**Object-Oriented Design Document for MTT Client Configuration File Editor**

**Part 2: Detailed Design**

**Prepared for Master of Software Engineering Capstone Project**

**Advisor: Joshua Hursey**

**Student: Han Chen**

**Date: May 11, 2015**

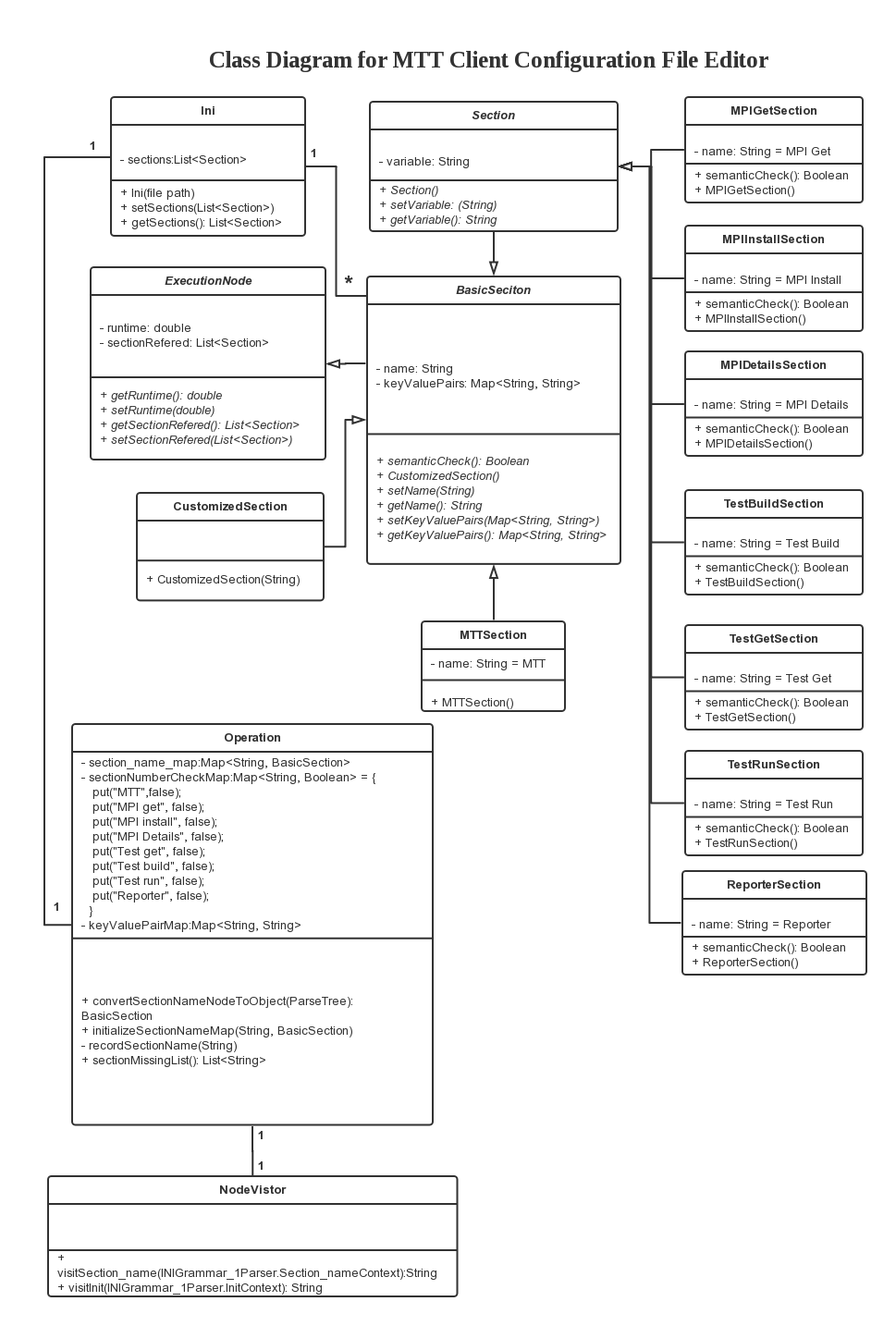
**1. About this document**

This document describes a detailed object-oriented design for MTT client configuration file editor. The architectural design for the editor is given in [3] and the requirements are given in [1,2]. The object-oriented design is described using a collection of class definitions where each class definition includes structural and behavioral properties. Since there is no GUI requirement, this document describes only those classes corresponding to the functional behavior and do not describe the classes corresponding to the graphical user interface.

**2. Design Decisions**

1. The configuration file type must be INI file (correct file extension).
2. The configuration file must satisfy the most basic INI file syntax requirement (i.e. correct comment symbol, correct section syntax, and so on).
3. The configuration file must contain all required default sections (i.e. MTT, MPI get, and so on)
4. In key-value pairs, the value of pair might be the name of other section that is called in this pair. Therefore the software should be able to connect different sections based on calling relationship.
5. The funclets are sequence of function names that are already defined. Each funclet has its own estimated runtime. Software should be able to dedicate those funclets occurring among the INI file and querying their runtimes from database to compute the total estimated runtimes.
6. User should be able to input command after software validated the configuration file in order to tune parameters in section. Therefore user should be able to use command to choose section first, then choosing parameter, and then input value. Finally the software would show the new estimated result to user. The validation of value that user input must be implemented based on different syntaxes of parameters.

**3. Class Diagram**

****

**4. Format of a Class Definition**

Each class definition is given in the following format:

***Class name***: must be unique within the entire design.

***Attributes or instance variables***:

each variable is given in following format:

<visibility> <type> <name>

where <visibility> is “public” or “private”.

**Unless otherwise indicated, there will be a getX() method and a setX() method  
for each private attribute X.**

**Methods**

Each method will be classified as “public” or “private”.

Each method is given in the following structural format:

*Name* of the method – must be unique within the class  
*Synopsis* of the method – calling syntax for the method  
*Purpose* of the method – a short description of the functionality implemented by this method  
*Visibility* – public or private  
*Input parameters* – a set of parameters in <type><name> format  
*Output parameter* – in <type><name> format

*Local variables* – a set of variables used in describing the pseudo code

(given next) for this method

*Pseudocode* – an algorithmic (structured) description of the method

*Exceptions* – a set of exceptions that might arise in executing this method and

their corresponding corrective actions

*Remarks* – additional information about this method and hints for the

programmers; typically, it may include the design decisions taken and choices of implementation that the programmer may consider

**5. Format of a Class Definition**

|  |
| --- |
| **Classname**: Ini  **Attributes**:  private List<Section> sectionsList |

**Methods**

Name: Ini

Synopsis: ini ← Ini(filePath)

Purpose: To Initialize an instance of editor and try to parse a file

Visibility: public

Input parameters: String filePath

Output parameter: Ini ini

Local variables: None

Pseudocode:

FileStream = inputStream(filePath)

if (checkFileType(FileStream))

Ini ini = splitFile(FileStream);

if(checkSyntax(ini))

if(checkSemantic(ini))

ini.displayStructureTree();

displayNavigation();

else reportSemanticError();

else reportSyntaxError();

else retportFileTypeError();

Exceptions: None

Remarks None

|  |
| --- |
| **Classname**: BasicSection  **Attributes**:  private String name  private String variable  private Map<String, String> keyValuePairs |

**Methods**

Name: semanticCheck

Synopsis: result ← semanticCheck (String key, String value)

Purpose: Abstract method for checking the semantic correctness

Visibility: abstract

Input parameters: String key, String value

Output parameter: Boolean result

Local variables: None

Pseudocode: None

Exceptions: None

Remarks Subclasses will define their own sematicCheck method based on their own different requirments.

|  |
| --- |
| **Classname**: CustomizedSection  **Attributes**: /\* This is a subclass extending from BasicSection class \*/ |

**Methods**

Name: semanticCheck

Synopsis: result ← semanticCheck (String key, String value)

Purpose: To check the semantic correctness of this object

Visibility: public

Input parameters: String key, String value

Output parameter: Boolean result

Local variables: Boolean result

Pseudocode:

Return true; /\* there is no semantic checking requirement for this object so far\*/

Exceptions: None

Remarks: None

|  |
| --- |
| **Classname**: ExecutionNode  **Attributes**:  private double runtime  private List<Section> sectionReferred /\*Storing the referred section by current section that extends this abstract class\*/ |

**Methods**

|  |
| --- |
| **Classname**: MPIDetailsSection  **Attributes**: /\* This is a subclass extending from BasicSection class \*/ |

**Methods**

Name: semanticCheck

Synopsis: result ← semanticCheck (String key, String value)

Purpose: To check semantic correctness of each object

Visibility: public

Input parameters: String key, String value

Output parameter: Boolean result

Local variables: Boolean result

Pseudocode:

/\* MPIDetailSection object doesn’t have specific requirement of semantic checking except that it must contain keyValuePair which key is “exec”. \*/

Boolean result;

if(!this.keyValuePairs.keyset.contains(“exec”)

result = false;

Return result;

else

result = true;

return result;

Exceptions: None

Remarks None

|  |
| --- |
| **Classname**: MPIGetSection  **Attributes**: /\* This is a subclass extending from BasicSection class \*/ |

**Methods**

Name: semanticCheck

Synopsis: result ← semanticCheck (String key, String value)

Purpose: To check semantic correctness of each object

Visibility: public

Input parameters: String key, String value

Output parameter: Boolean result

Local variables: Boolean result

Pseudocode:

/\* this method will be called after visitor encapsulate all keyValuePairs to objects. The main class will use corresponding required keyValuePairs as input parameter to check whether the object is matching the requirement or not. \*/

Boolean result;

if(this.keyValuePairs.keyset.contains(key)

if(this.keyValuePairs.get(key).equals(value))

result = true;

Return result;

else

result = false;

return result;

else

result = false;

return result;

Exceptions: None

Remarks None

|  |
| --- |
| **Classname**: MPIInstallSection  **Attributes**: /\* This is a subclass extending from BasicSection class \*/ |

**Methods**

Name: semanticCheck

Synopsis: result ← semanticCheck (String key, String value)

Purpose: To check semantic correctness of each object

Visibility: public

Input parameters: String key, String value

Output parameter: Boolean result

Local variables: Boolean result

Pseudocode:

/\* this method will be called after visitor encapsulate all keyValuePairs to objects. The main class will use corresponding required keyValuePairs as input parameter to check whether the object is matching the requirement or not. \*/

Boolean result;

if(this.keyValuePairs.keyset.contains(key)

if(this.keyValuePairs.get(key).equals(value))

result = true;

Return result;

else

result = false;

return result;

else

result = false;

return result;

Exceptions: None

Remarks None

|  |
| --- |
| **Classname**: ReporterSection  **Attributes**: /\* This is a subclass extending from BasicSection class \*/ |

**Methods**

Name: semanticCheck

Synopsis: result ← semanticCheck (String key, String value)

Purpose: To check semantic correctness of each object

Visibility: public

Input parameters: String key, String value

Output parameter: Boolean result

Local variables: Boolean result

Pseudocode:

/\* this method will be called after visitor encapsulate all keyValuePairs to objects. The main class will use corresponding required keyValuePairs as input parameter to check whether the object is matching the requirement or not. \*/

Boolean result;

if(this.keyValuePairs.keyset.contains(key)

if(this.keyValuePairs.get(key).equals(value))

result = true;

Return result;

else

result = false;

return result;

else

result = false;

return result;

Exceptions: None

Remarks None

|  |
| --- |
| **Classname**: TestBuildSection  **Attributes**: /\* This is a subclass extending from BasicSection class \*/ |

**Methods**

Name: semanticCheck

Synopsis: result ← semanticCheck (String key, String value)

Purpose: To check semantic correctness of each object

Visibility: public

Input parameters: String key, String value

Output parameter: Boolean result

Local variables: Boolean result

Pseudocode:

/\* this method will be called after visitor encapsulate all keyValuePairs to objects. The main class will use corresponding required keyValuePairs as input parameter to check whether the object is matching the requirement or not. \*/

Boolean result;

if(this.keyValuePairs.keyset.contains(key)

if(this.keyValuePairs.get(key).equals(value))

result = true;

Return result;

else

result = false;

return result;

else

result = false;

return result;

Exceptions: None

Remarks None

|  |
| --- |
| **Classname**: TestGetSection  **Attributes**: /\* This is a subclass extending from BasicSection class \*/ |

**Methods**

Name: semanticCheck

Synopsis: result ← semanticCheck (String key, String value)

Purpose: To check semantic correctness of each object

Visibility: public

Input parameters: String key, String value

Output parameter: Boolean result

Local variables: Boolean result

Pseudocode:

/\* this method will be called after visitor encapsulate all keyValuePairs to objects. The main class will use corresponding required keyValuePairs as input parameter to check whether the object is matching the requirement or not. \*/

Boolean result;

if(this.keyValuePairs.keyset.contains(key)

if(this.keyValuePairs.get(key).equals(value))

result = true;

Return result;

else

result = false;

return result;

else

result = false;

return result;

Exceptions: None

Remarks None

|  |
| --- |
| **Classname**: TestRunSection  **Attributes**: /\* This is a subclass extending from BasicSection class \*/ |

**Methods**

Name: semanticCheck

Synopsis: result ← semanticCheck (String key, String value)

Purpose: To check semantic correctness of each object

Visibility: public

Input parameters: String key, String value

Output parameter: Boolean result

Local variables: Boolean result

Pseudocode:

/\* this method will be called after visitor encapsulate all keyValuePairs to objects. The main class will use corresponding required keyValuePairs as input parameter to check whether the object is matching the requirement or not. \*/

Boolean result;

if(this.keyValuePairs.keyset.contains(key)

if(this.keyValuePairs.get(key).equals(value))

result = true;

Return result;

else

result = false;

return result;

else

result = false;

return result;

Exceptions: None

Remarks None

|  |
| --- |
| **Classname**: NodeVisitor  **Attributes**:  Private Operation op;  Private List<BasicSection> basicSectionList;  Private Map<BasicSection, List<KeyValuePairs> sectionKVPMap;  /\*\* There are no setX() and getX() methods for the two private variables listed above. \*\*/ |

**Methods**

Name: visitSection\_name

Synopsis: result ← visitSection\_name(INIGrammar\_1Parser.Section\_nameContext ctx)

Purpose: To traverse all section name nodes and encapsulate them to objects.

Visibility: public

Input parameters: INIGrammar\_1Parser.Section\_nameContext ctx

Output parameter: String result

Local variables: String result

Pseudocode:

/\* This is a overridden method, which will traverse all section name nodes and encapsulate them to objects.\*/

String result = “”;

this.basicSections.add(op.convertSectionNameNodeToObject(ctx));

return result;

Exceptions: None

Remarks None

--------\*--------------\*--------------\*------------

Name: visitInit

Synopsis: result ← visitInit(INIGrammar\_1Parser.InitContext ctx)

Purpose: To traverse all section nodes and encapsulate them to objects. Therefore to build the relationship among keyValuePairs and the sections they belong to.

Visibility: public

Input parameters: INIGrammar\_1Parser.Section\_nameContext ctx

Output parameter: String result

Local variables: String result

Pseudocode:

/\* This is a overridden method, which will traverse all section nodes and encapsulate them to objects.\*/

String result = “”;

List<KeyValuePair> kvplist = new List<KeyValuePair> ();

kvplist = op.convertKeyValuePairToObject(ctx.section.content);

sectionKVPMap.put(op.convertSectionNameNodeToObject(ctx.section.section\_name), kvplist);

return result;

Exceptions: None

Remarks None

|  |
| --- |
| **Classname**: Operation  **Attributes**:  private Map<String, BasicSection> section\_name\_map;  private Map<String, Boolean> sectionNumberCheckMap;  private Map<String, String> keyValuePairMap;  /\*\* There are no setX() and getX() methods for the two private variables listed above. \*\*/ |

**Methods**

Name: convertSectionNameNodeToObject

Synopsis: basicSection ← convertSectionNameNodeToObject (ParseTree node)

Purpose: To encapsulate parse tree nodes to objects.

Visibility: public

Input parameters: ParseTree node

Output parameter: BasicSection basicSection

Local variables: None

Pseudocode:

/\* This method will encapsulate parse tree nodes to objects. \*/

String sectionNameString = node.getChild(1).getChild(0).getText();

if(node.getChildCount() == 3){

switch (sectionNameString) {

case "MTT":

MTTSection newMttSection = new MTTSection();

initializeSectionNameMap(sectionNameString, newMttSection);

recordSectionName(sectionNameString);

return(newMttSection);

default:

CustomizedSection newCustomizedSection = new CustomizedSection(sectionNameString);

initializeSectionNameMap(sectionNameString, newCustomizedSection);

return(newCustomizedSection);

}

}

else if(node.getChildCount() == 5){

String sectionVariableString = node.getChild(3).getText();

switch(sectionNameString){

case "MPI get":

MPIGetSection newMpiGetSection = new MPIGetSection(sectionVariableString);

initializeSectionNameMap(sectionNameString+":"+sectionVariableString, newMpiGetSection);

recordSectionName(sectionNameString);

return(newMpiGetSection);

case "MPI install":

MPIInstallSection newInstallSection = new MPIInstallSection(sectionVariableString);

initializeSectionNameMap(sectionNameString+":"+sectionVariableString, newInstallSection);

recordSectionName(sectionNameString);

return(newInstallSection);

case "MPI Details":

MPIDetailsSection newDetailsSection = new MPIDetailsSection(sectionVariableString);

initializeSectionNameMap(sectionNameString+":"+sectionVariableString, newDetailsSection);

recordSectionName(sectionNameString);

return(newDetailsSection);

case "Test get":

TestGetSection newGetSection = new TestGetSection(sectionVariableString); initializeSectionNameMap(sectionNameString+":"+sectionVariableString, newGetSection);

recordSectionName(sectionNameString);

return(newGetSection);

case "Test build":

TestBuildSection newBuildSection = new TestBuildSection(sectionVariableString); initializeSectionNameMap(sectionNameString+":"+sectionVariableString, newBuildSection);

recordSectionName(sectionNameString);

return(newBuildSection);

case "Test run":

TestRunSection newTestRunSection = new TestRunSection(sectionVariableString); initializeSectionNameMap(sectionNameString+":"+sectionVariableString, newTestRunSection);

recordSectionName(sectionNameString);

return(newTestRunSection);

case "Reporter":

ReporterSection newReporterSection = new ReporterSection();initializeSectionNameMap(sectionNameString+":"+sectionVariableString, newReporterSection);

recordSectionName(sectionNameString);

return(newReporterSection);

default:

return null;

}

}

else return null;

Exceptions: None

Remarks None

--------\*--------------\*--------------\*------------

Name: initializeSectionNameMap

Synopsis: initializeSectionNameMap (String sectionNameString, BasicSection sectionObject)

Purpose: To build a dictionary of a section with its corresponding section object.

Visibility: public

Input parameters: String sectionNameString, BasicSection sectionObject Output parameter: None

Local variables: None

Pseudocode:

this.section\_name\_map.put(sectionNameString, sectionObject); Exceptions: None

Remarks None

--------\*--------------\*--------------\*------------

Name: sectionMissingList

Synopsis: missingSectionList ← sectionMissingList ()

Purpose: To store missed sections list.

Visibility: public

Input parameters: None

Output parameter: None

Local variables: List<String> outputList

Pseudocode:

/\* check the sectionNumberCheckMap map to get the missing section, and encapsulating them to a list. If there was no one of required sections missing, the list will be empty. \*/

List<String> outputList = new ArrayList<String>();

for (String o : this.sectionNumberCheckMap.keySet()) {

if (this.sectionNumberCheckMap.get(o).equals(false)) {

outputList.add(o);

}

}

return outputList;

Exceptions: None

Remarks None

**References**

1. Han Chen, “Software Requirements Document for MTT Client Configuration File Editor – Part 1: Product Overview and Assumptions”, May 2015.
2. Han Chen, “Software Requirements Document for MTT Client Configuration File Editor – Part 2: Functional Requirement”, May 2015.
3. Han Chen, “Object-Oriented Design Document for MTT Client Configuration File Editor – Part 1: Architectural Design”, May 2015.