

Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Určování nahraditelnosti a kompatibility webových služeb

Místo této strany bude
zadání práce.

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 17. května 2020

Zdeněk Valeš

Abstract

The text of the abstract (in English). It contains the English translation of the thesis title and a short description of the thesis.

Abstrakt

Text abstraktu (česky). Obsahuje krátkou anotaci (cca 10 řádek) v češtině. Budete ji potřebovat i při vyplňování údajů o bakalářské práci ve STAGu. Český i anglický abstrakt by měly být na stejné stránce a měly by si obsahem co možná nejvíce odpovídat (samozřejmě není možný doslovný překlad!).

Obsah

1	Úvod	8
2	Webové služby	9
2.1	Webová služba	9
2.1.1	Použití webových služeb	11
2.2	Techniky webových služeb	11
2.2.1	HTTP	12
2.2.2	Formáty pro přenos dat	13
2.3	Remote Procedure Call	14
2.3.1	Simple Object Access Protocol	15
2.4	REST	16
2.4.1	Zdroj a jeho reprezentace	17
2.4.2	RESTful služba	18
2.4.3	Verzování REST služeb	19
2.5	Formální popis rozhraní webových služeb	20
2.5.1	WSDL	20
2.5.2	WADL	21
2.5.3	JSON-WSP	22
2.5.4	OpenAPI, RAML	22
3	Datové typy a porovnávání	24
3.1	Datové typy a jejich reprezentace	24
3.1.1	Primitivní datové typy	24
3.1.2	Záznamy	24
3.1.3	Objektové datové typy	25
3.2	Typový systém	26
3.2.1	Primitivní datové typy, záznamy a funkce	27
3.2.2	Objektové typy	27
3.2.3	Rekurzivní typy	27
3.3	Typová kontrola	28
3.3.1	Subtyping	29
3.3.2	Matching na základě subtypingu	32
4	Získávání metadat webových služeb v CRCE	33
4.1	CRCE	33
4.1.1	Metadata komponent	33

4.1.2	Životní cyklus komponenty v CRCE	35
4.2	Indexování webových služeb	35
4.2.1	Obecná indexace komponenty	36
4.2.2	Indexace komponenty s webovou službou	36
4.2.3	Struktura metadat popisující webovou službu	36
4.2.4	Indexování implementace REST služeb	37
4.2.5	Indexování webových služeb na základě popisu	38
4.3	Ověření funkčnosti indexerů a jejich limity	39
4.3.1	Limity indexování webových služeb	39
5	Porovnávání webových služeb	40
5.1	Reprezentace rozdílů a kompatibility	40
5.2	Algoritmus porovnání	41
5.2.1	Popis algoritmu	42
5.2.2	Porovnání datových typů	45
5.2.3	Časová složitost algoritmu	46
5.3	Migrace webových služeb	47
5.3.1	Detekce změn v URL	48
5.3.2	Výběr entit k porovnání s příznakem MOV	49
5.3.3	Verze REST API v cestě k endpointu	50
5.4	Vyhodnocení výsledků	50
5.4.1	Dopad na klienta	51
6	Implementace rozšíření	54
6.1	Rozšíření CRCE	54
6.1.1	Integrace porovnávacího modulu	54
6.2	Porovnávací rozšíření	55
6.2.1	Struktura tříd porovnávačů	55
6.3	Sestavení a nasazení modulu	57
7	Ověření funkčnosti	59
7.1	Spouštění modulu	59
7.2	Použití modulu	60
7.3	Testování	62
7.3.1	REST služba založená na Jersey	64
7.3.2	Služba popsaná JSON-WSP	64
7.3.3	Služba FuelEconomy	65
7.3.4	Služba STAG-Číselníky	66
7.4	Shrnutí testů	67
8	Závěr	68

Literatura	69
Seznam zkratek	72
A WSDL webové služby pro komix Dilbert	73
B Výčet metadat služeb získaných z implementace	77
C Výčet metadat služeb získaných z popisu	79

1 Úvod

Webové služby jsou v dnešní době stále populárnější součástí vývoje aplikací a existuje spousta poskytovatelů, kteří skrze tyto nabízejí možnost řízené manipulace s jejich daty (Facebook, Twitter), integraci aplikací (Trello) nebo výpočetní výkon (AWS, Google). Webové služby také tvoří nedílnou součást vnitro-podnikové a mezi-podnikové komunikace v podobě ERP¹ a EDI² systémů.

Aby propojení služeb fungovalo správně a bylo možné je pohodlně škálovat, je nutná znalost operací nebo zdrojů, poskytovaných službou, a také způsobu, kterým služba komunikuje. Všechny tyto informace jsou popsány v rozhraní poskytnutém službou na jehož správnosti závisí úspěch celé komunikace. Případné změny implementace služby, jež přímo naruší kontrakt daný rozhraním, mohou na straně konzumenta zapříčinit problémy potenciálně vedoucí až k výpadku aplikace, která službu využívá.

V případě použití webových služeb v systémech, jejichž kontext se mění s časem nebo polohou, může být potřeba přepnout mezi různými poskytovateli ať už z důvodu přechodu mezi regiony nebo dočasného výpadku. Aby nedošlo k chybám při tomto přepnutí, je nutné kontrolovat vzájemnou kompatibilitu rozhraní těchto služeb a zaručit, že nebude narušen klientem očekávaný kontrakt.

Systém CRCE vyvíjený skupinou ReliSA na Západočeské Univerzitě je schopen indexace komponent a ukládání takto získaných metadat. V rámci diplomových prací Bc. Hessové a Bc. Pejřimovského vznikly pro systém CRCE rozšiřující moduly, schopné analyzovat různé typy webových služeb a transformovat jejich rozhraní do jednotného datového modelu. Cílem mé práce je na tyto navázat využitím metadat a na základě rozdílů získaných jejich porovnáním rozhodnout o vzájemné kompatibilitě webových služeb, které reprezentují.

V první části textu práce jsou nejprve představeny webové služby a technologie použité k jejich realizaci. Následně je popsán koncept datových typů a způsob vzájemného porovnání. Druhá část práce se zabývá návrhem a implementací porovnávacího algoritmu jako rozšiřujícího modulu pro systém CRCE. Na závěr je ověřena správnost funkce modulu a jeho integrace do CRCE na testovacích datech.

¹Enterprise Resource Planning

²Electronic Data Interchange

2 Webové služby

V této kapitole jsou definovány webové služby a související pojmy. Dále je popsán protokol HTTP, na kterém je primárně postavena komunikace mezi službami a stručně jsou také uvedeny formáty pro přenos dat. V druhé části kapitoly jsou představeny dva hlavní přístupy k realizaci služeb a na závěr je uveden výčet formátů použitých ke strojově čitelnému popisu rozhraní služeb.

2.1 Webová služba

Pojem 'webová služba' může mít odlišné významy [7] a existuje pro něj také spousta definic, z nichž některé jsou více či méně vhodné pro tento text. Například konsorcium W3C¹ definuje pojem webová služba následovně [24]:

Webová služba je softwarový systém navržený pro podporu mezistrojové komunikace po síti. Webová služba má rozhraní, které je popsáno ve strojově čitelném formátu (konkrétně WSDL). Ostatní systémy interagují s webovými službami předepsaným způsobem za použití zpráv protokolu SOAP, které jsou typicky zprostředkované protokolem HTTP s využitím serializace XML a dalších webových standardů.

Tato definice je pro účely mé práce problematická z několika důvodů. Prvním z nich je omezení formátu popisu služby pouze na WSDL. Strojově čitelný popis rozhraní sice snižuje náročnost vytvoření klienta pro danou službu, nicméně WSDL není jediným takovým formátem a netřeba tedy z definice vyloučit služby používající jiné formáty jako například WADL nebo JSON-WSP, jež přímo souvisí s mou prací. Závislost na jednom konkrétním formátu také nemusí být výhodná z hlediska budoucího vývoje, protože nezohledňuje možnost příchodu jiného formátu nebo změny přístupu k webovým službám obecně. WSDL do verze 1.1 například neumožňovalo popis REST² služeb (viz sekce 2.4), ty byly zohledněny až ve verzi 2.0 [17].

Definice dále svazuje implementaci webové služby s použitím konkrétního protokolu pro výměnu dat, v tomto případě SOAP, čímž opět nezahrnuje služby používající jakékoli jiné protokoly. I zde platí nevýhoda takové

¹World Wide Web Consortium

²Representational State Transfer

vazby v kontextu budoucího vývoje, jež byla zmíněna v předchozím odstavci. Webová služba by například přestala být webovou službou, pokud by došlo k změně protokolu pro výměnu dat ze SOAP na JSON-RPC, což není žádoucí.

Posledním důvodem je vymáhání použití XML jako jediného formátu pro přenos dat i přes rostoucí popularitu jiných formátů, jako například JSON³ nebo v poslední době YAML⁴. Není také výjimkou, že webové služby podporují více formátů pro serializaci dat a specifikace jednoho konkrétního formátu v definici tedy není nutná.

Z uvedených důvodů je potřeba zavést vhodnější definici s volnějším kritérii tak, aby webové služby co možná nejméně omezovala. Definici, která služby popisuje primárně skrze užití komunikačního protokolu HTTP ve své knize nabízí R. Daigneau [10]:

Webové služby poskytují prostředky pro integraci různorodých systémů a zpřístupňují znovupoužitelné business funkce pomocí protokolu HTTP. HTTP je použit buď jako jednoduchý komunikační protokol, skrze který jsou přenášena data (např. SOAP/WSDL služby), nebo jako kompletní aplikační protokol, jež definuje sémantiku chování služby (např. RESTful služby).

Tato definice netrpí výše zmíněnými nedostatky a zahrnuje webové služby jimiž se tato práce zabývá. Proto jsem se rozhodl ji využít pro účely mé práce a nebude-li určeno jinak, pojem 'webová služba' v tomto textu odpovídá zmíněné definici. S webovými službami se váže několik dalších pojmů, které jsou v průběhu textu užívány, a pro úplnost jsou zde taktéž definovány:

- **Poskytovatel služby:** server na kterém je služba dostupná, případě organizace, která ji spravuje,
- **Konzument služby:** klient využívající službu, typicky se jedná o aplikaci,
- **Endpoint:** Jedna konkrétní operace služby, případě URI na které je dostupný nějaký zdroj nebo celá služba,
- **Operace:** Operace webové služby, v tomto textu používáno ve stejném významu jako endpoint,
- **Service:** element `service` v popisu WSDL, při překladu do češtiny (služba) by mohl pojem kolidovat s pojmem 'webová služba', proto

³JavaScript Object Notation

⁴YAML Ain't Markup Language

je pro element WSDL označující část služby ponechán anglický tvar `service`,

- **API:** Rozhraní webové služby.

Ve své práci jsem se zabýval dvěma velkými skupinami webových služeb rozdělených podle přístupu k jejich architektuře. Těmi jsou RPC⁵-orientované služby a RESTové služby. Oba přístupy včetně technologií použitých k jejich realizaci jsou rozvedené v následujících sekcích.

2.1.1 Použití webových služeb

Samotný pojem služba z technického pohledu typicky označuje funkcionalitu, jež vykonává nějakou podnikovou operaci, zajišťuje přístup k souborům, nebo obstarává generickou funkcionalitu jako například přihlášení do systému. Webové služby umožňují tyto zpřístupnit různorodým klientům jako jsou například webový prohlížeč nebo mobilní aplikace, nezávisle na technologiích použitých k jejich realizaci.

Ve srovnání s alternativami jako například distribuované objekty jsou webové služby snazší na použití, zejména kvůli jednoduché komunikaci skrze protokol HTTP a standardizované serializaci dat do formátů jako XML nebo JSON. Použitím jednotného komunikačního kanálu (HTTP) je také docíleno oddělení implementace klienta od služby a tím i větší nezávislosti mezi nimi. Díky tomu jsou webové služby snadno propojitelné a lze je tak skládat do větších celků za účelem realizace komplexních procesů [10].

Architektura *Microservices* této skutečnosti využívá a na aplikace nahlíží jako na systém volně propojených služeb, z nichž každá je nezávislá na ostatních a soustředí se na svou přidělenou úlohu. Tou může být například správa uživatelů, nebo analýza dat. Služby v této architektuře se také vyznačují vzájemně nezávislým vývojem [16], což může vést na problémy se vzájemnou kompatibilitou, pokud je například do nové verze služby zanesena změna mutující její rozhraní. Právě tímto tématem se v kontextu webových služeb zabývá má práce.

2.2 Techniky webových služeb

Tato podkapitola krátce představuje nejzákladnější techniky a protokoly určené k realizaci webových služeb. Z hlediska komunikace se jedná o protokol HTML a formáty XML a JSON sloužící k výměně dat a popisu rozhraní služeb.

⁵Remote Procedure Call

2.2.1 HTTP

HTTP⁶ je bezstavový protokol aplikační vrstvy určený pro distribuované hypermediální systémy a tvoří základ informačního systému WWW⁷, který jej využívá od 90. let [5]. Celkem bylo specifikováno pět verzí protokolu: 0.9, 1.0, 1.1, 2.0 a 3.0. Tato sekce se zabývá protokolem HTTP/1.1, neboť právě vlastnosti představené v této verzi jsou relevantní mé práci. Komunikace přes HTTP typicky probíhá skrze TCP/IP spojení na portu 80.

Protokol je založen na principu volání-odpověď. Klient odešle žádost ve formě metody, identifikátoru a verze protokolu, následovanou tělem žádosti v MIME⁸ formátu. Identifikátor žádosti posílané klientem je tvořen URI⁹, což je sekvence znaků, která jasně určuje daný zdroj [6]. Tělo může obsahovat informace o klientovi, modifikátory žádosti a data pro příjemce. Server odpoví statusem, jež zahrnuje verzi protokolu a kód výsledku žádosti (chyba, nebo úspěch). Ten je následovaný zprávou, rovněž ve formátu MIME, obsahující informace o serveru a případně tělo odpovědi s daty pro klienta [5].

HTTP definuje množinu metod, jejichž účelem je specifikovat konkrétní operaci, která by měla být provedená nad daným zdrojem, identifikovaným volaným URI. Výčet metod společně s popisem je uveden v tabulce 2.1.

Název	Popis užití	Idempotentní
GET	Získání informace náležící volanému URI	ANO
HEAD	Identická GET, ale odpověď neobsahuje tělo	ANO
PUT	Vytvoření nové, nebo aktualizace existující entity odpovídající volanému URI	ANO
DELETE	Smazání zdroje identifikovaným volaným URI	ANO
OPTIONS	Získání informací o možnostech komunikace	ANO
TRACE	Příjemce odešle zpět tělo přijaté zprávy	ANO
POST	Vytvoření nových dat vztažených ke zdroji identifikovaným volaným URI	NE
CONNECT	Rezervovaný název	NE

Tabulka 2.1: Metody definované v HTTP protokolu

Implementace obsluhy metod sice záleží na konkrétním poskytovateli, nicméně HTTP definuje tzv. bezpečné metody, jež by měly označovat pouze

⁶Hypertext Transfer Protocol

⁷World Wide Web

⁸Multipurpose Internet Mail Extensions

⁹Uniform Resource Identifier

operace určené k získání dat. Jedná se o metody GET a HEAD. Protokol u metod také zavádí vlastnost idempotence, pokud má $N > 0$ identických volání stejně následky jako pouhé volání jedno [5].

2.2.2 Formáty pro přenos dat

Služby komunikující protokolem HTTP si obvykle předávají data v textové podobě. Způsobů, jak taková data formátovat existuje mnoho, nicméně během mé práce jsem přišel do styku převážně s formáty XML a JSON, proto se zde zaměřím právě na ně. Zmíněné syntaxe nemusí být použity pouze k serializaci předávaných dat, ale také k popisu rozhraní samotné služby, jak bude ukázáno v části 2.5.

XML

Extensible markup language (dále jen XML) je značkovací jazyk určený k reprezentaci dokumentů ve formátu, který je čitelný pro stroj i člověka. První verze specifikace¹⁰ byla zveřejněna roku 1998 konsorciem W3C, které tento formát vyvíjí a spravuje. MIME typ pro označení XML je `application/xml`.

Základními stavebními kameny jazyka XML jsou dokument, tagy, elementy a atributy. Dokument je definován jako datový objekt, jež se skládá z prologu označujícího verzi XML, jednoho kořenového elementu a případných dodatků. Všechny zmíněné části dokumentu musí odpovídat syntaxi jazyka XML. Element je logická jednotka dokumentu ohraničená tagy. Může být prázdný, nebo může vnořovat další elementy. Atributy jsou páry klíč-hodnota přiřazené elementu.

Příklad XML dokumentu je znázorněn na obrázku 2.1. Kořenovým elementem je v tomto případě `quiz`, jež je ohraničený tagem `<quiz>`. Jeho vnořeným elementem je `qanda` obsahující atribut `seq` s hodnotou 1.

```
1  <?xml version="1.0"?>
2  <quiz>
3    <qanda seq="1">
4      <question>
5        Who was the forty-second president of the U.S.A.?
6      </question>
7      <answer>
8        William Jefferson Clinton
9      </answer>
10   </qanda>
11 </quiz>
```

Obrázek 2.1: Příklad XML dokumentu

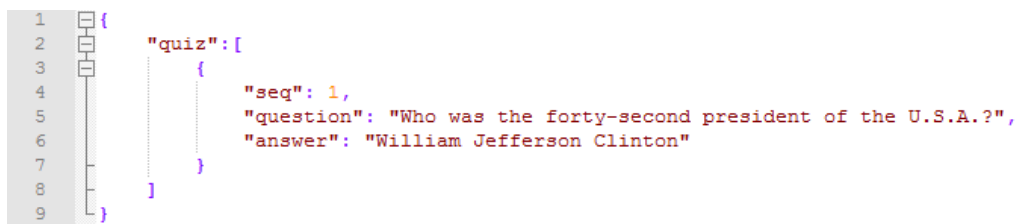
¹⁰<https://www.w3.org/TR/REC-xml/>

JSON

JavaScript object notation (dále jen JSON) je formát výměny dat vzešlý z části standardu jazyka JavaScript¹¹ publikovaného v roce 1999. Samotný JSON byl poprvé standardizován specifikací ECMA-404¹² v roce 2013. Stejně jako zmíněný XML, je i tento čitelný pro člověka a stroj. MIME typ pro označení JSON je `application/json`.

JSON je založen na dvou základních strukturách, jež by měly být podporovány většinou dnešních moderních programovacích jazyků. Těmito strukturami jsou kolekce párů klíč-hodnota zvaná *objekt* a seřazený seznam hodnot zvaný *pole*. Objekt je ohraničen složenými závorkami `{, }` a obsahuje neseřazený výčet párů klíč-hodnota. Klíč je od hodnoty oddělen dvojtečkou a celé páry jsou oddělené čárkami. Pole je ohraničeno hranatými závorkami `[,]` a obsahuje seřazený výčet hodnot, jež jsou vzájemně odděleny čárkou.

Obrázek 2.2 obsahuje příklad JSON objektu analogickému ke XML dokumentu na obrázku 2.1. Kořenový objekt obsahuje klíč `kviz`, ke kterému jako hodnota náleží pole otázek. Otázka je reprezentována objektem s hodnotami pro klíče `seq`, `question` a `answer`.



Obrázek 2.2: Příklad JSON objektu

2.3 Remote Procedure Call

Pojem Remote Procedure Call (dále jen RCP) obecně označuje technologii umožňující předání řízení mezi programy v různých adresních prostorech, které jsou mezi sebou propojeny komunikačním kanálem [18]. Tato sekce se zaměřuje na popis RPC zejména ve vztahu k webovým službám a komunikaci s nimi.

Obecně je RPC postaveno na principu volání-odpověď. Klient voláním předá poskytovateli informace o operaci, jež má vykonat, včetně vstupních dat. Příjemce (server) tuto operaci provede a v odpovědi vrátí zpracovaná

¹¹<https://www.ecma-international.org/publications/standards/Ecma-262-arch.htm>

¹²<https://www.ecma-international.org/publications/standards/Ecma-404-arch.htm>

data. Typickým příkladem použití je přesunutí náročného výpočtu z méně výkonného klienta na výkonný server.

V kontextu webových služeb se za 'RPC-like' považují služby založené na definici konsorcia W3C, tedy služby popsané WSDL předávající si data protokolem SOAP v těle HTTP volání a odpovědi. Klient takovou službu vnímá jako množinu operací které může volat, z nichž každá vykonává určitou úlohu [30]. Aby toto bylo možné, musí každá služba definovat jasné rozhraní pomocí kterého klient přistupuje k operacím a zpracovává vrácená data. Příkladem takového rozhraní je popis služby ve formátu WSDL.

Protože součástí mé práce bylo zpracování metadat služeb popsaných formátem WSDL, jež typicky používají protokol SOAP pro výměnu dat, zabývají se další odstavce právě tímto protokolem. Popisný formát WSDL je detailněji rozveden na konci této kapitoly.

2.3.1 Simple Object Access Protocol

Protokol SOAP, současně ve verzi 1.2, je určený pro výměnu strukturovaných dat v decentralizovaném, distribuovaném prostředí. Specifikaci protokolu [23], z níž tato sekce převážně čerpá, spravuje konsorcium W3C. Mezi cíle návrhu protokolu patří zejména jeho nezávislost na konkrétní implementační platformě a komunikačním protokolu [23]. Specifikace nijak neomezuje formát, ve kterém jsou data předávána, nicméně primárním způsobem pro jejich popis je XML.

Example 1: SOAP message containing a SOAP header block and a SOAP body

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
  <env:Header>
    <n:alertcontrol xmlns:n="http://example.org/alertcontrol">
      <n:priority>1</n:priority>
      <n:expires>2001-06-22T14:00:00-05:00</n:expires>
    </n:alertcontrol>
  </env:Header>
  <env:Body>
    <m:alert xmlns:m="http://example.org/alert">
      <m:msg>Pick up Mary at school at 2pm</m:msg>
    </m:alert>
  </env:Body>
</env:Envelope>
```

Obrázek 2.3: Příklad SOAP zprávy (převzato ze specifikace)

Základní komunikační jednotkou v SOAP je zpráva, která je předávána mezi komunikujícími uzly. Obsah zprávy, který je zapouzdřený v takzvané obálce, se skládá z hlaviček a těla zprávy. Obrázek 2.3, převzatý ze specifikace protokolu, znázorňuje příklad SOAP zprávy. V návaznosti na komunikační vzor RPC jsou SOAP zprávy mezi službami typicky používány k předání

detailů o proceduře, jež má být vykonána, včetně vstupních parametrů. Výsledky těchto operací jsou vráceny v obdobném formátu.

Použití protokolu SOAP jako formátu dat nad konkrétním komunikačním protokolem je definováno souborem pravidel, jež se nazývá SOAP Protocol Binding Framework. Tato pravidla popisují především detaily použití komunikačního protokolu pro předávání SOAP zpráv, dodržení kontraktu a zpracování případných chyb. V kontextu mé práce je důležité zejména napojení SOAP na HTTP u služeb popsanych WSDL dokumentem.

SOAP zprávy jsou skrze HTTP, vzhledem ke svému rozsahu, primárně přenášeny metodou POST a v rámci této práce je předpokládáno, že webové služby používající SOAP akceptují volání pouze s metodou POST. V dokumentu WSDL je propojení SOAP s HTTP definováno v elementu `<binding>` a příklad takového propojení je uveden na obrázku 2.4.

```
<binding type="glossaryTerms" name="b1">
  <soap:binding style="document"
    transport="http://schemas.xmlsoap.org/soap/http" />
  <operation>
    <soap:operation soapAction="http://example.com/getTerm"/>
    <input><soap:body use="literal"/></input>
    <output><soap:body use="literal"/></output>
  </operation>
</binding>
```

Obrázek 2.4: Propojení SOAP s HTTP v dokumentu WSDL

2.4 REST

Representational State Transfer (dále jen REST) je architektonický styl tvorby webových služeb, který ve své disertační práci [12] popsal R. T. Fielding. Služby tvořené v tomto stylu se od služeb definovaných konsorciem W3C liší zejména využitím protokolu HTTP pro sémantiku aplikace a vyznačuje se také absencí stavu.

R.T Fielding definoval styl REST pomocí množiny omezení, jež jsou aplikována na systém vzájemně komunikujících komponent, což jsou v rámci této práce webové služby. Vybraná omezení jsou popsána v následujících odstavcích.

Klient-server

Důvodem pro toto omezení je princip oddělení zodpovědnosti klienta (např. uživatelského rozhraní) od serveru, čímž je umožněn nezávislý vývoj obou stran.

Absence stavu

Omezení se týká zejména volání služby, jež musí obsahovat vše nutné pro jeho zpracování a není možné využít kontext uložený na straně serveru. Ten je tedy uložený na straně klienta a je proto nutná kontrola jeho konzistence při zpracování volání. Mezi výhody plynoucí z tohoto omezení patří zlepšení spolehlivosti a přehlednosti komunikace.

Uniformní rozhraní

Komponenty vzájemně komunikují přes jednotné rozhraní, které odděluje jejich funkcionalitu od implementace. Rozhraní REST komponenty se vyznačuje čtyřmi vlastnostmi: identifikace zdrojů, manipulace jimi skrze reprezentace, sebe-popisující zprávy a hypermediální řízení (viz sekce 2.4.2).

Vrstvený styl

Komponenty jsou rozděleny do vrstev a každá z nich komunikuje pouze s komponentami v sousedních vrstvách. Aplikací tohoto modelu by mělo být dosaženo omezení komplexity a větší nezávislosti jednotlivých vrstev, respektive komponent v nich.

2.4.1 Zdroj a jeho reprezentace

Na rozdíl od služeb vycházejících z RPC, jež jsou koncipovány jako množiny různých operací, jsou REST služby orientovány kolem zdrojů a manipulaci s jejich reprezentacemi. Architektura vycházející z toho stylu se nazývá ROA¹³ a ve své publikaci [22] ji popsal L. Richardson. Tato sekce se zabývá významem zdroje a jeho reprezentací v REST.

Zdroj (anglicky *resource*) je klíčovou abstrakcí informace v REST a může představovat jakoukoliv pojmenovatelnou informaci jako například dokument, obrázek, webovou službu, nebo kolekci jiných zdrojů. R. T. Fielding ve své práci [12] definuje zdroj *R* jako:

¹³Resource Oriented Architecture

funkci $M_R(t)$, která pro čas t vrací množinu ekvivalentních entit. Tato množina může být prázdná, nebo obsahovat reprezentace daného *resource*, případně jeho identifikátory

Pojem zdroj tedy neoznačuje konkrétní objekt nebo data, ale abstraktní koncept, který je konkrétním objektem nebo daty pouze reprezentován. Jako příklad lze uvést systém pro správu verzí, kde identifikátor 'poslední verze' označuje zdroj a samotná verze (0.8, 0.9, ...) jeho reprezentaci, která se s časem mění. Reprezentace různých zdrojů se mohou shodovat. V návaznosti na předchozí příklad by zdroje identifikované 'poslední verze' a 'verze 0.8' mohly být v určitý čas t reprezentovány stejnou verzí '0.8'.

Každý zdroj má alespoň jeden identifikátor, v případě Webu se jedná o URI. Tento identifikátor by měl být co nejvíce popisný a v rámci jedné služby dodržovat stejný styl [22].

S reprezentacemi zdrojů je manipulováno HTTP voláním jejich identifikátorů (tedy URI) s nastavenou HTTP metodou, jež byly popsány v části 2.2.

2.4.2 RESTful služba

Jedním z primárních účelů webových služeb založených na stylu REST je manipulace s XML reprezentacemi zdrojů pomocí jednotné množiny bezstavových operací, jimiž typicky jsou *create*, *retrieve*, *update* a *delete* [24]. Míra, kterou jednotlivé služby dodržují styl REST se různí a pojem 'RESTful' označuje ty, jež se REST drží nejvíce. Model zabývající se 'RESTful vyspělostí' dané služby byl představen L. Richardsonem a nazývá se Richardson Maturity Model. M. Fowler jej vysvětluje ve svém online článku [13], který je hlavním zdrojem informací pro následující odstavce. Model, uvedený na obrázku 2.5, definuje čtyři níže popsané úrovně a pokud tyto služba splňuje, je označena jako RESTful (plně RESTová).

0. úroveň

Služby v 0. úrovni se vyznačují použitím protokolu HTTP pro komunikaci, ale implementují vlastní sémantiku interakce, jež je typicky podobná stylu RPC. Služba je tedy například tvořena endpointem *bookingService* a na tento jsou směřována všechna volání obsahující informace o tom, co má služba vykonat.



Obrázek 2.5: Ilustrace Richardsonova modelu vyspělosti

Zdroje

1. úroveň zavádí do služby zdroje, tak jak byly popsány v sekci 2.4.1. Místo volání jednoho univerzálního endpointu služby jsou nyní volány endpointy jednotlivých zdrojů.

HTTP metody

Předchozí úrovně používají HTTP metody nezávisle na jejich účelu. Endpoint každého zdroje z předchozí úrovně by tedy například očekával volání POST i když by pouze vracel data. 2. úroveň tento přístup mění a služby používají HTTP metody tak, jak jsou v HTTP definované. Volání GET endpointu zdroje tedy vrací data, zatímco POST zdroj vytváří, nebo mění.

Hypermediální řízení

Poslední úroveň zavádí princip HATEOAS¹⁴, ve kterém je klientovi s reprezentacemi zdrojů navíc předána informace o další možné manipulaci s nimi. Tím je docíleno většího oddělení klienta od poskytovatele, neboť server může mimo jiné měnit část schématu URI zdrojů a nenarušit tím schéma používané klientem.

2.4.3 Verzování REST služeb

Webové služby dodržující poslední úroveň Richardsonova modelu by za předpokladu statických vstupních bodů nemusely svou verzi vystavovat. Nicméně existuje nezanedbatelná část, jež na tuto úroveň nedosahují a tedy udržují statické rozhraní pro každou verzi, protože jakákoliv změna v schématu URI

¹⁴Hypertext As The Engine Of Application State

se negativně projeví na straně klienta. Ten totiž díky absenci HATEOAS nemůže tuto změnu dynamicky nijak zjistit je proto odkázaný na rozhraní služby.

Může tedy nastat případ kdy poskytovatel v zájmu udržení zpětné kompatibility udržuje více verzí jedné služby současně. Existují různé způsoby jak vystavit konkrétní verzi, jedním z nich je uvedení verze v URI zdroje. Jedná se o běžnou a snadno implementovatelnou praktiku [29] [22]. Dopady použití této praktiky na mou práci jsou rozebrány v částech 4.2 a 5.3.3.

2.5 Formální popis rozhraní webových služeb

Aby bylo možné se službami komunikovat, je nutné znát jimi poskytované operace, zdroje a akceptovaný formát dat. Tyto informace jsou zahrnuté v rozhraní služby, jež může být popsáno různými způsoby s proměnlivou úrovní detailu. Služby popsané v části 2.3 takové rozhraní automaticky poskytují ve strojově čitelné podobě, čímž mimo jiné umožňují automaticky generovat klienta a kostru serverové aplikace. V případě REST služeb (část 2.4) je situace složitější, protože univerzální standardizovaná podoba rozhraní neexistuje a to pak může mít podobu například Word dokumentu, nebo emailové konverzace. Výsledkem snah řešit problém se standardizací popisu rozhraní REST služeb jsou například strojově čitelné formáty Swagger¹⁵, OpenApi¹⁶ nebo RAML¹⁷.

Jak bude popsáno v kapitole 4, systém CRCE, pro který byl vyvíjen rozšiřující modul, jež je předmětem této práce, je v současnosti schopný indexovat webové služby pouze na základě implementace nebo popisu JSON-WSP, WADL a WSDL.

2.5.1 WSDL

Formát WSDL je využíván převážně pro popis webových služeb využívajících protokol SOAP k výměně dat, nicméně od verze 2.0 lze popsat i REST služby. Formát současně existuje ve verzích 1.1 a 2.0. Z důvodů stáří verze 1.1, jež byla publikována v roce 2001, je tento popis zaměřen především na verzi 2.0.

¹⁵<https://swagger.io/>

¹⁶<https://www.openapis.org/>

¹⁷<https://raml.org/>

WSDL je formát založený na XML popisující webové služby jako množiny endpointů operujících nad zprávami (*messages*), jež obsahují buď dokumentově, nebo procedurálně orientované informace. Abstraktně popsané operace a zprávy jsou spojeny s konkrétním síťovým protokolem a formátem zpráv, čímž je definován endpoint. Související endpointy jsou sdruženy do *services* [27]. Tabulka 2.2 obsahuje popis vybraných elementů použitých ve formátu WSDL 2.0.

Název	Popis
service	Kolekce endpointů, na kterých je služba dostupná
endpoint	Detaily endpointu, na kterém je služba dostupná
message	Definice dat, jež jsou předmětem komunikace
portType	Množina operací služby
operation	Implementační detaily konkrétní operace
binding	Implementační detaily nezbytné k přístupu ke službě
types	Definice vlastních typů

Tabulka 2.2: Výběr elementů pro formát WSDL 2.0

2.5.2 WADL

WADL je strojově čitelný formát, jehož hlavním účelem je popis webových služeb založených na HTTP (například REST). Formát vznikl jako pokus o standardizaci popisu webových služeb, které jsou typicky popsány nestandardní, případě vzájemně odlišnou textovou dokumentací [25].

Formát je založený na XML a popisuje webové služby jako kolekce zdrojů, nad kterými definuje operace (*methods*). Zdroje *resource* je možné vnořovat a každý zdroj je identifikován URI sestaveným podle předem daného vzoru. Tabulka 2.3 obsahuje popis několika vybraných elementů, použitých k popisu webové služby.

Název	Popis
resources	Kolekce zdrojů poskytovaných službou
resource	Popis konkrétního zdroje
method	Popis operace, která může být nad zdrojem provedena

Tabulka 2.3: Výběr elementů pro formát WADL

2.5.3 JSON-WSP

Formát JSON-WSP je založený na JSON a vychází z JSON-RPC, což je protokol pro vzdálené volání procedur. JSON-WSP popisuje rozhraní webových aplikací a i když měl původně nahradit JSON-RPC, je zatím ve stavu neformálního návrhu a není nijak standardizovaný ani neexistuje RFC s jeho specifikací nebo JSON schéma. Neoficiální specifikace je udržována na Wikipedii [28].

JSON-WSP definuje službu pomocí objektů *description*, *request*, *response* a *fault*. Objekt *description* popisuje rozhraní služby, obsahuje mimo jiné název služby, URL služby a kolekci metod, které služba dokáže vykonat. Objekty *request*, *response* a *fault* definují podobu dat, které služba přijímá, respektive vrací. Tabulka 2.4 uvádí výběr objektů definovaných v *description* relevantních pro mou práci.

Název	Popis
methods	Kolekce metod vykonávaných službou
method	Popis konkrétní metody včetně parametrů a návratového typu
types	Definice vlastních typů

Tabulka 2.4: Výběr elementů pro formát WADL

2.5.4 OpenAPI, RAML

Indexování REST služeb na základě jejich popisu ve formátu OpenAPI nebo RAML je jedním z hlavních témat budoucích rozšíření CRCE, které se této práci přímo dotýká. Proto jsou zde tyto formáty představeny, i když není v současné době implementována jejich analýza.

RAML¹⁸ je popisný formát zaměřený na dokumentaci RESTful služeb. První návrh vznikl roku 2013 a v současné době jej spravuje RAML Workgroup. OAS¹⁹ je specifikace dokumentace rozhraní webových služeb. Specifikace vychází z formátu Swagger²⁰ a je spravována konsorciem OpenAPI Initiative.

Oba formáty jsou čitelné pro člověka i stroj a v jejich návrhu je kladen důraz na samo-dokumentovatelnost. Existují nástroje pro generování klientů a koster serverů na základě popisného dokumentu. Syntaxe obou formátů vychází z YAML a strukturálně je velmi podobná. Nejprve je definován obecný popis API (url, verze, název, vlastní datové typy, ...), pak

¹⁸RESTful API Modeling Language

¹⁹OpenApi Specification

²⁰<https://swagger.io/>

následuje hierarchický popis zdrojů pomocí jejich URI, včetně popisu volání (např. parametry) a struktury odpovědi.

3 Datové typy a porovnávání

Tato kapitola rozebírá vlastnosti datových typů programovacích jazyků, se zaměřením na jejich vzájemné porovnání. V první části kapitoly jsou představeny datové typy obecně, včetně jejich rozdělení v imperativních a objektových jazycích. Druhá část kapitoly se zabývá typovými systémy, způsoby kontroly typů a porovnáním datových typů. Vzhledem ke kontextu práce je text kapitoly zaměřen na staticky typované jazyky a typové systémy jazyků Java a XSD.

3.1 Datové typy a jejich reprezentace

Datový typ představuje množinu hodnot, kterých může nabýt proměnná, konstanta, funkce, nebo jiný výraz [15]. O programovacích jazycích mluvíme jako o explicitně typovaných, pokud je název datového typu součástí syntaxe. Je-li navíc typová kontrola prováděna v čase překladu nazývají se tyto staticky typované [4]. Takovými jazyky jsou například C nebo Java.

Datové typy lze rozdělit do tří hlavních skupin, kterými jsou primitivní, záznamové a objektové. Jednotlivé skupiny se od sebe liší především komplexností své struktury (popsána v následujících odstavcích) a z toho plynoucí složitostí způsobu porovnání (popsán v části 3.3).

3.1.1 Primitivní datové typy

Primitivní typy jsou základními typy jazyka, jejichž účelem je uložení konkrétních hodnot (znak, číslo, ...). Typy mají pevně danou délku, nedefinují vlastní metody (na rozdíl od objektů) a neexistuje zde ani koncept dědičnosti. Jazyk obvykle definuje základní aritmetické operace pouze pro primitivní typy, nicméně výjimku tvoří např. C++, kde je možné operátory přetížit i pro uživatelsky definované typy. Java verze 8 podporuje celkem 8 primitivních datových typů, jejichž výčet se nachází v tabulce 3.1.

3.1.2 Záznamy

Datový typ záznam (anglicky *record*) je kolekcí položek různých datových typů s typicky pevně daným pořadím [11]. Příkladem záznamu v jazycích je například *struct* v C nebo *record* v Pascalu. Jazyk Java datový typ záznam nedefinuje.

Název datového typu	Rozsah hodnot
byte	$\langle -128; 127 \rangle$
short	$\langle -32768; 32767 \rangle$
int	$\langle -2^{31}; 2^{31} - 1 \rangle$
long	$\langle -2^{63}; 2^{63} - 1 \rangle$
float	32-bit IEEE 754
double	64-bit IEEE 754
boolean	true, false
char	16-bit Unicode znaky

Tabulka 3.1: Primitivní datové typy jazyka Java

Jako na záznam se také dá nahlížet na vlastní datový typ definovaný v XSD elementem *complexType*, jež představuje pojmenovaný výčet položek s různými datovými typy. Pro srovnání je na obrázku 3.1 uveden příklad záznamu *personinfo* definovaného pomocí *complexType* v XSD a *struct* v C.

<pre> 1 <xs:complexType name="personinfo"> 2 <xs:sequence> 3 <xs:element name="firstname" type="xs:string"/> 4 <xs:element name="lastname" type="xs:string"/> 5 <xs:element name="age" type="xs:int"/> 6 </xs:sequence> 7 </xs:complexType> </pre>	<pre> 1 struct PersonInfo { 2 char firstname[50]; 3 char lastname[50]; 4 int age; 5 }; </pre>
--	---

Obrázek 3.1: Záznam v XSD a C

XSD umožňuje uživatelsky definované typy rozšiřovat elementem *extension* a vytvářet tak komplexní struktury, nicméně i ty lze rozvinout do podoby záznamu. Uživatelský typ v XSD také může být rekurzivně definovaný. Metody práce s rekurzivními typy jsou popsány v sekcích 3.2 a 3.3.

3.1.3 Objektové datové typy

Objektové typy se skládají ze stavu a metod, které reprezentují chování objektu. Stav je reprezentován kolekcí atributů různých datových typů, ke kterým je přístupováno metodami [19]. Účelem objektů je typicky zapouzdření určité části dat s logicky související funkcionalitou. Příkladem může být třída *Obdelnik*, jejíž stav tvoří rozměry *a*, *b* a chování je reprezentováno metodami pro výpočet obsahu a obvodu.

Třída (*class*) obsahuje definice všech atributů a metod, jež objekt používá. Objekt, vytvořený například klíčovým slovem **new**, je instancí třídy. Objektově orientované jazyky, včetně jazyka Java, podporují dědičnost tříd,

což umožňuje sdílet společný stav a chování mezi více třídami. Dědičnost a její dopady na porovnávání datových typů jsou rozvedeny v části 3.2.

Speciálními případy dědičnosti jsou v mnoha objektových jazycích také rozhraní a abstraktní třídy. Rozhraní (*interface*) obsahuje pouze popis funkcionality v podobě deklarace metod a jako takové jej nelze instanciovat, ale pouze implementovat (případně oddědit v jiném rozhraní). Pouhou deklarací metod je vytvořen kontrakt, který musí implementující třídy dodržet. Tento přístup oddělující popis funkcionality od samotné implementace je využíván například v komponentové architektuře. Abstraktní třídy též nelze instanciovat, nicméně mohou nést třídní proměnné a část implementace. Tyto třídy typicky deklarují abstraktní metody, které, podobně jako rozhraní, pouze popisují chování a konkrétní logika je obsažena v implementující třídě.

3.2 Typový systém

Fundamentální součástí většiny jazyků je metoda pro kontrolu typové konzistence programů. Tato metoda se nazývá typový systém a B.C. Pierce ji v [21] definuje takto:

Typový systém je technicky realizovatelný, na syntaxi založený způsob jak dokázat, že program nevykazuje určité chyby chování, klasifikováním jeho syntaktických konstrukcí podle druhů hodnot, které představují.

Typový systém tedy poskytuje formální pravidla pro kontrolu typů, jež jsou nezávislá na konkrétních algoritmech použitých k jejich vymáhání. Primárním účelem takového systému je zabránění výskytu tzv. *execution errors* za běhu programu, což je množina chyb, které by mohly vést k jeho okamžitému přerušení nebo poškození dat [4]. Tyto chyby se dělí na *trapped*, které způsobí okamžité ukončení programu, a *untrapped*, které samy o sobě chod programu nezastaví, nicméně mohou vést na nepredikovatelné chování jehož výsledkem může být ukončení chodu programu. Příkladem *untrapped* chyby (v jazyce C) je přístup na legální adresu a následné přečtení příliš velkého množství dat v důsledku špatného typování. Následek takové chyby může být provádění výpočtu nad neplatnými daty.

Jednou z forem popisu typů a pravidel typového systému je typovaný lambda kalkul, případně objektový kalkul představený v [2]. Základem obou je tzv. typovací relace, jež vázané proměnné x ve výrazu $\lambda x : T_1. t$ přiřazuje typ T_1 . Následující sekce uvádí popis datových typů zmíněnými nástroji.

3.2.1 Primitivní datové typy, záznamy a funkce

Jak již bylo zmíněno, základem kalkulů používaných pro definici pravidel typového systému je typovací relace. V případě primitivních datových typů nelze typ rozložit na atributy a funkce, jako je tomu např. u objektových typů, ale lze jej definovat pomocí typových pravidel. Například typ $T := Bool$ by mohl být definován pravidly $true : Bool$, $false : Bool$, které definují hodnoty datového typu a pravidlem 3.1 definujícím výsledný typ ternárního operátoru *if-then-else*.

$$t_1 : Bool, t_2 : T, t_3 : T \vdash (\text{if } t_1 \text{ then } t_2 \text{ else } t_3) : T \quad (3.1)$$

Na záznamy lze nahlížet jako na kolekci typovaných položek a záznam $R = \{r_1 : R_1 \dots r_n : R_n\}$ je tedy definován jako množina položek r_i typovaných na R_i .

Datové typy funkcí jsou typicky popsány jako relace mezi datovým typem argumentu a datovým typem výsledku. Například funkce $next(n_1)$ jež vrací následovníka celého čísla n_1 by byla zapsána jako $next : int \rightarrow int$. Lambda kalkulem by stejná funkce byla zapsána jako $\lambda x : int. t_2 : int \rightarrow int$.

3.2.2 Objektové typy

Jak již bylo zmíněno v sekci 3.1, na objekty lze nahlížet jako na kolekci proměnných a funkcí a ve výše zmíněných kalkulech jsou i takto popisovány. Jednou z vlastností objektových typů je sebe-reference, jež objektu umožňuje přistupovat k sobě samotnému (klíčové slovo **self**, případně **this**) a k jeho rodiči (**super**, **base**). Této vlastnosti lze využít i k popisu typu objektovým kalkulem, kde výraz $\varsigma(X)$ označuje *self* parametr X . Následující příklad definuje objekt o s metodou m_1 a tělem metody b :

$$o = [m_1 = \varsigma(x)b]$$

Volání metody nad objektem zapsané jako $o.m_1$ by za *self* parametr x dosadilo instanci objektu o . Použití *self* parametru k porovnání typů je dále rozvedeno v sekci 3.3.

3.2.3 Rekurzivní typy

Rekurzivní datový typ je takový, který ke své definici používá sám sebe, například v typu proměnné nebo argumentu funkce. Typickým příkladem takového typu je neměnný (*immutable*) seznam čísel **NumList**, tvořený poli **first: int**, **rest: NumList** a objektovým kalkulem zapsaný jako:

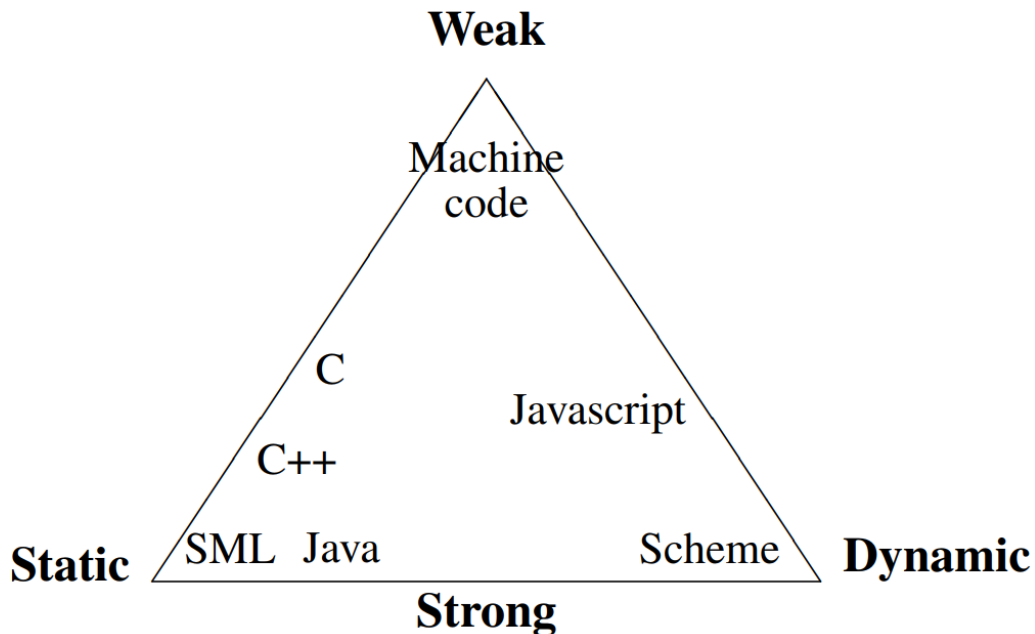
$$NumList \triangleq \mu(X)[first : int, rest : NumList]$$

Během typové kontroly (viz část 3.3) je nutné typy rozvinout, aby mohlo dojít k jejich porovnání. Tento rozvoj však v případě rekurzivních typů vede na nekonečný strom, což představuje problém. B.C. Pierce zminňuje [21] dva přístupy k řešení tohoto problému, založené na odlišnosti definice typu a prvního kroku jejího rozvoje, kterými jsou *equi-recursive* a *iso-recursive*. První z nich se na definici typu a první krok jeho rozvoje dívá jako na ekvivalentní, protože představují "stejně" nekonečné podstromy. Druhý tyto vnímá jako definice odlišných typů, mezi kterými lze přecházet operacemi `unfold()` a `fold()`. Přístup *equi-recursive* je více běžný v objektových jazycích.

3.3 Typová kontrola

Tato podkapitola shrnuje základní informace o typové kontrole a způsoby, kterými může být provedena. Text čerpá především z publikace [21] od B.C. Pierce.

Z pohledu doby, kdy ke kontrole typů dojde, rozlišujeme statickou a dynamickou typovou kontrolu. Podle striktnosti pravidel aplikovaných při typové kontrole také rozlišujeme silné a statické typování. Tyto termíny jsou popsány v následujících odstavcích a s příklady jazyků znázorněny na obrázku 3.2, jež byl převzat z [3].



Obrázek 3.2: Příklady jazyků rozdělených podle přístupu k typové kontrole

Statická typová kontrola probíhá v čase překladač a konkrétní datové typy proměnných jsou známy ještě před spuštěním programu. V případě detekce typovací chyby dojde k přerušení překladač a sestavení programu, čímž je zabráněno případným chybám, které by mohly vést k pádu programu za běhu.

Dynamická typová kontrola probíhá za běhu programu a provádí ji například interpret jazyka. Oproti statické typové kontrole umožňuje tato spuštění programů s chybami v typech (například `int a = 1 + "abc"`), jež se mohou projevit zastavením nebo pádem programu.

Silné typování v případě volání operace jazyk zaručuje, že bude volána se správně typovanými argumenty [3]. Definuje-li například jazyk operaci sčítání operátorem `+`, pak silně typovaný jazyk zaručí, že fragment `a + b` půjde spustit pouze pokud `a` a `b` jsou správně typované.

Slabé typování v případě volání operace jazyk nezaručuje, že operace prováděná nad argumenty bude z hlediska jejich typu dávat smysl. Výraz `"5" * "5"` by byl v silně typovaném jazyce vyhodnocen jako chyba, nicméně slabě typovaný jazyk, jako například JavaScript, tento výraz vyhodnotí, protože oba řetězce lze konvertovat na čísla a následně vynásobit [3].

Z uvedeného obrázku je vidět, že popsané rozdělení má spektrální charakter a jazyky typicky kombinují více přístupů. Důvodem je řešení typové kontroly v případě přístupu k polím, meta-programování, *downcastingu*, *reflexi* atp.

K určení vztahu mezi ne-primitivními typy, zejména v objektových jazycích, je užíváno relací *subtypingu* a *matchingu*. Tyto dvě relace, jež jsou popsány v následujících sekcích, měly původně tvořit základ pro porovnávání datových typů v mé práci. Z důvodů popsaných na konci kapitoly 4 bylo však toto porovnávání omezeno na minimum a je tedy předmětem budoucích prací.

3.3.1 Subtyping

Subtyping je binární relace definovaná nad datovými typy. Zápis $A <: B$ čteme jako “*A* je sub-typ *B*” což znamená, že typ *A* může být použit kdekoli tam, kde je očekáván typ *B*. Například *java.lang.Long* je sub-typ *java.lang.Number*, protože proměnnou typovanou na *Long* lze použít kdekoli, kde je očekávána proměnná typu *Number*. Relace *subtypingu* je reflexivní ($A <: A$) a tranzitivní ($A <: B, B <: C \vdash A <: C$).

Následující odstavce popisují způsob konstrukce této relace pro jednotlivé skupiny datových typů, jež byly uvedené v 3.1. Informace a postupy vycházejí především z publikací [21] a [1].

Primitivní typy

V případě primitivních typů subtyping označuje schopnost jednoho typu pojmout jiný. Tedy například primitivní typ *double* je schopný reprezentovat všechna čísla z *float* (ale ne naopak) a proto platí *float* <: *double*. V případě datových typů pro celá čísla v Jave pak platí následující:

$$byte <: short <: int <: long$$

Záznamy

Relace subtypingu je pro záznamové datové typy $S = \{k_1 : S_1 \dots k_m : S_m\}$ a $T = \{l_1 : T_1 \dots l_n : T_n\}$, kde dvojice $k_i : S_i$ označuje položku k_i typovanou na S_i , definována pomocí následujících třech kritérií:

- **subtyping do šířky:** Delší záznam je sub-typ kratšího záznamu ($n \leq m \Rightarrow S <: T$),
- **subtyping do hloubky:** Všechny položky na stejné pozici musí být v relaci subtypingu (pro každé $i : S_i <: T_i$),
- **permutační subtyping:** Na pořadí položek nezáleží, pokud existuje $\{k_j : S_j | j \in 1..n\}$, které je permutací $\{l_i : T_i | i \in 1..n\}$.

Uvažujeme-li na příklad typy S a T definované jako:

$$S = \{a : int, b : boolean, c : char\}, \quad T = \{a : long\} \quad (3.2)$$

platí, že $S <: T$, protože S má méně položek, než T a zároveň $int <: long$. V případě typů S a T definovaných jako:

$$S = \{a : int, b : boolean, c : char\}, \quad T = \{a : short, b : char\} \quad (3.3)$$

$S <: T$ neplatí, protože $\overline{int <: short}$ a $\overline{boolean <: char}$ a i když je dodrženo pravidlo "do šířky", není dodrženo pravidlo "do hloubky" ani v případě použití permutačního pravidla.

Funkce

Relaci subtypingu lze také definovat pro funkce. Zde se vychází z předpokladu, že funkci lze předat jako parametr a je tedy nutné určit, zda je možné použít typovanou funkci v kontextu, kde je očekáván jiný typ.

Uvažujeme-li funkce f_1 a f_2 definované v 3.4, pak relace subtypingu platí pokud jsou splněny podmínky $T_1 <: S_1$ a $S_2 <: T_2$ (viz 3.5).

$$f_1 : S_1 \rightarrow S_2, \quad f_2 : T_1 \rightarrow T_2 \quad (3.4)$$

$$T_1 <: S_1, S_2 <: T_2 \vdash f_1 <: f_2 \quad (3.5)$$

V případě typů na levé straně funkce (typy vstupních parametrů) je tedy relace subtypingu tzv. kontravariantní. Tato změna má následující zdůvodnění: pokud je někde očekávána funkce s argumentem typovaným na T_1 , je bezpečné použít funkci s argumentem typovaným na S_1 tak, že $T_1 <: S_1$, protože T_1 lze použít tam, kde je očekáváno S_1 (definice subtypingu).

Očekáváme-li v programu například funkci $f_1 : \text{short} \rightarrow \text{double}$, je bezpečné ji nahradit funkcí $f_2 : \text{int} \rightarrow \text{float}$, protože datový typ int dokáže pojmout typ short , který by při volání funkce byl použitý. Výsledek vrácený funkcí f_2 je typovaný na float a je tedy bezpečné jej uložit do proměnné typované na double , což je očekávaný návratový typ f_1 . Obě podmínky z 3.5 jsou tedy splněné a platí $f_2 <: f_1$.

Objekty

Z pohledu subtypingu se na objektové datové typy dá nahlížet jako na záznam s přidanými funkcemi. Dva objektové typy jsou v relaci subtypingu pokud jsou v této relaci i všechny jejich atributy a metody [1].

Problém s relací subtypingu nastává v případě rekurzivních typů s funkcemi jejichž argumentem je *self* a výstupem je hodnota typovaná stejně jako *self*, tedy například typ X s funkcí $f : X \rightarrow X$. Příčinou je nedodržení principu kontravariance popsaného u subtypingu funkcí. Následující příklad, který tento problém ilustruje, byl přejat z [1] a pro vyjádření typů používá upravenou notaci lambda kalkulu, která označuje funkce symbolem $^+$.

Uvažujeme-li typy Max , $MinMax$ definované v 3.6 a 3.7, pak při konstrukci relace $MinMax <: Max$ narazíme na funkce $max^+ : Max \rightarrow Max$ a $max^+ : MinMax \rightarrow MinMax$. Aby mezi těmito byla relace subtypingu, musí zároveň platit $Max <: MinMax$ a $MinMax <: Max$ což je spor a typ $MinMax$ tedy není pod-typ Max . Řešení tohoto problému je stručně popsáno v následující části.

$$Max \triangleq \mu(X)[n : Int, max^+ : X \rightarrow X] \quad (3.6)$$

$$MinMax \triangleq \mu(Y)[n : Int, max^+ : Y \rightarrow Y, min^+ : Y \rightarrow Y] \quad (3.7)$$

3.3.2 Matching na základě subtypingu

Relace subtypingu v případě rekurzivních objektových typů naráží na problém popsany v sekci 3.3.1. M. Abadi a L. Cardeli ve svém článku [1] zavádějí relaci *matching* nad tzv. protokoly typů, což jsou reprezentace jejich rozhraní. Tato relace vychází ze subtypingu a je zapisována jako $A <\# B$, což znamená “protokol typu A rozšiřuje protokol typu B”.

V příkladu z předešlé sekce 3.3.1 by protokoly typů *Min* a *MinMax* byly definovány jako:

$$MaxProt \triangleq n : Int, max^+ : Self \rightarrow Self \quad (3.8)$$

$$MinMaxProt \triangleq n : Int, max^+ : Self \rightarrow Self, min^+ : Self \rightarrow Self \quad (3.9)$$

Klíčové slovo *Self* označuje typ, pro který mimo jiné platí $Self <: Self$. Pokud použijeme postup konstrukce subtypingu na zmíněné protokoly, výsledkem bude $MinMaxProt <\# MaxProt$. Relace *matching* nad protokoly typů neimplikuje subtyping daných typů a nelze tedy vyvodit, že $MinMax <: Max$.

4 Získávání metadat webových služeb v CRCE

Realizační část této práce se zabývá návrhem a vytvořením nástroje, jež bude schopen vyhodnocovat vzájemnou kompatibilitu webových služeb. K tomu bylo vybráno existující úložiště CRCE¹, neboť pro něj již jsou implementovány moduly schopné analýzy webových služeb a uložení získaných metadat. Tato kapitola představuje samotné úložiště, formát metadat a obecný způsob jejich získání. Na konci kapitoly je popsán princip fungování konkrétních rozšíření, která indexují webové služby (tzv. indexery), včetně ověření jejich funkcionality.

4.1 CRCE

CRCE je komponentové úložiště dlouhodobě vyvíjené a spravované výzkumnou skupinou RelISA na Katedře Informatiky ZČU, jehož primárním účelem je indexace komponent a následná kontrola jejich vzájemné kompatibility. Proces indexace v sobě zahrnuje analýzu a sběr metadat, jež jsou následně uložena do CRCE. Úložiště je postaveno na modulární architektuře (viz obrázek 4.1) a je tedy možné přidat rozšíření pro indexaci a zpracování vlastních dat.

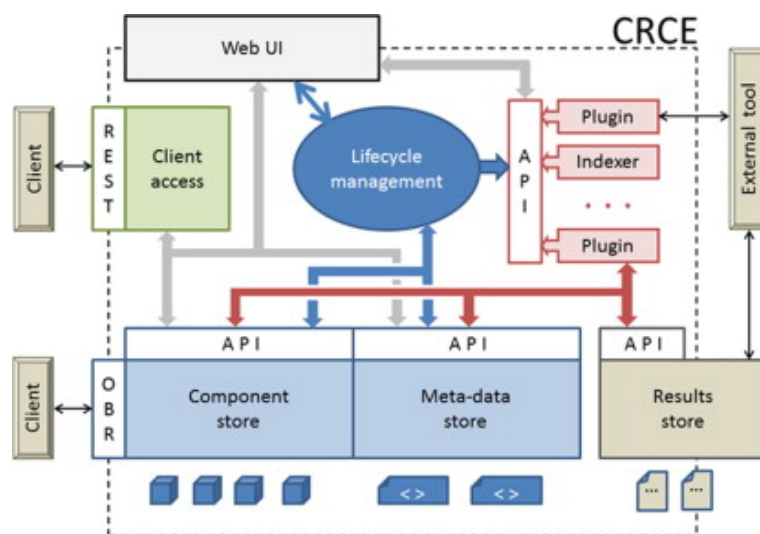
4.1.1 Metadata komponent

Data, která vzniknou indexací komponenty a případným dalším zpracováním (např. porovnáním) jsou uložena do souboru metadat a představují klíčový element systému CRCE. Návrh struktury těchto metadat, který je naznačen na obrázku 4.2, vychází z konceptu OBR², jehož základními entitami jsou mimo jiné *Resource*, *requirements* a *capabilities* [8].

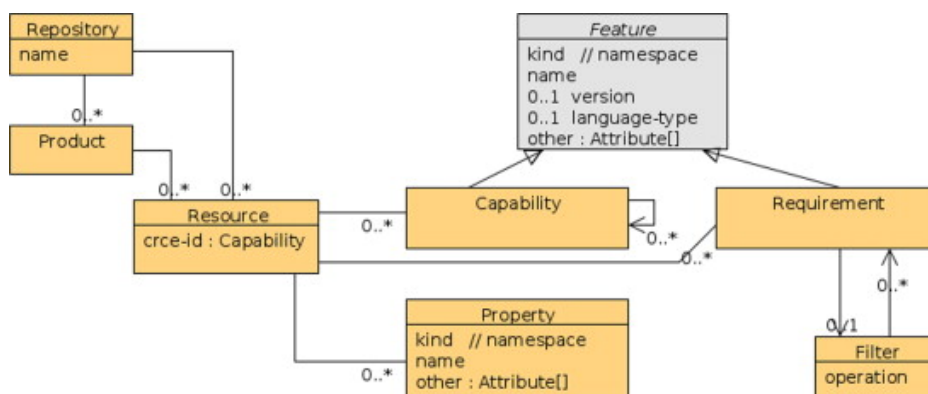
Entita *Resource* reprezentuje komponentu uloženou v CRCE, *requirements* a *capabilities* jsou množiny vlastností, popisující co komponenta ke své správné funkci vyžaduje, respektive co naopak poskytuje. Model také umožňuje přidání key-value atributů k jednotlivým vlastnostem a jejich detailům.

¹Component Repository supporting Compatibility Evaluation

²OSGi bundle repository



Obrázek 4.1: Architektura CRCE



Obrázek 4.2: Reprezentace metadat v CRCE

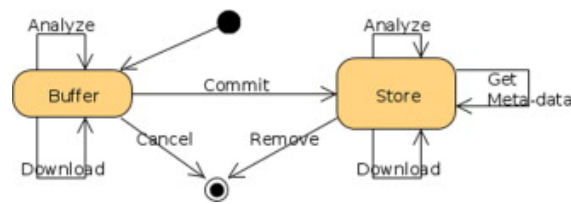
V mé práci jsem pracoval především s poskytovanými vlastnostmi (množina *capabilities*), a proto zde popíši hlavně jejich strukturu. Každá konkrétní vlastnost je reprezentována elementem *Capability* a od ostatních je odlišena identifikátorem *namespace*. Detaily konkrétní vlastnosti jsou popsány elementy *Property* a *Attribute*, kde *Property* reprezentuje logický celek několika atributů. Například parametr endpointu REST služby shlukuje atributy popisující jeho jméno, datový typ atp. *Attribute* pak představuje pár klíč-hodnota, který nese konkrétní informace jako např. jméno endpointu, nebo datový typ parametru.

Z obrázku 4.2 je vidět rekurzivní povaha elementu *Capability*, čehož je využito ke skládání jednodušších vlastností do složitějších celků. Vznikne tím stromová struktura, která je vhodná k modelování hierarchických dat, mezi něž patří například popisy webových služeb. V případě takto komplex-

ních vlastností je ke komponentně (*Resource*) přiřazena pouze jedna, tzv. kořenová, *Capability*, která reprezentuje celou vlastnost.

4.1.2 Životní cyklus komponenty v CRCE

Úložiště bylo původně navrženo pro ukládání OSGi komponent, nicméně indexovat lze jakoukoliv komponentu. Komponenta je v CRCE popsána již zmíněnými metadaty a prochází vlastním životním cyklem naznačeným na obrázku 4.3.



Obrázek 4.3: Životní cyklus komponenty v CRCE

Životní cyklus má dvě hlavní fáze, jimiž jsou *Buffer* a *Store*. Komponenta po nahrání do úložiště nejprve prochází fází *Buffer*, v níž dojde k indexaci obsahu, kontrole vnitřní konzistence a kompatibility komponenty. Pokud touto fází projde bez chyb, může uživatel pomocí operace *commit* komponentu nahrát do trvalého úložiště, čímž dojde k přechodu do fáze *Store*.

V obou fázích jsou nad komponentou prováděny operace, z nichž *analyze* je nejvíce relevantní mé práci, protože právě během této operace dochází ke sběru metadat (fáze *Buffer*) a dalším výpočtům nad nimi (fáze *Store*). Modul s rozšířením, který je předmětem mé práce, bude zařazen mezi výpočty prováděné nad metadaty během *analyze*, kde bude vyhodnocovat vzájemnou kompatibilitu webových služeb. Způsoby sběru těchto metadat, ke kterým dochází ve fázi *Store*, jsou popsány v následující sekci.

4.2 Indexování webových služeb

V této podkapitole je krátce popsán obecný způsob indexace komponent v CRCE. Následně je podrobněji rozebráno indexování webových služeb konkrétními moduly a reprezentace popisu API metadaty v CRCE. Na závěr jsou také uvedeny limity indexování.

4.2.1 Obecná indexace komponenty

Indexace komponenty a související sběr metadat je proveden ve fázi *Buffer* k tomu určenými moduly. Ty jsou vzájemně nezávislé a obecně platí, že každý nich je zaměřen na sběr nějaké logicky ucelené části dat, jako například informace o OSGi bundlu, maven koordináty, nebo popis webových služeb. Vlastní data komponenty zůstávají během tohoto procesu nezměněná, což v kombinaci s řetězením indexerů zaručuje, mimo snadnou rozšiřitelnost, také transparentní přístup ke komponentě každému z nich.

4.2.2 Indexace komponenty s webovou službou

Soubor obsahující implementaci nebo popis API, je v CRCE vnímán jako komponenta a prochází tedy zmíněným životním cyklem včetně výše popsané indexace. Pro popis webových služeb existuje mnoho standardních i nestandardních způsobů, jak již bylo zmíněno v kapitole 2. Z tohoto důvodu není možné je všechny analyzovat jedním indexerem a je nutné zaměřit se pouze na část z nich.

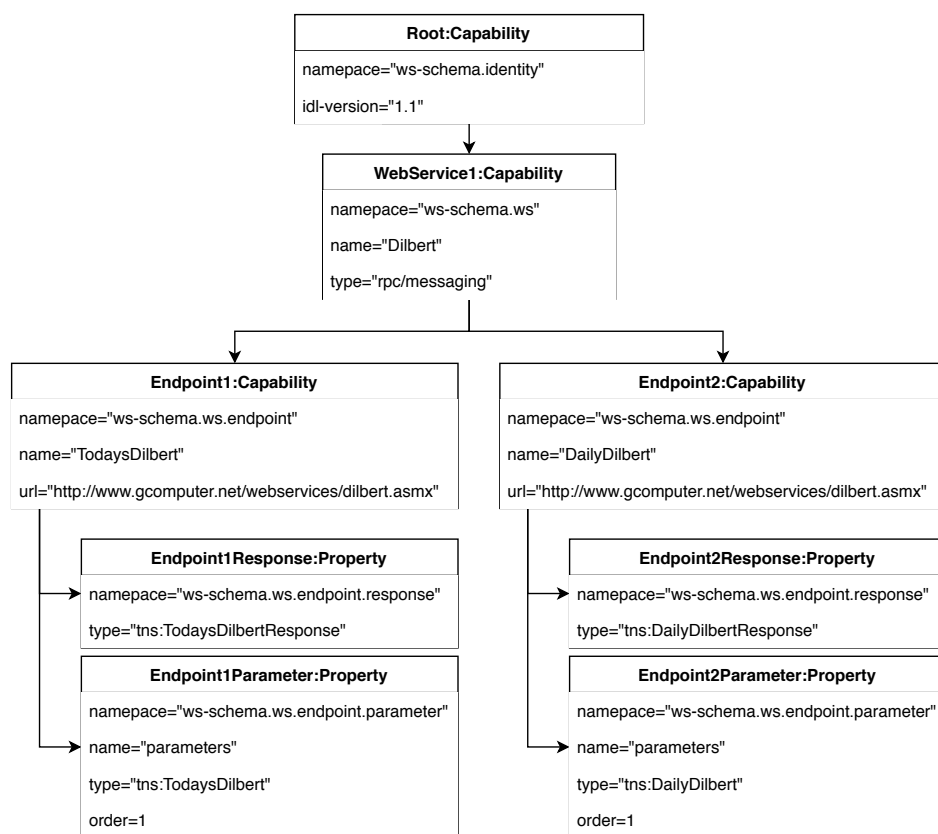
V současné době tedy existují dva moduly podporující několik popisných formátů a implementací. Konkrétně se jedná o modul pro indexaci webových služeb založených na architektonickém stylu REST [14] a o modul pro indexaci webových služeb s popisem ve formátu WSDL, WADL, nebo Json-WSP [20]. Oba dva vznikly v rámci diplomových prací a jsou stručně popsány v následujících sekcích.

4.2.3 Struktura metadat popisující webovou službu

Jak již bylo zmíněno v části 4.1.1, hierarchickou strukturu popisu API lze vhodně vyjádřit metadaty CRCE. Během indexování komponenty reprezentující API jsou shromážděny různé typy popisných vlastností. Jedním z těchto typů je i samotný popis webové služby, který je reprezentován stromem metadat a ke komponentě je přiřazen skrze kořenovou *Capability*.

I když jsou různé druhy API indexovány rozdílnými moduly, výsledná metadata mají podobnou strukturu. Příklad metadat API je zobrazen na objektovém diagramu 4.4, jedná se o webovou službu, která vrací strip komixu Dilbert pro dnešní den.

Z uvedeného obrázku je vidět, že klíčové elementy API, jako web service nebo endpoint, jsou reprezentovány objektem *Capability*. Detaily těchto elementů jsou popsány objekty *Property*. Jedná se zejména o parametry endpointů, těla požadavků a odpovědi. Objekt *Attribute* pak představuje konkrétní hodnoty, jež jsou na obrázku naznačeny jen jako páry "klíč=hodnota".



Obrázek 4.4: Příklad indexované SOAP webové služby pro komix Dilbert

Attribute nemusí být vázaný jen na *Property* a lze jej použít i pro popis *Capability*, jak je tomu např. u objektu *WebService1*.

4.2.4 Indexování implementace REST služeb

Modul pro indexování REST API vznikl v rámci diplomové práce Bc. Gabriely Hessové. Princip sběru dat je založen na binární analýze java archivů (JAR) obsahujících implementaci REST služeb pomocí frameworků splňujících specifikaci JAX-RS a frameworku Spring Web MVC. Modul byl testován na frameworkcích Jersey verze 2.26, RESTEasy verze 3.0.16 a Spring Boot verze 1.5.9 [14].

Z implementace REST služby modul rekonstruuje kolekci endpointů s jejich parametry, tělem požadavku, odpovědi a případnými parametry odpovědi. Každý endpoint je reprezentován entitou *Capability*, všechny další jeho vlastnosti pak entitami *Property*. Výčet všech indexovaných elementů rozhraní je uveden v tabulce 4.1.

Element API	Entita v metadatech	Vstaženo k
Endpoint	Capability	-
Request body	Property	Endpoint
Request parameter	Property	Request
Response	Property Endpoint	Endpoint
Response parameter	Property	Response

Tabulka 4.1: Seznam indexovaných elementů REST služby a jejich reprezentací v metadatech

4.2.5 Indexování webových služeb na základě popisu

Modul pro indexování webových služeb vznikl v rámci práce Bc. Davida Pejřimovského. Oproti předchozímu modulu pro indexaci REST služeb nepracuje tento s její implementací, ale s popisným souborem služby, tak jak bylo uvedeno v kapitole 2. Podporované formáty popisu služeb jsou WSDL (verze 1.1 i 2.0) pro SOAP webové služby, WADL a Json-WSP pro REST služby [20].

Struktura dat vytvořených pro REST služby (tedy z popisu WADL, nebo JSON-WSP) je podobná struktuře dat vytvořené předchozím indexerem. Endpoint je tedy reprezentován entitou *Capability*, jeho parametry entitami *Property*. Z popisu Json-WSP je ještě vytvořena reprezentace odpovědi pro daný endpoint (entita *Property*). Z popisu WADL se žádné další vlastnosti endpointů nezískávají.

Element API	Entita v metadatech	Vztaženo k
Service	Capability	-
Endpoint	Capability	Service
Endpoint parameter	Property	Endpoint
Response	Property	Endpoint
Response parameter	Property	Endpoint

Tabulka 4.2: Seznam indexovaných elementů webové služby a jejich reprezentací v metadatech

Z WSDL popisu je vytvořena reprezentace služeb, jež jsou popsány xml elementy `<wsdl:service>` a jejich vnořených endpointů. Endpoint je ve WSDL popsán elementem `<wsdl:port>` a má definované operace (elementy `<wsdl:operation>`), nicméně modul tyto nevnořuje a vytváří zjednodušenou reprezentaci. Model endpointu tedy obsahuje metadata získaná z elementů `<wsdl:operation>` a url definovanou v elementu `<wsdl:port>`. Služby i endpointy jsou v metadatech reprezentovány entitami *Capability*. Oproti

REST službám, které mají jednu úroveň vnoření *Capability*, zde vznikají úrovně dvě. Výčet elementů API a jejich reprezentace v metadatech je uveden v tabulce 4.2.

4.3 Ověření funkčnosti indexerů a jejich limity

Jedním z bodů zadání mé práce bylo seznámení se indexery a ověření jejich funkčnosti a správné integrace do CRCE. Toho jsem dosáhl prostudováním implementace obou zmíněných rozšíření a také provedením několika manuálních testů.

Během testování jsem v logice parsování WSDL souborů (verze 1.1. i 2.0) objevil chybu ve čtení adresy endpointu. Ta byla očekávána v atributu `action` elementu `<wsdl:operation>`, který ale není uveden ve specifikaci WSDL 1.1 [26] ani WSDL 2.0 [27]. Tuto chybu jsem v rámci mé práce opravil.

4.3.1 Limity indexování webových služeb

Současný proces indexování webových služeb naráží na dva známé problémy týkající se datových typů. Jedná se o indexování rekurzivních datových typů a absenci samotných definic typů.

První problém se týká zejména indexování webových služeb podle popisných souborů, protože ty definice typů obsahují. Tyto rekurzivní typy mohou mít různou složitost, což vede na problémy popsané v kapitole 3. Řešení těchto problémů, jež by umožnilo provést nad těmito datovými typy rozvoj a následně je uložit, existuje, nicméně logika zatím není implementována.

Druhý problém se týká binární analýzy REST služeb, protože archiv s implementací služby nemusí nutně obsahovat definice všech tříd, jež byly v API použité. Ty mohou být například v jiném artefaktu, na který se archiv pouze odkazuje skrze závislost.

Z těchto důvodů je do metadat uložen pouze název datového typu. Důsledkem toho nemůže porovnávací algoritmus (popsaný v kapitole 5) detekovat změny uvnitř typového rozvoje a během porovnání může pracovat pouze se jménem datového typu.

5 Porovnávání webových služeb

Cílem této práce je rozšířit systém CRCE o funkcionalitu pro porovnání webových služeb. Tato kapitola detailně rozebírá porovnávací algoritmus společně s daty, nad kterými je možné porovnávač použít. Zároveň je zde popsán způsob vyhodnocení výsledků porovnání a formát uložení takto získaných dat.

5.1 Reprezentace rozdílů a compatibility

Webové služby jsou popsány komplexní strukturou metadat, která byla uvedena v předchozí kapitole. Rozdíl mezi těmito strukturami lze vyjádřit pouhou pravdivostní hodnotou (stejně, nebo nestejně), nicméně takový přístup skrývá před klientem většinu informací, na základě kterých by se mohl rozhodnout o dalším postupu a tím značně snižuje použitelnost aplikace. Mimo to, samotná skutečnost, že metadata webových služeb se neshodují, nemusí nutně z pohledu klienta znamenat nekompatibilitu. Z těchto důvodů je k reprezentaci rozdílů mezi službami třeba použít vhodnější metodu.

Článek [9] zabývající se možnostmi evaluace compatibility komponent na základě relace subtypingu, nahlíží na komponentu skrze její rozhraní jako na datový typ a popisuje odlišnosti v různých úrovních stromu metadat (třída, operace, parametr, ...). Na základě těchto odlišností (a relace subtypingu) je pak určena míra compatibility dvou komponent. Protože je webová služba v CRCE reprezentována komponentou a na její rozhraní se dá také nahlížet jako na kontrakt datového typu, je tento přístup vhodný k reprezentaci rozdílů mezi webovými službami. Oproti pouhé pravdivostní hodnotě je navíc klientovi poskytnuta detailní informace o konkrétních rozdílech mezi službami. Z těchto důvodů jsem se rozhodl použít výše zmíněný způsob reprezentace rozdílů v mé práci.

Datové struktury použité pro reprezentaci rozdílů dvou entit a jejich vzájemné compatibility navržené v rámci citovaného článku se nazývají *Diff* a *Compatibility*. *Diff* je definován jako rekurzivní typ, který uchovává jak konkrétní informace o rozdílu skrze podřazené *Diff* tak i úroveň odlišnosti zvanou *Difference*. Tyto úrovně jsou popsány tabulkou 5.1 a v citovaném článku tvoří obor hodnot funkce $diff(a, b) : Type \times Type \rightarrow Difference$.

Třída *Compatibility* uchovává informace o kompatibilitě dvou komponent (reprezentovány entitami *Resource*) společně s detaily jejich rozdílů, jež jsou reprezentovány stromem *Diff*.

Název úrovně	Zkratka	Váha	Popis
None	NON	1	$a = b$
Insertion	INS	2	a není definováno, ale b ano
Deletion	DEL	2	a je definováno, ale b ne
Specialization	SPE	3	b je subtyp a ($b <: a$)
Generalization	GEN	3	a je subtyp b ($a <: b$)
Mutation	MUT	4	kombinace <i>INS/SPE</i> a <i>DEL/GEN</i>
Unknown	UNK	5	a nelze porovnat s b

Tabulka 5.1: Popis úrovní rozdílů

Způsob vyhodnocení rozdílů dvou webových služeb společně s významem jednotlivých úrovní *Difference* a jejich vah pro klienta je popsán na konci této kapitoly. Pro přehlednost jsou jednotlivé úrovně rozdílů v průběhu kapitoly nazývané jejich zkratkami.

5.2 Algoritmus porovnání

Porovnávací algoritmus pracuje s metadaty popsány v části 4.1.1. Jedná se o stromovou strukturu, jejíž uzly tvoří instance tříd *Capability*, *Property* a *Attribute*, kde objekty *Attributes* jsou listy této struktury. Soubor metadat může obsahovat další vlastnosti komponenty, která představuje webovou službu, ta však zůstanou nedotčena, protože algoritmus pracuje pouze s daty, která byla vytvořena indexery popsány v části 4.2.

Moduly pro indexování webových služeb popsané v předchozí kapitole používají dvě různé množiny *namespace* identifikátorů po pojmenování entit *Capability*, *Property* a *Attribute*. Do budoucna je zároveň plánované rozšíření těchto modulů o funkcionalitu pro indexování datových typů a datová struktura metadat je tedy předmětem změny. Z těchto důvodů je algoritmus schopen porovnat pouze metadata vytvořená stejným indexerem a používající stejné *namespace* identifikátory. Není tedy možné vzájemně porovnat například metadata REST služby získaná binární analýzou JAR s metadaty získanými čtením JSON-WSP dokumentu i když by se mohlo jednat o jednu službu.

5.2.1 Popis algoritmu

Vstupem algoritmu jsou reprezentace obou webových služeb v podobě objektů *Resource*, z kterých jsou následně k porovnání vybrány kolekce endpointů, případě kolekce *service* obsahující endpointy. Výstupem algoritmu je objekt *Compatibility*, jež obsahuje detailní popis rozdílů mezi webovými službami ve formátu popsaném v sekci 5.1.

Před porovnáváním datových struktur je zkontrolována jejich kompatibilita. Ta je dána následujícími vlastnostmi:

- typ popisu: z čeho byla metadata získána (implementace, formát popisného souboru),
- typ komunikace: jakým způsobem lze službu volat,
- *identity capability*: obsahuje informace o identitě komponenty v CRCE.

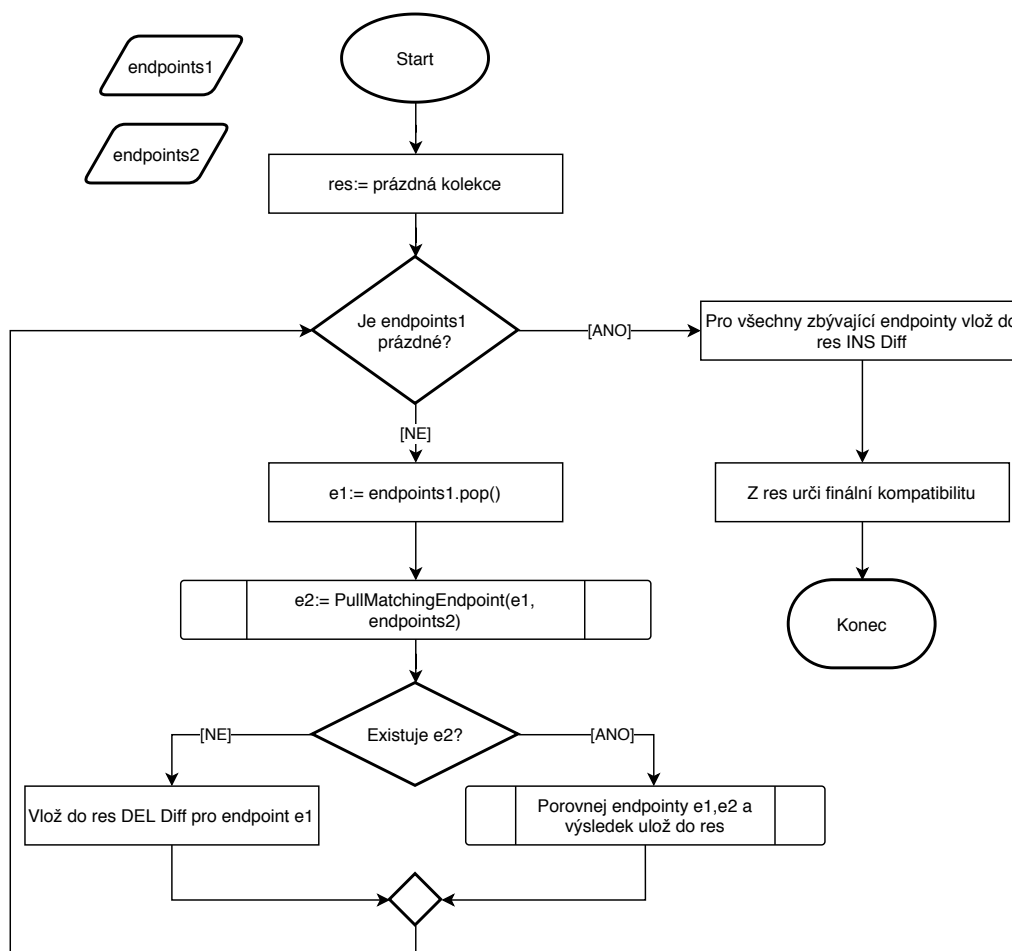
Pokud kontrola proběhne úspěšně, je spuštěn samotný algoritmus, jehož průběh je naznačen na vývojovém diagramu 5.1. Vstupní data na obrázku označená jako *endpoints1* a *endpoints2* označují množiny endpointů první a druhé webové služby.

Tento postup analogicky platí i pro porovnání *service* v případě SOAP webových služeb s tím rozdílem, že krok 'Porovnej endpointy e1, e2 a výsledek ulož do res' by obsahoval porovnání endpointů definovaných v rámci jedné *service*, která je v CRCE reprezentována jako kolekce endpointů. V případě porovnávání *services* by množiny vstupních dat byly označené jako *services1* a *services2*.

Mezivýsledky porovnání jsou ukládány v datové struktuře *Diff*, která je detailně popsána v sekci 5.1. Algoritmus postupuje po stromu metadat od kořene směrem k listům, ve kterých dojde k porovnání konkrétních hodnot a vyhodnocení rozdílů mezi nimi. Rozdíl mezi dvěma ne-koncovými uzly je vyhodnocen až po porovnání všech jejich potomků (nezávisle na pořadí). Algoritmus tedy postupuje zpět ke kořeni stromu, který na konci algoritmu obsahuje finální údaj o rozdílu obou služeb. Jednotlivé fáze porovnání jsou popsány v následujících odstavcích.

Výběr entit vhodných k porovnání

Tento odstavec se týká endpointů a *service*, protože tyto nejsou na pořadí závislé (na rozdíl např. od parametrů operace) a jejich pořadí v metadatech nelze ani předpokládat. Proto je nutné před samotným porovnáním nejprve vybrat dvojici entit k tomu vhodnou. V diagramu 5.1 je zde popsán výběr reprezentován krokem 'PullMatchingEndpoint()'.



Obrázek 5.1: Vývojový diagram porovnávacího algoritmu

Endpoint $e1$ (*service s1*) z první webové služby je vybrán sekvenčně, jak je naznačeno na diagramu 5.1. Druhý endpoint $e2$ ($s2$) je pak vybrán na základě určité shody s metadaty endpointu $e1$, respektive *service s2*. V případě endpointů se konkrétně jedná o počet povinných parametrů, jméno a URL, na které je daný endpoint dostupný. URL nemusí být shodné úplně. Pokud tak nastane, je nastavena vlajka MOV, která je detailně popsána v sekci 5.3.

O porovnatelnosti a rozdílu dvou *services* je rozhodnuto na základě úplné shody jejich jmen a typu (současně je používán pouze typ "rpc/messaging").

Porovnání dvou endpointů

Po výběru vhodné dvojice endpointů dojde k jejich porovnání. Postupně se porovnají (pokud jsou pro daný typ webové služby definovány) parametry, odpovědi a těla požadavků. Způsob porovnání parametrů a odpovědí je roz-

veden v následujících odstavcích. Metadata těla request jsou dostupná pouze v případě REST služeb indexovaných na základě implementace a detail jejich porovnání je znázorněn v tabulce 5.2. U atributu *isOptional* může dojít ke změně *GEN* pokud se povinný atribut stane nepovinným. V opačném případě se jedná o změnu *SPE*.

Název atributu	Možné výsledky
<i>isArray</i>	<i>NON, UNK</i>
<i>isOptional</i>	<i>NON, GEN, SPE</i>
<i>dataType</i>	<i>NON, GEN, SPE, UNK</i>

Tabulka 5.2: Porovnání atributů těl requestů endpointů

Porovnání odpovědí dvou endpointů

Element *response* je definovaný ve všech případech kromě služeb popsaných WADL. V případě REST služeb indexovaných na základě implementace jsou navíc definovány i parametry odpovědi a může také existovat více elementů *response* pro jeden endpoint. Ty jsou navzájem odlišeny atributem *id*, jehož hodnotu vytváří indexer. Detail porovnání odpovědí (včetně parametrů) REST služeb je uveden v tabulce 5.3, tabulka 5.4 pak obsahuje detail porovnání odpovědí ostatních služeb.

Název atributu	Možné výsledky
<i>isArray</i>	<i>NON, UNK</i>
<i>dataType</i>	<i>NON, GEN, SPE, UNK</i>
<i>status</i>	<i>NON, UNK</i>
<i>parameterName</i>	<i>NON, UNK</i>
<i>parameterCategory</i>	<i>NON, UNK</i>
<i>parameterIsArray</i>	<i>NON, UNK</i>
<i>parameterDataType</i>	<i>NON, GEN, SPE, UNK</i>

Tabulka 5.3: Porovnání atributů a parametrů odpovědí endpointů REST služby

Název atributu	Možné výsledky
<i>isArray</i> (pouze JSON-WSP)	<i>NON, UNK</i>
<i>dataType</i>	<i>NON, GEN, SPE, UNK</i>

Tabulka 5.4: Porovnání atributů odpovědí ostatních služeb

Porovnání parametrů endpointů

Množiny porovnávaných atributů parametrů se navzájem liší podle typu metadat služby (viz kapitola 4.2). Kompletní přehled je uveden v tabulce 5.5. Parametry endpointů vhodné k porovnání jsou vybrány na základě jejich jména (atribut *name*). Po výběru dojde k porovnání zbylých atributů a rozdíl mezi parametry je vyhodnocen na základě rozdílů mezi jejich atributy.

Název atributu	Typ metadat služby	Možné výsledky
<i>name</i>	všechny	<i>NON, UNK</i>
<i>dataType</i>	všechny	<i>NON, GEN, SPE, UNK</i>
<i>order</i>	všechny krom WADL	<i>NON, UNK</i>
<i>isOptional</i>	všechny krom WSDL	<i>NON, GEN, SPE, UNK</i>
<i>isArray</i>	REST a JSON-WSP	<i>NON, UNK</i>
<i>category</i>	REST	<i>NON, UNK</i>

Tabulka 5.5: Porovnání parametrů endpointů

5.2.2 Porovnání datových typů

V současnosti jsou největším omezením porovnávacího algoritmu datové typy. Jak již bylo zmíněno v kapitole 4.2, jméno datového typu je jedinou dostupnou informací a tedy také jediným kritériem, podle kterého je lze porovnávat. To je dostatečné v případě vestavěných typů, jako například třídy z balíku `java.lang`, nebo typy definované v xsd.

Nad těmito lze provádět plnohodnotné porovnání včetně kontroly generalizace (změny *GEN*, *SPE*). U vestavěných typů Javy je generalizace určena na základě dědičnosti a lze například určit, že datový typ s názvem `java.lang.Number` je generalizací typu s názvem `java.lang.Long`, protože třída *Number* je rodičem třídy *Long*. Obdobně je tomu i u vestavěných typů definovaných v XSD, kde je generalizace $a <: b$ definována v případě, že typ *b* je dost velký na pojmutí typu *a*. Například typ `xsd:long` dokáže reprezentovat číslo typu `xsd:int` a proto platí, že `xsd:int <: xsd:long`.

Porovnání vestavěných typů je založeno na podmínce správně zaindexovaného jména datového typu. V případě analýzy byte kódu je tato podmínka splněna, nicméně u metadat získaných čtením popisných XML souborů tomu tak vždy nemusí být. Problém spočívá v předponě datového typu, která je součástí jeho jména a porovnávací algoritmus očekává její určitou hodnotu (konkrétně *xs*). Hodnota předpony je však určena definicí namespace v XML popisu a ta se může od očekávané lišit. Tato definice není indexerem ukládána a není proto možné ji během porovnání přesně určit.

Pro uživatelsky definované typy je z těchto důvodů porovnání omezeno pouze na úplnou shodu názvu včetně prefixu, protože na základě pouhého jména datového typu nejde s jistotou usoudit nic dalšího. Změny pro jednotlivé rozdíly mezi datovými typy jsou uvedeny v tabulce 5.6.

Vztah typů	Výsledek
$A = B$	<i>NON</i>
$A <: B$	<i>GEN</i>
$B <: A$	<i>SPE</i>
$A \neq B$	<i>UNK</i>

Tabulka 5.6: Rozdíly mezi datovými typy A a B

5.2.3 Časová složitost algoritmu

Algoritmus v zásadě porovnává kolekce endpointů webových služeb a jejich počet je tedy hlavní parametr, od kterého se odvíjí časová složitost. V případě SOAP webových služeb jsou operace definované v rámci *service* a je tedy třeba brát v úvahu i jejich počet. Elementární operací algoritmu je výběr vhodného páru entit (*service*, nebo endpoint) a jejich následné porovnání.

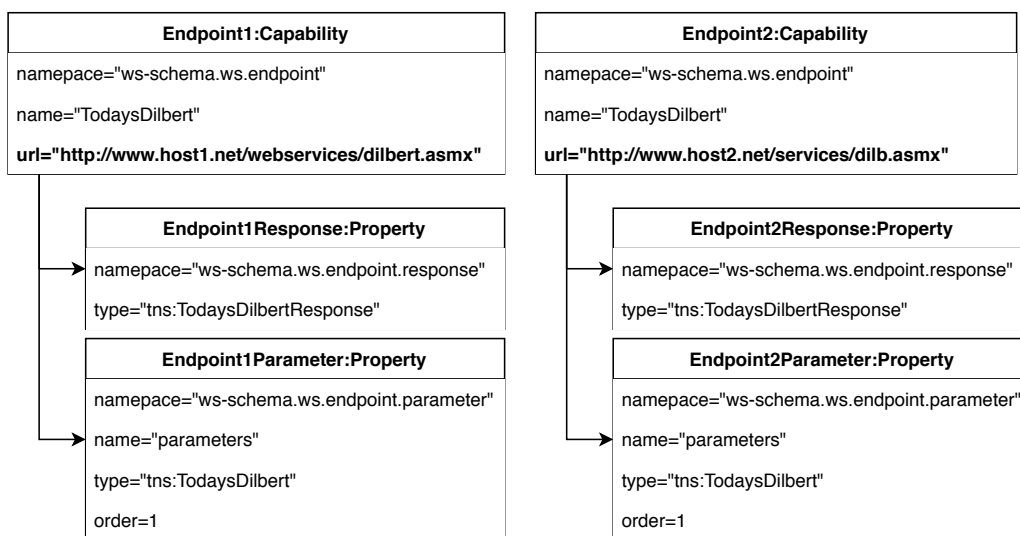
Proces výběru entit k porovnání, jež byl popsán v sekci 5.2.1, hledá k prvku z množiny první webové služby (*endpoint1*, *services1*) vhodný prvek v množině druhé webové služby (*endpoints2*, *services2*). Z obrázku 5.1 je vidět, že v případě nevhodného řazení entit bude potřeba zkontrolovat n prvků z množiny *endpoints2* (*services2*), než bude nalezen vhodný protějšek k porovnání. Pokud taková situace nastane pro každý prvek z množiny první webové služby, bude potřeba provést $n * n$ výběrů a následných porovnání. V případě vhodného řazení bude naopak potřeba provést pouze n výběrů, protože vhodné protějšky budou v obou množinách na stejném místě a k jejich nalezení bude vždy potřeba pouze jedno porovnání. Časová náročnost algoritmu pro jednotlivé typy služeb je shrnuta v tabulce 5.7.

Typ služby	Složitost
REST, WADL, JSON-WSP	$\Omega(n)$, $O(n^2)$ kde n je počet endpointů
WSDL	$\Omega(mn)$, $O((mn)^2)$ kde m je počet <i>service</i> a n je počet endpointů

Tabulka 5.7: Složitost algoritmu

5.3 Migrace webových služeb

V popisu porovnání metadat endpointů bylo zmíněno použití URL endpointu jako identifikátoru. Tento přístup naráží na problém v případě, že poskytovatel ponechá nezměněnou implementaci a webovou službu přesune, nebo provede změny v cestě k danému endpointu. Na obrázku 5.2 je uveden příklad dvou shodných endpointů, jež se liší pouze v URL. Aplikujeme-li výše popsany algoritmus na tato data, skončí negativním výsledkem i přes to, že endpointy mají totožné rozhraní a klient by k nim mohl bez obtíží přistoupit.



Obrázek 5.2: Příklad shodných endpointů s rozdílnou URL

Podobným příkladem je i verze API v cestě k endpointu. Klient může mít požadavek na zjištění kompatibility API dvou různých verzí, ale algoritmus vrátí rozdíl *MUT*, protože endpointy z prvního API vyhodnotí jako chybějící v API druhém a naopak, právě kvůli rozdílným cestám. Tím vznikne kombinace rozdílů *DEL* a *INS*, která vede na *MUT*. Takové chování není žádané a je potřeba těmto problémům předcházet, proto je nutné případné změny v URL detekovat a brát je při porovnávání v potaz.

Jako řešení popsaného problému byl zaveden příznak "MOV", který je ortogonální k úrovni změny (*Difference*) popsané v předchozích sekcích a je součástí objektu *Diff*. Díky ortogonalitě je možné stále určit méně nebezpečné rozdíly jako generalizaci v příznaku parametru (*SPE*) a zároveň předat klientovi informaci o změně v URL.

Příznak MOV má smysl brát v úvahu jen u případů porovnání, jejichž úroveň změny je v podmnožině *NON*, *GEN*, *SPE*, která je v rámci této

sekce označována jako bezpečná. Všechny ostatní úrovně představují v kontextu migrace příliš velkou změnu. Je tedy například nesmyslné nastavit příznak MOV u rozdílu dvou endpointů s úrovní změny *DEL*, protože samotná úroveň změny říká, že endpoint nebyl v druhé webové službě nalezen a proto nelze ani určit, zda byl přemístěn.

5.3.1 Detekce změn v URL

Součástí URL endpointu je také jeho název, změny v URL se tedy dají detekovat na třech místech. Prvním z nich je doména (včetně protokolu), druhým je cesta k endpointu a třetím jeho jméno. URL všech endpointů obou webových služeb jsou podle těchto částí rozděleny, čímž vznikne 6 množin (2 pro každou část URL). Pokud platí, že $url_{i,1} \subseteq url_{i,2}$ kde $url_{i,j}$ je i -tá část url j -té webové služby, je daná část URL považována za nezměněnou. Příklad rozdělení URL dvou webových služeb vycházející z obrázku 5.2 je uveden v tabulce 5.8. V tomto příkladu by byla detekována změna v částech URL "Doména" a "Cesta".

$url_{i,j}$	Služba 1 (j=1)	Služba 2 (j=2)
Doména (i=1)	$\{http : //host1.net\}$	$\{http : //host2.net\}$
Cesta (i=2)	$\{webservices/dilbert.asmx\}$	$\{services/dilb.asmx\}$
Operace (i=3)	$\{TodaysDilbert\}$	$\{TodaysDilbert\}$

Tabulka 5.8: Příklad rozdělení URL na části podle obrázku 5.2

Výsledek detekce změn v URL nese tři příznaky, kde každý z nich určuje, zda byla v dané části URL detekována změna. Vzhledem k tomu, že algoritmus detekce pracuje pouze s URL a jmény endpointů, může dojít k pozitivnímu výsledku i v případě, že jsou porovnávány dvě odlišné webové služby. Aby se redukoval počet false-positives, je potřeba určit, které kombinace příznaků změny mohou vést na MOV a které představují příliš velké odchýlení. Tyto kombinace jsou uvedeny v tabulce 5.9, kde proměnné h , p , n označují změnu v doméně, cestě a jménu endpointu. Hodnota *true* značí změnu.

Z tabulky je vidět, že změna ve jménech endpointů vždy vede na záporný výsledek. Jedním z důvodů je fakt, že změna ve jméně operace je mutací webové služby a jedná se tedy o nekompatibilní změnu, jejíž řešení bylo popsáno v předchozích sekcích. Dalším důvodem je použití jména endpointu jako druhého kritéria pro výběr vhodného páru endpointů (prvním je URL). Bez něj by algoritmus zdegradoval na porovnání "každý s každým", což by značně snížilo efektivitu.

Kombinace	MOV	Zdůvodnění
$!h \wedge !p \wedge !n$	ne	Nebyla detekována žádná změna.
$h \wedge !p \wedge !n$	ano	Změna v doméně, může se jednat o migraci webové služby.
$!h \wedge p \wedge !n$	ano	Změna v cestě k endpointům, může se jednat o restrukturalizaci služby.
$h \wedge p \wedge !n$	ano	Může se jednat o kombinaci obou předchozích.
všechny ostatní	ne	Změna je příliš velká.

Tabulka 5.9: Kombinace změn vedoucích na MOV

5.3.2 Výběr entit k porovnání s příznakem MOV

Protože jedním z kritérií pro výběr vhodného páru endpointů je i URL, je potřeba upravit logiku výběru tak, aby byly brány v potaz výsledky detekce popsané v předchozí sekci. Při porovnávání URL dvou endpointů je podle kombinace změn daná část URL ignorována a pracuje se pouze s nezměněnou částí, u které je vyžadována striktní rovnost.

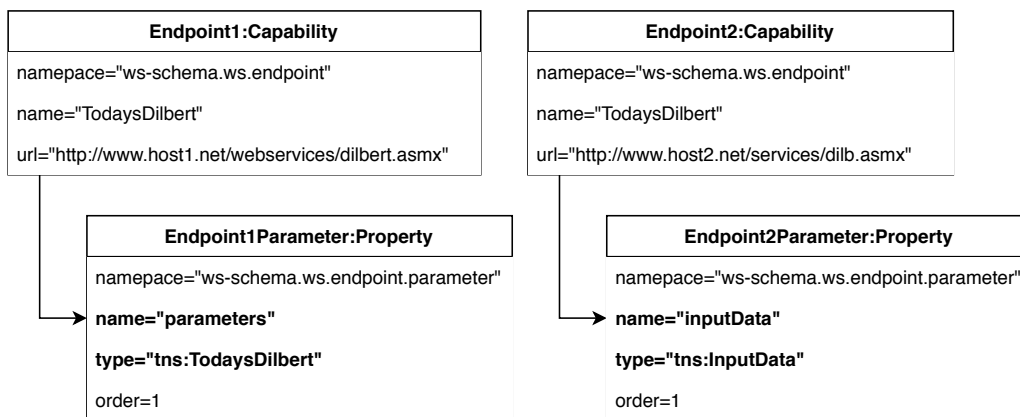
Tento způsob může vést k případům, kdy je dvojice endpointů vyhodnocena jako potencionálně vhodná k porovnání (s nastaveným příznakem MOV), nicméně porovnání skončí negativním výsledkem, například *UNK*. Pouhá akceptace takového výsledku a pokračování algoritmu by mohlo zapříčinit negativní vyhodnocení kompatibility webových služeb v případech, kdy lze mezi službami najít kompatibilní dvojice endpointů.

Příklad na obrázku 5.3 tento problém znázorňuje. Oba endpointy mají stejný název (atribut *name*) a proto by byly vybrány jako vhodné k porovnání s nastavenými MOV příznaky $h = t$, $p = t$, $o = f$. Hlubší porovnání by však skončilo výsledkem *UNK*, protože se neshodují jména a datové typy jejich parametrů a algoritmus by služby správně vyhodnotil jako nekompatibilní. V druhé webové službě by se nicméně mohl nacházet kompatibilní endpoint a je zde tedy možnost získání lepšího výsledku.

Algoritmus 'pick best'

Řešením zmíněného problému je algoritmus 'pick best', který postupně porovnává dvojice endpointů (k tomu předem vybrané) a jako výsledek vrátí dvojici s co možná nejmenším rozdílem. Vstupem algoritmu je tedy endpoint z první webové služby *e1* a množina endpointů druhé webové služby *endpoints2*.

Algoritmus postupně vybírá elementy *e2* z *endpoints2* a pokud je pár



Obrázek 5.3: Příklad dvou rozdílných endpointů, pro které by byl nastaven příznak MOV

$e1, e2$ vyhodnocen jako porovnatelný (viz sekce 5.2.1), dojde k detailnímu porovnání. V případě výsledku spadajícího do bezpečné množiny (NON, GEN, SPE) je pár $e1, e2$ vrácen a porovnání pro endpoint $e1$ je ukončeno. V opačném případě je negativní výsledek uložen a z množiny $endpoints2$ jsou vybírány další porovnatelné endpointy, dokud algoritmus nedojde ke dvojici, jejíž rozdíl spadá do bezpečné množiny, nebo dokud není $endpoints2$ vyčerpána. Pokud je množina $endpoints2$ vyprázdněna, znamená to (alespoň částečnou) nekompatibilitu obou webových služeb a je vrácen první porovnávaný pár $e1, e2$.

5.3.3 Verze REST API v cestě k endpointu

Jednou ze speciálních změn detekovatelných v cestě k endpointu je verze API. Jedná se o běžnou praxi [29] a k jejímu zpracování lze použít jednodušší přístup, než byl dosud popsán. Část cesty, která obsahuje verzi lze jednoduše vypustit a porovnat URL bez verze, což v případě stejného API (s odlišností verze) znamená porovnání dvou identických URL. Příznak MOV je potom nastaven, pokud jsou URL s verzemi rozdílné a URL bez verzí stejné.

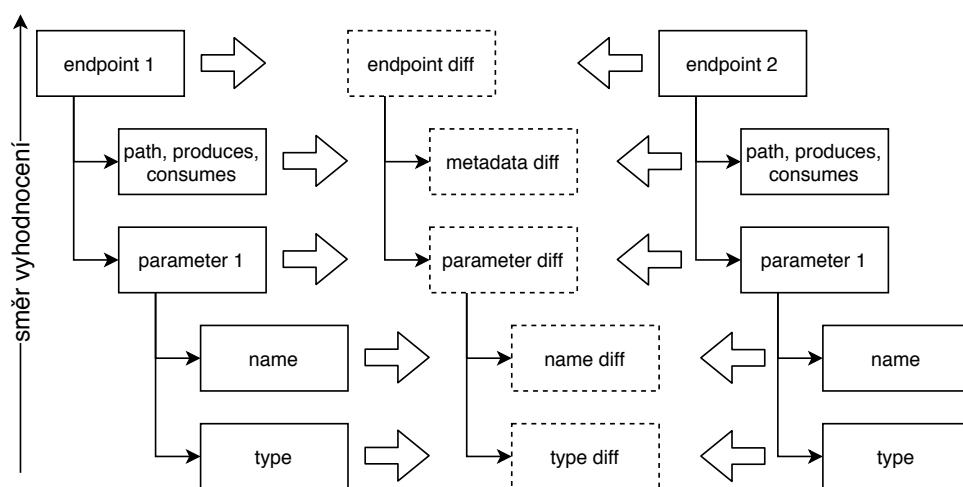
Algoritmus podporuje standardní verzovací formát `major.minor.micro`, který je vyjádřen regulárním výrazem: `/[vV] [0-9]+(?:[. -] [0-9]+){0,2}/`.

5.4 Vyhodnocení výsledků

Mezivýsledky porovnání jednotlivých elementů stromu metadat jsou ukládány do hierarchické struktury *Diff* a vždy po vyhodnocení všech rozdílů

potomků dvou uzlů, dojde na základě těchto i k vyhodnocení rozdílů uzlů samotných. Úroveň rozdílu, jež určuje finální kompatibilitu webových služeb, je dána úrovní kořenového *Diff*, který drží celou strukturu detailně popisující rozdíl mezi službami.

Příklad tvorby *Diff* při porovnání dvou operací je znázorněn na obrázku 5.4. Nejdříve dojde k vyhodnocení rozdílů listů, tedy atributů *name* a *type* parametrů obou operací, čímž vzniknou *type diff* a *name diff*. Na základě těchto se určí rozdíl mezi parametry samotnými (*parameter diff*) a po porovnání metadat operací (*metadata diff*) je vyhodnocen i výsledný rozdíl obou operací, jež je reprezentován objektem *endpoint diff*.



Obrázek 5.4: Tvorba rozdílů mezi dvěma endpointy

Vyhodnocení úrovně rozdílu *Diff* na základě jeho potomků je řízeno prioritou. Každá z úrovní rozdílů *Difference* popsanych v tabulce 5.1 má určitou prioritu, která reprezentuje závažnost rozdílu. Uzel od svých potomků vždy přejímá *Difference* s nejvyšší prioritou, čímž je zaručeno, že se rozdíly narušující kompatibilitu projeví na finálním verdiktu.

Pokud tedy například dojde k rozdílu *UNK* (maximální priorita) při porovnání dvou atributů parametru endpointu, postupným vyhodnocováním se tato *Difference* dostane až ke kořenovému uzlu a výsledná kompatibilita bude mít hodnotu *UNK*. Hodnoty jsou v tabulce 5.1 seřazeny od nejnižší priority po nejvyšší.

5.4.1 Dopad na klienta

Výše popsané úrovně *Difference* mají rozdílný dopad na klienta. Ten se dá rozdělit do skupin bezpečné, potenciálně nebezpečné a nebezpečné, tak jak

je tomu v tabulce 5.10. Následující odstavce popisují dopad na klienta pro jednotlivé úrovně a zdůvodňují jejich zařazení do dané skupiny.

Difference	Dopad na klienta
None (NON)	bezpečné
Specialization (SPE)	bezpečné
Insertion (INS)	bezpečné
Deletion (DEL)	potenciálně nebezpečné
Generalization (GEN)	potenciálně nebezpečné
Mutation (MUT)	nebezpečné
Unkown (UNK)	nebezpečné

Tabulka 5.10: Dopad jednotlivých úrovní rozdílu na klienta

Bezpečné rozdíly

Pokud je rozdíl mezi službami v bezpečné skupině, může klient transparentně volat obě služby, aniž by došlo k jakékoliv chybě v rámci kontraktu. Úroveň *SPE* sice označuje specializaci, nicméně změny *SPE* a *GEN* mohou nastat pouze v případě neshody datových typů parametrů, nebo odpovědi operace. V takovém případě je aplikován princip kontravariance [1], podle kterého platí, že $F'(a') <: F(a) <=> a <: a'$. *SPE* tedy znamená bezpečnou změnu *GEN* u parametrů nebo odpovědi operace.

Změna *INS* nastává v případě, že druhá webová služba obsahuje operace, které nejsou v první definované. Vzhledem k tomu, že je klient nemohl volat a nemůže tak dojít k porušení kontraktu webové služby, je tato změna brána jako bezpečná.

Potenciálně nebezpečné rozdíly

Změny v této skupině mohou mít nebezpečný dopad na klienta ve smyslu volání operace webové služby s neplatným kontraktem, nicméně nejsou tak závažné jako změny nebezpečné. Posouzení reálného rizika změny provádí klient na základě vráceného objektu popisujícího kompatibilitu služeb.

Úroveň *DEL* označuje element definovaný v první službě, ale chybějící v druhé. Může se jednat například o *service*, nebo operaci služby. Úroveň *GEN* je získána, stejně jako *SPE*, na základě principu kontravariance a označuje změnu v datovém typu (parametru, odpovědi operace, ...), například z *long* na *int*.

Nebezpečné rozdíly

Změny v této skupině představují buď kombinace několika méně závažných změn (*MUT*), nebo neporovnatelnost obou webových služeb (*UNK*). Stejně jako u předchozí skupiny, i zde může klient použít výsledky porovnání a sám rozhodnout o míře nekompatibility. Úroveň *MUT* může například nastat i kombinací rozdílů endpointů, které klient nepoužívá a druhá webová služba tak pro něj může být stále kompatibilní.

Úroveň *UNK* označuje buď neporovnatelnost metadat a v takovém případě nelze o kompatibilitě jednoznačně rozhodnout, nebo nerovnost metadat v případě, kde možným výsledkem je pouze 'stejně', nebo 'nestejně'.

6 Implementace rozšíření

Cílem mé práce bylo vytvořit rozšíření CRCE schopné automatické kontroly kompatibility indexovaných webových služeb. V předchozí kapitole byl představen algoritmus pro porovnání dvou webových služeb na základě jejich metadat a následného vyhodnocení výsledků. Tato kapitola popisuje způsob implementace algoritmu jako rozšiřujícího modulu pro systém CRCE.

První část kapitoly, v návaznosti na zmíněnou modulární architekturu CRCE, rozebírá způsob jeho rozšíření a integraci nových modulů. V druhé části je pak uveden stručný popis implementačních detailů výše zmíněného modulu, včetně procesu automatizovaného sestavení a nasazení celého CRCE.

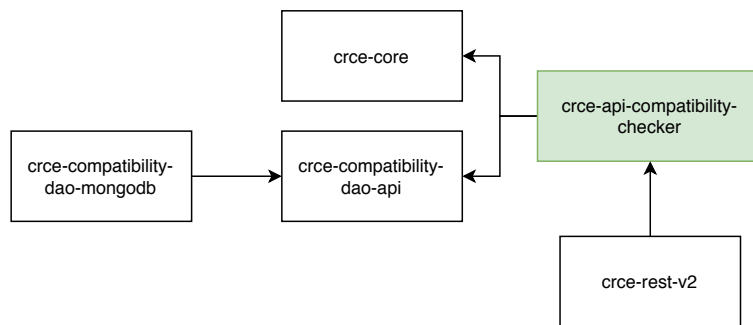
6.1 Rozšíření CRCE

V kapitole 4 byla zmíněna modulární architektura CRCE, jehož jednotlivé moduly jsou tvořeny OSGi bundly v podobě maven artefaktů. Základní funkcionality CRCE je rozprostřena mezi několik modulů tvořící jádro. Ostatní moduly představující nadstavbu jádra jsou podřazené modulu `crce-modules-reactor` a zde je umístěno i mnou vytvořené rozšíření.

6.1.1 Integrace porovnávacího modulu

Modul, jež je výsledkem mé práce, ke své správné funkčnosti vyžaduje několik dalších komponent. Tato závislost je znázorněna na obrázku 6.1, porovnávací modul je zde zeleně označený. Konkrétně se jedná o moduly *crce-core*, který sdružuje všechny komponenty jádra (takže není nutné definovat závislosti jednotlivě) a *crce-compatibility-dao-api*, který obsahuje rozhraní pro přístup k datovému úložišti objektů *Compatibility*.

Výsledky porovnání webových služeb jsou ukládány do perzistentního úložiště a není tedy nutné znovu počítat rozdíly pro již porovnané dvojice. Ukládání těchto dat obstarává modul *crce-compatibility-dao-mongodb*, jež implementuje výše zmíněné rozhraní. V rámci mé práce jsem tyto moduly rozšířil o funkcionality pro získání objektu *Compatibility* podle zadané dvojice *Resource* a mazání existujících objektů *Compatibility*.



Obrázek 6.1: Interakce rozšíření se zbytkem CRCE

6.2 Porovnávací rozšíření

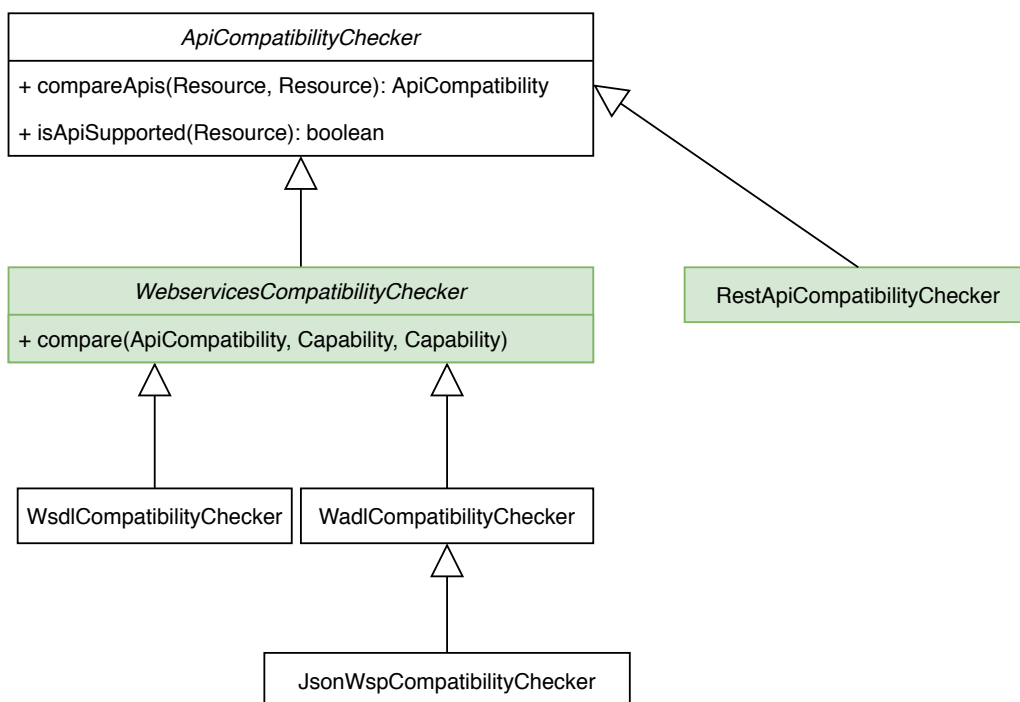
Vnitřně je struktura modulu rozdělena na několik částí. Těmi nejdůležitějšími jsou část implementační nacházející se v balíku `impl`, část pro reprezentaci výsledků v balíku `result` a veřejné rozhraní porovnávací služby v `ApiCompatibilityCheckerService`. Implementace tohoto rozhraní je zpřístupněná formou OSGi služby (anotace `Component`), v níž jsou obsaženy všechny porovnávače jednotlivých typů webových služeb a implementace sama je schopná vybrat správný porovnávač pro zadanou dvojici `Resource`.

6.2.1 Struktura tříd porovnávačů

Diagram 6.2 znázorňuje strukturu tříd, jež slouží k porovnávání webových služeb. Každý porovnávač dědí od abstraktní třídy `ApiCompatibilityChecker`, která obsahuje deklaraci metody určené k porovnání služeb (`compareApis()`). Z důvodů zmíněných v sekci 5.2 je algoritmus implementován ve dvou verzích (na obrázku zeleně vyznačené třídy) podle původu metadat. První verze (na obrázku `RestApiCompatibilityChecker`) pracuje s metadaty získanými z implementace služby, druhá (na obrázku `WebservicesCompatibilityChecker`) pak s metadaty získanými z popisu webové služby.

I když je struktura metadat podobná, rozhodl jsem se vytvářet co možná nejméně společné implementace, abych se vyhnul vytvoření silné vazby mezi jinak nezávislými indexery. Ta by v případě změny struktury metadat u jednoho z indexerů přinášela obtíže při následné refaktorizaci porovnávacího modulu. Jediným společným předkem je tedy obecná abstraktní třída pro porovnávač webových služeb.

Protože indexer pracující s popisem je schopen indexovat více typů služeb, je pro každý z těchto typů vytvořen zvláštní porovnávač. Na obrázku to jsou třídy dědící od `WebservicesCompatibilityChecker`. Tato obsahuje porovnávací logiku a abstraktní metody pro získání konkrétních detailů, jako



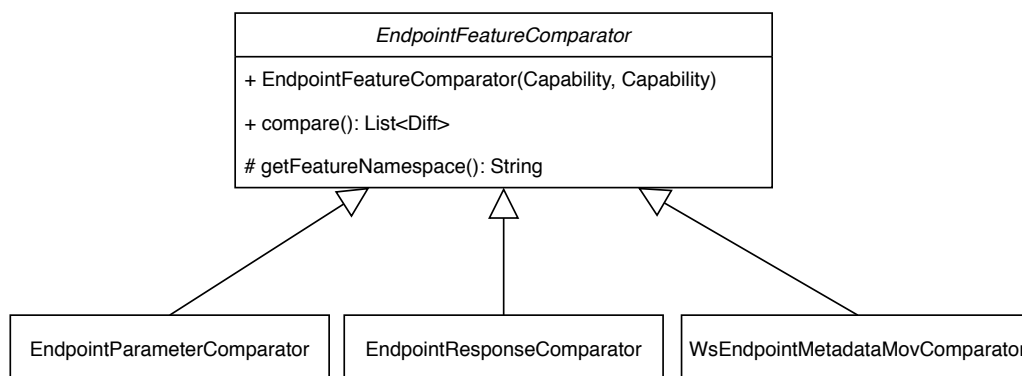
Obrázek 6.2: Diagram tříd popisující strukturu porovnávače

například jmen některých atributů, identifikátorů *Capability*, nebo instancí porovnávačů metadat, které jsou použité během porovnání. Daná implementace pak tyto metody překrývá, čímž poskytuje data specifická danému typu služby.

Porovnávání endpointů

Jak bylo zmíněno v kapitole 5.2, porovnávání endpointů služeb je rozděleno na několik částí. Z důvodů čistoty kódu a principu jedné odpovědnosti je logika porovnání každé části umístěna ve zvláštní třídě a pro každé jedno porovnání je vytvořena nová instance této třídy. Obrázek 6.3 obsahuje diagram těchto tříd společně s abstraktní třídou, od které dědí každý z porovnávačů.

Abstraktní třída *EndpointFeatureComparator* obsahuje pouze extrakci daných vlastností endpointů z objektů *Capability* a abstraktní metodu *compare()*, kterou volá klientský kód. Tím je dosaženo snadné rozšiřitelnosti v případě nutnosti porovnání dalších vlastností endpointu, například pokud by se rozšířila struktura metadat.



Obrázek 6.3: Struktura porovnávače vlastností endpointů

Vestavěné datové typy

Porovnávání vestavěných datových typů je řešeno obalovací třídou, jejíž vstupním parametrem je jméno obalovaného datového typu. Tato třída obsahuje logiku, která je schopná vyhodnotit, zda se doopravdy jedná o jméno vestavěného typu a také logiku porovnání dvou vestavěných typů. V současné době je toto porovnání omezeno na rovnost a generalizaci (jak bylo uvedeno v sekci 5.2.2). Konkrétní obalové třídy jsou *JavaTypeWrapper* pro vestavěné typy Javy a *XsdDataType* pro vestavěné typy XML.

Obalové třídy nepokrývají všechny vestavěné typy, ale pouze jejich podmnožinu. Zejména se jedná o čísla (i s plovoucí desetinnou čárkou) a řetězce.

6.3 Sestavení a nasazení modulu

Jedním z hlavních bodů oborového projektu, který předcházел této diplomové práci, bylo zprovoznění automatického sestavení a nasazení CRCE do vývojového prostředí. Toho jsem využil při vývoji porovnávacího modulu zejména k rychlému nasazení nových verzí, které pak bylo možné pohodlně otestovat. Sestavením CRCE na nezávislém stroji jsem zároveň ověřil, že mnou vytvořený kód netrpí chybami typu "funguje na mém stroji".

Kód CRCE je udržován na univerzitním Gitlabu, který kromě podpory verzování nástrojem GIT umožňuje také kontinuální integraci a nasazení pomocí Gitlab Pipelines¹. Vstupem tohoto nástroje je deklarativní konfigurace jednotlivých kroků od překladu až po nasazení. Pipeline se automaticky spustí vždy při nahrání nového commitu do úložiště. Součástí nakonfigurované pipeline je také spuštění unit testů.

¹<https://docs.gitlab.com/ee/ci/pipelines/index.html>

```

1  image: maven:3.6.2-jdk-11
2
3  # stages defined in this pipeline
4  stages:
5  |   - build
6  |   - deploy
7  |
8  build:
9  |   stage: build
10 |   script:
11 |     - cd ../core && mvn clean install
12 |     - cd ../modules && mvn clean install
13 |   artifacts:
14 |     paths:
15 |       - deploy/*
16 |
17 deploy_vps:
18 |   stage: deploy
19 |   script:
20 |     - scp -r deploy deploy@${DEV_ENV_IP}:/opt/deploy
21 |     - ssh deploy@${DEV_ENV_IP} "/opt/deploy/deploy.sh"

```

Obrázek 6.4: Příklad konfigurace Gitlab pipeline

Soubor `.gitlab-ci` obsahující nastavení pipeline ve formátu YAML se nachází v kořenovém adresáři projektu a obsahuje definici kroků pro sestavení a následné nasazení do testovacího prostředí. Obrázek 6.4 znázorňuje příklad zjednodušené pipeline, její plná verze je uvedena v příloženém archivu obsahujícím zdrojový kód CRCE.

7 Ověření funkčnosti

Rozšíření do systému CRCE, jež bylo předmětem mé práce, bylo testováno jednotkovými a integračními testy na kombinaci syntetických i reálných dat. Tato kapitola se postupně ve třech částech zabývá použitím modulu a ověřením jeho správné funkčnosti. V první části je rozebráno spuštění a použití modulu od nahrání artefaktů do CRCE až po získání výsledků porovnání. Druhá část této kapitoly obsahuje popis integračních testů, včetně testovacích dat. Na konci kapitoly jsou shrnuty provedené testy a jejich výsledky.

7.1 Spouštění modulu

Na rozdíl od indexerů zmíněných v kapitole 4.2 není modul pro počítání kompatibility přímo napojen na žádnou z fází životního cyklu komponenty v CRCE a je potřeba jej manuálně spouštět pomocí rozhraní webových služeb CRCE. To je v současné době implementováno v modulu *crce-rest-v2*. Zde je pomocí dependency injection zpřístupněná služba *ApiCompatibilityCheckerService*, která na základě předaných vstupních parametrů provede porovnání a vrátí výsledek v podobě objektu *Compatibility*. Ten je pak vrácen klientovi. Endpoint na němž je zpřístupněn porovnávací modul je dostupný na URI¹:

`http://<adresa.serveru>/rest/v2/apicomp/compare`

Parametry volání jsou uvedeny v tabulce 7.1. Nepovinný parametr *force*, jehož výchozí hodnota je *false*, byl přidán z důvodů nutnosti přepočítání výsledků v případě změny implementace. Pokud tedy dojde například k opravě chyby, díky které jsou v databázi uložena špatná data, není nutné je ručně mazat, ale stačí porovnání spustit znovu s nastaveným příznakem *force*. Příklady volání API a následně vrácených dat jsou v následující podkapitole.

Název	Popis	Povinný
id1	ID 1. <i>Resource</i> k porovnání	Ano
id2	ID 2. <i>Resource</i> k porovnání	Ano
force	Ignorovat existující <i>Comaptibility</i>	Ne

Tabulka 7.1: Parametry volání endpointu pro porovnání služeb

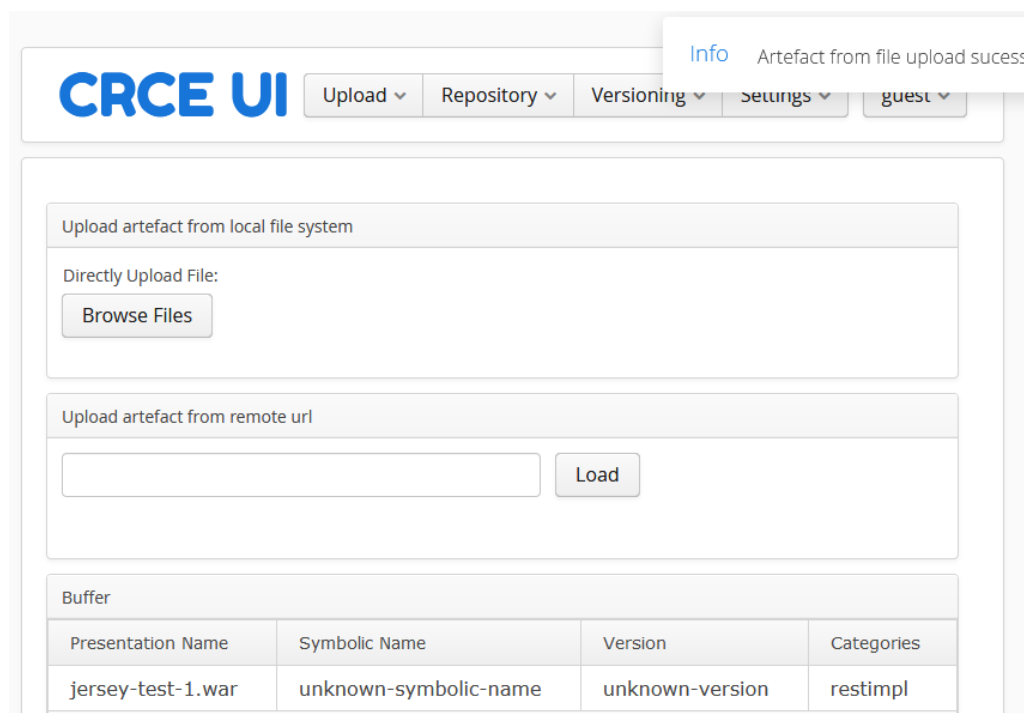
¹'<adresa.serveru>' slouží pouze jako zástupce pro reálnou IP adresu nebo hostname

7.2 Použití modulu

Aby bylo možné modul použít, je nejprve potřeba nahrát testovací data. Tato sekce popisuje proces nahrání artefaktu obsahujícího implementaci webové služby, do CRCE a jeho následného porovnání se sebou samým. Uživatelské rozhraní testovací instance systému CRCE je dostupné na adrese:

`http://<adresa.serveru>/crce-webui/`

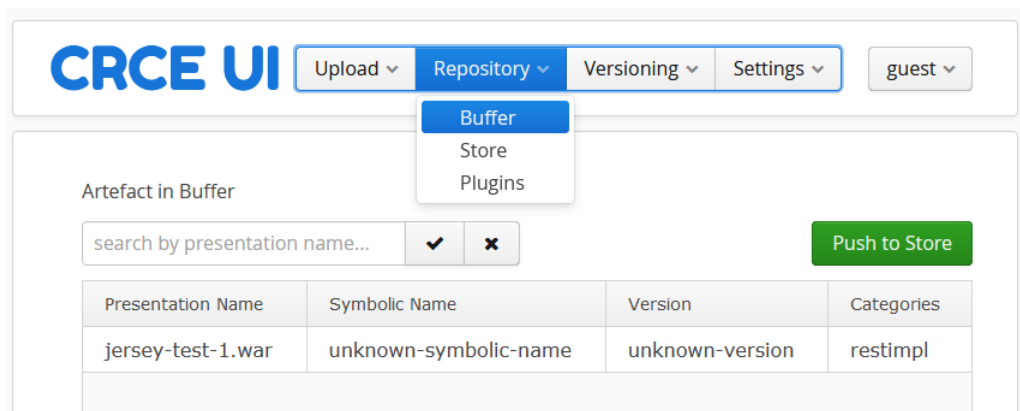
Nahrání artefaktu do CRCE Uživatelské rozhraní pro nahrávání artefaktů je zpřístupněno pod položkou *File/url* menu *Upload*. Zde je možné tlačítkem *Browse files* vybrat artefakt (v tomto případě *jersey-test-1.war*) a potvrdit nahrání, po jehož dokončení se v pravém horním rohu zobrazí zpráva oznamující úspěch, viz obrázek 7.1.



Obrázek 7.1: Nahrání artefaktu s implementací webové služby do CRCE

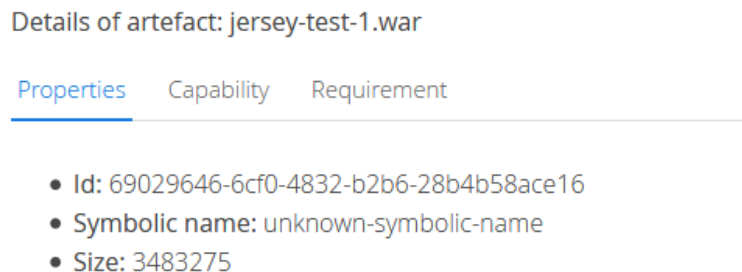
Nahrání artefaktu do Store V tuto chvíli se nahraná komponenta nachází ve fázi *Buffer*. Došlo tedy k získání metadat a indexaci artefaktu, nicméně pro porovnání je nutné provést operaci *commit* a tím komponentu posunout do fáze *Store*. To lze udělat tlačítkem *Push to Store*, v záložce *Plugins*, pod menu *Repository*, jak je znázorněno na obrázku 7.2. Tato záložka

mimo jiné obsahuje přehled všech komponent nacházejících se ve fázi *Buffer* svého životního cyklu.



Obrázek 7.2: Nahrání artefaktu s implementací z Buffer do Store

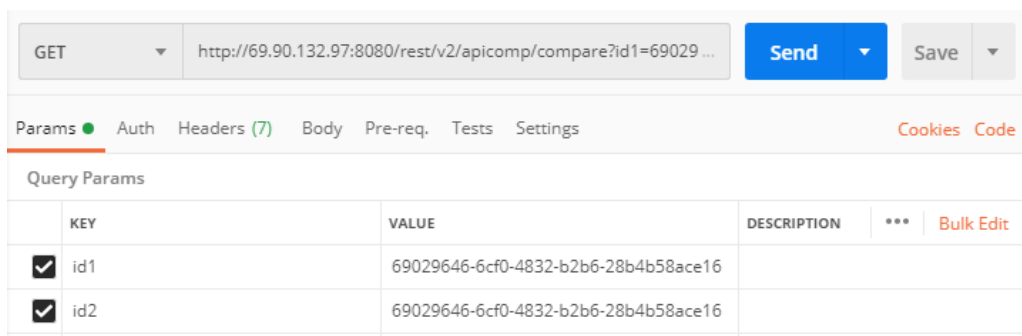
Získání ID artefaktu K porovnání webových služeb je nutné znát ID *Resource*, jež reprezentují nahrané artefakty. Označením daného artefaktu v tabulce se zobrazí menu s akcemi *Detail* a *Remove* pro daný artefakt. Kliknutím na tlačítko *Detail* je zobrazena stránka s metadaty artefaktu, včetně seznamů *Capability* a *Requirement*. Detail artefaktu nahraného v předchozím kroku je na obrázku 7.3.



Obrázek 7.3: Detail nahraného artefaktu

Volání porovnávací služby Identifikátor získaný v předchozím kroku lze použít k volání REST služby pro porovnávání. K tomu jsem pro jednoduchost použil nástroj Postman². Protože se tento příklad zabývá pouze jedním artefaktem, jsou parametry *id1* a *id2* totožné. Příklad volání služby s danými identifikátory je na obrázku 7.4.

²<https://www.postman.com/>



Obrázek 7.4: Volání porovnávací služby

Výsledky porovnání Výsledek vrácený porovnávací službou je uveden na obrázku 7.5. Pro jednoduchost jsou některé části výsledného JSON dokumentu vypuštěny. Ve výsledku je obsažena finální úroveň rozdílu (*diffValue*), příznaky nastavené během porovnání (*additionalInfo*) a také struktura *diffDetails* tvořenou stromem objektů *Diff* popisující detailní rozdíly mezi službami.

```

1 {
2   "id": "5eaad2db0cf23066dcf07704",
3   "resourceName": "69029646-6cf0-4832-b2b6-28b4b58ace16",
4   "baseResourceName": "69029646-6cf0-4832-b2b6-28b4b58ace16",
5   "diffValue": "NON",
6   "diffDetails": [...],
7   "contract": "INTERACTION",
8   "additionalInfo": {}
9 }
```

Obrázek 7.5: Volání porovnávací služby

Pokud by během porovnání došlo k nastavení příznaku MOV, byl by tento vrácený v *additionalInfo*. Protože byl artefakt porovnán sám se sebou, je výsledný rozdíl *NON*.

7.3 Testování

Funkčnost modulu byla ověřena pomocí několika testovacích scénářů, jejichž účelem je otestování změn v rozhraní webových služeb vedoucích na kompatibilní i nekompatibilní rozdíly. Každý ze scénářů je zaměřen na jeden typ indexované služby a všechny s CRCE komunikují skrze REST API, tak jak bylo popsáno v předchozí sekci. Vstupní data pro tyto scénáře byla získána z kombinace reálných i synteticky vytvořených služeb. Premisou pro úspěch testů je správná funkčnost indexovacích modulů popsanych v části

4.2 kapitoly 4. Ta byla ověřena analýzou implementace společně s manuálním testováním, během kterého byla objevena a opravena chyba ve zpracování WSDL dokumentu, rovněž popsána v sekci 4.2.

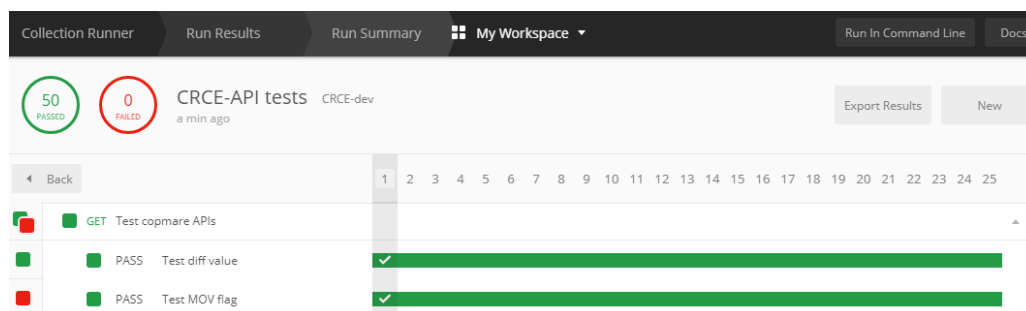
Princip každého z testovacích scénářů spočívá ve vytvoření několika verzí jedné služby (případně jejího popisu) a následném vzájemném porovnání. Tím lze otestovat jak vybrané změny, tak i závislost výsledku na pořadí porovnávaných služeb. Jednotlivé scénáře včetně výsledků jsou detailně popsány v následujících sekcích.

K testování jsem použil nástroj Postman, konkrétně jeho Collection Runner, který umožňuje vybrat šablonu volání REST API a soubor testovacích dat ve formátu CSV. jehož každá řádka představuje jeden test v podobě identifikátorů porovnávané dvojice služeb a očekávaného výsledku. Nástroj Collection Runner tento soubor čte po řádcích a data z každé řádky aplikuje na zmíněnou šablonu. Tím lze snadno zautomatizovat testování volání webové služby s mnoha kombinacemi vstupních dat. Příklad tohoto datového souboru je uveden na obrázku 7.6.

```
1 id1,id2,res-diff,mov-flag
2 4047c969-5abe-441b-9d49-6c614bce13bb,4047c969-5abe-441b-9d49-6c614bce13bb,NON,0
3 4047c969-5abe-441b-9d49-6c614bce13bb,ecc04b3a-8018-4d78-87f8-5390a3c330b8,SPE,0
4 ecc04b3a-8018-4d78-87f8-5390a3c330b8,4047c969-5abe-441b-9d49-6c614bce13bb,GEN,0
5 ecc04b3a-8018-4d78-87f8-5390a3c330b8,ecc04b3a-8018-4d78-87f8-5390a3c330b8,NON,0
```

Obrázek 7.6: Příklad testovacích dat pro nástroj Postman

Test, zda výsledek volání odpovídá datům uvedeným v CSV souboru provádí skript napsaný v jazyce JavaScript, jež je součástí zmíněné šablony volání (k dispozici na příloženém CD). Příklad výsledků testů vrácených nástrojem Collection Runner je zobrazen na obrázku 7.7.



Obrázek 7.7: Výsledek testování nástrojem Postman

7.3.1 REST služba založená na Jersey

V tomto scénáři jsou testovány dvě verze REST služby implementované frameworkem Jersey³. Metadata služby byla získána nahráním její implementace do CRCE způsobem popsáním v sekci 7.2. Popis rozhraní obou verzí služby se nachází na přiloženém CD.

Rozdíl mezi jednotlivými verzemi je pouze v datovém typu parametru *longParam* endpointu *myresource/special/pet/*. V první verzi je tento parametr typován na *java.lang.Number*, v druhé pak na *java.lang.Long*.

Účelem tohoto testovacího scénáře je kromě ověření funkčnosti modulu pro indexaci REST služeb z bytecode také ověření správné aplikace principu kontravariance v případě změn datových typů parametrů endpointů. Změna datového typu z *Long* na *Number* je změnou *GEN*. Po použití kontravariance by změna na úrovni endpointu měla být *SPE* a protože je to jediný rozdíl mezi službami, měla by tato změna označovat také celkový rozdíl.

Výsledky vrácené voláním porovnávací služby jsou zobrazeny v tabulce 7.2. Výsledky odpovídají očekávání a princip kontravariance v případě změn *GEN* a *SPE* u datového typu parametru endpointu služby je tedy aplikován správně.

	v1	v2
v1	<i>NON</i>	<i>SPEC</i>
v2	<i>GEN</i>	<i>NON</i>

Tabulka 7.2: Výsledky porovnání metadat získaných z implementace

7.3.2 Služba popsaná JSON-WSP

Testovací scénář je zaměřen na služby popsané formátem JSON-WSP. Vstupní metadata byla získána indexováním čtyř verzí popisu uměle vytvořeného rozhraní založeného na příkladu převzatém z Wikipedie⁴. Odlišnosti mezi jednotlivými verzemi, označenými *v1* až *v4*, jsou rozepsány v tabulce 7.3. Všechny verze popisů rozhraní služby jsou umístěny na přiloženém CD.

Účelem tohoto scénáře je ověřit porovnávání vlastních datových typů a mutací mezi endpointy. Verze služby *v2* mění definici typu *User*, ale ne jeho název. Vzhledem k faktu, že definice typů nejsou indexovány, mělo by porovnání skončit výsledkem *NON*, protože názvy typů se shodují. Verze *v3* a *v4* přidávají endpointy, čímž mutují rozhraní služby. Očekávané výsledky

³<https://eclipse-ee4j.github.io/jersey/>

⁴<https://en.wikipedia.org/wiki/JSON-WSP>

Verze služby	Popis změny
v1	Základní verze
v2	Modifikace datového typu <i>User</i>
v3	Přidána operace <i>deleteUser</i>
v4	Operace <i>listGroups</i> změněna na <i>getUserInGroup</i>

Tabulka 7.3: Přehled změn ve verzích služby

pro tyto verze jsou *INS* pro *v3* (přidaný endpoint) a *MUT* pro *v4* (změna názvu operace).

Tabulka 7.4 obsahuje výsledky vrácené porovnávací službou, které odpovídají předpokladům uvedeným v předchozím odstavci.

	v1	v2	v3	v4
v1	<i>NON</i>	<i>NON</i>	<i>INS</i>	<i>MUT</i>
v2	<i>NON</i>	<i>NON</i>	<i>INS</i>	<i>MUT</i>
v3	<i>DEL</i>	<i>DEL</i>	<i>NON</i>	<i>MUT</i>
v4	<i>MUT</i>	<i>MUT</i>	<i>MUT</i>	<i>NON</i>

Tabulka 7.4: Výsledky porovnání metadat získaných z JSON-WSP popisu

7.3.3 Služba FuelEconomy

Tento scénář je zaměřen na porovnání REST služeb, jejichž rozhraní je popsáno WADL dokumentem. Metadata, jež jsou vstupem tohoto testu, byla získána indexováním popisu rozhraní služby Fuel Economy⁵ a z něj vycházejících tří modifikovaných verzí. Popis změn jednotlivých verzí oproti verzi *v1* je uveden v tabulce 7.5. Všechny verze WADL dokumentu jsou na příloženém CD.

Verze služby	Popis změny
v1	Základní verze
v2	Odebrán resource <i>labelvehicle</i>
v3	Odebrán resource <i>labelvehicle</i> , přidán resource <i>somethingDifferent</i>
v4	Změna URL a změna cesty z <i>/fuelprices</i> na <i>fuel-prices</i>

Tabulka 7.5: Přehled změn ve verzích služby

⁵<https://www.fueleconomy.gov/ws/rest/application.wadl>

Scénář je zaměřen na mutace operací a změny v URL, na kterých jsou dostupné. Verze *v2* a *v3* testují mutaci, konkrétně přidávání a odebírání operací. Očekávané výsledky jsou *DEL* v případě *v2* a *MUT* v případě *v3*. Verze služby *v4*, kde jedinou změnou je URL a cesta k endpointu *fuelprices*, testuje správné nastavení příznaku *MOV* a očekávaný rozdíl je *NON*.

	v1	v2	v3	v4
v1	<i>NON</i>	<i>DEL</i>	<i>MUT</i>	<i>NON,MOV</i>
v2	<i>INS</i>	<i>NON</i>	<i>INS</i>	<i>INS</i>
v3	<i>MUT</i>	<i>DEL</i>	<i>NON</i>	<i>MUT</i>
v4	<i>NON,MOV</i>	<i>DEL</i>	<i>MUT</i>	<i>NON</i>

Tabulka 7.6: Výsledky porovnání metadat získaných z WADL popisu

V tabulce 7.6 jsou obsaženy výsledky vzájemného porovnání jednotlivých verzí služby. Pro přehlednost je v tabulce uveden příznak *MOV* pouze v případě, že byl vrácen. Pokud nastaven nebyl, je v tabulce uveden pouze rozdíl služeb.

7.3.4 Služba STAG-Číselníky

Testovací scénář je zaměřen na SOAP služby popsané dokumentem ve formátu WSDL. Vstupní metadata byla získána indexováním popisu rozhraní služby STAG-Číselníky⁶ a z něj vycházejících čtyř modifikovaných verzí. Celkem je tedy porovnáno pět verzí, jejichž rozdíly jsou popsány v tabulce 7.7. Všechny verze WSDL dokumentu popisující rozhraní této služby se nachází na příloženém CD.

Verze služby	Popis změny
v1	Základní verze s přidanou operací <i>testOperation</i>
v2	Změna hostname v url operací
v3	Změna typu <i>insertPracoviste</i>
v4	Změna názvu <i>service</i> z <i>CiselnikyServiceImplService</i> na <i>CiselnikyServiceImplServiceUpdate</i>
v5	Změna typu parametru operace <i>testOperation</i> z <i>xs:int</i> na <i>xs:long</i>

Tabulka 7.7: Přehled změn ve verzích služby

⁶<https://stag-ws.zcu.cz/ws/services/soap/ciselniky?wsdl>

Základní verze *v1* byla rozšířena o operaci *testOperation*, která má nastavený parametr typovaný na *xs:int*. Porovnání verze *v1* s verzemi *v2* a *v5* je využito k testování principu kontravariance společně se změnami v URL. Verze *v3* mění definici typu *insertPracoviste*. Tato změna by neměla porovnání nijak ovlivnit, protože detaily vlastních datových typů nejsou indexovány. Verze *v4* mění název *service*, což by mělo být vyhodnoceno jako mutace.

	v1	v2	v3	v4	v5
v1	<i>NON</i>	<i>NON,MOV</i>	<i>NON</i>	<i>MUT</i>	<i>SPE</i>
v2	<i>NON,MOV</i>	<i>NON</i>	<i>NON,MOV</i>	<i>MUT</i>	<i>SPE,MOV</i>
v3	<i>NON</i>	<i>NON,MOV</i>	<i>NON</i>	<i>MUT</i>	<i>SPE</i>
v4	<i>MUT</i>	<i>MUT</i>	<i>MUT</i>	<i>NON</i>	<i>MUT</i>
v5	<i>GEN</i>	<i>GEN,MOV</i>	<i>GEN</i>	<i>MUT</i>	<i>NON</i>

Tabulka 7.8: Výsledky porovnání metadat získaných z WSDL popisu

Výsledky vzájemných porovnání všech verzí služby obsažené v tabulce 7.8 odpovídají předpokladům a to včetně správně nastavených příznaků MOV.

7.4 Shrnutí testů

Testovací scénáře popsané v předchozích sekcích úspěšně ověřily základní funkčnost porovnávacího modulu na všech typech indexovaných služeb, včetně jeho dostupnosti skrze REST API a tedy i úspěšné integrace do systému CRCE.

V případě testování webové služby STAG-Číselníky bylo ověřeno správné vyhodnocení kombinace změn URL a datového typu parametru operace, jehož výsledkem bylo nastavení příznaku MOV v případě bezpečné změny tak jak bylo popsáno v sekci 5.3. Testy rovněž potvrdily limity indexování a následného porovnání vlastních datových typů. Jejich zdůvodnění je v sekcích 5.2.2 a 4.3 předchozích kapitol.

Jednotkové testy určené k ověření funkčnosti elementárních prvků implementace (např. porovnání metadat endpointu nebo detekce změn v URL) jsou spouštěny v rámci Gitlab Pipeline popsané v kapitole 6. Jejich případné selhání by znamenalo zastavení celé pipeline a tím by bylo zabráněno nasazení chybné verze.

Na příloženém CD jsou k dispozici testovací data k jednotlivým scénářům, včetně exportovaných metadat pro jednotlivé verze služeb. Stejně tak jsou přiloženy konfigurace testů pro nástroj Postman.

8 Závěr

Tato práce přímo navazuje na diplomové práce Bc. Gabriely Hessoové a Bc. Davida Pejřimovského, ve kterých byly představeny metody pro analýzu webových služeb a následného uložení jejich popisů do CRCE. Předmětem mé práce pak bylo takto získané popisy vzájemně porovnat a na základě rozdílů rozhodnout o kompatibilitě jimi reprezentovaných služeb.

Prvním krokem v mé práci byla analýza implementace a následné ověření funkčnosti výše zmíněných rozšíření. To odhalilo několik drobných chyb jež se projevovaly uložním vadných metadat a byly následně opraveny. Po ověření funkčnosti byl na základě analýzy datového modelu webových služeb navržen algoritmus porovnání zmíněných metadat, způsob reprezentace takto získaných výsledků a jejich následné vyhodnocení a uložení do CRCE.

Tento návrh byl realizován v podobě rozšiřujícího modulu integrovaného do CRCE, jež byl zpřístupněn skrze REST API, které CRCE poskytuje. Součástí integrace bylo také napojení modulu na komponenty, zajišťující persistenci dat. Celé řešení bylo následně nasazeno do vývojového prostředí a otestováno na kombinaci reálných a umělých dat, čímž byla potvrzena předpokládaná funkcionálnost.

V rámci budoucích prací by mohlo dojít k stávajících indexerů o podporu dalších typů služeb, například popsáných jazyky OpenAPI nebo RAML, jež byly zmíněny v kapitole 2. Důležitým rozšířením je také indexování datových typů definovaných v popisu služby, na základě kterého by mohl být porovnávací algoritmus doplněn o možnost sofistikovanějšího porovnání těchto typů, například způsoby, jež byly popsány v kapitole 3. Tím by bylo dosaženo přesnějších výsledků při porovnávání služeb a také zlepšení celkové user experience.

Literatura

- [1] ABADI, M. – CARDELLI, L. On Subtyping and Matching. In *European Conference on Object-Oriented Programming (ECOOP), Lecture Notes in Computer Science*, 952, s. 145–167. ACM Press, January 1995. Dostupné z: <https://www.microsoft.com/en-us/research/publication/on-subtyping-and-matching/>.
- [2] ABADI, M. – CARDELLI, L. *Imperative object calculus*, 1, s. 469–485. 01 2006. doi: 10.1007/3-540-59293-8_214.
- [3] MOGENSEN, T. *Basics of Compiler Design*. Department of Computer Science, University of Copenhagen, 2010. ISBN 978-87-993154-0-6.
- [4] ALLAN B. TUCKER, L. C. *Computer science handbook*, Type Systems. CRC Press, 2004. ISBN 1-58488-360-X.
- [5] BERNERS-LEE, T. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, RFC Editor, July 1999. Dostupné z: <https://tools.ietf.org/html/rfc2616>.
- [6] BERNERS-LEE, T. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986, RFC Editor, January 2005. Dostupné z: <https://tools.ietf.org/html/rfc3986>.
- [7] BERNERS-LEE, T. *Web Services - Program Integration across Application and Organization boundaries* [online]. aug 2009. [cit. 2020-05-01]. Dostupné z: <https://www.w3.org/DesignIssues/WebServices.html>.
- [8] BRADA, P. – JEZEK, K. Repository and Meta-Data Design for Efficient Component Consistency Verification. *Science of Computer Programming*. 2015, 97, part 3, s. 349–365. ISSN 0167-6423. doi: 10.1016/j.scico.2014.06.013. Dostupné z: <http://www.sciencedirect.com/science/article/pii/S0167642314002925>.
- [9] BRADA, P. – VALENTA, L. Practical Verification of Component Substitutability Using Subtype Relation. s. 38 – 45, 10 2006. doi: 10.1109/EUROMICRO.2006.50.
- [10] DAIGNEAU, R. *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services*, From Objects to Web Services. Addison-Wesley, 2011.

- [11] FELLEISEN, M. *How To Design Programs*. The MIT Press, 2001. ISBN 978-0262062183.
- [12] FIELDING, R. T. Architectural Styles and the Design of Network-based Software Architectures [online]. Disertační práce, University of California, Irvine, 2000 [cit. 2020-02-22]. Dostupné z: <https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>.
- [13] FOWLER, M. *Richardson Maturity Model* [online]. [cit. 2020-05-01]. Dostupné z: <https://www.martinfowler.com/articles/richardsonMaturityModel.html>.
- [14] HESSOVÁ, G. Automatické získání historických údajů z webových zdrojů [online]. Bakalářská práce, Západočeská univerzita v Plzni, Fakulta aplikovaných věd, Plzeň, 2015 [cit. 2020-02-22]. Dostupné z: <https://theses.cz/id/pzbgj7/>.
- [15] HOWE, D. *type* [online]. 12 2003. [cit. 2020-05-01]. Dostupné z: <http://foldoc.org/type>.
- [16] JAMSHIDI, P. et al. Microservices: The Journey So Far and Challenges Ahead. *IEEE Software*. 2018, 35, 3, s. 24–35.
- [17] MANDEL, L. *Describe REST Web services with WSDL 2.0* [online]. [cit. 2020-05-01]. Dostupné z: <https://www.ibm.com/developerworks/webservices/library/ws-restwsdl/>.
- [18] NELSON, B. J. *Remote Procedure Call*. PhD thesis, Palo Alto Research Center, USA, 5 1981. Dostupné z: <https://dl.acm.org/doi/book/10.5555/910306>.
- [19] ORACLE. *Object-Oriented Programming Concepts* [online]. [cit. 2020-05-01]. Dostupné z: <https://docs.oracle.com/javase/tutorial/java/concepts/object.html>.
- [20] PEJŘIMOVSKÝ, D. Vytváření a ukládání popisu webových služeb v úložišti CRCE [online]. Diplomová práce, Západočeská univerzita v Plzni, Fakulta aplikovaných věd, Plzeň, 2015 [cit. 2020-02-22]. Dostupné z: <https://theses.cz/id/bb74eq/>.
- [21] PIERCE, B. C. *Types and Programming Languages*. The MIT Press, 2002. ISBN 0-262-16209-1.
- [22] RICHARDSON, L. *RESTful Web Services*. O'Reilly, 2007. ISBN 978-0-596-52926-0.

- [23] W3C. *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)* [online]. 4 2007. [cit. 2020-05-01]. Dostupné z: <https://www.w3.org/TR/soap12-part1/>.
- [24] W3C. *Web Services Architecture* [online]. feb 2004. [cit. 2020-05-01]. Dostupné z: <http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/>.
- [25] W3C. *Web Application Description Language* [online]. aug 2009. [cit. 2020-05-01]. Dostupné z: <https://www.w3.org/Submission/wadl/>.
- [26] W3C. *Web Services Description Language (WSDL) 1.1* [online]. mar 2001. [cit. 2020-05-01]. Dostupné z: <https://www.w3.org/TR/2001/NOTE-wsdl-20010315>.
- [27] W3C. *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language* [online]. jun 2007. [cit. 2020-05-01]. Dostupné z: <https://www.w3.org/TR/wsdl/>.
- [28] WIKIPEDIA. *JSON-WSP* [online]. april 2020. [cit. 2020-05-01]. Dostupné z: <https://en.wikipedia.org/wiki/JSON-WSP>.
- [29] WOOD, T. *How are REST APIs versioned?* [online]. sep 2014. [cit. 2020-05-01]. Dostupné z: <http://www.lexicalscope.com/blog/2012/03/12/how-are-rest-apis-versioned/>.
- [30] XINYANG FENG – JIANJING SHEN – YING FAN. REST: An alternative to RPC for Web services architecture. In *2009 First International Conference on Future Information Networks*, s. 7–10, 2009.

Seznam zkratek

API	Application Programming Interface
CRCE	Component Repository supporting Compatibility Evaluation
CSV	Comma Separated Values
HATEOAS	Hypertext As The Engine Of Application State
HTTP	Hypertext Transfer Protocol
JAR	Java Archive
JSON	JavaScript Object Notation
JSON-RPC	JSON Remote Procedure Call
JSON-WSP	JSON Web-Service Protocol
OAS	OpenAPI Specification
OSGi	Open Services Gateway initiative
RAML	RESTful API Modeling Language
REST	Representational State Transfer
RPC	Remote Procedure Call
SOAP	Simple Object Access Protocol
UML	Unified Modeling Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
WADL	Web Application Description Language
WSDL	Web Services Description Language
XML	Extensible Markup Language
XSD	XML Schema Definition
YAML	YAML Ain't Markup Language

A WSDL webové služby pro komix Dilbert

```
<?xml version="1.0" encoding="utf-8"?>
<wsdl:definitions
  xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
  xmlns:tns="http://gcomputer.net/webservices/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:s="http://www.w3.org/2001/XMLSchema"
  xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
  xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
  targetNamespace="http://gcomputer.net/webservices/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">

  <wsdl:types>
    <s:schema elementFormDefault="qualified"
      targetNamespace="http://gcomputer.net/webservices/">
      <s:element name="TodaysDilbert">
        <s:complexType />
      </s:element>
      <s:element name="TodaysDilbertResponse">
        <s:complexType>
          <s:sequence>
            <s:element
              minOccurs="0" maxOccurs="1"
              name="TodaysDilbertResult" type="s:string" />
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="DailyDilbert">
        <s:complexType>
          <s:sequence>
            <s:element
              minOccurs="1" maxOccurs="1"
              name="ADate" type="s:dateTime" />
          </s:sequence>
        </s:complexType>
      </s:element>
    </s:schema>
  </wsdl:types>
</wsdl:definitions>
```

```

        </s:sequence>
    </s:complexType>
</s:element>
    <s:element name="DailyDilbertResponse">
        <s:complexType>
            <s:sequence>
                <s:element minOccurs="0" maxOccurs="1"
                    name="DailyDilbertResult" type="s:string" />
            </s:sequence>
        </s:complexType>
    </s:element>
</s:schema>
</wsdl:types>

<wsdl:message name="TodaysDilbertSoapIn">
    <wsdl:part name="parameters"
        element="tns:TodaysDilbert" />
</wsdl:message>
<wsdl:message name="TodaysDilbertSoapOut">
    <wsdl:part name="parameters"
        element="tns:DilbertResponse" />
</wsdl:message>
<wsdl:message name="DailyDilbertSoapIn">
    <wsdl:part name="parameters"
        element="tns:DailyDilbert" />
</wsdl:message>
<wsdl:message name="DailyDilbertSoapOut">
    <wsdl:part name="parameters"
        element="tns:DilbertResponse" />
</wsdl:message>

<wsdl:portType name="DilbertSoap">
    <wsdl:operation name="TodaysDilbert">
        <wsdl:input message="tns:TodaysDilbertSoapIn" />
        <wsdl:output message="tns:TodaysDilbertSoapOut" />
    </wsdl:operation>
    <wsdl:operation name="DailyDilbert">
        <wsdl:input message="tns:DailyDilbertSoapIn" />
        <wsdl:output message="tns:DailyDilbertSoapOut" />
    </wsdl:operation>

```

```

</wsdl:portType>

<wsdl:binding name="DilbertSoap" type="tns:DilbertSoap">
  <soap:binding
    transport="http://schemas.xmlsoap.org/soap/http" />
  <wsdl:operation name="TodaysDilbert">
    <soap:operation
      soapAction="http://gcomputer.net/webservices/TodaysDilbert"
      style="document" />
    <wsdl:input>
      <soap:body use="literal" />
    </wsdl:input>
    <wsdl:output>
      <soap:body use="literal" />
    </wsdl:output>
  </wsdl:operation>

  <wsdl:operation name="DailyDilbert">
    <soap:operation
      soapAction="http://gcomputer.net/webservices/DailyDilbert"
      style="document" />
    <wsdl:input>
      <soap:body use="literal" />
    </wsdl:input>
    <wsdl:output>
      <soap:body use="literal" />
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>

<wsdl:binding name="DilbertSoap12" type="tns:DilbertSoap">
  <soap12:binding
    transport="http://schemas.xmlsoap.org/soap/http" />
  <wsdl:operation name="TodaysDilbert">
    <soap12:operation
      soapAction="http://gcomputer.net/webservices/TodaysDilbert"
      style="document" />
    <wsdl:input>
      <soap12:body use="literal" />
    </wsdl:input>
  </wsdl:operation>
</wsdl:binding>

```

```

        <wsdl:output>
        <soap12:body use="literal" />
        </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="DailyDilbert">
        <soap12:operation
            soapAction="http://gcomputer.net/webservices/DailyDilbert"
            style="document" />
        <wsdl:input>
        <soap12:body use="literal" />
        </wsdl:input>
        <wsdl:output>
        <soap12:body use="literal" />
        </wsdl:output>
    </wsdl:operation>
</wsdl:binding>

<wsdl:service name="Dilbert">
    <wsdl:port name="DilbertSoap" binding="tns:DilbertSoap">
        <soap:address
            location="http://www.gcomputer.net/webservices/dilbert.asmx"/>
    </wsdl:port>
    <wsdl:port name="DilbertSoap12" binding="tns:DilbertSoap12">
        <soap12:address
            location="http://www.gcomputer.net/webservices/dilbert.asmx"/>
    </wsdl:port>
</wsdl:service>
</wsdl:definitions>

```

B Výčet metadat služeb získaných z implementace

Název elementu	Typ elementu	Popis
name	Attribute	Název endpointu
path	Attribute	Relativní cesta k endpointu
method	Attribute	Seznam HTTP metod endpointu
consumes	Attribute	Seznam přijímaných MIME typů
produces	Attribute	Seznam vrácených MIME typů
requestBody	Property	Tělo requestu
requestParameter	Property	Parametr endpointu
response	Property	Detail odpovědi endpointu
responseParameter	Property	Parametr odpovědi

Tabulka B.1: Metadata indexovaná pro endpoint

Název elementu	Typ elementu	Popis
type	Attribute	Datový typ těla
isArray	Attribute	Příznak určující zda je tělo kolekce
isOptional	Attribute	Příznak určující povinnost těla

Tabulka B.2: Metadata indexovaná pro tělo requestu

Název elementu	Typ elementu	Popis
name	Attribute	Název parametru
dataType	Attribute	Datový typ parametru
category	Attribute	Kategorie parametru (PATH, FORM, ...)
isArray	Attribute	Příznak určující zda je parametr kolekce
defaultValue	Attribute	Výchozí hodnota parametru
isOptional	Attribute	Příznak určující povinnost parametru

Tabulka B.3: Metadata indexovaná pro parametr endpointu

Název elementu	Typ elementu	Popis
id	Attribute	Id odpovědi
isArray	Attribute	Příznak určující zda je odpověď kolekce
dataType	Attribute	Datový typ odpovědi
status	Attribute	Číslo vráceného HTTP statusu

Tabulka B.4: Metadata indexovaná pro odpověď

Název elementu	Typ elementu	Popis
id	Attribute	Id odpovědi
name	Attribute	Název parametru
isArray	Attribute	Příznak určující zda je parametr kolekce
dataType	Attribute	Datový typ parametru
category	Attribute	Kategorie parametru (PATH, FORM, ...)

Tabulka B.5: Metadata indexovaná pro parametr odpovědi

C Výčet metadat služeb získaných z popisu

Formát JSON-WSP

Formát WADL

Formát WSDL