Stub Class Generator 1.0 Component Specification

1. Design

The Stub Class Generator component is able to generate class stubs for Java and C#, from the classes represented in UML Model.

This component provides supporting almost all standard of languages. It does not support “throws” in methods, because it is not useful feature. Actually it may be easy implemented if it is needed. But as for me I there are not reasons to implement this. For example if you have class with several method which throw some exception you should create dependency with stereotype “throw” to each of this methods. I think nobody want to do it.

Also component provides ability to log errors in time of creation code. Note that component will log only serious code problems which easy notify. It does provide neither full code validation nor style checking.

1.1 Design Patterns

Factory pattern is used for setting different instances of Generator (possible two – JavaGenerator and CSharpGenerator).

Strategy pattern is used for choosing proper implementation of Generator.

Façade pattern is used in CodeGenerator class. This class represent encapsulate all functionality of component in four its methods.

1.2 Industry Standards

UML 1.5

1.3 Required Algorithms

Component makes many code examples how to implement its methods. This part of specification description which Model element represent Java and C# languages key words and code structures.

Getting documentation

General documentation. It represents main documentation for some elements. For each element it is possible one such documentation node. This is saved in TaggedValue(attribute dataValue) which contain TagDefinition with tagType “documentation”. Note that “documentation” is configurable and such attributes exist in AbstractGenerator. Little code example (for ModelElement):

```
String getGeneralDocs(modelElement){
```

```
for (TaggedValue temp: modelElement.getTaggedValues()){

        if(temp.getType().getName().equals("documentation")){

        //We found documentation node. Just return it

        return temp.getDataValue();
        }

        }

        //We do not found documentation node. Just return null

        return null;

}
```

Tags documentation. The component provides the ability for users to create own tags, like "throw", "exception" or even "hello world". Tags documentation is saved in TaggedValue(attribute dataValue) which contain TagDefinition with tagType "documentation#" + tagname. Note that "documentation#" is configurable and such attributes (tagsDocumentation) exist in AbstractGenerator. Little code example (for ModelElement):

```
Map getTagDocs(modelElement)
      Map<String, List<String>> result = new HashMap<String, List<String>>();

      for (TaggedValue value : modelElement.getTaggedValues()) {
            TagDefinition tagDefinition = value.getType();

            if ((tagDefinition != null)
            && (tagDefinition.getTagType() != null)
            && tagDefinition.getTagType().startsWith(this.tagsDocumentation)) {

            String key
            =tagDefinition.getTagType().substring(this.tagsDocumentation.length(
      ));

            if (key.trim().length() == 0) {
                // ignore empty keys
                continue;
            }

            if (result.containsKey(key)) {
                // tag has been found before, add new value to its list
                result.get(key).add(value.getDataValue());
            } else {
                // tag has not been found before, create a new list
                List<String> list = new ArrayList<String>();
                list.add(value.getDataValue());
                result.put(key, list);
            }
         }
    }

    return result;
}
```

Parameter documentation. It represents documentation of methods parameter and used only for Operation instances. Actions are very similar as in previous

description. You just need list Parameter instances and for each of them run getGeneralDocs(), result write to map as it getTagDocs() made and return this map.

Parameters list of method represented by parameters attribute of BehaviourFeature.

Java code generating

- **Documentation generation**
  How to get documentation described above. For now it should be formatted in Java style. It should like this - /** + retrieve all documentation + */. Parameters and tags should be written in format – "@tagname tag documentation" or "@param parameter documentation". Simple example of formatting documentation:

  /**
  General documentation
  @tagname tag documentation
  @param param documentation
  */

  Note that for it may be many different tags and parameters, may be several tags with the same name and parameters should exist only for methods

- **Classifier header generating**
  Generation Class. Class header may represent in the next format:
  "private final abstract class Test extends Class1 implements Interface1". That is all which is possible for Class.
  Algorithm:
  1. Create String result where you will write class header
  2. If visibility attribute equals VisibilityKind.PUBLIC, VisibilityKind.PRIVATE or VisibilityKind.PROTECTED just add according modifier, if VisibilityKind.PACKAGE do not add something
  3. If isLeaf attribute true, just add to result "final", false do not add something.
  4. If isAbstract attribute true, just add to result "abstract", false do not add something.
  5. Add "class" + name of Class to result
  6. For now you need get list of classes from which extends current class or interfaces which it implements. Create two lists – classExtends and interfaceImplements . Both may be found in attribute generalizations. You need just for each Generalization instance in given list check parent attribute. If it instance of Class simply add it to classExtends, if it instance of Interface add it to interfaceImplements, if neither Class nor Interface just ignore it. And now just add like this: "extends" + each Class name from classExtends + "implements" + each Interface form interfaceImplements.

Interface generating. Creating header for Interface is very similar to Class one. But interface can not be final and can not implement other interface. Also it should be "interface" + Interface name. The rest is the same

Enumeration generating. It is the simplest procedure. It has only visibility, which should be created in the same manner as for Class. Also it should be "enum" + Enumeration name.

- **Attribute code generating**
  The full examples of attribute features are the next:
  public final static String str = "hello"
  Algorithm:
  1. Create String result where you will write attribute code
  2. If visibility attribute equals VisibilityKind.PUBLIC, VisibilityKind.PRIVATE or VisibilityKind.PROTECTED just add according modifier, if VisibilityKind.PACKAGE do not add something
  3. If changeability attribute equals ChangeableKind.FROZEN add "final" to result , otherwise do not add something
  4. If ownerScope equals ScopeKind.CLASSIFIER add "static" to result, otherwise do not add something
  5. Add to result type.getName(). Note that attribute type is Classifier
  6. Add name of attribute by using getName() method
  7. If initialValue not null add "=" plus initialValue.getBody()

- **Method code generating**
  Generation methods are most harmful action. Possible method features:
  "public abstract final static synchronized void test(String s)"
  Algorithm:
  1. Create String result where you will write method code
  2. If visibility attribute equals VisibilityKind.PUBLIC, VisibilityKind.PRIVATE or VisibilityKind.PROTECTED just add according modifier, if VisibilityKind.PACKAGE do not add something
  3. If isAbstract attribute true, just add to result "abstract", false do not add something.
  4. If isLeaf attribute true, just add to result "final", false do not add something.
  5. If changeability attribute equals ChangeableKind.FROZEN add "final" to result , otherwise do not add something
  6. If concurrency attribute equals CallConcurencyKind.GUARDED add "synchronized", otherwise do not add something
  7. Now it should be find return type. It should be found instance of Parameter from parameters attribute which kind equals ParameterDirectionKind.RETURN. If you find such Parameter instance simply add its classifier.getName() to result. Note that such parameter may not exist and in such case you should not add something (it is common practice for constructors).
  8. Add name of method to result
  9. For now you need add to result something like this "(" + parameters + ")". You retrieve a list of parameter and add to result in next format: parameter.getName() +":" + parameter.getClassifier().getName(). Note one

moment – you should check if kind of parameter is not equal ParameterDirectionKind.RETURN. We can add here return parameters.

10. Create method body, simply add "{};"

C# code generating

- **Documentation generating**
  How to get documentation described above. For now it should be formatted C#(each documentation line should start from"///") Parameters and tags should be written in format – "<tagname> tag documentation</tagname>" or "<param> parameter documentation</param>". Simple example of formatting documentation:

  ///General documentation
  ///<tagname>
  ///tag documentation
  ///</tagname>
  ///<param>
  /// param documentation
  ///</param>

  Note that for it may be many different tags and parameters, may be several tags with the same name and parameters should exist only for methods

- **Class header generating**
  Class header may represent in the next format:
  "private abstract class Test : Interface1, Class1". That is all which is possible for Class.
  Algorithm:
  1. Create String result where you will write class header
  2. If visibility attribute equals VisibilityKind.PUBLIC, VisibilityKind.PRIVATE or VisibilityKind.PROTECTED just add according modifier, if VisibilityKind.PACKAGE do not add something
  3. If isAbstract attribute true, just add to result "abstract", false do not add something.
  4. Add "class" + name of Class to result. Or if classifier contain "struct" attribute add "struct" + name of Class to result.
  5. It is similar as in Java version, but of course you should not use "extends" and "implements". Use ":" instead. Also you may implement only one list but I strongly suggest make the same as in Java version. It gives you ability to check for multiple inheritances.

  Interface generating. Creating header for Interface is very similar to Class one. But interface can not be const. Also it should be "interface" + Interface name. The rest is the same

  Enumeration generating. It is the simplest procedure. It has only visibility, which should be created in the same manner as for Class. Also it should be "enum" + Enumeration name.

- **Attribute code generating**
  The full examples of attribute features are the next:
  “[non standard descriptor] public const static string str = "hello"
  Algorithm:
  1. Create String result where you will write attribute code
  2. It is possible three non standard descriptor – “new”, “readonly” and “protected internal”. You need use stereotypes Map from AbstractGenerator to retrieve their stereotype names. And check if such Attribute contains such stereotypes.
  3. If visibility attribute equals VisibilityKind.PUBLIC, VisibilityKind.PRIVATE or VisibilityKind.PROTECTED just add according modifier, if VisibilityKind.PACKAGE do not add something. Note : if attribute has stereotype with name “protected internal” you do not write some modifier!
  4. If changeability attribute equals ChangeableKind.FROZEN add “const” to result , otherwise do not add something
  5. If ownerScope equals ScopeKind.CLASSIFIER add “static” to result, otherwise do not add something
  6. Add to result type.getName(). Note that attribute type is Classifier
  7. Add name of attribute by using getName() method
  8. If initialValue not null add “=” plus initialValue.getBody()

- **Method code generating**
  Generation methods are most harmful action. Possible method features:
  “[Non standard descriptor] public abstract const static synchronized void test(Strings)”
  Algorithm:
  1. Create String result where you will write method code
  2. It is possible next non standard descriptor – “new”, “virtual “override”, “sealed”, “extern”, “explicit”, and “implicit”. You need use stereotypes Map from AbstractGenerator to retrieve their stereotype names. And check if such Attribute contains such stereotypes.
  3. If visibility attribute equals VisibilityKind.PUBLIC, VisibilityKind.PRIVATE or VisibilityKind.PROTECTED just add according modifier, if VisibilityKind.PACKAGE do not add something
  4. If isAbstract attribute true, just add to result “abstract”, false do not add something.
  5. If isLeaf attribute true, just add to result “const”, false do not add something.
  6. Now it should be find return type. It should be found instance of Parameter from parameters attribute which kind equals ParameterDirectionKind.RETURN. If you find such Parameter instance simply add its classifier.getName() to result. Note that such parameter may not exist and in such case you should not add something (it is common practice for constructors).
  7. Add name of method to result. Now it is one more interesting detail. Method may be destructor. It is parameter of configuration “destructor”. Using

stereotypes Map from AbstractGenerator to find it stereotype name. And check if such Stereotype instance is present in stereotypes of current attribute. If present add before method name – “~”

8. For now you need add to result something like this “(” + parameters + “)”. You retrieve a list of parameter and add to result in next format: parameter.getName() +”:” + parameter.getClassifier().getName(). Note one moment – you should check if kind of parameter is not equal ParameterDirectionKind.RETURN. We can add here return parameters.
9. Create method body, simply add “{};”

- **Properties generating**
  There are three kind of properties in C# - getter and setter both, setter and getter. Actually you should get names of stereotypes for configuration parameters “property”, “property.get” and “property.set”. If operation contain at least one such stereotype it should be generated as a property.
  Actually it very similar to operation but it has no parameters and return type. Also it body depends on concrete stereotype. For “property” it should be:
  Generate body like
  {
  set{};
  get{};
  }

- **Delegates generating**
  Possible method features:
  “public delegate void test(String s)”
  Delegate is represented in Model as Classifier which list contains only one feature – Operation. Generating code for it will be combination of generating code for Classifier and for Operation. Visibility modifier gotten from Classifier, “delegate” it is stereotype, and rest is actually Operations with parameters

Imports generating
Each time when it is found new classifier in head classifier, attribute method etc. It package (namespaces) name should be added to list of imports. For example you get as parameter some class with name Test. Now you should get it namespace name and add it to list of imports like this –
getImports().add(getPackage(parameter.getNamespace())); In such case it will be generated proper list of imports(using).
The same situation is for inheritance. If you class extends from class from other package. You need add this package to imports list.

Mistake generating
Use logger to notify user with serious problem with code. Logging should be used for example for empty parameters, attributes names, using incorrect combination of languages key words (abstract and final in one time), multiple inheritances etc. Please

do not use logger for notifying about style, formatting problems. When we log something it should be serious code mistake and also it should be easy check. For example when we retrieve Classifier generalization we use list classExtends. For checking for multiple inheritances you need just check if list contain more then one parameter. Logging such mistakes should be in process of creating code.

General recommendation.

1. Note that it is possible add some new private methods in Generator instances. You may add new methods and provide the other combination of existing. It is up to you.
2. Code examples are given only for better understanding of ideas. They are not required to be implemented in such way. It also may be changed.

1.4 Component Class Overview

Class CodeGenerator

This is main class of component, which represent all its functionality. This class contains overloaded method generateCode() which provides ability to generate and write code to special location. The other configuration it provides through overloaded constructor. Class contains two configurable parameters – emptyLocationDirectory and createEmptyPackage.
emptyLocationDirectory – get user ability to choose if location directory should be cleaned before writing there files.
createEmptyPackage – get user ability to choose if should be written packages which do not contain any Classifiers.

Class is thread safety because it is immutable.

Class CodeWriter

This class is responsible for working with directories and files. Class provides creating necessary directories which structure reflect Java packages and C# namespaces. It provides also ability to write “*.java” and “*.cs” files. Name of file is the same as name of according classifier.

Class is thread safety because it is immutable.

Class GeneratorFactory

This class represents realization of Factory pattern for this component. It returns different instances of Generator depend on language parameter. It may return JavaGenerator or CSharpGenerator.

Class is thread safety because it is immutable.

Interface Generator

This interface represent all logic which connecting with model components. It may get name of package, create code of Classifier or check if Classifier is correct for this component (Class, Interface or Enumeration).

Classes which implement this interface may be mutable.

Class AbstractGenerator

This abstract class provides realization of methods which are common for C# and Java realization of Generator. Class implements interface Generator.

This class is not thread safe because it is mutable.

Class CSharpGenerator
This class represents Generator instance for C# language. It contains only one public method createClassifier() and many private methods which encapsulate code generation for some part of classifier.

This class is not thread safe because it is mutable.

Class JavaGenerator
This class represents Generator instance for Java language. It contains only one public method createClassifier() and many private methods which encapsulate code generation for some part of classifier.

This class is not thread safe because it is mutable.

1.5 Component Exception Definitions

Class CodeGenerationException
This exception encapsulate or possible exception which may be thrown by component. It may be thrown when problems with configuration and input/output procedures appear. It has two children to specialize different case of its appearing.

Class is thread safety because it is immutable.

Class GeneratorConfigurationException
This exception encapsulates all configuration problems of component. It extends from CodeGenerationException.

Class is thread safety because it is immutable.

Class GeneratorFileSystemException
This exception encapsulates all problems which may appear with file system and input/output process. It extends from CodeGenerationException.

It is possible two error situation:
1. Impossible write file (CodeWriter throw IOException)
2. Impossible create or delete directory (methods createDirectory() and deleteDirectory() returns false)

Class is thread safety because it is immutable.

It used IllegalArgumentException in many methods, when parameters of method is null, or when String is empty. It used IOException when appear problem until creation folder or writing files. ConfigManagerException may be thrown when problems with configuration appear.

1.6 Thread Safety

This component is thread safety. All its classes and attributes are immutable. But it use Configuration Manager and Logging Wrapper which are not thread safety. Also it used model component which are not thread safety too. In this case it is impossible to guarantee full thread safety.

Thread safety is not required for this component. UML Tool provides own external thread safety model.

2. Environment Requirements

2.1 Environment

Java 1.5

2.2 TopCoder Software Components

UML Model - Model Management 1.0

This component represents Package interface and it default realization.

UML Model - Core Requirements 1.0

This component represents many interfaces (Namespace, ModelElement,Classifier, Operation etc.) and their default realization.

UML Model - Data Types 1.0
This component represents data type’s interfaces and enumerations.

UML Model - Core Extension Mechanisms 1.0
This component represents Stereotype and TaggedValue interfaces and their default realization.

UML Model - Core Classifiers 1.0

This component represents supported Classifiers (Class, Interface and Enumeration) instances and their default realization.

Logging Wrapper 1.2
Used for logging mistakes which is in generated code

Configuration Manager 2.1.5
Used for configuration of component

Base Exception 1.0
Used as parent for all component exceptions.

Typesafe Enum 1.0
Used by Loggin Wrapper..

2.3 Third Party Components

None

3. Installation and Configuration

3.1 Package Name

com.topcoder.uml.stubclassgenerator

3.2 Configuration Parameters

Parameter	Description	Values
javaLanguage	Format of Java language name	Java
C#Language	Format of C# language name	C#
documentation	Represent name of documentation node.	documentation
tagsDocumentation	Represent part of name of tags documentation nodes.	documentation#
new	C# key word, value is corresponding stereotype name	new
readonly	C# key word, value is corresponding stereotype name	readonly
virtual	C# key word, value is corresponding stereotype name	virtual
override	Name of stereotype which value corresponds C# key word	override
sealed	C# key word, value is corresponding stereotype name	sealed
extern	C# key word, value is corresponding stereotype name	extern
delegate	C# key word, value is corresponding stereotype name	delegate
explicit	C# key word, value is corresponding stereotype name	explicit

implicit	C# key word, value is corresponding stereotype name	implicit
struct	C# key word, value is corresponding stereotype name	struct
property_get	This represents C# property which contain only getter, value- corresponding stereotype name	property_get
property_set	This represents C# property which contain only setter, value- corresponding stereotype name	property_set
property	This represents C# property which contain getter and setter, value - corresponding stereotype name	property
destructor	This represents destructor for C#, value corresponding stereotype name	destructor
protected internal	Special unsupported access modifier of C#	protected internal

3.3 Dependencies Configuration

Configure please Logging Wrapper 1.2 and Configuration Manager 2.1.5

4. Usage Notes

4.1 Required steps to test the component

- Extract the component distribution.
- Follow Dependencies Configuration.
- Execute 'ant test' within the directory that the distribution was extracted to.

4.2 Required steps to use the component

There are no some special requirements for running this component

4.3 Demo

Actually component does not provide some special scenario. It used only for one target – generate code with already prepared data.

```
// Create a new instance of CodeGenerator from a custom name space
// + emptyLocationDirectory == true: location will be cleared
// + createEmptyPackage == true: empty packages will be created
CodeGenerator generator = new CodeGenerator(true, true, NAMESPACE);

// Generate code for some classifier
// the pacakge of class1 is com.topcoder.uml.stubclassgenerator
// its name is Foo
// therefore it will be written to:
// location/com/topcoder/uml/stubclassgenerator/Foo.java
generator.generateCode("Java", location, class1);
```

```
// Generate code for a list of classifiers in C#
List<Classifier> classList = new ArrayList<Classifier>();
classList.add(class1);
classList.add(class2);
generator.generateCodeForClassifiers("C#", location, classList);

// Generate code for a whole package
generator.generateCode("Java", location, package1);

// Generate code for list of packages
List<Package> packageList = new ArrayList<Package>();
packageList.add(package1);
packageList.add(package2);
generator.generateCodeForPackages("Java", location, packageList);

// Generate combined code for package and for classifier which is not
// part of given package.
generator.generateCode("Java", location, class2);
generator.generateCode("Java", location, package1);
```

5. Future Enhancements

None