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

# Diplomová práce

# Popis procesních vzorů pro jejich automatizovanou detekci

Plzeň 2020 Bc. Lenka Šimečková

**Místo této strany bude zadání práce.**

# Prohlášení

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

V Plzni dne 21. května 2020

Bc. Lenka Šimečková

# Abstract

**Process patterns definition enabling their automated detection**

This thesis is concerned with anti-patterns in software development and their modelling utilising the EPF Composer tool. A set of two plug-ins was created for exporting these models as SQL queries. The resulting anti-pattern export can be used to identify the anti-pattern in project data.



# Abstrakt

Tato diplomová práce se zabývá anti-patterny ve vývoji software a jejich modelováním v rámci nástroje EPF Composer. Pro tyto modely je vytvořena sada dvou rozšiřujících modulů, které slouží k jejich exportu do podoby SQL dotazů. Výsledný export anti-patternu je použitelný pro jeho identifikaci v projektových datech.

# Poděkování

Ráda bych poděkovala doc. Ing. Přemyslu Bradovi, MSc., Ph.D. a Ing. Petru Píchovi za odborné rady a cenné připomínky, které mi pomohly tuto diplomovou práci vypracovat.

# Obsah

# 1 Úvod

Při vývoji software dochází ke vzniku chyb, které vedou k nekvalitnímu produktu případně selhání projektu. Většina těchto chyb, které se v projektech často opakují, jsou nazývány anti-patterny. Aby k těmto chybám nedocházelo, jsou projekty vedeny podle metodik, které specifikují vhodné postupy vedoucí k vyšší pravděpodobnosti vytvoření kvalitního software. K popisu těchto procesů lze využít různých aplikací, například Eclipse Process Framework (EPF) Composer.

Katedra informatiky a výpočetní techniky (KIV) Fakulty aplikovaných věd (FAV) Západočeské univerzity v Plzni (ZČU) za účelem identifikace anti-patternů v projektových datech vyvinula nástroj SPADe (Software Process Anti-pattern Detector). Ten z vybraných Application Lifecycle Management (ALM) nástrojů získává data o projektech, které následně uloží ve vhodném formátu do datového skladu a umožňuje jejich analýzu.

Tato práce má za cíl navrhnout vhodný způsob popisu jednotlivých anti-patternů v nástroji EPF Composer a implementovat pro tento nástroj možnost jejich exportu v takovém formátu, aby bylo možné takto exportované popisy využít pro detekci anti-patternů v datovém skladu SPADe.

# 2 Softwarové procesy vývoje

Tato kapitola má za cíl obecně definovat softwarové procesy a uvést základní používané přístupy a metodiky, což je důležité pro pochopení kontextu práce.

Softwarový proces vývoje je strukturovaná množina aktivit, artefaktů, zdrojů, rolí a podmínek, které vedou k produkci softwarového systému. Použití vhodného procesu na projektu je důležité, protože snižuje pravděpodobnost vzniku chyb a tedy přispívá ke kvalitnějšímu výsledku. Každý proces je definován pomocí čtyř základních konceptů:

- projekt
- úkol
- role
- výsledek práce

Projekt je konkrétní instancí softwarového procesu, případně jeho části. Má určený cíl, finální produkt, který je v rámci tohoto projektu vyvíjen. I přesto, že je instancí softwarového procesu, může si jej v závislosti na daném kontextu přizpůsobit.

Úkol je jednotka práce, která může být přidělena určité osobě. Je většinou odvozen z plánovaných činností v rámci projektu, například analýza modulu nebo schůzka se zákazníkem. Časový odhad úkolu se obvykle pohybuje v řádu hodin až dnů. Jeden úkol by měl být atomický a neměl měnit větší množství výsledků práce [1].

Role je množina zodpovědností, dovedností nebo kompetencí, které se vztahují k určité osobě. Příkladem role může být například vývojář, analytik nebo tester. Platí, že jedna role může označovat vícero osob (v týmu je většinou vícero analytiků nebo vývojářů), osoba může mít více rolí zároveň (například vývojář a tester; typické především pro malé projekty) a stejně tak může jedna role označovat i skupinu osob (typicky vývojový tým). Mezi rolemi a osobami je tedy vazba *M:N* [1].

Výsledek práce je prvek, který je vstupem nebo výstupem úkolu nebo prvek, který se v rámci úkolu mění. Může mít různé podoby, od náčrtků po vysoce formální dokumenty určené zákazníkovi. Formální výstupy práce jsou nazývány artefakty [1].

Model softwarového procesu je jeho abstraktním popisem z určité perspektivy. Slouží ke zjednodušenému popisu jednotlivých vývojových procesů. Vzhledem k tomu, že všechny modely proces znázorňují z určité perspektivy, neobsahují o daném procesu veškeré informace [2].

Z jednotlivých modelů jsou standardizací a dalším popisem tvořeny metodiky. Ty jsou určeny ke konkretizaci v rámci použitého projektového kontextu. Definují tedy, jak má proces vypadat, specifikují jeho fáze, úkoly, role, výsledky práce a podobně.

V následujících podkapitolách budou popsány základní třídy metodik a jejich významní zástupci.

## 2.1 Sekvenční metodiky

Sekvenční techniky jsou specifické tím, že definují etapy, během nichž jsou vykonávány činnosti pouze určitého aspektu projektu, jako například analýza nebo implementace.

Vyznačují se tím, že všechny etapy vývoje jsou prováděny v daném pořadí s minimálním nebo žádným množstvím iterací a neměly by se navzájem překrývat. Je tedy žádoucí tyto etapy pečlivě naplánovat ještě před jejich začátkem. Výstupem každé etapy je jeden nebo více výsledků práce. Tyto výstupy poté slouží jako vstupy dalších etap. Nevýhodou je nedostatečná schopnost reagovat na změny v rámci vývoje.

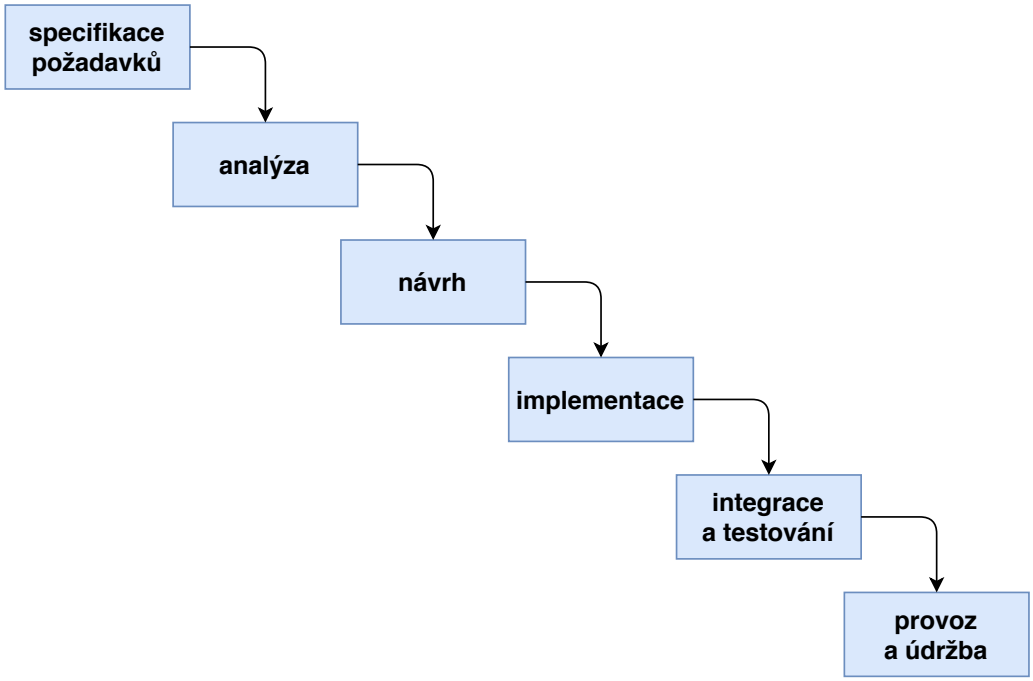
Typickým zástupcem sekvenčních metodik je vodopádový model [2]. Tento model je znázorněn na obrázku 2.1.

## 2.2 Iterativní metodiky

Iterativní metodiky odstraňují nevýhody sekvenčních metodik, u kterých byly funkční i akceptační testy jedny z posledních etap. Iterativní metodiky definují proces jako sekvenci iterací, v rámci nichž probíhají činnosti týkající se všech disciplín. Model jedné iterace je znázorněn na obrázku 2.2. Výsledkem každé iterace je nějaký přírůstek produktu, který je v rámci té samé iterace konzultován se zákazníkem. S případnými chybami nebo doplňujícími požadavky je tak možné se vypořádat již v raných fázích projektu.

### 2.2.1 Unified Process

Unified Process (UP) je zkráceninou z původního názvu Unified Software Development Process. Je základním zástupcem iterativních metod. Jedná se




Obrázek 2.1: Vodopádový model [3]

o generickou metodiku, u které se očekává, že bude dále upravována na míru konkrétnímu projektu.

Celým projektem v různé míře prostupují skupiny souvisejících aktivit a artefaktů, nazývané disciplíny. Unified Process definuje celkem šest disciplín: [5]

- **Business Modeling** (modelování businessu) – modelování systémových procesů
- **Requirements** (požadavky) – analýza požadavků (funkčních i mimofunkčních), určení případů užití
- **Analysis & Design** (analýza a návrh) – návrh architektury
- **Implementation** (vývoj) – vývoj systému
- **Test** (testování) – plánování jednotlivých testů, testování (regresní, akceptační testování, . . . )
- **Deployment** (nasazení) – nasazení vyvinutého systému

Unified Process dělí projekt do čtyř fází, z nichž každá obsahuje jednu nebo více iterací [5] [6]:


Obrázek 2.2: Model iterace [4]

- **Inception** (zahájení) – příprava podkladů k projektu, úvodní naplánování, určení rozsahu a klíčových požadavků, identifikace rizik
- **Elaboration** (rozpracování) – určení většiny požadavků, návrh architektury, naplánování fáze Construction, analýza rizik
- **Construction** (konstrukce) – časově nejnáročnější fáze, dokončení vyvíjeného produktu
- **Transition** (předání) – předání systému koncovým uživatelům

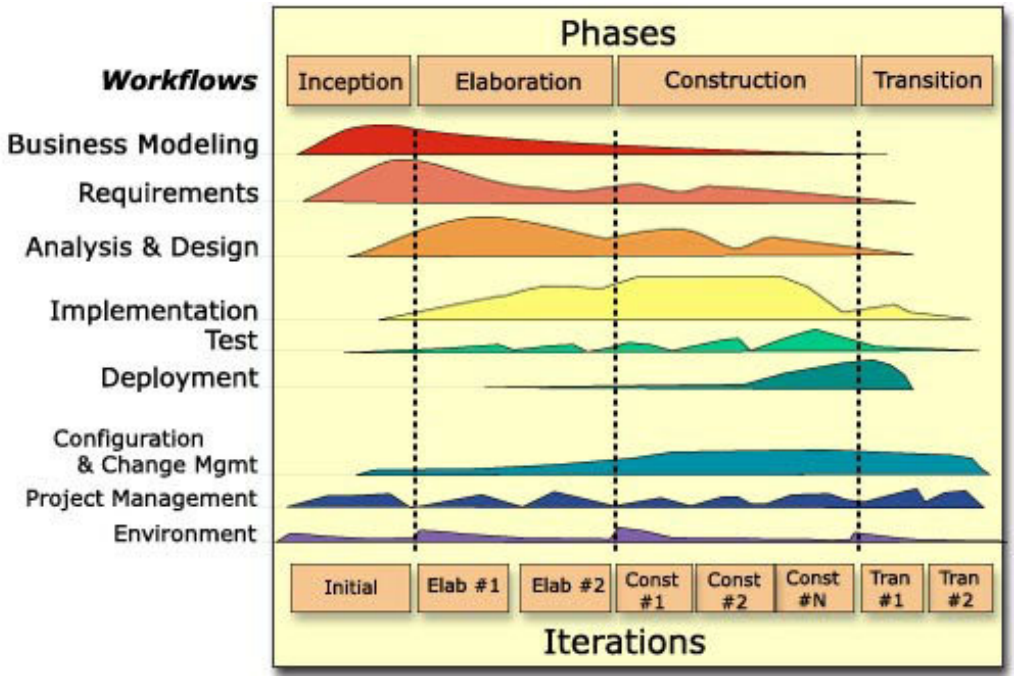
Každá fáze je ukončena milníkem, který definuje cíle, které musí být s koncem fáze splněny.

### 2.2.2 Rational Unified Process

Rational Unified Process (RUP) byl vytvořen IBM jako implementace Unified Processu. Stejně jako UP je RUP konfigurovatelný a předpokládá se, že každý projekt implementuje RUP podle vlastních potřeb. Disciplíny, které definuje UP jsou v RUP označeny jako klíčové. Mimo ně definuje tři pomocné disciplíny [7]:

- **Configuration & Change Management** (řízení verzí a změn) – správa změnových požadavků, vytvoření patřičných procesů
- **Project Management** (řízení projektu) – plánování projektu včetně časových odhadů, řízení rizik, vedení lidí
- **Environment** (prostředí) – nastavení a správa jednotlivých prostředí

Schéma tohoto procesu je znázorněno na obrázku 2.3.




Obrázek 2.3: Schéma Rational Unified Processu [8]

### 2.2.3 Enterprise Unified Process

Enterprise Unified Process (EUP) je další implementací Unified Processu. Vznikl s cílem odstranit nedostatky Rational Unified Processu, který se mimo jiné nezabývá podobou procesu po nasazení.

Oproti RUP specifikuje dvě nové fáze [9]:

- **Production** (provoz) – servis běžícího systému
- **Retirement** (vyřazení) – vyřazení běžícího systému, konec jeho podpory, případně výměna za nový systém

Mimo to specifikuje navíc osm nových disciplín [9]:

- **Operations and Support** (provoz a podpora) – zajištění provozu a podpora produkčního prostředí
- **Enterprise Business Modeling** (podnikové modelování businessu) – přizpůsobení agilních technik potřebám projektu nebo organizace
- **Portfolio Management** (řízení portfolia[1]) – rozšiřování používané iterativní nebo agilní metodiky takovým způsobem, aby se zvýšila celková efektivita všech projektů organizace

[1]softwarové portfolio je soubor dokončených i probíhajících projektů

- **Enterprise Architecture** (podniková architektura) – určení podnikové architektury, soubor referenčních architektur a prototypů
- **Strategic Reuse** (strategie znovupoužití) – aktivity pro znovupoužití artefaktů s cílem rychlejšího a kvalitnějšího vývoje aplikací
- **People Management** (řízení lidských zdrojů) – organizace, sledování, vedení a motivace zaměstnanců
- **Enterprise Administration** (podniková administrativa) – údržba a podpora technické a vývojářské infrastruktury (správa dat, sítě, nástrojů, ...)
- **Software Process Improvement** (vylepšování softwarového procesu) – volba vhodného procesu pro různé typy projektů, úprava a správa těchto procesů

### 2.2.4 OpenUP

OpenUP (Open Unified Process) je produktem Eclipse Foundation. Vznikl odvozením z Unified Processu a Rational Unified Processu. Narozdíl od ostatních Unified Processů obsahuje agilní prvky, jako například důraz na malé přírůstky produktu v průběhu projektu nebo méně formální artefakty [10].

## 2.3 Agilní metodiky

Agilní metodiky částečně vychází z iterativních metodik, z kterých převzaly koncept iterací. Iterace jsou krátké a mají přesné časové vymezení. Na konci každé z nich se očekává přírůstek k projektu. Důraz u agilních metodik je kladen na schopnost přizpůsobovat se změnám, které přicházejí v různých fázích projektu.

*Manifest pro agilní vývoj software* z roku 2001 vymezuje čtyři základní pravidla agilního vývoje [11]:

- jednotlivci a interakce mají přednost před procesy a nástroji
- fungující software má přednost před vyčerpávající dokumentací
- spolupráce se zákazníkem má přednost před vyjednáváním o smlouvě
- reagování na změny má přednost před dodržováním plánu

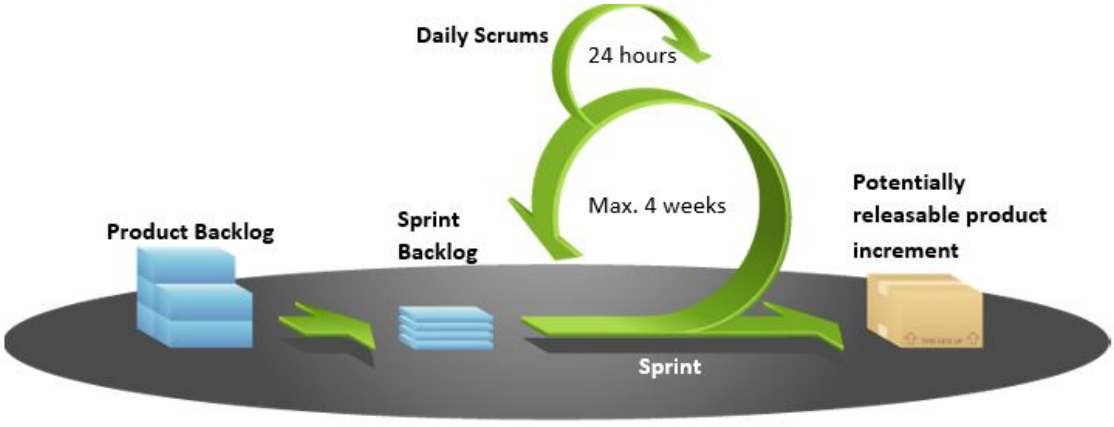
### 2.3.1 Scrum

Scrum je agilní metodika, která byla poprvé definována již v roce 1995 Kenem Schwaberem a Jeffem Sutherlandem. Model tohoto procesu je na obrázku 2.4. Oproti základní definici agilních technik navíc scrum definuje několik konceptů.

Iterace se nazývá sprint. Každý sprint má vlastní backlog, který specifikuje úkoly, které mají být během sprintu splněny. Každý sprint je plánován spolu se zákazníkem v rámci tzv. planning meetingu, kde se určuje právě backlog a probíhá další plánování. Sprint končí schůzkou se zákazníkem, která se nazývá sprint review a je na ní zákazníkovi prezentován přírůstek od minulého sprintu. Po skončení sprintu následuje retrospektiva, na které je proběhlý sprint sebekriticky zhodnocen a v případě nedostatků jsou naplánována taková opatření, aby se tyto nedostatky v dalším sprintu neopakovaly [12].

Dále Scrum definuje tři důležité role v rámci projektu, a to vývojový tým, vlastník produktu a scrum master. Vývojový tým se vyznačuje tím, že z vnějšku vystupuje jako ucelená jednotka. Je jeho zodpovědností rozdělení konkrétních rolí (analytik, tester, ...) mezi jeho členy. Vlastník produktu je zástupcem stakeholderů, který dbá na přínos projektu pro jejich potřeby. Scrum master je osoba, která dohlíží na dodržování principů Scrumu.

Každý den, ideálně hned ráno, probíhá „daily scrum“, tedy krátká schůzka s cílem zhodnotit práci z předchozího dne a domluvit se na práci pro současný den. Účastní se jí členové vývojového týmu a scrum master [13].




Obrázek 2.4: Model scrumu [14]

# 3 Meta-modely procesů

Tato kapitola se zabývá meta-modely softwarových procesů vývoje. Popisuje především meta-modely SPEM a UMA a dále nástroj pro modelování procesů Eclipse Process Framework Composer, který bude v rámci této práce využit pro modelování anti-patternů.

Meta-modely softwarových procesů slouží ke strukturovanému popisu jednotlivých modelů. Jejich význam spočívá především ve standardizaci jejich popisů včetně specifikace základních procesních entit, jako jsou například role, úkoly nebo výsledky práce, a jejich vzájemných vztahů. Tyto popisy jsou uskutečňovány pomocí formalizovaných grafických notací, například Business Process Modeling and Notation (BPMN) nebo Unified Modeling Language (UML).

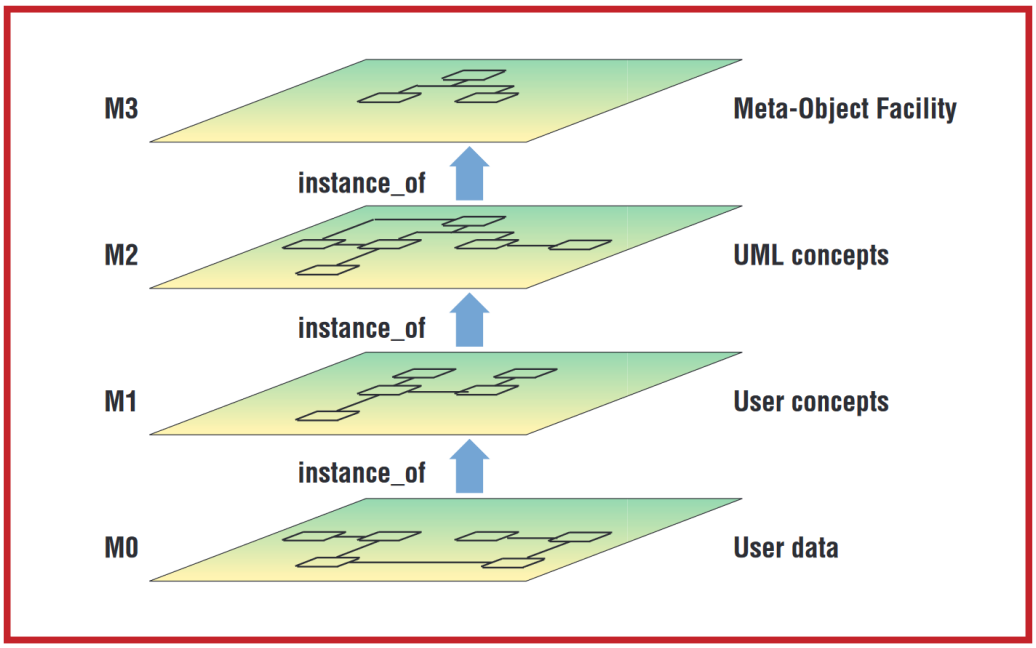
Společnost OMG pro svou rodinu modelovacích jazyků definovala v roce 1996 standard Meta Object Facility (MOF) [15]. Jedná se o meta-meta-model, tedy framework pro tvorbu meta-modelů, abstraktně specifikující jejich syntax.

Spolu s tímto meta-meta-modelem definovala OMG i formální hierarchii (meta-)modelů tak, jak je znázorněná na obrázku 3.1. Tato hierarchie je čtyřvrstvá, přičemž každá nižší vrstva odpovídá vrstvě vyšší. Nejvyšší vrstva, označovaná jako M3, představuje meta-meta-modely, především výše zmíněný MOF [16]. Nižší vrstva M2 představuje jednotlivé meta-modely. Příkladem může být UML, jeden z nejpoužívanějších jazyků pro specifikaci modelů spravovaný právě OMG. Vrstva M1 definuje jednotlivé modely dat, které přináleží nejnižší vrstvě M0 [16].

UML, náležící do vrstvy M2 výše popsané hierarchie, je široce využitelným jazykem, který lze využít pro modelování nebo dokumentování různých aspektů softwarových systémů a dále vytváření různých typů strukturálních a behaviorálních diagramů.

BPMN je dalším meta-modelem vytvořeným společností Business Process Management Initiative (BPMI), která byla později sloučena s OMG. Meta-model BPMN slouží k definování obchodního procesu takovým způsobem, aby byl pochopitelný i pro koncové uživatele. Srozumitelnost pro všechny zúčastněné stakeholdery je také definována jako jeho hlavní cíl [17]. Definuje koncepty jako jsou plány aktivit, informace v kontextu procesu nebo alokace rolí [18].

Oproti tomu například meta-model Eclipse Modeling Framework (EMF), vytvořený nadací Eclipse, má využití mimo jiné v jádře Eclipse aplikací.




Obrázek 3.1: Hierarchie (meta-)modelů podle OMG [16]

Skládá se ze dvou částí – *ecore*, sloužící k tvorbě datového modelu, a *genmodel*, která zajišťuje generování zdrojového kódu na základě vytvořeného modelu [19] [20].

## 3.1 Software & Systems Process Engineering Meta-Model

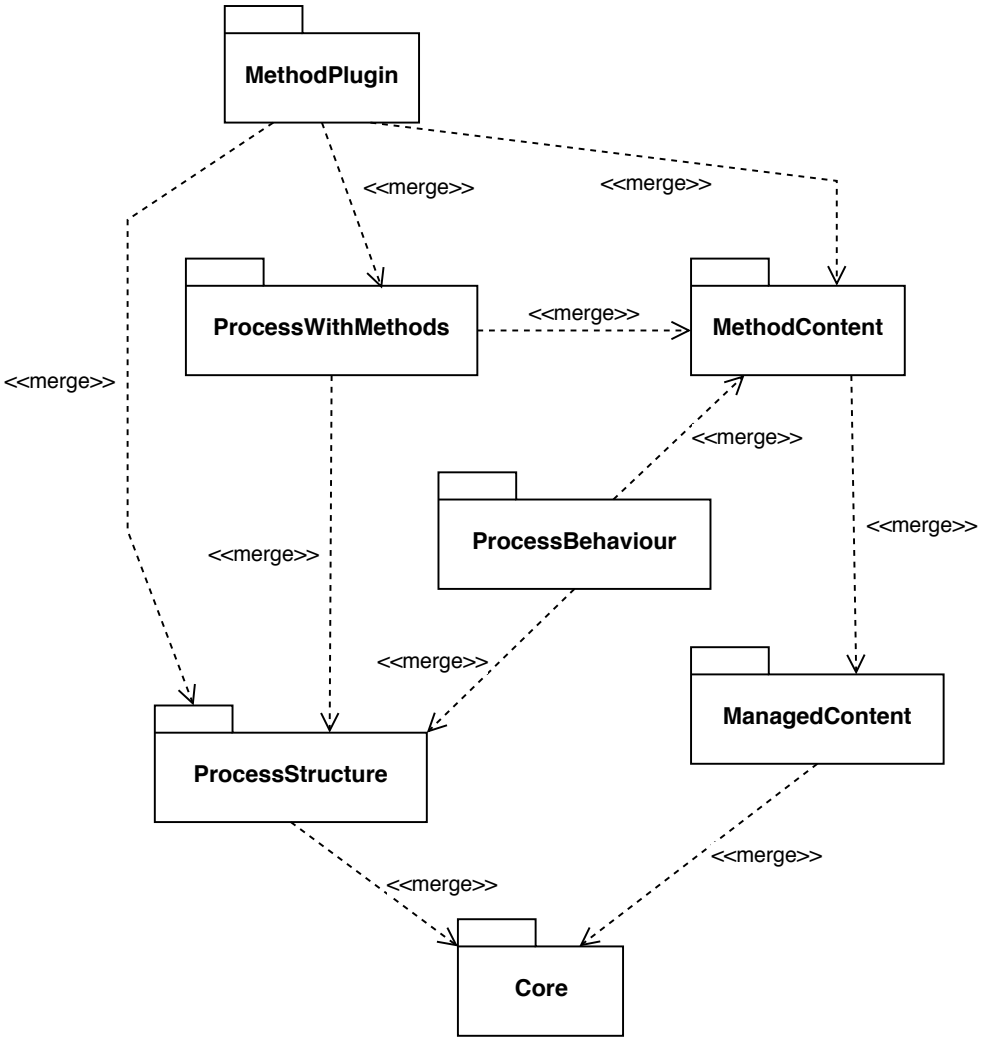
Software & Systems Process Engineering Meta-Model (SPEM) je meta-model vytvořený společností OMG [21]. Je standardem pro popis softwarových procesů [22]. V rámci hierarchie (meta-)modelů OMG se nachází na vrstvě M2 a implementuje tedy standard MOF, konkrétně jeho verzi 2.0 [21]. Zároveň je profilem UML.[1]

Existují jeho dvě formální verze, a to 1.1, vydaná v roce 2002, a 2.0, vydaná v roce 2008. V nové verzi byly provedeny opravy předchozí verze a přidány nové možnosti modelování. Jedním z hlavních rozdílů je, že zatímco SPEM 1.1 byl založen na UML ve verzi 1.4, SPEM 2.0 je založen na UML 2.0. Dále bude popisován pouze SPEM ve verzi 2.0.

Architektura SPEM je rozdělena do sedmi balíků, které jsou popsány níže

[1]Profil UML je rozšířením UML pro nějakou doménu.

[21]. Jejich struktura je znázorněna na obrázku 3.2.




Obrázek 3.2: Struktura SPEM 2.0 [21]

- **Core** (jádro) – Obsahuje abstrakce a třídy, které jsou společné všem ostatním balíkům. Obsahuje zejména element *Work Definition*, který slouží jako rodič pro všechny definice práce v rámci tohoto meta-modelu.
- **Process Structure** (struktura procesu) – Základ pro modelování procesů, umožňuje zanořování aktivit (*Activity*) a definování vztahů mezi nimi navzájem, výsledky prací v kontextu procesu (*Work Product Use*) a rolemi v kontextu procesu (*Role Use*).
- **Process Behavior** (chování procesu) – Doplnění struktury procesu o externě definované modely chování, jako je například UML 2.0 nebo diagramy aktivit.

- **Managed Content** (řízení obsahu) – Podpora pro slovně psanou dokumentaci. Umožňuje jednotlivým elementům přidávat popisné atributy jako například *Content Description* a definovat koncepty pro další popis procesů, tedy především *Guidiance*.
- **Method Content** (obsah metody) – Obsahuje elementy znalostní báze nezávislé na procesech, jako jsou úkoly (*Task*), role (*Role*) nebo výsledky práce (*Work Product*).
- **Process With Methods** (procesy s metodami) – Umožňuje vytvářet vazby mezi procesy a elementy obsahů metod.
- **Method Plug-in** (plug-in metody) – Poskytuje nástroje pro návrh a správu knihoven s obsahy metod a procesy.

Na obrázku 3.3 je příklad modelu vytvořeného pomocí SPEM 2.0. Konkrétně se jedná o jednu iteraci (sprint) metodiky Scrum, popsané v kapitole 2.3.1. Nejpeve je zadefinován úkol plánování sprintu, následován cyklem s denními úkoly. Sprint je na diagramu zakončen úkolem sprint review, po kterém následuje úkol s retrospektivou.

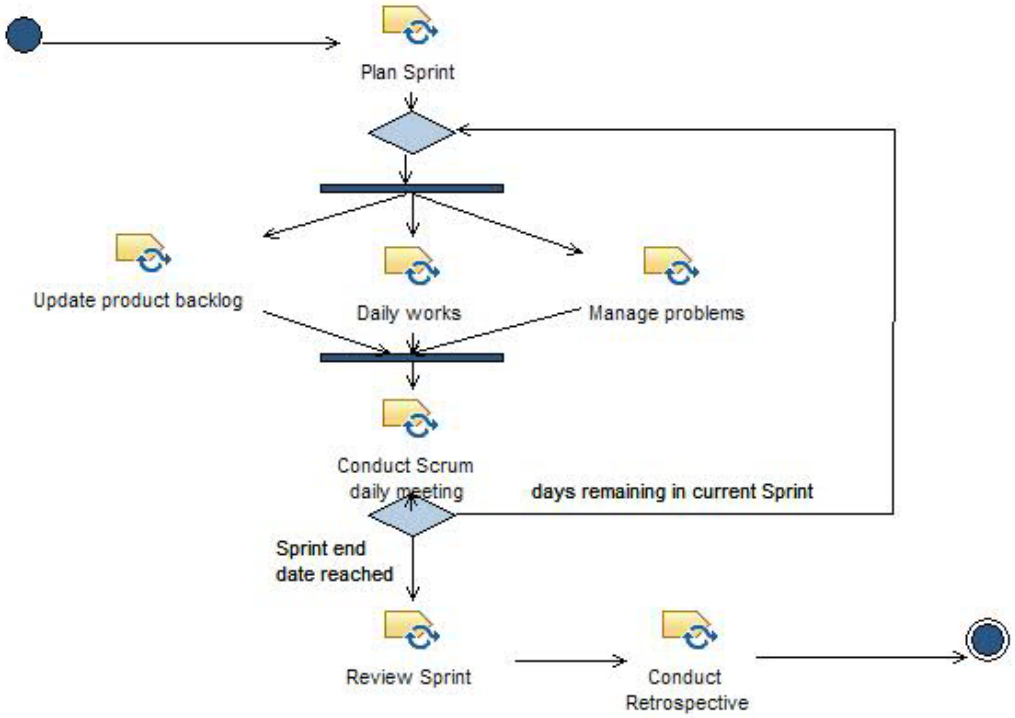
## 3.2 Unified Method Architecture

Unified Method Architecture (UMA) je meta-model vytvořený společností IBM s cílem sjednotit dosavadní schémata a terminologie procesního inženýrství [1]. Vychází z meta-modelu SPEM, který byl blíže popsán v kapitole 3.1. Původně byl založený na verzi SPEM 1.1, ale poté přijal změny SPEM 2.0. V budoucnu by měl být meta-modelem SPEM 2.0 kompletně nahrazen. Je používaný nástrojem pro modelování procesů EPF Composer, popsaném dále v kapitole 3.3.

Základní jednotkou UMA je knihovna (*library*). Ta obaluje plug-iny metod (*Method Plug-in*) a jejich společnou konfiguraci (*Configuration*).

Plug-in metody lze definovat jako soubor obsahu metody (*Method Content*) a procesů (*Process*). Oddělení těchto dvou typů obsahů je výsledkem snahy o dodržení principu *separation of concerns*.[2] V tomto konkrétním případě je oddělen obsah metody, který slouží jako znalostní báze standardizovaných podob jednotlivých elementů jako jsou role, úkoly nebo výsledky práce, od procesů, které jsou z těchto elementů postaveny a přistupují k nim jako k znovupoužitelným stavebním blokům [24].

[2]Princip *separation of concerns* je typ návrhu, při kterém je software rozdělen do komponent podle funkcionalit s co nejmenší mírou překrývání se.




Obrázek 3.3: Sprint metodiky Scrum vyjádřený pomocí meta-modelu SPEM 2.0 [23]

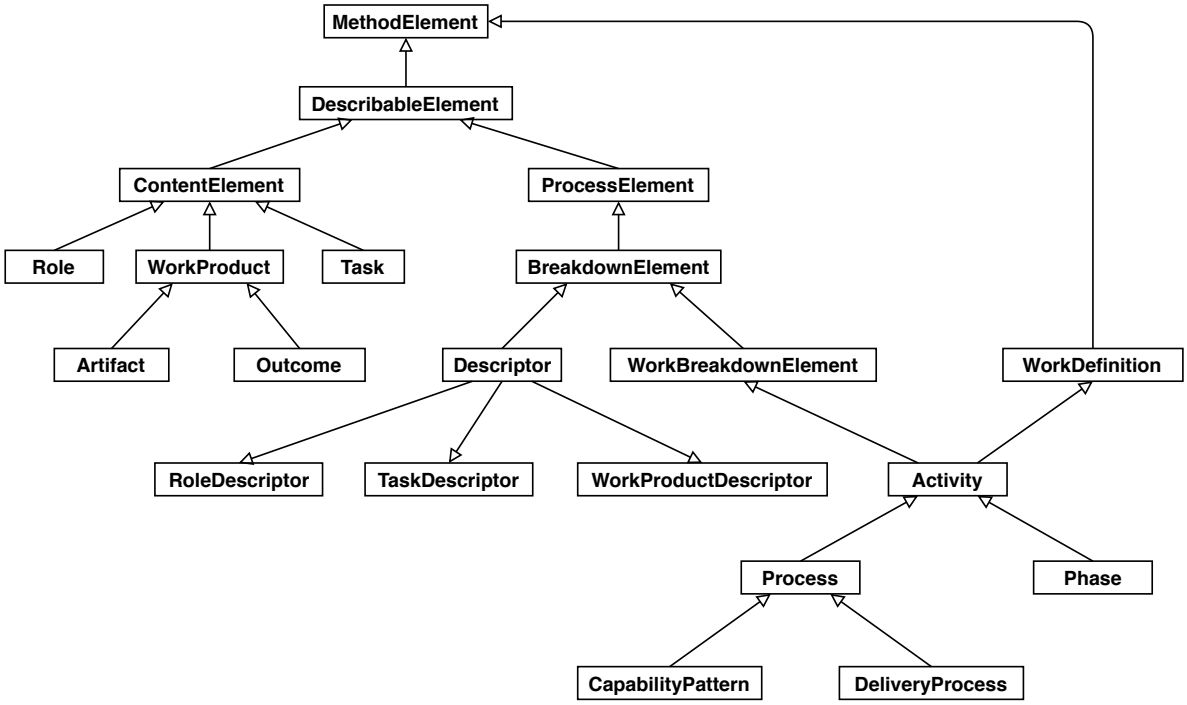
V tabulce 3.1 jsou shrnuty hlavní rozdíly mezi modely SPEM 2.0 a UMA. Diagram 3.4 zobrazuje dědičnost z pohledu této práce nejdůležitějších prvků UMA.

Tabulka 3.1: Rozdíly mezi SPEM 2.0 a UMA

| **SPEM 2.0** | **UMA** |
|---|---|
| Process Pattern | Capability Pattern |
| Guidance Kind | Guidance |
| Work Product Use | Work Product Descriptor |
| Role Use | Role Descriptor |
| Estimate | – |
| EstimationMetric | – |

### 3.2.1 Obsah metody

Obsah metody (*Method Content*) slouží k popisu tasků, artefaktů, rolí a dalších elementů. Tyto elementy nejsou přímo součástí popisu životního




Obrázek 3.4: Struktura dědičnosti vybraných prvků UMA

cyklu, slouží spíše jako elementy, na které je ze životního cyklu odkazováno. Na jeden element z obsahu metody lze z popisu životního cyklu odkazovat vícekrát.

Obsah metody lze rozdělit do těchto kategorií:

- jádro obsahu metody (*Core Method Content*)
- pomocný materiál (*Guidance*)
- obsahová kategorie (*Content Category*) [25]

### Jádro obsahu metody

Jádro obsahu metody obsahuje základní elementy procesu, a to **role** (*Role*), **úkoly** (*Task*) a **výsledky práce** (*Work Product*). Význam těchto tří základních elementů byl obecně popsán v kapitole 2.

UMA specifikuje tři typy výsledků práce – **artefakt** (*Artefact*), **výstup** (*Outcome*) a **předávatelný výsledek práce** (*Deliverable*). Mohou být vstupem, výstupem nebo předmětem úpravy vícero úkolů.

Artefakt představuje „hmatatelný“ výsledek práce, který má typicky nějakou formální definici. Často se tedy jedná o takové výsledky práce, pro

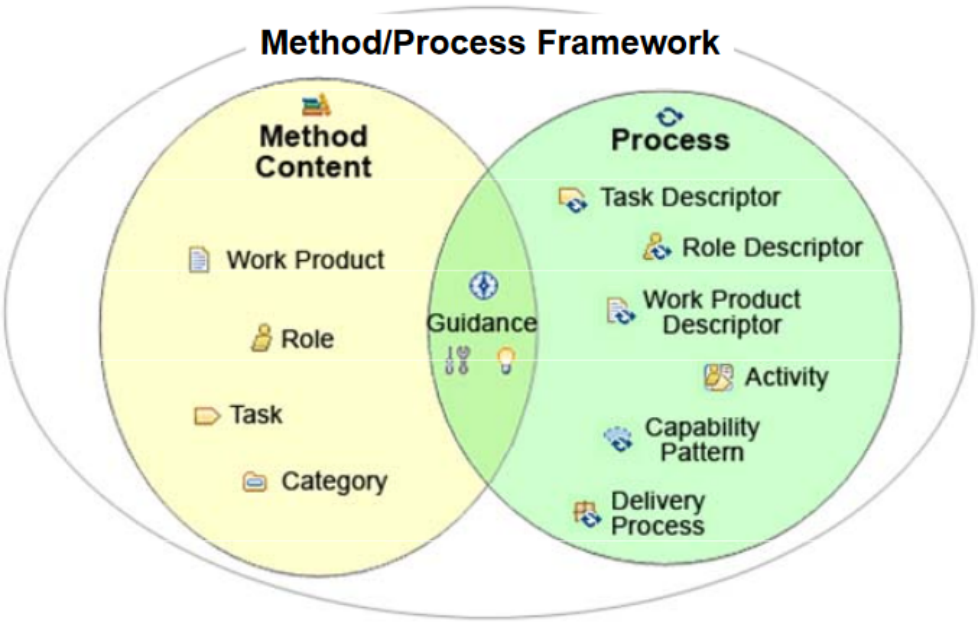
které lze vytvořit příklady nebo šablony. Zároveň artefakt může být složen z jiných artefaktů. Artefakt může mít podobu například formálního souboru, papírového dokumentu nebo Wiki stránky. Konkrétními příklady artefaktu může být dokument architektury nebo vize.

Výstup slouží spíše pro popis stavu, ve kterém se projekt má nacházet, případně takových výsledků práce, které nemají formální definici. Příkladem může být nakonfigurovaný server, zápis ze schůzky nebo i záznamy různých náčrtů na tabuli.

Jako předávatelný výsledek práce je označován takový výsledek práce, který může být dodán zákazníkovi nebo jinému stakeholderovi jako výstup projektu. Jedná se o kolekci výstupů práce, typicky artefaktů. Může se jednat ale i o výstup, například sadu konfiguračních souborů. Je to tedy výsledek práce, který má pro daného stakeholdera určitou hodnotu.

### Pomocný materiál

Pomocný materiál je souhrnné označení pro podpůrné elementy, které slouží pro doplnění vedlejších informací ostatním prvkům procesu. Někdy je pomocný materiál vnímán jako průnik mezi obsahem metody a procesy [24]. Tento přístup je znázorněn na obrázku 3.5. Důvodem je, že může vyjadřovat informace vztahující se jak k obsahu metody, tak k procesům. V rámci implementace UMA je však řazen do obsahu metody.




Obrázek 3.5: Pomocný materiál jako průnik mezi obsahem metody a procesy [24]

UMA celkem specifikuje čtrnáct typů pomocných materiálů:

- **seznam položek** (*Checklist*) – seznam položek, které je potřeba splnit
- **koncept** (*Concept*) – klíčové pojmy, které se pojí s odkazovaným elementem
- **příklad** (*Example*) – příklad výsledku práce nebo úkolu
- **návod** (*Guideline*) – detailní informace ke způsobu plnění daného úkolu
- **praktika** (*Practice*) – osvědčený způsob práce nebo dosažení výsledku, který má pozitivní vliv na kvalitu softwaru
- **hlášení** (*Report*) – automaticky vygenerovaný obsah výsledku práce
- **znovupoužitelná položka** (*Reusable Asset*) – předpřipravené řešení
- **rámcový harmonogram** (*Roadmap*) – lineární průchod komplexní činností
- **podpůrný materiál** (*Supporting Material*) – ostatní pomocný materiál
- **šablona** (*Template*) – udává formát výsledku práce
- **definice pojmu** (*Term Definition*) – definice pojmu, která je automaticky součástí slovníku
- **návod k nástroji** (*Tool Mentor*) – specifikuje způsob, jakým použít nějaký nástroj nebo provést nějakou aktivitu
- **informační dokument** (*Whitepaper*) – definice podobná konceptu, narozdíl od něj je v publikovatelné podobě

### Obsahová kategorie

Obsahová kategorie je někdy též nazývaná jako kategorie metody (*Method Category*). Označuje kategorie elementů. UMA specifikuje celkem pět základních kategorií, ale umožňuje i definování **vlastní kategorie** (*Custom Category*). Základní kategorie jsou následující:

- **disciplína** (*discipline*) – Sada úkolů, které náleží stejnému oboru činnosti jako například návrh nebo implementace. Vztahuje se k metodice RUP, popsané v kapitole 2.2.2.

- **doména** (*domain*) – Doména je sada výsledků práce, co do významu podobná disciplínám. Jedná se tedy o rozdělení podle oborů činnosti.
- **druh výsledku práce** (*work product kind*) – Jedná se o sadu výsledků práce pro jejich dělení podle charakteru a účelu. Příkladem druhu výsledku práce může být plán nebo specifikace.
- **sada rolí** (*role set*) – Představuje skupinu rolí, které jsou si navzájem v určitém ohledu podobné. Příkladem může být sada rolí „Analytikové“, do které by náleželi obchodní analytik nebo systémový analytik.
- **nástroj** (*tool*) – Slouží jako kategorie pro návody k nástrojům, popsané výše.

### 3.2.2 Procesy

Procesy (*Processes*) slouží k popisu životního cyklu, vymezení pořadí činností. Lze je obecně vyjádřit například pomocí pracovních postupů (*workflow*) nebo struktur rozkladů práce (*Breakdown Structure*). Ty obsahují odkazy (deskriptory) na elementy obsahu metod.

Eclipse Process Framework Composer nabízí dvě možnosti definování procesu – **proces životního cyklu** (*Delivery Process*) a **procesní vzor** (*Capability Pattern*). Procesy životního cyklu definují kompletní proces daného projektu, zatímco procesní vzory jsou určeny pro popis částí procesu životního cyklu, například osvědčených praktik (*best practices*) nebo určité domény.

V rámci těchto procesů lze definovat následující elementy:

- **fáze** (*Phase*) – významné období projektu, specifický typ aktivity, který se typicky váže se sadou předávatelných výsledků práce nebo milníkem
- **iterace** (*Iteration*) – soubor aktivit, které se opakují
- **milník** (*Milestone*) – významná událost, například dokončení předávatelného výsledku práce, ukončení fáze nebo učinění významného rozhodnutí
- **aktivita** (*Activity*) – hlavní stavební prvky procesů, soubory deskriptorů úkolů, výsledků prací a rolí a jiných aktivit
- **deskriptor úkolu** (*Task Descriptor*) – úkol v kontextu aktivity
- **deskriptor výsledku práce** (*Work Product Descriptor*) – výsledek práce v kontextu aktivity

- **deskriptor role** (*Role Descriptor*) – role v kontextu aktivity
- **profil týmu** (*Team Profile*) – slouží ke sdružení deskriptorů rolí a dalších profilů týmů, tvoří jejich hierarchii

**Diagramy**

UMA podporuje vizualizaci procesů v podobě několika různých typů diagramů: [26]

- **diagram pracovního postupu** (*Workflow Diagram*) – vysokoúrovňový pohled na pořadí a možnosti paralelizace aktivit, může obsahovat počáteční a koncový uzel, rozhodovací uzly, synchronizační lišty, aktivity, fáze, deskriptory úkolů
- **diagram detailu aktivity** (*Activity Detail Diagram*) – zobrazuje všechny role, úkoly a výstupy jedné aktivity, každá role má přehledně zobrazený seznam všech svých úkolů v rámci aktivity, každý úkol má znázorněný svůj výstup
- **diagram závislostí výsledků práce** (*Work-Product Dependency Diagram*) – pro danou aktivitu zobrazuje závislosti výsledků práce na dalších výsledcích práce

### 3.2.3 Konfigurace

Konfigurace (*Configuration*) je prvek společný pro skupinu plug-inů metod. Představuje výběr plug-inů určených k publikaci, například v podobě HTML stránek.

## 3.3 Eclipse Process Framework Composer

Eclipse Process Framework Composer (EPF Composer) je nástroj určený primárně pro popis softwarových procesů. Jedná se o otevřený („open-source“) software spravovaný organizací Eclipse Foundation. Obdobným nástrojem je Rational Method Composer (RMC) vyvíjený společností IBM, který je však komerční.

### 3.3.1 Funkce EPF Composeru

EPF Composer umožňuje modelování a správu softwarových procesů. Jedná se tedy o aplikaci, jejíž předpokládaní uživatelé jsou především pro-

jektoví manažeři, scrum masteři nebo vedoucí vývojových týmů. Jako meta-model těchto procesů využívá právě UMA, popsanou v kapitole 3.2.

Dále EPF Composer umožňuje konfiguraci procesu vyexportovat do HTML nebo jako Java EE aplikaci a tím i možnost jej zveřejnit na webu, což umožňuje všem členům vývojového týmu do procesu jednoduše nahlédnout, případně proces sdílet napříč vývojovými týmy.

Pro správu procesů nabízí uživatelské rozhraní EPF Composeru dvě hlavní perspektivy [27]:

- **authoring** – Slouží pro návrh a editaci obsahu metody i procesů. Zároveň nabízí i možnost náhledu části výsledné HTML stránky pro právě editovaný prvek.
- **browsing** – Poskytuje komplexní pohled na veškerou generovanou konfiguraci, ale bez možnosti editace.

Data projektů, vytvořených v EPF Composeru, jsou ukládána ve formě XMI souborů.[3]

### 3.3.2 Vnitřní struktura EPF Composeru

EPF se skládá ze souboru komponent (plug-inů). Obecný způsob jejich fungování je blíže popsán v kapitole 6. Následuje stručný přehled hlavních komponent, které jsou pro EPF Composer specifické.

#### *Common*

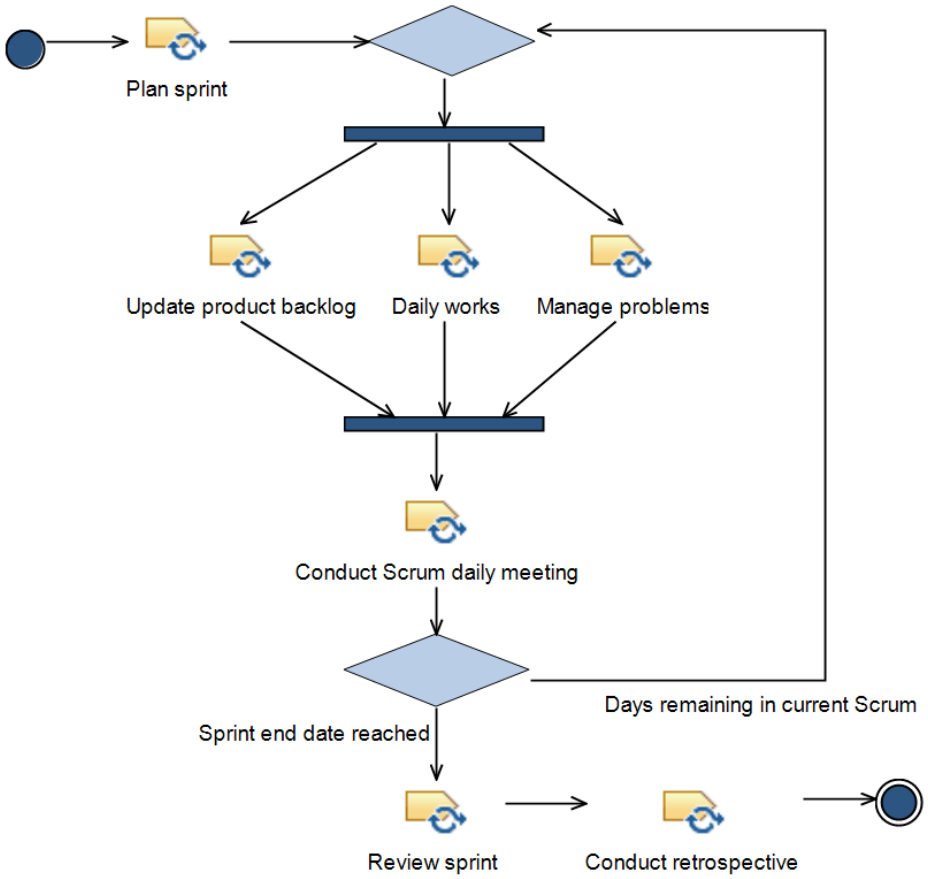
Komponenta Common poskytuje základní funkcionalitu pro všechny ostatní komponenty, jako například logování nebo zápis do a čtení ze souborů, parsování XML nebo logování.

#### *UMA*

Komponenta UMA zajišťuje manipulaci s jednotlivými elementy obsahu metody i procesů. Zároveň tvoří vnitřní datovou strukturu EPF Composeru podle stejnojmenného meta-modelu. Více je tento meta-model popsán v sekci 3.2.

Na obrázku 3.6 je znázorněný model sprintu z metodiky Scrum vytvořený pomocí EPF Composeru. Jeho vyjádření odpovídá právě meta-modelu UMA, který tento nástroj používá.

[3] XMI je specifikace vytvořená OMG pro ukládání metadat ve formátu XML.




Obrázek 3.6: Sprint metodiky Scrum vyjádřený pomocí meta-modelu UMA

#### Authoring

Komponenta Authoring zajišťuje vytváření jednotlivých elementů meta-modelu pomocí několikastránkových formulářových editorů. Zároveň poskytuje průvodce pro vytváření knihoven metod, plug-inů metod nebo konfigurací [25].

#### Library

Library je komponenta, která zajišťuje vytváření knihoven metod, plug-inů metod a jejich konfigurací. Zároveň je zodpovědná za ukládání a zálohu těchto knihoven, včetně správy výsledných XMI souborů.

### 3.3.3 Možnosti importu a exportu EPF Composeru

V současné době EPF Composer poskytuje několik možností importu a exportu dat. Jedním z nich je import a export do XML formátu. Ten slouží

především jako možnost, jak do EPF Composeru načíst modely třetích stran, které jsou do tohoto formátu převedené. Další možností je export dat do XML souborů specifických pro Microsoft Project a tedy převod namodelovaného procesu z EPF Composeru do Microsoft Projectu [25].

# 4 Procesní vzory

Tato kapitola se zabývá procesními vzory (*process patterns*) a anti-patterny v obecné rovině a dále uvádí několik příkladů. V předchozí kapitole byly popsány procesy vývoje software a způsoby jejich popisu. V rámci těchto procesů se mohou vyskytovat opakující se postupy či množiny souvisejících elementů – takzvané procesní vzory.

## 4.1 Patterny

Vzory (*patterny*) jsou prvky návrhu, které v příslušném kontextu poskytují řešení problému pomocí dané struktury [28]. Termín *pattern* je připisován stavebnímu architektovi Christopheru Alexanderovi. Ten jej definoval jako „opakovaně použitelnou formu řešení designového problému“ [29]. Termín procesní vzor (*process pattern*) poprvé použil James O. Coplien v roce 1994. Definoval jej jako „vzory činností v rámci organizace (a tedy i projektu)“ [30]. Organizovanou množinou procesních vzorů lze vytvořit proces, případně metodika [30].

Procesní vzory jsou utvářeny na základě minulých zkušeností. Je potřeba je však upravit a konkretizovat v kontextu současného problému [31]. Procesní vzory tedy popisují řešení, ale bez specifických detailů. Nepopisují tedy konkrétní postupy pro jednotlivé úkoly [30].

## 4.2 Anti-patterny

Anti-patterny lze definovat jako negativní patterny. Jsou to často se opakující špatné techniky. Phillip Laplante anti-patterny definuje jako řešení problémových situací s dobrým úmyslem, které ale ve výsledku mají více negativních důsledků než pozitivních [31].

Termín anti-pattern byl poprvé použit v roce 1995 Andrewem Koenigem v článku pro Journal of Object-Oriented Programming. Za jeho popularizací ale stojí knihy Williama Browna, které se začaly anti-patterny zabývat jako hlavním tématem [31]. Brown anti-patterny rozděluje do tří základních oblastí – architektonické, implementační a manažerské. Laplante oproti tomu uvádí následující čtyři typy:

- architektonické – popisují takové praktiky, které vedou ke špatným architektonickým rozhodnutím

- návrhové – popisují praktiky, které vedou ke špatnému návrhu aplikace
- manažerské – popisují praktiky manažerů, ať už jednotlivců nebo v obecné rovině, které vedou k neúspěchu projektu
- kulturní – příčinou neúspěchu není selhání jednotlivce nebo použití špatné praktiky, ale série špatných rozhodnutí, které způsobují toxické prostředí (kulturu)

Mezi příčinami anti-patternů může být nedostatek znalostí a zkušeností v řešení daného problému nebo nesprávné použití patternu [32]. Brown uvádí „sedm smrtelných hříchů“, které jsou hlavními příčinami chybování při vývoji softwaru [32]:

- spěch – Při plánování projektů jsou často vytvořeny cíle, které je ale potřeba splnit do příslušného termínu, většinou na úkor testování a celkové kvality dodaného produktu.
- apatie – Neochota se vůbec pokusit o nějaké řešení známých problémů.
- přízemnost – Spočívá v odmítnutí řešení, které je obecně považováno za efektivní.
- lenost – Zejména se týká přílišného využívání automatického generování kódu bez hlubší revize.
- chamtivost – Především záležitost architektury, která je málo abstraktní a navrhována do příliš velkých detailů, což vede k její neúměrné složitosti.
- ignorance – Jedná se o obdobu lenosti, pouze v intelektuální rovině.
- pýcha – Spočívá zejména v odmítnutí již funkčního řešení a pokus o implementaci vlastního, což s sebou přináší zbytečné finanční náklady a rizika.

Pro každý anti-pattern existuje řešení, kterým lze negativní důsledky omezit. Často ale nebývá aplikováno, protože o anti-patternech neexistuje dostatečné povědomí na to, aby mohly být vůbec rozpoznány [33].

V literatuře jsou anti-patterny většinou reprezentovány málo formálními slovními popisy. Brown anti-patterny uvádí ve více strukturované podobě, včetně ukázkových situací, důsledků i kroků k nápravě [31].

### 4.2.1 Příklady anti-patternů

Následují příklady některých anti-patternů a jejich stručné popisy.

#### Corncob

**Shrnutí:** Tento anti-pattern je manažerského typu. Jako corncob je v angličtině označována osoba, která nesdílí obecně uznávaný názor ostatních a tím působí problémy. V kontextu tohoto anti-patternu je to člověk, který má na projekt destruktivní vliv. Jedná se buď o člena týmu nebo významně postavenou osobu externě spolupracující organizace, který na projekt působí negativně v technické, politické nebo osobní rovině. Nesouhlasí s klíčovými rozhodnutími učiněnými v rámci projektu a snaží se je změnit.

**Důsledky:** Popsané chování má za následek časté změny učiněných rozhodnutí a neschopnost dosáhnout pokroku ve vývoji.

**Řešení:** Řešení tohoto anti-patternu jsou aplikovaná na různých úrovních, včetně strategické, operativní a taktické, všechny však spočívají v omezení podpory problémové osoby ze strany managementu. Na taktické úrovni lze například přenechat zodpovědnost za kritizované oblasti právě problémové osobě nebo, pokud používá nejasné či naučené fráze, ji na poradě přimět k objasnění jejich skutečného významu. Operativně se dá záležitost řešit pomocí „nápravného“ pohovoru, na němž je problémové osobě vysvětlena problémovost jejího chování, případně doporučení této osoby personalistům, kteří mu zajistí jiné zaměstnání. Strategickým řešením může být utvoření jednoho týmu z vícero takto problémových osob, kteří takto budou konfrontováni s negativními osobnostmi svých kolegů [32].

#### Fire Drill

**Shrnutí:** Na začátku projektu je příliš mnoho času věnováno vyjednávání technologických i politických záležitostí. Management tak vývojový tým vyzývá k čekání se začátkem implementace, případně mu dává nejasná a protichůdná zadání. To vede k časovému skluzu implementace, který je ale potřeba dohnat.

**Důsledky:** Časový tlak na dokončení implementace vede k nekvalitnímu kódu nebo nedostatečnému otestování a dokumentaci. Symptomy tohoto anti-patternu v projektu jsou tedy dlouhé období na začátku projektu, kdy je pozornost věnována zejména aktivitám souvisejícím se specifikací požadavků nebo návrhem, následované kratší a intenzivní implementační fází. Testování je buď velmi omezené, nebo žádné. Zároveň je možné pozorovat změnu v komunikaci mezi managementem a vývojáři, kdy na začátku spolu

nekomunikují prakticky vůbec, ale po zjištění časového skluzu je komunikace velmi živá.

**Řešení:** Řešením tohoto anti-patternu může být oddělení projektového prostředí na interní a externí. Interní prostředí se soustředí na průběžný pokrok projektu a konečnou dodávku, zatímco externí prostředí má za cíl udržovat vztahy s vnějším prostředím projektu, tedy vyšším managementem a zákazníky. Vnější prostředí tak vytváří „úkryt“ pro vnitřní prostředí, v rámci kterého jsou zaměstnanci odstíněni od technologicko-politických záležitostí a mohou se lépe soustředit na plnění takových aspektů projektu, které jsou na těchto záležitostech nezávislé. [32].

### Standing On The Shoulders Of Midgets

**Shrnutí:** Anti-pattern Standing On The Shoulders Of Midgets je návrhového a implementačního charakteru. Organizace požaduje vyřešit implementační problém znovupoužitím starší knihovny. Její kód je však velmi nekvalitní nebo daný problém řeší pouze částečně. Tento anti-pattern je podobný známějšímu Big Ball of Mud, který na tento problém nahlíží z obecnějšího hlediska.

**Důsledky:** Výsledkem popsaného rozhodnutí je nepřehledná a špatně udržovatelná aplikace [34]. To v důsledku může vést k velkému množství chyb v produktu a náročným opravám.

**Řešení:** Správným řešením je vlastní implementace, která daný problém odstraní, případně použití opravdu kvalitní knihovny. Tato rozhodnutí jsou obvykle v kompetenci softwarového architekta, jeho schopnosti jsou tedy klíčové.

# 5 SPADe

Software Process Anti-pattern Detector, zkráceně SPADe, je software vytvořený na Katedře informatiky a výpočetní techniky (KIV) Fakulty aplikovaných věd (FAV) Západočeské univerzity v Plzni (ZČU). Slouží pro dolování, ukládání a analýzu dat získaných z ALM nástrojů. Tato data budou následně využita pro zjišťování přítomnosti anti-patternů v projektech. Tato kapitola má za cíl uvést význam, funkci a datový model tohoto nástroje.

## 5.1 ALM nástroje

Application Lifecycle Management (ALM) je označení pro celý proces vzniku softwarového produktu, od prvotního nápadu jeho vzniku po vyřazení z provozu. Sestává ze tří základních aspektů, které se navzájem doplňují [35]:

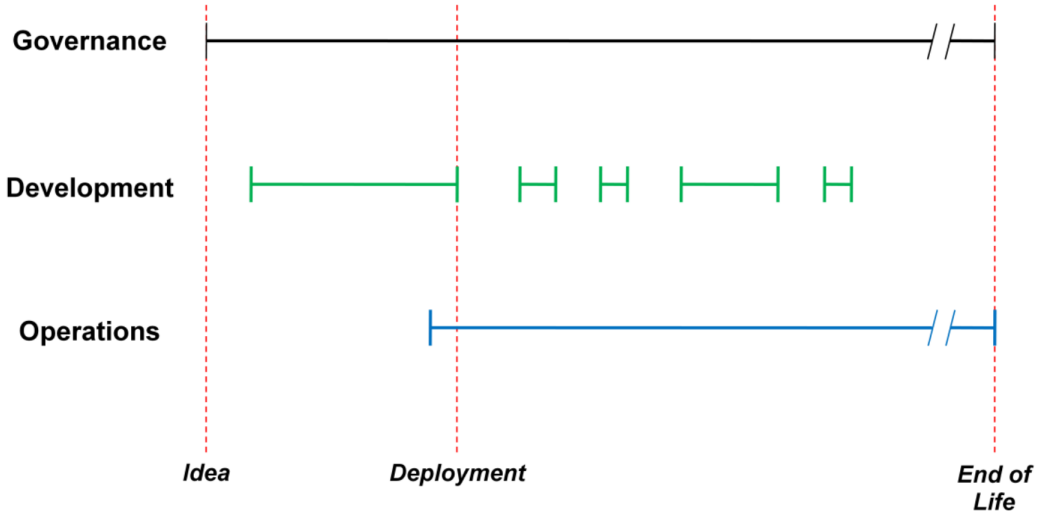
- řízení (governance) – Zahrnuje řízení projektu a rozhodovací procesy. Dělí se na několik fází, které se zabývají například vytvořením koncepce, kontrolní procesy směřující k nasazení softwaru nebo kontrola, zda produkt skutečně odpovídá požadavkům. Je tedy přítomno po celou dobu projektu.

- vývoj (development) – Samotná tvorba software zahrnující návrh, implementaci i testování. Probíhá od utvoření koncepce produktu do nasazení, poté v kratších fázích reaguje na změnové požadavky.

- provoz (operations) – Jedná se jak o samotné nasazení aplikace, tak následné monitorování běhu. Začíná ve chvíli, kdy je produkt dokončován a končí až s koncem celého projektu a vyřazením produktu z provozu.

Rozložení jednotlivých aspektů v průběhu projektu je přehledně znázorněno na obrázku 5.1.

ALM nástroje tedy slouží pro správu životního cyklu aplikace. Jedná se o software umožňující například projektové plánování a odhady, řízení požadavků a změn, správu kódu a verzí aplikace, automatizaci testování a nasazování, komunikaci v rámci týmu nebo evidenci odvedené práce. Kromě toho nabízejí i různé analytické funkcionality ve formě statistik a grafů. Tyto nástroje tak nabízí velmi komplexní pohled na proces vývoje software a podporují tím agilní přístup, trasovatelnost jednotlivých změn, celkovou přehlednost a lepší rozhodování v rámci procesu vývoje.




Obrázek 5.1: Aspekty ALM v průběhu projektu [35]

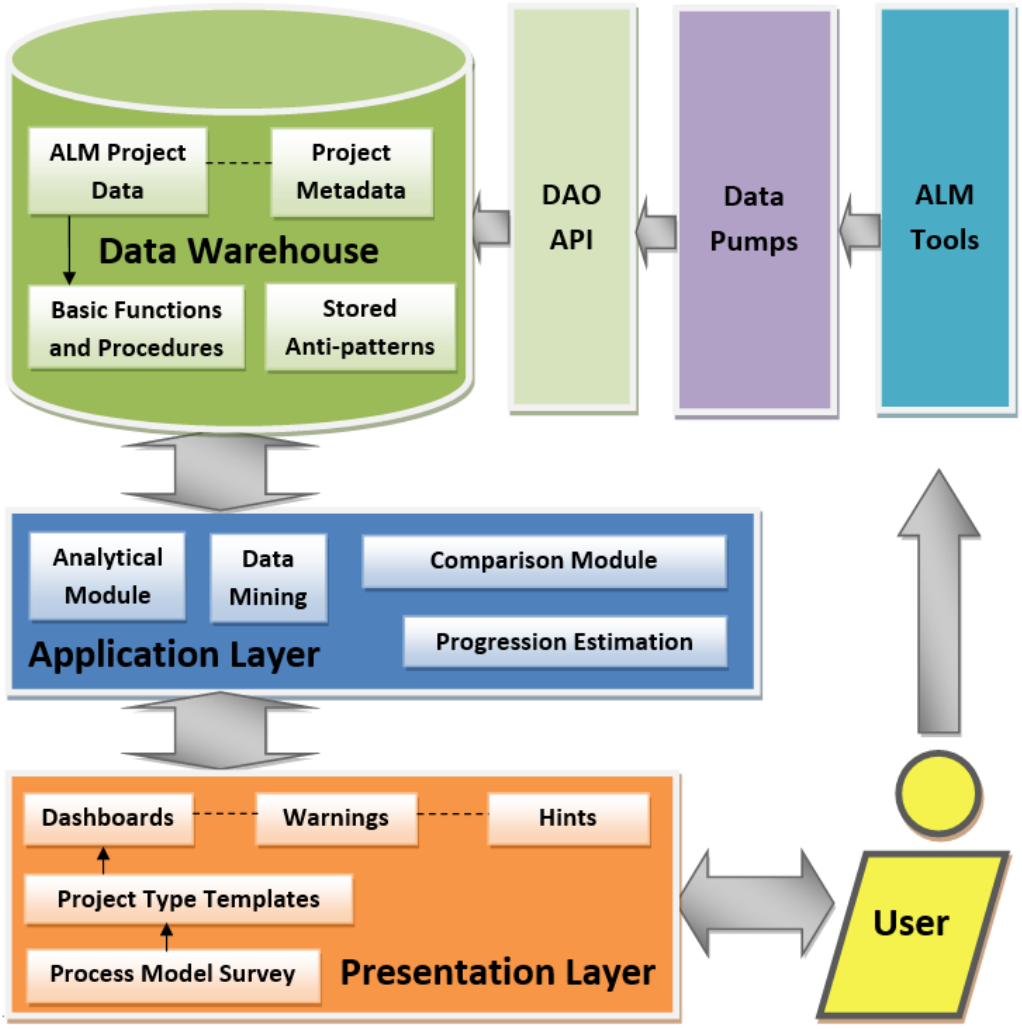
SPADe z ALM nástroje těží data, která jsou dále analyticky využitelná nad jejich rámec. V současné době využívá data z těchto ALM nástrojů [36]:

- Git
- GitHub
- Atlassian Jira
- Redmine
- Apache Subversion (SVN)
- Bugzilla
- Assembla
- IBM Rational Team Concert (RTC)

## 5.2 Architektura a funkce nástroje SPADe

Funkce nástroje SPADe je znázorněna na obrázku 5.2. SPADe doluje data pomocí datových pump z ALM nástrojů, konkrétně z jejich komponent systému verzování (*version control system* – VCS), správy úkolů (*issue-tracking system*) a Wiki stránek projektů.

Pro každý z dolovaných ALM nástrojů je v SPADe implementována vlastní datová pumpa. Tyto datové pumpy fungují na principu Extract-Transform-Load (ETL). Data jsou po extrakci z ALM nástroje převedena na jednotný datový formát, detailně popsaný v sekci 5.3, a následně uložena pomocí Data Access Object (DAO) rozhraní do datového skladu (Data Warehouse) [36]. Mimo těchto dat jsou do datového skladu ukládána i metadata, která obsahují například informace o doméně projektu nebo metodologii vývoje. Datový sklad je implementován pomocí MySQL databáze.




Obrázek 5.2: Architektura nástroje SPADe [36]

Takto uložená data jsou dále využívána aplikační vrstvou. Ta slouží k jejich další analýze. Například uložená data jednoho projektu SPADe porovnává s dalšími podobnými projekty, což může naznačit další vývoj zkoumaného

projektu. Hlavní funkcionalitou, kterou aplikační vrstva poskytuje, je však identifikace anti-patternů.

Prezentační vrstva pak výsledky analýzy předává zpět uživateli. Tato data jsou ve formě různých grafů, statistik, seznamu nalezených anti-patternů nebo předpovědi neúspěchu projektu. Pokud je uživatelem například projektový manažer zkoumaného projektu, může tak na základě těchto informací přizpůsobit plán projektu takovým způsobem, aby se vyhl případným problémům, které nemusely být na první pohled patrné a v budoucnu by projektu pravděpodobně hrozily.

## 5.3 Datový model nástroje SPADe

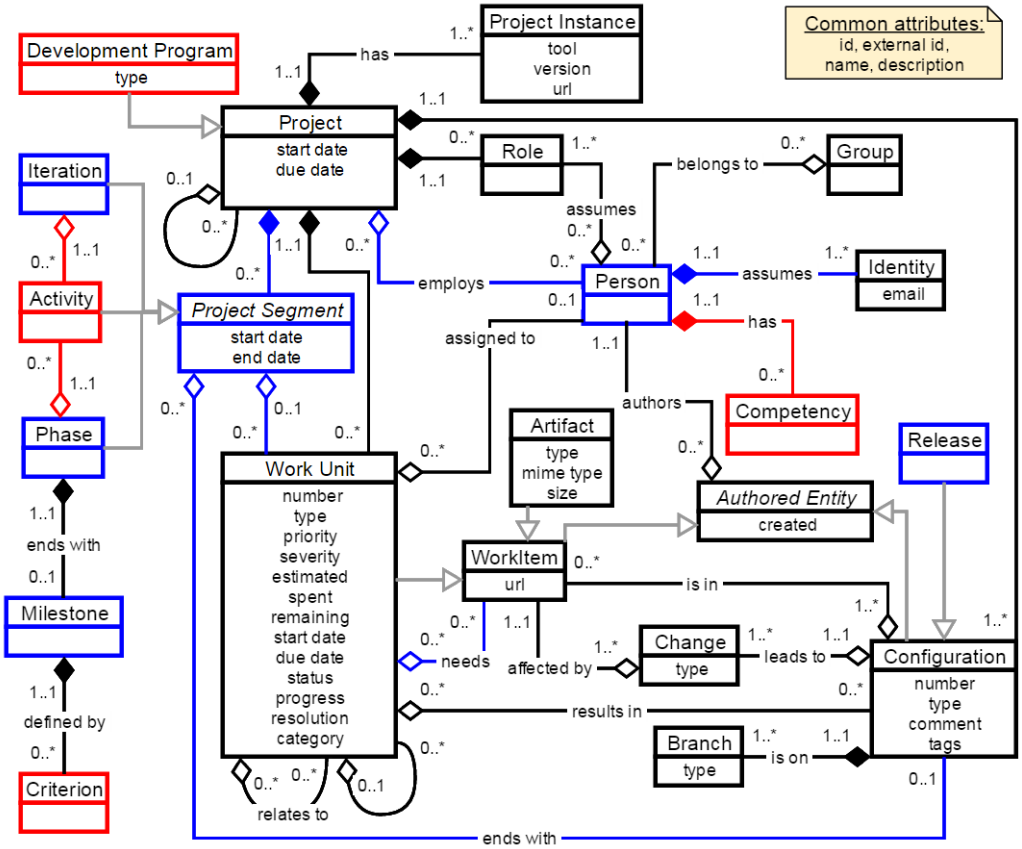
Jelikož je zadáním této práce vytvořit export dat z nástroje EPF Composer v takovém formátu, aby byl využitelný nástrojem SPADe, byl zvolen formát SQL skriptů, které lze jednoduše využít oproti databázi SPADe. Následuje tedy popis datového modelu s důrazem na koncepty využitelné při popisu anti-patternů.

Data ze všech ALM nástrojů jsou ukládána v jednotné podobě. Výhodou tohoto přístupu je, že je možné plošně využít stejný způsob vyhledávání anti-patternů pro všechny uložené projekty a zároveň je takto možné jednoduše porovnávat projekty původem z různých ALM nástrojů mezi sebou.

Datový meta-model, využívaný nástrojem SPADe, je znázorněný na obrázku 5.3. Jednotlivé barvy vyjadřují míru nutnosti specifikace jednotlivých entit uživatelem. Černě vyznačené entity lze přímo mapovat z ALM nástrojů. V případě modře vyznačených entit je nutná jistá míra spolupráce uživatele. Červeně vyznačené entity specifikuje sám uživatel.

Terminologie použitá k označení jednotlivých tříd je inspirována meta-modely SPEM 2.0, Software Engineering Workflow Language (SEWL) a metodologií RUP [37]. Týká se to především tříd jako jsou `Role`, `Artifact`, `Activity`, `Phase` nebo `Iteration`. Část názvů tříd je převzata i ze samotných ALM nástrojů. Těmi jsou například `Branch`, `Commit` nebo `Release`. Dále bylo potřeba specifikovat navíc i třídy jako `Configuration` (sada elementů změněná jednou atomickou akcí) nebo `Work Item` (rodičovská třída pro třídy `Artifact`, `Work Unit` a `Configuration`).

Na základě toho, že datový model SPADe obsahuje zmíněné základní koncepty, jako je role (`Role`), úkol (`Work Unit`) nebo artefakt (`Artifact`), lze usuzovat, že jím lze reprezentovat prakticky všechny procesy a metodiky, které tyto koncepty používají. Tyto koncepty a jejich základní atributy budou přítomny vždy. Rozdíly mohou být pouze v konkrétní specifikaci nebo




Obrázek 5.3: Doménový datový meta-model SPADe [36]

uspořádání těchto konceptů [36].

V kontextu této práce jsou důležité zejména třídy `Project`, `WorkItem`, `Person` a `Role`.

Třída `WorkItem` je totiž rodičovskou třídou pro třídy jako `Artifact`, `WorkUnit` nebo `Commit`. Pro získávání dat, která odpovídají těmto tabulkám, lze tedy využít rodičovskou `WorkItem`. Ta má atribut `WorkItemType`, který nabývá hodnot `ARTIFACT`, `COMMIT`, `COMMITED`, `CONFIGURATION` a `WORK_UNIT`, a podle kterého lze zjistit, o jaký typ položky se jedná. Bude tedy sloužit pro získávání informací o artefaktech, commitech a úkolech. Propojení třídy `WorkItem` s třídou `Projekt` je realizováno přes třídu `Person`, která má atribut `projectId` a kterou má `WorkItem` definovanou jako svého autora.

Jelikož UMA, která bude na tento model mapována, nepracuje s konceptem jednotlivých osob, ale pouze rolí, budou tyto role mapovány přes třídu `Person`.To znamená, že pokud má nějaký úkol definovanou roli jako vykonavatele, budou v datovém skladu SPADe hledány ty úkoly, které mají

jako vykonavatele osobu s danou rolí. V praxi má navíc datový sklad implementovány dvě úrovně detailu role, tedy detailnější `Role` a obecnější `RoleClassification`. `RoleClassification` pak slouží jako kategorie pro detailnější `Role`. Způsobu popisu anti-patternů spíše odpovídá obecnější `RoleClassification`.

# 6 Rozšiřující moduly Eclipse

Tato kapitola má za cíl popsat způsob fungování platformy Eclipse a rozšiřujících modulů Eclipse, které budou využity k rozšíření EPF Composeru pro možnost generování anti-patternů ve specifickém formátu.

Pojem rozšiřující modul obecně označuje jednotku modularity, kterou lze samostatně vyvíjet a vydávat, a která rozšiřuje funkcionalitu již existující aplikace.

Mnoho Eclipse aplikací, včetně Eclipse IDE nebo EPF Composeru, jsou tvořeny právě rozšiřujícími moduly a běhovým prostředím [38]. Jsou implementovány v jazyce Java a mají tedy podobu knihoven JAR. Jejich jádra jsou postavena na základním souboru rozšiřujících modulů, případně jeho části, nazvaném Eclipse Rich Client Platform (Eclipse RCP). Kromě toho obsahují další rozšiřující moduly, které zajišťují specifickou funkcionalitu jednotlivých aplikací [39].

Tyto rozšiřující moduly potřebují speciální prostředí, které obstarává jejich spouštění a běh. Eclipse takové prostředí zajišťuje pomocí frameworku Equinox, který je implementací standardu OSGi.

## 6.1 OSGi

Standard OSGi (Open Services Gateway initiative) specifikuje dynamický komponentový systém pro aplikace implementované v jazyce Java. Byl vytvořený v roce 2000 společností OSGi Alliance (dříve známou jako Open Services Gateway initiative).

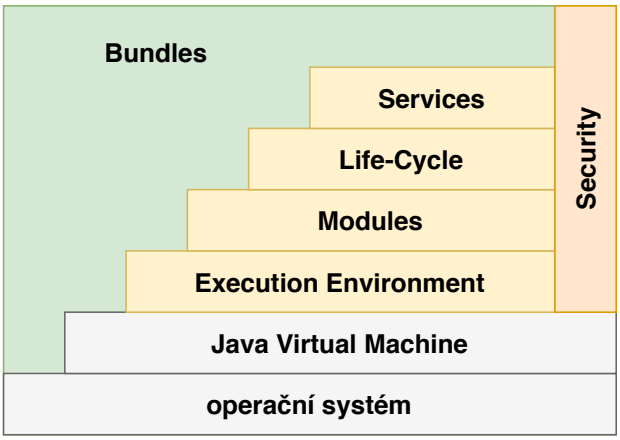
Jeho základem jsou balíky (*bundles*). Jedná se o samostatné komponenty, které lze za běhu celé aplikace dynamicky instalovat, spouštět, ukončovat nebo odinstalovávat bez nutnosti jejího restartu. Jsou specifické tím, že obsahují soubor `MANIFEST.MF`, který specifikuje vztahy vůči ostatním komponentám [40].

Každý framework, jenž implementuje OSGi, poskytuje funkcionality následujících oblastí [41]:

- **Bundles** – OSGi balíky
- **Services** – servisní vrstva, dynamicky propojuje balíky mezi sebou
- **Life-Cycle** – API pro správu životních cyklů balíků

- **Modules** – vrstva pro deklaraci závislostí a zapouzdření
- **Security** – vrstva zajišťující fungování balíků pouze v rámci definovaných možností
- **Execution Environment** – vrstva definující dostupnost jednotlivých tříd a metod pro danou platformu

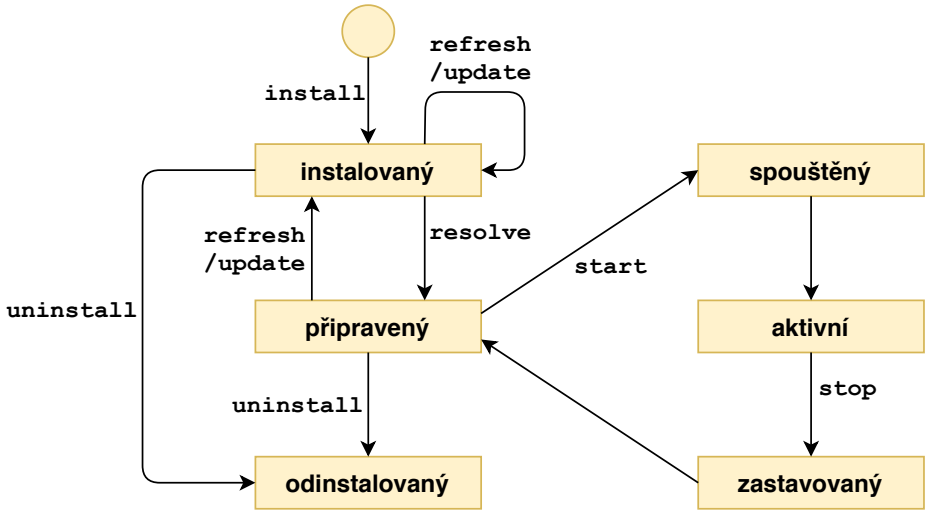
Tento model je znázorněn na obrázku 6.1.




Obrázek 6.1: Vrstvy OSGi [41]

Životní cyklus balíku je řízen vrstvou *Life-Cycle*. Stavy, ve kterých se může nacházet, a přechody mezi nimi jsou přehledně znázorněny na obrázku 6.2. Jejich popis je následující [42]:

- **instalovaný** – běhové prostředí ví o existenci balíku, balík se do tohoto stavu dostane po spuštění Eclipse aplikace, jsou všechny rozšiřující moduly nainstalovány
- **připravený** – balík a všechny jeho závislosti jsou připravené ke spuštění
- **spouštěný** – balík je spouštěný, například vykonáváním metody `BundleActivator.start()`
- **aktivní** – balík běží
- **zastavovaný** – balík je zastavovaný, například vykonáváním metody `BundleActivator.stop()`
- **odinstalovaný** – balík byl odstraněn z běhového prostředí OSGi




Obrázek 6.2: Životní cyklus OSGi balíku [42]

## 6.2 Eclipse Rich Client Platform

Eclipse Rich Client Platform je minimální sada rozšiřujících modulů, která je potřeba pro vývoj „bohatého klienta“ (*rich client*). Architekturou se bohatý klient v určitých aspektech podobá tlustému klientovi (*fat client*). Oba typy architektury poskytují funkcionalitu na straně klienta. V případě bohatého klienta je ale část business logiky sdílena s aplikačním serverem [43]. V tomto případě klient poskytuje komponentový model, uživatelské prostředí a funkcionality daného workspace a jako server je považována Java platforma nebo rozšiřující moduly a knihovny třetích stran.

Eclipse RCP slouží především jako modulární framework pro tvorbu uživatelského rozhraní. Modularita je zde zajištěna pomocí OSGi, popsaného v kapitole 6.1, konkrétně pomocí frameworku Equinox. Pro práci s uživatelským rozhraním obsahuje komponenty SWT a JFace.

Komponenta Standard Widget Toolkit (SWT) obsahuje nástroje pro tvorbu grafického uživatelského rozhraní různých okenních systémů. Poskytuje elementy jako například tlačítka, přepínače nebo vstupní pole.

Komponenta JFace obsahuje funkcionality pro práci s SWT. Poskytuje například fonty, dialogová okna, základy průvodců nebo ukazovátka postupu pro delší operace.

### 6.2.1 Definice rozšiřujícího modulu

Rozšiřující modul je definován buď souborem `plugin.xml` nebo speciální Java třídou. Mimo to je každý rozšiřující modul definován i souborem

`MANIFEST.MF`, který se nachází v adresáři `META-INF`. Tyto soubory určují vztah k ostatním rozšiřujícím modulům systému. Soubor `MANIFEST.MF` obsahuje deklaraci rozšiřujícího modulu, specifikující například název rozšiřujícího modulu (`Bundle-Name`), identifikátor (`Bundle-SymbolicName`), verzi (`Bundle-Version`) nebo specifikovaný aktivátor (`Bundle-Activator`). Identifikátor slouží jako unikátní název rozšiřujícího modulu a typicky odpovídá jmenné konvenci používané pro Java balíky (např. `org.<název společnosti>.<název rozšiřujícího modulu>`). Atributem manifestu `Require-Bundle` lze specifikovat závislosti rozšiřujícího modulu, tedy ty rozšiřující moduly, na které potřebuje vidět.

Textové řetězce, jako je například název rozšiřujícího modulu, mohou být z důvodu umožnění lokalizace přesunuty do zvláštního souboru `plugin.properties`. Z manifestu rozšiřujícího modulu je na ně poté pouze odkazováno. Oba soubory `plugin.xml` i `MANIFEST.MF` umožňuje Eclipse IDE upravovat ve speciálním editoru [39].

### 6.2.2 Aktivátor

Aktivátor je označení pro třídu, která je vstupním bodem rozšiřujícího modulu. Jedná se o třídu, která dědí od třídy `AbstractUIPlugin`. Implementuje tak metody `start`, která je zavolána po aktivaci rozšiřujícího modulu, a `stop`, která je zavolána, když má být rozšiřující modul ukončen. Typicky je aktivátor implementován s využitím návrhového vzoru jedináček. Ve chvíli, kdy je plug-in aktivován, je Eclipse aplikací vytvořena jediná instance této třídy, která je využívána po celou dobu života rozšiřujícího modulu.

### 6.2.3 Fragmenty

Fragment je nepovinná součást rozšiřujícího modulu, který se ve vztahu k němu nazývá hostující rozšiřující modul. Fragment lze nezávisle na hostujícím rozšiřujícím modulu instalovat nebo odinstalovávat. To je výhodné například pro části rozšiřujících modulů, které obsahují kód specifický pro konkrétní operační systém, lokalizace nebo testování. Ve chvíli, kdy je fragment načten, není sice fyzicky součástí hostujícího rozšiřujícího modulu, ale logicky se tak tváří.

Fragmenty jsou vyvíjeny jako samostatné projekty a oproti rozšiřujícím modulům se liší v několika specifikách. Namísto souboru `plugin.xml` je specifikován souborem `fragment.xml` a kořenovým elementem tohoto XML souboru je `fragment` s atributy `plugin-id` a `plugin-version`.

## 6.3 Vývoj rozšiřujícího modulu

Pro vývoj rozšiřujících modulů Eclipse je příhodné využít vývojové prostředí Eclipse IDE s rozšířením Plug-in Development Environment (PDE) [39] [44]. PDE poskytuje speciální nástroje pro vývoj, testování a sestavení rozšiřujících modulů Eclipse.

### 6.3.1 Cílová platforma

Cílová platforma (*target platform*) definuje sadu rozšiřujících modulů pro které je nový rozšiřující modul vyvíjen a se kterými bude používán. To je výhodné především z toho důvodu, že tím pádem není nutné mít všechny projekty otevřené ve vývojovém prostředí. Cílovou platformu lze definovat jako XML soubor s příponou `.platform`. V případě, že je nový rozšiřující modul vyvíjen pro EPF Composer, cílová platforma tohoto rozšiřujícího modulu obsahuje všechny rozšiřující moduly této aplikace.

Alternativou k takovémuto specifikování cílové platformy je vývoj rozšiřujícího modulu Eclipse ve workspace obsahujícím projekty všech rozšiřujících modulů, oproti kterým je rozšiřující modul vyvíjen.

### 6.3.2 Sestavení rozšiřujícího modulu

Rozšiřující modul je možné sestavit různými způsoby – manuálně pomocí průvodce v Eclipse IDE, s využitím nástroje Apache Ant nebo pomocí shellových skriptů. Výstupem sestavení je JAR soubor.

### 6.3.3 Instalace a spouštění rozšiřujícího modulu

Každá instalace Eclipse aplikace ve svém kořenovém adresáři obsahuje adresář `configuration` s konfiguračním souborem `config.ini`, adresář `plugins` a spustitelný `EXE` soubor.

Pro instalaci rozšiřujícího modulu do Eclipse aplikace je potřeba, aby tato aplikace neběžela. Poté je nutné sestavené JAR soubory umístit do adresáře `plugins` v kořenovém adresáři instalace příslušné Eclipse aplikace. Po jejím spuštění je rozšiřující modul automaticky nainstalován.

Instalace probíhá tak, že při spouštění Eclipse aplikace prohledává adresář s rozšiřujícími moduly a na základě souborů `MANIFEST.MF` a `plugin.xml` si vytváří vniřní model každého rozšiřujícího modulu. Samotný kód rozšiřujícího modulu je načten až ve chvíli, kdy je potřeba. Výhodou toho je rychlejší spuštění Eclipse aplikace a fakt, že rozšiřující moduly nezabírají mnoho místa v paměti. Zároveň je ale v případě potřeby rozšiřující modul nalezen a spuštěn

velmi rychle. Tento způsob práce se zdrojovými kódy nebo daty se obecně nazývá *lazy-loading*.

### 6.3.4 Testování rozšiřujících modulů Eclipse

Při testování lze volit ze tří možných architektur, které jsou popsány níže. V každém z těchto případů jsou testy obvykle umístěny v adresáři `src/test/java`.

#### Knihovna uvnitř rozšiřujícího modulu

První možností je umístit testy i testovací knihovnu přímo jako součást testovaného rozšiřujícího modulu. Tento přístup je vhodný zejména pro testování malého rozsahu. Výhodným může být i z důvodu, že všechen kód, včetně toho testového, je načítán pomocí stejného classloaderu. Tím pádem vzniká možnost testovat i ty metody, které nejsou deklarované jako veřejné. Problém tohoto přístupu spočívá v komplikovanějším procesu sestavování aplikace. Mimo to bude muset mít rozšiřující modul definovány i závislosti na testovací frameworky nebo knihovny (JUnit, EasyMock, ...). To lze však obejít definováním testovací knihovny nebo frameworku jako nepovinné závislosti.

#### Samostatný rozšiřující modul

Často využívanou možností je testový kód umístit do samostatného rozšiřujícího modulu. Výhodou tohoto přístupu je oddělenost testového a testovaného kódu včetně jejich závislostí, čímž se zjednodušuje proces sestavení. Nevýhodou však je, že testovací rozšiřující modul má přístup pouze k API, které je testovaným rozšiřujícím modulem exportováno jako veřejné. Tuto nevýhodu lze obejít definováním výjimky pro testovací rozšiřující modul v manifestu testovaného rozšiřujícího modulu. Další nevýhodou, jenž není možné tímto postupem odstranit, je fakt, že kód testovacího rozšiřujícího modulu bude načten jiným classloaderem a tyto testy tak nemají přístup k neveřejným metodám testovaného rozšiřujícího modulu.

#### Fragment

Obdobným řešením je vytvořit pro testování fragment. Výhodou tohoto přístupu je, že kód testovacího rozšiřujícího modulu je načten stejným classloaderem jako kód hostujícího testovaného rozšiřujícího modulu a má tak přístup i k jeho neveřejným metodám. Kód z fragmentů však není viditelný

z jiných než hostujících rozšiřujících modulů a z toho vyplývá nevýhoda nemožnosti vytváření *test-suitů* z vícero různých testovacích fragmentů.

# 7 Popis anti-patternů pomocí EPF

Tato kapitola popisuje navržený způsob popisu anti-patternů v EPF Composeru včetně jeho východisek a příkladů. Na základě navrženého popisu byly dále implementovány rozšiřující moduly EPF Composeru pro export takto popsaných anti-patternů.

V kapitolách 2 a 4 byly popsány softwarové procesy a anti-patterny pomocí procesní terminologie. Nástroj SPADe nicméně skladuje data konkrétních projektů, nikoli popisy procesů. Cíl této kapitoly je tedy navržení popisu procesu a z něj odvozeného SQL dotazu, který má najít výskyt tohoto procesu v datech projektu.

## 7.1 Způsob popisu anti-patternů

Pro popis anti-patternů bylo potřeba zvolit takový způsob, který by umožňoval jejich detekci v natěžených datech nástrojem SPADe a zároveň by respektoval meta-model UMA používaný nástrojem EPF Composer. Jelikož UMA a EPF Composer slouží především k popisu procesů a datový model SPADe vychází ze SPEM 2.0 používaného také k modelování procesů, bylo rozhodnuto o maximálním využití jejich možností a následném modelování anti-patternů jako podmnožiny procesů.

Jak již bylo popsáno v kapitole 4.2, existuje velké množství různých typů anti-patternů, většinou vyjádřených v textové podobě. Vzhledem k povaze této práce ve smyslu plánovaného převodu popisů anti-patternů na SQL dotazy byly pro popis v EPF Composeru vybrány pouze takové, které lze snadno vyjádřit formálním způsobem, tedy především těch, které lze popsat pomocí základních prvků procesu úkol, role a výsledek práce.

### 7.1.1 Podobnosti s popisem procesů

Popisy anti-patternů pomocí meta-modelů mají s popisy procesů některé společné vlastnosti. Jsou jimi například definice úkolů, rolí a výstupů práce. Tyto elementy také sdílejí příslušnost k určitým disciplínám a fázím Rational Unified Processu.

Stejně jako v případě procesů, jsou i anti-patterny definovány jako jednotlivé plug-iny metod. Obdobně je tomu v případě zmiňovaných definic

úkolů, rolí nebo výstupů práce, které jsou specifikovány v rámci jejich obsahů metod.

### 7.1.2 Základní prvky procesního modelu anti-patternů

Přítomnost některých anti-patternů v projektových datech může být detekována kontrolou přítomnosti jeho základních elementů, jako jsou například artefakty nebo úkoly. To se týká například anti-patternu Specify Nothing.[1] I v takových případech je ale potřeba meta-model mírně doplnit, a to minimálně o klíčové výrazy, podle kterých budou elementy v datech nacházeny. Konkrétně v případě anti-patternu Specify Nothing je potřeba detekovat nepřítomnost artefaktu specifikace požadavků. Aby mohl být tento artefakt v datech nalezen, je nutné zadefinovat klíčová slova, která budou porovnávána s názvy všech artefaktů v projektových datech. Způsob definování těchto klíčových slov je podrobněji popsán v podkapitole 7.1.3.

Mezi popisy anti-patternů a procesů jsou však určité rozdíly, popsané například v podkapitole 7.1.4. Jelikož jsou meta-modely jako je například UMA určené k popisu procesů, bylo i z tohoto důvodu potřeba tyto meta-modely doplnit o nové parametry. Ty jsou definovány v poli *Main Description* příslušných elementů, které dědí od třídy `DescribableElement`.

Dalším významným rozdílem oproti popisu procesů je fakt, že některé anti-patterny jsou definovány pomocí negativních vlastností, tedy neexistencí některých elementů v obsazích metod. Příkladem může být anti-pattern Business as Usual[2]. Meta-model UMA ale podporuje pouze modelování existence elementů. Z tohoto důvodu jsou takové anti-patterny modelovány v „invertované“ podobě. To znamená, že je namodelována správná podoba, tedy „pattern“. Do pole *Brief Description* příslušného plug-inu metody (*Method Plug-in*) je pak přidán atribut `pattern`, který je nastaven na hodnotu `true`. Pro přehlednost je zároveň vhodné dodržovat i jmennou konvenci pro daný plug-in metody a jeho název (*Name*), případně zobrazovaný název (*Presentation Name*), zakončit pojmem „Pattern“. Příklad specifikace takového plug-inu metody je na obrázku 7.2.

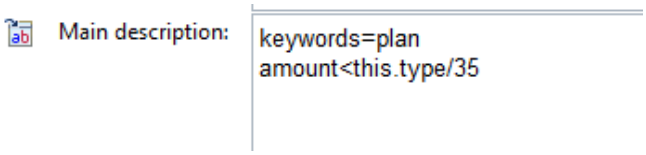
K vyjádření existence jednotlivých elementů anti-patternů popsaných pomocí EPF lze přistupovat dvěma základními způsoby.

První způsob uvažuje samotnou existenci úkolu nebo výsledku práce v obsahu metody jako požadavek na jejich existenci i v projektových datech.

[1] `https://github.com/ReliSA/Software-process-antipatterns-catalogue/blob/master/catalogue/Specify_Nothing.md`

[2] `https://github.com/ReliSA/Software-process-antipatterns-catalogue/blob/master/catalogue/Business_As_Usual.md`

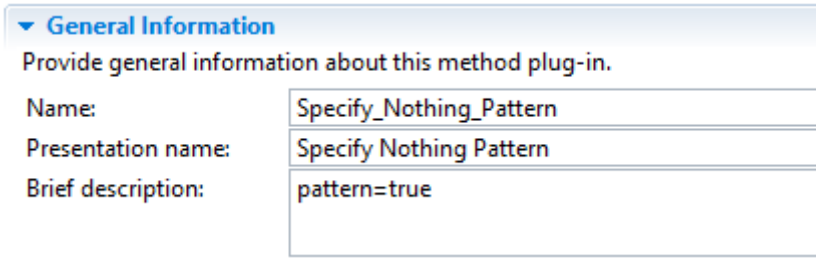
Pokud je pro jejich popis potřeba použít i vzorce definující jejich poměr v datech, popsané v podkapitole 7.1.4. Tyto vzorce jsou uvedeny v rámci *Main Description* těchto elementů, jak je znázorněno na obrázku 7.1. Výhodou tohoto přístupu je, že maximálně využívá možností EPF Composeru bez přílišné nutnosti dalších zásahů, a i z tohoto důvodu je uživatelsky přívětivý.




Obrázek 7.1: Ukázka dodatečné charakteristiky úkolu

Druhou možností je existenci úkolu nebo výstupu práce v obsahu metody tímto způsobem neinterpretovat. Požadavek na existenci úkolů a výsledků práce by v takovém případě byla definována v rámci *General Description* příslušného *Delivery Processu*. Tento způsob přináší možnost definovat vzorce, které odkazují na více než jeden element obsahu metody. Nevýhodou je, že by při použití tohoto způsobu bylo potřeba v rámci popisu procesu definovat existenci každého elementu, který se má v projektových datech nacházet, pomocí speciálního zápisu. To je však velmi nepraktické a navíc by to ovlivnilo negativně velikost celého projektu.

Nakonec byla zvolena první možnost, především z důvodu, že je bližší přirozenému použití EPF Composeru.




Obrázek 7.2: Ukázka specifikace „invertovaného“ plug-inu metody

### 7.1.3 Definování klíčových slov

Aby bylo možné specifikovaný element v datech projektu najít, je v popisu *Main Description* uveden atribut `keywords`, který uvádí klíčová slova, jako například „plan“. Podle nich budou poté v projektových datech identifikovány příslušné záznamy. Záznamy lze v projektových datech identifikovat

buď porovnáním klíčových slov s názvem, popisem, nebo jejich kombinací. Vzhledem k tomu, že porovnávání s popisem přinášelo příliš mnoho falešně pozitivních výsledků. Například tabulka 7.1 ukazuje podíl falešně pozitivních výsledků při dotazech na dokument specifikace. Z tohoto důvodu bylo rozhodnuto o porovnávání pouze s názvem. To se ukázalo jako plně dostačující, jelikož tyto falešně pozitivní výsledky byly odfiltrovány.

Tabulka 7.1: Podíl falešně pozitivních při porovnávání i s popisem.

| **ID projektu** | **1** | **2** | **3** | **4** | **5** |
|---|---|---|---|---|---|
| **pozitivních** | 6 | 5 | 6 | 13 | 8 |
| **z toho falešně** | 6 | 5 | 4 | 11 | 6 |

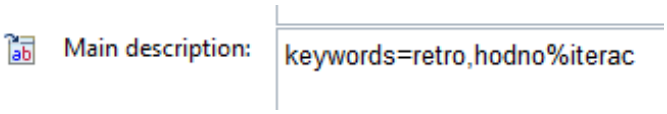
Vzhledem k tomu, že jsou různé analyzované projekty často vedeny v různých jazykových lokalizacích, především tedy v češtině a angličtině, je potřeba z důvodu univerzálnosti tato klíčová slova specifikovat ve více jazycích, tedy uvést více klíčových slov. Obdobný problém nutnosti specifikace vícero klíčových slov nastává v případě elementů, které jsou v datovém skladu reprezentovány různými názvy. V takovém případě jsou tato klíčová slova oddělena čárkou. Pokud je klíčový termín víceslovný, je nutné jednotlivá slova oddělit znakem `%`. Příklad takovýchto specifikací je znázorněn na obrázku 7.3.

### 7.1.4 Definování poměrů četností elementů

U některých anti-patternů, jako jsou například Death by Planning nebo Road to Nowhere, je potřeba specifikovat navíc poměr jednotlivých elementů v projektových datech. Toho bylo dosaženo přidáním atributu `amount`, který uvozuje vzorec tento poměr vyjadřující. Obecně má tento vzorec následující podobu: `amount<znaménko porovnání>this.type/<dělitel definující poměr>`. Člen `this.type` odkazuje na typ daného elementu, tedy zda se jedná například o artefakt nebo činnost. Příklad použití takového vzorce je na obrázku 7.1. Problémem může být fakt, že konkrétní poměr většinou není v popisech anti-patternů přímo definovaný. To je očekávatelné, protože každý projekt má specifické vlastnosti. Pro vyhledávání v datovém skladu je ale nutné tento poměr určit přesně. Při jeho definici je tak nutné brát ohled na kontext zkoumaných projektů.

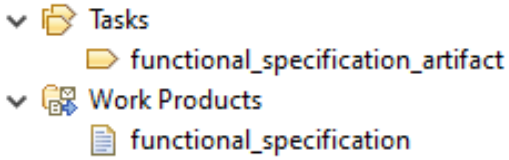
### 7.1.5 Popis výsledků práce

Výsledky práce jsou vždy reprezentovány jako *Work Product*. Každý výsledek práce má zároveň specifikovaný i vlastní *task*. Ten má příslušný výsledek



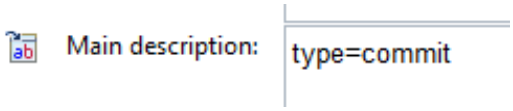

Obrázek 7.3: Ukázka specifikace klíčových slov

práce uveden jako svůj výstup. Jedním z důvodů tohoto přístupu je fakt, že u některých výsledků práce je potřeba specifikovat i roli, která jej modifikuje, což UMA dovoluje pouze touto cestou. Dalším důvodem je možnost naznačení, kdy je tento výsledek práce vytvářen v rámci *Delivery Process*, což je možné přes deskriptor příslušného tasku. Jmenná konvence pro task je `<název_artefaktu>_<artifact/outcome>`. Tento způsob reprezentace je znázorněn na obrázku 7.4.




Obrázek 7.4: Ukázka reprezentace artefaktu

Formální artefakty a dokumenty jsou reprezentovány pomocí elementu *Artifact*. Někdy je však nutné namodelovat i jiné výsledky práce, například commity. Toho lze docílit použitím elementu *Outcome* a specifikací klíčového slova `type` tak, jak je to uvedeno v ukázce 7.5.




Obrázek 7.5: Ukázka definice commitu

Mimo to lze pro artefakty specifikovat vzorce, které udávají podíl vůči dalším výsledkům práce v projektu.

Z důvodu přehlednosti je vhodné, aby byly atributy specifikovány pouze v příslušném elementu *task*. V opačném případě by mohlo kvůli nepozornosti dojít k duplicitní deklaraci některých atributů.

### 7.1.6 Popis činností

Činnosti jsou reprezentovány jako *tasky*. Protože jsou takto reprezentovány i některé artefakty, mají činnosti jmennou konvenci `<název_činnosti>_task`. Pole *Main Description* je určeno ke specifikaci atributů, zejména atributu `keywords`, ale i dalších, obdobně jako v případě artefaktů.

### 7.1.7 Popis procesů

Procesní pohled na anti-pattern je reprezentován jako *Delivery Process*. Struktura procesu obsahuje jednotlivé *tasky*. Ty by měly být vždy potomkem fáze, které přináleží. Zároveň každý *task* má odkaz (*Method Element Link*) na *task* z vytvořeného obsahu metody.

### 7.1.8 Popis fází a disciplín

Ačkoliv by bylo specifikování fází a disciplín vhodné, datový sklad na ně zatím nemá v dostatečné míře mapované činnosti. Vzhledem k tomu, že mapování elementů pouze podle jejich názvů je dostatečné, nepředstavuje toto omezení ani příliš velkou komplikaci.

Až bude na dotazování na fáze a disciplíny datový sklad připravený, lze pro jejich modelování v EPF Composeru použít následující postup. *Tasky* mohou být dále sdružovány do disciplín (*Discipline*) v rámci standardních kategorií (*Standard Categories*) obsahu metody. Pro určení těchto disciplín byly vybrány disciplíny z Rational Unified Processu (RUP), protože jsou všeobecně známé a je jich rozumný počet. Mohou jimi tedy být Bussiness Modelling, Requirements, Analysis and Design, Implementation, Test a Deployment. Tyto disciplíny by následně pomohly tyto *tasky* namapovat na reálná data.

Obdobný význam má sdružování *tasků* do fází. Tyto fáze (*Phase*) jsou fázemi Rational Unified Processu (RUP), tedy Inception, Elaboration, Constuction a Transition. Bylo by možné je definovat v rámci procesu jako druh prvku rozkladu práce (*Breakdown Element*).

## 7.2 Výsledné SQL skripty

Pro každý zvolený namodelovaný anti-pattern neboli plug-in metody EPF Composeru je vygenerovaný jeden SQL skript, který je určený ke zjištění přítomnosti anti-patternů v datovém skladu SPADe. Tento skript obsahuje několik SQL dotazů, z nichž každý vrací hodnotu buď 0 (symptom není obsažen), nebo 1 (symptom je obsažen). Toto rozdělení na několik jednotlivých skriptů namísto jednoho velkého je výhodnější z hlediska přehlednosti. Navíc lze ocekávat, že uživatel bude tyto skripty dále upravovat na míru konkrétnímu případu. Posouzení výsledků jednotlivých částí skriptů a konečné zhodnocení, zda testovaný projekt vykazuje či nevykazuje známky anti-patternu, závisí na uživateli.

Každý takovýto dotaz obsahuje samozřejmě i odkaz na ID zkoumaného projektu, které je univerzálně specifikované konstantou PROJECT_ID. Uži-

vatel poté provede záměnu za ID projektu, který zrovna potřebuje otestovat.

Příklad jednoduchého SQL dotazu je uveden v ukázce 7.1. Konkrétně se jedná o dotaz pro zjištění přítomnosti artefaktu, který je specifikovaný klíčovým slovem. Pokud se tento artefakt v datech nachází alespoň jednou, je navrácena hodnota 0, v opačném případě je navrácena hodnota 1.

```
SELECT (CASE WHEN COUNT(*) > 0 THEN 0 ELSE 1 END) FROM work_item wi
    JOIN person p ON p.id = wi.authorId
    AND p.projectId = PROJECT_ID
    AND wi.workItemType = 'ARTIFACT'
    AND (wi.name LIKE '...')
```

Ukázka kódu 7.1: Příklad vygenerovaného SQL dotazu

Ukázka 7.2 obsahuje strukturu dotazu pro zjištění poměru nějakého konkrétního typu úkolu nebo artefaktu vůči všem. Tento dotaz obsahuje dva vnitřní poddotazy označované jako `a` a `b`. Poddotaz `a` specifikuje konkrétní element, jehož podíl je zjišťován. Poddotaz `b` pak specifikuje obecný element, vůči kterému je uplatňován daný podíl. Na základě počtu elementů, které jsou oběma poddotazy zjištěny a daného dělitele je určena výsledná hodnota. Ta opět nabývá hodnot 0, nebo 1.

```
SELECT (CASE WHEN COUNT(*) > 0 THEN 0 ELSE 1 END) FROM
    (SELECT COUNT(*) as sum ...
    AS a,
    (SELECT COUNT(*) as sum ...
    AS b
WHERE a.sum < (b.sum/...)
```

Ukázka kódu 7.2: Struktura SQL dotazu se specifikací podílu

„Invertované“ anti-patterny generují velmi obdobné SQL dotazy, pouze se mění určení jejich výsledné hodnoty. Ta je oproti běžným dotazům převrácená. Takovéhoto určení návratové hodnoty je naznačeno v ukázce 7.3.

```
SELECT (CASE WHEN COUNT(*) > 0 THEN 1 ELSE 0 END) FROM ...
```

Ukázka kódu 7.3: Struktura SQL dotazu pro „invertovaný“ anti-pattern

## 7.3 Příklady anti-patternů

Následují příklady anti-patternů, které jsou a nejsou vhodné pro popis pomocí EPF Composeru.

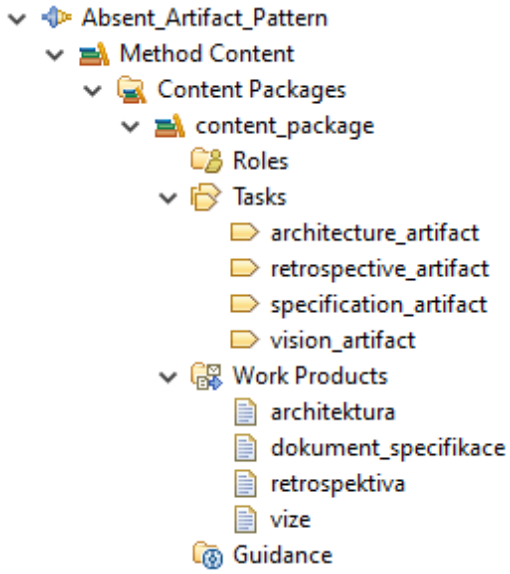
Níže uvedené anti-patterny byly modelovány v nástroji EPF Composer. Do této kapitoly byly vybrány takovým způsobem, aby ilustrovaly základní aspekty výše popsaného popisu na konkrétních příkladech. Takto vytvořené

modely budou následně využity pro validaci vytvářených rozšiřujících modulů Eclipse, jak je popsáno v kapitole 9.

### 7.3.1 Absent Artifact

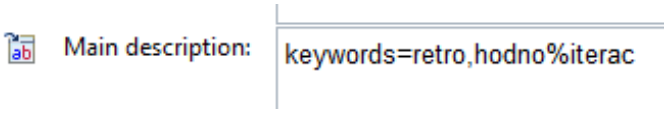
Absent Artifact specifikuje neexistenci důležitých artefaktů v projektových datech. Vychází z předpokladu, že pokud tyto artefakty chybí, je pravděpodobné, že implementace nebude příliš kvalitní [45]. Symptom je tedy následující:

- chybějící artefakty Vize, Architektura, Dokument specifikace a Retrospektiva




Obrázek 7.6: Obsah metody anti-patternu Absent Artifact v EPF Composeru

Obrázek 7.6 ukazuje obsah metody modelu tohoto anti-patternu v nástroji EPF Composer specifikující všechny artefakty a k nim náležící úkoly. Každý artefakt má samozřejmě specifikovaná klíčová slova, podle kterých bude nalezen v datech projektu. Na obrázku 7.7 je ukázka specifikace klíčových slov pro dokument retrospektivy.




Obrázek 7.7: Specifikace klíčových slov pro dokument retrospektivy

V ukázce kódu 7.4 je část generovaného SQL skriptu s dotazem na artefakt retrospektivy. Vzhledem k tomu, že se jedná o invertovaně definovaný anti-pattern, je jeho návratová hodnota taktéž invertovaná.

```
SELECT (CASE WHEN COUNT(*) > 0 THEN 0 ELSE 1 END) FROM work_item wi
    JOIN person p ON p.id = wi.authorId
    AND p.projectId = PROJECT_ID
    AND wi.workItemType = 'ARTIFACT'
    AND (wi.name LIKE '%retro%' OR wi.name LIKE '%hodno%iterac%')
```

Ukázka kódu 7.4: Část generovaného SQL skriptu pro anti-pattern Absent Artifact

### 7.3.2 Architects Don't Code

Anti-pattern Architects Don't Code [3] postihuje situaci, kdy si organizace uvědomuje, že pro efektivní vývoj softwaru je vhodné, aby návrh softwaru vytvářel systémový architekt. Jeho čas je však drahý a proto jej věnuje pouze abstraktnímu návrhu softwaru. Implementace jej nezajímá. [46] Symptomy anti-patternu Architects Don't Code jsou tedy následující:

- architekt modifikuje artefakty týkající se návrhu
- architekt interaguje s úkoly návrhu
- architekt nevytváří ani nemodifikuje implementaci

Obrázek 7.8 ukazuje obsah metody v nástroji EPF Composer. Role architekta je zde nazvaná jako designer, aby odpovídala názvosloví používanému SPADe. To je ukázáno na obrázku 7.9. Anti-pattern je v tomto případě modelován „invertovaným“ způsobem, aby bylo možné zachytit negativní vlastnost projektového manažera, a to neinterakci s implementací. Implementace je reprezentována commity, tedy výsledkem práce *Outcome* se specifikovaným typem tak, jak je to naznačeno na obrázku 7.5. Dále je součástí obsahu metody plánovací artefakt i task postihující vlastnost, že architekt se i v „invertovaném“ případě nadále věnuje návrhu.

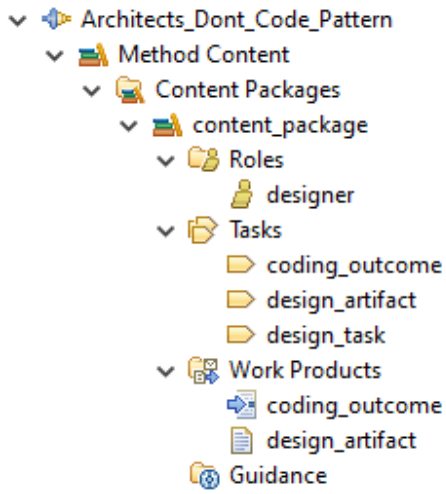
V ukázce kódu 7.5 je část generovaného SQL skriptu se specifikací interakce architekta s implementací. Opět se jedná o „invertovaný“ anti-pattern, což má vliv na návratovou hodnotu.

```
SELECT (CASE WHEN COUNT(*) > 0 THEN 0 ELSE 1 END) FROM work_item wi
    JOIN person p ON p.id = wi.authorId
```

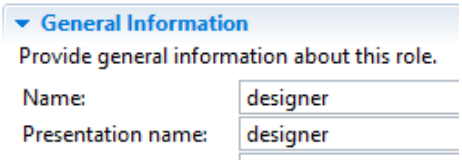
[3] https://github.com/ReliSA/Software-process-antipatterns-catalogue/blob/master/catalogue/Architects_Dont_Code.md

```
    JOIN person_role pr ON p.id = pr.personId
    JOIN role r ON r.id = pr.roleId
    JOIN role_classification rc ON r.classId = rc.id
    AND p.projectId = PROJECT_ID
    AND (rc.class = 'DESIGNER')
    AND wi.workItemType = 'COMMIT'
```

Ukázka kódu 7.5: Část generovaného SQL skriptu pro anti-pattern Architects Don't Code




Obrázek 7.8: Obsah metody anti-patternu Architects Don't Code v EPF Composeru




Obrázek 7.9: Specifikace role architekta anti-patternu Architects Don't Code v EPF Composeru

### 7.3.3 PMs Who Write Specs

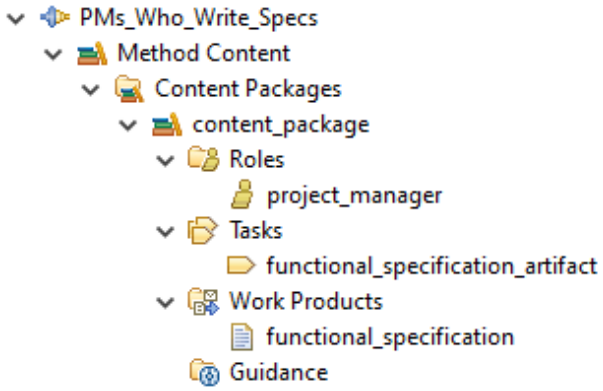
Podle anti-patternu PMs Who Write Specs[4] není vhodné, aby funkční specifikaci psal projektový manažer. Důvodem je, že by v takovém případě

[4] https://github.com/ReliSA/Software-process-antipatterns-catalogue/blob/master/catalogue/PMs_Who_Write_Specs.md

mohlo docházet ke střetu zájmů. Projektový manažer by měl projekt plánovat, nikoliv specifikovat, co má být v rámci něj splněno. Ideálně by funkční specifikace měla být dílem obchodních analytiků [47]. Symptom tohoto anti-patternu je tedy následující:

- funkční specifikace, na níž se podílí projektový manažer

Na obrázku 7.10 je obsah metody v nástroji EPF Composer. Specifikuje tři základní elementy nutné pro popis tohoto anti-patternu, tedy výsledek práce s funkční specifikací, k němu se vztahující úkol a role projektového manažera, která je jeho vykonavatelem. Ukázka kódu 7.6 pak obsahuje SQL skript generovaný na základě tohoto popisu.




Obrázek 7.10: Obsah metody anti-patternu PMs Who Write Specs v EPF Composeru

```
(CASE WHEN COUNT(*) > 0 THEN 1 ELSE 0 END) FROM work_item wi
   JOIN person p ON p.id = wi.authorId
   JOIN person_role pr ON p.id = pr.personId
   JOIN role r ON r.id = pr.roleId
   JOIN role_classification rc ON r.classId = rc.id
   AND p.projectId = PROJECT_ID
   AND (rc.class = 'PROJECTMANAGER')
   AND wi.workItemType = 'WORK_UNIT'
   AND (wi.name LIKE '%functional%' OR wi.name LIKE '%specification%')
```

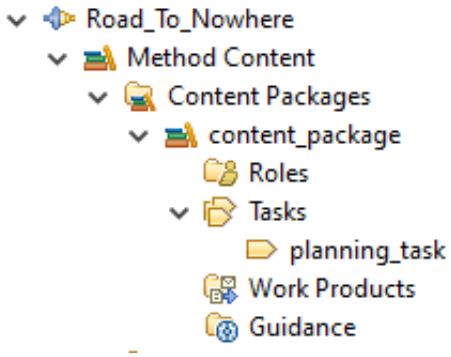
Ukázka kódu 7.6: Generovaný SQL skript pro anti-pattern PMs Who Write Specs

### 7.3.4 Road to Nowhere

Road to Nowhere[5] je anti-pattern, který spočívá v nedostatečném plánování projektu. Plánování je důležité pro správnou organizaci celého projektu a mimo jiné také proto, aby bylo možné jeho úspěch zopakovat, případně se poučit z neúspěchu [31]. Symptom anti-patternu je tedy následující:

- nedostatečné množství plánovacích činností (případně artefaktů) v průběhu projektu

Obrázek 7.11 zobrazuje obsah metody anti-patternu namodelovaný v EPF Composeru. Namodelovaný úkol má následně specifikovaný ještě vzorec pro vyjádření podílu tohoto elementu v projektových datech. Detail tohoto atributu je znázorněný na obrázku 7.1.




Obrázek 7.11: Obsah metody anti-patternu Road to Nowhere v EPF Composeru

V ukázce kódu 7.7 je složitější pro tento anti-pattern generovaný vzorec se specifikací podílu plánovacího úkolu na projektových datech.

```
SELECT (CASE WHEN COUNT(*) > 0 THEN 1 ELSE 0 END) FROM
(SELECT COUNT(*) as sum FROM work_item wi
    JOIN person p ON p.id = wi.authorId
    AND p.projectId = PROJECT_ID
    AND wi.workItemType = 'WORK_UNIT'
    AND (wi.name LIKE '%plan%'))
AS a,
(SELECT COUNT(*) as sum FROM work_item wi
    JOIN person p ON p.id = wi.authorId
    AND p.projectId = PROJECT_ID
    AND wi.workItemType = 'WORK_UNIT')
AS b
```

[5] https://github.com/ReliSA/Software-process-antipatterns-catalogue/blob/master/catalogue/Road_To_Nowhere.md

```
WHERE a.sum < (b.sum/35)
```

Ukázka kódu 7.7: Generovaný SQL skript pro anti-pattern Road to Nowhere

### 7.3.5 Anti-patterny, které definovaným popisem nelze vyjádřit

Z povahy meta-modelu UMA je jasné, že nebude možné popsat všechny možné anti-patterny, ale pouze takové, které lze vyjádřit pomocí jeho metod. Některé typy anti-patternů nejsou vhodné pro popis pomocí EPF Composeru vůbec, některé by bylo možné popsat po doplnění implementovaných rozšíření nebo přidání možnosti dalších pomocných atributů.

Mezi anti-patterny, které by bylo možné po dalších úpravách pomocí EPF Composeru vyjádřit a jejich popisy exportovat, patří například ty, které jsou definovány časovou sousledností. Časovou souslednost úkolů (a tedy i výsledků práce, které jsou vždy nějakému úkolu přiřazeny) by bylo možné v EPF Composeru modelovat v rámci procesu (například *Delivery Process*) pomocí předchůdců (*Predecessors*) ve *Work Beakdown Structure* procesu a následné úpravě implementace rozšíření.

Dalším typem anti-patternů, které zatím nelze z EPF Composeru exportovat, jsou anti-patterny, které definují iterace a příslušnost elementů k nim. Vyjádřit iterace a tyto příslušnosti by mělo být po úpravě impementace rozšíření pro export možné opět v rámci *Work Breakdown Structure* pomocí elementu *Iteration* a vnořených elementů.

Mimo těchto možností lze pomocí datového modelu SPADe nacházet i symptomy, které definují, že jednotlivé elementy byly změněny stejnou konfigurací. Tento vztah bohužel nelze pomocí UMA přirozeně vyjádřit. Pokud by bylo ovšem žádoucí modelovat i tento symptom, bylo by možné společnou konfiguraci pro více elementů definovat například pomocí jejich sdružení v rámci jedné *Custom Category* označené speciálním nově definovaným atributem.

Některé anti-patterny ale nejsou vhodné pro modelování pomocí EPF Composeru ani po případných úpravách. Následuje příklad anti-patternu, který není pomocí EPF Composeru snadné takřka vůbec vyjádřit.

#### Cart Before the Horse

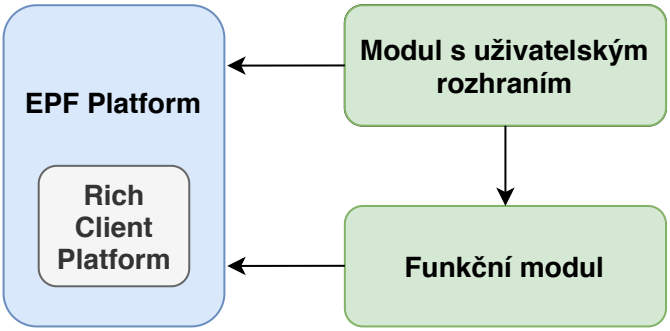
Cart Before the Horse vyjadřuje, že činnosti nejsou vykonávány v naplánovaném pořadí [48]. Lze jej eliminovat analýzou, které činnosti v procesu závisí na jiných, a jejich správným a důsledným plánováním.

Pro popis pomocí EPF Composeru však tento anti-pattern příliš vhodný není. Důvodem je fakt, že pro jeho odhalení je nutné nejprve znát správné

pořadí konkrétních činností daného procesu. Tyto procesy i činnosti v nich se ale projekt od projektu liší a nelze je tedy stanovit univerzálně. Za takových okolností by bylo možné tento anti-pattern v EPF Composeru popsat pouze pro jeden konkrétní proces nebo na míru konkrétnímu projektu. Při testování projektů by pak byla potřeba u každého z nich znalost o použitém procesu.

# 8 Implementace rozšiřujících modulů

Tato kapitola má za cíl popsat implementaci dvou rozšiřujících modulů EPF Composeru, které byly v rámci této práce vytvořeny. Mají za cíl exportovat namodelované anti-patterny způsobem popsaným v kapitole 7. Oba byly vyvíjeny ve vývojovém prostředí Eclipse SDK 2018-09 (4.9) s využitím nástrojů PDE. Oba projekty byly verzovány prostřednictvím systému pro správu verzí Git.[1] Vývoj byl veden oproti poslední verzi EPF Composeru, tedy 1.5.2. Obrázek 8.1 znázorňuje vztah mezi původní platformou EPF a dvěma novými rozšiřujícími moduly.




Obrázek 8.1: Vytvářené rozšiřující moduly ve vztahu k EPF platformě a sobě navzájem.

## 8.1 Rozšiřující modul uživatelského rozhraní

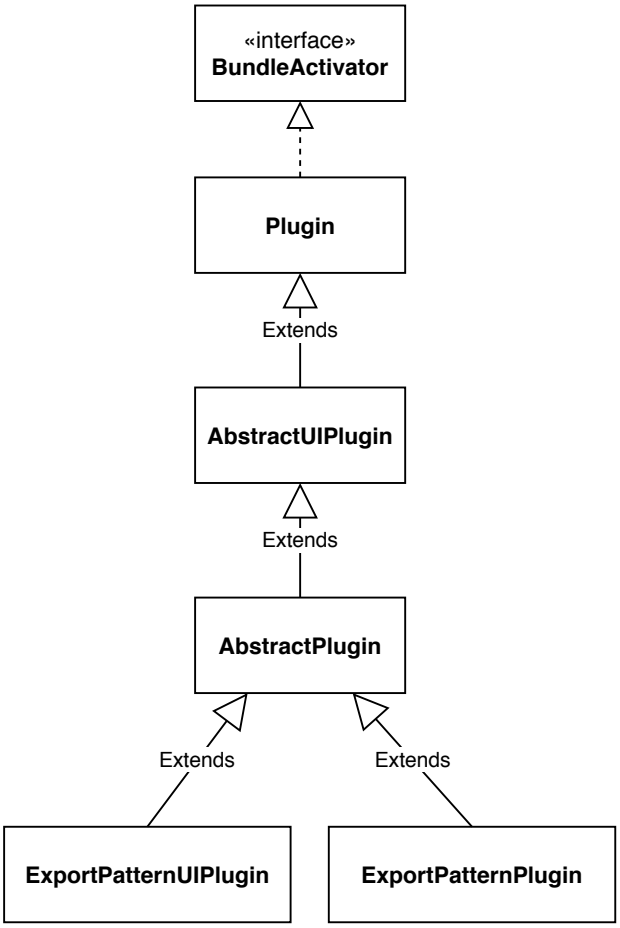
Tento rozšiřující modul se nazývá `cz.zcu.kiv.epf.spade.export.pattern.ui` a slouží pro zajištění grafického uživatelského rozhraní pro druhý, funkční rozšiřující modul. Obsahuje dva balíky, a to `cz.zcu.kiv.epf.spade.export.pattern.ui` a `cz.zcu.kiv.epf.spade.export.pattern.ui.wizards`.

První zmíněný balík obsahuje třídy `ExportPatternUIPlugin` a `ExportPatternUIResources`.

`ExportPatternUIPlugin` je aktivátor tohoto rozšiřujícího modulu. Dědí od abstraktní třídy `AbstractPlugin`, která slouží jako obecný základ pro

[1] https://github.com/ReliSA/EPF-plugin-SPEM-export

všechny rozšiřující moduly EPF. Jelikož třída `AbstractPlugin` dědí od třídy `AbstractUIPlugin`, mohou všechny jeho potomci pracovat s uživatelským rozhraním platformy Eclipse. UML model dědičnosti aktivátorů je znázorněn na obrázku 8.2.




Obrázek 8.2: UML diagram aktivátorů

Aktivátor `ExportPatternUIPlugin` je implementován jako jedináček, jelikož není potřeba vícero jeho instancí a platforma Eclipse sdílenou instanci získá voláním jeho metody `getDefault`. Tato sdílená instance je uložena jako atribut `plugin`. Dále přepisuje metody `start` a `stop`, které jsou využívány pro aktivaci a deaktivaci celého rozšiřujícího modulu.

Ačkoliv tento rozšiřující modul v současné době poskytuje pouze základní anglickou lokalizaci, je implementován tak, aby byl v tomto směru snadno rozšiřitelný. Jednotlivé lokalizačně závislé prvky implementace jsou tak reprezentovány pouze proměnnými. Třída `ExportPatternUIResources` pak slouží pro načítání jejich hodnot ze souboru `Resources.properties`, k čemuž využívá knihovnu `NLS`.

Druhý balík tohoto rozšiřujícího modulu obsahuje třídy specifikující průvodce pro export anti-patternů.

`ExportPatternWizard` je třída představující průvodce jako takového. K tomuto účelu dědí od třídy `BaseWizard` a implementuje rozhraní `IExportWizard`. Třída `BaseWizard` je základem pro všechny průvodce EPF Composeru a rozhraní `IExportWizard` slouží pro specifikaci všech průvodců exportem a přídání průvodce mezi položky menu pro export. Pro inicializaci průvodce a jeho parametrů slouží metoda `init`.

Pomocí metody `addPages` definuje jednotlivé strany průvodce, konkrétně `SelectPluginPage` a `SelectExportOptionsPage`. Pro tyto strany průvodce vytváří instanci třídy `ExportPatternData`, která je součástí implementace druhého rozšiřujícího modulu, popsaného v kapitole 8.2, a slouží pro uložení parametrů, které budou druhým rozšiřujícím modulem využity.

Dále tento průvodce přepisuje metody `canFinish` a `doFinish`. Metoda `canFinish` obsahuje kontrolu, zda je aktuální stránka poslední, uživatel může průvodce zakončit bez další interakce a je možné tedy odblokováno tlačítko *Finish*. Metoda `doFinish` potom slouží k specifikaci akcí, které se mají provést, pokud se uživatel rozhodne průvodce tlačítkem *Finish* zakončit. V tomto případě je volána zvlášť vytvořená metoda `exportPattern`, která volá metodu `export` třídy `ExportPatternService`, která již náleží do funkčního rozšiřujícího modulu popsaného v kapitole 8.2.

`SelectPluginPage` slouží pro výběr plug-inů metod pro export a dědí od stejnojmenné třídy `org.eclipse.epf.export.wizards.SelectPluginPage`. Této třídě přepisuje metodu `getNextPage`, která slouží pro určení následující stránky průvodce.

Následující stránka je implementována třídou `SelectExportOptionsPage`. Ta dědí od `BaseWizardPage`, abstraktní třídy pro všechny stránky průvodců v EPF Composeru. `SelectExportOptionsPage` slouží pro zadání cílového adresáře exportu a v budoucnu by mohla sloužit i pro specifikaci dalších jeho parametrů. Tyto parametry jsou ukládány jako atributy instance třídy `ExportPatternData`, kterou stránka přijímá jako parametr svého konstruktoru. Dále implementuje metodu `createControl`, která definuje rozložení stránky průvodce a jednotlivé elementy uživatelského rozhraní. V tomto případě vytváří pole pro zadání cíle exportu s možností jeho výběru pomocí *file chooseru*. Z této metody je také volána implementace metody `initControls`, která definuje akce pro elementy uživatelského rozhraní, tedy především akci pro aktualizaci cíle exportu v instanci `ExportPatternData` a akci pro otevření *file chooseru* po stisknutí tlačítka *Browse*. Pro zjištění, zda je možné pokračovat na další stránku tlačítkem *Next*, případně průvodce ukončit pomocí tlačítka *Finish*, je implementována metoda `isPageComplete`. Ta kontroluje, zda je

zadána nějaká cílová adresa exportu. K tomu využívá dvě pomocné metody `getPath` a `isValidPath`.

## 8.2 Funkční rozšiřující modul

Tento rozšiřující modul se nazývá `cz.zcu.kiv.epf.spade.export.pattern` a slouží pro definování funkcionalit exportu anti-patternů. Je rozdělen do dvou balíků, a to `cz.zcu.kiv.epf.spade.export.pattern` a `cz.zcu.kiv.epf.spade.export.pattern.domain`. První zmíněný rozšiřující modul obsahuje třídy poskytující funkcionality, zatímco druhý balík třídy definující vlastní datový model.

Aktivátor tohoto rozšiřujícího modulu `ExportPatternPlugin` taktéž dědí od abstraktní třídy `AbstractPlugin`, jak je naznačeno na obrázku 8.2.

Pro předání informací mezi rozšiřujícím modulem uživatelského rozhraní a funkčním rozšiřujícím modulem slouží třída `ExportPatternData`. Ta v současné době obsahuje jeden atribut, a to `directory` pro adresu cíle exportu.

Vstupním bodem tohoto rozšiřujícího modulu je třída `ExportPatternService` a její metoda `export`. Ta má na vstupu instanci právě výše popsané třídy `ExportPatternData`. Tato metoda zajišťuje volání metody pro mapování plug-inů metod na vnitřní datový model a následně i volání správného generátoru, v tomto případě generátoru SQL skriptů. Tento přístup byl zvolen z toho důvodu, aby byl rozšiřující modul snadno rozšiřitelný o jiné generátory. V případě vícero generátorů by tato metoda rozhodovala o volání těch, které byly zvoleny například pomocí uživatelského rozhraní.

Zároveň třída `ExportPatternService` v konstruktoru inicializuje logger `ExportPatternLogger`, který je využit v obou využívaných třídách. Tento logger dědí od třídy `FileLogger`, která umožňuje logování na třech úrovních severity – *info*, *warning* a *error*. `ExportPatternLogger` vytváří záznamy do souborů umístěných v adresáři `logs/export/pattern` v kořenovém adresáři příslušné aplikace EPF Composer.

### 8.2.1 Mapování na vlastní datový model

Vlastní datový model tohoto rozšiřujícího modulu byl vytvořen proto, že více odpovídá potřebám aplikace a následná manipulace s ním a získávání konkrétních dat z něj je podstatně snazší, než v případě UMA.

Tento datový model obsahuje třídy reprezentující základní aspekty popisu anti-patternů. Kořenovým elementem sdružujícím informace z jednoho pluginu metody je `PatternProject`. Úkoly jsou reprezentovány jako `PatternTask`

a role `PatternRole`. Artefakty jsou definovány pomocí třídy `PatternArtifact`, výstupy, zatím především commity, jako `PatternOutcome`. Tyto dvě třídy reprezentující výstupy práce mají společného předka `PatternWorkProduct`. Elementy, které uchovávají informaci o jedinečném identifikátoru GUID, implementují rozhraní `Descriptable`. Disciplíny a fáze jsou reprezentovány pomocí `PatternDiscipline` a `PatternPhase`. Ty však zatím nejsou z důvodů popsaných v kapitole 7.1.8 při generování využívané.

Pro samotné mapování slouží třída `ExportPatternMapService`. Jejím vstupním bodem je metoda `map`, která jako vstup přijímá kolekci plug-inů metod meta-modelu UMA a vrací seznam instancí třídy `PatternProject`. Pro získání potřebných dat z plug-inů metod je využito metod z tříd `TngUtil` a `ProcessUtil`. `TngUtil` je využita pro získání disciplín a procesu z plug-inu metody a `ProcessUtil` k získání seznamu úkolů, který je následně využit k zjištění dalších elementů, jako jsou role nebo výsledky práce. `ExportPatternMapService` dále zajišťuje i parsování *Main Description* u výsledků práce a úkolů a *Brief Description* u plug-inů metod, které následně ve vhodné podobě ukládá do vlastního datového modelu popsaného výše. Klíčová slova jsou pro dané elementy ukládána do pole `tokens`. Speficikace podílu je uložena jako řetězec a parsována až v SQL generátoru, a to z toho důvodu, že jiné generátory by případně vyžadovaly jiný způsob reprezentace.

### 8.2.2 Generování SQL skriptů

Aby byl vytvářený rozšiřující modul co se týká možností exportu snadno rozšiřitelný, bylo vytvořeno obecné rozhraní pro všechny generátory `IExportPatternSpecificService`. To definuje jednu metodu `export`, která má na vstupu seznam instancí `PatternProject` a nic nevrací.

Pro generování SQL skriptů slouží třída `ExportPatternSQLService` implementující rozhraní `IExportPatternSpecificService`. Metoda `export` v tomto případě pro každou instanci `PatternProject` volá metodu `createSQLScript`, která má za úkol vytvořit jeden SQL soubor s dotazy. Ta pro každý `PatternTask` volá metodu `generateSql`, která generuje jeden SQL dotaz. V závislosti na tom, zda se jedná o obyčejný nebo „invertovaný“, určí návratové hodnoty tohoto dotazu. Následně rozhodne, zda bude generován „obyčejný“ dotaz, nebo dotaz vyjadřující podíl v projektových datech. K tomu slouží metody `generateBasicSql` a `generateRateSql`.

Metoda `generateBasicSql` vytváří dva seznamy – seznam s konstrukcemi pro spojení `JOIN` a seznam pro podmínky `AND`. Následně volá metody zpracovávající jednotlivé elementy, jako jsou role, výsledky práce nebo úkoly. V těchto metodách jsou tyto seznamy doplňovány na základě podmínek

získaných z dodaných elementů. Nakonec je zavolána metoda `assembleBasicQuery`, která na základě těchto dvou seznamů složí prototyp základního dotazu.

Protože je složitější dotaz na podíl určitých elementů v projektových datech složen ze dvou rozdílných jednodušších poddotazů, využívá metoda `generateRateSql` volání metody `generateBasicSql`. První poddotaz je téměř totožný s obdobným „obyčejným“ dotazem. Pro vytvoření druhého poddotaazu je také využita metoda `generateBasicSql`, ovšem s tím rozdílem, že metodě nejsou dodány konkretizující atributy, které dotazu přidávaly podmínky například specifikací klíčových slov.

V rámci práce byl vytvořen i prototyp generování XML souborů, který se nachází v samostatné větvi `xml_generating`. V rámci tohoto prototypu má uživatel na výběr z exportu dat do SQL, XML, nebo obojího.

## 8.3 Testování

Implementované rozšiřující moduly byly testovány sadou jednotkových testů. Pro testování uživatelského rozhraní bylo využité uživatelské testování. Uživatelské testování bylo provedeno trojicí testerů. Všichni pochází z řad studentů softwarového inženýrství, tudíž jsou jejich znalosti pro posouzení vhodnosti uživatelského rozhraní dostačující. Ti s využitím již namodelovaných anti-patternů a testovacím scénářem definujícím základní případ užití aplikace uživatelské rozhraní vyzkoušeli. Během tohoto testování nebyly odhaleny žádné chyby a bylo navrženo vylepšení aplikace spočívající v nabídce naposledy použitých adresářů pro specifikaci cíle exportu.

Komplexní funkčnost byla ověřena pomocí sady namodelovaných anti-patternů a porovnání výsledků získaných pomocí SQL skriptů vygenerovaných na základě těchto modelů s manuálním auditem projektů v příslušném ALM nástroji. Tento způsob ověření plně testuje funkčnost rozšiřujícího modulu pro export a je popsán v kapitole 9.

### 8.3.1 Jednotkové testy

Pro jednotkové testování byl vytvořen samostatný fragment `cz.zcu.kiv.epf.spade.export.pattern.test`. Tento způsob umístění testů vůči hlavnímu rozšiřujícímu modulu byl zvolen z důvodu, že fragment není potřeba exportovat spolu s hostitelským rozšiřujícím modulem, ale zároveň má přístup i k neveřejným metodám jeho tříd. Z toho důvodu bylo ale potřeba samotné testy umístit do adresáře `cz.zcu.kiv.epf.spade.export.pattern`, aby názvem odpovídal adresáři testovaných tříd.

K testování byla využita knihovna JUnit4, která byla do projektu dodána pomocí Maven závislosti. Takto vytvořenými jednotkovými testy bylo otestován export anti-patternů.

## 8.4 Možná další vylepšení rozšiřujících modulů

Rozšiřující moduly by bylo vhodné dále vylepšovat tak, aby bylo možné popisovat více druhů anti-patternů podle více kritérií. Tyto popisy jsou limitovány možnostmi EPF Composeru. Konkrétně těmito rozšířeními mohou být:

- komplexnější vzorce pro vyjádření podílů
- v návaznosti na úpravu datového skladu přidání možnosti tvorby kritérií pro domény a fáze
- export časových sousledností
- export iterací
- specifikace konkrétních ID projektů, pro které se mají SQL skripty generovat
- možnost přímo provést SQL dotazy nad datovým skladem SPADe a zobrazit výsledky
- více formátů exportu pro použití v jiných aplikacích
- jazykové lokalizace
- nabídka naposledy použitých adresářů pro specifikaci cíle exportu

# 9 Ověření přístupu

V rámci této práce bylo v EPF Composeru namodelováno několik anti-patternů a byly pro ně vygenerovány SQL skripty. Tato kapitola shrnuje výsledky aplikace těchto výsledných SQL skriptů na datech z datového skladu SPADe.

## 9.1 Výsledky oproti datovému skladu

Jak již bylo zmíněno výše, v EPF Composeru bylo modelováno několik anti-patternů. Kritéria pro výběr anti-patternů, které lze této nástrojem modelovat, byly popsány v kapitole 7.1. Modely vybraných anti-patternů jsou také součástí DVD přiloženého k této práci. Některé z nich jsou popsané v kapitole 7.3.

SQL skripty vygenerované pomocí naimplementovaných rozšiřujících modulů byly testovány na projektových datech z datového skladu SPADe. Testovacími projekty jsou semestrální práce z předmětu KIV/ASWI z roku 2017, které byly spravovány pomocí ALM nástroje Redmine. Výsledky tohoto testování jsou v tabulce 9.1. Tyto výsledky jsou detekcí anti-patternů v datovém skladu SPADe a jsou tedy závislé na správnosti těchto dat. Jinými slovy, tyto výsledky reprezentují přítomnost anti-patternů v natěžených datech projektu a mohou se lišit od skutečného stavu v ALM nástrojích, jak je popsáno dále v kapitole 9.2.

Vzhledem k tomu, že se jedná o školní projekty, bylo možné nahlédnout i do projektů přímo v Redmine a porovnat tak skutečný stav přítomnosti anti-patternů s tím, který byl zjištěn pomocí aplikace SQL skriptů v datovém skladu SPADe. Tento manuální audit je popsán v kapitole 9.2.

## 9.2 Porovnání s manuálním auditem

Manuální audit projektů v Redmine odhalil rozdíly mezi detekcí anti-patternů v databázi SPADe a původními projekty na Redmine, tedy skutečným stavem projektů. Jeho výsledky jsou v tabulce 9.2.

Jedním z důvodů je způsob dolování Wiki stránek z ALM nástrojů. Ty jsou kvůli technickým okolnostem dolovány metodou web-crawlingu. Tato technika není obecně spolehlivá a v důsledku toho některé Wiki stránky v databázi chybí.

Tabulka 9.1: Výsledky oproti datovému skladu – symbol „+“ značí přítomnost anti-patternu v datech projektu, „-“ jeho nepřítomnost

| **Pattern / ID projektu** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |
|---|---|---|---|---|---|---|---|---|---|---|
| **Absent Artifact** | + | + | - | + | + | + | - | + | - | + |
| **Architects Don't Code** | - | - | - | - | - | - | - | - | - | - |
| **Business as Usual** | - | - | - | - | - | - | - | - | - | - |
| **Detailitis Plan** | - | - | - | - | - | - | - | - | - | - |
| **PMs Who Write Specs** | - | - | + | - | + | - | + | - | + | - |
| **Road to Nowhere** | - | - | - | - | - | - | - | - | - | - |
| **Specify Nothing** | + | + | - | - | + | + | - | + | - | + |

Dalším problémem je použití plug-inu DMS v některých projektech. Ten slouží pro uložení a správu souborů. Není však přítomen na všech instancích Redmine a proto není možné na jeho přítomnost při dolování dat spoléhat. Kromě toho je při implementaci datové pumpy kladen co největší nárok na její univerzálnost. Z těchto důvodů nejsou zatím artefakty uložené pomocí tohoto plug-inu dolovány.

Obdobně problematické je ukládání artefaktů na jiná úložiště mimo Redmine, například Google Docs, na které je z obsahu Wiki stránek pouze odkazováno.

Tabulka 9.2: Manuální audit Redmine – symbol „+“ značí přítomnost anti-patternu v projektu, „-“ jeho nepřítomnost

| **Pattern / ID projektu** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |
|---|---|---|---|---|---|---|---|---|---|---|
| **Absent Artifact** | - | - | - | - | - | + | - | + | - | - |
| **Architects Don't Code** | - | - | - | - | - | - | - | - | - | - |
| **Business as Usual** | - | - | - | - | - | - | - | - | - | - |
| **Detailitis Plan** | - | - | - | - | - | - | - | - | - | - |
| **PMs Who Write Specs** | - | - | + | - | + | - | + | - | + | - |
| **Road to Nowhere** | - | - | - | - | - | - | - | - | - | - |
| **Specify Nothing** | - | - | - | - | - | - | - | + | - | - |

Z výsledků je patrné, že mnoho anti-patternů nebylo detekováno v žádném z projektů. To je dáno pravděpodobně tím, že se jedná o školní projekty, které byly vypracovávány v rámci předmětu zabývajícím se procesním rámcem vývoje software a při zpracování těchto projektů byl na toto téma kladen důraz. Výjimku tvoří především anti-pattern PMs Who Write Specs, který byl zaznamenán u téměř poloviny zkoumaných projektů. Zde je příčinou malá velikost týmů, které se na projektech podílely, a bylo tudíž nutné, aby

členové týmu zastávali více rolí zároveň.

Na základě porovnání výsledků aplikace SQL skriptů v datovém skladu a výsledků manuálního auditu lze usoudit, že vygenerované SQL skripty dokázaly správně odhalit ty anti-patterny, pro které existovaly v datovém skladu SPADe odpovídající data. Problém v identifikaci byl především u těch anti-patternů, které měly některá data v Redmine uloženy nestandardním způsobem. Taková data nemohla být natěžena datovou pumpou SPADe a následně nalezena pomocí vygenerovaných SQL skriptů.

# 10 Závěr

Cílem této práce bylo navrhnout způsob popisu softwarových procesních anti-patternů pomocí meta-modelu SPEM a využitím jeho implementace v nástroji EPF Composer a dále vytvořit rozšíření EPF Composeru takovým způsobem, aby bylo možné tyto popisy ve vhodném formátu exportovat tak, aby je bylo možné využít k analýze dat projektů, natěžených nástrojem SPADe.

Navržený způsob popisu anti-patternů v EPF Composeru byl popsán v kapitole 7. Jelikož UMA a EPF Composer slouží především k popisu procesů a datový model SPADe vychází ze SPEM 2.0 používaného také k modelování procesů, vychází navrhovaný způsob popisu ze základních prvků procesu, tedy rolí, úkolů a výsledků práce. I přesto, že byl tento model doplněn o další potřebné parametry, je vhodný pouze pro typy anti-patternů, které lze pomocí těchto prvků vyjádřit.

Dále byla v rámci této práce implementována sada dvou rozšiřujících modulů pro EPF Composer, které slouží k exportu popisů anti-patternů do vhodného formátu. Výsledný export má pro každý namodelovaný anti-pattern podobu SQL skriptu s jedním nebo několika dotazy, které lze po doplnění ID testovaného projektu použít pro hledání anti-patternu v projektových datech.

Jeden z rozšiřujících modulů zajišťuje grafické uživatelské rozhraní a druhý rozšiřující modul funkční část. Implementace obou rozšiřujících modulů, v rámci které byl kladen důraz na další rozšiřitelnost, je popsána v kapitole 8. Rozšiřující moduly byly testovány kombinací jednotkových a uživatelských testů a analýzou výsledků SQL skriptů na přítomnost anti-patternů a manuálního auditu jednotlivých projektů z datového skladu SPADe.

Porovnáním výsledků SQL skriptů na projektových datech z datového skladu SPADe a manuálního auditu příslušných projektů v Redmine bylo zjištěno, že takto vygenerované skripty dokáží namodelované anti-patterny v projektových datech odhalit.

Výsledek práce v podobě článku *SPEM-Based Process Anti-Pattern Models for Detection in Project Data* byl přijat na recenzovanou konferenci Euromicro SEAA 2020.

Všechny body zadání byly splněny.

# Přehled zkratek

**ALM** Application Lifecycle Management

**API** application programming interface

**BPMN** Business Process Modeling and Notation

**BPMI** Business Process Management Initiative

**DAO** data access object

**DMS** Document Management System

**EE** Enterprise Edition

**EMF** Eclipse Modeling Framework

**EPF** Eclipse Process Framework

**ETL** Extract-Transform-Load

**EUP** Enterprise Unified Process

**FAV** Fakulta aplikovaných věd

**GUID** globally unique identifier

**HTML** Hypertext Markup Language

**IBM** International Business Machines Corporation

**IDE** integrated development environment

**JAR** Java Archive

**KIV** Katedra informatiky a výpočetní techniky

**MOF** Meta Object Facility

**OMG** Object Management Group

**OSGi** Open Services Gateway initiative

**OSLC** Open Services for Lifecycle Collaboration

**PDE** Plug-in Development Environment

**PDF** Portable Document Format

**RCP** rich client platform

**RMC** Rational Method Composer

**RTC** Rational Team Concert

**RUP** Rational Unified Process

**SEAA** Software Engineering and Advanced Applications

**SEWL** Software Engineering Workflow Language

**SPADe** Software Process Anti-pattern Detector

**SPEM** Software & Systems Process Engineering Meta-Model

**SQL** Structured Query Language

**SVN** Apache Subversion

**SWT** Standard Widget Toolkit

**UMA** Unified Method Architecture

**UML** Unified Modeling Language

**UP** Unified Process

**VCS** version control system

**XMI** XML Metadata Interchange

**XML** Extensible Markup Language

**ZČU** Západočeská univerzita v Plzni

# Seznam obrázků

# Seznam tabulek

# Seznam ukázek kódu

# Literatura


[1] *IBM Solution Design Method* [online]. Dostupné z: `https://files.ifi.uzh.ch/rerg/amadeus/teaching/courses/it_architekturen_hs10/Solution_Design_I.day.pdf`.

[2] Sommerville, I. *Softwarové inženýrství*. Albatros Media, 2013. ISBN 978-80-251-3826-7.

[3] Van Casteren, W. The Waterfall Model and the Agile Methodologies : A comparison by project characteristics. 03 2017. doi: 10.13140/RG.2.2.36825.72805.

[4] Moniruzzaman, A. B. M. – Hossain, S. Comparative Study on Agile software development methodologies. *Global Journal of Computer Science and Technology*. 07 2013.

[5] Züllighoven, H. *Object-Oriented Construction Handbook*. Morgan Kaufmann, 2004. ISBN 9781558606876.

[6] Osis, J. – Donins, U. Chapter 2 – Software Designing With Unified Modeling Language Driven Approaches. In *Topological UML Modeling*, Computer Science Reviews and Trends. Boston: Elsevier, 2017. s. 53 – 82. doi: 10.1016/B978-0-12-805476-5.00002-2. ISBN 978-0-12-805476-5.

[7] *Rational Unified Process – Best Practices for Software Development Teams*. Rational Software, 1998.

[8] Kruchten, P. *The Rational Unified Process–An Introduction*, s. 255. 01 2000. ISBN 0321197704.

[9] Ambler, S. W. *Introduction to the Enterprise Unified Process (EUP)*. 10 2005.

[10] Cossentino, M. – Hilaire, V. – Seidita, V. *The OpenUp process*, s. 491–566. 01 2014. doi: 10.1007/978-3-642-39975-6___15.

[11] Ambler, S. W. – Holitza, M. *Agile For Dummies*. John Wiley Sons, Inc., 2012. ISBN 978-1-118-30506-5.

[12] Schwaber, K. Scrum development process. *Proceedings of the 10th Annual ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*. 01 1995, s. 117–134.

[13] Hanslo, R. – Vahed, A. – Mnkandla, E. *Quantitative Analysis of the Scrum Framework*, s. 82–107. 01 2020. doi: 10.1007/978-3-030-37534-8_5. ISBN 978-3-030-37533-1.

[14] Tran, L. *The Scrum Methodology* [online]. Dostupné z: `https://www.inloox.com/company/blog/articles/the-scrum-methodology/`.

[15] OMG. *Meta-Modeling and the OMG Meta Object Facility (MOF)* [online]. 03 2017. Dostupné z: `https://www.omg.org/ocup-2/documents/Meta-ModelingAndtheMOF.pdf`.

[16] Atkinson, C. – Kuhne, T. Model-driven development: a metamodeling foundation. *IEEE Software*. 2003, 20, 5, s. 36–41. doi: 10.1109/ms.2003.1231149.

[17] Peixoto, D. et al. A Comparison of BPMN and UML 2.0 Activity Diagrams. 01 2008.

[18] Campos, A. – Oliveira, T. Software Processes with BPMN: An Empirical Analysis. s. 338–341, 06 2013. doi: 10.1007/978-3-642-39259-7_29.

[19] Budinsky, F. et al. *Eclipse Modeling Framework*. Addison Wesley, 2003. ISBN 0-13-142542-0.

[20] Gronback, R. C. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Pearson Education, Inc., 2009. ISBN 978-0-321-53407-1.

[21] *Software and Systems Process Engineering Meta-Model Specification (SPEM)*. Object Management Group, 2008.

[22] Nikulsins, V. – Nikiforova, O. Tool Integration to Support SPEM Model Transformations in Eclipse. *J. Riga Technical University*. 01 2010, 41, s. 60–67. doi: 10.2478/v10143-010-0025-7.

[23] García-Magariño, I. et al. *INGENIAS-SCRUM Development Process for Multi-Agent Development*, 50, s. 108–117. 09 2008. doi: 10.1007/978-3-540-85863-8_14. ISBN 978-3-540-85862-1.

[24] Haumer, P. *Eclipse Process Framework Composer – Part 1: Key Concepts*, 04 2007. Dostupné z: `https://www.eclipse.org/epf/general/EPFComposerOverviewPart1.pdf`.

[25] *Eclipse Process Framework (EPF) Composer 1.0 Architecture Overview* [online]. Dostupné z: https://www.eclipse.org/epf/composer_architecture/.

[26] Krebs, J. – Shuja, A. K. *IBM® Rational Unified Process® Reference and Certification Guide: Solution Designer*. IBM Press, 2007. ISBN 978-0-13-156292-9.

[27] Tuft, B. *Eclipse Process Framework (EPF) Composer – Installation, Introduction, Tutorial and Manual*, 2010. Dostupné z: https://www.eclipse.org/epf/general/EPF_Installation_Tutorial_User_Manual.pdf.

[28] Coplien, J. O. – Harrison, N. B. *Organizational patterns of agile software development*. Prentice-Hall, Inc., 2004. ISBN 978-0131467408.

[29] Alexander, C. – Ishikawa, S. – Silverstein, M. *A pattern language: towns, buildings, construction*. Oxford University Press, 1977. ISBN 0-19-501919-9.

[30] Ambler, S. W. *An Introduction to Process Patterns* [online]. Dostupné z: http://www.ambysoft.com/downloads/processPatterns.pdf.

[31] Laplante, P. A. – Neill, C. J. *Antipatterns – Identification, Refactoring, and Management*. CRC Press, 2006. ISBN 0-8493-2994-9.

[32] Brown, W. J. et al. *AntiPatterns – Refactoring Software, Architectures, and Projects in Crisis*. Robert Ipsen, 1998. ISBN 0471197130.

[33] Charikleia, R. et al. *Management Anti-patterns in Finnish Software Industry* [online]. Dostupné z: https://www.sis.uta.fi/~tp54752/pub/ManagementAnti-patternsinFinnishSoftwareIndustry.pdf.

[34] *Standing On The Shoulders Of Midgets* [online]. Dostupné z: https://wiki.c2.com/?StandingOnTheShouldersOfMidgets.

[35] Chappell, D. *What is Application Lifecycle Management?* [online]. Dostupné z: http://www.davidchappell.com/writing/white_papers/What_is_ALM_v2.0--Chappell.pdf.

[36] Pícha, P. Datový model nástroje SPADe a jeho mapování na projektová data z ALM nástrojů. 2016.

[37] Pícha, P. – Brada, P. ALM Tool Data Usage in Software Process Metamodeling. s. 1–8, 08 2016. doi: 10.1109/SEAA.2016.37.

[38] McAffer, J. – Lemieux, J.-M. *Eclipse: Rich Client Platform*. Addison-Wesley Professional, 2006. ISBN 0-321-33461-2.

[39] Clayberg, E. – Rubel, D. *Eclipse: Building Commercial-Quality Plug-Ins*. Addison-Wesley Professional, 2006. ISBN 0-321-42672-X.

[40] Castro Alves, A. *OSGi in Depth*. Manning Publications Co., 2012. ISBN 9781935182177.

[41] Alliance, O. *Architecture* [online]. Dostupné z: `https://www.osgi.org/developer/architecture/`.

[42] Volanakis, E. *Architecture* [online]. Dostupné z: `HowtotracklifecyclechangesofOSGibundles`.

[43] Beaton, W. *What is a Rich Client?* [online]. Dostupné z: `https://blogs.eclipse.org/post/wayne-beaton/what-rich-client`.

[44] *Eclipse Plugin Development Environment (PDE)* [online]. Dostupné z: `https://projects.eclipse.org/projects/eclipse.pde`.

[45] Janoch, V. Rozšírení editoru procesu vývoje software. Diplomová práce, FAV ZČU, 2019.

[46] *Architects Dont Code* [online]. Dostupné z: `https://wiki.c2.com/?ArchitectsDontCode`.

[47] Malik, N. *Project Management AntiPattern – PMs who write specs* [online]. Dostupné z: `https://blogs.msdn.microsoft.com/nickmalik/2006/01/03/project-management-antipattern-pms-who-write-specs/`.

[48] Winkler, R. *7 project management anti-patterns and how to avoid them* [online]. Dostupné z: `https://www.catalyte.io/7-project-management-anti-patterns-how-to-avoid-them/`.

# A Obsah DVD

DVD nosič přiložený k této práci obsahuje:

- adresář `examples` – projekt EPF Composeru s příklady popisů anti-patternů
- adresář `plugins` – přeložené JAR soubory rozšiřujících modulů určené k použití s EPF Composerem
- adresář `sql` – příklady vygenerovaných SQL skriptů
- adresář `text` – tento dokument ve formátu PDF a jeho zdrojové LaTeXsoubory v adresáři `latex`
- adresář `workspace` – zdrojové kódy rozšiřujících modulů
- soubor `poster.pdf` – poster
- soubor `README.txt` – tento rozpis obsahu nosiče

# B Přehled atributů

Tato příloha shrnuje atributy podporované v popisech anti-patternů nástrojem EPF Composer.

**Keywords**

- zápis: `keywords=<seznam klíčových slov>`
- specifikuje se v poli *Main Description*
- podrobněji v kapitole 7.1.3

**Type**

- zápis: `type=<typ>`
- určený pro specifikaci commitů – v takovém případě nabývá hodnoty `COMMIT`
- pro výsledky práce typu *Outcome*
- specifikuje se v poli *Main Description*
- podrobněji v kapitole 7.1.5

**Amount**

- zápis: `amount=<znaménko porovnání>this.type/<dělitel definující poměr>`
- pro specifikaci poměru úkolů (*tasky*) a výsledků práce v projektových datech
- specifikuje se v poli *Main Description*
- podrobněji v kapitole 7.1.4

**Pattern**

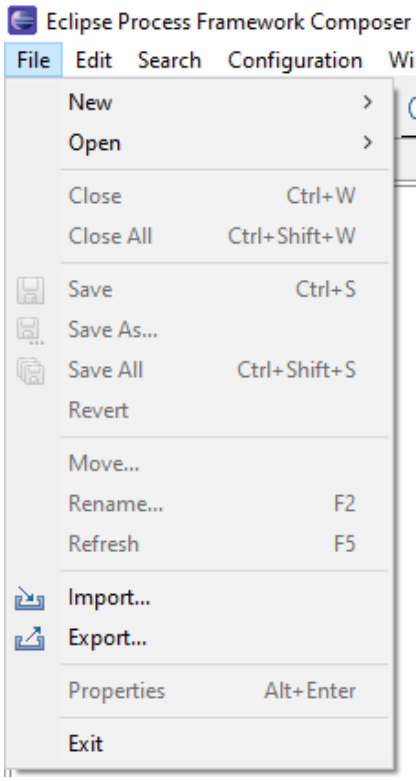
- zápis: `pattern=<true/false>`
- určený pro „invertovaných“ anti-patternů – v takovém případě nabývá hodnoty `true`

- hodnotu `false` není potřeba specifikovat, je výchozí
- specifikuje se v *Brief Description* plug-inu metody
- podrobněji v kapitole 7.1.2

# C Uživatelský manuál

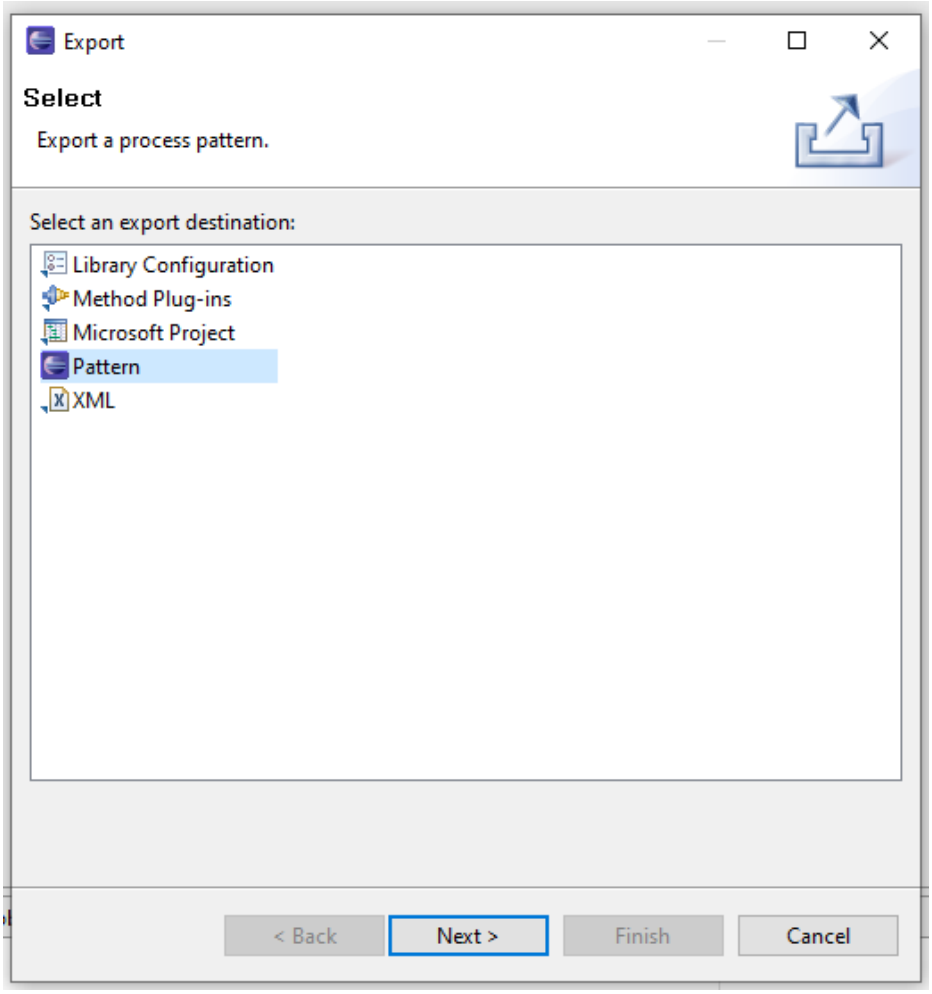
Tento uživatelský manuál obsahuje informace o základním způsobu použití vytvořeného rozšíření.

Pro export anti-patternů (plug-inů metod) z EPF Composeru je potřeba zvolit položku hlavního menu *File*, jak je naznačeno na obrázku C.1. Po jejím výběru se zobrazí kontextové menu, které již nabízí přímo možnost *Export...*.



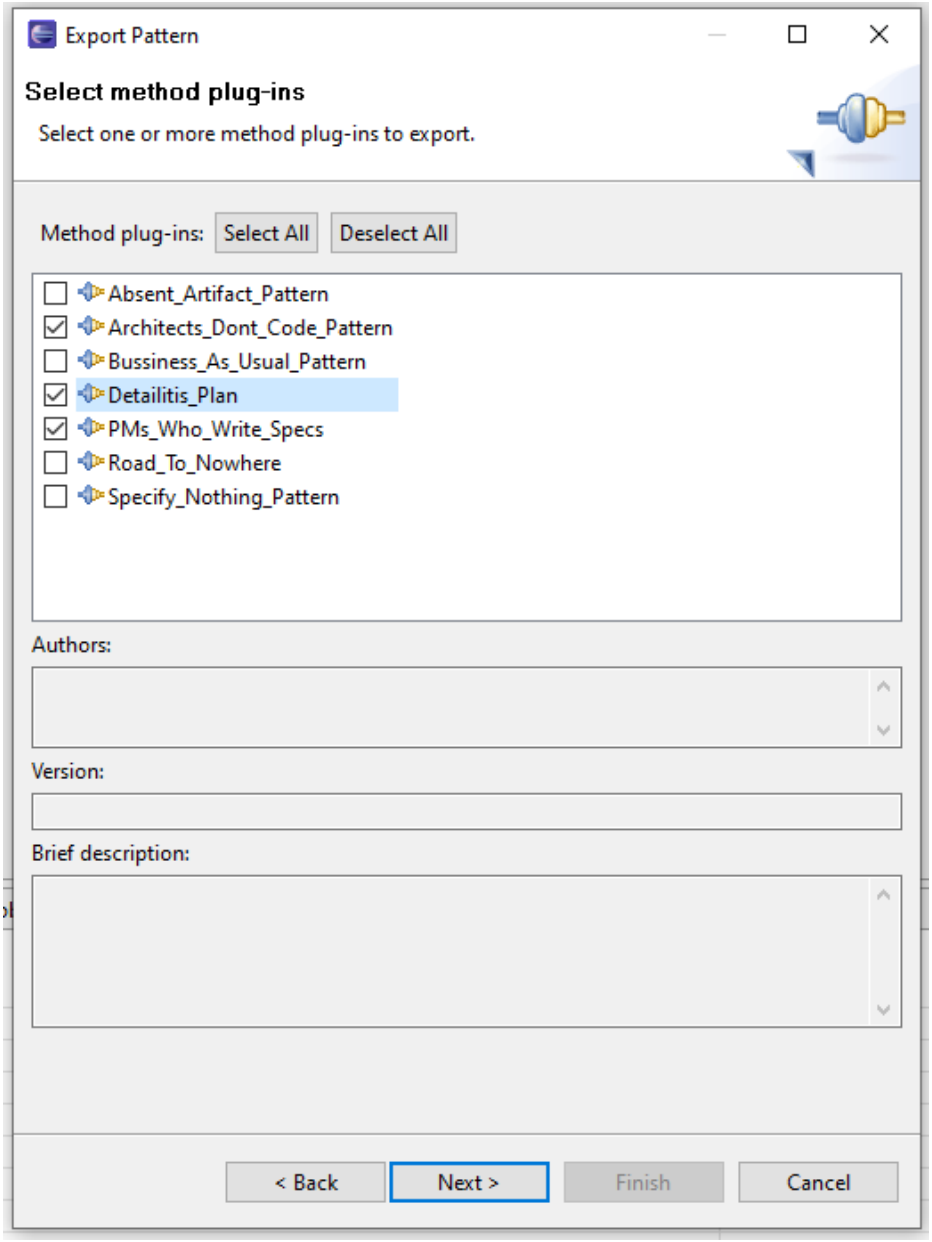

Obrázek C.1: Menu s položkou *Export*

Po jejím výběru se již spustí průvodce exportem. Jeho první stránka je na obrázku C.2. Slouží k výběru možností exportu a uživatel zde má na výběr i možnost exportu anti-patternů nazvanou *Pattern*.



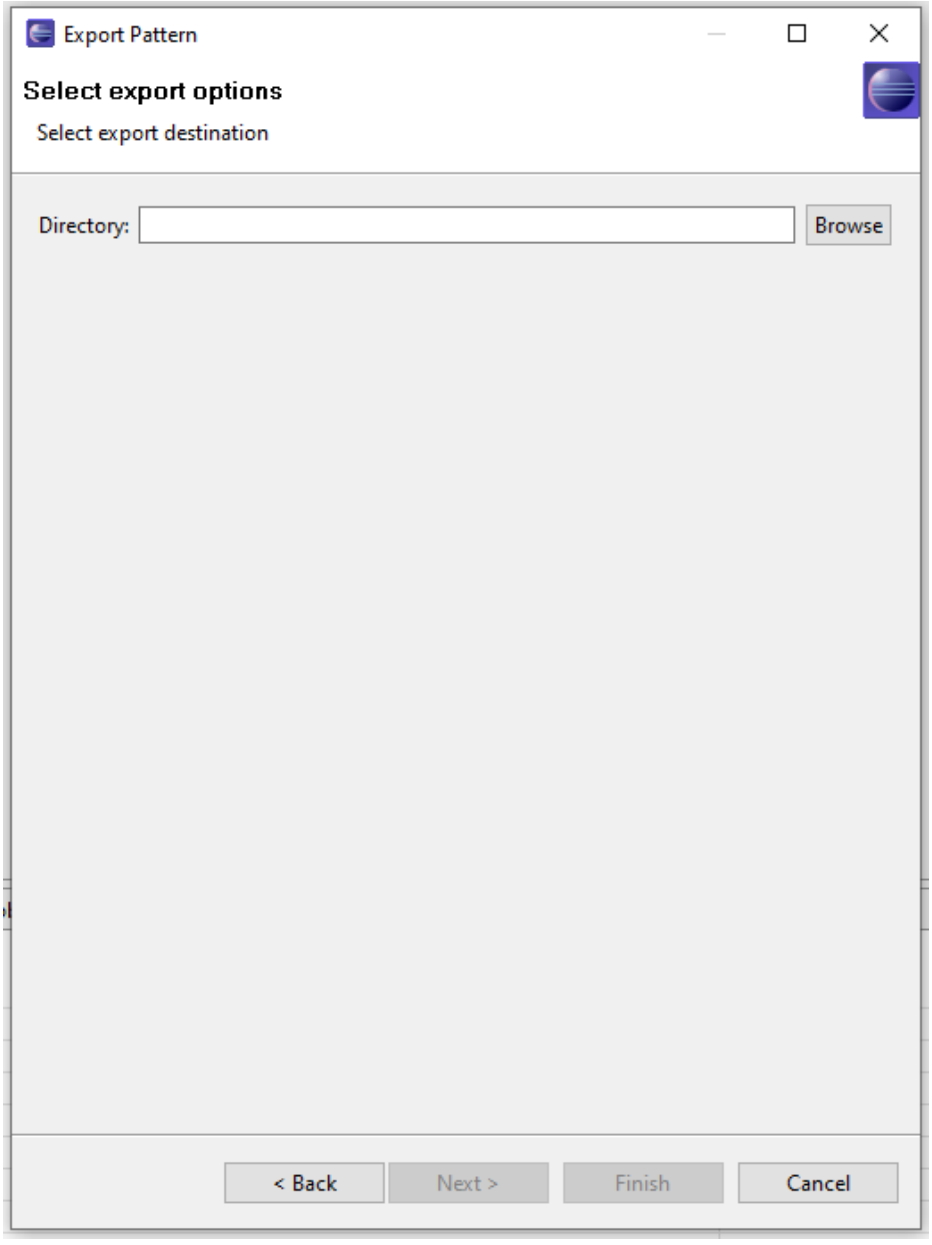

Obrázek C.2: Výběr z možností exportu

Po potvrzení této volby tlačítkem *Next >* je zobrazena další stránka průvodce vyzývající k výběru anti-patternů (plug-inů metod), které si uživatel přeje exportovat. Tato stránka je na obrázku C.3. Uživatel má možnost jednoduše zvolit výběr všech nabízených anti-patternů najednou pomocí tlačítka *Select All*, případně zrušení veškerých výběrů tlačítkem *Deselect All*. Zároveň si může při označení anti-patternu zobrazit informace o autorech, verzi nebo jeho popisek, pokud byly tyto informace uvedeny. Pokračovat na další stránku průvodce lze opět tlačítkem *Next >*.




Obrázek C.3: Výběr plug-inů metod pro export

Poslední stránka průvodcem je na obrázku C.4 a slouží pro zadání adresáře, kam se mají popisy exportovat. Pro usnadnění je možné jej najít pomocí *file chooseru*, který se otevře po stisknutí tlačítka *Browse*. Pokud je adresa vyplněna, je možné průvodce zakončit stisknutím tlačítka *Finish*. Vygenerované SQL skripty jsou nyní k nalezení v příslušném adresáři.




Obrázek C.4: Specifikace cílového adresáře