**MUnit Test Generator**

Software Design Document

Created By: Benjamin East

Last Updated: 7/1/2017

**Contents**

[1. Introduction 3](#_Toc486618389)

[1.1 Purpose 3](#_Toc486618390)

[1.2 Scope 3](#_Toc486618391)

[1.3 Overview 3](#_Toc486618392)

[1.4 Reference Material 4](#_Toc486618393)

[1.5 Definitions and Abbreviations 4](#_Toc486618394)

[2. System Overview 4](#_Toc486618395)

[3. System Architecture 4](#_Toc486618396)

[3.1 Architectural Design 4](#_Toc486618397)

[3.2 Design Decomposition 6](#_Toc486618398)

[3.3 Design Rationale 8](#_Toc486618400)

[4. Data Design 8](#_Toc486618401)

[4.1 Data Description 8](#_Toc486618402)

[4.2 Data Dictionary 9](#_Toc486618403)

[5. Component Design 10](#_Toc486618404)

[6. User Interface Design 13](#_Toc486618405)

[6.1 Overview of User Interface 13](#_Toc486618406)

[6.2 Screen Objects and Actions 13](#_Toc486618407)

[7. Appendices 13](#_Toc486618408)

# **Introduction**

## **Purpose**

This software design document describes the architecture and system design of my MUnit Test Generator, which is to be implemented in Python.

## **Scope**

The MUnit Test Generator should be able to generate the framework of an MUnit Test Suite when given a functioning Mule XML file. The project should allow for instant generation of mostly complete MUnit tests, resulting in a significantly shorter test development time.

The tests may lack specific input or output values, or may not check values against certain criteria. The user must specify the expected input or output values, as well as criteria when necessary.

## **Overview**

The document will provide information pertaining to the functionality, design, and context of the MUnit Generator.

System Architecture and Component Design both contain information pertaining to class structure and code design.

Data Design contains information pertaining to data types expected throughout the project.

Human Interface Design contains information pertaining to the use of the software and the overall design of the user interface.

Requirements details the required functions of the completed application.

## **Reference Material**

## **Definitions and Abbreviations**

MUnit: A unit test designed for MuleSoft projects, which is focused around mocking external dependencies of code to allow clean, isolated testing of the internals of Mule files.

# **System Overview**

This project originated from a need at my summer software development internship. The company was transitioning services to MuleSoft and was looking for a way to test code, ideally without wasting time backtracking to reorganize flows to allow for easy MUnit Test design. I decided to attempt to automate the process, by parsing the Mule XML code and writing equivalent MUnit XML blocks to a specified output file.

The project should able to parse a Mule XML file from a user specified location, and outputting an equivalent MUnit Test Suite to a user specified output directory.

# **System Architecture**

## **Architectural Design**

The software should be split into several modules to isolate responsibilities and allow for easier orchestration of project tasks. For this project, I want to create data structures to simplify the management of Mule XML parsing. To do this, I will need to create a TagPair object which will represent and XML tag paired with an ordered dictionary containing the XML attributes for the tag. I will also need to create a TagList data structure which will be a meta-class of a standard list. Each TagList will contain TagPair objects and provide methods to allow for simple access to the data.

Most of the work will take place in a MuleLines object, which will parse the lines of XML code and store them as one of its attributes. The MuleLines object will generate a TagList of the original XML, and will use the TagList to generate a corresponding TagList of MUnit XML code.

The orchestration of the main process for the program will take place in an MUnit Generator file which will handle command line argument parsing as well the creation of necessary objects and the calling of various methods.

Output File Path

Input File Path

Write File

Read File

MUnit Generator

Pass input and output file paths to MuleLines object

MuleLines

Read and Write to TagLists

MuleLines has multiple TagLists

TagList

TagList



TagPair

TagPair

## **Design Decomposition**

**MUnit Generator**

Required Attributes:

* parser: ArgumentParser
* mule: MuleLines

Required Methods:

* main() -> None

**MuleLines**

Required Attributes:

* inputFileName: string
* muleFileLines: string list
* muleTagList: TagList
* mUnitTagList: TagList

Required Methods:

* parseMuleFileLines(inputFilePath : string) -> None
* createMUnitTests() -> None
* createMUnitSuiteFile(outputFilePath : string) -> None
* convertMuleToMUnit(TagList) -> TagList

**TagList**

Required Attributes:

* list: TagPair list

Required Methods:

* append(pair: TagPair) -> None
* clear() -> None
* contains(targetPair: TagPair) -> boolean
* copy() -> TagList
* pairs() -> TagPair list
* remove(targetPair: TagPair) -> None

**TagPair**

Required Attributes:

* tag: string
* attributes: dictionary (string: string)

Required Methods:

* getAttributes() -> dictionary
* setAttributes(newAttributes: dictionary) -> None
* getAttribute(attributeName: string) -> string
* setAttribute(attributeName: string,

attributeValue: string) -> None

* removeAttribute(attributeName: string) -> None

## **Design Rationale**

The choice to create TagPair and TagList objects was led by a need to maintain an ordered list of XML tags, while tracking the tag name and its corresponding attributes.

The TagPair seemed to be a natural structure to create, as it allows for effective mapping of XML tags to their attributes. A list of TagPairs is preferable to a dictionary alternative, as it allows for repetition of XML tags, which is to be expected in a Mule or MUnit XML document.

Although the TagList is like a standard list object in Python, it’s custom design allows for more granular interaction with the contents of the list without exposing the contents to the user. For instance, the TagList can be given the option to search for a tag, and not a TagPair, allowing an easy search by the end user.

I chose to isolate all file parsing, writing, and code generation to MuleLines, as it allows for easy reusability of the object if ever necessary. It wouldn’t make sense to allow multiple classes to handle the process of XML parsing and conversion, as the process would involve passing of relatively large amounts of data between several objects. As a result, I opted for a single object to orchestrate all processing of Mule code.

The MUnit Generator class is a simple class that orchestrates user interaction and creates and calls all necessary processes for the software. This allows for a simple, accessible interaction for the user, while all complicated details are handled behind the scenes.

# **Data Design**

## **Data Description**

Data in this project is read in from a specified input file, which is broken down into lines and parsed. Each line is split to isolate the XML tag and the corresponding attributes of the tag. A TagPair is created, pairing the XML tag with a dictionary containing attribute : value pairs. Each of these TagPairs is then mapped to a TagList for future processing.

The original file lines are also stored within the system, in case future versions have a requirement to access the file twice. This will allow for a single access to the input file with a potential for multiple parsing iterations over the file structure.

All MUnit XML that is generated should be immediately mapped to a TagList for storage in the system. The MUnit TagList will eventually be iterated through and written to the user specified output file location.

## **Data Dictionary**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Type** | **Description** | **Usage** |
| TagPair | TagPair | Pair of (XML Tag, Attributes dictionary(string : string)) | Used to manage XML tags |
| TagList | List meta-class | List of TagPair objects, with additional methods to access TagPair data | Stores TagPairs throughout the system |
| MuleLines | MuleLines | Handles reading and writing of files and parsing of XML data. Stores TagLists of Mule and MUnit data as well as file lines | Creates TagLists and TagPairs for the system and manages all file data |
| muleFileLines | List | Stores the lines of the input mule file prior to parsing | Contained within a MuleLines object to store input file data |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

# **Component Design**

**MUnit Generator**

**Required Attributes:**

**Required Methods:**

main():

get input file path

get output file path

create MuleLines object

MuleLines.parseMuleFileLines(input file path)

MuleLines.createMUnitTests()

MuleLines.createMUnitSuiteFile(output file path)

**MuleLines**

**Required Attributes:**

* inputFileName: string
* muleFileLines: list of strings
* muleTagList: TagList
* mUnitTagList: TagList

**Required Methods:**

parseMuleFileLines(input file path: string):

store input file name

read input file

split file by lines

store lines into muleFileLines

for each line:

split tag and attributes

create TagPair

store TagPair in muleTagList

createMUnitTests():

store MUnit dependency tags in mUnitTagList

for each mule flow:

if flow contains choice block:

extract operations from choice blocks

replace choice blocks with a placeholder

for each choice operation:

generate an MUnit test flow

store the test flow in mUnitTagList

else:

generate an MUnit test flow

store the test flow in mUnitTagList

add /mule TagPair to mUnit TagList

createMUnitSuiteFile(output file path: string):

for each TagPair in mUnitTagList:

dynamically indent based on previous tags

write tag and attributes to file

convertMuleToMUnit():

for each mule line:

if the line has an MUnit equivalent:

add the MUnit to mUnitTagList

else:

skip the line

**TagList**

**Required Attributes:**

* list: list of TagPairs

**Required Methods:**

append(pair: TagPair):

append pair to list

clear():

for x in list:

remove x from list

contains(targetPair: TagPair):

for x in list:

if x == targetPair:

return True

return False

copy():

create new TagList

for x in list:

append x to new TagList

return new TagList

pairs():

create output list

for x in list:

add x to output list

return output list

remove(targetPair: TagPair):

if targetPair in list:

remove targetPair from list

else:

return

**TagPair**

**Required Attributes:**

* tag: string
* attributes: dictionary (string: string)

**Required Methods:**

getAttributes():

return attributes

setAttributes(newAttributes: dictionary):

attributes = newAttributes

getAttribute(attributeName: string):

if attributeName in attributes:

return attribute value from attributes

setAttribute(attributeName: string,

attributeValue: string):

add (attributeName: attributeValue) to attributes

removeAttribute(attributeName: string):

if attributeName in attributes:

remove attributeName from attributes

# **User Interface Design**

## **Overview of User Interface**

The user interface for this software will be a command line interface. The user will simply need to open command prompt and call the script as follows:

python MUnit\_Generator.py --input INPUT\_FILE\_PATH --output OUTPUT\_FILE\_PATH

The script will check that the input file path is valid and proceed from that point.

If the input file cannot be found, an exception will be thrown and the user will be notified that the file path is invalid.

If the output file path does not exist, it will be automatically created.

## **6.2 Screen Objects and Actions**

None, as the user interface is CLI based for this project.

# **Appendices**