



國立臺灣大學電機資訊學院資訊網路與多媒體研究所  
碩士論文

Graduate Institute of Networking and Multimedia  
College of Electrical Engineering and Computer Science  
National Taiwan University  
Master Thesis

生成用於資訊網框架的Python服務元件  
Generating Python-Based Service Components for Web  
Frameworks

陳耀新  
Yao-Hsin Chen

指導教授：李允中 博士  
Advisor: Jonathan Lee, Ph.D.

中華民國 109 年 8 月  
August, 2020

國立臺灣大學碩士學位論文  
口試委員會審定書

生成用於資訊網框架的 Python 服務元件  
Generating Python -Based Service Components for Web  
Frameworks

本論文係陳耀新君（學號 R07944037）在國立臺灣大學資訊網路與多媒體研究所完成之碩士學位論文，於民國一百零九年八月十四日承下列考試委員審查通過及口試及格，特此證明

口試委員：

李允中

(簽名)

(指導教授)

劉三松

黃昭聰

王宜中

劉建宏

施吉昇

所長：



# 致謝

首先我要感謝我的指導教授李允中多年以來的教導以及做研究的方法；尋找研究的主題、相關研究的探討、問題的發掘、以及解決問題的方法；再來我要感謝台灣大學軟體工程研究室的成員，李冠緯、李俊緯、陳泓文、謝秉緯、余治杰、陳榮偉、蕭景忠、謝明軒、曾勁棠、錢怡君等，在一些共同的專案中一同努力、互相合作、相處融洽；由於許多人的幫助，讓我得以完成此篇論文。



# 摘要

在本研究中我們將已經修正完成的開源專案進行分析，並從中擷取能做為網路服務的部分，再為其生成可以部署在Flask上執行的伺服器程式碼及描述網路功能的WSDL，讓使用者可以透過服務組合技術來使用不同的服務元件建構成所需要的服務。

**關鍵詞** — 網路服務元件, Flask, PythonPoet, Python 抽象語法樹, 自動生成  
程式碼



# Abstracts

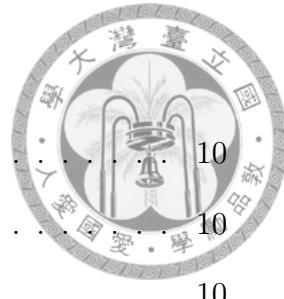
In this research, we analyzed the fixed project from open source and extract the informations we need. We also generate service code which can be deployed on Flask and a WSDL to describe the service, so that user can compose these service components to construct the needed service by service composition technique.

***Index terms*** — Web service component, Flask, PythonPoet, Python AST, Code generation



# Contents

致謝	ii
摘要	iii
Abstracts	iv
List of Figures	vii
Chapter 1 Introduction	1
Chapter 2 Related work	3
2.1 JavaPoet . . . . .	3
2.2 網路服務元件 . . . . .	4
2.3 Flask . . . . .	6
2.4 Python 抽象語法樹 . . . . .	6
2.5 生成Java服務元件 . . . . .	9
2.5.1 分析Java原始碼 . . . . .	10



2.5.2 生成Controller . . . . .	10
2.5.3 生成Invoker . . . . .	10
2.5.4 生成WSDL . . . . .	10
<b>Chapter 3 PythonPoet</b>	<b>13</b>
3.1 系統架構 . . . . .	13
3.2 API規則及範例 . . . . .	14
<b>Chapter 4 生成Python服務元件</b>	<b>19</b>
4.1 系統架構 . . . . .	19
4.2 利用Python AST分析程式碼 . . . . .	20
4.2.1 AST Analyzer . . . . .	20
4.3 生成Controller . . . . .	27
4.4 生成Invoker . . . . .	27
4.5 API server . . . . .	28
<b>Chapter 5 Conclusion</b>	<b>30</b>
<b>Chapter 6 Future work</b>	<b>31</b>
<b>Bibliography</b>	<b>34</b>



# List of Figures

2.1	JavaPoet class structure . . . . .	4
2.2	JavaPoet範例 . . . . .	5
2.3	Python直譯流程 . . . . .	7
2.4	AST的Abstract Grammar及存取AST的範例 . . . . .	7
2.5	Patient class example . . . . .	8
2.6	AST中拜訪class的方法 . . . . .	8
2.7	AST輸出範例結果 . . . . .	9
2.8	Spring Controller Example . . . . .	11
2.9	Play Controller (Top) and Route (Bottom) Example . . . . .	11
2.10	WSDL example for Java Service . . . . .	12
3.1	PythonPoet System Architecture . . . . .	15
3.2	PythonPoet class structure . . . . .	16
3.3	PythonPoet範例程式 . . . . .	17
3.4	PythonPoet輸出結果 . . . . .	18



4.1 生成Python服務元件的系統架構 . . . . .	21
4.2 GitHub範例 . . . . .	22
4.3 Generators架構 . . . . .	23
4.4 WSDL Generator . . . . .	24
4.5 AST中常見的method資訊 . . . . .	25
4.6 AST Analyzer class diagram . . . . .	26
4.7 Flask Controller Features . . . . .	27
4.8 Generated Controller Example . . . . .	28
4.9 Generated Invoker Example . . . . .	29
6.1 語言特性比較 . . . . .	32
6.2 Return type範例 . . . . .	33
6.3 使用JDEI來推斷data type的範例 . . . . .	33



# Chapter 1

## Introduction

為了提供符合使用者需求的Python網路服務，[6]會從GitHub上抓取Python open source，透過不同的策略處理無法正常執行的專案並將其做分類及修正。而這些修正好的專案我們以Method為單位進行切割，形成一個一個的網路服務元件。

由於抓取下來的元件無法直接地被使用，為了要將其部署在網路上，我們利用Flask框架來處理使用者請求，它是一個支援Python語言實作的web framework，根據MVC(Model View Controller)設計架構，Controller會負責處理使用者請求的流程，Invoker則實際將服務載入(MVC架構中的Model)，我們透過這個概念將抓取下來的服務元件部署在網路上，讓使用者可以透過存取這些元件來組成自己想要的服務。

不同的服務請求對應不同的服務元件，需要動態的將這些元件載入，



再利用服務組合技術來組合元件形成各類服務。為了將服務元件部署在網路上，需要為這些元件生成Controller及Invoker，以及用來描述服務的WSDL。然而，要自動生成Python程式碼，現今常見的open source中並沒有符合我們需求的工具，為此我們以JavaPoet為模板，開發了一個以Python為主的API來自動生成Controller及Invoker，並且再為該服務生成描述這個服務的WSDL。另外，我們也實作了API server用於負責與BPEL engine間的互動，處理serialize及deserialize等功能，使Controller和Invoker能夠和BPEL engine溝通。



# Chapter 2

## Related work

### 2.1 JavaPoet

做為自動生成Java文件的第三方函式庫，[2]提供了簡潔且易懂的API，並且將文件中不同的元素分成不同的規格，這些元素再以各自所屬的Spec來呈現。透過JavaPoet來存取class, method及field...等元素的內容，相對來說也較直接存取字串型式的Java檔案來得更為容易。

舉例來說：一個Java文件就是一個JavaFile，其中可能包含了abstract class, interface或是一般常見的class，JavaPoet會使用TypeSpec來處理；而class中可能包含自己的method，這時method則是被當成一個MethodSpec。JavaPoet的class structure如圖2.1所示。

圖2.2用JavaPoet產生出一段Helloworld的程式碼。另外，JavaPoet並不會檢查



生成的程式碼正確性，因此它雖然提供方便的API，但是在使用上仍然需要特別注意。

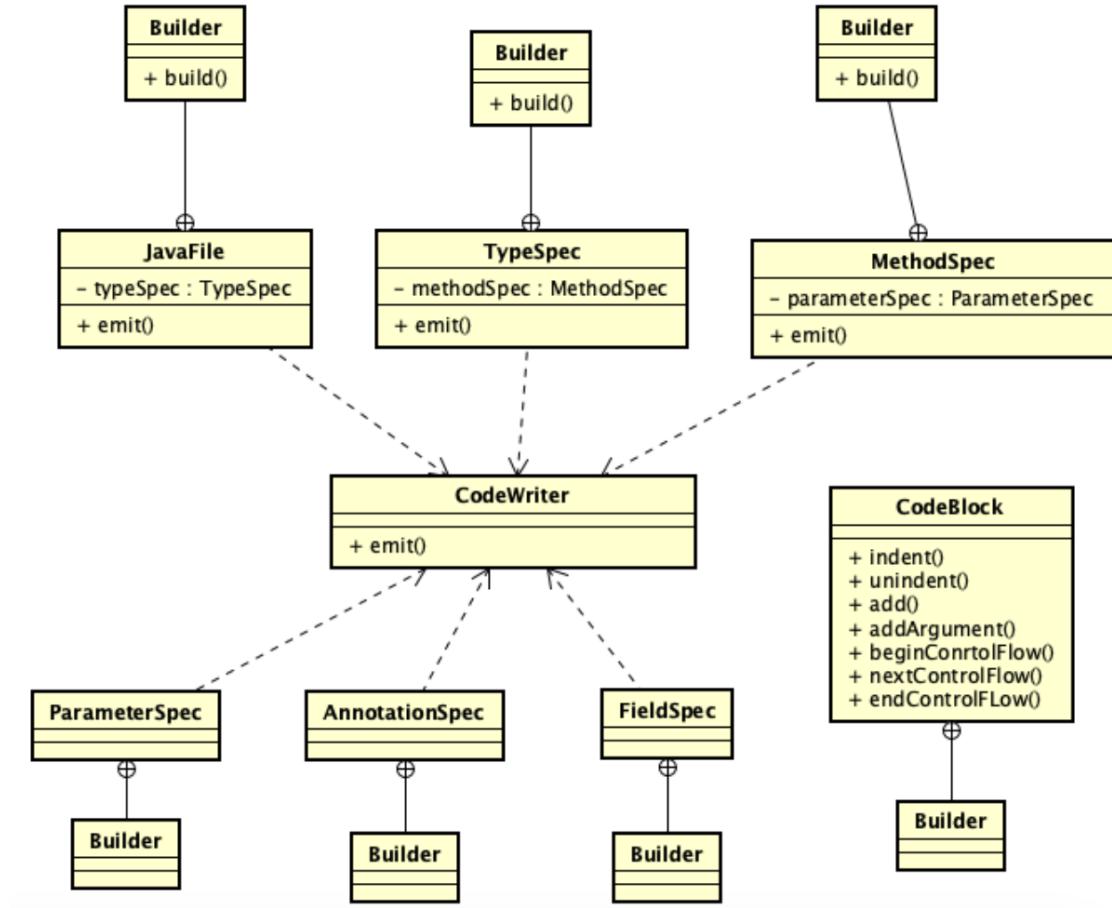


Figure 2.1: JavaPoet class structure

## 2.2 網路服務元件

在[9]中，Jian Yuang 等人將物件導向服務視為元件，並且依照不同建構及處理訊息的方式將元件分成了六大類。他們將服務元件看成是一個class，而message視



```

public class Test{
    public static void main(String[] args){
        MethodSpec main = MethodSpec.methodBuilder( name: "main")
            .addModifiers(Modifier.PUBLIC, Modifier.STATIC)
            .returns(void.class)
            .addParameter(String[].class, name: "args")
            .addStatement("$T.out.println($S)", System.class, "Hello, JavaPoet!")
            .build();

        TypeSpec helloWorld = TypeSpec.classBuilder( name: "HelloWorld")
            .addModifiers(Modifier.PUBLIC, Modifier.FINAL)
            .addMethod(main)
            .build();

        JavaFile javaFile = JavaFile.builder( packageName: "com.example.helloworld", helloWorld)
            .build();
    }
}

public final class HelloWorld {
    public static void main(String[] args) { System.out.println("Hello, JavaPoet!"); }
}

```

API rules (input)

output

Figure 2.2: JavaPoet範例

為attribute，service logic則做為method。

在[8]中，Jian Yang將網路服務元件也視為是class，然而不同的是，網路元件不再總是繼承自上述的六大類class，而是能夠繼承其他service class，這個特性使服務元件能夠被其他的服務使用，並且可以重新定義自己的message或service logic。

有別於上述提出的兩種方法，我們是把物件導向的原始碼直接轉換為服務元件來組成各種不同的服務。



## 2.3 Flask

Flask是以Python編寫而成的資訊網框架，輕量化的核心設計使Flask有著很高的擴充性，因此Flask又被稱為microframework。

與另一個常見得Python框架Django相比，Flask給予開發者們較大的彈性，讓使用者能夠根據需求來選擇自己想要的extension。而Django雖然功能較為完善，但是在ORM、表單驗證及模版引擎都有一套自己的做法，因此比起Django我們更傾向於選擇使用Flask來做為本論文中使用的主要框架，[1]中為Flask的官方文檔。

## 2.4 Python 抽象語法樹

Python AST(Abstract Syntax Tree)是Python官方所提供的module，[3]包含了Python AST的詳細介紹。在Python直譯的過程中，一個寫好的Python檔案(.py)會先被轉成語法樹，這個語法樹會再轉換為抽象語法樹，透過AST這個module我們便可以存取抽象語法樹的內容，來抽取我們想要的資訊。

Python在進行直譯的流程如圖2.3，經過不斷轉換後的抽象語法樹之後會再被轉成流程控制圖(Control Flow Graph, CFG)，最後成為byte code被執行。Python AST會透過visitor pattern來拜訪抽象語法樹中的節點。舉例來說，我們有一個名為Patient的class，如圖2.5所示。



要存取AST我們首先需要瞭解它所提供的Abstract Grammar(圖2.4)，而圖2.6中解釋如何存取class及method，範例中使用的這個method可以拜訪一個.py檔案中所有的function節點，我們用node.name就可以存取到該節點的名稱，此處的node是一個class，因此我們便可以拿到這個檔案中所有的class name。再來我們可以存取這個class的body，body儲存著這個class的子節點，檢查n是否包含'name'及'args'這兩個attribute，如果敘述為真，則n就是一個method節點，此時n.name可以得知這個class下(也就是Patient)的method name，最後的輸出結果如圖2.7所示。

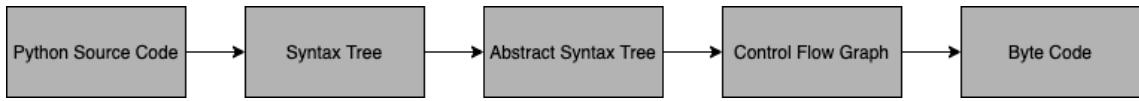


Figure 2.3: Python直譯流程

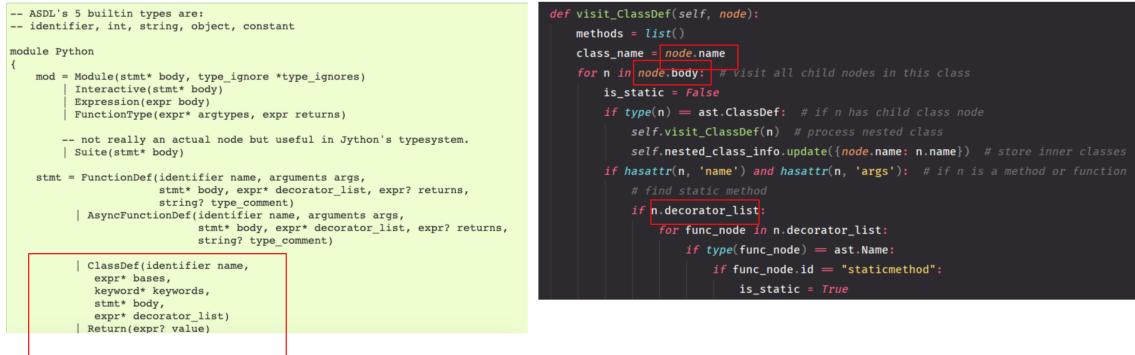


Figure 2.4: AST的Abstract Grammar及存取AST的範例



```

class Patient:

    def __init__(self, name, frequency):
        self.name = name
        self.frequency = frequency
        self.sensors = list()
        self.safeMap = dict()

    def attachSensor(self, sensor, safeRange):
        self.sensors.append(sensor)
        self.safeMap.update({sensor: safeRange})

    def getSensors(self):
        return self.sensors
  
```

Figure 2.5: Patient class example

```

def visit_ClassDef(self, node):
    methods = list()
    class_name = node.name
    for n in node.body: # visit all child nodes in t
        if hasattr(n, 'name') and hasattr(n, 'args'):
            method_name = n.name
            print(class_name, method_name)
  
```

Figure 2.6: AST中拜訪class的方法



```
Patient __init__  
Patient attachSensor  
Patient getSensors
```

Figure 2.7: AST輸出範例結果

## 2.5 生成Java服務元件

[7]從GitHub上抓取Java專案，透過不同的策略來進行修正，修正後的程式碼經過剖析後，以服務元件的架構進行拆解，再分別產出Controller、Invoker及WSDL。

為了能夠將他們部署在網路上，每個拆解好的元件再依照各個框架的請求形式，為該元件生成可被呼叫的介面(Controller)，而Controller內部再用Invoker來實際載入服務。

此外，為了要讓使用者能夠了解各個元件的功能，每個服務會產出一份WSDL(Web Service Description Language)，用來描述這個服務，告訴使用者這個服務需要有什麼輸入和輸出。藉由這個方式，便可以打破open source之間的分界，讓不同的open source都能夠被使用，使用者可以按照自己的需求，透過網路請求的方式使用不同的服務元件來組成需要的服務功能。



[5]以Java為例子實現上述架構，並透過以下步驟，分析Java原始碼、產生Controller及Invoker及生成WSDL，建立Java的服務元件。

### 2.5.1 分析Java原始碼

透過Java Reflection API以及Java Class Loader來parse byte code，class loader可以幫我們從JVM中載入class object，再透過Java Reflection API來進行分析。

### 2.5.2 生成Controller

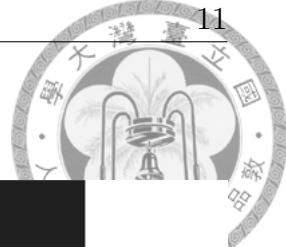
為每個服務產生自己的Controller，Controller的用途在替web framework註冊服務以及處理model和view之間的互動關係。由於每個Framework處理request的方式不盡相同，因此我們需要根據該Framework的作法，為它產出可以部署在上面的Controller，圖2.8是一個Spring Controller的例子。而圖2.9則是Play的Controller及route file，Play除了Controller外還需要定義好自己的route file，因此也需要產生一個route file。

### 2.5.3 生成Invoker

Invoker在這邊扮演的角色為MVC架構中的Model，用來呼叫 target method。

### 2.5.4 生成WSDL

WSDL中需要包含service name, service inputs, service outputs以及service request URL等資訊，[4]中包含了WSDL的介紹及各個元素所描述的功能。圖2.10中



```

@RestController("selab_service_controller_package_database_model_DatabaseObject_Controller")
public class DatabaseObject_Controller {
    private DatabaseObject_Invoker instanceInvoker = new DatabaseObject_Invoker();

    @Autowired
    private HttpServletRequest request;

    @Autowired
    private SpringStorage storage;

    @RequestMapping("/database/model/DatabaseObject/putDouble--java.lang.String--java.lang.Double")
    public String putDouble(java.lang.String java.lang.Double)
        @RequestParam(value = "self", required = true) String self;
        @RequestParam(value = "key", defaultValue = "", required = false) String key;
        @RequestParam(value = "value", defaultValue = "0", required = false) Double value) {
        RequestWrapper requestWrapper = ServiceUtil.createSpringRequestWrapper(request);
        if(!ServiceUtil.hasID(requestWrapper, storage)) {
            return ServiceUtil.createErrorMessage("There is no this ID: " + ServiceUtil.getProcessID(requestWrapper));
        }
        ServiceUtilObject serviceUtilObject = new ServiceUtilObject(requestWrapper, storage, "database");
        serviceUtilObject.loadDependency("model.DatabaseObject.putDouble(java.lang.String, java.lang.Double)");
        String invokerResult =
            this.instanceInvoker.putDouble__java_lang_String__java_lang_Double(serviceUtilObject, self, key, value);
        return invokerResult;
    }
}

```

Service URL: /database/model/DatabaseObject/putDouble--java.lang.String--java.lang.Double  
 Service input: self, key, value

Figure 2.8: Spring Controller Example

Controller

```

public class DatabaseObject_Controller extends Controller {
    private DatabaseObject_Invoker instanceInvoker = new DatabaseObject_Invoker();

    @Inject
    private PlayStorage storage;

    public Result putDouble(java.lang.String java.lang.Double(
        Http.Request request,
        String self,
        String key,
        Double value) {
        RequestWrapper requestWrapper = ServiceUtil.createPlayRequestWrapper(request);
        if(!ServiceUtil.hasID(requestWrapper, storage)) {
            return ok("There is no this ID: " + ServiceUtil.getProcessID(requestWrapper));
        }
        ServiceUtilObject serviceUtilObject = new ServiceUtilObject(requestWrapper, storage, "database");
        serviceUtilObject.loadDependency("model.DatabaseObject.putDouble(java.lang.String, java.lang.Double)");
        String invokerResult =
            this.instanceInvoker.putDouble__java_lang_String__java_lang_Double(serviceUtilObject, self, key, value);
        return ok(invokerResult);
    }
}

```

Service input: self, key, value

Route

```

@GET
"/database/model/DatabaseObject/putDouble--java.lang.String--java.lang.Double"
controller.package_database.model.DatabaseObject_Controller.putDouble__java_lang_String__java_lang_Double (
    request : play.mvc.Http.Request,
    self : java.lang.String,
    key : java.lang.String,
    value : java.lang.Double
)

```

Service URL: /database/model/DatabaseObject/putDouble--java.lang.String--java.lang.Double  
 Service input: self, key, value

Figure 2.9: Play Controller (Top) and Route (Bottom) Example



是一個描述Java服務的WSDL範例。

```
1 <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2 <wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:PatientMonitor_Pair="http://140.112.90.1
44:7122/PatientMonitorGradle/PatientMonitor/Pair" xmlns:selfNameSpace="http://140.112.90.144:7122/PatientMonit
orGradle/PatientMonitor/Pair/getValue" xmlns:wsdlrestful="http://schemas.xmlsoap.org/wsdl/restful/" xmlns:xsd=
"http://www.w3.org/2001/XMLSchema" targetNamespace="http://140.112.90.144:7122/PatientMonitorGradle/PatientMon
itor/Pair/getValue">
3   <wsdl:types>
4     <xsd:schema elementFormDefault="qualified"/>
5   </wsdl:types>
6   <wsdl:message name="getValueRequest">
7     <wsdl:part element="PatientMonitor_Pair:PatientMonitor_Pair" name="self" wsdlrestful:contentType="applicat
ion/json" wsdlrestful:requestType="get"/>
8   </wsdl:message>
9   <wsdl:message name="getValueResponse">
10    <wsdl:part name="serviceResult" type="xsd:double" wsdlrestful:contentType="application/json"/>
11    <wsdl:part name="standardOutput" type="xsd:string" wsdlrestful:contentType="application/json"/>
12  </wsdl:message>
13  <wsdl:portType name="getValuePort">
14    <wsdl:operation name="getValue">
15      <wsdl:output message="selfNameSpace:getValueResponse" name="out"/>
16      <wsdl:input message="selfNameSpace:getValueRequest" name="in"/>
17    </wsdl:operation>
18  </wsdl:portType>
19  <wsdl:service name="getValueService">
20    <wsdlrestful:address method="get" url="http://140.112.90.144:7122/PatientMonitorGradle/PatientMonitor/Pair
/getValue"/>
21  </wsdl:service>
22  <wsdl:import location="http://140.112.90.144/wsdl/PatientMonitorGradle/PatientMonitor/Pair.wsdl" namespace="
http://140.112.90.144:7122/PatientMonitorGradle/PatientMonitor/Pair"/>
23 </wsdl:definitions>
```

Figure 2.10: WSDL example for Java Service



# Chapter 3

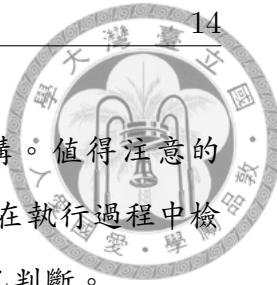
## PythonPoet

由於Python上並沒有相似於JavaPoet的API套件供使用，因此我們以JavaPoet為模板開發了一個Python API(PythonPoet)，用來生成Controller及Invoker。附帶一提，PythonPoet除了為我們生成Controller及Invoker之外，也可以根據不同的需求生成Python程式碼。

### 3.1 系統架構

與JavaPoet相似，在Python中也以程式碼中不同的元素做分類並提供不同的Spec，這些Spec會有自己的emit()方法，針對不同的Spec進行處理，即不同的Spec會使用自己的emit()方法來生成屬於自己的程式碼格式。

Spec的內容被包裝成一個CodeBlock，這個CodeBlock包含了該Spec的特徵資訊，最後CodeWriter會去剖析CodeBlock做相對應的處理，再將這些程式碼輸

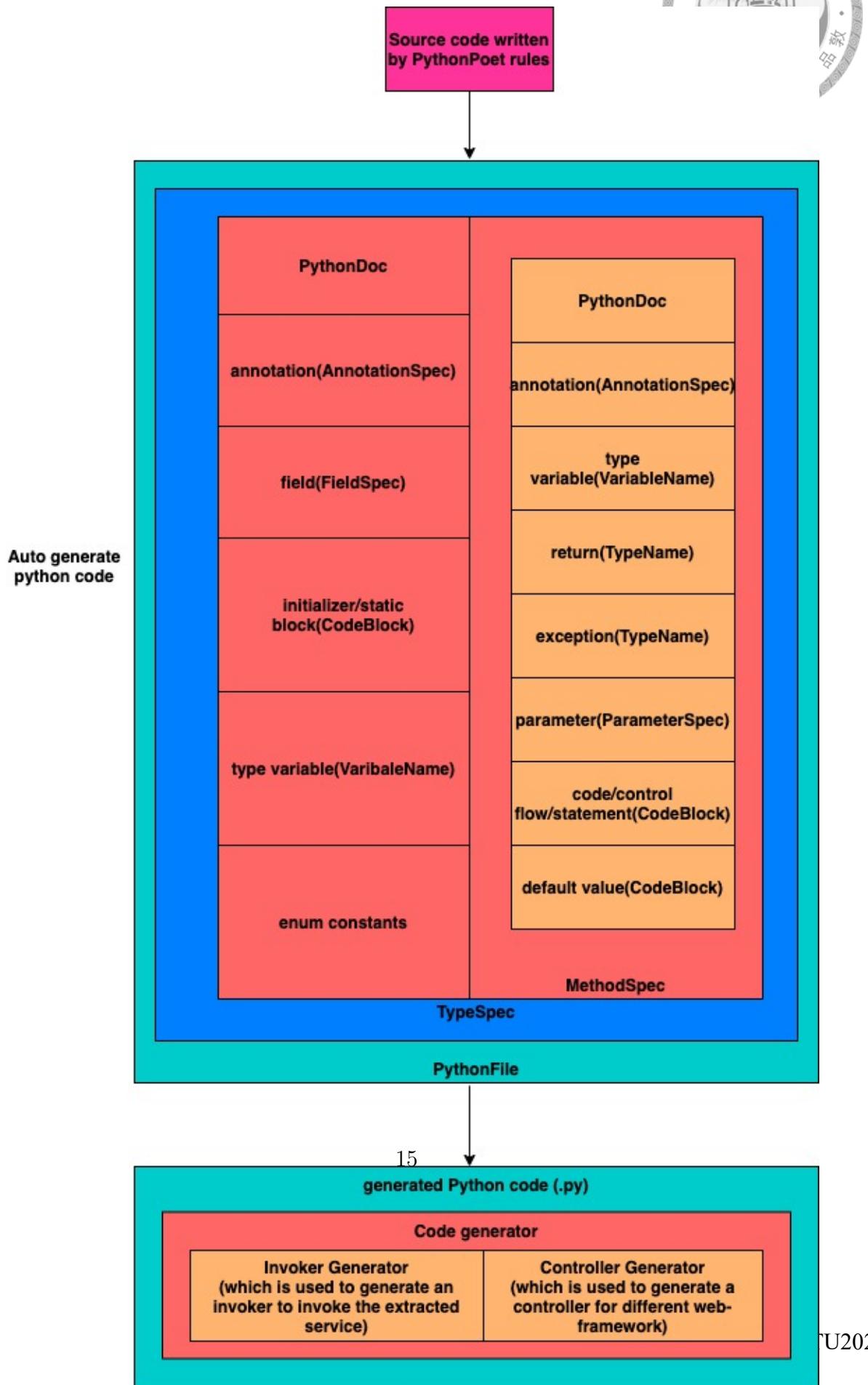


出，圖3.1為整體的架構，而圖3.2則是PythonPoet的類別圖架構。值得注意的是，PythonPoet使用上和JavaPoet相同，因此PythonPoet也不會在執行過程中檢查程式碼是否正確，因此生成的程式碼正確與否仍然由使用者自己判斷。

## 3.2 API規則及範例

PythonPoet在不同Spec中有實作API供使用，透過這些API可以簡化生成程式碼的流程，讓使用者只需要知道這個Spec有提供哪些API，再了解簡單的規則便可以產生自己想要的Python程式碼，無需額外考慮關鍵字及縮排的問題。

舉例來說，要產生一個method，除了method本身的名字外，我們需要知道這個方法傳入的參數資訊，假設method的名字及參數都已經給定，分別為test\_method、parameter1及parameter2，這時只需要利用add\_parameter()便可以在欲生成的方法中加入該參數，而要產生method的body我們需要先建立一個CodeBlock的instance，在CodeBlock中提供的add\_code\_segment()可以在method中加入一段敘述，\$L是一個place holder，不同的place holder在PythonPoet中會有不同的處理，此處\$L為替換p1的place holder，最後我們再透過MethodSpec的Builder建構出整個方法。這裡給出一個簡單的範例程式(圖3.3)，經過PythonPoet處理過後產生的程式碼為圖3.4。



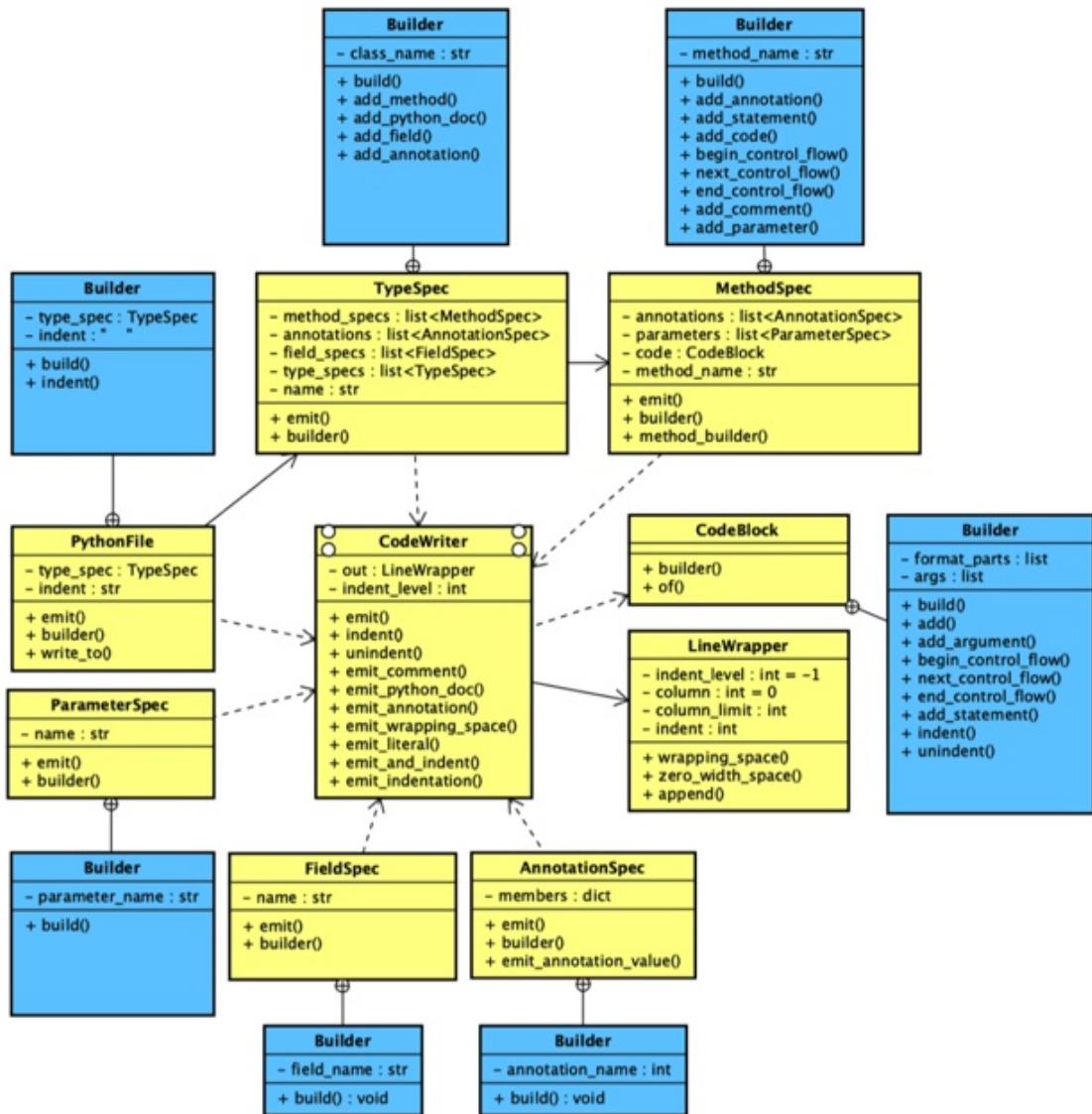
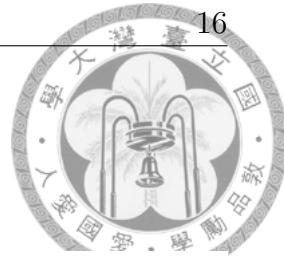
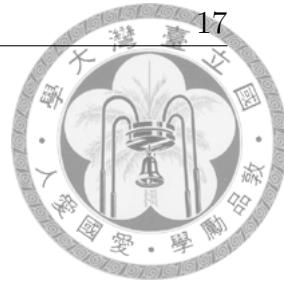


Figure 3.2: PythonPoet class structure



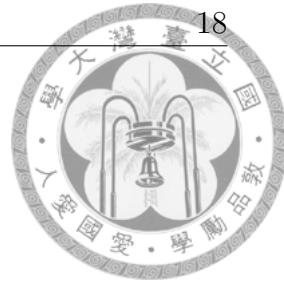
```
methodName = 'test_method'
p1 = 'parameter1'
p2 = 'parameter2'

MethodSpec.method_builder(methodName)
method_builder = MethodSpec.method_builder(methodName)

def get_method_body(parameter):
    builder = CodeBlock.builder()
    builder.add_code_segment('print("hello PythonPoet")\n')
    builder.add_code_segment('return $L', parameter)
    return builder.build()

method_spec = method_builder.add_parameter('p1') \
    .add_parameter('p2') \
    .add_code_block(get_method_body(p1)) \
    .build()
```

Figure 3.3: PythonPoet範例程式



```
def test_method(p1,p2):  
    print("hello PythonPoet")  
    return parameter1
```

Figure 3.4: PythonPoet輸出結果



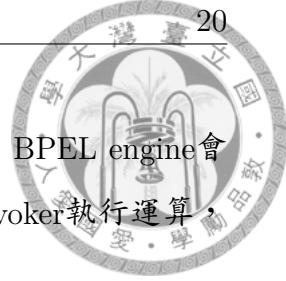
# Chapter 4

## 生成Python服務元件

在生成Python服務元件的這部分，參考了生成Java服務元件中的架構。<sup>[6]</sup>會從GitHub中抓取Python的開源專案，並且也使用不同的策略來修正。這些修正好的程式碼會透過Python AST來分析，並且以method為單位將其進行切割，我們從中擷取要建成服務元件所需的資訊，使用這些資訊來為該元件生成controller、invoker以及用來描述服務的WSDL。

### 4.1 系統架構

在生成Python服務元件部分，我們的系統架構如圖4.1。AST Analyzer主要用於分析Python程式碼，Method and Function Parser用來剖析我們分析後的資訊。Json Generator用來產生WSDL Generator所需的資訊，並且透過WSDL Generator來產生WSDL。Code Generator負責控制生成Controller及Invoker的流程，我們利用Controller Generator來產生Controller，再透過Invoker Generator來產



生Invoker。API server則用來處理與BPEL engine間的互動關係，BPEL engine會發送請求，API server收到請求後交由Controller處理routing，Invoker執行運算，再將結果回傳。

以下我們從GitHub中任意抓取一個Python檔案如圖4.2，之後將其交由AST Analyzer分析，經由ControllerGenerator及InvokerGenerator及WSDLGenerator為它產出Controller、Invoker及WSDL，圖4.3為Generator間的簡單架構，CodeGenerator會先從DadaModel拿到資訊，呼叫自己的generate()方法，再交由ControllerGenerator產生Controller或是InvokerGenerator產生Invoker。而WSDL的部分，JSONGenerator會負責設定生成WSDL所需的資訊再交由WSDLGenerator來產生(圖4.4)。

## 4.2 利用Python AST分析程式碼

修正過後的專案(或程式碼)，會透過AST Analyzer來分析，我們再從AST中抓取想要的資訊。Python可以寫成class的形式透過物件來呼叫方法，或單純只以function來執行運算，由於兩者的呼叫方式不同，因此依據Python的這些語言特性，我們儲存時再將資訊分為兩大類(method service unit及function service unit)。

### 4.2.1 AST Analyzer

AST可以存取Python程式碼的內容，像是class name、method name或是arguments...等，圖4.5列出一些常使用的資訊，以及我們在產生服務元件的過程中使用了哪些。

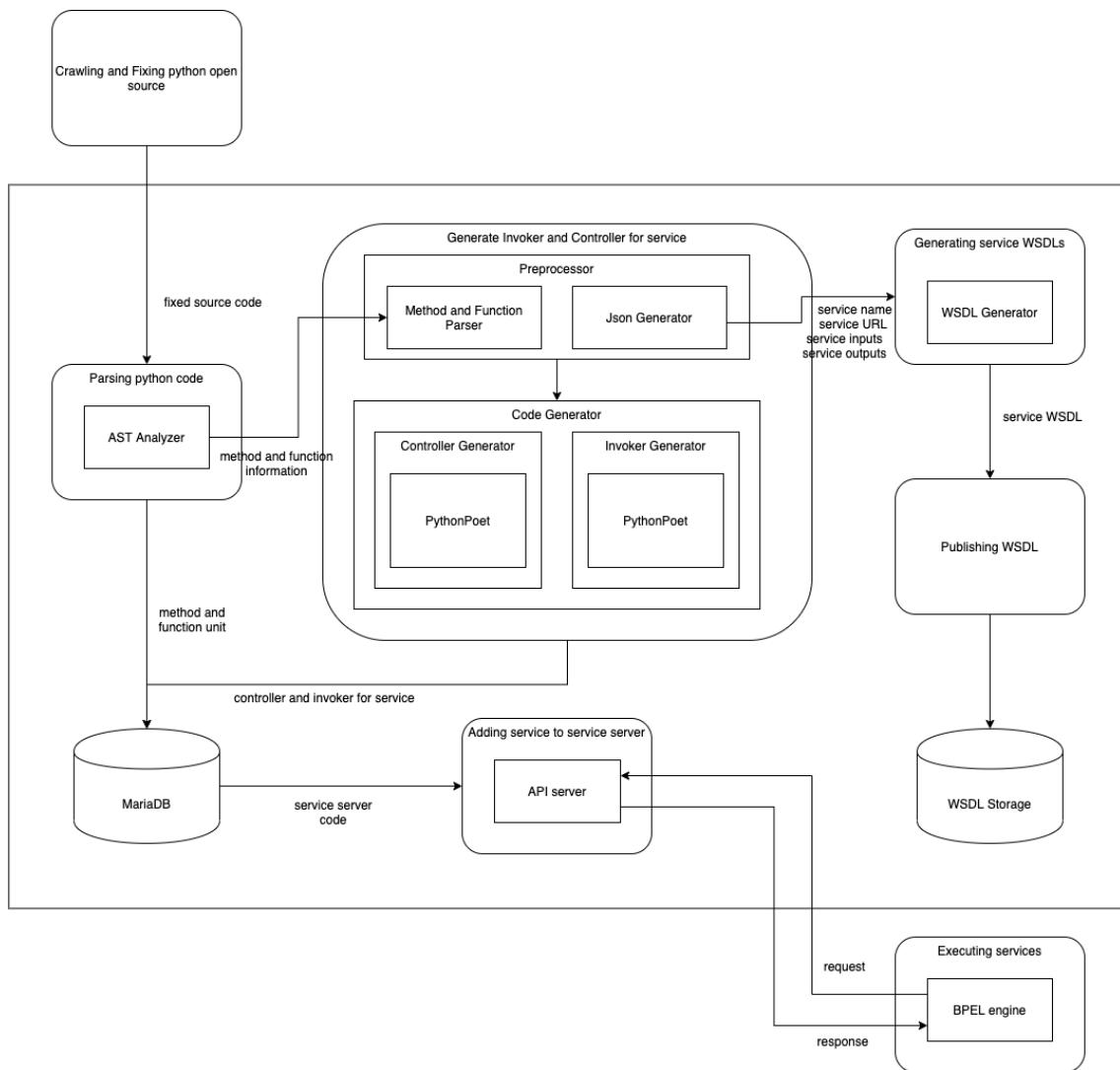
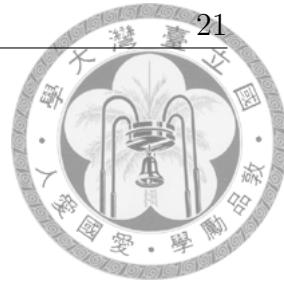
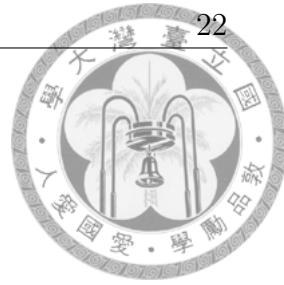


Figure 4.1: 生成Python服務元件的系統架構



```
class Criterion:  
    def __init__(self, name, levels):  
        self.name = name  
        self.levels = levels  
        self.descriptions = dict()  
  
    def addDescription(self, level, description):  
        self.descriptions.update({level: description})  
  
    def getLevels(self):  
        return self.levels  
  
    def measure(self, assignment):  
        pass  
  
    def toString(self):  
        return self.name
```

Figure 4.2: GitHub範例

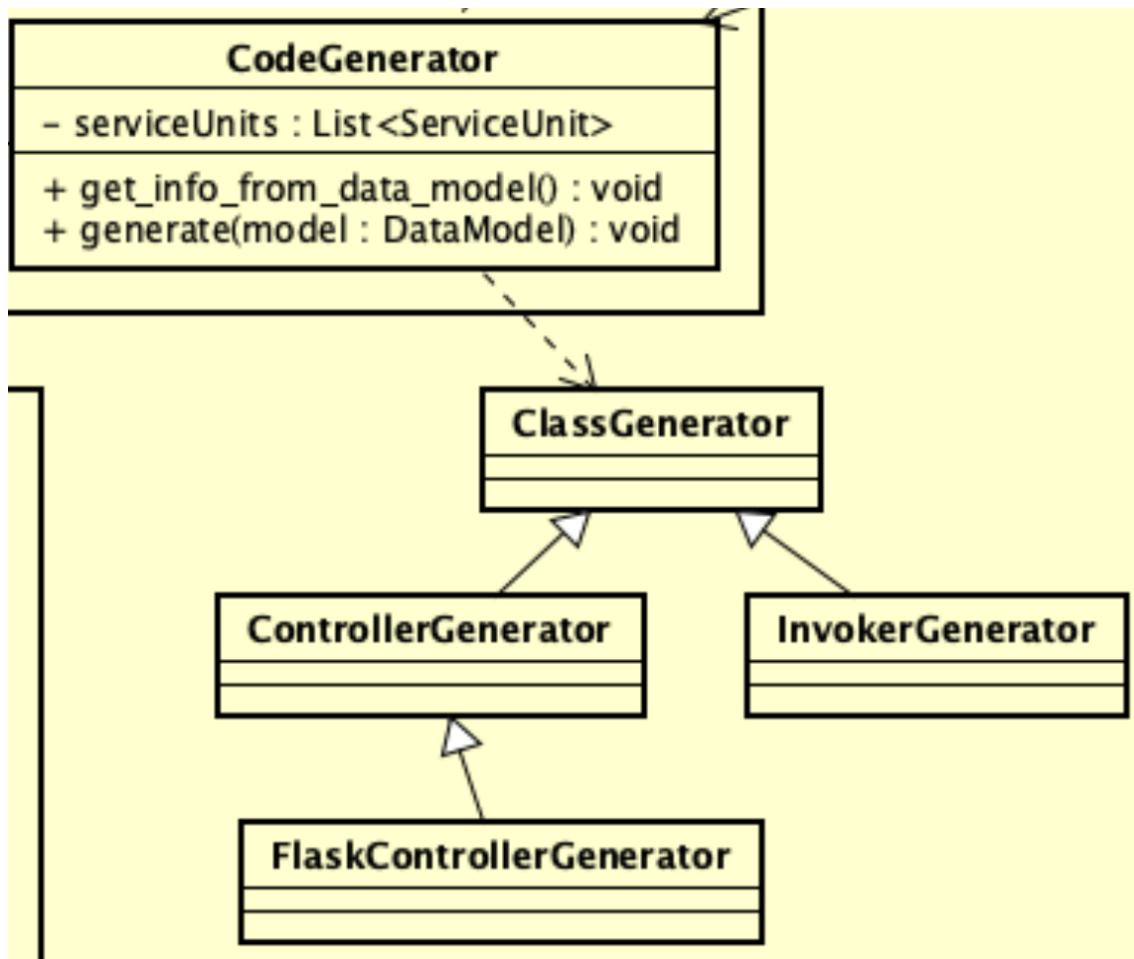
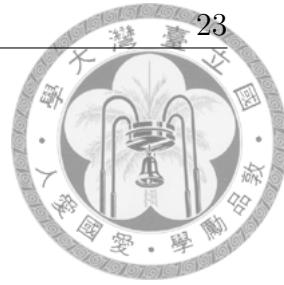


Figure 4.3: Generators架構

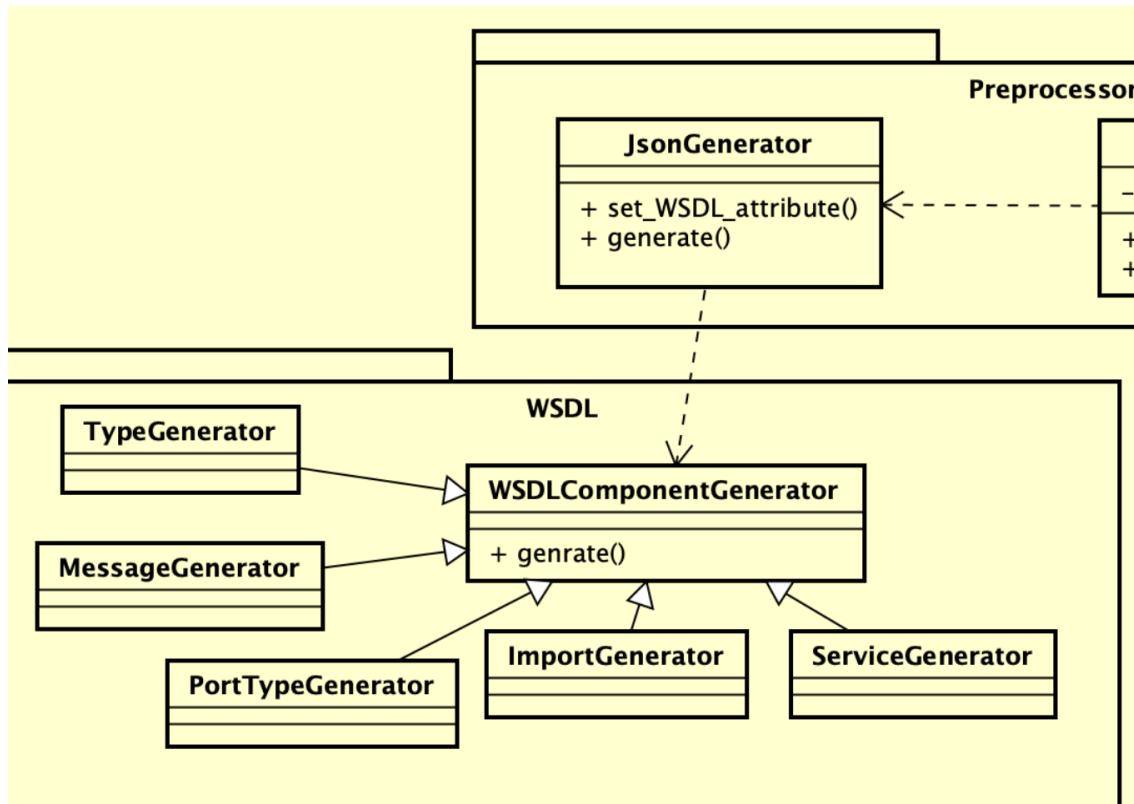
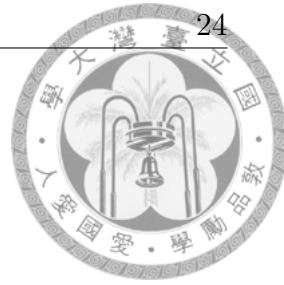


Figure 4.4: WSDL Generator



AST Analyzer主要負責擷取Method(Function)的資訊，以及將抓取的資訊整理過後存進data model，如圖4.6所示。可以看到，Main class中放著一個DataModel的instance，Analyzer從Main class中分析完檔案後會分別建立FunctionServiceUnit以及MethodServiceUnit，並且暫存在自己的List中，我們利用get\_method\_info()從Analzyer拿到所有建立的ServiceUnits，之後DataModel再呼叫自己的add\_method\_info\_to\_storage()方法來儲存建立的資訊。

<b>Method Information</b>	<b>Descriptions</b>	<b>Use or not in service component</b>
name	a raw string of the function name.	O
args	a arguments node	O
body	the list of nodes inside the function.	X
decorator_list	the list of decorators to be applied, stored outermost first	O
returns	the return annotation	X
type_comment	a string containing the PEP 484 type comment of the function (added in Python 3.8)	X

Figure 4.5: AST中常見的method資訊

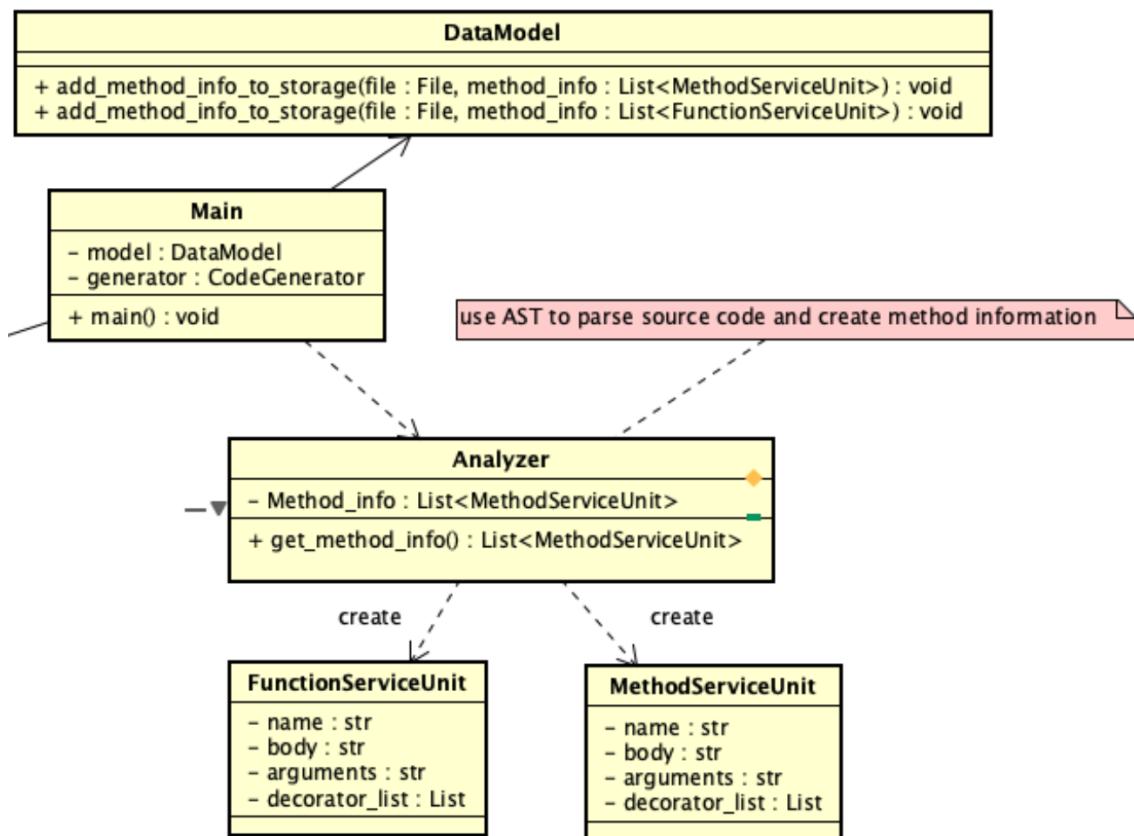
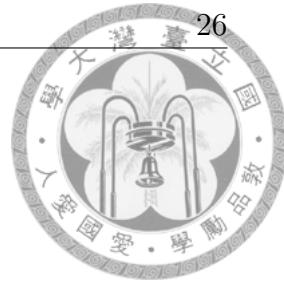


Figure 4.6: AST Analyzer class diagram



### 4.3 生成Controller

Controller需要針對Flask處理routing的方式來為生成可以在Flask上執行的Controller程式碼，Flask中會使用blueprint來管理routing，每個Controller中都需要定義好自己的blueprint，並且在Main Application(或Server)上進行註冊，如圖4.7。

Routing的部分，Flask使用decorator來處理，以圖4.7為例，@index\_blueprint.route()中，只需要填入service url即可。最後，要拿到request parameter，我們可以透過HTTP GET的方式，method所需的parameter會透過URL傳入，由於request.args回傳的是一個request的dictionary，只需要簡單使用request.args.get('your parameter')，便可以從request中拿到變數。圖4.8是我們產生的Controller例子。

```
from flask import Blueprint, request
index_blueprint = Blueprint('index', __name__)
@index_blueprint.route("/", methods=['GET'])
def index():
    t = request.args.get('t')
    s = request.args.get('s')
    # testing url: http://127.0.0.1:5000/?t=5&s=2
    return str(t) + str(s)
```

blueprint  
routing  
get parameter

Figure 4.7: Flask Controller Features

### 4.4 生成Invoker

在Invoker這裡，我們會從Controller拿到request parameter，做為method的input。



```

@app.route('/test_code/untitled/getLevels')
def getLevels__Criterion():
    self = request.args.get('self')
    requestWrapper = ServiceUtil.createFlaskRequestWrapper(request)
    storage = FlaskStorage()
    if not ServiceUtil.hasID(requestWrapper, storage):
        return ServiceUtil.createErrorMessage('There is no ID: ' + ServiceUtil.getProcessID(requestWrapper))
    serviceUtilObject = ServiceUtilObject(requestWrapper, storage, 'test_code')
    invokerResult = Criterion_Invoker.getLevels__Criterion(self, serviceUtilObject)
    return invokerResult

@app.route('/test_code/untitled/measure')
def measure__Criterion():
    self = request.args.get('self')
    assignment = request.args.get('assignment')
    requestWrapper = ServiceUtil.createFlaskRequestWrapper(request)
    storage = FlaskStorage()
    if not ServiceUtil.hasID(requestWrapper, storage):
        return ServiceUtil.createErrorMessage('There is no ID: ' + ServiceUtil.getProcessID(requestWrapper))
    serviceUtilObject = ServiceUtilObject(requestWrapper, storage, 'test_code')
    invokerResult = Criterion_Invoker.measure__Criterion(self, assignment, serviceUtilObject)
    return invokerResult
  
```

Figure 4.8: Generated Controller Example

除了input外，另外也需要取得class本身的instance來呼叫該method，在這裡我們透過API server中的ServiceUtilObject來為我們處理收到的parameter。

Invoker中的ServiceUtilObject class會負責將收到的Json String進行deserialize，獲得該instance之後，ServiceUtilObject再透過getattr()的方式，取得method並存放在executable中，最後我們再執行invokeService()，並將結果回傳給Controller。生成的Invoker結果如圖4.9所示。

## 4.5 API server

API server主要可以分成兩部分，一部分為Service Library，一部分則是Flask Server。Service Library主要負責處理從BPEL engine收到的parameter，將其序列化(serialize)，或是將Invoker運算過後的結果反序列化(deserialize)並回傳給BPEL engine。Flask Server則為Server主程式，它會把所有產生的Controller抓進來，再



```
@staticmethod
def getLevels_Criterion(self, serviceUtilObject):
    self__Instance = serviceUtilObject.deserializeInstance(self)
    executable = serviceUtilObject.getDeclaredMethod(self__Instance, 'getLevels')
    serviceResultObject = serviceUtilObject.invokeService(executable)
    return ServiceUtil.serialize(serviceResultObject)

@staticmethod
def measure_Criterion(self, assignment, serviceUtilObject):
    self__Instance = serviceUtilObject.deserializeInstance(self)
    assignment__Instance = serviceUtilObject.deserializeInstance(assignment)
    executable = serviceUtilObject.getDeclaredMethod(self__Instance, 'measure')
    serviceResultObject = serviceUtilObject.invokeService(executable, assignment__Instance)
    return ServiceUtil.serialize(serviceResultObject)
```

Figure 4.9: Generated Invoker Example

透過這些抓進來的Controller來負責處理不同的routing，由於我們無法事先知道需要Import哪些Controller以及註冊blueprint，這裡的Main Application也需要使用PythonPoet來自動產出。



# Chapter 5

## Conclusion

為了要提供適合的Python服務元件，我們從GitHub上抓取開源專案，之後透過AST Analyzer來剖析修正好的程式碼，並以Method為單位從中擷取函數及方法。

我們選擇使用Flask做為應用程式的框架，開發了PythonPoet，並使用它來產生Controller及Invoker。另外，為了使Controller及Invoker能夠正確的被使用，我們額外實作了一個API server，主要負責處理與BPEL engine間的互動。

為了要使BPEL engine能夠了解更服務的功能及資訊，我們還需要WSDL來描述每個服務，因此我們透過從JSONGenerator中產生的資訊，交由WSDLGenerator來負責生成對應該服務的WSDL。



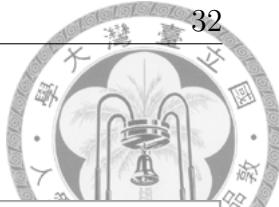
# Chapter 6

## Future work

圖6.1我們比較了Java與Python，可以發現Python的特性是在run time時，才決定每個parameter的data type。從圖6.2的範例中我們可以看到在做完運算後，回傳的每個return type有可能都是不同的，因此在處理生成WSDL的步驟時，沒辦法事先知道input與output的data type，因此WSDL中的type欄位需要另外找方法解決。

JEDI是一個open source的Python套件，可以用來推斷每個變數的資料型態，我們可以將AST中的所有input及output都交由JEDI處理，並且一一將這些資料型態儲存，在WSDL的type欄位我們為Python定義好新的type，用於處理所有可能的資料型態，讓BPEL engine來取得。

JEDI推測資料型態的方式如圖6.3所示，給定一存放在source中的字串，我們可以使用script.infer()來推測每個variable的type有哪些，最後我們再與mariaDB進



行整合，讓這些服務元件可以透過存取mariaDB的方式來取得。

JAVA	Python
strongly typed	strongly typed
static typed	dynamically-typed

Figure 6.1: 語言特性比較



```

if statement1:
    for i in range(0, 100):
        ...
        ...
    return 1
elif statement2:
    return 1.0
else:
    return "1"
  
```

Figure 6.2: Return type範例

```

>>> from jedi import Script
>>> source = """
... import keyword
...
... class C:
...     pass
...
... class D:
...     pass
...
... x = D()
...
... def f():
...     pass
...
... for variable in [keyword, f, C, x]:
...     variable"""
  
```

➡ ➡ ➡

```

>>> script = Script(source)
>>> defs = script.infer()
  
```

➡ ➡ ➡

```

Finally, here is what you can get from type :
>>> defs = [d.type for d in defs]
>>> defs[0]
'module'
>>> defs[1]
'class'
>>> defs[2]
'instance'
>>> defs[3]
'+function'
  
```

Valid values for type are `module`, `class`, `instance`, `function`, `param`, `path`, `keyword`, `property` and `statement`.

Figure 6.3: 使用JDEI來推斷data type的範例



# Bibliography

- [1] Flask. <https://flask.palletsprojects.com/en/1.1.x/>.
- [2] Javapoet. <https://github.com/square/javapoet>.
- [3] Python ast. <https://docs.python.org/3/library/ast.html>.
- [4] Wsdl. <https://www.tutorialspoint.com/wsdl/index.htm>.
- [5] H.-Y. Huang. Atomic service generation from java-based open source software, 2019.
- [6] G.-W. Lee. Auto-classifying-fixing build errors in python-based open source projects, 2019.
- [7] J. Lee, H.-W. Chen, and H.-Y. Huang. Analyzing the build-ability of open source java projects on github. In *Taiwan Conference of Software Engineering 2019*, 2019.
- [8] J. Yang. Web service componentization. *Commun. ACM*, 46(10):35–40, Oct. 2003.
- [9] J. Yang and M. P. Papazoglou. Web component: A substrate for web service reuse and composition. In A. B. Piddock, M. T. Ozsu, J. Mylopoulos, and C. C. Woo, editors, *Advanced Information Systems Engineering*, pages 21–36, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.