Politechnika Poznańska

Projekt semestralny

Web scrapper

Meta-wyszukiwarka

Autorzy:
Robert Kowalski
Jarosław Kosowski
Mateusz Szynka

Prowadzący:
mgr inż Przemysław Walkowiak

Spis treści

Część 1

Opis teoretyczny

1 Wprowadzenie

1.1 Opis aplikacji

Zadaniem naszej grupy było napisanie aplikacji zbierającej informacje o strukturze, stronach i treściach znajdujących się w internecie zwanego inaczej robotem internetowym. W zależności od potrzeb może on badać zawartość i kod strony, gromadzić dodatkowe informacje o stronie, monitorować aktualizacje, a także tworzyć mirrory stron. WebCrawler jest botem internetowym systematycznie przeglądającym strony www, zazwyczaj w celu indeksowania stron. Roboty indeksujące mogą weryfikować hiperłącza oraz kod HTML. Mogą być również użyte do web scrapping'u, czyli wydobywania informacji z serwisów internetowych.

Web scrapping jest ściśle związany z indeksowaniem stron, które indeksują informacje na stronie za pomocą bota lub web crawlera i jest uniwersalną techniką przyjętą przez większość wyszukiwarek. W przeciwieństwie do indeksowania, web scrapping bardziej skupia się na przekształceniu niestrukturyzowanych danych na stronie (zazwyczaj w HTML) w dane, które mogą być analizowane w centralnej lokalnej bazie danych lub arkuszu kalkulacyjnym. Zastosowanie tak przetworzonych danych jest już ograniczone tylko zapotrzebowaniem klienta oraz dostępnymi narzędziami na rynku.

Nasze rozwiązanie pozwala na skalowanie wyjścia danych przez zastosowanie odpowiedniego szablonu. Wyświetlanie danych sprowadza się do wytworzenia odpowiednich widoków oraz pobrania odpowiednich danych z bazy danych *crawlera*.

1.2 Podobne aplikacje

Aktualnie na rynku większość rozwiązań tego typu posiada bardzo ograniczoną funkcjonalność i nie udostępnia podstawowych narzędzi wyszukiwania. Rozwiązania komercyjne dzielą się na dwie kategorie:

1. **dla pojedynczego użytkownika** - tego typu rozwiązania są z reguły drogie, wolne oraz rozwinięte w stopniu średniozaawansowanym pod względem użytecznych funkcji

2. **dla serwerowni/datacenter** - rozwiązanie relatywnie tanie, ale wymagają ogromnych zasobów sprzętowych oraz stałych nakładów czasu by utrzymać wyniki na odpowiednim poziomie

1.3 Powód wyboru tematu

Aktualnie użytkownicy internetu produkując terabajty danych dziennie, co powoduje, że wyszukiwanie informacji w takim natłoku nie jest już możliwe w sposób "ręczny". Potrzeba jest dobrych narzędzi do wyszukiwania danych oraz wiązania ich z odpowiednimi zasobami, typami informacji. Web scraping, uczenie maszynowe oraz wyszukiwanie kontekstualne obierają główną rolę w przeciążonym danymi internecie.

2 Cel i zakres pracy

2.1 Podstawowe założenia

- Implementacja web crawlera automatycznie odwiedzającego różne serwisy i zbierającego z nich relewatne informacje do wskazanych słów lub innych danych wejściowych (np. zdjęcia) (opcjonalne)

- jednolity sposób wyświetlania wyników z deduplikacją oraz informacją o źródle danych

- podstawową właściwością projektu ma być jego rozszerzalność o nowe źródła danych np. poprzez implementacje parsera interpretującego strony poprzez wcześniej dostarczone szablony

  - można to również zrealizować poprzez automatyczną analizę strony i identyfikacje jej kluczowych elementów (searchbox, seria wyników, odnośnik do następnej strony)

- uwzględnić potrzebę odwiedzenia kolejnych stron wyników z pojedynczego źródła oraz ich odpowiedniego sortowania

2.2 Zasada działania

Na wejściu aplikacji należy podać link bazowy, od którego chcemy zacząć *crawlowanie* po sieci. Następnie moduł odpowiedzialny za *crawlowanie* wysyła odpowiednie żądania (*HttpRequest*) do serwerów docelowych pobierając *HttpResponse*, który przekazuje do modułu mediatora. Następnie, asynchronicznie, moduł parsera pobiera dane z mediatora, przetwarza je parsując po szablonach oraz wyszukując linki i taki zestaw danych wraca do mediatora w celu dalszej pracy. Następnie *Crawler* pobiera ze słownika linków kolejny link do przejścia. W tym samym czasie na osobnym wątku parser zajmuje się kolejnymi odpowiedziami. Osobny wątek bazy danych, również asynchronicznie, pobiera przeparsowane dane z mediatora i zapisuje je do bazy zgodnie z konwencją *SqlAlchemy* przewidzianą w zaimplementowanych przez nas modelach. Działanie aplikacji kończy się, gdy wszystkie linki, które zostały zwrócone z parsera zostały już przecrawlowane przez moduł *Crawler'a*.

3 Metodyka pracy

Nasze zadania oraz role zostały usystematyzowane w sposób, który pozwalał naszej grupie na jak najszybsze osiągnięcie efektów (działającej aplikacji) przy jednoczesnym zachowaniu standardów programowania obiektowego. Do synchronizacji kodu źródłowego wykorzystaliśmy narzędzie *Git*.

3.1 Podział zadań

Zadania zostały podzielone w taki sposób, aby każdy mógł najlepiej wykorzystać swoje umiejętności i zainteresowania oraz rozwijać się w tych kierunkach. Szczegółowy podział prac przedstawia poniższa tabelka.

Osoba	Zadania
Mateusz Szynka	Implementacja menagera, crawlera oraz podstawowego parsera
Robert Kowalski	Koncepcja i implementacja szablonów
Jarosław Kosowski	Implementacja bazy danych, Data Mediator

Tabela 1.1: Przydział zadań poszczególnym członkom grupy

3.2 Zastosowana metodyka

Podczas prac próbowaliśmy wykorzystać metodologię *Scrum*, co nie jest do końca możliwe w warunkach akademickich. Ze względu na sporą liczbę projektów musieliśmy pominąć większą część elementów wybranej przez nas metodologii.

1. Sprinty - udało nam się zrealizować sześć sprintów
2. Sprint planning - podczas pracy wystąpily nieplanowane problemy przez co nakreślony wcześniej harmonogram mocno odbiega od stanu faktycznego
3. Daily Scrum - zamiast codziennych spotkań ograniczyliśmy naszą komunikację do komunikatorów internetowych dzięki czemu każdy z nas miał dostęp do informacji bez konieczności organizacji spotkań
4. Sprint Review - w ramach prac nie robiliśmy przeglądów każdego sprintu, ponieważ wiele kluczowych aspektów zostało ustalonych już na początku prac. Ostatni sprint, który dotyczył łączenia części funkcjonalnych zabrał nam najwięcej czasu podczas przeglądu.
5. Sprint Retrospective - patrząc na postępy prac stwierdzamy, iż ilość pracy wykonanej podczas sprintów jest nierównomiernie rozłożona. Jest to niestety również uwarunkowane ilością równoległych projektów.

Dla lepszego wykorzystania czasu skorzystaliśmy z narzędzia *Kanboard*, które poza synchronizacją zadań pozwala na wygenerowanie danych statystycznych oraz diagramu Gantta.

3.3 Modelowanie

Faza modelowania nie odbyła się z wykorzystaniem nowoczesnych narzędzi. Wykorzystaliśmy pojemność umysłów oraz przestrzeń kartki do wytworzenia wewnętrznej oraz nieformalnej specyfikacji modułów. Na potrzeby prezentacji projektu powstało kilka diagramów UML ukazujących m. in. sekwencję pracy parsera (Diagram sekwencji parsera., strona 16) czy układ komponentów w systemie (Diagram komponentów aplikacji., strona 8).

4 Proces implementacji

4.1 Platformy programowania

- **Język Python** w wersji 3

- **Baza danych SQLite**

4.2 Biblioteki

- **SQLAlchemy** - pythonowe narzędzie do pracy z bazami danych, a także ORM udostępniający programistom aplikacji całą funkcjonalność SQL bez konieczności pisania surowych zapytań.
- **PyQuery** - biblioteka do języka pythona służąca do wyszukiwania w kodzie HTML danych wykorzystując znaczniki w stylu CSS lub jQuery.
- **minidom** - biblioteka języka python pozwalająca na łatwy sposób przetworzenie dokumentów XML

4.3 Wykorzystane narzędzia

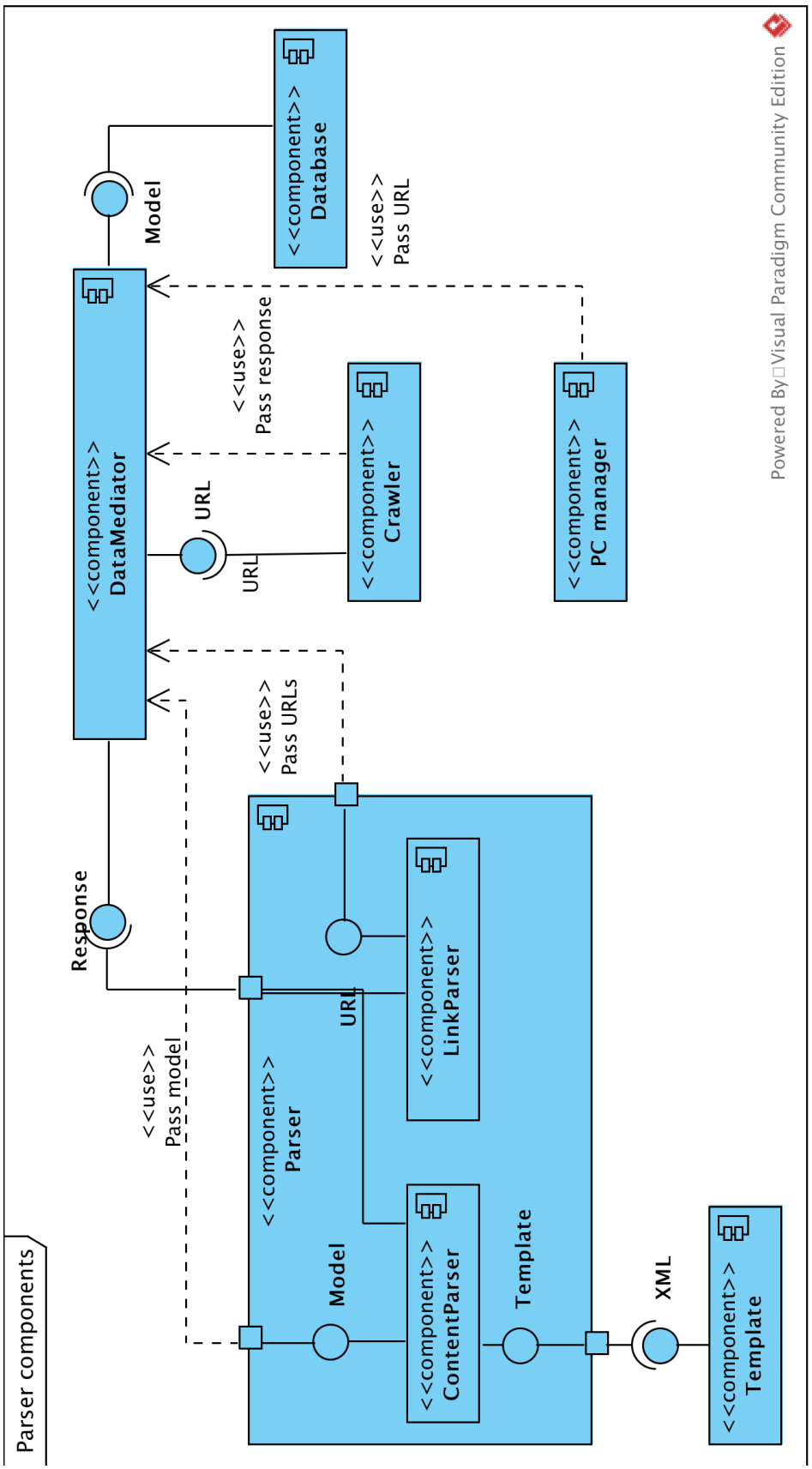
- **PyCharm Professional** - zintegrowane środowisko programistyczne (IDE) dla języka programowania Python firmy JetBrains.
- **Git** - system kontroli wersji umożliwiający dostęp do kodu aplikacji z każdego komputera, na którym w danym momencie pracowaliśmy oraz pozwalający na analizowanie zmian dokonywanych w jej kodzie.
- **GitHub** - do hostowania repozytorium z kodem źródłowym z wykorzystaniem narzędzia *Git*.

5 Modele aplikacji

Podczas prac specyfikacja projektu była bardzo umowna i tworzona w razie potrzeby na kartkach używając symboli umownych. Na potrzeby prezentacji nieniejszego projektu wyspecyfikowaliśmy kilka diagramów wykorzystując modelowanie UML. Nie jest to najpraktyczniejszy sposób (brak zastosowania komercyjnego), ale jest najbardziej zformalizowany z dostępnych rozwiązań.

5.1 Komponenty aplikacji

Komponent zarządzający inicjuje zbiory linków mediatora. Crawler odpytuje linki pobrane z mediatora i przesyła uzyskane odpowiedzi do tego komponentu. Komponent parsera pobiera odpowiedzi, podaje do szablonów URL i odpowiedź jednocześnie parsując linki z odpowiedzi. Parser zwraca modele oraz listę URLi do mediatora. Baza danych pobiera modele z mediatora i zapisuje do bazy SQLite.



Rysunek 1.1: Diagram komponentów aplikacji.

Część 2

Szczegóły implementacyjne

---

1 Szablony

Szablony zostały zaimplementowane w celu umożliwienia rozszerzania aplikacji o nowe źródła danych. W tym celu powstały trzy klasy:

- TemplateResolver
- TemplateManager
- TemplateParse

Same szablony są definiowane jako dokumenty XML. Każdy z szablonów dostarcza adres url jako atrybut głownego elementu typu site. Następnie tworzony jest model elementu, który chcemy uzyskać w wyniku działania aplikacji. By taki element zdefiniować tworzymy element concept z atrybutem name, który zawiera nazwę modelu. Każdy model składa się z elementów object, które zawierają dwa atrybuty:

- name - zawierajacy nazwę pola w modelu
- query - zwierający selector wyszukujący na stronie

1.1 TemplateManager

TemplateManager zarządza całym procesem wyszukiwania szablonów i wstepnym przygotowaniem do parsowania. Metoda parse() na wejściu oczekuje obiektu typu response.

```
<site url="http://spiewnikreligijny.pl/teksty/">
    <concept name="Track">
        <object name="title" query=".entry-title"></object>
        <object name="text" query=".entry-content pre:first-of-type"></object>
        <object name="grips" query=".entry-content pre:nth-of-type(2)"></object>
        <object name="visited" query=".entry-content pre:nth-of-type(3)"></object>
        <object name="added" query=".entry-content pre:nth-of-type(4)"></object>
    </concept>
</site>
```

Rysunek 2.1: Przykładowy szablon

```
class TemplateManager(BaseClass):
   def __init__ (self):
      super().__init__()
      self.xml = 'Parse/Templates/spiewnik.xml'
      self.TResolver = TemplateResolver(self.xml)
      self.TemplateParser = TemplateParser()

   def parse (self, response):
      template = self.TResolver.resolve(response.url)
      if template:
         models = self.TemplateParser.parse(response.html, template)

         return models
      return None
```

1.2 TemplateResolver

Głównym zadaniem TemplateResolvera jest dopasowanie podanego url do odpowiedniego modelu zawartego w pliku xml szablonów. Dla danego TemplateManagera tworzona jest pojedyncza instancja TemplateResolvera, która zawiera przetworzone do postaci słownika (*dict*) wpisy, gdzie kluczem jest URL, a wartoscią lista tagów służących do parsowania.

```
class TemplateResolver(BaseClass):
   def __init__ (self, xml):
      super().__init__()
      self.Xml = xml
      self.UrlDict = { }
      xmldoc = minidom.parse(xml)
```

```
7       for site in xmldoc.getElementsByTagName('site'):
8          listamoja = []
9          for concept in site.getElementsByTagName('concept'):
10            li = [{ obj.attributes['name'].value: obj.attributes['query'].value
                  } for obj in
11                  concept.getElementsByTagName('object')]
12            la = { concept.attributes['name'].value: li }
13            listamoja.extend([la])
14         self.UrlDict[site.attributes['url'].value] = listamoja
15
16    def resolve (self, url):
17       for a in sorted(self.UrlDict, reverse=True):
18          if re.match(r'^' + a + '(\D*\d*\D*\d*)*$', url):
19             return self.UrlDict[a]
```

1.3 TemplateParser

Parser przejmuje odpowiedź z serwera oraz listę szablonów. Następnie wykorzystując *list comprehension* tworzy modele z podanych danych opierając się o listę szablonów. Identyfikator obiektu (krotki) jest generowany za pomocą biblioteki UUID generującej pseudolosowe ciągi znaków.

```
1  class TemplateParser(BaseClass):
2     def parse (self, html, template):
3        pq_parser = PyQuery(html)
4
5        for con in template:
6           ConName = [x for x in con][0]
7           ConId = uuid.uuid4()
8           newConcept = Concepts(ConceptName=ConName, ConceptId=ConId)
9           lista = [CData(ConceptId=ConId, Key=[i for i in v][0],
                 Value=pq_parser([v[i] for i in v][0])) for v in
10                    [con[x] for x in con][0]]
11          lista.append(newConcept)
12       return lista
```

2 Mediator

Do wymiany danych pomiędzy modułami pracującymi w trybie asynchronicznym wykorzystujemy wzorzec projektowy mediatora. W ten sposób zapobiegamy bezpośredniemu wywoływaniu metod z modułów na różnych poziomach logicznych. W pierwszej fazie projektu zastosowaliśmy wzorzec *Chain-of-responsibility*, który zakłada, że dane są przesyłane od modułu do modułu zgodnie z ich priorytetm i kolejnością wykonywania. Ten sposób się sprawdzał dla jednego wątku, ale nie był dobrze zarządzalny oraz skalowalny. Wzorzec mediatora dał nam ogromne możliwości rozszerzalności i skalowalności, ponieważ dane nie są więzione w żaden sposób, ale wymieniane przez współdzielony moduł.

```
1  class Mediator(BaseClass):
2     def __init__ (self, seed: list, input_size: int):
3        super().__init__()
4
5        self._url_qlock = threading.Lock()
6        self._urls = dict()
7
8        self._model_qlock = threading.Lock()
9        self._model_queue = Queue()
10
11       self._response_qlock = threading.Lock()
12       self._response_queue = Queue()
13
14       for item in seed:
15          self._urls[item] = False
16
17    def get_url (self, count=1) -> list
18    def push_urls (self, urls: list) -> None
19    def get_models (self) -> Models
20    def push_models (self, models: Models) -> None
21    def get_responses (self, count=1) -> Responses
22    def push_response (self, response: Response) -> None
23    def push_responses (self, responses: Responses) -> None
```

Dodatkowo do przechowywania danych kategoryzując po typach utworzyliśmy dodatkową klasę *BaseIterable*, która dziedziczona przez klasy *Responses* oraz *Models* czytelnie zaznacza typ przekazywanych danych oraz pozwala na odfiltrowanie nieprawidłowych wyników już na wstępie przetwarzania danych. Ten zabieg pozwolił nam kolejny raz zwiększyć czytelność kodu oraz jego bezawaryjność wprowadzając elementy języków typowanych statycznie do dynamicznie typowanego języka *Python*. Schemat zależności ukazuje diagram

2.4 na stronie 20.

```
class BaseIterable(BaseClass, metaclass=ABCMeta):
   def __init__ (self, objects=None):
      super().__init__()
      if objects is None:
         self._objects = list()
      else:
         assert isinstance(objects, list)
         self._objects = objects

   def append (self, obj) -> None
   def extend (self, obj) -> None
   def first(self) -> item
   def __len__ (self) -> int
   def __iter__ (self) -> yield item
```

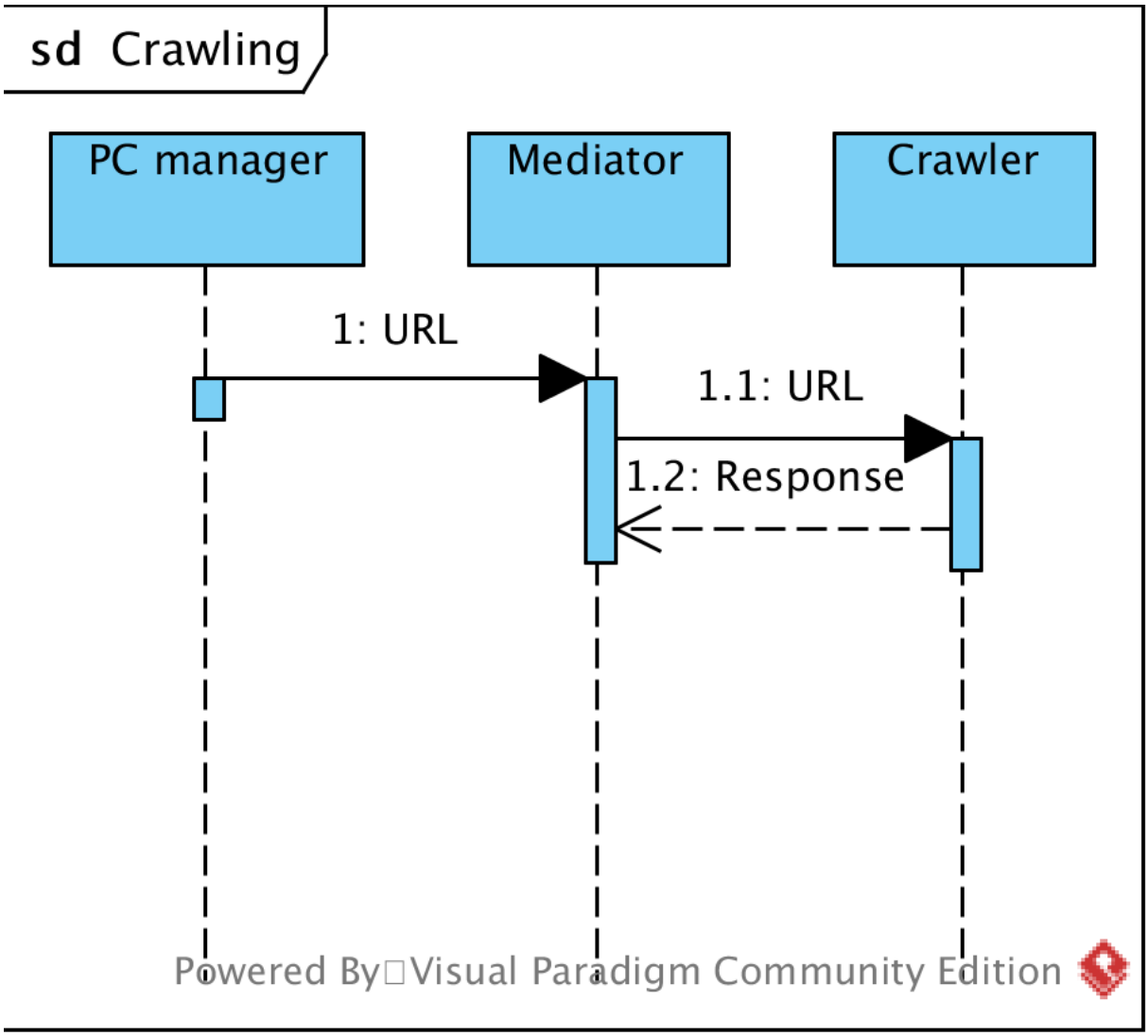
3 Crawler

Zasadą działania modułu *Crawler'a* jest pobieranie danych typu HTTP Response z sieci wykorzystując popularną bibliotekę *urllib*. Podczas testów okazało się, że niektóre serwery nie odpowiadają na zapytania, które w nagłówku nie podają źródła tego zapytania (pole *User-Agent* w nagłówku HTTP Request). Z tego powodu zdecydowaliśmy się na podszywanie się pod znane nam narzędzie *Wget*, które służy właśnie do pobierania danych z sieci po adresie url. Schemat działania modułu przedstawia Diagram sekwencji crawlera. na stronie 14.

Podczas testów okazało się również, że niektóre serwery nie reagują pomyślnie na wiele zapytań na sekundę z tego samego źródła na zmianę odrzucając połączenie lub wyświetlając informację o błędzie dostepu np. do bazy danych. Problem ten nie jest do rozwiązania stosując pojedynczą maszynę zatem ograniczyliśmy się do opóźnienia zapytania w przypadku uzyskania wyjątku z biblioteki *urllib*.

```
class Crawler(BaseClass):
   def _get_response (self, url) -> Response:
      # User-Agent hack
      request = urllib.request.Request(url, headers={ 'User-Agent':
          'Wget/1.9.1' })
      return Response(url, urlopen(request))

   def execute_request (self, url: str) -> Response:
```



Rysunek 2.2: Diagram sekwencji crawlera.

```
8       """
9       Executes a request on given URL
10      :param url: URL to get the response
11      :returns: urllb.request.urlopen response
12      """
```

4 Parser

Moduł parsera dzieli się na kilka klas docelowych:

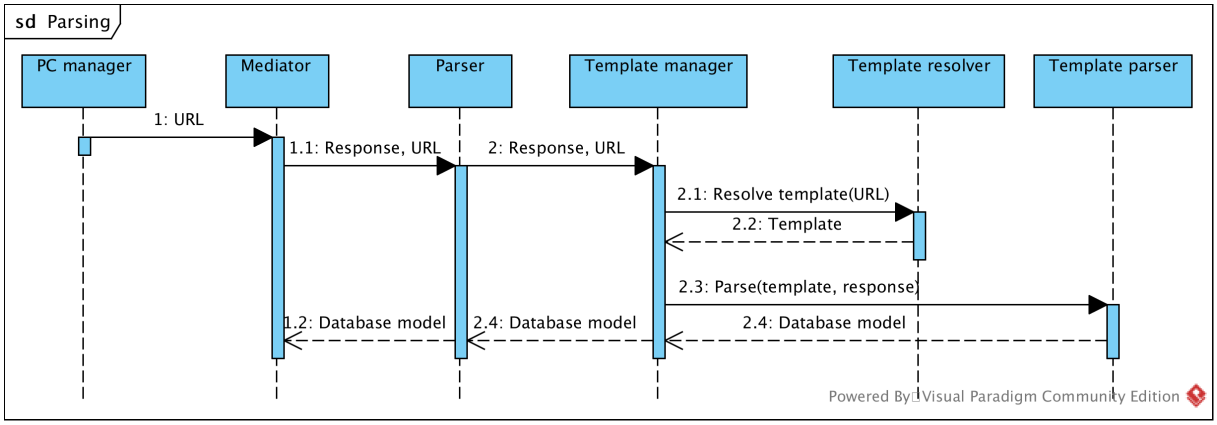
- *BaseParser* - Odpowiednik interfejsu, z którego później dziedziczą kolejne klasy (diagram 2.4, strona 20)
- *ContentParser* - odpowiedzialny za parsowanie treści strony z wykorzystaniem szablonów
- *LinkParser* - służy do parsowania linków z zawartości strony

Do parsowania po tagach oraz z wykorzystaniem wyszukiwania po selektorach w stylu selektorów *CSS* użyliśmy biblioteki PyQuery, która bazuje funkcjonalnością na popularnej bibliotece dla języka *JavaScript* zwanej jQuery. Schemat parsowania przedstawia Diagram sekwencji parsera. na stronie 16.

```
1  class Parser(BaseParser):
2     def __init__ (self):
3        self._content_p = ContentParser()
4        self._link_p = LinkParser()
5        super().__init__()
6
7     def parse (self, response: Response):
8        """
9        Parses response from urlopen
10       :param response: urrlib.request.urlopen product
11       :returns: Models, list(links)
12       """
```

4.1 ContentParser

Moduł *ContentParser* parsuje dane ze strony z wykorzystaniem szablonów do formy, która zostanie później zapisana w modelach do bazy danych. Więcej o szablonach w sekcji Szablony, strona 9.



Rysunek 2.3: Diagram sekwencji parsera.

```
class ContentParser(BaseParser):
   def __init__ (self):
      super().__init__()
      self.template_manager = TemplateManager()

   def parse (self, response: Response):
      return self.template_manager.parse(response)
```

4.2 LinkParser

Moduł *LinkParser* zajmuje się parsowaniem linków z kodu HTML do postaci listy, która następnie zostanie przekazana do modułu Crawler (strona 13).

```
class LinkParser(BaseParser):
   def parse (self, response: Response):
      self._response = response
      self._parse_links()
      self._fix_links()
      return self._links

   def _parse_links (self):
      pq_parser = PyQuery(self._response.html)
      self._links = [x.attrib["href"] for x in pq_parser("a[href]")]

   def _fix_links (self):
      self._get_host_name()
      self._prepend_links_with_hostname()
```

```
15      self._links = list(set(self._links)) # Returns unique values
16
17    def _get_host_name (self)
18    def _prepend_links_with_hostname (self)
```

5 Database

Do obsługi bazy danych w sposób wykorzystujący programowanie obiektowe z poziomu *Python'a* wykorzystujemy:

- Baza SQLite - Prosta baza danych oparta o plik bazy zapisany w lokalnym katalogu.
- SQL Alchemy - prosty ORM(Object-relational mapping) do bindowania danych w formie modeli (obiektów dziedziczących z bazowej klasy). Systemy ORM pozwalają na wygodną realizację zadań wymiany i zapisu danych do bazy typu SQL bez konieczności pisania czystego kodu SQL, co ogranicza liczbę języków, których znajomość jest konieczna.

Wzory modeli przedstawia poniższy kod.

```
1  class Concepts(Base):
2    '''
3    Database SqlAlchemy model for simple data to SQLite binding.
4    This is top level container model.
5    ConceptId is an UUID generated string to bind data from CData.
6    ConceptName is a name of the container.
7    '''
8    __tablename__ = 'Concepts'
9
10   Id = Column(Integer, primary_key=True)
11   ConceptId = Column(String)
12   ConceptName = Column(String)
13
14
15 class CData(Base):
16   '''
17   Database SqlAlchemy model for simple data to SQLite binding.
18   This is key-value child of Concepts table.
19   ConceptId is duplicate from parent to bind data after database commit.
20   Key is name of the property and Value is a value for given key.
```

```
21    '''
22    __tablename__ = 'CData'
23
24    Id = Column(Integer, primary_key=True)
25    ConceptId = Column(String)
26    Key = Column(String)
27    Value = Column(String)
```

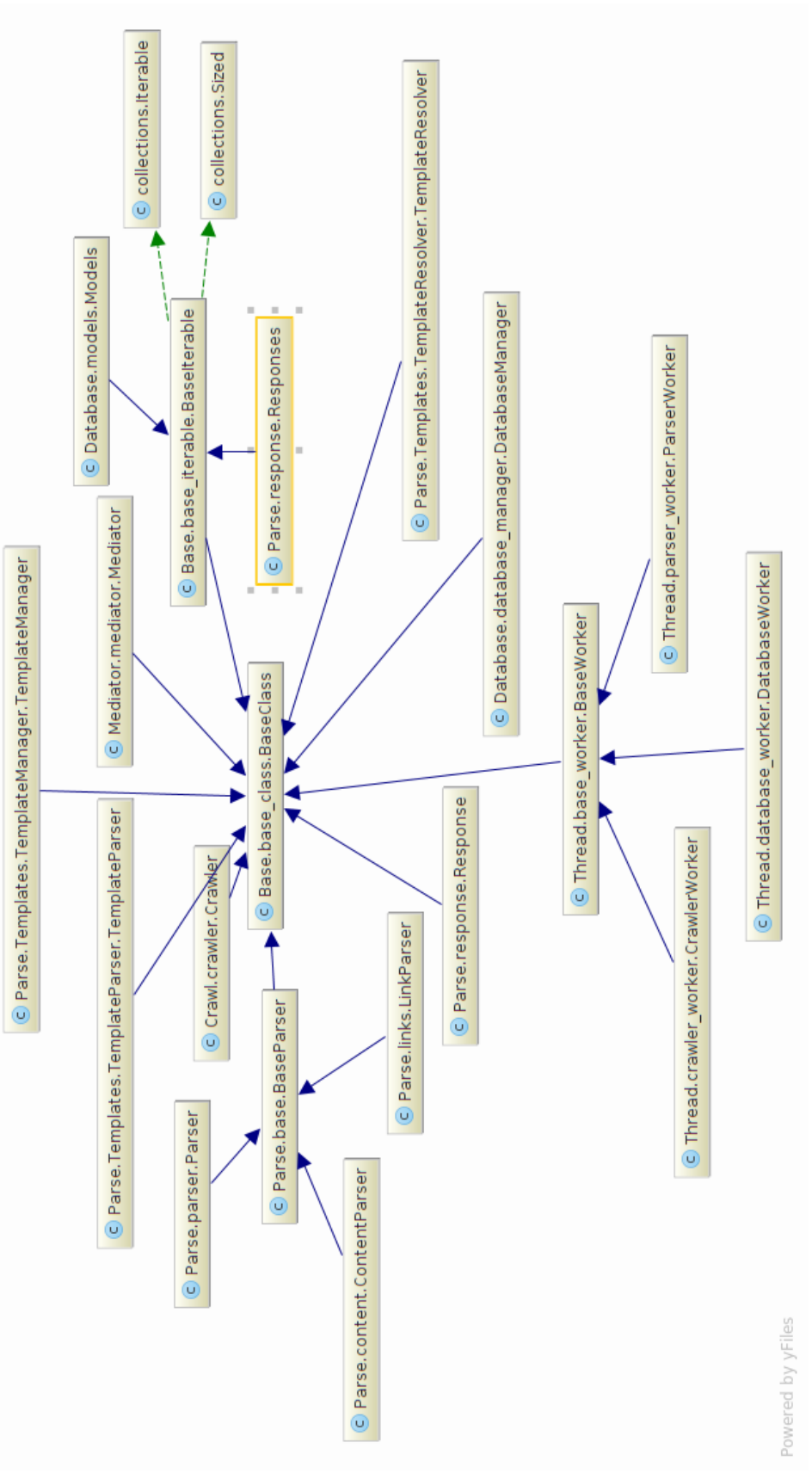
6 Worker

Za zrównoleglenie prac w naszym projekcie użyliśmy wzorca *Worker*. Polega on na zastosowaniu kilku wątków (tzw. *workerów*), których zadanie każdego jest pobranie danych ze źródła, przetworzenie ich i zwrócenie przetworzonych do odpowiedniego kontenera źródła. Ten wzorzec pozwala na jednoczesne zrównoleglenie wszystkich trzech modułów systemu, które wykorzystujemy, tj:

- crawlera
- parsera
- bazę danych

```
1  class Worker(BaseClass):
2     def __init__ (self, mediator: Mediator, max_workers: int):
3        """
4        Default constructor
5        :type mediator: Mediator Design Pattern
6        :type max_workers: Max workers
7        """
8        super().__init__()
9        self.mediator = mediator
10
11       # Define workers
12       self.parser = Parser()
13       self.db = DatabaseManager()
14
15       self.parse_worker = ParserWorker(self.mediator, self, self.parser)
16       self.crawl_worker = CrawlerWorker(self.mediator, self)
17       self.database_worker = DatabaseWorker(self.mediator, self, self.db)
18
```

```
19    def run (self) -> None:
20        keep_crawler = keep_parser = keep_db = True
21
22        while keep_crawler and keep_parser and keep_db:
23            keep_crawler = self.crawl_worker.run()
24            keep_parser = self.parse_worker.run()
25            keep_db = self.database_worker.run()
```



Rysunek 2.4: Diagram zaimplementowanych klas.