ve třetí kapitole se prezentují dosažené výsledky a jsou představeny možnosti dalšího vývoje konektoru.

Tato práce navazuje na diplomovou práci Ing. Petra Košvance, která se mimo jiné zabývala návrhem sjednoceného datového modelu pro analýzu datových toků reportingových nástrojů v rámci Manty.

Kapitola 1

Analýza

V první kapitole je věnován prostor analýze dvou hlavních nástrojů diplomové práce. V první sekci je popsán nástroj Manta, jeho vznik a souvislost se SAP BusinessObjects. Druhá sekce se věnuje technologii SAP BusinessObjects, především jeho hlavním objektům, které jsou relevantní pro analýzu datových toků.

1.1 Manta

Manta je souborem různých SW technologií, jejichž primárním cílem je pomoci zákazníkům analyzovat a vizualizovat toky dat napříč organizací. Jakožto nástroj je vlajkovou lodí společnosti Manta Tools, s.r.o.

1.1.1 Vznik

První verze nástroje Manta vznikaly v sesterské společnosti Profinit pro zákazníka v bankovním sektoru. Původně se nástroj nazýval „Profinit Syntax Analysis Tool“ a pomáhal s analýzou SQL výrazů v nástroji Teradata. Brzy poté se však jeho schopnosti rozšiřovaly, např. o analýzu XML. Jelikož se nástroj osvědčil, byl nasazen u mnohých zákazníků Profinitu ve střední Evropě. [4]

1.1.2 Expanze

V roce 2014 se povedlo projektu Manta získat první místo v soutěži Czech ICT Incubator @ Silicon Valley [5]. Díky tomu se zástupcům firmy povedlo dostat do Kalifornie, poznat tamější prostředí, a dokonce založit pobočku. Po pár letech firma přestěhovala svou pobočku do New Yorku, přičemž v Praze měla stále svou, především vývojářskou, základnu. Společnost Manta, která byla mezitím vyčleněna z firmy Profinit jako dceřiná a později sesterská společnost,

rozšířila svůj vliv kromě zahraničí i na české univerzity, mezi nimiž lze najít i FIT ČVUT.

1.1.3 Aktuální stav

Manta má aktuálně mnoho funkcionalit, přičemž primární funkcionalitou je vizualizace a analýza datových toků. Takové toky mohou pocházet nejen z Teradaty, ale i dalších databází, např. Oracle, Hive, Microsoft SQL Server, IBM DB2, PostgreSQL a dalších [6]. Následně mohou být data integrována nástroji jako IBM InfoSphere DataStage, Informatica Power Center, Talend a dalšími [7]. Rovněž mohou být zanalyzovány datové toky v programovacím jazyce Java [8] či některém z modelovacích nástrojů SAP Power Designer, ER/Studio či erwin Data Modeler [9]. A také mohou datové toky proudit do reportingových nástrojů jako Tableau, Power BI, Oracle Business Intelligence Enterprise Edition, Cognos, Microsoft Excel a dalších [10]. Do této skupiny náleží i nástroj SAP BusinessObjects, jehož integrace do Manty je předmětem této diplomové práce.

1.1.4 BI nástroje a Manta

SAP BO je jedním z mnoha používaných business intelligence (BI) nástrojů. Dalšími jsou např. Microsoft SQL Server Analysis Services, IBM Cognos Analytics, Microsoft Excel, Microsoft Power BI, Tableau či Oracle Business Intelligence Suite Enterprise Edition. Všechny vyjmenované nástroje jsou aktuálně podporované Mantou skrze tzv. scannery, které byly vyvinuty a jsou udržovány ostatními kolegy.

Jelikož si jsou jednotlivé BI nástroje velmi podobné, i implementace jejich scannerů může být velmi podobná, což s sebou nese mnoho výhod – např. pokud má dojít k vytvoření nového scanneru, lze se u ostatních při dodržení společné struktury velmi dobře inspirovat, rovněž pokud někdy programátor pracoval na scanneru pro jeden BI nástroj, rychle se zorientuje i v jiném scanneru. Zároveň bylo nutné řešit problém s integrováním grafu datových toků do nástrojů třetích stran, a proto vzniklo společné rozhraní pro reportingové nástroje pod vedením Petra Košvance s pomocí od ostatních kolegů. [11]

Nástroje třetích stran, do kterých Manta umožňuje export, jsou následující:

- Collibra – nástroj pomáhající s chápáním dat a rozhodováním na jejich základu [12].
- Informatica Enterprise Data Catalog (EDC) – katalog dat postavený na strojovém učení, který pomáhá klasifikovat a organizovat data napříč organizací [13].
- IBM InfoSphere Information Governance Catalog (IGC) – webový nástroj, který umožňuje procházet, chápat a analyzovat informace [14].

Dalším reportingovým nástrojem jsou např. SAP Crystal Reports, jedná se tedy o BI software od stejné firmy jako BusinessObjects. Tyto dvě platformy mají mnoho společného, dokonce se dají provozovat na stejném serveru, nicméně dají se provozovat i odděleně. Mají rozdílné klientské aplikace a rovněž mají rozdílnou strukturu reportů. Kvůli přílišné odlišnosti se v této práci věnuji pouze platformě BusinessObjects.

1.2 SAP BusinessObjects

Na stránkách výrobce se SAP BusinessObjects (SAP BO) představuje jako BI (Business Intelligence) centralizovaná platforma pro reporting a vizualizaci s flexibilní a škálovatelnou architekturou, s mnoha podpůrnými aplikacemi a optimalizovaným výkonem [15].

SAP BO se řadí do skupiny reportingových (BI) nástrojů (jako jsou MS Excel, Cognos Analytics a další), s nimiž má mnoho podobného – od funkcionality, po architekturu a vizuální stránku.

Proč používat BusinessObjects, když existuje tolik alternativ? Jedním z hlavních důvodů zřejmě bude celá platforma, kterou SAP poskytuje, která je do BusinessObjects dobře integrovatelná (např. databáze SAP HANA a další). Pokud tedy organizace používá software z portfolia firmy SAP, proč využívat BusinessObject a ne Crystal Reports (jiný BI nástroj od firmy SAP)? Důvodem je škálovatelnost – Crystal Reports jsou mířené na menší a středně velké podniky, zatímco BusinessObjects se dají velmi dobře škálovat. Oproti jiným reportingovým nástrojům navíc poskytují pevné zázemí ve formě organizace SAP, schopnost zvládat komplexní reportingové úlohy, integrace s mnoha dalšími technologiemi či solidní administrativa a data governance. S tím vším se ale pojí vyšší cena, potenciálně vyšší požadavky na hardware (především servery) či vyšší technická zdatnost, neboť se jedná o poměrně komplexní systém. [16]

1.2.1 Platforma

SAP BO platforma se skládá ze dvou balíčků aplikací – Server a tzv. Client Tools. Tyto budou nyní samostatně představeny.

1.2.1.1 Server

Serverová část vychází pro operační systémy rodiny Windows i Linux - nicméně pouze linuxové distribuce Red Hat linux a Suse.

Server má na starost několik funkcí, především slouží jako úložiště reportů a všech souvisejících objektů. Většina metadat týkající se těchto objektů je k dispozici skrze RESTové rozhraní. Jádro serverové aplikace je nasazeno na aplikačním serveru Tomcat (který je automaticky nainstalován společně se SAP BO). K dispozici je i grafické webové rozhraní, pomocí něhož lze jako

uživatel prohlížet reporty, a jako administrator lze spravovat přístupy a veškerá potřebná nastavení.

1.2.1.2 Client Tools

Client Tools je balíček aplikací určený především pro tvůrce reportů a univerz, ale lze jej použít i pro jejich čtení a vizualizaci. K dispozici pouze pro operační systém rodiny Windows.

Pro potřeby závěrečné práce bylo nutné pracovat především s těmito nástroji:

- Web Intelligence Rich Client – tento nástroj slouží pro vytváření, editaci a zobrazování reportů. Reporty může načíst a uložit buď lokálně, nebo se lze připojit na některý ze SAP BO serverů.
- Universe Design Tool (UDT) – tato aplikace slouží ke všemu týkající se objektů Universe staršího formátu UNV. K API této aplikaci lze přistoupit skrze její COM SDK.
- Information Design Tool (IDT) – lze pokládat za novější verzi UDT, která má na starost novější formát univerz – UNX. Umožňuje rovněž převod staršího formátu objektů Universe na novější formát. K API této aplikaci lze přistoupit skrze její Java SDK.

1.2.2 Objekty zájmu

Z pohledu analýzy datových toků nás zajímají především objekty na diagramu1.1, které budou v dalších podsekcích detailněji popsány.

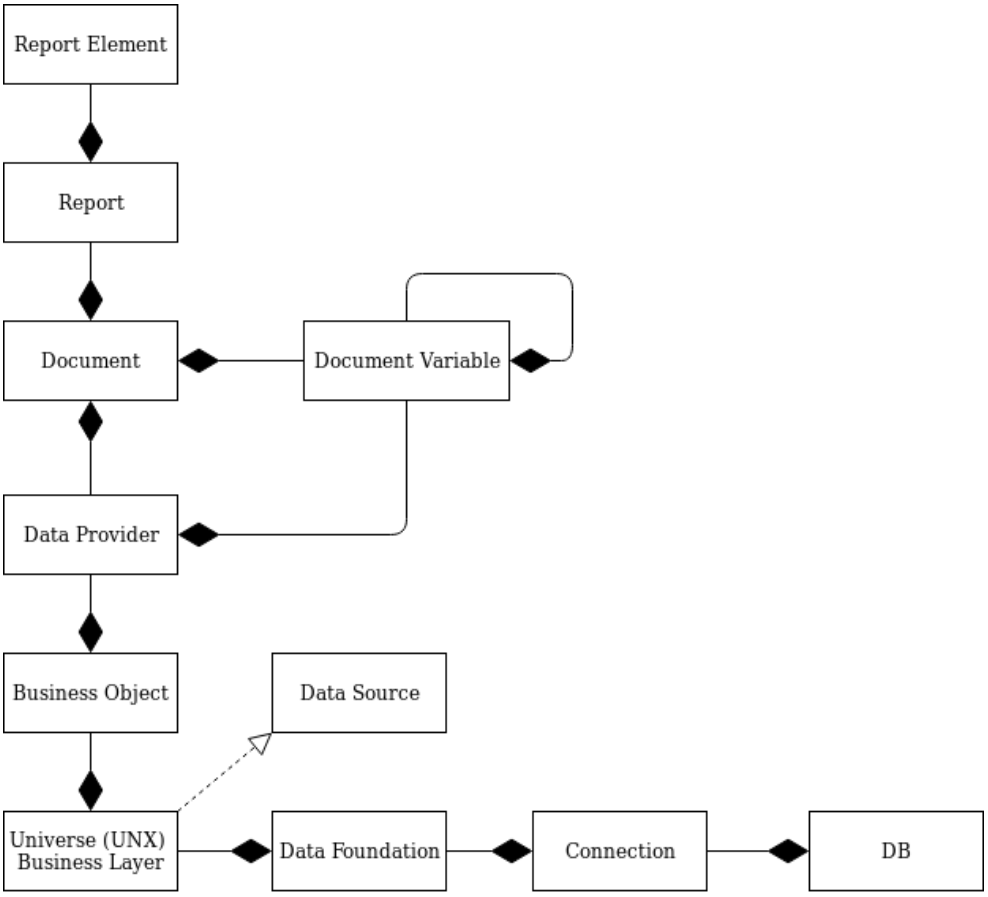
1.2.2.1 Document

Dokumenty jsou základní jednotkou se kterou se pracuje na prezentační vrstvě aplikace. Dokument je to, co vidí koncoví uživatelé. S louží především pro agregaci relevantních objektů a na jejich úrovni se definují tzv. „Document Variable“.

V kontextu Manty a jejího interního názvosloví pro reportingové nástroje se jedná o objekt typu „Report“, což může být matoucí.

Dokumenty obvykle obsahují:

- Název
- ID – unikátní identifikátor napříč dokumenty na daném serveru
- Cestu k souboru
- Reporty



Obrázek 1.1: Diagram relevantních objektů modelu SAP BO a vztahů mezi nimi

- Data Providery
- Proměnné (Document Variable)

Cesta k souboru společně s názvem tvoří jednoznačný identifikátor dokumentu na daném serveru.

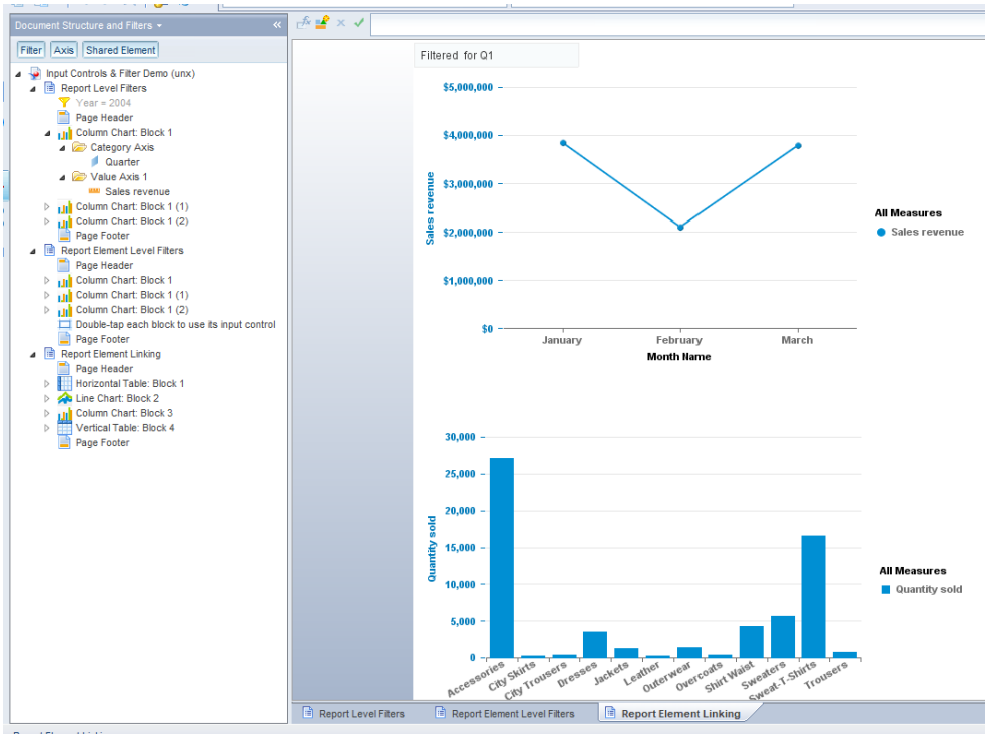
Na obrázku 1.2 je příklad dokumentu s názvem „Input Controls & Filter Demo“.

1.2.2.2 Report

Jednotlivé dokumenty jsou dělené do jednotlivých reportů, přičemž jeden report ztělesňuje jednu stránku či záložku dokumentu.

V interním názvosloví Manty se jedná o objekty typu „Page“.

Reporty obvykle obsahují:



Obrázek 1.2: Snímek obrazovky dokumentu otevřeném v nástroji Web Intelligence Rich Client [1]

- Název
- ID – unikátní identifikátor napříč reporty daného dokumentu
- Report Elementy

Na obrázku 1.2 je příklad dokumentu se třemi reporty – Report Level Filters, Report Element Level Filters a Report Element Linking. Celá struktura dokumentu (včetně report elementů) lze vidět v levém menu, ve spodním menu lze přepínat mezi jednotlivými reporty. Na obrázku je zobrazen report s názvem Report Element Linking.

1.2.2.3 Report Element

Reporty obsahují tzv. report elementy.

V interním názvosloví se jedná o objekty typu „Report Item“.

Report elementy obvykle obsahují:

- Název

- ID – unikátní identifikátor napříč elementy daného reportu

Navíc existuje několik různých druhů report elementů, které dále určují strukturu objektu report element. Každý Report element má přiřazen právě jeden typ.

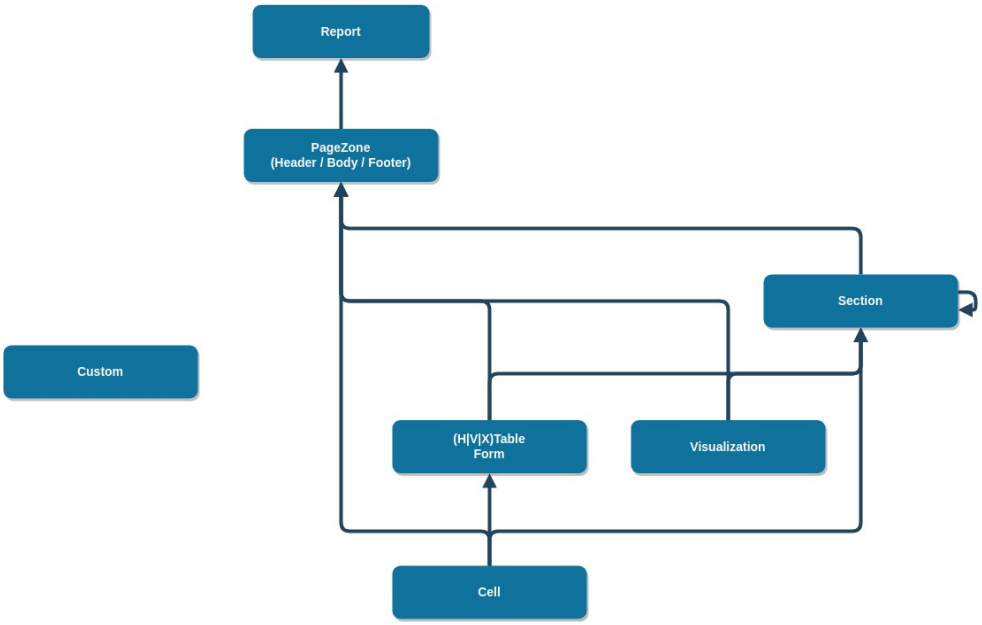
Jednotlivé typy:

- PageZone – slouží ke grafickému rozvržení stránky a jako kontejner pro ostatní prvky. Nabývá typu header, body nebo footer.
- Cell – odpovídá jednomu typu buněk, základní stavební buňka pro tabulkové typy elementů. Obvykle obsahuje jediný výraz, který určuje její hodnotu.
- XTable – dvourozměrná tabulka, v základním rozložení může obsahovat až 9 různých typů buněk.
- VTable – vertikální jednorozměrná tabulka.
- HTable – horizontální jednorozměrná tabulka.
- Form – speciální typ tabulkového elementu.
- Section – kontejner pro jiné elementy, který se nakopíruje pro každou hodnotu dané dimenze a takovou hodnotu aplikuje na své potomky. Odpovídá různým hodnotám jednoho řezu datovou kostkou [1].
- Visualization – různé druhy vizualizací, grafů, map a dalších grafických elementů.
- Custom – speciální typ elementů, které si může zákazník sám nadefinovat. Jelikož ve vzorových reportech nebyl žádný prvek tohoto typu a jeho vytváření není triviální záležitostí, nebyl tento typ dále analyzován.

Různé druhy report elementů se mohou různě zanořovat. V žádné dokumentaci jsem tyto vztahy nenašel, jedná se o vztahy založené na analýze dodaných reportů. Zjištěné vztahy jsou zakresleny na diagramu 1.3.

Reporty pod sebou tedy přímo obsahují pouze report elementy typu „PageZone“. Pod elementy typu „PageZone“ mohou být všechny typy elementů kromě nich samých. Elementy typu XTable, HTable, VTable a Form jsou v diagramu sloučeny v jeden uzel, jelikož u nich byl zjištěn stejný vztah co se týče zanořování mezi různými typy elementů. Tyto elementy pod sebou mohou obsahovat pouze elementy typu Cell. Element typu Section je jediným, u kterého byla zjištěna možnost „seberodičovství“ – tedy že potomkem elementu typu Section může být další Section. Elementy typu Visualization a Cell pod sebou

[1] Datovou kostkou, či OLAP krychlí, je myšleno multidimenzionální pole dat.



Obrázek 1.3: Zjištěné vztahy rodičovství různých typů Report Elementů. Šipky směřují k možnému rodiči.

již nemívají žádné další elementy. Report Element typu „Custom“ stojí stranou, jelikož ve vzorových datech jsem na žádný element tohoto typu nenarazil, a proto jsem nemohl zanalyzovat jeho vztah s ostatními elementy.

Na obrázku 1.2 je zobrazen report s názvem Report Element Linking, obsahující několik elementů, přičemž viditelné jsou především dva elementy, oba různého typu vizualizace – Block 2 (Line Chart) a Block 3 (Column Chart). Ačkoliv to v levém menu není zobrazené, oba jsou vnořeny do elementu typu PageZone typu body.

1.2.2.4 Data Provider

Data Provider je objekt, který danému Dokumentu zprostředkovává data. Jeden Dokument může obsahovat několik různých Data Providerů. Jedná se o jakousi abstrakci, neboť Data Provider může čerpat data z různých druhů Data Source, ale sám o sobě poskytuje jednotné rozhraní pro přístup k datům, nezávisle na zdroji dat. Data Provider obsahuje seznam objektů, každý je unikátně identifikovatelný skrze své „formulaLanguageId“ – toto id je následně používané ve výrazech Report Elementů.

1.2.2.5 Document Variable

Document Variable je objekt definovaný na úrovni Dokumentu. Definicí této proměnné je výraz, stejný jako se používá v Report Elementech. Motivací pro vytvoření nové proměnné může být výraz, který se používá na několika různých místech najednou a bylo by tudíž dobré mít tuto stejnou informaci na jednom místě pojmenovanou a definovanou. Podobně jako objekty Data Provideru, i tato proměnná je unikátně identifikovatelná skrze své „formula-LanguageId“, čehož se využívá v Report Elementech či dalších proměnných.

1.2.2.6 Data Source

Data Source je zdrojem dat, který může být použit pro různé Data Providery.

Jednotlivé typy:

- Universe – definovaný níže.
- Excel – dokument vytvořený ve stejnojmenném nástroji.
- BEx – SAP Business Explorer, další sada nástrojů v oblasti BI od společnosti SAP.
- SAP HANA – typ databáze od společnosti SAP.
- Analysis View – speciální objekt definovaný v nástroji SAP NetWeaver BW (typ datového skladu od společnosti SAP).
- Text – textový soubor, podporované formáty jsou TXT, CSV, PRN a ASC.
- Web Service – přijímá jako zdroj URL definované web service.
- Free-hand SQL – manuálně definovaný SQL příkaz nad databází.

1.2.2.7 Universe

Objekty typu Universe jsou speciální modely postavené nad různými zdroji dat, které poskytují přístup k těmto datům skrze jednotné abstraktní rozhraní ve formě objektů typu Business Object. Univerza tvoří jediný Data Source, který je aktuálně podporován v konektoru pro SAP BO.

Jsou dvojího typu:

- UNV – starší formát. K manipulaci univerz tohoto typu slouží nástroj UDT. Blíže popsáno v sekci 1.2.1.2.
- UNX – novější formát. K manipulaci univerz tohoto typu slouží nástroj IDT. Blíže popsáno v sekci 1.2.1.2.

1.2.2.8 Business Object

Business objekty tvoří obsah univerz a jsou zároveň objekty, na které se odkazuje Data Provider dokumentu.

Existují 4 typy business objektů [17] [18]:

- Dimension – hlavní objekty analýzy určující kontext, např. služba či zákazník. Obvykle obsahují jména nebo kalendářní data.
- Attribute – vždy navázaný na objekt typu dimension nebo measure, obsahuje dodatečná data týkající se daného objektu, např. pro dimension typu zákazník by mohly být atributy adresa či telefonní číslo.
- Filter – objekt ztělesňující podmínku, např. pro výběr objektů dle určitého roku či lokace.
- Measure – statistická informace číselného typu, vzniká aplikací funkcí jako počet, součet, minimum, maximum, průměr apod. Příkladem může být obrat či cena za jednotku.

1.2.2.9 Connection

Connection je objekt obsahující parametry připojení k relační databázi. Existují dva typy:

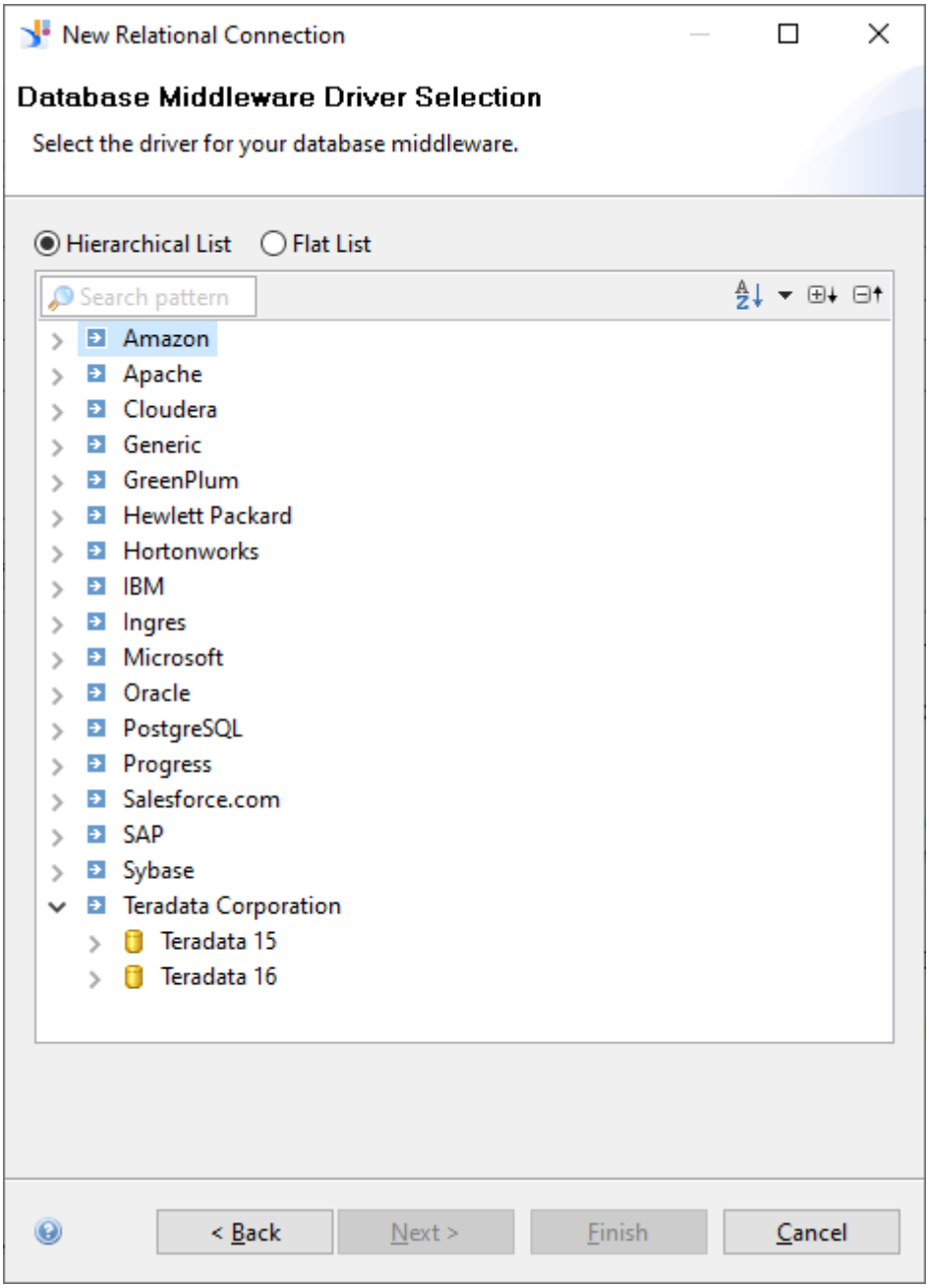
- Lokální, nezveřejněné na SAP BO serveru
- Zveřejněné, zabezpečené

Druhů relačních databází, nad kterými může být Connection postaveno, je mnoho, jak dokazuje obrázek 1.4.

1.2.2.10 Data Foundation

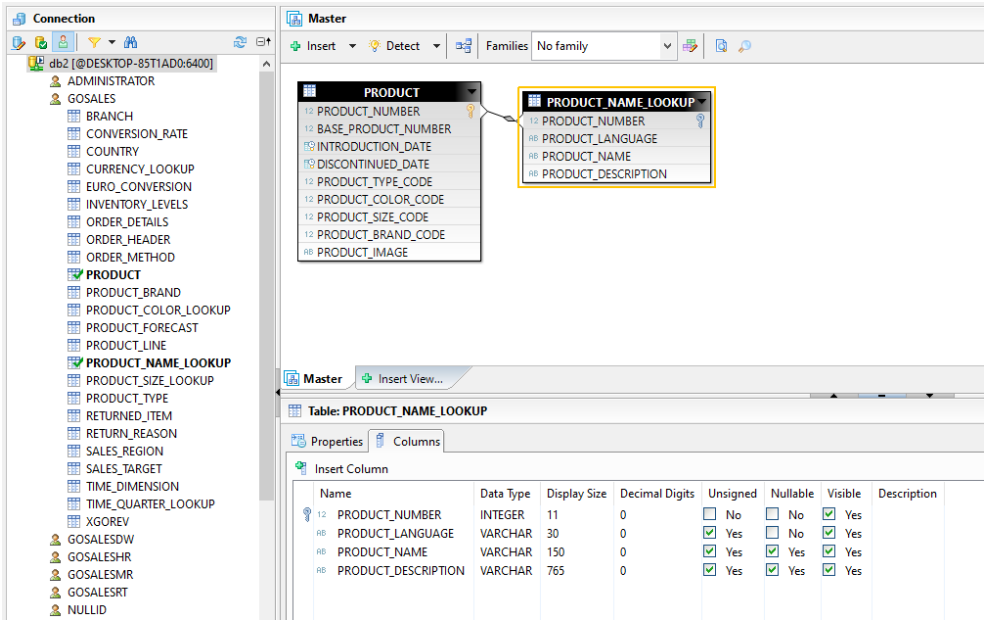
Data Foundation je speciální vrstvou vytvořenou nad jedním či více objekty typu Connection. Existují dva typy Data Foundation:

- Single-source, umožňující být postavené pouze nad jedním objektem typu Connection. Má výhodu, že toto připojení může být definováno pouze lokálně a i Univerzum vytvořené nad tímto Data Foundation může být uloženo lokálně.
- Multi-source, umožňující být postavené nad jedním či více objektech typu Connection. Tato připojení musí být zveřejněná a zabezpečená na daném SAP BO serveru, kde musí být rovněž zveřejněné Univerzum založené na tomto Data Foundation. [19]



Obrázek 1.4: Různé druhy podporovaných databází, dialog v aplikaci IDT [1] při vytváření nového Connection.

V Data Foundation jsou definované tabulky (a jejich sloupce), které jsou podmnožinou tabulek databází, nad jejichž Connection je Data Foundation postaveno. Rovněž lze nad těmito tabulkami definovat JOINy, které ale nemusí



Obrázek 1.5: Příklad Data Foundation, zobrazené v nástroji IDT [1].

být podmnožinou definovaných JOINů v původních databázích – toho se dá využít např. pro definování vztahů mezi tabulkami různých databází.

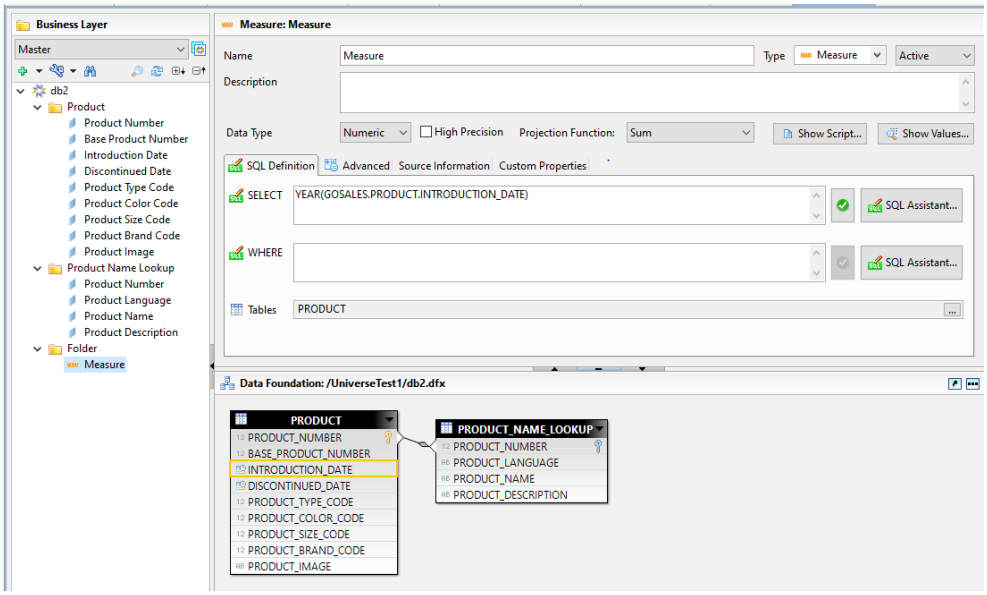
Na obrázku 1.5 lze vidět příklad single-source Data Foundation, kde v levém menu lze vidět dostupné objekty typu Connection a jejich schémata, tabulky, sloupce a další relevantní objekty (v tomto konkrétním případě byla jako zdroj použita DB2 databáze). V hlavním okně pak lze spatřit vybrané tabulky, sloupce a joiny, které tvoří danou Data Foundation.

1.2.2.11 Business Layer

Business Layer je objekt vytvořený na základě jednoho Data Foundation. Jak lze vypozorovat na obrázku 1.6, v levém menu je seznam vytvořených Business Objectů. v pravém horním okně detail vybraného Business Objectu (včetně SQL definice) a v pravém dolním okně je Data Foundation se zvýrazněnými sloupci, kterých se vybraný Business Object týká.

Platí, že na jednom Data Foundation lze postavit několik objektů typu Business Layer.

Jakmile je práce na Business Layer hotová, lze výsledek nahrát a zveřejnit na SAP BO serveru, a od této chvíle lze tento objekt používat jako UNX Univerzum.



Obrázek 1.6: Příklad Business Layer, zobrazené v nástroji IDT [1].

1.2.3 Mapování objektů na společný reportingový model

Petr Košvanec ve své práci [11] navrhl společný model pro reportingové nástroje. Tento model byl navržen do značné míry induktivně z BI nástrojů, pro které se tehdy v Mantě vyvíjely konektory – SQL Server Reporting Services, Oracle Business Intelligence Enterprise Edition, Microsoft Excel a IBM Cognos.

Z tabulky 1.1 lze vyčíst, že se téměř všechny objekty daly namapovat na společný reportingový model. Výjimkou je Document Variable, což by se dalo namapovat na Field, pokud by definice tohoto objektu umožňovala prvku existovat bez rodiče (Data set). Druhou výjimkou je Business Object typu Filter, pro který ve společném modelu neexistuje obraz. Jelikož úplné mapování na společný reportingový model není vyžadováno, tyto dvě výjimky ničemu nevadí.

Tabulka 1.1: Mapování SAP BO objektů na společný reportingový model.

SAP BO objekt	**Vrstva**	**Uzel společného modelu**
Document	prezentační	Report
Report	prezentační	Page
Report Element	prezentační	Report Item
Document Variable	logická	x
Data Provider	logická	Data set
Data Provider expression	logická	Field
Universe	analytická	Cube
Business Object (Measure)	analytická	Measure
Business Object (Dimension)	analytická	Dimension
Business Object (Attribute)	analytická	Attribute
Business Object (Filter)	analytická	x
Data Foundation	analytická	Model
Data Foundation table	analytická	Table
Data Foundation column	analytická	Column

Kapitola 2

Technologie

V druhé kapitole práce se budu věnovat ostatním technologiím, se kterými jsem během práce na konektoru přišel do kontaktu. Jmenovitě se jedná o programovací jazyk Java, testovací framework JUnit, buildovací nástroj Maven, verzovací systém SVN, integrační server Jenkins a generátor parseru ANTLR.

2.1 Java

Jelikož je veškerá část Manty, se kterou jsem přišel do styku, naprogramovaná v Javě, byla i pro implementaci mého konektoru zvolena Java. Java přišla vhod i díky tomu, že pro extrakci potřebných dat ze SAP BO serveru bylo nutné využít dodávané Java SDK – tedy žádný jiný přístup k některým klíčovým částem dat než skrze Javu není podporován.

Java se ve firmě používá ve verzi 8, což s sebou nese své výhody, ale i omezení, která jsou popsána níže.

2.1.1 Java Streams

Java 8 s sebou oproti starším verzím přináší mimo jiné i Java Streamy (Java Streams).

Java Streams tvoří novou abstraktní vrstvu, která umožňuje zpracovávání a přístup k datům deklarativním [20] a funkcionálním stylem – na rozdíl od spíše imperativního přístupu u klasických kolekcí. Od klasických kolekcí se streamy liší několika vlastnostmi:

- Žádné ukládání – streamy nejsou datová struktura, data tedy neukládají, ale přepravují. Zdroji takových data mohou být kolekce, pole, generátory či I/O kanály.
- Funkcionální – operace nad streamem produkuje výsledek, ale nemodifikuje původní zdroj dat.

- „Laziness-seeking“ – mnohé operace nad streamy , jako filtrování, mapování, nebo odstranění duplicit, mohou být implementovány „lazy“ způsobem (tedy operace se nemusí provést hned, ale lze ji odložit na později), což poskytuje příležitosti k optimalizaci.
- Potenciálně neomezené – zatímco kolekce mají konečnou velikost, streamy mohou být ze své podstaty neomezeně velké.
- Spotřební – jednotlivé elementy streamu jsou po dobu života streamu navštívené pouze jednou. [21]

Díky funkcionální nátuře streamů lze jednoduše docílit paralelního zpracování, což může při správné implementaci znatelně urychlit výpočty na vícejádrových procesorech. Paralelní streamy tak mohou postytovat jednoduchý mechanismus, kterým lze urychlit výpočty v budoucnosti, pokud počet jader v procesorech bude růst.

Nad streamy jsem používal především tyto operace:

- map – jako argument přijímá funkci, která se provede nad každým prvek, a takto vzniklé prvky tvoří nový stream, který tato funkce vrací.
- filter – jako argument přijímá funkci, která přijímá jako vstup prvek streamu, a jeko výstup vrací boolean – tedy buď true nebo false. Výsledkem je stream pouze takových prvků, které vyhověly filtrovací funkci (funkce pro ně vrátila true).
- foreach – pro každý prvek streamu provede určitou funkci.
- collect – umožňuje vrátit stream jako kolekci, např. List nebo Set.

2.1.2 Optional

Společně s Java Streamy přišla do Javy 8 i podpora struktury Optional, která pomáhá řešit problém nakládání s null hodnotami.

Jedním z problémů, které struktura Optional řeší, je jak co nejlépe vracet prázdné výsledky v případě referenčního typu. V případě kolekcí se nabízí přímočaré řešení - vrátí se prázdná kolekce. Co ale když se vrací referenční typ, který není kolekcí? Co by měla taková funkce vracet, jak by měla vypadat?

Po diskuzi s kolegy jsme dospěli k následujícím možnostem:

1. V případě prázdného výsledku vracet null, ale taková možnost by měla být dokumentovaná v javadocu příslušné funkce.
2. Využít anotaci @Nullable pro indikaci možné návratové hodnoty null v případě prázdného výsledku.
3. Použití konstruktu Optional, který umožňuje pracovat s prázdnou návratovou hodnotou bez používání hodnoty null.

Všechna řešení mají své výhody a nevýhody. První řešení nenutí kompilátor kontrolovat, zda je v klientském kódu metody nějak speciálně řešen null, což může při opomenutí vést k neslavným NullPointerException (NPE). Druhý případ, pokud je integrován do IDE, poskytuje více kontroly než první. Má však velkou nevýhodu ve své nejednotnosti – anotace není v samotné Javě podporovaná, JSR 305 [2], jejíž je součástí, je ve stavu „dormant“ – neimplementovaná [22]. To mělo za důsledek, že vniklo několik různých implementací této anotace – např. anotace od JetBrains [23], Eclipse [24] či Spring implementace [25]. Třetí možnost je ze všech zmíněných nejdůslednější ve své kontrole – nutí klientský kód speciálně řešit možnost prázdného výsledku. Oproti anotacím je tento konstrukt součástí Javy 8.

Nicméně používání Optional není tak přímočaré, jak by se mohlo zdát – má dobře definované své způsoby užití a nebylo by správné jej nadužívat a používat všude kde by mohlo nějakým způsobem zacházet s null hodnotou.

Brian Goetz, architekt Javy, uvádí, že Optional je zamýšlen k poskytování omezeného mechanismu návratových typů metod knihoven, kde je potřeba čistší způsob, jak reprezentovat prázdný výsledek, a používání null pro takové případy by mohlo působit chyby [26].

Joshua Bloch ve své knize Effective Java sdílí několik postřehů doporučení:

- Nikdy nevracet null z metody, která má vracet Optional: to jde proti smyslu Optional.
- Optional je v podstatě velmi podobný checkovaným výjimkám.
- Kontejnerové typy zahrnující kolekce, mapy, streamy, pole, a Optional by nikdy neměly být zabaleny do Optional - místo toho by měly vracet prázdný kontejner.
- Nepoužívat Optional v případě referenčních typů primitivních typů (jako Integer, Long, Double).
- Není téměř nikdy vhodné používat Optional jako klíč, hodnotu, či element v kolekci či poli.
- Deklarovat návratový typ metody Optional, pokud metoda nemusí vždy vracet výsledek a klienti budou muset tuto skutečnost speciálně zpracovat [27].

Stephen Colebourne přišel s dalším seznamem pravidel pro vhodné zacházení s Optional [28]:

- Nedeklarovat instanční proměnné typem Optional.
- Používat null pro indikaci možných null hodnot v privátních oblastech třídy.

[2] JSR – Java Specification Request

- Používat Optional pro gettery které přistupují k potenciálně prázdné instanční proměnné.
- Nepoužívat Optional v setterech nebo konstruktorech.
- Používat Optional jako návratový typ pro metody business logiky, které mohou mít prázdný výsledek.

Ve svém řešení používám Optional pouze jako návratové typy metod, kde se v klientovi musí speciálně zacházet s prázdným výsledkem, což přesně odpovídá doporučením zde popsaným. Jedinou výjimkou je použití Optional jako argumentů v některých pomocných privátních metodách, kde by při snaze o nepoužívání Optional vzniklo mnoho kódu navíc, který by neměl žádný speciální význam, kromě znepřehlednění dosavadního kódu.

2.2 JUnit

JUnit je jedním z testovacích frameworků pro Javu. První verze frameworku JUnit vznikla na palubě letadla letícího z Atlanty do Curychu, kdy Kent Beck (jedna z hlavních postav TDD[3] a Extrémního programování[4]) a Erich Gamma (známý především jako člen „Gang of Four“) spojili síly a vytvořili prototyp frameworku JUnit. Od té doby se framework dál vyvíjel, vylepšoval a rozšiřoval, dnes již máme na světě verzi 5[5]. Pro testování konektoru pro SAP BO byla použita o něco starší verze frameworku, verze 4, protože to je defaultní verze knihovny ve firemním repozitáři.

2.3 Maven

O automatizaci buildů se stará nástroj Maven. Bylo tak nutné rozdělit zdrojový kód do jednotlivých maven modulů, každý se svými definovanými závislostmi. Je důležité poznamenat, že výsledný graf závislostí musí tvořit stromovou strukturu, tedy nejsou povoleny cyklické závislosti.

2.4 SVN a Jenkins

Pro verzování softwaru byl použit nástroj SVN [29], protože se jedná o verzovací nástroj používaný interně ve společnosti Manta Jakmile skrze SVN dojde ke commitu kódu a aktualizování modulu, Jenkins tuto skutečnost zaznamená a automaticky pustí build aktualizovaných modulů. Jenkins [30] je volný nástroj běžící na interním serveru, který pomáhá automatizovat vývojový proces – spouští buildy, testy, dokáže i nasadit vytvořený build na další prostředí.

[3] TDD – Test-Driven Development
[4] Extrémní programování je jednou z agilních metodologií vývoje softwaru
[5] https://junit.org/junit5/

Tento nástroj tedy pomáhá s průběžnou integrací (continuous integration – CI) a průběžným nasazováním (continuous delivery – CD). Může být i dobře upravován a rozšiřován o další procesy, např. statickou analýzu kódu, který se rovněž provádí s buildem modulu, a tím pomáhá udržet lepší kvalitu kódu (za předpokladu, že jeho varování nejsou programátory ignorována).

2.5 Spring

Pro snadnější konfiguraci je využíváno frameworku Spring a jeho mechanismu dependency injection (DI).

Proč vůbec používat DI framework? Ve zkratce – jedná se o oddělení dvou různých fází projektu – jeho konstrukce a jeho běžného chodu. Dr. Kevin Dean Wampler píše v kapitole *Systems* knihy Clean Code [31] o snadno pochopitelnému přirovnání k tomu, jak fungují města a budovy. Píše, že konstrukce a běžný provoz jsou dva velmi odlišné procesy. Když se staví budova, vypadá úplně jinak než již postavená budova – je tam např. lešení, stavební dělníci, pomocné stoje na stavbě apod. V již postavené budově a uvedené do provozu žádná lešení nebývají, je tam zaměstnaný jiný personál, a rovněž i pomocné stroje tam bývají úplně jiné. Dr. Wampler doporučuje, aby softwarové systémy oddělovaly proces inicializace, kdy jsou aplikační objekty konstruovány a jednotlivé závislosti „propojovány“ dohromady, od logiky běhu programu, která je spuštěna po dokončení inicializace projektu.

A právě zde nastupuje Spring.

Veškerá konfigurace je definována v XML a *.properties souborech, jedná se o použití DI bez používání anotací (jako `@Autowired` a podobných). Tento přístup má několik výhod. Např. tu, že veškerá konfigurace je centralizovaná v XML souborech, namísto roztříštěnosti napříč zdrojovým kódem. Díky tomu pak může být jednodušší na pochopení konfigurace projektu. Zároveň je možné kdykoliv upravit pouze požadované XML soubory kdykoliv chceme upravit konfiguraci, na rozdíl od přístupu pomocí anotací, kdy je pak potřeba i znovu kompilovat zdrojový kód. Dále je tu důležitý prvek neinvazivnosti – zdrojový kód je čistá Java, kdykoliv dojde k nějaké modifikaci frameworku, zdrojový kód zůstane nedotčen, bude potřeba pouze změnit XML soubory. Z toho vyplývá, že XML konfigurace vede k dobrému oddělení dvou různých fází projektu (inicializace a běžného chodu), a to i na úrovni kódu.

2.6 ANTLR

ANTLR (ANother Tool for Language Recognition) je nástroj, pomocí kterého lze z předepsané gramatiky vygenerovat příslušný parser (ve formě zdrojového kódu) pro jazyk, který je potřeba parsovat. ANTLR je používám v takových projektech, jako je Twitter, Hive, Pig, Hadoop, SQL Developer, NetBeans či Hibernate. [32]

V nástroji se rozlišují dva typy pravidel – pravidla pro lexer a pravidla pro parser. Důležité je poznamenat, že názvy pravidel pro lexer jsou tvořeny velkými písmeny, zatímco názvy pravidel pro parser jsou tvořeny malými písmeny. K čemu ale přesně slouží tyto dvě části?

Lexer je modul s následujícím rozhraním:

- Vstup – řetězec znaků.
- Výstup – seznam tokenů.

Parser navazuje na lexer s rozhraním:

- Vstup – seznam tokenů.
- Výstup – abstraktní syntaktický strom, známý pod označením AST (abstract syntax tree).

Společně tak oba moduly vytvoří z řetězce znaků abstraktní syntaktický strom, který je na interpretaci jednodušší než původní řetězec znaků.

Kapitola 3

Návrh

V rámci praktické části bylo potřeba navrhnout a implementovat jednotlivé moduly tak, aby byly snadno integrovatelné s architekturou Manty, kterou lze vidět na diagramu 3.1. V části „Manta Flow CLI“ lze vidět dva související moduly, v diagramu je lze najít pod názvy „Extractor“ a „Analyzer“ – přesně tyto moduly, resp. jejich rozhraní, bylo potřeba brát při návrhu v potaz.

Výsledkem praktické části práce jsou moduly zobrazené na diagramu 3.2 (maven artefakty). Šipkou jsou vyjádřeny závislosti:

V následujících kapitolách jsou popsány a na příkladech vysvětleny jejich základní zodpovědnosti.

3.1 Extractor

Modul extractor slouží k získání všech objektů určených k následnému zpracování a uložení do definované adresářové struktury.

Zjednodušené rozhraní extractoru:

- Vstup – parametry připojení k SAP BO serveru.
- Výstup – složka se staženými soubory – reprezentace objektů zájmu, které chceme v budoucnosti analyzovat.

3.1.1 Objekty zájmu

V rámci jeho vývoje jsem se musel nejprve seznámit s prostředím SAP BO, jednotlivými objekty, vztahy a toky dat mezi nimi, a zároveň i jakým způsobem (API) a v jakém formátu lze objekty získat.

3.1.1.1 Způsoby získávání souborů

Veškeré objekty lze získat jedním některým ze dvou způsobů:

1. REST API

2. Java SDK API

Preferován je první způsob získávání potřebných dat skrze REST požadavky, jelikož oproti Java SDK nevyžaduje žádné speciální běhové prostředí. Rozhraní Java SDK má naopak v případě nástroje SAP BO poměrně exotické požadavky (exotické na poměry ostatních konektorů v Mantě). Tyto požadavky vycházejí z původu tohoto SDK – sady aplikací Client Tools, což jsou nástroje určené k instalaci na osobní počítače s rodinou operačních systémů Windows.

Požadavky na úspěšné užívání Java SDK:

- Build path – je potřeba dodat seznam jar souborů dodávaných s Client Tools.
- Java Virtual Machine (JVM) flag – je nutné říci extractoru, jaká je cesta ke složce connectionServer. Výchozí hodnota (pokud uživatel složku nikam nepřesouval) je

  ```
  -Dbusinessobjects.connectivity.directory="C:\Program
  ↪ Files (x86)\SAP BusinessObjects\SAP BusinessObjects
  ↪ Enterprise XI 4.0\dataAccess\connectionServer"
  ```

- 32-bit Java Runtime Environment (JRE) ve verzi 8 – JRE v této verzi je používané pro běh potřebných programů a je nutné jej využívat i pro spouštění extractoru. Je dodávané s Client Tools, lze jej tak přepoužít i pro extractor. Toto JRE se nachází na cestě:

  ```
  C:\Program Files (x86)\SAP BusinessObjects\SAP
  ↪ BusinessObjects Enterprise XI 4.0\win32_x86\jre8
  ```

- Proměnná prostředí (environment variable) – je potřeba přidat do proměnné PATH cestu ke složce s binárními soubory souvisejících s Client Tools. Výhozí hodnotou je

  ```
  C:\Program Files (x86)\SAP BusinessObjects\SAP
  ↪ BusinessObjects Enterprise XI 4.0\win32_x86
  ```

- Nainstalované Client Tools na počítači (s operačním systémem rodiny Windows) – plyne z předchozího bodu. Odkazovaná složka obsahuje mnoho dynamicky linkovaných knihoven, které se odkazují na jiné knihovny ve složce `C:\Windows\System32`

3.1.1.2 Formáty souborů

Získané objekty jsou jednoho či druhého formátu:

- JSON – preferovaný formát. U objektů získaných skrze REST požadavky lze vybrat formát (buď JSON nebo XML), vybírán je tedy JSON. U objektů, které jsou získávány pomocí Java SDK, a je to možné, jsou rovněž data ukládána do struktury JSON souboru.
- XML – v tomto formátu jsou uloženy pouze objekty typu Connection.

3.1.1.3 Získávání jednotlivých objektů

V tabulce 3.1 lze vidět, jaké způsoby API a formátu jsou pro jednotlivé typy objektů použity. Kromě těchto objektů známých z analýzy se do výsledné struktury souborů ukládá ještě dodatečná informace pro každý objekt typu Universe o tom, jaké objekty typu Connection používá – tato data jsou získána pomocí REST API a uložena v JSON formátu.

Tabulka 3.1: Přehled objektů a způsobu jejich získávání

Typ objektu	**API**	**Formát**
Document	REST	JSON
Report	REST	JSON
Report Element	REST	JSON
Document Variable	REST	JSON
Data Provider	REST	JSON
Universe	REST	JSON
Business Object	Java SDK	JSON
Data Foundation	Java SDK	JSON
Connection	Java SDK	XML

Z objektů typu Universe je SAP BO konektorem podporován pouze druh UNX – díky přívětivému Java SDK. To je dáno tím, že pro extrakci potřebných dat UNV univerza by bylo vyžadováno užití Component Object Model (COM) SDK [33] – což bylo vyhodnoceno jako zbytečná komplikace. Navíc převod UNV na UNX je díky aplikaci IDT velmi jednoduchý.

3.1.2 Integrace SDK extractoru

Jelikož je část extrakce využívající SDK knihovny velmi náročná na konfiguraci (speciální JRE apod.), bylo rozhodnuto, že se tato část bude moci spouštět samostatně a nebude povinnou součástí klasického extractoru - tuto část implementuje třída `SdkExtractor`. Pokud se bude scénář extrakce spouštět na systému, kde jsou nainstalované Client Tools, bude moci zákazník spustit celou extrakci najednou. Pokud ale na daném stroji nejsou nainstalované Client Tools, či zákazník nebude chtít spouštět SDK extrakci, dojde pouze k částečné extrakci, přičemž SDK extrakci je možné spustit na jiném stroji samostatně, a takto nově získané soubory dodat do výstupu původní extrakce.

Příklad šablony properties souboru:

```
# Enter a host name or IP address of the SAP BO server. For
↪  example DESKTOP-85T1AD0.
sapbo.extractor.host=

# Enter a port for the SAP BO server. The default is 6400.
sapbo.extractor.port=6400

# Enter a port for REST requests for the SAP BO server. The
↪  default is 6405.
sapbo.extractor.port_rest=6405

# Enter a user name for a connection to the SAP BO server.
sapbo.extractor.user=

# Enter a password for a connection to the SAP BO server.
sapbo.extractor.password=

# Enter an auth method for a connection to the SAP BO server.
↪  The default is secEnterprise.
sapbo.extractor.auth=secEnterprise

# Enter an encoding of extracted SAP BO projects. Only needed,
↪  if input has been served manually by the client.
sapbo.input.encoding=UTF-8
```

3.2 Object Mapper

Tento maven modul vznikl nejprve jako součást modulu Extractor a slouží k mapování objektů na adresářovou cestu. Proč je toto potřeba? Během extrakce dochází ke stahování mnoha souborů - každý z objektů zmíněných v analýze má svůj. Nicméně obsah těchto souborů postrádá kontext, hierarchickou strukturu – např. který Report patří pod který Document apod. Proto byla vytvořena následující adresářová struktura, aby se žádná informace neztratila. Tato adresářová struktura byla inspirována URL REST požadavků a její podoba je zachycena na diagramu 3.2.

Proč byla tato funkcionalita (mapování objektů na adresářovou cestu) vyextrahována do vlastního modulu? Když jsem začínal pracovat na Resolveru, jehož úkol je načíst vyextrahované objekty, bylo mi řečeno, že ne každý zákazník chce u sebe pouštět naše Extractory (které mohou představovat určité bezpečnostní riziko) a že takoví zákazníci pak dodají požadované soubory manuálně. Tudíž Resolver není přímo závislý na výstupu Extractoru, ale na struktuře takového výstupu. Zároveň je na této struktuře závislý i Extrac-

tor. Který modul by měl tuto informaci obsahovat? Jelikož tyto dva moduly nic jiného nesdílí a závislost jednoho na druhém by nedávala konceptuálně smysl, bylo rozhodnuto, že informace o struktuře extrahovaných souborů bude vyčleněna do samostatného modulu. Na tomto nově vytvořeném modulu jsou závislé moduly Extractor a Resolver.

3.2.1 Rozhraní

Rozhraní modulu, respektive v tomto případě by bylo vhodnější použít obecnější výraz veřejný kontrakt, tvoří třída `ModelToFullPathTransformer`, konkrétně její statické metody s rozhraním:

```
void setRootDir(String pathToRootDir)
String fullPathToFile(SapObject sapObject, String... relevantIds)
```

3.3 Resolver

Modul Resolver slouží k načtení vyextrahovaných souborů, vyfiltrování užitečných informací, a na základě nich vytvořit strukturu javovských objektů.

Zjednodušené rozhraní resolveru:

- Vstup – složka s vyextrahované soubory.
- Výstup – struktura javovských objektů.

3.3.1 Reader

Reader, vlastním názvem *SapboInputReader*, je třídou, která je vstupním bodem Resolveru. Její hlavní zodpovědností je metoda *readFile* s rozhraním `SapboServer readFile(File file)`. Tato metoda je spouštěna pro každý dříve extrahovaný soubor, který prošel určitými filtry. U těchto souborů pak zkontroluje, že se jedná o soubory objektů typu Document. Pro každý Document pak spustí resolving všech jeho částí (všech Reportů, Data Providerů...), a Univerz, na kterých daný Document závisí (a všech objektů s nimi souvisejícími). Jelikož ale může jedno Univerzum sloužit jako zdroj dat pro mnoho různých Documentů, zodpovědností Readeru je i udržování cache těchto již jednou načtených Univerz, aby se nemusela načítat vícekrát. Výsledkem je instance rozhraní `SapboServer`, která obsahuje všechna potřebná data pro vytvoření datových toků od Documentů až po databáze. Tento výstup pak slouží jako vstup pro modul Generator.

3.4 Model

Tento maven modul slouží jako rozhraní, skrze které se přistupuje k objektům vytvořených v modulu Resolver. Je implementován jako soubor ob-

jektů typu `Interface` v Javě, poskytovány jsou pouze metody pro čtení, nikoli pro úpravu vzniklých objektů. Tedy jakmile jsou objekty v modulu Resolver vytvořeny, zachází se s nimi jako s hotovými produkty s neměnnou podstatou.

3.5 Dataflow Generator

Modul Generator slouží k vytvoření grafu, kterému bude Manta rozumět, ze struktury vytvořených javovských objektů. Takový graf obsahuje vrcholy a orientované hrany mezi nimi – tyto hrany ztělesňují toky dat a vedou od zdroje k cíli (v případě SAP BO od databází, přes všechny prostředníky, až ke konkrétním položkám Report Elementů).

Zjednodušené rozhraní Generatoru:

- Vstup – struktura javovských objektů.
- Výstup – graf uzlů s toky.

Za vstupní bod Generatoru se dá považovat třída `ServerAnalyzer` s metodou `Node analyze(SapboServer inputServer, SapboGraphHelper gh)`. Tato metoda je volána ze třídy `SapboDataflowTask`, která zajišťuje správnou integraci s Mantou. Metoda vrací objekt typu `Node`, což odpovídá kořenu grafu vytvořenému pro danou instanci `SapboServer`, která byla dříve vytvořena v Readeru 3.3.1. Instance třídy `SapboGraphHelper` slouží jako pomocný mechanismus pro vytváření grafu. Třída `ServerAnalyzer` ve své metodě `analyze` deleguje práci na další třídy typu `Analyzer` (*DocumentAnalyzer*, *UniverseAnalyzer* apod.).

3.6 Integrace s Mantou

Poté, co došlo k implementaci samotných modulů, bylo nutné zařídit jejich propojení – integraci s Mantou a orchestraci jejich činností.

3.6.1 Integrace s rozhraním Extractor

Aby byl modul Extractor spouštěn v rámci Manty, musely se upravit dva soubory v modulu CLI – *_run_extract.bat* a *_run_extract.sh*.

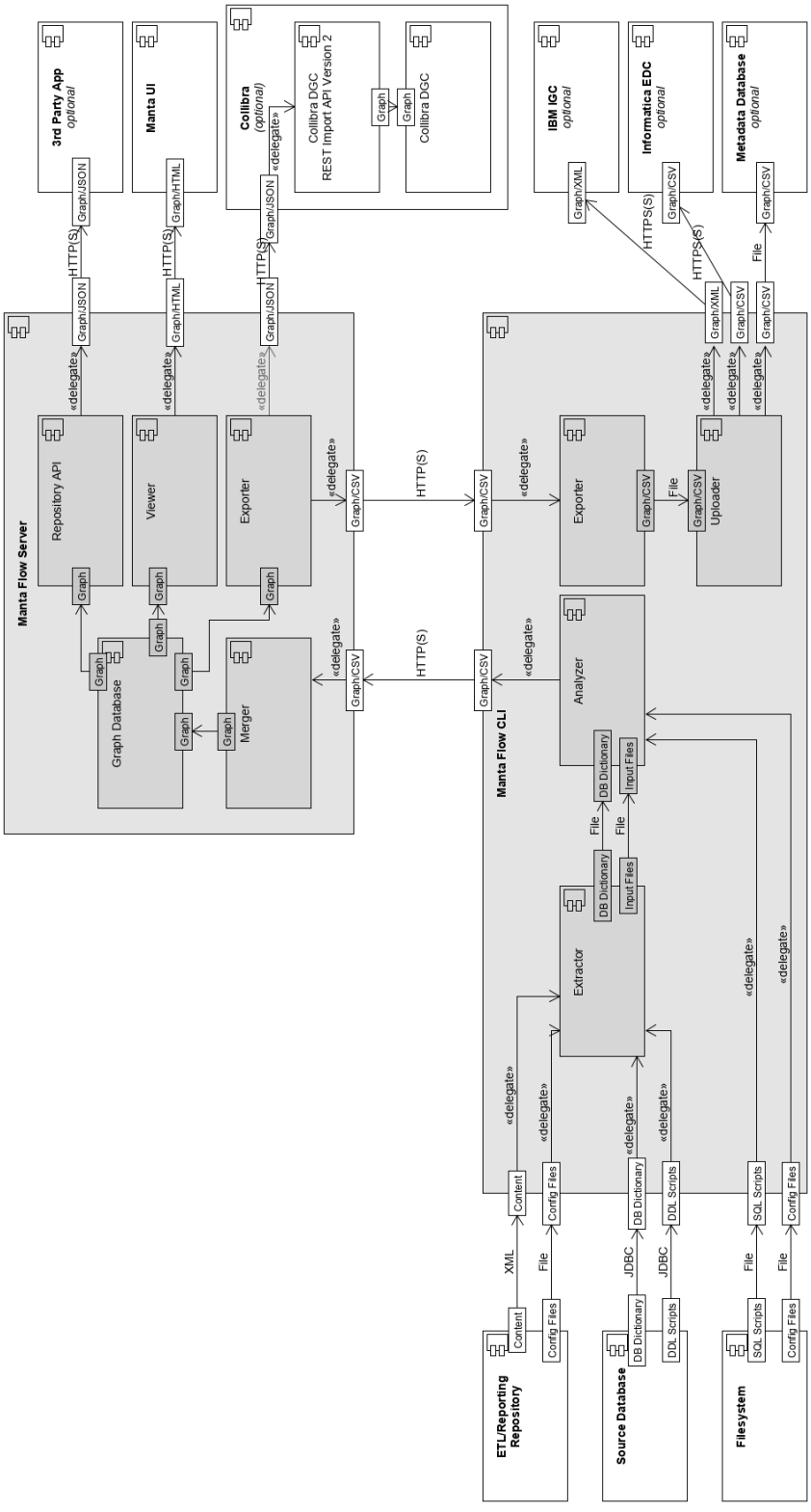
Jak přípona souboru napovídá, skript *_run_extract.bat* je spouštěn, pokud je Manta nainstalovaná na operačním systému rodiny Windows. Na obrázku 3.4 lze vidět sekvenční diagram extrakce. Skript postupně spustí oba typy extrakce – nejprve první typ, který je skrze objekty typu `ExtractorScenario` a `ExtractorTask` konfigurován s běhovým prostředím Manty, v této fázi dojde k extrakci objektů pomocí REST API. Až první typ extrakce doběhne, spustí se druhý typ – skrze pomocný skript se spustí metoda `main` třídy `SdkExtractorStarter`, která následně spustí `SdkExtractor`, který se skrze

JAVA API postará o extrakci zbylých objektů, které se v první fázi neextrahovaly. Tato druhá část se spustí pouze tehdy, pokud uživatel vyplní v konfiguračním souboru cestu ke složce, kde jsou nainstalované Client Tools.

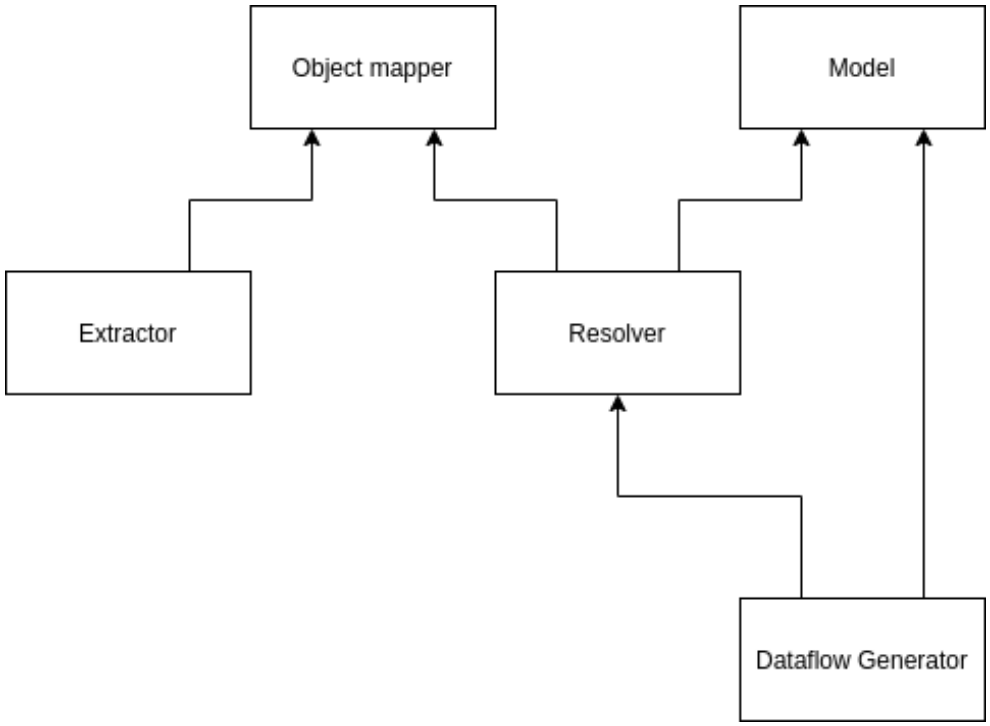
Skript *_run_extract.sh* je spouštěn pod linuxovým prostředím. V rámci tohoto skriptu dochází k extrakci pouze takových objektů, které lze získat skrze REST API, jelikož pro spouštění SDK Extractoru je nutné mít nainstalované Client Tools, které nelze mít na operačním systému s linuxovým jádrem.

3.6.2 Integrace s rozhraním Analyzer

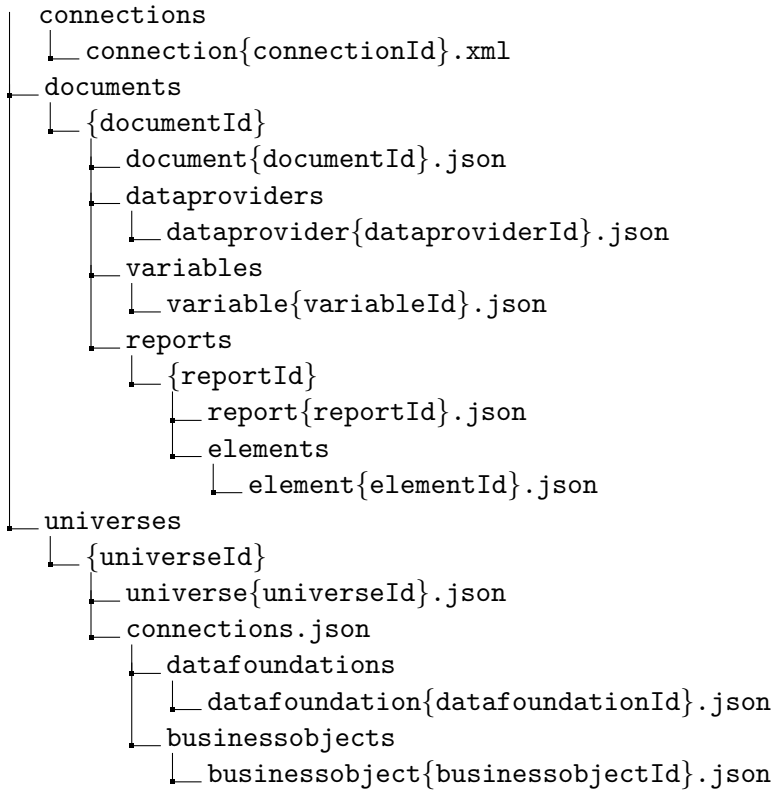
Na obrázku 3.5 lze vidět proces analýzy objektů. Analýza je spouštěna v rámci scénáře `SapboScenario` v cyklu – do té doby, dokud má `SapboInputReader` co číst. Pro každý přečtený `Document` se vytvoří použitím modulu Resolver instance `SapboServer` (z modulu Model), která pod sebou obsahuje všechny objekty potřebné pro následnou analýzu. Tato vytvořená instance se pak použije jako vstup pro `SapboDataflowTask`, který následně předá práci třídě `ServerAnalyzer`, která pro předanou instanci `SapboServer` vytvoří graf, který se ve finální fázi pošle na server.



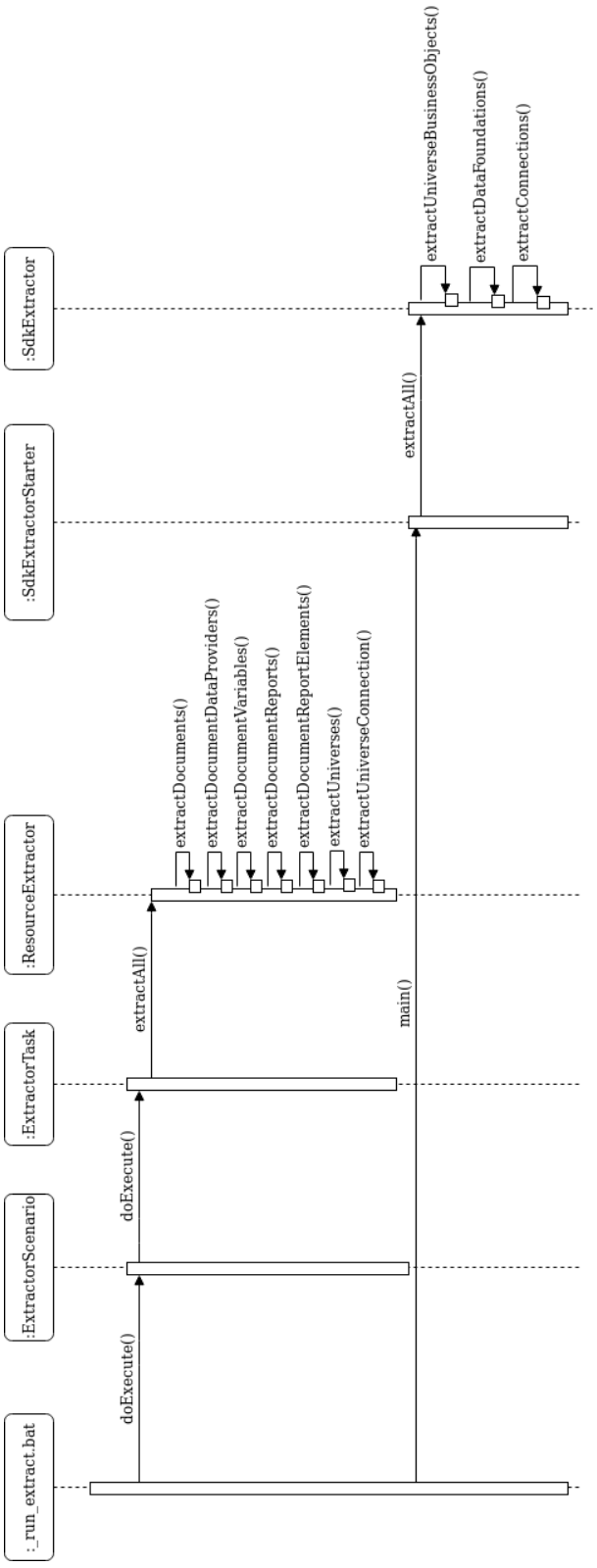
Obrázek 3.1: Architektura Manty. Převzato z [2]



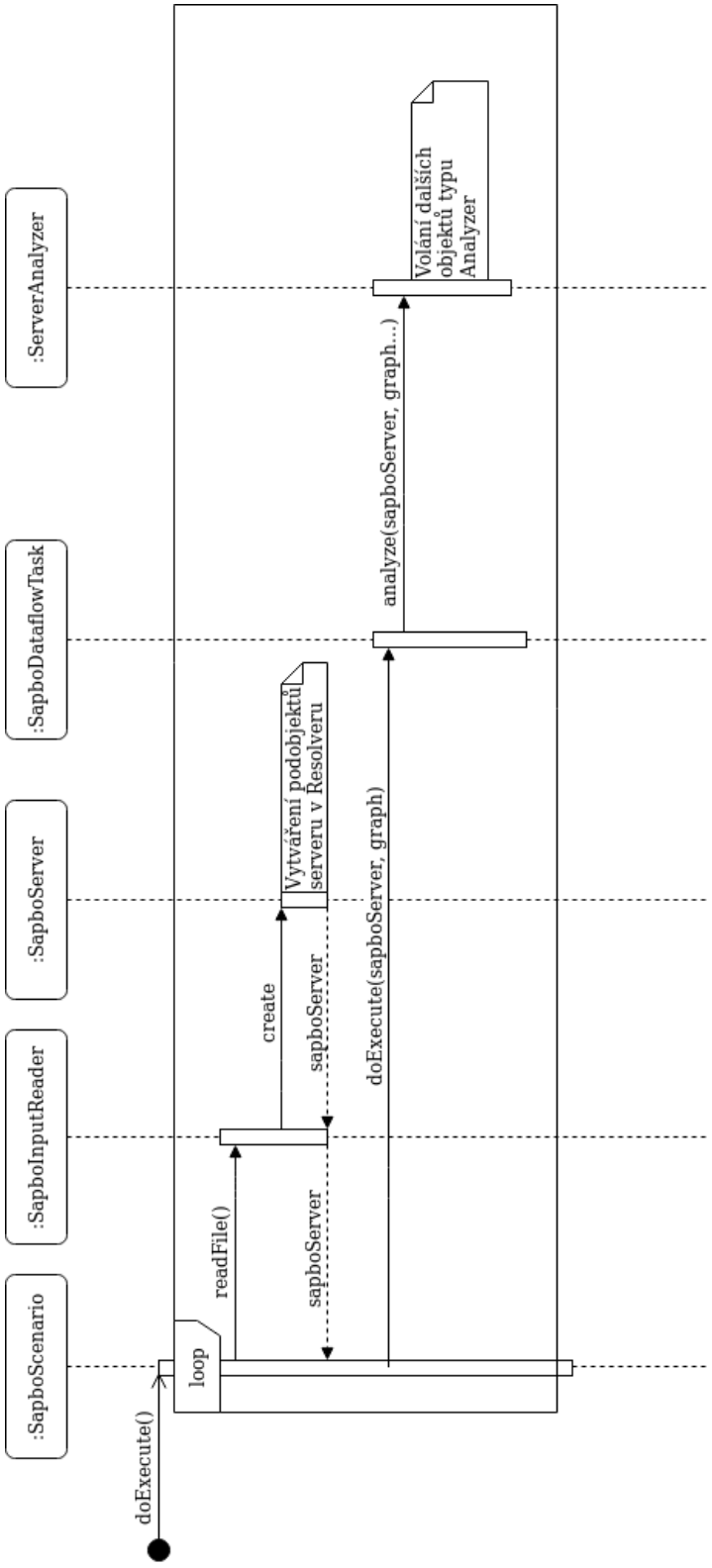
Obrázek 3.2: Diagram modulů a jejich závislostí vyjádřených šipkami. Šipky v diagramu směřují pouze směrem vzhůru, jedná se tedy o acyklický graf, s cyklickým grafem závislostí by měl maven potíže [3].



Obrázek 3.3: Adresářová struktura výstupu extractoru.



Obrázek 3.4: Sekvenční diagram extrakce v případě *_run_extract.bat* skriptu.



Obrázek 3.5: Sekvenční diagram pro integraci s rozhraním Analyzer.

Kapitola 4

Implementace

V této kapitole se budu věnovat vybraným implementačním detailům navržených modulů - Extractoru, Object Mapperu, Resolveru, Modelu a Dataflow Generatoru. Poslední sekce je věnována dosaženému výsledku.

4.1 Extractor

Rozhraní extractoru je implementováno třídou `ResourceExtractor` s metodou `extractAll()`. Veškeré potřebné parametry jsou předány instanci třídy `ResourceExtractor` v konstruktoru:

```
ResourceExtractor(ServerConnection serverConnection, String
↪ rootFolder)
```

- serverConnection – instance třídy ServerConnection, která obsahuje všechny potřebné informace pro navázání spojení se SAP BO serverem.
- rootFolder – cesta k výstupní složce.

4.2 Object Mapper

Používání modulu Object Mapper se dá demonstrovat na úryvku testu:

```
String documentId = "123";
ModelToFullPathTransformer.setRootDir(".");
String fullPathToDocument =
↪ ModelToFullPathTransformer.fullPathToFile(SapObject.DOCUMENT,
↪ documentId);
assertEquals("./documents/123/document123.json",
↪ fullPathToDocument);
```

Třída `ModelToFullPathTransformer` se chová jako návrhový vzor singleton, nedá se instanciovat, poskytuje pouze statické metody a má svůj statický stav (`rootDir`). Při jejím prvním použití je potřeba nastavit kontext metodou `setRootDir`. A pak již lze opakovaně volat metodu `fullPathToFile` s různými parametry.

Třídu by bylo možné implementovat i bezstavově, tedy že by součástí metody `fullPathToFile` byla informace o kontextu (`rootDir`) jako další parametr. Tuto možnost jsem zvážil a zamítl, jelikož tato metoda je volána z mnoha různých míst v kódu, a ne všechny objekty volající tuto metodu musí mít přístup k požadovanému kontextu.

Rovněž lze třídu implementovat i nestaticky – stav by zůstal zachován, ale instanciaci a propojení se závislými objekty by měl na starost framework Spring. Tato možnost byla rovněž zamítnuta, hlavně kvůli pracnosti, která by benefitů moc nepřinesla. Pokud by však někdy došlo k situaci, kdy by statičnost singletonu mohla vadit (mohl by vyvstat např. požadavek na dědění od jiné třídy, či nutnost serializace), je toto cestou, kterou se vydat. Jelikož jsem ale shledal v tomto kontextu statičnost velmi užitečnou, která nepřidává mnoho kódu navíc, a navíc myslitelné požadavky, které by mohly vést k nutnosti tuto skutečnost upravit, jsem odhadl na téměř nulové, nechal jsem ve finální verzi konektoru toto statické řešení.

4.3 Resolver

V rámci implementace tohoto modulu se nejprve musela provést analýza extrahovaných souborů a jejich formátů. Tato analýza nebyla vůbec jednoduchá, jelikož dokumentace k extrahovaným objektům byla velmi strohá, obsahující pouze pár příkladů dokumentů. Od té doby byla dokumentace mírně rozšířená o více příkladů, např. na stránce s dokumentací k Report Elementům [34] přibylo mnoho příkladů, především různých typů elementů, včetně Custom Elementů, které jsem v době provádění analýzy (léto 2019) vůbec neměl k dispozici. Aktuální dokumentace má však stále velké nedostatky – předně příklady jsou pouze v XML formátu, ne v JSONu, který moje aplikace využívá. Dále není vůbec jasné, jak se do sebe jednotlivá pole mohou zanořovat, a která pole jsou v souboru vždy, a která jsou v souboru pouze někdy. A v případě Report Elementů dokumentace stále neobsahuje informaci, jak přesně se mohou jednotlivé typy elementů zanořovat.

Na základě vyextrahovaných dokumentů jsem tak musel induktivně odhadnout obecnou strukturu jednotlivých formátů. V případě odhadnutí volitelnosti jednotlivých položek v dokumentu to bylo jednoduché – pro každý typ dokumentu (tím jsou na mysli JSON formáty objektů typu Document, Document Variable, Data Provider, Report Element či Universe) došlo k součtu četností jednotlivých položek napříč různými instancemi a k rozdělení na dvě kategorie:

- Povinné položky – pokud byly přítomny ve všech dokumentech daného typu, pak jsou pokládány za povinné. Mezi takové položky patří např. *id* či *name* (ve většině případech).
- Volitelné položky – pokud v některém z dokumentů daného typu dotyčná položka chyběla, je považována za volitelnou. Příklady takových položek mohou být např. pole *hide* u Report Elementu (které může obsahovat formuli, podle které dojde ke zviditelnění / zneviditelnění elementu), či pole *name* taktéž u Report Elementu.

4.3.1 Document

Příklad extrahovaného objektu typu *Document*:

```
{
  "document": {
    "createdBy": "Administrator",
    "cuid": "ATeBlMbXn.xCuSaEElUEGI0",
    "folderId": 5683,
    "id": 5684,
    "lastAuthor": "Administrator",
    "name": "MonitoringTrend Data-Sample",
    "path": "Public Folders/Monitoring Report Sample",
    "refreshOnOpen": "false",
    "scheduled": "false",
    "size": 41777,
    "state": "Unused",
    "updated": "2019-03-14T21:42:10.832Z"
  }
}
```

Na příkladu lze vidět JSON formát, se kterým se v případě objektu typu Document zachází. Ze všech položek jich nás zajímá pouze několik – konkrétně *name*, *id* a *path*. Situace, že se z extrahovaných dat využívá pouze jejich zlomek, je celkem běžná i u ostatních objektů. Zároveň stojí za povšimnutí, že o žádném ze závislých objektů na tomto Documentu (např. Report) v souboru není zmínka.

4.3.2 Report

Zde je příklad jednoho Reportu pod zmíněným Documentem:

```
{
  "report": {
    "@hasDatafilter": "false",
    "@hasDriller": "false",
```

```
    "id": 1,
    "name": "Watch and Metrics",
    "pageSettings": {
      ...
    },
    "paginationMode": "QuickDisplay",
    "reference": "1.RS",
    "section": [],
    "showDataChanges": "false",
    "showFolding": "false",
    "style": {
      "hyperLinkColors": {
        "@active": "#000000",
        "@hover": "#000000",
        "@link": "#0000ff",
        "@visited": "#551a8b"
      }
    }
  }
}
```

Ani v Reportu není žádná zmínka o jeho dětech (Report Elementech), dokonce ani o jeho rodiči. To je dáno tím, že informace o vztazích rodič-dítě je uložená mimo samotné soubory – je totiž uložená v adresářové struktuře, kterou bylo potřeba během extrakce vytvořit, jinak by se tato informace ztratila.

4.3.3 Report Element

Situace se ale mění u Report Elementů příslušejících k danému Reportu. Takové Report Elementy jsou všechny extrahované v jedné složce, na první pohled však není jasné, jak jsou hierarchicky uspořádané. Tato informace je však uložena v každém Report Elementu (s výjimkou PageZone, což jsou Elementy ve vrchní vrstvě rodičovské hierarchie) pod položkou *parentId*, což je Id rodiče.

Na příkladu lze vidět zkrácený soubor Elementu typu Cell. Jeho rodič má Id 12.

```
{
  "element": {
    "@isLinkedToSharedElement": "false",
    "@type": "Cell",
    "content": {
      "expression": {
        "formula": {
          "$": "=NameOf([Name])",
          "@dataType": "String",
```

```
          "@qualification": "Measure",
          "@type": "Text"
        }
      }
    },
    "hide": {
      "@always": "false"
    },
    "id": 10,
    "padding": {
      "@bottom": 350,
      "@left": 250,
      "@right": 250,
      "@top": 300
    },
    "parentId": 12,
    ...
  }
}
```

V Report Elementech pak bylo potřeba vyhledat tu nejdůležitější informaci z hlediska toků – položky s formulemi. Na příkladu lze vidět řádek `"$": "=NameOf([Name])",` – přesně toho je ukázka položky s formulí. Bylo zjištěno, že veškeré položky obsahující formuli mají vždy klíč `"$"`. Takových položek může být v různých typech Elementů různý počet a veškerá implementace s tím musí počítat. V Resolveru tak byl úkol tyto formule v souborech vyhledat a uložit do objektů, analýza těchto formulí se provádí až v modulu Generator.

4.3.4 Data Provider

Objekty typu Data Provider obsahují všechna data o zdrojích, ze kterých se v daném Documentu čerpá.

Příklad jednoho extrahovaného Data Providera:

```
{
  "dataprovider": {
    "@refreshable": "true",
    "dataSourceCuid": "AamfjLdF3Y1IoWCqLq7LobA",
    "dataSourceId": "5628",
    "dataSourceType": "unx",
    "dictionary": {
      "expression": [
        {
          "@dataType": "String",
```

```
          "@qualification": "Dimension",
          "dataSourceObjectId": "_U1gFwHThEeadwP7kSqONqw",
          "formulaLanguageId": "[Athlete]",
          "id": "DP18.DO7",
          "name": "Athlete"
        },
        ...
        {
          "@dataType": "Numeric",
          "@qualification": "Dimension",
          "dataSourceObjectId": "_U16VcHThEeadwP7kSqONqw",
          "formulaLanguageId": "[Year]",
          "id": "DP18.DOb",
          "name": "Year"
        }
      ]
    },
    "duration": 1,
    "flowCount": 1,
    "id": "DP18",
    "isPartial": "false",
    "name": "Result",
    "query": "SELECT  Table__1.\"Code Country\",
    ↪  Table__1.\"Country\",  Table__1.\"Medal\",
    ↪  Table__1.\"Sport\",  Table__1.\"Event\",
    ↪  Table__1.\"Gender\",  Table__1.\"Name\",  Table__1.\"Web
    ↪  Link\",  Table__1.\"Olympic Games\",  Table__1.\"Ctry
    ↪  Organizer\",  Table__1.\"Year\",  Table__1.\"Saison\"
    ↪  FROM  \"JO\$\" Table__1 WHERE  (  Table__1.\"Olympic
    ↪  Games\" = 'Rio de Janero'  AND  Table__1.\"Ctry
    ↪  Organizer\" = 'Brazil'  )",
    "rowCount": 974,
    "updated": "2017-03-07T16:31:39.000Z"
  }
}
```


Zajímavé položky jsou následující:

- dataSourceType – typ zdroje dat, v konektoru je podporovaný pouze typ objektu Universe *unx*.
- dataSourceId – id zdroje dat, v tomto případě Univerza.
- dictionary-expression – definice jednotlivých objektů. Z hlediska toků dat jsou zajímavé především dvě položky. První je *formulaLanguageId*,

což je identifikátor objektu v rámci Documentu a je využívaný ve výrazech napříč Documentem. Druhou zajímavou položkou je *dataSourceObjectId*, což je identifikátor Business Objectu v Univerzu, ze kterého se čerpají data.

- name – název Data Providera.
- query – pomocná položka, která obsahuje SQL script pro získání všech dat ze všech obsažených položek dictionary-expression. Z pohledu datových toků se ale jedná o zbytečnou informaci, neboť úkolem je propojit toky jednotlivých objektů na nejnižších detailech, tedy jednotlivé objekty dictionary-expression.

4.3.5 Document Variable

Všechny objekty druhu Document Variable mají podobnou strukturu jako následující příklad:

```
{
  "variable": {
    "@dataType": "Numeric",
    "@qualification": "Measure",
    "definition": "=Count ([Event]; All)",
    "formulaLanguageId": "[Nb Event]",
    "id": "L41A",
    "name": "Nb Event"
  }
}
```

Nejzajímavějšími položkami z pohledu datových toků jsou *definition* – formule, podle které se vyhodnocuje obsah proměnné, a *formulaLanguageId* – unikátní identifikátor napříč zdroji dat v podobjektech typu Document.

4.3.6 Universe

Objekty typu Universe mají v extrahované podobě velmi podobnou strukturu, která se dobře vysvětlí na příkladu:

```
{
  "universe": {
    "connected": "true",
    "cuid": "AX_fkdjuubJFrhAVg7nkWFk",
    "description": "eFashion retail Data Warehouse dated 14 Oct
    ↪  2007. 89,000+ row fact table...",
    "folderId": 5622,
    "id": 9154,
```

```
    "maxRetrievalTime": 300,
    "maxRowsRetrieved": 90000,
    "name": "eFashion.unx",
    "outline": {
      "@aggregated": "false",
      "folder": [
        {
          "customProperty": [],
          "description": "Time hierarchy",
          "id": "CLS_18",
          "item": [
            {
              "@dataType": "Numeric",
              "@hasLov": "true",
              "@type": "Measure",
              "aggregationFunction": "Sum",
              "customProperty": [],
              "description": "Margin \$ = Revenue - Cost of
              ↪  sales",
              "id": "OBJ_146",
              "name": "Margin",
              "path": "Measures|folder\\Margin|measure"
            },
            ...
          ],
          "name": "Measures"
        }
      ]
    },
    "path": "Universes/Samples",
    "type": "unx"
  }
}
```

Z hlediska vizualizace datových toků jsou zajímavé položky následující:

- id – identifikátor Universa.
- name – lidsky čitelné jméno Universa.
- path – adresářová cesta k objektu.
- type – typ, buď *unx* (novější, konektorem podporovaný formát), nebo *unv* (starší, konektorem nepodporovaný formát).

Navíc pod položkou *outline* obsahuje základní informace o jednotlivých Business Objectech, uložených v pomocných složkách (položka *folder*). Pro Resolver jsou zajímavé především položky:

- @type – typ Business Objectu (Dimension, Measure, Attribute nebo Filter).
- id – identifikátor objektu, který se používá např. i v definici v rámci Data Provideru.
- name – lidsky čitelný identifikátor, v rámci adresářové struktury unikátní.
- path – cesta k objektu, v rámci Universa se jedná o unikátní identifikátor.

Takto uložené Business Objecty ale neobsahují všechny informace potřebné ke konstrukci toků až k databázím. Na to bylo potřeba využít informace speciálně extrahovaných pomocí SDK.

4.3.7 Business Object

Výstup SDK extractoru pro Business Object může vypadat takto:

```
{
  "extra-tables": [],
  "id": "OBJ_146",
  "path": "Measures|folder\\Margin|measure",
  "select": "@Aggregate_Aware(
  ↪  sum(Agg_yr_qt_mt_mn_wk_rg_cy_sn_sr_qt_ma.Margin),
  ↪  sum(Shop_facts.Margin))",
  "where": ""
}
```

Jednotlivé položky:

- extra-tables – nezjištěna přesná funkcionalita, ve všech ukázkových Business Objectech je prázdná.
- id – identifikátor objektu, odpovídá položce *id* dostupné z Universa.
- path – cesta k objektu, odpovídá položce *path* dostupné z Universa.
- select – select fragment z SQL dotazu, který definuje data objektu.
- where – where fragment z SQL dotazu, který definuje data objektu.

4.3.8 Data Foundation

Objekty tohoto druhu jsou získávané také pomocí SDK a následně uloženy jako JSON dokument. Příklad takového dokumentu:

```
{
  "joins": [
    {
      "expression": "Outlet_Lookup.Shop_id=Shop_facts.Shop_id",
      "left table": "Outlet_Lookup",
      "right table": "Shop_facts"
    },
    ...
  ],
  "tables": [
    {
      "columns": [
        "Color_code",
        "Article_id",
        "Amount_sold",
        "Shop_id",
        "Shop_facts_id",
        "Week_id",
        "Margin",
        "Quantity_sold"
      ],
      "name": "Shop_facts"
    },
    ...
  ]
}
```

Každý objekt typu Data Foundation obsahuje položku *tables* - seznam tabulek s jejich jmény a jmény sloupců, a položku *joins*, která obsahuje seznam joinů mezi tabulkami, přičemž každý join obsahuje tři informace – dva názvy tabulek, které join propojuje, a propojovací výraz.

4.3.9 Connection

Objekty typu Connection obsahují informace o připojeních k dříve definovaným databázím. Pomocí SDK můžeme získat např. tento soubor:

```
XML=<?xml version\="1.0" standalone\="yes"?>

<ConnectionDefinition>
```

```
    <Parameter ID\="NETWORKLAYER" Category\="IDENTIFICATION"
    ↪  Type\="String">ODBC</Parameter>
    <Parameter ID\="ARRAY_FETCH_SIZE" Category\="CONFIGURATION"
    ↪  Type\="Integer">10</Parameter>
    <Parameter ID\="AUTHENTICATION_MODE" Category\="CREDENTIALS"
    ↪  Type\="Enum">ConfiguredIdentity</Parameter>
    <Parameter ID\="USER" Category\="CREDENTIALS"
    ↪  Type\="String"></Parameter>
    <Parameter ID\="POOL_TIME" Category\="CONFIGURATION"
    ↪  Type\="Integer">10</Parameter>
    <Parameter ID\="LOG_TIMEOUT" Category\="CONFIGURATION"
    ↪  Type\="Integer">600</Parameter>
    <Parameter ID\="CONNECTIVITY_TYPE"
    ↪  Category\="IDENTIFICATION"
    ↪  Type\="String">Relational</Parameter>
    <Parameter ID\="PASSWORD" Category\="CREDENTIALS"
    ↪  Type\="String"></Parameter>
    <Parameter ID\="MAX_PARALLEL_QUERIES"
    ↪  Category\="CONFIGURATION" Type\="String">4</Parameter>
    <Parameter ID\="CONNECT_INIT" Category\="CONFIGURATION"
    ↪  Type\="String"></Parameter>
    <Parameter ID\="DATASOURCE" Category\="CREDENTIALS"
    ↪  Type\="String">efashion</Parameter>
    <Parameter ID\="VERSION" Category\="IDENTIFICATION"
    ↪  Type\="String">7</Parameter>
    <Parameter ID\="DBMS" Category\="IDENTIFICATION"
    ↪  Type\="String">MS Access 2007</Parameter>
    <Parameter ID\="POOL_MODE" Category\="CONFIGURATION"
    ↪  Type\="Enum">Pooling</Parameter>
    <Parameter ID\="ARRAY_BIND_SIZE" Category\="CONFIGURATION"
    ↪  Type\="Integer">5</Parameter>
</ConnectionDefinition>

;USER=;PASSWORD=;
```

Pro další zpracování jsou zajímavé především položky *NETWORKLAYER* (nejčastěji ODBC nebo JDBC), *DATASOURCE* (identifikátor v případě ODBC), *DBMS* (typ databáze) a connection string (důležitý v případě JDBC, v tomto případě ODBC je prázdný, jak lze vidět na příkladu – *;USER=;PASSWORD=;*).

4.4 Model

Jelikož se předpokládá, že se k vytvořeným objektům přistupuje z Dataflow Generatoru pouze skrze toto rozhraní, je totéž rozhraní použito i přímo v testech Resolveru:

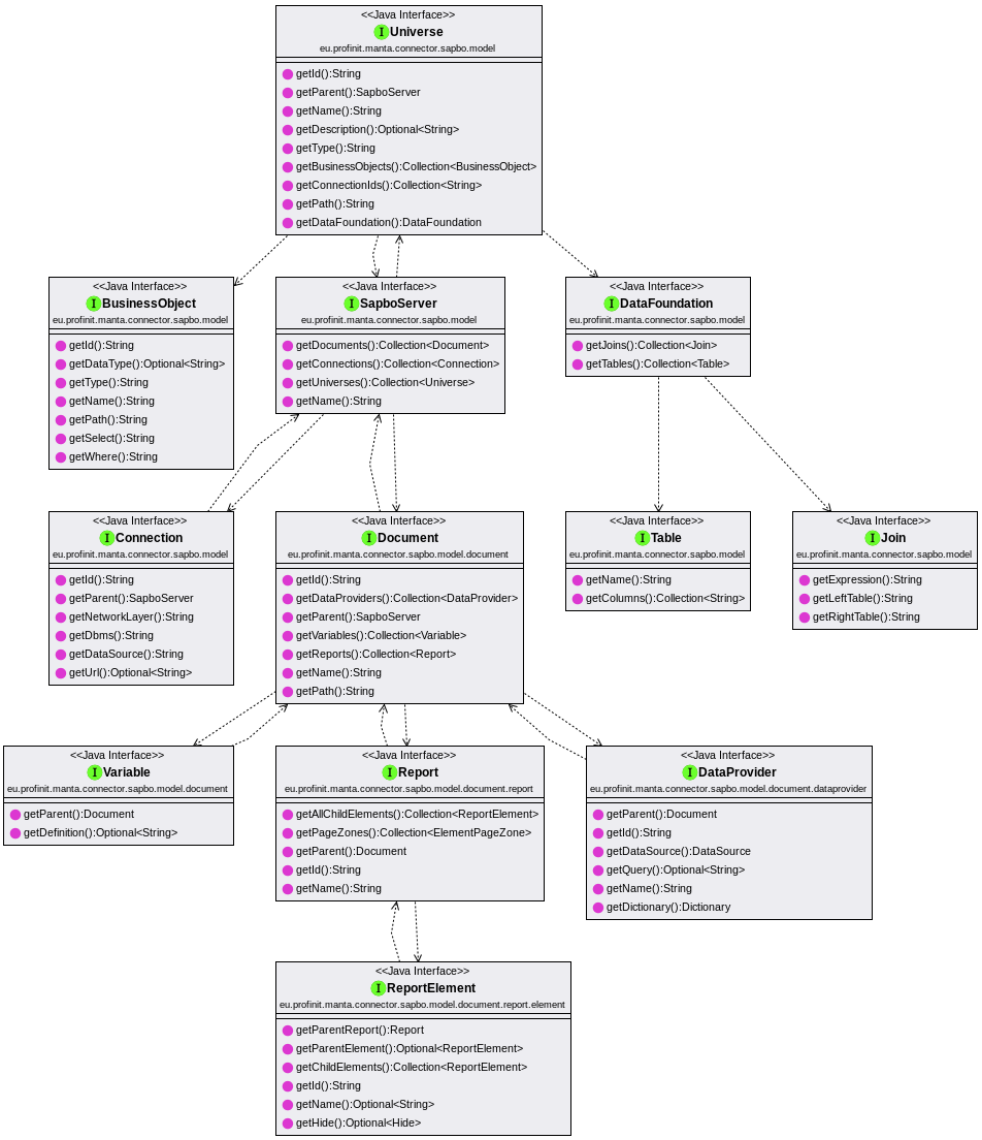
```
@Test
public void efashion() throws FileCouldNotBeReadException {
    String connectionId = "5630";

    Connection connection = resolveConnection(connectionId);

    assertEquals(connectionId, connection.getId());
    assertEquals("ODBC", connection.getNetworkLayer());
    assertEquals("MS Access 2007", connection.getDbms());
    assertEquals("efashion", connection.getDataSource());
}

private ConnectionImpl resolveConnection(String connectionId)
↪   throws FileCouldNotBeReadException {

    ↪   ModelToFullPathTransformer.setRootDir(TestParams.EXTRACTED_ROOT_FOLDER);
    String pathToConnection =
    ↪   ModelToFullPathTransformer.fullPathToFile(SapObject.CONNECTION,
    ↪   connectionId);
    return new
    ↪   ConnectionImpl(FileContentReader.readConnectionAsXmlElement(new
    ↪   File(pathToConnection)),
            connectionId, null);
}
```

Na ukázce kódu lze vidět, že pomocná metoda `resolveConnection` vrací instanci `ConnectionImpl`, která je ale následně testována jen skrze rozhraní `Connection`. `Connection` je rozhraní definované v modulu Model, zatímco `ConnectionImpl` je jeho konkrétní implementace v Resolveru.

Na obrázku 4.1 je class diagram hlavních tříd Modelu. Jako vstupní bod, přes který se k jednotlivým třídám přistupuje, je rozhraní `SapboServer`, které obsahuje (tranzitivně) referenci na všechny objekty Modelu.

4.5 Dataflow Generator

Vytvořit graf uzlů bez toků nebylo nic těžkého – stačilo vzít pečlivě vytvořenou hierarchii javovských objektů a tu víceméně překlopit do hierarchie uzlů. O něco obtížnější však bylo do takového grafu přidání hran, které



Obrázek 4.1: Hlavní třídy modulu Model.

jsou pro přehlednost zobrazené přerušovanými šipkami na obrázku 4.2. Aby bylo možné přidání kompletních toků, musely se analyzovat celkem dva typy výrazů.

4.5.1 Výrazy v (pod)objektech typu Document

Tyto výrazy jsou používané v celkem třech typech objektů - Report Element, Document Variable a Data Provider. Z hlediska uživatelů zacházejících s re-

porty jsou tou nejdůležitější informací, kterou Resolver z Report Elementů dostává a kterou musí Generator dále analyzovat a propojit s ostatními objekty.

Příklady výrazů:

```
[Sales revenue]
NameOf([Sales revenue])
[Margin]/[Sales revenue]
[Address]+" " +[City]+" "+[State]
([Sales revenue]-[NewYork2016SalesRevenue]) *100 / [NewYork2016SalesRevenue]
ReportName()
"Bar width : with Maximum value"
Max([Sales revenue]) ForAll([State];[Quarter])
If Count([Quarter])>1 Then "All Quarters" Else "Filtered for "+[Quarter]
NameOf([Sales Details].[State])
```

Všechny tyto výrazy jsou vzaty z různých položek vzorových Report Elementů.

4.5.1.1 formulaLanguageId

Jak lze vidět na ukázkách, formule mohou obsahovat mnoho různých výrazů - čísla, text, funkce a další složitější struktury. A také tzv. *formulaLanguageId*, které označují zdroj dat – těmito zdroji dat mohou být jak objekty v objektu Data Provider, ale také různé instance typu Document Variable. Výraz *formulaLanguageId* slouží k jednoznačné identifikaci těchto zdrojů, každý zdroj má přiřazen právě jeden tento identifikátor, a (logicky) více zdrojů nemůže sdílet jeden identifikátor.

Tyto identifikátory mají jasně danou strukturu – jedná se o řetězce znaků ohraničené hranatými závorkami, přičemž tyto výrazy se mohou řetězit znakem `.` (tečka). Z příkladů výše uvedených si tedy můžeme všimnout, že obsahují tyto identifikátory:

```
[Sales revenue]
[Margin]
[Address]
[City]
[State]
[NewYork2016SalesRevenue]
[Quarter]
[Sales Details].[State]
```

Ano, i `[Sales Details].[State]` funguje jako identifikátor pro jeden objekt, nikoliv dva. Oproti tomu výraz `[State];[Quarter]` obsahuje identifikátory dva, protože identifikátory nejesou spojeny tečkou, ale znakem `;` (středník).

4.5.1.2 Zpracování výrazů

Z časových důvodů bylo domluveno, že tyto výrazy není nutné parsovat celé, ale stačí z nich vytáhnout pouze to nejdůležitější – tedy použité identifikátory *formulaLanguageId*. V budoucnosti se však rozšíření na parsování celého výrazu nevylučuje, a proto došlo k implementaci základního parseru v nástroji ANTLR, který je snadno rozšiřitelný.

Z definice *formulaLanguageId* šlo vytvořit poměrně přímočaře pravidlo pro lexer:

```
fragment ESCAPED_BRACKET: '\\[' | '\\]';
fragment FORMULA_LANGUAGE_ID_CHAR: ~('[' | ']') |
↪  ESCAPED_BRACKET;
fragment FORMULA_LANGUAGE_ID_FRAGMENT: '['
↪  FORMULA_LANGUAGE_ID_CHAR+ ']';
fragment FORMULA_LANGUAGE_ID_FRAGMENT_WITH_DOT:
↪  FORMULA_LANGUAGE_ID_FRAGMENT '.';
FORMULA_LANGUAGE_ID: FORMULA_LANGUAGE_ID_FRAGMENT_WITH_DOT*
↪  FORMULA_LANGUAGE_ID_FRAGMENT;
```

Z ukázky implementace lze vidět, že bylo potřeba speciálně ošetřit situaci, kdy se hranaté závorky vyskytly v názvu *formulaLanguageId*, ale přitom nebyly odděleny tečkou.

Na základě výstupu z lexeru šel vytvořit jednoduchý parser, který z příchozích tokenů vybral jen to, co bylo potřeba – výrazy typu *formulaLanguageId*.

```
start: root | EOF;
root: (nonformulalanguageid | formulalanguageid)+;
nonformulalanguageid: ~(FORMULA_LANGUAGE_ID)!;
formulalanguageid: form = FORMULA_LANGUAGE_ID ->
↪  ^(AST_FORMULA_LANGUAGE_ID \$form);
```

Použití vygenerovaného parseru se dá demonstrovat na jednom z vytvořených testů (třída `ElMain` ztělesňuje parser, zatímco třída `ElLexer` ztělesňuje lexer):

```
@Test
public void complexExpression5() throws RecognitionException {
    CommonTree tree = parse("=Max ([Sales revenue]
    ↪  ForEach([Lines]))*1.15");
    Set<String> actualFormulas = extractFormulasFrom(tree);
    Set<String> expectedFormulas = new
    ↪  HashSet<>(Arrays.asList("[Sales revenue]", "[Lines]"));
    assertEquals(expectedFormulas, actualFormulas);
}
```

```
private CommonTree parse(String expression) throws
↪  RecognitionException {
    CharStream charStream = new ANTLRStringStream(expression);
    ElMain parser = new ElMain(new BufferedTokenStream(new
    ↪  ElLexer(charStream)));
    return (CommonTree) parser.start().getTree();
}
```

Díky pomocným metodám se použití parseru zjednoduší na následující rozhraní:

- Vstup – výraz jako String.
- Výstup – množina identifikátorů *formulaLanguageId*.

A s touto informací lze pak propojit jednotlivé zdroje (identifikovatelné pomocí *formulaLanguageId*) s místy jejich použití ve výrazech.

4.5.2 Výrazy v (pod)objektech typu Universe

Každý Business Object má svou definici jakožto SQL dotaz, ze kterého se dají následně zjistit objekty (sloupce) v databázi, na kterých je závislý. Parsování a zpracování takovéhoto SQL dotazu má na starost služba s názvem Dataflow Query Service, která je používaná napříč různými konektory v Mantě. Jelikož je tato služba v Mantě již implementovaná a používaná, stačilo pouze nakonfigurovat SAP BO konektor pro její využívání.

Aby však bylo možné ji využít, musel jsem se nejprve dostat k SQL definici každého Business Objektu. Grafické uživatelské rozhraní nástroje IDT (Information Designer Tools) z balíčku programů Client Tools umožňuje se k těmto definicím dostat (jak lze vidět na obrázku 4.3), nicméně SDK API tohoto nástroje tuto funkcionalitu neposkytuje [35].

Lze však využít SDK API k vyextrahování dvou fragmentů SQL dotazu – fragmentu „SELECT“ a „WHERE“. Pozorný čtenář si uvědomí, že pouze tyto dvě informace nestačí k přímočarému vytvoření plnohodnotného SQL dotazu – úplně totiž chybí klauzule „FROM“. Fragment dotazu „WHERE“ také není kompletní, neboť jeho součástí by měly být i JOINy mezi jednotlivými tabulkami databáze. Nutno dodat, že obvykle jsou JOINy v SQL dotazech součástí FROM klauzule, nicméně při zkoumání nástroje IDT a jeho mechanismu tvorby SQL dotazu jsem zjistil, že JOINy jsou součástí WHERE klazule, nikoliv FROM klazule (tedy FROM klauzule obsahují jen názvy tabulek – odpovídá to situaci, kdy mezi tabulkami dojde ke kartézskému součinu, a výsledek vznikne filtrací pomocí WHERE klauzulí definující JOINy).

Aby komplikací nebylo málo, součástí extrahovaných fragmentů SELECT a WHERE mohou být i speciální funkce používáné pouze v nástroji IDT, které se při překladu na SQL dotaz náležitě interpretují. Jedná se o několik funkcí, jedinou podporovanou funkcí mým konektorem je `@Aggregate_Aware`.

Jelikož se tedy nejedná o fragmenty čistého SQL jazyka, ale mohou být jejich součástí i funkce interpretované IDT, a bylo s nimi tedy potřeba speciálně zacházet, byly tyto fragmenty pracovně pojmenovány jako „connection expressions“. Pro jejich zpracování byla vytvořena třída `SqlCreator` se statickou metodou s rozhraním:

```
Set<String> createQueriesFor(BusinessObject businessObject,
↪  Collection<Join> joins)
```

- Vstup – Business Object, pro který chceme získat SQL dotazy, a související JOINy (z definice Data Foundation).
- Výstup – množina SQL dotazů.

Proč je výstupem množina dotazů a ne právě jeden dotaz? Reálně vždy dojde k překladu na jeden konkrétní dotaz, nicméně v případě použití funkce `@Aggregate_Aware` je na výběr z několika různých dotazů, přičemž s daty, které jsou k dispozici, metoda není schopná určit, který konkrétní dotaz se v daném okamžiku vybere. Z tohoto důvodu se zpracovávají všechny definice, přičemž výběr konkrétního dotazu by teoreticky měl být možný, ale je záležitostí budoucího vývoje.

Na příkladu 4.3 lze vidět Business Object s názvem *Discount*, jehož SELECT fragment tvoří kód

```
@Aggregate_Aware( sum(Shop_facts.Quantity_sold *
↪  Article_lookup.Sale_price - Shop_facts.Amount_sold),
↪  sum(Shop_facts.Quantity_sold *
↪  Article_Color_Lookup.Sale_price - Shop_facts.Amount_sold))
```

a WHERE fragment tvoří následující kód.

```
Shop_facts.Week_id=Calendar_year_lookup.Week_id
```

Reálně se z funkce `@Aggregate_Aware` (konkrétně z jeho argumentů SELECT fragmentu) vybere právě jeden argument, který se uplatní. Jelikož ale zatím není možné určit, který konkrétní argument se vybere, vytvoří se SQL dotazy pro oba:

```
SELECT
 sum(Shop_facts.Quantity_sold * Article_lookup.Sale_price -
 ↪  Shop_facts.Amount_sold)
FROM
 Calendar_year_lookup,
 Shop_facts,
 Article_lookup
WHERE (
 Article_lookup.Article_id=Shop_facts.Article_id)
```

```
 AND (Shop_facts.Week_id=Calendar_year_lookup.Week_id)

SELECT
 sum(Shop_facts.Quantity_sold * Article_Color_Lookup.Sale_price
 ↪  - Shop_facts.Amount_sold)
FROM
 Article_Color_Lookup,
 Calendar_year_lookup,
 Shop_facts
WHERE
 (Shop_facts.Week_id=Calendar_year_lookup.Week_id)
 AND (Article_Color_Lookup.Article_id=Shop_facts.Article_id and
 ↪  Article_Color_Lookup.Color_code=Shop_facts.Color_code)
```

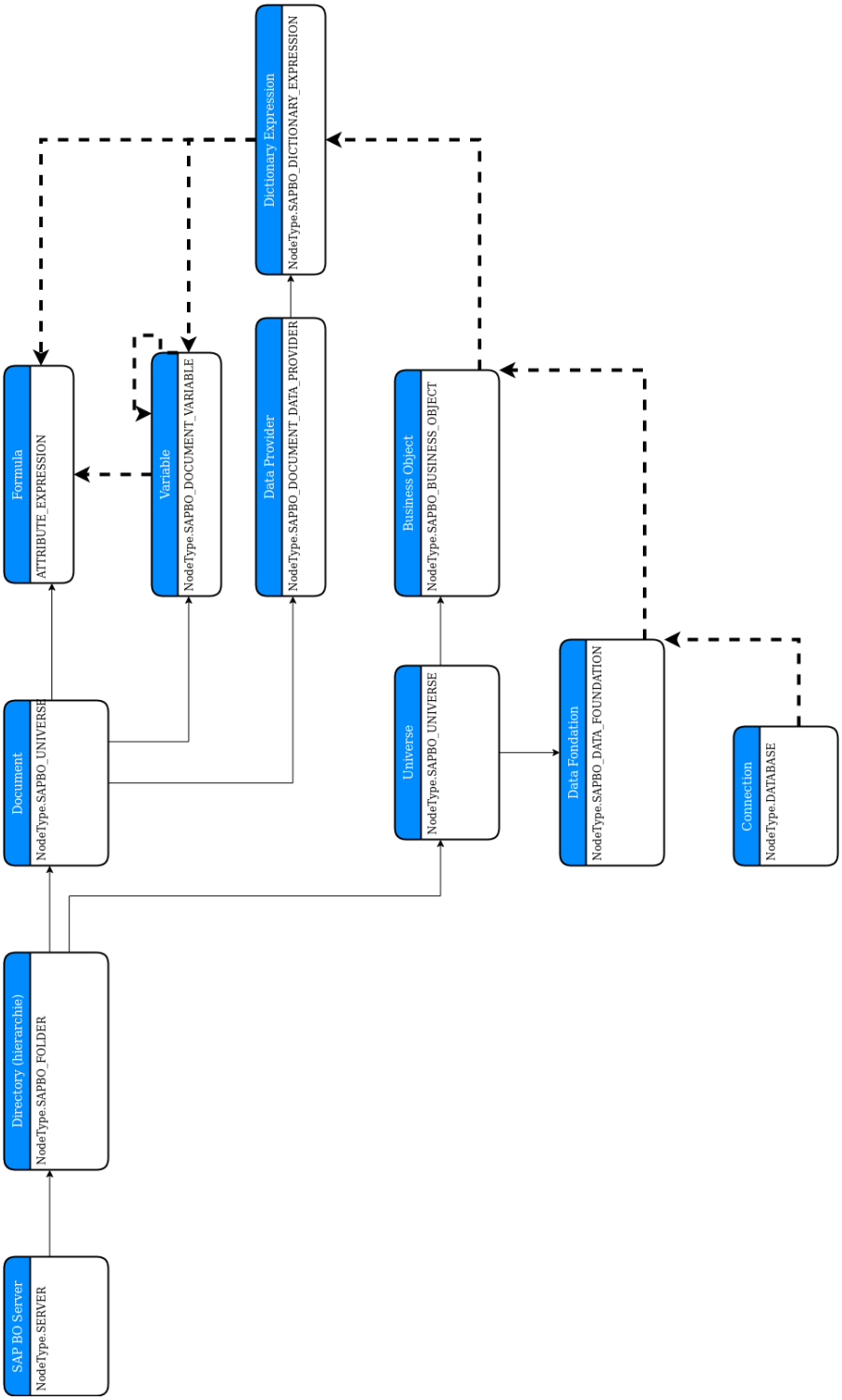
Na příkladu 4.3 lze vidět, že reálně dojde ke konstrukci prvního dotazu.

Poté, co je pro daný Business Objekt získána množina dotazů, je každý z těchto dotazů zpracován službou Dataflow Query Service, která společně s informacemi o související databázi (zejména typem a connection stringem) vrátí seznam použitých sloupců, pro každý sloupec se pod daným Business Objectem vytvoří vrchol, a poté se tyto nově vzniklé sloupcové vrcholy propojí s použitými sloupci v dané databázi. Tím dojde ke konstrukci toků na úrovni Universe - databáze, což je už jen krůček k finální podobě toků.

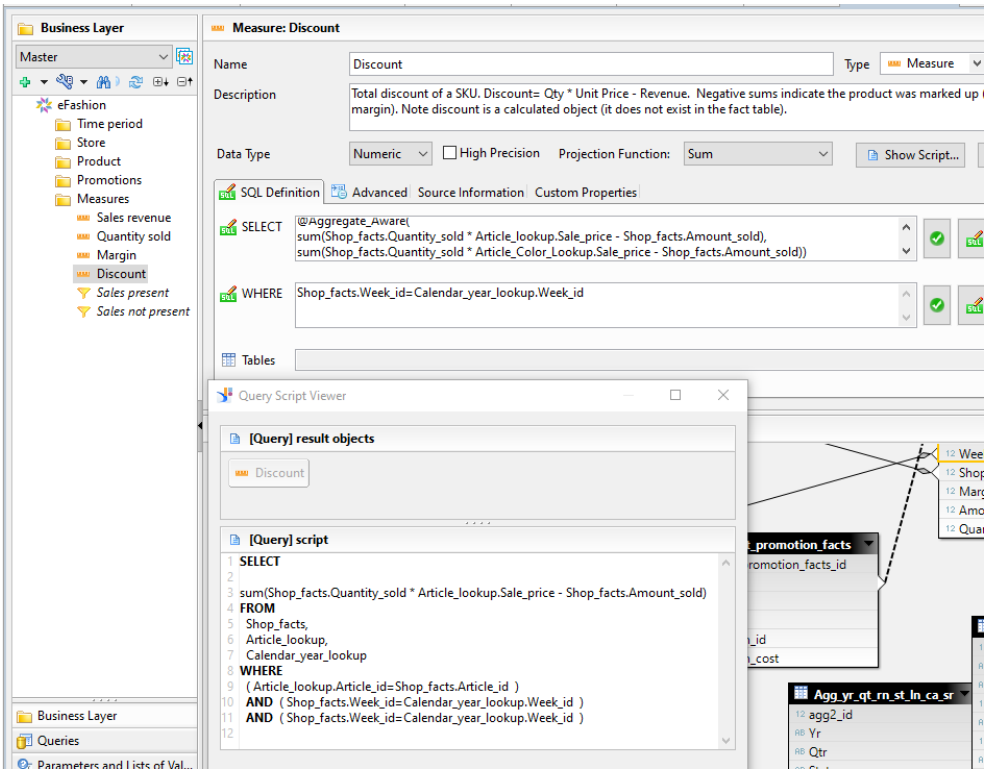
Následně se totiž do nově zkonstruovaného toku musí zapojit i Data Foundation, a to tak, že se vloží mezi Universe a databázi. Nově vytvořené sloupcové vrcholy se pak zkopírují a vytvoří pod každým závislým vrcholem (podvrcholy objektů Data Provider, Document Variable, Report Element) a dojde tak k propojení na nejnižší úrovni detailů. A tímto posledním krokem je tok kompletní.

4.6 Ukázka výsledku

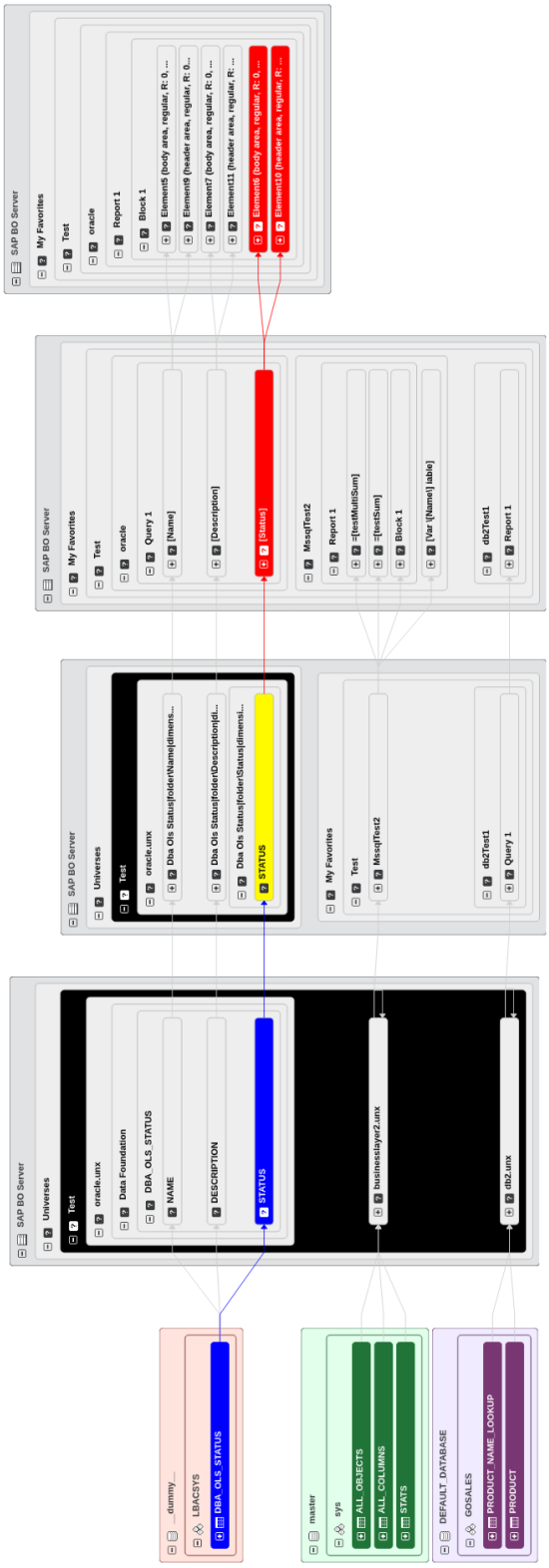
Na obrázku 4.4 lze vidět příklad výsledného grafu pro tři různé reporty, čerpajících data ze tří různých objektů typu Universe, které jsou postaveny na třech různých databázích (konkrétně databázích typu MS SQL, Oracle a DB2). Graf je kompletní, datové toky proudí od databází, přes všechny prostředníky, až k Report Elementům.



Obrázek 4.2: Vztahy mezi nejdůležitějšími objekty, plné šipky směřují k dětem, přerušované šipky znázorňují datové toky.



Obrázek 4.3: V nástroji IDT lze pomocí GUI zobrazit konkrétní SQL dotaz, který se pro daný Business Object použije, zveřejněná API však tuto možnost neposkytují



Obrázek 4.4: Příklad grafu vytvořeném SAP BO konektorem, vizualizovaném v Mantě

Kapitola 5

Testování

V následujících sekcích jsou analyzovány a diskutovány principy testování, které byly používané během implementace konektoru. Součástí kapitoly je i sekce o dosaženém „code coverage“.

5.1 Test-driven development

TDD je jednou z metodik vývoje software, ve které hrají testy psané během vývoje produkčního kódu důležitou roli.

Jakožto praktika s tímto jménem vznikla kolem Extrémního programování a Kenta Becka, který nicméně netvrdí, že je původním autorem originálního konceptu, ale pouze ho „znovuobjevil“. Jsou známy příklady z 50. a 60. let minulého století, kdy byla podobná metodika hojně využívaná. [36]

Údajně např. i autoři AWK vyvinuli onen slavný bashový program užíváním velmi podobné metodiky [37].

V následujících podsekcích se věnuji různým přístupům a chápáním TDD.

5.1.1 Různé definice

Aby o pojmu mohla být vedena smysluplná debata, je nejprve nutné si ujasnit, co přesně TDD znamená a obnáší. Z mé zkušenosti plyne mnoho nepochopení ohledně TDD právě z rozdílných představ o tom, co TDD vlastně je.

5.1.1.1 Kent Beck

Kent Beck jej ve své knize Test-Driven Development by Example představuje [38] ve formě dvou pravidel:

1. Nepsat ani řádek nového kódu, pokud nejprve neselhává automatizovaný test.

2. Odstraňovat duplicitu.

Tato dvě pravidla následně implikují tzv. „TDD mantru“ – „red/green/refactor cyklus“:

1. Red – napsat malý test, který selhává, možná se ani nezkompiluje.
2. Green – uspokojit test co nejrychleji, v této fázi jsou povoleny jakékoli hříchy vůči „čistému kódu“.
3. Refactor – odstranit všechny duplicity v kódu, které vznikly ve snaze uspokojit test co nejrychleji.

Jako primární důvod pro TDD uvádí Kent Beck to, že TDD je způsob, jakým lze v programování udržet strach pod kontrolou.

5.1.1.2 Robert Cecil Martin

Robert Cecil Martin (též známý pod přezdívkou „Uncle Bob“) zavádí mírně odlišnou definici pro TDD, kterou shrnuje ve svých třech pravidlech [39]:

1. Psaní produkčního kódu je povoleno pouze za předpokladu, že opraví selhávající jednotkový test.
2. Pokud již některý jednotkový test selhává, není možné psát další testy.
3. Psaní produkčního kódu je povoleno pouze v rozsahu opravení selhávajícího jednotkového testu.

Pozorný čtenář si všimne, že tato tři pravidla – pokud jsou striktně dodržována – nenechávají žádný prostor pro refaktoring, tedy pohybovali bychom se pouze v cyklu *red-green*. Rovněž se zde mluví o pojmu „jednotkový test“ (unit test) namísto obecného automatizovaného testu. Toto je rovněž v rozporu s definicí Kenta Becka, který ve své knize vyloženě píše [38], že TDD je zvědomováním hranice mezi rozhodováním a zpětnou vazbou během programování, a kontrole nad touto hranicí. Je možné mít pouze testy na aplikační úrovni a stále dělat TDD. Hranice mezi rozhodováním a zpětnou vazbou by v takovém případě byla velká, možná i několikadenní – ale pro extrémně dovedného programátora to může být postačující.

5.1.2 Různé směry

Nicméně pojem „unit test“ se ujal a různí lidé si pod ním představovali různé věci. Toto popisuje i Martin Fowler, nicméně uvádí i společné znaky, které obvykle unit testy mívají i v mnoha různých pojetích:

1. Nízkoúrovňové, zaměřené na malou část softwarového systému.
2. Psané samotnými programátory, s dopomocí určitého testovacího frameworku.

3. Znatelně rychlejší než jiné typy testů.

Martin Fowler uvádí, že jednou z rozdílností ve vnímání je to, co vlastně považujeme za jednotku (unit). V OOP (objektově orientovaném programování) míváme tendenci za jednotku považovat třídu, procedurální či funkcionální paradigma zase za jednotku považuje funkci. Martin Fowler to spíše považuje za věc závislou na konkrétní situaci – někdy to může být třída, někdy úzce související skupina tříd, jindy skupina metod v rámci jedné třídy. [40]

Co považuje za výraznější rozdíl v jednotlivých chápáních unit testů je přístup k testování z pohledu interakce se závislými objekty – má být testovaná jednotka sociální, nebo samotářská? Tato různá pojetí interakce časem vykrystalizovala do dvou různých pojetí testů, které Martin Fowler nazývá „classic“ a „mockist“. „Classic“ je sociální, a klasické je proto, že vzniklo dříve, a tím se mu dostalo více prostoru. „Mockist“ je, co se pojetím interakce týče, řadí mezi samotářský směr a více se rozšířil až později. Tyto dvě různé školy testování jsou známé též s přívlastky Detroit či Chicago (classic) a London (mockist). [41]

V praxi se tedy dva styly liší např. v tom, že sociální testy počítají s tím, že jednotky závislé na testované jednotce fungují správně. Oproti tomu samotářský test odstiňuje od implementace závislých jednotek a netestuje spolupráci samotnou, ale komunikaci mezi jednotlivými objekty.

V tomto směry jsou průkopníky autoři knihy *Growing Object-Oriented Software, Guided by Tests* (GOOS), jejichž kniha bývá vnímána jako základ pro Londýnskou školu. To proto, že autoři představují nový přístup k TDD, který zahrnuje časté užívání mocků. [42]

Vladimir Khorikov ve své obsáhlé kritice GOOS rozebírá koncepty z knížky, které mu přijdou vhodné a které ne. Jedním z konceptů, který odmítá, je časté používání mocků v TDD procesu, které podle něj vede ke křehkým testům a vytváření zbytečně složitého designu s cyklickými závislostmi, „header interfaces“ (interface, který plně napodobuje jedinou třídu, která jej implementuje) a zbytečně vysoký počet mezivrstev. Na příkladu úlohy z knihy pak ukazuje, jak jej lze navrhnout mnohem lépe, než je prezentované řešení v knize, což následně vede k nepotřebě užívání jakýchkoliv mocků při testování. [43]

Naopak Dan North, jenž bývá označován jako zakladatel Behavior Driven Development (BDD), vyzdvihuje užívání mocků jakožto mechanismu, který nám umožňuje uvažovat o testované jednotce v izolaci, což podle něj vede k možnosti aktivního ignorování okolí jednotky. Tím získáme možnost uvažovat o jednotce samostatně, což je jednodušší než uvažování o jednotce se všemi jejími vazbami. Mocky přirovnává k lékům proti bolesti, kdy můžeme přijít ke komplikovanému a hojně provázanému systému, o kterém je obtížné uvažovat v celé své šíři, a díky mockům jej můžeme rozdělit na menší jednotky. [44]

Další kritiku TDD a jeho jednotlivých směrů je možné nalézt v příloze A.

5.2 Osobní zkušenost a praktiky související s TDD

Následující koncepty a praktiky byly po celou dobu využívány, či začaly být využívány během implementace konektoru.

5.2.1 Požadavky na testy

Co od ideálních testů očekávám? V podstatě to, že pokud test prochází, aby to signalizovalo, že systém se chová tak, jak očekávám, a naopak - tedy neprocházející test signalizoval chybu v systému (chybu v tomto případě vnímám jako odchýlení od požadovaného chování). A navíc, statisticky řečeno – vyvarovat se chybám typu I a II, tedy testům falešně procházejícím či selhávajícím.

Čili pokud od testu požaduji verifikaci určitého chování systému a reálné chování systému neodpovídá mým představám, test musí selhat. Někdy se programátor může přepsat a nedopatřením napsat test jako tautologii - tedy ať je testovaný systém jakýkoliv, test vždy projde. TDD má proti takovýmto případům 1. fázi cyklu zvanou „red“ – programátor před implementací produkčního kódu test nejprve spustí, aby uviděl, že test selhal, tedy že se nejedná o tautologii. Následně implementuje testovaný požadavek a pomocí „green“ fáze se ujistí, že test opravdu testuje právě implementovaný požadavek a ne jiný.

Rovněž se může stát, že test z nějakého důvodu začne selhávat, ale přitom se chování systému nemění. Toto je důsledek toho, že daný test netestuje pouze chování, ale i něco jiného, mnohdy nějaký implementační detail, který ale test vůbec mít na starost nemá. Taková situace nastává v systémech, které jsou nadmíru mockované a nedochází k testování chování samotného. TDD se snaží těmto chybám vyvarovat především kvůli tomu, aby bylo možné provádět refaktoring.

Uncle Bob ve své knize Clean Code popisuje požadavky na „čisté testy“: [31]

- Fast – testy by měly běžet krátkou dobu a poskytnout co nejrychlejší výsledek.
- Independent – testy by měly běžet nezávisle na sobě, mělo by být možné je spouštět v libovolném pořadí, a selhání jednoho testu by nemělo ovlivnit selhání jiného testu.
- Repeatable – testy by měly být spustitelné pod různými prostředími a poskytovat stejné výsledky.
- Self-Validating – testy by měly poskytovat jasnou návratovou hodnotu – buď projdou, nebo selžou. Nemělo by se stávat, že pro vyhodnocení testu by měl tester něco kontrolovat manuálně (např. výpisy v logu).

- Timely – testy by měly být psané těsně předtím, než je napsán produkční kód. Opačný postup by mohl vést k hůře testovatelnému produkčnímu kódu.

Tyto požadavky považuji za vcelku rozumné, nicméně není nutné vše dodržovat dogmaticky.

Rychlé testy jsou rozhodně lepší než pomalé, nicméně nesmí se této rychlosti dosáhnout na úkor objemu testovaného požadovaného chování.

Nezávislost jednotlivých testů je velmi důležitou vlastností nejen pro debugování, ale navíc nám umožňuje spouštět testy paralelně, tudíž rychleji. Tato vlastnost zřejmě nabude více na významu v budoucnosti, kdy se očekává škálování výkonu procesorů ve formě přidávání výpočetních jader. O nezávislost ve smyslu nízké provázanosti a vysoké modularity usilujeme i v produkčním kódu – v tomto aspektu nám test-first přístup ve skutečnosti sám o sobě pomáhá, jak vyplynulo z publikované studie o TDD [45].

Vskutku, sám jsem na sobě pozoroval silnější tíhnutí k praktikám známých spíše z funkcionálního programování – čisté (pure) funkce a neměnné (immutable) objekty či struktury, což se v mém Java kódu projevuje především používám takových jazykových konstruktů, které toto umožňují (např. Lambda výrazy a jejich používání v Java Streams).

Opakovatelnost není plně dosažena, neboť testy modulu extractor vyžadují spojení se SAP BO serverem. Rovněž vyžadují být spuštěné na stroji, kde jsou nainstalované Client Tools, což omezuje operační systém pouze na Windows (na jiné operační systémy Client Tools nevycházejí).

Self-Validating jsou mé testy automaticky, neboť je pro jejich běh využit framework JUnit.

A jelikož vyvíjím stylem TDD, jsou mé testy i dle definice Uncla Boba „timely“.

5.2.2 Testy jako specifikace

Testy tedy slouží jako prostředek vymáhání požadavků na systém. Kdykoli se změní požadavky, měly by se změnit i testy, pokud chceme mít jistotu, že se implementovaný systém chová skutečně tak, jak potřebujeme.

Co tedy dělat, když se narazí na chybu? Nejprve napsat test, který danou chybu zachytává (tedy selže), a následně implementovat opravu v produkčním kódu.

5.2.3 Testy jako dokumentace

Testy dále slouží jako dobrá dokumentace toho, jak systém funguje. Taková dokumentace má několik výhod - je automaticky ověřitelná, pokrývá mnoho úrovní detailu (od nejnižších unit testů až po systémové testy), a na konkrétních příkladech je velmi dobře pochopitelné, jak se daná funcionalita používá.

Testy ale nemusí sloužit jako dokumentace jen k našemu kódu, ale i ke kódu třetích stran. Tato praktika se nazývá „learn test“ - test, který obsahuje chování takových knihoven, na kterých je náš kód závislý. Toto se hodí zejména v případě, kdy se programátor seznamuje s knihovnou, kterou nezná, a do learn testů napíše základní případy jejího očekávaného používání. Až se dostane k její aplikaci v produkčním kódu, stačí se pro inspiraci podívat do těchto testů a zkopírovat kód. Takové testy rovněž pomohou vývojáři, který se k modulu dostane po původním vývojáři, neboť i on může být nezkušený s danou knihovnou.

5.2.4 Nedogmatičnost „green“ fáze

Co se týče zelené fáze, zde se držím doporučeného přístupu Kenta Becka, že zelená fáze (implementační) není dogmatická v tom, že můžeme napsat pouze tolik produkčního kódu, kolik stačí na projití právě selhávajícího testu (jako doporučuje ve svém 3. pravidle Uncle Bob). Takový přístup by dle mého názoru pouze zbytečně zdržoval. Pokud rovnou vím, jak má daná implementace vypadat, napíšu ji, a testy pro dosud neověřené chování dodám krátce poté. I zde však dodržuji červenou fázi – do assertu obvykle nejprve napíšu ne očekávanou, ale záměrně chybnou hodnotu – a pokud test spustím a selže, opravím to na správnou (očekávanou) hodnotu. Pokud je test zelený, přejdu na refaktoring, jinak opravím implementaci. Tímto přístupem předcházím tautologii a jiným formám nic nevypovídajících testů.

5.2.5 Testy nahrazují debugger

Další věcí, které jsem si všiml ve své každodenní praxi, je nepotřeba aplikace debuggeru. To je dáno tím, že v TDD je možné dělat mezi jednotlivými červenými a zelenými fázemi libovolně velké kroky. Typicky čím zkušenější programátor, tím větší kroky si může dovolit. Tyto kroky lze ale libovolně upravovat – pokud před sebou máme jednoduchou funkcionalitu a cítíme se na to, můžeme dělat větší kroky. Když naopak narazíme na problém, můžeme původní problém rozebrat na menší problémy a skrze testy menších funkcionalit se dostat k finálnímu řešení, zatímco jiní vývojáři by v takové situaci využili funkci debuggeru.

5.3 Testy v konektoru

Až na poslední fázi implementace konektoru – integraci s Mantou – jsem po celou dobu vývoje uplatňoval TDD.

5.3.1 Pokrytí kódu

Díky aplikaci TDD bylo v jednotlivých modulech dosaženo poměrně dobrých výsledků pokrytí kódu testy (měřeno 16.5.2020):

- Extractor – 72,3% (SDK Extractor 87,8%)
- Resolver – 97,8%
- Object Mapper – 100%
- Model – 100%
- Dataflow Generator – 85% (měřeno bez vygenerovaného ANTLR parseru)

Při pohledu na výsledky zjistíme, že nejhůře je na tom modul Extractor. Jsou k tomu především dva důvody:

- V produkčním kódu extractoru se zachytává a loguje mnoho různých chybových stavů ve vztahu se SAP BO serverem, které se obtížně navozují. Nyní tyto situace testovány nejsou, ale pokud by to bylo žádoucí, bylo by možné vytvořit novou vrstvu (tzv. wrapper) pro komunikaci se serverem, a tuto novou vrstvu v testech mockovat. Takováto úprava kódu by jistě zvýšila míru pokrytí testy. Proč vytvářet novou vrstvu a nemockovat rovnou server? Je to proto, že chování serveru je z velké části neznámé, a mockování něčeho neznámého nemusí být úplně přesné. Zatímco pokud by se mockoval wrapper pro něco neznámého, dochází už k mockování něčeho, co máme pod kontrolou – takové testy nám ve výsledku poskytnou více jistoty a stability.
- Extractor musel být v nedávné době integrován s Mantou. TDD v tomto případě nebylo použito, protože nutným předpokladem pro jeho aplikaci je možnost přesné simulace prostředí Manty. Taková simulace by nebyla triviální, a proto se testovalo manuálně vůči reálnému prostředí Manty. Dodání automatizovaných testů pro integraci s Mantou je odloženo na později.

U Dataflow Generatoru vyvstal podobný problém jako u Extractoru – manuální testování vůči reálnému prostředí Manty bylo v této fázi projektu (měsíc před plánovaným releasem) vyhodnoceno jako užitečnější než automatizované testování vůči simulovanému prostředí. Automatizované integrační testování je záležitostí budoucího vývoje.

U ostatních modulů bylo dosaženo vysokého procenta pokrytí testy, a to především proto, že nad těmito moduly jsem měl plnou kontrolu (ve smyslu vstupů a výstupů) a nemusely komunikovat s žádnou technologií třetí strany, což vedlo k jednoduché simulaci prostředí, což vedlo ke snadno vytvářeným testům.

5.3.2 Testování výjimek

Ve své implementaci modulů jsem musel několikrát řešit případ, kdy jsem očekával, že testovaný kód vyhodí výjimku, a tuto skutečnost jsem chtěl testovat. Jako příklad uvádím test v modulu Extractor, ve třídě ServerConnectionTest.

```
@Test
public void wrongPassword() {
    ServerConnection connection = new ServerConnection(
            TestParameters.HOST, TestParameters.PORT_CMC,
            TestParameters.USER_NAME,
            TestParameters.USER_PASSWORD + "someWrongPasswordSuffix",
            TestParameters.AUTH, TestParameters.PORT_REST);

    boolean wasThrown = false;
    try {
        ServerSession session = connection.establishSession();
    } catch (RuntimeException e) {
        wasThrown = true;
    }

    if (!wasThrown) {
        session.logoff();
        fail();
    }
}
```

Třída ServerConnection slouží jako šablona, která obsahuje všechna potřebná data k navázání spojení se serverem. Toto navázání se provádí metodou *establishSession()*, která vrátí buď aktivní instanci `ServerSession`, nebo je vyhozena výjimka typu `RuntimeException`. V produkčním kódu dává smysl použít tento „zákeřný“ typ výjimky, jelikož pokud se nepodaří navázat spojení se serverem, nemá smysl dál pokračovat v extrakci. Použití RuntimeException je tedy v souladu s doporučením v knize Effective Java, kde se radí používat tento typ výjimek u chyb, ze kterých se nedá zotavit [27].

V testu se dále vyskytuje třída `TestParameters`, která obsahuje různé konstanty, v tomto případě přistupujeme ke korektním parametrům připojení k serveru. Výraz *TestParameters.USER_PASSWORD + "someWrongPasswordSuffix"* slouží k předání nekorektního hesla pro daného uživatele – nekorektnost hesla je zajištěna připojením neprázdné přípony ke korektnímu heslu.

Smyslem testu je zkontrolovat, zda se veřejná metoda *establishSession()* bude chovat jak má – tedy že pro nekorektní heslo vyhodí výjimku *RuntimeException*. Příkaz `session.logoff();` slouží k ukončení případného navázaného

připojení, které by sice správně nemělo vůbec vzniknout, toto ukončení je tam spíše pro jistotu – pokud by se totiž nějakým nedopatřením podařilo navázat připojení a korektně se neukončilo, mohlo by to působit problémy (tento kód jsem testoval na serveru, jehož instalace má omezení na 5 souběžně existujících relací – pokud by se tyto relace korektně neuzavíraly, mohly by jiné testy testující korektní navázání připojení selhávat).

Jak lze na příkladu vidět, zvolil jsem řešení manuálního odchytávání výjimky a následný assert na to, že správná výjimka byla zachycena.

Ve frameworku JUnit 4 jsem ale mohl zvolit i jiné řešení, s typem očekávané výjimky v anotaci:

```
@Test(expected = RuntimeException.class)
public void wrongPassword() {
    ServerConnection connection = new ServerConnection(
            TestParameters.HOST, TestParameters.PORT_CMC,
            TestParameters.USER_NAME,
            TestParameters.USER_PASSWORD + "someWrongPasswordSuffix",
            TestParameters.AUTH, TestParameters.PORT_REST);

    ServerSession session = connection.establishSession();
    session.logoff();
}
```

Toto řešení je o něco kratší, nicméně je tam menší kontrola v tom, odkud očekáváná výjimka může být vyhozena – očekávaná výjimka by mohla být klidně vyhozena v konstruktoru `ServerConnection` či během uzavírání relace příkazem `session.logoff();`.

V JUnit 5 lze ale toto chování jednoduše testovat pomocí metody *assertThrows()*:

```
@Test
public void wrongPassword() {
    ServerConnection connection = new ServerConnection(
            TestParameters.HOST, TestParameters.PORT_CMC,
            TestParameters.USER_NAME,
            TestParameters.USER_PASSWORD + "someWrongPasswordSuffix",
            TestParameters.AUTH, TestParameters.PORT_REST);

    assertThrows(RuntimeException.class, () -> {
        ServerSession session = connection.establishSession();
        session.logoff();
    });
}
```

Toto řešení je rovněž krátké, nicméně také obsahuje nejistotu ohledně toho, zda očekávanou výjimku vyhodí příkaz `connection.establishSession();` či `session.logoff();`, takže úplně korektní řešení v JUnit 5 by bylo podobně dlouhé jako mnou implementované v JUnit 4.

Závěr

Cílem práce bylo navrhnout a implementovat funkční prototyp modulu, který provede syntaktickou a sémantickou analýzu reportů z nástroje SAP BusinessObjects a následně její výsledek využije pro analýzu datových toků, přičemž implementovaný modul bude zařazen do projektu Manta.

Během práce na konektoru bylo potřeba řešit hned několik nečekaných problémů. V rámci extrakce se musely řešit speciální požadavky na běhové prostředí. Implementaci modulu Resolver komplikovala absence kvalitní dokumentace k nástroji SAP BusinessObjects. A při přidávání toků do výsledného grafu bylo potřeba zpracovávat nejen jeden, ale rovnou dva různého druhy výrazů.

I přes všechny nečekané problémy se povedlo konektor včas dostat do požadovaného stavu. Stanovený cíl byl plně dosažen – implementovaný modul dokonce dosáhl takové kvality, že je v plánu jej zařadit do další verze Manty, která bude vypuštěna k zákazníkům v půlce června tohoto roku. V rámci implementace bylo z velké míry využito metodiky TDD, což mělo za důsledek vysokou míru pokrytí kódu testy v případech, kde byla tato metodika použita.

Do budoucna je v plánu konektor dále podporovat a opravovat případné chyby. Rovněž se počítá s dalším rozšiřováním konektoru – především různých typů datových zdrojů (aby mohly pocházet např. ze souborů a ne pouze z databází), rozšířením parserů pro oba typy výrazů, aby bylo možné rozlišit mezi přímými a nepřímými toky, a podporou ostatních funkcionalit, které budou požadovat zákazníci.

Literatura

[1] SAP SE. BI Platform Client Tools [software]. [přístup 14. 5. 2020]. Dostupné z: https://www.sap.com/cmp/td/sap-bi-edge-edition.html [Požadavky na systém: procesor 2.0 GHz Core CPU, operační paměť 2 GB RAM, operační systém Microsoft Windows Vista, Windows 7/8/10, volné místo na disku 3,5 GB].

[2] MANTA Flow Architecture [online]. Manta. [vid. 2020-05-10]. Dostupné z: https://mantatools.atlassian.net/wiki/spaces/MTKB/pages/70230122/MANTA+Flow+Architecture.

[3] Introduction to the Dependency Mechanism. [online]. Apache Maven Project. [vid. 2020-05-10]. Dostupné z: http://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html.

[4] Andrš, J.: Manta Tools: How It All Started [online]. Manta, 2014. [vid. 2020-05-10]. Dostupné z: https://getmanta.com/blog/manta-tools-how-it-all-started/.

[5] Cyprich, P.: Rozhodnuto. S Czech ICT Alliance jedou do Silicon Valley dva startupy – Manta a realPad. [online]. Tyinternety.cz, 2015. [vid. 2020-05-10]. Dostupné z: https://tyinternety.cz/startupy/rozhodnuto-s-ict-alliance-jedou-silicon-valley-dva-startupy-manta-realpad/.

[6] Supported technologies, Databases. [online]. Manta. [vid. 2020-05-10]. Dostupné z: https://getmanta.com/technologies/databases/.

[7] Supported technologies, Data Integration. [online]. Manta. [vid. 2020-05-10]. Dostupné z: https://getmanta.com/technologies/data-integration/.

[8] Supported technologies, Programming Languages. [online]. Manta. [vid. 2020-05-10]. Dostupné z: https://getmanta.com/technologies/programming-languages/.

[9] Supported technologies, Modelling. [online]. Manta. [vid. 2020-05-10]. Dostupné z: https://getmanta.com/technologies/modeling/.

[10] Supported technologies, Reporting and Analysis. [online]. Manta. [vid. 2020-05-10]. Dostupné z: https://getmanta.com/technologies/reporting-analysis/.

[11] Košvanec, P.: Analýza datových toků v reportovacích nástrojích. Praha, 2019. Diplomová práce. ČVUT v Praze, Fakulta informačních technologií, Katedra softwarového inženýrství.

[12] Collibra Platform Gain full visibility across your data landscape [online]. Collibra. [vid. 2020-05-10]. Dostupné z: https://www.collibra.com/platform.

[13] Enterprise Data Catalog [online]. Informatica. [vid. 2020-05-10]. Dostupné z: https://www.informatica.com/products/data-catalog/enterprise-data-catalog.html.

[14] IBM InfoSphere Information Governance Catalog [online]. IBM. [vid. 2020-05-10]. Dostupné z: https://www.ibm.com/us-en/marketplace/information-governance-catalog.

[15] Business Intelligence Software. [online]. SAP. [vid. 2020-05-10]. Dostupné z: https://www.sap.com/cz/products/bi-platform.html.

[16] Straight Talk: Pros and Cons of SAP Business Objects [online]. 5000fish, Inc. [vid. 2020-05-10]. Dostupné z: https://www.yurbi.com/blog/straight-talk-pros-and-cons-of-sap-business-objects/.

[17] Dimension,Detail and Measure. [online]. SAP. [vid. 2020-05-10]. Dostupné z: https://answers.sap.com/questions/8745194/dimensiondetail-and-measure.html.

[18] SAP IDT - Business Layer. [online]. Tutorials Point. [vid. 2020-05-10]. Dostupné z: https://www.tutorialspoint.com/sap_idt/sap_idt_business_layer.htm.

[19] SAP IDT - Data Foundation Layer. [online]. Tutorials Point. [vid. 2020-05-10]. Dostupné z: https://www.tutorialspoint.com/sap_idt/sap_idt_data_foundation_layer.htm.

[20] Java 8 - Streams. [online]. Tutorials Point. [vid. 2020-05-10]. Dostupné z: https://www.tutorialspoint.com/java8/java8_streams.htm.

[21] Package java.util.stream. [online]. Oracle. [vid. 2020-05-10]. Dostupné z: https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html.

[22] JSR 305: Annotations for Software Defect Detection. [online]. Java Community Process. [vid. 2020-05-10]. Dostupné z: https://jcp.org/en/jsr/detail?id=305.

[23] @Nullable and @NotNull. [online]. JetBrains. [vid. 2020-05-10]. Dostupné z: https://www.jetbrains.com/help/idea/nullable-and-notnull-annotations.html#.

[24] Using null annotations. [online]. Eclipse. [vid. 2020-05-10]. Dostupné z: https://help.eclipse.org/photon/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Ftasks%2Ftask-using_null_annotations.htm.

[25] Annotation Type Nullable. [online]. Spring Framework. [vid. 2020-05-10]. Dostupné z: https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/lang/Nullable.html.

[26] Should Java 8 getters return optional type? [online]. Stack Exchange Inc. [vid. 2020-05-10]. Dostupné z: https://stackoverflow.com/a/26328555.

[27] Bloch, J.: *Effective Java*. Addison-Wesley Professional, třetí vydání, ISBN 978-0134685991.

[28] Colebourne, S.: Java SE 8 Optional, a pragmatic approach. Stephen Colebourne's blog [online]. Stephen Colebourne, 2015. [vid. 2020-05-10]. Dostupné z: https://blog.joda.org/2015/08/java-se-8-optional-pragmatic-approach.html.

[29] Apache Software Foundation. Apache Subversion [software]. Říjen 2000. [přístup 14. 5. 2020]. Dostupné z: https://subversion.apache.org/ [Požadavky na systém: Operační systém - Multiplatformní software].

[30] CD Foundation The Linux Foundation. Jenkins [software]. Únor 2011. [přístup 14. 5. 2020]. Dostupné z: https://www.jenkins.io/ [Požadavky na systém: operační paměť 256 MB RAM, volné místo na disku 1 GB].

[31] Martin, R. C.: *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, první vydání, ISBN 978-0132350884.

[32] About The ANTLR Parser Generator [online]. ANTLR / Terence Parr. [vid. 2020-05-10]. Dostupné z: https://www.antlr.org/about.html.

[33] Universe SDK. [online]. SAP. [vid. 2020-05-10]. Dostupné z: https://archive.sap.com/documents/docs/DOC-38810.

[34] Getting the Details of a Report Element [online]. SAP SE. [vid. 2020-05-10]. Dostupné z: https://help.sap.com/viewer/58f583a7643e48cf944cf554eb961f5b/4.2.4/en-US/7da1b5766f701014aaab767bb0e91070.html.

[35] How to fetch the query script of a business object using SDK (or some other API)? [online]. SAP SE. [vid. 2020-05-10]. Dostupné z: https://answers.sap.com/questions/12931313/how-to-fetch-the-query-script-of-a-business-object.html.

[36] Why does Kent Beck refer to the "rediscovery"of test-driven development? What's the history of test-driven development before Kent Beck's rediscovery? [online]. Quora. [vid. 2020-05-10]. Dostupné z: https://www.quora.com/Why-does-Kent-Beck-refer-to-the-rediscovery-of-test-driven-development-Whats-the-history-of-test-driven-development-before-Kent-Becks-rediscovery.

[37] Hamilton, N.: The A-Z of programming languages: AWK. COMPUTERWORLD [online]. IDG Communications, Inc., 2008. [vid. 2020-05-15]. Dostupné z: https://www.computerworld.com/article/2535126/the-a-z-of-programming-languages--awk.html?page=2.

[38] Beck, K.: *Test Driven Development: By Example*. Addison-Wesley Professional, první vydání, ISBN 978-0321146533.

[39] The Three Laws of TDD. [online]. BUTUNCLEBOB.COM. [vid. 2020-05-10]. Dostupné z: http://www.butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd.

[40] Fowler, M.: UnitTest. martinFowler.com [online]. Martin Fowler, 2014. [vid. 2020-05-10]. Dostupné z: https://martinfowler.com/bliki/UnitTest.html.

[41] Fowler, M.: Mocks Aren't Stubs. martinFowler.com [online]. Martin Fowler, 2007. [vid. 2020-05-10]. Dostupné z: https://martinfowler.com/articles/mocksArentStubs.html.

[42] Freeman, S.; Pryce, N.: *Growing Object-Oriented Software, Guided by Tests*. Addison-Wesley Professional, první vydání, ISBN 978-0321503626.

[43] Khorikov, V.: Growing Object-Oriented Software, Guided by Tests Without Mocks. Enterprise Craftsmanship [online]. Vladimir Khorikov, 2016. [vid. 2020-05-10]. Dostupné z: https://enterprisecraftsmanship.com/posts/growing-object-oriented-software-guided-by-tests-without-mocks/.

[44] North, D.: GOTO 2016 • Software that Fits in Your Head • Dan North [video]. YouTube [online]. GOTO Conferences, 2016. [vid. 2020-05-10]. Dostupné z: https://www.youtube.com/watch?v=4Y0tOi7QWqM.

[45] Madeyski, L.: *Test-Driven Development: An Empirical Evaluation of Agile Practice*. Springer, první vydání, ISBN 978-3642042874.

[46] TDD is dead. Long live testing. [online]. David Heinemeier Hansson. [vid. 2020-05-10]. Dostupné z: https://dhh.dk/2014/tdd-is-dead-long-live-testing.html.

[47] Is TDD Dead? [online]. Martin Fowler. [vid. 2020-05-10]. Dostupné z: https://martinfowler.com/articles/is-tdd-dead/.

[48] Cooper, I.: DevTernity 2017: Ian Cooper - TDD, Where Did It All Go Wrong [video]. YouTube [online]. DevTernity, 2017. [vid. 2020-05-10]. Dostupné z: https://www.youtube.com/watch?v=EZ05e7EMOLM.

[49] Introducing BDD [online]. Dan North & Associates . [vid. 2020-05-10]. Dostupné z: https://dannorth.net/introducing-bdd/.

[50] Parker, M.: TDD: The Bad Parts — Matt Parker [video]. YouTube [online]. VMware Tanzu, 2016. [vid. 2020-05-10]. Dostupné z: https://youtu.be/xPL84vvLwXA.

[51] TestPyramid [online]. Martin Fowler. [vid. 2020-05-10]. Dostupné z: https://martinfowler.com/bliki/TestPyramid.html.

[52] Martin, R. C.: The Little Mocker. The Clean Code Blog [online]. Robert C. Martin, 2014. [vid. 2020-05-10]. Dostupné z: https://blog.cleancoder.com/uncle-bob/2014/05/14/TheLittleMocker.html.

[53] Snyder, C.; Lopez, S. J.: *Handbook of Positive Psychology*. Oxford University Press, první vydání, ISBN 978-0195135336.

[54] Souder, B.: Flow at Work: The Science of Engagement and Optimal Performance. PositivePsychology.com [online]. PositivePsychology.com, 2020. [vid. 2020-05-10]. Dostupné z: https://positivepsychology.com/flow-at-work/.

Příloha A

Kritika TDD

TDD od 90. let minulého století prošlo určitým vývojem a v posledních letech se dočkalo i vln kritiky, jejíž vybrané případy jsou dále popsány.

A.1 TDD is dead. Long live testing.

Tento článek napsal [46] David Heinemeier Hansson (zkráceně DHH, autor frameworku Ruby on Rails) na jaře roku 2014 a rozvířil tím vody kolem TDD.

Ve svém článku kritizuje několik věcí – má pocit, že stoupenci TDD jsou příliš fanatičtí a jakýkoliv jiný způsob vývoje je jimi zavrhován, dále kritizuje unit testy a snahy o jejich dosažení, kdy během toho procesu vzniká mnoho mocků a nepřímých závislostí. Jako řešení doporučuje méně unit testů a více systémových testů, konkrétně doporučuje nástroj Capybara.

Následně vzniklo několik nahraných rozhovorů, kde se o TDD bavil společně s Kentem Beckem a Martinem Fowlerem. Společně se snažili přijít na to, kde mívají programátoři s TDD problémy. [47]

Spíše než o hledání jediné pravdy šlo o sdílení zkušeností a vyjasňování si pojmů.

Kent Beck vyzdvihuje induktivní podstatu TDD – řešení problémů od konkrétních případů k obecnému řešení, snahu docílit co nejkratších „feedback loops“ (cyklů zpětné vazby) a tím co nejmenší úzkosti související s vývojem programů.

DHH poznamenává, že zažil situace, kdy mu TDD poskytovalo velmi dobré výsledky, ale šlo o situace, kde velmi dobře znal vstupy a výstupy a nemuselo se řešit spoustu věcí okolo (kontexty), šlo o čistá („pure“)[6] rozhraní.

Kent Beck dále zmiňuje, že softwarový vývoj (a tedy i TDD) je v jeho vnímání o neustálém hledání kompromisů, a to se týká i návrhu (designu) různých druhů meziproduktů. A zdůrazňuje, že často je to otázka pocho-

[6] Označení „pure“ se mj. používá pro funkce, jejich výstup je plně daný pouze vstupem, tedy nedochází se zacházením s jinými daty a k žádným vedlejším účinkům („side effects“).

pení správného návrhu. Jako příklad zmiňuje kompilátory – pokud budeme mít pouze end-to-end testy, časem můžeme přijít na to, že bychom chtěli mít k dispozici i nějaký meziprodukt ve formě parsovacího stromu, protože tím získáme dvě ortogonální dimenze a můžeme testovat zvlášť produkci parsovacího stromu, zvlášť výstup kompilátoru při daném parsovacím stromu, a tím tedy dostaneme možnost testování na nižších úrovních detailů, přičemž toto rozdělení bylo způsobeno návrhem. Kent zmiňuje, že často je to o hledání nejlepšího možného designu, a takový design obvykle nevyžaduje mockování. Dodává, že ve své praxi mocky téměř nevyužívá. Toto vysvětluje tím, že pro správné užívání TDD potřebuje pouze jednoduše refaktorovat. Pokud ve svých testech (nad)užíváme mocky, vzniká nám tím závislost na konkrétní implementaci a nikoliv na rozhraní testovaného modulu. A pokud změna v implementaci bez ovlivnění funkcionality (= refaktoring) rozbije testy, považuje to za moc vysokou cenu za postupný vývoj.

Martin Fowler se přidává do diskuze s tím, že TDD a unit testing nemají nic společného s izolací, která by nutně vyžadovala mockování, vnímá to spíš jako různá chápání TDD. Dodává, že ve své praxi má stejný přístup jako Kent, tedy že téměř nepoužívá mocky.

A.2 TDD, Where Did It All Go Wrong?

Tato přednáška od Iana Coopera [48] se dotýká mnoha problémů praktikování TDD, které ve své praxi zažil. Tvrdí o sobě, že TDD praktikuje již přes 10 let, a jelikož pracuje na dlouhodobých projektech, setkává se tak běžně se starším kódem, který byl vyvíjen stylem TDD.

Jedna z věcí, které si za tu dobu všiml, je problematický refaktoring staršího kódu vyvíjeném stylem TDD. Podle něj se při změnách v implementaci mívají tendenci rozbíjet jejich testy (zmiňuje, že především ty silně mockované). TDD tak nedostál svého slibu bezproblémového refaktoringu. TDD slibovalo snadné zapracování změn implementačních detailů, ale místo toho se báli cokoliv změnit, protože pak museli opravovat i spoustu dalších testů.

Další věcí je nečitelnost starších testů - pokud test selže, nejen, že se často neví proč, ale dokonce z něj ani nejsou schopní odvodit, co se vlastně přesně testovalo, mnohdy kvůli nadměrnému používání mocků - a pak přicházejí na to, že se testují pouze mocky, tedy nic, co by mělo reálný dopad na produkční kód, a v takovém případě za nejlepší řešení považuje daný test jednoduše smazat.

Následně se věnuje ATDD (akceptační TDD) – jedná se o vysokoúrovňové testy, které bývají psané v přirozeném jazyce a slouží jako komunikace mezi programátory a zákazníky. Takové testy, pak ale Ian vysvětluje, tráví většinu času v „red“ fázi, jelikož trvá poměrně dlouho, než se jeden takový testový scénář implementuje, a tím dochází ke značnému narušení programátorského

„flow“, které je do značné míry podporováno téměř okamžitou zpětnou vazbou vývojářských testů.

Ian následně přichází s tvrzením, že TDD má všechny odpovědi na problémy s TDD – tedy že problémem je, že původní TDD popsané Kentem Beckem funguje, my jsme na něj ale nabalili spoustu praktik, které mu ublížily. Doporučuje tedy původní knihu „TDD by Example“ [38], ke které se sám často vrací a podotýká, že mnoho problémů s TDD (i nynějších) je v této knize diskutováno a je tedy vhodná jak pro začátečníky, tak i pokročilé.

Jako nejdůležitější poučku z celé knihy považuje větu: „Avoid testing implementation details, test behaviors.“ (Testovat chování, nikoliv implementační detaily.). To je podle Iana klíčová věc, která se v dnešním TDD opomíjí. Tedy že podnětem pro přidání testu není přidání metody, ale přidání požadavku, který chceme implementovat. Reálně se pak samozřejmě testují třídy a metody (ale ne všechny) – zde zdůrazňuje, že se testuje veřejné API daných modulů, kontrakt modulu vůči ostatním. Jedním z důvodů, proč testovat veřejné API, je, že bývá stabilní a příliš často se nemění – jeho implementace se může měnit, ale jeho vnější forma je stabilnější. Dodává, v unit testech se obvykle klade důraz na testování tříd, ale mnohdy jsou třídy implementačním detailem nějakého modulu, tudíž nemusí být nutné je všechny testovat.

Ve své přednášce dále zmiňuje Behavior Driven Develpment (BDD), ale spíše jeho rané fáze a postřehy, které sepsal Dan North [49] – tedy že když se v TDD kladl důraz ne na jednotky (unit testy a vše, co si pod tím programátoři vybaví), ale na chování, TDD funguje lépe.

A co tedy izolace? Podle Iana je jednotkou izolace test, nikoliv testovaný systém. Tedy testy se vzájemně nemají ovlivňovat, ale jejich konkrétní implementaci nemusíme nijak izolovat, což často bývá podnět pro vytváření mocků. Ian ale mocky úplně nezavrhuje – hodí se např. pro testování databází, ale z jiného důvodu, než si lidé často myslí – nejde o izolaci implementace od databáze, ale testů od vedlejších účinků (side effects), kdy jeden test může během svého provádění upravit záznamy v databázi, a pokud se po testu stav databáze neuvede do původního stavu před testem, takový stav pak ovlivňuje další testy, a tím je porušena izolace testů. Dalším důvodem může být čas, protože testy by měly běžet co nejkratší dobu. Podotýká ale, že může být naprosto v pořádku, pokud testy sdílí nějaký společný kontext a např. komunikují s databází nebo souborovým systémem.

A.3 TDD: The bad parts

Matt Parker ve své přednášce vysvětluje [50], proč se věnovat TDD – chceme vyvíjet rychle, ale nejen na počátku projektu, ale i v dlouhodobém měřítku. Abychom toho byli schopni, potřebujeme pracovat s čistým kódem. Naše codebase je ale udržována čistou jen skrze pravidelný refaktoring, a abychom jej mohli bezpečně provádět, potřebujeme mít jistotu, že jsme v průběhu něj nic

nerozbili. A jistotu, že naše změny skutečně nic nerozbily, poskytují kvalitní testy, které nám zároveň poskytují rychlou zpětnou vazbu.

Dále se věnuje praktikám, které jsou potenciálně nebezpečné:

1. Outside-in[7] BDD – podle něj bývají testy vyvíjené tímto způsobem pomalé, křehké (stačí jedna změna a selže mnoho testů), silně provázané a „flaky“ – že střídavě procházejí a neprocházejí. Dále se odkazuje na koncept testové pyramidy[8] od Martina Fowlera [51], přičemž outside-id BDD podle něj mívá zlozvyk tuto pyramidu obracet – tedy závisí především na akceptačních a GUI testech namísto unit testů. Neodsuzuje vysloveně outside-in přístup, ale podle něj je vhodné tento přístup uplatňovat na business vrstvě, nikoliv GUI vrstvě, která se často mění.

2. Mocking – jejich nadužívání podle něj vede k velmi rychlým testům, nicméně takové testy pak svým obsahem postrádají smysl, protože netestují chování, ale implementaci. Pokud dojde ke změně implementace, takové testy začnou selhávat. Zdůrazňuje ale, že nejsou mocky jako mocky, existuje jich několik různých druhů, které pěkně popisuje Uncle Bob ve svém článku „The Little Mocker“ [52]:

   - Dummy – něco, co někam předáváme, ale nezajímá nás, jak se to používá, a od testovaného kódu očekáváme, že se ani používat nebude. V takových případech doporučuje vracet „null“, aby případné použití (které se neočekává) selhalo (v případě Javy na NPE).
   - Stub – inteligentnější Dummy v tom smyslu, že očekává volání a poskytuje přímočarou implementaci, např. vrací fixně daný boolean.
   - Spy – chytřejší Stub, který zároveň monitoruje, zda je správně volán. Může např. počítat počet volání.
   - Mock – rozšířený Spy, který navíc sám ví, jak má být volán a testuje očekávané chování.
   - Fake – speciální objekty, které implementují určité business chování. Spadají do úplně jiné skupiny než předešlé objekty a mohou být velmi komplikované.

   Uncle Bob dodává, že používání těchto fake objektů zavádí do testů vyšší provázanost, přičemž čím více toho fake objekt umí, tím více provázanosti zavádí. Proto ve své praxi nejčastěji používá objekty typu „stub“ či „spy“, „mockům“ se spíše vyhýbá a „fake“ prý nepoužil už přes 30 let.

[7] Outside-in TDD je založeno na testování od nejviditelnějších vrstev aplikace (obvykle *view* vrstva) po nejhlubší vrstvy (obvykle databáze).

[8] Testová pyramida upřednostňuje unit testy před GUI testy.

Matt dále uvádí, že pro správné použití mocků musíme dobře znát Dependency Inversion Principle (DIP) a obecně SOLID[9] principy, které se podle něj dají uplatit jak na objektové, tak i funkcionální programovací paradigma. Tota znalost nám prý pomůže odhalit, kdy je vhodné používat mocky a kdy ne.

3. Unit testy – podle Matta je zlozvykem vytvářet pro každou třídu novou testovou třídu, a každou veřejnou (public) metodu mít otestovanou. To je podle něj problémem např. u návrhových vzorů, které ztělesňují implementační detail. Pokud pro každou veřejnou metodu každé třídy figurující v návrhovém vzoru vytvoříme test, a později se rozhodneme např. pro jiný návrhový vzor, rozbije nám to předchozí testy. Unit testy tedy podle něj nemají testovat všechny jednotky implementace (třídy, metody), ale všechny jednotky chování systému. Takový přístup vede k zúžení veřejného rozhraní, kterého využíváme ve svých testech, a umožňuje někdy později změnit implementaci bez dodatečných úprav testů.

[9]SOLID je akronymem pěti principů v softwarovém inženýrství – Single responsibility principle (SRP), Open-closed principle (OCP), Liskov substitution principle (LSP), Interface segregation principle (ISP) a Dependency inversion principle (DIP).

Příloha B

Flow a TDD

Při své aplikaci TDD jsem se často nacházel ve stavu, který je z pozitivní psychologie znám jako *Flow*.

Flow je takový stav, který se dle knihy Handbook of Posivite Psychology vyznačuje několika vlastnostmi [53]:

- Intenzivní a zaměřenou koncentrací na to, co jedinec aktuálně provádí.
- Splynutím akce a vědomím jedince.
- Ztrátou reflexivního sebe-vědomí (např. ztrátou vnímání sebe sama jakožto sociální bytosti).
- Vědomím, že jedinec má pod kontrolou své činy; to jest vědomí, že jedinec si v principu může poradit se situací, protože ví, jak reagovat na cokoliv, co se stane dál.
- Narušením vnímání toku času (typicky ztráta pojmu o čase, či že uplynulý čas je subjektivně vnímaný jako mnohem kratší než ten, který ve skutečnosti uběhl).
- Prožíváním aktivity jakožto vnitřně hodnotné, např. že často je cíl aktivity vnímaný jen jako záminka pro aktivitu samotnou.

Vyvstává otázka - jak v dosažení Flow pomáhá TDD?

Nakamura a Csikszentmihalyi uvádí dvě nutné podmínky k dosažení Flow [53]:

1. Odpovídající obtížnost úlohy vzhledem k dovednostem člověka – pokud jedinec řeší nějakou úlohu, její obtížnost by měla být škálovatelná dle měnících se schopností řešitele, aby nebyla pro jedince moc jednoduchá, ale zároveň ani moc obtížná.
2. Jasné blízké cíle a okamžitá zpětná vazba k prováděnému postupu.

S oběma těmito podmínkami TDD pomáhá, jelikož velikost testované jednotky si vývojář může škálovat přesně podle svých schopností (pokud narazí na větší problém, než odpovídá jeho schopnostem, může jej rozdělit na menší podproblémy, které již zvládne vyřešit – přičemž tento proces je řízený testy). A vytvořené testy zároveň ztělesňují jasně definované cíle, u kterých lze vždy zjistit, zda jsou splněné či nikoliv, přičemž testy se vytvářejí často a průběžně, což přesně odpovídá druhé zmíněné podmínce Flow. Kent Beck si tento fakt zřejmě uvědomoval, neboť ve své knize opakovaně klade důraz na tzv. „feedback loop“, tedy proces kontinuálně poskytované okamžité zpětné vazby [38].

Proč se vůbec zajímat o flow? Programování, ačkoliv produkuje kód, který lze analyzovat čistě technicky, zůstává lidskou činností, při které je důležité brát v potaz to, že člověk nějak funguje jakožto biologická bytost, má své nedostatky i přednosti, a tyto lidské aspekty stojí za to zkoumat, neboť díky správnému zacházení s nimi lze dosáhnout Flow a tím i o něco lepších výsledků a šťastnějších pracovníků. [54]

Příloha C

Seznam použitých zkratek

ANTLR Another Tool for Language Recognition

API Application programming interface

AST Abstract Syntax Tree

BDD Behavior-driven development

BI Business intelligence

CD Continuous Delivery

CI Continuous Integration

COM Component Object Model

CSV Comma Separated Values

ČVUT České vysoké učení technické v Praze

DHH David Heinemeier Hansson

DI Dependency injection

DIP Dependency inversion principle

EDC Informatica Enterprise Data Catalog

ETL Extract, transform, load

FIT Fakulta informačních technologií

GOOS Growing Object-Oriented Software Guided by Tests

GUI Grafické uživatelské rozhraní

ICT Information and communications technology

IDE Integrated development environment

IDT Information Design Tool

IGC IBM InfoSphere Information Governance Catalog

ISP Interface segregation principle

JDBC Java Database Connectivity

JSON JavaScript Object Notation

JSR Java Specification Request

JVM Java Virtual Machine

LSP Liskov substitution principle

MS Microsoft

MSSQL Microsoft SQL Server

NPE NullPointerException

OCP Open–closed principle

ODBC Open Database Connectivity

OLAP Online Analytical Processing

OOP Object-oriented programming

REST Representational state transfer

SAP BO SAP BusinessObjects

SDK Software development kit

SQL Structured Query Language

SRP Single-responsibility principle

SW Software

TDD Test-Driven Development

UDT Universe Design Tool

URL Uniform Resource Locator

XML Extensible Markup Language

Příloha D

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF