Software architecture recovery - Django

Kateřina Mušková Maroš Vančo

Fakulta informatiky a informačných technológií
Slovenská technická univerzita v Bratislave

14. prosince 2021

Abstrakt

Webové frameworky sú dnes nepostradateľným pomocníkom pri tvorbe webových stránok. Existuje veľké množstvo návodov ako s nimi správným zpôsobom pracovať, ale len minimum publikáci skúma do hĺbky ich vnútornú štruktúru. Táto práca má za cieľ priblížiť fungovanie dvoch z hlavných častí populárneho frameworku Django, a to Django Template Language a Objektovo relačné mapovanie.

1 Úvod

Táto práca sa sústredí na architektúru webového frameworku Django. Ide o projekt napísaný v programovacom jazyku Python, ktorý v dnešnej dobe používajú aj globálne spoločnosti. Príkladom môže byť Instagram, Bitbucket, alebo Mozilla. Má veľmi obšírnu dokumentáciu, je kvalitne otestovaný a pravidelne updatovaný.

Na rozdiel od ostatných webových frameworkov, ktoré používajú návrhový vzor Model-View-Controller, Django je založený na architektúre Model-View-Template [1], ktorá vycháza z prvého spomenutého. Táto práca je zameraná na architektúru a fungovanie jazyka Django Template Language, ktorý je implementovaný v časti Template a tiež na Objektovo relačné mapovanie v časti Model.

V sekcii 2 je zhrnutý základný princíp fungovania návrhového vzoru Model-View-Template. Ďalšia sekcia 3 všeobecne popisuje štruktúru návrhových vzorov. Na ňu nadväzuje hlavná časť tejto práce, sekcia 4 zaoberajúca sa jazykom Django Template Language (DTL). V rámci nej sú popísané konkrétne výskyty návrhových vzorov zo sekcie 3. V sekcii 5 je podrobne popísaná architektúra Objektovo relačného mapovania (ORM). Záverečná časť (sekcia 7) nakoniec obsahuje zhrnutie opisujúce poznatky získané skúmaním implementácií návrhových vzorov v DTL.

2 Model-View-Template

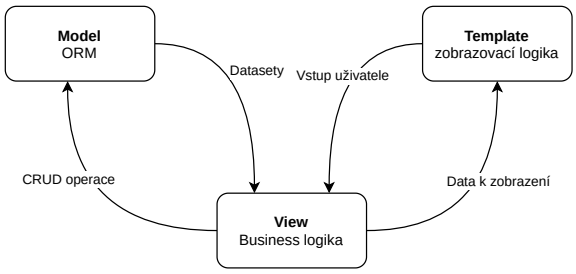
Nasledujúci obsah vychádza z knihy Mastering Django [4], ktorá slúži ako východiskový zdroj tejto sekcie.

Návrhový vzor Model-View-Template (MVT) používaný výhradne Djangom do istej miery vychádza z Model-View-Controller (MVC), ktorý funguje nasledujúcim spôsobom:

- Model slúži na reprezentáciu dát. Nie sú to dáta samotné, ale rozhranie, ktoré poskytuje prístup k nim. Model sa nemení pri zmene databázy.
- View je prezentačná vrstva, ktorá predstavuje používateľské rozhranie. Zároveň zbiera dáta od používateľa.
- Controller organizuje tok informácií medzi Modelom a View. Rozhoduje o zmene oboch ďalších modelov a implementuje business logiku.

Popísané rozdelenie nie je striktné. Časť zodpovednosti môže napríklad preberať iná vrstva. Django do istej miery dodržiava vzor MVC. Najmarkantnejším rozdielom však je, že Controller je implementovaný priamo vo vnútri frameworku. Programovaná logika sa tak presúva predovšetkým na View a v menšej miere aj na Model a Template. V tomto rozložení má každá vrstva nasledujúcu zodpovednosť:

- Model opäť poskytuje prístup k dátam. Je implementovaný ako Objektovo-relačné mapovanie (angl. Object–relational mapping, ORM)
- Template preberá zodpovednosť za reprezentáciu a tvorbu užívateľského rozhrania na strane klienta.
- View preberá business logiku. Definuje spôsob prístupu k dátam a obsluhu zvoleného Templatu. Funguje ako most medzi medzi Modelom a View.



Obrázek 1: Komunikácia mezi vrstvami návrhového vzoru MVT

Komunikácia medzi jednotlivými vrstvami je zobrazená na obrázku 1.

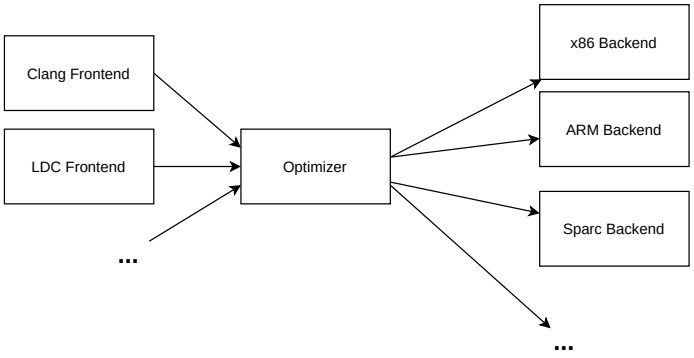
3 Prehľad použitých vzorov

V tejto kapitole sa nachádza všeobecný opis návrhových vzorov, ktoré boli nájdené. Konkrétne nájdený výskyt a jeho popis sa nachádza v nasledujúcej sekcii 4.

Prekladač

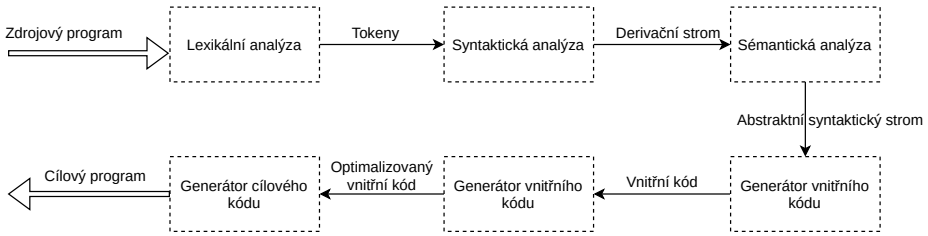
Pokiaľ nie je uvedené inak, táto sekcia čerpá z knihy Formal Languages and Computation: Models and Their Applications [6].

Prekladač (angl. compiler) je tradične program, ktorého vstupom je kód a výstupom taktiež kód, ale v inom jazyku, ktorý býva nižšej úrovne. Môže ísť buď o monolit, alebo o architektúru rozčlenenú na frontend, middleware a backend. Frontend spracováva vstupný kód na vnútornú reprezentáciu, Midddleware má na starosti optimalizáciu a Backend generuje nízkoúrovňový kód pre konkrétnu architektúru. Toto rozdelenie je vidieť na obrázku 2. [5]



Obrázek 2: Časti nemonolitického prekladača

Podrobnejšie sa prekladač delí na *lexikálny analyzátor* (lexer), *syntaktický analyzátor* (parser), *sémantický analyzátor*, *generátor vnútorného kódu*, *optimalizátor* a *generátor cieľového kódu*. Prepojenie medzi týmito časťami zobrazuje obrázok 3.



Obrázek 3: Príklad vnútornej štruktúry prekladača

Lexer zo znakov zdrojového textu zostavuje *lexémy*, čo sú jednotky oddeliteľné od zvyšku dokumentu. Tie sú následne reprezentované *tokenmi*, ktoré môžu mať atribúty (typ, hodnota, atď.). Implementovaný je typicky konečným automatom.

Parser kontroluje reťazec tokenov, či reprezentuje správne syntakticky napísaný program. Vytvára pri tom derivačný strom za použitia gramatických pravidiel. Pokiaľ sa ho podarí vytvoriť, program je v poriadku. Pokiaľ nie, nastala syntaktická chyba. Na implementáciu sa používa zásobníkový automat.

Sémantická analýza kontroluje sémantické prvky programu. Tými sa rozumie napríklad kontrola správneho použitia dátových typov, či je premenná deklarovaná, delenie nulou a iné nepovolené akcie. Často sa zlučuje dohromady so syntaktickou analýzou a vzniká tzv. *syntaxou riadený preklad*.

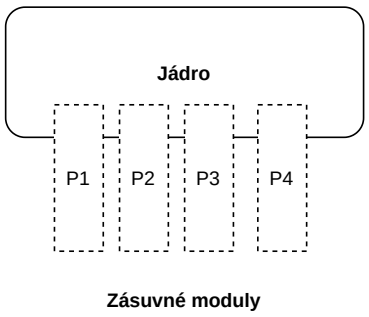
V rámci generátora vnútorného kódu sa vytvára vnútorná reprezentácia, najčastejšie vo forme trojadresného kódu, s ktorým sa potom ďalej pracuje. Je vstupom optimalizátora a po ňom generátora cieľového kódu, ktorý vnútorný kód prevádza na výsledný program.

Zásuvný modul

Tento text je vycháza z článku Plug-in Architecture [3].

Zásuvný modul (angl. plugin) funguje na jednoduchom princípe. Umožňuje pridanie novej funkcionality do už existujúcich komponentov bez toho, aby tieto komponenty museli vedieť o implementačných detailoch zásuvného modulu.

Všeobecne sa táto architektúra skladá z dvoch častí, a to jadra systému a samotného zásuvného modulu. Dodatočná funkcionalita sa pridáva k už existujúcemu a samostatne fungujúcemu jadru, vďaka čomu je umožnená rozšíriteľnosť, flexibilita a izolácia logiky novej funkcionality od základnej. Okrem zaistenia základného fungovania, jadro poskytuje spoločný kód pre zásuvné moduly, aby nedochádzalo k duplikácii kódu.



Obrázek 4: Základná štruktúra návrhového vzoru zásuvný modul

Zásuvné moduly sú samostatne stojace komponenty, ktoré by nemali závisieť na sebe navzájom. V niektorých prípadoch však medzi nimi prebieha komunikácia, napriek tomu je dôležité interakciu medzi nimi čo najviac minimalizovať.

Aby mohol zásuvný modul správne fungovať, musí o ňom jadro vedieť a musí existovať cesta, ako ho zaintegrovať. Jadro preto definuje spôsob, akým sa má zásuvný modul pripojiť. Pri pripojení sa odovzdávajú informácie akými sú meno modulu, komunikačný protokol, obslužné funkcie, formát dát. Súhrn odovzdávaných informácii môže byť veľmi rôznorodý, preto by rozhranie malo byť jasne definované.

Presný spôsob, akým sa modul pripája k jadru záleží na konkrétnej architektúre a implementácii.

Nezávislosť modulov dáva možnosť ich pružne pridávať, či odoberať. Podľa použitej implementácie je možné modul samostatne vyvíjať, testovať či rozširovať. Na druhej strane ľahká zmena implementácie jadra môže zásadným spôsobom ovplyvniť moduly. Určite je teda na mieste najprv veľmi dôkladne premyslieť návrh jadra samotného.

4 Django Template System

Ako webový framework potrebuje Django nejaký spôsob, ako dynamicky generovať HTML. Najbežnejšie riešenie je *šablóna* (angl. template), ktorá obsahuje statické časti HTML spolu so špeciálnou syntaxou, ktorá špecifikuje, akým spôsobom sa majú vkladať dynamické časti. Django podporuje použitie niekoľkých template enginov, ale väčšinou sa využíva natívny systém nazvaný Django Template Language (DTL) [2].

4.1 Prekladač

Keďže sa jedná o jazyk, je nutný aj preklad tohto jazyka do výstupného kódu. Súčasťou DTL je teda aj prekladač.

Tradičné prekladače sú často riadené syntaxou. Keďže má ale DTL veľmi špecifickú štruktúru, nedáva táto možnosť veľmi zmysel. Riadenie prebieha v rámci triedy `Template`, ktorá dostane na vstupe reťazec predstavujúci vstupný dokument a následne volá jednotlivé časti prekladu, pričom každá ďalšia časť môže začať až po dokončení predchádzajúcej.

Popisovaná funkcionalita je ďalej demonštrovaná na príklade vstupného kódu zobrazenom na obrázku 5. Každá z nasledujúcich sekcií sa zameriava na popis jednej časti prekladača, na záver je uvedené porovnanie implementácie s návrhovým vzorom prekladača.

```
<h1>Dukovany homepage</h1>
{% if reactor_temperature > 200 %}
    Warning: Reactor temperature is {{ reactor_temperature }}
{% endif %}
```

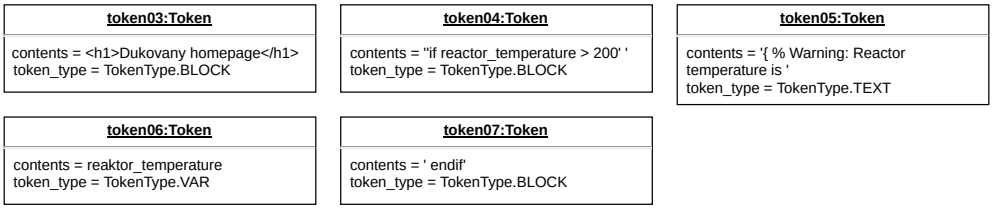
Obrázek 5: Fragment vstupného kódu v DTL

Lexikálna analýza

Inštancia triedy `Lexer` prostredníctvom svojej funkcie `tokenize()` vykonáva lexikálnu analýzu. Vracia list tokenov, ktoré môžu byť štyroch druhov, a tými sú text (`Token.TEXT`), premenná (`Token.VAR`), komentár (`TokenType.BLOCK`) alebo blok (`Token.Blok`). Token je v DTL inštancia triedy `Token` s minimálne štvormi atribútmi. Okrem typu tokenu obsahuje aj reťazec s pôvodným obsahom a dva voliteľné argumenty k ladeniu.

Funkcionalita `tokenize()` by sa síce dala previesť na konečný automat, ale nie je týmto spôsobom realizovaná. Najskôr je vstup rozdelený pomocou regulárneho výrazu na časti. Cez tieto časti sa následne iteruje a podľa obsahu sa vytvára konkrétny typ tokenu.

Pokiaľ by sme aplikovali lexikálnu analýzu na fragment kódu z obrázku 5, dostaneme tokeny zobrazené v obrázku 6. Pôvodný kód sa rozdelil na päť častí. Úvodný nadpis je prvým tokenom. Podmienka sa rozdelila na hlavičku, telo a ukončenie. Telo podmienky sa ešte rozdelilo na úvodný textový token a token obsahujúci premennú.



Obrázek 6: Časť vytvorených tokenov

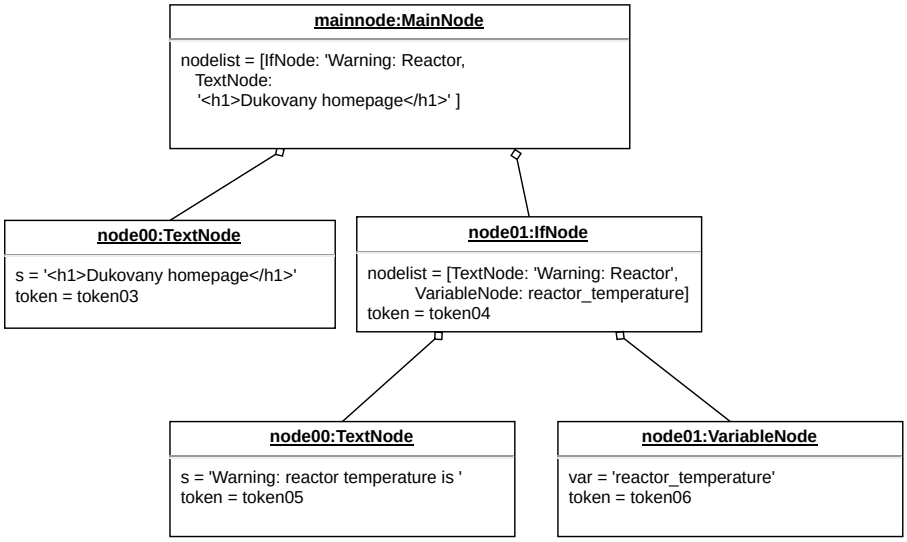
Syntaktická analýza

Výsledný list tokenov je vstupom inštancie triedy `Parser`. Tá prejde cez všetky jednotlivé tokeny a vytvorí z nich `Node` (uzol). Základné typy sú napríklad `TextNode`, `ForNode` alebo `IfNode`.

Token typu `TEXT` a `VAR` znamenajú Node toho istého typu. Pri `BLOCK` type je situácia zložitejšia. Z bloku sa vyberie úvodný príkaz, podľa ktorého sa kontroluje, či sa jedná o niektorý z validných príkazov. Ak áno, je s ním

zviazaná funkcia, ktorá skontroluje syntax a vráti vytvorený konkrétny `Node`. Zoznam príkazov, ktoré parser podporuje je rozšíriteľný. Viac k rozšíriteľnosti parseru je popísané v podsekcii 4.2.

Výsledná štruktúra uzlov avšak nie je lineárna. Uzly sa rekurzívne vnárajú do seba. V našom príklade (obrázok 7) v sebe napríklad uzol `IfNode` reprezentujúci podmienku obsahuje všetko, čo logicky spadá do jeho tela, teda text a premennú. Tento uzol potom spolu s uzlom reprezentujúcim nadpis stránky spadajú do ďalšieho nadradeného uzla.



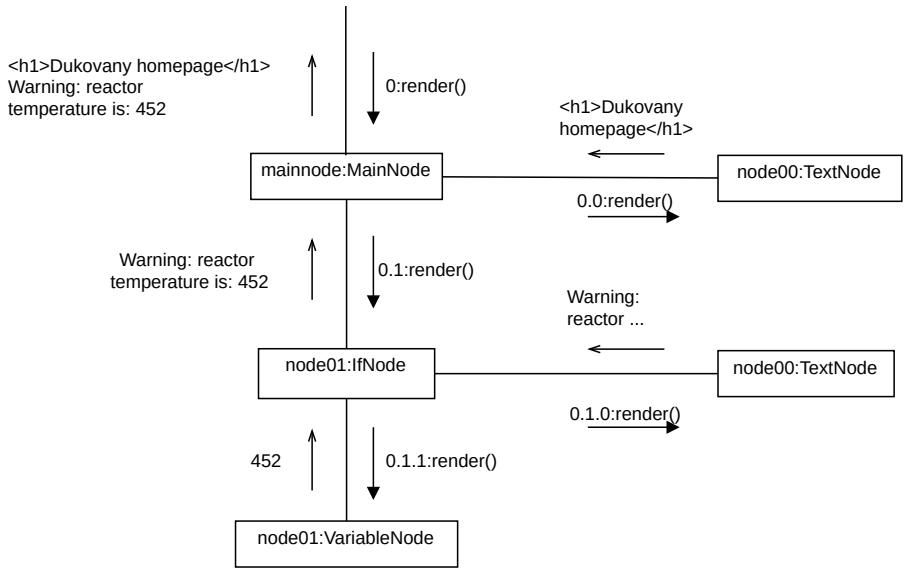
Obrázek 7: Štruktúra vytvorených uzlov

Generovanie výsledného kódu

Každý `Node` má k dispozícii metódu `render()`, ktorá interpretuje obsah uzla. V prípade najjednoduchšieho `TextNode` je vrátený jeho text. `VariableNode` vracia po vykonaní tejto metódy hodnotu premennej, ak sa nachádza v kontexte, a `IfNode` vykoná porovnávaciu logiku a vráti výsledok. Metóda `render()` teda môže podľa typu uzla pokrývať časti prekladača sémantická analýza a generátor výstupného kódu.

Keďže sú uzly hierarchicky prepojené, tak aj renderovanie je volané postupne iteratívne. Volanie na našom príklade je zachytené komunikačným diagramom na obrázku 8. Generovanie hlavného uzla spočíva vo vlastnej činnosti a vo volaní metódy `render()` na nižšie úrovne. Od prvého textového uzla dostane ako odpoveď reťazec obsahujúci nadpis stránky. Pri druhom volaní na `IfNode` sa najprv vyhodnotí podmienka. Pokiaľ by vyšla nepravda, vrátený bude prázdny reťazec. V opačnom prípade sa budú volať uzly prislúchajúce k telu podmienky. Prvý z nich opäť rovno vráti text. Druhý uzol je

však typu `VariableNode`. Je nutné teda zistiť, či premenná v kontexte generovanej šablóny vôbec existuje, načo sa prípadne vráti jej hodnota v textovej forme. Kontrola premennej predstavuje zjednodušený príklad sémantickej analýzy.



Obrázek 8: Komunikačný diagram volánia metody `render()`

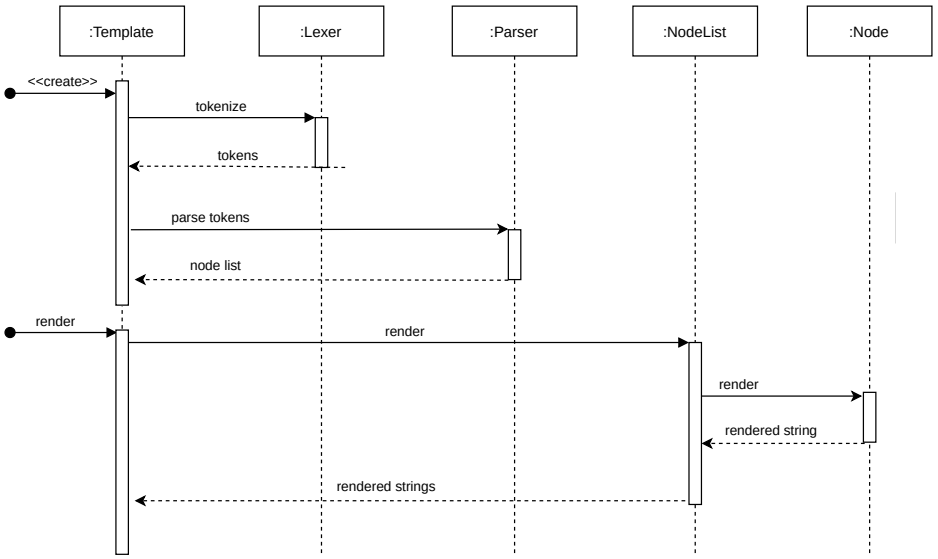
Zhodnotenie

Opísaný prekladač je typu monolit. Vo všeobecnosti je táto implementácia návrhového vzoru prekladač veľmi voľná a v žiadnom prípade nejde o klasickú verziu. Explicitne oddelené časti sú lexikálna analýza, syntaktická analýza a generovanie výstupného kódu. V rámci generovania výstupného kódu je v niektorých prípadoch implementovaná aj sémantická analýza. Súhrnný prehľad riadenia prekladu je zobrazený na obrázku 9.

4.2 Zásuvný modul

Rozšíriteľnosť DTL je zaistená pomocou možnosti pridania knižníc, ktoré predstavujú zásuvný modul. V predvolenej konfigurácii sú používané knižnice definované v rámci priečinkov `template` a `teamplatetags`.

Po pridaní knižnice sa lexikálna analýza nemení. Pridaná funkcionalita môže byť iba v rámci blokov, výsledný token teda bude typu `BLOCK`. Rozširujú sa však možnosti parseru, ktorý zastáva funkciu jadra a ktorému sú knižnice (inštancie triedy `Library`) predkladané.

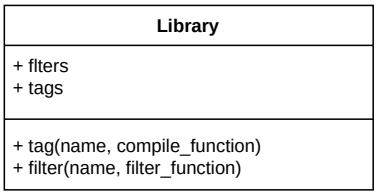


Obrázek 9: Riadenie prekladu

V rámci tejto podsekcie je najskôr popísaná funkcionalita knižnice a následne akým spôsobom sú knižnice spracovávané v parsere. Následne je v krátkosti zhrnuté pripojenie knižníc. Posledná podsekcia porovnáva popisovanú implementáciu s návrhovým vzorom.

Knižnica

Knižnica je implementovaná ako trieda `Library` (obrázok 10) s metódami, ktoré slúžia pre registráciu. Každá z nich je implementovaná ako dekorátor. Rozširovať je možné buď tzv. *tagy* alebo *filtre*.



Obrázek 10: Zjednodušený diagram triedy `Library`

Tag je úvodné označenie bloku, ktoré určuje jeho význam. Príkladom môže byť tag `comment` (obrázok 11), `if`, `for` alebo `lorem`. Tagy sú v knižnici uchovávané v slovníku `tags`, v kterom kľúč predstavuje názov tagu a hodnotu predstavuje obslužná funkcia tagu.

```
{% comment %}
    all here skipped
{% endcomment %}
```

Obrázek 11: Príklad tagu komentára v DTL

Príklad registrácie tagu `comment` je zobrazený vo výpise 1. Dekorátorom sa označí obslužná funkcia. Pokiaľ sa jej meno zhoduje s názvom tagu, automaticky sa určí z neho. Pokiaľ sa ale jedná o problematické mená, ktoré by mohli korelovať s jazykom Python, určí sa explicitne meno ako argument dekorátora.

Uvnitř obslužné funkce je definována část syntaktické analýzy, jejímž výsledkem, pokud nenastane chyba, je instance konkrétního druhu uzlu. Argumenty obslužné funkce jsou zpracovávaný token a instance aktuálního parseru, se kterou může funkce dále interagovat.

```
register = Library()

@register.tag
def comment(parser, token):
    """
    Ignore everything between ``{% comment %}`` and ``{% endcomment %}``.
    """
    parser.skip_past('endcomment')
    return CommentNode()
```

Výpis 1: Registrace tagu komentář

Kromě tagů se dá dají registrovat i *filtry*. Filter sám o sobě je tag, který ale může být různých druhů s různou funkcionalitou. Může například odstranit HTML znaky, převést text na malé písmo, nebo spočítat délku textu. Většinou se jedná o filtry, které pracují s řetězci. V rámci třídy `Library` se opět uchovávají stejným způsobem jako tagy, jedná se tedy o slovník, kde klíčem je název filtru a hodnotou obslužná funkce.

```
{% filter lower %}
    SoMe MiXed TEXT
{% endfilter %}
```

Obrázek 12: Příklad filtru převádějící text na malé písmo

Příklad registrace filtru do knihovny je naznačen ve výpise 2. Jelikož se jedná o textový filter, musí být nad obslužnou funkcí krom registrujícího dekorátoru uveden i dekorátor označující, že vstupem má být text. Výstu-

pem však už může být jakýkoliv datový typ. Samotné tělo obsahuje filtrační funkci, která se má použít.

```
register = Library()

@register.filter(is_safe=True)
@stringfilter
def lower(value):
    """Convert a string into all lowercase."""
    return value.lower()
```

Výpis 2: Registrace filteru lower

Parser

Základní funkcionalita parseru je popsána v sekci 4.1. Parser sám o sobě bez přidání knihoven dokáže vytvářet pouze `TextNode` a `VarNode`. Další typy uzlů, které jsou vytvářeny z tokenu typu `BLOCK`, mohou být použity teprve až po nahrání knihoven. Pokud existuje tag, který odpovídá použitému příkazu v bloku, pak se provede obslužná funkce náležející k tomuto bloku. Fragment kódu obsahující základní strukturu metody `parse` je ve výpise 1.

```
class parser():
    def parse(self, parse_until=None):
        nodelist = NodeList()
        while self.tokens:
            token = self.next_token()
            if token.token_type.value == 0: # TokenType.TEXT
                self.extend_nodelist(nodelist, TextNode(token.contents), token)
            elif token.token_type.value == 1: # TokenType.VAR
                ...
                self.extend_nodelist(nodelist, var_node, token)
            elif token.token_type.value == 2: # TokenType.BLOCK
                command = token.contents.split()[0]
                try:
                    compile_func = self.tags[command]
                except KeyError:
                    self.invalid_block_tag(token, command, parse_until)

                try:
                    compiled_node = compile_func(self, token)
                except Exception as e:
                    raise self.error(token, e)
                self.extend_nodelist(nodelist, compiled_node, token)
```

Výpis 3: Fragment metody parse

Pokud parser narazí na blok, který je filtrem, provolá se zaregistrovaná obslužná funkce, jako v ostatních případech. Oproti běžnému případu ale filter úzce spolupracuje s parserem a využívá jeho metody, jak je vidět ve výpise 4. Nejdříve je extrahován text obsahující typ, nebo typy filtru. Následně je volána metoda parseru, která zodpovídá za vytvoření instance třídy `FilterExpression`. Při inicializaci se v rámci ní separují použité typy filtrů

a přes přes parser se získávají přiřazené obslužné funkce.

```
@register.tag('filter')
def do_filter(parser, token):
    _, filter_text = token.contents.split(None, 1)
    filter_expr = parser.compile_filter("var|%s" % (filter_text))
    nodelist = parser.parse(('endfilter',))
    parser.delete_first_token()
    return FilterNode(filter_expr, nodelist)
```

Výpis 4: Obslužná funkce pro tag filter

V tělo obslužné funkce pro filter se po navrácení zmiňované instance zpracují další tokeny a výsledkem je `FilterNode` obsahující instanci `FilterExpression`, která je použita pro filtrování při volání metody `render()`.

Připojení knihovny

Knihovny, které jsou v DTL standardně používány, se definují v `Enginu`, který je předá vytvořeným instancím třídy `Template`, které je následně předají parseru. Uživatelé Djanga však nemusí programově zasahovat do nástroje samotného. Stačí vytvořit nový soubor s registrací (výpis 5), který se nahraje přímo v template pomocí tagu *load* (obrázek 13).

```
# myapp/templatetags/customtags.py
from django import template
register = template.Library()

@register.filter
def first_letters(text):
    return text[0]
```

Výpis 5: Registrace vlastního filtru

```
{% load customtags %}
```

Obrázek 13: Připojení vlastní knihovny

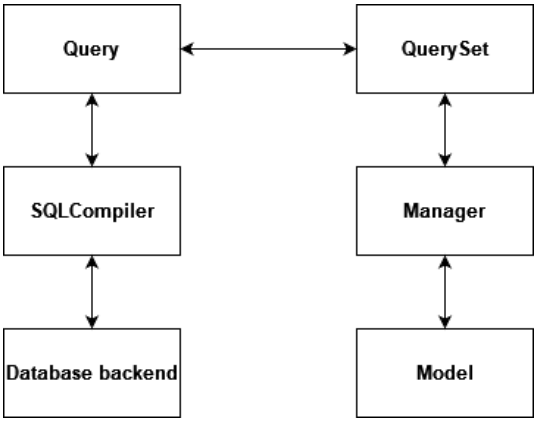
Zhodnocení

Implementace parseru (jádro) a knihoven (zásuvný modul) plně odpovídá návrhovému vzoru popsaném v podsekci 3. Praser je část, která je schopna s omezenou mírou fungovat samostatně. Užitečnější funkcionalitu nabízí ale až po přidání knihoven, které je možno nahrát dvěma způsoby. Moduly mezi sebou nekomunikují ani jinak neinteragují. Často ale využívají kód implementovaný v parseru, díky čemuž mohou zasahovat i do řízení chodu jádra.

5 Objektovo relačné mapovanie

Django má v sebe zabudované vlastné Objektovo relačné mapovanie (angl. Object–relational mapping, ORM), čo umožňuje používateľom jednoduchý prístup k databáze bez akejkoľvek zmeny kódu alebo písania SQL príkazov. Namiesto toho sa využíva objektovo-orientovaný Python kód. ORM dokáže automaticky konvertovať Python kód na SQL, čiže na jazyk, ktorému databázy rozumejú. To znamená, že vo svojom jadre je ORM abstrakcia pre interagovanie s relačnými databázami, slúžiaca na výmenu dát z Django aplikácie do databázy alebo naopak. V tejto sekcii sa podrobne pozrieme na architektúru ORM a odhalíme, ako vlastne Django spracováva dopyty do databázy. Zdrojom týchto informácii je vo väčšine prípadov oficiálna dokumentácia Djanga [1].

Django ORM pozostáva z niekoľkých vrstiev. Sú to Model, Manažér, QuerySet, Query, SQL Prekladač a Databázový backend (obrázok 14).



Obrázek 14: Hlavné zložky Django ORM

Databázový backend

Databázový backend sa v zdrojovom kóde Djanga nachádza na ceste `django/db/backends/`. Táto vrstva ORM predstavuje najnižší level interakcie s databázou. Je to hranica medzi Djangom a databázovými modulmi ovládačov (angl. driverov), akými sú napr. cx_Oracle. Database backend je tvorený niekoľkými komponentmi, ktoré sú reprezentované triedami.

Jednou z najkomplexnejších z nich je **DatabaseWrapper**. Ten obsahuje informácie o databáze, ktorá sa používa, teda aj o jazyku, ktorý táto databáza využíva. (Nejedná sa presne o jazyk, databázy využívajú SQL. Lepšie pomenovanie je anglické dialect.) Na základe týchto informácii je potom

možné pretvoriť dopyt na správny SQL tvar, ktorému bude databáza rozumieť. DatabaseWrapper tiež vie kontrolovať správanie transakcií.

Ďalšou dôležitou triedou je **DatabaseOperations**. Ako už názov napovedá, v tejto triede vieme zistiť, ako správne vykonávať operácie s databázou. Je tu opísané, ako správne robiť flushing, teda synchronizovať dočasný stav dát aplikácie s trvalým stavom dát, taktiež je tu metóda na sequence resets. Podstatnou úlohou DatabaseOperations triedy je spracovanie toho, ako naša databáza bude pracovať s typmi premenných. Špecificky ako robiť type cast. Napríklad ak máme premennú, ktorá obsahuje nejaký dátum a my chceme z neho extrahovať rok, táto trieda nám povie, ako to urobiť.

DatabaseFeatures obsahuje informácie o tom, aké možnosti používaná databáza podporuje. Django priamo podporuje niekoľko databáz, ale je možné pripojiť aj inú databázu. Rozličné databázy fungujú rôzne, a preto sa v tejto triede nachádza vyše 100 premenných, ktorých hodnota sa nastaví buď na `True` alebo `False` na základe používanej databázy. Príkladom takejto premennej je `interprets_empty_strings_as_nulls`, ktorá vyjadruje, či rozlišujeme prázdny string(’’) a `None`, alebo `supports_unspecified_pk`, čo určuje, či môže byť objekt uložený bez primárneho kľúča.

DatabaseCreation slúži najmä na vytváranie indexov v databáze. Na vytvorenie samotnej databázy sa avšak v súčastnosti už používajú iné frameworky. Táto trieda sa podieľa aj na serializácii dát v databáze.

DatabaseIntrospection pracuje hlbšie s databázou, pozná interný typ stĺpca (angl. internal column type) a vie ho premapovať na typ charakteristický pre Django. Interným typom stĺpca sa myslí číslená hodnota, ktorá prislúcha typu stĺpca v databáze. DatabaseIntrospection dokáže tiež vytvoriť model databázy.

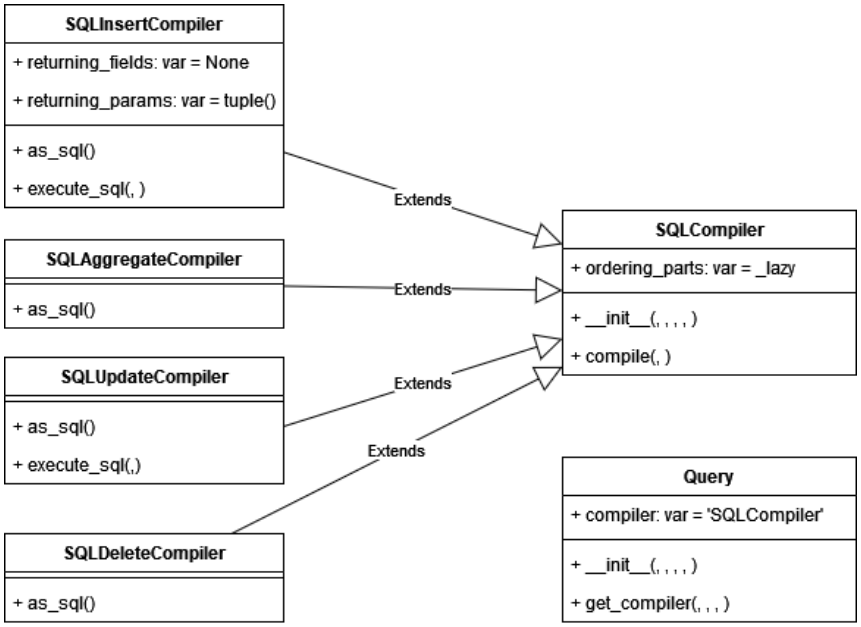
Ak uskutočňujeme migrácie databázy, využíva sa **DatabaseSchemaEditor**. Rôzne databázy robia migrácie odlišne a tieto informácie trieda DatabaseSchemaEditor pozná.

DatabaseClient je používaný dbshell-om, umožňuje používateľovi otvoriť konzolu, pomocou ktorej môže spúšťať Python príkazy v databáze, alebo aj písať SQL príkazy.

SQL Prekladač

SQL Prekladač zahŕňa niekoľko tried, ktoré nájdeme v `django/db/models/sql/compiler.py`. Niektoré z nich sú uvedené na obrázku 15. Informácie zobrazené na obrázku sú limitované kvôli lepšej čitateľnosti.

Tento prekladač dokáže zmeniť Django query na SQL dopyt vhodný pre našu databázu. SQL Prekladač slúži ako most medzi Djangom a databázou. Funkcia `Query.get_compiler()` vráti inštanciu SQL Prekladača pre dané Query. Následne sa zavolá metóda `compiler()`, ktorá sa nachádza v už spomínanej triede DatabaseOperations.



Obrázek 15: Vzťahy medzi triedami prekladača

Query

Ďalšou triedou, ktorá veľmi významne podporuje fungovanie ORM, je Query. Tú nájdeme v `django/db/models/sql/query.py`. Nachádzajú sa tam dátové štruktúry a metódy reprezentujúce databázové query. Tieto query avšak stále nie sú vo forme SQL, ale namapované na vysokoúrovňovú Pythonovú reprezentáciu, ktorá sa už ale značne podobá na výsledné SQL query. Je možné využívať raw query, ktoré nám umožňuje vyhnúť sa používaniu ORM, no štandardnejší postup je využiť klasické query s ORM operáciami. Na základe výberu sa to realizuje v príslušnej triede. Metóda `as_sql()` zlúči všetky atribúty reprezentujúce časti query a vytvorí z nich SQL.

Zjedodušene, Query je ako stromová dátová štruktúra, ktorá ak sa má zmeniť na validné SQL, musí postupne prejsť všetkými uzlami, kde zavolá metódu `as_sql()` na získanie jednotlivých výstupov.

SQL Prekladač a Query majú veľmi rozsialhy kód. To je zapríčinené primárne tým, že vysokoúrovňové query metódy sú ľubovoľne reťaziteľné. To

znamená, že ak uskutočníme ľubovoľný dopyt a dostaneme nový QuerySet (QuerySet si môžeme predstaviť ako list výsledkov), tak na tomto QuerySete môžeme opäť vykonávať ľubovoľný dopyt a akciu opakovať. V jednoduchosti, musíme vedieť zobrať čokoľvek, čo sa dá v Djangu spraviť, spojiť to s niečím iným čo nám Django dovoľuje a stále dostať z databázy relevantný výsledok. Väčšina tejto komplexnosti pochádza z procesu spájania QuerySetov. Každá validná kombinácia QuerySet metód tu musí byť podporovaná a takisto aj podpora pre iné rozšírenia vyhľadávania, ktoré môžu byť doprogramované užívateľom alebo ich môže ponúkať tretia strana.

Query výrazy a funkcie

Query výrazy dávajú používateľovi širší prístup k SQL a teda celkovo lepšiu možnosť vyjadriť dopyty jednoduchšie. Dodávajú Djangu komplexnejšiu logiku a referencie. Príkladom sú triedy nachádzajúce sa v `django/db/models/expressions.py`. Trieda `When` dovoľuje použiť `WHEN ... THEN` v SQL, čo v mnohých prípadoch umožňuje využiť len jeden príkaz namiesto skupiny niekoľkých. Trieda `Case` implmentuje možnosť využívať `CASE` tvrdenia v SQL, trieda `F` umožňuje referencovať stĺpce alebo anotácie a trieda `Func` implmentuje volania na SQL funkcie. To zapríčiňuje, že query môže zavolať niektoré SQL funkcie. Django poskytuje vstavanú podporu napríklad pre funkcie `UPPER` alebo `SUBSTRING`.

QuerySet

V rovnakom priečinku ako trieda Query sa nachádza aj trieda QuerySet. Táto trieda reprezentuje lenivé vyhľadávanie množiny objektov v databáze (angl. lazy database lookup). QuerySet okrem iného kvázi obalí inštanciu Query do API, aby bolo možné s query manipulovať alebo ho inak meniť. Lenivosť v tomto prípade značí, že QuerySet niekedy nič nerobí, ak zavoláme jeho metódy.

Ako príklad tejto lenivosti použijeme ukážku z [1]. Môžeme zavolať ľubovoľné množstvo filtrov alebo iných funkcií, no Django v skutočnosti nespustí query, kým nie je QuerySet použitý.

```
>>> q = Entry.objects.filter(headline__startswith="What")
>>> q = q.filter(pub_date__lte=datetime.date.today())
>>> q = q.exclude(body_text__icontains="food")
>>> print(q)
```

V uvedenom príklade sa môže zdať, že sa jedná o tri zásahy do databázy. No v skutočnosti sa tak stane iba raz, a to na poslednom riadku, keď má byť QuerySet použitý. Vo všeobecnosti sa výsledky QuerySetu nezískajú z databázy, pokiaľ ich používateľ priamo nevyžaduje. Ďalšou možnosťou, ako

okamžite získať výsledok query je zavolať metódu, ktorá vyžaduje vykonanie query na overenie existencie výsledkov, zistenie počtu výsledkov a pod., vrátane špeciálnych Python metód.

QuerySet je kvázi kontajner. Každá inštancia tejto triedy má internú vyrovnávaciu pamäť, ktorá je na začiatku prázdna. Ak prvýkrát vykonáme query, vyrovnávacia pamäť sa zaplní. Ak iterujeme nad rovnakým QuerySetom znova, query sa nevykoná, ale namiesto toho dostaneme výsledok z vyrovnávacej pamäte. Tento princíp zabezpečuje vysokú efektivitu.

Volanie QuerySet metód zvyčajne naklonuje už existujúci QuerySet, aplikuje zmeny a vráti novú inštanciu.

```
>>> queryset = SomeModel.objects.all()
>>> queryset[3].attribute
7
>>> queryset[3].attribute = 8
>>> queryset[3].save()
>>> queryset[3].attribute
7
```

Na uvedenom príklade vidíme, že vždy keď sa snažíme s QuerySetom operovať, vytvoríme jeho ďalšiu inštanciu namiesto znovupoužitia rovnakého QuerySetu. To ale neplatí pre iterácie, zisťovanie dĺžky alebo existencie prvku. Tu dochádza k opätovnému využitiu tej istej inštancie QuerySetu na zistenie výsledku.

V triede QuerySet je definovaných niekoľko užitočných metód, ktoré nám umožňujú vykonávať rôzne operácie nad databázou. Sú to napríklad metóda `update()`, ktorá zavolá `SQL UPDATE` nad celým QuerySetom, metóda `delete()` fungujúca podobne ako update, ale zavolá `SQL DELETE` nad zvoleným QuerySetom, metóda `exist()`, ktorá vráti `True` alebo `False` podľa toho, či sa hľadaný prvok nachádza v QuerySete. Ak bol QuerySet už nejakým spôsobom použitý, výsledok sa zistí z jeho vyrovnávacej pamäte. Ak nie, vykoná sa veľmi rýchle query na získanie výsledku. Keďže táto metóda má len jediný účel a tým je zistenie existencie prvku a nie vrátenie vzorky, bola veľmi dobre zoptimalizovaná a je takmer vždy rýchlejšia ako akýkoľvek iný spôsob na zistenie existencie prvku.

Manažér

Manažér je vysokoúrovňové rozhranie, ktoré je priamo prepojené s triedou modelu. Primárnou úlohou manažéra je vytvoriť a vrátiť modelu QuerySet a umožniť tak používateľovi jednoduchšie pristupovať k metódam QuerySetu. Manažér obsahuje metódu `get_queryset()`, ktorá vracia modelu QuerySet. Túto metódu možno prepísať pri vytváraní vlastného Manažéra. Ak pri vytváraní modelu nešpecifikujeme manažéra, Django ho automaticky vytvorí.

Ak ho ale špecifikujeme, Django žiaden nevytvorí. Jeden model môže mať niekoľko manažérov, pričom základným sa stane vždy prvý definovaný.

V nižšie uvedenom príklade z [1] vidíme model, ktorý má dvoch manažérov. Jeden z nich, základný, vracia všetky objekty, zatiaľ čo druhý manažér vracia len knihy, pri ktorých je autorom Roald Dahl.

```
# First, define the Manager subclass.
class DahlBookManager(models.Manager):
    def get_queryset(self):
       return super().get_queryset().filter(author='Roald Dahl')

# Then hook it into the Book model explicitly.
class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.CharField(max_length=50)

    objects = models.Manager() # The default manager.
    dahl_objects = DahlBookManager() # The Dahl-specific manager
```

V tomto vzorovom modeli vidíme, že metóda `Book.objects.all()` má ako návratovú hodnotu všetky knihy v databáze, pričom `Book.dahl_objects.all()` vráti iba tie knihy, ktoré boli napísané Roaldom Dahlom.

Model

Na záver sa dostávame k samotnej reprezentácii dát, modelu. Všeobecne platí, že každá trieda symbolizujúca model spravidla značí jednu tabuľku v databáze a pole v tejto triede značí stĺpec v príslušnej tabuľke.

Model sa nachádza v `django/db/models/base.py` a je napísaný ako Metatrieda. To znamená, že vopred definuje, ako budú niektoré triedy vytvárané. V projekte využívajucom Django sa to odzrkadlí tak, že ak vytvoríme nový model, spolu s ním sa vytvorí aj mnoho iných vecí, ktoré používateľ samotný nevytvoril. Uplatňuje sa tu dedenie. Táto metatrieda priradí modelu manažéra, ak žiadneho nemá.

Priečinok `/django/db/models/fields/` obsahuje Polia modelov (angl. Model fields). Jednou z najdôležitejších vecí, o ktoré sa polia modelov starajú je typ uložených dát. Deje sa to pomocou metód `get_internal_type()` a `db_type()`.

`db_type` použijeme, ak vieme aký typ dát na úrovni databázy chceme, zatiaľ čo `get_internal_type` použijeme, ak chceme dostať typ veľmi podobný skutočnému.

Trieda Field obsahuje metódy, ktoré zabezpečujú premenu typu nejakej hodnoty. Deje sa tak vždy, keď vyberáme dáta z databázy, taktiež aj keď vkladáme dáta do databázy, ale aj vždy, ak vykonáme nejaké query. V kóde

existuje niekoľko podtried hlavnej triedy a tie sú určené na konkrétny typ hodnoty. Každá z týchto tried obsahuje metódy:

`formfield()` - vracia základnú formu poľa pre vybranú triedu

`value_from_object()` - vezme inštanciu modelu a vráti hodnotu pre toto pole v tej inšancii

`deconstruct()` - využíva sa pri migrácii dát

Dedenie modelov

Abstraktný model nevytvára tabuľku v databáze a nemôže ani poskytnúť svoju inštanciu. Je možné napísať podtriedu abstraktnej triedy, ktorá (v prípade, že nie je abstraktná,) bude obsahovať svoje vlastné polia, ale aj polia abstraktného rodiča. Podtriedy avšak dokážu prepísať dáta nadobudnuté od rodiča. Abstraktnú triedu indikujeme pomocou `abstract = True` v Meta deklarácii.

Viactabuľkové dedenie jednoducho znamená, že jeden model sa stane podtriedou druhého. Využíva sa, ak potrebujeme aby oba modely mali svoju tabuľku v databáze. V tomto prípade nie je možné prepísať dáta rodiča.

Proxy modely sa používajú, ak chceme znovu použiť tabuľku a polia rodičovskej triedy, no zmeny môžeme vykonať len na úrovni Pythonu a nie databázy. Môžeme definovať nové metódy alebo aj manažéra. Hlavným využitím tohto typu dedenia je prepísať hlavnú funkcionalitu existujúceho modelu. Proxy triedu vytvoríme pomocou `proxy = True` v Meta deklarácii.

Zhodnotenie

Django ORM je špecifické v tom, ako umožňuje používateľovi pristupovať k databáze pomocou Python kódu namiesto SQL. Je avšak esenciálne pochopiť mechaniku a logiku toho, ako to funguje. Táto sekcia nazrela do hĺbky zdrojového kódu Djanga a vysvetlila jednotlivé procesy transformácie obiektovo-orientovaného kódu na SQL dopyt.

6 Související práce

Odborná literatura zabývající se webovými frameworky se věnuje povětšinou architektuře jen z hlediska základního návrhovému vzoru pro daný framewrk (Model-View-Compiler, Model–View–Viewmodel, atd.). Jedním z příkladů je bakalářská práce na téma Použití Nette frameworku pro vývoj a tvorbu webových aplikací. Dalším častým tématem je porovnávání více frameworků se stejnou základní architekturou jako v odborném článku MVC Architecture: A comparasive study between Ruby on rails and Laravel.

Nejlepším zdrojem k Djangu je jeho oficiální dokumentace. Dále pak jsou sepsány knihy jako například Mastering Django. Obojí se však zaměřuje především jak správně Django používat, ne na vnitřní strukturu a fungování.

7 Závěr

V rámci této práce byl blíže popsán jazyk Django Template Language. Podsekce 4.1 obsahuje podrobný popis postupu překladu tohoto jazyka. Kvůli specifickému použití se struktura překladu značně liší od klasických překladačů. Druhá část (podsekce 4.2) se věnovala rozšiřitelnosti překladače, konkrétně rozšiřitelnosti parseru. K tomuto účelu byl využit návrhový vzor zásuvný modul, který naopak představuje ukázkový příklad, kde je jsou dodrženy všechny koncepty popsané v návrhovém vzoru.

Reference

[1] *Django documentation* [online]. 2021. Dostupné na: `https://docs.djangoproject.com/en/4.0/`.

[2] *The Django template language* [online]. 2021. Dostupné na: `https://docs.djangoproject.com/en/3.2/ref/templates/language/`.

[3] Elgabry, O. *Plug-in Architecture* [online]. 2019. Dostupné na: `https://medium.com/omarelgabrys-blog/plug-in-architecture-dec207291800`.

[4] George, N. *Mastering Django: Core.* [b.m.]: Packt, 2016. ISBN 1787281140; 9781787281141.

[5] Lopes, B. C. a Auler, R. *Getting Started with LLVM Core Libraries.* [b.m.]: Packt Publishing, 2014. Community experience distilled. ISBN 978-1-78216-692-4.

[6] Meduna, A. *Formal Languages and Computation: Models and Their Applications.* [b.m.]: CRC Press, 2014. ISBN 9781466513495.