

# 需求文档结构化处理软件设计报告

**团 队 名 称 TEAM**

**团 队 成 员 BY1817164 孙竖敬**

**团 队 成 员 SY1806116 赵健宏**

**团 队 成 员 SY1806611 张鹏程**

**团 队 成 员 ZY1806106 刘 良**

**2018年12月**

**目录**

[需求文档结构化处理软件设计报告 1](#_Toc532080790)

[一、系统概述 3](#_Toc532080791)

[二、架构设计 5](#_Toc532080792)

[1、系统架构设计 5](#_Toc532080793)

[2、类图主要类功能介绍 6](#_Toc532080794)

[3、组件图及相应组件层对应类图分析 9](#_Toc532080795)

[4、系统状态图 13](#_Toc532080796)

[5、系统时序图 14](#_Toc532080797)

[三、详细设计及软件展示 14](#_Toc532080798)

[1、OCL约束 14](#_Toc532080799)

[(1)展示层 14](#_Toc532080800)

[(2)控制层 15](#_Toc532080801)

[(3)响应层 16](#_Toc532080802)

[(4)功能支持层 18](#_Toc532080803)

[(5)数据层 21](#_Toc532080804)

[2、转换规则详细设计 25](#_Toc532080805)

[(1) 规约的GWT描述方式： 25](#_Toc532080806)

[(2) 转换规则说明 27](#_Toc532080807)

[3、软件样例展示 30](#_Toc532080808)

[4、界面展示及数据库设计说明 37](#_Toc532080809)

[(1)界面展示 37](#_Toc532080810)

[(2)数据调度器 38](#_Toc532080811)

[(3)数据库设计说明 39](#_Toc532080812)

# 一、系统概述

项目原始要求：给定自然语言文档，设计一套标签，手工嵌入到需求文档中，使其成为流程式的需求。并使用机器学习方法来自动插入标签。然后根据标签来提取信息，形成结构化的需求。

在此要求的基础上，明确给定的自然语言文档按照GWT形式进行描述，产生的结构化需求描述遵循RUCM格式并组成完整文档。

本项目立足软件需求描述方法，旨在建立一个将GWT形式描述的需求文档进行自动化处理转换为RUCM文档的实用系统。RUCM的需求描述方法相比GWT描述形式既保留了自然语言的表达能力与易懂性，又减少了表达的随意性与二义性。项目预期主要应用在软件工程的需求描述环节，在实现上要综合GWT需求文档与RUCM文档的特点设计文档标签，使用机器学习方法实现标签自动嵌入，由机器根据标签提取信息，转换成RUCM形式的需求文档。

项目要求的具体条目有

1. 给定自然语言需求文档：自然语言描述的需求文档是系统的输入数据。给定的自然语言文档以GWT形式描述。在软件开发过程中，需求文档记录了需求分析和导出的结果，是对用户和系统需求的全面详细的描述，自然语言的描述方法是最直观易懂且应用广泛的需求描述方法，但是自然语言的二义性使得这种方式有着明显的弊端，GWT格式的需求描述相比一般的自然语言描述方式准确性更高，但仍然可以转换成结构化程度更高的描述方式。
2. 设计一套标签：标签可以较高的层次上对需求文档的结构进行抽象，反映出软件需求的流程特性，在本项目中，需求文档标签是GWT描述与RUCM描述之间的桥梁，通过标签标注出GWT描述中与RUCM描述对应的实体与关系，在此基础上可以迅速地根据标签抽取信息产生RUCM文档。
3. 使用机器学习方法自动插入标签
4. 使用机器学习方法：使机器能够对自然语言描述的需求文档进行处理需要自然语言处理技术的支撑，在本项目中主要体现为自然语言的信息抽取（Information Extraction）过程，作为信息抽取的子过程，分词、标注、分块等文本处理操作都可以结合机器学习技术，通过引入机器学习方法可以得到准确性更高、灵活性更好的文档处理工具。
5. 自动插入标签：自动插入标签是项目中应用机器学习方法的目的，通过对自然语言描述的处理，在对描述内容达到一定理解程度的基础上可以由机器自行判断文档标签的位置，特别是在在处理规模较大的文档时可以拥有高效率。
6. 根据标签提取信息：标签完成了对原始的自然语言描述的结构划分，显示出了需求的流程特点。在为文档插入标签后，GWT描述中的RUCM内容被标记出来，根据标签可以快速提取出文档中对应于RUCM描述的内容。

形成结构化的需求：在通过标签提取出信息后，提取出的信息遵循RUCM描述格式，在此基础上，可以将信息保存为RUCM描述的需求文档，获得结构化的需求。

# 二、架构设计

## 1、系统架构设计

展示层

控制层

接收事件

响应层

GWT导入器

领域背景导入器

RUCM Generator

响应事件

获取支持

调用数据

数据支持层

领域背景

数据调度器

TagedGWT

GWT

调用数据

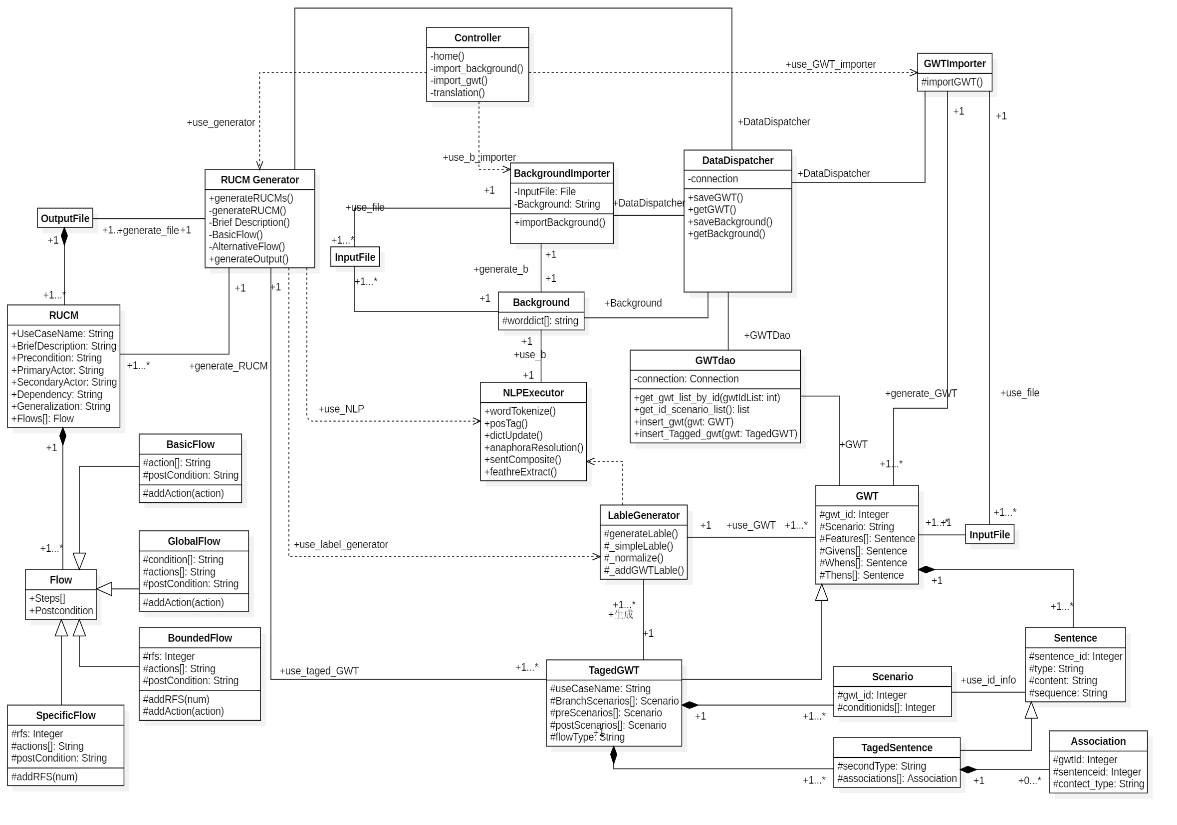
功能支持层

自然语言处理器

中间标签生成器

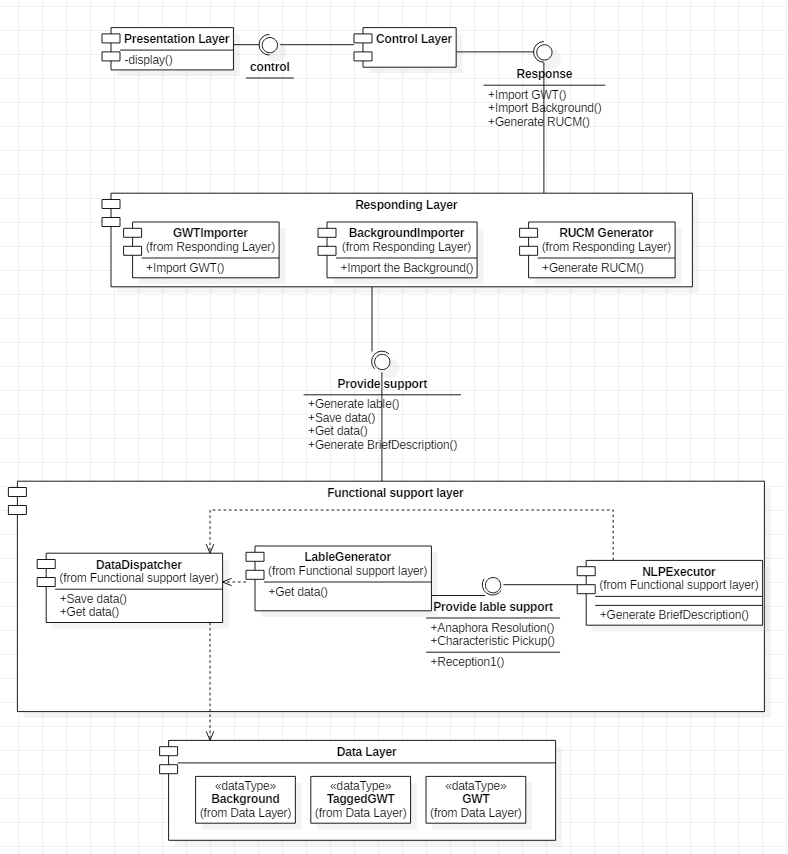
* + - 展示层：用于显示界面用于用户操作和查看
    - 控制层：接收展示层所产生的事件，并调用相应操作
    - 响应层：真正用于响应事件的实现
    - 功能支持层：提供支持用于响应层中组件实现响应（类图中，现在不存在数据调度器，因为当前类图已经十分庞大，接下来在设计阶段会进行分解）
    - 数据支持层：存储数据

## 2、类图主要类功能介绍

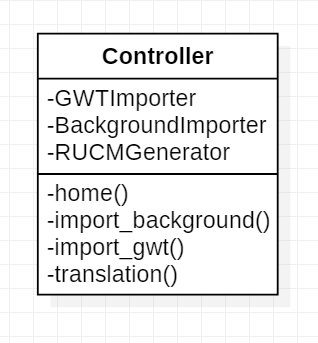


* + - Controller：负责相应外部事件，产生相应动作，负责调用RUCM Generator::generateRUCMs()，领域背景导入器::领域背景信息导入()，GWT导入器::导入GWT。
    - RUCM Generator：提供将GWT转化为RUCM的接口。
    - BackgroundImporter：提供导入领域背景的接口，将领域背景信息结构化存储。
    - GWTImporter：提供导入GWT的接口，将输入文档中的GWT文档，按照GWT类进行存储。
    - NLPExecutor：提供自然语言处理的一系列操作的接口，现在均为public为了使调用方便。但是在未来确认了自然语言处理器和RUCM Generator所使用的方法之后，可以进一步进行封装。
    - LableGenerator：将GWT转化为TagedGWT，便于接下来的完全转化过程。
    - Sentence：
    - Id：独一的表示出GWT中的语句。
    - Type：Feature、Given、When、Then的二级级标签。
    - Content：例：Given中的一行。或Feature中的一段话。
    - TagedGWT：添加标签后的GWT。在GWT的基础，添加GWT之间的关系，确定自身所在的Flow类型。每个GWT由Scenario进行表示，存储id和与当前GWT相关的condition的id。
    - TagedSentence：TagedGWT中的Sentence会变为TagedScence。该类除了GWT本身的内容意外，还存储二级标签下的三级标签以及句子之间的关系。
    - Association：句子之间的关系，首先需要由另一个句子的id，另外存储关系的类型，比如两句意思相同，或两句意思相反等。
    - GWTdao：GWT的Data Access Object，用于对于GWT类的存储。
    - DataDispatcher：数据调度器，为其他模块提供存储和读取数据的支持，具体实现为对领域背景信息和GWT数据的存储与读取。

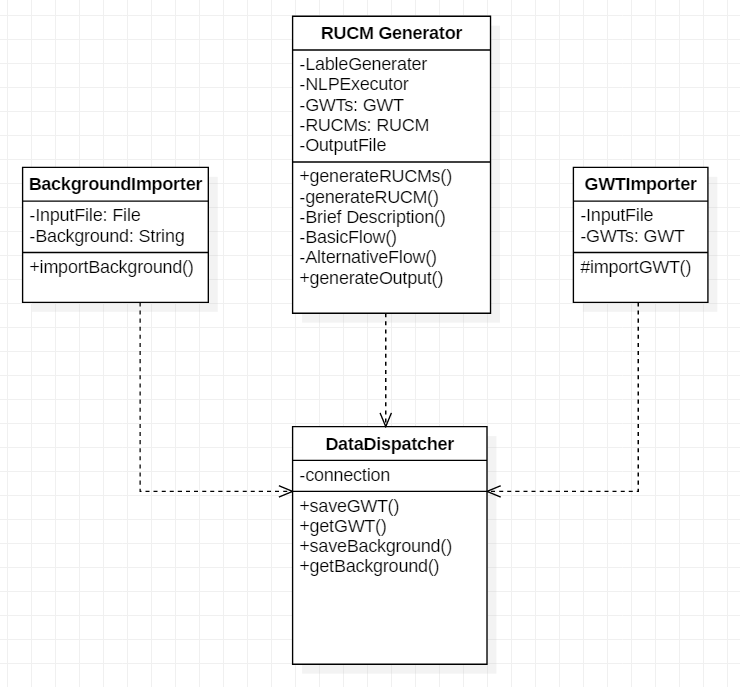
## 3、组件图及相应组件层对应类图分析



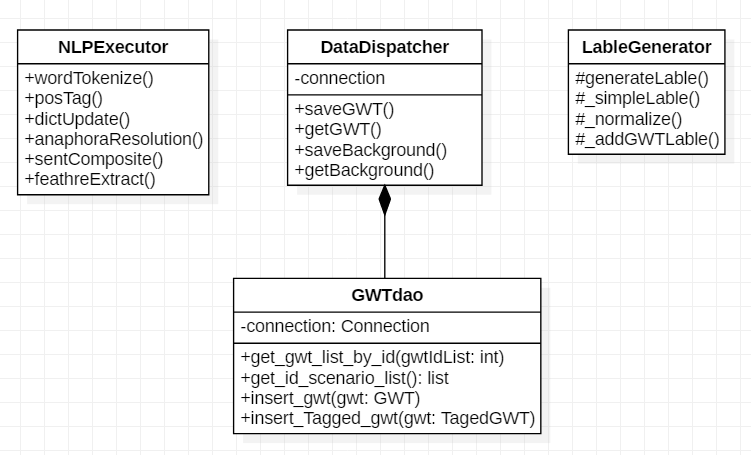
系统分为展示层、控制层、响应层、功能支持层、数据层，由于展示层涉及到后端技术，类图中就没有写具体的类，其他四个结构层对应着不同不分的类图，展示层中收到前端的输入，控制层收到相应信息，并调用相应的操作，响应层真正用于响应事件的实现，功能支持层对应着中间标签生成器和自然语言处理器用于提供支持用于响应层中组件实现响应；在最内部的数据支持层中对应着领域背景、TagedGWT、GWT、数据调度器，通过数据调度器调用其他三个组件。



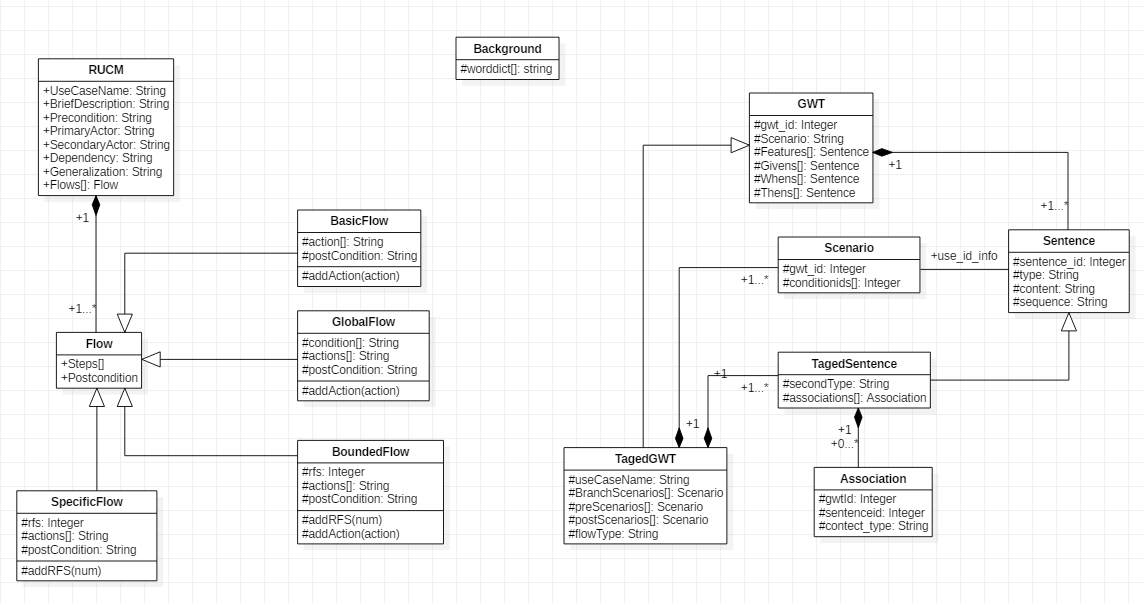
控制层控制GWT输入、领域背景输入以及RUCM生成的接口，对应着总体的三个用例。分别对应输入相应规则的GWT文档、GWT文档对应领域的数据库以及利用转换规则将GWT文档生成RUCM结构化文档。



该类图部分对应着组件图中的响应层，上面的控制层所控制的即是现在的响应层，响应层中的领域背景输入、GWT输入以及RUCM生成器都将分别利用数据层中的数据调度器来调用数据层中的领域背景数据库、GWT输入数据以及利用中间标签标记的GWT文档数据。

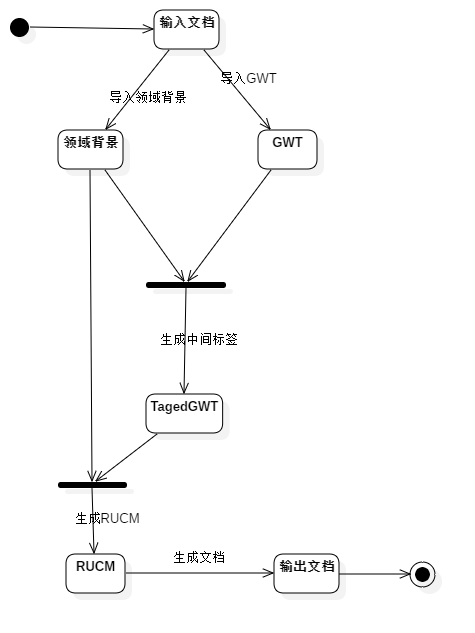


组件图中的功能支持层中对应着类图中的部分如上，该部分包含自然语言处理器和中间标签生成器。用于实现响应层中部件的具体实现，所以同样利用数据调度器调用数据进行支持分析。



数据层分为三类数据：领域背景数据、GWT描述数据、中间转换规则，用于提供相应功能的数据。

## 4、系统状态图



本系统的主要目的是文档的转化，而该状态图展示了在这个系统中文档的状态变化，声明了每种文档状态生成之前所需要的文档，表明了文档之间的相互依赖。

## 5、系统时序图



该图展示了整个系统的时序过程，展示了系统通过controller对整个系统的调度。同时中间标签生成器与自然语言处理器是通过递归方式进行调用，从而与controller隔离开来，减少了类与类之间的耦合关系，使得整个系统更加结构鲜明。

# 三、详细设计及软件展示

## 1、OCL约束

### (1)展示层

* 显示初始页面

context controller::home():file

pre:

post:return = index.html

### (2)控制层

* 调用领域背景导入，导入领域背景，显示导入结果

context controller::import\_background():(file, message)

pre: input\_file <> Null

index.html->领域背景导入->button->submit = true

post:if 导入成功

then return = ('index.html', message='领域背景导入成功')

else return = ('index.html', message='领域背景导入失败')

* 调用GWT导入，导入GWT，显示导入结果

context controller::import\_gwt():(file, message)

pre: input\_file <> Null

index.html->gwt导入->button->submit = true

post:if 导入成功

then return = ('index.html', message='GWT导入成功')

else return = ('index.html', message='GWT导入失败')

* 调用RUCMGenerator，首先选择需要导入的GWT，然后进行GWT->RUCM的转化，得到转化成功的RUCM，并且显示生成结果

context controller::translation():(file, message)

pre: index.html->开始转换->button->submit = true

post:if 选择GWT列表

then return = ('index.html', message=转化结果)

else return = ('index.html', message='已经取消转化')

RUCM.doc--或者这个不是在这里出现，而是在RUCMgenerator中出现

inv: notEmpty(转化结果)

### (3)响应层

* BackgroundImporter为领域背景信息导入器，将用户上传的领域背景信息文件保存到数据库供训练自然语言处理器，每个BackgroundImporter对应一个输入的领域背景文件，每一个BackgroundImporter都使用一个DataDispatcher数据调度器对象。

context: BackgroundImporter

inv:self.InputFile ->size()=1

self.DataDispatcher->size()=1

* 调用importBackground方法将InputFile保存的领域背景信息通过DataDispatcher数据调度器对象存入服务器。操作完成后DataDispatcher数据调度器关联的领域背景信息数量增加。

context BackgroundImporter::importBackground()

pre self.DataDispatcher -> size() = 1

self.InputFile -> size()=1

post self.DataDispatcher.Background -> size() >= (self.DataDispatcher.Background -> size())@pre

* GWTImporter为GWT描述导入器，将用户上传的GWT文件使用自然语言处理器NLPExecutor进行处理，提取主要内容保存到数据库供用户选择处理。GWTImporter每次读取一个用户输入的GWT文件，每一个GWTImporter都使用一个DataDispatcher数据调度器对象。

context:GWTImporter

inv:self.inputFile->size()=1

self.DataDispatcher->size()=1

self.NLPExecutor->size()=1

* 传入GWT文件路径或内存中保存的GWT文档内容，通过NLPExecutor提取出文档各字段内容转换成GWT对象，使用DataDispatcher将GWT对象属性保存到数据库。操作完成后DataDispatcher数据调度器关联的GWT数据对象数量增加。

context:GWTImporter::importGWT(filepath:string,filecontent:string )

pre:filepath.size>0 or filecontent.size()>0

self.DataDispatcher->size()=1

self.NLPExecutor->size()=1

post: self.DataDispatcher.GWT -> size() >= (self.DataDispatcher.GWT -> size())@pre

* RUCMGenerator为RUCM描述生成器，对用户选择的GWT进行分析并添加中间标签，提取中间标签标记的内容生成RUCM对象，将一组RUCM对象输出为一个文本文档OutputFile。RUCMGenerator使用DataDispatcher数据调度器按照用户选择的GWT的唯一id从数据库中读取GWT数据创建一组GWT对象，使用标签生成器LabelGenerator分析GWT对象属性并添加中间标签转换成TaggedGWT，使用自然语言处理器NLPExecutor分析一组TaggedGWT对象生成一个完整的RUCM描述。

Context RUCMGenerator

inv:

self.NLPExecutor->size()=1

self.LabelGenerator->size()=1

self.dataDispatcher->size()=1

self.OutputFile->size()=1

self.RUCM->size()>=1

### (4)功能支持层

* 将输入的一组GWT数据id传入DataDispatcher从数据库中读取对应的GWT数据，将其转换成一组对应的RUCM描述，生成输出文档内容。

Context RUCMGenerator::generateRUCMs(gwtIdList:set)

Pre: self.NLPExecutor->size()=1

self.LabelGenerator->size()=1

self.DataDispatcher->size()=1

post: self.OutputFile->size()=1

self.RUCM->size()>=1

* 将输入的一组TaggedGWT对象转换成一个完整的RUCM描述。

Context RUCMGenerator::generateRUCM(taggedList:set)

Pre: self.NLPExecutor->size()=1

self.LabelGenerator->size()=1

taggedList->size()>0

post: RUCM -> size() = (RUCM -> size())@pre+1

* 由输入的一组TaggedGWT对象生成对应RUCM描述的Brief Description。

Context RUCMGenerator::briefDescription(taggedList:set，rucm:RUCM)

Pre: self.NLPExecutor->size()=1

taggedList->size>0

post:rucm.briefDescription<>None

* 由输入的一组TaggedGWT对象生成对应RUCM描述的Brasic Flow。

Context RUCMGenerator::basicflow(taggedList:set，rucm:RUCM)

Pre: self.NLPExecutor->size()=1

taggedList->size>0

post: rucm.basicFlow<>None

* 由输入的一组TaggedGWT对象生成对应RUCM描述的Specific Alternative Flow、Bounded Alternative Flow、Global Alternative Flow(若RUCM含有这一种或几种可选事件流)。

Context RUCMGenerator::alternativeFlow(taggedList：set，rucm:RUCM)

Pre: self.NLPExecutor->size()=1

taggedList->size>0

post: rucm.specificAlt<>None or rucm.boundedAlt<>None or rucm.globalAlt<>None

* 将生成的一组RUCM对象转换成输出文件文本内容。

Context RUCMGenerator::generateOutput()

Pre: self.RUCM->size>0

post: self.OutputFile->size()=1

* 初始化\_connect=connect\_db

Context GWTdao::\_connection()

init:connect\_db()

* 各函数的作用：

context GWTDao::getGWTListById(id int):GWT

post: return=gwtList

context GWTDao::getid\_scenario\_list():(GWT.id:int,GWT.Scenario:String)List

post: return self.\_\_connection.execute('select id,scenario from GWT\_tb').fetchall()

context GWTDao::insert\_gwt(data:GWT)

post:self.\_\_connection.commit()

context GWTDao::insert\_Tagged\_gwt(data:GWT):Boolean

post: self.\_\_connection.execute()

### (5)数据层

* DataDispatcher为数据调度器，为其他模块提供存储和读取数据的支持，具体实现为对领域背景信息和GWT数据的存储与读取。

Context DataDispatcher

inv:self.connection->size()=1

self.GWTDao->size()=1

* 将读取的领域背景信息文件保存到本地。

Context DataDispatcher::saveBackground(file:str)

pre:file.size()>0

post:self.Background->size()=（self. Background ->size())@pre+1

* 将输入的GWT通过GWTDao对象保存到数据库中

Context DataDispatcher::saveGWT(gwt:GWT):set

pre: self.connection->size()=1

self.GWTDao->size()=1

post:self.GWTDao.GWT->size()=(self.GWTDao.GWT->size())@pre+1

* 读取本地的领域背景信息文件。

Context DataDispatcher::getBackground():str

pre:self.Background->size()=1

post:return=backgroundFile

* 读取数据库中指定id的GWT数据生成GWT对象。

Context DataDispatcher::getGWT(gwtid:integer):GWT

pre: self.connection->size()=1

self.GWTDao->size()=1

post:return=gwt

* LableGenerator为中间标签生成器，对输入的一组GWT进行分析并添加中间标签，转换成TaggedGWT对象。

Context LableGenerator

inv:self.NLPExecutor->size()=1

* 对传入的一组GWT对象进行分析，添加标签生成对应的一组TaggedGWT对象

Context generateLable(gwtlist:set):set

pre:gwtList->notEmpty()

self.NLPExecutor->size()=1

post:self.TaggedGWT->size()>（self.TaggedGWT->size())@pre

* 对传入的一组GWT对象进行分析，检索语句中的As a、GLOBAL等关键词，添加标签UseCaseName和global的flowType

Context \_simpleLable(gwtList:set):set

pre: gwtList->notEmpty()

self.NLPExecutor->size()=1

post: self.TaggedGWT->forAll(useCaseName<>None or flowType=’global’)

* 对传入的一组GWT对象进行分析，添加标签为basic、specific或bounded的flowType，添加标签BranchScenarios.

Context \_addGWTLable()

pre: gwtList->notEmpty()

self.NLPExecutor->size()=1

post: self.TaggedGWT->forAll(flowType=’basic’ or flowType=’specific’ or flowType=’bounded’ or flowType=’global’ )

* NLPExecutor为自然语言器，向上为RUCMGenerator和LabelGenerator提供自然语言处理功能支持

Context NLPExecutor

inv:self.Background->size()=1

self.nlpTool->size()=1

* 对输入的中文句子进行分词，返回句子词汇的集合。

Context NLPExecutor::wordTokenize(sentence:string):set

pre: self.nlpTool->size()=1

sentence.size()>1

post:return=wordlist

* 对输入的中文句子进行词性标注，返回句子词汇词性二元组的集合。

Context NLPExecutor::posTag(sentence:String)

pre: self.nlpTool->size()=1

sentence.size()>1

post: return=wordtaglist

* 使用用户输入的自定义词典重新训练自然语言处理模型，提高领域理解能力。

Context NLPExecutor::dictUpdate(wordDict:string)

pre: self.Background->size()=1

self.nlpTool->size()=1

post: self.nlpTool<> [self.nlpTool@pre](mailto:self.nlpTool@pre)

* 将输入的两个句子向量化，计算并返回句子的相似度。

Context NLPExecutor::similarity(sent1:string,sent2:string):real

pre: self.nlpTool->size()=1

sent1.size()>0

sent2.size()>0

post:return=similarity

* 将输入的中文文本切分成句子，返回句子的集合。

Context NLPExecutor::splitSentence(text:string)：set

pre: self.nlpTool->size()=1

text.size()>0

post:return=sentlist

* 从输入的文本中提取重要信息，生成文本摘要。

Context NLPExecutor::generateSummary(text:string):string

pre: self.nlpTool->size()=1

text.size()>0

post:return=summary

## 2、转换规则详细设计

### (1) 规约的GWT描述方式：

一个GWT用来描述一个用例中的某个具体场景，在用GWT详细描述用例时，首个GWT描述用例各步骤正常执行的场景，当用例中某一个或几个步骤运行异常而引发新的场景时，新的场景分别由后续新的GWT进行描述，在任意时刻发生某特殊事件而引发新的场景时，新的场景也由后续新的GWT进行描述。

**GWT格式**

**Story:** 以“As a 参与者，用例”形式说明参与者与此GWT描述的用例名称，用As a开头表示参与者，有多个参与者时按主次排序，顿号分隔，参与者与用例名称以逗号分隔。

**Scenario:**  概括描述用例发生时此GWT所适用的一个具体场景。

**Business Rule:** 此GWT描述的用例场景所遵循的基本规则。

**Given:**

Preconditions: 此GWT所描述的具体场景发生的前置条件，描述同一用例的所有GWT有公共的前置条件，即为用例的前置条件；描述用例各步骤正常执行场景的GWT与其他GWT在异常步骤处具有不同的前提，因此也具有对应的不同的前置条件。对描述任意时刻发生某特殊事件所引发的新场景的GWT，除具有用例的前置条件，还具有该特殊事件发生作为前置条件，且该前置条件以关键词GLOBAL标识。当此用例对其他用例有包含、扩展或泛化关系时，分别以关键词加用例名形式“INCLUDE 用例名”、“EXTENDED 用例名”、“GENERALIZE 用例名”进行标识。

Fixed data:前置条件中要求的固定数据。

**When:**

Action:以简单句形式描述完成此用例场景的具体步骤。对描述用例中某一个或几个步骤运行异常而引发的新场景的GWT，其Action序列只在异常发生后与首个GWT的Action序列不同；对描述任意时刻发生某特殊事件所引发的新场景的GWT，其Action序列仅描述该事件发生后的用例场景步骤。

Input data:具体场景中的输入数据。

**Then:**

Output data:用例场景的个步骤执行完毕后的输出数据。

Postcondition：描述用例场景的个步骤执行完毕后对系统 和参与者的影响结果。

### (2) 转换规则说明

#### (a) 添加中间标签

输入文档存储时，将输入的GWT按照Feature、Scenario、Given、When、Then五个部分进行存储，每个部分包含若干个GWT描述句子，每个句子都用一个Sentence类对象表示，添加标签时对GWT中的每个句子都进行标记，保存在Sentence的对象实例中，使用stype属性存储句子类型标签，content属性保存句子文本，sequence属性保存句子在本部分描述中的序号标签。

输入文件以纯文本格式编写，文件读入内存后以字符串类型存在，使用字符串的split方法以Story字段切分出每一个GWT描述，对任意一个GWT描述去除其换行符，通过正则表达式

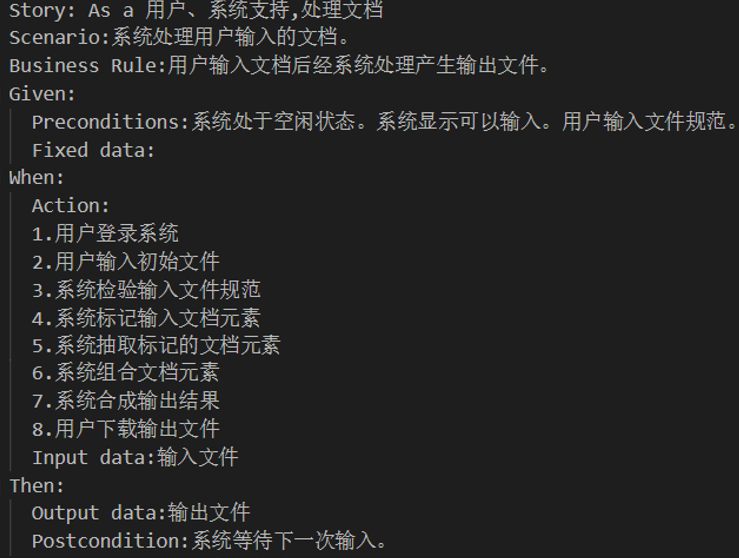
”\s\*(.\*)\s\*Scenario:\s\*(.\*)\s\*Business\sRule:\s\*(.\*)Given:\s\*Preconditions:\s\*(.\*)\s\*Fixed\sdata:\s\*(.\*)When:\s\*Action:\s\*(.\*)\s\*Input\sdata:\s\*(.\*)'Then:\s\*Output\sdata:\s\*(.\*)\s\*Postcondition:\s\*(.\*)\s\*”

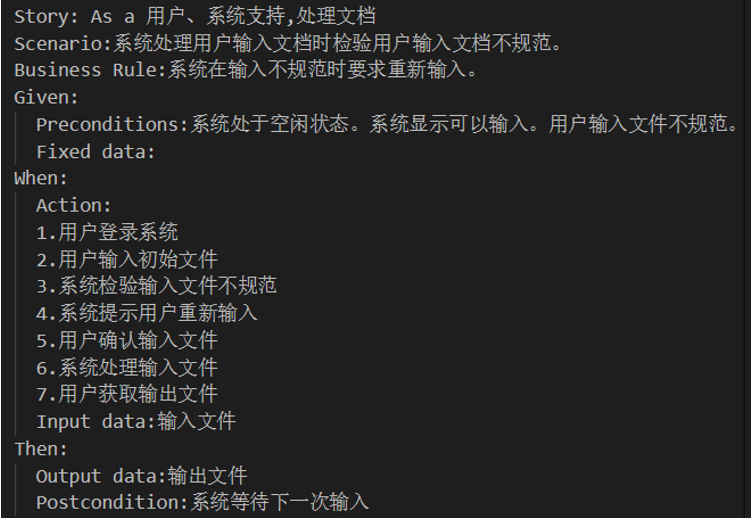
提取GWT中每个字段的文本内容。对Story字段内容，使用正则表达式匹配提取出逗号分隔符后的用例名称，保存到GWT对象的useCaseName属性下，逗号分隔符前的语句添加标签“story”，保存到 GWT对象的Features属性下；对Scenario字段的内容，将其文本保存到GWT对象的Scenario属性下；对Business字段的内容，添加标签“business\_rule”，保存到 GWT对象的Features属性下；对Given字段下的Preconditions文本内容，通过自然语言处理工具进行分句，为每一句添加标签”precondition”保存到GWT对象的Givens属性下；对Given字段下的Fixed data文本内容，添加标签“fixed\_data“并保存到GWT对象的Givens属性下；对When字段下的Action文本内容，通过自然语言处理工具进行分句，为每一句添加标签”action”保存到GWT对象的Whens属性下，并添加其”sequence”标签；对When字段下的Input data文本内容，添加标签“input\_data“并保存到GWT对象的Whens属性下；对Then字段下的Output data文本内容，添加标签”output\_data”保存到GWT对象的Thens属性下；对Then字段下的Postcondition文本内容，添加标签“postcondition“并保存到GWT对象的Thens属性下。

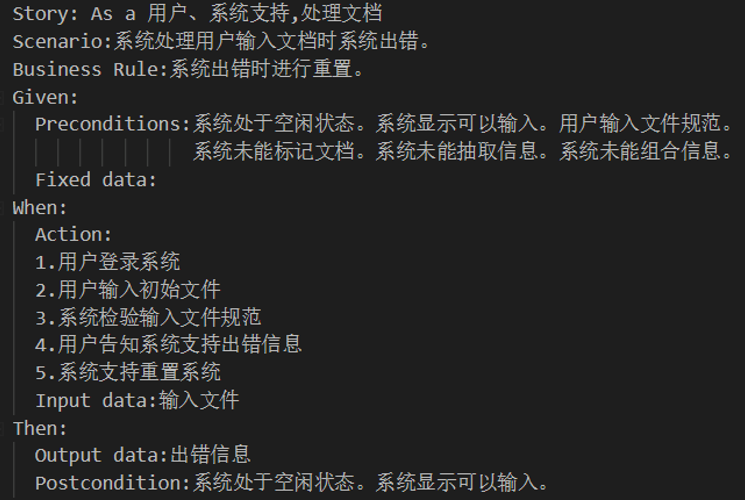
在已经初步添加标签的GWT基础上，查看输入GWT的useCaseName属性，将useCaseName相同的GWT划分为同一组，首个GWT描述用例各步骤正常执行的场景，将其flowType标记为“basic”，以此GWT的Given属性下的precondition语句为基准GWT，与其他GWT进行比较。对只有一条precondition与基准GWT不同的GWT，将其flowType标记为“specific”，认为此GWT描述用例中某一个步骤运行异常而引发的新场景，按顺序比较此GWT与基准GWT在Whens属性下的“action”语句，在两语句不相等时表明此处为发生异常的步骤，将此步骤的序号保存到GWT的BrachScenarios属性下；对有几条precondition与基准GWT不同的GWT，将其flowType标记为“bounded”，认为此GWT描述用例中某几个步骤运行异常而引发的新场景，比较此GWT独有的precondition语句与基准GWT的Whens属性下的所有”action”标记的句子，通过句子向量化衡量相似性的方法，对与precondition句子相似但不完全相等的action句子，认为此步骤是此GWT描述场景对应的异常步骤之一，将其sequence添加到此GWT的BrachScenarios属性下；在检验其他GWT的preconditon语句时，使用正则表达式检测语句中是否含有“GLOBAL”关键字，对具有此关键字的GWT，认为其描述的是在任意时刻发生某特殊事件而引发的新场景，将其flowType标记为“global”，提取出GLOBAL关键字标记的语句，保存到此GWT的BrachScenarios属性下。

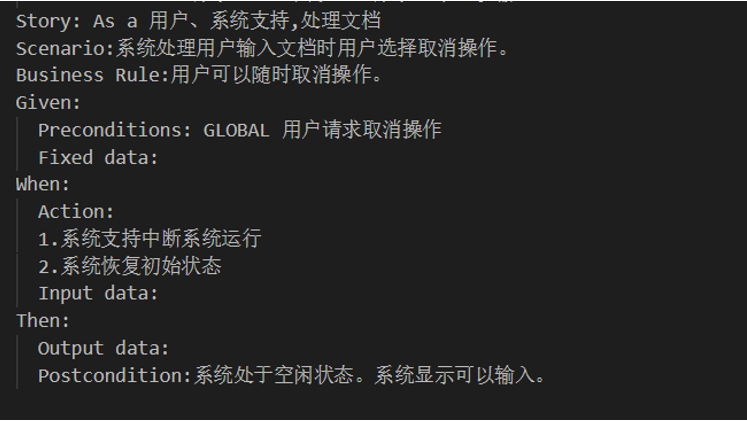
## 3、软件样例展示

## 对以下的输入示例：

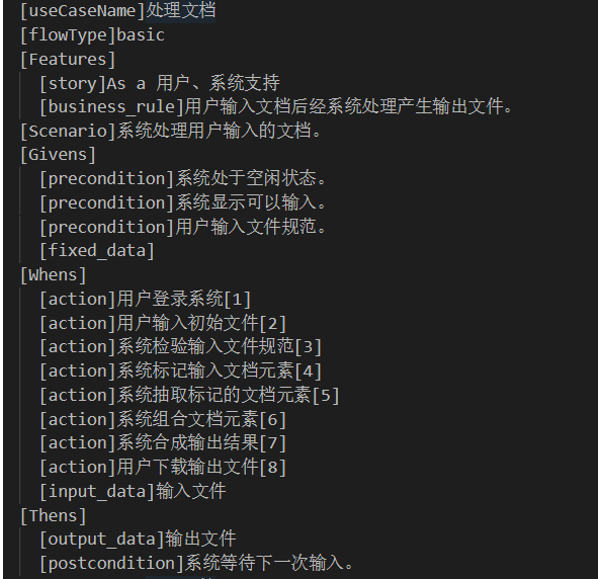


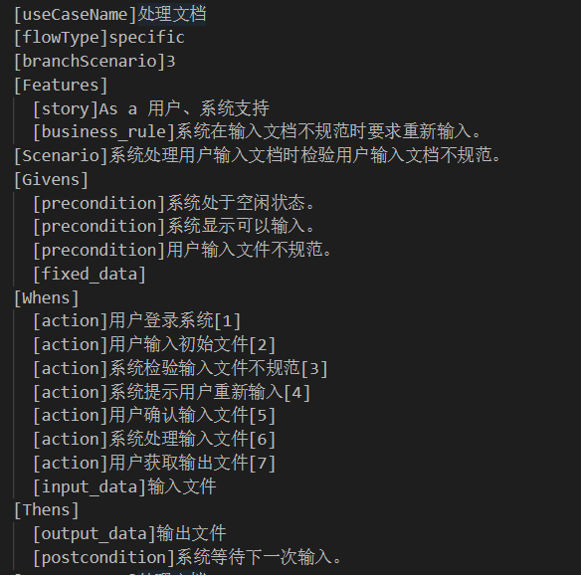


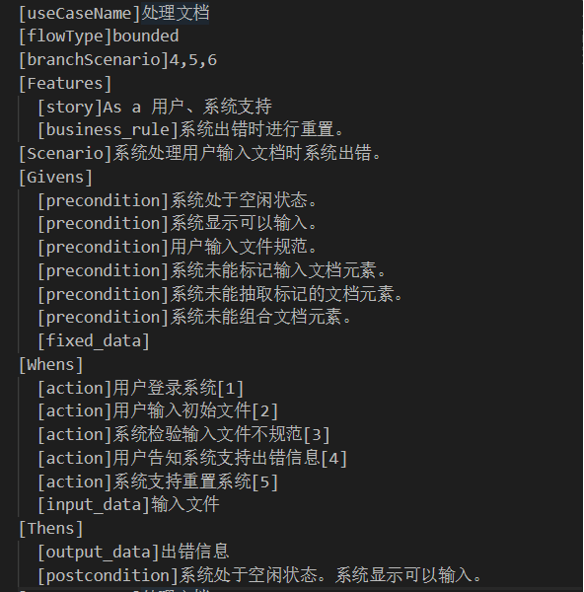


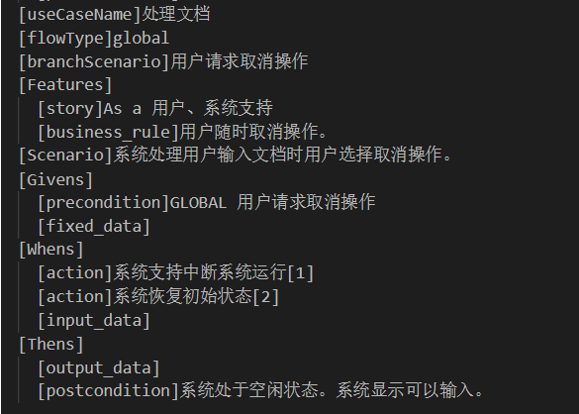


经过添加标签后可以表示为







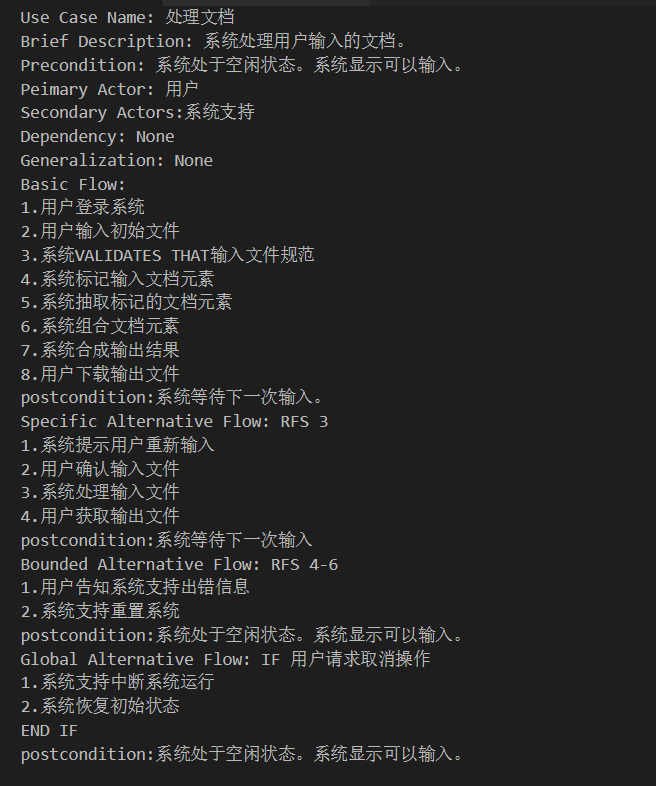


#### 由中间结果转化为RUCM

在添加中间标签后获得一组TaggedGWT对象，在此基础上，按照RUCM的字段依次完善生成完整的RUCM描述。

对Use Case Name，提取GWT中的useCaseName属性内容；对Brief Description，将useCaseName相同的一组GWT的Scenario属性内容合并成一个文本，通过自然语言处理工具进行摘要生成，将摘要作为Brief Description；对Precondition，依次比较各GWT的precondition句子，将所有GWT共有的precondition句子作为RUCM的Precondition；对flowType标记为“basic”的GWT，对其“story”句子使用正则表达式提取出参与者序列，将首个参与者作为Peimary Actor，其余参与者组成Secondary Actors；对Dependency，使用正则表达式匹配GWT中含有关键词INCLUDE和EXTENDED的“precondition”句子和“action”句子，提取出关键词标记的用例名称，组成RUCM的Dependency字段；对Generalization，使用正则表达式匹配GWT中含有关键词GENERALIZE的“precondition”句子，提取出关键词标记的用例名称，组成RUCM的Generalization字段；对Basic Flow，提取flowType为“basic”的GWT的“action”句子，按照“sequence”顺序组成Basic Flow的事件流，此GWT的postcondition句子作为Basic Flow的postcondition；对Specific Alternative Flow，提取flowType为“specific”的GWT的“action”句子，按照“sequence”顺序组成Specific Alternative Flow的事件流，此GWT的postcondition句子作为Specific Alternative Flow的postcondition，将此GWT的Branch Scenario保存的sequence记录为此Specific Alternative Flow的RFS，并按照此sequence修改Basic Flow中的流事件，将该句分词并标注后以VALIDATES THAT关键词替换主语的动词；对Bounded Alternative Flow，提取flowType为“bounded”的GWT的“action”句子，按照“sequence”顺序组成Bounded Alternative Flow的事件流，此GWT的postcondition句子作为Bounded Alternative Flow的postcondition，将此GWT的BranchScenario保存的一组sequence记录为此Bounded Alternative Flow的一组RFS；对Global Alternative Flow，提取flowType为“global”的GWT的“action”句子，按照“sequence”顺序组成Global Alternative Flow的事件流，将此GWT的BranchScenario保存的条件句子作为Bounded Alternative Flow的IF条件，以END IF结尾，将此GWT的postcondition句子作为Global Alternative Flow的postcondition，从而形成完整的RUCM描述。

按照上述规则对中间结果进行组合，可以获取如下的RUCM输出



## 4、界面展示及数据库设计说明

### (1)界面展示



Browse：浏览本地文件

导入：将选择的文件传输至控制层

开始转换：触发控制层中的translation

### (2)数据调度器

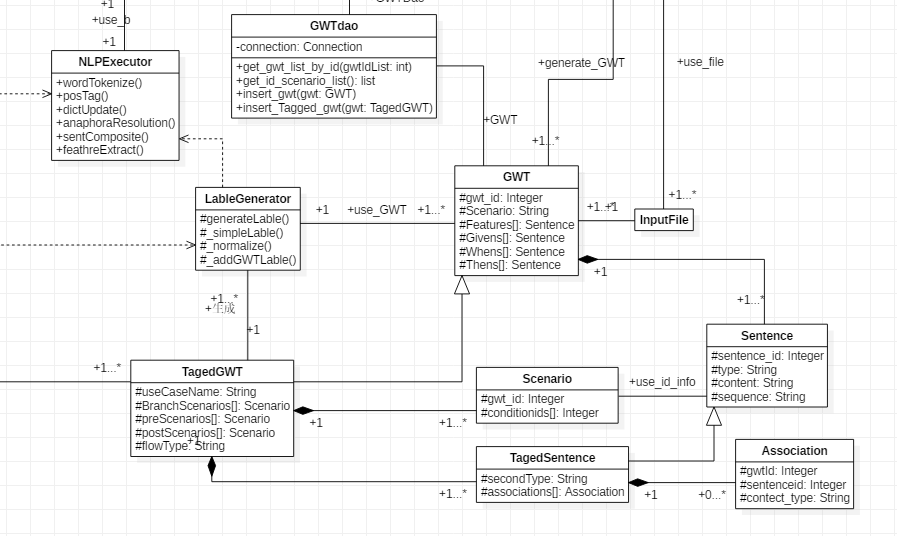
分为两个大的部分，分别是对领域背景信息的调度，以及对GWT的调度。两个类来进行实现。一个是backgroundDao，用于调度领域背景信息，提供三种接口：

1. getContentList：依据领域背景信息要求，提取数据库中相同领域背景标识的所有领域背景数据。
2. insert：根据领域背景标识插入当前背景的所有领域背景，若当前领域背景标识已经存在则不能进行插入，并返回错误信息。
3. update：根据领域背景标识，对当前背景所有领域背景进行修改。
4. 另一个为GWTdao，提供以下四种接口：
5. get\_gwt\_list\_by\_id：通过提供的id列表获取数据库中已经存储的GWT，需要提供的id都是数据库中已经存在的。
6. get\_id\_scenario\_list：获取数据库中所有GWT的id和scenario
7. insert\_gwt：插入一个全新的GWT，需要GWT的scenario在数据库中未出现过
8. insert\_Tagged\_gwt：插入一个已经放入数据库中的GWT，将该GWT加标签的部分放入数据库中。

### (3)数据库设计说明

本部分将内存中数据的存储与硬盘中数据的存储联系起来，将短期存储转化为长期存储，从而提高转化效率，不需要每次转化时都将所有的GWT进行导入重新进行存储和完全的标签化。其中，RUCM是要求生成的文档，其随着GWT组合的不同而不同，会经常发生变化，所以不在数据库中进行存储。而领域背景信息可以以文件方式进行存储不需要使用数据库可以方便的进行使用。所以，目前我们仅设计了GWT的数据库存储方案。

下面是数据库生成代码与GWT相关类图：



CREATE TABLE IF NOT EXISTS GWT\_tb(

id INT UNSIGNED AUTO\_INCREMENT,

scenario text NOT NULL,

useCaseName tinytext null,

flowType tinytext null,

PRIMARY KEY (id)

);

create table if not exists Sentence(

    id int unsigned auto\_increment,

Stype tinytext null,

secondType tinytext null,

content text not null,

sequence tinyint unsigned,

primary key (id)

);

-- engine=innodb default charset = utf8mb4

create table if not exists SentenceSentence(

    s1 int unsigned not null,

s2 int unsigned not null,

connectType tinytext not null

);

create table if not exists gwtSentence(

    gwt\_id int unsigned not null,

sentence\_id int unsigned not null,

position\_in\_gwt tinytext not null

);

我们将GWT中最小的单元视为一个个句子，每个句子都有自己的标签，同时GWT也有自己和其他GWT关系从而形成的标签。

下面，我们将按照表的视角对数据库的设计进行表述：

* GWT\_tb
  + Id：设计为不可重复的int类型，是GWT的独一标识方便数据库的检索
  + Scenario：设计为text类型，用来存储字符串。理论上也需要保持独一性，但是在数据库中不进行该检查。
  + useCaseName：加标签后，其归属的Use Case应已经规定下来，这里便是存储RUCM用例名称的位置
  + flowType：表明该GWT在RUCM中所在的位置。
  + 优化方向：一个GWT描述可能可以用在多个方向，从而节省存储，所以这里的useCaseName与flowType都可以放在另外的表中，与独一的GWT建立链接。
* Sentence
  + Id：Sentence的独一标识
  + Stype：input、action等
  + secondType：其他的一些标签，当前暂未定义，是为未来准备的冗余
  + sequence：action在GWT中的顺序
* SentenceSentence：表述句子与句子关系的表
  + S1：句子1的id
  + S2：句子2的id
  + connectType：句子关系的类型，如“相同”、“相反”等
* gwtSentence：描述GWT由哪些句子组成
  + gwt\_id：
  + sentence\_id:
  + position\_in\_gwt: Given、When、Then等
  + 优化方向：Stype、secondType、sesequence应从Sentence移动到gwtSentence中，从而将GWT和Sentence完全解耦合从而将内容相同的sentence删除，从而减少了相同句子的存储